Instruction Set Architectures

Part 2

- Application
- Compiler
- Operating System
- I/O system
- Digital Design
- Circuit Design

Instruction Set Architecture
Key ISA decisions

- instruction length
  - are all instructions the same length?
- how many registers?
- where do operands reside?
  - e.g., can you add contents of memory to a register?

instruction format
  - which bits designate what??

operands
  - how many? how big?
  - how are memory addresses computed?

operations
  - what operations are provided??
Instruction formats
- what does each bit mean?

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

Machine needs to determine quickly,
  • “This is a 6-byte instruction”
  • “Bits 7-11 specify a register”
  • ...
MIPS Instruction Formats

<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 format</td>
<td>OP</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>sa</td>
<td>funct</td>
</tr>
<tr>
<td>i format</td>
<td>OP</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>j format</td>
<td>OP</td>
<td></td>
<td></td>
<td>target</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- for instance, “add r1, r2, r3” has
  - OP=0, rs=2, rt=3, rd=1, sa=0, funct=32
  - 0000000 00010 00011 00001 000000 100000

- opcode (OP) tells the machine which format
VAX Instruction Formats

1 Byte OP code
specifies number, type and length of operands

Each operand is 1 to many bytes
first byte specifies addressing mode

“move” can be load, store, mem copy, jump,... depending on operands
How Many Operands?

- **Two-address code**: target is same as one operand
  - E.g., $x = x + y$

- **Three-address code**: target can be different
  - E.g., $x = y + z$
  - x86 doesn’t have three-address instructions; others do

- Some operands are also specified implicitly
  - “condition code” setting shows if result was $+$, $0$, or $-$
  - PowerPC’s “Branch on count” uses special “count register”

- Well-known ISA’s have 0-4 (explicit) operands
  - PowerPC has “float multiply add”, $r = x + y*z$
  - It also has fancy bit-manipulation instructions
Addressing Modes
how do we specify the operand we want?

Immediate

#25

The operand (25) is part of the instruction

Register

R3  (sometimes written $3)

The operand is the contents of register 3

Register indirect

M[R3]

Use contents of R3 as address into memory; find the operand there. This is a special case of...

Base+Displacement

M[R3 + 160]

Add the displacement (160) to contents of R3, look in that memory location for the operand
- If register is PC, this is “PC-relative addressing”

All our example ISA’s have the above modes
More Addressing Modes
(not included in MIPS ISA)

Base+Index $M[R3 + R4]$
Add contents of R3 and R4 to get memory address

Autoincrement $M[R3++]$ (or $M[R3+=d]$)
Find value in memory location designated by R3, but also increment R3
- Useful for accessing array elements
- Autodecrement is similar

Scaled Index $M[R3 + R4*d]$ (VAX and x86)
Multiply R4 by d (d is typically 1,2,4, or 8), then add R3 to get memory address

Memory Indirect $M[ M[R3] ]$ (only VAX)
Find number in memory location R3, use THAT as address into memory to find
VAX addressing mode usage

- Half of all references were register-mode
- Remaining half distributed as follows:

<table>
<thead>
<tr>
<th>Program</th>
<th>Base + Displacement</th>
<th>Immediate</th>
<th>Scaled Index</th>
<th>Memory Indirect</th>
<th>All Others</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEX</td>
<td>56%</td>
<td>43%</td>
<td>0</td>
<td>1%</td>
<td>0</td>
</tr>
<tr>
<td>Spice</td>
<td>58%</td>
<td>17%</td>
<td>16%</td>
<td>6%</td>
<td>3%</td>
</tr>
<tr>
<td>GCC</td>
<td>51%</td>
<td>39%</td>
<td>6%</td>
<td>1%</td>
<td>3%</td>
</tr>
</tbody>
</table>

- Similar measurements show that 16 bits is enough for the immediate field 75% to 80% of the time.
- And 16 bits is enough for displacement 99% of the time.
### MIPS addressing modes

#### register

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

- **add $1, $2, $3**
  - rs: $2
  - rt: $3
  - rd: $1

#### immediate

| OP  | rs  | rt   | immediate |
|-----|-----|------|           |

- **addi $1, $2, #35**
  - rs: $2
  - rt: $1

#### base + displacement

<table>
<thead>
<tr>
<th>OP</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
</table>

- **lw $1, 24($2)**
  - rs: $2
  - rt: $1

#### register indirect

- **disp = 0**
- **(rs) = 0**
MIPS ISA decisions

instruction length
  ▪ all instructions are 32 bits long.

how many registers?
  ▪ 32 general purpose registers (R0 always 0).

where do operands reside?
  ▪ load-store architecture.

instruction formats
  ▪ three (r-, i-, and j-format).

operands
  ▪ 3-address code.
  ▪ immediate, register, and base+displacement modes.
MIPS operations

- arithmetic
  - add, subtract, multiply, divide, ...

- logical
  - and, or, shift left, shift right, ...

- data transfer
  - load word, store word

- conditional branch
  - beq & bne are PC-relative, since most targets are nearby

- unconditional jump
  - jump, jump register, branch and link (we’ll study later)
Details in choosing operations

• How do you boot initial program?
  - Suppose an assembly language program wants to load a register. The only addressing mode is base+displacement. So you need to have a useful value in the base register. How do you get it there??

• How can you jump to an arbitrary memory location?
  - The “jump” instruction needs some of the 32 bits to say “jump”, so there are fewer than 32 bits left to specify which address.
Let’s revisit addressing mode ...

- VAX memory references distributed as follows:

<table>
<thead>
<tr>
<th>Program</th>
<th>Base + Displacement</th>
<th>Immediate</th>
<th>Scaled Index</th>
<th>Memory Indirect</th>
<th>All Others</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEX</td>
<td>56%</td>
<td>43%</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Spice</td>
<td>58%</td>
<td>17%</td>
<td>16%</td>
<td>6%</td>
<td>3%</td>
</tr>
<tr>
<td>GCC</td>
<td>51%</td>
<td>39%</td>
<td>6%</td>
<td>1%</td>
<td>3%</td>
</tr>
</tbody>
</table>

- MIPS decided to provide only B+D and Immed formats.
- Often, Scaled Index and Autoincrement instructions are used for stepping through elements of an array.
A[i] in a loop

• Base-displacement code:
  - Unoptimized:
    \[
    \text{\texttt{sll r3 r1 2}}
    \]
    \[
    \text{\texttt{add r3 r2 r3}}
    \]
    \[
    \text{\texttt{load r4 0(r3)}}
    \]
  - Optimized:

Assumptions:
  r1 is index i
  r2 holds address of A[0]
  use r3 for address of A[i]
  we want to load A[i] into r4

• Autoincrement code:

• VAX scaled index code:
PowerPC’s solution

• Limitation of autoincrement: what if i is increased by something other than one?

• “load update” – enhanced autoincrement
  - “ldu r4 16(r3)” increments contents of r3 by 16, then loads r4 with that memory location.
  - r3 is ready for next iteration of loop
MIPS vs PowerPC

• PowerPC has powerful instructions, e.g.
  - Load Update
  - Float Multiply-Add
  - Branch on Count

• Example - Dot product
  - a[i]*b[i]
  - Very important - core of Linpack benchmark, used for "Top500" list (see www.netlib.org)
1984 VAX re-implementation

- Digital team discovered:
  - 20% of VAX instructions were responsible for
  - 60% of microcode
  - 0.2% of executed instructions

- New design didn’t implement them in HW
  - “Illegal instruction” caused HW interrupt, then software would execute instructions.
  - Result (along with some other changes):
    - reduced amount of silicon by factor of 5
    - reduced performance by only 10%

- Moral: Smaller can be better!
Intel x86 evolution

- 1978: Intel 8086 announced (16 bit architecture)
- 1980: The 8087 floating point coprocessor added
- 1982: The 80286 - more ops, 24-bit address space
- 1985: The 80386 - 32-bit address space + new modes
- 1989-1995: The 80486, Pentium, and Pentium Pro add a few instructions
- 1997: MMX is added (Pentium II is P. Pro + MMX)
- 1999 Pentium III (same architecture)
- 2000 Pentium 4 (144 new multimedia instructions)
- 2001 Itanium - new ISA (still can execute x86 code)
Anthropology of ISA’s

• VAX design goal was small code size and simple compilation (since all combinations of addressing modes were possible).
  - Met goals, but cheap & fast memories and better compilers made goals less important.
  - But couldn’t pipeline well. Replaced by Alpha (a RISC).

• x86’s goal was to get to market quickly.
  - Ugly, but most common instructions can be implemented relatively efficiently.

• MIPS’ goal: simplicity. Reflects university heritage.

• PowerPC: super-RISC. Compiler group influence.
## 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></td>
<td></td>
<td>target</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### What if?
- 64 registers
- 20-bit immediates
- 4 operand instruction (e.g. \( Y = X + AB \))
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
Recent developments

• **VLIW** - Very Long Instruction Word
  - 1 “packet” has multiple instructions
  - Tera MTA has 26, 21, and 14 bit-long RISC operations (plus 3 “lookahead” bits) in 64 bits
  - Intel Itanium has three 41-bit RISC ops (plus 5 “type” bits) in 128-bit packet

• **JVM (Java Virtual Machine)**
  - a new level of abstraction
    - between Java language and ISA
  - stack based - is this a good choice??