The Program Counter

PC = 0040003c EPC = 00000000 Cause = 00000000 BadVAddr= 00000000
Status = 3000ff10 HI = 00000000 LO = 00000000

General Registers

R0 (r0) = 00000000	R8 (t0) = 00000000	R16 (s0) = 00000000	R24 (t8) = 00000000
R1 (at) = 00000000	R9 (t1) = 00000000	R17 (s1) = 00000000	R25 (t9) = 00000000
R2 (v0) = 00000000	R10 (t2) = 00000000	R18 (s2) = 00000000	R26 (k0) = 00000000
R3 (v1) = 00000000	R11 (t3) = 00000000	R19 (s3) = 00000000	R27 (k1) = 00000000
R4 (a0) = 00000000	R12 (t4) = 00000000	R20 (s4) = 00000000	R28 (gp) = 10008000
R5 (a1) = 00000000	R13 (t5) = 00000000	R21 (s5) = 00000000	R29 (sp) = 7fffeffc
R6 (a2) = 00000000	R14 (t6) = 00000000	R22 (s6) = 00000000	R30 (s8) = 00000000
R7 (a3) = 00000000	R15 (t7) = 00000000	R23 (s7) = 00000000	R31 (ra) = 00000000

- **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).**
- **The PC can be accessed/modified by jump and branch instructions.**
- **In the actual screen shot above, the PC holds the address 0x0040003c, which means that the currently-executing MIPS instruction is at memory location 0x00400038.**

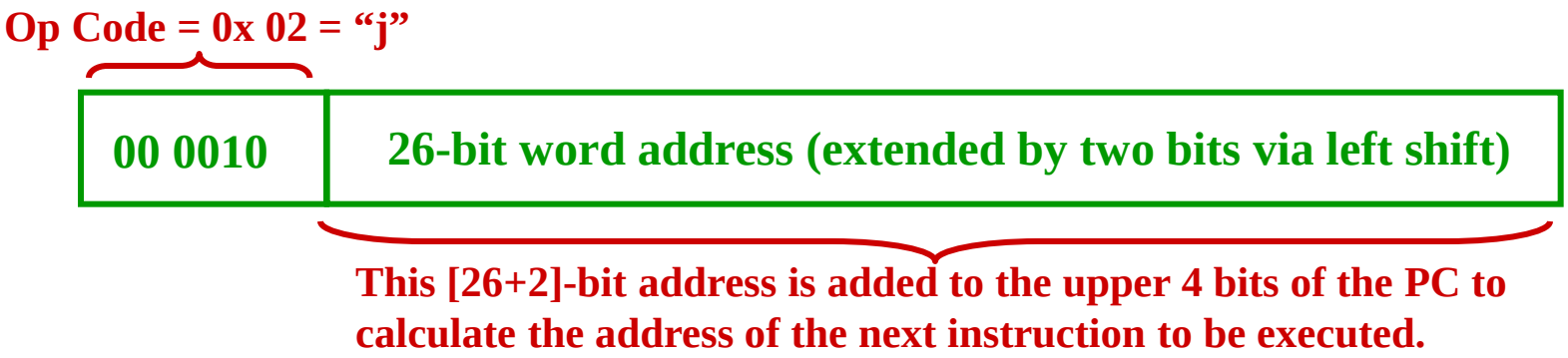
Where are the Program and Data Stored?

- 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.

Jump Instructions

- As noted on the previous slide, the normal form of the jump instruction is: j label
- 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; 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



- **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.**

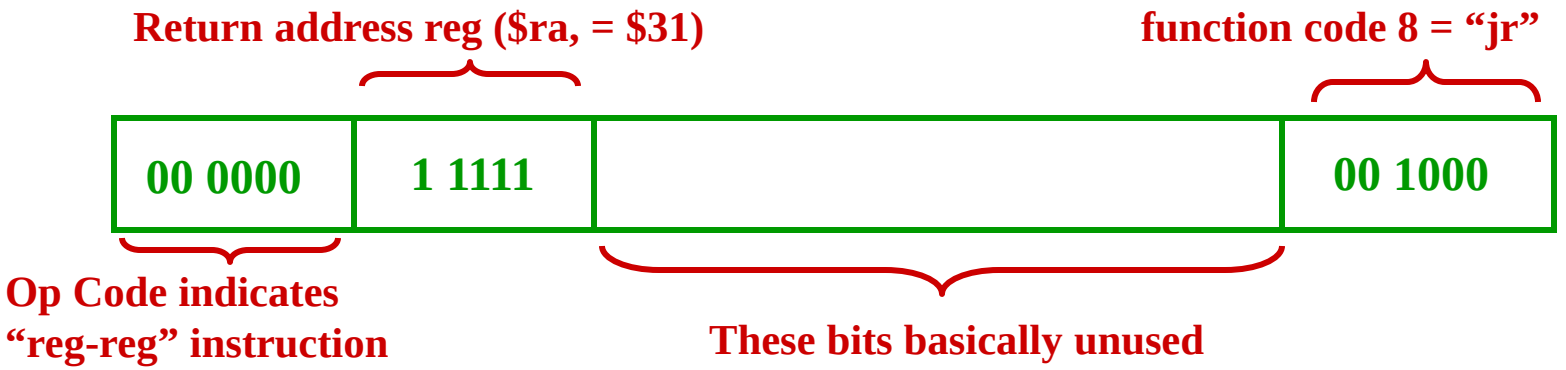
Jump and Link (or Load)

- 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



- **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.**

Examples of Jump Machine Instruction Generation

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

Strip out upper 4 PC bits

0000 0100 0000 0000 0000 1000 0000

right shift 2

0000 0100 0000 0000 0000 1000 00

Regroup

26-bit field of j instruction → 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

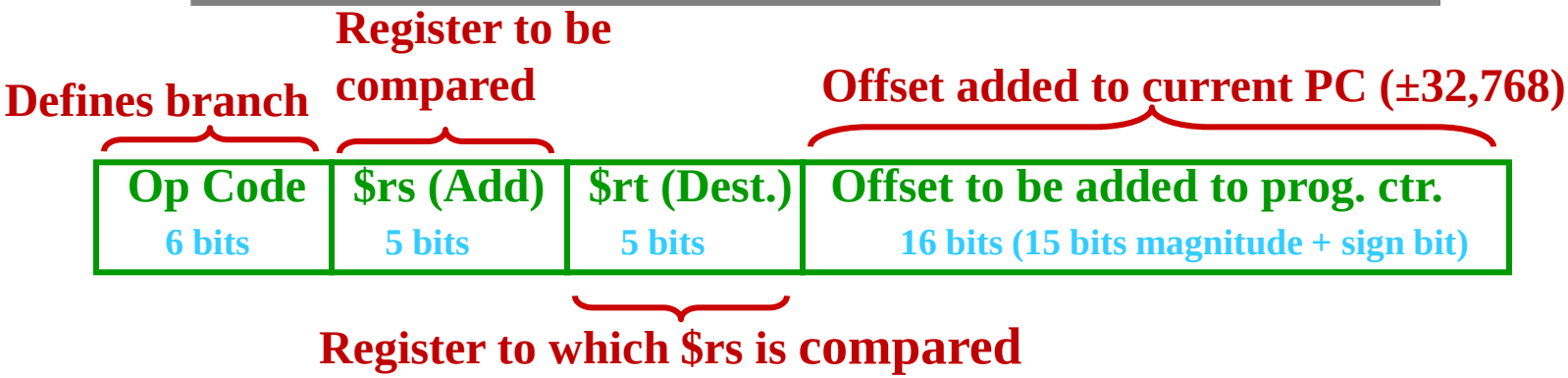
0x 0 0 4 0 0 2 0 0

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.**

Format for Branch Instructions



- Branch instructions are in the I-format (same as load and 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, $rt=$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]).

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.**

```
#               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                # Ouput 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

- **In the MIPS RISC architecture, program memory is only accessed via jump and branch instructions. 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 will compare the numbers in the three registers and output the smallest one.**
- **Yes, you know which is smaller, but the MIPS doesn't. Write your program so that it will compare the three registers and output the smallest number, regardless of which register it is in.**
- **Check your program by changing the li instructions to put the smallest 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 null-terminated!**
- **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.