

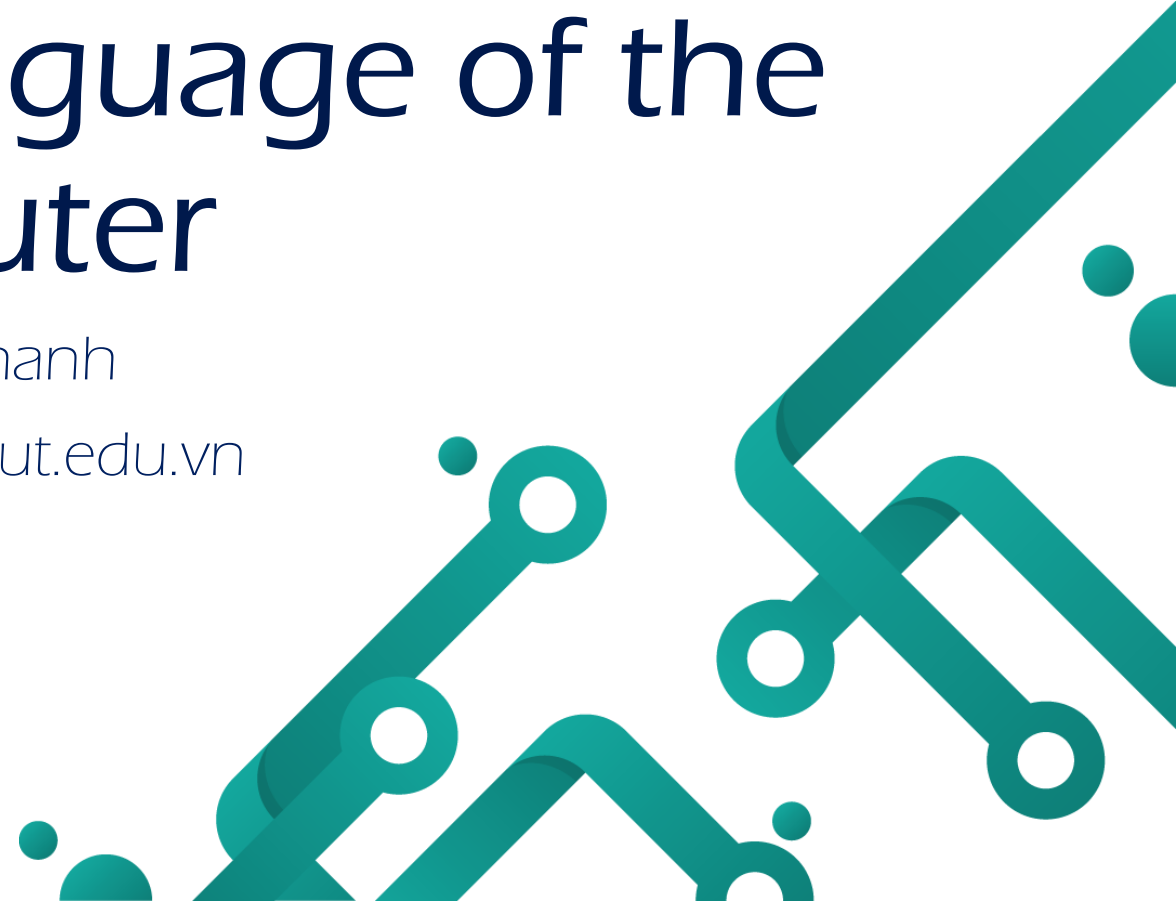


Computer Architecture  
Faculty of Computer Science & Engineering - HCMUT

# Chapter 2

## Instructions: Language of the Computer

Binh Tran-Thanh  
[thanhbinh@hcmut.edu.vn](mailto:thanhbinh@hcmut.edu.vn)



# Objectives

```
swap(int v[], int k){  
    int temp;  
    temp = v[k];  
    v[k] = v[k+1];  
    v[k+1] = temp;  
}
```

Compiler

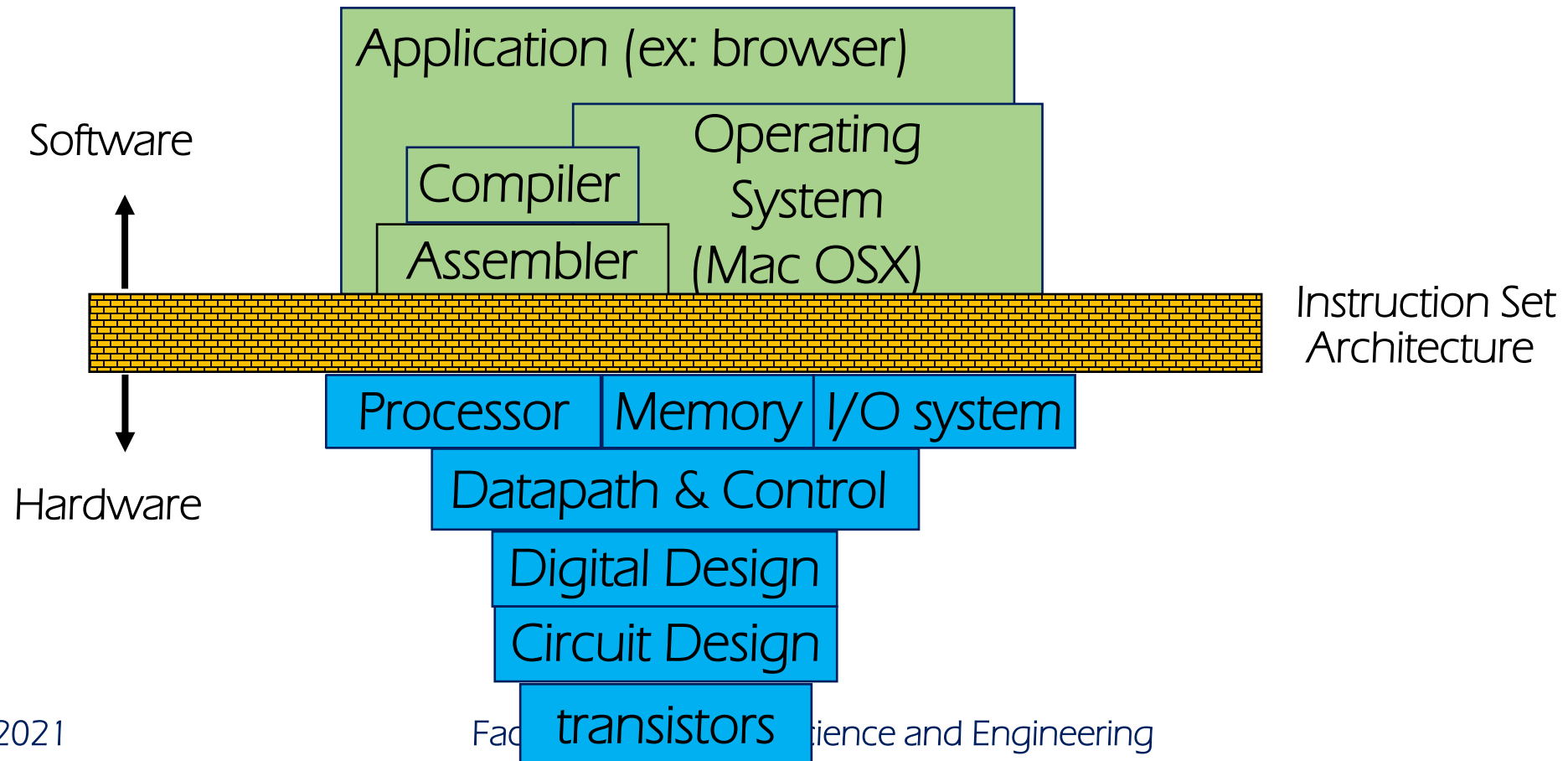
```
swap: multi $2, $5, 4  
      add $2, $4, $2  
      lw  $15, 0($2)  
      lw  $16, 4($2)  
      sw  $16, 0($2)  
      sw  $15, 4($2)  
      jr  $31
```

Assembler

```
000000001010001000000000100011000  
000000001000001000010000000100001  
100011011110001000000000000000000  
1000111000010010000000000000000100  
101011100001001000000000000000000  
1010110111100010000000000000000100  
00000011111000000000000000000001000
```

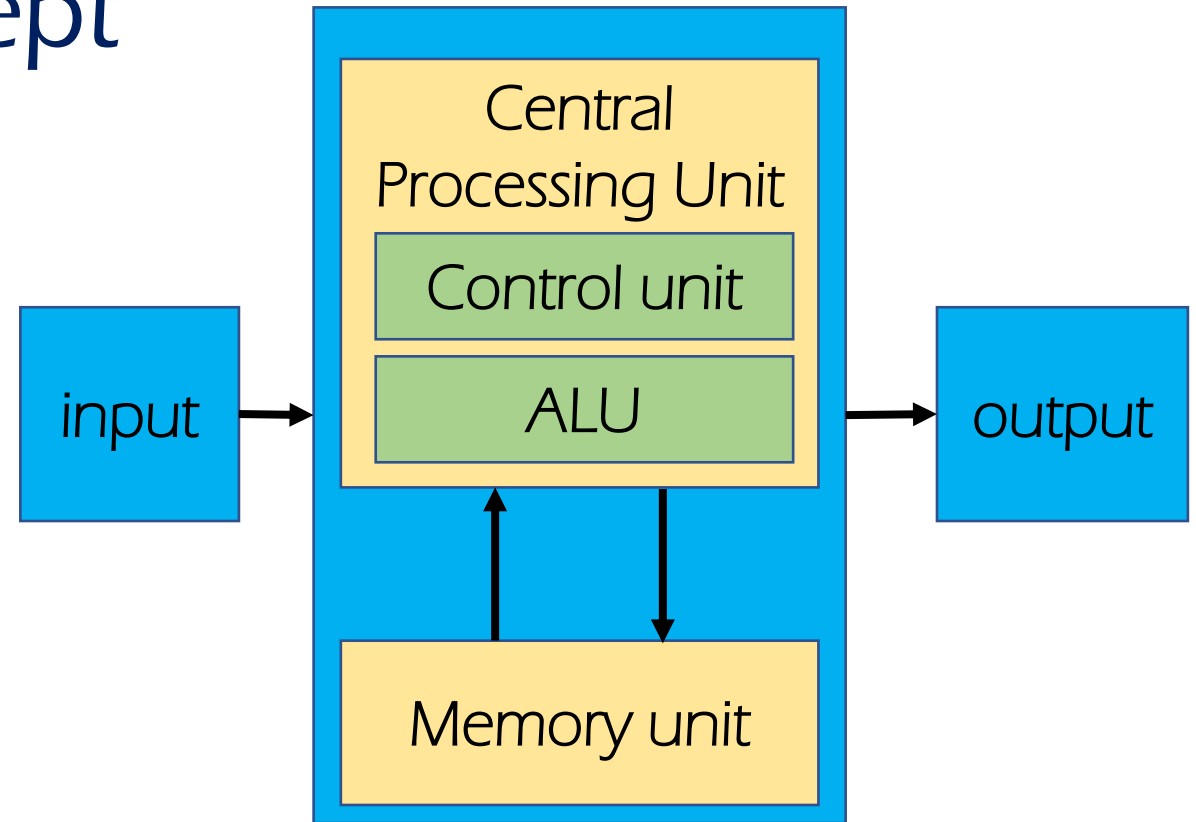
# Abstract layer of ISA

- Coordination of many levels (layers) of abstraction

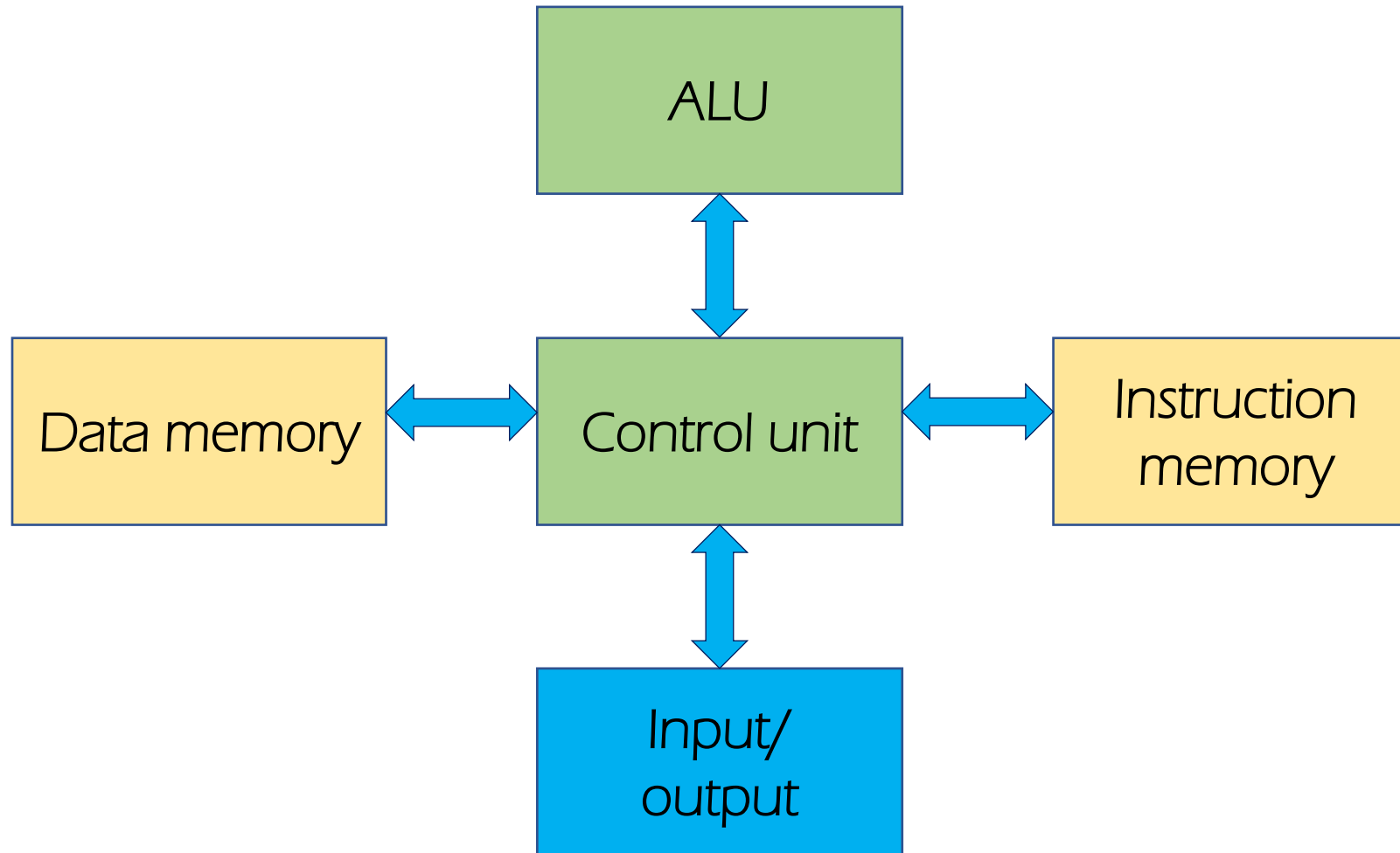


# Von Neumann Architecture

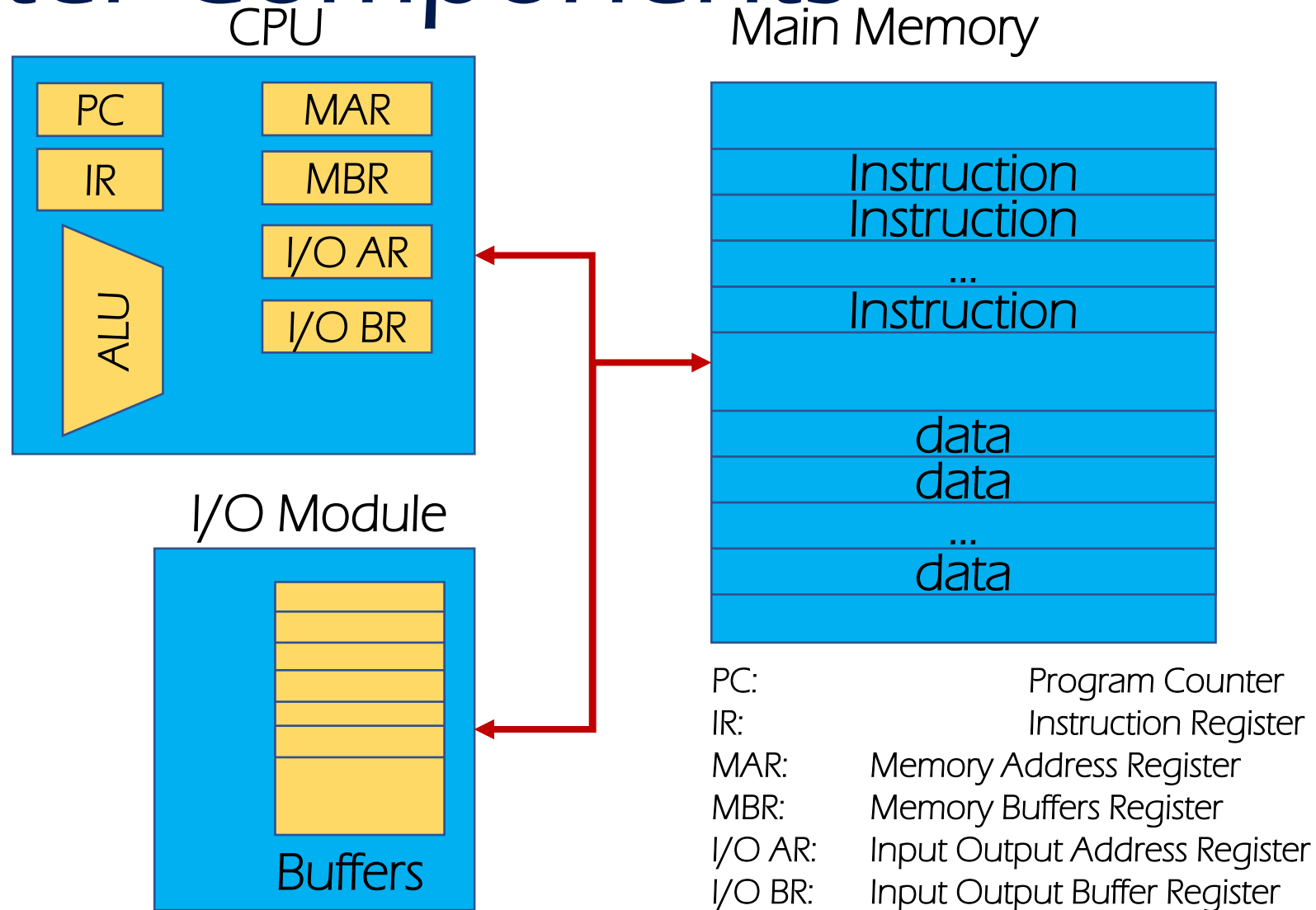
- Stored program concept
- Instruction category
  - Arithmetic
  - Data transfer
  - Logical
  - Conditional branch
  - Unconditional jump



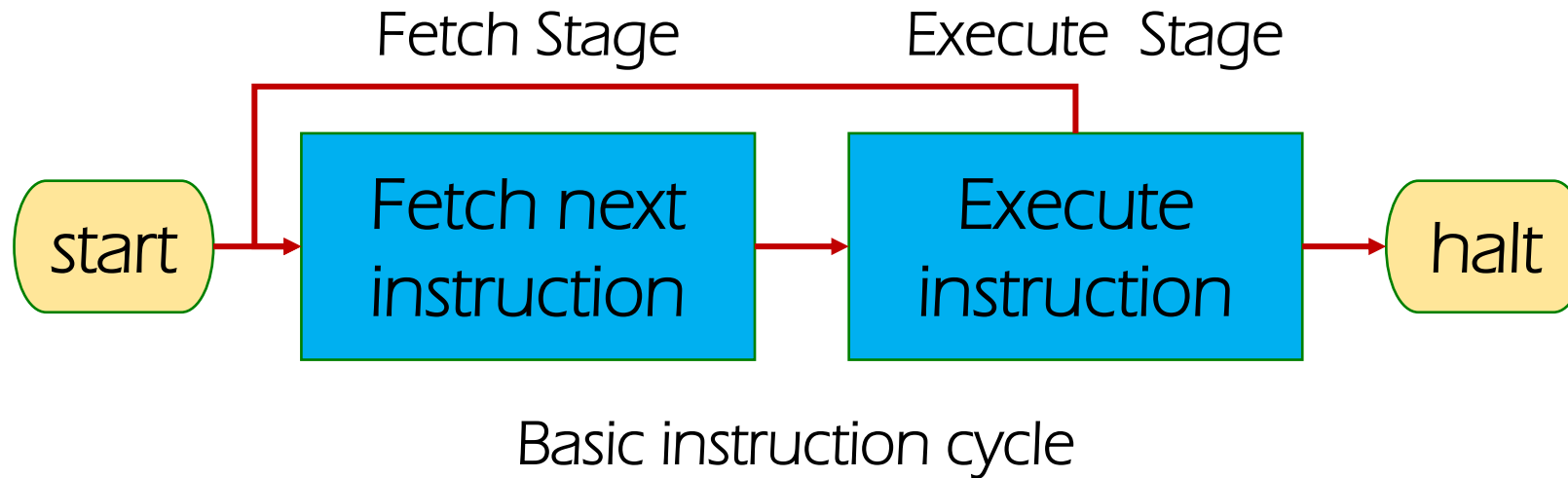
# Harvard architecture



# Computer Components



# Instruction execution process



- Fetch: from memory
  - PC increases after the fetch
  - PC holds the address of the next instruction
- Execution: Encode & Execution

# Instruction Set

- The repertoire 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



# RISC vs. CISC Architectures

## RISC

- Reduced Instruction Set Computers
- Emphasis on software
- Single-clock, reduced instruction only
- Low cycles per second, large code sizes
- Spends more transistors on memory registers

## CISC

- Complex Instruction Set Computers
- Emphasis on hardware
- Includes multi-clock, complex instructions
- Small code sizes, high cycles per second
- Transistors used for storing complex instructions

# The MIPS Instruction Set

- Used as the example throughout the book
- Stanford MIPS commercialized by MIPS Technologies ([www.mips.com](http://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

# Design Principles of ISA

01 Simplicity favors regularity

02 Smaller is faster

03 Make the common case fast

04 Good design demands good compromises



# 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 favors 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, $s1, $s2 # t0 = g + h  
add $t1, $s3, $s4 # t1 = i + j  
sub $s0, $t0, $t1 # f = t0 - t1
```

# Register Operands

- Arithmetic instructions use register operands
- MIPS has a  $32 \times 32$ -bit register file
  - Use for frequently accessed data
  - Numbered 0 to 31
  - 32-bit data called a “word”
- Assembler names
  - $\$t0, \$t1, \dots, \$t9$  for temporary values
  - $\$s0, \$s1, \dots, \$s7$  for saved variables
- **Design Principle 2:** Smaller is faster
  - c.f. main memory: millions of locations

# Register Operand Example

C code:

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

```
f, g, h, i, j in $s0, $s1, $s2,  
$s3, $s4, respectively
```

Compiled MIPS code:

```
add $t0, $s1, $s2 # t0 = g + h
```

```
add $t1, $s3, $s4 # t1 = i + j
```

```
sub $s0, $t0, $t1 # f = 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
```

Compiled MIPS code:

Index 8 requires offset of 32

4 bytes per word

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

base register

offset

# 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!

# 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
  - Move between registers
    - `add $a0, $t0, $zero` # move \$t0 to \$a0
  - Assign immediate to registers
    - `addi $a0, $zero, 100` # \$a0 = 100

# Unsigned Binary Integers

- Given an n-bit number

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

- Range: 0 to  $+2^n - 1$

- Example

- $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$   
 $= 0 + \dots + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$   
 $= 0 + \dots + 8 + 0 + 2 + 1 = 11_{10}$

- Using 32 bits

- 0 to +4,294,967,295

# 2s-Complement Signed Integers

- Given an n-bit number

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

- Range:  $-2^{n-1}$  to  $+2^{n-1} - 1$

- Example

- $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$   
 $= -1 \times 2^{31} + 1 \times 2^{30} + \dots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$   
 $= -2,147,483,648 + 2,147,483,644 = -4^{10}$

- 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



# Signed Negation

- Complement and add 1
  - Complement means  $1 \rightarrow 0, 0 \rightarrow 1$

$$x + \bar{x} = 1111\dots111_2 = -1$$

$$\bar{x} + 1 = -x$$

- Example: negate +2
  - $+2 = 0000 \ 0000 \ \dots \ 0010_2$
  - $-2 = 1111 \ 1111 \ \dots \ 1101_2 + 1$   
 $= 1111 \ 1111 \ \dots \ 1110_2$

# 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: extend 8-bit to 16-bit for signed number
  - +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

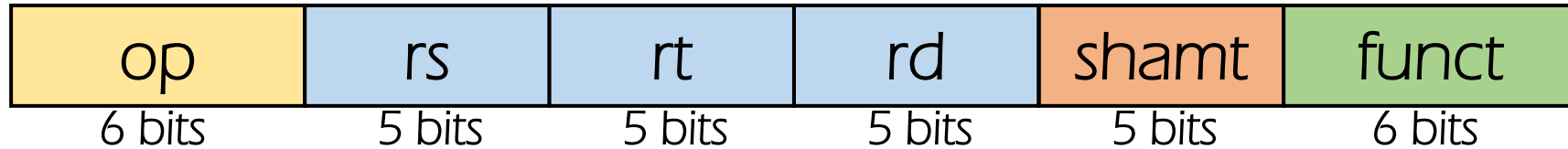
# Exercise

Given a piece of MIPS code as below:

```
.data
int_a: .word 0xCA002020
.text
    la    $s0, int_a
    lb    $t1, 0($s0)
    lbu   $t2, 0($s0)
    lb    $t3, 3($s0)
    lbu   $t4, 3($s0)
```

What are values of t1, t2, t3, t4?

# MIPS R-format Instructions

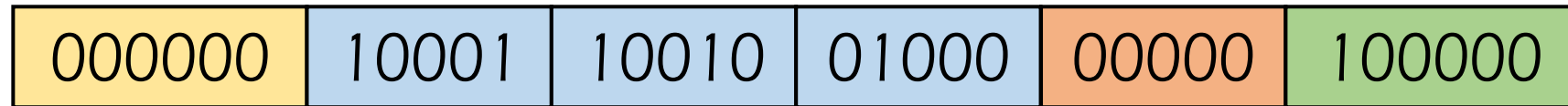
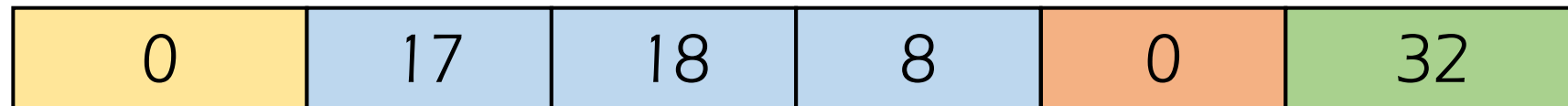
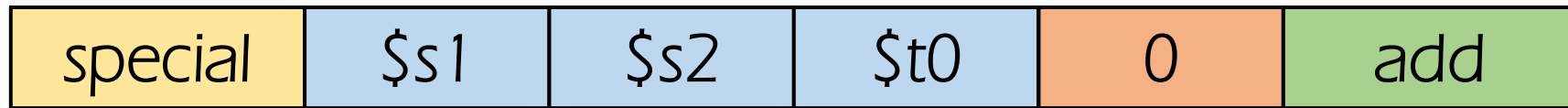


- Instruction fields
- op: operation code (opcode)
- rs: first source register number
- rt: second source register number
- rd: destination register number
- shamt: shift amount (00000 for now)
- funct: function code (extends opcode)

# R-format Example



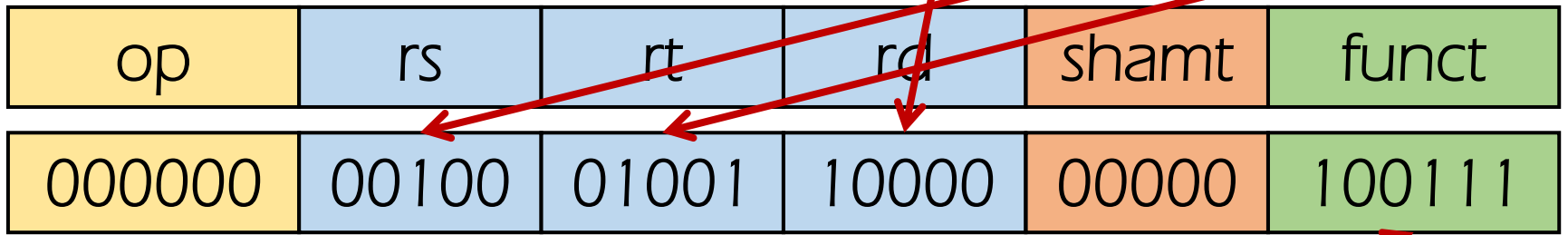
*add \$t0, \$s1, \$s2*



$$000000 \mathbf{10001} 10010 \mathbf{01000} 00000 \mathbf{100000}_2 = 02324020_{16}$$

# Exercise (MIPS to machine code)

- What is machine code of **`nor $s0, $a0, $t1`**



00000000001000110000000001001111<sub>2</sub> = ~~00898027~~<sub>16</sub>

Function	Code (Hex)	Function	Code (Hex)	Function	Code (Hex)	Function	Code (Hex)
Add	20	Sltu	2b	Mflo	12	Divu	1b
Addu	21	Srl	02	Mfc0	0	Mfhi	10
And	24	Sub	22	Mult	18	Or	25
Jump register	08	Subu	23	Multu	19	Slt	2A
Nor	27	Div	1A	Sra	03		

# Hexadecimal

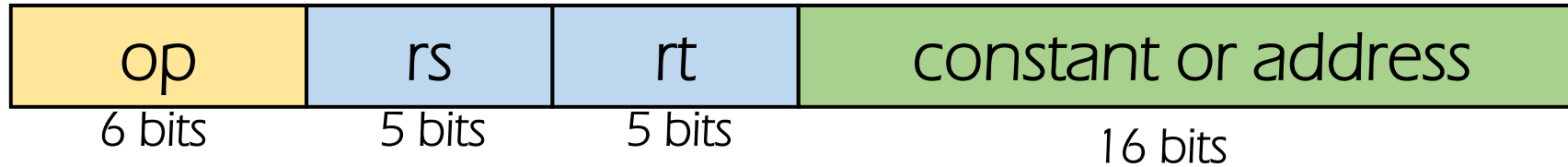
- Base 16
  - Compact representation of bit strings
  - 4 bits per hex digit

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: 0xCAFE FACE
  - 1100 1010 1111 1110 1111 1010 1100 1110



# MIPS I-format Instructions

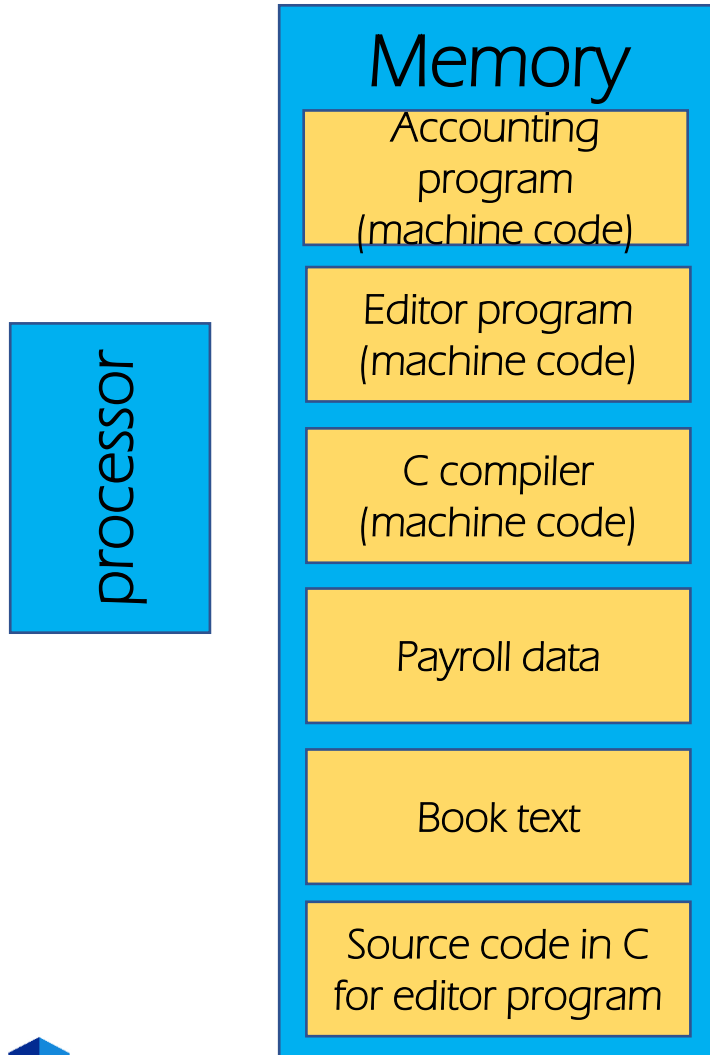


- Immediate arithmetic and load/store instructions
  - rt: destination or source register number
  - Constant:  $-2^{15} \rightarrow +2^{15} - 1$
  - Address: offset added to base address in  $\$rs$
- **Design Principle 4:** Good design demands good compromises
  - Different formats complicate decoding,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

# Exercise

- Given a MIPS instruction: `addi $s3, $s3, X`
- What is the maximum value of `X`?
- How do we assign `$s0 = 0x1234CA00 (= 305,449,472)`

# 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

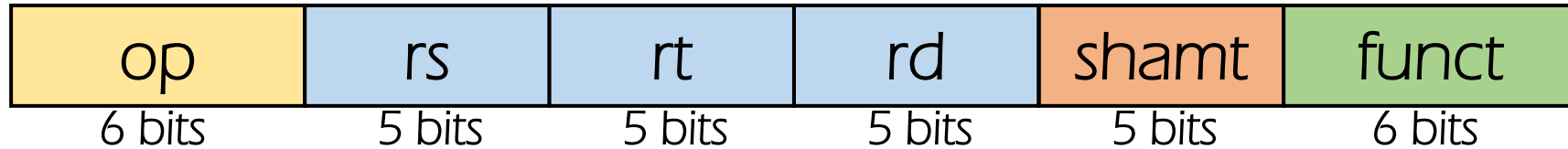
# Logical Operations

- 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
- 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)

# 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

# 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

# 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  
(\$zero): always  
read as zero

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



# Conditional Operations

- Branch to a labeled instruction if a condition is true. Otherwise, continue sequentially
- `beq rs, rt, Label`
  - if ( $rs == rt$ ) branch to instruction labeled;
- `bne rs, rt, Label`
  - if ( $rs \neq rt$ ) branch to instruction labeled;
- `j Label`
  - unconditional jump to instruction labeled;

# Compiling If Statement

C code:

```
int x, y;  
if (x < y) {  
    y = y - x;  
}  
■ x in $a0, y: $a1
```

MIPS assembly:

```
slt    $t0, $a0, $a1  
      beqz $t0, endif  
      sub  $a1, $a1, $a0  
endif:
```

# Compiling If-else Statement

C code:

```
int x, y;  
if (x < y) {  
    y = y - x;  
}else{  
    y = y * 4;  
}  
■ # x in $a0, y in $a1
```

MIPS assembly:

```
slt    $t0, $a0, $a1  
beqz   $t0, else  
sub    $a1, $a1, $a0  
j      end_if  
else:  
sll    $a1, $a1, 2  
end_if:
```

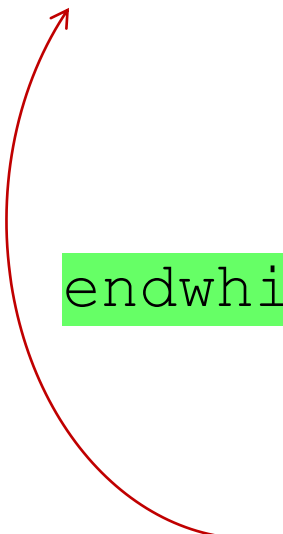
# Compiling Loop Statement Example

C code:

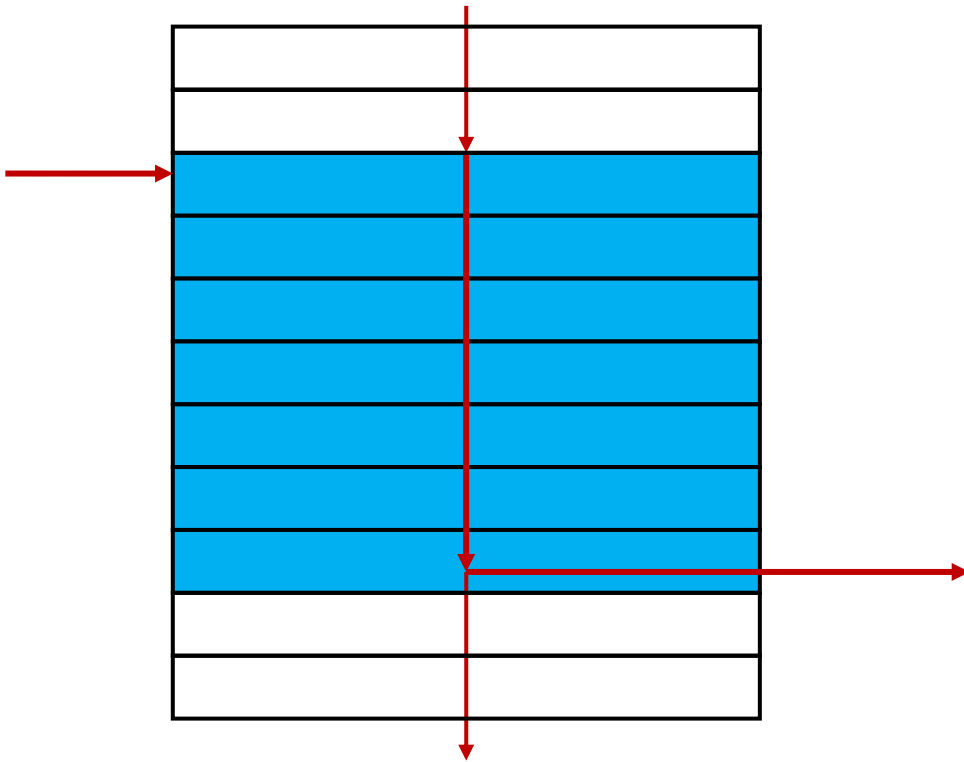
```
while (save[i] == k) {  
    i += 1;  
}  
  
i in $s3, k in $s5,  
address of save in $s6
```

MIPS assembly:

```
while: sll $t1, $s3, 2  
      add $t1, $t1, $s6  
      lw  $t0, 0($t1)  
      bne $t0, $s5,  
endwhile  
      addi $s3, $s3, 1  
      j while  
endwhile:
```



# 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 **then** rd = 1
  - **else** rd = 0
- **slti** rt, rs, constant
  - **if** rs < constant **then** 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?
- 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
  - (Design Principle 4)

# 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
  - $-1 < +1 \Rightarrow \$t0 = 1$
- **sltu** `$t0, $s0, $s1` # unsigned
  - $+4,294,967,295 > +1 \Rightarrow \$t0 = 0$



# Exercise

- Assume  $\$s0 = 0xCA002020$ .

- Given MIPS instruction:

**andi**  $\$t0, \$s0, 0xFFFF$

- Which instruction type does the given instruction belong to?

- What is value of  $\$t0$ ?

# Procedure Calling

- Steps required
- Place parameters in registers
- Transfer control to procedure
- Acquire storage for procedure
- Perform procedure's operations
- Place result in register for caller
- 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:			
<b>addi</b>	\$sp,	\$sp,	-4
<b>sw</b>	\$s0,	0 (\$sp)	<i># Save \$s0 on stack</i>
<b>add</b>	\$t0,	\$a0,	\$a1
<b>add</b>	\$t1,	\$a2,	\$a3 <i>#Procedure body</i>
<b>sub</b>	\$s0,	\$t0,	\$t1
<b>add</b>	\$v0,	\$s0,	\$zero <i># Result</i>
<b>lw</b>	\$s0,	0 (\$sp)	<i># Restore \$s0</i>
<b>addi</b>	\$sp,	\$sp,	4
<b>jr</b>	\$ra		<i># Return</i>

# 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

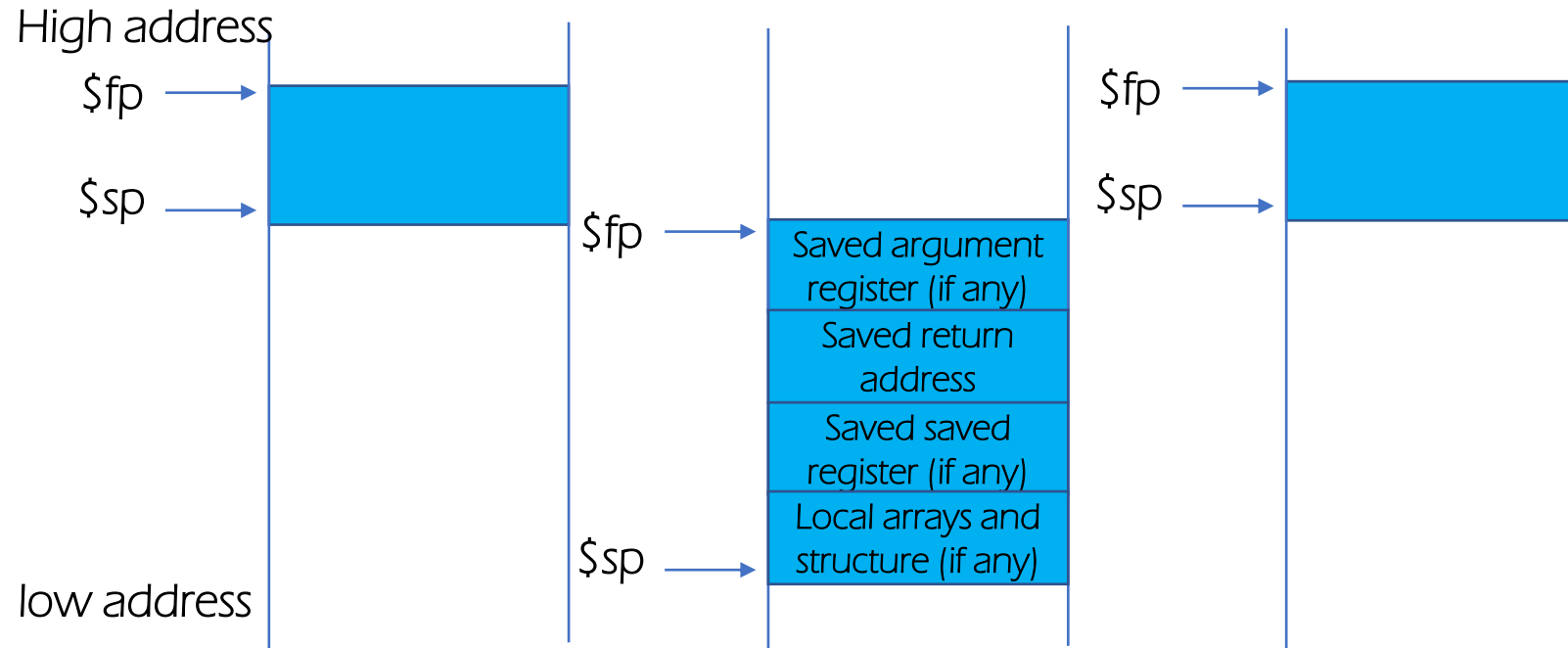


# Non-Leaf Procedure Example

MIPS:

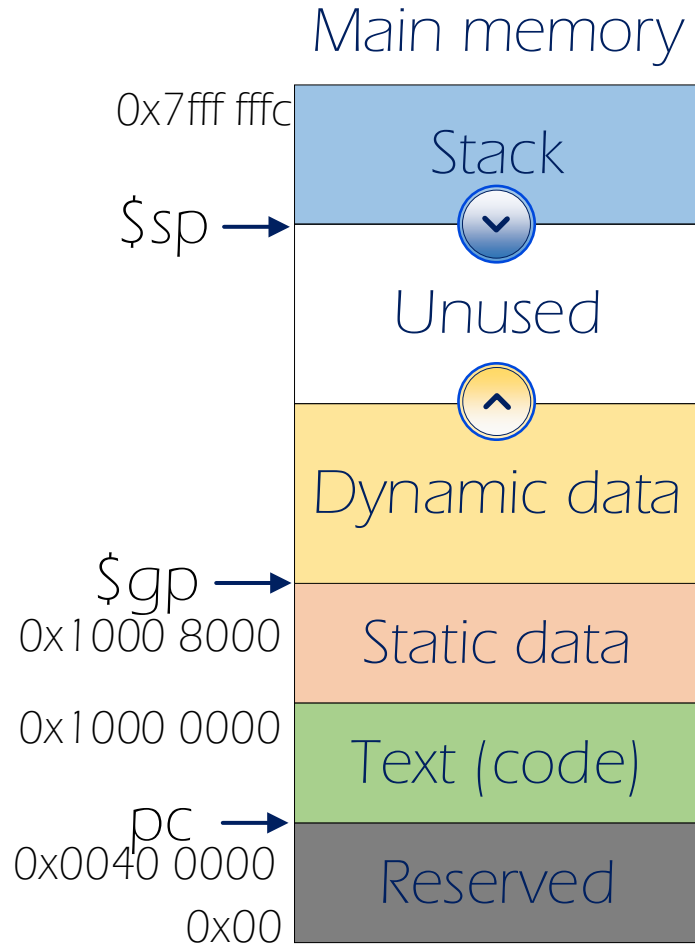
fact:
<b>addi</b> \$sp, \$sp, -8    # adjust stack for 2 items
<b>sw</b> \$ra, 4(\$sp)    # save return address
<b>sw</b> \$a0, 0(\$sp)    # save argument
<b>slti</b> \$t0, \$a0, 1    # test for n < 1
<b>beq</b> \$t0, \$zero, L1
<b>addi</b> \$v0, \$zero, 1   # if so, result is 1
<b>addi</b> \$sp, \$sp,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

# Memory Layout



- Text: program code
- Static data: global variables
  - e.g., static variables in C, constant arrays and strings
  - \$gp initialized to address allowing  $\pm$ offsets into this segment
- Dynamic data: heap
  - E.g., malloc in C, new in Java
- Stack: automatic storage

# Character Data

- Byte-encoded character sets
  - ASCII: 128 characters
    - 95 graphic, 33 control
  - Latin-1: 256 characters
    - ASCII, +96 more graphic characters
- Unicode: 32-bit character set
  - Used in Java, C++ wide characters, ...
  - Most of the world's alphabets, plus symbols
  - UTF-8, UTF-16: variable-length encodings

# 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

# String Copy Example

C code (naïve):

- Null-terminated string

```
void strcpy (char x[], char y[]) {  
    int i;  
    i = 0;  
    while ( (x[i]=y[i]) != '\0' )  
        i += 1;  
}
```

- Addresses of x, y in \$a0, \$a1
- i in \$s0

# String Copy Example

## MIPS code:

strcpy:

```
    addi $sp, $sp, -4      # adjust stack for item
    sw    $s0, 0($sp)      # save $s0
    add   $s0, $zero, $zero # i = 0
L1: add   $t1, $s0, $a1     # addr of y[i] in $t1
    lbu   $t2, 0($t1)      # $t2 = y[i]
    add   $t3, $s0, $a0     # addr of x[i] in $t3
    sb    $t2, 0($t3)      # x[i] = y[i]
    beq   $t2, $zero, L2   # exit loop if y[i]== 0
    addi  $s0, $s0, 1      # i = i + 1
    j     L1              # next iteration of loop
L2: lw    $s0, 0($sp)      # restore saved $s0
    addi  $sp, $sp, 4      # pop 1 item from stack
    jr    $ra              # and return
```

# 32-bit Constants

- Most constants are small
  - 16-bit immediate is sufficient
- For the occasional 32-bit constant
  - lui rt, constant**
    - Copies 16-bit constant to left 16 bits of rt
    - Clears right 16 bits of rt to 0

Below shows how to assign 32 bits constant (4 000 000 DEC) to a register (s0)

4 000 000 Dec = 3D 0900 HEX;

3D hex = 60 DEC, 0900 Hex = 2304 Dec

lui \$s0, 61

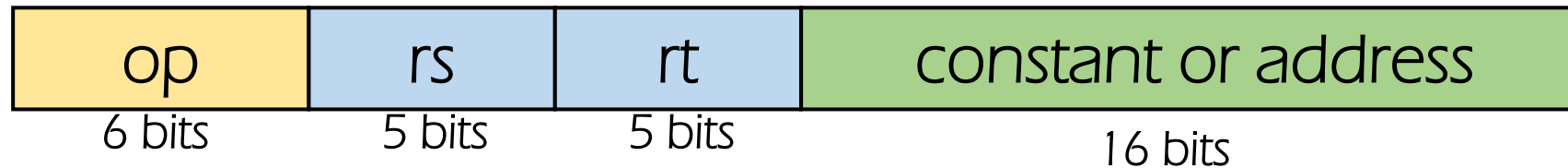
ori \$s0, \$s0, 2304

0000 0000 0011 1101	0000 0000 0000 0000
0000 0000 0011 1101	0000 1001 0000 0000



# Branch Addressing

- Branch instructions specify
  - Opcode, two registers, target address
- Most branch targets are near branch
  - Forward or backward



- PC-relative addressing
  - **Target address = PC + 4 + offset × 4**
  - PC already incremented by 4 by this time

# Jump Addressing

- Jump (j and jal) targets could be anywhere in text segment

- Encode full address in instruction



- (Pseudo)Direct jump addressing
  - **Target address = PC31...28 : (address × 4)**

# Target Addressing Example

- Loop code from earlier example
  - Assume Loop at location 80000

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: ...
    
```

Address

Instruction memory

80000	0	0	19	9	4	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2		
80016	8	19	19	1		
80020	2	20000				
80024						

# Branching Far Away

- If branch target is too far to encode with 16-bit offset, assembler rewrites the code
- Example

**beq** \$s0, \$s1, L1

↓

**bne** \$s0, \$s1, L2

**j** L1

L2:

# Addressing Mode Summary

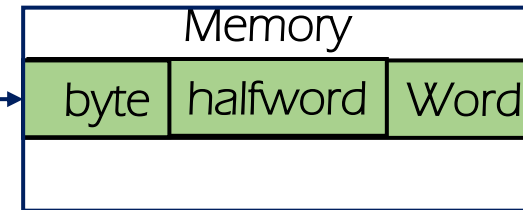
## 1. Immediate addressing



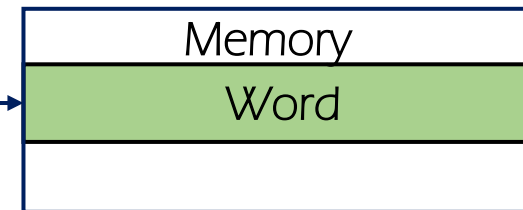
## 2. Register addressing



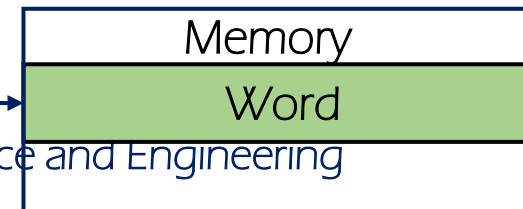
## 3. Immediate addressing



## 4. Pc-relative addressing



## 5. Pseudo-direct addressing



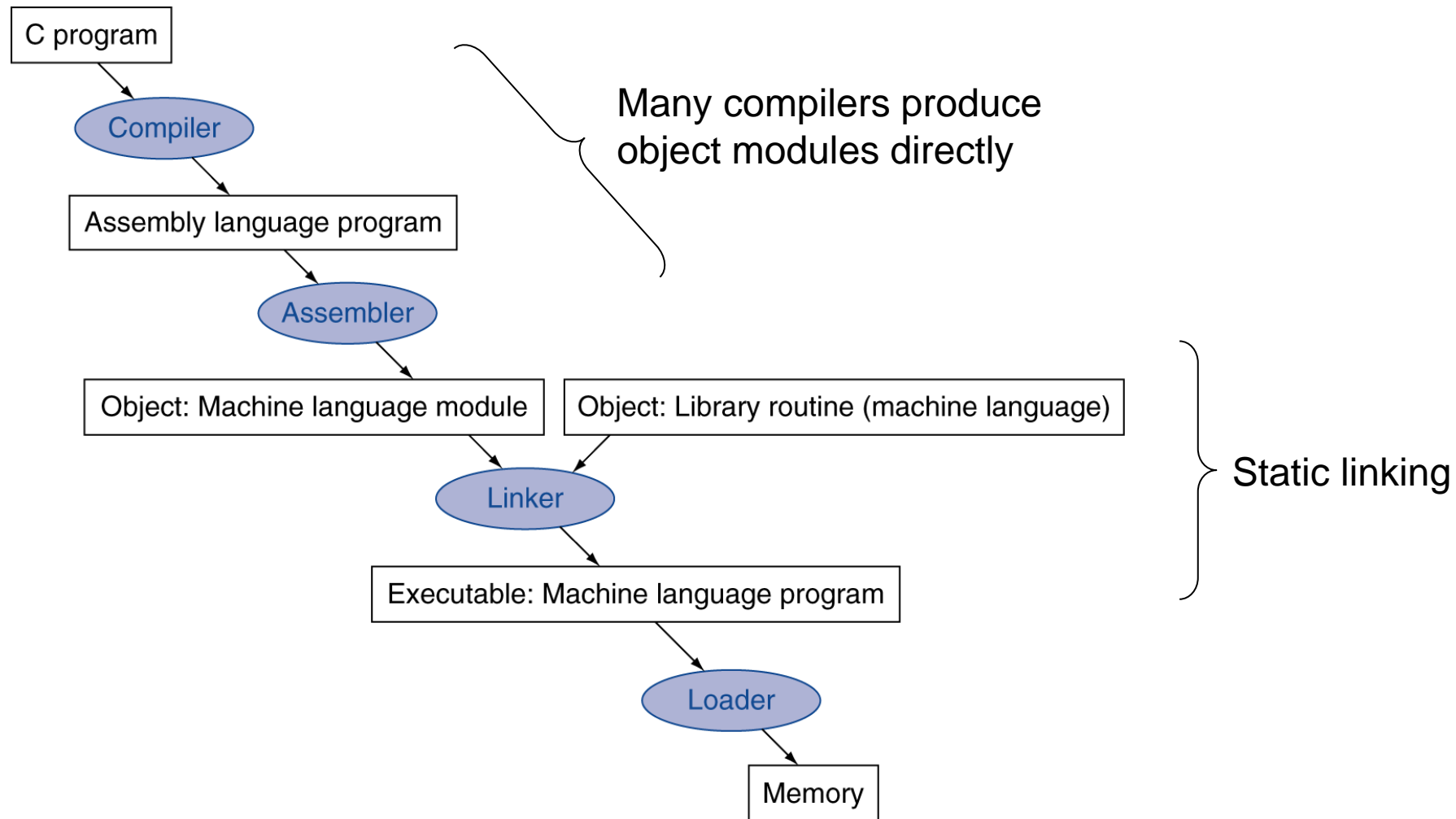
# Synchronization

- Two processors sharing an area of memory
  - P1 writes, then P2 reads
  - Data race if P1 and P2 don't synchronize
    - Result depends of order of accesses
- Hardware support required
  - Atomic read/write memory operation
  - No other access to the location allowed between the read and write
- Could be a single instruction
  - E.g., atomic swap of register  $\Leftrightarrow$  memory
  - Or an atomic pair of instructions

# Synchronization in MIPS

- Load linked: `ll rt, offset(rs)`
- Store conditional: `sc rt, offset(rs)`
  - Succeeds if location not changed since the `ll`
    - Returns 1 in `rt`
  - Fails if location is changed
    - Returns 0 in `rt`
- Example: atomic swap (to test/set lock variable)  
try: `add $t0, $zero, $s4 # copy exchange value`  
`ll $t1, 0($s1) # load linked`  
`sc $t0, 0($s1) # store conditional`  
`beq $t0, $zero, try # branch store fails`  
`add $s4, $zero, $t1 # put load value in $s4`

# Translation and Startup





# Assembler Pseudo-instructions

- Most assembler instructions represent machine instructions one-to-one
- Pseudo-instructions: figments of the assembler's imagination

**move** \$t0, \$t1 → **add** \$t0, \$zero, \$t1  
**blt** \$t0, \$t1, L → **slt** \$at, \$t0, \$t1  
**bne** \$at, \$zero, L

\$at (register 1): assembler temporary

# Producing an Object Module

- Assembler (or compiler) translates program into machine instructions
- Provides information for building a complete program from the pieces
  - Header: described contents of object module
  - Text segment: translated instructions
  - Static data segment: data allocated for the life of the program
  - Relocation info: for contents that depend on absolute location of loaded program
  - Symbol table: global definitions and external refs
  - Debug info: for associating with source code

# Linking Object Modules

- Produces an executable image
  - Merges segments
  - Resolve labels (determine their addresses)
  - Patch location-dependent and external refs
- Could leave location dependencies for fixing by a relocating loader
  - But with virtual memory, no need to do this
  - Program can be loaded into absolute location in virtual memory space

# Loading a Program

- Load from image file on disk into memory
  - 1. Read header to determine segment sizes
  - 2. Create virtual address space
  - 3. Copy text and initialized data into memory
    - Or set page table entries so they can be faulted in
  - 4. Set up arguments on stack
  - 5. Initialize registers (including \$sp, \$fp, \$gp)
  - 6. Jump to startup routine
    - Copies arguments to \$a0, ... and calls main
    - When main returns, do exit syscall

# Dynamic Linking

- Only link/load library procedure when it is called
  - Requires procedure code to be relocatable
  - Avoids image bloat caused by static linking of all (transitively) referenced libraries
  - Automatically picks up new library versions

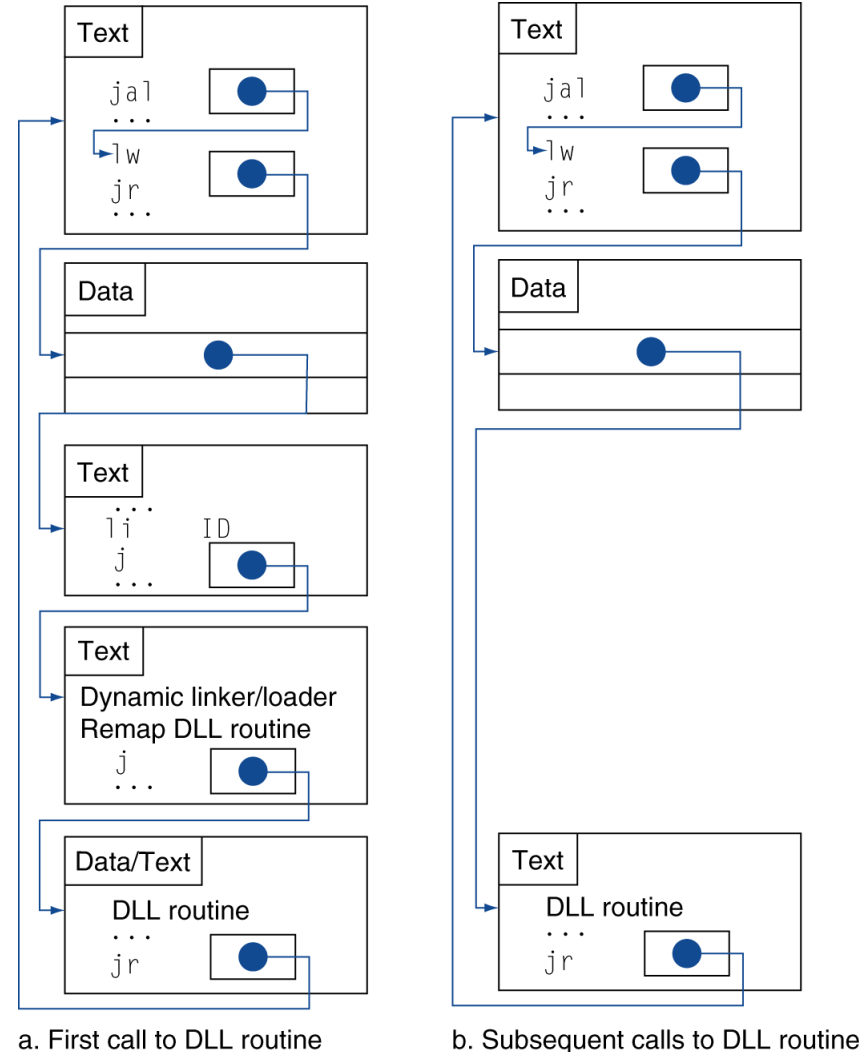
# Lazy Linkage

Indirection table

Stub: Loads routine ID,  
Jump to linker/loader

Linker/loader code

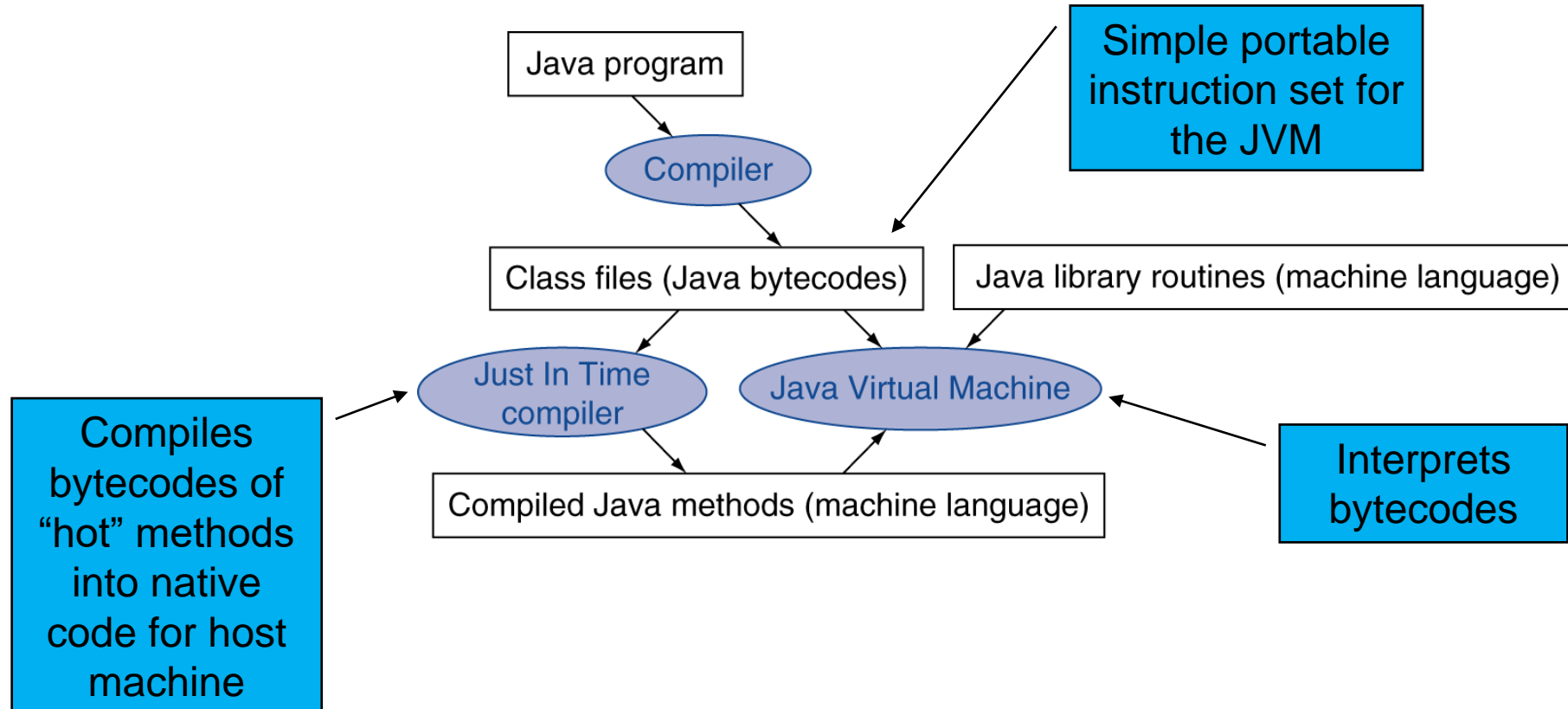
Dynamically  
mapped code



a. First call to DLL routine

b. Subsequent calls to DLL routine

# Starting Java Applications



# C Sort Example

- Illustrates use of assembly instructions for a C bubble sort function

- Swap procedure (leaf)

```
void swap(int v[], int k) {  
    int temp;  
    temp = v[k];  
    v[k] = v[k+1];  
    v[k+1] = temp;  
}
```

- **v, k, temp in \$a0, \$a1, and \$t0, respectively**



# The Procedure Swap

swap:

```
sll $t1, $a1, 2      # $t1 = k * 4
add $t1, $a0, $t1    # $t1 = v + (k*4)
                    # (address of v[k])
lw  $t0, 0($t1)      # $t0 (temp) = v[k]
lw  $t2, 4($t1)      # $t2 = v[k+1]
sw  $t2, 0($t1)      # v[k] = $t2 (v[k+1])
sw  $t0, 4($t1)      # v[k+1] = $t0 (temp)
jr  $ra              #return to calling routine
```

# The Sort Procedure in C

Non-leaf (calls swap)

```
void sort (int v[], int n) {  
    int i, j;  
    for (i = 0; i < n; i += 1) {  
        for (j = i - 1;  
             j >= 0 && v[j] > v[j + 1];  
             j -= 1) {  
            swap(v, j);  
        }  
    }  
}
```

- **v, k, temp in \$a0, \$a1, and \$t0, respectively**

# The Procedure Body

```

        move $s2, $a0          # save $a0 into $s2
        move $s3, $a1          # save $a1 into $s3
        move $s0, $zero        # i = 0
for1tst: slt  $t0, $s0, $s3      # $t0 = 0 if $s0 ≥ $s3 (i ≥ n)
        beq  $t0, $zero, exit1  # go to exit1 if $s0 ≥ $s3 (i ≥ n)
        addi $s1, $s0, -1       # j = i - 1
for2tst: slti $t0, $s1, 0        # $t0 = 1 if $s1 < 0 (j < 0)
        bne  $t0, $zero, exit2  # go to exit2 if $s1 < 0 (j < 0)
        sll  $t1, $s1, 2        # $t1 = j * 4
        add  $t2, $s2, $t1      # $t2 = v + (j * 4)
        lw   $t3, 0($t2)        # $t3 = v[j]
        lw   $t4, 4($t2)        # $t4 = v[j + 1]
        slt  $t0, $t4, $t3      # $t0 = 0 if $t4 ≥ $t3
        beq  $t0, $zero, exit2  # go to exit2 if $t4 ≥ $t3
        move $a0, $s2           # 1st param of swap is v (old $a0)
        move $a1, $s1           # 2nd param of swap is j
        jal  swap               # call swap procedure
        addi $s1, $s1, -1       # j -= 1
        j    for2tst           # jump to test of inner loop
exit2:  addi $s0, $s0, 1        # i += 1
        j    for1tst           # jump to test of outer loop

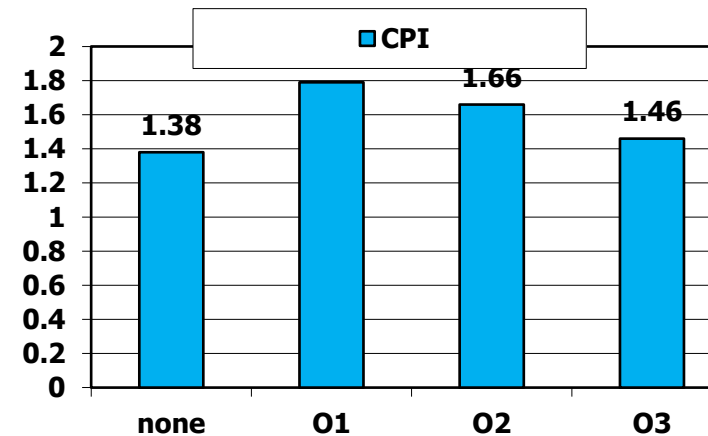
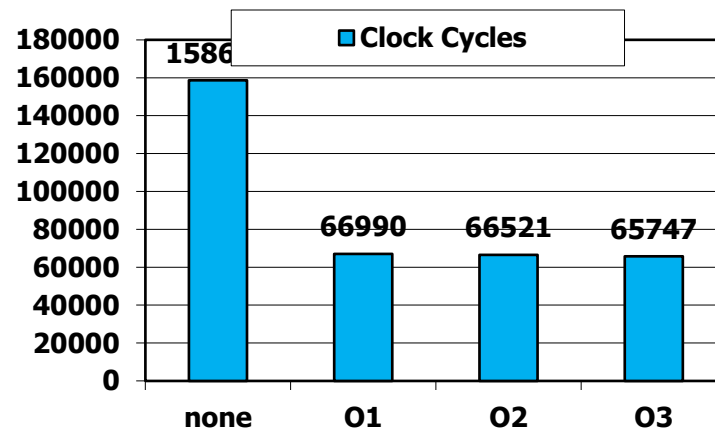
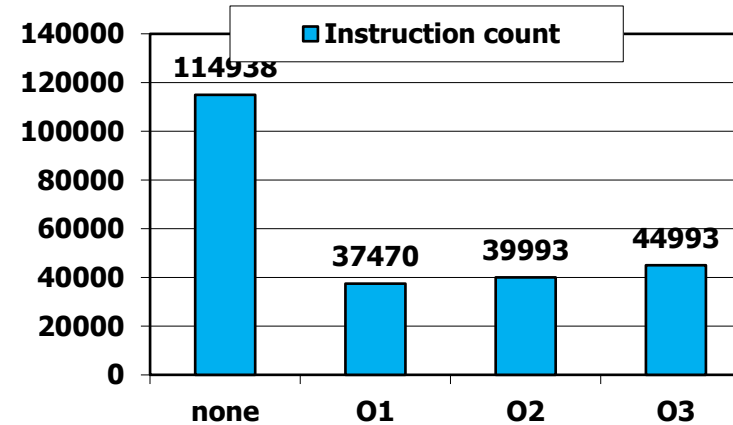
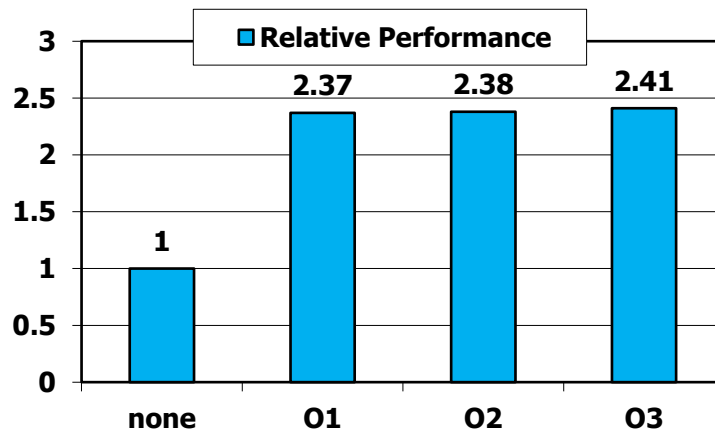
```

# The Full Procedure

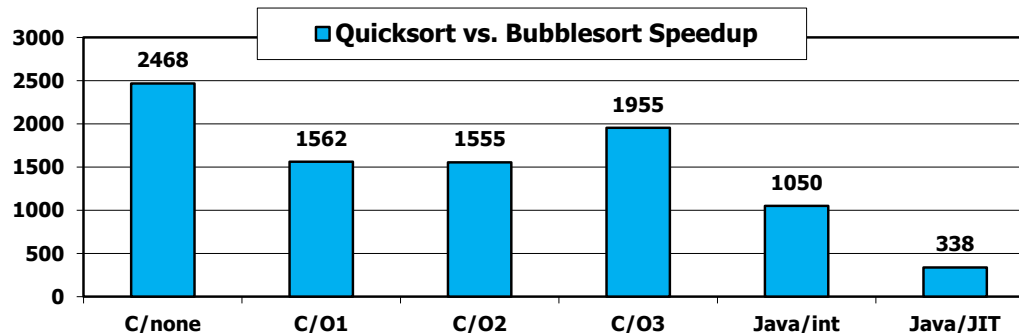
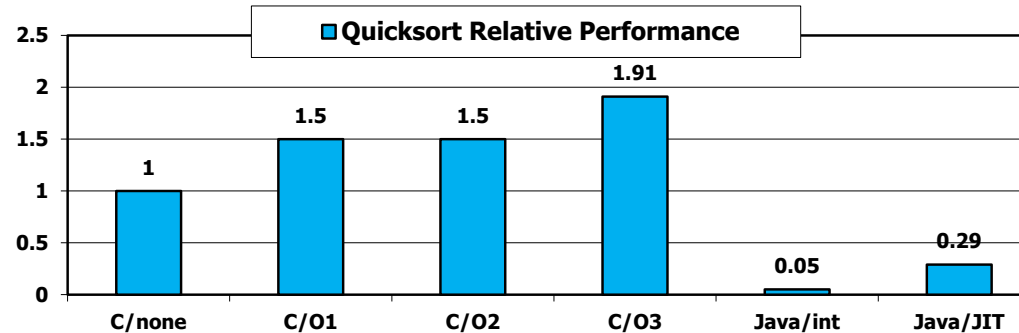
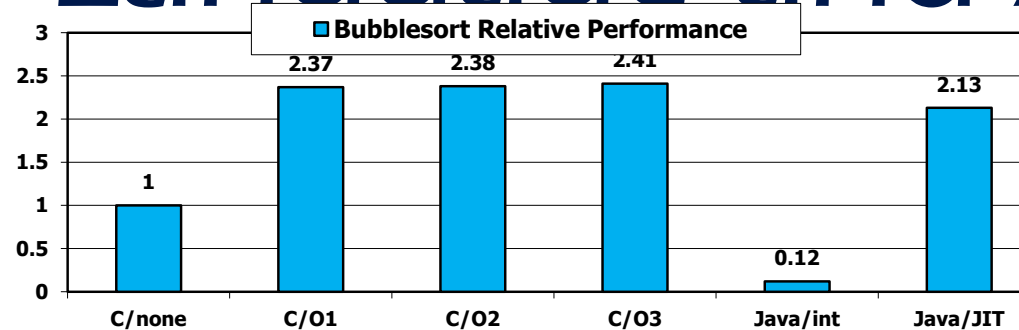
```
sort:      addi $sp,$sp, -20      # make room on stack
                                     # for 5 registers
sw $ra, 16($sp)      # save $ra on stack
sw $s3, 12($sp)      # save $s3 on stack
sw $s2, 8($sp)       # save $s2 on stack
sw $s1, 4($sp)       # save $s1 on stack
sw $s0, 0($sp)       # save $s0 on stack
...        # procedure body
...
exit1: lw $s0, 0($sp)  # restore $s0 from stack
lw $s1, 4($sp)       # restore $s1 from stack
lw $s2, 8($sp)       # restore $s2 from stack
lw $s3, 12($sp)      # restore $s3 from stack
lw $ra, 16($sp)      # restore $ra from stack
addi $sp,$sp, 20     # restore stack pointer
jr $ra              # return to calling routine
```

# Effect of Compiler Optimization

Compiled with gcc for Pentium 4 under Linux



# Effect of Language and Algorithm



# Lessons Learnt

- Instruction count and CPI are not good performance indicators in isolation
- Compiler optimizations are sensitive to the algorithm
- Java/JIT compiled code is significantly faster than JVM interpreted
  - Comparable to optimized C in some cases
- Nothing can fix a dumb algorithm!

# Arrays vs. Pointers

- Array indexing involves
  - Multiplying index by element size
  - Adding to array base address
- Pointers correspond directly to memory addresses
  - Can avoid indexing complexity



# Example: Clearing and Array

```
clear1(int array[], int size) {  
    int i;  
    for (i = 0; i < size; i += 1)  
        array[i] = 0;  
}
```

```
loop1: move $t0,$zero    # i = 0  
      sll $t1,$t0,2      # $t1 = i * 4  
      add $t2,$a0,$t1    # $t2 = &array[i]  
      sw $zero, 0($t2)   # array[i] = 0  
      addi $t0,$t0,1     # i = i + 1  
      slt $t3,$t0,$a1    # $t3 = (i < size)  
      bne $t3,$zero,loop1 # if (...) goto loop1
```

```
clear2(int *array, int size) {  
    int *p;  
    for (p = &array[0]; p < &array[size];  
          p = p + 1)  
        *p = 0;  
}
```

```
      move $t0,$a0      # p = & array[0]  
      sll $t1,$a1,2      # $t1 = size * 4  
      add $t2,$a0,$t1    # $t2 =  
                          # &array[size]  
loop2: sw $zero,0($t0)   # Memory[p] = 0  
      addi $t0,$t0,4     # p = p + 4  
      slt $t3,$t0,$t2    # $t3 = (p<&array[size])  
      bne $t3,$zero,loop2 # if (...) goto loop2
```

# Comparison of Array vs. Ptr

- Multiply “strength reduced” to shift
- Array version requires shift to be inside loop
  - Part of index calculation for incremented  $i$
  - c.f. incrementing pointer
- Compiler can achieve same effect as manual use of pointers
  - Induction variable elimination
  - Better to make program clearer and safer

# ARM & MIPS Similarities

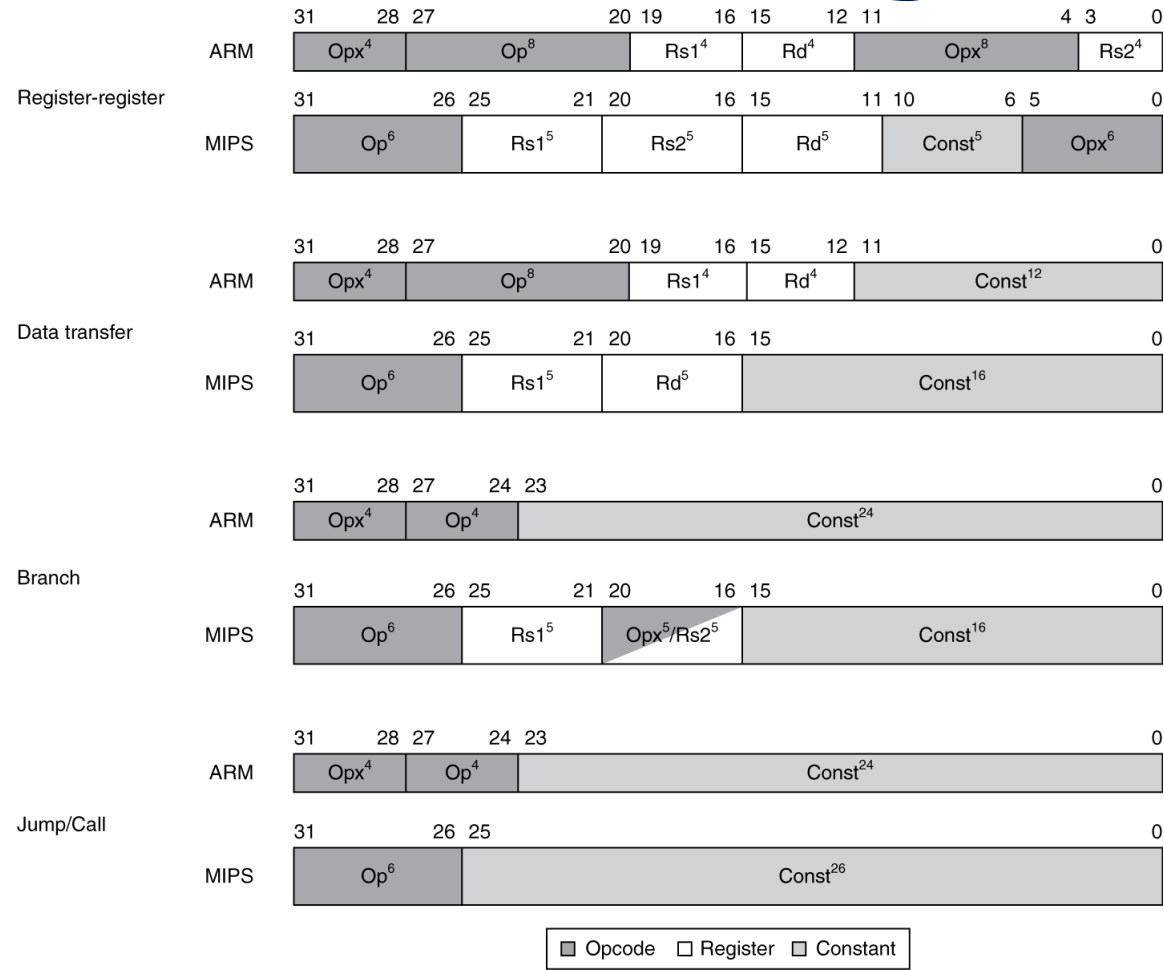
- ARM: **the most popular embedded core**
- Similar basic set of instructions to MIPS

	ARM	MIPS
Date announced	1985	1985
Instruction size	32 bits	32 bits
Address space	32-bit flat	32-bit flat
Data alignment	Aligned	Aligned
Data addressing modes	9	3
Registers	15 × 32-bit	31 × 32-bit
Input/output	Memory mapped	Memory mapped

# Compare and Branch in ARM

- Uses condition codes for result of an arithmetic/logical instruction
  - Negative, zero, carry, overflow
  - Compare instructions to set condition codes without keeping the result
- Each instruction can be conditional
  - Top 4 bits of instruction word: condition value
  - Can avoid branches over single instructions

# Instruction Encoding



# The Intel x86 ISA

- Evolution with backward compatibility
  - 8080 (1974): 8-bit microprocessor
    - Accumulator, plus 3 index-register pairs
  - 8086 (1978): 16-bit extension to 8080
    - Complex instruction set (CISC)
  - 8087 (1980): floating-point coprocessor
    - Adds FP instructions and register stack
  - 80286 (1982): 24-bit addresses, MMU
    - Segmented memory mapping and protection
  - 80386 (1985): 32-bit extension (now IA-32)
    - Additional addressing modes and operations
    - Paged memory mapping as well as segments

# The Intel x86 ISA

- Further evolution...
  - i486 (1989): pipelined, on-chip caches and FPU
    - Compatible competitors: AMD, Cyrix, ...
  - Pentium (1993): superscalar, 64-bit datapath
    - Later versions added MMX (Multi-Media eXtension) instructions
    - The infamous FDIV bug
  - Pentium Pro (1995), Pentium II (1997)
    - New microarchitecture (see Colwell, The Pentium Chronicles)
  - Pentium III (1999)
    - Added SSE (Streaming SIMD Extensions) and associated registers
  - Pentium 4 (2001)
    - New microarchitecture
    - Added SSE2 instructions

# The Intel x86 ISA

- And further...
  - **AMD64 (2003): extended architecture to 64 bits**
  - EM64T – Extended Memory 64 Technology (2004)
    - AMD64 adopted by Intel (with refinements)
    - Added SSE3 instructions
  - Intel Core (2006)
    - Added SSE4 instructions, virtual machine support
  - **AMD64 (announced 2007): SSE5 instructions**
    - **Intel declined to follow, instead...**
  - Advanced Vector Extension (announced 2008)
    - Longer SSE registers, more instructions
- If Intel didn't extend with compatibility, its competitors would!
  - Technical elegance ≠ market success



# Basic x86 Addressing Modes

- Two operands per instruction

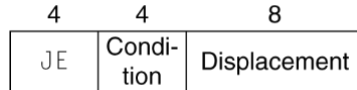
Source/dest operand	Second source operand
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate

- Memory addressing modes

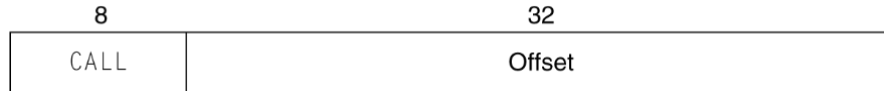
- Address in register
- $\text{Address} = \text{Rbase} + \text{displacement}$
- $\text{Address} = \text{Rbase} + 2^{\text{scale}} \times \text{Rindex}$  (scale = 0, 1, 2, or 3)
- $\text{Address} = \text{Rbase} + 2^{\text{scale}} \times \text{Rindex} + \text{displacement}$

# x86 Instruction Encoding

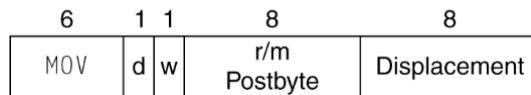
a. JE EIP + displacement



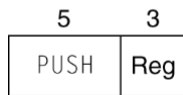
b. CALL



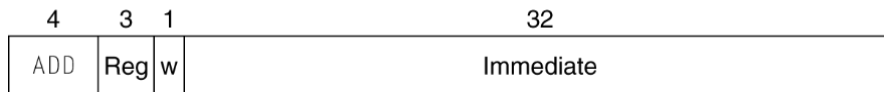
c. MOV EBX, [EDI + 45]



d. PUSH ESI



e. ADD EAX, #6765



f. TEST EDX, #42



- Variable length encoding
  - Postfix bytes specify addressing mode
  - Prefix bytes modify operation
    - Operand length, repetition, locking, ...

# Implementing IA-32

- Complex instruction set makes implementation difficult
  - Hardware translates instructions to simpler microoperations
    - Simple instructions: 1–1
    - Complex instructions: 1–many
  - Microengine similar to RISC
  - Market share makes this economically viable
- Comparable performance to RISC
  - Compilers avoid complex instructions

# ARM v8 Instructions

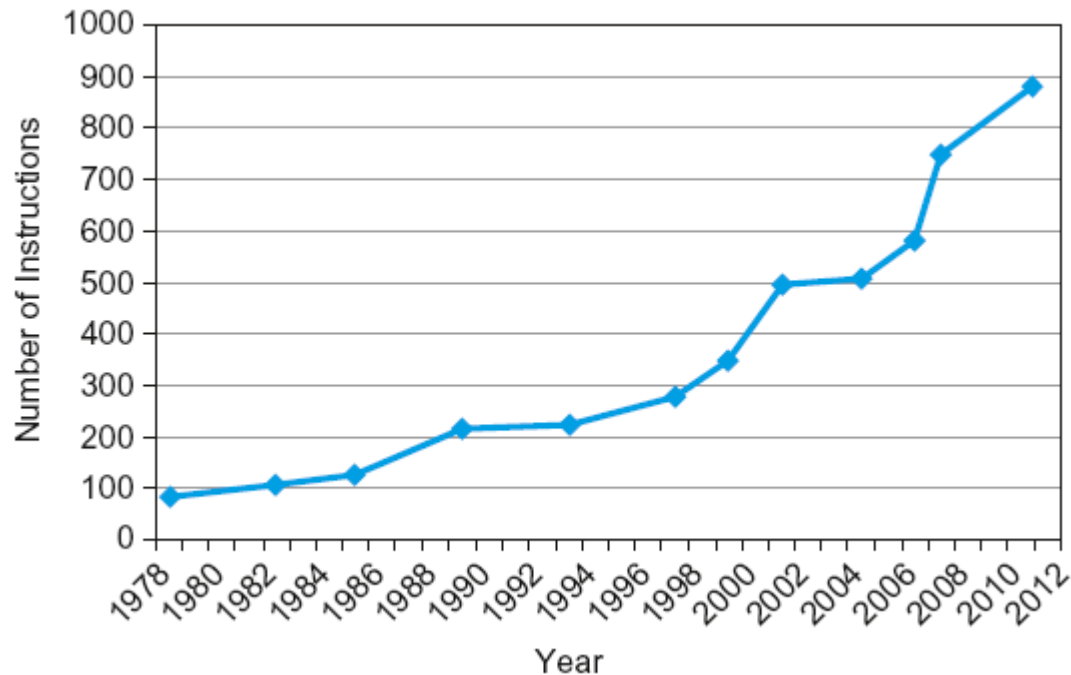
- In moving to 64-bit, ARM did a complete overhaul
- ARM v8 resembles MIPS
  - Changes from v7:
    - No conditional execution field
    - Immediate field is 12-bit constant
    - Dropped load/store multiple
    - PC is no longer a GPR
    - GPR set expanded to 32
    - Addressing modes work for all word sizes
    - Divide instruction
    - Branch if equal/branch if not equal instructions

# Fallacies

- Powerful instruction  $\Rightarrow$  higher performance
  - Fewer instructions required
  - But complex instructions are hard to implement
    - May slow down all instructions, including simple ones
  - Compilers are good at making fast code from simple instructions
- Use assembly code for high performance
  - But modern compilers are better at dealing with modern processors
  - More lines of code  $\Rightarrow$  more errors and less productivity

# Fallacies

- Backward compatibility  $\Rightarrow$  instruction set doesn't change
  - But they do accrete more instructions



x86 instruction set

# Pitfalls

- Sequential words are not at sequential addresses
  - Increment by 4, not by 1!
- Keeping a pointer to an automatic variable after procedure returns
  - e.g., passing pointer back via an argument
  - Pointer becomes invalid when stack popped

# Concluding Remarks

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



# Concluding Remarks

- Measure MIPS instruction executions in benchmark programs
  - Consider making the common case fast
  - Consider compromises

Instruction class	MIPS examples	SPEC2006 Int	SPEC2006 FP
Arithmetic	<i>add, sub, addi</i>	16%	48%
Data transfer	<i>lw, sw, lb, lbu, lh, lhu, sb</i>	35%	36%
Logical	<i>and, or, nor, andi, ori, sll, srl, sra</i>	12%	4%
Cond. Branch	<i>beq, bne, slt, slti, sltiu</i>	34%	8%
Jump	<i>j, jr, jal</i>	2%	0%