EECS 314
Computer Architecture

Spring 2018

Chapter 2
Instructions: Language of the Computer

Instructor: Ming-Chun Huang, PhD

ming-chun.huang@case.edu

Office: Glennan 514B

MIPS Memory Access Instructions

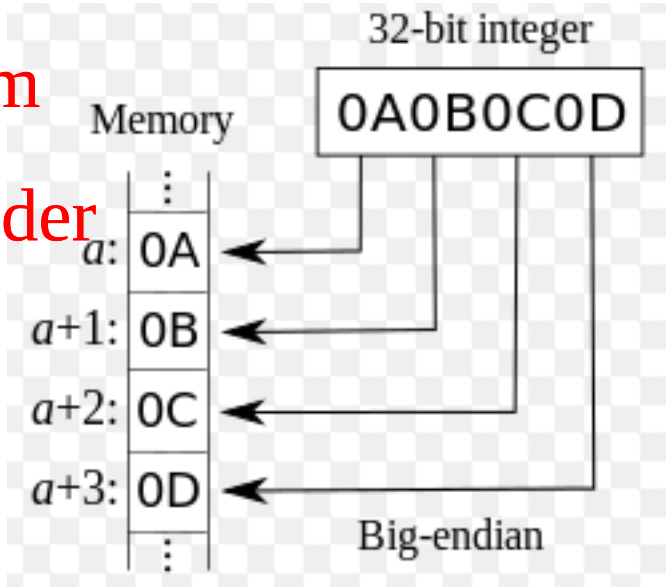
- MIPS has two basic data transfer instructions for accessing memory

```
lw    $t0, 4($s3)   #load word from memory
sw    $t0, 8($s3)   #store word to memory
```


- The data is loaded into (lw) or stored from (sw) a register in the register file – a 5 bit address
- The memory address – a 32 bit address – is formed by adding the contents of the base address register to the offset value
  - A 16-bit field meaning access is limited to memory locations within a region of $\pm 2^{13}$ or 8,192 words ($\pm 2^{15}$ or 32,768 bytes) of the address in the base register
  - Note that the offset can be positive or negative

Memory Operands

- Main memory used for composite data
  - Arrays, structures, dynamic data
- To apply arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory
- Memory is byte addressed
  - Each address identifies an 8-bit byte
- Words are aligned in memory
  - Address must be a multiple of 4
- MIPS is Big Endian
  - Most-significant byte at least address of a word
  - *c.f.* Little Endian: least-significant byte at least address



Memory Operand Example 1

- C code:

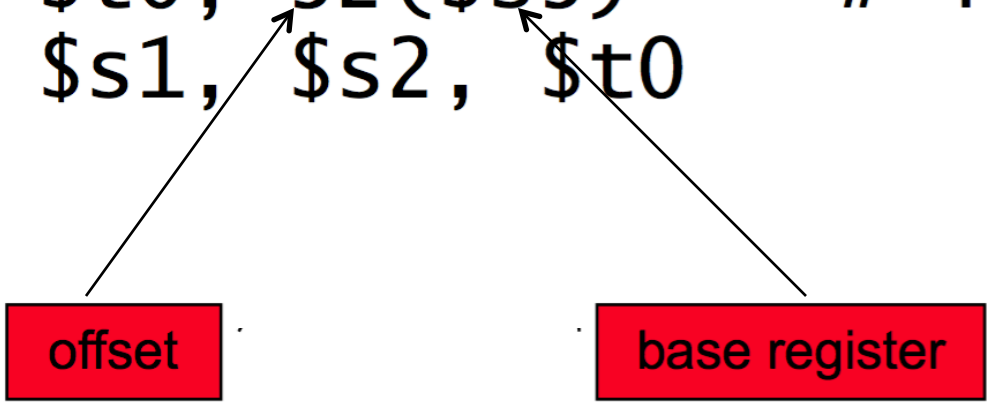
```
g = h + A[8];
```

  - g in $s1, h in $s2, base address of A in $s3

- Compiled MIPS code:

  - Index 8 requires offset of 32
    - 4 bytes per word


```
lw  $t0, 32($s3)    # load word
add $s1, $s2, $t0
```



Memory Operand Example 2

- C code:

```
A[12] = h + A[8];
```

  - h in $s2, base address of A in $s3

- Compiled MIPS code:
  - Index 8 requires offset of 32

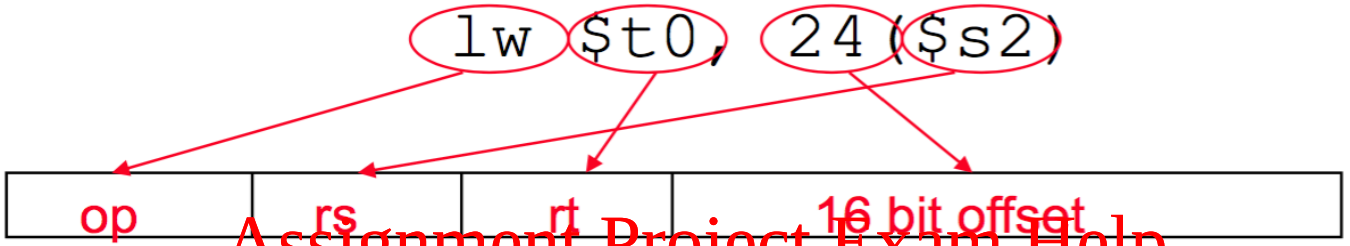
```
lw  $t0, 32($s3)    # load word
add $t0, $s2, $t0
sw  $t0, 48($s3)    # store word
```

Registers vs. Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
  - More instructions to be executed
- Compiler must use registers for variables as much as possible
  - Only spill to memory for less frequently used variables
  - Register optimization is important!

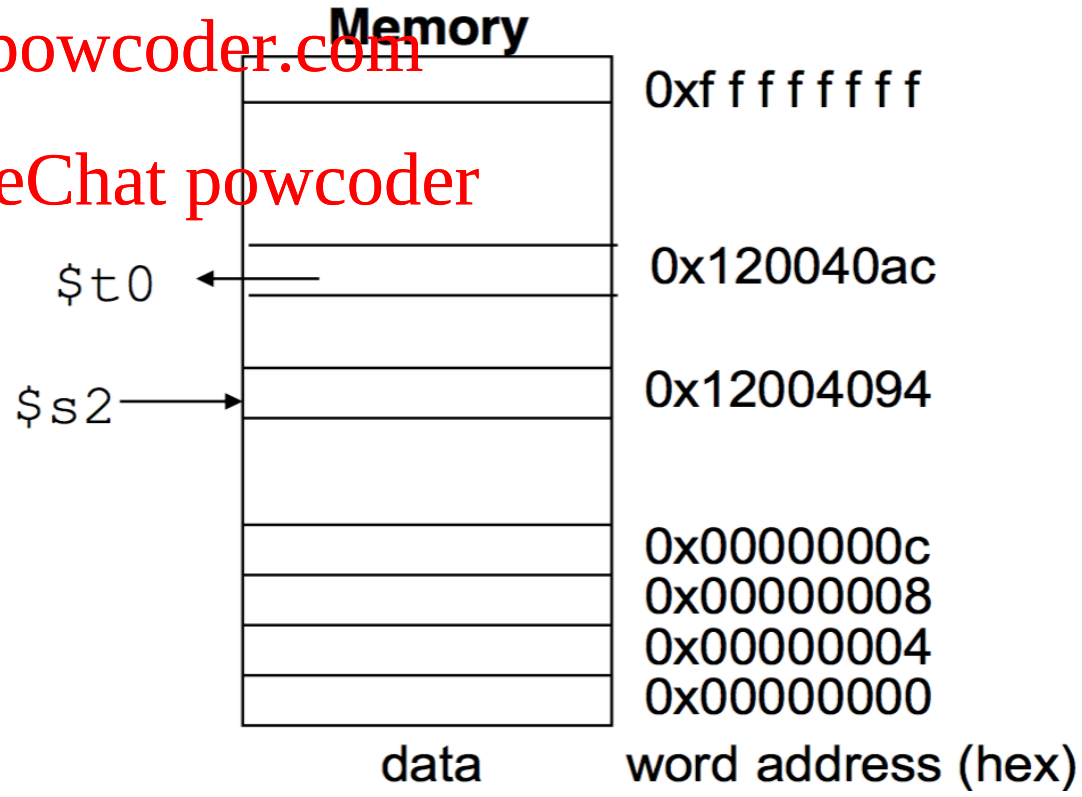
Machine Language - Load Instruction

- Load/Store Instruction Format (I format):



24_{10} + $s2 =

. . . 0001 1000
+ . . . 1001 0100
. . . 1010 1100 =
0x120040ac

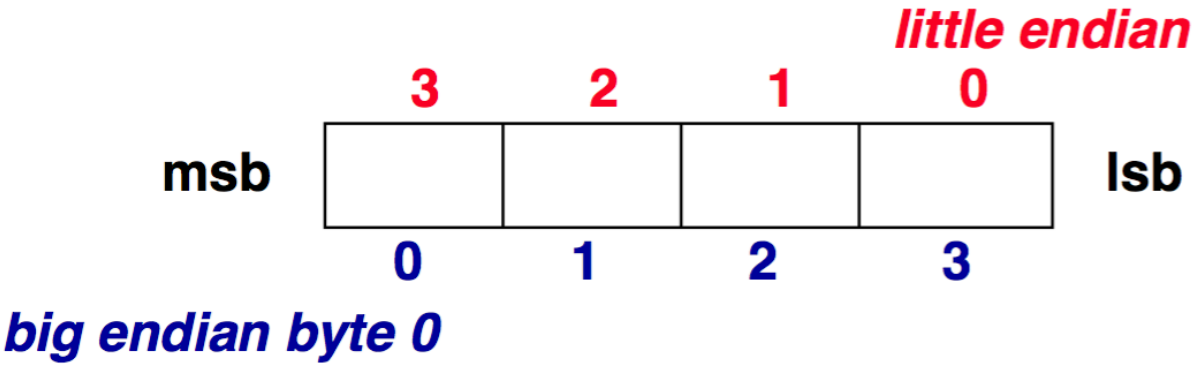


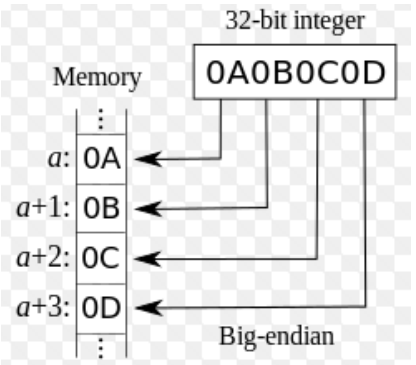
Byte Addresses

- Since 8-bit bytes are so useful, most architectures address individual bytes in memory
  - The memory address of a word must be a multiple of 4 (alignment restriction)
- Big Endian: leftmost byte is word address

  IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
- Little Endian: rightmost byte is word address

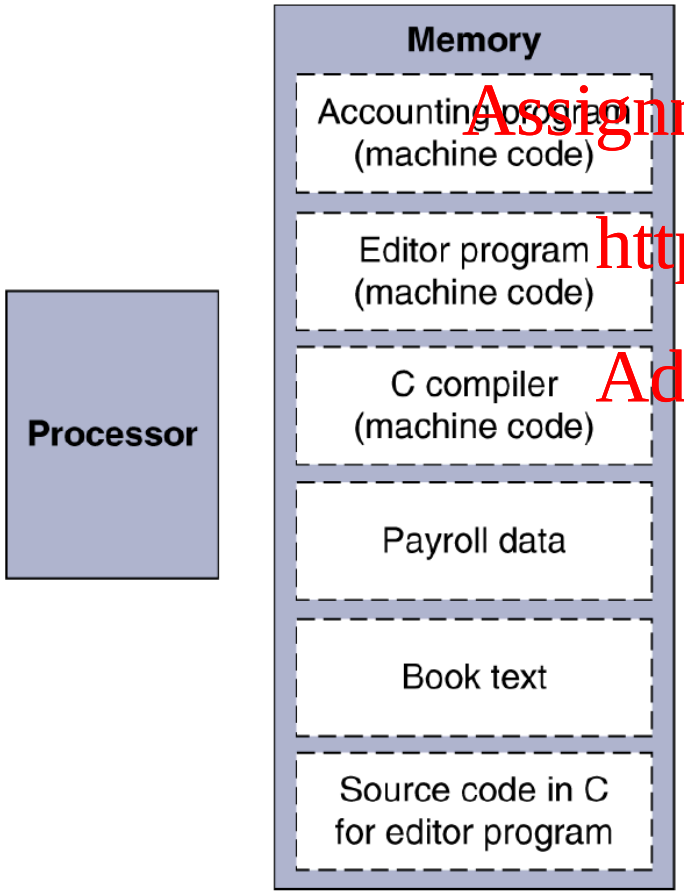
  Intel 80x86, DEC Vax, DEC Alpha (Windows NT)





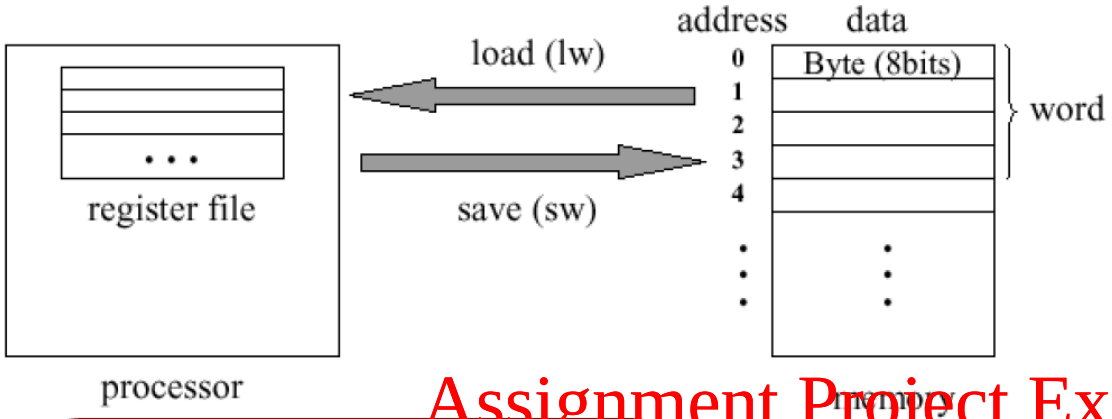
Stored Program Computers

The BIG Picture



- Instructions represented in binary, just like data
- Instructions and data stored in memory
- Programs can operate on programs
  - e.g., compilers, linkers, …
- Binary compatibility allows compiled programs to work on different computers
  - Standardized ISAs

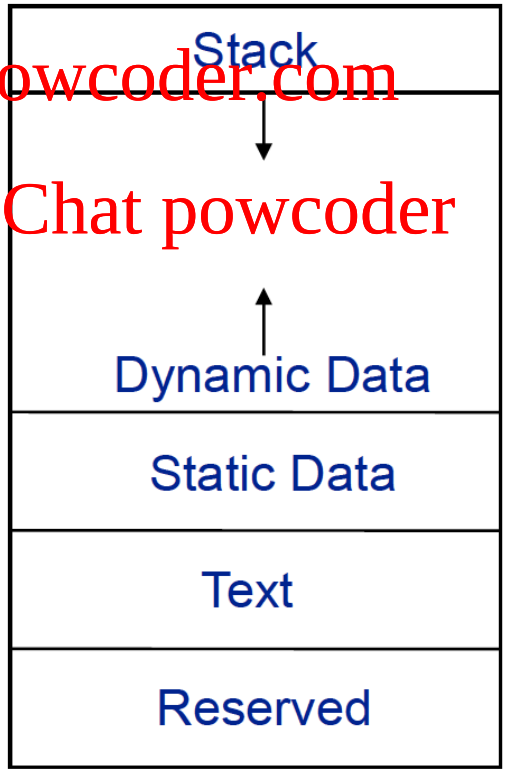
MIPS Memory



Processor Memory Interaction*

- The program code starts at 0x00400000
- The stack starts at top and grows down towards data segments
- The static data starts at 0x00400000. The dynamic data starts right after.
- The $gp helps accessing static data

```
lw $t0, 24($s2)
lw $t0, 24($gp)
```



0xFFFFFFFF

0x7FFFFFFF → SP ($sp)

0x10008000 → GP ($gp)

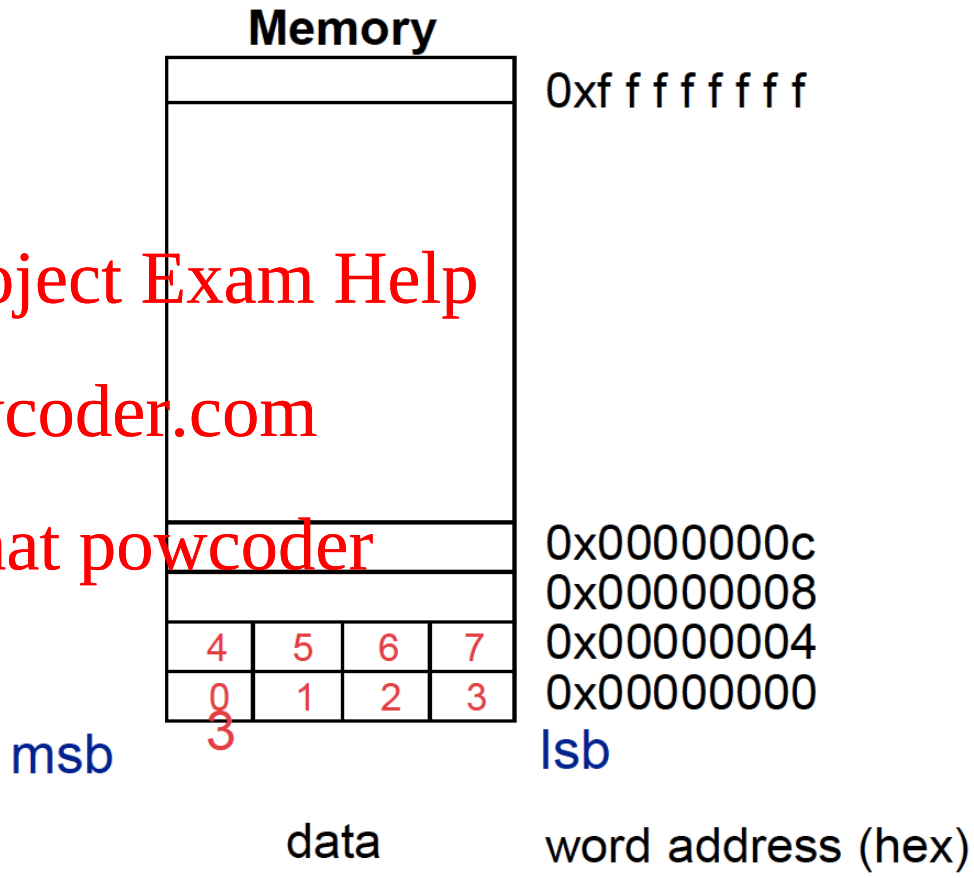
0x00400000

0x00400000 → PC

0x00000000

MIPS Memory

- Each memory word is 32-bit wide (4 bytes)
- Big-Endian
- MIPS memory is
  - Byte-addressable
  - BUT
  - Word-alligned!



Loading and Storing Bytes / halfwords

- MIPS provides special instructions to move bytes

```
lb    $t0, 1($s3)   #load byte from memory
sb    $t0, 6($s3)   #store byte to  memory
```


- What 8 bits get loaded and stored?
  - load byte places the byte from memory in the rightmost 8 bits of the destination register
    - what happens to the other bits in the register?
  - store byte takes the byte from the rightmost 8 bits of a register and writes it to a byte in memory
    - what happens to the other bits in the memory word?

Byte/Halfword Operations

- Could use bitwise operations
- MIPS byte/halfword load/store
  - String processing is a common case

```
lb rt, offset(rs)      lh rt, offset(rs)
```

  - Sign extend to 32 bits in rt

```
lbu rt, offset(rs)     lhu rt, offset(rs)
```

  - Zero extend to 32 bits in rt

```
sb rt, offset(rs)      sh rt, offset(rs)
```

  - Store just rightmost byte/halfword

MIPS Control Flow Instructions

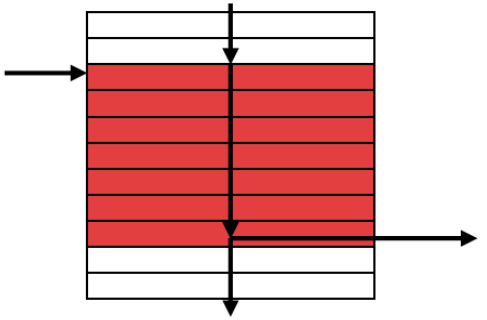
- MIPS conditional branch instructions:

```
bne $s0, $s1, Lbl #go to Lbl if $s0≠$s1
beq $s0, $s1, Lbl #go to Lbl if $s0=$s1
```

- Ex:

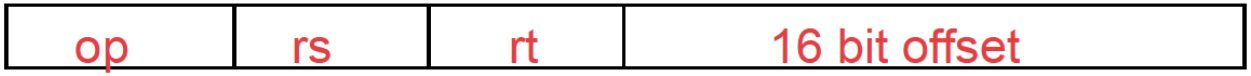
```
if (i==j) h = i + j;
```

```
        bne $s0, $s1, Lbl1
        add $s3, $s0, $s1
Lbl1:   ...
```

A ***basic block*** is a sequence of instructions with
No embedded branches (except at end)
No branch targets (except at beginning)

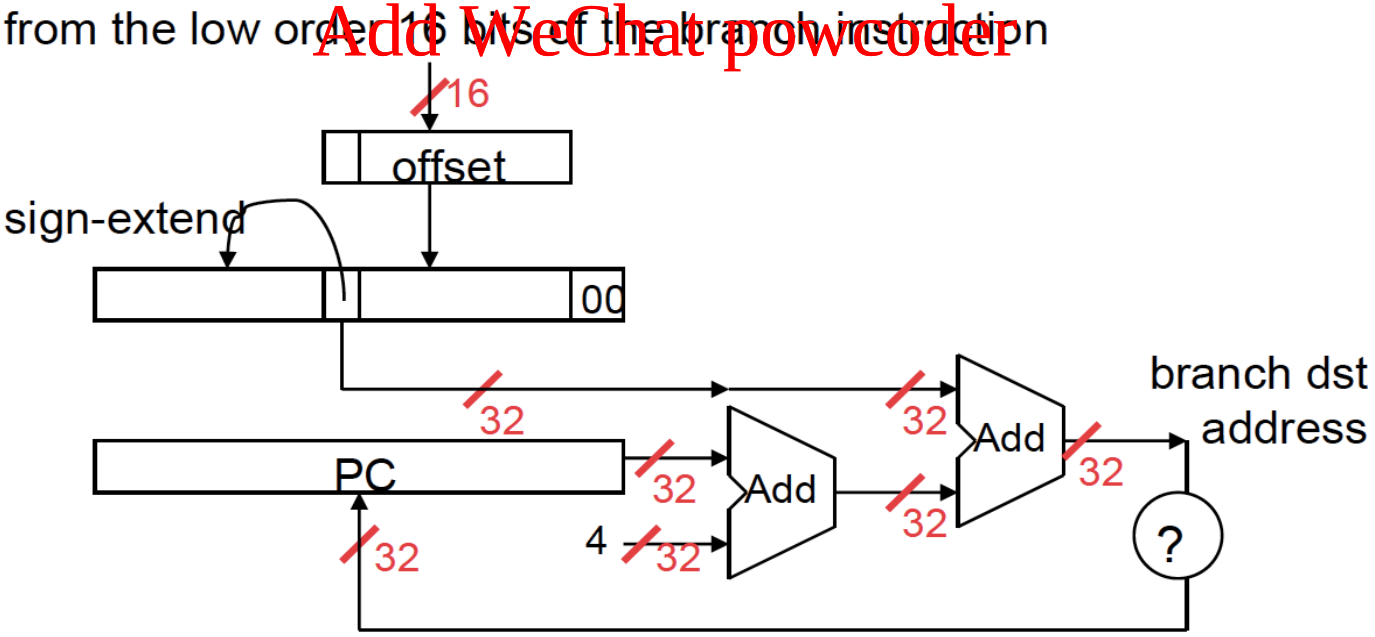
- Instruction Format (I format):



- How is the branch destination address specified?

Specifying Branch Destinations

- Use a register (like in lw and sw) added to the 16-bit offset
  - which register? Instruction Address Register (the PC)
    - its use is automatically implied by instruction
    - PC gets updated (PC+4) during the fetch cycle so that it holds the address of the next instruction
  - limits the branch distance to -2^{15} to $+2^{15}-1$ instructions from the (instruction after the) branch instruction, but most branches are local anyway



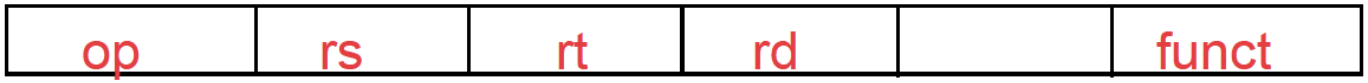
More Branch Instructions

- We have `beq`, `bne`, but what about other kinds of branches (e.g., branch-if-less-than)? For this, we need yet another instruction, `slt`

- Set on less than instruction:


```
slt $t0, $s0, $s1     # if $s0 < $s1     then
                      # $t0 = 1          else
                      # $t0 = 0
```


- Instruction format (R format):



More Branch Instructions, Con't

- Can use `slt`, `beq`, `bne`, and the fixed value of 0 in register `$zero` to create other conditions
  - less than **`blt $s1, $s2, Label`** ⇒

```
slt  $at, $s1, $s2        # $at set to 1 if
bne  $at, $zero, Label    # $s1 < $s2
```

  - less than or equal to `ble $s1, $s2, Label`
  - greater than `bgt $s1, $s2, Label`
  - great than or equal to `bge $s1, $s2, Label`

Name	Register Number	Usage	Preserve on call?
$zero	0	constant 0 (hardware)	n.a.
$at	1	reserved for assembler	n.a.
$v0 - $v1	2-3	returned values	no
$a0 - $a3	4-7	arguments	yes
$t0 - $t7	8-15	temporaries	no
$s0 - $s7	16-23	saved values	yes
$t8 - $t9	24-25	temporaries	no
$k0 - $k1	26-27	reserved for OS kernel	n.a.
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return addr	yes

- Such branches are included in the instruction set as pseudo instructions
- Its why the assembler needs a reserved register (**`$at`**)

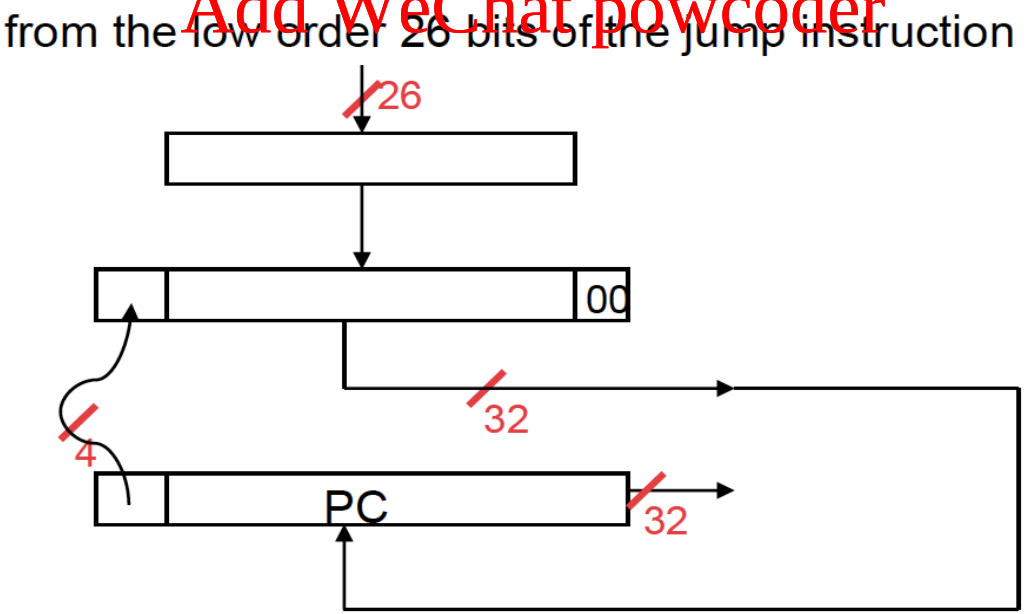
- recognized (and expanded) by the assembler

Other Control Flow Instructions

- MIPS also has an unconditional branch instruction or jump instruction:

```
j   label          #go to label
```

- Instruction Format (J Format):



Aside: Branching Far Away

- What if the branch destination is further away than can be captured in 16 bits?

- The assembler comes to the rescue – it inserts an unconditional jump to the branch target and inverts the condition


```
beq  $s0, $s1, L1
```

becomes

```
     bne  $s0, $s1, L2
     j    L1
L2:
```

Instructions for Accessing Procedures

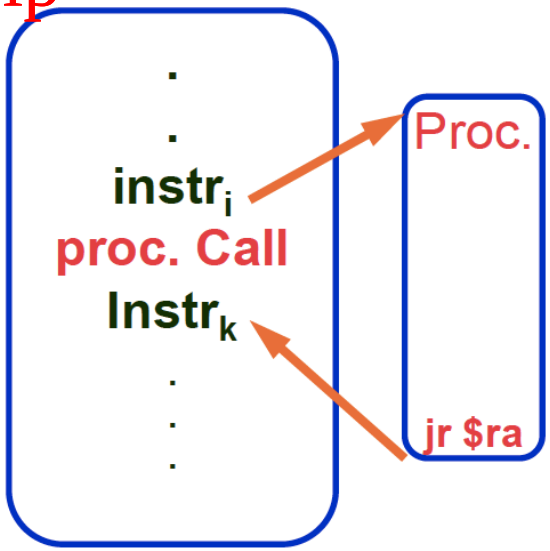
- MIPS procedure call instruction:

```
jal   ProcedureAddress      #jump and link
```

- Saves PC+4 in register **$ra** to have a link to the next instruction for the procedure return

- Machine format (J format):

op	26 bit address



- Then can do procedure return with a

```
jr    $ra                   #return
```

- Instruction format (R format):

op	rs				funct

Aside: Spilling Registers

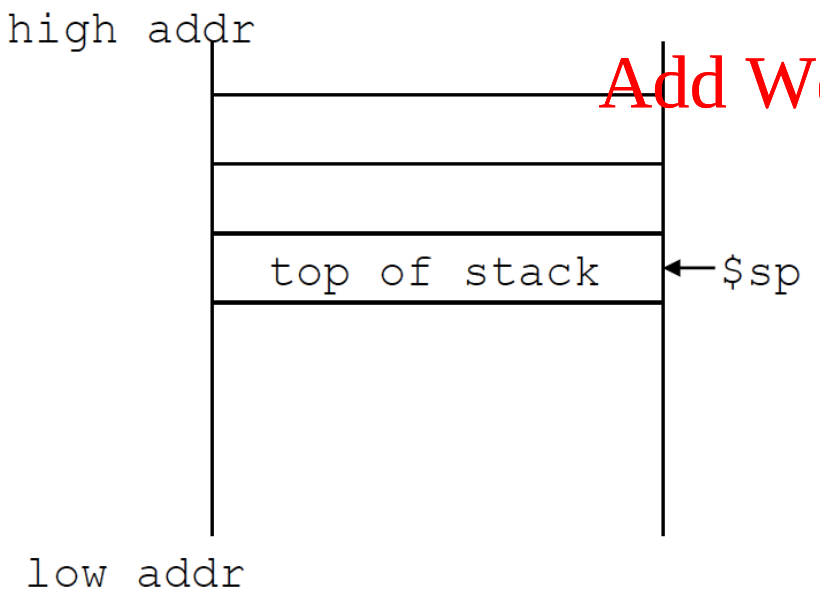
- What if the callee needs more registers? What if the procedure is recursive?
  - uses a stack – a last-in-first-out queue – in memory for passing additional values or saving (recursive) return address(es)

- One of the general registers, $sp, is used to address the stack (which "grows" from high address to low address)
  - add data onto the stack – push

    $sp = $sp – 4
    data on stack at new $sp

  - remove data from the stack – pop

    data from stack at $sp
    $sp = $sp + 4



Leaf Procedure Example

- C code:

```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```


  - Arguments g, …, j in $a0, …, $a3
  - f in $s0 (hence, need to save $s0 on stack)
  - Result in $v0

Leaf Procedure Example

- MIPS code:

```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

```
leaf_example:
   addi $sp, $sp, -4
   sw   $s0, 0($sp)
   add  $t0, $a0, $a1
   add  $t1, $a2, $a3
   sub  $s0, $t0, $t1
   add  $v0, $s0, $zero
   lw   $s0, 0($sp)
   addi $sp, $sp, 4
   jr   $ra
```

Lecture 2
Slides 24

Save $s0 on stack

Procedure body

Result

Restore $s0

Return

Non-Leaf Procedures

- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
  - Its return address
  - Any arguments and temporaries needed after the call
- Restore from the stack after the call

Non-Leaf Procedure Example

- C code:

```
int fact (int n)
{
  if (n < 1) return f;
  else return n * fact(n - 1);
}
```


  - Argument n in $a0
  - Result in $v0

Non-Leaf Procedure Example

e.g. 5! = 5x4x3x2x1=120

- MIPS code:

```
int fact (int n)
{
  if (n < 1) return f;
  else return n * fact(n - 1);
}
```

```
fact:
   addi $sp, $sp, -8      # adjust stack for 2 items
   sw   $ra, 4($sp)       # save return address
   sw   $a0, 0($sp)       # save argument
   slti $t0, $a0, 1       # test for n < 1
   beq  $t0, $zero, L1
   addi $v0, $zero, 1     # if so, result is 1
   addi $sp, $sp, 8       #   pop 2 items from stack
   jr   $ra               #   and return
L1: addi $a0, $a0, -1     # else decrement n
   jal  fact              # recursive call
   lw   $a0, 0($sp)       # restore original n
   lw   $ra, 4($sp)       #   and return address
   addi $sp, $sp, 8       # pop 2 items from stack
   mul  $v0, $a0, $v0     # multiply to get result
   jr   $ra               # and return
```