



Department of Electrical and Computer Engineering  
Rutgers, The State University of New Jersey

## Computer Architecture and Assembly Language Lab

Fall 2016

---

### Lab 4

## CPU Structure, Pipeline Programming and Hazards, Exceptions and Interrupts

---

### Goal

After completing this lab, you will:

- ☐ Know the difference between pipelined and no-pipelined processors
  - ☐ Learn how the pipeline works; know how to prevent hazards in the pipeline
  - ☐ Know the exception mechanism in MIPS
  - ☐ Be able to write an interrupt handler for a MIPS machine
- 

### Preparation

Please read Chapter 4 and Appendices A.7 and A.8 in the textbook. This knowledge is required for this lab.

---

### Introduction

A CPU has 5 basic components, the Arithmetic and Logical Unit (ALU), the Instruction Memory, the Register File, the Data Memory and its Control. The ALU is responsible for executing the arithmetic functions, such as addition, multiplication, subtraction, and the logical functions, e.g. AND, OR, logical shift etc. The instruction memory is the memory where an assembly program is stored before execution. The Register File contains the state of the processor. The data memory is the memory where all the data produced or not are stored. Finally, the control is responsible for decoding an instruction and



Department of Electrical and Computer Engineering  
Rutgers, The State University of New Jersey

setting all necessary control signals with the proper value. In figure 1 a diagram of this simple CPU is shown. Computers execute instructions in a serial manner. A way to speed up a program's execution is by using Pipeline. Pipeline is a method to increase the throughput of the executing instruction without actually affecting the time a single instruction needs to execute. Finally, control is the most challenging aspect of processor design: it is both the hardest part to get right and the hardest part to make fast. One of the difficult parts of control design is implementing *exceptions* and *interrupts*. An exception is an unscheduled event that disrupts program execution. An interrupt is an exception that comes from outside of the processor [1].

## CPU Structure

In order to examine the structure of the MIPS CPU, the implementation of actual instructions must be analyzed, so that the organization of the memory, the ALU and the control of the CPU are understood. In figure 1, an overview of the parts of a CPU is depicted. The most important parts are:

1. The Program Counter (PC) points to the position where the instruction that is to be executed in the instruction memory.
2. The Registers contain the state of the processor in every clock cycle.
3. The Arithmetic and Logical Unit (ALU) is responsible to execute every arithmetic or logical operation that is needed or each command.
4. The Data Memory contains all the data that either are generated by the execution of the instructions or are the initial data that a program may need.
5. The Instruction Memory where the compiled and linked instructions are stored before execution.
6. The Control of the processor decodes each instruction and sets all the control signals according to the instruction.
7. The ALU control is responsible to pass the correct control signal to the ALU according to the arithmetic or logical operation that is requested by the Control unit.

In order to understand how an instruction is executed and how the datapath operates, two examples will be given, the first explaining the instruction *add \$t1, \$t2, \$t3* and the second the instruction *lw \$t1, offset(\$t2)*. Table 1 shows the values the control signals have for the two examples.

Example 1

*add \$t1, \$t2, \$t3*



Department of Electrical and Computer Engineering  
Rutgers, The State University of New Jersey

The execution can be shown in four steps [1]:

1. The instruction is fetched and the PC is incremented.
2. Two registers,  $\$t2$  and  $\$t3$ , are read from the register file. Also, the main control unit computes the setting of the control lines.
3. The ALU operates on the data read from the register file, using the function code (bits :0 of the instruction) to generate the ALU function.
4. The result of the ALU is written into the register file using the bits 15:11 of the instruction to select the destination register  $\$t1$ .

Example 2

*lw  $\$t1$ , offset( $\$t2$ )*

The execution can be shown in 5 steps [1]:

1. The instruction is fetched and the PC is incremented.
2. A register ( $\$t2$ ) value is read from the register file.
3. The ALU computes the sum of the value read from the register file and the sign-extended, lower 16 bits of the instruction (*offset*).
4. The sum of the ALU is used as the address for the data memory.
5. The data from the memory unit is written into the register file; the register destination is given by bits 20:16 of the instruction ( $\$t1$ ).

Instruction	RegDst	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Brach	ALUOp [1:0]
<i>add <math>\\$t1</math>, <math>\\$t2</math>, <math>\\$t3</math></i>	1	0	0	1	0	0	0	10
<i>lw <math>\\$t1</math>, offset(<math>\\$t2</math>)</i>	0	1	1	1	1	0	0	00

**Table 1** The setting of the control lines of examples 1 and 2 based on figure 1



Department of Electrical and Computer Engineering  
Rutgers, The State University of New Jersey

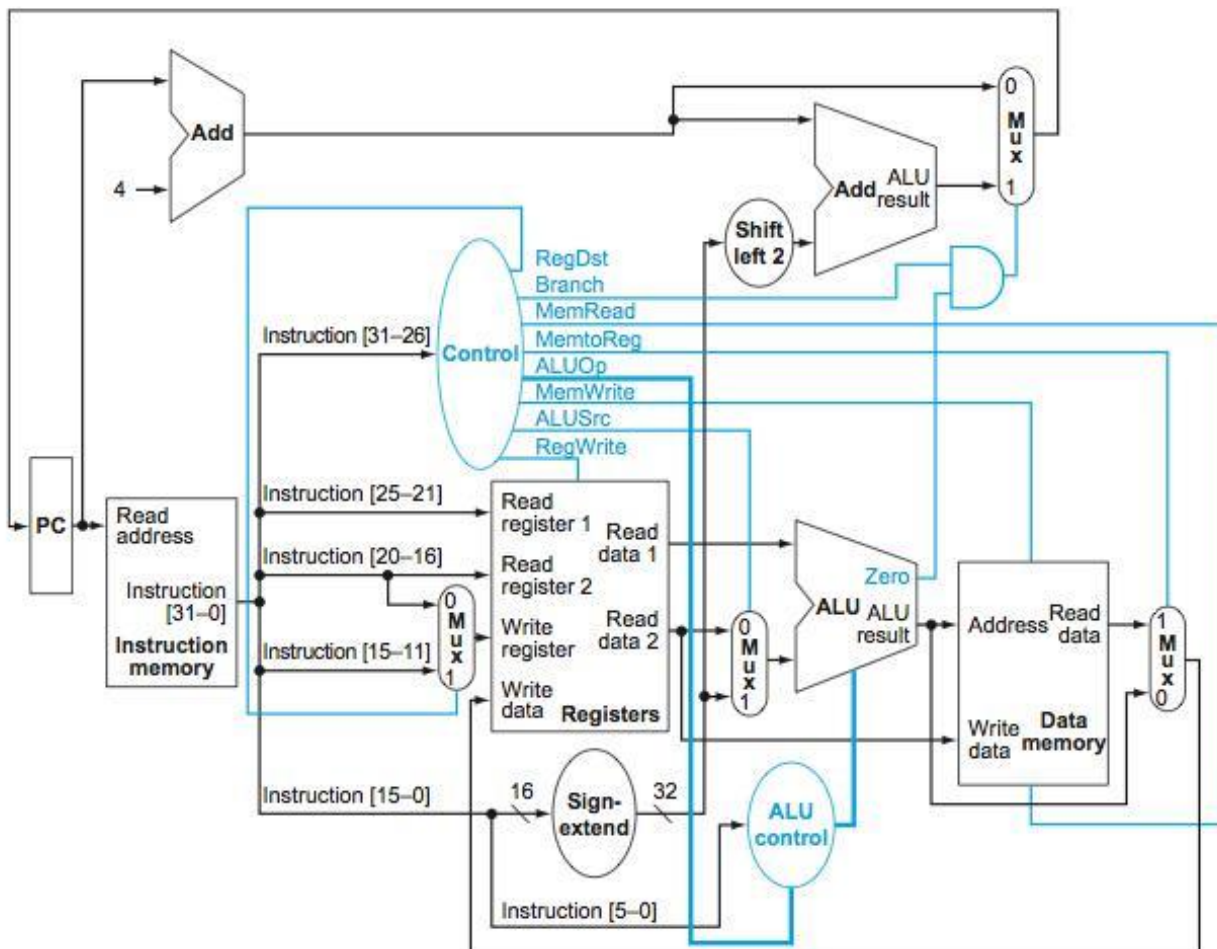


Figure 1 The simple datapath with the control unit

The explanation of the control signals is as follows:

Signal Name	Explanation
<b>RegDst</b>	Selects the register destination number either from rt field (bits 20:16) or the rd field (bits 15:11) of the instruction
<b>ALUSrc</b>	Selects the second input of the ALU either from the second output of the register file or the sign extended lower 16 bits of the instruction



Department of Electrical and Computer Engineering  
Rutgers, The State University of New Jersey

<b>MemtoReg</b>	Selects the data fed to the register Write data input either from the ALU output or the data memory output
<b>RegWrite</b>	The register on the Write register input is written with the value on the Write data input
<b>MemRead</b>	Data memory contents designated by the address input are put on the Read data output
<b>MemWrite</b>	Data memory contents designated by the address input are replaced by the value on the Write data input
<b>Branch</b>	Indicates if the instruction is a branch instruction
<b>ALUOp[1:0]</b>	The ALU operation. See Figure 4.12 of the text book, pp 260

## Pipeline Programming and Hazards

When a processor supports pipeline, an instruction can be divided in four or five steps which are 1) Instruction Fetch [IF], 2) Instruction Decode [ID], 3) Execution [EX], 4) Data Memory Access [MEM] and 5) Write Back the result to the register file [WB]. For a processor to achieve its maximum throughput the pipeline must always be kept full. Also, the clock cycle time is defined by the slowest step of an instruction.

Let us now see an example. Consider 3 consecutive instructions A, B and C. Each instruction has the above-mentioned five steps. When a processor does not support pipeline each instruction will be executed after the previous one has finished, as it is shown in Table 2. Also, suppose that instruction A start at cycle 1

Instr/Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	IF	ID	EX	MEM	WB										
B						IF	ID	EX	MEM	WB					
C											IF	ID	EX	MEM	WB

Table 2 Instruction Execution without Pipeline

On the other hand, when pipeline is supported each instruction can start executing even though the previous one has not necessarily finished. Table 3 shows this scheme. Again, instruction A starts executing in cycle 1.



Department of Electrical and Computer Engineering  
Rutgers, The State University of New Jersey

Inst/Cycle	1	2	3	4	5	6	7	8
A	IF	ID	EX	MEM	WB			
B		IF	ID	EX	MEM	WB		
C			IF	ID	EX	MEM	WB	

Table 3 Instruction Execution with pipeline support

There are, though, situations in the pipelining when the next instruction cannot execute in the following clock cycle. These events are called Hazards and there are three different types [1]:

- Structural Hazard. When a planned instruction cannot execute in the proper clock cycle because the hardware does not support the combination of instructions that are set to execute [1].
- Data Hazard or Pipeline Data Hazard. When a planned instruction cannot execute in the proper clock cycle because data that is needed to execute the instruction is not yet available [1].
- Control Hazard or Branch Hazard. When the proper instruction cannot execute in the proper pipeline clock cycle because the instruction that was fetched is not the one that is needed; that is, the flow of instruction addresses is not what the pipeline expected [1].

To understand a Data Hazard, consider the following code fragment:

```
lb $t0, 1($t1)
```

```
add $t2, $t0, $t1
```

The execution of the code with pipeline support is as follows:

Instr/Cycle	1	2	3	4	5	6
<i>lb \$t0, 1(\$t1)</i>	IF	ID	EX	MEM	WB	
<i>add \$t2, \$t0, \$t1</i>		IF	ID	EX	MEM	WB

The *lb* will write the value from the memory in the register file in step WB, in the mean time the *add* will have retrieved its operands in the ID step. This actually means that the old value of register \$t0 will be used for the addition.

Such hazards can be prevented by either writing a program that does not provoke any hazard or by



Department of Electrical and Computer Engineering  
Rutgers, The State University of New Jersey

adding an ample number of no-operation instructions (NOP) between the instructions that depend on the result of each other. NOP operations act like bubbles inside the pipeline delaying the execution of the second instruction (and subsequent instructions) until the values it requires as input are ready. When the NOP technique is used in the above example, the following happens:

Instr/Cycl	1	2	3	4	5	6	7	8
lb \$t0, 1(\$t1)	IF	ID	EX	MEM	WB			
add \$t2, \$t0, \$t1		IF	Stall	Stall	ID	EX	MEM	WB

An alternative technique to the use of NOP operations is Instruction Forwarding. In this technique, the result of an instruction will be sent forward to the next instruction before it is written back to the register file and it is possible that stalling is not fully needed. The above example can become as the following with instruction forwarding:

Instr/Cycl	1	2	3	4	5	6	7	8
lb \$t0, 1(\$t1)	IF	ID	EX	MEM	WB			
add \$t2, \$t0, \$t1		IF	Stall	ID	EX	MEM	WB	

It can be easily seen that a Stall cycle still exists. The reason is that the *lb* instruction will not have the data ready for use before it is read from the memory. When it writes back the value in the register file, it also forwards it to the input of the ALU for immediate use. If that stall does not exist even with instruction forwarding the forwarded result would arrive too late and the result of the *add* instruction would be wrong.

The third technique that a programmer can use to avoid data hazards is “Instruction Reordering”. As it implied by its name, the independence between instructions can be exploited in order to reorder them in a way that keeps the pipeline as full as possible.

Consider the following code segment:

```
lw $t0, 0($t1)
addi $t2, $t0, 2
subi $t3, $t4, 2
```



Department of Electrical and Computer Engineering  
Rutgers, The State University of New Jersey

Without instruction reordering the execution will be the following:

Instr/Cycle	1	2	3	4	5	6	7	8	9
lw \$t0, 0(\$t1)	IF	ID	EX	MEM	WB				
add \$t2, \$t0, 2		IF	Stall	Stall	ID	EX	MEM	WB	
subi \$t3, \$t4, 2			IF	Stall	Stall	ID	EX	MEM	WB

It is clear that two cycles are lost due to the NOP instructions. Also, it is easily seen that *subi* does not need the result of the *lw* or the *add*. In this case *subi* is independent of the other two and it can be move between them. After reordering the execution will look like this:

Instr/Cycle	1	2	3	4	5	6	7	8
lw \$t0, 0(\$t1)	IF	ID	EX	MEM	WB			
subi \$t3, \$t4, 2		IF	ID	EX	MEM	WB		
add \$t2, \$t0, 2			IF	Stall	ID	EX	MEM	WB

### Exceptions and Interrupts

The MIPS convention calls an *exception* any unexpected change in control flow regardless of its source (i.e. without distinguishing between a cause within the processor and an external cause).

#### □ The MIPS exception mechanism[3]

The method implemented by the MIPS designers to interrupt the currently running program is to include some additional hardware referred to as **coprocessor 0**. This coprocessor contains a number of specialized registers that can be accessed at the assembly language level for exception handling. The top window of the *QtSpim* simulator displays these registers:





Department of Electrical and Computer Engineering  
Rutgers, The State University of New Jersey

Register Number in coprocessor 0	Register Name	Usage
8	<b>BadVAddr</b>	Virtual Memory address where exception occurred
12	<b>Status</b>	Interrupt mask, enable bits, and status when exception occurred
13	<b>Cause</b>	Type of exception and pending interrupt bits
14	<b>EPC</b>	Address of instruction that caused exception

The **BadVAddr (Bad Virtual Address) register** will contain the memory address where the exception has occurred. An unaligned memory access, for instance, will generate an exception and the address where the access was attempted will be stored in BadVAddr.

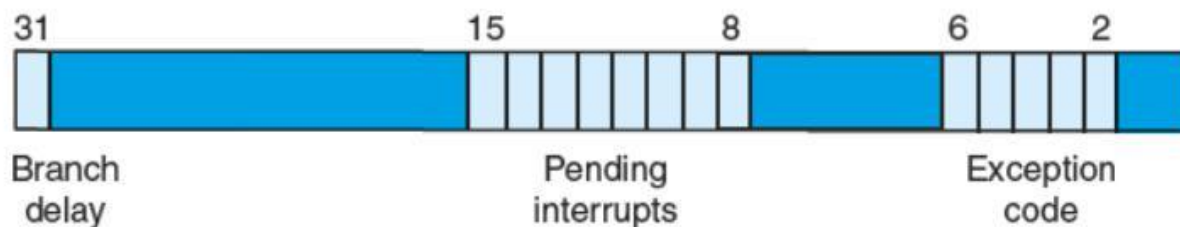


Figure 2 The layout of the Cause Register

The **Cause register** provides information about the level of pending interrupts (IP2 to IP7) and the cause of the exception. The exception code is stored as an unsigned integer using bits 6-2 in the Cause register. The layout of the Cause register is presented above.

Bit  $IP_i$  becomes 1 if an interrupt has occurred at level  $i$  and is pending (has not been serviced yet). The bits  $IP_1$  and  $IP_0$  are used for simulated interrupts that can be generated by software.

The *exception code* indicates what caused the exception.

Number	Name	Cause of exception
0	Int	Interrupt (Hardware)
4	AdEL	Address Error Exception (load or instruction fetch)
5	AdES	Address Error Exception (store)



Department of Electrical and Computer Engineering  
Rutgers, The State University of New Jersey

6	IBE	Bus Error on Instruction Fetch
7	DBE	Bus Error on data load or store
8	Sys	Syscall Exception
9	Bp	Breakpoint Exception
10	RI	Reserved Instruction Exception
11	CpU	Coprocessor Unimplemented
12	Ov	Arithmetic Overflow Exception
13	Tr	Trap
15	FPE	Floating Point

Table 4 Exception codes supported by SPIM

**The Status register** contains an interrupt mask on bits 15-10 and status information on bits 5-0. The layout of the Status register is presented below.

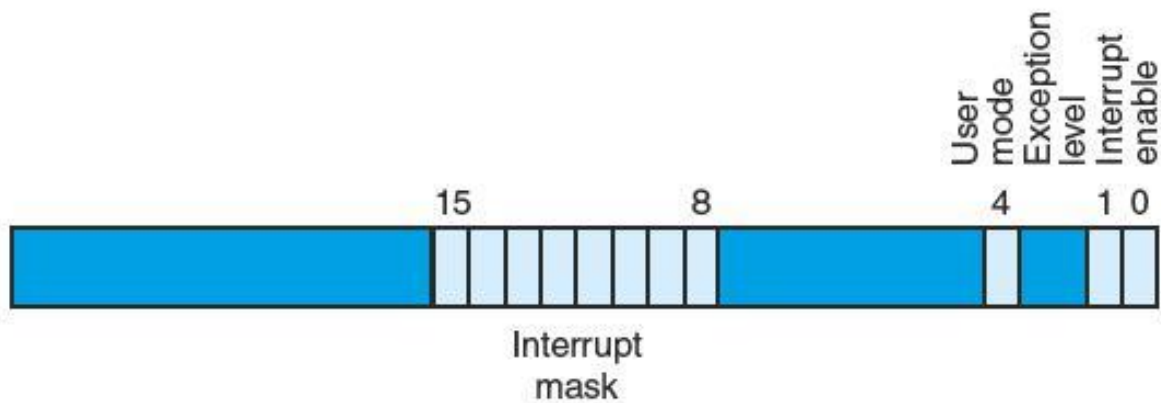


Figure 3 The layout of the Status Register

The interrupt mask field contains a bit for each of the six hardware and two software interrupt levels. If a mask bit is 1, interrupts are allowed at that level. For an interrupt to interrupt the processor both the pending bit in the Cause register and the mask bit must be enabled. The



Department of Electrical and Computer Engineering  
Rutgers, The State University of New Jersey

user mode bit is 0 if the processor runs in kernel mode and 1 when it runs in user mode. The exception bit becomes 1 when an exception has occurred, otherwise is 0. The interrupt enable bit is 1 when interrupts are allowed, otherwise it is 0 [1].

**The EPC register**, when a procedure is called using `jal`, two things happen:

- Control is transferred to the address provided by the instruction
- The return address is saved in the return address register `$ra`

In the case of an exception there is no explicit call. In MIPS the control is transferred at a fixed location, `0x80000180` when an exception occurs. The exception handler must be located at that address. The **Exception Program Counter (EPC)** is used to store the address of the instruction that was executing when the exception was generated.

The CPU operates in one of the two possible modes, **user** and **kernel**. User programs run in user mode. The CPU enters the kernel mode when an exception happens. **Coprocessor 0** can only be used in kernel mode. When running in kernel mode the registers of coprocessor 0 can be accessed using the following instructions:

Instruction	Comment
<code>mfc0 Rdest, C0src</code>	Move the content of coprocessor's register <code>C0src</code> to <code>Rdest</code>
<code>mtc0 Rsrc, C0dest</code>	Integer register <code>Rsrc</code> is moved to coprocessor's register <code>C0dest</code> (Be careful! The destination is the right-hand one)
<code>lwc0 C0dest, address</code>	Load word from <code>address</code> into register <code>C0dest</code>
<code>swc0 C0src, address</code>	Store the content of register <code>C0src</code> to <code>address</code> in memory

Table 5 Instructions for accessing coprocessor 0 registers



Department of Electrical and Computer Engineering  
Rutgers, The State University of New Jersey

---

### **Assignment 1**

Suppose you have the CPU of Figure 1 and that there is not any pipeline scheme. For the following program indicate the values of the control signals for each instruction.

*addiu \$t0, 10*

*lb \$t1, 32(\$t0)*

*or \$t2, \$t0, \$t1*

*bne \$t2, \$t0, QUIT*

*j 2500*

### **Assignment 2**

Suppose you have a machine that supports the five pipeline stages mentioned above. Using the following programs to answer the questions:

*lw \$t0, 0(\$t3)*

*add \$t1, \$t0, \$t2*

*sub \$t3, \$t3, \$t1*

*addi \$t4, \$t4, 4*

*sub \$t5, \$t5, \$t4*

1. Complete the table of execution with five pipeline stages. Ignore all hazards.
2. Suppose that register \$t0, \$t1, \$t2, \$t3, and \$t4 have the respective values of 2, 5, 8, 2, and 4. Give the values of the registers after running the program without the processor handling any hazards.
3. Insert the necessary NOP instructions so that the registers would get the values as when the processor executes the program one instruction per cycle.
4. Discuss how instruction reordering can avoid the hazards in the code section above.
5. Discuss how forwarding can affects the execution of the code section above.



Department of Electrical and Computer Engineering  
Rutgers, The State University of New Jersey

Compare it with NOP and instruction reordering.

### **Assignment 3**

Consider the following code:

```
addiu $t0,3  
loop: $t0, $t0, 1  
bne $t0, $zero, loop
```

1. Determine how many cycles would it take for the program to execute without hazards handling.
2. Draw the execution table and determine the type of hazards.
3. Write program using NOP instructions to avoid the hazards.
4. Describes the effects of instruction forwarding to this code.

### **Assignment 4**

File “**io.s**” lets you read characters from and echo characters to the console by using interrupts and polling. When you run the program, you need to run it step by step and try to see the changes in the Status and Cause Registers (they can be found on the top left of QtSPIM).

Enable the memory-mapped I/O, load the file “**io.s**” and run the program. When running it, write to the console 6 lines of numbers or text. For example:

```
This  
Is  
Computer  
Architecture  
Lab  
4
```

Try to see the changes that occur. Write your observations and discuss what the differences between polling and interrupts are. Explain how the interrupts are enabled. Are the interrupts enabled or disabled during polling? Include what would happen if they were enabled. Also, check the registers that save the changes before and after executing the exception. Analyze the exception response code.

*Note: Before doing this assignment, read Appendix A of the manual. In Appendix B you will find the ASCII codes for all the characters.*



Department of Electrical and Computer Engineering  
Rutgers, The State University of New Jersey

## Assignment 5

File "**quad\_sol.s**" contains a quadratic polynomial solver, which calculates the integer solution of a quadratic polynomial equation.

1. Rewrite the program using instructions reordering to reduce the number of cycles needed to execute the program. Indicate the number of cycle reduction.
2. Describe how forwarding would affect the execution of the program.

---

## Experiment report

Write a proper report including your codes, results (snapshot of output) and the conclusion of assignments and convert it to **pdf** format. Please also attach the **code** files (\*.s, \*.asm) to **Sakai** together with the report. Each lab report is **due** before the start of the next lab. Please include your name and Student ID in both report and the code.

---

## References

1. Patterson and Hennessy, "Computer Organization and Design: The Hardware / Software interface", 5<sup>th</sup> Edition.
  2. Daniel J. Ellard, "MIPS Assembly Language Programming: CS50 Discussion and Project Book", September 1994.
  3. "Exceptions in MIPS", [www.cs.iit.edu/~virgil/cs470/Labs/Lab7.pdf](http://www.cs.iit.edu/~virgil/cs470/Labs/Lab7.pdf)
  4. "Low-level IO", <http://courses.knox.edu/cs201/>
  5. Rabert Britton, "MIPS Assembly Language Programming", Pearson Education Inc. 2004
-



## APPENDIX A

### 1. SPIM Keyboard Input

The SPIM simulator for the MIPS processor has a device emulator that allows you to read characters from the keyboard. The keyboard I/O registers are mapped to the locations 0xFFFF0000 and 0xFFFF0004, called Receiver Control register and Receiver Data register respectively. The Receiver Control register has an input Ready bit – bit 0 – and an output Interrupt Enable bit – bit 1. If the Ready bit is 0, then there is no data waiting to be read from the keyboard, otherwise, if the Ready bit is 1, then there is data to be read from the keyboard. If the Interrupt Enable bit is set to 0, an interrupt is not triggered by the device. On the other hand, if the Interrupt Enable bit is set to 1, an interrupt is triggered whenever new data is ready to be read. The last byte of the Receiver Data register is an input byte representing the key pressed. When the Receiver Data register is read, it also sets the Ready bit to 0 until a new byte is received.

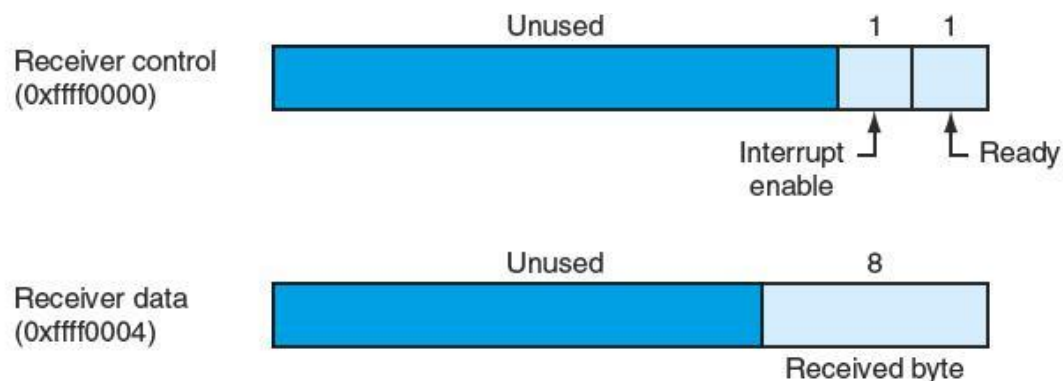


Figure 4 The Receiver Control and Data registers

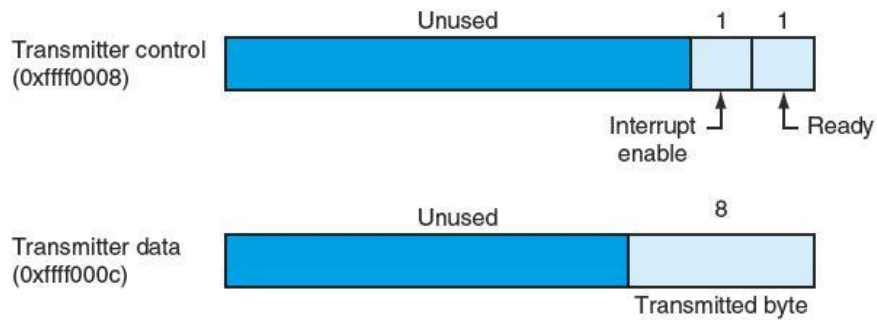
### 2. SPIM Terminal Output

The SPIM simulator for the MIPS processor has a device emulator that allows you to send characters to a terminal. The terminal I/O registers are mapped to the locations 0xFFFF0008 and 0xFFFF000C, which are called Transmitter Control register and Transmitter Data register respectively. The Transmitter Control register consists of an input Ready bit – bit 0 – and an output Interrupt Enable bit – bit 1. If the Ready bit is 0, then the terminal is not ready to receive any data, otherwise, if it is 1, the terminal is accepting data. If the Interrupt Enable bit is set to 0, an interrupt is not triggered by the device, otherwise if it is set to 1 an interrupt is triggered whenever the terminal is accepting new data. The last byte of the Transmitter Data register is an output byte representing the data being sent to the



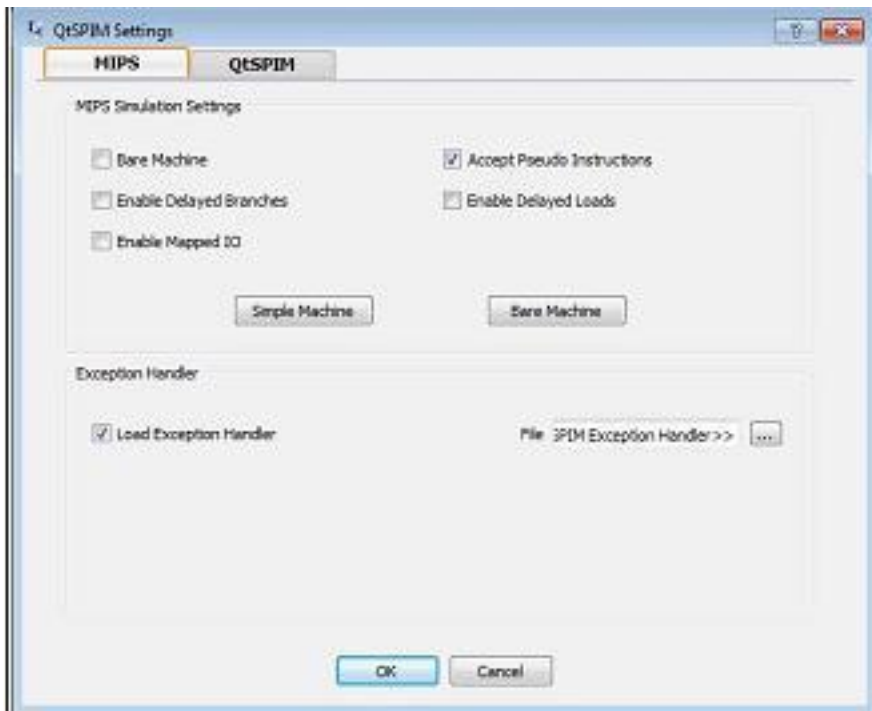
Department of Electrical and Computer Engineering  
Rutgers, The State University of New Jersey

terminal. When a byte is written to the Transmitter data register, it sets the ready bit to 0 until a new byte is ready to be sent.



**Figure 5 The Transmitter Control and Data registers**

3. In order to enable the memory-mapped I/O you need to find the Setting command under the Simulator menu. In the MIPS tab of the QtSPIM Setting window check the box "Enable Mapped IO".







Department of Electrical and Computer Engineering  
Rutgers, The State University of New Jersey

## APPENDIX B. ASCII Value Table

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	<b>NUL</b> (null)	32	20	040	&#32;	<b>Space</b>	64	40	100	&#64;	<b>@</b>	96	60	140	&#96;	<b>`</b>
1	1	001	<b>SOH</b> (start of heading)	33	21	041	&#33;	<b>!</b>	65	41	101	&#65;	<b>A</b>	97	61	141	&#97;	<b>a</b>
2	2	002	<b>STX</b> (start of text)	34	22	042	&#34;	<b>"</b>	66	42	102	&#66;	<b>B</b>	98	62	142	&#98;	<b>b</b>
3	3	003	<b>ETX</b> (end of text)	35	23	043	&#35;	<b>#</b>	67	43	103	&#67;	<b>C</b>	99	63	143	&#99;	<b>c</b>
4	4	004	<b>EOT</b> (end of transmission)	36	24	044	&#36;	<b>\$</b>	68	44	104	&#68;	<b>D</b>	100	64	144	&#100;	<b>d</b>
5	5	005	<b>ENQ</b> (enquiry)	37	25	045	&#37;	<b>%</b>	69	45	105	&#69;	<b>E</b>	101	65	145	&#101;	<b>e</b>
6	6	006	<b>ACK</b> (acknowledge)	38	26	046	&#38;	<b>&amp;</b>	70	46	106	&#70;	<b>F</b>	102	66	146	&#102;	<b>f</b>
7	7	007	<b>BEL</b> (bell)	39	27	047	&#39;	<b>'</b>	71	47	107	&#71;	<b>G</b>	103	67	147	&#103;	<b>g</b>
8	8	010	<b>BS</b> (backspace)	40	28	050	&#40;	<b>(</b>	72	48	110	&#72;	<b>H</b>	104	68	150	&#104;	<b>h</b>
9	9	011	<b>TAB</b> (horizontal tab)	41	29	051	&#41;	<b>)</b>	73	49	111	&#73;	<b>I</b>	105	69	151	&#105;	<b>i</b>
10	A	012	<b>LF</b> (NL line feed, new line)	42	2A	052	&#42;	<b>*</b>	74	4A	112	&#74;	<b>J</b>	106	6A	152	&#106;	<b>j</b>
11	B	013	<b>VT</b> (vertical tab)	43	2B	053	&#43;	<b>+</b>	75	4B	113	&#75;	<b>K</b>	107	6B	153	&#107;	<b>k</b>
12	C	014	<b>FF</b> (NP form feed, new page)	44	2C	054	&#44;	<b>,</b>	76	4C	114	&#76;	<b>L</b>	108	6C	154	&#108;	<b>l</b>
13	D	015	<b>CR</b> (carriage return)	45	2D	055	&#45;	<b>-</b>	77	4D	115	&#77;	<b>M</b>	109	6D	155	&#109;	<b>m</b>
14	E	016	<b>SO</b> (shift out)	46	2E	056	&#46;	<b>.</b>	78	4E	116	&#78;	<b>N</b>	110	6E	156	&#110;	<b>n</b>
15	F	017	<b>SI</b> (shift in)	47	2F	057	&#47;	<b>/</b>	79	4F	117	&#79;	<b>O</b>	111	6F	157	&#111;	<b>o</b>
16	10	020	<b>DLE</b> (data link escape)	48	30	060	&#48;	<b>0</b>	80	50	120	&#80;	<b>P</b>	112	70	160	&#112;	<b>p</b>
17	11	021	<b>DC1</b> (device control 1)	49	31	061	&#49;	<b>1</b>	81	51	121	&#81;	<b>Q</b>	113	71	161	&#113;	<b>q</b>
18	12	022	<b>DC2</b> (device control 2)	50	32	062	&#50;	<b>2</b>	82	52	122	&#82;	<b>R</b>	114	72	162	&#114;	<b>r</b>
19	13	023	<b>DC3</b> (device control 3)	51	33	063	&#51;	<b>3</b>	83	53	123	&#83;	<b>S</b>	115	73	163	&#115;	<b>s</b>
20	14	024	<b>DC4</b> (device control 4)	52	34	064	&#52;	<b>4</b>	84	54	124	&#84;	<b>T</b>	116	74	164	&#116;	<b>t</b>
21	15	025	<b>NAK</b> (negative acknowledge)	53	35	065	&#53;	<b>5</b>	85	55	125	&#85;	<b>U</b>	117	75	165	&#117;	<b>u</b>
22	16	026	<b>SYN</b> (synchronous idle)	54	36	066	&#54;	<b>6</b>	86	56	126	&#86;	<b>V</b>	118	76	166	&#118;	<b>v</b>
23	17	027	<b>ETB</b> (end of trans. block)	55	37	067	&#55;	<b>7</b>	87	57	127	&#87;	<b>W</b>	119	77	167	&#119;	<b>w</b>
24	18	030	<b>CAN</b> (cancel)	56	38	070	&#56;	<b>8</b>	88	58	130	&#88;	<b>X</b>	120	78	170	&#120;	<b>x</b>
25	19	031	<b>EM</b> (end of medium)	57	39	071	&#57;	<b>9</b>	89	59	131	&#89;	<b>Y</b>	121	79	171	&#121;	<b>y</b>
26	1A	032	<b>SUB</b> (substitute)	58	3A	072	&#58;	<b>:</b>	90	5A	132	&#90;	<b>Z</b>	122	7A	172	&#122;	<b>z</b>
27	1B	033	<b>ESC</b> (escape)	59	3B	073	&#59;	<b>;</b>	91	5B	133	&#91;	<b>[</b>	123	7B	173	&#123;	<b>{</b>
28	1C	034	<b>FS</b> (file separator)	60	3C	074	&#60;	<b>&lt;</b>	92	5C	134	&#92;	<b>\</b>	124	7C	174	&#124;	<b> </b>
29	1D	035	<b>GS</b> (group separator)	61	3D	075	&#61;	<b>=</b>	93	5D	135	&#93;	<b>]</b>	125	7D	175	&#125;	<b>}</b>
30	1E	036	<b>RS</b> (record separator)	62	3E	076	&#62;	<b>&gt;</b>	94	5E	136	&#94;	<b>^</b>	126	7E	176	&#126;	<b>~</b>
31	1F	037	<b>US</b> (unit separator)	63	3F	077	&#63;	<b>?</b>	95	5F	137	&#95;	<b>_</b>	127	7F	177	&#127;	<b>DEL</b>