Instruction Set Architecture Level
- defines the set of machine instructions
- interface between software and hardware
- high-level language programs are compiled into ISA
- assembly language programs are assembled into ISA

Instruction Set Architecture Level Diagram

Primary Memory
- linear sequence of addressable cells
- cell: smallest addressable unit, e.g. 8 bits (1 byte)
- size of address ⇒ memory size (n bits: $2^n$ cells)

Primary Memory Diagram

Instructions
- opcode and 0 or more operands

Instructions Table

Instructions
- given instruction length, design instruction set
- trade-offs: address size vs memory size vs resolution
  1.  
  2.  
  3.  

Instructions
- 1-2: given mem size, trade address size for resolution
- 1-3: given resolution, trade address size for mem size
- 2-3: given address size, trade mem size for resolution
Instructions

- another trade-off: opcode size vs #/type of operands
  - large opcode ⇒ many instructions but less space for operands
- observation: space available for opcode varies:
  - 0-operator instruction can use entire space
  - register-only operands also need little space
  - memory addresses are the largest consumers

⇒ expanding opcodes

Expanding Opcodes

- with k bits: $2^k$ different opcodes
- idea:
  - implement only $2^k-1$ opcodes
  - use last opcode ($111 \ldots 1$) to indicate expansion:
    - subsequent m bits are part of opcode ⇒ $2^m$ additional
    - total: $2^k-1+2^m$ opcodes, some need k bits, some k+m bits

Example:

- $k=3$, $m=2$
- $2^3=8$, $2^2=4$

Error-Detecting/Correcting Codes

- memory, CPU, comm. lines all make mistakes
  - need to detect and/or correct
- key concept: Hamming distance
  - number of bits in which 2 code words differ
  - Ex: 0010, 0111 have distance $d=2$
- principle of error detection:
  - assume any two codes $C_j$, $C_k$ have a distance $d$
  - $d$ bit flips to arrive from $C_j$ to $C_k$
  - need $d$ bit flips to arrive from $C_j$ to $C_k$
- rule: to detect $d$ errors, need distance of $d+1$
Error-Detecting Codes

- **Example 1**: encode 0, 1, 2, 3 as follows
  - 0 = 000
  - 1 = 011
  - 2 = 101
  - 3 = 110
  - all d=2
- consider single flip, e.g.: 000 → 100 or 010 or 001
- all 3 are illegal code words ⇒ error

- **Example 2**: parity: add one bit to original code such that # of 1's is always even (odd)
  - 0010 ⇒ error
  - 1000 ⇒ error
  - 0011 ⇒ not detected (valid code)

Error-Correcting Codes

- **Rule**: to correct d errors, need distance of 2d+1
- **Hamming Code**
  - best known error-correcting code
  - **Principles**:
    - insert parity bits at positions 2i, i.e., 1, 2, 4, 8, ... such that:
      1. each parity bit checks the next i bits (including itself), then skips next i bits, then checks the next i bits, etc:
        - bit 1: 1 (2)
        - bit 2: 2 (3 4)
        - bit 3: 1+2 (4 5)
        - bit 4: 4 (5 6)
        - bit 5: 1+4 (6 7)
        - bit 6: 2+4 (7 8)
        - bit 7: 1+2+4 (8 9)
      2. each bit b is checked by those parity bits that add up to b:
        - bit 1: 1
        - bit 2: 2
        - bit 3: 1+2
        - bit 4: 4
        - bit 5: 1+4
        - bit 6: 2+4
        - bit 7: 1+2+4
      3. Example: 4 data bits plus 3 parity bits
        - correction:
          - determine which parity bits are wrong
          - add up all incorrect parity bit positions ⇒ the sum is the incorrect bit
        - Examples:
          - 0010000: b1, b2 wrong ⇒ error in bit 3
          - 0000100: b1, b4 wrong ⇒ error in bit 5
          - 0100000: b2 wrong ⇒ error in bit 2
          - 0000001: b1, b2, b4 wrong ⇒ error in bit 7
          - 1101111: b1, b2 wrong ⇒ error in bit 3

### Error-Correcting Codes

<table>
<thead>
<tr>
<th>parity code</th>
<th>b1 b2 b3 b4 b5 b6 b7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>1 1 0 1 0 1 0</td>
<td>1 1 1 1 0 1 0</td>
</tr>
<tr>
<td>2 2 0 0 1 0 0</td>
<td>2 0 1 0 1 1 0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>15 1 1 1 1 1 1</td>
<td>15 1 1 1 1 1 1</td>
</tr>
</tbody>
</table>