Instruction Set Architecture

"Speaking with the computer"
The Instruction Set Architecture

Instruction Set Architecture

Operating System

Compiler

Instr. Set Proc.

I/O system

Digital Design

Circuit Design

Application

CSE 141, S2'06

Jeff Brown
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

**Instruction Fetch**
- Obtain instruction from program storage

**Instruction Decode**
- Determine required actions and instruction size

**Operand Fetch**
- Locate and obtain operand data

**Execute**
- Compute result value or status

**Result Store**
- Deposit results in storage for later use

**Next Instruction**
- 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?

\[ y = x + b \]

(destination operand) operation

(source operands)

(y = x + b) (add r1, r2, r5)

how does the computer know what 0001 0100 1101 1111 means?
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: [blank] [blank] [blank] ... [blank]

Fixed: [blank]

Hybrid: [blank] [blank] [blank] [blank]
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**

```
Byte 0 1  n  m
OpCode A/M A/M A/M

operand specifier

register
5  r autoinc 8  r

disp
A  r byte
C  r half word
E  r word

index
4  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>
</tbody>
</table>

<table>
<thead>
<tr>
<th>6 bits</th>
<th>5 bits</th>
<th>5 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>rs</td>
<td>rt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>immediate</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>6 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
</tr>
<tr>
<td>target</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 ($2^{32}$ 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
and
  load r3, M(address)

⇒ forces heavy dependence on registers, which is exactly what you want in today’s CPUs

can’t do
  add r1 = r2 + M(address)

- 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   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>
</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></td>
<td></td>
<td>(register-memory)</td>
<td>(load-store)</td>
</tr>
<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></td>
<td>Add R3,R1,R2</td>
</tr>
<tr>
<td>Pop C</td>
<td></td>
<td></td>
<td>Store C,R3</td>
</tr>
</tbody>
</table>
Alternate ISA’s

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

Stack Architecture  Accumulator  GPR  GPR (Load-store)

Accumulator  Stack  Memory

R1  R2  R3

A  X  Y  B  C  temp

12  3  4  5

\[ ? \]  \[ ? \]  \[ ? \]
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

**register direct**

```
| OP | rs | rt | rd | sa | funct |
```

```
add $1, $2, $3
```

**immediate**

```
| OP | rs | rt | immediate |
```

```
add $1, $2, #35
```

**base + displacement**

```
lw $1, disp($2)
```

```
(R1 = M[R2 + disp])
```

**register indirect**

```
register indirect
```

```
disp = 0
```

```
absolute
```

```
(rs) = 0
```
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.

```
0       8 bits of data
1       8 bits of data
2       8 bits of data
3       8 bits of data
4       8 bits of data
5       8 bits of data
6       8 bits of data
...    
```
Memory Organization

- Bytes are nice, but most data items use larger "words"
- 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"
  (what are the least-significant 2 bits of a word address?)

Registers hold 32 bits of data
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`  
  \[ \text{beq } r1, r2, \text{ addr } \Rightarrow \text{if } (r1 == r2) \text{ goto } \text{ addr} \]

- `slt $1, $2, $3`  
  \[ \text{if } ($2 < $3) \text{ $1 }= 1; \text{ else } \text{ $1 }= 0 \]

- these, combined with $0, can implement all fundamental branch conditions
  Always, never, !=, = =, >, <=, >=, <, ...

```plaintext
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

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

• jump register -- jr $31 => PC = $31
  – used for returns, but can be useful for lots of other things.
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.)
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, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at</td>
<td>Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants.</td>
</tr>
<tr>
<td>$a0-$a3, $v0-$v1, gp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$fp, $sp, $ra, $at</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory[0], Memory[4], ..., Memory[4294967292]</td>
<td>Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.</td>
<td></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 $s1, $s2, $s3</td>
<td>$s1 = $s2 + $s3</td>
<td>Three operands; data in registers</td>
</tr>
<tr>
<td></td>
<td>subtract</td>
<td>sub $s1, $s2, $s3</td>
<td>$s1 = $s2 - $s3</td>
<td>Three operands; data in registers</td>
</tr>
<tr>
<td></td>
<td>add immediate</td>
<td>addi $s1, $s2, 100</td>
<td>$s1 = $s2 + 100</td>
<td>Used to add constants</td>
</tr>
<tr>
<td>Data transfer</td>
<td>load word</td>
<td>lw $s1, 100($s2)</td>
<td>$s1 = Memory[$s2 + 100]</td>
<td>Word from memory to register</td>
</tr>
<tr>
<td></td>
<td>store word</td>
<td>sw $s1, 100($s2)</td>
<td>Memory[$s2 + 100] = $s1</td>
<td>Word from register to memory</td>
</tr>
<tr>
<td></td>
<td>load byte</td>
<td>lb $s1, 100($s2)</td>
<td>$s1 = Memory[$s2 + 100]</td>
<td>Byte from memory to register</td>
</tr>
<tr>
<td></td>
<td>store byte</td>
<td>sb $s1, 100($s2)</td>
<td>Memory[$s2 + 100] = $s1</td>
<td>Byte from register to memory</td>
</tr>
<tr>
<td></td>
<td>load upper immediate</td>
<td>lui $s1, 100</td>
<td>$s1 = 100 * 2 ^ 16</td>
<td>Loads constant in upper 16 bits</td>
</tr>
<tr>
<td>Conditional branch</td>
<td>branch on equal</td>
<td>beq $s1, $s2, 25</td>
<td>if ($s1 == $s2) go to PC + 4 + 100</td>
<td>Equal test; PC-relative branch</td>
</tr>
<tr>
<td></td>
<td>branch on not equal</td>
<td>bne $s1, $s2, 25</td>
<td>if ($s1 != $s2) go to PC + 4 + 100</td>
<td>Not equal test; PC-relative</td>
</tr>
<tr>
<td></td>
<td>set on less than</td>
<td>slt $s1, $s2, $s3</td>
<td>if ($s2 &lt; $s3) $s1 = 1; else $s1 = 0</td>
<td>Compare less than; for beq, bne</td>
</tr>
<tr>
<td></td>
<td>set less than immediate</td>
<td>slti $s1, $s2, 100</td>
<td>if ($s2 &lt; 100) $s1 = 1; else $s1 = 0</td>
<td>Compare less than constant</td>
</tr>
<tr>
<td>Unconditional jump</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 register</td>
<td>jr $ra</td>
<td>go to $ra</td>
<td>For switch, procedure return</td>
</tr>
<tr>
<td></td>
<td>jump and link</td>
<td>jal 2500</td>
<td>$ra = PC + 4; go to 10000</td>
<td>For procedure call</td>
</tr>
</tbody>
</table>
Review -- Instruction Execution in a CPU

CPU components:
- Registers
  - R0: 0
  - R1: 36
  - R2: 60000
  - R3: 45
  - R4: 198
  - R5: 12
- Program Counter: 10000
- Instruction Buffer
- CPU operations:
  - ALU (operation, in1, in2, out)

Memory components:
- Address: 10000, 10004
- Data: 1000110001000110111000100000
- Immediate/displacement
- Load/Store Unit

Diagram showing the flow of data and instructions between the CPU and Memory.
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;
}
```

```
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

- 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.