CS2100 COMPUTER ORGANISATION

http://www.comp.nus.edu.sg/~cs2100/

Lecture #8

MIPS

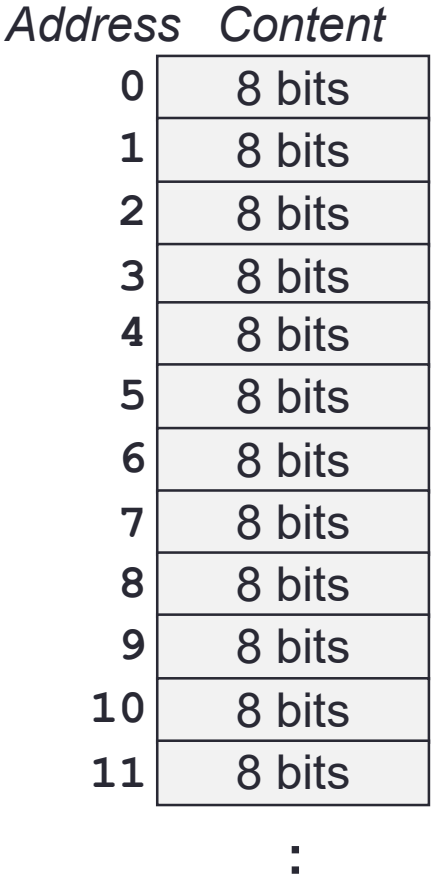
Part II: More Instructions

Lecture #8: MIPS Part 2: More Instructions

Lecture #8: MIPS Part 2: More Instructions

1. Memory Organisation (General)

- The main memory can be viewed as a large, single-dimension array of memory locations.
- Each location of the memory has an **address**, which is an index into the array.
  - Given a *k*-bit address, the address space is of size 2^k.
- The memory map on the right contains one byte (8 bits) in every location/address.
  - This is called byte addressing



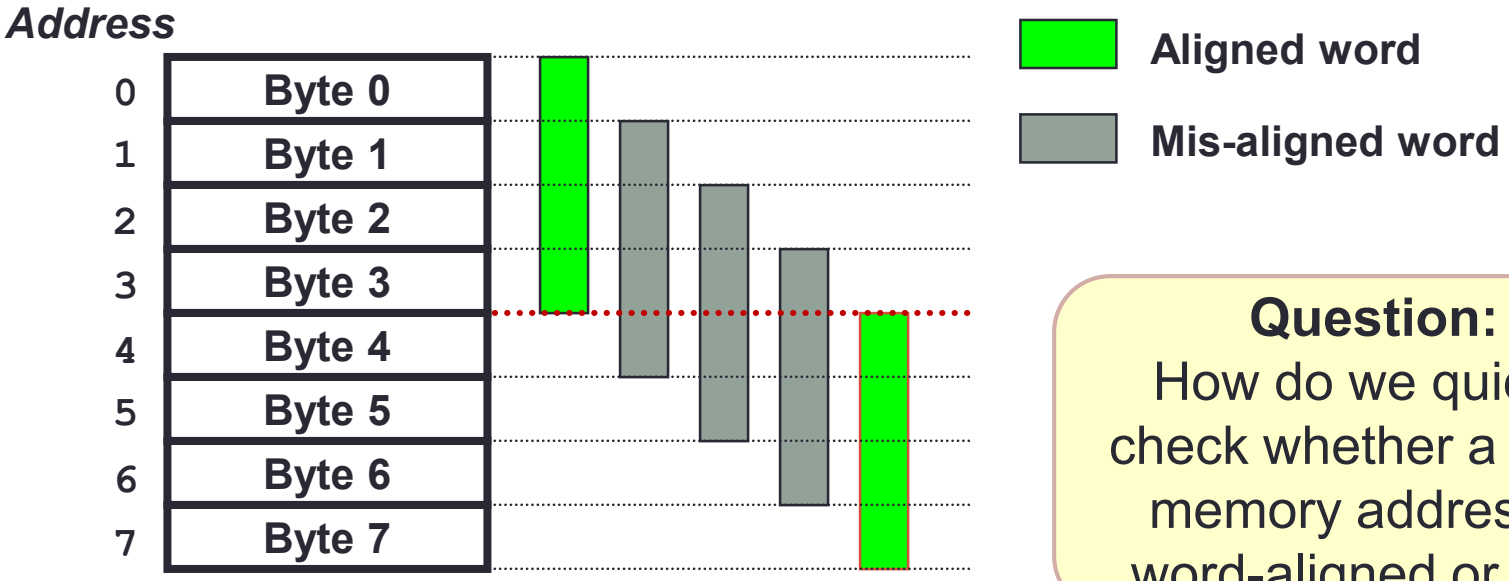
1.1 Memory: Transfer Unit

- Using distinct memory address, we can access:
  - a single **byte** (**byte addressable**) or
  - a single **word** (**word addressable**)
- **Word** is:
  - Usually $\mathbf{2}^{n}$ bytes
  - The common unit of transfer between processor and memory
  - Also commonly coincide with the register size, the integer size and instruction size in most architectures

1.2 Memory: Word Alignment

- **Word alignment**:
  - Words are aligned in memory if they begin at a byte address that is a multiple of the number of bytes in a word.

Example: If a word consists of 4 bytes, then:



Question:
How do we quickly check whether a given memory address is word-aligned or not?

2. MIPS Memory Instructions

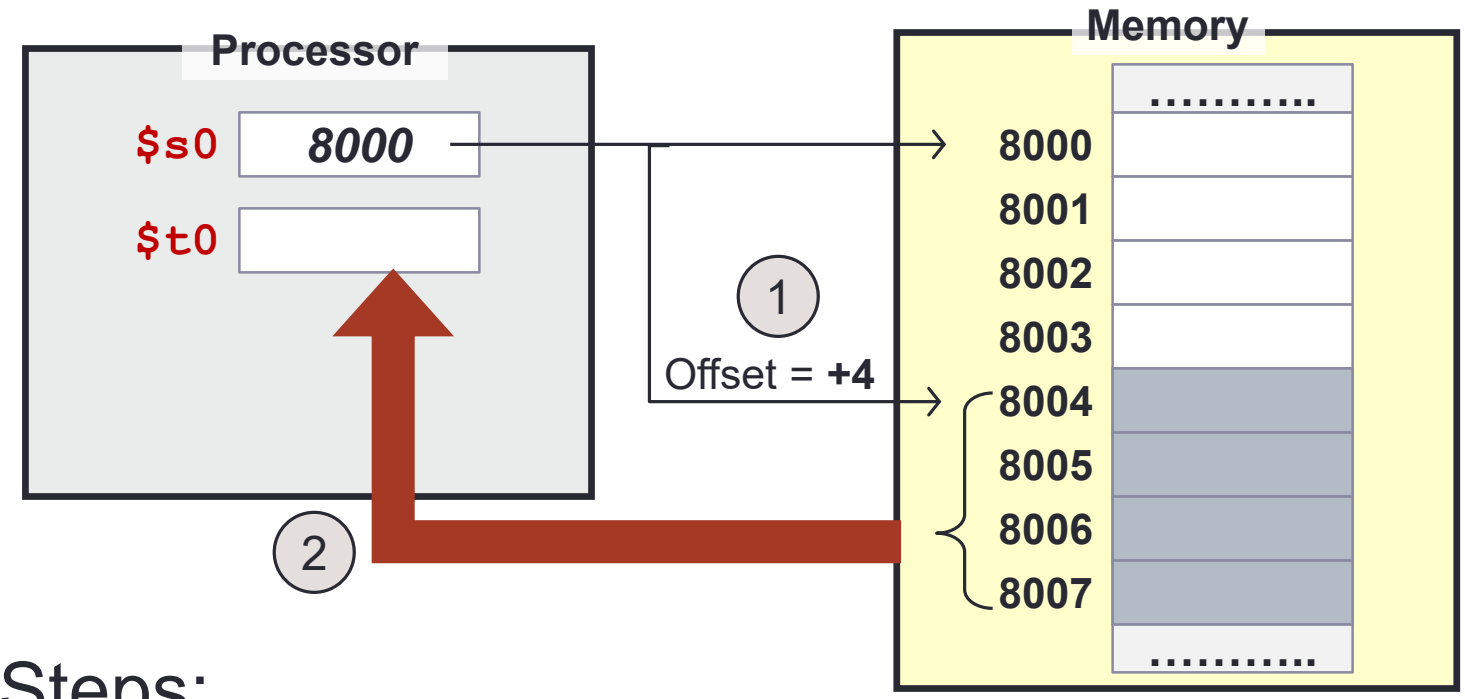
- MIPS is a load-store register architecture
  - 32 registers, each 32-bit (4-byte) long
  - Each word contains 32 bits (4 bytes)
  - Memory addresses are 32-bit long

NOTE:
Magic number is 32

Name	Examples	Comments
32 registers	$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at	Fast processor storage for data. In MIPS, data must be in registers to perform arithmetic.
2^{30} memory words	**Mem[0],** **Mem[4],** …, **Mem[4294967292]**	Accessed only by data transfer instructions. MIPS uses **byte addresses**, so consecutive words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

2.1 Memory Instruction: **Load Word**

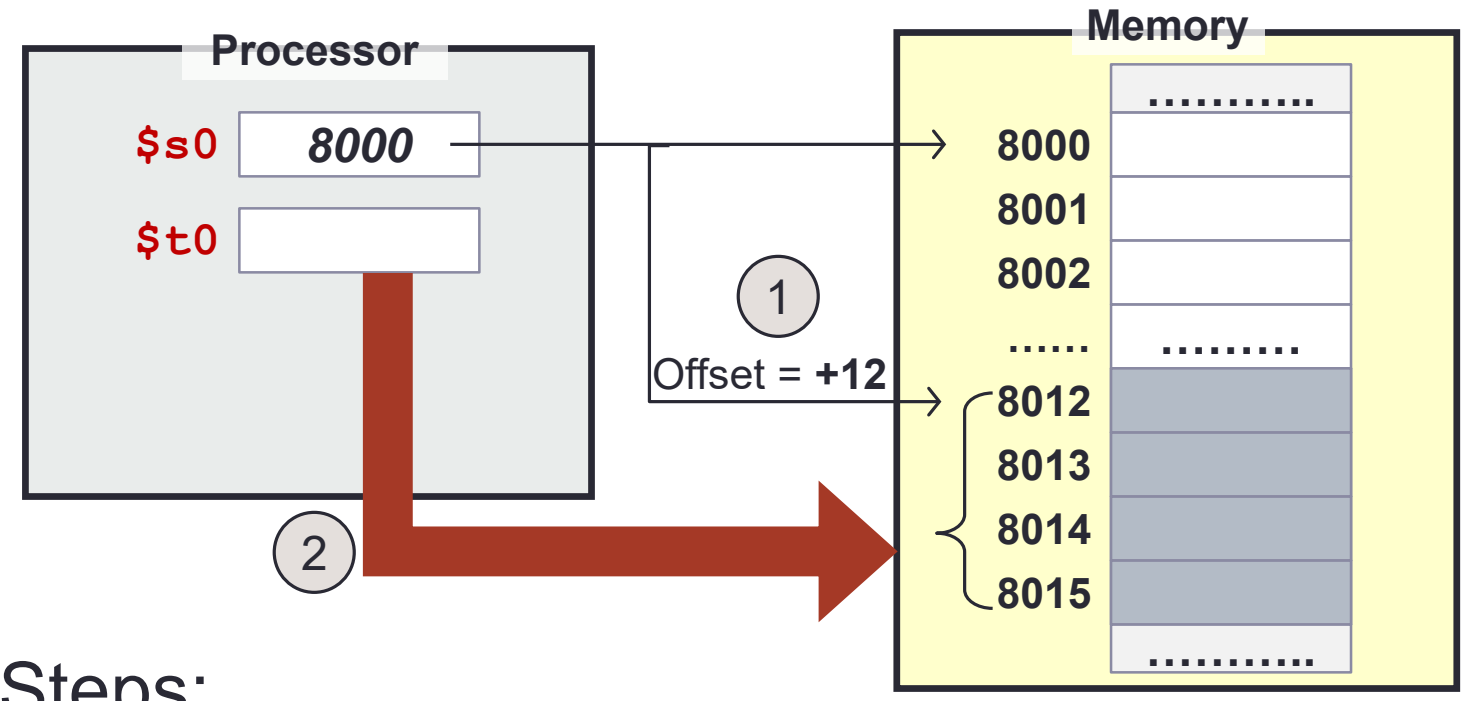
- Example: `lw $t0, 4($s0)`



- Steps:
  1. Memory Address = `$s0` + 4 = 8000 + 4 = **8004**
  2. Memory word at `Mem[8004]` is loaded into `$t0`

2.2 Memory Instruction: **Store Word**

- Example: `sw $t0, 12($s0)`



- Steps:
  1. Memory Address = `$s0` + 12 = 8000 + 12 = **8012**
  2. Content of `$t0` is stored into word at `Mem[8012]`

2.3 **Load** and **Store** Instructions

- Only **`load`** and **`store`** instructions can access data in memory.
- Example: Each array element occupies a word.

C Code	MIPS Code

`A[7] = h + A[10];`
 	
`t0 = A[10];`
`t0 = h + t0;`
`A[7] = t0;`
	`lw $t0, 40($s3)`
	`add $t0, $s2, $t0`
	`sw $t0, 28($s3)`

  - Each array element occupies a word (4 bytes).
  - **`$s3`** contains the **base address** (address of first element, A[0]) of array A. Variable **`h`** is mapped to **`$s2`**.
- Remember arithmetic operands (for **`add`**) are registers, not memory!

2.4 Memory Instructions: Others (1/2)

- Other than load word (**`lw`**) and store word (**`sw`**), there are other variants, example:
  - load byte (**`lb`**)
  - store byte (**`sb`**)
- Similar in format:

```
lb   $t1, 12($s3)
sb   $t2, 13($s3)
```

- Similar in working except that one byte, instead of one word, is loaded or stored
  - Note that the offset no longer needs to be a multiple of 4

2.4 Memory Instructions: Others (2/2)

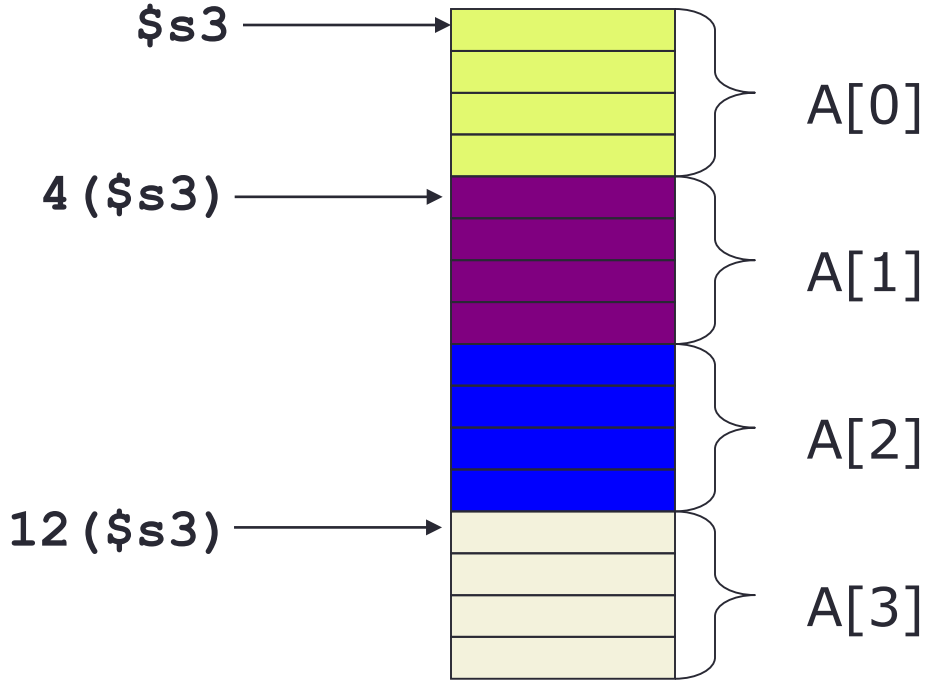
- MIPS disallows loading/storing unaligned word using **`lw`**/**`sw`**:
  - Pseudo-Instructions ***unaligned load word*** (**`ulw`**) and ***unaligned store word*** (**`usw`**) are provided for this purpose
- Other memory instructions:
  - **`lh`** and **`sh`**: load halfword and store halfword
  - **`lwl`**, **`lwr`**, **`swl`**, **`swr`**: load word left / right, store word left / right.
  - etc…

NOTE:
ulw/usw can be translated to sequence of lb/sb + other operations

2.5 Example: Array (assume 4 bytes per element)

C Statement to translate	Variables Mapping
`A[3] = h + A[1];`	`h` ➔ `$s2` base of `A[]` ➔ `$s3`

```
lw   $t0, 4($s3)
add  $t0, $s2, $t0
sw   $t0, 12($s3)
```



2.6 Common Questions: **Address** vs **Value**

Key concept:

Registers do NOT have types

- A register can hold any 32-bit number:
  - The number has no implicit data type and is interpreted according to the instruction that uses it
- Examples:
  - `add $t2, $t1, $t0`
    - ➔ `$t0` and `$t1` should contain data values
  - `lw $t2, 0($t0)`
    - ➔ `$t0` should contain a memory address

2.6 Common Questions: **Byte** vs **Word**

Important:

Consecutive word addresses in machines with byte-addressing do not differ by 1

- Common error:
  - Assume that the address of the next word can be found by incrementing the address in a register by 1 instead of by the word size in bytes
- For both `lw` and `sw`:
  - The sum of base address and offset must be a multiple of 4 (i.e. to adhere to word boundary)

2.7 Example: Swapping Elements

C Statement to translate	Variables Mapping

`swap( int v[], int k )`
`{`
`  int temp;`
`  temp = v[k]`
`  v[k] = v[k+1];`
`  v[k+1] = temp;`
`}`
	`k` ➔ `$5`
	Base address of `v[]` ➔ `$4`
	`temp` ➔ `$15`

```
swap:
    sll   $2, $5, 2
    add   $2, $4, $2
    lw    $15, 0($2)
    lw    $16, 4($2)
    sw    $16, 0($2)
    sw    $15, 4($2)
```

Example: k = 3; to swap v[3] with v[4].
Assume base address of v is 2000.

`$5` (k) ← 3
`$4` (base addr. of v) ← 2000

`$2` ← 12
`$2` ← 2012

`$15` ← content of mem. addr. 2012 (v[3])
`$16` ← content of mem. addr. 2016 (v[4])

content of mem. addr. 2012 (v[3]) ← `$16`
content of mem. addr. 2016 (v[4]) ← `$15`

Note: This is simplified and may not be a direct translation of the C code.

Reading

- Instructions: Language of the Computer
  - Read up COD Chapter 2, pages 52-57. (3rd edition)
  - Read up COD Section 2.3 (4th edition)

3. Making Decisions (1/2)

- We cover only sequential execution so far:
  - Instruction is executed in program order
- To perform general computing tasks, we need to:
  - Make decisions
  - Perform iterations (in later section)
- Decisions making in high-level language:
  - `if` and `goto` statements
  - MIPS decision making instructions are similar to `if` statement with a `goto`
    - `goto` is discouraged in high-level languages but necessary in assembly ☺

3. Making Decisions (2/2)

- Decision-making instructions
  - Alter the control flow of the program
  - Change the next instruction to be executed
- Two types of decision-making statements in MIPS
  - **Conditional** (branch)

```
bne $t0, $t1, label
beq $t0, $t1, label
```

  - **Unconditional** (jump)

```
j label
```

NOTE:
Selection: branch/jump down
Repetition: branch/jump up

In both cases, it's a "goto" operation.

- A label is an "anchor" in the assembly code to indicate point of interest, usually as branch target
  - Labels are NOT instructions!

3.1 Conditional Branch: **beq** and **bne**

- Processor follows the branch only when the condition is satisfied (true)
- **`beq $r1, $r2, L1`**
  - Go to statement labeled `L1` if the value in register `$r1` equals the value in register `$r2`
  - `beq` is "**branch if equal**"
  - C code: `if (a == b) goto L1`
- **`bne $r1, $r2, L1`**
  - Go to statement labeled `L1` if the value in register `$r1` does not equal the value in register `$r2`
  - `bne` is "**branch if not equal**"
  - C code: `if (a != b) goto L1`

3.2 Unconditional Jump: j

- Processor **always** follows the branch
- `j L1`
  - Jump to label `L1` unconditionally
  - C code: `goto L1`
- Technically equivalent to such statement

```
beq $s0, $s0, L1
```

NOTE:
The general technique of compiling from C to MIPS with selection/repetition is to first remove the construct and replace it with either one of these:

```
if (x == y) goto L;  // this is beq
if (x != y) goto L;  // this is bne
```

Some tricks can be used (e.g., inversion) to generate fewer lines of codes.

3.3 IF statement (1/2)

C Statement to translate	Variables Mapping

`if (i == j)`
`  f = g + h;`
	`f → $s0`
	`g → $s1`
	`h → $s2`
	`i → $s3`
	`j → $s4`

```
      beq $s3, $s4, L1
      j   Exit
L1:   add $s0, $s1, $s2
Exit:
```

```
      bne $s3, $s4, Exit
      add $s0, $s1, $s2
Exit:
```

- Two equivalent translations:
  - The one on the right is more efficient

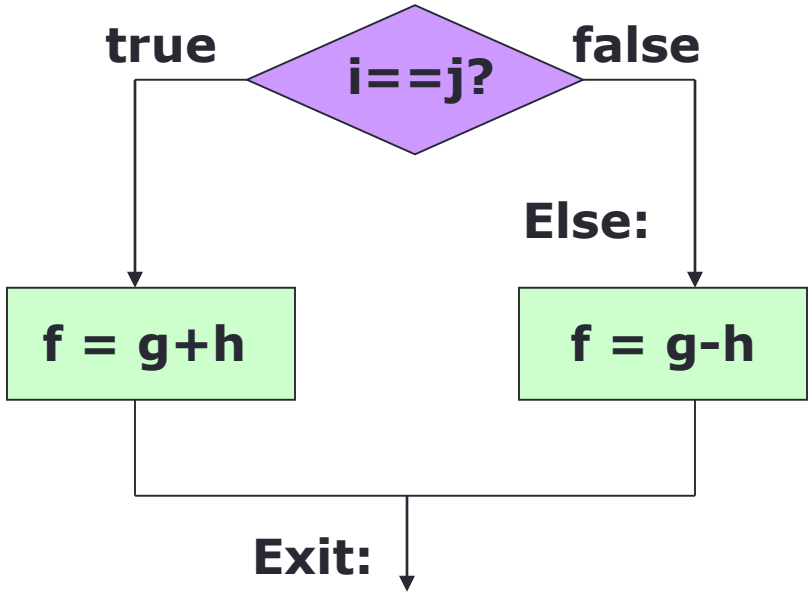
Common technique: Invert the condition for shorter code

3.3 IF statement (2/2)

C Statement to translate	Variables Mapping

`if (i == j)`
`  f = g + h;`
`else`
`  f = g - h;`
	`f` → `$s0`
	`g` → `$s1`
	`h` → `$s2`
	`i` → `$s3`
	`j` → `$s4`

```
      bne $s3, $s4, Else
      add $s0, $s1, $s2
      j   Exit
Else: sub $s0, $s1, $s2
Exit:
```



- Question: Rewrite with **`beq`**?

3.4 Exercise #1: IF statement

MIPS code to translate into C	Variables Mapping

`beq $s1, $s2, Exit`
`add $s0, $zero, $zero`
`Exit:`
	`f` → `$s0`
	`i` → `$s1`
	`j` → `$s2`

- What is the corresponding high-level statement?

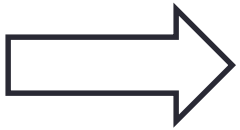
```
if (i != j) {
    f = 0;
}
```

NOTE:
Remember the inversion? Also works in "reverse" (or decompilation) process. beq becomes if (i != j).

4. Loops (1/2)

- C while-loop:

```
while (j == k)
    i = i + 1;
```



- Rewritten with goto

```
Loop:   if (j != k)
            goto Exit;
        i = i+1;
        goto Loop;

Exit:
```

Key concept:

Any form of loop can be written in assembly with the help of conditional branches and jumps.

4. Loops (2/2)

C Statement to translate	Variables Mapping

`Loop: if (j != k)`
`goto Exit;`
`i = i+1;`
`goto Loop;`
 	
`Exit:`
	`i` ➔ `$s3`
	`j` ➔ `$s4`
	`k` ➔ `$s5`

NOTE:
This shows the process clearly:
1. Convert from while to if(…) goto
2. Convert from there to MIPS

- What is the corresponding MIPS code?

```
Loop: bne  $s4, $s5, Exit   # if (j!= k) Exit
      addi $s3, $s3, 1
      j    Loop             # repeat loop
Exit:
```

4.1 Exercise #2: FOR loop

- Write the following loop statement in MIPS

C Statement to translate	Variables Mapping
`for ( i=0; i<10; i++)` `  a = a + 5;`	`i → $s0` `a → $s2`

```
        add  $s0, $zero, $zero
        addi $s1, $zero, 10
Loop:   beq  $s0, $s1, Exit
        addi $s2, $s2, 5
        addi $s0, $s0, 1
        j    Loop
Exit:
```

NOTE:
Alternatively, if you know how to compile while-loop, then you can translate the for-loop into:

```
i = 0;
while (i < 10) {
  a = a + 5;
  i++;
}
```

4.2 Inequalities (1/2)

- We have **`beq`** and **`bne`**, what about branch-if-less-than?
  - There is no real **`blt`** instruction in MIPS
- Use **`slt`** (set on less than) or **`slti`**.

```
slt $t0, $s1, $s2
```

=

```
if ($s1 < $s2)
    $t0 = 1;
else
    $t0 = 0;
```

NOTE:
If we use the "ternary operator" discussed in tutorial, then:

```
$t0 = ($s1 < $s2) ? 1 : 0;
```

4.2 Inequalities (1/2)

- To build a "`blt $s1, $s2, L`" instruction:

```
slt $t0, $s1, $s2
bne $t0, $zero, L
```

==

```
if ($s1 < $s2)
    goto L;
```

- This is another example of **pseudo-instruction**:
  - Assembler translates (`blt`) instruction in an assembly program into the equivalent MIPS (two) instructions

Reading

- Instructions: Language of the Computer
  - Section 2.6 Instructions for Making Decisions. (3rd edition)
  - Section 2.7 Instructions for Making Decisions. (4th edition)

NOTE:
If we know how to compile:

```
if (a < b) ...
```

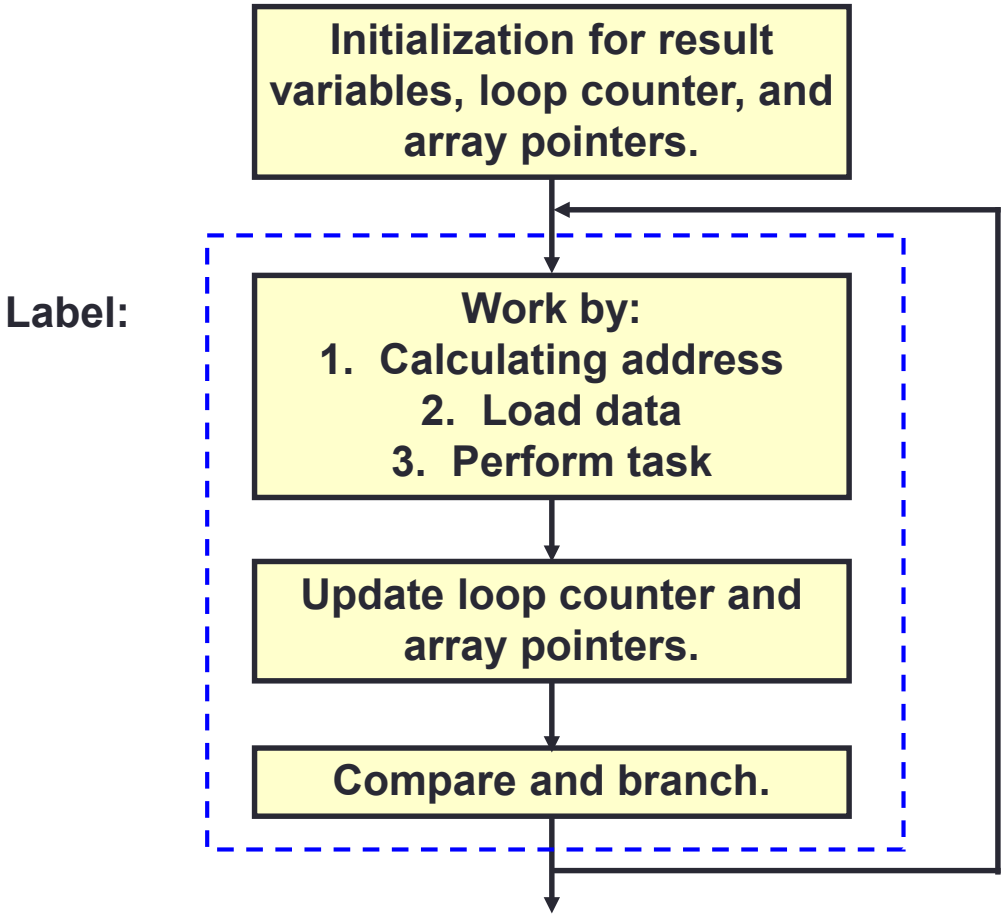
Then we can compile:

```
if (a < b)  → if (b < a)
if (a >= b) → if (!(a < b))
if (a <= b) → if (b >= a) → if(!(b < a))
```

So, with an addition of slt, we can do any other kind of inequalities.

5. Array and Loop

- Typical example of accessing array elements in a loop:



5. Array and Loop: Question

Count the number of zeros in an Array **A**

- **A** is word array with 40 elements
- Address of A[] ➔ **`$t0`**, Result ➔ **`$t8`**

Simple C Code

```
result = 0;

i = 0;

while ( i < 40 ) {
    if ( A[i] == 0 )
        result++;
    i++;

}
```

- Think about:
  - How to perform the right comparison
  - How to translate A[i] correctly

5. Array and Loop: Version 1.0

Address of A[] ➔ $t0 Result ➔ $t8 i ➔ $t1	Comments
`addi $t8, $zero, 0`	
`addi $t1, $zero, 0`	
`addi $t2, $zero, 40`	`# end point`
`loop: bge $t1, $t2, end`	
`sll $t3, $t1, 2`	`# i * 4`
`add $t4, $t0, $t3`	`# $t4 ← &A[i]`
`lw $t5, 0($t4)`	`# $t5 ← A[i]`
`bne $t5, $zero, skip`	
`addi $t8, $t8, 1`	`# result++`
`skip: addi $t1, $t1, 1`	`# i++`
`j loop`	
`end:`	

5. Array and Loop: Version 2.0

Address of A[] ➔ $t0
Result ➔ $t8
&A[i] ➔ $t1

```
        addi $t8, $zero, 0
        addi $t1, $t0, 0
        addi $t2, $t0, 160
loop:   bge  $t1, $t2, end
        lw   $t3, 0($t1)
        bne  $t3, $zero, skip
        addi $t8, $t8, 1
skip:   addi $t1, $t1, 4
        j loop
end:
```

Comments

NOTE:
Consider the code using pointer arith:

```
result = 0;
ptr = A;
end = &A[40];
while ( ptr < end ) {
    if ( *ptr == 0 )
        result++;
    ptr++;
}
```

The resulting MIPS looks like the code on the left.

Use of “pointers” can produce more efficient code!

6.1 Exercise #3: Simple Loop

- Given the following MIPS code:

```
        addi $t1, $zero, 10
        add  $t1, $t1, $t1
        addi $t2, $zero, 10
Loop:   addi $t2, $t2, 10
        addi $t1, $t1, -1
        beq  $t1, $zero, Loop
```

NOTE:
You may want to "decompile" into C so that you can reason in C rather than in MIPS.
Alternatively, you may just "trace" but it may take longer time.
So try to get familiar with a few "compilation" pattern for a few C construct (*e.g., while, do-while, for, if, if-else, switch-case, etc*).

i. How many instructions are executed? **Answer: (a)**

(a) 6 (b) 30 (c) 33 (d) 36 (e) None of the above

ii. What is the final value in `$t2`? **Answer: (b)**

(a) 10 (b) 20 (c) 300 (d) 310 (e) None of the above

6.2 Exercise #4: Simple Loop II

- Given the following MIPS code:

```
        add  $t0, $zero, $zero
        add  $t1, $t0, $t0
        addi $t2, $t1, 4
Again:  add  $t1, $t1, $t0
        addi $t0, $t0, 1
        bne  $t2, $t0, Again
```

i. How many instructions are executed? **Answer: (c)**

(a) 6 (b) 12 (c) 15 (d) 18 (e) None of the above

ii. What is the final value in `$t1`? **Answer: (c)**

(a) 0 (b) 4 (c) 6 (d) 10 (e) None of the above

6.3 Exercise #5: Simple Loop III (1/2)

- Given the following MIPS code accessing a word array of elements in memory with the starting address in **`$t0`**.

```
        addi $t1, $t0, 10
        add  $t2, $zero, $zero
Loop:   ulw  $t3, 0($t1) # ulw: unaligned lw
        add  $t2, $t2, $t3
        addi $t1, $t1, -1
        bne  $t1, $t0, Loop
```

i. How many times is the **`bne`** instruction executed?

(a) 1 (b) 3 (c) 9 (d) 10 (e) 11 **Answer: (d)**

ii. How many times does the **`bne`** instruction actually branch to the label **`Loop`**?

(a) 1 (b) 8 (c) 9 (d) 10 (e) 11 **Answer: (c)**

6.3 Exercise #5: Simple Loop III (2/2)

- Given the following MIPS code accessing a word array of elements in memory with the starting address in **`$t0`**.

```
        addi $t1, $t0, 10
        add  $t2, $zero, $zero
Loop:   ulw  $t3, 0($t1) # ulw: unaligned lw
        add  $t2, $t2, $t3
        addi $t1, $t1, -1
        bne  $t1, $t0, Loop
```

iii. How many instructions are executed?

(a) 6 (b) 12 (c) 41 (d) 42 (e) 46 **Answer: (d)**

iv. How many unique bytes of data are read from the memory?

(a) 4 (b) 10 (c) 11 (d) 13 (e) 40 **Answer: (d)**

Summary: Focus of CS2100

- Basic MIPS programming
  - Arithmetic: among registers only
  - Handling of large constants
  - Memory accesses: load/store
  - Control flow: branch and jump
  - Accessing array elements
  - System calls (covered in labs)
- Things we are not going to cover
  - Support for procedures
  - Linkers, loaders, memory layout
  - Stacks, frames, recursion
  - Interrupts and exceptions

End of File

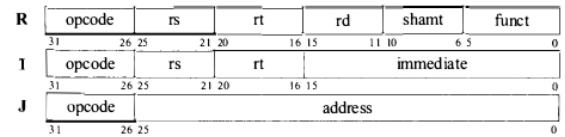
MIPS Reference Data

CORE INSTRUCTION SET

NAME, MNEMONIC		FOR-MAT	OPERATION (in Verilog)		OPCODE / FUNCT (Hex)
Add	add	R	R[rd] = R[rs] + R[rt]	(1)	0 / 20_{hex}
Add Immediate	addi	I	R[rt] = R[rs] + SignExtImm	(1,2)	8_{hex}
Add Imm. Unsigned	addiu	I	R[rt] = R[rs] + SignExtImm	(2)	9_{hex}
Add Unsigned	addu	R	R[rd] = R[rs] + R[rt]		0 / 21_{hex}
And	and	R	R[rd] = R[rs] & R[rt]		0 / 24_{hex}
And Immediate	andi	I	R[rt] = R[rs] & ZeroExtImm	(3)	c_{hex}
Branch On Equal	beq	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr	(4)	4_{hex}
Branch On Not Equal	bne	I	if(R[rs]!=R[rt]) PC=PC+4+BranchAddr	(4)	5_{hex}
Jump	j	J	PC=JumpAddr	(5)	2_{hex}
Jump And Link	jal	J	R[31]=PC+8;PC=JumpAddr	(5)	3_{hex}
Jump Register	jr	R	PC=R[rs]		0 / 08_{hex}
Load Byte Unsigned	lbu	I	R[rt]={24'b0,M[R[rs]+SignExtImm](7:0)}	(2)	24_{hex}
Load Halfword Unsigned	lhu	I	R[rt]={16'b0,M[R[rs]+SignExtImm](15:0)}	(2)	25_{hex}
Load Linked	ll	I	R[rt] = M[R[rs]+SignExtImm]	(2,7)	30_{hex}
Load Upper Imm.	lui	I	R[rt] = {imm, 16'b0}		f_{hex}
Load Word	lw	I	R[rt] = M[R[rs]+SignExtImm]	(2)	23_{hex}
Nor	nor	R	R[rd] = ~ (R[rs] | R[rt])		0 / 27_{hex}
Or	or	R	R[rd] = R[rs] | R[rt]		0 / 25_{hex}
Or Immediate	ori	I	R[rt] = R[rs] | ZeroExtImm	(3)	d_{hex}
Set Less Than	slt	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0		0 / $2a_{hex}$
Set Less Than Imm.	slti	I	R[rt] = (R[rs] < SignExtImm)? 1 : 0	(2)	a_{hex}
Set Less Than Imm. Unsigned	sltiu	I	R[rt] = (R[rs] < SignExtImm) ? 1 : 0	(2,6)	b_{hex}
Set Less Than Unsig.	sltu	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0	(6)	0 / $2b_{hex}$
Shift Left Logical	sll	R	R[rd] = R[rt] << shamt		0 / 00_{hex}
Shift Right Logical	srl	R	R[rd] = R[rt] >> shamt		0 / 02_{hex}
Store Byte	sb	I	M[R[rs]+SignExtImm](7:0) = R[rt](7:0)	(2)	28_{hex}
Store Conditional	sc	I	M[R[rs]+SignExtImm] = R[rt]; R[rt] = (*atomic*) ? 1 : 0	(2,7)	38_{hex}
Store Halfword	sh	I	M[R[rs]+SignExtImm](15:0) = R[rt](15:0)	(2)	29_{hex}
Store Word	sw	I	M[R[rs]+SignExtImm] = R[rt]	(2)	$2b_{hex}$
Subtract	sub	R	R[rd] = R[rs] - R[rt]	(1)	0 / 22_{hex}
Subtract Unsigned	subu	R	R[rd] = R[rs] - R[rt]		0 / 23_{hex}

(1) May cause overflow exception
(2) SignExtImm = { 16{immediate[15]}, immediate }
(3) ZeroExtImm = { 16{1b'0}, immediate }
(4) BranchAddr = { 14{immediate[15]}, immediate, 2'b0 }
(5) JumpAddr = { PC+4[31:28], address, 2'b0 }
(6) Operands considered unsigned numbers (vs. 2's comp.)
(7) Atomic test&set pair; R[rt] = 1 if pair atomic, 0 if not atomic

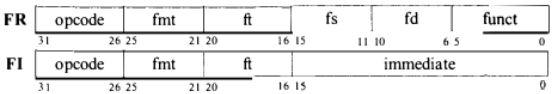
BASIC INSTRUCTION FORMATS



ARITHMETIC CORE INSTRUCTION SET

NAME, MNEMONIC		FOR-MAT	OPERATION		OPCODE / FMT /FT / FUNCT (Hex)
Branch On FP True	bc1t	FI	if(FPcond)PC=PC+4+BranchAddr	(4)	11/8/1/--
Branch On FP False	bc1f	FI	if(!FPcond)PC=PC+4+BranchAddr	(4)	11/8/0/--
Divide	div	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]		0/--/--/1a
Divide Unsigned	divu	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]	(6)	0/--/--/1b
FP Add Single	add.s	FR	F[fd]= F[fs] + F[ft]		11/10/--/0
FP Add Double	add.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} + {F[ft],F[ft+1]}		11/11/--/0
FP Compare Single	c.x.s*	FR	FPcond = (F[fs] *op* F[ft]) ? 1 : 0		11/10/--/*y*
FP Compare Double	c.x.d*	FR	FPcond = ({F[fs],F[fs+1]} *op* {F[ft],F[ft+1]}) ? 1 : 0		11/11/--/*y*
* (*x* is eq, lt, or le) (*op* is ==, <, or <=) (*y* is 32, 3c, or 3e)					
FP Divide Single	div.s	FR	F[fd] = F[fs] / F[ft]		11/10/--/3
FP Divide Double	div.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} / {F[ft],F[ft+1]}		11/11/--/3
FP Multiply Single	mul.s	FR	F[fd] = F[fs] * F[ft]		11/10/--/2
FP Multiply Double	mul.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} * {F[ft],F[ft+1]}		11/11/--/2
FP Subtract Single	sub.s	FR	F[fd]=F[fs] - F[ft]		11/10/--/1
FP Subtract Double	sub.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} - {F[ft],F[ft+1]}		11/11/--/1
Load FP Single	lwc1	I	F[rt]=M[R[rs]+SignExtImm]	(2)	31/--/--/--
Load FP Double	ldc1	I	F[rt]=M[R[rs]+SignExtImm]; F[rt+1]=M[R[rs]+SignExtImm+4]	(2)	35/--/--/--
Move From Hi	mfhi	R	R[rd] = Hi		0 /--/--/10
Move From Lo	mflo	R	R[rd] = Lo		0 /--/--/12
Move From Control	mfc0	R	R[rd] = CR[rs]		10 /0/--/0
Multiply	mult	R	{Hi,Lo} = R[rs] * R[rt]		0/--/--/18
Multiply Unsigned	multu	R	{Hi,Lo} = R[rs] * R[rt]	(6)	0/--/--/19
Shift Right Arith.	sra	R	R[rd] = R[rt] >>> shamt		0/--/--/3
Store FP Single	swc1	I	M[R[rs]+SignExtImm] = F[rt]	(2)	39/--/--/--
Store FP Double	sdc1	I	M[R[rs]+SignExtImm] = F[rt]; M[R[rs]+SignExtImm+ 4] = F[rt+1]	(2)	3d/--/--/--

FLOATING-POINT INSTRUCTION FORMATS



PSEUDOINSTRUCTION SET

NAME	MNEMONIC	OPERATION
Branch Less Than	blt	if(R[rs]<R[rt]) PC = Label
Branch Greater Than	bgt	if(R[rs]>R[rt]) PC = Label
Branch Less Than or Equal	ble	if(R[rs]<=R[rt]) PC = Label
Branch Greater Than or Equal	bge	if(R[rs]>=R[rt]) PC = Label
Load Immediate	li	R[rd] = immediate
Move	move	R[rd] = R[rs]

REGISTER NAME, NUMBER, USE, CALL CONVENTION

NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
$zero	0	The Constant Value 0	N.A.
$at	1	Assembler Temporary	No
$v0-$vl	2-3	Values for Function Results and Expression Evaluation	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved Temporaries	Yes
$t8-$t9	24-25	Temporaries	No
$k0-$k1	26-27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	Yes

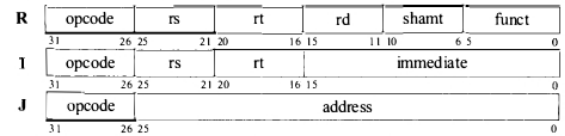
MIPS Reference Data ①

CORE INSTRUCTION SET

NAME, MNEMONIC		FOR-MAT	OPERATION (in Verilog)		OPCODE / FUNCT (Hex)
Add	add	R	R[rd] = R[rs] + R[rt]	(1)	0 / 20_{hex}
Add Immediate	addi	I	R[rt] = R[rs] + SignExtImm	(1,2)	8_{hex}
Add Imm. Unsigned	addiu	I	R[rt] = R[rs] + SignExtImm	(2)	9_{hex}
Add Unsigned	addu	R	R[rd] = R[rs] + R[rt]		0 / 21_{hex}
And	and	R	R[rd] = R[rs] & R[rt]		0 / 24_{hex}
And Immediate	andi	I	R[rt] = R[rs] & ZeroExtImm	(3)	c_{hex}
Branch On Equal	beq	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr	(4)	4_{hex}
Branch On Not Equal	bne	I	if(R[rs]!=R[rt]) PC=PC+4+BranchAddr	(4)	5_{hex}
Jump	j	J	PC=JumpAddr	(5)	2_{hex}
Jump And Link	jal	J	R[31]=PC+8;PC=JumpAddr	(5)	3_{hex}
Jump Register	jr	R	PC=R[rs]		0 / 08_{hex}
Load Byte Unsigned	lbu	I	R[rt]={24'b0,M[R[rs]+SignExtImm](7:0)}	(2)	24_{hex}
Load Halfword Unsigned	lhu	I	R[rt]={16'b0,M[R[rs]+SignExtImm](15:0)}	(2)	25_{hex}
Load Linked	ll	I	R[rt] = M[R[rs]+SignExtImm]	(2,7)	30_{hex}
Load Upper Imm.	lui	I	R[rt] = {imm, 16'b0}		f_{hex}
Load Word	lw	I	R[rt] = M[R[rs]+SignExtImm]	(2)	23_{hex}
Nor	nor	R	R[rd] = ~ (R[rs] | R[rt])		0 / 27_{hex}
Or	or	R	R[rd] = R[rs] | R[rt]		0 / 25_{hex}
Or Immediate	ori	I	R[rt] = R[rs] | ZeroExtImm	(3)	d_{hex}
Set Less Than	slt	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0		0 / $2a_{hex}$
Set Less Than Imm.	slti	I	R[rt] = (R[rs] < SignExtImm)? 1 : 0	(2)	a_{hex}
Set Less Than Imm. Unsigned	sltiu	I	R[rt] = (R[rs] < SignExtImm) ? 1 : 0	(2,6)	b_{hex}
Set Less Than Unsig.	sltu	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0	(6)	0 / $2b_{hex}$
Shift Left Logical	sll	R	R[rd] = R[rt] << shamt		0 / 00_{hex}
Shift Right Logical	srl	R	R[rd] = R[rt] >> shamt		0 / 02_{hex}
Store Byte	sb	I	M[R[rs]+SignExtImm](7:0) = R[rt](7:0)	(2)	28_{hex}
Store Conditional	sc	I	M[R[rs]+SignExtImm] = R[rt]; R[rt] = (*atomic*) ? 1 : 0	(2,7)	38_{hex}
Store Halfword	sh	I	M[R[rs]+SignExtImm](15:0) = R[rt](15:0)	(2)	29_{hex}
Store Word	sw	I	M[R[rs]+SignExtImm] = R[rt]	(2)	$2b_{hex}$
Subtract	sub	R	R[rd] = R[rs] - R[rt]	(1)	0 / 22_{hex}
Subtract Unsigned	subu	R	R[rd] = R[rs] - R[rt]		0 / 23_{hex}

(1) May cause overflow exception
(2) SignExtImm = { 16{immediate[15]}, immediate }
(3) ZeroExtImm = { 16{1b'0}, immediate }
(4) BranchAddr = { 14{immediate[15]}, immediate, 2'b0 }
(5) JumpAddr = { PC+4[31:28], address, 2'b0 }
(6) Operands considered unsigned numbers (vs. 2's comp.)
(7) Atomic test&set pair; R[rt] = 1 if pair atomic, 0 if not atomic

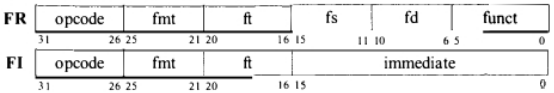
BASIC INSTRUCTION FORMATS



ARITHMETIC CORE INSTRUCTION SET ②

NAME, MNEMONIC		FOR-MAT	OPERATION		OPCODE / FMT /FT / FUNCT (Hex)
Branch On FP True	bc1t	FI	if(FPcond)PC=PC+4+BranchAddr	(4)	11/8/1/--
Branch On FP False	bc1f	FI	if(!FPcond)PC=PC+4+BranchAddr	(4)	11/8/0/--
Divide	div	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]		0/--/--/1a
Divide Unsigned	divu	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]	(6)	0/--/--/1b
FP Add Single	add.s	FR	F[fd]= F[fs] + F[ft]		11/10/--/0
FP Add Double	add.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} + {F[ft],F[ft+1]}		11/11/--/0
FP Compare Single	c.x.s*	FR	FPcond = (F[fs] *op* F[ft]) ? 1 : 0		11/10/--/*y*
FP Compare Double	c.x.d*	FR	FPcond = ({F[fs],F[fs+1]} *op* {F[ft],F[ft+1]}) ? 1 : 0		11/11/--/*y*
* (*x* is eq, lt, or le) (*op* is ==, <, or <=) (*y* is 32, 3c, or 3e)					
FP Divide Single	div.s	FR	F[fd] = F[fs] / F[ft]		11/10/--/3
FP Divide Double	div.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} / {F[ft],F[ft+1]}		11/11/--/3
FP Multiply Single	mul.s	FR	F[fd] = F[fs] * F[ft]		11/10/--/2
FP Multiply Double	mul.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} * {F[ft],F[ft+1]}		11/11/--/2
FP Subtract Single	sub.s	FR	F[fd]=F[fs] - F[ft]		11/10/--/1
FP Subtract Double	sub.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} - {F[ft],F[ft+1]}		11/11/--/1
Load FP Single	lwc1	I	F[rt]=M[R[rs]+SignExtImm]	(2)	31/--/--/--
Load FP Double	ldc1	I	F[rt]=M[R[rs]+SignExtImm]; F[rt+1]=M[R[rs]+SignExtImm+4]	(2)	35/--/--/--
Move From Hi	mfhi	R	R[rd] = Hi		0 /--/--/10
Move From Lo	mflo	R	R[rd] = Lo		0 /--/--/12
Move From Control	mfc0	R	R[rd] = CR[rs]		10 /0/--/0
Multiply	mult	R	{Hi,Lo} = R[rs] * R[rt]		0/--/--/18
Multiply Unsigned	multu	R	{Hi,Lo} = R[rs] * R[rt]	(6)	0/--/--/19
Shift Right Arith.	sra	R	R[rd] = R[rt] >>> shamt		0/--/--/3
Store FP Single	swc1	I	M[R[rs]+SignExtImm] = F[rt]	(2)	39/--/--/--
Store FP Double	sdc1	I	M[R[rs]+SignExtImm] = F[rt]; M[R[rs]+SignExtImm+ 4] = F[rt+1]	(2)	3d/--/--/--

FLOATING-POINT INSTRUCTION FORMATS



PSEUDOINSTRUCTION SET

NAME	MNEMONIC	OPERATION
Branch Less Than	blt	if(R[rs]<R[rt]) PC = Label
Branch Greater Than	bgt	if(R[rs]>R[rt]) PC = Label
Branch Less Than or Equal	ble	if(R[rs]<=R[rt]) PC = Label
Branch Greater Than or Equal	bge	if(R[rs]>=R[rt]) PC = Label
Load Immediate	li	R[rd] = immediate
Move	move	R[rd] = R[rs]

REGISTER NAME, NUMBER, USE, CALL CONVENTION

NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
$zero	0	The Constant Value 0	N.A.
$at	1	Assembler Temporary	No
$v0-$v1	2-3	Values for Function Results and Expression Evaluation	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved Temporaries	Yes
$t8-$t9	24-25	Temporaries	No
$k0-$k1	26-27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	Yes