

**COMPUTER ORGANIZATION AND DESIGN**  
The Hardware/Software Interface

4

## 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

Chapter 2 — Instructions: Language of the Computer — 2

## The MIPS Instruction Set

- Used as the example throughout the book
- Stanford MIPS commercialized by MIPS Technologies ([www.mips.com](http://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 tear-out card, and Appendixes B and E

Chapter 2 — Instructions: Language of the Computer — 3

## 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 favours regularity*
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

Chapter 2 — Instructions: Language of the Computer — 4

## 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
```

Chapter 2 — Instructions: Language of the Computer — 5

## 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

Chapter 2 — Instructions: Language of the Computer — 6

## Register Operand Example

- C code:
  - $f = (g + h) - (i + j);$
  - $f, \dots, j$  in  $\$s0, \dots, \$s4$
- 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
```



## 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



Chapter 2 — Instructions: Language of the Computer — 13

### Unsigned Binary Integers

§ 2.4 Signed and Unsigned Numbers

- Given an n-bit number
 
$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$
- Range: 0 to  $+2^n - 1$
- Example
  - 0000 0000 0000 0000 0000 0000 1011<sub>2</sub>  
= 0 + ... + 1×2<sup>3</sup> + 0×2<sup>2</sup> + 1×2<sup>1</sup> + 1×2<sup>0</sup>  
= 0 + ... + 8 + 0 + 2 + 1 = 11<sub>10</sub>
- Using 32 bits
  - 0 to +4,294,967,295



Chapter 2 — Instructions: Language of the Computer — 14

### 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 1100<sub>2</sub>  
= -1×2<sup>31</sup> + 1×2<sup>30</sup> + ... + 1×2<sup>2</sup> + 0×2<sup>1</sup> + 0×2<sup>0</sup>  
= -2,147,483,648 + 2,147,483,644 = -4<sub>10</sub>
- Using 32 bits
  - -2,147,483,648 to +2,147,483,647



Chapter 2 — Instructions: Language of the Computer — 15

### 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



Chapter 2 — Instructions: Language of the Computer — 16

### Signed Negation

- Complement and add 1
  - Complement means 1 → 0, 0 → 1
$$x + \bar{x} = 1111 \dots 1111_2 = -1$$

$$\bar{x} + 1 = -x$$
- Example: negate +2
  - +2 = 0000 0000 ... 0010<sub>2</sub>
  - -2 = 1111 1111 ... 1101<sub>2</sub> + 1  
= 1111 1111 ... 1110<sub>2</sub>



Chapter 2 — Instructions: Language of the Computer — 17

### Sign Extension

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



Chapter 2 — Instructions: Language of the Computer — 18

## 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

Chapter 2 — Instructions: Language of the Computer — 19

## MIPS R-format Instructions

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

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

Chapter 2 — Instructions: Language of the Computer — 20

## 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
--------	-------	-------	-------	-------	--------

$0000001000110010010000000100000_2 = 02324020_{16}$

Chapter 2 — Instructions: Language of the Computer — 21

## 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

Chapter 2 — Instructions: Language of the Computer — 22

## MIPS I-format Instructions

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

- Immediate arithmetic and load/store instructions
  - rt: destination or source register number
  - Constant:  $-2^{15}$  to  $+2^{15} - 1$
  - Address: offset added to base address in rs
- Design Principle 4: Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

Chapter 2 — Instructions: Language of the Computer — 23

## 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

Chapter 2 — Instructions: Language of the Computer — 24

## 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

Chapter 2 — Instructions: Language of the Computer — 25

## 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)

Chapter 2 — Instructions: Language of the Computer — 26

## 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

Chapter 2 — Instructions: Language of the Computer — 27

## OR Operations

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

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

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

Chapter 2 — Instructions: Language of the Computer — 28

## NOT Operations

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

nor \$t0, \$t1, \$zero

Register 0: always read as zero

\$t1	0000 0000 0000 0000 0011 1100 0000 0000
\$t0	1111 1111 1111 1111 1100 0011 1111 1111

Chapter 2 — Instructions: Language of the Computer — 29

## 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

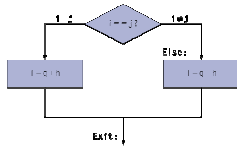
Chapter 2 — Instructions: Language of the Computer — 30

### 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;
```

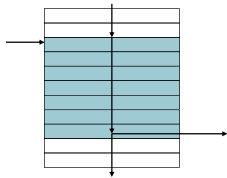
  - i in \$s3, k in \$s5, address of save in \$s6
- Compiled MIPS code:
 

```
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 `b<, 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 ⇒ $t0 = 1`
  - `sltu $t0, $s0, $s1 # unsigned`
    - `+4,294,967,295 > +1 ⇒ $t0 = 0`



## Procedure Calling

- Steps required
  - Place parameters in registers
  - Transfer control to procedure
  - Acquire storage for procedure
  - Perform procedure's operations
  - Place result in register for caller
  - Return to place of call

Chapter 2 — Instructions: Language of the Computer — 37

## 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)

Chapter 2 — Instructions: Language of the Computer — 38

## Procedure Call Instructions

- Procedure call: jump and link
 

```
jal ProcedureLabel
```

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

```
jr $ra
```

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

Chapter 2 — Instructions: Language of the Computer — 39

## Leaf Procedure Example

- C code:
 

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

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

Chapter 2 — Instructions: Language of the Computer — 40

## 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

Chapter 2 — Instructions: Language of the Computer — 41

## 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


Chapter 2 — Instructions: Language of the Computer — 42

### 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




Chapter 2 — Instructions: Language of the Computer — 43

### Non-Leaf Procedure Example

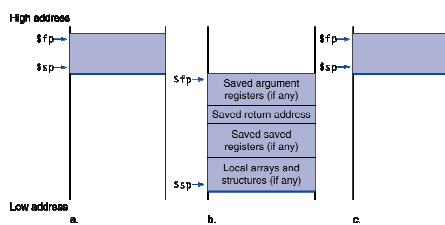
- MIPS code:
 

```
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
```




Chapter 2 — Instructions: Language of the Computer — 44

### Local Data on the Stack



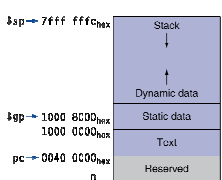

- Local data allocated by callee
  - e.g., C automatic variables
- Procedure frame (activation record)
  - Used by some compilers to manage stack storage



Chapter 2 — Instructions: Language of the Computer — 45

### 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

Chapter 2 — Instructions: Language of the Computer — 46

### 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




Chapter 2 — Instructions: Language of the Computer — 47

### 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)
lbu rt, offset(rs)  lhu rt, offset(rs)
sb rt, offset(rs)   sh rt, offset(rs)
```

- Sign extend to 32 bits in rt
- Zero extend to 32 bits in rt
- Store just rightmost byte/halfword



Chapter 2 — Instructions: Language of the Computer — 48



### 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

Chapter 2 — Instructions: Language of the Computer — 49

### 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

Chapter 2 — Instructions: Language of the Computer — 50

### 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
---------------------	---------------------

Chapter 2 — Instructions: Language of the Computer — 51

### 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

Chapter 2 — Instructions: Language of the Computer — 52

### 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<sub>31...28</sub> : (address × 4)

Chapter 2 — Instructions: Language of the Computer — 53

### Target Addressing Example

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

Loop: sll	\$t1, \$s3, 2	80000	0	0	19	9	4	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							

Chapter 2 — Instructions: Language of the Computer — 54

### 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: ...
            
```

Chapter 2 — Instructions: Language of the Computer — 55

### Addressing Mode Summary

Chapter 2 — Instructions: Language of the Computer — 56

### 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

Chapter 2 — Instructions: Language of the Computer — 57

### 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
            
```

Chapter 2 — Instructions: Language of the Computer — 58

### Translation and Startup

Chapter 2 — Instructions: Language of the Computer — 59

### Assembler Pseudoinstructions

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

```

                move $t0, $t1 → add $t0, $zero, $t1
                blt $t0, $t1, L → slt $at, $t0, $t1
                               bne $at, $zero, L
            
```

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

Chapter 2 — Instructions: Language of the Computer — 60

## Producing an Object Module

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



Chapter 2 — Instructions: Language of the Computer — 61

## Linking Object Modules

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



Chapter 2 — Instructions: Language of the Computer — 62

## 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



Chapter 2 — Instructions: Language of the Computer — 63

## 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



Chapter 2 — Instructions: Language of the Computer — 64

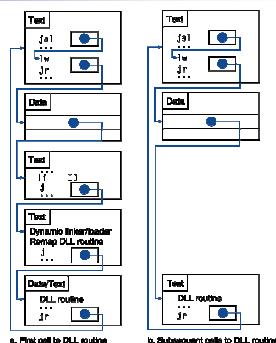
## Lazy Linkage

Indirection table

Stub: Loads routine ID, Jump to linker/loader

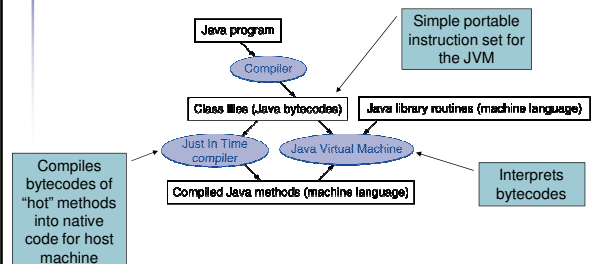
Linker/loader code

Dynamically mapped code



Chapter 2 — Instructions: Language of the Computer — 65

## Starting Java Applications



Chapter 2 — Instructions: Language of the Computer — 66

### C Sort Example

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

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

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

§2.19 A C Sort Example to Put It All Together



### 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, k in \$a1, i in \$s0, j in \$s1



### The Procedure Body

move \$s2, \$a0 # save \$a0 into \$s2	Move params
move \$s3, \$a1 # save \$a1 into \$s3	
move \$s0, \$zero # i = 0	Outer loop
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)	Inner loop
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)	Pass params & call
move \$a1, \$s1 # 2nd param of swap is j	
jal swap # call swap procedure	
addi \$s1, \$s1, -1 # j = j - 1	Inner loop
j for2tst # jump to test of inner loop	
exit2: addi \$s0, \$s0, 1 # i = i + 1	Outer loop
j for1tst # jump to test of outer loop	

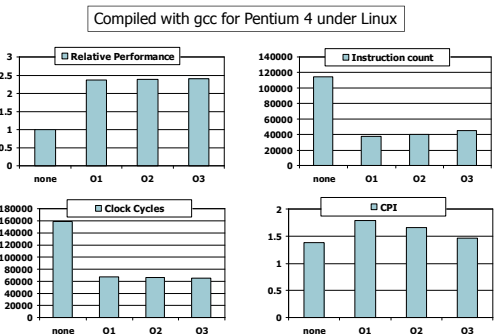


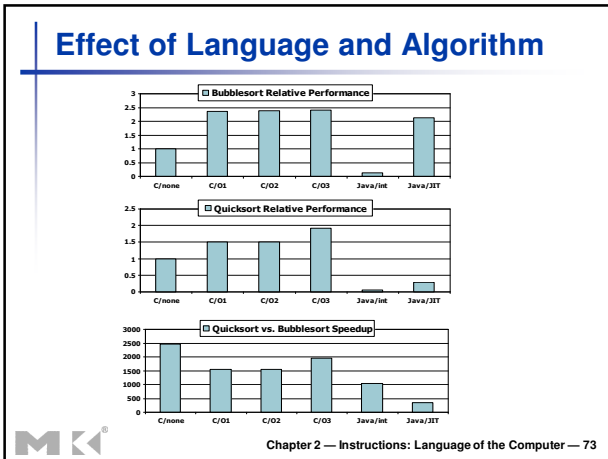
### The Full Procedure

```
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
```



### Effect of Compiler Optimization





### 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!

Chapter 2 — Instructions: Language of the Computer — 74

### 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

Chapter 2 — Instructions: Language of the Computer — 75

### 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;
}
    
```

Chapter 2 — Instructions: Language of the Computer — 76

### 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

Chapter 2 — Instructions: Language of the Computer — 77


### ARM & MIPS Similarities

	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

Chapter 2 — Instructions: Language of the Computer — 78

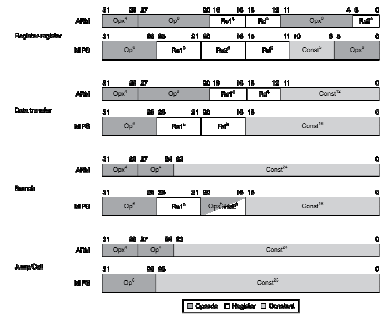

## Compare and Branch in ARM

- 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



Chapter 2 — Instructions: Language of the Computer — 79


## Instruction Encoding

Chapter 2 — Instructions: Language of the Computer — 80

## 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



Chapter 2 — Instructions: Language of the Computer — 81

## The Intel x86 ISA


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



Chapter 2 — Instructions: Language of the Computer — 82

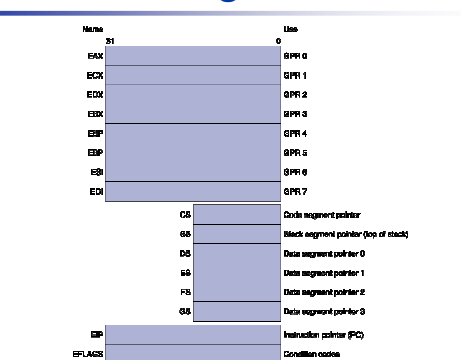

## The Intel x86 ISA

- 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



Chapter 2 — Instructions: Language of the Computer — 83

## Basic x86 Registers

Chapter 2 — Instructions: Language of the Computer — 84

## Basic x86 Addressing Modes

- Two operands per instruction

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

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

Chapter 2 — Instructions: Language of the Computer — 85

## x86 Instruction Encoding

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

Chapter 2 — Instructions: Language of the Computer — 86

## Implementing IA-32

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

Chapter 2 — Instructions: Language of the Computer — 87

## 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

Chapter 2 — Instructions: Language of the Computer — 88

## Fallacies

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

Chapter 2 — Instructions: Language of the Computer — 89

## Pitfalls

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

Chapter 2 — Instructions: Language of the Computer — 90

## 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%

