Extensible On-Chip Peripherals

Bharat Sukhwani, Alessandro Forin, Richard Neil Pittman
Microsoft Research

September 2007

Technical Report
MSR-TR-2007-120

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Extensible On-Chip Peripherals

Bharat Sukhwani, Alessandro Forin, Richard Neil Pittman
Microsoft Research

Abstract

This document describes the I/O subsystem of the eMIPS dynamically self-extensible processor. This processor, during execution, can load additional logic blocks that can perform a variety of functions from adding new instructions to the base instruction set to controlling I/O pins. A dynamically loaded logic block that acts as an I/O peripheral to software is what we term an Extensible I/O Peripheral.

On eMIPS, the type, number and memory space allocation of on-chip peripherals is known only at runtime, when it can change dynamically with the loading and unloading of processor Extensions. We have added to the eMIPS design additional mechanisms for a newly loaded Extensible On-Chip Peripheral to connect to the memory controller, to disconnect from it, to interact with system software in the discovery process, and to obtain the I/O space and interrupt resources that it needs to operate correctly.

A general purpose operating system running on eMIPS is able to verify the security level of any processor Extension before it is enabled. Because it only executes in the address space of the application that uses it, other applications are insulated against potentially malicious Extensions. We have extended the security model to Extensible On-Chip Peripheral and their software drivers. Privileged peripherals can request access to additional interface signals that are not normally available to non-privileged Extensions. These signals allow access to physical memory, interrupt lines and I/O pins.

Extensible On-Chip Peripherals can interact with system software via memory-mapped I/O. But they can also add new I/O instructions to the processor. For instance, atomic multi-register data transfers can simplify the interaction between software and interrupt routines, especially on multi-core systems.

1 Introduction

eMIPS is a dynamically extensible processor that includes a standard MIPS trusted ISA and extensible hardware slots. The programmable logic can plug into the main pipeline stages during the execution of a program. In addition to providing hardware acceleration of certain basic blocks of code for improved performance, the extension slots can be used for a variety of other purposes. In this document, we introduce the idea of using them to implement Extensible On-Chip I/O Peripherals – on-chip I/O peripherals and I/O interfaces that can be dynamically loaded and unloaded during program execution.

In order to provide the self-extensible feature, the eMIPS processor is implemented on a field programmable gate array. The amount of hardware real-estate available on any given FPGA chips is limited, thus limiting the number of on-chip peripherals that can be supported on the eMIPS platform. Here, we propose to load and unload the peripherals as needed during program runtime, thus supporting a larger number of on-chip peripherals, albeit not all at the same time. Furthermore, this approach provides on-chip support for any new peripheral that might become available in the future, without the need for redesigning or even recompiling the existing hardware.

Extensible on-chip peripherals enable hardware reuse by allowing the extension slots to be shared between extension instructions and peripherals, without tying up the hardware resources. In other words, using eMIPS, one can provide support for many peripherals without allocating area resources to them until they are actually used. This feature makes available more hardware resources for use in extension instructions, thus achieving higher performance speed-ups. In addition, the extensible peripherals can implement specialized instructions capable of performing multiple atomic I/O operations without the need for disabling the interrupts, locking the memory bus or switching to a different processing core.

Figure 1 shows the block diagram of an eMIPS processor with one of the extension slots used for an extensible peripheral. As shown in the figure, the eMIPS processor is divided into two main parts: the hard fabric and the soft fabric, both implemented on the same FPGA chip. The hard fabric, as the name suggests, represents the fixed logic that does not change during program execution. It consists of the minimal hardware required for the processor to securely run standard MIPS instructions (MIPS ISA, memory controller, interrupt controller, etc.) along with the modules that are required to support the

dynamic loading of extension slots. The hard fabric is sometimes referred to as the Trusted Instruction Set Architecture (TISA), for the role it plays in the security architecture of the eMIPS processor.

The soft fabric part of eMIPS consists of the extension slots used to implement specialized instructions that are not part of the MIPS instruction set. These instructions replace a block of code (e.g. a for loop) with a single instruction running on specialized hardware, thereby improving the program performance. Figure 1 shows one such extension slot used for implementing an on-chip peripheral instead of extension instructions. It is in this area that we realize the signals. System software can unload a peripheral, for instance when it is finished using it. This relinquishes both the portion of address space used by the peripheral and the reconfigurable extension slot which can then be reused by another extension.

During different phases of the peripherals' life cycle (configuration, normal operation etc.), its access to the memory bus is controlled by system software using the coprocessor 0 register. Only a limited set of signals are accessible to an Extension until system software enables the rest using this register. This register resides inside the TISA, making the extensible peripherals scheme secure

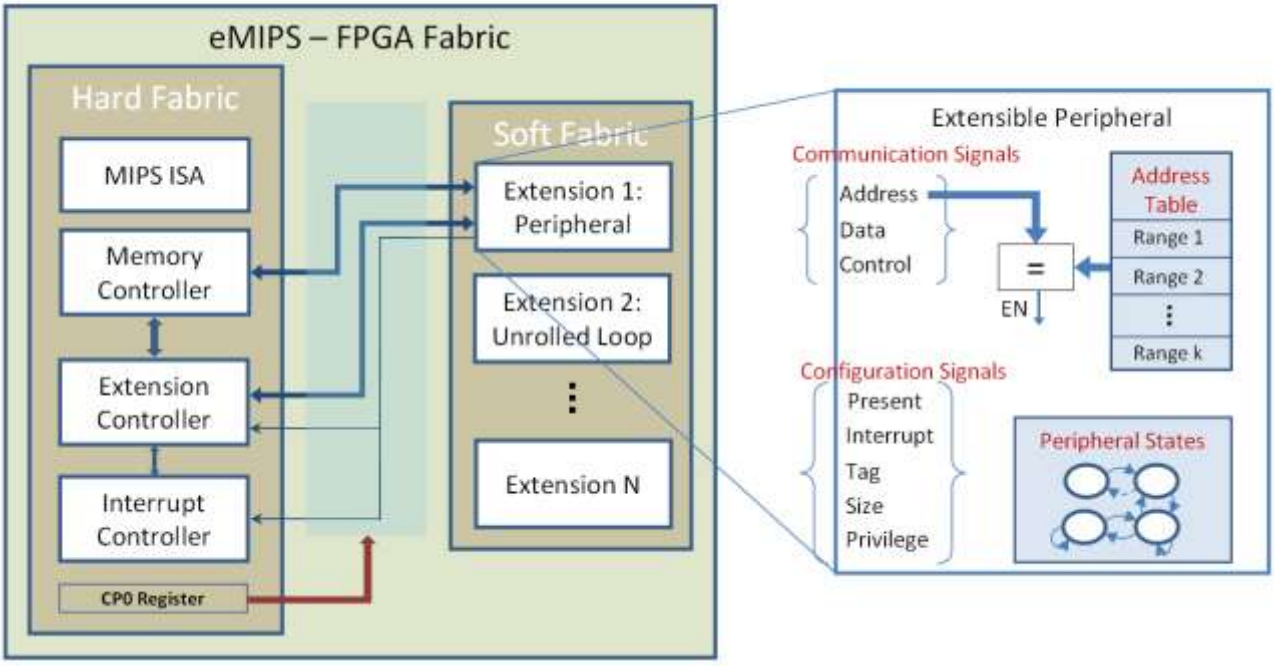

Figure 1: The eMIPS processor with an extensible peripheral loaded in Extension slot #1.

Extensible On-Chip Peripherals.

Dynamic loading and unloading of Extensible On-Chip Peripherals is performed by software. System software initiates the loading of an Extensible On-Chip Peripheral by sending a signal to the Xilinx System Ace FPGA control chip, which in turn loads the configuration bit file for the extension into one of the slots in the soft fabric of the reconfigurable hardware.

To appear on the memory bus, the peripheral must be configured into the system memory map and assigned any additional resources it may require (such as interrupts). This is done using the configuration signals of the extension interface (Figure 1). These signals are sent to the extension controller, the module that administers the peripheral configuration process. Once configured, the peripheral appears on the memory bus and communicates to the memory controller using its own communication from malicious attacks.

The remainder of this document is structured as follows. Section 2 summarizes the related work. Section 3 is a high-level discussion of the issues and trade-offs in I/O system design. In section 4, we discuss the atomicity issues and show how eMIPS can be used to solve them. Section 5 presents the structure and details of the Extensible On-Chip Peripheral configuration process. In section 6 we describe the hardware security model of the eMIPS processor with respect to extensible peripherals. Section 7 discusses our implementation and validation of the Extensible On-Chip Peripheral concept. Section 8 concludes the paper, with discussion on future work and further applications of extensible peripherals.

2 Related Work

Dynamic insertion and removal of peripheral boards from off-chip I/O busses is a well known and standardized feature for reliable computer systems [11] [18]. Our work addresses the case of on-chip peripherals, which (by definition) cannot be implemented as physically removable boards. Nonetheless, many ideas and techniques apply to both cases.

Partial Reconfiguration [25] is a relatively recent technique developed for FPGAs that allows dynamic loading and unloading of logic modules into certain areas of the FPGA chip, while other areas continue to function unperturbed. The eMIPS microcomputer [19] relies on this technique for realizing and managing all Extensions, including the case of Extensible On-Chip Peripherals. Other extensible processors [22] have been realized with a fixed I/O subsystem e.g. in the ASIC side along with the main data path, but the ideas and techniques described herein can be adapted to those designs as well.

Most of the research on extensible processors has focused on using reconfigurable logic to improve application performance [7] [21] [16] [23] [26] [14] [20] [10] [5] [15] [9] [6] [3] and very little has been reported on realizing more flexible I/O subsystems [23] [3]. OneChip-96 [23] did implement a USART, but in this rather limited prototype the processor and the extensions were compiled together and therefore the system could not explore any of the problems addressed in this work. The Egret platform [3] uses the OneWire protocol to communicate with external I/O devices. Once a device is identified the system loads some more specialized logic to handle all communications with the external component and the OneWire logic module is overwritten. In Egret, the OPB bus is the processor interface to the peripherals and it is pre-compiled into the fixed part of the design. Therefore Extensions cannot be used to realize any other functionality, for instance specialized I/O instructions. In eMIPS, the interface is compiled in the peripheral itself and the system is therefore more flexible. Our approach does not require the OneWire module. Egret does not appear to explicitly handle the case of peripheral removals.

The concept of a dynamically self-extensible processor is relatively new and to date not enough progress has been made towards an actual implementation of this and related concepts in a complete, usable and safe multi-user system. The analysis of the security models is therefore non-existent and so are the implications for system software and any practical usability studies.

Synchronizing an interrupt service routine (ISR) with the rest of a device driver is a well known system programming problem. Less understood is the case where the ISR runs in one process and the rest of the device driver in another. Forin et al. [8] encountered this case for user-mode device drivers in the Mach Operating System. They used a special system service to solve the ensuing synchronization issues.

Specialized I/O instructions have usually appeared in architectures in the form of data movement to and from the processor registers and some specific I/O component or address [17] [12]. The IBM System/360 [2] evolved from an earlier form that used specialized I/O instructions executed by the processor and mutated them into "Channel Instructions", executed directly by the I/O peripheral. A single "START I/O" instruction is used either to start the sequence of I/O instructions in a separate I/O channel, or directly by the CPU in the older/simpler computer models. The 8086 processor [12] used "IN" and "OUT" instructions to access its I/O ports, logical I/O locations in a space distinct from the memory map. A bit on the bus separates I/O transactions from regular memory transactions. The 80286 [13] processor introduced a bit in the task descriptor to permit or deny execution of the specialized I/O instructions, regardless of the privilege level of the processor. In later versions of the x86 architecture the port model of I/O has been largely abandoned in favor of memory mapping of the I/O locations.

3 Models of I/O

In the design space for I/O architectures, there are two clear extremes: the memory-mapped I/O approach (M) on one end, and the specialized instruction approach (S) on the other. Somewhere in the middle falls the case of queues of messages (Q) used to send data and commands to and from peripheral devices. The Q model has recently gained popularity especially with 3-D graphical accelerators and the Infiniband cluster interconnect. Note that the Q model can be implemented on top of either the M or S models.

There are a number of dimensions along which we can compare I/O models. One dimension is how the model scales with the number of processor cores. The M model has a few deficiencies in this respect. The routing of I/O interface traffic over the same bus as memory traffic can cause congestion, especially if the transactions are synchronous and/or non-interruptible. Interrupts might be dispatched to a different core, creating serialization concerns. Synchronization between the ISR and non-interrupt level software is more problematic when multiple cores are involved. Both the S and Q models can more flexibly address these issue in hardware.

The issue with ISRs is but one of those discussed later on in section 4 under the "atomicity" umbrella. The result of this discussion is that the S model is clearly a winner in this respect. Provided it can be implemented atomically, the Q model shares the same advantages as the S model.

Processor virtualization – e.g. virtual machines, requires hardware support for best performance and I/O is one of the areas where performance suffers the most from the virtualization process. In this respect, the amount of information contained in a load/store instruction is much less than what is carried in a specialized instruction, especially if such an instruction can be tied to a specific peripheral. Indeed, a common approach to solve the I/O problem is to modify the guest OS with special device drivers that make use of specialized instructions. The Q model of I/O might provide even better performance, provided it could be used to deposit the payloads directly into the guest OS' address space. The price to pay for direct user mode access by a peripheral is a duplicated MMU on the device and the costs of keeping the MMUs synchronized.

User-mode access to peripherals [8] is one way to virtualization, with the potential for minimizing the data movement costs. All three I/O models are equally applicable here, but the M model does not provide any obvious way to synchronize with I/O completions. Both of the S and Q models can easily provide for a wait operation.

I/O devices very often must be multiplexed among many software components, which is the essential function of a device driver. The Q model is clearly superior here, a queue being but the simplest form of a multiplexor. The S model can be made to work equally well, and the M model has serious synchronization and atomicity issues.

Clusters are groups of processors that do not share memory but are nonetheless meant to operate as a single machine. For these systems the Q model is clearly superior and the M model is not applicable. The S model is best used to implement the Q model.

An I/O model requires a certain amount of logic resources to implement the hardware interface of the model. This is a recurring per-peripheral cost - the interface must be replicated in each peripheral. A model that requires less resources is therefore desirable from a cost perspective. The M model is easier to implement and therefore requires less resources, at least as long as the memory bus protocols are kept at a reasonable level of complexity.

This limited discussion suffices to show that there is no I/O model that is clearly superior to the others, each has its advantages and disadvantages depending on other system and software factors that are outside the control of the I/O designer. With eMIPS, we have the unique opportunity of not having to choose one particular model, all can be supported by the same basic hardware architecture. Extensible On-Chip Peripherals can be made visible on the memory bus to implement the M model. They can also recognize individual processor instructions to implement the S model. Those instructions can be of the send-receive form to implement the Q model of I/O. Note also that the architecture does not favor one model over any other.

4 Atomicity

In this section we define "atomicity" with respect to I/O peripherals, using a simple example and analyzing the practical problems it raises. We use an example to contrast the load-store I/O model with the additional possibilities offered by the eMIPS I/O architecture.

As a working example, we use the case of a hypothetical FM radio peripheral. Actual radios do exist that have similar interfaces, but the details are not important to the discussion. Table 1 shows the commands that can be sent to our radio; they include tuning the radio to a specific radio station, asking it to scan up/down to the next station, and to retrieve the basic properties of a radio station.

Table 1. FM radio command set

Command	Arguments	Results
SetFrequency	Frequency	ErrorCode
GetFrequency		Frequency
StartScan	Direction	
StopScan		ErrorCode
SetScanLimits	LowFrequency, HighFrequency	ErrorCode
GetMode		Mono/Stereo, Caption, etc.

Consider the pseudo code in Figure 2, which defines two software functions that need access to the radio. The first is a loop that constantly monitors the current frequency and displays it on the graphical readout. The second is the interrupt handling routine, which we presume is invoked when a StartScan command reaches a valid station, or if it times out. The software application could be run either on a single or a multi core system.

To illustrate the atomicity issues created by the load/store model for I/O, Figure 3 shows the MIPS assembly code for performing one of the radio commands present in Table 1, the others being quite similar. The first instruction loads into register t0 a constant defining the command for the radio, which is sent to the peripheral by the store instruction that immediately follows. The next load instruction retrieves the result from the peripheral's ARGUMENT register, and the function then returns.

If we identify with "S0" or "L0" the store/load from the DisplayThread and with "S1" or "L1" the store/load from the InterruptHandler, during execution we can conceivably observe the following six permutations:

S0L0S1L1, S1L1S0L0, S0S1L1L0, S0S1L0L1, S1S0L1L0, and S1S0L0L1. If we assume that the DisplayThread can be context-switched away by the operating system we could also observe sequences containing repetitions of S0L0 (or S1L1), but this case can clearly be reduced to one of the above six ones. Of these six, only the first two produce the expected results; in all the other cases one function will erroneously return the result of the other. The last three sequences can only be realized on a multi core system, and only if the interrupt is dispatched to a different core than the one where the DisplayThread runs.

The errors occur because the functions are not atomic with respect to interrupts and/or to each other. The only solution is to force them atomic, preventing the last four erroneous sequences from occurring. This can be done on a uniprocessor by disabling interrupts inside e.g. the GetFrequency function. On a multiprocessor it is necessary to disable interrupts on the processor where the InterruptHandler is dispatched, not necessarily the current one. If disabling of interrupts is done on a per-peripheral basis, it is also possible to context-switch away the DisplayThread while interrupts are disabled and to leave the interrupts disabled potentially for a long time. Disabling interrupts is generally undesirable for real-time scheduling. It is inefficient on architectures where the action must be performed on the interrupt target processor and therefore the action requires rescheduling the software thread from one processor to another.

```
DisplayThread() {
      while (1) {
            Radio.Station = GetFrequency(Radio.Address);
   ShowRadio(Radio);
      }
}
InterruptHandler() {
      Radio.Mode = GetMode(Radio.Address);
}
```

Figure 2: FM radio application fragment

```
#define Radio %a0
#define COMMAND 0
#define ARGUMENT 4

GetFrequency:
   li  t0, CMD_GET_FREQUENCY
   sw    t0, COMMAND(Radio)
   lw    v0, ARGUMENT(Radio)
   jr ra
```

Figure 3: Load/Store model for FM radio

On eMIPS, the functions in Figure 2 can be realized as Extended Instructions, instructions that are recognized by the peripheral itself which runs in close proximity to the base processor pipeline. The peripheral can execute the instruction by transferring multiple registers at once and in this case by performing both the store and the load operations atomically. Interrupts are dispatched either before or after any processor instruction completes, never inside it, and this provides the desired atomicity on a uniprocessor. On a multiprocessor the peripheral can lock the I/O bus during the two transactions, serializing properly any concurrent access by multiple processors. Extended instructions can selectively be enabled in any processor context, regardless of mode. A user-mode device driver could use a separate software thread to wait for I/O completions, without the need for any additional OS services.

5 Peripheral Configuration

In this section we describe the Extensible On-Chip Peripheral configuration process first from the hardware, then from the software point of view. All signals and control bits involved in the process are defined here. The general idea is that software does not trust a peripheral during the configuration process. It must verify the degree of privilege requested in hardware against the privilege indicated in the peripheral configuration bitfile, and the signature and security digest therein.

Size (bytes)	Starting Address	Address Valid
Size 1	Address 1	1
Size 2	Address 2	0
Size n	Address n	0

Figure 4: Peripheral BAT table

5.1 Hardware Configuration Interface

The Extension Controller

A newly loaded peripheral must be configured by system software before it can operate. The peripheral must indicate to software its tag and all the address ranges it needs, including its standard control region. For example, the SRAM controller of Figure 9 indicates two ranges; a 32 byte range for its control region and a SRAM_SizeInBytes range for the SRAM. The sizes of the resources required by the peripheral are stored inside the peripheral in the Base Address Translation (BAT) table (Figure 4), which must be accessible to system software. This is also the table that is used by the peripheral to determine if a memory request belongs to it. Until it is deemed safe by software and the configuration process is complete, however, the peripheral

is ignored by the memory controller and cannot respond to the requests on the memory bus. A separate module, the extension controller, acts as a secure bridge for the communication between the peripheral and the memory controller during the configuration process. It is the extension controller that interacts with system software while mapping the peripheral and allocating all the required resources.

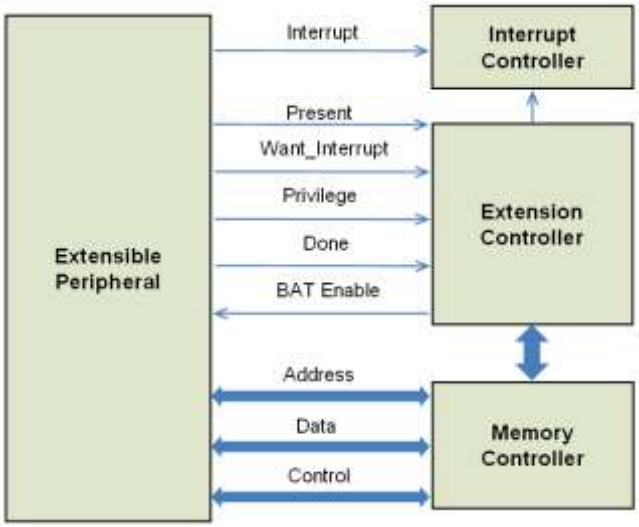

Figure 5: Peripheral interfaces to the memory controller and to the extension controller

The control signals that interface the peripheral to the memory controller and the extension controller are depicted in Figure 5. Note that a single extension controller is used to configure all the extensible peripherals. Here we show the interface signals of only one of the peripherals. Multiple sets of such signals are connected to the extension controller, one for each extension slot. The extension slot currently being configured is selectable by system software.

The extension controller contains a 32 bit control register, the lower 16 bits of which are used to indicate the size of the extension controller itself. The size field is read only for software. The upper 16 bits of the control register are used for handshaking between the peripheral currently being configured and system software. The fields of the control register are shown in Figure 6.

Upon loading, the peripheral notifies the extension controller by raising the Present signal. If the slot number of this peripheral is the one currently selected by software (bits S_No[1] and S_No[0] of the control register), the extension controller sets the LOAD and IRQ bits. If the interrupt is enabled (IRQ_EN = 1), it also raises an interrupt to the processor. After raising the interrupt request, the extension controller waits for system software to respond.

When software tries to read the tag or the BAT table entries from the peripheral, the extension controller indicates this to the peripheral by raising the BAT_Enable signal. BAT_Enable is used during the configuration process to allow the peripheral to temporarily "see" the request on the memory bus and respond to it. Upon receiving the BAT_Enable, the peripheral reads the address from the bus, places the tag/size on the data bus and raises the Done signal. After the Done signal is raised, the extension controller removes the BAT_Enable signal. A timeout is used to override the Done signal if the peripheral does not respond within a certain time. This is done to prevent the memory bus from locking-out when there is no peripheral present, or when a malicious/misbehaving peripheral tries to obtain and retain the control of the bus.

25	24	23	22	21	20	19	18	17	16	15 -0
S_No[1]	S_No[0]	PRIV	INTR	UNLOAD	LOAD	X	IRQ_EN	IRQ	RST	SIZE

Figure 6: Control register for the extension controller

A similar procedure is followed when system software tries to assign the address ranges to the peripheral by writing to the BAT table. The extension controller recognizes the BAT address and raises the BAT_Enable signal to the peripheral. Therefore during the entire configuration, it is the extension controller that decides whether the peripheral has access to the memory bus or not, not the peripheral itself. Each extension slot is controlled by a set of bits in the coprocessor0 register set, which is a privileged resource inside the TISA. One of these bits (PERIPHERAL) is logically OR-ed with the BAT_Enable bit and it is set by system software once it decides that it trusts the extension and it wants to fully enable it. Note that peripherals are given direct access to the memory bus only after they have been verified and fully configured by system software. This makes the dynamic loading of peripherals as secure and verifiable as other eMIPS Extensions.

The Want_Interrupt signal is used by the peripheral to indicate whether it needs interrupt resources or not. If this signal is high, the extension controller sets the INTR bit of its control register. System software sees this signal and enables the corresponding interrupt line in the interrupt controller. We currently use a fixed set of interrupt lines, one per extension slot.

An Extensible peripheral can also request privileged access to the memory bus. Normally an extension can only see virtual addresses, to be translated by the MMU inside the TISA. Certain peripherals will need to use physical addresses on the bus. Such a peripheral should raise the Privilege signal to the extension controller, which in turn

sets the PRIV bit of the control register. System software decides whether to grant access or not.

The peripheral keeps the Present bit high while it is loaded. The Present bit should be lowered only during the unload process. On the falling edge of the Present signal, the extension controller sets the UNLOAD bit in its control register and issues an interrupt to the processor in much the same way as during the load event. This indicates to system software that the extension slot is now free for other uses.

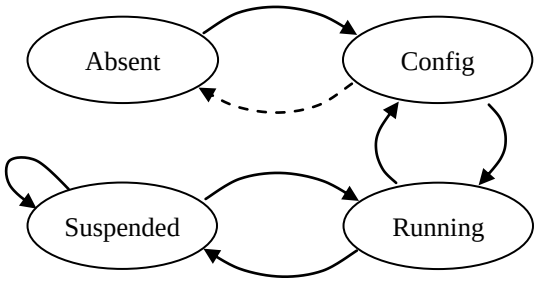

Figure 7: Peripheral configuration state machine

The Configuration State Machine

Extensible peripherals can be in one of the four states shown in the state machine of Figure 7. The state of the peripheral is set by system software and indicates to the peripheral the amount of control it has over the main memory bus. These states are:

Absent: The peripheral is not present. This can be either because the peripheral was never loaded or it was loaded but then removed. In the latter case, the peripheral might still be present in the extension slot with its state set to absent and its BAT entries invalidated by software. A peripheral in the absent state does not appear in the system memory map and cannot respond to memory requests. The extension controller returns this value when a peripheral does not respond (Done signal) before the timeout expires. In this state the PERIPHERAL bit is off.

Config: The peripheral is in the process of being configured. During this state, the peripheral can access the memory bus only through the extension controller, using the BAT_Enable signal. In this state the PERIPHERAL bit is off.

Running: This is the normal operating state of the peripheral, once it has been verified and configured. In this state, the peripheral is directly present on the main memory bus and can respond to it without the need of a BAT_Enable signal. The peripheral sees every request on the memory bus and responds to those belonging to it. In this state the PERIPHERAL bit is on.

Suspended: Software can temporarily put a peripheral in the suspended mode in order to save power or for some other reason. During this state, the peripheral suspends its normal operation and waits to be brought back to the running state. In this state, however, the BAT entries of the peripheral are not invalidated and the peripheral still appears in the peripheral mapping table. In this state the PERIPHERAL bit is on.

When a peripheral is first loaded it is in the Config state. The extension controller and system software must configure it before it can operate. After the configuration process is completed, system software updates the state of the peripheral to Running. In the Running state, software may decide to put the peripheral in low-power mode. This is done by writing the state Suspended into the peripheral. Software can also take the peripheral back to the Config state, for instance in order to change its resource assignments. To remove the peripheral, software changes its state to Absent. Once in the Absent state, all the BAT entries of the peripheral are invalidated and the peripheral can be brought back to the running state only via the Config state.

31 16	15 0
PhysicalAddressHigh	Tag

Figure 8: Entries in the peripheral mapping table

31 0	
PhysicalAddressHigh	Tag = 2
Control	SizeInBytes = 32
SRAM_SizeInBytes	
Unused	

Figure 9: Descriptor for the SRAM controller

5.2 Software Configuration Interface

The Peripheral Mapping Table

System software can explore the on-chip peripheral space using a table located in the uppermost portion of the physical memory space. The Peripheral Mapping Table (PMT) is implemented as an on-chip dual-ported RAM. The format of the entries in the PMT is depicted in Figure 8. Each entry contains a tag that identifies one of a number of peripheral types. The upper portion of the entry is the top 16 bits of (the physical address of) a 64 KB portion of memory, reserved for (a number of) peripherals of the

indicated type. Figure 9 depicts what we would typically find by following one such entry, in this case for a SRAM controller. Each peripheral auto-describes itself using the Tag and the SizeInBytes fields. The size indication is used to allocate more peripherals of the same type in the same 64 KB region. The SRAM controller also contains a PhysicalAddressHigh field, used to allocate the SRAM memory itself within the physical address space. Simpler peripherals do not need further resource allocations and will dispense with this field. The size of the SRAM bank is indicated by the SRAM_SizeInBytes field. The Unused field is only used to round-up the controller memory range to a 2^N multiple.

Software Processes

System software uses the PMT to configure peripherals into its memory map. The PMT starts at the top of the address space and grows downwards. The first entry tags the table itself and for verification purposes contains a self-pointer. The last entry of the table is also tagged specially and plays a minor role in the dynamic peripheral configuration process.

When the interrupt from the extension controller is generated, software reads the control register of the extension controller to determine if it is a load or an unload event. It also writes a 1 to the IRQ bit of the control register to indicate to the extension controller that its interrupt request has been handled. Upon receiving this indication, the extension controller removes the IRQ and clears the LOAD (or UNLOAD) bit.

In the case of a LOAD event, software reads the tag of the peripheral requesting to be enabled and the number of bytes requested for mapping in the physical address space. This is done through the extension controller, using the BAT_Enable, as explained in section 0. System software decides where to allocate the peripheral given the current state of the memory map. It looks in the PMT for segments with the same tag until it finds one with enough space left. If there are already peripherals of the same type (tag) mapped, software will try to allocate the new one in an existing segment. If there is no room, or if this is a new tag, then a new segment must be allocated and mapped into the PMT. This is done by overriding the last entry in the table and adding a new last-entry. Upon making room for the peripheral, software writes the starting address of that segment into the peripheral BAT table with the valid bit (least significant bit of this address) set to 1. Software then reads the next BAT entry for any additional memory resources needed by the peripheral. This process is repeated until the maximum number of BAT entries (currently 5) has been reached or the peripheral returns zeros for requested size.

After assigning all the memory resources, software reads the control register of the extension controller to determine if the peripheral requires interrupts and privileged access. If any of these bits are set, software checks to see if the peripheral qualifies for them. If the peripheral qualifies to get interrupts, software unmasks the corresponding bit in the interrupt controller, thereby enabling interrupts from the peripheral's extension slot. Similarly, if the peripheral qualifies for privileged memory access, software sets the corresponding bits in the coprocessor 0 register (explained in detail in section 6). Finally, software changes the state of the peripheral from Config to Running. At this point, the peripheral is fully configured and appears on the main memory bus. Software can now communicate with the peripheral directly, without the need of a BAT_Enable signal. Reading of the peripheral tag via the newly allocated memory range can be used to verify that the memory controller has in fact enabled it, that the peripheral is fully configured and that it recognizes its own addresses.

During an UNLOAD procedure, system software reads the tag of the peripheral. It unconditionally masks the peripheral interrupt in the interrupt controller. Then it clears the BAT entries of the peripheral and removes the peripheral from the PMT, thereby releasing all the memory resources used by the peripheral. It then changes the peripheral's state to ABSENT and updates the coprocessor 0 register. The peripheral lowers its Present bit and this generates an UNLOAD interrupt. This completes the unload process and the peripheral no longer appears on the memory bus.

6 Security Model

Access to the I/O pins is a potential security threat for a multi-user operating system and standard techniques should be used to restrict such access to trusted system modules. The implementer of an Extensible On-Chip Peripheral can use standard I/O memory mapping and protection techniques to implement these restrictions. Mapping of the peripheral is restricted to privileged-mode processes, e.g. the kernel process, and the peripheral is therefore invisible to other processes. If the peripheral provides Extended Instructions, those techniques are insufficient; a user-mode and/or non-trusted module must be prevented from executing the new instructions as well as being prevented from accessing the physical memory ranges allocated to the peripheral. In this section we describe the general mechanisms used on eMIPS to control the operation of Extensions.

The chip area is subdivided at processor design time into a fixed area for the TISA and a number of slots for Extensions. Each slot is controlled by four individual control bits in a special register of the eMIPS system coprocessor 0. These bits are LOADED, ENABLED, PRIVILEGED and PERIPHERAL.

The LOADED bit gates all wires to/from the corresponding Extension slot. When de-asserted, this bit effectively isolates the Extension and stops any clock signals into it. If the FPGA chip architecture allows it, this bit also disables power to the Extension's area.

The ENABLED bit determines what happens if the Extension recognizes one of the Extended Instructions. If the bit is set, the Extension is allowed to interact with the TISA and to access registers, memory and/or I/O pins. If the bit is clear the instruction is ignored and the processor proceeds to the next instruction.

The PRIVILEGED bit is used to indicate whether the extension has the privilege to access physical addresses and interrupts. This bit is set by system software upon receiving a Privilege indication from the extension. As stated earlier, peripheral extensions request privileged access by raising the privilege signal to the extension controller, which sets the corresponding bit in its control register.

Note that PRIVILEGED access can be granted to any Extension, not just to I/O peripherals. An example where an extension might require privileged access is a memory snooping extension. Such an extension can run in parallel with the main pipeline to perform on-line monitoring of the memory transactions. If the bit is set, all accesses to the memory bus are profiled, from all processes, using the physical addresses post-MMU translation. If the bit is clear, only the corresponding process is profiled, using virtual addresses pre-MMU translation.

The PERIPHERAL bit is used to indicate whether the current extension is a peripheral. It is set by system software once the peripheral configuration process is completed. During normal operations (peripheral in running state), a peripheral is allowed to place data on the memory bus only if this bit is set. In that respect, the PERIPHERAL bit can be thought of as a permanent BAT_Enable signal that stays high for as long as the peripheral is loaded and running. At any time, however, system software can reset this bit to zero, cutting the peripheral off from the memory bus. This might be necessary, for instance, to stop a misbehaving peripheral that had already been granted control of the memory bus.

Thus, every extension slot is controlled using the secure coprocessor 0 register, making the extensions fully secure and controllable by system software. Further, these bits can operate independently of the User/Kernel mode bits; therefore it is possible to grant access to the Extended Instructions to user mode modules [8].

7 Implementation and Validation

To realize and test the dynamic loading and unloading of I/O peripherals in the extension slots, we have modified the eMIPS base design in two ways. We have changed the Extension interface as discussed in Section 5.1 and Section 6 and implemented the extension controller as a new module on the TISA side. We have then implemented two extensible peripherals, a timer and a debugger interface.

The extensible timer acts as a second timer peripheral, the first (default) being the on-chip peripheral already present on the hard fabric. It contains a 64 bit free-running counter and a 64 bit down counter. A software test program loads this timer, configures it, dumps the control registers of both timers on the screen and then unloads the second (extensible) timer. The loading and unloading is verified by dumping the contents of the peripheral mapping table. The output from running this program on the ML401 board is similar to the output from a previous simulation [1].

The extensible debugger peripheral contains two parts. One is a USART interface that uses two pins of the FPGA to connect to a Maxim serial line interface. The other is the control logic to interface to a software debugger. Only the USART portion is relevant to this discussion.

We have implemented these extensible peripherals on the Xilinx Virtex-4 XC4LX25 FPGA, using the synthesis tools from the Xilinx ISE v8.2i.. The area resources required for the extension controller and for the extensible peripherals are shown in Table 2. As shown in the table, the extension controller requires less than 1% of the chip area. It currently has support logic for two extensible peripheral slots. We anticipate that adding the capability to support more peripherals will not increase the area overhead significantly. The additional interface logic required on the peripherals themselves in order to support dynamic loading and unloading is also minimal. Thus, the addition of the extensible peripheral feature to eMIPS does not claim a significant portion of the hardware resources that are available to the eMIPS extensions and hence it does not limit the potential for application speedup.

Table 2. Area Requirements on Virtex-4 FPGA

Module	# of Slices	% of Total
Extension Controller	55	< 1
Extensible Timer Peripheral	716	6
Extensible USART Peripheral	33	< 1

8 Conclusions

In this paper, we introduced the concept of Extensible On-Chip Peripherals to enable dynamic loading and unloading of I/O peripherals in the extension slots of the eMIPS processor. Extensible peripherals do not occupy any hardware resource until actually loaded, thus providing support for virtually infinite number of peripherals. They can also be used to implement a variety of instructions that

can perform atomic multi-register operations, provide debugger support and possibly enable virtualization of peripherals. To the best of our knowledge, this is the first system that provides the capability of securely loading either on-chip peripherals or extension instructions at run-time, from within a multi-user environment. Furthermore, our implementation of Extensible On-Chip Peripherals shows that they can consume minimal FPGA real-estate, thus maintaining the same speedups originally achievable by the eMIPS processor.

References

[1] Almeida, O., Forin, A., Garcia, P., Helander , J., Khantal, N., Lu, H., Meier, K., Mohan, S., Nielson, H., Pittman, R. N., Serg, R., Sukhwani, B., Veanes, M., Zorn, B., Berry, S., Boyce, C., Chaszar, D., Culrich, B., Kisin, M., Knezek, G., Linam-Church, W., Liu, S., Stewart, M., Toney, D. *Embedded Systems Research at DemoFest'07*. Microsoft Research Technical Report MSR-TR-2007-94, July 2007.

[2] Amdahl, G., M., Blaauw, G., A., Brooks, F. P. Jr. 1964 Architecture of the IBM System/360. IBM Journal of Research and Development, Vol. 8, No. 2, 1964

[3] Bergmann, N., Lu, Y., Williams, J., A. 2007. Automatic Self-Reconfiguration of System-on-Chip Peripherals. Poster at IEEE Symposium on Field-Programmable Custom Computing Machines, April 2007, Napa CA.

[4] Borgatti, M., et al. 2003. A Reconfigurable System Featuring Dynamically Extensible Embedded Microprocessor, FPGA, and Customizable I/O. IEEE Journal of Solid-State Circuits, March 2003, Vol. 38, pp 521-529.

[5] Carrillo, J. E., Chow, P. 2001. The Effect of Reconfigurable Units in Superscalar Processors. Proceedings of International Symposium on Field-Programmable Gate Arrays, pp. 141-150, February 2001.

[6] Clark, N., Blome, J., Chu, M., Mahlke, S., Biles, S., Flautner, K. 2005. An Architecture Framework for Transparent Instruction Set Customization in Embedded Processors. Proceedings of International Symposium on Computer Architecture, pp. 272-283.

[7] Estrin, G. 1960. Organization of computer systems: The fixed plus variable structure computer. Proceedings of Western Joint Computer Conference, pp 33-40, New York 1960.

[8] Forin, A., D. Golub and B. Bershad. 1991. An I/O System for Mach 3.0. Proceedings of the First USENIX Conference on Mach, 1991.

[9] Hauck, S. et al. 2004. The Chimaera Reconfigurable Functional Unit. IEEE Transactions on VLSI, 2004.

[10] Hauser, J. R., Wawrzynek, J. 1997. Garp: A MIPS Processor with a Reconfigurable Coprocessor. Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines, pp 12-21, April 1997.

[11] Herrig, H., W. et al. 1989. Method and apparatus for controlled removal and insertion of circuit modules. US Patent No. 4835737, May 1989.

[12] Intel Corp. 1985. 8086 16-Bit HMOS Microprocessor Order No. 231455-003 September 1985.

[13] Intel Corp. 1989/ Microprocessor and Peripheral Handbook, Volume I – Microprocessor ISBN 1-55512-041-5, pp. 3.1-56, 1989.

[14] Lau, D., Pritchard, O., Molson, P. 2006. Automated Generation of Hardware Accelerators with Direct Memory Access from ANSI/ISO Standard C Functions. Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines, pp. 45-54, April 2006.

[15] Lysecky, R., Vahid, F. 2004. A Configurable Logic Architecture for Dynamic Hardware/Software Partitioning. DATE, 2004.

[16] Mitrionics, Inc. 2001. http://www.mitrionics.com

[17] Nielsen, M., J., K., Titan System Manual Digital Equipment Corporation WRL, August 1988.

[18] PCI Hot-Plug Specification Revision 1.1, June 20, 2001. http://www.pcisig.com

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

[20] Razdan, R., Smith, M. D. 1994. High-Performance Microarchitectures with Hardware-Programmable Functional Units. 27th ISM, pp. 172-180, November 1994.

[21] SRC Computers Inc. 1996. http://www.srccomp.com

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

[23] Tarari, Inc. 2002. http://www.tarari.com

[24] Wittig, R. D., Chow, P. 1996. OneChip: An FPGA Processor with Reconfigurable Logic. Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines, pp. 126-135, 1996.

[25] Xilinx, Inc. Using Partial Reconfiguration to Time Share Device Resources in Virtex II and Virtex II Pro. Xilinx Inc., May 2005

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