CPT_S 260 Intro to Computer Architecture
Lecture 14

Intro to MIPS III
February 11, 2022

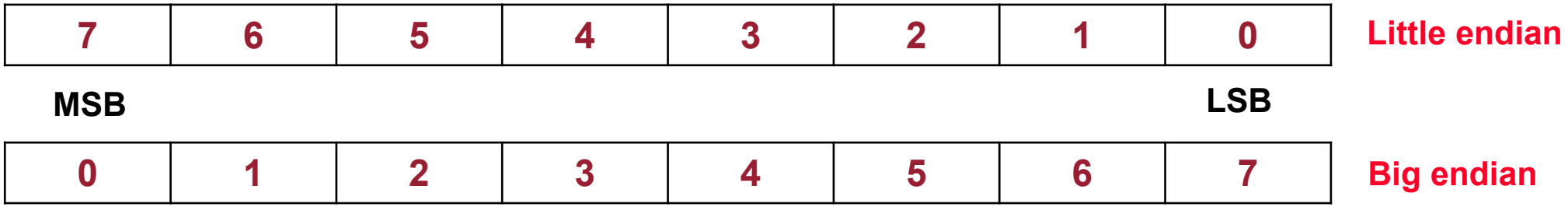
Ganapati Bhat
School of Electrical Engineering and Computer Science
Washington State University

Recap: 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

Byte Ordering

- **There are two main conventions**
  - Little endian
    - » Specifies the address of the least significant byte in a word
  - Big endian
    - » Specifies the address of the most significant byte in a word



- **Fun fact: Name based on Gulliver's travels**
  - https://en.wikipedia.org/wiki/Endianness#Etymology

Logical Operations

- **Instructions for bitwise manipulation**

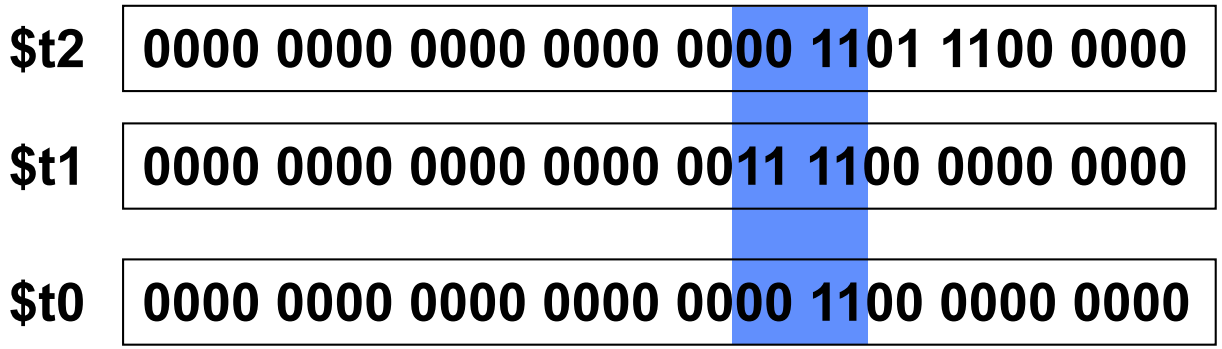
Operation	C	Java	MIPS
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bitwise AND	&	&	and, andi
Bitwise OR	|	|	or, ori
Bitwise NOT	~	~	nor

- **Useful for extracting and inserting groups of bits in a word**

AND Operations

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

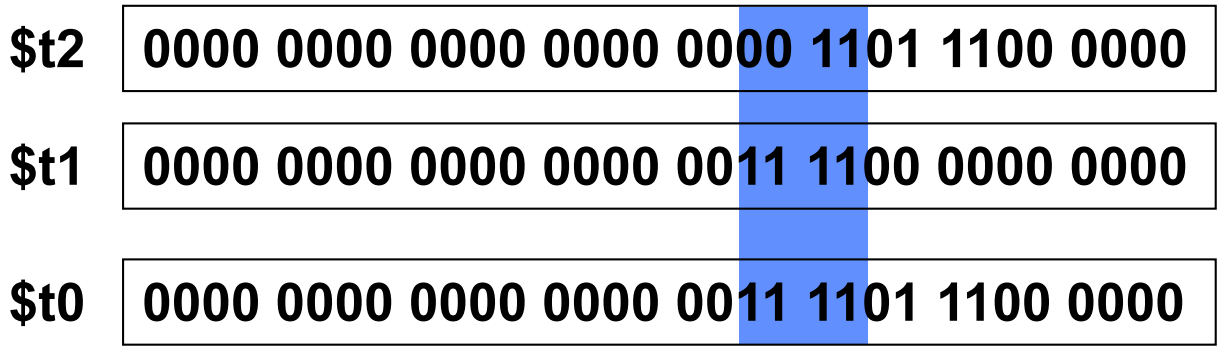
```
and $t0, $t1, $t2
```



OR Operations

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

```
or $t0, $t1, $t2
```



NOT Operations

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

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

Register 0: always read as zero

$t1	**0000 0000 0000 0000 0011 1100 0000 0000**
$zero	**0000 0000 0000 0000 0000 0000 0000 0000**
$t0	**1111 1111 1111 1111 1100 0011 1111 1111**

Representing Instructions

- **Instructions are encoded in binary**
  - Called machine code
- **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

MIPS Instruction Computer Representation

- Instructions are kept in the computer as a series of high and low electronic signals
- May be represented as numbers
- Each piece of an instruction can be considered as an individual number
- Placing these numbers side by side forms the instruction.

MIPS Instructions

- Keep all instructions the same length
- Require different kinds of instruction formats for different kinds of instructions.
- R-format: r for registers (R-type)
- I-format: i for immediate (I-type)
- The formats are distinguished by the values in the first field each format is assigned a distinct set of values in the first field (op)
- The hardware knows whether to treat the last half of the instruction as three fields (R-type) or as a single field (I-type)

R-format Instructions

- op: Basic operation of the instruction, traditionally called the opcode.
- rs: The first register source operand
- rt: The second register source operand.
- rd: The register destination operand. It gets the result of the operation
- shamt: Shift amount (00000 for now)
- funct: Function. This field, often called the function code, selects the specific variant of the operation in the op field

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

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

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

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

Compiling If Statements

- **C code:**

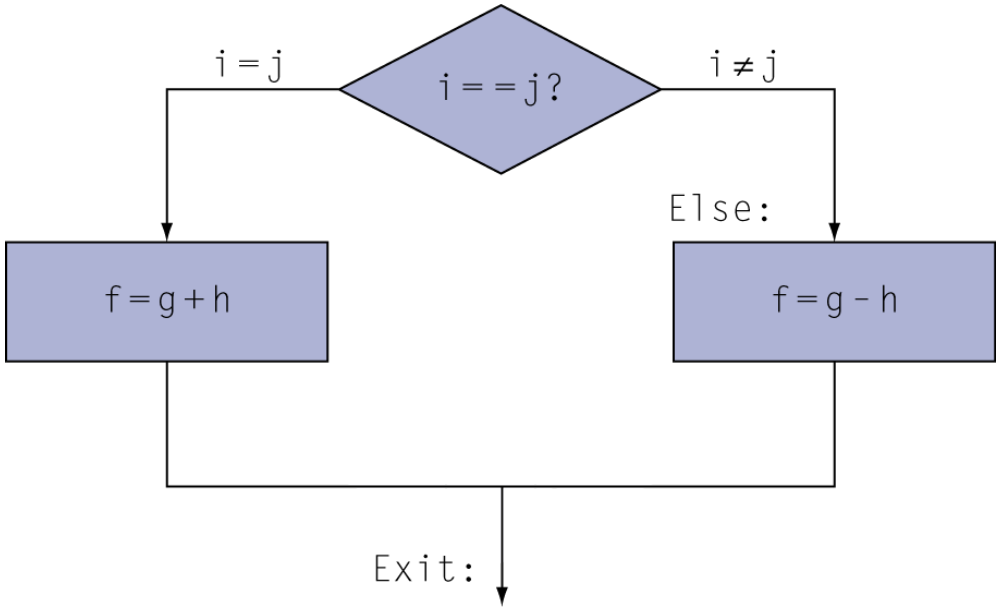
```
if (i==j) f = g+h;
else f = g-h;
```

  - 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 Loop Statements

- **C code:**

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

  – i in $s3, k in $s5, address of save in $s6

- **Steps**
  – Load save[i] into a temporary register.
    » Base Address of save
    » Multiply i to get the byte address of the index (by 4)
  – Loop test
    » Choose to use the bne or beq
  – Make branch labels for each portion of the loop test

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 = address 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:
```

More Conditional Operations

- **Set result to 1 if a condition is true**
  - Otherwise, set to 0
- **`slt rd, rs, rt`**
  - if (rs < rt) rd = 1; else rd = 0;
- **`slti rt, rs, constant`**
  - if (rs < constant) rt = 1; else rt = 0;
- **Use in combination with beq, bne**

```
slt $t0, $s1, $s2  # if ($s1 < $s2)
bne $t0, $zero, L  #   branch to L
```

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`
    - » –1 < +1 ⇒ $t0 = 1
  - `sltu $t0, $s0, $s1  # unsigned`
    - » +4,294,967,295 > +1 ⇒ $t0 = 0