컴퓨터 공학 개론
Lecture 5

Instructions: Language of the Computer

2017

김태성

Hexadecimal

- **Base 16**
  - **Compact representation of bit strings**
  - **4 bits per hex digit**

Hex	Binary	Decimal	Hex	Binary	Decimal	Hex	Binary	Decimal	Hex	Binary	Decimal
0	0000	0	**4**	0100	4	**8**	1000	8	**c**	1100	12
1	0001	1	**5**	0101	5	**9**	1001	9	**d**	1101	13
2	0010	2	**6**	0110	6	**a**	1010	10	**e**	1110	14
3	0011	3	**7**	0111	7	**b**	1011	11	**f**	1111	15

- Example: **eca8 6420**$_{\mathbf{HEX}}$
  - 1110 1100 1010 1000 0110 0100 0010 0000

Representing Instructions

- Instructions are encoded in binary
  - Called machine code
- MIPS instructions
  - Encoded as 32-bit instruction words
  - E.g. `add $t0, $s1, $s2`

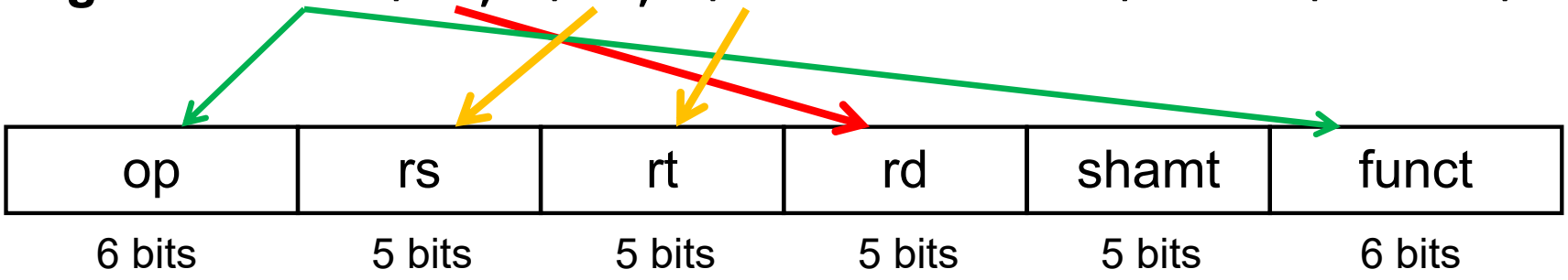
    => 0000 0010 0011 0010 0100 0000 0010 0000
  - Represented as fields of binary numbers including operation code (opcode), register numbers, …
  - Regularity! (IA-32: 8-bit, 16-bit, 24-bit, 32-bit,… instructions)
- Register numbers
  - $t0 – $t7 are reg's 8 – 15 (i.e. $t0 = $8, $t1 = $9, …, $t7 = $15)
  - $t8 – $t9 are reg's 24 – 25 ($t8 = $24, $t9 = $25)
  - $s0 – $s7 are reg's 16 – 23 ($s0 = $16, $s1 = $17, …, $s7 = $23)

MIPS Instruction Formats

- **R-type instructions**
  - **E.g.** `add $t0, $s1, $s2  // $t0 = $s1 + $s2`
- **I-type instructions**
  - **E.g.** `addi $t0, $s1, 4  // $t0 = $s1 + 4`
  - `lw $t0, 32($s3)  // $t0 = MEM[$s3+32]`
- **J-type instructions**
  - **E.g.** `j Exit  // jump to Exit`

MIPS R-format Instructions

- R-format instructions
  - Instructions with 3 register operands
  - E.g. `add $t0, $s1, $s2` `// $t0 = $s1 + $s2`



- Instruction fields
  - op: operation code (opcode) – '0' for R-type instructions
  - 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)

R-format Example

```
add $t0, $s1, $s2        // $t0 = $s1 + $s2
```

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

0	$s1	$s2	$t0	0	add

0	17	18	8	0	32

000000	10001	10010	01000	00000	100000

$$0000\ 0010\ 0011\ 0010\ 0100\ 0000\ 0010\ 0000_2 = 0232\ 4020_{16}$$

MIPS I-format Instructions

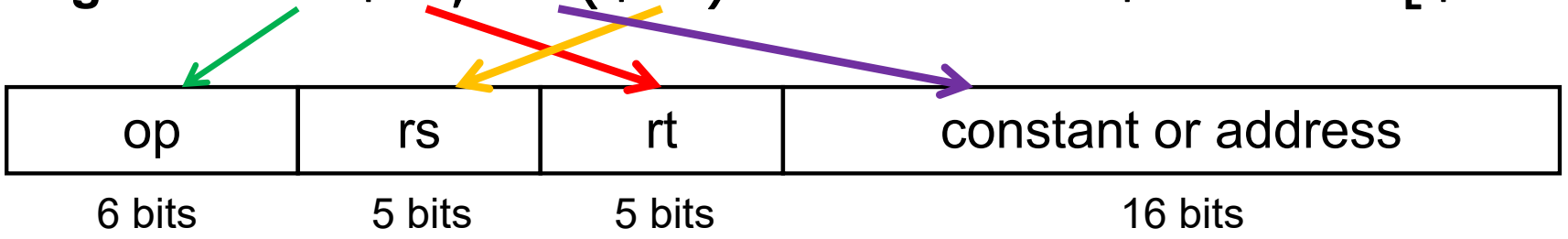
- I-format Instructions
  - Immediate arithmetic and load/store instructions
  - Two register operands + one immediate operand (constant)
  - E.g. `addi $t0, $s1, 4` `// $t0 = $s1 + 4`

  - rt: destination register number
  - Constant: -2^{15} to $+2^{15} - 1$

- *Design Principle 4:* Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

MIPS I-format Instructions

- **E.g.** `lw $t0, 32($s3)` `// $t0 = MEM[$s3+32]`



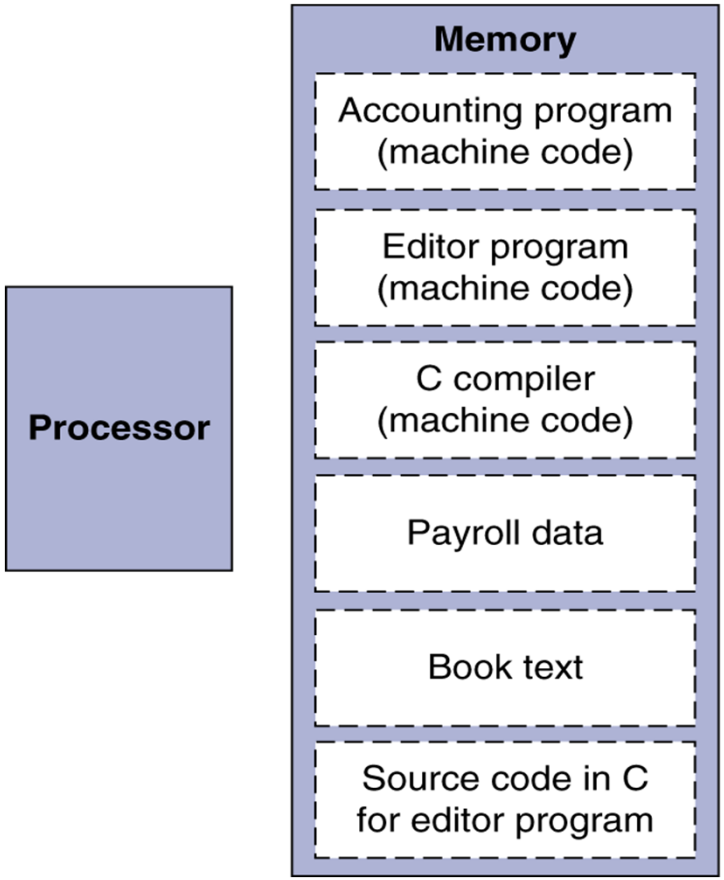
  - **rt: destination register number**
  - **Address: offset added to base address in rs**

- **E.g.** `sw $t0, 32($s3)` `// MEM[$s3+32] = $t0`

  - **rt: source register number**
  - **Address: offset added to base address in rs**

Stored Program Computers

The BIG Picture



- Instructions represented in binary, just like data
- Instructions and data stored in memory
  - => Stored program computer
  - => *Von Neumann Architecture*
- Binary compatibility allows compiled programs to work on different computers
  - Standardized ISAs (e.g. IA-32)
  - Intel core i5, AMD K10

Logical Operations

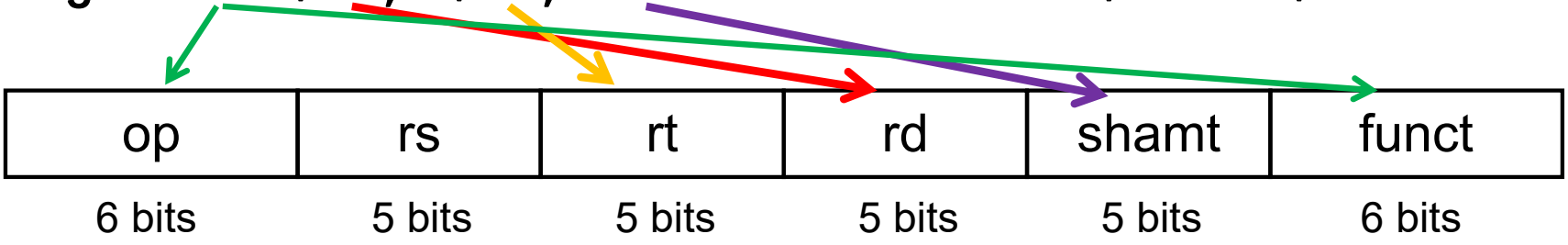
- **Instructions for bitwise manipulation**

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
- Cf. logical AND: &&, logical OR: ||

Shift Operations

- **Shift left logical**
  - **Shift left and fill with 0 bits**
  - **`sll` by *i* bits multiplies by 2^i**
  - **`E.g. sll $t2, $s0, 4`** `// $t2 = $s0 << 4`



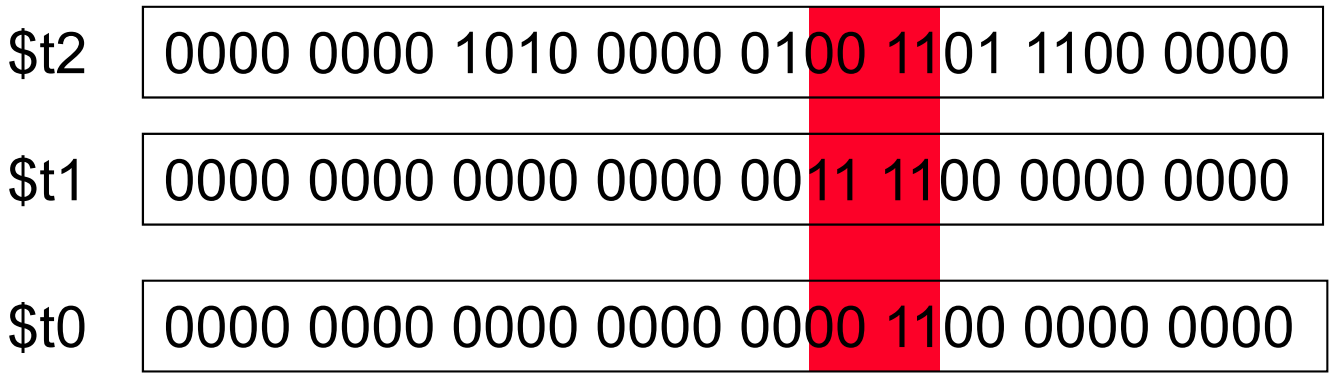
    - **Use R-format**
    - **shamt: how many positions to shift**

- **Shift right logical**
  - **Shift right and fill with 0 bits**
  - **`srl` by *i* bits divides by 2^i (unsigned only)**
  - **`E.g. srl $t0, $s1, 2`** `// $t0 = $s1 >> 2`

AND Operations

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

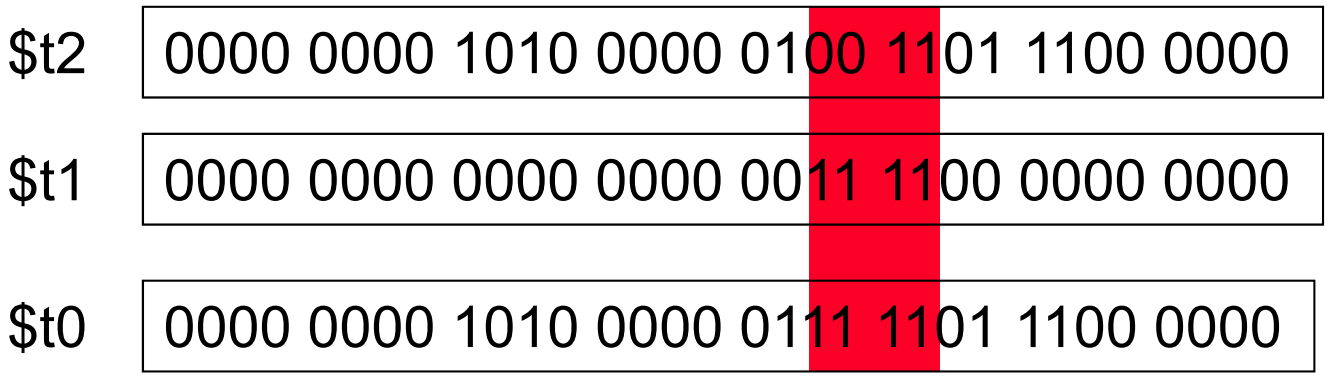
```
and $t0, $t1, $t2   // $t0 = $t1 & $t2
```



OR Operations

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

```
or $t0, $t1, $t2    // $t0 = $t1 | $t2
```



NOT Operations

- Useful to invert bits in a word
  - Change 0 to 1, and 1 to 0
- MIPS doesn't have NOT because …
- NOR can be used in place of NOT
  - a NOR 0 == NOT (a OR 0) == NOT (a)

```
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  // branch if equal`
  - if (rs == rt) branch to instruction labeled L1;
- `bne rs, rt, L1  // branch if not equal`
  - if (rs != rt) branch to instruction labeled L1;
- `j L1  // jump`
  - unconditional jump to instruction labeled L1

Compiling If Statements

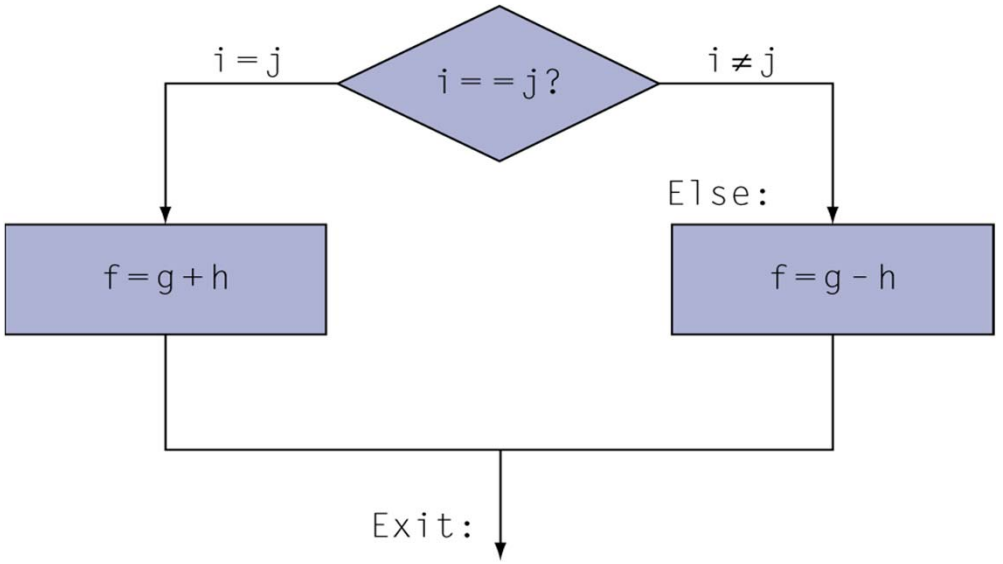
° C code:

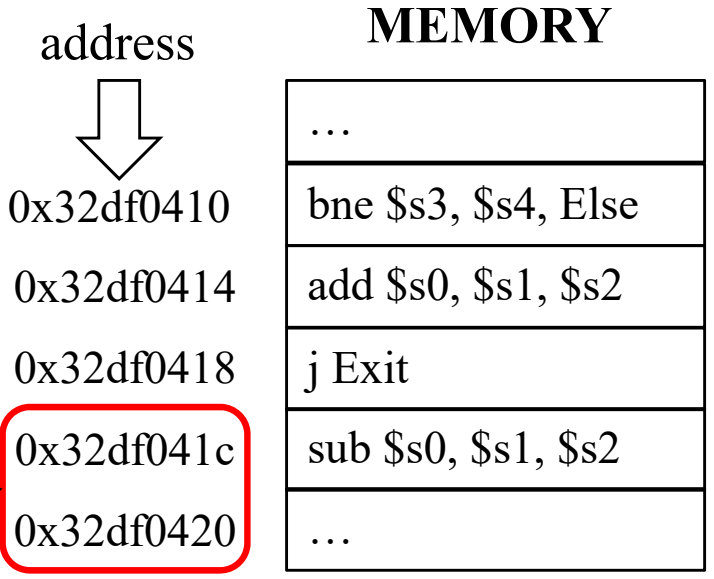
```
if (i==j) f = g+h;
else f = g-h;
```

• f, g, … in $s0, $s1, …

° Compiled MIPS code:

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





Labels represent address of the instruction

Compiling Loop Statements

- **C code:**

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

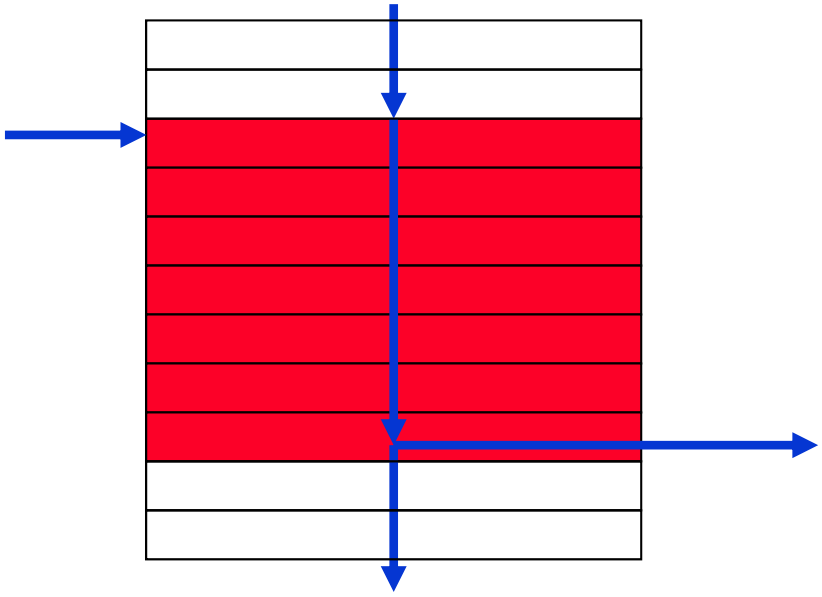
  - **i in $s3, k in $s5, address of save in $s6**

- **Compiled MIPS code:**

```
Loop: sll  $t1, $s3, 2      // $t1 = i * 4
      add  $t1, $t1, $s6    // $t1 = $s6+4*i
                            // $t1 gets the address of
                            // save[i]
      lw   $t0, 0($t1)      // $t0 = mem[$s6+4*i]
                            // $t0 gets save[i]
      bne  $t0, $s5, Exit   // branch if (save[i] == k)
      addi $s3, $s3, 1      // i = i + 1
      j    Loop
Exit: …
```

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

More Conditional Operations

- Set result to 1 if a condition is true
  - Otherwise, set to 0
- `slt rd, rs, rt`

```
// set less than
// 'set' is '1', 'reset' is '0'
```

  - 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
```

Branch Instruction Design

- Why not `blt`, `bge`, etc instead of (slt+bne, slt+beq)?
- Hardware for <, ≥, … is slower than =, ≠
  - Combining with branch involves more work per instruction, requiring a slower clock
  - All instructions penalized if clock becomes slow!
- `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`
    - **(–1) < +1 ⇒ $t0 = 1**
  - `sltu $t0, $s0, $s1  # unsigned`
    - **+4,294,967,295 > +1 ⇒ $t0 = 0**