1 Interrupts!

Interrupts provide a mechanism for devices external to the processor to steal processor cycles without the normal software knowing about it.

For this checkpoint you will be implementing interrupts in your MIPS processor. There are many different versions of the MIPS processor, and we will try to stay consistent with what’s in your book, and what’s in the simplest of the commercially available MIPS processors.

When an interrupt occurs the current instruction is completed and then the PC is loaded with the interrupt vector, which is the address of the interrupt service routine (ISR). When the ISR is done, execution resumes at the next instruction as if there had never been an interruption.

In order to take care of storing some of the extra state information needed to handle interrupts, a set of registers known as CP0 (co-processor 0) will be added to your Checkpoint 2 datapath. A register in CP0, ‘Cause’, will have bits turned on when peripherals signal an interrupt request or when certain clock-related interrupts occur. An output of CP0, InterruptRequest, will go high when one of these interrupt pending bits go on for an enabled interrupt. It is then the responsibility of the PC multiplexing logic to handle loading the ISR PC.

In a pipelined processor, the definition of current instruction can be a little fuzzy, so you get to decide what the simplest implementation is. The interrupt must save the address of the next instruction. On the MIPS, this is done in a register called EPC, or Exception Program Counter. For our implementation there will be only one ISR, at address 0xc000.0180. The type of interrupt (timer, UART, etc.) is specified in the Cause register with a set of bits that indicate an Interrupt Pending (IP).

2 CPU Modification

The major modifications to your datapath are the addition of a new memory (isr_mem) to fetch from and store to and the integration of CP0 (interchangably called COP0150 after the Verilog module name) to your datapath. The selection of the PC in the pre-instruction fetch stage (the next PC) must be adapted to move to the ISR when an interrupt is being handled; this will require the addition of more PC select logic.
CoProcessor 0

CP0 is really just a collection of 32 registers that are not a part of the normal register set for the MIPS. In some MIPS implementations there are actually completely separate register files called shadow registers which contain copies of all of the usual MIPS registers (like $s0, $t0, etc.), but that’s a separate capability from what CP0 provides. We won’t be using shadow registers, but we will use CP0. We won’t need most of the registers in CP0 in fact we have given you the coprocessor in the skeleton files. The ones we will need are listed below:

Count register (CP0 reg 9)
The Count register increments at the CPU clock rate. This is useful for tracking the number of cycles elapsed in a certain sequence of instructions – subtracting the value of ‘Count’ at the beginning from the value at the end. You already have this functionality memory-mapped from Checkpoint 2. You can just use that mechanism to get count if you want to, or use the same register and access it through CP0 – your choice.

Compare register (CP0 reg 11)
Used to set a timer. By setting the ‘Compare’ register to the sum of the ‘Count’ register and an offset, an interrupt can be raised the desired number of clock cycles later. The ISR will then do whatever is desired, since programming the ISR is part of this checkpoint.

Status register (CP0 reg 12)
This register is used to control the behavior of interrupts.

- Status[15:10] – mask to control individual interrupts (1 is enabled, 0 is disabled)
  - Status[15] – enable/disable timer interrupts (these occur when Count == Compare)
  - Status[14] – enable/disable RTC (real-time clock) interrupts (occur when Count overflows)
  - Status[10] – enable/disable UART receive interrupts (covered in detail in part 5)
- Status[0] – global interrupt enable – 1 to allow any interrupts, 0 to disable all interrupts

Cause register (CP0 register 13)
This register is used to indicate when interrupts have been requested by various devices. If the bit in ‘Cause’ corresponding with a particular device is high when the ISR starts, that device produced an interrupt if and only if the corresponding status bit is also high.

- Cause[15:10] – interrupt pending bits for devices
  - Cause[14] – RTC – goes high when Count overflows
  - Cause[11] – UART transmit interrupt pending – this gets turned on after the first cycle where DataInReady is high after having been low, meaning the UART is ready to send more data (covered in more detail in part 5)
  - Cause[10] – UART receive interrupt pending – this gets turned on after the first cycle where DataOutValid is high after having been low, meaning the UART has new valid data (covered in more detail in part 5)
- Cause[9:0] – not covered in CP3 – includes flags for software interrupts

**Exception Program Counter, EPC (CP0 reg 14)**

This register contains the PC that should be returned to after the ISR has completed its execution. As the interrupt can be handled a few clock cycles after InterruptRequest first goes high, this should hold the PC of the first instruction that did not complete due to branching to the ISR. This is much like $ra for normal procedure calls.

**Timer – built into COP0150**

The timer functionality of CP0 should generate a timer interrupt when Counter==Compare and a Real Time Clock (RTC) interrupt when Count rolls over from 0xFFFF.FFFF to 0x0.

When the Compare register is equal to the Count register, Cause[15] (interrupt pending 7, or IP7 – indicates timer interrupt is pending) is set. If both Status[15] (interrupt mask 7, or IM7 – enables timer interrupts) and Status[0] (global interrupt enable) are set, then an interrupt is generated immediately. Writing to the Compare (using the mtc0 instruction discussed below) register clears Cause[15].

When the Count register rolls over (increments from 0xFFFF.FFFF), Cause[14] (interrupt pending 6, or IP6 – indicates RTC interrupt is pending) is set. If both Status[14] (interrupt mask 6, or IM6 – enables RTC interrupts) and Status[0] (global interrupt enable) are set, the InterruptRequest output of COP0150 goes high, and the ISR starts.

Based on the Cause and Status contents, the beginning of the ISR will see that an RTC ISR was responsible, and the portion of the ISR responsible for this type of interrupt (usually a label such as ‘RTC’ is jumped to) will increment a value at a fixed memory location, known as the software Real Time Clock.

CP0 will need to be integrated into your CPU in several ways. The module COP0150 has several ports that will connect with elements of your datapath and control logic.

- **Clock, Reset**: these connect to the top-level clock and reset
- **Enable**: this input must be high for any change of state to occur in CP0, including updates to Cause based on the interface with peripherals
- **DataInEnable**: write enable for CP0 register writing
- **DataAddress[4:0]**: this input takes the address of the CP0 register that will be read asynchronously. This register will also be written on the positive edge of the clock if DataInEnable is high. For example, if DataInAddress is set to 0xb (11 or Compare), the current value of Compare will emerge asynchronously on DataOut. If the DataInEnable is high, the value on DataIn will be copied into Compare on the next positive edge of the clock.
- **DataIn[31:0]**: write data input
- **DataOut[31:0]**: read data output
- **InterruptedPC[31:0]**: this input is fed the value of the PC that should be stored in the EPC register. This should be related to the PC that would have been fetched when the ISR PC was muxed in. This will only be written into EPC when InterruptHandled is asserted; therefore, it is okay to always be driving it with the same function of the PC from some stage.
- **InterruptRequest**: this output is high when an interrupt is requested (determined by bitwise AND of IP and IM followed by a reduction to see if any bit of the result is on, along with the global interrupt enable). This will stay high until InterruptHandled is asserted.
• **InterruptHandled**: assert this input when the ISR is about to be fetched to let CP0 know that the interrupt is being handled.

• **UART0Request**: this should be asserted only on the first clock cycle where the DataOutValid output of the UART is high after having been low. If the ISR handles UART Rx interrupts by simply doing nothing, the fact that the DataOutValid signal stays high should not cause infinite interrupts – therefore, this cannot simply be fed with DataOutValid. A transition must be sensed, and a simple FSM to implement this behavior is described in the appendix.

• **UART1Request**: this should be asserted only on the first clock cycle where the DataInReady output of the UART is high after having been low. If the ISR doesn’t find any data to send out the UART (how it finds data is covered in section 5), the fact that DataInReady stays high should not cause infinite interrupts either.

As suggested by the read and write ports on CP0, the capability to read from CP0 (copying the contents of a CP0 register to a normal, architectural register) and to write to CP0 (copying the contents of a normal, architectural register to a CP0 register). This will be implemented through the addition of two new instructions, mfc0 and mtc0.

**mfc0**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
<th>mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>mfc0 rt, rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010000</td>
<td>00000</td>
<td>dest</td>
<td>src</td>
<td>–</td>
<td>–</td>
<td>mfc0 k0, Cause # opcode=16, rs=0, rd=13, rt=26; get the Cause register</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Copy contents of coprocessor register $rd into architectural register $rt. The data should be read from CP0 in the execute stage, and the write to the architectural register should be set up in the writeback stage just like any other instruction that writes into the architectural register file.

Consider muxing the data coming out of CP0 into the input of the ALUOut pipeining register – this will allow you to leverage your existing forwarding logic, as the writeback data from an mfc0 must be forwarded, whereas before only results of R-type instructions (not loads, etc.) were forwarded.

**mtc0**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
<th>mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>mtc0 rt, rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010000</td>
<td>00100</td>
<td>src</td>
<td>dest</td>
<td>–</td>
<td>–</td>
<td>mtc0 k1, Status # opcode=16, rs=4, rd=12, rt=27; put k1 in Status register</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Copy contents of architectural register $rt into coprocessor register $rd (notice that the coprocessor register address is always $rd for mfc0 and mtc0, while the architectural register address is always $rt).

This can be set up in the execute stage – in other words, the DataIn input of your COP0150 instance can get the contents of $rt in the execute stage and DataInEnable can be set by control logic in the execute stage that detects whether the instruction in that stage is an mtc0 (the only time DataInEnable is high).

This allows elimination of any possibility of needing to forward data from an incomplete mtc0 to an immediately subsequent mfc0 (nice!)
3 Placement of COP0150 in datapath

The functionality outlined above will require hooking up an instance of COP0150 instantiated inside MIPS150. The diagram below shows a rough outline of one possible placement in the datapath, but omits several details covered in the text of this specification. Feel free to use this as a guide.

4 Memory map modifications

The memory map of your CPU will necessarily be updated to accomodate the a memory to hold the interrupt service routine (ISR). This program will live it its own memory, isr_mem. This is a 4096-line SRAM with one read port and one write port. The procedure for adding a new memory with both fetch and data accesses should be familiar from the previous checkpoint.

When considering the new memory map required to utilize this and other devices, it is easier to segment the accesses by type: fetch, read, or write. The three tables below specify the devices usable for each type of access and the associated top nibbles. As always, undefined behavior is undefined.

<table>
<thead>
<tr>
<th>Fetch Access Map</th>
<th>Read Access Map</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PC[31:28]</strong></td>
<td><strong>Addr[31:28]</strong></td>
</tr>
<tr>
<td>4'b0001</td>
<td>4'b00x1</td>
</tr>
<tr>
<td>4'b0100</td>
<td>4'b0100</td>
</tr>
<tr>
<td>4'b1100</td>
<td>4'b1000</td>
</tr>
<tr>
<td>I-Cache</td>
<td>D-Cache</td>
</tr>
<tr>
<td>BIOS</td>
<td>BIOS</td>
</tr>
<tr>
<td>isr_mem</td>
<td>MMIO</td>
</tr>
<tr>
<td>Addr[31:28]</td>
<td>Device</td>
</tr>
<tr>
<td>-------------</td>
<td>--------</td>
</tr>
<tr>
<td>4'b00x1</td>
<td>D-Cache</td>
</tr>
<tr>
<td>4'b001x</td>
<td>I-Cache</td>
</tr>
<tr>
<td>4'b1000</td>
<td>MMIO</td>
</tr>
<tr>
<td>4'b1100</td>
<td>isr_mem</td>
</tr>
</tbody>
</table>

### 5 Serial Output Software FIFO

Currently, any program wanting to send data out the UART must constantly check `DataInReady` to see if another transmission is possible. This is quite inefficient, as a program waiting to complete its output is wasting time that could be spent on computational tasks. Checkpoint 3 will address this issue by using a software FIFO (or queue) to hold data that the ISR will send out through the UART whenever the UART transmit module becomes ready.

In the user application running from the instruction cache, a string summarizing the running time of a particular series of 100 million increments performed on a variable is transmitted over the UART at the end of the benchmarking sequence. This is done by writing the data from the string into the software FIFO.

After the write into the software FIFO (explained in greater detail below), the user application will check the status of `DataInReady` on the UART once. If it is high, the first character of the string will be stored to the UATransmit data register, sending it over the serial line. Otherwise, the user application will not send any data for that string.

After this first write attempt, the `DataInReady` signal from UATransmit must be low. Then, the utility of the software FIFO will come into play. The user application can immediately begin new computation. However, as soon as `DataInReady` goes high again, the UART1Request input to COP0150 will go high, causing a UART Tx interrupt.

The CPU will then branch to the start of the ISR. When the starting portion of the ISR recognizes a UART Tx interrupt, it will branch to a segment of the ISR designed to handle such an interrupt. This segment will check if there is data waiting in the software FIFO (discussed below); if there is, it will dequeue it and store it to the UATransmit data register, sending out that character. This will happen again every time the DataInReady signal goes high, until the whole string has been sent out.

The fact that UART Tx interrupts only occur once after the transition of DataInReady to a high value means that the ISR will sometimes execute and find no data in the FIFO to send to the UART. No UART Tx interrupts will then be requested again until the DataInReady signal goes low and then high again. This is why the user application must attempt to send out one character itself: it must drive `DataInReady` low if it isn’t already, so that some later interrupt will start the process of draining the FIFO.

The specific implementation details of the software FIFO are entirely free. There are certainly many other implementations, many of which are better. However, this section attempts to provide a general outline for a intuitively simple FIFO that is usable by both the ISR and the user application.

A software FIFO can be realized using three variables: an array `buffer` of the desired data type (in this case, single bytes or characters), an integer `inIdx` which indicates what position in the queue will be written to on an ‘enqueue’ operation, and an integer `outIdx` which indicates which position in the queue will be read from on a ‘dequeue’ operation.
When the FIFO is written to by the program producing the output, data is copied to the element specified by \texttt{inIdx} and \texttt{inIdx} is incremented. If this pushes \texttt{inIdx} past the end of the buffer, it is wrapped around using the modulo operator. When the FIFO is read by the ISR, a load is performed from the element specified by \texttt{outIdx} and \texttt{outIdx} is incremented (also modulo the size of the FIFO). If \texttt{inIdx} and \texttt{outIdx} are equal, the FIFO must be empty.

The following is a code sample using this concept to implement the basic data structure. However, several important implementation details are omitted, which will be discussed below.

```c
#define BUFFER_SIZE 2048

// software FIFO in a basic C program
char buffer[BUFFER_SIZE];
int inIdx, outIdx;
inIdx = 0;
outIdx = 0;

// enqueue first character from char *string
buffer[inIdx] = *string; // copy character
inIdx++; // increment index for next write
inIdx %= BUFFER_SIZE; // wrap this index around if it goes past the end
string++; // increment the string pointer to get the next character
```

Although the code above has the right pieces, it neglects a very important detail: the necessity of two separate programs interacting with the FIFO. Writes into the FIFO will be done by the application, while reads from the FIFO will be done by the ISR.

This means that they both must be aware of the address of the start of the FIFO buffer and the addresses at which the two index variables are stored (along with the constant buffer size). Furthermore, the programs must be written in such a way as to be aware of the fact that the contents of memory might be modified outside the program's own execution. These issues may be solved with some embedded C programming techniques.

The ‘volatile’ keyword indicates that a particular variable may have its value changed outside the current program. Declaring \texttt{volatile char *buffer} creates a pointer to a volatile character, while \texttt{volatile char[BUFFER\_SIZE]} creates an array of volatile characters.

Pointers in C may also be set to point at specific addresses. The following excerpt from \texttt{echo.c} uses pointers set to literal addresses to point to volatile variables. This is used to read data from the MMIO UART DataOutValid register, which is at address 0x80000004.

```c
#define RECV_CTRL (*((volatile unsigned int*)0x80000004) & 0x01)
...
while (!RECV_CTRL) ;
```

Similar steps may be taken to deal with making \texttt{buffer}, \texttt{inIdx}, and \texttt{outIdx} visible to the ISR and the application. Some questions to think about before attempting an implementation include:

- Which memory should these variables live in?
- What address should the buffer begin at? How will this address appear in both programs?
- What addresses should the index integers reside at? How will the integers be retrieved from these addresses?
A buffer size of 20 will allow you to ensure that the buffer never fills up if the correct steps are taken at the software level (such as comparing the in and out indices to see if \texttt{inIdx} is one element behind \texttt{outIdx}).

6 Interrupt Service Routine

The ISR will need to examine the Cause register to figure out what caused the interrupt. When a device requests an interrupt, the corresponding bit is set in the Cause register. This does not necessarily generate an interrupt though. If the interrupt mask for that interrupt is zero, or if interrupts are disabled, then no interrupt will be generated.

Interrupts are disabled while executing the ISR, as they are disabled when \texttt{InterruptHandled} is asserted. If a device requests an interrupt while the ISR is executing (servicing another interrupt), then the IE bit will be zero, and a new interrupt will not be generated. As soon as the ISR is finished and interrupts are re-enabled, then the pending interrupt will cause a new interrupt to be generated (assuming the interrupt mask enables interrupts from that particular source).

The ISR is responsible for clearing the appropriate IP bit in the Cause register. Writing the ISR will be a major portion of this checkpoint; however, the example code and this section of the specification will provide some implementation details. It is quite possibly best to write the ISR in assembly; to get familiarized, examine the CP3 example code. The example code provides the beginning of a skeleton for an ISR. Since the opening of the ISR is responsible for figuring out which type of interrupt was pending, the first segment of the example code examines the Cause and Status vectors.

Depending on the source of the interrupt, a particular label will be jumped to, and a specific part of the ISR will perform the desired task associated with that interrupt. The description of the software FIFO above details the functionality of UART Tx interrupts that occur when \texttt{DataInReady} goes high; however, the ISR also handles UART Rx interrupts when \texttt{DataOutValid} goes high.

The UART Rx interrupts are handled by a UART Rx section of the ISR. This section loads the new, valid data. Then, depending on the contents of the received byte, it either sets the timer interrupt enable bit (for ‘e’), clears that bit (for ‘d’), or stores the byte into a variable called ‘STATE’ – another volatile variable (in the C program running from the I-Cache) mapped to a fixed memory location (only ‘r’, ‘R’, ‘v’, or ‘V’ should be stored in ‘STATE’).

The ‘STATE’ variable, like the buffer and indices of the software FIFO, may be accessed by the application through a pointer to a volatile or by the ISR using loads and stores to specific addresses. The value of ‘STATE’ controls the execution of the main program.

Although there are other interrupt types to handle, the diagram on the next page shows the behavior of the ISR for UART-based interrupts. At the end of the ISR execution, the interrupted PC is loaded from the EPC register in CP0 to $\text{k0}$ or $\text{k1}$, reserved registers that the ISR may overwrite. This is handy, as the ISR must restore the contents of any saved registers (like a subroutine) or temporary or other registers (unlike a subroutine) before it jumps to the EPC.

This is followed by a \texttt{jr} to the register EPC was loaded to and then by a write to the Status register in CP0 to turn the mask back on (using a \texttt{mtc0}).
7 Application code

The application code is roughly described by a portion of the example code. It loops forever, constantly checking the value of the ‘STATE’ variable (which is a volatile variable pointed to by a pointer to a specific address agreed upon between the ISR and application). This relies on the ISR to handle UART Rx interrupts to update ‘STATE’ with keystrokes by updating ‘STATE’ on a receive interrupt if it receives ‘r’, ‘R’, ‘v’, or ‘V’.

There is a switch statement on the value of ‘STATE’; this causes the application code to start one of four possible types of benchmarks if the right key is pressed.

- Pressing ‘r’ should run a tight loop of 100 million addi instructions (plus the associated branches) to increment a register

```assembly
# kl = Status + 1 - global en on
ori $k1,$k1,1
mfcu $k0,SFC
jr $k0
mtc0 $k1,Status
```
• Pressing ‘v’ should enable the mode where the count uses the memory location

• Pressing ‘R’ should enable the mode where the count uses the register and a function to call the increment – one line in the loop is a jal to a function with an addi instruction

• Pressing ‘V’ should enable the mode where the count uses the memory location and a function to call the increment

These loops of 100 million increments can use pure C (check the object dump to make sure it’s consistent with the intent) or inline assembly. Throughout this process, CP0 will always be generating timer interrupts every second when the timer interrupts are enabled (using ‘e’). In addition, every keystroke will cause a UART receive interrupt.

Once the incrementing is done, the application code will count the elapsed cycles. This can be done by using the cycle counting from CP2 or by using mfc0 instructions to get the value of Count before the benchmark begins and after it ends and taking the difference.

Then, there will have a call to sprintf to generate a string summarizing the elapsed time. This will use a declared char* and make a string using an input like sprintf(strname,"%c : %d",STATE,count_diff).

This string will be the copied to the static software FIFO memory location, and one write will be attempted to the UART (as discussed in section 5). Then, subsequent UART Tx interrupts will handle sending the data. This copying and single UART transmit attempt are encapsulated in the ‘out’ function used (but not defined) in the example code.

8 Example code, tips, and recommendations

See handout on the agenda page. This is meant to be a guide to get groups going on thinking about how different parts of the ISR work and how the application space program generates data.

Before you start looking over the code, try to you modify your existing block diagram to determine where the hardware goes and what you need to add. In addition, try to determine how all control signals datapath signals should be connected. Then, think about the overall structure of the ISR and the application code that uses it.

Programs in this checkpoint can be written in assembly or C: you could write the ISR in assembly and the main program in C. Adding MIPS assembly instructions to a C program requires adding asm("<instr>");
to the appropriate location in your C code. For instance, adding asm("mfc0 $26,$13"); results in the instruction mfc0 $26,$13 being added to the assembly at the appropriate place.

Start early on homework 8 – this will be key for finishing the checkpoint on time. We also recommend that you look over the example code and make sure that you understand its intent.

You will need to know some C programming skills specific to embedded programming. The use of the software FIFO to put data into an output buffer that is drained through DataInReady-triggered interrupts is quite powerful, but deeper knowledge of pointers and the volatile keyword is necessary for a successful implementation.
9 Checkpoint Specifications & Deliverables

Your job is to perform all hardware integration, to write an ISR program that will be transferred to the isr_mem (via coe_to_serial isr.coe c0000180), and to write the program that runs from the instruction cache (via coe_to_serial <program_name>.coe 30000000) and leverages interrupts to perform benchmarking functionality. How you do this will depend on your design.

- **Hardware Integration**
  - Implement the new instructions for CP0 accesses and integrate the necessary hardware and control logic into your design for interrupts.
  - The COP0150 module has been provided for you. Connect the correct inputs to the coprocessor module and be careful when determining the peripheral input signals.

- **Timer**
  - Implement a 1 second timer (using Compare) and have the ISR send the current time out through the UART. The current time should be computed from the values of the hardware Count register and the software RTC register.
  - The time should be printed in mm:ss format where mm is the two digit minute value and ss is the two digit seconds value.
  - If you can't get the normalized time values for minutes and seconds you may print your software RTC value for ‘mm’ and the value of the hardware Count register divided by clock frequency for ‘ss’.

- **Timer Enable and Disable**
  - You must be able to disable the timer interrupts by pressing the d key and enable the timer interrupts by pressing the e key.
  - When the timer interrupt is disabled, the clock time should not print. However, the time should still increase in the background so that when you re-enable the interrupts for the time should still print correctly. For example, if you disable interrupts after you see 00:05, then wait 5 seconds, and re-enable interrupts, you should see 00:10, not 00:06. Thus, the RTC interrupts should also still update the software RTC and the Count should still be getting incremented.

- **Interrupt-based UART Tx/Rx**
  - Assert UART0Request for one clock cycle after DataOutValid goes high.
  - Assert UART1Request for one clock cycle after DataInReady goes high.
  - You should be able to tell from the bits in ‘Cause’ whether an interrupt was thrown by the transmitter or the receiver.
  - When your UART receiver throws an interrupt, the section of your ISR that handles a UART Rx interrupt must handle the data appropriately – changing the timer interrupt mask (‘d’, ‘e’), updating ‘STATE’ (‘r’, ‘R’, ‘v’, or ‘V’), or leaving the mode unchanged (other characters).
  - When the interrupt from DataInReady is thrown, the ISR should store the next character in the software FIFO (if it is not empty) to the UART and your ISR should handle incrementing the ‘out’ index of the software FIFO.

- **Benchmarking – putting it all together**
  - The Count register in the coprocessor0 module is a built in counter.
– You will need to use Count (with mfc0) in your application code to determine how long it takes to various types of increments (value in register, value in register with subroutine call, value in memory, and value in memory with subroutine calls) 100 million times.

– Your application code should format this result as a string and put the string in the software output FIFO.

– The UART Tx interrupts will then send this data out over the UART without polling of DataInReady by the application code.

– The UART will be used to determine which mode the program counts in:
  * Pressing ‘r’ should enable the mode where the count uses the register
  * Pressing ‘v’ should enable the mode where the count uses the memory location
  * Pressing ‘R’ should enable the mode where the count uses the register and a function to call the increment
  * Pressing ‘V’ should enable the mode where the count uses the memory location and a function to call the increment

10 Appendix: FSM for UART request inputs

The UART0Request and UART1Request inputs of CP0 should only be asserted for one clock cycle to let CP0 know that DataOutValid or DataInReady output of the UART has just turned on.

A simple Mealy machine to implement this behavior is shown below, where the input is one of the two UART control signals and the output is the corresponding UART request signal. What should the reset behavior be?

**References used in preparing this document**
MIPS M4K Software Users Manual, available from mips.com