Instruction Set Architecture

"Speaking with the computer"

The Instruction Set Architecture

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

1. **Instruction Fetch**
2. **Instruction Decode**
3. **Operand Fetch**
4. **Execute**
5. **Result Store**
6. **Next Instruction**

- 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?

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: 

Fixed: 

Hybrid: 

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.

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 11 instruction format

<table>
<thead>
<tr>
<th>Byte 0</th>
<th>1</th>
<th>n</th>
<th>m</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>A/Fi</td>
<td>A/Fi</td>
<td>A/Fi</td>
</tr>
</tbody>
</table>

**Register specifiers**

- register $r$
- autoinc $i$
- disp (byte, half word, word)
- index $r m r$ displacement

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>
<tr>
<td>opcode</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>opcode</td>
<td>target</td>
<td></td>
<td></td>
<td></td>
<td></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 00001 00000 100000

Accessing the Operands

- operands are generally in one of two places:
  - registers (32 int, 32 fp)
  - memory ($2^{12}$ 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.

### How Many Operands?

**Basic ISA Classes**

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

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

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

**Load/Store:**
- 3 address  
  \[
  \text{add Ra Rb Rc} \\
  \text{Ra} \leftarrow \text{Ra} + \text{Rc}
  \]
  \[
  \text{load Ra Rb} \\
  \text{Ra} \leftarrow \text{mem[Rb]}
  \]
  \[
  \text{store Ra Rb} \\
  \text{mem[Rb]} \leftarrow \text{Ra}
  \]

### 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</th>
<th>GP Register</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>(register-memory)</td>
<td>(load-store)</td>
</tr>
</tbody>
</table>

- **Stack:**
  - **Push A**
  - **Push B**
  - **Add**
  - **Pop C**

- **Accumulator:**
  - **Load A**

- **GP Register:**
  - **ADD C, A, B**

- **GP Register:**
  - **Load R1,A**
  - **Load R2,B**
  - **Add R3,R1,R2**
  - **Store C,R3**

_CSE 141, S2'06_  
_Jeff Brown_
Alternate ISA’s

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

Stack Architecture  Accumulator  GPR  GPR (Load-store)

<table>
<thead>
<tr>
<th>Accumulator</th>
<th>Stack</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>R2</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>R3</td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>temp</td>
</tr>
</tbody>
</table>

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: \( M[R3 + R4 \times d + 10000] \)
- 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>
</tbody>
</table>

- add \$1, \$2, \$3

<table>
<thead>
<tr>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>OP</td>
</tr>
</tbody>
</table>

- add \$1, \$2, \#35

<table>
<thead>
<tr>
<th>base + displacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $1, disp$($2)</td>
</tr>
</tbody>
</table>

- \( R1 = M[R2 + \text{disp}] \)

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 for branch displacement 99% of the time.
- (so: yes, as long as we can handle all cases, somehow)
Memory Organization

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

![Memory Map]

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>Registers hold 32 bits of data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: 32 bits of data</td>
</tr>
<tr>
<td>1: 32 bits of data</td>
</tr>
<tr>
<td>2: 32 bits of data</td>
</tr>
<tr>
<td>3: 32 bits of data</td>
</tr>
<tr>
<td>4: 32 bits of data</td>
</tr>
<tr>
<td>5: 32 bits of data</td>
</tr>
<tr>
<td>6: 32 bits of data</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

Memory Organization

- \(2^{32}\) 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"
  (what are the least-significant 2 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.
- registers are 32-bits wide (word)
- 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 (operations)?
- 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

- Jump
  - Jump ("goto", "break", ...)
  - Jump subroutine (procedure or function call)
- Conditional branch
  - If-then-else logic, loops, etc.
- A conditional branch must specify two things
  - Condition: determines whether the branch is taken
  - Target: location that the branch jumps to, if taken

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 + 100 * 4
- How do we specify the condition of the branch?
MIPS conditional branches

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

if (i < j)
  \[ w = w + 1; \]
else
  \[ w = 5; \]

CSE 141, S2'06  
Jeff Brown

Branch and Jump Addressing Modes

- Branches (e.g., `beq`) use PC-relative addressing mode.
  - base+displacement mode, with current PC as the base
  - opcode is 6 bits, register numbers are 10 bits; how many bits are available for displacement? How far can you jump?
- Jump uses pseudo-direct addressing mode.
  - The low 26 bits of the target comes directly from the instruction; the rest is taken from the PC. (No addition.)

<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>4</td>
<td>26</td>
</tr>
<tr>
<td>2</td>
<td>00</td>
</tr>
</tbody>
</table>

jump destination address

CSE 141, S2'06  
Jeff Brown

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

<table>
<thead>
<tr>
<th>OP</th>
<th>target (26 bits)</th>
</tr>
</thead>
</table>

- `jump register -- jr $31`  
  `=> PC = $31`
- `use for returns, but can be useful for lots of other things.`

CSE 141, S2'06  
Jeff Brown

To summarize:

<table>
<thead>
<tr>
<th>Name</th>
<th>Example</th>
<th>MIPS operands</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 registers</td>
<td>$0 = $1 + 4, $0 = $1 + 5</td>
<td>$0 = $1 + 4, $0 = $1 + 5</td>
<td>Three operands, data in registers</td>
</tr>
<tr>
<td>2nd memory words</td>
<td>Memory[0], ...</td>
<td>Memory[0]</td>
<td>Two memory words, no sequential words differ by 4. Memory holds data structures, such as arrays, and control registers, such as those reserved for procedure calls.</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</td>
<td>add $3, $2, $1</td>
<td>$3 = $3 + $1</td>
<td>Three operands, data in registers</td>
</tr>
<tr>
<td>Data transfer</td>
<td>addw</td>
<td>addw $3, $2, $1</td>
<td>$3 = $3 + $1</td>
<td>Signed in lower 16 bits</td>
</tr>
<tr>
<td>Data transfer</td>
<td>lw</td>
<td>lw $1, $2, 0</td>
<td>$1 = Memory[$2]</td>
<td>Used from memory to register</td>
</tr>
<tr>
<td>Data transfer</td>
<td>sw</td>
<td>sw $1, $2, 0</td>
<td>Memory[$2] = $1</td>
<td>Used from register to memory</td>
</tr>
<tr>
<td>Data transfer</td>
<td>load</td>
<td>load $1, 100</td>
<td>$1 = Memory[0]</td>
<td>Data transfer to register</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>loadc</td>
<td>loadc $1, 100</td>
<td>$1 = Memory[0] + 100</td>
<td>Load constant in upper 16 bits</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>lesb</td>
<td>lesb $1, $2, $3</td>
<td>$2 = Memory[$1]</td>
<td>Load boolean expression</td>
</tr>
<tr>
<td>Conditional branch</td>
<td>beq</td>
<td>beq $3, $2, $1</td>
<td>$3 = $2 + $1</td>
<td>Equal (branch)</td>
</tr>
<tr>
<td>Conditional branch</td>
<td>jlt</td>
<td>jlt $3, $2, $1</td>
<td>$3 = $2 + $1</td>
<td>Not equal (branch)</td>
</tr>
<tr>
<td>Jump</td>
<td>jr</td>
<td>jr $1, $2, $3</td>
<td>$1 = Memory[0]</td>
<td>Jump register</td>
</tr>
</tbody>
</table>

CSE 141, S2'06  
Jeff Brown
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+i];
    v[k+i] = temp;
}
```

```c
mul $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></th>
<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>R</td>
<td>OP</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>sa</td>
<td>funct</td>
</tr>
<tr>
<td>I</td>
<td>OP</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>J</td>
<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 or specialized 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)

---

**80x86**

- 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.