FAST: A FLEXIBLE ARCHITECTURE FOR SIMULATION AND TESTING OF MULTIPROCESSOR AND CMP SYSTEMS

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING

AND THE COMMITTEE ON GRADUATE STUDIES

OF STANFORD UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

By

John D. Davis

December 2006

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

_____

(Oyekunle Olukotun) Principal Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

_____

(Christoforos Kozyrakis)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

_____

(Stephen E. Richardson)

Approved for the University Committee on Graduate Studies.

Abstract

Empowered by the virtually endless supply of transistors provided by today's leading-edge silicon process technology, computer architects seem to have an infinite number of design possibilities at their disposal. At the same time, the diminishing returns of instruction level parallelism (ILP) have forced designers to utilize designs embracing thread level parallelism (TLP). As a result, new CMP designs try to connect multiple processors using novel memory system designs instead of making larger and more complex uniprocessors. Unfortunately, these CMPs cannot be easily explored and evaluated using existing software-only simulations tools.

Historically, software simulation has been the vehicle of choice for studying computer architecture because of its flexibility and low cost. Regrettably, designers of software simulators must choose between building simulators that provide either high performance or detailed hardware emulation. Building actual hardware, in contrast, provides high performance *and* accurate results, but lacks the flexibility to explore multiple designs and is very expensive. These tradeoffs have impeded our ability to thoroughly explore and evaluate new computer architectures.

This dissertation describes the architecture, implementation and evaluation of a simple prototype. This proof of concept is implemented on a new hybrid hardware prototyping platform enabled by integrating a variety of hardware components on a printed circuit board (PCB) to implement Chip Multiprocessor (CMP) or Multiprocessor (MP) systems. The Flexible Architecture for Simulation and Testing (FAST) combines the flexibility of software simulation with the accuracy and speed of a hardware implementation enabling computer architects to implement new multithreaded, multiprocessor, or CMP architectures for in-depth evaluation and software development. This is accomplished by combining dedicated microprocessor chips and SRAMs with Field Programmable Gate Arrays (FPGAs), enabling FAST to operate at hardware speeds, yet still have the ability to emulate the latency and bandwidth characteristics of a modern CMP architecture. FAST provides the foundation for future TLP-focused computer architecture research and software development that is currently not possible or practical using software-only or hardware-only solutions.

Acknowledgements

There are many people who have supported me throughout my Ph.D. career at Stanford University. Without the Stanford community, I know that my research would have been impossible to successfully complete. First, I would like to thank my adviser, Kunle Olukotun, for all of the support he has provided over the years. Kunle provided an opportunity for me to exceed my own expectations. I was able to design, build, and run applications on a large hardware prototype. This project touched on all aspects of computer architecture and produced a functional system.

I would also like to thank my reading committee, Kunle, Steve, and Christos, for their support and feedback. Steve Richardson has been a mentor, colleague, and friend. Steve's involvement in the software development provided the motivation for me to continue the project after delivering the assembled PCB. Without Steve, my career at Stanford would have ended without experiencing the success of building a fully functional PCB. I cannot express how much I appreciate Steve's honesty, technical critiques, and encouragement.

There have been a few other key individuals that have made this project a success by either providing support or consulting. The initial PCB design was based on the Hydra Verilog produced by Lance Hammond. Lance also provided numerous hours of critical evaluation and support. Wade Gupta developed the PCB buzz out plan and provided some foundational work for the PCB layout. Alan Swithenbank provided valuable consulting support for the PCB design and manufacturing. Alan was also the Cadence representative. Alan's interaction with the project reduced the time required to resolve the numerous CAD tool issues.

Building hardware is a very difficult coordination effort. Darlene shielded me from the bureaucracy and daunting paperwork details. In general, Darlene also provided translation skills to help me navigate the Stanford maze. My project would not be a success without Darlene's coordination and administrative support as well as attention to detail.

I have had the pleasure of working with several other talented Master's and Undergraduate students. I would like to thank all those that contributed something to the FAST project: Charis Charitsis, Wade Gupta, Ben Hubbert, Lark-Hoon Leem, Amarachi Okorie, Bob Bell, Brian Mariner, Bryan Golden, Garrett Smith, Bert Shen, John Vinyard, Joshua Baylor, Daxia Ge, Michael Yu and Mark Lim.

The members of my research group also provided support along my difficult journey. In particular Lance, Mike, Mano, Valeria, Hassan, Brian, Nju, and Brad provided words of encouragement and/or sympathy.

The computer support staff at Stanford, Charlie Orgish, Joe Little, Jason Conroy, and Kevin Colton, provided the infrastructure and knowledge to keep all the machines running smoothly under my watch.

I would also like to thank Laura Roman from the Stanford Writing Center for proofreading my dissertation.

The most valuable part of my Stanford experience has been the close friends that have helped me in classes by pushing my scholastic breadth, working with me on projects, and listening to my talks. In particular, Andrew, Amy, David, Cher, and Eric have supported me throughout my Ph.D. journey.

My graduate education would not have been possible without the financial support from my NSF fellowship and my RA funded by DARPA.

I would also like to acknowledge the incredible social network that I developed and maintained while at Stanford. The Black Graduate Student Association (BGSA) has been my family away from family. This small community has motivated me to make Stanford better for those that follow me. The BGSA's social activities have provided balance to my Stanford experience. Similarly, my involvement in the Graduate Student Council (GSC) has exposed me to the social and cultural wealth of the graduate population/community. The GSC also served as a vehicle for me to interact with the administration and dramatically change the university structure.

My family also includes friends outside of Stanford. They have provided continual support and always accepted the "two-year" plan for finishing my Ph.D. no matter how many times the plan started over. Tony and Patty have always encouraged me along the

way. Ed Lazowska has also encouraged me along the way and kept me connected to UW. I know that I would have not considered continuing my education without Ed's support and guidance.

I would like to give special thanks to my parents, Anne and David. They set an example of excellence and overcame greater challenges and adversity to achieve their goals. Furthermore, I would not be here without them. My sisters, Joanne and Eloise, brother Byron, and extended family have also been bedrocks of support and encouragement.

Riff and Ruann have become part of my family. I have been blessed to be a part of their lives and they have welcomed me into their home and incorporated me into their family. They have been the parents away from home and the corresponding support system. They have been great role models with the ability to spark spirited conversations.

Finally, Stanford has also introduced me, indirectly, to my future family: Jessica, her sister Alison, and her parents Michael and Margaret. Jessica supported me in my darkest hours with her love and words of encouragement. I finish my time at Stanford and continue the exciting journey with Jessica.

Table of Contents

Appendices

List of Tables

List of Figures

Chapter 1

Introduction

In the recent era, computer architects have leveraged increasing transistor density on microprocessor chips to implement single, large processors that exploit instruction level parallelism (ILP). However, continued performance gains from ILP are becoming increasingly difficult to achieve due to limited parallelism among instructions in typical applications [97]. Likewise, the problems associated with designing ever-larger and more complex monolithic processor cores are becoming increasingly significant. The increased design complexity also adds new design problems that compound the normal first-order design problems that accompany larger designs, i.e., increased bug rates and longer design and verification. These design problems are exacerbated by including new first-order effects like wire delay in the design requirements [70]. This reality has spurred great interest in exploiting thread-level parallelism (TLP) among independent threads of instructions to continue historical microprocessor performance improvement trends.

Multithreaded microprocessor architectures such as chip multiprocessors (CMPs) are now ubiquitous in industry and research. These multithreaded architectures effectively integrate symmetric multiprocessor (SMP) systems onto a single chip. These architectures improve the throughput of multi-process applications such as commercial server or network processing workloads or of multiple independent programs aggregated on a single system that integrates multiple homogeneous or heterogeneous processors onto a single chip [30, 63]. The high performance of CMP architectures is achieved by integrating various forms of multithreading with novel memory systems utilizing large, on-chip caches that can be shared or not shared. Examples of these architectures are the UltraSPARC T1 [61] and Montecito processors [71].

Analytical performance models cannot predict complex memory orderings and, as a result, cannot quantify system performance. Likewise, as on-chip memory performance begins to dominate system performance, software simulators must maintain and track memory interactions to predict performance. Also, the majority of software simulators

are single threaded. The combination of simulating large on-chip caches and simulating multiple processors or threads using a single processor dramatically slows down the effective speed of the software simulators. As a result, these traditional methods of performance prediction and software development are impractical for these architectures, because of the high level of integration of these systems.

This dissertation explores a new hybrid methodology for rapidly investigating the CMP architecture space by combining dedicated processors with novel memory system designs. The system that we propose enables highly detailed, rapid prototyping, either emulation or implementation, of full systems. This system addresses the drawbacks of analytic modeling, software simulation, and hardware prototyping.

1.1 Architecture, performance, and technology scaling

The CMP is the dominant processor architecture across many domains including embedded, desktop, laptop, and server processors. Instruction level parallelism and frequency scaling are no longer viable options for improving processor performance [98]. Furthermore, silicon transistor scaling has reduced the size of even the most complex processors, facilitating two or more complex cores on a single die with minimal integration effort [56]. As a result, 2005 was the first year where all major chip manufacturers introduced CMPs, and current processor roadmaps forecast nothing but CMPs [3, 57, 58, 77, 85, 92].

The current International Technology Roadmap for Semiconductors (ITRS) continues to forecast silicon process scaling through 22 nm [58]. This forecast will provide computer architects with plenty of transistors to develop the CMPs on processor roadmaps [3, 57, 58, 77, 85, 92]. Moving forward into 65 nm silicon process technology, transistors are no longer increasing in speed, and wire delay dominates critical paths. These trends reinforce local, short wires and small-area clock domains that are particularly suited for CMP architectures, which inherently exploit TLP. These restrictions inhibit the design of structures for ILP, which use high clock frequency, large structures, and several pipeline stages to issue multiple instructions per cycle. Furthermore, very few applications exhibit the massive levels of ILP that these (previously monolithic) processors could exploit.

Finally, Sun Microsystems has taken the most aggressive steps in exploiting TLP with its Niagara processor, which includes 8 processor cores with 4 hardware threads per core for a total of 32 threads or virtual processors [61]. The Niagara processor also integrates other system-on-a-chip (SOC) components, like on-chip memory controllers and cryptography functional units. It is only a matter of time until other industry players follow this lead.

Current general-purpose and commodity processors integrate hundreds of millions of transistors on a single chip, at multiple gigahertz clock frequencies. Intel's Montecito chip is a two processor CMP with over 1.7 billion transistors, although most of these transistors are used for two third level caches [71]. Future silicon generations will be able to exploit these high transistor counts by implementing special processing units rather than massive on-chip memories. Regardless of the integration of special processing units, future generations of CMPs will still have many processor cores and several megabytes of aggregate on-chip memory.

Higher transistor counts, short wires, and a slower rate of increase in clock frequency combine to reinforce the trend toward CMP architectures. The higher transistor count can be exploited in CMPs for more processor cores, on-chip cache, or other SOC units such as cryptography units, memory controllers, or on-chip network controllers. Davis et al. focused on the core and cache trade-off for commercial server applications [30]. This design-space study demonstrated that simple cores enabled with fine-grain multithreading using moderate area for caches (25-40% of the CMP area) garnered maximum throughput for this application space when compared to CMPs using more complex superscalar in-order processor cores. Furthermore, when extrapolating these results with an area model for out-of-order processor core based CMPs, simple scalar in-order fine-grain multithreaded cores continue to outperform these very complicated ILP-focused out-of-order cores with respect to thread throughput and throughput oriented commercial server applications [30].

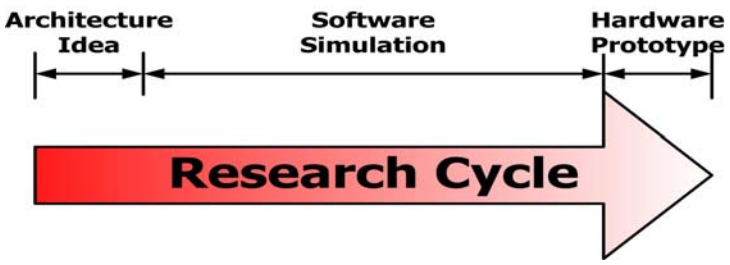


Figure 1.1. Previous computer architecture research cycle.

As a result of the advances in transistor scaling technology and the adoption of CMP architectures, the architecture simulation and validation problem is growing in complexity. In the past, the research cycle involved the following steps: (1) formulating an idea, (2) using software simulation to do further investigation of the idea, and (3) finally, building a hardware prototype to validate that specific idea. Figure 1.1 illustrates these steps with an arrow transitioning from left to right, dark to white, reflecting the increased clarity that each step brings to the architecture idea.

1.2 Additional terminology

Before diving into further details, it is beneficial to define some of the more frequently used terminology. Field programmable Gate Arrays (*FPGAs*) are digital chips that can be programmed to provide various digital interfaces and functionality. *Morphware* refers to the source code used to describe hardware functionality programmed in the FPGAs. *SRAMs* are large, fast storage elements used for implementing memory structures. A printed circuit board (PCB) is a substrate that provides the interconnectivity for all the components in the design, both power and ground as well as chip interconnectivity, i.e., a connection on pin A of chip 1 to pin B on chip 2. The dedicated processors refer to the central processing unit (CPU) used for integer computation, and the floating-point unit (FPU) used for floating-point computation. We combine several components, the dedicated processor, FPGAs, and SRAMs, on the PCB to build an equivalent modern processor.

Discussion of hardware systems can lead to some confusion with regard to the fidelity and mapping of the hardware or software tools. For the purposes of this dissertation, the host machine (for software simulators) or host hardware (the PCB and morphware) are

the platforms that support the execution of the research target system, in this case a simulator, emulator, or implementation. Generally and historically, the difference between simulators and emulators are a few shades of gray. Most people use these terms interchangeably. It is useful to differentiate simulation, emulation, and implementation.

In the case of *simulation*, a software model tries to approximate a target system. This approximation can vary from a very coarse-grain resemblance to a very detailed fine-grain model. In some cases, the operating system and applications cannot differentiate between highly detailed software simulators and real hardware. Today, software programs are written to model and investigate new systems. The simulation tracks the state transitions of key characteristics of the system. In the case of computer systems, this may include memory state changes, instruction interaction, and/or program behavior. Regardless of the level of detail, certain simplifications and approximations must be made in the simulation framework. Simulation is an extension of a mathematical model that uses initial conditions to predict system behavior. For computer systems, simulation attempts to predict program behavior. However, the simplifying assumptions and approximations determine the fidelity and validity of the resulting simulation predictions. Software simulation provides far more transparency than hardware systems, down to the level of detail that is modeled in software. However, these simulators convert a parallel machine or process into a sequential process, possibly abstracting away timing and other timing related interactions.

Emulation provides the highest level of fidelity when compared to the target system. An emulator tries to imitate the target system and accepts the same data and executes the same programs achieving the same results as the target system. Emulators can be software-based or hardware-based. Both try to maintain the state of the target machine, whereas simulation is generally only concerned with the program behavior. Emulation requires extreme detail of both documented and undocumented features, down to individual clock cycles. In some cases, the emulator must deviate from the original specification and include bugs. Traditionally, emulation (in software or hardware systems) has been associated with gate-level simulation of designs. Finally, in the case of hardware emulation, the entire design executes in parallel, whereas software emulation and simulation typically executes the system in a well-defined sequential manner.

Finally, an *implementation* or *prototype* of a computer system is an instantiation of the real system. In general, researchers build hardware prototypes that implement and validate their design. These prototypes generally compromise on raw system performance, high clock frequencies, but scale all of the on and off chip latencies in order to validate the design. Hardware implementations make certain trade-offs with respect to the initial specification and thus, may not be as cycle accurate for every signal with respect to old designs or new designs. An implementation provides high fidelity, but it may differ from the true specification. Simulators and emulators can be built from software, while hardware can be used to build emulators and implementations, but with varying degrees of flexibility. Emulation is differentiated from an implementation or a prototype by the system transparency. Emulation generally provides greater system transparency for debugging purposes and requires more effort to model every detail. Prototypes generally have less transparency and real computer systems have far less transparency for system debugging. External observation, using a logic analyzer, and limited performance counters are the common methods used for prototype computer system debugging. A system that can emulate or implement a variety of computer architectures is the goal of this work.

We introduce a hybrid system that can either emulate or implement target systems. We define a hybrid prototyping platform as a combination of fixed-function and programmable hardware and software that yield a system with the benefits of a software simulator and hardware prototype. This hybrid system emulates computer systems when it must approximate particular components, i.e., using a simple processor instead of a more complex processor. Likewise, it can fully implement target systems, but not at the target frequencies. In the case of implementations, the hybrid system scales all of the relevant latencies to preserve the ratios, thereby truly implementing the target system. Implementation and emulation are used interchangeably in this dissertation; the target fidelity determines which term is appropriate. Thus, our system has the flexibility to implement many different hardware prototyping systems on the same PCB platform by changing the PCB using higher-level software.

1.3 Computer systems development

There are three main methods used for developing and validating new computer systems. These methods range from coarse, back-of-the-envelope, analytical evaluation to building a hardware prototype. Given the complexity of modern and future systems, it is beneficial to describe these methods and enumerate the benefits and drawbacks of each method. Analytic modeling, software simulation, and hardware prototyping enable researchers to study and understand computer systems behavior, particularly hardware and software interaction, at various levels of detail. Regardless of the method used to understand computer systems, new computer systems development is application driven and tries to improve the performance of applications on a particular computer system.

1.3.1 Analytic modeling

Analytic models are mathematical models of the system that can be used for the initial validation of computer architecture ideas. In general, these models are high level. These models can accurately predict performance of simple uniprocessors with blocking caches. For these simple systems, a decoupled processor and memory model can be used. These models can be very basic back-of-the-envelope calculations or very complicated queuing theory models with many variables. The processor and system performance can be derived from a simple memory performance model. Regardless of the complexity of the model, various simplifying assumptions must be made to use these analytic models. Unfortunately, modern computer systems are too complex for these high-level abstractions, which produce invalid analytical models. The number of simplifying approximations increases with hardware and software complexity. Furthermore, software development is not hardware agnostic, forcing developers and compilers to work together to improve program performance. This hardware/software interaction cannot be predicted with analytical models. As a result, the applicability of analytic models is very limited and not adequate for modern and future processor architectures. CMP systems with shared resources, large on-chip caches, and multiple outstanding and interacting memory reference make these mathematical models even less relevant because of the number of first and second-order effects that cannot be accounted for with these approximating models. Furthermore, these complex CMP systems further confound the analytic models, which cannot quantify the impact of software on these complex

computer systems or enable software development or tuning. At best, these models provide gross analysis and potential trends that rely on explicit and implicit approximations and assumptions.

1.3.2 Software simulation

Software simulation has become the *de facto* standard for computer architecture research facilitated by a trio of software simulators: SimOS [81], Simics [68], and SimpleScalar [12]. These simulators provide many benefits. The computer architecture community reuses these software simulators, using them as the foundation for new research. The flexibility of these software systems makes them very attractive because software simulators are modular and easy to extend to encompass additional functionality. Software simulators provide maximum observability or transparency; all internal components of the software-simulated system are visible, unlike hardware systems. Software simulators are often deterministic, making them easier to debug because the results and behavior are reproducible. As the *de facto* standard, software simulators have a large user base and development time is greatly reduced because of the additional software modules provided by the large developer/user community. Finally, these particular software simulators have the right price for academia: *free*. These factors combine to create a foundation for all current and future research.

In terms of the research cycle shown in Figure 1.1, the software simulators are essential for initial research, i.e., limit studies or rapid system prototyping, using fast, inaccurate software simulators. As the software simulators mature, they become more complex, modeling the system with more detail, which makes the simulator slower, but more accurate. Using execution-driven simulation or binary translation, full system simulation can provide highly detailed and accurate results. However, these software-based simulators allow one to evaluate only small benchmarks or fragments of larger benchmarks using instruction-level simulation, but are too slow to simulate entire applications within a reasonable time. Complicated CMP and multiprocessor designs exacerbate this problem by requiring that many processors be simulated simultaneously with complicated memory systems. Multiple simulated processors linearly slow down sequential software simulators, while the complex memory systems require memory

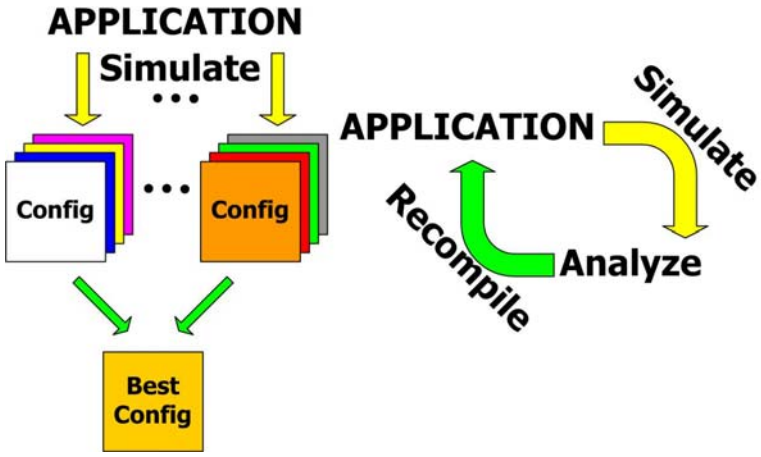
address tracking to correctly simulate memory interactions [22, 33, 39, 48, 66, 76]. As a result, the complexity of even the most basic multithreaded architectures presently limits instruction-level simulation to an effective "clock rate" of about 0.05 MHz; most simulators, especially RTL ones, achieve much less [5, 31, 60]. Simulation speed therefore limits the scope and effectiveness of research that can be performed in reasonable amounts of time.

Furthermore, software simulators are inaccurate as shown in the Flash hardware, software simulator comparison [40] and as a result, suffer from credibility problems. Thus, software simulators can be used to indicate trends, but not necessarily absolute results without significant validation effort. Moreover, academia lacks the resources to fully validate software simulators for machines that exist and software simulator validation is even more challenging for completely new computer systems or programming paradigms. Finally, the combination of all of these factors means we cannot use software simulators to facilitate software development for new computer systems because of the inherent speed and fidelity trade-offs.

Recent computer architecture research has focused on ILP or instruction throughput on monolithic processors. In these studies, the applications did not require development or tuning because the microprocessor designs tried to simultaneously execute as many instructions as possible. For this type of research, a slow, sequential (single-threaded) simulator could be run on several machines, with each machine having its own particular simulated machine configuration as shown in Figure 1.2 A. After all the simulation runs complete, the best configuration can be selected from the collection of configurations run for that particular study.

Unfortunately, only industry can afford to explore limited design trade-offs by using large server farms (on the order of 10,000 or more processors) to evaluate several design elements simultaneously [30]. However, if we focus on designs that require software development or application tuning, we transition from a potentially parallel simulation system to a sequential simulation process that requires application tuning and evaluation before proceeding to the next iteration, as shown in Figure 1.2 B. This sequential simulation process is required because recompilation and application tuning is based on

simulated results. Thus, the simulation process becomes sequential and can no longer benefit from multiple simulator instantiations running simultaneously. A single simulation that takes days, weeks, or months now forces software development to wait for hardware to be available, further limiting the scope of computer systems research facilitated solely using software simulators.



(A) (B)

Figure 1.2. (A) Parallel simulation of multiple configurations vs. (B) sequential simulation required by application development and/or tuning.

1.3.3 Hardware prototyping

Relying solely on software simulation to evaluate performance and develop applications limits the scope and effectiveness of computer architecture research. Simulators are fast enough to make basic architectural decisions based on the performance of small sections of a limited number of benchmarks, but they are too slow for running a wide variety of complete benchmarks or for analyzing entire applications in detail to determine how to improve performance. Historically, researchers have built hardware prototyping systems to validate their designs and enable software development and highly detailed investigation.

The main advantage of the hardware prototype is its speed when compared to software simulators. Implementing an architecture flushes out all of the details that are not described in the architecture. The main disadvantages of building hardware prototypes are the cost, time required, and the inflexibility. With each successive silicon process generation, building hardware prototypes is increasing dramatically in cost. Just creating

the mask set for a 90 nm process has surpassed $1 million, pushing processor hardware prototypes outside the financial viability of most research projects.

Time-to-market of research ideas is also significantly impacted if a hardware prototype is required. Building a working hardware prototype is very time intensive and includes making the hardware and developing all the software to make it operational. Finally, traditional hardware prototypes were built for one particular idea and cannot be extended to other ideas. The benefits of building hardware prototypes do not outweigh the drawbacks because of the inability to amortize the cost or leverage some of the hardware and software infrastructure across multiple systems. As a result, Figure 1.1 has evolved into Figure 1.3. The expense of hardware prototyping and its inflexibility force the research cycle to solely rely on software simulation. Moving forward, software simulators are becoming more and more inadequate to provide useful information about large computer systems in a reasonable amount of time, making software development impossible.

Those that rely entirely on software simulation for computer systems studies as shown in Figure 1.3, lack the clarity and validation and resulting credibility that come from building a real system. To break out of this current research cycle, we need a solution that bridges the gap between slow software simulators and expensive one-off hardware prototypes.

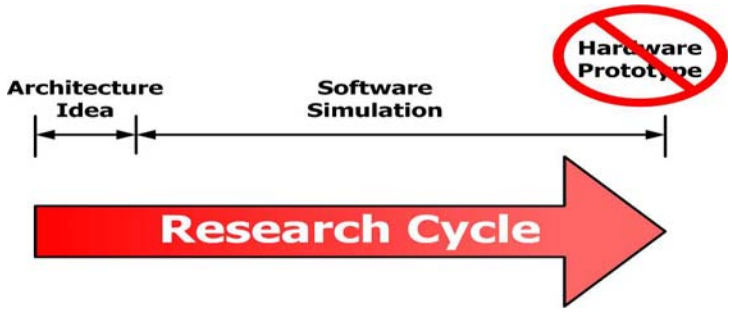


Figure 1.3. Current computer architecture research cycle.

1.4 Enabling computer systems development

It is clear that new solutions are required to do computer systems research. Many efforts have been made to overcome software simulator speed limitations using hardware emulation [6, 10, 80, 115]. Historically, hardware emulation platforms use arrays of Field Programmable Gate Arrays (FPGAs) to generate rapid prototyping systems that can

emulate entire designs at an RTL level [9, 32, 33, 96]. Unfortunately, efforts to compile multiprocessor designs to these systems have been limited by poor FPGA logic utilization, limited interconnectivity in the FPGA arrays, and poor word-size data manipulation by bit-width FPGA logic units [96]. However, improving transistor densities continue to improve FPGA applicability. Furthermore, the integration of hard processor cores or software-defined processor cores inside the FPGAs has extended FPGA prototyping system capabilities by providing optimized compute infrastructure that can run real applications and operating systems.

1.4.1 Flexible hardware

FPGAs are an enabling technology for detailed emulation or implementation of computer systems. Industry already provides expensive examples of such systems [17, 18, 32, 73, 93]. Unfortunately, these systems have drawbacks similar to normal hardware prototyping, but they do solve the flexibility problem. These systems are expensive and require full designs to be partitioned and mapped to the FPGA array. Even with the recent introduction of hard and software-defined processor cores within the FPGA, the main limitations still exist: limited on-chip memory and the lack of word-size optimized datapaths built from combinational logic. Even with two embedded hard processors, the FPGAs still suffer from limited on-FPGA memory (BRAM) and awkward processor interface required if the processor cache system is redefined. As silicon technology improves, another alternative is software-defined processors cores or soft cores. Researchers are able to leverage existing software intellectual property and modify the soft cores and surrounding subsystems to fit their research needs. Soft cores become more and more compelling because of the high number of soft cores that fit on a FPGA and the availability of operating systems and other software tools. Thus, technology advances have enabled FPGAs to become the main building blocks of future high performance flexible prototyping systems. Thus, these prototyping systems are limited by the FPGA building blocks in terms of memory and FPGA interconnect. These systems also tend to be application-specific integrated circuit (ASIC) or single chip focused and not system focused, which further limits their applicability to computer systems research. Furthermore, once the design is validated, mapped to the array and working, it still operates at speeds close to that of software simulators, about 50 kilohertz. In their

current form, these generic arrays only solve one out of three problems associated with hardware prototyping. Hardware cost and software development are still limited using this solution. Likewise, these generic FPGA arrays have very limited application for large-scale computer systems exploration space because they are ASIC focused. Finally, unlike software simulators, generic FPGA arrays do not leverage previous work, either hardware or software, which would reduce development time.

1.4.2 Hybrid hardware prototyping platform

We are at the point in the area of multithreaded microprocessor architectures where further progress will require the development of a hardware prototype to enable software development and in depth analysis. This prototype should support more than two parallel threads and novel memory systems like those for thread level speculation (TLS) and transactional consistency and coherence (TCC). Currently, no commercial microprocessor has these multithreading capabilities and this prevents the serious OS, compiler, and application development that is required to take full advantage of multiple threads and other novel memory systems. Without the resulting optimized software, it will be difficult to understand the true benefits of these techniques or make the appropriate hardware/software design tradeoffs to achieve the best performance [28]. However, the problem with building a microprocessor is that it requires a large amount of time and money. The immense task of chip design verification before tape out, in particular, can make microprocessor design a difficult undertaking in an academic environment. This is the primary reason why all multithreading and novel CMP memory system research has thus far relied on software simulation.

Fortunately, by focusing on the multithreading and novel memory system capabilities of future microprocessor architectures, instead of ILP extraction mechanisms, we do not need to make modifications to the core CPU pipeline of our base processors. We can build a hardware prototyping platform around relatively simple processors. Therefore, it is possible to build a flexible research prototype around those existing processors without doing any new VLSI design. We can also leverage the flexibility of FPGAs to provide the digital interfaces and controllers along with specific latencies between the memory subsystem and the processors. Finally, because FPGAs lack the memory density to create

large on-chip memories, we can leverage off-chip static random access memories (SRAMs) to provide the fast access and large capacity that is not available in current or future generation FPGAs. By combining off-the-shelf dedicated processors, FPGAs, and SRAMs, we can build a flexible platform for CMP and multiprocessor research at hardware speeds. Thus, software development is possible because of the hardware capabilities and speed. However, the hardware is only half of the solution, base software is required to really make this system useful, modular and expandable, similar to software simulators. The combined hardware and software solution is called FAST, or Flexible Architecture for Simulation and Testing. It provides the speed of actual hardware with the flexibility of a software simulator for emulating or implementing CMP or multiprocessor designs.

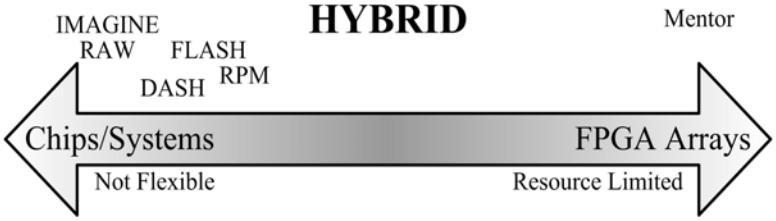


Figure 1.4. Computer systems hardware implementation spectrum.

We thus bridge the hardware prototype and software simulator gap by combining programmable hardware (FPGAs), dedicated hardware, and software, making a heterogeneous hybrid solution. When surveying the hardware landscape, hardware prototypes exist at one end of the spectrum and FPGA arrays exist at the other end, as shown in Figure 1.4. We list some inflexible prototyping systems on the left and an FPGA array solution on the right, all of which were available at the start of this project, circa 2002. Some hardware prototypes did include FPGAs or other programmable components, which enable system observation or transparency or provided limited configurability for narrow investigation around the original design point [5, 40, 60, 66, 96].

A hybrid solution combines the benefits of software simulators with the benefits of hardware prototyping by leveraging the strengths of the base hardware components and the modularity of software. Table 1.1 provides a (subjective) qualitative comparison of software simulators, hardware prototyping systems, and an *idealized* hybrid solution.

Table 1.1 explicitly lists the benefits and drawbacks described in Section 1.3. We compare software and hardware with an idealized hybrid solution to normalize these mature systems to a new solution. It should be noted that because the hybrid solutions contain hardware, the hybrid solution would not have a zero cost as do most software simulators used in academia. In general, academic hardware projects benefit from the generous donations of industry to make them possible. This is the only reason why hybrid system costs are an "A." Likewise, in the ideal case, development time would be on par with software development. This would be achieved by hiding the timing challenges and using high-level languages or system generators to build the target prototype.

Table 1.1. Software, hardware, *idealized* hybrid systems trade-offs.

Feature	**Software**	**Hardware**	**Hybrid**
Reuse	A+	F	A+
Flexibility	A+	F	A+
Transparency	A+	D	A+
Reproducibility	A+	C	A+
Community	A	D	A
Development Time	A	F	A
Cost	A+	F	A
Credibility	F	A	A
Performance	F	A	A
Software Development	F	A	A

Both academia and industry need tools to enable further investigation of CMPs and multithreaded architectures. Because of its flexibility, hybrid solutions promise to be the next method to provide credible validation that can be reused across multiple architectures. This not only brings hardware emulation or implementation back into the research cycle, but it has the potential to incorporate hardware into the research cycle far earlier than building a hardware prototype, as shown in Figure 1.5 when comparing it to Figure 1.1.

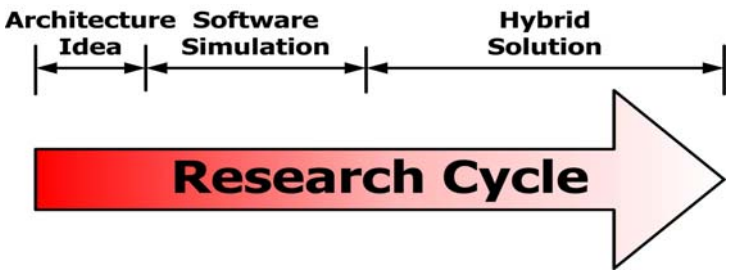


Figure 1.5. Research cycle with the hybrid solution.

Even in the ideal case, hybrid solutions would not completely replace software simulation. Hybrid solutions will have some inherent limitations that must be understood that restrict what can be mapped to these systems. The physical limitations are less restrictive in software making them ideal for limit studies and rapid hypothesis test beds. Ideal hybrid solutions do not possess the same level of flexibility compared to software simulation. Thus, once the idea is initially proven, the hybrid prototyping can start in parallel with the software simulation effort, both maturing in parallel with the hybrid prototype drafting the software simulation development. Obviously, we have not built the ideal hybrid prototyping platform. This dissertation describes the challenges that we encountered and proposes the next generation hybrid prototyping platform design.

1.5 FAST: A Flexible Architecture for Simulation and Testing

This dissertation introduces a Flexible Architecture for Simulation and Testing (FAST), a hybrid heterogeneous platform for hardware emulation or implementation. The FAST project started in 2002 with the goal of building both hardware and software infrastructure capable of emulating or implementing a variety of CMP and multiprocessor computer systems. By combining FPGAs, SRAMs, and dedicated processor chips with a software toolbox, FAST is able to provide the benefits of software simulation and hardware prototyping, bridging the research gap that inhibits software development and in depth research. This dissertation details the efforts starting in 2004 with the redesign and implementation of the printed circuit board and initial software infrastructure contained in the FAST Software Toolbox. FAST enables full system implementation at the hardware and software level including a variety of memory levels and structures and operating system and application development.

FAST enables hardware implementation or emulation by providing base functionality with both hardware and software. The hardware leverages the functionality of the base components. FPGAs provide the flexibility, expandability, and interconnectivity to map a variety of designs to a single hardware platform. SRAMs provide high speed and high-density memories for a variety of storage structures and multiple memory levels. The dedicated microprocessors provide optimized 32-bit datapaths for both integer and floating-point operations. Combined on to a printed circuit board (PCB), these components provide the foundation for reusable, modular, and scalable framework for multiprocessor systems research.

Layered on top of the hardware is the FAST Software Toolbox. This toolbox is composed of base modules to describe the connectivity and functionality of the PCB. Additional modules define functionality that can be modified for specific computer systems, for example, level 1 and level 2 caches or other memory structures, predictors, or performance counters. FAST also has a predefined "morphware" tool chain for mapping designs to the FPGAs. The application tool chain is also provided for application development. Finally, the base components and in particular the microprocessors can run a ported, fully functional operating system, like Linux.

Figure 1.6. FAST system hardware and software stack.

Figure 1.6 illustrates the complete FAST hardware and software stack. The FAST PCB is at the bottom of the stack with the base functionality and connectivity described in Verilog for the base FAST morphware. There is also a collection of FAST morphware modules that can be modified by FAST users to model different systems. These three components are specific to the FAST hardware and enable morphing FAST to the target architecture. The next two components are enabled by the hardware. The selected

dedicated processor supports fully functional operating systems and all applications, no instruction emulation required. We have not included the hardware or software tool chains, but they are a crucial aspect of the system as well.

FAST is an initial proof of concept that demonstrates the feasibility of a hybrid hardware prototyping platform that integrates hardware back into the research cycle, enabling both software development and in-depth research. In addition to describing the FAST hardware and software toolbox, this dissertation will discuss the limitations of the current implementation and propose the next implementation that addresses the shortcomings of the current system, both on the hardware and software side of a standardized hybrid hardware prototyping platform for computer system implementation or emulation.

1.6 Dissertation contributions

This research has provided insight into the development of a flexible heterogeneous hybrid platform that can be used to implement or emulate a class of computer architectures. The following contributions were made in the process of developing the working hardware proof of concept:

- Design of the ***first*** heterogeneous hybrid platform for full system CMP computer architecture research.
- Definition of the FAST architecture for CMP emulation and/or implementation.
- Implementation of a working printed circuit board prototype using dedicated microprocessors, SRAMs, and FPGAs that required no rework.
- Definition and development of the software toolbox used as the base infrastructure for future research including:
    - Base Verilog infrastructure modules
    - Software Toolbox, including tool chains
    - Working FAST 4-way decoupled CMP implementation
    - Blueprint for mapping Hydra, a thread-level speculative 4-way CMP, to FAST
- Definition of the next generation hardware and software hybrid platform

By combining all of these contributions, significant progress can be made by leveraging these base building blocks of software and hardware to develop more computer systems and software support for future research.

1.7 Dissertation organization

This research has addressed the limitations of current and future software simulators and hardware prototypes by designing, building, and providing the software infrastructure for FAST, a heterogeneous hybrid platform. The FAST design is separated into two main categories, the hardware platform and the software toolbox.

Chapter 2 describes the FAST architecture. This chapter details the initial design goals and connectivity. Chapter 3 details the PCB implementation, including component selection and architecture compromises required to produce the FAST PCB.

Chapter 4 focuses on the software required to make the PCB functional and useful. The FAST Software Toolbox is comprised of the hardware tool chain and the normal software tool chain. The hardware tool chain manages the Verilog infrastructure required for programming FPGAs. We call these software modules that change the PCB functionality, morphware. The other component of the FAST Software Toolbox is the software and software tool chain, which includes the framework for the operating system and applications development. Chapter 4 describes the morphware to software running on the FAST PCB.

Chapter 5 describes a simple 4-way CMP mapped to FAST. It shows how a user can map more complex designs to FAST as well as leverage the base morphware/software infrastructure. The performance results for the 4-way CMP running the Stanford Benchmark Suite is also presented in this chapter.

Chapter 6 details the next generation of flexible hardware prototyping systems. Related work and detailed conclusions are provided in Chapter 7.

Finally, additional FAST details are provided in the Appendices. Appendix A provides an overview of the FAST PCB development process and detailed PCB fabrication information. Appendix B details the FAST Software Toolbox, the collection of Verilog

modules and related constraints and software, with links to the software packages. Appendix C provides key insights used to make FAST operational and additional details for designing the next generation prototyping system. Appendix D provides additional explanation of the results and all of the data in tabular form for the initial prototype presented in Chapter 5.

Chapter 2

FAST architecture

This chapter outlines the motivation for building FAST and the FAST architecture. The first part of this chapter describes the impetus for building FAST and the evolution from hardware prototype to flexible prototyping platform. This section starts by providing a brief description of FAST's original target, the thread level speculative (TLS) Hydra CMP. We then expand the application of FAST to other CMP and multiprocessor systems that can be mapped to FAST.

The second part of this chapter describes the resulting FAST architecture. The FAST architecture defines an intelligent flexible interconnect that enables an entire class of computer architectures to be mapped to the same platform simply by changing the underlying hardware description. The FAST vision is to amortize the hardware costs and to leverage software, both for programming FAST and for using the same operating systems and applications, across multiple projects. The FAST vision and FAST's flexibility are demonstrated by describing various architectures that can be mapped to FAST.

2.1 Motivation and history

The FAST project started in 2002. It started as the Hydra Board Design (HBD). At that point, Hydra, a four-way thread level speculative CMP [44], was not going to be implemented as a chip. By combining FPGAs, dedicated processors, and SRAMs, Hydra could be implemented as a printed circuit board with the correct latency ratios, but an overall slower processor clock frequency. Because FAST initially started as a hardware prototype of Hydra, most of the architecture for FAST was derived from the Hydra design. Extending the Hydra architecture for the PCB implementation added the flexibility to implement other architectures or computer systems. This converted HBD, a one-off hardware prototype, to a flexible substrate for multiple system implementations. After a couple years of stalling, the FAST project started in earnest in January of 2004.

At that time, CMPs were on the horizon for all computer systems, exacerbating the future computer systems simulation research problem, with no real alternative to software simulation in sight.

Software simulation, the driving force behind new computer architectures, traditionally, is much more limited than hardware because of the impracticality of doing software development with analytical models or software simulators. FAST bridges the resulting research gap by providing a platform that enables all types of software development, from operating system (OS) to user applications. This enables many research areas that previously were impossible or impractical to do using other methods. Finally, a ***mature*** FAST system provides the benefits of a mature software simulator with respect to modularity, flexibility, and transparency, along with the high fidelity and speed of hardware, all at a fraction of hardware's cost.

2.2 Hydra: A Thread-Level Speculative (TLS) CMP

The base Hydra design is a 4-way CMP with private per-processor L1 data and instruction caches and a shared on-chip L2 cache. Hydra uses a separate 32-bit Write-through (write) bus for store traffic from the L1 data cache to the L2 cache and a 128-bit Read/Replace (read) bus to fill L1 cache misses. Distributed shared memory controllers and a central arbitration mechanism manage access to the on-chip L2 cache. The base Hydra design will be described in more detail in Section 2.2.1. However, Hydra is more than just a simple CMP, it integrates a very novel memory system that enables thread level speculation (TLS) [44]. TLS is a technique that can be used to speed up sequential (single threaded) applications by cutting the application into several threads that can be optimistically run in parallel. Section 2.2.2 describes the hardware mechanisms for TLS.

Figure 2.1 illustrates the process that maps a single threaded application that runs on processors 0 (P0) and splits that single thread and maps these threads onto the Hydra TLS CMP. First, a sequential application is partitioned into threads. Initial research has focused on loop-level parallelism and partitioning the loops into multiple threads [46]. Next, the threads are mapped, round robin, onto Hydra's 4 processors. Initially, P0 executes the first non-speculative thread and speculative threads are assigned, using round

robin, to the other processors, P1, P2, and P3. The hardware memory system handles data forwarding and enforces logical memory ordering between all of the processors and speculative threads. If the logical memory ordering is violated, a younger threads reads the same data before the older thread writes that data, the speculative thread is stopped, all speculative data is discarded, and then the speculative thread is restarted. Threads become non-speculative as logically older threads complete and are assigned new threads in round robin manner. This guarantees forward progress and correct application behavior. Figure 2.1 illustrates the non-speculative thread assigned to P0, in light, with speculative threads, in dark, assigned in round-robin order to P1, P2, and P3. Once P0 completes executing, the next speculative thread is assigned to P0 and P1 becomes non-speculative. Thus, TLS enables logically older threads to run simultaneously with logically younger threads as shown in Figure 2.1. An in depth discussion of TLS can be found in Lance Hammond's dissertation, "Hydra: A Chip Multiprocessor with support for Speculative Thread-Level Parallelization" [44].

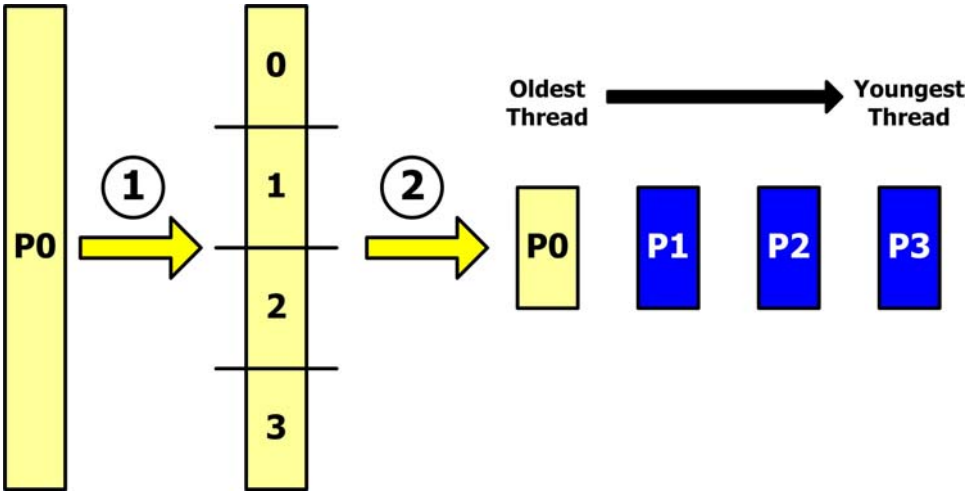


Figure 2.1. Thread level speculation on the Hydra CMP. Processor P0 runs normally, while P1, P2, and P3 run speculatively.

Research in TLS has been limited to application loop-level parallelism because of slow software simulation. A TLS hardware prototype would enable in-depth application and operating system research and development. TLS can be implemented on top of an existing CMP using simple hardware mechanisms. The next sections provide a generic overview of the Hydra CMP and then Hydra enabled with speculation.

2.2.1 Base Hydra CMP

Before diving into the hardware specifics of TLS, it helps to move gently into the base Hydra design, a 4-way CMP, and build from there. As mentioned in the previous section, Hydra has four simple processors; each processor has private L1 data and instruction caches. Figure 2.2 provides a high-level diagram of the entire CMP.

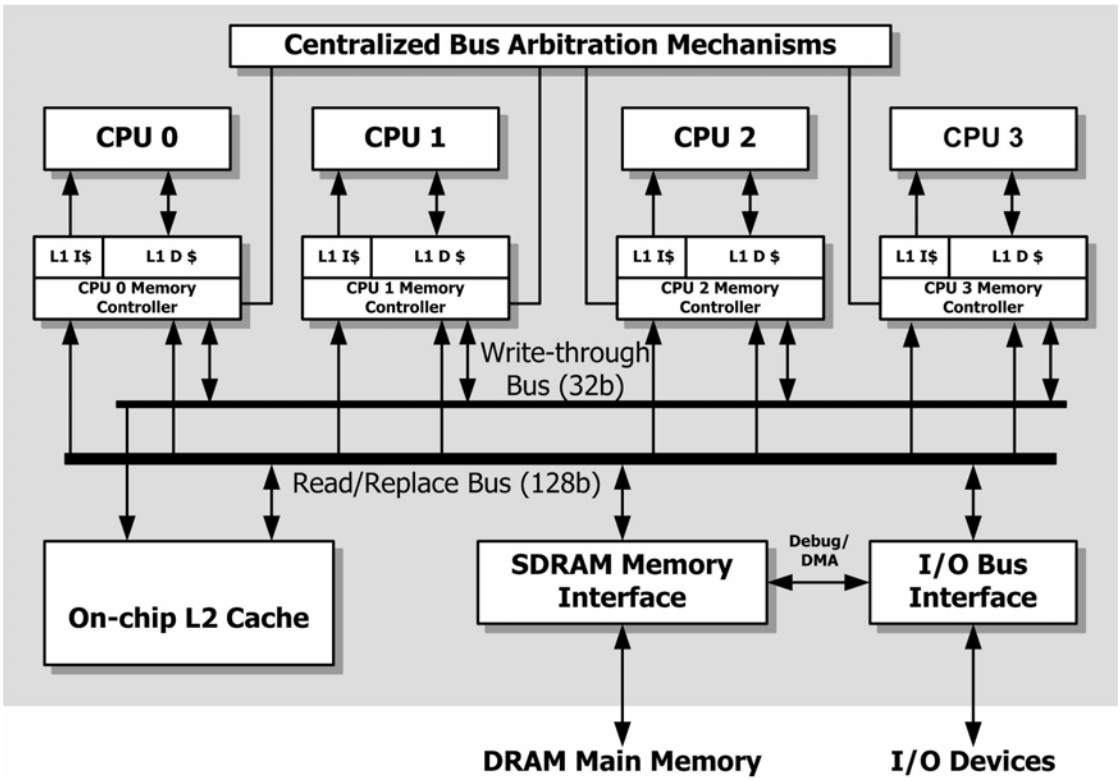


Figure 2.2. Base Hydra CMP with the major components and buses shown.

The Hydra processors use simple 5-stage pipelined RISC embedded processor cores similar to a MIPS R3000 [54]. Each processor has a private L1 data cache that is 8 KB, 2-way associative, write-through, and uses a no-allocate-on-write policy. The private instruction cache is 8 KB, 2-way associative for each processor as well. Because the original Hydra processor core used an off-the-shelf embedded core from IDT, the caches were designed with embedded applications in mind and are too small for typical workstation application. There is also a special cache-invalidation port used to invalidate cache lines for cache coherence. A distributed memory controller and two arbiters manage access to the higher-level memory resources. These arbiters control which

processor can access which bus and prevent multiple accesses to the same address, basically enforcing logical memory ordering of all memory requests.

Hydra differs from many other CMP designs by having *two* system buses, the wide Read/Replace (read) bus and the narrow Write-through (write) bus. These shared buses carry all the data between the processors' caches and the higher levels (L2 and beyond) of memory. The write bus is 32 bits wide and carries all the writes (stores) from any processor to the on-chip L2 cache. The write bus is the key synchronization point, and as such, snooping on this bus is responsible for invalidating all older data in the other L1 caches that may be sharing the same data that had been written. The read bus is much wider, with 128 bits between the on-chip L2 caches and L1 caches and 256 bits between the on-chip L2 cache and main memory. The different bus widths enable entire cache line fills in a single cycle. The initial Hydra design envisioned a 128 KB on-chip L2 with at least 4-way associativity.

Finally, Hydra also has various mechanisms for off-chip communication, both to main memory and basic I/O operations. A much more complete description of the base Hydra CMP can be found in Lance Hammond's dissertation [44].

2.2.2 TLS Hydra CMP

The base Hydra architecture is tailored for thread-level parallelism (TLP) because of the four tightly coupled processors that can exploit the low interprocessor communication latencies. Even though traditional parallel programming will map well to CMP architectures like Hydra, changing sequential programs to a parallel software paradigm is difficult for programmers. Furthermore, correct, high performance parallel programming is very challenging because of the lack of tools, programmers, and parallel machines. TLS was developed to use existing uniprocessor programs and dynamically partition these single threaded programs into multiple threads that can be run on a multiprocessor with very little programmer effort. Attacking the software development component is only a partial solution, the hardware must also be practical. The beauty of Hydra enabled with TLS is that TLS does not add significant complexity to the base Hydra implementation. This section provides a brief overview of Hydra with TLS. More details about speculative thread-level parallelization, the additional low-level software protocol

handlers, and TLS hardware can be found in Hammond's dissertation, "Hydra: A Chip Multiprocessor with support for Speculative Thread-Level Parallelization" [44].

Uniprocessor programs do not consider communication patterns within a program because everything is computed on the same processor. However, when moving programs to a multiprocessor, properly scheduled interprocessor communication is fundamental to good performance. Additionally, for TLS, hardware must be incorporated beyond the normal interprocessor communication mechanisms to support speculation, a specialized form of coherency that tracks and forwards data shared between threads. Figure 2.3 provides a high-level overview of the additional hardware required by Hydra to execute speculative threads. The additional TLS hardware, highlighted in Figure 2.3, is required to provide speculative memory support and control the speculative threads.

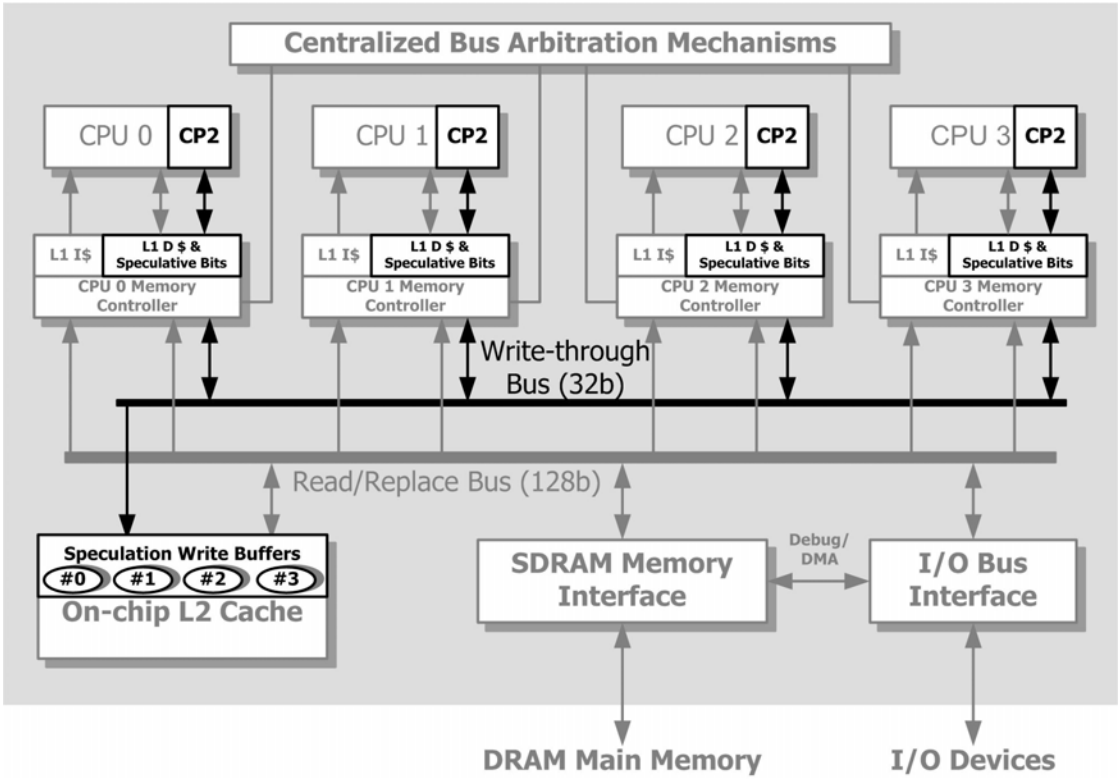


Figure 2.3. TLS Hydra CMP with additional TLS hardware highlighted in bold.

When in speculation mode, if the correct memory ordering cannot be preserved, speculative threads must be stopped and all speculative data must be thrown away. There

are five basic requirements that the memory system hardware must fulfill in order to speculatively execute threads in parallel.

1. Data must be forwarded between threads. The initial uniprocessor programs had no concept of communicating data between processors – initially, there was only a single thread for the program. When running speculatively, data from logically "older" threads must be forwarded to logically "younger" threads running on a different processor, thereby maintaining the initial sequential memory ordering.

2. There may be cases when a read by a "younger" thread gets the wrong data because a write to the same memory by a logically "older" thread occurs after the read. Hardware must be able to track these dependencies and signal when there is a violation. In this case, a true dependency exists which prevents running the speculative threads in parallel.

3. When a violation is detected, hardware must be able to rollback the processor state by discarding all of the speculative machine state. Furthermore, no permanent machine state can be lost as a result of discarding the speculative state.

4. The speculative state that has been successfully executed must be committed, changing the permanent machine state, in sequential program order. This requires the hardware to preserve total store ordering (TSO) when retiring the speculative state.

5. Finally, because there are three speculative threads executing simultaneously in Hydra, the memory system must maintain multiple memory views. This prevents logically "older" threads from seeing changes to the memory that logically "younger" threads make. You want to forward data from "older" threads to "younger" threads, but it would violate program memory ordering if the "younger" threads forwarded data to "older" threads.

A combination of hardware and software handlers is used for controlling and sequencing speculative threads across multiple processors at runtime. In general, software specifies where the speculative threads exist and hardware is used to quickly spawn threads to be

run on the other processors in the Hydra CMP. Selecting the sections in the program to divide into speculative threads is beyond the scope of this work, but it should be noted that selection of good speculative sections is critical for application speed-up. The speculative thread coprocessor, or CP2, in Figure 2.3 is the key additional hardware used to control the speculative memory system and speculative threads.

The CP2 has four main functions that are required for speculative execution.

1. The CP2 interfaces with the L1 cache and controls all the speculative state that has been added to the L1 cache.

2. There is interprocessor communication, in the form of interrupts, that is required for speculation and this communication is coordinated by the CP2.

3. The speculative software handlers are accelerated by specific speculative data managed by CP2.

4. Finally, like many other processors, the CP2 provides hardware support for a variety of runtime statistics for speculation.

The other main hardware features required for speculation are associated with the L1 and L2 memories. Hydra uses private write-through L1 data caches and a bus system, which simplifies the coherence protocols. Ten bits are added to the tag array to indicate and track the cache state during speculation. Using these additional bits, read-after-write (RAW) violations can be detected. RAW violations occur when a logically "younger" speculative thread reads data before it has been written by a logically "older" speculative or non-speculative thread. This forces the system to stop the "younger" thread, clear all of its speculative data, and then restart the speculative thread. These additional speculative state bits differ from normal tag bits because they have to be gang cleared if a violation or commit of speculation occurs. There is a modified bit that acts like a dirty bit in a writeback cache. Upon violation, all entries with a set modified bit are invalidated. There is also a pre-invalidate bit to mark cache lines that have been written by other processors. This bit enables renaming of a particular cache line or multiple views of a particular cache line. After speculation, this bit indicates if that cache line needs to be

updated for the next thread. The remaining number of bits used for the speculative state in the L1 data cache depends on the L1 data cache specifics. In the case of Hydra, the L1 data cache is four words wide, thus there is 1 read bit and 1 write bit for each word in each cache line for a total of 8 read/write bits plus the pre-invalidate and modified bits, for an overall total of 10 bits. These speculation bits are included in that tag array that is used for coherence invalidations.

Because Hydra uses write-through L1 caches, speculative data must be buffered in the L2 cache and stored until it is ready to be committed to the permanent machine state. Additional buffers are added, one per core, to store all speculative writes at the L2. If the speculative thread is restarted, all the data in that thread's L2 write buffer is cleared. However, if the speculative thread successfully completes, then that data in the L2 write buffer is written to the L2 cache.

For performance reasons, Hydra double-buffers the L2 write buffers. This enables the processor to continue by committing the data in the background while the other buffer is used to store new speculative data. Each processor has its own L2 write (double) buffer, as shown in Figure 2.3. However, this double buffer design could be replaced by a single L2 write buffer per CPU and an additional retire buffer shared among all four processors. This additional buffer would enable the processor that is committing all of the data in the L2 write buffer to proceed by using the additional retire buffer to start the next speculative thread. Otherwise, the processor must wait until all the data in the L2 write buffer is transferred to the L2 cache before it can continue to run the next speculative thread. This would reduce the buffering constraints from 2N down to N+1, where N is the number of processor cores in the CMP.

Finally, when data is requested from the L2 cache during speculation, priority encoders are used to find the appropriate data version from the L2 cache, non-speculative write buffer, or speculative write buffers, respectively.

In reality, the TLS Hydra CMP is one example in the spectrum of possible TLS CMPs. Hydra uses both software and hardware to execute threads speculatively, but designs exist that are more hardware or software centric, providing a spectrum of TLS solutions [62,

86, 87]. University of Wisconsin's Multiscalar distributed speculative basic block sized threads using a dedicated ring that connected the processors. This schema required centralized resources or complicated cache coherence protocols that made it difficult to scale [36, 43]. Similar to Hydra, the TLDS project used simple cores and a software system to manage speculation, making speculative thread violations very expensive [87]. Finally, the University of Illinois started with a tightly coupled system similar to Multiscalar that was then relaxed to resemble TLDS [62]. Thus, within the TLS framework, many academic design points exist that could be explored in further detail and enable software development using a hybrid platform.

2.3 Prototyping targets

CMPs are everywhere. Unfortunately, the ability to contribute software research to this area is diminished because of the lack and speed of the current tools, mainly software simulators. The concept of a hybrid prototyping platform grew out of the inability to build hardware prototypes. In particular, we wanted to build machines with four or more tightly coupled processors that enabled software and further systems research. Even though the processor industries' ITRS roadmap continues to predict higher processor clock frequencies, we have witnessed the rate of clock frequency increases slowing down [98]. The increase in external memory speed continues to lag processor frequencies and this memory wall continues to provide opportunities for improved system performance enabled by novel memory system design. This new focus on the memory system de-emphasizes processor centric design methodologies because of the lack of instruction level parallelism (ILP). Thus, the investigation of ever-larger and more complex processors has been throttled and simple processors can be used in place of complex processors to focus on memory system and overall system design. Fortunately, by focusing on the multithreading capabilities of future microprocessor architectures, instead of ILP extraction mechanisms, we greatly simplify the core CPU pipeline of our base processors. Therefore, it is possible to build a flexible research prototype platform around existing simple processors without doing any new VLSI design.

The need to build a hybrid hardware prototyping platform grew out of the desire to build the Stanford Hydra processor [46]. In designing a hybrid prototyping platform, we

realized that we could prototype a variety of architectures with the same base platform by simply changing the software that configures the platform. Thus, with this hardware prototyping platform, we can amortize the cost and development time over multiple architectures within the CMP and novel memory systems design space. There are several other architectures that can easily map to this base hybrid prototyping platform that fall outside the TLS architectures mentioned at the end of Section 2.2.2.

These are just two of the possible architectures that can by mapped, in part or the entire system, to the hybrid prototyping platform.

- Transactional memory systems like Stanford's transactional consistency and coherence (TCC) CMP, which uses similar hardware and a different programming paradigm to achieve optimistic parallel execution [47].

- Stanford's Smart Memories project focused on configurable memory structures within the processor and a hierarchical on-chip network for interprocessor communication [69].

Hydra, TCC, and Smart Memories illustrate new architectures that require application development to realize the architecture's full potential. Initial research has focused on the low hanging fruits, scratching the surface with regard to application development and in-depth studies. For Hydra, research has been limited to loop-level speculation. Likewise, TCC research has been limited to modifying existing parallel applications, a few targeted applications and some other components of the software stack, like a JVM [19, 45, 78]. Similarly, Stanford's Smart Memories project follows suit, like many other academic computer systems projects that have limited software development capabilities and limited research depth.

2.4 FAST architecture

The FAST system is a collection of hardware and software components that manage and configure on-board resources to enable full-system prototyping and software development. The on-board resources exist as functional layers that together can prototype multiprocessor hardware systems at high speeds. The layers include: (1) the

hardware: fixed-function SRAM memories, microprocessors, FPGA devices that can be "morphed" to provide different system-level functionality using Verilog models (morphware), and (2) the software, what we call the FAST Software Toolbox: the morphware, application benchmarks to be evaluated, low-level software and operating system functionality to manage functions such as program loading and I/O. The FAST architecture facilitates the combination of these components to prototype a variety of architectures. FPGAs are the key components that have moved beyond simple glue logic implementations to more complex system integration implementations due to their high density, enabling FAST and other FPGA prototyping systems. For the rest of this chapter, we describe the system architecture and defer discussion of the physical implementation and FAST Software Toolbox until later chapters.

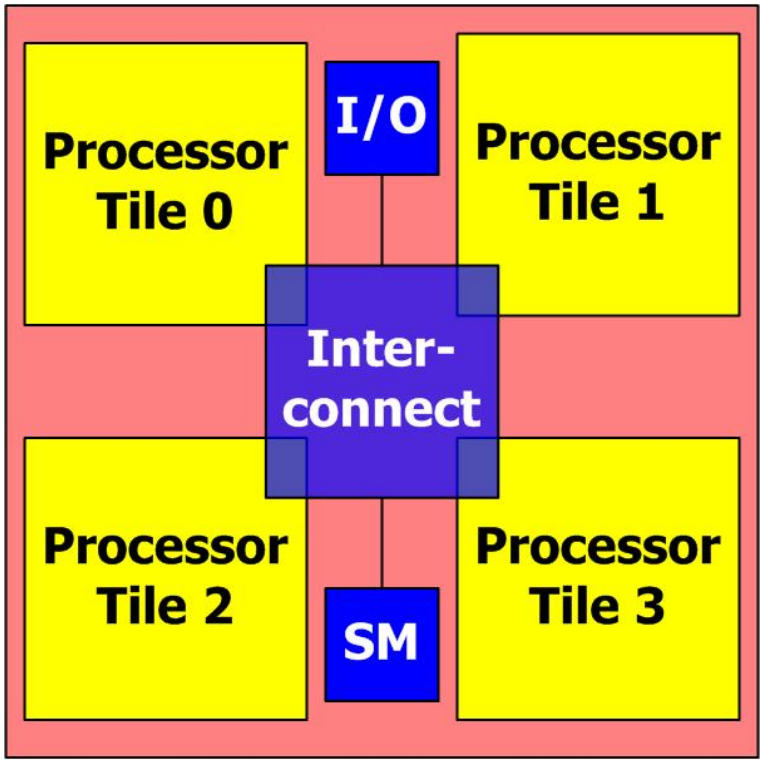


Figure 2.4. FAST high-level architecture.

Figure 2.4 illustrates the high-level architecture for FAST, which was initially based on the Hydra architecture. Using a PCB substrate, FAST tightly couples four processors and provides shared memory (SM) and I/O capabilities. We have defined *processor tiles* and not just processors because we want to add functionality to the processor, which requires more chips, like the additional functionality of the speculative thread processor (CP2) in

Hydra. There is also a PCB interconnect that enables interprocessor and memory system communication. For the Hydra CMP, this interconnect would include the read and write buses, as well as some of the control signals for memory arbitration. The shared memory (SM) provides the L2 cache infrastructure required by Hydra and other CMPs. Finally, FAST requires I/O capabilities for moving program data and other information on and off the PCB.

2.4.1 Processor tile

FAST is made up of four processor tiles. As shown in Figure 2.5, the processor tile is made up of a CPU, Processor FPGA, and L1 memory. The CPU is the base processor that can perform both integer and floating-point computation. Floating point emulation is very expensive, thus it is very beneficial to have a processor that is IEEE 754 floating point compliant to remove any instruction emulation overhead. We include a FPGA in the processor tile to serve as the L1 memory controller and provide additional functionality. Finally, L1 memory is included in the processor tile to serve as the L1 cache or other memory structures. The L1 memory structures depend on the functionality programmed into the Processor FPGA. By adding another level of logic between the CPU and L1 memory, the FPGA can provide the appropriate interface that both subsystems expect, thereby enabling the user to create novel memory structures oblivious to the CPU.

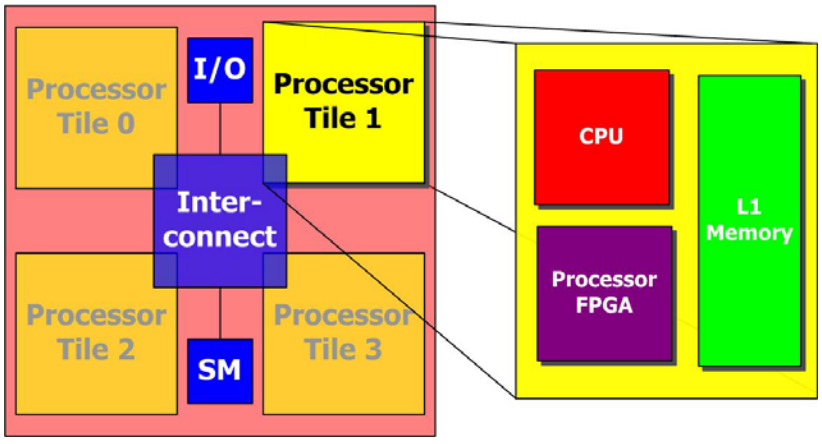


Figure 2.5. FAST Processor tile.

Figure 2.6 shows the interconnect between the components in the processor tile as well as components outside of a tile, for example, the top-level interconnect and the

interprocessor tile (PT) buses. The CPU is agnostic to the FPGA. The CPU provides an address and expects data, along with a tag to verify that the data is correct. In general, we designate bi-directional buses with double-headed arrows and unidirectional buses with single-headed arrows. The CPU generates addresses that must be forwarded to the local L1 memory to be processed. If there is a memory miss in the L1 memory, the address must be forwarded to the higher memory levels to be serviced. The FPGA provides this functionality. Furthermore, the FPGA can provide additional functionality, for instance, instruction set architecture (ISA) extensions and auxiliary compute engines like the CP2 in Hydra. The Hydra coherence protocol also requires cache invalidation by other processors. In order to invalidate lines in the cache, the FPGA must be able to provide multiple addresses simultaneously and the L1 memory must be dual ported or time multiplexed. Thus, there are multiple data and address interfaces between the FPGA and L1 memory. In a traditional processor configuration, the FPGA would manage data and tag buses for the instruction and data segments of the L1 memory. This requires a data and tag array for the data and instructions. In order to snoop or invalidate the cache, the L1 memory requires two address ports and two data ports for the data and tag arrays for the data and instruction caches.

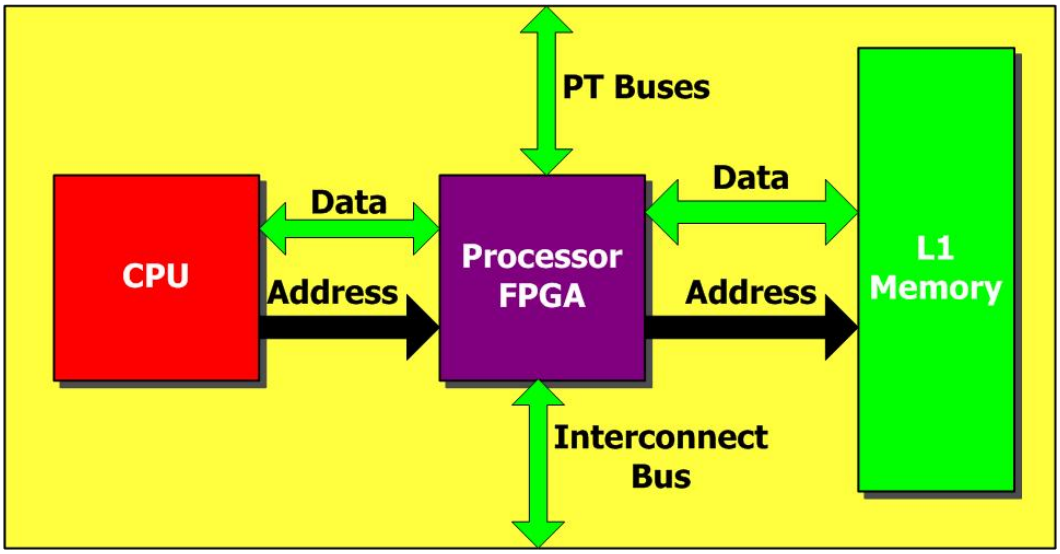


Figure 2.6. Processor tile buses.

As a result of the four dual-ported data arrays (i.e., the data and tag arrays for the data and instruction caches) the FPGA interface requires 8 32-bit bi-directional data buses and 8

unidirectional addresses to provide any combination of data and address to dual ported L1 memory. This would enable all possible addressing modes and independent memory structure within the processor tile.

The FPGA also provides point-to-point buses and a shared bus to the other processor tiles for rapid interprocessor communications, as shown by the PT Buses at the top of the FPGA in Figure 2.6. This could be used for global coordination and point-to-point communication. Likewise, a wide bus from the FPGA to the higher level of memory is required to model multi-level memory structures and centralized communication required for coherence. Mapping the interconnect bus in Figure 2.6 back to Hydra, this bus would include, but not be limited to, the read and write buses, centralized arbitration signals, and distributed memory control signals.

2.4.2 Interprocessor communication

FAST requires a central or Hub FPGA to orchestrate interprocessor communication and manage shared resources. The Hub FPGA is shown in Figure 2.7. The Hub FPGA distributes the read and write buses to all four processor tiles, PT 0 to PT 3. This FPGA also controls access to the higher-level shared memory and provides an interface to the off-PCB I/O. An FPGA is fundamental to the FAST design because it has the ability to implement multiple interfaces using sophisticated logic.

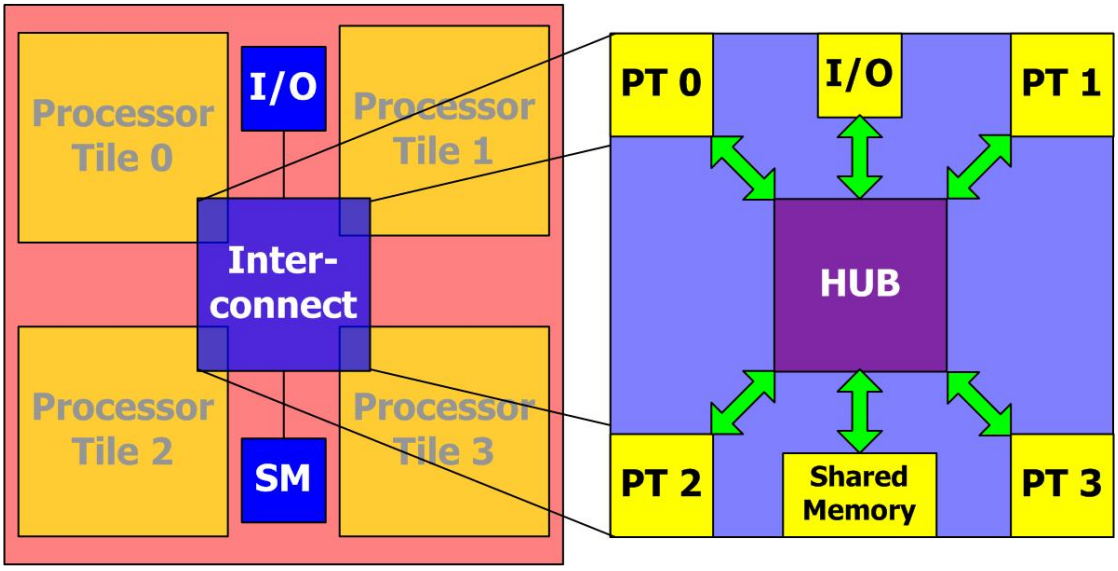


Figure 2.7. FAST interconnect architecture.

Input/Ouput

The Hub FPGA also manages the on/off PCB I/O. The PCB requires I/O to service higher memory misses and file I/O, enable serial terminal communication, and enable TCP/IP connections to the FAST system. FAST I/O extends the capabilities of the system by providing more levels of memory and/or the ability to off-load functionality onto a host machine. For example, FAST I/O can be used to service OS operations like file I/O by using a host machine to open and close files and service the actual requests. This reduces the infrastructure development required for a fully functional system, thereby reducing development time.

Shared memory

FAST has an L2 memory that can be configured using the Hub FPGA memory controller. This memory can be shared among all four processor tiles as indicated in Figure 2.7, or the memory could also be partitioned as separate memories, one for each processor tile. Thus, the FPGA interface to the shared memory must be able to service four simultaneous memory accesses when the memory is shared. Likewise, multiple memory accesses can be used to replicate multi-way caches. Finally, significant amounts of shared memory must be available for a variety of memory structures and to satisfy the trend of enormous higher levels of cache, like Intel's Montecito processor with an aggregate of over 26 MB of cache [71].

2.4.3 Miscellaneous architecture features

Several other FAST architectural features help FAST bridge the gap between hardware prototyping and software simulators. Most of these architectural features are required to make the hardware have the same useful characteristics as software simulators. The PCB management must have fine-grain control over the clocking and reset, as well as the ability to program the FPGAs on the FAST PCB and debug applications running on FAST. FAST must also have the same level of transparency or observability as software simulators. As silicon technology scales, microprocessors are more susceptible to faults. Thus, FAST must include facilities for fault injection and observation for fault tolerance studies. Finally, FAST must have scalability in mind for building larger systems or adding

functionality that was not explicitly defined at FAST's conception. Given the configurable building blocks, FAST can have all of these features.

PCB management

Software simulators provide very fine-grain manipulation, and this can also be implemented in FAST. The PCB management encompasses the ability to reset, distribute and control the clock, and control the state of the PCB. The state of the PCB changes from programming the FPGAs on the PCB, running applications, to debugging applications. We must define these attributes in the architecture to ensure the functionality in the FAST System, both hardware and software. Each processor tile has both an individual reset as well as a system reset signal. This allows both targeted reset and restart or entire PCB reset and restart. This is useful for sensitivity studies or fault-injection studies where inputs, i.e., switches, can be changed and the same setup can be used to rerun the experiment under slightly different conditions. The FPGA and global reset signals provide fine-grain and coarse-grain control.

The PCB clocking is centrally controlled and distributed. FAST has the option of using the global clock that is distributed to all the FPGAs or FAST can modify the clocking at each FPGA using clock management hardware in each FPGA. FAST can use a predetermined clock crystal to set the base clock frequency or use an external clock source for even more flexibility. In all, there are at least two different clock domains on FAST. The first clock domain is the frequency of the processors and the second clock domain is the frequency of the memories. We envision the memories using a higher frequency than the system clock, enabling time-division multiplexing to create multi-way set associative caches, control and vary the latency of memory accesses, and create other novel memory structures. The FPGAs provide the digital interfaces, frequency synchronization, and clock generation to make the devices and multiple clock domains work together.

FAST also has multiple ways to program the FPGAs. Having two ways to do the same thing provides a backup mechanism if something fails. For FPGA programming, we can trade-off programming speed using a parallel method versus ease of programming using a serial method. By storing the FPGA programs locally on the board, we can use the

FPGA parallel programming methodology for rapid FPGA programming. The main drawback of this technique is the additional intellectual property required and a higher level of integration. The main benefit is programming speeds of at least 8 times faster. The secondary FPGA programming methodology uses the JTAG port. JTAG, or Joint Test Action Group, is the usual name used for the IEEE 1149.1 standard, "Standard Test Access Port and Boundary-Scan Architecture." This standard was defined for test ports on PCBs, but today it is a rich interface that can be used to monitor I/O pins and debug integrated circuits and their subcomponents [52]. We already use the JTAG port for visibility and we can also use it to download programs to the FPGA. Thus, we can leverage the JTAG infrastructure for multiple purposes, which is always a win.

Finally, changing the state of the PCB is required for different functionality. At power-on, the PCB must start in a state that allows the FPGAs to be programmed. Programming the FPGAs can be a manual or automated process depending on the underlying infrastructure. Once the system has the hardware programmed, the PCB transitions to a ready state that enables it to run experiments with various inputs or applications. When running applications on FAST, it may be necessary to single-step through the applications to understand the systems behavior. Enabling a debugging mode initiated by the Hub FPGA and controlled by either the Hub or Processor FPGAs provides the same functionality as software debuggers. This would require fine-grain clock control and PCB mode control to indicate the desired debugging mode. Furthermore, debugging can be focused on one processor tile or the entire system could be operated in lock-step, single cycle mode. Finally, debugging mode could also be used to turn on and off tracing and performance counters based on triggers in the application or other external events. PCB management and other future functionality require including these control signals.

Transparency

The biggest advantage of using software simulators is the ability to observe the complete system. All components in the software simulator are transparent to the user either through pre-defined interfaces or by using a software debugger. This key advantage is lost with hardware prototypes. However, FAST is transparent because of the use of FPGAs and the use of JTAG enabled components. The Corelis boundary scan tools can operate up to 80 MHz, much faster then maximum scan rate of the FPGAs,

approximately 10 MHz [26]. FAST uses the JTAG interface to observe chip-to-chip interfaces, making these signals transparent without using a logical analyzer. Moving into the FPGAs, embedding monitoring Verilog modules enables transparency at the subcomponent level. Thus, the combination of JTAG and monitoring modules yield the transparency that approximates that of software simulators. Unlike software simulators, FAST's transparency is orthogonal to system performance. There are resource limitations with respect to the FPGAs that can limit the number of monitoring Verilog modules and the quantity of data gathered, but external I/O can solve this problem.

Scalability

The Hub FPGA also must have an expandable interface. As silicon technology progresses, CMP systems will incorporate more and more processor cores. An expandable interface will enable multiple FAST PCBs to be connected together, thus, enabling large systems to be mapped to a FAST compute fabric. Because this expandable interface uses an FPGA, we have the ability to not only scale the FAST compute fabric, but we can also add other digital interfaces. The base FAST PCB does not include a L3 memory. The expandable interface can be used as a L3 memory interface to DRAM or a Compact Flash daughter card. This interface could also be used to interface to an IDE hard drive. In general, FAST provides both system extensibility and scalability.

Fault tolerance

As transistors continue to shrink, they become more susceptible to soft errors. This phenomenon has forced more and more recovery-oriented solutions into the processor, like ECC protected memories. Most research has focused on fixing errors once they have been detected. Far less research has tried to detect faults. FAST gives access to the buses, memory, and computation for the user to inject deterministic faults to enable further research into fault tolerance and in particular fault detection and recovery.

2.4.4 Complete FAST architecture

Putting all of these features together yields the high-level FAST architecture shown in Figure 2.8. There are four tightly coupled processor tiles composed of the L1 Memory, CPU, and Processor FPGA connected to the Hub FPGA. There is a very wide bus between the Hub FPGA and the processor tiles. There are both point-to-point and

shared buses connecting all of the FPGAs for a variety of data movement and control signal configurations. There are centralized auxiliary components to provide a base system clock, PCB management, and transparency. There are also several features that enable off-PCB I/O and scalability to create a FAST compute fabric and extensibility. The Hub FPGA is also connected to a second level of memory that can be shared or partitioned into private blocks. The expandable interface can also be used for a third level of memory, if desired. Because FPGAs are the flexible building blocks of the FAST architecture, we are able to morph the hardware into a variety systems, all running at hardware speeds.

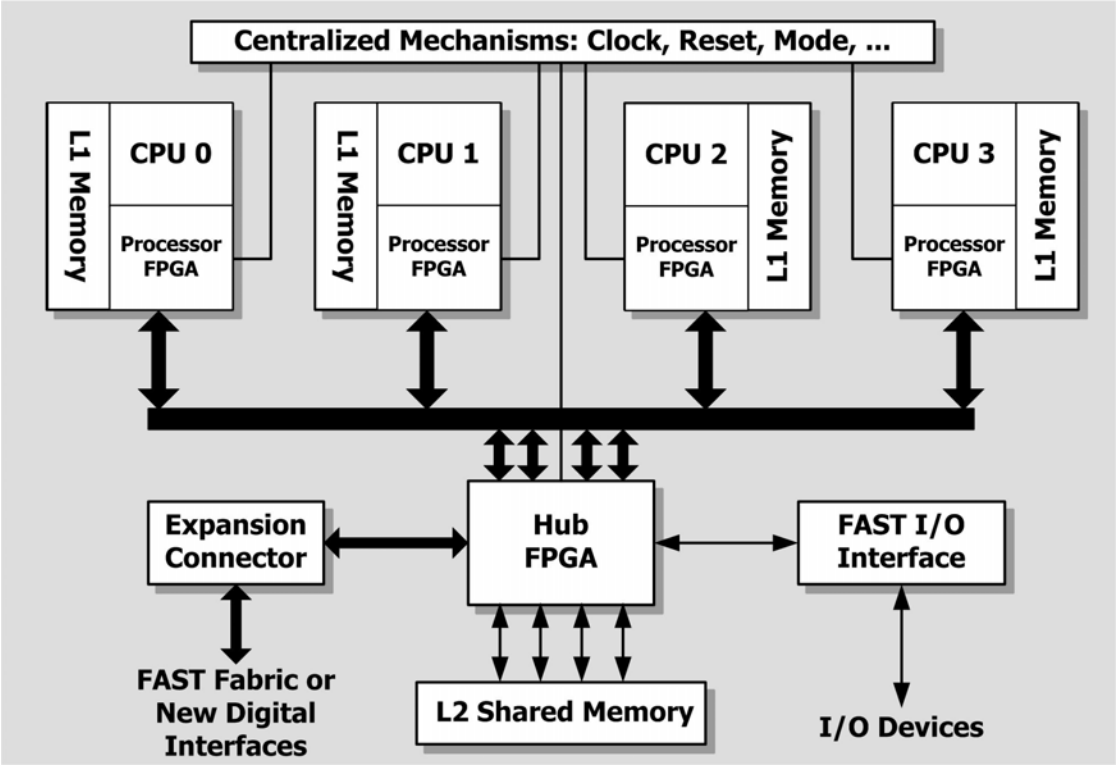


Figure 2.8. Detailed FAST architecture.

2.4.5 FAST prototyping candidates

Section 2.3 provided a list of initial prototyping candidates for FAST. This section provides more details on mapping those designs to FAST. Overall, the FAST PCB is a fixed core CMP platform coupled to highly configurable memory hierarchy. Designs that are based on multiple processing cores map well to the FAST PCB and all or part of the design can be implemented. Furthermore, like Tensilica or other designs that augment a

base instruction set architecture (ISA), FAST can use the FPGAs to implement the ISA extensions [91]. The CPU ISA can also be completely disregarded if the user wants to implement a software-defined processor core in the FPGAs and create the related software infrastructure. Finally, an abundant number of co-processors can be implemented because the Processor FPGA decouples the processor memory system, integer datapath, floating-point datapath, and the coprocessor datapath. As a result, the memory system can be configured as caches of varying set associativity, FIFOs, or other interesting memory structures. Furthermore, FAST's architecture supports sizes of the L1 and L2 memories that far exceed current and near term available processor on-chip memory.

Mapping Hydra

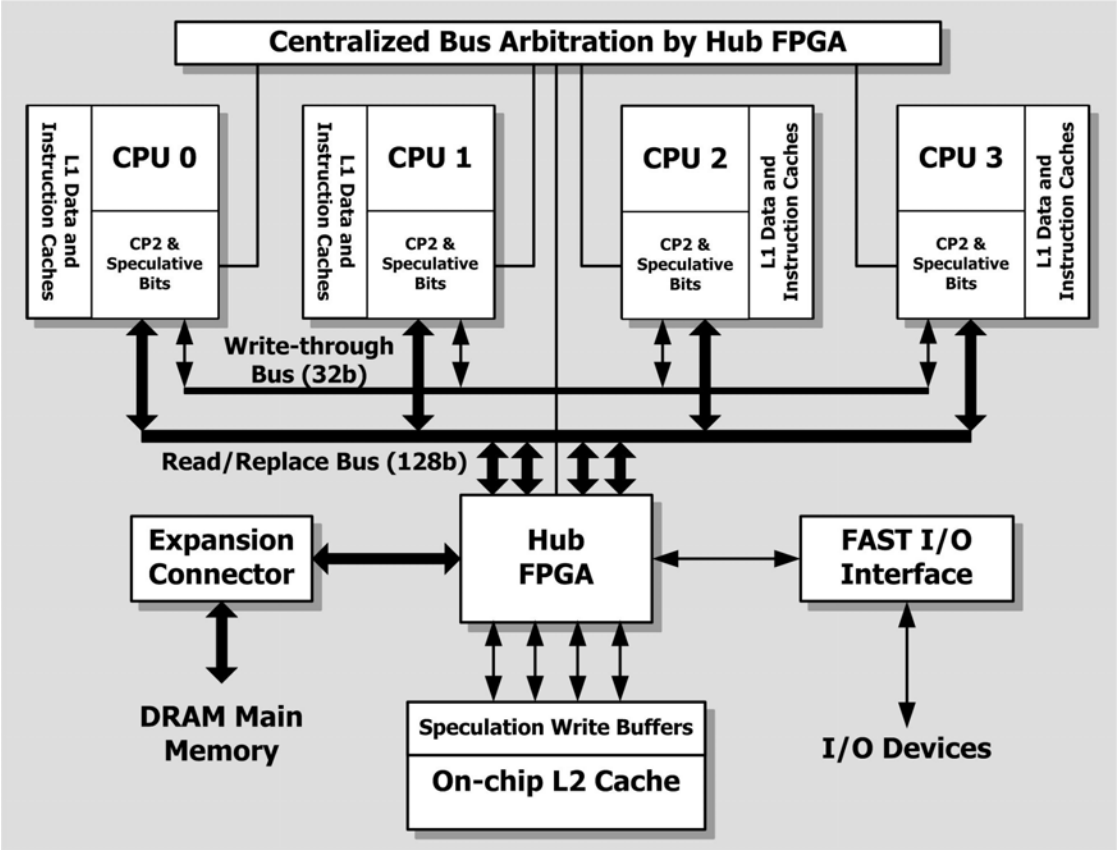


Figure 2.9. Mapping the Hydra architecture on to the FAST architecture.

Hydra motivated the development of the FAST architecture, so it is only appropriate to show how the Hydra architecture maps onto the FAST architecture. Both FAST and

Hydra are four processor systems. In each of FAST's processor tiles, the CPU and L1 caches map to the Hydra equivalent components. Likewise, the processor FPGA implements Hydra's speculative coprocessor, CP2, and maintains and manages all of the speculative bits. A portion of the very wide bus between the processor tiles and the Hub FPGA is allocated to Hydra's Write-through and Read/Replace buses. Furthermore, the Hub FPGA is the point of coherence for the Hydra system and as such, controls the arbitration over the buses. The Hub FPGA also manages the L2 SRAM's as an on-chip L2 Cache with the speculation write buffers residing in the SRAM's or the FPGA. The expansion connector can be used as a DRAM interface using an 80-pin daughter card for a main memory. Finally, the FAST I/O interface can serve as the Hydra I/O interface. The mapping of the Hydra architecture onto the FAST architecture is illustrated in Figure 2.9.

A large-scale, networked CMP

Several research projects have proposed architectures composed of multiple "tiled" processors on a single chip [69, 83, 96]. Since FAST can only prototype four-processor systems, at most, one might initially conclude that it is not useful for emulating these architectures. However, FAST can still prove useful. Many insights can still be gained by prototyping just a four-processor subsection of a larger design. Furthermore, one can use the expansion connector to implement or emulate larger systems by connecting multiple FAST boards together. While the expansion connector only has a relatively limited number of pins, many advanced architectures use a network that limits the number of long wires needed to connect processors together when they are physically distant from one another on the chip. Such relatively narrow networks should map well to the expansion port. If necessary, some of the secondary memory could also easily be used to support network buffering requirements, if they became too large for on-FPGA buffers.

In order to verify these ideas, we looked closely at the Stanford Smart Memories design [69] while designing the expansion port. This CMP has 10's of processor tiles clustered together into small groups of tiles that share a common network port. A single FAST board could emulate a group of four processors, while the expansion port could be daisy chained to additional FAST boards to allow emulation of a large system.

Emulating more complex cores

The CPU's external interfaces to both cache and coprocessors provide visibility that can transform FAST's simple in-order single-issue cores into a wide variety of other microarchitectures, because the Processor FPGA in each processor tile has full control over the data and instruction streams fed into the processor. Auxiliary structures can be maintained in this FPGA, such as counters and monitors, to make it possible to change the definition of a "simulated machine cycle" or to adjust or interpret the instruction streams. With this FPGA, we can define several CPU cycles as one target machine cycle and thereby gang instructions together into "single-cycle" packets.

Depending on the underlying architecture, these instruction packets could be executed intra-tile (very long instruction word, *VLIW*) or inter-tile (single instruction multiple data, *SIMD*). We prefer to execute VLIW packets serially, instead of in parallel across the processor tiles, because simulating the shared register file used by all VLIW issue slots across processor tiles would require many extra store instructions to make all instruction results visible to the other Processor FPGAs. This method could use instructions that mark the VLIW boundaries or a fixed number of clock cycles are allocated for a VLIW instruction. On the other hand, SIMD packets can be spread across processor tiles executing identical instruction streams because one instruction specifies the operation for several parallel data "lanes" of execution, which do not share data between lanes on a cycle-by-cycle basis. When the number of "lanes" exceeds the number of processor tiles, both serial and parallel execution methods may be used. Finally, cores with more complex instruction fetch mechanisms could also be implemented using these techniques with variable-length "simulated machine cycle" times. Implementing a *small* instruction window in the Processor FPGA could be used to enable fine-grain multithreaded emulation and/or wide-issue superscalar core emulation.

Embedded SOC architectures

VLSI process scaling has increased the complexity of embedded and application-specific processor-based designs. FAST enables full or partial system emulation for a wide variety of these systems by manipulating the memory hierarchy and defining the number of processor tile cycles per target machine cycle. Simple processor cores are widely used for

embedded systems and we envision collections of these processors working in concert for system-on-a-chip (SOC) designs. FAST is an ideal prototyping platform for continued research in these types of embedded systems [80].

2.4.6 Architectures not suitable for FAST

There is a class of architectures that does not map well to the FAST PCB. This class of architecture contains a sea of ALUs or other functional units. The FAST PCB has a restricted number of fixed functional units and a fixed number of synthesized units that fit in the FPGAs, thus several designs that tightly couple tens to hundreds or more functional units do not map well to FAST. We have included a scalability port in the FAST architecture that would enable building a FAST fabric using several PCBs, but it quickly becomes impractical to build a FAST fabric of more than 16 PCBs given the technology and connectivity provided in this first generation PCB. Although significant effort could be used to map portions of these architectures to FAST, it would be counterintuitive to the FAST PCB architecture and thus present a difficult challenge. Likewise, because FAST focuses on TLP extraction systems and novel memory system design, attempting to model complex processors with out-of-order execution and *very large* instruction windows would not map to the FAST infrastructure. Limited superscalar and VLIW designs can be mapped to FAST, but in general, processor pipeline and execution design falls outside to scope of the FAST system.

2.5 Software infrastructure for FAST

This chapter has described the hardware architecture of FAST. FAST is more than just a PCB hardware architecture, it is a collection of hardware and software components that manage and configure on-board resources to run operating systems and user applications. As shown in Figure 2.10, these resources exist as functional layers that together can prototype multiple multiprocessor hardware systems at high speeds. The layers include: several fixed-function memories and microprocessors (hardware), FPGA devices that can be "morphed" to provide different system-level functionality using a variety of Verilog memory hierarchy models (morphware), and application benchmarks to be evaluated, low-level software and a batch operating system to manage functions such as program loading and I/O (software). These are depicted from the bottom up in Figure 2.10. The

FAST Software Toolbox is the collection of modules and pre-defined interfaces that provide the base functionality of the PCB and application benchmarks, along with all of the software tools required for the development of the morphware and software used with FAST. Chapter 3 describes the FAST PCB implementation. Chapter 4 discusses the FAST Software Toolbox, completing the description of the FAST prototyping system.

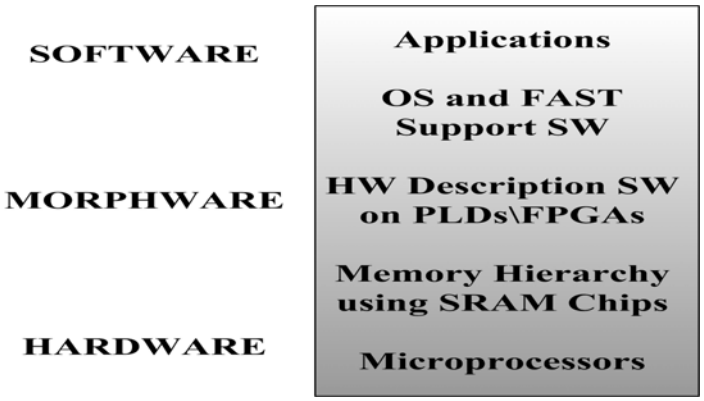


Figure 2.10. All FAST hardware and software components.

2.6 Related work: old and new

Related work fits into three main categories: previous work, on-going solutions, and future solutions. Reprogrammable logic and FPGAs, in particular, have been used as prototyping platforms since their inception. Low cost solutions are used in the classroom and high cost solutions are used in industry [6, 17, 18, 73, 115]. Regardless of the scale, the purpose is the same: rapid prototyping of a system or component. As silicon technology has improved, the capacity of reconfigurable devices has dramatically improved. Today, simple processors can be mapped to a single FPGA [38]. Older technology required either partial system prototyping or partitioning a design across multiple FPGAs. Crossing chip boundaries has been a fundamental limitation because off-chip bandwidth is nowhere near that of on-chip bandwidth. Furthermore, some interfaces are not amenable to partitioning because of the wide bus interfaces or similar hard-to-map structures. Moving forward, both FPGA capacity and increased per-pin bandwidth enable much greater prototyping flexibility. For these next generation prototyping platforms, the chip boundaries may be able to be erased by the high bandwidth I/O between chips.

There have been several research projects that have incorporated reconfigurable hardware to enable limited design exploration around a single design point. These hardware prototypes were one-off systems that validated a particular idea [5, 40, 66, 96]. Producing chips has become so expensive that most research projects can no longer afford to build systems based on new chip designs. In general, FPGAs have been used for system observation and not full system prototyping. FAST, as well as some other research projects, integrates various off-the-shelf components with FPGAs onto a PCB substrate to build complete systems, pushing FPGAs beyond system observation to system integration and/or system implementation.

The Rapid Prototyping engine for Multiprocessors (RPM) was an initial hardware emulator for multiprocessor architectures that moved beyond simply observing system behavior like bus traffic [9]. RPM was built to prototype Multiple Instruction Multiple Data (MIMD) multiprocessor machines. The configurable technology available for this project enabled reconfigurable memory system implementations atop a fixed SPARC processor. Each processor module was implemented on a single PCB with up to eight PCBs connected to a backplane and host machine. RPM's coupling of configurable logic and fixed processors is very similar to FAST. However, FAST differs from RPM in many ways. The level of integration is much higher in FAST, as would be expected with newer technology. Furthermore, RPM focused on the memory system of large multiprocessor systems, while FAST can prototype the memory system and additional compute engines, like speculative coprocessors or other off-load engines, in large multiprocessor (MP) or small, fast CMP systems. As Chapter 5 demonstrates, FAST is able to replicate the memory latency for both MP and CMP systems, as well as somewhere in between. FAST also integrates a more complex memory system with multiple levels of memory and transparency. Like RPM, FAST preserved the memory system transparency all the way down to the processor.

RPM is a clear predecessor to FAST that provides a similar solution to the implementation problem for a more narrow class of architectures using much older commodity parts. FAST is able to implement a broader class of architectures than RPM, but both lack the software infrastructure to make them easy to use and to be adopted like free software simulators [12, 68, 81].

Generic FPGA arrays have also been used to prototype both academic and industrial projects [9, 17, 18, 32, 73, 93]. These generic FPGA arrays connect a sea of FPGAs with some peripheral devices on the edges to support ASIC or chip development. These systems are either developed in-house for research projects, like the RAW FPGA fabric [9] or else they cost several million dollars and are only affordable for industry [17, 18, 73, 93].

The RAW FPGA fabric enabled researchers at MIT to validate the RAW CMP design using a network of 64 FPGAs on a single PCB to emulate a network of processors on a single chip [96]. The FPGAs used for this project had enough capacity to implement simple MIPS R2000 processors and change how the FPGAs communicated. The adaptive communication network was a main thrust of this research project. The RAW FPGA fabric enabled in depth research on RAW, but the RAW fabric lacked the capacity and the ease of use for mapping other designs. Finally, the design mapping was very labor intensive because the programming file needed to be composed of all the individual FPGA programming bit files, which consumed many computer resources and copious amounts of time. Partitioning designs across 64 very small FPGAs also presented a significant challenge and considerable communication delays because of pin pressure. The FPGAs used for RAW are twenty times smaller than the XCV1000 FPGAs used for FAST [100, 101]. Even though RAW used a commercially available emulation platform, the difficulty experienced when mapping other designs to this resource-constricted FPGA array prevented the RAW FPGA array from being used for other research projects [96].

The cost of industrial solutions puts them outside the grasp of academic projects. Furthermore, the industrial FPGA arrays are more focused on chip development and not full system development. Using the FPGA array requires the complete Verilog register transfer logic (RTL) of a given design, which increases the cost and time required to use these FPGA arrays. Software is provided with the FPGA array that partitions the complete RTL design across multiple FPGAs. After the design has been partitioned and mapped to the FPGAs, the resulting system runs at tens of kilohertz as opposed to tens or hundreds of megahertz. This platform is sufficient for small diagnostic programs, but presents the same tedious software development environment available using highly

detailed software simulators. Thus, this design framework slowly validates the design, but using FPGA arrays cannot enable software development because of the slow operating frequency.

Industry now provides an affordable FPGA prototyping platform based upon a single FPGA. A plethora of single FPGA prototyping boards can be used for class projects and small research projects [6, 115]. As these prototype boards continue to improve their capabilities, this enables initial prototyping of academic research [23, 75]. These prototyping boards, like the ML310 [115], provide an FPGA with several interfaces for implementing a wide array of digital devices. The latest prototyping boards include multiple I/O ports that can be used to build large FPGA prototyping fabrics or process a wide variety of data. Until recently, these prototyping boards had very limited resources that could implement simple 16-bit or 32-bit processors. However, the latest prototyping boards come with embedded hard processor cores that can run real operating systems like Linux, out-of-the-box. The combined hardware and software broadens the applicability of these boards, making them a great initial platform for research projects [23, 75].

The BEE2 PCB, initially designed for ASIC and DSP research [21], is an excellent example of a prototyping platform using the next generation of hardware available after the FAST 1.0 PCB. A detailed comparison of FAST 1.0, BEE2, and a proposed FAST 2.0 is provided in Section 6.4.

The Research Accelerator for MultiProcessing (RAMP) project is a clear successor to the FAST project that also validates the FAST concept and methodology [11]. RAMP is a consortium of schools that are using the BEE2 PCB to develop the software infrastructure for prototyping a variety of systems. This project faces the same financial hurdles that face any project trying to distribute a system that is not free, but they also are trying to conquer the software infrastructure problem by collaborating and creating an open source like community for the hardware infrastructure. The combination of hardware and software continues to be essential for the next generation prototyping platform. The hardware design should be flexible, but the software infrastructure is crucial for wide adoption.

Recently, there has been an increase in academic interest for computer architecture research using FPGAs, as demonstrated by the Workshop on Architecture Research using FPGA Platforms [1]. These workshops focus on computer architecture research that is enabled by FPGAs. Many novel approaches have been proposed at the workshop, including the RAMP project [11].

Computer architecture research will continue to incorporate hardware prototypes given the increased capacity of FPGAs and the ability to leverage both hardware and software infrastructure. The ability of configurable hardware causes a shift from historical practices of building one-off prototypes to utilizing the same hardware prototyping platform and changing the configurable logic to implement and explore new systems. This amortizes the hardware and software development and cost across multiple projects and enables both hardware and software infrastructure leverage. By developing the hardware and software community, computer architecture research can once again validate ideas with working hardware, hardware that is easier and cheaper to build. Furthermore, the hardware enables more thorough system and software investigation and software development and tuning, completing the once broken research cycle.

Computer architecture research requires a methodology to validate new ideas and designs. Implementing a design is the final validation step that reveals all the design trade-offs. The birth and early maturation of reconfigurable logic enabled hardware implementations to validate and probe around a single design point using real hardware. Now, FPGAs have reached the point where they can implement multiple designs given an intelligent framework. RPM, FAST, and BEE2 have demonstrated the ability to build working, configurable prototyping platforms. The RAMP project validates this methodology and has already demonstrated that multiple designs can be mapped to the same hardware substrate [11, 75]. The FAST 2.0 PCB takes this one step further, describing the next generation hardware prototyping platform that extends the goal of reintroducing hardware back into the research cycle for a broader class of architectures.

2.7 FAST vision

The FAST architecture is a flexible prototyping substrate for TLP architecture evaluation. By building a flexible hardware platform and coupling that platform with a library of software components, FAST can be a prototyping platform that brings hardware back into the research cycle. Past computer architecture projects used hardware prototypes at the end of the project to validate that idea. Increasing hardware costs prevent current and future computer architecture projects from incorporating one-off hardware prototypes in the research cycle. Fundamentally, FAST or follow on systems like it have the ability to bring hardware back into the research cycle much earlier than previous hardware prototyping systems at a much lower cost.

The FAST architecture provides a flexible substrate using the FPGAs in an intelligent interconnect. The FPGAs used in the FAST architecture enable it to prototype many TLP architectures. By configuring the interconnect to match the requirements of the TLP architecture, FAST can morph into that TLP architecture and implement or emulate the TLP memory system combined with simple processors. A mature FAST system has the added benefit of being reusable across multiple designs because the interconnect and logic can be reconfigured. FAST also is scalable, giving it the ability to build larger systems. FAST ushers in a new era of computer architecture research that includes hardware in the research cycle much earlier and enables infrastructure reuse across multiple projects. This amortization may not be realized with FAST, but FAST demonstrates this direction for future computer architecture and computer systems research.

Chapter 3

FAST PCB implementation

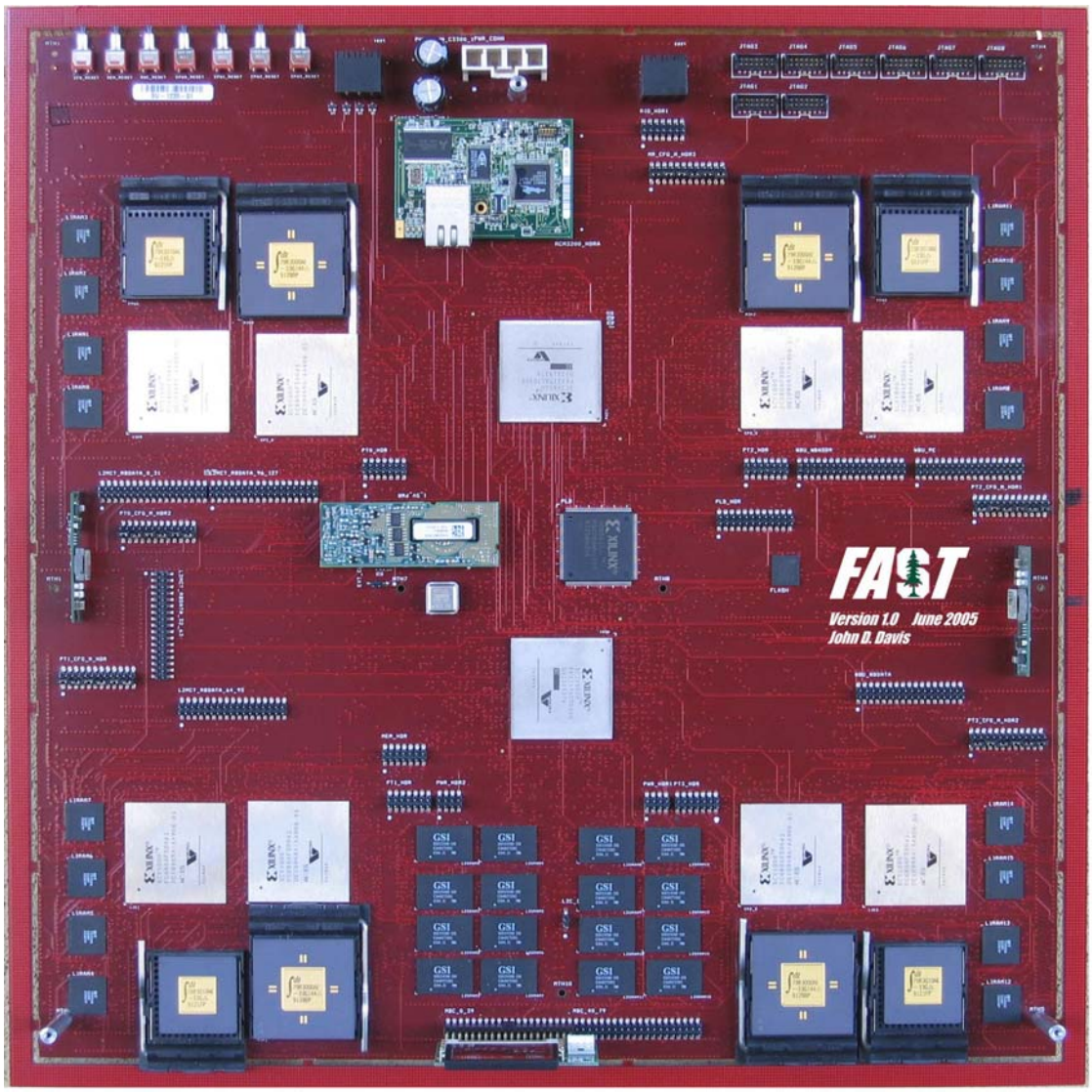


Figure 3.1. Fully assembled FAST PCB.

The previous chapter describes the FAST architecture, as well as the kinds of architectures that map well to the FAST platform. We start this chapter with a picture of the actual FAST printed circuit board (PCB), shown in Figure 3.1, and present the PCB implementation details that result from instantiating the FAST architecture. As described in Chapter 2, FAST is a single PCB system comprised of four replicated processor tiles

and an associated top-level interconnect. Multiple FAST PCBs can be connected together, creating a larger FAST compute fabric. Each processor tile contains: an integer and a floating point datapath, over 1 MB of processor local memory, and two FPGAs that manage the processor tile resources and facilitate reconfiguration. The top-level interconnect is composed of two larger FPGAs to allow communication between the four processor tiles. These top level FPGAs also manage several shared resources, including on/off-PCB I/O via Ethernet and TCP/IP, an expansion connector, 64 MB of second level memory, on-board Flash memory, and hardware to manage these resources and facilitate reconfiguration.

The FAST PCB was conceived, designed, implemented, and tested at Stanford University. The PCB manufacturing and assembly was the only outsourced process, done by Sanmina Corporation's small volume PCB prototyping facility.

This concise high-level overview of the FAST PCB gives some hints as to FAST's possibilities. This chapter starts off by presenting the component selection process, followed by a top-down description of the FAST PCB implementation. The PCB implementation is discussed first because of the proximity of this section to the FAST architecture described in Chapter 2. This section is followed with a discussion of the hardware limitations and how they evolve with silicon technology. The PCB design process, describing how to take a concept and build a PCB follows. Finally, we focus on the fully functional FAST PCB and describe its capabilities.

3.1 FAST component selection

The components must be selected before starting the schematic because the component connectivity, voltage, and spacing requirements are all determined by the component and its package. There is often some flexibility in terms of the part's core voltage and I/O voltage as well as the type of package and number of package pins. The FAST PCB uses 35 unique components, but there are 4260 total components required for each PCB. Table 3.1 shows the main FAST components, quantities, maximum operating frequencies, and core voltages.

Table 3.1. FAST primary components.

Component	Quantity	Maximum Frequency (MHz)	Core Voltage (V)
XC2V6000 [111]	2	400	1.5
XCV1000 [101]	8	200	2.5
L1 Memory SRAM [53]	16	100	3.3
L2 Memory SRAM [41]	16	200	3.3
Ethernet Module [79]	1	44	3.3
CPLD [127]	1	100	3.3
Flash Memory [2]	1	10	3.3
MIPS R3000, R3010 [54]	4,4	33	5

The full parts list and related details can be found in Appendix A. For each component, the package determines how many control, power and ground, and I/O pins exist. Unlike other components, FPGAs come in a rich variety of packages. FAST uses the FPGA packages with the largest number of I/O pins.

Component selection was one of the most difficult parts of the first step after defining the FAST architecture. The architecture was based on the single-chip Hydra design, but was then extended in order to map multiple architectures to a single PCB platform. However, by changing any design from a single chip to a component-based PCB design required splitting buses, using components based on various trade-offs, and compromising some requirements for feasibility purposes. This is required because on-chip properties, like copious amounts of bandwidth, cannot be replicated when single-chip designs are redesigned and partitioned across multiple components of a PCB.

The key advantage of the FAST hardware was its ability to leverage the hardware components: FPGAs, SRAMs, and dedicated processors [41, 53, 54, 101, 111]. The FPGAs provide the connectivity and logic flexibility. FPGAs have moved beyond the point of implementing simple glue logic, by virtue of very high-density silicon technology. The FPGAs allow the user to configure the interconnect and complex logic (interfaces and controllers) required to implement different architectures beyond Hydra. The main drawbacks of using FPGAs are their lack of on-chip memory and their package pin limitations or chip bandwidth limitations. Even the next generation FPGAs from Xilinx or Altera only have a maximum of about 1 MB of on-chip memory [8, 126]. Therefore,

FAST uses SRAMs to provide fast, large storage that is not available in the FPGAs. This frees the FPGA memory for use as small, fast supplemental storage elements, instead of multi-way, multi-ported caches or other large memory structures.

The Processor

FAST focuses on prototyping thread-level parallel (TLP) or other novel memory and system-level architectures, not on novel ISA or processor pipelines. This eliminates processor design from the prototyping system and allows us to use a simple processor across this entire architecture space. However, FAST required a dedicated processor that had exposed interfaces for the caches and coprocessors. FPGA densities could support software-defined processor cores at very slow frequencies, but such cores suffer from poorly defined external cache interfaces and non-existent coprocessor interfaces. A variety of hard processors existed at the time of conception, but none provided access or significant on-chip visibility. Moving back a decade or so to an era when computer systems were built with multiple discrete components and accelerators, we found the MIPS R3000 family of processors [54].

The MIPS R3000 CPU and R3010 FPU provided the most benefit compared to all available processors at design time. No floating-point instruction emulation is required, because of the CPU and FPU combination. The R3000 also has a predefined co-processor interface, which enables additional functionality like Hydra's CP2 or ISA extensions by overloading or redefining pre-defined coprocessor instructions. Finally, the MIPS R3000 has an exposed cache interface that makes all memory traffic visible down to the processor core. This enables the FPGAs to implement any memory system from the processor core's L1 cache out to the higher levels because the processor core is memory system agnostic. We "borrowed" surplus R3000's and R3010's from the Stanford DASH project [66] and used zero-insertion force (ZIF) sockets to hold the CPU and FPU for FAST's processor tiles.

The Processor FPGA

The selection of the MIPS R3000 impacted the rest of the FAST implementation. The R3000 and R3010 use 5 V power. At FAST design time, no contemporary FPGAs had 5

V tolerant I/O's. This presented an implementation challenge with respect to the Processor FPGA. There were two options: (1) use current generation FPGAs and place level shifters between the FPGA and CPU and FPU or (2) use previous generation FPGAs that are 5 V tolerant to interface to the R3000's and R3010's. Adding level shifters would have increased the complexity of the FAST design by increasing the parts count and potentially impacting performance because of the delay introduced by the level shifting transitions. Using a previous generation FPGA removed the need for level shifters, but these FPGAs are more resource limited than the current generation FPGAs. Therefore, FAST uses two previous generation FPGAs, instead of one current generation FPGA with level shifters, in each processor tile. The Processor FPGA has two distinct operations: local memory controller, e.g. cache controller, and coprocessor interface, e.g. Hydra's CP2. These distinct roles map well to the two FPGAs used in the processor tile. The additional integration and partitioning effort is minimized without compromising performance or correctness, as would have happened with a delay element (the level shifters) in the data bus and control signals.

The Hub FPGA

Like the Processor FPGA, the Hub FPGA had two distinct functions, shared memory controller and interprocessor communication. Thus, we split the Hub FPGA into two XC2V6000 FPGAs to provide similar functionality and increase the amount of logic, reducing the resource constraints. In both cases, for the Hub and Processor FPGAs, the implementation required two reconfigurable components, even though the initial architecture specified one. Furthermore, parts availability and lead time was also a concern with some of the newest parts. We intentionally selected available parts instead of selecting and waiting on parts that were never going to be available, e.g. Virtex II Pro FPGAs with 4 embedded Power PC hard processor cores.

The Support Components

Once the main components were selected, datasheets and application engineers provided all the necessary information for the support components. We categorize the decoupling capacitors and pull-up or pull-down resistors as *support components*. The support components are crucial because without these components, the system will not function.

There are about 2000 capacitors and resistors required for the design. The FPGAs required a decoupling capacitor, (across multiple decades, e.g., 10 μF, 1μF, 0.1 μF, and 0.01 μf) per power and ground pin pair. The SRAMs and MIPS parts require far fewer capacitors. Finally, because the FPGAs and SRAMs are ball grid array (BGA) packages that have no pin access for probes, we added about 2000 test points and headers to provide full pin visibility in case we were not able to use the JTAG ports to observe transitions or wanted finer observation granularity provided by a logic analyzer.

3.2 FAST implementation details

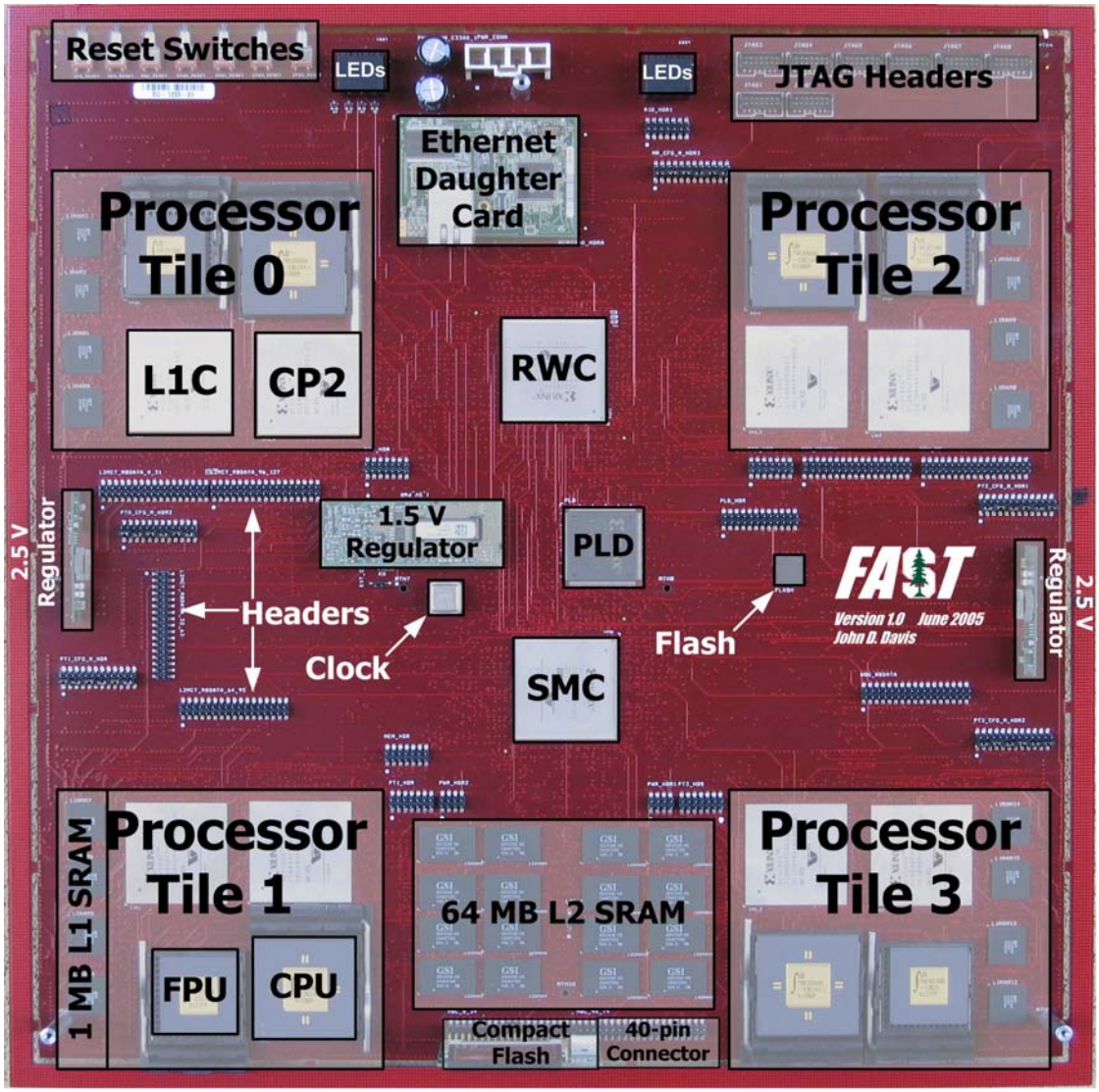


Figure 3.2. FAST PCB labeled with some of the top-level details.

Chapter 2 described the FAST architecture in general detail. Given the component selection process, that actual FAST implementation can be discussed before describing the PCB development process. In this section, the FAST implementation details are presented and the differences in initial architecture versus the implementation are highlighted. First, this chapter starts with a high-level overview of the FAST PCB and then provides further details. For brevity purposes, the complete FAST PCB connectivity and implementation details can be found in Appendix A. However, this section provides enough detail to determine what other designs might feasibly map to the FAST PCB.

FAST uses four different generations of chips requiring four different power supplies. The oldest chips, the MIPS R3000's and R3010's, use 5 V. The Xilinx XCV1000's use a core voltage of 2.5 V for internal logic and 3.3 V for all I/O. The SRAM chips use 3.3 V. The newest chips on the FAST PCB, the Xilinx XC2V6000's, use a core voltage of 1.5V and 3.3 V for all I/O. External power supplies provide 5 V and 3.3 V, while the 2.5 V and 1.5 V are locally generated by DC-to-DC voltage regulators using the 5 V input. There are two power headers on the PCB to supply all four voltages for reference or to use for auxiliary components or daughter cards. Figure 3.2 labels the assembled FAST PCB with all the main components.

The Hub FPGA is comprised of two FPGAs, the read/write controller (RWC) and the shared memory controller (SMC). The RWC coordinates the interprocessor communication, arbitration for the higher-level memory, level 2 (L2) and beyond, and FAST I/O. The SMC FPGA provides access to the 64 MB, plus parity, of L2 memory and the 80-pin expansion port. The Processor FPGA is comprised of two FPGAs as well, the coprocessor 2 (CP2) and the level 1 memory controller (L1C). The CP2 FPGA provides the system interface to the MIPS R3000, MIPS coprocessor interface, the interface to the RWC FPGA, and additional interprocessor tile communication. The L1C FPGA provides the memory interface between the MIPS R3000, coprocessors, and the 1 MB, plus parity, of L1 SRAMs.

The FAST PCB also has a variety of miscellaneous components for FAST I/O, PCB management, transparency, and clock distribution. There is an embedded Ethernet

daughter card that provides FAST I/O. The FAST PCB also includes a programmable logic device (PLD) that retains its programming even with the power off. The PLD provides the core PCB management. The PLD and the Flash memory can be used to store FPGA programming files and the PCB operating system (OS). Thus, given a programmed PLD and the correct contents in Flash memory, the FAST PCB can be powered on, booted up, and ready to run applications. The PLD can also be used to control the PCB state, changing it from FPGA programming mode, to running or debugging user applications. The PLD also has two single-ended clock inputs. The standard clock input comes from a half-size clock oscillator that fits into a 4-pin socket. There is also an external clock header that can connect to a frequency generator or other external clock generator. Finally, various headers and test points provide system transparency for all chip I/O signals. Figure 3.2 highlights the top-level features of the FAST PCB.

3.2.1 Hub FPGA

From Figure 3.2, it is clear that the Hub FPGA is not a single FPGA. Instead, it is two FPGAs, the read/write controller (RWC) and the shared memory controller (SMC). The Hub FPGA is implemented as two FPGAs to increase the available system gates on the PCB and also provide more I/O pins. Increasing the number of system gates from 8 million using a single FPGA to 12 million using two slightly smaller FPGAs increases the capabilities of the FAST hardware by more than 50%. By using two FPGAs instead of one, we are able to create more logic and utilize more I/O pins for increased integration.

Looking back at the Hydra specification combined with the FAST architecture, a single FPGA was too pin limited and could not have connected to all four processors, provide the read and write buses in the Hydra design, and connect to a shared memory. The functionality of the RWC and SMC was a clear bifurcation point, which further reinforced the use of two FPGAs. The RWC and SMC FPGAs provide over 1100 I/O pins, a maximum operating clock frequency of 400 MHz, and 6 million system gates each. These I/O pins are quickly exhausted when mapping the FAST architecture to a single FPGA. Appendix B provides a link to an online archive containing the user constraint

files (UCF) that fully specify how each I/O pin is used on the FAST PCB for all the components.

RWC

The Read/Write Controller (RWC) handles memory hierarchy events that propagate past the primary caches, such as write-throughs and cache misses. A wide bus permits the observation of memory traffic to and from all processor tiles on a cycle-by-cycle basis, making it possible to implement cache coherence protocols using snooping on primary cache contents in other processor tiles. This controller could also be used for inter-processor tile messaging in systems that do not use traditional memory coherence [27].

Table 3.2. FAST RWC primary connectivity.

FAST Components	Connection	Width
RWC to SMC	Point-to-Point	408
RWC to each CP2	Point-to-Point	4x140
RWC to all CP2's	Shared	44
RWC to PLD	Point-to-Point	35
RWC to all L1C's & all CP2's	Shared	33
RWC to RCM3200 & SMC	Shared	28

The RWC connectivity details are listed in Table 3.2. The majority of connections exist between the RWC and SMC with 408 pins between the two FPGAs and the CP2s with 560 pins. Appendix B also points to the UCF files that map Hydra to FAST. These files give a better understanding of the potential pin mappings and how other architectures may map to the FAST PCB. Table 3.2 is an abbreviated list of connections to the RWC.

The very wide RWC to SMC bus can be used for a variety of purposes: a low latency, high bandwidth connection to L2 memory, various control signals, and access to higher levels of memory, like a Compact Flash daughter card, or off PCB I/O. The RWC connectivity also illustrates the other important RWC function, processor tile intercommunication. Each processor tile allocates 140 pins of the CP2 FPGA to communicate with the RWC FPGA. This connectivity is also illustrated in Figure 3.3.

The RWC FPGA also has dedicated point-to-point and shared buses to the PLD, the processor tiles, and the Ethernet daughter card (RCM3200) [79]. The PLD provides

access to the Flash memory and the PCB management and clock distribution. There is also a shared bus between the RWC and all of the processor tile FPGAs. This shared bus can be used for global communication or broadcasts, synchronization, or time-division multiplexed point-to-point communication. The RCM3200 provides a TCP/IP interface to the FAST PCB. This RCM3200 interface is shared with the SMC and there are a few point-to-point control pins that are not specified in Table 3.2. Furthermore, several low-level connectivity details have been left out of Table 3.2 and Figure 3.3. The RWC is connected to four LEDs that can be used for visual debugging or other types of indicators. The initial test RWC program incremented a counter whose high order bits controlled the LEDs. The LEDs thus indicated a successfully programmed FPGA. The RWC FPGA also has a hardwired unique ID. This can be read by modules in the FPGA to differentiate the RWC FPGA from any other FPGA on the FAST PCB. There is also a global and local reset pin that can be used as the system requires and a globally distributed clock signal. The complete connectivity and an example pin mappings are provided in the UCF files pointed to by Appendix B.

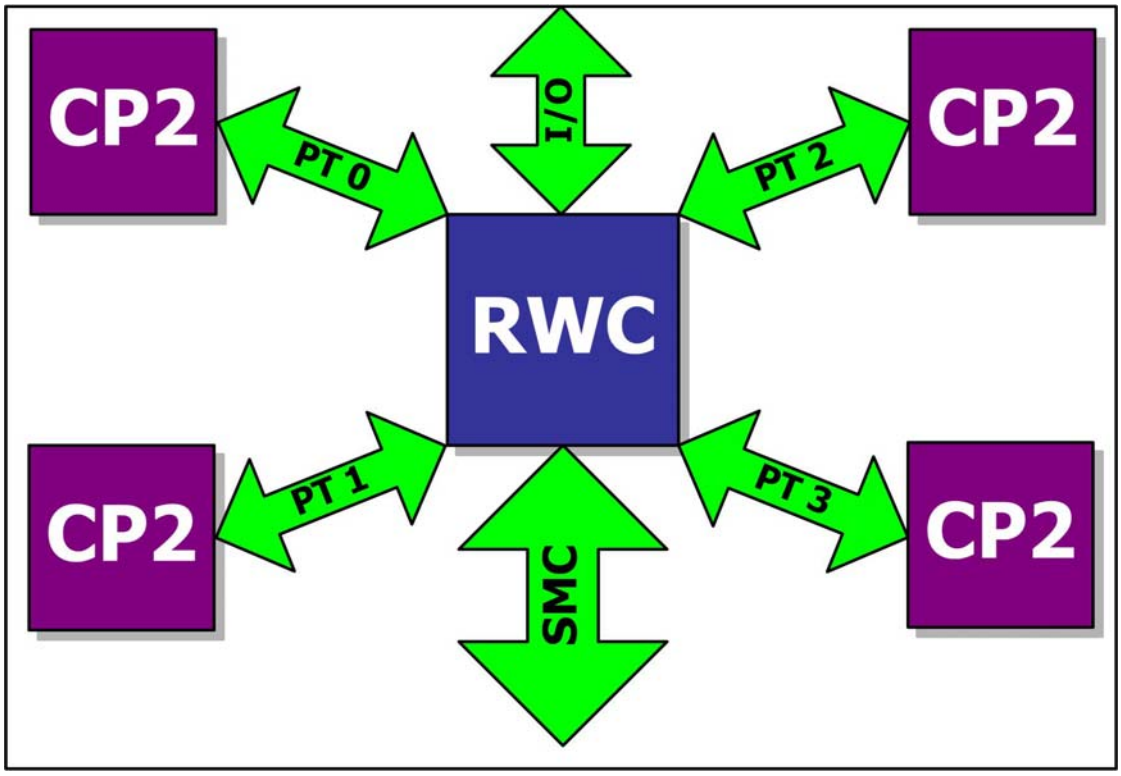


Figure 3.3. High-level RWC connectivity.

SMC

The Shared Memory Controller (SMC) manages the 16M x 36 bit (including 4 parity bits) of secondary memory. These synchronous SRAMs can be configured as a secondary cache or general-purpose, off-chip memory. The entire memory can be shared by all processor tiles or segmented, e.g., into 4M x 36 bit private partitions assigned to each processor tile. Furthermore, time-division multiplexing can be utilized to implement various set associative cache configurations. Table 3.3 specifies the rest of the major SMC FAST PCB connectivity. To reduce repetition, Table 3.3 specifies how many connections exist followed by the connection width. For example, there are four SRAM banks and each bank has a 72 bit bus between the SMC FPGA and the SRAM bank.

The SMC controls an 80-pin expansion connector. This multi-purpose header can be used to connect multiple FAST prototyping substrates together to create a larger FAST emulation fabric, to add daughter cards, or to attach additional memory, such as a DRAM main memory bank or Compact Flash daughter cards. This 80-pin connector can have two 40-pin IDE interfaces connected simultaneously, used for either a Compact Flash or a hard drive. The SMC FPGA also connects to the PLD, which provides access to the Flash memory. Finally, the SMC shares a 16-bit data bus and a few control signals with the RWC and the RCM3200, the embedded Ethernet module. This connectivity provides flexible access to I/O on and off the FAST PCB.

Table 3.3. FAST SMC primary connectivity.

FAST Components	Connection	Width
SMC to RWC	Point-to-Point	408
SMC to Expansion Port	Point-to-Point	80
SMC to each SRAM Bank	Point-to-Point	4x72
SMC to each L1C & CP2	Shared	4x45
SMC to each CP2	Point-to-Point	4x16
SMC to PLD	Point-to-Point	35
SMC to RCM3200 & RWC	Shared	28
SMC to all CP2s	Shared	19

Figure 3.4 illustrates the high-level connectivity from the SMC to the rest of the FAST PCB. This figure only illustrates the connectivity of a single SRAM bank because the other three are the same. Each SRAM bank contains four SRAM chips providing over

four MB of memory. The SRAMs in a bank share the address and data pins, forcing a single read or write to occur, or reads and writes to all SRAMs at once, or some subset depending on the SRAM chip enables and control signals. Time-division multiplexing can be used to sequentially access multiple SRAMs in a single system clock cycle. The L2 SRAMs are single ported, but can operate at up to 200 MHz, much faster than the MIPS components' maximum system clock speed of 33 MHz.

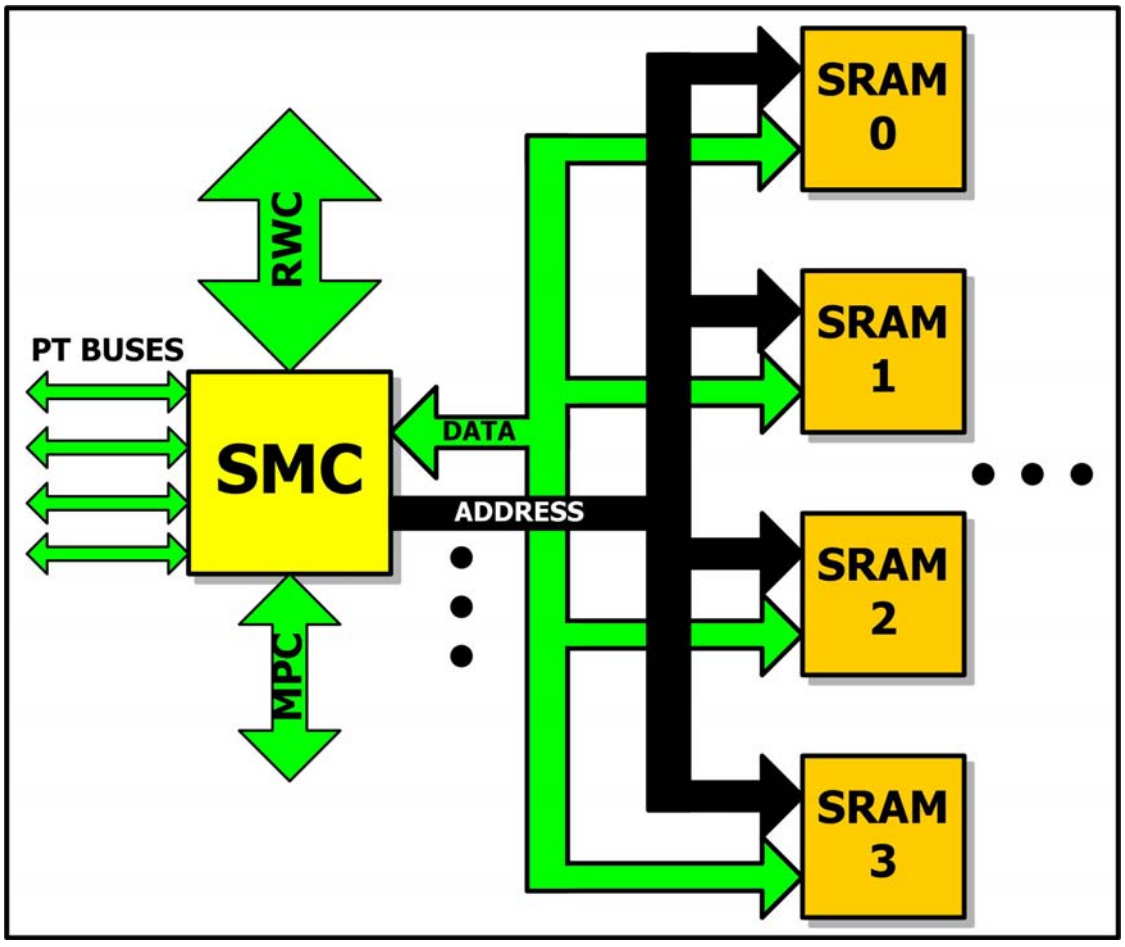


Figure 3.4. High-level SMC connectivity with a single SRAM bank.

The SMC is also connected by a shared bus to the two FPGAs in each processor tile. There is an additional point-to-point bus between the SMC and each CP2 FPGA in every processor tile. This pair of shared and point-to-point buses enable the processor tiles to bypass the RWC FPGA to communicate directly to the SMC and L2 memory. This direct connection to each processor tile provides another level of interconnectivity flexibility. Thus, it is not required to send memory traffic through the RWC, if high bandwidth is not required. This direct connection also reduces the latency, in addition to

the ability to "over clock" the SRAMs and FPGAs with respect to the global system clock.

3.2.2 Processor FPGA

The processor tile is one of the novel aspects of the FAST PCB. The processor tile integrates a dedicated processor with configurable logic and memory. The Processor FPGA defined in the architecture has been implemented with two FPGAs in the processor tile. Two FPGAs are used to reduce the integration complexity due to the high 5 V output voltage of the MIPS components, as explained earlier. Furthermore, each FPGA in the processor tile serves a clear non-overlapping purpose. The clean bifurcation made functional separation very easy. The hardwired processor in the tile enables rapid prototyping because there is no processor design required, only memory system design. The FAST PCB processor tile also provides the capability to create a variety of memory subsystems by inserting an FPGA between the MIPS components and the L1 SRAMs. Thus, the L1 controller (L1C) FPGA can define different interfaces between the MIPS processor and the SRAMs, and also can create the translation mechanisms to service memory or cache requests. Thus, the L1C can *fake* the normal L1 cache interface for the MIPS components, while simultaneously using a completely different memory structure resident in the L1 SRAMs. The other FPGA, the coprocessor 2 (CP2), can provide additional functionality.

Table 3.4. FAST CP2 primary connectivity.

FAST Components	Connection	Width
CP2 to RWC	Point-to-Point	140
CP2 to L1C & CPU	Shared	54
CP2 to L1C & SMC	Shared	45
All CP2s to RWC	Shared	44
CP2 to L1C, CPU, & FPU	Shared	43
All CP2s TO all L1Cs & RWC	Shared	33
All CP2s to SMC	Shared	19
All CP2s	Shared	18
CP2 to CP2	Point-to-Point	6x18
CP2 to SMC	Point-to-Point	16
CP2 to L1C	Point-to-Point	14

Each processor tile exploits the flexibility provided by its two Xilinx XCV1000 FPGAs. The main XCV1000 FPGA is the CP2 FPGA that can be used to add instructions to the MIPS ISA, add new compute engines, maintain statistics counters, or facilitate interprocessor communication. Figure 3.5 illustrates the connectivity of the CP2 FPGA defined in Table 3.4. Table 3.4 does not enumerate all six CP2 to CP2 connections that are required for full point-to-point connectivity nor does it list all the connections to the CP2. The MIPS-I ISA has a well-defined coprocessor interface that can be exploited by this FPGA to add instructions to implement software control over cache coherence protocols, additional functional units, or other features. MIPS defines the FPU as coprocessor 1, thus the FPGA follows as coprocessor 2.

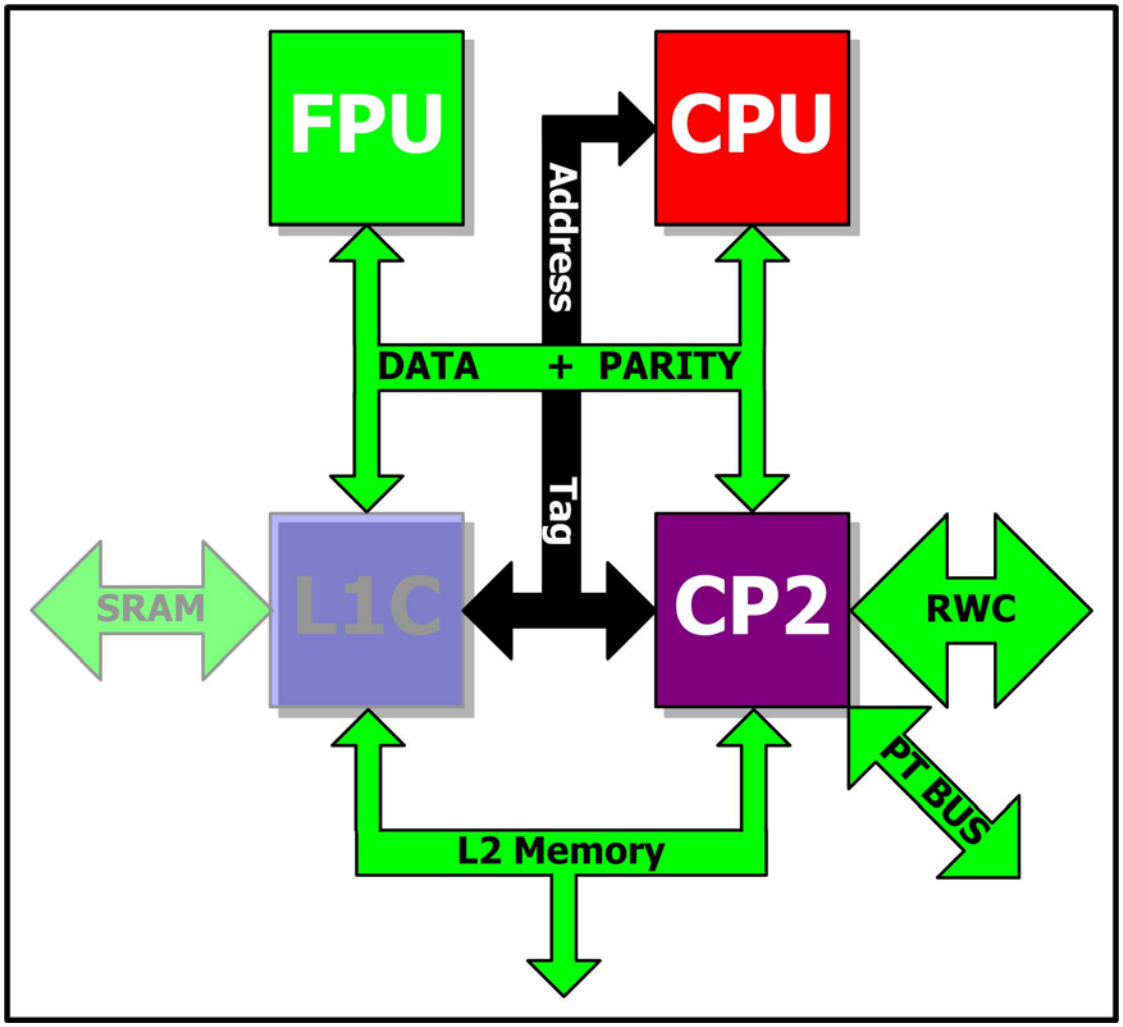


Figure 3.5. High-level CP2 connectivity.

The largest processor tile interface travels between the CP2 and RWC. This interface handles the higher-level memory traffic and interprocessor tile communication. The CP2 can also utilize either point-to-point or broadcast information to the processor tiles using the PT Bus, a secondary means of interprocessor tile communication. In general, there are two control bits and 16 data bits defined for the direct interprocessor tile communication. The address, tag, and data are distributed to all the components (CPU, FPU, L1C, and CP2) in the processor tile as inputs, outputs, or both. The CPU generates addresses, which then can be processed by the cache system controlled by the L1C FPGA or the memory system in the CP2 FPGA. The FPU or CP2 FPGA can also process data and coprocessor instructions. The CP2 can also directly access the SMC FPGA using the shared L2 Memory bus.

The L1C FPGA manages the 256K x 36 bit (including 4 parity bits), dual-ported local processor tile memory. The L1C serves as the MIPS R3000's external cache interface, enabling a wide range of virtual cache configurations, while the morphware fakes the expected direct-mapped cache behavior. Like the SMC, the L1C can utilize time-division multiplexing to implement set associative caches or other types of memory systems. Furthermore, the CPU and FPU always access the local memory through the L1C, giving it the ability to modify the instruction and data stream on-the-fly, if necessary. Table 3.5 provides the high-level connectivity of the L1C.

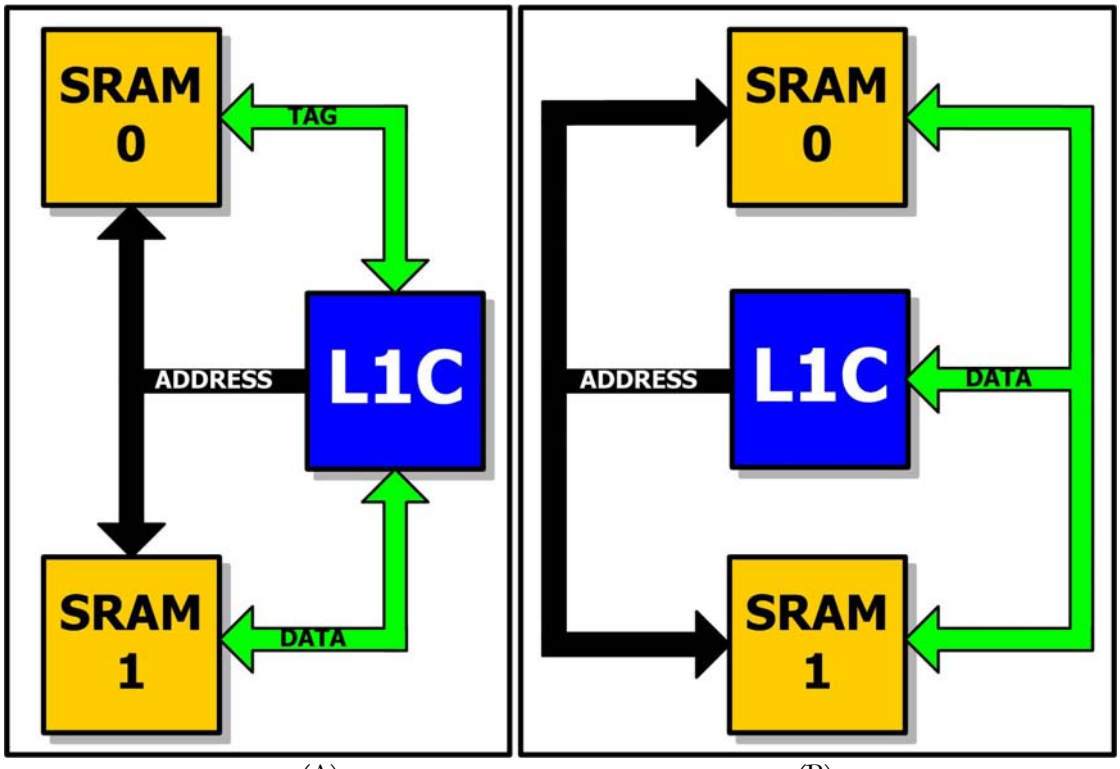
Table 3.5. FAST L1C primary connectivity.

FAST Components	Connection	Width
L1C to SRAM0 & SRAM1	Shared	68
L1C to SRAM2 & SRAM3	Shared	68
L1C to CP2 & CPU	Shared	54
L1C to CP2, CPU, & FPU	Shared	43
L1C to CP2 & SMC	Shared	45
All L1Cs TO All CP2s & RWC	Shared	33
L1C to each SRAM	Point-to-Point	4x40
L1C to CP2	Point-to-Point	14

Besides the L1C interface to the CPU, FPU, and CP2, the L1C's main function is providing an interface to the L1 SRAMs. Pin limitations inhibit the flexibility of the memory system by restricting the L1C FPGA to SRAM connectivity. The SRAM

connectivity is set up for the canonical instruction and data cache structures. The left SRAM port provides this interface shown in Figure 3.6 A. The L1C provides a single address to the SRAMs, in this case SRAM 0 holds the tag array and SRAM 1 holds the data array. The L1C can then read or write the data and tag arrays simultaneously. Only two SRAM chips are shown in Figure 3.6 A and B because the interface is the same for the other pair of SRAM chips. One pair services the data cache and the other pair services the instruction cache.

Figure 3.6 B illustrates the right port of the SRAMs. This port can be used for snooping and cache invalidation. Due to the pin limitations of the L1C FPGA, only one of the two SRAM chips can be accessed at a time because these two chips must share the same data bus, as shown in Figure 3.6 B. However, time-division multiplexing can be used to access both chips in a single system clock cycle. Originally, FAST specified independent address and data buses to each SRAM, but the implementation limitations prevented a fully flexibly SRAM interconnect.



(A) (B)

Figure 3.6. (A) L1C left port SRAM interface and (B) L1C right port SRAM interface.

The ability to leverage pre-existing hardware and software reduces the development time for future projects. The FAST processor tile uses a dedicated processor core to reduce the design effort by leveraging the design, validation, and functionality of the established processor, coprocessor interface and external cache interface. Also, limited processor design is feasible using FAST by manipulating the number of MIPS clock cycles in a system cycle or using other creative techniques coupled with accurate performance counters.

However, another processor alternative has emerged over the span of the FAST project. Several software-defined processor cores or soft cores can be licensed and loaded into the FPGA as an alternative to hardware processors. This software-based solution is an attractive alternative as it matures. Currently, a majority of the soft cores are still in their nascent state. These soft cores do not support modern operating systems and require application retargeting. At the minimum, these applications require recompilation. However, soft cores will dominate future platforms because of their additional flexibility and density.

Some FPGAs also have integrated embedded hard processor cores. These efficient compute engines are optimal for applications that do not alter any part of the memory system. If a new memory system, especially one that modifies the processor cache, is required, these hard embedded cores currently must use awkward processor interface buses.

3.2.3 PLD

The programmable logic device (PLD) [127] and associated Flash memory [2] are the system bootstrapping and monitoring devices. The PLD can program the FPGAs using the Xilinx 8-bit parallel programming ports. There are 8 JTAG groups or zones on the PCB as well, to provide a secondary system programming and debugging port. The PLD also acts as the memory controller for the Flash memory chip. This 16M x 8 bit Flash chip has the capacity to hold all 10 unique FPGA configurations with approximately 5 MB remaining for storing bootstrap software code right on the board [2, 101, 111]. The PLD also controls the overall state of the board, choosing among FPGA configuration,

application execution, application debugging, and reset modes. Table 3.6 lists the primary PLD connections on the FAST PCB.

The PLD can communicate directly with the RWC or SMC FPGAs. These point-to-point links are predominantly defined to provide access to the Flash memory. The PLD controls the access to the 16M x 8 bit Flash chip that is used to store the FGPA programs and that has additional memory reserved for PCB bootstrapping, batch OS code, or other non-cacheable address space requirements.

Table 3.6. FAST PLD primary connectivity.

FAST Components	Connection	Width
PLD to RWC	Point-to-Point	35
PLD to SMC	Point-to-Point	35
PLD to Flash	Point-to-Point	38
PLD to RCM3200	Point-to-Point	18
PLD to Header	Point-to-Point	12

The PLD also can communicate directly to the embedded Ethernet controller, the RCM3200. This link provides an interface for the PCB management software to communicate to the RCM3200, thereby using the embedded Ethernet controller to relay information between the PLD and host machine.

The PLD is also responsible for PCB clock distribution. The system clock can be generated on the PCB using a half-size clock oscillator or using a two-pin header for an external frequency generator. All ten clock traces, from the PLD to all the FPGAs, are hand routed, such that all traces are the same length within a 10 mil tolerance.

Finally, the PLD also has an exposed header for adding new functionality not specified at design time, or for observing specific functionality. Figure 3.7 illustrates the high-level connectivity between the PLD and the other FAST PCB components described in Table 3.6.

The AMD Flash memory provides the non-volatile memory required to bootstrap the FAST PCB. Each XCV1000 FPGA requires 6,127,744 configuration bits to program the device [101]. Likewise, each XC2V6000 requires 21,849,504 configuration bits [111]. To store unique configuration bit streams for all FPGAs, 8 XCV1000's and 2 XC2V6000's,

consumes 92,720,960 bits. The Flash memory has 256 64K byte sectors [2]. Each XCV1000's configuration bit stream consumes 12 64 KB sectors and each XC2V6000's configuration bit stream consumes 42 64 KB sectors for a total of 180 sectors for unique FPGA configuration bit streams. This leaves 4,864 KB of Flash memory reserved for bootstrap code, batch operating system, or other non-volatile storage purposes. If the L1C and CP2 processor tile FPGAs use the same configuration bit streams, the amount of Flash memory increases to 9,472 KB.

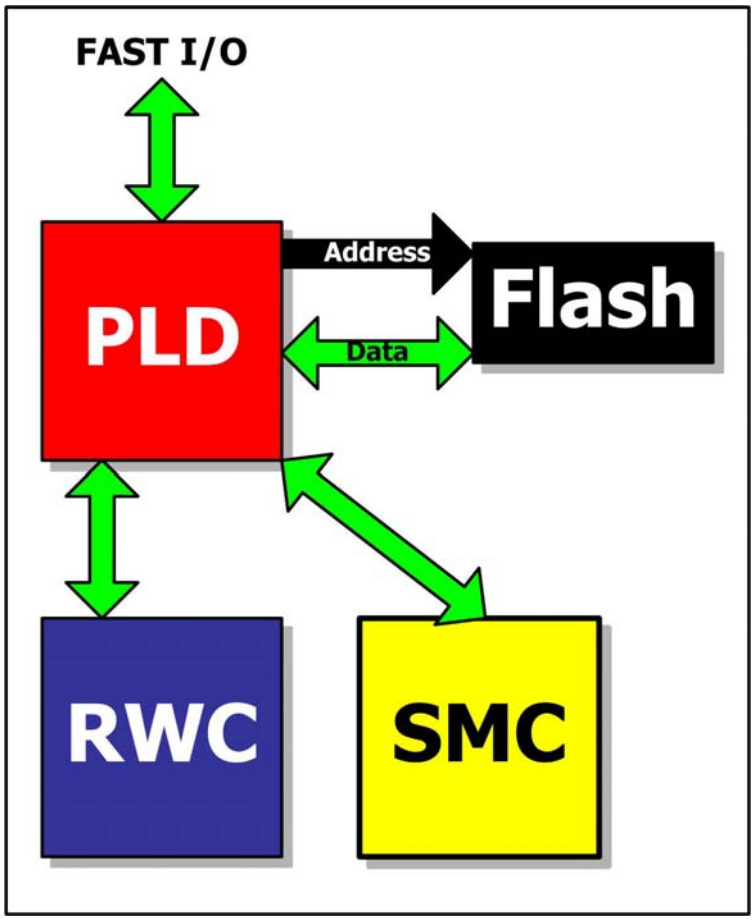


Figure 3.7. High-level PLD connectivity.

3.2.4 Miscellaneous FAST PCB features

As with all complicated projects, there are countless low-level details that impact the design. Some of these details improve functionality, while other details provide back up mechanisms. Several headers and test points were added to the PCB because of the large ball grid array (BGA) components used. The BGA components provide no access to the pins, so the test points were added in case the design requires a logic analyzer for

diagnostics. There are two types of headers, those headers connected to the I/O pins and those headers connected to power and ground. The I/O pin headers and test points provide the transparency or observation available in software simulators. Over 2000 test points and I/O header pins were added to the PCB in order to guarantee transparency at a chip interface level.

Xilinx provides software to do limited monitoring within the FPGA by inserting a monitoring module using ChipScope [113]. This monitoring module could also monitor the I/O pins, but these modules have very limited data collection capacity. Thus, the test points and I/O pin headers enable external data collection between the chips on the PCB. All of the test points and headers are listed in the support files provided in Appendix A with respect to the Hydra design for a design reference. The FAST PCB also has an 80-pin header specifically dedicated for future additional components or new digital interfaces.

Headers also provide power and ground for external uses by daughter cards or voltage references. All four voltages, 5 V, 3.3 V, 2.5 V, and 1.5 V, are supplied to two headers along with ground. These power headers can be used to verify the correct voltage output by the external power supplies and on-PCB DC-to-DC voltage regulators. The power headers can also be used to power external daughter cards, e.g., a Compact Flash, or RS-232 buffer. This is especially useful for add-ons, which require more power than can be supplied by I/O pins. For example, an RS-232 daughter card powered by the 3.3 V and ground power header pins facilitated the data collection presented in Chapter 5.

The FAST PCB provides fine-grain and coarse-grain external interrupts to the FPGAs. Momentarily-on switches can be used to signal a processor tile, SMC, RWC, or the PLD. The initial intent is to use these switches as a reset or restart signal. Furthermore, the PLD also distributes a reset signal to all of the FPGAs for global reset or restart. The headers and test points allow changing inputs or bus conditions, especially for fault tolerance studies, under the same experimental conditions with respect to the FPGA programming. Thus, using the reset switches, multiple experiments can be run by changing the header or test point inputs and resetting the FAST PCB. Targeted reset or

restart can correct errors that occur during an experiment, or can enable multiple experimental runs.

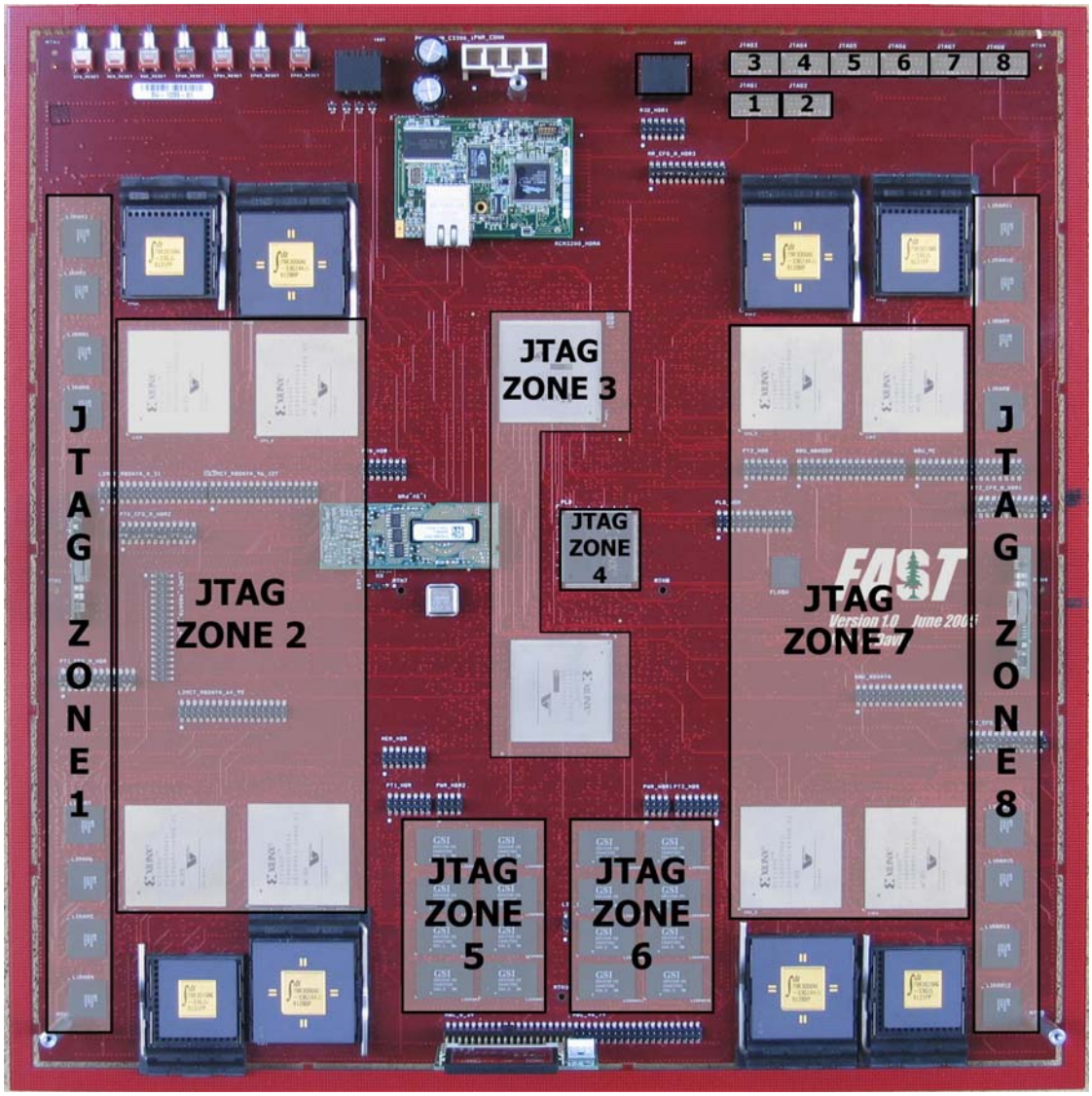


Figure 3.8. FAST JTAG zones and corresponding JTAG headers in the upper right corner.

The FAST PCB incorporates eight JTAG chains as shown in Figure 3.8. These JTAG chains can program the FPGAs and the PLD, perform memory operations, test PCB integrity, and statically observe I/O pin transitions. The FAST PCB is broken into eight JTAG zones to preserve signal integrity in the JTAG chain; making the JTAG chains too long can cause signal integrity issues. The FAST PCB also groups components with

similar functionality together on a JTAG chain. The JTAG zones and the corresponding JTAG headers in the upper right corner are labeled in Figure 3.8.

JTAG zone 1 provides a boundary scan port for the L1 SRAM chips in processor tiles 0 and 1. Likewise, JTAG zone 8 provides a boundary scan port for the L1 SRAM chips in processor tiles 2 and 3. JTAG zones 2 and 7 provide a boundary scan port to the corresponding processor tile FPGAs. The L1C FPGA is programmed before the CP2 FPGA. This means that the processor memory system is set up and initialized before the CP2 FPGA initializes the MIPS R3000. This intelligent JTAG chain layout eliminates the need to stall the processor initialization. JTAG zone 3 provides a boundary scan port to the RWC and SMC FPGAs. JTAG zone 4 provides a boundary scan port to the PLD. The L2 SRAMs are separated into two groups and placed on two separate boundary scan chains or zones, 5 and 6. Each FPGA boundary (JTAG) scan chain has about 2000 boundary scan cells, while the SRAM boundary scan chains have about 600 boundary scan cells.

The JTAG interface can be used to program the Xilinx devices on the PCB. The number of boundary scan cells determines how long it takes to shift serial data in and out of the JTAG or boundary scan chain. Minimizing the boundary scan chain length reduces the programming time and also increases the frequency of static JTAG sampling. The JTAG interface can also be used to program memory devices with JTAG, e.g., all the SRAMs. The normal memory programming facility uses JTAG enabled devices that are connected to memory devices, e.g. using the PLD to program the Flash memory. The JTAG interface can also be used to sample the I/O pin scan cells while the PCB is running experiments. This is a static sampling process because the sampling rate is lower than the system clock frequency. JTAG static sampling is very useful for observing steady state operation. This methodology can also be used to latch events that occur with a low frequency and are hard to visually observe. The JTAG interface can also be used to test PCB integrity during the bring-up process. Using JTAG to verify PCB integrity will be discussed in Section 3.5.

The FAST PCB also has 32 LEDs that can be used for a variety of purposes. Initially, the LEDs were used to indicate that the FPGAs had been programmed correctly. Section

3.5 will show the initial LED test code for the PLD and RWC. The PLD, RWC, and SMC each control four LEDs. Each processor tile controls four LEDs with two LEDs assigned to the CP2 FPGA and two LEDs assigned to the L1C FPGA. The remaining four LEDs are used for power indicators for the four power planes.

Finally, the FAST PCB was designed with the past and future in mind. The combination of old MIPS components with the latest available FPGAs motivated the ability to have replaceable components. For the MIPS components, the FAST PCB uses zero-insertion force (ZIF) sockets for easy replacement of the old parts. The ZIF sockets also allow the FAST PCB to run with or without an FPU depending on the desired operation. Operating the FAST PCB without the FPUs results in about a 5 W power savings. The DC-to-DC voltage regulators can also be replaced if necessary. By using voltage regulator headers, the FAST PCB can use improved efficiency regulators, higher current rated regulators, or even change the voltage output. The latter capability would allow the FAST PCB to upgrade the FPGAs with pin-compatible next generation FPGAs that use different core voltages by swapping the FPGAs and the voltage regulators. The voltage headers also enable limited voltage scaling studies using external power supplies to supply the FPGA core voltages.

3.3 FAST hardware limitations

The FAST PCB leverages the strengths of its base components, but still has some hardware limitations. Unlike VLSI design that uses copious amounts of on-chip bandwidth, the amount of bandwidth on the FAST PCB is limited by the pin constraints of the FPGAs. The designs mapped to the FAST PCB must be partitioned across multiple FPGAs, requiring internal buses to be mapped onto the existing FPGA interconnect. This may force wide on-chip buses to be mapped to much narrower FAST buses that require double or quad data pumping to create wide virtual buses. Currently, users must specify and manage these virtual buses. However, Verilog modules are being developed with these mechanisms, e.g., DRAM interfaces [121]. As mentioned in Section 3.2.2, the pin limitations also restricted the flexibility of the memory system design by forcing shared address and data pins across multiple SRAM chips. Again, time division multiplexing or similar techniques can be used to regain some memory system flexibility,

but this definitely is a limitation. Moving forward, using FPGAs with denser pin packages would help, but that most likely results in removing the MIPS components from the design because of the voltage incompatibilities; the Virtex II FPGAs and beyond are not 5 V tolerant devices.

Similarly, some other structures that are easy to implement in a VLSI environment cannot be translated well into the FPGA environment. One such example is gang-clearing bits in cache structures. Gang clearing can occur in a single cycle in a VLSI implementation. Using FPGAs, gang clearing or gang invalidation requires multiple cycles for normal cache structures. The gang clearing operation time can be reduced by keeping a table of the entries that require this operation, but there is no easy way to reduce this time below ½ the number of entries that require the gang operation, assuming a dual ported memory can be used. Other similar operations that require multiple events to happen for a single structure fall into this category of difficult FPGA implementations. However, the FAST PCB has complete control of the system clock and performance counters. Thus, the appropriate counters and system components can be stalled in order to perform the gang operations and in some cases, the gang or similar operations can be performed in the background while normal operation continues.

Many VLSI designs are adopting a system-on-a-chip (SOC) configuration, integrating multiple components with different clocking domains and interfaces. FPGAs have limited clocking resources making it difficult to generate arbitrary clock frequencies based on a reference system clock. Likewise, FPGAs require dedicated clock resources for all unique asynchronous components. Asynchronous clocks are inferred for capturing edge-triggered events for various performance counters. This limits what can be monitored inside the FPGA. Furthermore, limited memory in the FPGA restricts the amount of information or system tracing that can be stored in the FPGA. Thus, performance or monitoring data must be streamed out of the FPGA for full system traces or monitoring. Gathering system traces is possible, but requires more infrastructure development.

Finally, using the MIPS R3000 is not without its own challenges. The R3000 was selected because of its high speed relative to software-defined cores (at the beginning of the project), transparency, exposed cache and coprocessor interfaces, and integrated FPU.

No software-defined or hardware processor core offered the same capabilities. Even four years after starting the FAST project, software-defined cores are still immature and no commercial general-purpose processor has all the capabilities of the R3000, although modern processors are much faster.

The R3000 uses dual-phase (two-phase) non-overlapping clocking as shown in Figure 3.9. The benefit of this clocking scheme is that you can guarantee the system will work if the clock frequency is slowed down. However, it adds complexity to the clock and processor interface by enabling events to occur in each phase of each clock. Figure 3.9 is a scale drawing of the MIPS R3000 clocking scheme illustrating the two phases and the four edges that occur in each phase.

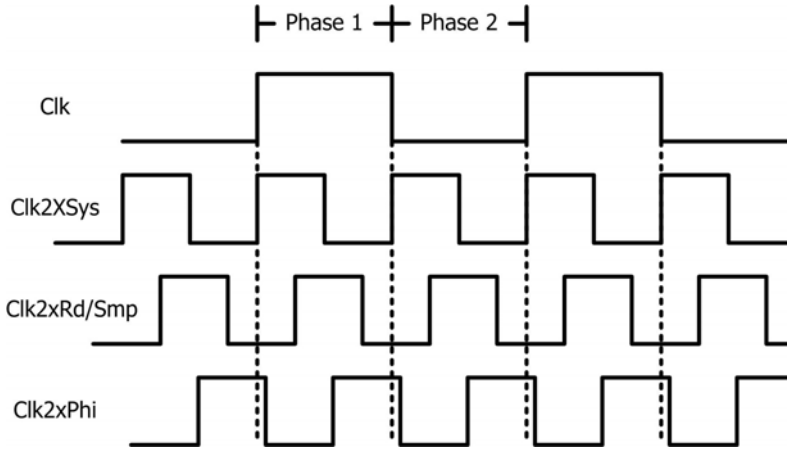


Figure 3.9. MIPS R3000 dual-phase clocking.

Over a single clock cycle, eight different events can occur, making the system and processor integration more difficult due to the precise timing constraints. The Clk2XSys is used to guarantee an even 50% duty cycle for each phase and is the master system clock and first positive edge trigger for events in a phase. The second and third positive clock edges are provided by the read (Clk2XRd) and sample (Clk2XSmp) positive edges that occur in each phase at the same time. Clk2XPhi supplies the fourth positive edge transition. The R3000/R3001 Designer's Guide provides more detail on the clocking behavior and corresponding processor timing dependencies [54]. The processor tile FPGAs not only have to generate the various clocks, but the FPGAs are forced to meet the timing dependencies or the FAST system will not operate. This required explicit buffer placement to create the MIPS clock phase delay.

For the system operating frequencies of interest, FAST uses an independent clock oscillator to generate the system clocking scheme. The PLD distributes the system clock from the clock oscillator to all FPGAs. The CP2 FPGA in each processor tile is responsible for generating all four MIPS clocks. The FPGA generates the MIPS clocks using two delay-locked-loops (DLL's) linked together to generate a 2X clock and a 2X clock phase shifted by 90 degrees. Using a system clock between 16 MHz and 25 MHz requires a delay of 6 ns for the read and sample clocks. Using wire delay in the CP2 FPGA, a pre-routed buffer was placed in the FPGA to produce the desired 6 ns delay. The post placement and route report provides an approximate delay that was also verified using an oscilloscope. The 90-degree phase shifted version of the double frequency clock provided the correct ClkPhi2X signal. As long as the system clock is in the 16-25 MHz range, the FPGA generated clocking meets the IDT clocking specifications [54].

FPGAs enjoy the same benefits of silicon process scaling as the general-purpose commodity processors. Each new generation of FPGAs operates faster and has much larger capacity. However, the device density has far outpaced the package pin growth and the amount of available on-chip memory. The number of pins on the package is a physical constraint. While the FPGA designer determines the amount of on-chip memory, there is a constant struggle between making FPGAs with more on-chip memory and keeping the FPGAs a general-purpose configurable device. This was evident in Xilinx's short-lived "E" line of FPGAs that incorporated far more on-chip memory than normal [103]. Moving forward, the pin limitations and on-chip memory will continue to be a problem. The pin limitations can be addressed by using a limited number of high bandwidth narrow serial links, but the on-chip memory limitations will still exist as a result of design and functionality decisions.

3.4 Building the FAST PCB

This section changes gears and describes the PCB design process. The FAST PCB is the result of seven well-defined steps. Chapter 2 and Section 3.1 provide the process for step one. As Figure 3.10 shows, some of these steps may require some design iteration that makes the entire process much longer than seven steps.

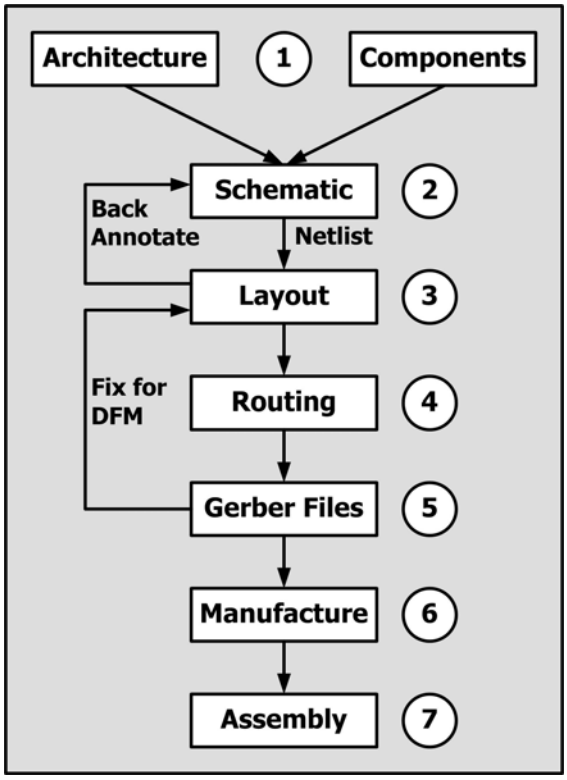


Figure 3.10. PCB production process.

Starting with the pre-defined architecture described in Chapter 2, the other part of the first step in the PCB process starts with parts or component selection, which was described in Section 3.1. The second step creates a schematic that defines the overall connectivity of the components on the PCB. The third step creates to-scale drawings of the packages and places these packages, based on connectivity, onto the PCB. The fourth step, routing, combines the layout and schematic in order to route the connections of the components using multiple layers of the PCB. Once the routing is completed, the fifth step produces Gerber files for PCB manufacturing and assembly. This standard output file format describes each layer of the PCB. Additional support files are required for the final two outsourced steps: manufacturing and assembly. The following subsections provide more detail about these seven steps and how multiple iterations were required in the FAST PCB production process.

3.4.1 Schematic generation

The second step in the PCB production process is creating the schematic. The schematic defines the netlist. The netlist defines how the components are connected. The schematic uses one or more symbols for each component. The schematic symbols have a

one-to-one mapping between the component package pin names and the schematic symbol pin names, defining connections between components. The user can define a net that connects multiple symbol pins of various components together. The schematic can have multiple levels in a hierarchy containing many pages or just one page or file. The FAST schematic uses a hierarchy to instantiate the entire board. Furthermore, for each logical schematic component, we segregate the components into I/O schematics and power and ground schematics to isolate the architecture connectivity. Thus, some component packages are defined by multiple schematic symbols. FAST has a top-level schematic that includes the four processor tiles plus the two FPGAs used for the Hub FPGA. A second top-level schematic contains all the top-level connectors and components not associated with any of the FPGAs. The schematics and further details of the schematics are presented in Appendix A. Once the schematic is complete, the netlist is generated and exported to the layout tool. The netlist contains a list of components required for the design along with the component interconnects.

FAST used Cadence Capture CIS, later renamed Design Entry CIS 15.2, to generate the schematic. This tool had several problems with copying and pasting large symbols, resulting in crashing the program. FAST also uses very descriptive net names, using more than 8 characters per net name. This caused problems with exporting the netlist to other tools because several tools expect no more than 8 characters per net name. These descriptive nets were also an issue for the tools used to process the Gerber files by various PCB suppliers.

The lack of verification tools was the major drawback of the schematic tool. Verifying the schematic was a semi-manual process. The first step in verifying the schematic was dumping the entire netlist. Cadence Capture CIS provides over 25 output formats for the netlist. The format may have certain restrictions, like 8 character long netlist names, requiring judicious output format selection. Once the output format is selected, the schematic netlist can be compared to the expected netlist. For components that were replicated, like the processor tiles, simple scripts could be used to compare all four processor tiles to make sure that the connectivity was consistent. Once the processor tile consistency was confirmed, another script compared an individual processor tile netlist with the expected netlist. There were minor netlist variations that required some manual

intervention during the process, as well as some preprocessing before using the compare scripts. Unfortunately, the Cadence Orcad suite of tools did not have any predefined method to insure schematic correctness. This made creating the schematic very prone to user error. The FAST schematic contained over 4200 nets in the netlist. A single netlist error, like inadvertently connecting 3.3 V to ground, would be catastrophic and can be detected in the layout tool, but other connections or lack of connections can be more difficult to track down because of the manual verification process of the schematic, which percolates the errors to the later tools.

3.4.2 PCB layout

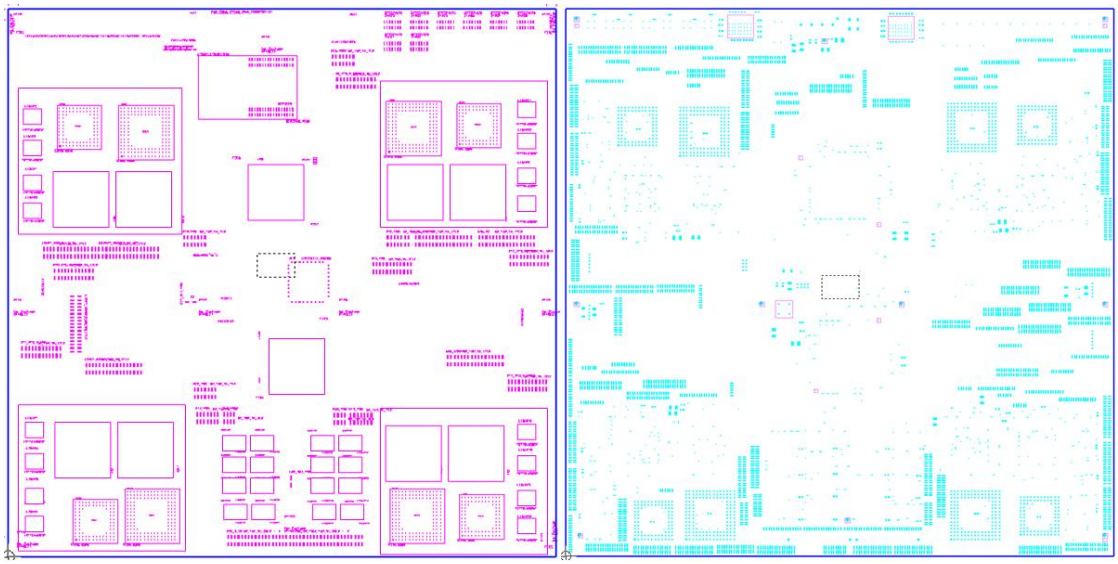
In the third step, a layout tool transforms the connectivity defined by the netlist into a physical PCB design, by placing all the component packages. The layout imports the netlist that defines all the parts and all the connections. Package footprints are created for each unique component. The footprint is the physical design of the component's package. Component datasheets provide the physical dimensions and pin or ball grid array dimensions required to generate a footprint. The connectivity and footprints are used for component placement on the PCB. The PCB itself is just a substrate that enables multiple components to be connected in a wide variety of ways.

Layout can enforce space constraints between components and place components on different layers, as well as many other sophisticated features. For the FAST PCB, many features like embedded resistors and capacitors and blind vias were not used because of the high manufacturing costs. Components are placed on both the top and bottom layers of the FAST PCB. This reduces parts clutter and provides a clear separation between the main components and the support components, capacitors, resistors, and test points.

All 4260 components were placed by hand to insure well-defined placement. This was especially important for the test points and capacitors. For the test points, we defined arrays of test points in the schematic that then could be placed on the PCB. These arrays provided predefined net placement that was difficult to achieve using the grid placement functionality in the Cadence LayoutPlus tool. By creating arrays of test points, we minimized the silk screen labeling required to define the test point net name and reduced the effort required to identify a test point on the actual PCB. The capacitors presented a

different layout challenge. Capacitors are most efficient when placed as close as possible to the point of use, in this case the power and ground pins of the device; the smaller the capacitance, the more critical the placement proximity.

Finally, high-density BGAs were the other main driving force behind parts placement. The large XC2V6000 FPGAs required at least 6 layers to route the I/O pins out from under the chip. The escape routing strategy for these FPGAs presented a significant part placement and routing challenge. These parts were placed far enough apart to facilitate routing, which required larger PCB dimensions. Figure 3.11 shows the component placement on the top layer (A) and the bottom layer (B).



(A) (B)

Figure 3.11. FAST PCB layout showing the (A) top and (B) bottom layers using inverted colors.

The layer definitions are created in the layout tool. Cadence LayoutPlus can manage up to 32 layers having the following pre-defined roles: sixteen layers map to actual electrical (trace, power, and ground) layers on the board. Thirteen other layers are non-electrical and used for documentation and PCB specifications, and the remaining three layers are undefined, but can be used for additional user-defined documentation.

Initially, FAST had four power layers and one ground layer. The four power layers corresponded to the 5 V, 3.3 V, 2.5 V, and 1.5 V power planes. (FAST uses 3.3 V for all I/O pins. This enables communication between the XCV1000 processor tile FPGAs and

the R3000 and R3010. Using a lower voltage output, i.e., 2.5 V or 1.5 V, would not guarantee that the FPGA high output would be translated as a logic high signal by the MIPS parts.) We also had defined 14 signal layers for routing the traces (wires) between the chips for a total of 19 electrical layers: 5 power and ground layers plus 14 signal/trace electrical layers. This required us to use the three undefined layers as electrical layers because LayoutPlus only defines 16 electrical layers including power and ground planes, as discussed above. The result, after using six ground planes, was a 24 layer FAST PCB. After completing this initial design, we found that LayoutPlus cannot export more than 16 electrical layers, which includes power, ground, and I/O layers. The extra three undefined layers can only be used for additional documentation or other non-electrical layer purposes.

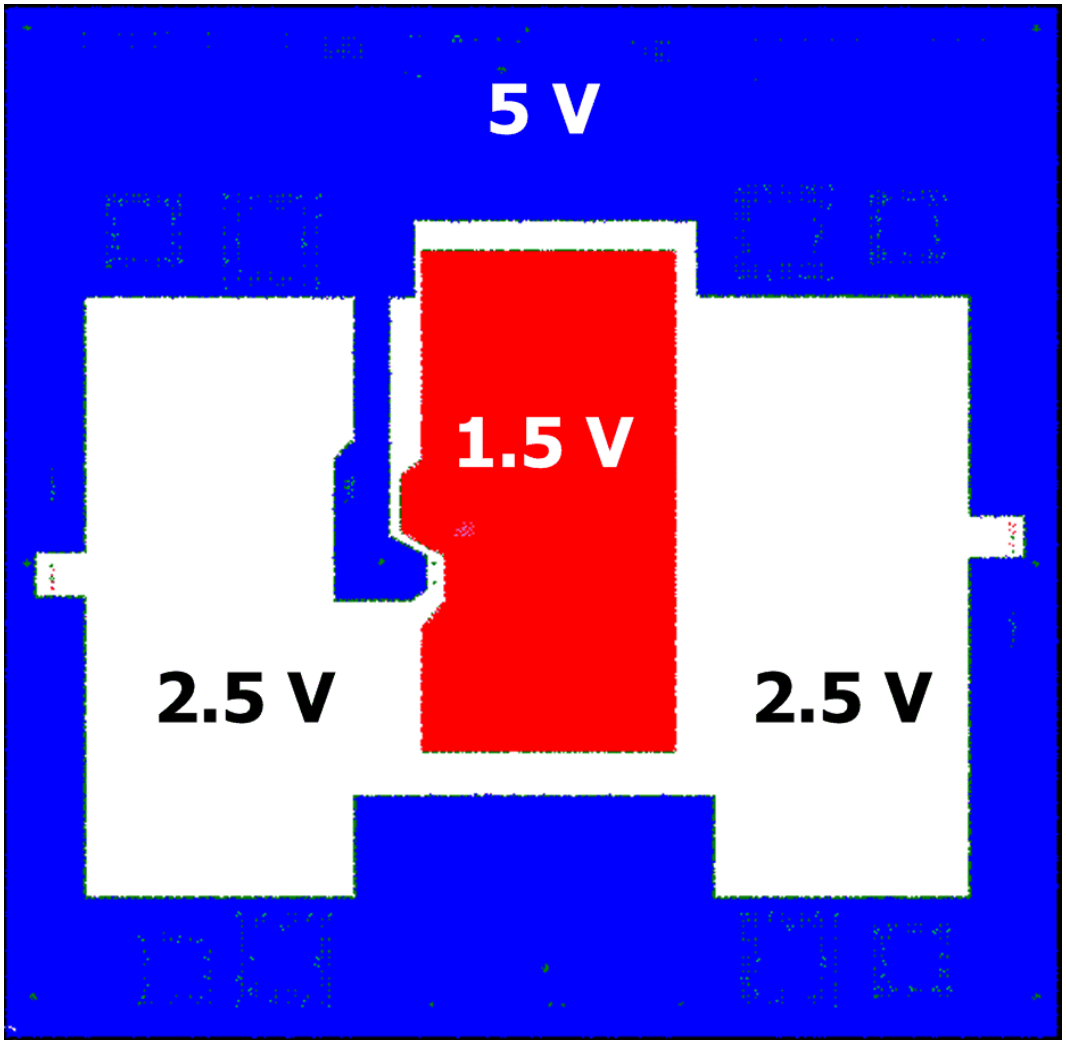


Figure 3.12. Outer 5 V plane, middle 2.5 V plane, and central 1.5 V plane on compressed power layer.

As a result, the 5 V, 2.5 V, and 1.5 V power layers were compressed into one layer that resembles a bull's eye pattern, as shown in Figure 3.12. This convenient layer compression was a result of premeditated parts placement. The result of the layer compression provided one ground layer, two power layers (one for 3.3 V and one for 1.5 V, 2.5 V, and 5 V), and 13 signal layers. The ground layer was replicated to produce 5 layers for signal and power plane shielding resulting in a total of 20 layers. Further layout details can be found in Appendix A.

Changes to the initial layout can require back annotation of the design to synchronize the schematic with the layout. Back annotation generally occurs when new parts or connections are added to the layout that were not initially in the schematic.

3.4.3 PCB routing

The fourth step, PCB routing, takes the output of the layout and creates wires (traces) and holes (vias) on the various layers of the PCB, thereby connecting all of the components and pins defined for each net for the entire board. The signal layers connect pin A on chip 1 to pin B on chip 2, while the power and ground planes provide the ground plane and the I/O and core voltages. The FAST PCB was too complicated for the routing software that is tightly bundled with the LayoutPlus software. Therefore, we exported the design to SPECCTRA, Cadence's high-end routing tool. The initial route took 7 days of compute time using diagonal routing. The diagonal routing had given SPECCTRA too much flexibility and slowed the routing process down dramatically. We then used orthogonal layer routing, which dramatically reduced the route time from 7 days down to 3 days. Bundling bus wires together and routing the difficult buses first, followed by the easy-to-route buses, also reduced routing time. Figure 3.13 illustrates the PCB routing steps used for FAST. There are seven steps, once the trace width and spacing constraints are defined. Each component's BGA spacing and the design for manufacturing rules used by the PCB fabrication house determine the trace width and spacing [84, 102].

Once the PCB is completely routed, the 90-degree angles in the routes are rounded with a mitering step. Mitering converts a 90-degree angle into two 45-degree angles, thereby reducing the EMI radiation caused by electrons turning sharp corners. With the mitering

complete, the full PCB can use a design rule checker (DRC), step 6 in Figure 3.13, to verify that the PCB meets all the constraints and is fully routed. SPECCTRA does not have the capability to directly export the resulting fully routed PCB. Instead, SPECCTRA generates a file that is imported back into LayoutPlus, which re-runs the DRC on the fully routed board. The translation process can place traces off-grid, breaking previously connected components. An additional clean-up step must be done before creating the Gerber files.

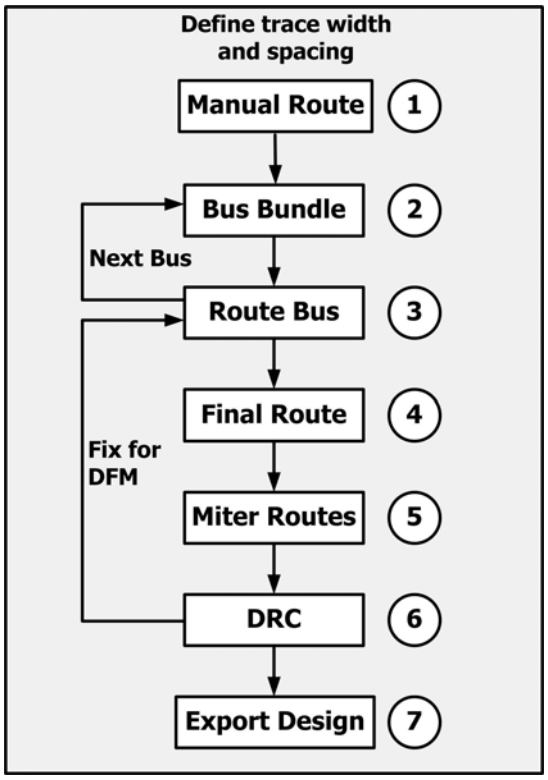


Figure 3.13. FAST PCB routing process.

3.4.4 Gerber file generation

FAST is a 20 layer PCB with 13 signal layers and 7 power and ground plane layers. Each layer is independent of all others, thus the layer stack can be specified with respect to arbitrary electrical constraints. The layer stack describes how the PCB layers will be sandwiched together. The PCB routing is really a 3D problem because each via, the connection that connects two layers together, must not interfere with any other trace or object on any other layer. We used additional ground planes to shield the power planes and separate various signal layers, as shown in Figure 3.14.

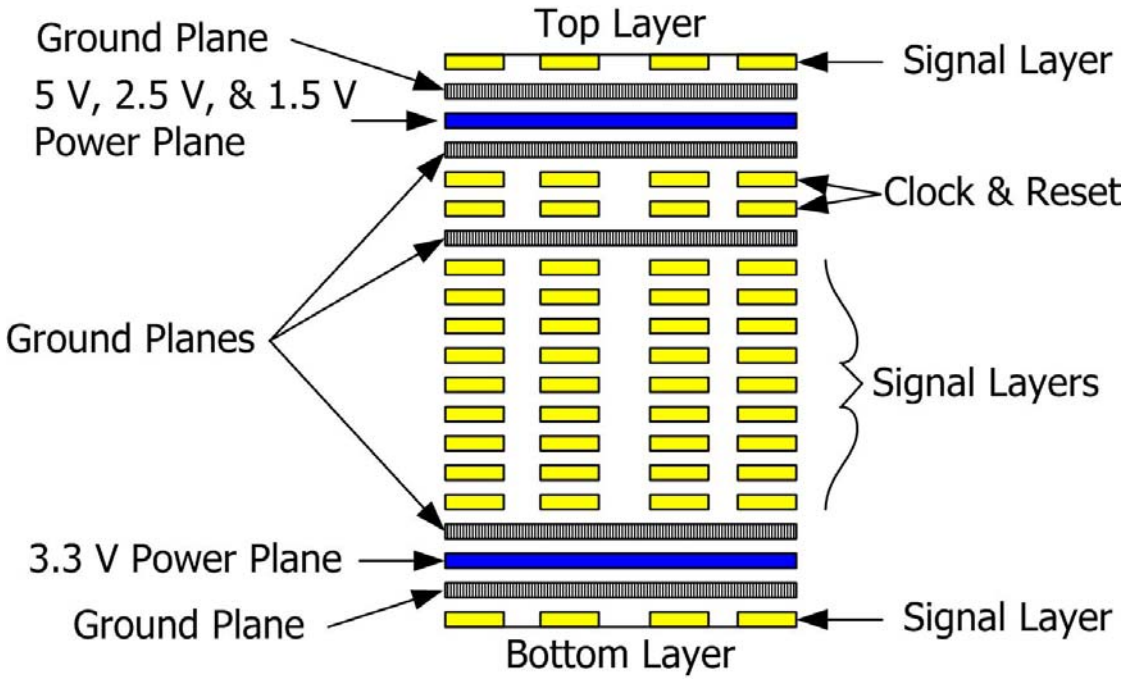


Figure 3.14. 20-layer stack up for the FAST PCB.

The two signal layers clustered below the top layer contain the clock and reset signals. The clock traces have the most frequent oscillations and FAST separates these traces from the rest of the design to reduce cross talk. The additional ground plane below these two signal layers isolates these traces from the rest of the design. The 5 V, 2.5 V, and 1.5 V power plane is placed near the top of the board to reduce the voltage drop for the lower voltages by locating the power plane close to the components. More ground layers could have been used to separate the rest of the signal layers, but that would dramatically increase the PCB manufacturing costs.

There are two other purposes of the Gerber file translation. First, GerbTool manipulates the Gerber files and removes all the unused pads associated with each via. In order to maintain the trace spacing requirements for a via, SPECCTRA places a pad on each layer. This prevents a trace from being routed too close to the via. The pad is the connection point between the via and the trace and is only necessary in the (typically) two layers that are being connected. The second purpose of the Gerber files translation is the ability to reconstruct the netlist from the Gerber files and compare that with the layout netlist to verify the correct connectivity. Figure 3.15 shows all the layers superimposed on the PCB in the GerbTool. GerbTool is a third party tool licensed by Cadence and included in the

PCB tool chain. The black areas in Figure 3.15 are void of traces, components, or silk screens. The center of the FAST PCB has the highest trace density.

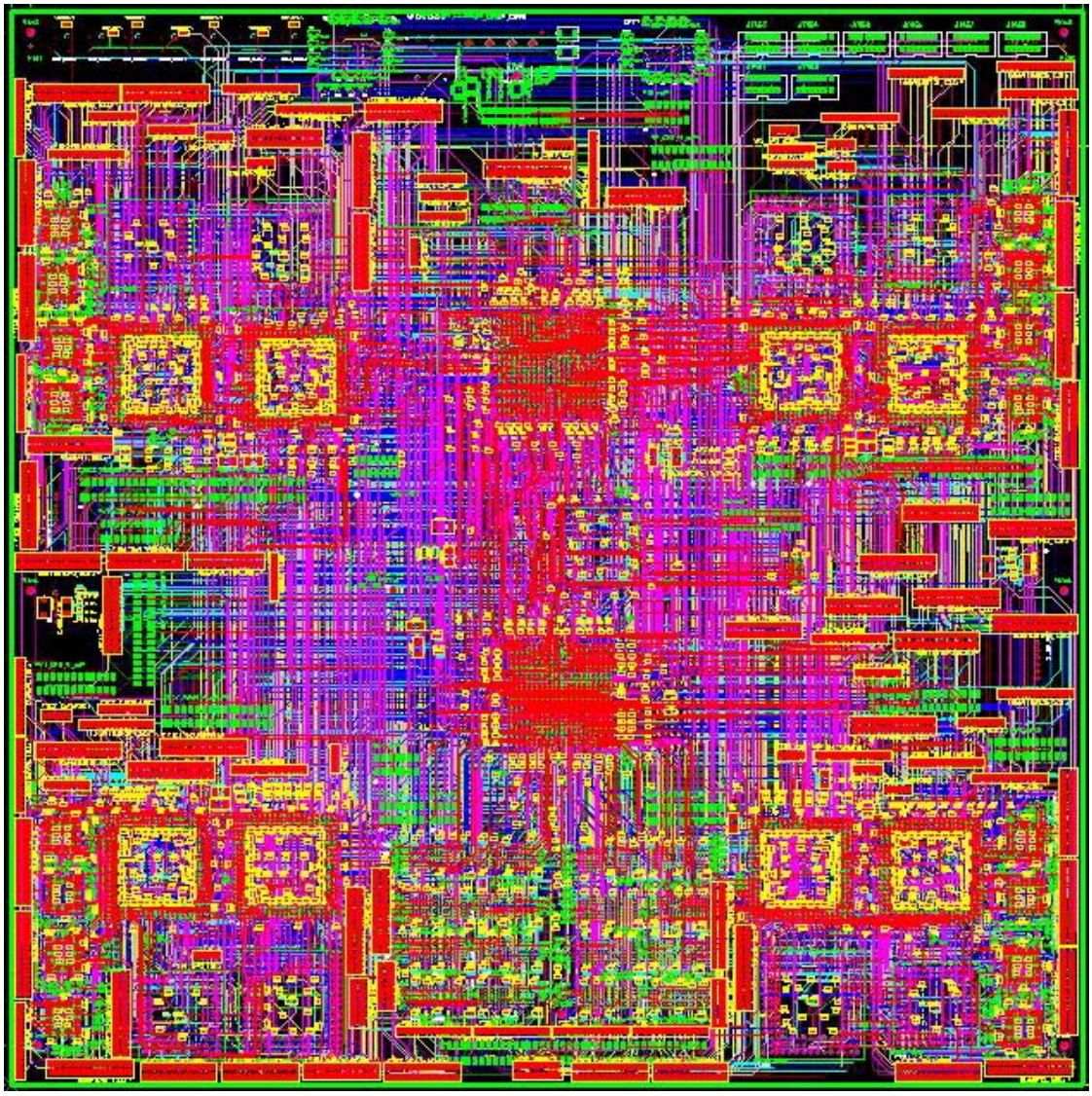


Figure 3.15. Routed FAST PCB in GerbTool with the black areas void of traces and parts.

3.4.5 PCB manufacturing

The PCB complexity resulted in two PCB manufactures unable to bid on the project because the FAST PCB exceeded the manufacturers' capabilities of a maximum of 16 layers and 14" X 17" dimensions. The first FAST PCB specification had design for manufacturing (DFM) issues. First, the trace spacing between the traces and the vias was

too close and required rerouting. With 30,000 vias, rerouting was more efficient than moving traces around each via.

Second, the FAST PCB routed two traces between the vias of the large BGAs of the XC2V6000 FPGAs. This reduced the number of layers required to escape the BGA, but the tight spacing was not feasible given Sanmina's DFM guidelines. The last major issue with the FAST PCB was the high aspect ratio. The aspect ratio is defined as the board thickness divided by the smallest drill hole size. The fine BGA FPGAs forced FAST to use a very small drill hole size of 6.5 mils. The tolerance on these drill holes was also very tight, only 2 mils on a side. Using standard layer thicknesses, a 24-layer FAST board would have been in the range of 120 mils or about 1/8 of an inch. This produced an aspect ratio of about 18.5. This high aspect ratio made it more difficult to produce reliable vias using the standard PCB manufacturing processes. Normal PCBs have an aspect ratio of about 10. By reducing the layer thicknesses, reducing the number of layers to 20, and increasing the smallest drill hole size to 8 mils, the FAST PCB has a total thickness of just under 100 mils and a resulting aspect ratio of less than 12, not including the top and bottom solder masks. This change in aspect ratio made this project have a much higher yield from a manufacturing perspective, resulting in the bare PCB shown in Figure 3.16.

There are several other PCB manufacturing details that are addressed in the PCB specification documentation that was sent to Sanmina. The FAST documentation includes contact information, version number and date, general PCB specifications, layer stack-up and associated Gerber files, and a description of all the files required to produce the PCB. The PCB specifications and layer stack-up can be found in Appendix A.

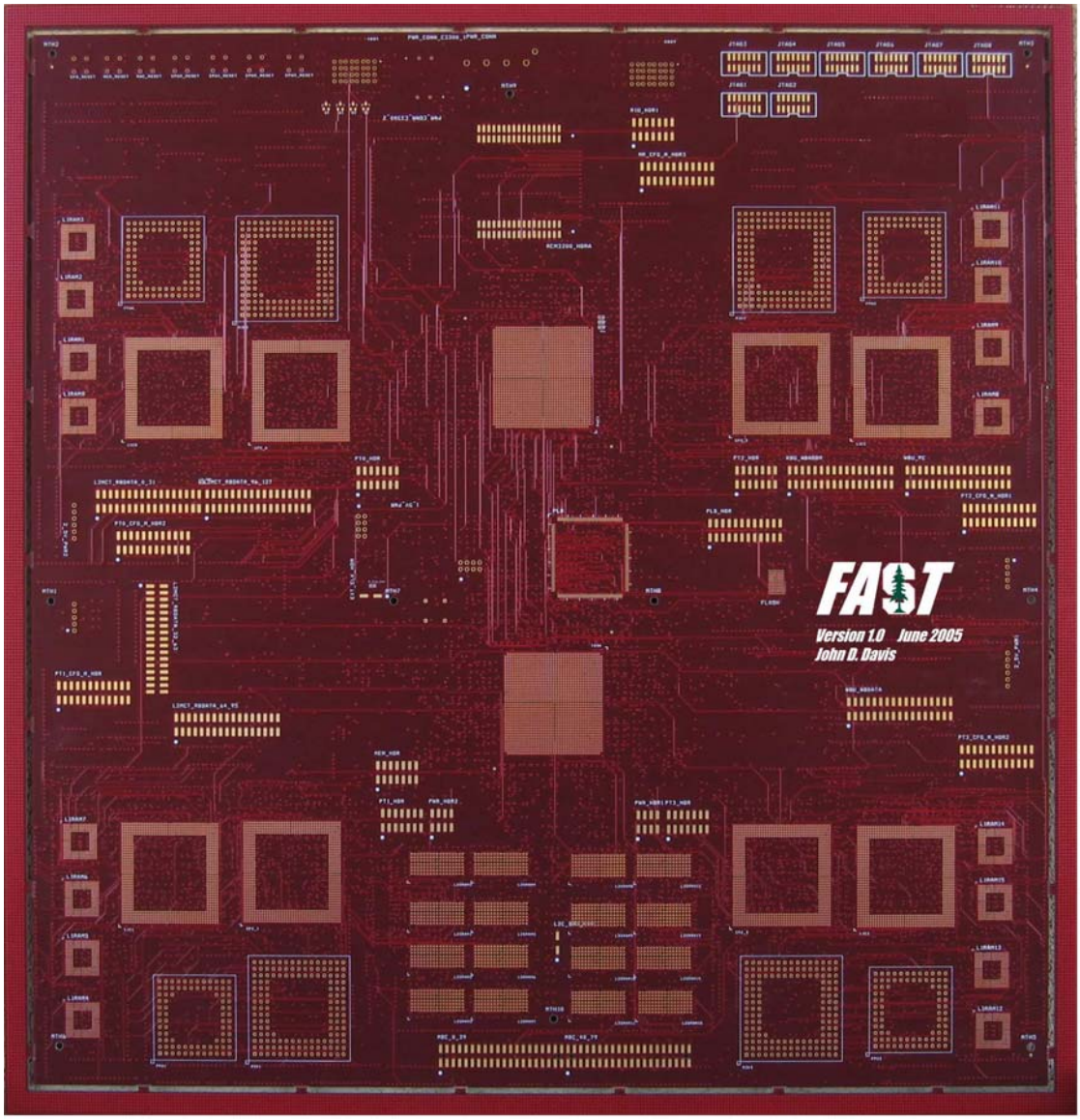


Figure 3.16. Top of bare FAST PCB.

3.4.6 PCB assembly

Although there are only 35 unique building blocks used to make the FAST PCB, there are 4260 individual parts. There are 43 large BGA components that require specialized multi-stage baking ovens that solder the BGA components to the PCB. There are about 2000 test points, each only 40 mils by 105 mils. These small components would have been both difficult and tedious to solder to the FAST PCB by hand. There are also about 1200 surface mount (SMT) capacitors that are in a 0402 package with similar dimensions as the test points. Thus, the BGA components combined with the copious quantities of very small devices made the PCB impossible to assemble outside a PCB assembly plant.

For assembly, the top and bottom assembly layer Gerber files specify where each individual component is placed on the top and bottom layers. PCB dimensions and mount hole dimensions are provided in the fabrication Gerber file. Additional fiducial marks are placed near the large high-density XC2V6000 FPGAs to help the optical machines align the device placement and are specified in the fabrication Gerber file as well. The final result is Figure 3.1, the fully assembled FAST PCB. There are a few devices in Figure 3.1 that are socketed rather than placed during the assembly process. The embedded Ethernet module, the three DC-to-DC voltage converters, and the MIPS R3000s and R3010s all used sockets for easy replacement or upgrade purposes, depending on the device. The ability to replace components was especially important when using the 15-year-old MIPS components. The full component list and relevant assembly deals are provided in Appendix A.

3.4.7 The FAST PCB realized

The FAST PCB design, manufacturing, and assembly process took 18 months, from January 2004 to June 2005 (also see Section 3.6 and Figure 3.19). The PCB design took the first year and included the schematic, layout, routing, and Gerber file generation and validation. The last 6 months included one month of gathering estimates, 3 months of DFM routing revisions, and 2 months of manufacturing and assembly. There are several parts of this process that could have been reduced from 18 months down to 8 months or so by outsourcing step 3 (Layout) through 7 (Assembly) in Figure 3.10. Regardless of the extended PCB timeline, the resulting FAST PCB has required ***no additional rework***. There were a few minor mistakes. Out of about 2000 capacitors, two SMT capacitors had the wrong package specified and were not placed on the PCB. Furthermore, the silkscreen pin 1 designator for PT3_HDR is misplaced. Finally, by mistake, the spacing for FPU1 and R3K1 in processor tile 1 is different from all the other processor tiles. This prevented the symmetric placement of the ZIF sockets for this particular processor tile. However, unlike most prototype PCBs, the FAST PCB has ***NO PCB rework*** and has not faced any issues that inhibit FAST PCB functionality.

3.5 See FAST run

When the FAST PCB was first delivered, several tests were performed before power was applied to the PCB. The initial step is a visual inspection of the PCB. This checks that all the correct components are present and have the correct orientation. There are several headers on the PCB, but only the FPGA mode headers need to have the correct jumpers set. Finally, components with visible solder joints are inspected to verify that there are no solder bridges. Solder bridges are inadvertent connections caused by adjacent solder combining. The FAST PCB has over 4000 components that were visually inspected. There are 43 BGA components that cannot be visually inspected. Sanmina used X-ray inspection to verify the solder ball integrity and the lack of solder bridges under the BGA package.

Using a digital multimeter (DMM), the PCB is checked for shorts between the various power and ground layers and the I/O pins by measuring the resistance without applying power to the PCB. The resistance of the same network varies, but is generally below 5 Ohms. The measured resistance increases when the distance between the two measured points on the same network increases. The measured resistance between two *different* networks, e.g. 5 V and 1.5 V, depends on the devices used between the two networks, but is generally at least an order of magnitude higher, above 50 Ohms. If passive components like capacitors, resistors, or inductors connect two networks together, the resistance could be lower than 50 Ohms. This resistance measurement process is generally referred to as *Buzz out*. The FAST PCB examined the following networks using the Buzz out procedure: 5 V, 3.3 V, 2.5 V, 1.5 V, Ground, JTAG headers, clock network, and reset network, producing a large Cartesian product of combinations.

The next phase of the PCB bring up process applies power to the PCB using current-limited power supplies to prevent excessive current draw and potential PCB damage. With power applied to the PCB, the JTAG chain integrity was verified by reading all the devices on each chain and comparing the device count and device IDs to the number of devices on the PCB and the JTAG device ID listed in the datasheets [41, 54, 55, 101, 111, 127]. FAST uses GSI Technology GS832436B SRAM chips that actually contain two

SRAM chips in a single package [41]. When probed by JTAG hardware and software, each GS832436B actually appears as two devices on the JTAG chain and not one device.

Once the integrity of all the JTAG chains was verified, the entire PCB interconnectivity and integrity was verified using third party software and hardware from Corelis [26]. Corelis makes hardware and software packages that can verify the connectivity and integrity of PCB's using the JTAG chains. The JTAG interface daisy-chains devices together and serial-shifts data in and out of the devices. The Corelis JTAG hardware provides 8 JTAG ports for simultaneous testing and probing. Thus, one JTAG zone can be tested and the interface between it and another JTAG zone can be probed and verified simultaneously. The Corelis interconnectivity testing verified the chip interface functionality.

The Corelis tools can also be used to test various memory interfaces. The initial memory tests were done with the Corelis tools. Traditionally, memory tests use a JTAG-enabled device, like an FPGA, to read and write a Flash memory or SRAM. However, this memory test infrastructure was modified to interface directly to the JTAG-enabled SRAM chips to verify the SRAM functionality. All JTAG tests using the Corelis tools were performed at 5 MHz, although the FPGAs can operate the JTAG interface at up to about 10 MHz.

PLD LED counter Verilog:

```
module PLD_LED(SYS_CLK, CFG_LED);
  input SYS_CLK;
  output [3:0] CFG_LED;

  reg [31:0] count;
  reg [31:0] next_count;

  always @ (posedge SYS_CLK) begin
    count [31:0] <= next_count
[31:0];
    next_count [31:0]
        <= count [31:0] + 1'b1;
  end

  assign CFG_LED =    count[23:20];

endmodule
```

PLD LED UCF pin mapping:

```
NET "CFG_LED<0>" LOC = "p208";
NET "CFG_LED<1>" LOC = "p207";
NET "CFG_LED<2>" LOC = "p206";
NET "CFG_LED<3>" LOC = "p205";
NET "SYS_CLK" LOC = "p183";
```

Figure 3.17. FAST PLD LED counter Verilog and UCF file with pin mapping.

Finally, with the initial bring-up test complete, the programmable devices can be tested. The first test program downloaded to the FAST PCB used the clock oscillator and a counter in the PLD to activate the 4 LEDs connected to the PLD. Figure 3.17 provides

the Verilog code and accompanying user constraint file (UCF) required to create the configuration bit stream for the PLD. The UCF file provides the net or port name mapping to the device pin name. This simple test blinks the LEDs at a human perceptible rate and demonstrates the correct programming of the PLD.

Figure 3.18 provides the extended PLD Verilog that distributes the clock to the other FPGAs including an LED counter for the RWC FPGA. These simple tests were done the first day the FAST PCB was powered on. From there, the software infrastructure was built and every time the FPGAs are programmed, some small counter that activates some LEDs is used to verify that the FPGA is programmed and operational. By extending the PLD UCF file with the clock distribution pins and creating a similar UCF file for all of the FPGAs, all the LEDs on the FAST PCB came to life once they were programmed with their configuration bit stream. All the Verilog code and corresponding UCF files used for the LED counters can be found in Appendix B.

```
PLD LED counter and clock distribution
Verilog:

module PLD_LED(SYS_CLK, CFG_LED,
CP2_GCLK, L1C_GCLK, MEM_GCLK, RWC_GCLK,
 CFG_M, CFG_PROGB, CFG_INITB);
  input SYS_CLK;
  output [3:0] CFG_LED;
  output [3:0] CP2_GCLK;
  output [3:0] L1C_GCLK;
  output MEM_GCLK;
  output RWC_GCLK;
  output [2:0] CFG_M;
  output CFG_PROGB;
  output CFG_INITB;

  reg [31:0] count;
  reg [31:0] next_count;

  always @ (posedge SYS_CLK) begin
     count [31:0] <= next_count [31:0];
     next_count [31:0]
        <= count [31:0] + 1'b1;
  end

  assign CFG_LED =   count[23:20];

//distribute the system clock to FPGAs

BUF clk_bufg
 (.I(SYS_CLK),.O(MEM_GCLK));
BUF clk_bufg1
 (.I(SYS_CLK),.O(RWC_GCLK));
BUF clk_bufg2
 (.I(SYS_CLK),.O(CP2_GCLK[0]));
BUF clk_bufg3
   (.I(SYS_CLK),.O(CP2_GCLK[1]));
BUF clk_bufg4
(.I(SYS_CLK),.O(CP2_GCLK[2]));
BUF clk_bufg5
(.I(SYS_CLK),.O(CP2_GCLK[3]));
BUF clk_bufg6
(.I(SYS_CLK),.O(L1C_GCLK[0]));
BUF clk_bufg7
(.I(SYS_CLK),.O(L1C_GCLK[1]));
BUF clk_bufg8
(.I(SYS_CLK),.O(L1C_GCLK[2]));
BUF clk_bufg9
(.I(SYS_CLK),.O(L1C_GCLK[3]));

endmodule

RWC LED counter Verilog:

module RWC_LED(SYSCLK,LED);
  input SYSCLK;
  output [3:0] LED;
  reg [31:0] count;
  reg [31:0] next_count;
  wire [3:0] LED;
  wire MY_CLK;

  always @ (posedge MY_CLK) begin
     count [31:0] <= next_count [31:0];
      next_count [31:0]
        <= count [31:0] + 1'b1;
  end

assign LED [3:0] = count[23:20];
BUFG clk_bufg (.I(SYSCLK), .O(MY_CLK));

endmodule
```

Figure 3.18. FAST PLD clock distribution and LED counter Verilog and RWC LED counter Verilog.

Hardware technology convergence made it possible to integrate a variety of components to build a flexible four-processor emulation and prototyping system called FAST. Both the architecture and implementation also provide the capability to extend the system, building a larger prototyping fabric. This chapter has described the PCB implementation process and successful functioning. The FAST PCB was demonstrated to be functional and ready for the FAST Software toolbox to be built because building the PCB hardware is a small fraction of the required functionality. Without software, the PCB is useless. The next chapter will cover the software development and resulting FAST Software toolbox.

3.6 FAST hard lessons learned

The FAST project has run into several problems along the way that have stalled the eventual success of the project. Project staffing was the first major FAST roadblock. The initial graduate student responsible for designing, layout, and routing the PCB design left the project in December of 2003, after a year of no progress. Unfortunately, from this point forward, it was easier to restart the project and move forward than to try to use the incomplete work that existed.

The initial time scheduled for the project was one month for the schematic, 3 months for the layout and routing and 2 months for manufacturing and assembly. Two major factors compromised the completion of this project and resulted in a much longer schedule. First, it was recommended that we use the mid-grade Cadence tools to design the PCB. The main reason was that these tools had a much shorter learning curve and thus could produce results much faster. However, inherent limitations to the tools made them inadequate for the FAST PCB. The FAST PCB project broke every tool in the PCB tool chain. In some cases, saving the files frequently or not copying and pasting large section in the schematic tool could work around the problems. However, there were problems with the layout and routing tools that stopped all forward progress. Unfortunately, each time this happened, the problem would have to be escalated through the customer service process until it was routed to the development team to fix the bug, provide a work around or reject as an unsupported tool feature. This customer service cycle required 1-2 months added to the PCB development time.

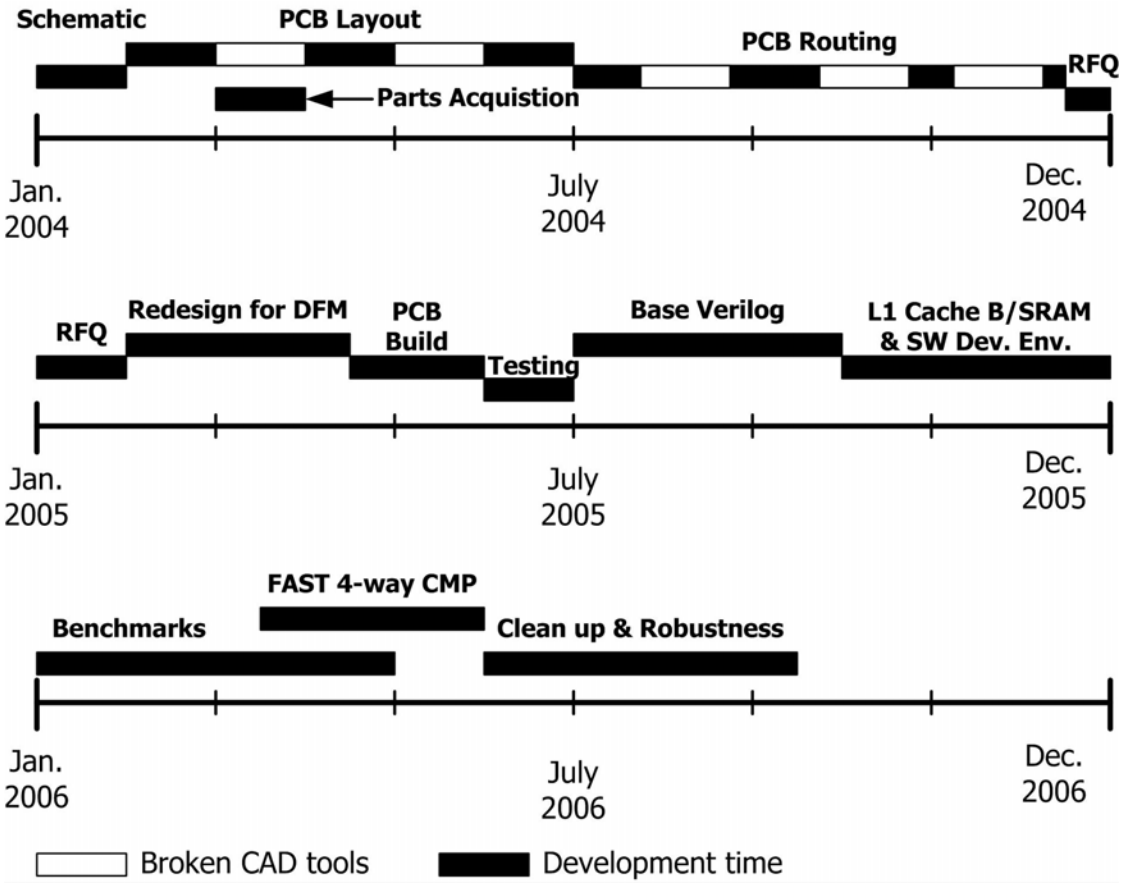


Figure 3.19. FAST PCB and software timeline.

Figure 3.19 provides a timeline for the FAST PCB and software development. This timeline starts when the FAST project was restarted in January 2004. The work from the prior two years, starting in 2002, was redone. This figure also shows the delays incurred from CAD software problems. The PCB layout and routing was extending by about 5 months due to CAD tool problems. The request for a project quote (RFQ) required 6 weeks to send out the design and select a PCB prototyping vendor. Because of the complexity of the PCB, only four out of the six original vendors could produce the FAST PCB. About 10 weeks were required to redesign the PCB to meet Sanmina Corporation's Design for Manufacturability (DFM) guidelines. There were three iterations and each iteration took 2 weeks to get feedback from Sanmina before the next iteration could begin. Once the FAST PCB was manufactured and assembled and tested, the rest of the time after July 2005 was spent on building the software required to make the PCB functional. The FAST software development is discussed in Chapter 4. Initial software

development ramped up until June 2006. In June, all systems of the FAST 4-way CMP were partially working with sporadic successful results. Thus, from June until September 2006, all major software development focused on system robustness, loading programs successfully all the time and code clean up.

To provide some insight into some of the broken CAD tool issues, as mentioned previously in this chapter, the initial FAST PCB design had 24 layers, using 10 power and ground layers and 14 signal layers. Three of the signal layers could not be exported as Gerber files from LayoutPlus. Unfortunately, the 16 electrical layer limitation was not prominently described in the tool documentation. Furthermore, it took over a month to promote the problem to the development team, while at the same time trying various workarounds to solve the problem locally. Once the development team looked at the problem, they rejected fixing the problem or providing a workaround. This forced the layer redesign and re-layout.

There were a few other problems that were similar in terms of additional time added to the project: LayoutPlus router failure and Gerber file netlist comparison failure. On several occasions, the project tried to upgrade to the high-end Cadence tools, but the conversion tool failed and the process would have required starting completely over from the beginning, including a very long learning curve. As a result, future large-scale PCB projects, more than 10 layers and larger than 5" X 8", should be outsourced for timely delivery and reduced tool problems. Outsourcing the PCB layout and routing would have kept the FAST PCB phase on target with regard to the time schedule and accelerated the time to research for this and future projects. As Figure 3.19 illustrates, five months could have been saved in the layout and routing process and another 10 weeks could have been saved because there would be no redesign for DFM. Overall, that accounts for almost eight months that could have been shaved off the timeline presented in Figure 3.19. Finally, if more resources were available, the software development could have been done in parallel as well, further reducing the time required to produce a fully functional FAST system.

Chapter 4

FAST software architecture

The previous chapters describe the motivation, architecture, and hardware implementation of the FAST PCB. The result of these chapters is a fully functional PCB that functioned correctly, out of the box, with ***no rework***. Unfortunately, hardware is not the complete answer. Truly to be useful, the FAST system must include software that enables kernel and application development. The FAST software architecture is the other half of the system equation. The PCB was designed to have very few software limitations. The complete hardware and software stack is shown in Figure 4.1. Chapter 2 and 3 described the hardware architecture and implementation. This chapter describes the FAST software architecture from the FAST Verilog Abstraction Layer (VAL) up to the application layer.

The FAST software architecture encompasses many levels of software and tools, from low-level software that makes the PCB functional (morphware), to applications running on the prototype system. This chapter presents a bottom-up view of the software components and tools required to make FAST functional. These tools and software components apply to all similar flexible prototyping systems. This chapter will describe all of the software layers that build on top of the FAST PCB, as shown in Figure 4.1.

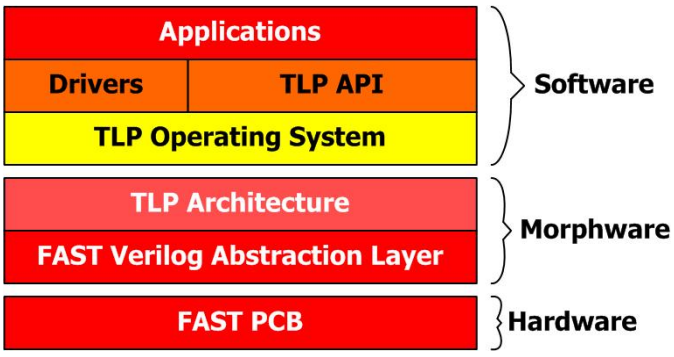


Figure 4.1. FAST complete hardware and software stack.

First, we present FAST VAL. The FAST VAL is the software that describes and implements the base hardware functionality. These base Verilog modules provide the

MIPS interface for all designs. Similarly, additional Verilog modules are built on top of the FAST VAL and provide the prototype or thread level parallel (TLP) architecture definition. There are also hardware definitions that provide hooks for profiling and performance counter definition. FAST also has the ability to run fully functional operating systems (OS). However, the OS must be ported and auxiliary software must be provided to start up FAST and support the OS operations. For simplicity, we group the low-level boot software and OS together. There are also additional drivers and (TLP) API's that provide prototype specific functionality. Finally, building software using common widely available tools, applications are compiled for the FAST or TLP prototype target. The following subsections present the tools and software components of Figure 4.1 in a bottom-up approach. This presentation basically follows how the software was developed for the FAST PCB.

4.1 Building software on top of hardware

Working hardware does not exist unless the software exists to make it work! The FAST PCB is nothing without software to support system prototyping. The FAST Verilog Abstraction Layer (VAL) manages the basic functionality of the FAST PCB. The FAST VAL is a collection of Verilog modules that defines interfaces that every prototyping system will use. This section describes the various layers of the FAST VAL.

There are two basic components to VAL, the Verilog modules and the corresponding User Constraint Files (UCF). The Verilog files define the hardware functionality and are part of the higher-level *morphware*. Morphware is all Verilog or other code that defines the FPGA functionality. Because FAST is built with configurable hardware, the hardware configuration software morphs the hardware into a virtually unlimited number of configurations, thus, transforming "software" to a more aptly named "morphware," as shown in Figure 4.1. The UCF files map port names of the Verilog modules to pins on the PCB devices. The UCF files can also define hardware parameters, timing constraints, and initial values for internal registers. The Verilog files and modules are managed, compiled, and synthesized using the Xilinx Integrated Software Environment (ISE) because the FAST PCB uses Xilinx programmable devices [116]. However, other development systems exist [90]. First, we describe the tools used to create the FAST

VAL. The Verilog development environment is followed by a discussion of the actual FAST VAL.

4.1.1 Xilinx development tools

As mentioned in Chapter 3, the FAST PCB uses Xilinx programmable devices to facilitate the flexible system prototyping substrate. Xilinx provides a set of development tools that enable system development and testing. There are two tool packages that are available: the Embedded Development Kit (EDK) [120], primarily for Virtex II Pro with embedded PowerPC 405 processors; and the Integrated Software Environment (ISE) [116]. The FAST VAL uses the ISE tools because we are not using FPGAs with embedded hard cores or other Xilinx IP that requires the EDK tools. The ISE tools bundle a development environment, compilation, logic synthesis, placement and routing, bit file generation, FPGA programming tools, and simulation tools in one graphical user interface framework.

The primary function of the ISE tools is to transform synthesizable Verilog for a particular FPGA into an FPGA programming file that realizes the intended function of the Verilog code. This is accomplished by a collection of tools that are invoked within the ISE framework. The ISE framework bundles Xilinx devices with all the relevant files and support information. There are two main components required for each programmable device, the Verilog code and the user constraint file (UCF). The Verilog code defines the hardware function to be implemented on the FPGA. The UCF file defines the port to device pin mapping, initialization values, signal and clock timing constraints, and a variety of other user specified values. A list of the possible constraints can be found in Xilinx's Constraints Guide [106].

There are three main steps that take place within the ISE framework: design synthesis, design implementation, and programming file generation. Once the device and its associated files go through these steps, the ISE framework provides a design summary of device utilization, performance score, and failed constraints. In each case, there are reports that provide detailed information for all the individual steps in the process.

Design synthesis processes the structural Verilog and converts the code into Boolean logic. The design implementation phase translates the Boolean logic into structures that map to the Xilinx logic building blocks. In the case of the FPGAs, these structures are four input look-up tables or 4-LUTs. After translation and mapping, the most complicated process happens: the design is placed and routed in the target devices. This phase of the process is referred to as PAR and determines the device's design operating frequency. The design's operating frequency is a function of the design complexity, where the logic is placed in the device, and how wires are routed in the device. For the XCV1000's, the maximum chip operating frequency is 200 MHz and for the XC2V6000's, the maximum chip operating frequency is 400 MHz. Needless to say, implemented designs have an operating frequency that is less than the previously stated maximum device operating frequencies. The final stage in this process generates the programming or bit file that is downloaded to the Xilinx device and that then molds the flexible FPGA substrate into complex logic.

There are specific tools in the ISE framework that were repeatedly used to improve correctness and performance. Unlike software simulators, hardware must meet the device timing. FAST uses dedicated processor cores that enforce specific timing requirements on the FPGA interfaces. If the FPGA interfaces are too slow or too fast, the system will not work. There are three steps that were used to produce the correctly functioning FAST VAL framework. First, the device placement and correct routing defines the functional correctness. Second, the FAST VAL used Verilog simulation to prove the implementation correctness. Finally, the internal and external FAST observation tools are presented and demonstrated both implementation and functional correctness.

Placement and routing (PAR)

An FPGA is basically a sea of gates, wires, and memory blocks. FAST must meet the timing requirements of the MIPS R3000 and R3010. If these hard timing constraints are not met, the system will not function because the processors cannot get correct data. The logic placement and wire routing in the FPGA affects the timing of FPGA. The location of the logic block and the length and type of wires used dramatically impacts the signal propagation time. The FPGAs must capture and supply data at specific times based on

MIPS timing diagrams [54]. If the logic is placed too far away from the signal source or the wires used to route signals are very long, the data will not reach its destination in time. The best example of this is the fixed placement of the delay buffer used to generate the phase delay for the Clk2XSmp signal. For repeatable results, it was best to fix the buffer placement and routing as opposed to using a timing constraint.

The Xilinx FPGA Editor can display logic placement and wire routing. This tool can also dump the placement and routing of particular signals and complex logic and wire routes. By combining the information from the FPGA Editor with the post PAR timing report, the timing dimension can be resolved. Post PAR, the FPGA Editor provides timing information about individual segments, whereas the timing report provides pin-to-pin timing information for signals, with the latter generally begin more useful. If logic and routes are manually placed in the FPGA Editor, the FPGA Editor no longer provides credible timing information for the manual routes.

In order to tackle difficult timing problems, user-timing constraints are specified in the UCF file. A timing report provides the information on signals that do and do not meeting the timing constraints. Using the FPGA Editor, the signals that meet timing constraints are recorded and added to the UCF as a fixed placement and route of the logic and wires. The signals that do not meet timing are re-routed in the next PAR iteration. As each individual signal is successfully routed, it is fixed in the UCF file. Fixing the routing also has the side benefit of reducing the amount of time it takes to do the PAR. Once all the signals meet the timing constraint, the iterative process is complete. Sometimes, just placing a particular logic cell can improve signal timing and enable the tools to meet the timing constraints. Section 4.1.3 discusses design timing in more depth.

Simulation

Simulation is used to verify the correctness of the Verilog code. The FAST project used ModelSim to verify its structural Verilog [72]. There are two ways to simulate Verilog modules. The simplest method simulates the module functionality without regard to timing. The second method incorporates signal timing using post PAR and device information. FAST uses the device models and libraries supplied by Xilinx.

Furthermore, the SRAMs also had Verilog modules supplied by the SRAM providers that we used in our memory simulations.

In general, the FPGA libraries and related Xilinx libraries were mature and provided accurate results. However, for some of the SRAM simulations, the simulation would produce functionally correct waveforms, but the programmed FPGAs had certain problems, like bus contention, that were not exposed in the simulation environment. As a result, implementation and observation on the PCB was the top priority with simulation-based validation taking a back seat in the development process, unlike a normal development cycle.

JTAG programming and observation

The Xilinx devices are programmed with the Xilinx tool iMPACT [108]. This tool interfaces with a Xilinx programming cable and the JTAG headers on the FAST PCB. The FAST project uses the Xilinx parallel IV cable and a JTAG header cable to program the Xilinx devices [119]. iMPACT can also be used with JTAG chains that have non-Xilinx devices on the chain. Any device on the JTAG chain can be bypassed. When the FAST PCB was first powered on, iMPACT was used to verify the JTAG chain integrity and download each device ID code. This initial verification step enabled relatively rapid testing and development using the JTAG ports.

The JTAG interface can also be used to monitor other Verilog modules or events inside the programmed Xilinx devices. Xilinx provides ChipScope, an infrastructure to embed Verilog modules that can monitor signals or other Verilog modules and collect a limited amount of data using the JTAG port[113]. The main advantage of ChipScope is its ability to monitor internal signals and events, but it can also perform limited monitoring of I/O pins and record a limited amount of samples. As mentioned in Chapter 3, we can also use JTAG hardware and software to statically monitor I/O pins and chip interfaces. The FAST project primarily used J-SCAN from Macraigor Systems [67]. This tool samples the values on the input or output boundary scan cells and shifts them out to a GUI. This static monitoring enables sampled steady state observation of the FAST PCB and related software.

Updating bit files

Bit files define how the FPGA is programmed to instantiate the collection of Verilog modules.

The initial FAST prototype system, to be described in Section 4.2.2, relied on the Xilinx specified FPGA-resident block RAM or BRAM. The L1 and L2 memories were instantiated using these Xilinx defined primitives. Using these memory primitives reduced the integration effort and enabled rapid progress because all the design efforts focused on the FPGAs and no other components. The BRAMs stored the program information, both instructions and data. The actual data format is discussed in Section 4.3. Thus, given a fully functional bit file, the only thing that needs to change to load different programs is the contents in the BRAMs.

Xilinx provides the Data2Mem tool that can replace the contents in BRAM with new data, producing a new bit file containing a different application program (benchmark) than the original bit file. The other logic specified in the bit file remains unchanged. This is the perfect solution to changing the bit file to load new test programs or rapidly create multiple bit files for a variety of applications. The Data2Mem tool reduces the bit file generation time from hours to a few minutes. The FAST PCB uses two different FPGAs from two different generations, each with its own BRAM primitive. The transition from Virtex I to Virtex II ushered in a new, larger BRAM primitive. This also requires two different Data2Mem wrappers to process the memory contents and push it into memory. Using Data2Mem to swap out BRAM contents is a multi-step process that is different for Virtex I and Virtex II FPGAs.

A batch file invokes various scripts and tools that generate new bit files for the caches in the L1C FPGA. This enables L1 cache preloading. By preloading the L1 cache, same application and/or diagnostic tests can be executed without a higher-level memory system.

The BRAM update process is outlined in Figure 4.2. As Figure 4.2 illustrates, updating the contents of BRAM and creating a new bit file has three basic steps. The first step

uses gawk to convert a disassembled binary text file into a memory image file (MEM) that contains an address followed by a user defined number of values after each address. For the scripts used in this example, each line contains 8 32-bit words. Figure 4.3 provides a snippet of the memory image file for bubble sort.

Next, in step 2, Xilinx's Data2Mem tool uses an old bit file, a bit file memory map (BMM), and a new memory image file (MEM), and creates a new bit file. Unfortunately, for Virtex I FPGAs, Data2Mem cannot generate the correct checksum for the new bit file, requiring another step. Furthermore, the Xilinx tools cannot ignore the checksum in the Virtex I bit file, resulting in a bit file that cannot yet be used for programming.

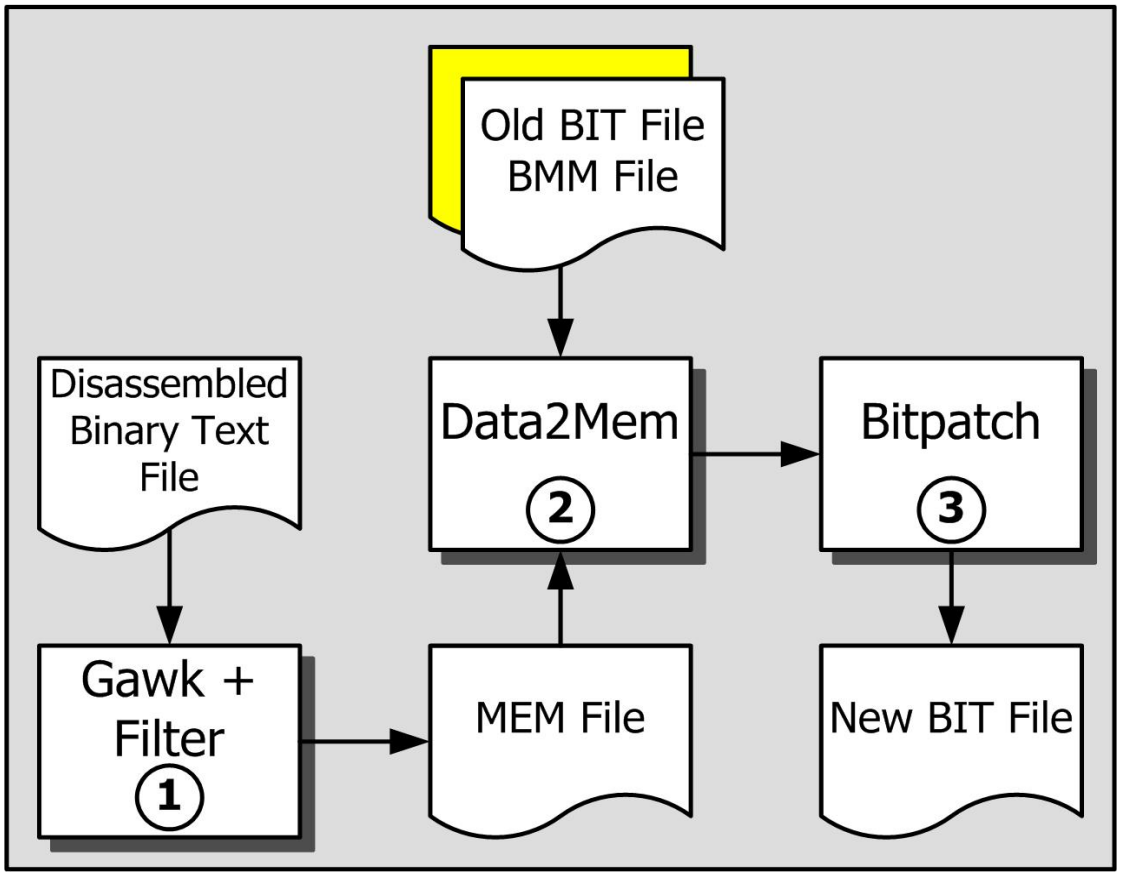


Figure 4.2. Virtex I BRAM bit file modification script.

The final step takes the new bit file and reads through it to create the correct checksum and modifies the bit file by adding this correct checksum. By reading through the Xilinx bit file specification, one can derive this bitpatch script functionality for Virtex I FPGAs. A user from the fpga.arch news group supplied our bitpatch script [25]. Leveraging

software like bitpatch from the FPGA community saved several days of development time, resulting in using Data2Mem with Virtex I FPGAs.

```
Start.s :
    nop
    nop
    nop
    nop
    la      $1, Setup       #setup jump address
    jr      $1              #jump to cacheable kernel space, kseg0
    nop

Setup:
    la      $gp,_gp                 # setting up the GP using Macro
    nop
    lui     $sp,0x8002              # setting up SP at 8000 8000
    ori     $sp,$sp,0x0000
    nop
    addiu   $sp,$sp,-8              # making some room on the stack
    sw      $sp,0($sp)              # storing SP
    move    $fp,$sp                 # setting up FP
    nop
    move    $sp,$fp                 # restoring SP
    jal     main                    # jumping to main

MEM File for Start.s :
@00000 00000000 00000000 00000000 00000000 3C018000 24210020 00200008 00000000
@00020 3C1C8001 279CBFF0 00000000 3C1D8002 37BD0000 00000000 27BDFFF8 AFBD0000
@00040 03A0F021 00000000 03C0E821 0C000080 00000000 00000000 00000000 00000000
...
```

Figure 4.3. Start.s assembly code followed by the memory image file snippet for bubble sort that jumps into cacheable kernel address space and sets up the stack and global pointer.

The L2 memory was specified in the RWC FPGA using Virtex II BRAM primitives. Instead of directly interfacing to the underlying BRAM primitive, the L2 memory was specified with the Xilinx CORE Generator™ tool [114]. This is a GUI that specifies the memory size, memory primitive, hand shaking signals, area or speed optimizations, and memory initialization file. With this information, CORE Generator™ creates the Verilog wrapper and the other support files required for the memory block. In our case, 32 KB of L2 memory was instantiated with sixteen 2 KB BRAM blocks for the example architecture presented in Section 4.2.2. Each BRAM block has four additional bits for parity and uses a 512 entry by 36-bit BRAM block primitive. Thus, for this memory configuration, multiplexers are required to route the data to and from the appropriate BRAM block. This could present a performance bottleneck because we are not bit slicing the BRAMs, but the maximum operating frequency of the RWC using this memory configuration was reported to be about 275 MHz, over an order of magnitude higher than the global system clock.

The Virtex II BRAM modules use a coefficient file to initialize the BRAM modules. The coefficient files (COE) are very similar to the memory images files. An example COE file is shown in Figure 4.4 for the same start.s code shown in Figure 4.3. Comparing the two figures, the main difference between the COE file and MEM file is the header information and lack of address in the MEM file.

```
memory_initialization_radix=16;

memory_initialization_vector=
00000000,00000000,00000000,00000000,3C018000,24210020,00200008,00000000,
3C1C8001,279CBFF0,00000000,3C1D8002,37BD0000,00000000,27BDFFF8,AFBD0000,
03A0F021,00000000,03C0E821,0C000080,00000000,00000000,00000000,00000000,
...
```

Figure 4.4. Memory coefficient (COE) file for initializing Virtex II BRAM.

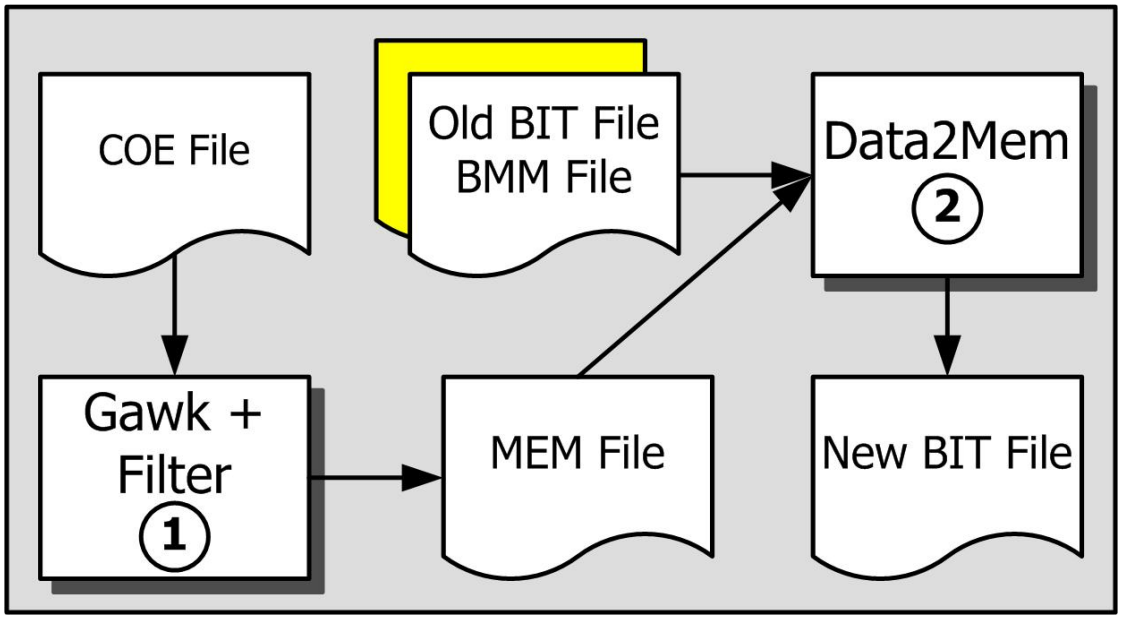


Figure 4.5. Virtex II BRAM bit file modification script.

The Data2Mem tools require less manipulation for Virtex II BRAM data updating than their older Virtex I BRAM modules. As shown in Figure 4.5, creating new bit files with updated BRAM contents only takes two steps. First, a gawk script converts the COE file to a MEM file by removing the header and adding the explicit address mapping. Second, Data2Mem is used with an old bit file, bit file memory map (BMM) file and a MEM file to create a new bit file.

Data2Mem, operating on Virtex II FPGA bit files, does not suffer from the same checksum update error as for Virtex I FPGA bit files. Thus, bitpatch is not required and the Data2Mem bit files can be loaded immediately to the Virtex II FPGA. Regardless of the number of steps, using Data2Mem reduces the bit file generation from a minimum of 20 minutes to a maximum of 2 minutes. For some of the larger FPGAs, the bit file build time can extend to several hours without using fixed routing. By using Data2Mem to eliminate the bit file build process, significant time savings are realized for updating BRAM contents.

Building bit files

Building bit files for FPGAs or jed files for PLDs is very straight forward. Jed files define the functionality of PLDs much like bit files define the functionality of FPGAs. PLDs are a different devices and thus use a different programming file format. The previous subsections were presented because they deviate from the normal programming file generation process. Using the ISE project navigator, the device type is selected and the particular Verilog and UCF files are loaded. The user must associate the UCF file with the top-level Verilog file. When the user selects the "Generate Programming File" in the Process View window, all the relevant Xilinx tools are invoked and, if successful, a bit file is generated for iMPACT to download to the programmable device.

4.1.2 FAST VAL

The FAST Verilog Abstraction Layer (VAL) provides the basic functionality of the FAST PCB. The FAST VAL includes: UCF files to define the overall connectivity; Verilog port name wrappers for all FPGAs; clock distribution and generation; MIPS R3000 initialization and handoff code; and the MIPS R3000 cache and memory interface in the processor tile. The base Verilog modules that make up the Verilog Abstraction Layer (VAL) are shown in Figure 4.6, with the development order shown moving from the bottom PCB layer up to the top cache and system interface layer.

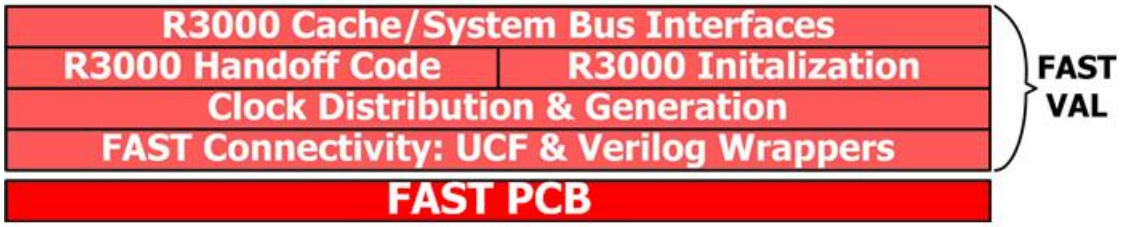


Figure 4.6. The FAST PCB with the Verilog Abstraction Layer components on top.

The FAST VAL is a crucial part of the FAST Software Toolbox because it provides an abstract interface to the user. By maintaining an abstraction layer, the underlying VAL and PCB implementation can change without breaking the software built on top of the FAST VAL. Moving forward, leveraging as much software as possible reduces the software development time and effort of next generation hardware projects. Thus, the next generation FAST can leverage both the advances in silicon technology realized in FPGAs and SRAMs, as well as take advantage of the pre-existing software built on top of the VAL.

FAST connectivity

The FAST PCB has 11 configurable devices with almost 6500 I/O pins that are mapped to various components on the PCB. The fixed function components like the SRAMs and Flash chips have no configuration to manage. For each FPGA and PLD on the FAST PCB, a generic user constraint file (UCF) maps the device pin name to the Verilog port name. Also, a Verilog wrapper for each FPGA provides the top-level port list that corresponds to the names used in the UCF file. This is the base infrastructure required to use the FAST PCB. By bundling the UCF file with a supplied Verilog FPGA wrapper, the end user can use all or a portion of the UCF file and Verilog wrapper to implement a particular architecture. Furthermore, the Verilog wrappers in combination with the UCF files provide some initial guidelines on mapping new prototype architectures to FAST by guiding prototype design partitioning across the FPGAs and defining the inter-FPGA buses.

FAST system clock

The FAST PCB has two global clock sources, a half-size clock oscillator and a header for an external frequency generator, both driving inputs of the PLD. The PLD takes the clock inputs and distributes the system clock to all FPGAs using matched, hand-routed

traces that minimize clock skew at the FPGAs. The PLD uses global buffers to distribute and drive the clock outputs to the FPGAs as shown in Figure 4.7. Figure 4.7 also shows the internal clock (SYS_CLK) generated for all PLD internal logic.

```
module PLD_LED(CRYSTAL_CLK, CFG_SPARE, CP2_GCLK, L1C_GCLK, MEM_GCLK,
               RWC_GCLK);
  ...
  BUFG clk_bufg  (.I(CRYSTAL_CLK), .O(MEM_GCLK));
  BUFG clk_bufg1 (.I(CRYSTAL_CLK), .O(RWC_GCLK));
  BUFG clk_bufg2 (.I(CRYSTAL_CLK), .O(CP2_GCLK[0]));
  BUFG clk_bufg3 (.I(CRYSTAL_CLK), .O(CP2_GCLK[1]));
  BUFG clk_bufg4 (.I(CRYSTAL_CLK), .O(CP2_GCLK[2]));
  BUFG clk_bufg5 (.I(CRYSTAL_CLK), .O(CP2_GCLK[3]));
  BUFG clk_bufg6 (.I(CRYSTAL_CLK), .O(L1C_GCLK[0]));
  BUFG clk_bufg7 (.I(CRYSTAL_CLK), .O(L1C_GCLK[1]));
  BUFG clk_bufg8 (.I(CRYSTAL_CLK), .O(L1C_GCLK[2]));
  BUFG clk_bufg9 (.I(CRYSTAL_CLK), .O(L1C_GCLK[3]));
  BUFG clk_bufg10 (.I(CRYSTAL_CLK), .O(SYS_CLK));
   ...
endmodule

PLD UCF file:
NET "CRYSTAL_CLK" LOC = "p183"; # static CRYSTAL_CLK [CLOCK1,3] [PLD,183]
NET "CP2_GCLK<0>" LOC = "p54"; # dynamic CP2_GCLK0 [CP2_0,A20] [PLD,54]
NET "CP2_GCLK<1>" LOC = "p55"; # dynamic CP2_GCLK1 [CP2_1,A20] [PLD,55]
NET "CP2_GCLK<2>" LOC = "p56"; # dynamic CP2_GCLK2 [CP2_2,A20] [PLD,56]
NET "CP2_GCLK<3>" LOC = "p57"; # dynamic CP2_GCLK3 [CP2_3,A20] [PLD,57]
NET "L1C_GCLK<0>" LOC = "p61"; # dynamic L1C_GCLK0 [L1C0,D21] [PLD,61]
NET "L1C_GCLK<1>" LOC = "p60"; # dynamic L1C_GCLK1 [L1C1,D21] [PLD,60]
NET "L1C_GCLK<2>" LOC = "p59"; # dynamic L1C_GCLK2 [L1C2,D21] [PLD,59]
NET "L1C_GCLK<3>" LOC = "p58"; # dynamic L1C_GCLK3 [L1C3,D21] [PLD,58]
NET "MEM_GCLK" LOC = "p52"; # dynamic MEM_GCLK [MEM1,AP21] [PLD,52]
NET "RWC_GCLK" LOC = "p53"; # dynamic RWC_GCLK [RWC1,AP21] [PLD,53]
```

Figure 4.7. PLD global buffers used to drive the clock to all FPGAs.

Each FPGA uses a global clock pad for the clock input. There are a limited number of I/O pins on each FPGA that can be explicitly used for clock distribution. Associated with each global clock pad is a clock buffer and some type of clock management unit. For the Virtex I, this is a delay locked-loop (DLL) and for the Viretx II, a more sophisticated digital clock management unit (DCM) [109, 122, 123]. Figure 4.8 illustrates the clock distribution from the PLD to all the FPGAs, including the processor tiles (PT) on the FAST PCB.

The interesting thing to note from Figure 4.8 is that the CP2 FPGA generates the four MIPS double frequency clocks (Clk2XSys, Clk2XSmp, Clk2XRd, and Clk2XPhi) and the R3000 uses a DLL to lock onto the supplied clocks and generates the system clock that is distributed back to all of the components in the processor tile. The Verilog modules for

generating the double frequency clocks and the related UCF files are provided in Appendix B.

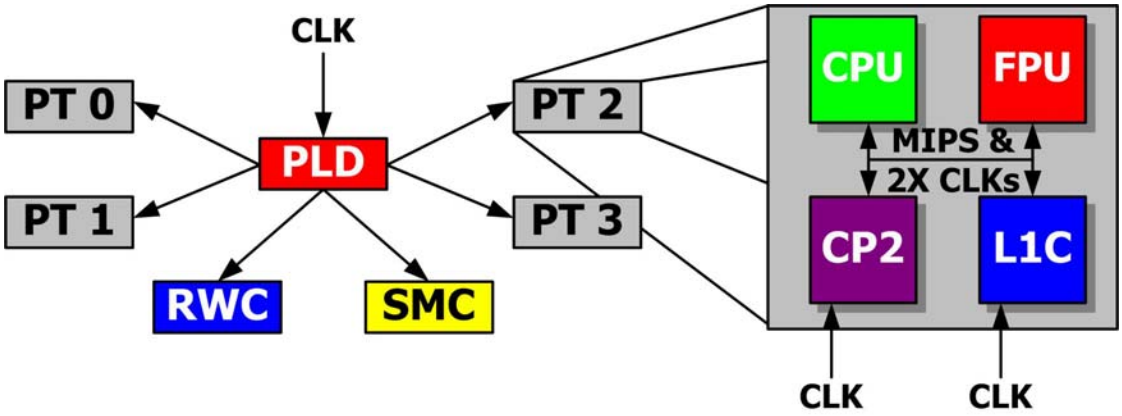


Figure 4.8. FAST clock distribution.

The MIPS R3000 Designer's Guide specifies the delay for all four clocks with respect to the double frequency system clock [54]. For 16 MHz - 25 MHz, there is a constant 6 ns delay between the system clock and the read and sample clocks. A delay line is used in the FPGA to generate the appropriate phase shift. This delay line required both a synthesis primitive to keep the signals from being optimized away, as well as fixed placement of the buffer in the UCF file. The phase clock (Clk2XPhi) used a 180-degree phase shift to generate the appropriate delay required for the system clock frequency range. Thus, a combined hardware (buffer placement and DLL's) and Verilog solution generates the processor tile clocking.

FAST processor initialization

At power-on or system reset, the MIPS R3000 must be initialized. First, the R3000 must lock the clocks generated by the CP2 FPGA and output the MIPS system clock. This requires about 4000 cycles in the FAST initialization phase. Five additional cycles are used at the end of this period to initialize the R3000. The initialization uses the interrupt pins to set the data block refill size, extended cache size, byte order, output tri-state, cacheless operation, data/tag bus drive control, additional phase delay, instruction streaming, CPU mode (R3000 or R2000), partial stores, and multi-processor support [54].

Once the processor is initialized, it jumps to the reset vector address at hexadecimal physical address 0xbfc0_0000. This physical address is in the uncacheable kernel space, kseg1. Figure 4.9 shows how the MIPS memory space is divided into four segments: kuseg, kseg0, kseg1, and kseg2.

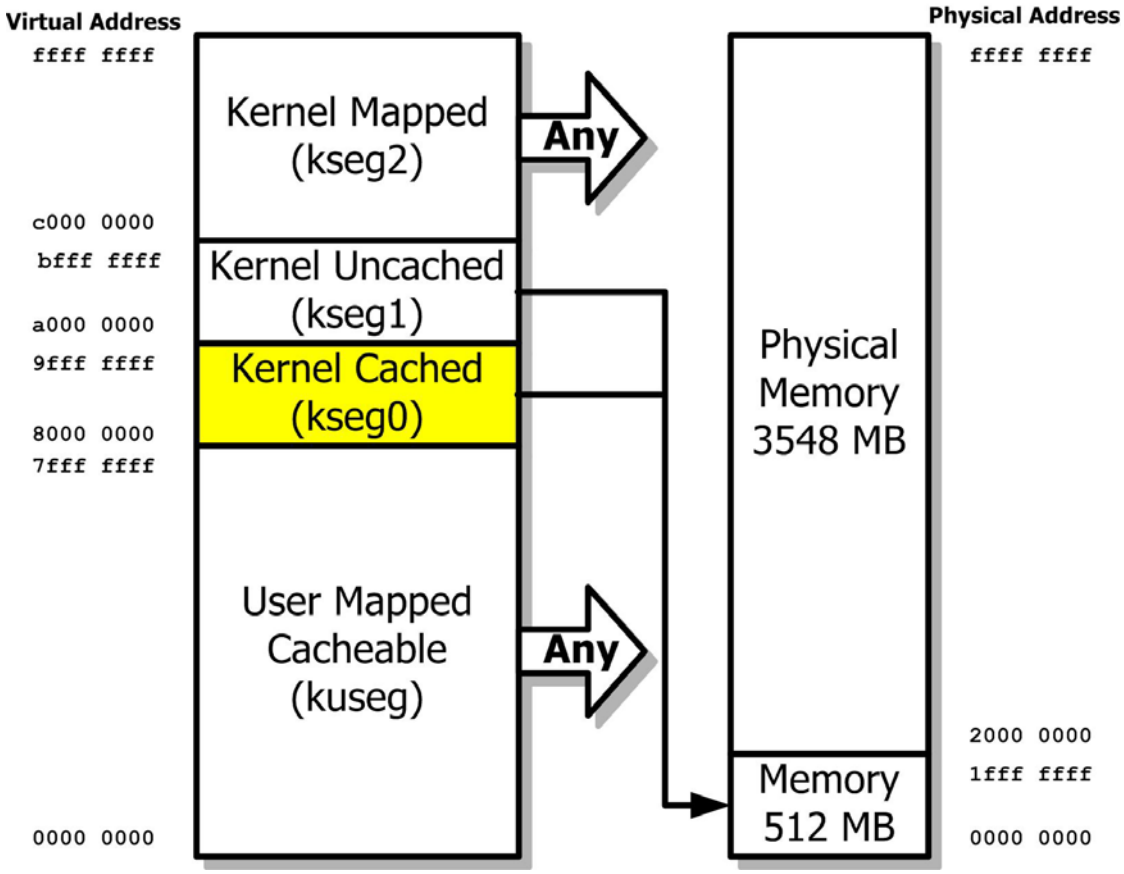


Figure 4.9. 4 GB MIPS address space divided into 4 segments with the initial target memory region, kseg0, highlighted and the physical and virtual address mappings.

Both kuseg and kseg2 enable the translation lookaside buffer (TLB), an address translation mechanism that requires handlers to manage the TLB entries. The processor automatically starts in kseg1, the kernel uncached memory space. In this mode, the processor uses its system bus to fill all memory requests. Each memory request from this space takes at least four cycles to fill, dramatically degrading system performance. The kseg0 memory segment highlighted in Figure 4.9 can use the caches without requiring a TLB. This address space is the most attractive for the initial working prototypes because of the reduced software handler development. Finally, as Figure 4.9 shows, the MIPS R3000 supports protected and unprotected memory accesses by segregating the memory

space into kernel and user space. This allows the R3000 and the FAST system to run modern, multi-tasking operating systems, an option still unavailable to current software-defined processor cores from Altera and Xilinx [7, 117].

Building FAST systems

The MIPS R3000 and R3010 use a shared data and tag bus that serves as the cache and system bus interface. This shared bus also services two different transactions per clock cycle. The instruction address is supplied during phase two of the clock and the instruction comes back on the bus during the subsequent phase one. The R3000 overlaps the data request by providing the data address during phase one and then reading or writing on the bus during the next phase two. If a cache miss occurs, the R3000 holds the address constant and enters at least one stall cycle while it waits for the memory request to be filled. Once the data is presented to the R3000, it transitions to a fix up cycle before continuing to the normal run cycle. Thus, at least four cycles are required to service a cache miss.

The split transaction, dual-purpose tag and data buses provide the external and internal interface to the R3000 and all the coprocessors. It is crucial for this interface to work, but its operation and complexity should be abstracted from the system. For all cache transactions, the L1C FPGA services the requests. When a cache miss occurs, the CP2 handles the memory request by forwarding the address to the higher levels of memory.

The L1C FPGA latches the instruction and data addresses in order to provide a full clock cycle to fulfill the cache request in the correct phase. As mentioned in Section 4.1.3, even though the R3000 operates at a relatively low frequency, meeting its timing requirements was challenging, regardless of using the BRAM or SRAM cache implementation. The CP2 FPGA services all non-cache memory requests for the processor tile, while the R3000 stalls, waiting for the data. Thus, given these predefined interfaces, the FAST user only needs to provide the higher-level memory implementations and interface to a data and address bus with a few control signals for transaction handshaking. An example implementation is provided later in Section 4.2.2 and the corresponding FAST VAL is provided in Appendix B.

4.1.3 Timing, an added dimension of complexity in FAST software

There are two components to a correct design using a hardware prototype: functional correctness and timing correctness. The functional correctness can be validated using a Verilog simulator like ModelSim [72]. To some degree, the timing correctness can be verified using the same simulator, but more detail and timing information must be specified in the model. Unfortunately, not all the details of the FAST PCB are easily specified in the Verilog modules. Furthermore, by specifying copious amounts of timing information, the Verilog modules become part of two different branches of code. The first branch specifies the normal structural Verilog of the design. The second branch specifies all the additional timing information for simulation purposes. These two branches of code add to the system complexity and provide a potential point of inconsistency in the two code branches. Finally, for a large PCB design like FAST, an extreme amount of effort is required to capture all the timing information solely for simulation purposes.

Creating the structural Verilog modules for the FAST PCB is only half of the solution. The FAST Software Toolbox also includes the base user constraint files (UCF) that define the device pin name and port name mappings. The UCF files can also define a wide array of other design constraints ranging from register initialization values to timing constraints. The FAST PCB primarily uses the UCF files to define the connectivity, as provided by the base UCF files, and to define timing constraints by setting the various clocking constraints and signal timing constraints. All four MIPS clocks are defined in the UCF files and then timing groups are used to bundle signals into buses and define timing requirements with respect to the particular MIPS clock of interest.

During the place and route (PAR) phase of building the FPGA bit file, the timing constraints are used to determine if the placement and routing are successful. Any signals or paths that do not meet timing are reported on the automatically generated design summary page and in the timing report. Using the FPGA Editor, the routes that meet the timing constraints can be output to a temporary file that can then be copied into the UCF as a fixed signal placement and route. Fixing the placement and routing makes this

phase of the FPGA bit file build process much faster. This process is done iteratively until all the routes for a particular bus meet their timing constraints.

Adding timing constraints to the UCF files, particularly for the CP2 and L1C FPGAs, presented a challenge that doesn't exist in software simulators. Timing convergence is an added dimension of complexity that was not expected, given that the FPGAs can operate at an order of magnitude higher frequency than the MIPS components. Timing still presented a challenge that had to be addressed outside of the functional correctness of the structural Verilog. Furthermore, timing constraints change depending on whether the design uses internal BRAMs for caches and other memory structures or the external SRAMs for caches. Thus, one set of memory constraints does not fit all applications. Examples of the timing constraints and UCF files can be found in Appendix B for the simple motivating example that will be discussed in Section 4.2.2.

4.2 Prototyping new systems with FAST

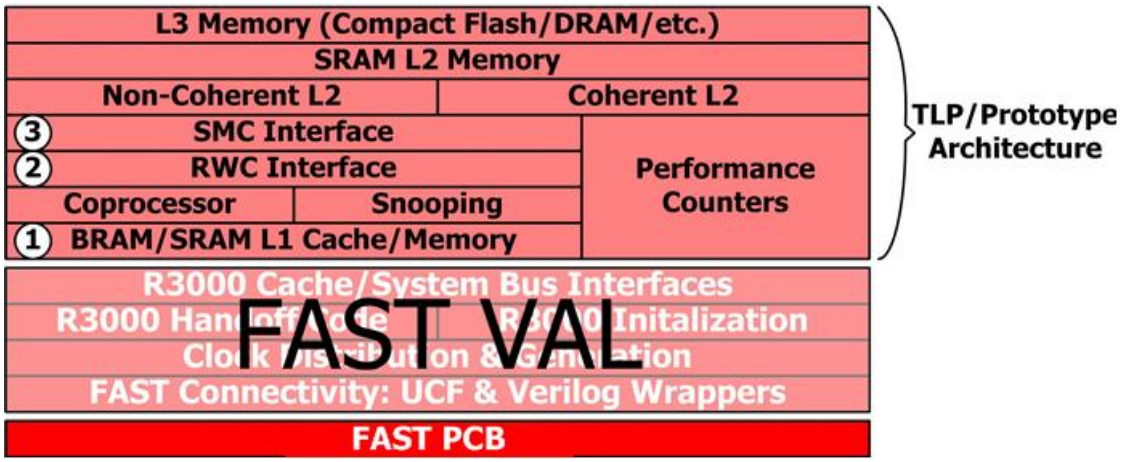


Figure 4.10. Some of the additional Verilog modules required to define a prototype architecture on FAST.

The FAST Software Toolbox provides the interfaces to map TLP or other architectures to the FAST PCB. The FAST VAL provides the minimal FAST functionality. Additional Verilog modules are required to prototype TLP architectures. The FAST Software Toolbox includes these other FAST components built on top of the VAL, as illustrated in Figure 4.10. Only some of these prototype architecture modules were implemented. These modules are provided in Appendix B. In general, the structural

Verilog is very modularized. However, there are some aspects like the performance counters that are implemented in many FPGAs and associated with many Verilog modules, close to the point of use for that counter. Likewise, there is a lot of flexibility with respect to the memory system implementation. Thus, only a simple motivating example is presented that provides the base implementation for future architecture prototypes. The next subsection describes the API that is required to define new TLP architectures. This is followed by a simple example of a 4-way decoupled CMP architecture.

4.2.1 Prototyping TLP architectures on FAST

The FAST VAL provides the base interface to the MIPS R3000 and memory interfaces, both the local caches and the higher levels of memory. However, computer architecture is more than just the processor and primary cache. Thus, FAST provides PCB infrastructure to implement the entire system, enabling full system prototyping. The majority of development for the architecture resides in additional structural Verilog modules for the memory system and additional compute engines. These additional modules provide architecture specific implementations that can be amortized across multiple projects. The small collection of modules in Figure 4.10 scratches the surface with respect to the possible architecture modules for the FAST Software Toolbox. The FAST architecture modules fall into two different categories: PCB specific modules and architecture specific modules. The SRAM interfaces defined in the L1C and SMC are examples of PCB specific Verilog modules. Snooping, coprocessor, and performance counters are examples of architecture specific modules. By developing the FAST Software Toolbox, a software repository is created that can be leveraged across multiple projects and multiple teams.

There are three PCB specific Verilog interfaces that are defined for the FAST Software Toolbox, described from the bottom-up and labeled in Figure 4.10. The first interface is defined in the processor tile, where the L1C FPGA implements the interface between the R3000 and the SRAMs or BRAMs used for local memory. The FAST VAL defines the R3000 to L1C interface and the architecture defines the interface between the L1C and

the SRAMs, as well as any additional translation logic that is required to communicate between the architecture-specific local memory and the R3000 cache interface.

The second interface, labeled "2 – RWC Interface" in Figure 4.10, partially defines the higher level memory interface and the interprocessor communication. The RWC FPGA interface communicates with the processor tiles' CP2 FPGAs to facilitate interprocessor communication and forward memory traffic to the appropriate source. The CP2 FPGA also facilitates additional coprocessor and snooping functionality that is specific to the prototype architecture. The RWC FPGA communicates with the SMC FPGA for access to the L2 SRAMs. Thus, the RWC FPGA can coordinate memory traffic and arbitrate access to the L2 SRAMs via the SMC FPGA. The RWC FPGA can also facilitate interprocessor communication by monitoring and tracking memory addresses and values. This enables implementations of fully coherent multiprocessor systems with the point of coherence managed by the RWC FPGA.

The final PCB specific interface, labeled "3 – SMC Interface" in Figure 4.10, manages the higher levels of memory and expandability. The SMC FPGA controls four banks of L2 SRAMs that can be used as a large shared 64 MB memory or cache. Alternatively, the four banks can be partitioned into processor tile private memory. The SMC also has an 80-pin expansion header that can be used for a variety of purposes. This expansion header can be used for additional levels of memory using a 40-pin connector for up to two Compact Flash daughter cards or IDE hard drive interfaces. Alternatively, the expansion header could be used for a DRAM interface using an adapter daughter card. This expansion header can also be used to create larger FAST compute fabrics by daisy chaining multiple FAST PCBs together. Inter-PCB communication is accomplished use a 64-bit data bus with 16 control bits. Using a simple 1-hot control protocol, 16 FAST PCB's can communicate to create a 64-processor system.

Finally, there are several test suites for reading and writing the L1 and L2 SRAM chips. By using the L1C FPGA for the L1 SRAMs and the SMC FPGA for the L2 SRAMs, the memory chips were tested and the functionality verified. The memory test consisted of writing every memory location with a known value and reading back the value from memory and comparing it to the expected value. This is the first step in defining the

actual interface between the FPGAs and SRAMs. Thus, by testing the SRAMs, all components on the FAST PCB were determined to be functional, even though the actual interface has not been implemented for the L2 SRAMs, only the L1 SRAM cache interface has been implemented.

4.2.2 FAST 4-way decoupled CMP

In order to demonstrate FAST's abilities, a simple 4-way decoupled CMP (FAST CMP 4W-NC) was developed as part of the FAST Software Toolbox. This base functionality demonstrates the potential of FAST as an architecture prototyping platform. Before the rest of the FAST Software Toolbox is described, it is useful to present this motivating example that the rest of the software components build upon. This simple FAST CMP 4W-NC was the result of 10 man-months of software infrastructure development, mainly FAST VAL development.

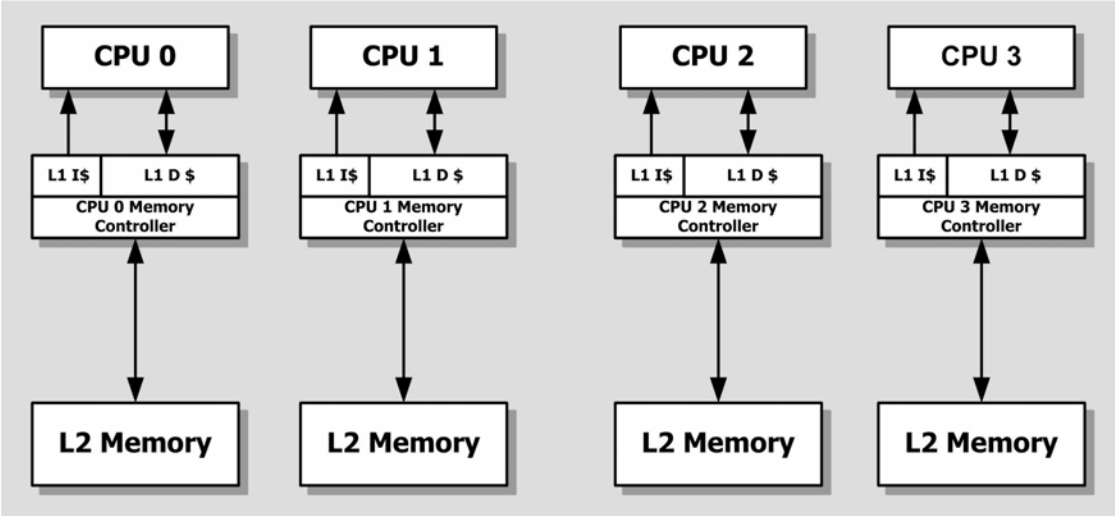


Figure 4.11. High-level architecture of the FAST CMP 4W-NC.

Most of the developed FAST CMP 4W-NC infrastructure can be reused for other projects. Thus, the development time can be amortized over multiple prototypes. The FAST CMP 4W-NC is a simple 4-way CMP, with each processor having a private L1 data and instruction cache and a private L2 memory, shown in Figure 4.11. This subsection provides a bottom-up description of the FAST CMP 4W-NC.

The processor tile contains the majority of this decoupled system. This base implementation has key components that would exist in most other architectures. Given the FAST VAL layer, the FAST CMP 4W-NC required a local data and instruction cache. The first cache implementations used block RAMs (BRAMs) in the L1C FPGA. Using the small amount of memory in the FPGA reduced the integration effort and accelerated the FAST CMP 4W-NC development time. The L1C FPGA has 128 Kb (bits) of BRAM. FAST CMP 4W-NC required 4 memory arrays to implement the data and instruction caches for each processor tile, 1 memory array for the data portion and 1 memory array for the tag portion of each cache. The Virtex I provides 4096 bit BRAM primitives that can be arranged in a variety of configures. The data and instruction caches contain 1024 32-bit words.

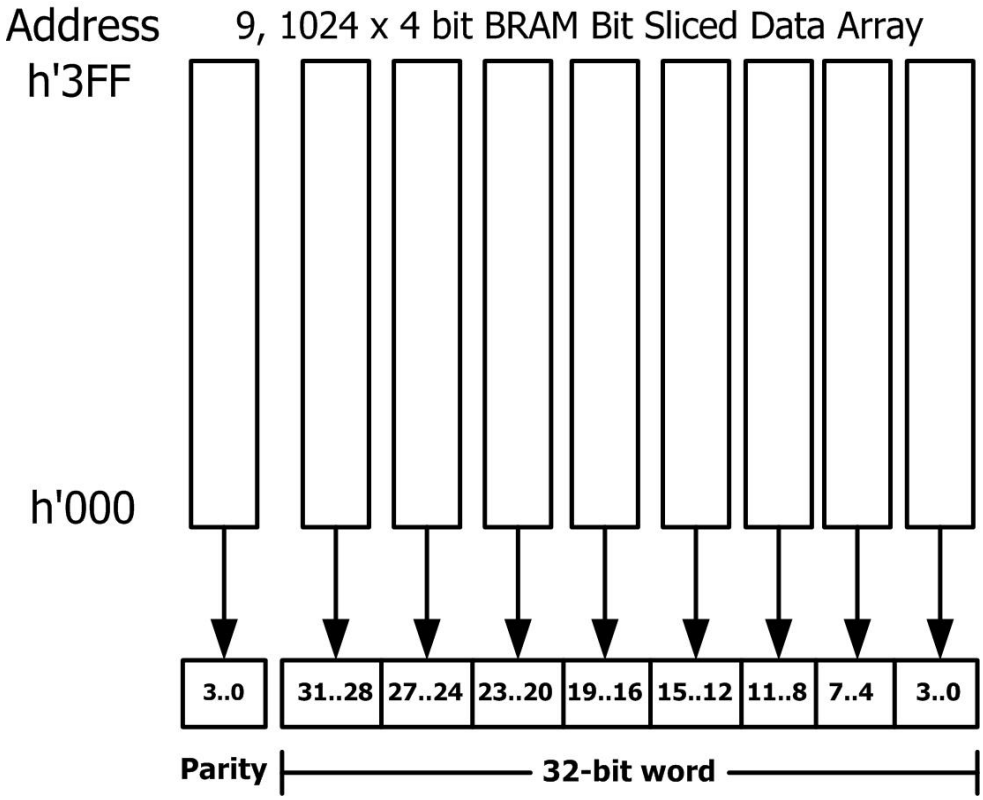


Figure 4.12. The cache data array using 9, 1024 X 4-bit BRAMs, including 4 bits for parity.

In order to minimize performance penalties, the memory arrays are composed of multiple 4-bit wide BRAMs, as shown in Figure 4.12. Thus, no multiplexing is required to address the BRAMs for reading or writing data because each BRAM in the memory array

contributes a 4-bit portion of the data. As Figure 4.12 shows, the cache memory array implementations included an additional 4 bits for parity. The memory array primitive was used for each data and tag array in each cache. This very simple array supported a direct mapped cache that provides single cycle memory access latency. These cache sizes are fine for embedded processor research, but not for general purpose or scientific workloads.

The second stage of the cache development involved using SRAM chips, instead of BRAM arrays, to implement the L1 caches. By using the SRAM chips, the L1 cache capacity for the data and tag arrays increases by two orders of magnitude, to 256 KB plus parity bits. Again, a single SRAM chip Verilog interface was implemented and used for both the data and instruction caches. As would be expected, the timing and control signals are the only Verilog interface definitions required for the SRAM data and instruction caches.

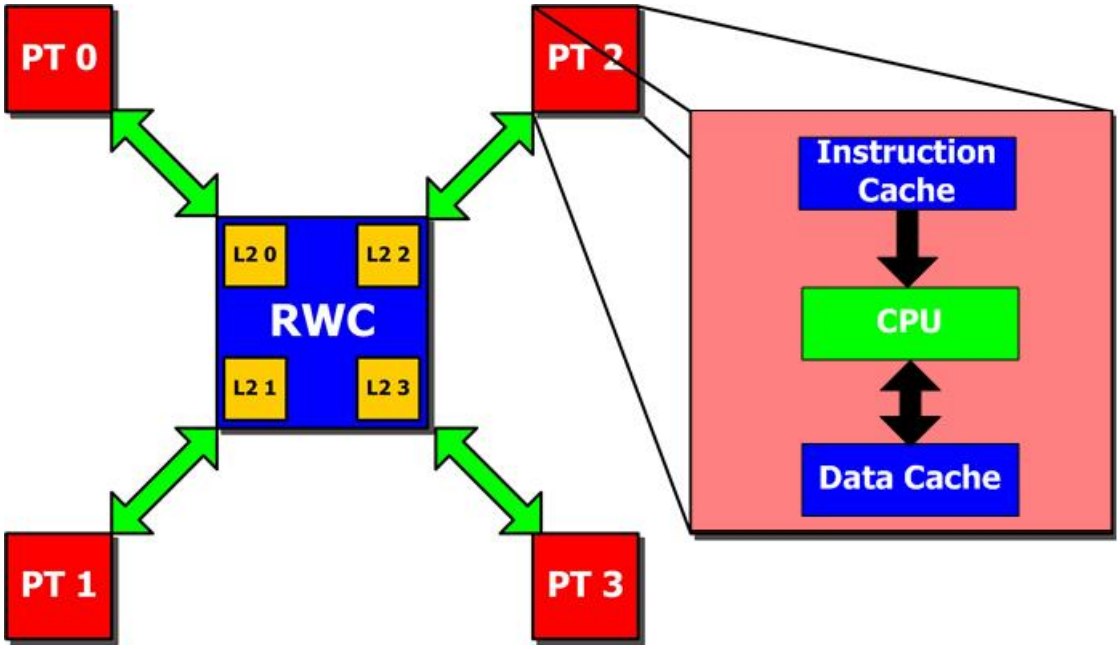


Figure 4.13. FAST 4-way decoupled CMP.

The next component of the FAST CMP 4W-NC is the private per-tile L2 memory. The L2 memory is implemented in the RWC using four BRAMs blocks, one block for each processor tile. Each BRAM block is generated using Xilinx Core Generator™. This tool specifies all the configuration details for BRAMs including: BRAM width, BRAM number

of entries, handshaking, BRAM primitive, and initialization file, to name a few. Each L2 memory is 40 KB or 8096 x 36 bits. The BRAM blocks use the 512 X 36 bit primitive, which requires multiplexers to select the BRAM primitive for reading or writing. This BRAM configuration is not optimized for performance, but it allows the memory block to use the parity bits in the primitive. If it were an issue, bit slicing could be used for better performance.

By developing a simple motivating example for FAST, as illustrated by the FAST CMP 4W-NC mapping in Figure 4.13, the full potential of the system can be demonstrated. Chapter 5 describes the performance and behavior of applications running on this simple architecture. This simple system integrates the L1 SRAMs, CP2 FPGA, L1C FPGA, and RWC FPGA. Even though this system is not cache coherent, communication between the CP2 and the RWC demonstrates that the additional coherence functionality is possible to implement. Thus, time and software implementation effort are the only limiting factors to mapping new architectures onto the FAST system.

4.3 FAST OS, drivers & APIs

Moving up a level in the FAST Software Toolbox moves out of the structured Verilog modules into assembly language and higher-level languages. Once the prototype architecture is well defined and implemented in Verilog, software development is required to make the whole system work. Figure 4.14 shows the FAST software stack up to this point. Before applications can be run on the prototype architecture, the operating system (OS) or at least some portion of the OS must work to run the applications.

The MIPS R3000 contains all the necessary components to run a multitasking operating system (OS). This factor is currently missing from the Xilinx and Altera software-defined processor cores [7, 117]. However, the LEON software-defined processor core is SPARC V8 compliant [38] with all the necessary components to run an OS, like Linux [94]. Getting applications to run on FAST was the top priority. As a result, porting a complete OS like Linux was not on the critical path of getting applications running on FAST because it is very time consuming. Having an OS running on FAST would create a

tipping point for future architecture and subsequent software development, making the system much more attractive to future development.

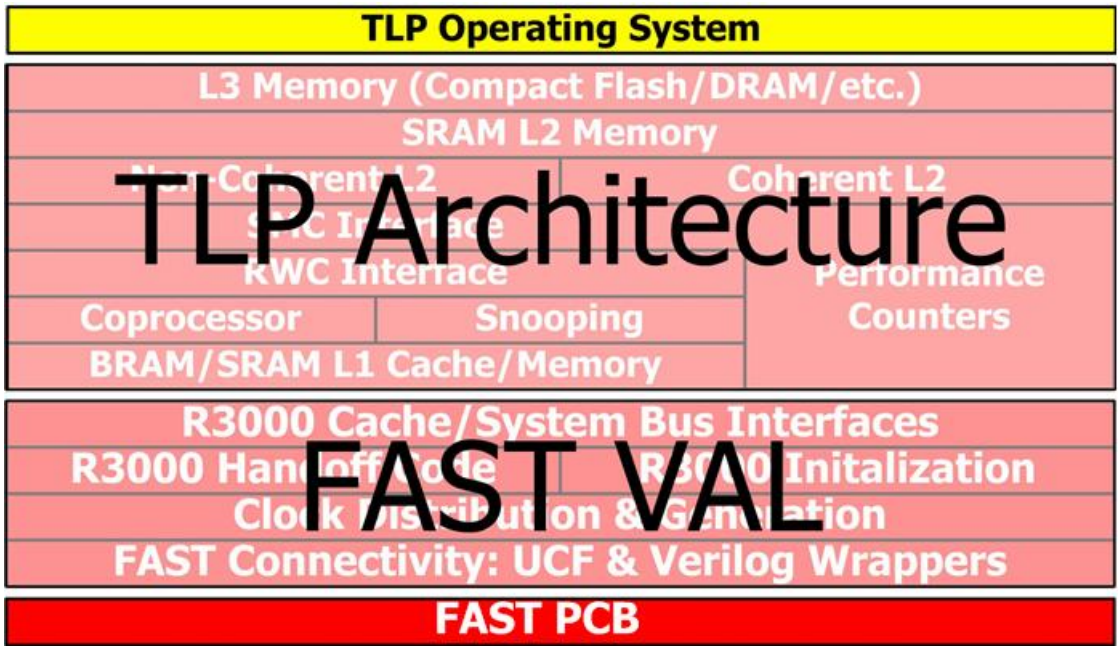


Figure 4.14. FAST software stack up to the operating system.

Given the limited resources, only minimal OS functionality has been implemented for FAST. First, the software development environment is presented, followed by the OS support functions required for FAST to run simple applications. Potential OS's that can be ported to run on FAST and be part of the FAST Software Toolbox follows this discussion.

4.3.1 FAST software development tools

The FAST software development environment changes when moving from structural Verilog to assembly language or higher-level languages like C. The text editor is the main instrument of choice, followed by a series of free tools combined with scripts to generate memory images for the FAST system. Three steps are required to build applications for FAST: the compiler, linker, and post processing tools create the memory image that is downloaded to the FAST PCB.

There are a few options to use for the software tool chain. Because FAST is based on the MIPS R3000 and the MIPS I instruction set architecture (ISA), the natural tool choice is

the native tools running on MIPS-based SGI machines. Using various compiler and linker flags, the resulting binaries can run natively on the SGI or, with some minor address mapping modifications, on FAST.

The other alternative is using a cross-compilation environment. MIPS Technologies, Inc. provides a software development environment (SDE) [74] that works with Cygwin [35], a Linux emulation environment, for developing MIPS embedded applications. This environment supplies a compiler (gcc), linker (ld), and various other tools required for building MIPS applications. The FAST Software Toolbox uses gcc to build the object files for applications targeting the MIPS R3000 and the MIPS I ISA.

The linker combines all of the object files and sets the starting address of the application. The MIPS SDE had the greatest flexibility with regard to setting the starting address and not compiling with libc or other startup files like crt.o. The FAST Software Toolbox contains its own start and ext routine, discussed later in this section. The linker starts all applications at hexadecimal physical address 0x8000_0000, in the cacheable kernel space. The linker combines all of the object files and partitions the memory space for the data and instruction or text segments.

Post processing is the final stage in creating a memory map of the application for FAST. The memory image is loaded into the highest level of the memory hierarchy, L2 memory for the FAST CMP 4W-NC, to be demand missed into the L1 cache. The binary created by the linker is disassembled and then processed with some text processing utilities like sed and awk.

Figure 4.15 shows the process used to build the memory map file for the L2 memory in the FPGA. First, the binary is created using gcc and ld, labeled "1" in Figure 4.15. Next, the objdump utility disassembles the binary into the hexadecimal representation and the ASCII instructions. In step 3, Sed is used to filter out everything but the hexadecimal instructions. Finally, in step 4, Gawk, a variant of awk, is used to create the COE format file. Alternatively, step 4 can produce a Verilog initialization file formatted for the older Virtex I BRAM modules, if desired. The resulting COE file can be added to the Xilinx build environment to initialize the L2 BRAM memory or used with Data2Mem to update the BRAM contents of an existing FPGA programming bit file.

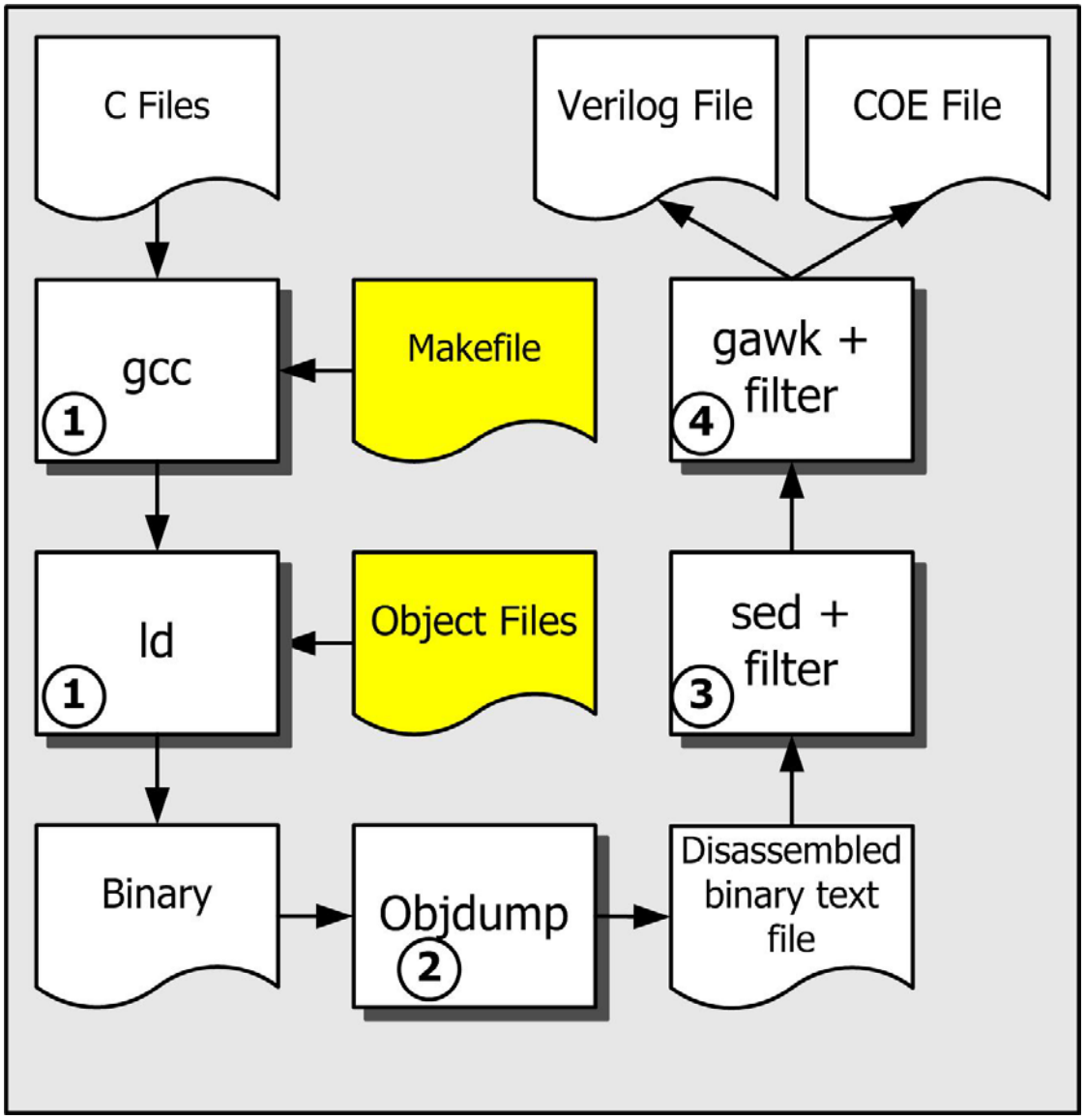


Figure 4.15. Make script for building FAST memory map files or COE files.

4.3.2 FAST OS support

There are two essential components required to run applications, a start routine and an exit routine. The start return sets up the memory space and processor to run applications. The exit routine signals successful completion of the application and can also be used to undo the state setup in the start routine. Together, these routines enter and exit the main function of the application and are linked at compile time by the linker. The start routine is described first, followed by the exit routine.

```
      la      $1, Setup       # load address of Setup label
      jr      $1              # Jump to Setup address in kseg0
      nop

Setup:
      la      $gp,_gp         # setting up the GP using Macro
      nop
      lui     $sp,0x8002      # setting up SP at 8002 8000
      ori     $sp,$sp,0x0000
      nop
      addiu   $sp,$sp,-8      # making some room on the stack
      sw      $sp,0($sp)      # storing SP
      move    $fp,$sp         # setting up FP
      nop
      move    $sp,$fp         # restoring SP
      jal     main            # jumping to main
```

Figure 4.16. R3000 handoff code for initializing pointers and jumping to the start of a program at main().

The R3000 uses a small sequence of assembly code to jump from its default start location in uncached kernel space, kseg1, to kseg0, where the local cache can fill instruction and data requests without any system performance degradation. Thus, each processor running in kseg0 memory space can execute up to one instruction per cycle (IPC). Figure 4.16 provides the annotated assembly code that jumps into kseg0, initializes the stack and global pointers, and jumps to the beginning of main(). The linker sets the base address of the application to hexadecimal physical address 0x8000_000, the start of kseg0. Thus, when the application is linked with this start code, all the addresses are resolved and set when the binary is created.

The exit routine jumps to a particular address in kseg1 to halt the processor. The jump address depends on the successful completion of the application. The main application passes a "1" upon successful completion of the application or "-1" if the application computes the wrong answer. The routine in Figure 4.17 builds a loop in the target address space and then jumps to that loop address. If the program executes successfully, it jumps to hexadecimal physical address 0xaff8_2ffc and results in the processor halting at that address. If the program exits with an incorrect solution, it jumps to hexadecimal physical address 0xaff8_34fc and the processor halts. Using the JTAG static observations tools, the halted processor and the address on the data bus can be observed to find out if the program executed successfully.

```
        nop
        addiu   $sp,$sp, -16          # setting up stack
        sw      $s8,8($sp)
        move    $s8,$sp
        sw      $a0,16($s8)           #storing function argument on stack
        lw      $v0,16($s8)           #loading argument into local register
        nop
        sw      $v0,0($s8)      #swap registers
        lw      $v1,16($s8)
        li      $v0,1
        bne     $v1,$v0,BadEx  # compare exit argument for success
        nop
        lui     $v1, 0xaff8    # jumping into uncached kernel space
        ori     $v1,$v1,0x2ffc # construct address to stall on
        lui     $v0, 0x1000    # construct branch back 1 instr
        ori     $v0,$v0,0xffff
        sw      $v0,4($v1)     # store near jump target
        jr      $v1            # Jump to Address that stalls the processor
        nop
BadEx:
        li      $v0,-1
        sw      $v0,0($s8)
        lui     $v1, 0xaff8    # jumping into uncached kernel space
        ori     $v1,$v1,0x34fc # construct address to stall on
        lui     $v0, 0x1000    # construct branch back 1 instr
        ori     $v0,$v0,0xffff
        sw      $v0,4($v1)     # store near jump target
        jr      $v1            # Jump to Address that stalls the processor
        nop
```

Figure 4.17. R3000 exit code with jump target address determined by program execution.

4.3.3 FAST OS options

There are several operating system (OS) options for the FAST PCB. The options range from a batch OS like PMON to a fully functional OS like Linux or Irix. Because the R3000 has been used as an embedded processor, there are many real-time OS and cooperative OS options. PMON and Linux are most appealing because they are open source, well documented and, as a result, more easily ported because similar ports for other MIPS processors exist.

The PMON system monitor produced by LSI Logic and Algorithmics can provide low-level OS support for applications executing on the FAST PCB [15]. PMON provides application monitoring and debugging for MIPS-based evaluation boards; batch operating system functionality can be added to FAST by modifying PMON [15]. PMON provides the gdb-like interface for application debugging that makes FAST's processor state and memory state visible to the user in interactive execution mode. The FAST PCB would also use PMON to provide limited OS support for libc functions when any of the MIPS processors execute SYSCALL instructions. If I/O with the host workstation is necessary, PMON drivers could use Ethernet through the embedded microcontroller

(RCM3200) on the FAST PCB to handle the TCP/IP protocol details. The RCM3200 transmits messages from the FAST PCB to the host using both simple terminal-style I/O and using special messages when OS functionality requires external support for functions like file I/O. These "OS" messages reduce the complexity of the on-board FAST OS significantly while still enabling simulation of real-world applications that require significant OS support. For example, in the case of file I/O, the on-board OS only needs to communicate file handles and buffers associated with read() and write() calls between the application and host interface, while leaving the details of disk management solely to the host's operating system.

Conversely, the FAST system could be completely independent of a host system by porting a full service OS like Linux. There are MIPS ports to Linux for older SGI machines that could be leveraged for the FAST system. Porting Linux to the FAST PCB requires more effort, particularly with regard to driver porting and the console. However, it may be possible to leverage pre-existing memory controllers and hard drive controllers, making the FAST system a full system prototyping platform. Previous versions of the MIPS Linux kernel for SGI Indy's required less than 3 MB of memory, easily fitting in the Flash memory. By porting Linux to the FAST PCB, the FAST PCB would have no application restrictions. FAST's 32-bit processors do, however, restrict the use of large applications or the use of a large address space.

4.4 FAST applications

The FAST Software Toolbox applications fall into two categories: small directed diagnostics and regular applications. The small diagnostics allow focused and repeated activation of a particular function in the FAST system. Initially, the directed diagnostics were used to verify the memory system operation. Using characteristics that normally a computer architect would try to avoid, like address aliasing, to test the memory system operation, directed diagnostics are very small snippets of assembly code, generally 2-8 instructions.

The regular applications come from the Stanford Small Benchmark Suite. These are simple, well-understood benchmarks that come from the same era as the MIPS R3000.

Together, these directed diagnostics and applications complete the FAST Software Toolbox. A *mature* FAST system is the combination of the Verilog modules, OS functionality, and the applications. Together, this combination makes a turnkey system that minimizes the effort to create new prototyping systems by leverage pre-existing hardware and software.

The directed diagnostics range from 2-8 assembly instructions. These small programs test particular FAST functionality, from jumping between address spaces to testing loads and stores. There are about 20 small directed diagnostics used to test various scenarios with respect to the R3000 and the memory subsystems. The directed diagnostics fall into three broad categories: control flow instructions (branches and jumps), instruction cache tests, and data cache tests.

The control flow was tested first. Simple branch instructions were used to verify the loop address running on the processor. Once branch instructions were verified, the simple tests moved on to more complex jump instructions. The R3000 starts off in uncacheable kseg1 space and, based on a register value address, the processor can jump into any other address space, in particular cacheable kseg0 space. By jumping to another address space, the processor was able to access the cache, facilitating cache interface testing and not just the system interface testing that occurred in kseg1. The final control flow test related to function calls and the jump and link *(jal)* instruction. This test had two loops in the test, the first loop would execute if the *jal* instruction failed, thereby falling through to execute the first loop. If the *jal* instruction succeeded, the second loop would be executed at a different address location that could be observed using the J-SCAN tool. Thus, the static observation of infinite loops determined that all the control flow in the FAST system worked.

The instruction cache directed diagnostic tests were the next set of tests to develop and use. The instruction cache functionality could be tested once the processor executed in an address space that actually used caches, like kseg0. The instruction fetch was tested both from the L1 cache interface perspective and the system interface. From the control flow tests, it was clear that instructions could be fetched using the system bus. For some of the instruction cache tests, L1 cache aliasing was used to force valid entries out of the

L1 cache and thus make the CPU swap out instructions via the system interface. These simple diagnostics leveraged self-modifying code segments to construct instructions in memory that are later fetched based on the control flow of the diagnostic.

Once the instruction cache interface was established, the data cache interfaces could be tested. There are two components to the data cache. The first component verifies that data is written and read to the data cache. The second component exercises the write-through nature of the data cache. The data cache interface was tested using the diagnostics that constructed addresses or data in the data cache to direct the control flow of the processor. Data values could be used for branches or used to jump into particular address spaces.

The FAST Software Toolbox also includes the Stanford Small Benchmark Suite [49]. The Stanford Small Benchmark Suite is a small benchmark suite that was assembled by John Hennessy and Peter Nye around the same time period of the MIPS R3000 processors. The benchmark suite contains ten applications, eight integer benchmarks and two floating-point benchmarks. The original suite measured the execution time in milliseconds for each benchmark in the suite. The Stanford Small Benchmark Suite includes the following programs:

- Permute A tightly recursive permutation program.
- Towers The canonical Towers of Hanoi problem.
- Queens The eight Queens chess problem solved 50 times.
- Integer MM Two 2-D integer matrices multiplied together.
- FP MM Two 2-D floating-point matrices multiplied together.
- Puzzle A compute bound program.
- Quicksort An array sorted using the quicksort algorithm.
- Bubblesort An array sorted using the bubblesort algorithm.

- Treesort An array sorted using the treesort algorithm.

- FFT A floating-point Fast Fourier Transform program.

For simplicity, the FAST Software Toolbox focused on using integer programs that did not require libc or floating-point support because that would require more FAST infrastructure development time.

4.5 Putting all the FAST software together

This chapter has presented 4 out of the 5 software layers presented in Figure 4.1. The fifth layer, the driver and API layer, is specific to the particular architecture being mapped to the FAST system and therefore, was not discussed in this chapter in any detail.

The results to be presented in Chapter 5 are based on the simple FAST CMP 4W-NC that demonstrates FAST's potential as a system prototyping platform. Before moving to Chapter 5, it is beneficial to summarize the steps required to build and run applications for the FAST CMP 4W-NC on the FAST prototyping substrate. These steps apply to any system mapped to FAST. There are five steps required to run an application on the FAST PCB. These five steps are illustrated in Figure 4.18 with boundaries defined in the figure to clearly demarcate the steps.

The first step is to develop the application to run on FAST. In the example shown in Figure 4.18, the Towers of Hanoi application is used. This is a simple ANSI C application that can be validated on another system to make sure it operates correctly. Porting it to the FAST CMP 4W-NC requires removing any system calls and libc functions, like malloc or file I/O.

The second step compiles and links the application with any other object files. The FAST Software Toolbox supplies the makefile and various helper object files shown as shaded documents in Figure 4.18. Using the MIPS SDE, a binary is created that will be executed on the FAST CMP 4W-NC.

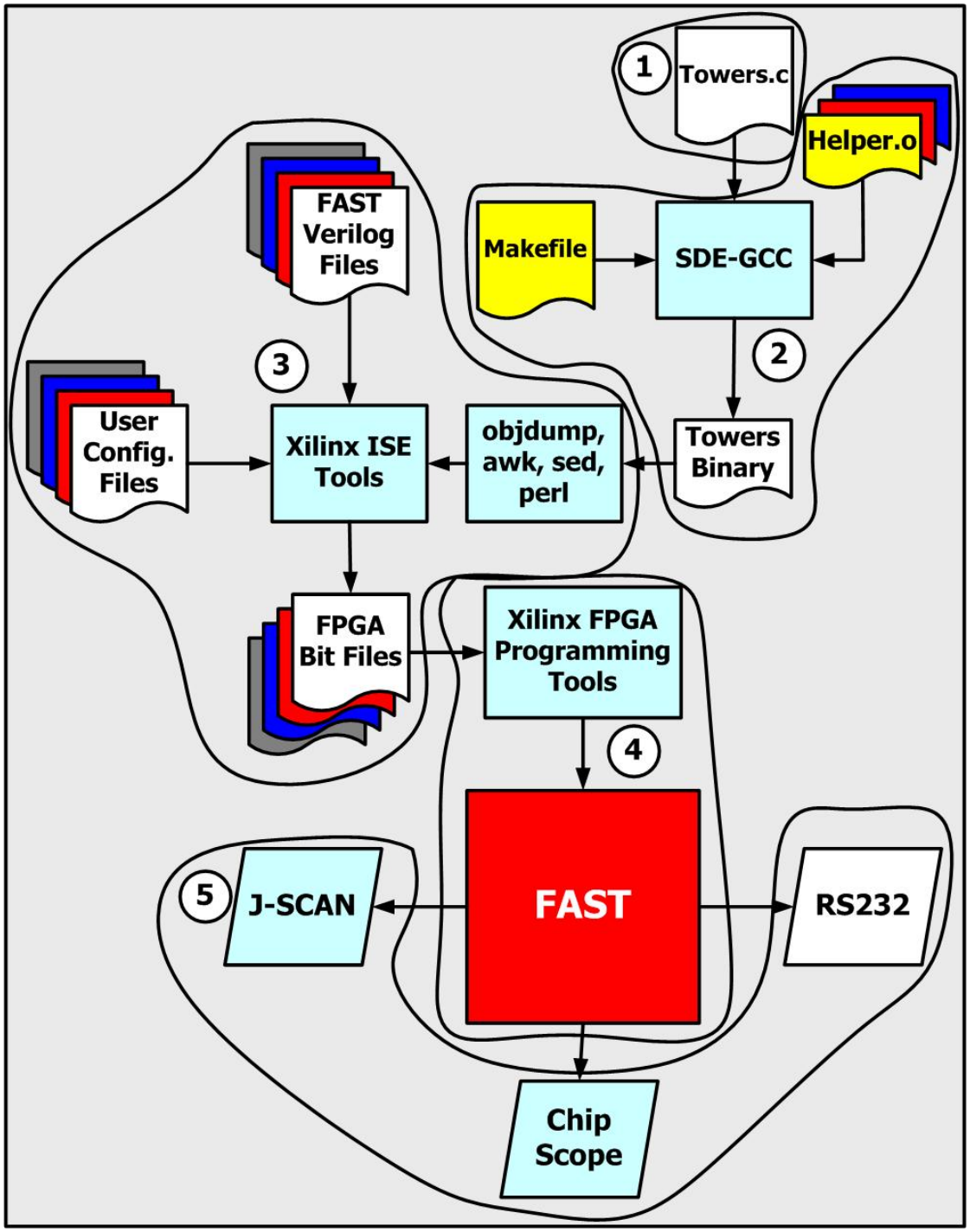


Figure 4.18. Five steps required to run and observe applications on FAST: (1) Develop the application, (2) Build the application binary, (3) Build the FPGA programming bit files, (4) Program the FAST FPGAs, and (5) Observe the steady-state application behavior.

The third step combines the FAST VAL, UCF files, and TLP architecture Verilog files, with the memory map file to produce a collection of FPGA programming bit files. The

application binary created in step 2 must be disassembled and transformed into a memory map file that the Xilinx tools use to initialize the BRAM memory structures. For the FAST CMP 4W-NC, only three FPGA programming bit files were required: CP2, L1C, and RWC. This 4-way decoupled system had fast on-chip L2 memory latencies like a CMP, but could be easily changed to model the slower off-chip latencies of multiprocessor systems.

The fourth step takes the FPGA programming bit files and uses the Xilinx iMPACT tool to download the bit files to the FPGAs using a parallel port driven JTAG controller. The RWC FPGA is programmed first to set up the L2 memories and their contents. These memories use active-high logic and handshaking to communicate. Next, the L1C FPGA is programmed with either blank BRAM L1 cache modules or an SRAM L1 cache interface. With the memory systems set up, the final FPGA to be programmed is the CP2. Once the CP2 FPGA is programmed, the R3000 is initialized. This takes approximately 4000 clock cycles. After initialization, the instructions are demand missed into the L1 cache from the L2 Memory in the RWC FPGA.

The fifth step uses the JTAG tools or RS-232 interface to observe the steady-state behavior of the application. The application uses an exit routine that jumps to a particular address in kseg1. This address begins an infinite loop; furthermore, the L2 memory system is programmed to stop fulfilling in memory requests for that particular address range. Thus, the application steady state is either a halted processor waiting for a memory request to be completed, or an application in an infinite loop. Initially, the infinite loop was used because it required no hardware support. However, halting the processor is much easier and enabled a mechanism to start and stop the statistics collection for the applications.

These five steps apply to all applications and architectures that are mapped to the FAST substrate. If a collection of FPGA programming bit files exists and only the BRAM contents needs to be updated, step 3 can be replaced with the Data2Mem scripts, updating the BRAM memory contents, but nothing else. For more details, all of the scripts and wrapper files for the FAST Software Toolbox are provided in Appendix B.

4.6 FAST soft lessons learned

The software development for FAST falls into three categories: morphware, software, and tools. In general, the morphware and software development is not difficult. Creating Verilog code for the FAST VAL is straightforward and not conceptually complex. The interfaces for the FAST VAL are well defined. Unfortunately, the MIPS R3000 is not always well documented resulting in some implementation ambiguity, which has not dramatically hindered progress [54]. The FAST CMP 4W-NC Verilog demonstrated FAST's system prototyping potential. The FAST CMP 4W-NC development was much easier because of the initial infrastructure development for FAST VAL.

The simple FAST CMP 4W-NC provided a hardware platform for running standard MIPS I ISA applications. The main software development limitation is no sophisticated FAST application debugging capabilities similar to GDB [13]. Applications must be debugged on a MIPS-based machine first. Simple ANSI C applications run on FAST by removing all OS supported application components such as library functions and system calls. Even though FAST currently cannot be used for application debugging, other systems exist that can be used, such as MIPS-based emulators. Software development follows normal practices and is done on other available MIPS-based system and then recompiled for the FAST system. The applications are rather rudimentary because of the lack of an OS running on FAST. FAST is theoretically able to run multitasking OS's, but there has been no OS porting effort yet.

There is a large learning curve for the Xilinx tools beyond doing any Verilog programming. Most of the Verilog development time is dominated by FPGA and tool configuration issues. FPGAs have another layer of configuration that maps FPGA resources to Verilog components or modules. Furthermore, timing constraints are also required in order to meet the timing requirements of the MIPS R3000. Finally, depending on the resource utilization, sometimes the Xilinx tools will automatically add components, like buffers, to the design that improve performance. Unfortunately, as the FPGA device utilization increases, no components are automatically added, resulting in a broken design that previously worked. This provides many hours of hardware debugging that results in FPGA constraint updates. This can affect non-critical path circuits that

operate as slow as 10 KHz, making the timing dimension a crucial additional design dimension. However, it should be noted that providing better tools and various software libraries for new designs would dramatically decrease the learning curve and development time.

The software infrastructure for FAST faces the normal software development hurdles. FAST requires software development from Verilog to assembly language and higher-level languages. The software development tools, like emacs or other text editors, are pretty mature for the software development. Unfortunately, the additional FPGA tools required to meet timing are still immature and require significant effort to track down timing errors. An easy to use FPGA constraint and timing tool that post-processes the FPGA design could greatly reduce design effort and hardware debugging. With this type of tool, FPGA development will become much easier. Designing and building a PCB is difficult. Likewise, mapping designs with timing constraints to an FPGA or collection of FPGAs is equally challenging. The FAST design effort has been dominated by manipulating the tools to meet the modest timing constraints and not actual software development, be it Verilog, assembly language, or ANSI C.

Chapter 5

FAST Prototype Results

This chapter outlines the performance of the FAST CMP 4W-NC running real benchmarks. Previous chapters have described all of the foundational building blocks, but FAST is not useful unless it can run and characterize applications. This chapter describes the various facets of collecting data from FAST using the FAST CMP 4W-NC as an example.

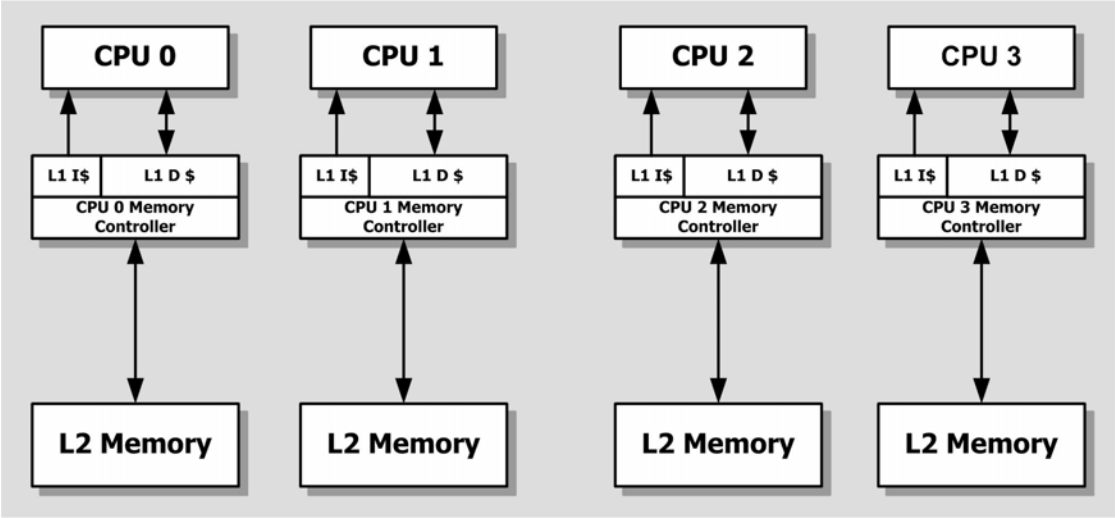


Figure 5.1. 4-way decoupled FAST CMP 4W-NC.

The FAST CMP 4W-NC is a decoupled 4-way CMP with private L1 data and instruction caches and private L2 memory. Figure 5.1 provides a high-level overview of the FAST CMP 4W-NC architecture. The FAST CMP 4W-NC can use 4 KB BRAM L1 data array caches or 256 KB SRAM L1 data array caches. In either L1 cache size case, the caches were direct-mapped and used a write-through policy. All of the data reported in this dissertation used the four 256KB SRAM chips for the data and tag arrays in the instruction and data caches. The private L2 memories are 32 KB (although 72 KB was available). The level of integration in the FAST CMP 4W-NC demonstrates that a variety of other more complex architectures can be mapped to FAST.

Before diving into the results, it is useful to understand the visibility, flexibility, performance architecture and performance counters in the FAST system. This provides a framework for understanding what is possible and not possible using FAST for data collection. The visibility refers to one of the most important aspects of any prototyping system, the ability to record or observe a number of events based on triggers. Software simulators traditionally provide the most visibility and one of the goals of FAST is to mimic that level of transparency. Second, the flexibility of the FAST system to collect data is also an important aspect of any prototyping system. Combining the visibility and flexibility creates a performance architecture that defines what and how data can be collected. Finally, actual FAST performance counters are presented with a description of other performance counter implementation options. With the framework defined, the results of running the Stanford Small Benchmark Suite on the FAST CMP 4W-NC are presented, followed with conclusions.

5.1 FAST data collection

FAST provides some data collection primitives that can be instantiated as Verilog modules that count synchronous events in any FGPA. This is very useful, but FAST is not limited to this sort of data collection. FAST can also stream data off the PCB to be stored on a host system. Furthermore, data collection is not limited to simple counters. FAST was designed to be flexible with respect to the performance monitor because of the amount of system visibility. The FPGAs enable visibility inside any FPGA structure or interface, facilitating FAST data collection across several domains, from canonical processor architecture performance counters to component and system power monitoring. FAST combines visibility and flexibility to create a performance architecture suitable for a variety of applications and only limited by the user's imagination and of course, FPGA resources.

5.1.1 FAST visibility

Software simulators provide the gold standard with respect to transparency or visibility within a system. Anything within the software simulator can be probed, monitored, or saved in a file for post processing. Furthermore, events that cannot be observed on real hardware can be observed or synthesized using software simulators. FAST brings this

high level of detail back into the hardware prototyping systems. Projects like RPM [9, 33] spurred on the visibility in real systems by adding FPGAs in order to increase system transparency. The FPGAs were programmed to monitor bus traffic, particular events or combinations of events in the system. These initial systems increased visibility down to the level of the processor, while the processor or other components in the system still remained opaque.

FAST illuminates the hardware prototype by enabling visibility at all levels of the system, save the simple processors. Thus, performance counters can be placed at the point of use and interfaced with the system, or can be completely external to the system. System performance counters are counters that can be accessed by the system software, like the operating system (OS) or other APIs. External performance counters monitor or record information that can only be accessed by software or hardware agents that are external to the system prototype. Examples of the latter are service processors used as system monitors or JTAG interfaces for diagnostics. By programming the FPGAs in FAST, the user has the option of adding complex monitors as they see fit.

Ideally, the visibility exists at multiple levels with a particular focus on increasing visibility at run-time and making it accessible through the system interface. As a mature system, FAST would provide these mechanisms through the OS much like modern-day processors. However, this would require significant infrastructure development in the structural Verilog and higher software layers like the OS and other drivers and API's.

More likely, and as we will show, it is much easier to implement performance counters that provide the same visibility, but are accessed by external agents. The downside is that the system cannot use the data on a real-time basis, but must incorporate performance data much like current profiling is done, in an iterative tuning process. FAST uses external agents like ChipScope from Xilinx to monitor various FPGA-internal modules [113]. ChipScope provides the visibility via the JTAG interface to record and observe events. Likewise, the chip boundaries can be observed using a logic analyzer for very fine-grain dynamic observations. If this level of detail is not required, static JTAG observation tools like J-SCAN can provide steady-state application behavior at a coarse granularity [67]. Finally, simple bi-directional interfaces like RS-232 can be implemented

as an interface to internal counters. This provides a simple, fast, and easy way to collect single events or to stream low bandwidth data from the FAST system.

With regard to the memory system or any other component that uses the FPGA as an interface, FAST provides levels of visibility similar to software simulators. FAST does not provide a window into the simple R3000, but many of the R3000 performance statistics can be derived from the FPGAs that interface to the R3000. The only other major restriction is the limited logic and storage on the FPGA. Simulation time is the major limitation of using software simulators to generate data, given the copious amounts of disk space available. FAST must use an external interface if it is used to generate 100's or 1000's of MB of data because the PCB, without a DRAM or Compact Flash interface, cannot store all of the data. Finally, software simulators have the ability to generate data that normal hardware does not contain because of the global knowledge that exists in a software simulator. FAST has clear FPGA functional boundaries that make some global data gathering difficult, but post data processing and aggregation are possible. For example, each processor tile gathers data with respect to its components. The RWC and SMC FPGAs could aggregate the processor tiles' data if the infrastructure was there to transmit the data to the SMC or RWC FPGAs. Similarly, all the processor tiles' data could be collected and then aggregated and processed.

In the end, FAST can only provide visibility for structures built in the FPGA or data that flows through the FPGA. As a result, the FAST system does provide transparency similar to software simulators, but this visibility requires much more collaborative hardware/software support compared to hardware prototype systems. FAST is a true hybrid of hardware and software, providing the visibility of software simulators by integrating both software and hardware FAST components.

5.1.2 FAST flexibility

Modern hardware systems can provide copious amounts of data, but only with respect to predefined hardware and API's. FAST breaks this mold by allowing the user to add arbitrary monitoring at various levels in the system. There are some restrictions with regard to synthesized logic size and complexity, but partitioning and post processing data may address these issues.

Performance counters or other monitoring engines easily integrate into the FPGA fabric. Furthermore, RS-232 or other bi-directional communication protocols can be used to retrieve the data from FAST. System monitoring flexibility provides the visibility found in software simulators. This is key because FAST is a system that can be used to develop systems targeting a wide range of metrics, from raw performance to power savings and computational efficiency. These systems have different monitoring requirements that are only limited by the user and not the framework. Like software simulators, the user can add monitoring infrastructure that suits their needs.

Monitoring software and hardware is added to FAST at the point of observation. This observation point can be inside the processor tile for processor performance purposes, inside the RWC FPGA to monitor coherence or interprocessor communication, or inside the SMC FPGA to monitor memory utilization, I/O traffic or memory tracking. Thus, performance metrics are added at the point of use, much like software simulators. Furthermore, after the application runs on FAST, the data across many FPGAs can be aggregated and post processed to synthesize or derive other data.

There are hard limitations of fixed FPGA logic resources and on-chip memory that reduces the amount of data that can be collected. If the system is symmetric and homogeneous, monitoring can be divided among the FPGAs with each FPGA in the processor tile collecting mutually exclusive data that can later be aggregated. Likewise, multiple experimental runs can be performed using a different suite of monitors for each run to collect the necessary data. Running multiple experiments also may solve the limited on-chip memory constraints that all FPGAs have. Another alternative is to develop the infrastructure to stream high bandwidth data off FAST to accommodate any performance monitoring that is required. FAST is not as flexible as a software simulator with regard to adding numerous monitors or performance counters, but it does bridge the gap between inflexible hardware and flexible software simulators.

5.1.3 FAST performance architectures

Combining the visibility and flexibility of FAST monitoring and performance creates a performance architecture. This performance architecture defines the association between FAST's flexible components and the type or function of the performance monitors. This

architecture provides some structure to building data collection systems on top of the FAST substrate.

Figure 5.2 illustrates a high-level view of the FAST performance architecture. In this case, performance is a user defined metric that spans canonical processor performances like cycles per instruction (CPI) to performance per watt or watt per instruction, to name a few. Power estimates can be derived from logic transition monitoring, similar to methods used for Wattch [14]. Regardless of the metric used, the performance architecture directs FAST users to the correct component or components used to implement the monitor. Figure 5.2 is an incomplete list of all of the possible statistics or monitors that can be created for the FAST system. Furthermore, some monitors can be implemented in any one of multiple FPGAs, giving the system designer some freedom. Similarly, some of the metrics like power, shown in all FPGAs in Figure 5.2, might need to be aggregated to derive total system power.

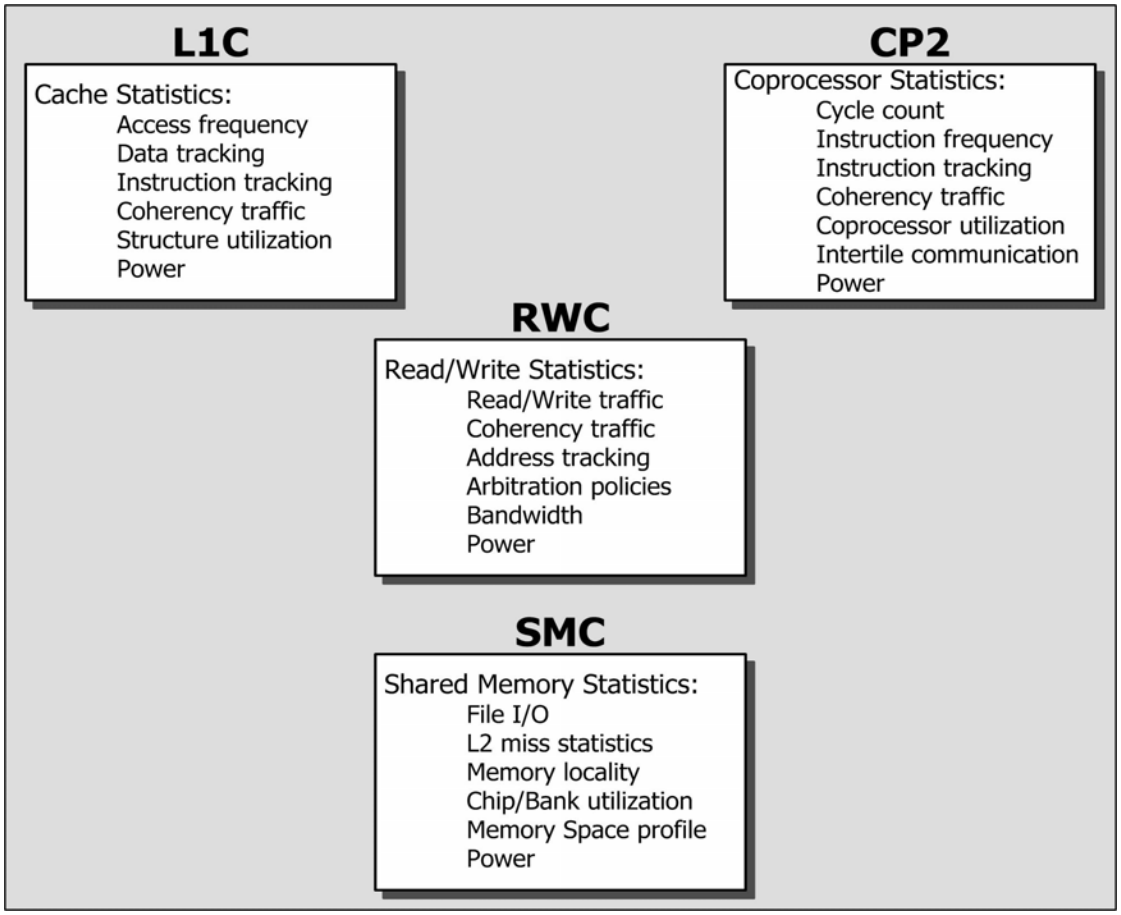


Figure 5.2. FAST non-exhaustive FPGA performance monitor architecture.

Conversely, other performance counters are very FPGA specific and can only be collected in the FPGA that has access to the signals of interest. Cache statistics are an example of performance counters that can only be collected in the L1C FPGA because the appropriate signals from the MIPS R3000 are only routed to that FPGA. Finally, not all FPGAs have to collect the same data. There are four sets of processor tile FPGAs and each FPGA could collect different data that could stand on its own or be aggregated with other data from the other processor tile FPGAs.

Section 5.2 provides more details about the performance counters implemented for the FAST CMP 4W-NC prototype and some discussion about other types of statistics counters or monitors.

5.2 FAST performance counters

The FAST CMP 4W-NC has several performance counters associated with the memory system. All of the performance counters are synchronous with respect to the system clock. FPGAs have limited clock resources that can be easily exhausted if multiple asynchronous performance counters are instantiated within the FPGA. Figure 5.3 provides an example of a structural Verilog module that counts the number of signal (Sig) transitions.

```
module perfCntrClk(Clk, Sig, Reset, Counter);

input Clk;                wire Clk;
input Sig;                wire Sig;
input Reset;              wire Reset;
output [31:0] Counter;    reg [31:0] Counter;

reg loHi;

   always @ (posedge Clk) begin
          if (Reset) begin
                 Counter <= 0;
                 loHi <= 0;
                 end
          else begin
                 // Only want to count once per transition
                 if(Sig & !loHi) begin
                        Counter <= Counter + 1;
                        loHi <= 1;
                 end
                 // Signal low know, go back to check for the edge
                 if(!Sig) loHi <= 0;
          end
   end
endmodule
```

Figure 5.3. Synchronous transition counting Verilog performance module.

In Figure 5.3, there is a guard signal, `loHi`, that prevents the synchronous signal transition from being counted more than once per transition. The 32-bit counter size could also be a user-configured parameter, making the counters any size.

Referring back to the FAST CMP 4W-NC shown in Figure 5.1, performance counters can monitor a variety of components in the CMP. Figure 5.4 illustrates where some of the performance monitors (PMs) would be placed in the FAST CMP 4W-NC and the corresponding FPGAs to collect statistics. The left side of Figure 5.4 illustrates the PM placement in a single FAST CMP 4W-NC processor, while the right side of Figure 5.4 illustrates where these PMs reside in the FPGAs on FAST. PMs for the processor and cache reside in the processor tile FPGAs, L1C and CP2. The L1C FPGA interfaces to the L1 caches and thus can provide statistics on cache behavior (PM2). The CP2 FPGA not only monitors and collects statistics for the processor (PM1), but the CP2 FPGA also collects data for part of the memory controller (PM3). The memory controller statistics (PM3) are also collected in the RWC FPGA where the L2 memory resides. Furthermore, more in depth L2 memory statistics can also be monitored in the RWC controller (PM4).

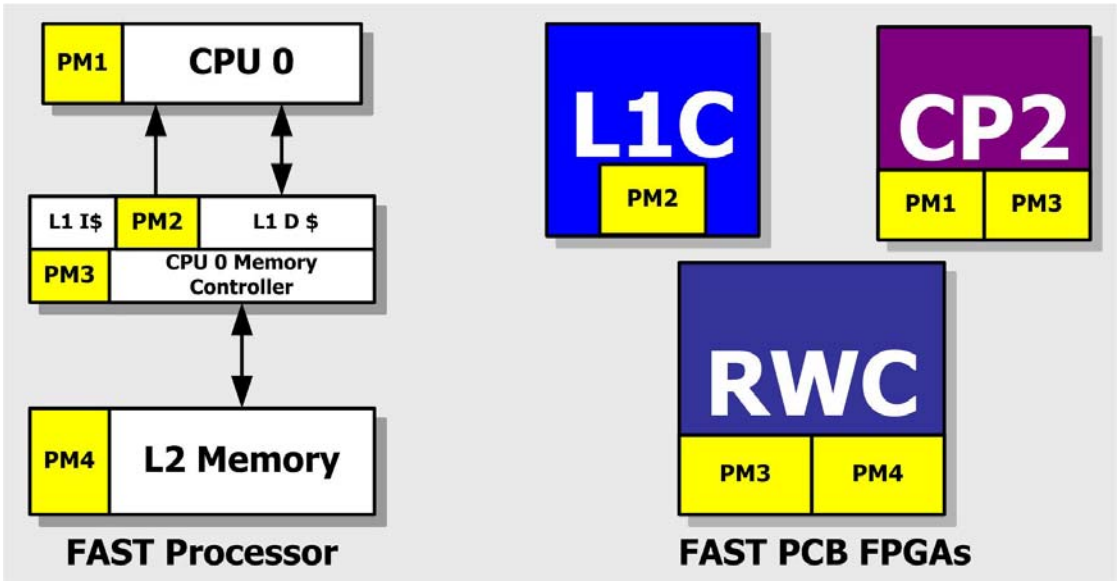


Figure 5.4. Performance monitors (PMs) in the FAST CMP 4W-NC and corresponding FAST FPGAs.

The FAST CMP 4W-NC has a variety of performance counters implemented for program behavior statistics. Performance counters similar to those shown in Figure 5.3 are implemented in the RWC FPGA to collect L2 read and write frequencies. Moving

into the CP2 FPGA, total program execution cycles, number of stall cycles, and the number of dynamic instructions executed are collected in PM1 counters. Finally, the L1C FPGA provides a window into cache performance. The numbers of reads and writes, as well as cache misses, are counted for the data and instruction caches. Thus, for every application that is run on the FAST CMP 4W-NC, the PM data verifies correct application behavior *and* correct FAST CMP 4W-NC model behavior. Simple back-of-the-envelope calculations or application statistics from other MIPS-based SGI machines can also validate application behavior. Interestingly, all of the performance monitors presented in Figure 5.4 can also be implemented in the CP2 FPGA by forwarding or capturing the relevant signals in the CP2 FPGA.

Performance counters provide data for application and/or architecture performance. Changing aspects of the architecture can affect system behavior and not necessarily the correct program execution, once the new prototype has been debugged. For example, changing the L2 memory latency increases program execution time, but does not hamper the correct application execution. PMs also have intuitive locations in the FAST system, but these PM locations are flexible and user-defined. Section 5.3 presents data for all the benchmarks run on FAST CMP 4W-NC with six different L2 memory latencies, as well as the normal performance statistics discussed earlier in this paragraph.

5.3 FAST results

There are two categories of data that have been collected with the FAST system. The first category of data characterizes the application and architecture and the second category of data reflects system sensitivity to a particular changing parameter. These two categories illustrate the versatility of the FAST system and how FAST can produce a variety of data very rapidly. For each set of data, it is also useful to describe how the system setup changed in order to derive the data and how the data was collected.

5.3.1 FAST data collection

Before diving into the results, it is useful to describe the data collection process. There were two methods for observing data on the FAST system. The first method used unassigned I/O pins on the FPGAs as an output for the counter values. Using tools like

J-SCAN, the JTAG static observation tool, the performance counters could be read once the application had completed and the processor was stalled.

Figure 5.5 is an example of using J-SCAN to read processor local performance counters from the CP2 FPGA. J-SCAN is set up to read the FPGA boundary scan chains for processor tiles 0 and 1 or processor tiles 2 and 3. Thus, Figure 5.5 shows J-SCAN reading the results for processor tiles 0 and 1, with the circles on the right of the figure over the two performance counters for each processor tile. The total cycle count appears just above total number of stall cycles, both displayed in 32-bit hexadecimal numbers.

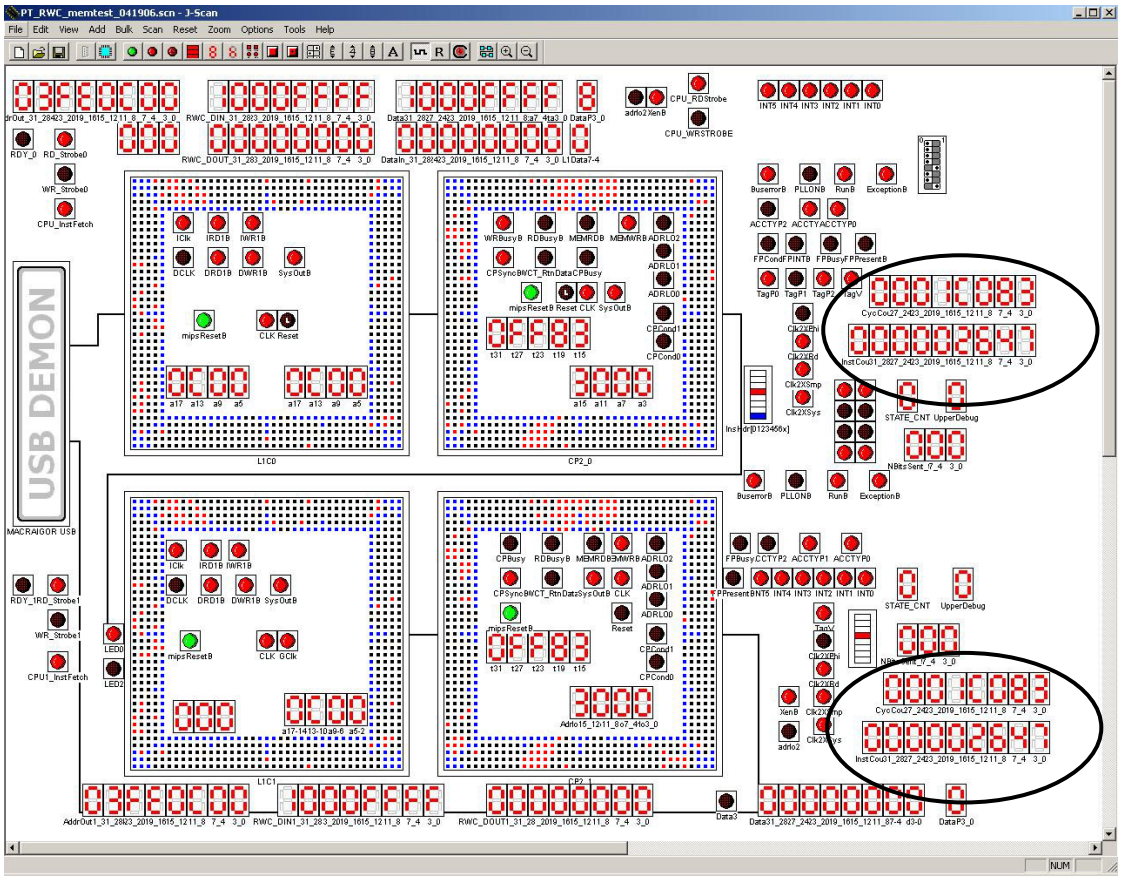


Figure 5.5. Two sets of CP2 performance counters for total number of cycles executed and total number of stall cycles running Quicksort.

Figure 5.5 also illustrates the plethora of data that J-SCAN provides. Inside the CP2 FPGAs, the two large devices on the right, J-SCAN displays the contents of the tag and the address bus. In this case, the processor has jumped to 0X0ff83000, the stall address location for successfully executed programs. The data bus is also shown in Figure 5.5.

The data bus for processor tile 0 is located at the top of the figure with hexadecimal value 0x1000 FFFF. Likewise, processor tile 1's data bus display is located at the bottom of Figure 5.5 with the same 32-bit hexadecimal value. Finally, the L2 memory is word addressed and the address bus is shown in Figure 5.5 for the two-processor tiles. The upper left hand corner of Figure 5.5 shows the processor tile 0 L2 address bus with hexadecimal value 0x03FE0C00. Likewise, the bottom left hand corner provides the L2 address bus with the same data for processor tile 1.

There is another alternative for streaming data out of the FAST PCB. FAST uses a daughter card with an RS-232 buffer to stream data off the FAST PCB using the additional FPGA header pins that most of the FPGAs have; only the L1C FPGAs do not have physical header pins on the PCB. However, the L1C FPGA does have unassigned additional pins between each pair of L1C and CP2 FPGAs that can be used to relay information through the CP2 FPGA. Power to the RS-232 buffer is supplied by one of the FAST power headers. Initial RS-232 development used a 9600-baud data rate. However, this is an arbitrarily set data rate that can easily be changed. Furthermore, the performance counters can be daisy-chained together using the interprocessor tile buses if only one point of data streaming is desired. A host machine using a perl script initiates the RS-232 communication with the FAST PCB and prints the performance counter information to a terminal window. The perl script opens the host machine's COM port with the appropriate settings and data rate. When the host machine's "enter" key is hit, the RS-232 module in the target FPGA sends back the performance counters to the host machine. The data is displayed on the host terminal.

5.3.2 Building FAST applications

The FAST system is still in its infancy with respect to software development. There are many software components that require development and/or additional refinement. Fortunately, FAST can leverage native MIPS-based machines for software development. The native MIPS environment allows the software to be developed, compiled, and debugged, rapidly [64]. Similar software tools could be used to develop and debug software on FAST if the software infrastructure was available. Likewise, cross-platform development tools also enable software development for FAST [35, 74]. By leveraging a

host environment to produce correctly running programs, FAST is able to rapidly run full applications. New prototyping platforms suffer from the same software development problems: while development and debugging are possible on the prototyping system, initial software development on the prototyping platform is cumbersome or near impossible because the OS, software libraries, and compilers do not exist or are very immature.

FAST can run applications compiled for the prototyping system at hardware speed. This yields rapid system and application results. By leveraging native or cross-development environments, application development can be completed faster because the software development and debugging is done at current hardware speeds, 2 orders of magnitude faster than the FAST system. As outlined in Section 4.4 *FAST applications*, once the initial FPGA programming bit file has been created, it only takes a few minutes to update the L2 memory with a new program and download the FPGA bit files to the FAST PCB. This assumes that a host environment is used to do the initial software development, compiling, and debugging. Further software development can be done on FAST using the current software infrastructure, but at the granularity of a single instruction. Likewise, prototype system development can be monitored at the application level or single instruction level. As the FAST software system matures, the software development and debugging process will become much easier than it currently is, enabling the same rich environment available using current tools like gcc and gdb [13, 65].

It should be noted that without native systems or a cross-development environment, FAST would not be able to run applications rapidly because of the lack of required tools. Mapping other architectures to FAST requires the same ability to leverage a development environment, either native or emulated, for rapid software development and application run-time [35, 64, 74]. Moving forward, FAST requires developing or porting an OS, compiler, and debugger to enable a full-system prototyping substrate. FAST's goal is to provide rapid feedback for application and system tuning. In the process of creating that infrastructure, FAST would also be able to natively compile and debug applications using the full-system software infrastructure.

5.3.3 FAST CMP 4W-NC running Stanford Small Benchmark Suite

FAST ran 6 out of 8 integers benchmarks from the Stanford Small Benchmark Suite to demonstrate its utility [49]. The remaining two out of the eight integer benchmarks would have required additional FAST software infrastructure development before they could be executed on the FAST system. Using FAST, data was collected that provided detailed application behavior, along with data collected for an L2 memory latency sensitivity study.

As discussed in Section 5.2, the FAST CMP 4W-NC has performance counters in each FPGA to monitor program behavior. The L1C FPGA monitors data and instruction cache statistics including: read and write frequencies and cache miss statistics. The CP2 FPGA monitors the processor execution time including: total number of program execution cycles, total number of memory stall cycles, and total number of instructions executed. These three separate counters have the well-defined relationship that the total number of executed cycles equals the number of stall cycles plus the number of instructions executed. This provides another quick and easy check of the performance counters. Finally, the RWC FPGA contains a small L2 memory for each processor tile and monitors L2 memory statistics including: the number of reads and writes. The RWC FPGA can also be used to implement the other performance counters shown in Figure 5.2.

Figure 5.6 shows the L1 instruction cache read and write statistics for all six benchmarks using a SRAM L1 cache with the program information loaded into the 32 KB L2 memory. As this graph shows, the instruction reads far out number the instruction writes. As would be expected with non-self modifying code, all the writes to the L1 instruction cache are due to cold demand misses. The instruction working set fits inside the L1 cache, therefore, no conflict or capacity cache misses exist for these small benchmarks. The number of instruction cache writes determines the static instruction set for the application. The static instruction set is the number of unique instructions executed at least once during the program's execution. As one would expect, the program does not execute all of the compiled instructions, as shown in Table 5.1.

Similarly, the number of instruction cache reads indicates the dynamic number of instructions executed during the lifetime of the program.

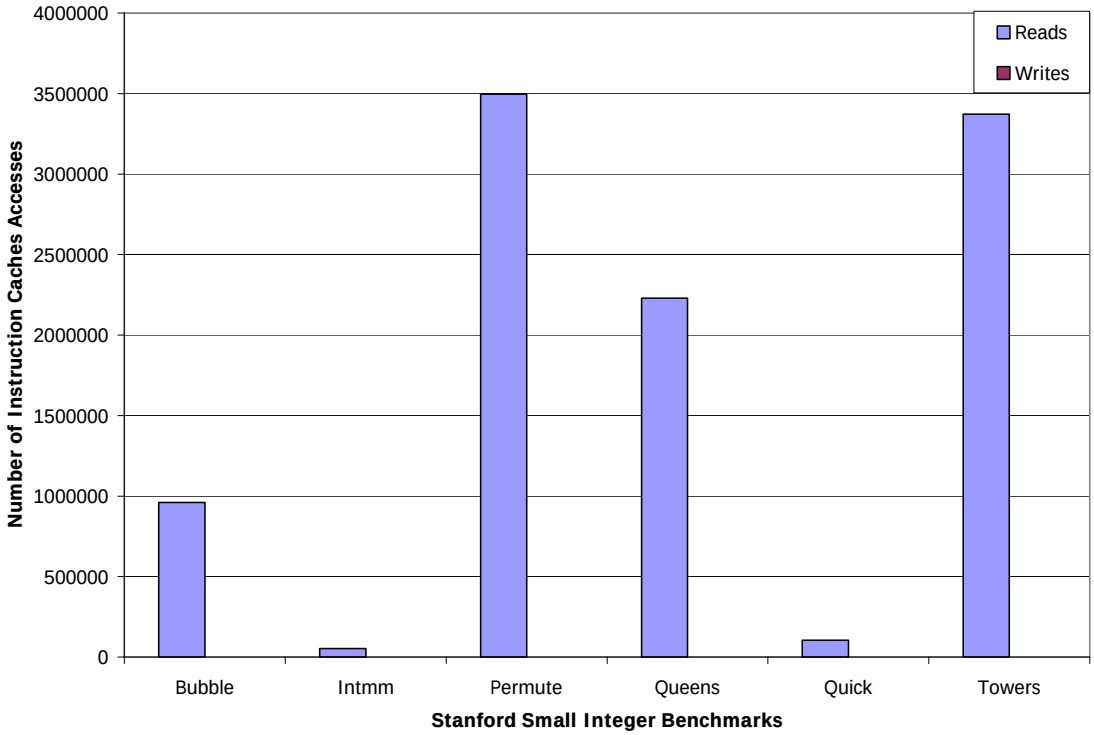


Figure 5.6. Instruction cache read and write frequencies for the Stanford Benchmark Suite.

Figure 5.7 provides the same data for the L1 data cache. The applications have been compiled with no optimizations. The lack of code optimization results in a large number of data cache writes. Furthermore, all data for these applications is generated at runtime. Thus, every load or data cache read in the program is preceded by a store or data cache write. The data cache reads and writes are very similar in number as a result of running unoptimized code. Furthermore, the number of data cache reads and writes is far less than the number of instruction cache reads, shown in Figure 5.6. Comparing Figures 5.6 and 5.7 derives the ratio of memory operations to instructions executed for each benchmark.

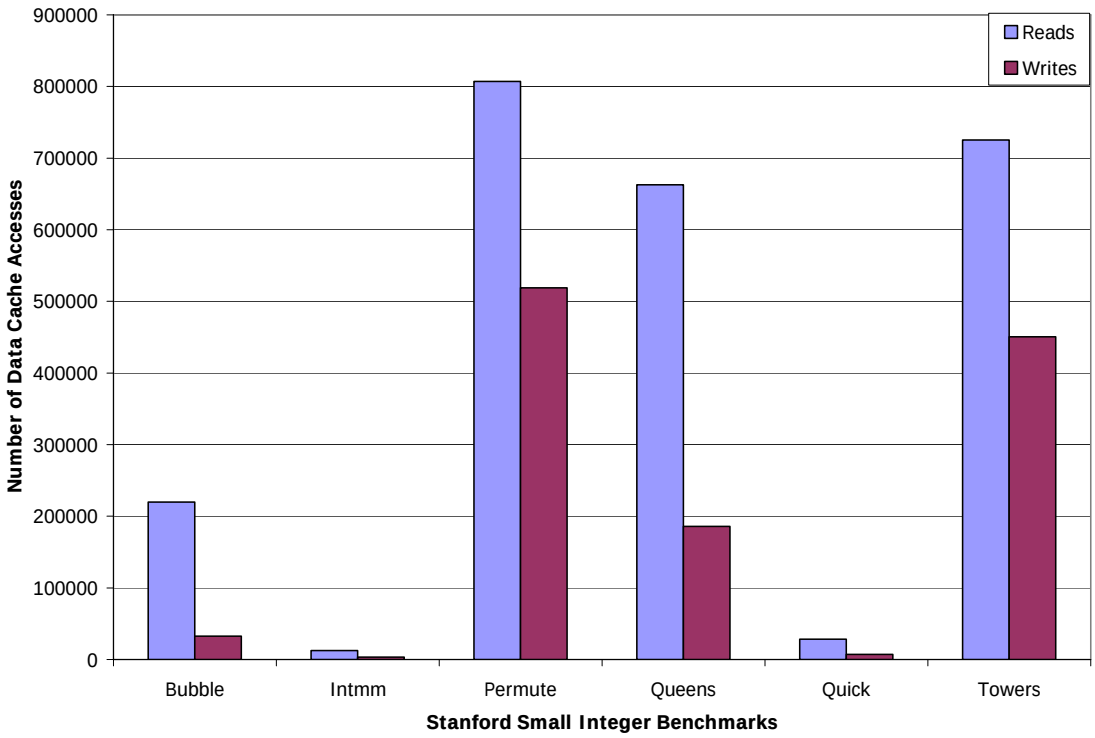


Figure 5.7. Data cache read and write frequencies for the Stanford Benchmark Suite.

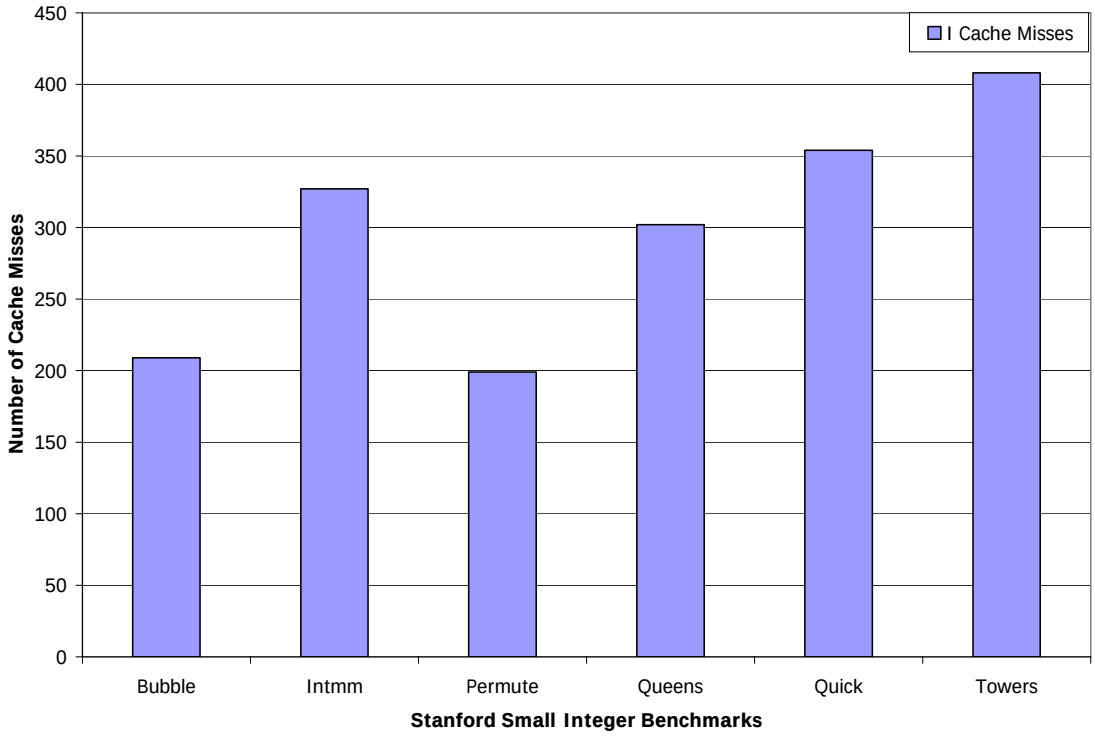


Figure 5.8. Instruction cache misses for the Stanford Benchmark Suite.

Rounding out the cache statistics, Figure 5.8 tallies the number of instruction cache misses for the Stanford Small Benchmark Suite. Because there are no conflict or capacity misses in the instruction count, the number of instruction misses is equivalent to the number of instruction cache writes. This makes it very easy to validate the performance numbers generated in the L1C FPGA. Finally, there are no data cache misses because all of the data is generated at runtime, so there is no data to demand miss from the L2 memory, if the L1 data cache is present. Likewise, there are no conflict or capacity misses in the data cache because of the small data working set size.

The performance counters instantiated in the CP2 FPGA correspond to PM1 in Figure 5.4, the processor performance monitors. There are eight performance counters implemented in the CP2 FPGA, and these are listed in Table 5.1, along with a summary reduction of information from all the performance counters. As seen in Table 5.1, the number of dynamic instructions dwarfs the number of stall instructions due to instruction misses in the instruction count. Furthermore, as previously mentioned, these three independent performance counters have the relationship that the total cycles equal the stall cycles plus the dynamic instruction count. Thus, the counters are validated against one another. Similarly, the CP2 performance counters can be verified by comparing to the results from the L1C FPGA performance counters. In this case, the number of instruction cache reads is equal to the number of dynamic instructions and the number of stall cycles equals the number of instruction cache misses times the L2 memory latency.

Table 5.1. Total number of program execution cycles, total number of instructions executed, and total number of stall cycles counted in the CP2 FPGA.

	L1 Data Cache		L1 Instr.Cache		L1 Cache		Total	Total	Total Dyn.	Compiled	L2 Memory	
Applications	Reads	Writes	Reads	Writes	D Miss	I Miss	Cycles	Stalls	Instr. Cnt	Instr.	L2 Reads	L2 Writes
Bubble	219,630	32,888	960,577	209	0	209	969,647	9,070	960,577	301	218	32,887
Intmm	12,433	3,252	53,271	327	0	327	68,133	14,862	53,271	355	336	3,251
Permute	807,149	518,930	3,495,775	199	0	199	3,496,814	1,039	3,495,775	254	208	518,929
Queens	662,753	185,809	2,230,110	302	0	302	2,231,664	1,554	2,230,110	358	311	185,808
Quick	28,375	7,182	105,020	354	0	354	114,819	9,799	105,020	376	363	7,181
Towers	725,319	450,769	3,372,684	408	3	408	3,374,783	2,099	3,372,684	468	405	450,765

Table 5.1 provides a detailed view of all the performance counter results for all the benchmarks. Because the number of stall instructions are so small, presenting the data in Table 5.1 as a graph provides no insight because of the extreme difference in magnitude between the overall program execution time and the number of stall cycles.

Figure 5.9 shows the reduction in stall cycles over time during the development of the memory system. In this case, the performance counters are used to verify the correctness of the memory subsystem implementation. Moving from left to right in Figure 5.9, the stall cycles decrease because the L1 SRAM data cache is being added to the design. The far left bar illustrates performance with no L1 data cache and the far right bar illustrates a fully functional L1 SRAM data cache. The bar in the middle shows an instantiated L1 SRAM data cache structure that is not fully functional because of timing constraint violations. When the L1 data cache fails, the CPU treats the failure as an L1 data cache miss. The L2 then supplies correct data, all at the cost of extra stall cycles. As Figure 5.9 demonstrates, performance counters can aid in the hardware development cycle by indicating correct hardware operation. In all cases in Figure 5.9, the program calculated the correct result, thus, correct program execution does not necessarily imply correct hardware operation.

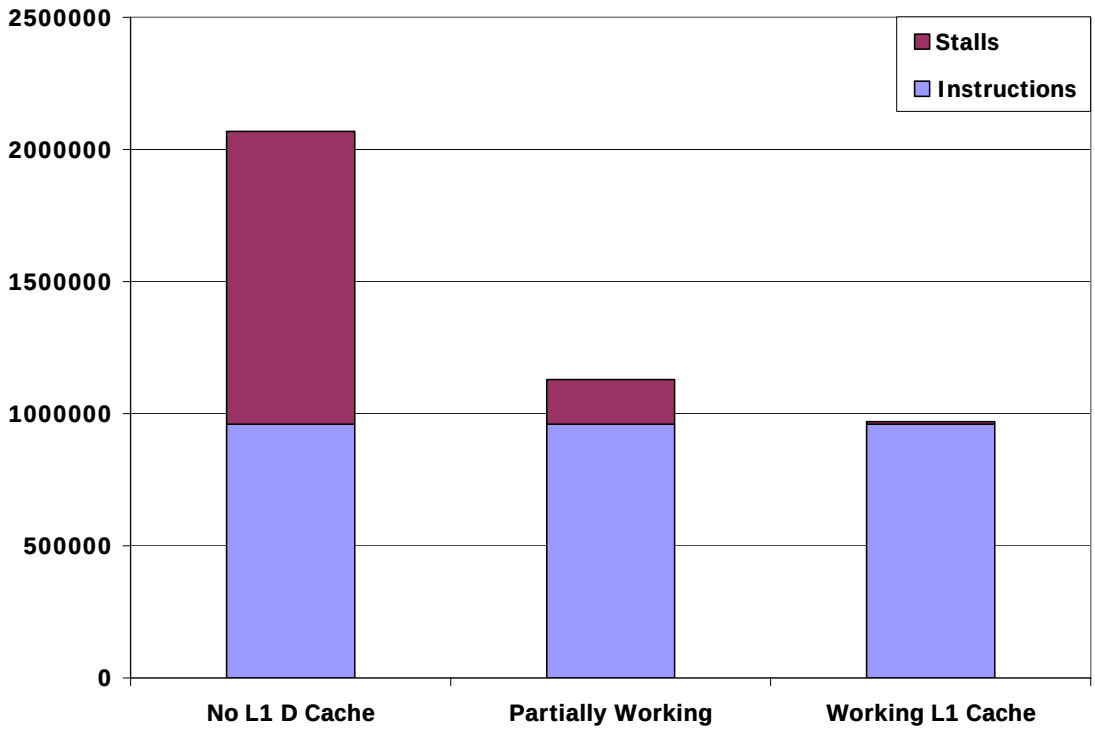


Figure 5.9. The performance of Bubblesort using no L1 cache vs. a fully functional working L1 cache. The middle bar is an instantiated, partially working L1 cache with unmet timing constraints.

Finally, the RWC FPGA contains the four private L2 memories, one for each processor tile. For these studies, the programs are preloaded into the RWC FPGA's L2 memories and when a cache miss occurs, the CP2 FPGA relays the cache miss request to the RWC FPGA, which fulfills the request. The L2 memory performance counters can also be implemented in the CP2 FPGA because the CP2 FPGA relays the memory miss to the RWC FPGA. However, L2 memory specific data is easier to track in the RWC FPGA and implement in the RWC FPGA, e.g., L2 memory tracking data.

Because the RWC FPGA holds the program data, only the RWC FPGA programming bit file needs to change in order to run the 6 applications presented. The L1C and CP2 FPGAs remain unchanged across all of the experiment runs required to collect the data for Figures 5.6 to 5.9 and Table 5.1. Using Data2Mem [107], new RWC FPGA programming bit files are created by replacing the RWC BRAM contents to reflect the new application stored in the BRAMs. Thus, there are a total of 8 FPGA programming bit files: the L1C, the CP2, and six RWC bit files, one per Stanford Small Benchmark Suite application.

The data presented thus far demonstrates how the FAST system can collect application and architecture behavior data. It should be stressed that the FAST system is not limited to just the performance statistics shown in Figures 5.6 to 5.9 and Table 5.1. Nor are the performance statistics restricted to the short list of possible statistics or instrumentation provided in Figure 5.2. Like software simulators, FAST has the ability to collect both system and application statistics at run time.

Sensitivity studies are possible using the FAST system. FAST targets TLP architectures with an emphasis on novel memory system designs. Techniques to hide memory latency are fundamental to TLP architectures and any processor that operates at several orders of magnitude faster than the main memory subsystem. In order for the FAST system to investigate TLP architectures, FAST must be able to implement the novel memory system *and* alter the memory latency for various memory levels in the system. Figure 5.10 illustrates an L2 memory latency sensitivity study for the FAST CMP 4W-NC system implementation.

The L2 memory sensitivity study increases the L2 memory latency from 5 cycles up to 257 cycles. As expected, the program execution time and number of stall cycles increases linearly with the L2 latency, as shown in Figure 5.11. By extrapolating the L2 latency, the cold instruction cache misses are correlated to the static instruction set of the application. The y-axis intercept for a one-cycle L2 latency corresponds to the number of static instructions executed by the application, or cold instruction cache misses. The program fits in the instruction cache; thus if the L2 latency were one cycle, the number of stall cycles would be equal to the number of static instructions. This observation was an easy validation mechanism for this sensitivity study. However, in depth investigation was necessary to explain the `div-mflo` additional latency.

The other interesting observation from the L2 latency sensitivity study is that the L2 memory latency spans the range from a very aggressive L2 memory latency all the way to an off-chip memory latency. As Figures 5.10 and 5.11 demonstrates, the FAST system is able to implement CMP or MP systems spanning the range of on-chip and off-chip memory latencies. The FAST system does have a lower limit of 4 cycles required to fulfill a cache miss. The MIPS R3000 enforces this lower cache miss latency because the R3000 requires 4 cycles to request and fulfill a cache miss [54].

Figure 5.10 requires further explanation for the lack of consistency in the data. Permute, Queens, and Towers generate data that matches the increase in L2 latency. The lines are parallel with the same slope. Bubble, Intmm, and Quick use a random number generator that injects more latency because of the instruction sequence: `div-bnez-nop-mflo` [54]. During the initialization of the 250 element arrays, the divide latency adds significant stall cycle overheard that must be subtracted in order to observe the same behavior as Permute, Queens, and Towers. If the extrapolation takes into account ISA features that produce data, one's intuition is preserved. The complete dataset for this chapter and the L2 stall cycle validation discussion can be found in Appendix D.

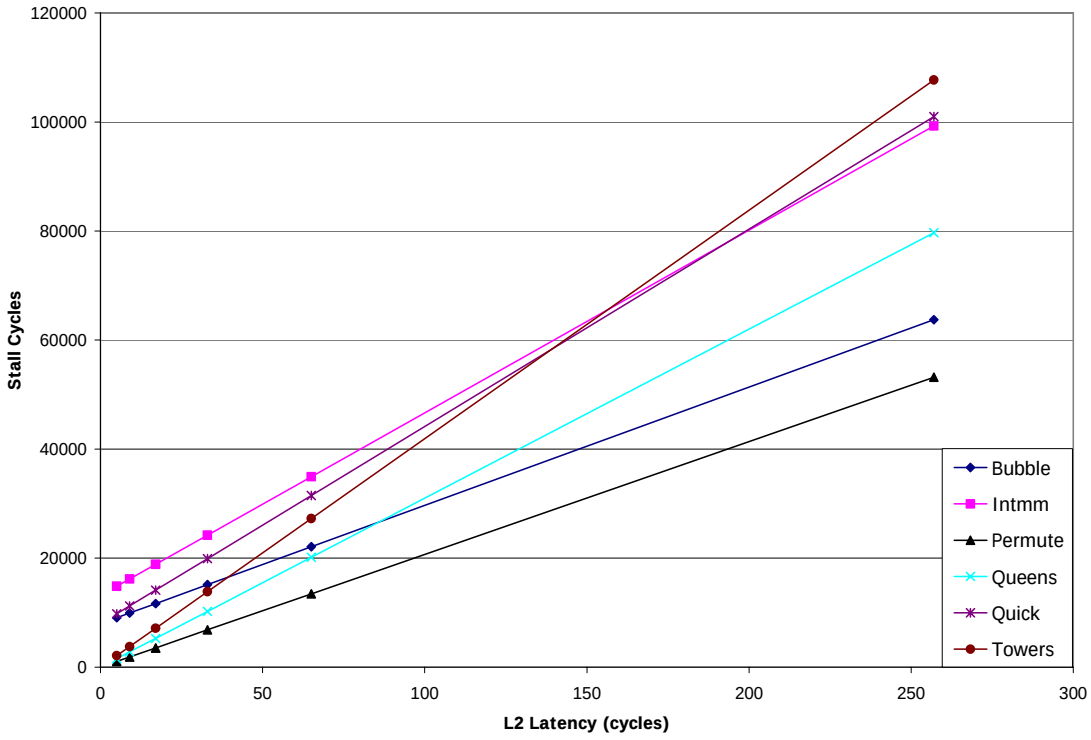


Figure 5.10. Varying the L2 memory latency for the Stanford Benchmark Suite.

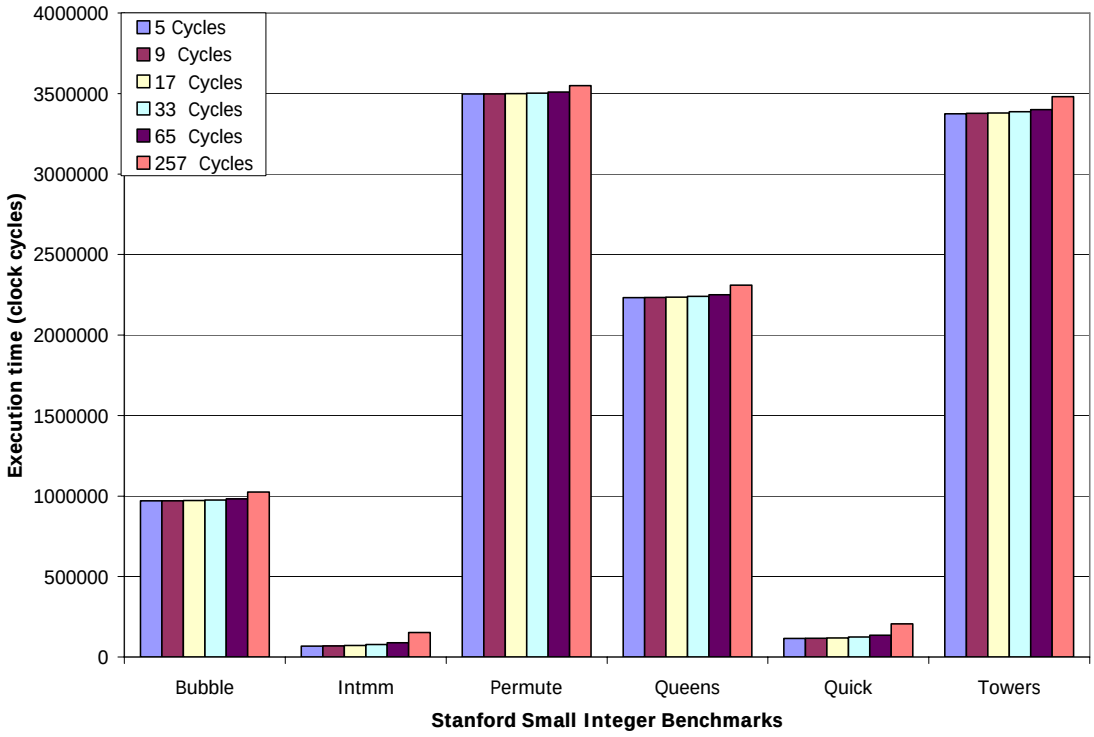


Figure 5.11. Increased execution time for the Stanford Small Benchmark suite when increasing the L2 memory latency.

The L2 memory sensitivity required far more FPGA programming bit files for the experimental runs. The L2 memory latency was defined and changed in the CP2 FPGA. There were six L2 latencies requiring six CP2 FPGA bit files. Likewise, there were six RWC bit files, one for each Stanford Small Benchmark Suite application. Finally, the L1C FPGA was unchanged across all the configurations. The resulting Cartesian product of experimental runs is manageable, only 36 data points, but for large sensitivity studies [30], the number of experimental runs could spiral out of control. This is important because unlike software simulators, the FAST infrastructure has not been developed for automating the experimental process. However, it should be noted that the Flash memory controlled by the PLD can implement and manage such sensitivity or other studies that require experimental iteration using several FPGA bit files. The PLD could be used to monitor each iteration of the large study. When the iteration completed and the data was collected, the PLD could reprogram the FPGAs with the FPGA bit files stored in the Flash for the next iteration. Only a small number of FPGA bit files need to be stored in the Flash memory, which enable multiple iterations for sensitivity or other studies.

Fundamentally, the hardware infrastructure exists for a wide variety of system development and a variety of data collection methods. Unfortunately, the modular and reusable software systems require more development, thereby creating a system software library. This FAST software library would be similar to software simulator modules and APIs. As a result of the software library, development of prototype TLP architectures would be dramatically accelerated.

5.4 FAST performance conclusions

The FAST system is able to sustain 10's of MIPS with peak performance of about 133 MIPS, four processor tiles running at 33 MHz. FAST's high performance enables rapid software development for prototype systems. The FAST CMP 4W-NC was used as a simple motivating example that illustrates the power and flexibility of the FAST system. The FAST CMP 4W-NC demonstrated the ability to collect application and hardware performance data. The memory system is also highly configurable and the L2 memory

latency study further demonstrated FAST's ability to implement this key aspect of any system used to study and prototype TLP architectures.

FAST's performance results demonstrate its utility and potential. The FAST prototyping system is fully functional, with ***no PCB rework***. The simple FAST CMP 4W-NC motivating example demonstrates both the utility and the potential. The FAST system met its initial design goals and is able to prototype new TLP systems. Furthermore, FAST is able to implement a variety of performance or monitoring counters at various levels in the system providing transparency similar to software simulators. The transparency yields rapid application and system behavior analysis. Likewise, sensitivity studies are also part of the FAST prototyping toolbox as demonstrated by the L2 memory latency results.

The performance results of the FAST CMP 4W-NC allude to many other aspects of the FAST system. Overall, FAST provides the capabilities that bridge the gap between software simulators and one-off hardware prototypes with respect to observing application and system performance. Furthermore, this initial work just scratches the surface of FAST's potential. The multiple memory levels implemented in the FAST CMP 4W-NC demonstrate that coherent memory designs can be implemented with FAST. Most importantly, FAST provided a nascent rapid software development environment that inherently provides high fidelity. FAST demonstrates the flexibility of the PCB design, and with further software development, FAST provides even more functionality. More detailed system and application analysis and improving FAST's ease of use all require more software development to produce a software library for computer architects. Unfortunately, most of the software library development is beyond the scope of this work.

Producing the initial FAST results was accelerated and made possible by leveraging MIPS-based systems and a cross-development environment [35, 64, 74]. This significantly reduced the software development effort because there was an environment to run, debug, and estimate the application performance. This produced performance result estimates that validated the FAST CMP 4W-NC performance results. The performance counters further validated system performance. All of the applications ran

self-checks to verify correctness, but the performance counters are required to verify system correctness, as demonstrated in Figure 5.9.

As shown in Figure 5.9, specifying timing constraints and having the design meet timing is one of the most critical Verilog development procedures. Overall, providing better tools to specify, track, and modify the design timing will reduce the hardware design time. Once the FAST PCB was tested, most of the VAL and TLP Verilog effort was spent meeting the design time constraints. This required physical placement of buffers, latches and iterative placement and routing of the design. Once timing was met, the placement and routing for the design was fixed. As a result of the fixed placement and routing, future design modification will need to be carefully added to maintain design timing. Creating the FAST system has demonstrated that a FAST software library is not the sole solution for the Verilog portions of the designs. Tools that make timing easier to understand, specify, and achieve are required to make the prototype system work.

Overall, building a hybrid prototyping platform is difficult, with significant software development still required to truly make this system easy to use. As this work has shown, by leveraging existing systems, some or all of the software development can be circumvented, accelerating the prototyping process. Likewise, leveraging a larger community, like the open source community, will also accelerate system prototyping by building a large software library used as the foundation for future work. The results presented here provide the proof that flexible hardware prototyping systems are useful, provide system and application insight, and enable rapid full-system prototyping. Moving forward, building new systems with FAST will be easier given the initial building blocks presented in this work. Ultimately, with a mature infrastructure, FAST or systems like it would be able to implement prototype systems with the same effort required to create new architectures with mature software simulators like SimpleScalar, Simics, or SimOs [4, 12, 28, 34, 50, 68, 81].

Chapter 6

FAST 2.0: The next generation hybrid platform

This chapter takes the concepts and lessons from the FAST 1.0 system and proposes the next generation flexible hardware prototyping platform, FAST 2.0. Like FAST 1.0, FAST 2.0 is a combination of hardware and software. The combination of the FAST 1.0 research and related work provide the insight for producing a much improved prototyping platform. Previous work has laid the foundation for FAST 1.0, while successors to FAST 1.0 define the technology trajectory, thereby improving the FAST 2.0 proposal [9, 11, 20, 21, 29, 31, 33].

The next generation PCB will leverage advances in hardware, both in terms of device speed and device density. However, FAST 2.0, or systems like it, will only succeed if the research community embraces it. And the community will embrace it only if it leverages a well-developed software base. This software foundation can be developed two ways. First, the software development becomes part of the successful open source movement, similar to Linux [94, 95]. Creating an open source software library accelerates research and development of the FAST 2.0 system by increasing the available resources to the project. An alternative approach is to leverage industry to develop the hardware and software required for the hardware prototype combined with a rich collection of APIs, much like Simics [68]. Regardless of the method used to develop the FAST 2.0 system software, this system software base was absent from FAST 1.0 and similar early projects [9, 20, 31], prolonging development and limiting usage. Learning from the FAST 1.0 experience, given an intelligent design combined with software libraries, FAST 2.0 can implement a wider variety of architectures.

Creating a truly useful system also requires easy to use tools and configuration processes that abstract away the complexity of the hardware. Thus, FAST 2.0 requires the development of configuration tools and auto generated Verilog and support files. Combining all of these components results in the FAST 2.0 system.

6.1 Building the next generation hardware prototyping platform

The next generation hardware prototyping platform will provide improved performance, flexibility, and capability. Improvements in device speed and density guarantee the superiority of the next generation system, but will only yield two out of the three improvements. An intelligent hardware architecture and design are required to capture all three benefits simultaneously. FAST 2.0 is a proposal for the next generation hardware prototyping platform that leverages existing hardware and software to implement a broader class of new computer architectures, enabling full-system prototyping.

6.1.1 Leverage existing hardware

FPGAs and SRAM devices continue to run faster with greater device density. The operating frequency of FPGAs may not increase as quickly as that of general-purpose processors, but increases in device density appear to be unabated. FPGAs in 65-nm technology represent three generations of technology advances beyond the FPGAs used in FAST 1.0. This translates into much faster and much denser FPGAs from both Xilinx and Altera [8, 126]. The Virtex-5 from Xilinx and the Stratix II from Altera provide about 3 times the BRAM density and 2.5 times the logic density found in the largest Virtex-II FPGA. These FPGAs also have a number of other improvements: lower core voltage, 6-input LUTs, higher device density, improved clock management, more DSP blocks, faster DSP blocks, and high speed serial links, to name a few of the features [126]. These FPGAs also have more user-defined I/O pins, increasing the available I/O pins from 1100 up to 1200. Unfortunately, over the last two generations of FPGAs, the amount of on-chip BRAM has stagnated. The combined SRAM and BRAM in a FAST 1.0 processor tile still exceeds the amount of BRAM in these new FPGAs [8, 126].

FAST 2.0 must integrate the latest generation FPGA, SRAM, DRAM, various I/O mechanisms, and have expansion interfaces to provide a complete hardware foundation. Further details about leveraging existing hardware are provided in Appendix C. FAST 2.0 requires PCB integration skills beyond the technical scope of an in-house small research project. FAST 2.0 presents new technical design challenges related to high-speed PCB design. The FPGA interconnect, as well as the high-speed serial links, require special design rules to accommodate the high-speed signaling, to reduce signal skew and to

minimize noise. The FAST 1.0 PCB did not require such attention to detail because of its low frequency operation, at tens of megahertz. FAST 2.0 will operate in the hundreds of megahertz with some high-speed signals operating in the low gigahertz, requiring special high-speed PCB signal design rules.

6.1.2 Leverage existing software

Software is required in order to make the system functional. Without a developed software infrastructure, FAST 2.0 is basically useless. It is essential for FAST 2.0 or projects like it to provide functionality out-of-the-box. This is the key enabler for making FAST 2.0 adopted by the rest of the research community. FAST 1.0 leveraged existing software infrastructure in the form of tools, like compilers and Verilog synthesis tools. However, FAST 1.0 lacked a software code base that could be leveraged, inhibiting the system usage and/or prolonging prototype development. In essence, there was an uphill battle to produce the PCB infrastructure for the FAST 1.0 Verilog Abstraction Layer (VAL) while, at the same time, building and mapping the prototype architecture to the new system.

FAST 2.0 proposes to solve this problem by re-using some of FAST 1.0's Verilog, the code that interfaces to the VAL, and by building a software open source community similar to the community using Linux. For FAST 2.0, leveraging existing software is not limited to the VAL or other Verilog used to program the FPGAs. The software developed by the open source community also incorporates: the architecture prototyping layer, operating system (OS) [94], drivers and APIs, and applications and benchmarks. With a mature FAST 2.0 system in a perfect world, FAST 2.0 would leverage all layers of the software stack, accelerating FAST 2.0's adoption as a *de facto* prototyping platform. Further details about leveraging existing software are provided in Appendix C.

The key to any hardware prototyping platform is leveraging the open source development community, thereby providing a development, distribution, and maintenance methodology and platform. Unifying the FAST VAL through the application layer, at one source instead of across multiple sources, also reduces coordination effort. Currently, there are the GNU tools, Verilog open source cores, and third party Verilog tools that are all freely available, but scattered over the Internet [13, 38, 65, 94, 95].

Creating a central repository for the FAST 2.0 hardware and related software would leverage the strengths of the entire community, enabling research far beyond the abilities of any one university or organization.

6.1.3 Implement new architectures

FAST 2.0 is not just an FPGA upgrade to FAST 1.0, but an evolution in the rapid prototyping platform space. FAST 2.0 combines all of the advantages from FAST 1.0 and related systems, while at the same time removing as many prior disadvantages as possible. The result is a prototyping platform that can implement a broader class of architectures than previous prototyping platforms. FAST 2.0 combines an intelligent FPGA interconnect, design for expansion, and intelligent device placement.

Finally, FAST 1.0 was designed as a scalable compute fabric. FAST 1.0 used an expansion header for expanding the system, up to 16 PCBs. FAST 2.0 continues this trend by incorporating dedicated PCB and FPGA communication links. Enabling both FPGA-to-FPGA and board-to-board communication expands the scope of architectures that FAST 2.0 can prototype, both in terms of system complexity and system scalability. By enabling high bandwidth FPGA-to-FPGA communication, the functional boundaries between chips, on the same PCB or across multiple PCBs, can be virtually eliminated, creating "super" FPGAs capable of implementing very complex logic and subsystems. Likewise, board-to-board communication can be used to provide a logical boundary or unit in a communication hierarchy. These logical and functional boundaries provide intuitive cleavage points in the system. However, because of the high bandwidth board-to-board communication, PCB boundaries can be abstracted away. In this way, we can create very large-scale systems, on the order of one thousand compute nodes, perhaps using a rack-mounted system for large scale research. FAST 2.0 places components on the PCB with building large scale PCB systems in mind. As a result of the intelligent interconnect, daughter card expandability, and flexible high bandwidth communication, FAST 2.0 can implement small or large scale digital systems from a wide array of compute domains.

6.2 FAST 2.0 building blocks

Thus far, this chapter has discussed the hardware and software components required for building the next generation hardware prototyping platform. Before a large community can adopt FAST 2.0, or systems like it, tools or building blocks are required to make it easy to use. The building blocks are the tools that transform the code, Verilog, assembly language or some higher-level language, into FAST system components, FPGA programming files or software executables or software libraries. These building blocks range from the Verilog development environment, software development environment and software tools, to the overall system design framework.

The Verilog and FPGA development tools are specific to the FPGA components used for the system. FAST 1.0 used Xilinx's ISE tool chain to develop the FAST VAL and FAST 4W-NC prototype. The ISE tools can leverage some of Xilinx's pre-defined Verilog modules, but they are not intended for use with the embedded PowerPC 405 processor cores and thus would not necessarily be useful in next-generation designs [126]. There are also some JTAG features that are more easily integrated in Xilinx's Embedded Development Kit (EDK) tool chain than the ISE tools. These JTAG features are specific to using the embedded PowerPC processor found in later generation Xilinx FPGAs [112, 125].

Constraint management exists in the current Xilinx tools, but leaves much to be desired. Signal timing is the most important constraint that requires management. I/O drive strength is another useful constraint. The FAST 1.0 framework includes the initial signal-to-pin mappings and a matching top-level Verilog port list. The Xilinx PACE tool can be used for associating the Verilog port name with the device pin, but not much more than that [110]. A graphical interface that provides the signal timing, placement, and properties that can be easily manipulated would reduce the iterative FPGA bit file build process required to meet signal timing. This would especially be useful for managing internal bus signal propagation time. Starting with the Virtex-5 family of FPGAs, users will be able to skew signals at the I/O pad [126]. This partially addresses the signal timing issue from an external standpoint, but not the whole signal propagation including propagation through configurable logic. The first step to provide a solution would be to

integrate the PACE, Floorplanner, and FPGA Editor tools with the timing report [116]. For each FPGA pin, all of these properties (such as skew, drive strength) can be displayed, modified, and fixed in the UCF file.

Likewise, automatically generated FAST system Verilog would accelerate the prototyping process by producing a baseline architecture for researchers. High-level languages like System C can already convert C-like programs into Verilog [37]. The Verilog framework at the minimum would use parameterized Verilog modules that can be customized based on user input and combined to build a system. This Verilog framework would manage the port lists and all module connectivity and instantiations. This framework would also have the flexibility to include additional user-defined buses and control signals. With this framework, a researcher can build a functioning baseline system with the additional hardware connectivity required for their prototype, thereby minimizing connectivity and other user error. Researchers will be able to manage the entire FAST 2.0 hardware system by combining the Verilog framework with an FPGA and/or system graphical constraint manager.

FAST 1.0 was conceived to prototype new TLP architectures ***and*** enable software development. FAST 2.0 continues this trend for a broader class of architectures. The compiler and debugging infrastructure is critical for software development. Building a MIPS-based prototyping platform benefited from the availability of a plethora of compilers and debugging tools. The team had access to both native and cross-compilation environments using both embedded and general-purpose MIPS platforms.

The variety of tools and the availability of a native MIPS-based system accelerated initial software development by reducing the software validation time. Software validation was performed on the native MIPS-based system, leveraging the mature development and debugging environment. FAST 2.0 will benefit in a similar way if native systems can be used for validation. This would most likely skew the software defined processor core selection heavily towards the LEON3 core because it is SPARC V8 compliant and there are many native systems available [38]. Native and GNU compiler and debuggers are also widely available for SPARC like Forte, GCC, and DBX and GDB [13, 89].

The final component for a full system prototyping platform is the OS. It is essential to provide a configurable OS with the ability to add and subtract modules similar to Linux so that real workloads can be run on the prototype system. Linux is an ideal OS candidate given its wide adoption, and support by the open source community. Porting a multiprocessor version of Linux to FAST 2.0 completes this system. Likewise, having a configuration environment to manage the OS modules for peripheral support, multiprocessor support, and user-defined modules makes FAST 2.0 even easier to configure and use.

Finally, a graphical user interface (GUI) can bundle all of this together to create a FAST tool chain flow or FAST Flow. FAST Flow would manage all aspects of the FAST 2.0 system, enabling user configuration for all aspects of the design. FAST Flow's GUI can be drilled up or down, varying the level of detail and user control. The FAST Flow GUI uses windows or tabs for each specific operation: FAST VAL, prototype architecture, OS, APIs and drivers, applications, and FAST runtime. The FAST Flow GUI configures the PCB, maps the prototype architecture, integrates IP, specifies the OS and related modules, manages user-defined APIs and drivers, manages the software development environment for the applications, and provides the system programming, monitoring and observation, and debugging capabilities wrapped into a single framework that invokes all of the related tools. Table 6.1 lists all of the software features and infrastructure required to make an easy to use, full-system prototyping substrate for FAST 2.0.

Table 6.1. FAST 2.0 Software tools and software infrastructure.

1	Open source software development community or industry leadership
2	Open source and industry sponsored software repository
3	Out-of-the-box, cache coherent, multiprocessor OS
4	Improved user constraint file (signal timing) manager
5	Auto-generated Verilog modules and/or libaries
6	Software Development Environment (SDE)
7	Software tool chain GUI: FAST Flow GUI

6.3 FAST 2.0

The combined hardware and software vision for FAST 2.0 corrects the shortcomings of the previous generation. The hardware improvements are a result of the normal technology scaling. The software improvements are a result of the larger open source community or industry-led development providing more software infrastructure, creating a FAST software library. Focusing on the hardware, FAST 2.0 can be easily defined as a result of the FAST 1.0 experience. The building blocks for FAST 2.0 are similar: FPGAs, memories, I/O, LEDs, switches, and expansion headers/connectors. The main building blocks for the FAST 2.0 PCB are: 1200 I/O pin FPGAs, dual-ported 512K X 36 bit SRAMs, single-ported 2M X 36 bit SRAMs, dual-channel FB-DIMMs and some miscellaneous components like LEDs, switches, and headers. It should be noted that the FPGAs also have an additional 20 dedicated I/O channels for serial high-speed links that are not counted in the general purpose 1200 I/O pins. The FAST 2.0 software relies on a community to build it, and as such, only the hardware implementation of FAST 2.0 is presented here, from the bottom-up, the software vision having already been presented in the previous sections.

The FAST 2.0 PCB is designed for manufacturability (DFM) with respect to component selection and placement, although the components and their placement are not shown to scale in the following figures. FAST 2.0 requires professional PCB layout and routing due to the high-speed signals, making it an ideal outsourcing candidate to a professional PCB design house. Keeping the board dimensions below 14" X 17" also improves PCB DFM because this is a physical design constraint for some PCB manufacturing companies. Finally, the devices and device density on the PCB determine the number of PCB layers. Intelligent I/O pin mapping and routing can reduce the trace density, thereby reducing the number of layers. However, using fully populated FPGA fine-pitch ball grid array (fBGA) packages pushes the PCB layer requirements to at least 10 layers when considering power and ground planes as well as trace layers. Realistically, FAST 2.0 would use between 20 to 24 layers. Given a 14" X 17" PCB with 24 layers, using fBGAs with very small drill sizes produces a PCB with an estimated manufacturing cost around $8,000-$10,000/PCB for small quantities. There are additional components, assembly, and testing costs not included in the PCB manufacturing costs.

6.3.1 FAST 2.0 FPGAs

The Virtex-5 is the near term next generation FPGA that boasts several advanced features like I/O pad signal skewing, 6-input LUTs, dedicated high-speed serial links, and copious amounts of configurable logic [126]. This FPGA or something similar is the basis for the FAST 2.0 PCB. First, the processor FPGA (P FPGA) connectivity is presented. Each P FPGA is connected to all of the other P FPGAs using a wide 160-pin bus, as shown in Figure 6.1.

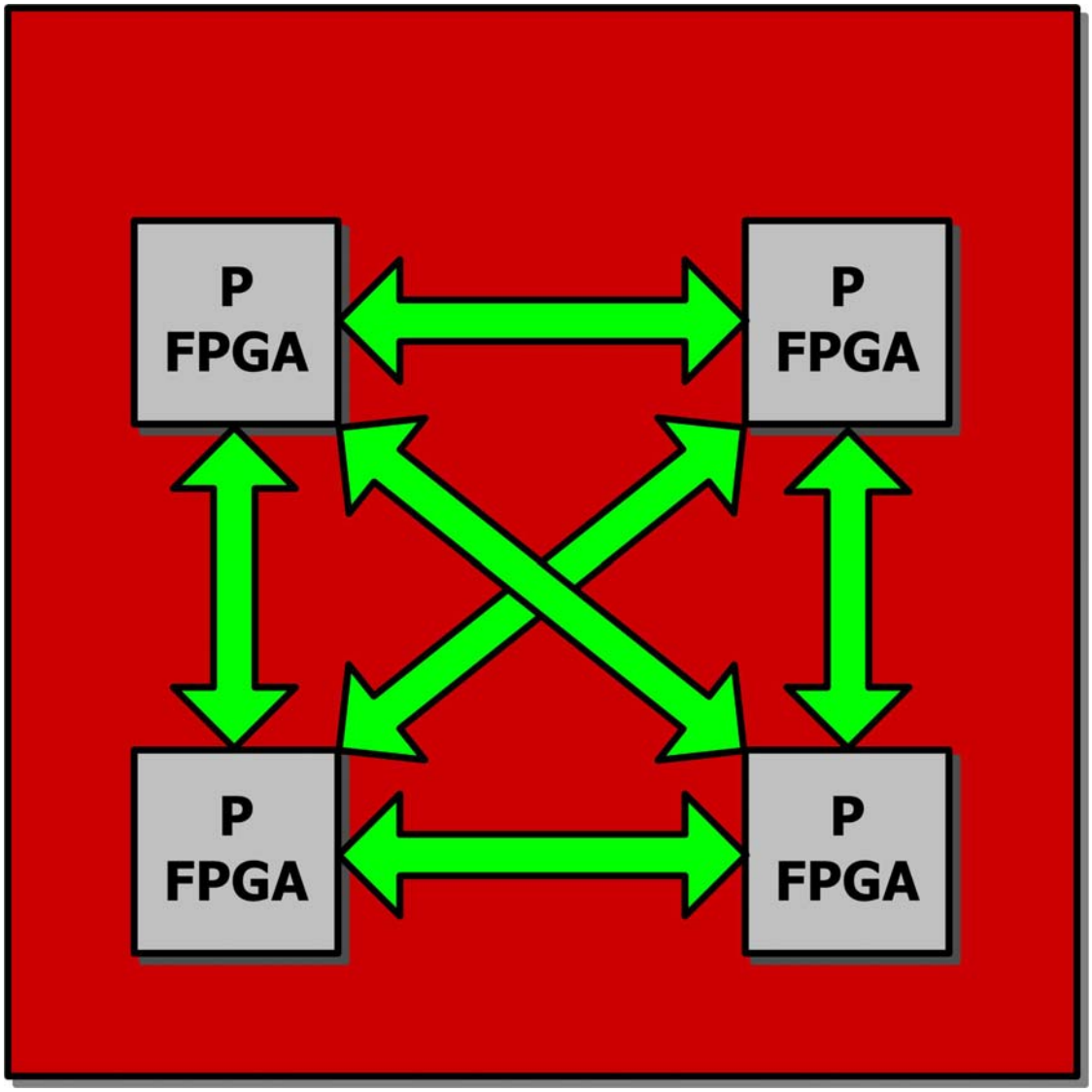


Figure 6.1. FAST 2.0 P FPGA connectivity.

Each wide bus can be subdivided, for example, into four 40-pin mini buses. This provides four dedicated FPGA-to-FPGA buses with a 32-bit data bus, 4-bits for parity, and 4 control bits per mini bus. Thus, if multiple processors are mapped to a single P FPGA, the communication between P FPGAs can be divided among the mini buses or aggregated across the large main bus. Likewise, if there is a single processor instantiated in the P FPGA, each mini bus can serve a different purpose or a single large multipurpose bus can be shared for high bandwidth communication. This dedicated FPGA interconnect uses 40% of the FPGA I/O pin resources or 480 pins.

The P FPGAs are used to instantiate prototype system processors. The P FPGA has the flexibility of implementing prototypes using the embedded PowerPC processor cores or software-defined processors (soft cores). Both options are retained because it increases the system flexibility and increases FAST 2.0's capabilities. Soft cores can be used for the main prototype processor and the embedded cores can implement other compute engines or off-load engines. Furthermore, a single embedded core could be shared across multiple soft cores. Likewise, the roles of the embedded cores and soft cores can be reversed. However, it should be noted that investigating processor design is more easily explored using soft cores like the LEON3 [38] because of the soft core's configuration granularity. The entire soft core is specified in software and thus can be modified by adding components internally or externally to the soft core. On the other hand, for the embedded processors, only processor external additions are possible using user-defined interfaces or predefined interfaces, like Xilinx's APU [105]. To be completely fair, limited internal changes are possible, for example, components like the data and instruction caches can be turned off in the embedded processors.

So far, there have been at most two embedded cores in a single FPGA, while it is possible to implement multiple soft cores in a single FPGA. The advancement in soft cores increases the attractiveness of soft cores as the foundation for a hardware prototyping platform. 6-input LUTs in the Virtex-5 also increases the number of soft cores that can be mapped to a single FPGA. As a result of the greater number of soft cores per FPGA, fewer FPGAs are integrated on FAST 2.0, but more cores are available per PCB than FAST 1.0. Using multiple cores per FPGA motivated both the fully connected P FPGA interconnect and the wide interconnect that can be partitioned as needed. This

interconnect, combined with soft cores, broadens the applicability of the FAST 2.0 prototyping platform beyond just the FAST 1.0 TLP architecture focus.

Technology advances enable FAST 1.0's separate RWC and SMC FPGAs to combine into a single FAST 2.0 service FPGA or S FPGA. This also corresponds to the HUB FPGA described in the FAST architecture, Chapter 2. The S FPGA shown in Figure 6.2 coordinates higher-level memory accesses, interprocessor communication, and off-PCB communication.

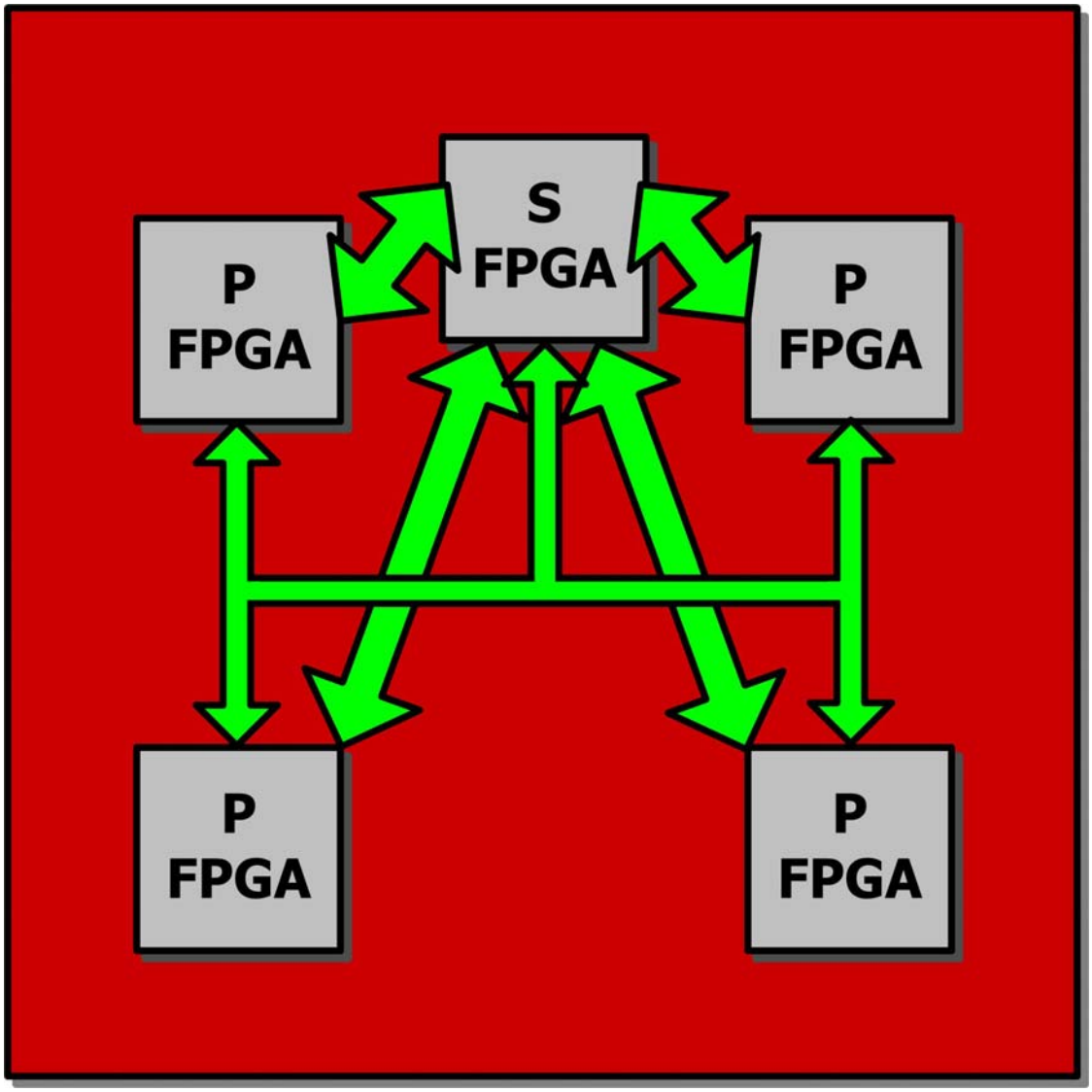


Figure 6.2. FAST 2.0 S FPGA to P FPGA connectivity.

The S FPGA has a 160-pin point-to-point connection to each P FPGA, as well as the shared P FPGA 36-pin bus. Each point-to-point bus serves as a high bandwidth wide bus, a collection of partitioned private buses, or something in between. The shared 36-bit bus is used for distributing shared data, like a global clock, or for synchronization purposes. Of course, these buses are not limited to just these functions, but are provisioned to provide a myriad of configuration options. The S FPGA can also be used for system clock generation or other global information distribution, similar to the PLD functions in FAST 1.0. The point-to-point and shared buses exhaust 676 pins or about 55% of the FPGA I/O pins. Finally, the S FPGA PCB placement reduces trace congestion in the center of the PCB for easier routing.

6.3.2 FAST 2.0 memory

FPGAs continue to lack sufficient on-chip memory resources for large-scale computer architecture research. Instantiating multiple soft cores in a single FPGA further exacerbates the on-chip memory shortage. To overcome the FPGA memory resource issue, FAST 2.0, like FAST 1.0, associates high speed dual-ported SRAMs with every P FPGA. Each P FPGA controls two banks of dual-ported 1024K x 36 bit SRAMs [55], shown in Figure 6.3 as a single black block. Each bank can support one to four SRAM chips for a total capacity of approximately 16 MB of level one cache. This is further augmented by the approximately 1 MB BRAM on each FPGA that can be used for auxiliary memory structures.

Unlike FAST 1.0, FAST 2.0 also integrates a dual channel FB-DIMM DRAM interface at each P FPGA, labeled as DRAM in Figure 6.3. This adds a second level of memory close to the processor with virtually unlimited capacity, when compared to SRAMs, on the order of several gigabytes. The FB-DIMMs can be used for data or statistics storage, or an upper level of cache. This P FPGA memory configuration provides two levels of configurable memories close to the processor. This memory can be used to implement varying data structures including caches of varying size, set associativity, and latency. Furthermore, various memory structure policies can be implemented in the P FPGA, e.g., write-through versus write-back caches.

The S FPGA also has memory associated with it to serve as higher levels of memory. As Figure 6.4 illustrates, the S FPGA has both SRAMs and FB-DIMM sockets for DRAM. There are two banks of SRAMs, with each bank represented by a gray box in Figure 6.4. Each SRAM is single-ported and is configured as 2M x 36 bit [42]. The bank can support from one to eight SRAM chips providing approximately 128 MB of shared memory. The S FPGA is similar to the P FPGA because it also provides approximately 1 MB of FPGA BRAM.

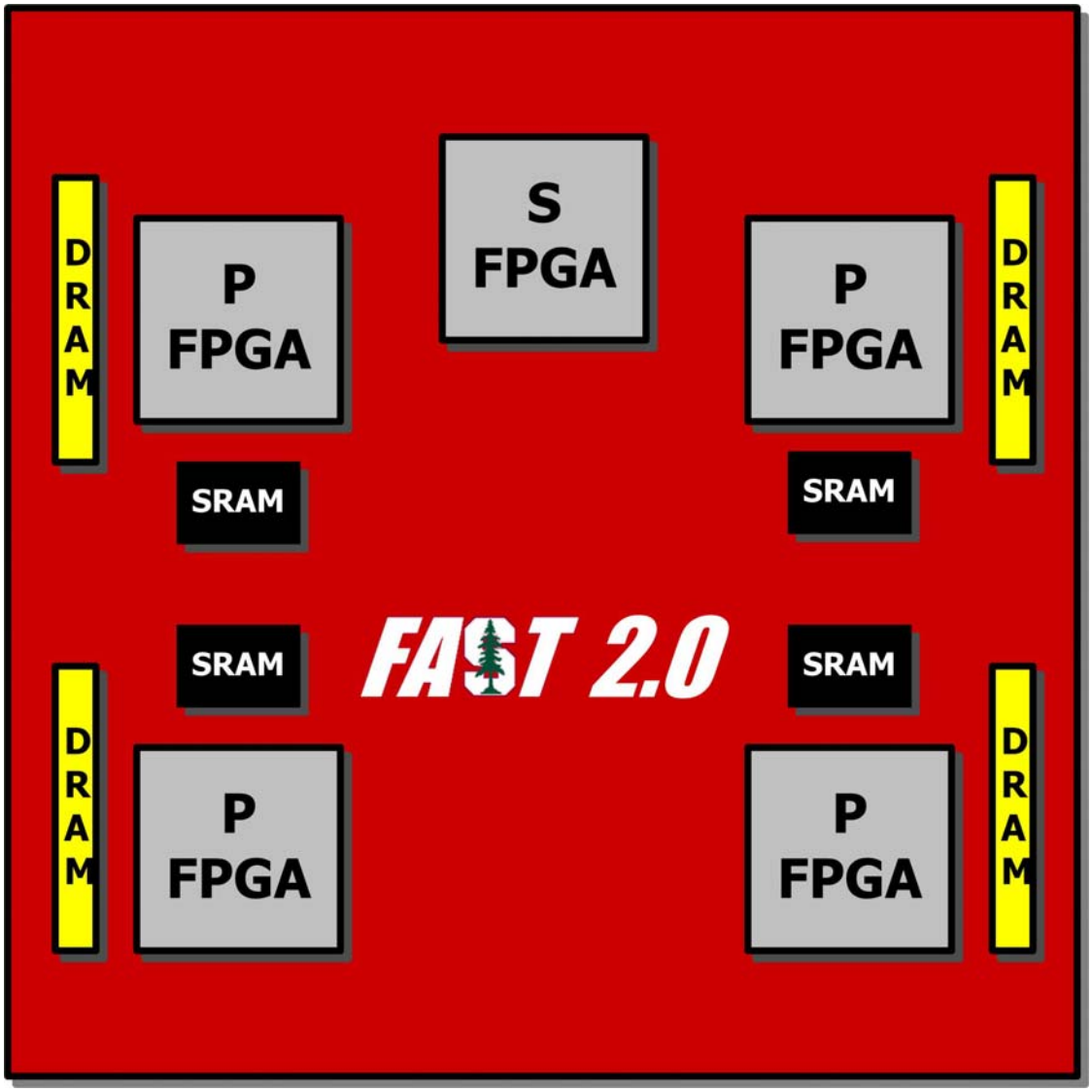


Figure 6.3. FAST 2.0 P FPGA memory configuration.

Single-ported SRAMs operate faster than dual-ported SRAMs. Thus, time division multiplexing and SRAM partitioning can be used for building high set associative shared

memories. The single-ported SRAMs can operate around 300 MHz, while the FPGA BRAMs are predicted to operate up to 550 MHz [42, 126]. In comparison to the currently available single-ported SRAMs, the dual-ported SRAM has a maximum operating frequency of 250 MHz [55].

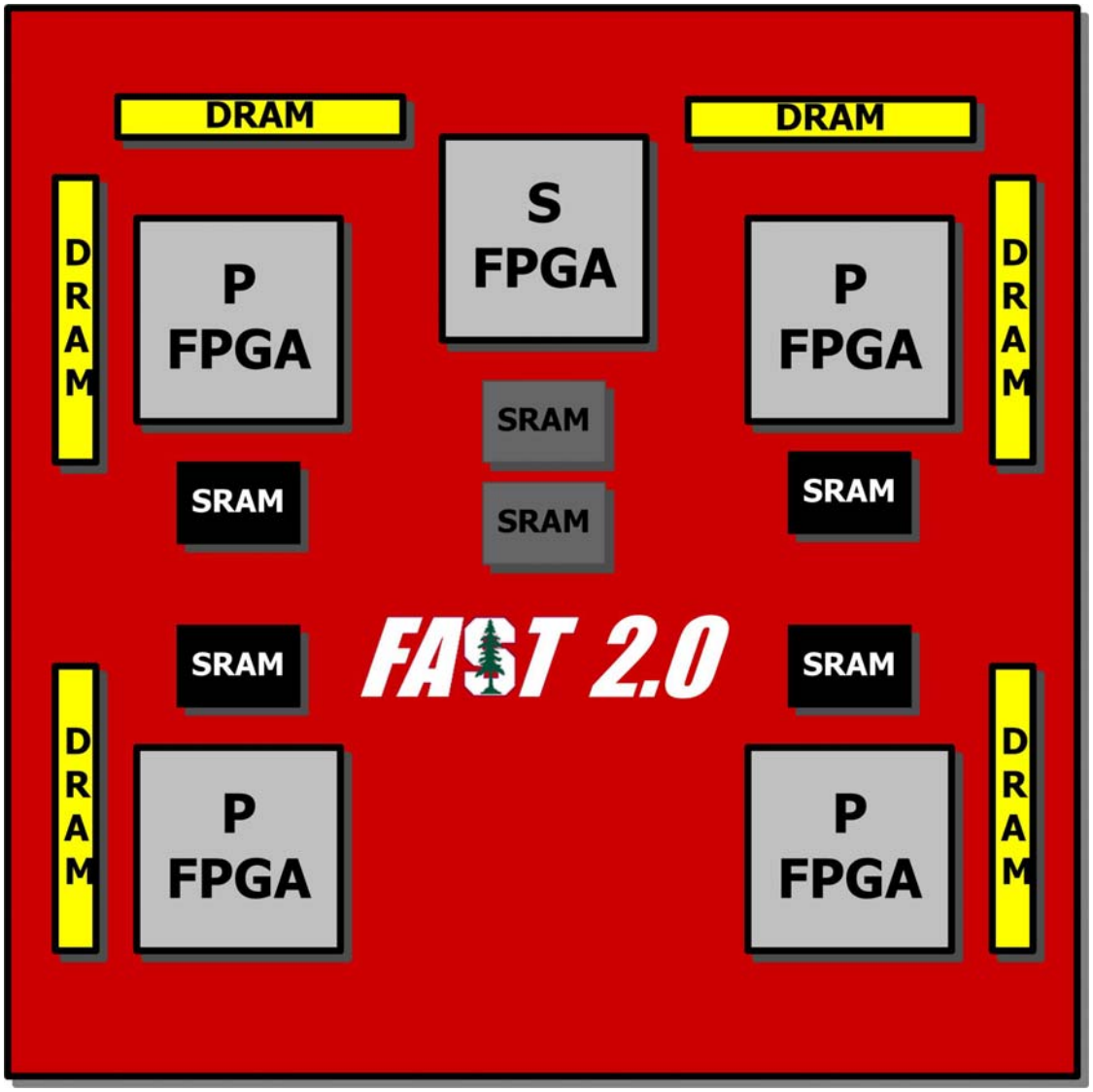


Figure 6.4. FAST 2.0 S FPGA memory configuration.

Two dual channel FB-DIMM sockets are controlled by the S FPGA and are labeled as DRAM blocks at the top of Figure 6.4. The two dual channel FB-DIMM interfaces, together, require only 220 pins. Each FB-DIMM interface requires far fewer pins compared to a single channel DDR interface, which requires about 250 pins. The two channels double the available shared memory capacity at the S FPGA. Overall, FAST 2.0

can be configured with an aggregate of tens to hundreds of gigabytes of FB-DIMM DRAM. From the P FPGA perspective, there are at least 4 levels of memory that the P FPGA has access to in its memory hierarchy. Two memory levels are available locally (L1 SRAM and L1 DRAM) and two memory levels are available at the S FPGA (L2 SRAM and L2 DRAM). If the BRAMs are included, there are six distinct memory levels that any P FPGA can access in its memory hierarchy. The quantity and capacity of the memory levels in FAST 2.0 exceeds current and next generation general-purpose processors. Furthermore, while the operating frequency of SRAMs is not going to dramatically increase, the capacity of the SRAMs will double with every silicon technology generation, at least for a while. Thus, the SRAM capacities specified for FAST 2.0 are actually lower limits and will probably be greater, depending on the future component availability.

Finally, soft cores may have a maximum operating frequency of 200 MHz, but actual maximum operating frequencies of 100 MHz can be expected for implemented designs given past experience. Thus, both the SRAMs and embedded processor still operate faster than their soft core counterparts, enabling time division multiplex and multi-way memory structures in a single soft core clock cycle. It should be noted that careful PCB layout and routing are required in order for the FPGAs, SRAMs, and DRAMs to perform at their peak operating frequency. For that matter, because of the high-speed operation, the entire FAST 2.0 PCB requires very careful PCB layout and routing.

6.3.3 FAST 2.0 I/O

Communication is the final aspect of FAST 2.0 that greatly improves on the FAST 1.0 implementation. The FAST 2.0 I/O infrastructure is designed for stand-alone operation, coupled to a host PC, or in a rack mounted, multi-PCB installation. Each FPGA has an associated I/O cluster that is mounted at one of the edges of the PCB. The I/O cluster placement facilitates cabling in rack-mounted systems. Figure 6.5 illustrates the I/O cluster placement. The S FPGA has its I/O cluster or S I/O Cluster placed at the top of Figure 6.5. The S I/O Cluster is responsible for traditional PCB communication for large FAST 2.0 compute fabrics by connecting into a backplane or other substrate for a traditional hierarchical interconnect. The S I/O Cluster also supports multiple 10 Gb Ethernet ports, as well as SATA interfaces all the way down to a simple RS-232 interface.

Providing various communication interfaces facilitates greater flexibility for data movement and resulting system configuration. For example, a traditional bus-based communication protocol and interconnect can be provided with FAST 2.0 by daisy chaining eight FAST 2.0's, in this case an arbitrary number of PCBs. However, the Ethernet ports could also be used for PCB-to-PCB communication to examine different latency and bandwidth characteristics. Taking this a step further, researchers can implement their own protocols and interconnect fabrics. The S I/O Cluster also has a generic digital I/O interface for other digital interfaces.

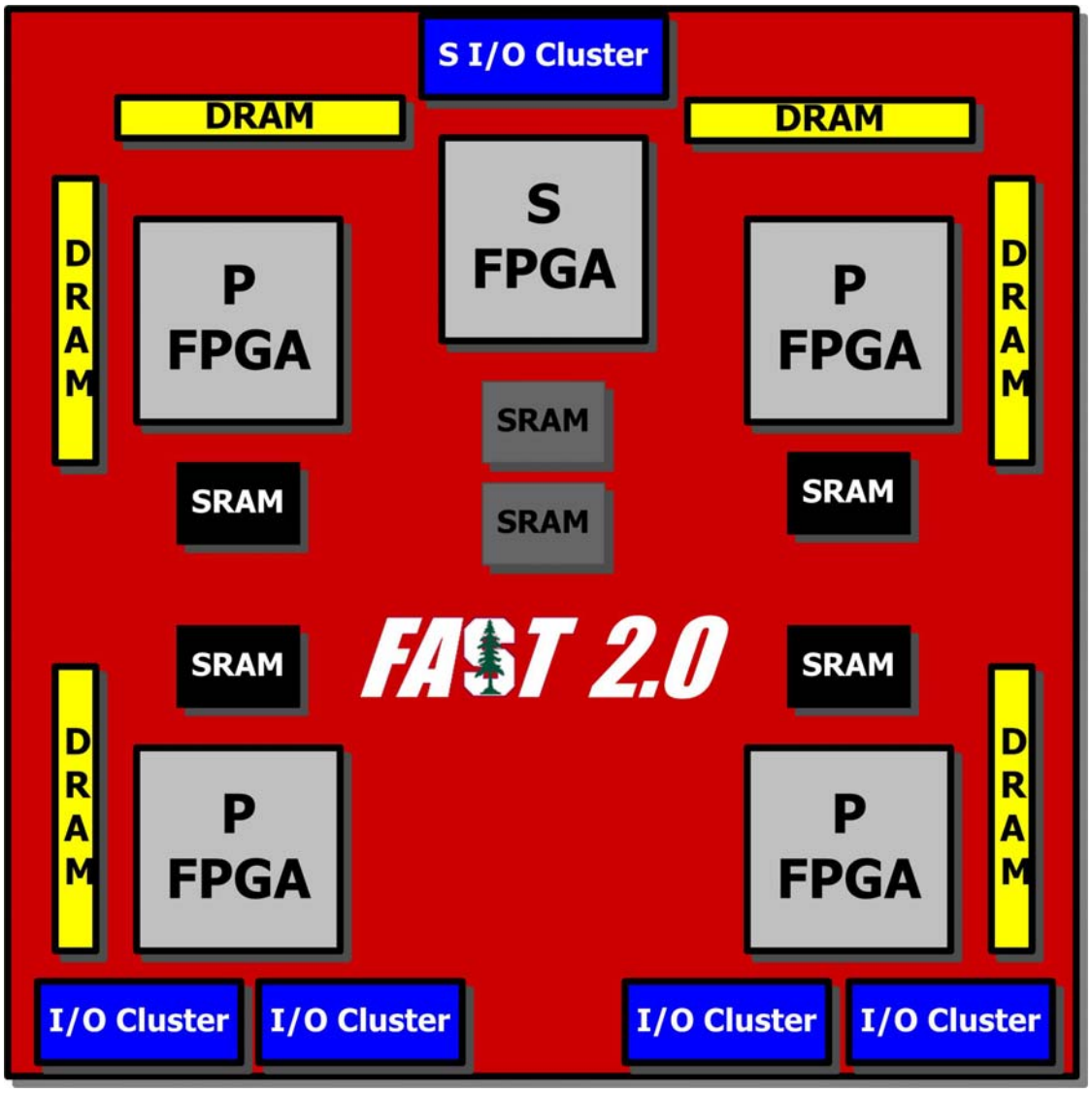


Figure 6.5. FAST 2.0 I/O cluster per FPGA and placement.

Each P FPGA I/O Cluster or I/O Cluster is similar to the S I/O Cluster, but has fewer I/O interfaces. The I/O Clusters are placed on the bottom edge of Figure 6.5. The primary function of the I/O cluster is to provide FPGA-local communication and expansion, for example, the I/O cluster enables FPGA-to-FPGA communication for network architecture studies. The placement of the I/O clusters takes into account space on the edge of the PCB, FPGA-to-FPGA communication and expansion, and thereby reduces physical cabling constraints. Each P FPGA has its own I/O Cluster. The I/O Cluster has multiple Ethernet ports, SATA interfaces, and a simple RS-232 interface. The P FPGA also has a generic digital interface for data generation or acquisition. This generic digital interface enables FAST 2.0 to process digital data for DSP or sensor network research, for example.

FAST 2.0 also has an interconnect bonus of 20 high-speed serial links per FPGA, for both S and P FGPAs. Initial, optimistic, publications state these high-speed serial links can operate at up to 10 Gb/s [126]. By combining pairs of high-speed serial links, a guaranteed 10 Gb/s is provided using a 50% degradation factor on all high-speed serial links. This degradation factor has been added to compensate for the lack of available hardware that operates at 10 Gb/s, at the time of publication. Currently available hardware can operate at the degraded bandwidth of 5 Gb/s per serial link [124, 125]. Ten high-speed serial links provide a myriad of interesting interconnect options for building large-scale FAST 2.0 prototyping fabrics. These high-speed links can be configured as InfiniBand links to provide a generic I/O framework [24]. Thus, FPGA-to-FPGA inter-PCB and/or intra-PCB communication is possible in a multi-PCB FAST 2.0 fabric, enabling novel interconnect and network research. Furthermore, using the high-speed serial links allows the other I/O devices to be used for out-of-band communication or for other purposes, increasing FAST 2.0 system flexibility. Overall, FAST 2.0 can connect over one hundred PCBs, in a variety of ways, to create very large prototyping systems with over one thousand soft cores.

6.3.4 FAST 2.0 system

Figure 6.6 illustrates a complete high-level view of the FAST 2.0 PCB. Both single PCB and rack-mounted (edge-viewable) LEDs, switches, and headers (L/S/H) are added to

the PCB to provide visual and manual I/O. Each P FPGA controls 32 LEDs and collects input from 32 switches. There is also a 40-pin header that can be used for a variety of purposes including a 40-pin IDE interface. All of the pin mappings for the P FPGAs are listed in Table 6.2.

As Figure 6.6 illustrates, the L/S/H of each P FPGA and S FPGA can be distributed between a central location and the edge of the PCB in a rack-mounted system. The S FPGA controls only sixteen LEDs and collects input from sixteen switches, versus 32 for the P FPGA's. The header pins are also reduced for the S FPGA down to 32 pins. These reductions are made in order to provide a 40-pin Compact Flash interface that serves as a non-volatile memory interface. Similar to the P FPGAs, the S FPGA can distribute the L/S/H components across the center of the PCB and on the edge for rack-mounted configurations. All of the pin mappings for the S FPGAs are listed in Table 6.3.

Table 6.2. FAST 2.0 P FPGA pin mapping.

P FPGA Connection	Pin Requirement	Total
P FPGA	3 x 160	480
2 Dual-ported SRAM banks	2 x 132	264
Dual channel FB-DIMM	1 x 110	110
S FPGA point-to-point bus	1 x 160	160
S FPGA shared bus	1 x 36	36
I/O cluster	1 x 46	46
LEDs	1 x 32	32
Switches	1 x 32	32
Headers	1 x 40	40
Total I/O Pins		**1200**
Dedicated high speed serial links		20

Current Compact Flash devices can store up to 8 GB of data [82]. Technology improvements point to increased Compact Flash capacity in the future. Today and moving forward, the Compact Flash provides ample storage for most FAST 2.0 configurations and applications. Large commercial server and scientific applications that require large databases or large data sets, like TPC-C, require large hard drives instead of Compact Flash devices, which cannot store the hundreds of gigabytes of data required for these applications. Alternatively, Ethernet or InfiniBand can be used to access large datasets or databases.

Figure 6.6 shows the next generation PCB for the FAST 2.0 system. It has at least twice as many resources when compared to FAST 1.0. FAST 2.0 integrates a DRAM interface for the P FPGA and S FPGA. FAST 2.0 has sixteen times as much processor local SRAM and at least twice as much shared SRAM. FAST 2.0 can implement more complex interconnects and has the ability to build a compute fabric an order of magnitude larger than FAST 1.0.

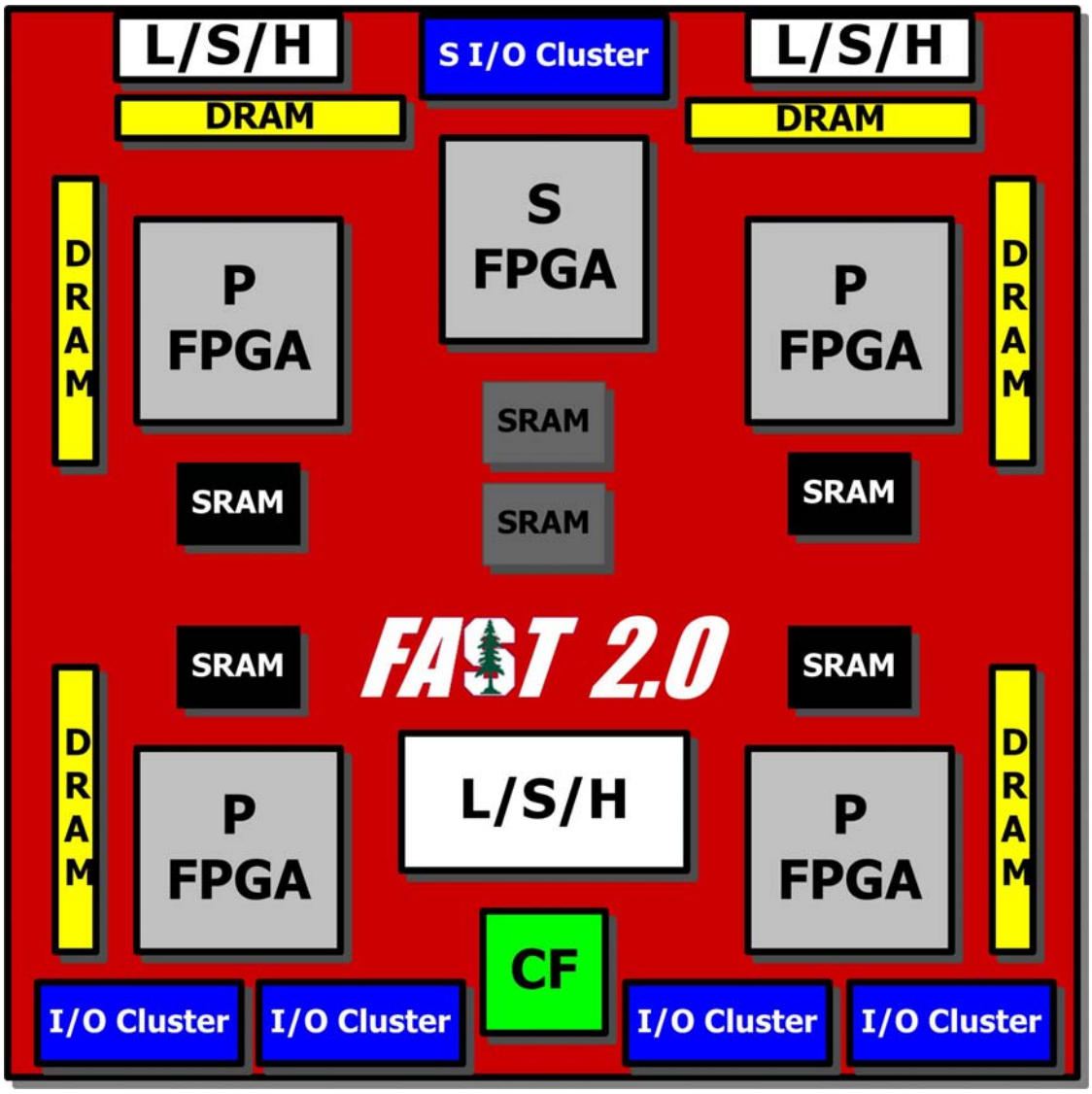


Figure 6.6. Complete FAST 2.0 PCB design.

FAST 2.0 retains the ability to configure systems with very large on-chip memory systems and to configure memory latencies for these systems. FAST 2.0 also expands its memory system infrastructure to include DRAM, and hard drive interfaces like SATA, providing

the entire hardware memory stack of a modern computer system. Finally, using soft cores, FAST 2.0 should be able to achieve a true 100 MHz operating frequency with a single cycle load-use latency. Thus, each FAST 2.0 PCB would have a peak performance of at least 800 MIPS (8 processors X 100 MIPS) if each P FPGA implemented just two soft cores. Each P FPGA should be able to implement four soft cores, which would yield 1.6 GIPS peak performance, far better than any currently available sequential software simulator with high fidelity.

Table 6.3. FAST 2.0 S FPGA pin mapping.

S FPGA Connection	Pin Requirement	Total
P FPGA	4 x 160	640
2 single-ported SRAM banks	2 x 66	132
Dual channel FB-DIMM	2 x 110	220
P FPGA shared	1 x 36	36
I/O cluster	1 x 68	68
Compact Flash	1 x 40	40
LEDs	1 x 16	16
Switches	1 x 16	16
Headers	1 x 32	32
Total I/O Pins		**1200**
Dedicated high speed serial links		20

6.4 Comparing FAST 1.0, FAST 2.0, and BEE2

Many of the feature differences between FAST 1.0 and FAST 2.0 result from technology improvements. Table 6.4 compares FAST 1.0 to the BEE2 PCB and the proposed FAST 2.0 PCB. BEE2 was designed for DSP and wireless ASIC prototyping [21]. Even though FAST 1.0 predates BEE2, FAST 1.0 contains more programmable logic measured in the number of slices. It should be noted that Xilinx redefines the notion of a slice with Virtex-5, with a Virtex-5 slice equal to two older generation Virtex slices. Beyond technological advances, the main differences between the FAST PCBs and the BEE2 PCB are the FPGA interconnect topology, clock distribution, use of SRAM, and an integrated FPU. These differences stem from the fact that the BEE2 PCB was designed for ASIC and, in particular, DSP emulation.

The FPGA interconnect topology determines the class of architectures that can be prototyped by the PCB. The FAST PCBs target CMPs in general. As a result of this general focus, both FAST PCBs implement a fully connected point-to-point (P2P) interconnect to facilitate rapid and efficient inter-processor communication. Similarly, multiprocessor systems may require global communication, which is facilitated by the shared bus. The shared bus can be used for global synchronization or broadcast data. FAST 2.0 proposes a similar topology and uses a newer technology, enabling wider buses.

Table 6.4. FAST 1.0 PCB comparison to BEE2 and FAST 2.0.

Feature	FAST 1.0	BEE2	FAST 2.0
Prototyping focus	MP/CMP	ASIC/DSP	MP/CMP
FPGAs (Slices)	10 (264,192)	5 (165,440)	5 (518,400*)
FPGA interconnect topology**	P2P, Shared	Ring	P2P, Shared
Clock distribution	Flexible	Rigid	Flexible
BRAM	640 KB	3 MB	5 MB
SRAM	68 MB	0 MB	196 MB
DRAM	Expansion	Yes	Yes
Compact Flash	Expansion	Yes	Yes
Embedded processor core	No	Yes	Yes
Software-defined core	Yes	Yes	Yes
Floating-Point Unit	Hard	Soft (-)	Soft (+)
JTAG	Yes	Partial	Yes
RS-232	Yes	Yes	Yes
SATA	No	Yes	Yes
Ethernet	Yes	Yes	Yes
Infiniband	No	Yes	Yes
High-speed serial links	No	No	Yes

* Virtex-5 slice is equivalent to 2 older generation Virtex slices [126]. FAST 2.0 contains 209,200 Virtex-5 slices.

**The FPGA interconnect topology for FAST 1.0 and 2.0 is a combination of a point-to-point (P2P) fully connected network and a shared bus network. The fully connected topology allows for single hop inter-processor communication and the shared bus enables broadcast. The BEE2 implements a ring network for the user FPGAs. The ring network requires two network hops to communicate to non-nearest neighbor FPGAs.

The BEE2 PCB was designed for ASIC and DSP research. The BEE2 PCB has one control FPGA and four user FPGAs. The control FPGA is similar to a service processor or master node. New designs are generally mapped to the user FPGAs and initiated, monitored, and controlled by the control FPGA. The PCB was designed with ring

FPGA interconnect topology for the user FPGAs. This topology restricts the communication by forcing data to be transmitted through an intermediary for the user FPGAs that are not nearest neighbors. This FPGA interconnect topology may complicate architectures that require flexible inter-processor communication.

Clock distribution is also flexible in the FAST PCBs. In FAST 1.0, the PLD distributes the clock to all the FPGAs. The PLD can be used to gate the clock to any of the FPGAs at any time. Furthermore, the PLD has two different clock input sources: the clock crystal socket and a header for an external frequency generator. Thus, using this framework, FAST 1.0 has a very flexible clock distribution infrastructure. Once the clock is distributed to the FPGAs, the internal FPGA clock management hardware can be used to produce various permutations of the system clock. FAST 2.0 builds on this flexible clock distribution infrastructure. The latest Virtex-5 FPGA also has more clock generation and skew management resources than previous Virtex FPGAs.

The BEE2 PCB uses clocks and clock buffers to distribute a global clock. Like the FAST designs, the BEE2 PCB can also use external clock sources using SMA connectors. The improvement of digital clock management (DCM) hardware inside the FPGA has dramatically increased the clocking resources available, thereby reducing the need for global clock management. However, there is no central clock gating mechanism like that which exists on the FAST PCBs. This gating mechanism could be useful for the central processor if it acts like a service processor to control the PCB.

FAST 2.0 is able to integrate DRAM as well as SRAM for every FPGA, expanding the memory hierarchy. Again, this is a feature that is enabled by technology advances. Providing both types of memory enables a wider array of architectures. The SRAM can be used for cache structures and has functional characteristics similar to caches. Likewise, DRAM can be used at each FPGA as a higher level of memory, enabling the FPGA to compensate for DRAM refresh and memory request collisions. Each FPGA also has access to far more memory than what was available to each FPGA in FAST 1.0 or the BEE2 PCBs.

The BEE2 PCB focused on ASIC and DSP research and in general, ASIC and DSP architectures do not use cache structures because the application space requires deterministic memory access latencies. DRAM is sufficient as a deterministic storage, but not as a foundation for cache structures, because of the issues with DRAM refresh. Of course, if the DRAM access time is set longer to incorporate the extra latency required to resolve DRAM refresh collisions, then the DRAM can function as a slow, large cache.

The FAST 1.0 PCB is the only PCB solution that also integrated a dedicated hard FPU. The FPU broadened the application space of the prototyped architecture. There was no need to implement very slow floating-point emulation that would be required for later generations, in this case the BEE2 and FAST 2.0 PCBs. Thus, floating-point applications operate with reasonable latencies compared to the CPU in FAST 1.0. The BEE2 and FAST 2.0 PCB integrate the software-defined FPU differently. The BEE2 PCB requires the use of the multi-cycle PLB interface or some other custom interface, while the FAST 2.0 PCB can use the much faster APU interface [51, 104, 118]. The APU interface dramatically reduces the communication time between the processor and the FPU, realizing more realistic FPU latencies with respect to the embedded CPU. If software defined processor cores can be directly coupled to software-defined FPUs, FAST 2.0 has the performance advantage over BEE2 because of the technology speed step and higher logic densities using 6-input LUTs. This makes the software-defined FPU option for the FAST 2.0 PCB much better than the BEE2 software-defined FPU.

Finally, as Table 6.4 illustrates, FAST 2.0 merges the benefits of new technology with FAST 1.0 and BEE2 PCB design concepts, creating a more flexible hardware prototyping platform. Like its predecessors, FAST 2.0 also incorporates standard expansion interfaces for new devices and I/O. This expansion interface was especially useful for FAST 1.0 because it enabled both a Compact Flash and DRAM interface. Likewise, FAST 2.0 will be able to benefit in much the same way using its expansion interfaces. FAST 2.0 clearly demonstrates its broader applicability and increased flexibility by incorporating both DRAM and SRAM, combined with a wider, more flexible FPGA interconnect topology using the latest FPGAs.

6.5 Are flexible hardware prototypes the answer?

FAST 2.0 has been designed for manufacturability and high performance. It should outperform FAST 1.0 and any other currently available hardware prototyping system. There are still several low level details that require definition before it can be built, but this ambiguity only exists in the complete definition of the L/S/H and I/O Clusters, or about 10% of the available I/O pins. These final decisions enable on-PCB peripheral I/O devices, like a keyboard, and other miscellaneous interfaces. FAST 2.0 delivers 6-12 times higher peak performance compared to FAST 1.0 and copious amounts more memory: sixteen times more L1 memory, twice as much shared SRAM, and gigabytes of DRAM and Compact Flash. However, the question still remains if a flexible hardware prototyping system is the answer for software development and system implementation and validation.

FAST 2.0 does not usurp software simulation from the computer architecture research life cycle. FAST 2.0 expands its prototyping applicability and performance. FAST 2.0 still relies on software infrastructure for wide spread adoption and use. Without an out-of-the-box hardware and software solution with a working multiprocessor example, FAST 2.0 suffers from the same underutilization as FAST 1.0. Given a baseline example to build from and a rudimentary collection of tools, FAST 2.0 can gain traction in the research community because of its performance and reuse. FAST 2.0 has the same cost barrier as FAST 1.0, but this cost barrier is far less than developing, manufacturing, and debugging actual custom microchips.

FAST 2.0 or systems like it are required for computer architecture or parallel systems research because of the emergence of chip multiprocessors (CMPs) as the dominant processor architecture. Every additional processor in a CMP configuration linearly slows down sequential software simulators. Likewise, tracking memory interactions in the large on-chip caches also requires a lot of simulation overhead. Both the number of processors and the size of on-chip caches are on the rise, further exacerbating the software simulation speed/fidelity tradeoff problem. FAST 2.0 and systems like it enable full system prototyping at hardware speed. Prototyping novel architectures exposes any shortcomings in their concept and provides the ultimate credibility with reduced

validation effort. More specifically, new architectures developed in software have no reference systems for validation and generally this validation is beyond the scope and resources of a small (on the order of five or less people) research group. FAST 2.0 leverages pre-validated building blocks, reducing the validation effort to build larger, working hardware systems.

Finally, CMPs require an iterative software development approach because of the complex memory and shared resource interactions that change with each new CMP configuration. As a result, there is no "golden binary" that works across all CMP configurations. The tremendous wall clock time required for software-based CMP simulators makes iterative software development impractical and very tedious at best. FAST 2.0 is a viable alternative until and unless parallel simulation with very low slowdown becomes a reality. Parallel programming is very difficult for existing hardware platforms. Creating a correct and accurate parallel software simulation infrastructure is even more difficult. FAST 2.0, a PCB and collection of software modules, libraries and tools, is the next generation hybrid hardware and software solution addressing system prototyping and software development, as demonstrated with FAST 1.0.

Chapter 7

Conclusions

FAST is another successful Stanford hardware project. The FAST PCB was conceived and designed at Stanford and manufactured and assembled by Sanmina's prototyping PCB facility. The result is a working PCB with *no PCB rework!* FAST ushers in an era of FPGA prototyping that will facilitate software development for new computer architectures. FAST combined various prior generations of hardware to achieve this goal. Moving forward, the next generation of hardware prototyping platforms will have more features, greater flexibility, and fewer drawbacks. FAST 2.0 was proposed as the next generation prototyping system that would far exceed current hardware prototyping capabilities. However, the next generation hardware substrate is nothing without various levels of software. Creating an open source community using a standardized hardware prototyping platform will produce the software foundation required for a successful FAST 2.0 system.

FAST 1.0 and FAST 2.0 are not without their predecessors. Fortunately, FAST also has a list of successors that validate the need for hardware prototyping in academia and research in general. To wrap up this dissertation, a discussion of the scope of research facilitated by FAST is followed by a critical evaluation of the FAST system. FAST 2.0 is revisited and discussed. Finally, FAST conclusions are provided given the benefit of hindsight.

7.1 FAST scope

A simple, decoupled 4-way CMP prototype demonstrated the capabilities of the FAST system. By developing the FAST VAL and integrating the SRAM chips, L1C, CP2, and RWC FPGAs, FAST 4W-NC provided a proof of concept for hybrid prototyping platforms. This simple prototype illustrates the ability to use a central communication structure as a point of coherence for more complex systems like Hydra. However, FAST is not limited to TLS designs like Hydra. FAST can also be used to implement other

TLP-focused architectures or CMPs. Stanford University projects like TCC [45] and Smart Memories [69] are systems that can be mapped to FAST. Other embedded CMP systems can also be mapped to FAST, like MPOC [80].

Unlike other systems [9, 20, 21, 33], FAST was designed and built to implement CMP/MP architectures to investigate methods focused on parallel programming paradigms and hiding or ameliorating the memory latency problems of future processor designs. FAST can also prototype multiprocessor systems by changing the memory latency. Unlike RPM, FAST also incorporates ample configurable logic that can be used to implement more than just a memory system [9, 33]. FAST uses newer components, which enable larger designs. Likewise, the BEE2 board was developed after FAST and incorporates newer components. The BEE2 project was focused on DSP architectures, whereas FAST was designed to implement TLP architectures. The BEE2 PCB also uses FPGAs with two embedded hard processor cores, but interfacing to these cores requires a multi-cycle interface bus [51]. By leveraging the next generation hardware and broadening the system capabilities, FAST 2.0 improves upon all previous hardware prototyping platform and yields at least two times the performance of any previous system. As Figure 7.1 illustrates, FAST or systems like it enable research to incorporate hardware prototyping and implement designs sooner, as has been demonstrated by FAST 4W-NC and the TCC Atlas project [75].

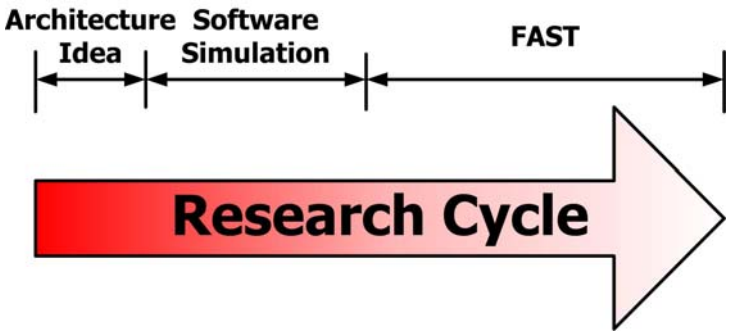


Figure 7.1. FAST reintroduces hardware back into the research cycle and brings it in even earlier than previous hardware prototypes.

FAST 2.0 was proposed as a system that extends the scope of research compared to previous configurable hardware prototyping systems. FAST has demonstrated that both hardware and software research is enabled by this platform. Furthermore, by providing

the necessary base infrastructure, FAST or systems like it provide a reusable validation platform that may accelerate the flow of research ideas from academia to industry. Likewise, these hybrid systems can be used much like current software simulators that provide APIs for extensibility, as discussed in Njoroge et al. [75].

Not so FAST

FAST 1.0 has significant limitations that push many current research projects outside of its scope. Several aspects of software research like compilers, operating systems, and application tuning, are outside of FAST 1.0's scope because of the lack of software infrastructure and tools. However, it should be noted that this is not a fundamental system limitation, but a software resources limitation. Similarly, the hardware development time also inhibits hybrid prototype development, but this will improve with advancement in tools and the creation of software module libraries. FAST 1.0 does have fundamental limitations that prevent prototyping processor pipeline design, complex large memory structures, and complex processor designs. Some of these issues are resolved with FAST 2.0 and the use of software-defined processor cores, but physical resource limitations will always restrict the designs that are mapped to FPGA-based prototyping platforms. Finally, advances in FPGAs are required to easily implement real hardware, like fully associative memories or gang cleared or set memory structures. The desire to build real prototypes will continue until these features are available to computer architects in hybrid prototype platforms.

7.2 FAST grade

The initial goal of the FAST project was to create a prototyping platform that could be reused across multiple research projects. By developing a hybrid system, FAST tried to combine the best benefits of software and hardware systems. Table 7.1 revisits the initial comparison Table 1.1 with FAST specific grades instead of the hypothetical goals presented in Chapter 1. Table 7.1 provides a critical evaluation of the system and its future use. A single full-time graduate student developed the entire FAST system. A larger FAST development team would have been able to develop a more mature FAST

system. Table 7.1 evaluates the current system and not its potential. FAST has demonstrated that it can be much more than it is, given the development resources.

Table 7.1. FAST comparison to software simulators and hardware prototypes.

Feature	Software	Hardware	Ideal Hybrid	FAST
Reuse	A+	F	A+	D
Flexibility	A+	F	A+	B
Transparency	A+	D	A+	A
Reproducibility	A+	C	A+	A+
Community	A	D	A	C
Development Time	A	F	A	B
Cost	A+	F	A	B
Credibility	F	A	A	A
Performance	F	A	A	C
Software Development	F	A	A	B

Table 7.1 highlights where FAST deviates from the initial features in the white boxes for software, hardware, and an ideal hybrid solution. FAST has the potential to be reused, but currently, there is not enough infrastructure in terms of software and additional hardware (PCBs). The lack of mature software prevents more hardware systems from being developed and distributed. There is no critical mass in the research community to use this platform. However, the formation of the RAMP project [11] demonstrates the ideas inspired by FAST are gaining critical mass. FAST is flexible and can be used for a variety of projects. However, it is not as easy to develop and use as software simulators. The initial prototype used very few of the available resources, demonstrating that many more complex designs can be mapped to the FAST system.

FAST did meet or nearly meet three of its design goals. FAST provides approximately the same transparency that is found in software simulators. FAST is limited in the number of asynchronous statistics that can be collected. However, FAST provides a variety of ways to collect data and process that data. FAST may require a combination of hardware and software to collect statistics, making it a bit more cumbersome than a software simulator. Likewise, the results from FAST are reproducible with the same determinism of software simulators when no external events are required. Furthermore, given an OS and I/O, FAST can reproduce the results of real systems, exceeding the ability of software simulators to reproduce real systems, giving it credibility.

While the FAST project community members can be counted on a single hand, the overall community interested in using FPGAs to conduct computer architecture has grown. This community even helped the FAST project by providing some utilities that swapped BRAM contents in the Virtex FPGAs [25]. Moving forward, projects like RAMP will not only leverage the existing community, but unite a large part of the academic community using FPGAs to investigate computer architecture. The development time of the FAST PCB and software was reasonable given its serial nature and external delays. Prototyping the FAST 4W-NC system was relatively quick. It only required a few months to run simple applications on the PCB. This is similar to developing a simple software simulator from scratch.

Finally, the system cost was relatively low if human capital and donations are not accounted for in the process. The actual hard cost for the FAST project including test equipment was about $25,000 to produce one fully functional PCB and 3 additional unpopulated PCBs. If FAST were to go into low-volume production with donated Xilinx parts, the cost per unit would be about $7,500. There is about $4000 in parts, mainly SRAM chips, and $3500 in manufacturing, assembly, and testing fees. Any capital costs cannot compare to the free software simulators. However, the FAST software would be distributed for free with the system, reducing the prototyping effort. At less than $10,000, FAST is a bargain compared to the cost of producing a chip.

FAST also met the credibility design goal and was validated using small test programs and other reference hardware [64]. Because FAST is a hardware platform, the results are not subject to the same lack of confidence associated with software simulators. Similarly, future hybrid prototyping platforms will enjoy the same high credibility.

Finally, FAST is two orders of magnitude slower than current hardware. The performance grade was decremented by one for each order of magnitude slowdown compared to real hardware. Ironically, prototypes are rarely as fast as real hardware, so this comparison is probably overly harsh given that one-off prototypes are approximately one order of magnitude slower than the fastest general-purpose hardware. Likewise, FAST was downgraded for the software development difficulty. The FAST project leveraged existing working systems for software development. This dramatically

accelerated the software development process, but may not be realistic for other systems. Had the FAST project not been able to leverage a native or emulated software development platform, FAST would have gotten a much lower grade.

Obviously, given more time, the FAST system could be developed into a mature configurable prototyping platform. Given the lack of resources and the short development time, FAST was able to demonstrate a proof of concept. Overall, the FAST system has far exceeded the initial goals. The combination of the FAST 4W-NC prototype proof of concept and system functionality tests demonstrates that FAST can prototype a variety of TLP architectures. Given an experienced PCB development team, the FAST project could have reduced the 1.5 man-years required for the PCB design and production down to 0.5 man-years. Similarly, with a larger FAST team, the software environment could have been developed in parallel, overlapping the one man-year of software development time with the PCB design and production time. Overall, the tools required to develop the entire system were barely sufficient to accomplish the tasks. Hopefully, future systems will have better tools and more resources to develop even better configurable prototyping platforms.

7.3 FAST 2.0

The current computer architecture landscape requires a rapid prototyping platform for small and large-scale research. FAST and BEE2 demonstrate how the evolution in hardware can enable more complex research. FAST 2.0 was presented as a platform with even broader applicability compared to FAST 1.0 and the BEE2 PCB using the next generation of FPGAs, SRAMs, and interfaces. The FAST 2.0 PCB must be designed by a professional PCB design firm to enable high-speed communication and is beyond the capabilities of a graduate student design team.

FAST 2.0 provides an improved hardware platform. FPGAs continue to increase in density and speed, but lack significant on-chip memory for building large caches and other memory structures. FAST 2.0 integrates both SRAM and DRAM to address the on-chip memory shortage. Using time-division multiplexing and multiple ports enables the prototyping of multi-way set associative caches and other interesting memory

structures. The larger FPGAs also provide the flexibility to use embedded hard processor cores or software-defined cores like the LEON3 [38]. Using these cores as the system foundation enables both rapid processor prototyping and/or system prototyping. Moving forward, software-defined cores are more attractive because of their flexibility, extensibility, and availability of native systems for software development [89].

Finally, FAST 2.0 or similar systems will only be successful with a community to develop and maintain the software and support systems. Building the hardware requires a finite amount of time that is amortized over each instance of the hardware. However, without the software, the hardware is useless, preventing research community adoption. FAST 2.0 tries to build a hardware platform that can be used across multiple architecture domains. This hardware is only useful if software exists. Research institutions will buy and use FAST 2.0 if both pieces of the prototyping puzzle exist, the PCB hardware platform and a rich software library and development environment. Providing the working out-of-the-box examples is key to FAST 2.0 or any other system adoption, similar to current software simulators.

7.4 Conclusions

It is a pleasure to conclude that FAST was a successful project. Along the way, several lessons were learned that made the FAST project more difficult than necessary. These lessons and conclusions are enumerated in the next section and described in more detail in the last section.

7.4.1 Lessons and conclusions

Each of the following summary points below are described in more detail in section 7.4.2.

1) Complex PCB designs require high-end PCB design tools. These tools are worth the extended learning curve.
    a. Future PCB prototyping platforms using high-speed interfaces and high-speed signaling will require professional layout and routing.
    b. Future PCB unit costs will hover around $10,000 assuming donated configurable logic and many other hardware components.
2) Undergraduate research in the early hardware development stage was unused.

3) A team is required to develop the hardware and a community is required to develop the software in a reasonable timeframe.

4) A software library is required to accelerate research using a hybrid platform.

5) Better timing and constraint management tools are required for FPGAs, because timing and constraint management requires significant effort.

6) Incorporating redundant systems in hardware was essential and actually accelerated FAST software development.

7) Providing a variety of PCB I/O options, from simple and slow to complex and very fast, addresses different I/O requirements and increases overall system flexibility.

8) The FAST 4W-NC proof of concept and system functionality tests demonstrate that FAST can prototype a variety of TLP architectures.

9) A PCB with ***no rework*** was developed.

10) Intelligent FPGA interconnect broadens the prototyping domain of the hybrid systems.

11) The next generation hybrid system will be even better!

7.4.2 Detailed conclusions

As a result of shrinking research dollars, CAD tool support has been drastically reduced over the term of this project. This major limitation makes future in-house large-scale PCB efforts impractical. One of the major benefits of FAST and similar projects, is its ability to accelerate research using a common, shared infrastructure. However, this research acceleration cannot happen with an initial infrastructure that is severely under-resourced. The FAST project could have had greater impact if it had not encountered significant implementation delays. The PCB implementation delays were the result of using mid-grade OrCad PCB design tools, from Cadence, and not the high-end Allegro PCB design tools [16]. Because the next generation PCB will incorporate a variety of high-speed interfaces and high-speed signaling, the actual design will require more money and/or a professional design team. If the next generation has similar industry support in terms of donations and free samples, the PCB components, manufacturing, assembly and testing costs will remain about $10,000 per unit for low-volume production.

The second main problem FAST encountered was mismanaged and wasted human capital. The project initially had two graduate students. This enabled parallel development

in the project. There are many facets to a project of this magnitude, but a simple bifurcation occurs along the hardware and software lines. Once the first graduate student left, the project had to start over because the hardware designer had left. This forced the timeline out by over two years from initial inception.

Several undergraduates also had the opportunity to contribute to the FAST project. Unfortunately, without the FAST PCB, their contributions were very limited and have served as time sinks for the senior member of the project.

In order to make useful contributions, two things must be present. First, a hardware system must be available with several small manageable projects that are off the critical path. Second, the project must not always be on the critical path. Because of the delays incurred at the beginning of the project, the focus of the project changed to bare minimum functionality instead of building infrastructure. Furthermore, the project lacked the resources to do significant FAST software development for the FAST PCB. This is the final fatal flaw that will prevent FAST 1.0 from being distributed in the research community and fully utilized.

One way to address the software development deficiency is to use an open source software development and maintenance methodology. Community software development can occur in parallel and leverage a larger community than any one university or research group within that university can allocate to software development. This builds a software library that includes several foundational examples for new research, similar to the out-of-the-box working examples provided with software simulators [4, 12, 34, 50, 68, 81]. This software library will accelerate future research projects because of the available working building blocks. Furthermore, working hardware prototypes will accelerate the transfer of research ideas to industry.

The FAST project did not anticipate the difficulty and increased design time required to manage the FPGA constraints. Specifying and meeting the design timing constraints was more difficult than expected. Developing tools and GUIs to manage the FPGA constraints by incorporating design feedback from the post-PAR information will speed up the prototype development. Furthermore, implicit hardware instantiation by the

Xilinx tools can also lead to many days of frustration. The tools produce so many warnings, creating information overload, making it difficult or impossible to track down automatically instantiated components.

From the beginning, FAST incorporated at least two alternatives for every major function of the PCB and these were controlled by software. One method required some software development to be functional, while the secondary method required little or no software development to be functional. For example, the FPGAs have two different programming interfaces. The PLD can be used to program the FPGAs, but additional infrastructure is required to use this potentially faster FPGA programming method. Xilinx also provides Impact and the Parallel IV cable to program FPGAs using the JTAG interface without any additional user developed infrastructure. Due to time pressure, the easiest method was always used. Having multiple ways to do the same thing provided designer flexibility and reduced prototyping effort.

The FAST PCB also had several headers that could be used for yet-to-be-defined functions. FAST implemented an RS-232 protocol that required a daughter card for the RS-232 buffer. As described in the FAST 2.0 design, incorporating multiple I/O options will make that design even more flexible and easy to use. These I/O options must provide a range of functionality and bandwidth with integrated buffers and physical layer chips. Fortunately, this does not require significantly more integration effort.

The FAST project produced both hardware and software that demonstrated the feasibility of hybrid hardware prototyping platforms. Incredibly, the FAST project produced a single run PCB that was a Stanford design, layout, and route, requiring *no rework*. Normally, the design schematic and specification is defined and professional PCB design houses do the PCB layout and routing. Even then, most PCBs require an additional spin to correct any mistakes and rework. By comparison, the NetFPGA II PCB, a much smaller and less complicated design, provided a schematic and specification to a PCB design house and the first revision required rework [99].

The FAST 4W-NC prototype was developed as a proof of concept that demonstrated that more complex designs could be mapped to the FAST platform. Time was the only

factor that prevented more complex designs like Hydra or TCC from being mapped to the substrate. In the process of creating a simple prototype, all the systems on the FAST PCB were tested and deemed functional. Furthermore, the base software infrastructure was developed for FAST and demonstrated by running programs on FAST 4W-NC. This software foundation would be used in all future prototypes mapped to FAST.

Future systems that require chip design will become increasingly more difficult because of the high cost of chip manufacturing. Thus, developing a flexible hardware prototyping platform will enable research beyond the software simulator, and in particular, software development research. Careful attention to the FPGA interconnect is required because an intelligent interconnect will broaden the prototyping domain, whereas a poor FPGA interconnect will narrow the prototyping domain, in the worst case to a single instance. FAST and other follow-on projects have demonstrated the utility of and demand for reconfigurable prototyping platforms. Future hybrid systems are going to be even better than FAST or any current system from a hardware point of view. However, hybrid systems will require significant software infrastructure to truly exploit their vast potential.

Appendix A

Making FAST

There are several steps that go into making a functioning hybrid prototyping system like FAST. This section provides a list of those steps. This is not an exhaustive list that covers all aspects of the project, but it provides an understanding of the details not discussed in the main body of this dissertation. The following sections detail the process required to build the FAST system. Not all the information can be provided, but the FAST PCB summary README information required by the PCB manufacturer and assembler is provided for reproduction purposes or as templates. The manufacturing and assembly files are located online: http://ogun.stanford.edu/fast/FAST_PCB_Files.zip.

A.1 Building a hybrid prototyping platform

Building a hybrid prototyping platform can be broken into two main steps: all the processes and information required to build the PCB, and all of the processes and information required to make the PCB functional. The steps for the pre-PCB development and post-PCB development are presented in outline form for brevity. The following subsections list the steps, sometimes in no particular order, required to realize the FAST system. Documentation is an overarching requirement for all phases of the system development and is not explicitly included in any given step. Therefore, the documentation process is left out of the steps and any further descriptions.

A.1.2 Pre-PCB development process

There are five main components to making the PCB. The PCB development process requires: define the PCB architecture, select the PCB tools, design the PCB, specify the prototype details, and verify the PCB. The PCB architecture defines the overall design and functionality. The PCB tools are the framework used to realize the design. The PCB design specifies all the parts, the schematic, the layout and routing, and design verification processes. The prototype specification is required to describe how to build and assemble the PCB. Finally, a process to verify that the PCB is functional is also required. The outline of these five categories is as follows:

1. Define the PCB Architecture
    a. Define the function and connectivity.
        i. Example: four processors connected to FPGAs via a common shared bus with a shared memory.
    b. Select the PCB components: source, availability, package, I/O and core voltage, create a Bill of Materials (BOM), additional 10% parts margin, thermal profile parts
2. Select the PCB Tools (Cadence)
    a. Tools selected to design the FAST PCB
        i. Schematic editor: OrCad Capture
        ii. PCB Layout (including footprint generation): OrCad LayoutPlus
        iii. PCB Router: Specctra
        iv. Gerber file tool: GerbTool
    b. Tools that **SHOULD** have been selected to design the FAST PCB
        i. Schematic editor: Concept
        ii. PCB Layout (including footprint generation): Allegro
        iii. PCB Router: Specctra
        iv. Gerber file tool: Allegro/APD
3. Design the PCB
    a. Schematic editor
        i. Use a flat or hierarchical schematic to define component connectivity
        ii. Use net names of 8 or less characters for tools compatibility
    b. PCB Layout
        i. Define footprints: dimensions, SMT, through-hole, sockets, pin spacing, clearance
        ii. Determine component placement: layer placement (top, bottom, sandwich between layers)
        iii. Place components and additional support components: decoupling capacitors, pull-up/down resistors, no connection pins
    c. PCB Router
        i. Define layer routing: Manhattan, Diagonal, combination
            1. Manual routing for critical nets
            2. Automated: routing order, bus routing
        ii. Validate Routing: connected nets, trace spacing, off-grid components, off-grid routes
        iii. Generate Gerber Files and modify them for PCB production
            1. Visually check Gerber files
            2. Remove unused pads
            3. Compare Gerber files to Layout netlist.

4. PCB Prototype Constraints and Specification
    a. Design for Manufacturability (DFM) constraints
    b. Create PCB specifications (README file): PCB material, layer count, layer stack-up, Gerber file descriptions, hole plating, pad details, miscellaneous details, manufacturing standards, parts orientation.
    c. Create PCB assembly specifications: BOM, parts orientation, and post assembly PCB testing (X-ray and integrity)
5. Verify PCB
    a. Pre-PCB delivery process
        i. Test equipment
        ii. Power estimates
        iii. Software
    b. Post-PCB delivery process
        i. Visual inspection: parts placement and orientation, solder inspection
        ii. Buzz-out: resistance measurements and power connectivity
        iii. Power-up PCB: voltage rails and voltage noise
        iv. Check JTAG integrity: Corelis, J-SCAN, Xilinx Impact
        v. Program FPGA/CPLD with simple counter and LED tests
        vi. Perform Corelis interconnectivity test
        vii. Perform Memory tests: L1 SRAM, L2 SRAM, Flash
        viii. Perform Processor tests: R3000, R3010

This outline provides a glimpse into the steps required to produce and do initial testing of a prototype PCB. This outline does not provide all the details, but provides enough detail to reproduce the process. Copious amounts of documentation is collected or generated after going through all five of these steps.

A.1.3 Post-PCB development process

There are five main processes required to make the PCB functional. These required processes include: select morphware tool chain, build Verilog infrastructure, add supplemental hardware, software tool chain, and software infrastructure. The morphware tool chain provides the overall framework for Verilog infrastructure development (morphware). The Verilog infrastructure is broken down into the main modules required for the functioning FAST system. Supplemental hardware was added to the system, after the fact, to extend system functionality. The software tool chain provides the overall framework for the software infrastructure development. Finally, the software

infrastructure is the foundation for software development and tuning. The outline of these five categories is as follows:

1. Select Morphware tool chain
    a. Verilog and FPGA development environment (Xilinx ISE)
        i. Coregen
        ii. Data2Mem
        iii. Impact
    b. JTAG
        i. Corelis
        ii. J-Scan
        iii. ChipScope
    c. Support tools
        i. Sed/awk/perl
2. Create Verilog infrastructure
    a. FAST Verilog and UCF wrappers
    b. Clock distribution and generation
    c. MIPS initialization
    d. Memory testing
        i. BRAM, SRAM, Flash, and Compact Flash
    e. Memory initialization
        i. BRAM, SRAM, Flash, and Compact Flash
    f. Performance counter
    g. Off-board I/O
        i. RCM3200 and RS-232
    h. Timing
        i. R3000 and UCF timing constraints
    i. TLP prototype architecture
        i. FAST 4W-NC
        ii. Future prototypes
            1. Hydra
            2. TCC
3. Build supplement hardware
    a. RS-232
    b. Compact Flash daughter card
    c. Ethernet
4. Select software tool chain
    a. Software development environment (SDE): Compiler/Linker/Debugger
        i. Cygwin + MIPS SDE
        ii. Native SGI SDE
    b. Support tools and scripts
        i. Binary to Verilog converter
        ii. Binary to MEM file converter
5. Software infrastructure
    a. Operating system
        i. Start.s and exit.s
        ii. Porting Linux

b. Drivers
    i. Compact Flash
    ii. RS-232
c. Applications
    i. Stanford Small Benchmark Suite
    ii. All applications

A.2 PCB specifications

The PCB manufacturing and assembly house requires a README file, a collection of Gerber files, parts for the PCB assembly (prototype PCB and thermal profile PCB for BGA parts), and the related BOM. The README file is included as an example. Version numbers and dates for the files were used in order to track changes resulting from DFM revisions to the design files, only one PCB was built and required NO rework.

```
STANFORD FAST BOARD (V1.3) :                    Last Updated: 03/30/05

     Contact:
     --------
          John D. Davis
          Stanford Computer Systems Lab
          Gates CS 4a, Room 456, M/C 9030
          Stanford University
          Stanford, CA 94305-9030

          phn: 650-723-6891
          fax: 650-725-6949

          johnd@stanford.edu

     General information:
     --------------------
          Fabricate to IPC standards
          FR4 Material
          No impedance matching
          ENIG Board Finish
          Red soldermask
          Silkscreen top and bottom
          20 layers, stack up below
          1/2 oz. copper (if possible)
          4 mil traces with 4 mil spacing
                There are about 20 5 mil traces.
          Standard Layer thickness, total board thickness ~0.087"
          4260 components
          31673 vias, 13 drill sizes
           Teardrop pads are ok.
          Minimum drill hole size is 8 mils
                 ALL vias not associated with BGAs use 10 mil vias with 26 mil pad
                Most BGA pads are 24 mils for the 8 mil via
                Some 9 mil vias have 18 mil pads for the 0.8 mm pitch BGAs
          Mount holes are not plated.
           Alignment pins are not plated: MBC_* and JTAG* headers have alignment pins.
           All other holes are plated.
           +/- 0.003 drill hole tolerance.
          Board dimensions are all on the Assembly Top layer Gerber file, *.AST
          Fiducial marks are measured from right to left
          Mount Holes are measured from left to right

          Manufacturing files archive: FAST_Fab.zip

          Stack up:                     Gerber File Name

             TOP                        FAST_DONE.TOP
             GND1                       FAST_DONE.GND
             PWR                        FAST_DONE.PWR
             GND2                       FAST_DONE.GND
```

```
            INNER2                  FAST_DONE.IN2
            INNER3                  FAST_DONE.IN3
            GND3                    FAST_DONE.GND
            INNER4                  FAST_DONE.IN4
            INNER5                  FAST_DONE.IN5
            INNER6                  FAST_DONE.IN6
            INNER7                  FAST_DONE.IN7
            INNER8                  FAST_DONE.IN8
            INNER9                  FAST_DONE.IN9
            INNER10                 FAST_DONE.IN10
            INNER11                 FAST_DONE.IN11
            INNER12                 FAST_DONE.IN12
            GND4                    FAST_DONE.GND
            3V                      FAST_DONE.3v
            GND5                    FAST_DONE.GND
            BOTTOM                  FAST_DONE.TOP

     Manufacturing files archive contents:
     -------------------------------------
         Orcad Layout Plus 10.2 produced, Gerbtool 13.0.1 modified (remove isolated pads)
         Gerber file format: RS274X, m.n.:2.4, Terminator *\r\n, Absolute Coordinate Mode,
         Leading Zero Suppression, 'G' Command Included, ASCII Character Set

            file name                           contents
           ---------------------------------------------------------
        FAST_DONE.TOP               top layer gerber, RS274X
        FAST_DONE.GND               main ground plane gerber, RS274X, NOTE: use for ALL GND* planes
(1-5)
        FAST_DONE.PWR               5.0 V, 2.5 V, and 1.5 V power plane gerber, RS274X
        FAST_DONE.3V                3.3 V power plane gerber, RS274X
        FAST_DONE.IN2               signal layer 2 gerber, RS274X
        FAST_DONE.IN3               signal layer 3 gerber, RS274X
        FAST_DONE.IN4               signal layer 4 gerber, RS274X
        FAST_DONE.IN5               signal layer 5 gerber, RS274X
        FAST_DONE.IN6               signal layer 6 gerber, RS274X
        FAST_DONE.IN7               signal layer 7 gerber, RS274X
        FAST_DONE.IN8               signal layer 8 gerber, RS274X
        FAST_DONE.IN9               signal layer 9 gerber, RS274X
        FAST_DONE.I10               signal layer 10 gerber, RS274X
        FAST_DONE.I11               signal layer 11 gerber, RS274X
        FAST_DONE.I12               signal layer 12 gerber, RS274X
        FAST_DONE.SMT               top soldermask gerber, RS274X
        FAST_DONE.SMB               bottom soldermask gerber, RS274X
        FAST_DONE.SPT               top solderpaste gerber, RS274X
        FAST_DONE.SPB               bottom solderpaste gerber, RS274X
        FAST_DONE.SST               top silkscreen gerber, RS247X
        FAST_DONE.SSB               bottom silkscreen gerber, RS247X
        FAST_DONE.AST               top assembly layer and board dimension gerber, RS274X
        FAST_DONE.ASB               bottom assmebly layer gerber, RS274X
        FAST_DONE.DRD               drill drawing gerber, RS274X
        FAST_DONE.FAB               fabrication drawing gerber(no info. in this file), RS274X
        FAST_DONE.BOT               bottom layer gerber, RS274X
        FAST_DONE.IPC               IPC-D-356 format netlist
         FAST_DONE.DTS              Drill tape summary report
        FAST_FINAL.MNL              Orcad Layout Netlist
        README_1.3_FAST.txt             This file.
        FAST-FinalPartsChecklist.xls    Parts shipping list, excel file
         FAST-ThermalProfileChecklist.xls Parts for the thermal profile PCB.
        FAST_FINAL.BOM                  Bill of Materials, ASCII
        thruhole.tap                drill tape file, ASCII
        FAST_DONE.APT               aperature file, ASCII
         COMPPROP.txt               alternate  format  of  the  pick  and  place  file  with  X/Y
coordinates
         PICKNPLC.txt               Orcad layout pick and place file

    Assembly Notes:
    ---------------
    Some components listed in the BOM are actually two separate receptacle
    components or a header. Please look at the FAST-FinalPartsChecklist.xls
    for the part number mappings.

Here is a brief description:

    BOM Item #          Description
        2               clock header, part number A463-ND
        35              two 2mm dual row headers,17-pins per row
        40              two dual row headers with 4-pins per row
        41              two different single row receptacles, one 5-pin and one 6-pin

    All TPA_* and TEST POINTS are on reels, part number 5015KTR-ND
    They  are  group  in  115  "parts"  for  placement  conveniences,  but  are  2452  individual  parts.
```

Appendix B

FAST software stack

The Verilog modules for the FAST system are provided in this section with the intention of copying and pasting the text from the PDF file to an appropriate text editor. The font size was selected to conserve space while at the same time providing all the UCF files and Verilog modules. The Verilog modules are broken up into two sections: FAST VAL modules and the modules required to instantiate the FAST 4W-NC prototype. The FAST software is packaged into one archive file with several subdirectories. This archive can be found: http://ogun.stanford.edu/fast/FAST_SW_Archive.zip.

B.1 FAST VAL

The FAST VAL is made up of two different collections of files: User Constraint Files (UCF) and Verilog modules. This section presents the generic UCF files. Only one UCF file is provided for the L1C and CP2 FPGAs. There are minor naming differences between the set of UCF files and some of the mappings to other components. The Verilog wrapper modules and the FAST VAL Verilog modules are located online in the software repository directory FAST_VAL of the complete FAST software archive.

B.2 FAST 4W-NC

The FAST CMP 4W-NC is a decoupled 4-way CMP with private L1 data and instruction caches and private L2 memory. Figure B.1 provides a high-level overview of the FAST CMP 4W-NC architecture. The FAST CMP 4W-NC used 4 KB BRAM L1 caches or 256 KB SRAM L1 caches. In either L1 cache size case, the caches were direct-mapped and used a write-through policy. The private L2 memories are 32 KB (not including parity bits) each with a maximum size of 72 KB, if all BRAM blocks are used. The level of integration in the FAST CMP 4W-NC demonstrates that a variety of other more complex architectures can be mapped to FAST.

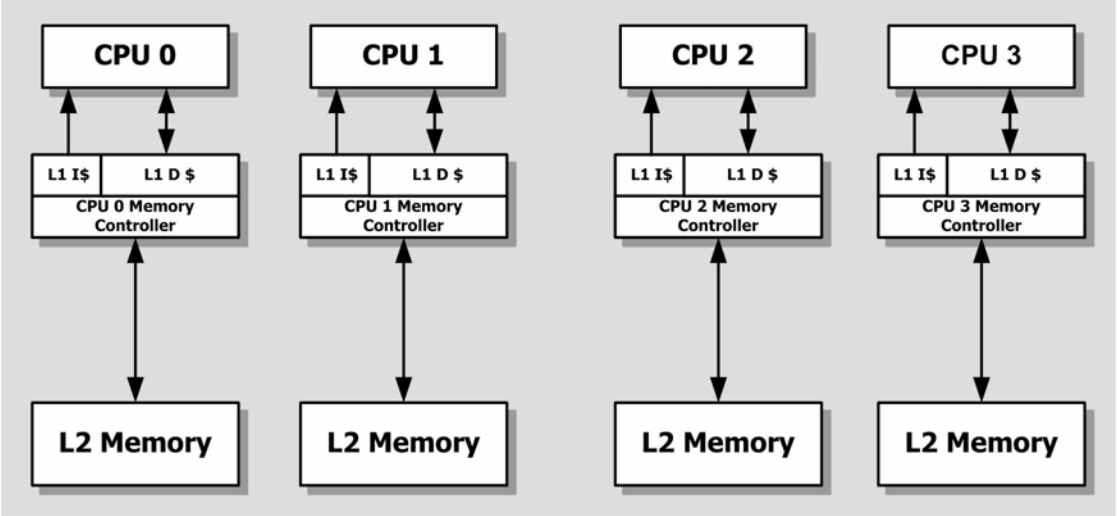


Figure B.1. 4-way decoupled FAST CMP 4W-NC.

The Verilog modules that instantiate the FAST 4W-NC proof of concept and related helper Verilog modules that purge the L1 SRAM caches are located online in the software repository directories FAST_SW\Verilog\RWC and FAST_SW\Verilog\PT.

B.3 FAST software

The FAST System has three main software components that make up the FAST Software Toolbox: short Assembly language diagnostics, system software and applications, and support software in the form of scripts. Once the FAST PCB is programmed with the morphware, the prototype system is ready to run diagnostic tests or applications. This section provides the software used to provide the proof of concept for the FAST 4W-NC prototype. The small diagnostics, operating system functions, small applications, and Stanford Small Benchmarks are located online in the software repository directory FAST_SW\SSB.

Appendix C

FAST 2.0 consideration details

There are many details that need to be specified for a new hardware prototyping platform. The details provided below outline the high-level considerations that will enable more flexibility, thereby extending the systems capabilities. Leveraging existing technology has the added side benefit of reducing development time. Moving forward, technology enables more feature and presents greater design challenges and considerations.

C.1 Leveraging hardware

The major advance in FPGAs is the device density. High device density combined with the 6-input LUTs, improves the speed and density of mapped logic by reducing the number of logic levels. The 6-input LUTs also reduce the design routing pressure. These latest FPGAs have also added high-speed serial links. The high-speed serial links increase the chip bandwidth and compensate for the limited number of I/O pins.

Unfortunately, FPGAs continue to struggle with resource allocation. In particular, on-chip memory (BRAMs), special purpose embedded units (embedded DSP and hard processor cores), and configurable logic continue to battle for on-chip real estate, with the most general or flexible configuration determining the FPGA building block mix. As previously mentioned, the quantity of FPGA on-chip memory has stagnated over the last few generations, losing on-chip real estate, but maintaining the same on-chip memory capacity. The memory density is an allocation issue and not a technological issue. Thus, FAST 2.0 must provision additional memory to compensate for the lack of on-chip memory in the FPGAs, as did FAST 1.0.

Like FPGAs, memory continues its growth in both device speed and device density. As a result of the memory improvements, both static random access memories (SRAMs) and dynamic random access memories (DRAMs) can interface to FPGAs and provide enough bandwidth for the processor and low access latency for memory requests. SRAMs are fast enough to fulfill memory requests in a single clock cycle and provide more capacity

than any available level one cache or level two cache, as demonstrated by FAST 1.0. Furthermore, DRAMs can be added for even greater memory capacity beyond the SRAMs, as demonstrated by BEE2 that integrated DRAM and not SRAM [21]. DRAM access time, though slower than SRAM, is certainly sufficient for prototyping level two or higher levels in the memory subsystem. New DRAM interfaces have also significantly reduced the interface pin requirements. Dual channel fully-buffered DIMMs (FB-DIMMs) can provide similar bandwidth to that of a single channel DDR interface [59]. More importantly, the DDR interface requires about 250 pins, while the FB-DIMM interface requires only about 110 pins [59]. Using FB-DIMM dramatically reduces the pin pressure on the FPGA, allowing for more FB-DIMM channels or other interfaces. Unfortunately, there is no free lunch and the FB-DIMMs require higher integration because an additional Advanced Memory Bus (AMB) chip is required to communicate between the memory controller (FPGA) and the FB-DIMM modules [59]. This adds to the design complexity, but the package pin savings more than compensates for the additional chip integration and reduces the pin pressure on the FPGAs.

There are several I/O devices that can be incorporated into a PCB design that will improve system functionality and flexibility. Technology advances have yielded high-speed serial links that provide several gigabits per second. Coupling these high-speed serial links with standard network protocols like Ethernet or Infiniband accelerates system development by providing pre-defined interconnects. Likewise, low-speed alternatives should not be ignored. Using a low-speed RS-232 or other low-speed communication interface can provide a system or FPGA interface or diagnostic back door. It is important to plan for communication because RS-232, Ethernet, and other communication methods require additional components integrated on the PCB.

There are also some things that cannot be planned at project inception. Thus, to compensate for this lack of foresight, additional flexibility must be designed into the prototyping platform. For FAST 2.0, this takes the form of additional interfaces for communication and hardware modules. Daughter cards are the primary hardware module interface. This requires a standard connector for the daughter cards on the PCB and distributed power and ground connections. By including a daughter card interface,

the FAST 2.0 PCB can expand its capabilities beyond the initial design. Thus, digital interfaces can be easily added using this standard interface.

C.2 Leveraging software

The new VAL will depend on the processor used in FAST 2.0 as the basis for the prototyping system. VAL development is required for any new PCB design because the processor and memory interfaces will be different than the FAST 1.0 interfaces. Current and future FPGA generations have embedded processor cores that run much faster than the MIPS R3000's used with FAST 1.0. Software-defined processor cores (soft cores) are also an option, as they operate at or above the speed of the MIPS R3000's as well. Using soft cores has the additional benefit of improving system flexibility by adding more extensive processor design capabilities to FAST 2.0. If the VAL interface is maintained, then previous prototypes that use the VAL interface can be used with the next PCB design and new VAL. Maintaining the same VAL interface leverages the existing software. In this case, the existing software consists of previous architecture prototypes or just prototype building blocks.

Another crucial software component is the operating system (OS). A flexible, open source OS with cache coherent multiprocessor support can supply the building blocks for all other prototype systems enabling full-system prototyping. Providing such an OS along with the board will enable full-system prototyping. Given this baseline functionality, other systems can add or subtract OS functionality as needed to suit their prototype system. Using a similar open source development methodology, the FAST OS can be created as a branch off of the existing Linux software tree with various modules that can be added or subtracted. This provides a familiar software repository and community to build and support the FAST OS version of Linux.

Selecting a processor that runs an OS out of the box also reduces the software development effort. Having an available OS to run also implies a mature software toolset: compilers, debuggers, etc. Both software-defined cores and hard cores can run ported versions of Linux. Furthermore, the LEON3 software defined core is Ultra SPARC compliant and can run older versions of Solaris [88].

Application profiling also facilitates rapid prototyping. Application profiling is possible without a full operating system and related OS overhead. The base OS functionality can be provided in the form of statically linked libraries included within the application binary, thereby focusing on the potential of the new prototype architecture without the need for extensive OS development. Runtime libraries statically linked, for simplicity, into the application provide the essential OS functions, enabling FAST 2.0's rapid application profiling. These standalone libraries are essential to rapid application and system prototyping because the libraries can easily be modified and integrated without any OS overheads, as well as, minimizing the FAST 2.0 system and system software learning curves. This also enables API and driver development in parallel with OS development and integration. Using the standalone libraries enables parallel software and system development, which also accelerates the rapid prototyping process.

Likewise, having an available suite of pre-compiled benchmarks for the base system also accelerates the prototyping process. A broad selection of applications also enables a wide array of research, from embedded systems to high performance computing. The benchmark source code combined with a software development environment (SDE) is all that is needed to target the new system. The SDE also enables OS, API, and driver development for the new prototyping environment. Furthermore, using open source development methods, the applications, OS, APIs and drivers can be leveraged by the entire research and development community.

C.3 Implementing new architectures

The FPGA interconnect is the most critical element required for prototyping platforms from different domains. Mapping various architectures from different domains like networking, DSP, embedded processors, and scientific computing is only possible by mapping the FPGA interconnect in an efficient and flexible manner. PCB prototyping substrates use reconfigurable FPGAs and fixed component routing or interconnect. Because FPGAs are reconfigurable, it is possible to work around the fixed routing and pin assignments of the FPGAs, but potentially using cumbersome interfaces. Unfortunately, the PCB substrate forces interconnect choices, which if done poorly, only inhibits mapping prototype architectures to the system. If the communication resources

for an architecture exist, an architecture is much easier to map to the prototyping substrate compared to an architecture that must work around insufficient communication resources. The dedicated high-speed serial links on FPGAs reduce some of the design pressure because these links reduce the pin pressure required for high bandwidth FPGA-to-FPGA, and off-PCB communication.

One system cannot provide the substrate for all computer systems and computer architecture research. Thus, a standard expansion mechanism is required to enable new functionality. Using a standard digital interface for daughter cards, FAST 2.0 will be able to expand both its compute and communication capabilities. The daughter cards extend the capabilities of FAST 2.0 by providing additional configurable logic or new communication interfaces. The daughter card interface provides I/O pins and power and ground pins to provide a complete module interface. This complete interface eliminates the need for external power supplies or other infrastructure. Dedicated high-speed serial links in the daughter card interface provide high bandwidth communication with a limited number of pins. Parallel to serial data conversion may be required, but this does not inhibit design flexibility.

C.4 Making FAST work

Creating a PCB with ***no rework*** is an amazing task that was validated using JTAG tools from Corelis [26]. By using the Corelis tools, all of the FAST system problems could be quickly isolated to software issues and not a combination of software and hardware issues. Furthermore, no additional workarounds were required to make the PCB fully functional, saving many hours of PCB diagnostics and resulting rework. Validating the PCB was one small step in making the FAST system work. This was a small step because the PCB did not have any hardware problems that impacted functionality.

Making the FAST system work was more involved than designing and manufacturing a PCB and writing some software. A majority of the complexity involved mating the MIPS R3000s to FPGAs. This combination enforced hard timing constraints that had to be met in order for the system to function. Defining timing constraints is part of all Verilog designs that are mapped to FPGAs. FAST required in depth knowledge of the timing

constraints and very low-level logic placement was required to meet timing. For example, the MIPS R3000 requires four externally generated clock inputs for its dual-phase clocking scheme. The MIPS R3000 can operate in a range of clock frequencies, from 16 MHz to 33 MHz. Two of the clock signals lag the initial system clock by 6 ns, while the other clock signal is almost 180 degrees phase shifted. In order to produce these two different phase shifts, a fixed buffer was used along with a delay lock loop (DLL). The DLL produces the correct clock frequency and shifts the clock by 180 degrees. Fixed placement and routed buffers provide the additional delay required to meet the MIPS R3000 timing specification [54]. Using a placed and routed buffer achieves the 6 ns delay regardless of input clock frequency. Using the DLL plus a buffer achieves the frequency dependent approximately half clock cycle delay. This example illustrates how difficult arbitrary delay generation is using an FPGA. Later generations of FPGAs, Virtex II and beyond, enable more flexibility and more clock resources, but these resources can be quickly exhausted.

The FAST 4W-NC model, discussed in Section 4.2.2, used less than 2% of the logic resources on the FPGA. When the instruction and data cache were instantiated on the FPGA, almost 100% of the BRAMs were allocated for the 4 KB data and tag arrays. Using the SRAM chips for the instruction and data caches freed all the BRAM resources, keeping the logic utilization constant. Even at this low logic utilization level, some signals were unable to meet the timing constraints that were applied to the bus. In particular, the 32-bit data bus required several placement iterations to meet timing. Signals that met timing were fixed in the UCF file and signals that did not meet timing were rerouted until those signals did meet timing. Each routing pass produced a few more signals that met the timing constraint and could be fixed in the UCF file. Once the signals were fixed, the subsequent placement and routing completed very quickly because of the reduced complexity.

Xilinx allows the user to specify the placement and routing effort. By increasing this effort to high, more signals could be routed, but this required much more time for this phase of the FPGA bit file generation, in some cases several hours. Unfortunately, the entire bus would not meet the timing constraints, forcing the iterative fixed routing process using high effort. It was much faster to use the standard placement and routing

effort when iteratively fixing the placement and routing for a particular bus, even though more iteration were required to fix the bus placement and routing because each iteration was much faster.

When designing the SRAM interface, the contents of the 32-bit data bus and/or address bus needed to be captured by the L1C FPGA and sent to the appropriate cache, because the dual-phase clocking scheme provides an instruction and data request each clock cycle. The latches used to capture and hold the data and/or address were initially automatically placed by the Xilinx tools. The Xilinx tools placed the latches close to the pins that connected the FPGA to the SRAM and not the pins close to the MIPS R3000. Thus, the data and address were forced to traverse the entire FPGA chip before they were captured. Using this placement, the entire bus could not be routed to meet the timing constraints. However, by fixing the placement of these latches in the I/O pads associated with the pins connected to the R3000, timing was met and the correct data and/or address could be provided to the SRAM chips. Moving the latches is a relatively simple concept, but required several hours of combing through the documentation because the latches in the I/O pads are specified differently than the latches in the normal combinational logic block (CLB).

The FAST system also implemented a bi-directional RS-232 module that can be used with any of the FPGAs on the FAST PCB. This enabled a host PC to download performance counter results. The initial RS-232 module was designed and tested in isolation. The RS-232 module was fully functional and then integrated into the FAST 4W-NC prototype. Integrating the RS-232 module was very easy, but it was non-functional. Some of the counters were not transmitted to the host correctly. By comparing the previous results from experimental runs and displaying the current counter values using the J-SCAN GUI with what was being received by the host PC verified that the RS-232 module was corrupting the counter values. The non-functional RS-232 module was surprising given a 9600-baud rate for the data transmission. After extensive investigation, the difference between the working isolated RS-232 module and the non-working integrated RS-232 module was the use of a global clock buffer. In the isolated RS-232 module instantiation, the Xilinx tools implicitly added a global clock buffer to the RS-232 9600 Hz clock. In the integrated module, the clocking resources were limited and

the Xilinx tools did not implicitly promote the 9600 Hz clock to a global clock buffer, but kept this clock local. By explicitly defining the 9600 Hz clock as a global clock buffer, the integrated RS-232 module functioned correctly. This minor difference required several weeks of investigation to pin down. The 9600 Hz clock was also not an initial suspect in the investigation because of its very low frequency. However, even very slow clocks can cause problems, as proven by the integrated RS-232 module.

As the RS-232 module has demonstrated, the level of detail required to make various parts functional in the FAST 4W-NC design was underestimated because of the relative operating frequency difference between the MIPS R3000, submodules like RS-232, and Virtex I FPGAs. In order to create working interfaces, the timing constraints of the FPGA and placement of logic, down to individual look-up tables (LUTs), had to be managed for the FAST VAL and FAST 4W-NC prototype. Interfacing to the MIPS R3000 presented the greatest challenge, but this interface is encapsulated in the FAST VAL and requires no further modifications. The memory system interface is also defined, reducing the effort for future prototype systems.

Expanding FAST's capabilities will require a similar level of attention to detail with respect to timing constraints and logic placement. These factors become more critical as the prototype systems use more and more of the FPGA resources. Fixed placement and routing becomes even more critical because it will enforce the timing constraints and reduce morphware build effort and time. Imagining a mature FAST system, it is necessary to provide a better interface to deal with timing constraints because it is very easy to shoot yourself in the foot with timing problems, as demonstrated by the RS-232 integration problem. Removing the additional timing constraint problems from FAST or future system will make them more like software simulators. Just providing a software library is not sufficient for building new prototype systems. The Xilinx tools are slowly improving the constraint management, but the improvement is limited to configurable Xilinx modules that can be incorporated into the design. Easy constraint management has not been extended to user-specified Verilog modules. Creating such a constraint management framework would greatly reduce the prototype development effort.

C.5 Scalable hardware prototyping platforms

Creating a flexible and scalable hardware prototyping platform is very challenging. Exploring FAST 1.0's scalability was outside of the scope of this work, but was a PCB feature. FAST 2.0 greatly improves on system scalability by enabling FPGA-to-FPGA and PCB-to-PCB communication or anything in between. In order to scale FAST 2.0, the front panel connectivity would enable FPGA-to-FPGA connectivity, while the back panel would enable PCB-to-PCB connectivity for a rack-mounted system. FAST 1.0 also demonstrated that having back-up hardware mechanisms added to the system flexibility and reduced system software development. Implementing an intelligent interconnect also broadens system applicability. However, these design practices alone do not make a scalable system.

Power, form factor, and system control are very important design considerations for scalable prototyping systems. The increased device density of FPGAs and DRAMs has the added down side of increased power. Starting with Virtex-II Pro, Xilinx FPGAs required active cooling using small fans. Likewise, multiple DRAM modules increase the power dissipation dramatically. The combined power dissipation poses the same power distribution and cooling challenges that server racks have. Creating a high-density prototyping platform can require the same unsustainable power density of blade or rack servers.

FAST 2.0, or systems like it, can be produced with PCB dimensions of 14" X 17", which will easily fit in a 1U server form factor. Heat dissipation is the major challenge for this form factor. FAST 2.0 can easily dissipate 350 W, at 150 W for the FPGAs and 200 W for DRAMs (four per FPGA). This is a similar power budget for a 1U rack server. Exotic power dissipation measures like heat pipes and several fans would enable proper cooling. Using a 2U enclosure with two FAST 2.0 PCBs would relax the heat dissipation constraints by allowing cheaper active cooling solutions. The 2U enclosure would contain a FAST 2.0 board mounted on the bottom with the other FAST 2.0 PCB mounted upside down on the top of the enclosure. The two FAST 2.0 boards would be staggered to prevent the integrated fan and heat sink from colliding. Using these

standard enclosures would enable inexpensive infrastructure for scalable prototyping systems.

Finally, as systems like FAST 2.0 scale, they require control systems. Inherently, FAST 2.0 has a service (S) FPGA that can act like a service processor used in larger servers. The server processor monitors system behavior, can restart the system, and reconfigure the system. The S FPGA serves the same process as traditional service processors, but the prototyping system also requires a system hierarchy that scales. A well-defined S FPGA hierarchy enables full system control required for initializing, synchronizing, and executing applications on large systems.

Scaling was a design parameter for FAST 1.0, but scaling presents more challenges for FAST 2.0. FAST 1.0 dissipated 20 W while FAST 2.0 can easily dissipate 20 times that amount of power. Technology also enables FAST 2.0 scalability, with greater design complexity as a result. Likewise, FAST 2.0 leverages more hardware and software by using commodity components and intellectual property. Every next generation hardware prototyping platform will have greater capabilities and will be easier to use.

Appendix D

FAST 4W-NC data

Initial results using FAST were difficult to gather because the system was unreliable. Transmitting the data clock from the processor tile to the RWC FPGA solved this reliability problem. The SRAM instruction cache for the FAST CMP 4W-NC was the last fully functional component. The instruction cache required both iterative routing and the addition of multiplexers with the correct conditions for the L1 SRAMs to be fully functional. There were also some additional steps required to make the system complete. The next section describes the data collection process. This is followed by a stall cycle discussion using Bubblesort as an example, keeping in mind that Intmm and Quick also experience the same latencies. We will conclude this section with the Chapter 5 data in tabular form.

D.1 Collecting the data

Consistent data collection from the FAST 4W-NC prototype was achieved with the addition of an L1 instruction cache scrubber. This Verilog module uses the R3000 to issue instructions that invalidate the entries. Running this module in between data collection runs guarantees that the instruction cache has no residual data from a previous run. Unlike DRAM modules, SRAMs can retain data for very long periods of time without applied power because the charge is trapped between the gates of the inverters and leakage is very minimal. The same procedure can be applied to the data cache. Data cache scrubbing is not required because the benchmarks are unoptimized; all data loads have a preceding store to the data cache.

Scrubbing the caches enables data collection without power cycling the FAST PCB, which turns out to be ineffective in clearing the SRAM memory. For all the runs below, the L2 memory can be programmed and then the processor tile configuration can be changed to perform latency and/or cache studies. In between each processor tile configuration update, the cache scrubbing Verilog modules are loaded. The other side benefit of using the processor tile to scrub the L1 caches is that the processor tile configuration files are smaller and load faster than the larger Virtex-II configuration files.

D.2 Explaining the data

Running without an instruction cache, the `div` instruction in bubble is responsible for 4250 stalls. The processor requires 35 cycles to pass after a `div` instruction before it can execute an mflo. In bubble, the div is part of a sequence `div-bnez-nop-mflo`. Without an instruction cache, each instruction `bnez-nop-mflo` takes 6 cycles to come in from the L2 cache (1 normal cycle plus five stall cycles), for a total of 18 cycles by the time the processor sees the `mflo` instruction [54]. It thus must stall 17 cycles for all 250 iterations of the loop because there is no instruction cache.

When an instruction cache is present, the `div` instruction causes a delay of 17 cycles for the first time through the loop, because each instruction has a cold miss, and must be brought in from the L2 memory. In each of the succeeding 249 loop iterations, however, the three instructions `bnez-nop-mflo` only take one cycle each to bring in from the L1, leaving 32 cycles to stall. This results in the total stall cycles of (17 + (249 x 32)) or 7985 cycles. For the benchmarks that use this instruction sequence, the number of stall cycles is dominated by `div-bnez-nop-mflo` sequence and not cold, conflict, or capacity misses. Thus, increasing the L2 latency has a reduced impact compared to programs that only experience cache misses and not ISA related stall cycles.

To further confirm this, we changed the sequence from `div-bnez-nop-mflo` to `div-nop-nop-nop-mflo`, and the number of stalls increased to (16 + 249 x 31), or 7735. Figure D.1 shows the counters reporting the program statistics for Bubblesort (Bubble):

counter1:	960,577	Instruction Cycles
counter2:	9,070	Stall Cycles
counter3:	219,630	L1 Data Reads
counter4:	209	Instruction Misses
counter5:	32,888	L1 Data Writes
counter6:	218	L2 Reads
counter7:	32,919	L2 Writes

Figure D.1. Performance counters for Bubble sort with a 250 element array.

Table D.1 further breaks down the program behavior and tabulates the stall cycles.

Table D.1 Bubblesort stall cycle break down.

	Instruction (Instr) Cycles	Stall Cycles	L1 D Read	L1 D Write	L1 I Miss	L2 Read	L2 Write
NC* Inst.	4	40	0	0	0	1	0
Cold Misses	209	1,045	0	0	209	209	0
Inst. Hits	960,360	0	0	0	0	0	0
Load Hits	0	0	219,630	0	0	0	0
Stores	0	0	0	32,888	0	0	32,919
MFLO Inst.	0	7,985	0	0	0	0	0
Unknown*	4	0	0	0	0	1	0
TOTAL	960,577	9,070	219,630	32,888	209	218	32,919

*Non-cacheable (NC) instructions are used to jump from the reset memory address into the cacheable address space. There are also four instructions and one L2 read that was not accounted for in this tally, but do not warrant further investigation because the consistency across all benchmarks.

D.3 FAST 4W-NC Stanford Small Integer Benchmark data

Table D.2. L1 Cache study investigating program performance behavior without a data and/or instruction cache.

	No I Cache			No I Cache			No I Cache			No I Cache		
	L1 Data Cache		L1 Instr.Cache		L1 Cache		Total	Total	Dyn Intsr.	Compiled	L2 Memory	
Applications	Reads	Writes	Reads	Writes	D Miss	I Miss	Cycles	Stalls	Cycles	Instr.	L2 Reads	L2 Writes
Bubble	219,630	32,888	960,577	960,569	0	960,569	5,767,716	4,807,139	960,577	301	218	32,887
Intmm	12,433	3,252	53,271	53,263	0	53,263	326,758	273,487	53,271	355	336	3,251
Permute	807,149	518,930	3,495,775	3,495,767	0	3,495,767	20,974,654	17,478,879	3,495,775	254	208	518,929
Queens	662,753	185,809	2,230,110	2,230,102	0	2,230,102	13,380,664	11,150,554	2,230,110	358	311	185,808
Quick	28,375	7,182	105,020	105,012	0	105,012	634,374	529,354	105,020	376	363	7,181
Towers	725,319	450,769	3,372,684	3,372,676	3	3,372,676	3,376,459	3,775	3,372,684	468	408	450,765

	No D Cache			No D Cache			No D Cache			No D Cache		
	L1 Data Cache		L1 Instr.Cache		L1 Cache		Total	Total	Dyn Intsr.	Compiled	L2 Memory	
Applications	Reads	Writes	Reads	Writes	D Miss	I Miss	Cycles	Stalls	Cycles	Instr.	L2 Reads	L2 Writes
Bubble	219,630	252,518	960,577	209	219,630	209	2,067,801	1,107,224	960,577	301	218	32,887
Intmm	12,433	15,685	53,271	327	12,433	327	130,298	77,027	53,271	355	336	3,251
Permute	807,149	1,326,079	3,495,775	199	807,149	199	7,532,559	4,036,784	3,495,775	254	208	518,929
Queens	662,753	848,562	2,230,110	302	662,753	302	5,545,429	3,315,319	2,230,110	358	311	185,808
Quick	28,375	35,557	105,020	354	28,375	354	256,694	151,674	105,020	376	363	7,181
Towers	725,319	1,176,085	3,372,684	408	725,319	408	7,001,363	3,628,679	3,372,684	468	408	442,569

	No I or D Cache			No I or D Cache			No I or D Cache			No I or D Cache		
	L1 Data Cache		L1 Instr.Cache		L1 Cache		Total	Total	Dyn Intsr.	Compiled	L2 Memory	
Applications	Reads	Writes	Reads	Writes	D Miss	I Miss	Cycles	Stalls	Cycles	Instr.	L2 Reads	L2 Writes
Bubble	219,630	252,518	960,577	960,569	219,630	960,569	6,865,866	5,905,289	960,577	301	218	32,887
Intmm	12,433	15,685	53,271	53,263	12,433	53,263	388,923	335,652	53,271	355	336	3,251
Permute	807,149	1,326,079	3,495,775	3,495,767	807,149	3,495,767	25,010,399	21,514,624	3,495,775	254	208	518,929
Queens	662,753	848,562	2,230,110	2,230,102	662,753	2,230,102	16,694,429	14,464,319	2,230,110	358	311	185,808
Quick	28,375	35,557	105,020	105,012	28,375	105,012	776,246	671,226	105,020	376	363	7,181
Towers	725,319	1,176,085	3,372,684	3,372,676	725,319	3,372,676	23,862,703	20,490,019	3,372,684	468	408	442,569

There are two experimental series that were performed using the FAST 4W-NC prototype. These experiments provide a proof of concept that demonstrates that more complex non-coherent ***and*** coherent multiprocessor systems can be prototyped using the FAST system. Table D.2 provides all of the data that supplements Table 5.1. In these tables, the

instruction (I) cache, data (D) cache or both L1 caches are turned off and the program still executes correctly with a consistent number of dynamic instructions.

Similarly, Table D.3 presents the L2 memory latency study results. Most of the redundant data was removed and is provided in Table 5.1. Table D.3 is the tabular form of Figures 5.10 and 5.11 and is included for completeness.

Table D.3. L2 memory latency results spanning an L2 latency of 5 cycles up to 257 cycles.

	5 cycle L2 Latency			9 cycle L2 Latency			17 cycle L2 Latency		
Applications	Total Cycles	Total Stalls	Dyn Intsr. Cycles	Total Cycles	Total Stalls	Dyn Intsr. Cycles	Total Cycles	Total Stalls	Dyn Intsr. Cycles
Bubble	969,647	9,070	960,577	970,507	9,930	960,577	972,238	11,661	960,577
Intmm	68,133	14,862	53,271	69,458	16,187	53,271	72,130	18,859	53,271
Permute	3,496,814	1,039	3,495,775	3,497,642	1,867	3,495,775	3,499,298	3,523	3,495,775
Queens	2,231,664	1,554	2,230,110	2,232,904	2,794	2,230,110	2,235,384	5,274	2,230,110
Quick	114,819	9,799	105,020	116,255	11,235	105,020	119,146	14,126	105,020
Towers	3,374,783	2,099	3,372,684	3,376,459	3,775	3,372,684	3,379,811	7,127	3,372,684

	33 cycle L2 Latency			65 cycle L2 Latency			257 cycle L2 Latency		
Applications	Total Cycles	Total Stalls	Dyn Intsr. Cycles	Total Cycles	Total Stalls	Dyn Intsr. Cycles	Total Cycles	Total Stalls	Dyn Intsr. Cycles
Bubble	975,710	15,133	960,577	982,654	22,077	960,577	1,024,318	63,741	960,577
Intmm	77,490	24,219	53,271	88,210	34,939	53,271	152,530	99,259	53,271
Permute	3,502,610	6,835	3,495,775	3,509,234	13,459	3,495,775	3,548,978	53,203	3,495,775
Queens	2,240,344	10,234	2,230,110	2,250,264	20,154	2,230,110	2,309,784	79,674	2,230,110
Quick	124,938	19,918	105,020	136,522	31,502	105,020	206,026	101,006	105,020
Towers	3,386,515	13,831	3,372,684	3,399,923	27,239	3,372,684	3,480,371	107,687	3,372,684

References

[1] "*Workshop on Architecture Research using FPGA Platforms.*" 2006; Available from: http://www.cag.csail.mit.edu/warfp2006.

[2] Advanced Micro Devices Inc.; "*Am29LV652D: 128 Megabit (16 M x 8-Bit) CMOS 3.0 Volt-only Uniform Sector Flash Memory with VersatileI/O Control.*" 2003; Available from: http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24961.pdf

[3] Advanced Micro Devices Inc.; "*AMD 3-Year Technology Outlook.*" 2006; Available from: http://www.amdcompare.com/techoutlook/.

[4] Advanced Micro Devices Inc.; "*SimNow.*" 2006; Available from: http://developer.amd.com/simnow.aspx.

[5] Agarwal, A., et al., "*The MIT Alewife machine: architecture and performance*," in *Proceedings of the 22nd annual international symposium on Computer architecture*. 1995, ACM Press: S. Margherita Ligure, Italy.

[6] Altera Corporation; "*Altera Development Kits.*" 2006; Available from: http://www.altera.com/products/devkits/kit-dev_platforms.jsp.

[7] Altera Corporation; "*Nios II Processor Reference Handbook* " 2006; Available from: http://www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf.

[8] Altera Corporation; "*Stratix II GX Device Handbook.*" 2006; Available from: http://www.altera.com/literature/hb/stx2gx/stxiigx_handbook.pdf.

[9] André, L., et al., "*RPM: A Rapid Prototyping Engine for Multiprocessor Systems.*" Computer, 1995. **28**(2): p. 26-34.

[10] Artesyn Technologies; "*Processor Blades and Processor subsystems.*" 2006; Available from: http://www.artesyncp.com/products/index.html.

[11] Arvind, et al.; "*RAMP: Research Accelerator for Multiple Processors - A Community Vision for a Shared Experimental Parallel HW/SW Platform.*" 2005; Available from: http://ramp.eecs.berkeley.edu/publications.php.

[12] Austin, T., E. Larson, and D. Ernst, "*SimpleScalar: An Infrastructure for Computer System Modeling.*" Computer, 2002. **35**(2): p. 59-67.

[13] Blandy, J., et al.; "*GDB: The GNU Project Debugger.*" 2006; Available from: http://www.gnu.org/software/gdb/gdb.html.

[14] Brooks, D., V. Tiwari, and M. Martonosi, "*Wattch: a framework for architectural-level power analysis and optimizations*," in *Proceedings of the 27th annual international symposium on Computer architecture*. 2000, ACM Press: Vancouver, British Columbia, Canada.

[15] Bunce, P.; "*PMON5 (Including SerialICE).*" 2002; Available from: http://www.linux-mips.org/wiki/PMON.

[16] Cadence, "*Allegro System Interconnect Design Platform.*" 2005.

[17] Cadence; "*INCISIVE PALLADIUM family with INCISIVE XE software.*" 2005; Available from: http://www.cadence.com/products/functional_ver/palladiumII/index.aspx.

[18] Cadence; "*INCISIVEXTREME series of accelerators/emulators.*" 2006; Available from: http://www.cadence.com/products/functional_ver/acel_emul/xtreme.aspx.

[19] Carlstrom, B.D., et al., "*The Atomos transactional programming language*," in *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*. 2006, ACM Press: Ottawa, Ontario, Canada.

[20] Chang, C., et al., "*Implementation of BEE: a real-time large-scale hardware emulation engine*," in *Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*. 2003, ACM Press: Monterey, California, USA.

[21] Chang, C., J. Wawrzynek, and R.W. Brodersen, "*BEE2: A High-End Reconfigurable Computing System."* IEEE Des. Test, 2005. **22**(2): p. 114-125.

[22] Chen, M.K. and K. Olukotun, "*Exploiting Method-Level Parallelism in Single-Threaded Java Programs*," in *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*. 1998, IEEE Computer Society.

[23] Chiou, D., et al., "*FPGA-based Fast, Cycle-Accurate, Full-System Simulators*," in *Proceedings of the Workshop on Architecture Research using FPGA Platforms, held at HPCA-12*. 2006: Austin, Texas.

[24] Cisco Systems, et al.; "*Infiniband Trade Assoication.*" 2006; Available from: http://www.infinibandta.org/home.

[25] comp.arch.fpga, "*News FPGA Users Group."* 2005.

[26] Corelis Corporation; "*ScanPlus Boundary-Scan (JTAG) Controllers.*" 2005; Available from: http://www.corelis.com/products/JTAG_Controllers.htm.

[27] Culler, D.E., A. Gupta, and J.P. Singh, "*Parallel Computer Architecture: A Hardware/Software Approach*. 1997: Morgan Kaufmann Publishers Inc.

[28] Davis, J.D., C. Fu, and J. Laudon, "*The RASE (Rapid, Accurate Simulation Environment) for chip multiprocessors."* SIGARCH Comput. Archit. News, 2005. **33**(4): p. 14-23.

[29] Davis, J.D., L. Hammond, and K. Olukotun, "*A Flexible Architecture for Simulation and Testing (FAST) Multiprocessor Systems*," in *Proceedings of the Workshop on Architecture Research using FPGA Platforms, held at HPCA-11*. 2005: San Francisco, CA.

[30] Davis, J.D., J. Laudon, and K. Olukotun, "*Maximizing CMP Throughput with Mediocre Cores*," in *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*. 2005, IEEE Computer Society.

[31] Davis, J.D., et al., "*A chip prototyping substrate: the flexible architecture for simulation and testing (FAST)."* SIGARCH Comput. Archit. News, 2005. **33**(4): p. 34-43.

[32] DiNI Group; "*ASIC Prototyping Logic Emulation Overview and Selection Guide*." 2006; Available from: http://www.dinigroup.com/files/dini_brochure.pdf.

[33] Dubois, M., et al., "*Rapid Hardware Prototyping on RPM-2."* IEEE Des. Test, 1998. **15**(3): p. 112-118.

[34] Emer, J., et al., "*Asim: A Performance Model Framework."* Computer, 2002. **35**(2): p. 68-76.

[35] Faylor, C. and C. Vinschen; "*Cygwin*." 2006; Available from: http://www.cygwin.com/.

[36] Franklin, M. and G.S. Sohi, "*ARB: A Hardware Mechanism for Dynamic Reordering of Memory References."* IEEE Trans. Comput., 1996. **45**(5): p. 552-571.

[37] Fujitsu, et al.; "*System-C Version 2.1 User Guide*." 2005; Available from: http://www.systemc.org/.

[38] Gaisler Research AB.; "*LEON3 Processor*." 2005; Available from: http://www.gaisler.com/leonmain.html.

[39] Gelsinger, P.P., "*Microprocessors for the New Millennium: Challenges, Opportunities, and New Frontiers*," in *ISSCC 2001 Digest of Technical Papers*. 2001.

[40] Gibson, J., et al., "*FLASH vs. (Simulated) FLASH: closing the simulation loop*," in *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*. 2000, ACM Press: Cambridge, Massachusetts, United States.

[41] Giga Semiconductor Inc.; "*2M x 18, 1M x 36, 512K x 72 36Mb S/DCD Sync Burst SRAMs*." 2001; Available from: http://www.gsitechnology.com/8324183672.pdf

[42] Giga Semiconductor Inc.; "*72Mb Pipelined and Flow Through Synchronous NBT SRAM.*" 2006; Available from: http://www.gsitechnology.com/8640Z1836.pdf.

[43] Gopal, S., et al., "*Speculative Versioning Cache*," in *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*. 1998, IEEE Computer Society.

[44] Hammond, L., "*Hydra: A Chip Multiprocessor with Support for Speculative Thread-Level Parallelization*," in *Electrical Engineering*. 2002, Stanford University: Stanford.

[45] Hammond, L., et al., "*Programming with transactional coherence and consistency (TCC)*," in *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*. 2004, ACM Press: Boston, MA, USA.

[46] Hammond, L., et al., "*The Stanford Hydra CMP.*" IEEE Micro, 2000. **20**(2): p. 71-84.

[47] Hammond, L., et al., "*Transactional Memory Coherence and Consistency*," in *Proceedings of the 31st annual international symposium on Computer architecture*. 2004, IEEE Computer Society.

[48] Hennessy, J., "*The Future of Systems Research.*" Computer, 1999. **32**(8): p. 27-33.

[49] Hennessy, J. and P. Nye, "*Stanford Small Benchmark Suite.*" 1989.

[50] Hughes, C.J., et al., "*RSIM: Simulating Shared-Memory Multiprocessors with ILP Processors.*" Computer, 2002. **35**(2): p. 40-49.

[51] IBM; "*Processor Local Bus (64-bit)* " 2001; Available from: http://www-3.ibm.com/chips/techlib/techlib.nsf/techdocs/8BA965C773B2E0ED87256AB20082CC9F.

[52] IEEE, "*IEEE Std 1149.1-1990 IEEE Standard Test Access Port and Boundary-Scan Architecture -Description.*" 1990.

[53] Integrated Device Technology Inc.; "*High-Speed 3.3V 64K x 36 Asynchronous Dual-Port Static Ram.*" 2001; Available from: http://www.idt.com/products/pages/Multi-Ports-70V658.html

[54] Integrated Device Technology Inc., "*R3000/3001 Designer's Guide*. 1990.

[55] Integrated Device Technology Inc.; "*High-speed 2.5V 1024K x 36 Synchronous Dual-Port Static RAM with 3.3V OR 2.5V Interface.*" 2006; Available from: http://www.idt.com/products/files/10504531/5682pgm.pdf.

[56] Intel Corporation; " *Intel® Pentium® D Processor Powered by Intel dual-core technology* " 2006; Available from: http://www.intel.com/products/processor/pentium_D/index.htm.

[57] Intel Corporation; "*Processor Roadmap*." 2006; Available from: http://www.intel.com/products/roadmap/index.htm.

[58] ITRS; "*International Technology Roadmap for Semiconductors*." 2005; Available from: http://www.itrs.net/Links/2005ITRS/Home2005.htm.

[59] JEDEC SOLID STATE TECHNOLOGY ASSOCIATION, "*FBDIMM Specification: High Speed Differential PTP Link at 1.5 V.*" 2006.

[60] Keckler, S.W., et al., "*Exploiting fine-grain thread level parallelism on the MIT multi-ALU processor*," in *Proceedings of the 25th annual international symposium on Computer architecture*. 1998, IEEE Computer Society: Barcelona, Spain.

[61] Kongetira, P., K. Aingaran, and K. Olukotun, "*Niagara: A 32-Way Multithreaded Sparc Processor.*" IEEE Micro, 2005. **25**(2): p. 21-29.

[62] Krishnan, V. and J. Torrellas, "*A Chip-Multiprocessor Architecture with Speculative Multithreading.*" IEEE Trans. Comput., 1999. **48**(9): p. 866-880.

[63] Kumar, R., et al., "*Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance*," in *Proceedings of the 31st annual international symposium on Computer architecture*. 2004, IEEE Computer Society.

[64] Laudon, J. and D. Lenoski, "*The SGI Origin: a ccNUMA highly scalable server*," in *Proceedings of the 24th annual international symposium on Computer architecture*. 1997, ACM Press: Denver, Colorado, United States.

[65] Law, J., et al., "*GCC, the GNU Compiler Collection* ". 2006.

[66] Lenoski, D., et al., "*The DASH prototype: implementation and performance*," in *Proceedings of the 19th annual international symposium on Computer architecture*. 1992, ACM Press: Queensland, Australia.

[67] Macraigor Systems LLC; "*J-SCAN the Ultimate JTAG debugger*." 2005; Available from: http://www.jscan.com/jscan_product.htm.

[68] Magnusson, P.S., et al., "*Simics: A Full System Simulation Platform.*" Computer, 2002. **35**(2): p. 50-58.

[69] Mai, K., et al., "*Smart Memories: a modular reconfigurable architecture*," in *Proceedings of the 27th annual international symposium on Computer architecture*. 2000, ACM Press: Vancouver, British Columbia, Canada.

[70] Matzke, D., "*Will Physical Scalability Sabotage Performance Gains?*" Computer, 1997. **30**(9): p. 37-39.

[71] McNairy, C. and R. Bhatia, "*Montecito: A Dual-Core Dual-Thread Itanium Processor*," in *IEEE Micro*. 2005.

[72] Mentor Graphics Corporation; "*ModelSim SE*." 2006; Available from: http://www.model.com/products/products_se.asp.

[73] Mentor Graphics Corporation; "*Scalable Verification: Emulation*." 2006; Available from: http://www.mentor.com/products/fv/emulation/index.cfm.

[74] MIPS Technologies Inc.; "*MIPS SDE Lite*." 2006; Available from: http://www.mips.com/content/Products/SoftwareTools/SDE_Lite/content_html.

[75] Njoroge, N., et al., "*Building and Using the ATLAS Transactional Memory System*," in *Proceedings of the Workshop on Architecture Research using FPGA Platforms, held at HPCA-12*. 2006: Austin, Texas.

[76] Olukotun, K., L. Hammond, and M. Willey, "*Improving the performance of speculatively parallel applications on the Hydra CMP*," in *Proceedings of the 13th international conference on Supercomputing*. 1999, ACM Press: Rhodes, Greece.

[77] Papadopoulos, G. and D. Yen; "*The New System* " 2003; Available from: http://www.sun.com/events/analyst2003/presentations/Papadopoulos_Yen_WWAC_022503.pdf.

[78] Prabhu, M.K. and K. Olukotun, "*Exposing speculative thread parallelism in SPEC2000*," in *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*. 2005, ACM Press: Chicago, IL, USA.

[79] Rabbit Semiconductor, "*RCM3200 RabbitCore Models RCM3200, RCM3210, RCM3220 "* 2006.

[80] Richardson, S.E., "*MPOC: A Chip Multiprocessor for Embedded Systems*," in *HPL Technical Report*. 2002, Hewlett Packard: Palo Alto.

[81] Rosenblum, M., et al., "*Using the SimOS machine simulator to study complex computer systems."* ACM Trans. Model. Comput. Simul., 1997. **7**(1): p. 78-103.

[82] Sandisk Corporation, "*SanDisk Extreme® IV CompactFlash® 8GB."* 2006.

[83] Sankaralingam, K., et al., "*Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture*," in *Proceedings of the 30th annual international symposium on Computer architecture*. 2003, ACM Press: San Diego, California.

[84] Sanmina-SCI; "*Printed Circuit Board Assembly and Test DFX Guidelines*." 2004; Revision 5:[Available from: www.sanmina-sci.com.

[85] Shankland, S.; "*IBM chip architect guns for gigahertz*." 2006; Available from: http://news.zdnet.com/2100-9584_22-6039293.html.

[86] Sohi, G.S., S.E. Breach, and T.N. Vijaykumar, "*Multiscalar processors*," in *Proceedings of the 22nd annual international symposium on Computer architecture*. 1995, ACM Press: S. Margherita Ligure, Italy.

[87] Steffan, J. and T. Mowry, "*The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization*," in *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*. 1998, IEEE Computer Society.

[88] Sun Microsystems Inc.; "*Solaris Enterprise System*." 2006; Available from: http://www.sun.com/software/solaris/index.jsp.

[89] Sun Microsystems Inc.; "*Sun Studio*." 2006; Available from: http://developers.sun.com/sunstudio/index.jsp.

[90] Synplicity Inc.; "*Synplify Pro*." 2006; Available from: http://www.synplicity.com/products/synplifypro/index.html.

[91] Tensilica Inc.; "*Xtensa Configurable Processors - Overview*." 2006; Available from: http://www.tensilica.com/products/xtensa_overview.htm.

[92] Texas Instruments Inc.; "*C6000 DSP Roadmap*." 2006; Available from: http://focus.ti.com/dsp/docs/dspplatformscontento.tsp?sectionId=2&familyId=132&tabId=493.

[93] Tharas Systems; "*Hammer SX & Hammer MX*." 2006; Available from: http://www.tharas.com/products/.

[94] Torvalds, L.; "*GNU/Linux*." 2006; Available from: http://www.linux.org/.

[95] Usselmann, R.; "*Open source single precision floating point unit*." 2000; Available from: http://www.opencores.org/projects.cgi/web/fpu/overview.

[96] Waingold, E., et al., "*Baring It All to Software: Raw Machines."* Computer, 1997. **30**(9): p. 86-93.

[97] Wall, D.W., "*Limits of Instruction-Level Parallelism*," in *WRL Research Report*. 1993, Digital Western Research Laboratory: Palo Alto, CA.

[98] Wasson, S.; "*Intel reveals details on new CPU design*." 2005; Available from: http://www.techreport.com/onearticle.x/8695.

[99] Watson, G., N. McKeown, and M. Casado, "*NetFPGA: A Tool for Network Research and Education*." in *Proceedings of the Workshop on Architecture Research using FPGA Platforms, held at HPCA-12*. 2006.

[100] Xilinx Inc.; "*XC4000XLA/XV Field Programmable Gate Arrays*." 1999; Available from: http://direct.xilinx.com/bvdocs/publications/ds015.pdf.

[101] Xilinx Inc.; "*Virtex™ 2.5 V Field Programmable Gate Arrays*." 2001; Available from: http://direct.xilinx.com/bvdocs/publications/ds003.pdf.

[102] Xilinx Inc.; "*Board Routability Guidelines with Xilinx Fine-Pitch BGA Packages*." 2002; Available from: http://www.xilinx.com/bvdocs/appnotes/xapp157.pdf.

[103] Xilinx Inc.; "*Virtex™-E 1.8 V Extended Memory Field Programmable Gate Arrays*." 2002; Available from: http://direct.xilinx.com/bvdocs/publications/ds025.pdf.

[104] Xilinx Inc.; "*PLB vs. OCM Comparison Using the Packet Processor Software*." 2004; Available from: http://direct.xilinx.com/bvdocs/appnotes/xapp644.pdf.

[105] Xilinx Inc.; "*Accelerated System Performance with APU-Enhanced Processing*." 2005; Available from: http://www.xilinx.com/publications/xcellonline/xcell_52/xc_pdf/xc_v4acu52.pdf .

[106] Xilinx Inc.; "*Constraints Guide*." 2005; Available from: http://toolbox.xilinx.com/docsan/xilinx7/books/docs/cgd/cgd.pdf.

[107] Xilinx Inc.; "*Data2Mem Overview*." 2005; Available from: http://toolbox.xilinx.com/docsan/xilinx7/books/data/docs/dev/dev0200_29.html.

[108] Xilinx Inc.; "*iMPACT Overview*." 2005; Available from: http://toolbox.xilinx.com/docsan/xilinx7/help/iseguide/mergedProjects/impact/html/imp_b_overview.htm.

[109] Xilinx Inc.; "*Local Clocking Resources in Virtex-II Devices*." 2005; Available from: http://direct.xilinx.com/bvdocs/appnotes/xapp609.pdf.

[110] Xilinx Inc.; "*PACE Features*." 2005; Available from: http://toolbox.xilinx.com/docsan/xilinx7/help/iseguide/mergedProjects/pace/html/pace_b_featuresofpace.htm.

[111] Xilinx Inc.; "*Virtex-II Platform FPGAs: Complete Data Sheet*." 2005; Available from: http://direct.xilinx.com/bvdocs/publications/ds031.pdf.

[112] Xilinx Inc.; "*Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*." 2005; Available from: http://direct.xilinx.com/bvdocs/publications/ds083.pdf.

[113] Xilinx Inc.; "*ChipScope Pro Serial I/O Toolkit User Guide*." 2006; Available from: http://www.xilinx.com/ise/verification/chipscope_pro_siotk_8_1i_ug213.pdf.

[114] Xilinx Inc.; "*CORE Generator*." 2006; Available from: http://www.xilinx.com/products/design_tools/logic_design/design_entry/coregenerator.htm.

[115] Xilinx Inc.; "*FPGA Development Boards*." 2006; Available from: http://www.xilinx.com/products/devboards/index.htm.

[116] Xilinx Inc.; "*ISE Foundation*." 2006; Available from: http://www.xilinx.com/ise/logic_design_prod/foundation.htm.

[117] Xilinx Inc.; "*MicroBlaze Processor Reference Guide*." 2006; Available from: http://www.xilinx.com/ise/embedded/mb_ref_guide.pdf.

[118] Xilinx Inc.; "*OPB PCI v1.02a User Guide*." 2006; Available from: http://www.xilinx.com/bvdocs/userguides/opb_pci_ug_241.pdf.

[119] Xilinx Inc.; "*Parallel IV Cable*." 2006; Available from: http://www.xilinx.com/xlnx/xebiz/designResources/ip_product_details.jsp?key=HW-PC4&iLanguageID=1.

[120] Xilinx Inc.; "*Platform Studio and the EDK* " 2006; Available from: http://www.xilinx.com/ise/embedded_design_prod/platform_studio.htm.

[121] Xilinx Inc.; "*QDR II SRAM Interface for Virtex-4 Devices*." 2006; Available from: http://www.xilinx.com/bvdocs/appnotes/xapp703.pdf.

[122] Xilinx Inc.; "*Using Digital Clock Managers (DCMs) in Spartan-3 FPGAs*." 2006; Available from: http://direct.xilinx.com/bvdocs/appnotes/xapp462.pdf.

[123] Xilinx Inc.; "*Using the Virtex Delay-Locked Loop*." 2006; Available from: http://direct.xilinx.com/bvdocs/appnotes/xapp132.pdf.

[124] Xilinx Inc.; "*Virtex-4 RocketIO Multi-Gigabit Transceiver User Guide*." 2006; Available from: http://www.xilinx.com/bvdocs/userguides/ug076.pdf.

[125] Xilinx Inc.; "*Virtex-4 User Guide*." 2006; 387]. Available from: http://direct.xilinx.com/bvdocs/userguides/ug070.pdf.

[126] Xilinx Inc.; "*Virtex-5 Family Overview - LX and LXT Platforms*." 2006; Available from: http://direct.xilinx.com/bvdocs/publications/ds100.pdf.

[127] Xilinx Inc.; "*XCR3512XL: 512 Macrocell CPLD*." 2006; Available from: http://direct.xilinx.com/bvdocs/publications/ds081.pdf.