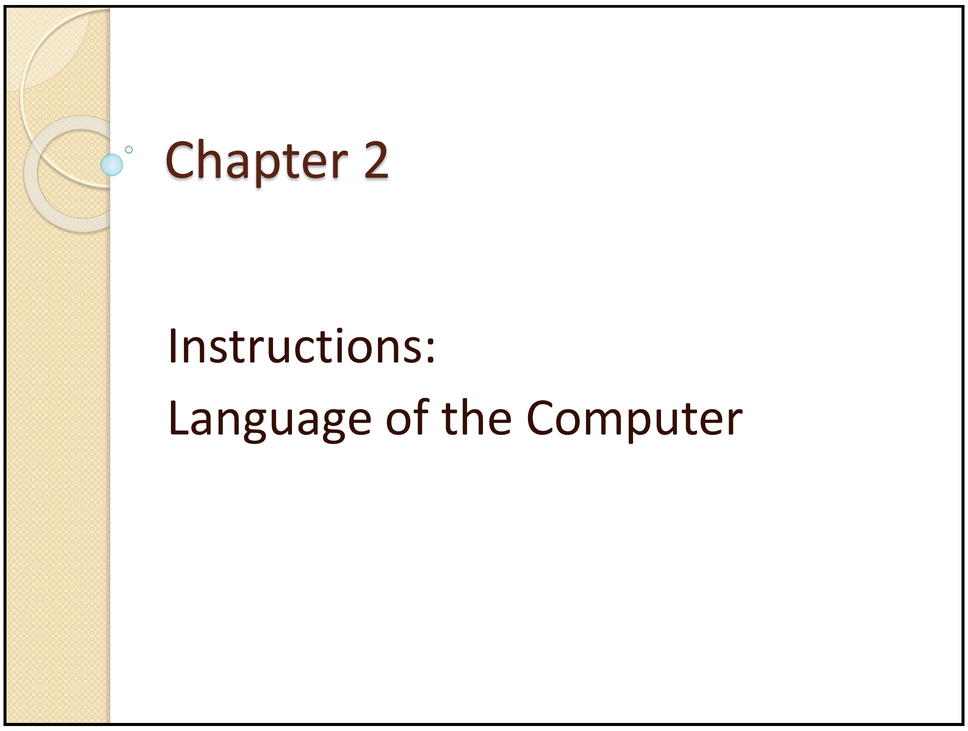
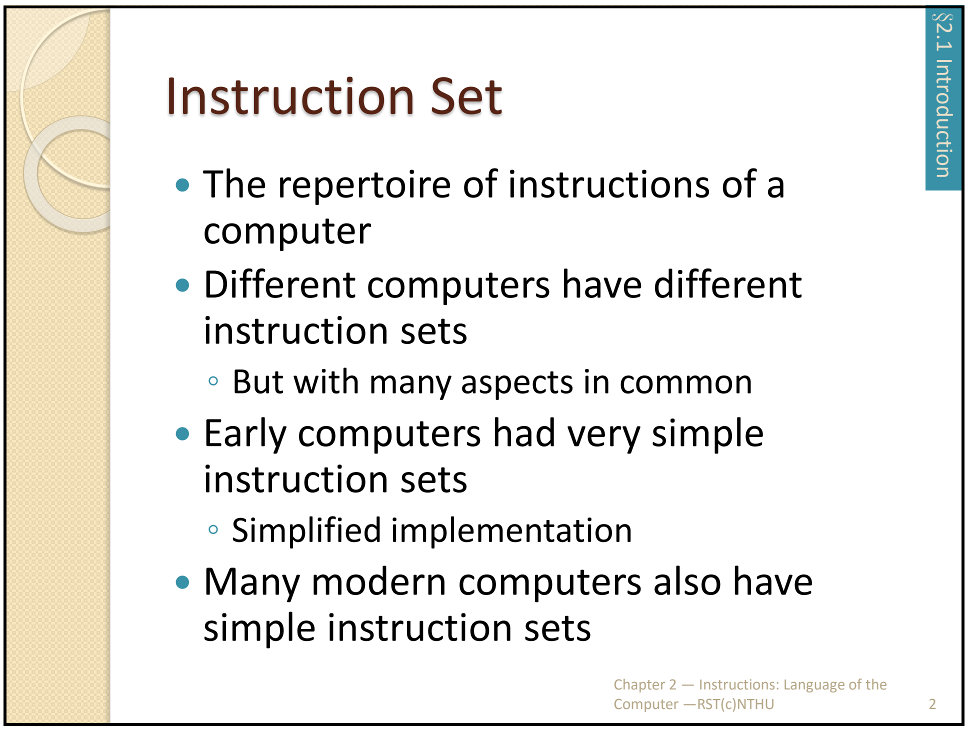


Chapter 2

Instructions:

Language of the Computer

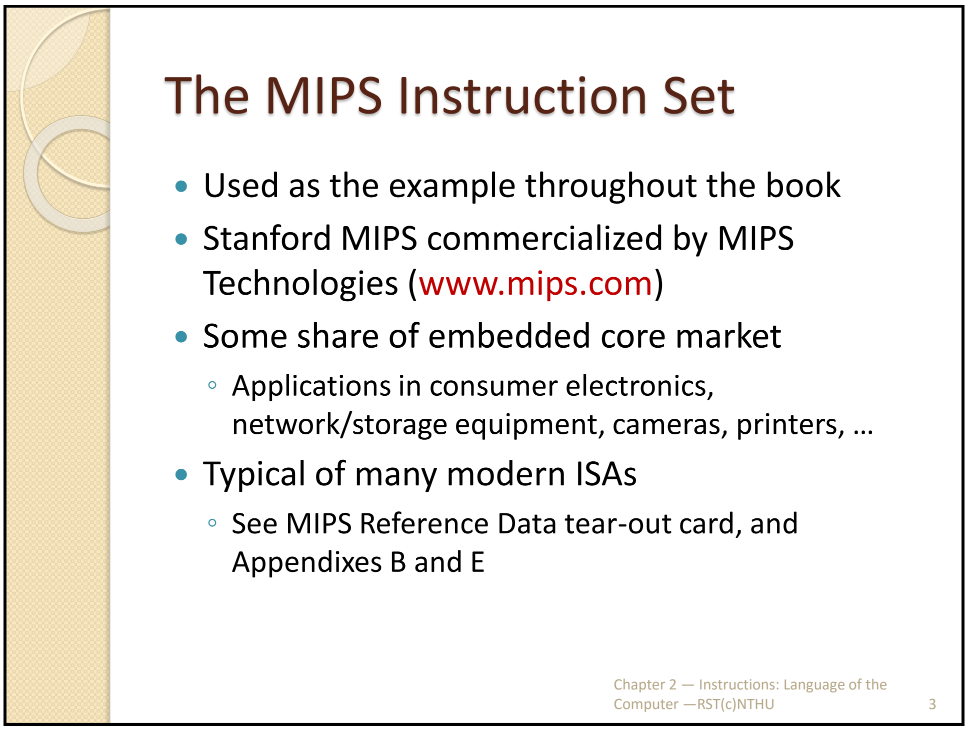


§2.1 Introduction

Instruction Set

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

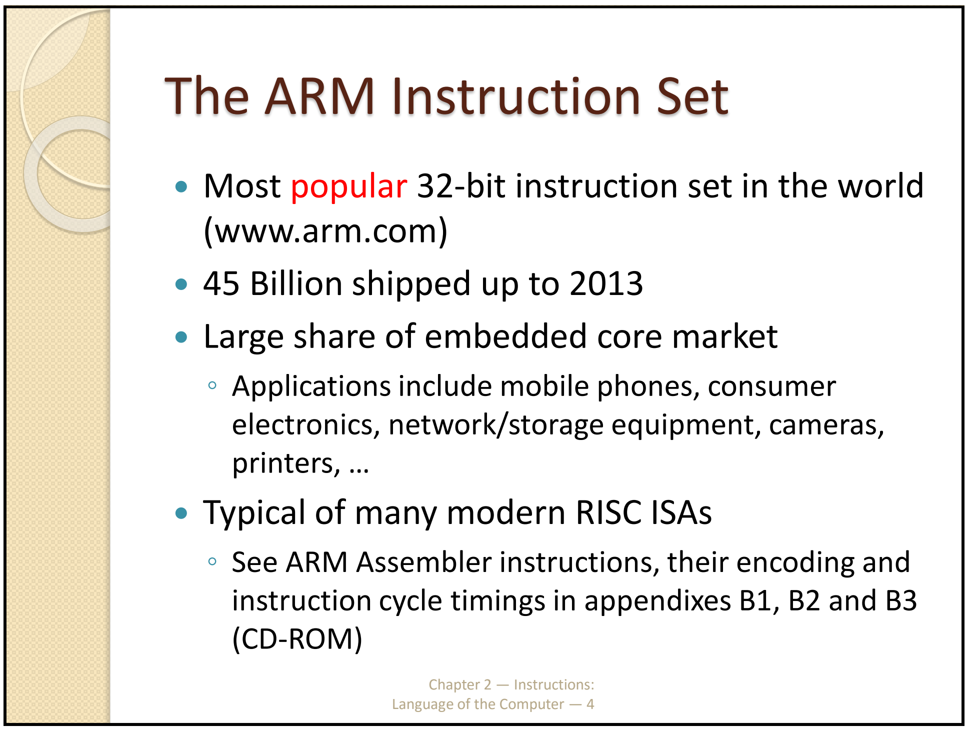
Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 2



The MIPS Instruction Set

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

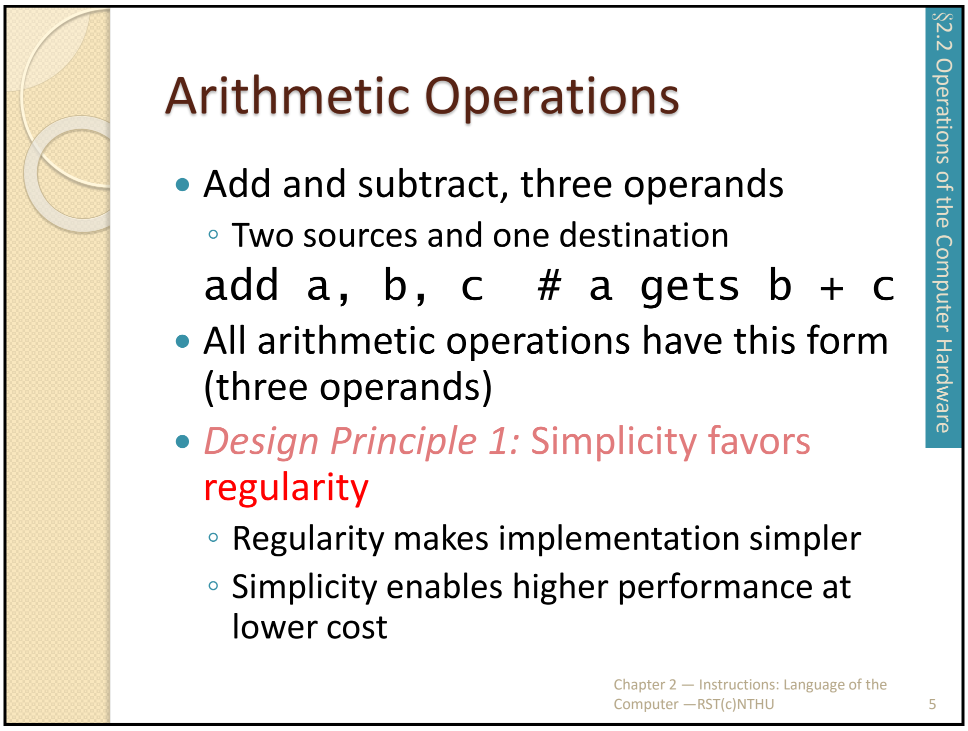
Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 3



The ARM Instruction Set

- Most popular 32-bit instruction set in the world (www.arm.com)
- 45 Billion shipped up to 2013
- Large share of embedded core market
  - Applications include mobile phones, consumer electronics, network/storage equipment, cameras, printers, …
- Typical of many modern RISC ISAs
  - See ARM Assembler instructions, their encoding and instruction cycle timings in appendixes B1, B2 and B3 (CD-ROM)

Chapter 2 — Instructions: Language of the Computer — 4



§2.2 Operations of the Computer Hardware

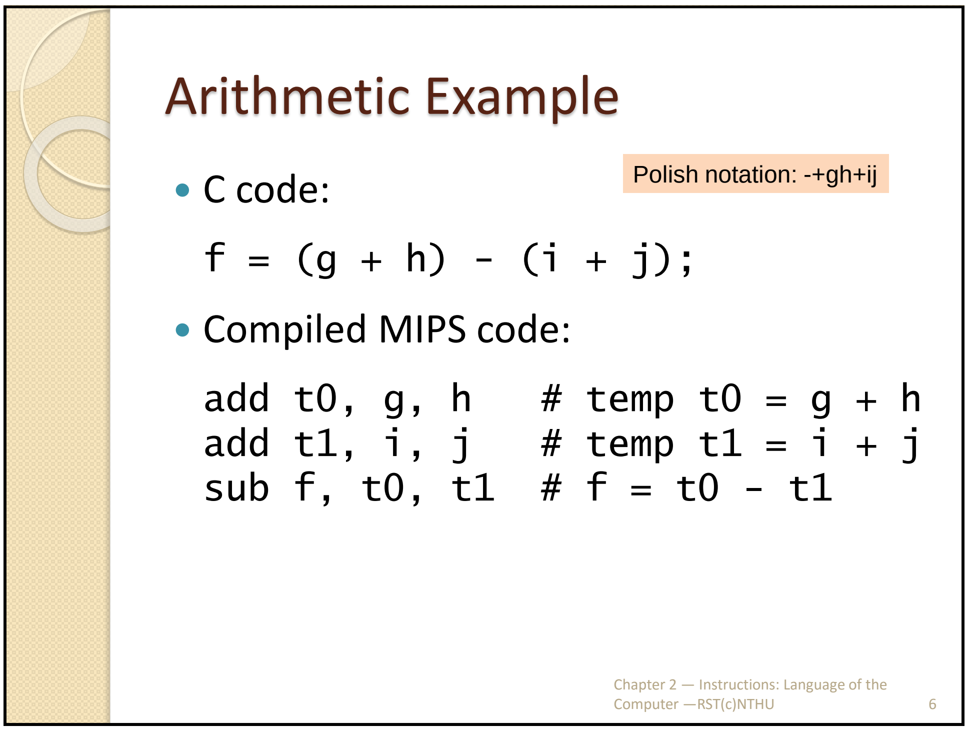
Arithmetic Operations

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

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

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

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 5



Arithmetic Example

Polish notation: -+gh+ij

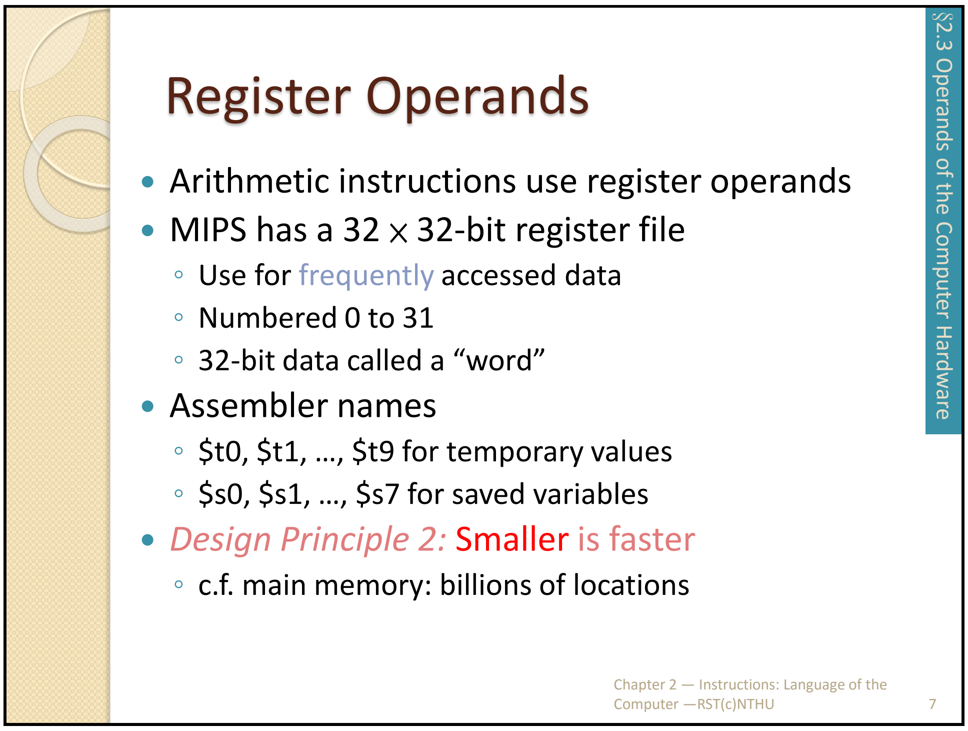
- C code:

```
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
```

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 6

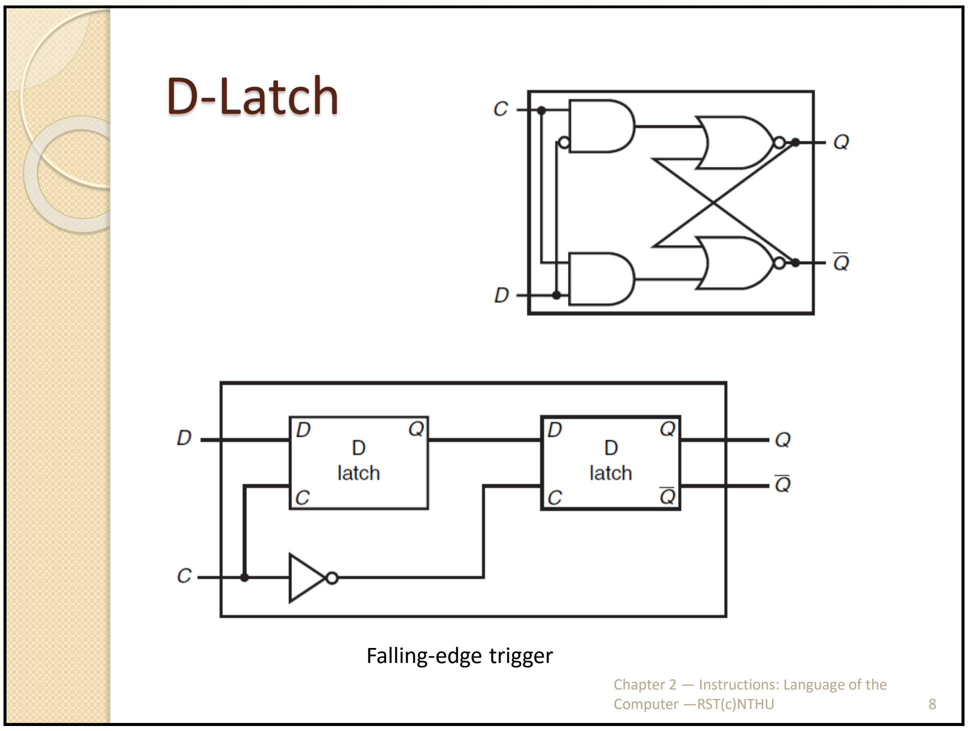


Register Operands

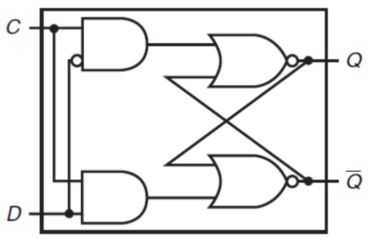
- Arithmetic instructions use register operands
- MIPS has a 32 × 32-bit register file
  - Use for frequently accessed data
  - Numbered 0 to 31
  - 32-bit data called a "word"
- Assembler names
  - $t0, $t1, …, $t9 for temporary values
  - $s0, $s1, …, $s7 for saved variables
- *Design Principle 2:*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

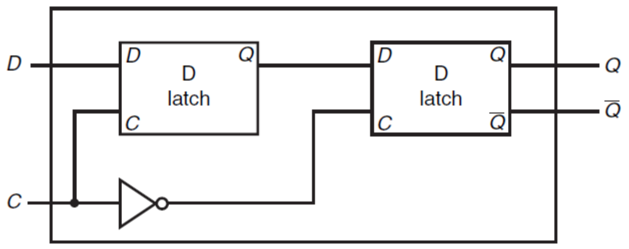
§2.3 Operands of the Computer Hardware

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 7



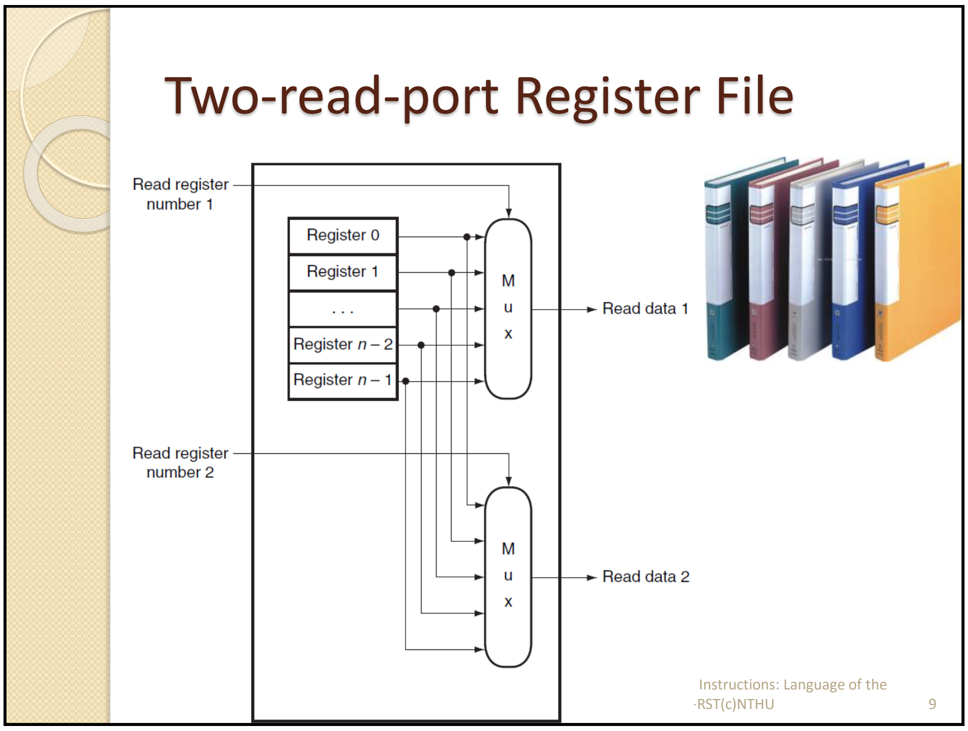
D-Latch



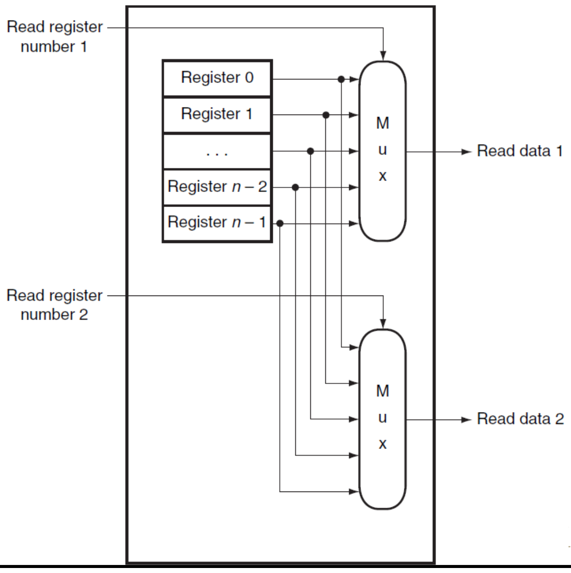


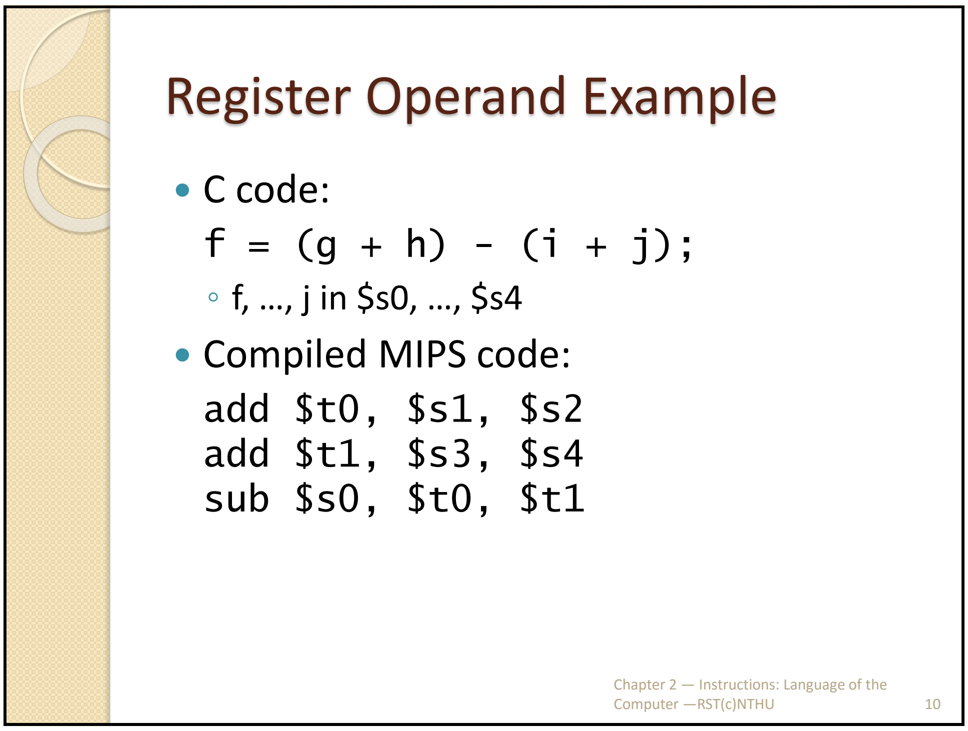
Falling-edge trigger

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 8



Two-read-port Register File





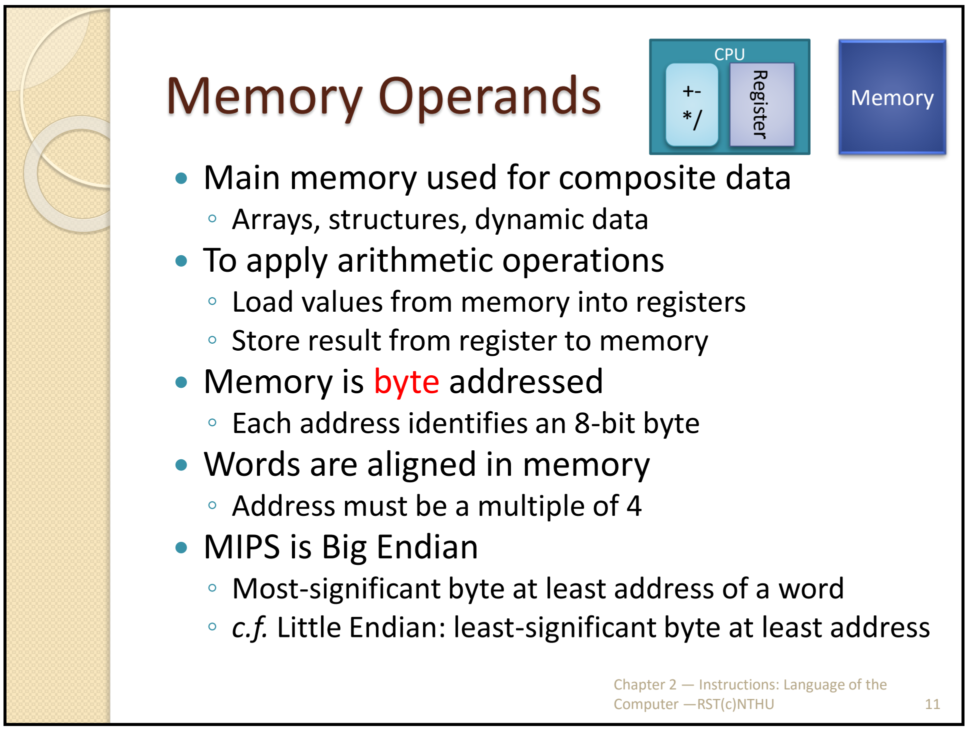
Register Operand Example

- C code:

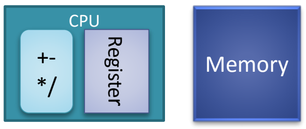
  `f = (g + h) - (i + j);`

  - f, …, j in $s0, …, $s4
- Compiled MIPS code:

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

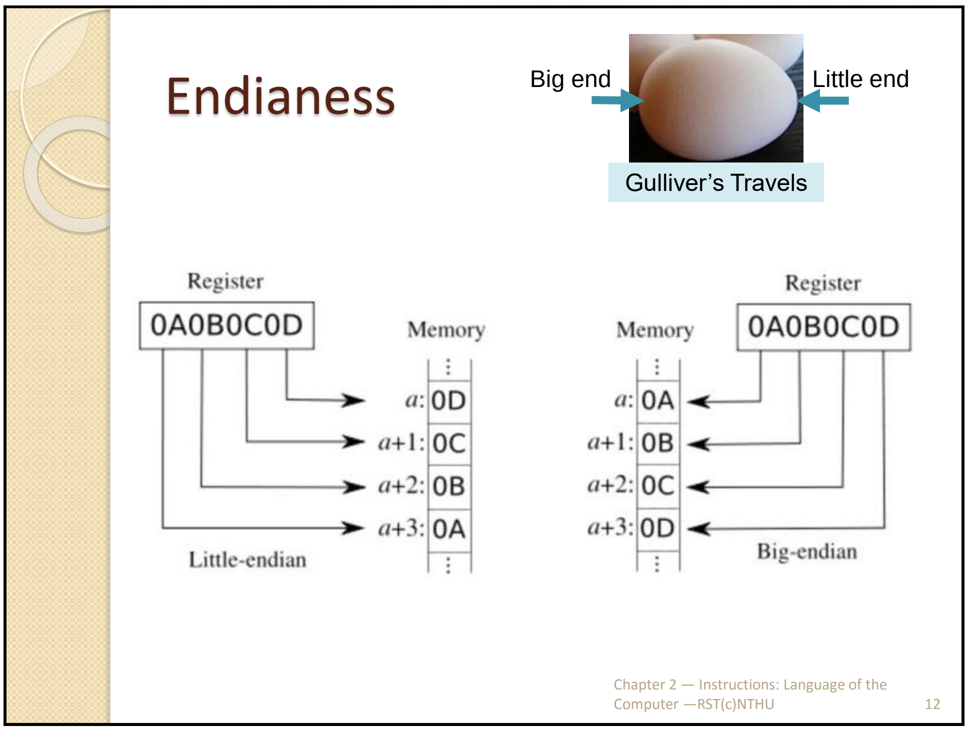


Memory Operands

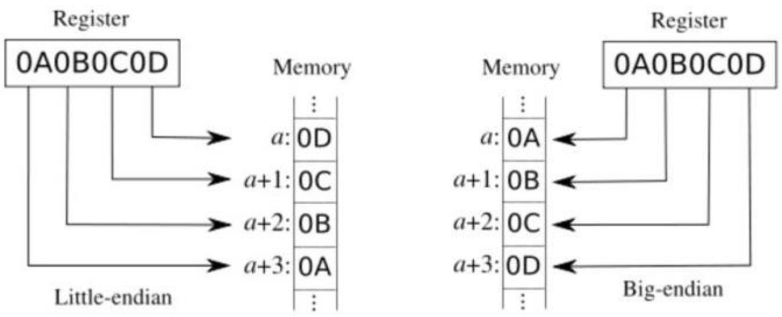


- Main memory used for composite data
  - Arrays, structures, dynamic data
- To apply arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory
- Memory is byte addressed
  - Each address identifies an 8-bit byte
- Words are aligned in memory
  - Address must be a multiple of 4
- MIPS is Big Endian
  - Most-significant byte at least address of a word
  - *c.f.* Little Endian: least-significant byte at least address

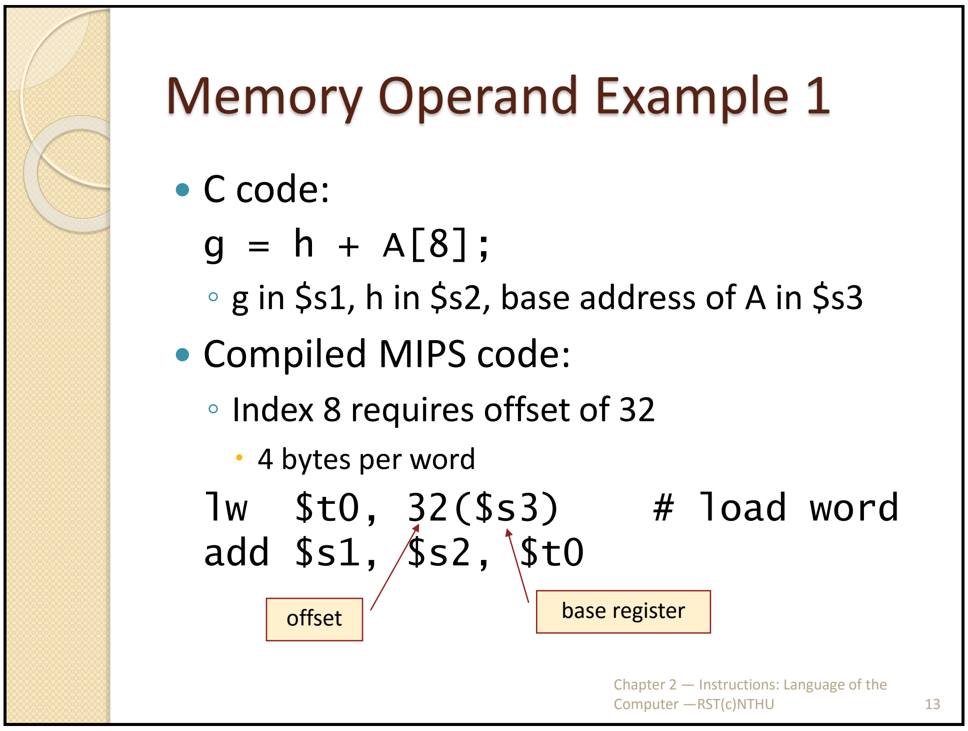
Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 11



Endianess



Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 12



Memory Operand Example 1

- C code:

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

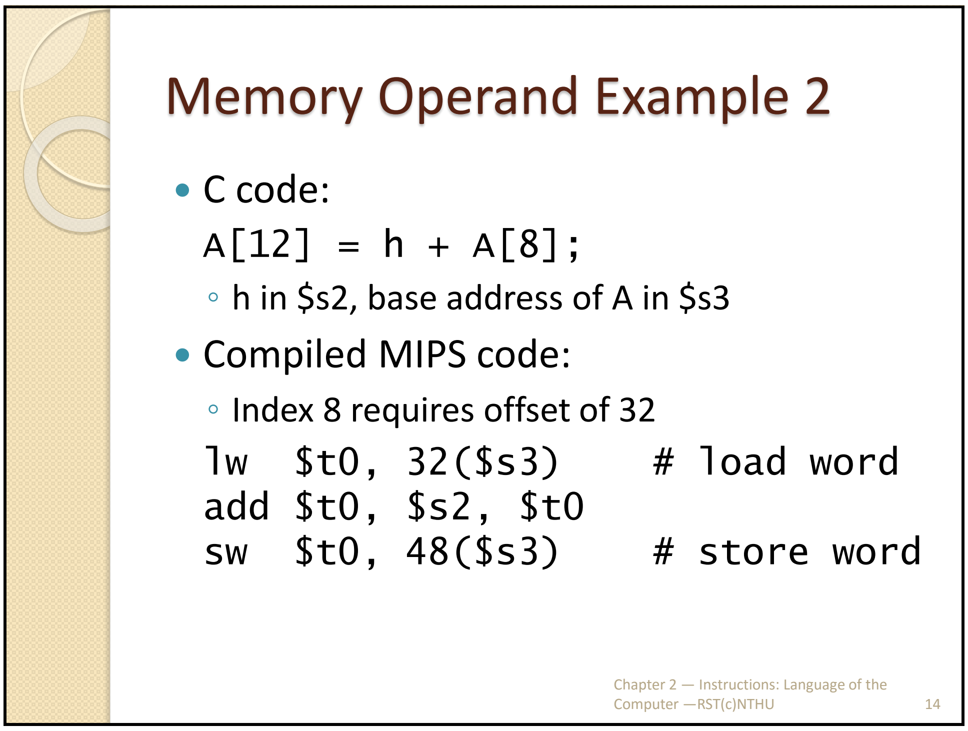
  - g in $s1, h in $s2, base address of A in $s3
- Compiled MIPS code:
  - Index 8 requires offset of 32
    - 4 bytes per word

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

offset

base register

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 13



Memory Operand Example 2

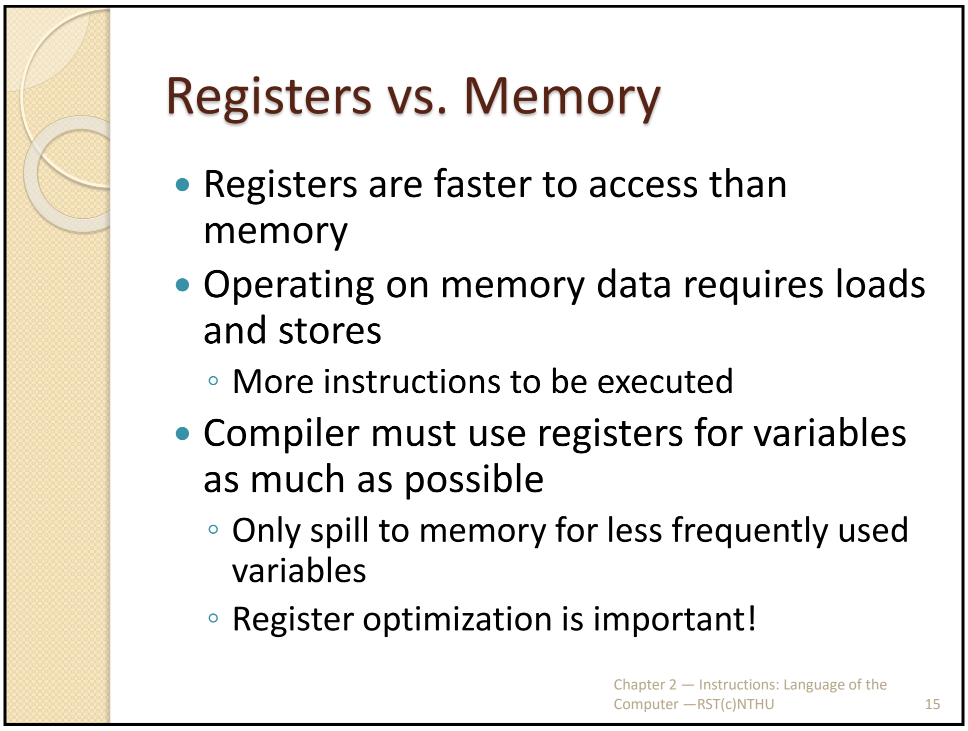
- C code:

```
A[12] = h + A[8];
```

  - h in $s2, base address of A in $s3
- Compiled MIPS code:
  - Index 8 requires offset of 32

```
lw  $t0, 32($s3)    # load word
add $t0, $s2, $t0
sw  $t0, 48($s3)    # store word
```

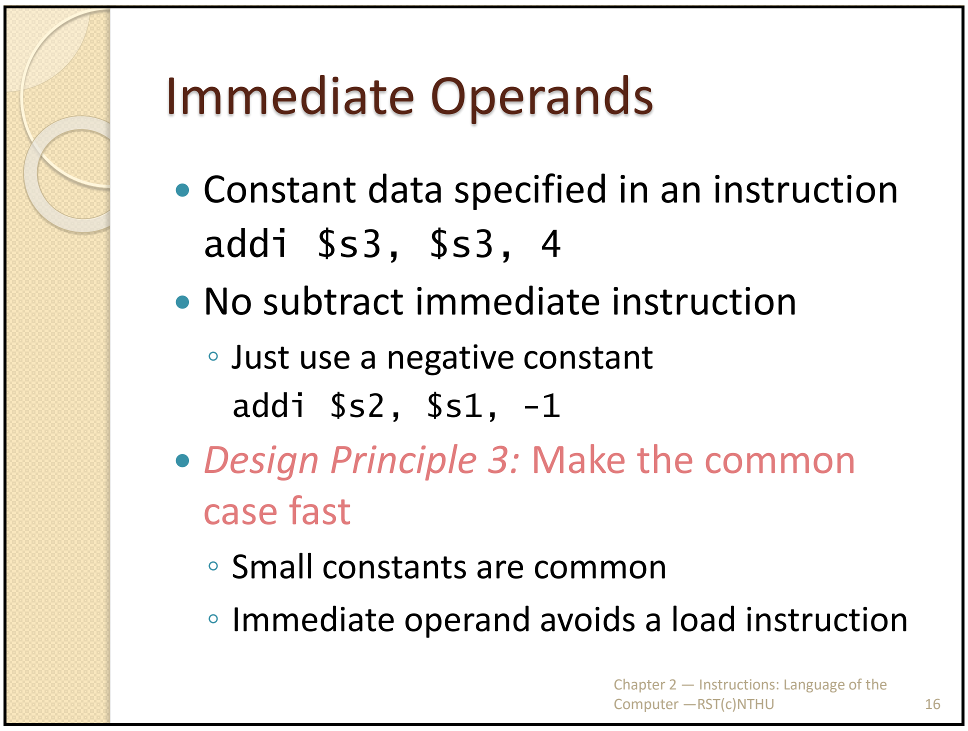
Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 14



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!

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 15



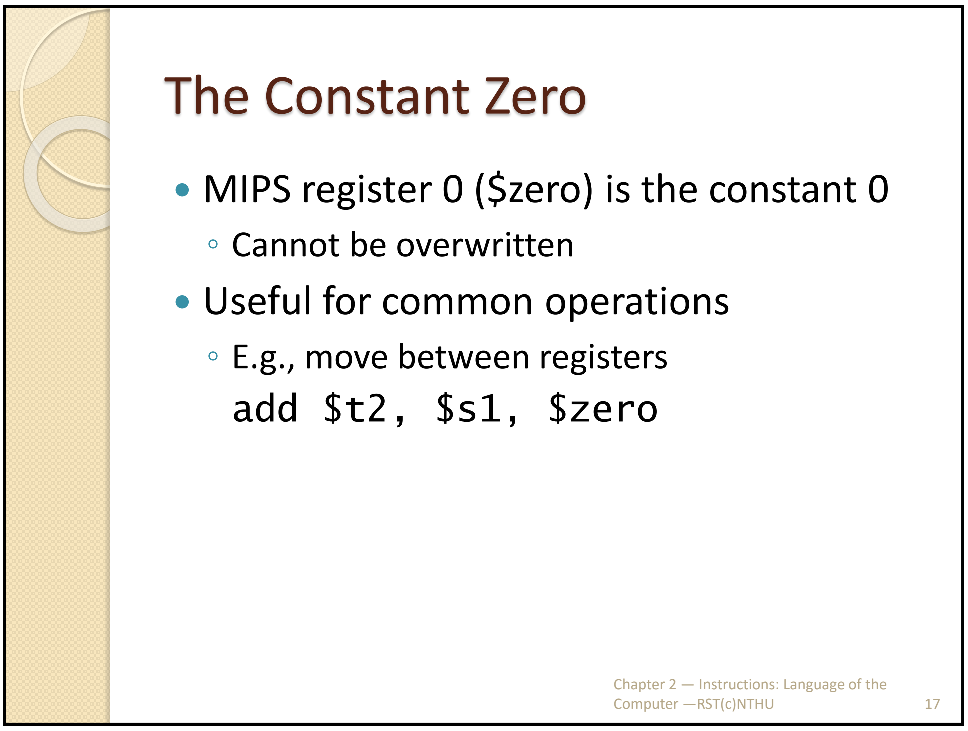
Immediate Operands

- Constant data specified in an instruction

  `addi $s3, $s3, 4`
- No subtract immediate instruction
  - Just use a negative constant

    `addi $s2, $s1, -1`
- *Design Principle 3:* Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 16

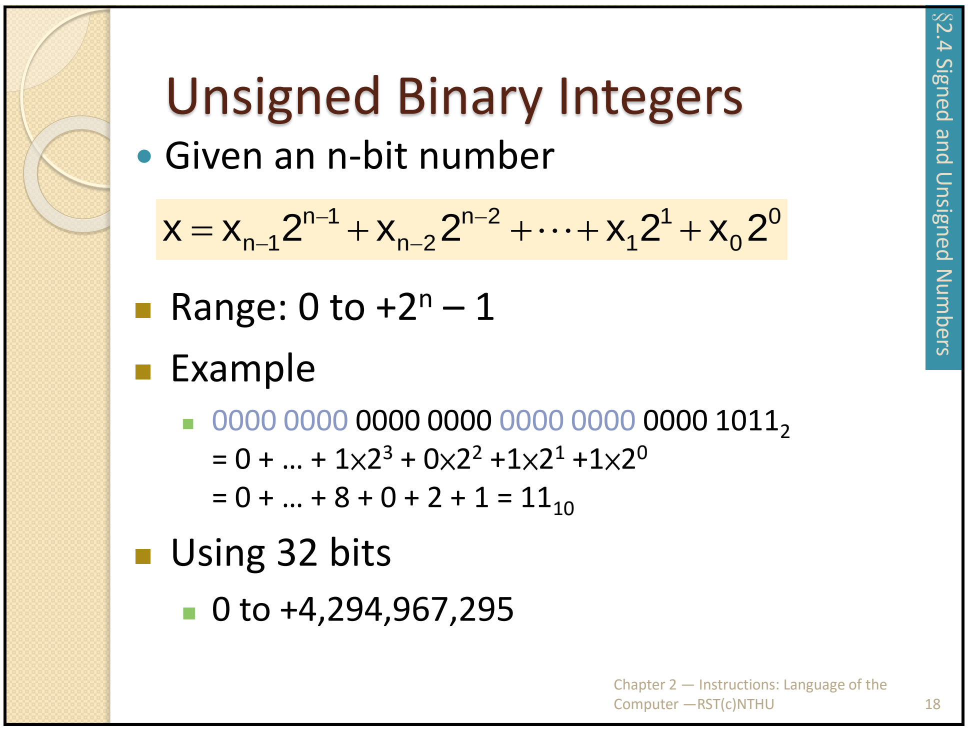


The Constant Zero

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

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

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 17



Unsigned Binary Integers

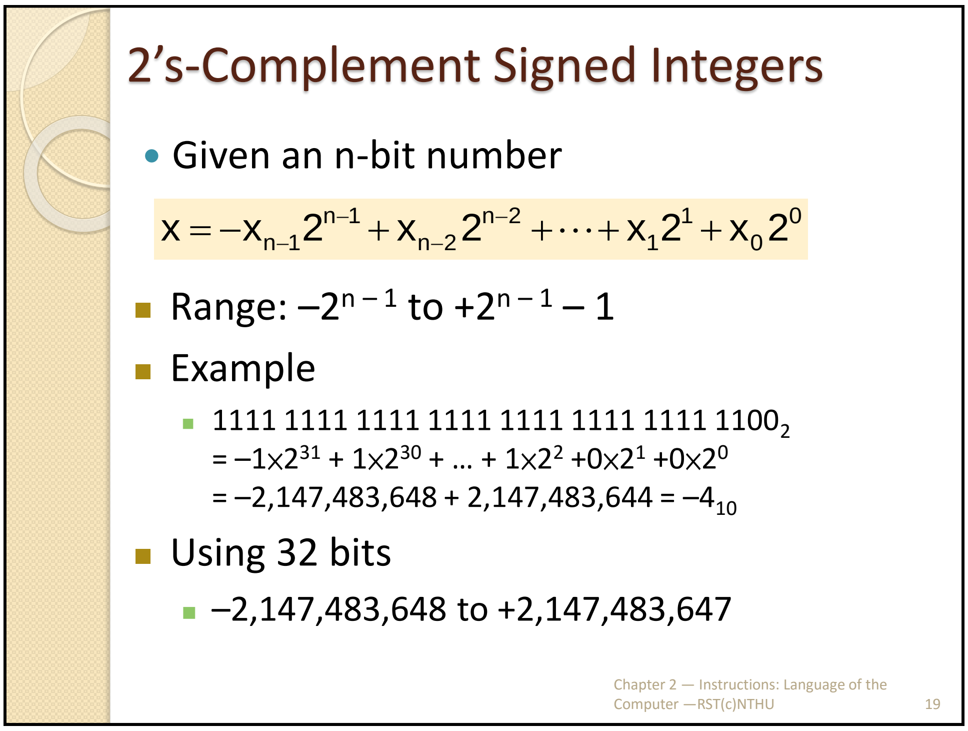
§2.4 Signed and Unsigned Numbers

- Given an n-bit number

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

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

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 18



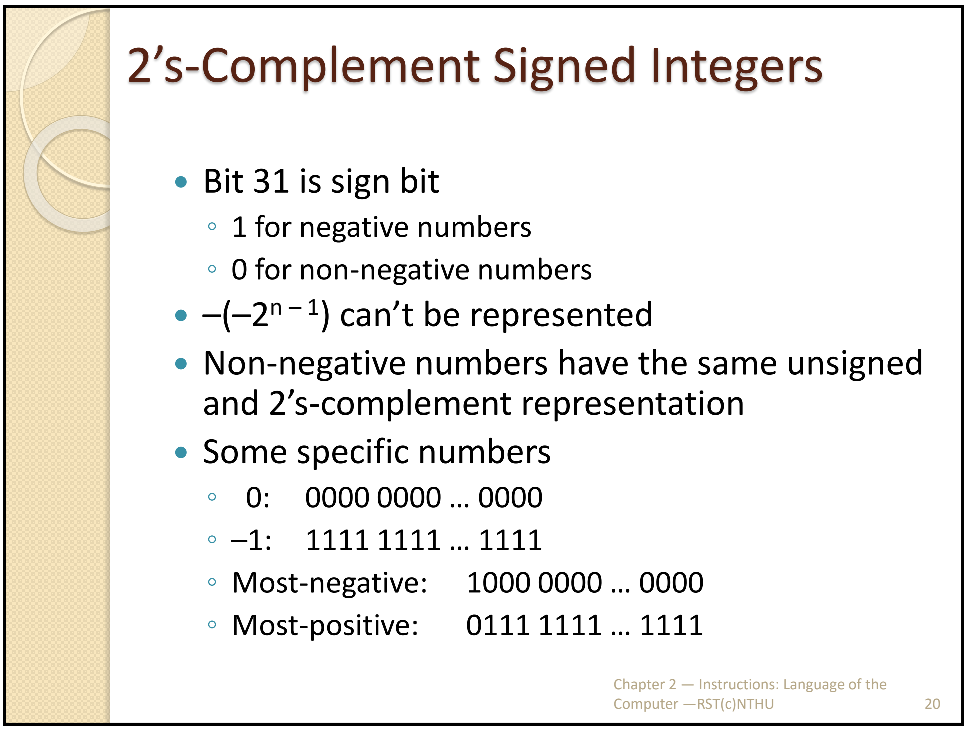
2’s-Complement Signed Integers

- Given an n-bit number

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

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

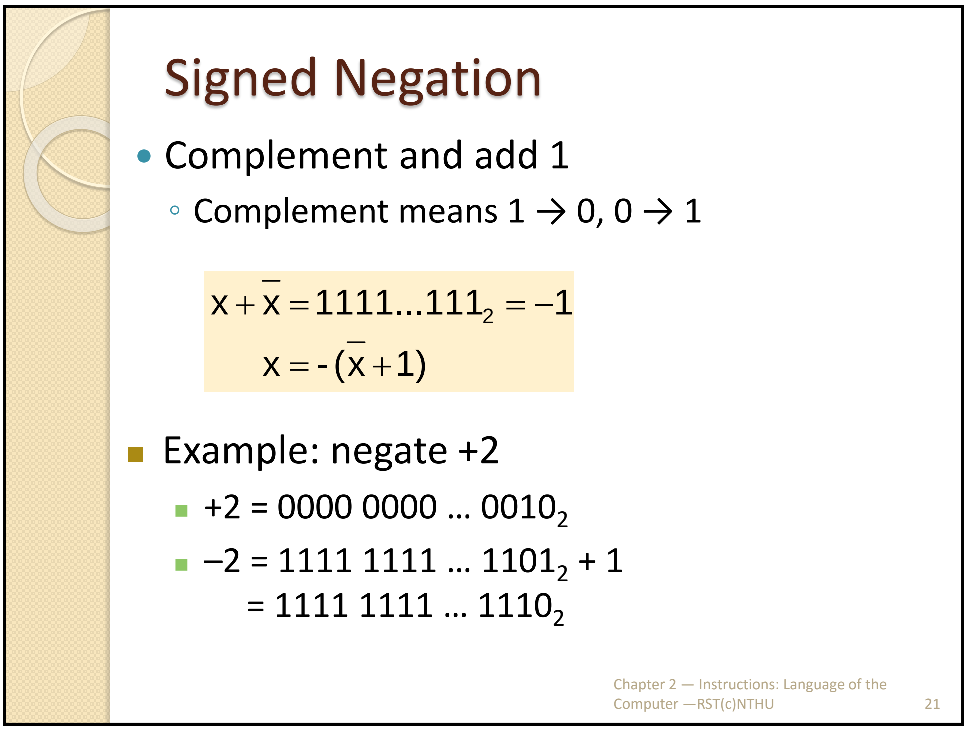
Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 19



2’s-Complement Signed Integers

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

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 20



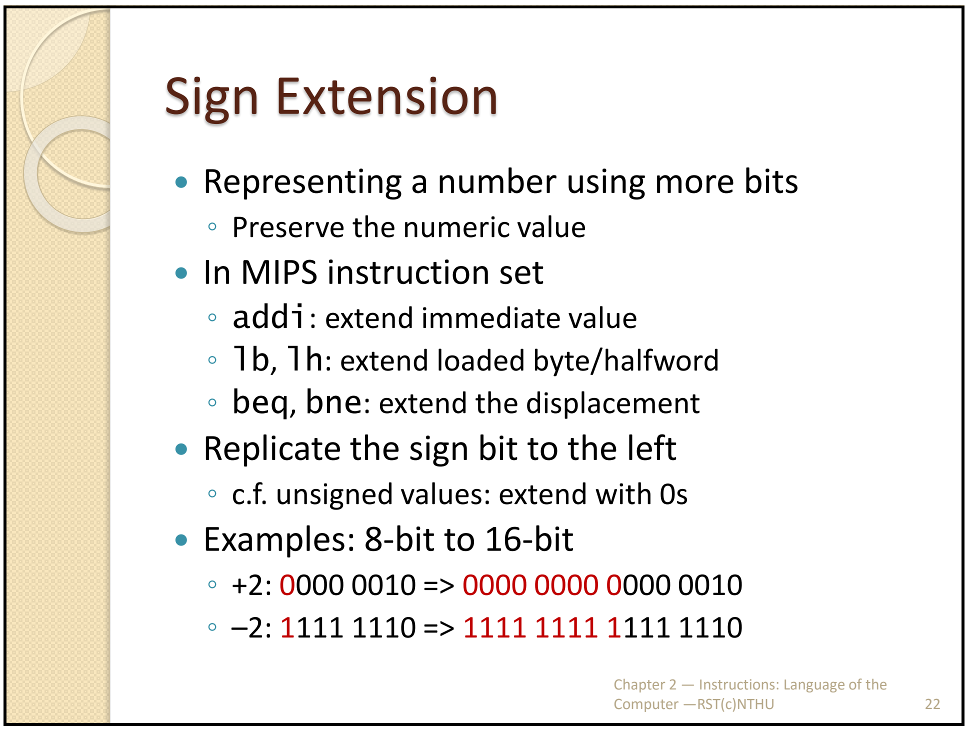
Signed Negation

- Complement and add 1
  - Complement means 1 → 0, 0 → 1

$$x + \overline{x} = 1111\ldots111_2 = -1$$
$$x = -(\overline{x} + 1)$$

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

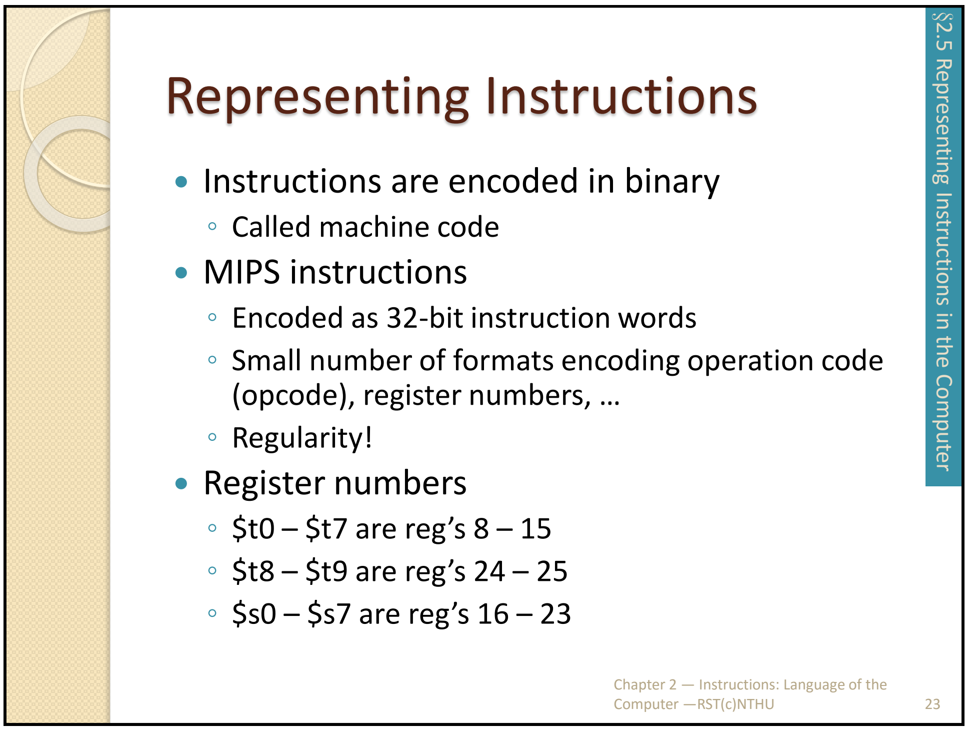
Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 21



Sign Extension

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

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 22

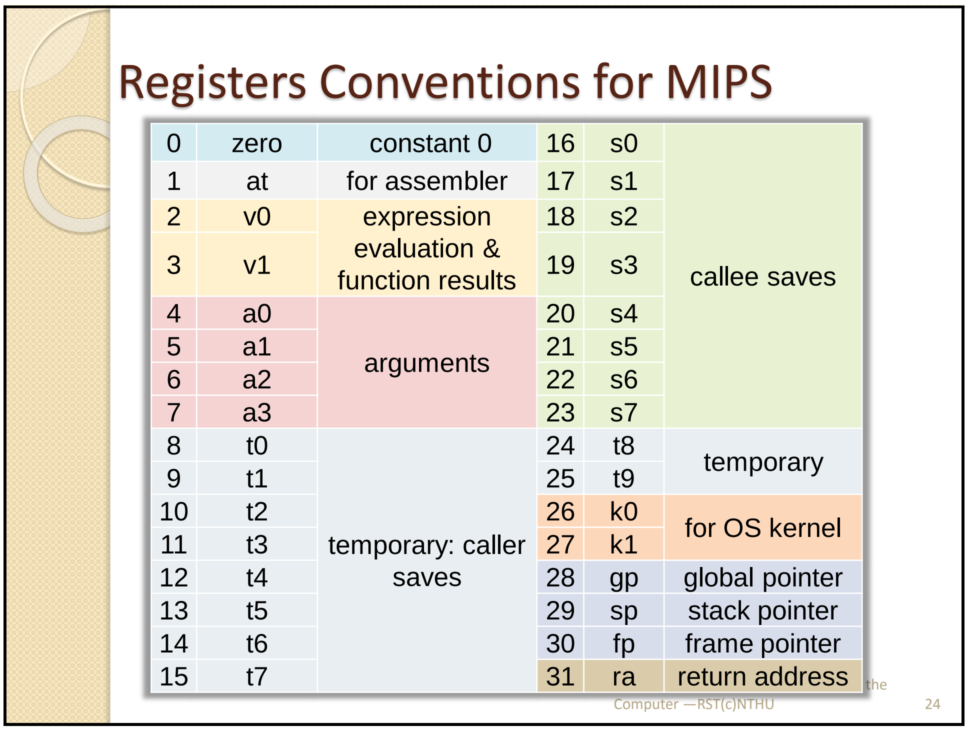


Representing Instructions

§2.5 Representing Instructions in the Computer

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

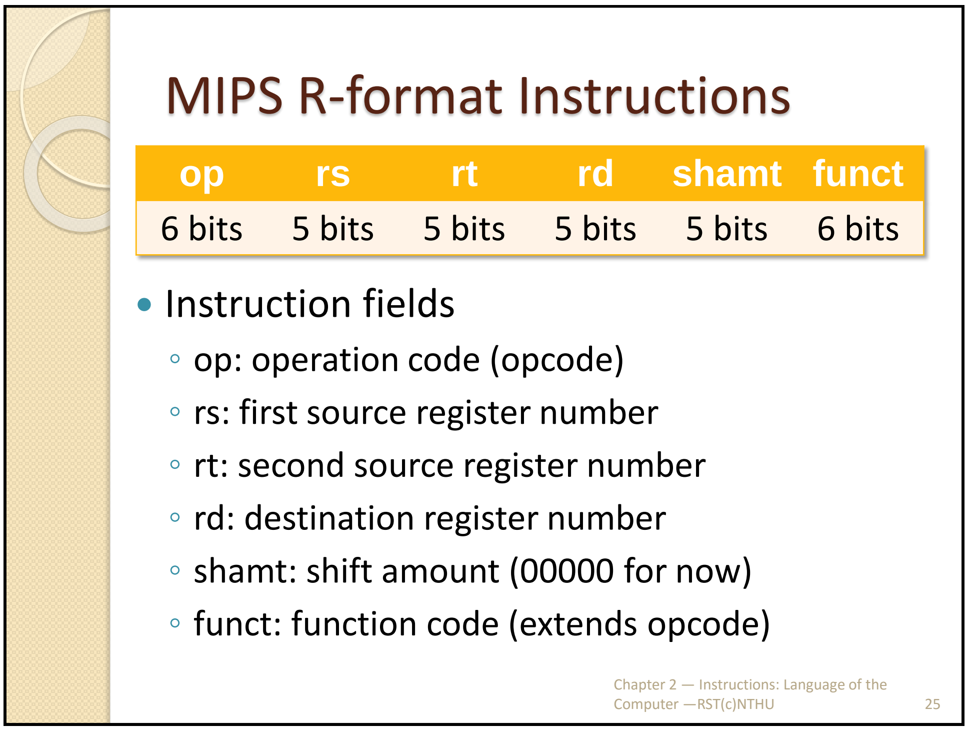
Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 23



Registers Conventions for MIPS

0	zero	constant 0	16	s0	callee saves
1	at	for assembler	17	s1	
2	v0	expression evaluation & function results	18	s2	
3	v1		19	s3	
4	a0	arguments	20	s4	
5	a1		21	s5	
6	a2		22	s6	
7	a3		23	s7	
8	t0	temporary: caller saves	24	t8	temporary
9	t1		25	t9	
10	t2		26	k0	for OS kernel
11	t3		27	k1	
12	t4		28	gp	global pointer
13	t5		29	sp	stack pointer
14	t6		30	fp	frame pointer
15	t7		31	ra	return address

the Computer —RST(c)NTHU 24

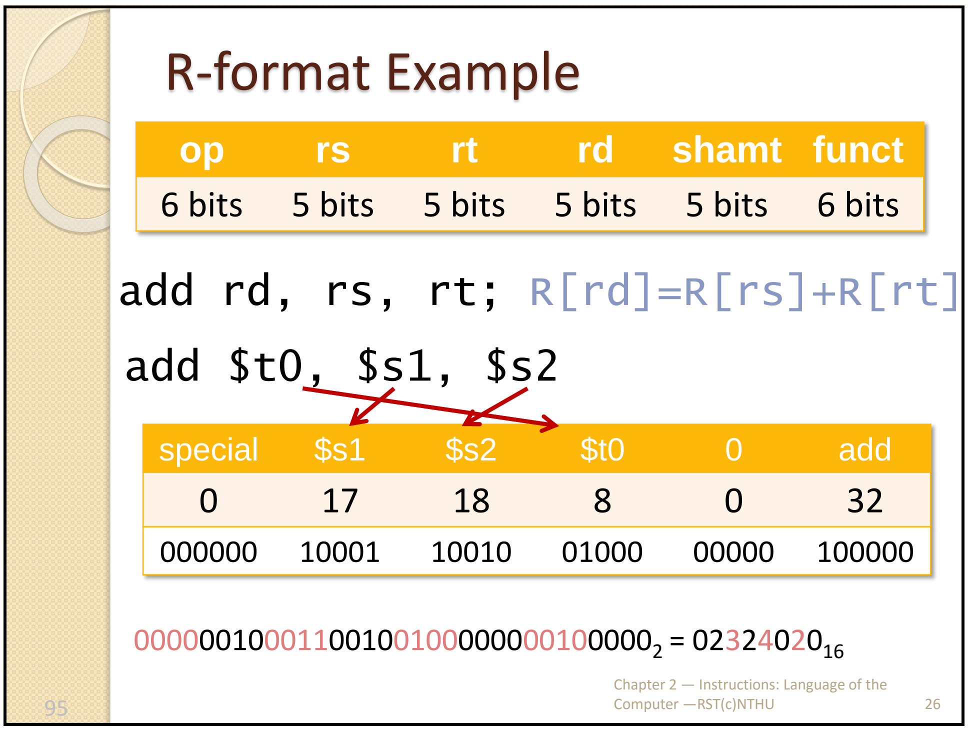


MIPS R-format Instructions

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)

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 25



R-format Example

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

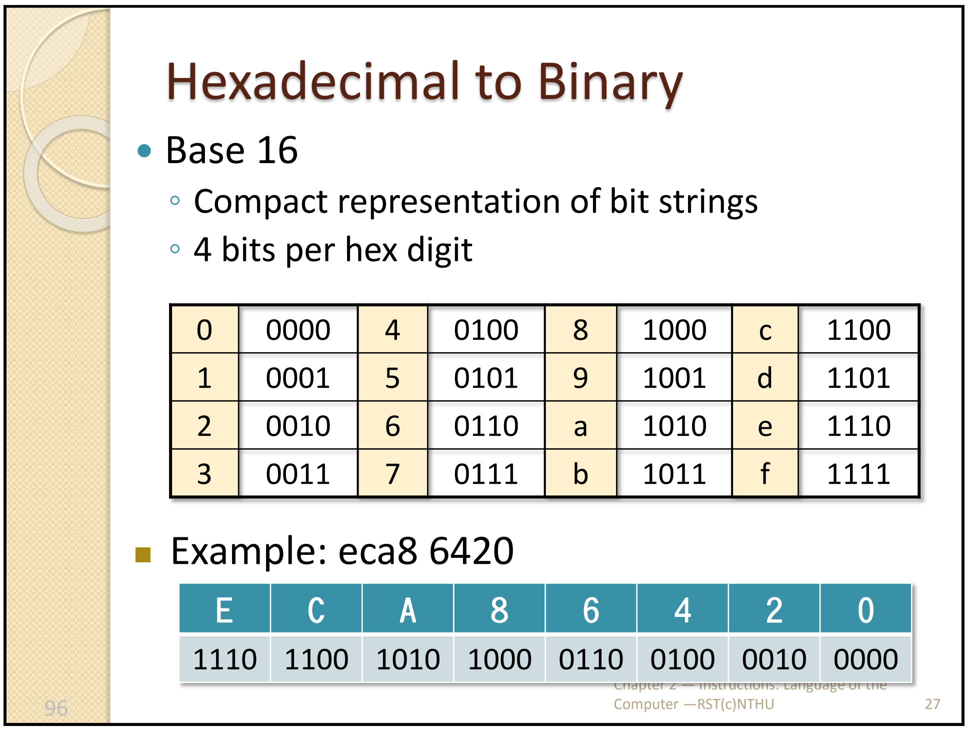
```
add rd, rs, rt; R[rd]=R[rs]+R[rt]
add $t0, $s1, $s2
```

special	$s1	$s2	$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

$00000010001100100100000000100000_2 = 02324020_{16}$

95

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 26



Hexadecimal to Binary

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

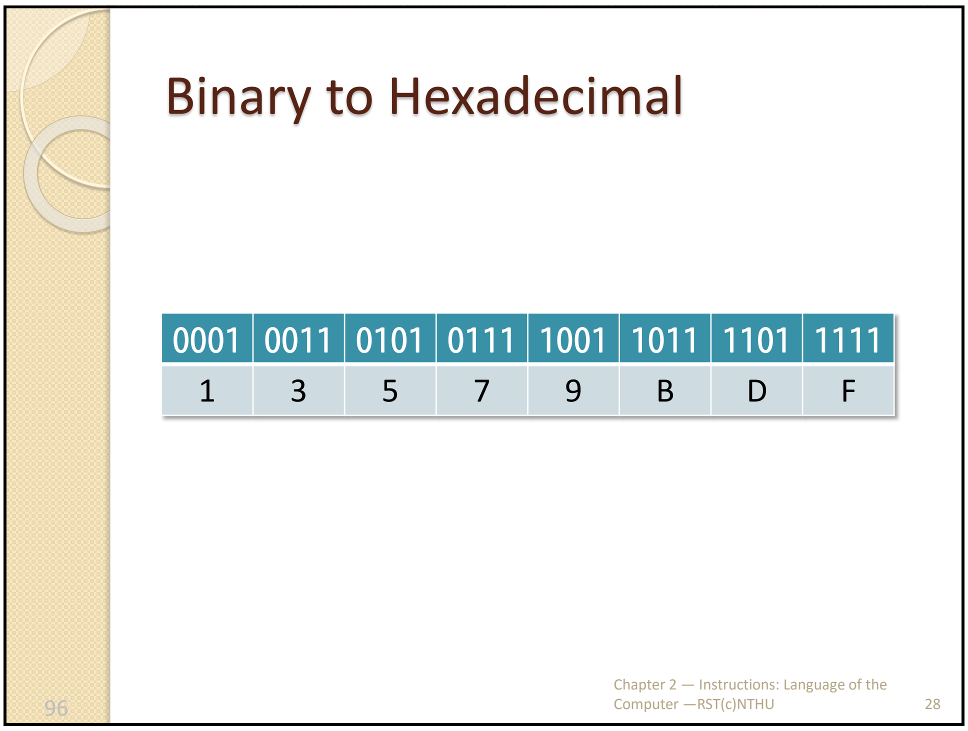
0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

- Example: eca8 6420

E	C	A	8	6	4	2	0
1110	1100	1010	1000	0110	0100	0010	0000

96

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 27

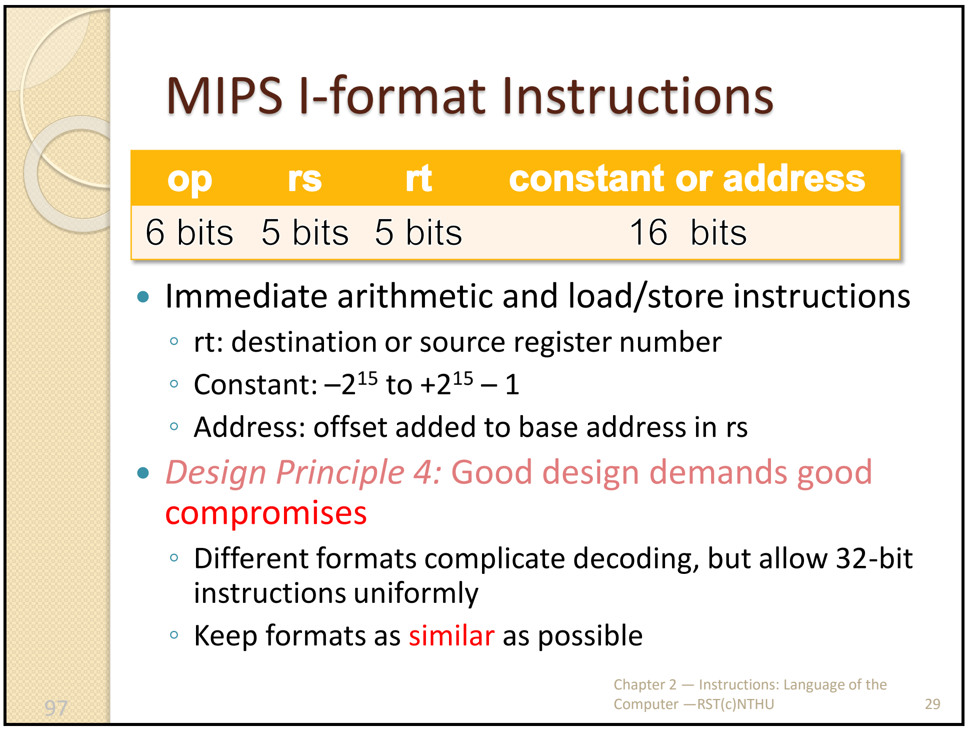


Binary to Hexadecimal

0001	0011	0101	0111	1001	1011	1101	1111
1	3	5	7	9	B	D	F

96

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 28



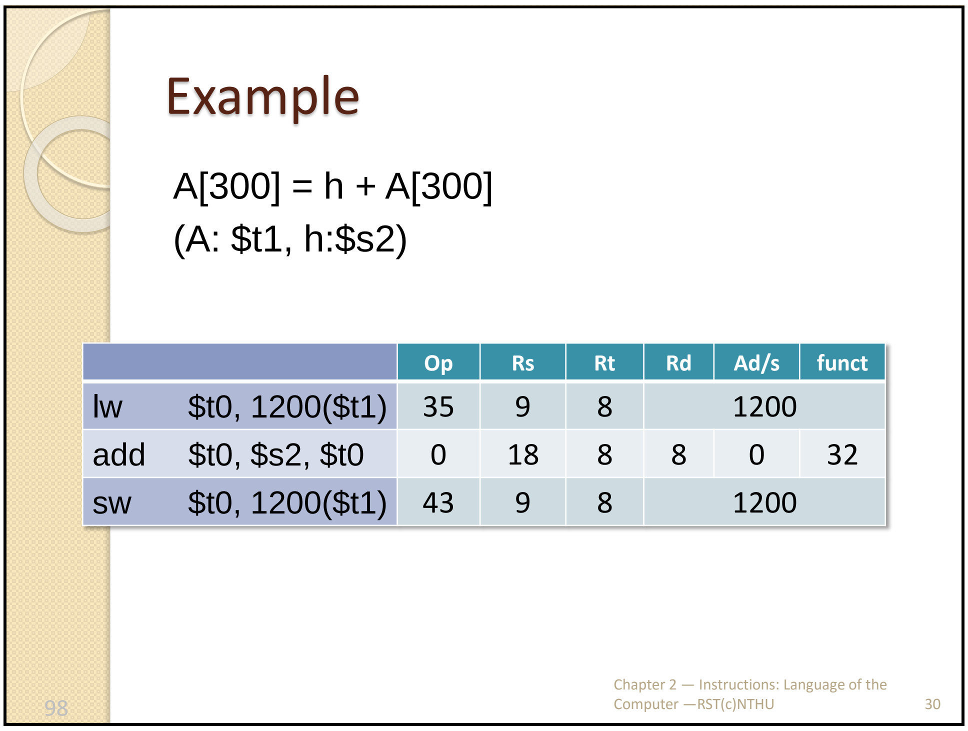
MIPS I-format Instructions

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 $+2^{15}-1$
  - 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

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 29

97



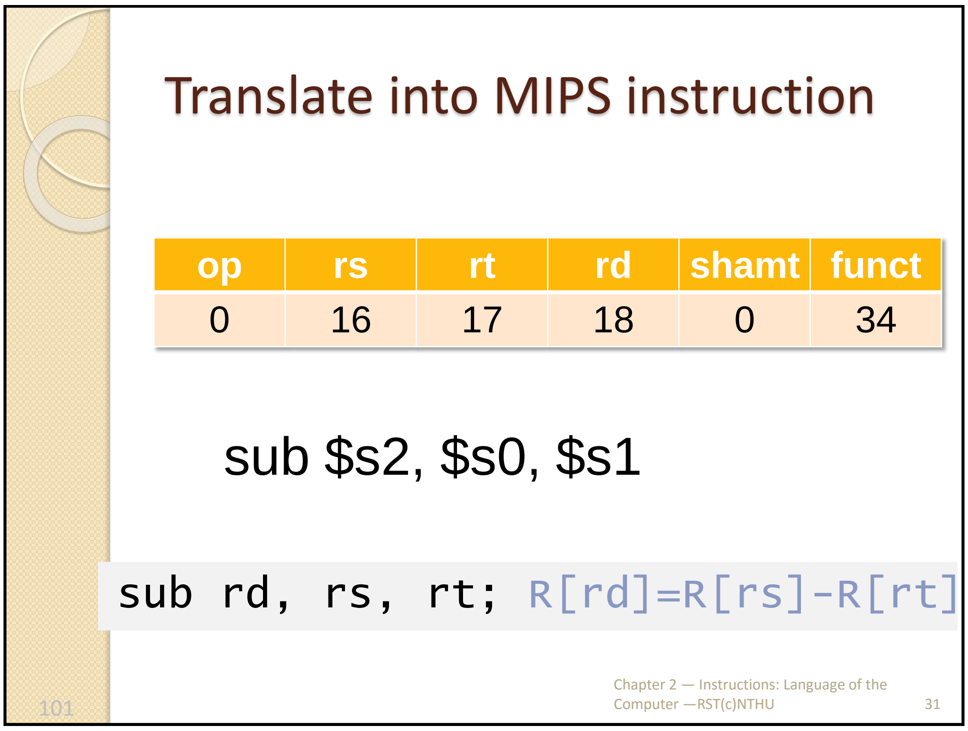
Example

A[300] = h + A[300]
(A: $t1, h:$s2)

	Op	Rs	Rt	Rd	Ad/s	funct
lw $t0, 1200($t1)	35	9	8	1200		
add $t0, $s2, $t0	0	18	8	8	0	32
sw $t0, 1200($t1)	43	9	8	1200		

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 30

98

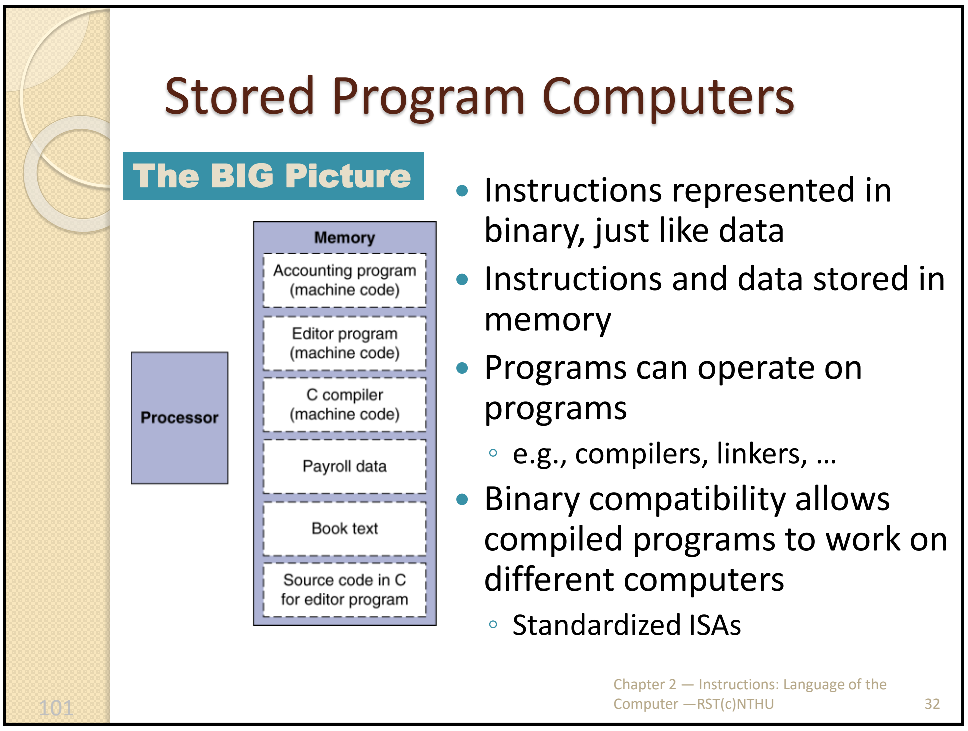


Translate into MIPS instruction

op	rs	rt	rd	shamt	funct
0	16	17	18	0	34

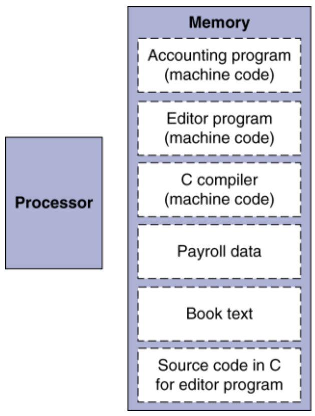
sub $s2, $s0, $s1

```
sub rd, rs, rt; R[rd]=R[rs]-R[rt]
```

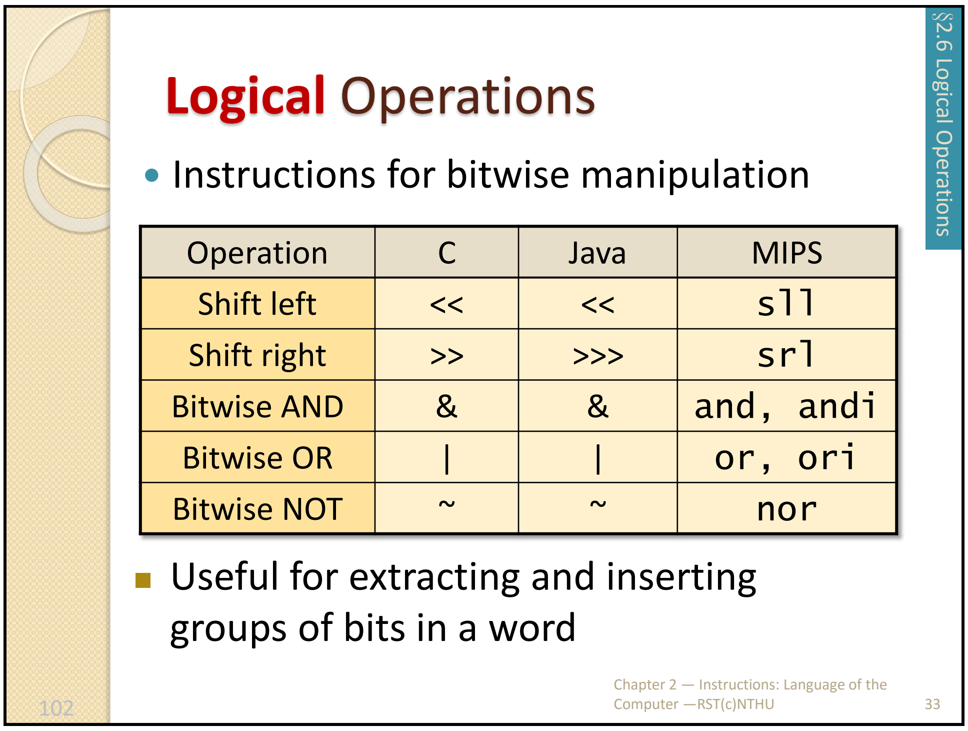


Stored Program Computers

The BIG Picture


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 represented in binary, just like data
- Instructions and data stored in memory
- Programs can operate on programs
  - e.g., compilers, linkers, …
- Binary compatibility allows compiled programs to work on different computers
  - Standardized ISAs



§2.6 Logical Operations

Logical Operations

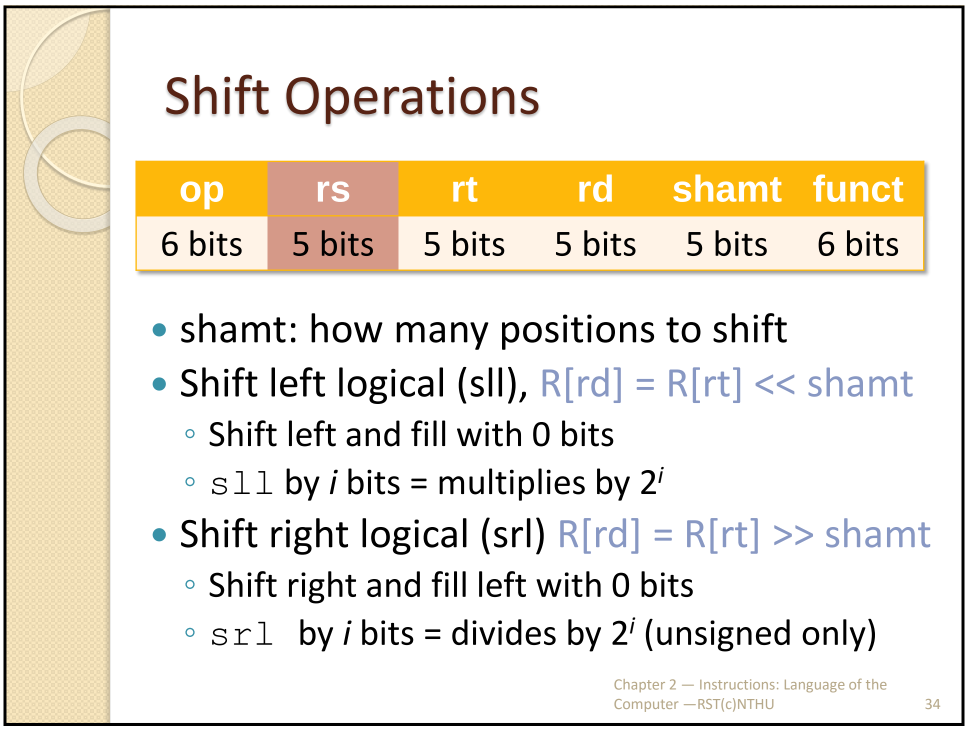
- Instructions for bitwise manipulation

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

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

102

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 33

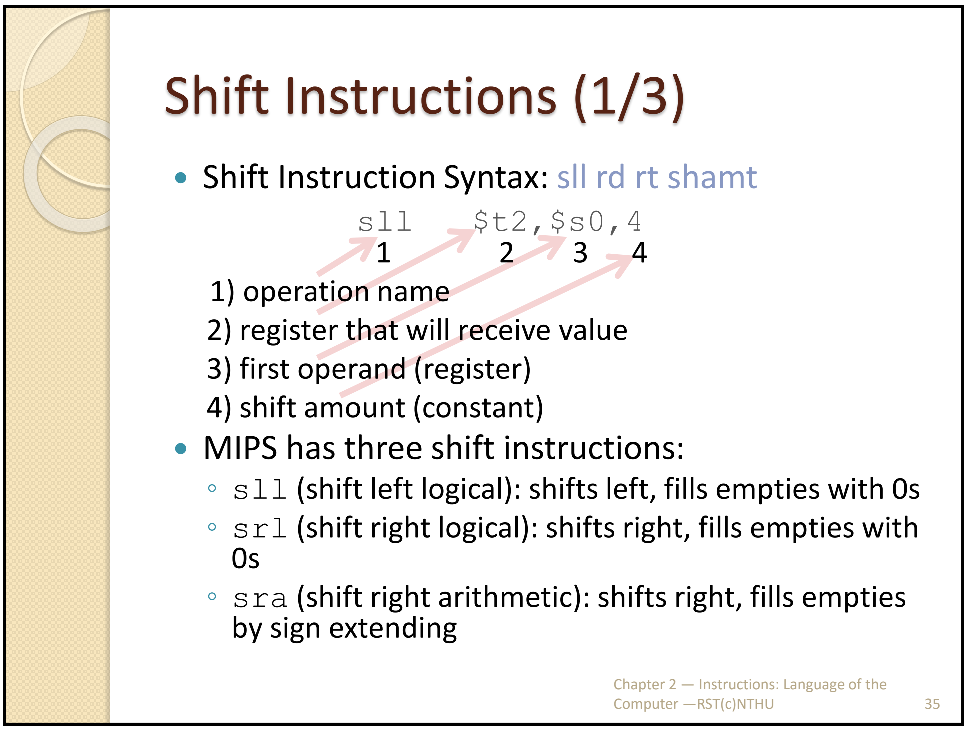


Shift Operations

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), R[rd] = R[rt] << shamt
  - Shift left and fill with 0 bits
  - `sll` by *i* bits = multiplies by 2^i
- Shift right logical (srl) R[rd] = R[rt] >> shamt
  - Shift right and fill left with 0 bits
  - `srl` by *i* bits = divides by 2^i (unsigned only)

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 34



Shift Instructions (1/3)

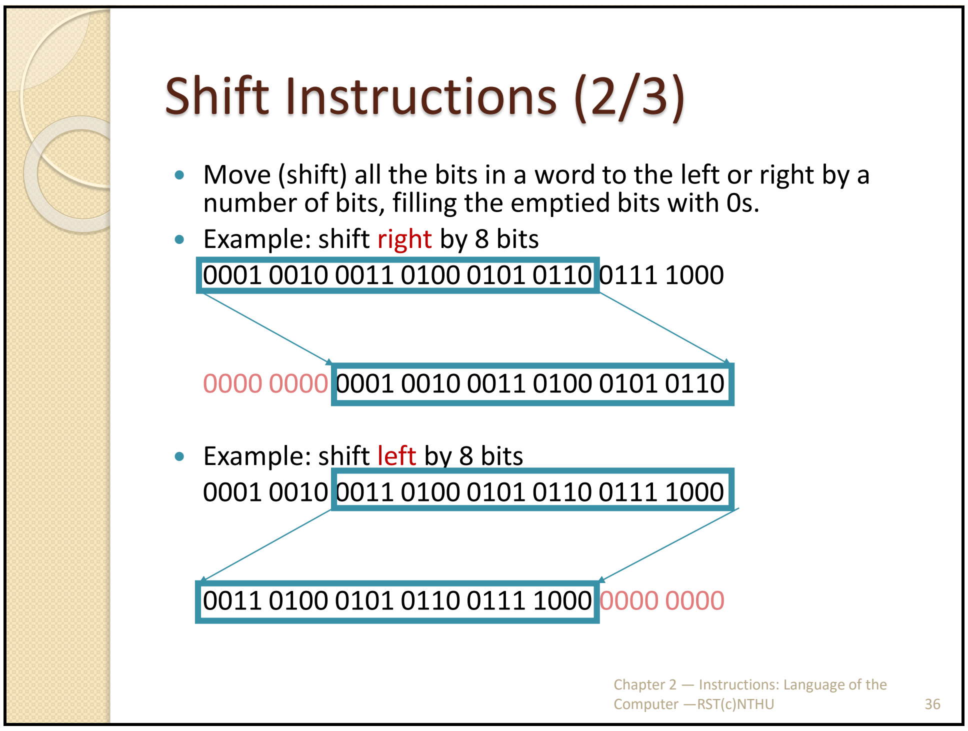
- Shift Instruction Syntax: sll rd rt shamt

```
sll    $t2,$s0,4
  1        2   3  4
```

1) operation name
2) register that will receive value
3) first operand (register)
4) shift amount (constant)

- MIPS has three shift instructions:
  - `sll` (shift left logical): shifts left, fills empties with 0s
  - `srl` (shift right logical): shifts right, fills empties with 0s
  - `sra` (shift right arithmetic): shifts right, fills empties by sign extending

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 35



Shift Instructions (2/3)

- Move (shift) all the bits in a word to the left or right by a number of bits, filling the emptied bits with 0s.
- Example: shift right by 8 bits

$\boxed{\text{0001 0010 0011 0100 0101 0110}}$ 0111 1000

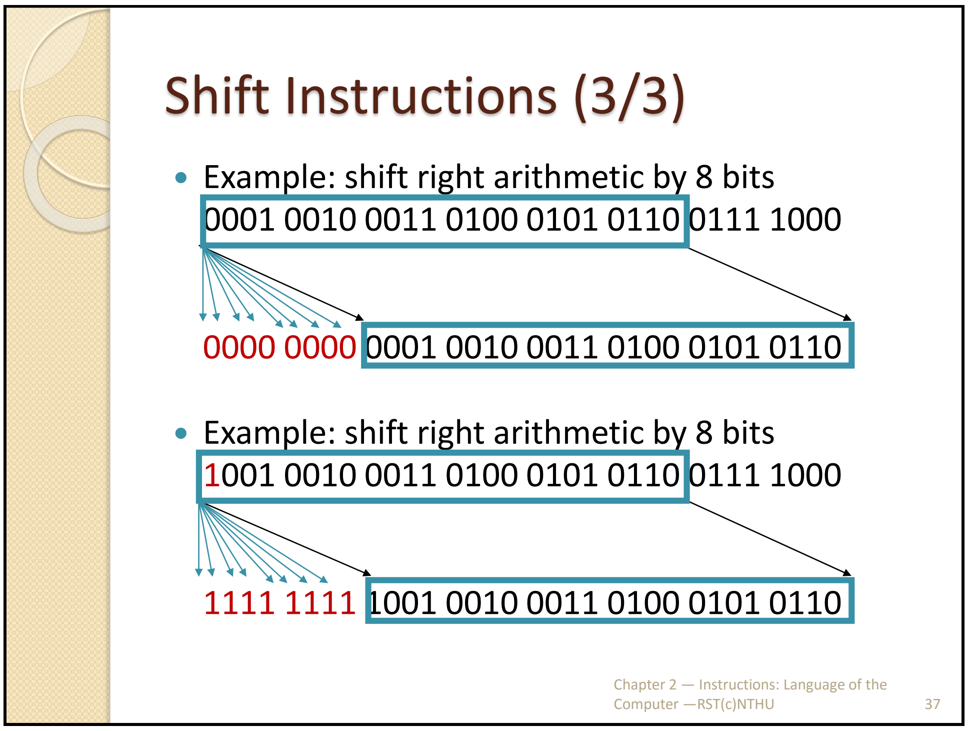
0000 0000 $\boxed{\text{0001 0010 0011 0100 0101 0110}}$

- Example: shift left by 8 bits

0001 0010 $\boxed{\text{0011 0100 0101 0110 0111 1000}}$

$\boxed{\text{0011 0100 0101 0110 0111 1000}}$ 0000 0000

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 36



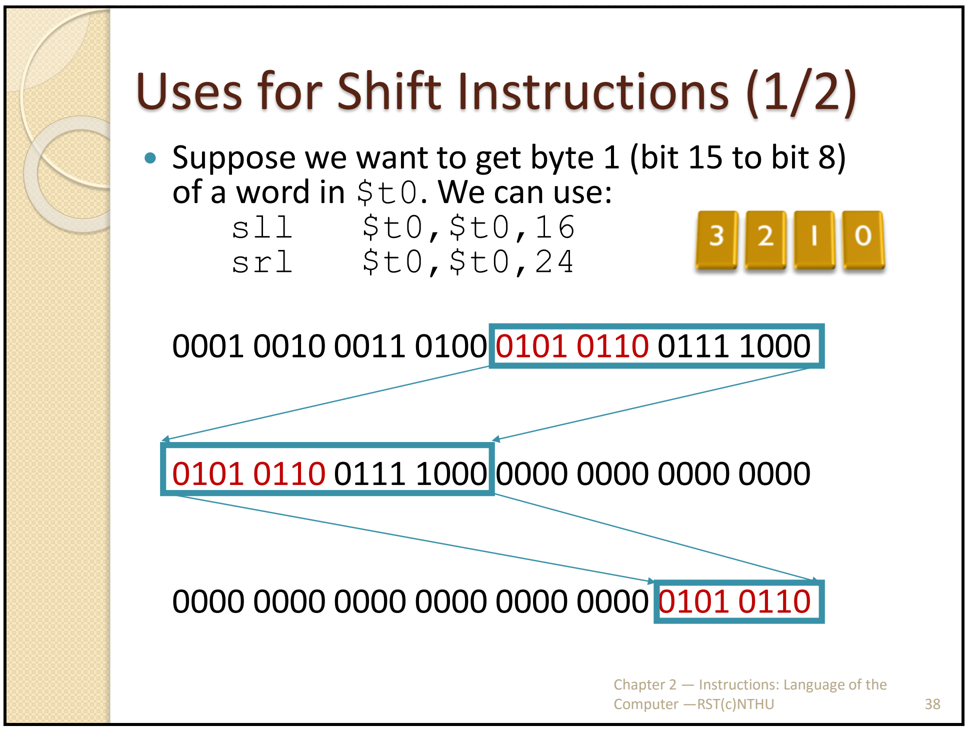
Shift Instructions (3/3)

- Example: shift right arithmetic by 8 bits
  0001 0010 0011 0100 0101 0110 0111 1000

  0000 0000 0001 0010 0011 0100 0101 0110
- Example: shift right arithmetic by 8 bits
  1001 0010 0011 0100 0101 0110 0111 1000

  1111 1111 1001 0010 0011 0100 0101 0110

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 37



Uses for Shift Instructions (1/2)

- Suppose we want to get byte 1 (bit 15 to bit 8) of a word in `$t0`. We can use:

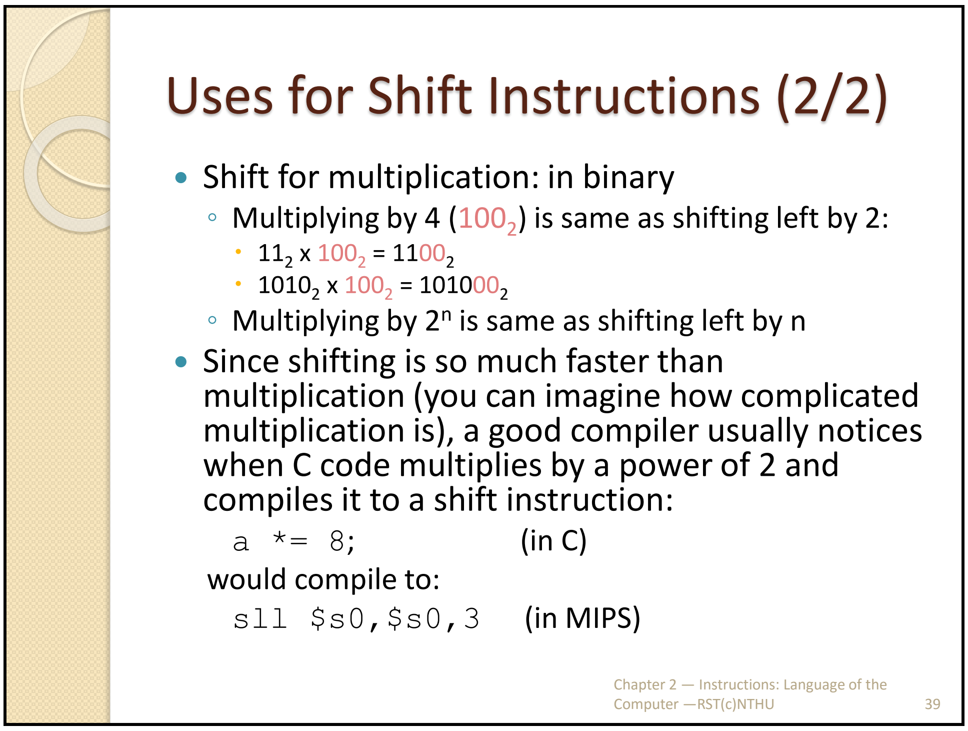
```
    sll   $t0,$t0,16
    srl   $t0,$t0,24
```

0001 0010 0011 0100 0101 0110 0111 1000

0101 0110 0111 1000 0000 0000 0000 0000

0000 0000 0000 0000 0000 0000 0101 0110

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 38



Uses for Shift Instructions (2/2)

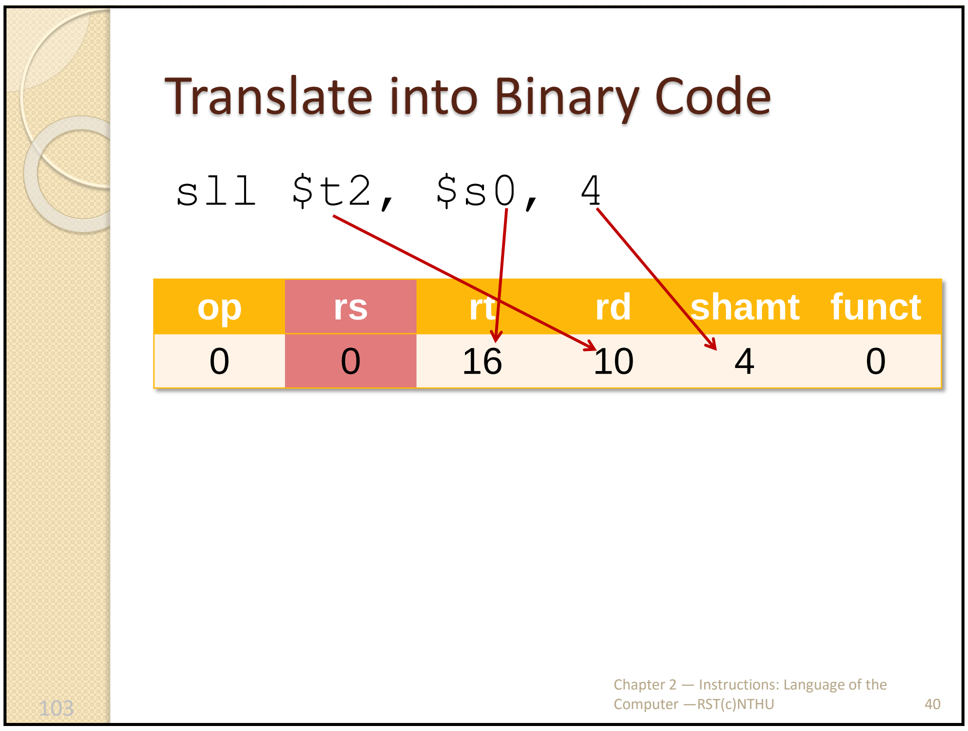
- Shift for multiplication: in binary
  - Multiplying by 4 (100_2) is same as shifting left by 2:
    - 11_2 x 100_2 = 1100_2
    - 1010_2 x 100_2 = 101000_2
  - Multiplying by 2^n is same as shifting left by n
- Since shifting is so much faster than multiplication (you can imagine how complicated multiplication is), a good compiler usually notices when C code multiplies by a power of 2 and compiles it to a shift instruction:

```
a *= 8;
```
(in C)

would compile to:

```
sll $s0,$s0,3
```
(in MIPS)

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 39



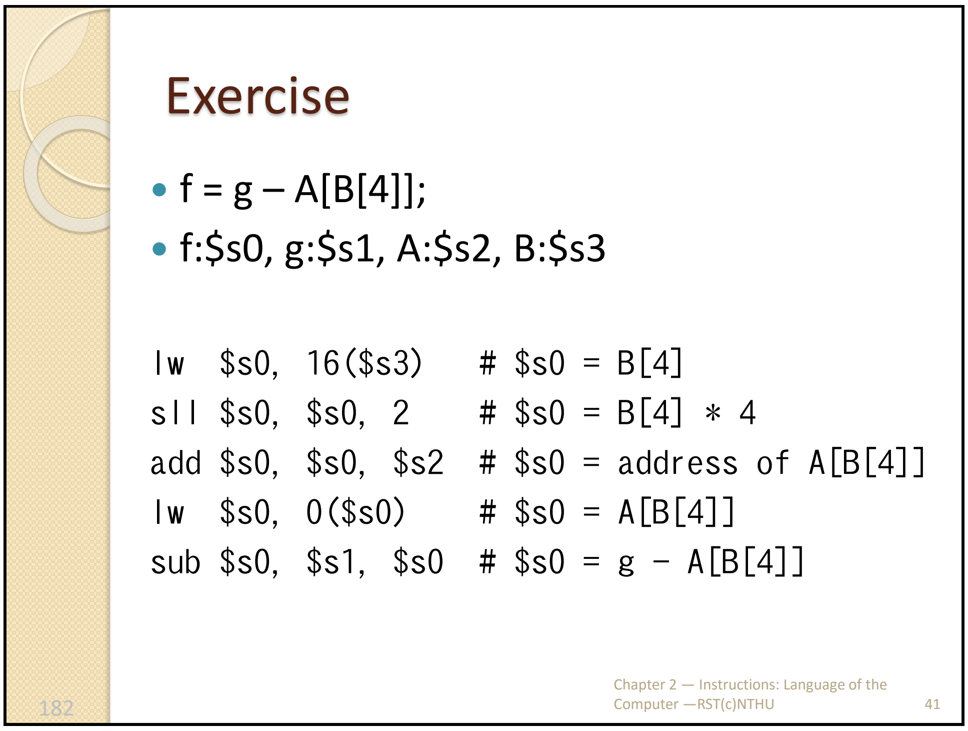
Translate into Binary Code

```
sll $t2, $s0, 4
```

op	rs	rt	rd	shamt	funct
0	0	16	10	4	0

103

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 40



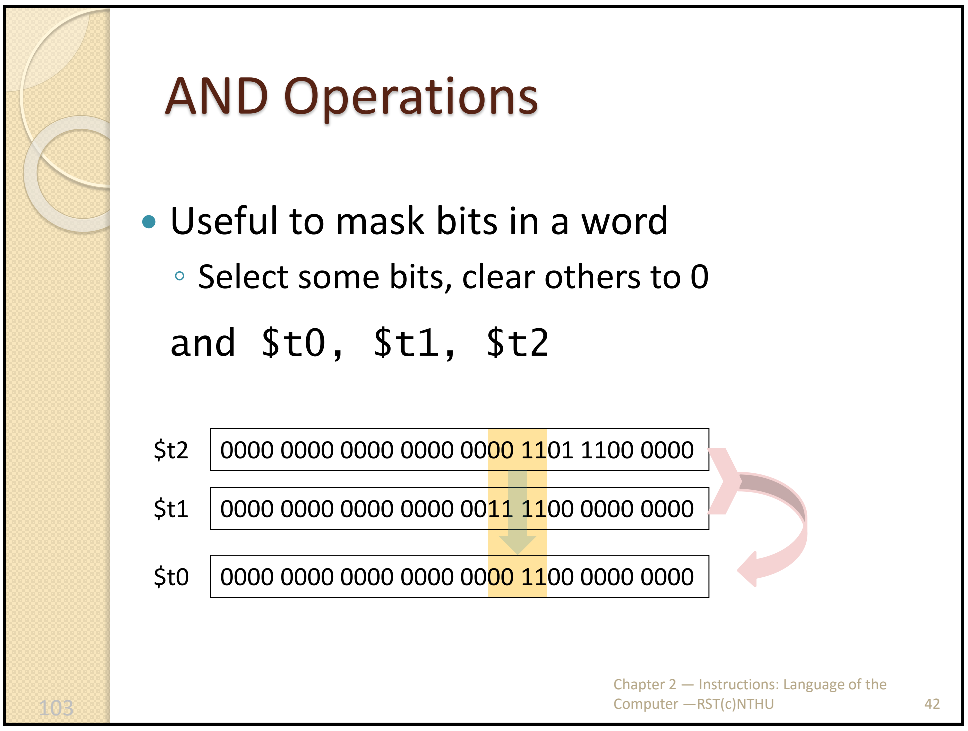
Exercise

- f = g – A[B[4]];
- f:$s0, g:$s1, A:$s2, B:$s3

```
lw  $s0, 16($s3)   # $s0 = B[4]
sll $s0, $s0, 2    # $s0 = B[4] * 4
add $s0, $s0, $s2  # $s0 = address of A[B[4]]
lw  $s0, 0($s0)    # $s0 = A[B[4]]
sub $s0, $s1, $s0  # $s0 = g - A[B[4]]
```

182

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 41



AND Operations

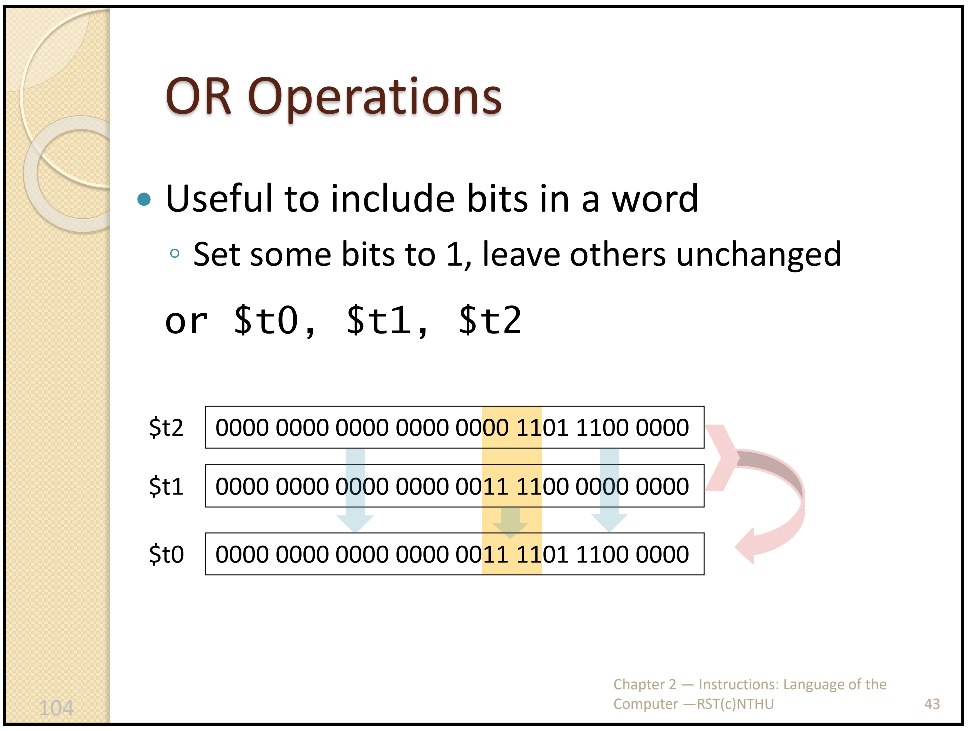
- Useful to mask bits in a word
  - Select some bits, clear others to 0

`and $t0, $t1, $t2`

```
$t2  0000 0000 0000 0000 0000 1101 1100 0000
$t1  0000 0000 0000 0000 0011 1100 0000 0000
$t0  0000 0000 0000 0000 0000 1100 0000 0000
```

103

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 42



OR Operations

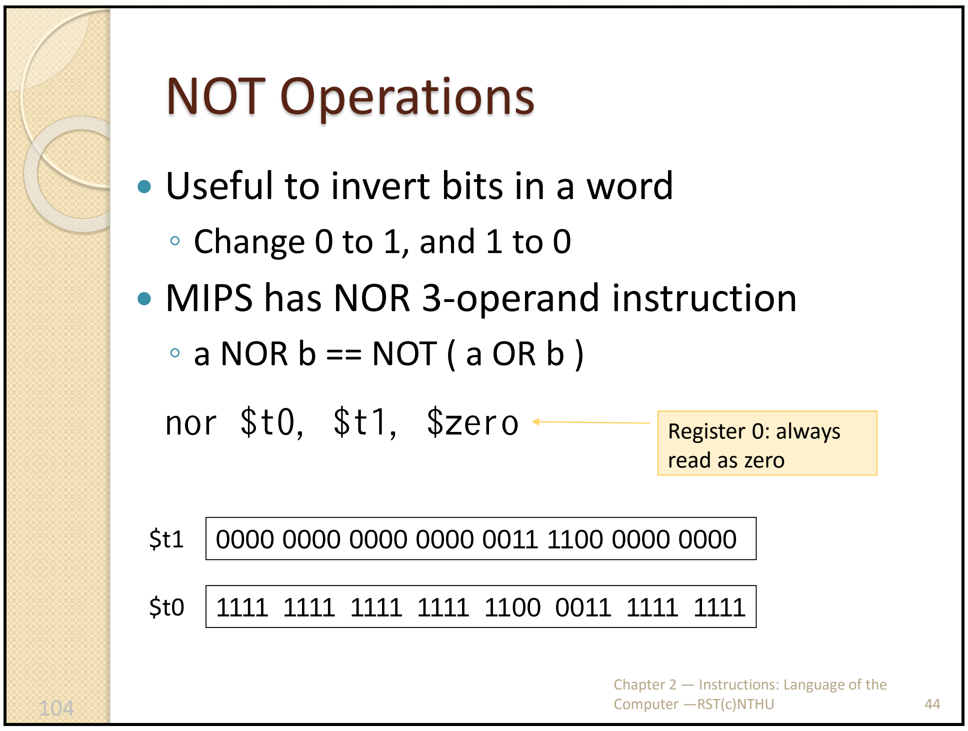
- Useful to include bits in a word
  - Set some bits to 1, leave others unchanged

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

```
$t2   0000 0000 0000 0000 0000 1101 1100 0000
$t1   0000 0000 0000 0000 0011 1100 0000 0000
$t0   0000 0000 0000 0000 0011 1101 1100 0000
```

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 43

104



NOT Operations

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

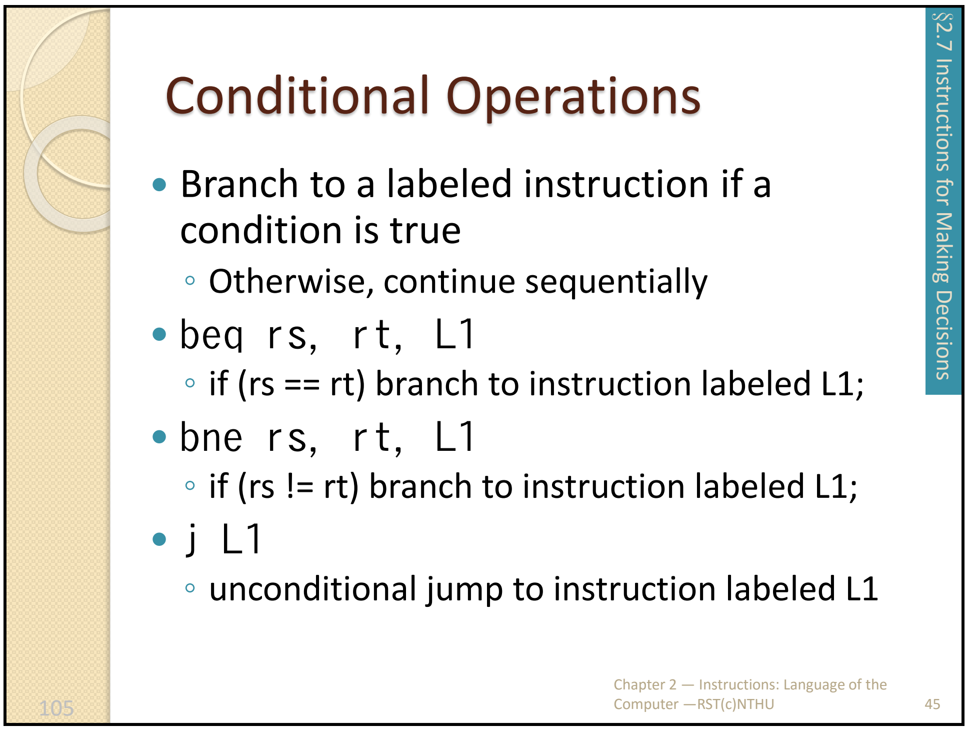
```
nor $t0, $t1, $zero
```

Register 0: always read as zero

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

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 44

104



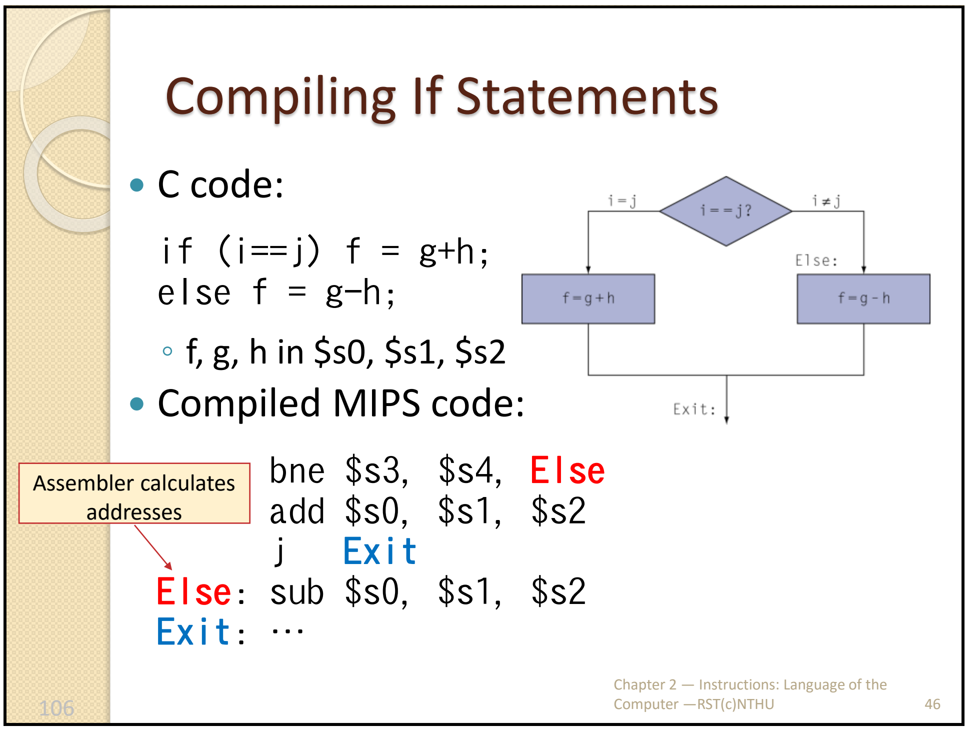
Conditional Operations

§2.7 Instructions for Making Decisions

- 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

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 45

105

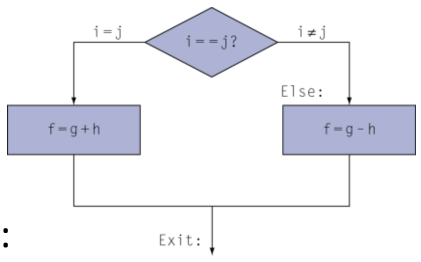


Compiling If Statements

- C code:

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

  - f, g, h in $s0, $s1, $s2



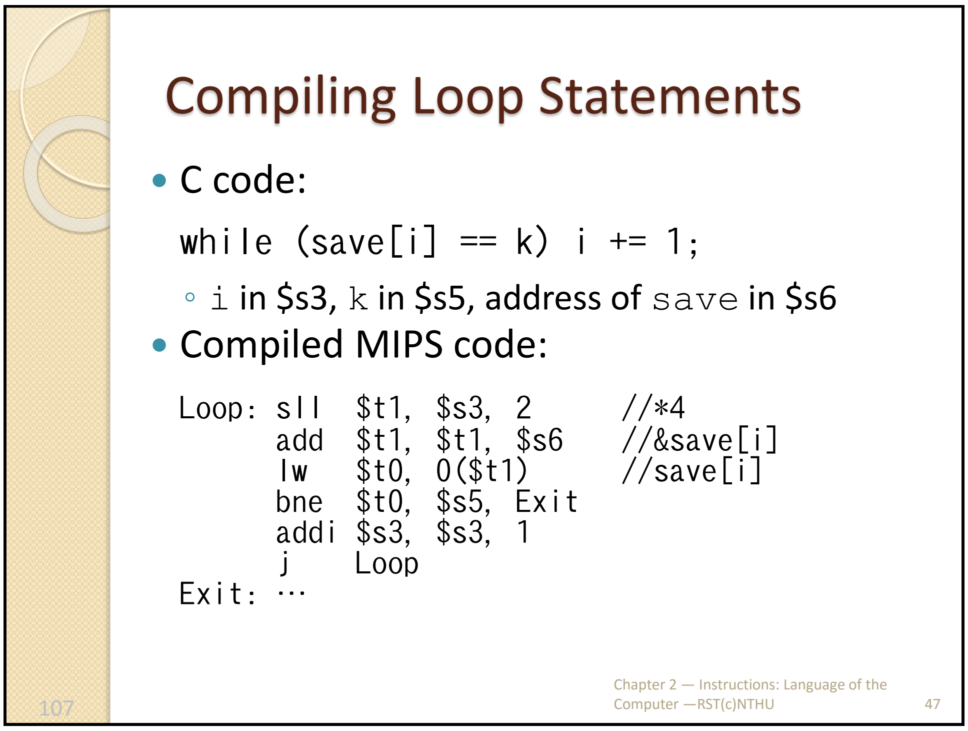
- Compiled MIPS code:

Assembler calculates addresses

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

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 46

106



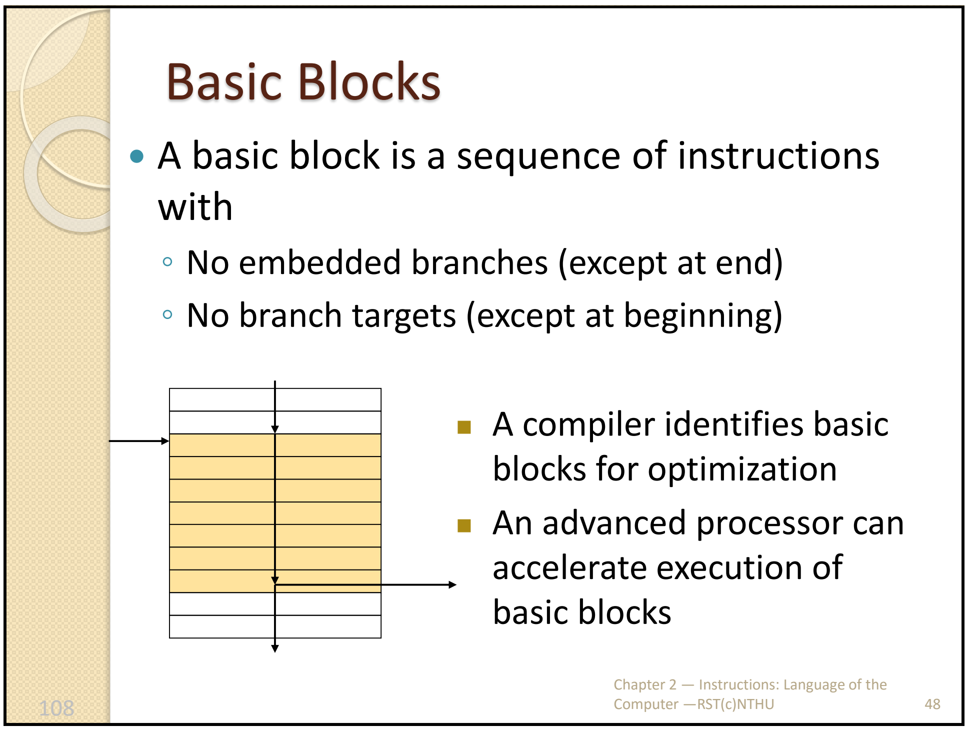
Compiling Loop Statements

- C code:

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

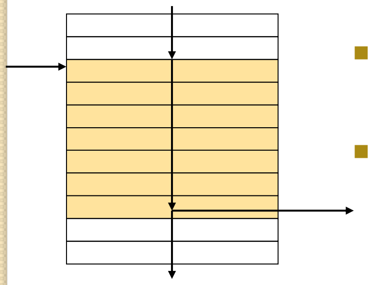
  - `i` in $s3, `k` in $s5, address of `save` in $s6
- Compiled MIPS code:

```
Loop: sll  $t1, $s3, 2      //*4
      add  $t1, $t1, $s6    //&save[i]
      lw   $t0, 0($t1)      //save[i]
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
Exit: …
```

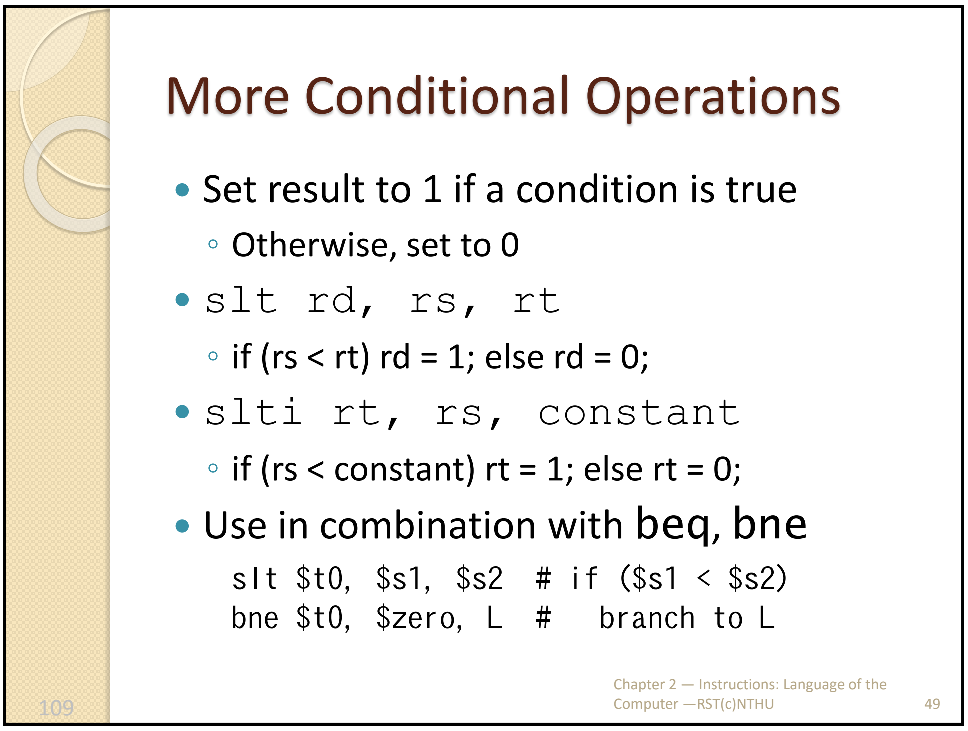


Basic Blocks

- A basic block is a sequence of instructions with
  - No embedded branches (except at end)
  - No branch targets (except at beginning)



- A compiler identifies basic blocks for optimization
- An advanced processor can accelerate execution of basic blocks



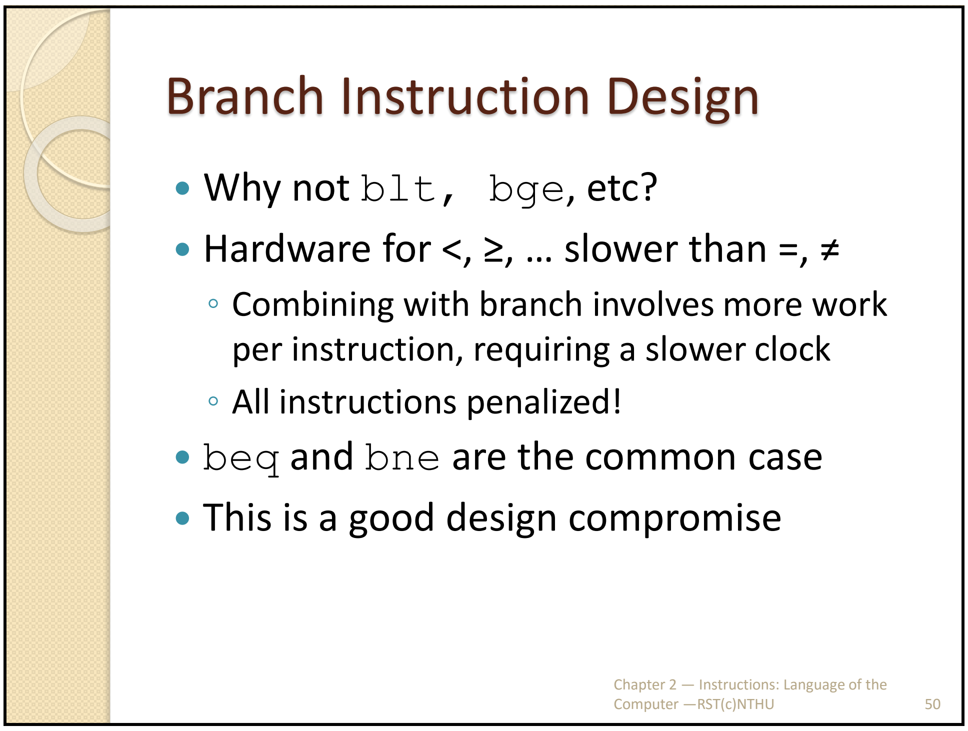
More Conditional Operations

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

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

109

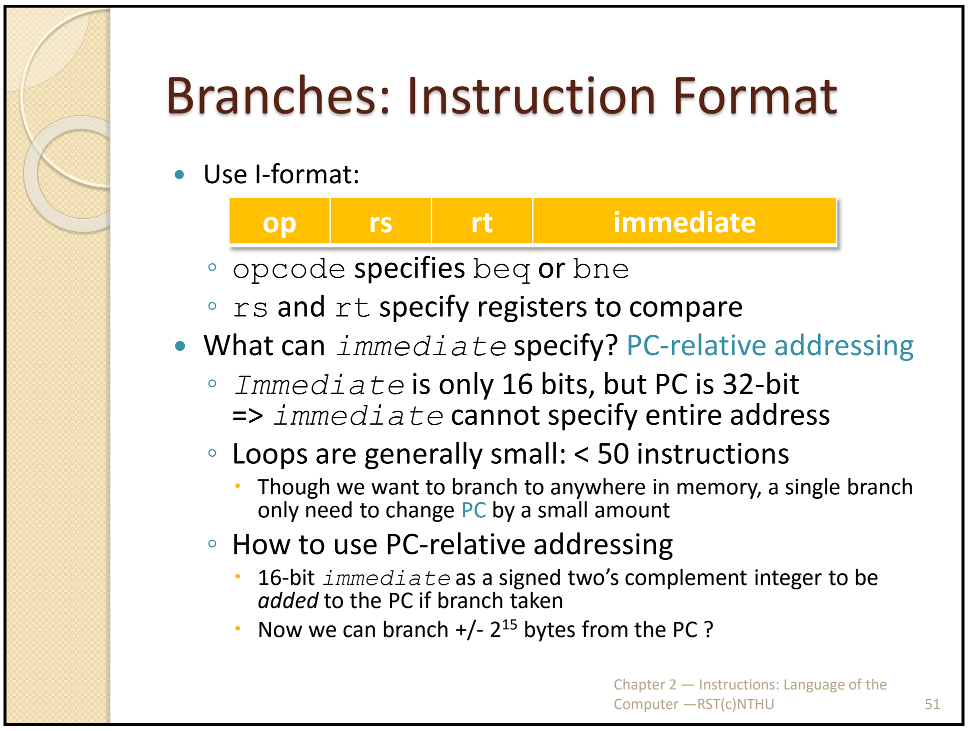
Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 49



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

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 50



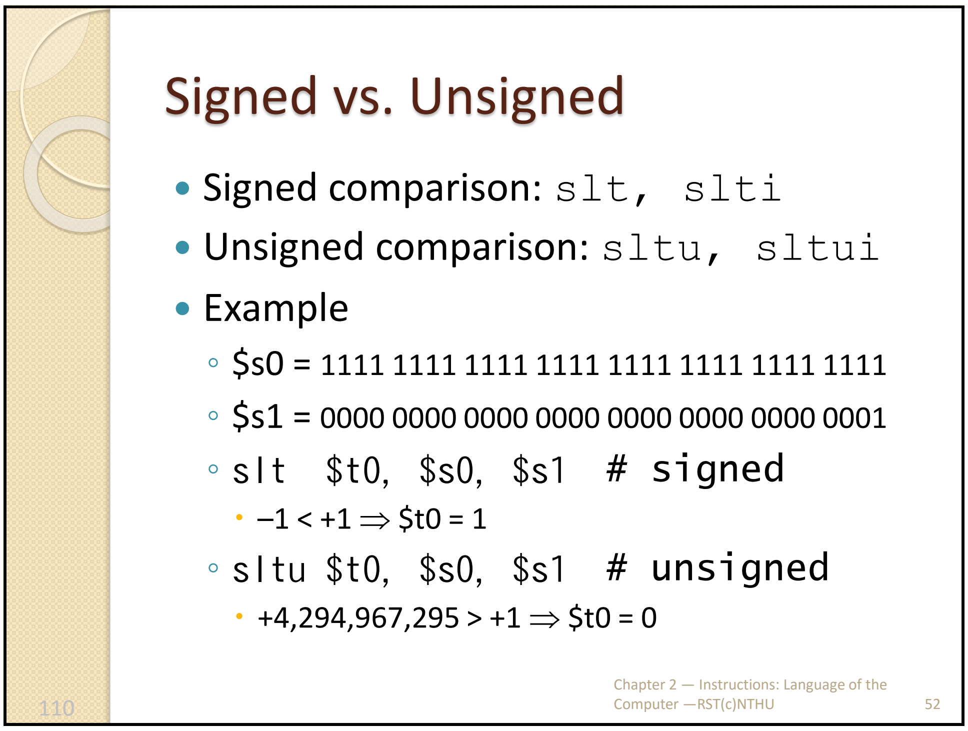
Branches: Instruction Format

- Use I-format:

op	rs	rt	immediate

  - `opcode` specifies `beq` or `bne`
  - `rs` and `rt` specify registers to compare
- What can *`immediate`* specify? PC-relative addressing
  - *`Immediate`* is only 16 bits, but PC is 32-bit
    => *`immediate`* cannot specify entire address
  - Loops are generally small: < 50 instructions
    - Though we want to branch to anywhere in memory, a single branch only need to change PC by a small amount
  - How to use PC-relative addressing
    - 16-bit *`immediate`* as a signed two's complement integer to be *added* to the PC if branch taken
    - Now we can branch +/- 2^{15} bytes from the PC ?

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 51

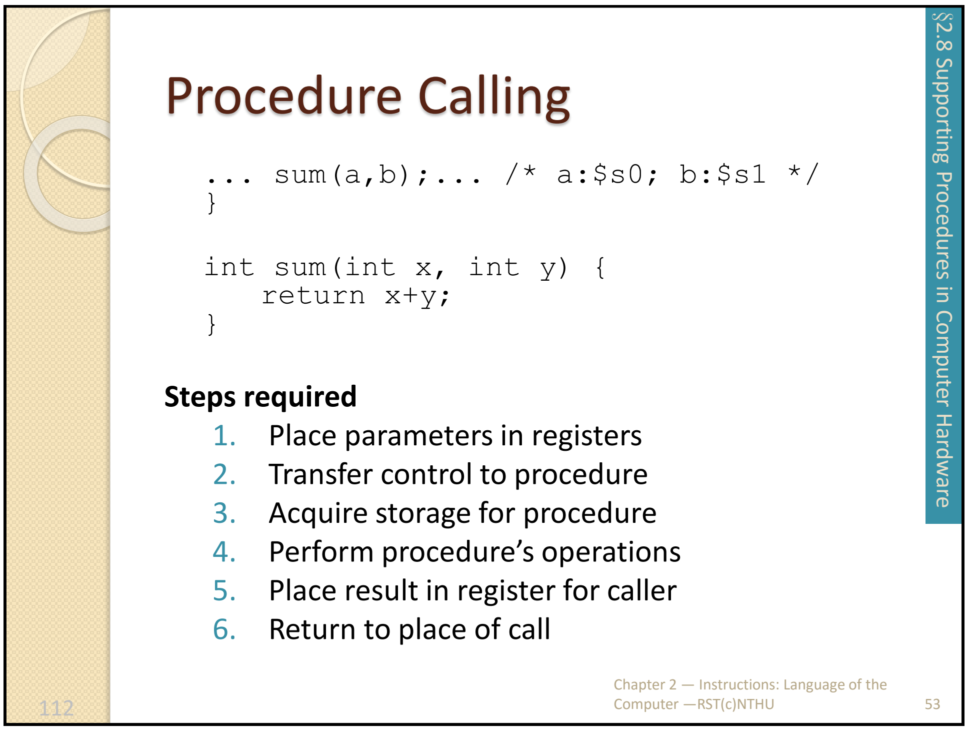


Signed vs. Unsigned

- Signed comparison: `slt, slti`
- Unsigned comparison: `sltu, sltui`
- Example
  - \$s0 = 1111 1111 1111 1111 1111 1111 1111 1111
  - \$s1 = 0000 0000 0000 0000 0000 0000 0000 0001
  - `slt  $t0, $s0, $s1  # signed`
    - –1 < +1 $\Rightarrow$ \$t0 = 1
  - `sltu $t0, $s0, $s1  # unsigned`
    - +4,294,967,295 > +1 $\Rightarrow$ \$t0 = 0

110

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 52



Procedure Calling

```
... sum(a,b);... /* a:$s0; b:$s1 */
}

int sum(int x, int y) {
    return x+y;
}
```

Steps required

1. Place parameters in registers
2. Transfer control to procedure
3. Acquire storage for procedure
4. Perform procedure's operations
5. Place result in register for caller
6. Return to place of call

§2.8 Supporting Procedures in Computer Hardware

112

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 53

MIPS R2000 Organization

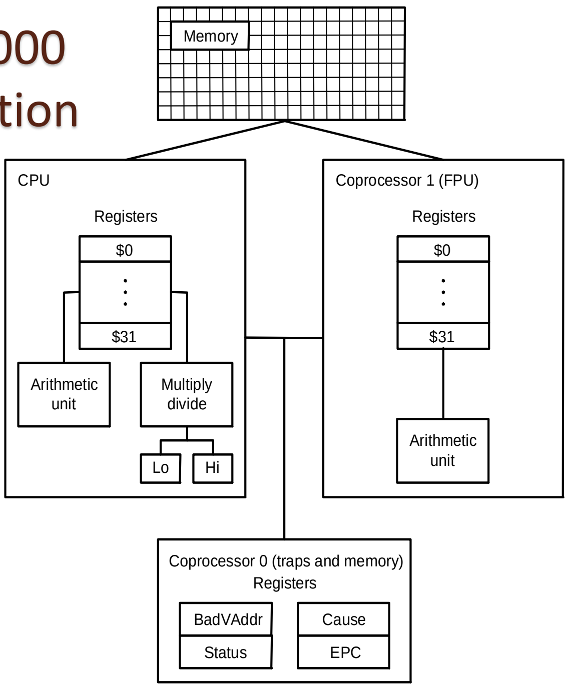
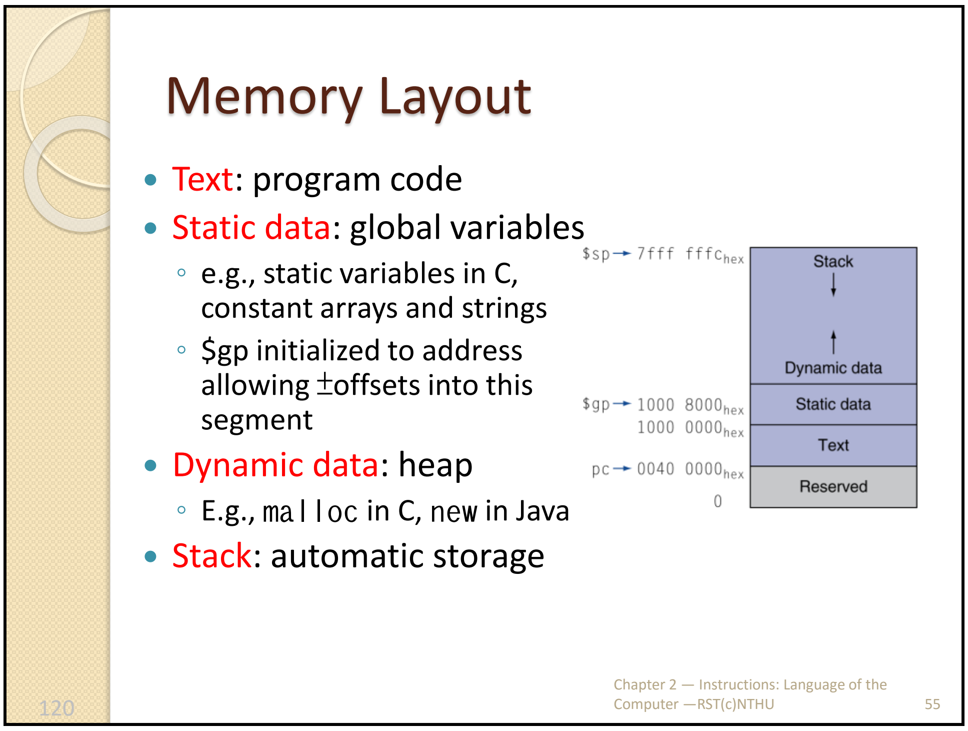


Fig. B.10.1

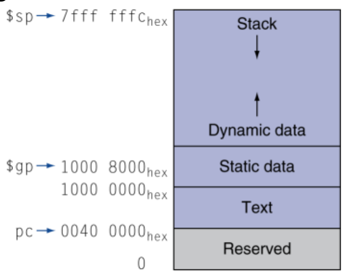
B-46

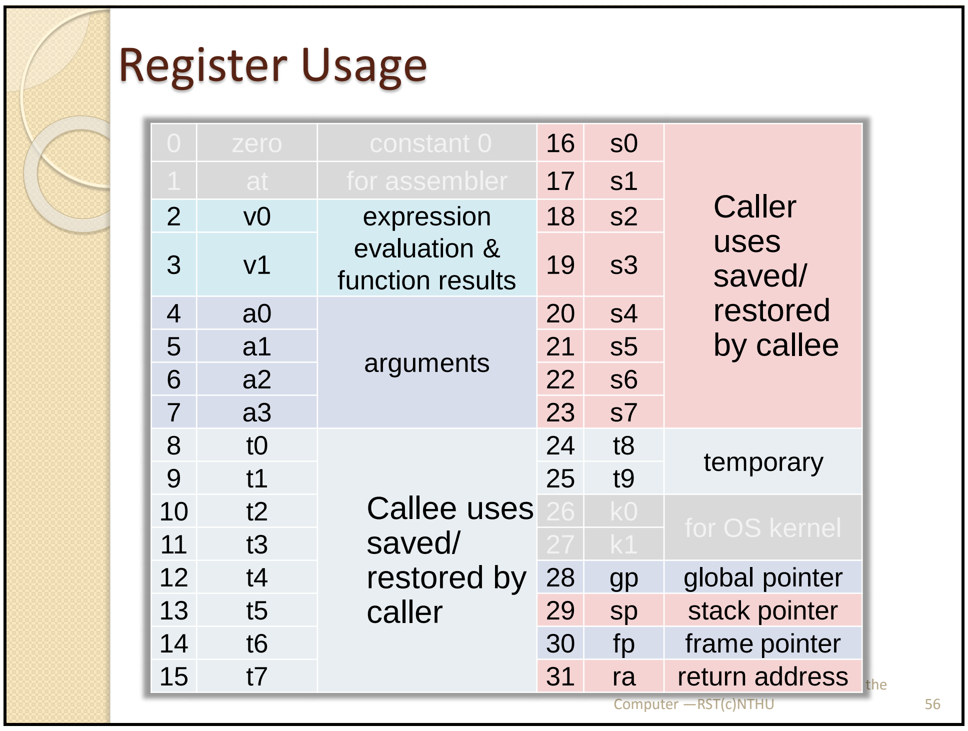
54



Memory Layout

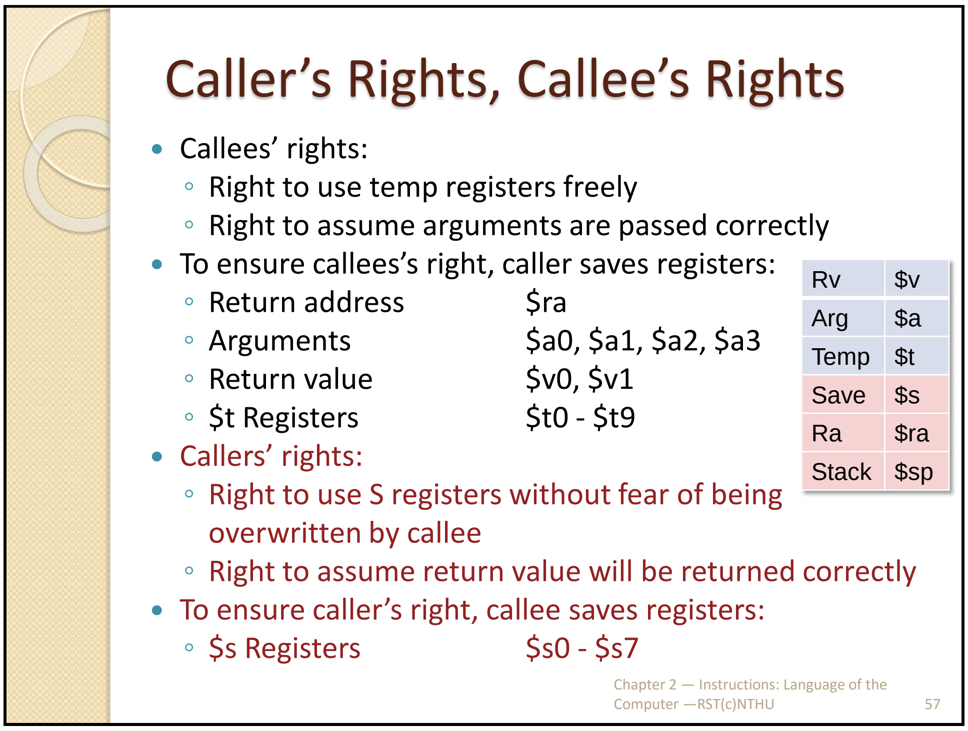
- Text: program code
- Static data: global variables
  - e.g., static variables in C, constant arrays and strings
  - $gp initialized to address allowing ±offsets into this segment
- Dynamic data: heap
  - E.g., malloc in C, new in Java
- Stack: automatic storage





Register Usage

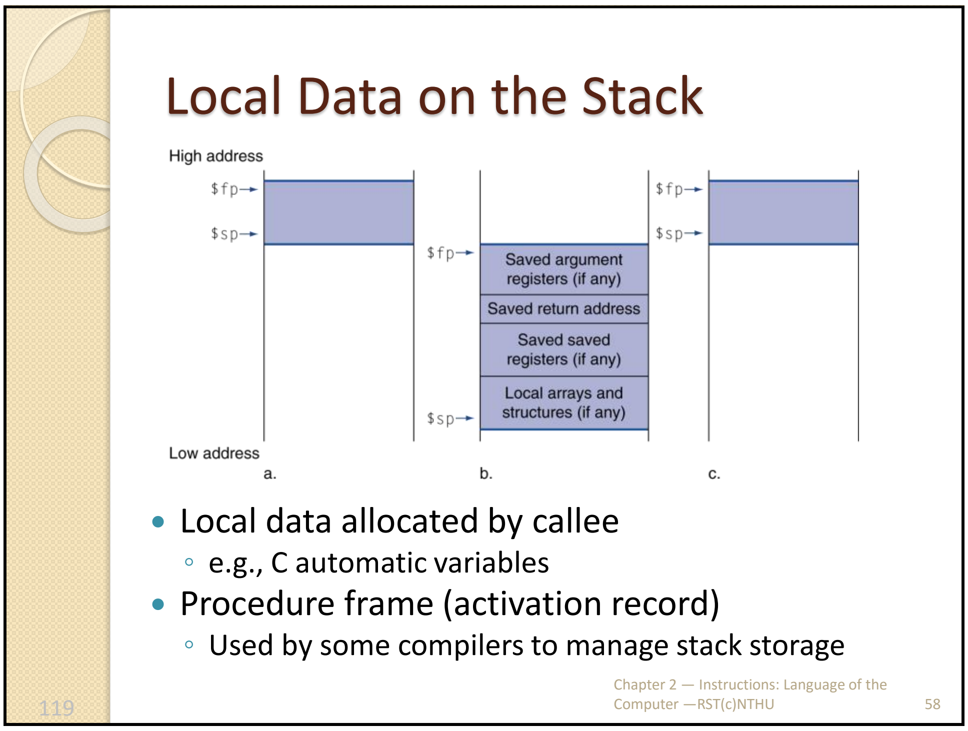
0	zero	constant 0	16	s0	Caller uses saved/ restored by callee
1	at	for assembler	17	s1	
2	v0	expression evaluation & function results	18	s2	
3	v1		19	s3	
4	a0	arguments	20	s4	
5	a1		21	s5	
6	a2		22	s6	
7	a3		23	s7	
8	t0	Callee uses saved/ restored by caller	24	t8	temporary
9	t1		25	t9	
10	t2		26	k0	for OS kernel
11	t3		27	k1	
12	t4		28	gp	global pointer
13	t5		29	sp	stack pointer
14	t6		30	fp	frame pointer
15	t7		31	ra	return address



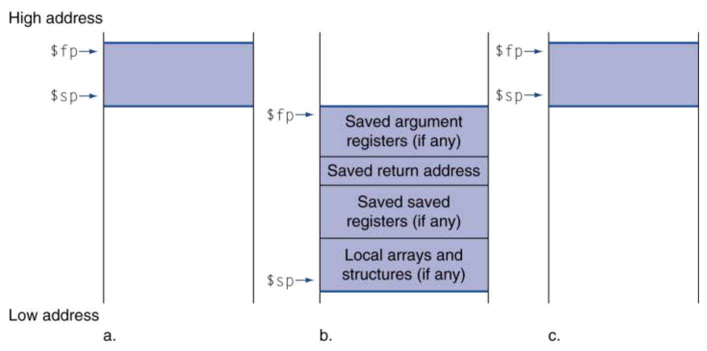
Caller’s Rights, Callee’s Rights

- Callees’ rights:
  - Right to use temp registers freely
  - Right to assume arguments are passed correctly
- To ensure callees’s right, caller saves registers:
  - Return address $ra
  - Arguments $a0, $a1, $a2, $a3
  - Return value $v0, $v1
  - $t Registers $t0 - $t9
- Callers’ rights:
  - Right to use S registers without fear of being overwritten by callee
  - Right to assume return value will be returned correctly
- To ensure caller’s right, callee saves registers:
  - $s Registers $s0 - $s7

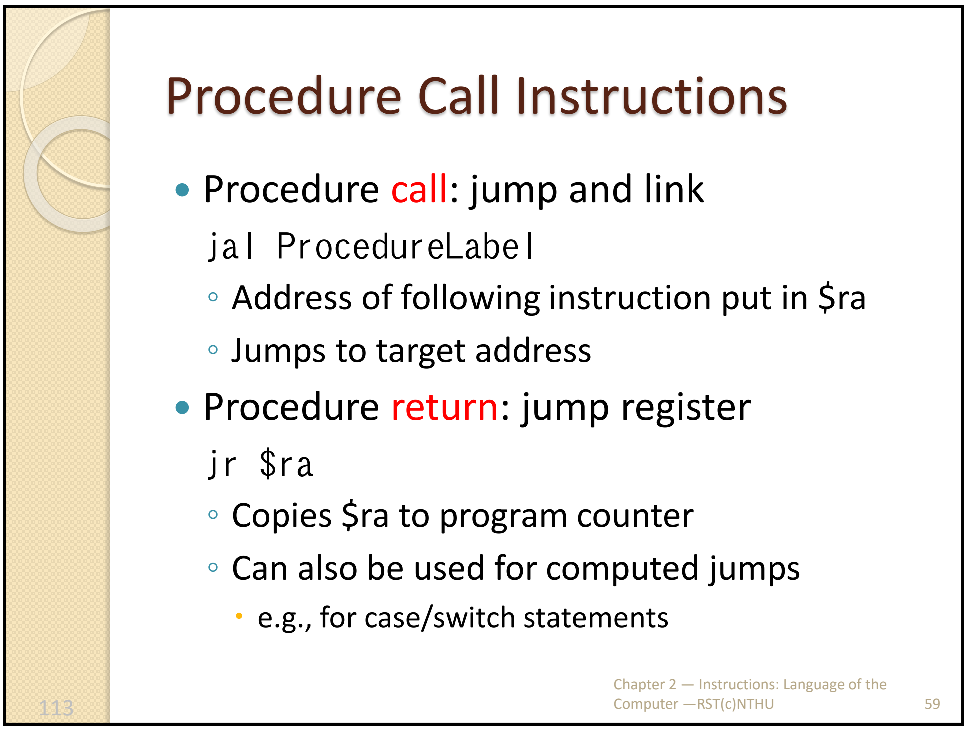
Rv	$v
Arg	$a
Temp	$t
Save	$s
Ra	$ra
Stack	$sp



Local Data on the Stack


High address
$fp
$sp
$fp
Saved argument registers (if any)
Saved return address
Saved saved registers (if any)
Local arrays and structures (if any)
$sp
$fp
$sp
Low address
a.
b.
c.

- Local data allocated by callee
  - e.g., C automatic variables
- Procedure frame (activation record)
  - Used by some compilers to manage stack storage



Procedure Call Instructions

- Procedure call: jump and link

```
jal ProcedureLabel
```

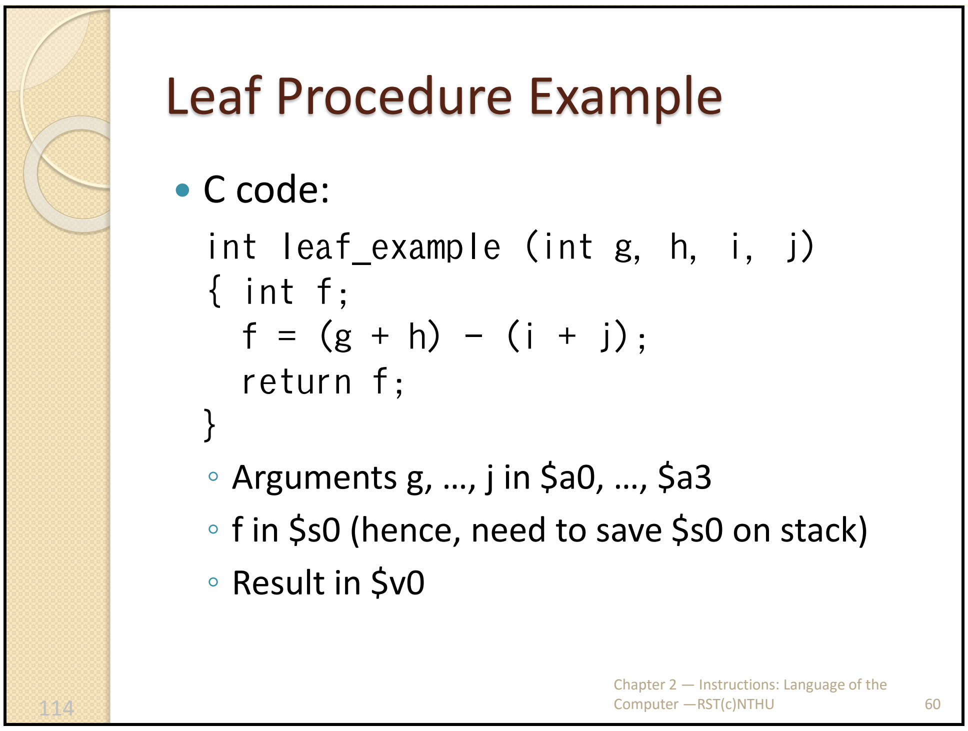
  - Address of following instruction put in $ra
  - Jumps to target address
- Procedure return: jump register

```
jr $ra
```

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

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 59

113



Leaf Procedure Example

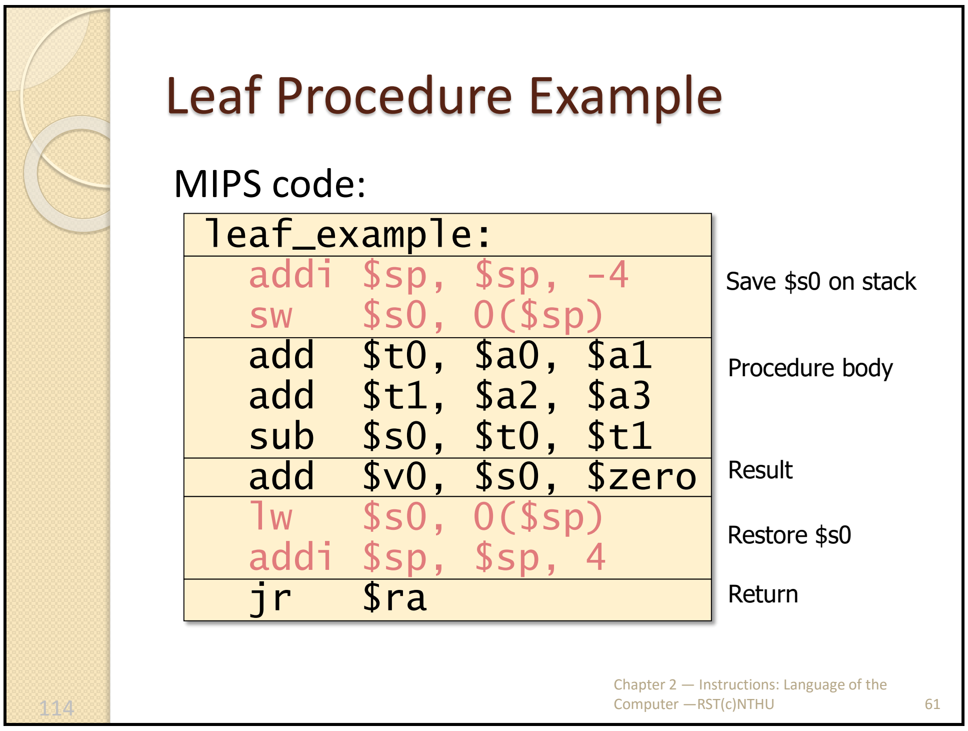
- C code:

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

  - Arguments g, …, j in $a0, …, $a3
  - f in $s0 (hence, need to save $s0 on stack)
  - Result in $v0

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 60

114



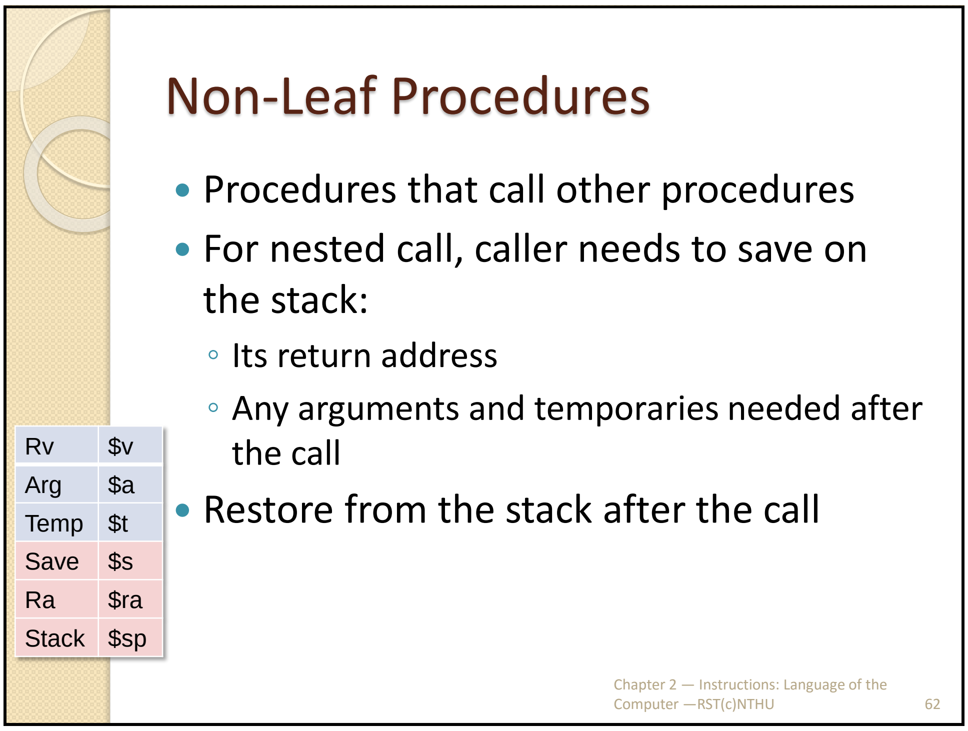
Leaf Procedure Example

MIPS code:

```
leaf_example:
    addi $sp, $sp, -4      Save $s0 on stack
    sw   $s0, 0($sp)
    add  $t0, $a0, $a1     Procedure body
    add  $t1, $a2, $a3
    sub  $s0, $t0, $t1
    add  $v0, $s0, $zero   Result
    lw   $s0, 0($sp)       Restore $s0
    addi $sp, $sp, 4
    jr   $ra               Return
```

114

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 61

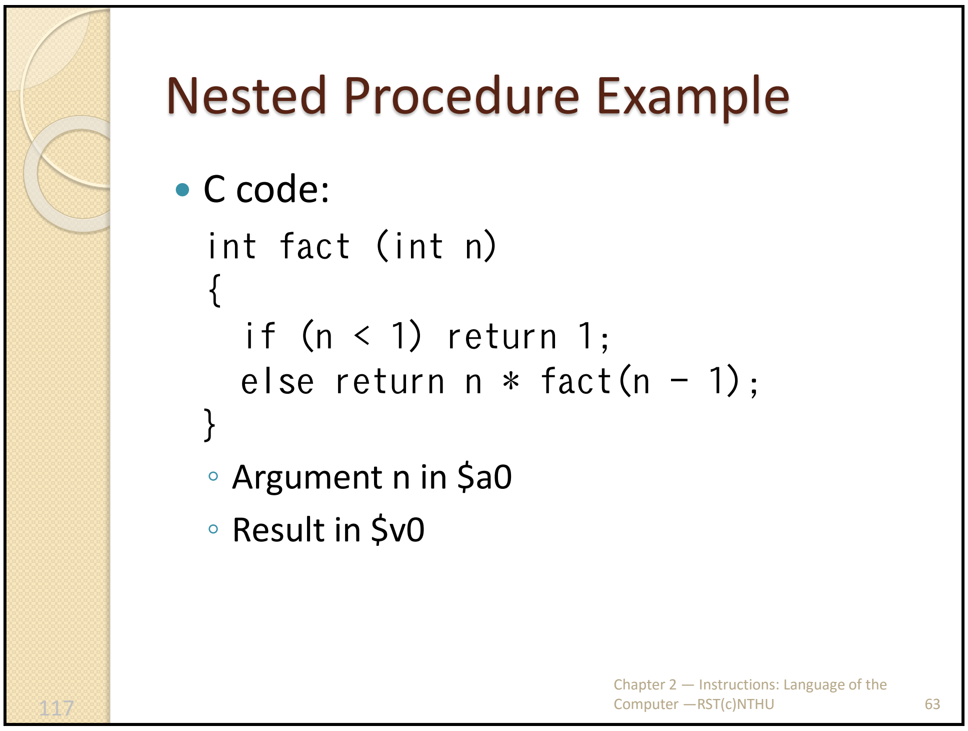


Non-Leaf Procedures

- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
  - Its return address
  - Any arguments and temporaries needed after the call
- Restore from the stack after the call

Rv	$v
Arg	$a
Temp	$t
Save	$s
Ra	$ra
Stack	$sp

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 62

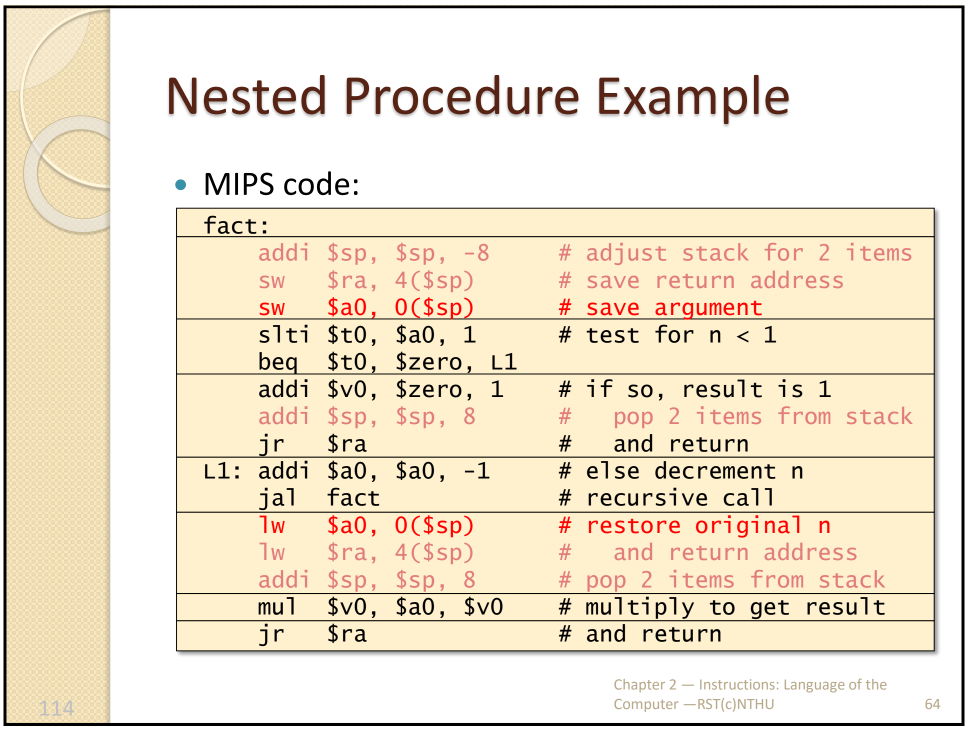


Nested Procedure Example

- C code:

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

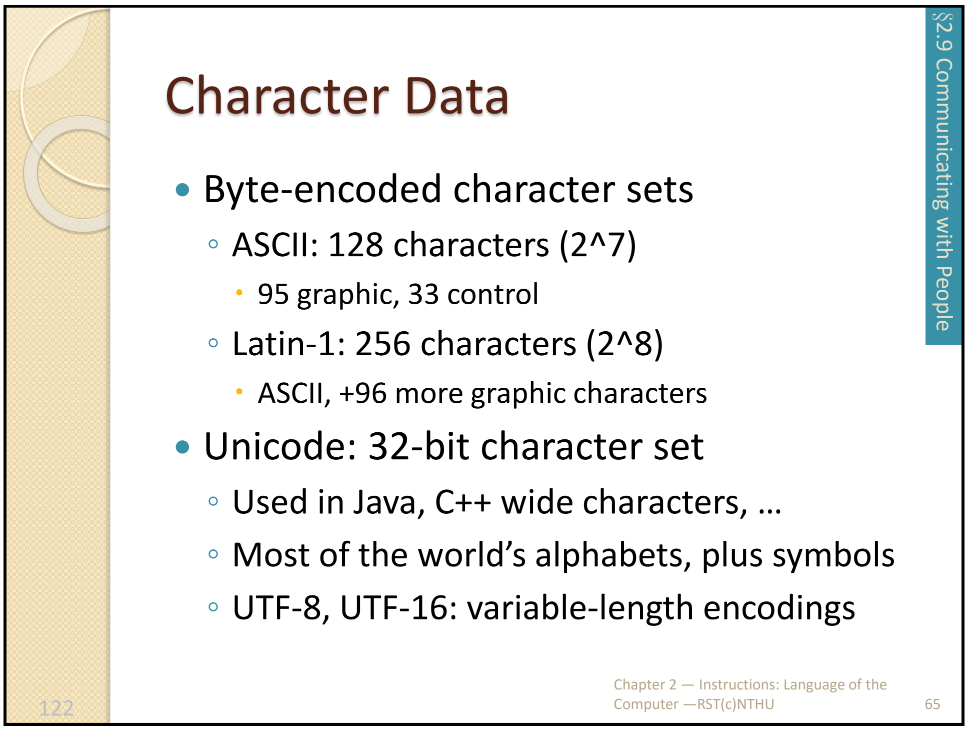
  - Argument n in $a0
  - Result in $v0



Nested Procedure Example

- MIPS code:

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



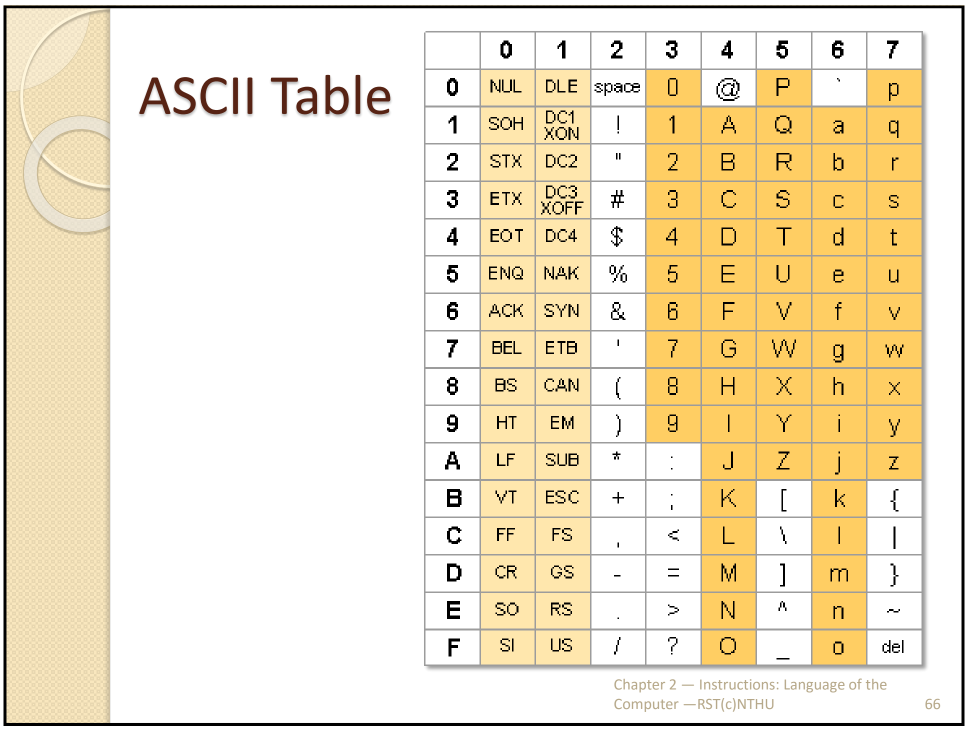
§2.9 Communicating with People

Character Data

- Byte-encoded character sets
  - ASCII: 128 characters (2^7)
    - 95 graphic, 33 control
  - Latin-1: 256 characters (2^8)
    - ASCII, +96 more graphic characters
- Unicode: 32-bit character set
  - Used in Java, C++ wide characters, …
  - Most of the world’s alphabets, plus symbols
  - UTF-8, UTF-16: variable-length encodings

122

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 65

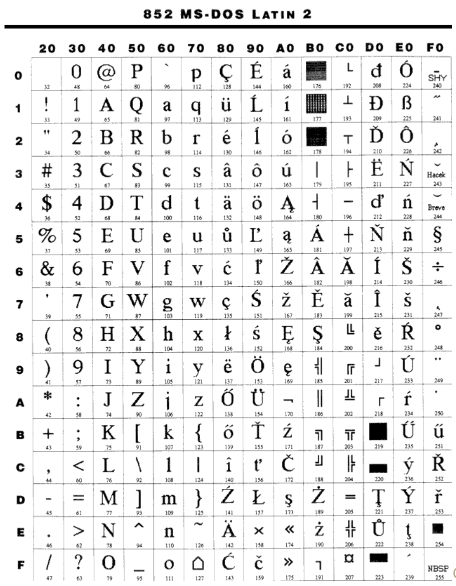


ASCII Table

	0	1	2	3	4	5	6	7
0	NUL	DLE	space	0	@	P	`	p
1	SOH	DC1 XON	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3 XOFF	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	del

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 66

Latin-2 Table



852 MS-DOS Latin 2

	20	30	40	50	60	70	80	90	A0	B0	C0	D0	E0	F0
0	32	0 48	@ 64	P 80	` 96	p 112	Ç 128	É 144	á 160	░ 176	└ 192	đ 208	Ó 224	SHY 240
1	! 33	1 49	A 65	Q 81	a 97	q 113	ü 129	Ĺ 145	í 161	▒ 177	┴ 193	Đ 209	ß 225	˝ 241
2	" 34	2 50	B 66	R 82	b 98	r 114	é 130	ĺ 146	ó 162	▓ 178	┬ 194	Ď 210	Ô 226	˛ 242
3	# 35	3 51	C 67	S 83	c 99	s 115	â 131	ô 147	ú 163	│ 179	├ 195	Ë 211	Ń 227	ˇ Hacek 243
4	$ 36	4 52	D 68	T 84	d 100	t 116	ä 132	ö 148	Ą 164	┤ 180	─ 196	ď 212	ń 228	˘ Breve 244
5	% 37	5 53	E 69	U 85	e 101	u 117	ů 133	Ľ 149	ą 165	Á 181	┼ 197	Ň 213	ň 229	§ 245
6	& 38	6 54	F 70	V 86	f 102	v 118	ć 134	ľ 150	Ž 166	Â 182	Ă 198	Í 214	Š 230	÷ 246
7	' 39	7 55	G 71	W 87	g 103	w 119	ç 135	Ś 151	ž 167	Ě 183	ă 199	Î 215	š 231	¸ 247
8	(40	8 56	H 72	X 88	h 104	x 120	ł 136	ś 152	Ę 168	Ş 184	╚ 200	ě 216	Ŕ 232	° 248
9	) 41	9 57	I 73	Y 89	i 105	y 121	ë 137	Ö 153	ę 169	╣ 185	╔ 201	┘ 217	Ú 233	¨ 249
A	* 42	: 58	J 74	Z 90	j 106	z 122	Ő 138	Ü 154	¬ 170	║ 186	╩ 202	┌ 218	ŕ 234	˙ 250
B	+ 43	; 59	K 75	[91	k 107	{ 123	ő 139	Ť 155	ź 171	╗ 187	╦ 203	█ 219	Ű 235	ű 251
C	, 44	< 60	L 76	\ 92	l 108	| 124	î 140	ť 156	Č 172	╝ 188	╠ 204	▄ 220	ý 236	Ř 252
D	- 45	= 61	M 77	] 93	m 109	} 125	Ź 141	Ł 157	ş 173	Ż 189	═ 205	Ţ 221	Ý 237	ř 253
E	. 46	> 62	N 78	^ 94	n 110	~ 126	Ä 142	× 158	« 174	ż 190	╬ 206	Ů 222	ţ 238	■ 254
F	/ 47	? 63	O 79	_ 95	o 111	⌂ 127	Ć 143	č 159	» 175	┐ 191	¤ 207	▀ 223	´ 239	NBSP 255

Byte/Halfword Operations

- Could use bitwise operations
- MIPS byte/halfword load/store
  - String processing is a common case

```
lb rt, offset(rs)      lh rt, offset(rs)
```

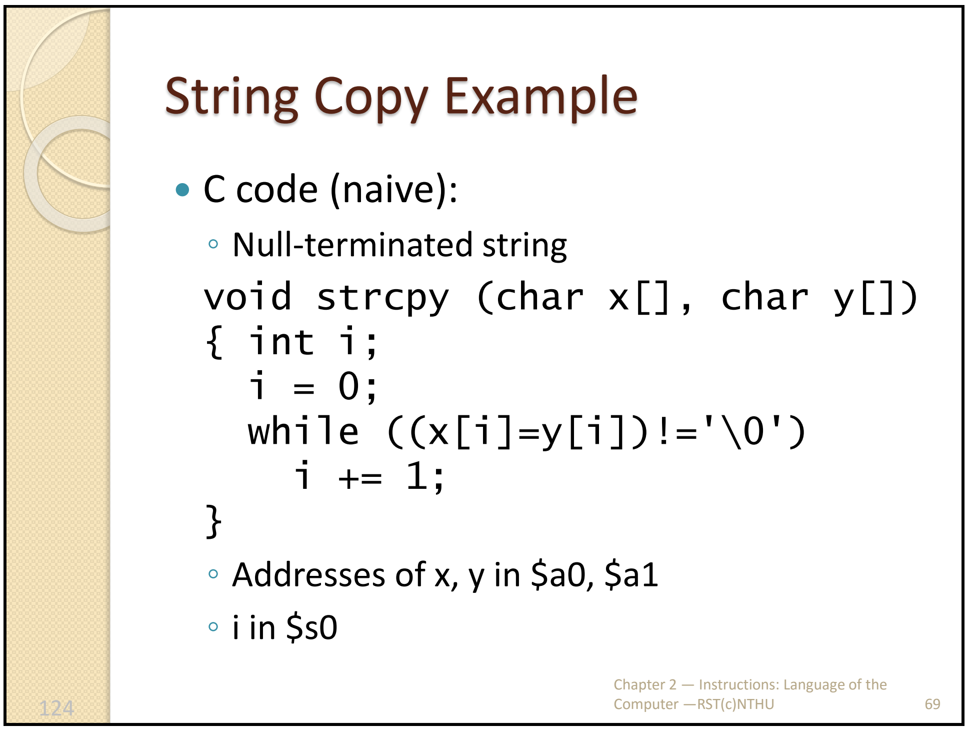
  - Sign extend to 32 bits in rt

```
lbu rt, offset(rs)     lhu rt, offset(rs)
```

  - Zero extend to 32 bits in rt

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

  - Store just rightmost byte/halfword



String Copy Example

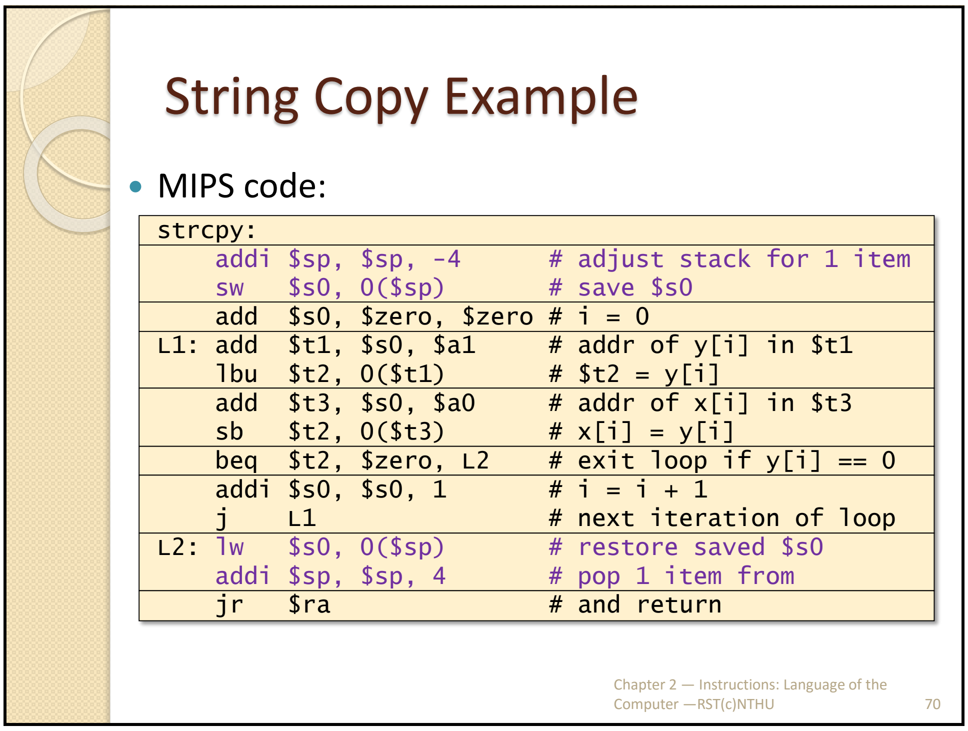
- C code (naive):
  - Null-terminated string

```
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

124

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 69

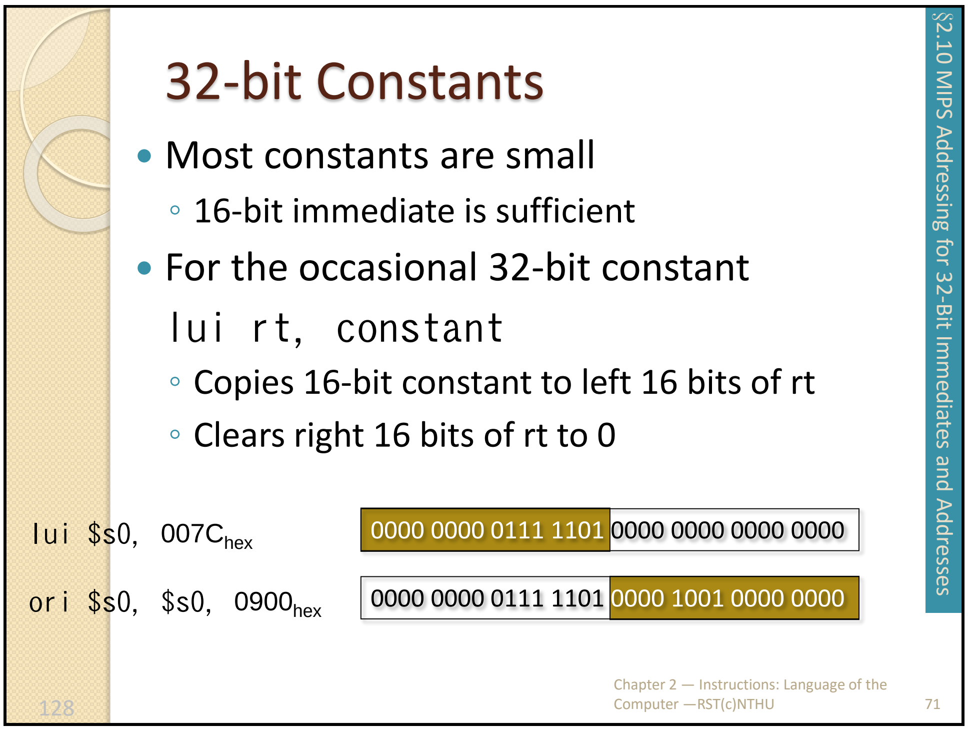


String Copy Example

- MIPS code:

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

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 70



32-bit Constants

- Most constants are small
  - 16-bit immediate is sufficient
- For the occasional 32-bit constant

```
lui rt, constant
```

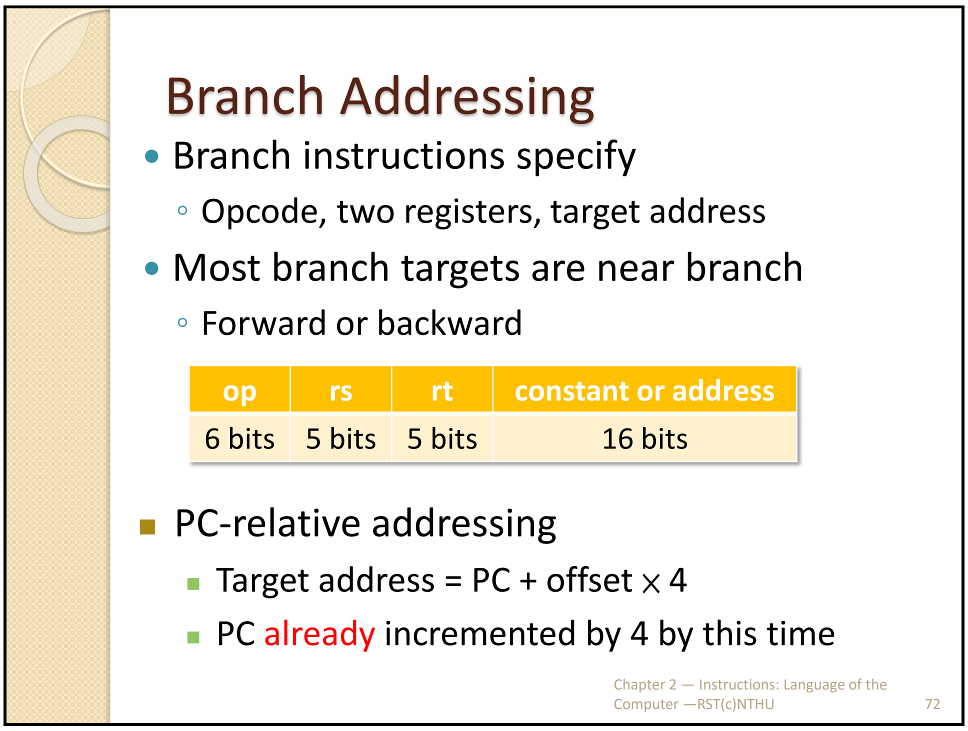
  - Copies 16-bit constant to left 16 bits of rt
  - Clears right 16 bits of rt to 0

lui \$s0, 007C$_{hex}$	0000 0000 0111 1101	0000 0000 0000 0000
ori \$s0, \$s0, 0900$_{hex}$	0000 0000 0111 1101	0000 1001 0000 0000

§2.10 MIPS Addressing for 32-Bit Immediates and Addresses

128

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 71



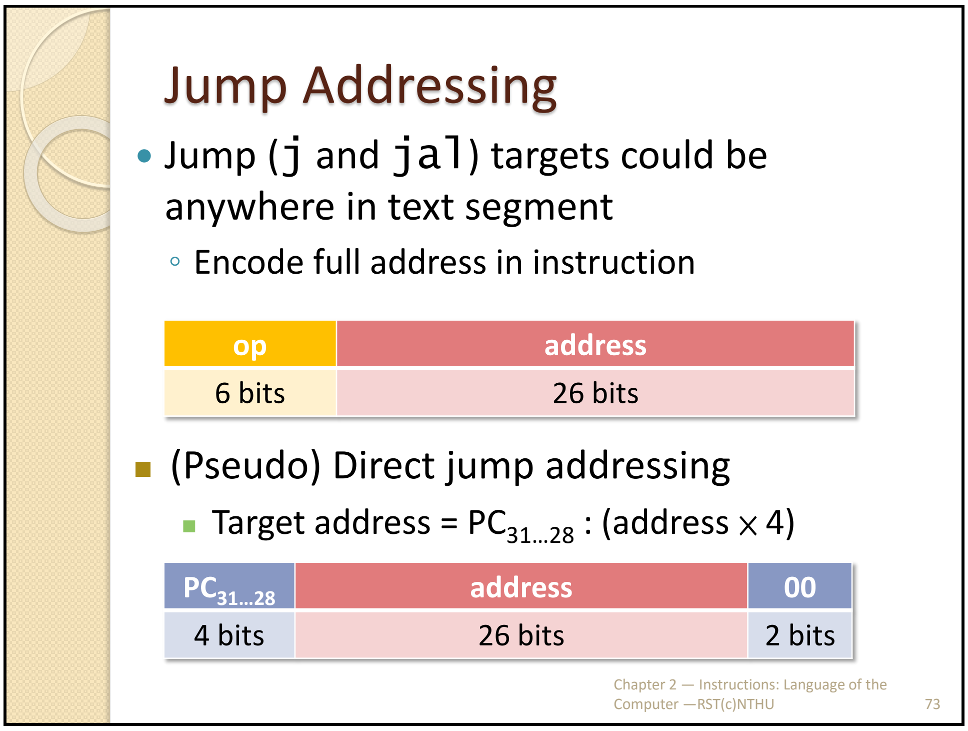
Branch Addressing

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

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

- PC-relative addressing
  - Target address = PC + offset $\times$ 4
  - PC already incremented by 4 by this time

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 72



Jump Addressing

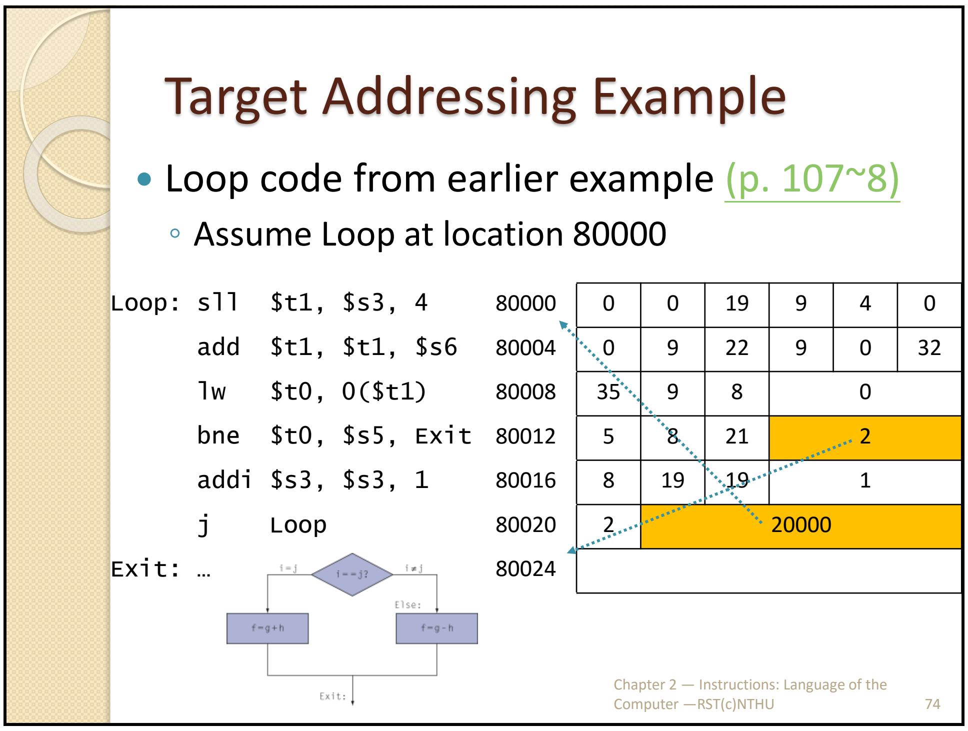
- Jump (`j` and `jal`) targets could be anywhere in text segment
  - Encode full address in instruction

op	address
6 bits	26 bits

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

$PC_{31\ldots28}$	address	00
4 bits	26 bits	2 bits

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 73

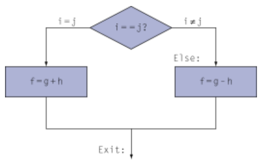


Target Addressing Example

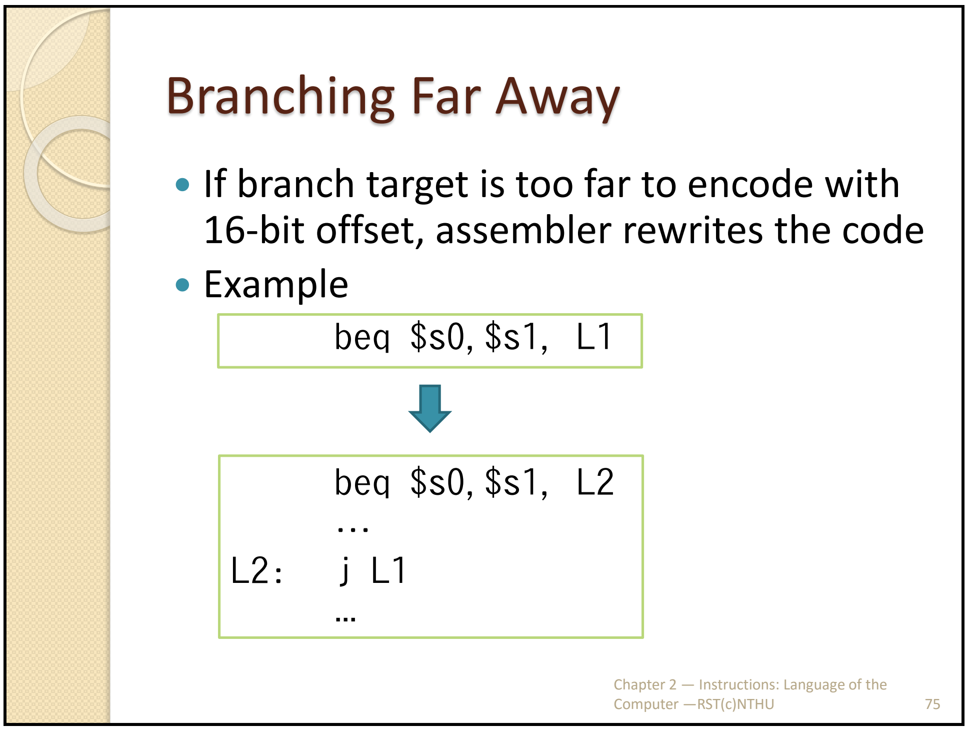
- Loop code from earlier example (p. 107~8)
  - Assume Loop at location 80000

```
Loop: sll  $t1, $s3, 4
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
Exit: …
```

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



Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 74



Branching Far Away

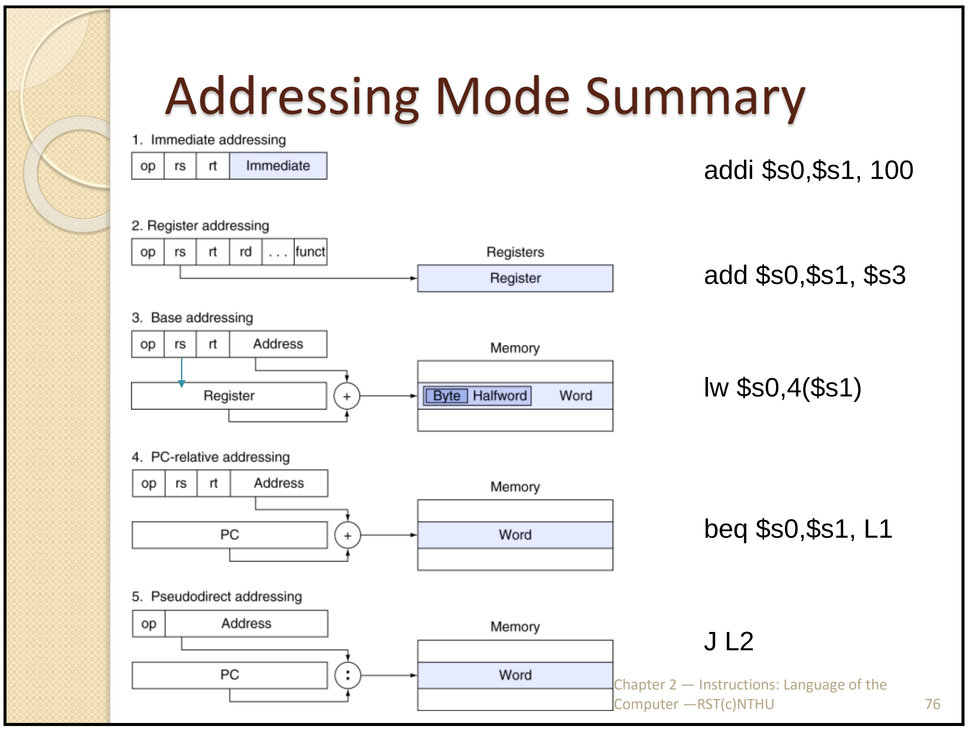
- If branch target is too far to encode with 16-bit offset, assembler rewrites the code
- Example

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

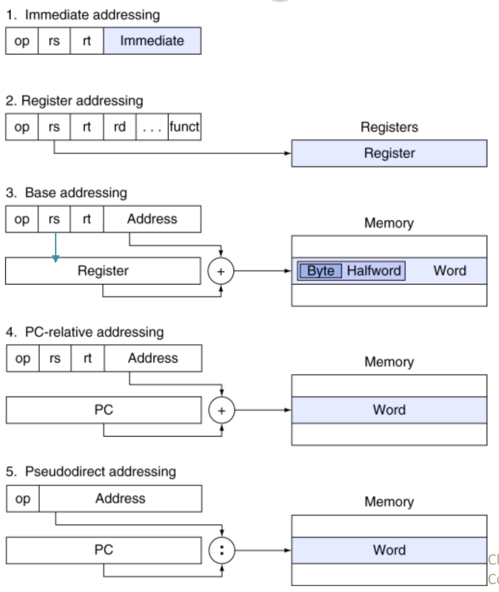
↓

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

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 75



Addressing Mode Summary



addi $s0,$s1, 100

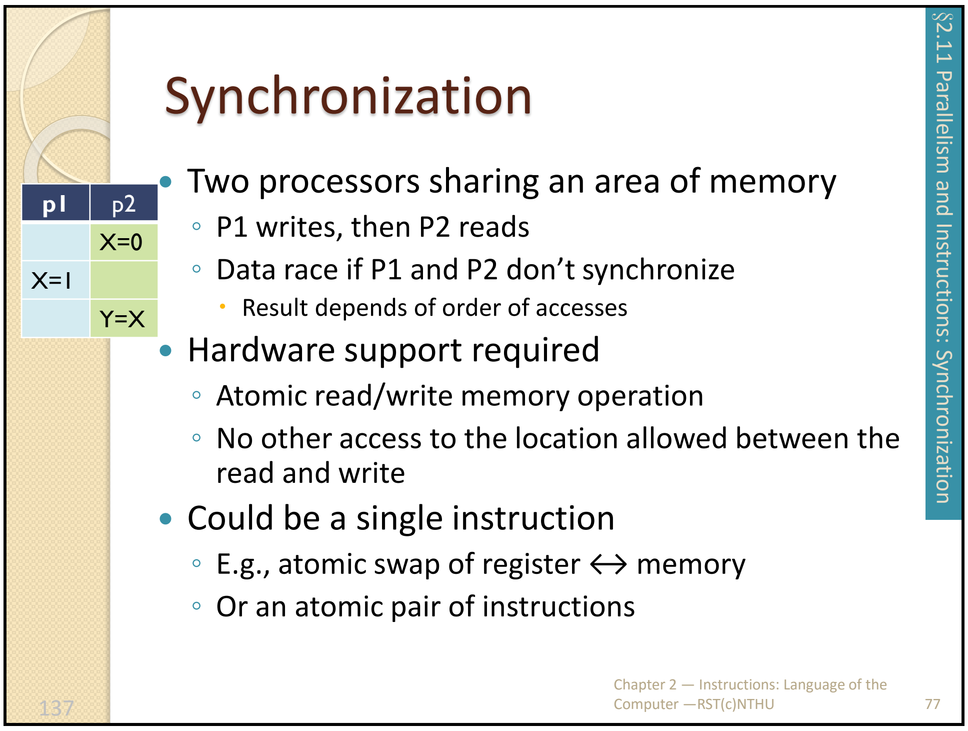
add $s0,$s1, $s3

lw $s0,4($s1)

beq $s0,$s1, L1

J L2

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 76



Synchronization

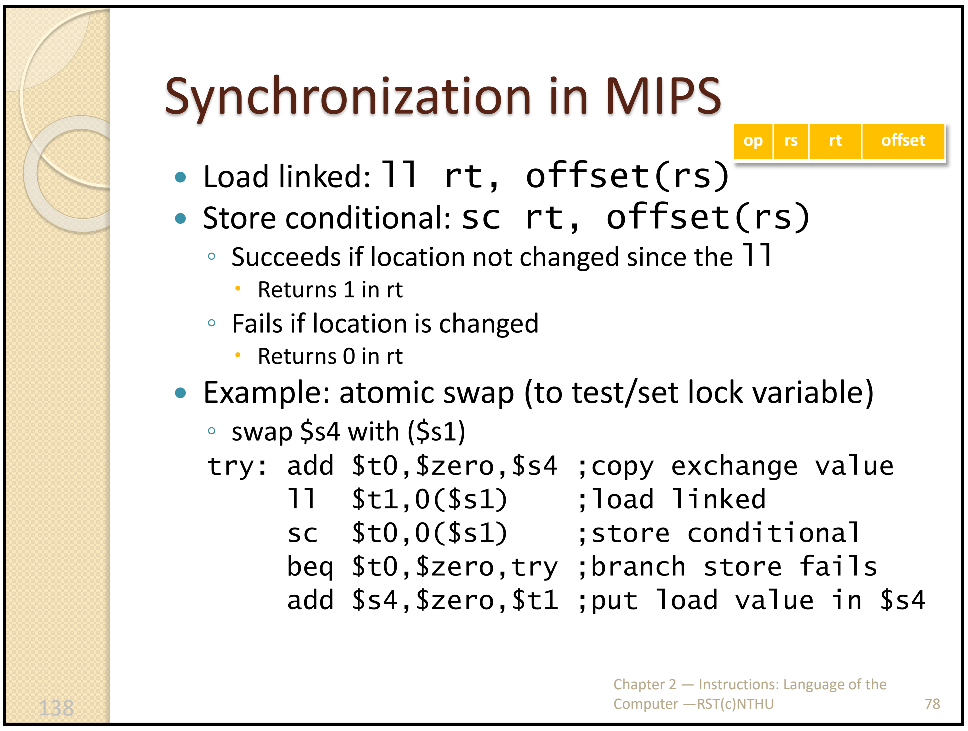
§2.11 Parallelism and Instructions: Synchronization

pI	p2
	X=0
X=I	
	Y=X

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

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 77

137



Synchronization in MIPS

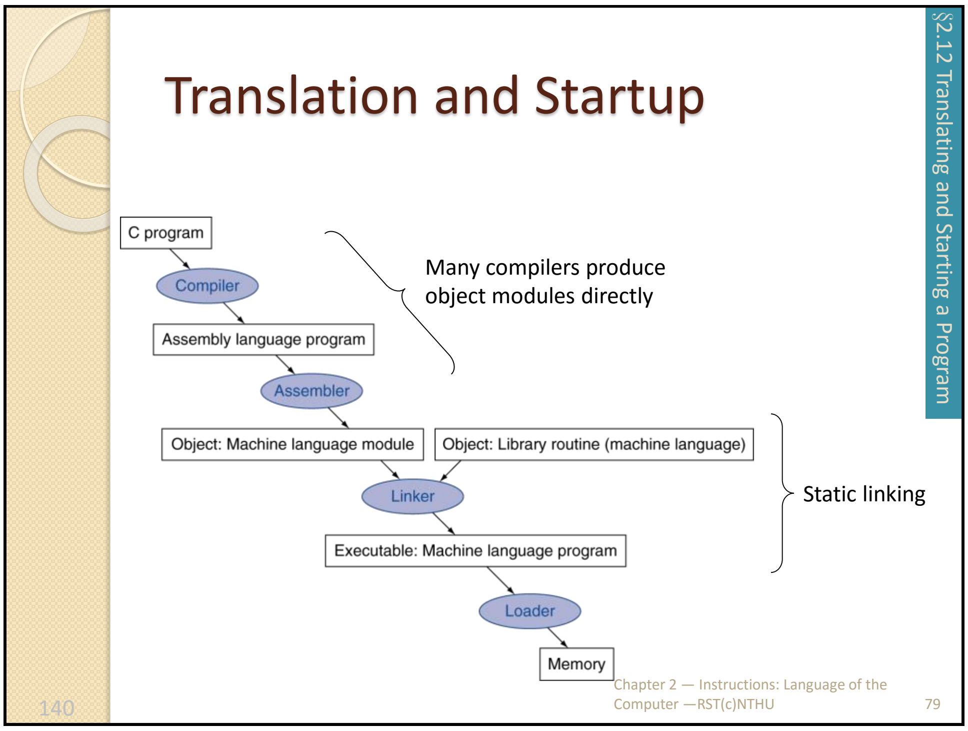
op	rs	rt	offset

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

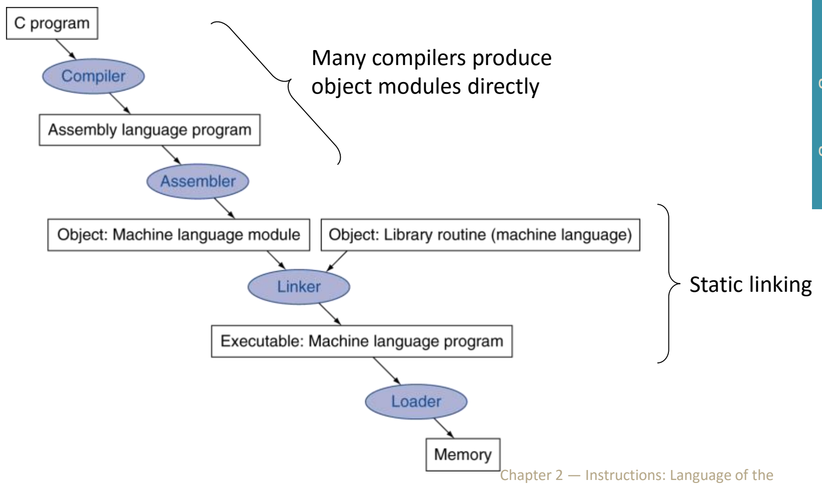
Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 78

138



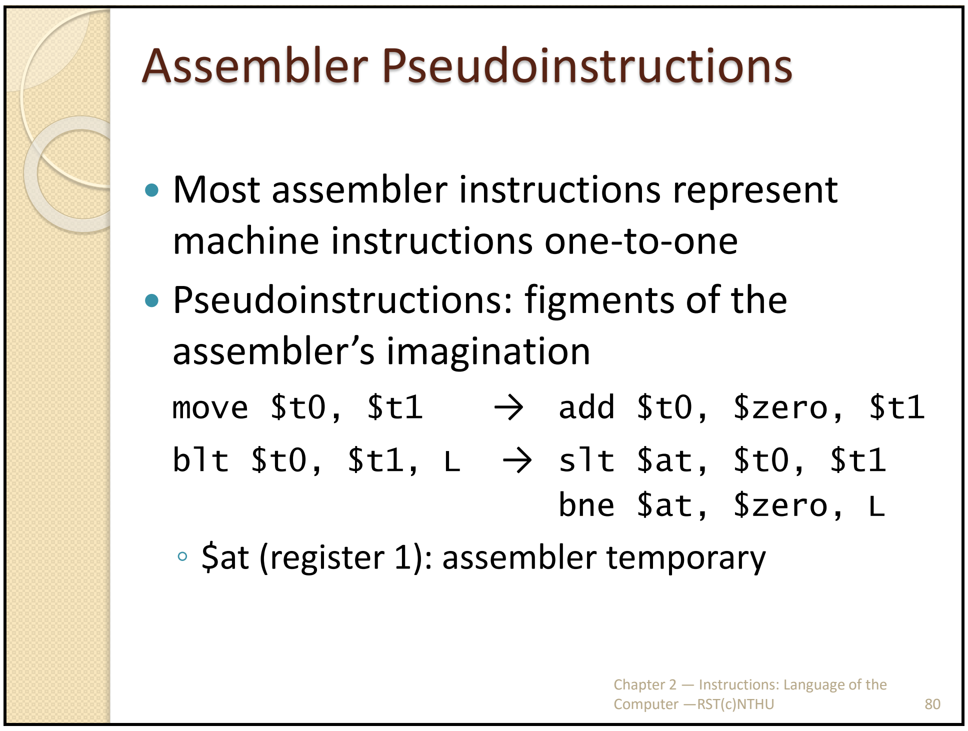
Translation and Startup

§2.12 Translating and Starting a Program


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

140

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 79



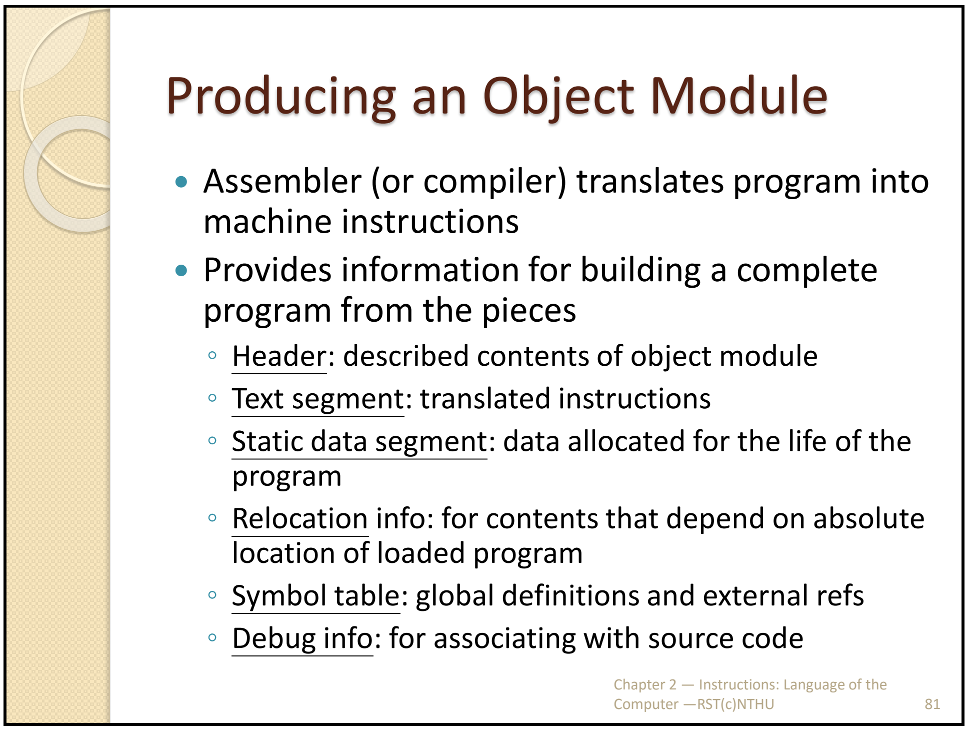
Assembler Pseudoinstructions

- 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
```

  - $at (register 1): assembler temporary

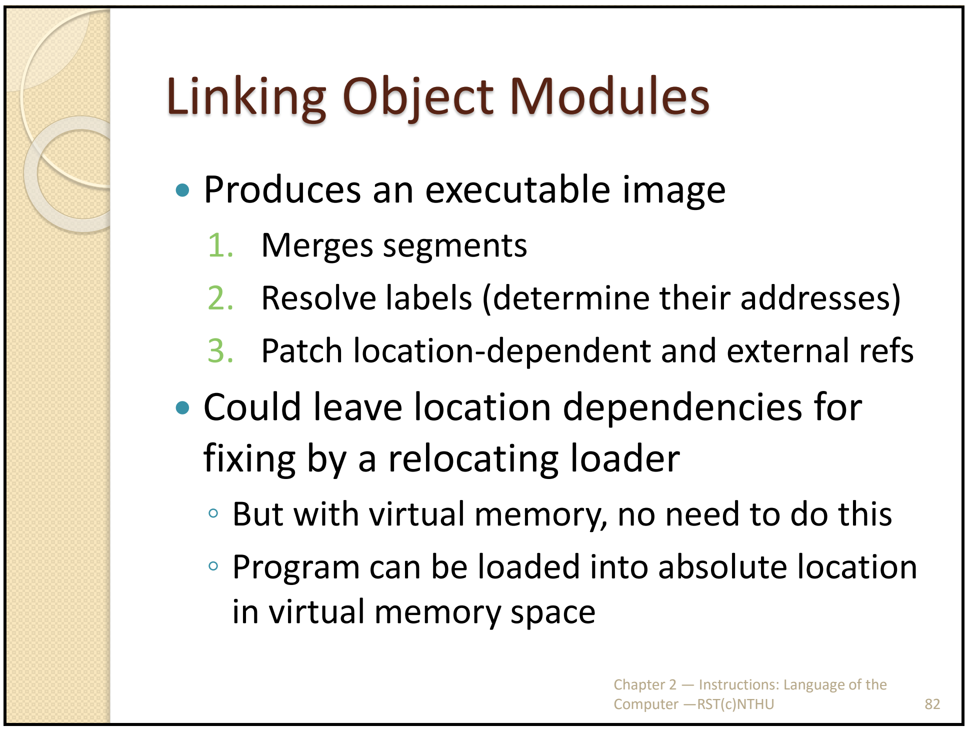
Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 80



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

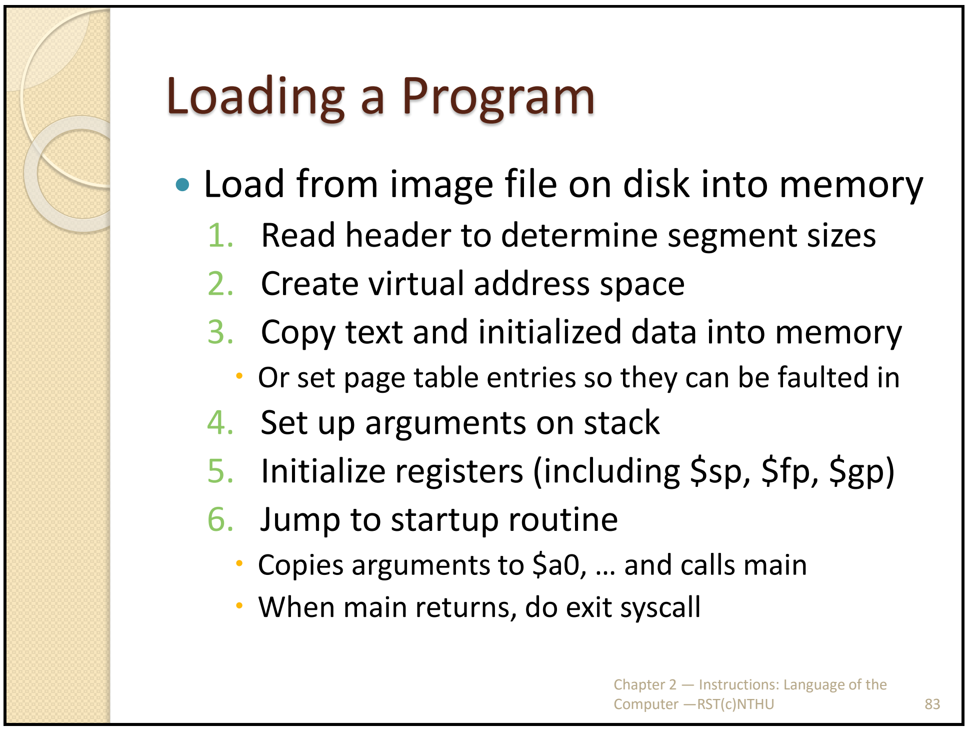
Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 81



Linking Object Modules

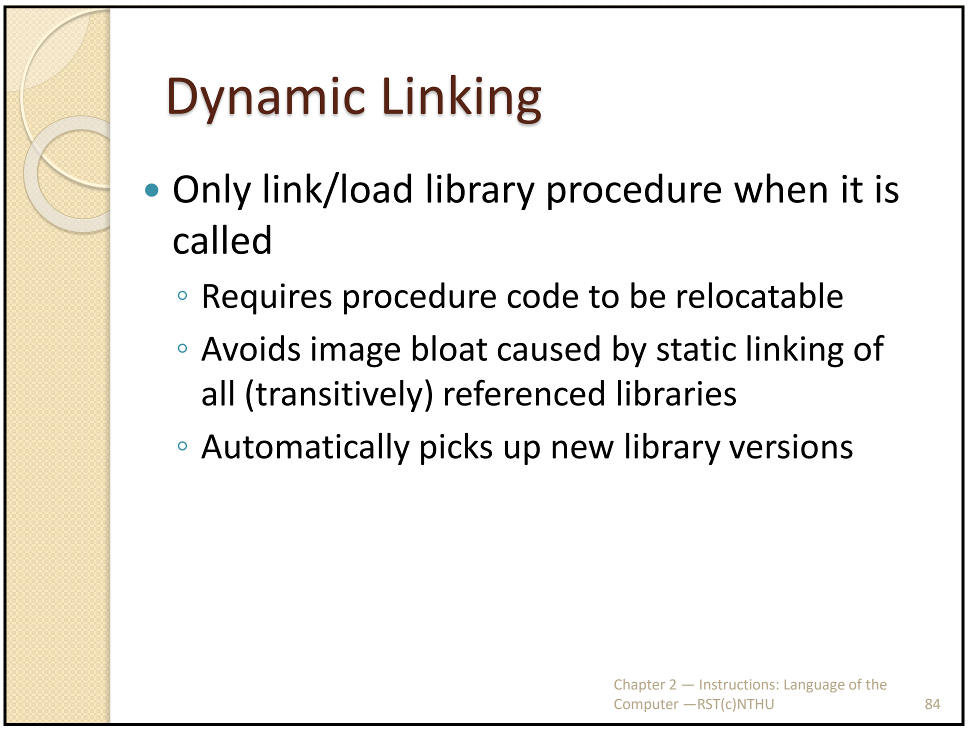
- Produces an executable image
  1. Merges segments
  2. Resolve labels (determine their addresses)
  3. Patch location-dependent and external refs
- Could leave location dependencies for fixing by a relocating loader
  - But with virtual memory, no need to do this
  - Program can be loaded into absolute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

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 82



Loading a Program

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



Dynamic Linking

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

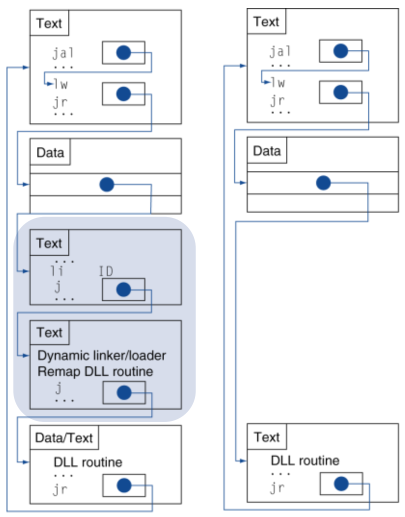
Lazy Linkage

Indirection table

Stub: Loads routine ID,
Jump to linker/loader

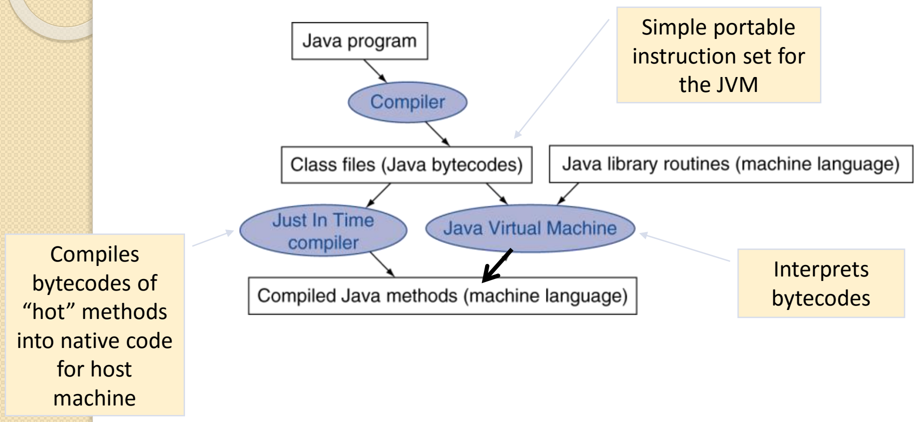
Linker/loader code

Dynamically
mapped code

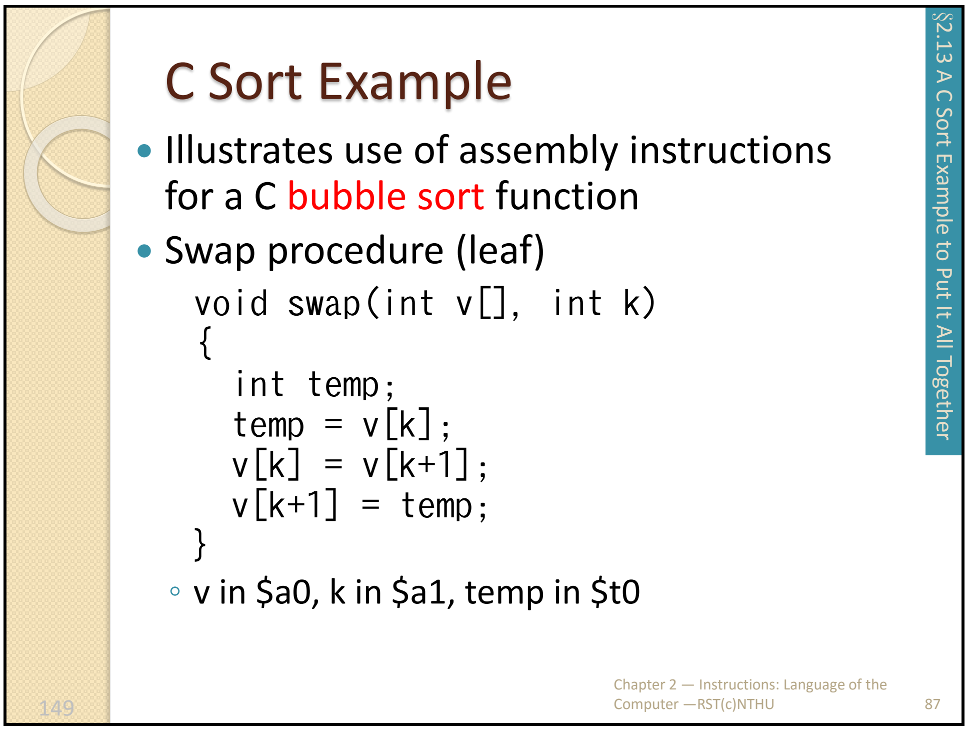


a. First call to DLL routine   b. Subsequent calls to DLL routine

Starting Java Applications


Java program
Compiler
Class files (Java bytecodes)
Java library routines (machine language)
Just In Time compiler
Java Virtual Machine
Compiled Java methods (machine language)
Simple portable instruction set for the JVM
Compiles bytecodes of "hot" methods into native code for host machine
Interprets bytecodes

Note: For more details, please refer to Sec. 2.15 on the CD



C Sort Example

- Illustrates use of assembly instructions for a C bubble sort function
- Swap procedure (leaf)

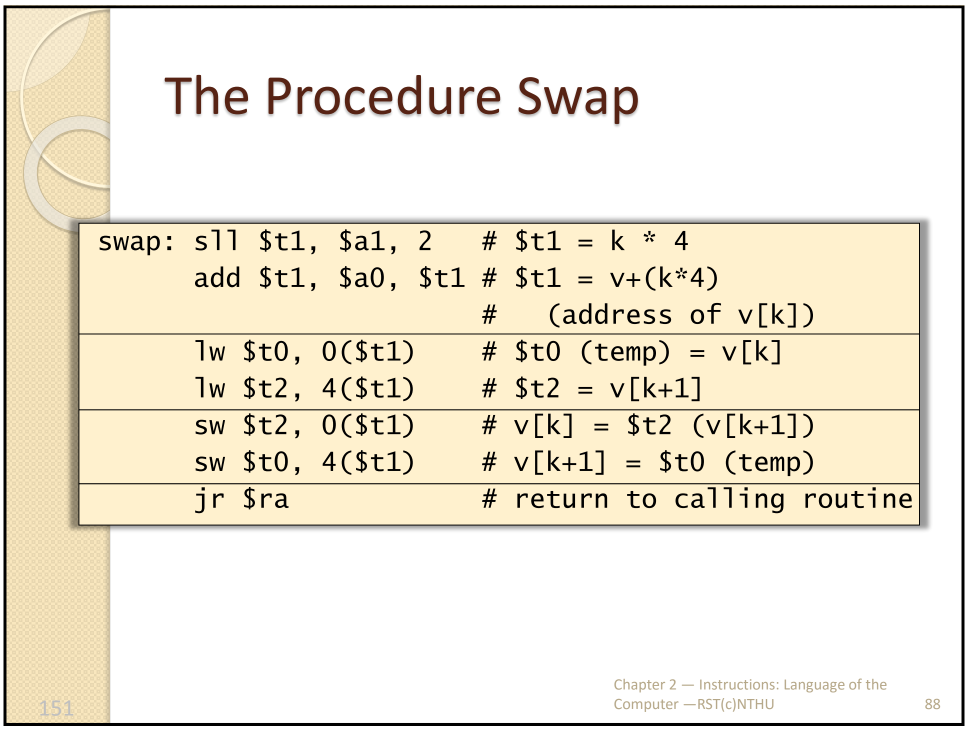
```
void swap(int v[], int k)
{
  int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

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

§2.13 A C Sort Example to Put It All Together

149

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 87

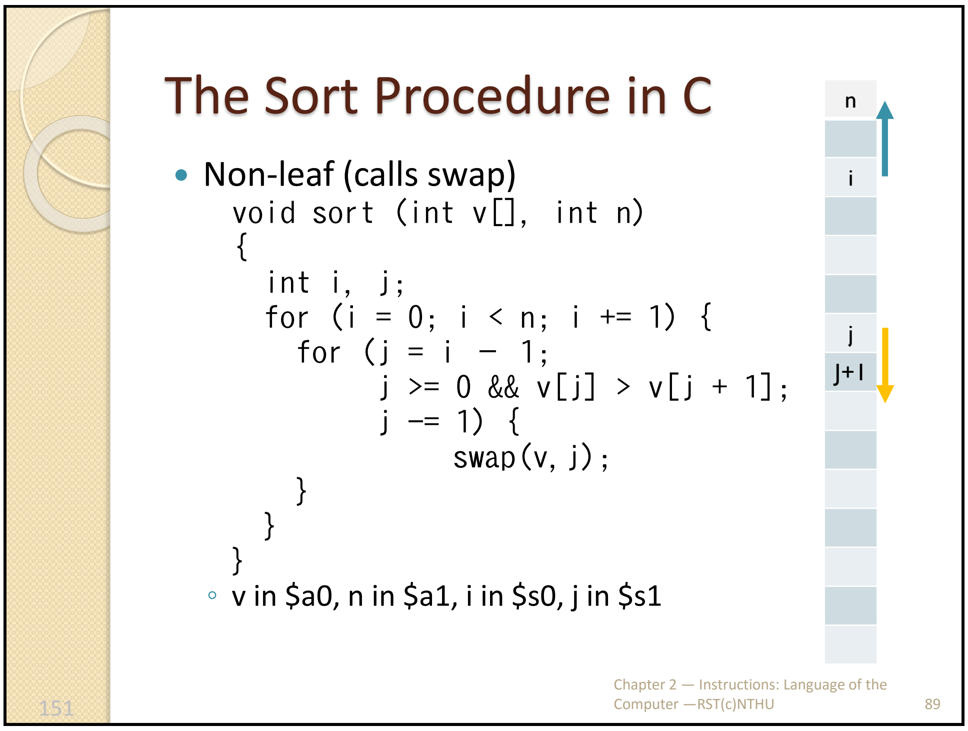


The Procedure Swap

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

151

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 88

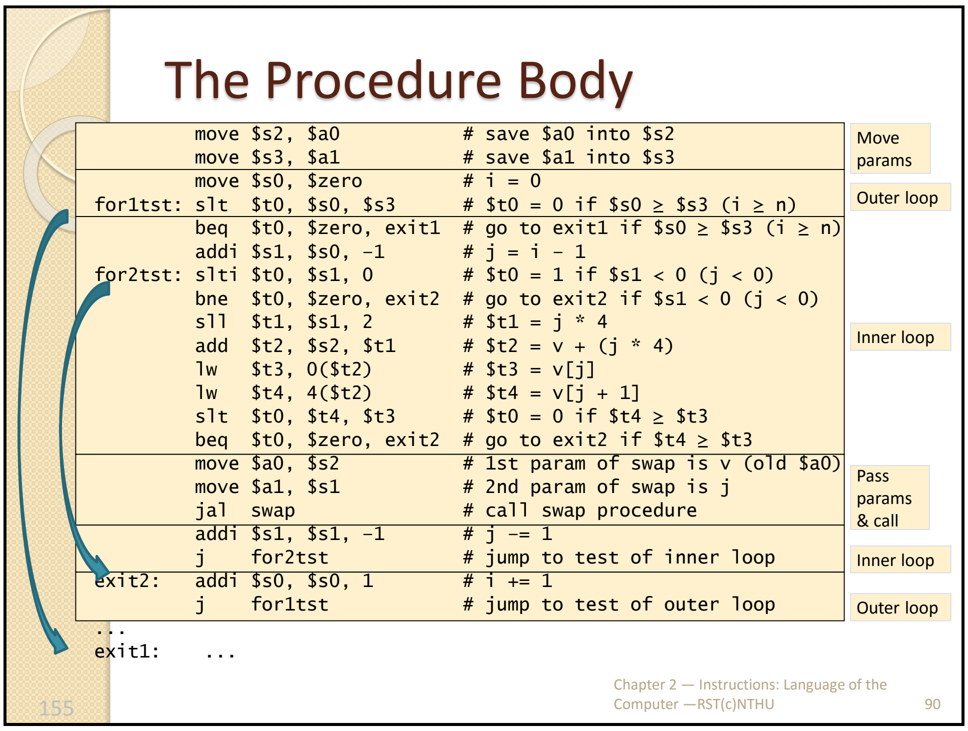


The Sort Procedure in C

- Non-leaf (calls swap)

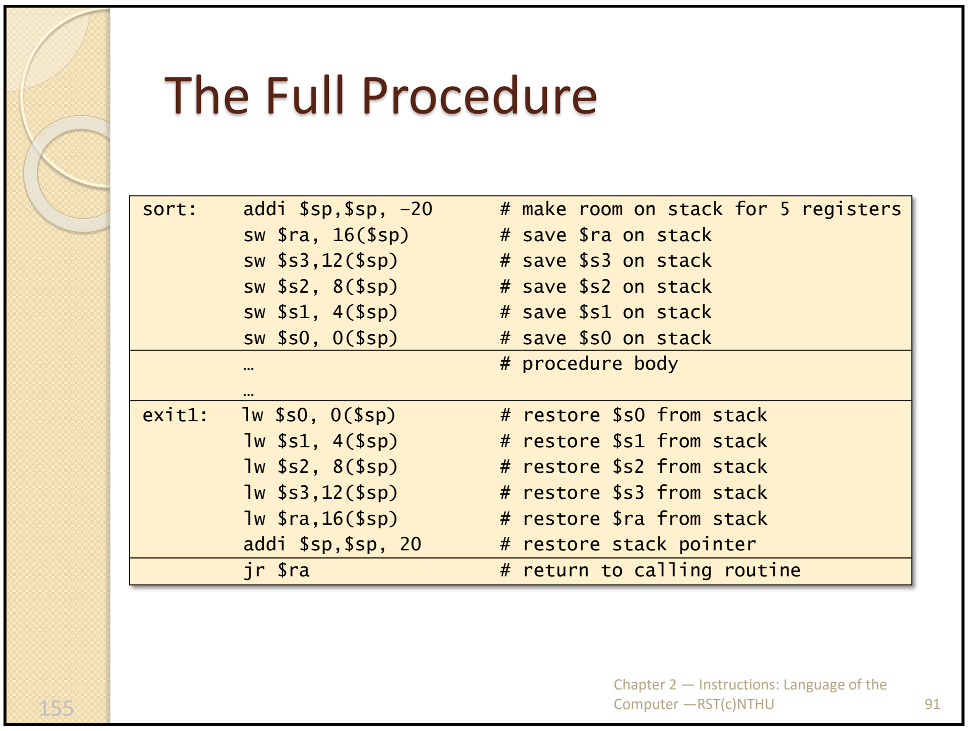
```
void sort (int v[], int n)
{
  int i, j;
  for (i = 0; i < n; i += 1) {
    for (j = i – 1;
         j >= 0 && v[j] > v[j + 1];
         j -= 1) {
           swap(v, j);
    }
  }
}
```

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



The Procedure Body

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

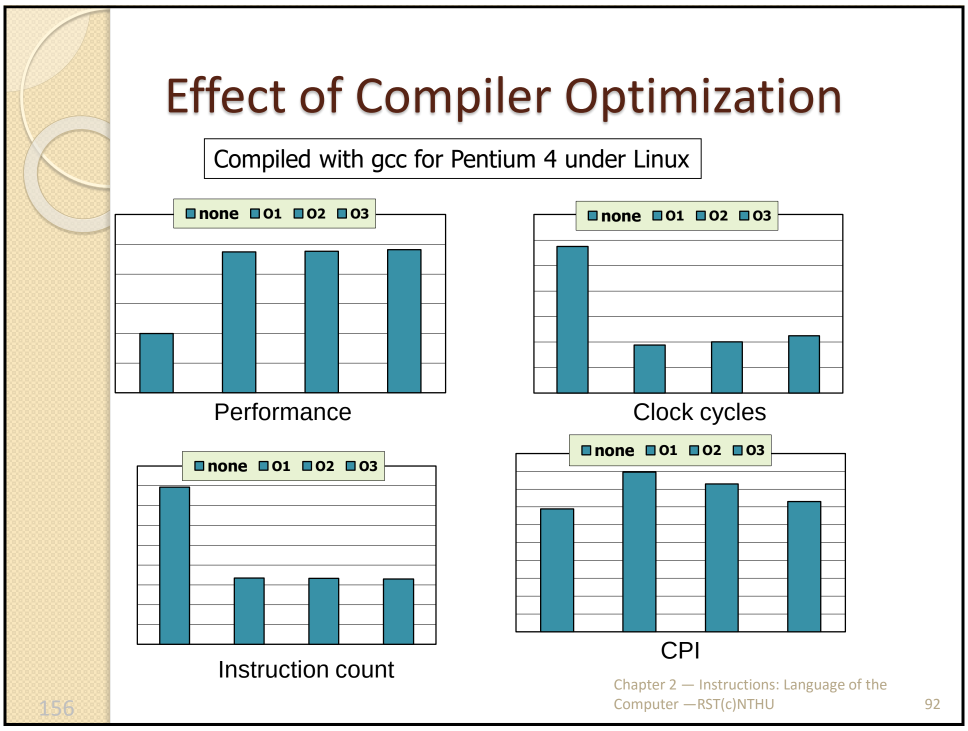


The Full Procedure

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

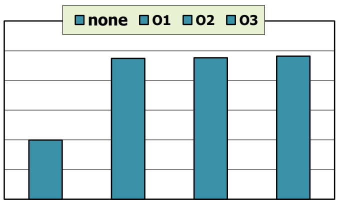
Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 91

155

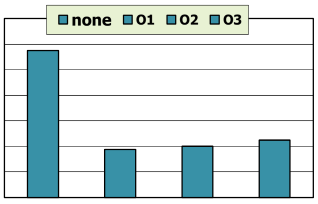


Effect of Compiler Optimization

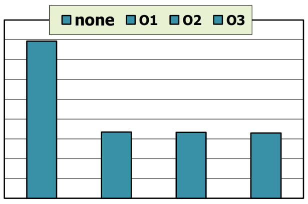
Compiled with gcc for Pentium 4 under Linux



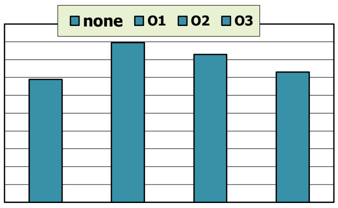
Performance



Clock cycles



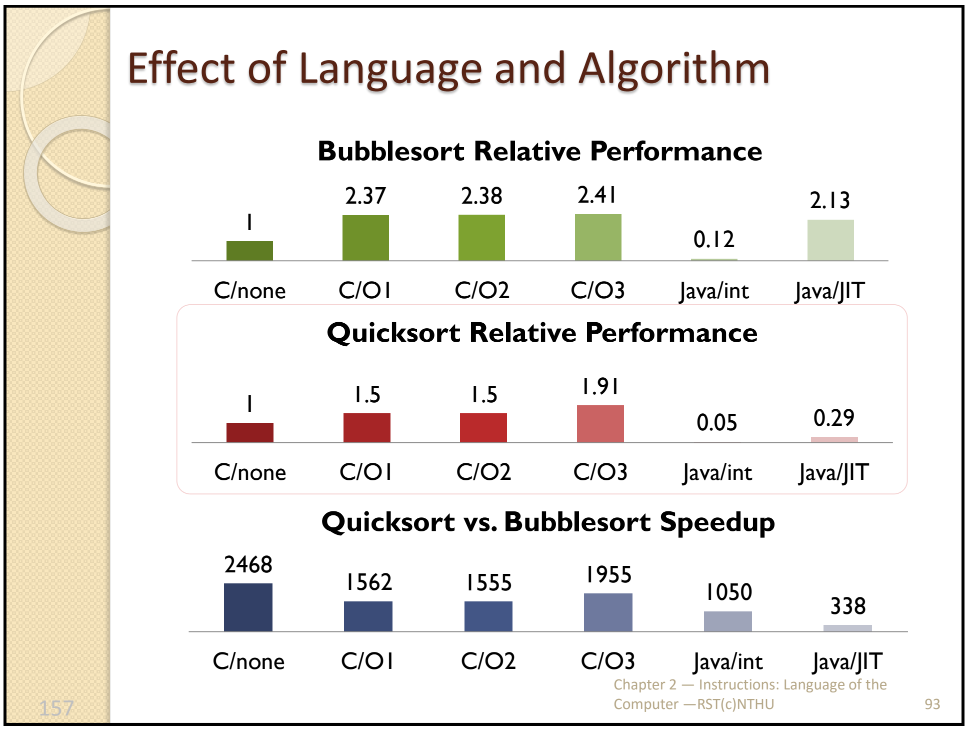
Instruction count



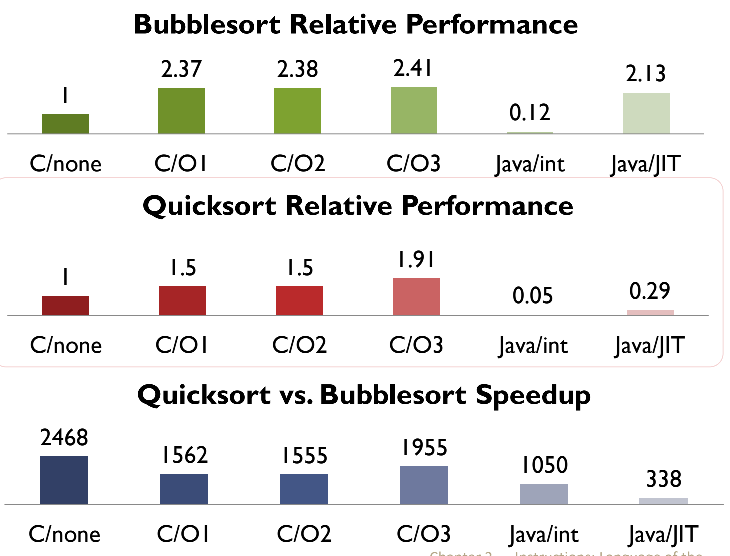
CPI

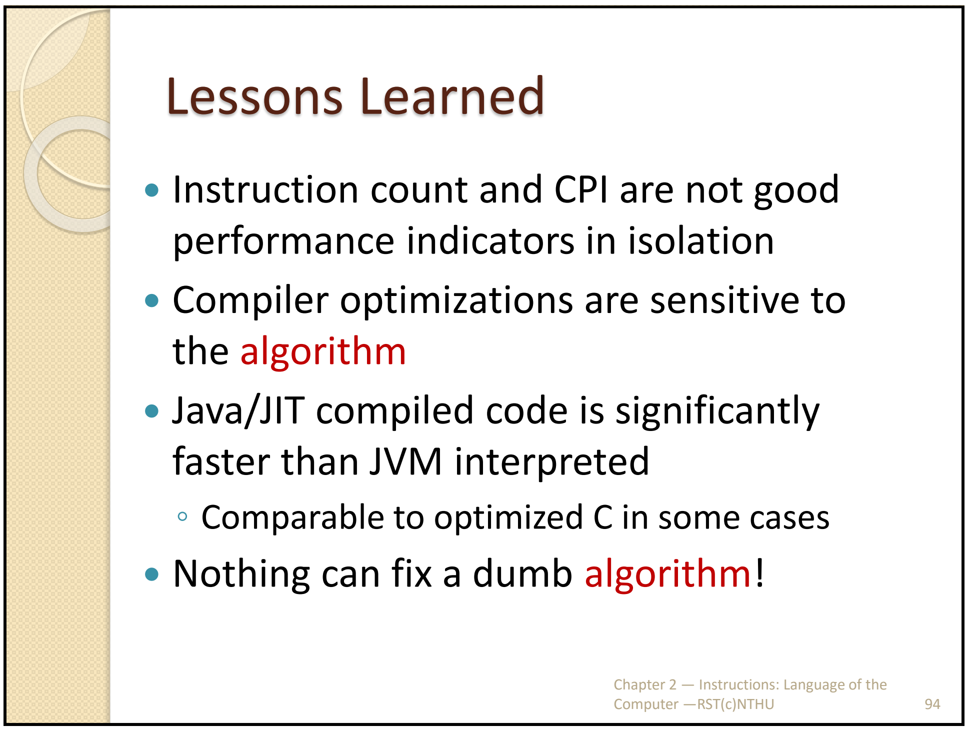
Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 92

156



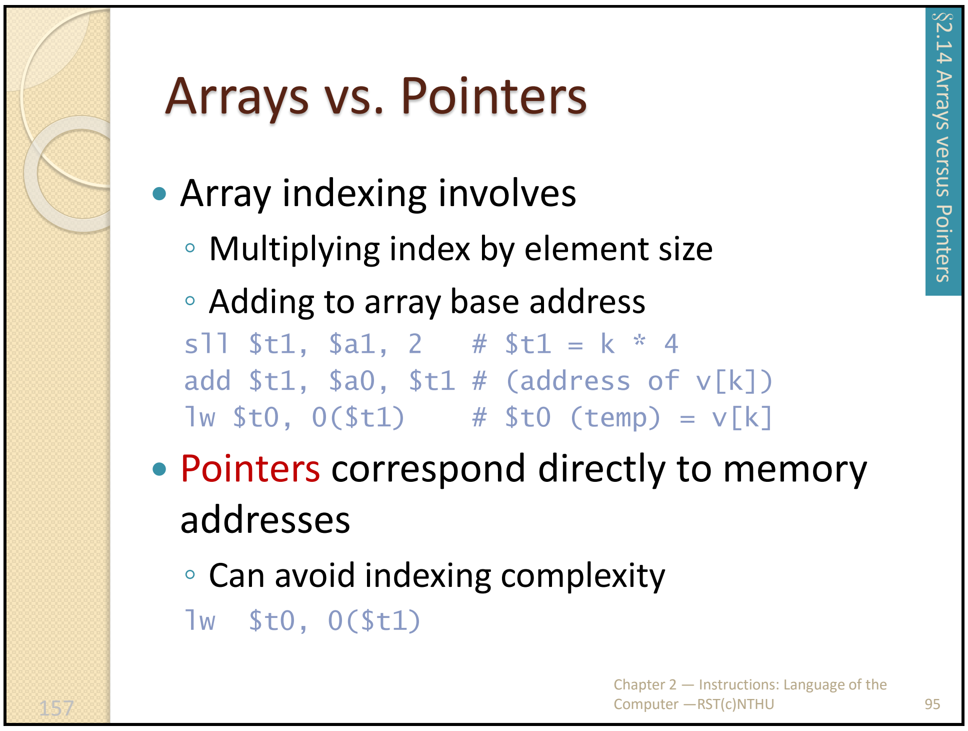
Effect of Language and Algorithm





Lessons Learned

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



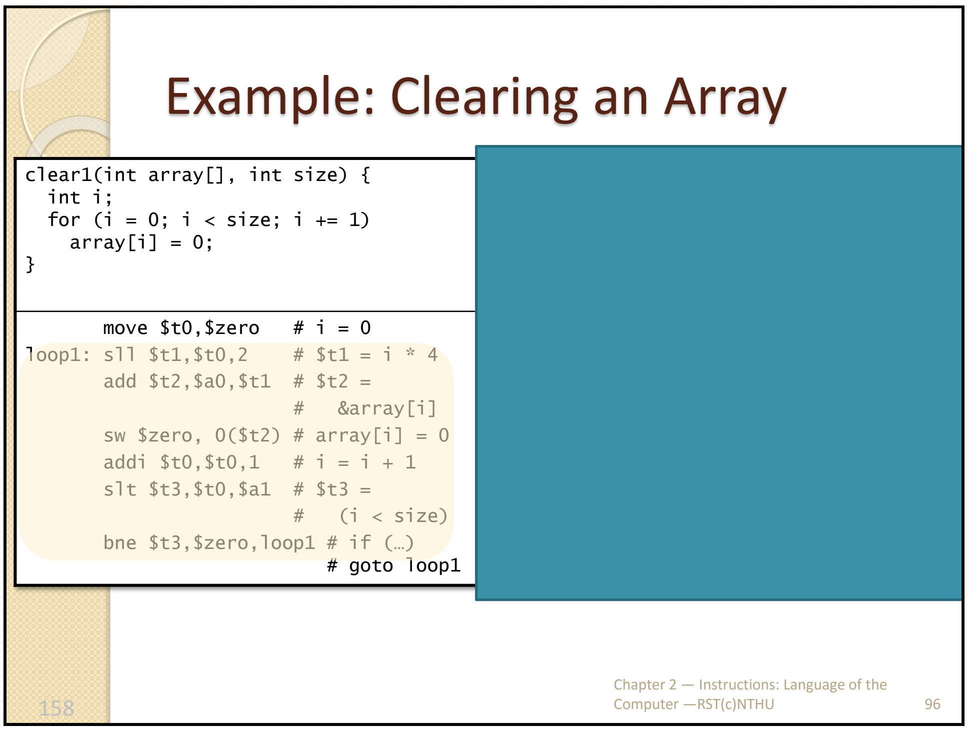
Arrays vs. Pointers

- Array indexing involves
  - Multiplying index by element size
  - Adding to array base address

```
sll $t1, $a1, 2   # $t1 = k * 4
add $t1, $a0, $t1 # (address of v[k])
lw $t0, 0($t1)    # $t0 (temp) = v[k]
```

- Pointers correspond directly to memory addresses
  - Can avoid indexing complexity

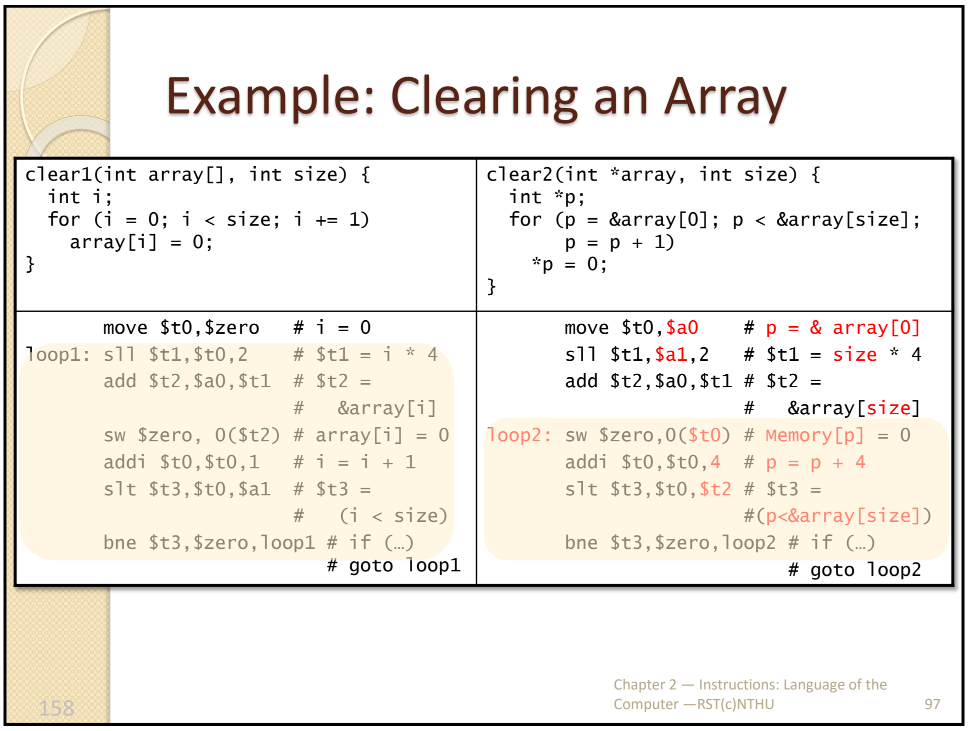
```
lw  $t0, 0($t1)
```



Example: Clearing an Array

```
clear1(int array[], int size) {
  int i;
  for (i = 0; i < size; i += 1)
    array[i] = 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 (…)
                           # goto loop1
```



Example: Clearing an Array

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

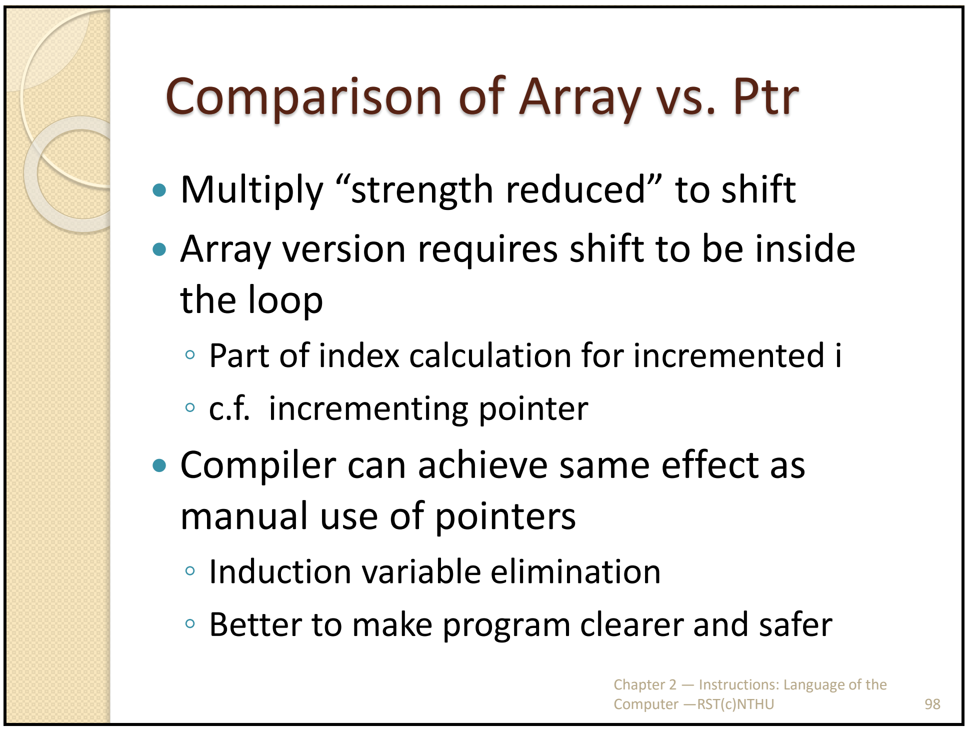
```
clear2(int *array, int size) {
  int *p;
  for (p = &array[0]; p < &array[size];
       p = p + 1)
    *p = 0;
}
```

```
       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 (…)
                           # 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 (…)
                           # goto loop2
```

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 97

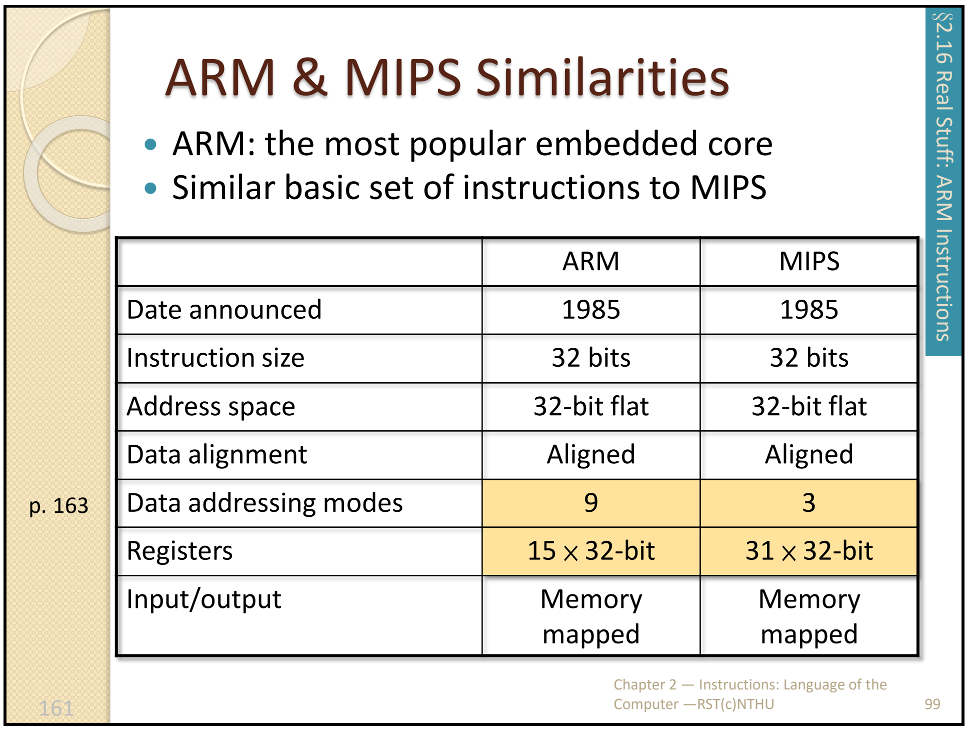
158



Comparison of Array vs. Ptr

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

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 98



ARM & MIPS Similarities

§2.16 Real Stuff: ARM Instructions

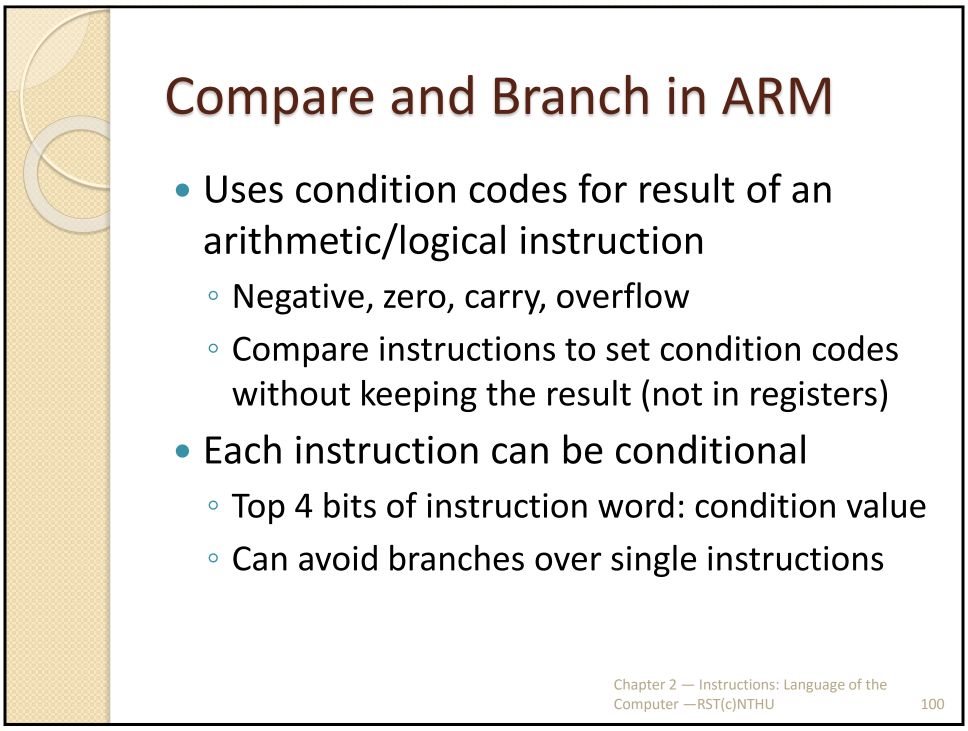
- ARM: the most popular embedded core
- Similar basic set of instructions to MIPS

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

p. 163

161

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 99

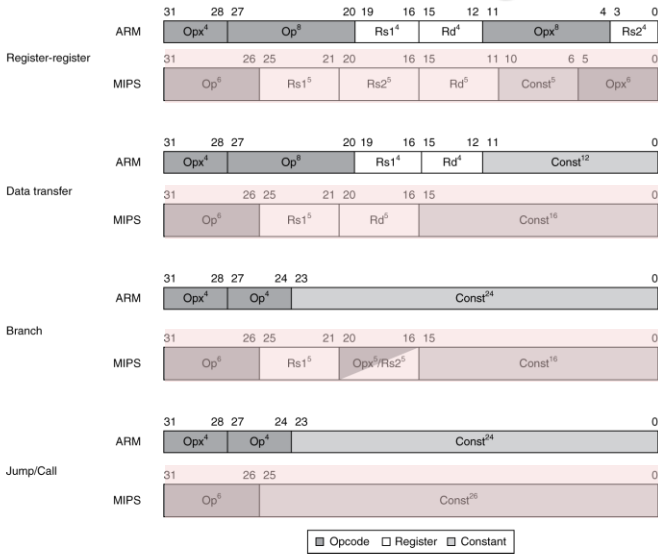


Compare and Branch in ARM

- Uses condition codes for result of an arithmetic/logical instruction
  - Negative, zero, carry, overflow
  - Compare instructions to set condition codes without keeping the result (not in registers)
- Each instruction can be conditional
  - Top 4 bits of instruction word: condition value
  - Can avoid branches over single instructions

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 100

Instruction Encoding

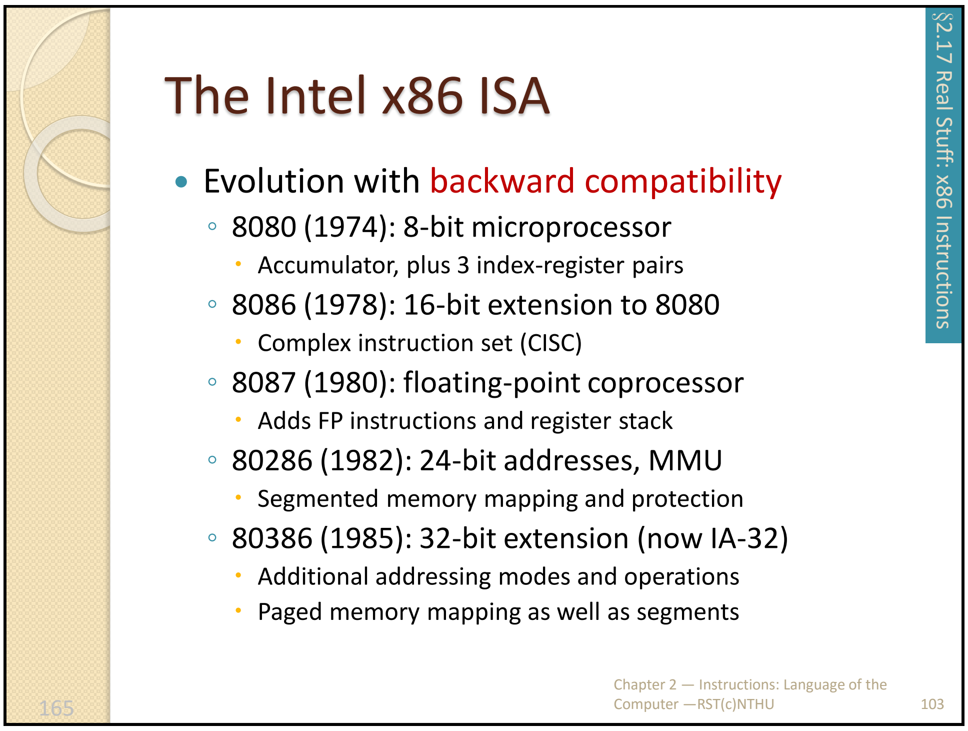


Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 101

Intel 4004 4-bit CPU

- 1971~1981
- the first complete CPU on one chip
- 46K~92K instructions per second
- Maximum clock speed was 740 kHz
- A single multiplexed 4-bit bus for transferring:
  - 12-bit addresses
  - 8-bit instructions
  - 4-bit data words
- 46 instructions (of which 41 were 8 bits wide and 5 were 16 bits wide)
- The register set has 16 registers of 4 bits each

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 102



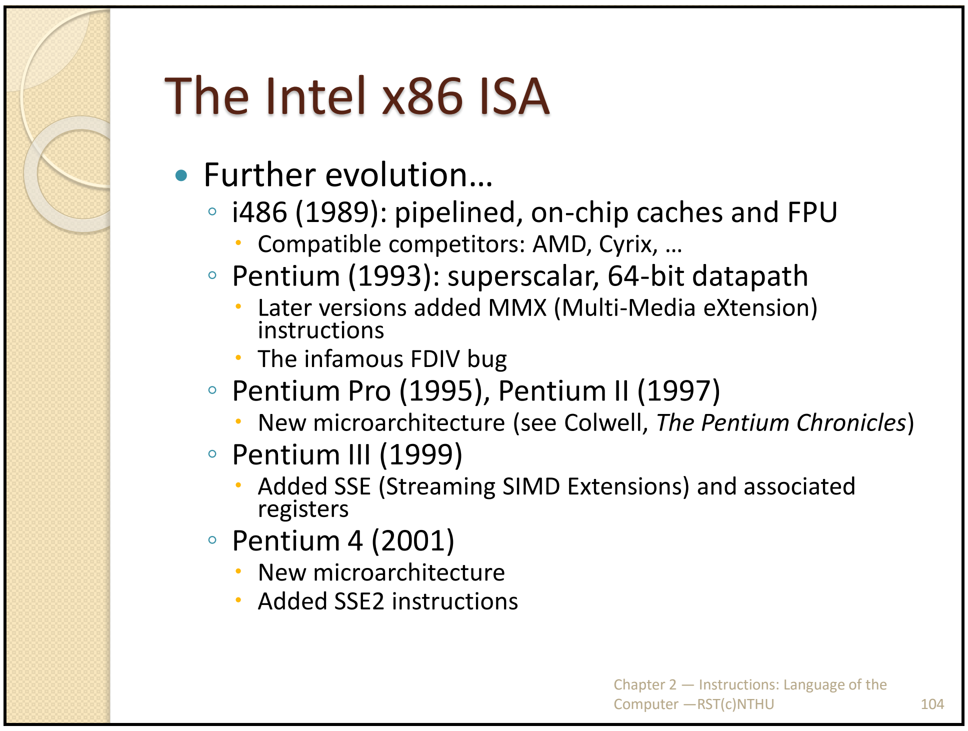
The Intel x86 ISA

§2.17 Real Stuff: x86 Instructions

- 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

165

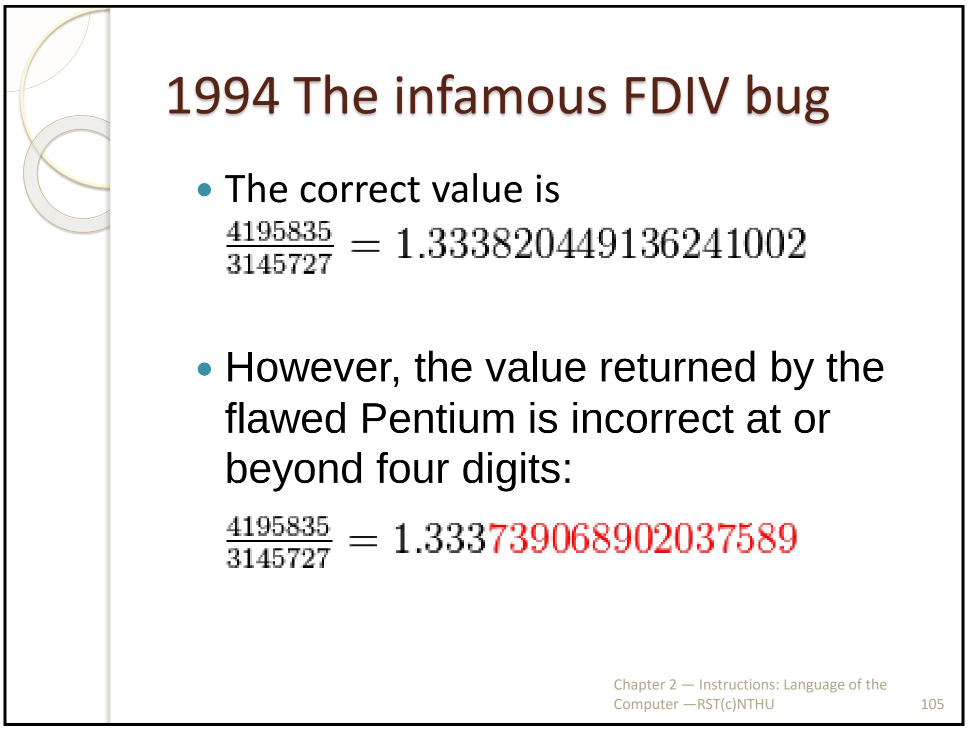
Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 103



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

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 104



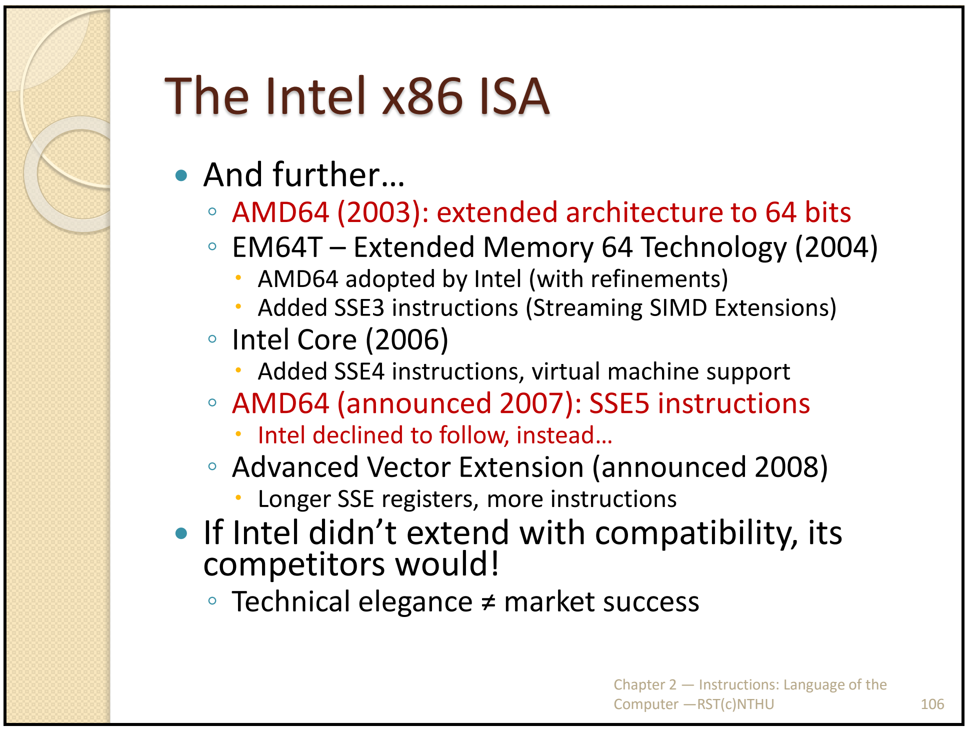
1994 The infamous FDIV bug

- The correct value is

$$\frac{4195835}{3145727} = 1.333820449136241002$$

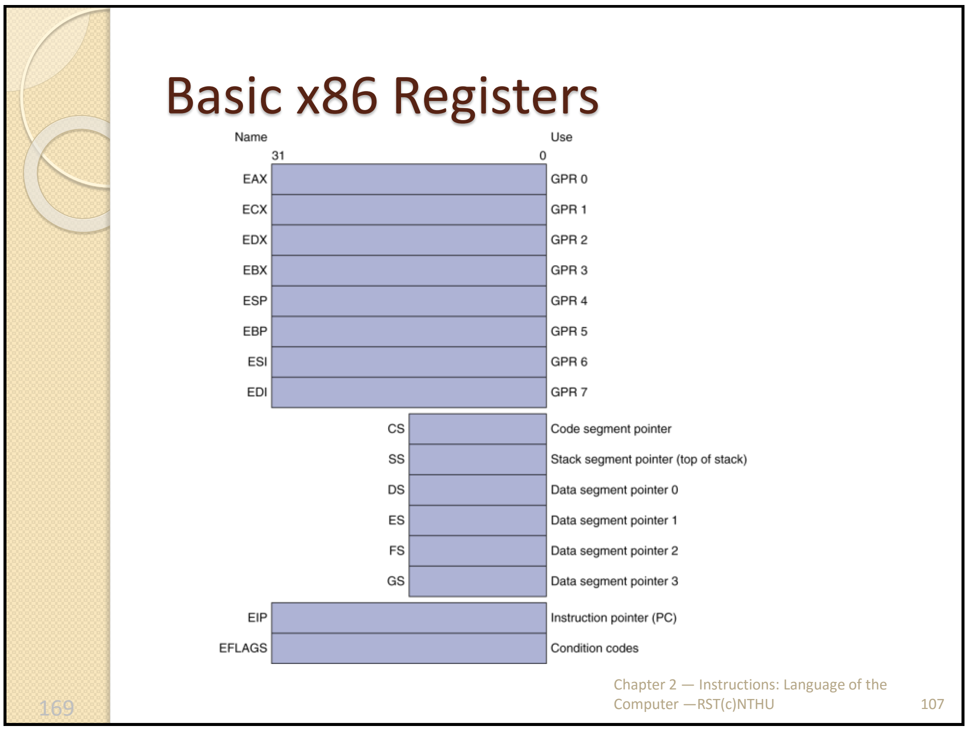
- However, the value returned by the flawed Pentium is incorrect at or beyond four digits:

$$\frac{4195835}{3145727} = 1.333739068902037589$$

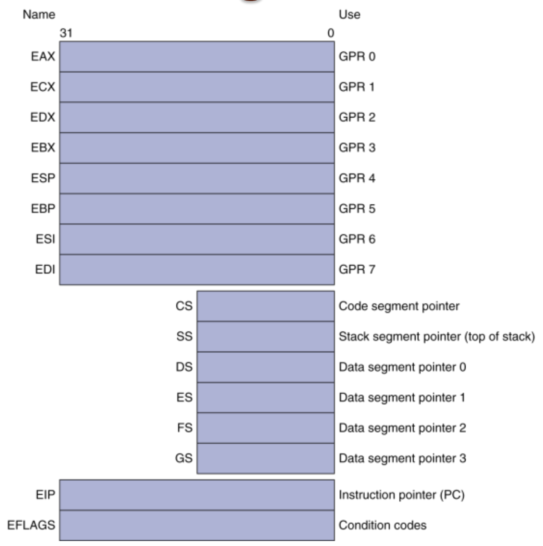


The Intel x86 ISA

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

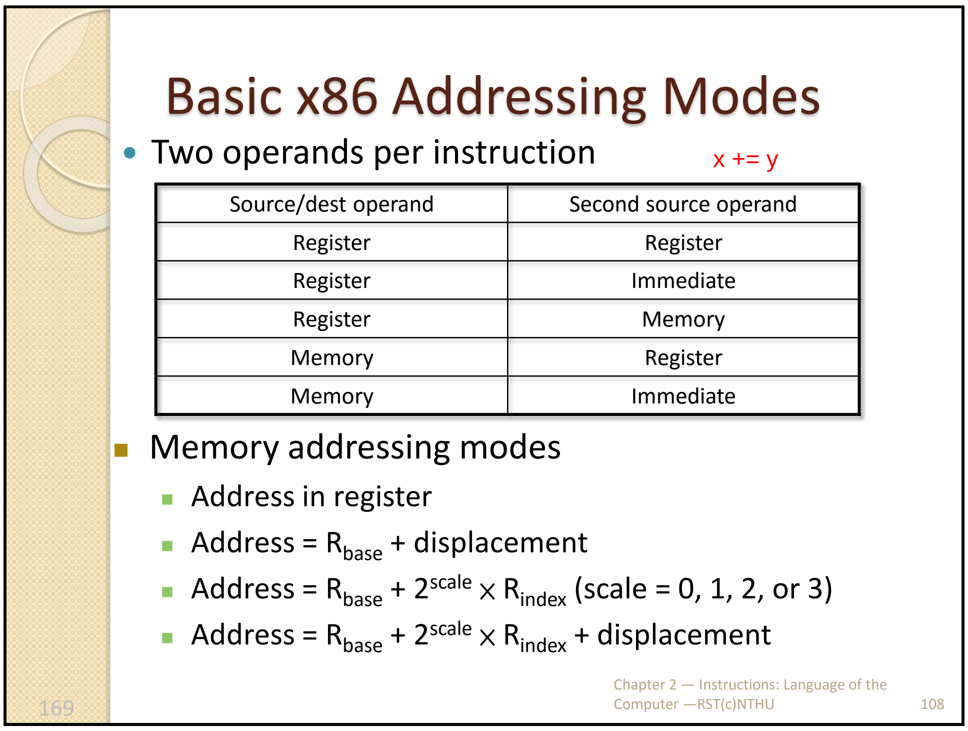


Basic x86 Registers



169

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 107



Basic x86 Addressing Modes

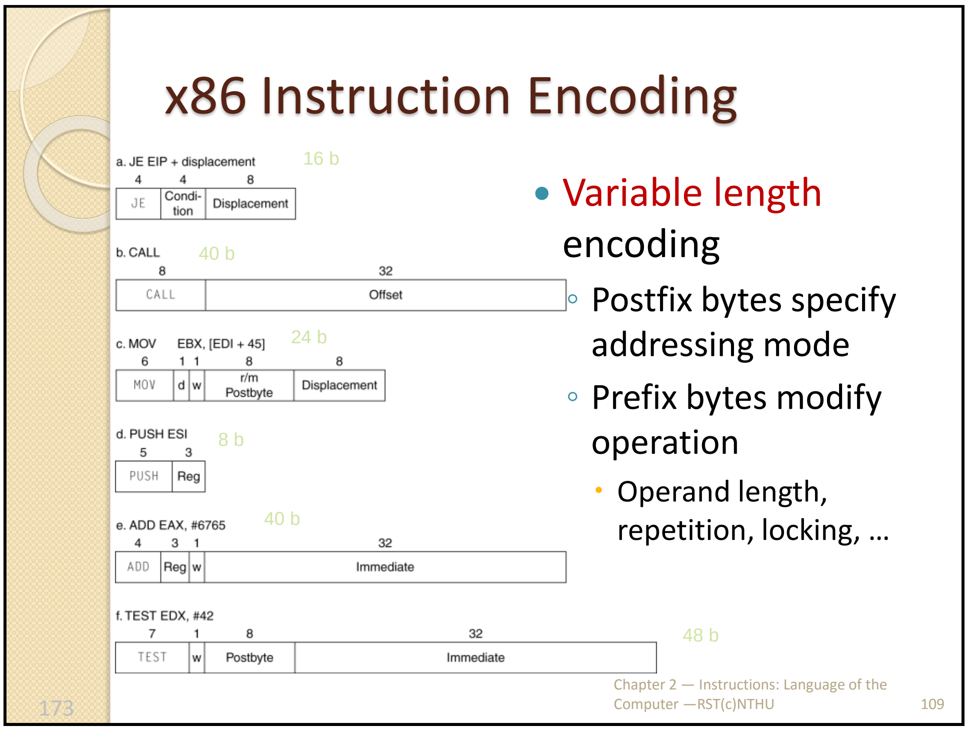
- Two operands per instruction x += y

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

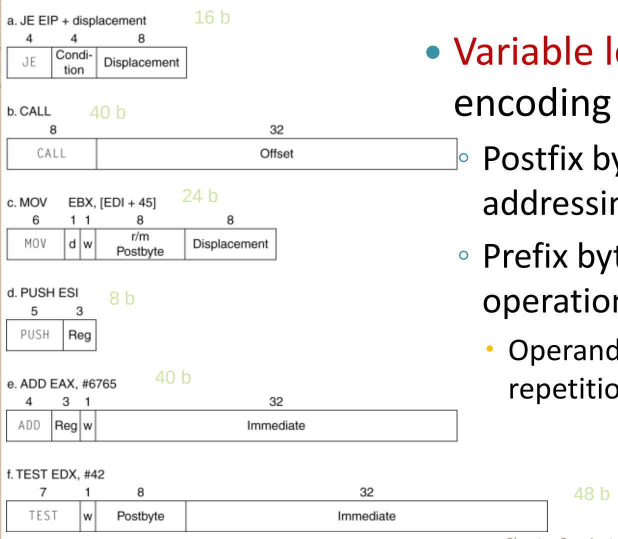
- Memory addressing modes
  - Address in register
  - Address = R_{base} + displacement
  - Address = $R_{base} + 2^{scale} \times R_{index}$ (scale = 0, 1, 2, or 3)
  - Address = $R_{base} + 2^{scale} \times R_{index}$ + displacement

169

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 108



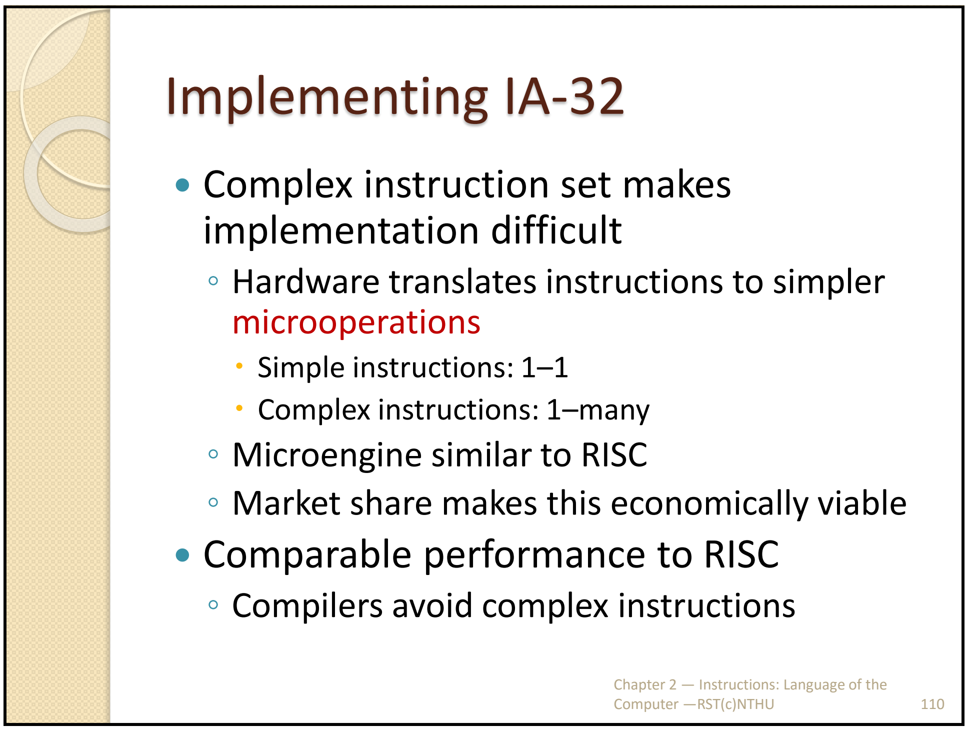
x86 Instruction Encoding



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

173

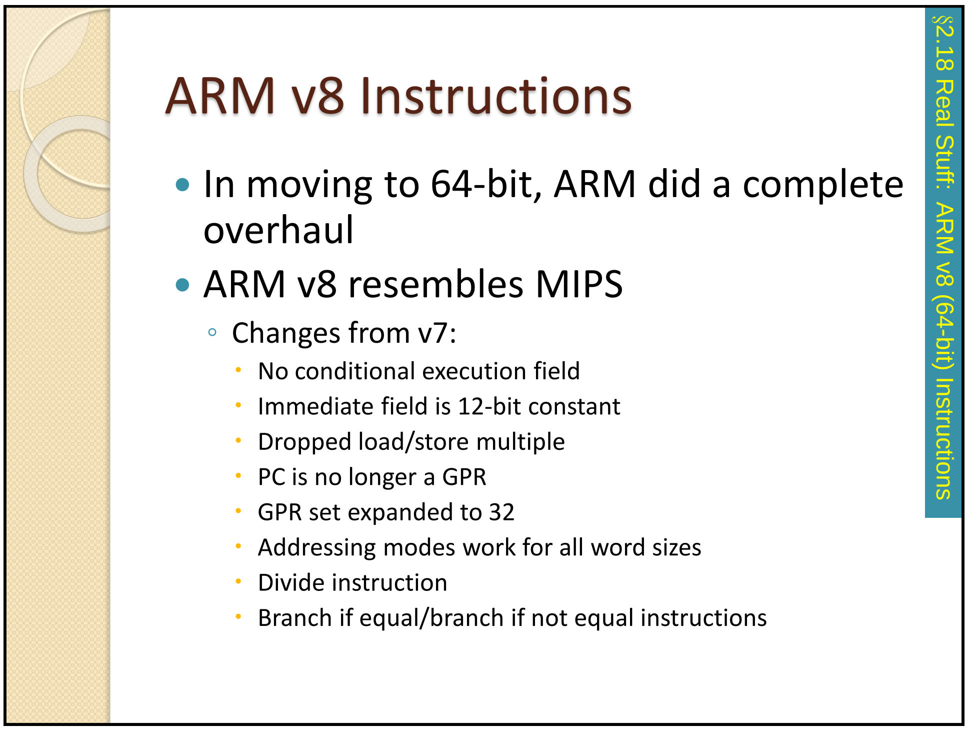
Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 109



Implementing IA-32

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

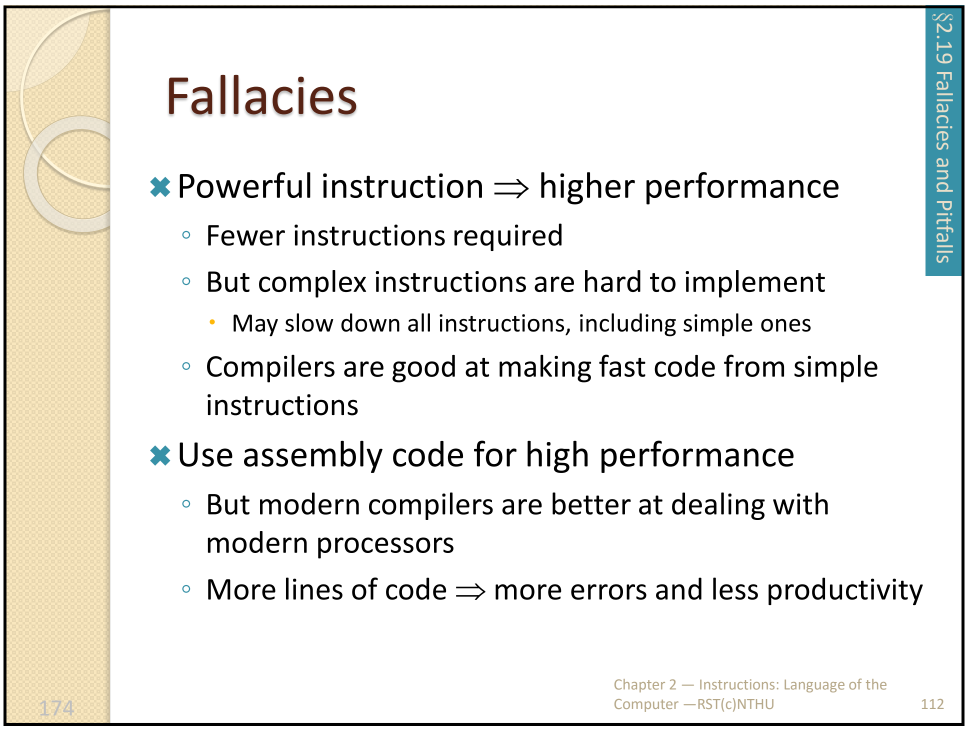
Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 110



§2.18 Real Stuff: ARM v8 (64-bit) Instructions

ARM v8 Instructions

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



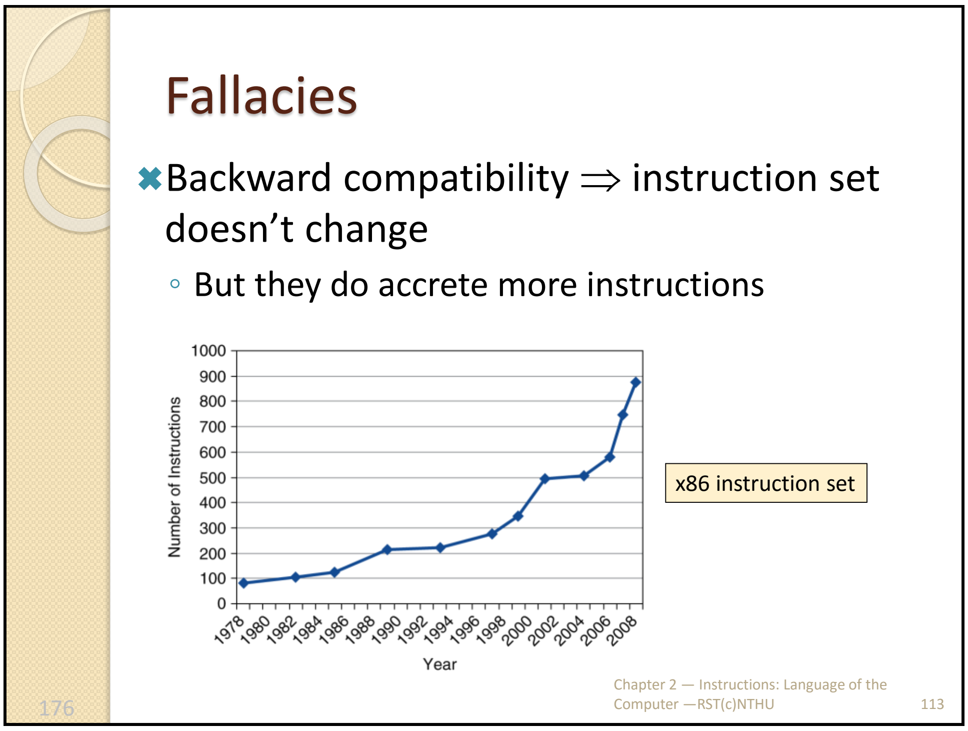
§2.19 Fallacies and Pitfalls

Fallacies

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

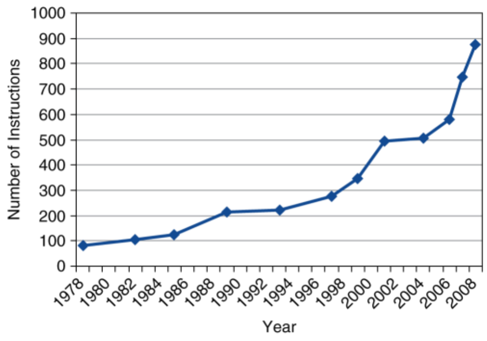
174

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 112



Fallacies

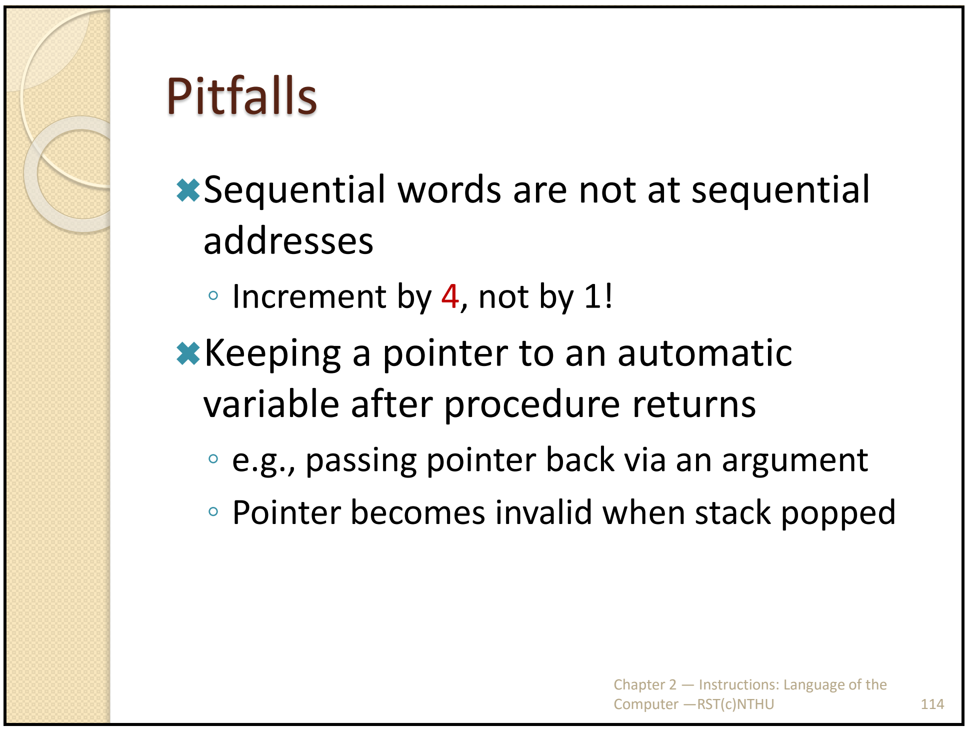
- Backward compatibility ⇒ instruction set doesn't change
  - But they do accrete more instructions



x86 instruction set

176

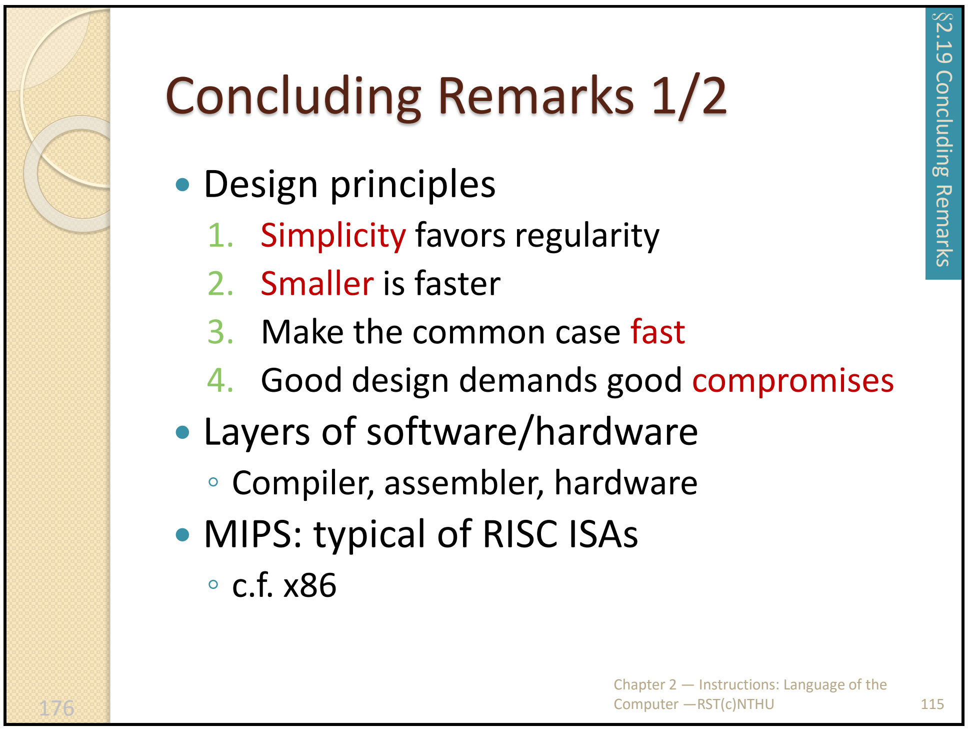
Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 113



Pitfalls

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

Chapter 2 — Instructions: Language of the Computer —RST(c)NTHU 114

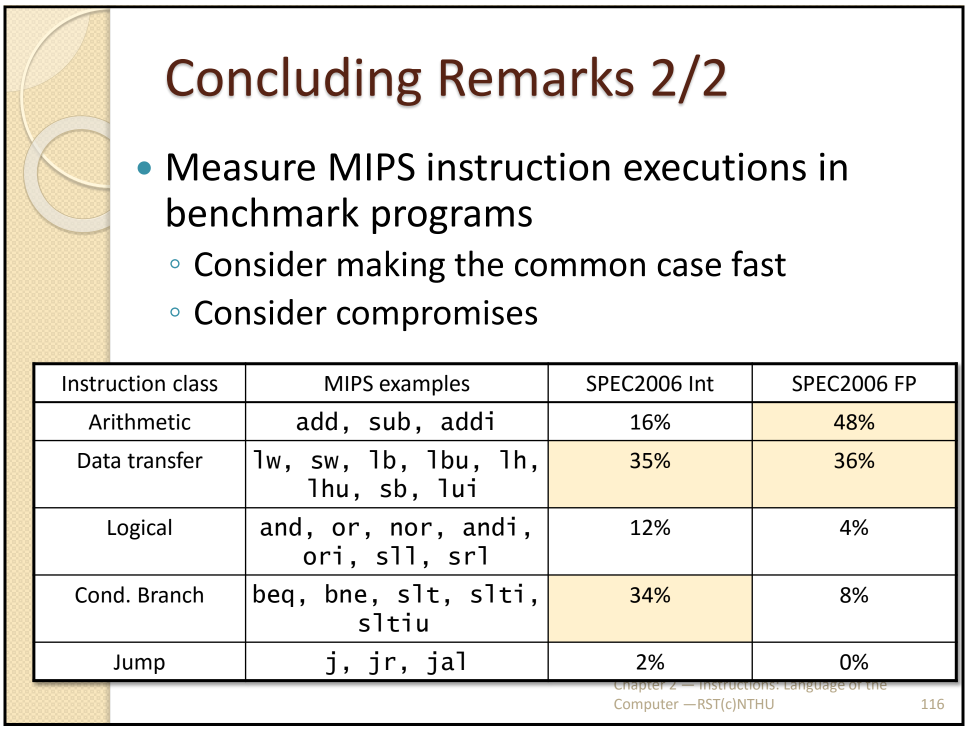


§2.19 Concluding Remarks

Concluding Remarks 1/2

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

176



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%