

Chapter 2

Instructions: Language of the Computer

heeyoul choi
hchoi@handong.edu

School of CS and EE
Handong Global University

Instruction Set

- Instructions
 - the words of a computer's language
- Instruction set
 - the vocabulary of commands understood by a given architecture
- 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

The MIPS Instruction Set

- Used as the example throughout the book
- Stanford MIPS commercialized by MIPS Technologies
- 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 A
- other popular instruction sets
 - ARMv7, Intel x86, ARMv8
 - similar because of the same goal on similar hardware

goal of this chapter

- to teach an instruction set
- to show how it is represented in hardware
- to show the relationship between high-level language and more primitive one.
 - for the high-level language, examples are in C

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
- at most one instruction at a line

Arithmetic Example

- C code:

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

- Compiled MIPS code:

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

Arithmetic Example

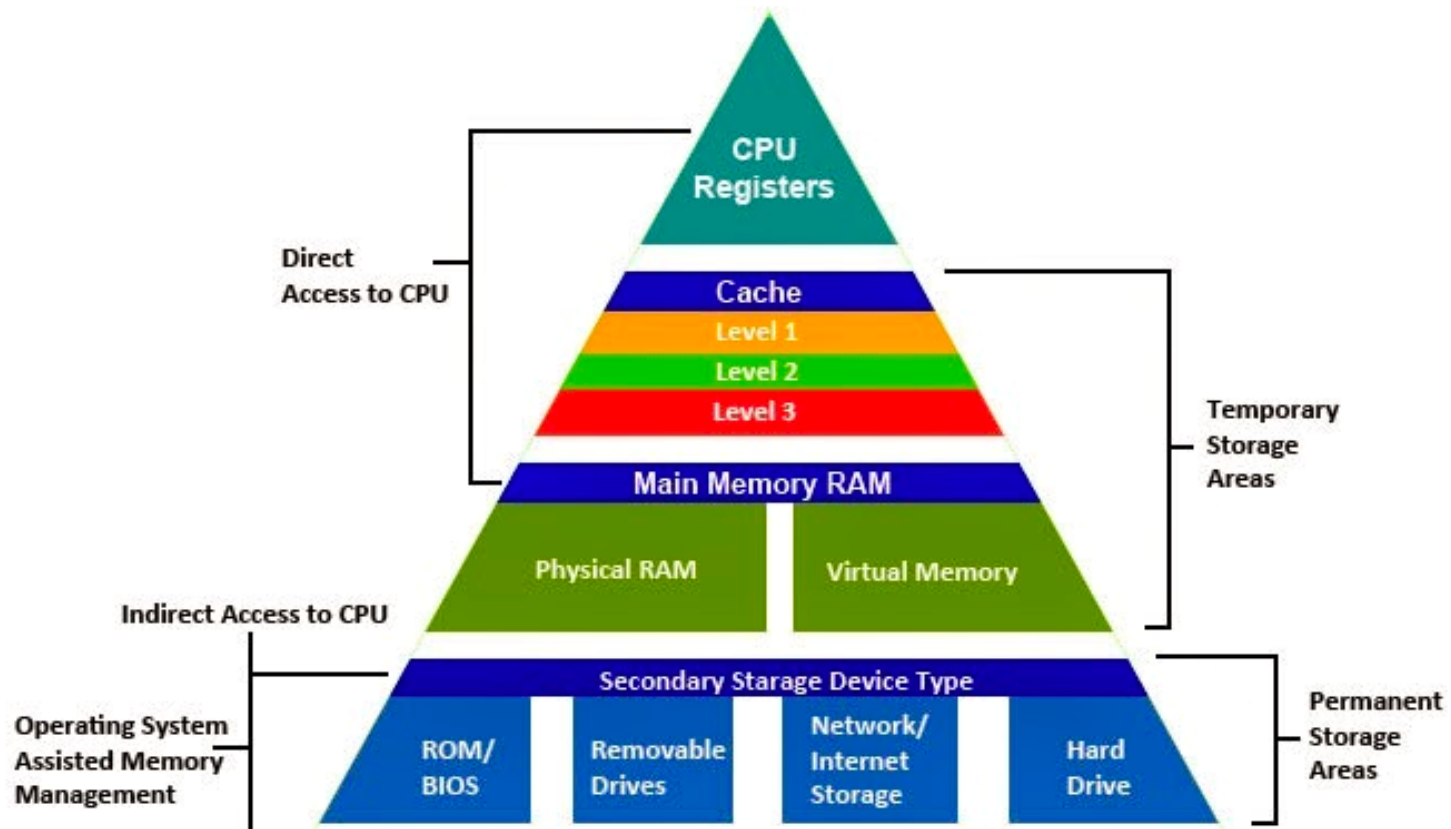
- C code:

$a = b + c + d$

- Compiled MIPS code:

Register

- Reside in CPU
- Faster and smaller than main memory



<https://tvtropes.org/pmwiki/pmwiki.php/UsefulNotes/MemoryHierarchy>

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

NAME	NUMBER	USE
\$zero	0	The Constant Value 0
\$at	1	Assembler Temporary
\$v0-\$v1	2-3	Values for Function Results and Expression Evaluation
\$a0-\$a3	4-7	Arguments
\$t0-\$t7	8-15	Temporaries
\$s0-\$s7	16-23	Saved Temporaries
\$t8-\$t9	24-25	Temporaries
\$k0-\$k1	26-27	Reserved for OS Kernel
\$gp	28	Global Pointer
\$sp	29	Stack Pointer
\$fp	30	Frame Pointer
\$ra	31	Return Address

- *Design Principle 2: Smaller is faster*

- c.f. main memory: millions of locations

only 32 registers

- the more registers may increase the clock cycle time.
- trade off between more registers and clock cycle.

Register Operand Example

- C code:

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

- `f, ..., j` in `$s0, ..., $s4`

- Compiled MIPS code:

`add $t0, $s1, $s2`

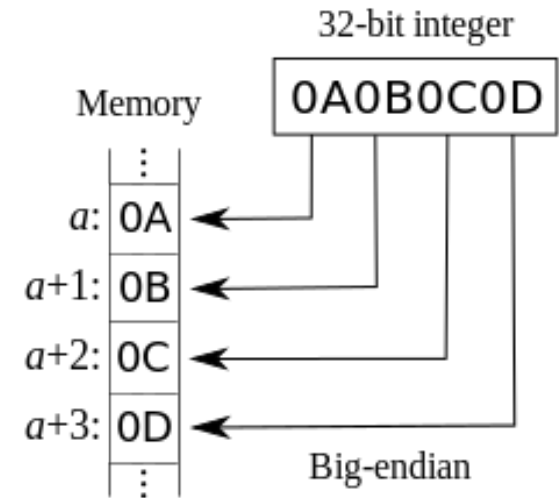
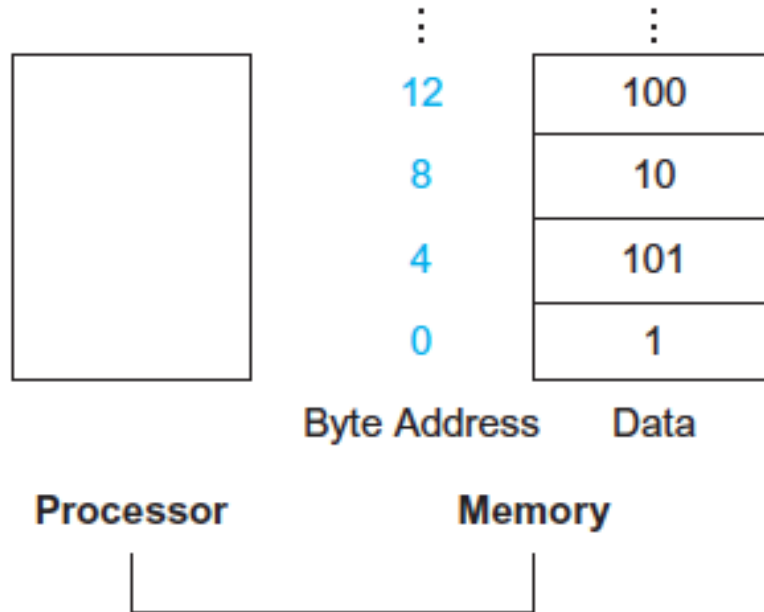
`add $t1, $s3, $s4`

`sub $s0, $t0, $t1`

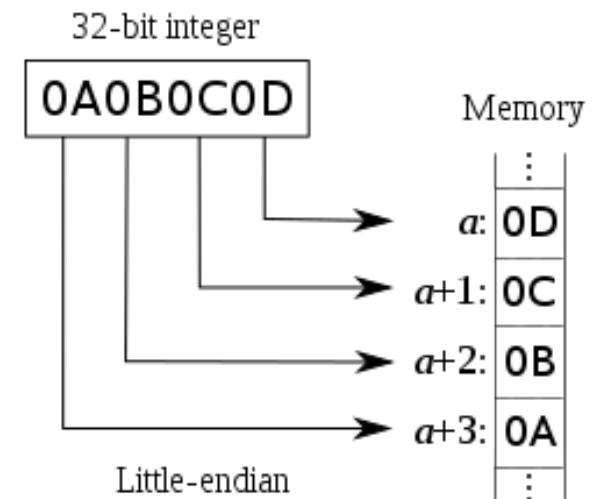
Memory Operands

- Main memory used for composite data
 - Arrays, structures, dynamic data
→ more elements than registers
- To apply arithmetic operations
 - Load values from memory into registers
 - Store the results from registers 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
 - The address of the leftmost byte as the word address.
 - *c.f.* Little Endian: least-significant byte at least address

Memory Operands



from wiki



Memory Operand Example 1

- C code: assuming that A is an array of 100 words

`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

Memory Operand Example 2

- C code:

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

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

- Compiled MIPS code:

- Index 8 requires offset of 32

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

Registers vs. Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
 - More instructions to be executed
- Compiler must use registers for variables as much as possible
 - Only spill to memory for less frequently used variables
 - Register optimization is important!

Immediate Operands

- Constant data specified in an instruction
`addi $s3, $s3, 4`
- No subtract immediate instruction
 - Just use a negative constant
`addi $s2, $s1, -1`
- *Design Principle 3: Make the common case fast*
 - Small constants are common
 - Immediate operand avoids a load instruction

The Constant Zero

- MIPS register 0 (\$zero) is the constant 0
 - Cannot be overwritten
- Useful for common operations
 - E.g., move between registers
add \$t2, \$s1, \$zero ← move \$t2, \$s1

Unsigned Binary Integers

- Given an n-bit number

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

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

- Example

$$\begin{aligned} & \blacksquare \quad 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 1011_2 \\ & \quad = 0 + \dots + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\ & \quad = 0 + \dots + 8 \quad + 0 \quad + 2 \quad + 1 \quad = 11_{10} \end{aligned}$$

- Using 32 bits

$$\blacksquare \quad 0 \text{ to } +4,294,967,295$$

2s-Complement Signed Integers

- Given an n-bit number

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

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

- Example

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

- Using 32 bits

- $-2,147,483,648$ to $+2,147,483,647$

2s-Complement Signed Integers

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

Signed Negation

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

$$\overline{x} + \overline{x} = 1111 \dots 111_2 = -1$$

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

two's complement

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

8-bit ones'-complement integers

Bits	Unsigned value	Ones' complement value
0111 1111	127	127
0111 1110	126	126
0000 0010	2	2
0000 0001	1	1
0000 0000	0	0
1111 1111	255	-0
1111 1110	254	-1
1111 1101	253	-2
1000 0001	129	-126
1000 0000	128	-127

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

signed integer

from textbook p. 79

What is the decimal value of this 64-bit two's complement number?

1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1000_{two}

1) -4_{ten}

2) -8_{ten}

3) -16_{ten}

4) $18,446,744,073,709,551,609_{\text{ten}}$

Representing Instructions

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

NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
\$zero	0	The Constant Value 0	N.A.
\$at	1	Assembler Temporary	No
\$v0-\$v1	2-3	Values for Function Results and Expression Evaluation	No
\$a0-\$a3	4-7	Arguments	No
\$t0-\$t7	8-15	Temporaries	No
\$s0-\$s7	16-23	Saved Temporaries	Yes
\$t8-\$t9	24-25	Temporaries	No
\$k0-\$k1	26-27	Reserved for OS Kernel	No
\$gp	28	Global Pointer	Yes
\$sp	29	Stack Pointer	Yes
\$fp	30	Frame Pointer	Yes
\$ra	31	Return Address	No

MIPS R-format Instructions



■ Instruction fields

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

R-format Example

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

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

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

$$00000010001100100100000000100000_2 = 02324020_{16}$$

Hexadecimal

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

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

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

MIPS I-format Instructions



- Immediate arithmetic and load/store instructions
 - rt: destination or source register number (load or store)
 - 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

MIPS I-format Instructions

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

- lw \$t0, 32(\$s3) # load word
- sw \$t0, 48(\$s3) # store word

35	19(\$s3)	8(\$t0)	32
6 bits	5 bits	5 bits	16 bits

43	19(\$s3)	8(\$t0)	48
6 bits	5 bits	5 bits	16 bits

MIPS Instruction encoding

Instruction	Format	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32 _{ten}	n.a.
sub (subtract)	R	0	reg	reg	reg	0	34 _{ten}	n.a.
add immediate	I	8 _{ten}	reg	reg	n.a.	n.a.	n.a.	constant
lw (load word)	I	35 _{ten}	reg	reg	n.a.	n.a.	n.a.	address
sw (store word)	I	43 _{ten}	reg	reg	n.a.	n.a.	n.a.	address

FIGURE 2.5 MIPS instruction encoding. In the table above, “reg” means a register number between 0 and 31, “address” means a 16-bit address, and “n.a.” (not applicable) means this field does not appear in this format. Note that add and sub instructions have the same value in the op field; the hardware uses the funct field to decide the variant of the operation: add (32) or subtract (34).

from C to machine language

C code

```
A[300] = h + A[300];
```

\$t1 has the base of the array A
\$s2 corresponds to h

compiled into assembly language

```
lw    $t0,1200($t1) # Temporary reg $t0 gets A[300]
add   $t0,$s2,$t0    # Temporary reg $t0 gets h + A[300]
sw    $t0,1200($t1) # Stores h + A[300] back into A[300]
```

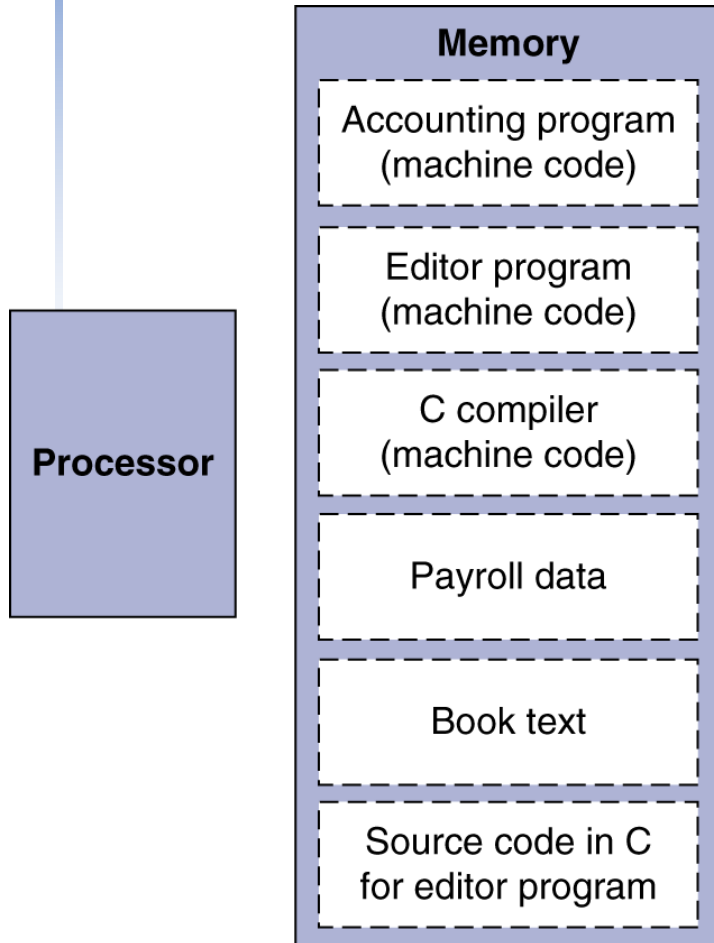
translated into machine language instructions (in decimal numbers)

Op	rs	rt	rd	address/ shamt	funct
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		
100011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000 0100 1011 0000		



Stored Program Computers

The BIG Picture



- 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

example

What MIPS instruction does this represent? Choose from one of the four options below.

from textbook p. 87

op	rs	rt	rd	shamt	funct
0	8	9	10	0	34

1. sub \$t0, \$t1, \$t2
2. add \$t2, \$t0, \$t1
3. sub \$t2, \$t1, \$t0
4. sub \$t2, \$t0, \$t1

NAME	NUMBER
\$zero	0
\$at	1
\$v0-\$v1	2-3
\$a0-\$a3	4-7
\$t0-\$t7	8-15
\$s0-\$s7	16-23
\$t8-\$t9	24-25
\$k0-\$k1	26-27
\$gp	28
\$sp	29
\$fp	30
\$ra	31

Instruction	Format	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32 _{ten}	n.a.
sub (subtract)	R	0	reg	reg	reg	0	34 _{ten}	n.a.

Logical Operations

- Instructions for bitwise manipulation

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

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

Shift Operations

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
0					0/2 sll/srl

- 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)

Shift Operations

```
sll $t2,$s0,4 # reg $t2 = reg $s0 << 4 bits
```

op	rs	rt	rd	shamt	funct
0	0	16	10	4	0
unused		\$s0	\$t2		

0000 0000 0000 0000 0000 0000 0000 1001_{two} = 9_{ten}



shift left by 4

0000 0000 0000 0000 0000 0000 1001 0000_{two} = 144_{ten}
 = 9×2^4

AND Operations

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

and \$t0, \$t1, \$t2

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

OR Operations

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

or \$t0, \$t1, \$t2

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

NOT Operations

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

`nor $t0, $t1, $zero`

Register 0: always read as zero

\$t1 0000 0000 0000 0000 0011 1100 0000 0000

\$t0 1111 1111 1111 1111 1100 0011 1111 1111

example: isolate a field

from textbook p. 89

Which operations can isolate a field in a word?

1. AND
2. A shift left followed by a shift right

Conditional Operations

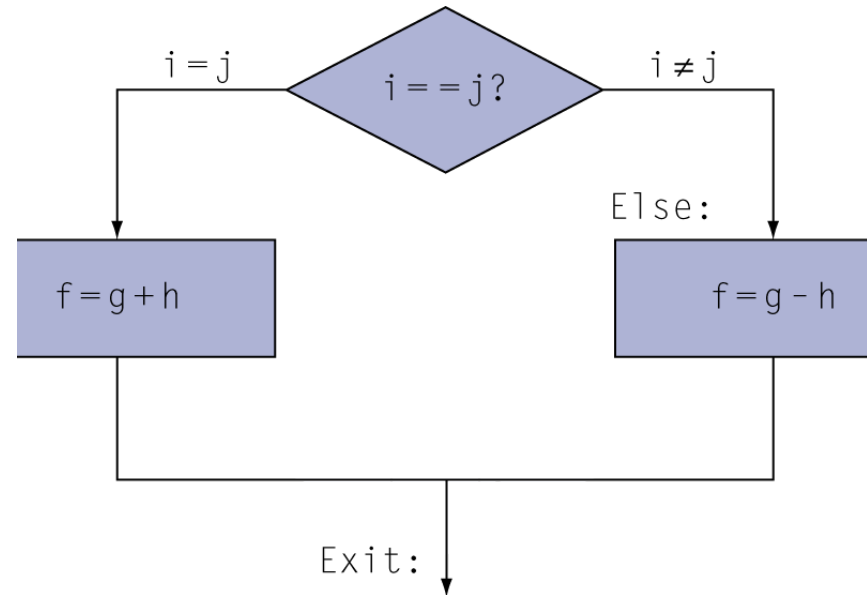
- Branch to a labeled instruction if a condition is true
 - Otherwise, continue sequentially
- `beq rs, rt, L1` *branch if equal*
 - if (`rs == rt`) branch to instruction labeled L1;
- `bne rs, rt, L1` *branch if not equal*
 - if (`rs != rt`) branch to instruction labeled L1;
- `j L1`
 - unconditional jump to instruction labeled L1

Compiling If Statements

- C code:

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

- f, g, ... in \$s0, \$s1, ...



- Compiled MIPS code:

```
        bne $s3, $s4, Else    conditional branch  
        add $s0, $s1, $s2  
        j    Exit             unconditional branch  
Else:    sub $s0, $s1, $s2  
Exit:    ...
```

Assembler calculates addresses

Compiling Loop Statements

- C code:

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

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

- Compiled MIPS code:

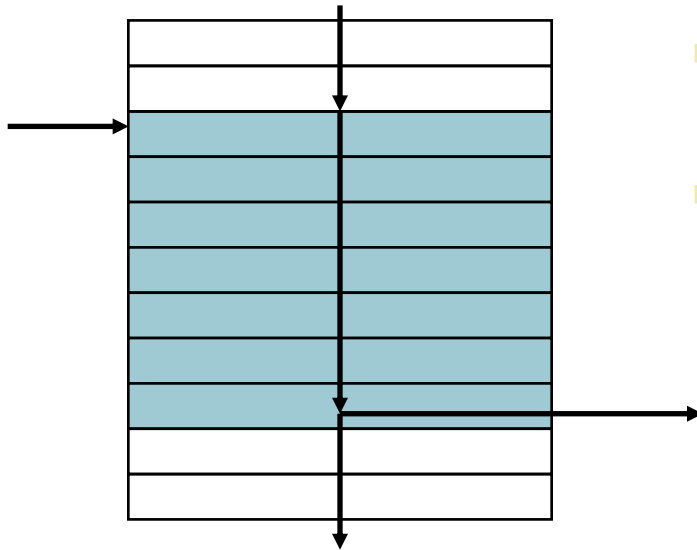
```
Loop:  sll    $t1, $s3, 2
       add    $t1, $t1, $s6
       lw     $t0, 0($t1)
       bne    $t0, $s5, Exit
       addi   $s3, $s3, 1
       j      Loop
Exit:  ...
```

$\$t1 = i * 4$

byte addressing problem

Basic Blocks

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



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

More Conditional Operations

- Set result to 1 if a condition is true
 - Otherwise, set to 0
- `slt rd, rs, rt` set on less than
 - if ($rs < rt$) $rd = 1$; else $rd = 0$;
- `slti rt, rs, constant`
 - if ($rs < \text{constant}$) $rt = 1$; else $rt = 0$;
- Use in combination with `beq`, `bne`
`slt $t0, $s1, $s2 # if ($s1 < $s2)`
`bne $t0, $zero, L # branch to L`

Compiling Loop Statements

- C code:

```
for (i=0; i< k; i++) save[i] = 2+i;
```

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

- Compiled MIPS code:

```
        move $s3, $zero           # i = 0
```

```
Loop:
```

```
Exit: ...
```

Compiling Loop Statements

- C code:

```
for (i=0; i < k; i++) save[i] = 2+i;
```

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

- Compiled MIPS code:

```
      move    $s3, $zero           # i = 0
Loop: beq     $s3, $s5, Exit
      sll     $t1, $s3, 2
      add     $t1, $t1, $s6        #t1 = &save[i]
      addi    $t2, $s3, 2          #t2 = 2+i
      sw      $t2, 0($t1)
      addi    $s3, $s3, 1
      j       Loop
Exit: ...
```

Branch Instruction Design

- Why not `blt`, `bge`, etc? (branch on less than)
- Hardware for `<`, `≥`, ... is slower than `=`, `≠`
 - Combining with branch involves more work per instruction, requiring a slower clock
 - All instructions penalized!
- `beq` and `bne` are the common cases
 - for `blt`, (`slt` and `beq`) would be enough
- This is a good design compromise

Signed vs. Unsigned

- Signed comparison: `slt`, `slti`
- Unsigned comparison: `sltu`, `sltui`
- Example
 - `$s0 = 1111 1111 1111 1111 1111 1111 1111 1111`
 - `$s1 = 0000 0000 0000 0000 0000 0000 0000 0001`
 - `slt $t0, $s0, $s1 # signed`
 - $-1 < +1 \Rightarrow \$t0 = 1$
 - `sltu $t0, $s0, $s1 # unsigned`
 - $+4,294,967,295 > +1 \Rightarrow \$t0 = 0$
- treating signed numbers as if they were unsigned.
 - “ $x < y$ ” \rightarrow “ $0 \leq x \text{ AND } x < y$ ”
 - `sltu $t0, $s1, $t2` # `$t0 = 0`, if `$s1 >= $t2` or `$s1 < 0`
 - `beq $t0, $zero, IndexOutOfBounds`

Procedure Calling

- 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

Register Usage

- \$a0 – \$a3: arguments to pass params (reg's 4 – 7)
- \$v0, \$v1: result values to return (reg's 2 and 3)
- \$t0 – \$t9: temporaries
 - Can be overwritten by callee
- \$s0 – \$s7: saved
 - Must be saved/restored by callee
- \$gp: global pointer for static data (reg 28)
- \$sp: stack pointer (reg 29)
- \$fp: frame pointer (reg 30)
- \$ra: return address to return (reg 31)

Procedure Call Instructions

- Procedure call: jump and link

`jal ProcedureLabel`

- Puts the address of following instruction in `$ra`
- Jumps to the 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

Procedure Call Instructions

■ Jump

- J 1000 : PC \leftarrow 1000

J:

op=2	target=250 (word address)
------	---------------------------

■ Jump Register

- JR \$ra : PC \leftarrow \$ra

R:

op=0	rs=31	rt=0	rd=0	shamt=0	func=8
------	-------	------	------	---------	--------

■ Jump and Link

- JAL 1000 : \$ra \leftarrow PC; PC \leftarrow 1000

J:

op=3	target=250 (word address)
------	---------------------------

Leaf Procedure Example

- C code:

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

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

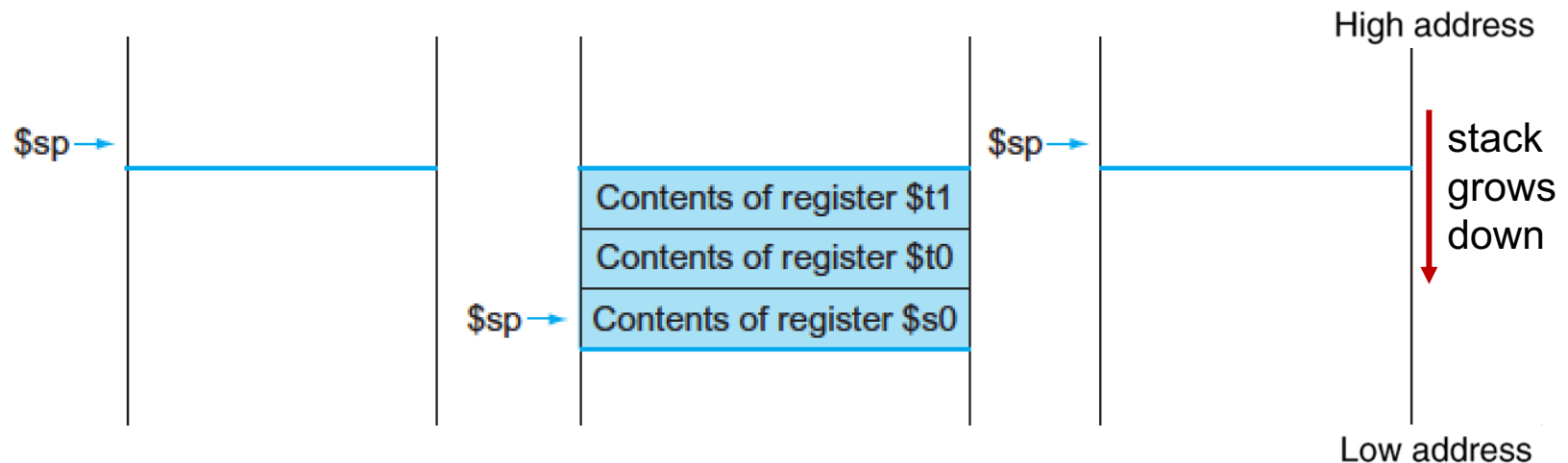
Leaf Procedure Example

- MIPS code:

leaf_example:		
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

Leaf Procedure Example

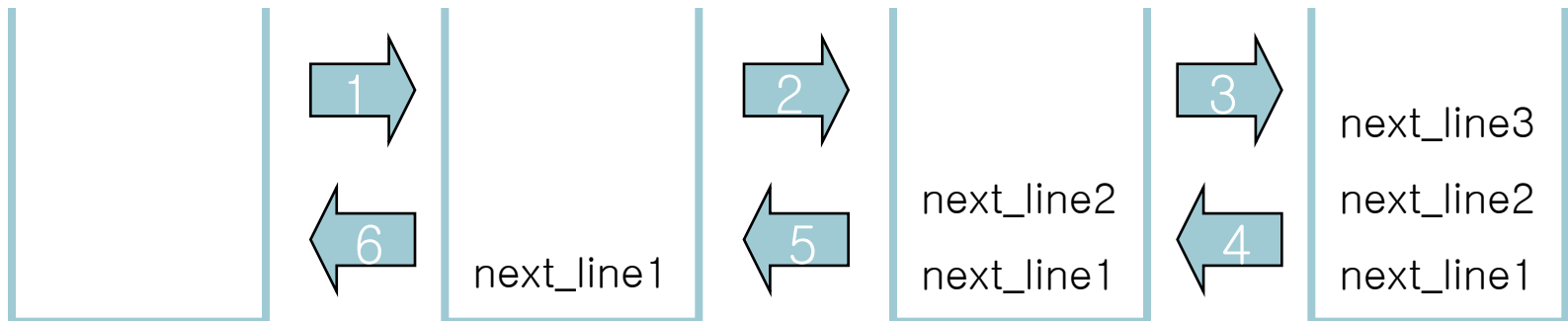
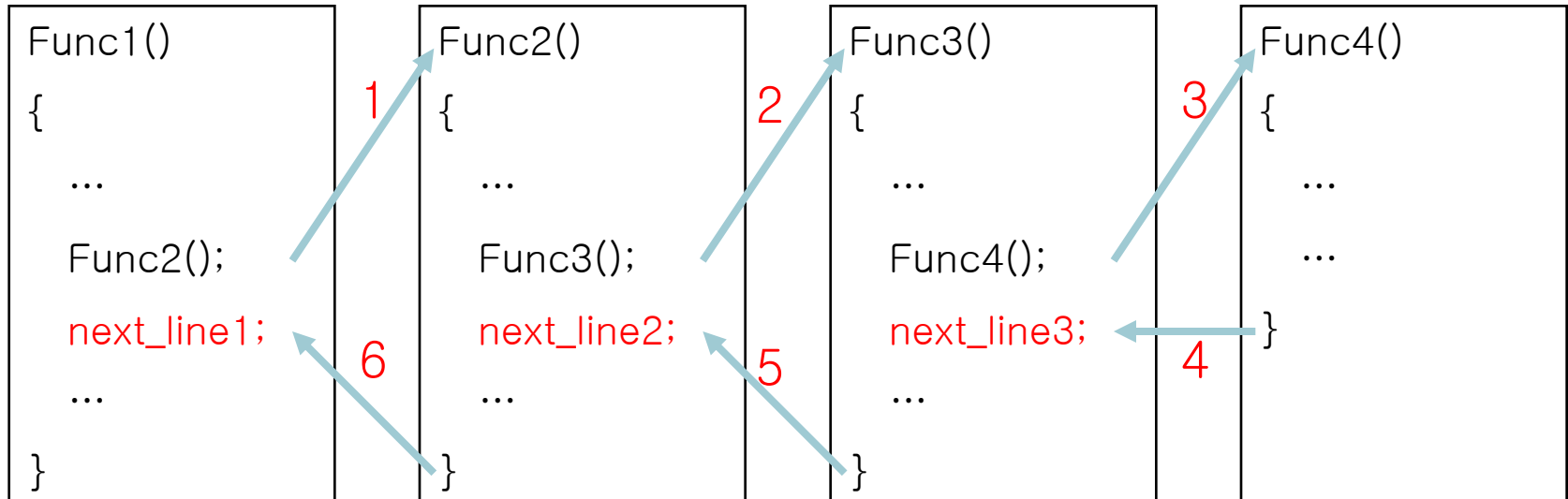
```
addi $sp, $sp, -12 # adjust stack to make room for 3 items
sw  $t1, 8($sp)    # save register $t1 for use afterwards
sw  $t0, 4($sp)    # save register $t0 for use afterwards
sw  $s0, 0($sp)    # save register $s0 for use afterwards
```



```
lw  $s0, 0($sp)    # restore register $s0 for caller
lw  $t0, 4($sp)    # restore register $t0 for caller
lw  $t1, 8($sp)    # restore register $t1 for caller
addi $sp,$sp,12    # adjust stack to delete 3 items
```


Stack in Computer System

- At function call, return address is saved in stack



Non-Leaf Procedures

- Procedures that call other procedures
- For nested call, use stack to save:
 - Its return address
 - Any arguments and temporaries needed after the call
- Restore from the stack after the call
- Caller pushes
 - argument registers (\$a0-\$a3)
 - temporary registers (\$t0-\$t9) that are needed after the call
- Callee pushes
 - the return address register (\$ra)
 - any saved registers (\$s0-\$s7) used by callee

Non-Leaf Procedure Example

- C code:

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

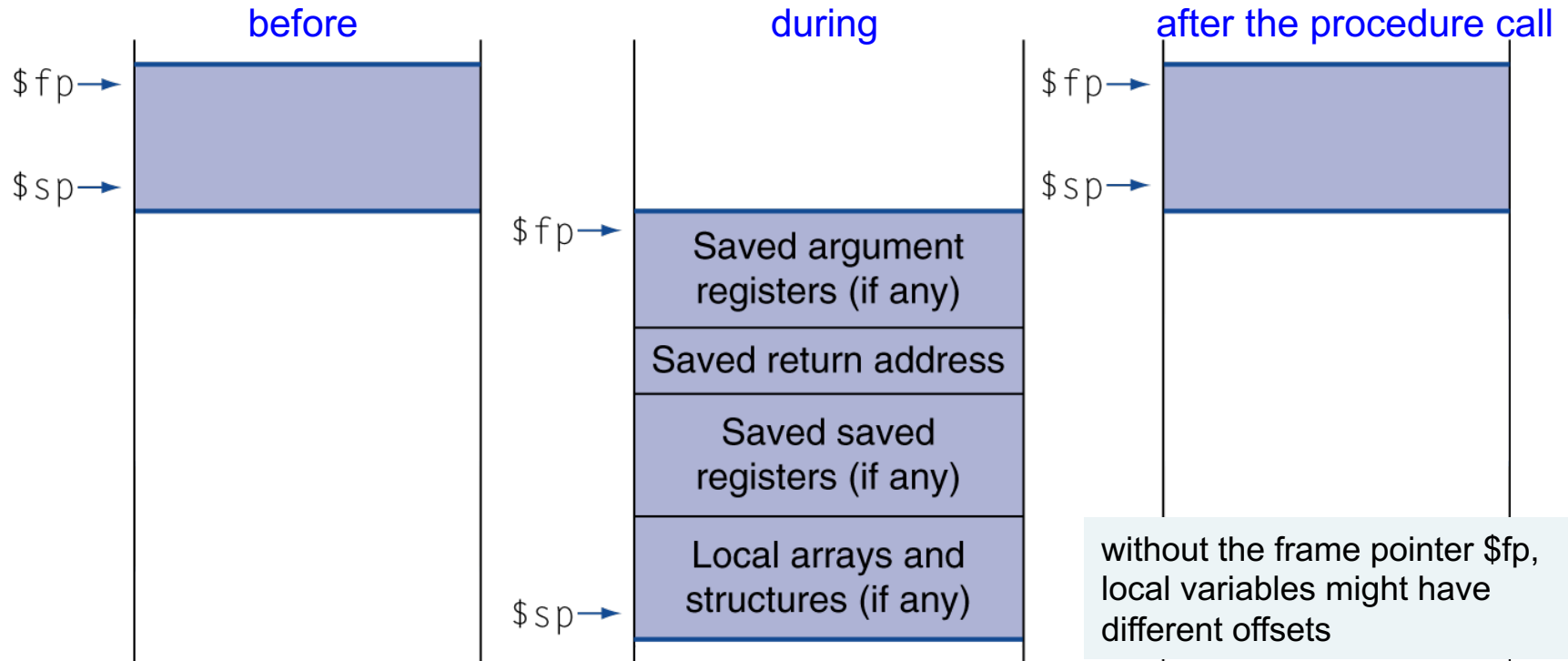
- Argument n in \$a0
- Result in \$v0

Non-Leaf 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

Local Data on the Stack



- Local data allocated by callee (e.g., C automatic variables)
- Procedure frame (activation record)
 - Used by some compilers to manage stack storage
- Frame Pointer
 - the location of the saved regs and local vars for a given procedure

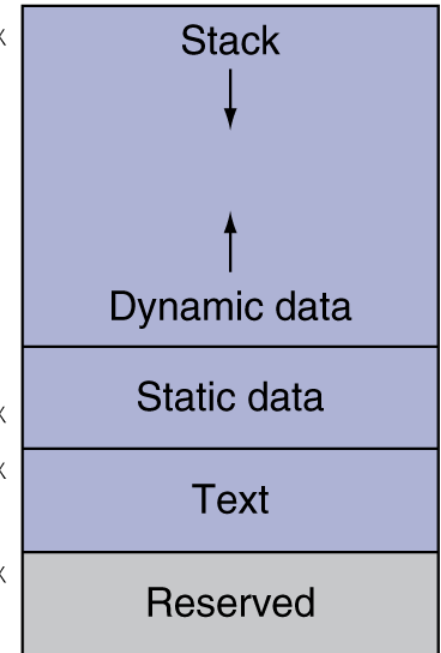
Memory Layout

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

\$sp → 7fff fffc_{hex}

\$gp → 1000 8000_{hex}
1000 0000_{hex}

pc → 0040 0000_{hex}
0



Character Data

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

Byte/Halfword Operations

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

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

- Sign extend to 32 bits in rt

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

- Zero extend to 32 bits in rt

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

- Store just rightmost byte/halfword

String Copy Example

- C code (naïve):

- Null-terminated string

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

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

String Copy Example

■ MIPS code:

strcpy:		
addi	\$sp, \$sp, -4	# adjust stack for 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 stack
jr	\$ra	# and return

32-bit Constants

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

`lui rt, constant`

load upper immediate

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

`lui $s0, 61`

0000 0000 0011 1101	0000 0000 0000 0000
---------------------	---------------------

`ori $s0, $s0, 2304`

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

decimal

Branch Addressing

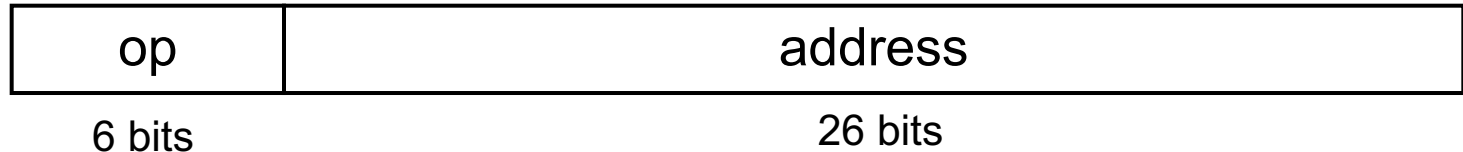
- The conditional branch instructions specify
 - Opcode, two registers, target address
- Most branch targets are near branch
 - Forward or backward $-2^{15} \sim +2^{15}$ words



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

Jump Addressing

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



- (Pseudo)Direct jump addressing
 - Target address = $PC_{31...28} : (\text{address} \times 4)$ concat
leaving the upper 4 bits unchanged.

Target Addressing Example

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

```
Loop: sll    $t1, $s3, 2      80000
      add    $t1, $t1, $s6    80004
      lw     $t0, 0($t1)      80008
      bne    $t0, $s5, Exit   80012
      addi   $s3, $s3, 1      80016
      j      Loop            80020
Exit: ...                    80024
```

0	0	19	9	2	0
0	9	22	9	0	32
35	9	8	0		
5	8	21	2		
8	19	19	1		
2	20000				

Branching Far Away

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

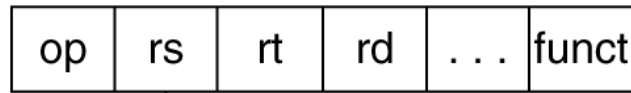
```
        beq $s0,$s1, L1
           ↓
        bne $s0,$s1, L2
        j  L1
L2:      ...
```

Addressing Mode Summary

1. Immediate addressing



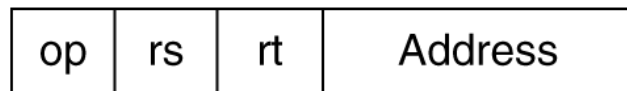
2. Register addressing



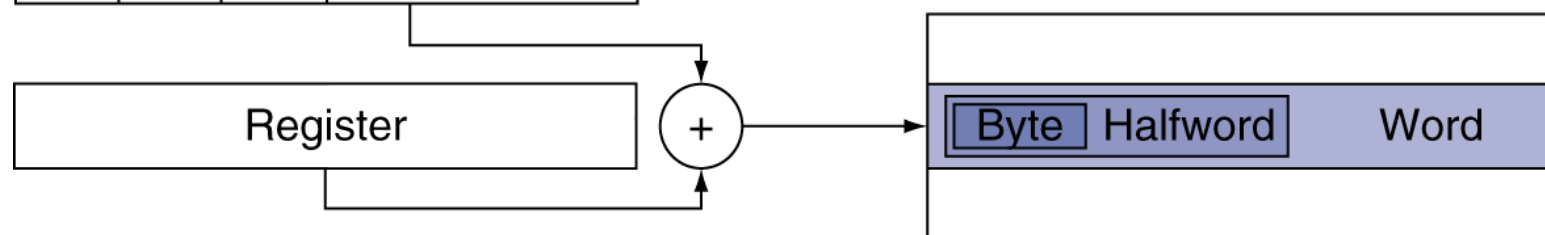
Registers

Register

3. Base addressing

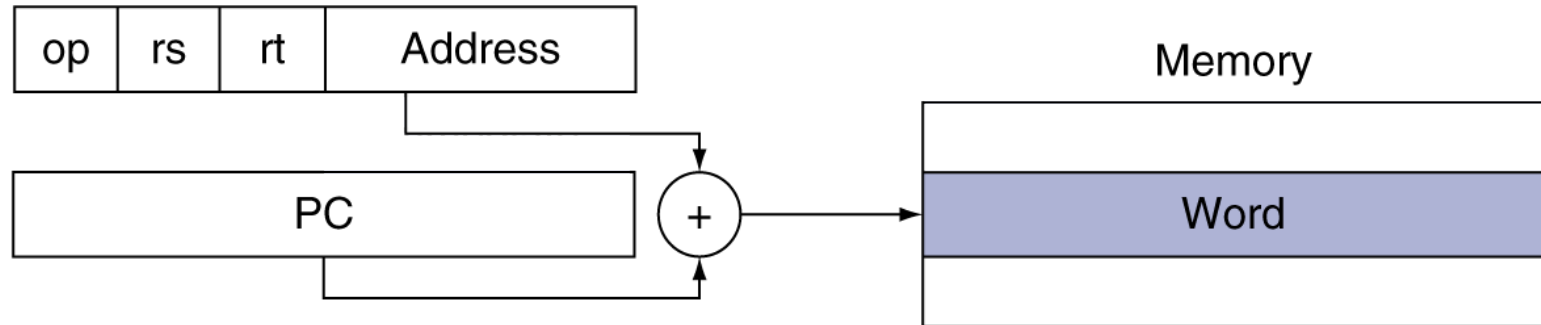


Memory

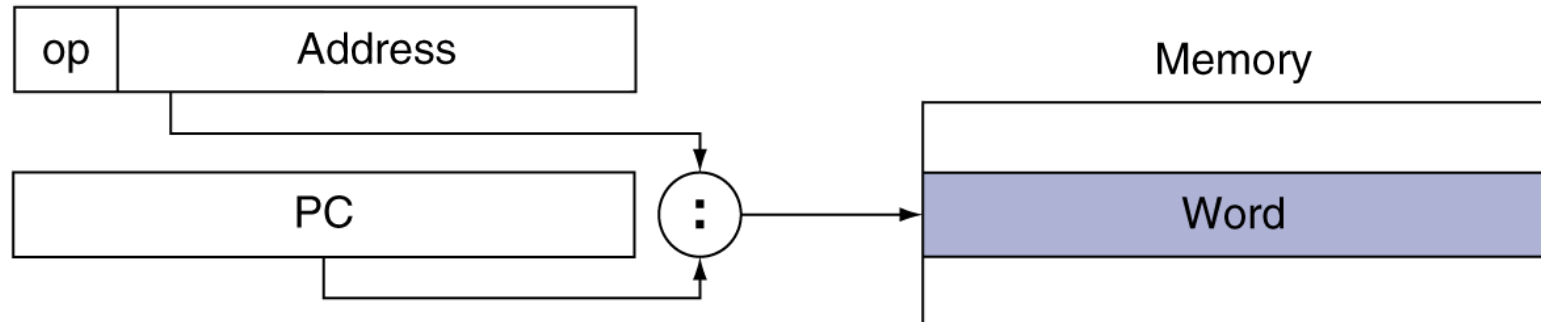


Addressing Mode Summary

4. PC-relative addressing



5. Pseudodirect addressing



MIPS instruction formats

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				

Name	Comments
Field size	All MIPS instructions are 32 bits long
R-format	Arithmetic instruction format
I-format	Transfer, branch, imm. format
J-format	Jump instruction format

Decoding machine code

assembly language corresponding to this machine instruction?

00af8020hex

→ Binary representation, reformat/decoding and translation.

1. binary

0000 0000 1010 1111 1000 0000 0010 0000

2. reformat/decoding

op	rs	rt	rd	shamt	funct
000000	00101	01111	10000	00000	100000

3. translation

add \$s0,\$a1,\$t7

Synchronization

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

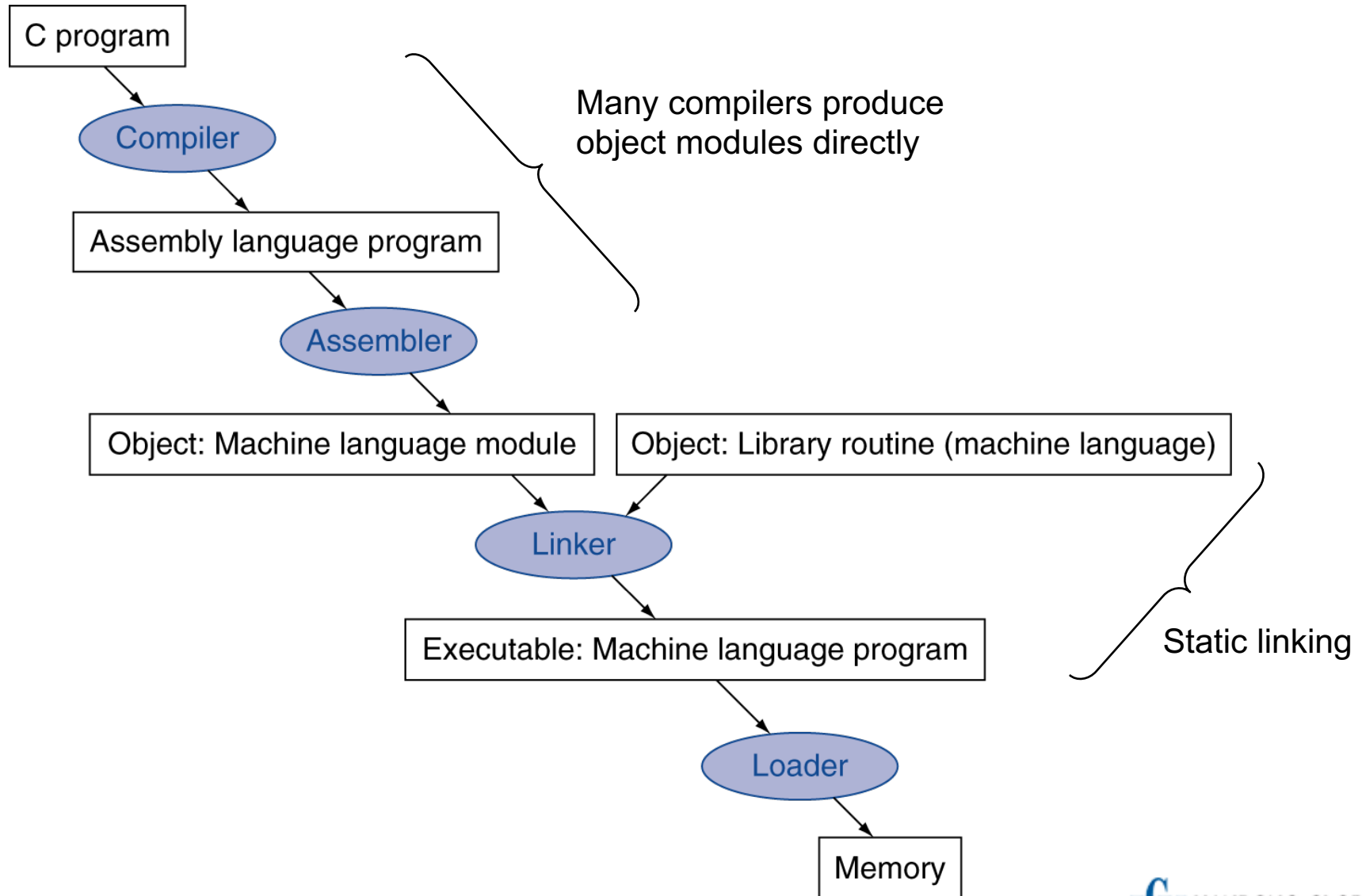
Synchronization in MIPS

- Load linked: `ll rt, offset(rs)`
- Store conditional: `sc rt, offset(rs)`
 - Succeeds if location not changed since the `ll`
 - Returns 1 in `rt`
 - Fails if location is changed
 - Returns 0 in `rt`
- Example: atomic swap (to test/set lock variable)

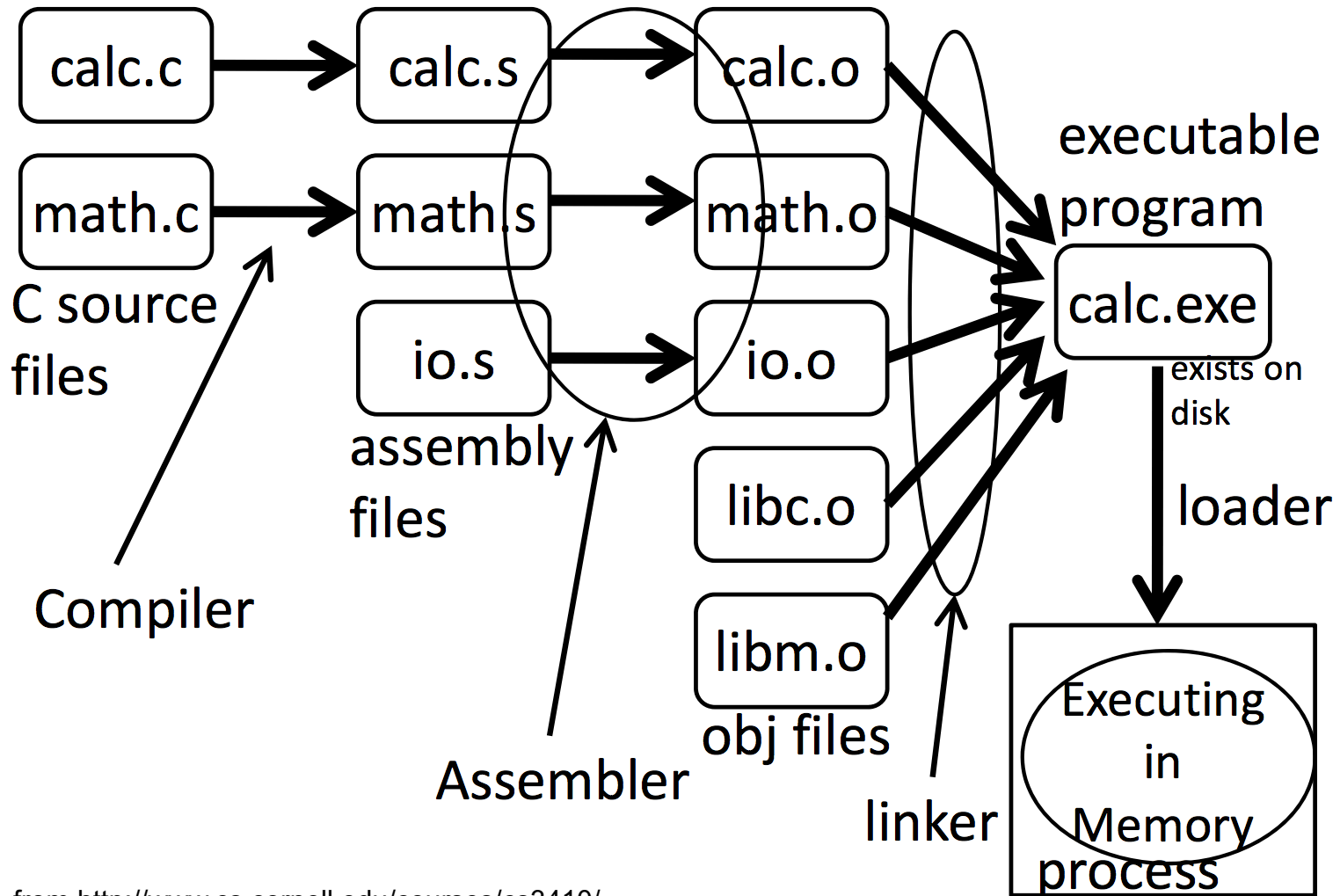
```
try: add $t0,$zero,$s4 ;copy exchange value
      ll $t1,0($s1)    ;load linked
      sc $t0,0($s1)    ;store conditional
      beq $t0,$zero,try ;branch store fails
      add $s4,$zero,$t1 ;put load value in $s4
```

the content of `$s4` and the memory at `$s1`: atomically exchanged

Translation and Startup



translation example



from <http://www.cs.cornell.edu/courses/cs3410/>

program layout example

calc.c

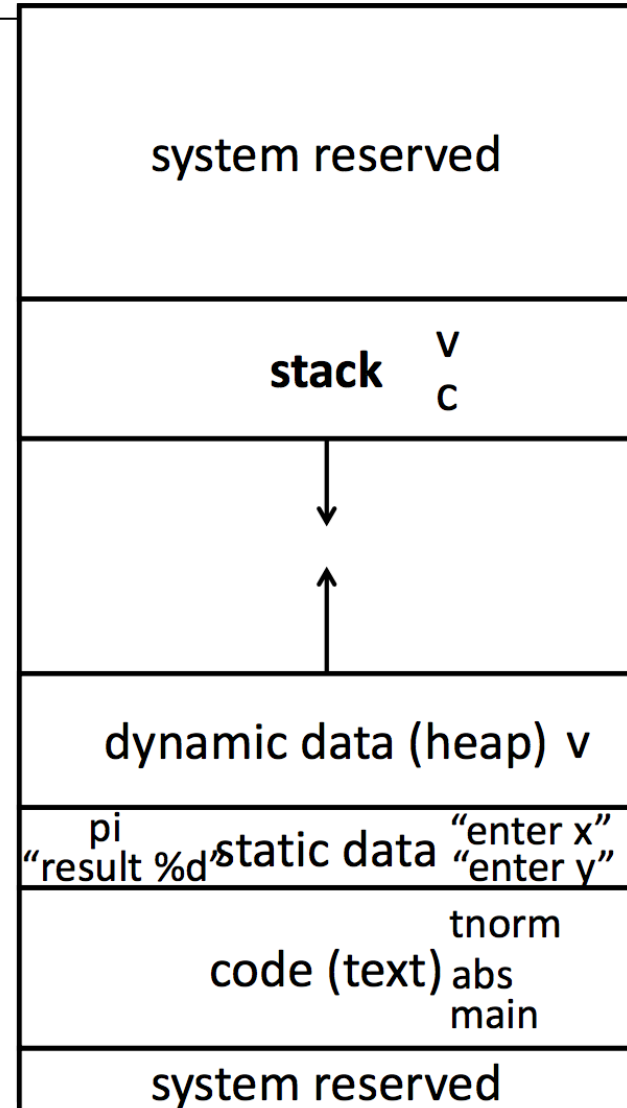
```
vector* (v) = malloc(8);
v->x = prompt("enter x");
v->y = prompt("enter y");
int (c) = pi + tnorm(v);
print("result %d", c);
```

math.c

```
int tnorm(vector* v) {
    return abs(v->x)+abs(v->y);
}
```

lib3410.o

global variable: (pi)
 entry point: prompt
 entry point: print
 entry point: malloc



Assembler Pseudoinstructions

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

`move $t0, $t1` \rightarrow `add $t0, $zero, $t1`

`b1t $t0, $t1, L` \rightarrow `slt $at, $t0, $t1`
 `bne $at, $zero, L`

- `$at` (register 1): assembler temporary

`bgt, bge, and ble`

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
 - global vars, strings, constants
 - Relocation info: for contents that depend on absolute location of loaded program (lw/sw, jal)
 - Symbol table: global definitions and external refs (in other files)
 - Debug info: for associating with source code

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

example

after compile

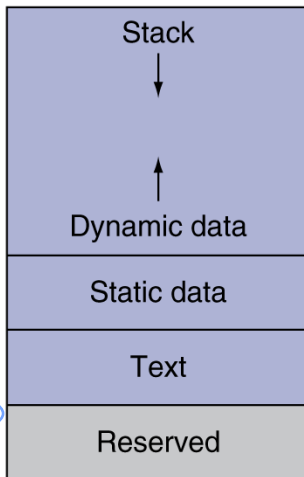
A starts at 40 0000 and the size is 100
Then B starts at 40 0100

\$sp → 7fff fffc_{hex}

\$gp → 1000 8000_{hex}
1000 0000_{hex}

pc → 0040 0000_{hex}

0



Object file header			
	Name	Procedure A	
	Text size	100 _{hex}	
	Data size	20 _{hex}	
Text segment	Address	Instruction	
	0	lw \$a0, 0(\$gp)	
	4	jal 0	
	
Data segment	0	(X)	
	
Relocation information	Address	Instruction type	Dependency
	0	lw	X
	4	jal	B
Symbol table	Label	Address	
	X	–	
	B	–	
Object file header			
	Name	Procedure B	
	Text size	200 _{hex}	
	Data size	30 _{hex}	
Text segment	Address	Instruction	
	0	sw \$a1, 0(\$gp)	
	4	jal 0	
	
Data segment	0	(Y)	
	
Relocation information	Address	Instruction type	Dependency
	0	sw	Y
	4	jal	A
Symbol table	Label	Address	
	Y	–	
	A	–	

example

after link

$\$gp = 1000\ 8000_{\text{hex}}$

$1000\ 8000 + (8000) = 1000\ 0000$

signed 16bit

Executable file header		
	Text size	300 _{hex}
	Data size	50 _{hex}
Text segment	Address	Instruction
	0040 0000 _{hex}	lw \$a0, 8000 _{hex} (\$gp)
	0040 0004 _{hex}	jal 40 0100 _{hex}

	0040 0100 _{hex}	sw \$a1, 8020 _{hex} (\$gp)
	0040 0104 _{hex}	jal 40 0000 _{hex}

Data segment	Address	
	1000 0000 _{hex}	(X)

	1000 0020 _{hex}	(Y)

load from X

jal B

store to Y

jal A

Loading a Program

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

Dynamic Linking

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

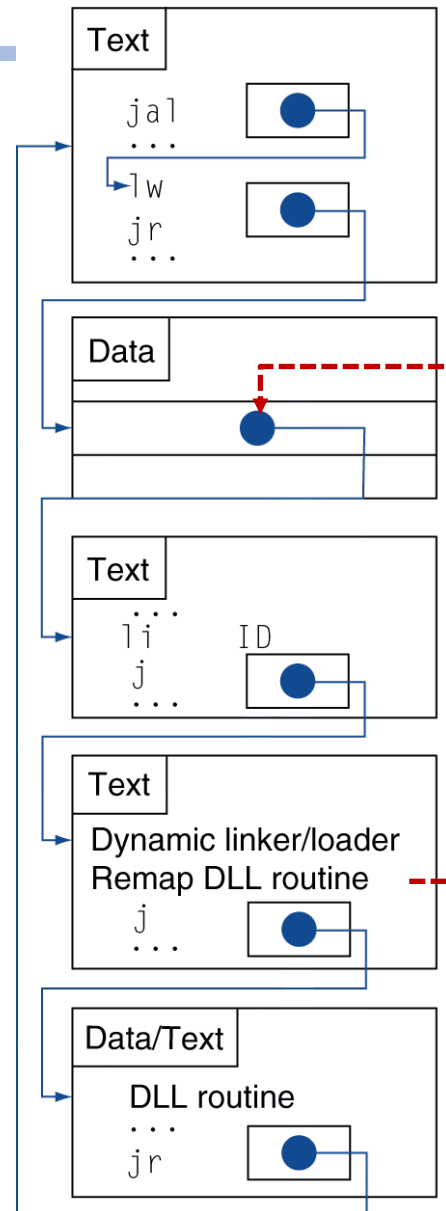
Lazy Linkage

Indirection table

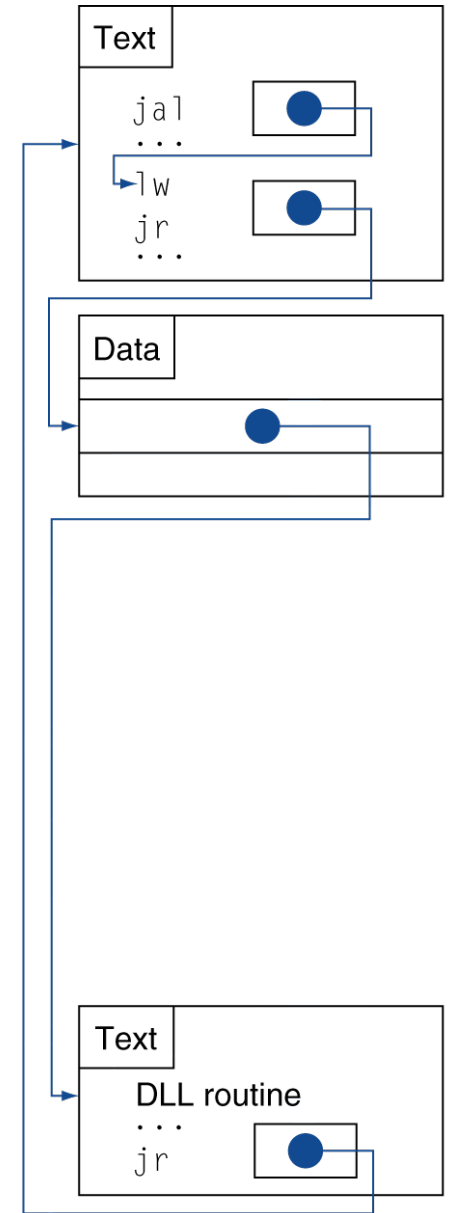
Stub: Loads routine ID,
Jump to linker/loader

Linker/loader code

Dynamically
mapped code

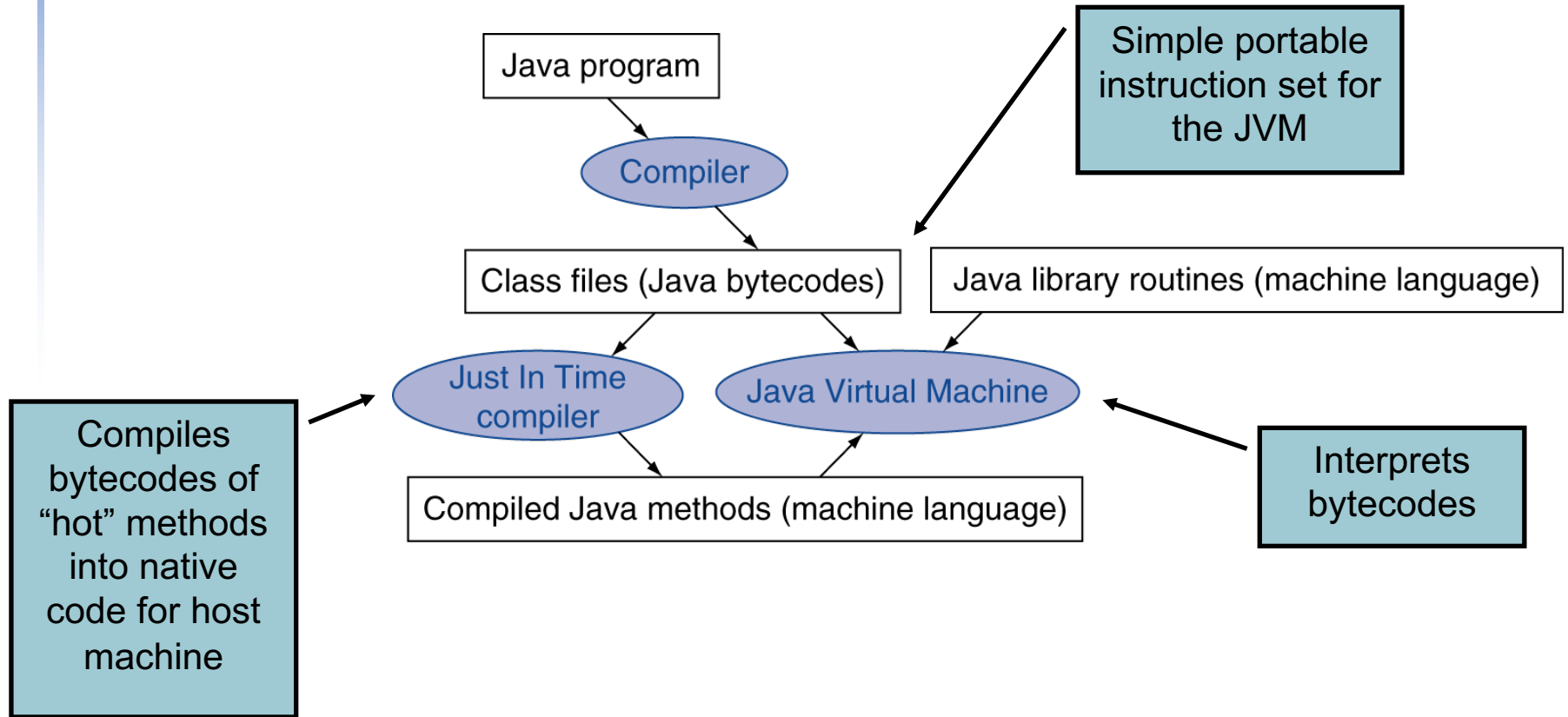


a. First call to DLL routine



b. Subsequent calls to DLL routine

Starting Java Applications



C Sort Example

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

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

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

The Procedure Swap

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

The Sort Procedure in C

- Non-leaf (calls swap)

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

- v 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 params
move \$s0, \$zero # i = 0 for1tst: slt \$t0, \$s0, \$s3 # \$t0 = 0 if \$s0 ≥ \$s3 (i ≥ n)		Outer loop
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		Inner loop
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		Pass params & call
addi \$s1, \$s1, -1 # j -= 1 j for2tst # jump to test of inner loop		Inner loop
exit2: addi \$s0, \$s0, 1 # i += 1 j for1tst # jump to test of outer loop		Outer loop

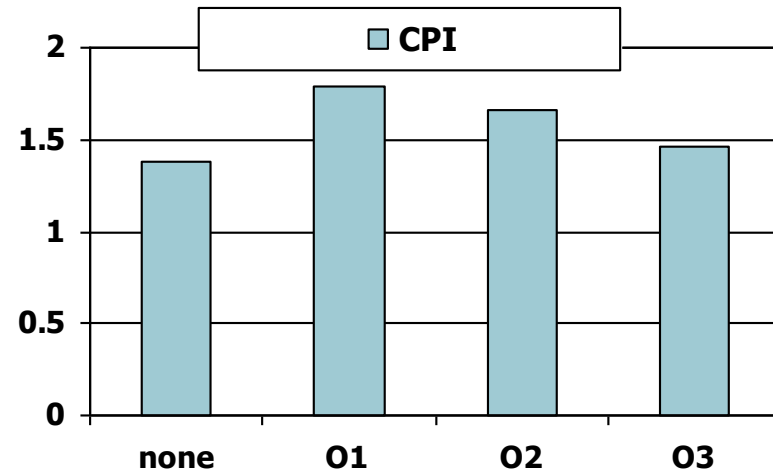
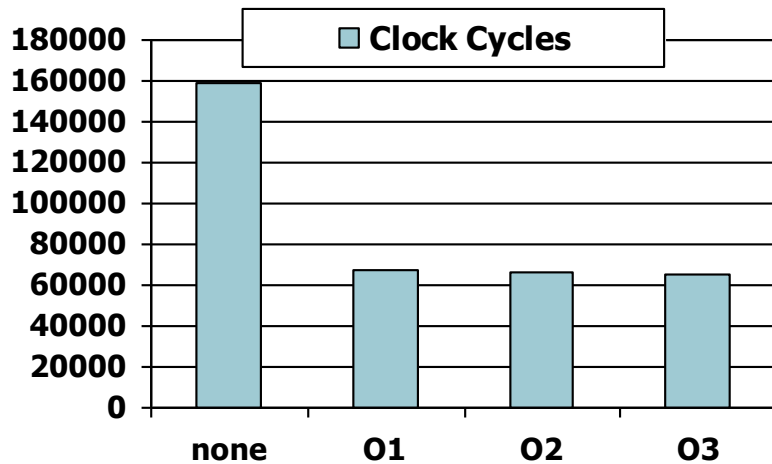
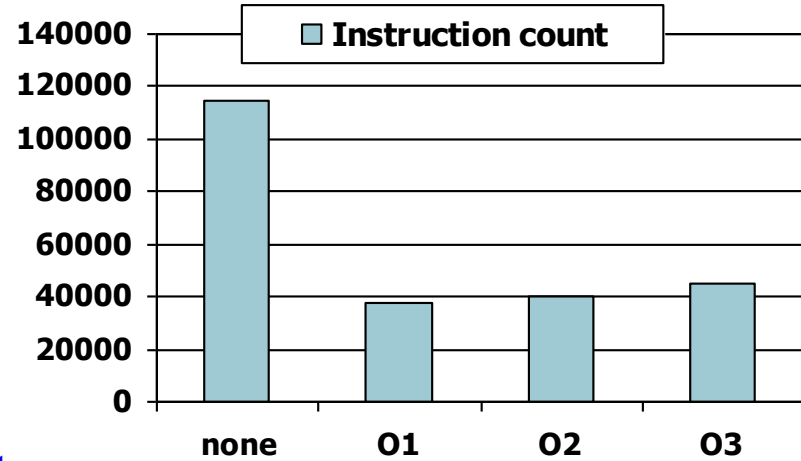
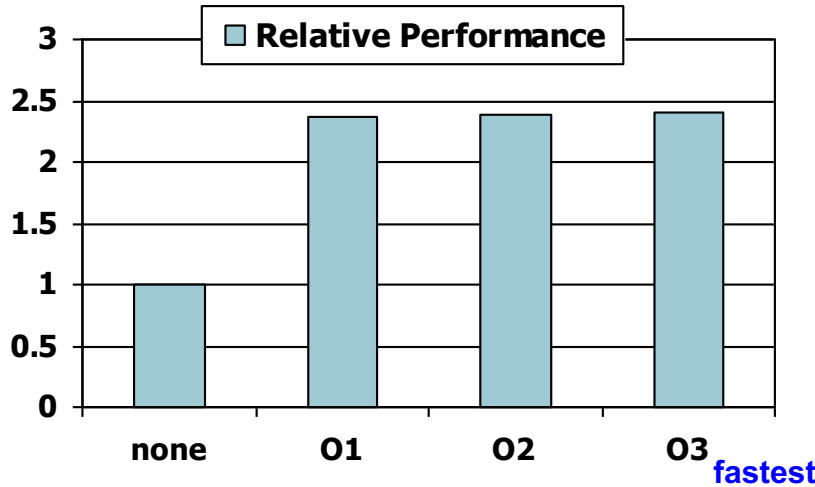
The Full Procedure

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

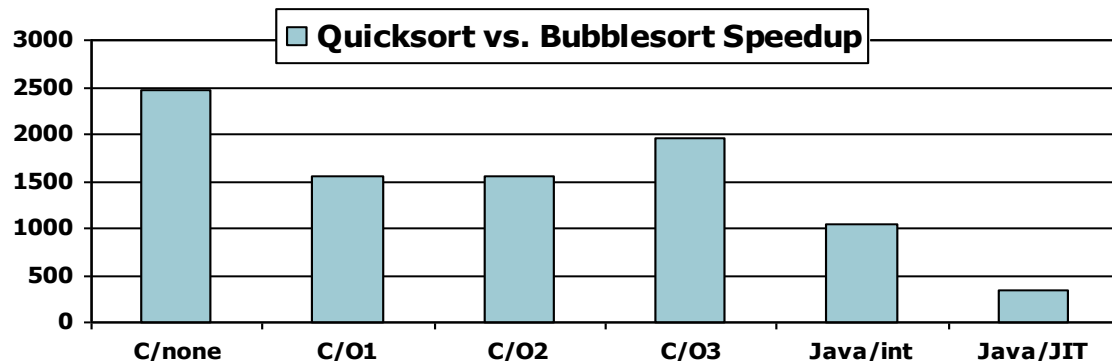
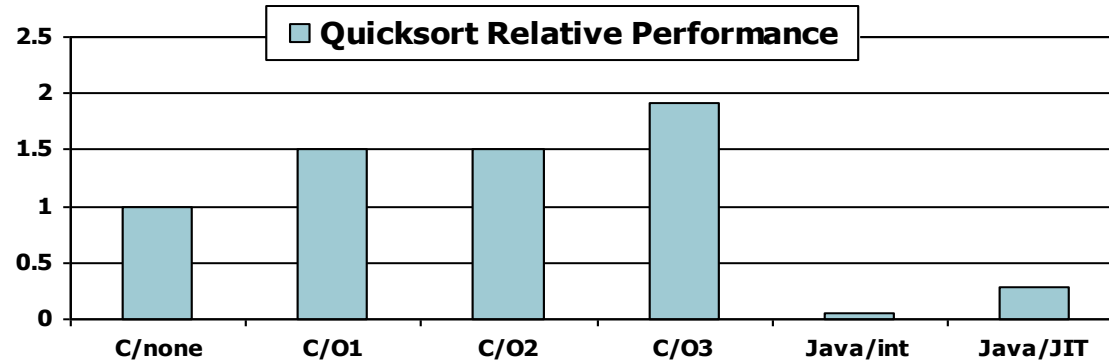
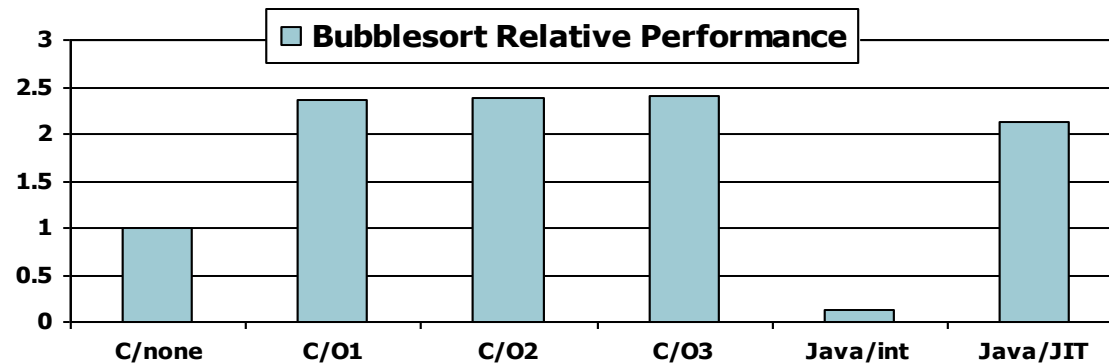
Effect of Compiler Optimization

Compiled with gcc for Pentium 4 under Linux

sorting 100K words



Effect of Language and Algorithm



Lessons Learnt

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

Arrays vs. Pointers

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

Example: Clearing and Array

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

```
        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
```

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

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

Comparison of Array vs. Ptr

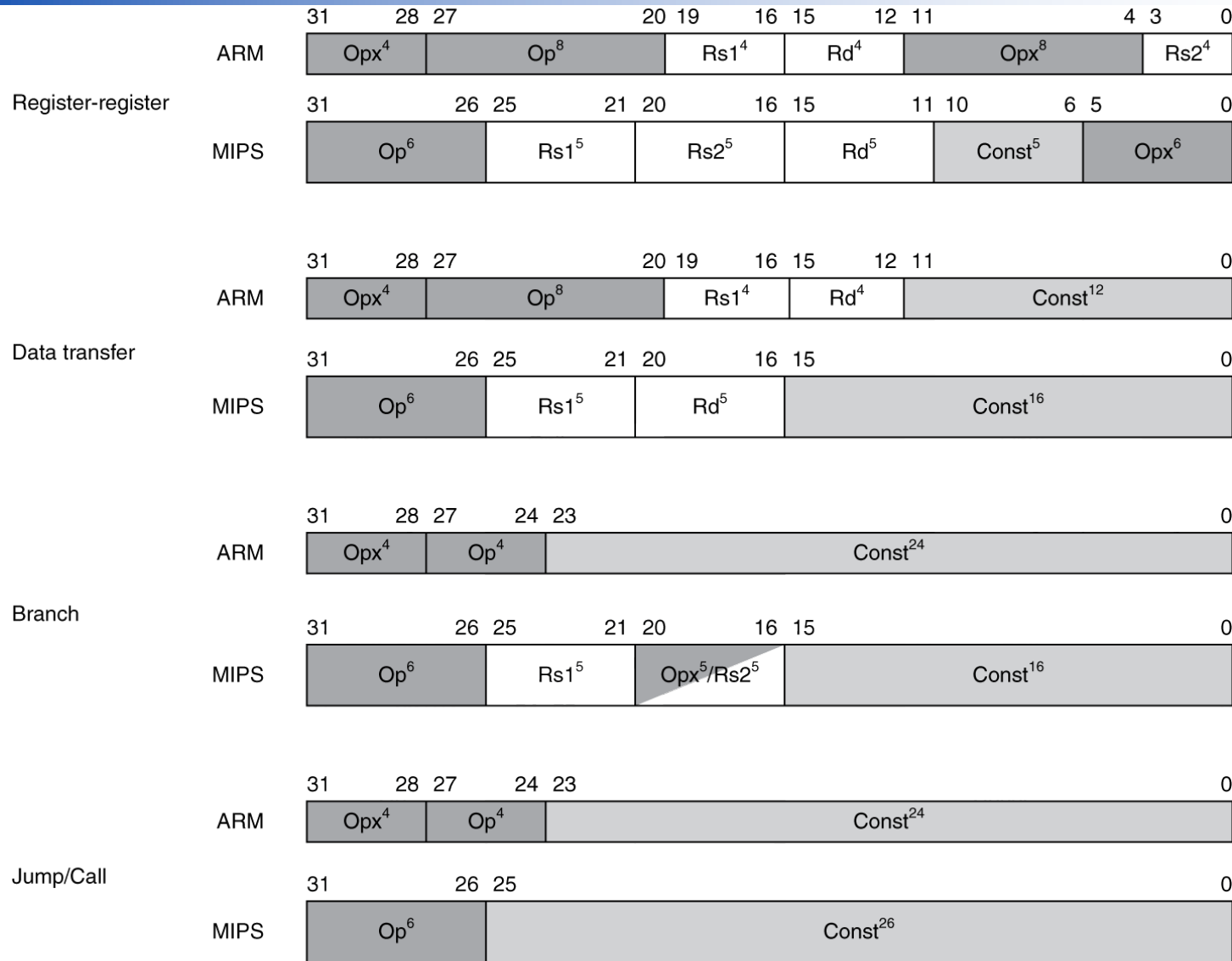
- Multiply “strength reduced” to shift
- Array version requires shift to be inside loop
 - Part of index calculation for incremented i
 - c.f. incrementing pointer
- Compiler can achieve same effect as manual use of pointers
 - Induction variable elimination
 - Better to make program clearer and safer

ARM & MIPS Similarities

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

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

Instruction Encoding



The Intel x86 ISA

- Evolution with backward compatibility
 - 8080 (1974): 8-bit microprocessor
 - Accumulator, plus 3 index-register pairs
 - 8086 (1978): 16-bit extension to 8080 not general purpose register architecture
 - Complex instruction set (CISC)
 - 8087 (1980): floating-point coprocessor
 - Adds FP instructions and register stack
 - 80286 (1982): 24-bit addresses, MMU (memory management unit)
 - Segmented memory mapping and protection
 - 80386 (1985): 32-bit extension (now IA-32) Intel architecture 32-bit
 - Additional addressing modes and operations
 - Paged memory mapping as well as segments
 - nearly general purpose register architecture

The Intel x86 ISA

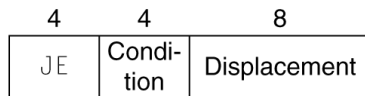
- Further evolution...
 - i486 (1989): pipelined, on-chip caches and FPU
 - Compatible competitors: AMD, Cyrix, ...
 - Pentium (1993): superscalar, 64-bit datapath
 - The infamous FDIV bug
 - Pentium Pro (1995), Pentium II (1997)
 - added MMX (Multi-Media eXtension) instructions
 - New microarchitecture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

The Intel x86 ISA

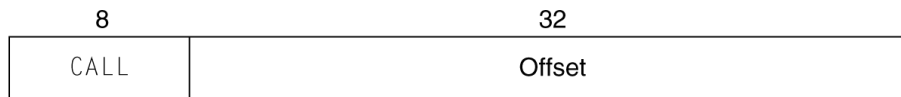
- And further...
 - AMD64 (2003): extended architecture to 64 bits
 - EM64T – Extended Memory 64 Technology (2004)
 - AMD64 adopted by Intel (with refinements)
 - Added SSE3 instructions
 - Intel Core (2006)
 - Added SSE4 instructions, virtual machine support
 - AMD64 (announced 2007): SSE5 instructions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

x86 Instruction Encoding

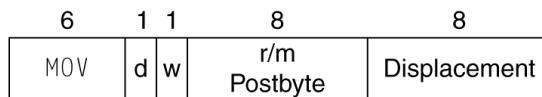
a. JE EIP + displacement



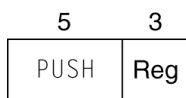
b. CALL



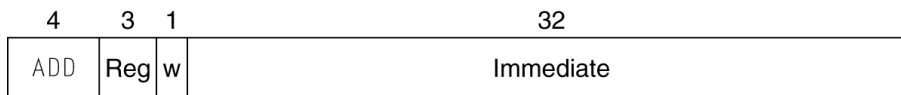
c. MOV EBX, [EDI + 45]



d. PUSH ESI



e. ADD EAX, #6765



f. TEST EDX, #42



■ Variable length encoding from 1 byte to 15 bytes

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

Implementing IA-32

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

ARM v8 Instructions

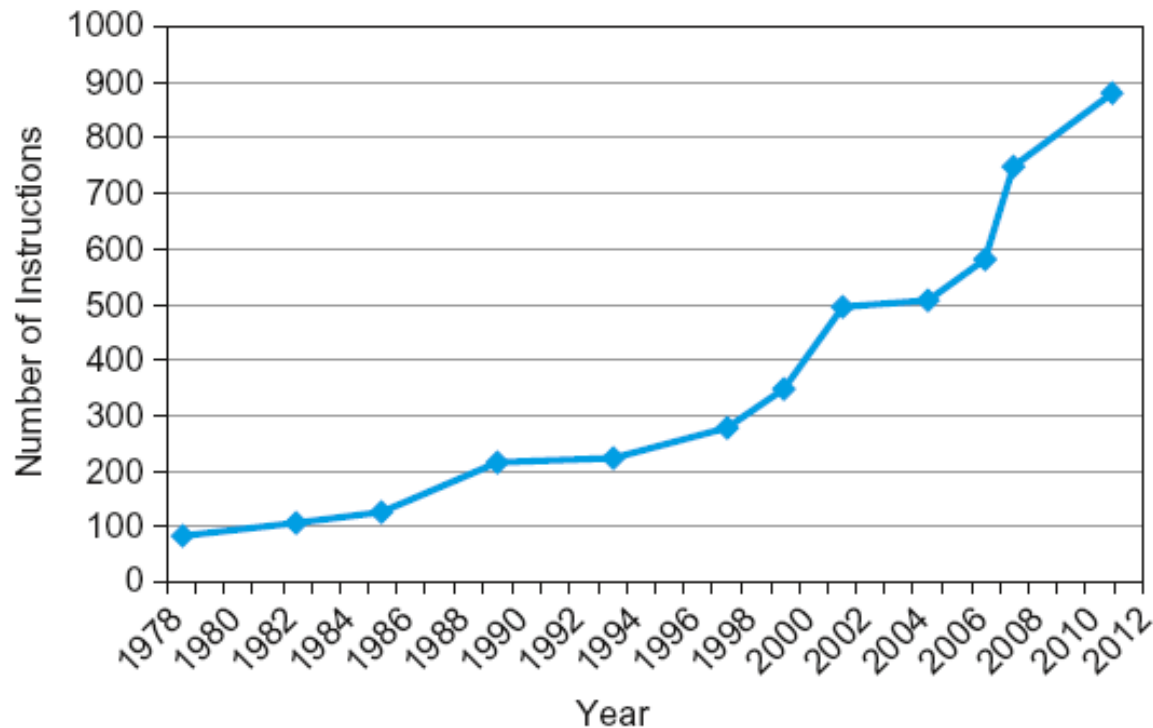
- In moving to 64-bit, ARM did a complete overhaul
- ARM v8 resembles MIPS
 - Changes from v7:
 - No conditional execution field
 - Immediate field is 12-bit constant
 - Dropped load/store multiple
 - PC is no longer a GPR
 - GPR set expanded to 32
 - Addressing modes work for all word sizes
 - Divide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

Fallacies (wrong)

- Powerful instruction \Rightarrow higher performance (X)
 - 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 (X)
 - But modern compilers are better at dealing with modern processors
 - More lines of code \Rightarrow more errors and less productivity

Fallacies

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



x86 instruction set

Pitfalls (risk)

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

Concluding Remarks

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

Concluding Remarks

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

Instruction class	MIPS examples	SPEC2006 Int	SPEC2006 FP
Arithmetic	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%



Thanks!