

Stored Program Computers

The BIG Picture

The diagram illustrates the architecture of a stored program computer. On the left, a box labeled 'Processor' is connected to a larger box labeled 'Memory'. The 'Memory' box is divided into several sections, each representing a different type of data or program stored in memory. These sections are: 'Accounting program (machine code)', 'Editor program (machine code)', 'C compiler (machine code)', 'Payroll data', 'Book text', and 'Source code in C for editor program'. The first three sections are labeled as 'machine code', indicating they are instructions that the processor can execute. The other three sections represent data or source code that can be processed or compiled.

- 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 — 2

MIPS Machine Language

- The processor doesn't *understand* things like this:

`add $s0,$s0,$s2`

- It does *understand* things like this:

`10000101001010001100010000000101`

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 the MIPS Reference Data tear-out card, as well as Appendix A

Chapter 2 — Instructions: Language of the Computer — 5

Arithmetic Operations

- Add and subtract, three operands
 - Two sources and one destination
 - add a, b, c # a = b + c
 - sub a, b, c # a = b - c
- All arithmetic operations use this form
- *Design Principle 1:*
 - **Simplicity favors regularity**
 - Regularity makes implementation simpler
 - Simplicity enables higher performance at lower cost

\$2.2 Operations of the Computer Hardware

Chapter 2 — Instructions: Language of the Computer — 6

Instruction Formats

- How do we encode instructions?
- Break up the 32-bit word into *fields*.
- Each field is an encoding of part of the instruction, including:
 - what registers to use
 - what operation should be performed
 - constants
 - etc.
- (We'll return to this in a few minutes)

Chapter 2 — Instructions: Language of the Computer — 7

Simplicity Favors Regularity

- Let's translate this example *C* code...
 $a = b + c;$
 $d = a - e;$
- ...into MIPS assembly language code:
`add a, b, c # a = b + c`
`sub d, a, e # d = a - e`
- We have therefore achieved: $d = b + c - e;$
- Typical MIPS instruction only has two source operands and places the results in a destination operand
 - There are a few exceptions to this rule

CSCI-2500 Fall 2010, Ch2 P&H

Arithmetic Example

- C code:

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

- Corresponding 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 — 9

Arithmetic Example (HW 2)

- C code (no parentheses!):

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

- Corresponding MIPS code:

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

(really this is MIPS pseudo-code...)

Chapter 2 — Instructions: Language of the Computer — 10

Register Operands

§2.3 Operands of the Computer Hardware

- Arithmetic instructions make use of specific register operands
- MIPS has a 32×32-bit register file
 - Used for frequently accessed data/variables
 - Registers are numbered 0 to 31
 - Each 32-bit register contains a “word”
- Assembler register names
 - \$t0, \$t1, ..., \$t9 for temporary values
 - \$s0, \$s1, ..., \$s7 for “saved” variables
- **Design Principle 2: Smaller is faster**
 - Registers provide a very small and very fast memory (as compared to main memory)

Chapter 2 — Instructions: Language of the Computer — 11

32/64-bit MIPS

- Most 32-bit systems fall into ILP32
 - Integer, Long, Pointer all 32 bits long (i.e., 4 bytes)
- Most 64-bit systems fall into LP64
 - Long, Pointer both 64 bits (i.e., 8 byte)
- In both ILP32 and LP64:
 - char is 8 bits (i.e., 1 byte)
 - short is 16 bits (i.e., 2 bytes)
 - long long is 64 bits (i.e., 8 bytes)

Chapter 2 — Instructions: Language of the Computer — 12

Arithmetic Example (HW 2)

- C code (no parentheses!):

$f = g + h - i - j;$

- Corresponding actual MIPS code (not pseudo-code):

```
add $t0,$s1,$s2 # temp t0 = g + h
sub $t1,$t0,$s3 # temp t1 = t0 - i
sub $s0,$t1,$s4 # f = t1 - j
```

```
f = g + h - i - j ;
s0 = s1 + s2 - s3 - s4;
```

Chapter 2 — Instructions: Language of the Computer — 13

Memory Operands

- Main memory used for composite data
 - Arrays, structures, dynamic data
- To apply arithmetic operations
 - Load values from memory into registers
 - Store results from registers (back) into 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 address of a word

Chapter 2 — Instructions: Language of the Computer — 14

Memory as Words

20	01001000	11010100	01111001	11010001
16	11010111	01011010	10000100	00001000
12	01001010	11001010	01000111	01000000
8	00000000	00000000	00000000	00000000
4	00000000	00000000	00000000	00000100
0	01001000	11010100	01111001	11010001

Address

Data

Memory Operand Example 1

- C code:
g = h + A[3];
 - g in \$s1, h in \$s2, base address of A in \$s3
- Corresponding MIPS code:
 - Index 3 requires offset of 12
 - 4 bytes per word

```
lw $t0, 12($s3)    # load word
add $s1, $s2, $t0
```



Memory Operand Example 2

- C code:

`A[5] = h + A[3];`

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

- Corresponding MIPS code:

- Index 3 requires offset of 12

- Index 5 requires offset of 20

```
lw  $t0, 12($s3)    # load word
```

```
add $t0, $s2, $t0
```

```
sw  $t0, 20($s3)    # store word
```

Chapter 2 — Instructions: Language of the Computer — 17

Registers vs. Memory

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

Chapter 2 — Instructions: Language of the Computer — 18

Bytes vs. Words

- MIPS registers are each 32 bits wide (i.e., 1 word).
- Memory is organized into 8-bit bytes
- In the MIPS architecture, *words* must begin at addresses that are a multiple of 4
 - inherent alignment restriction
 - on 64-bit architectures, note that words are 8-byte aligned

Chapter 2 — Instructions: Language of the Computer — 19

Immediate Operands

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

Chapter 2 — Instructions: Language of the Computer — 20

The Constant Zero

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

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

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

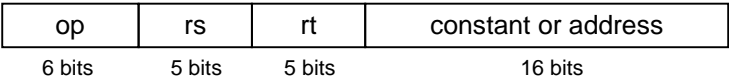
op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

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

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

$00000010001100100100000000100000_2 = 02324020_{16}$

MIPS I-format Instructions



- 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 to be uniform
 - Keep formats as similar as possible

Load Instructions

- *Load* means to move from memory into a register.
- The load instruction needs two things:
 - which register??
 - which memory location (the address)??

lw: Load Word

- The *load word* instruction needs to be told an address that is a multiple of 4
- In MIPS, the way to specify an address is as the sum of:
 - a constant
 - name of a register that holds an address.
 - here we have only 2 register operands and the 3rd is the constant (a compromise!!)

lw destreg, const(addrreg)

“Load Word”

A number

Name of register
to put value in

Name of register to get
base address from

destreg = contents of (*addrreg* + *const*)

Example: `lw $s0, 4($s3)`

- If `$s3` has the value 100, this will copy the word at memory location 104 to the register `$s0`

`$s0 <- Memory[104]`

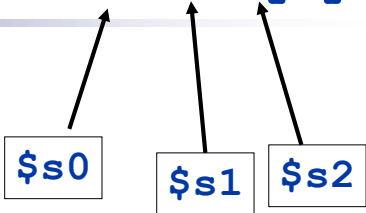
Chapter 2 — Instructions: Language of the Computer — 29

Why the odd address mode?

- We need to supply a *base* (the contents of the register) and an *offset* (the constant).
- Why not just specify the address as a constant?
 - some instruction sets include this type of addressing.
- It simplifies the instruction set and helps support arrays and structures.

Chapter 2 — Instructions: Language of the Computer — 30

Integer Array Ex: $a=b+c[8]$



```
lw $t0, 8($s2)      # $t0 = c[8]
add $s0, $s1, $t0    # $s0 = $s1 + $t0
```

Is this right? NO!

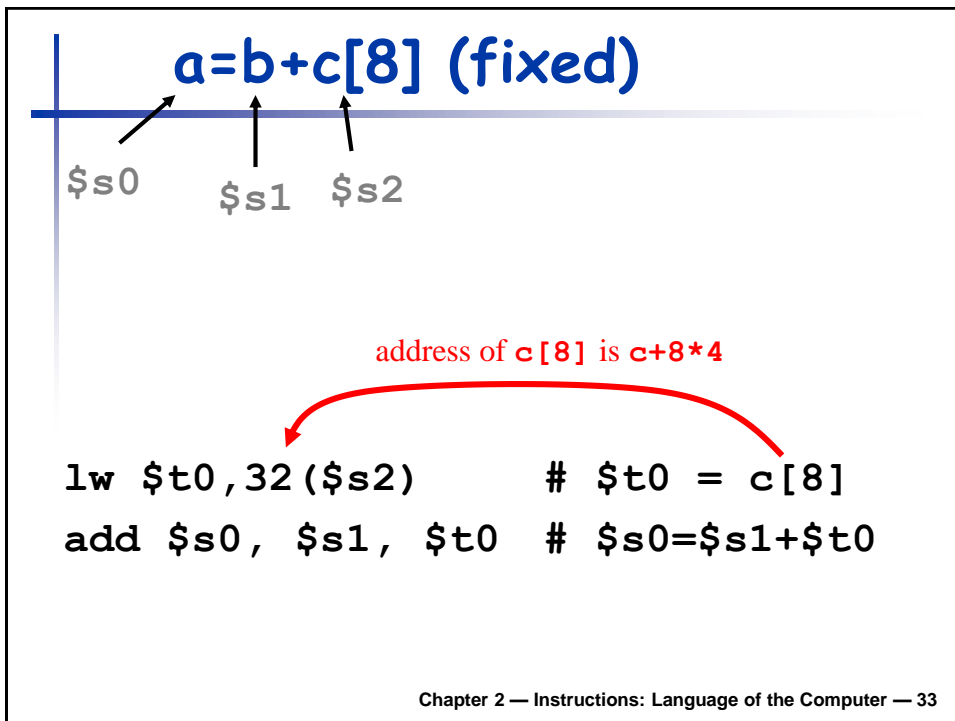
Chapter 2 — Instructions: Language of the Computer — 31

Words vs. Bytes

- Each byte in memory has a unique address.
- If the integer array C starts at address 100:
 - $C[0]$ starts at address 100
 - $C[1]$ starts at address 104
 - $C[2]$ starts at address 108

$C[i]$ starts at address $100 + i * 4$

Chapter 2 — Instructions: Language of the Computer — 32



Moving from Reg -> Memory

- Store means to move from a register to memory.
- The store instruction looks like the load instruction - it needs two things:
 - which register
 - which memory location (the address).

sw srcreg, const(addrreg)

“Store Word”

A number

Name of register to get value from

Name of register to get base address from

Actual address = (addrreg + const)

Chapter 2 — Instructions: Language of the Computer — 35

Example: sw \$s0, 4(\$s3)

If \$s3 has the value 100, this will copy the word in register \$s0 to memory location 104.

Memory[104] <- \$s0

Chapter 2 — Instructions: Language of the Computer — 36

Example: C to MIPS

- Write the MIPS instructions that would correspond to the following C code:

```
c[3]=a+c[2];
```

- assume that `a` is `$s0` and that `c` is an array of 32 bit integers whose starting address is in `$s1`

c[3] = a+c[2]

```
lw    $t0, 8($s1)      # $t0 = c[2]
add   $t0, $t0, $s0     # $t0=$t0+$s0
sw    $t0, 12($s1)     # c[3] = $t0
```

Variable Array Index: $a=b+c[i]$

- Now the index to the array is a variable.
- We have to remember that the address of $c[i]$ is the base address + $4*i$
- We haven't done multiplication yet, but we can still do this example.

$a=b+c[i]$

<code>add \$t0,\$s3,\$s3</code>	<code># \$t0=i+i</code>
<code>add \$t0,\$t0,\$t0</code>	<code># \$t0=i+i+i+i</code>
<code>add \$t0,\$t0,\$s1</code>	<code># \$t0=c+i*4</code>
<code>lw \$t1,0(\$t0)</code>	<code># \$t1=c[i]</code>
<code>add \$s0,\$s1,\$t1</code>	<code># \$s0=b+c[i]</code>

§2.4 Signed and Unsigned Numbers

Unsigned Binary Integers

- Given an n-bit number
$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_12^1 + x_02^0$$
- Range: 0 to $+2^n - 1$
- Example
 - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
 $= 0 + \dots + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$
 $= 0 + \dots + 8 + 0 + 2 + 1 = 11_{10}$
- Using 32 bits
 - 0 to +4,294,967,295

Chapter 2 — Instructions: Language of the Computer — 41

2s-Complement Signed Integers

- Given an n-bit number
$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_12^1 + x_02^0$$
- Range: -2^{n-1} to $+2^{n-1} - 1$
- Example
 - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
 $= -1 \times 2^{31} + 1 \times 2^{30} + \dots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$
 $= -2,147,483,648 + 2,147,483,644 = -4_{10}$
- Using 32 bits
 - $-2,147,483,648$ to $+2,147,483,647$

Chapter 2 — Instructions: Language of the Computer — 42

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 $1 \rightarrow 0, 0 \rightarrow 1$

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

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

Pop Quiz

- What is -31 in 2s complement? (1 byte of data)

C

- A: 00011111
- B: 11100000
- C: 11100001
- D: 10010000

Chapter 2 — Instructions: Language of the Computer — 45

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
 - 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 — 46

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

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
 - srl 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

Pop Quiz

- How do we get the following mask value in C?

0000 0000 0000 0000 0011 1100 0000 0000
take some value and shift it over 10 digits

- A: `0xC << 10`
 - B: `0xF << 10`
 - C: `1111 << 10`
 - D: `0xFF << 10`
- B

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
 - $a \text{ NOR } b == \text{NOT} (a \text{ OR } b)$

`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

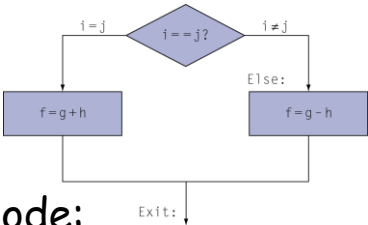
■ C code:

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

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

■ Corresponding 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
```

■ Corresponding 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:   ...
```