2

MIPS: The Virtual Machine

Objectives

After this lab you will know:

- what synthetic instructions are
- how synthetic instructions expand to sequences of native instructions
- how MIPS addresses the memory

Introduction

MIPS is a 'typical' RISC architecture: it has a simple and regular instruction set, only one memory addressing mode (base plus displacement) and instructions are of fixed size (32 bit).

Surprisingly, it is not very simple to write assembly programs for MIPS (and for any RISC machine for that matter). The reason is that programming is not supposed to be done in assembly language. The instruction sets of RISC machines have been designed such that:

- they are good targets for compilers (thus simplifying the job of compiler writers)
- they allow a very efficient implementation (pipelining)

Pipelining means that several instructions are present in the CPU at any time, in various stages of execution. Because of this situation it is possible that some sequences of instructions just don't work the way they are listed on paper. For instance

Ex 1:

```
lw $t0, 0($gp)          # $t0 <- M[$gp + 0]
add $t2, $t2, $t0       # $t2 <- $t2 + $t0
```

loads a word from memory in register **`$t0`** and then adds it, in the next instruction, to register **`$t2`**. The problem here is that, by the time the `add` instruction is ready to use register **`$t0`**, the value of **`$t0`** has not changed yet. This is not to say that the load instruction (`lw`) has not completed. More precisely, the clock cycle when `lw` accesses the memory is the same clock cycle when `add` is doing the addition. If we let the two instruction to proceed, then add will use the value in **`$t0`** that had been stored there before `lw`.

The solution to this problem is to do nothing for one clock cycle thus allowing data from memory to become available. Then special hardware makes this data available to the `add` instruction, without waiting for the load to terminate. In reality, the proper hand-written sequence of code should be:

Ex 2:

```
lw $t0, 0($gp)          # $t0 <- M[$gp + 0]
nop                     # stall
add $t2, $t2, $t0       # $t2 <- $t2 + $t0
```

This load is said to be a *delayed load*. Please note that there is no `nop` instruction in the native MIPS instruction set. However, an instruction like `add $0, $0, $0` does exactly the same thing, i.e. nothing.

Something similar happens with branches, where we fetch a new instruction (literally, the one after the branch in the code) before we know whether the branch is taken or not. To avoid executing an instruction that should not be executed, we may have to insert a `nop` after the branch in the source code. The branch is said to be a *delayed branch*.

Doing assembly programming in these conditions may be very frustrating. Therefore, MIPS has decided to hide these complexities from the programmer. Their assembler presents a *virtual computer* that has no delayed loads or branches and has a richer instruction set than the underlying hardware.

The SPIM simulator simulates this MIPS virtual machine (this is the default). It can also simulate the actual hardware when the `-bare` command-line option is used.

Instructions in the instruction set for the virtual machine, which are not in the instruction set of the bare machine, are said to be *synthetic instructions*. Their sole purpose is to simplify programming. There is no binary representation for them and, as a matter of fact, the assembler either replaces them with equivalent native instructions or with a sequence of instructions from the native instruction set.

Laboratory 2: Prelab

Date ______________ Section ______________

Name ______________

Introduction

This exercise will ask you to get familiar with synthetic instructions in the Instruction Set of the virtual MIPS machine. There are many synthetic instructions but we only explore some of them during this lab. We will see more of them as we continue exploring the Instruction Set in subsequent lab sessions.

Keep in mind that MIPS is a register-register (or Load/Store), three address machine.

- register-register means that all operations are performed in registers. Access to memory is provided only by load and store instructions. There is no arithmetic or logic instruction that has part of operands in registers and part in memory. Operands for such instructions are always in registers.
- three-address means that all arithmetic and logic instructions have three operands, one destination register which is always listed immediately after the instruction name, and two source registers. `add $t0, $t1, $t2` adds source registers **`$t1`** and **`$t2`** and stores the result in the destination register **`$t0`**.

Step 1

Using a text editor, enter the program P.2.

P.1:

```
            .data 0x10010000
var1:       .word 0x55              # var1 is a word (32 bit) with the ..
                                    # initial value 0x55
var2:       .word 0xaa

            .text
            .globl main
main:       addu $s0, $ra, $0# save $31 in $16

            li $t0, X
            move $t1, $t0
            la $t2, var2
            lw $t3, var2
            sw $t2, var1

# restore now the return address in $ra and return from main
            addu $ra, $0, $s0       # return address back in $31
            jr $ra                  # return from main
```

Before you continue, make sure you have entered the last digit of your SSN instead of X in the instruction `li $t0, X`

Save the file under the name *lab2.1.asm*.

Step 2

Start the simulator, load *lab2.1.asm*, and then step through it. Identify synthetic instructions and fill out the following table. Remember that synthetic instructions are those that are in the instruction set of the virtual machine but not in the native instruction set (i.e. not in the instruction set for the bare machine). You can recognize them because they are different in memory from the source file. Sometimes there is a one to one replacement (a synthetic instruction is replaced by a native instruction), other times several native instructions are used to replace a synthetic instruction. The fact that a register name (like **`$t0`**) is replaced by a register number (**`$8`**) does not denote a synthetic instruction.

Address	Synthetic Instruction	Native Instruction(s)	Effect

In the 'Effect' section of the table indicate what is the effect of each native instruction. Be very specific, do not just give a general description of what the instruction does. You may follow the example below

Ex 1:

```
add $t0, $t1, $t2        # $t0 <- $t1 + $t2
```
■

Step 3

The instruction `lui` is used to load a 32 bit constant in a register. Since all instructions are the same size (32 bit wide) there is no way an instruction could initialize a register with a 32 bit *immediate* value[1]. Therefore a mechanism should be in place to allow us to load a 32 bit constant in a register even if this can not be always done in a single step. This mechanism is the `lui` instruction.

1. The size of an immediate value is 16 bits.

The `lui` instruction uses a register (usually **`$1`** which is a register reserved for the assembler) and a constant. In the table below enter the constant you find with the first `lui` in your program, both in decimal and in 16 bit hexadecimal format.

Decimal constant	16 bit hexadecimal

Reload *lab2.1.asm* and set a breakpoint at the address of that `lui` instruction. Run, and the program will stop at that breakpoint. Write down the contents of the register used by the instruction. Execute that instruction (`step 1`) and write down the content of the same register.

Register	Before executing `lui`	After executing `lui`

Q 1:

Assume you want to load the constant `0xabcd0000` in register **`$t0`**. What native instruction(s) should be executed?

Step 4

What is the content of register **`$t2`** after the instruction `la $t2, var2` has been executed? Note that this

synthetic instruction contains a `lui` followed by an immediate instruction.

Q 2:

What does the immediate instruction do in this case?

Q 3:

Assume you want to load the constant `0xabcd00ef` in register **`$t0`**. What native instruction(s) should be executed?

Laboratory 2: Inlab

Date ______________ Section ______________

Name ______________

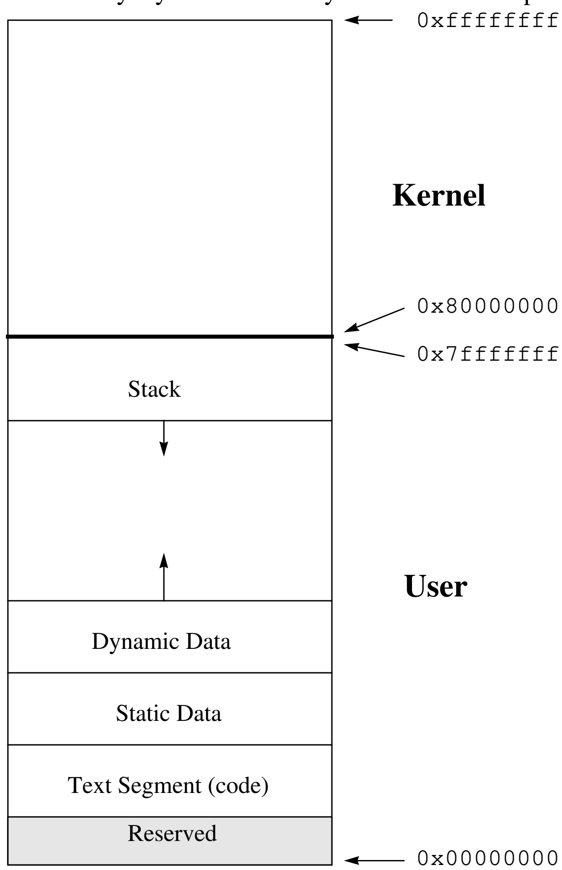
Addressing in MIPS

During this exercise you will use the load word (`lw`) and store word (`sw`) instructions.

Background

The only way CPU can access the memory in MIPS is through load and store instructions. There is only one addressing mode (base+displacement). Having just one addressing mode is part of the RISC philosophy to keep instructions simple thus allowing for a simple control structure and efficient pipelining.

The figure below shows the memory layout for MIPS systems. The **user** space is reserved for user programs.



The **kernel** space can not be directly accesses by user programs, and it is reserved for the use of the operating system[1]. Note that the kernel space is half of the address space (it contains those addresses that have the most significant bit 1). The kernel space may be organized the same way as the user space (with a text segment, data segment, stack), though this is not shown on the figure.

The *text segment* contains the user code the system is executing. The *data segment* has two sections:

- static data, which contains the space for static and global variables; generally speaking this is the storage place for data object whose life is the same as the program's
- dynamic data, where space is allocated for data objects at run time (typically using malloc)

The *stack segment* grows towards small addresses and is used for the call/return mechanism as well as to hold local variables (those defined inside a block and which are not declared to be static).

In base+displacement addressing, a register (the base) is used together with an offset (displacement) to create the address used to accress the memory. The displacement is an immediate value supplied in the instruction.

Ex 1:

```
lw $t0, 4($t2)     # $t0 <- M[$t2 + 4]
```

reads a word from the memory address formed by adding 4 to the content of register **`$t2`**. The word read from that memory location is stored in register **`$t0`**. ■

To access the memory two steps are necessary if the base address is not in a register already:

- load the base address in the base register: in general this is done using the `lui` instruction since the address is 32 bit wide and all other instructions that can load an immediate value do only load a 16 bit constant
- access the memory using the base register and the proper displacement.

Ex 2:

```
lui $1, 0x1001     # $1 <- 0x10010000
lw $8, 4($1)       # $8 <- M[0x10010004]
```
■

To access data on the stack we use register **`$sp`** (the stack pointer). Since this register is a reserved register, all stack accesses will consist of only one instruction, as the stack pointer has already been set to the proper value when the program was loaded. In this context 'reserved register' means that the MIPS register usage convention indicates that the register **`$29`** (**`$sp`**) be used as a stack pointer. But there is no mechanism to restrict its usage for a different purpose.

To access data in the static data section two different possibilities are available.

- data declared within a '.data' section of the program will be accessed using as base an address within the data segment (most likely the start address of the data segment). There are two instructions the assembler generates for each load/store instruction.
- data that is declared using the '.extern' assembler directive will be stored in a special area within the data segment and will be accessed using the reserved **`$gp`** (global pointer) register. Therefore each

1. Laboratory 7 presents the exception mechanism in MIPS and how the kernel space is used.

access to such data will be only one instruction as the **`$gp`** register has been set to the proper value when the program was loaded.

Step 1

Ask your lab instructor to provide you with four integer values you will use when creating the program *lab2.2.asm*

Variable name	Initial value
var1	
var2	
var3	
var4	

Create the program as follows

- reserve space in memory for four variables called *var1* through *var4* of size word. The initial values of these variables will be those provided by your lab instructor
- also reserve space in memory for two variables called *first* and *last* of size byte. The initial value of *first* should be the first letter of your first name and the initial value of *last* should be the first letter of your last name
- the program swaps the values of variables in memory: the new value of *var1* will be the initial value of *var4*, the new value of *var2* will be the initial value of *var3*, *var3* will get the initial value of *var2*, and finally *var4* will get the initial value of *var1*. *first* and *last* will be left unchanged
- use registers **`$t0`** to **`$t8`** if you need to
- use the extended instruction set
- each line in the program has a comment indicating what the instruction does

Step 2

Find the displacement (offset) of each variable in your program from the beginning of the data segment. The displacement will be the distance in bytes between the beginning of the data segment and the place where the variable is stored. Use the `print` command to see what is stored in memory in the data segment.

Variable	Displacement
var1	
var2	
var3	
var4	
first	
last	

Step 3

Run the program and make sure it works properly.

Q 1:

What is the number of instructions executed? Count only instructions between the label 'main' and the last instruction executed from your program.

Instruction Count =

Step 4

Start the simulator using the `-bare` command line option. This will make SPIM simulate a bare MIPS processor.

Load *lab2.2.asm*. What is the reason you get error messages?

Laboratory 2: Postlab

Date ________________ Section ________________

Name ________________________________

Addressing in MIPS (continued)

In this exercise you will continue exploring addressing in MIPS. You are also required to detail the memory model SPIM implements.

Step 1

As you could see during the inlab exercise, one can not run on the bare machine code that uses the extended instruction set.

Based on *lab2.2.asm* create a new program that will do the same thing but will be able to run on the bare machine. Save this as *lab2.3.asm*. Optimize your code as much as possible.

Q 1:

What is the number of instructions executed? Count only instructions between the label 'main' and the last instruction executed from your program.

Instruction Count =

Step 2

Create the program *lab2.4.asm* as follows:

- reserve space in memory for two variables called *var1* and *var2* of size word. The initial values of these variables will be the first two digits of your SSN for *var1* and the next two digits of your SSN for *var2*
- also reserve space in memory for two variables called *ext1* and *ext2* of size word. Use the '.extern' declaration for these two variables. The assembler will reserve space for them in the data segment that can be accessed using the `$gp` register
- the program copies the values of *var1* and *var2* in *ext2* and *ext1* respectively
- use registers `$t0` to `$t8` if you need to
- use the extended instruction set
- each line in the program has a comment indicating what the instruction does

Q 2:

What are the displacements of *ext1* and *ext2* from the global pointer (**`$gp`**) value?

Variable	Displacement (decimal)	Displacement (hexadecimal)

Q 3:

What exactly are the addresses where variables are stored in memory?

Variable	Address (hexadecimal)
var1	
var2	
ext1	
ext2	

Q 4:

How many native instructions are needed for each of the following memory accesses?

Memory Access	Native Instructions
`lw $t0, var1`	
`sw $t0, ext1`	

Step 3

Return to your lab instructor copies of *lab2.3.asm* and *lab2.4.asm* together with this postlab description. Ask your lab instructor whether copies of programs must be on paper (hardcopy), e-mail or both.