Instruction Set Design

- One goal of instruction set design is to minimize instruction length
- Many instructions were designed with compilers in mind.
- Determining how operands are addressed is a key component of instruction set design
Instruction Format

• Defines the layout of bits in an instruction
• Includes opcode and includes implicit or explicit operand(s)
• Usually there are several instruction formats in an instruction set
• Huge variety of instruction formats have been designed; they vary widely from processor to processor
Instruction Length

• The most basic issue
• Affected by and affects:
  – Memory size
  – Memory organization
  – Bus structure
  – CPU complexity
  – CPU speed
• Trade off between a powerful instruction repertoire and saving space with shorter instructions
Instruction format trade-offs

- Large instruction set => small programs
- Small instruction set => large programs
- Large memory => longer instructions
- Fixed length instructions same size or multiple of bus width => fast fetch
- Variable length instructions may need extra bus cycles
Instruction format trade-offs

• Processor may execute faster than fetch
  – Use cache memory or use shorter instructions
• Note complex relationship between word size, character size, instruction size and bus transfer width
  – In almost all modern computers these are all multiples of 8 and related to each other by powers of 2
Allocation of bits

Determines several important factors

• Number of addressing modes
  – Implicit operands don’t need bits
  – X86 uses 2-bit mode field to specify Interpretation of 3-bit operand fields

• Number of operands
  – 3 operand formats are rare
  – For two operand instructions we can use one or two operand mode indicators
  – X86 uses only one 2-bit indicator
Allocation of bits

Determines several important factors

• Register versus memory
  – Tradeoff between # of registers and program size
  – Studies suggest optimal number between 8 and 32
  – Most newer architectures have 32 or more
  – X86 architecture allows some computation in memory
Allocation of bits

Determines several important factors (cont'd)

• Number of register sets
  – RISC architectures tend to have larger sets of uniform registers
  – Small register sets require fewer opcode bits
  – Specialized register sets can reduce opcode bits further by implicit reference (address vs. data registers)
Allocation of bits

Determines several important factors (cont’d)

• Address range
  – Large address space requires large instructions for direct addressing
  – Many architectures have some restricted or short forms of displacement addressing
    Ex: x86 short jumps and loops,
    PowerPC 16-bit displacement addressing

• Address granularity
  – Size of object addressed
  – Typically 8, 16, 32 and 64 instruction variants
Addressing Modes

• For a given instruction set architecture, addressing modes define how machine language instructions identify the operand (or operands) of each instruction.

• An addressing mode specifies how to calculate the effective memory address of an operand by using information held in registers and/or constants contained within a machine instruction or elsewhere.

• Different types of addresses involve tradeoffs between instruction length, addressing flexibility, and complexity of address calculation.
Addressing Modes

Common addressing modes

– Immediate
– Direct
– Indirect
– Register
– Register indirect
– Displacement
– Implied (stack)
Immediate Addressing

- the instruction itself contains the value to be used; located in the address field of the instruction; operand = address field.
- e.g. add_immediate(5)
  - add 5 to the contents of the accumulator
  - 5 is the operand
- the value is stored in memory immediately after the instruction opcode in memory
- Similar to using a constant in a high level language
Immediate Addressing

• Advantage
  – fast, since the value is included in the instruction; no memory reference to fetch data

• Disadvantage
  – not flexible, since the value is fixed at compile-time
  – can have limited range in machines with fixed length instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>operand</td>
</tr>
</tbody>
</table>
## Immediate Addressing

For the following example, assume an accumulator machine structure and that an `add_immediate` instruction is stored in memory, beginning at location 12.

<table>
<thead>
<tr>
<th>Assembly Lang</th>
<th>Addr</th>
<th>Contents</th>
<th>Hardware Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>...</code></td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td><code>add_immediate(5)</code></td>
<td>12</td>
<td>41</td>
<td><code>acc ← acc + 5</code></td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td><code>...</code></td>
<td></td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Since an `add` must have different hardware actions than an `add_immediate`, `add_immediate` has to be a different opcode (or there has to be an extra type-of-addressing-mode code in the instruction format to go along with the opcode).
Direct Addressing

- The instruction tells where the value can be found, but the value itself is out in memory.
- The address field contains the address of the operand
- Effective address (EA) = address field (A)
- e.g. add(A)
  - Add contents of memory address A to the accumulator
  - Look in memory at address A for operand
- In a high level language, direct addressing is frequently used for things like global variables.
Direct Addressing

• Advantage
  – Single memory reference to access data
  – No additional calculations to determine effective address
  – More flexible than immediate

• Disadvantage
  – Limited address space
Direct Addressing

Instruction

| Opcode | Address A |

Memory

Operand
Direct Addressing

for the following example, assume an accumulator machine structure and that an add instruction is stored in memory, beginning at location 12

\[
\begin{array}{cccc}
\text{assembly lang} & \text{addr} & \text{contents} & \text{hardware actions} \\
\hline
\text{add(one)} & 12 & | 40 | & \text{acc} \leftarrow \text{acc} + \text{memory}[24] \\
 & 13 & | 24 | & = \text{acc} + 1 \\
\text{word(one,1)} & 24 & | 1 | & \text{effective address} = 24 \\
\end{array}
\]

so, when the PC points to 12:

- 40 (i.e., the contents of location 12) is interpreted as an opcode
- 24 (i.e., the contents of location 13) is interpreted as an address
- 1 (i.e., the contents of location 24) is interpreted as data

note that there are no tags or other indicators that the number 40 in location 12 has to be an opcode; it could just as well be used as an address or as data
Example of Immediate and Direct Addressing Modes

Suppose we have a statement in C like

\[ b = a + 10; \]

\(a\) and \(b\) are variables, so they are out in memory.

To execute this statement, we will need to fetch \(a\) from memory, and write our result to \(b\).

That means the instructions we generate need to have the addresses of \(a\) and \(b\), and need to read and write those addresses as appropriate.

The number 10 is an actual value appearing in the statement. So, our code needs to include 10 itself.
Memory-Indirect Addressing

• The memory cell pointed to by the address field contains the address of (pointer to) the operand

• \( EA = (A) \)
  – Look in A, find address in A and look there for the operand

• e.g. \( \text{add\_indirect}(A) \)
  – add contents of memory cell pointed to by contents of A to accumulator
Memory-Indirect Addressing

Advantages:
- Large address space
  \[ 2^n \text{ where } n = \text{word length} \]
- May be nested, multilevel, cascaded

Disadvantage
- Multiple memory accesses; hence, slower
Indirect Addressing

Instruction

Opcode | Address A

Memory

Pointer to operand

Operand
## Indirect Addressing

For the following example, assume an accumulator machine structure and that an add instruction is stored in memory, beginning at location 12.

<table>
<thead>
<tr>
<th>assembly lang</th>
<th>memory addr</th>
<th>contents</th>
<th>hardware actions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>------------</td>
<td>------------</td>
<td>-------------------------------------------</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>add indirect(ptr)</td>
<td>12</td>
<td>42</td>
<td>acc ← acc + memory[memory[memory[36]]]</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>36</td>
<td>= acc + memory[24]</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>word(one,1)</td>
<td>24</td>
<td>1</td>
<td>= acc + 1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>word(ptr,one)</td>
<td>36</td>
<td>24</td>
<td>effective address = 24</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

The address included in the instruction is that of a pointer, that is, a word that holds another address.
Register Addressing

- Operand is held in register named in address field
- \( EA = R \)

Advantages

- No memory accesses
- Very fast execution
- Very small address field needed
  - Shorter instructions
  - Faster instruction fetch
Register Addressing

Advantages (cont'd)

• Multiple registers improve performance

Note: in C you can specify register variables

    register int a;
    
    – This is only advisory to the compiler; no guarantees

Disadvantage

    – Limited number of registers
Register Addressing

| Opcode | Register Address R |

Instruction

Registers

Operand
Register Indirect Addressing

- Similar to indirect addressing
- \( EA = (R) \)
- Operand is in memory cell pointed to by contents of register \( R \)
- Large address space \( (2^n) \)
- One fewer memory access than indirect addressing
Register Indirect Addressing Diagram

Instruction

- Opcode
- Register Address R

Registers

- Pointer to Operand

Memory

Operand
Displacement Addressing

• $EA = A + (R)$
• Address field hold two values
  – $A = \text{base value}$
  – $R = \text{register that holds displacement}$
  – or vice versa
Displacement Addressing
Relative Addressing

• A version of displacement addressing
• R = Program counter, PC
• EA = A + (PC)
• i.e. get operand from A cells from current location pointed to by PC
Base-Register Addressing

- A holds displacement
- R holds pointer to base address
- R may be explicit or implicit
Indexed Addressing

• A = base
• R = displacement
• EA = A + R
• Good for accessing arrays
  – EA = A + R
  – R++
Combinations

- Postindex
  - $EA = (A) + (R)$

- Preindex
  - $EA = (A+(R))$
Stack Addressing

- Operand is (implicitly) on top of stack
- e.g.
  - ADD Pop top two items from stack and add