Instruction Set Design

An instruction is a binary code, which specifies a basic operation (e.g. add, subtract, and, or) for the computer
- 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)
--- Storage utilization
--- 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 (e.g., A <- B + C)
    Some are monadic (e.g., 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**

- **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

**What Must be Specified?**

- **Instruction Fetch**
  - Instruction Format or Encoding
    - how is it decoded?
  - Location of operands and result
    - where other than memory?
    - how many explicit operands?
    - how are memory operands located?
    - which can or cannot be in memory?
- **Operand Fetch**
  - Data type and Size
- **Execute**
  - Operations
    - what are supported
  - Successor instruction
    - jumps, conditions, branches
- **Result Store**
- **Next Instruction**
  - fetch-decode-execute is implicit!
Topics to be covered

- Location of operands and result
  - where other than memory?
  - how many explicit operands?
  - how are memory operands located?
  - which can or cannot be in memory?
- Operations
- Instruction Format or Encoding
  - how is it decoded?
- Data type and Size
  - what are supported

Basic ISA Classes

Accumulator: (earliest machines)

1 address
add A
acc ← acc + mem[A]

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

Stack: (HP calculator, Java virtual machines)

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

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></td>
<td>(register-memory)</td>
<td>(load-store)</td>
<td></td>
</tr>
<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</td>
<td>Store C</td>
<td>Store C, R1</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>

• 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)
Examples of Register Usage

Number of memory addresses per typical ALU instruction

Maximum number of operands per typical ALU instruction

Examples

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

Example:

In VAX:

```
ADDL (R9), (R10), (R11)  
```

In MIPS:

```
lw  R1, (R10);    load a word
lw  R2, (R11)     
add R3, R1, R2;  R3 <- R1+R2
sw  R3, (R9);     store a word
```
### Pros and Cons of Number 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 vary 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 register
- Pipelining => 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.
BIG Endian versus Little Endian

Example 1: Memory layout of a number #ABCD

In Big Endian:  CD → AB

In Little Endian:  AB → CD

Example 2: Memory layout of a number #FF00

Byte Swap Problem

When words are transferred the data

Each system is self-consistent, but causes problems when they need communicate!
### Addressing Modes

<table>
<thead>
<tr>
<th>Addressing mode</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>Add R4,R3</td>
<td>R4 ← R4 + R3</td>
</tr>
<tr>
<td>Immediate</td>
<td>Add R4,#3</td>
<td>R4 ← R4 + d</td>
</tr>
<tr>
<td>Displacement</td>
<td>Add R4,100(R1)</td>
<td>R4 ← R4 + Mem[100 + R1]</td>
</tr>
<tr>
<td>Register indirect</td>
<td>Add R4,(R1)</td>
<td>R4 ← R4 + Mem[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)+</td>
<td>R1 ← R1 + Mem[R2]; R2 ← R2 + d</td>
</tr>
<tr>
<td>Auto-decrement</td>
<td>Add R1,–(R2)</td>
<td>R2 ← R2 − d; R1 ← R1 + Mem[R2]</td>
</tr>
<tr>
<td>Scaled</td>
<td>Add R1,100(R2)[R3]</td>
<td>R1 ← R1 + Mem[100 + R2 + R3 * d]</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

3 programs avg, 17% to 43%

--- Register deferred (indirect): 13% avg, 3% to 24%
--- Scaled: 7% avg, 0% to 16%
--- Memory indirect: 3% avg, 1% to 6%
--- Misc: 2% avg, 0% to 3%

### Displacement Address Size

- Average of 5 programs from SPECint92 and Average of 5 programs from SPECfp92
- X-axis is in powers of 2: 4 => addresses > 2^3 (8) and 8 => 2^4 (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 Average</th>
<th>Percent total executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>load</td>
<td>22%</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>conditional branch</td>
<td>20%</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>compare</td>
<td>16%</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>store</td>
<td>12%</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>add</td>
<td>8%</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>and</td>
<td>6%</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>sub</td>
<td>5%</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>move register-register</td>
<td>4%</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>call</td>
<td>1%</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>return</td>
<td>1%</td>
<td></td>
</tr>
</tbody>
</table>

Total: 96%

*Simple instructions dominate instruction frequency*
### 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.

ex: `add r1, r2, r3
bz label`

° **Condition Register**

Ex: `cmp r1, r2, r3;` compare r2 with r3, 0 or 1 is stored in r1

`bgt r1, label;` branch on greater

° **Compare and Branch**

Ex: `bgt r1, r2, label;` if r1 > r2, then go to label

---

### Condition Codes

Setting CC as side effect can reduce the # of instructions

<table>
<thead>
<tr>
<th>X: . . . . . . . . . . .</th>
<th>X: . . . . . . . . . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUB r0, #1, r0</td>
<td>SUB r0, #1, r0</td>
</tr>
<tr>
<td>BRP X</td>
<td>CMP r0, #0</td>
</tr>
<tr>
<td></td>
<td>BRP X</td>
</tr>
</tbody>
</table>

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 them with an instruction that does not change the CC

<table>
<thead>
<tr>
<th>ifetch</th>
<th>read</th>
<th>compute</th>
<th>write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Old CC read</td>
<td>New CC computed</td>
<td>ifetch</td>
<td>read</td>
</tr>
</tbody>
</table>
### Branches

--- Conditional control transfers

**Four basic conditions:**
- N -- negative
- Z -- zero
- V -- overflow
- C -- carry

Sixteen combinations of the basic four conditions:
- Always
- Never
- Not Equal
- Equal
- Greater
- Less or Equal
- Greater or Equal
- Less
- Greater Unsigned
- Less or Equal Unsigned
- Carry Clear
- Carry Set
- Positive
- Negative
- Overflow Clear
- Overflow Set

---

### Conditional Branch Distance

![Graph showing branch distance distribution]

- Distance from branch in instructions $2^i \Rightarrow \hat{S} \pm 2^{i-1} \& > 2^{i-2}$
- 25% of integer branches are $> 2 \& \hat{S} \pm 4$ or $-2 \& -4$
Conditional Branch Addressing

- PC-relative since most branches at least 8 bits suggested (± 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 (VAX: word)
32 bits is a word (VAX: long word)

Character:
ASCII 7 bit code
EBCDIC 8 bit code

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(~X)
2’s Complement: 0X vs. (1’s comp) + 1

Floating Point:
Single Precision
Double Precision
Extended Precision

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

<table>
<thead>
<tr>
<th>Variable:</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hybrid:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Summary of Instruction Formats

• If code size is most important, use variable length instructions

• If performance is over is most important, use fixed length instructions

Compilers and Instruction Set Architectures

• Ease of compilation
  ° 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

• Register Assignment is critical too
  • Keep args and local variables in registers
  • Works well if at least 16 registers
Summary of Compiler Considerations

• Provide at least 16 general purpose registers plus separate floating-point registers,

• Be sure all addressing modes apply to all data transfer instructions,

• Aim for a minimalist instruction set.

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.