ECE 4680: Computer Architecture and Organization

Instruction Set Architecture

Different styles of ISA.
Basic issues when designing an ISA.
What a good ISA should be?

Instruction Set Design

- Operation Code: defines the operation type
- Operands: operation source and destination

Instruction Set Architecture

Programmer's View

ADD 01010
SUBTRACT 01110
AND 10011
OR 10001
COMPARE 11010

Computer's View

Princeton (Von Neumann) Architecture
- Data and Instructions mixed in same memory ("stored program computer")
- Program as data (dubious advantage)
- Single memory interface

Harvard Architecture
- Data & Instructions in separate memories
- Has advantages in certain high performance implementations

Basic Issues in Instruction Set Design

- What operations (and how many) should be provided
  - LD/ST/INC/BRN sufficient to encode any computation
  - But not useful because programs too long!

- How (and how many) operands are specified
  - Most operations are dyadic (eg, A < B + C)
  - Some are monadic (eg, A < ~B)

- How to encode these into consistent instruction formats
- Instructions should be multiples of basic data/address widths

Typical instruction set:

- 32 bit word
- Basic operand addresses are 32 bits long
- Basic operands, like integers, are 32 bits long
- In general case, instruction could reference 3 operands (A := B + C)

Challenge: encode operations in a small number of bits!

Execution Cycle

- 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

What Must be Specified?

- Instruction Format or Encoding
  - How is it decoded?
- Data type and Size
  - What are supported
- Location of operands and result – addressing mode
  - Where other than memory?
  - How many explicit operands?
  - How are memory operands located?
  - Which can or cannot be in memory?
- Operations
  - What are supported
- Successor instruction – flow control
  - Jumps, conditions, branches

- Fetch-decode-execute is implicit!
Topics to be covered

We will discuss the following topics which determine the Complexity of IS.

- Instruction Format or Encoding
  - how is it decoded?
- Data type and Size
  - what are supported
- Location of operands and result – addressing mode
  - where other than memory?
  - how many explicit operands?
  - how are memory operands located?
  - which can or cannot be in memory?
- Operations
  - what are supported
- Successor instruction – flow control
  - jumps, conditions, branches

Basic ISA Classes

<table>
<thead>
<tr>
<th>Accumulator: (earliest machines)</th>
<th>Data Type and Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 address add A acc ← acc + mem[A]</td>
<td>- what are supported</td>
</tr>
<tr>
<td>1× address addx A acc ← acc + mem[A × x]</td>
<td>- Location of operands and result – addressing mode</td>
</tr>
<tr>
<td>Stack: (HP calculator, Java virtual machines)</td>
<td>- where other than memory?</td>
</tr>
<tr>
<td>0 address add tos ← tos + next</td>
<td>- how many explicit operands?</td>
</tr>
</tbody>
</table>

General Purpose Register: (e.g. Intel 80x86, Motorola 68xxx)

| 2 address add A B EA(A) ← EA(A) + EA(B) | - how are memory operands located? |
| 3 address add A B C EA(A) ← EA(B) + EA(C) | - which can or cannot be in memory? |

Load/Store: (e.g. SPARC, MIPS, PowerPC)

| 3 address add Ra Rb Rc Ra ← Rb + Rc | - Operations |
| load Ra Rb Ra ← mem[Rb] | - what are supported |
| store Ra Rb mem[Rb] ← Ra | - Successor instruction – flow control |

Comparison:

- Bytes per instruction? Number of Instructions? Cycles per instruction?

Comparing Instructions

Comparing 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</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push A</td>
<td>Load A</td>
<td>Load R1,A</td>
<td>Load R1,A</td>
</tr>
<tr>
<td>Push B</td>
<td>Add B</td>
<td>Add R1,B</td>
<td>Load R2,B</td>
</tr>
<tr>
<td>Add C</td>
<td>Store C</td>
<td>Store C, R1</td>
<td>Add R3,R1,R2</td>
</tr>
<tr>
<td>Pop C</td>
<td>Store C</td>
<td>Store C,R3</td>
<td></td>
</tr>
</tbody>
</table>

General Purpose Registers Dominate

Since 1975 all machines use general purpose registers

- Java Virtual Machine adopts Stack architecture

Advantages of registers

- registers are faster than memory
- registers are easier for a compiler to use
  - e.g., (A*B) – (C*D) – (E*F) can do multiplies in any order vs. stack
- registers can hold variables
  - memory traffic is reduced, so program is sped up (since registers are faster than memory)
  - code density improves (since register named with fewer bits than memory location)

More About registers?

- integrated together in process chip.
- In top level in memory hierarchy.
- Faster to access and simpler to use.
- Special role in MIPS ISA: only registers not symbolic variables can be in instructions.
- The number of registers can not be too more, not be too less.
- Effective use of registers is a key to program performance.

Examples of Register Usage

Number of memory addresses per typical ALU instruction

- Maximum number of operands per typical ALU instruction

<table>
<thead>
<tr>
<th>1</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SPARC, MIPS, Precision Architecture, Power PC</td>
</tr>
<tr>
<td>1</td>
<td>Intel 80x86, Motorola 68000</td>
</tr>
<tr>
<td>2</td>
<td>VAX (also has 3-operand formats)</td>
</tr>
<tr>
<td>3</td>
<td>VAX (also has 2-operand formats)</td>
</tr>
</tbody>
</table>
Example:

In VAX: \( \text{ADDL (R9), (R10), (R11)} \)
\[ \text{mem[R9]} \leftarrow \text{mem[R10]} + \text{mem[R11]} \]

In MIPS: \( \text{lw R1, (R10); load a word} \)
\( \text{lw R2, (R11); R3 \leftarrow R1+R2} \)
\( \text{sw R3, (R9); store a word} \)

Pros and Cons of Number of Memory Operands/Operands

Register-register: 0 memory operands/instr, 3 (register) operands/instr

- Simple, fixed-length instruction encoding. Simple code generation model. Instructions take similar numbers of clocks to execute.
- Higher instruction count than architectures with memory references in instructions. Some instructions are short and bit encoding may be wasteful.

Register-memory (1,2)

- Data can be accessed without loading first. Instruction format tends to be easy to encode and yields good density.
- Operands are not equivalent since a source operand in a binary operation is destroyed. Encoding a register number and a memory address in each instruction may restrict the number of registers. Clocks per instruction varies by operand location.

Memory-memory (3,3)

- Most compact. Doesn’t waste registers for temporaries.
- Large variation in instruction size, especially for three-operand instructions. Also, large variation in work per instruction. Memory accesses create memory bottleneck.

Summary on Instruction Classes

- Expect new instruction set architecture to use general purpose registers
- Pipelining \( \Rightarrow \) Expect it to use load store variant of GPR ISA

Memory addressing

- BYTE Addressing:
  - Since 1980, almost every machine uses addresses to level of 8-bits
- Two Questions for design of ISA
  - For a 32-bit word, read it as four loads of bytes from sequential byte addresses or as one load work from a single byte address. How byte address map onto words?
  - Can a word be placed on any byte boundary?

Addressing Objects

Big Endian: address of most significant IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
Little Endian: address of least significant Intel 80x86, DEC Vax

Alignment: require that objects fall on address that is multiple of their size. (p 112)

Example 1: Memory layout of a number #ABCD

In Big Endian: \( \begin{array}{c} \text{CD} \\ \text{AB} \end{array} \) $1001 $1000

In Little Endian: \( \begin{array}{c} \text{AB} \\ \text{CD} \end{array} \) $1001 $1000

Example 2: Memory layout of a number #FF00

BIG Endian versus Little Endian (P113 & A-46)
Byte Swap Problem

<table>
<thead>
<tr>
<th>GH</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>EF</td>
<td>2</td>
</tr>
<tr>
<td>CG</td>
<td>1</td>
</tr>
<tr>
<td>AB</td>
<td>0</td>
</tr>
</tbody>
</table>

Increasing byte address

Big Endian

Little Endian

Memory layout of a number of ABCDEFGH

Each system is self-consistent, but causes problems when they need to communicate!

Addressing Modes

<table>
<thead>
<tr>
<th>Addressing mode</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>Add R4, R3</td>
<td>R4 ← R4 + 3</td>
</tr>
<tr>
<td>Register</td>
<td>Add R4, R3</td>
<td>R4 ← R4 + R3</td>
</tr>
<tr>
<td>Register indirect</td>
<td>Add R4, (R1)</td>
<td>R4 ← R4 + Mem[R1]</td>
</tr>
<tr>
<td>Displacement</td>
<td>Add R4, 100 + (R1)</td>
<td>R4 ← R4 + Mem[100 + R1]</td>
</tr>
<tr>
<td>Indexed</td>
<td>Add R3, (R1 + R2)</td>
<td>R3 ← R3 + Mem[R1 + R2]</td>
</tr>
<tr>
<td>Direct or absolute</td>
<td>Add R1, (1001)</td>
<td>R1 ← R1 + Mem[1001]</td>
</tr>
<tr>
<td>Memory indirect</td>
<td>Add R1, (R3)</td>
<td>R1 ← R1 + Mem[Mem[R3]]</td>
</tr>
<tr>
<td>Auto-increment</td>
<td>Add R1, (R2 + R3)</td>
<td>R1 ← R1 + Mem[R2 + R3]</td>
</tr>
<tr>
<td>Auto-decrement</td>
<td>Add R1, (R2 - R3)</td>
<td>R2 ← R2 - R1 + Mem[R2]</td>
</tr>
<tr>
<td>Scaled</td>
<td>Add R1, 100(R2)(R3)</td>
<td>R1 ← R1 + Mem[100 + R2 + R3]</td>
</tr>
</tbody>
</table>

Addressing Mode:

- Addressing modes have the ability to significantly reduce instruction counts
- They also add to the complexity of building a machine

Addressing Mode Usage

<table>
<thead>
<tr>
<th>Mode</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 programs avg.</td>
<td>17% to 43%</td>
</tr>
<tr>
<td>--- Register deferred</td>
<td>13% avg. 3% to 24%</td>
</tr>
<tr>
<td>--- Scaled</td>
<td>7% avg. 0% to 16%</td>
</tr>
<tr>
<td>--- Memory indirect</td>
<td>3% avg. 1% to 6%</td>
</tr>
<tr>
<td>--- Misc</td>
<td>2% avg. 0% to 3%</td>
</tr>
</tbody>
</table>

Displacement Address Size

- Average of 5 programs from SPECint92 and Average of 5 programs from SPECfp92
- X-axis is in powers of 2: <= addresses > 2^16 (8) and < 2^16 (16)
- 1% of addresses > 16-bits

Immediate Size

- 50% to 60% fit within 8 bits
- 75% to 80% fit within 16 bits
Addressing Summary

- Data Addressing modes that are important: Displacement, Immediate, Register Indirect
- Displacement size should be 12 to 16 bits
- Immediate size should be 8 to 16 bits

Typical Operations

- Data Movement: Load (from memory), Store (to memory), memory-to-memory move
- Register-to-register move
- Input (from I/O device), Output (to I/O device), Push, Pop (to/from stack)
- Arithmetic: integer (binary + decimal) or FP
- Add, Subtract, Multiply, Divide
- Logical: not, and, or, set, clear
- Shift: shift left/right, rotate left/right
- Control (Jump/Branch): unconditional, conditional
- Subroutine Linkage: call, return
- Interrupt: trap, return
- Synchronization: test & set (atomic r-m-w)
- String: search, translate

Top 10 80x86 Instructions

<table>
<thead>
<tr>
<th>Rank</th>
<th>Instruction</th>
<th>Integer</th>
<th>Average</th>
<th>Percent</th>
<th>Total executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>load</td>
<td></td>
<td></td>
<td></td>
<td>22%</td>
</tr>
<tr>
<td>2</td>
<td>conditional branch</td>
<td></td>
<td></td>
<td></td>
<td>20%</td>
</tr>
<tr>
<td>3</td>
<td>compare</td>
<td></td>
<td></td>
<td></td>
<td>16%</td>
</tr>
<tr>
<td>4</td>
<td>store</td>
<td></td>
<td></td>
<td></td>
<td>12%</td>
</tr>
<tr>
<td>5</td>
<td>add</td>
<td></td>
<td></td>
<td></td>
<td>8%</td>
</tr>
<tr>
<td>6</td>
<td>and</td>
<td></td>
<td></td>
<td></td>
<td>6%</td>
</tr>
<tr>
<td>7</td>
<td>sub</td>
<td></td>
<td></td>
<td></td>
<td>5%</td>
</tr>
<tr>
<td>8</td>
<td>move-register-register</td>
<td></td>
<td></td>
<td></td>
<td>4%</td>
</tr>
<tr>
<td>9</td>
<td>call</td>
<td></td>
<td></td>
<td></td>
<td>1%</td>
</tr>
<tr>
<td>10</td>
<td>return</td>
<td></td>
<td></td>
<td></td>
<td>1%</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td></td>
<td></td>
<td></td>
<td>96%</td>
</tr>
</tbody>
</table>

Methods of Testing Condition

- Condition Codes
  - Processor status bits are set as a side-effect of arithmetic instructions (possibly on Moves) or explicitly by compare or test instructions.
  - Example: `add r1, r2, r3, bz label`
- Condition Register
  - Example: `cmp r1, r2, r3; cmp r2 with r3, 0 or 1 is stored in r1; bgt r1, label; branch on greater`
- Compare and Branch
  - Example: `bgt r1, r2, label; if r1 > r2, then go to label`

Condition Codes

- Setting CC as side effect can reduce the # of instructions
- Example: `sub r0, #1, r0, BRP X
  cmp r0, #0, BRP X`
- But also has disadvantages:
  - Not all instructions set the condition codes which do and which do not often confusing!
    - E.g., shift instruction sets the carry bit
  - Dependency between the instruction that sets the CC and the one that tests it; to overlap their execution, may need to separate with an instruction that does not change the CC

Branches

- Conditional control transfers
- Four basic conditions: N -> negative, Z -> zero, V -> overflow, C -> carry
- Sixteen combinations of the basic four conditions:
  - Always
  - Unconditional
  - Never
  - NOP
  - Not Equal
  - Z
  - Equal
  - Z
  - Greater
  - [Z + (N * V)]
  - Greater or Equal
  - [Z + (N * V)]
  - Less
  - N * V
  - Less or Equal
  - [N * V]
  - Greater Unsigned
  - (C * Z)
  - Less or Equal Unsigned
  - C * Z
  - Carry Clear
  - C
  - Carry Set
  - C
  - Positive
  - -N
  - Negative
  - N
  - Overflow Clear
  - V
  - Overflow or Set
  - V

Methods of Testing Condition

- Condition Codes
  - Processor status bits are set as a side-effect of arithmetic instructions (possibly on Moves) or explicitly by compare or test instructions.
  - Example: `add r1, r2, r3, bz label`
- Condition Register
  - Example: `cmp r1, r2, r3; cmp r2 with r3, 0 or 1 is stored in r1; bgt r1, label; branch on greater`
- Compare and Branch
  - Example: `bgt r1, r2, label; if r1 > r2, then go to label`
Conditional Branch Distance

- Distance from branch in instructions $2^i \Rightarrow \delta \pm 2^i$
- 25% of integer branches are $> 2^i$

Conditional Branch Addressing
- PC-relative since most branches at least 8 bits suggested ($\pm 128$ instructions)
- Compare Equal/Not Equal most important for integer programs

Operation Summary
- Support these simple instructions, since they will dominate the number of instructions executed:
  - load, store, add, subtract, move register-register, and, shift, compare equal, compare not equal, branch (with a PC-relative address at least 8-bits long), jump, call, return;

Data Types
- Bit: 0, 1
- Bit String: sequence of bits of a particular length
  - 4 bits is a nibble
  - 8 bits is a byte
  - 16 bits is a half-word
  - 32 bits is a word
- Character:
  - ASCII 7 bit code
  - EBCDIC 8 bit code (IBM)
  - UNICODE 16 bit code (Java)
- Decimal:
  - digits 0-9 encoded as 0000b thru 1001b
  - two decimal digits packed per 8 bit byte
- Integers:
  - Sign & Magnitude: 0X vs. 1X
  - 1’s Complement: 0X vs. 1(‘s comp) + 1
  - 2’s Complement: 0X vs. 1(‘s comp) + 1
- Floating Point:
  - Single Precision
  - Double Precision
  - Extended Precision
  - Positive #’s same in all First 2 have two zeros
  - Last one usually chosen
  - First 2 have two zeros
  - Last one usually chosen
  - Where is decimal pt?
  - How are +/- exponents represented?

Operand Size Usage
- Support these data sizes and types:
  - 8-bit, 16-bit, 32-bit integers and
  - 32-bit and 64-bit IEEE 754 floating point numbers

Instruction Format
- If have many memory operands per instructions and many addressing modes, need an Address Specifier per operand
- If have load-store machine with 1 address per instr. and one or two addressing modes, then just encode addressing mode in the opcode
Generic Examples of Instruction Formats

Variable:

Fixed:

Hybrid:

Summary of Instruction Formats

- If code size is most important, use variable length instructions
- If performance is most important, use fixed length instructions

Instruction Set Metrics

Design-time metrics:
- Can it be implemented, in how long, at what cost?
- Can it be programmed? Ease of compilation?

Static Metrics:
- How many bytes does the program occupy in memory?

Dynamic Metrics:
- How many instructions are executed?
- How many bytes does the processor fetch to execute the program?
- How many clocks are required per instruction?
- How "lean" a clock is practical?

Best Metric: Time to execute the program!

NOTE: this depends on instructions set, processor organization, and compilation techniques.

Lecture Summary: ISA

- Use general purpose registers with a load-store architecture;
- Support these addressing modes: displacement (with an address offset size of 12 to 16 bits), immediate (size 8 to 16 bits), and register deferred;
- Support these simple instructions, since they will dominate the number of instructions executed: load, store, add, subtract, move register-register, and, shift, compare equal, compare not equal, branch (with a PC-relative address at least 8-bits long), jump, call, and return;
- Support these data sizes and types: 8-bit, 16-bit, 32-bit integers and 64-bit IEEE 754 floating point numbers;
- Use fixed instruction encoding if interested in performance and use variable instruction encoding if interested in code size;
- Provide at least 16 general purpose registers plus separate floating-point registers, be sure all addressing modes apply to all data transfer instructions, and aim for a minimalist instruction set.