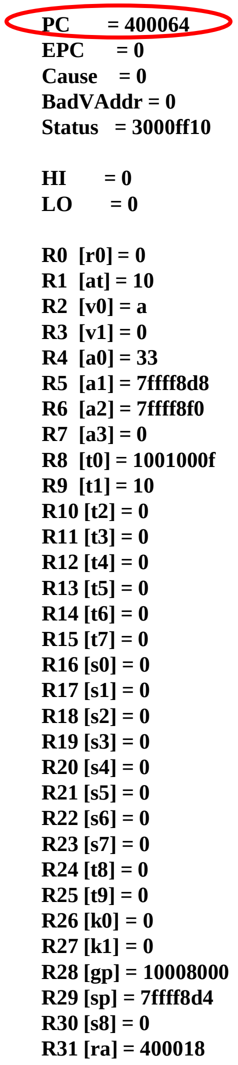
The Program Counter



```
PC       = 400064
EPC      = 0
Cause    = 0
BadVAddr = 0
Status   = 3000ff10

HI       = 0
LO       = 0

R0  [r0] = 0
R1  [at] = 10
R2  [v0] = a
R3  [v1] = 0
R4  [a0] = 33
R5  [a1] = 7ffff8d8
R6  [a2] = 7ffff8f0
R7  [a3] = 0
R8  [t0] = 1001000f
R9  [t1] = 10
R10 [t2] = 0
R11 [t3] = 0
R12 [t4] = 0
R13 [t5] = 0
R14 [t6] = 0
R15 [t7] = 0
R16 [s0] = 0
R17 [s1] = 0
R18 [s2] = 0
R19 [s3] = 0
R20 [s4] = 0
R21 [s5] = 0
R22 [s6] = 0
R23 [s7] = 0
R24 [t8] = 0
R25 [t9] = 0
R26 [k0] = 0
R27 [k1] = 0
R28 [gp] = 10008000
R29 [sp] = 7ffff8d4
R30 [s8] = 0
R31 [ra] = 400018
```

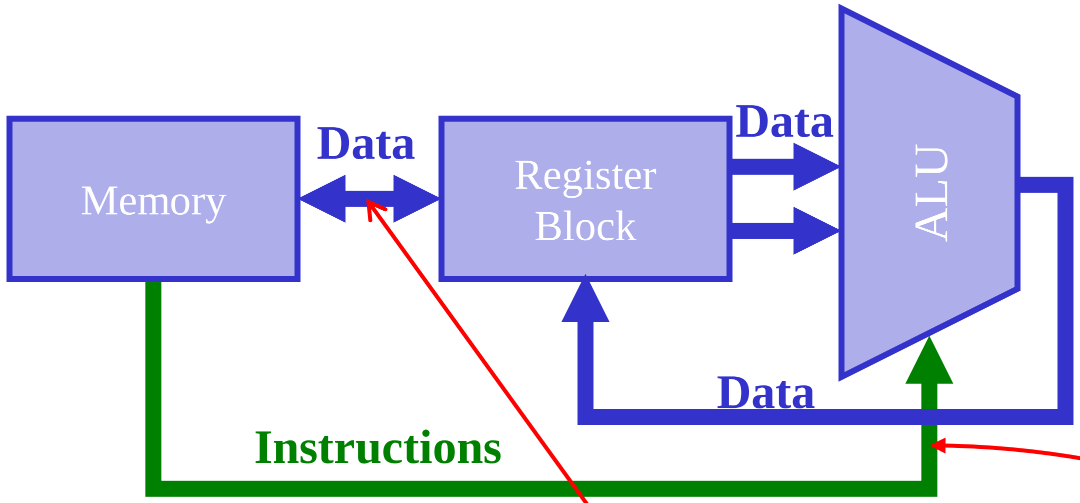
QtSPIM Register Display (Fixed-Point Registers)

- **The program counter is a register that always contains the memory address of the next instruction (i.e., the instruction following the one that is currently executing). It is the first register displayed in the Fixed Point Register list on the far left of the QtSPIM display.**
- **The PC can be accessed/modified by jump and branch instructions.**
- **In the actual screen shot above, the PC (very top; see red circle) holds the address 0x00400064, which means that the currently-executing MIPS instruction is at memory location 0x00400060.**

Where are the Program and Data Stored?

- **Both data and text (instructions) are stored in memory.**
- **However, these two quantities are stored in different areas of memory, and accessed by different data pathways.**
- **The MIPS computer can address 4 Gbyte of memory, from address 0x0000 0000 to 0xffff ffff.**
- **User memory is limited to locations below 0x7fff ffff.**
- **Text (program) storage always starts at 0x0040 0000.**
- **Data storage always starts at 0x1001 0000.**

Data and Instruction Pathways

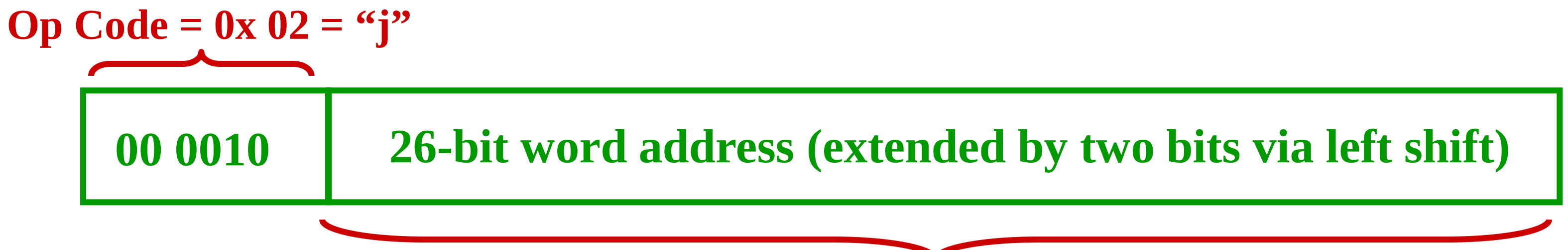


- Data and instructions enter the CPU via different pathways. Data **must** enter via the register block.
- Instructions proceed **directly** to the instruction decoder in the CPU.

Jump Instructions

- The normal form of the jump instruction is: j label
- "jump" always points to a labeled memory address.
- The next instruction executed is the one at memory location "label." This transfer is unconditional. Examples:
  - j loop – The next instruction executed is the one labeled "loop."
  - j go – The next instruction to be executed is labeled "go."
  - j start – The next instruction executed is at the memory location labeled "start."
  - There is NO option on jump instructions. Again, a jump is always to a labeled location.
  - Jump and branch instructions are the reason that text lines (instructions) are labeled in a program.

Format of the Jump Instruction



This [26+2]-bit address is added to the upper 4 bits of the PC to calculate the address of the next instruction to be executed.

- The jump instruction has its own unique format.
  - During assembly, SPIM calculates the real memory address of the destination, removes the top 4 bits, does a shift right 2, and inserts the resulting 26 bits into the instruction for the label.
  - On execution, the CPU reverses this process to create the address.
- The new instruction address is loaded into the PC.
- This address format provides a “range” of ~ 268 Mbytes.

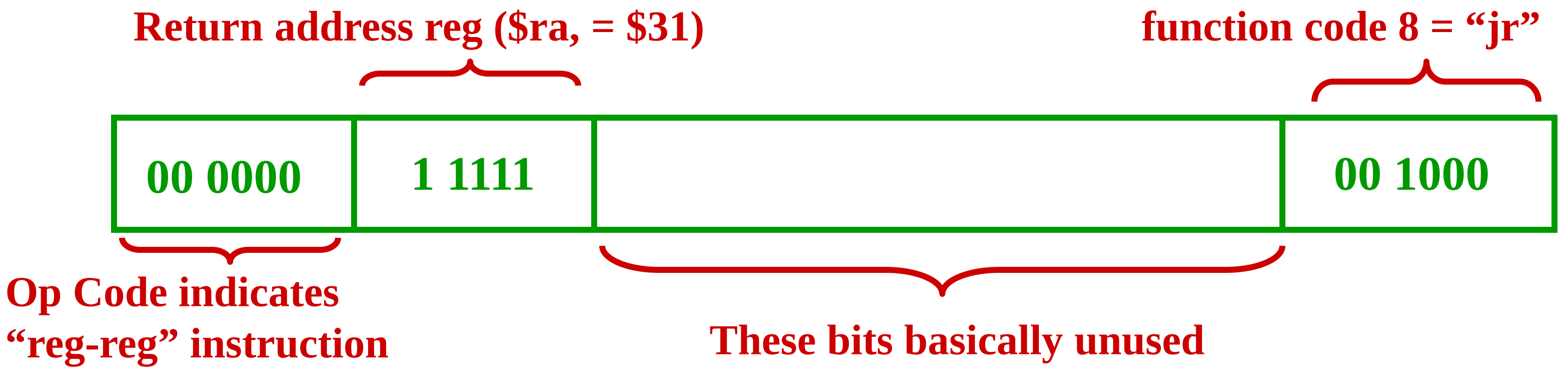
Jump and Link (or Load)

- The form of the jal instruction ("jump and link") is identical to jump, except that the op code is 0x 3 (binary 00 0011).
- In the jal instruction, however, there is an additional step:
  - j go – the next instruction executed will be from the memory location labeled "go."
  - jal go – identical to j go, plus the value of [PC]+4 → \$ra (\$31). That is, the 32-bit address of the next instruction after the jal is loaded into \$ra.
  - jal is used with the "jr" instruction to facilitate entering and exiting procedures.

Jump Register (or Jump Return)

- As mentioned on the previous slide, jr is used with jal.
- jr normally employs the return address register, $ra (=$31), but it can be used with other registers as well.
- The form of the jr instruction is different than j or jal. jr uses the "register-register" op code, and a function code of 8 (next slide).
- The assembly language version of the instruction is:
  - jr $rs – "Unconditionally jump to instruction address [$rs]."
  - jr $ra is the usual form, since jal stores the next address in $ra.
  - We will not use the instruction jalr, "jump and link register."

Format for jr Instructions


Return address reg ($ra, = $31)
function code 8 = "jr"
00 0000
1 1111
00 1000
Op Code indicates "reg-reg" instruction
These bits basically unused

- jr has a format like the register-register instructions.
- The op code is "0" and the "rs" field is the 5-bit address of the register containing the next instruction address. This address is normally 31 (= $ra), since jal stores the address [PC+4] there.
- The function code is the tag denoting that this is a jump instruction, specifically jr.
- The remaining bits are unused.

Use of jal/jr Together

- The uses of jal and jr are tied together.
- These instructions may be used for subroutine or procedure calls, to provide a convenient entry to/exit from utility software.
  - jal sub – the next instruction executed is in a different code segment, starting at memory location "sub." The processor stores the address of the instruction following the jal instruction ([PC]+4) in $ra.
  - At the conclusion of the called routine (starting at label "sub"), the last instruction is jr $ra. This returns the program to its primary flow at the instruction that followed the original jal.
- We will use jal and jr in future exercises.

Example: Jump Assembly-to-Machine Instruction

1. A program instruction is j loop. If the instruction labeled "loop" is at text memory location 0x0040 0080, what value (in hex) goes into the 26-bit field of the jump instruction?
2. The 26-bit field in a jump instruction is 0x0100080. What is the actual address the jump instruction refers to if the top four bits of the Program Counter are 0000.

Example 1 Solution

0000 0000 0100 0000 0000 0000 1000 0000

0x 0 0 4 0 0 0 8 0

$\downarrow$ Strip out upper 4 PC bits

0000 0100 0000 0000 0000 1000 0000

$\downarrow$ right shift 2

0000 0100 0000 0000 0000 1000 00

$\downarrow$ Regroup

26-bit field of j instruction $\rightarrow$ 00 0001 0000 0000 0000 0010 0000

0x 0 1 0 0 0 2 0

The hex content of the field is 0x0100 020

Example (3)

00 0001 0000 0000 0000 1000 0000

↓ Left shift 2

00 0001 0000 0000 0000 1000 0000 00

↓ Regroup

0000 0100 0000 0000 0010 0000 0000

↓ Add upper 4 bits of PC

0000 0000 0100 0000 0000 0010 0000 0000

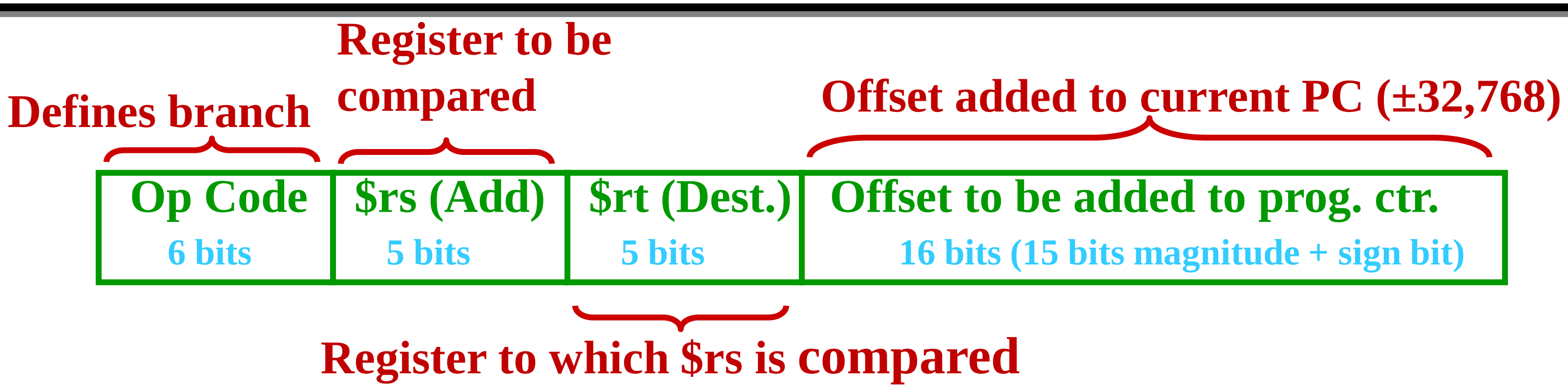
0x 0 0 4 0 0 2 0 0

The address = 0x0040 0200

Branch Instructions

- **Branch instructions (often used with jump instructions) allow SPIM programmers to design decision-making ability into a program.**
- **For that reason, they can be referred to as "program control" instructions, since they support the ability for a program to determine when to change operation.**
- **In general, a branch performs a comparison. If the comparison is successful, the next instruction executed is at another point in the program.**
- **If the desired comparison is not achieved, the program simply executes the instruction following the branch.**

Branch Uses the "Immediate" Instruction Format


Register to be compared
Defines branch
Offset added to current PC (±32,768)
Op Code
6 bits
$rs (Add)
5 bits
$rt (Dest.)
5 bits
Offset to be added to prog. ctr.
16 bits (15 bits magnitude + sign bit)
Register to which $rs is compared

- Branch instructions use the I-format (same as load/store).
- In this case, however, $rs is the register to be compared or evaluated.
- $rt contains the standard of comparison. If an immediate is specified (real number), $rt=$at (immediate → $at).
- The op codes for branch are usually 01 and 04-07.

The Concept of Program Control and Branching

- The branch instructions allow decision points in the program to assure that the program:
  - Enters or exits a loop (discussed shortly);
  - Determines when a repetitive calculation has reached a certain number of interactions;
  - Decides when a program has run to completion;
  - Decides when to call a procedure to execute;
  - Decides when a program should end.
- Branch instructions give the assembly language programmer the key tool for designing programs which can perform multiple, repetitive calculations.

Branch Instruction Types

- The following list is not complete, but composes a relatively useful group of the branch instruction subset.
  - Branch on equal: beq $rs, $rt, label – If [$rs]* = [$rt]*, branch to label; otherwise execute the next instruction.
  - Branch on greater than or equal zero: bgez $rs, label – If [$rs] ≥ 0, branch to label; otherwise execute next instruction.
  - Branch on greater than zero: bgtz $rs, label – If [$rs] > 0, branch to label; otherwise execute the next instruction.
  - Branch on less than or equal zero: blez $rs, label – If [$rs] ≤ 0, branch to label; otherwise execute the next instruction.
  - Branch on less than zero: bltz $rs, label – If [$rs] < 0, branch to label; otherwise execute next instruction.

* [ ] = "Contents of."

Branch Instructions (2)

- Branch on not equal: **bne $rs, $rt, label** – If [$rs] ≠ [$rt], branch to label; otherwise → next instruction.
- Branch on equal zero: **beqz $rs, label** – If [$rs] = 0, branch to label; otherwise → next instruction.
- Branch on greater than or equal: **bge $rs, $rt, label** – If [$rs] ≥ [$rt], branch to label; otherwise → next instruction.
- Branch on greater than: **bgt $rs, $rt, label** – If [$rs] > [$rt], branch to label; otherwise → next instruction.
- Branch on less than or equal: **ble $rs, $rt, label** – If [$rs] ≤ [$rt], branch to label; otherwise → next instruction.
- Branch on less than: **blt $rs, $rt, label** – If [$rs] < [$rt], branch to label; otherwise → next instruction.
- Branch on not equal zero: **bnez $rs, label** – If [$rs] ≠ 0, branch to label; otherwise → the next instruction.

Set Instructions

- **The set instructions are used in decision-making functions (often with branch instructions) within a program.**
- **The format for set instructions is the same as for MIPS register-register instructions unless an immediate is involved, in which case the format of a branch instruction is employed.**
- **The set instruction list shown on the next page is not complete, but is a good representation of types.**
- **In most cases, our examples will not use set instructions; however, they can be valuable in certain cases.**

Set Instructions

- Set less than: slt \$rd, \$rs, \$rt – [\$rd] =1 if [\$rs] < [\$rt], 0 otherwise.
- Set less than immediate: slti \$rd, \$rs, immediate – [\$rd] = 1 if [\$rs] < immediate, 0 otherwise.
- Set equal: seq \$rd, \$rs, \$rt – [\$rd] = 1 if [\$rs] = [\$rt], 0 otherwise.
- Set greater than or equal: sge \$rd, \$rs, \$rt – [\$rd] = 1 if [\$rs] ≥ [\$rt], 0 otherwise (an immediate may be substituted for [\$rt]).
- Set greater than: sgt \$rd, \$rs, \$rt – [\$rd] = 1 if [\$rs] > [\$rt], 0 otherwise (an immediate may be substituted for [\$rt]).
- Set less than or equal: sle \$rd, \$rs, \$rt – [\$rd] = 1 if [\$rs] ≤ [\$rt], 0 otherwise (an immediate may be substituted for [\$rt]).
- Set not equal: sne \$rd, \$rs, \$rt – [\$rd] = 1 if [\$rs] ≠ [\$rt], 0 otherwise (an immediate may be substituted for [\$rt]).

Definition: "set" is like "branch," except that for a successful comparison, a 1 goes into a destination register rather than the program branching to a different location.

The Most Useful Programming Function: The Loop

- **Branch instructions enable the most important computer function, the loop.**
- **Modern electronic computers have the ability to perform computing actions repetitively, at very high speed.**
  - **Many modern engineering and scientific problems cannot be solved "exactly;" approximate solutions are found using iterative calculation techniques.**
  - **Business functions performed on computers involve repetitive arithmetic and clerical functions, such as payroll calculations, receivables, and taxes.**
- **All these functions are done with loops.**

Demo Program 1: Loop Example

- **The following is a simple loop program that looks for the letter "l" in the phrase "hello world."**
- **Since there are three l's in the phrase, we know the final result, but notice how the program is structured to:**
  - **Set up the loop to examine the entire phrase.**
  - **Do the comparison once in each loop iteration, using a beq.**
  - **Exit the loop and report the results.**
- **The loop uses the ".asciiz" declaration. Since the phrase is declared as null-terminated (".asciiz"), the loop hunts for a null (ASCII 0x00) character (using beqz) to determine when all characters have been examined.**

The Loop, Step by Step

- We are going to go through the string “Hello, world!” looking for l’s. The first thing to do is the data declaration (which we will call “str” for string:

```
            .data
str:        .asciiz “Hello, world!”  # asciiz declaration always put
                                     # in quotes.
```

- What do we do next? Start to write the program with a text declaration.

Starting the Loop Program

```
        .text
main:
```

- **What will the first instruction be?**
  - **We have to examine every character in the string. Since we will be in a loop, we must somehow change the address of each letter in the string each time through the loop.**
  - **That means we must use the "register contents plus offset" addressing method.**
  - **What address can we use? We remember that the address of a string is the address of the first letter in the string.**

Program: First Step

```
         .text
main:    la $t0,str           # address of the H, first letter in the string, is
                              # the address of "str."

         .data
str:     .asciiz "Hello, world!"
```

- **We put the address of the string in $t0. Once it is in $t0, we can treat it like any other number. We will see what an advantage this is in a moment.**
- **What next?**
- **Now we can start to construct a loop.**

Starting the Loop

```
        .text
main:   la $t0,str          # address of the H, first letter in the string, is
                            # the address of "str."

loop:   lb $t1,0($t0)       # We start the loop by loading the first letter.
        beqz $t1,over       # Since the loop is null-terminated (.asciiz), we
                            #  put in a test for null (0). When we find a null,
                            #  we know we are at the end of the loop, so we
                            #  exit.
        .data
str:    .asciiz "Hello, world!"
```

- **We start the loop, but we also immediately put in a way to get OUT of the loop.**

Doing the "Work" of the Loop

- In the loop, what are we looking for? The letter "l."
- The ASCII code for l is 0x6c in hex.
- Thus we test for the value 0x6c. When we find it, we make note of it. How? We add one to a register to effectively count each occurrence of l.
- If a letter is not l, we ignore it and simply get the next letter to check.
- We do that by changing the address in $t0, and then loading the next letter in $t1.
- Thus (next page):

Counting the "l's"

```
        beq $t1,0x6c,cnt    # if the character is an l, go to count
incr:   addi $t0,$t0,1      # add 1 to current byte address
        j loop              # get next byte to compare
cnt:    addi $t2,$t2,1      # add one to count of letter l's in phrase
        j incr              # go back into loop
```

- The five instructions above check each letter to see if it is an l and move the loop forward.
- If the letter is an l, the count is registered by adding one to $t2, effectively counting l's.
- Note that whether or not we count an l, we still add one to the address in $t0, so that when we go back through the loop, we get the next letter in the string to test.

Finishing the Program

- The loop is complete: We enter it, load each letter sequentially into $t1, and test for l's. We exit the loop when we encounter a null character.
- We finish the program by printing out the answer.
- We want to have an announcement of the answer contents, so we change the data declaration a bit to add the answer leader:

```
        .data
str:    .asciiz "Hello, world!\n"
rept:   .asciiz "The total count of the letter l is "
```

Finishing the Program (2)

- Now all we have to do is write the output sequence and stop command.
- We will output the leader with a system call 4.
- We output the number of l's by moving the number from $t2 to $a0 and doing syscall 1 (remember, syscall 1 outputs the contents of $a0 as a decimal number).
- Then we use syscall 10 to stop.
- The final sequence of instructions is then (next page):

Finishing the Program (3)

```
over:   la $a0,rept      ┐
        li $v0,4         ├ # output report phrase
        syscall          ┘
        move $a0,$t2       # move total l-count to $a0 for output
        li $v0,1         ┐
        syscall          ┘ # output letter total
        li $v0,10        ┐
        syscall          ┘ # end program
```

- Our first loop program is now complete. The full program is shown on the next slide.

```
#               Lecture 13 Demo Program 1:  “L Finder”
# This program counts the number of l’s in “Hello, world!\n”
# The number of l’s is printed on the console.

            .text
main:       la $t0,str              # put starting address of "hello world" into t0
loop:       lb $t1,0($t0)           # load byte in phrase
            beqz $t1,over           # if character null, we are finished
            beq $t1,0x6c,cnt        # if the character is an l, go to count
incr:       addi $t0,$t0,1          # add 1 to current byte address
            j loop                  # get next byte to compare
cnt:        addi $t2,$t2,1          # add one to count of letter l's in phrase
            j incr                   # go back into loop
over:       la $a0,rept
            li $v0,4                # Output report phrase
            syscall
            move $a0,$t2            # move total l-count to $a0 for output
            li $v0,1
            syscall                 # output letter total
            li $v0,10
            syscall                 # end program

            .data
str:        .asciiz "Hello, world!\n"
rept:       .asciiz "The total count of the letter l is "
```

Summary

- **<u>In the MIPS RISC architecture, program memory is only accessed via jump and branch instructions</u>. That is, jump and branch instructions are the only way to modify the program counter.**
- **Both j and jal unconditionally transfer program control from one section of a program to another.**
- **Jal and jr allow calling a subroutine and then returning from it to the point from which the subroutine was called.**
- **Branches allow the programmer to add "intelligence" to a program. The branch instruction uses comparisons to allow decision-making with respect to two alternatives within a program.**

Program 2

- Using the li instruction, put 23, 67, and 45 into registers \$t0-\$t2, respectively.
- Now, write a program that compares the numbers in the three registers and outputs the largest. Note: this program will not be a loop!
- Yes, you know which is larger, but the MIPS doesn't. Write your program so that it will compare the three registers and output the largest number, regardless of which register it is in.
- Check your program by changing the li instructions to put the largest number in each of the other two registers to check the logic of your program.

Program 3

- In the data section of your program, declare an .asciiz string: "Hello, world!\n".*
- Now write a brief program to count the lower-case letters in the phrase. Hint: the hex values of the ASCII codes for a-z are 0x61-0x7a.
- When you have completed the count, output that number to the console.
- How do you end your loop? Remember, the .asciiz string of letters is <u>null-terminated</u>!
- If you wish, you can output an answer leader such as "The number of lower case letters in Hello, world! is:"

Remember that "\n" is the symbol for CR/LF in a SPIM program.