Pseudo-Instructions

- Pseudo instructions are instructions that exist in the SPIM assembler, but are not instructions designed into the MIPS computer.
- These assembler instructions make SPIM slightly more compiler-like. They are typically made up of two or more MIPS machine instructions, although occasionally, they are a single instruction. As an example, move is a pseudo instruction:

  move $t2,$t1 is converted by the assembler to:

  addu $t2,$0,$t1
- We will see a number of pseudo instructions today.

Multiply as a Pseudo Instruction

- **Since multiplying two 32-bit numbers can have a 64-bit result, the MIPS computer provides two 32-bit registers for a multiply result, LO and HI.**
- **Multiply sends the upper 32 bits to HI and the lower 32 bits to LO.**
  - **MIPS instruction: mul $t1,$t2 means [$t1]•[$t2]→** { **HI (upper 32 bits)** / **LO (lower 32 bits)**
  - **In SPIM, mul $t0, $t1, $t2 implies [$t1]•[$t2]→[$t0].**
  - **The assembler accomplishes this by adding a second instruction: mul $t1, $t2; mflo $t0. (mflo means [LO] →[$t0])**
  - **SPIM recognizes mul $t1,$t2, with results left in LO and HI.**
- **Note: <u>You must check HI for a residual result</u> (product > 32 bits). SPIM does NOT do this!**

Notes: 1. [ ] = "contents of;" 2. The second operand for multiply can be a constant, as for add.

Divide as a Pseudo Instruction

- **Similarly, the MIPS divide is:**
  - **div $t2,$t1, which means [$t2]/[$t1]→** { **HI (32 bit remainder)**, **LO (fixed point 32 bit quotient)**
- **SPIM transfers the quotient from LO to the destination register, as it does the lower 32 bits in multiply.**
- **Thus the SPIM instruction div $t0, $t2, $t1 becomes:***
  - **div $t2,$t1; mflo $t0, where mflo means [LO]→[$t0].**
- **SPIM recognizes div $t2,$t1, which results in the quotient remaining in LO and the remainder in HI.**
- **In divide instructions, the 1st source is divided by the 2nd.**
  - **That is, div $t0, $t2, $t1 means: [$t2]/[$t1] → [LO] → [$t0].**

*** The second operand (divisor) can be a number, as for mul, add, sub, etc.**

Neg and Abs

- **The neg instruction simply changes the mathematical sign of the number in the source register. Thus:**
  - **neg $t2, $s5: ([$s5]+1) → [$t2].**
- **"abs" is also a pseudo instruction. It takes the absolute value of the source register.**
  - **If the source is positive, the number → destination "as is."**
  - **If the source is negative, the two's complement → destination.**
  - **The destination always ends up with a positive number equaling the magnitude of the number in the source.**
  - **Example – abs $t1, $t0: |[$t0]| → [$t1].**

Other Register-Register Instructions

- **lui $rd, #: "load upper immediate." The upper 16 bits $rd are loaded with the immediate ("#"). The lower half is cleared to 0. Since lui is not a pseudo instruction, it is limited to a 16 bit immediate (convenient, since there are 16 bits in the upper half of a register). Example:**
  - **lui $t1, 23 – The number 23 (0000 0000 0001 0111) → upper 16 bits of $t1.**
- **li $rd, #: "load immediate." Although this is a register-type instruction, li is a pseudo instruction (Pervin, Appendix D) which combines ori and lui to create the desired load. Example:**
  - **li $t0, 2405 (# < 16 bits): ori $t1, $at, (binary number [less than 16 bits])**
  - **li $t1, 78645329 (# < 16 bits):**
    lui $at, [upper 16 bits of binary number]
    ori $t1, $at, [lower 16 bits of binary number]

Other Register-Register Instructions (2)

- **move: We discussed move earlier. Move is the only way to transfer the contents of one register to another.**
  - **move $s5, $t3: [$t3] → [$s5] ($t3 contents are not changed).**
- **rem: Rem operates like divide, except that the contents of HI are moved to the destination register. Can be dangerous to use, since the sign of the numbers involved can cause the answer to be undefined. Example of use:**
  - **rem $t0, $t1, $t2 – [$t1]/[$t2], remainder → [HI] → [$t0].**
  - **That is, the instruction becomes two separate instructions, as in the case of divide: div $t1, $t2; mfhi $t0. Yes, the quotient of the divide → LO, but it is not used.**

Segments of a SPIM Program

- We started writing simple SPIM programs last class. What are all the parts of a SPIM program?
- There are three parts of a SPIM program in general:
  - Comments to introduce, comment on, or end the program.
  - The text section (actual program instructions).
  - The data section (identifying data elements).

  Either text or data may come first.
- All data must be labeled (given an identifying acronym for the program [text] to refer to).
- Text statements (i.e., program instructions) may also be labeled for identification and reference in the program.
- "Text" and "data" directives must precede the text and data sections. We will discuss other directives later.

Typical SPIM Program Outline

- **Comments: Title, any special notes on the program, explanation of instructions or instruction groups.**
  - **Comments always preceded by pound sign (#).**
- **Text (the actual program):**
  - **Started with the ".text" directive.**
  - **One SPIM instruction per line; can comment on the instruction line.**
  - **Any text line may contain a label (not required). The only required label is "main:" on the first line of text. A label must be followed by a colon (:).**
- **Data:**
  - **Starts with the ".data" directive.**
  - **Each data statement must include a definition and a label.**
  - **Data may be listed before or after text.**

Types of MIPS Instructions

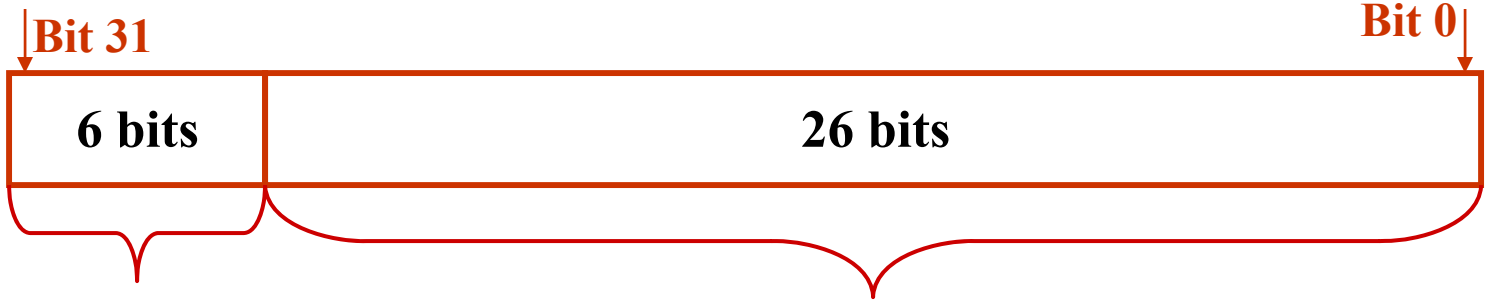
- **The MIPS instruction set is small, and SPIM adds only a few pseudo-instructions.**
- **There are also some special instructions that we will cover later today, directives and system calls.**
- **MIPS instructions are divided into several groups.**
- **The fundamental instruction types, with examples, are:**
  - **Register-to-register instructions: Includes mathematical, logical, and manipulative instructions, such as add, subtract, multiply, divide, logical instructions such as AND, OR, and manipulations such as neg, move, rem, shift and rotate. We covered most of these last lecture and earlier today.**

Types of MIPS Instructions (2)

- **Instructions (continued):**
  - **Special instructions: Nop (no operation), syscall (input/output).**
  - **Decision-making instructions (program control): Branch (beq, beqz, bge, ble, etc.), jump (j, jal, jr), comparison (seq, sge, sgt).**
  - **Memory reference instructions (mainly load and store, since data in memory may only be accessed by loading and storing to/from a register – lw, lb, sw, sb). Note: Load address ("la") is covered with memory reference instructions, although it references a memory address only.**
- **First, we will look at the three fundamental instruction types and their unique formats.**

Instruction Formats

- **There are only three basic MIPS instruction formats: Register (as seen in last lecture), Immediate, and Jump. Jump is shown below.**

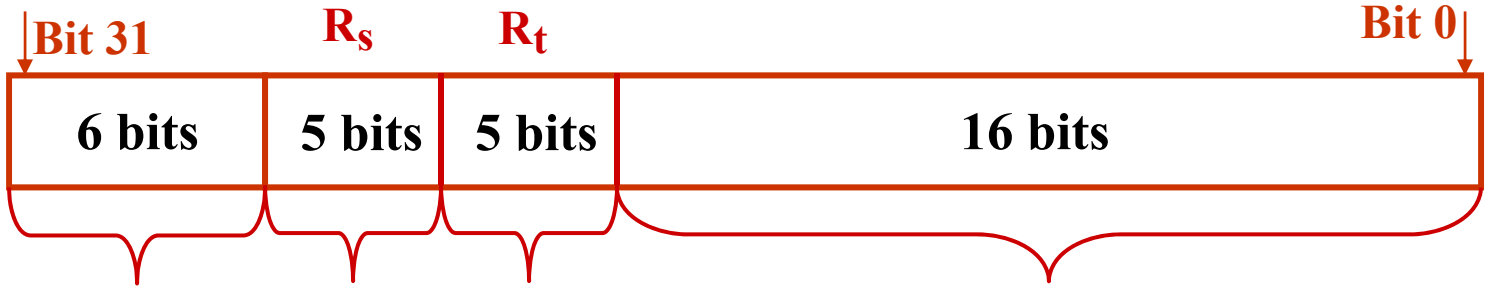


Op Code Field (tells computer what to do when decoded, in this case, jump).

Address Field (added to a part of the Program Counter register contents to create the 32-bit jump address).

Immediate Format Instructions

- **Immediate instruction formats are mainly for branch instructions and load/store instructions, which retrieve/store data in memory, and which we will study later. Their format is shown below.**



Op Code Field (tells computer what to do when decoded).

Register source of part of memory address.

Data source/ destination register.

Address Component Field (16-bit signed number). In a load or store, added to the contents of register R_s to create 32-bit address in memory from which data is loaded or to which data is stored. Added to PC contents when a branch is executed.

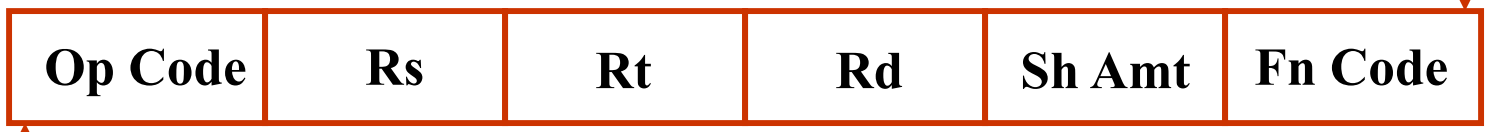
Register-Register Instructions

For most R-type instructions, the six-bit op code is 0x0 (00 0000).

Rs = 0 for shifts

This 5-bit number tells the ALU the shift amount in a shift instruction.

Bit 0



No of bits: 6 5 5 5 5 6

Bit 31

Register addresses are 5 bits (since 2^5=32). Example: If Rt is 0 1001 [09], then Rt = $9 = $t1.

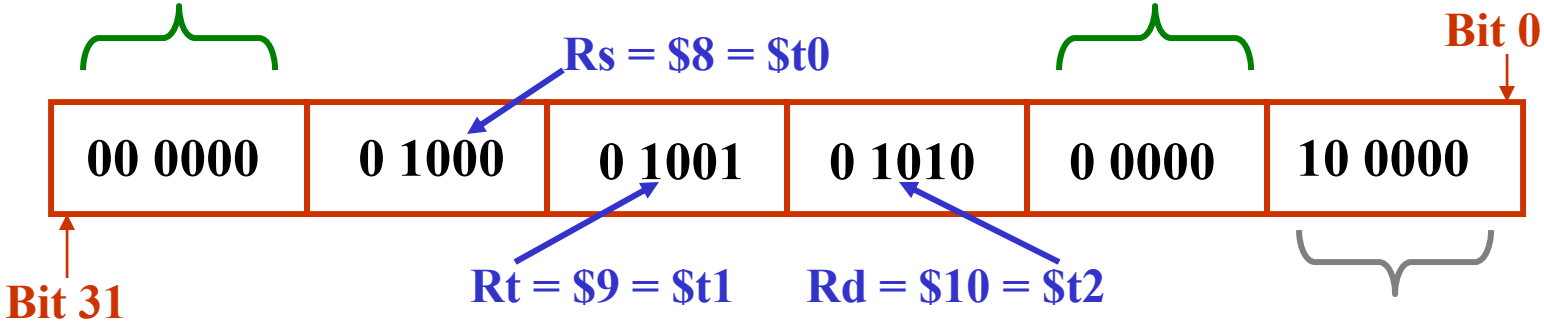
The function code is the register-register function to be performed. Example: Function code 10 0000 [0x20] means "add."

Example on next slide →

Example Register-Register Instruction

For most R-type instructions, the six-bit op code* is 0 (00 0000).

This is not a shift instruction, so the shift amount = 0.



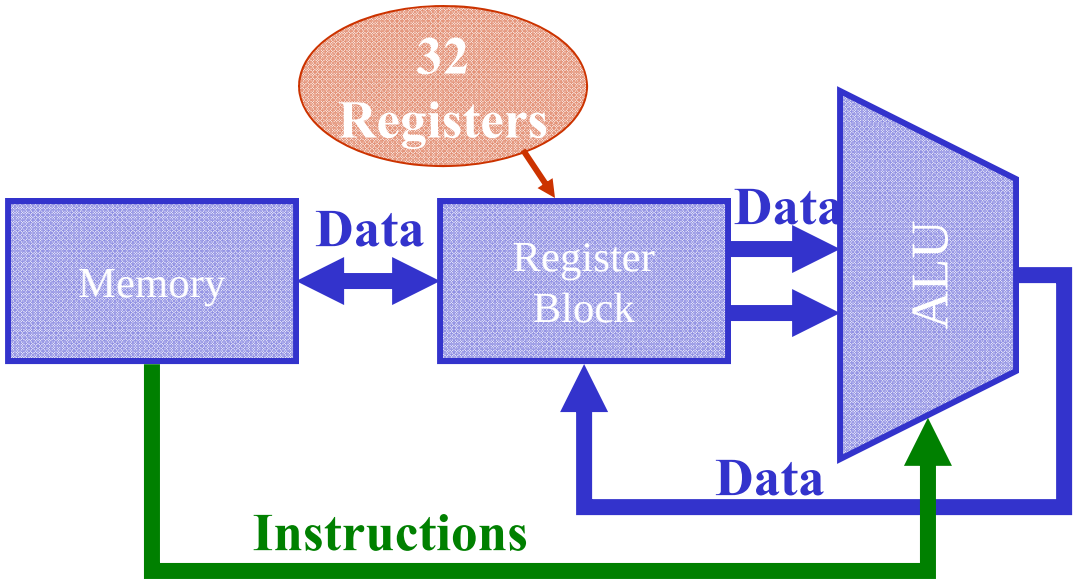
The assembly language acronym for this instruction is: **add $t2, $t0, $t1.**

The function code* for add is 10 0000, or 0x 20 (this is technically "add with overflow").

* A list of op/function codes can be found in the P&H inside front cover or back cover. (Note that if you have an older hard-cover book with codes on the inside back cover, they are shown in decimal form).

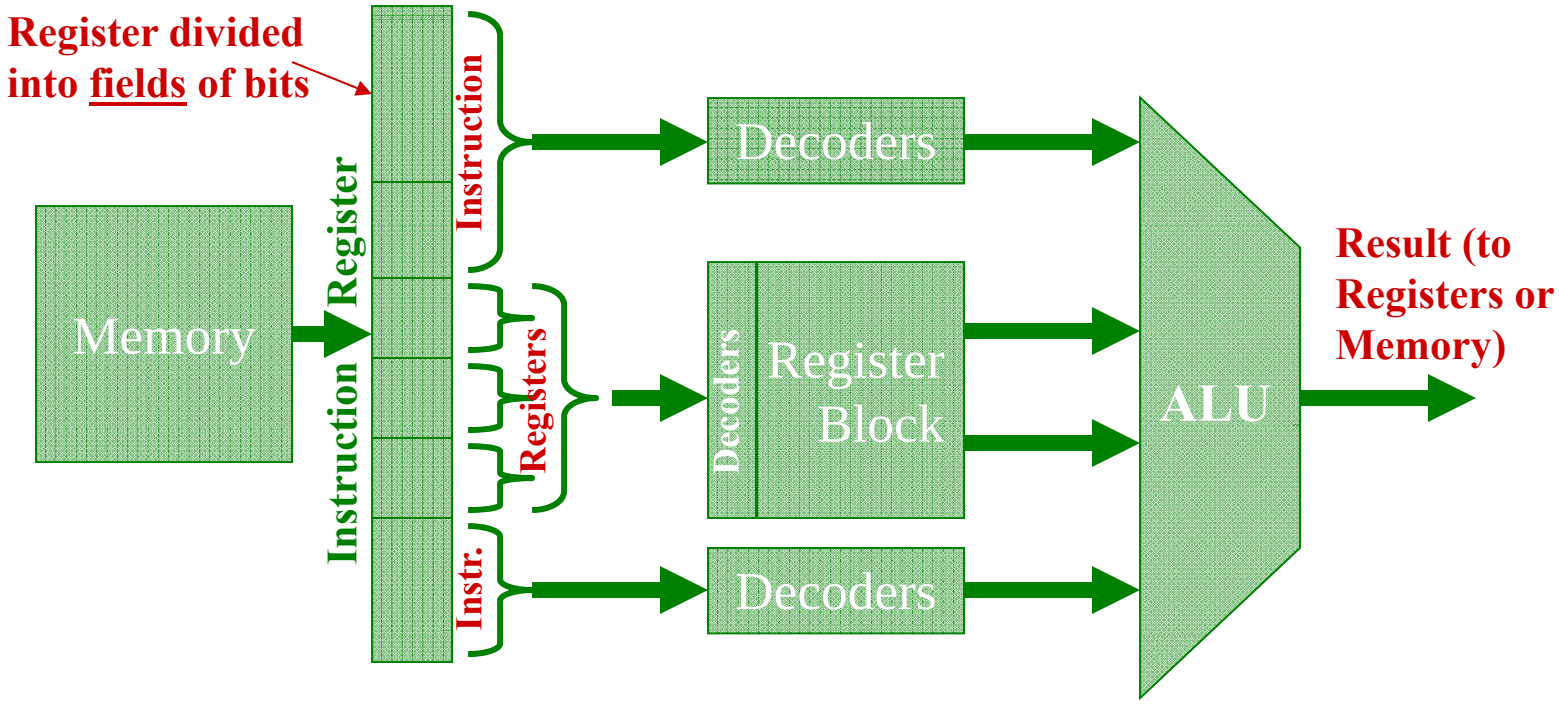
Instruction Execution on a MIPS Computer

- **As we noted in the last lectures the MIPS computer (and in fact, most computers) execute instructions in the method shown below:**
  - **Obtain instruction and information to process (usually from data registers).**
  - **Interpret instruction and do processing in CPU arithmetic unit (ALU or datapath).**
  - **Put results in storage (in data registers).**



Instruction Processing

- **In all computers, an instruction is retrieved from memory, the fields of the instruction are decoded, and the decoded commands are processed.**



Exercise 1

00 0000	1 1001	0 1110	0 1111	0 0000	10 0100

00 0000	0 0111	1 0011	0 1000	0 0000	01 1000

- **This problem will familiarize you with R-R instruction formats.**
  - **Instructions above are R-R. Write the SPIM acronym for them.**
  - **Add function code = 0x20, Sub = 0x22, Mul = 0x18, AND = 0x24.**
  - **Remember to convert register numbers to appropriate acronyms.**

Register-to-Register Instructions

- **A review of R-R instructions studied earlier:**
  - **add $s1, $t2, $t6: [$t2] + [$t6] → [$s1].**
  - **addi $s1, $t2, 27: [$t2] + 27 → [$s1].**
  - **sub $t3, $t4, $t5: [$t4] – [$t5] → [$t3]. Can also be immediate.**
  - **mul $t0, $t1, $t2: [$t1]• [$t2] → [$t0].**
  - **div $t0, $t2, $t1: [$t2]/[$t1], quotient → [LO] → [$t0].**
  - **rem $t0, $t1, $t2: [$t1]/[$t2], remainder → [HI] → [$t0].**
  - **and $t0, $t1, $t2: [$t1]• [$t2] → [$t0]. Also OR, NOT, NOR, XOR.**
  - **neg $t2, $s5: ([$s5]+1) → [$t2].**
  - **abs $t1, $t0: |[$t0]| → [$t1].**
  - **lui $t1, 23: 23 (0000 0000 0001 0111) → upper 16 bits of $t1.**
  - **li $t1, 78645329: 78645329 → $t1 (pseudo; uses $at).**
  - **move $s5, $t3: [$t3] → [$s5]**

Note: In R-R instructions, the source register contents are NOT changed.

Miscellaneous Instructions

- **The following instructions are convenient to mention at this point:**
  - **nop: "No operation." The computer does nothing for one instruction cycle.**
  - **The "set" instructions: These instructions (which are technically register-register instructions) generally put a 1 or 0 into a destination register, depending on the comparative values of two other registers. We will cover these later in the lecture on branch instructions, as they are decision-making (or more properly, decision-support) instructions.**
  - **Shift, and rotate instructions (also R-R instructions) will also be covered in that lecture.**

System Calls

- **System calls allow our simulated MIPS to communicate to the user and perform a few other functions.**
- **Most system calls either initiate a print sequence to the system console, or read data from the keyboard.**
- **Syscall 10 (as you know!) halts the program.**
- **Certain system calls use specific registers (primarily $v0 and $a0). The system calls will not execute unless the designated registers contain the correct information.**
- **System calls are the user interface in SPIM.**

System Calls of Interest

1 - Print int., $a0 = int. $a0 contents printed as decimal #.

4 - Print string; $a0 = pointer to string. ASCII string is printed out.

5 - Read int; $v0 holds #. # from keyboard → $v0.*

8 - Read string; $a0 = buffer, $a1 = length. String is read from keyboard.*

10 - Exit program. Program ends.

11 - Print character, Lowest byte of $a0 → console as ASCII.

12 – Read character. One ASCII character from kbd → $v0

Note: System Calls 2, 3, 6, 7, 9 and 13-16 will not be used.

*Input system calls 5 and 8 are a little tricky; more about them later.

Mechanics of Writing a System Call

- **Executing a system call requires two instruction:**
  - **(1) # of syscall → $v0; (2) syscall**
  - **For example, we have used syscall 10 ("stop the program"):**

```
li $v0, 10
syscall
```

"syscall 10"

- **Sometimes $a0 or $a1 are used in syscalls. For example, system call 1 prints out the contents of $a0.**

```
li $a0, 0x ab67  (might also come from memory or via move)
li $v0, 1
syscall
```

← **syscall 1 prints out a decimal value**

More Example System Calls

```
la $a0, string
li $v0, 4
syscall
```

In syscall 4, $a0 is loaded with the memory address of the first byte in the ASCII string to be output. Characters are output until a null (character = 0) is encountered.

```
lb $a0,num (or use li or move)
li $v0,11
syscall
```

Syscall 11 prints out the lower 8 bits of $a0 as an ASCII character.

```
li $v0,12
syscall
```

In syscall 12, the 8 bits representing the ASCII character input from the keyboard → the lowest 8 bits of $v0 (erasing the 12!).

Directives

- **Directives give the SPIM assembler program information. The main ones of interest are shown here:***
  - .text -- "Program instructions follow."
  - .data -- "Data required by the program follows."
  - .word [number] -- "The data element that follows is a 32-bit number."
  - .space [number] -- "Reserve the number of bytes in memory shown and clear them to 0."
  - .asciiz [character string] – "Store the following in memory as a byte string and null terminate."
  - .ascii [character string] – Same as .asciiz but no null termination.
  - .frame -- "Reserve the frame as shown on the stack."

*** See Pervin appendix D for full list. All directives are preceded by a period.**

Labels

- **Labels attach a name to data or to a text (program) line. Labels allow easy reference to a specific place in a program or to a specific piece of data.**
  - **Each data entry must be labeled with a unique name, or label.**
  - **The specific name "main:" MUST label the first line of text in a PCSPIM program.**
  - **Labels are always followed by a colon (:).**
- **Examples:**
  - **str: .asciiz "Hello, world!\n" -- Characters are named "str"**
  - **adder: add $t1,$t2,$t3 – This instruction is named "adder."***
  - **calc: mul $t5,$t5,$t6 -- This instruction is named "calc."**

*** Note: Do NOT use an instruction as a label. The assembler will reject it.**

Comments

- **Comments are VERY useful in describing a program. You may not think you need them, but you do. They help remind you (later) how the program works!**
  - **Comments are always preceded by a pound sign. The assembler ignores a line (or part of a line) after a #.**
  - **Comment lines do not wrap. A two or more line comment must have a # sign at the beginning of each line.**
  - **A comment may be appended on any instruction or data line. Simply enter a # and then the comment. Examples:**
  - **add $t1,$t1,$t6 # Computing the value K.**
  - **mul $s1,$t1,$t1 # Calculating the area of the square.**
  - **temp: .word 212 # Defining temperature constant.**
  - **.main: and $t1,$t7,$t8 # Start program with logical AND.**

Reminder: Form of SPIM Program

- **Comments: Title, any special notes on the program, explanation of instructions or instruction groups.**
  - **Comments always preceded by pound sign (#).**
- **Text (the actual program):**
  - **Started with the ".text" directive.**
  - **One instruction per line; can comment on the instruction line.**
  - **Any text line may contain a label (not required). The only required label is "main:" on the first line of text. A label must be followed by a colon (:).**
- **Data:**
  - **Starts with the ".data" directive. Data declarations covered later.**
  - **Each data statement must include a definition and a label.**
  - **Data may be listed before or after text.**

Program 1

- **Let's do a program to practice system call 4.**
  - **Declare an .asciiz data declaration of "Hello, world!\n". ***
  - **Since all data declarations must be labeled, label your .asciiz character string as "str:"**
  - **Your program should set up and execute a system call 4 to output the greeting, "Hello, world!" to the console.**
  - **Stop the program using the appropriate syscall. (What is that syscall?)**
  - **Remember to use the directives for program(".text") and data (".data").**
  - **Remember to label the first line of your program properly.**
- **This is a 5-line program! (I.e., 5 lines after .text)**
- **The data declaration is also one line! (after .data)**

***Note that \n is the symbol for the ASCII character CR/LF (carriage return/line feed).**

Program 2

- **Now, compose and run a program to calculate the equation $5x^2-3$.**
  - **As a new requirement, a data declaration says "The answer is. "**
  - **This will allow a data section in this program as in the first program and also allow the use of system call 4 again.**
- **When you finish writing the program in NotePad, open it in SPIM and run it to make sure it operates properly.**
  - **If it executes properly on your SPIM emulator, you should see "The answer is 242" on the console, and the program should stop.**
  - **Remember to do your data declaration so that so that "The answer is" is properly declared in the data section.**
  - **This program is a bit longer. You must do the calculation, output the answer header ("The answer is "), and then output the answer itself.**
  - **Remember, to output a decimal number, load it into $a0, and execute a system call 1.**

Program 3

- **Compose a program to do the following:**
  - **Using the li instruction, load $t0-$t3 as follows.**
  - **$t0 = 22, $t1 = 15, $t2 = 7, $t3 = 18.**
  - **Multiply register $t0 by $t1 and store the result in $s0.**
  - **Multiply register $t2 by $t3 and store the result in $s1.**
  - **Multiply $s0 by $s1 and store the result in $a0.**
- **After processing, output the contents of register $a0 to the console using the appropriate system call.**
- **You need NO data section, as there are no data declarations. End the program with syscall 10.**
- **What is the answer?**

MIPS INTEGER and GENERAL INSTRUCTIONS

Instruction	Arg1	Arg2	Arg3	Description
* abs	rd	rs		put the absolute value of rs into rd
add	rd	rs	rt	rd = rs + rt (with overflow)
addu	rd	rs	rt	rd = rs + rt (without overflow)
addi	rt	rs	imm	rt = rs + imm (with overflow)
addiu	rt	rs	imm	rt = rs + imm (without overflow)
and/andi	rd	rs	rt	put rs AND rt into rd (imm, not rt in andi)
* b	label			branch to label
beq	rs	rt	label	branch to label if (rs==rt)
* beqz	rs	label		branch to label if (rs==0)
* bge	rs	rt	label	branch to label if (rs>=rt)
bgez	rs	label		branch to label if (rs>=0)
bgt	rs	rt	label	branch to label if (rs>rt)
bgtz	rs	label		branch to label if (rs>0)
ble	rs	rt	label	branch to label if (rs<=rt)
blez	rs	label		branch to label if (rs<=0)
* blt	rs	rt	label	branch to label if (rs<rt)
bltz	rs	label		branch to label if (rs<0)

*** Indicates a pseudo instruction (assembler generates a different instruction or more than one instruction to produce the same result; note that the pseudo instruction has no op code).**

MIPS INTEGER and GENERAL INSTRUCTIONS (Continued)

bne	**rs**	**rt**	**label**	**branch to label if (rs≠rt)**
*** bnez**	**rs**	**label**		**branch to label if (rs≠0)**
div	**rd**	**rs**	**rt**	**rd = rs DIV rt**
j	**label**			**jump to label**
jal	**[rd]**	**label**		**jump to label; save next instruction in rd**
jalr	**[rd]**	**rs**		**jump to instruction at (rs), save next in rd**
jr	**rs**			**jump to instruction at (rs)**
la	**rd**	**label**		**load address of word at label into rd**
lb	**rt**	**address**		**load byte at address into rt, sign xtnd**
lbu	**rt**	**address**		**load byte at address into rt**
*** li**	**rd**	**number**		**load number into rd**
lui	**rt**	**number**		**upper halfword of rt = 16-bit number**
lw	**rd**	**address**		**load the word at address into rd**
lw	**rd**	**offset(base)**		**load word at addr offset+base into rd**
*** move**	**rd**	**rs**		**move rs to rd**
*** mul**	**rd**	**rs**	**rt**	**rd = rs • rt**
*** neg**	**rd**	**rs**		**rd = – rs**
nop				**do nothing**
nor	**rd**	**rs**	**rt**	**rd = rs NOR rt**
*** not**	**rd**	**rs**		**rd = bitwise logical negation of rs**
or	**rd**	**rs**	**rt**	**rd = rs OR rt**
ori	**rt**	**rs**	**imm**	**rt = rs OR imm**

*** Indicates a pseudoinstruction (assembler generates a different instruction or more than one instruction to produce the same result; note that the pseudoinstruction has no opcode).**

MIPS INTEGER and GENERAL INSTRUCTIONS (Concluded)

* rem	rd	rs	rt	rd = rs MOD rt
* rol(ror)	rd	rs	ra	rd = rs rotated left(right) by ra
sb	rt	address		store byte in rt to address
* seq	rd	rs	rt	if (rs==rt) rd=1;else rd=0
* sge	rd	rs	rt	if (rs>=rt) rd=1;else rd=0
* sgt	rd	rs	rt	if (rs>rt) rd=1;else rd=0
* sle	rd	rs	rt	if (rs<=rt) rd=1;else rd=0
sll	rd	rt	sa	rd = rt shifted left by distance sa
slt	rd	rs	rt	if (rs<rt) rd=1;else rd=0
* sne	rd	rs	rt	if (rs!=rt) rd=1;else rd=0
sra	rd	rt	sa	rd = rt shifted right by sa, sign-extended
srl	rd	rt	sa	rd = rt shifted right by sa, 0-extended
sub	rd	rs	rt	rd = rs - rt
sw	rt	address		store the word in rt to address
sw	rt	offset(base)		store word in rt to addr offset+base
syscall				do a system call depending on contents of $v0
xor	rd	rs	rt	rd = rs XOR rt

*** Indicates a pseudo instruction (assembler generates a different instruction or more than one instruction to produce the same result; note that the pseudo instruction has no op code).**

Note: The above list is not complete, but contains all instructions of interest in EE 2310.

Supplementary Material

- **The following material will not be covered in the lecture.**
- **Read and study it on your own, and you will have the opportunity (last slide) to earn a homework bonus.**

Rationale for a Computer Instruction Set

- **Another thing that writing programs in assembly language does for us is to give us visibility into the fundamentals of the computer instruction set.**
- **A great deal of research and development has gone into determining just what instructions a computer should embody in its repertoire, and why those instructions should exist.**
- **The study of what and how many instructions to include in a computer design is referred to as "Instruction Set Architecture."**
- **The "Instruction Set Architecture," or ISA, of a computer refers not only to what instructions a computer will execute, but how those instructions relate to each other, and why a certain set of instructions make the computer effective at executing certain types of programs.**

Designing an Instruction Set

- **Many questions must be answered to determine the ISA. Some examples:**
  - **What kinds of programs will be run (graphical or media, or extended accuracy for scientific applications)?**
  - **Device basis (type of manufacturing technology – MOS, etc.)**
  - **Integer unit size (bits) to support applications**
  - **Instruction encoding (that is, how the bits in an instruction word will be put together to signify each instruction).**
  - **Computational operations supported in hardware (for example, should there be a multiplier in hardware?).**
  - **The types of data that will be manipulated and stored (fixed point or floating point, for example).**

Kinds of Instruction Set Architectures

- **Following are examples of some of the most pervasive ISAs over the last 30 years:**
  - **Accumulator Architecture:**
    - **CPU has 1 (and only 1) register, which is called the Accumulator. Operations always include memory reference.**
    - **Example-- Add X: X + [contents of Accumulator] ⟶ Accumulator, X the contents of a memory location**
  - **Memory-Based Computational Architecture:**
    - **No registers in CPU; operations always involve operands in memory.**
    - **Example -- Add X, Y, Z: Contents of memory locations Z and Y are added and stored in memory location X.**

Kinds of Instruction Set Architectures (2)

- **General-purpose register set: There are two classes:**
  - **Non-restrictive design (may allow multiple-word instructions):**
    - **Instructions may involve both register-to-register and register-to-memory operations.**
    - **Examples: add r1,X,Y; rshift r3, 4, X, X and Y memory locations.**
  - **RISC* Load/store architecture:**
    - **All arithmetic/logical operations use register operands only.**
    - **Data in memory accessible only via load and store instructions.**

*** Reduced Instruction Set Computer**

Advantages and Disadvantages of Architectures

- **Accumulator:**
  - + **Minimizes CPU hardware and bus complexity.**
  - − **Maximizes memory-to-CPU traffic (slow).**
- **Memory:**
  - + **Simplest possible CPU design.**
  - − **Very slow (was faster once upon a time, when memories were lots faster than CPUs).**
- **General-purpose register set, non-restrictive design:**
  - + **Most general model for code generation, very flexible.**
  - − **Very complex CPU and memory interface; also longer (i.e., multi-word) instructions may be necessary.**
- **General-purpose register set, RISC load/store design:**
  - + **Single memory access mode simplifies design, ensures maximum speed.**
  - − **Increases total instruction count, makes programming less flexible.**

Why RISC Load-Store Architecture is Superior*

- **Since 1980, logic circuits have become much faster than DRAM.**
  - **Registers are faster than main memory.**
  - **Frequent memory references (except cache) slow computational speeds.**
- **Load-store architecture offers many advantages.**
  - **Operand registers can be used in any order.**
  - **Do not have to go to memory to store data between computations (fast!).**
  - **Load-store architectures allow very straightforward (though less flexible) programming.**
  - **Fixed-length memory addresses → fixed-length instructions (fast!).**
  - **For a register machine, can optimize most frequently used instructions; also reduce the instructions complement to a minimum (fast!).**
  - **CPU is generally less complex (fast and cheap!).**

*** For the present, of course – things always change!**

RISC Architecture is Effective

- Consider the following analysis of Intel X86 programs:

Rank	80x86 Instruction	% Total executed
1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move register-register	4%
9	call (procedure entry)	1%
10	return (procedure exit)	1%
Total		96%

RISC Architecture is Effective (2)

- From the previous chart, we see that although the X86 has hundreds of instructions, with tens of thousands of variations (20,000+ add instructions), only ten instruction types made up 96% of all instructions in most programs.
- Thus the general RISC load-store architecture (as embodied, in our case, by the MIPS architecture), further improves efficiency by restricting the instruction set to a few very valuable (and most-used) instructions.
- Keeping the instruction set small further reduces complexity and cost and, additionally, increases operating speed.

Instruction Set Design and Computer Architecture

- We see that the choice of ISA heavily influences the overall design, or architecture, of the computer.
- The clear winner is the RISC load-store architecture:
  - Fixed-length instructions (increases speed)
  - Reduction in overall instructions (increases speed)
  - Minimization of memory references (increases speed)
  - Reduction in size and complexity of processor (speed and cost)
- The MIPS, our target processor/assembly language to study is based on this "victory" of this ISA.
- The MIPS processor employs a RISC load/store ISA.

Bonus Homework Assignment

- **This homework assignment is worth 100 points on your homework average. However, it will be "invisible" when calculating the average (thus you get an extra bundle of points, 100 maximum, without adding to the divisor when the average is calculated).**
- **Turn in your answers on a separate sheet of paper. The true/false questions may simply be marked T or F, along with the question number.**
- **This assignment is due at the beginning of the next class.**

Bonus Questions

Name _____ **EE 2310 Class Section** _____

1. **Provide a one-sentence definition of a computer ISA.**

2. **(T___/F___) Fixed/floating point choice is not an important ISA condition.**
3. **(T___/F___) The RISC architecture allows mathematical/logical operations directly on arguments in memory.**
4. **(T___/F___) The old-fashioned accumulator architecture is very simple.**
5. **(T___/F___) Computer memory is much faster than the CPU.**
6. **(T___/F___) Load and branch instructions combined occur much more often, on average, than a combination of add, and, sub, and move instructions.**
7. **What does "RISC" stand for?**

8. **(T___/F___) The fastest architecture is the accumulator architecture.**