3

Control Structures in MIPS

Objectives

After completing this lab you will:

- know how conditional and unconditional branches work in MIPS
- better understand the advantages of having fixed size instructions
- be able to use conditional and unconditional branches in your programs

Introduction

Except for some very simple programs all others use instructions that control the program flow. In a high level language they may be called *if* (with its associated *then* and *else*), *goto* (which fans of structured programming hate), *for*, *while*, *do*. In assembly language they may be called *branch*, *jump*, *call*, *return*. They all have the fundamental property that they change the order in which instructions are executed. They allow the programmer to specify different sequences of instructions to be executed based on the input and/or results calculated earlier in the program.

The pointer to the present instruction is the Program Counter (PC). PC is normally updated as to point to the next sequential instruction to execute. Control instructions that may change the PC based on testing some condition are called conditional (or more often *branches*). Control instructions that always change the PC are called unconditional (or *jumps*).

Any branch must indicate the condition which is tested as to decide whether the branch is *taken* (change the PC) or not (continue with the next sequential instruction). Any branch instruction must also indicate where to start fetching instructions in case the branch is taken. In other words the instruction must also provide the *target*).

Since branches are usually used to implement loops, and because loops tend to have a small number of instructions, the target is in most cases close to the current instruction. In other words, the difference between the target address and the current PC is small in absolute value. There is no need to include the absolute value of the target in the instruction[1]. What the instruction can specify instead, is the distance in bytes from the current instruction to the target. This distance (offset) can be positive (for forward branches) or negative (backward branch). If the branch is taken, then the offset is added to the current PC to obtain the address from which the next instruction will be fetched. This is called *PC-relative addressing*.

1. In particular this would not be possible in MIPS (nor in other RISC architectures) since all instructions have the same size (32 bits) as an address.

Aside from the fact that PC-relative addressing allows for a better instruction encoding, it has also the property that it allows the code to run independent of where it is loaded in memory. This *position-independence* can eliminate some work when the program is loaded in memory.

Jumps, on the other hand, may reach instructions that are far away from the currently executing instruction. Since a jump is an unconditional change of PC, there is no need to specify anything else in the instruction but the opcode and the address of the target (no, jumps are **not** PC-relative). Of course an instruction that is 32 bit wide can not hold an address that is 32 bit itself. The field in the jump instruction that specifies the address is 26 bit wide.

The architecture also allows for the function call/return mechanism through a set of dedicated jump instructions.

A brief summary of branches and jumps in the native MIPS instruction set is given below.

Instruction	Effect
`beq Rs, Rt, label`	`if (Rs == Rt) PC ←label`
`bne Rs, Rt, label`	`if (Rs != Rt) PC ←label`
`bltz Rs, label`	`if (Rs < 0) PC ←label`
`blez Rs, label`	`if (Rs <= 0) PC ←label`
`bgtz Rs, label`	`if (Rs > 0) PC ←label`
`bgez Rs, label`	`if (Rs >= 0) PC ←label`
`j jlabel`	`PC ←jlabel`
`jr Rs`	`PC ←Rs`
`jal jlabel`	`$ra ←PC+4, PC ←jlabel`
`jalr Rs`	`$ra ←PC+4, PC ←Rs`

In this table `label` and `jlabel` mean:

- a symbolic name (i.e. a label in the program) in the user's program
- a 16 bit offset for a branch in the binary code. In this case `PC ← label` stands for PC-relative addressing
- a 26 bit address for a jump in the binary code. In this case `PC ← jlabel` means that this 26 bit address replaces the PC during the execution of that jump instruction[1].

This laboratory is focused on branches and the regular jump (`j`). The other jumps (`jr, jal, jalr`) will be studied in a different lab session.

1. In reality the process is a little bit more complicated and will be described in detail in the Inlab section of Lab #3.

Laboratory 3: Prelab

Date ____________________ Section ____________________

Name ____

Introduction

Let's have a closer look at the branch instructions and how they work. In particular we want to know in detail how

- branches are encoded
- the target address is calculated

Branch encoding

The size of the offset field is 16 bits (the least significant bits in the instruction). Since instructions are of fixed size (four bytes each), specifying the offset in bytes would be wasteful: the offset would always be a multiple of four number. Instead, the offset indicates the number of words (1 word = 4 bytes) to the target.

The offset may be positive or negative. Negative numbers use 2's complement representation.

Caveat: to be efficient, any real implementation of MIPS is pipelined. This means a new instruction is fetched every clock cycle. Therefore the PC must point to a new instruction every clock cycle, a few clock cycles before an instruction is complete. This means that PC is updated (PC $\leftarrow$ PC+4) very early in an instruction to point to the next one. Hence, when it comes to branches, the offset is actually relative to the next instruction (PC+4) as opposed to the current one (PC). The SPIM simulator hides this complexity from the user. When you inspect the memory you will see the offsets relative to the current instruction.

Computing the target

The following happen inside the CPU when the target of a branch is calculated:

- the offset (16 bits) is left-shifted with two bits (which is equivalent to multiplying by 4) as to express the distance to the target in bytes.
- the offset (18 bits now) is then sign-extended to 32 bits; if the offset is positive (the most significant bit of the offset is 0), then 0's will be padded on the most significant positions up to 32 bits, otherwise, if the offset is negative (the most significant bit of the offset is 1), 1's will be padded. Sign-extension does not change the number.
- add the offset (32 bits) to the PC (32 bits) and obtain the target address.

The three steps described above are logical steps. It does **not** take threee clock cycles to calculate the target address.

Implementing if-then-else

The *if-then-else* is one of the fundamental programming constructs. The following example indicates how such a construct would be translated in assembly language. Note that *then* is not a keyword in C. However, the construct presented below is of the type ‘if-then-else’, where the block of code #1 corresponds to the ‘then’ part.

Ex 1:

```
if (var1 == var2) {
      ....        /* block of code #1 */
}
else {
      ....        /* block of code #2 */
}
```

Let’s assume that the values of variables *var1* and *var2* are in registers **$t0** and **$t1**. Then, this piece of C code would be translated as:

```
            bne $t0, $t1, Else      # go to Else if $t0 != $t1
            ....                    # code for block #1
            beq $0, $0, Exit        # go to Exit (skip code for block #2)
Else:
            ....                    # code for block #2
Exit:                               # exit the if-else
```

The instruction `beq $0, $0, Exit` is equivalent to an unconditional jump since the test always succeeds (you could substitute any register name in place of **$0** but then it wouldn’t be that obvious what you want to do). The same effect would be obtained by using `j Exit` (unconditionally jump to label ‘Exit’). ■

Step 1

Create a program called *lab3.1.asm* as follows:

- reserve space in memory for three variables called *var1* through *var3* of size word. The initial values of *var1* and *var2* will be the first and the second digit of your SSN respectively. *var3* will be initialized to minus the number of this year.
- the program will implement the piece of C code described below. `tmp` is a local variable for which you may use any of the registers **$t0** through **$t9**.
- use only branches found in the native instruction set.

```
if (var1 == var2) {
      var1 = var3;      /* change the values of var1 and .. */
      var2 = var3;      /* var2 to the value of var3 */
}
else {                  /* execute when var1 != var2 */
      tmp = var1;       /* swap the values of var1 and var2 */
      var1 = var2;
      var2 = tmp;
}
```

Q 1:

What values do you expect for *var1* and *var2* after executing this code?

Variable	Expected value
var1	
var2	

Step 2

Load *lab3.1.asm*. Fill out the 'Before run' section of the table below. Use `print` to look in the memory for the values of *var1* and *var2*.

Variable	Before run	After run
var1		
var2		

Run the program and fill out the 'After run' section of the same table. Compare the expected values for *var1* and *var2* with the values you get after running the program.

Step 3

For each branch in your program calculate by hand the target address. Use `step` to find out what is the current PC and the offset in the branch instructions.

Instruction	Offset	Shifted and sign extended offset	Current PC	Target address

Step 4

In the C code presented at Step 1, replace the equality test (==) by greater than (>). Create a program called *lab3.2.asm* (use *lab3.1.asm* as a skeleton) which implements the modified C code. As you can easily see there is no native branch that uses the greater than condition. The reason is that such a comparison would lenghten the time it takes a branch to decide whether it is taken or not.

To implement such a test, two instructions are required, a set followed by a branch.

Ex 2:

```
slt $t0, $t1, $t2       # if ($t1 < $t2) $t0 = 1; else $t0 = 0;
bgtz $t0, label         # jump at label if $t0 is greater than zero
```

■

Q 2:

What values do you expect for *var1* and *var2* after executing this code in *lab3.2.asm*?

Variable	Expected value
var1	
var2	

Step 5

Load *lab3.2.asm*. Fill out the 'Before run' section of the table below: Use `print` to look in the memory for

Variable	Before run	After run
var1		
var2		

the values of *var1* and *var2*.

Run the program and fill out the 'After run' section of the same table. Compare the expected values for *var1* and *var2* with the values you get after running the program.

Q 3:

What sequence of instructions would you use to implement the test

```
if (var1 >= var2)
```

Assume that *var1* and *var2* are in registers **`$t0`** and **`$t1`** respectively.

Laboratory 3: Inlab

Date ______________________ Section ______________________

Name ____

Branches (cont'd)

We continue exploring the branches in MIPS and we introduce the jump (`j`) instruction.

Q 1:

How far away can a branch go from the current PC? Use the value furnished by your lab instructor for the current PC.

Current PC	Most positive offset	Target address for a forward branch with the most positive offset	Most negative offset	Target address for a backward branch with the most negative offset

Implementing a for loop

The following is a typical `for` loop in C:

Ex 1:

```
for (i=begin; i<limit; i++) {
      ....                    /* for body */
}
```

The for loop is equivalent to

```
i = begin;                    /* initialize the loop index */
while (i < limit) {           /* test for termination */
      ....                    /* for body */
      i++;                    /* prepare i for next iteration */
}
```

Assuming that the initial value (`begin`) for the loop index is in register **`$a0`**, the `limit` is in **`$a1`**, and that the loop index `i` is in register **`$t0`**, then the for loop could be implemented as:

```
            move $t0, $a0         # i is in $t0
Loop:       ble $a1, $t0, Exit    # exit if limit <= i
```

```
            ....                    # body of the for loop
            addi $t0, $t0, 1        # i = i+1
            j Loop
Exit:       ....                    # this is outside the loop
```
■

Step 1

Create a program called *lab3.3.asm* based on this description:

- reserve space in memory for a variable called *var1* of size word. The initial value of *var1* will be the first digit of your SSN.
- the program will implement the piece of C code described below.

```
for (i=var1; i<100; i++) {
      var1 = var1+1;
}
```

Q 2:

What value would *var1* have after executing the for loop?

Variable	Initial value	Expected value
var1		

Step 2

Load *lab3.3.asm*. Fill out the 'Before run' section of the table below. Use `print` to look in the memory for

Variable	Before run	After run
var1		

the value of *var1*.

Run the program and fill out the 'After run' section of the same table. Compare the expected values for *var1* with the value you obtain after running the program.

Step 3

The implementation of the for loop presented in Ex. 1 uses a branch from the extended instructions set. Step through the program to find out how the assembler replaces the synthetic instruction (`ble`) with native instructions.

Synthetic Instruction	Native Instructions	Effect

Synthetic Instruction	Native Instructions	Effect

Step 4

The implementation of the for loop presented in Ex. 1 uses a jump instruction. The 26 least significant bits in the instruction represent an instruction address. Since all instruction addresses are multiples of four, representing the address as such would be wasteful (the least significant two bits would always be zero). Rather,. the address divided by four is stored in the 26 bit field. The following example should clarify this.

Ex 2:

```
label1:.    ...         # some interesting stuff here
            ....        # in between code
            j label1    # jump to the instruction labeled 'label1'
```

Assume we know that the address of the instruction labeled 'label1' is `0x409abc`. Then the 26 least significant bits in the jump instruction are `0x109abc` since this is the address `0x409abc` divided by four. To convince yourself this is true, take the number `0x409abc` and shift it right by two positions (thus discarding the least significant two bits) which is equivalent to an integer division by four. This is exactly what the assembler does when generating code for the jump. ■

When the jump instruction is executed, the following happen:

- the address (26 bits) is left-shifted with two bits (which is equivalent to multiplying by 4) as to find the true address.
- *replace* the least significant 28 bits of the PC with the address.

The reason the 28 bit address replaces the least significant 28 bits in the PC is speed. The designer has chosen to avoid an addition (if the jump were PC-relative) thus saving one clock cycle in the execution.

Caveat: the 4 most significant bits in the PC remain unchanged in the process described above. The linker and the loader must make sure that the program is placed in the memory in such a way as to have the same most significant 4 bits for the address of the jump instruction and for the target.

Q 3:

Give an example of what can happen during execution if the condition stated in the above caveat is not respected.

Step 5

Step through the program to find the jump instruction. Fill out the next table

Instruction	Address field in instruction (26 bits)	Shifted left by two bitst	Target address

Q 4:

What is the largest address a jump can reach?

Laboratory 3: Postlab

Date ______________________ Section ______________________________

Name ____

Step 1

Create a program called *lab3.4.asm* as follows:

- reserve space in memory for an array of words of size 10. Use the '.space' directive. The array is called *my_array*.
- the program will implement the piece of C code described below. The value of `initial_value` is the first digit of your SSN. `i` and `j` will be in one of the registers **`$t0`** to **`$t9`**.

```
j = initial_value;
for (i=0; i<10, i++) {
      my_array[i] = j;
      j++;
}
```

Run the program and make sure it works. Do not forget the comments at each line of code indicating what they do.

Hint: A common mistake here is to forget that sequential word addresses in memory differ by 4 not by one.

Step 2

Most branches have as a target an instruction that is nearby. Occasionally however, a branch may have a target that is very far away, much farther than can be represented using the 16 bit offset. Write a program called *lab3.5.asm* that shows such a situation. The description of the program follows:

- prompts the user to enter two integers; store them in **`$t0`** and **`$t1`**
- if the two integers are equal, then the program branches to a label called 'Far' that is very far away (farther than a 16 bit offset can indicate), prints the message "I'm far away" and terminates.
- if the two integers are different, then the program prints the message "I'm nearby" and terminates.

Q 1:

What is the sequence of instructions the assembler generates to implement this branch?

Synthetic Instruction	Native Instructions	Effect

Step 3

Return to your lab instructor copies of *lab3.4.asm* and *lab3.5.asm* together with this postlab description. Ask your lab instructor whether copies of programs must be on paper (hardcopy), e-mail or both.