Lab 3 (All Sections) Prelab: MIPS-Control Instructions

Name:

Chenjie Luo

Sign the following statement:

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

Chenjie Luo

1 Objective

The main objective of this lab is to understand control structures in MIPS. Before proceeding with this lab, you should be familiar with the branch and jump instructions in MIPS (Chapter 2 of the textbook).

2 Flow Control

In previous labs, your programs have mostly been purely sequential (each instruction is executed in a sequential order); most programs, however, require flow control. MIPS contains control instructions, which allow programs to execute in a non-sequential manner. In a high-level language, such as C, control structures have the form *if-then-else*, *goto*, *for*, or *do-while*. In assembly language, the flow control statements are generally simpler and

are of the form *branch*, *jump*, *call*, and *return*. Each has the ability to alter the flow of the program.

The program counter (PC) register contains the address of current instruction to be executed. In sequential flow, the PC is incremented to the next instruction (by 4) each time an instruction starts its execution. The PC register itself cannot be accessed directly by the program, but MIPS provides other means to alter its value, and thus change the location where the program will continue its execution.

There are 2 types of control instructions:

Conditional branches. These control instructions may or may not change the PC depending on a certain condition. These instructions are generally referred to as *branch* instructions, because the program flow will follow one of two possible paths. A branch instruction must both specify the condition on which it will branch, as well as the target address if the conditions are satisfied. If the conditions are not satisfied, the program continues execution at the next instruction.

Unconditional jumps. These control instructions always change the PC without any conditions. They are generally referred to as *jump* instructions because the program flow will jump to a new location. A jump instruction must always specify the target address of the jump.

Branch instructions are typically used to form loops. Since most loops tend to be small in size, the target address tends to be close to the branch instruction. When the target instruction is close enough to the current instruction, then the target can be specified relative to the current instruction with an offset. A branch instruction could be described as the equivalent of "If `$t0` is not `0`, then skip the next `5` instructions." In addition to the offset being positive (forward branch), the branch instruction also allows a negative offset (backwards branch). Such a branch would be useful in an instruction that does the equivalent of "If the loop is not done, go back `100` instructions and execute it again." This is called *PC-relative addressing*.

Aside from smaller instruction encoding, PC-relative address also allows a program to be comprised of "Position Independent Code (PIC)." This means that a program will execute in the same manner no matter where it is located in memory. A forward branch of 5 instructions will work the same way whether it is at address `100`, or at address `345,236`.

Branches, however, are limited in the distance away from the current PC that the execution may shift. Jumps, on the other hand, may reach instructions which are much further away than those that can be reached by a branch instruction. Since they execute irrespective of any conditions, there is no need to use space in the instruction encoding to store a condition like there is in the branch instructions. This leaves more space to encode the target address. The jump instruction has a 26-bit target address field. In cases where this

is not big enough, the jump instruction can also jump to an address specified in a register instead of an immediate value.

There is also a special kind of jump instruction which will also save the address of where it came from. This is useful in subroutines which will return back to their called after their execution has completed. You have already seen behavior like this in the *syscall* routines you have used in previous labs.

Table 1 summarizes the MIPS branch and jump instructions:

Tab. 1: MIPS Jump and Branch Instructions

Instruction	Effect
`beq Rs, Rt, label`	$if(Rs == Rt), PC \leftarrow label$
`bne Rs, Rt, label`	$if(Rs \neq Rt), PC \leftarrow label$
`bltz Rs, Rt, label`	$if(Rs < Rt), PC \leftarrow label$
`blez Rs, Rt, label`	$if(Rs \leq Rt), PC \leftarrow label$
`bgtz Rs, Rt, label`	$if(Rs > Rt), PC \leftarrow label$
`bgez Rs, Rt, label`	$if(Rs \geq Rt), PC \leftarrow label$
`j jlabel`	$PC \leftarrow jlabel$
`jr Rs`	$PC \leftarrow Rs$
`jal jlabel`	$\$ra \leftarrow PC + 4, PC \leftarrow jlabel$
`jalr Rs`	$\$ra \leftarrow PC + 4, PC \leftarrow Rs$

In table 1, *label* can either be a symbolic name (a label in the program) or a 16-bit offset. In the latter case, $PC \leftarrow label$ represents PC-relative addressing. The field *jlabel* can also represent a symbolic name or a 26-bit address field. In the latter case, the $PC \leftarrow jlabel$ represents the lower 28-bits of the PC being replaced with the 26-bits of the jlabel, and the lowest 2 bits being set to 0s.

3 Questions

1. (a) What is the range of addresses for a conditional branch instruction in MIPS with respect to `X`, where `X` contains the address of the branch instruction? Assume the current `PC` value is $\texttt{0x10000000}_{hex}$.

   0x10000000 - 2^15 < address < 0x10000000 + 2^15 - 1

(b) What is the range of addresses for jump (`j`) and the jump and link (`jal`) instructions in MIPS. Assume the current PC value is $\texttt{0x10000000}_{hex}$.

0x10000000 <= Address <= 0x10000000 + 2^26 -1

2. Consider the following C code and its MIPS translation, assuming x is in $s3 and y is in $s4.

```
if(x==y)
x++;
else
y++;
```

```
1  bne $s3, $s4, Else
2  addi $s3, $s3, 1
3  j Exit
4  Else: addi $s4, $s4, 1
5  Exit: noop
```

Write MIPS code for the following C codes:

(a)
```
if(x!=y && x==0)
x++;
else
y++;
```

```
    beq $s3, $s4, go
    addi $s3, $s3, 1
go: addi $s4, $s4, 1
```

(b)

```
while(x<10){
  y=y+10;
  x++;
}
```

```
addi $t0, $t0, 10
jump: bge $s3,$t0, go
      addi $s4, $s4, 10
      addi $s3, $s3, 1
      j jump
go:
```

(c)

```
switch(x){
  case 0: y++; break;
  case 1: y--; break;
  default: break;
}
```

Hint: Consider the switch statement to be a series of *if-then-else* statements.

```
beqz $s3, case_0
addi $t0, $t0, 1
beq $t0, $s3, case_1
j go
case_0: addi $s4, $s4, 1
      j go
case_1: addi $s4, $s4, -1
      j go
go:
```

3. Convert this MIPS machine code into MAL (MIPS Assembly Language) instructions. Your final answers should use the register names, not the numbers (i.e. `$t0`, not `$8`). Also, values which represent absolute addresses (if any) should be converted into the full 32 bit address.

ADDRESS:	Instructions:
0x00400000	001000 11101 11101 11111 11111 111100
0x00400004	101011 11101 11111 00000 00000 000000
0x00400008	001000 00100 00100 00000 00000 000010
0x0040000C	101011 11100 00100 10000 00000 000000
0x00400010	001000 00000 00101 00000 00000 000011
0x00400014	101011 11100 00101 10000 00000 000100
0x00400018	000011 00000 10000 00000 00000 001011

```
addi $sp, $sp, 65532
sw $ra, 0($sp)
addi $a0, $a0, 2
sw $a0, -32768($gp)
addi $a1, $0, 3
sw $a1, -32765($gp)
jal 0x0040002c
```

4. Consider the following fragment of MIPS assembly code. Assume that before the invocation of the program the register `$a0` contains a positive value.

```
1          START:  addi $t0, $a0, 0
2          addi $t1, $zero, 1
3          addi $t3, $zero, 0
4          LOOP:   beq $t0, $zero, FINISH
5          and $t2, $t0, $t1
6          srl $t0, $t0, 1
7          xor $t3, $t3, $t2
8          j LOOP
9          FINISH: addi $v0, $t3, 0
```

What will register `$v0` contain after the execution of the program?

If the value of $a0 is odd power of 2, $v0 will contain 0.
If the value of $a0 is even power of 2, $v0 will contain 1.