

Computer Organization & Assembly Language Programming (CSE 2312)

Lecture 7: More on Logical Operations , Conditional
Operations, and Procedure.

Taylor Johnson

Important Concepts from Previous Lectures

- Arithmetic Operations
- Some Processor Components
 - Register Operands
 - Memory Operands

Announcements and Outline

- Homework 2 Due
 - Read chapter 2 (ARM version on Blackboard site)
- Review from last time / Chapter 2
 - Arithmetic Operations
 - Logical Operations
- More on Logical Operations
- Conditional Operations
- Procedure

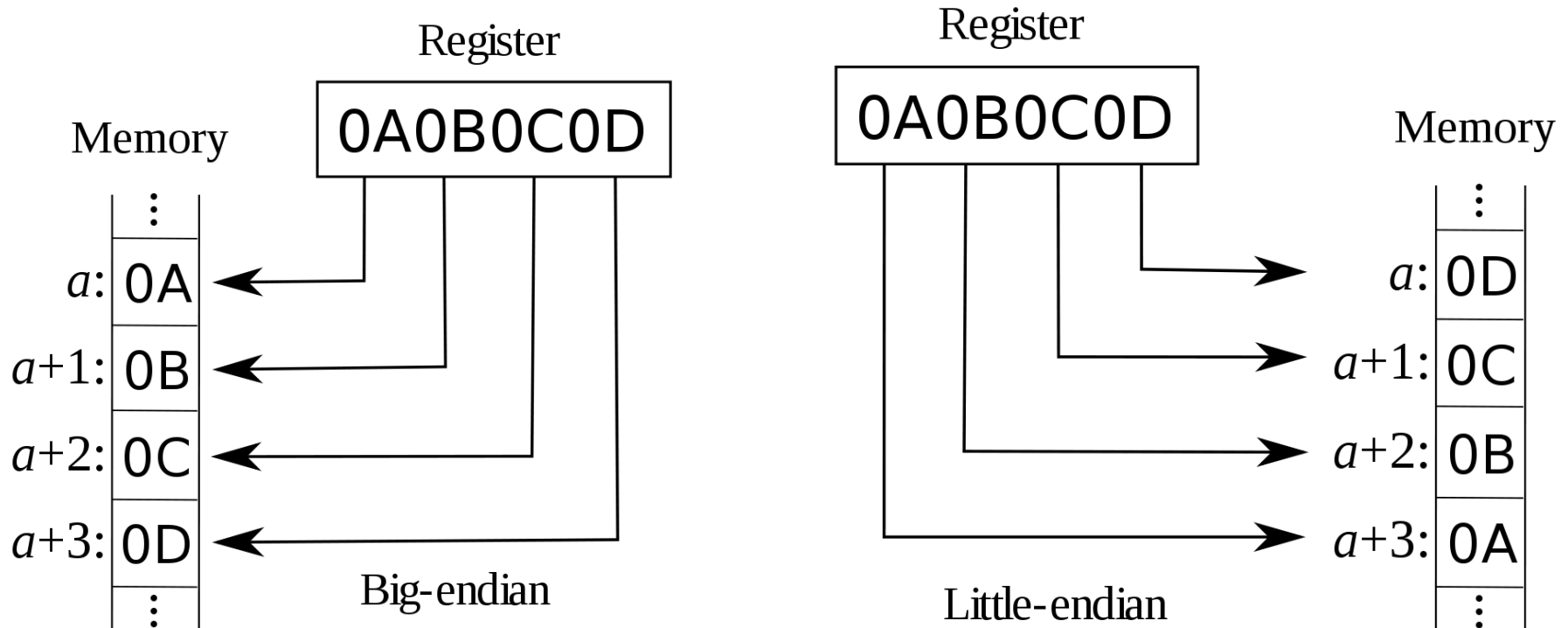
Review: Byte Ordering - Endianness

- How do we store an integer in memory?
- Simple answer: in binary
- Actual answer: yes, in binary, but this does not fully specify how we store the number
- Unfortunately, we have two choices
- Common architectures may follow either choice, and mess ensues, unless we are aware of this issue and we deal with it explicitly
- This is the problem of **endianness**

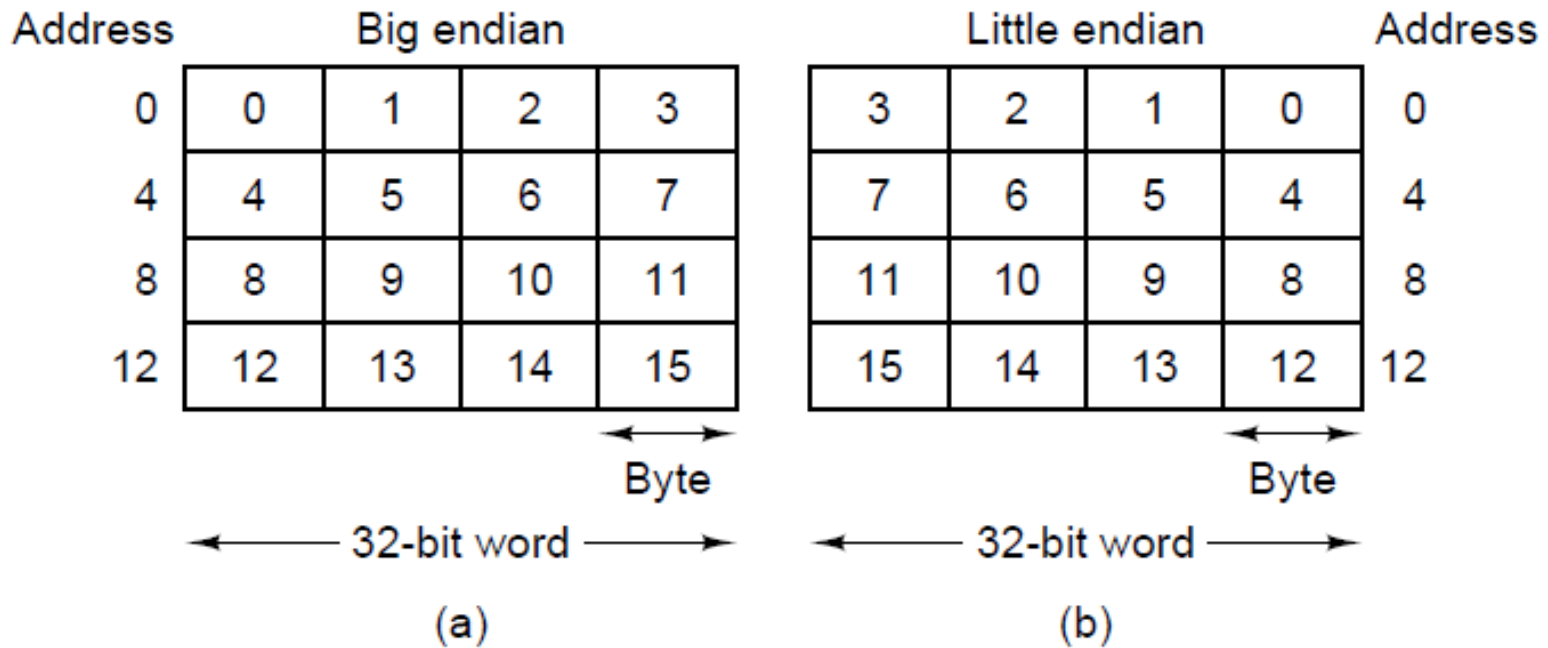
Review: Endianness

- Little-endian: increasing numeric significance with increasing memory addresses
- Big-endian: decreasing numeric significance with increasing memory addresses
- Little-Endian Examples
 - x86, x86-64, 8051, DEC Alpha, Atmel AVR
- Big-Endian Examples
 - Motorola 6800 and 68k series, Xilinx Microblaze, IBM POWER, and System/360
- Bi-Endianness
 - Ability for computer to operate using either
 - SPARC
 - ARM architecture: little-endian before version 3, now bi-endian

Review: Endianness Example



Review: Byte Ordering Visualization



(a) Big endian memory. (b) Little endian memory.

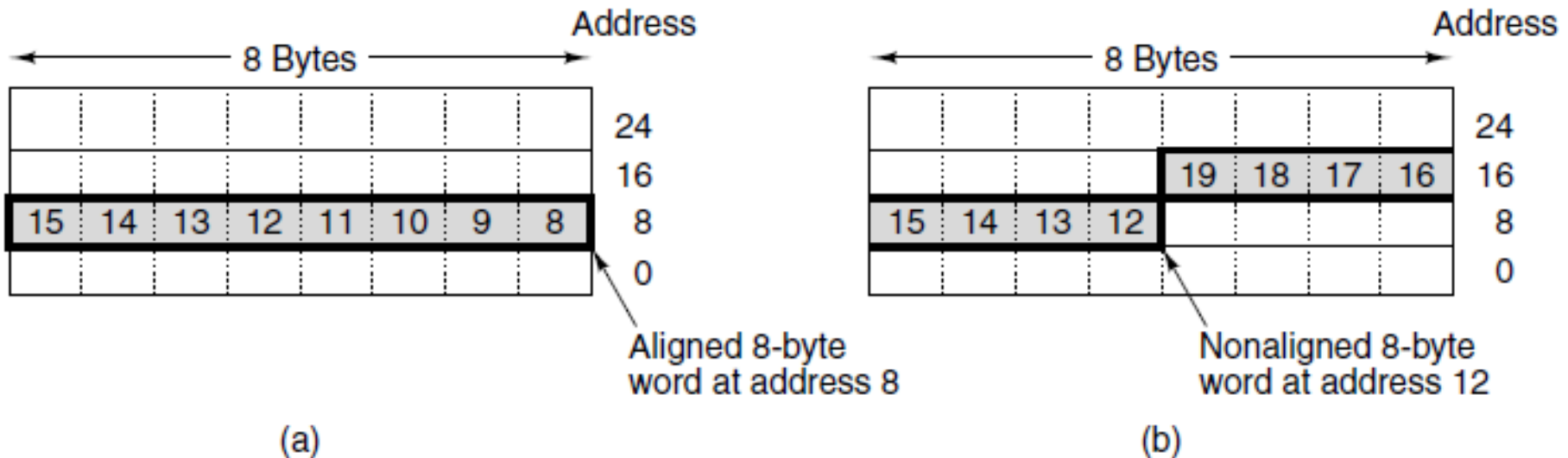
Main difference: ordering of bytes in a word

- Left-to-right in big endian.
- Right-to-left in little-endian.

Review: Memory: Words and Alignment

- Bytes are grouped into **words**
- Depending on the machine, a word can be:
 - 32 bits (4 bytes) , or
 - 64 bits (8 bytes), or ... (16-bits, 128 bits, etc.)
- Oftentimes it is required that words are aligned
- This means that:
 - 4-byte words can only begin at memory addresses that are multiples of 4: 0, 4, 8, 12, 16...
 - 8-byte words can only begin at memory addresses that are multiples of 8: 0, 8, 16, 24, 32, ...

Review: Memory Models



An 8-byte word in a little-endian memory.
 (a) Aligned. (b) Not aligned. Some machines require that words in memory be aligned.

Review: Memory Cells and Addresses

- Memory cell: a piece of memory that contains a specific number of bits
 - How many bits depends on the architecture
 - In modern architectures, it is almost universal that a cell contains 8 bits (1 byte), and that will be also our convention in this course
- Memory address: a number specifying a location of a memory cell containing data
 - Essentially, a number specifying the location of a byte of memory

Review: Memory Cells and Addresses

- The number of unique memory addresses depends on the size of the memory and the size of each cell
- For example, suppose we have a 96-bit memory.
- If each cell is 8 bits, we have ??? addresses?
- If each cell is 12 bits, we have ??? addresses?
- If each cell is 16 bits, we have ??? addresses?

Review: Memory Cells and Addresses

- The number of unique memory addresses depends on the size of the memory and the size of each cell
- For example, suppose we have a 96-bit memory.
- If each cell is 8 bits, we have 12 addresses?
- If each cell is 12 bits, we have 8 addresses?
- If each cell is 16 bits, we have 6 addresses?

- Convention used almost everywhere, and in this course: if a memory has n cells, the addresses of these cells will be from **0** to **$n-1$** .

Review: Address Spaces For Instructions and Data

- Typically memory can be accessed using a single address space
 - For example, if we have 4 GB of memory, each byte has an address from 0 to $2^{32} - 1$.
 - Each memory location may store instructions at some point and data at some other point
- An alternative is to have separate address spaces for instructions and data
 - In that case, a memory location is permanently dedicated to either storing instructions or to storing data
 - Instead of a single **load** instruction, we have **load_instructions** and **load_data**

Review: Effects of Separate Address Spaces

- If A is a valid memory address, `load_instructions A` and `load_data A` access different memory locations.
 - `load_instructions A` accesses address A in the instructions space.
 - `load_data A` accesses address A in the data space.
- This makes it harder for malware to cause trouble. Why?

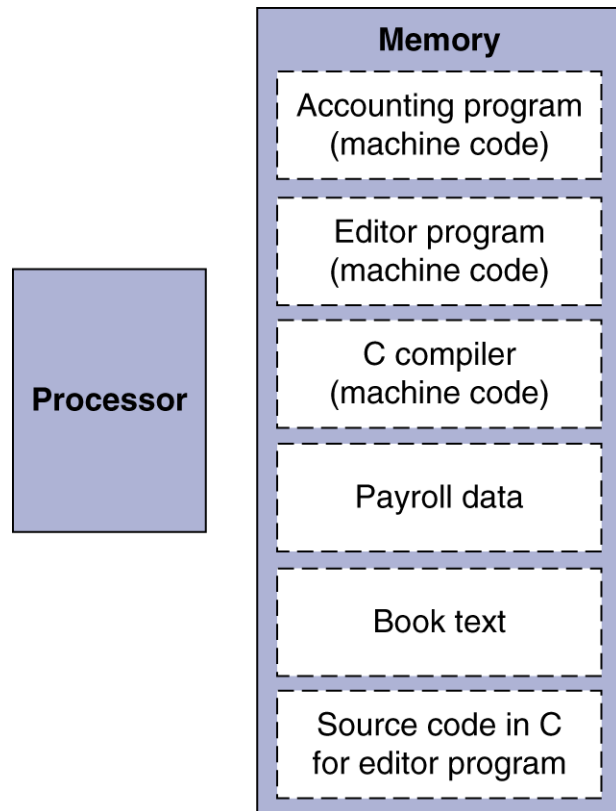
Review: Effects of Separate Address Spaces

- If A is a valid memory address, `load_instructions A` and `load_data A` access different memory locations.
 - `load_instructions A` accesses address A in the instructions space.
 - `load_data A` accesses address A in the data space.
- This makes it harder for malware to cause trouble. Why?
- A common way for malware to attack is to:
 - Run as regular program.
 - Modify memory locations that store instructions, thus modifying other programs (such as the operating system).
- If instruction memory is accessed with different instructions, such behavior can easily be prevented.

Review: Registers vs. Memory

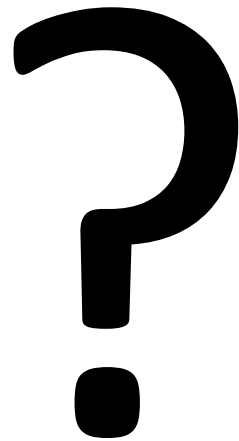
- Registers are faster to access than memory
- Operating on memory data requires loads and stores
 - More instructions to be executed
- Compiler must use registers for variables as much as possible
 - Only spill to memory for less frequently used variables
 - Register optimization is important!

Review: Stored Program Computers



- 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

Questions?



Review: 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
```

offset



base register



Review: 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
```

Review: Immediate Operands

- Constant data specified in an instruction

```
addi $s3, $s3, 4
```

- No subtract immediate instruction

- Just use a negative constant

```
addi $s2, $s1, -1
```

- *Design Principle 3*: Make the common case fast

- Small constants are common
- Immediate operand avoids a load instruction

Review: Sign Extension

- Representing a number using more bits
 - Preserve the numeric value
- In MIPS instruction set
 - `addi`: extend immediate value
 - `lb`, `lh`: extend loaded byte/halfword
 - `beq`, `bne`: extend the displacement
- Replicate the sign bit to the left
 - c.f. unsigned values: extend with 0s
- Examples: 8-bit to 16-bit
 - +2: `0000 0010` => `0000 0000 0000 0010`
 - -2: `1111 1110` => `1111 1111 1111 1110`

Review: Representing Instructions

- Instructions are encoded in binary
 - Called machine code
- ARM (and MIPS) instructions
 - Encoded as 32-bit instruction words
 - Small number of formats encoding operation code (opcode), register numbers, ...
 - Regularity!
- Register numbers
 - \$t0 – \$t7 are reg's 8 – 15
 - \$t8 – \$t9 are reg's 24 – 25
 - \$s0 – \$s7 are reg's 16 – 23

Review: MIPS R-format Instructions



- Instruction fields
 - op: operation code (opcode)
 - rs: first source register number
 - rt: second source register number
 - rd: destination register number
 - shamt: shift amount (00000 for now)
 - funct: function code (extends opcode)

Review: R-format Example

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

add \$t0, \$s1, \$s2

special	\$s1	\$s2	\$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

$$00000010001100100100000000100000_2 = 02324020_{16}$$

Review: MIPS I-format Instructions



- Immediate arithmetic and load/store instructions
 - rt: destination or source register number
 - Constant: -2^{15} to $+2^{15} - 1$
 - Address: offset added to base address in rs
- *Design Principle 4: Good design demands good compromises*
 - Different formats complicate decoding, but allow 32-bit instructions uniformly
 - Keep formats as similar as possible

Review: ARMR-format Instructions



- Instruction fields
 - op: operation code (opcode)
 - rs: first source register number
 - rt: second source register number
 - rd: destination register number
 - shamt: shift amount (00000 for now)
 - funct: function code (extends opcode)

Review: ARM Instructions in Machine Language

Cond	F	I	Opcode	S	Rn	Rd	Operand2
4 bits	2 bits	1 bit	4 bits	1 bit	6 bits	6 bits	12 bits

- Opcode: Basic operation of the instruction
- Rd: The register destination operand. It gets the result of the operation
- Rn: The first register source operand
- Operand2: The second source operand
- I: Immediate. If I is 0, the second source operand is a register. If I is 1, the second source operand is a 12-bit immediate
- S: Set Condition Code. This field is related to conditional branch instructions
- Cond: Condition. Related to conditional branch instructions
- F: Instruction Format. This field allows ARM to different instruction formats when needed

Review: ARM Instructions in Machine Language

Instruction	Format	Cond	F	I	op	S	Rn	Rd	Operand2
ADD	DP	14	0	0	4_{ten}	0	reg	reg	reg
SUB (subtract)	DP	14	0	0	$2s_{\text{ten}}$	0	reg	reg	reg
ADD (immediate)	DP	14	0	1	4_{ten}	0	reg	reg	constant
LDR (load word)	DT	14	1	n.a.	24_{ten}	n.a.	reg	reg	address
STR (store word)	DT	14	1	n.a.	25_{ten}	n.a.	reg	reg	address

- “**reg**” means a register number between 0 and 15
- “**constant**” means a 12-bit constant
- “**address**” means a 12-bit address
- “**n.a.**” (not applicable) means this field does not appear in this format
- **Op** stands for opcode.

Review: Logical Operations

Instructions for bitwise manipulation

Useful for extracting and inserting groups of bits in a word

Logical operations	C operators	Java operators	ARM instructions
Bit-by-bit AND	&	&	AND
Bit-by-bit OR			ORR
Bit-by-bit NOT	~	~	MVN
Shift left	<<	<<	LSL
Shift right	>>	>>>	LSR

C and Java logical operators and their corresponding ARM instructions.

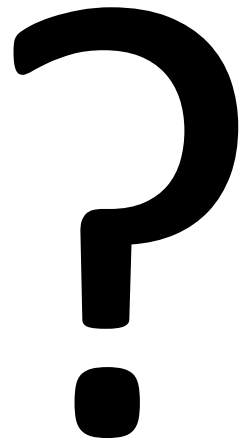
ARM implements **NOT** using a **NOR** with one operand being zero.

Review: Shift Operations



- shamt: how many positions to shift
- Shift left logical
 - Shift left and fill with 0 bits
 - sll by i bits multiplies by 2^i
- Shift right logical
 - Shift right and fill with 0 bits
 - srl by i bits divides by 2^i (unsigned only)

Questions?



Review: Shift Operations (for ARM)

Cond	F	I	opcode	S	Rn	Rd	Shift_imm		Shift		Rm
							Rs	0	Shift		Rm
14	0	0	4	0	2	5	2		0	0	5
14	0	0	13	0	0	6	4		1	0	5
14	0	0	13	0	0	6	3	0	1	1	5

ARM allows shifting by the value found in a register. The following instruction shifts register r5 right by the amount in register r3 and places the result in r6.

```
MOV r6, r5, LSR r3 ; r6 = r5 >> r3
```

Review: AND Operations

- Useful to mask bits in a word
 - Select some bits, clear others to 0

and \$t0, \$t1, \$t2

\$t2	0000 0000 0000 0000 0000 1101 1100 0000
\$t1	0000 0000 0000 0000 0011 1100 0000 0000
\$t0	0000 0000 0000 0000 0000 1100 0000 0000

Review: OR Operations

- Useful to include bits in a word
 - Set some bits to 1, leave others unchanged

or \$t0, \$t1, \$t2

\$t2	0000 0000 0000 0000 0000 1101 1100 0000
\$t1	0000 0000 0000 0000 0011 1100 0000 0000
\$t0	0000 0000 0000 0000 0011 1101 1100 0000

Review: NOT Operations

- Useful to invert bits in a word
 - Change 0 to 1, and 1 to 0
- MIPS has NOR 3-operand instruction
 - $a \text{ NOR } b == \text{NOT}(a \text{ OR } b)$

```
nor $t0, $t1, $zero
```

Register 0: always
read as zero

```
$t1 0000 0000 0000 0000 0011 1100 0000 0000
```

```
$t0 1111 1111 1111 1111 1100 0011 1111 1111
```

Conditional Operations

- Branch to a labeled instruction if a condition is true
 - Otherwise, continue sequentially
- **beq** *rs*, *rt*, **L1**
 - if ($rs == rt$) branch to instruction labeled L1;
- **bne** *rs*, *rt*, **L1**
 - if ($rs != rt$) branch to instruction labeled L1;
- **j** **L1**
 - unconditional jump to instruction labeled L1

Conditional Operations

ARM-7:

This pair of instructions means go to the statement labeled L1 if the value in register1 equals the value in register2.

The mnemonic CMP stands for *compare* and BEQ stands for *branch if equal*.

```
CMP register1, register2  
BEQ L1
```

Compiling If Statements

- C code:

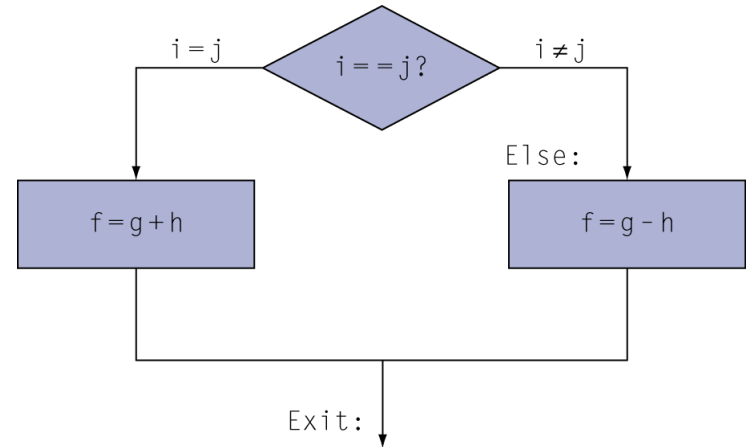
```
if (i==j) f = g+h;
else f = g-h;
```

- f, g, ... in \$s0, \$s1, ...

- Compiled MIPS code:

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

Assembler calculates addresses



Compiling if-then-else into Conditional Branches

- C code:

```
if (i == j) f = g + h; else f = g - h;
```

- `f` through `j` correspond to the five `r0` through `r4`

- Compiled ARM code:

```
CMP    r3, r4           ; compare i and j
BNE,   Else           ; go to Else if i ≠ j
Add    r0, r1, r2      ; f = g + h (skipped if i ≠ j)
B      Exit           ; Go to Exit (unconditional branch)
Else:  SUB    r0, r1, r2 ; f = g - h (skipped if i = j)
Exit:  ; Go to Exit
```


Compiling Loop Statements

- C code:

```
while (save[i] == k) i += 1;
```

- i in \$s3, k in \$s5, address of save in \$s6

- Compiled MIPS code:

```
Loop:  sll    $t1, $s3, 2           # Temp reg $t1 = I * 4
      add    $t1, $t1, $s6        # $t1 = addr of save[i]
      lw     $t0, 0($t1)          # Temp reg $t0 = save[i]
      bne   $t0, $s5, Exit        # Go to Exit if save[i] = k
      addi  $s3, $s3, 1           # i = i + 1
      j     Loop                 # Go to Loop

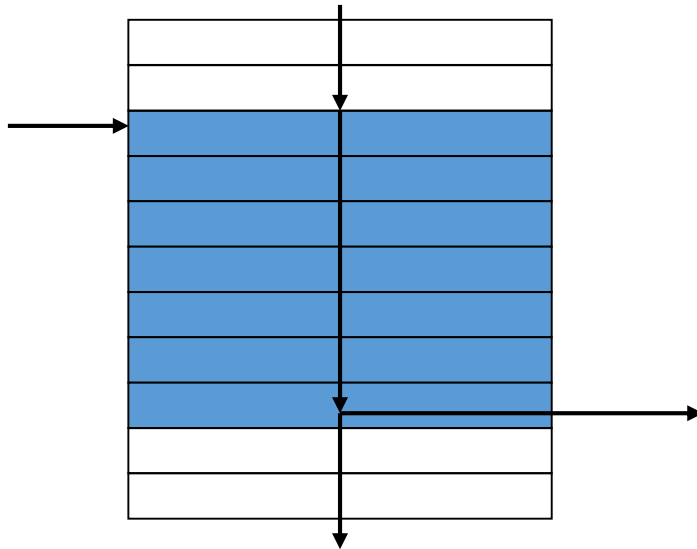
Exit:  ...
```

Branch Instruction Design

- Why not `blt`, `bge`, etc?
- Hardware for `<`, `≥`, ... slower than `=`, `≠`
 - Combining with branch involves more work per instruction, requiring a slower clock
 - All instructions penalized!
- `beq` and `bne` are the common case
- This is a good design compromise

Basic Blocks

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



- A compiler identifies basic blocks for optimization
- An advanced processor can accelerate execution of basic blocks

Branch Instruction Design

- Why not blt, bge, etc?
- Hardware for $<$, \geq , ... slower than $=$, \neq
 - Combining with branch involves more work per instruction, requiring a slower clock
 - All instructions penalized!
- beq and bne are the common case
- This is a good design compromise

Signed vs. Unsigned

- Signed comparison: `slt, slti`
- Unsigned comparison: `sltu, sltui`
- Example
 - `$s0 = 1111 1111 1111 1111 1111 1111 1111 1111`
 - `$s1 = 0000 0000 0000 0000 0000 0000 0000 0001`
 - `slt $t0, $s0, $s1 # signed set on less than`
 - $-1 < +1 \Rightarrow \$t0 = 1$
 - `sltu $t0, $s0, $s1 # unsigned set on less than`
 - $+4,294,967,295 > +1 \Rightarrow \$t0 = 0$

Procedure Calling

- Steps required
 1. Place parameters in registers
 2. Transfer control to procedure
 3. Acquire storage for procedure
 4. Perform procedure's operations
 5. Place result in register for caller
 6. Return to place of call

Register Usage

- ❑ $\$a0 - \$a3$: arguments (reg's 4 – 7)
- ❑ $\$v0, \$v1$: result values (reg's 2 and 3)
- ❑ $\$t0 - \$t9$: temporaries
 - Can be overwritten by callee
- ❑ $\$s0 - \$s7$: saved
 - Must be saved/restored by callee
- ❑ $\$gp$: global pointer for static data (reg 28)
- ❑ $\$sp$: stack pointer (reg 29)
- ❑ $\$fp$: frame pointer (reg 30)
- ❑ $\$ra$: return address (reg 31)

Procedure Call Instructions

- Procedure call: jump and link

`jal ProcedureLabel`

- Address of following instruction put in `$ra`
- Jumps to target address

- Procedure return: jump register

`jr $ra`

- Copies `$ra` to program counter
- Can also be used for computed jumps
 - e.g., for case/switch statements

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:

```
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
```

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);
}
```

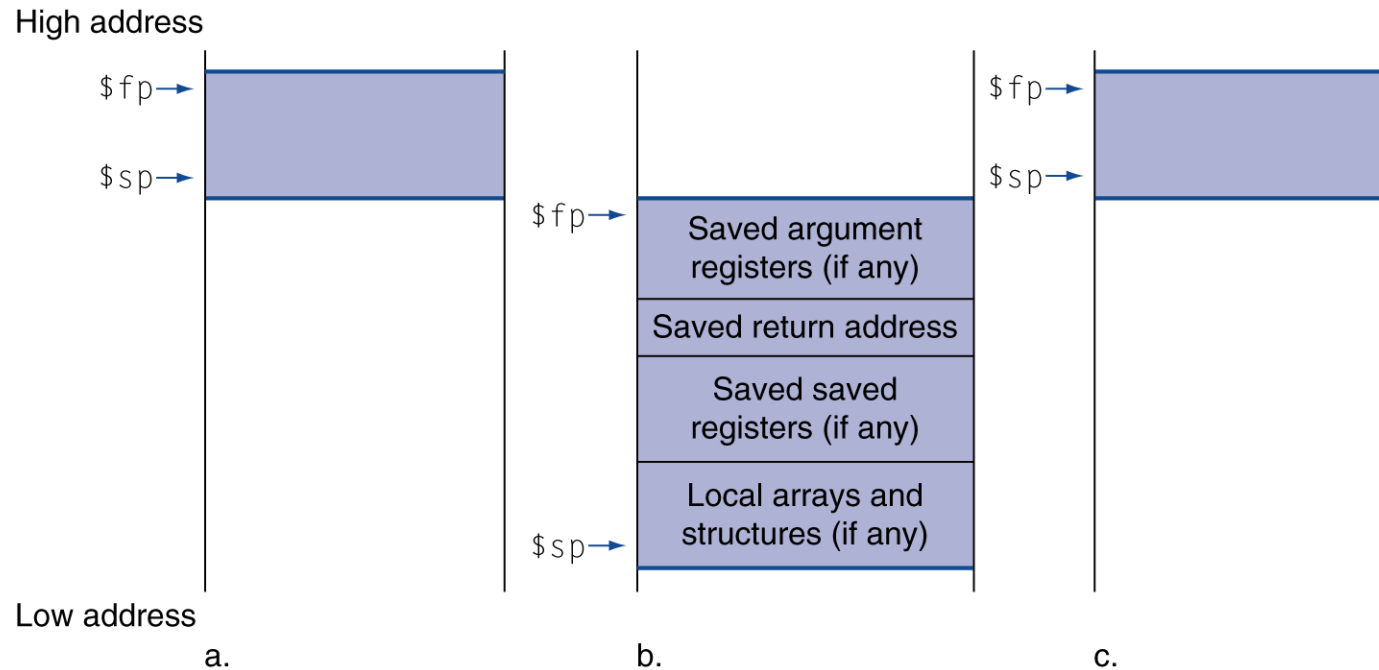
- Argument n in \$a0
- Result in \$v0

Non-Leaf Procedure Example

- MIPS code:

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

Local Data on the Stack



- Local data allocated by callee
 - e.g., C automatic variables
- Procedure frame (activation record)
 - Used by some compilers to manage stack storage