Instruction Set Architecture

or

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

The Instruction Set Architecture

I/O system

Instr. Set Proc.

Compiler

Operating System

Application

Digital Design

Circuit Design

The Instruction Execution Cycle

Instruction Fetch

Instruction Decode

Operand Fetch

Execute

Result Store

Next Instruction

Brief Vocabulary Lesson

- *superscalar processor* -- can execute more than one instructions 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

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**
  - how many?
  - which ones

- **operands**
  - how many?
  - location
  - types
  - how to specify?

- **instruction format**
  - size
  - how many formats?

\[
\begin{align*}
y &= x + b \\
(\text{add } r1, r2, r5)
\end{align*}
\]

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.
- 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.

Instruction Length

<table>
<thead>
<tr>
<th>Variable:</th>
<th>Fixed:</th>
<th>Hybrid:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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.

⇒ 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 instruction bits (to specify the format)

VAX 11 instruction format

<table>
<thead>
<tr>
<th>Byte</th>
<th>0</th>
<th>n</th>
<th>m</th>
</tr>
</thead>
<tbody>
<tr>
<td>spCode</td>
<td>A/R</td>
<td>A/R</td>
<td>A/R</td>
</tr>
</tbody>
</table>

operand specifier

<table>
<thead>
<tr>
<th>register</th>
<th>disp</th>
<th>autoinc</th>
<th>8 r</th>
</tr>
</thead>
<tbody>
<tr>
<td>A r</td>
<td>r byte</td>
<td>half word</td>
<td>word</td>
</tr>
</tbody>
</table>

VAX 11 instruction format

| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |
| OP | rs | rt | rd | sa | funct |

MIPS Instruction Formats

- 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 0010 00011 00000 00001 00000 100000

Accessing the Operands

- operands are generally in one of two places:
  - registers are
  - index

- 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.

Basic ISA Classes

**Accumulator:**
1 address
\[ \text{add } A \quad \text{acc} \leftarrow \text{acc} + \text{mem}[A] \]

**Stack:**
0 address
\[ \text{add } \quad \text{tos} \leftarrow \text{tos} + \text{next} \]

**General Purpose Register:**
2 address
\[ \text{add } A \ B \quad \text{EA}(A) \leftarrow \text{EA}(A) + \text{EA}(B) \]
3 address
\[ \text{add } A \ B \ C \quad \text{EA}(A) \leftarrow \text{EA}(B) + \text{EA}(C) \]

**Load/Store:**
3 address
\[ \text{add } R_a \ R_b \ R_c \quad R_a \leftarrow R_b + R_c \]
\[ \text{load } R_a \ R_b \quad R_a \leftarrow \text{mem}[R_b] \]
\[ \text{store } R_a \ R_b \quad \text{mem}[R_b] \leftarrow R_a \]

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>Register (register-memory)</th>
<th>Register (load-store)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push A</td>
<td>Load A</td>
<td>ADD C, A, B</td>
<td></td>
</tr>
<tr>
<td>Push B</td>
<td>Add B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td>Store C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pop C</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Alternate ISA’s

$A = X*Y - B*C$

Stack Architecture

Accumulator

GPR

GPR (Load-store)

Memory

\[ \begin{array}{|c|}
\hline
\text{Accumulator} \quad \text{Stack} \quad \text{Memory} \\
\hline
\text{R}_1 \\
\text{R}_2 \\
\text{R}_3 \\
\hline
\end{array} \]

CSE 141  Dean Tullsen  
CSE 141  Dean Tullsen  
CSE 141  Dean Tullsen  
CSE 141  Dean Tullsen
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]
  
  if register is the program counter, this is PC-relative
- Base+Index M[R3 + R4]
- Scaled Index M[R3 + R4^d + 10000]
- Autodecrement M[R3 - -]

- Memory Indirect M[ M[R3] ]

MIPS addressing modes

<table>
<thead>
<tr>
<th></th>
<th>OP</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>sa</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>register direct</td>
<td>add $1, $2, $3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>immediate</td>
<td>add $1, $2, #35</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>base + displacement</td>
<td>lw $1, disp($2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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 is enough for the immediate 75 to 80% of the time
• and that is enough of a displacement 99% of the time.

Memory Organization

• Viewed as a large, single-dimension array, with an address.
• A memory address is a into the array
• " " 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.
  
  | 0 | 32 bits of data |
  | 4 | 32 bits of data |
  | 8 | 32 bits of data |
  | 12| 32 bits of data |

  Registers hold 32 bits of data

- $2^{32}$ bytes with byte addresses from 0 to 232-1
- $2^{30}$ words with byte addresses 0, 4, 8, ... 232-4

The MIPS ISA, so far

- 3 instruction formats
  
  - 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.
  - 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

Conditional branch

- How do you specify the target 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 target of the branch?

MIPS conditional branches

- beq, bne  
  
- 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)
  w = w+1;
else
  w = 5;
```

Jumps

- need to be able to jump to an address sometime
- need to be able to do and returns

- jump -- j 10000 => PC = 10000
- jump and link -- jal 100000 => $31 = PC + 4; PC = 10000
  - used for procedure calls

```
| OP | target |
```

- 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 **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 addressing mode. 26 bits of the address is in the instruction, the rest is taken from the PC.

```
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Program Counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>26</td>
</tr>
<tr>
<td>4</td>
<td>26</td>
</tr>
<tr>
<td>00</td>
<td></td>
</tr>
</tbody>
</table>
```

Jump destination address

To summarize:

**MIPS operands**

<table>
<thead>
<tr>
<th>Name</th>
<th>Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 registers</td>
<td>$s0-$s7, $t0-$t9, $zero</td>
<td>Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants.</td>
</tr>
<tr>
<td>32 registers</td>
<td>$a0-$a3, $v0-$v1, $gp</td>
<td>For switch, procedure return</td>
</tr>
<tr>
<td>32 registers</td>
<td>$fp, $sp, $ra, $at</td>
<td>Reserved for the assembler to handle large constants.</td>
</tr>
<tr>
<td>Memory</td>
<td>Memory[0], Memory[4], ...</td>
<td>Sequential words differ by 4. Memory holds data structures, such as arrays, and stack frames, such as those saved on procedure calls.</td>
</tr>
<tr>
<td>Registers</td>
<td>R0-R31</td>
<td>32 registers</td>
</tr>
</tbody>
</table>

**MIPS assembly language**

<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 $s1, $s2, $s3</td>
<td>$s1 = $s2 + $s3</td>
<td>Three operands; data in registers</td>
<td></td>
</tr>
<tr>
<td>Data transfer</td>
<td>load $s1, 100($s2)</td>
<td>$s1 = Memory[$s2 + 100]</td>
<td>Word from memory to register</td>
<td></td>
</tr>
<tr>
<td>Conditional branch</td>
<td>branch on equal</td>
<td>bne $s1, $s2, 25</td>
<td>if ($s1 != $s2) go to PC + 4 + 100</td>
<td></td>
</tr>
<tr>
<td>Unconditional branch</td>
<td>jump</td>
<td>j 2500</td>
<td>go to 10000</td>
<td></td>
</tr>
</tbody>
</table>

**Review -- Instruction Execution in a CPU**

- **Registers**
- **Program Counter**
- **CPU**
- **Memory**
- **Instruction Buffer**
- **Operation**
- **ALU**
- **Load/Store Unit**
- **Data**
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 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?

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 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:lw $t0,4($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 “be 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)
- 2000 Pentium 4 (144 new multimedia instructions)

### 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
- Four principles of IS architecture
  - Regularity produces simplicity
  - Smaller is faster
  - Good design demands compromise
  - Make the common case fast