Instruction Set Architectures

Dr. Soner Onder
CS 4431
Michigan Technological University
Instruction Set Architecture (ISA)

- 1950s to 1960s: Computer Architecture Course Computer Arithmetic
- 1970 to mid 1980s: Computer Architecture Course Instruction Set Design, especially ISA appropriate for compilers
- 1990s: Computer Architecture Course Design of CPU, memory system, I/O system, Multiprocessors
Instruction Set Architecture (ISA)
Interface Design

A good interface:

- Lasts through many implementations (portability, compatibility)
- Is used in many different ways (generality)
- Provides **convenient** functionality to higher levels
- Permits an **efficient** implementation at lower levels
Evolution of Instruction Sets

- Single Accumulator (EDSAC 1950)
- Accumulator + Index Registers (Manchester Mark I, IBM 700 series 1953)
- Separation of Programming Model from Implementation

  - High-level Language Based (B5000 1963)
  - Concept of a Family (IBM 360 1964)

General Purpose Register Machines

  - Complex Instruction Sets (Vax, Intel 432 1977-80)
  - Load/Store Architecture (CDC 6600, Cray 1 1963-76)

RISC (Mips, Sparc, 88000, IBM RS6000, . . . 1987)
Evolution of Instruction Sets

- Major advances in computer architecture are typically associated with landmark instruction set designs
  - Ex: Stack vs GPR (System 360)
- Design decisions must take into account:
  - technology
  - machine organization
  - programming languages
  - compiler technology
  - operating systems
- And they in turn influence these
Design Space of ISA

Five Primary Dimensions
- Number of explicit operands (0, 1, 2, 3)
- Operand Storage Where besides memory?
- Effective Address How is memory location specified?
- Type & Size of Operands byte, int, float, vector, . . .
  How is it specified?
- Operations add, sub, mul, . . .
  How is it specified?

Other Aspects
- Successor How is it specified?
- Conditions How are they determined?
- Encoding Fixed or variable? Wide?
- Parallelism
ISA Metrics

Aesthetics:

- Orthogonality
  - No special registers, few special cases, all operand modes available with any data type or instruction type

- Completeness
  - Support for a wide range of operations and target applications

- Regularity
  - No overloading for the meanings of instruction fields

- Streamlined
  - Resource needs easily determined

Ease of compilation (programming?)
Ease of implementation
Scalability
Basic ISA Classes

Accumulator:

1 address  add A acc ← acc + mem[A]
1+x address addx A acc ← acc + mem[A+x]

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
Stack Machines

- **Instruction set:**
  
  +, -, *, /, ...
  
  push A, pop A

- **Example:** a*b - (a+c*b)
  
  push a
  
  push b
  
  *
  
  push a
  
  push c
  
  push b
  
  *
  
  +
  
  -
Kinds of Addressing Modes

- Register direct: \( R_i \)
- Immediate (literal): \( v \)
- Direct (absolute): \( M[v] \)
- Register indirect: \( M[R_i] \)
- Base+Displacement: \( M[R_i + v] \)
- Base+Index: \( M[R_i + R_j] \)
- Scaled Index: \( M[R_i + R_j \times d + v] \)
- Autoincrement: \( M[R_i++] \)
- Autodecrement: \( M[R_i - -] \)
- Memory Indirect: \( M[M[R_i]] \)
- [Indirection Chains]
A "Typical" RISC

- 32-bit fixed format instruction (3 formats)
- 32 32-bit GPR (R0 contains zero, DP take pair)
- 3-address, reg-reg arithmetic instruction
- Single address mode for load/store:
  - base + displacement
    - no indirection
- Simple branch conditions
- Delayed branch

see: SPARC, MIPS, MC88100, AMD29000, i960, i860
PARisc, DEC Alpha, Clipper,
CDC 6600, CDC 7600, Cray-1, Cray-2, Cray-3
Operations that need an immediate operand
Distribution of data accesses by size for benchmark programs

- Double word (64 bits): 70% floating-point average, 59% integer average
- Word (32 bits): 29% floating-point average, 26% integer average
- Half word (16 bits): 0% floating-point average, 5% integer average
- Byte (8 bits): 1% floating-point average, 10% integer average
The bar chart shows the frequency of branch instructions for different types:

- **Call/return**: 8% (Floating-point average) and 19% (Integer average).
- **Jump**: 6% (Floating-point average) and 10% (Integer average).
- **Conditional branch**: 75% (Floating-point average) and 82% (Integer average).

The graph illustrates that conditional branches are the most frequent, followed by call/return and jump instructions.
**Variations of Instruction Encoding**

<table>
<thead>
<tr>
<th>Operation and no. of operands</th>
<th>Address specifier 1</th>
<th>Address field 1</th>
<th>⋮</th>
<th>Address specifier n</th>
<th>Address field n</th>
</tr>
</thead>
</table>

(a) Variable (e.g., Intel 80x86, VAX)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Address field 1</th>
<th>Address field 2</th>
<th>Address field 3</th>
</tr>
</thead>
</table>

(b) Fixed (e.g., Alpha, ARM, MIPS, PowerPC, SPARC, SuperH)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Address specifier</th>
<th>Address field</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Address specifier 1</th>
<th>Address specifier 2</th>
<th>Address field</th>
</tr>
</thead>
</table>

(c) Hybrid (e.g., IBM 360/370, MIPS16, Thumb, TI TMS320C54x)
State-of-the Art Compilers

**Dependencies**
- Language dependent; machine independent
- Somewhat language dependent; largely machine independent
- Small language dependencies; machine dependencies slight (e.g., register counts/types)
- Highly machine dependent; language independent

**Function**
- Transform language to common intermediate form
- For example, loop transformations and procedure inlining (also called procedure integration)
- Including global and local optimizations + register allocation
- Detailed instruction selection and machine-dependent optimizations; may include or be followed by assembler
Program, compiler optimization level

Percentage of unoptimized instructions executed

- Lucas, level 3: 11%
- Lucas, level 2: 12%
- Lucas, level 1: 21%
- Lucas, level 0: 100%
- Mcf, level 3: 76%
- Mcf, level 2: 76%
- Mcf, level 1: 84%
- Mcf, level 0: 100%

©2007 Elsevier, Inc. All rights reserved
Example: MIPS

Register-Register

\[
\begin{array}{cccccc}
31 & 26 & 25 & 21 & 20 & 16 \\
\hline
Op & Rs1 & Rs2 & Rd & \multicolumn{2}{c}{Op} \\
\end{array}
\]

Register-Immediate

\[
\begin{array}{cccccc}
31 & 26 & 25 & 21 & 20 & 16 \\
\hline
Op & Rs1 & Rd & \multicolumn{3}{c}{immediate} \\
\end{array}
\]

Branch

\[
\begin{array}{cccccc}
31 & 26 & 25 & 21 & 20 & 16 \\
\hline
Op & Rs1 & \multicolumn{2}{c}{Rs2/Op} & \multicolumn{2}{c}{immediate} \\
\end{array}
\]

Jump / Call

\[
\begin{array}{cccc}
31 & 26 & 25 & \hline
Op & \multicolumn{3}{c}{target} \\
\end{array}
\]
Overview of MIPS

- simple instructions all 32 bits wide
- very structured, no unnecessary baggage
- only three instruction formats

```
R | op | rs | rt | rd | shamt | funct |
---|-----|----|----|----|--------|--------|
I | op  | rs | rt |    | 16 bit address |
J | op  |    |    |    | 26 bit address |
```

- rely on compiler to achieve performance — what are the compiler's goals?
- help compiler where we can
Addresses in Branches and Jumps

- **Instructions:**
  
  bne $t4,$t5,Label  
  Next instruction is at Label if $t4 \neq $t5

  beq $t4,$t5,Label  
  Next instruction is at Label if $t4 = $t5

  j Label  
  Next instruction is at Label

- **Formats:**

  \[
  \begin{array}{c|c|c|c}
  \text{op} & \text{rs} & \text{rt} & \text{16 bit address} \\
  \hline
  \text{op} & & & \text{26 bit Address} \\
  \end{array}
  \]

  - Addresses are not 32 bits
    - How do we handle this with load and store instructions?
**Addresses in Branches**

- **Instructions:**
  
  - `bne $t4, $t5, Label`  
    Next instruction is at Label if $t4 ≠ $t5
  
  - `beq $t4, $t5, Label`  
    Next instruction is at Label if $t4 = $t5

- **Formats:**

  \[
  \begin{array}{cccc}
    & \text{op} & \text{rs} & \text{rt} & \text{16 bit address} \\
  \end{array}
  \]

- Could specify a register (like `lw` and `sw`) and add it to address
  
  - use Instruction Address Register (PC = program counter)
  
  - most branches are local (principle of locality)

- **Jump instructions just use high order bits of PC**
  
  - address boundaries of 256 MB
## Summary of MIPS

### MIPS operands

<table>
<thead>
<tr>
<th>Name</th>
<th>Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>32 registers</strong></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 $at is reserved for the assembler to handle large constants.</td>
</tr>
<tr>
<td></td>
<td>$a0-$a3, $v0-$v1, $gp,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$fp, $sp, $ra, $at</td>
<td></td>
</tr>
<tr>
<td><strong>2^{30} memory words</strong></td>
<td>Memory[0], Memory[4], ...</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>
</tr>
<tr>
<td></td>
<td>Memory[4294967292]</td>
<td></td>
</tr>
<tr>
<td>Category</td>
<td>Instruction</td>
<td>Example</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td><strong>Arithmetic</strong></td>
<td>add</td>
<td>add $s1, $s2, $s3</td>
</tr>
<tr>
<td></td>
<td>subtract</td>
<td>sub $s1, $s2, $s3</td>
</tr>
<tr>
<td></td>
<td>add immediate</td>
<td>addi $s1, $s2, 100</td>
</tr>
<tr>
<td><strong>Data transfer</strong></td>
<td>load word</td>
<td>lw $s1, 100($s2)</td>
</tr>
<tr>
<td></td>
<td>store word</td>
<td>sw $s1, 100($s2)</td>
</tr>
<tr>
<td></td>
<td>load byte</td>
<td>lb $s1, 100($s2)</td>
</tr>
<tr>
<td></td>
<td>store byte</td>
<td>sb $s1, 100($s2)</td>
</tr>
<tr>
<td></td>
<td>load upper immediate</td>
<td>lui $s1, 100</td>
</tr>
<tr>
<td><strong>Conditional</strong></td>
<td>branch on equal</td>
<td>beq $s1, $s2, 25</td>
</tr>
<tr>
<td></td>
<td>branch on not equal</td>
<td>bne $s1, $s2, 25</td>
</tr>
<tr>
<td></td>
<td>set on less than</td>
<td>slt $s1, $s2, $s3</td>
</tr>
<tr>
<td></td>
<td>set less than immediate</td>
<td>slti $s1, $s2, 100</td>
</tr>
<tr>
<td><strong>Uncondi</strong></td>
<td>jump</td>
<td>j 2500</td>
</tr>
<tr>
<td></td>
<td>jump register</td>
<td>jr $ra</td>
</tr>
</tbody>
</table>
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

- Sometimes referred to as “RISC vs. CISC”
  - 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!
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
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

“This history illustrates the impact of the “golden handcuffs” of compatibility

“adding new features as someone might add clothing to a packed bag”

“an architecture that is difficult to explain and impossible to love”
A dominant architecture: 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

“what the 80x86 lacks in style is made up in quantity, making it beautiful from the right perspective”
Tips for Helping the Compiler Writer

- Provide regularity
  - How does it affect the architecture?

- Provide primitives, not solutions
  - Why is it hard?

- Simplify tradeoffs among alternatives
  - How does this affect architecture?

- Provide instructions that bind the quantities known at compile time as constants.
  - How does it help with compiler/Hw interaction?