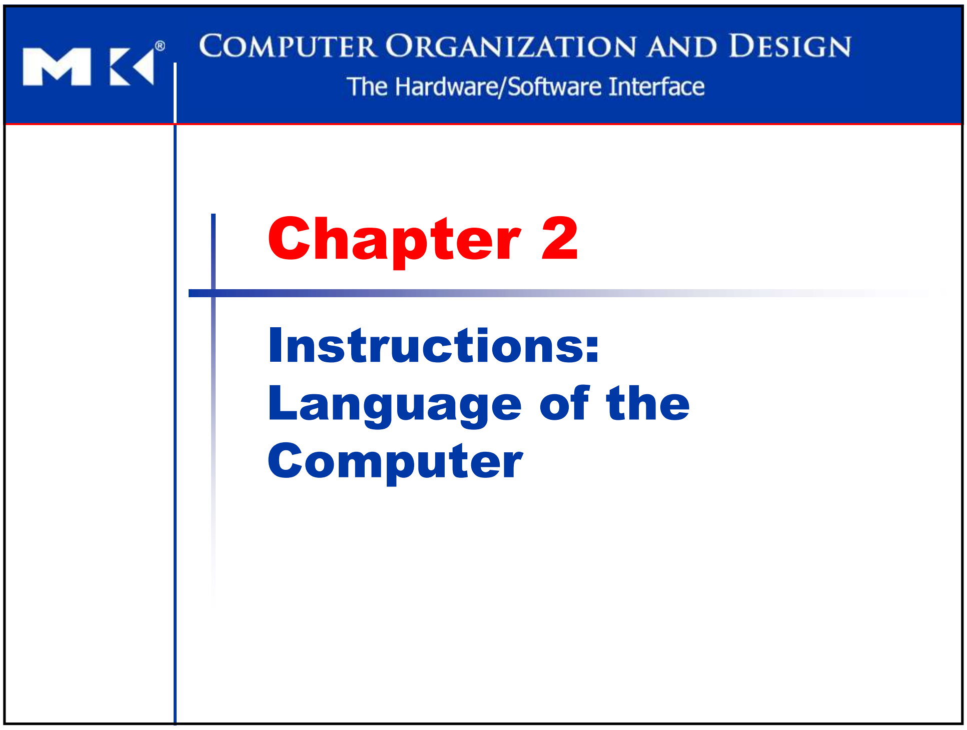


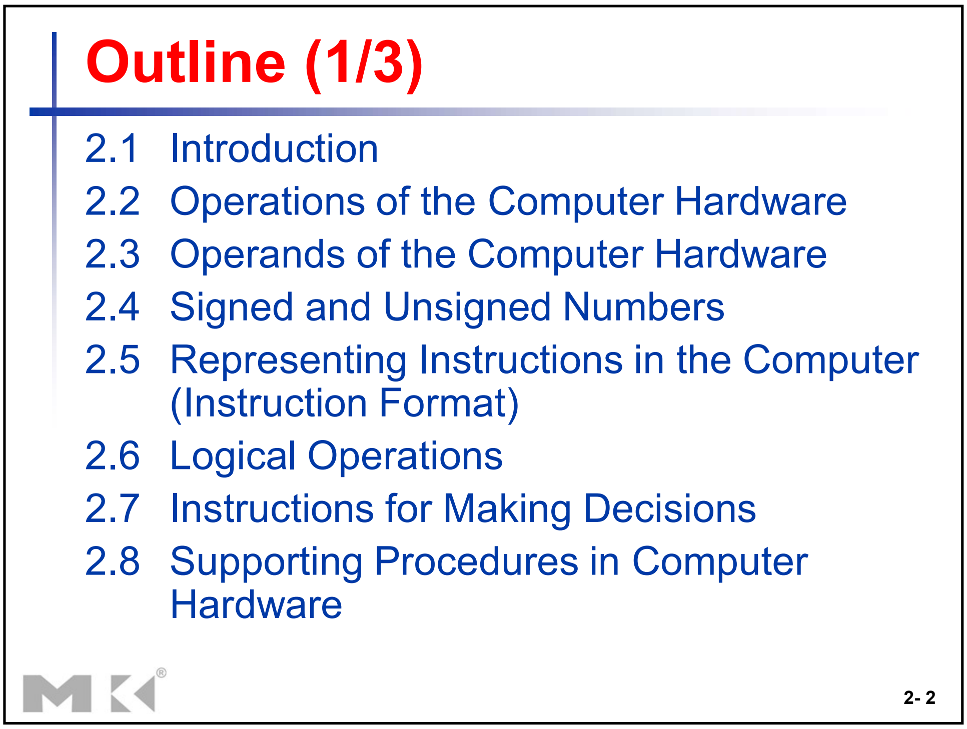
COMPUTER ORGANIZATION AND DESIGN

The Hardware/Software Interface

Chapter 2

Instructions: Language of the Computer

1

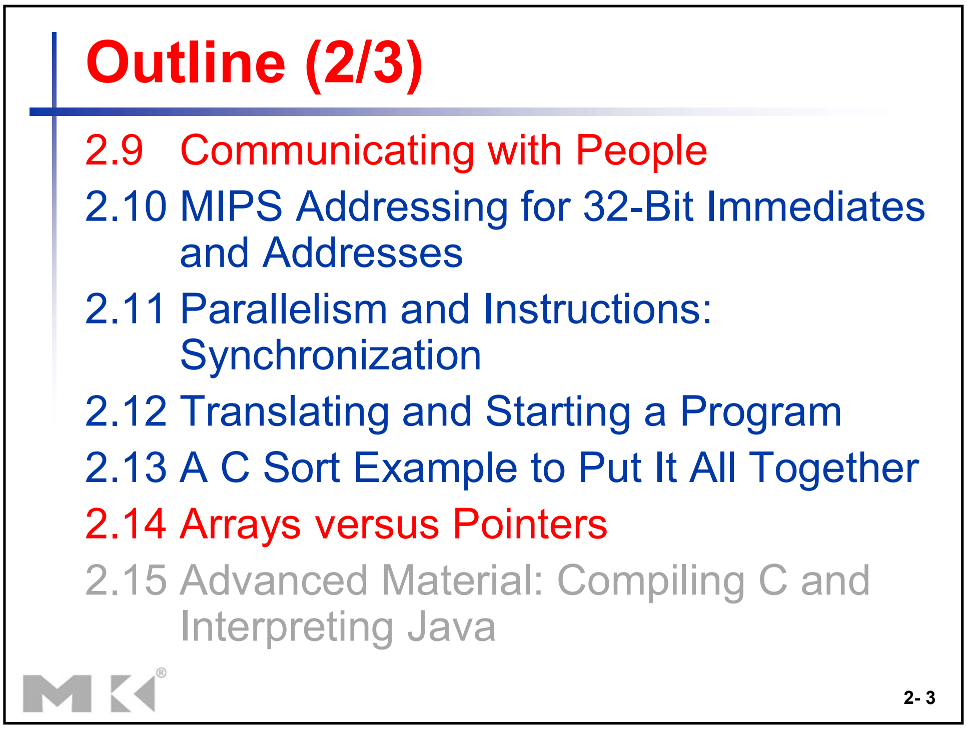


Outline (1/3)

- 2.1 Introduction
- 2.2 Operations of the Computer Hardware
- 2.3 Operands of the Computer Hardware
- 2.4 Signed and Unsigned Numbers
- 2.5 Representing Instructions in the Computer (Instruction Format)
- 2.6 Logical Operations
- 2.7 Instructions for Making Decisions
- 2.8 Supporting Procedures in Computer Hardware

2- 2

2



Outline (2/3)

2.9 Communicating with People

2.10 MIPS Addressing for 32-Bit Immediates and Addresses

2.11 Parallelism and Instructions: Synchronization

2.12 Translating and Starting a Program

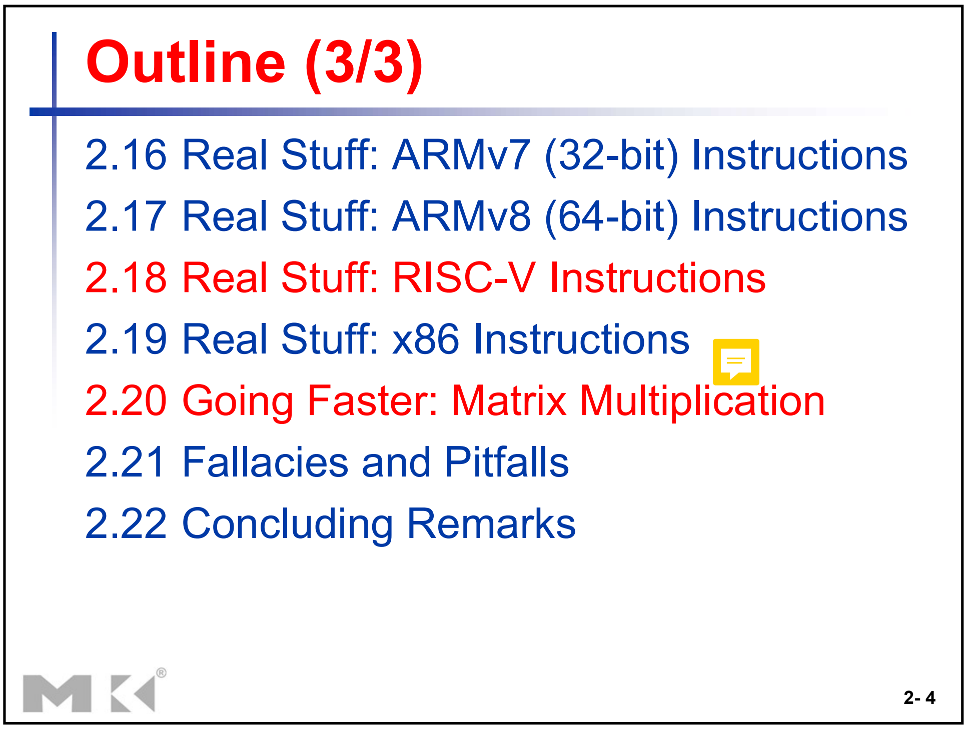
2.13 A C Sort Example to Put It All Together

2.14 Arrays versus Pointers

2.15 Advanced Material: Compiling C and Interpreting Java

2- 3

3



Outline (3/3)

2.16 Real Stuff: ARMv7 (32-bit) Instructions

2.17 Real Stuff: ARMv8 (64-bit) Instructions

2.18 Real Stuff: RISC-V Instructions

2.19 Real Stuff: x86 Instructions

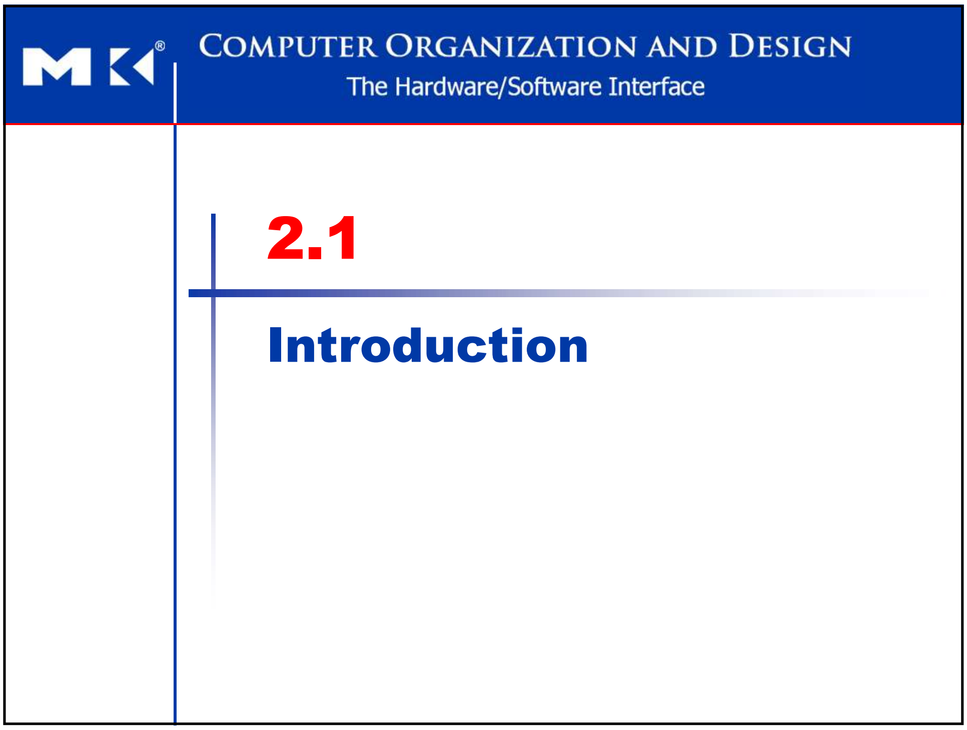
2.20 Going Faster: Matrix Multiplication

2.21 Fallacies and Pitfalls

2.22 Concluding Remarks

2- 4

4



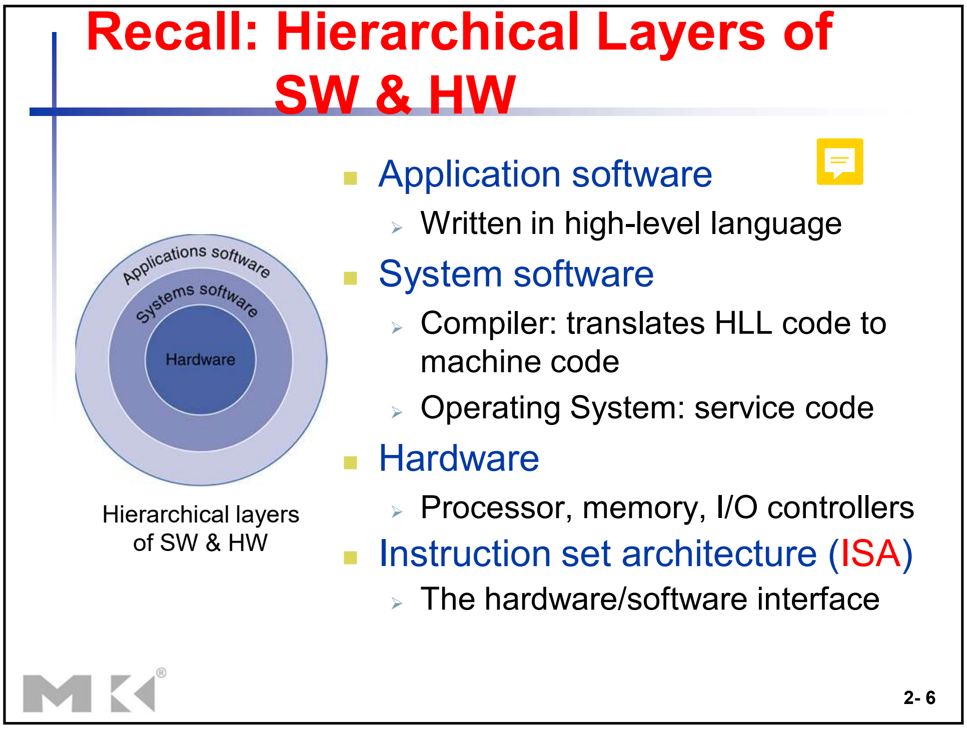
COMPUTER ORGANIZATION AND DESIGN

The Hardware/Software Interface

2.1

Introduction

5



Recall: Hierarchical Layers of SW & HW

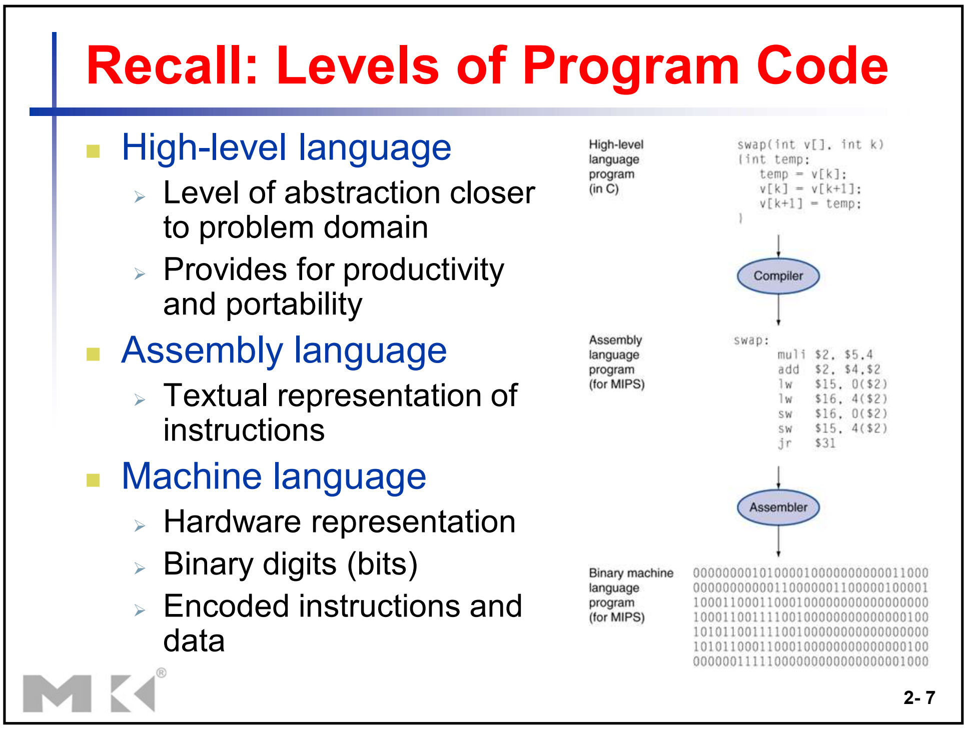
Applications software
Systems software
Hardware

Hierarchical layers of SW & HW

- Application software
  - Written in high-level language
- System software
  - Compiler: translates HLL code to machine code
  - Operating System: service code
- Hardware
  - Processor, memory, I/O controllers
- Instruction set architecture (ISA)
  - The hardware/software interface

2- 6

6



Recall: Levels of Program Code

- High-level language
  - Level of abstraction closer to problem domain
  - Provides for productivity and portability
- Assembly language
  - Textual representation of instructions
- Machine language
  - Hardware representation
  - Binary digits (bits)
  - Encoded instructions and data

High-level language program (in C)

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

Compiler

Assembly language program (for MIPS)

```
swap:
      muli $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
```

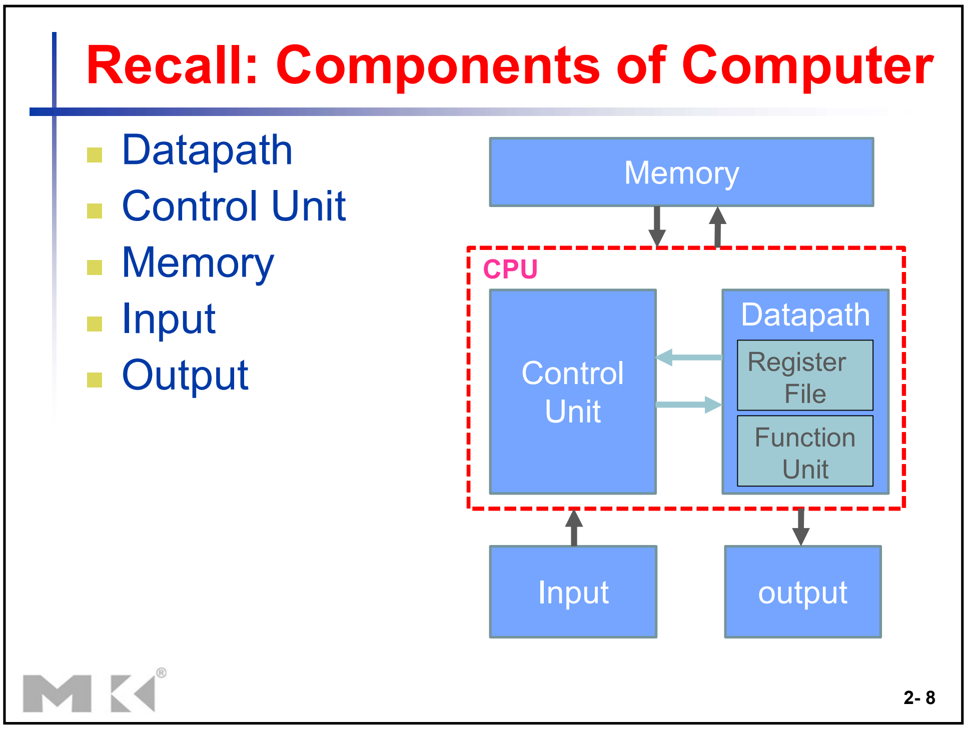
Assembler

Binary machine language program (for MIPS)

```
00000000101000010000000000011000
00000000000110000001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

2- 7

7

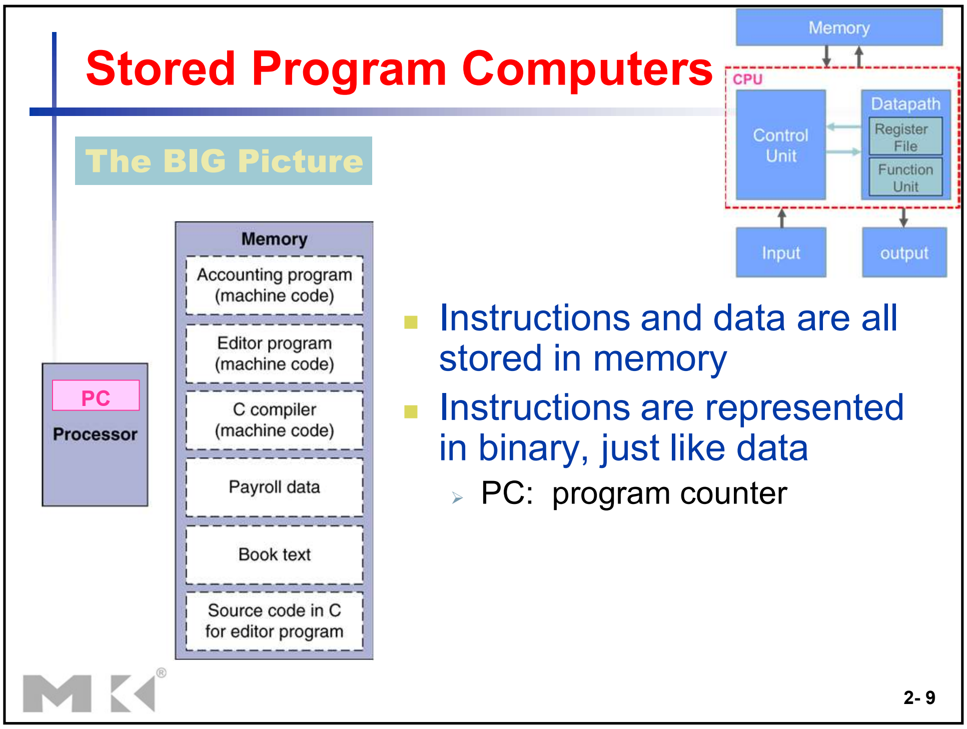


Recall: Components of Computer

- Datapath
- Control Unit
- Memory
- Input
- Output

2- 8

8



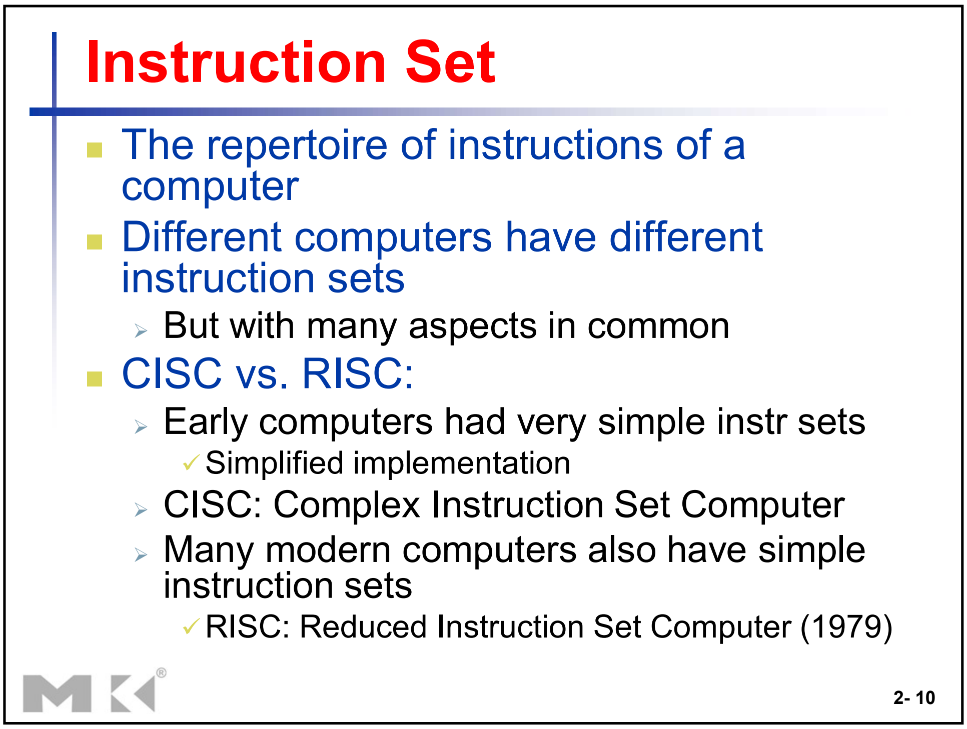
Stored Program Computers

The BIG Picture

Memory
CPU
Control Unit
Datapath
Register File
Function Unit
Input
output
PC
Processor
Memory
Accounting program (machine code)
Editor program (machine code)
C compiler (machine code)
Payroll data
Book text
Source code in C for editor program

- Instructions and data are all stored in memory
- Instructions are represented in binary, just like data
  - PC: program counter

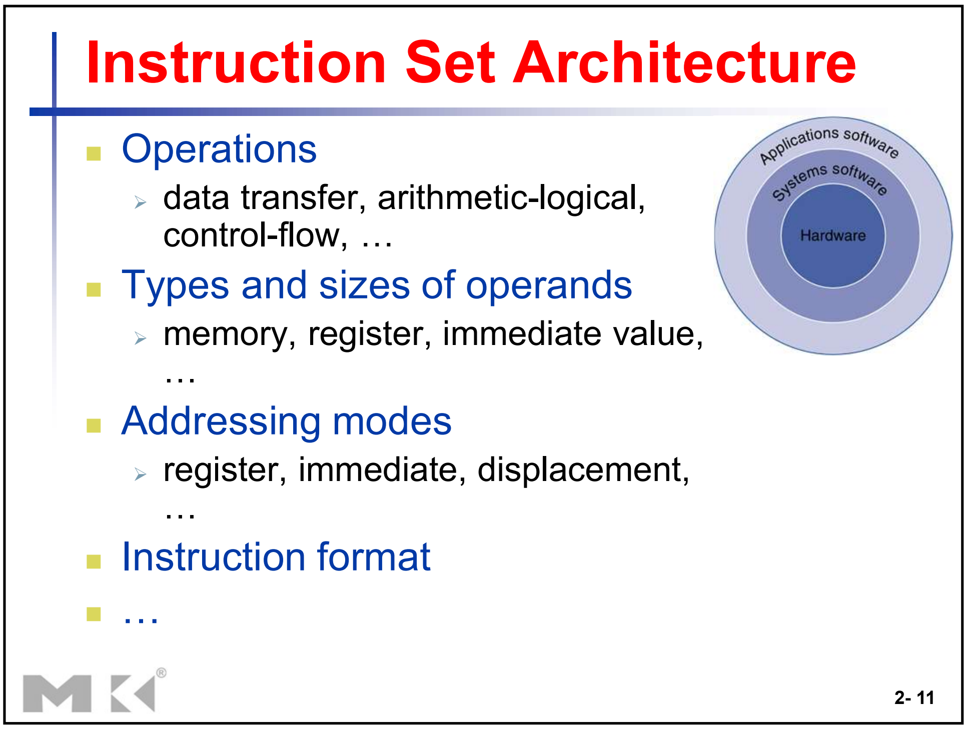
2- 9



Instruction Set

- The repertoire of instructions of a computer
- Different computers have different instruction sets
  - But with many aspects in common
- CISC vs. RISC:
  - Early computers had very simple instr sets
    - ✓Simplified implementation
  - CISC: Complex Instruction Set Computer
  - Many modern computers also have simple instruction sets
    - ✓RISC: Reduced Instruction Set Computer (1979)

2- 10



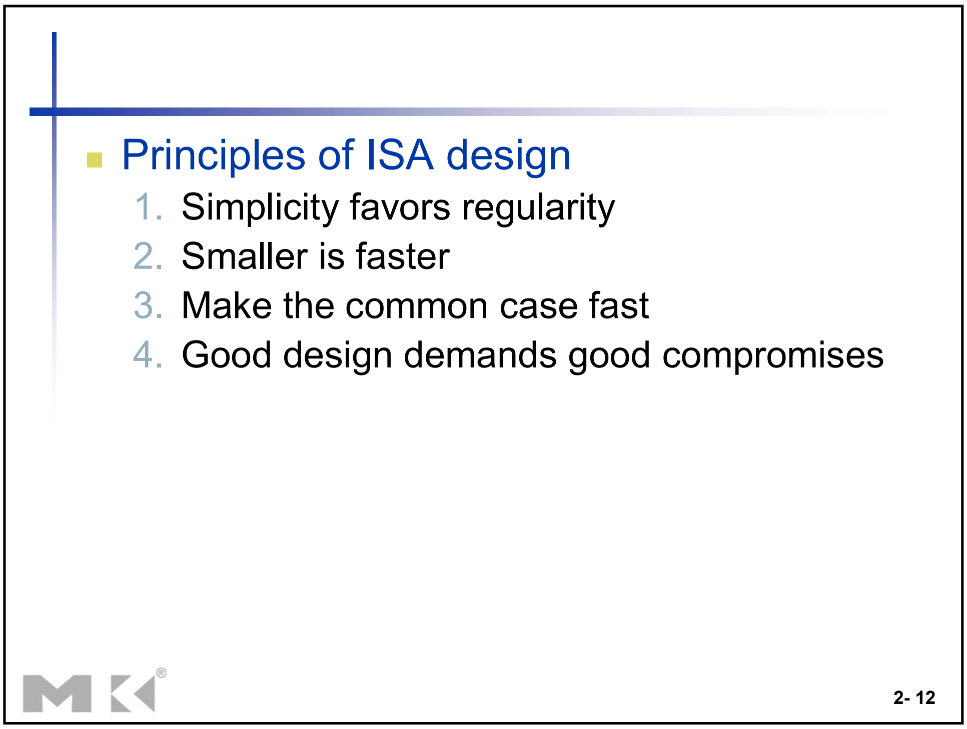
Instruction Set Architecture

- Operations
  - data transfer, arithmetic-logical, control-flow, …
- Types and sizes of operands
  - memory, register, immediate value, …
- Addressing modes
  - register, immediate, displacement, …
- Instruction format
- …

Applications software
Systems software
Hardware

2- 11

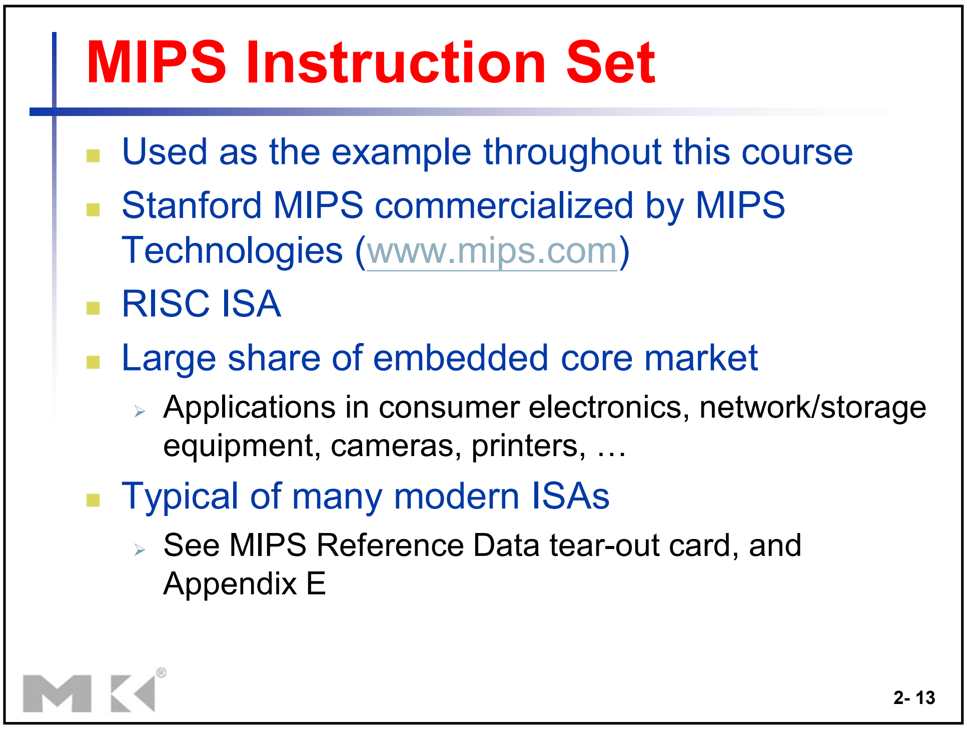
11



- Principles of ISA design
  1. Simplicity favors regularity
  2. Smaller is faster
  3. Make the common case fast
  4. Good design demands good compromises

2- 12

12

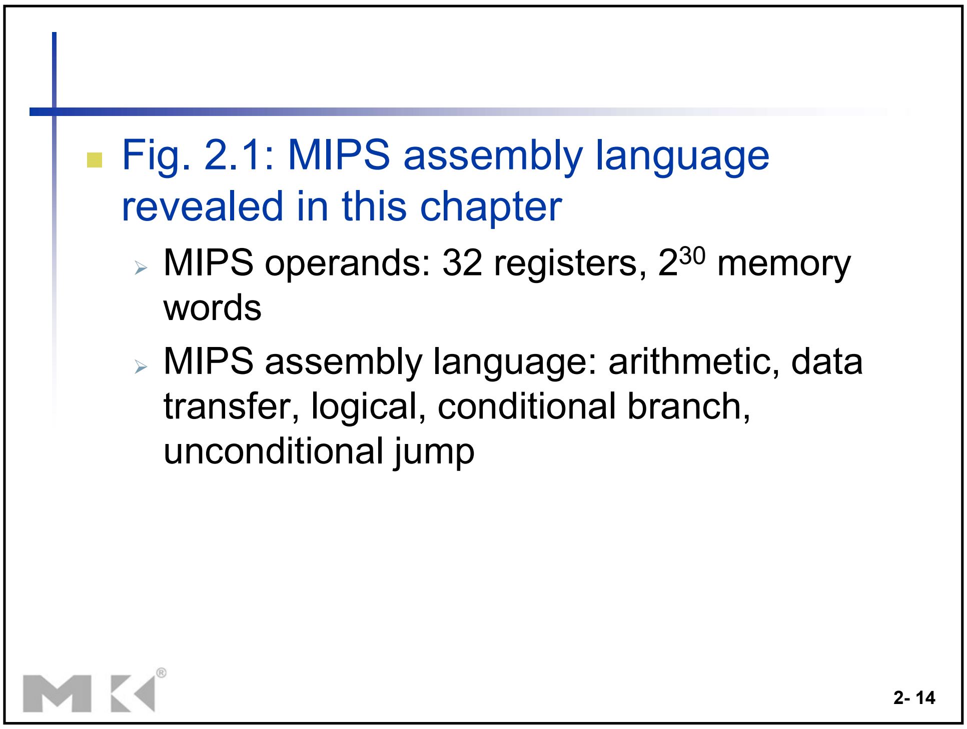


MIPS Instruction Set

- Used as the example throughout this course
- Stanford MIPS commercialized by MIPS Technologies (www.mips.com)
- RISC ISA
- 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 Appendix E

2- 13

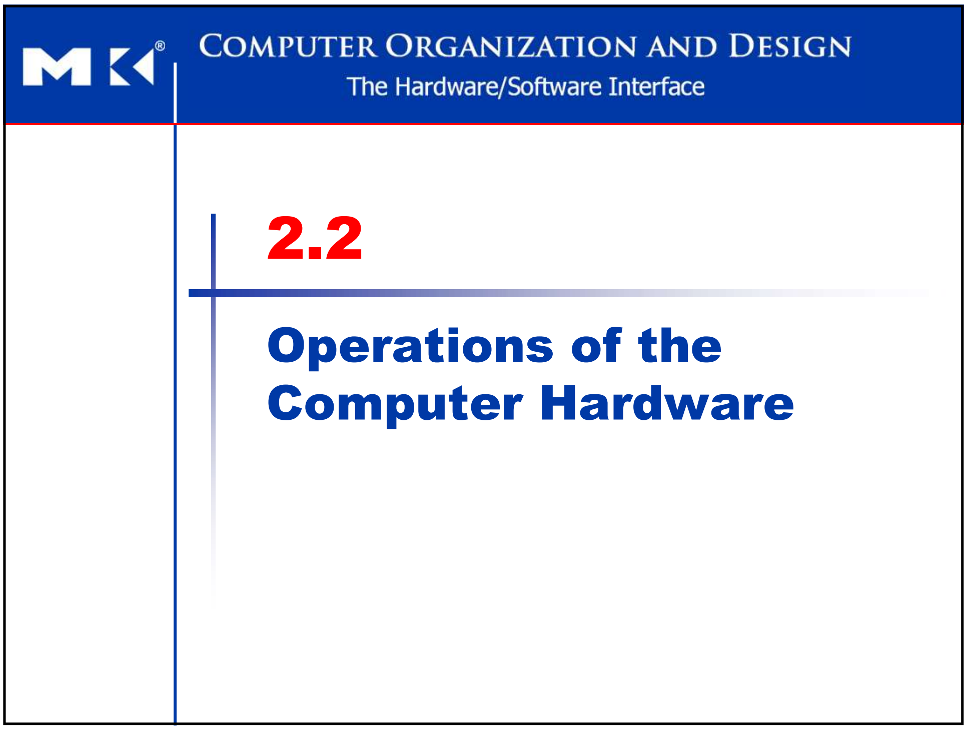
13



- Fig. 2.1: MIPS assembly language revealed in this chapter
  - MIPS operands: 32 registers, 2^{30} memory words
  - MIPS assembly language: arithmetic, data transfer, logical, conditional branch, unconditional jump

2- 14

14



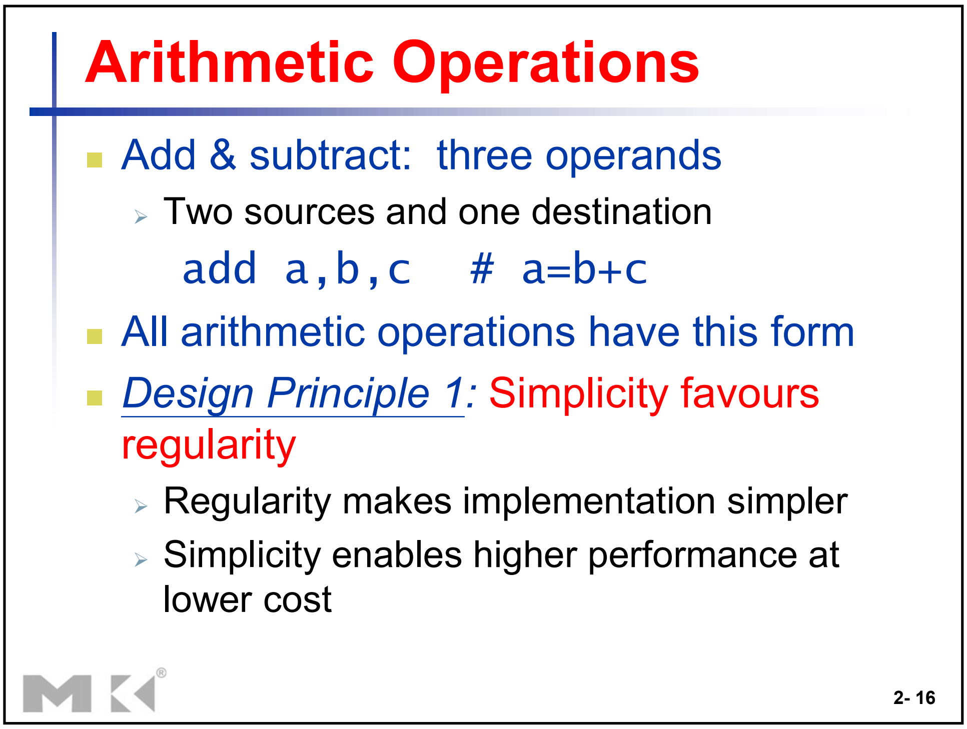
COMPUTER ORGANIZATION AND DESIGN

The Hardware/Software Interface

2.2

Operations of the Computer Hardware

15



Arithmetic Operations

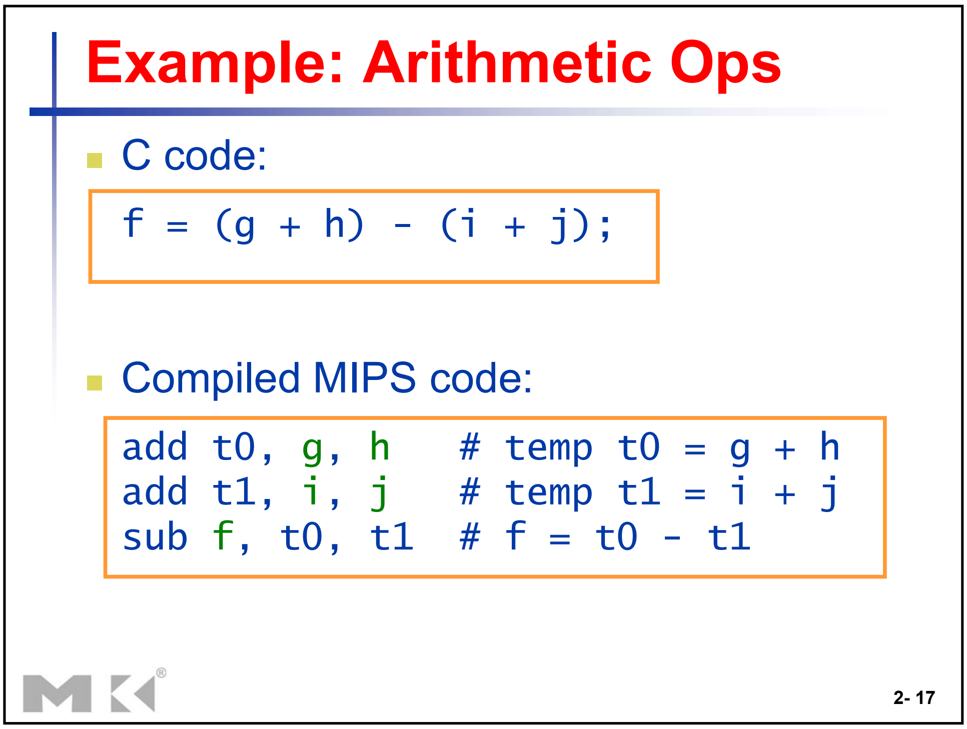
- Add & subtract: three operands
  - Two sources and one destination

```
add a,b,c   # a=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

2- 16

16



Example: Arithmetic Ops

- C code:

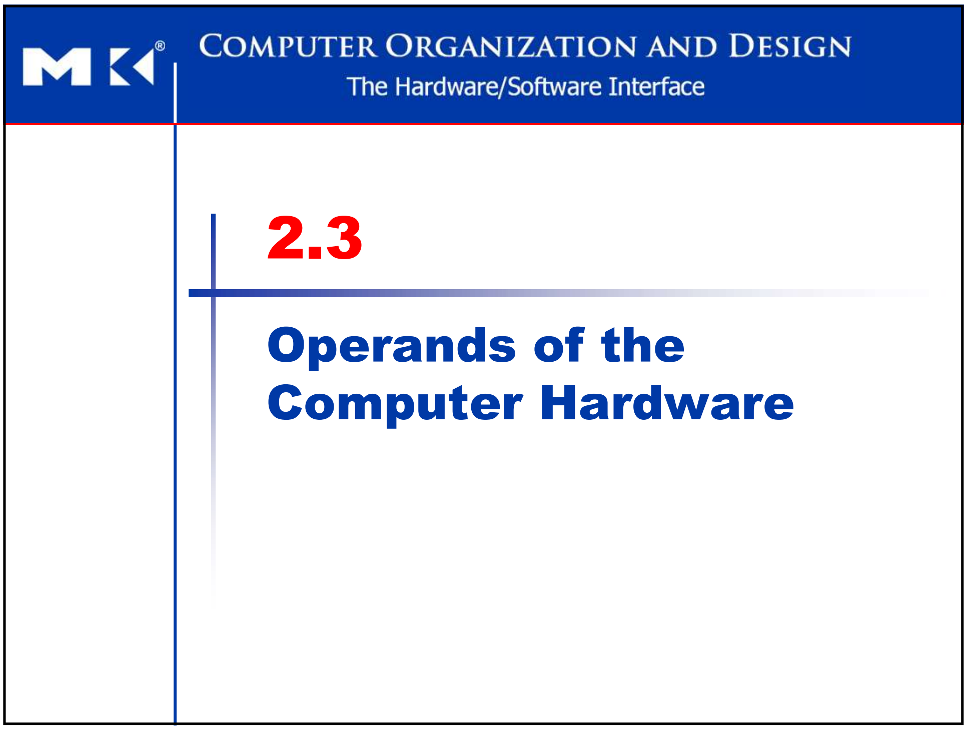
```
f = (g + h) - (i + j);
```

- 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
```

2- 17

17



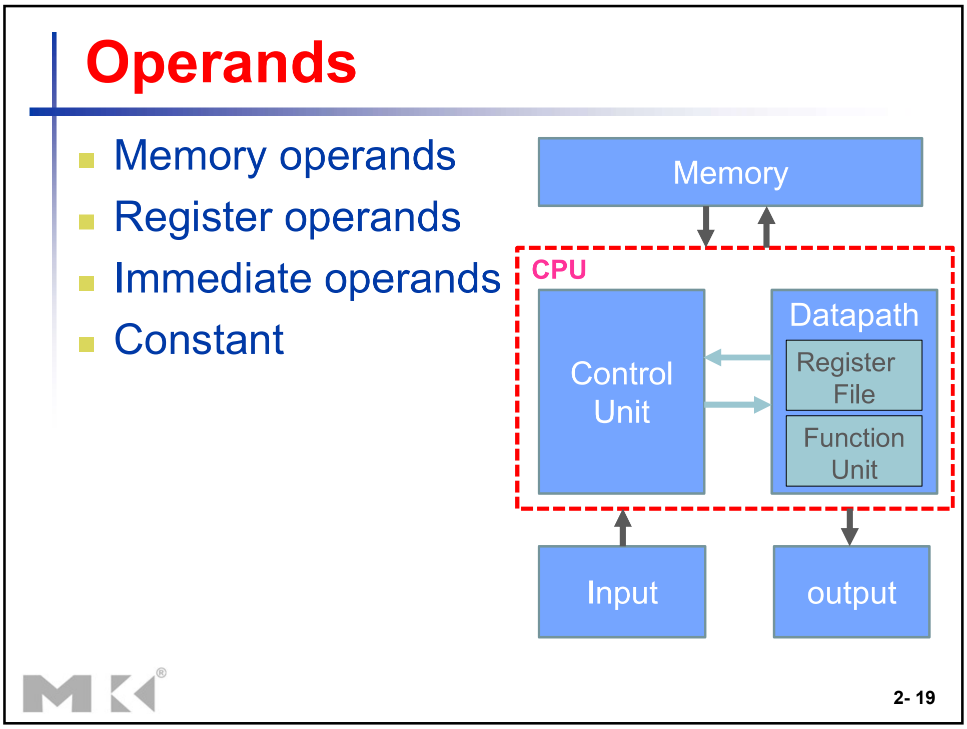
COMPUTER ORGANIZATION AND DESIGN

The Hardware/Software Interface

2.3

Operands of the Computer Hardware

18

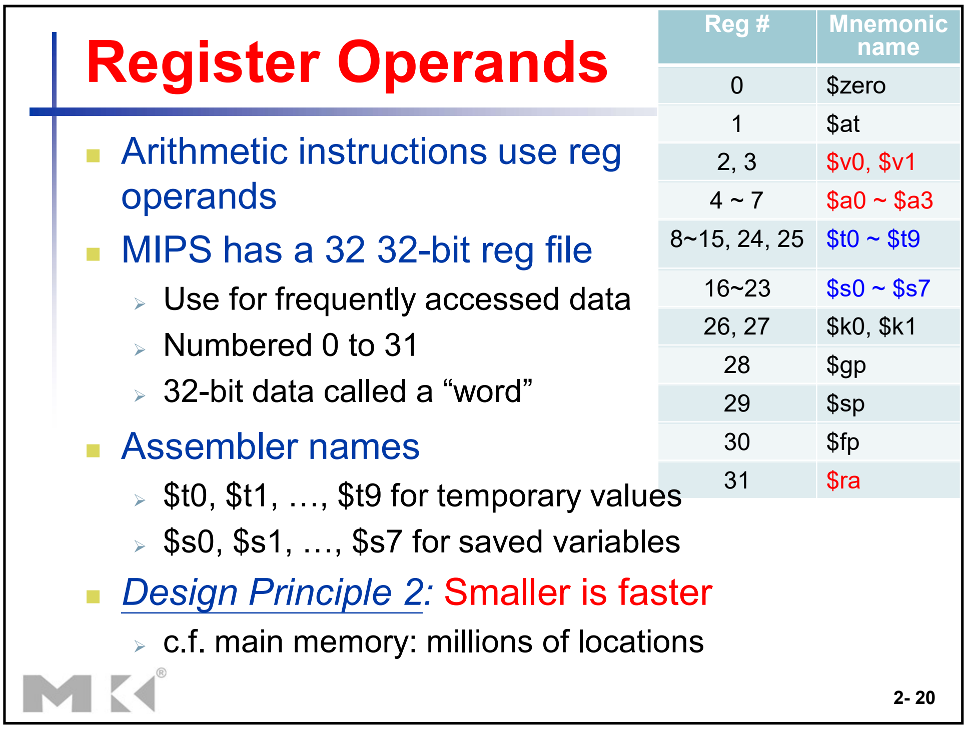


Operands

- Memory operands
- Register operands
- Immediate operands
- Constant

2- 19

19



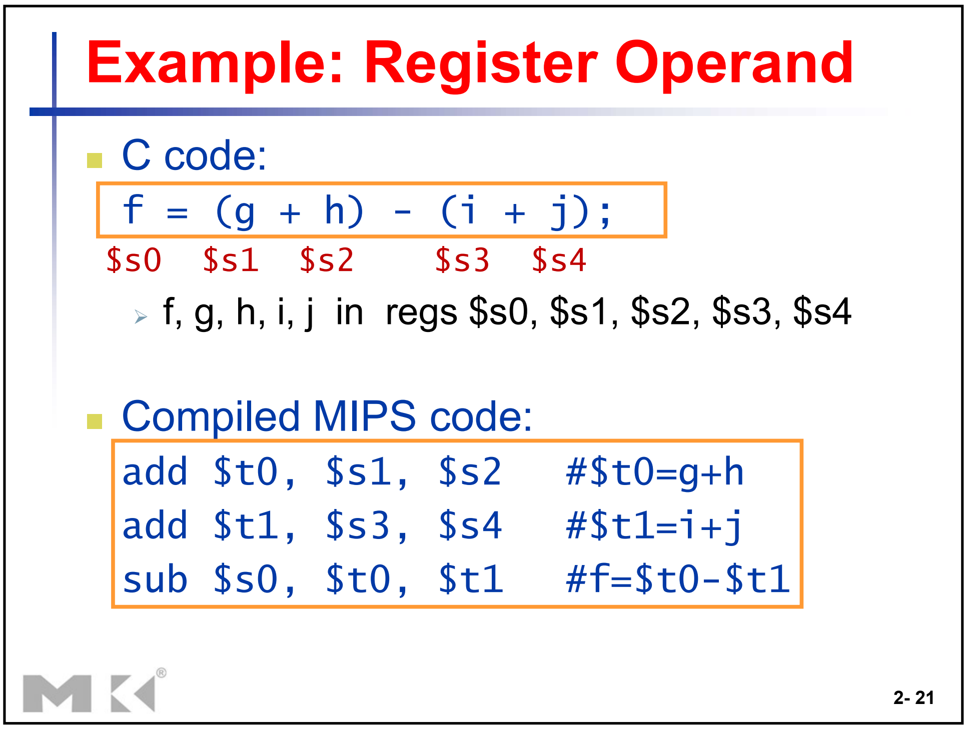
Register Operands

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

Reg #	Mnemonic name
0	$zero
1	$at
2, 3	$v0, $v1
4 ~ 7	$a0 ~ $a3
8~15, 24, 25	$t0 ~ $t9
16~23	$s0 ~ $s7
26, 27	$k0, $k1
28	$gp
29	$sp
30	$fp
31	$ra

2- 20

20



Example: Register Operand

- C code:

```
f = (g + h) - (i + j);
$s0  $s1  $s2    $s3  $s4
```

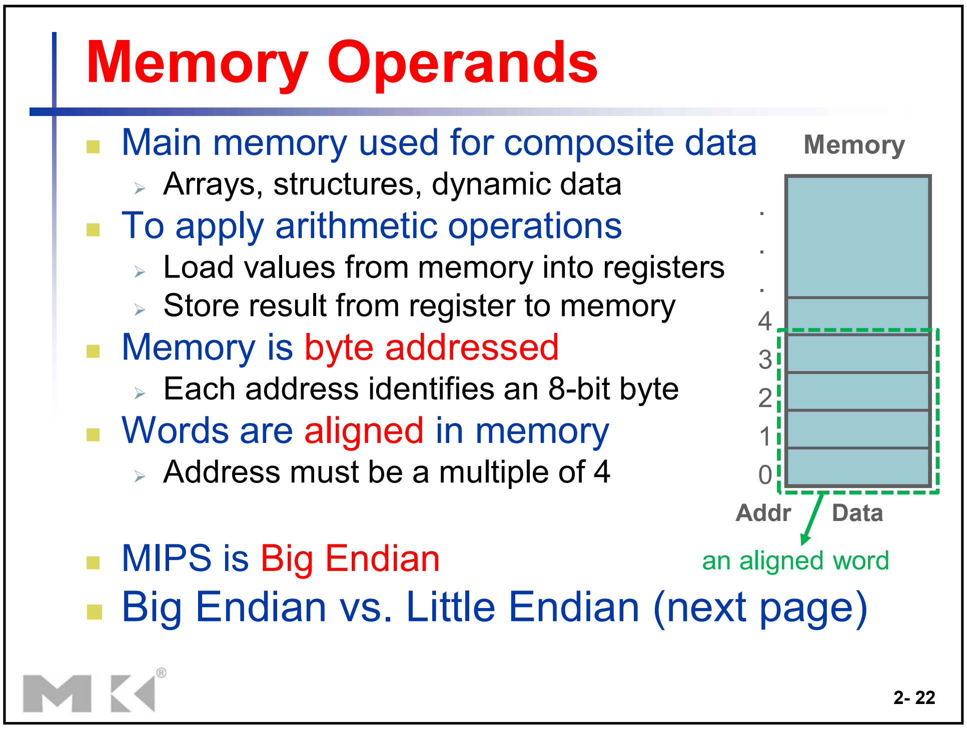
  - f, g, h, i, j in regs $s0, $s1, $s2, $s3, $s4

- Compiled MIPS code:

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

2- 21

21

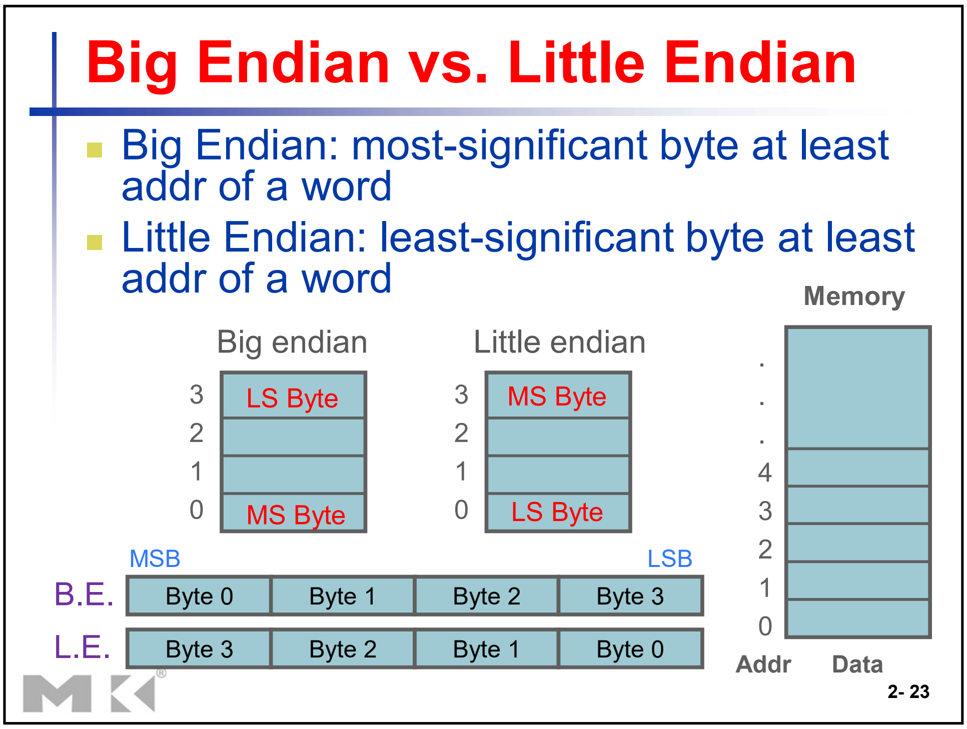


Memory Operands

- 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
- Big Endian vs. Little Endian (next page)

2- 22

22



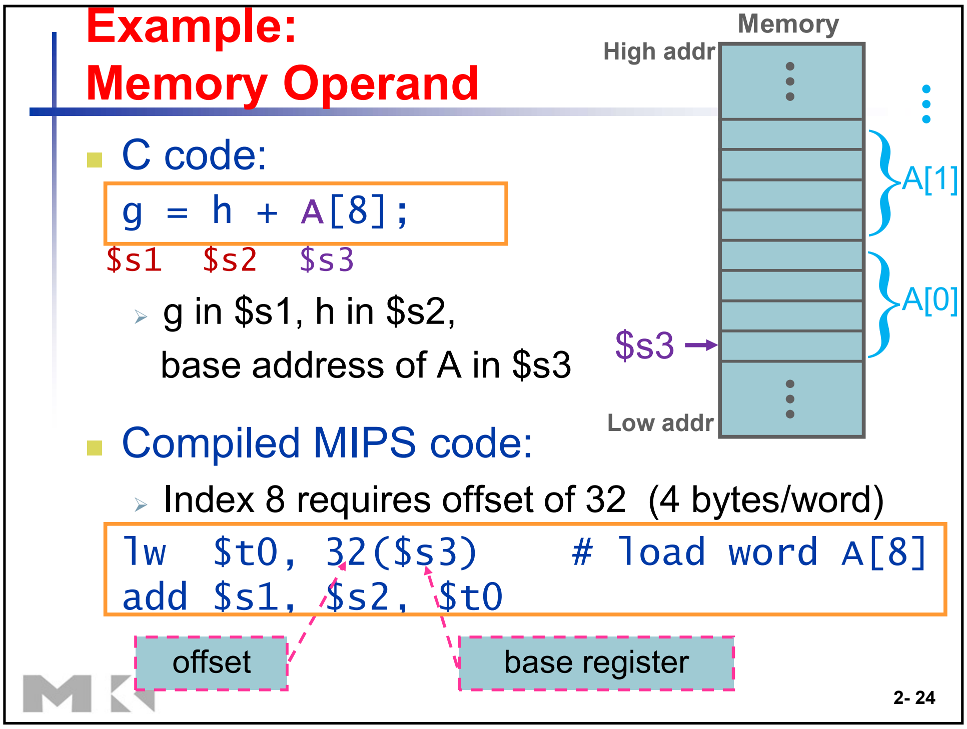
Big Endian vs. Little Endian

- Big Endian: most-significant byte at least addr of a word
- Little Endian: least-significant byte at least addr of a word

	MSB			LSB
B.E.	Byte 0	Byte 1	Byte 2	Byte 3
L.E.	Byte 3	Byte 2	Byte 1	Byte 0

2- 23

23



Example: Memory Operand

- C code:

```
g = h + A[8];
$s1  $s2  $s3
```

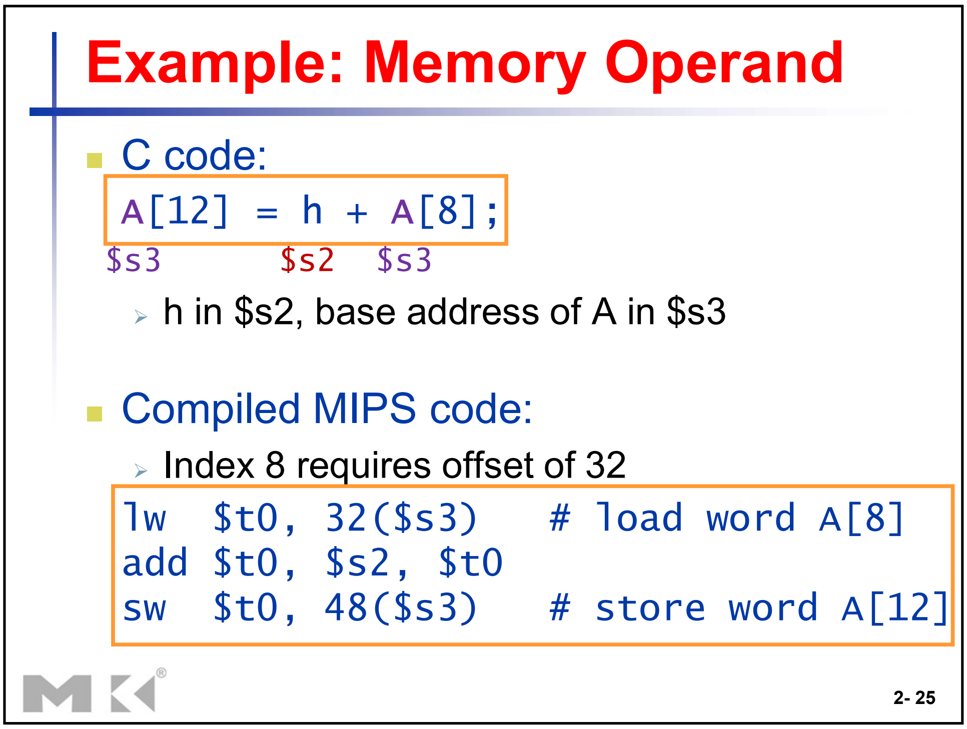
  - g in $s1, h in $s2,
    base address of A in $s3

- Compiled MIPS code:
  - Index 8 requires offset of 32 (4 bytes/word)

```
lw  $t0, 32($s3)    # load word A[8]
add $s1, $s2, $t0
```

2- 24

24



Example: Memory Operand

- C code:

```
A[12] = h + A[8];
$s3      $s2  $s3
```

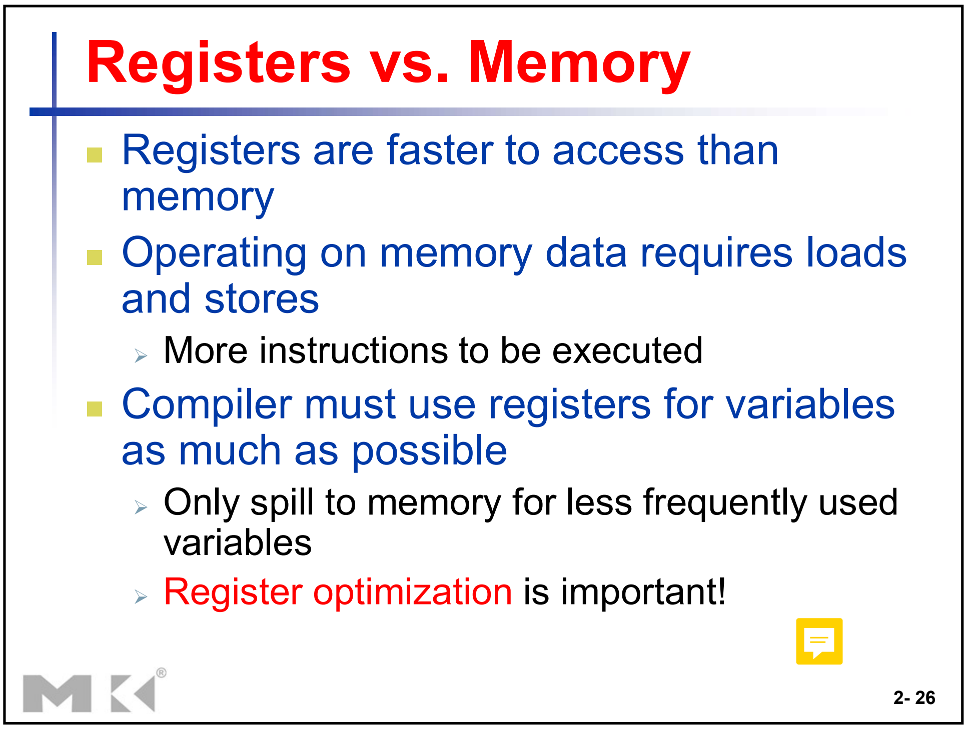
  - h in $s2, base address of A in $s3

- Compiled MIPS code:
  - Index 8 requires offset of 32

```
lw  $t0, 32($s3)   # load word A[8]
add $t0, $s2, $t0
sw  $t0, 48($s3)   # store word A[12]
```

2- 25

25

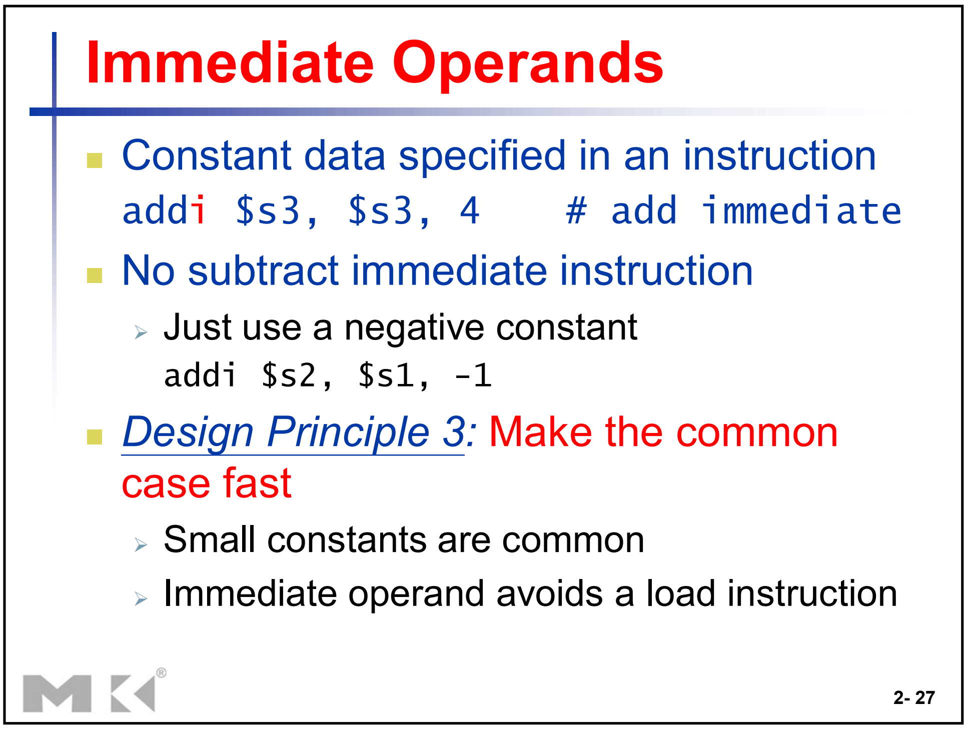


Registers vs. Memory

- 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!

2- 26

26



Immediate Operands

- Constant data specified in an instruction

```
addi $s3, $s3, 4    # add immediate
```

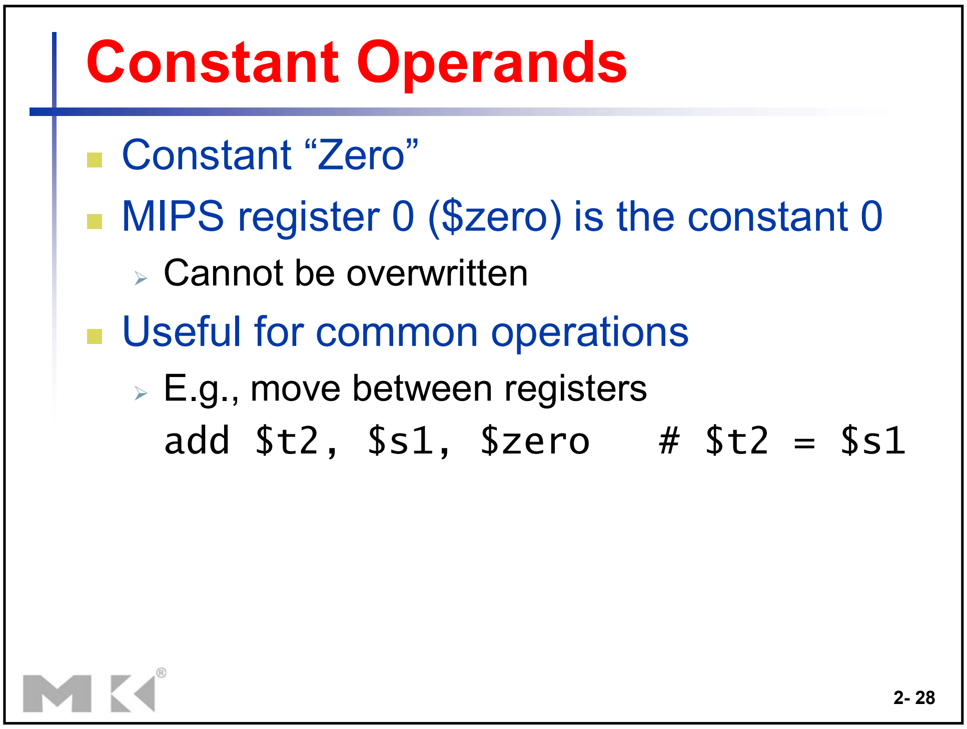
- No subtract immediate instruction
  - Just use a negative constant

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

- *Design Principle 3:* Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction

2- 27

27



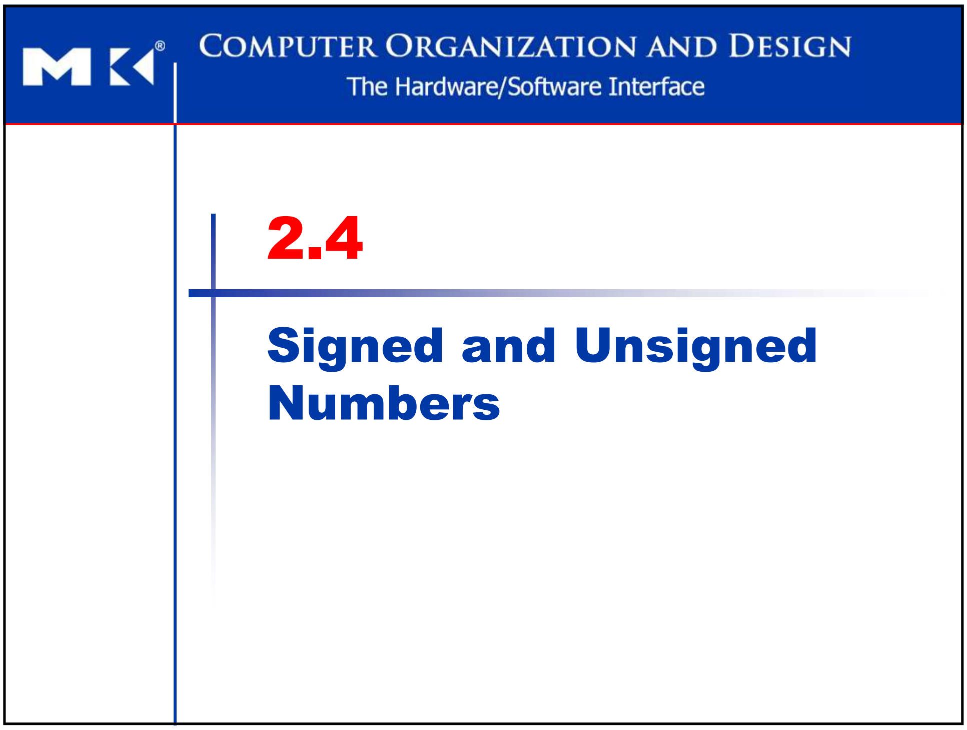
Constant Operands

- 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   # $t2 = $s1
```

2- 28

28

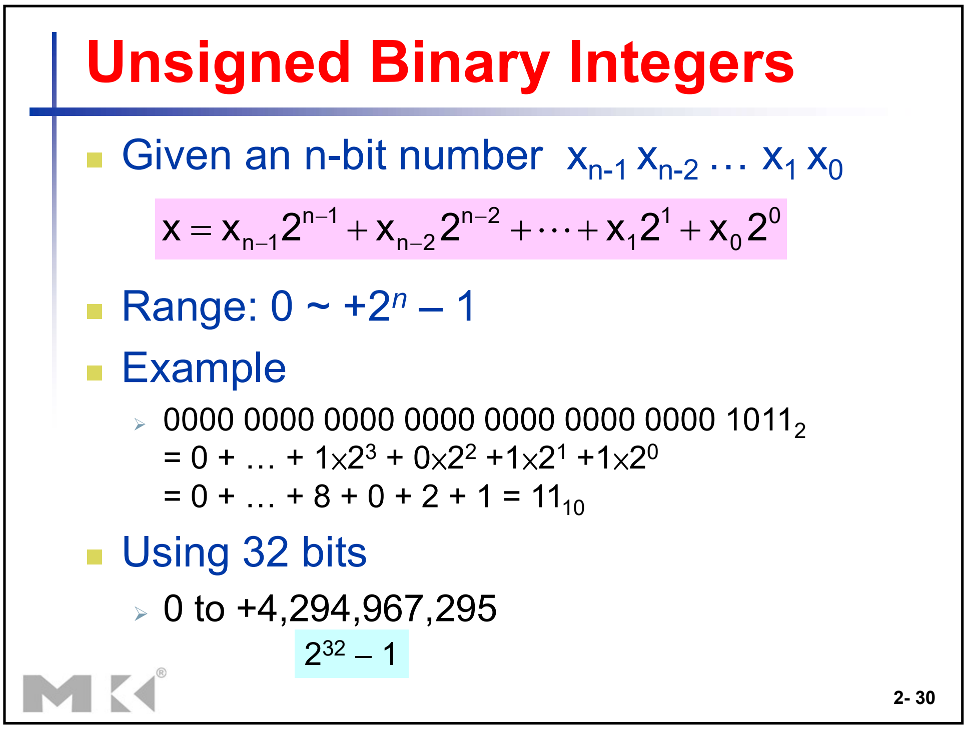


COMPUTER ORGANIZATION AND DESIGN
The Hardware/Software Interface

2.4

Signed and Unsigned Numbers

29



Unsigned Binary Integers

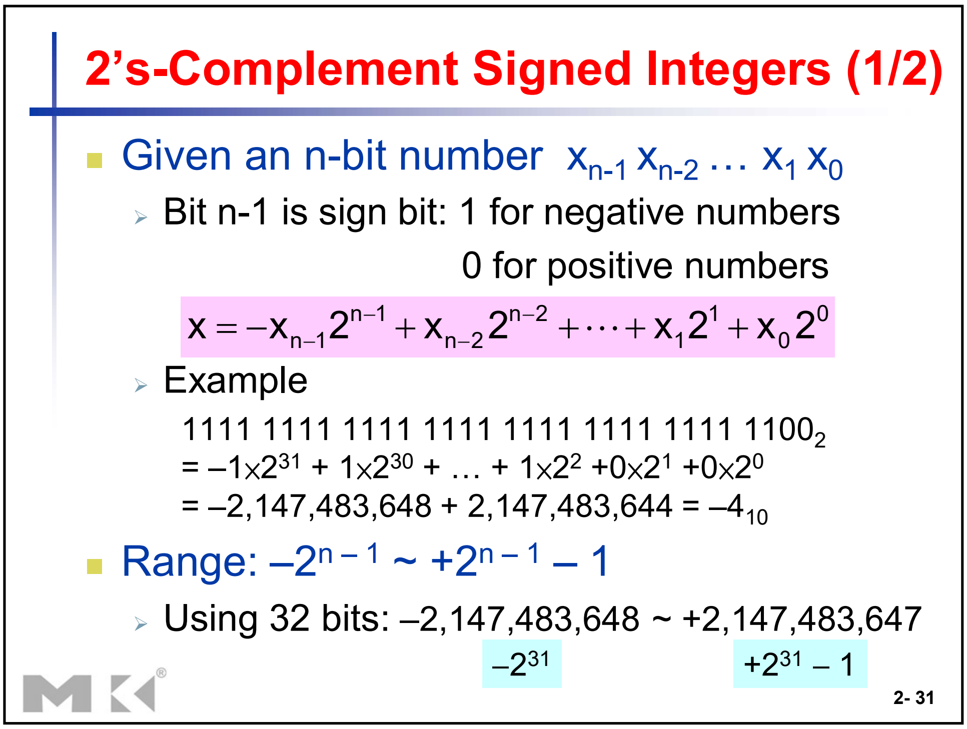
- Given an n-bit number $x_{n-1}\, x_{n-2} \ldots x_1\, x_0$

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

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

$2^{32} - 1$

30



2's-Complement Signed Integers (1/2)

- Given an n-bit number $x_{n-1}\, x_{n-2} \ldots x_1\, x_0$
  - Bit n-1 is sign bit: 1 for negative numbers
    0 for positive numbers

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

  - Example

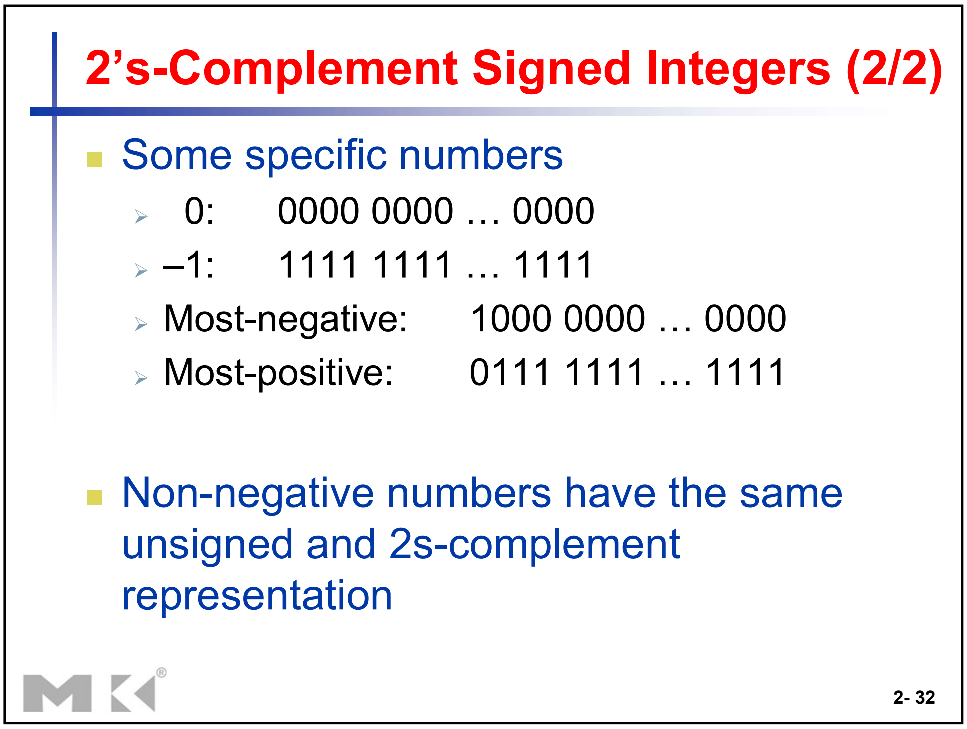
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
$= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^2 + 0\times2^1 + 0\times2^0$
$= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$

- Range: $-2^{n-1} \sim +2^{n-1} - 1$
  - Using 32 bits: –2,147,483,648 ~ +2,147,483,647

-2^{31} $+2^{31} - 1$

2- 31

31

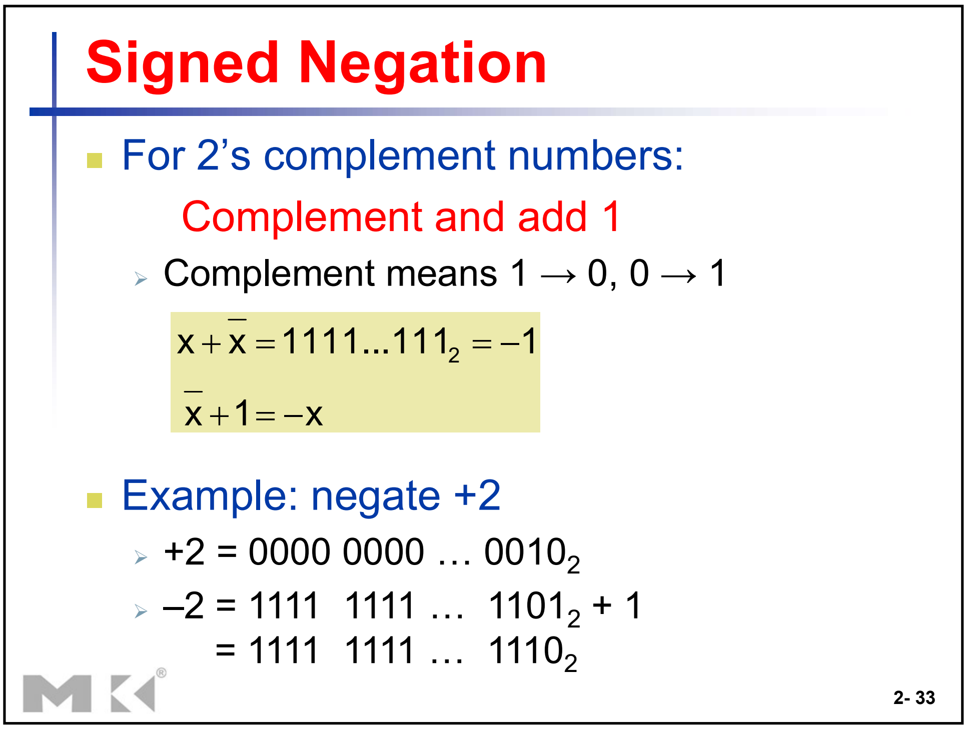


2's-Complement Signed Integers (2/2)

- Some specific numbers
  - 0: 0000 0000 … 0000
  - –1: 1111 1111 … 1111
  - Most-negative: 1000 0000 … 0000
  - Most-positive: 0111 1111 … 1111
- Non-negative numbers have the same unsigned and 2s-complement representation

2- 32

32



Signed Negation

- For 2's complement numbers:

  Complement and add 1

  - Complement means 1 → 0, 0 → 1

$$x + \bar{x} = 1111...111_2 = -1$$

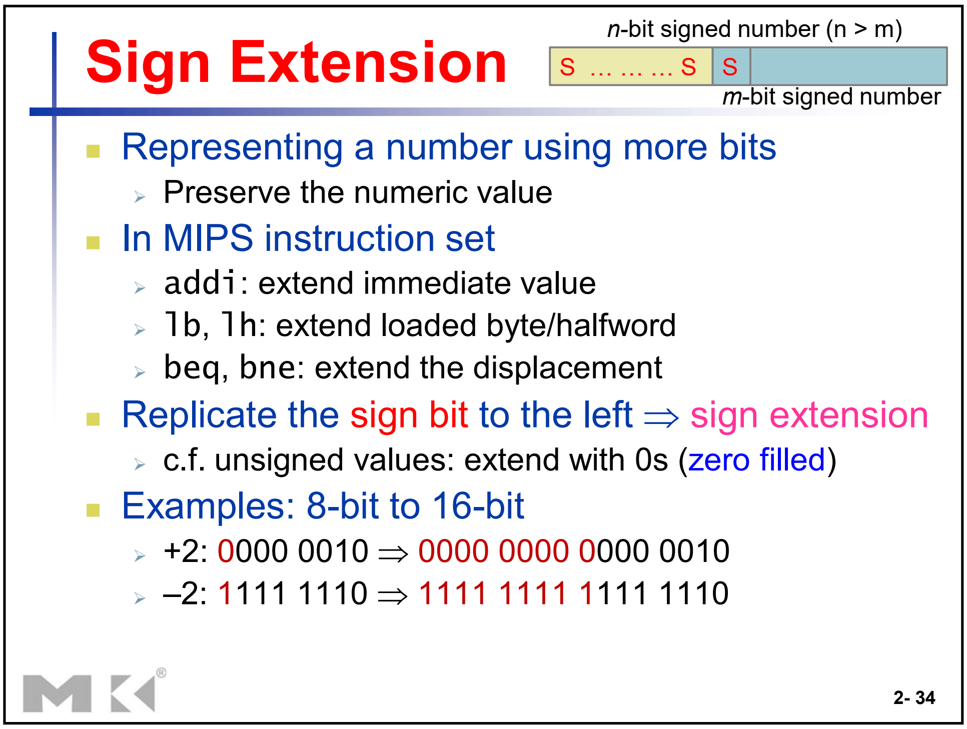
$$\bar{x} + 1 = -x$$

- Example: negate +2
  - +2 = 0000 0000 … 0010_2
  - –2 = 1111 1111 … 1101_2 + 1

    = 1111 1111 … 1110_2

2- 33

33



Sign Extension

n-bit signed number (n > m)

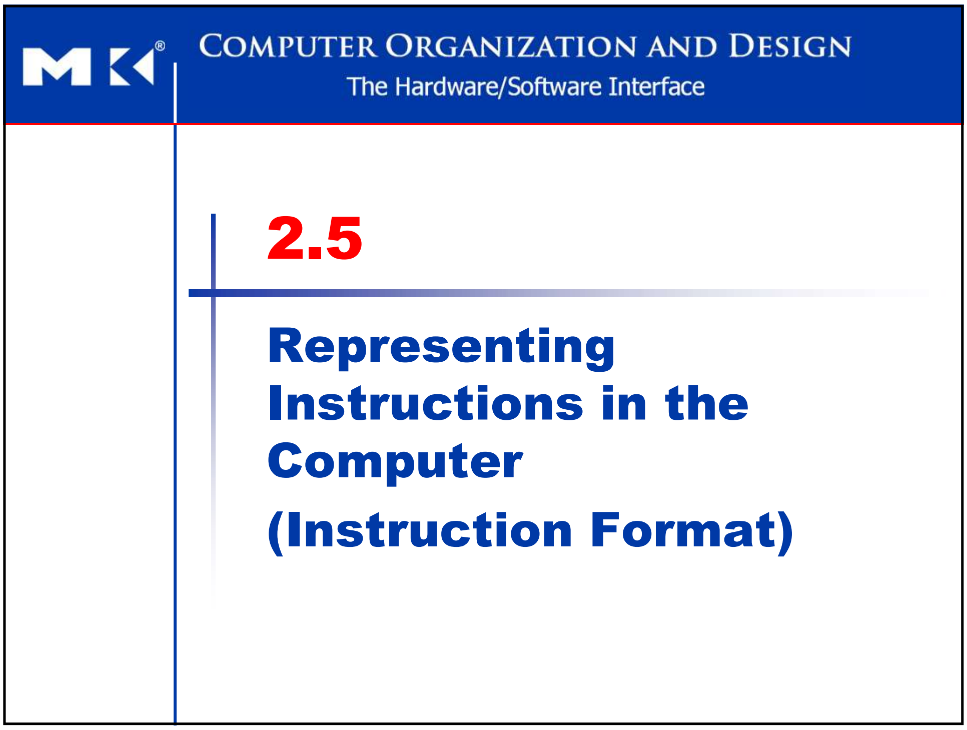
S … … … S | S |

m-bit signed number

- 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 ⇒ sign extension
  - c.f. unsigned values: extend with 0s (zero filled)
- Examples: 8-bit to 16-bit
  - +2: 0000 0010 ⇒ 0000 0000 0000 0010
  - –2: 1111 1110 ⇒ 1111 1111 1111 1110

2- 34

34



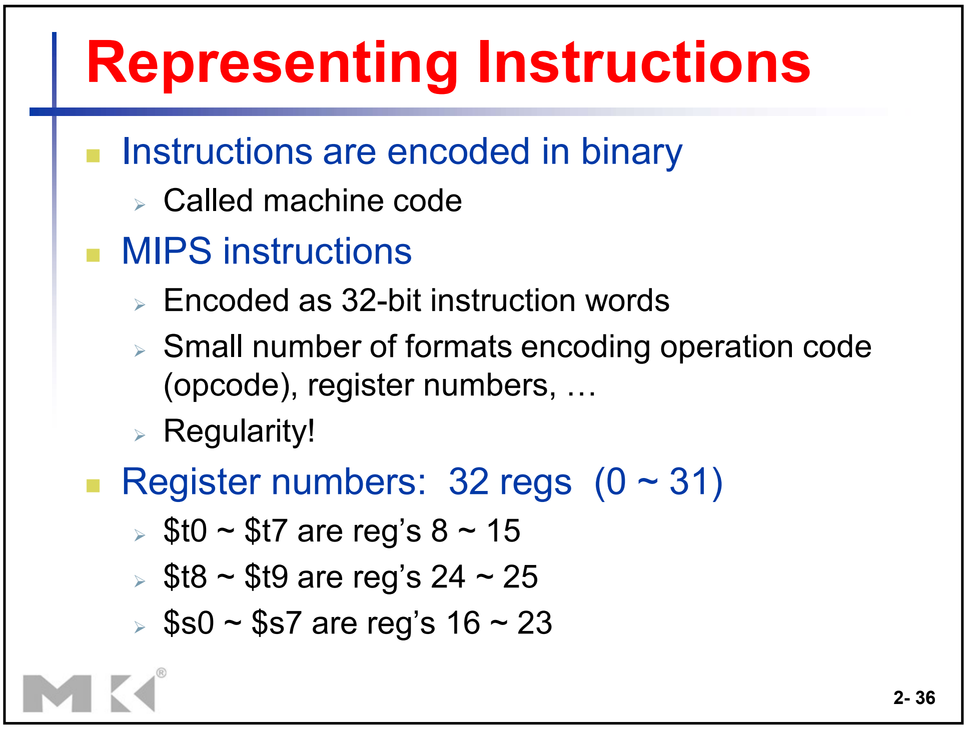
COMPUTER ORGANIZATION AND DESIGN

The Hardware/Software Interface

2.5

Representing Instructions in the Computer (Instruction Format)

35

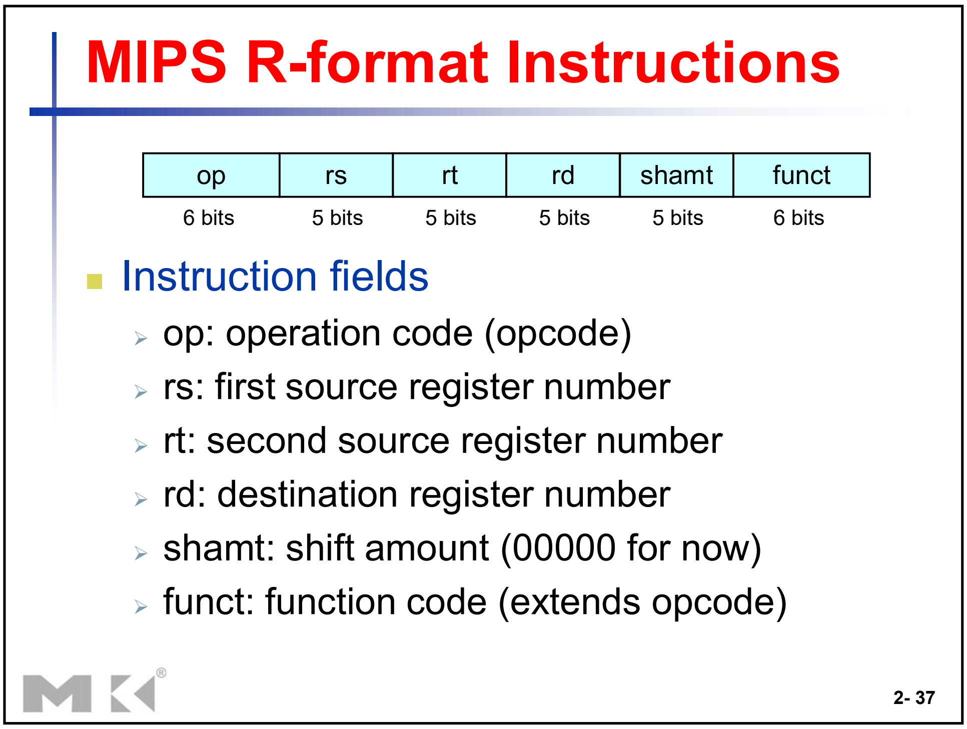


Representing Instructions

- 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: 32 regs (0 ~ 31)
  - \$t0 ~ \$t7 are reg's 8 ~ 15
  - \$t8 ~ \$t9 are reg's 24 ~ 25
  - \$s0 ~ \$s7 are reg's 16 ~ 23

2- 36

36



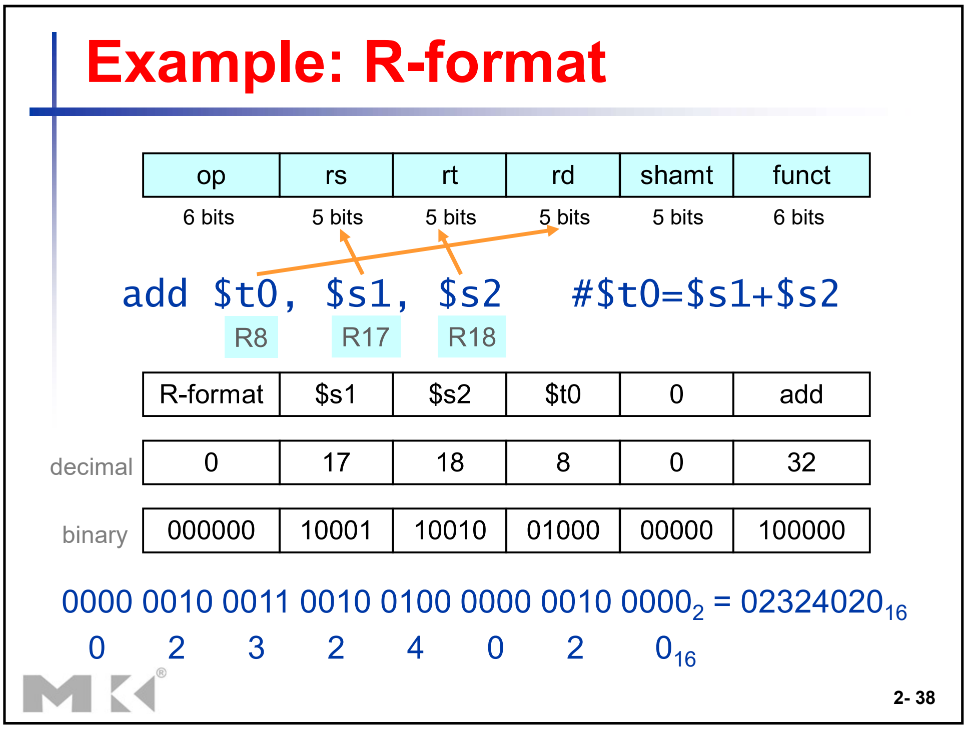
MIPS R-format Instructions

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

- 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)

2- 37

37



Example: R-format

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

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

R8 R17 R18

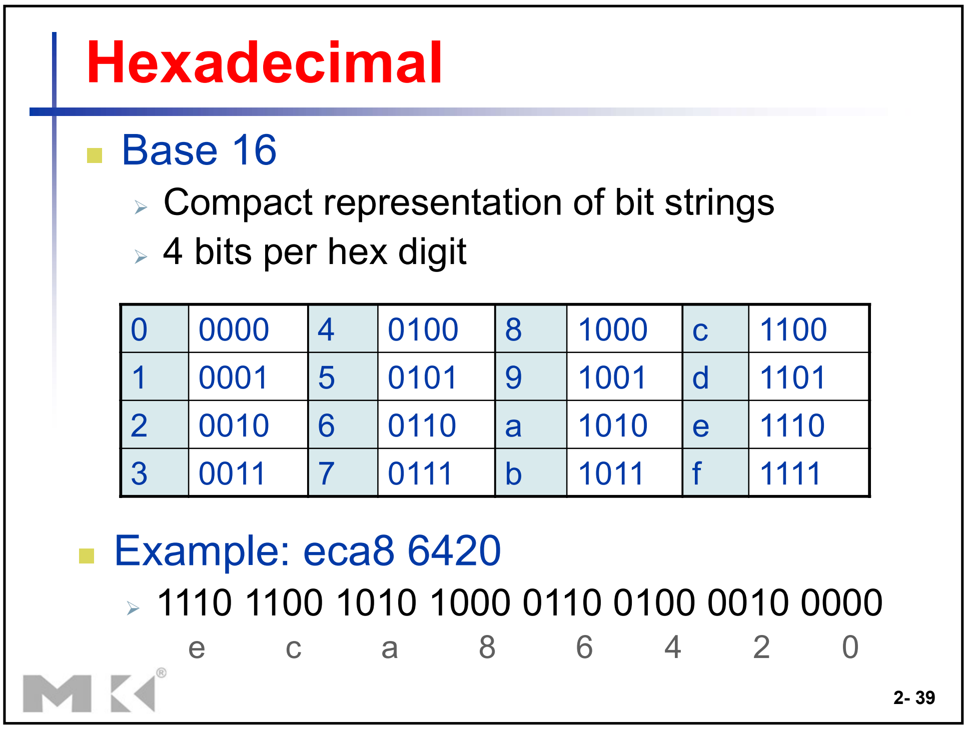
	R-format	$s1	$s2	$t0	0	add
decimal	0	17	18	8	0	32
binary	000000	10001	10010	01000	00000	100000

0000 0010 0011 0010 0100 0000 0010 0000_2 = 02324020_{16}

0 2 3 2 4 0 2 0_{16}

2- 38

38



Hexadecimal

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

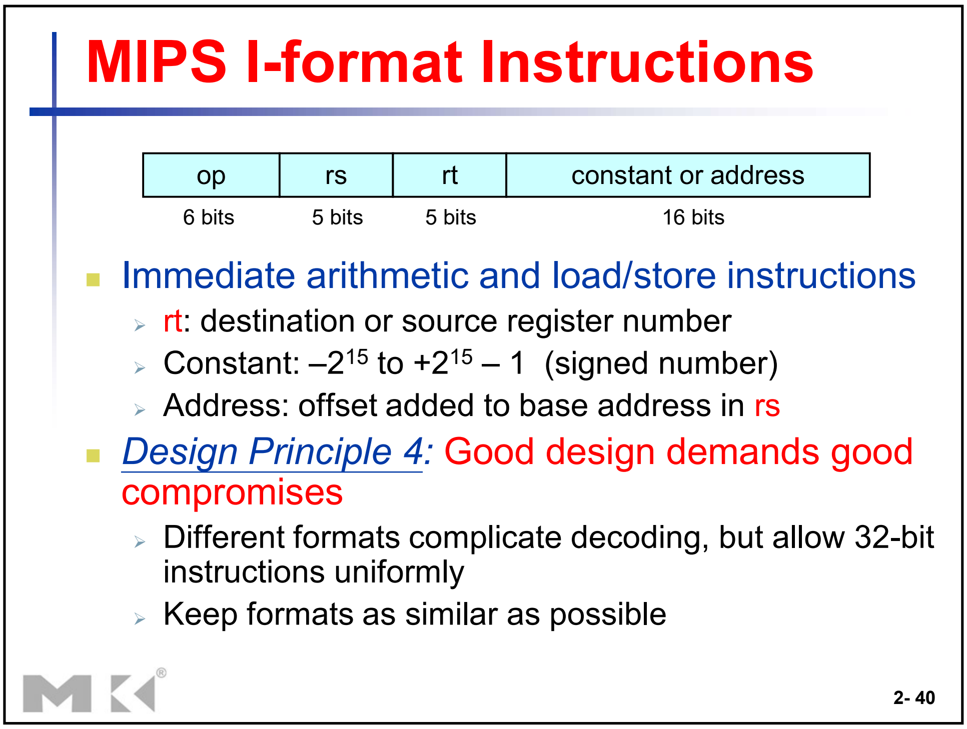
0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

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

```
1110 1100 1010 1000 0110 0100 0010 0000
  e    c    a    8    6    4    2    0
```

2- 39

39



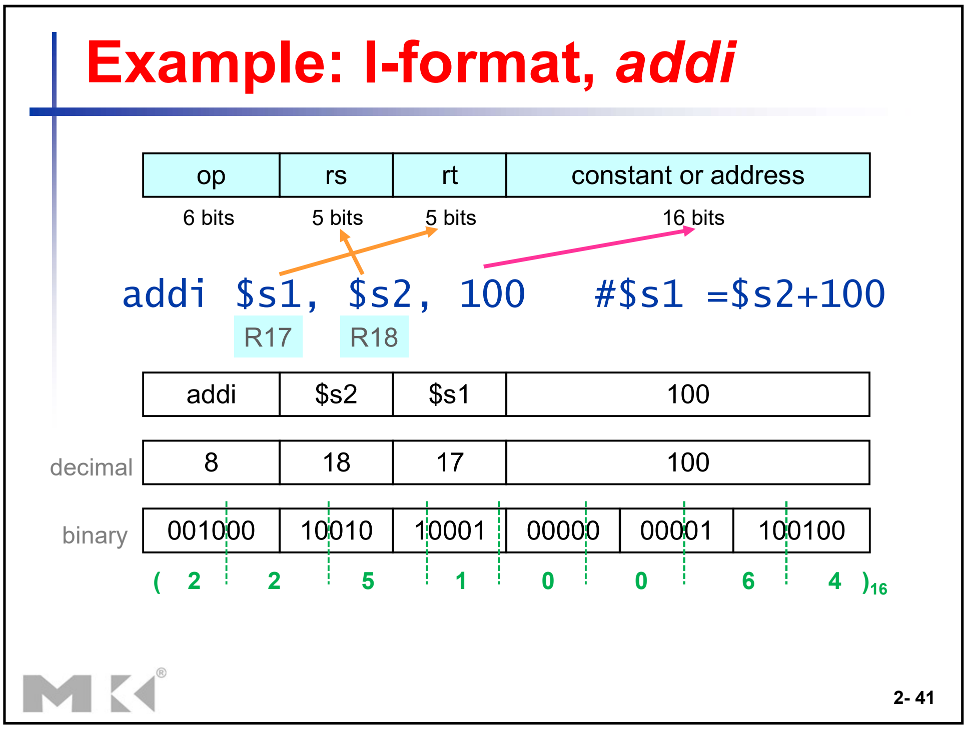
MIPS I-format Instructions

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- Immediate arithmetic and load/store instructions
  - rt: destination or source register number
  - Constant: -2^{15} to $+2^{15}-1$ (signed number)
  - 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

2- 40

40



Example: I-format, *addi*

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

```
addi $s1, $s2, 100   #$s1 =$s2+100
```

R17 R18

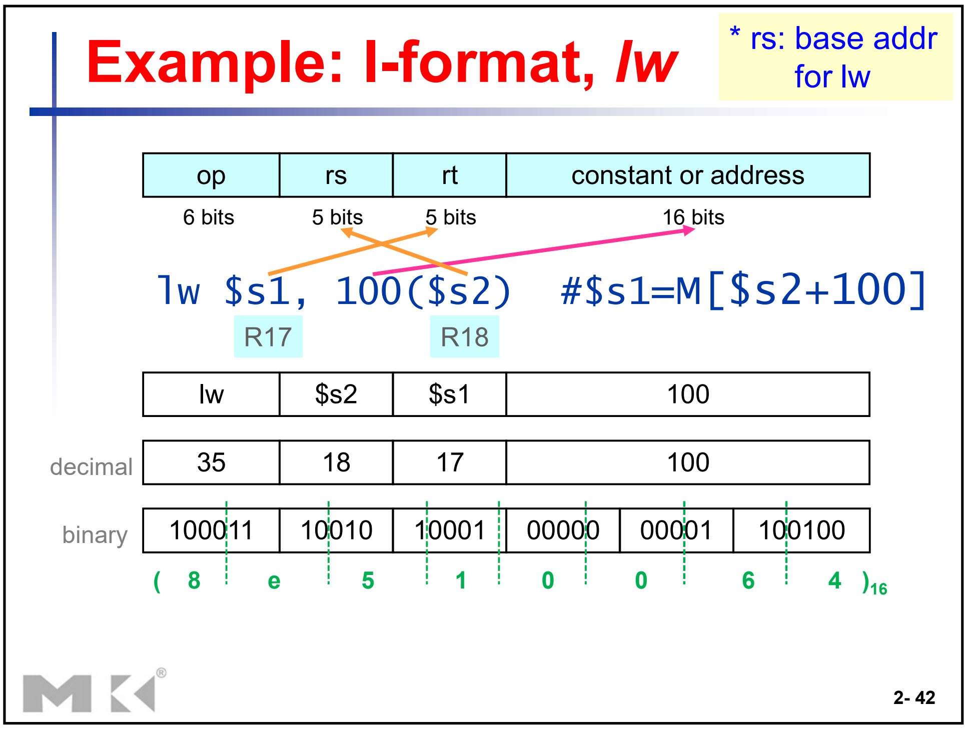
	addi	$s2	$s1	100
decimal	8	18	17	100

binary	001000	10010	10001	00000	00001	100100

$(2\ 2\ 5\ 1\ 0\ 0\ 6\ 4)_{16}$

2- 41

41



Example: I-format, *lw*

* rs: base addr for lw

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

```
lw $s1, 100($s2)   #$s1=M[$s2+100]
```

R17 R18

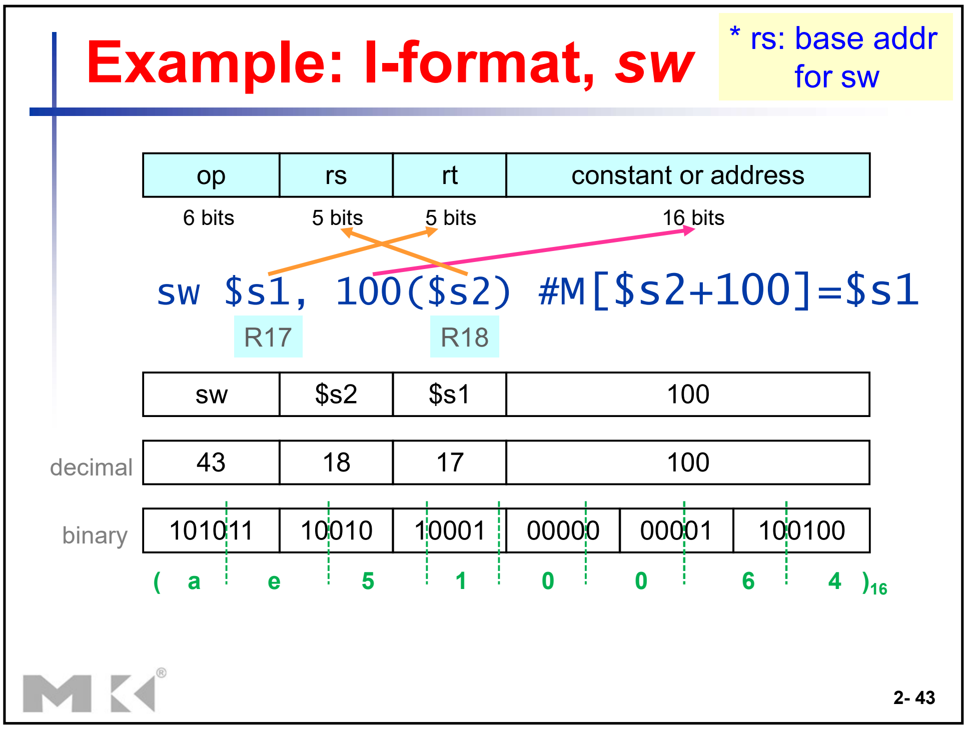
	lw	$s2	$s1	100
decimal	35	18	17	100

binary	100011	10010	10001	00000	00001	100100

$(8\ e\ 5\ 1\ 0\ 0\ 6\ 4)_{16}$

2- 42

42



Example: I-format, *sw*

* rs: base addr for sw

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

```
sw $s1, 100($s2) #M[$s2+100]=$s1
```

R17 R18

sw	$s2	$s1	100

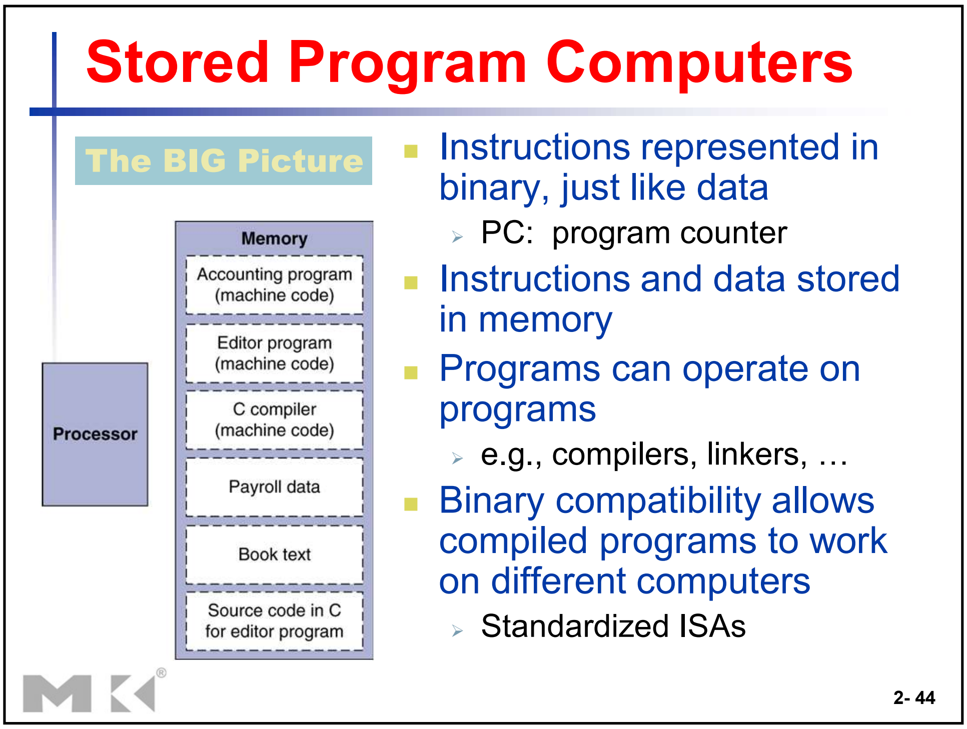
decimal	43	18	17	100

binary	101011	10010	10001	00000	00001	100100

$(a\ e\ 5\ 1\ 0\ 0\ 6\ 4)_{16}$

2- 43

43



Stored Program Computers

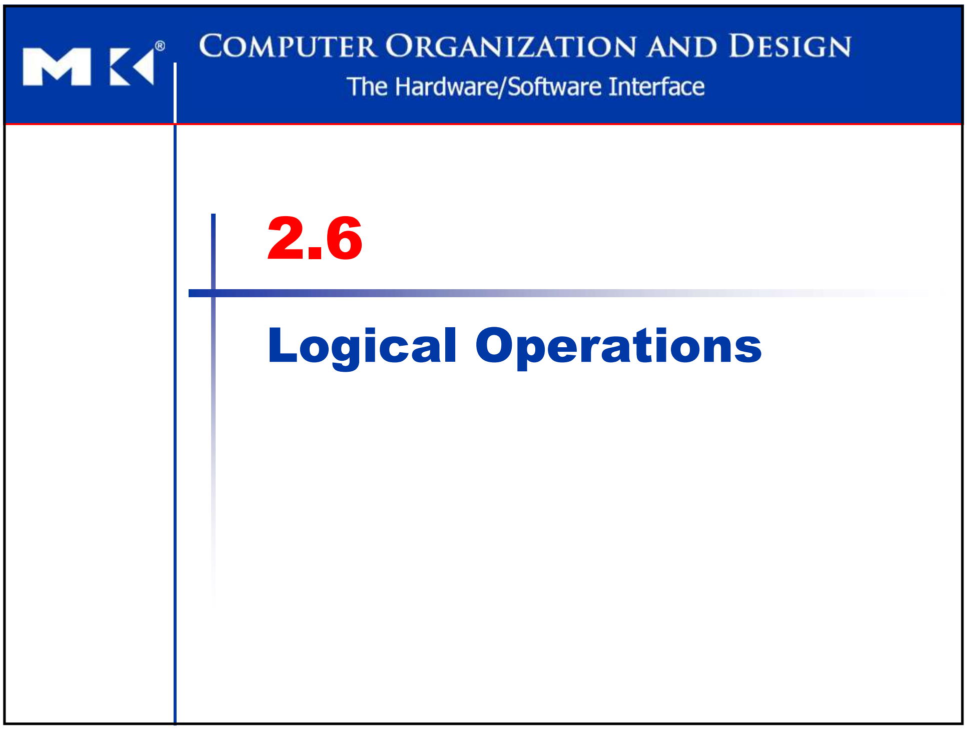
The BIG Picture

Memory
Accounting program (machine code)
Editor program (machine code)
C compiler (machine code)
Payroll data
Book text
Source code in C for editor program
Processor

- Instructions represented in binary, just like data
  - PC: program counter
- 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

2- 44

44

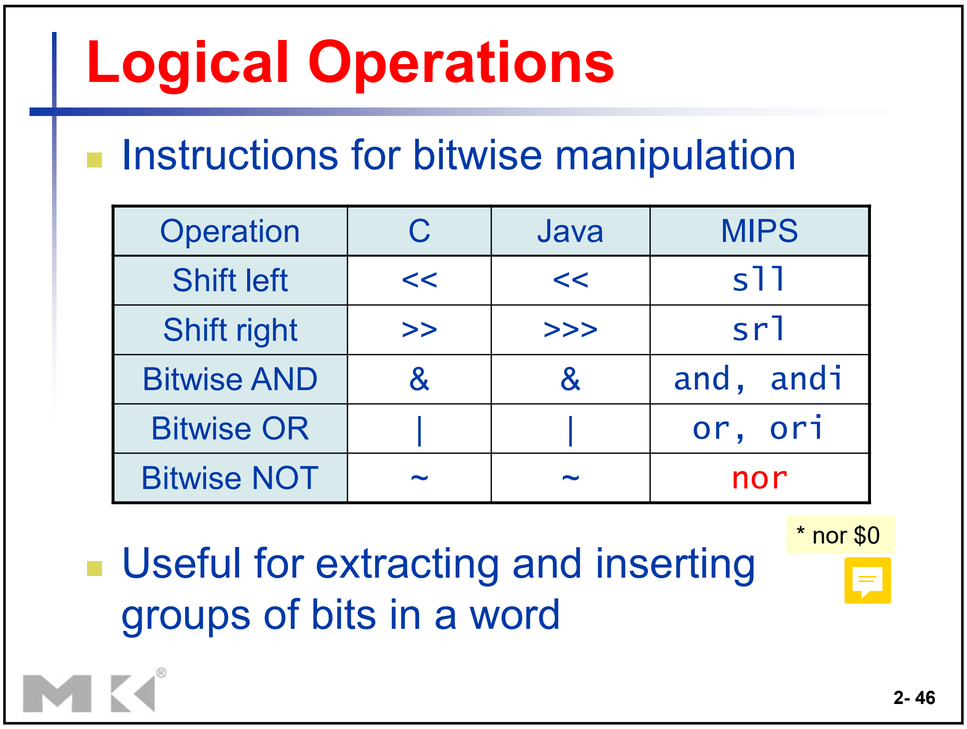


COMPUTER ORGANIZATION AND DESIGN
The Hardware/Software Interface

2.6

Logical Operations

45



Logical Operations

- Instructions for bitwise manipulation

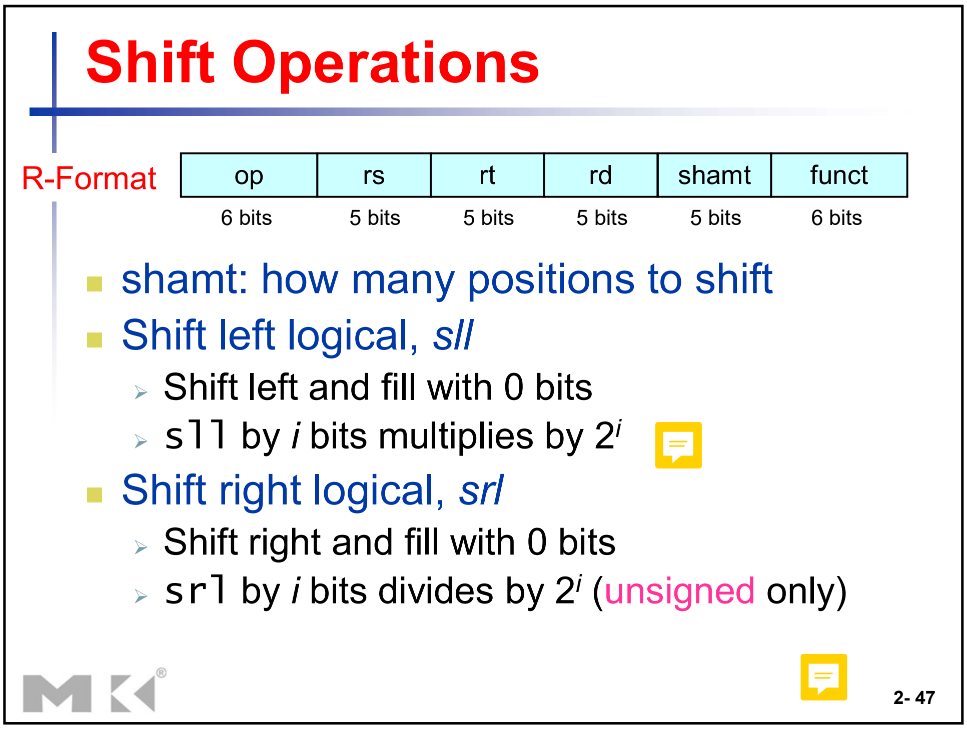
Operation	C	Java	MIPS
Shift left	<<	<<	`sll`
Shift right	>>	>>>	`srl`
Bitwise AND	&	&	`and, andi`
Bitwise OR	|	|	`or, ori`
Bitwise NOT	~	~	`nor`

* nor $0

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

2- 46

46



Shift Operations

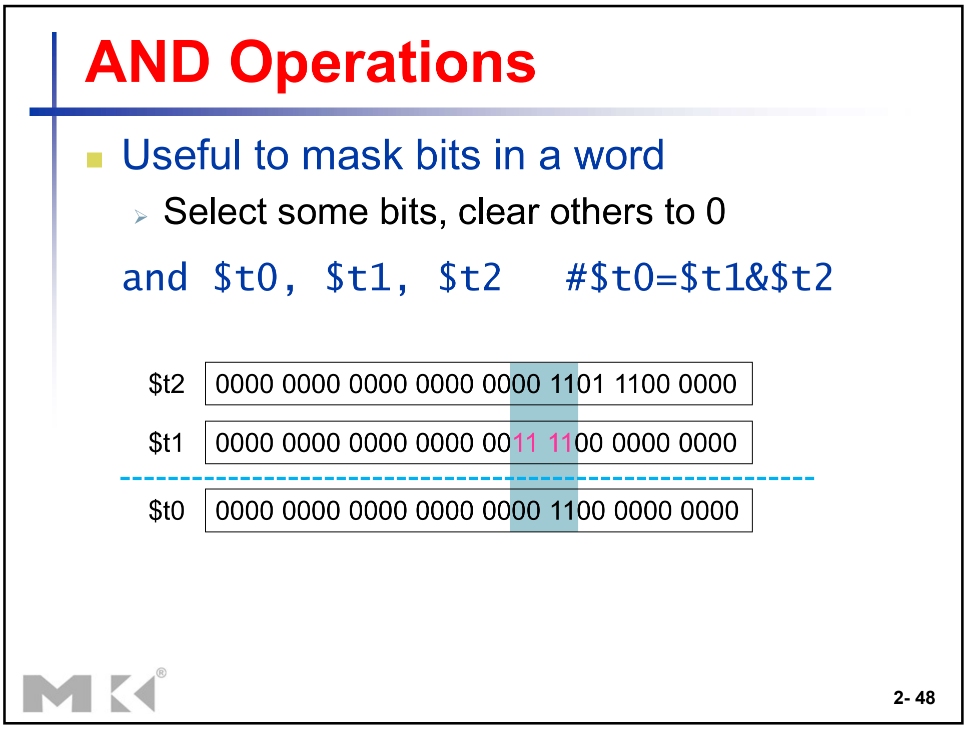
R-Format

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

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

2- 47

47



AND Operations

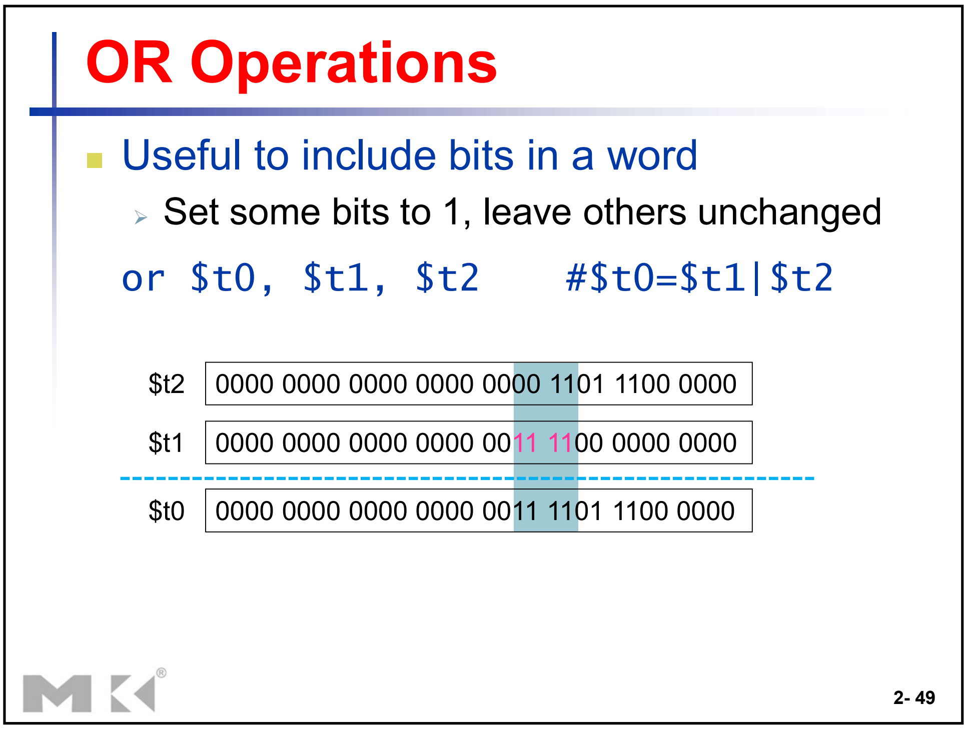
- Useful to mask bits in a word
  - Select some bits, clear others to 0

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

2- 48

48



OR Operations

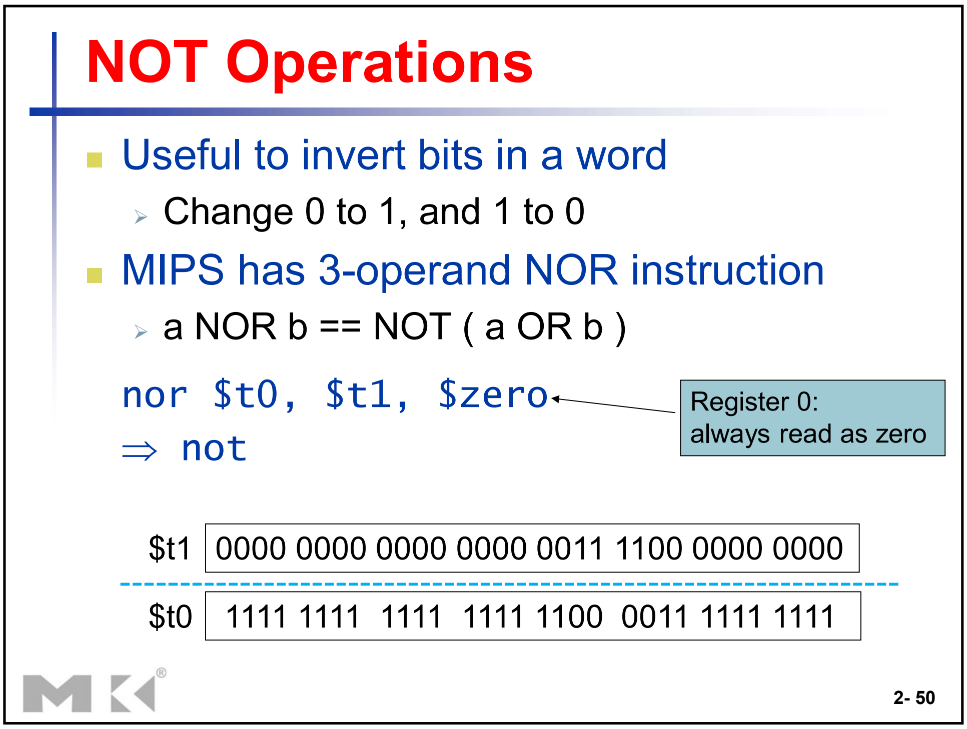
- Useful to include bits in a word
  - Set some bits to 1, leave others unchanged

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

2- 49

49



NOT Operations

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

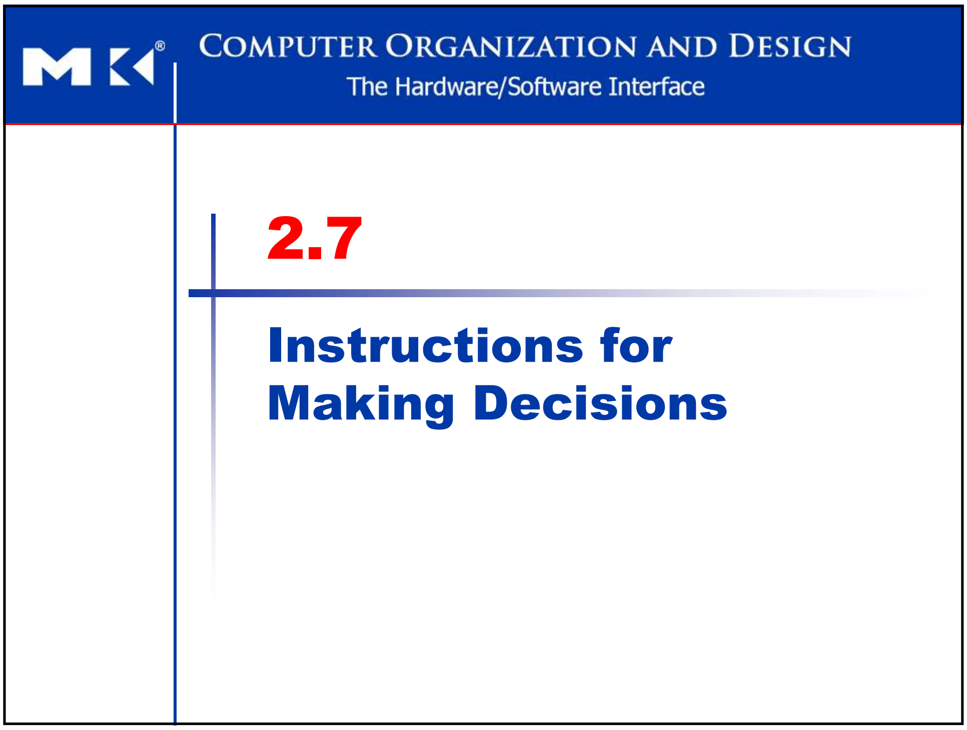
```
nor $t0, $t1, $zero
⇒ not
```

Register 0: always read as zero

```
$t1   0000 0000 0000 0000 0011 1100 0000 0000
-----------------------------------------------
$t0   1111 1111  1111  1111 1100  0011 1111 1111
```

2- 50

50

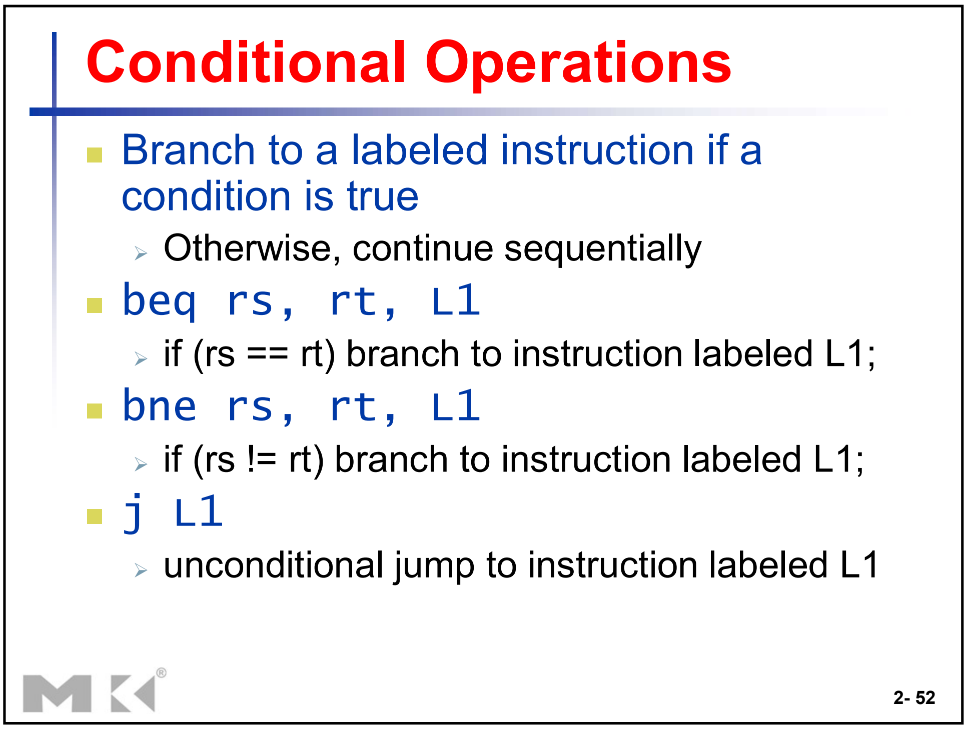


COMPUTER ORGANIZATION AND DESIGN
The Hardware/Software Interface

2.7

Instructions for Making Decisions

51

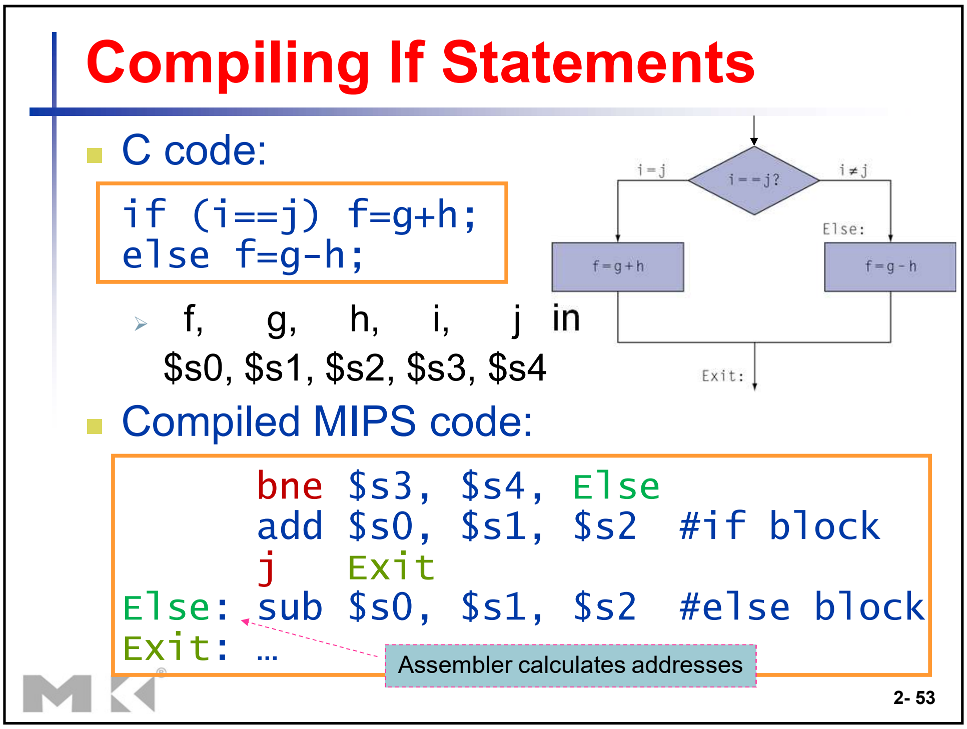


Conditional Operations

- 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

2- 52

52



Compiling If Statements

- C code:

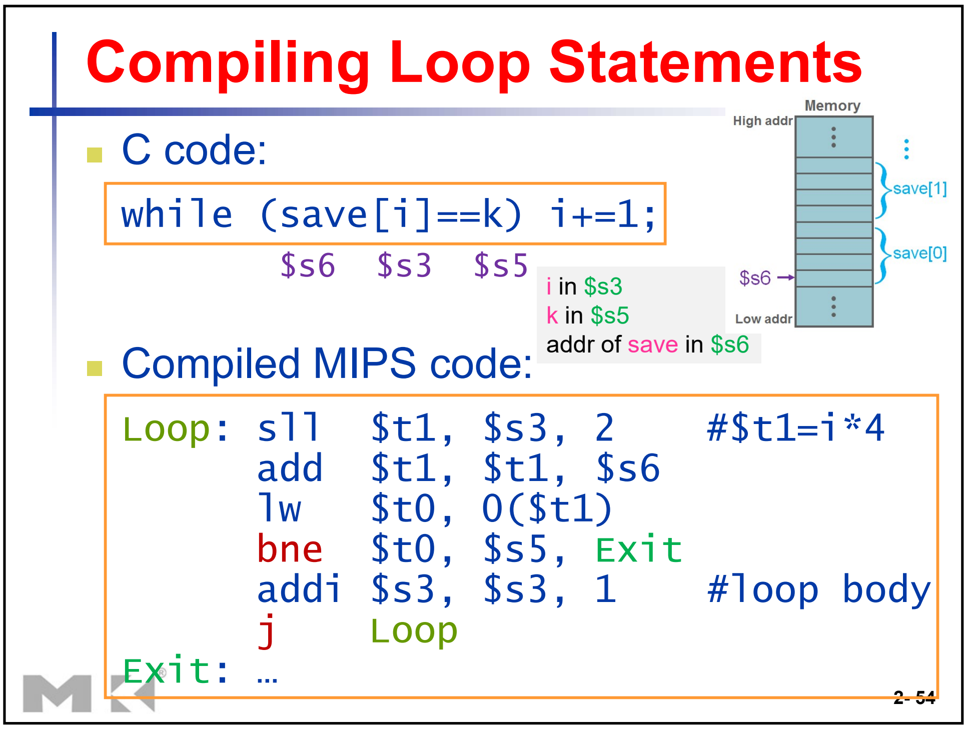
```
if (i==j) f=g+h;
else f=g-h;
```

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

- Compiled MIPS code:

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

Assembler calculates addresses



Compiling Loop Statements

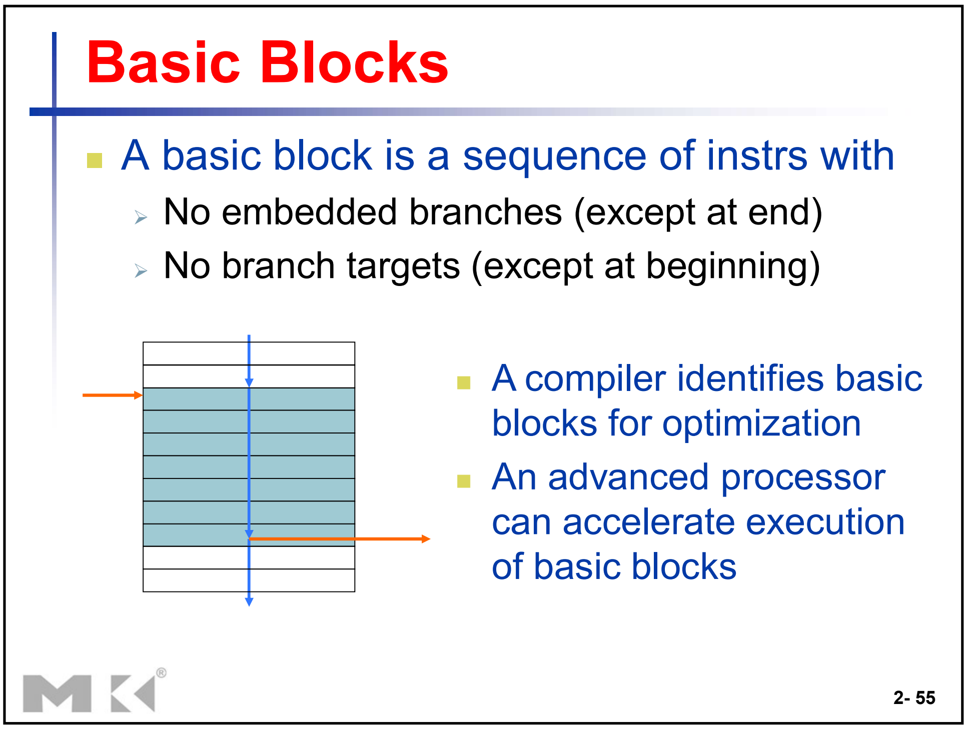
- C code:

```
while (save[i]==k) i+=1;
          $s6  $s3  $s5
```

i in $s3
k in $s5
addr of save in $s6

- Compiled MIPS code:

```
Loop: sll  $t1, $s3, 2    #$t1=i*4
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1    #loop body
      j    Loop
Exit: …
```



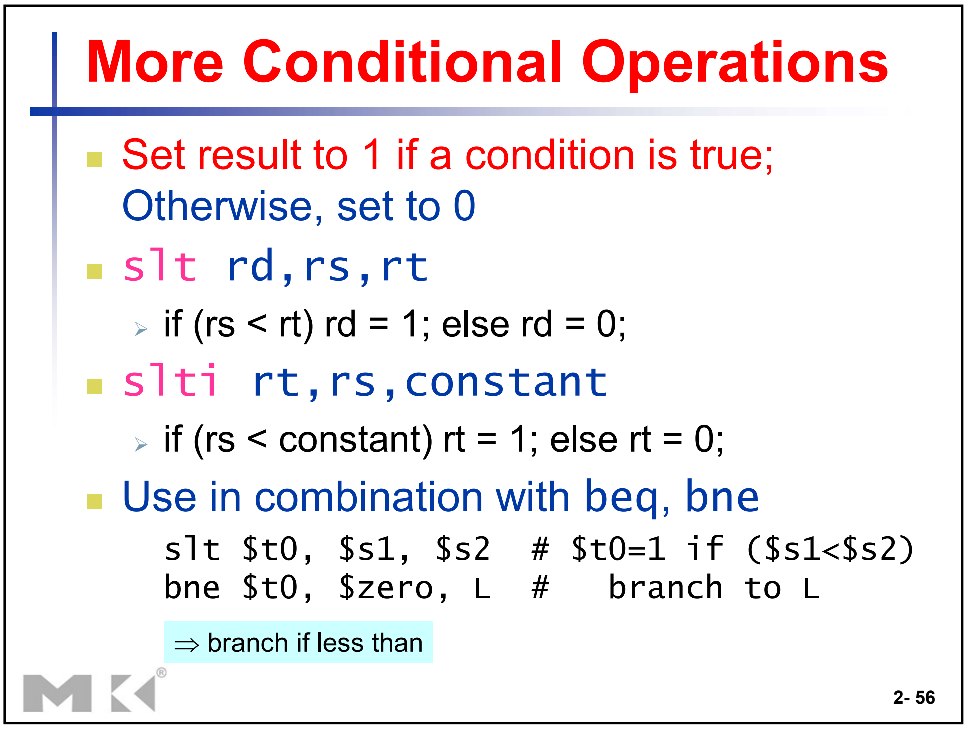
Basic Blocks

- A basic block is a sequence of instrs 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

2- 55

55



More Conditional Operations

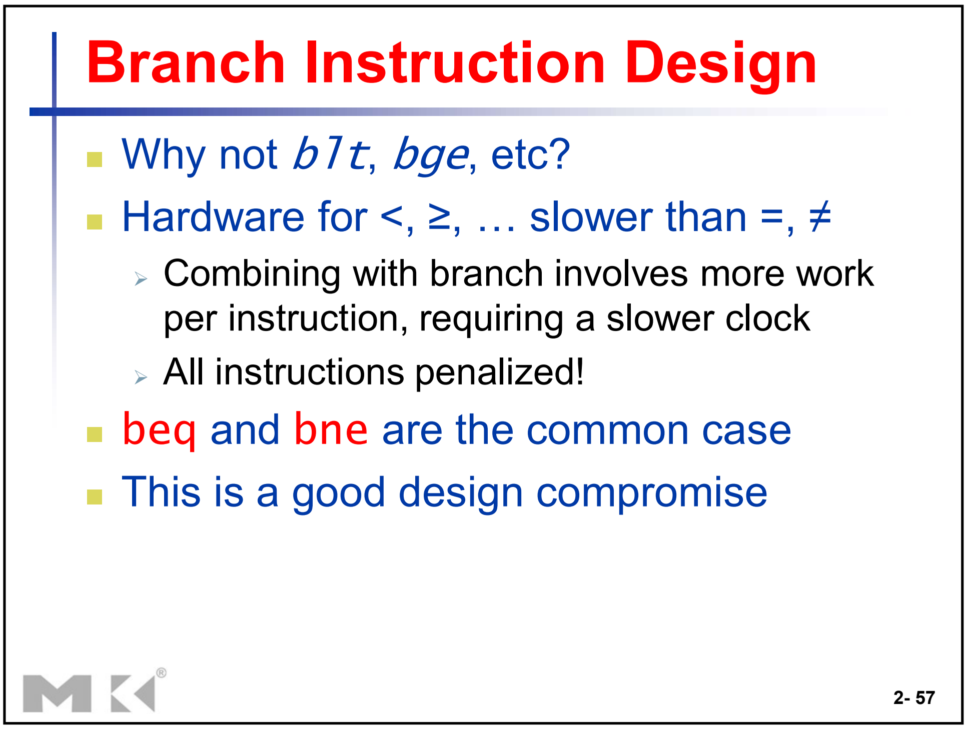
- Set result to 1 if a condition is true; Otherwise, set to 0
- `slt rd,rs,rt`
  - if (rs < rt) rd = 1; else rd = 0;
- `slti rt,rs,constant`
  - if (rs < constant) rt = 1; else rt = 0;
- Use in combination with beq, bne

```
slt $t0, $s1, $s2  # $t0=1 if ($s1<$s2)
bne $t0, $zero, L  #   branch to L
```

⇒ branch if less than

2- 56

56

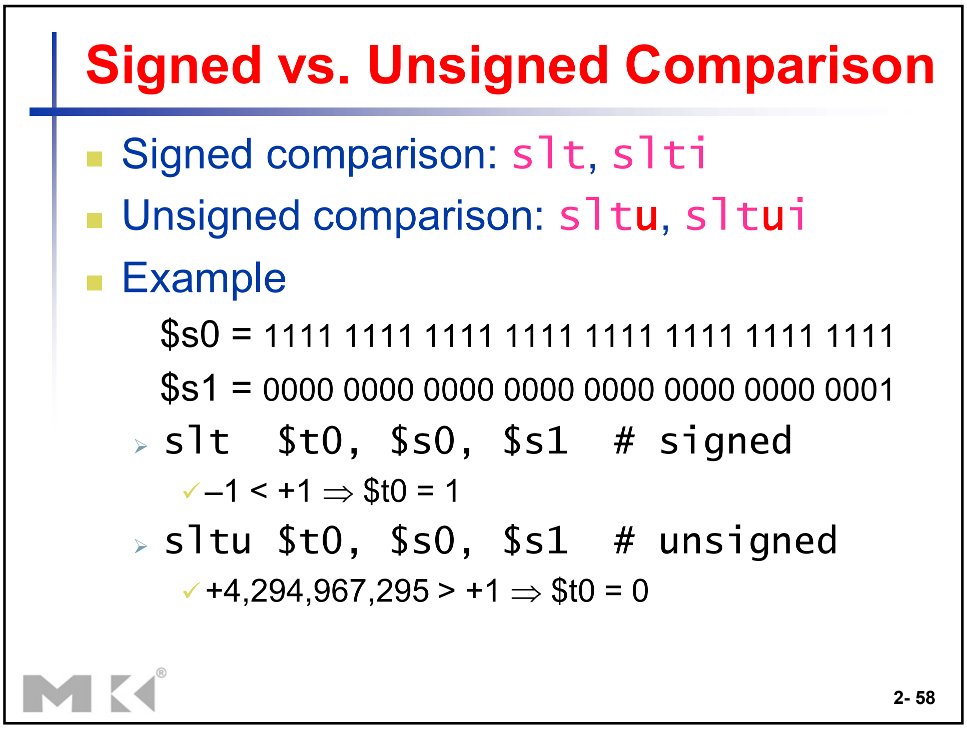


Branch Instruction Design

- 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

2- 57

57



Signed vs. Unsigned Comparison

- 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

2- 58

58

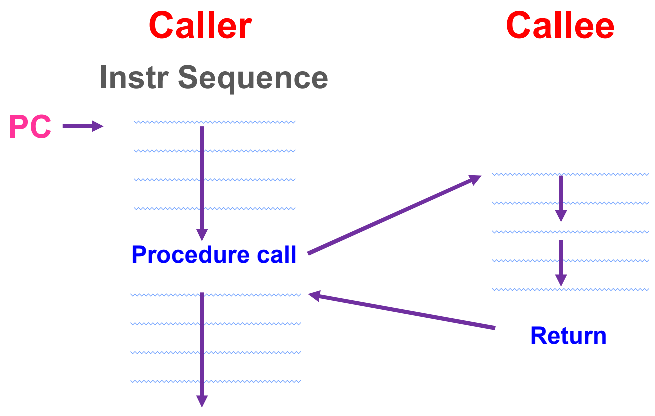
COMPUTER ORGANIZATION AND DESIGN
The Hardware/Software Interface

2.8

Supporting Procedures in Computer Hardware

59

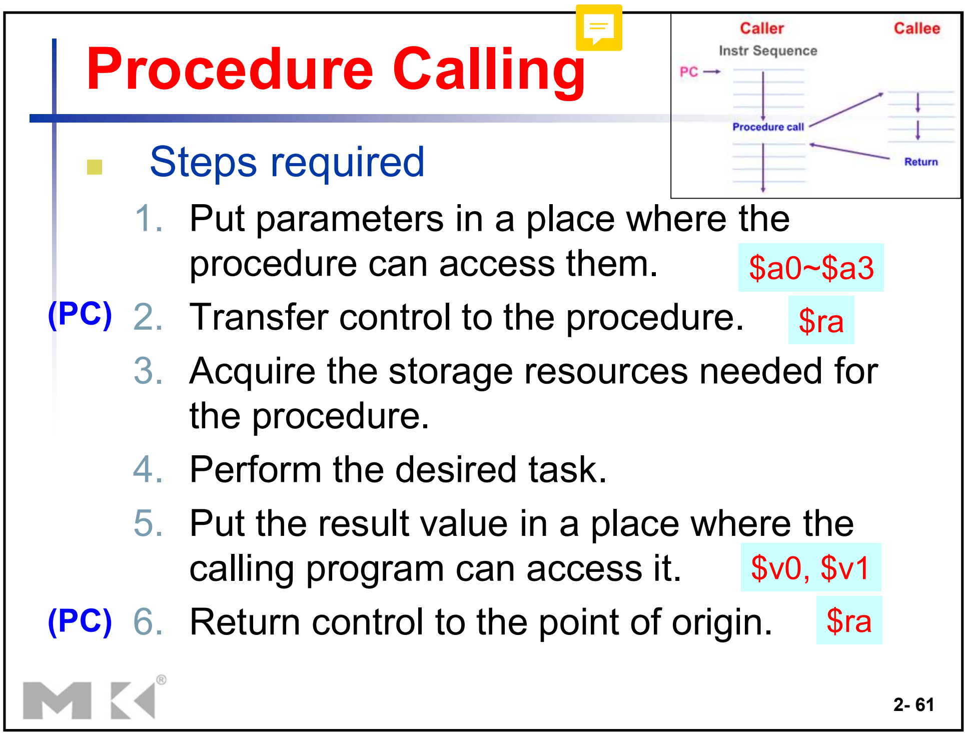
Procedure Calling



* PC: program counter

2- 60

60



Procedure Calling

Caller

Callee

Instr Sequence

PC →

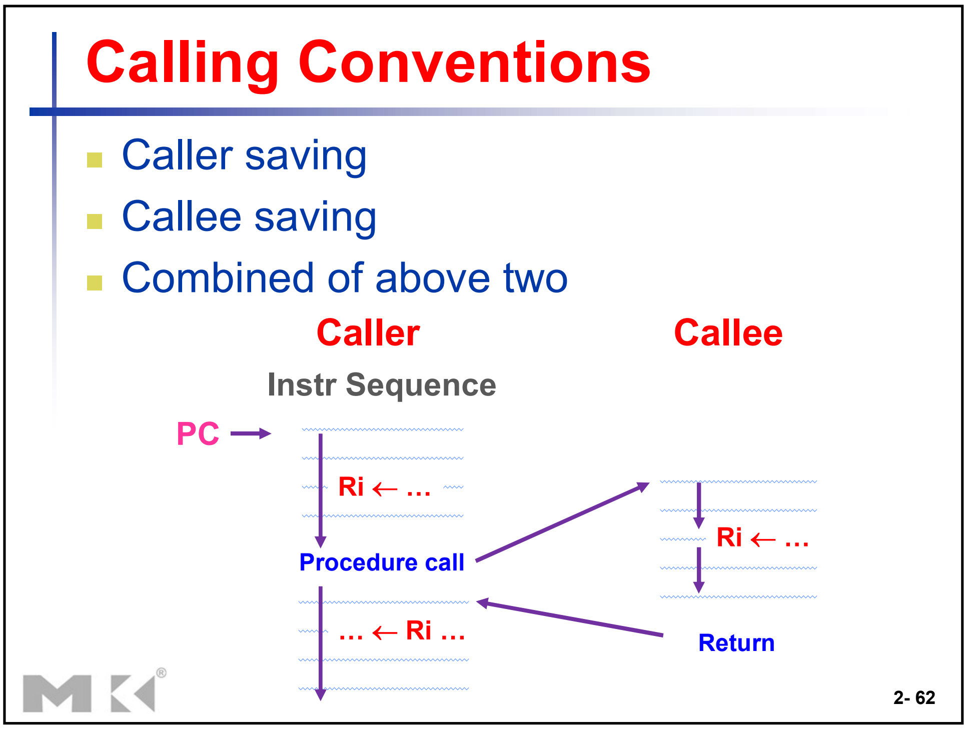
Procedure call

Return

- Steps required
  1. Put parameters in a place where the procedure can access them. $a0~$a3
  2. (PC) Transfer control to the procedure. $ra
  3. Acquire the storage resources needed for the procedure.
  4. Perform the desired task.
  5. Put the result value in a place where the calling program can access it. $v0, $v1
  6. (PC) Return control to the point of origin. $ra

2- 61

61



Calling Conventions

- Caller saving
- Callee saving
- Combined of above two

Caller

Callee

Instr Sequence

PC →

Ri ← ...

Procedure call

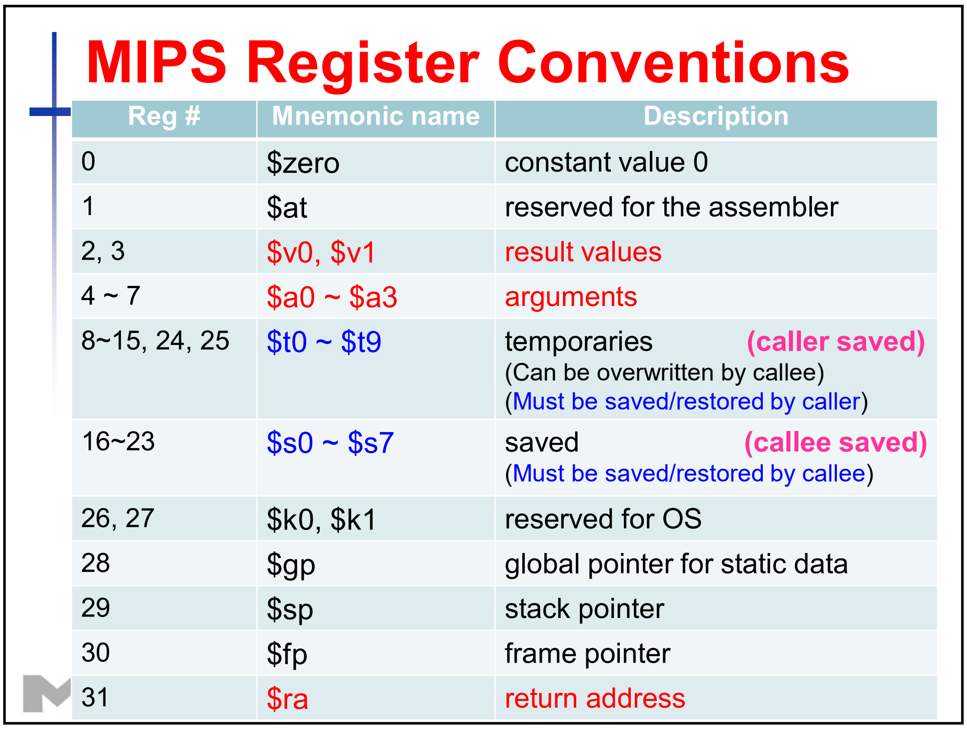
Ri ← ...

... ← Ri ...

Return

2- 62

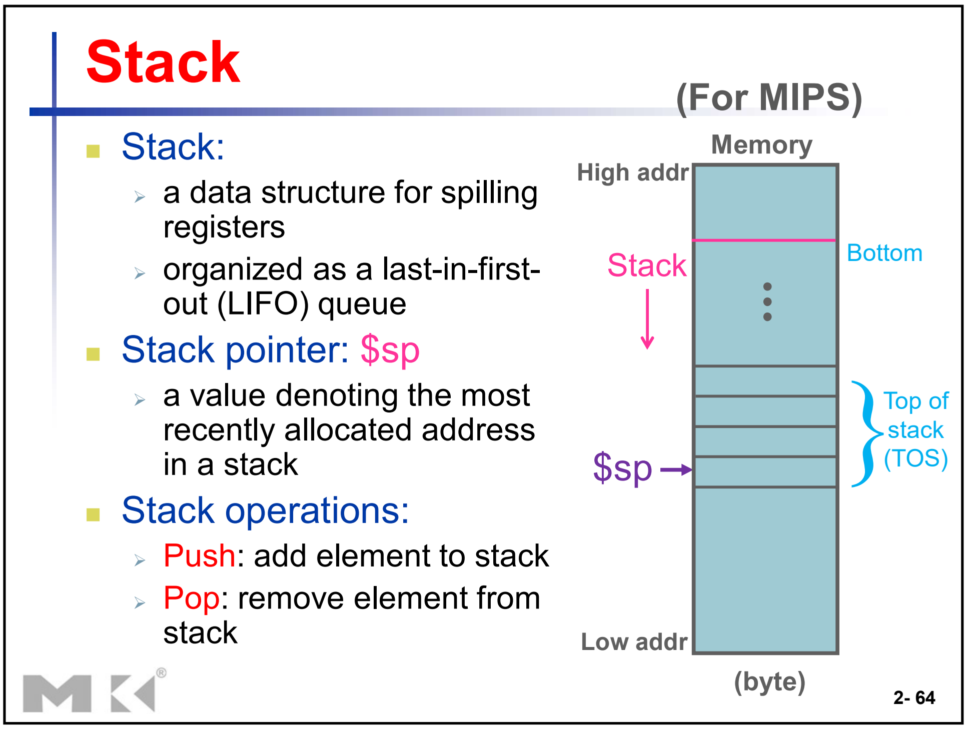
62



MIPS Register Conventions

Reg #	Mnemonic name	Description
0	$zero	constant value 0
1	$at	reserved for the assembler
2, 3	$v0, $v1	result values
4 ~ 7	$a0 ~ $a3	arguments
8~15, 24, 25	$t0 ~ $t9	temporaries **(caller saved)** (Can be overwritten by callee) (Must be saved/restored by caller)
16~23	$s0 ~ $s7	saved **(callee saved)** (Must be saved/restored by callee)
26, 27	$k0, $k1	reserved for OS
28	$gp	global pointer for static data
29	$sp	stack pointer
30	$fp	frame pointer
31	$ra	return address

63

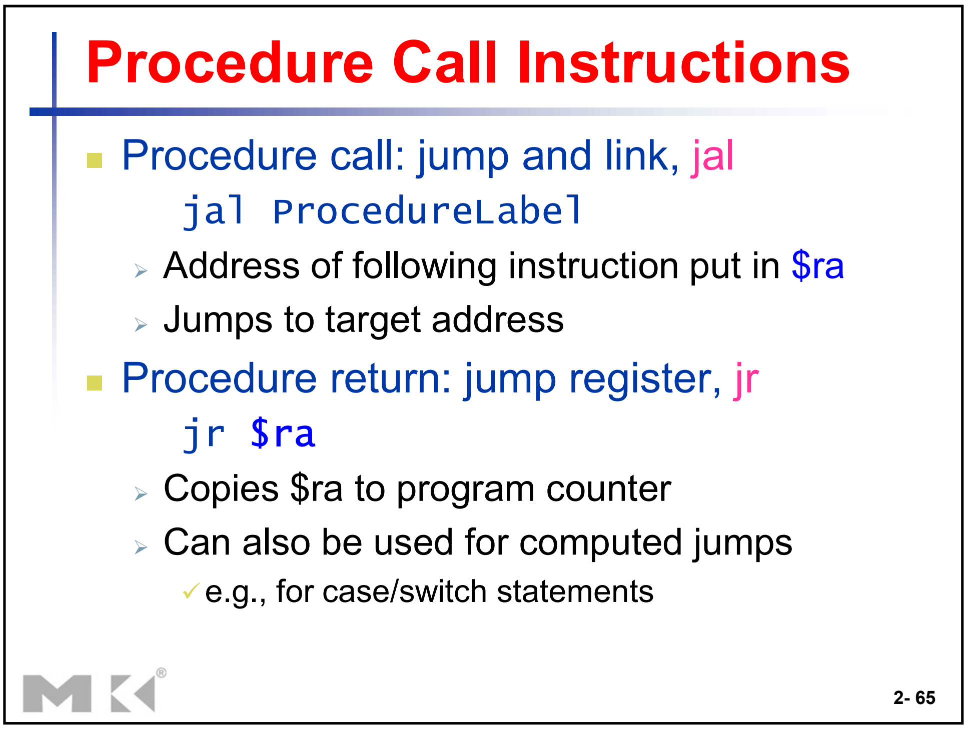


Stack

- Stack:
  - a data structure for spilling registers
  - organized as a last-in-first-out (LIFO) queue
- Stack pointer: $sp
  - a value denoting the most recently allocated address in a stack
- Stack operations:
  - Push: add element to stack
  - Pop: remove element from stack

2- 64

64



Procedure Call Instructions

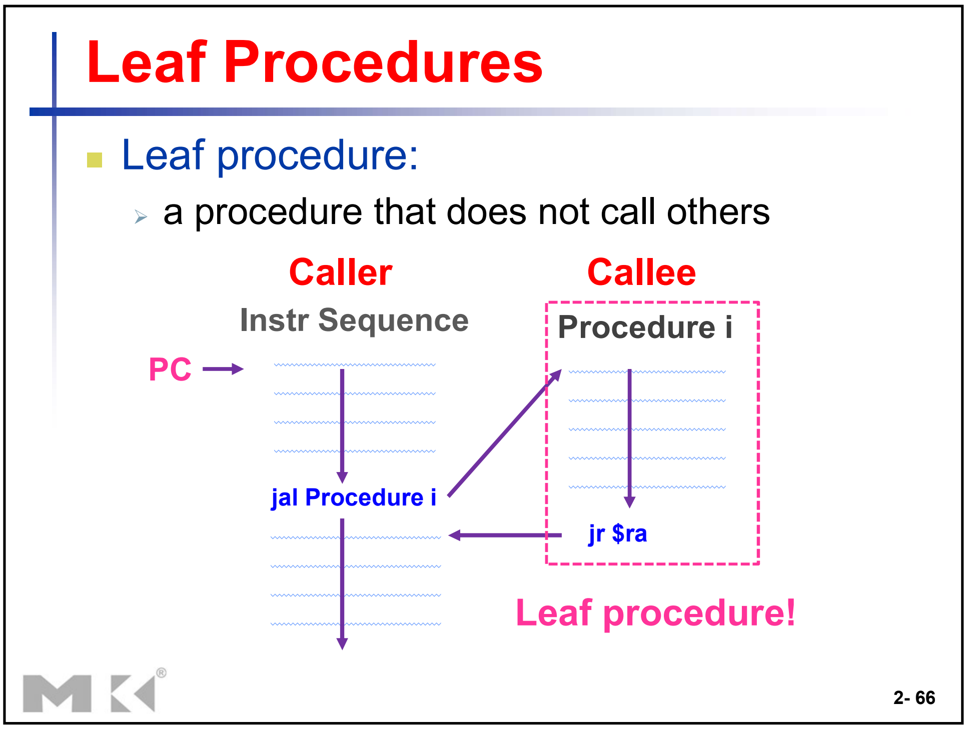
- Procedure call: jump and link, jal

  ```
  jal ProcedureLabel
  ```

  - Address of following instruction put in $ra
  - Jumps to target address
- Procedure return: jump register, jr

  ```
  jr $ra
  ```

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



Leaf Procedures

- Leaf procedure:
  - a procedure that does not call others

Caller

Instr Sequence

PC

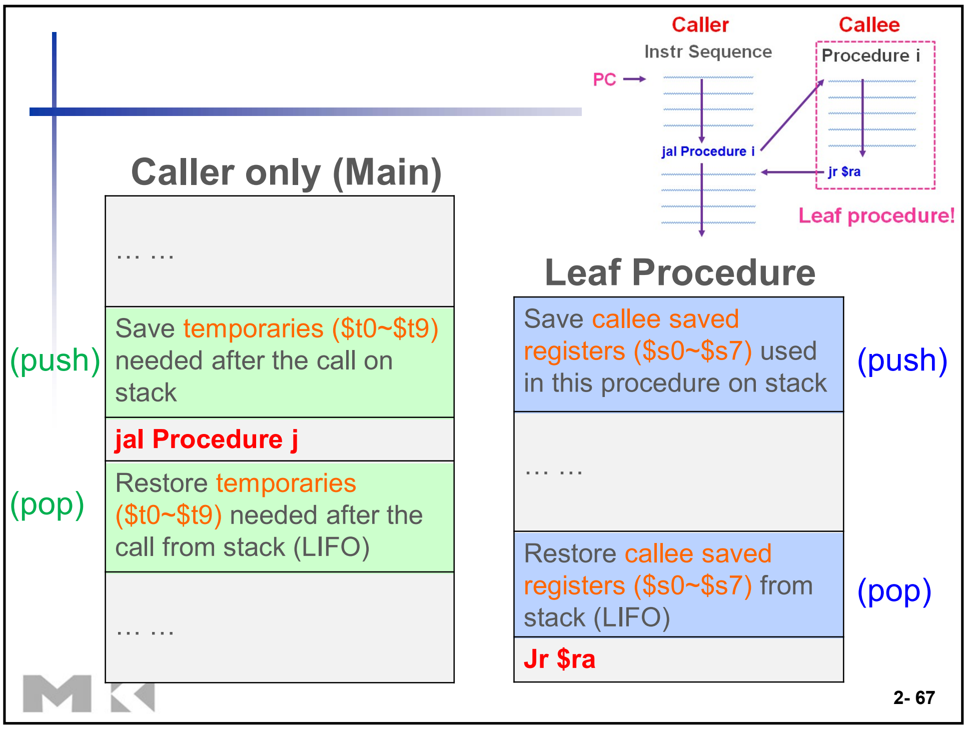
jal Procedure i

Callee

Procedure i

jr $ra

Leaf procedure!



Caller only (Main)

… …

(push) Save temporaries ($t0~$t9) needed after the call on stack

jal Procedure j

(pop) Restore temporaries ($t0~$t9) needed after the call from stack (LIFO)

… …

Leaf Procedure

Save callee saved registers ($s0~$s7) used in this procedure on stack (push)

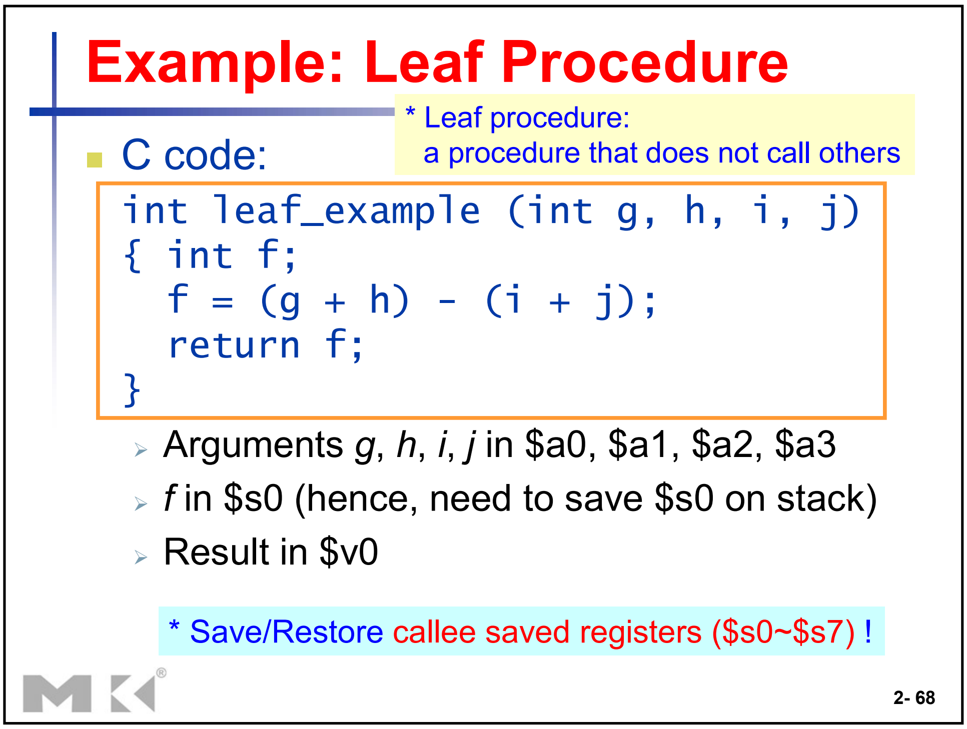
… …

Restore callee saved registers ($s0~$s7) from stack (LIFO) (pop)

Jr $ra

2- 67

67



Example: Leaf Procedure

* Leaf procedure:
a procedure that does not call others

- C code:

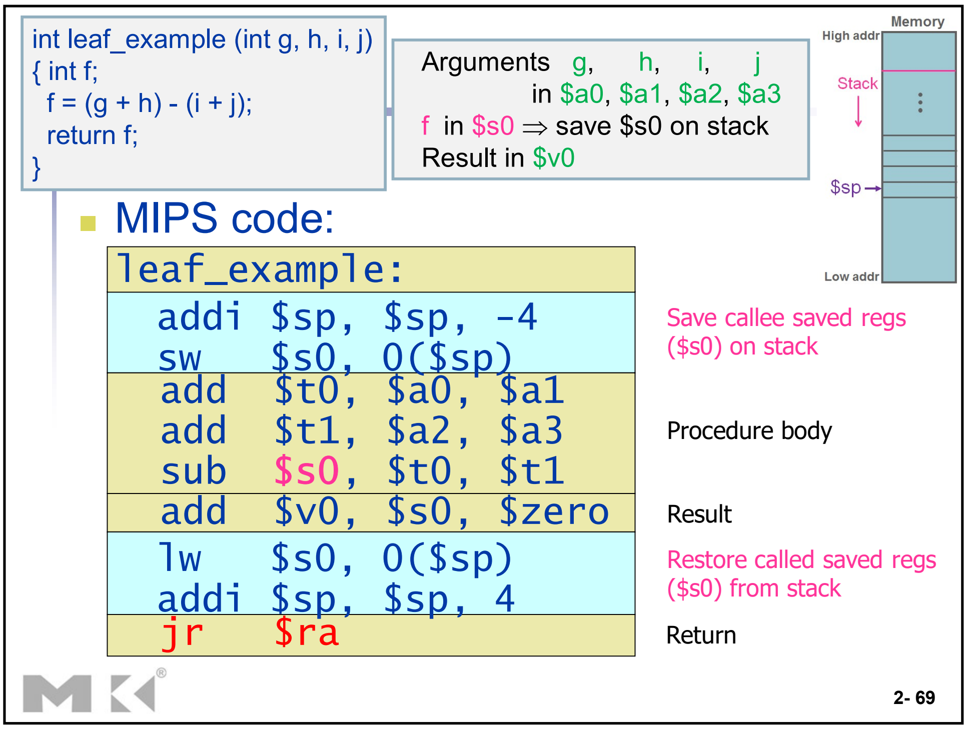
```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

- Arguments *g*, *h*, *i*, *j* in $a0, $a1, $a2, $a3
- *f* in $s0 (hence, need to save $s0 on stack)
- Result in $v0

* Save/Restore callee saved registers ($s0~$s7) !

2- 68

68



```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

Arguments g, h, i, j in $a0, $a1, $a2, $a3

f in $s0 ⇒ save $s0 on stack

Result in $v0

Memory
High addr
Stack
$sp
Low addr

- 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 callee saved regs ($s0) on stack

Procedure body

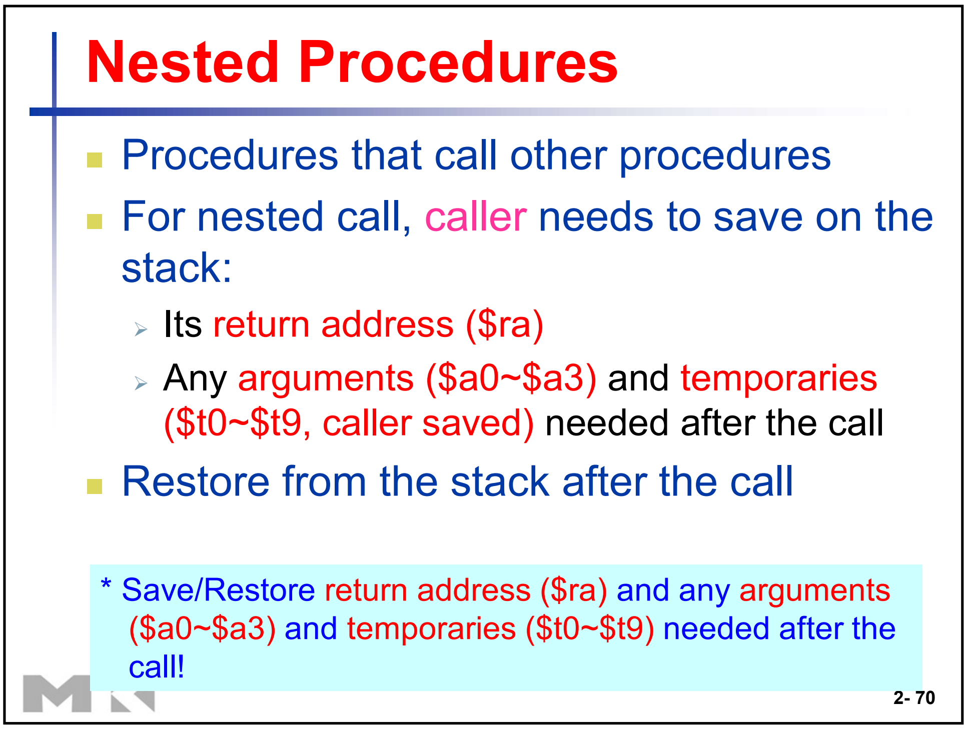
Result

Restore called saved regs ($s0) from stack

Return

2- 69

69



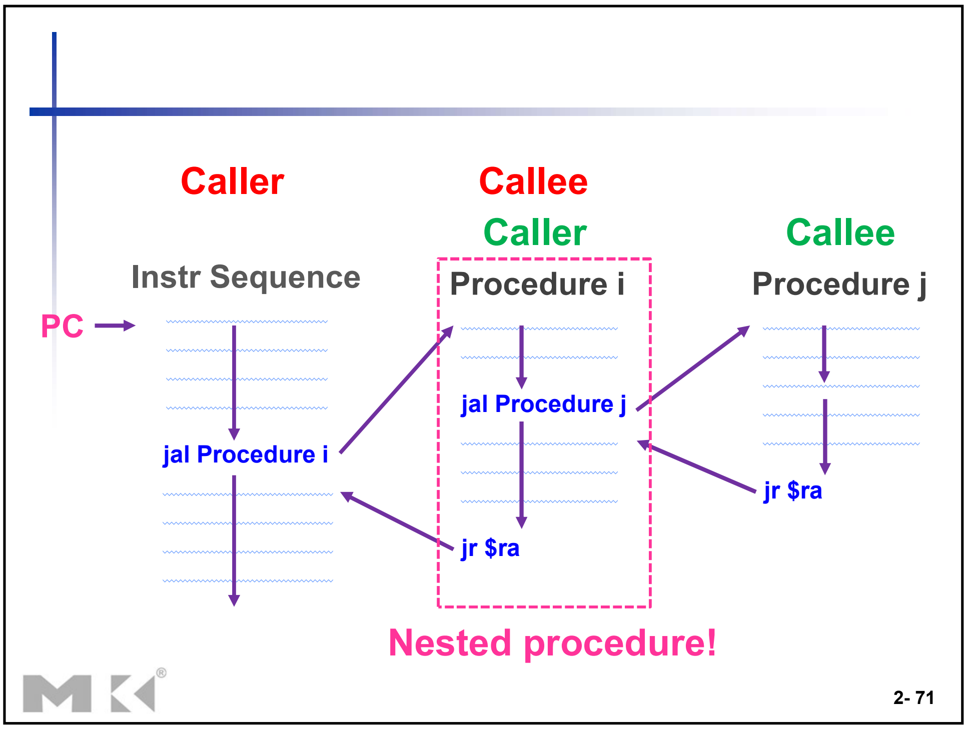
Nested Procedures

- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
  - Its return address ($ra)
  - Any arguments ($a0~$a3) and temporaries ($t0~$t9, caller saved) needed after the call
- Restore from the stack after the call

* Save/Restore return address ($ra) and any arguments ($a0~$a3) and temporaries ($t0~$t9) needed after the call!

2- 70

70



Caller

Callee

Caller

Callee

Instr Sequence

Procedure i

Procedure j

PC

jal Procedure i

jal Procedure j

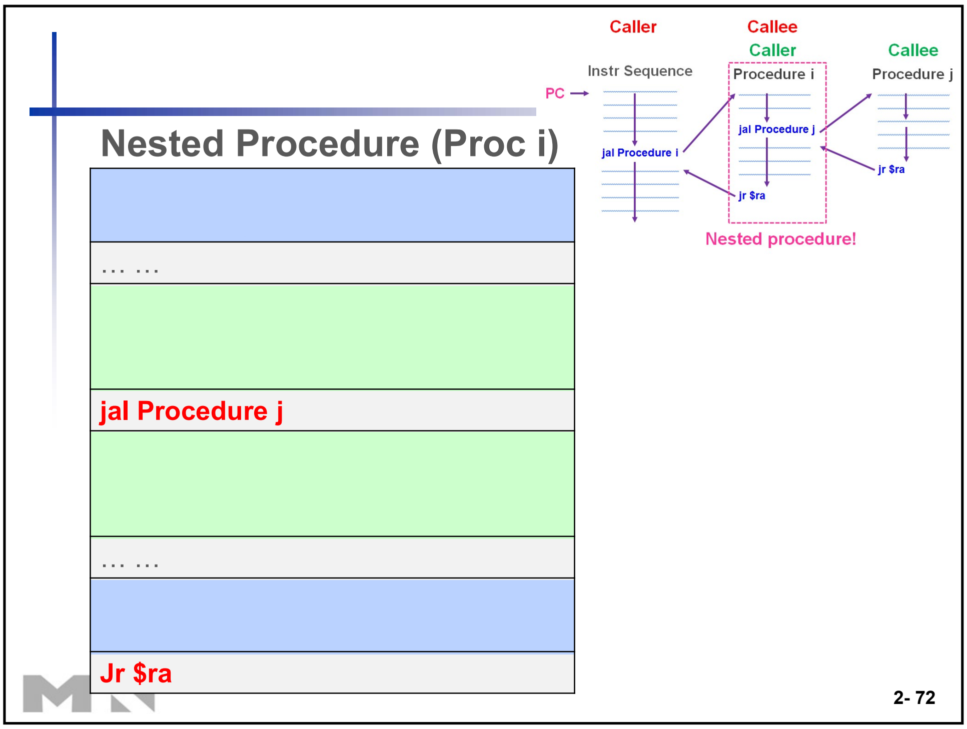
jr $ra

jr $ra

Nested procedure!

2- 71

71



Nested Procedure (Proc i)

… …

jal Procedure j

… …

Jr $ra

Caller

Callee

Caller

Callee

Instr Sequence

Procedure i

Procedure j

PC

jal Procedure i

jal Procedure j

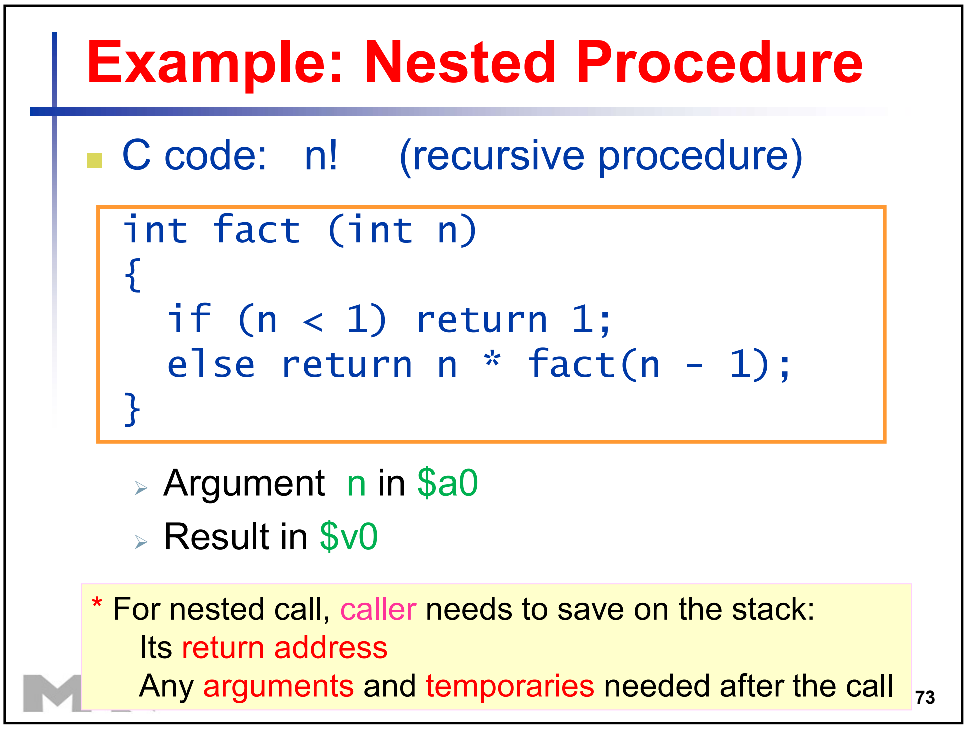
jr $ra

jr $ra

Nested procedure!

2- 72

72



Example: Nested Procedure

- C code: n! (recursive procedure)

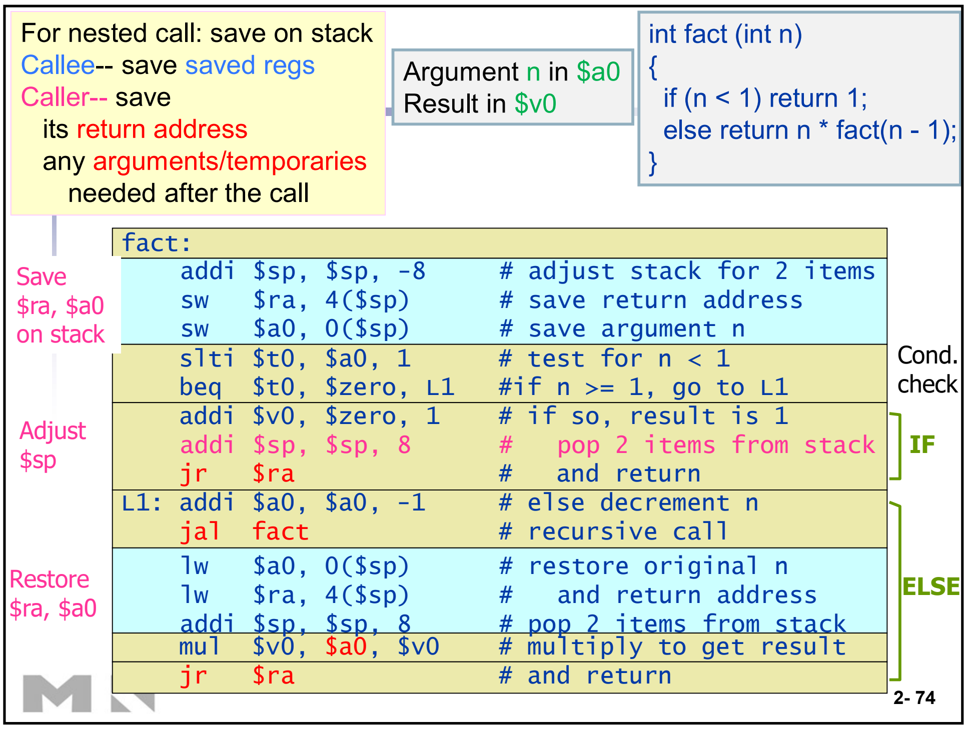
```
int fact (int n)
{
  if (n < 1) return 1;
  else return n * fact(n - 1);
}
```

- Argument n in $a0
- Result in $v0

* For nested call, caller needs to save on the stack:
Its return address
Any arguments and temporaries needed after the call

73

73



For nested call: save on stack
Callee-- save saved regs
Caller-- save
its return address
any arguments/temporaries
needed after the call

Argument n in $a0
Result in $v0

```
int fact (int n)
{
  if (n < 1) return 1;
  else return n * fact(n - 1);
}
```

Save $ra, $a0 on stack

Adjust $sp

Restore $ra, $a0

```
fact:
      addi $sp, $sp, -8     # adjust stack for 2 items
      sw   $ra, 4($sp)      # save return address
      sw   $a0, 0($sp)      # save argument n
      slti $t0, $a0, 1      # test for n < 1
      beq  $t0, $zero, L1   #if n >= 1, go to 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
```

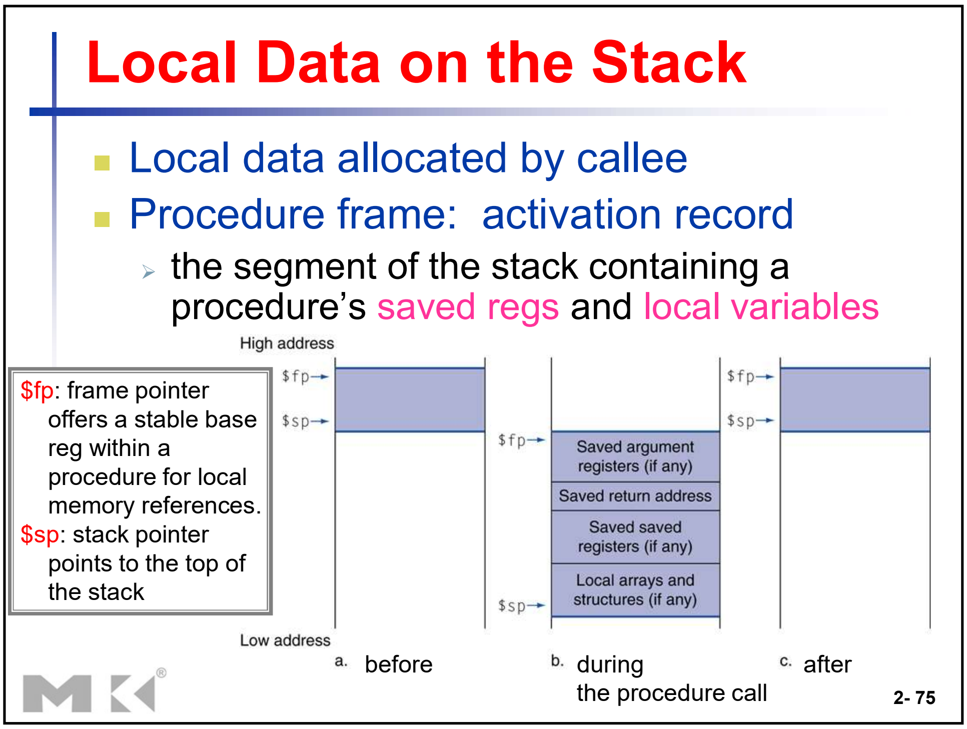
Cond. check

IF

ELSE

2- 74

74



Local Data on the Stack

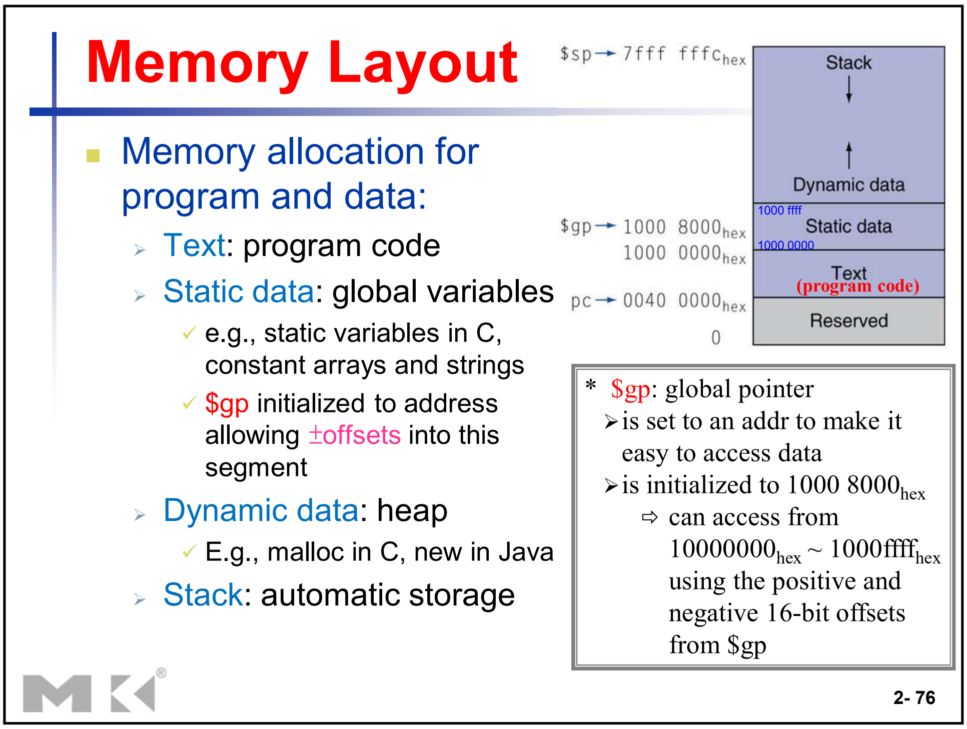
- Local data allocated by callee
- Procedure frame: activation record
  - the segment of the stack containing a procedure's saved regs and local variables

$fp: frame pointer offers a stable base reg within a procedure for local memory references.
$sp: stack pointer points to the top of the stack

High address
$fp
$sp
Low address
a. before
$fp
Saved argument registers (if any)
Saved return address
Saved saved registers (if any)
Local arrays and structures (if any)
$sp
b. during the procedure call
$fp
$sp
c. after

2- 75

75



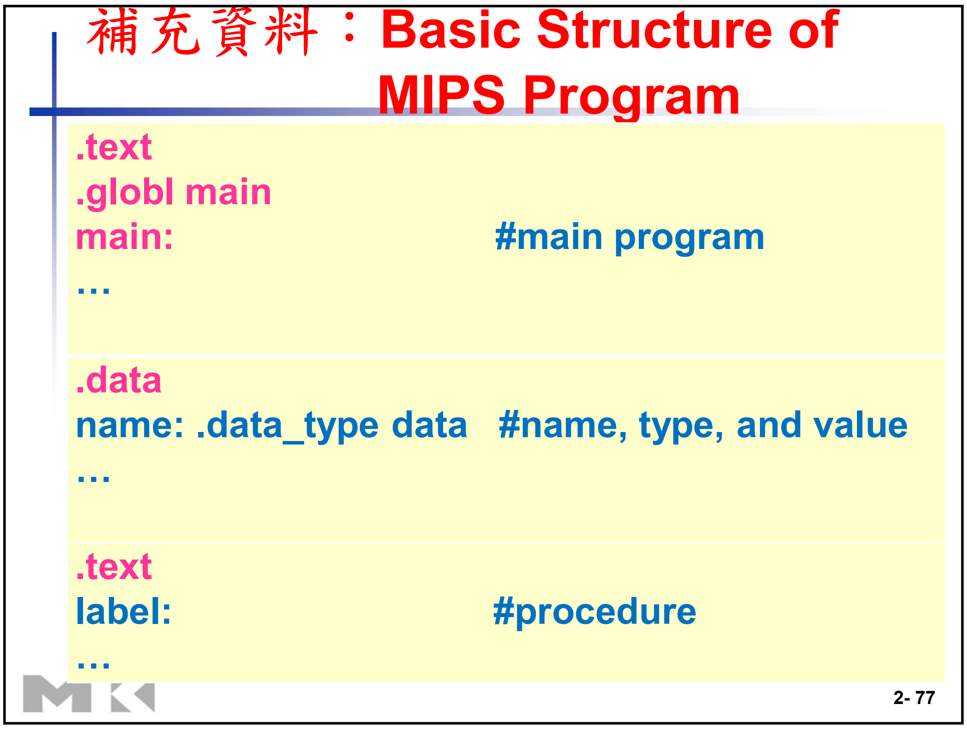
Memory Layout

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

* $gp: global pointer
- is set to an addr to make it easy to access data
- is initialized to 1000 8000$_{hex}$
  - ⇨ can access from 10000000$_{hex}$ ~ 1000ffff$_{hex}$ using the positive and negative 16-bit offsets from $gp

2- 76

76

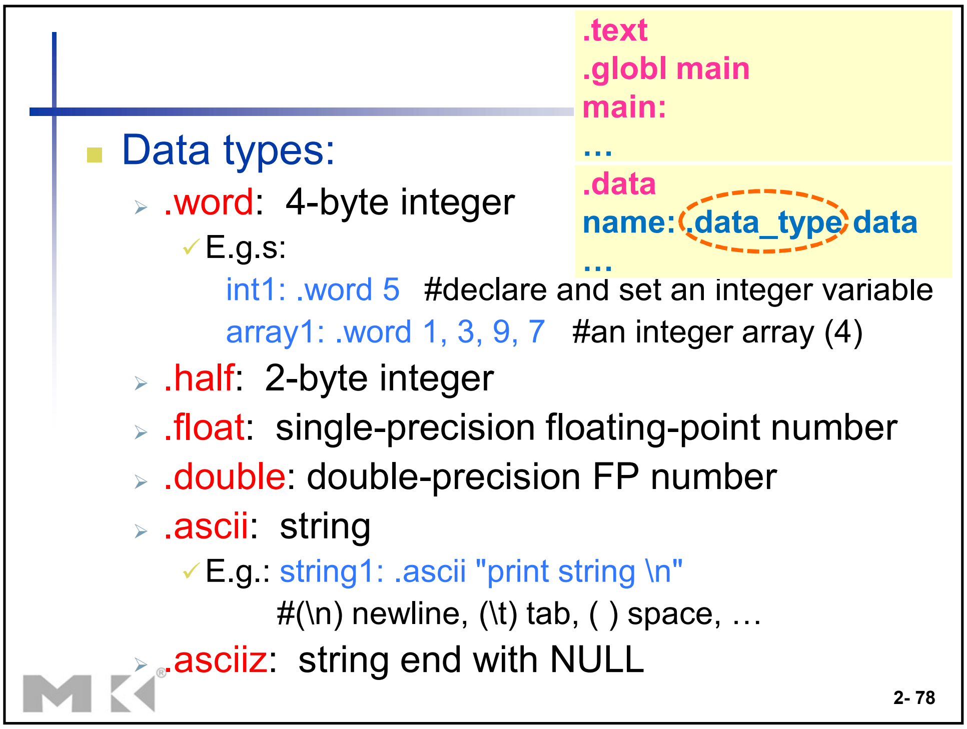


補充資料：Basic Structure of MIPS Program

```
.text
.globl main
main:                          #main program
…

.data
name: .data_type data   #name, type, and value
…

.text
label:                         #procedure
…
```

2- 77

77



```
.text
.globl main
main:
…
.data
name: .data_type data
…
```

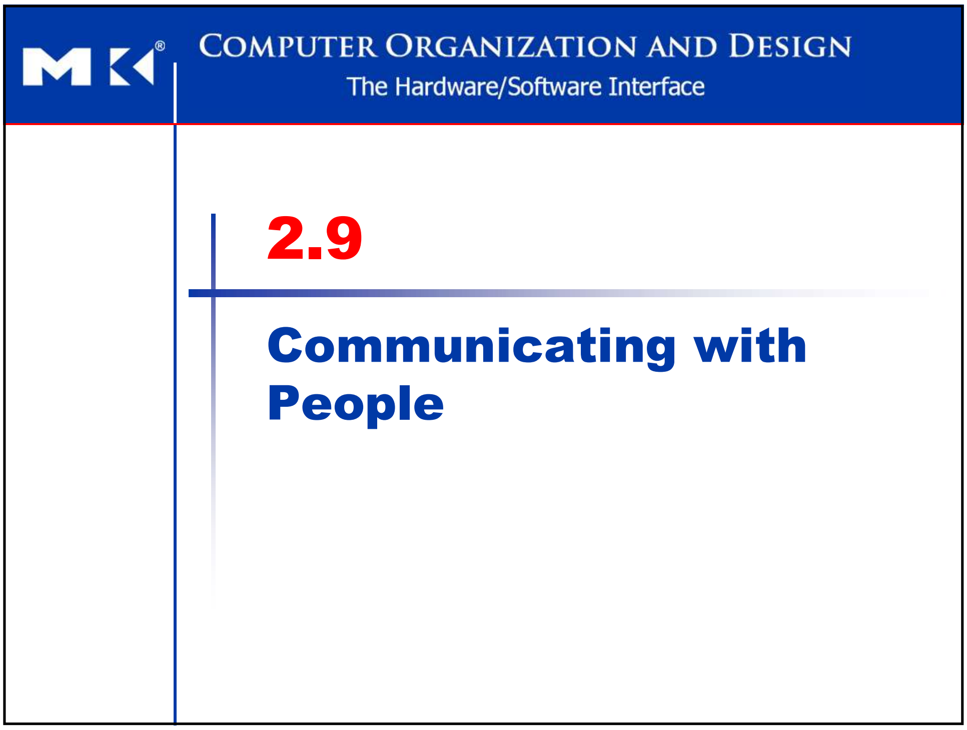
- Data types:
  - .word: 4-byte integer
    - ✓ E.g.s:

      ```
      int1: .word 5   #declare and set an integer variable
      array1: .word 1, 3, 9, 7   #an integer array (4)
      ```
  - .half: 2-byte integer
  - .float: single-precision floating-point number
  - .double: double-precision FP number
  - .ascii: string
    - ✓ E.g.: string1: .ascii "print string \n"

      #(\n) newline, (\t) tab, ( ) space, …
  - .asciiz: string end with NULL

2- 78

78



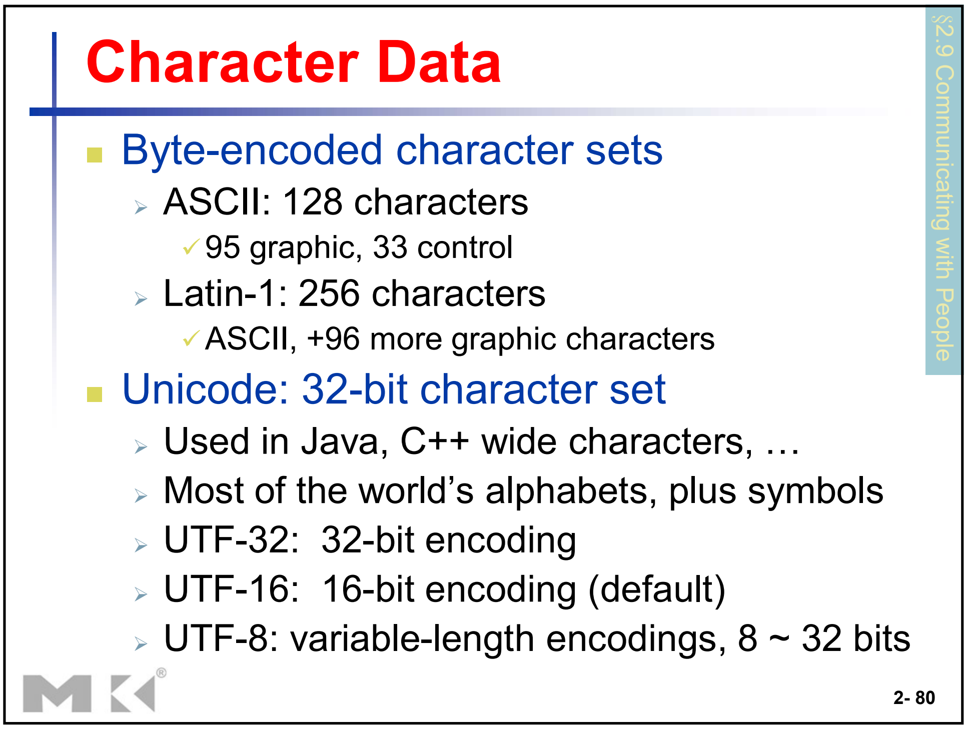
COMPUTER ORGANIZATION AND DESIGN

The Hardware/Software Interface

2.9

Communicating with People

79



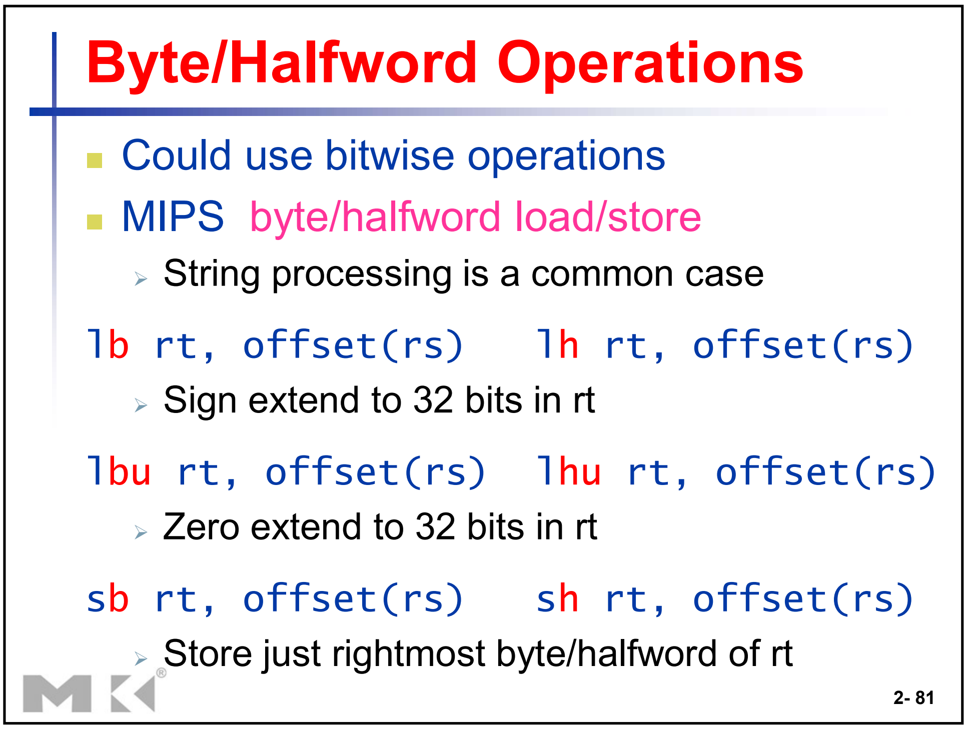
Character Data

- 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-32: 32-bit encoding
  - UTF-16: 16-bit encoding (default)
  - UTF-8: variable-length encodings, 8 ~ 32 bits

§2.9 Communicating with People

2- 80

80



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)
```

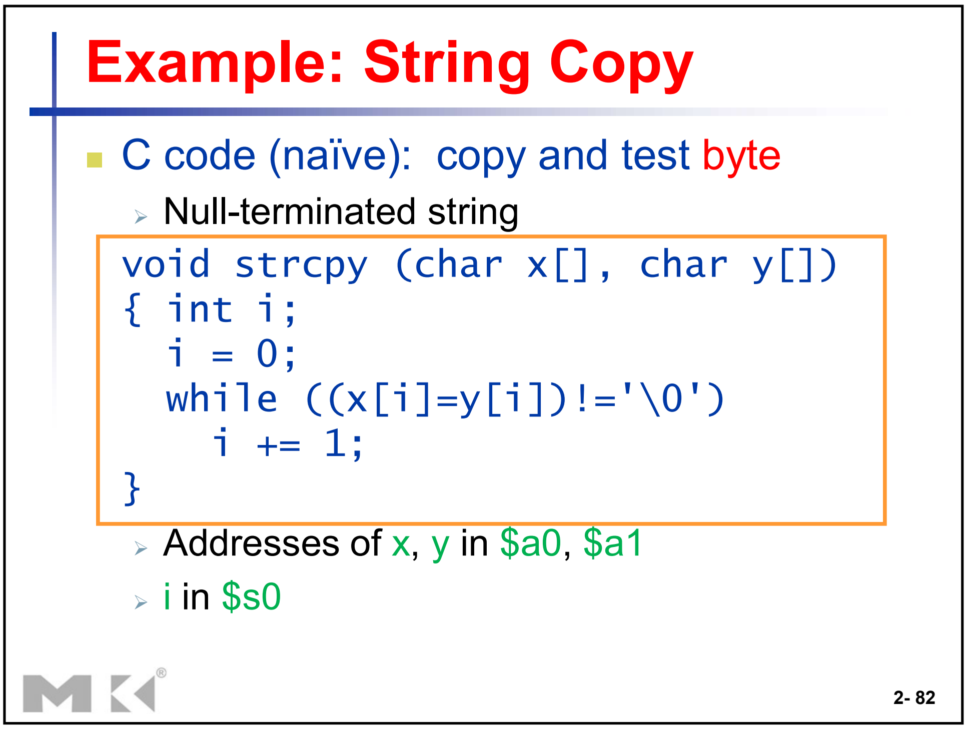
- Zero extend to 32 bits in rt

```
sb rt, offset(rs)    sh rt, offset(rs)
```

- Store just rightmost byte/halfword of rt

2- 81

81



Example: String Copy

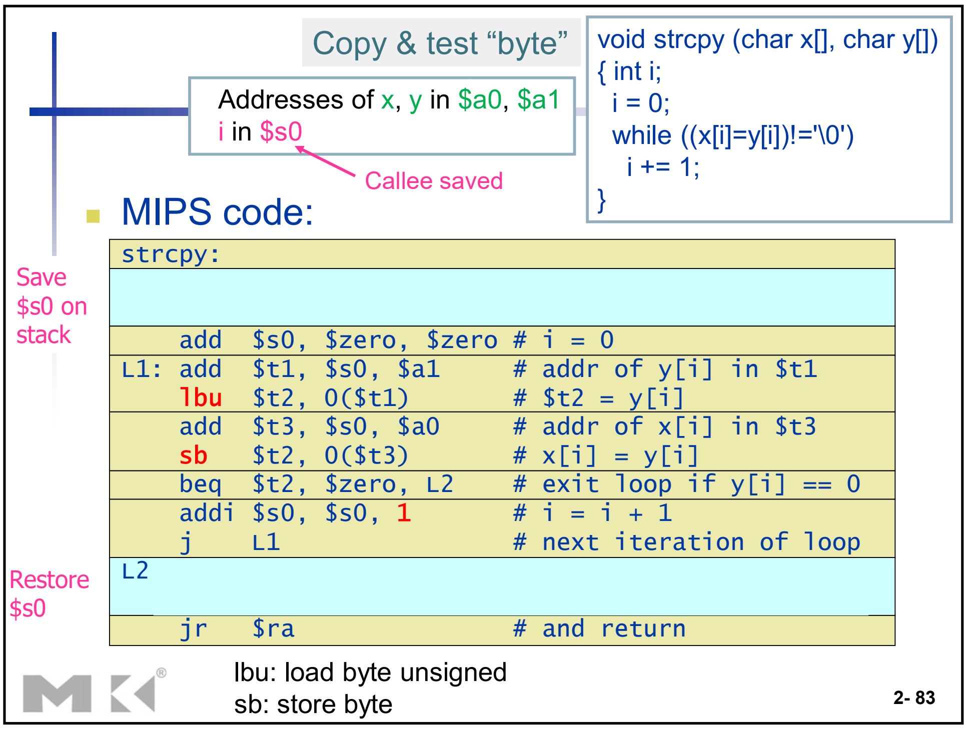
- C code (naïve): copy and test byte
  - Null-terminated string

```
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

2- 82

82



Copy & test "byte"

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

Callee saved

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

- MIPS code:

Save $s0 on stack

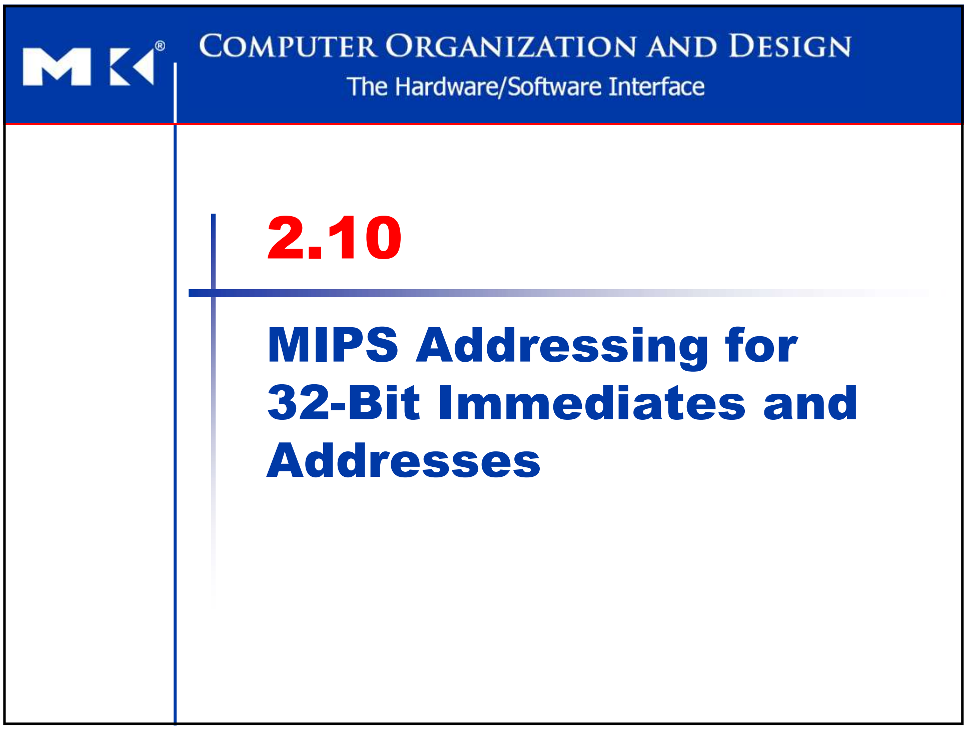
```
strcpy:

     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

     jr   $ra               # and return
```

Restore $s0

lbu: load byte unsigned
sb: store byte

2- 83

83

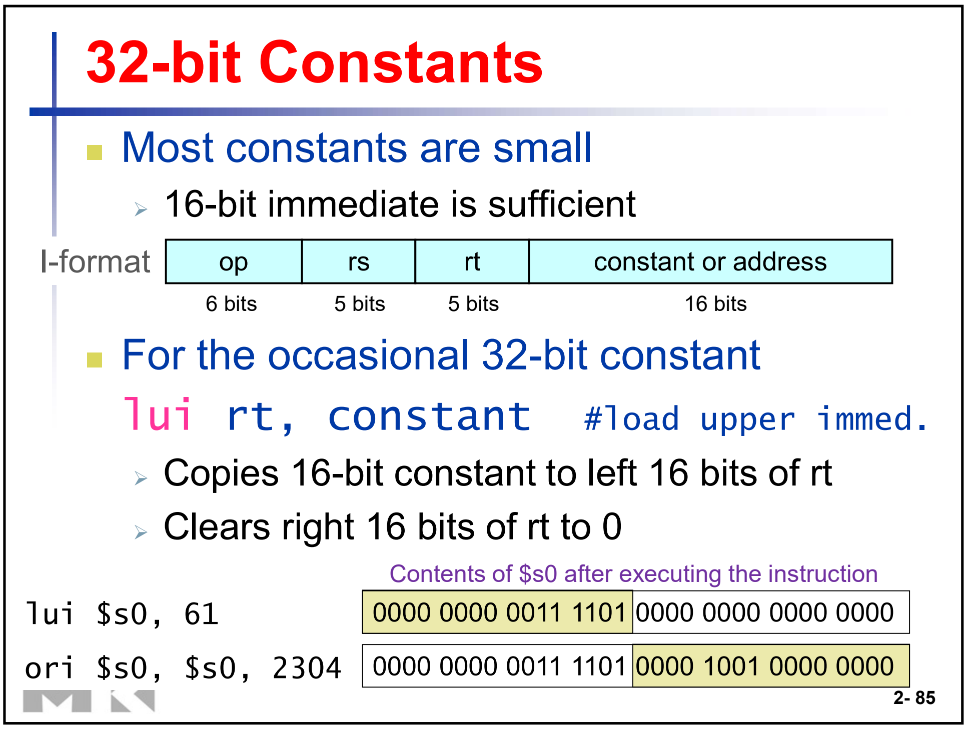


COMPUTER ORGANIZATION AND DESIGN
The Hardware/Software Interface

2.10

MIPS Addressing for 32-Bit Immediates and Addresses

84



32-bit Constants

- Most constants are small
  - 16-bit immediate is sufficient

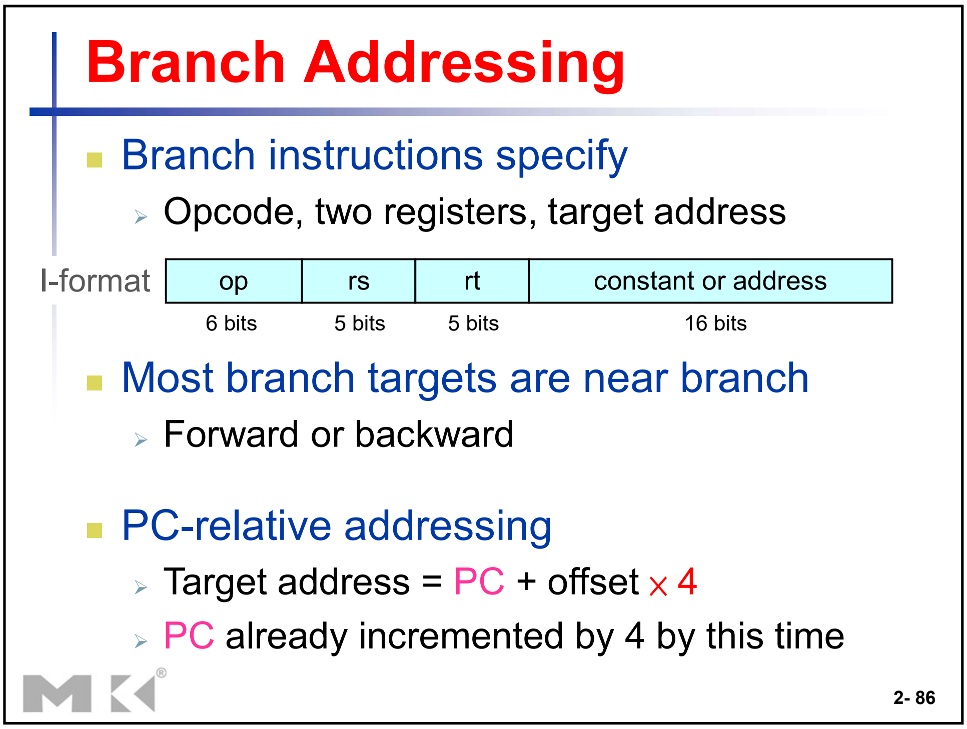
I-format	op	rs	rt	constant or address
	6 bits	5 bits	5 bits	16 bits

- For the occasional 32-bit constant

```
lui rt, constant   #load upper immed.
```

  - Copies 16-bit constant to left 16 bits of rt
  - Clears right 16 bits of rt to 0

	Contents of $s0 after executing the instruction	
`lui $s0, 61`	0000 0000 0011 1101	0000 0000 0000 0000
`ori $s0, $s0, 2304`	0000 0000 0011 1101	0000 1001 0000 0000

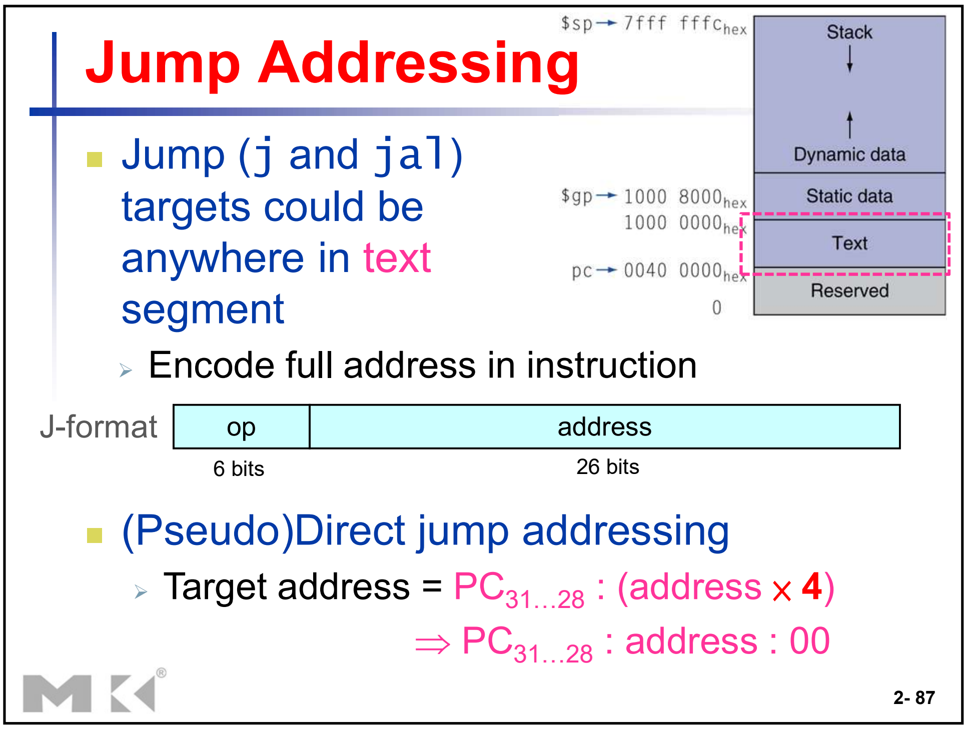


Branch Addressing

- Branch instructions specify
  - Opcode, two registers, target address

I-format	op	rs	rt	constant or address
	6 bits	5 bits	5 bits	16 bits

- Most branch targets are near branch
  - Forward or backward
- PC-relative addressing
  - Target address = PC + 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

$sp → 7fff fffc$_{hex}$

Stack

Dynamic data

$gp → 1000 8000$_{hex}$

Static data

1000 0000$_{hex}$

Text

pc → 0040 0000$_{hex}$

Reserved

0

J-format

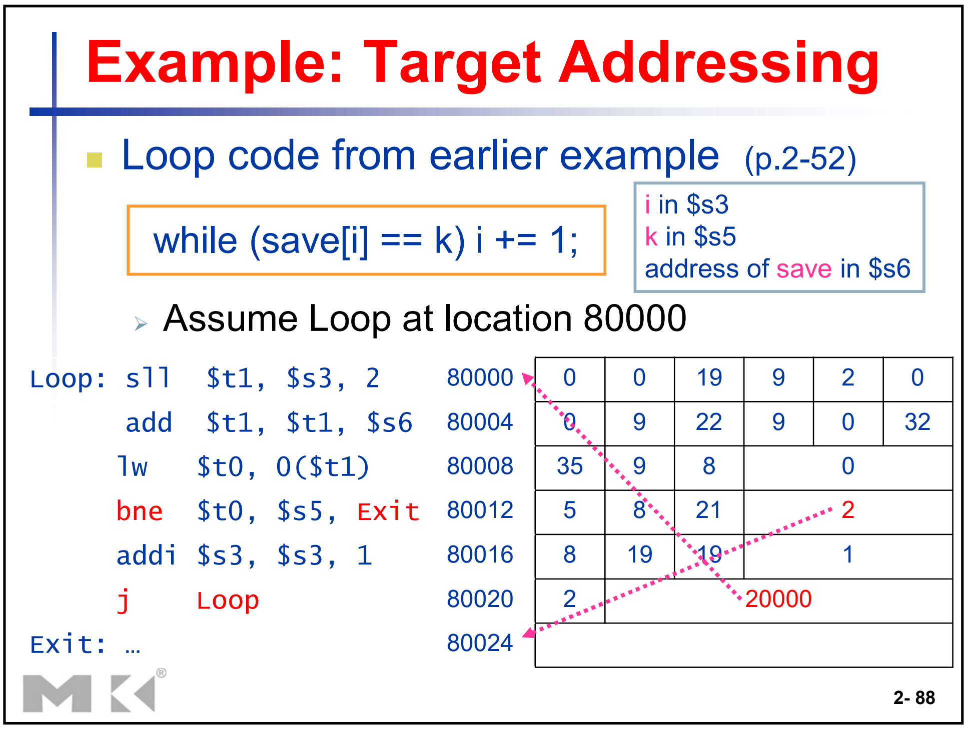
op	address
6 bits	26 bits

- (Pseudo)Direct jump addressing
  - Target address = $PC_{31\ldots28}$: (address × **4**)

    ⇒ $PC_{31\ldots28}$: address : 00

2- 87

87



Example: Target Addressing

- Loop code from earlier example (p.2-52)

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

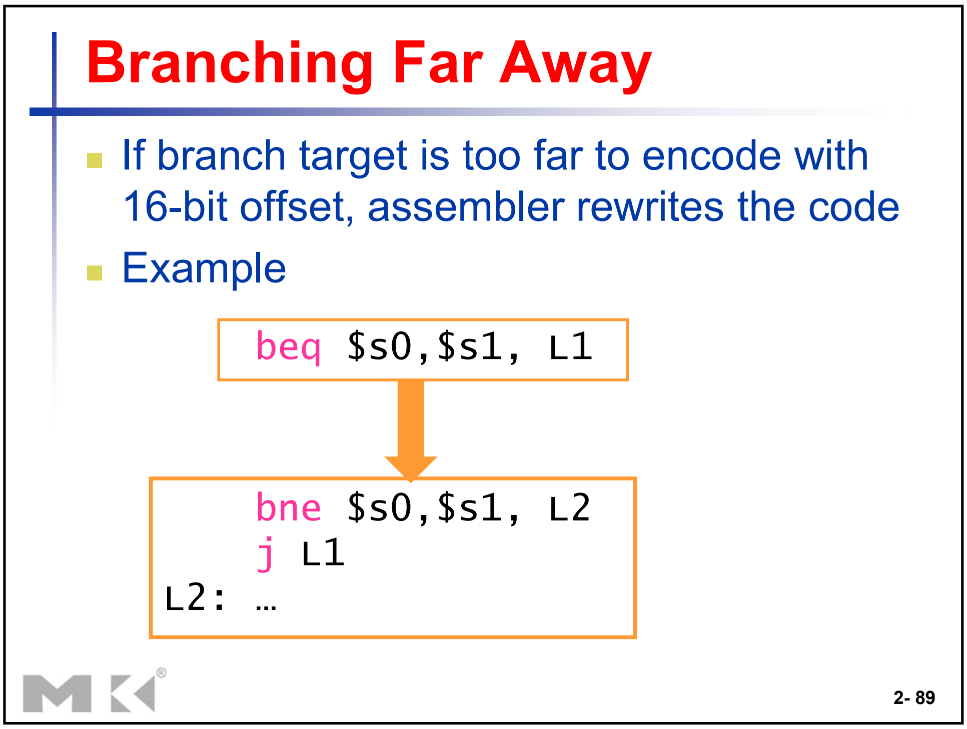
i in $s3
k in $s5
address of save in $s6

  - Assume Loop at location 80000

`Loop: sll $t1, $s3, 2`	80000	0	0	19	9	2	0
`add $t1, $t1, $s6`	80004	0	9	22	9	0	32
`lw $t0, 0($t1)`	80008	35	9	8	0		
`bne $t0, $s5, Exit`	80012	5	8	21	2		
`addi $s3, $s3, 1`	80016	8	19	19	1		
`j Loop`	80020	2	20000				
`Exit: …`	80024						

2- 88

88



Branching Far Away

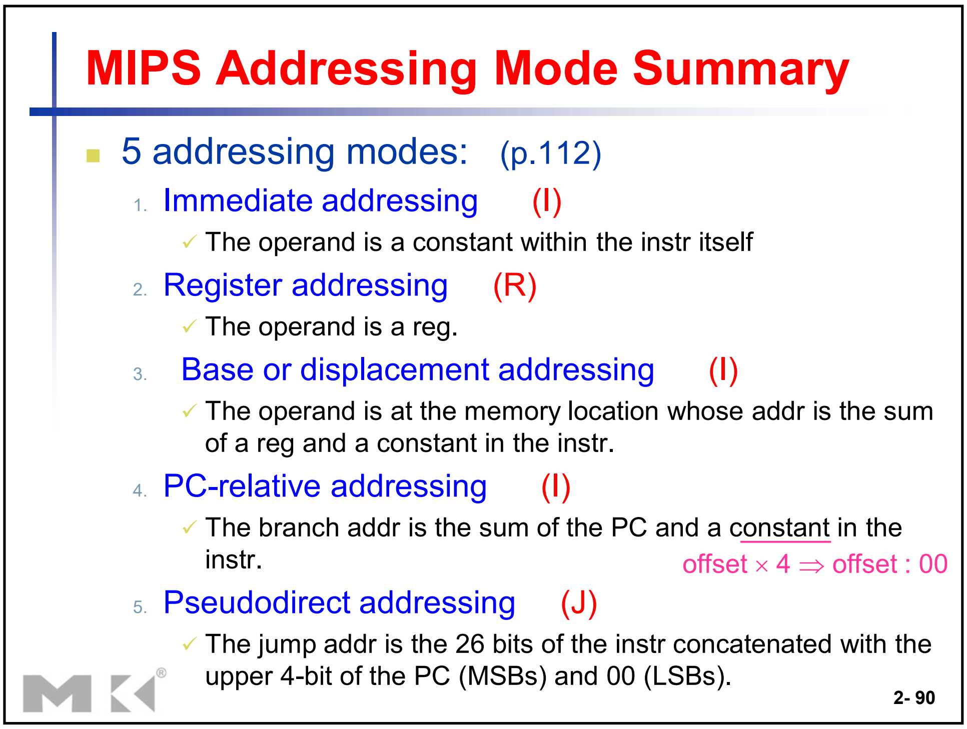
- If branch target is too far to encode with 16-bit offset, assembler rewrites the code
- Example

```
    beq $s0,$s1, L1
```

```
    bne $s0,$s1, L2
    j L1
L2: …
```

2- 89

89



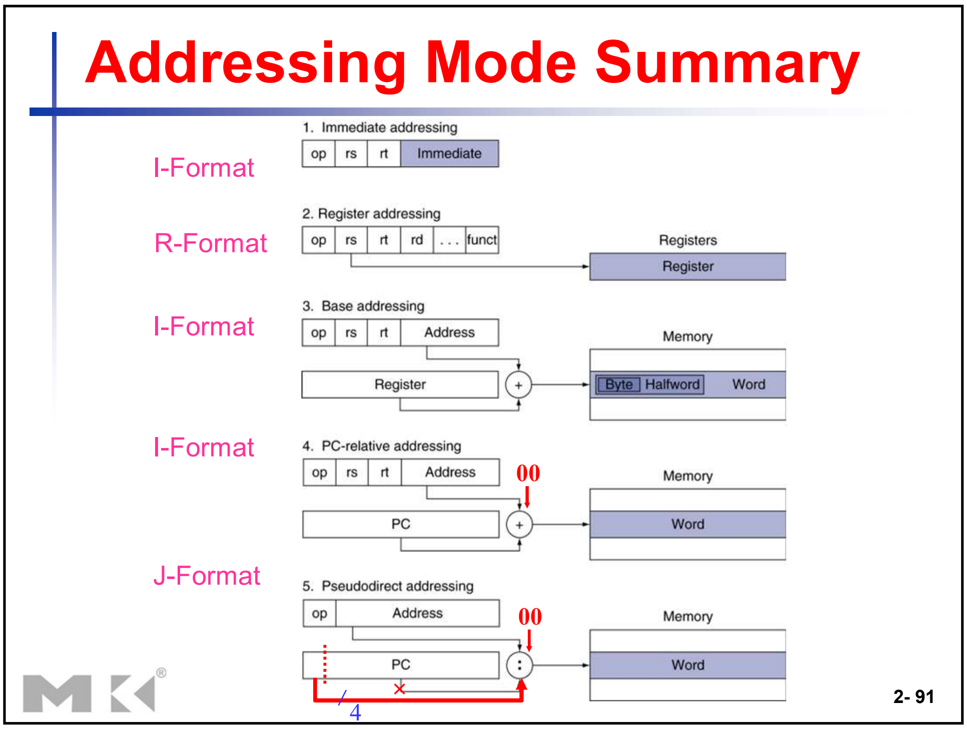
MIPS Addressing Mode Summary

- 5 addressing modes: (p.112)
  1. Immediate addressing (I)
     - ✓ The operand is a constant within the instr itself
  2. Register addressing (R)
     - ✓ The operand is a reg.
  3. Base or displacement addressing (I)
     - ✓ The operand is at the memory location whose addr is the sum of a reg and a constant in the instr.
  4. PC-relative addressing (I)
     - ✓ The branch addr is the sum of the PC and a constant in the instr.

       offset × 4 ⇒ offset : 00
  5. Pseudodirect addressing (J)
     - ✓ The jump addr is the 26 bits of the instr concatenated with the upper 4-bit of the PC (MSBs) and 00 (LSBs).

2- 90

90



Addressing Mode Summary

I-Format

1. Immediate addressing

R-Format

2. Register addressing

I-Format

3. Base addressing

I-Format

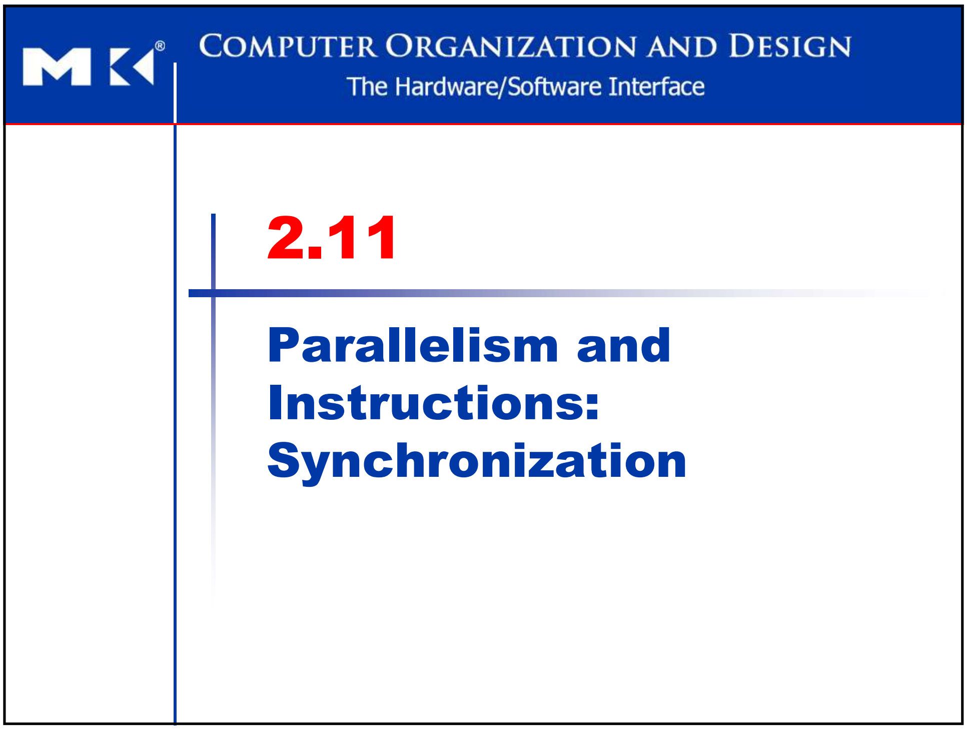
4. PC-relative addressing

J-Format

5. Pseudodirect addressing

2- 91

91



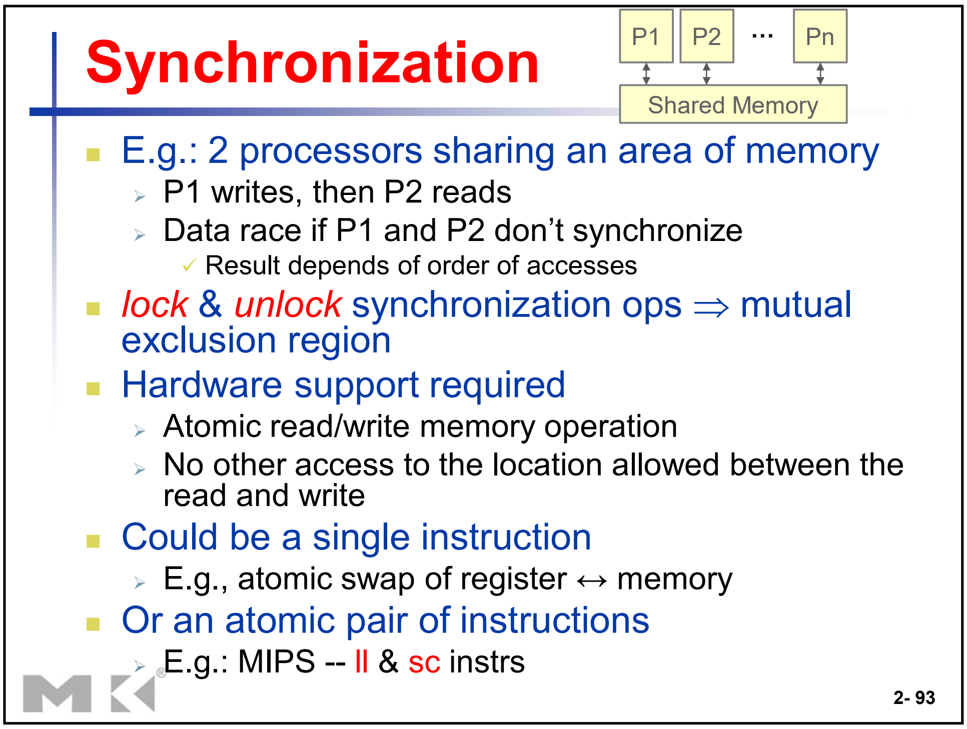
COMPUTER ORGANIZATION AND DESIGN

The Hardware/Software Interface

2.11

Parallelism and Instructions: Synchronization

92



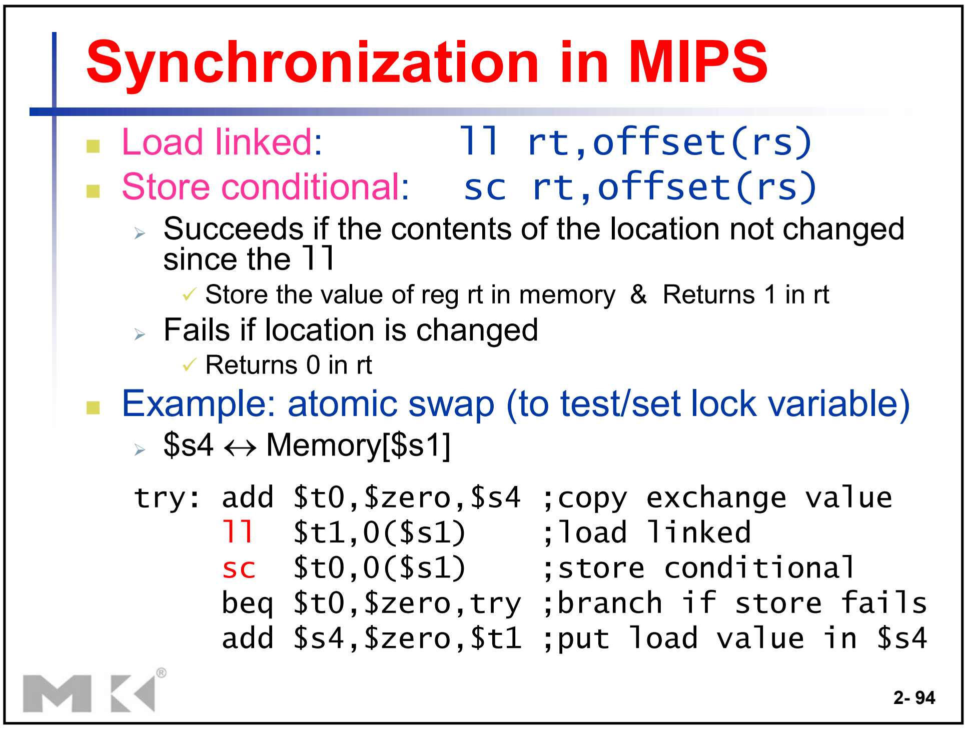
Synchronization

P1
P2
···
Pn
Shared Memory

- E.g.: 2 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
- *lock* & *unlock* synchronization ops ⇒ mutual exclusion region
- 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 ↔ memory
- Or an atomic pair of instructions
  - E.g.: MIPS -- ll & sc instrs

2- 93

93



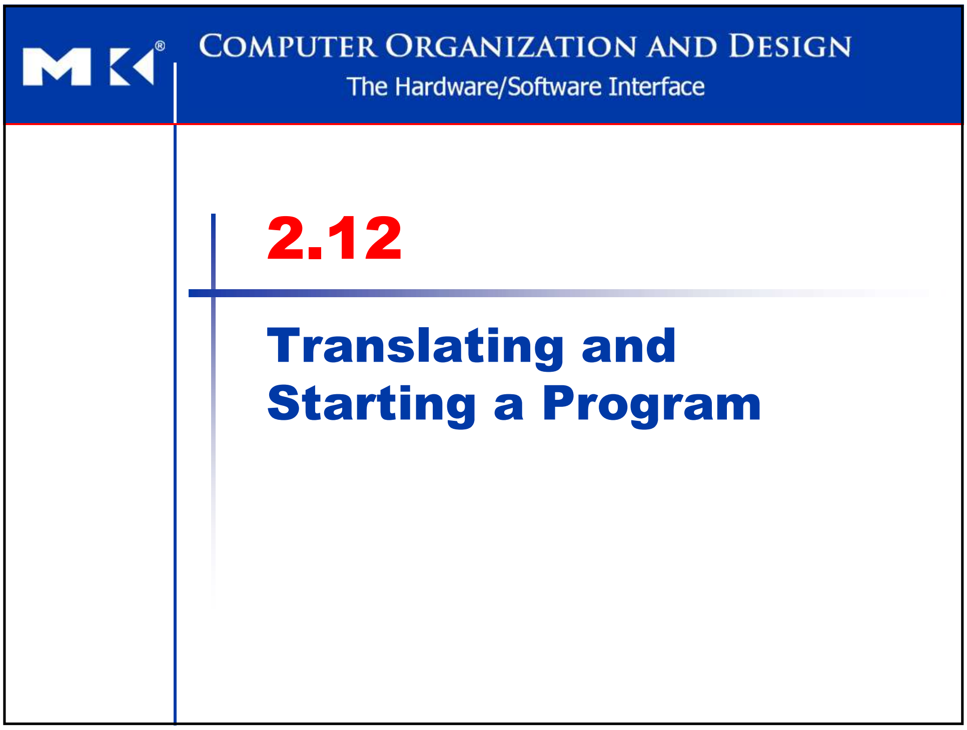
Synchronization in MIPS

- Load linked: `ll rt,offset(rs)`
- Store conditional: `sc rt,offset(rs)`
  - Succeeds if the contents of the location not changed since the `ll`
    - ✓ Store the value of reg rt in memory & Returns 1 in rt
  - Fails if location is changed
    - ✓ Returns 0 in rt
- Example: atomic swap (to test/set lock variable)
  - $s4 ↔ Memory[$s1]

```
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 if store fails
     add $s4,$zero,$t1 ;put load value in $s4
```

2- 94

94



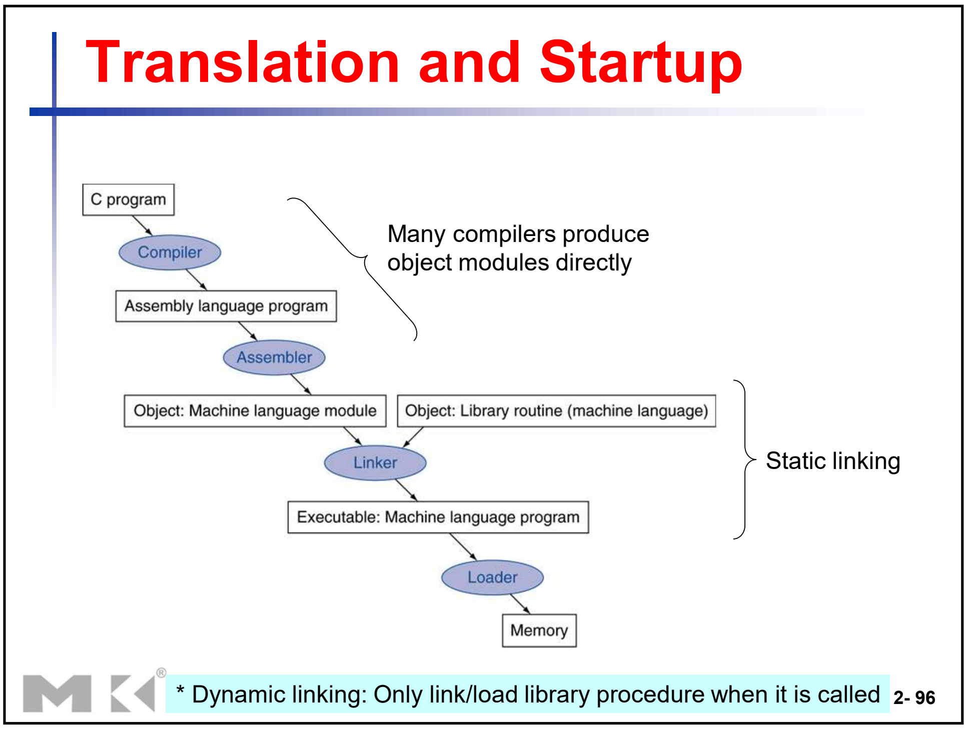
COMPUTER ORGANIZATION AND DESIGN

The Hardware/Software Interface

2.12

Translating and Starting a Program

95



Translation and Startup

C program

Compiler

Assembly language program

Assembler

Object: Machine language module

Object: Library routine (machine language)

Linker

Executable: Machine language program

Loader

Memory

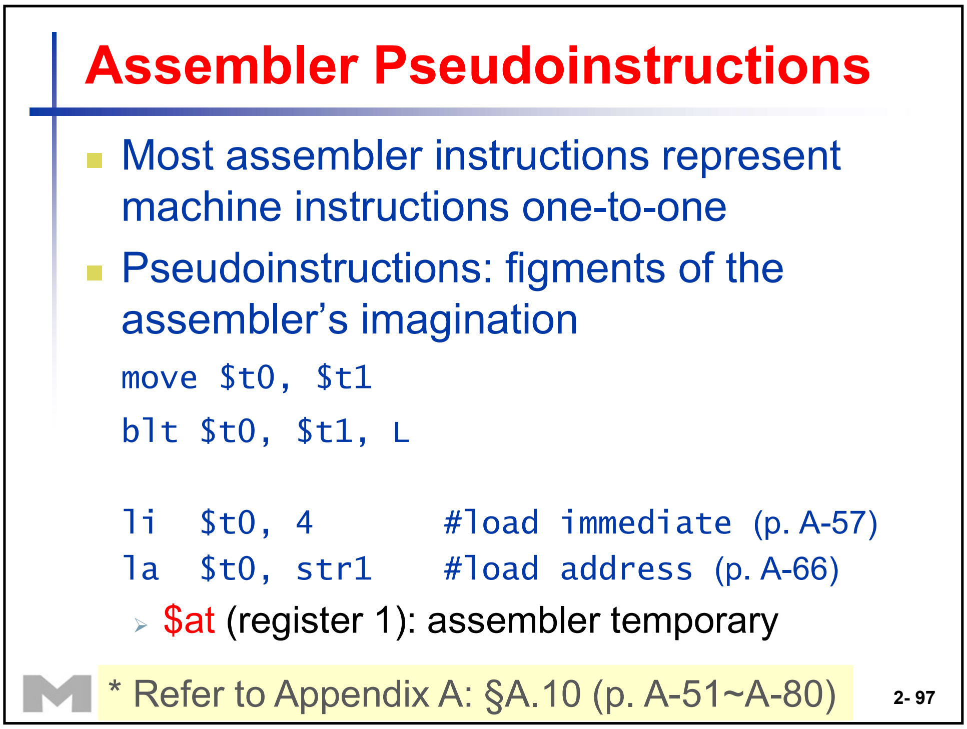
Many compilers produce object modules directly

Static linking

* Dynamic linking: Only link/load library procedure when it is called

2- 96

96



Assembler Pseudoinstructions

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

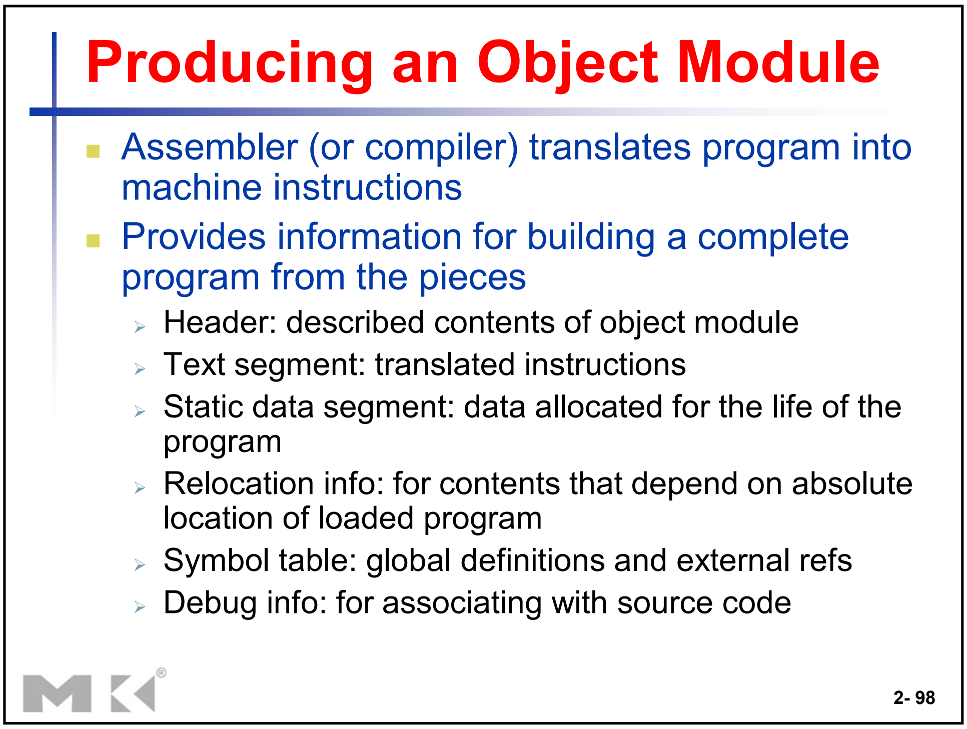
```
move $t0, $t1
blt $t0, $t1, L

li  $t0, 4       #load immediate (p. A-57)
la  $t0, str1    #load address (p. A-66)
```

  - $at (register 1): assembler temporary

* Refer to Appendix A: §A.10 (p. A-51~A-80)

2- 97

97

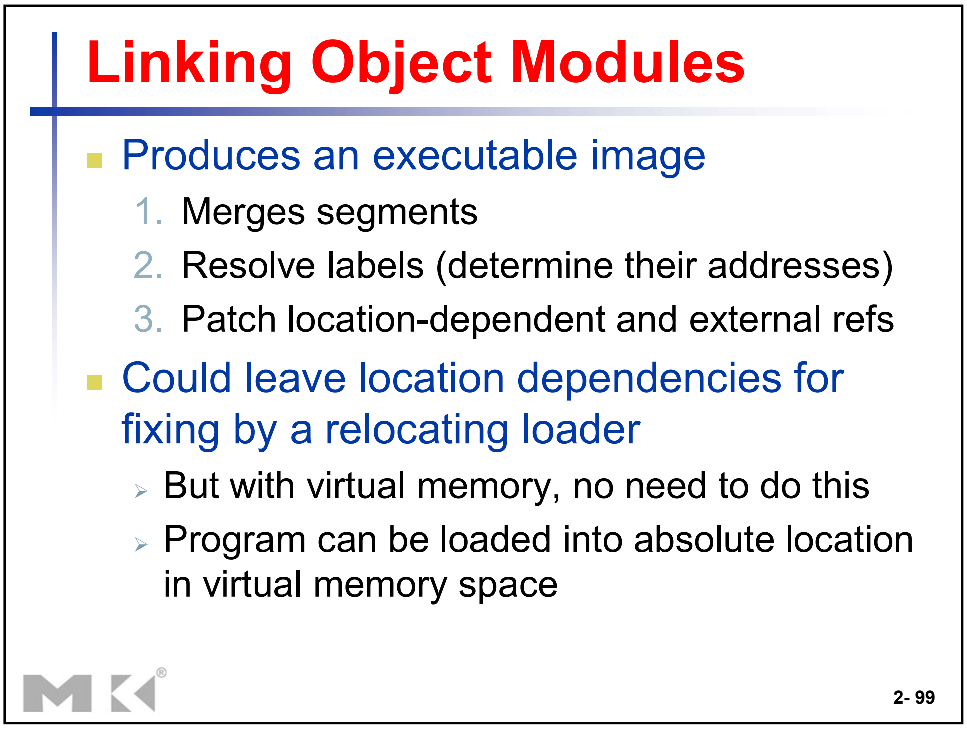


Producing an Object Module

- 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

2- 98

98

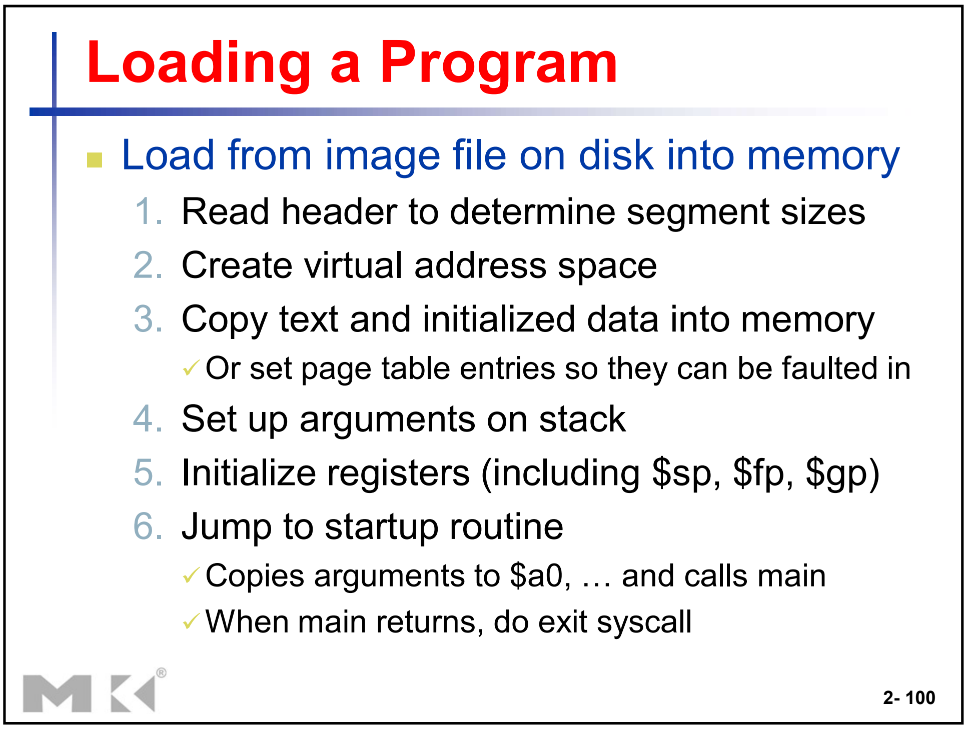


Linking Object Modules

- Produces an executable image
  1. Merges segments
  2. Resolve labels (determine their addresses)
  3. 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

2- 99

99

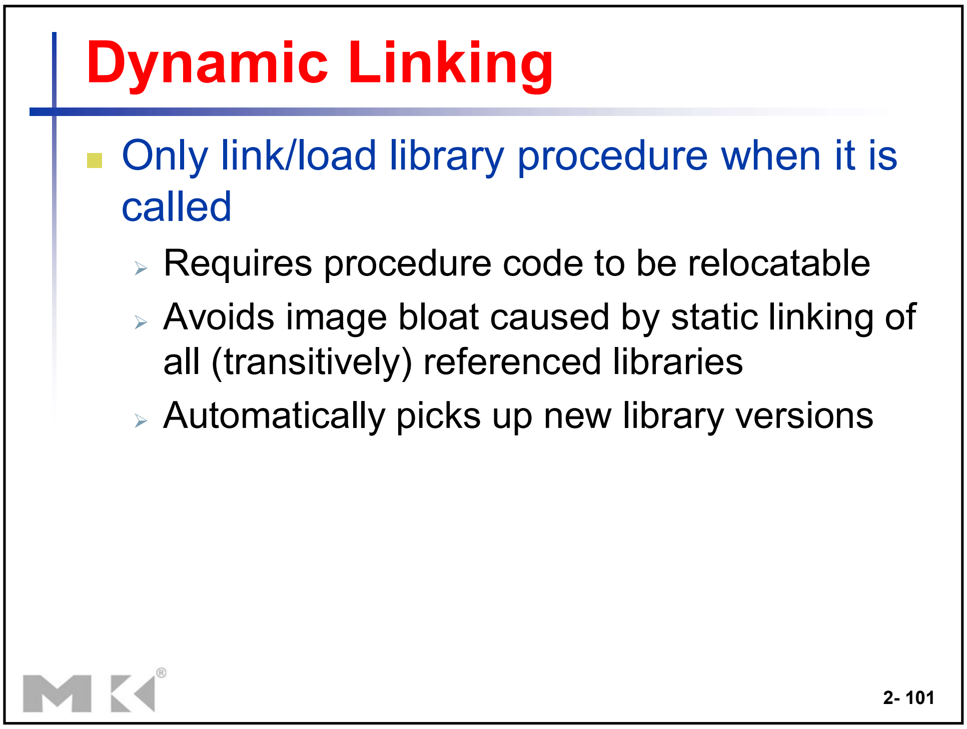


Loading a Program

- 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

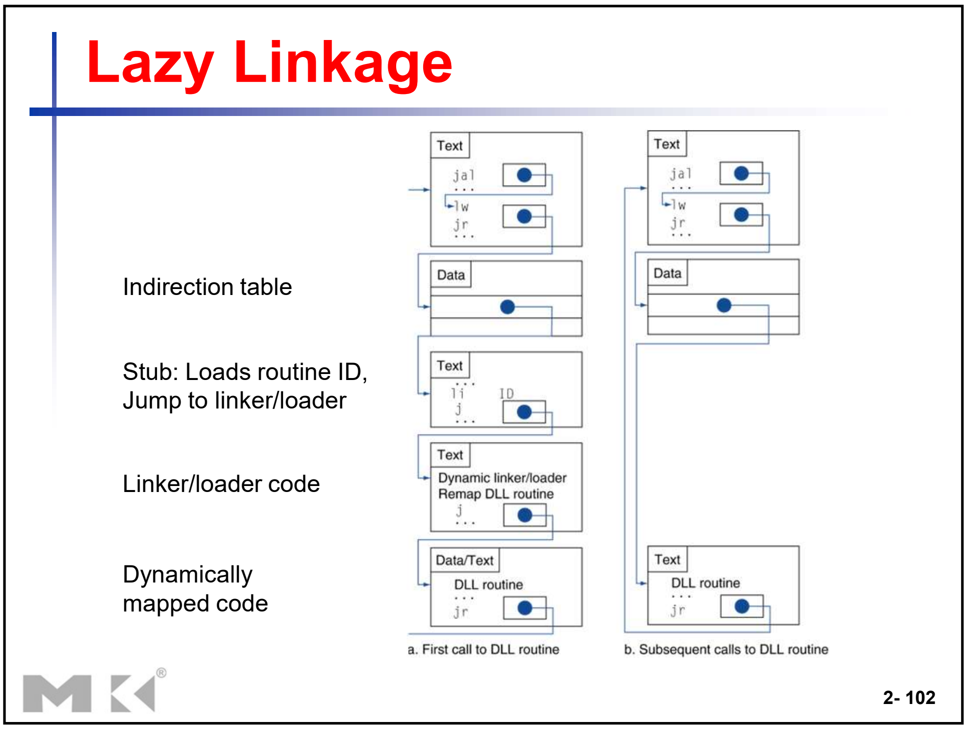
2- 100

100



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

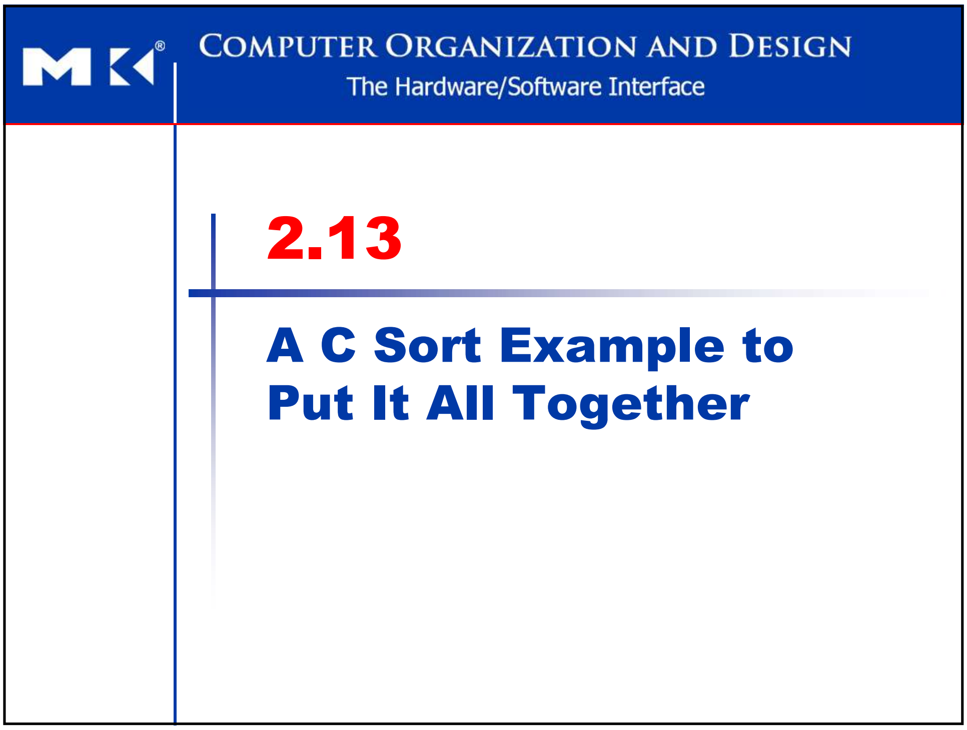
Stub: Loads routine ID, Jump to linker/loader

Linker/loader code

Dynamically mapped code

a. First call to DLL routine

b. Subsequent calls to DLL routine



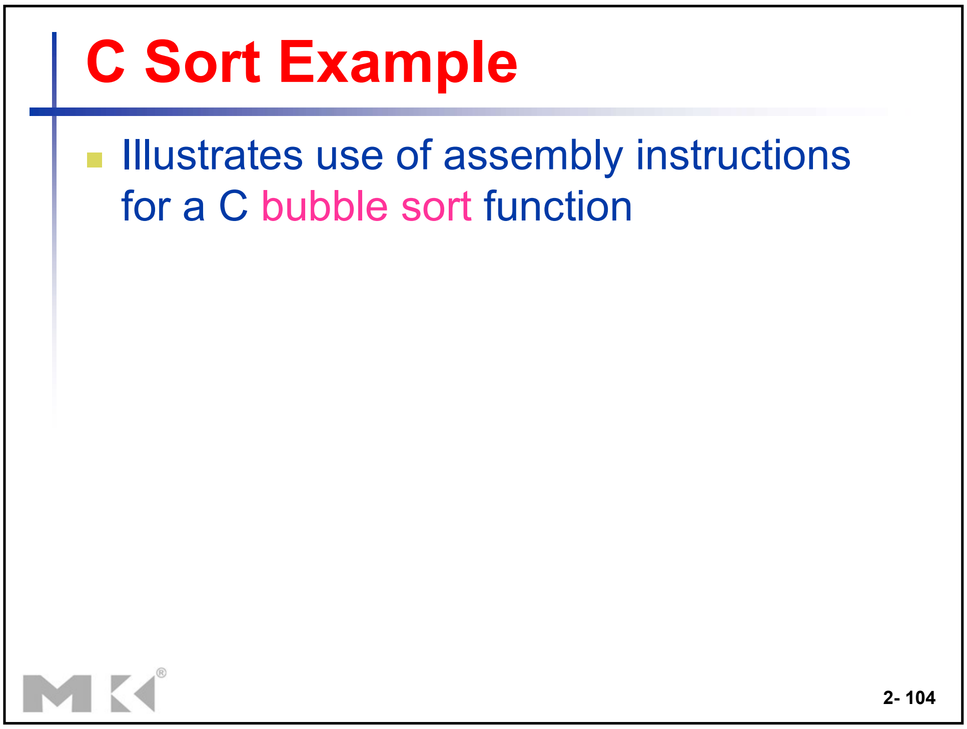
Computer Organization and Design

The Hardware/Software Interface

2.13

A C Sort Example to Put It All Together

103

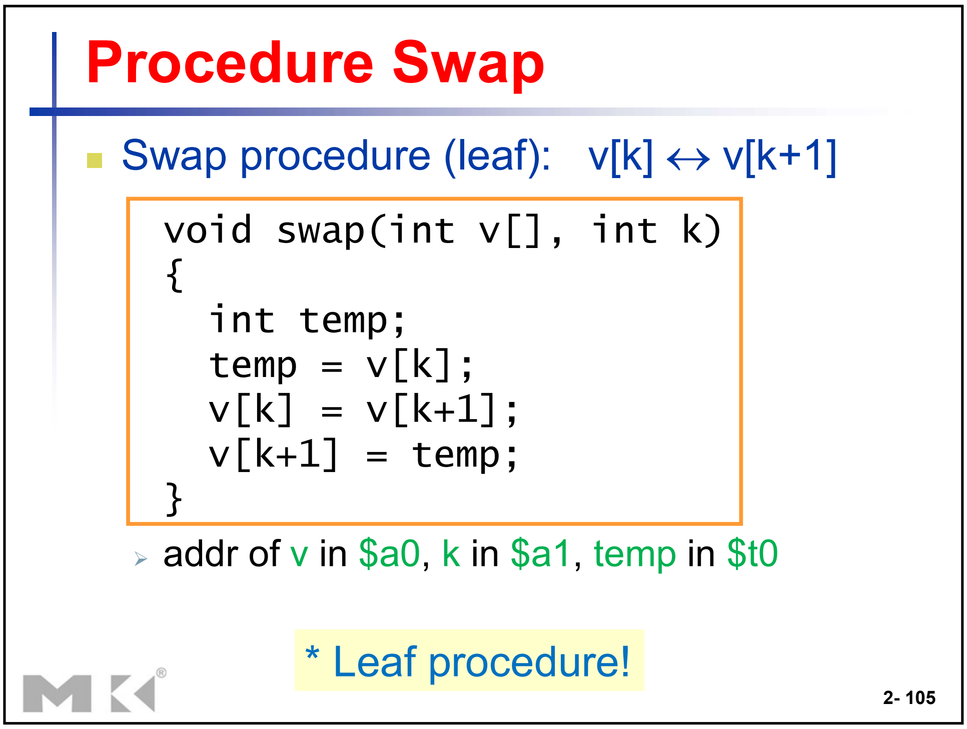


C Sort Example

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

2- 104

104



Procedure Swap

- Swap procedure (leaf): v[k] ↔ v[k+1]

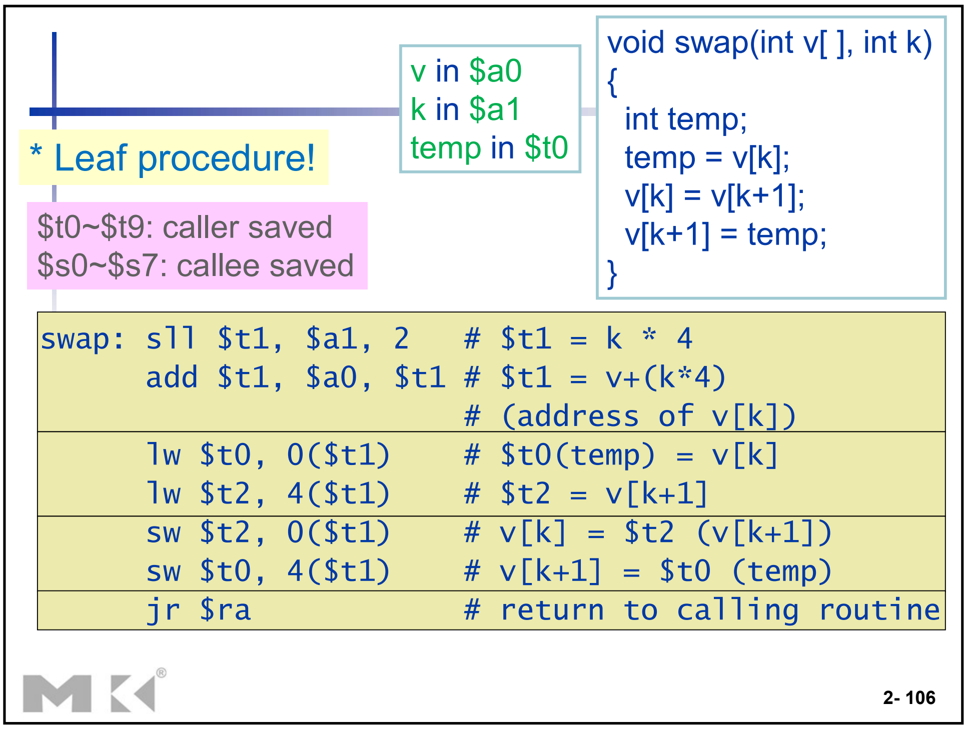
```
void swap(int v[], int k)
{
  int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

  - addr of v in $a0, k in $a1, temp in $t0

* Leaf procedure!

2- 105

105



* Leaf procedure!

$t0~$t9: caller saved
$s0~$s7: callee saved

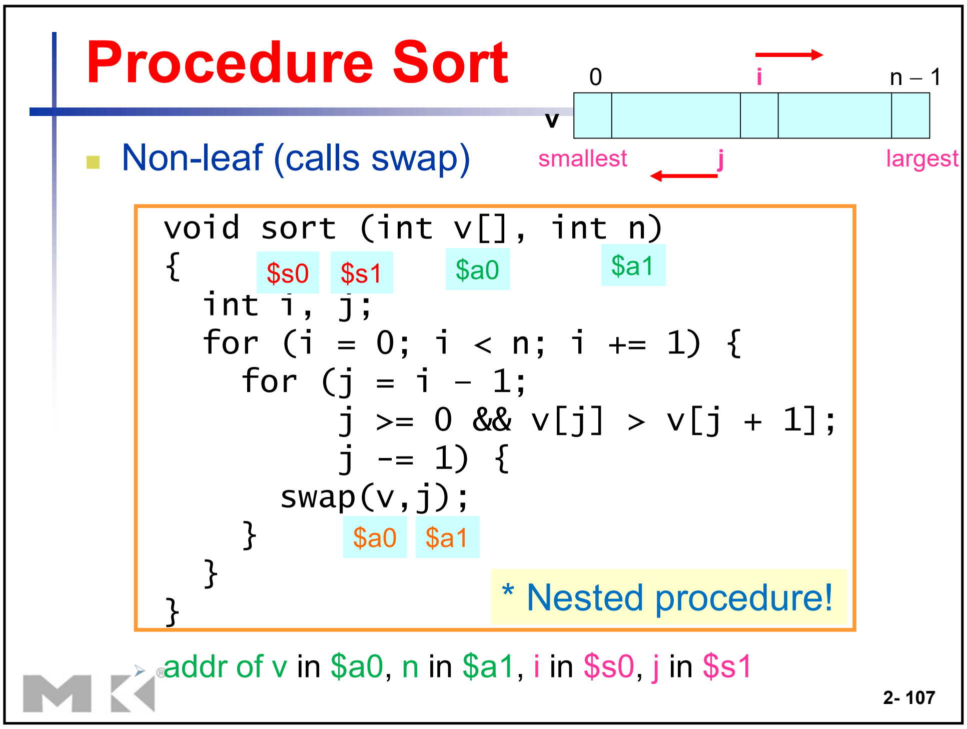
v in $a0
k in $a1
temp in $t0

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

```
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
```

2- 106

106



Procedure Sort

0
i
n – 1
v
smallest
j
largest

- Non-leaf (calls swap)

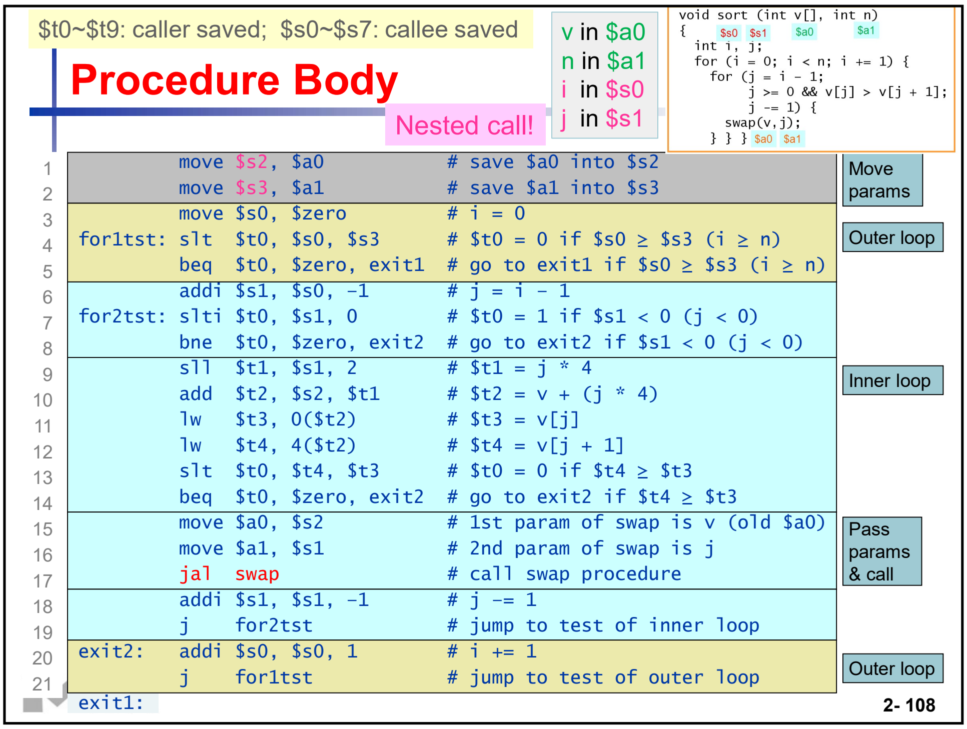
```
void sort (int v[], int n)
{
  int i, j;
  for (i = 0; i < n; i += 1) {
    for (j = i – 1;
         j >= 0 && v[j] > v[j + 1];
         j -= 1) {
      swap(v,j);
    }
  }
}
```

$s0 $s1 $a0 $a1

$a0 $a1

* Nested procedure!

addr of v in $a0, n in $a1, i in $s0, j in $s1



$t0~$t9: caller saved; $s0~$s7: callee saved

Procedure Body

Nested call!

v in $a0
n in $a1
i in $s0
j in $s1

```
void sort (int v[], int n)
{   $s0 $s1  $a0  $a1
  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);
  } } } $a0 $a1
```

```
         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
exit1:
```

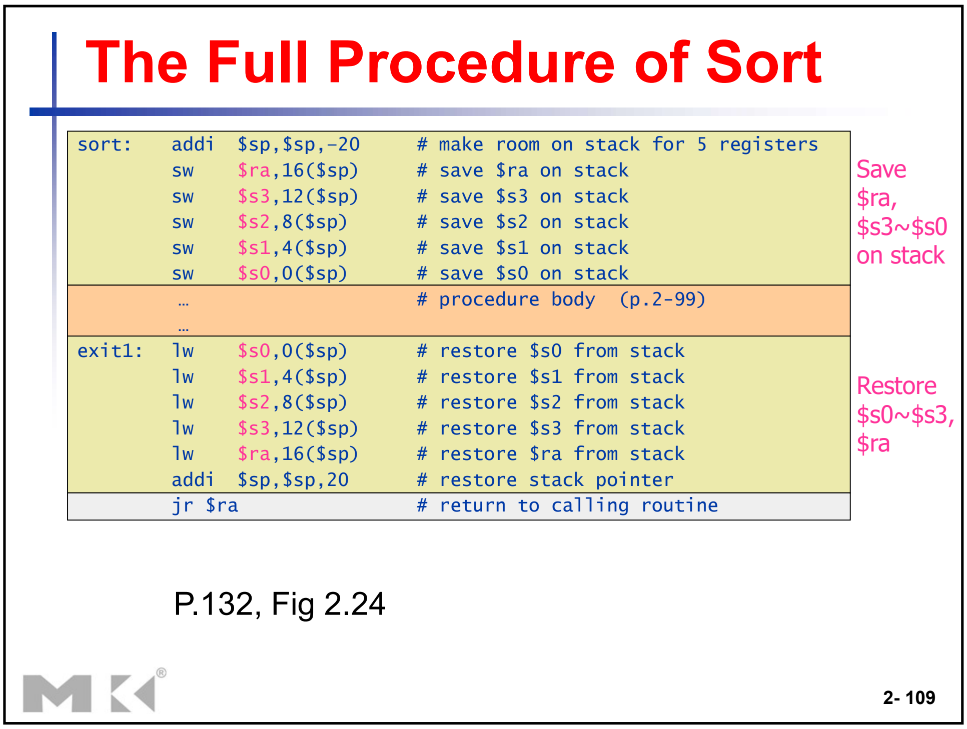
Move params

Outer loop

Inner loop

Pass params & call

Outer loop



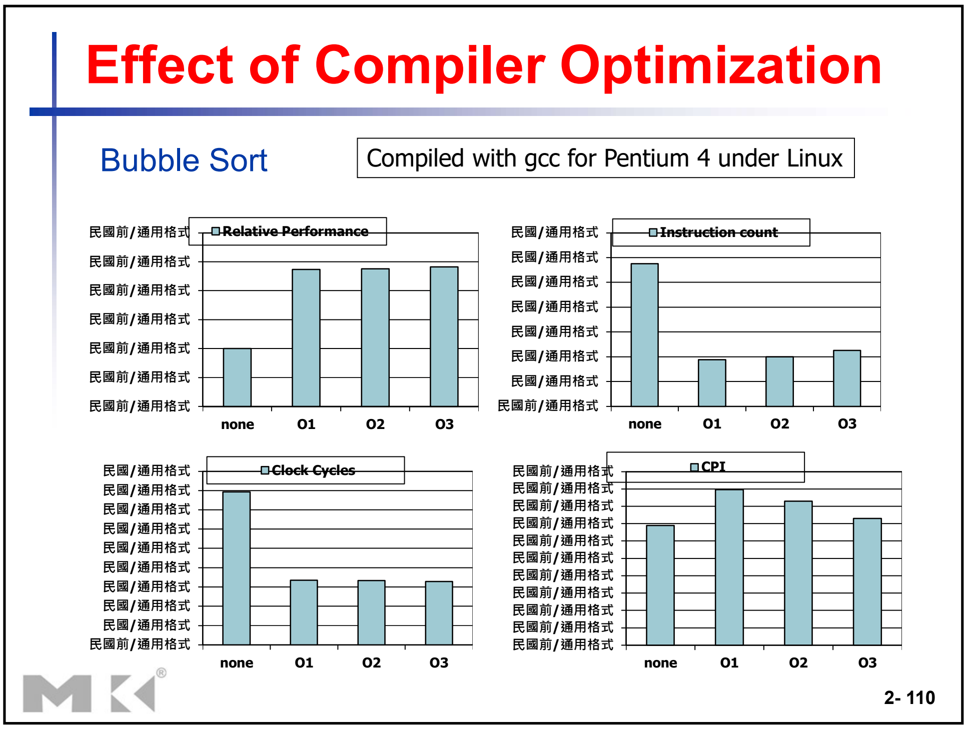
The Full Procedure of Sort

```
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  (p.2-99)
        …
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
```

Save $ra, $s3~$s0 on stack

Restore $s0~$s3, $ra

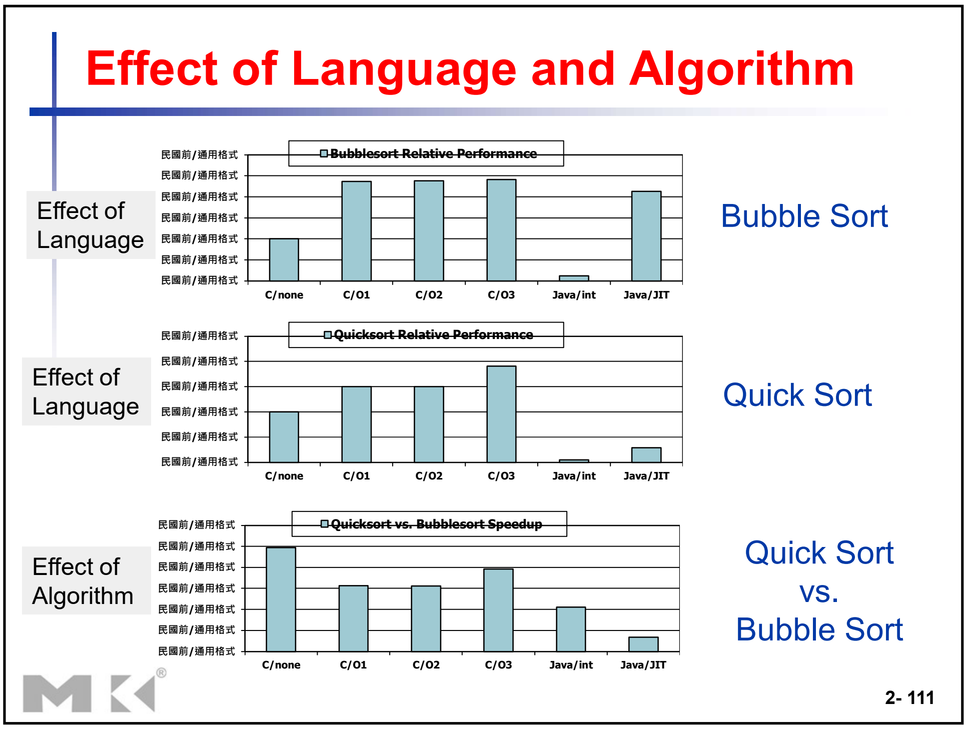
P.132, Fig 2.24



Effect of Compiler Optimization

Bubble Sort

Compiled with gcc for Pentium 4 under Linux



Effect of Language and Algorithm

Effect of Language

Bubble Sort

Effect of Language

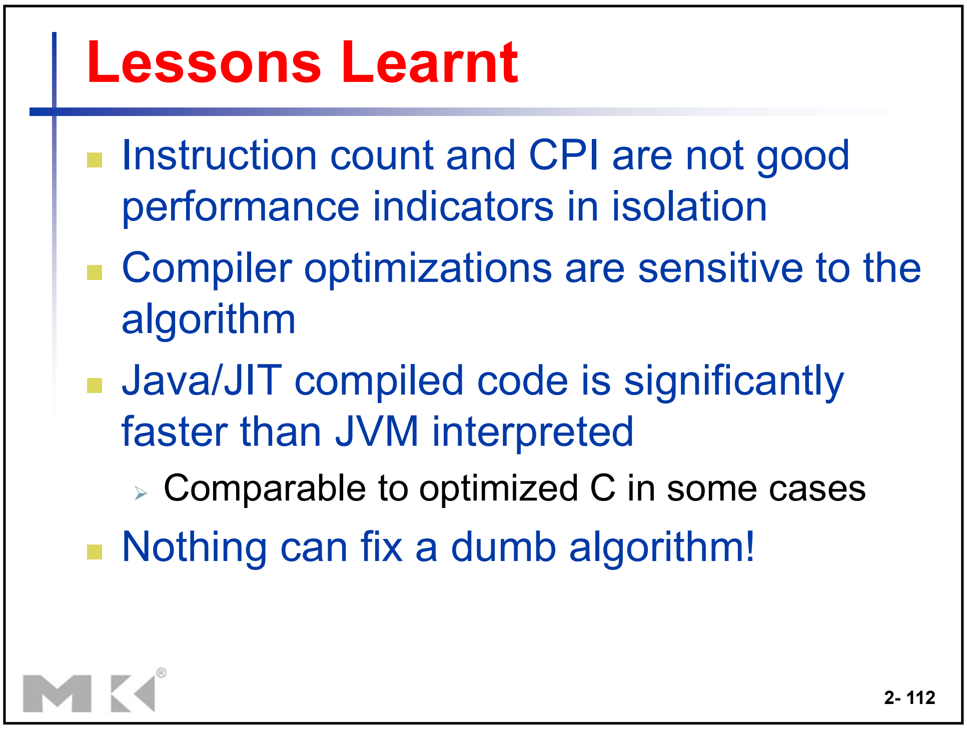
Quick Sort

Effect of Algorithm

Quick Sort
vs.
Bubble Sort

2- 111

111

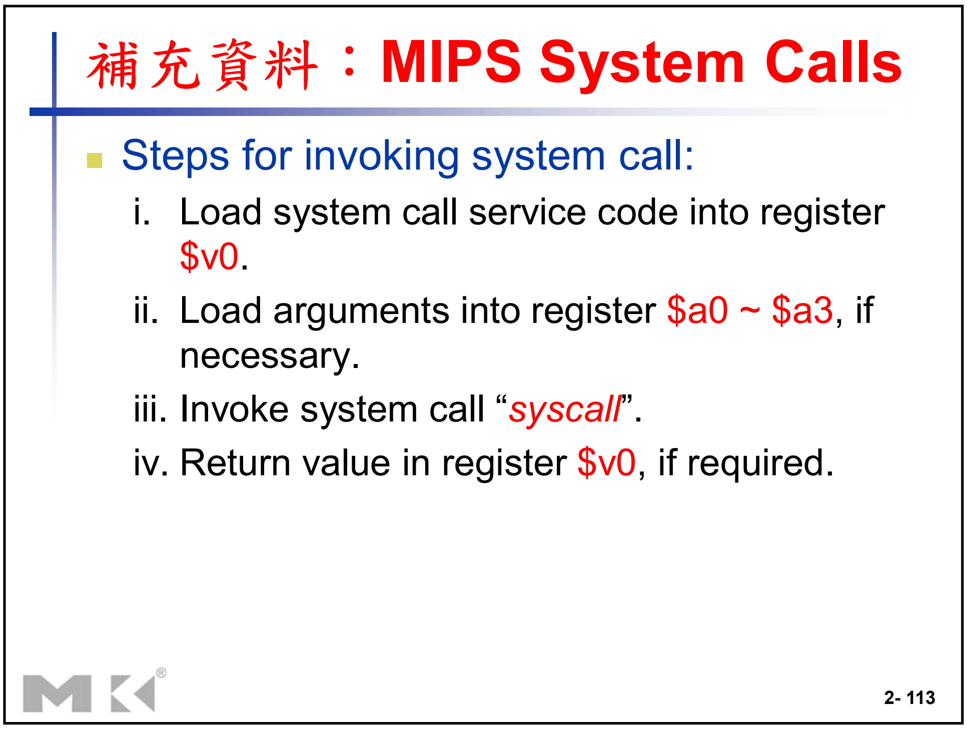


Lessons Learnt

- 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!

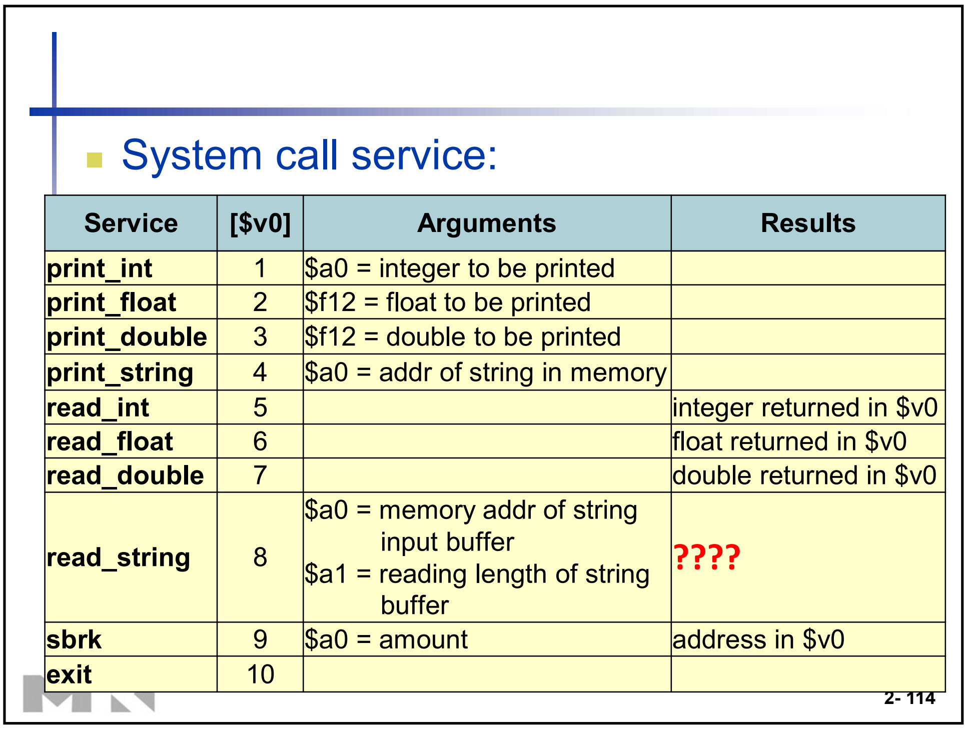
2- 112

112



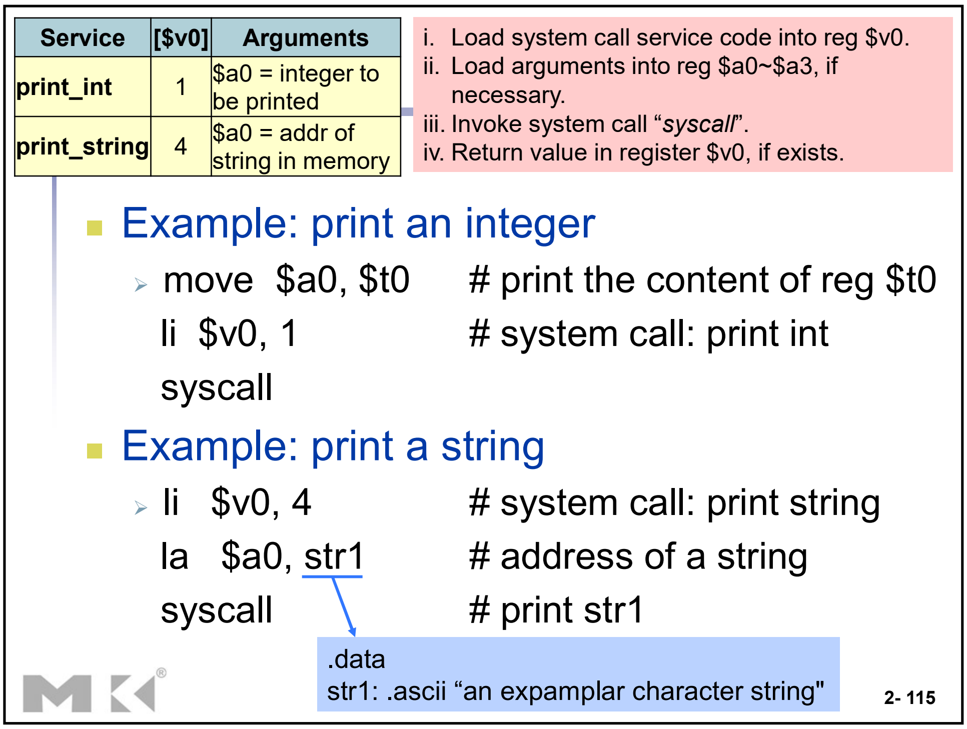
補充資料：MIPS System Calls

- Steps for invoking system call:
  1. Load system call service code into register $v0.
  2. Load arguments into register $a0 ~ $a3, if necessary.
  3. Invoke system call "*syscall*".
  4. Return value in register $v0, if required.



- System call service:

Service	[$v0]	Arguments	Results
print_int	1	$a0 = integer to be printed	
print_float	2	$f12 = float to be printed	
print_double	3	$f12 = double to be printed	
print_string	4	$a0 = addr of string in memory	
read_int	5		integer returned in $v0
read_float	6		float returned in $v0
read_double	7		double returned in $v0
read_string	8	$a0 = memory addr of string input buffer $a1 = reading length of string buffer	**????**
sbrk	9	$a0 = amount	address in $v0
exit	10		



Service	[$v0]	Arguments
print_int	1	$a0 = integer to be printed
print_string	4	$a0 = addr of string in memory

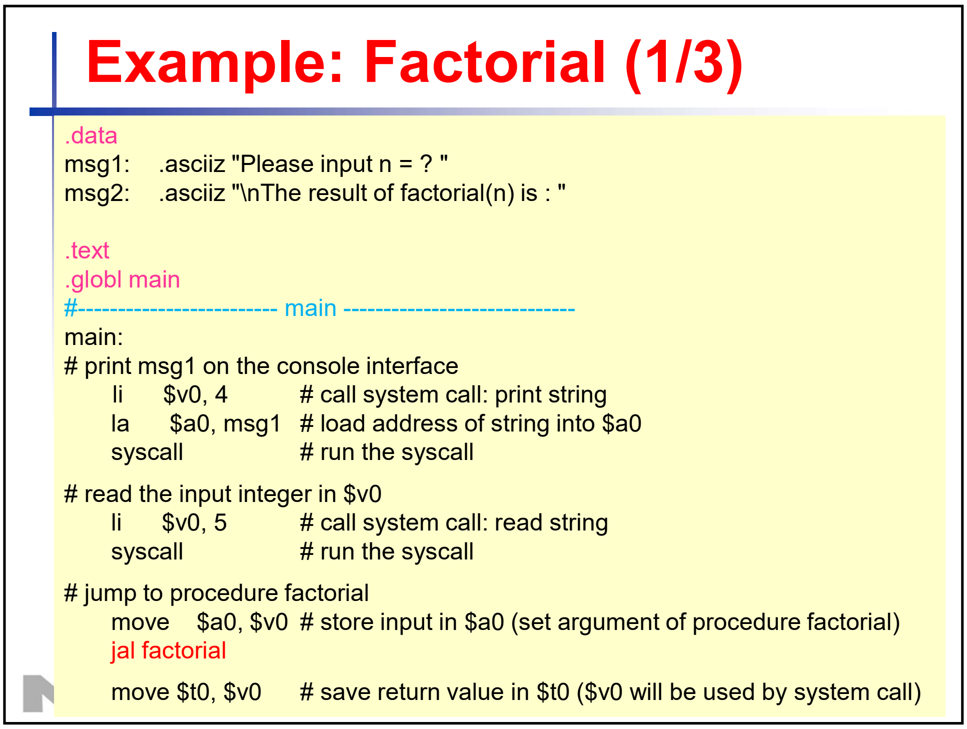
i. Load system call service code into reg $v0.
ii. Load arguments into reg $a0~$a3, if necessary.
iii. Invoke system call "*syscall*".
iv. Return value in register $v0, if exists.

- Example: print an integer
  - move $a0, $t0 # print the content of reg $t0
    li $v0, 1 # system call: print int
    syscall
- Example: print a string
  - li $v0, 4 # system call: print string
    la $a0, str1 # address of a string
    syscall # print str1

.data
str1: .ascii "an expamplar character string"

2- 115

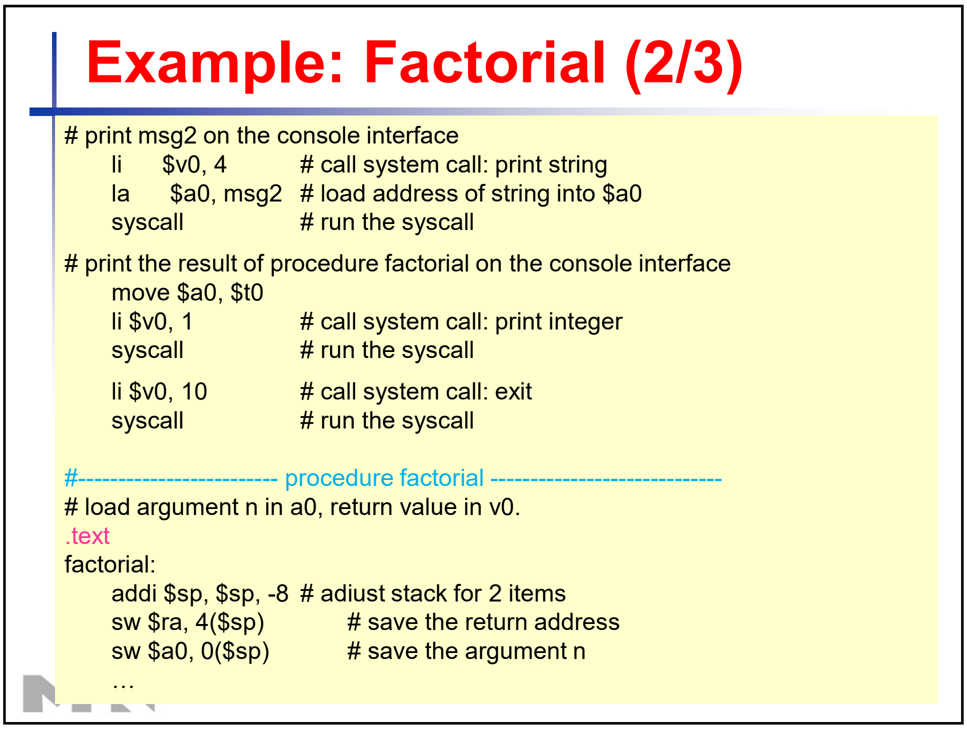
115



Example: Factorial (1/3)

```
.data
msg1:    .asciiz "Please input n = ? "
msg2:    .asciiz "\nThe result of factorial(n) is : "

.text
.globl main
#------------------------ main ----------------------------
main:
# print msg1 on the console interface
        li      $v0, 4        # call system call: print string
        la      $a0, msg1     # load address of string into $a0
        syscall               # run the syscall

# read the input integer in $v0
        li      $v0, 5        # call system call: read string
        syscall               # run the syscall

# jump to procedure factorial
        move    $a0, $v0      # store input in $a0 (set argument of procedure factorial)
        jal factorial
        move $t0, $v0         # save return value in $t0 ($v0 will be used by system call)
```

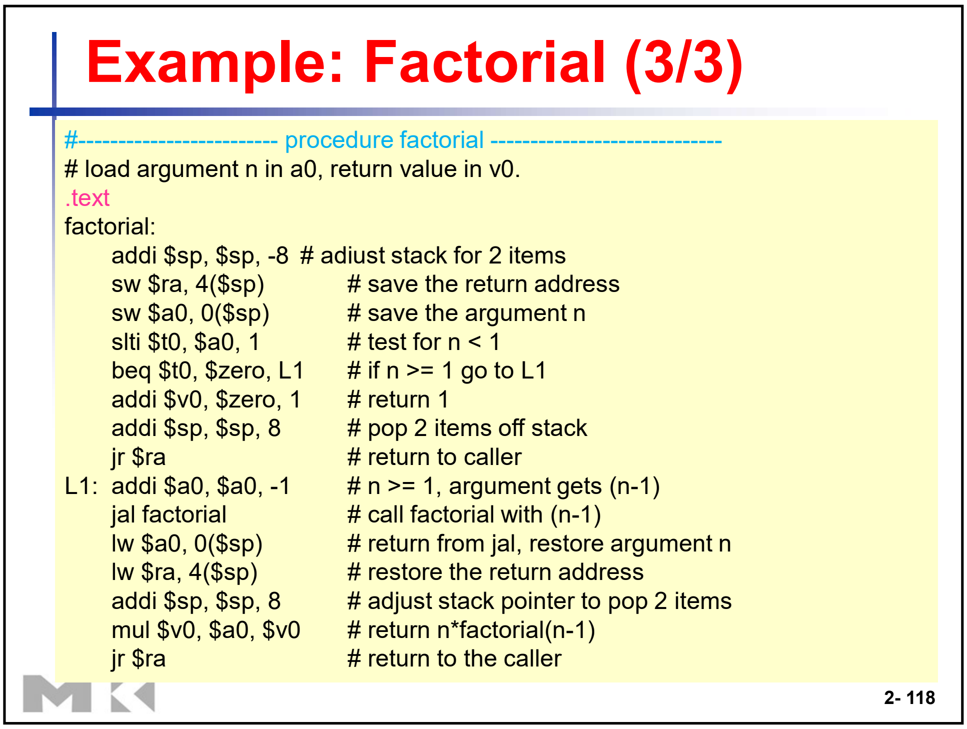
116



Example: Factorial (2/3)

```
# print msg2 on the console interface
        li      $v0, 4          # call system call: print string
        la      $a0, msg2       # load address of string into $a0
        syscall                 # run the syscall

# print the result of procedure factorial on the console interface
        move $a0, $t0
        li $v0, 1               # call system call: print integer
        syscall                 # run the syscall

        li $v0, 10              # call system call: exit
        syscall                 # run the syscall

#------------------------- procedure factorial -----------------------------
# load argument n in a0, return value in v0.
.text
factorial:
        addi $sp, $sp, -8  # adiust stack for 2 items
        sw $ra, 4($sp)          # save the return address
        sw $a0, 0($sp)          # save the argument n
        …
```

117

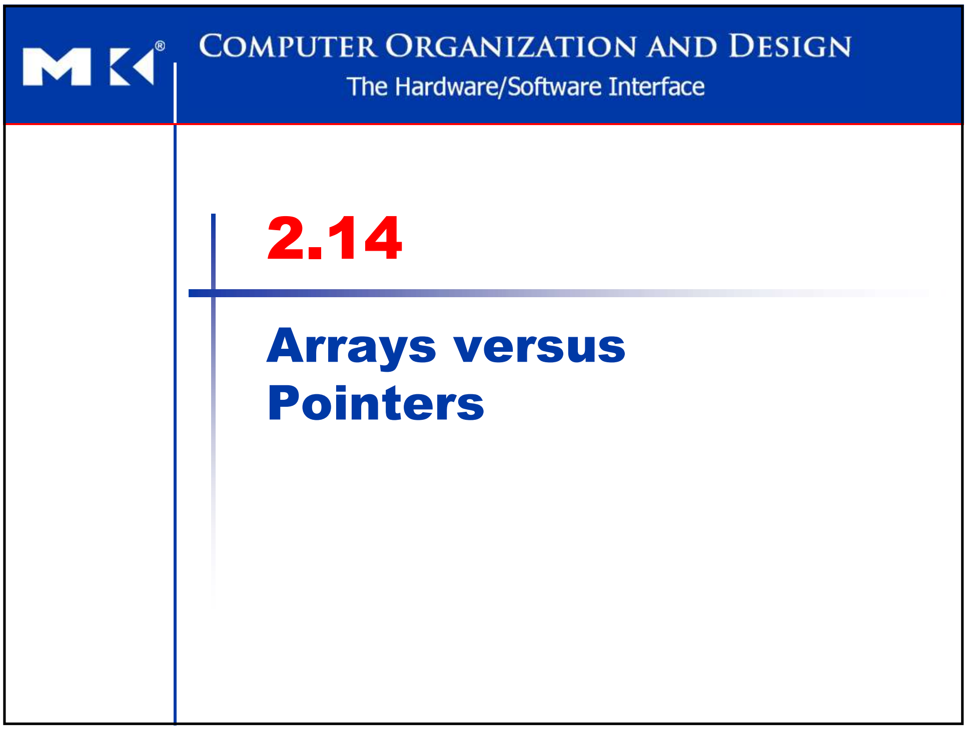


Example: Factorial (3/3)

```
#------------------------- procedure factorial -----------------------------
# load argument n in a0, return value in v0.
.text
factorial:
        addi $sp, $sp, -8  # adiust stack for 2 items
        sw $ra, 4($sp)          # save the return address
        sw $a0, 0($sp)          # save the argument n
        slti $t0, $a0, 1        # test for n < 1
        beq $t0, $zero, L1      # if n >= 1 go to L1
        addi $v0, $zero, 1      # return 1
        addi $sp, $sp, 8        # pop 2 items off stack
        jr $ra                  # return to caller
L1:     addi $a0, $a0, -1       # n >= 1, argument gets (n-1)
        jal factorial           # call factorial with (n-1)
        lw $a0, 0($sp)          # return from jal, restore argument n
        lw $ra, 4($sp)          # restore the return address
        addi $sp, $sp, 8        # adjust stack pointer to pop 2 items
        mul $v0, $a0, $v0       # return n*factorial(n-1)
        jr $ra                  # return to the caller
```

2- 118

118

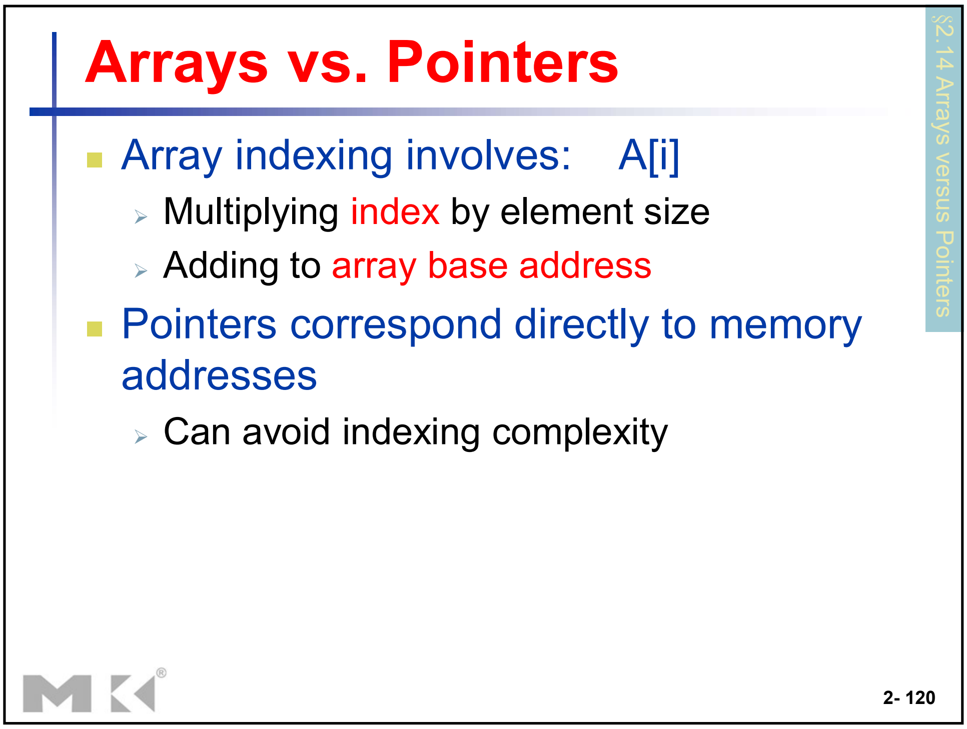


COMPUTER ORGANIZATION AND DESIGN
The Hardware/Software Interface

2.14

Arrays versus Pointers

119



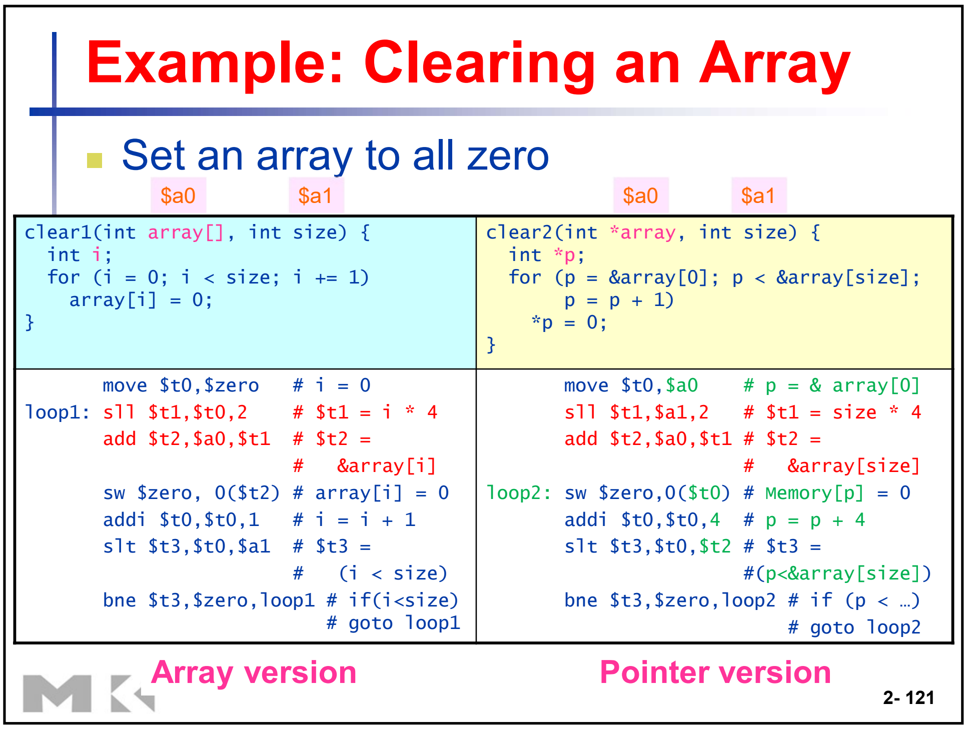
Arrays vs. Pointers

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

§2.14 Arrays versus Pointers

2- 120

120



Example: Clearing an Array

- Set an array to all zero

$a0 $a1 $a0 $a1

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

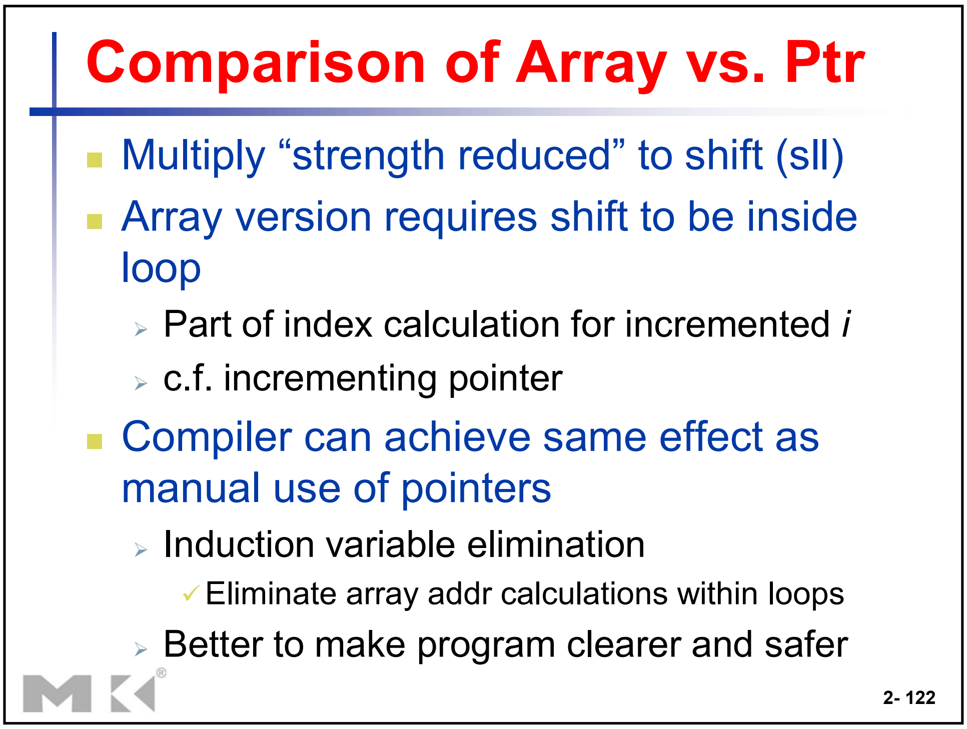
```
clear2(int *array, int size) {
  int *p;
  for (p = &array[0]; p < &array[size];
       p = p + 1)
    *p = 0;
}
```

```
       move $t0,$zero    # i = 0
loop1: 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(i<size)
                             # goto loop1
```

```
       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 (p < …)
                           # goto loop2
```

Array version   Pointer version

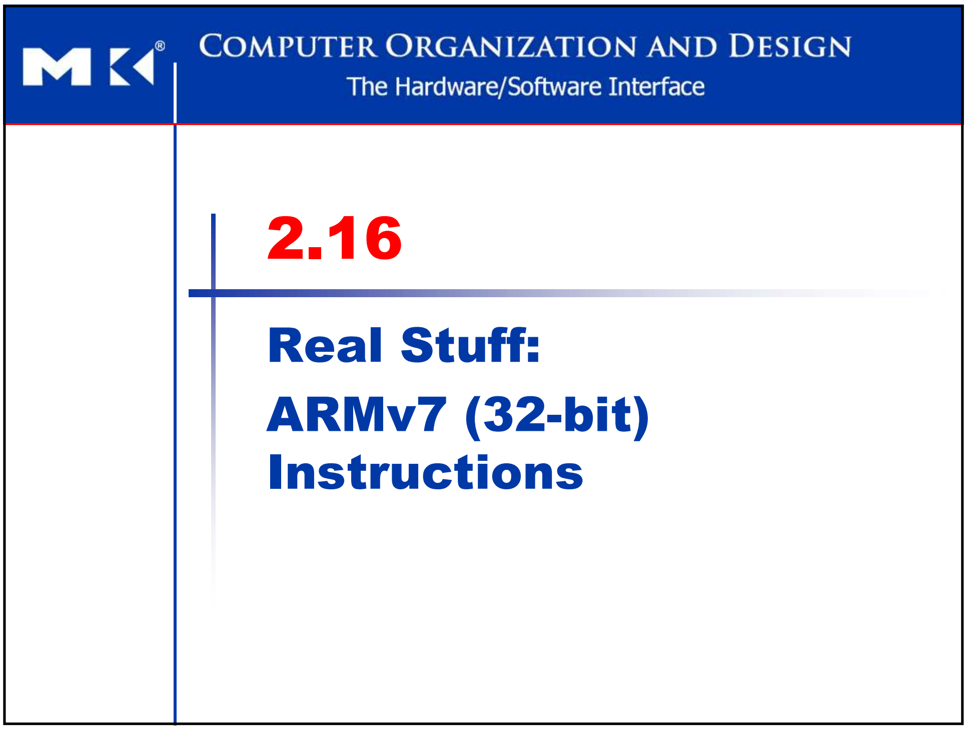
2- 121



Comparison of Array vs. Ptr

- Multiply “strength reduced” to shift (sll)
- 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
    - ✓Eliminate array addr calculations within loops
  - Better to make program clearer and safer

2- 122



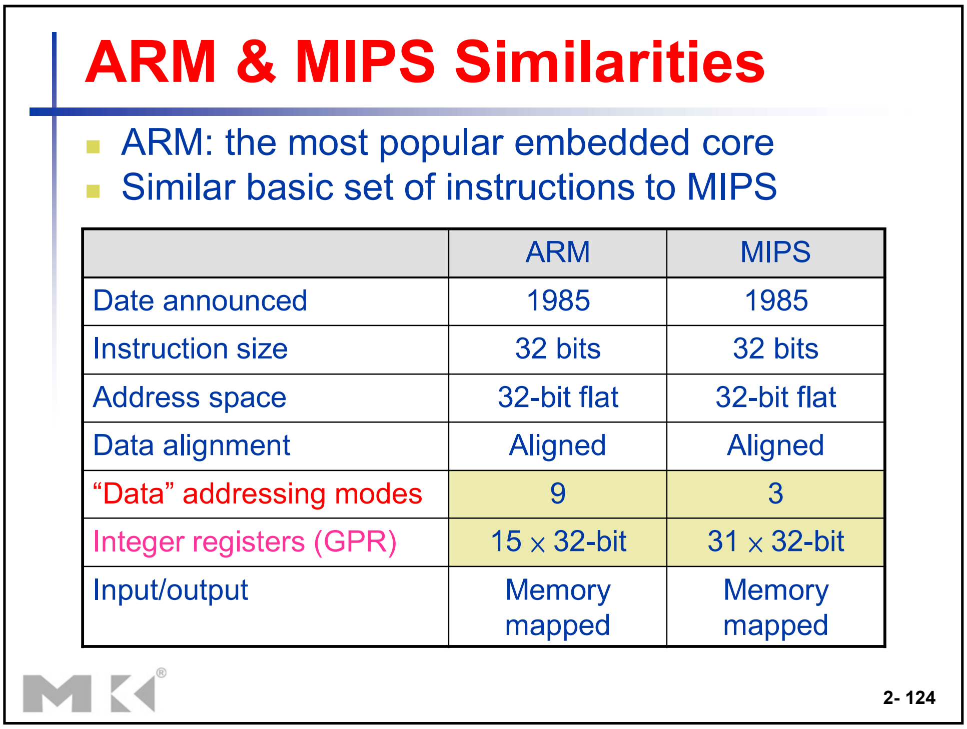
COMPUTER ORGANIZATION AND DESIGN

The Hardware/Software Interface

2.16

Real Stuff: ARMv7 (32-bit) Instructions

123



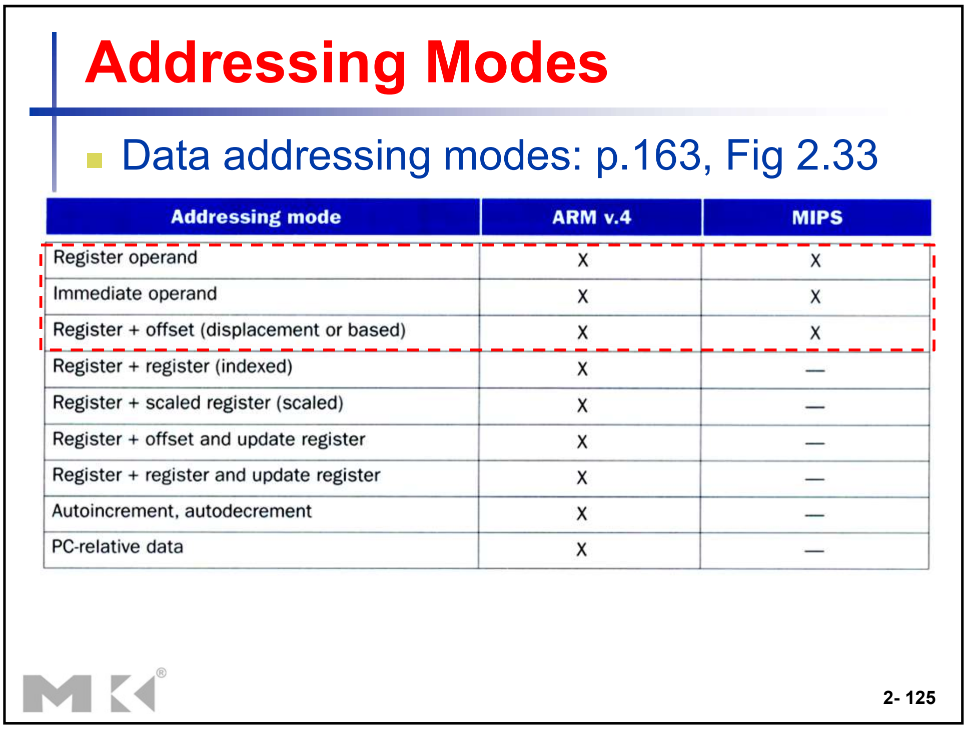
ARM & MIPS Similarities

- 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
Integer registers (GPR)	15 × 32-bit	31 × 32-bit
Input/output	Memory mapped	Memory mapped

2- 124

124



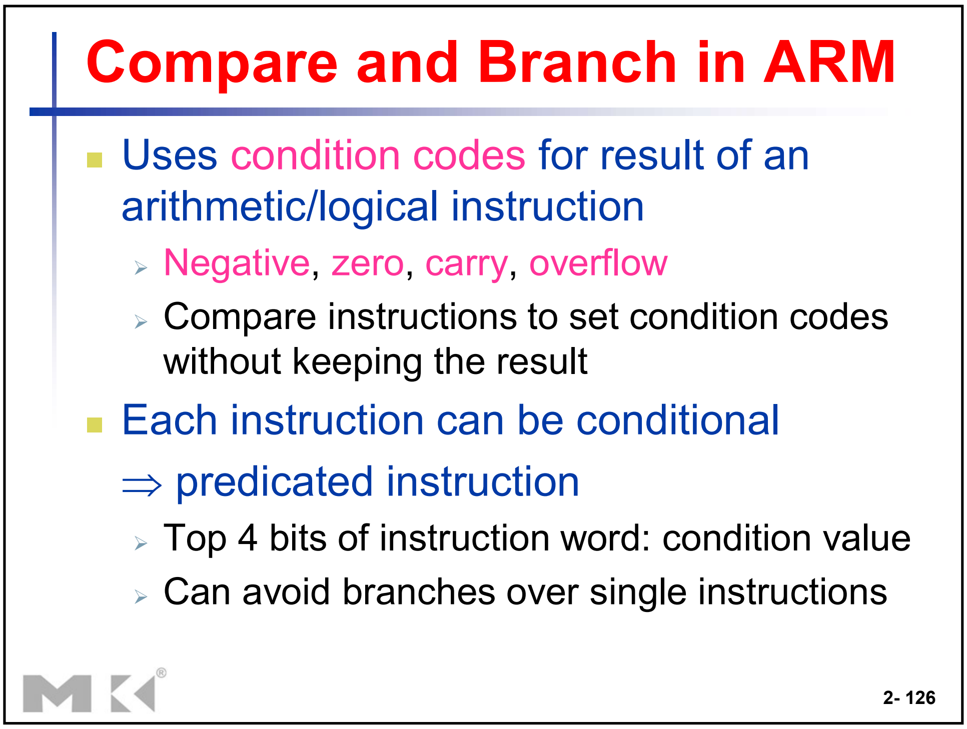
Addressing Modes

- Data addressing modes: p.163, Fig 2.33

Addressing mode	ARM v.4	MIPS
Register operand	X	X
Immediate operand	X	X
Register + offset (displacement or based)	X	X
Register + register (indexed)	X	—
Register + scaled register (scaled)	X	—
Register + offset and update register	X	—
Register + register and update register	X	—
Autoincrement, autodecrement	X	—
PC-relative data	X	—

2- 125

125



Compare and Branch in ARM

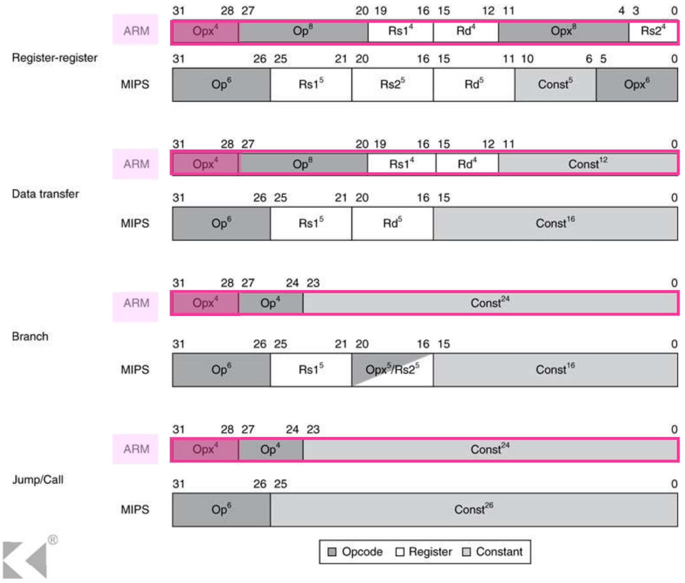
- Uses condition codes for result of an arithmetic/logical instruction
  - Negative, zero, carry, overflow
  - Compare instructions to set condition codes without keeping the result
- Each instruction can be conditional

  ⇒ predicated instruction
  - Top 4 bits of instruction word: condition value
  - Can avoid branches over single instructions

2- 126

126

Instruction Encoding



2- 127

127

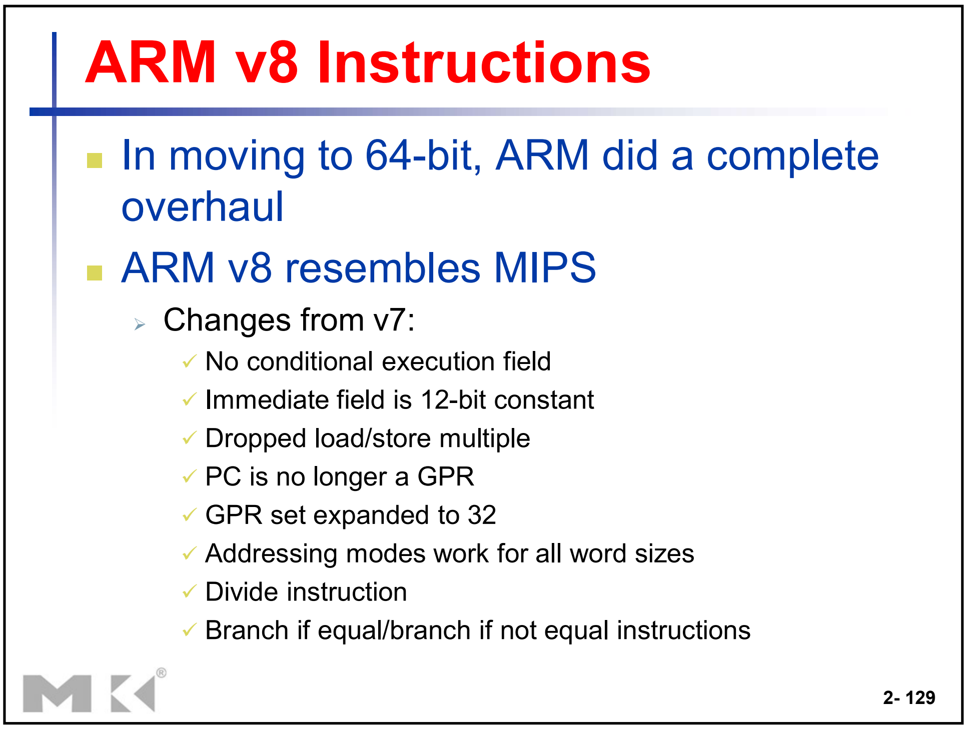
COMPUTER ORGANIZATION AND DESIGN

The Hardware/Software Interface

2.17

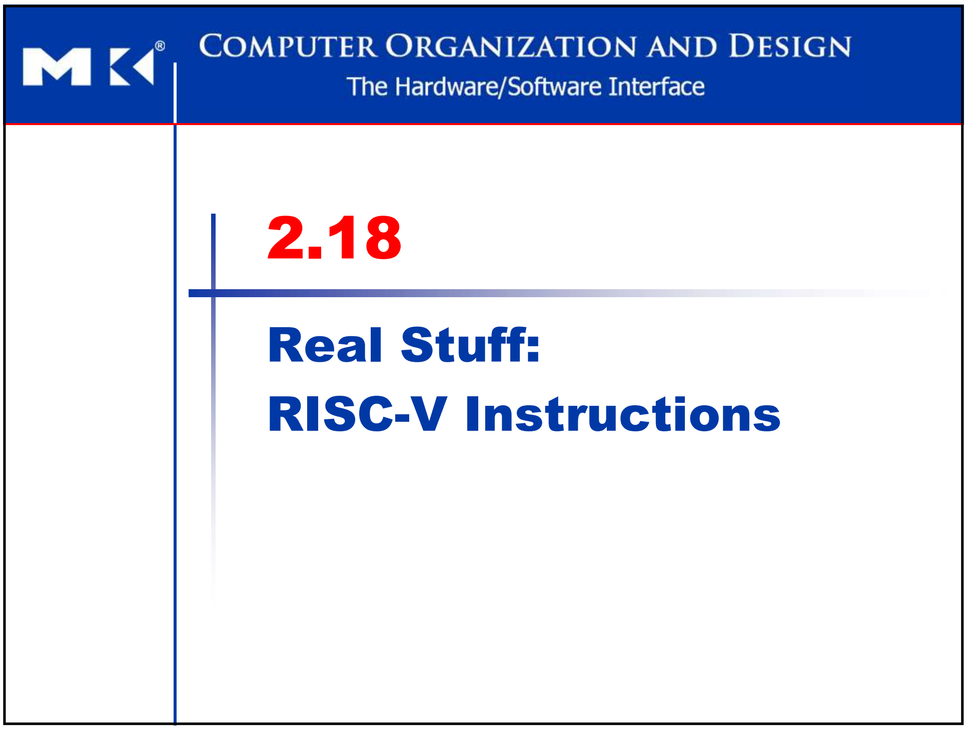
Real Stuff: ARMv8 (64-bit) Instructions

128



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

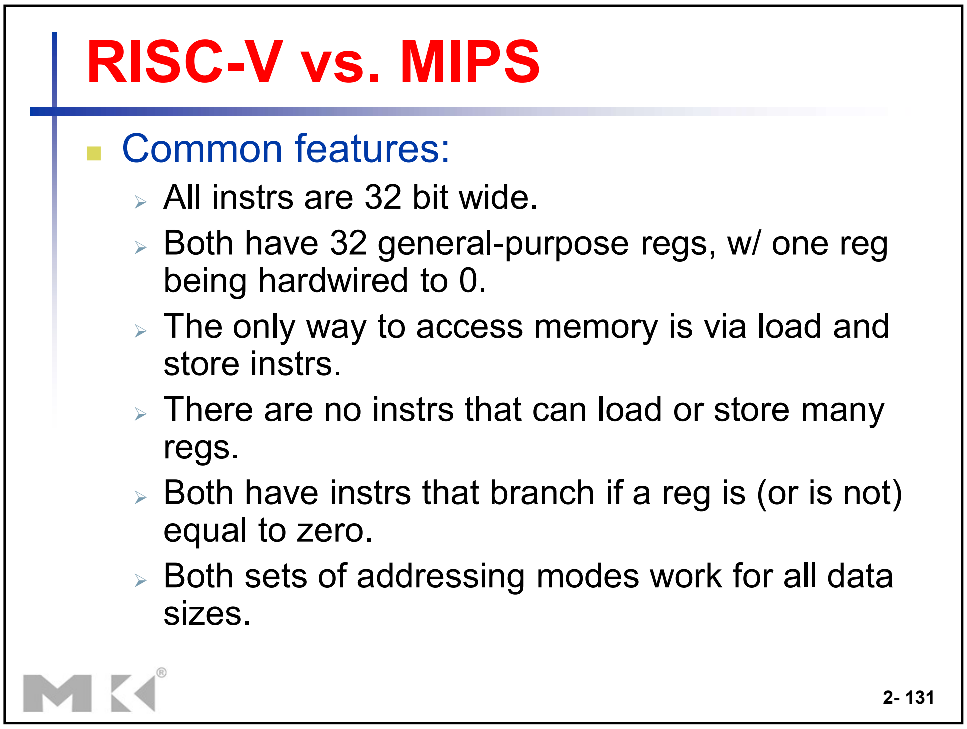


COMPUTER ORGANIZATION AND DESIGN

The Hardware/Software Interface

2.18

Real Stuff: RISC-V Instructions

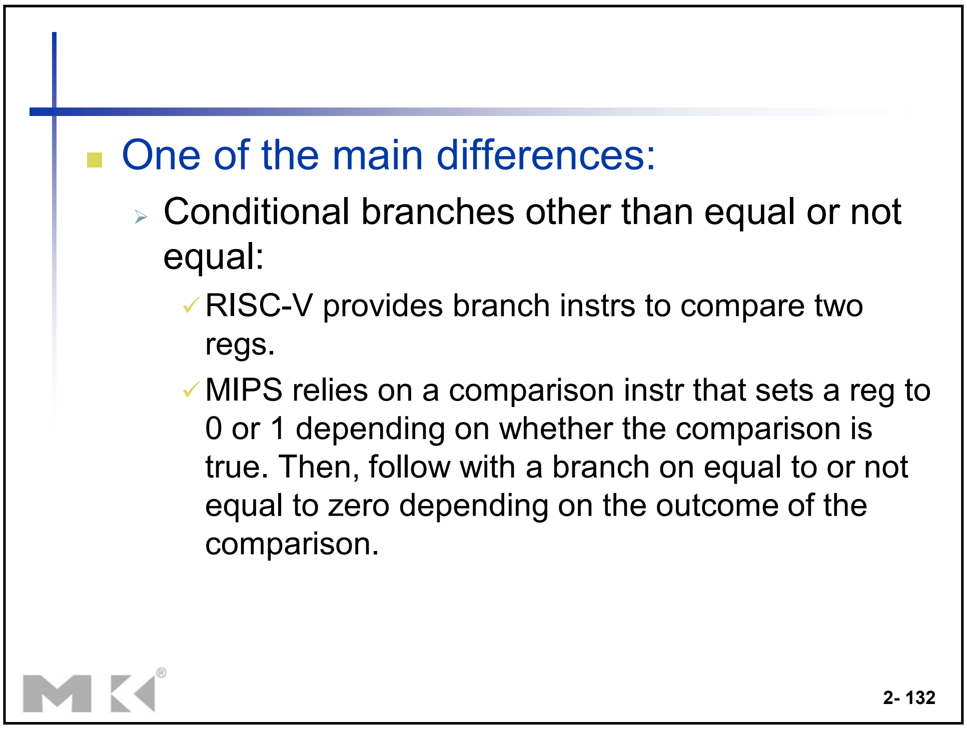


RISC-V vs. MIPS

- Common features:
  - All instrs are 32 bit wide.
  - Both have 32 general-purpose regs, w/ one reg being hardwired to 0.
  - The only way to access memory is via load and store instrs.
  - There are no instrs that can load or store many regs.
  - Both have instrs that branch if a reg is (or is not) equal to zero.
  - Both sets of addressing modes work for all data sizes.

2- 131

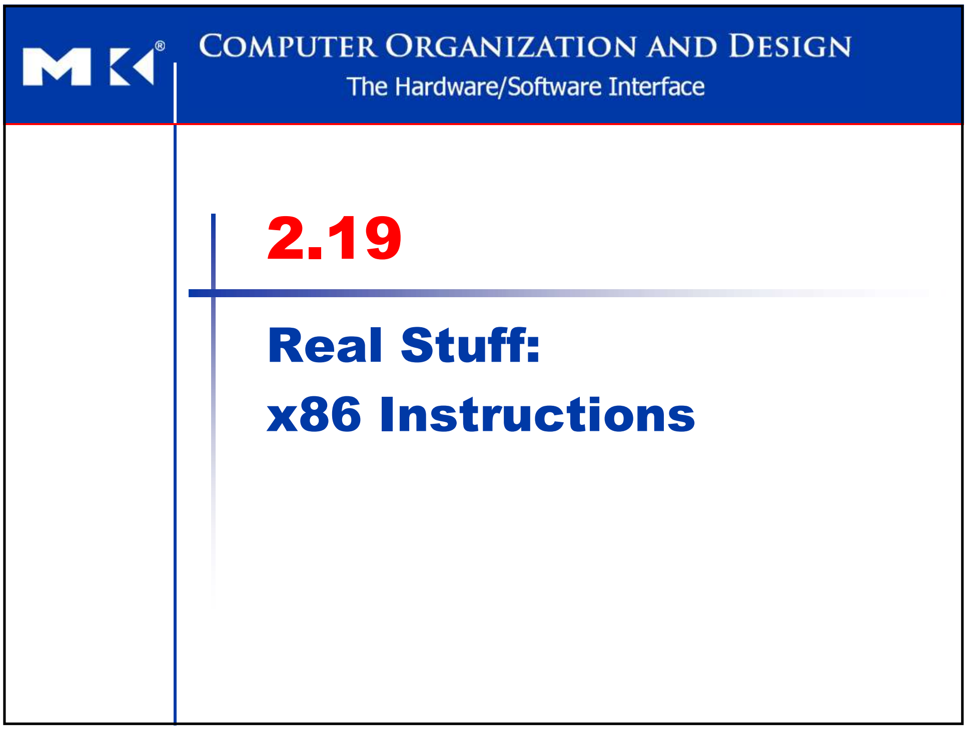
131



- One of the main differences:
  - Conditional branches other than equal or not equal:
    - ✓RISC-V provides branch instrs to compare two regs.
    - ✓MIPS relies on a comparison instr that sets a reg to 0 or 1 depending on whether the comparison is true. Then, follow with a branch on equal to or not equal to zero depending on the outcome of the comparison.

2- 132

132

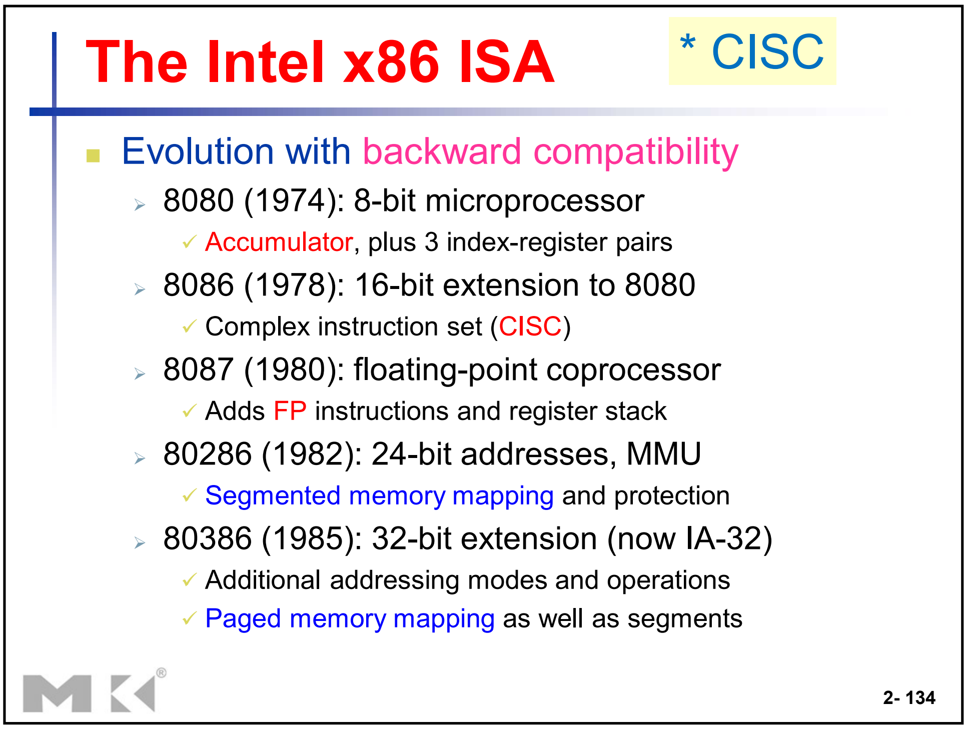


COMPUTER ORGANIZATION AND DESIGN
The Hardware/Software Interface

2.19

Real Stuff: x86 Instructions

133



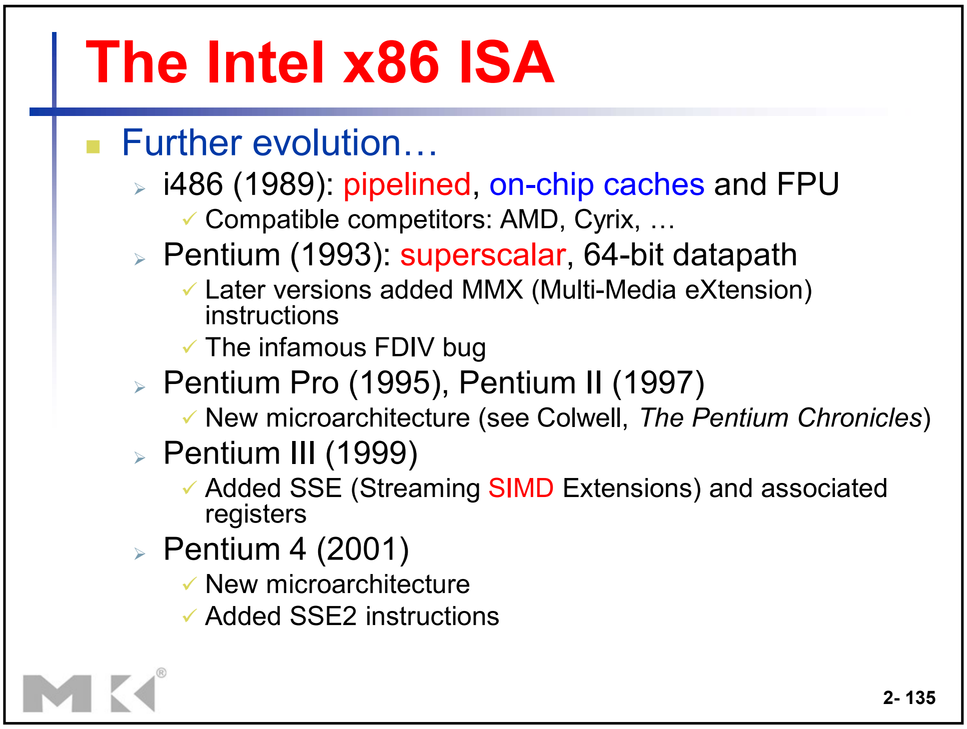
The Intel x86 ISA

* CISC

- 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

2- 134

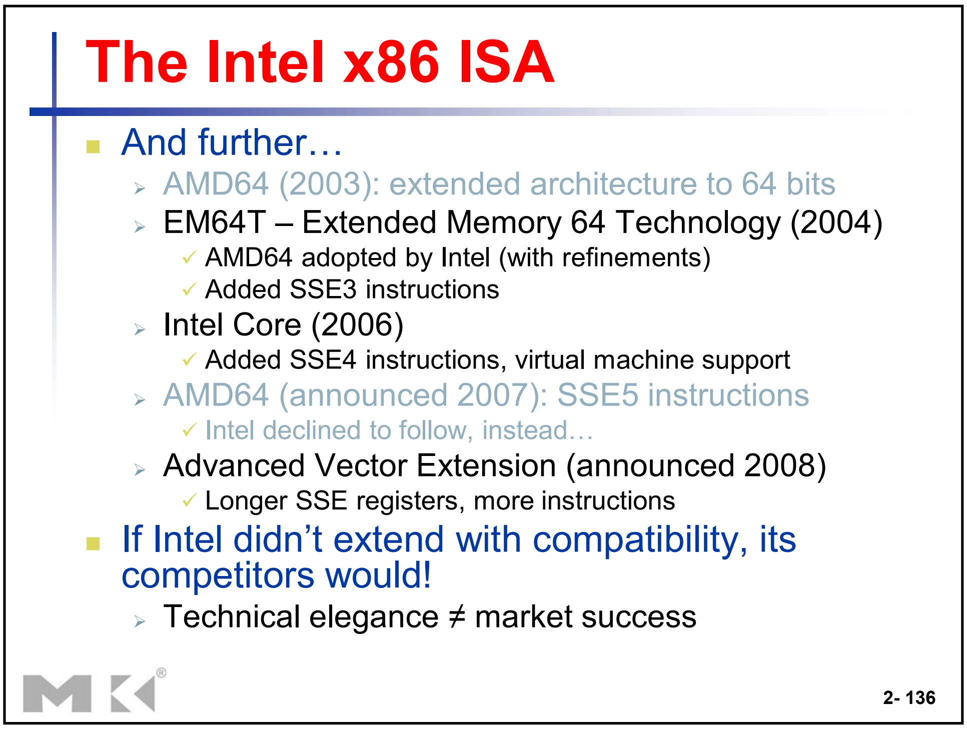
134



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

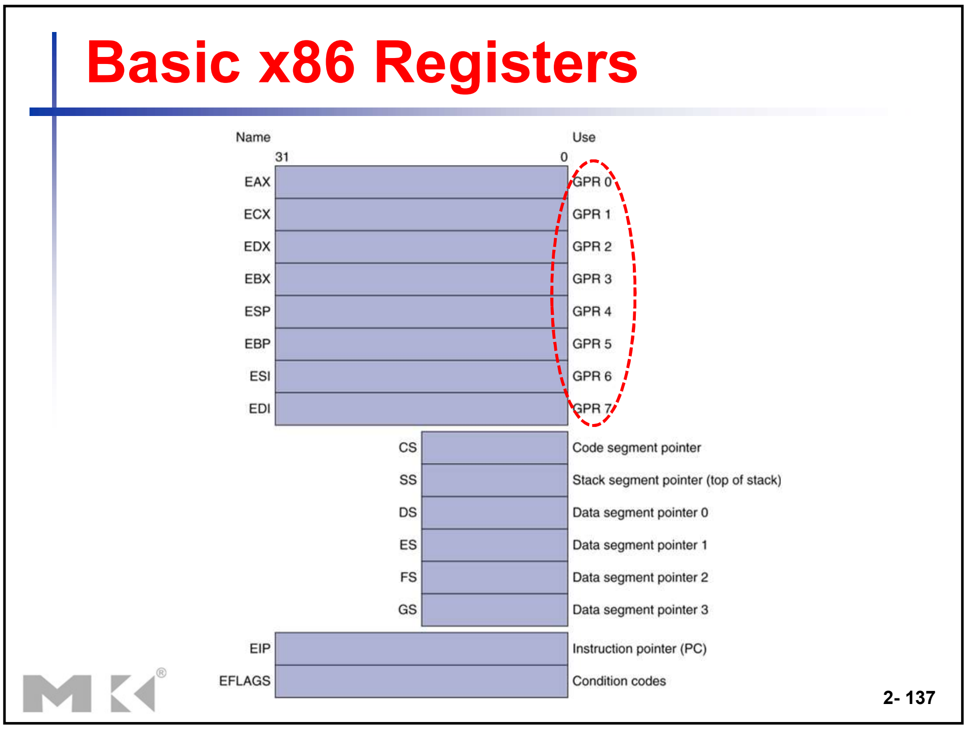
2- 135



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

2- 136

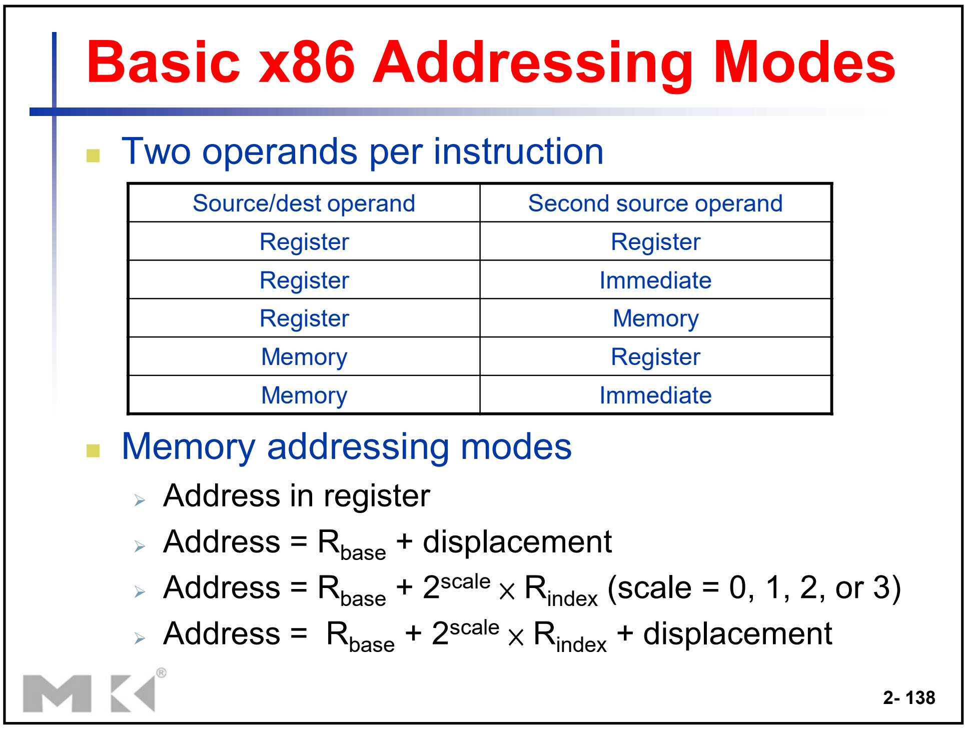


Basic x86 Registers

Name	31 … 0	Use
EAX		GPR 0
ECX		GPR 1
EDX		GPR 2
EBX		GPR 3
ESP		GPR 4
EBP		GPR 5
ESI		GPR 6
EDI		GPR 7
CS		Code segment pointer
SS		Stack segment pointer (top of stack)
DS		Data segment pointer 0
ES		Data segment pointer 1
FS		Data segment pointer 2
GS		Data segment pointer 3
EIP		Instruction pointer (PC)
EFLAGS		Condition codes

2- 137

137



Basic x86 Addressing Modes

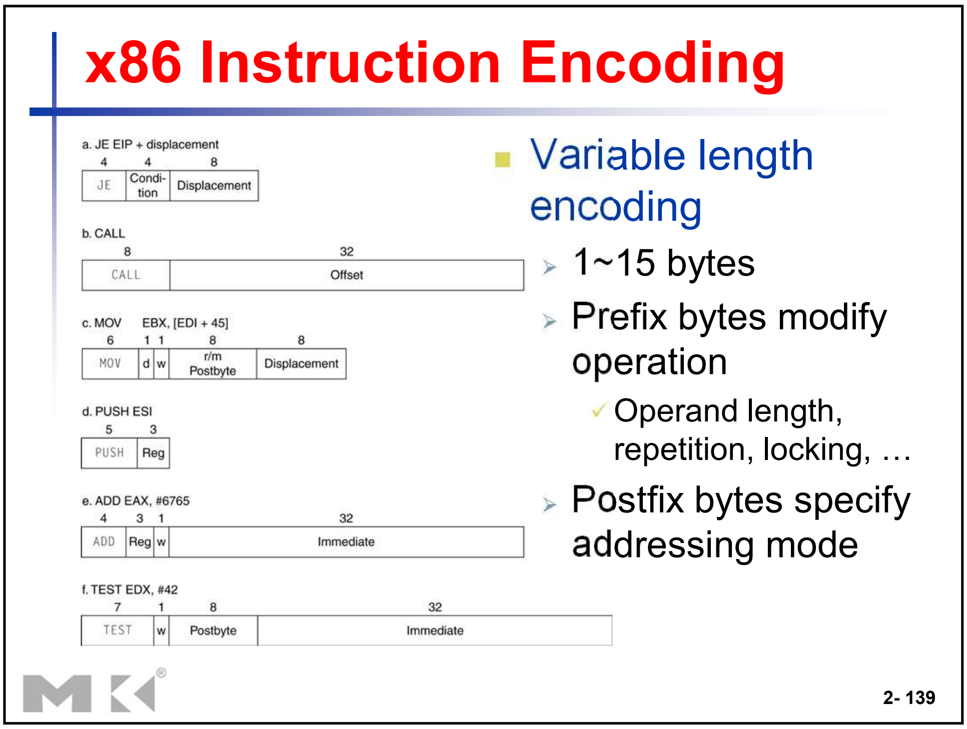
- Two operands per instruction

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

- Memory addressing modes
  - Address in register
  - Address = R_{base} + displacement
  - Address = $R_{base} + 2^{scale} \times R_{index}$ (scale = 0, 1, 2, or 3)
  - Address = $R_{base} + 2^{scale} \times R_{index}$ + displacement

2- 138

138



x86 Instruction Encoding

a. JE EIP + displacement

4	4	8
JE	Condi-tion	Displacement

b. CALL

8	32
CALL	Offset

c. MOV EBX, [EDI + 45]

6	1	1	8	8
MOV	d	w	r/m Postbyte	Displacement

d. PUSH ESI

5	3
PUSH	Reg

e. ADD EAX, #6765

4	3	1	32
ADD	Reg	w	Immediate

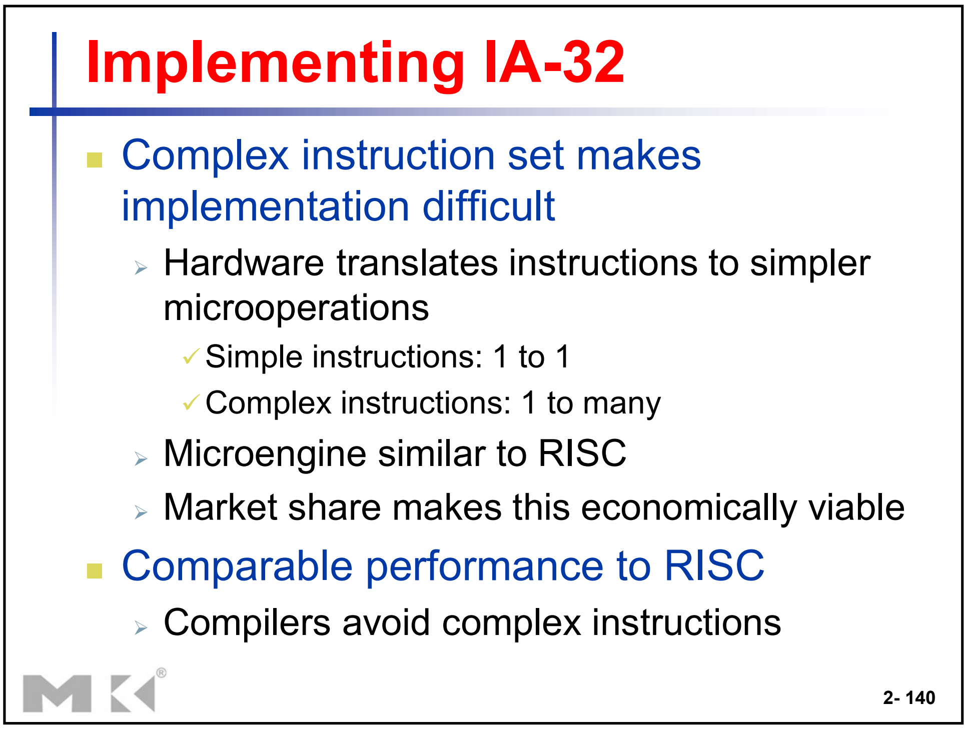
f. TEST EDX, #42

7	1	8	32
TEST	w	Postbyte	Immediate

- Variable length encoding
  - 1~15 bytes
  - Prefix bytes modify operation
    - ✓ Operand length, repetition, locking, …
  - Postfix bytes specify addressing mode

2- 139

139

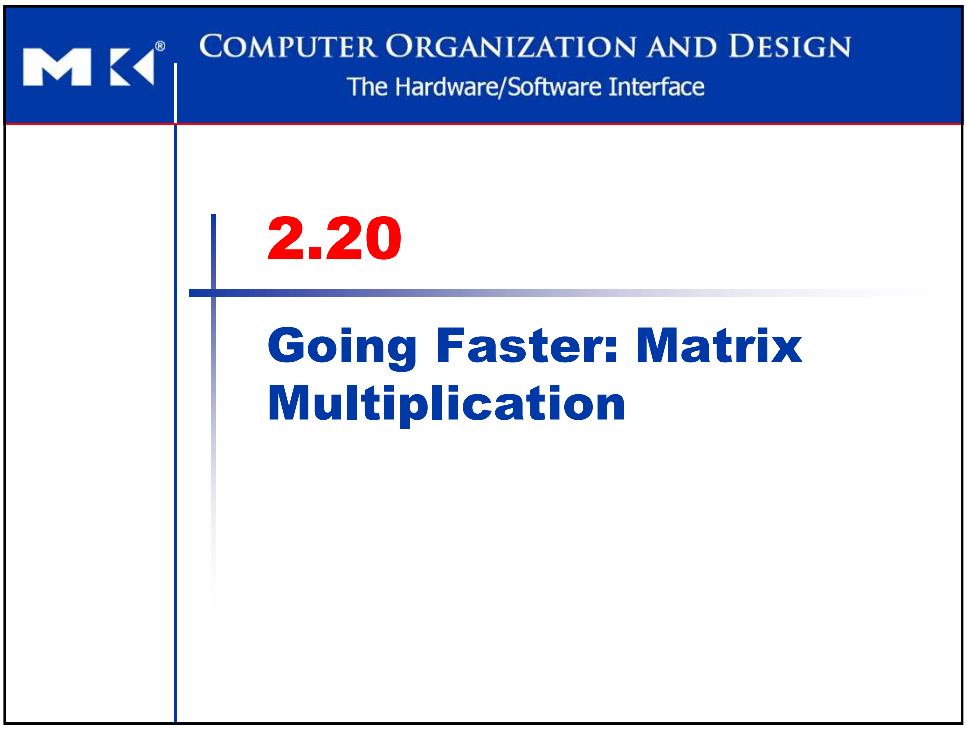


Implementing IA-32

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

2- 140

140



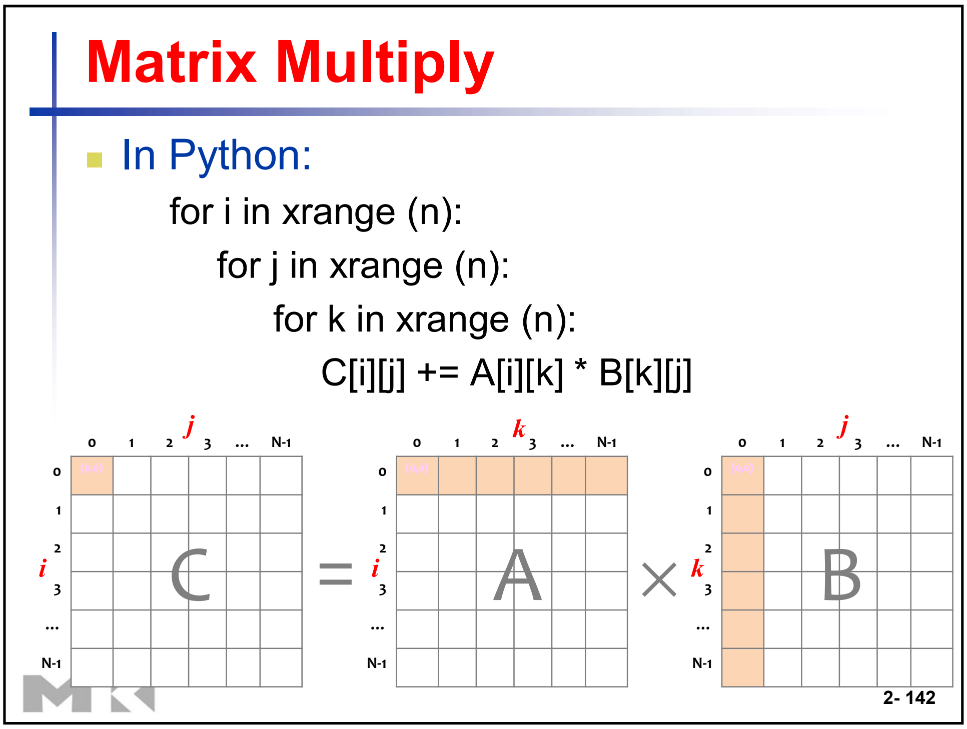
COMPUTER ORGANIZATION AND DESIGN

The Hardware/Software Interface

2.20

Going Faster: Matrix Multiplication

141



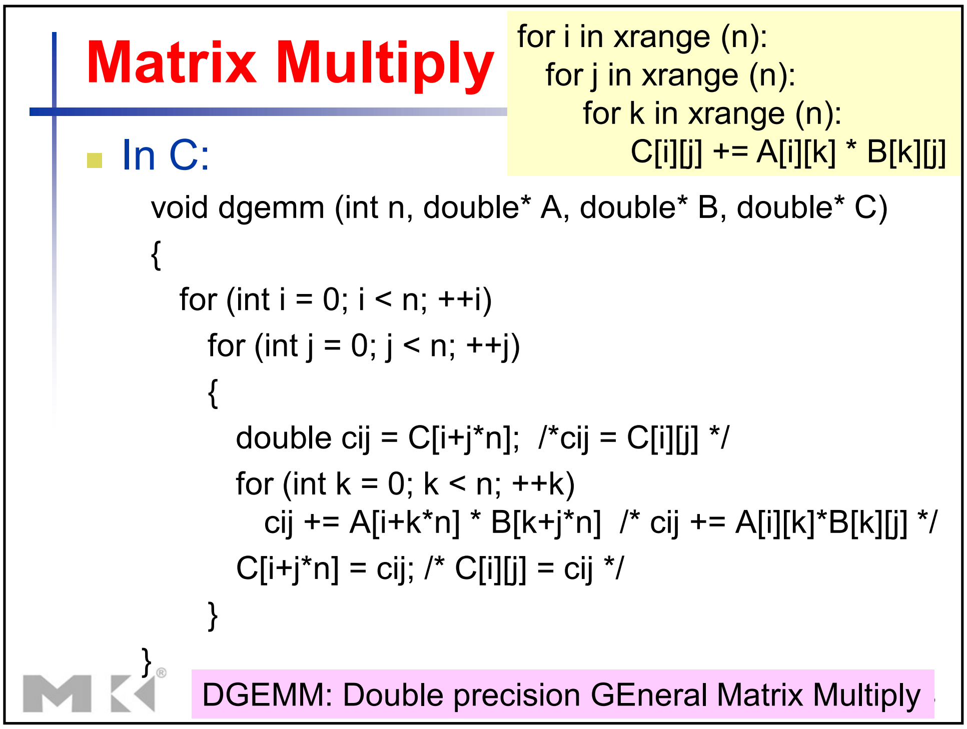
Matrix Multiply

- In Python:

```
for i in xrange (n):
    for j in xrange (n):
        for k in xrange (n):
            C[i][j] += A[i][k] * B[k][j]
```

2- 142

142



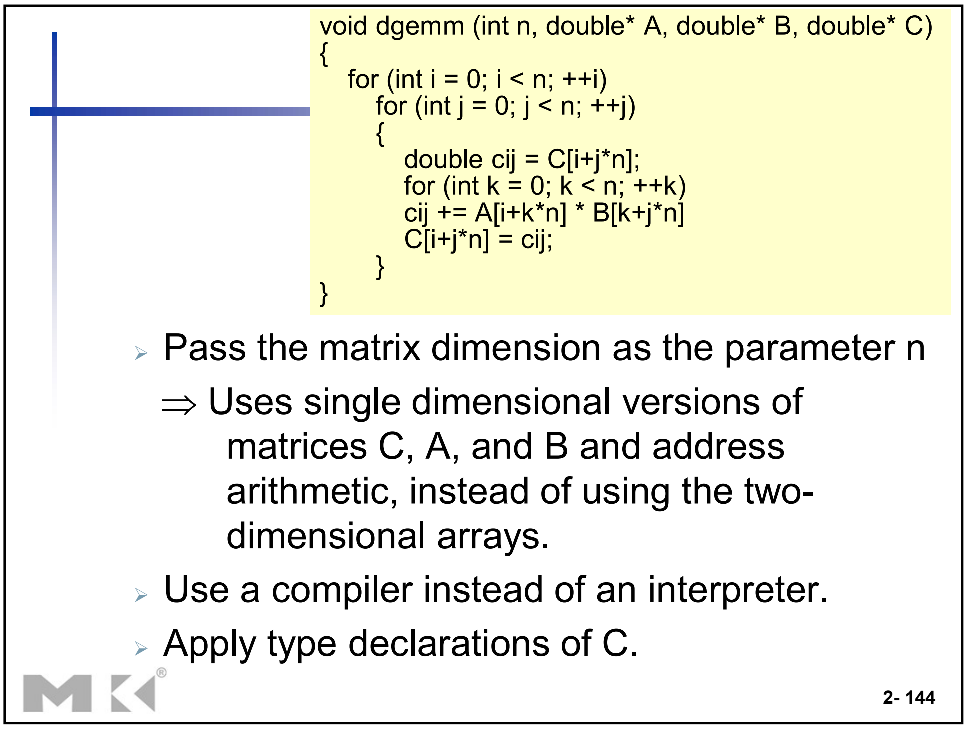
Matrix Multiply

```
for i in xrange (n):
  for j in xrange (n):
     for k in xrange (n):
        C[i][j] += A[i][k] * B[k][j]
```

- In C:

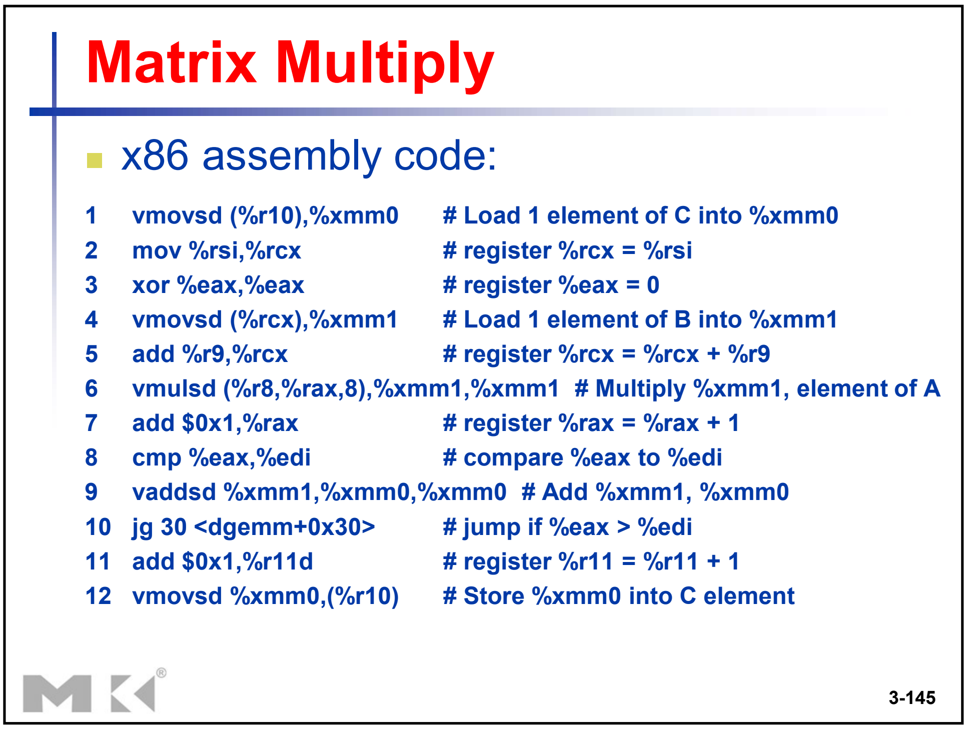
```
void dgemm (int n, double* A, double* B, double* C)
{
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
    {
      double cij = C[i+j*n];  /*cij = C[i][j] */
      for (int k = 0; k < n; ++k)
        cij += A[i+k*n] * B[k+j*n]  /* cij += A[i][k]*B[k][j] */
      C[i+j*n] = cij; /* C[i][j] = cij */
    }
}
```

DGEMM: Double precision GEneral Matrix Multiply



```
void dgemm (int n, double* A, double* B, double* C)
{
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
    {
      double cij = C[i+j*n];
      for (int k = 0; k < n; ++k)
      cij += A[i+k*n] * B[k+j*n]
      C[i+j*n] = cij;
    }
}
```

- Pass the matrix dimension as the parameter n
  ⇒ Uses single dimensional versions of matrices C, A, and B and address arithmetic, instead of using the two-dimensional arrays.
- Use a compiler instead of an interpreter.
- Apply type declarations of C.



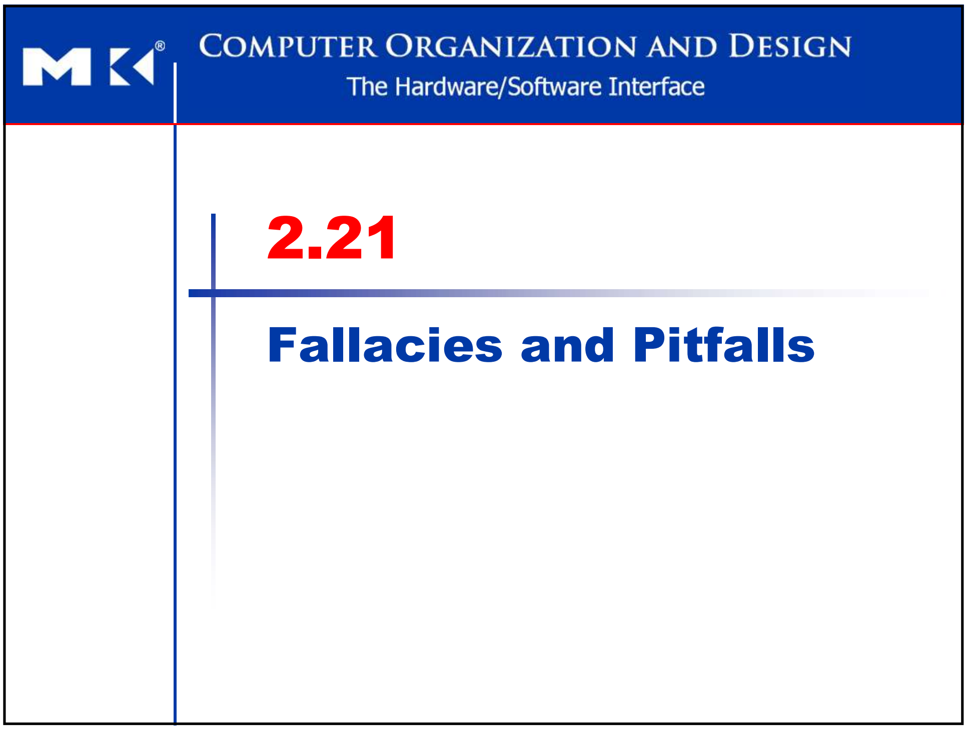
Matrix Multiply

- x86 assembly code:

```
vmovsd (%r10),%xmm0          # Load 1 element of C into %xmm0
mov %rsi,%rcx                # register %rcx = %rsi
xor %eax,%eax                # register %eax = 0
vmovsd (%rcx),%xmm1          # Load 1 element of B into %xmm1
add %r9,%rcx                 # register %rcx = %rcx + %r9
vmulsd (%r8,%rax,8),%xmm1,%xmm1  # Multiply %xmm1, element of A
add $0x1,%rax                # register %rax = %rax + 1
cmp %eax,%edi                # compare %eax to %edi
vaddsd %xmm1,%xmm0,%xmm0  # Add %xmm1, %xmm0
jg 30 <dgemm+0x30>           # jump if %eax > %edi
add $0x1,%r11d               # register %r11 = %r11 + 1
vmovsd %xmm0,(%r10)          # Store %xmm0 into C element
```

3-145

145

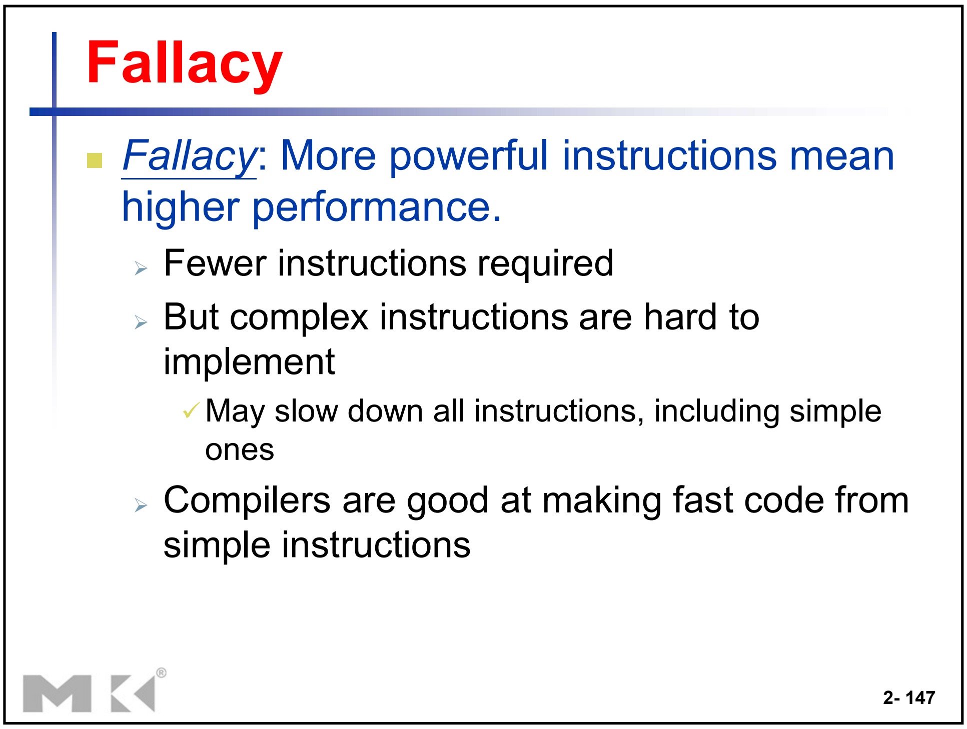


COMPUTER ORGANIZATION AND DESIGN
The Hardware/Software Interface

2.21

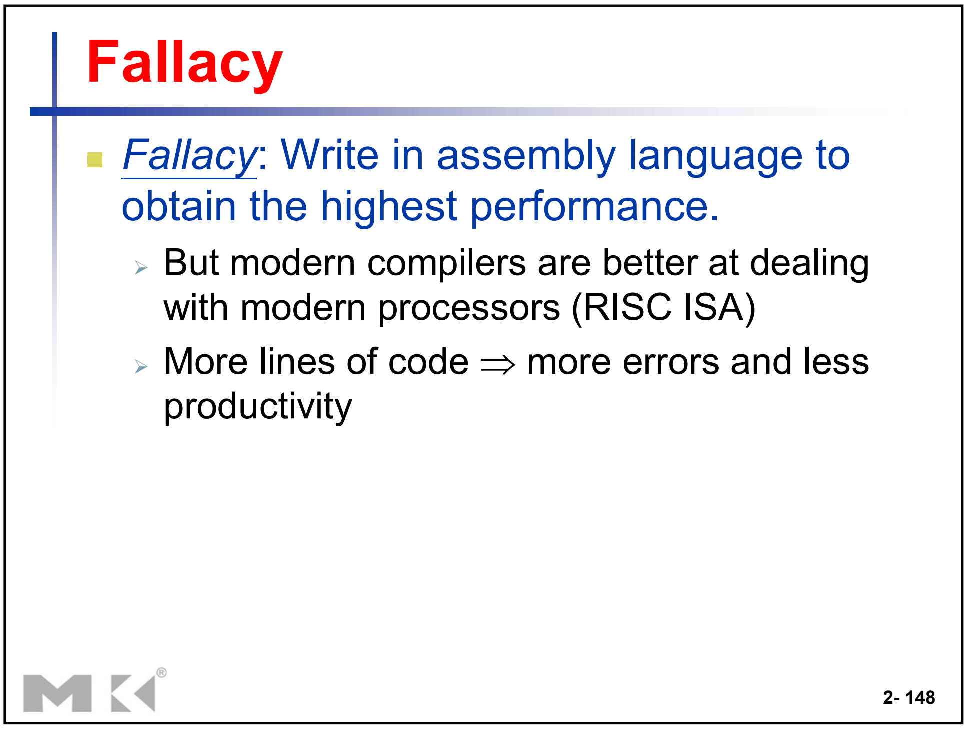
Fallacies and Pitfalls

146



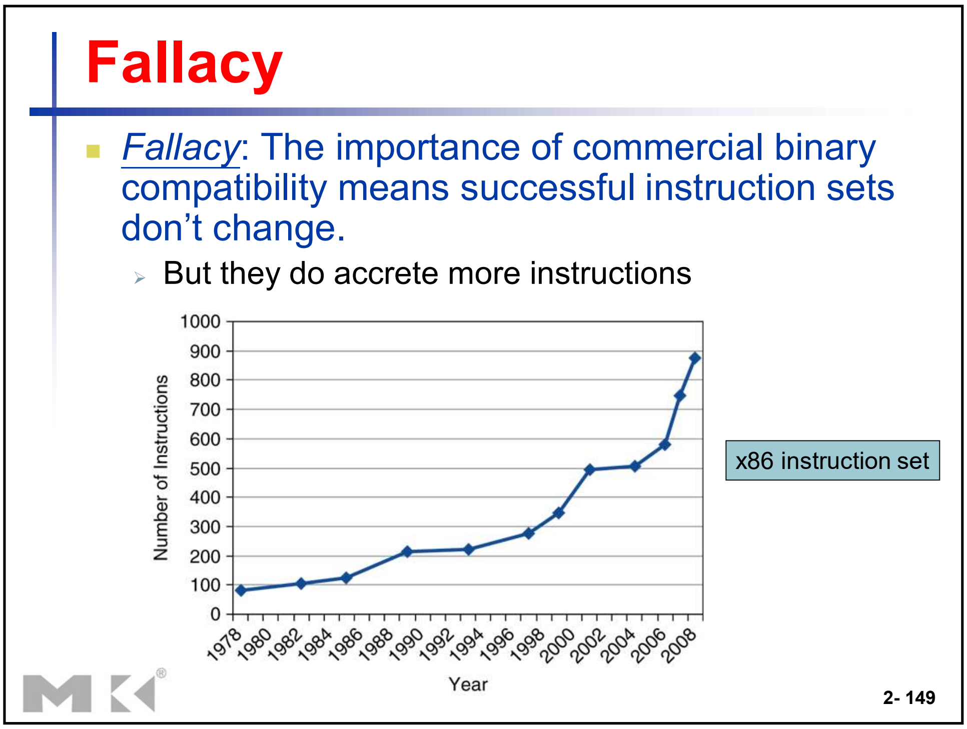
Fallacy

- *Fallacy*: More powerful instructions mean 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



Fallacy

- *Fallacy*: Write in assembly language to obtain the highest performance.
  - But modern compilers are better at dealing with modern processors (RISC ISA)
  - More lines of code ⇒ more errors and less productivity

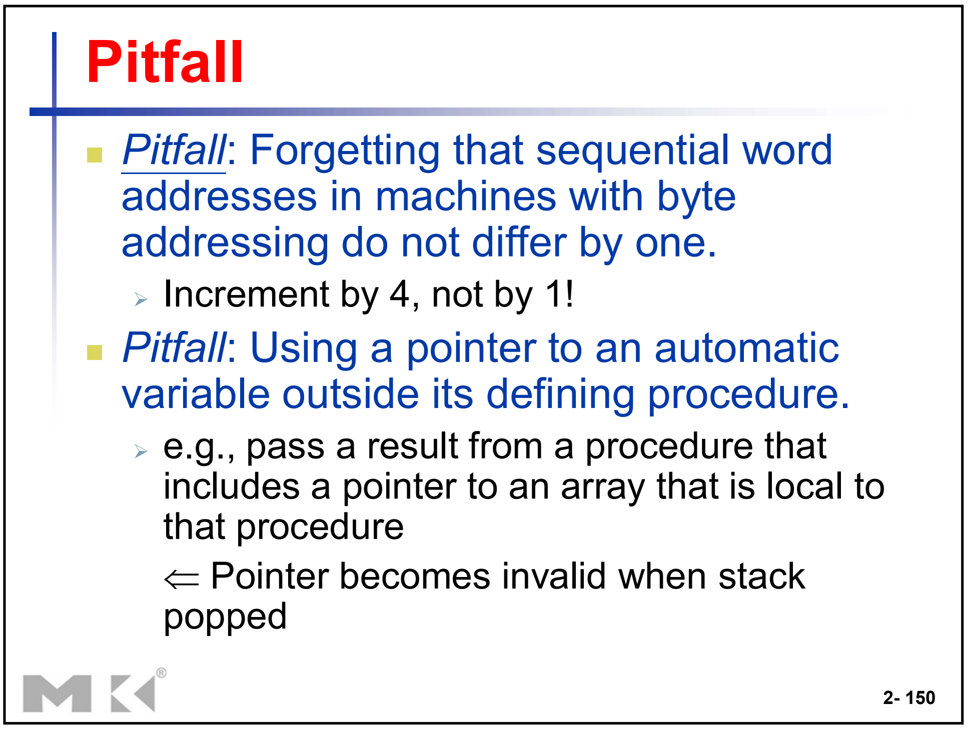


Fallacy

- *Fallacy*: The importance of commercial binary compatibility means successful instruction sets don't change.
  - But they do accrete more instructions

2- 149

149



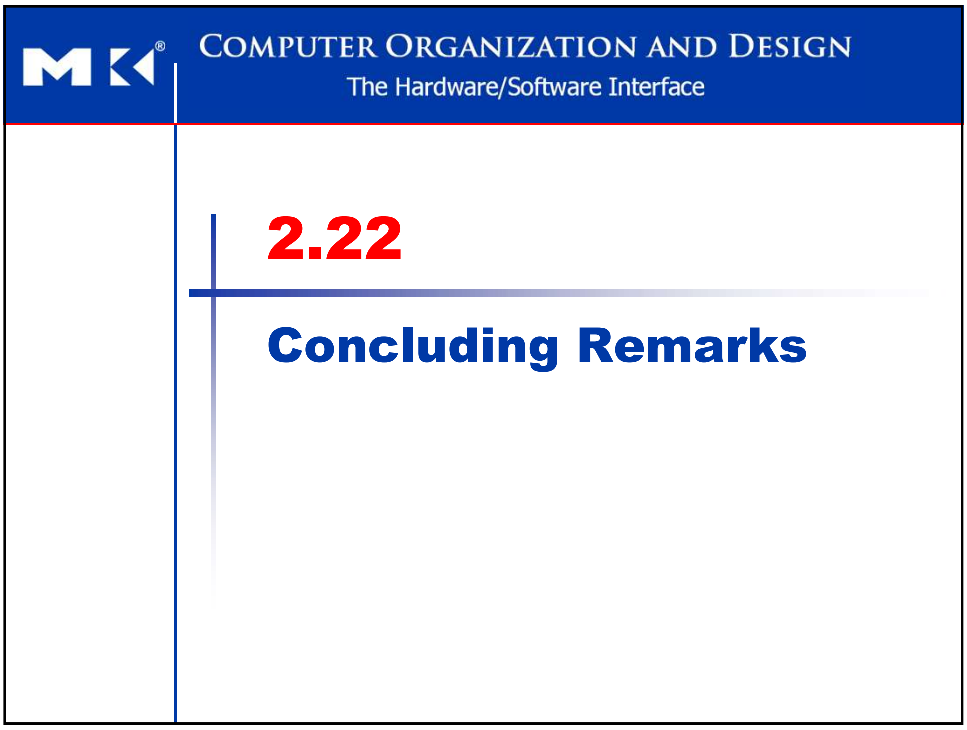
Pitfall

- *Pitfall*: Forgetting that sequential word addresses in machines with byte addressing do not differ by one.
  - Increment by 4, not by 1!
- *Pitfall*: Using a pointer to an automatic variable outside its defining procedure.
  - e.g., pass a result from a procedure that includes a pointer to an array that is local to that procedure

    ⇐ Pointer becomes invalid when stack popped

2- 150

150

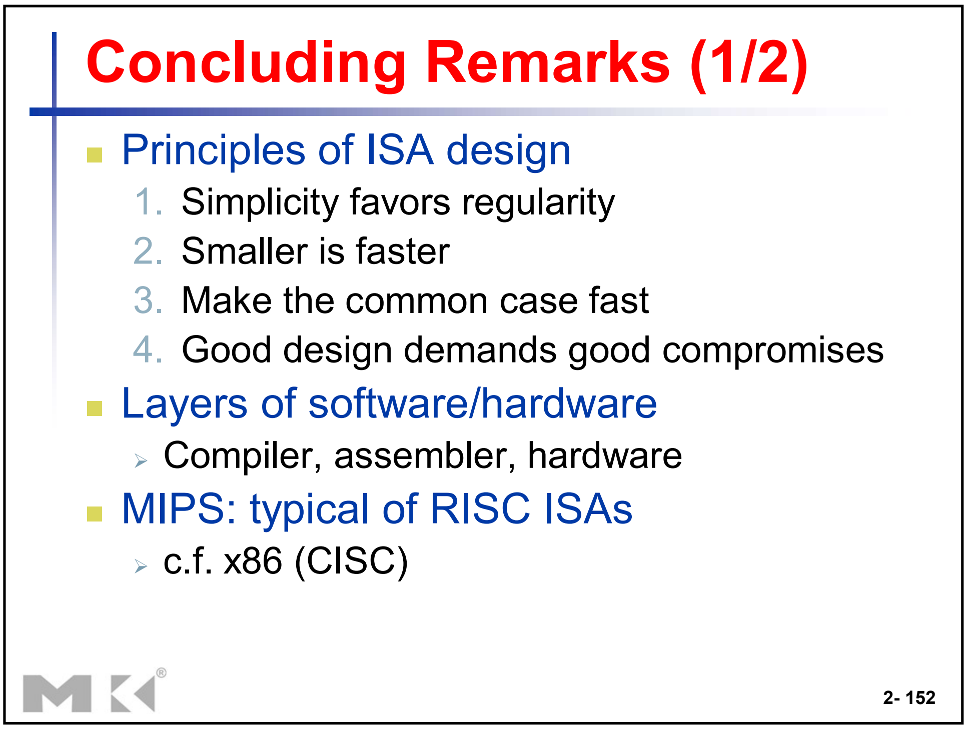


COMPUTER ORGANIZATION AND DESIGN
The Hardware/Software Interface

2.22

Concluding Remarks

151

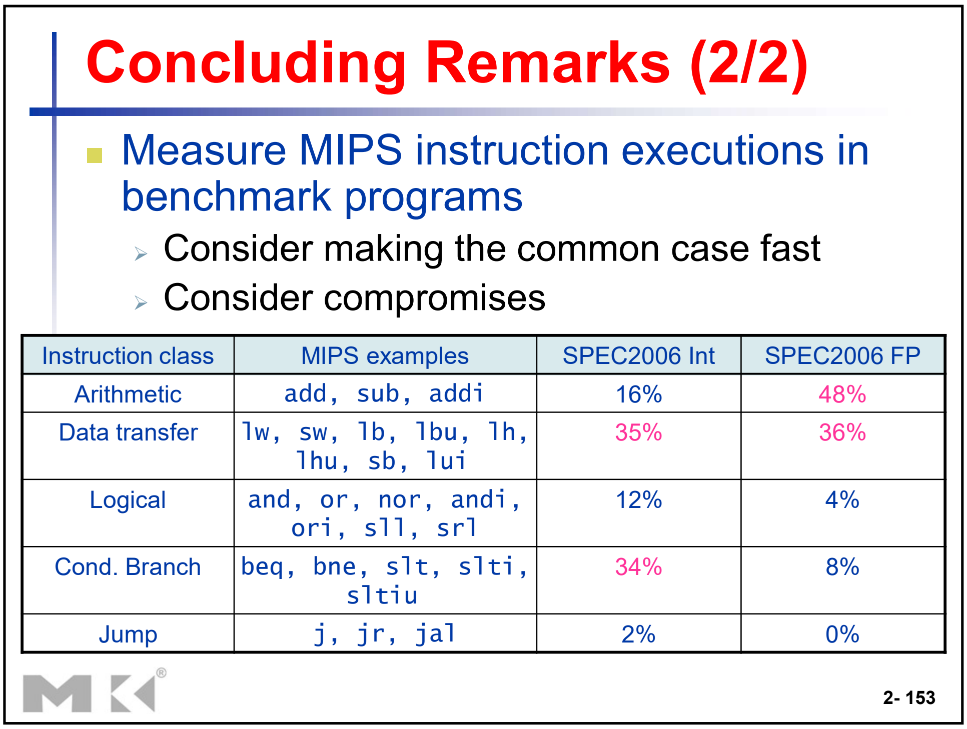


Concluding Remarks (1/2)

- Principles of ISA design
  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 (CISC)

2- 152

152



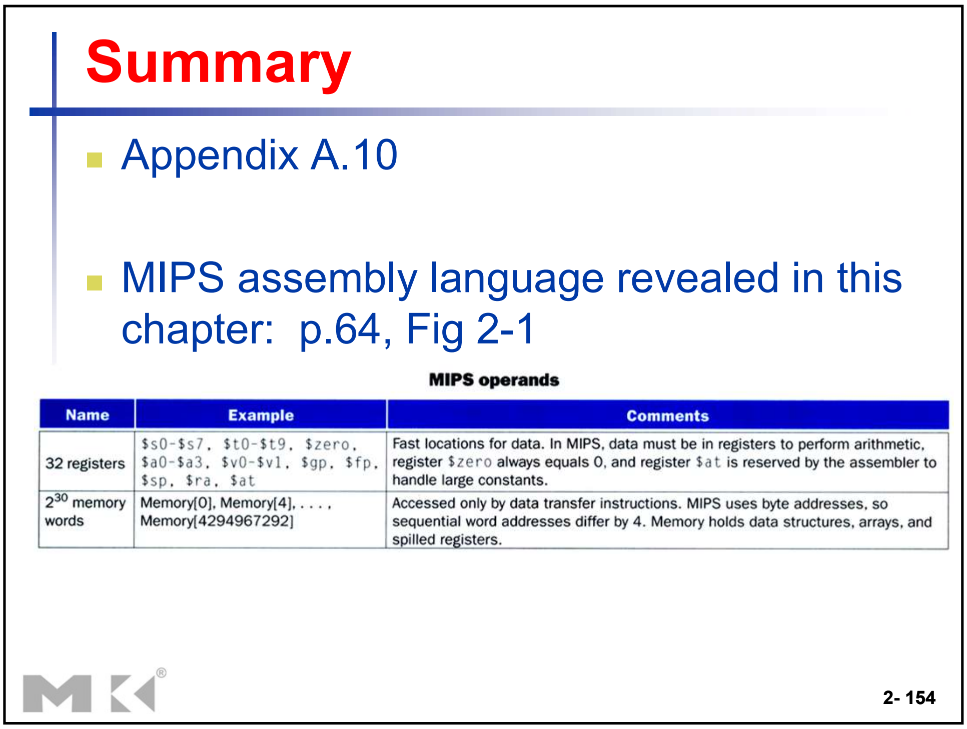
Concluding Remarks (2/2)

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

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

2- 153

153



Summary

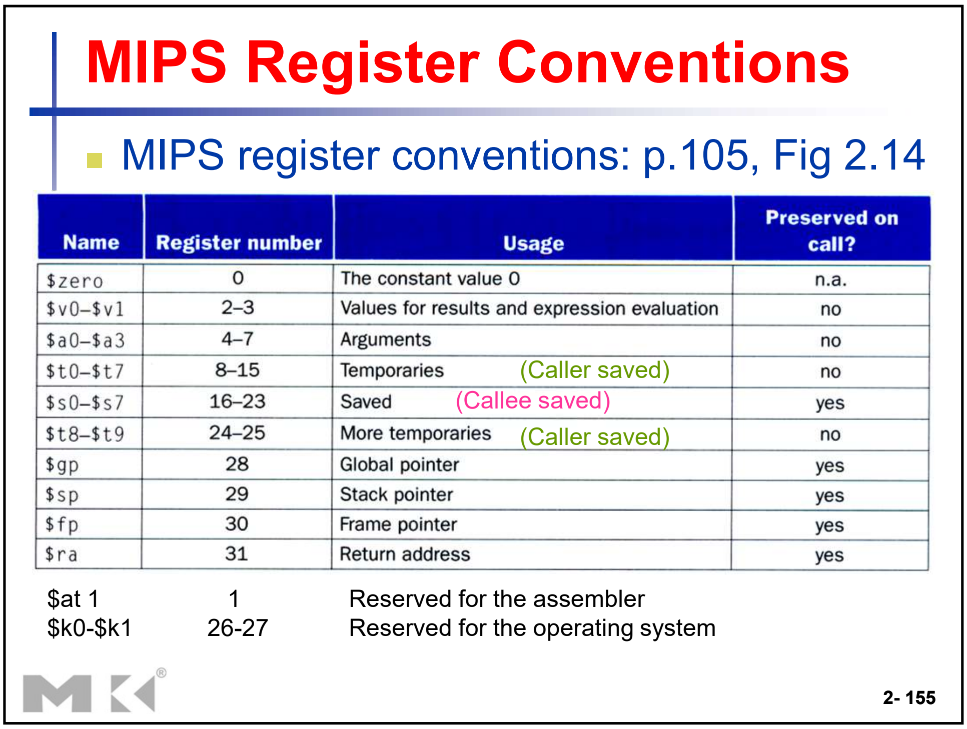
- Appendix A.10
- MIPS assembly language revealed in this chapter: p.64, Fig 2-1

MIPS operands

Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic, register `$zero` always equals 0, and register `$at` is reserved by the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

2- 154

154

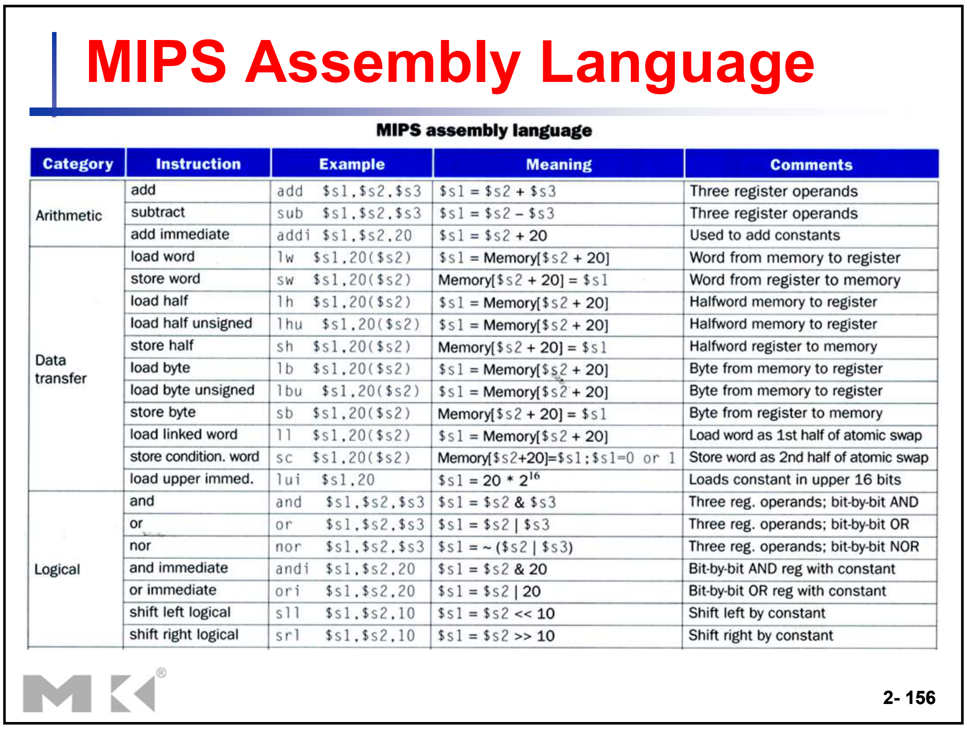


MIPS Register Conventions

- MIPS register conventions: p.105, Fig 2.14

Name	Register number	Usage	Preserved on call?
$zero	0	The constant value 0	n.a.
$v0–$v1	2–3	Values for results and expression evaluation	no
$a0–$a3	4–7	Arguments	no
$t0–$t7	8–15	Temporaries (Caller saved)	no
$s0–$s7	16–23	Saved (Callee saved)	yes
$t8–$t9	24–25	More temporaries (Caller saved)	no
$gp	28	Global pointer	yes
$sp	29	Stack pointer	yes
$fp	30	Frame pointer	yes
$ra	31	Return address	yes

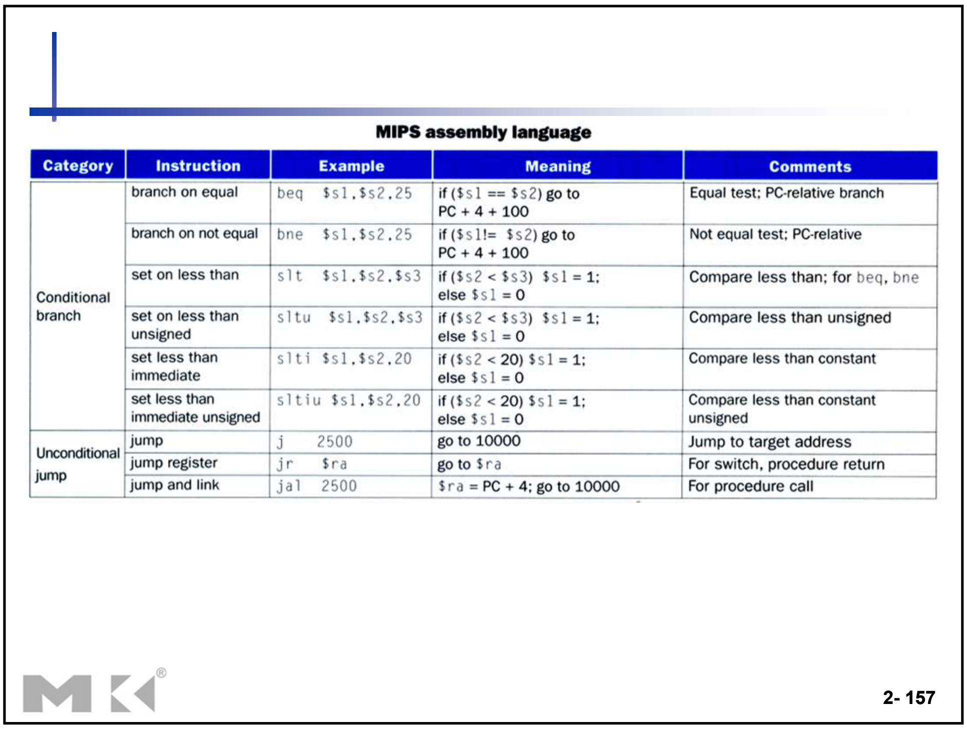
$at 1 1 Reserved for the assembler
$k0-$k1 26-27 Reserved for the operating system



MIPS Assembly Language

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three register operands
	subtract	sub $s1,$s2,$s3	$s1 = $s2 – $s3	Three register operands
	add immediate	addi $s1,$s2,20	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	lw $s1,20($s2)	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	sw $s1,20($s2)	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	lh $s1,20($s2)	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	lhu $s1,20($s2)	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	sh $s1,20($s2)	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	lb $s1,20($s2)	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	lbu $s1,20($s2)	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	sb $s1,20($s2)	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	ll $s1,20($s2)	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	sc $s1,20($s2)	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	lui $s1,20	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	and $s1,$s2,$s3	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	or $s1,$s2,$s3	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	nor $s1,$s2,$s3	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	andi $s1,$s2,20	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	ori $s1,$s2,20	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	sll $s1,$s2,10	$s1 = $s2 << 10	Shift left by constant
	shift right logical	srl $s1,$s2,10	$s1 = $s2 >> 10	Shift right by constant

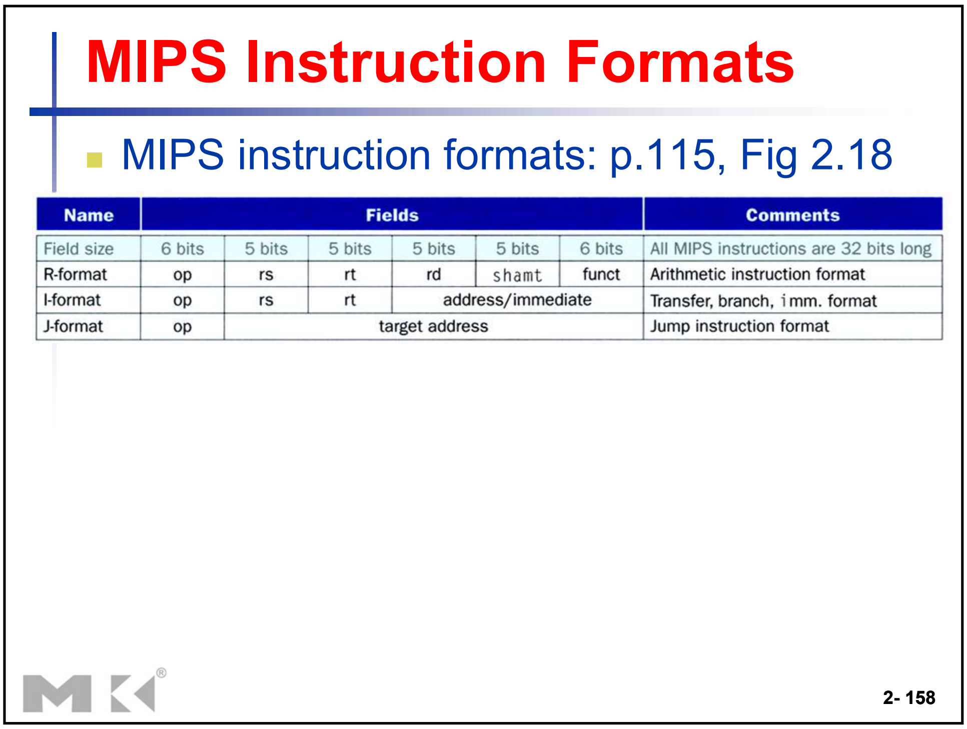


MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Conditional branch	branch on equal	`beq $s1,$s2,25`	if (`$s1` == `$s2`) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1,$s2,25`	if (`$s1`!= `$s2`) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1,$s2,$s3`	if (`$s2` < `$s3`) `$s1` = 1; else `$s1` = 0	Compare less than; for `beq`, `bne`
	set on less than unsigned	`sltu $s1,$s2,$s3`	if (`$s2` < `$s3`) `$s1` = 1; else `$s1` = 0	Compare less than unsigned
	set less than immediate	`slti $s1,$s2,20`	if (`$s2` < 20) `$s1` = 1; else `$s1` = 0	Compare less than constant
	set less than immediate unsigned	`sltiu $s1,$s2,20`	if (`$s2` < 20) `$s1` = 1; else `$s1` = 0	Compare less than constant unsigned
Unconditional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to `$ra`	For switch, procedure return
	jump and link	`jal 2500`	`$ra` = PC + 4; go to 10000	For procedure call

2- 157

157



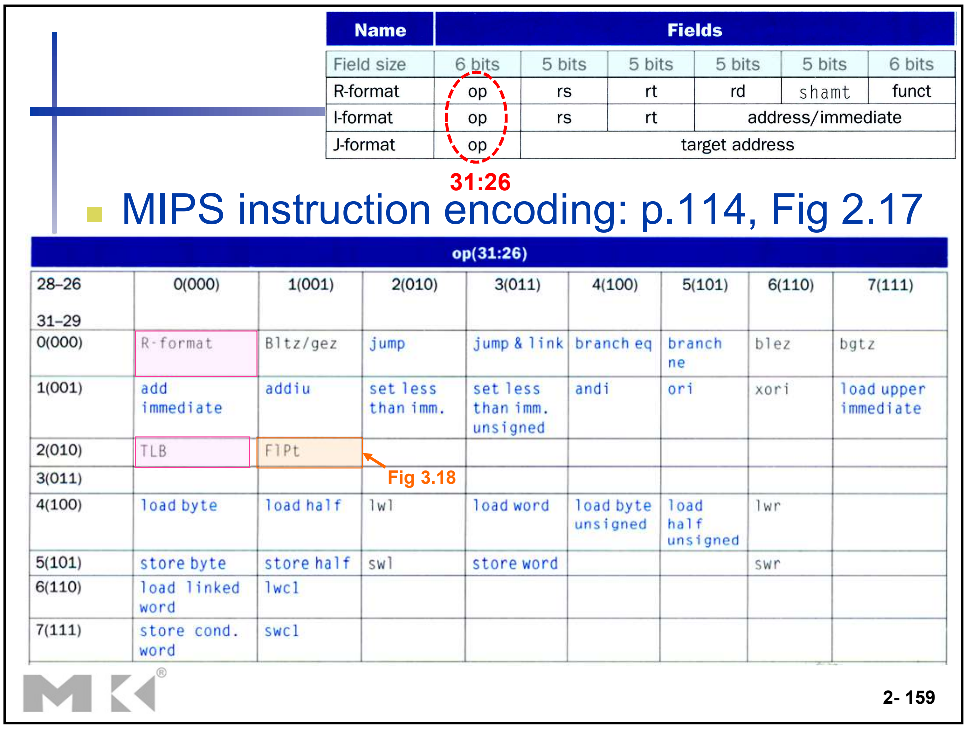
MIPS Instruction Formats

- MIPS instruction formats: p.115, Fig 2.18

Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	op	rs	rt	rd	`shamt`	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, `imm`. format
J-format	op	target address					Jump instruction format

2- 158

158



Name	Fields					
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
R-format	op	rs	rt	rd	shamt	funct
I-format	op	rs	rt	address/immediate		
J-format	op	target address				

31:26

- MIPS instruction encoding: p.114, Fig 2.17

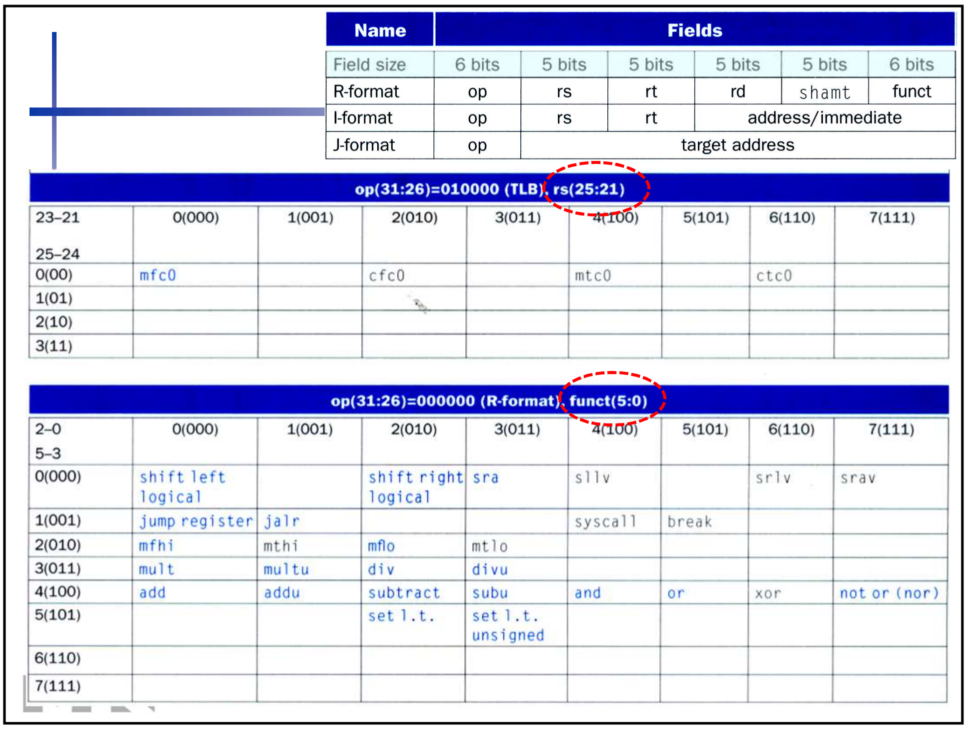
op(31:26)

28–26 31–29	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	R-format	Bltz/gez	jump	jump & link	branch eq	branch ne	blez	bgtz
1(001)	add immediate	addiu	set less than imm.	set less than imm. unsigned	andi	ori	xori	load upper immediate
2(010)	TLB	FlPt						
3(011)								
4(100)	load byte	load half	lwl	load word	load byte unsigned	load half unsigned	lwr	
5(101)	store byte	store half	swl	store word			swr	
6(110)	load linked word	lwcl						
7(111)	store cond. word	swcl						

Fig 3.18

2- 159

159



Name	Fields					
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
R-format	op	rs	rt	rd	shamt	funct
I-format	op	rs	rt	address/immediate		
J-format	op	target address				

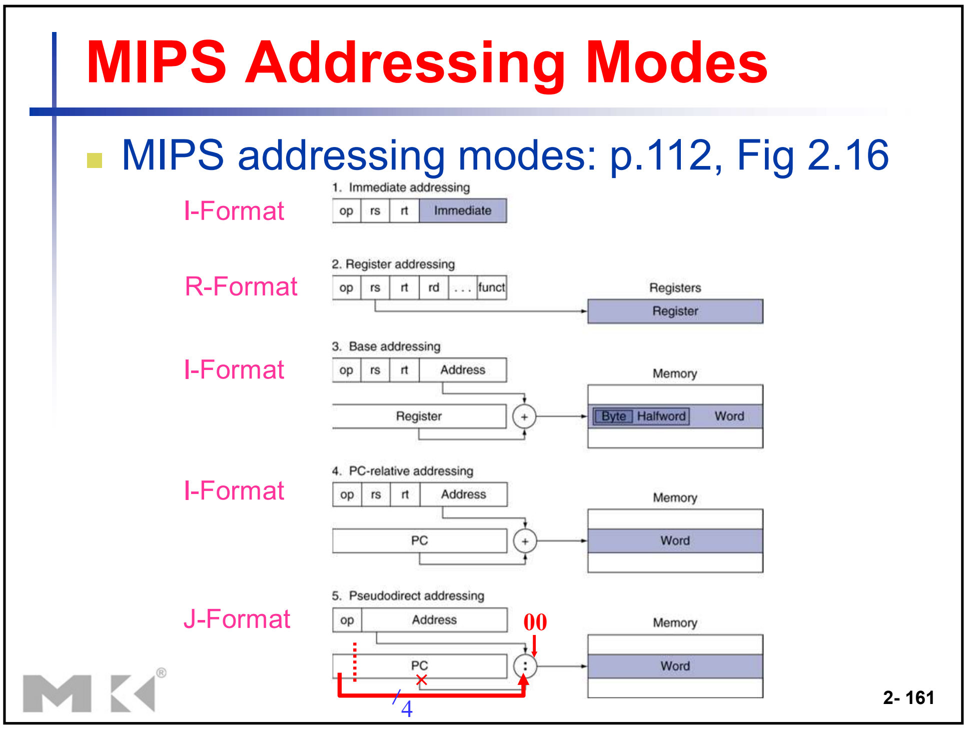
op(31:26)=010000 (TLB), rs(25:21)

23–21 25–24	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(00)	mfc0		cfc0		mtc0		ctc0	
1(01)								
2(10)								
3(11)								

op(31:26)=000000 (R-format), funct(5:0)

2–0 5–3	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	shift left logical		shift right logical	sra	sllv		srlv	srav
1(001)	jump register	jalr			syscall	break		
2(010)	mfhi	mthi	mflo	mtlo				
3(011)	mult	multu	div	divu				
4(100)	add	addu	subtract	subu	and	or	xor	not or (nor)
5(101)			set l.t.	set l.t. unsigned				
6(110)								
7(111)								

160

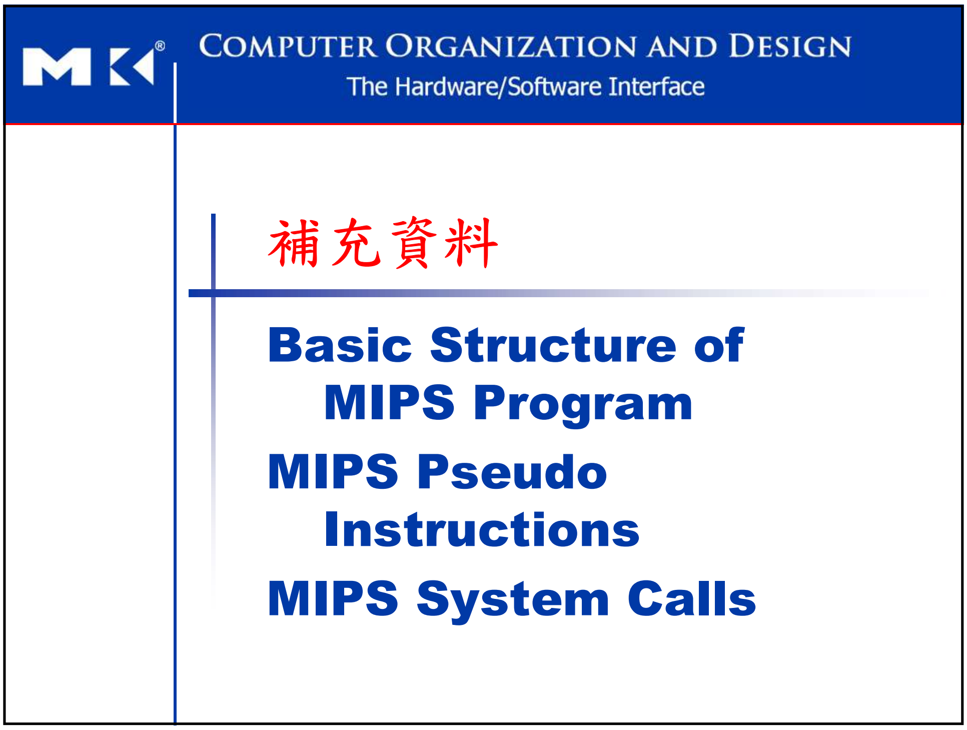


MIPS Addressing Modes

- MIPS addressing modes: p.112, Fig 2.16

2- 161

161



COMPUTER ORGANIZATION AND DESIGN
The Hardware/Software Interface

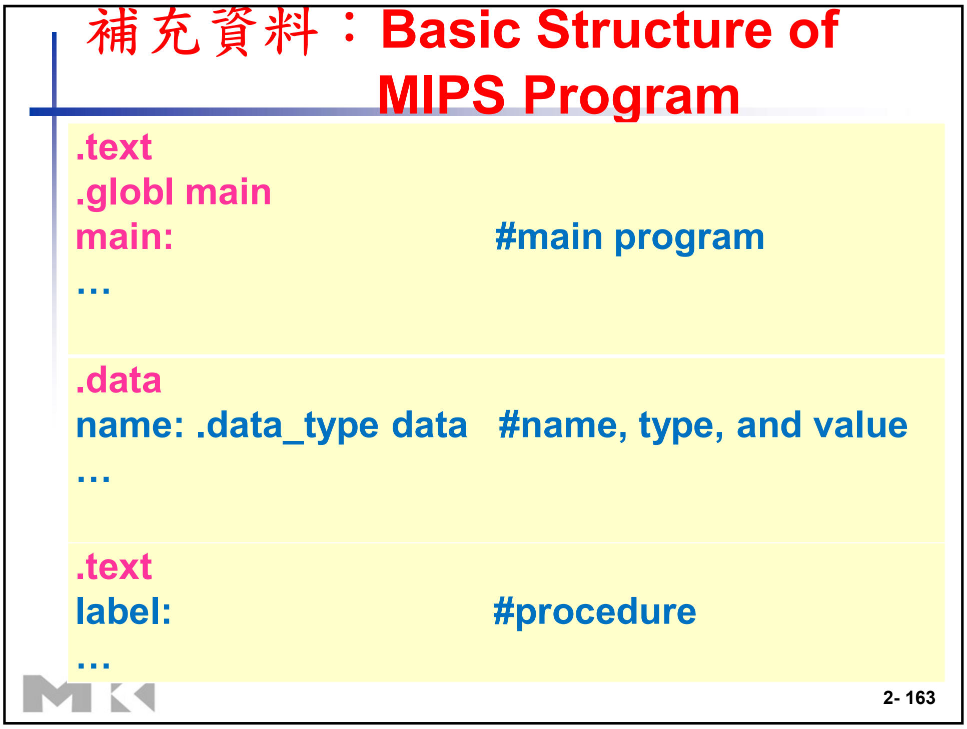
補充資料

Basic Structure of MIPS Program

MIPS Pseudo Instructions

MIPS System Calls

162

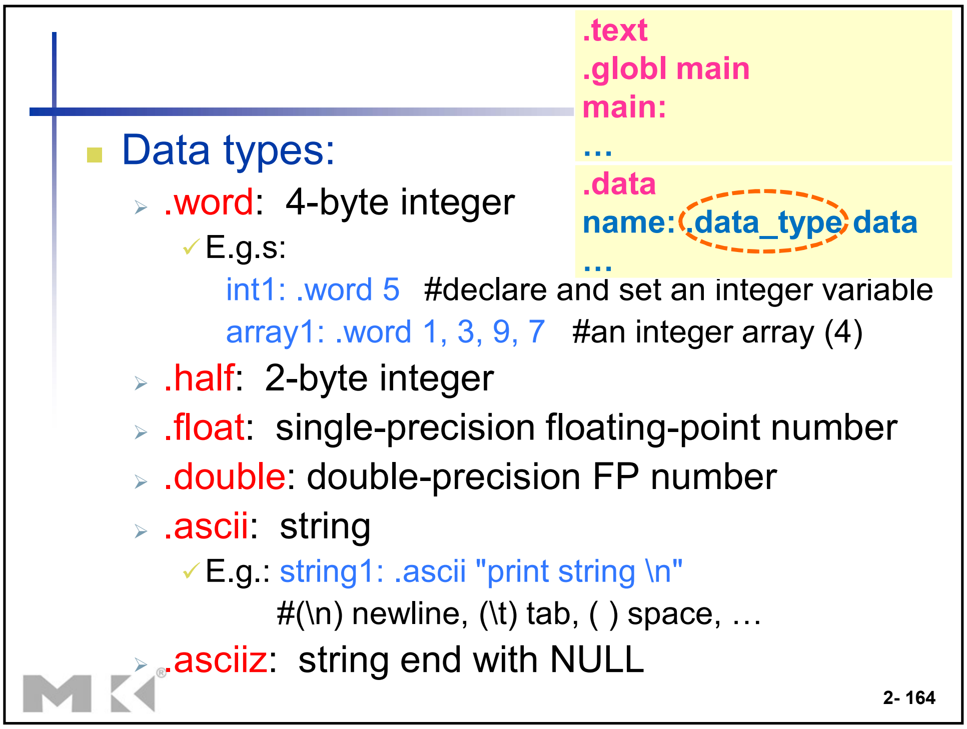


補充資料：Basic Structure of MIPS Program

```
.text
.globl main
main:                          #main program
…

.data
name: .data_type data   #name, type, and value
…

.text
label:                         #procedure
…
```

2- 163

163

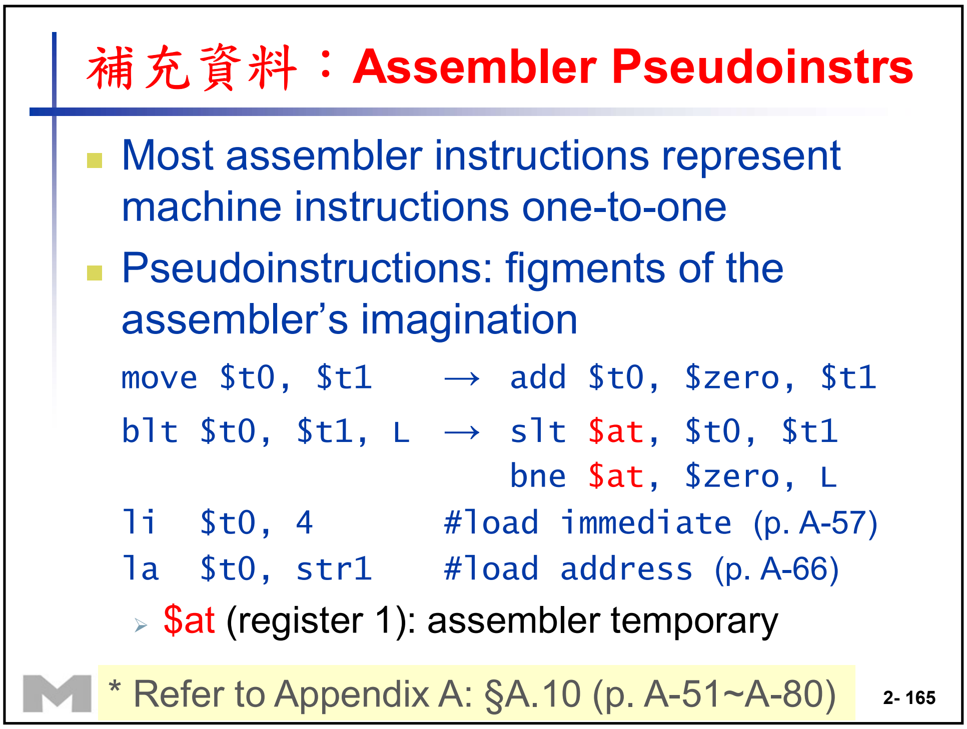


```
.text
.globl main
main:
…
.data
name: .data_type data
…
```

- Data types:
  - .word: 4-byte integer
    - E.g.s:
      int1: .word 5   #declare and set an integer variable
      array1: .word 1, 3, 9, 7   #an integer array (4)
  - .half: 2-byte integer
  - .float: single-precision floating-point number
  - .double: double-precision FP number
  - .ascii: string
    - E.g.: string1: .ascii "print string \n"
      #(\n) newline, (\t) tab, ( ) space, …
  - .asciiz: string end with NULL

2- 164

164



補充資料：Assembler Pseudoinstrs

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

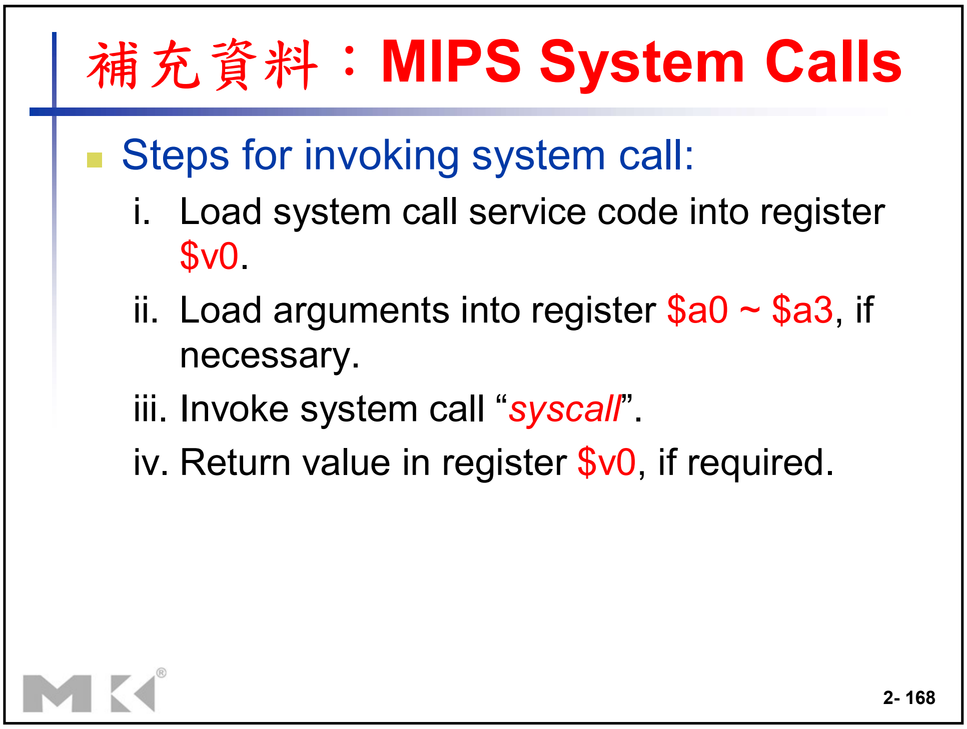
```
move $t0, $t1     →  add $t0, $zero, $t1
blt $t0, $t1, L   →  slt $at, $t0, $t1
                     bne $at, $zero, L
li  $t0, 4          #load immediate (p. A-57)
la  $t0, str1       #load address (p. A-66)
```

  - $at (register 1): assembler temporary

* Refer to Appendix A: §A.10 (p. A-51~A-80)

2- 165

165

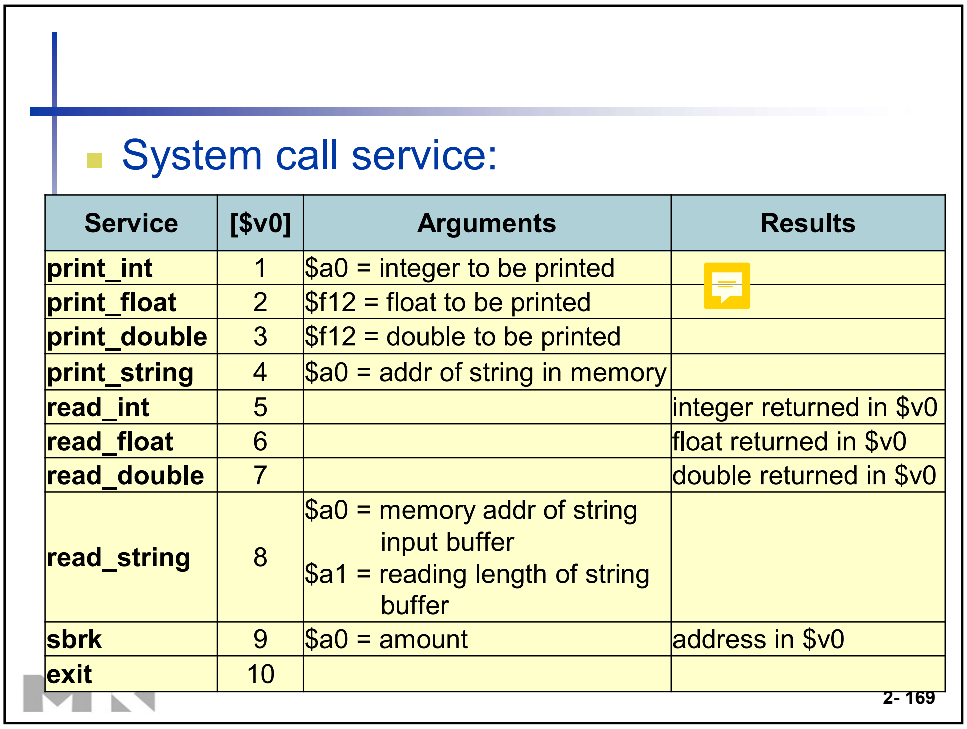


補充資料：MIPS System Calls

- Steps for invoking system call:
  - i. Load system call service code into register $v0.
  - ii. Load arguments into register $a0 ~ $a3, if necessary.
  - iii. Invoke system call "*syscall*".
  - iv. Return value in register $v0, if required.

2- 168

168



- System call service:

Service	[$v0]	Arguments	Results
print_int	1	$a0 = integer to be printed	
print_float	2	$f12 = float to be printed	
print_double	3	$f12 = double to be printed	
print_string	4	$a0 = addr of string in memory	
read_int	5		integer returned in $v0
read_float	6		float returned in $v0
read_double	7		double returned in $v0
read_string	8	$a0 = memory addr of string input buffer $a1 = reading length of string buffer	
sbrk	9	$a0 = amount	address in $v0
exit	10		

2- 169

169