

Hardware Support for Process Protection in Existing Processors and Processors with reconfigurable hardwares.

Jay Hoon Byun(jaybyun@uiuc.edu)

May 11, 1998

1 Introduction

Trends in semiconductor and VLSI/CAD technology suggest that having reconfigurable logic blocks in key components of the processor to realize an application-adaptive architecture will not only be feasible but desirable in the future. These processors will allow customized mechanisms, bindings, and policies per application to overcome the performance fragility in today's rigid microprocessors [1]. But having reconfigurability in hardware would mean that the underlying hardware is more exposed to and under the control of software systems, which raises many issues concerning safety and validity of the systems with hardwares in this class. One of them would be ensuring process isolation and protection in multiuser/multiprocess systems if reconfigurable microprocessors are to become mainstream components of future systems. We would need to develop an architectural framework for safe reconfiguration. Solutions are likely to include a mix of formal verification and limiting the reconfigurability to regions of functionality that can be contained. This report presents an initial investigation for developing such a safe architectural framework. We begin by looking at what hardware support mechanisms there are in existing RISC microprocessors such as MIPS processors and ARM processors and then discuss these mechanisms with respect to reconfigurable hardwares.

2 Hardware support for process protection in R10000

Process protection scheme in R10000 can and should be implemented through *System Control Processor(CP0)* operations(kernel mode privileged instructions) which is the implementation specific(in our case, R10000) extension to the general MIPS IV instruction set architecture. Following the RISC philosophy, there are only a few CP0 instructions which are **MOVE** instructions to set the CP0 registers, namely the **DMFC0**, **DMTC0**, **MFC0**, **MTC0** instructions. It is the registers set and read by these instructions and other hardware events that control the processor state and report its status. Hence, much of the description of hardware support for process protection is given with respect to CP0 register along with the CP0 instructions [2].

2.1 Processor Mode

There are three operating modes for R10000, i.e. *Kernel mode*, *Supervisor mode*, and *User mode*. In order to use the aforementioned CP0 instructions (and access/change the CP0 registers), the processor has to be in Kernel mode or the *Coprocessor Usable* bit in the *Status* register(a CP0 register) has to be set.

Register No.	Register Name	Description
2	EntryLo0	Low half of TLB entry for even VPN(Physical Page No.)
3	EntryLo1	Low half of TLB entry for odd VPN(Physical Page No.)
4	Context	Pointer to kernel virtual PTE table in 32-bit addressing mode
8	BadVaddr	Bad virtual address
10	EntryHi	High half of TLB entry(Virtual Page No. & ASID)
12	Status	Processor Status Register
13	Cause	Cause of the last exception taken
14	EPC	Exception Program Counter
18	WatchLo	Memory reference trap address(low bits)
19	WatchHi	Memory reference trap address(high bits)
20	XContext	Pointer to kernel virtual PTE table in 64-bit addressing mode
30	ErrorEPC	Error Exception Program Counter

Table 1: Important CP0 Registers

OpCode	Description
CACHE	Cache Operation
DMFC0	Doubleword Move From CP0
DMTC0	Doubleword Move To CP0
ERET	Exception Return
MFC0	Move From CP0
MTC0	Movet To CP0
TLBP	Probe TLB for Matching Entry
TLBR	Read Indexed TLB Entry
TLBWI	Write Indexed TLB Entry
TLBWR	Write Random TLB Entry

Table 2: CP0 Instructions

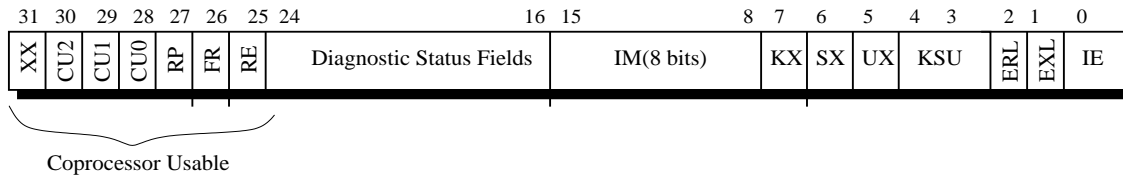


Figure 1: The Status CP0 Register

If a user process tries to alter the processor state, which has to be through CP0 privileged instructions, the processor will raise a **Coprocessor 0 Unuable exception** after which the OS will generally hand that process a fatal UNIX SIGILL/ILL_PRIVIN_FAULT (illegal instruction/privileged instruction fault) signal. Thus the system is protected from a malignant user process trying to alter processor state including the processor operation mode.

Field	Description
KSU	Mode bits 11 -> Undefined(implemented as User mode) 10 -> User 01 -> Supervisor 00 -> Kernel
ERL	Error Level; set by the processor when Reset, Soft Reset, NMI, or Cache Error exception are taken 0 -> normal 1 -> error
EXL	Exception Level; set by the processor when any exception other than Reset, Soft Reset, NMI, or Cache Error exception are taken 0 -> normal 1 -> exception

Table 3: Fields in the Status Register

The **Status** CP0 register controls processor state including processor mode, address size mode. Also note that the **Status** register contains the **Interrupt Mask** so that the **Interrupt Mask** can only be altered in *Kernel* mode using **CP0 Move** instructions.

The processor mode can be switched when:

1. **Status** register's **KSU** field is changed through CP0 instructions.
2. Processor is handling an error(the **ERL** bit in status register is set) or an exception(the **EXL** bit in status register is set). It is forced into **Kernel** mode.

Different processor modes have access to different virtual memory address space so that kernel data stored in kernel segments cannot be altered by processes with lower privilege. This will be described in the next section.

When the system is started up, the system is in Error level(**ERL** bit is set) so the processor is in the kernel mode. The cold-reset exception handler will then boot strap the OS, which will run in kernel mode. More on exception handling is given in the later section.

2.2 Process Isolation(Memory Space Isolation)

The hardware mechanism to support memory space isolation/protection is centered around the TLB as we shall see in the following.

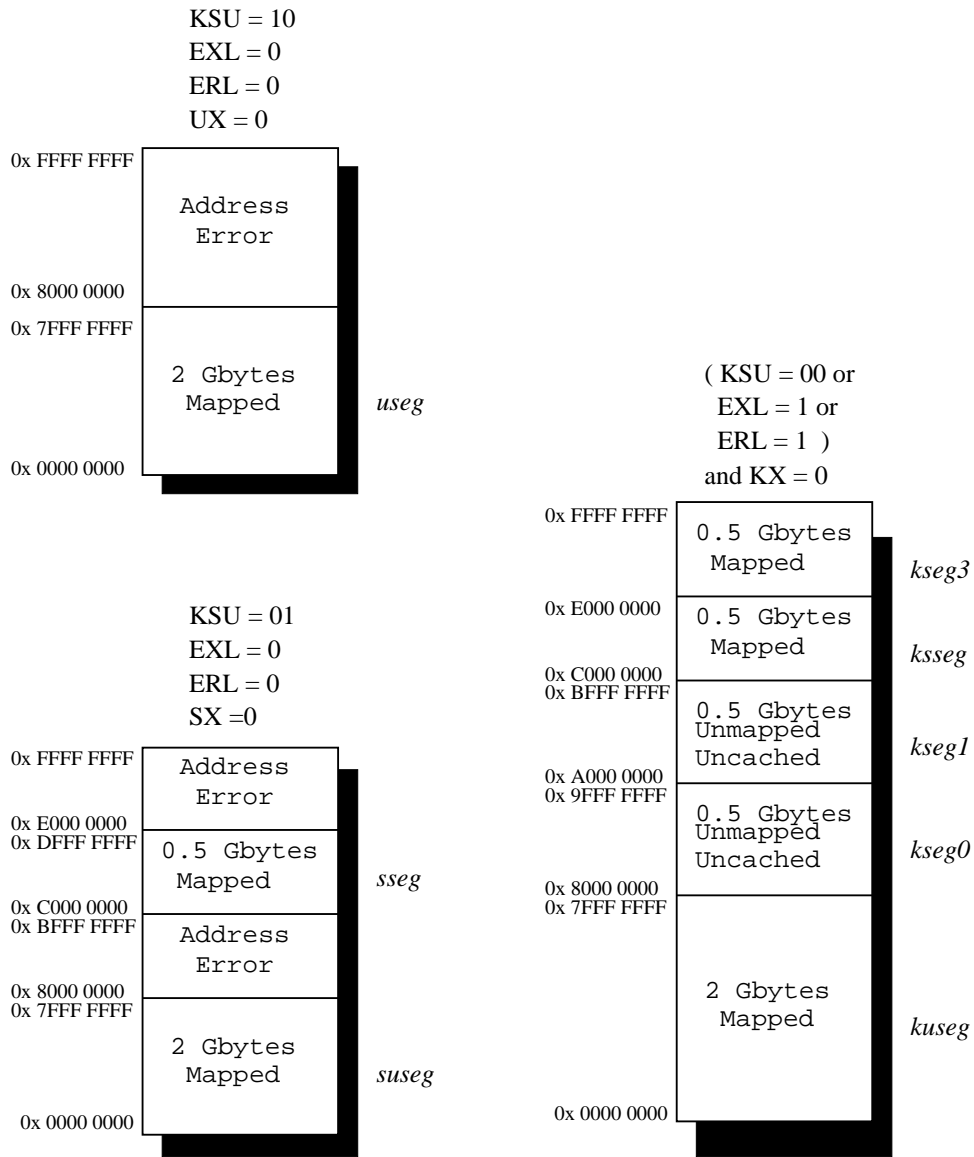


Figure 2: Address Space in each Mode(32-bit)

2.3

The processor is allowed to access only a certain subset of the memory space that can be accessed in kernel mode when it's in a lower privileged mode, as shown in figure 2 for 32-bit mode. Any attempt to reference an address not accessible in its mode will cause an **Address Error Exception**, which is service by the OS and generally hand that process a fatal UNIX SIGSEGV(segmentation violation) signal(the *Region* field in the TLB entry is used to check this violation).

Thus, the system is protected from a lower privileged mode processe tempering with Kernel data(which would reside in kernel segments *kseg0*, *kseg1*, *kseg3* above). Some kernel segments(*kseg0*, *kseg1*) even bypasses TLB for address translation or the cache for added protection.

2.3.1

For user processes that share the user mode address space, process address space isolation can be achieved when we do address translation. Support for this can be found in the 8-bit **ASID**(Address Space ID) field in the TLB entry. **ASID** is a unique id that is assigned to each process. *EntryHi* and *EntryLo* CP0 registers are the CP0 registers that is used to read/write TLB entry by the OS(e.g. when the OS has to update TLB after a TLB refill exception). After a context switch and new value is loaded into *EntryHi*,**ASID** field of each TLB entry is compared with that of *EntryHi* register and the entries are enabled if they are equal or set to global. The accesses to TLB is done only through **TLBP**, **TLBR**, **TLBWI**, **TLBWR** CP0 instructions, so user processes cannot alter the TLB directly. *Context* and *XContext* CP0 registers point to the base of the page table which reside in kernel segment *kseg3*(see fig.), so user processes cannot access the translation information stored in page tables either. PCBs, including the content of Context registers are stored in kernel segments. It still remains, though, for the OS to keep the entries in the page table consistent. It will help if we use an unused CP0 register as a page-table length register or add a bit indicating the frame is in the process's address space or not(valid-invalid bit) to prevent access to other process's page table or physical frame. Also, *WatchHi* and *WatchLo* registers are used to set the boundaries of *physical* memory trap locations. They are used primarily for debugging, however.

2.3.2

The L1 caches in R10000 are *virtually indexed* and *physically tagged*. It is virtually indexed in order to reduce access time to the cache by allowing finding set/reading tag and address translation for tag to occur concurrently. Because it is still*physically* tagged, accesses to the cache cannot by pass the TLB, where most of memory protection scheme is implemented, even though it is *virtually* indexed.

2.4 System Call and Trap instruction

The user process should use the **SYSCALL** instruction, a non-CP0 instruction, to transfer the control to the OS and get serviced for a system call. This instruction will cause a system call exception. *Code* field contains additional information on the type of the system call.

R10000 also has *trap* non-privileged instructions, namely **TGE**, **TGEU**, **TLT**, **TLTU**, **TEQ**, **TNE**, **TGEI**, **TGEUI**, **TLTI**, **TLTUI**, **TEQI**,and **TNEI**, to cause conditional trap exceptions to handle unexpected events.

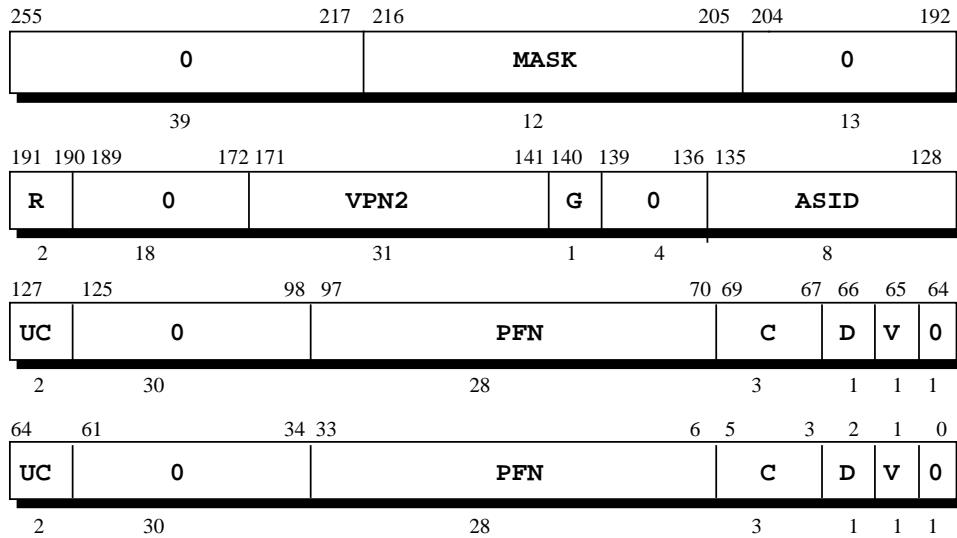


Figure 3: TLB entry

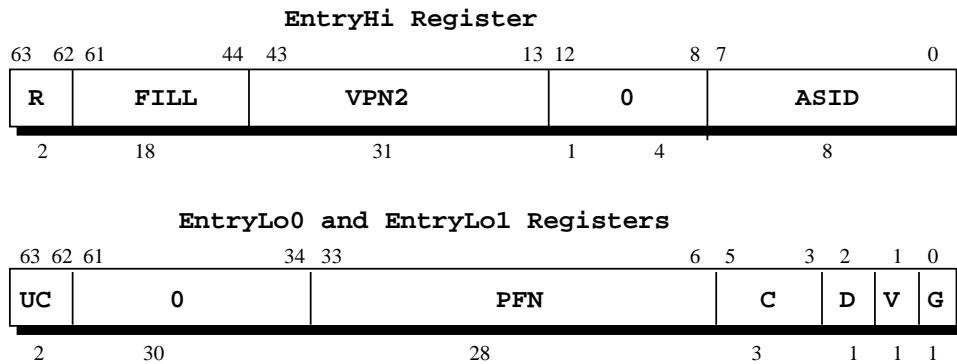


Figure 4: EntryHi, EntryLo CP0 Registers

2.5 Exception Handling

When the processor takes an exception, it is handled by the appropriate (selected by *Cause* CP0 register) vectored exception handler. It is then serviced by the OS. The **EXL** bit in the *Status* register is set when the exception is taken so the processor changes to kernel mode. And after saving the appropriate state, the exception handler typically changes the **KSU** field in the *Status* register to kernel and resets the **EXL** bit back to 0. So the OS can continue servicing exception in Kernel mode. When returning from an exception, the **ERET** CP0 instruction should be used. **ERET** instruction loads the return address from appropriate CP0 registers (*EPC* or *ErrorEPC* register), clears **EXL** or **ERL** bit, cause SC to fail if it occurred between an LL and SC.

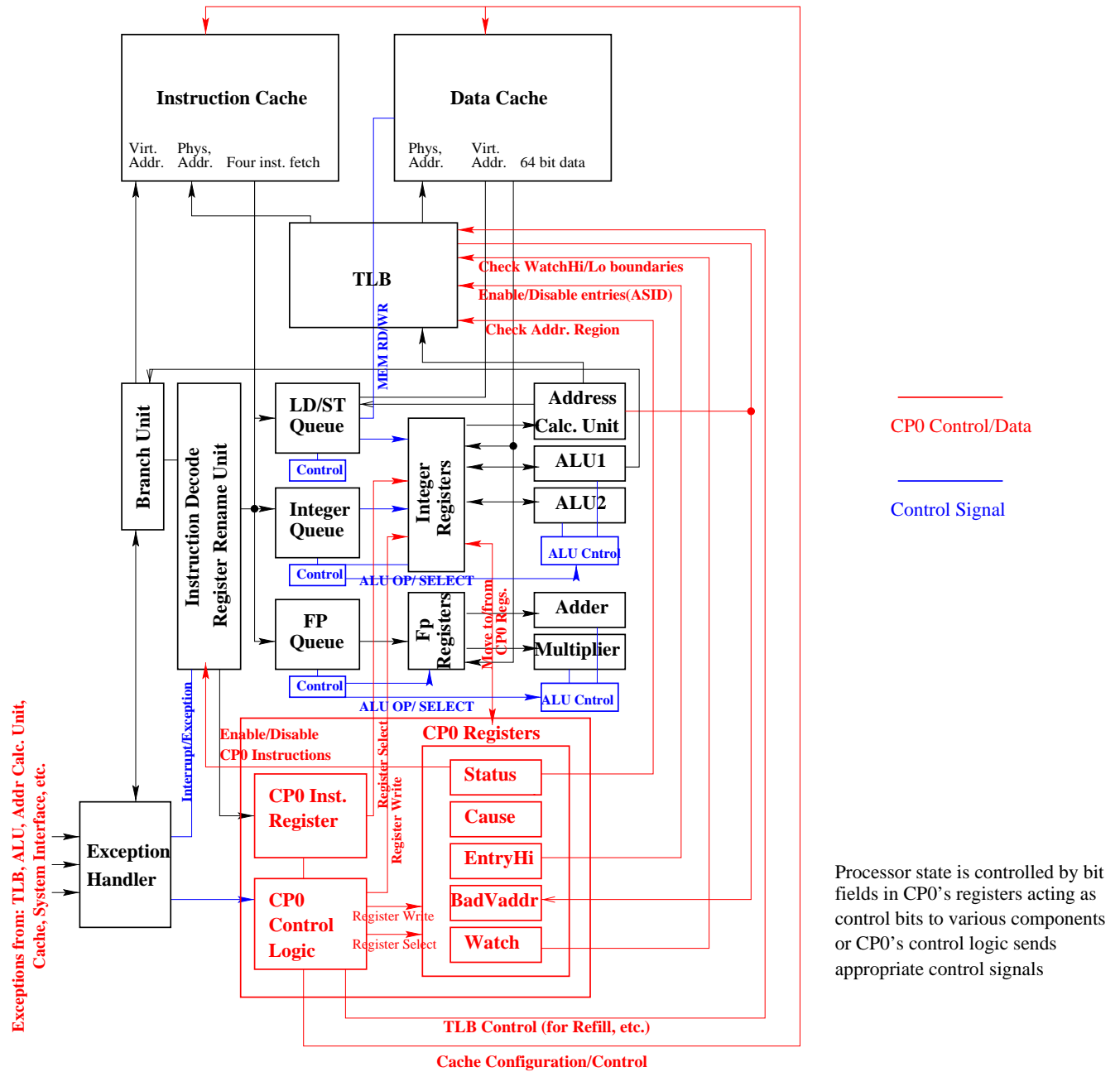


Figure 5: Logical View of Hardware Mechanism for Protection

3 Hardware support for process protection in ARM chips(or PowerPC)

4 Process Protection in Reconfigurable Hardwares

From the basic hardware mechanisms found above, we derived a logical diagram of hardware mechanisms for protection with blocks denoting each functional unit of a microprocessor. With this logical view, we discuss the relationship between the extent of reconfigurability in each block and the safety of the system. This initial analysis should serve as a starting point in developing an architectural framework for safe reconfiguration.

Reconfigurable logic can indeed go into many of the components(or replace interconnection between components). The reasoning behind this is that

1. we want to allow as much reconfigurability as we can get without being too pessimistic. For example, we would even need a reconfig. logic between the TLB and the caches if we want to optimize the address-physical memory mapping(this usually applies when we have more than one Processing element and memory element), even though TLB can be thought of a high-risk area when it comes to process/memory isolation since much of the protection and isolation of processes is done by TLB by access right checking and translation to physical memory. As long as the control interface to these two fundamental functions of TLB stays the same or stays within a predefined set of interfaces, we can have highly customizable TLB and caches while ensuring process protection.
2. part of the job of maintaining process protection/isolation and system integrity would and should be delegated to the software system(namely, the OS and the compiler) as in the conventional non-reconfigurable processor. We would be altering the reconfigurable logic when we first load the compiled code or by OS intervention in privilege mode. (But this will require the reconfigured logic-or more specifically the netlist to follow a certain template or rules to prove correctness to the OS or the compiler.)

In light of this fact,

- the caches(and interface between cache and memory/ L1 cache and L2 cache) can and should include reconfigurable logic. In fact, the cache may be the component that should have the most reconfigurability since optimizing it will significantly reduce latencies and communication overhead. Customized prefetching, varying block size, adaptive coherence protocols, and many other mechanisms can be implemented in the reconfig. logic in the cache. Surprisingly, customizable cache structure doesn't seem to present protection problems as mentioned above, provided the implementation of cache is correct and control interface stays the same.
- there could be a reconfigurable logic in the TLB and between the TLB and the caches as mentioned above.
- there can be reconfigurable logic where one of the ALUs should be in order to map a specific function such as complex floating point computation or as a DSP. It is not clear yet how the control logic interface to these units should be.
- branch prediction unit can be customized.
- Though it is possible for control logics that control the datapath and the sequence of instructions(instruction decode unit, controls in instruction queues, and hardware scheduling units)

to have reconfigurability, it is safe not to allow it or allow it to be customized to one of a few predefined scheduling schemes(e.g. for dynamic disambiguation schemes for LD/ST queue.)

- control logic that handles exception signals can NOT have reconfigurability. The mapping to/from coprocessor system registers and exception control logic should NOT be reconfigurable.
- The interface for control for caches and TLB should be fixed or stay within a predefined set, however.
- In a modern-day dynamically scheduled superscalar processors, control tend to be distributed so it is hard to grasp a integral view of the control in the processor. It is even harder to figure out how the controls to all the components can be integrated in reconfigurable processors. This is an important point not yet organized.

5 Conclusions and Future Works

From a logical view or hardware mechanisms for protection derived from mechanisms in existing processors, we tried to indentify the "choke" points for maintaining multiprocess/user safety in a highlevel viewpoint. We now need to refine and verify the "controlled" adaptation in those control points in a runtime/operating environment simulation(possibly with SimOs) and feedback cycle. Also, we need to speculate on how to make reconfiguration mechanism itself fit into the overall control for process protection. Simplest way to view this would be to treat reconfiguration mechanism as similar to context switching which is under the control of the process protection mechanism. There still remains many challenges in making a processor with reconfigurability multiprocess/multiuser safe since it implies an expansion of the software-hardware interface where more of the underlying hardware is exposed to and in the hands of the software. Also because we do not want to limit reconfigurability in exchange for system safety and lose what we've gained by having an adaptive hardware. The development of the framework for safe adaptation should therefore grow with the designing and the implementation of a processor with adaptive hardware. What needs to be studied more concerning multiprocess/user safety that was not mentioned here is the software systems, namely OS and compilers, which have even more impact on the architecture than current systems.

References

- [1] Andrew A. Chien and Rajesh K. Gupta. Morph: A system architecture for robust high performance using customization. Project Report for NSF Award.
- [2] MIPS Technologies, Inc. *MIPS R10000 Microprocessor User's Manual*, 1996.