Instruction Set Architecture

or

“How to talk to computers”

The Instruction Set Architecture

The Instruction Execution Cycle

Brief Vocabulary Lesson

• superscalar processor -- can execute more than one instruction per cycle.
• cycle -- smallest unit of time in a processor.
• parallelism -- the ability to do more than one thing at once.
• pipelining -- overlapping parts of a large task to increase throughput without decreasing latency
Key ISA decisions

- operations
  - how many?
  - which ones
- operands
  - how many?
  - location
  - types
  - how to specify?
- instruction format
  - size
  - how many formats?

Crafting an ISA

- We’ll look at some of the decisions facing an instruction set architect, and
- how those decisions were made in the design of the MIPS instruction set.

Instruction Length

Variable: \[\begin{array}{cccccc}
\hline
\vdots
\hline
\end{array}\] Fixed: \[\begin{array}{ccc}
\hline
\hline
\end{array}\] Hybrid: \[\begin{array}{ccc}
\hline
\hline
\hline
\hline
\end{array}\]

- Variable-length instructions (Intel 80x86, VAX) require multi-step fetch and decode, but allow for a much more flexible and compact instruction set.
- Fixed-length instructions allow easy fetch and decode, and simplify pipelining and parallelism.

\[\Rightarrow\] All MIPS instructions are 32 bits long.
  - this decision impacts every other ISA decision we make because it makes instruction bits scarce.
Instruction Formats
• what does each bit mean?

- Having many different instruction formats...
  • complicates decoding
  • uses more instruction bits (to specify the format)

VAX II instruction format

<table>
<thead>
<tr>
<th>Byte 0</th>
<th>1</th>
<th>n</th>
<th>A/H</th>
<th>A/H</th>
<th>A/H</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ByteCode]</td>
<td>[A/H]</td>
<td>[A/H]</td>
<td>[AutoInc]</td>
<td>[Reg]</td>
<td></td>
</tr>
</tbody>
</table>

- VAX instruction format

MIPS Instruction Formats

<table>
<thead>
<tr>
<th>6 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>6 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>sa</td>
<td>funct</td>
</tr>
</tbody>
</table>

- the opcode tells the machine which format
- so add r1, r2, r3 has
  - opcode=0, funct=32, rs=2, rt=3, rd=1, sa=0
  - 000000 00010 00011 00000 00000 10000

Accessing the Operands

- operands are generally in one of two places:
  - registers (32 int, 32 fp)
  - memory (2^32 locations)

- registers are
  - easy to specify
  - close to the processor (fast access)

- the idea that we want to access registers whenever possible led to load-store architectures.
  - normal arithmetic instructions only access registers
  - only access memory with explicit loads and stores

Load-store architectures

- can do:
  - add r1=r2+r3
  - load r3, M(address)

- can’t do:
  - add r1 = r2 + M(address)

⇒ forces heavy dependence on registers, which is exactly what you want in today’s CPUs
  - more instructions
  - fast implementation (e.g., easy pipelining)
How Many Operands?

- Most instructions have three operands (e.g., \( z = x + y \)).
- Well-known ISAs specify 0-3 (explicit) operands per instruction.
- Operands can be specified implicitly or explicitly.

Basic ISA Classes

**Accumulator:**

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>add A</td>
<td>acc ← acc + mem[A]</td>
</tr>
</tbody>
</table>

**Stack:**

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>add</td>
<td>tos ← tos + next</td>
</tr>
</tbody>
</table>

**General Purpose Register:**

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>add A B</td>
<td>EA(A) ← EA(A) + EA(B)</td>
</tr>
<tr>
<td>3</td>
<td>add A B C</td>
<td>EA(A) ← EA(B) + EA(C)</td>
</tr>
</tbody>
</table>

**Load/Store:**

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>add Ra Rb Rc</td>
<td>Ra ← Rb + Rc</td>
</tr>
<tr>
<td></td>
<td>load Ra Rb</td>
<td>Ra ← mem[Rb]</td>
</tr>
<tr>
<td></td>
<td>store Ra Rb</td>
<td>mem[Rb] ← Ra</td>
</tr>
</tbody>
</table>

Comparing the Number of Instructions

Code sequence for \( C = A + B \) for four classes of instruction sets:

<table>
<thead>
<tr>
<th>Stack</th>
<th>Accumulator</th>
<th>GP Register</th>
<th>GP Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push A</td>
<td>Load A</td>
<td>ADD C, A, B</td>
<td>Load R1,A</td>
</tr>
<tr>
<td>Push B</td>
<td>Add B</td>
<td></td>
<td>Load R2,B</td>
</tr>
<tr>
<td>Add</td>
<td>Store C</td>
<td>Add R3,R1,R2</td>
<td></td>
</tr>
<tr>
<td>Pop C</td>
<td></td>
<td>Store C,R3</td>
<td></td>
</tr>
</tbody>
</table>

Alternate ISA’s

\[ A = X \times Y - B \times C \]
Addressing Modes

how do we specify the operand we want?

- Register direct  R3
- Immediate (literal)  #25
- Direct (absolute)  M[10000]
- Register indirect  M[R3]
- Base+Displacement  M[R3 + 10000]
- Base+Index  M[R3 + R4]
- Scaled Index  M[R3 + R4*d + 10000]
- Autoincrement  M[R3++]
- Autodecrement  M[R3 - -]
- Memory Indirect  M[ M[R3] ]

MIPS addressing modes

<table>
<thead>
<tr>
<th>register direct</th>
</tr>
</thead>
<tbody>
<tr>
<td>OP</td>
</tr>
<tr>
<td>add $1, $2, $3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>OP</td>
</tr>
<tr>
<td>add $1, $2, #35</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>base + displacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $1, disp($2)</td>
</tr>
<tr>
<td>register indirect</td>
</tr>
<tr>
<td>( \Rightarrow disp = 0 )</td>
</tr>
<tr>
<td>absolute</td>
</tr>
<tr>
<td>( \Rightarrow (rs) = 0 )</td>
</tr>
<tr>
<td>(R1 = M[R2 + disp])</td>
</tr>
</tbody>
</table>

Is this sufficient?

- measurements on the VAX show that these addressing modes (immediate, direct, register indirect, and base+displacement) represent 88% of all addressing mode usage.
- similar measurements show that 16 bits is enough for the immediate 75 to 80% of the time
- and that 16 bits is enough of a displacement 99% of the time.

Memory Organization (digression)

- Viewed as a large, single-dimension array, with an address.
- A memory address is an index into the array
- "Byte addressing" means that the index points to a byte of memory.
Memory Organization

- Bytes are nice, but most data items use larger "words"
- For MIPS, a word is 32 bits or 4 bytes.

For MIPS, a word is 32 bits or 4 bytes.

2^{32} bytes with byte addresses from 0 to 2^{32}-1
2^{30} words with byte addresses 0, 4, 8, ... 2^{32}-4
Words are aligned
i.e., what are the least 2 significant bits of a word address?

The MIPS ISA, so far

- fixed 32-bit instructions
- 3 instruction formats
- 3-operand, load-store architecture
- 32 general-purpose registers (integer, floating point)
  - R0 always equals 0.
- 2 special-purpose integer registers, HI and LO, because multiply and divide produce more than 32 bits.
- registers are 32-bits wide (word)
- register, immediate, and base+displacement addressing modes

Which instructions?

- arithmetic
- logical
- data transfer
- conditional branch
- unconditional jump

Which instructions (integer)

- arithmetic
  - add, subtract, multiply, divide
- logical
  - and, or, shift left, shift right
- data transfer
  - load word, store word
Control Flow

- Jumps
- Procedure call (jump subroutine)
- Conditional Branch
  - Used to implement, for example, if-then-else logic, loops, etc.

A conditional branch must specify two things
  - Condition under which the branch is taken
  - Location that the branch jumps to if taken (target)

Conditional branch

- How do you specify the destination of a branch/jump?
- Studies show that almost all conditional branches go short distances from the current program counter (loops, if-then-else).
  - We can specify a relative address in much fewer bits than an absolute address
  - E.g., `beq $1, $2, 100` => if ($1 == $2) PC = (PC+4) + 100 * 4
- How do we specify the condition of the branch?

MIPS conditional branches

- `beq, bne  beq r1, r2, addr => if (r1 == r2) goto addr`
- `slt $1, $2, $3 => if ($2 < $3) $1 = 1; else $1 = 0`
- These, combined with $0, can implement all fundamental branch conditions
  - Always, never, !=, = =, >, <=, >, >=(unsigned), <= (unsigned), ...

```c
if (i<j)
  w = w+1;
else
  w = 5;
```

Jumps

- Need to be able to jump to an absolute address sometime
- Need to be able to do procedure calls and returns

- Jump -- `j 10000` => PC = 10000
- Jump and link -- `jal 100000` => $31 = PC + 4; PC = 10000
  - Used for procedure calls
- Jump register -- `jr $31` => PC = $31
  - Used for returns, but can be useful for lots of other things.
Branch and Jump Addressing Modes

- Branch (e.g., beq) uses PC-relative addressing mode (uses few bits if address typically close). That is, it uses base + displacement mode, with the PC being the base. If opcode is 6 bits, how many bits are available for displacement? How far can you jump?

- Jump uses pseudo-direct addressing mode. 26 bits of the address is in the instruction, the rest is taken from the PC.

To summarize:

<table>
<thead>
<tr>
<th>Category</th>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>add</td>
<td>add $s_1, s_2, s_3</td>
<td>$s_1 + s_2 + s_3</td>
<td>Three operands, data in registers</td>
</tr>
<tr>
<td></td>
<td>sub</td>
<td>sub $s_1, s_2, s_3</td>
<td>$s_1 - s_2 - s_3</td>
<td>Three operands, data in registers</td>
</tr>
<tr>
<td></td>
<td>sb</td>
<td>sb $s_1, 100</td>
<td>$s_1 + 100</td>
<td>Used to add constants</td>
</tr>
<tr>
<td></td>
<td>lw</td>
<td>lw $s_1, 100($s_2)</td>
<td>$s_1 = Memory[$s_2 + 100]</td>
<td>Load word from memory to register</td>
</tr>
<tr>
<td></td>
<td>sw</td>
<td>sw $s_1, 100($s_2)</td>
<td>Memory[$s_2 + 100] = $s_1</td>
<td>Store word to memory</td>
</tr>
<tr>
<td></td>
<td>lui</td>
<td>lui $s_1, 100</td>
<td>$s_1 = 100 * 2 ^ 26</td>
<td>Load upper 16 bits</td>
</tr>
<tr>
<td></td>
<td>jal</td>
<td>jal 2500</td>
<td>$ra = PC + 4; go to 10000</td>
<td>For procedure call</td>
</tr>
<tr>
<td></td>
<td>branch on equal</td>
<td>beq $s_1, $s_2, 25</td>
<td>If ($s_1 == $s_2) go to PC + 4 + 25</td>
<td>Equal test; PC-relative branch</td>
</tr>
<tr>
<td></td>
<td>branch on not equal</td>
<td>bne $s_1, $s_2, 25</td>
<td>If ($s_1 != $s_2) go to PC + 4 + 25</td>
<td>Not equal test; PC-relative branch</td>
</tr>
<tr>
<td></td>
<td>get less than</td>
<td>slt $s_1, $s_2, $s_3</td>
<td>If ($s_2 &lt; $s_3) $s_1 = 1; else $s_1 = 0</td>
<td>Compare less than; for beq, bne</td>
</tr>
<tr>
<td></td>
<td>get less than immediate</td>
<td>slti $s_1, $s_2, 100</td>
<td>If ($s_2 &lt; 100) $s_1 = 1; else $s_1 = 0</td>
<td>Compare less than constant</td>
</tr>
<tr>
<td></td>
<td>jump</td>
<td>j 2500</td>
<td>go to 10000</td>
<td>Jump to target address</td>
</tr>
<tr>
<td></td>
<td>jump and link</td>
<td>jal 2500</td>
<td>$ra = PC + 4; go to 10000</td>
<td>For procedure call</td>
</tr>
</tbody>
</table>

Review -- Instruction Execution in a CPU

An Example

- Can we figure out the code?

```c
swap(int v[], int k):
{ int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
} swap:
  multi $2, $5, 4
  add $2, $4, $2
  lw $15, 0($2)
  lw $16, 4($2)
  sw $16, 0($2)
  sw $15, 0($2)
  jr $31
```
MIPS ISA Tradeoffs

<table>
<thead>
<tr>
<th>6 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>6 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>OP</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>sa</td>
<td>funct</td>
</tr>
<tr>
<td>OP</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OP</td>
<td>target</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

What if?
- 64 registers
- 20-bit immediates
- 4 operand instruction (e.g. Y = AX + B)

RISC Architectures

- MIPS, like SPARC, PowerPC, and Alpha AXP, is a RISC (Reduced Instruction Set Computer) ISA.
  - fixed instruction length
  - few instruction formats
  - load/store architecture
- RISC architectures worked because they enabled pipelining. They continue to thrive because they enable parallelism.

Alternative Architectures

- Design alternative:
  - provide more powerful operations
  - goal is to reduce number of instructions executed
  - danger is a slower cycle time and/or a higher CPI (cycles per instruction)
- Sometimes referred to as “RISC vs. CISC”
  - Reduced (Complex) Instruction Set Computer
  - virtually all new instruction sets since 1982 have been RISC
  - VAX: minimize code size, make assembly language easy instructions from 1 to 54 bytes long!
- We’ll look (briefly!) at PowerPC and 80x86

PowerPC

- Indexed addressing
  - example: lw $t1,$a0+$s3 #$t1=\text{Memory}[$a0+$s3]
  - What do we have to do in MIPS?
- Update addressing
  - update a register as part of load (for marching through arrays)
  - example: lwu $t0,4($s3) #$t0=\text{Memory}[$s3+4] ; $s3=$s3+4
  - What do we have to do in MIPS?
- Others:
  - load multiple/store multiple
  - a special counter register “bc Loop”
    decrement counter, if not 0 goto loop
80x86

- 1978: The Intel 8086 is announced (16 bit architecture)
- 1980: The 8087 floating point coprocessor is added
- 1982: The 80286 increases address space to 24 bits, +instructions
- 1985: The 80386 extends to 32 bits, new addressing modes
- 1989-1995: The 80486, Pentium, Pentium Pro add a few instructions (mostly designed for higher performance)
- 1997: MMX is added
- 1999: Pentium III (same architecture)
- 2001: Pentium 4 (144 new multimedia instructions), simultaneous multithreading (hyperthreading)
- 2005: dual core Pentium processors
- 2006: quad core (sort of) Pentium processors
- 2009: Nehalem – eight-core multithreaded processors

CSE 141  Dean Tullsen

80x86

- Complexity:
  - Instructions from 1 to 17 bytes long
  - one operand must act as both a source and destination
  - one operand can come from memory
  - complex addressing modes e.g., "base or scaled index with 8 or 32 bit displacement"

- Saving grace:
  - the most frequently used instructions are not too difficult to build
  - compilers avoid the portions of the architecture that are slow

CSE 141  Dean Tullsen

x86 Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Mode</th>
<th>Registers</th>
<th>Memory</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEAL, LEL, LEA</td>
<td>4</td>
<td>4</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>CALL</td>
<td>8</td>
<td>8</td>
<td>offset</td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td>4</td>
<td>4</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>ADD</td>
<td>4</td>
<td>4</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>SUB</td>
<td>4</td>
<td>4</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>XOR</td>
<td>4</td>
<td>4</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>OR</td>
<td>4</td>
<td>4</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>AND</td>
<td>4</td>
<td>4</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>TEST</td>
<td>4</td>
<td>4</td>
<td>8</td>
<td>4</td>
</tr>
</tbody>
</table>

Key Points

- MIPS is a general-purpose register, load-store, fixed-instruction-length architecture.
- MIPS is optimized for fast pipelined performance, not for low instruction count
- Historic architectures favored code size over parallelism.
- MIPS most complex addressing mode, for both branches and loads/stores is base + displacement.