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
- 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
- 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

- 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)

- 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.

Common addressing modes
- Direct
- Immediate
- Indirect
- Register
- Register indirect
- Displacement
- Implied (stack)
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)
• In a high level language, direct addressing is frequently used for things like global variables.
• Advantage
  – Single memory reference to access data
  – More flexible than immediate.

![Diagram of Direct Addressing](image-url)
**Direct Addressing**

for the following examples, 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>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(one)</td>
<td>12</td>
<td>40</td>
<td>acc &lt;- acc + memory[24]</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>24</td>
<td>= acc + 1</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>effective address = 24</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

so, when the PC points to 12:

40 (that is, the contents of location 12) is interpreted as an opcode

24 (that is, the contents of location 13) is interpreted as an address

1 (that is, 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
Immediate Addressing

• the instruction itself contains the value to be used; located in the address field of the instruction
• the value is stored in memory immediately after the instruction opcode in memory
• Similar to using a constant in a high level language
• 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 translation-time
  – can have limited range in machines with fixed length instructions

Instruction

|        | operand |
### Immediate 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>addr</th>
<th>contents</th>
<th>hardware actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>add Immediate(1)</td>
<td>12</td>
<td></td>
<td>41</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>...</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)
Example of direct and immediate addressing
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

- \( EA = (A) \)
**Indirect Addressing**

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

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Contents</th>
<th>Hardware Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>42</td>
<td>( \text{acc} \leftarrow \text{acc} + \text{memory}[\text{memory}[36]] )</td>
</tr>
<tr>
<td>13</td>
<td>36</td>
<td>( \text{acc} = \text{acc} + \text{memory}[24] )</td>
</tr>
<tr>
<td>24</td>
<td>1</td>
<td>( \text{effective address} = 24 )</td>
</tr>
<tr>
<td>36</td>
<td>24</td>
<td>( \text{acc} = \text{acc} + 1 )</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(s) is (are) registers
- EA = R
  - Register R is EA (not contents of R)
Register Addressing

• There is a limited number of registers
  – A very small address field is needed
  – Shorter instructions
  – Faster instruction fetch
  – X86: 3 bits used to specify one of 8 registers

• No memory access needed to fetch EA
• Very fast execution
• Very limited address space
• Multiple registers can help performance
• Requires good assembly programming or compiler writing

  Note: in C you can specify register variables
  register int a;
  – This is only advisory to the compiler; no guarantees
Register-Indirect Addressing

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

- Combines register-indirect addressing and direct addressing
- \( EA = A + (R) \)
- Address field holds two values
  - \( A = \) base value
  - \( R = \) register that holds displacement
  - Or visa versa
Types of Displacement Addressing

- Relative Addressing
- Base-register addressing
- Indexing

Relative Addressing

- \[ EA = A + (PC) \]
- Address field A is treated as 2’s complement integer to allow backward references
- Fetch operand from PC+A
- Can be very efficient because of locality of reference & cache usage
  - But in large programs code and data may be widely separated in memory
Base-Register Addressing
• A holds displacement
• R holds pointer to base address
• R may be explicit or implicit
  – E.g. segment registers in 80x86 are base registers and are involved in all EA computations
  – X86 processors have a wide variety of base addressing
Indexed Addressing

- \( A = \text{Base} \)
- \( R = \text{displacement} \)
- \( EA = A + R \)
- Good for accessing arrays
  - \( EA = A + R \)
  - \( R++ \)
- Iterative access to sequential memory locations is very common
- Some architectures provide auto-increment or auto-decrement
- Preindex \( EA = A + (R++) \)
- Postindex \( EA = A + (++R) \)
## Indexed Addressing

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

<table>
<thead>
<tr>
<th>memory assembly lang.</th>
<th>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_indexed(b0,x)</td>
<td>12</td>
<td>43</td>
<td>acc &lt;- acc + memory[20+memory[36]]</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>20</td>
<td>= acc + memory[20+4]</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>36</td>
<td>= acc + memory[24]</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
<td>= acc + 1</td>
</tr>
<tr>
<td>word(b0,5)</td>
<td>20</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>word(b1,-2)</td>
<td>21</td>
<td>-2</td>
<td>effective address = 24</td>
</tr>
<tr>
<td>word(b2,3)</td>
<td>22</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>word(b3,9)</td>
<td>23</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>word(b4,1)</td>
<td>24</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>word(x,4)</td>
<td>36</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Addressing modes using registers**

On machines with multiple registers, addresses and index values can be held in registers, for example:

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>Machine Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct</td>
<td><code>load(x,r1)</code></td>
<td><code>r1 &lt;- memory[x]</code></td>
</tr>
<tr>
<td>Immediate</td>
<td><code>load_imm(3,r2)</code></td>
<td><code>r2 &lt;- 3</code></td>
</tr>
<tr>
<td>Indexed for array access (fixed array base address and index in a register)</td>
<td><code>load_ind(a,r3,r4)</code></td>
<td><code>r4 &lt;- memory[a + r3]</code></td>
</tr>
<tr>
<td>Register indirect as part of indexed (i.e., a pointer is in a register)</td>
<td><code>load_ind(0,r5,r6)</code></td>
<td><code>r6 &lt;- memory[0 + r5]</code></td>
</tr>
<tr>
<td>Base plus displacement as part of indexed (i.e., structure access w/ ptr. in reg. and constant offset)</td>
<td><code>load_ind(2,r7,r8)</code></td>
<td><code>r8 &lt;- memory[2 + r7]</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>// accesses 3rd word of // a structure</code></td>
</tr>
</tbody>
</table>
## Branch addressing modes

Direct addressing, such as the accumulator machine

<table>
<thead>
<tr>
<th>Assembly lang</th>
<th>addr</th>
<th>contents</th>
<th>hardware actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>ba(target)</td>
<td>20</td>
<td>70</td>
<td>pc &lt;- 30</td>
</tr>
<tr>
<td></td>
<td>21</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>label(target)</td>
<td>30</td>
<td></td>
<td>next instruction after branch</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
**Branch addressing modes**

pc-relative addressing, such as the JVM Memory

<table>
<thead>
<tr>
<th>Assembly lang</th>
<th>addr</th>
<th>contents</th>
<th>hardware actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>goto(target)</td>
<td>20</td>
<td>167</td>
<td>pc &lt;- 21 + 9</td>
</tr>
<tr>
<td></td>
<td>21</td>
<td>9</td>
<td>= 30</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>label(target)</td>
<td>30</td>
<td></td>
<td>next instruction after goto</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

note that other machines may make the offset relative to the address of the branch (e.g., 20 above) or the fully-updated pc (e.g., 22 above)
Stack Addressing

- Operand is implicitly on top of stack
  - PUSH
  - POP
Alternate Addressing

• Offset addressing – offset is added or subtracted from value in base register

• Preindex addressing
  – Memory address is formed the same way as offset addressing, but the memory address is written back to the base register after adding or subtracting the displacement
  – The writeback occurs before the store to memory

• Postindex addressing
  – Similar to preindex addressing, but the writeback of the effective address occurs after the store to memory