

Instruction Sets - Part 1

Ref: Chapter 3

CompOrg Fall 2000 - Instruction Sets

1

Stored Program Computer

- Recall that computers read instructions from memory (memory is a big array of bits).
- Each instruction is represented by a *bunch* of bits.
- We can think of the program as input to the processor – each instruction is fed in to the (very complex) network of gates that makes up the processor.

CompOrg Fall 2000 - Instruction Sets

2

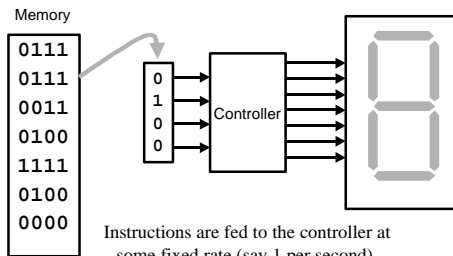
An Example Program

- Consider the controller you must design for homework (7-segment display controller).
- We could put a sequence of inputs together and feed them to your controller one at a time.
- The result would show up on the display (it would change according to the sequence of 4-bit inputs)

CompOrg Fall 2000 - Instruction Sets

3

7-Segment Controller In Action



CompOrg Fall 2000 - Instruction Sets

4

Our *Instruction Set*

- For our 7-segment controller computer, the instruction set is simple:
 - each instruction can set the display.
 - There are 16 possible instructions (but only 11 possible *actions*).
- For a real computer, we need more complex instructions!

CompOrg Fall 2000 - Instruction Sets

5

Processor Instruction Sets

- In general, a computer needs a few different kinds of instructions:
 - mathematical and logical operations
 - data movement (access memory)
 - jumping to new places in memory
 - if the right conditions hold.
 - I/O (sometimes treated as data movement)

CompOrg Fall 2000 - Instruction Sets

6

Instruction Set

- An Instruction Set provides a *functional* description of a processor.
- We could create a 7-segment controller that displays different patterns than we use for HW1.
 - this would be a different instruction set!
 - We need different instructions to generate the same pattern of displays.

CompOrg Fall 2000 - Instruction Sets

7

Instruction Set Architecture

- Most instruction sets are very similar.
- There have been a few different ways to define the location of operands.
 - operands are the data that are operated on, for example the numbers added together in an addition instruction.
- There is one primary architecture in use now, but it's worth looking at the others.

CompOrg Fall 2000 - Instruction Sets

8

Stack Architecture

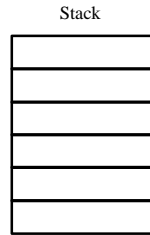
- In a stack-based instruction set the processor supports the notion of a *stack*:
 - A stack is a *last-in-first-out* list.
 - putting something on the stack is called a *push*
 - getting something off the stack is a *pop*
- In a stack-based instruction set all operands are on *the stack*.

CompOrg Fall 2000 - Instruction Sets

9

Stack Example: $A=B+C$

Program
push B
push C
add
pop A



CompOrg Fall 2000 - Instruction Sets

10

Stack Architecture: Implicit Operands

- In the Add instruction the operands are not explicitly defined in the instruction.
 - There are no bits in the machine code that are used to encode the location of the operands.
- The location where the result should be placed is also implicit (the top of the stack).

CompOrg Fall 2000 - Instruction Sets

11

Stack Architectures: Pros & Cons

- Small instructions (don't need many bits to specify the operation).
- Few options (compiler writer has it easy).
- Lots of memory accesses required
 - everything that is not on the stack is in memory.

CompOrg Fall 2000 - Instruction Sets

12

Notes on Stack Architectures

- The stack instruction set architecture has not been used for many years...
- People are talking about a hardware implementation of the Java *virtual machine* that is stack based!

CompOrg Fall 2000 - Instruction Sets

13

Accumulator Instruction Set Architecture

- Many early processors were based on a different way to support *implicit* operands.
- A single *word* of internal memory called the *accumulator* is always one of the operands.
- The result of an operation is always stored in the *accumulator*.

CompOrg Fall 2000 - Instruction Sets

14

Accumulator Example: $A=B+C$

Load	B	# Acc = B
Add	C	# Acc = Acc + C
Store	A	# A = Acc

CompOrg Fall 2000 - Instruction Sets

15

Accumulator Pros and Cons

- Easier to implement than stack.
- Small instructions (one implicit operand)
- More memory access required than stack.
 - Lots of *spill code* necessary. Consider a program to do this:

$$A=B*C+D*E$$

General Purpose Register Architecture

- A Register is a *word* of internal memory (like the accumulator).
- A General Purpose Register architecture supports many registers – each can be used for anything:
 - holding operands for operations
 - holding temporary values

Number of Registers

- Early Register Instruction Sets supported a few registers (8 or less).
- Many current processors support 32 registers.
- The more registers available the fewer memory accesses will be necessary.

Register Example: A=B+C

```
Load R1,B      # R1 = B
Load R2,C      # R2 = C
Add  R3,R1,R2  # R3 = R1+R2
Store R3,A     # A = R3
```

Register Pros and Cons

- Instructions must include bits to specify which registers to operate on (large instruction size).
- Memory access can be minimized (registers can hold lots of intermediate values).
- Compiler writer now has to attempt to maximize register usage (minimize *spill code*). This is a tough job!

Modern Processors

- Most processors in use today are register based.
 - there are still a number of microcontrollers that are widely used that are accumulator based.
- Pentium, MIPS, Sparc, Alpha, PowerPC, PA-RISC are all G.P. Register architectures.

MIPS and the book

- The book uses the MIPS instruction set for most of the examples.
- So will I.
- The MIPS processor was designed by Hennessy (one of the authors of the text).
- Although we will learn MIPS in detail, the concepts apply to any instruction set.

Assembly Language

- We don't need to look at the machine code representation of instructions to understand them.
- Assembly Language is an English-like version of machine code.

Assembly Language Instructions

- One instruction per line.
 - each corresponds to an actual machine code instruction (bunch of bits).
- Use symbols instead of numbers (we can treat memory as *variables* instead of worrying about memory locations).

First Example Instruction

```
add a, b, c
```

- Adds **b** to **c** and stores the result in **a**.
- In 'C' this would be: **a=b+c;**

CompOrg Fall 2000 - Instruction Sets

25

```
a = b+c+d+e;
```

```
add a, b, c    # a=b+c  
add a, a, d    # a=a+d  
add a, a, e    # a=a+e
```

Everything to the right of “#” is a comment (ignored by the assembler).

CompOrg Fall 2000 - Instruction Sets

26

OK Dave, addition is easy, but how on Earth can we subtract things?

```
a=b+c-d;
```

```
add a, b, c    # a = b+c  
sub a, a, d    # a = a-d
```

CompOrg Fall 2000 - Instruction Sets

27

A tougher one: $a = (b+c) - (d+e);$

```
add t0, b, c      # t0 = b+c
sub t1, d, e      # t1 = d+e
sub a, t0, t1     # a = t0-t1
```

We have to use *temporary* holders `t0` and `t1` (actually we could get away with just one temporary holder).

What are `a,b,c,t0,t1,...`?

The operands are all registers (for the MIPS instruction set).

Some instruction sets allow some operands to be in memory and some in registers.

Some instruction sets don't support 3 operands, so the destination must be one of the 2 operands.

MIPS registers

- The MIPS processor has 32 general purpose registers (each is 32 bits wide).
- In MIPS assembly language the register names look like this:

`$s0, $s1, ...` and `$t0, $t1, ...`

We will find out why they are named like this a bit later.

MIPS Registers and 'C'

- For examples derived from 'C' code we will use:
 \$s0, \$s1, \$s2, ... for 'C' variables.
 \$t0, \$t1, \$t2, ... for temporary values.

a=(b+c)-(d+e);
\$s0 \$s1 \$s2 \$s3 \$s4

```
add $t0, $s1, $s2 # t0 = b+c
sub $t1, $s3, $s4 # t1 = d+e
sub $s0, $t0, $t1 # a = $t0-$t1
```

Registers vs. Memory

- In the MIPS instruction set, arithmetic operations occur *only* on registers.
- There may be more variables than registers.
- What about arrays?
- What about subroutines?
 - inside a subroutine we use different variables.

Load Instructions

- *Load* means to move from memory in to 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.

lw *destreg*, *const(addrreg)*

↑
"Load Word"

↑
A number

↑
Name of register
to put value in

↑
Name of register to get
base address from

address = (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]
```

Why the weird *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.

Array Example: `a=b+c[8];`



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

Words vs. Bytes

- Each byte in memory has a unique address.
- If the 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**

CompOrg Fall 2000 - Instruction Sets

43

a=b+c[8]; (*fixed*)

\nearrow \uparrow \uparrow
 $\$s0$ $\$s1$ $\$s2$

address of **c[8]** is $c+8*4$

lw \$t0,32(\$s2) # \$t0 = c[8]
add \$s0, \$s1, \$t0 # \$s0=\$s1+\$t0

CompOrg Fall 2000 - Instruction Sets

44

Moving from Register to 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).

CompOrg Fall 2000 - Instruction Sets

45

sw srcreg, const(addrreg)

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

CompOrg Fall 2000 - Instruction Sets 46

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

CompOrg Fall 2000 - Instruction Sets 47

Time for a Quiz

- 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**

CompOrg Fall 2000 - Instruction Sets 48

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

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

CompOrg Fall 2000 - Instruction Sets

49

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.

CompOrg Fall 2000 - Instruction Sets

50

a=b+c[i];

\$s0 \$s1 \$s2 \$s3

```
add $t0,$s3,$s3    # $t0=i+i
add $t0,$t0,$t0    # $t0=i+i+i+i
add $t0,$t0,$s2    # $t0=c+i*4
lw $t1,0($t0)     # $t1=c[i]
add $s0,$s1,$t1    # $s0=b+c[i]
```

CompOrg Fall 2000 - Instruction Sets

51

MIPS Instruction Summary (so far)

- MIPS has 32 32-bit registers with names like **\$s0**, **\$s1**, **\$t0**, **\$t1**, ...
- Data must be in registers for arithmetic operations.
- We've seen 2 arithmetic ops: **add** & **sub**
 - 3 operands – all registers.
- 2 Data transfer instructions: **lw**, **sw**
 - base/index addressing

CompOrg Fall 2000 - Instruction Sets

52

MIPS Machine Language

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

```
add $s0,$s0,$s2
```

- It does *understand* things like this:

```
10000101001010001100010000000101
```

CompOrg Fall 2000 - Instruction Sets

53

MIPS Machine Code Instructions

- Each instruction is encoded as 32 bits.
 - many processors have *variable* length instructions.
- There are a few different *formats* for MIPS instructions
 - which bits mean what.

CompOrg Fall 2000 - Instruction Sets

54

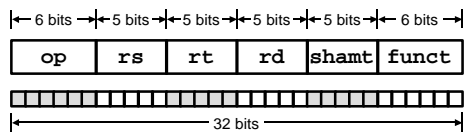
Instruction Formats

- break up the 32 bits in to *fields*.
- Each field is an encoding of part of the instruction:
 - fields that specify what registers to use.
 - what operation should be done.
 - constants.

CompOrg Fall 2000 - Instruction Sets

55

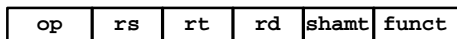
MIPS **add** instruction format



This format is used for many MIPS instructions (not just add). Instructions that use this format are called “*R-Type*” instructions.

CompOrg Fall 2000 - Instruction Sets

56



- op:** basic operation (opcode)
rs: first register source operand
rt: second register source operand
rd: destination register
shamt: shift amount (we can ignore for now)
funct: function code (indicates a specific type of operation **op**)

CompOrg Fall 2000 - Instruction Sets

57

Encodings

- For **add**:
 - **op** is 0_{10} (000000)
 - **funct** is 32_{10} (100000)
- Register encodings:
 - **\$s0** is 16_{10} (10000), **\$s1** is 17_{10} , ...
 - **\$t0** is 8_{10} (01000), **\$t1** is 9_{10} , ...

CompOrg Fall 2000 - Instruction Sets

58

add \$s0, \$s1, \$t0

000000 10001 01000 10000 00000 100000

op	rs	rt	rd	shamt	funct
----	----	----	----	-------	-------

In HEX, this add instruction is:

02288020

CompOrg Fall 2000 - Instruction Sets

59

MIPS **sub** Instructions

- Same format as the **add** instruction.
- **op** is 0_{10} (000000)
- **funct** is 34_{10} (100100)

CompOrg Fall 2000 - Instruction Sets

60

lw and sw instructions

lw: The **op** field is 35₁₀ (100011)

sw: The **op** field is 43₁₀ (101011)

↑
Only 1 bit difference!

CompOrg Fall 2000 - Instruction Sets 64

lw \$s0, 24(\$t1)

100011 01001 10000 0000000000011000

op	rs	rt	address
----	----	----	---------

sw \$s0, 24(\$t1)

101011 01001 10000 0000000000011000

op	rs	rt	address
----	----	----	---------

CompOrg Fall 2000 - Instruction Sets 65

Exercise

- What is the MIPS machine code for the following C statement:

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

CompOrg Fall 2000 - Instruction Sets 66

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

- First we can work on the Assembly instructions – assume a is \$s0 and the base address of c is in \$s1:

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

CompOrg Fall 2000 - Instruction Sets

67

lw \$t0, 8(\$s1)

100011 10001 01000 0000000000001000

op	rs	rt	address
----	----	----	---------

op is 35₁₀ for lw
rs is 17₁₀ for \$s1
rt is 8₁₀ for \$t0
address is 8₁₀

CompOrg Fall 2000 - Instruction Sets

68

add \$t0,\$t0,\$s0

000000 01000 10000 01000 00000 100000

op	rs	rt	rd	shamt	funct
----	----	----	----	-------	-------

op is 0₁₀ for add
funct is 32₁₀ for add
rs is 8₁₀ for \$t0
rt is 16₁₀ for \$s0
rd is 8₁₀ for \$t0
shamt is 0₁₀

CompOrg Fall 2000 - Instruction Sets

69

sw \$t0, 12(\$s1)

101011 10001 01000 0000000000001000

op	rs	rt	address
----	----	----	---------

op is 43₁₀ for **lw**
rs is 17₁₀ for **\$s1**
rt is 8₁₀ for **\$t0**
address is 12₁₀

CompOrg Fall 2000 - Instruction Sets

70

Machine code for `c[3] = a+c[2];`

```
10001110001010000000000000001000    lw $t0, 8($s1)
00000001000100000100000000100000    add $t0,$t0,$s0
10101110001010000000000000001000    sw $t0, 12($s1)
```

Congratulations – you are now on your way to
being qualified to be an *assembler!*

CompOrg Fall 2000 - Instruction Sets

71
