Automatic Generation of Interrupt-Aware Hardware Accelerators with the M2V Compiler

Abilash Sekar, Alessandro Forin
Microsoft Research

August 2008

Technical Report
MSR-TR-2008-110

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Automatic Generation of Interrupt-Aware Hardware Accelerators with the M2V Compiler

Abilash Sekar, Alessandro Forin
Microsoft Research

Abstract

The MIPS-to-Verilog (M2V) compiler and the Basic Block Tools (BBTools) can automatically generate a hardware accelerator for selected blocks of machine code in an application. The compiler translates blocks of MIPS machine code into a hardware design captured in Verilog (an "Extension"). The BBTools patch the application binary by inserting the extension instruction that triggers the accelerator. The original code is preserved, so that execution can fall back to software when necessary.

This work extends the M2V compiler with support for memory load and store instructions, and for interrupts. We use a transactional model to handle interrupts and/or traps due to TLB misses in the Extension. We implemented and tested the interrupt support mechanism using a 64-bit division basic block, with added instructions for memory loads off the stack pointer.

We also added the feature of allowing the BBTools to automatically create the best encoding for an extension instruction. The tool now evaluates which pair of roots in the dependency graph leads to the shortest execution cycle time for the Extension. With this addition, the process of creating Extensions for the eMIPS processor can now be fully automated and applied to practical applications, where loads and stores inside the Extension are of paramount importance.

1 Introduction

Extensible processors have a simple RISC pipeline and the ability to augment the Instruction Set Architecture (ISA) with custom instructions. The ISA can be augmented statically, at tape-out, or it can be augmented dynamically when applications are loaded. Extensible processors differ from other accelerators in their tight integration with the basic data path, which leads to minimal latencies and therefore greater flexibility. Examples of dynamically extensible processors include eMIPS [6] and Stretch [13]. Tensilica's Xtensa [14] is an example of a statically extensible processor.

Extensible processors take advantage of the fact that a small amount of code takes the majority of execution time in a typical program. The code that executes most often is a candidate for hardware acceleration. The code is identified by a special instruction that will initiate the accelerator.

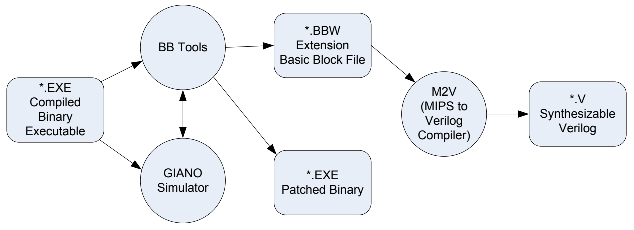

BB Tools
*.BBW Extension Basic Block File
*.EXE Compiled Binary Executable
M2V (MIPS to Verilog Compiler)
*.V Synthesizable Verilog
GIANO Simulator
*.EXE Patched Binary

Figure 1: The eMIPS Tool Chain to automate the generation of hardware accelerators.

Selection of the best code to accelerate is an active area of research. The eMIPS tool-chain, shown in Figure 1, restricts the code selection problem to the set of basic blocks in the application. Using the strict definition in [1], the basic block is a directed acyclic graph (DAG). A DAG is a set of machine instructions that do not contain branches and are branched-to only for the very first instruction. The best candidate blocks are currently selected by executing the application using the Giano full-system simulator [21], in concert with the data obtained via static analysis of the application binary. The BBTools select the basic blocks to accelerate and patch the binary image with the special instructions for the accelerator. The M2V compiler [16] automatically generates the design for the hardware accelerator, which is then synthesized using the Xilinx tools for the ML40x boards.

The accelerator can be statically loaded when eMIPS is reset or it can be dynamically loaded when an application is loaded using partial reconfiguration of the FPGA. By dynamically loading and unloading accelerators, the area of the programmable hardware can be used more efficiently.

In previous versions of eMIPS, the accelerator blocks could be specified and given to a hardware designer to hand design the accelerator. While this can lead to an efficient implementation, manual designs do not scale well as dynamically extensible processors are more widely used and the hardware becomes more complex. To effectively utilize dynamically extensible processors, different parts of an application can be accelerated using different Extensions, which can be loaded at appropriate times. The use of the tool chain along with M2V can expand the use of hardware acceleration and completely automate the process of generating hardware accelerators for a variety of basic blocks in an application.

The work described herein addresses three limitations in the tool chain that prevented the M2V tool chain from being usable in but a few practical cases. The first limitation was the lack of support for load and store operations, and more generally for variable-cost operations. M2V was previously only able to cope with MIPS instructions that took zero or one cycle. We added support for all the instructions that took a fixed number of cycles, accounting for the cost in the generation of the schedule. Then we added support for the instructions that have variable costs, prime and foremost loads and stores. We were able to find a way to preserve the overall structure of the compiler, while dealing with the variable costs. The dependency graph still leads to a state machine that controls the overall execution. The transitions are now defined not only by the clock, but also by the signals that indicate completion of the variable-cost operations.

The second limitation concerns the inability of the M2V compiler to generate logic for handling interrupts within an Extension. External interrupts could only happen before or after the extension instruction, never inside it. This assumption is invalid in the presence of TLB misses due to loads and stores. Furthermore, the previous tool required that an Extension never encounter errors, such as arithmetic overflows or unaligned addresses. For a real-time system, it is important to respond to interrupts in a timely and predictable manner. Even in a general-purpose OS it is unacceptable to allow a user process to ignore interrupts and lock the machine. To address the interrupt limitation, we used the concept of transactions in dealing with the write-backs to the register file and the stores to memory. The overall execution of the Extension is subdivided in sets that execute “atomically”. Interruptions of any sort are accepted at the transaction boundaries. On interruption, the write-back machine cancels all write-backs from future transactions, completes the write-backs for the current one, and then relinquishes control back to the data-path in a limited amount of time. The restart-address is set to the point in the original basic block that corresponds to the current write-back state. It is therefore mandatory that extension instructions are simply inserted in the original image, and that they do *not* replace the original basic block.

The third limitation addresses the task of automatically choosing an instruction encoding for the new extension instruction. We observed that the selection of which registers or constants to encode in the instruction can have an effect on the overall execution time. These values are available early in the execution pipeline. It is therefore important to select those that allow the most work to proceed before stalling on a dependency. Our algorithm uses two parameters in deciding which two register numbers to encode – fan-out and depth of the root register read nodes. It selects the pair of registers with the maximum combined fan-out and depth.

The remainder of this document is structured as follows. Section 2 discusses related work. Section 3 gives an overview of the eMIPS hardware platform. Section 4 discusses the automatic encoding of the extension instruction by the BBTools. Section 5 defines the support for memory operations in the M2V compiler. Section 6 discusses the model and implementation for handling interrupts in the Extension. Section 7 explains the M2V generated hardware in more detail. Section 8 compares the synthesis reports obtained for Extensions generated with and without the transactional model enabled. Section 9 discusses the experimental results, and Section 10 concludes the report. Appendix I contains the BBW file for the example basic block. Appendix II contains synthesis reports for the Extension generated by the M2V compiler. Appendix III contains the Verilog output from M2V, with the transactional model enabled and interrupt support. Appendix IV shows an automated visualization of the dependency graphs from M2V.

2 Related Work

Work on extensible processors can be divided in several ways. One avenue of exploration is to define the underlying hardware. Chimaera [7] and GARP [8] are two examples of extensible hardware from the late 1990’s. Commercial FPGA manufacturers today all provide examples of soft-cores, microprocessor designs that the customer can modify and extend for their application [15, 3, 13]. M2V uses the eMIPS processor [6] as its underlying hardware platform. eMIPS is the first design that is secure for general purpose multi-user loads, and the set of potential applications is therefore more open-ended than those found in the typical embedded system alone.

A common approach to generate code for an extensible processor is to modify an existing C compiler. Tensilica [14] automatically regenerates a full GNU compilation system given the RTL of the new instruction. Ienne et al. [4] use the SUIF compiler. M2V accepts as input binary machine code rather than source code. There are trade-offs between accelerating from source code in a high-level language or from binaries. One of the major advantages when accelerating from binaries is that any application can be accelerated, even applications where the source code is controlled by an outside party and not available to the system developer. A disadvantage is that some of the information that has been discarded must be reconstructed, and there are limits to this reversal process.

The FREEDOM compiler [17, 18] is similar to M2V; the compiler accepts binary machine code as input and maps it to an FPGA. The Extensions generated by the M2V compiler are meant for a general-purpose environment and therefore execute in coordination with the main processor data path, whereas FREEDOM maps the entire program to the bare FPGA. M2V generates Extensions that are explicitly interrupt-aware, whereas there is no mention of handling interrupts in the FREEDOM compiler. Additionally, the Extensions generated by the M2V compiler for the eMIPS have secure access to the memory subsystem via the Memory Management Unit (MMU), which is not a requirement for the DSP-like programs handled by the FREEDOM compiler. FREEDOM is a more mature product and can handle a larger body of codes than what M2V currently does.

Another avenue of research in extensible processors is the identification of the Instruction Set Extensions (ISE) that most benefit a given program, see for instance [5] for a recent overview. Bonzini [5] advocates generating the ISE from within the compiler, Tensilica [14] from profiling data. M2V currently follows the application profiling approach; it uses the BBTools and dynamic full-system simulation with Giano to select the candidate basic blocks. The current approach can extend to handling chains of blocks e.g. in frequently executed loops that are automatically recognized via full-system simulation [19]. A possible addition to our work is to use M2V in concert with a high-level compiler. Once the ISE is identified from within the compiler, the Extension's definition could be output in the form of a BBW file.

A related area is the generation of HDL code from C, the so-called C-to-gates design flows [11, 12]. The common target is the automated generation of HDL code from sequential programs. The main difference with M2V is that the input is binary code. Using binary code supports all programming languages, included dynamically generated (jitted) code. It is the only viable option in case the high level source code is not available, e.g. for third-party code and libraries. The drawback is that makes the problem harder. The binary code has already been optimized (register allocation, loop unrolling, etc) during its compilation hence identifying parallelism is more difficult. The BBTools framework tries to account for some of these optimizations by using a canonical form of the basic block, so that it can identify repeating basic blocks in the binary.

3 eMIPS Hardware Overview

The extensible MIPS (eMIPS) processor [6] has been developed at Microsoft Research as an example of a RISC processor integrated with programmable logic. The programmable logic has many uses, such as: extensible on-line peripherals, zero overhead online verification of software, hardware acceleration of general-purpose applications, and in-process software debugging [2]. This document is concerned with automatically generating hardware accelerators within the context of the eMIPS extensible processor.

The instruction set for the eMIPS processor is the instruction set for the R4000 MIPS processor [10]. The R4000 is an example of a classic RISC architecture. The eMIPS pipeline follows the classic RISC pipeline [9] consisting of five stages: instruction fetch (IF), instruction decode (ID), execute (EX), memory access (MA), and register write-back (WB).

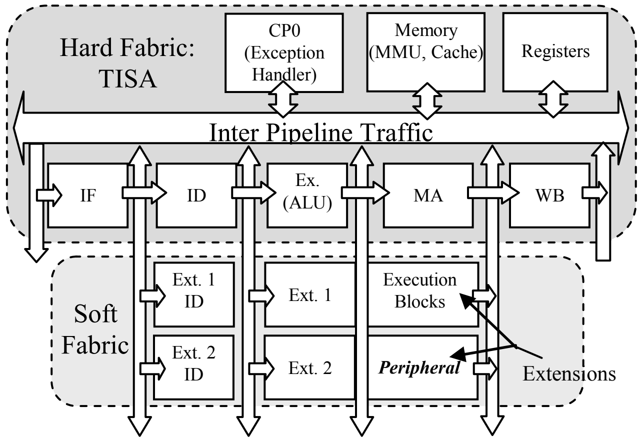


Figure 2. eMIPS block diagram. The Soft Fabric can be reconfigured at run time to extend the ISA.

The eMIPS processor departs from a standard RISC processor by adding an interface to programmable logic. The programmable logic is tightly integrated with the RISC pipeline, it can synchronize with it and it can access the same resources as the RISC pipeline. Figure 2 shows a logical block diagram for the eMIPS processor. The tight coupling of the pipeline and programmable logic creates a very low latency interface between the accelerator and the RISC pipeline.

Figure 3 illustrates the pipelining of instructions through eMIPS. The decode logic in the extension logic is always an observer of the main pipeline and is trying to decode the instruction in the instruction decode (ID) phase of the pipeline. When the instruction is not an extension instruction, the Extension fails to decode it and the instruction is executed in the main pipeline. If instead the extension logic successfully decodes the instruction, the extension becomes active and hardware acceleration takes over execution. Instructions flowing through the main RISC pipeline prior to the extension instruction complete normally. Instructions following the extension instruction are stalled until the Extension is near completion, in the EXn-1 cycle.

The RISC pipeline imposes micro-architectural constraints on the extension logic, for instance in the arbitration for access to the register file and other resources. The extension logic needs to read and write the register file and access the memory management unit (MMU). M2V automatically schedules all resource accesses in the extension logic to avoid conflicts with the primary RISC pipeline.

Cycle Number	0	1	2	3	4	5	...	n+2	n+3	n+4	n+5	n+6	n+7	n+8
Instruction m-2	IF	ID	EX	MA	WB									
Instruction m-1		IF	ID	EX	MA	WB								
Extension Instruction m			IF	ID	EX1	EX2	...	Exn-2	EXn-1	EXn	MA	WB		
Instruction m+1				IF										
Instruction j									IF	ID	EX	MA	WB	
Instruction j+1										IF	ID	EX	MA	WB

Figure 3: Instruction flow through the eMIPS pipeline.

The primary RISC pipeline uses two read ports on the register file when an instruction is in the ID stage, one MMU port when in the MA stage, and one write port on the register file when in the WB stage. The eMIPS register file has four ports which are multiplexed between four read ports and two write ports. The extension logic has the potential to use all of the eMIPS register file ports, but it must not conflict with the primary RISC pipeline. Thus, register writes must be delayed by the Extension until previous instructions are retired and register reads must be finished a couple of cycles before trailing instructions get to the ID stage. As a specific example, consider the case in Figure 3 when the extension instruction is in the EX1 cycle of execution, instruction m-1 is in the MA pipeline stage and so instruction m-1 has access to the MMU. Instruction m-2 is in the WB pipeline stage and it has control of the register file write ports. The extension instruction does not have control of all the resources until stage EX3 when the previous instructions have been retired.

The eMIPS processor has been implemented on Xilinx Virtex 4 FPGAs using the ML401 and ML402 evaluation boards. The partial reconfiguration capabilities of this FPGA model allow software to load dynamically the hardware for the instruction extensions.

4 Extension Instruction Encoding

An extension instruction is an instruction that is not part of the base ISA of the eMIPS processor. It is inserted in the instruction stream for the specific purpose of triggering an Extension. If the Extension is present and active, it recognizes the instruction and takes over execution, effectively replacing the block of instructions that (would) follow. Otherwise the instruction is treated as a NOP and execution continues with the original basic block. There is a large degree of freedom in encoding an extension instruction; the only real practical restriction is that the top six bits must not match a valid opcode. It is also clearly impossible to encode all of the information contained in an arbitrarily large block of instructions into a single one. The implementation of the eMIPS decoder presents an opportunity for optimization. By default, the decoder expects "I" format instructions and fetches the corresponding *rs* and *rt* registers in advance. Consequently, the current revision of M2V generates Extensions that decode their extension instruction assuming the MIPS "I" format. The format is illustrated in Figure 4 below.

Opcode_name rt, rs, immediate

opcode	rs	rt	immediate
31:26	25:21	20:16	15:0

Figure 4: The MIPS "I" format encoding.

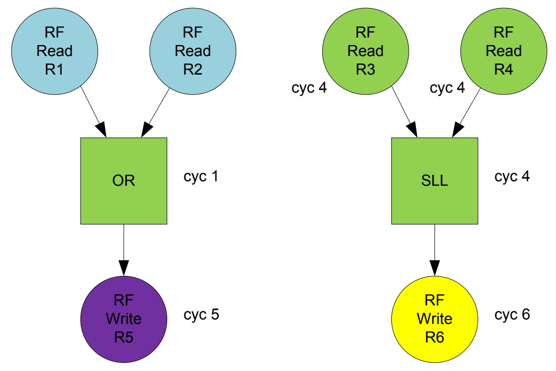
The M2V compiler uses the block descriptions in the so-called BBW file to generate synthesizable Verilog code for the Extension hardware of the eMIPS processor. The BBW file describes a list of basic blocks; each description consists of the following main sections: machine name, extension instruction encoding, canonical register and value relationships, code size and the basic block of MIPS instructions. *BBMatch.exe* is a program, part of the BBTools framework, used for creating the BBW source file automatically, from a MIPS executable file. The BBW file is therefore the core interface between BBTools and M2V compiler. The current version of BBMatch creates the BBW file automatically, but leaves the *encoding* section empty. If we could also generate the encoding section automatically the whole process of creating Extensions could be automated, from ELF image all the way to a working Extension.

The encoding of the registers in the extension instruction plays an important part in the schedule that M2V will generate for the Extension. In the eMIPS architecture, the Extension is tightly coupled with the standard MIPS pipeline. This gives us the advantage of having the two registers *rs & rt*, encoded in the extension instruction, available directly from the decode stage of the MIPS pipeline. This feature can be used to reduce the execution time of the Extension by appropriately selecting the registers to encode in the extension instruction. This is illustrated by the simple example basic block shown below in Figure 5. The cost of the "OR" and the "SLL" instructions in the basic block are 1 and 0 cycles respectively. We shall consider two cases to illustrate the importance of encoding the correct registers.

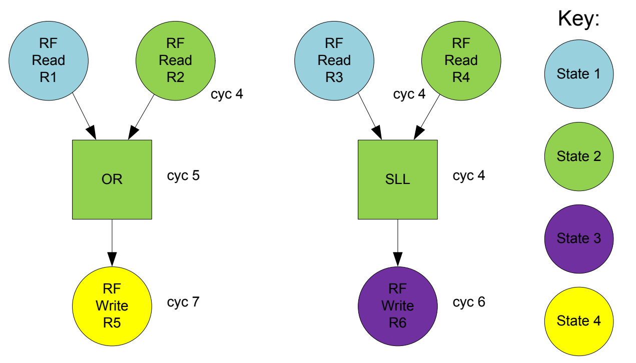
```
[0] ext0 rx, ry, offset
[4] or r5, r1, r2
[8] sll r6, r3, r4
```

Figure 5: The choice of rx and ry in this basic block affects the performance of the generated Extension.

The circuit graph generated by M2V is shown in Figure 6 for two different encodings of the extension instruction. In the graphs, the clock cycle when the respective node completes is depicted next to the node.



Case (i): Extension instruction encoded with R1, R2



Case (ii): Extension instruction encoded with R1, R3

Figure 6: Circuit graphs for the block in Figure 2, using different encoding schemes.

The graphs show that though the number of states in the Extension remains the same, the number of clock cycles taken by the Extension to execute the set of instructions differs based on the encoding of the registers.

Considering the first case, registers R1 and R2 are encoded, thus making them available directly from the decode phase of the pipeline in stage 2 of the extension state machine. The "OR" instruction can be executed immediately, and will complete in cycle 1 since the Extension has all the registers available and no unmet dependencies. However, the "SLL" instruction requires both source registers to be read from the register file, which takes 4 clock cycles. This causes the SLL instruction to complete in cycle 4. A pipeline stage is inserted by the extension state machine after execution of the instructions at cycle 4. The two register write-backs are performed after the pipeline stage, in cycle 5 (R5) and cycle 6 (R6). Thus, the Extension requires 6 cycles to complete execution with this encoding.

Considering the second case, registers R1 and R3 are encoded. In this case, none of the instructions can be executed directly as both have unmet dependencies and require register reads from the register file. Assuming there are at least two read ports in the register file, the OR instruction completes after 5 cycles, 4 cycles for reading register R2 and 1 cycle for execution. Similarly, the SLL instruction completes execution in cycle 4. Again, a pipeline stage is inserted after execution of the instructions in cycle 5. The register write-backs are performed in cycle 6 (R6) and cycle 7(R5). Thus the Extension requires 7 cycles to complete execution with this encoding.

In this minimal example, a two instruction basic block shows a difference of 1 execution cycle depending on the selected encoding. The encoding scheme will have a greater impact on the execution time when there are long latency paths in the basic block. Thus, it is of prime importance to create an optimal encoding of the Extension's registers to reduce Extension latency.

4.1 Register Selection Algorithm

In the new version of *"bbmatch.exe"*, the encoding algorithm uses two main parameters in selecting the *rs & rt* registers – fan-out and depth of the root register read nodes. Fan-out is the number of instructions dependent on the root register read node. Depth is a count of the register nodes and the cost of the instruction nodes till a dependency is met in the graph.

Using the circuit graphs in Figure 6, all the register read nodes, R1, R2, R3 and R4 have a fan-out of 1. For the depth calculation, all the registers have a dependency at the instruction nodes, with the only differentiating factor being the cost of the "OR" instruction node compared to the "SLL" instruction node. This gives the depth of registers R1 & R2 as 2 and R3 & R4 as 1.

The algorithm takes the sum of the fan-out and depth of the register nodes and encodes the nodes with the maximum value. In the mentioned example, the nodes encoded by the algorithm would be R1 and R2, which is the best encoding scheme as seen from the circuit graphs in Figure 6.

Other algorithms are possible. The total number of general-purpose registers is limited even in the case of MIPS where they are abundant, and the calling convention further restricts the number of maximum potential roots in any practical dependency graph. It is therefore conceivable to perform a brute-force search of all possible selections to find the one with the optimal cycle count. The worst-case number of alternatives for a processor with N usable registers is N*(N-1)/2, or 465 for MIPS. We will try this alternative when the compiler has full code coverage.

4.2 Immediate Value Encoding

The encoded extension instruction can be used to match more than one basic block in the binary executable. The description of a block is in the form of a code pattern, parameterized by the register assignments and the constants in the immediate fields. The encoding of the instruction defines constraints on the register assignments and constants. Any block with a compatible register and constants assignment will match the pattern. For example, if two basic blocks differs only in the destination of a jump, then encoding the extension instruction with the branch offset would make the Extension work in both cases.

In the new version of BBMatch, the immediate value encoded in the Extension is determined by user options. The program simplistically allows for encoding either the first encountered load instruction offset or the branch instruction offset. By default, the program encodes the branch offset.

4.3 Implementation

We assume the reader is familiar with the internals of the M2V compiler implementation, as described in [16]. From the top level, the main functions involved in the decision making are part of the "ext_encode.cpp" file – *RegEncodingCond & ValEncodingCond,* which are the functions for encoding the canonical register relationships and values, and the value relationships respectively.

The BBTools must generate the circuit graph in order to decide which registers should be encoded as *rs & rt*, just like M2V does. *RegEncodingCond* first creates the circuit graph by using the *Circuit* class from M2V. *RegEncodingCond* then calls *getEncRegs*, which we have added to the *Circuit* class to determine *rs & rt*. The rest of the *RegEncodingCond* function determines the canonical register relationships once *rs & rt* are determined.

The *getEncRegs* function calculates the fan-out and depth of each root register read node of the circuit graph. The root register read nodes are stored in the *regRdVec* queue data structure in M2V. *rs & rt* are determined based on the additive maximum of fan-out and depth.

ValEncodingCond encodes the value based on the specified user option. "mips_dissect.c" contains records for each instruction based on its disassembly with information such as the number of reads/writes required, cost of the instruction, etc. A flag indicating a branch/load instruction is added to this record to help in identifying the value to be encoded.

4.4 Tool interface changes

The Makefile provided with the BBTools is used to compile both the BBTools and M2V. The command "nmake bbmatch.exe" compiles just the BBMatch program. The usage information can be obtained by invoking a tool without arguments. The command line for *bbmatch.exe* is:

```
bbmatch [-v] [-c [-l] [-b]] PatternFile BBSFile
```

The "-l" or "-b" options are used along with the "-c" option for creating the BBW file from the basic block symbols (BBS) file. The new "-l" option encodes the load instruction offset while the new "-b" option encodes the branch instruction offset.

The command line for *m2v.exe* is:

```
m2v [-v] [-t] [-g] bbwFileName [VerilogFileName]
```

The "-v" option generates more verbose output. The new option "-t" enables the transactional model. The "-g" option creates a graphic representation of the circuit graph, in the "DOT" file format usable with the MSAGL tool for Automatic Graph Layout[20]. Appendix IV shows the visualization of the Circuit graph for the basic block in Figure 10, using the DOT file generated by M2V and rendered by the MSAGL tool.

5 Memory access support in M2V

This section describes our additions to the M2V compiler to support memory accesses. It is well known in the literature that the lack of load and store operations leads to limited speedups from hardware acceleration. The operations are not only important for performance, but in our experience they are almost always present in the most-frequently executed basic blocks of an application, precisely the blocks that M2V wants to speed up.

M2V is a three-pass compiler which automatically generates eMIPS Extensions from a BBW source file. The first pass processes the BBW file and consists of three major steps: map the encoding for the extension instruction to the basic block, analyze the MIPS instructions, and build a circuit graph. The second pass schedules the operations that are represented in the graph. The third pass emits the Verilog that will be synthesized and placed in the eMIPS FPGA.

The eMIPS architecture allows for Extensions to access memory through the MMU just like the main MIPS processor data path. The MMU is part of the Trusted ISA portion of the eMIPS processor; The MMU is the only path to memory available to general, untrusted Extensions. The Extension is tightly coupled with the main MIPS pipeline. The Extension has access to the memory interface only after a few cycles it received control. This restriction is taken care of in the scheduling pass of the M2V compiler. To ensure correct execution of the memory instructions, the Extension must obey the protocols shown below in Figure 7 and Figure 8. The M2V compiler generates this logic using the scheduling of memory instructions and the states of the extension state machine. The protocols for the memory read and write requests are shown in Figure 7 and 8 respectively.

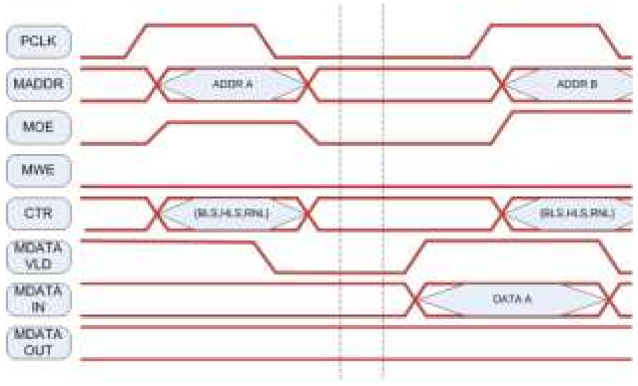


Figure 7: Memory Read Protocol.

In both protocols, the MDATA_VLD signal serves as an indicator that the memory request is acknowledged by the controller. The signal indicates when the data is available from, or to be written to memory.

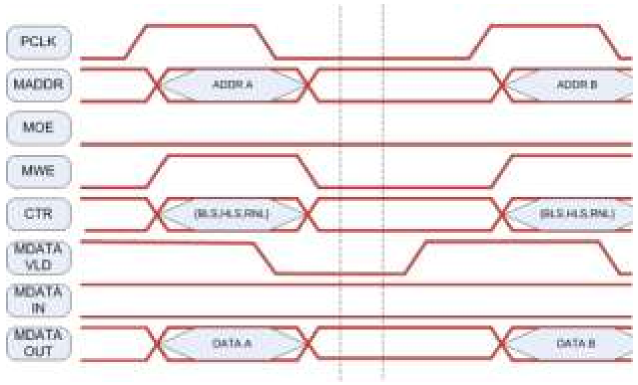


Figure 8: Memory Write Protocol.

The M2V compiler implements the above protocols in Verilog, in the form of a memory state machine. The memory state machine is then integrated into the existing extension state machine. The memory state machine is currently boilerplate (invariant) code.

5.1 Memory State Machine

The memory state machine shown in Figure 9 is a simple implementation of the memory protocols. M2V maintains an array of memory operations in a particular state that is integrated into the extension state machine. Once the Extension transitions from one state to a state with a memory operation, the memory state machine is activated.

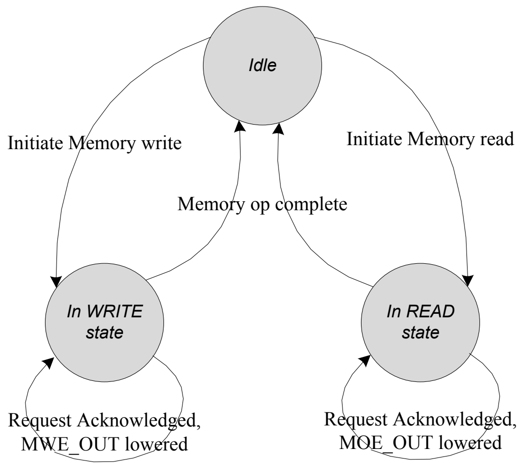


Figure 9: Extension Memory State machine.

On the rising edge of the Pipeline CLK (PCLK), the address is latched onto MADDR signal and the Memory Output Enable (MOE) signal is raised in case of a read or the Memory Write Enable (MWE) is raised in case of a write. MDATA_VLD then falls down once the memory request is acknowledged, and the memory state machine moves onto the next state, waiting for the MDATA_VLD to go high, indicating the availability of the data in MDATA_IN for a read or completion of the write for a write operation.

When a state involves a memory access, the main extension state machine waits on the completion of the memory state machine and transitions to the next state when all other operations (ex: register reads, register writes) for that state are completed.

5.2 Implementation

The set of memory instructions in the basic block are stored in a separate queue called *mmuInstVec* in the *Circuit* class. *mmuInstVec* is used to generate the memory address and memory data for the particular memory transaction. M2V schedules the memory operations just like other normal instructions. The register encoding algorithm described earlier takes care of the necessity to schedule loads as early as possible in the basic block. The encoding plays an important part in selecting two critical paths (of high depth). This will ensure that operations are executing in parallel to the load instruction, thereby hiding the latency of the load instruction.

Scheduling of memory operations by M2V is limited by the number of memory ports available to the Extension. In the current version, there is a single port available to the Extension, limiting the number of memory operations in a state of the Extension to one.

During the scheduling pass, when M2V encounters a memory instruction it first checks if the memory port has already been used. In the event that the port has already been used, the function returns with a *RET_FAIL_RSRC* error, indicating resource constraints. The register node supplying the base address is pipelined to allow access in the next state of the Extension.

If the memory port is available in the current state, the compiler proceeds to checking if the required register reads are already available. Once all the dependencies are met, the compiler marks the node with the cycle to be scheduled and updates the counter to reflect the usage of the memory port in the current state of the Extension.

M2V ensures that the ADDR_OUT/IN and DATA_OUT/IN pins are correctly MUX'd based on the instruction stream in the basic block. The M2V compiler generates a logic block to handle this requirement. If a basic block consists of more than one memory instruction, the ADDR_OUT pins must reflect the correct address for each of the instructions. As mentioned previously, only one memory operation is allowed in each state of the extension. Thus the ADDR_OUT/IN and DATA_OUT/IN pins are determined and MUX'd based on the current state of the extension.

M2V pipelines the destination register of the memory instruction to account for the variable latency of the load instruction node.

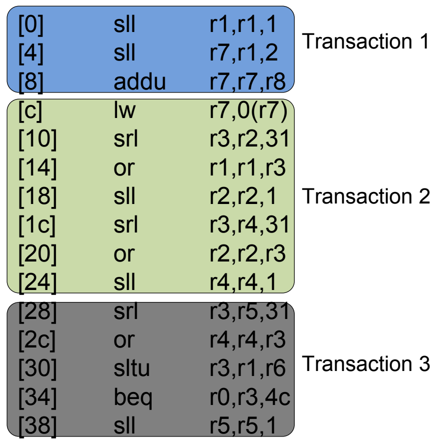
6 Interrupt Support

We have modified the M2V compiler to generate Verilog code with the ability to handle interruptions while the processor is executing in the Extension. Interruptions can be due to three different sources, but we will use the single term "interrupt" to indicate any and all of them. Our approach handles all cases in the same manner. The first cause of interrupts is address translation misses and errors in the MMU while the Extension is trying to access memory. A second cause is actual interrupts from peripherals such as timers and I/O devices. A third cause of interrupts is the case of errors inside the Extension, such as unaligned addresses and overflows. We use a transaction model based approach to handle all interrupts in the Extension.

The basic block to be accelerated is analyzed and divided into "transactions". A transaction is a set of instructions that terminate just before a memory instruction. Even in the event the basic block has no memory operations, there is still a maximum number of cycles allowed before interrupts are permitted. The maximum number of operations in a particular transaction is currently fixed at 7. Future work might include taking into account the actual latency/cost of the instructions rather than an arbitrary fixed number.

The Extension must correctly indicate to the TISA what is the re-start address for execution, e.g. after the software interrupt handler returns. This address is termed the Virtual PC (VIRPC), as the Extension is keeping track of the PC as seen by the MIPS pipeline, even though the Extension has no concept of instruction fetch or instruction ordering mechanisms. The VIRPC address simply corresponds to the start of each transaction in the original basic block.

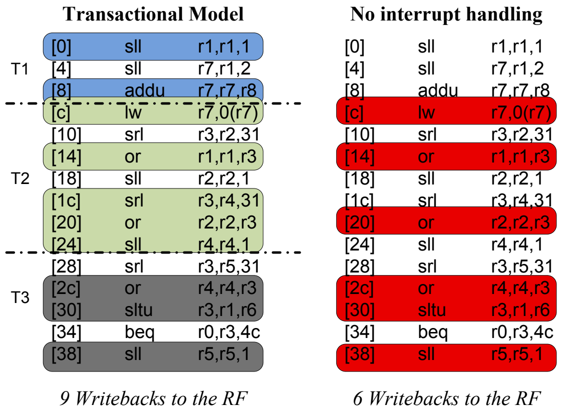
We illustrate the subdivision of a basic block in transactions with an example in Figure 10. Transaction 1 terminates once the load instruction ([c]) is encountered. Transaction 2 is terminated at the end of the maximum allowed 7 instructions in the transaction. The remaining instructions are part of transaction 3. The basic idea behind the transactions scheme is to preserve the original program order, while at the same time allowing for more optimistic and more parallel execution inside the Extension. The Extension will recognize an interrupt at the next transaction boundary should an interrupt occur during the Extension's execution. Any write-backs that are due to a subsequent transaction are aborted.



```
[0]    sll    r1,r1,1
[4]    sll    r7,r1,2
[8]    addu   r7,r7,r8
[c]    lw     r7,0(r7)
[10]   srl    r3,r2,31
[14]   or     r1,r1,r3
[18]   sll    r2,r2,1
[1c]   srl    r3,r4,31
[20]   or     r2,r2,r3
[24]   sll    r4,r4,1
[28]   srl    r3,r5,31
[2c]   or     r4,r4,r3
[30]   sltu   r3,r1,r6
[34]   beq    r0,r3,4c
[38]   sll    r5,r5,1
```

Figure 10: Example basic block illustrating the concept of transactions.

The transaction model is used to perform write backs in-order, but from the abstracted viewpoint of a transaction, that is, the write backs in transaction 1 must complete prior to any write backs in transaction 2. However, the write backs inside a particular transaction can be performed out of order. This limits the parallelism generated by the circuit graph to some extent by imposing the restriction of performing certain write backs in order. We perceive this to not be a huge problem as the eMIPS TISA interface allows for two register writes every cycle, thus decreasing the possibility of bottlenecks at the register file.



```
      [0]    sll    r1,r1,1
T1    [4]    sll    r7,r1,2
      [8]    addu   r7,r7,r8
- - - - - - - - - - - - - - - -
      [c]    lw     r7,0(r7)
      [10]   srl    r3,r2,31
      [14]   or     r1,r1,r3
T2    [18]   sll    r2,r2,1
      [1c]   srl    r3,r4,31
      [20]   or     r2,r2,r3
      [24]   sll    r4,r4,1
- - - - - - - - - - - - - - - -
      [28]   srl    r3,r5,31
T3    [2c]   or     r4,r4,r3
      [30]   sltu   r3,r1,r6
      [34]   beq    r0,r3,4c
      [38]   sll    r5,r5,1
```

```
[0]    sll    r1,r1,1
[4]    sll    r7,r1,2
[8]    addu   r7,r7,r8
[c]    lw     r7,0(r7)
[10]   srl    r3,r2,31
[14]   or     r1,r1,r3
[18]   sll    r2,r2,1
[1c]   srl    r3,r4,31
[20]   or     r2,r2,r3
[24]   sll    r4,r4,1
[28]   srl    r3,r5,31
[2c]   or     r4,r4,r3
[30]   sltu   r3,r1,r6
[34]   beq    r0,r3,4c
[38]   sll    r5,r5,1
```

Figure 11: Overhead of handling interrupts.

Figure 11 illustrates the overhead in terms of register write-backs. In the case of generating hardware without the support of interrupts only 6 write-backs are necessary to the register file. It can be clearly seen that further optimizations can be applied to this basic transaction model. By just terminating Transaction 2 an instruction before or after would have reduced the number of write-backs by 2. This would have ensured only one extra write-back in the transactional model approach.

M2V adds transactions registers to the Extension to keep track of the current transaction being written back for a particular state of the Extension. When the Extension encounters an interrupt, the extension state machine checks to see if the current state of the Extension is an end of a transaction or not. If it is an end of a transaction, the VIRPC is updated to reflect the address of the start of the next transaction and the extension state machine is stalled in that state. The Extension then waits for the resources to be taken away by the pipeline arbiter and the Enable and Grant signal to go low. Once the enable goes low, the Extension lowers the ACK signal to signal the end of the Extension at that transaction. The program then re-starts execution from the VIRPC address on the main MIPS processor, with the registers and other structures in the correct state.

6.1 Implementation

We added a *Transactions* class to the M2V. The *Transactions* class records the write-backs in a particular transaction, and the start and end states of the write-backs for a particular transaction.

Prior to the scheduling pass, M2V determines the registers that must be written back to the Register File (RF). If the transaction model is enabled by the M2V user option, then the *Transactions* class is populated at this point with the registers that need to be written back to the RF. This information is stored in a queue data structure named *writeBacks*, as part of the *Transactions* class. The start address for each transaction, used as VIRPC, is also populated at this point.

There is no change in the scheduling of operations in the transaction model, the only difference being the write-backs to the RF. For a given state, the compiler first determines which transaction is currently being written back. The compiler then iterates through its write-back queue, *regWrVec*, to check if there are any pending write backs for the current transaction, issuing the write-backs in case of a match. This ensures that write-backs within a transaction are performed as soon as the values are available, potentially out-of-order. As part of this scheduling, the starting state and ending state for write-backs of a particular transaction are populated in the *Transactions* class. This is later used in the Verilog generating pass of the compiler as part of the extension state machine.

In the hardware implementation pass, we use a register array in the Extension to maintain the transaction information for a particular state of the Extension. This is generated from the start and end states of the write-backs as populated in the scheduling pass.

The extension state machine performs in the same way without interrupts. In the event of an interrupt, the state machine transitions states only till the end of the current transaction. Once the end of the transaction is identified, the extension state machine stalls and does not perform any more operations/write-backs. This ensures that the registers and memory are in the correct state to resume execution at the next transaction PC address.

7 Hardware Implementation Details

Appendix III lists the complete Verilog code that M2V generates for the basic block in Appendix I. There are four contributions to the final Verilog file: the eMIPS invariant code, the BBW dependent code, the circuit graph dependent code, and the cycle dependent code.

Lines 1-580 of the Appendix are the first lines of eMIPS invariant code in the accelerator definition. Lines 1-340 define the Extension's top-level module, lines 341-465 define the bus macros for the execution-to-write-back interface, and lines 466-580 define the bus macros for the instruction-decode-to-execution interface. Lines 1-340 are simply copied from m2v_mod_bp.v at runtime.

The Extension's top-level module defines the interface signals between the Extension and the rest of the eMIPS design. It contains multiplexor logic for the shared data busses to the register file and the program counter update logic. It also instantiates four modules that make up the core of the Extension: the instruction decode logic, the execution logic, and the two bus macro modules.

The bus macros provide connectivity between the extension logic and the primary eMIPS logic. They represent physical routing locations and are required for partial reconfiguration.

The instruction decode logic defined in lines 581-640 is BBW dependent code. This logic decodes the instruction in parallel with the primary RISC pipeline. If the opcode of the instruction matches the opcode of the Extension, the logic will assert the RI signal so that the extension logic can take control from the RISC pipeline. The fall-through address for the basic block is sent to the program counter. The fields within the instruction are decoded and sent to the execution logic. This version of M2V hardcodes the extension instruction to the MIPS "I" format. The *Circuit.emit_decode* method generates this code.

The extension execution logic is defined in lines 641-1311. The execution logic is composed of invariant code, BBW dependent code, circuit graph dependent code, and cycle dependent code.

Lines 641-722 define the interface signals between the execution logic and the rest of eMIPS. The signals are invariant for every Extension and are copied from m2v_ex_bp.v at runtime.

Lines 722-767 define the Verilog registers that are used later in the execution logic. This code is circuit graph and cycle dependent. The registers for the register node values follow a convention to create an identifiable mapping between the generated logic and the circuit graph. The format is rX_Y[_r], where X is the actual MIPS register, Y is the sequence number of the register node, "_r" indicates that the value comes directly from a register, and the absence of "_r" indicates that the value comes from combinatorial logic. Thus, r9_3 is a combinatorial value for MIPS register 9 that corresponds to the register node with sequence number 3. The *Circuit.emitVarDecl* method generates this code.

The last part of the above Verilog block, from 761-767 is dependent on the transactional model and are present only if the transactional model is enabled.

Lines 770-1077 define the state machines that interface with the register file, the program counter logic and the Memory Management Unit (MMU). This code is invariant and is copied from m2v_state_mc.v at runtime. These state machines are eMIPS-specific.

Lines 1113-1176 define the register file, program counter and memory port usage for each cycle in the Extension. Additionally, if the transactional model is enabled, it contains information regarding the progress of transactions with respect to the cycles in the Extension. This information is used by the state machines defined in lines 770-1077. Lines 800-900 define the memory state machine part of the Extension, as shown in Figure 9. This code is generated by the *Circuit.emitCycState* method.

Lines 1080 – 1113 define the transactional model logic and the state machine used for the transactional model. Methods *Circuit.emitTransactionStateMachine* and *Circuit.emitTransactionLogic* generate this code.

Lines 1176-1216 define the register file interface logic. Since there are limited ports on the register file, the read and write addresses are scheduled onto the register file address lines. Likewise, read data from the register file must be routed to the correct register node, and write data to the register file must come from the correct calculation. The *Circuit.emitRFLogic* method generates this code.

Lines 1220-1240 define the pipeline registers needed by the extension logic. When a calculation must be pipelined, it is latched at the end of the calculation cycle and held for the remainder of the Extension's execution. The *Circuit.emitPipeReg* method generates this code.

Lines 1240-1250 define the logic for generating the address and data out to the memory for the memory instruction in the basic block, if any. The *Circuit.emitMemLogic* method generates this code.

Lines 1250-1290 define the combinatorial logic for the instruction nodes. The *Circuit.emitInstLogic* method generates this code.

Lines 1290-1311 define the primary extension state machine. The state machine is 1-hot encoded with one state representing one cycle in the schedule so the states can be directly used as control signals. The machine is idle until an Extension is successfully decoded and it steps through each state in the Extension. The *Circuit.emitESM* method generates this code.

8 Synthesis Reports

Appendix II contains the synthesis reports for three different versions of the Extension hardware generated for the same example basic block from Appendix I.

Figure A-1 is the synthesis report for the hardware generated by M2V without the support for memory instructions and interrupt handling. This report was generated on an earlier version of eMIPS, with a different data path and a different version of the Xilinx ISE tools. It represents the state of the M2V compiler at the start of the work described herein.

Figure A-2 is the synthesis report for the design generated by the current M2V, with the support for memory instructions but without the interrupt handling capability. The addition to the design is mainly the memory state machine logic. The reports show the effects of the changes in the interfaces to the base data path that have occurred since the initial release of eMIPS. There is an approximate 1% reduction in area utilization due mainly to the simplification of the memory interface. On the other hand, the interface has grown considerably in number of signals, as indicated by the number of IOs parameter.

Figure A-3 is for the hardware generated with the support for both memory operations and interrupts. It can be seen from the statistics that there are extra registers used in the case when the transactional model is enabled. In addition, the percentage of total slices used increases from 3% to 4%.

Overall, the added complexity from interrupts causes a penalty in area utilization. This extra cost is balanced almost exactly by the improvements garnered by the new data path interfaces. A second improvement is shown in the maximum frequency of the design, grown from about 170 MHz to about 210 MHz This is also due to the interface changes.

9 Experimental Results

We tested the changes to the compiler with the example basic block shown in Figure 6. The basic block is a 64-bit division block seen in Doom with an extra memory instruction (load from the stack pointer) inserted to illustrate the working of memory access through the instruction. To test interrupt handling, we generated timer interrupts at short random intervals. The Extension was simulated using ModelSim and the test program simulation was run in Giano. The test program checked over 500 test vectors for the 64-bit division and the test passed successfully in all cases. The Extension always reported the correct Virtual PC (VIRPC) address and the transactional state machine worked as designed. There was no time left in the internship to test on the actual boards.

To test for code coverage, we ran the BBMatch and the M2V compiler on 325 executable files from the code base of the MMLite RTOS [21]. BBMatch extracted and encoded about 150,000 blocks from these files. We then ran the compiler on all the extracted blocks. The results are shown in Table 1.

Table 1: M2V code coverage test results

No. files	325	
No. blocks	146,057	Percent Total
Compiled ok	25,029	17.1%
Warnings	44,800	30.7%
Failure	76,228	52.2%

The large number of failures is actually due in large part to a small number of unsupported instructions, especially JAL, BLTZ, BGEZ, MULT, DIV, SLLV, and SRLV.

10 Conclusions

We have modified the eMIPS tool chain to remove the last remaining obstacles for a fully-automated generation of hardware accelerators. By supporting load and stores, interrupts, and the automatic encoding of extended instructions the compiler can now attack all of the single-block cases of practical applications. Code coverage is currently 50% of the blocks in more than 300 executable files, with only a few unsupported instructions responsible for most of the failures. The addition of interrupt support to the M2V compiler is especially relevant because there is now no limit to the span of an accelerator, even in a general-purpose environment. An arbitrarily long sequence of instructions can be accelerated, without concerns for security or real-time responsiveness.

Support for interrupts in the compiler causes the loss of a little amount of parallelism, because of the in-order write-backs requirement. Using a transactional model mitigates this effect. The ability to perform two write-backs to the register file in every cycle of the Extension further mitigates this effect. The overhead of transactions would be minimal in the case of large basic blocks with a large number of extension states.

In future revisions of M2V, optimization algorithms can be implemented to combine certain transactions to minimize the amount of write-backs and reduce the pressure on the register file.

Support for branches will have to be tested in the next revision of M2V. The framework used for supporting branches should be the same framework used for interrupts. A (conditional) branch can be considered as simply terminating a transaction.

References

[1] Aho, A. V.., Lam, M. S., Sethi, R., Ullman, J. D. *Compilers: Principles, Techniques, and Tools.* Addison Wesley Publishers, Boston, MA. 2007.

[2] Almeida, O., et al. *Embedded Systems Research at DemoFest'07.* Microsoft Research Technical Report MSR-TR-2007-94, July 2007.

[3] Altera Corp. *Excalibur Embedded Processor Solutions*, 2005. .http://www.altera.com/products/devices/excalibur/excindex.html,

[4] Biswas, P., Banerjee, S., Dutt, N., Ienne, P., Pozzi, L. *Performance and Energy Benefits of Instruction Set Extensions in an FPGA Soft Core* VLSID'06, pag. 651-656

[5] Bonzini, P., Pozzi, L. *Code Transformation Strategies for Extensible Embedded Processors* CASES'06, pagg. 242-252.

[6] Forin, A., Lynch, N., L., Pittman, R. N. *eMIPS,A Dynamically Extensible Processor.* Microsoft Research Technical Report MSR-TR-2006-143, October 2006.

[7] Hauck, S. et al. *The Chimaera Reconfigurable Functional Unit*. IEEE VLSI, 2004.

[8] Hauser, J. R., Wawrzynek, J. *Garp: A MIPS Processor with a Reconfigurable Coprocessor*. FCCM'97 pagg 12-21, April 1997.

[9] Hennessy, J. L., Patterson, D.A. *Computer Organization and Design: The Hardware/Software Interface.* Morgan Kaufmann Publishers, San Francisco, CA. 1998.

[10] Kane, G., Heinrich, J. *MIPS RISC Architecture*. Prentice Hall, Upper Saddle River, NJ. 1992.

[11] Kastner, R., Kaplan, A., Ogrenci Memik, S. Bozorgzadeh, E. *Instruction generation for hybrid reconfigurable systems* TODAES vol. 7, no. 4, pagg. 605-632, October 2002.

[12] Lau, D., Pritchard, O., Molson, P. *Automated Generation of Hardware Accelerators with Direct Memory Access from ANSI/ISO Standard C Functions. FCCM'06, pagg. 45-54, April 2006.*

[13] *Stretch, Inc*. http://www.stretchinc.com 2006.

[14] Tensilica, Inc. http://www.tensilica.com, 2006.

[15] Xilinx Inc. *Virtex 4 Family Overview*. Xilinx Inc., June 2005. Available at http://direct.xilinx.com/bvdocs/publications/ds112.pdf

[16] Meier, K., Forin, A. *MIPS-to-Verilog, Hardware Compilation for the eMIPS Processor*, MSR-TR-2007-128, Microsoft Research, WA, September 2007.

[17] Mittal, G., Zaretsky, D.C., Xiaoyong Tang, Banerjee, P. *Automatic translation of software binaries onto FPGAs* Design Automation Conference, 2004. Proceedings. 41st

[18] Mittal, G., Zaretsky, D.C., Xiaoyong Tang, Banerjee, P. *An overview of a compiler for mapping software binaries to hardware* IEEE VLSI, 2007.

[19] Chandrasekhar, V., Forin, A. *Mining Sequential Programs for Coarse-grained Parallelism using Virtualization*, MSR-TR-2008-113, Microsoft Research, WA, August 2008.

[20] Available at http://research.microsoft.com/research/msagl/

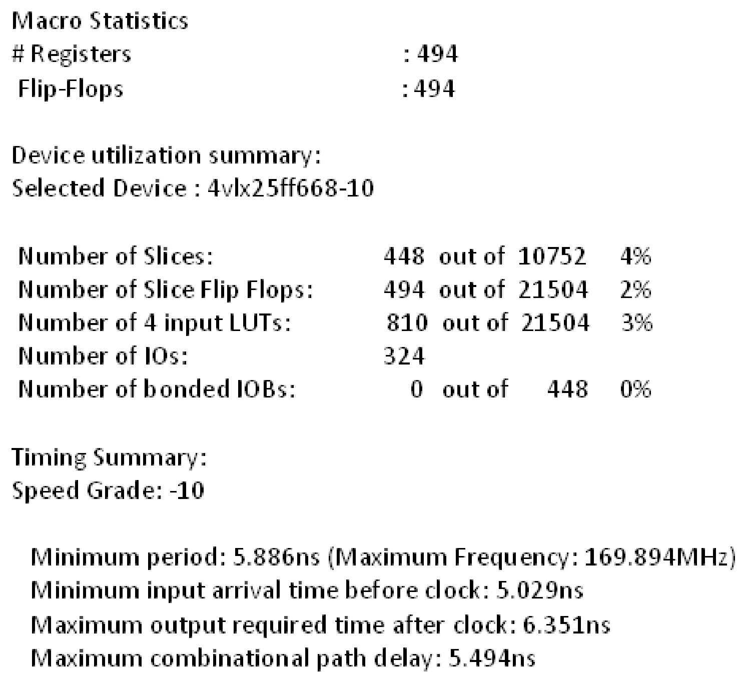
[21] Available at http://research.microsoft.com/invisible/

[22] Available at http://research.microsoft.com/research/EmbeddedSystems/Giano/giano.aspx

Appendix I – BBW File for Example Basic Block

```
[bbname __ull_div]
MIPSBE
[encoding]
[r1=r2+1;r3=r0+11;r5=r4+1;r6=r5+1]b26.6:c011110;b21.5:r4;b16.5:r2;b0.16:v0;
[code 48]
40080100
c21f0200
25082300
40100200
c21f0400
25104300
40200400
c21f0500
25208300
2b182600
5000310
40280500
[disasm]
sll  r1,r1,1
srl  r3,r2,31
or  r1,r1,r3
sll  r2,r2,1
srl  r3,r4,31
or  r2,r2,r3
sll  r4,r4,1
srl  r3,r5,31
or  r4,r4,r3
sltu   r3,r1,r6
beq   r0,r3,40
sll  r5,r5,1
[registers 7]
0,9,8,11,4,5,6
[valuess 1]
{40,11,5}
```

Appendix II - Synthesis reports for the Generated Extension hardware



```
Macro Statistics
# Registers                   : 494
 Flip-Flops                   : 494

Device utilization summary:
Selected Device : 4vlx25ff668-10

 Number of Slices:             448  out of  10752    4%
 Number of Slice Flip Flops:   494  out of  21504    2%
 Number of 4 input LUTs:       810  out of  21504    3%
 Number of IOs:                324
 Number of bonded IOBs:          0  out of    448    0%

Timing Summary:
Speed Grade: -10

   Minimum period: 5.886ns (Maximum Frequency: 169.894MHz)
   Minimum input arrival time before clock: 5.029ns
   Maximum output required time after clock: 6.351ns
   Maximum combinational path delay: 5.494ns
```

Figure A-1. Synthesis report for the Verilog code generated from version 1 of M2V

```
Macro Statistics
# Registers                   : 489
 Flip-Flops                   : 489

Device utilization summary:
---------------------------

Selected Device : 4vlx25ff668-10

 Number of Slices:             366  out of  10752    3%
 Number of Slice Flip Flops:   489  out of  21504    2%
 Number of 4 input LUTs:       653  out of  21504    3%
 Number of IOs:                558
 Number of bonded IOBs:        0  out of    448    0%

Timing Summary:
---------------
Speed Grade: -10

   Minimum period: 4.864ns (Maximum Frequency:        205.579MHz)
   Minimum input arrival time before clock:           4.189ns
   Maximum output required time after clock:          6.486ns
   Maximum combinational path delay:                  5.811ns
```

Figure A-2. Synthesis report for the Verilog code generated from M2V with the memory state machine

```
Macro Statistics
# Registers                          : 496
 Flip-Flops                          : 496

Device utilization summary:
---------------------------

Selected Device : 4vlx25ff668-10

 Number of Slices:                450  out of  10752     4%
 Number of Slice Flip Flops:      496  out of  21504     2%
 Number of 4 input LUTs:          796  out of  21504     3%
 Number of IOs:                   558
 Number of bonded IOBs:           0  out of    448     0%

Timing Summary:
---------------
Speed Grade: -10

Minimum period: 4.739ns (Maximum Frequency: 211.006MHz)
Minimum input arrival time before clock:    4.064ns
Maximum output required time after clock:   7.037ns
Maximum combinational path delay:           6.362ns
```

Figure A-3. Synthesis report for the Verilog code generated from M2V with the Transactional model for interrupt support enabled

Appendix III – Verilog Output for the Example Basic Block

```
// a.v
// auto-generated by m2v revision 1 on Fri Aug 15 15:46:50 2008
//
// INFO: reading from m2v_mod_bp.v
//
// m2v_mod_bp.v
// 8/15/07
// Karl Meier, Neil Pittman
//
// MIPS to Verilog (m2v) module (_mod) boilerplate (_bp)
//
// Copyright (c) Microsoft Corporation. All rights reserved.

`timescale 1ns / 1ps

module mmlite_div64 (
/*****Ports*************************************************************/
  /* INPUT PORTS */
  input      CLK,       /* System Clock 50 - 100 MHZ */
  input      EN,      /* Enable */
  input      EXCEXT,      /* Exception Flush */
  input      EXTNOP_MA,     /* Extension Bubble in Memory Access Phase */
  input      GR,      /* Grant Pipeline Resources */
  input [31:0] INSTR,      /* Current Instruction */
  input [31:0] PC,       /* Current PC External */
  input      PCLK,       /* Pipeline Clock */
  input [31:0] RDREG1DATA,    /* Register Read Port 1 Register Data */
  input [31:0] RDREG2DATA,    /* Register Read Port 2 Register Data */
  input [31:0] RDREG3DATA,    /* Register Read Port 3 Register Data */
  input [31:0] RDREG4DATA,    /* Register Read Port 4 Register Data */
  input      REGEMPTY,    /* Register Write Buffer Empty */
  input      REGFULL,     /* Register Write Buffer Full */
  input      REGRDY,      /* Register Write Buffer Ready */
  input      RESET,     /* System Reset */
  input [31:0] MDATA_IN,   /* Memory Data In */
                  /* Multiplexed: */
                  /*  Memory Data In */
                  /*  Peripheral Memory Data In */
                  /*  Memory Data Monitor */
  input     MDATA_VLD_IN,  /* Memory Data Valid */

  /* OUTPUT PORTS */
  output     ACK,       /* Enable Acknowledged */
  output [31:0]  EXTADD,      /* Extension Address */
                    /* Multiplexed: */
                    /*  Next PC */
                    /*  Exception Address */
                    /*  PC Memory Access Phase */
  output     PCNEXT,      /* Conditional PC Update */
  output [4:0] RDREG1,      /* Register Read Port 1 Register Number */
                    /* Multiplexed: */
                    /*  Register Read Port 1 Register Number */
                    /*  Register Write Port 1 Register Number */
                    /*  Write Register Memory Access Phase */
  output [4:0] RDREG2,      /* Register Read Port 2 Register Number */
                    /* Multiplexed: */
                    /*  Register Read Port 2 Register Number */
                    /*  Register Write Port 2 Register Number */
                    /*   <0> Register Write Enable Memory Access Phase */
                    /*  <1> Memory to Register Memory Acess Phase */
  output [4:0] RDREG3,      /* Register Read Port 3 Register Number */
                    /* Multiplexed: */
                    /*  Register Read Port 3 Register Number */
  output [4:0] RDREG4,      /* Register Read Port 4 Register Number Internal */
```

```
                    /* Multiplexed: */
                    /*  Register Read Port 4 Register Number */
                    /*  <1:0> Data Address [1:0] Memory Access Phase */
                    /*    <2> Right/Left Unaligned Load/Store Memory Access Phase */
                    /*  <3> Byte/Halfword Load/Store Memory Acess Phase */
    output     REGWRITE1,     /* Register Write Port 1 Write Enable */
    output     REGWRITE2,     /* Register Write Port 2 Write Enable */
    output     REWB,        /* Re-enter at Writeback */
    output     RI,       /* Reserved/Recognized Instruction */
    output [31:0]  WRDATA1,     /* Register Write Port 1 Data Internal */
                    /* Multiplexed: */
                    /*  Register Write Port 1 Data */
                    /*  ALU Result Memory Access Phase */
    output [31:0]  WRDATA2,     /* Register Write Port 2 Data Internal */
                    /* Multiplexed: */
                    /*  Register Write Port 2 Data */
                    /*  Memory Data Out Memory Access Phase */
    output    BLS_OUT,    /* Byte Load/Store */
    output    HLS_OUT,    /* Halfword Load/Store */
    output    RNL_OUT,    /* Memory Right/Left Unaligned Load/Store */
    output [31:0]  MADDR_OUT,   /* Memory Address */
    output [31:0]  MDATA_OUT,   /* Memory Data Out */
                  /* Multiplexed: */
                  /*  Memory Data Out */
                  /*  Peripheral Memory Data Out */
    output    MOE_OUT,    /* Memory Output Enable */
    output    MWE_OUT     /* Memory Write Enable */
    );

  /*****Signals*****************************************************************/

    wire [31:0] ALURESULT_WB;    /* ALU Result to Writeback Phase */
    wire      BHLS_WB;     /* Byte/Halfword Load/Store to Writeback Phase */
    wire [31:0] CJMPADD;     /* Conditional Jump address to offset from Current PC */
    wire [15:0] DIMM_EX;     /* Data Immediate Execute Phase */
    wire [15:0] DIMM_ID;     /* Data Immediate Instruction Decode Phase */
    wire [1:0]  DMADD_WB;    /* Least Significant Bits of Data Address to Writeback Phase */
    wire [31:0] DMDATAOUT_WB;    /* Memory Data Out to Writeback Phase */
    wire      DNE;        /* Execution Done */
    wire      EN_EX;      /* Enable Execute Phase */
    wire [31:0] JMPADD;      /* Jump address to end of basic block */
    wire      MEMTOREG_WB;   /* Memory to Register to Writeback Phase */
    wire [31:0] PC_EX;      /* PC Execute Phase */
    wire [31:0] PC_WB;      /* PC to Writeback Phase */
    wire [4:0]  RD_EX;      /* Destination Register Execution Phase */
    wire [4:0]  RDREG1_EX;     /* Register Read Port 1 Register Number Execute Phase */
    wire [31:0] RDREG1DATA_EX;   /* Register Read Port 1 Register Data Execute Phase */
    wire [4:0]  RDREG2_EX;     /* Register Read Port 2 Register Number Execute Phase */
    wire [31:0] RDREG2DATA_EX;   /* Register Read Port 2 Register Data Execute Phase */
    wire [4:0]  RDREG3_EX;     /* Register Read Port 3 Register Number Execute Phase */
    wire [4:0]  RDREG4_EX;     /* Register Read Port 4 Register Number Execute Phase */
    wire    REGWRITE_EX;   /* Register Write Execute Phase */
    wire    REGWRITE_ID;   /* Register Write Instruction Decode Phase */
    wire    REGWRITE_WB;   /* Register Write to Writeback Phase */
    wire    RESET_EX;    /* Reset Execute Phase */
    wire [31:0] RESULT_EX;     /* Result Execution Phase */
    wire    RNL_WB;      /* Right/Left Unaligned Load/Store to Writeback Phase */
    wire [4:0]  RS_EX;      /* Operand Register 1 Execute Phase */
    wire [4:0]  RS_ID;      /* Operand Register 1 Instruction Decode Phase */
    wire [4:0]  RT_EX;      /* Operand Register 2 Execute Phase */
    wire [4:0]  RT_ID;      /* Operand Register 2 Instruction Decode Phase */
    wire      SLL128_EX;     /* Shift Left Logical 128 bits Execute Phase */
    wire      SLL128_ID;     /* Shift Left Logical 128 bits Instruction Decode Phase */
    wire [31:0] WRDATA1_EX;    /* Register Write Port 1 Data Execute Phase */
    wire [31:0] WRDATA2_EX;    /* Register Write Port 2 Data Execute Phase */
    wire [4:0]  WRREG_WB;    /* Write Register Number to Writeback Phase */
    wire [4:0]  WRREG1_EX;     /* Register Write Port 1 Register Number Execute Phase */
    wire [4:0] WRREG2_EX;     /* Register Write Port 2 Register Number Execute Phase */
    wire [31:0] VIRPC;

  /*****Registers****************************************************************/

    reg en_reg; /* Enable */
    reg gr_reg; /* Grant Pipeline Resources */

  /*****Initialization***********************************************************/
  /*
    initial
    begin
      en_reg = 1'b0;
      gr_reg = 1'b0;
    end
  */

  /***********************************************************************/

    assign EXTADD  = (EXCEXT)?       VIRPC:
              (en_reg)?       JMPADD:
              (PCNEXT)?       CJMPADD:
              (REWB)?         PC_WB:
                           32'hffffffff;
      /*
       * The rest cannot be zero'ed out as the extension state machine
       * might still have to be completed till a particular transaction ends
       */
    assign RDREG1  = (gr_reg & REGWRITE1)?WRREG1_EX:
                //(EXCEXT)?       5'b0:
              (REWB & gr_reg)?   WRREG_WB:
              (gr_reg)?       RDREG1_EX:
                            5'b11111;
    assign RDREG2  = (gr_reg & REGWRITE2)?WRREG2_EX:
                //(EXCEXT)?       5'b0:
              (REWB & gr_reg)?   {3'b0,MEMTOREG_WB,REGWRITE_WB}:
              (gr_reg)?       RDREG2_EX:
                            5'b11111;
    assign RDREG3  = (REWB & gr_reg)?    5'b0:
                //(EXCEXT)?       5'b0:
              (gr_reg)?       RDREG3_EX:
                            5'b11111;
    assign RDREG4  = (REWB & gr_reg)?    {1'b0,BHLS_WB,RNL_WB,DMADD_WB}:
                //(EXCEXT)?       5'b0:
              (gr_reg)?       RDREG4_EX:
                            5'b11111;
    assign WRDATA1 = (gr_reg & REGWRITE1)? WRDATA1_EX:
                 //(EXCEXT)?        32'b0:
               (REWB)?        ALURESULT_WB:
                                      32'hffffffff;
    assign WRDATA2 = (gr_reg & REGWRITE2)? WRDATA2_EX:
                 //(EXCEXT)?        32'b0:
               (REWB)?        DMDATAOUT_WB:
                                32'hffffffff;


      //
      // instantiate the instruction decode module for the extension instruction
      //   - the instruction decode module is auto generated and appended to the
      //     end of the verilog file (a.v unless redefined)
      //

    ext_id id (
      .CLK(CLK),
      .DIMM(DIMM_ID),
      .EN(EN),
      .JMPADD(JMPADD),
      .INSTR(INSTR),
      .PC(PC),
      .REGWRITE(REGWRITE_ID),
      .RESET(RESET),
      .RI(RI),
      .RS(RS_ID),
      .RT(RT_ID),
      .SLL128(SLL128_ID)
      );

/*****Instruction Decode -> Execute*****************************************************************/

   mmldiv64_toex to_ex(
      .ACK(ACK),
      .CLK(CLK),
      .DIMM_EX(DIMM_EX),
      .DIMM_ID(DIMM_ID),
      .EN_EX(EN_EX),
      .EN_ID(EN),
      .EXCEXT(EXCEXT),
      .PC_EX(PC_EX),
      .PC_ID(PC),
      .PCLK(PCLK),
      .RDREG1DATA_EX(RDREG1DATA_EX),
      .RDREG1DATA_ID(RDREG1DATA),
      .RDREG2DATA_EX(RDREG2DATA_EX),
      .RDREG2DATA_ID(RDREG2DATA),
      .REGWRITE_EX(REGWRITE_EX),
      .REGWRITE_ID(REGWRITE_ID),
      .RESET(RESET),
      .RESET_EX(RESET_EX),
      .RS_EX(RS_EX),
      .RS_ID(RS_ID),
      .RT_EX(RT_EX),
      .RT_ID(RT_ID),
      .SLL128_ID(SLL128_ID),
      .SLL128_EX(SLL128_EX)
      );


      //
      // instantiate the execution module for the extension instruction
      //   - the execution module is auto generated and appended to the
      //     end of the verilog file (a.v unless redefined)
      //

   ext_ex ex(
      .ACK(ACK),
      .DIMM(DIMM_EX),
      .DNE(DNE),
      .CLK(CLK),
      .CJMPADD(CJMPADD),
      .EN(EN_EX),
      .EXTNOP_MA(EXTNOP_MA),
      .GR(GR),
      .PC(PC_EX),
      .PCLK(PCLK),
      .PCNEXT(PCNEXT),
      .RD(RD_EX),
      .RDREG1(RDREG1_EX),
      .RDREG1DATA(RDREG1DATA),
      .RDREG1DATA_ID(RDREG1DATA_EX),
      .RDREG2(RDREG2_EX),
      .RDREG2DATA(RDREG2DATA),
      .RDREG2DATA_ID(RDREG2DATA_EX),
      .RDREG3(RDREG3_EX),
      .RDREG3DATA(RDREG3DATA),
      .RDREG4(RDREG4_EX),
      .RDREG4DATA(RDREG4DATA),
      .REGEMPTY(REGEMPTY),
      .REGFULL(REGFULL),
      .REGRDY(REGRDY),
      .REGWRITE1(REGWRITE1),
      .REGWRITE2(REGWRITE2),
      .RESET(RESET_EX),
      .RESULT(RESULT_EX),
      .RS(RS_EX),
      .RT(RT_EX),
      .SLL128(SLL128_EX),
      .WRDATA1(WRDATA1_EX),
      .WRDATA2(WRDATA2_EX),
      .WRREG1(WRREG1_EX),
      .WRREG2(WRREG2_EX),
      .MDATA_IN(MDATA_IN),
      .MDATA_VLD_IN(MDATA_VLD_IN),
      .BLS_OUT(BLS_OUT),
      .HLS_OUT(HLS_OUT),
      .RNL_OUT(RNL_OUT),
      .MOE_OUT(MOE_OUT),
      .MWE_OUT(MWE_OUT),
      .MADDR_OUT(MADDR_OUT),
      .MDATA_OUT(MDATA_OUT),
      .EXCEXT(EXCEXT),
      .VIRPC(VIRPC)
      );

   /*****Execute -> to Writeback**************************************************************/

     mmldiv64_topipe_wb to_wb(
      .ACK(ACK),
      .ALURESULT_WB(ALURESULT_WB),
      .BHLS_WB(BHLS_WB),
      .CLK(CLK),
      .DMADD_WB(DMADD_WB),
      .DMDATAOUT_WB(DMDATAOUT_WB),
      .DNE(DNE),
      .EN_EX(EN_EX),
      .EXCEXT(EXCEXT),
      .EXTNOP_MA(EXTNOP_MA),
      .PC_EX(PC_EX),
      .PC_WB(PC_WB),
      .PCLK(PCLK),
      .MEMTOREG_WB(MEMTOREG_WB),
      .RD_EX(RD_EX),
      .REGWRITE_EX(REGWRITE_EX),
      .REGWRITE_WB(REGWRITE_WB),
      .RESET(RESET),
      .RESULT_EX(RESULT_EX),
      .REWB(REWB),
      .RNL_WB(RNL_WB),
      .WRREG_WB(WRREG_WB)
      );

   /**********************************************************************/

     always@(posedge CLK)
     begin
      if (RESET == 1'b0)
      begin
        en_reg <= 1'b0;
        gr_reg <= 1'b0;
      end
      else
      begin
```

```
      en_reg <= EN;
      gr_reg <= GR;
    end
  end

endmodule



/*****Execute -> to Writeback***************************************************************/

module mmldiv64_topipe_wb(
/*****Ports*************************************************************/
  /* INPUT PORTS */
  input     ACK,      /* Enable Acknowledged */
  input     CLK,      /* System Clock 50 - 100 MHZ */
  input     DNE,      /* Execution Done */
  input     EN_EX,    /* Enable Execute Phase */
  input     EXCEXT,    /* Exception Flush */
  input     EXTNOP_MA,   /* Extension Bubble in Memory Access Phase */
  input [31:0] PC_EX,    /* Current PC Execute Phase */
  input     PCLK,      /* Pipeline Clock */
  input [4:0]  RD_EX,    /* Destination Register Execution Phase */
  input       REGWRITE_EX, /* Register Write Execute Phase */
  input     RESET,    /* System Reset */
  input [31:0] RESULT_EX,   /* Result Execution Phase */
  /* OUTPUT PORTS */
  output [31:0]  ALURESULT_WB,  /* ALU Result to Writeback Phase */
  output      BHLS_WB,    /* Byte/Halfword Load/Store to Writeback Phase */
  output [1:0] DMADD_WB,  /* Least Significant Bits of Data Address to Writeback Phase */
  output [31:0]  DMDATAOUT_WB,  /* Memory Data Out to Writeback Phase */
  output      MEMTOREG_WB, /* Memory to Register to Writeback Phase */
  output [31:0]  PC_WB,    /* Current PC to Writeback Phase */
  output      REGWRITE_WB, /* Register Write to Writeback Phase */
  output      REWB,      /* Re-enter at Writeback */
  output      RNL_WB,    /* Right/Left Unaligned Load/Store to Writeback Phase */
  output [4:0] WRREG_WB    /* Write Register Number to Writeback Phase */
  );

/*****Signals***********************************************************/

  wire EN_WB;  /* Enable to Writeback Phase */
  wire RESET_WB; /* Reset to Writeback Phase */

/*****Registers*********************************************************/

  reg [70:0] ex_wb;   /* Execute -> to Writeback Pipeline Register */
  reg [1:0] pclkcnt;  /* Pipeline Clock edge detection */
  reg reset_reg;    /* Reset to Writeback Phase */
  reg rewb_reg;     /* Re-enter at Writeback */

/*****Initialization****************************************************/
/*
  initial
  begin
    ex_wb = 71'b0;
    pclkcnt = 2'b0;
    rewb_reg = 1'b0;
    reset_reg = 1'b0;
  end
*/
/*********************************************************************/

  assign RESET_WB   = reset_reg;
  assign REWB     =  rewb_reg & EN_WB;
  assign EN_WB      = ex_wb[70];   //EN_EX;
  assign REGWRITE_WB  = ex_wb[69];   //REGWRITE_EX;
  assign MEMTOREG_WB  = 1'b0;
  assign RNL_WB     = 1'b0;
  assign BHLS_WB    = 1'b0;
  assign DMADD_WB   = 2'b0;
  assign WRREG_WB   = ex_wb[68:64];  //RD_EX;
  assign ALURESULT_WB = ex_wb[63:32];  //RESULT_EX;
  assign DMDATAOUT_WB  = 32'b0;
  assign PC_WB    = ex_wb[31:0]; //PC_EX;

/*********************************************************************/

  always@(posedge CLK)
  begin
    /* Pipeline Clock edge detection */
    pclkcnt = {pclkcnt[0],PCLK};   // karl, 9/19, change to non-blocking to
                                   // match Neil
  end

  always@(posedge CLK)
  begin
    case(pclkcnt)
      2'b01   : begin
            /* Synchronize Reset to Pipeline Clock */
            reset_reg <= RESET;
          end
      default : begin
          end
    endcase
  end

  always@(posedge CLK)
  begin
    /* Execute -> to Memory Access Pipeline Register */
    casex({pclkcnt,RESET_WB,EXTNOP_MA,rewb_reg,ACK,DNE,EXCEXT})
      8'bxx0xxxxx : begin
              /* Reset */
              rewb_reg <= 1'b0;
              ex_wb <= 71'b0;
            end
      8'b011xxxx1 : begin
              /* Exception in Pipeline, Flush */
              rewb_reg <= 1'b0;
              ex_wb <= 71'b0;
            end
      8'bxx1x0110 : begin
              /* Latch Data and Control after Execution Finishes */
              ex_wb <= {EN_EX,REGWRITE_EX,RD_EX,RESULT_EX,PC_EX};
            end
      8'b101100x0 : begin
              /* Raise REWB at next Negedge of PCLK after ACK Lowers */
              rewb_reg <= 1'b1;
            end
      8'b011x1xx0 : begin
              /* Lower REWB at next Posedge and reset register */
              rewb_reg <= 1'b0;
              ex_wb <= 71'b0;
            end
      default :  begin
              /* NOP */
            end
    endcase
  end
endmodule


/*****Instruction Decode -> Execute*****************************************************************/

```

```
module mmldiv64_toex(
/*****Ports*****************************************************************/
  /* INPUT PORTS */
  input     ACK,      /* Enable Acknowledged */
  input     CLK,      /* System Clock 50 - 100 MHZ */
  input [15:0] DIMM_ID,    /* Data Immediate Instruction Decode Phase */
  input     EN_ID,    /* Enable Instruction Decode Phase */
  input     EXCEXT,    /* Exception Flush */
  input [31:0] PC_ID,    /* Current PC Decode Phase */
  input     PCLK,      /* Pipeline Clock */
  input [31:0] RDREG1DATA_ID, /* Register Read Port 1 Register Data Instruction Decode Phase */
  input [31:0] RDREG2DATA_ID, /* Register Read Port 2 Register Data Instruction Decode Phase */
  input     REGWRITE_ID, /* Register Write Instruction Decode Phase*/
  input     RESET,    /* System Reset */
  input [4:0]  RS_ID,    /* Operand Register 1 Instruction Decode Phase */
  input [4:0]  RT_ID,    /* Operand Register 2 Instruction Decode Phase */
  input     SLL128_ID,   /* Shift Left Logical 128 bits Instruction Decode Phase */
  /* OUTPUT PORTS */
  output [15:0]  DIMM_EX,    /* Data Immediate Execute Phase */
  output      EN_EX,    /* Enable Execute Phase */
  output [31:0]  PC_EX,    /* Current PC Instruction Decode Phase */
  output [31:0]  RDREG1DATA_EX, /* Register Read Port 1 Register Data Execute Phase */
  output [31:0]  RDREG2DATA_EX, /* Register Read Port 2 Register Data Execute Phase */
  output      REGWRITE_EX, /* Register Write Execute Phase*/
  output      RESET_EX,  /* Reset Execute Phase */
  output [4:0] RS_EX,    /* Operand Register 1 Execute Phase */
  output [4:0] RT_EX,    /* Operand Register 2 Execute Phase */
  output      SLL128_EX  /* Shift Left Logical 128 bits Execute Phase */
  );

/*****Registers*****************************************************************/

  reg [124:0] id_ex;  /* Instruction Decode -> Execute Pipeline Register */
  reg [1:0] pclkcnt;  /* Pipeline Clock edge detection */
  reg    reset_reg; /* Reset Execute Phase */

/*****Initialization*****************************************************************/

/*
  initial
  begin
    id_ex = 125'b0;
    pclkcnt = 2'b0;
    reset_reg = 1'b0;
  end
*/

/**********************************************************************/

  assign RESET_EX   = reset_reg;
  assign EN_EX    = id_ex[124];    //EN_ID;
  assign SLL128_EX  = id_ex[123];    //SLL128_ID;
  assign REGWRITE_EX  = id_ex[122];    //REGWRITE_ID;
  assign RS_EX      = id_ex[121:117];  //RS_ID;
  assign RT_EX      = id_ex[116:112];  //RT_ID;
  assign DIMM_EX    = id_ex[111:96];   //DIMM_ID;
  assign PC_EX    = id_ex[95:64];    //PC_ID;
  assign RDREG1DATA_EX = id_ex[63:32];    //RDREG1DATA_ID;
  assign RDREG2DATA_EX = id_ex[31:0];    //RDREG2DATA_ID

/**********************************************************************/

  always@(posedge CLK)
  begin
    /* Pipeline Clock edge detection */
    pclkcnt = {pclkcnt[0],PCLK};   // karl, 9/19, change to non-blocking to
                                   // match Neil
  end

  always@(posedge CLK)
  begin
    case(pclkcnt)
      2'b01   : begin
              /* Synchronize Reset to Pipeline Clock */
              reset_reg <= RESET;
            end
      default : begin
            end
    endcase
  end

  always@(posedge CLK)
  begin
    /* Instruction Decode -> Execute Pipeline Register */
    casex({pclkcnt,RESET_EX,ACK,EXCEXT})
      5'bxx0xx: begin
              /* Reset */
              id_ex <= 109'b0;
            end
      5'b011x1: begin
              /* Exception in Pipeline, Flush */
              id_ex <= 109'b0;
            end
      5'bxx110: begin
              /* Hold state during Execute Phase */
            end
      5'b01100: begin
              /* Clocking the Pipeline */
              id_ex <= {EN_ID,SLL128_ID,REGWRITE_ID,RS_ID,RT_ID,DIMM_ID,PC_ID,RDREG1DATA_ID,RDREG2DATA_ID};
            end
      default : begin
              /* NOP */
            end
    endcase
  end
endmodule


//
// INFO: finished reading from m2v_mod_bp.v
//

//
// extension instruction decode
//
module ext_id(
   input         CLK,
   input         EN,
   input [31:0]  INSTR,
   input [31:0]  PC,
   input         RESET,

   output reg [15:0] DIMM,
   output reg [31:0] JMPADD,
   output reg        REGWRITE,
   output reg        RI,
   output reg [4:0]  RS,
   output reg [4:0]  RT,
   output reg        SLL128
   );

   reg [31:0] jmpadd_c;
   reg en_r;
   reg [5:0] op_r;
   reg [31:0] pc_r;
   reg opcode_match;

   // combinatorial logic for instruction decode
   always @ (*) begin
      jmpadd_c = pc_r + 48 + 4;
      opcode_match = (op_r == 30);
   end

   // sequential logic for instruction decode
   always @ (posedge CLK) begin
      if (!RESET) begin
         DIMM     <= 16'h0;
         op_r     <= 6'h0;
         RS       <= 5'h0;
         RT       <= 5'h0;
         en_r     <= 1'h0;
         pc_r     <= 32'h0;
         JMPADD   <= 32'h0;
         RI       <= 1'h1;
         SLL128   <= 1'h0;
         REGWRITE <= 1'h0;
      end else begin
         DIMM     <= INSTR[15:0];
         op_r     <= INSTR[31:26];
         RS       <= INSTR[25:21];
         RT       <= INSTR[20:16];
         en_r     <= EN;
         pc_r     <= PC;
         JMPADD   <= jmpadd_c;
         RI       <= ~opcode_match;
         SLL128   <= en_r & opcode_match;
         REGWRITE <= en_r & opcode_match;
      end
   end
endmodule

//
// INFO: reading from m2v_ex_bp.v
//
// m2v_ex_bp.v
// 8/15/07
// Karl Meier, Neil Pittman
//
// MIPS to Verilog (m2v) execution (_ex) boilerplate (_bp)
//
// Copyright (c) Microsoft Corporation. All rights reserved.
//

module ext_ex (
/*****Ports****************************************************************/
  /* INPUT PORTS */
  input     CLK,        /* System Clock 50 - 100 MHZ */
  input [15:0] DIMM,        /* Data Immediate */
  input     EN,       /* Enable  */
  input     EXTNOP_MA,     /* Extension Bubble in Memory Access Phase */
  input     GR,       /* Grant Pipeline Resources */
  input [31:0] PC,       /* Current PC */
  input     PCLK,        /* Pipeline Clock */
  input [31:0] RDREG1DATA,    /* Register Read Port 1 Register Data */
  input [31:0] RDREG1DATA_ID,  /* Register Read Port 1 Register Data Instruction Decode Phase */
  input [31:0] RDREG2DATA,    /* Register Read Port 2 Register Data */
  input [31:0] RDREG2DATA_ID,  /* Register Read Port 2 Register Data Instruction Decode Phase */
  input [31:0] RDREG3DATA,    /* Register Read Port 3 Register Data */
  input [31:0] RDREG4DATA,    /* Register Read Port 4 Register Data */
  input     REGEMPTY,    /* Register Write Buffer Empty */
  input     REGFULL,     /* Register Write Buffer Full */
```

```
    input     REGRDY,      /* Register Write Buffer Ready */
    input     RESET,      /* System Reset */
    input [4:0]  RS,       /* Operand Register 1 */
    input [4:0]  RT,       /* Operand Register 2 */
    input     SLL128,      /* Shift Left Logical 128 bits */
    input [31:0] MDATA_IN,   /* Memory Data In */
                    /* Multiplexed: */
                    /* Memory Data In */
                    /* Peripheral Memory Data In */
                    /* Memory Data Monitor */
    input     MDATA_VLD_IN, /* Memory Data Valid */
    input    EXCEXT,       /* Exception Signal */

    /* OUTPUT PORTS */
    output reg     ACK,      /* Enable Acknowledged */
    output reg [31:0] CJMPADD,     /* Conditional Jump address to offset from Current PC */
    output reg     DNE,      /* Execution Done */
    output reg     PCNEXT,     /* Conditional PC Update */
    output reg [4:0]  RD,     /* Destination Register */
    output reg     REGWRITE1,   /* Register Write Port 1 Write Enable */
    output reg     REGWRITE2,   /* Register Write Port 2 Write Enable */
    output reg [4:0]  RDREG1,     /* Register Read Port 1 Register Number */
    output reg [4:0]  RDREG2,     /* Register Read Port 2 Register Number */
    output reg [4:0]  RDREG3,     /* Register Read Port 3 Register Number */
    output reg [4:0]  RDREG4,     /* Register Read Port 4 Register Number */
    output reg [31:0] RESULT,     /* Result */
    output reg [31:0] WRDATA1,     /* Register Write Port 1 Data */
    output reg [31:0] WRDATA2,     /* Register Write Port 2 Data */
    output reg [4:0]  WRREG1,     /* Register Write Port 1 Register Number */
    output reg [4:0]  WRREG2,     /* Register Write Port 2 Register Number */
    output reg BLS_OUT,    /* Byte Load/Store */
    output reg  HLS_OUT,    /* Halfword Load/Store */
    output reg RNL_OUT,    /* Memory Right/Left Unaligned Load/Store */
    output reg [31:0] MADDR_OUT,   /* Memory Address */
    output reg [31:0] MDATA_OUT,   /* Memory Data Out */
                    /* Multiplexed: */
                    /* Memory Data Out */
                    /* Peripheral Memory Data Out */
    output reg MOE_OUT,    /* Memory Output Enable */
    output reg MWE_OUT,    /* Memory Write Enable */
    output reg [31:0] VIRPC   /* Virtual PC for interrupt support */
    );

      // tie off outputs that are not used in the automated accelerator
      always @ (posedge CLK) begin
         RD <= 0;
         RESULT <= 0;
      end

  /*********************************************************************/

  //
  // INFO: finished reading from m2v_ex_bp.v
  //

      // parameters for extension execution block
      parameter MAX_STATE = 8;
      parameter REG_READ_WAIT_STATES = 5;

      // declarations for extension state machine
      reg[MAX_STATE:1] state_r;
      reg[7:1] branch_state_r;
      reg[7:1] write_state_r;
      reg[7:1] read_state_r;

      // declarations for the extension memory state machine
      reg[7:1] mem_write_state_r;
      reg[7:1] mem_read_state_r;

      // declarations for memory data in variable
      reg[31:0] mdata_in;

      // declarations for register read variables
      reg[31:0] r9_1;
      reg[31:0] r8_4, r8_4_r;
      reg[31:0] r4_11, r4_11_r;
      reg[31:0] r5_18;
      reg[31:0] r6_23;
      // declarations for register temp variables
      reg[31:0] r9_3;
      reg[31:0] r11_6;
      reg[31:0] r9_8, r9_8_r;
      reg[31:0] r8_10;
      reg[31:0] r11_13, r11_13_r;
      reg[31:0] r8_15, r8_15_r;
      reg[31:0] r4_17, r4_17_r;
      reg[31:0] r11_20;
      reg[31:0] r4_22, r4_22_r;
      reg[31:0] r11_25, r11_25_r;
      reg[31:0] r5_29, r5_29_r;
      // declarations for memory address temp variables

      // declarations for transaction model support
      reg tran_state_done;
      reg [31:0] virpc_tr0, virpc_tr1, virpc_tr2;
      reg [7:1] tran_end_state_r;
      reg transaction_end_this_state;

  //
  // INFO: reading from m2v_state_mc.v
  //
      // m2v_state_mc.v
      //
      // Karl Meier
      // 8/15/07
      //
      // invariant state machine logic for the read, write, and branch state
      // machines
      //

      reg [1:0] pclk_del_r;
      reg pclk_rise, pclk_fall;
      reg en_r, sll128_r, gr_r, regrdy_r, regfull_r, regempty_r, extnop_ma_r;
      reg clr_dne, DNE_c, ACK_c;
      reg done_state, done_state_r;
      reg wsm_idle, wsm_idle_r, wsm_pulse, wsm_pulse_r, wsm_wait, wsm_wait_r;
      reg write_this_state, wsm_done;
      reg rsm_idle, rsm_idle_r, rsm_latch, rsm_latch_r;
      reg rsm_wait, rsm_wait_r, rsm_wait2, rsm_wait2_r;
      reg [3:0] rsm_count, rsm_count_r;
      reg read_this_state, rsm_done;
      reg bsm_idle, bsm_idle_r, bsm_calc, bsm_calc_r;
      reg bsm_wait, bsm_wait_r, bsm_waitpf, bsm_waitpf_r;
      reg bsm_waitpr, bsm_waitpr_r;
      reg branch_this_state, bsm_done;
      reg fsm_idle, fsm_idle_r, fsm_wait2, fsm_wait2_r, fsm_wait, fsm_wait_r;
      reg final_state, fsm_done;
      reg take_branch, take_branch_r;

      reg [1:0] mdata_vld_r;
      reg mdata_vld_rise, mdata_vld_fall;
      reg mem_read_this_state, mrs_done;
      reg mem_write_this_state, mws_done;
      reg [1:0] mem_this_state, mdne, mdne_c;

     // State machine logic for MMU access instructions
     // memory access control for the extension
     always @ (*) begin
          mdata_vld_rise = (mdata_vld_r == 2'b01);
          mdata_vld_fall = (mdata_vld_r == 2'b10);

        // Place memory read/write request on rising edge of PCLK
        casex ({RESET, mem_this_state})
            3'b0xx : begin
                         // Reset stage
                           RNL_OUT <= 1'b0;
                           BLS_OUT <= 1'b0;
                           HLS_OUT <= 1'b0;
                           MOE_OUT <= 1'b0;
                           MWE_OUT <= 1'b0;

                           mdata_in <= 32'b0;
                           mem_this_state <= 2'b00;
                           mdne_c <= 0;
                      end
            3'b100 : begin
                        casex ({mem_read_this_state, mem_write_this_state})
                            2'b10 : begin
                                      if (pclk_del_r == 2'b01) begin
                                          /* Memory read state */
                                           RNL_OUT <= 1'b0;
                                          BLS_OUT <= 1'b0;
                                          HLS_OUT <= 1'b0;
                                          MOE_OUT <= 1'b1;
                                          MWE_OUT <= 1'b0;

                                          mem_this_state <= 2'b01;   // next state for read operation
                                      end
                                    end

                              2'b01 : begin
                                        if (pclk_del_r == 2'b01) begin
                                            /* Memory write state */
                                            RNL_OUT <= 1'b0;
                                            BLS_OUT <= 1'b0;
                                            HLS_OUT <= 1'b0;
                                            MOE_OUT <= 1'b0;
                                            MWE_OUT <= 1'b1;

                                            mem_this_state <= 2'b10;
                                        end
                                      end

                              default : begin
                                               /* No memory access this state */
                                               RNL_OUT <= 1'b0;
                                               BLS_OUT <= 1'b0;
                                               HLS_OUT <= 1'b0;
                                               MOE_OUT <= 1'b0;
                                               MWE_OUT <= 1'b0;

                                               mdata_in <= 32'b0;
                                               mem_this_state <= 2'b00;
                                        end
                           endcase
                       end
              3'b101 : begin
                          // Remove memory read request after the falling edge of MDATA_VLD_IN
                          if (mdata_vld_fall) begin
                        MOE_OUT <= 1'b0;
                      end
```

```
                   if(~MOE_OUT & mdata_vld_rise) begin  // Look for MDATA_VLD_IN signal only after initiating
the request
                                 // and after the falling edge of PCLK
                   mdata_in <= MDATA_IN;  // Assign the input data to the result register

                   // Latch the memory done signal (MDATA_VLD_IN)
                              mdne_c <= 1;
                mem_this_state <= 2'b00;
             end
                   end
          3'b110 : begin
                       // Remove memory write request after the falling edge of MDATA_VLD_IN
                       if (mdata_vld_fall) begin
                     MWE_OUT <= 1'b0;
                   end
                   if(~MWE_OUT & mdata_vld_rise) begin  // Look for MDATA_VLD_IN signal only after initiating
the request
                                 // and after the falling edge of PCLK

                   // Latch the memory done signal (MDATA_VLD_IN)
                   mdne_c <= 1;
                mem_this_state <= 2'b00;
             end
                   end
          default : begin
                    end
      endcase
  end

   // state machine logic for compiled extension
   always @ (*) begin
      pclk_rise = (pclk_del_r == 2'b01);
      pclk_fall = (pclk_del_r == 2'b10);

      // start the extension instruction
      clr_dne = state_r[1] & en_r & sll128_r;

      // state machine for read logic
      read_this_state = (| (state_r & read_state_r));
      rsm_wait = read_this_state &
                 (rsm_idle_r & gr_r) |
                 (rsm_wait_r & (rsm_count_r != REG_READ_WAIT_STATES));
      rsm_latch = rsm_wait_r & (rsm_count_r == REG_READ_WAIT_STATES);
      rsm_wait2 = (rsm_wait2_r | rsm_latch_r) & ~done_state;
      rsm_idle = ~rsm_wait & ~rsm_wait2 & ~rsm_latch;
      rsm_count = rsm_idle_r ? 4'h0 : (rsm_count_r + 1);
      rsm_done = ~read_this_state |
                 (read_this_state & (rsm_latch_r | rsm_wait2_r));

      // state machine for write logic
      write_this_state = (| (state_r & write_state_r));
      wsm_pulse = wsm_idle_r & write_this_state & gr_r & regrdy_r & ~regfull_r;
      wsm_wait = (wsm_pulse_r & ~done_state) |
                 (wsm_wait_r  & ~done_state);
      wsm_idle = ~wsm_pulse & ~wsm_wait;
      wsm_done = ~write_this_state |
                 (write_this_state & (wsm_pulse_r | wsm_wait_r));

      // state machine for Memory read logic
      mem_read_this_state = (| (state_r & mem_read_state_r)) & ~mdne;
      mrs_done = ~mem_read_this_state |
                      (mem_read_this_state & mdne);

      // state machine for Memory write logic
      mem_write_this_state = (| (state_r & mem_write_state_r)) & ~mdne;
      mws_done = ~mem_write_this_state |
                       (mem_write_this_state & mdne);

      // state machine for branch logic
      branch_this_state = (| (state_r & branch_state_r));
      bsm_calc = bsm_idle_r & branch_this_state;
      bsm_waitpf = (bsm_calc_r & take_branch_r) |
                   (bsm_waitpf_r & ~pclk_fall);
      bsm_waitpr = (bsm_waitpf_r & pclk_fall) |
                   (bsm_waitpr_r & ~pclk_rise);
      bsm_wait = (bsm_calc_r & ~take_branch_r & ~done_state) |
                 (bsm_waitpr_r & pclk_rise & ~done_state) |
                 (bsm_wait_r & ~done_state);
      bsm_idle = ~bsm_calc & ~bsm_wait & ~bsm_waitpr & ~bsm_waitpf;
      bsm_done = ~branch_this_state |
                 (branch_this_state &
                    ((bsm_calc_r & ~take_branch_r) |
                     (bsm_waitpr_r & pclk_rise) |
                      bsm_wait_r));

      // state machine to finish up the extension instruction
      final_state = state_r[MAX_STATE];
      fsm_wait = final_state & rsm_idle_r |
                 (fsm_wait_r & ~(gr_r & regempty_r & extnop_ma_r));
      fsm_wait2 = (fsm_wait_r & gr_r & regempty_r & extnop_ma_r) |
                  (fsm_wait2_r & ~en_r);
      fsm_idle = ~fsm_wait & ~fsm_wait2;
      fsm_done = final_state & fsm_wait2_r & ~en_r;

      // clear DNE as the extension instruction is entered
      // set DNE as the extension instruction is exited
      DNE_c = (DNE | (fsm_wait_r & gr_r & regempty_r & extnop_ma_r)) & ~clr_dne;
      ACK_c = (ACK | (~DNE & ~ACK)) & ~(ACK & DNE & pclk_rise) & EN;
      // & EN -> to bring down ACK when it loses control/resources in case of an interrupt
   end

   always @ (*) begin
      // true when all conditions for a state have been satisfied
      done_state = clr_dne |
                   (~state_r[1] & bsm_done & rsm_done & wsm_done & mrs_done & mws_done & tran_state_done);
   end


   // state to determine rising and falling edges of pclk
   always @ (posedge CLK) begin
      pclk_del_r <= {pclk_del_r[0], PCLK};
      // rise and falling edge of MDATA_VLD_IN
      mdata_vld_r <= {mdata_vld_r[0], MDATA_VLD_IN};
   end

   // buffer signals that may be heavily loaded or come from a distance
   //   - is this needed?  this is present to maintain compatibility with Neil
   always @ (posedge CLK) begin
      if (!RESET) begin
         en_r <= 1'h0;
         sll128_r <= 1'h0;
         gr_r <= 1'h0;
         regrdy_r <= 1'h0;
         regfull_r <= 1'h0;
         regempty_r <= 1'h0;
         extnop_ma_r <= 1'h0;
      end else begin
         en_r <= EN;
         sll128_r <= SLL128;
         gr_r <= GR;
         regrdy_r <= REGRDY;
         regfull_r <= REGFULL;
         regempty_r <= REGEMPTY;
         extnop_ma_r <= EXTNOP_MA;
      end
   end

   // misc control for the extension
   always @ (posedge CLK) begin
      if (!RESET) begin
         ACK <= 1'h0;
         DNE <= 1'h1;
         done_state_r <= 1'b0;

         wsm_idle_r <= 1'b1;
         wsm_pulse_r <= 1'b0;
         wsm_wait_r <= 1'b0;

         rsm_idle_r <= 1'b1;
         rsm_latch_r <= 1'b0;
         rsm_wait_r <= 1'b0;
         rsm_wait2_r <= 1'b0;
         rsm_count_r <= 4'b0;

         bsm_idle_r <= 1'b1;
         bsm_calc_r <= 1'b0;
         bsm_wait_r <= 1'b0;
         bsm_waitpr_r <= 1'b0;
         bsm_waitpf_r <= 1'b0;
         take_branch_r <= 1'h0;

         fsm_idle_r <= 1'b1;
         fsm_wait_r <= 1'b0;
         fsm_wait2_r <= 1'b0;

      end else begin
         /*
         // clear ack
         if (ACK & DNE & pclk_rise)
            ACK <= 1'h0;
         // set ack
         else if (~DNE & ~ACK)
            ACK <= 1'h1;
         */

         ACK <= ACK_c;
         DNE <= DNE_c;
         done_state_r <= done_state;

         wsm_idle_r <= wsm_idle;
         wsm_pulse_r <= wsm_pulse;
         wsm_wait_r <= wsm_wait;

         rsm_idle_r <= rsm_idle;
         rsm_latch_r <= rsm_latch;
         rsm_wait_r <= rsm_wait;
         rsm_wait2_r <= rsm_wait2;
         rsm_count_r <= rsm_count;

         bsm_idle_r <= bsm_idle;
         bsm_calc_r <= bsm_calc;
         bsm_wait_r <= bsm_wait;
         bsm_waitpr_r <= bsm_waitpr;
         bsm_waitpf_r <= bsm_waitpf;
         // if take_branch_r is ever used outside of the branch state machine,
         // it may need to be cleared at the end of the branch operation
         take_branch_r <= bsm_calc ? take_branch : take_branch_r;

         fsm_idle_r <= fsm_idle;
         fsm_wait_r <= fsm_wait;
         fsm_wait2_r <= fsm_wait2;
      end
   end

//
// INFO: finished reading from m2v_state_mc.v
//

   // state machine for transaction model
   always @ (*) begin
      transaction_end_this_state = (| (state_r & tran_end_state_r));

      virpc_tr0 = PC;
      virpc_tr1 = PC + 28;
      virpc_tr2 = CJMPADD;

      VIRPC =  ({32{state_r[1]}} & virpc_tr0)
             | ({32{state_r[2]}} & virpc_tr0)
             | ({32{state_r[3]}} & virpc_tr1)
             | ({32{state_r[4]}} & virpc_tr1)
             | ({32{state_r[5]}} & virpc_tr1)
             | ({32{state_r[6]}} & virpc_tr2)
             | ({32{state_r[7]}} & virpc_tr2);
   end

   // transaction model control for the extension
   always @ (posedge CLK) begin
      if (!RESET) begin
         tran_state_done <= 1;
      end else begin
         if (EXCEXT & EN) begin
            if (transaction_end_this_state) begin
               tran_state_done <= 0;
            end
         end else begin
            if (~EN) begin
               tran_state_done <= 1;
            end
         end
      end
   end


   // registers that contain state about this cycle
   always @ (posedge CLK) begin
      if (~RESET) begin
         branch_state_r[1] <= 1'b0;
         write_state_r[1]  <= 1'b0;
         read_state_r[1]   <= 1'b1;
         mem_write_state_r[1]  <= 1'b0;
         mem_read_state_r[1]   <= 1'b0;
         tran_end_state_r[1]   <= 1'b1;

         branch_state_r[2] <= 1'b0;
         write_state_r[2]  <= 1'b0;
         read_state_r[2]   <= 1'b1;
         mem_write_state_r[2]  <= 1'b0;
         mem_read_state_r[2]   <= 1'b0;
         tran_end_state_r[2]   <= 1'b1;

         branch_state_r[3] <= 1'b1;
         write_state_r[3]  <= 1'b1;
         read_state_r[3]   <= 1'b1;
         mem_write_state_r[3]  <= 1'b0;
         mem_read_state_r[3]   <= 1'b0;
         tran_end_state_r[3]   <= 1'b0;

         branch_state_r[4] <= 1'b0;
         write_state_r[4]  <= 1'b1;
         read_state_r[4]   <= 1'b0;
         mem_write_state_r[4]  <= 1'b0;
         mem_read_state_r[4]   <= 1'b0;
         tran_end_state_r[4]   <= 1'b0;

         branch_state_r[5] <= 1'b0;
         write_state_r[5]  <= 1'b1;
         read_state_r[5]   <= 1'b0;
         mem_write_state_r[5]  <= 1'b0;
         mem_read_state_r[5]   <= 1'b0;
         tran_end_state_r[5]   <= 1'b1;

         branch_state_r[6] <= 1'b0;
         write_state_r[6]  <= 1'b1;
         read_state_r[6]   <= 1'b0;
         mem_write_state_r[6]  <= 1'b0;
         mem_read_state_r[6]   <= 1'b0;
         tran_end_state_r[6]   <= 1'b0;

         branch_state_r[7] <= 1'b0;
         write_state_r[7]  <= 1'b1;
         read_state_r[7]   <= 1'b0;
         mem_write_state_r[7]  <= 1'b0;
         mem_read_state_r[7]   <= 1'b0;
         tran_end_state_r[7]   <= 1'b1;

      end else begin
         branch_state_r <= branch_state_r;
         write_state_r  <= write_state_r;
         read_state_r   <= read_state_r;
         mem_write_state_r  <= mem_write_state_r;
         mem_read_state_r   <= mem_read_state_r;
         tran_end_state_r   <= tran_end_state_r;
      end
   end


   // combinatorial logic to/from the register file
   always @ (*) begin
      // combinatorial logic for register reads
      // use read ports 3 & 4 to prevent write conflicts
      RDREG1 = 0;
      RDREG2 = 0;
      r9_1 =  RDREG3DATA;
      r8_4 =  RDREG2DATA_ID;
      r4_11 =  RDREG1DATA_ID;
      r5_18 =  RDREG4DATA;
      r6_23 =  RDREG3DATA;
      RDREG3 =  ({5{state_r[2]}} & (RT + 1))
              | ({5{state_r[1]}} & RT)
              | ({5{state_r[3]}} & (RS + 2));
      RDREG4 =  ({5{state_r[1]}} & RS)
              | ({5{state_r[2]}} & (RS + 1));

      // combinatorial logic for register writes
      WRREG1 =  ({5{state_r[3]}} & 11)
              | ({5{state_r[4]}} & RT)
              | ({5{state_r[5]}} & (RT + 1))
              | ({5{state_r[6]}} & RS)
              | ({5{state_r[7]}} & 11);
      WRDATA1 = ({32{state_r[3]}} & r11_13_r)
              | ({32{state_r[4]}} & r8_15_r)
              | ({32{state_r[5]}} & r9_8_r)
              | ({32{state_r[6]}} & r4_22_r)
              | ({32{state_r[7]}} & r11_25_r);
      REGWRITE1 = wsm_pulse_r & (state_r[3]
               | state_r[4]
               | state_r[5]
               | state_r[6]
               | state_r[7]);
      WRREG2 =  ({5{state_r[4]}} & RS)
               | ({5{state_r[6]}} & (RS + 1));
      WRDATA2 = ({32{state_r[4]}} & r4_17_r)
               | ({32{state_r[6]}} & r5_29_r);
      REGWRITE2 = wsm_pulse_r & (state_r[4]
               | state_r[6]);
   end

   // internal pipeline logic
   always @ (posedge CLK) begin
      if (~RESET) begin
         r8_4_r <= 32'h0;
         r4_11_r <= 32'h0;
         r9_8_r <= 32'h0;
         r11_13_r <= 32'h0;
         r8_15_r <= 32'h0;
         r4_17_r <= 32'h0;
         r4_22_r <= 32'h0;
         r11_25_r <= 32'h0;
         r5_29_r <= 32'h0;
      end else begin
         r8_4_r <=  state_r[1] ? r8_4 : r8_4_r;
         r4_11_r <=  state_r[1] ? r4_11 : r4_11_r;
         r9_8_r <=  state_r[2] ? r9_8 : r9_8_r;
         r11_13_r <=  state_r[2] ? r11_13 : r11_13_r;
         r8_15_r <=  state_r[2] ? r8_15 : r8_15_r;
         r4_17_r <=  state_r[2] ? r4_17 : r4_17_r;
         r4_22_r <=  state_r[2] ? r4_22 : r4_22_r;
         r11_25_r <=  state_r[3] ? r11_25 : r11_25_r;
         r5_29_r <=  state_r[2] ? r5_29 : r5_29_r;
      end
   end


   // logic for the Memory address and data out
   always @ (posedge CLK) begin
      MADDR_OUT <=  (32'h0);
   MDATA_OUT <= (32'h0);
   end

   // combinatorial logic for the instruction nodes
   always @ (*) begin
      // [0x0] 0x10840 sllr9, r9, 1
      r9_3 = r9_1 << 1;

      // [0x4] 0x21fc2 srlr11, r8, 31
      r11_6 = r8_4_r >> 31;

      // [0x8] 0x230825or r9, r9, r11
      r9_8 = r9_3 | r11_6;

      // [0xc] 0x21040 sllr8, r8, 1
      r8_10 = r8_4_r << 1;

      // [0x10] 0x41fc2 srlr11, r4, 31
      r11_13 = r4_11_r >> 31;

      // [0x14] 0x431025or r8, r8, r11
      r8_15 = r8_10 | r11_13;

      // [0x18] 0x42040 sllr4, r4, 1
      r4_17 = r4_11_r << 1;

      // [0x1c] 0x51fc2 srl r11, r5, 31
      r11_20 = r5_18 >> 31;

      // [0x20] 0x832025 or r4, r4, r11
      r4_22 = r4_17 | r11_20;

      // [0x24] 0x26182b sltu r11, r9, r6
      r11_25 = ({1'b0, r9_8_r} < {1'b0, r6_23}) ? 1 : 0;

      // [0x28] 0x10030005 beq r0, r11, 20
      take_branch = (32'h0 == r11_25);
      CJMPADD = take_branch ? (PC + 4 + {{16{DIMM[15]}},DIMM}) : PC;
      PCNEXT = state_r[3] & bsm_waitpr & take_branch;

      // [0x2c] 0x52840 sll r5, r5, 1
      r5_29 = r5_18 << 1;

   end

   // primary extension state machine
   always @ (posedge CLK) begin
      if (~RESET) begin
         state_r <= 1;
         mdne <= 0;
      end else begin
         mdne <= mdne_c;
         if (en_r) begin
            if (done_state) begin
               state_r <= {state_r[MAX_STATE-1:1], 1'b0};
               mdne <= 0;
            end
         end
         else begin
            state_r <= 1;
         end
      end
   end

endmodule
```

Appendix IV – Circuit Graph Visualization

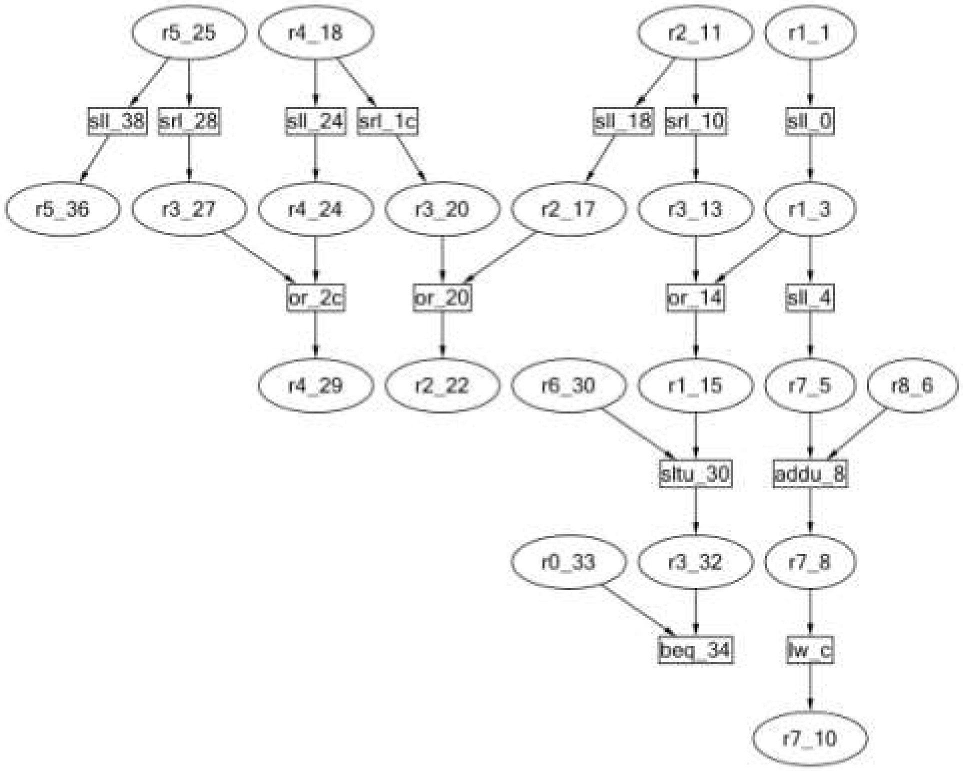


Figure A-4: Visualization of the Circuit graph using the DOT file generated by the M2V compiler, for the example basic block from Figure 10.