Instruction Set Architecture

or

“How to talk to computers if you aren’t in Star Trek”

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

The Instruction Execution Cycle

- Obtain instruction from program storage
- Determine required actions and instruction size
- Locate and obtain operand data
- Compute result value or status
- Deposit results in storage for later use
- Determine successor instruction

Key ISA decisions

- operations
  - source operands
  - destination operand
- operands
  - location
  - types
  - how to specify?
- instruction format
  - how does the computer know what 0001 0100 1101 1111 means?
  - 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-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.

 رغم أن جميع تعليمات MIPS هي 32 بت، هذا القرار ي 影响每其他 ISA 决策，因为这使得 instruction bits 少。

Instruction Formats

- Having many different instruction formats...
  - uses more (to specify the format)

VAX 11 instruction format

- register
- disp
- index
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= , funct= , rs= , rt= , rd= , sa= 
  - 000000 00010 00011 00001 00000 100000

Accessing the Operands

- operands are generally in one of two places:
  - (32 int, 32 fp)
  - (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)

and

⇒ 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 operands (e.g., z = x + y).
- Well-known ISAs specify 0-3 (explicit) operands per instruction.
- Operands can be specified or .
How Many Operands?

Basic ISA Classes

Accumulator:
1 address
add A
acc ← acc + mem[A]

Stack:
0 address
add
tos ← tos + next

General Purpose Register:
2 address
add A B
EA(A) ← EA(A) + EA(B)

3 address
add A B C
EA(A) ← EA(B) + EA(C)

Load/Store:
3 address
add Ra Rb Rc
Ra ← Rb + Rc

load Ra Rb
Ra ← mem[Rb]

store Ra Rb
mem[Rb] ← Ra

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 (register-memory)</th>
<th>GP Register (load-store)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A = X<em>Y - B</em>C</td>
<td>Stack Architecture</td>
<td>Accumulator</td>
<td>GPR (Load-store)</td>
</tr>
</tbody>
</table>

Alternate ISA’s

A = X*Y - B*C

Addressing Modes

how do we specify the operand we want?

- Register direct       R3
- Immediate (literal)   #25
- Direct (absolute)     M[10000]
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] ]

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.

MIPS addressing modes

register direct

<table>
<thead>
<tr>
<th>OP</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>sa</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

dd $1, $2, $3


immediate

<table>
<thead>
<tr>
<th>OP</th>
<th>rs</th>
<th>rt</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

dd $1, $2, #35

base + displacement

lw $1, disp($2)

RS

rt immediate

(RI = M[R2 + disp])

Memory Organization

• 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.

<table>
<thead>
<tr>
<th></th>
<th>32 bits of data</th>
<th>32 bits of data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>registers hold 32 bits of data</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- $2^{32}$ 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

What’s left

- which instructions?
- odds and ends

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 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 + 100 * 4
- How do we specify the of the branch?

MIPS conditional branches

- beq, bne  
  \[ \text{beq } r1, r2, addr \Rightarrow \text{if } (r1 == r2) \text{ goto addr} \]
- 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), ...

  if (i<j)
  \[
  \begin{align*}
  w &= w + 1; \\
  \text{else} & \\
  w &= 5;
  \end{align*}
  \]
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 , 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>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>add</td>
<td>add $s1, $s2, $s3</td>
<td>$s1 = $s2 + $s3</td>
</tr>
<tr>
<td></td>
<td>sub</td>
<td>sub $s1, $s2, $s3</td>
<td>$s1 = $s2 - $s3</td>
</tr>
<tr>
<td></td>
<td>addi</td>
<td>addi $s1, $s2, 100</td>
<td>$s1 = $s2 + 100</td>
</tr>
<tr>
<td>Data transfer</td>
<td>lw</td>
<td>lw  $s1, 100($s2)</td>
<td>$s1 = Memory($s2 + 100)</td>
</tr>
<tr>
<td></td>
<td>sw</td>
<td>sw  $s1, 100($s2)</td>
<td>Memory($s2 + 100) = $s1</td>
</tr>
<tr>
<td></td>
<td>lb</td>
<td>lb  $s1, 100($s2)</td>
<td>$s1 = Memory($s2 + 100)</td>
</tr>
<tr>
<td></td>
<td>sb</td>
<td>sb  $s1, 100($s2)</td>
<td>Memory($s2 + 100) = $s1</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;
}
```

```mips
muli $2, $5, 4
add $2, $4, $2
lw $15, 0($2)
lw $16, 4($2)
sw $16, 0($2)
sw $15, 4($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.
  – instruction length
  – instruction formats
  – 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=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)` #$t0=Memory[$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

- See your textbook for a more detailed description
- 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

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.