

## Project 3: Precise Interrupts (15%)

ENEE 646: Digital Computer Design, Fall 2002

Assigned: Thursday, October 24; Due: Tuesday, November 19

### 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.  $CTL_1$  and its associated control line,  $WE_{rf}$ , are eliminated. Because of the way the  $rT$  registers are used and the fact that  $r0$  is defined to be read-only,  $CTL_1$  and  $WE_{rf}$  were a bit superfluous.
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 raising 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. If a stage detects a non-zero *EXC* value, the associated instruction is disabled ( $op \leq ADD$ ,  $rT \leq 0$ ).
3.  $CTL_1$  now represents control logic for handling exceptions and interrupts in the writeback stage.
4.  $CTL_8$  has been added in the memory-access stage, and both  $CTL_4$  and  $CTL_5$  have been updated in the decode stage.  $CTL_8$  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.  $CTL_4$  and  $CTL_5$  handle TLB-modify instructions.

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 different 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, instruction fetch is disabled, and all instructions in the pipeline behind the exceptional instruction are turned into NOPs.

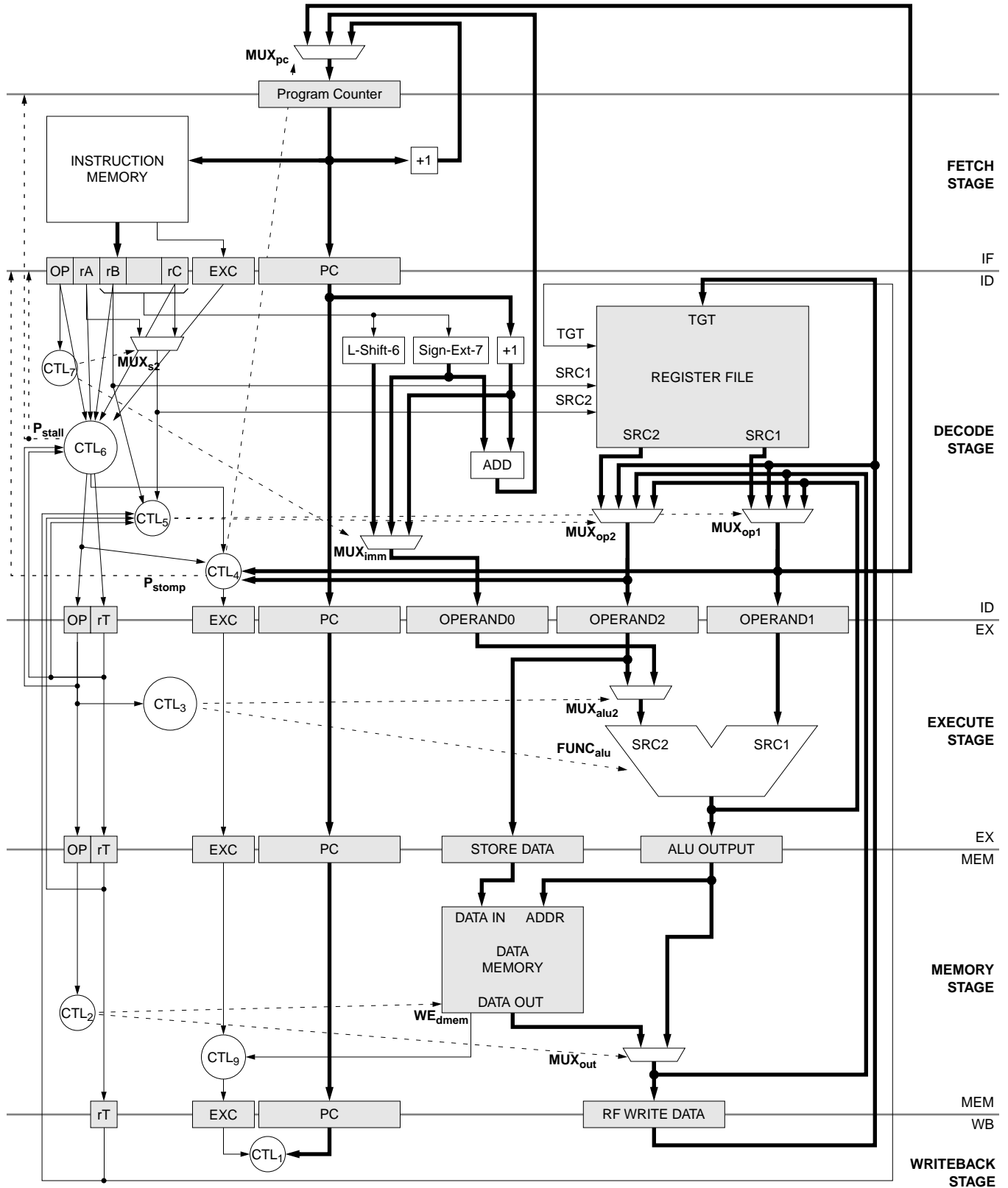


Fig. 1: RiSC-16 5-stage pipeline

- 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), and it 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:

- 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*).
- 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:

- Save the EPC in a safe location. This is done in case another exception or interrupt occurs before the handler has completed execution.
- Handle the exception/interrupt.
- Reload the EPC.
- Return the processor to user/unprivileged mode and jump to the (modified) EPC.

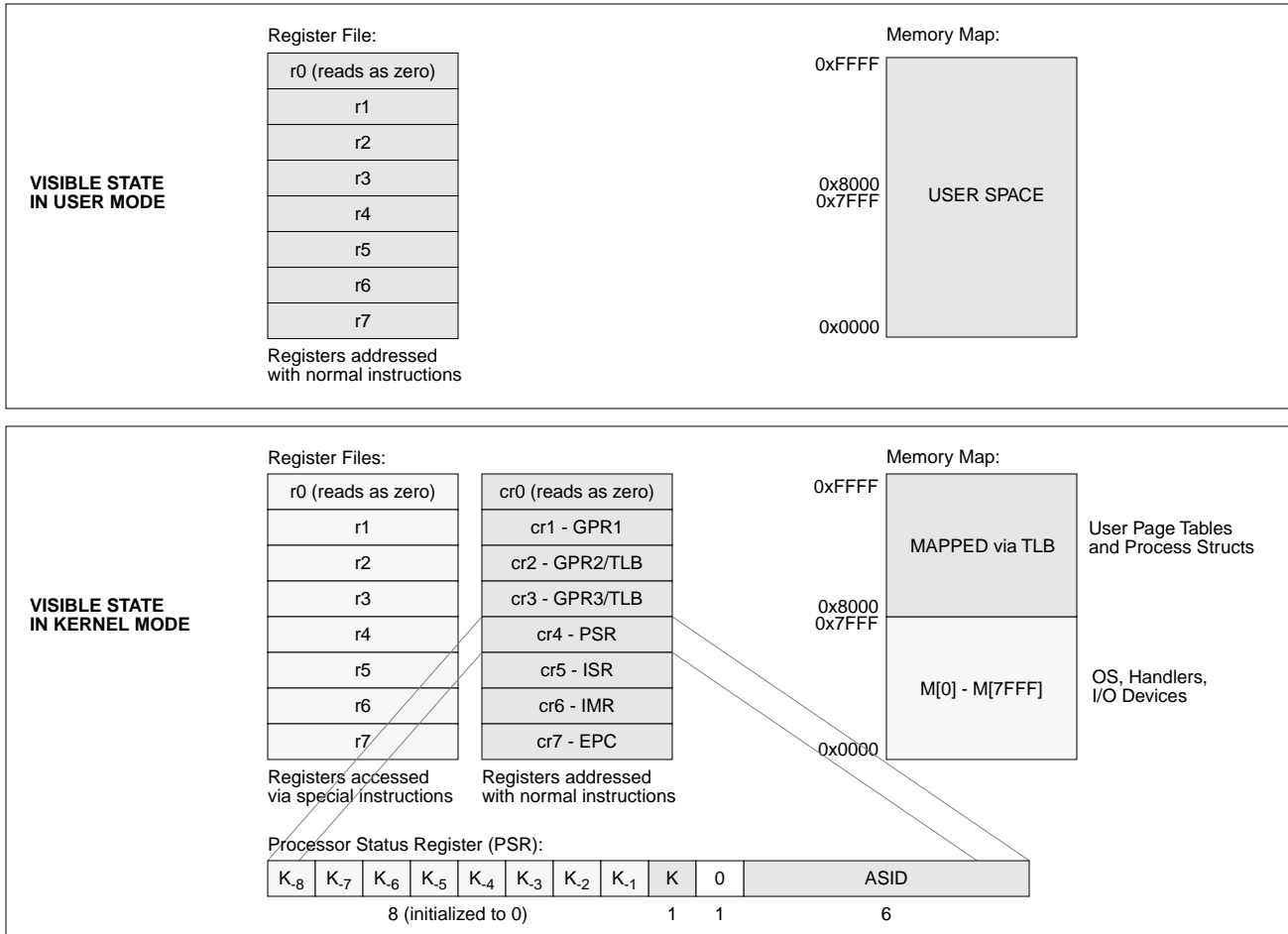
Most architectures have a facility (“return-from-exception”) that performs step 4 in an atomic manner.

### 3. Extended RiSC-16 ISA

To do all this, we need to 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:

- 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 the figure above. “GPR” refers to a general-purpose register. The PSR is the processor status register, which contains mode bits that directly influence processor operation. Access to the general-purpose register file is still possible while in kernel mode through special instructions (which are not needed 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 FIFO replacement policy. On every TLB write, flip a special bit—let that bit choose which TLB entry to replace. This will ensure that your code and my code behave the same way, even if our handlers are of different lengths.
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.
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.

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, in this case, software can always ignore this address (treat it as a “hint” that need not be followed) to implement whatever page table it wants. However, it would then have to generate its own PTE addresses.

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. The more complete instruction set is given below:

Assembly-Code Format		Meaning
add	regA, regB, regC	$R[\text{regA}] \leftarrow R[\text{regB}] + R[\text{regC}]$
addi	regA, regB, immed	$R[\text{regA}] \leftarrow R[\text{regB}] + \text{immed}$
nand	regA, regB, regC	$R[\text{regA}] \leftarrow \sim(R[\text{regB}] \& R[\text{regC}])$
lui	regA, immed	$R[\text{regA}] \leftarrow \text{immed} \& 0\text{xffc0}$
sw	regA, regB, immed	$R[\text{regA}] \rightarrow \text{Mem}[R[\text{regB}] + \text{immed}]$
lw	regA, regB, immed	$R[\text{regA}] \leftarrow \text{Mem}[R[\text{regB}] + \text{immed}]$
bne	regA, regB, immed	<pre> if ( R[regA] != R[regB] ) {     PC &lt;- PC + 1 + immed     (if label, PC &lt;- label) }                     </pre>
jalr	regA, regB	$\text{PC} \leftarrow R[\text{regB}], R[\text{regA}] \leftarrow \text{PC} + 1$
<b>PSEUDO-INSTRUCTIONS:</b>		
nop		do nothing
trap	type	trap to the operating system with vector <i>type</i>
halt	or trap TRAP_HALT	ask operating system to stop machine & print state
lli	regA, immed	$R[\text{regA}] \leftarrow R[\text{regA}] + (\text{immed} \& 0\text{x3f})$
movi	regA, immed	$R[\text{regA}] \leftarrow \text{immed}$
.fill	immed	initialized data with value <i>immed</i>
.space	immed	zero-filled data array of size <i>immed</i>
<b>PRIVILEGED INSTRUCTIONS:</b>		
tlbw	regB	write TLB entry (held in regB) to the TLB
sys	class	cause exceptional condition of specified class
sys	MODE_HALT	stop machine & print state
rfe	regB	return from exception: waits until writeback to jump through regB and return processor to user mode (does not save the link pointer)

This is not the full instruction set, but it is enough to do this project.

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

### 3.1 Exceptions and Interrupts

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.

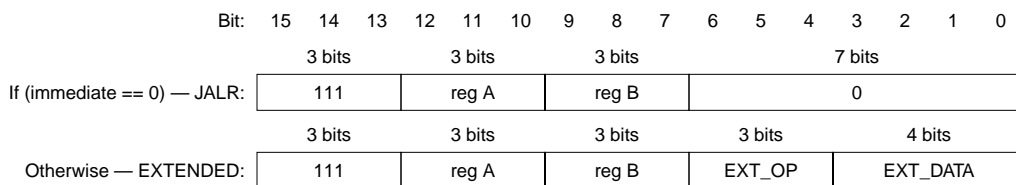
In addition, we have a number of exception types defined, including the following:

```
EXC_TLBUMISS
EXC_TLBKMISS
EXC_INVALIDOPCODE
EXC_PRIVILEGES
SYS_TRAP
```

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) except for SYS\_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. 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. This implementation is different not for any specific reason, but rather to explore the design space.

### 3.2 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 an 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), each of which has the following pipeline timing:

1. **Exceptions, interrupts, and traps** as well as all MODE changes (for example, MODE\_HALT) are shaded in the following table and handled in the **writeback stage**.
2. **Privileged operations** except for MODE changes are left unshaded in the following table and are handled in the **decode stage** (similar to BRANCHES in that regard).

These extended opcodes are used to directly effect 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 shut-down 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.

The following table 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.

Opcode Extension (EXT_OP)	Data Extension (EXT_DATA)	Semantics
<b>SYS_MODE (000)</b>	MODE_RUN (0) MODE_SLEEP (1) <b>MODE_HALT (2)</b> 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 <b>Halt machine</b> No definition yet  Halt and output panic value (8..15) (meaning is software-defined)
<b>SYS_TLB (001)</b>	TLB_READ (0) <b>TLB_WRITE (1)</b> TLB_CLEAR (2)	Probe TLB for PTE matching VPN in rB <b>Write contents of rB to TLB (random)</b> 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
<b>SYS_RFE (011)</b>	<b>Data value ignored</b>	<b>Return From Exception: JUMP (without link) to address held in rB (a control register), and right-shift (zero-fill) the kmode history vector</b>
SYS_RESERVED (100)		Has no definition yet
<b>SYS_EXCEPTION (101)</b>	EXC_GENERAL (0) <b>EXC_TLBUMISS (1)</b> <b>EXC_TLBKMISS (2)</b> EXC_INVALIDOPCODE (3) EXC_INVALIDADDR (4) EXC_PRIVILEGES (5)	General exception vector <b>User address caused TLB miss</b> <b>Kernel address caused TLB miss</b> 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
<b>SYS_TRAP (111)</b>	TRAP_GENERAL (0) <b>TRAP_HALT (1)</b>	General operating system TRAP vector <b>Ask operating system to perform HALT</b>

Those mechanisms that your Project 3 implementation must support are in **bold**. Note that in all cases but TLB\_READ, TLB\_WRITE, and SYS\_RFE, the rA and rB fields of the JALR instruction are ignored.

These last three instructions, however, do use the rB and/or rA fields of the instruction, specifying a register to read and/or write.

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

### 3.3 Control Registers

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 and TLB interface
cr3 - For general-purpose use and TLB interface
cr4 - Processor Status Register
cr5 - Interrupt Status Register
cr6 - Interrupt Mask Register
cr7 - EPC Register
```

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.

The control registers behave as follows:

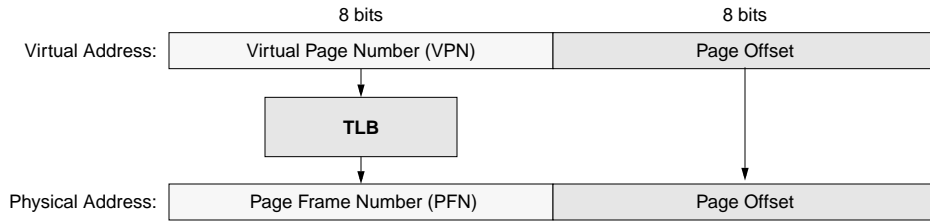
- cr0** Like rf[0], this is always zero.
- cr1 (gpr1)** This register is for general-purpose use. However, if an interrupt handler is going to use the register, it should save the register's contents before writing to it and restore the contents prior to exiting, just in case the handler happened to preempt another handler using the register.

- cr2 (gpr2)** This register is for general-purpose use, just like cr1, with the addition that on TLBMISS exceptions the hardware places into this register the address of the user page-table entry (see the next section). Though the contents of the register may be overwritten at any moment, it only happens as a result of causing a TLB miss while executing in user mode, so the kernel should not have to worry about losing register contents.
- cr3 (gpr3)** Another general-purpose register, with the addition that on TLBMISS exceptions the hardware places into this register the VPN of the faulting address. For a UMIS, the VPN is placed in the top 8 bits of the register, with the bottom 8 bits set to zero. Thus, once the matching PFN is loaded (obtained from the PTE), it can simply be added to the VPN and then written to the TLB. For KMISS, the VPN is placed in the bottom 8 bits. 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 UMIS handler should first move the VPN into a different register before attempting to load the user PTE.
- cr4 (psr)** The *process-status register* is often considered the heart of the machine, as it contains some of the most important state information in a processor. In the RISC-16, the status 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. Every time an exception or interrupt is handled, the kernel-mode bit should be left-shifted into this array (which is itself shifted to the left to accommodate the incoming bit), and the kernel most should be set appropriately (usually turned on). Every invocation of the return-from-exception instruction should right-shift this bit-array (while zero-filling from the left) and place the rightmost bit of the array into the kernel-mode bit. This implements a hardware stack of previous mode bits, which 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.
- 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.

To implement the control registers, it might be simplest to declare **rf** as 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 comes 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.

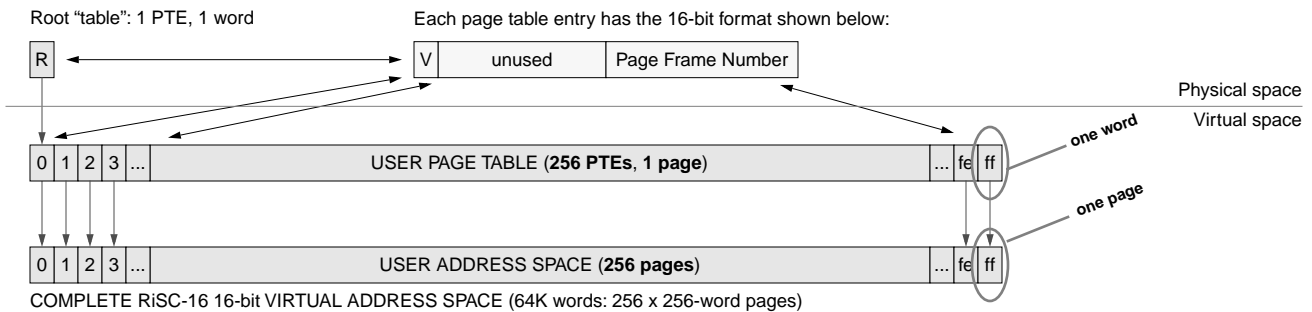
### 3.4 Memory Management Architecture and Virtual Memory Organization

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 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):

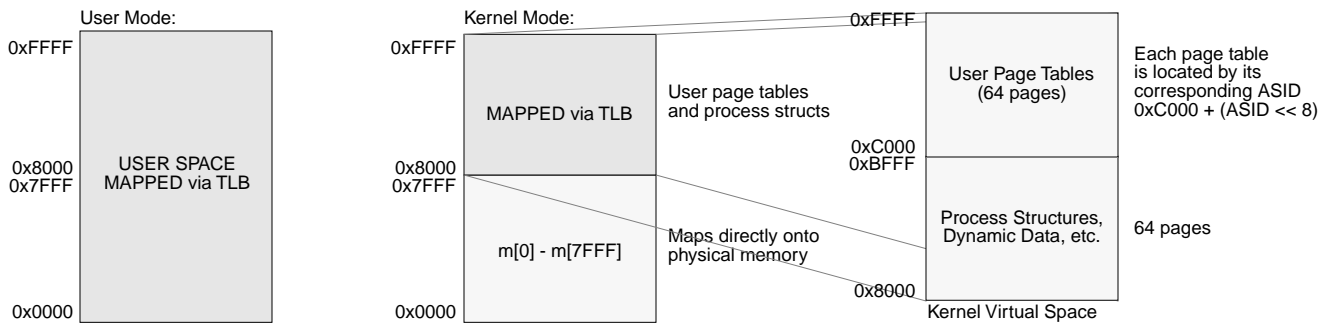


The full address space contains 256 pages, which requires 256 PTEs to map it. Each PTE is a single word, and 256 PTEs can thus fit in a single page. Therefore, a single page of PTEs can map the entire user space. The kernel keeps a set of pages in its virtual space, each of which holds one user page 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 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.

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 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:

$$ADDR_{pte} = 0xC000 + (ASID \ll 8) + VPN$$

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. The address is placed in **cr2**, control register 2. In addition, the hardware places the faulting address into **cr3** and zeroes out the bottom eight bits (the page offset). After performing these steps, the hardware vectors to the UMIS handler. When the handler runs, it will use the virtual address in cr2 to reference the PTE. When the PTE is loaded, the handler obtains the PFN (see figures above & below for specifics on the PTE format). The handler then combines the VPN and PTE into the TLB’s format—this is accomplished by simply adding the contents of the two registers—and performs a **tlbw** (TLB write) instruction.

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.

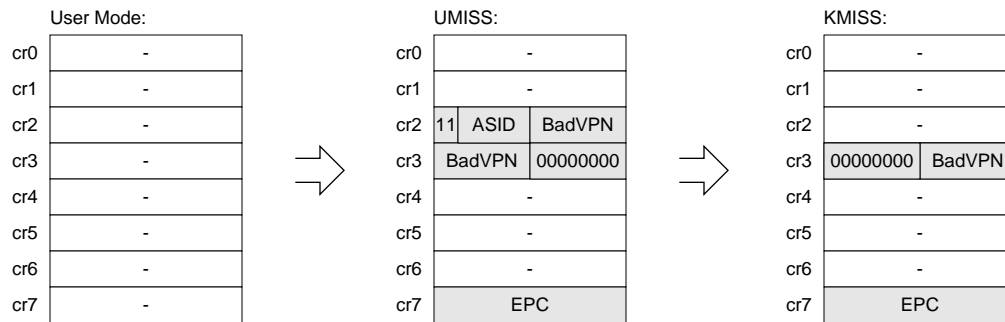
When handling a kernel TLB miss (EXC\_TLBKMISS), the page table needed is the kernel’s own page table that maps the top half of the address space (the kernel’s virtual space). This page table is illustrated in the next section; it is located at address 128 in physical memory and extends to address 255. 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. Unlike the UMIS handler, the hardware places the VPN in the LOW EIGHT BITS of the register. This is so that the handler can

use the value first as an address and then as the VPN later. This avoids the need for the EXC\_TLBKMISS hardware to write more than one value to the register file.

If an address in user mode causes a TLB miss, the hardware places the PTE address in cr2 and the VPN of the faulting instruction (labeled *BadVPN*) in cr3. If this handler (or any kernel reference) causes a TLB miss using a virtual address with the top bit set, the hardware places the VPN of the faulting instruction into cr3. The format of the VPN in UMISSE is chosen to facilitate quick generation of the TLB entry. The format of the VPN in KMISS is chosen to avoid having to place both the address and the VPN into the control registers. The handler can use the VPN first as an address, and then left-shift it eight places by successive **add** instructions to get the VPN into a TLB-entry format.

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). The locations in the register file and formats of the various data that are placed into the register file *by hardware* are illustrated in the following figure:

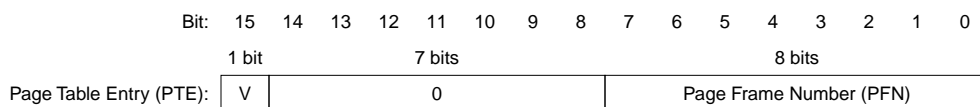


If, during user mode, an access causes a TLB miss, the *umiss* handler is invoked. The hardware places the PTE’s virtual address into cr2. The hardware also places the faulting VPN, labeled *BadVPN*, into cr3 (using the correct format for a TLB entry so that later the PFN can simply be added to the value in cr3 to produce a TLB entry used by TLB\_WRITE). This is done in addition to saving the return PC in cr7, and it is all done before vectoring to the *umiss* handler.

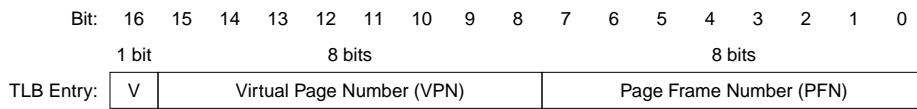
If, during the execution of the *umiss* handler (or, in fact, during any kernel code at all), a memory access causes a TLB miss, the *kmiss* handler is invoked. The hardware places the faulting 8-bit VPN into cr3, in the lower 8 bits of the register. The hardware places the return PC (which, in most cases will be the PC of the LW instruction in the *umiss* handler) into cr7. This is done before vectoring to the *kmiss* handler.

Because the *umiss* handler uses a virtual address for its PTE load, it can invoke the *kmiss* handler if it misses the TLB and must therefore worry about the contents of cr3 (which will be overwritten if the load causes a TLB miss). It executes using physical addresses for instructions, so i-fetch will not cause a TLB miss. When the *kmiss* handler runs, it does not need to worry about either cr2 or cr3, because the only way they can be overwritten is if a user address causes a TLB miss, and because the *kmiss* handler uses only physical addresses, this will not happen.

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 relatively straightforward. First, the handler must verify the validity of the PTE, then extract the page frame number, concatenate it with the BadVPN, and write the result to the TLB. This can be accomplished with the following instructions. Assume that BadVPN is in r1 and the PTE is in r2.

```

lui    r3, 0x8000          # will be used to test top bit
nand   r3, r3, r2          # r3=0111111111111111 => val; r3=1111111111111111 => inv
nand   r3, r3, r3          # r3=0x8000 => val; r3=0x0000=>inv
beq    r3, r0, invalid
add    r2, r2, r3          # clears top bit of PTE, PFN is left
add    r1, r1, r2          # concatenates BadVPN with PFN
tlbw  r1                    # writes contents of r1 to TLB, sets 'v' bit in TLB entry
    
```

The steps that the KMISS handler goes through are very similar, except that the BadVPN as given to the handler is not in the correct format for the TLB entry. Thus, it must be left-shifted eight places before adding it to the page frame number (assume BadVPN is in r1 and the PTE is in r2):

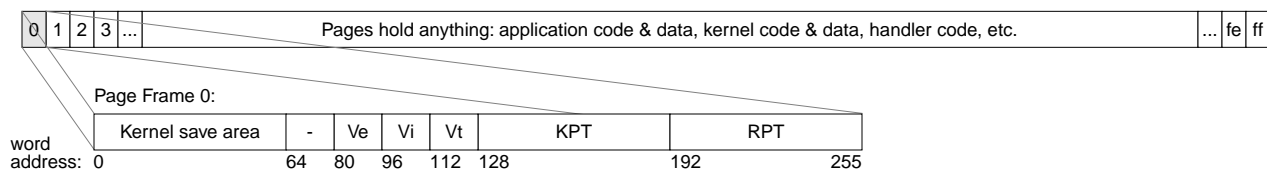
```

lui    r3, 0x8000          # will be used to test top bit
nand   r3, r3, r2          # r3=0111111111111111 => val; r3=1111111111111111 => inv
nand   r3, r3, r3          # r3=0x8000 => val; r3=0x0000=>inv
beq    r3, r0, invalid
add    r2, r2, r3          # clears top bit of PTE, PFN is left
add    r1, r1, r1          # left-shift VPN by 1
add    r1, r1, r1          # left-shift VPN by 1
add    r1, r1, r1          # left-shift VPN by 1
add    r1, r1, r1          # left-shift VPN by 1
add    r1, r1, r1          # left-shift VPN by 1
add    r1, r1, r1          # left-shift VPN by 1
add    r1, r1, r1          # left-shift VPN by 1
add    r1, r1, r1          # left-shift VPN by 1
add    r1, r1, r1          # left-shift VPN by 1
add    r1, r1, r1          # left-shift VPN by 1
add    r1, r1, r2          # concatenates VPN with PFN
tlbw  r1                    # writes contents of r1 to TLB, sets 'v' bit in TLB entry
    
```

Note that these do not represent complete handlers: for example, error-checking 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, because the PTEs you reference in your page tables should never be invalid.

### 3.5 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:



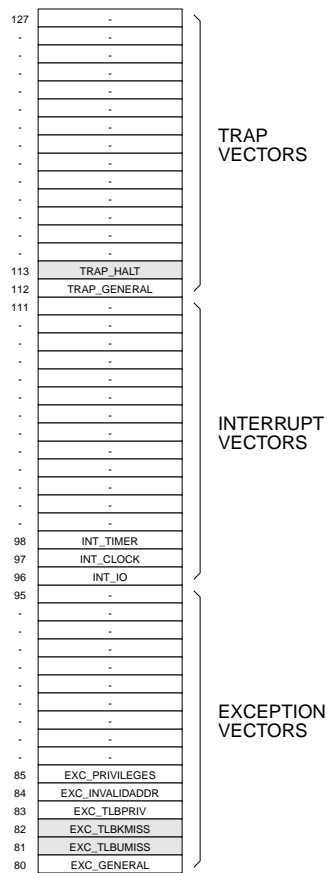
The *kernel save area* is used for saving state during handler execution, etc. The *Ve*, *Vi*, and *Vt* regions contain vector addresses for *exceptions*, *interrupts*, and *traps*, respectively. The *kernel page table (KPT)*

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 an ASID of your choosing). Note that the PSR should be initialized appropriately to contain whatever ASID you have chosen, 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 the ASID chosen to page 2.

### 3.6 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 following figure:



Those vectors that must be implemented are shaded; vectors that are not shaded do not need to 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; you can start there or start with your own project 2 code. You need to build the control registers, and give each the described characteristics; note that neither the ISR nor the IMR need to be implemented (cr5 and cr6). 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 some ASID. 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 the ASID you have chosen (and written to the bottom bits of the PSR), 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 in 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 '8', 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 8).

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 8 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