University of Arizona

CSc 340

Foundations of Computer Systems

Christian Collberg

February 23, 2001

Instruction Set Architectures

ISAs

- CPUs differ in the instructions they support. One important distinction is in the type and number of operands an instruction may have.
- The MIPS is called a three-address, load/store architecture. Each instruction can use up to three registers (addresses), and memory is accessed only through load and store operations.
- Important note: I will be describing non-MIPS architectures in this lecture, although I will retain a MIPS-like syntax for assembly code. Do not mistake this for real MIPS code!!!

Memory Operands

- RISC machines such as the MIPS are typically load/store architectures. Operations must be performed on registers; memory is only accessed through load/store instructions.
- Suppose we want to compute

$$a = (a + b + c) * d$$

on a MIPS. a, b, c, d are variables stored in memory at addresses that have labels with those names. The MIPS can only perform one operation per instruction, so the pseudo-code is:

Memory Operands...

Pseudo Code	MIPS Code for $a = (a + b + c) * d$
	`lw $t0, a # load a`
	`lw $t1, b # load b`
`a = a + b`	`add $t2, $t0, $t1 # a = a + b`
	`lw $t1, c # load c`
`a = a + c`	`add $t2, $t2, $t1 # a = a + c`
	`lw $t1, d # load d`
`a = a * d`	`mul $t0, $t2, $t1 # a = a * d`
	`sw $t0, a # store a`

Slide 11–4

Memory Operands...

- One must write 8 instructions to compute the expression.
- The computer actually executes 14 instructions, because the `lw`, `sw`, and `mul` instructions are actually pseudo-instructions that get translated into two instructions each by the assembler.
- It takes 5 memory accesses to evaluate the expression, four to load the four values, and one to store the result. Memory accesses can be very expensive, and may limit the overall performance of the code.

Slide 11–5

Memory Operands...

If MIPS could perform operations directly on the values stored in memory, the instruction

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

would add the values stored in memory at addresses a and b, and store the result in memory at the address c. We can now write our program as the following:

```
add   a, b, a
add   a, c, a
mul   a, d, a
```

Slide 11–6

Memory Operands...

- This took only three instructions!
- But, the CPU must still access memory to get the operands and store the result. Each instruction requires two memory accesses to load the operands and one to store the result, for a total of nine. You'll still want to use registers to hold intermediate values to avoid extra memory accesses:

```
add   $t0, b, a
add   $t0, c, $t0
mul   a, d, $t0
```

Slide 11–7

Registers vs. Addresses

- It takes more bits to specify an address than a register number. On the MIPS it takes 5 bits to specify a register, and 32 bits to specify an address. An instruction containing three addresses would have to be at least 96 bits long, much larger than a 32-bit MIPS word.
- You can make instructions variable-length to try to save space, but this complicates the CPU.
- It's harder for the CPU to decode the instruction. The operands can be either addresses or registers.

Three-address machine

- For the rest of the examples assume that operands can be addresses.
- The MIPS is a three-address machine, so you should be pretty familiar with this architecture. Each instruction can have up to three explicit addresses (registers). Usually two are the operands and one is the result.

Two-address machine

- In a two-address machine one of the addresses is both the destination and an (implicit) source. Thus,

$$\texttt{add a, b} \Rightarrow \texttt{a = a + b}$$

- The sample program becomes:

```
add   a, b     # a = a + b
add   a, c     # a = a + c
mul   a, d     # a = a * d
```

- Why build two-address machines? Often, the third address isn't needed, and not specifying it reduces the size of the instruction.

One-address (accumulator) machine

- A one-address machine has a special register called the accumulator that is the destination of (nearly) every instruction and also a source.
- For example,

$$\texttt{add a} \Rightarrow \texttt{acc = acc + a}$$

- Load and store instructions are needed to move data into and out of the accumulator.

One-address machine. . .

- `load a` ⇒ `acc = value at address a`
- `store a` ⇒ `value at address a = acc`
- Examples: PDP-8, Motorola 6809.
- The sample program:

```
load  a     # acc = a
add   b     # acc = acc + b
add   c     # acc = acc + c
mul   d     # acc = acc * d
store a     # a = acc
```

Zero-address machine (Stack machine)

- A stack machine has no explicit addresses in instructions! All operands and results are maintained on a stack. Each instruction pops its operands off the stack, and pushes its result. E.g.
  - `add` ⇒ pop two values off stack, push result
- Push and pop instructions are used to move data to and from the stack
  - `push a` ⇒ push value at address a onto stack
  - `pop a` ⇒ pop value off stack and store at address a
- Example: Burroughs 5500 (PostScript, Java Bytecode)

Zero-address machine. . .

- The sample program:

```
push  a
push  b
add
push  c
add
push  d
mul
pop   a
```


Stack Machine Example I

Source Code	Stack Code
VAR X,Y,Z : INTEGER;	[1] pusha X
BEGIN	[2] push 1
X := 1;	[3] store
WHILE X < 10 DO	[4] push X [5] push 10 [6] GE [7] BrTrue 14
X := Y + Z;	[8] pusha X [9] push Y [10] push Z [11] add [12] store
ENDDO	[13] jump 4
END;	

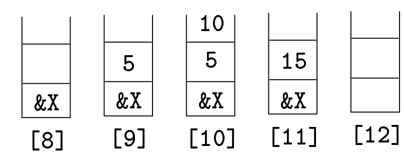
Stack Machine Example II (a)

Stack Code	Stack	Memory
[1] pusha X [2] push 1 [3] store	[1]: &X [2]: 1, &X [3]: (empty)	X: 1 Y: 5 Z: 10
[4] push X [5] push 10 [6] GE [7] BrTrue 14	[4]: 1 [5]: 10, 1 [6]: 0 [7]: (empty)	X: 1 Y: 5 Z: 10

Slide 11–16

Stack Machine Example II (b)

Stack Code	Stack	Memory
`[8] pusha X`		X: 15
`[9] push Y`		Y: 5
`[10] push Z`		Z: 10
`[11] add`		
`[12] store`		
`[13] jump 4`		



Slide 11–17

Another Example

- Compute

$$c = a/(b + (c * d))$$

- Three-address:

```
mul   x,c,d    # x = c * d
add   x,b,x    # x = b + x
div   c,a,x    # c = a / x
```

- Note: `x` is a temporary value.

Slide 11–18

Another Example

- Two-address:

```
mul   d,c      # d = c * d
add   b,d      # b = b + d
div   a,b      # a = a / b
move  c,a      # c = a
```

- Note that not having the third address hurt us in the div instruction. We wanted the result in "`c`", but had to settle for "`a`" and then move it to "`c`".

Slide 11–19

Another Example...

- One-address:

```
load  c        # acc = c
mul   d        # acc = acc * d
add   b        # acc = acc + b
store b        # b = acc
load  a        # acc = a
div   b        # acc = acc / b
store c        # c = acc
```

Another Example...

- Zero-address:

```
push  a       # stack = a
push  d       # stack = a d
push  c       # stack = a d c
mul           # stack = a d*c
push  b       # stack = a d*c b
add           # stack = a d*c+b
div           # stack = a/(d*c+b)
pop   c       # c = stack
```

Slide 11–20

Readings and References

- Maccabe, Section 4.0–4.2, pp. 115–130.

Slide 11–21