UNIVERSITÉ Concordia UNIVERSITY
GINA CODY
SCHOOL OF ENGINEERING
AND COMPUTER SCIENCE

Chapter 2

Part I

Instructions: Language of the Computer

Dr. Fadi Alzhouri
COEN: 316

[Based on Figures from *Computer Organization and Design*: *The Hardware/Software Interface* Patterson & Hennessy, 5th ed. © 2014 Elsevier Inc.
&
Computer Organization and Architecture: *Designing for Performance* William Stallings, 8th ed. © 2010 Pearson Education Inc.]

Department of Electrical & Computer Engineering (ECE)

Instruction Set

- Collection of instructions of a computer
- Different computers have different instruction sets
  - But with many aspects in common
- Early computers had very simple instruction sets
  - Simplified implementation
- Many modern computers also have simple instruction sets

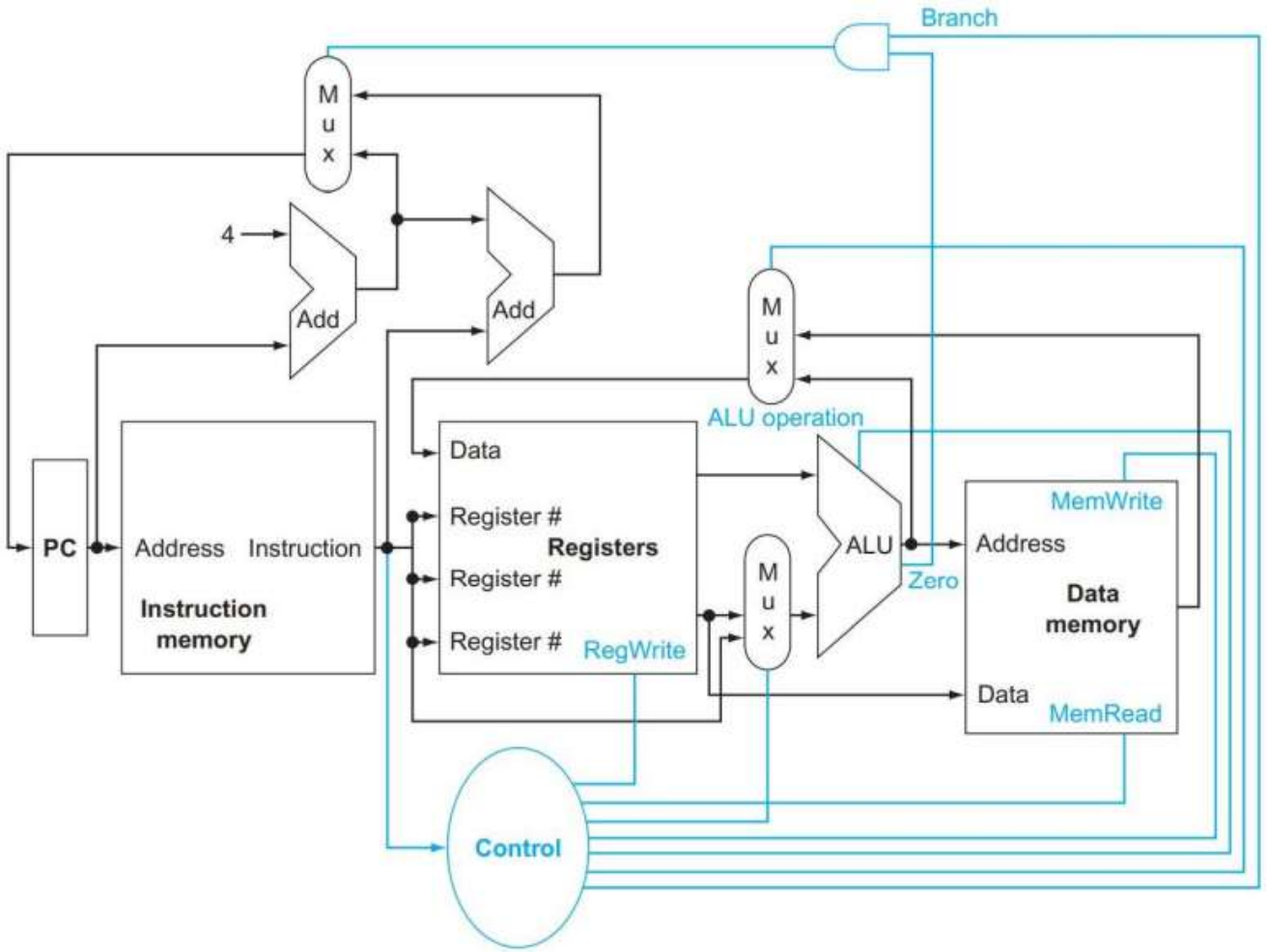
MIPS Instruction Set

- Used as the example throughout the book
- Stanford MIPS commercialized by MIPS Technologies (www.mips.com)
- Large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …
- Typical of many modern ISAs
  - See MIPS Reference Data tear-out card, and Appendixes B and E

MIPS Design Principles

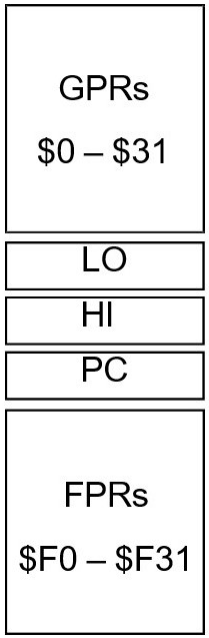
- Simplicity favors regularity
- Smaller is faster
- Make the common case fast
- Good design demands good compromises.

Logical View of the MIPS Processor



Overview of the MIPS Registers

- 32 General Purpose Registers (GPRs)
  - **32-bit registers are used in MIPS32**
  - **Register 0 is always zero**
  - **Any value written to R0 is discarded**
- Special-purpose registers LO and HI
  - Hold results of integer **multiply** and **divide**
- Special-purpose Program Counter (PC)
- 32 Floating Point Registers (FPRs)
  - Floating Point registers can be either **32**-bit or **64**-bit
  - A pair of registers is used for double-precision floating-point



MIPS General Purpose Registers

- 32 General Purpose Registers (GPRs)
- Assembler uses the **dollar notation** to name registers
- $0 is register 0, $1 is register 1, …, and $31 is register 31
- Register $0 is always zero
  - Any value written to $0 is discarded
- Software conventions
  - Software defines names to all registers
    - To standardize their use in programs
- Assembler can refer to registers by name or by number
  - It is easier for you to remember registers by name
  - Assembler converts register name to its corresponding number

$0 = $zero	$16 = $s0
$1 = $at	$17 = $s1
$2 = $v0	$18 = $s2
$3 = $v1	$19 = $s3
$4 = $a0	$20 = $s4
$5 = $a1	$21 = $s5
$6 = $a2	$22 = $s6
$7 = $a3	$23 = $s7
$8 = $t0	$24 = $t8
$9 = $t1	$25 = $t9
$10 = $t2	$26 = $k0
$11 = $t3	$27 = $k1
$12 = $t4	$28 = $gp
$13 = $t5	$29 = $sp
$14 = $t6	$30 = $fp
$15 = $t7	$31 = $ra

Register Operands

- Arithmetic instructions use register operands
- MIPS has a 32 × 32-bit register file
  - Use for frequently accessed data
  - Numbered 0 to 31
  - 32-bit data called a “word”
- Assembler names
  - $at reserved for assembler.
  - $v0,$v1 stores results
  - $a0,…,$a3 stores arguments
  - $t0, …$t7,$t8,$t9 for temporary values, not saved
  - $s0, $s1, …, $s7 for saved variables, for later use.
  - $k0,$k1 reserved by operating system
  - $gp global pointer
  - $sp stack pointer
- *Design Principle 2:* Smaller is faster

Limited number of registers

$0 = $zero	$16 = $s0
$1 = $at	$17 = $s1
$2 = $v0	$18 = $s2
$3 = $v1	$19 = $s3
$4 = $a0	$20 = $s4
$5 = $a1	$21 = $s5
$6 = $a2	$22 = $s6
$7 = $a3	$23 = $s7
$8 = $t0	$24 = $t8
$9 = $t1	$25 = $t9
$10 = $t2	$26 = $k0
$11 = $t3	$27 = $k1
$12 = $t4	$28 = $gp
$13 = $t5	$29 = $sp
$14 = $t6	$30 = $fp
$15 = $t7	$31 = $ra

Arithmetic Operations

- Add and subtract, three operands
  - Two sources and one destination

```
add a, b, c  # a gets b + c
```

- All arithmetic operations have this form
- *Design Principle 1:* Simplicity favours regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

Arithmetic: Example

- C code

  f = (g + h) - (i + j);

- Compiled MIPS code

```
add $t0, $g, $h      # temp t0 = g + h
add $t1, $i, $j      # temp t1 = i + j
sub $f, $t0, $t1     # f = t0 - t1
```

Register Operand: Example

- C code

  f = (g + h) - (i + j);

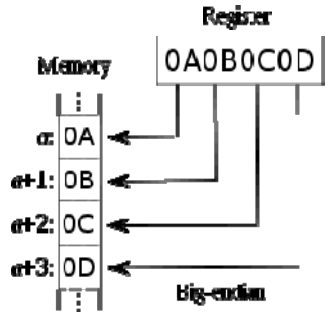
  - f, …, j in $s0, …, $s4

- Compiled MIPS code

```
add $t0, $s1, $s2        # temp t0 = s1 + s2
add $t1, $s3, $s4        # temp t1 = s3 + s4
sub $s0, $t0, $t1        # s0 = t0 + t1
```

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

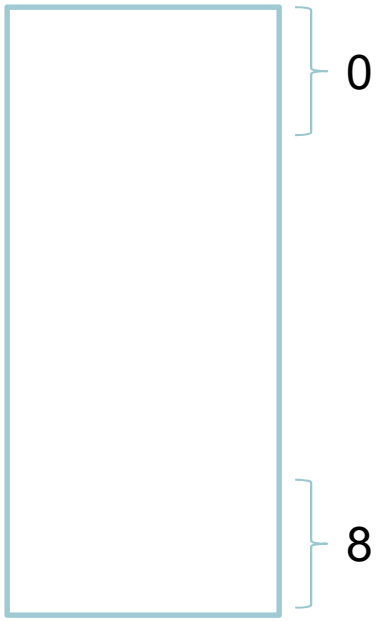


Memory Operand Example 1

- C code:

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

  - g in $s1, h in $s2, base address of A in $s3
- What is the MIPS assembly code?



Memory Operand: Example 1

- C code

  g = h + A[8];

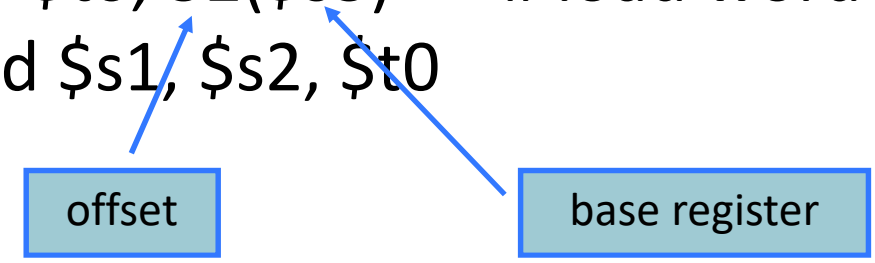
  - g in $s1, h in $s2, base address of A in $s3

- 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 (1/2)

- 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
```

Memory Operand: Example 2 (2/2)

- C code

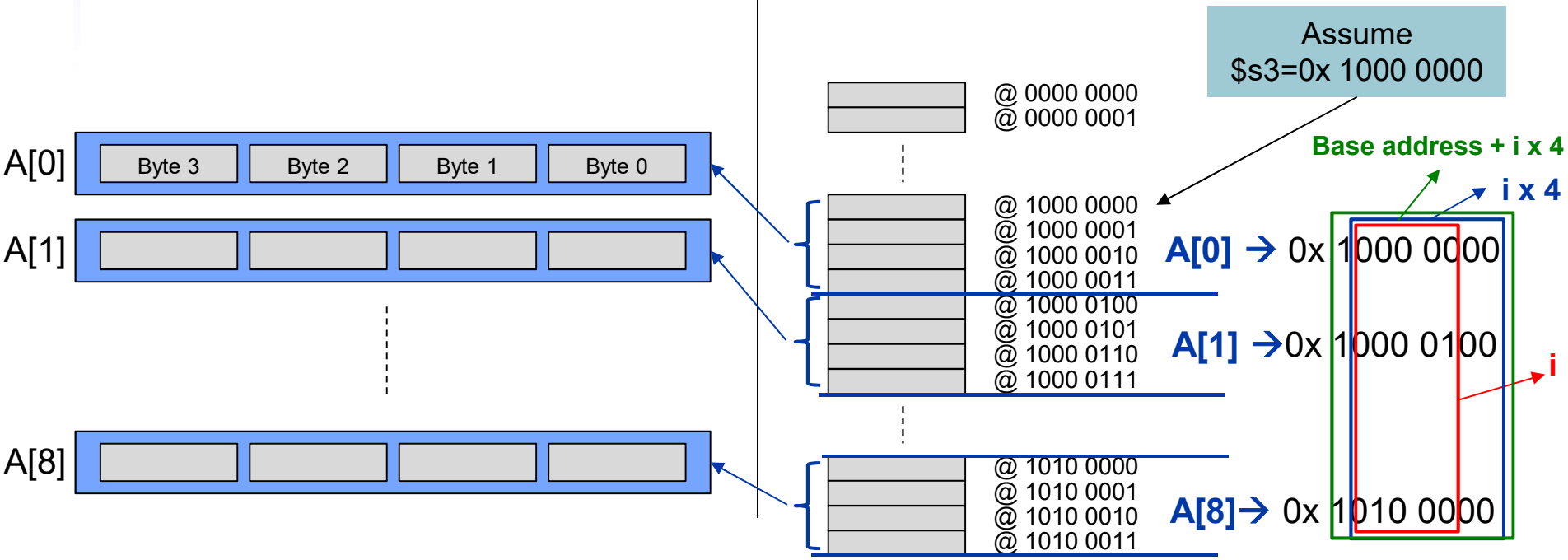
  A[12] = h + A[**8**]; // h in $s2, base address of A in $s3

- Compiled MIPS code //Index **8** requires offset of **32** (i.e. **4** x **8**)

```
lw  $t0, 32($s3)    # load word (i.e. 4 bytes)
    ...
```

My array in C : each data in the array corresponds to a **word**

My memory (hardware) : each address in the memory stores 1 **byte**



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 escape to memory for less frequently used variables
  - Register optimization is important!

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

The Constant Zero

- MIPS register 0 ($zero) is the constant 0
  - Cannot be overwritten
- Useful for common operations
  - E.g., move between registers

```
add $t2, $s1, $zero
```

Unsigned Binary Integers

- Given an n-bit number

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + ! \quad + x_1 2^1 + x_0 2^0$$

- Range: 0 to $+2^n - 1$
- Example
  - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
    $= 0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$
- Using 32 bits
  - 0 to +4,294,967,295

2s-Complement Signed Integers

- Given an n-bit number

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + ! \quad + x_1 2^1 + x_0 2^0$$

- Range: -2^{n-1} to $+2^{n-1} - 1$
- Example
  - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
    $= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^2 + 0\times2^1 + 0\times2^0$
    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$
- Using 32 bits
  - –2,147,483,648 to +2,147,483,647

2s-Complement Signed Integers

- Bit 31 is sign bit
  - 1 for negative numbers
  - 0 for non-negative numbers
- $-(-2^{n-1})$ can't be represented
- Non-negative numbers have the same unsigned and 2s-complement representation
- Some specific numbers
  - 0: 0000 0000 … 0000
  - –1: 1111 1111 … 1111
  - Most-negative: 1000 0000 … 0000
  - Most-positive: 0111 1111 … 1111

Sign Extension

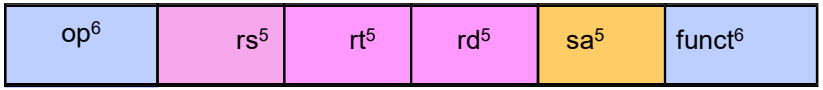
- Representing a number using more bits
  - Preserve the numeric value
- In MIPS instruction set
  - addi: extend immediate value
  - lb, lh: extend loaded byte/halfword
  - beq, bne: extend the displacement
- Replicate the sign bit to the left
  - c.f. unsigned values: extend with 0s
- E.g., 8-bit to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

Representing Instructions

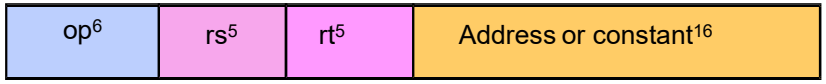
- Instructions are encoded in binary
  - Called machine code
- MIPS instructions
  - Encoded as 32-bit instruction words
  - Small number of formats encoding operation code (opcode), register numbers, …
  - Regularity!
- Register numbers
  - $t0 – $t7 are reg's 8 – 15
  - $t8 – $t9 are reg's 24 – 25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

MIPS Instruction Formats

- All instructions are 32-bit wide, Three instruction formats:
- Register (R-Type)
  - **Register-to-register** instructions
  - **Op**: **operation cod**e specifies the format of the instruction

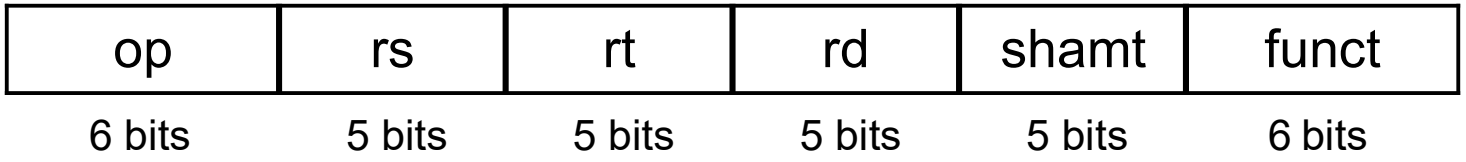


- Immediate (I-Type)
  - **16-bit immediate** constant is part in the instruction



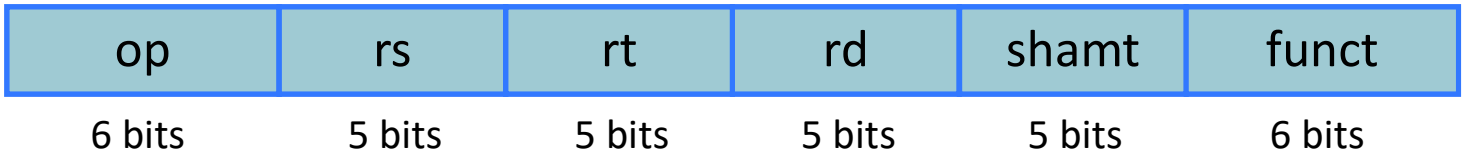
- Jump (J-Type)
  - Used by **jump** instructions

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)

R-format: Example



add $t0, $s1, $s2

special	$s1	$s2	$t0	0	add
0	17	18	8	0	32

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

$00000010001100100100000000100000_2 = 02324020_{16}$

Hexadecimal

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

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

- Example: eca8 6420
  - 1110 1100 1010 1000 0110 0100 0010 0000

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

I-format: Example

- lw $t0, 1200($t1)

special	$t1	$t0	1200
35	9	8	300
100011	01001	01000	0000 0100 1011 0000

$1000\ 1101\ 0010\ 1000\ 0000\ 0100\ 1011\ 0000_2 = 8d2804b0_{16}$

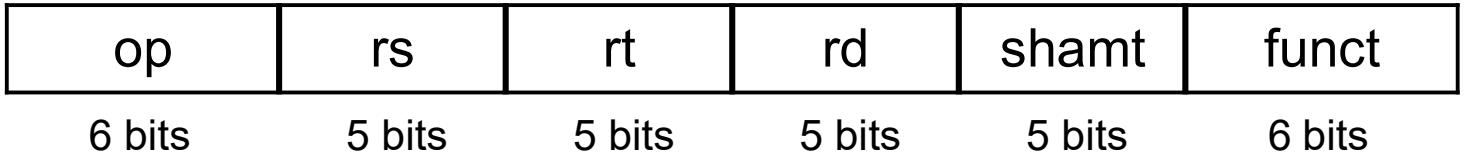
Logical Operations

- 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

Shift Operations

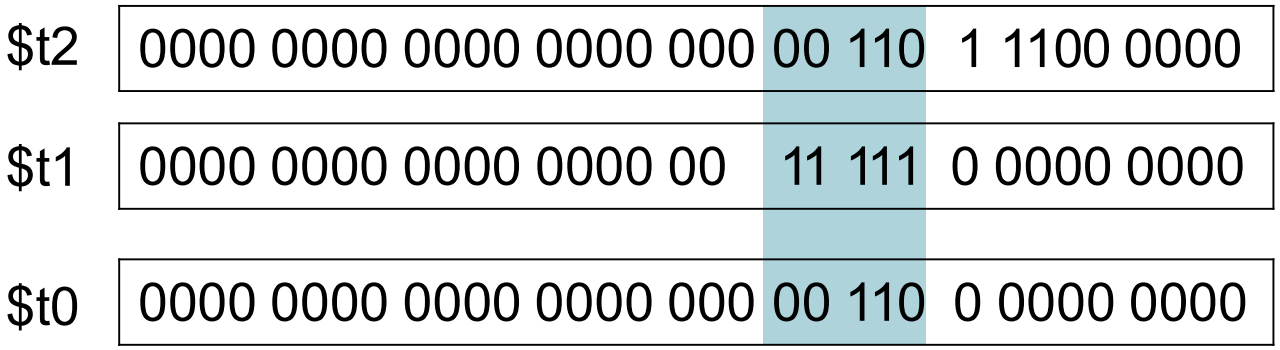


- shamt: how many positions to shift
- **sll**:Shift left logical
  - Shift left and fill with 0 bits
  - `sll` by *i* bits multiplies by 2^i
- **srl**:Shift right logical
  - Shift right and fill with 0 bits
  - `srl` by *i* bits divides by 2^i (unsigned only)

AND Operations

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

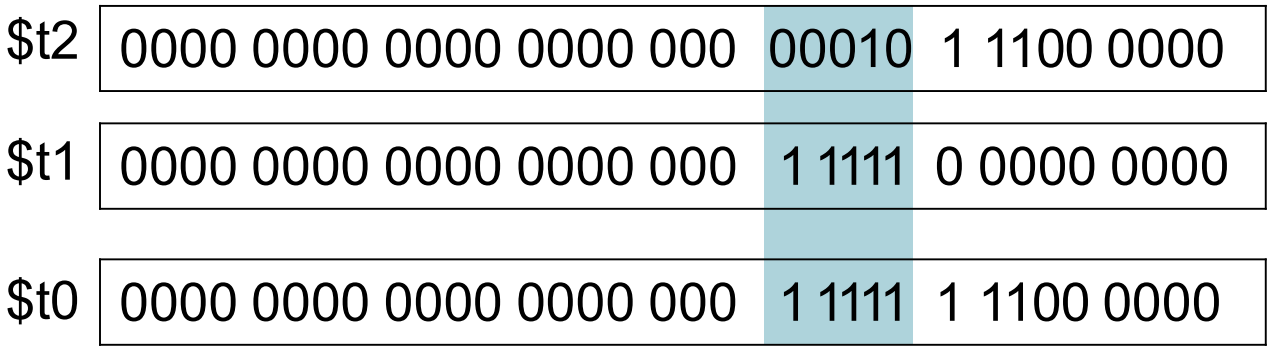
```
and $t0, $t1, $t2
```



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
$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

UnConditional Operation

- `j L1`
  - unconditional jump to instruction labeled L1


```
j    label      # jump to label
  . . .
label:
```

26-bit immediate value is stored in the instruction

- Immediate constant specifies address of target instruction

Program Counter (PC) is modified as follows:

- Next PC =
- Upper 4 most significant bits of PC are unchanged

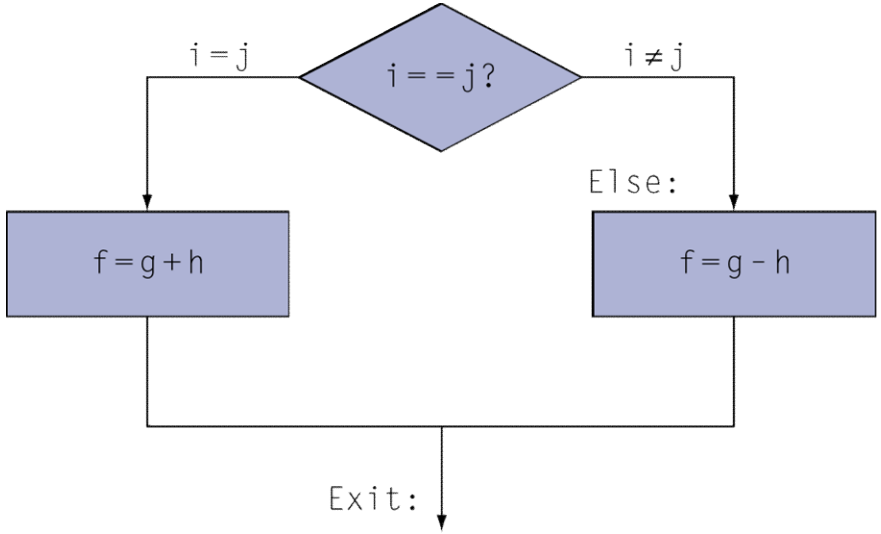
Compiling If Statements

- 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 (1/2)

- 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
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
Exit: …
```

Compiling Loop Statements (2/2)

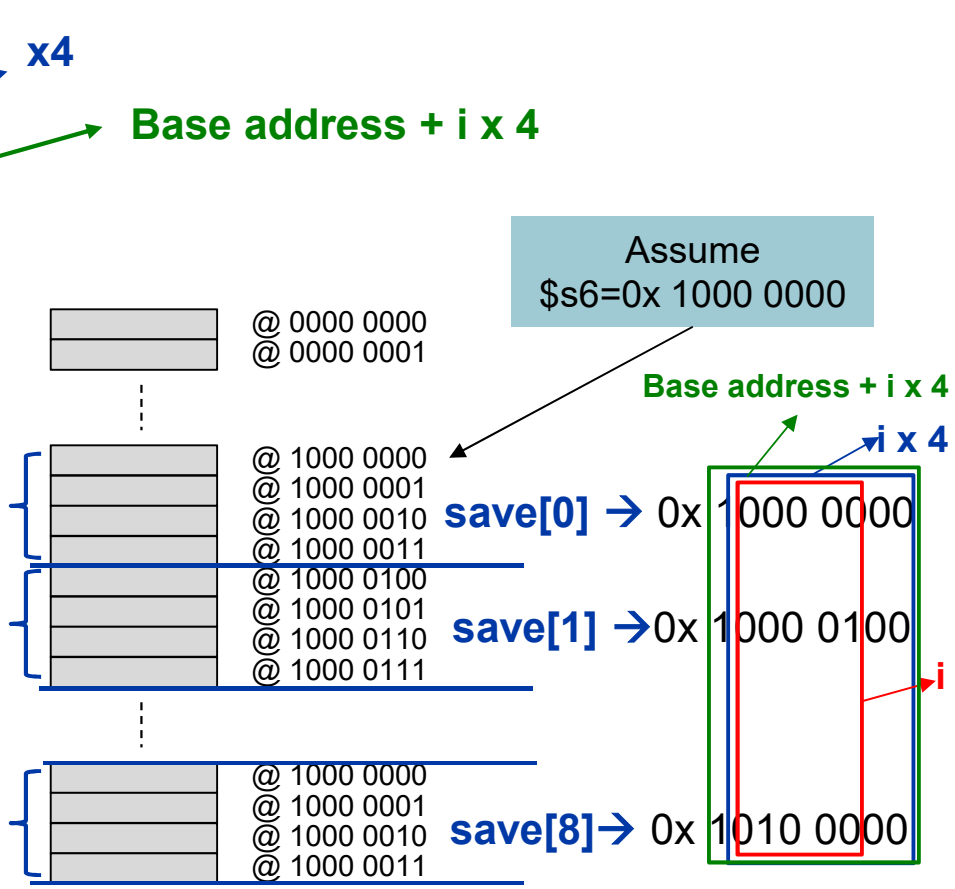
- C code

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

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

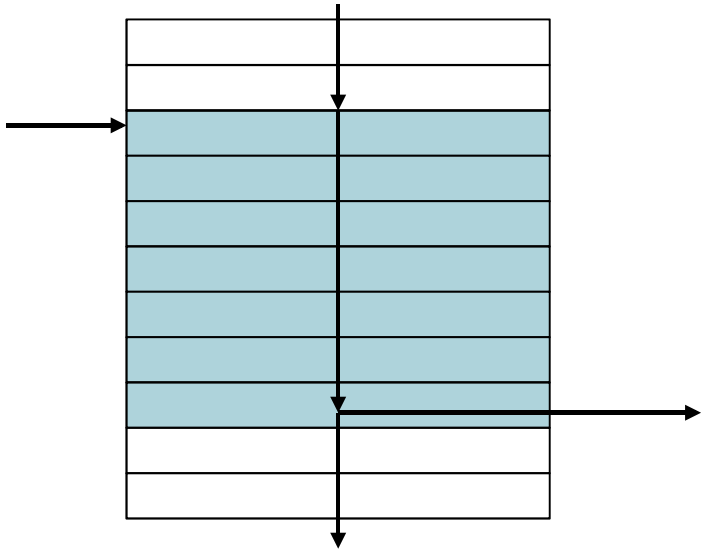
- Compiled MIPS code

```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 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
  - 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?
- MIPS compilers use slt, slti, beq, bne and $zero to create all relative conditions
- Hardware for <, ≥, … slower than =, ≠
  - Combining with branch involves more work per instruction, requiring a slower clock

- beq and bne are the common case
- Good design compromise

Pseudoinstructions: Example

- How would you implement in the assembler a logical-shift-left (LSL) pseudo-operation for a machine that didn't have this particular instruction? Be sure your LSL instruction can shift up to W-bits where W is the machine word size in bits
- Solution
  - Logical left shift operation corresponds to multiplication by 2 Implementing

    sll $s1, $s2, n

    can be done via the following loop:

    ```
    add $t0, $zero, $zero
    addi $t1, $zero, n
    add $s1, $s2, $zero
    loop:
    mul $s1, $s1, 2
    addi $t1, $t1, -1
    bne $t0, $t1, loop
    ```

    Here the mul instruction can easily be implemented using add operations (using a loop similar to above) if it is not provided natively in the instruction set

Signed vs. Unsigned

- Signed comparison: slt, slti
- Unsigned comparison: sltu, sltui
- E.g.,
  - $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

Summary

- Design principles
  1. Simplicity favors regularity
  2. Smaller is faster
  3. Make the common case fast
  4. Good design requires good compromises
- Layers of software/hardware
  - Compiler, assembler, hardware
- MIPS: typical of RISC ISAs
  - c.f. x86