4 Flow Control

4.1 Objectives

After completing this Assignment, you will:

- Get familiar with MIPS Jump and Branch instructions
- Learn about pseudo instructions in MIPS
- Learn how to translate high-level flow control constructs (if-then-else, for loop, while loop) to MIPS code

4.2 MIPS Jump and Branch Instructions

Like all processors, MIPS has instructions for implementing unconditional and conditional jumps. The MIPS Jump and Branch instructions are shown in Table 4.1.

Instruction		Meaning	Format			
j	label	jump to label	$op^6 = 2$	imm^{26}		
beq	rs, rt, label	branch if (rs == rt)	$op^6 = 4$	rs^5	rt^5	imm^{16}
bne	rs, rt, label	branch if (rs != rt)	$op^6 = 5$	rs^5	rt^5	imm^{16}
blez	rs, label	branch if (rs<=0)	$op^6 = 6$	rs^5	0	imm^{16}
bgtz	rs, label	branch if (rs > 0)	$op^6 = 7$	rs^5	0	imm^{16}
bltz	rs, label	branch if (rs < 0)	$op^6 = 1$	rs^5	0	imm^{16}
bgez	rs, label	branch if (rs>=0)	$op^6 = 1$	rs^5	1	imm^{16}

Table 4.1: MIPS Jump and Branch Instructions.

For unconditional jump, the instruction `j label` is used where label is the address of the target instruction as shown below:

```
j   label               # jump to label

. . .

label:
```

There are two MIPS conditional branch instructions that branch based on the condition whether two registers are equal or not as follows:

```
beq Rs, Rt, label      # branch to label if (Rs == Rt)
bne Rs, Rt, label      # branch to label if (Rs != Rt)
```

Four additional MIPS instructions are provided based on comparing the content of a register with **0** as follows:

```
bltz Rs, label         # branch to label if (Rs < 0)
bgtz Rs, label         # branch to label if (Rs > 0)
blez Rs, label         # branch to label if (Rs <= 0)
bgez Rs, label         # branch to label if (Rs >= 0)
```

Note that MIPS does not provide the instructions **beqz** and **bnez** as these can be implemented using the **beq** and **bne** instructions with register **$0** used as the second operand.

MIPS also provides four set on less than instructions as follows:

```
slt   rd, rs, rt        # if (rs < rt) rd = 1 else rd = 0
sltu  rd, rs, rt        # unsigned <
slti  rt, rs, im16      # if (rs < im16) rt = 1 else rt = 0
sltiu rt, rs, im16      # unsigned <
```

Note that the instructions **slt** and **slti** are used for signed comparison while instructions **sltu** and **sltiu** are used for unsigned comparison.

For example, assume that **$s0 = 1** and **$s1 = -1 = 0xffffffff**, then the following two instructions produce different results as shown below:

```
slt   $t0, $s0, $s1     # results in $t0 = 0
sltu  $t0, $s0, $s1     # results in $t0 = 1
```

4.3 Pseudo Instructions

Pseudo instructions are instructions introduced by an assembler as if they were real instructions. We have seen an example of a pseudo instruction before, which is the **li** instruction. Pseudo instructions are useful as they facilitate programming in assembly language.

For example, the MIPS processor does not have the following useful conditional branch comparison instructions:

```
blt, bltu      branch if less than          (signed/unsigned)
ble, bleu      branch if less or equal      (signed/unsigned)
bgt, bgtu      branch if greater than       (signed/unsigned)
bge, bgeu      branch if greater or equal   (signed/unsigned)
```

The reason for not implementing these instructions as part of the MIPS instruction set is that they can be easily implemented based on a set of two instructions.

For example, the instruction **`blt $s0, $s1, label`** can be implemented using the following sequence of two instructions:

```
slt $at, $s0, $s1
bne $at, $zero, label
```

Similarly, the instruction **`ble $s2, $s3, label`** can be implemented using the following sequence of two instructions:

```
slt $at, $s3, $s2
beq $at, $zero, label
```

Table 4.2 shows more examples of pseudo instructions. Note that the assembler temporary register **`$at=$1`** is reserved for its own use.

Pseudo-Instructions	Conversion to Real Instructions
`move $s1, $s2`	`addu $s1, $zero, $s2`
`not $s1, $s2`	`nor $s1, $s2, $zero`
`li $s1, 0xabcd`	`ori $s1, $zero, 0xabcd`

`li $s1, 0xabcd1234`	`lui $at, 0xabcd`
	`ori $s1, $at, 0x1234`

`sgt $s1, $s2, $s3`	`slt $s1, $s3, $s2`

`blt $s1, $s2, label`	`slt $at, $s1, $s2`
	`bne $at, $zero, label`

Table 4.2: Examples of pseudo instructions.

4.4 Translating High-Level Flow Control Constructs

We can translate any high-level flow construct into assembly language using the jump, branch and set-less-than instructions. For example, let us consider the following **`if`** statement:

```
if (a == b) c = d + e; else c = d - e;
```

Let us assume that variables **a**, **b**, **c**, **d**, **e** are stored in registers **$s0** thru **$s4** respectively. The following assembly code implements this IF statement:

```
        bne   $s0, $s1, else
        addu  $s2, $s3, $s4
        j     exit
else:   subu  $s2, $s3, $s4
exit:   . . .
```

We can also implement an IF statement with a compound condition involving logical AND operation. For example, let us consider implementing the following IF statement:

```
  if (($s1 > 0) && ($s2 < 0)) {$s3++;}
```

The IF statement is implemented efficiently using the following assembly code which uses the fall through concept which skips the execution of the instruction if the first condition is false otherwise it continues the execution:

```
    blez    $s1, next          # skip if false
    bgez    $s2, next          # skip if false
    addiu   $s3, $s3, 1        # both are true
next:
```

Similarly, we can translate an IF statement with a compound condition involving logical OR operation. For example, let us consider implementing the following IF statement:

```
  if (($sl > $s2) || ($s2 > $s3)) {$s4 = 1;}
```

The IF statement is implemented efficiently using the following assembly code which checks the first condition and if it is true, it skips the second condition:

```
    bgt $s1, $s2, L1           # yes, execute if part
    ble $s2, $s3, next         # no: skip if part
L1: li  $s4, 1                 # set $s4 to 1
next:
```

We can also implement all types of loops. Let us consider implementing the following **for** loop:

```
  for (i=0; i<n; i++) {
    loop body
  }
```

Let us assume that variable **`i`** is stored in register **`$s0`** and **`n`** is stored in register **`$s1`**. Then, the **`for`** loop is implemented using the following assembly code:

```
    li $s0, 0                # i = 0
ForLoop:
    bge $s0, $s1, EndFor
    loop body
    addi $s0, #s0, 1         # i++
    j ForLoop
EndFor:
```

Consider the implementing of the following **`while`** loop:

```
   i=0;
   while (i<n) {
     loop body
     i++;
   }
```

We can note that this **`while`** loop has identical behavior to the **`for`** loop and hence its assembly code will be identical.

Finally, let us consider implementing the following **`do-while`** loop:

```
   i=0;
   do {
     loop body
     i++;
   } while (i<n)
```

The **`do-while`** loop can be translated using the following assembly code:

```
    li $s0, 0                # i = 0
WhileLoop:
    loop body
    addi $s0, $s0, 1         # i++
    blt  $s0, $s1, WhileLoop
```

4.5 Exercise

1. Write a program that asks the user to enter an integer and then displays the number of 1's in the binary representation of that integer. For example, if the user enters **9**, then the program should display **2**.
2. Write a program that asks the user to enter two integers: **n1** and **n2** and prints the sum of all numbers from **n1** to **n2**. For example, if the user enters **n1=3** and **n2=7**, then the program should display the sum as **25**.
3. Write a program that asks the user to enter an integer and then display the hexadecimal representation of that integer.
4. The Fibonacci sequence are the numbers in the following integer sequence: **0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...**

   The first two numbers are **0** and **1** and each subsequent number is the sum of the previous two. Write a program that asks the user to enter a positive integer number **n** and then prints the **n**th number in the Fibonacci sequence. The following algorithm can be used:

```
Input: n positive integer
Output: nth Fibonacci number
    Fib0 = 0  Fib1 = 1
    for (i=2; i <= n; i++) do
        temp = fib0
        fib0 = fib1
        fib1 = temp + fib1
    if  (n > 0)  fib = fib1
    else fib = 0
```

4.6 Bonus Problem

5. One method for computing the greatest common divisor of two positive numbers is the binary gcd method, which uses only subtraction and division by 2. The algorithm of the binary gcd is outlined below:

```
Input: a, b positive integers
Output: g and d such that g is odd and gcd(a, b) = g×2^d
   d = 0
   while (a and b are both even) {
      a = a/2
      b = b/2
      d = d + 1
   }
   while (a != b) {
      if (a is even) a = a/2
      else if (b is even) b = b/2
      else if (a > b) a = (a - b)/2
      else b = (b - a)/2
   }
   g = a
```

Write a program that asks the user to enter two positive numbers ***a*** and ***b*** and outputs the greatest common divisor of the two numbers by implementing the given algorithm. If the user enters **a=48** and **b=18**, your program should output the gcd as **6**.