Lab 5: MIPS

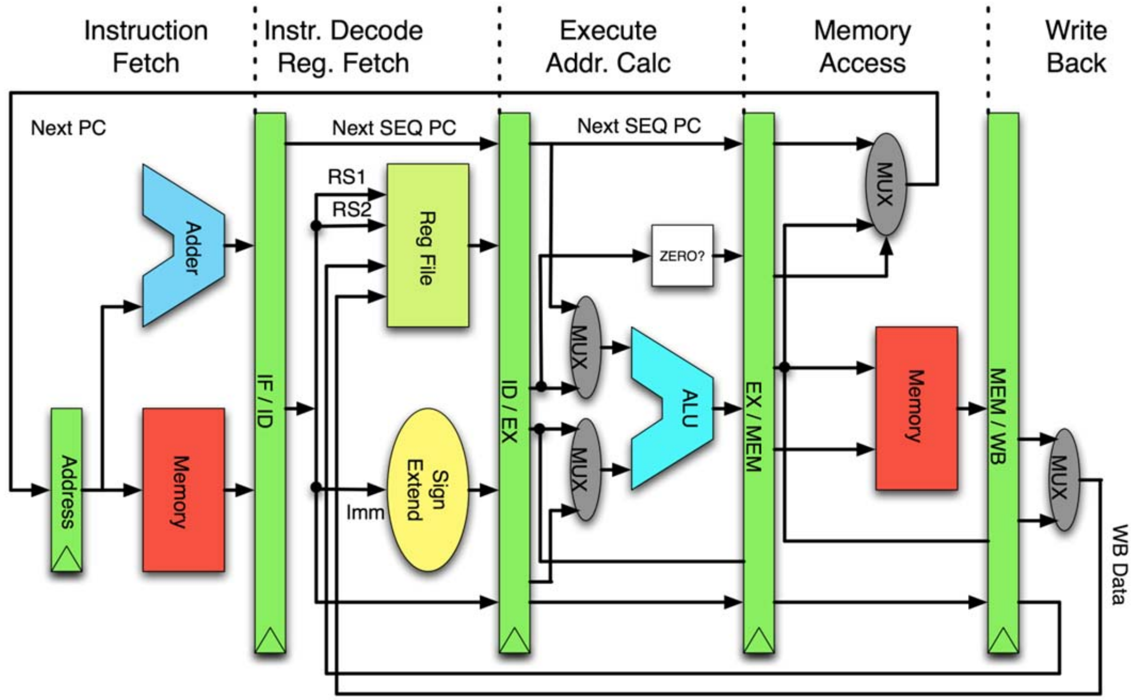
Background: Machine Language

We have now brought a simplified version of the Pascal programming language to life by writing a Pascal interpreter in Java. But what brought our Java program to life? First, we wrote our program in a text file with the .java extension. Then we compiled that code into a .class file containing Java bytecode instructions. These instructions were then run by the Java Virtual Machine (JVM), which means the JVM itself must be an interpreter. The rest of the story depends on the machine you're using. Perhaps the JVM was written in C and compiled to the x86 code that your Pentium processor runs. In fact, your processor is really just an interpreter written in hardware—transistors, resistors, wires, etc. Each processor is designed to interpret a particular programming language, such as AMD64, IA-64, MIPS, 68k, VAX, x86, Chumpanese, or possibly even Java bytecode. These languages typically support:

- only binary values—any other types must ultimately be represented in binary
- only values consisting of the same fixed number of bits—larger data types and structures must be represented as multiple data values
- only a fixed number of variables (called registers)—additional data must be kept in RAM
- only single-operation instructions of a fixed size—complex expressions, nested statements, procedure calls, etc., must be represented as sequences of instructions

In this course, we'll be studying the MIPS instruction set. The MIPS processor (Microprocessor without Interlocked Pipeline Stages) was developed by a team at Stanford University in 1981. This very popular processor chip has appeared in TiVo, Cisco routers, Nintendo 64, and Sony Playstation.

If you're curious, here's a diagram of the MIPS processor architecture (as found on Wikipedia).



Here's our first MIPS program. It consists of 3 instructions, each of which is represented by 32 bits. In fact, all MIPS instructions are 32-bit values, as are the MIPS data values they manipulate.

```
00110100000010000000000000000010
00110100000010010000000000000011
00000001000010010101000000100001
```

A MIPS processor sees each 32-bit value as 32 different voltage levels, each carried by a different wire. The processor responds to these voltages in the manner called for by the particular instruction.

But what do these 32-bit values mean? The following listing shows the binary values grouped by their purpose.

```
001101 00000 01000 0000000000000010
001101 00000 01001 0000000000000011
000000 01000 01001 01010 00000 100001
```

The opcode (operation code) `001101` in the first instruction tells the computer to load the immediate (constant) value `0000000000000010` (decimal value 2) into register number `01000`. Since our register stores a 32-bit value, this 16-bit value is padded with 16 extra zeroes. Likewise, the second instruction loads the decimal value 3 into register `01001`. The opcode `000000` and ending `100001` in the third instruction tells the processor to add the value in register `01000` to the value in register `01001`, and to store the result in register `01010`. In other words, we've just computed 2 + 3.

Even low-level programmers rarely program directly in binary. Instead, they use mnemonics to represent the various machine instructions, like the ones used below to describe our simple 2 + 3 program.

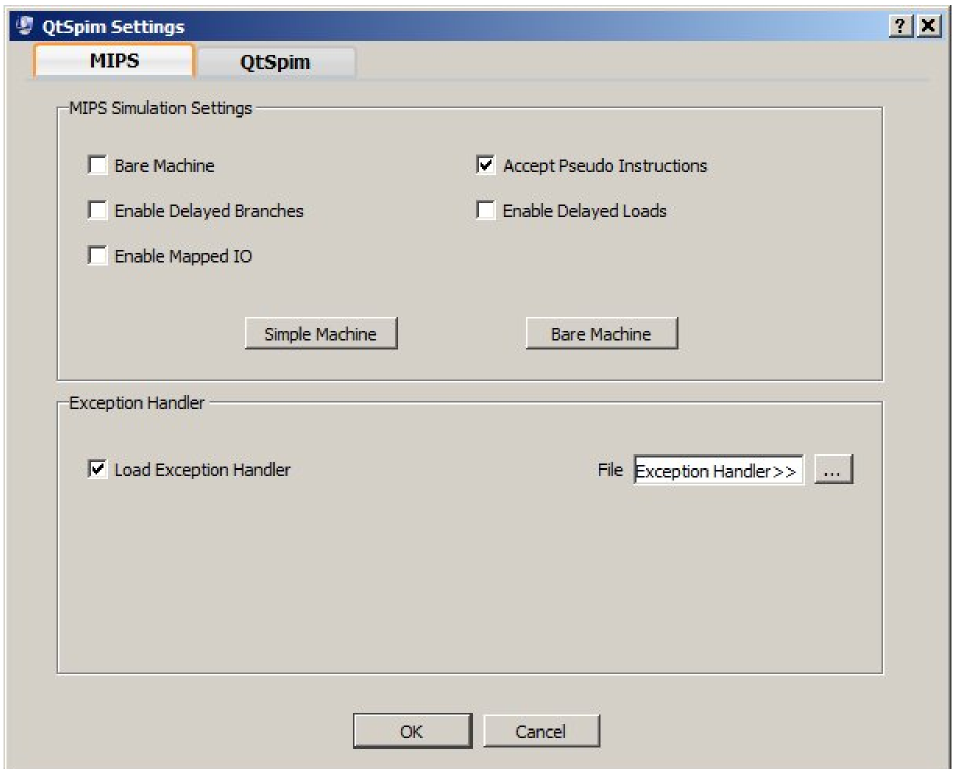
```
li $t0, 2
li $t1, 3
addu $t2, $t0, $t1
```

Here, `$t1`, `$t2`, and `$t3` are register names. `li` loads the immediate value `2` into register `t0` and then `3` into `$t1`. `addu` stores their sum in `$t2`. This kind of language is called *assembly language*, and such programs still need to be *assembled*—translated into binary—before they can be run. Usually, this translation is performed by a program called an *assembler*. An assembler is far simpler than a full scale compiler, since there is a direct one-to-one correspondence between an assembly instruction and the equivalent binary instruction.

Exercise 1: Setting Up SPIM

Since our processors don't process MIPS instructions, we will use an emulator to run our programs. This emulator is called QTSPIM, and you can find it at http://www.cs.wisc.edu/~larus/spim.html. Download and install the appropriate version on your machine.

Run the emulator, and open up the simulator -> settings dialog. It is recommended that you configure these settings as follows.



Exercise 2: Loading a Simple Program

Next, copy the following sample program to a text file. Save it as simple.asm.

```
    .text 0x00400000
    .globl main

main:
    li $t0, 2           # load 2 into $t0
    li $t1, 3           # load 3 into $t1
    addu $t2, $t0, $t1  # store $t0 + $t1 in $t2
    li $v0, 10
    syscall
```

You know what the 3 boldfaced lines do. The last two lines of our program tell the execution to halt. We will include these lines at the end of every program. Notice that you can use the # symbol to comment your code. Assembly code is very difficult to read, so you should plan on writing a comment every 2 or 3 lines.

Using the SPIM software, load simple.asm. You should see the following in the user text segment:

```
[0x00400000] 0x34080002 ori $8,$0,2    li $t0, 2
[0x00400004] 0x34090003 ori $9,$0,3    li $t1, 3
[0x00400008] 0x01095021 addu $10,$8,$9 addu $t2, $t0, $t1
```

`0x00400000` is the location in memory where the first line of your program is stored. It turns out that your `li` instruction has been expanded into an `ori` instruction, for reasons we don't particularly care about. Since `$t0` is register number `01000`, it appears as `$8`.

Your instruction has also been assembled into the base-16 (called "hexadecimal") value `0x34080002`. This is a more compact way of writing the 32-bit binary number you saw earlier. Since it takes 4 bytes (32 bits) to represent one instruction, our instruction actually appears occupies four locations in memory: `0x00400000`, `0x00400001`, `0x00400002`, and `0x00400003`. That is why the second instruction appears at memory address `0x00400004`.

Exercise 3: Running Your Program

Press F10 once to step through one instruction. What happens to the `PC` value at the top of the screen? Did you see the value 2 get loaded into register `$t0` at the top of the screen?

If you continue pressing F10, you can step through your program. Alternatively, you can simply run the entire program. Since our simple program has no output, running it won't be particularly exciting.

Background: MIPS

For now, we will only use temporary registers `$t0` - `$t9`, argument registers `$a0` - `$a3`, and return value register `$v0`. Later, we'll learn about registers `$ra` (return address), `$sp` (stack pointer), and `$fp` (frame pointer).

Here is a list of the MIPS instructions we'll be using. Of course, you will find more complete resources online.

Instruction	Description
`li` *reg*`,` *num*	*reg* = *num*
`move` *reg1*`,` *reg2*	*reg1* = *reg2*
`addu` *reg1*`,` *reg2*`,` *num* `addu` *reg1*`,` *reg2*`,` *reg3*	*reg1* = *reg2* + *num* *reg1* = *reg2* + *reg3*
`subu` *reg1*`,` *reg2*`,` *num* `subu` *reg1*`,` *reg2*`,` *reg3*	*reg1* = *reg2* - *num* *reg1* = *reg2* - *reg3*
`mult` *reg1*`,` *reg2*	compute *reg1* × *reg2*
`div` *reg1*`,` *reg2*	compute *reg1* ÷ *reg2*
`mflo` *reg*	*reg* = computed product/quotient
`syscall`	execute system call indicated by `$v0`
`la` *reg*`,` *label*	load address of *label* into *reg*
`beq` *reg1*`,` *num*`,` *label* `beq` *reg1*`,` *reg2*`,` *label*	if *reg1* = *num* goto *label* if *reg1* = *reg2* goto *label*
`bne` *reg1*`,` *num*`,` *label* `bne` *reg1*`,` *reg2*`,` *label*	if *reg1* ≠ *num* goto *label* if *reg1* ≠ *reg2* goto *label*
`blt` *reg1*`,` *num*`,` *label* `blt` *reg1*`,` *reg2*`,` *label*	if *reg1* < *num* goto *label* if *reg1* < *reg2* goto *label*
`bgt` *reg1*`,` *num*`,` *label* `bgt` *reg1*`,` *reg2*`,` *label*	if *reg1* > *num* goto *label* if *reg1* > *reg2* goto *label*
`ble` *reg1*`,` *num*`,` *label* `ble` *reg1*`,` *reg2*`,` *label*	if *reg1* ≤ *num* goto *label* if *reg1* ≤ *reg2* goto *label*
`bge` *reg1*`,` *num*`,` *label* `bge` *reg1*`,` *reg2*`,` *label*	if *reg1* ≥ *num* goto *label* if *reg1* ≥ *reg2* goto *label*
`j` *label*	goto *label*
`sw` *reg1*`,(`*reg2*`)` `sw` *reg1*`,` *num*`(`*reg2*`)`	mem[*reg2*] = *reg1* mem[*num* + *reg2*] = *reg1*
`lw` *reg1*`,(`*reg2*`)` `lw` *reg1*`,` *num*`(`*reg2*`)`	*reg1* = mem[*reg2*] *reg1* = mem[*num* + *reg2*]
`jal` *label*	goto *label*, `$ra` = next instruction
`jr` *reg*	goto *reg*

Here are some of the most useful system calls:

Summary	`$v0`	Input	Output
print integer	1	integer in `$a0`	
print string	4	string address in `$a0`	
read integer	5		integer in `$v0`
read string	8	buffer address in `$a0` maximum length in `$a1`	
allocate memory	9	number of bytes in `$a0`	address in `$v0`
exit	10		

The classic "Hello World!" program below illustrates how to use a system call to print a string to the console. Notice that the string is listed in the ".data" section of the program. The label `msg` was chosen arbitrarily.

```
    .text 0x00400000
    .globl main
main:
    li $v0, 4
    la $a0, msg
    syscall
    li $v0, 10
    syscall

    .data
msg:
    .asciiz "Hello World!\n"
```

Exercise 4: Your Own Simple Program

Write a program to read user input perform a simple computation, and output the result.

Background: Jumping

Most instructions cause program execution to continue with the next line. The `j` instruction tells execution to *jump* (back or forward) to an instruction at a given *label*. For example:

```
    j later
    skipped instructions

  later:
    execution continues here
```

The `bxx` *branch* instructions are conditional jumps. Each one tests a certain condition. For example, `beq` tests if the value in a given register is equal to a given number (or to the value in another register). If the condition is true, execution continues at the designated label. Otherwise, the instruction has no effect, and execution continues at the next line. For example:

```
    beq $t0, 0, after
    code that is only executed if $t0 DOES NOT contain 0
  after:
    code that is executed regardless of value in $t0
```

We can use the `bxx` and `j` instructions together to create an if/else structure.

```
    bge $t5, 1, positive
    code that is only executed when $t5 < 1
    j after
  positive:
    code that is only executed when $t5 ≥ 1
  after:
    code that is executed regardless of value in $t5
```

Exercise 5: Conditionals

Write a program to prompt for two numbers, and print whichever one is greater.

Exercise 6: Loops

We can also use the `b`*`xx`* and `j` instructions to write a loop as follows.

```
looptest:
   blt $t0, $t1, endloop
   code that executes each time $t0 ≥ $t1 (body of loop)
   j looptest
endloop:
   code that executes when $t0 is finally less than $t1
```

Use this idea to write a program that prints the numbers from 1 to 10.

If You Finish Early

Try writing either or both of the following programs. Alternatively, go implement a more interesting MIPS program of your own choice.

- Pick a number. Write a program that repeatedly asks the user to guess a number, and reports if the user's guess is too high or too low. Your program should stop with an appropriate message when the user guesses correctly.

- This time you think of a number. Write a program that repeatedly takes a guess at your number, asks if the guess was too high or too low, and then makes a better guess. Your program should stop with an appropriate message when it guesses your number correctly.

If You're Really Clever

Write a program to choose a random number. You might use this to improve your guessing game.