Lab 2 (All Sections) Prelab: Pseudo Instructions and Memory Access

Name:

Joseph DeLeonardis

Sign the following statement:

On my honor, as an Aggie, I have neither given nor received unauthorized aid on this academic work

Joseph DeLeonardis

1 Objective

The main objectives of this lab are to:

1. Learn several pseudo-instructions which are available in the MIPS architecture;
2. Gain experience with memory access instructions.

Before proceeding with this lab, you should be familiar with basic MIPS instructions.

2 Pseudo-Instructions

MIPS has a simple and regular instruction set, which is typical for RISC architectures. In particular, it contains only one memory addressing mode (base + displacement) and all instructions have a fixed size (32 bit).

For programmer convenience, the MIPS assembly language includes several *pseudo-instructions*. These are not "real" instructions (that is a MIPS processor will not implement them). The assembler translates these instructions into equivalent real machine instructions.

Consider the following pseudo-instruction:

```
move $t0, $t1   #register $t1's value is copied into register $t0
```

The assembler will translate this pseudo instruction into the equivalent instruction:

```
add $t0, $0, $t1   #register $t1's value is added to $zero and stored in register $t0
                   #Remember, $zero always contains the value 0
```

3 Accessing Memory

MIPS is a load-store architecture, which means that only the **load** and **store** instructions access memory. Computation instructions only operate on the values in registers. In addition, MIPS is limited to accessing memory in a single mode (base + displacement), written as `c(rx)`. The address is computed by adding the **constant displacement** `c` with the **base** register, `rx`. For example

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

Most load and store instructions only operate on aligned data. That means that a word (4 bytes) must be stored at addresses which are multiples of 4 and a half-word (2 bytes) must be stored at even address. MIPS does contain some instructions which can manipulate unaligned data.

Figure 1 shows the memory layout of the MIPS processor. User space is reserved for user programs and the Kernel space is reserved for use by the operating system. User programs cannot directly use the Kernel space. The text segment contains the user code (executable binary) 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).

Static data can be accessed through two different methods:

- Data declared within a **.data** section of the program will be accessed using an address within the data segment as base (most likely the start address of the data segment). There are two instructions the assembler generates for each load/store instruction.

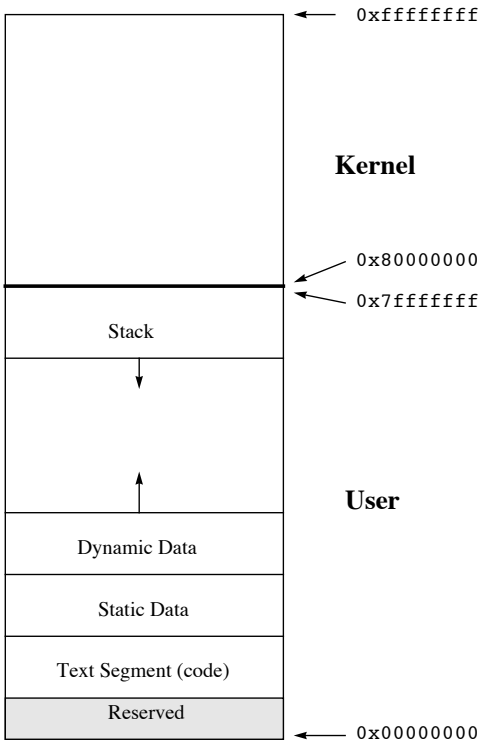

Fig. 1: MIPS Memory Layout

- 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 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.

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).

To access data on the stack we use register $sp (the stack pointer). Stack pointer register is a *reserved register*, that means its 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 the memory we need to do the following:

- Load the base address in the base register: in general this is done using the lui instruction (load upper immediate).

- Access the memory using the base register and by adding proper displacement.

4 Questions

1. State whether the following translations for the given pseudo-instructions are valid or not?

Pseudo-Instruction	Translation	Valid/Invalid
move $rt, $rs <**$rt := $rs**>	addi $rt, $rs, $0	Valid
li $rs, small <**$rs[15..0] := small**>	addi $rs, $0, small	Valid
li $rs, big <**$rs := big**>	lui $rs, upper(big) ori $rs, $rs, lower(big)	Valid

Note: **small** represents a 16-bit number and **big** represents a 32-bit number. The function **upper(big)** represents the upper 16-bits of a 32-bit value and **lower(big)** represents the lower 16-bits of a 32-bit value.

2. What is the content of register $t2 after the instruction **la $t2, var** has been executed? Note that this pseudo-instruction contains a **lui** followed by an immediate instruction. Assume **var** is stored at address **0x00001000** and it contains a value of **10**? **Also write the native MIPS instructions to which this instruction is translated.**

la means load address so the address of var 0x00001000 will be loaded into $t2

lui $t2, 0x0000 Loads the upper 16 bits
ori $t2, $t2, 0x1000 Loads the lower 16 bits

3. Write a sequence of instructions for loading a value of 0xabcd0080 into register $1.

lui $1, 0x2BCD Loads the upper 16 bits
ori, $1, $1, 0x0080 Loads the lower 16 bits

4. Write a MIPS assembly language implementation of the following C language instruction:

```
A[2*i]=A[2*k+j];
```

where A is an array of integers that starts at address `Astart` (32-bit value), and variables i, j, and k are stored in registers `$s0,$s1`, and `$s3`, respectively. You may use registers `$t0,$t1`, and `$t2` for temporary values, if needed. Each element of A is four bytes long.

```
la $t0, Astart          Loads Astart into register $t0
sll $s0, $s0, 3         Since shift left logical means multiply by 2, we shift $s0 s times to
                        multiply 8. Each element is 4 bytes long. Store it back in $s0 again
sll $s3, $s3, 1         Shift 1 to get 2*k
add $t1, $s3, $s1       2*k+j
add $t2, $t0, $t1       A[2*k+j]
add $s3, $t0, $s0       A[2*i]
sw $s3, 0($t2)          Saves register $s3 into $t2 ( 0 is just an offset for the address)
```

5. Explain how the following program can be used to determine whether a computer is big-endian or little-endian

```
li $t0, 0xABCD9876
sw $t0, 100($0)
lb $s5, 101($0)
```

If register $s5 is a number then it is little-endian.
If register $s5 is a letter then it is big-endian.