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

MIPS operands

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

The MIPS Instruction Set

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three register operands
	subtract	sub $s1,$s2,$s3	$s1 = $s2 - $s3	Three register operands
	add immediate	addi $s1,$s2,20	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	lw $s1,20($s2)	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	sw $s1,20($s2)	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	lh $s1,20($s2)	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	lhu $s1,20($s2)	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	sh $s1,20($s2)	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	lb $s1,20($s2)	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	lbu $s1,20($s2)	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	sb $s1,20($s2)	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	ll $s1,20($s2)	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	sc $s1,20($s2)	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	lui $s1,20	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	and $s1,$s2,$s3	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	or $s1,$s2,$s3	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	nor $s1,$s2,$s3	$s1 = ~($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	andi $s1,$s2,20	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	ori $s1,$s2,20	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	sll $s1,$s2,10	$s1 = $s2 << 10	Shift left by constant
	shift right logical	srl $s1,$s2,10	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	beq $s1,$s2,25	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne $s1,$s2,25	if ($s1!= $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for beq, bne
	set on less than unsigned	sltu $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than unsigned
	set less than immediate	slti $s1,$s2,20	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
	set less than immediate unsigned	sltiu $s1,$s2,20	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant unsigned
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr $ra	go to $ra	For switch, procedure return
	jump and link	jal 2500	$ra = PC + 4; go to 10000	For procedure call
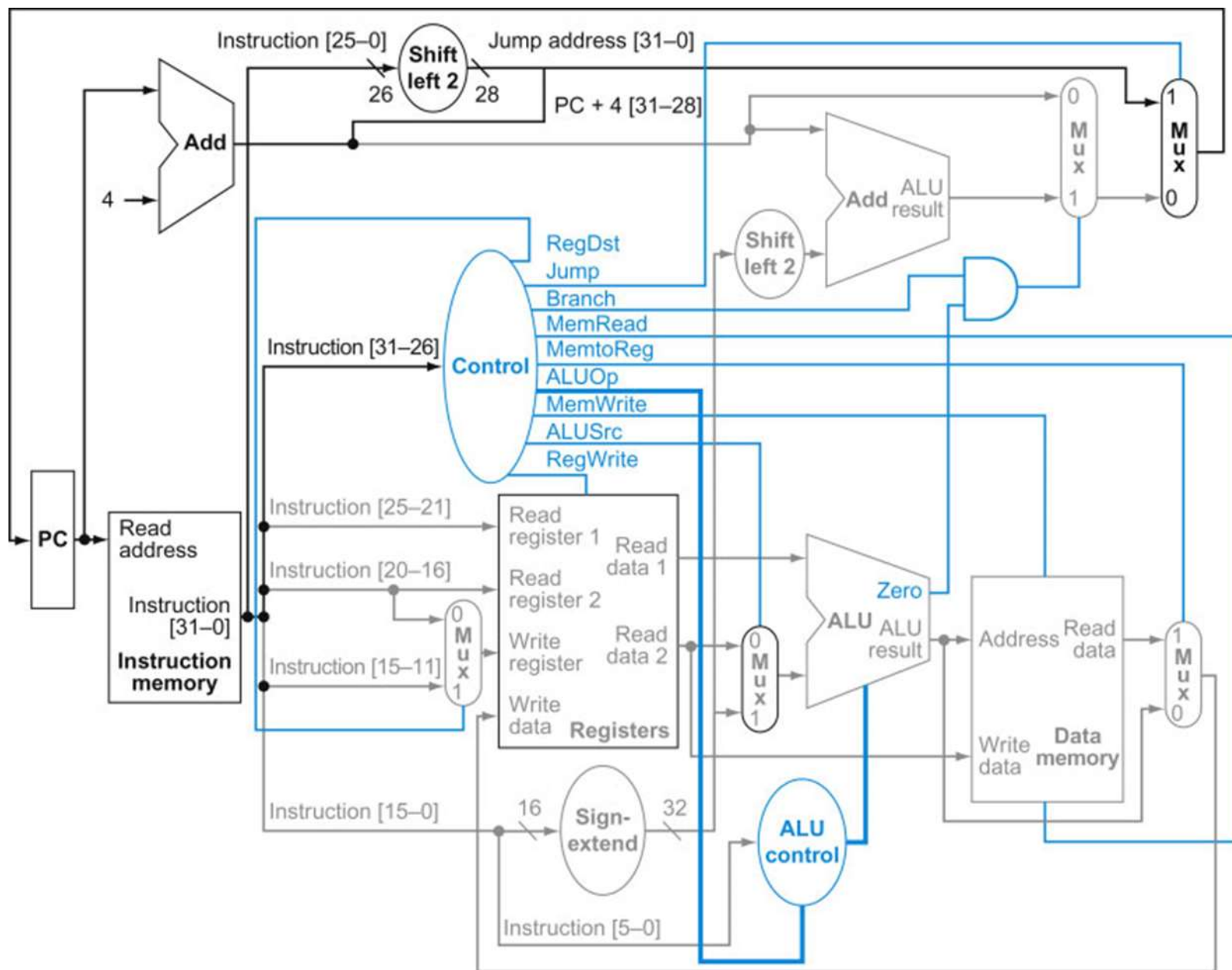
Arithmetic Operations

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

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

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

Arithmetic Example

- C code:

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

- Compiled MIPS code:

  ```
  add t0, 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);
```
  - 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

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

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`

Unsigned Binary Integers

■ Given an n-bit number

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

■ Range: 0 to $+2^n - 1$

■ Example

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

■ Using 32 bits

- 0 to +4,294,967,295

2s-Complement Signed Integers

- Given an n-bit number

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

- 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 $+2{,}147{,}483{,}647$

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 $1 \rightarrow 0$, $0 \rightarrow 1$

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

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

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

Sign Extension

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

Representing Instructions

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

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

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

Stored Program Computers

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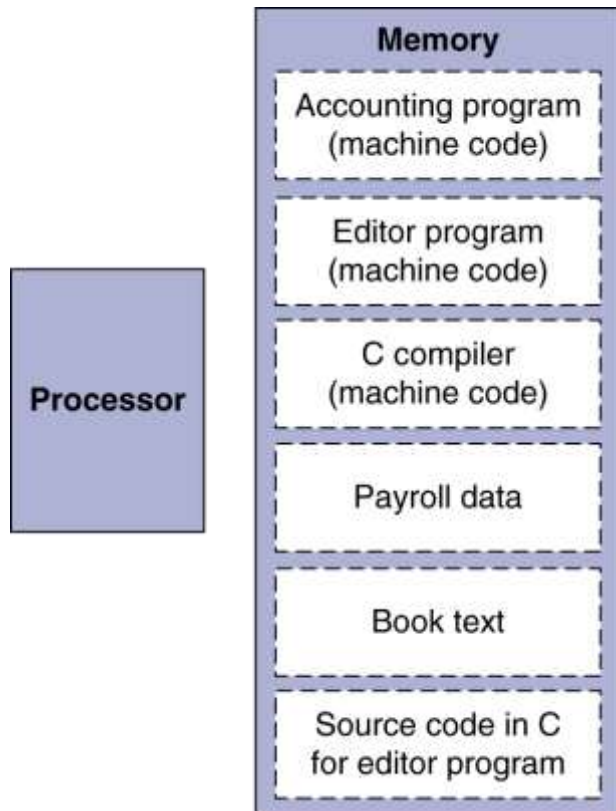
- Instructions represented in binary, just like data
- Instructions and data stored in memory
- Programs can operate on programs
  - e.g., compilers, linkers, …
- Binary compatibility allows compiled programs to work on different computers
  - Standardized ISAs

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

■ **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

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

AND Operations

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

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

$t2 | 0000 0000 0000 0000 0000 1101 1100 0000

$t1 | 0000 0000 0000 0000 0011 1100 0000 0000

$t0 | 0000 0000 0000 0000 0000 1100 0000 0000

OR Operations

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

`or $t0, $t1, $t2`

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

NOT Operations

- Useful to invert bits in a word
  - Change 0 to 1, and 1 to 0
- MIPS has NOR 3-operand instruction
  - a 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

Conditional Operations

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

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: …
  ```
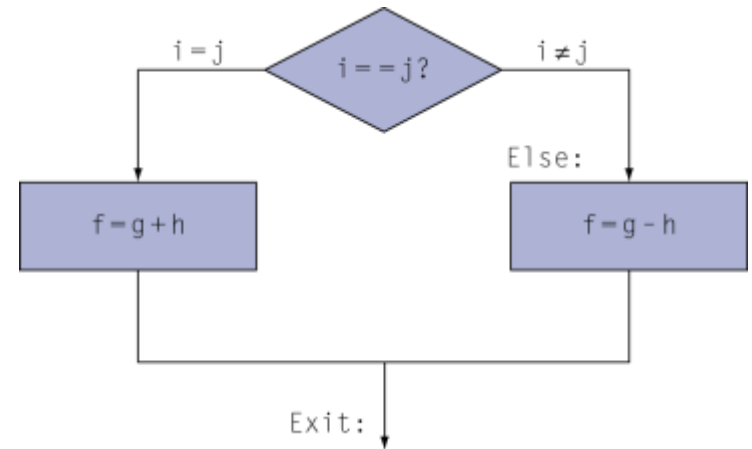
Assembler calculates addresses

Compiling Loop Statements

- C code:

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

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

Basic Blocks

- A basic block is a sequence of instructions with
  - No embedded branches (except at end)
  - No branch targets (except at beginning)
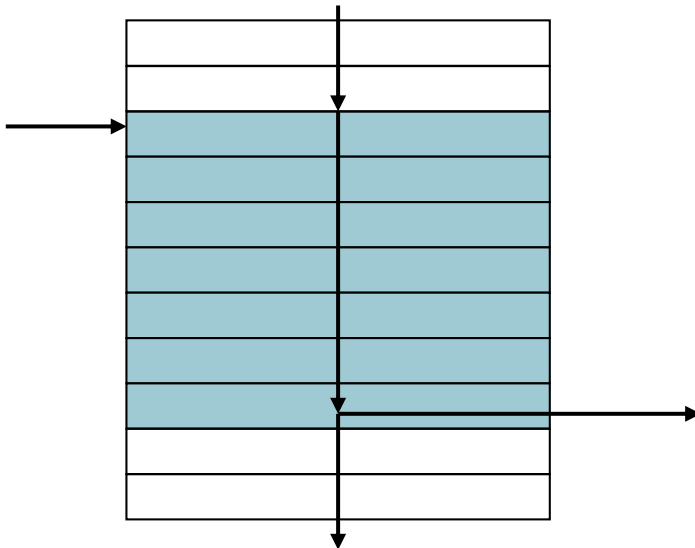
- A compiler identifies basic blocks for optimization
- An advanced processor can accelerate execution of basic blocks

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
```

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

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

MIPS J-format Instructions

op	Target address
6 bits	26 bits

- Fixed op code and instruction length in MIPS
  - There are only 26 bits target address
- The MIPS instructions are fixed length as well, each one is 32 bits
  - And the byte address is used, so when the 32-bit instructions address are 4 byte apart
  - So the 26 bits can be used as 28 bits in byte address space for instructions
  - 2 zeros are padded at the end for that

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)

Procedure Call Instructions

- Procedure call: jump and link

    `jal ProcedureLabel`

    - Address of following instruction put in $ra
    - Jumps to target address

- Procedure return: jump register

    `jr $ra`

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

Leaf Procedure Example

- C code:

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

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

Leaf Procedure Example

- MIPS code:

	Save $s0 on stack
	Procedure body
	Result
	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 f;
  else return n * fact(n - 1);
}
```

- Argument n in $a0
- Result in $v0

Non-Leaf Procedure Example

- MIPS code:

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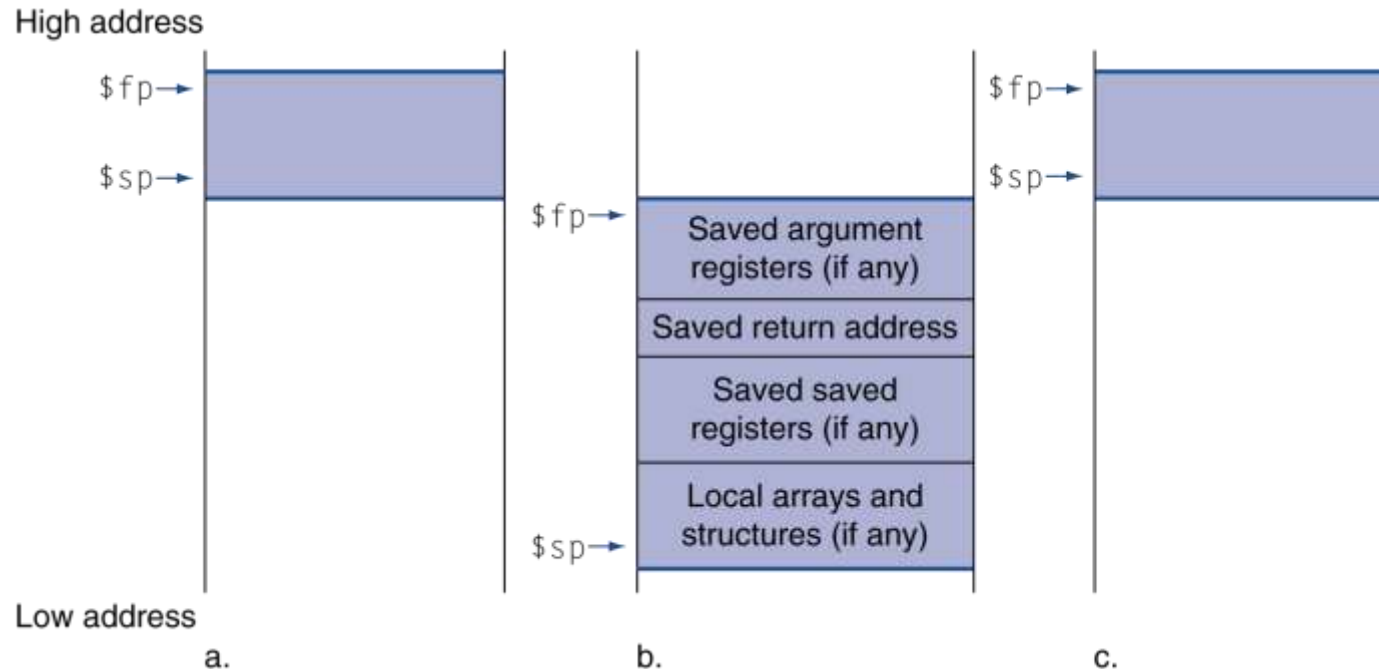
- Local data allocated by callee
  - e.g., C automatic variables
- Procedure frame (activation record)
  - Used by some compilers to manage stack storage

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

$sp → 7fff fffc_hex

Stack

↓

↑

Dynamic data

$gp → 1000 8000_hex

1000 0000_hex

Static data

Text

pc → 0040 0000_hex

0

Reserved
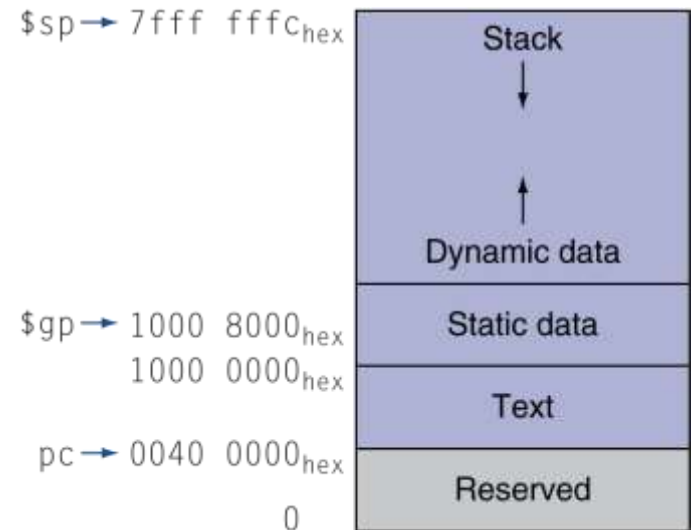
Character Data

- **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:

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

`lhi $s0, 61`

0000 0000 0111 1101	0000 0000 0000 0000

`ori $s0, $s0, 2304`

0000 0000 0111 1101	0000 1001 0000 0000

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 × 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

op	address
6 bits	26 bits
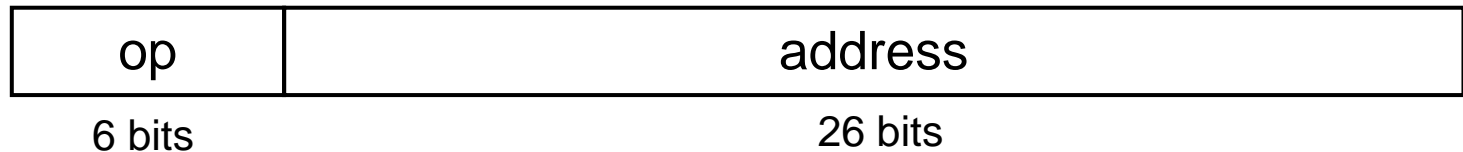
- (Pseudo)Direct jump addressing
  - Target address = $PC_{31\ldots28}$: (address × 4)

Target Addressing Example

- Loop code from 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	4	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:  …
```

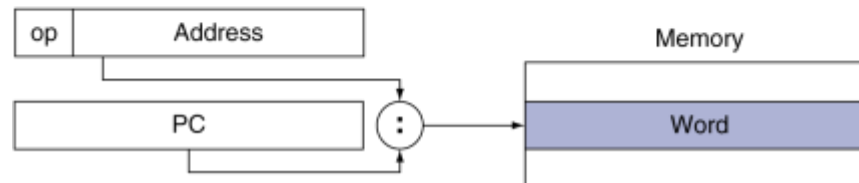# Addressing Mode Summary

1. Immediate addressing

op	rs	rt	Immediate

2. Register addressing

op	rs	rt	rd	. . .	funct

Registers

Register

3. Base addressing

op	rs	rt	Address

Register

+

Memory

Byte	Halfword	Word

4. PC-relative addressing

op	rs	rt	Address

PC

+

Memory

Word
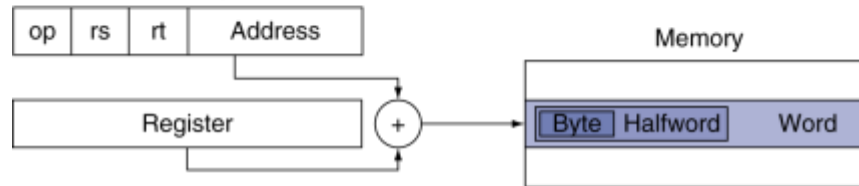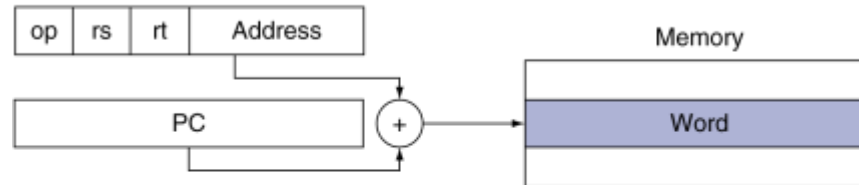
5. Pseudodirect addressing

op	Address

PC

:

Memory

Word

Synchronization

- Two processors sharing an area of memory
  - P1 writes, then P2 reads
  - Data race if P1 and P2 don't synchronize
    - Result depends of order of accesses

- Hardware support required
  - Atomic read/write memory operation
  - No other access to the location allowed between the read and write

- Could be a single instruction
  - E.g., atomic swap of register ↔ memory
  - Or an atomic pair of instructions

Synchronization in MIPS

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

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

Translation and Startup

C program
↓
Compiler
↓
Assembly language program

Many compilers produce
object modules directly

↓
Assembler
↓
Object: Machine language module   Object: Library routine (machine language)
↓
Linker

Static linking

↓
Executable: Machine language program
↓
Loader
↓
Memory
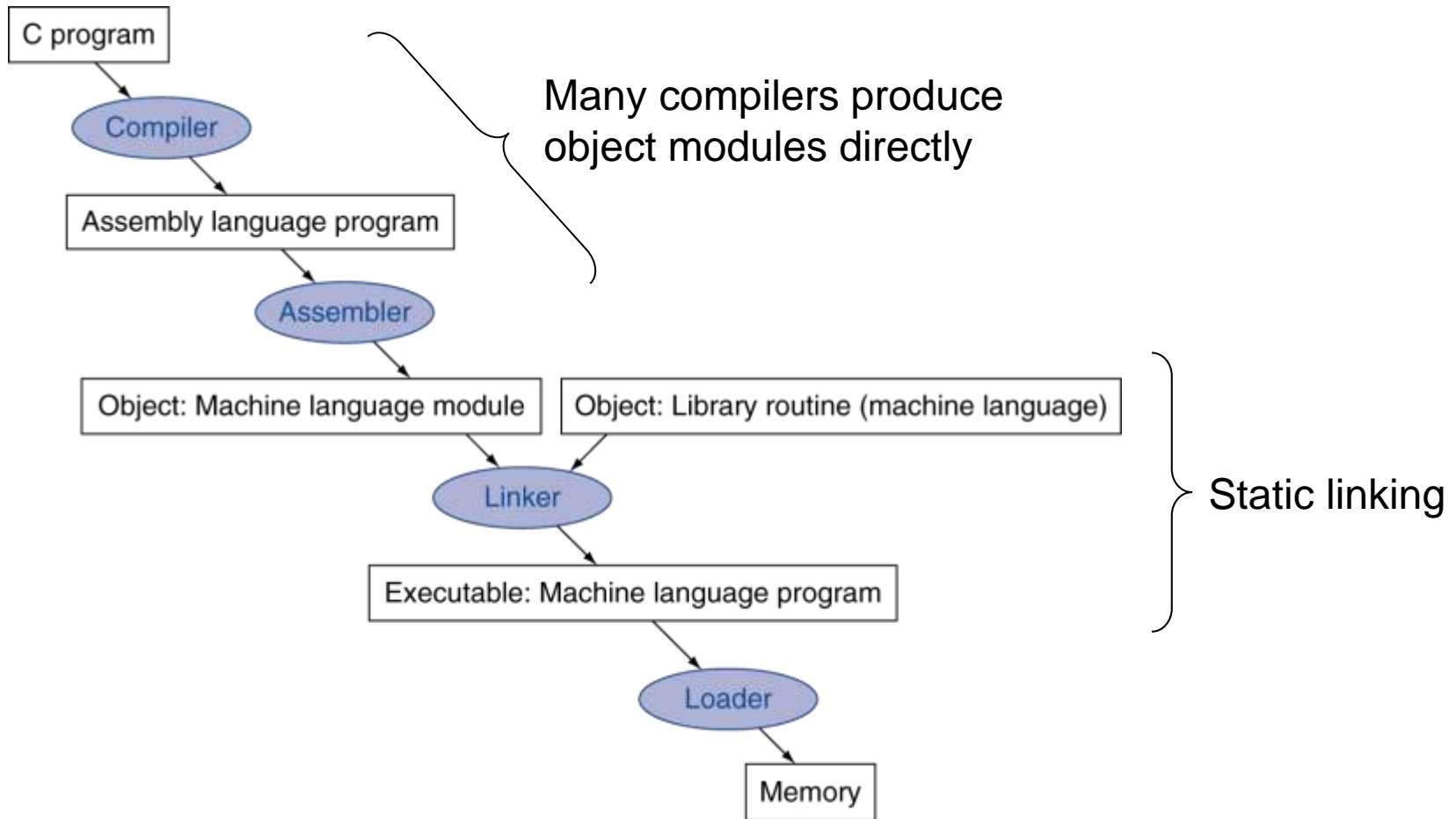
Assembler Pseudoinstructions

- 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

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
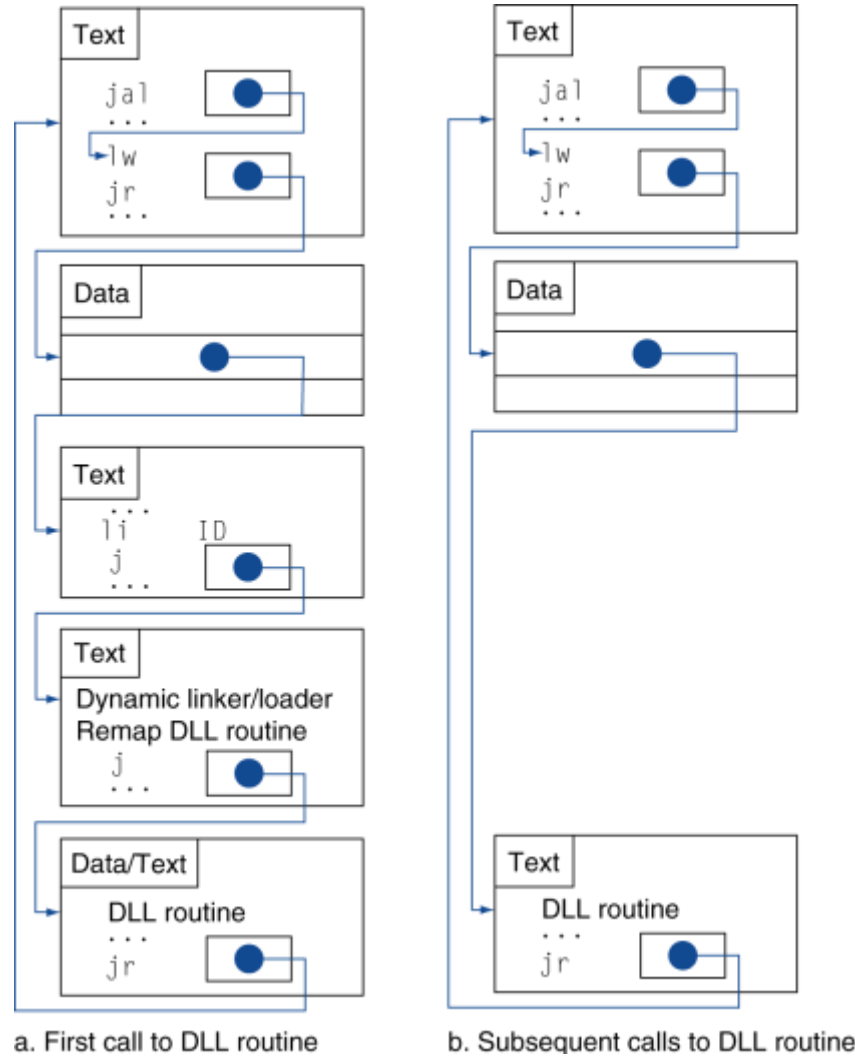
Starting Java Applications

Java program

Compiler

Simple portable instruction set for the JVM

Class files (Java bytecodes)

Java library routines (machine language)

Just In Time compiler

Java Virtual Machine

Compiles bytecodes of "hot" methods into native code for host machine

Compiled Java methods (machine language)

Interprets bytecodes
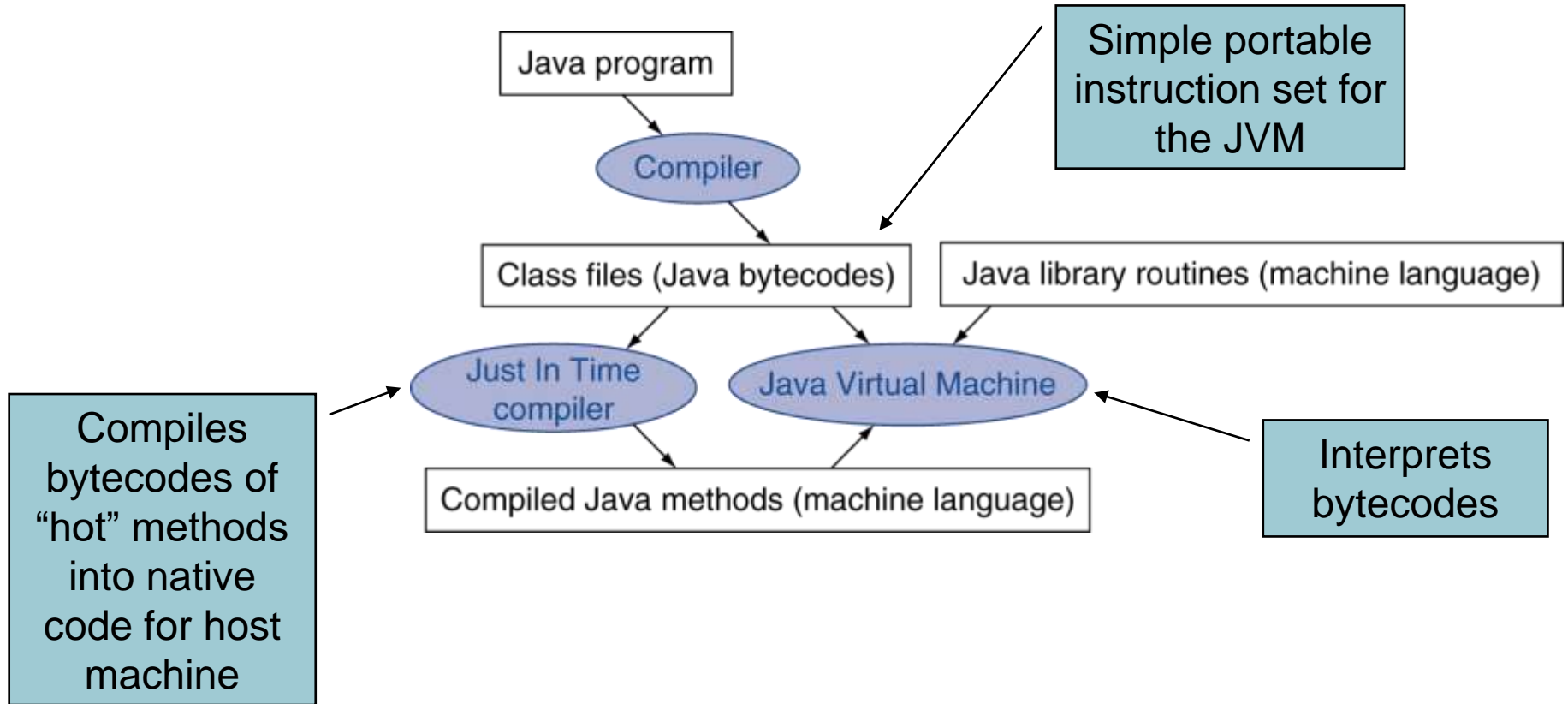
C Sort Example

- 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

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 params

Outer loop

Inner loop

Pass params & call

Inner loop

Outer loop
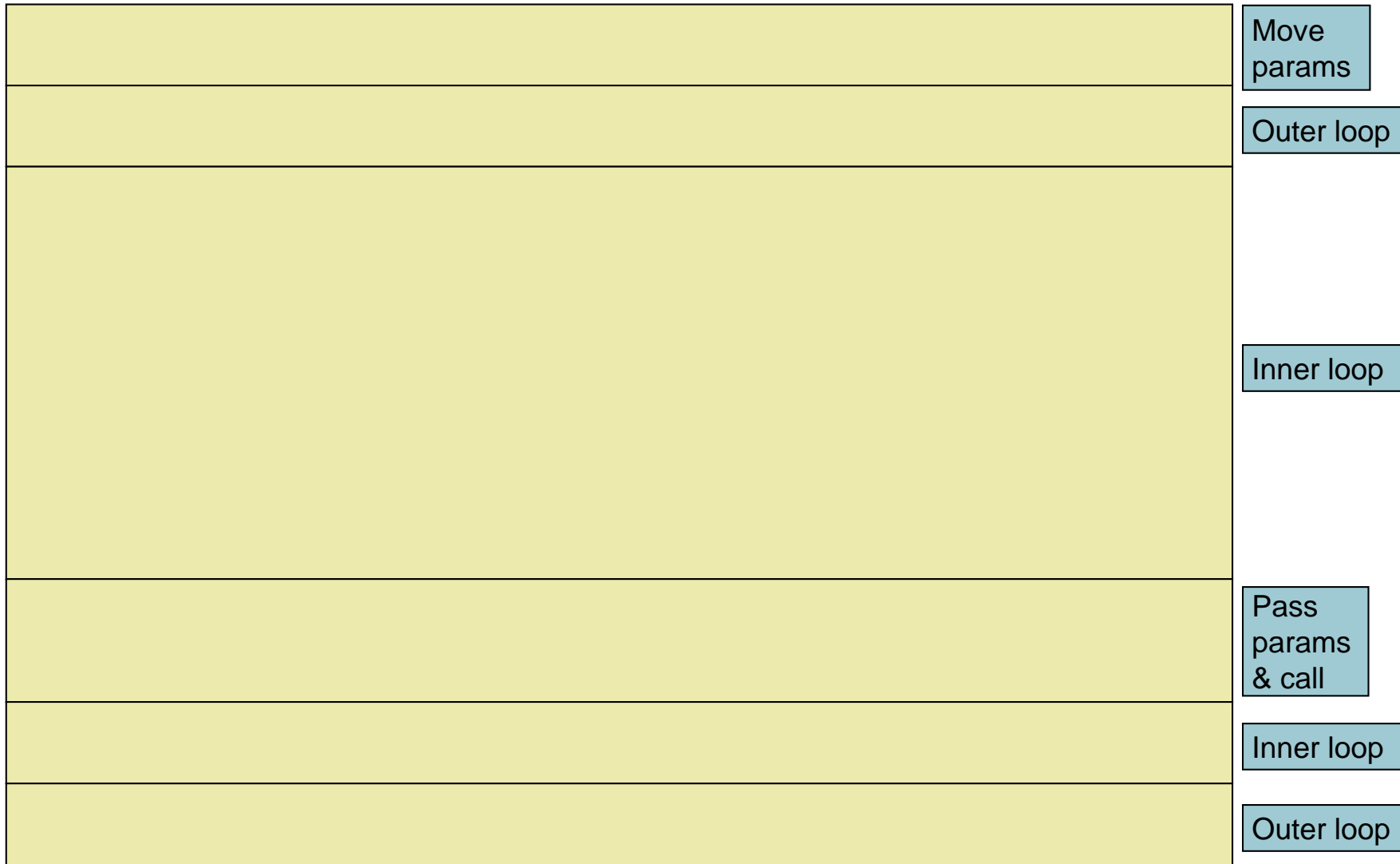
The Full Procedure

Effect of Compiler Optimization

Compiled with gcc for Pentium 4 under Linux
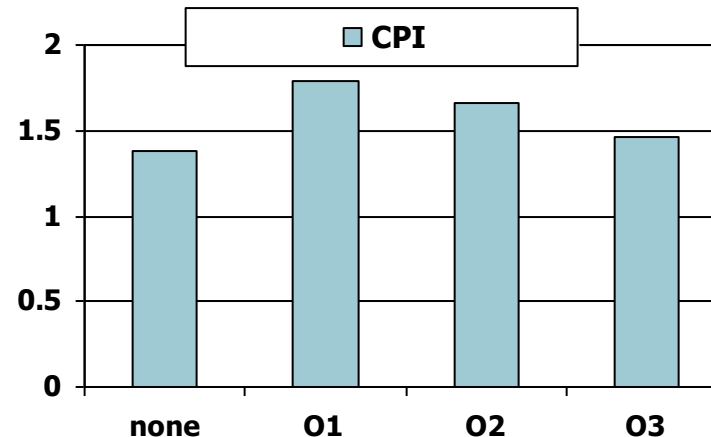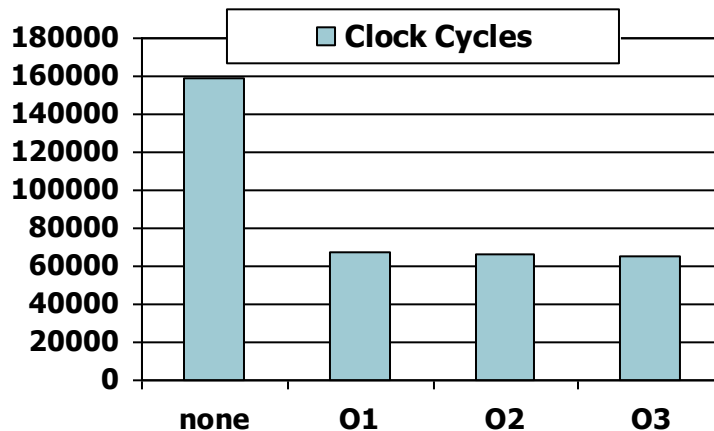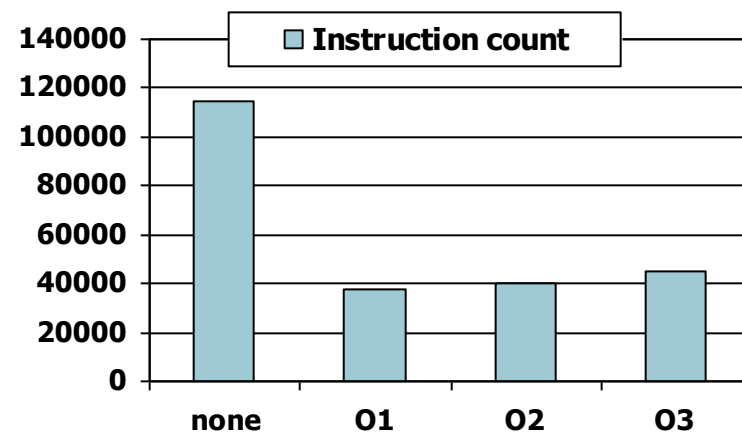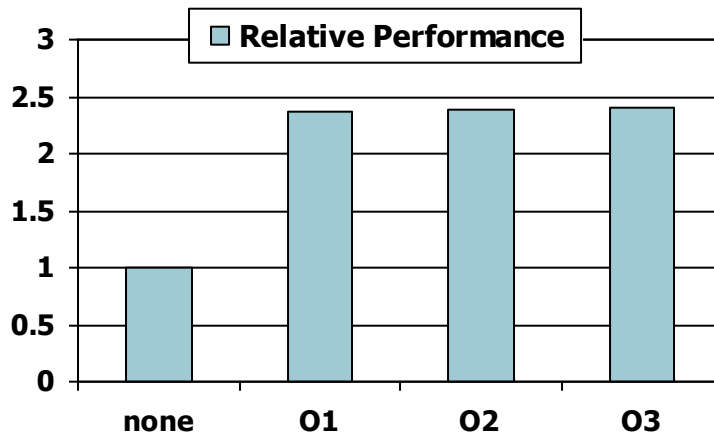
Effect of Language and Algorithm
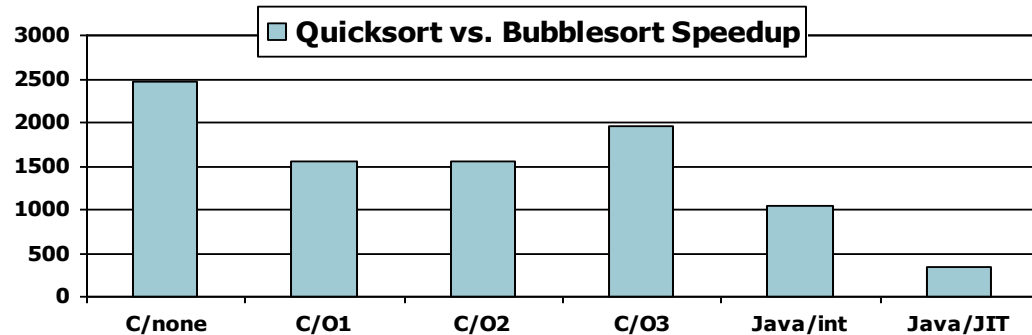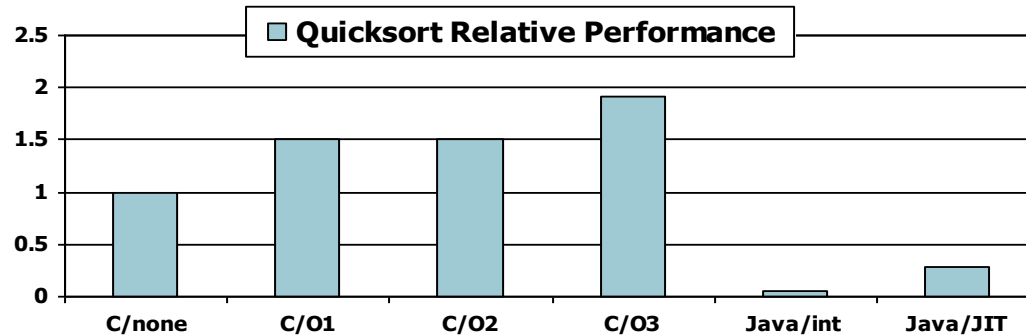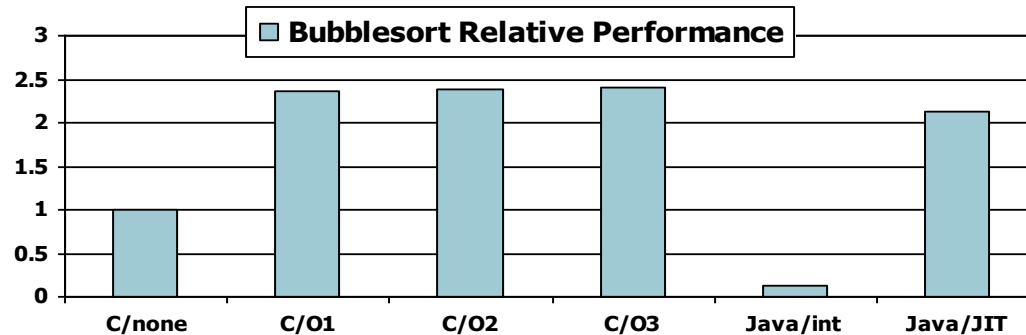
Lessons Learnt

- Instruction count and CPI are not good performance indicators in isolation

- Compiler optimizations are sensitive to the algorithm

- Java/JIT compiled code is significantly faster than JVM interpreted

  - Comparable to optimized C in some cases

- Nothing can fix a dumb algorithm!

Arrays vs. Pointers

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

Example: Clearing and Array

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

```
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
```

Comparison of Array vs. Ptr

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

ARM & MIPS Similarities

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

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

Compare and Branch in ARM

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

Instruction Encoding
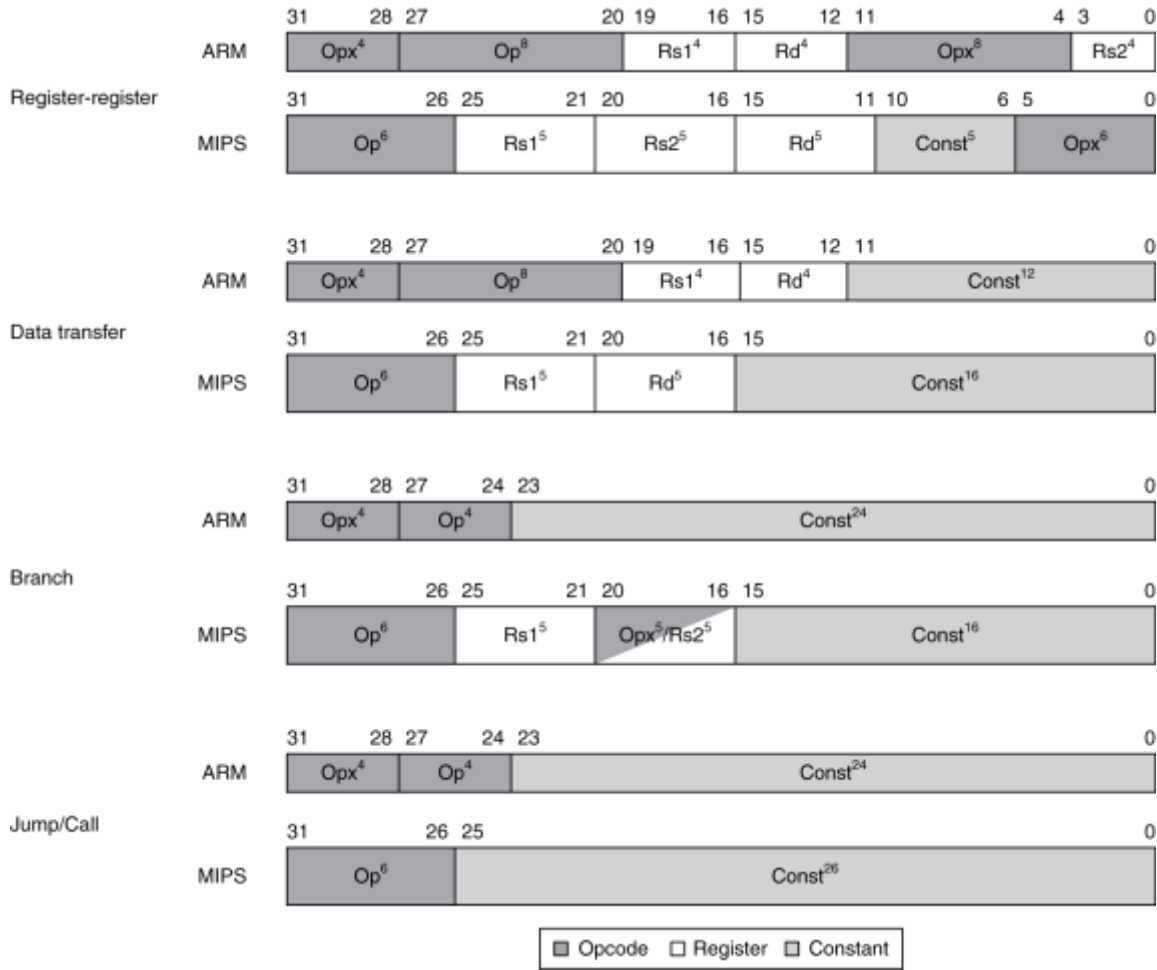
The Intel x86 ISA

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

The Intel x86 ISA

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

The Intel x86 ISA

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

Basic x86 Registers

Basic x86 Addressing Modes

- Two operands per instruction

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

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

x86 Instruction Encoding

a. JE EIP + displacement

4	4	8
JE	Condi-tion	Displacement

b. CALL

8	32
CALL	Offset

c. MOV   EBX, [EDI + 45]

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

d. PUSH ESI

5	3
PUSH	Reg

e. ADD EAX, #6765

4	3	1	32
ADD	Reg	w	Immediate

f. TEST EDX, #42

7	1	8	32
TEST	w	Postbyte	Immediate
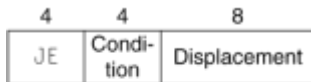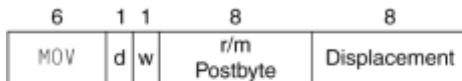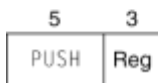
- Variable length encoding
  - Postfix bytes specify addressing mode
  - Prefix bytes modify operation
    - Operand length, repetition, locking, …

Implementing IA-32

- Complex instruction set makes implementation difficult
  - Hardware translates instructions to simpler microoperations
    - Simple instructions: 1–1
    - Complex instructions: 1–many
  - Microengine similar to RISC
  - Market share makes this economically viable
- Comparable performance to RISC
  - Compilers avoid complex instructions
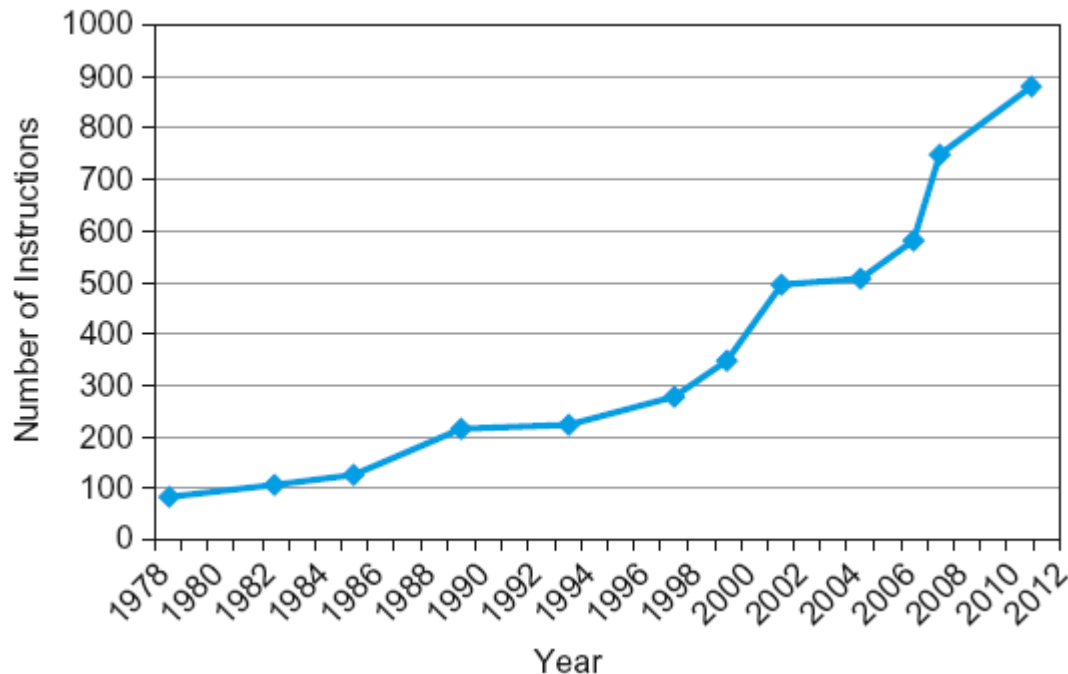
ARM v8 Instructions

- In moving to 64-bit, ARM did a complete overhaul

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

Fallacies

- Powerful instruction $\Rightarrow$ higher performance
  - Fewer instructions required
  - But complex instructions are hard to implement
    - May slow down all instructions, including simple ones
  - Compilers are good at making fast code from simple instructions

- Use assembly code for high performance
  - But modern compilers are better at dealing with modern processors
  - More lines of code $\Rightarrow$ more errors and less productivity

Fallacies

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

x86 instruction set

Pitfalls

- ## Sequential words are not at sequential addresses

  - ### Increment by 4, not by 1!

- ## Keeping a pointer to an automatic variable after procedure returns

  - ### e.g., passing pointer back via an argument
  - ### Pointer becomes invalid when stack popped

Concluding Remarks

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

Concluding Remarks

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

Instruction class	MIPS examples	SPEC2006 Int	SPEC2006 FP
Arithmetic	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%

Copyright Statement

The copyrighted materials posted to this course are intended for your personal use in this course and should not be shared with others or retained beyond the end of this course. To learn more about copyright visit http://www.copyright.gov/.