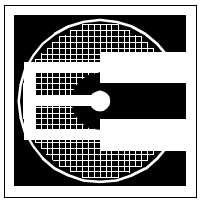


Project 3: Precise Interrupts (15%)

ENEE 646: Digital Computer Design, Fall 2008
Assigned: Thursday, October 24; Due: Thursday, November 20

1. Purpose

This project is intended to help you understand in detail how a modern microprocessor operates in concert with an operating system. You will build a *precise-interrupt facility* into your pipeline, you will add support for memory management via a *translation lookaside buffer (TLB)*, and, using RiSC-16 assembly code, you will write a software *TLB-miss handler*—the heart of a typical virtual memory system, which happens to be one of the most fundamental services that a modern operating system provides. Therefore, you will see the interaction between OS-level software and specialized control hardware (e.g. control registers and TLBs, as opposed to simple instruction-execution hardware), and you will see how the OS uses and responds to *interrupts*—arguably the fundamental building block of today's multitasking systems.

2. Precise Interrupts in Pipelined Computers

The new & improved RiSC-16 pipeline is shown in Fig. 1 on the next page. In the figure, shaded boxes represent clocked registers; thick lines represent 16-bit buses; thin lines represent smaller data paths; and dotted lines represent control paths. The pipeline is slightly different from the one illustrated and described in the previous project, reflecting the following changes:

1. The TLB is added; it has two ports: one for instruction fetch, and one for data-memory access.

2. Support for detecting and handling precise interrupts has been added by the creation of a 7-bit exception register (labeled *EXC* in the figure) in pipeline registers IF/ID through MEM/WB. Also, the instruction's PC is maintained all the way to the MEM/WB register. If a stage's *EXC* register is non-zero, the corresponding instruction is interpreted as having raised an exception. The pipeline uses these values to ensure that all instructions following an exceptional instruction become NOPs: if there is an exception in the writeback stage, all other instructions in the pipe should be squashed.

3. CTL_1 now represents control logic for handling exceptions and interrupts in the writeback stage. When a non-zero value is present in the MEMWB_exc register, all pipeline registers except for the program counter are reset, and the program counter latches the value coming from memory port 2, which corresponds to the contents of the corresponding entry in the interrupt-vector table.

4. CTL_8 is new in the memory-access stage; it handles the interaction of exceptional instructions and memory access. For instance, MEMWB_exc should get a non-zero value if either EXMEM_exc is non-zero or the data access (assuming LW or SW) raises an exception. In addition, it should set the new pipeline register MEMWB_ifx (which stands for *instruction-fetch exception*, indicating that the TLB-miss exception, if present, was generated in the fetch stage and not the memory stage.

5. CTL_9 is new in the memory-access stage; it handles TLB_WRITE events.

6. CTL_0 is new in the fetch stage; it handles TLB misses by inserting the appropriate exception code into IFID_exc when a TLB miss occurs.

As mentioned in class, interrupts must be handled in the writeback stage, otherwise it might be possible for interrupts to be handled out-of-order if back-to-back instructions cause exceptions but do so in differ-

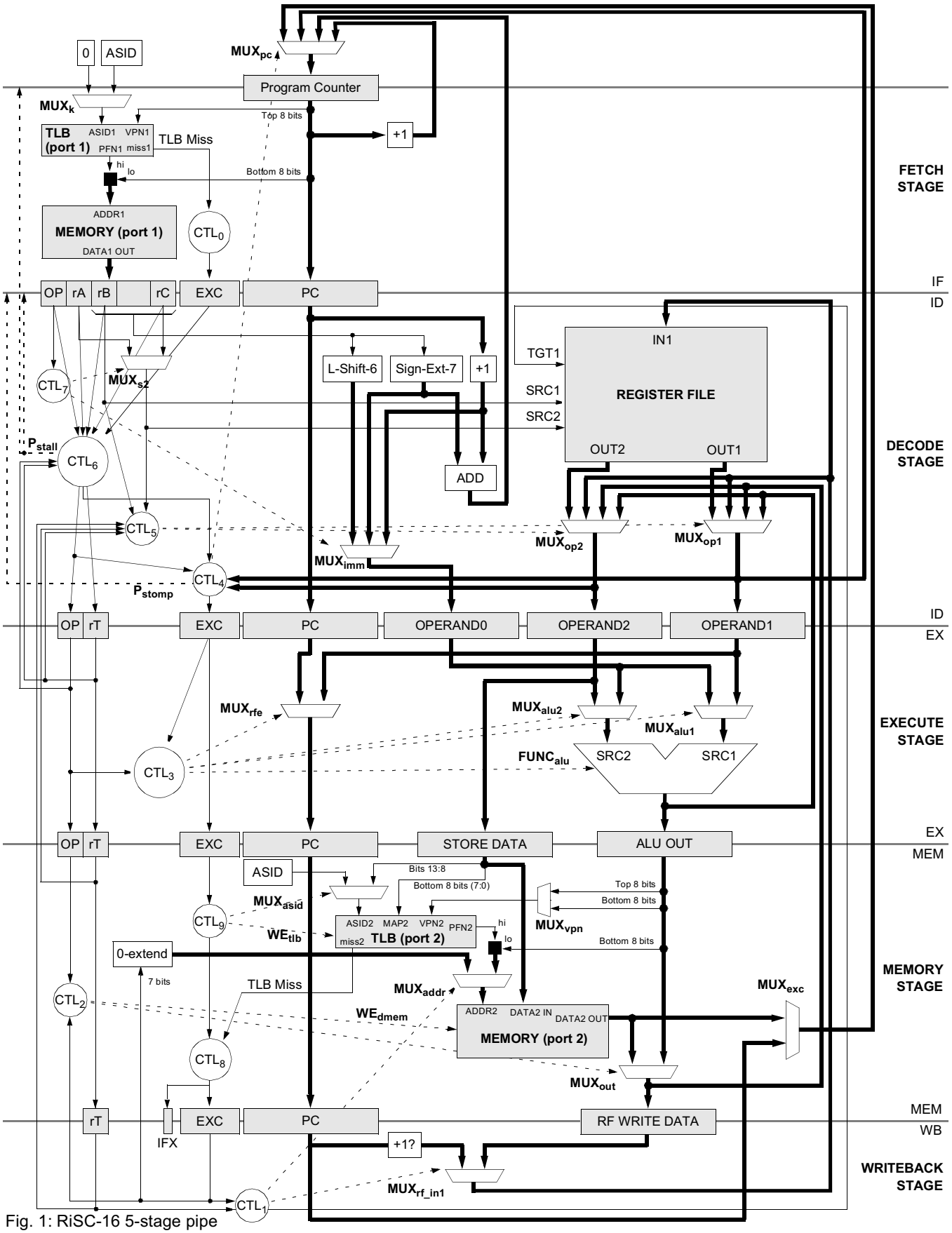


Fig. 1: RiSC-16 5-stage pipe

ent stages of the pipeline. If an exceptional instruction is flagged as such at the moment the exception is detected, it is safe to handle that exceptional condition during writeback because all previous instructions by that time have finished execution and committed their state to the machine.

For this to work, the following things must happen in the pipeline:

1. Once an exceptional condition is detected in the writeback stage, all instructions in the pipeline behind the exceptional instruction are turned into NOPs, and a NOP is inserted into IFID instead of whatever was fetched during that cycle.

2. Each pipeline stage must suspend normal operation if the instruction has caused an exception and the pipeline stage modifies machine state: for example, do not write to memory if the instruction caused a privilege violation in a previous stage—this is indicated by CTL_2 taking into account the value in MEMWB_exc. Each stage also forwards the exception code on to the following pipeline stage. If the instruction has not already caused an exception but does so during the stage in question, the *EXC* field in the following pipeline register must be set appropriately.

Otherwise, pipeline operation is as normal. In the simplest form of an exceptional condition, when an exceptional instruction reaches the writeback stage, the following steps are performed by the hardware:

1. The PC of the exceptional instruction, or perhaps the instruction *after* the exceptional instruction (PC+1) is saved in the EPC control register *(exceptional PC)*.

2. The exception type is used as an index into the *interrupt vector table (IVT)*, located at physical address 80, and the vector corresponding to the exception type is loaded into the program counter. This is known as vectoring to the exception/interrupt handler.

Some exceptions cause the hardware to perform additional steps before vectoring to the handler. For instance, you will implement a TLB-miss exception facility and handler routine. In addition to the steps listed above, before vectoring to the handler, the hardware will create an address for the handler to use.

The reason hardware might store PC+1 and not PC in EPC is that some exception-raising instructions should be "retried" at the end of a handler's execution (e.g., an instruction that causes a TLB miss), while others should be jumped over (e.g., TRAP instructions that invoke the operating system—jumping back to a TRAP instruction will simply re-invoke the trap and so cause an endless loop). If the exceptional instruction should be re-tried, the handler jumps to PC; if the exceptional instruction should not be re-executed or retried, the handler jumps to PC+1. Thus, EPC must have the correct value.

The general form of a RiSC-16 exception/interrupt handler looks like the following:

1. Save the EPC in a safe location. This is done in case another exception or interrupt occurs before the handler has completed execution.

2. Handle the exception/interrupt.

3. Reload the EPC.

4. Return the processor to user/unprivileged mode and jump to the (modified) EPC.

Most architectures have a facility ("return-from-exception") that performs step 4 in an atomic manner.

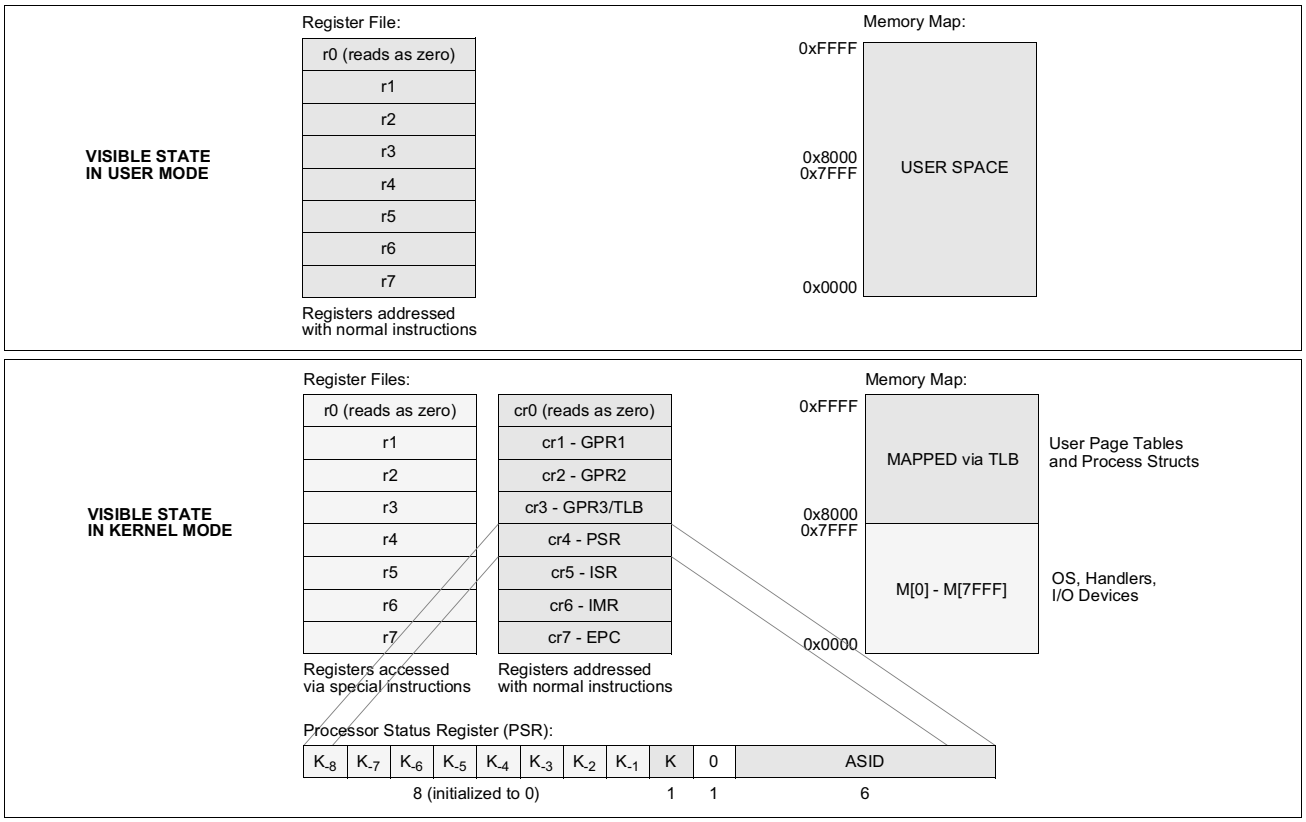


Fig. 2: Registers and memory maps visible in privileged and non-privileged modes

3. Extended RiSC-16 ISA

We must extend the RiSC-16 instruction-set architecture with system-level mechanisms. The extensions are the usual facilities found in any microarchitecture that supports operating systems and privileged mode. We need a way to protect the operating system from user processes; we need a way to distinguish between processes; we need a way to translate virtual addresses; we need an exception-handling facility; the operating system needs some control registers and would do well to have a set of general-purpose registers that it can use without disturbing user processes (otherwise, it would have to save/restore the entire register state on every exception, interrupt, or trap); etc. Briefly, the extensions include the following:

1. Addition of an exception/interrupt-handling facility, as well as a mechanism that allows software to directly enter the machine into an exceptional state—for example, traps. Some of the "exceptions" that this mechanism supports are actually privileged instructions that the machine handles at the time of instruction execution, instead of vectoring to a software handler routine. This includes TLB handling routines, the HALT instruction, etc.

2. Addition of a privileged kernel mode that is activated upon handling an exception or interrupt or upon handling a TRAP instruction, which raises an exception.

3. Addition of a set of 8 control registers that are used while in privileged mode. These are shown in Figure 2. "GPR" refers to a general-purpose register. The *processor status register (PSR)* contains mode bits that directly influence processor operation. The *interrupt service register (ISR)* indicates what interrupts have been received by the processor. The *interrupt mask register (IMR)* allows software to ignore selected interrupts. The *exceptional program counter (EPC)* register is filled by hardware when vectoring to a handler and indicates the PC of the exceptional instruction. Access to the

general-purpose register file is still possible while in kernel mode through special instructions (which are not necessary for this assignment).

4. Addition of a translation lookaside buffer (TLB) that performs address translation. For this project, the TLB will have two (2) entries and be fully associative, with a random replacement policy. On every TLB write, consult the counter bit to choose which TLB entry to replace.

5. The definition of a *memory map* that delineates portions of the virtual space as mapped through the TLBs, accessible in kernel mode only, etc. This is illustrated in the figure above. In user mode, all of the address space is mapped through the TLB (all virtual addresses are first translated by the TLB before being used to reference memory locations; note this implies that all virtual addresses are valid in user mode). In kernel mode, the top half of the address space is mapped through the TLB, and the bottom half is not: this means that addresses in this region, while the computer is in privileged kernel mode, will be sent directly to the memory system without first being translated.

6. Addition of the concept of an *address-space identifier (ASID)*. While this is not strictly necessary, it does simplify the virtual memory mechanisms and organization. The ASID s used to distinguish between different process that might run on the machine, and its use allows state from many different processes to reside in the TLB and cache at the same time (otherwise the TLB and, at least potentially, the cache as well would have to be flushed on context switches). ASID 0 is interpreted by the hardware to indicate the kernel, executing in privileged mode. When the processor is in kernel mode, instruction fetch will always use ASID 0, and data-memory access will use whatever ASID is in the processor status register. This last mechanism allows the operating system to read and write locations within different user address spaces (i.e., “masquerade” as different processes), but it prevents the operating system from executing instructions from unprivileged processes (which might otherwise constitute a security hole).

7. Definition of some memory-management constructs, including the user page table organization. Having hardware define this structure is beneficial in that the hardware can quickly generate the address that the TLB-miss handler needs to locate the PTE. In many systems, this limits flexibility because the hardware dictates a page table format to the operating system. However, if desired, software can always ignore this address (treat it as a “hint” that need not be followed) and implement whatever page table it wants. Note that, in such a scenario, the operating system would then have to generate its own PTE addresses without support from hardware.

3.1 Terminology

Following Motorola’s terminology, we will distinguish two classes of exceptional conditions: those stemming from internal actions (*exceptions*) and those stemming from external actions (*interrupts*). For instance, the following interrupt types are defined:

```
INT_CLOCK
INT_TIMER
INT_IO
```

Though this is not a particularly exhaustive list, it is more than enough for the purposes of this project, in which we will not even bother with interrupts, save possibly the timer or clock interrupt, both of which could be used to debug the system. In addition, several exception types are defined, including:

```
EXC_TLBUMISS
EXC_TLBKMISS
EXC_INVALIDOPCODE
EXC_PRIVILEGES
```

Corresponding interrupt/exception numbers are listed in the following section. Each interrupt or exception type corresponds to a single vector point and thus a single handler routine.

Another type of exception (another class of internally generated exceptional condition) is a TRAP. For TRAP instructions, there is a whole class of trap types that can implement various operating-system routines, because the trap type is interpreted by the hardware to indicate a particular vector, just like each exception and interrupt type has a separate vector. Thus one can think of *trap* as equal in stature to *interrupt* or *exception*. Each of the trap vectors is OS-defined (e.g., TRAP 1 can mean *read* or *write* or *open* or *close* ...); the hardware simply vectors to the corresponding handler, so the operating system can attach arbitrary semantics to each of the trap handlers. The most noticeable effect of using this style of mechanism is to reduce the register-file pressure in handling system calls. Note that this is unlike most architectures, in which there is a single TRAP exception and all system calls are vectored through the same exception, and the user code first places the trap type into a user-visible register for the operating system to read once the handler runs. Our implementation is different not for any specific reason, but rather to explore the OS design space.

3.2 New Instructions

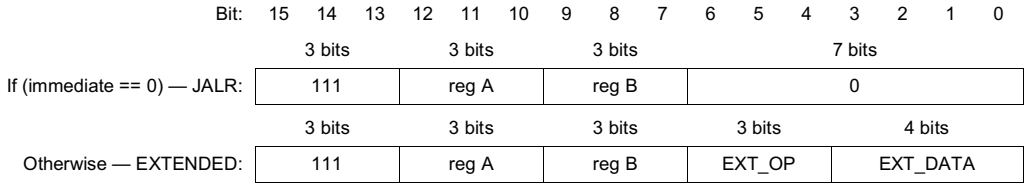
As mentioned, the instruction set has changed, now that software can insert exceptions directly into the pipeline and can execute a number of new privileged instructions. Additional instructions are given in the table below. Pseudo-instructions are generated by the compiler; privileged instructions are a new class.

Assembly-Code Format	Meaning
PSEUDO-INSTRUCTIONS:	
`nop`	do nothing (add r0, r0, r0)
`trap type`	trap to the operating system with vector *type*
`halt` *or* `trap TRAP_HALT`	ask operating system to stop machine & print state
`lli regA, immed`	R[regA] <- R[regA] + (immed & 0x3f)
`movi regA, immed`	R[regA] <- immed, implemented as lui/lli combo
`.fill immed`	initialized data with value *immed*
`.space immed`	zero-filled data array of size *immed*
PRIVILEGED INSTRUCTIONS:	
`tlbw regA, regB`	write TLB entry (held in regA and regB) to the TLB: bottom 8 bits of regA contain the PFN; bits 9–14 of regB contain ASID; bottom 8 bits of regB contain the VPN; all other bits in regA and regB are ignored
`rfe regB`	return from exception: waits until writeback stage to jump through regB; simultaneously returns processor to previously stored mode (does not save the link pointer)
`sys class`	cause exceptional condition of specified class (inserts the value of “class” directly into IDEX.exc; good for testing & debugging)
`sys MODE_HALT`	an example of the previous instruction: stop machine & print state

These and related modifications/extensions are described in more detail in the following sections.

3.3 Privileged Instructions (and TRAP)

As shown in the figure below, the JALR instruction is overlapped with a number of other instructions.



If the immediate field of the instruction is zero, the JALR is treated normally. Otherwise, the bottommost 7 bits of the instruction are treated as an opcode and associated data for a system-level instruction other than a JALR. In this case, the extended opcode EXT_OP takes precedence over the JALR opcode—i.e., the instruction does not perform a jump and link but rather performs the operation specified by EXT_OP.

For these non-JALR instructions, we have two classes (privileged system-level operations, and the raising of exceptional conditions through user-level TRAP instructions), each of which has the following timing:

1. **Exceptions, interrupts,** and **traps** as well as all MODE changes (for example, MODE_HALT and RFE, which changes the kernel/user mode bit) are shaded in the following table and handled in the **writeback stage** to prevent unwanted interaction with earlier instructions in the pipeline.

2. **Privileged operations** except for MODE changes are left unshaded in the following table and are handled wherever is most appropriate: for instance, the TLB-management instructions are handled in the **memory stage** because doing so guarantees that multiple data-TLB accesses will not happen (i.e., the TLB will not be probed for a data address at the same time that the TLB-WRITE instruction is writing a new entry into the TLB).

These extended opcodes are used to directly affect change in the operation of the hardware; many of them insert exceptional conditions into the pipeline, but some are used to move data around the system. All are privileged operations (i.e. they require that the processor be in kernel mode) except for the TRAP instructions, which simply invoke the operating system and thus enable kernel mode securely. Note that this brings up an interesting point: the HALT instruction is now defined as a slightly more complex process than before. HALT is now defined as one of the several modes of execution (including RUN and SLEEP), and putting the processor in a specific mode is a privileged action—user code cannot HALT the processor. Thus, user code must ask the operating system to perform a HALT (which allows the graceful shutdown of the machine, were there a file system or something similar attached). Thus, HALT is now a two-stage process: first, user code calls a TRAP instruction with HALT as the argument. This causes an exceptional condition, and the machine vectors to the operating system's corresponding TRAP handler, which in turn cleans up any system state necessary (not needed in this implementation) and then calls the MODE_HALT instruction.

Note that (for the extent of this project) in all cases but TLB_WRITE and SYS_RFE, the rA and rB fields of the JALR instruction are ignored. These two instructions, however, **do** use the rB and/or rA fields of the instruction, specifying a register to read and/or write. RFE uses the rB field to provide a jump address taken from the register file. TLB_WRITE constructs a TLB entry from two different 16-bit words read from the register file (i.e., it uses both rA and rB).

The table on the following page describes the various extended opcodes and their associated possible data values. Items that are shaded in the table represent conditions that cause hardware to vector to a software

routine; all other items are essentially instructions that the hardware executes, just like ADD, ADDI, NAND, etc. Those mechanisms that your Project 3 implementation must support are in **bold**.

Opcode Extension (EXT_OP)	Data Extension (EXT_DATA)	Semantics
SYS_MODE (000)	MODE_RUN (0) MODE_SLEEP (1) **MODE_HALT (2)** MODE_RFU3 .. MODE_RFU7 (3 .. 7) MODE_PANIC8 .. MODE_PANIC15 (8 .. 15)	Normal mode—ignore (equivalent to JALR) Low-power doze mode, awakened by interrupt **Halt machine** No definition yet Halt and output panic value (8..15) (meaning is software-defined)
SYS_TLB (001)	TLB_READ (0) **TLB_WRITE (1)** TLB_CLEAR (2)	Probe TLB for PTE matching VPN in rB **Write contents of rB to TLB (random)** Clear contents of TLB
SYS_CRMOVE (010)	Top bit specifies to/from Bottom 3 bits identify CR#	Moves a value to/from the control registers from/to the general-purpose registers
SYS_RFE (011)	**Data value ignored**	***Return From Exception*: JUMP (without link) to address held in rB (a control register), and right-shift (zero-fill) the kmode history vector**
SYS_RESERVED (100)		Has no definition yet
SYS_EXCEPTION (101)	EXC_GENERAL (0) **EXC_TLBUMISS (1)** **EXC_TLBKMISS (2)** EXC_INVALIDOPCODE (3) EXC_INVALIDADDR (4) EXC_PRIVILEGES (5)	General exception vector **User address caused TLB miss** **Kernel address caused TLB miss** Opcode the execute stage does not recognize Memory address is out of valid range Decoded privileged instruction in user mode
SYS_INTERRUPT (110)	INT_IO (0) INT_CLOCK (1) INT_TIMER (2)	General I/O interrupt Used to synchronize with external real-time clock Raised by a watchdog timer
SYS_TRAP (111)	TRAP_GENERAL (0) **TRAP_HALT (1)**	General operating system TRAP vector **Ask operating system to perform HALT**

Any of these exceptional conditions or extended opcodes can be invoked through the assembler, using the EXTEND opcode (looks like JALR with a non-zero immediate field). In most cases, there is no need to specify both EXT_OP and EXT_DATA because the EXT_DATA name uniquely identifies the exceptional condition or extended operation. The new assembler (can be found on the course website) supports this facility via several mechanisms:

1. First, the **sys** opcode, which takes a single value as an operand (rA and rB are both zero):

```
sys    MODE_HALT     # halts the machine
sys    INT_CLOCK     # vectors to the CLOCK interrupt handler
sys    EXC_TLBUMISS  # vectors to the TLBUMISS exception handler
sys    TRAP_HALT     # vectors to the HALT trap handler (which executes a MODE_HALT)
```

2. Second, the **ext** (EXTEND) opcode, which functions like JALR in that two registers are specified, but an additional argument is also given to be used as the immediate value. This is how the TLB_WRITE and TLB_READ functions are specified. Examples of its use:

```
ext    rA, rB, TLB_READ    # VPN to search for is in rB, match is written to rA
ext    rA, rB, TLB_WRITE   # reads entry from rB, rA is ignored
```

```
ext    rA, rB, MODE_HALT   # identical to "sys MODE_HALT" ... rA and rB are ignored
ext    rA, rB, SYS_RFE     # this is identical to "rfe rB"
```

3. Last, several of the operations are common enough to warrant their own opcodes:

```
rfe    rB                  # identical to: ext r0, rB, SYS_RFE
trap   type                # identical to: sys type
halt                       # identical to: trap HALT or sys TRAP_HALT
tlbw   rA, rB              # identical to: ext rA, rB, TLB_WRITE
```

3.4 Processor Status Register

The *processor-status register* is often considered the heart of a machine, as it contains some of the most important state information in a processor. See the earlier figure for the RiSC-16's PSR; it includes three pieces of information: (1) the 6-bit ASID of the executing process, (2) the kernel-mode bit that enables the privileged *kernel mode*, and (3) an 8-bit wide bit-array that represents a history of previous *kernel-mode* bit values.

1. The *address-space identifier (ASID)* is a number that identifies the process currently active on the CPU. This is used to extend the virtual address when accessing the TLB. Note that the operating system gets special treatment in this regard: first, ASID 0 is considered to be synonymous with kernel mode (this allows the TLB to translate kernel addresses appropriately without having to know whether the machine is in privileged mode or not). Second, the kernel can operate in kernel mode with an ASID other than 0 in the processor status register—this would allow the operating system to perform I/O operations to and from a user-level process address space. However, to prevent potential security holes, and to clearly delineate handler code and operating-system code from user-level code, instruction fetch should never proceed with a non-zero ASID if the processor is in kernel mode. For instance, when the processor vectors to a handler, the ASID of the previously running process is still in the PSR, but the handler instructions will be fetched from kernel space.

2. The K *(kernel-mode)* bit indicates whether the processor is in privileged mode or user mode; privileged instructions are only allowed to execute while the processor is in privileged mode; they cause an exception otherwise (EXC_PRIVILEGES). The memory map also changes depending on the mode: as mentioned previously, user space is mapped through the TLB; kernel space is divided into mapped and unmapped regions.

3. The K_1 through K_8 bits make up a shift register that maintains the previous eight modes of operation; every time an exception or interrupt is handled, the kernel-mode bit is left-shifted into this array (which is itself shifted to the left to accommodate the incoming bit), and the kernel most is set appropriately (usually turned on). Every invocation of the *return-from-exception* instruction right-shifts the K_1 through K_8 bit-array (while zero-filling from the left) and places the rightmost bit of the array into the kernel-mode bit. Note that if the architecture is defined such that user mode cannot be entered from kernel mode except by a *return-from-exception*, then the shift-register implementation can be replaced with a simple counter.

This implementation of maintaining previous history bits allows the hardware to handle *nested interrupts*, a facility that is extremely important in modern processors and operating systems. A nested interrupt is a situation where the hardware handles an exception or interrupt while in the middle of handling a completely different exception or interrupt. In our implementation of virtual memory, the ability to handle nested interrupts will be of crucial importance.

3.5 Control Registers, Generally

The control registers are those extra 8 registers that are visible only in kernel mode:

```
cr0 - reads as 0, read-only
cr1 - For general-purpose use
cr2 - For general-purpose use
cr3 - For general-purpose use and TLB interface
cr4 - Processor Status Register
cr5 - Interrupt Status Register
cr6 - Interrupt Mask Register
cr7 - EPC Register
```

They behave as follows:

cr0 Like rf0, this is always zero.

cr1 *(gpr1)* This register is for general-purpose use. However, if an interrupt handler is going to use the register, it should save the register's contents before writing to it and restore the contents prior to exiting, just in case the handler happened to preempt another handler using the register.

cr2 *(gpr2)* This register is for general-purpose use, just like cr1.

cr3 *(gpr3)* Another general-purpose register, with the addition that on TLBMISS exceptions the hardware places into this register the address of the mapping page-table entry. Details are presented in a later section. Because the contents of this register may be overwritten at any moment due to a TLB miss in either user or kernel mode, the kernel should use this only as a scratch register. In particular, the UMISS handler should first move the VPN into a different register before attempting to load the user PTE.

cr4 *(psr)* The *processor-status register* described earlier.

cr5 *(isr)* The *interrupt status register*, which contains a single bit for every interrupt type. Whenever an interrupt occurs, its corresponding bit in this register is set high. Thus, a simple poll of the interrupt status is possible by comparing its contents to r0; if they are equal, no interrupts have occurred. Interrupts cause the hardware to vector to a handler routine, provided that the *interrupt mask register* (described below) is not disabling the interrupt type. **Not used in this project**.

cr6 *(imr)* The *interrupt mask register*, which is set by the operating system. It defines the interrupts that the processor is allowed to handle, indicated by a '1' in the appropriate bit position. On every cycle, a bit-wise AND is performed by the hardware between the ISR and the IMR; if the result is non-zero, the hardware places the appropriate interrupt class into the IFID_exc register. **Not used in this project**.

cr7 *(epc)* The *exceptional PC*, representing the return address for the exception/interrupt handler. Hardware loads this right before vectoring to an exception, interrupt, or trap handler. The first thing that a handler should do is save this value in a safe place, just in case another handler is invoked in a preemptive manner.

These eight registers perform two separate functions: first, they provide the operating system access to mode-control registers such as the PSR and ISR; facilities similar to these are found in nearly every processor architecture in existence and are necessary for most system-level software. The second function provided to the operating system is a small set of *shadow registers* not visible to user-level processes. While not necessary for implementing most system-level software, shadow registers—such as those found in processors as diverse as the Alpha, SPARC, PA-RISC, Xscale, and M-CORE—provide the operating system a safe haven in which to operate. These registers do not need to be saved when moving to and from privileged mode, as would be necessary if the operating system shared the same register file as user-level code. For instance, the MIPS architecture has only one space of registers, and the kernel, to avoid having to save and restore user-level registers, claims two of the registers as its own: the assembler and compiler assure that user-level code does not access these registers, and they are not saved or restored

on interrupts or system calls and traps. The disadvantage of having shadow registers is either ensuring that data can be moved between the two sets of registers (e.g. by providing a larger register specifier in some or all privileged-mode instructions, or by providing a special data-move operation whose sole function is to move data between register namespaces), or ensuring that such a scenario is never needed.

As mentioned previously, these are the default registers when kernel mode is active, i.e. when the K-mode bit in the processor status register contains a '1' value. Thus, when the operating system performs instructions like the following:

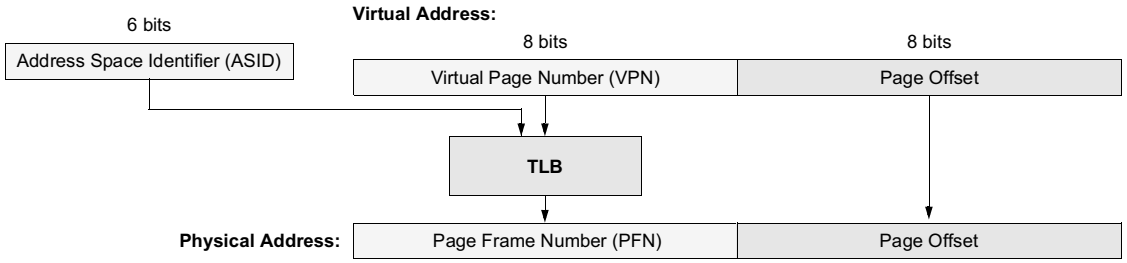
```
# kernel mode is on
add    r1, r0, r4
```

the operand values are read from control registers, and the result is written to a control register. This example moves the contents of the processor status register into cr1.

There is an obvious trade-off between implementing the sixteen registers (eight user registers plus eight control registers) as a unified register file using the kernel-mode bit as the fourth register-specifier bit, or as two separate register files selected by a MUX controlled by the kernel-mode bit. Depending on implementation, the unified design can have a faster access time; the MUXed design can have lower power consumption. We will do a unified design: the register file is now a 16-entry array of registers, as opposed to an 8-entry array. When reading or writing the register file, the top bit of the register identifier needs to come from the mode bit in the process status register—i.e., the bottom 8 registers are referenced in user mode, and the top 8 registers are referenced in kernel mode. You need to implement this connection.

3.6 Address Translation and TLBs

The memory-management implementation defines pages to be 256 words in length. Therefore, a 16-bit virtual address is composed of an 8-bit VPN and an 8-bit page offset, as illustrated in the figure below:



The figure also illustrates the mechanism of address translation: virtual addresses are translated by the TLB into physical addresses. Translation consists of nothing more than replacing the virtual page number with the corresponding page frame number. The page offset is identical in both addresses (a given word is at the same location within a page, whether the page is virtual or physical).

The TLB is a cache, usually implemented as a content addressable memory (CAM), or fully associative cache. In this implementation, the cache's tag field is the concatenation of a 6-bit ASID and an 8-bit virtual page number. The corresponding data field of the cache entry is the page frame number where the indicated virtual page can be found.

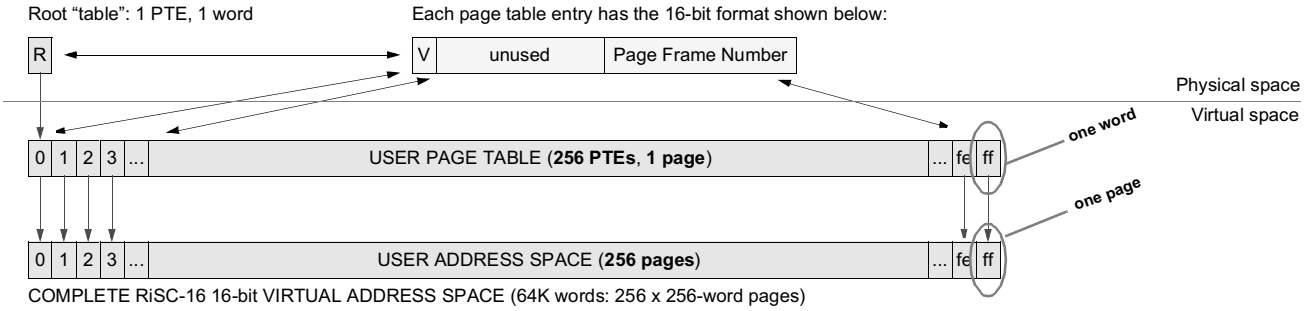
In this implementation, all user addresses from 0x0000 to 0xFFFF are mapped through the TLB. The top half of the kernel's virtual space is also mapped through the TLB. The bottom half of the kernel's address space, ranging from address 0x0000 to 0x7FFF is mapped directly onto the bottom half of main memory—the physical address equals the virtual address. Thus, references to this space cannot cause a TLB miss. This is an important consideration to remember when designing your TLB-miss handler.

The RiSC-16 has a *software-managed TLB*. This simply means that the operating system, and not the hardware, is responsible for handling *TLB refill*, the act of walking the page table on a TLB miss to find

the appropriate page-table entry and inserting it into the TLB. When the TLB fails to find a given VPN for a mappable virtual address, the TLB raises an exception, invoking the operating system. If the address being translated is a user address (i.e., if the CPU was in user mode at the time of the exception), then the exception raised is a "user miss," EXC_TLBUMISS. If the CPU is in kernel mode, and the top bit of the virtual address to be translated is a '1,' then a "kernel miss," or EXC_TLBKMISS, is raised. If in kernel mode, and the top bit of the address is '0,' the address cannot cause an exception because it is not translated through the TLB but instead is mapped directly onto physical memory.

3.7 Organization of the Page Table

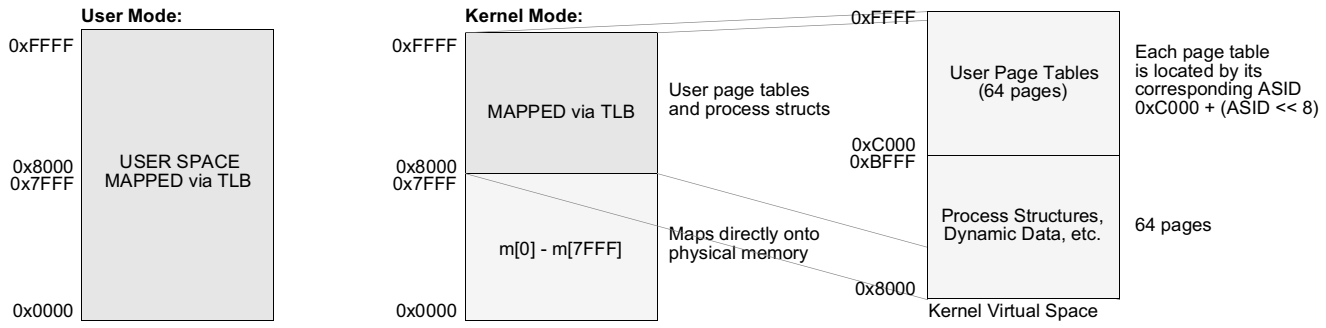
The page table format is very similar to that used in the MIPS architecture: it is a two-tiered table, where the topmost level is in physical space, wired down when the application is executing, and the lower level is pageable and addressed virtually. We will call the top level the "root" for obvious reasons and the lower level the "user page table" because it maps the user address space. The page table organization is illustrated below (note the difference in scale between the user page table and the user address space):



The full address space contains 256 pages, which requires 256 PTEs to map it. Each PTE is a single word, and 256 PTEs can thus fit in a single page. Therefore, a single page of PTEs can map the entire user space. The kernel keeps a set of pages in its virtual space, each of which holds one user page table. There are 64 of these tables (there are 64 unique ASIDs: the ASID is 6 bits wide), and the corresponding user tables are held in the top 64 virtual pages of the kernel's address space. These are in turn mapped by root PTEs that are held in the top 64 words of page frame 0. Thus, the virtual page number of the user page table is equal to the physical address of the root PTE that maps it.

As mentioned, all user addresses are translated through the TLB, and kernel space is typically divided into regions that are translated through the TLB and other regions that map directly onto physical memory. The kernel's translated regions typically hold data that is seldom used, for example the various data structures (including process page tables) that are used to keep track of the running processes. If a process is not currently running, then none of these structures are in use, and they need not occupy physical memory. Thus, it makes sense to put them into virtual space.

The different views of the 16-bit address space are shown below:



As mentioned, all user addresses are translated through the TLB, and kernel space is typically divided into regions that are translated through the TLB and other regions that map directly onto physical memory. The kernel's translated regions typically hold data that is seldom used, for example the various data structures (including process page tables) that are used to keep track of the running processes. If a process is not currently running, then none of these structures are in use, and they need not occupy physical memory. Thus, it makes sense to put them into virtual space.

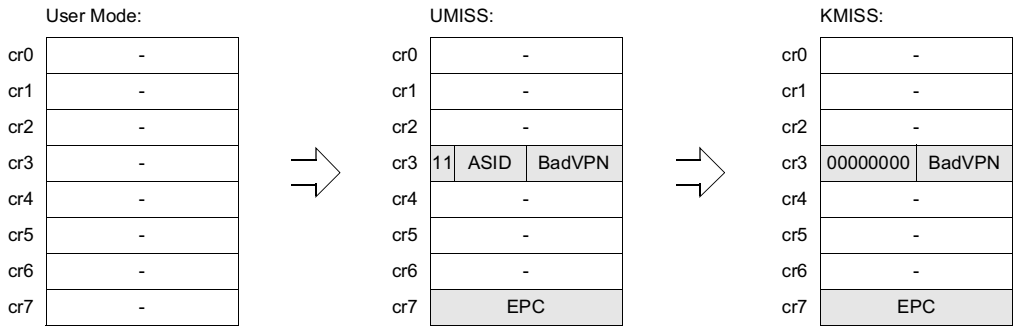
3.8 Nested TLB-Miss Exceptions & Faulting PTE Addresses

When the TLB fails to find a given VPN, it raises an exception. If the address being translated is a user address (i.e., the CPU is in user mode), then the exception raised is EXC_TLBUMISS. If the CPU is in kernel mode and the top bit of the address to be translated is a '1' then the exception raised is EXC_TLBKMISS. If the top bit of the address is '0' the address cannot cause an exception because it is not translated through the TLB but instead is mapped directly onto physical memory.

Both exceptions behave as normal and perform additional functions before vectoring to the handler. When the TLBUMISS exception handler runs, its job is to find the user page table entry corresponding to the page that missed the TLB. The user page table is located in the top quarter of the address space, as shown above. The (virtual) location of the PTE, given the ASID of the current user process and the VPN of the address that caused the TLB miss, is computed according to the following equation:

```
ADDRpte = 0xC000 + (ASID << 8) + VPN
```

To aid in the handling of the exception, the construction of this address is performed by hardware. This is similar to the memory-management facilities offered by MIPS processors and UltraSPARC processors. As soon as a TLBUMISS exception is detected, the hardware takes the VPN of the faulting address and the ASID currently stored in the process status register (PSR) and performs this computation. Because all of the values in question (number of unique ASID values, number of unique VPNs) are all powers of two, the additions in the equation above simply to ordinary concatenation of bit-fields. This is illustrated below, which shows the locations in the register file and formats of the various data that are placed into the register file *by hardware* when vectoring to a TLB-miss handler:



The address is placed into **cr3**, control register 3. After this, the hardware vectors to the UMISS handler. When the handler runs, it will use the virtual address in cr3 to reference the PTE. When the PTE is loaded, the handler obtains the PFN (see figure below for specifics on the PTE format). The handler then performs a **tlbw** (TLB write) instruction to move the loaded mapping into the TLB.

Note that the handler loads the PTE into the processor using a virtual address. Thus, it is possible for the handler itself to cause a TLB miss. This is what invokes the TLBKMISS handler.

When handling a kernel-level TLB miss (EXC_TLBKMISS), the type of TLB miss that happens while the kernel is executing, the page table needed is the kernel's own page table that maps the top half of the address space (the kernel's virtual space). This page table is illustrated in the next section; it is located at address 128 in physical memory and extends to address 255. The top half of this page table (addresses

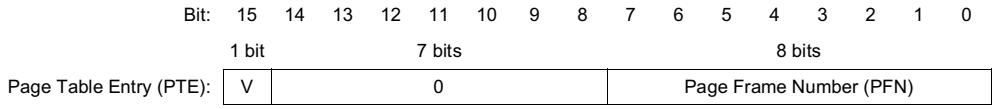
192–255) maps the user page tables referenced by the user TLB-miss handler. By construct, because the user page tables begin at virtual address 0xC000, their VPNs range from 0xC0 to 0xFF—in decimal, the range is 192–255. Therefore, by construction, the VPN of the virtual address for the user PTE that the UMISS handler loads equals the physical address of the kernel PTE that maps the user page table. Before vectoring to the KMISS handler, the hardware places this VPN (which, as described, is equal to the physical address required by the KMISS handler) into **cr3**, control register 3.

This is actually a fairly intricate process, and it demands careful attention on your part in the development of your TLB-miss handlers, otherwise data can get stepped on without the software realizing it (for instance, when the *umiss* handler causes an exception when it performs the PTE load using a virtual address).

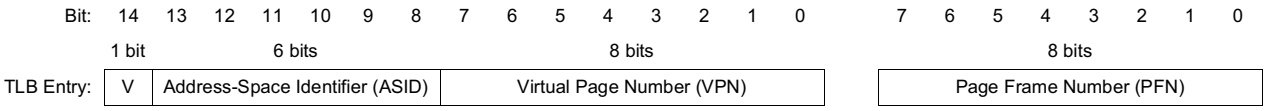
If, during user mode, an access causes a TLB miss, the *umiss* handler is invoked. The hardware places the PTE's virtual address into cr3. This is done in addition to saving the return PC in cr7, and it is all done before vectoring to the *umiss* handler.

If, during the execution of the *umiss* handler (or, in fact, during any kernel code at all), a memory access causes a TLB miss, the *kmiss* handler is invoked. The hardware places the faulting 8-bit VPN into cr3, in the lower 8 bits of the register. This happens to be the physical address of the root PTE (the root table is in physical page 0—see later diagram). The hardware places the return PC (which, in most cases will be the PC of the LW instruction in the *umiss* handler) into cr7. Then hardware vectors to the *kmiss* handler.

Once a handler has loaded a PTE, what happens next? The PTE format looks like the following:



The bottom eight bits represent the physical location of the page (the PFN); the top bit is a valid bit (0=invalid; 1=valid). Ultimately, this will be used to create a new entry in the TLB, which has the following format:



Once the PTE has been loaded, generating the TLB entry is straightforward: the handler must verify the validity of the PTE and then write it to the TLB. The TLB-write instruction, **tlbw**, takes two operands:

1. The corresponding Page Frame Number—i.e., the contents of the PTE just loaded.

2. The ASID and Virtual Page Number (i.e., the same thing as the address used to load the PTE, which the hardware stored into **cr3** at the time of vectoring to the handler).

Hardware uses the bottom 8 bits of #1 and the bottom 14 bits of #2; all else is ignored. TLB update can be accomplished with the following instructions. Assume that the load address is in r2 and the PTE is in r1.

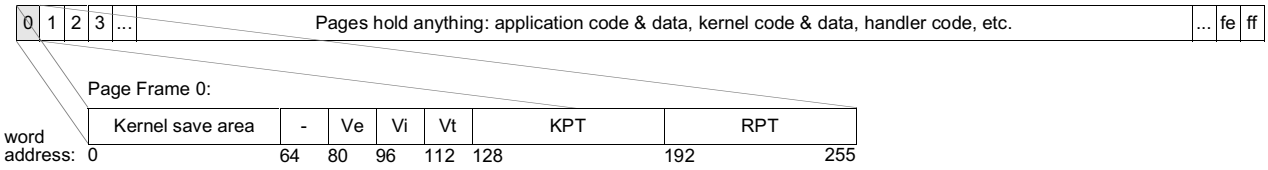
```
lui    r3, 0x8000          # will be used to test top bit
nand   r3, r3, r1          # r3=011111111111111 => val; r3=1111111111111111 => inv
nand   r3, r3, r3          # r3=0x8000 => val; r3=0x0000=>inv
bne    r3, r0, valid
# error-handling code
valid:
tlbw   r1, r2              # writes contents of r1+r2 to TLB, sets 'v' bit in TLB entry
```

The steps that the UMISS and KMISS handlers go through are very similar.

Note that this is not a complete handler: for example, error-handling is missing, the beginnings of the handlers are missing (in which register contents are saved and the PTE loaded), the ends of the handlers are missing (in which register contents are restored from memory), and return-from-exception is missing. For error-checking, you can simply HALT the machine prematurely or use a special syscall (e.g., PANIC mode), because the PTEs you reference in your page tables are predefined and should never be invalid.

3.9 Physical Memory Map

Previous figures have illustrated the layout of virtual space. The following figure illustrates the layout of *physical* memory, including the all-important first page:



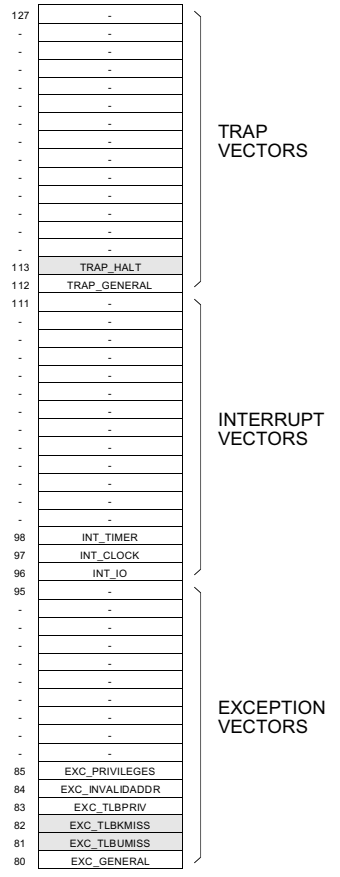
The *kernel save area* is used for saving state during handler execution, etc. The *Ve*, *Vi*, and *Vt* regions contain vector addresses for *exceptions*, *interrupts*, and *traps*, respectively. The *kernel page table (KPT)* maps the region of memory in which process structures and associated data are held. You will not use the KPT region in this project. The *root page table (RPT)* contains the mappings for the various user page tables that occupy the top quarter of the virtual address space. It is no accident that the RPT is placed in the top quarter of page frame 0—as mentioned earlier, the result is that the VPN of any kernel virtual address can be used directly as a physical address to obtain the appropriate root-level PTE.

For this project, you need to put data into page 0 (set up Ve and Vt regions, as well as one PTE in the root page table for ASID number 9). Note that the PSR should be initialized appropriately to contain ASID 9 as well, so that the hardware can create the correct address as part of responding to a TLBUMISS exception. You will also need to put handler code somewhere in physical memory, with the vector addresses initialized to point to the handlers using physical addresses. For example, you could very well put the handler code in page 1 and point the addresses in the Ve and Vt regions to these locations. Lastly, you must create a page table for an application. For example, you could put this page table into physical page 2 and point the root PTE corresponding to ASID 9 to page 2.

3.10 Interrupt Vector Table

The interrupt vector table has a simple format: for every exceptional condition that the hardware recognizes (including exceptions, interrupts, and/or traps), there must be an address in the table that points to a handler routine. The table is located at physical address 80 in memory and has 48 entries (16 exception types, 16 interrupt types, and 16 trap types). This is illustrated in the figure to the right.

Those vectors that must be implemented are shaded; vectors that are not shaded need not be implemented in this project.



4. Your Task

Your task in this project is to build the memory-management scheme described in this document. You have been given my solution for Project 2, extended with a larger register file; you can start there or start with your own project 2 code. You are to build an exception facility that handles interrupts precisely. You are to implement the TLB (2-entry and fully associative). You are to implement two exception types, one trap type, one TLB-management instruction, and one mode instruction:

```
EXC_TLBUMISS
EXC_TLBKMISS
TRAP_HALT
TLB_WRITE
MODE_HALT
```

You are to write handlers for each of the exceptions and trap types in RiSC-16 assembly code. You are to build a memory image containing your OS code (at this point, comprised of only handlers), kernel data, kernel save area, interrupt vector table, and an initialized kernel page table mapping all of the kernel's virtual code & data as well as the user page table for ASID 9. Follow the example given in the previous section:

1. Create your handlers and load them in physical memory: page frame 1 (starting at physical memory address 256).

2. Initialize your IVT so that the entries point to the appropriate handler locations.

3. Put the user page table into page frame 2 (starting at physical address 512). You do not need to initialize the user page table entries—my test code will choose a physical location for the application and initialize the page table for you (therefore, this step requires no work). However, you *will* need to initialize this page table when testing your own code.

4. Lastly, you need to put the user page table's physical location into the root page table (RPT): at the RPT entry corresponding to ASID 9, you must put a valid page table entry (see the format above) and set the page frame number to '2' (where the user page table has been placed).

Your processor should start running with user mode enabled (K bit is PSR set to '0'), nothing but zeroes in the kernel-mode history vector (i.e. the top eight bits of the PSR), the ASID set to the value '9', and the program counter set to 0.

What I want from you:

1. Your RiSC.v pipeline.

2. A file called "init.sys" that represents the contents of the first 3 pages of physical memory (the initial page frame, a page of handler code, and the user page table for ASID 9).

I will test your code by loading a random program (probably *laplace.s* because it is large) into the memory space at a randomly chosen physical location and initializing the user page table for ASID 9 to point to the appropriate physical locations. The grade will break down along the following lines:

- Correctness of hardware exception recognition and TRAP handler invocation, 5 points
- Correctness of memory-management/address-translation hardware, 5 points
- Correctness of TLB-miss handler implementations (*umiss* and *kmiss* handlers), 5 points