Chapter 2

Instructions: Language of the Computer

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

The MIPS Instruction Set

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

MIPS Instruction Set

- MIPS Instruction Types
  - Arithmetic
  - Logical operation
  - Data transfer
  - Flow control
- Design Principles
  - Design Principle 1: Simplicity favors regularity.
  - Design Principle 2: Smaller is faster.
  - Design Principle 3: Good design demands a compromise
  - Design Principle 4: Make common case fast.

Arithmetic Operations

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

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

- All arithmetic operations have this form
- *Design Principle 1:* Simplicity favors regularity
  - The natural number of operands for an operation like addition is three … requiring every instruction to have exactly three operands will keep the hardware simple.
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

Arithmetic Example

- C code:

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

- Compiled MIPS code:

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

Register 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: millions of locations

Register Operand Example

- C code:

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

  - g, …, j in $s1, …, $s4 ← moved from memory to registers by data transfer instructions.

- Compiled MIPS code:

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

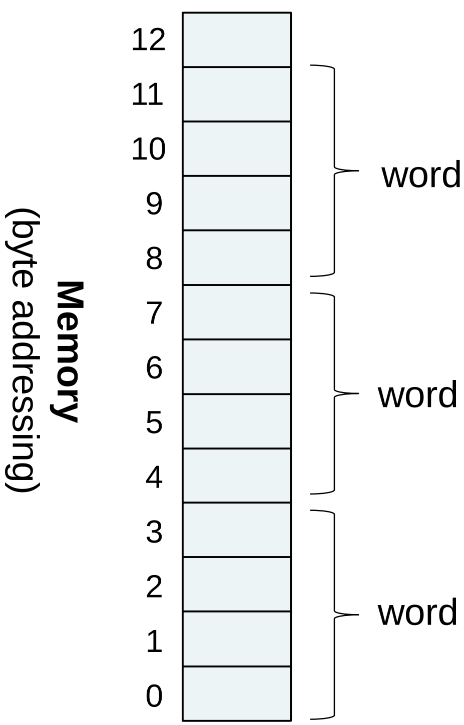
Memory Operands

- 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

Memory Operands

Word alignment

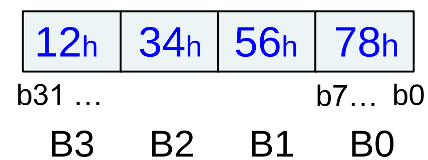
- Word address must be a multiple of 4



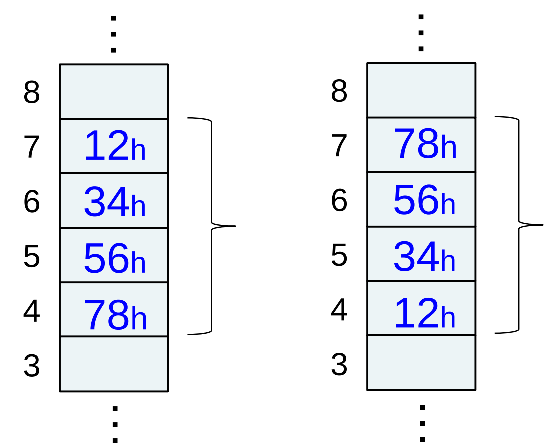
To store a word (32 bits)

0x12345678

In register ➔
(32 bits)



In memory ➔



Big Endian
(MIPS)

Little Endian

Data Transfer Operations

- C code:

```
g = h + A[8];
```
// g, h, A[] are type of int (4 bytes)

  - 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

Data Transfer Operations

- 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
```

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

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

Signed v.s. Unsigned

3 bits

Signed binary (2's complement) (10-based)	(Binary)	Unsigned binary (10-based)
0	0 0 0	0
1	0 0 1	1
2	0 1 0	2
3	0 1 1	3
-4	1 0 0	4
-3	1 0 1	5
-2	1 1 0	6
-1	1 1 1	7

4 bits

Signed binary (2's complement) (10-based)	(Binary)	Unsigned binary (10-based)
0	0 0 0 0	0
1	0 0 0 1	1
2	0 0 1 0	2
3	0 0 1 1	3
4	0 1 0 0	4
5	0 1 0 1	5
6	0 1 1 0	6
7	0 1 1 1	7
-8	1 0 0 0	8
-7	1 0 0 1	9
-6	1 0 1 0	10
-5	1 0 1 1	11
-4	1 1 0 0	12
-3	1 1 0 1	13
-2	1 1 1 0	14
-1	1 1 1 1	15

2's-Complement Signed Integers

- Bit 31 is sign bit
  - 1 for negative numbers
  - 0 for non-negative numbers
- Some specific numbers
  - Most-negative: 1000 0000 … 0000
  - Most-positive: 0111 1111 … 1111
  - 0: 0000 0000 … 0000
  - –1: 1111 1111 … 1111

```
  01111111111111
+              1
----------------
  10000000000000
```

- Positive numbers have the same unsigned and 2s-complement representation

Signed Negation

- Negate a signed integer: *Complement and add 1*
  - Complement means 1 → 0, 0 → 1

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

- Ex1: To negate +2
  - $+2 = 0000\ 0000\ \ldots\ 0010_2$
  - $-2 = 1111\ 1111\ \ldots\ 1101_2 + 1$
    $= 1111\ 1111\ \ldots\ 1110_2$
- Ex2: To negate -2

Ex3

How to represent -11 in 5-bit binary?

+11 binary → **01011**$_2$

Negation → **10101**$_2$

Sign Extension

- Purpose: Representing a number using more bits, but preserve the numeric value
- How: 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
- In MIPS instruction set, sign extension is needed.
  - Example:
    - `addi`: extend immediate value
    - `lb`, `lh`: extend loaded byte/halfword
    - `beq`, `bne`: extend the displacement

Representing Instructions

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

MIPS 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)

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

I-type:

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

- Immediate arithmetic and load/store instructions
  - rt: destination reg (I-type) or source reg (R-type)
  - 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

R-format v.s. I-format

R-type:

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

I-type:

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

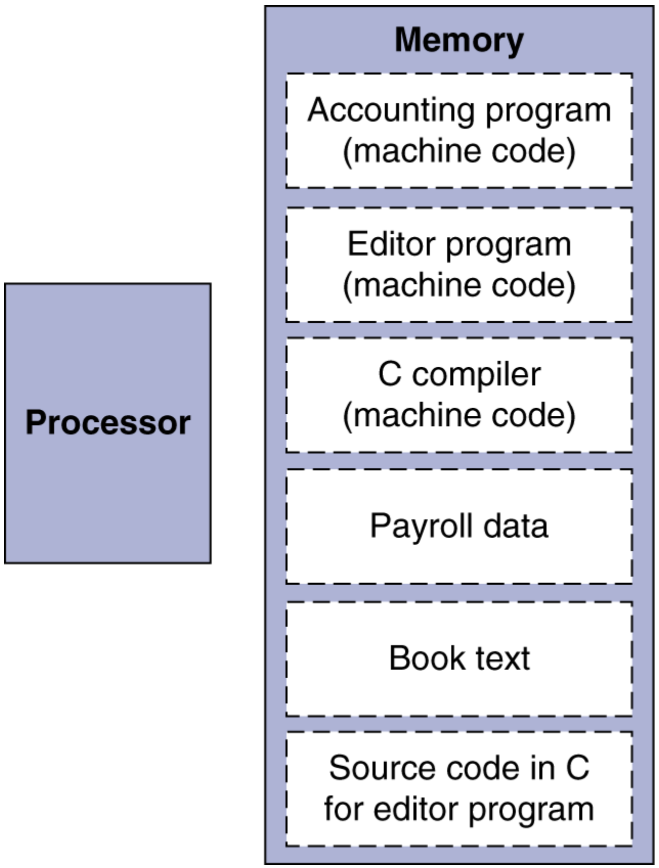
Example:

```
C code :          A[300] = h + A[300];
Assembly code:    lw  $t0, 1200($t1)   rs
                  add $t0, $s2, $t0
                  sw  $t0, 1200($t1)   rs
Machine code:
```

op	rs	rt	rd	Addr/ shamt	funct
35	9	8	1200		
0	18	8	8	0	32
45	9	8	1200		

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

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`

= NOT OR

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

If one operand of NOR is zero, then it is equivalent to NOT

Shift Operations

R-type:	op	rs	rt	rd	shamt	funct
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

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

- `sll $t2, $s0, 4`
- `srl $t2, $s0, 6`

rt

op	rs	rt	rd	shamt	funct
0	0	16	10	4	0
0	0	16	10	6	2

AND Operations

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

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

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

OR Operations

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

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

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

NOT Operations

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

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

Register 0: always read as zero

(use NOR to perform NOT)

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

Category	Instr	Example	Meaning	Comments
Logical	and	and \$s1, \$s2, \$s3	\$s1 = \$s2 & \$s3	bit-by-bit AND
	or	or \$s1, \$s2, \$s3	\$s1 = \$s2 | \$s3	bit-by-bit OR
	nor	nor \$s1, \$s2, \$s3	\$s1 = ~(\$s2|\$s3)	bit-by-bit NOR
	andi	andi \$s1, \$s2, 100	\$s1 = \$s2 & 100	AND with const
	ori	ori \$s1, \$s2, 100	\$s1 = \$s2 | 100	OR with const.
	sll	sll \$s1, \$s2, 10	\$s1 = \$s2 << 10	shift left by const
	srl	srl \$s1, \$s2, 10	\$s1 = \$s2 >> 10	shift right by const

Assembly code

MIPS
Logical Instruction

Machine code

Name	Format	op	rs	rt	rd	shamt	func
and	R	0	18	19	17	0	36
or	R	0	18	19	17	0	37
nor	R	0	18	19	17	0	39
andi	I	12	18	17	100		
ori	I	13	18	17	100		
sll	R	0	0	18	17	10	0
srl	R	0	0	18	17	10	2

Flow Control Operations

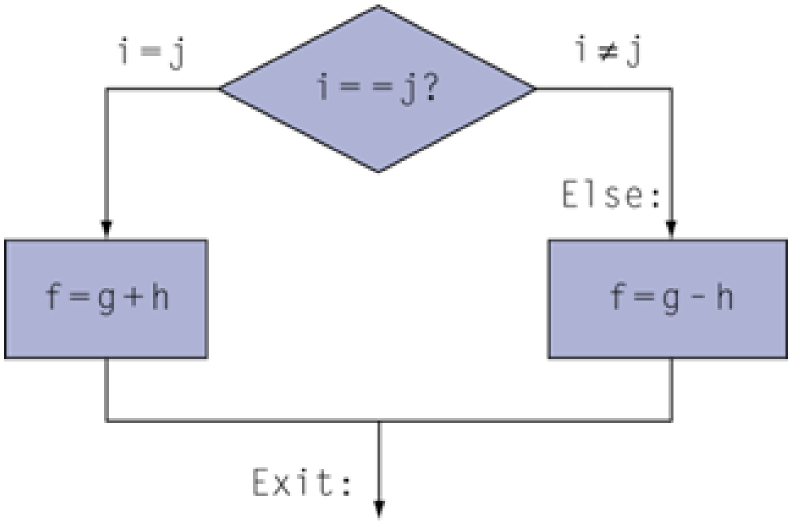
- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- `beq rs, rt, L1`
  - if (rs == rt) branch to instruction labeled L1;
- `bne rs, rt, L1`
  - if (rs != rt) branch to instruction labeled L1;
- `j L1`
  - unconditional jump to instruction labeled L1

Compiling If Statements

- C code:

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

  - f, g, … in $s0, $s1, …



- Compiled MIPS code:

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

Assembler calculates addresses

Compiling Loop Statements

- C code:

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

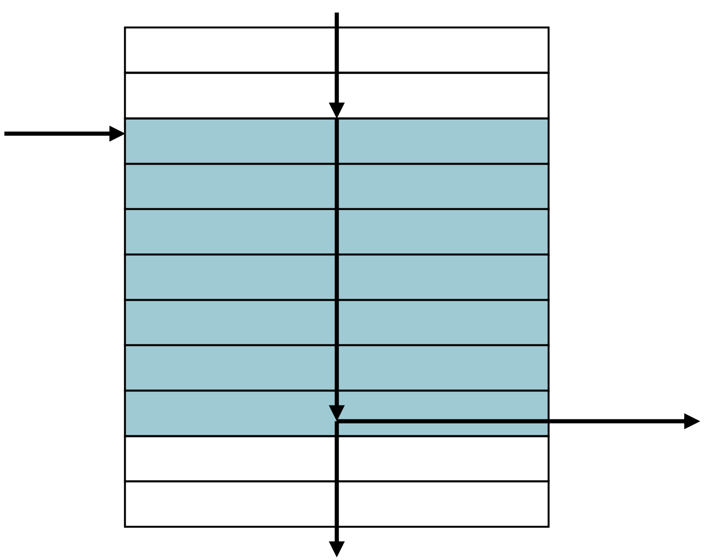
  - i in $s3, k in $s5, address of save in $s6

- MIPS code:

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

Basic Blocks

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



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

Critical thinking…

- We have used beq/bne for structures below

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

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

- How about following structures?

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

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

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
bne $t0, $zero, L
```

```
if ($s1<$s2)
branch to L
```

(blt)

Branch Instruction Design

- Why not `blt`, `bgt`, 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

Critical thinking …

```
slt $t0, $s1, $s2
bne $t0, $zero, L
```

```
if ($s1<$s2)
branch to L
```

(blt)

?

```
if ($s1>$s2)
branch to L
```

(bgt)

?

```
if ($s1<=$s2)
branch to L
```

(ble)

?

```
if ($s1>=$s2)
branch to L
```

(bge)

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**

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

Register 1 ($at) reserved for assembler, 26-27 for operating system

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

```
     …
     jal  Proc1
     xxx

Proc1:
     …
     jr  $ra
```

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
  sw   $s0, 0($sp)
  add  $t0, $a0, $a1
  add  $t1, $a2, $a3
  sub  $s0, $t0, $t1
  add  $v0, $s0, $zero
  lw   $s0, 0($sp)
  addi $sp, $sp, 4
  jr   $ra
```

Save $s0 on stack

Procedure body

Restore $s0

Return

Non-Leaf Procedures

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

Non-Leaf Procedure Example

- **C code:**

```
int fact (int n)
{
  if (n < 1) return 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     # branch if n >= 1
    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
```

Non-Leaf Pro

- **MIPS code:**

```
fact:
      addi $sp, $sp, -8
      sw   $ra, 4($sp)
      sw   $a0, 0($sp)
      slti $t0, $a0, 1
      beq  $t0, $zero, L1
      addi $v0, $zero, 1
      addi $sp, $sp, 8
      jr   $ra
L1:   addi $a0, $a0, -1
      jal  fact
M →   lw   $a0, 0($sp)
      lw   $ra, 4($sp)
      addi $sp, $sp, 8
      mul  $v0, $a0, $v0
      jr   $ra
```

a0 = 3
jal (ra = jal+4 in main)
fact
…
a0 = 2
jal (ra = M)
fact
…
a0 = 1
jal (ra = M)
fact
…
a0 = 0
jal (ra = M)
fact

ra (in main) a0 = 3
ra (M) a0 = 2
ra (M) a0 = 1
ra (M) a0 = 0

v0=1 pop stack

pop stack (a0=1, ra=M)
v0= 1x1 = 1
pop stack (a0=2, ra=M)
v0= 2x1 = 2
pop stack (a0=3, ra:in main)
v0= 3x2 = 6

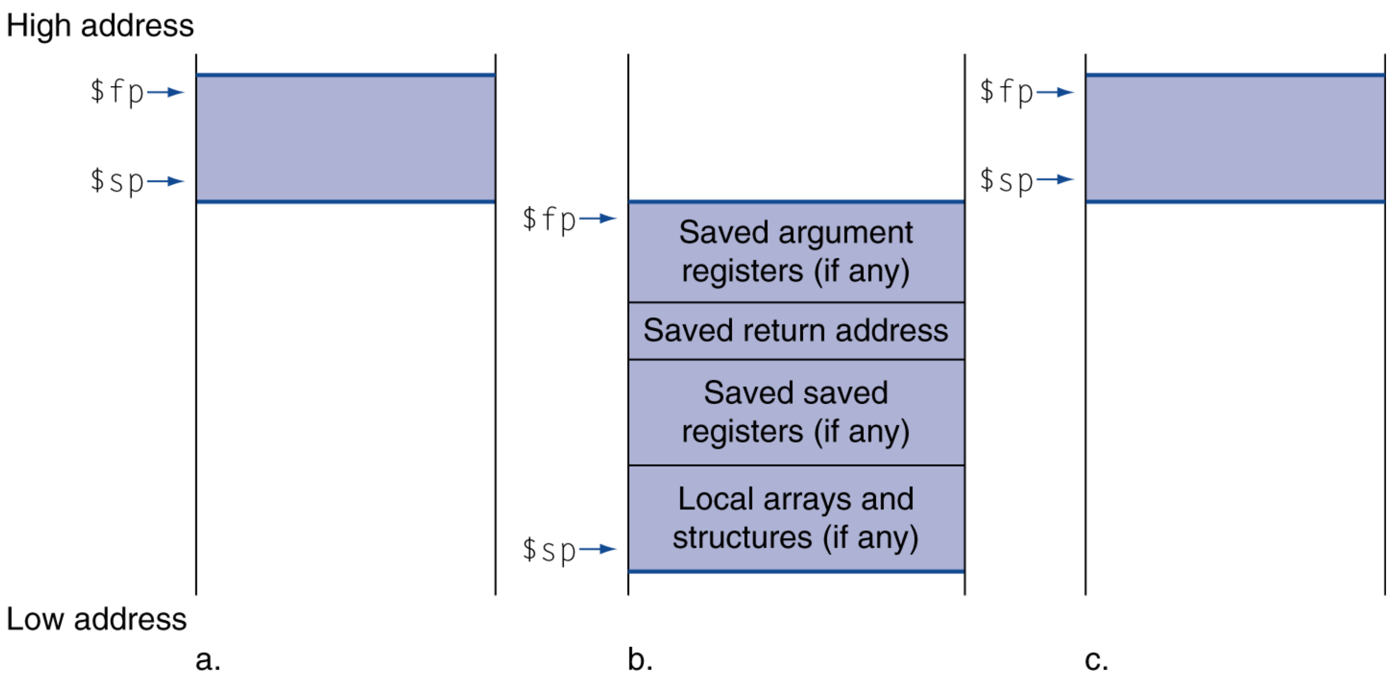
Procedure – cont.

- Data preservation in MIPS
  - To avoid saving and restoring a register whose value is never used, MIPS registers are separated to two groups.

Preserved	Not preserved
Saved registers: \$s0 - \$s7	Temporary registers: \$t0 - \$t9
Stack pointer register: \$sp	Argument registers: \$a0 - \$a3
Return address register: \$ra	Return value registers: \$v0 - \$v1
Stack above the stack pointer	Stack below the stack pointer

  - Preserved: The caller assumes them will be preserved on a procedure call. If they are used in the procedure, the callee must save and restore them.
    - In leave procedure, we need to save and restore \$s0

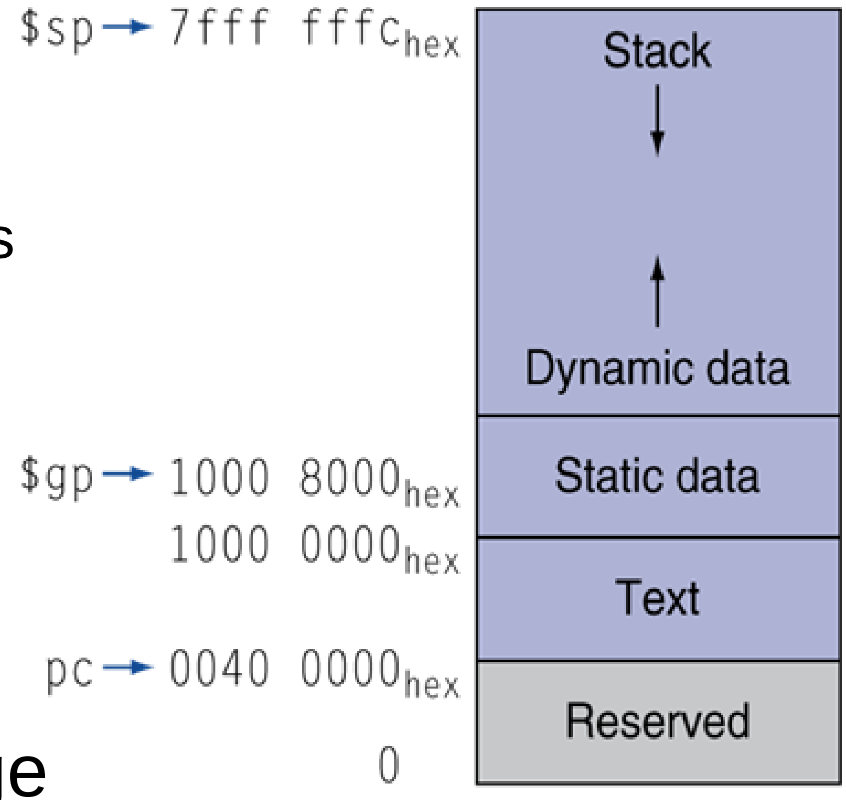
Local Data on the Stack



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

Memory Layout

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



Character Data

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

Byte/Halfword Operations

- MIPS byte/halfword load/store
  - String processing is a common case
  - Sign extend to 32 bits in rt

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

  - Zero extend to 32 bits in rt

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

  - Store just rightmost byte/halfword

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

String Copy Example

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

```
void strcpy (char x[], char y[])
{
  int 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

- How to load the 32-bit constant below into register $s0 using MIPS assembly code?

- **0000 0000 0011 1101 0000 1001 0000 0000₂**

lw $t0, 0($t1)

address of the constant

How to move the address to the reg?

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

$61 = 0000\ 0000\ 0011\ 1101_2$

`lui $s0,61`	0000 0000 0011 1101	0000 0000 0000 0000
`ori $s0,$s0,2304`	0000 0000 0011 1101	0000 1001 0000 0000

$2304 = 0000\ 1001\ 0000\ 0000_2$

32-bit Constants

- How to load the 32-bit constant below into register $s0 using MIPS assembly code?

0000 0000 0011 1101   0000 1001 0000 0000

```
lui $s0,0000000000111101b
ori $s0,$s0,0000100100000000b
```

0000 0000 0011 1101	0000 1001 0000 0000

Branch Addressing

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

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

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

Jump Addressing

- Jump targets could be anywhere in text segment
  - Encode full address in instruction

J format

op	Address (word)
6 bits	26 bits

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

(十進位)

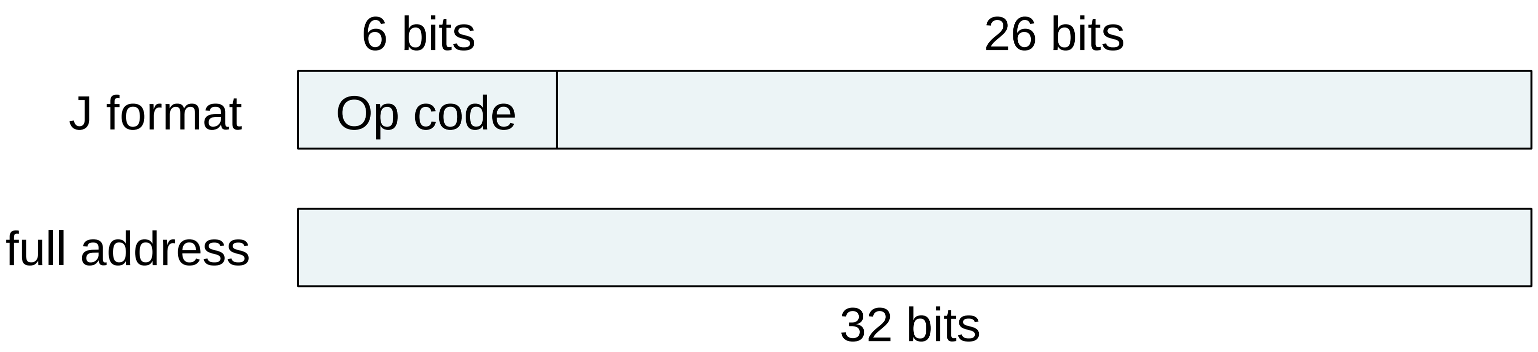
10000 L: …

…

J L	2	2500 (word address)					Go to 2500 (word)
jal L	3	2500 (word address)					$ra = PC (cur+4); go to 2500 (word)
Jr $ra	0	31	0	0	0	8	Go to $ra

Jump Addressing

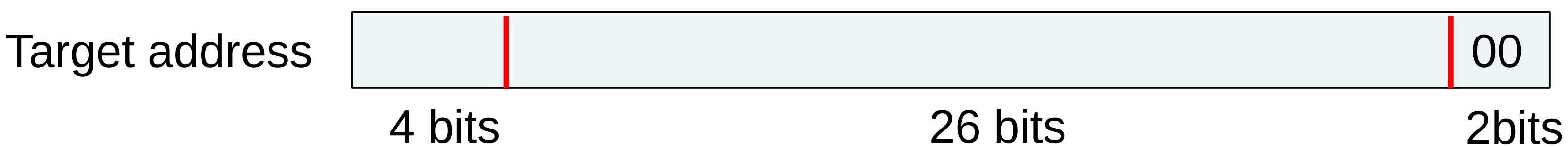
1. Why the jump addressing is so complex as to concatenate the 26 bits in the instruction and 4 bits from PC?



The space is insufficient to store a full address.

2. Why only needs 4 bits from PC?

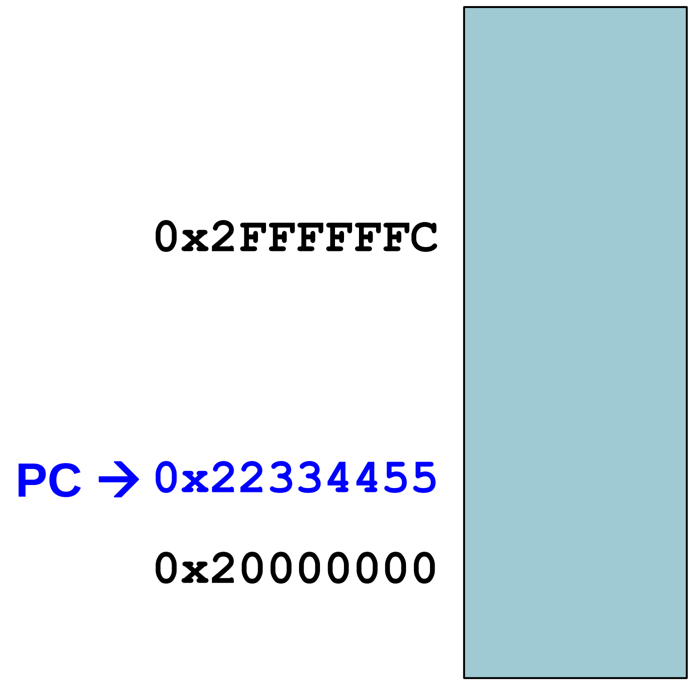
Instr. are word alignment → target address is a multiple of 4
→ rightmost two bits are 00 → 32 - (26+2) = 4



Jump Addressing

3. How to make sure that the first 4 bits of the target address is the same to the first 4 bits of PC?

To limit the range that j instruction can reach.



4. What if we need to go to the address with the first 4 bits different from the first 4 bits of PC?

With the help of beq

Target Addressing Example

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

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

(十進位)

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

Branching Far Away

- What is the range that *beq* can reach?
  - +/- 2^15 (word) = +/- 2^17 (bytes) = +/-128k (PC relative)
- What is the range that *j* can reach?
  - 2^26 (word) = 2^28 (bytes) = 2^8 MB = 256MB address where the PC supplies the upper 4bits
- If branch target is too far to encode with 16-bit offset, assembler rewrites the code

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

Overview of Instruction Formats

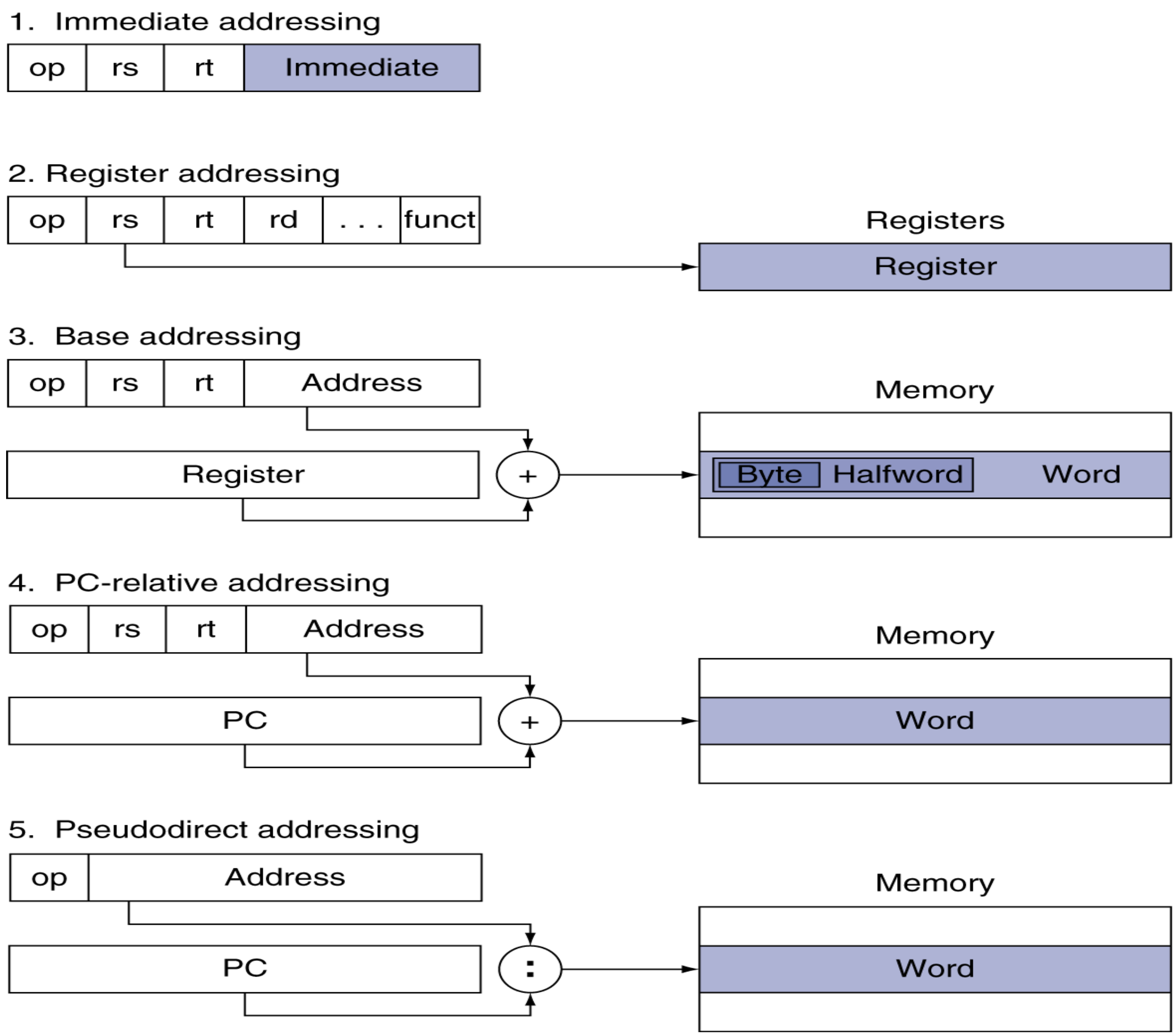
- MIPS instr formats:
  - Simple instructions all 32 bits wide
  - Very structured, no unnecessary baggage
  - Three instruction formats:

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

Addressing Mode Summary

1. **Immediate addressing**
   - the operand is a constant inside the instr.
2. **Register addressing**
   - the operand is a register
3. **Base or displacement addressing**
   - the operand is at the memory location whose addr. is the sum of a reg and a constant inside instr. (lw/sw/lh/sh/lb/sb)
4. **PC-relative addressing**
   - the address is a 16-bit constant in instr., shifted left 2 bits (word to byte), and then added to (PC+4). (i.e., next instr.)
5. **Pseudodirect addressing**
   - the address is a 26-bit constant in instruction, shifted left 2 bits (word to byte), and then concatenated with the 4 upper bits of PC. Address boundary: 2^{28} = 256MB.

Addressing Mode Summary



Put it all together – Sort

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

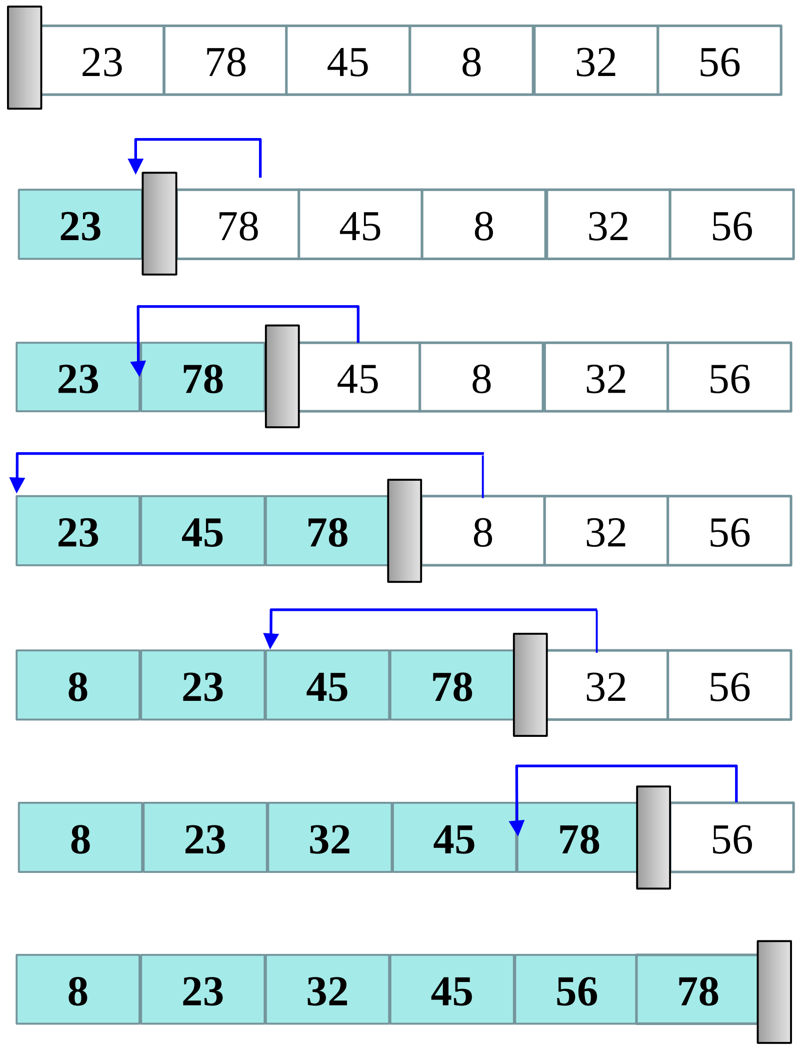
```
void swap(int v[], int k)
{
  int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

  - 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 Insertion Sort Algorithm



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, k 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
```

Lines 1–2	Move params
Lines 3–4	Outer loop
Lines 5–14	Inner loop
Lines 15–17	Pass params & call
Lines 18–19	Inner loop
Lines 20–21	Outer loop

The Full Procedure

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

Put it all together – 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

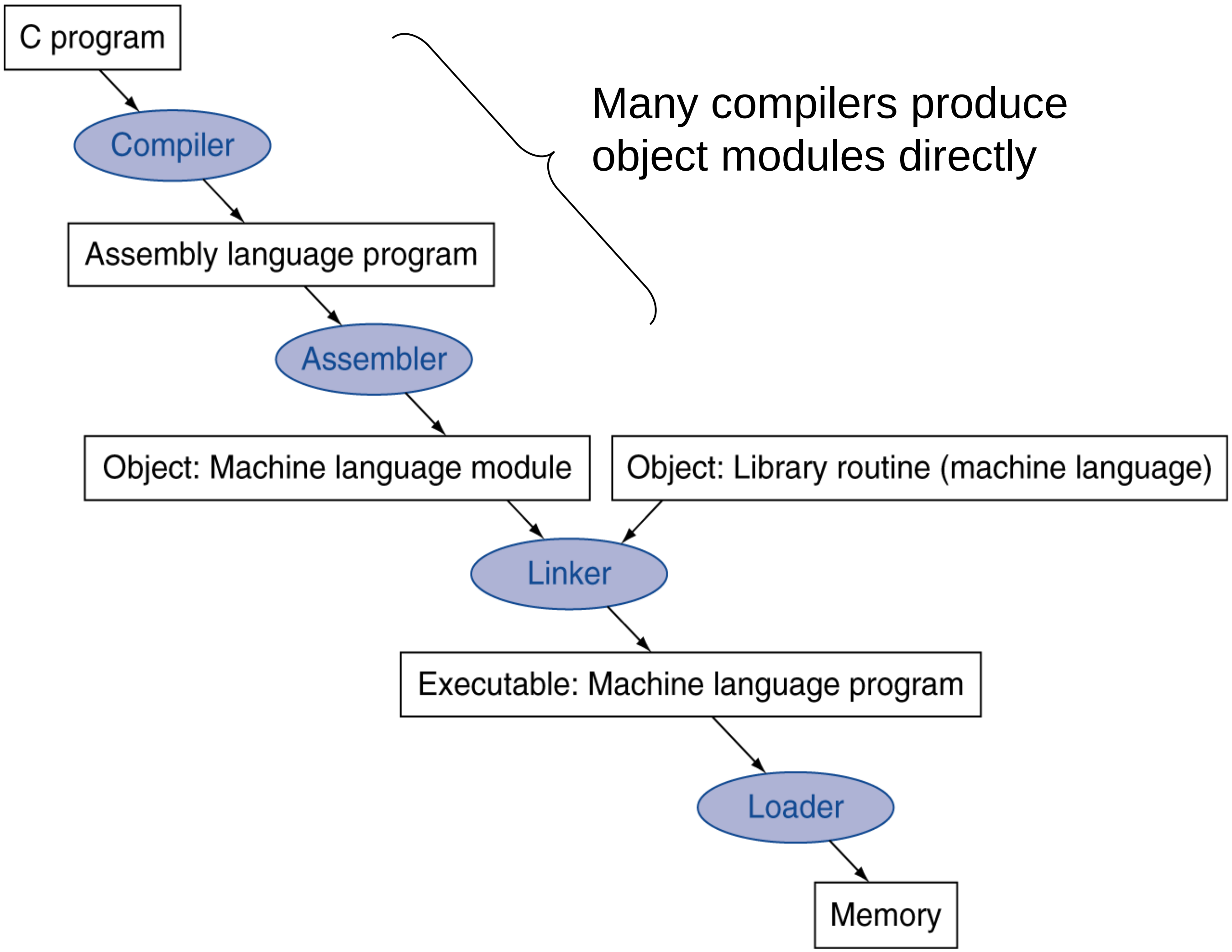
Example: Clearing an Array

<pre>clear1(int array[], int size) { int i; for (i = 0; i < size; i += 1) array[i] = 0; }</pre>	<pre>clear2(int *array, int size) { int *p; for (p = &array[0]; p< &array[size]; p = p + 1) *p = 0; }</pre>
<pre> 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</pre>	<pre> 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</pre>

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

Translation and Startup



Many compilers produce object modules directly

Static linking

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

Producing an Object Module

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

Linking Object Modules

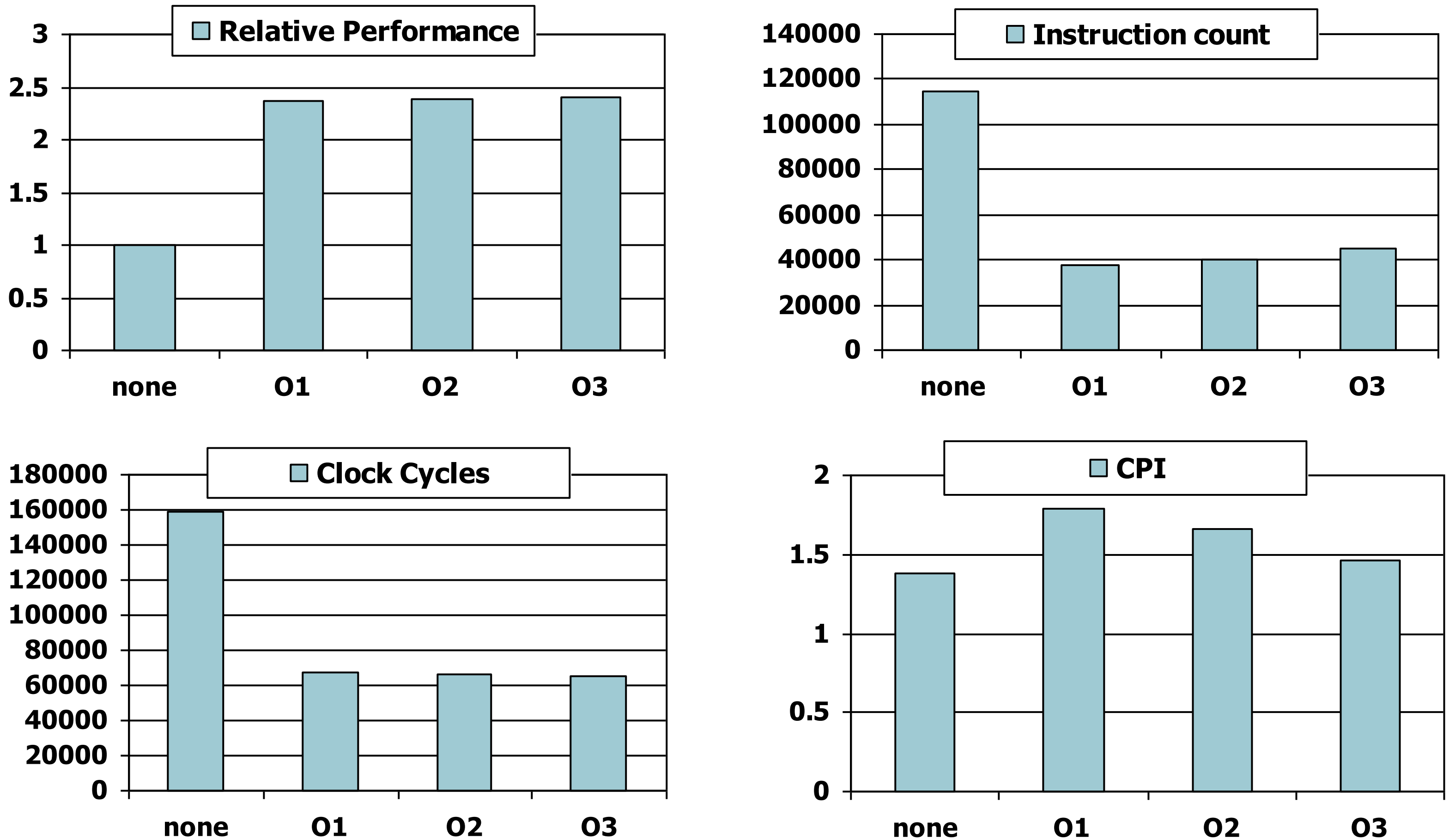
- Produces an executable image
  1. Merges segments
  2. Resolve labels (determine their addresses)
  3. Patch location-dependent and external refs
- Could leave location dependencies for fixing by a relocating loader
  - But with virtual memory, no need to do this
  - Program can be loaded into absolute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

Loading a Program

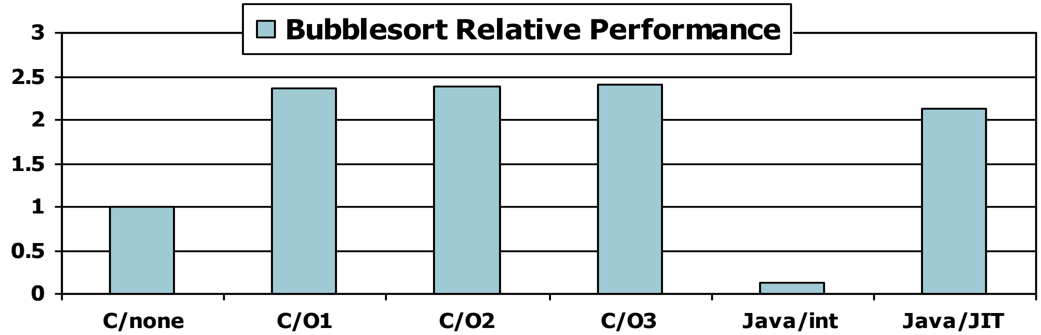
- Load from image file on disk into memory
  1. Read header to determine segment sizes
  2. Create virtual address space
  3. Copy text and initialized data into memory
     - Or set page table entries so they can be faulted in
  4. Set up arguments on stack
  5. Initialize registers (including $sp,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

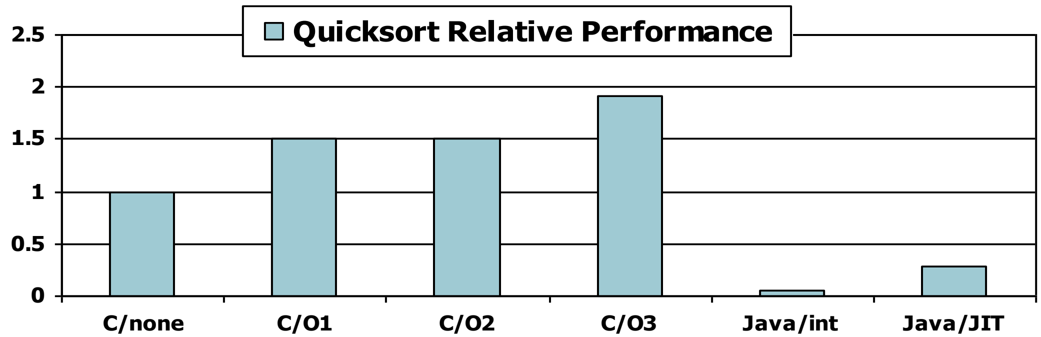
Effect of Compiler Optimization

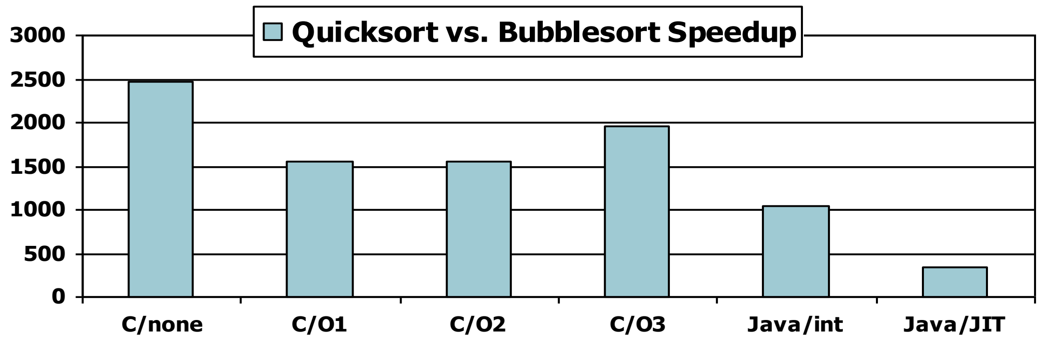
Compiled with gcc for Pentium 4 under Linux



Effect of Language and Algorithm







Lessons Learnt

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

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

Implementing IA-32

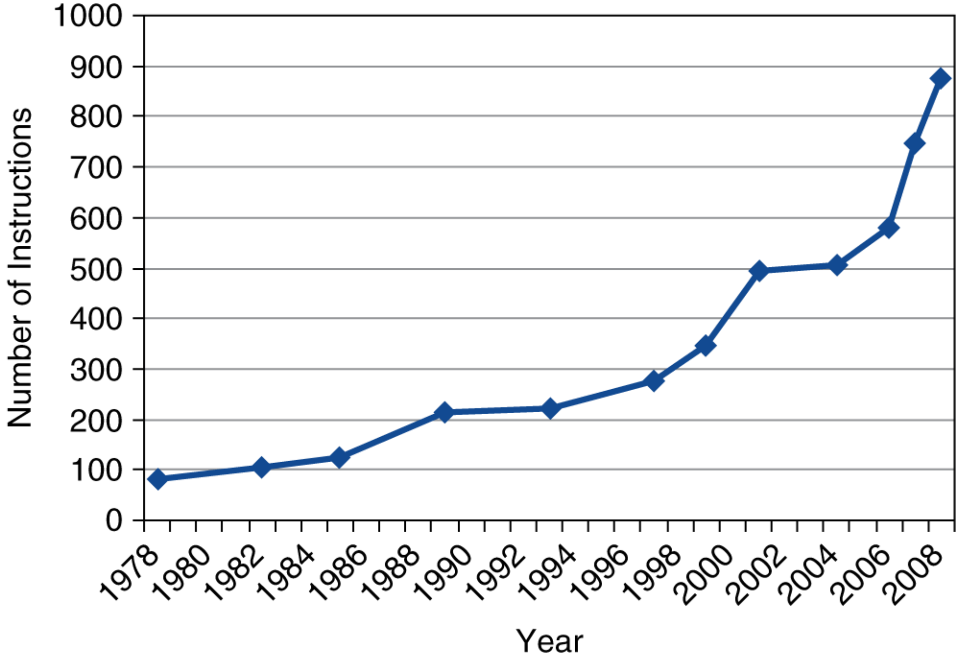
- Complex instruction set makes implementation difficult
  - Hardware translates instructions to simpler microoperations
    - Simple instructions: 1–1
    - Complex instructions: 1–many
  - Microengine similar to RISC
  - Market share makes this economically viable
- Comparable performance to RISC
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

Fallacies

- Powerful instruction ⇒ 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 ⇒ more errors and less productivity

Fallacies

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



x86 instruction set

Pitfalls

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

Concluding Remarks

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

Concluding Remarks

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

Instruction class	MIPS examples	SPEC2006 Int	SPEC2006 FP
Arithmetic	`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%