

Instruction Set

What's in a box?

- processor
- memory
- peripherals
 - keyboard
 - mouse
 - monitor
 - disk
 - floppy, CD, zip drive, ...
- network interface
 - modem, LAN tap (Ethernet, ...)

Processor Design

- processor resources
 - registers
 - arithmetic, logic unit,
 - datapath
- other resources
 - memory, input/output devices
 - buses to connect the resources
- instruction set
 - manipulate the computer resources
- how do you design instruction set?

How to understand Instr. Sets?

- Compile language constructs
- arithmetic
 - example 1
 - $a = b+c;$
 - $d = e-f;$
 - example 2
 - $f = (a+b) - (d+e);$
 - three operands per instruction
- where to keep operands of instructions?
 - I/O, memory, processor?

Operands

– Compiling with registers

$f = (a+b) - (d+e);$

– when to use operands in memory?

– Data transfer instructions

- load/store

- addressing memory operands

- » base+offset

- word Vs. byte operands

- alignment of word operands

- big endian Vs. little endian

- location of most significant byte of the word

Endian-ness

- at word address 100 (assume a 4-byte word)

long a = 0x11223344;

- big-endian (MSB at word address) layout

100

11	22	33	44
----	----	----	----

+0 +1 +2 +3

- little-endian (LSB at word address) layout

11	22	33	44
----	----	----	----

 100

+3 +2 +1 +0

- string layout starting at word addr 100
char a[12] = “RAMACHANDRAN”
char b[12] = “WAMACHANDRAN”

big-endian

100	R	A	M	A
104	C	H	A	N
108	D	R	A	N
112	W	A	M	A
116	C	H	A	N
120	D	R	A	N
	+0	+1	+2	+3

little endian

	A	M	A	R	100
	N	A	H	C	104
	N	A	R	D	108
	A	M	A	W	112
	N	A	H	C	116
	N	A	R	D	120
	+3	+2	+1	+0	

- Compiling with memory operands
 - $a[7] = b + a[7];$
- base register for “a”
- compiling with variable array index
 - $a[j] = h + a[j];$
 - base+index addressing mode
- spilling
- compiling “if”
 - if (j == k) go to L1;
 - a = b+c;
 - L1: a = a-1;
- Instructions and data have memory addresses

– Compiling if-then-else

if (j == k) a = b+c; else a = b-c;

– compiling loops

- example 1

loop: b = b + a[j]

if (j != 100) go to loop;

- example 2:

j = 0;

while (j < 100) {

 a[j] = 0;

}

– Compiling case statement

```
switch (k) {
```

```
  case 0:
```

```
  case 1:
```

```
  case 2:
```

```
  case 3:
```

```
}
```

Instruction Format

- Single format for all instructions?
- All instructions of the same length?
 - MIPS chose the latter over the former

Procedures

- Procedure abstraction
 - what is the programmer's model?
 - what does the compiler have to do?
 - simple hardware to support procedures
 - shadow area for processor “state”
 - save/restore on call/return
 - problems?

- hardware & instructions to support this model?
 - call/return
 - remember where we are in the program
 - » program counter (PC)
 - » what should happen on every instruction execution?
 - » will we ever need a PC if there were no procedure abstraction?
 - load/store
 - stack
 - » push and pop
 - » stack pointer (\$sp)
 - » stack frames
 - what do we store and restore on call/return?

- Software conventions
 - reserve some number of registers for parameters, return values, and return address
 - e.g. MIPS:
 - 4 for params, 2 for return values, one for return address
 - JAL <proc-addr>; \$ra <= return-addr
 - JR \$ra
 - what if we have more params or return values?
 - registers used in procedures
 - temporary registers
 - caller does not expect value to be preserved upon return
 - » MIPS: \$t0 to \$t9
 - saved registers
 - caller does expect value to be preserved on return
 - » MIPS: \$s0 to \$s7

- Compiling simple procedures
 - just need to save/restore registers used by called procedure
 - who should do this? caller? callee?
 - what regs in the MIPS example?
- Compiling nested procedures
 - registers needed after return may be trashed
 - e.g. MIPS: \$t0 to \$t9; \$a0 to \$a3; \$ra
 - solution:
 - save restore registers that will be needed upon return
 - who should do this?
- Compiling recursive procedures

- Summary

- caller saves registers (outside the agreed upon convention) at point of call

- MIPS: any \$t0 to \$t9; any \$a0 to \$a3; adjust \$sp

- callee saves registers (per convention) at point of entry

- MIPS: any \$s0 to \$s7 it uses in the body; \$ra; adjust \$sp

- callee restores saved registers, and re-adjusts stack before return

- caller restores saved registers, and re-adjusts stack before resuming from the call

- local variables for procedures
 - allocate on stack
 - software convention
 - designate a register as a frame pointer (\$fp)
 - makes references to local vars easier
 - why?
 - save/restore \$fp on call/return
 - push activation record on stack on call
 - pop activation record on return
 - additional params placed above \$fp

Other Goodies

- Extract/insert instructions
- Byte/half-word/word/multi-word load/store
 - this is the norm these days...
- Constants or immediate operands
- Branching
 - absolute jumps
 - relative branches
 - PC-relative
 - how to simulate jumping “far away”?

- MIPS addressing mode summary
 - register addressing
 - base+offset
 - immediate
 - PC-relative addressing
 - Pseudo-direct addressing

Beyond MIPS

- Architecture styles
 - stack-oriented
 - Burroughs machines
 - memory oriented
 - richer addressing modes for memory operands
 - IBM 360 series
 - register oriented
 - modern RISC architectures
 - load/store
 - simple addressing modes
 - arithmetic and logic using registers only