What is an instruction set?

- The complete collection of instructions that are understood by a CPU
- Machine Code
- Binary
- Usually represented by assembly codes

Elements of an Instruction
- Operation code (Op code)
  - Do this
- Source Operand reference
  - To this
- Result Operand reference
  - Put the answer here
- Next Instruction Reference
  - When you have done that, do this...
**Instruction Representation**

- In machine code each instruction has a unique bit pattern
- For human consumption (well, programmers anyway) a symbolic representation is used
  - e.g. ADD, SUB, LOAD
- Operands can also be represented in this way
  - ADD A,B

**Simple Instruction Format**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Operand Reference</th>
<th>Operand Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 bits</td>
<td>6 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

**Instruction Types**

- Data processing
- Data storage (main memory)
- Data movement (I/O)
- Program flow control

**Number of Addresses**

- 3 addresses
  - Operand 1, Operand 2, Result
  - a = b + c;
  - May be a forth - next instruction (usually implicit)
  - Not common
  - Needs very long words to hold everything
- 2 addresses
  - One address doubles as operand and result
  - a = a + b
  - Reduces length of instruction
  - Requires some extra work
    - Temporary storage to hold some results
- 1 address
  - Implicit second address
  - Usually a register (accumulator)
  - Common on early machines
- 0 (zero) addresses
  - All addresses implicit
  - Uses a stack
    - e.g. push a
    - push b
    - add
    - pop c
  - c = a + b
How Many Addresses

- More addresses
  - More complex (powerful?) instructions
  - More registers
    - Inter-register operations are quicker
  - Fewer instructions per program
- Fewer addresses
  - Less complex (powerful?) instructions
  - More instructions per program
  - Faster fetch/execution of instructions

Design Decisions (1)

- Operation repertoire
  - How many ops?
  - What can they do?
  - How complex are they?
- Data types
- Instruction formats
  - Length of op code field
  - Number of addresses
- Registers
  - Number of CPU registers available
  - Which operations can be performed on which registers?
- Addressing modes (later…)
- RISC v CISC
Types of Operand

- Addresses
- Numbers
  - Integer/floating point
- Characters
  - ASCII etc.
- Logical Data
  - Bits or flags
(Aside: Is there any difference between numbers and characters? Ask a C programmer!)

Pentium Data Types

- 8 bit Byte
- 16 bit word
- 32 bit double word
- 64 bit quad word
- Addressing is by 8 bit unit
- A 32 bit double word is read at addresses divisible by 4
Specific Data Types

- General - arbitrary binary contents
- Integer - single binary value
- Ordinal - unsigned integer
- Unpacked BCD - One digit per byte
- Packed BCD - 2 BCD digits per byte
- Near Pointer - 32 bit offset within segment
- Bit field
- Byte String
- Floating Point

Pentium Floating Point Data Types
PowerPC Data Types

- 8 (byte), 16 (halfword), 32 (word) and 64 (doubleword) length data types
- Some instructions need operand aligned on 32 bit boundary
- Can be big- or little-endian
- Fixed point processor recognises:
  - Unsigned byte, unsigned halfword, signed halfword, unsigned word, signed word, unsigned doubleword, byte string (<128 bytes)
- Floating point
  - IEEE 754
  - Single or double precision

Types of Operation

- Data Transfer
- Arithmetic
- Logical
- Conversion
- I/O
- System Control
- Transfer of Control
Branch Instruction

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>SUB X, Y</td>
</tr>
<tr>
<td>201</td>
<td></td>
</tr>
<tr>
<td>202</td>
<td>BRZ 211</td>
</tr>
<tr>
<td></td>
<td>Unconditional Branch</td>
</tr>
<tr>
<td>210</td>
<td>BR 202</td>
</tr>
<tr>
<td>211</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Conditional Branch</td>
</tr>
<tr>
<td>225</td>
<td>BRE R1, R2, 235</td>
</tr>
<tr>
<td></td>
<td>Conditional Branch</td>
</tr>
<tr>
<td>235</td>
<td></td>
</tr>
</tbody>
</table>

Nested Procedure Calls

(a) Calls and returns

(b) Execution sequence
Use of Stack

- Initial stack content
- After CALL Proc1
- After CALL Proc2
- After RETURN
- After RETURN
- After RETURN

Byte Order

- What order do we read numbers that occupy more than one byte
- e.g. (numbers in hex to make it easy to read)
- 12345678 can be stored in 4x8bit locations as follows

<table>
<thead>
<tr>
<th>Address</th>
<th>Value (1)</th>
<th>Value (2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>184</td>
<td>12</td>
<td>78</td>
</tr>
<tr>
<td>185</td>
<td>34</td>
<td>56</td>
</tr>
<tr>
<td>186</td>
<td>56</td>
<td>34</td>
</tr>
<tr>
<td>186</td>
<td>78</td>
<td>12</td>
</tr>
</tbody>
</table>

- i.e. read top down or bottom up?
Byte Order Names

- The problem is called Endian
- The system on the left has the least significant byte in the lowest address
- This is called big-endian
- The system on the right has the least significant byte in the highest address
- This is called little-endian

Example of C Data Structure

```c
struct {
    int a; // 0x11_1314  word
    int b; // 0x234_2526_2728  double
    char c; // 0x3346  word
    char d[7]; // 'A', 'B', 'C', 'D', 'E', 'F', 'G'  byte array
    short e; // 0x5152  halfword
    int f; // 0x6161_6364  word
} s;
```

<table>
<thead>
<tr>
<th>Byte Address</th>
<th>Big-endian address mapping</th>
<th>Little-endian address mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00 01 02 03 04 05 06 07</td>
<td>07 06 05 04 03 02 01 00</td>
</tr>
<tr>
<td>08</td>
<td>08 09 0A 0B 0C 0D 0E 0F</td>
<td>0F 0E 0D 0C 0B 0A 09 08</td>
</tr>
<tr>
<td>10</td>
<td>10 11 12 13 14 15 16 17</td>
<td>17 16 15 14 13 12 11 10</td>
</tr>
<tr>
<td>18</td>
<td>18 19 1A 1B 1C 1D 1E 1F</td>
<td>1F 1E 1D 1C 1B 1A 19 18</td>
</tr>
<tr>
<td>20</td>
<td>20 21 22 23</td>
<td>23 22 21 20</td>
</tr>
</tbody>
</table>
Alternative View of Memory Map

Standard...What Standard?

- Pentium (80x86), VAX are little-endian
- IBM 370, Motorola 680x0 (Mac), and most RISC are big-endian
- Internet is big-endian
  - Makes writing Internet programs on PC more awkward!
  - WinSock provides htoi and itoh (Host to Internet & Internet to Host) functions to convert
Addressing Modes

- Immediate
- Direct
- Indirect
- Register
- Register Indirect
- Displacement (Indexed)
- Stack

Immediate Addressing

- Operand is part of instruction
- Operand = address field
- e.g. ADD 5
  - Add 5 to contents of accumulator
  - 5 is operand
- No memory reference to fetch data
- Fast
- Limited range

<table>
<thead>
<tr>
<th>Instruction</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>Operand</td>
</tr>
</tbody>
</table>
Direct Addressing

- Address field contains address of operand
- Effective address (EA) = address field (A)
- e.g. ADD A
  - Add contents of cell A to accumulator
  - Look in memory at address A for operand
- Single memory reference to access data
- No additional calculations to work out effective address
- Limited address space

```
Instruction
  Opcode  Address A

Memory
  Operand
```

Indirect Addressing

- Memory cell pointed to by address field contains the address of (pointer to) the operand
- EA = (A)
  - Look in A, find address (A) and look there for operand
- e.g. ADD (A)
  - Add contents of cell pointed to by contents of A to accumulator
- Large address space
- $2^n$ where $n =$ word length
- May be nested, multilevel, cascaded
  - e.g. EA = (((A)))
    - Draw the diagram yourself
- Multiple memory accesses to find operand
- Hence slower
**Indirect Addressing Diagram**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>Address A</td>
</tr>
<tr>
<td></td>
<td>Pointer to operand</td>
</tr>
<tr>
<td></td>
<td>Operand</td>
</tr>
</tbody>
</table>

**Register Addressing**

- Operand is held in register named in address filed
- EA = R
- Limited number of registers
- Very small address field needed
  - Shorter instructions
  - Faster instruction fetch
- No memory access
- Very fast execution
- Very limited address space
- Multiple registers helps performance
  - Requires good assembly programming or compiler writing
  - N.B. C programming
    - register int a;
- c.f. Direct addressing
Register Addressing Diagram

Instruction

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Register Address R</th>
</tr>
</thead>
</table>

Registers

Operand

Register Indirect Addressing

- C.f. 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

Instruction

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Register Address R</th>
</tr>
</thead>
</table>

Registers

Pointer to Operand

Memory

Operand
Displacement Addressing

- EA = A + (R)
- Address field hold two values
  - A = base value
  - R = register that holds displacement
  - or vice versa

Other Addressing Modes

- 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
  - c.f locality of reference & cache usage
- Base-Register Addressing
  - A holds displacement
  - R holds pointer to base address
  - R may be explicit or implicit
  - e.g. segment registers in 80x86
- Indexed Addressing
  - A = base; R = displacement; EA = A + R
  - Good for accessing arrays \(\rightarrow\) EA = A + R; R++
- Stack addressing
  - Operand is stored on top of the stack
Pentium Addressing Modes

- Virtual or effective address is offset into segment
  - Starting address plus offset gives linear address
  - This goes through page translation if paging enabled
- 12 addressing modes available
  - Immediate
  - Register operand
  - Displacement
  - Base
  - Base with displacement
  - Scaled index with displacement
  - Base with index and displacement
  - Base scaled index with displacement
  - Relative

Pentium Addressing Mode Calculation
PowerPC Addressing Modes

- Load/store architecture
  - Indirect
    - Instruction includes 16 bit displacement to be added to base
      register (may be GP register)
    - Can replace base register content with new address
  - Indirect indexed
    - Instruction references base register and index register (both may be
      GP)
    - EA is sum of contents
- Branch address
  - Absolute
  - Relative
  - Indirect
- Arithmetic
  - Operands in registers or part of instruction
  - Floating point is register only

PowerPC Memory Operand

Addressing Modes

![Diagram showing Memory Operand Addressing Modes](attachment:image)
Instruction Formats, Instruction Length

Instruction Formats
• Layout of bits in an instruction
• Includes opcode
• Includes (implicit or explicit) operand(s)
• Usually more than one instruction format in an instruction set

Instruction Length
• Affected by and affects:
  – Memory size
  – Memory organization
  – Bus structure
  – CPU complexity
  – CPU speed
• Trade off between powerful instruction repertoire and saving space

Allocation of Bits
• Number of addressing modes
• Number of operands
• Register versus memory
• Number of register sets
• Address range
• Address granularity
### VAX Instruction Examples

<table>
<thead>
<tr>
<th>Hexadecimal Format</th>
<th>Explanation</th>
<th>Assemble Notation and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>B 0</td>
<td>Opcode for MOVW</td>
<td>MOVW 25H:65H, ESI</td>
</tr>
<tr>
<td>C 4</td>
<td>Word decrement mode, register list</td>
<td>DEC EAX, EAX</td>
</tr>
<tr>
<td>0 1</td>
<td>Byte displacement mode, register list</td>
<td>MOV EAX, [EBX+8]</td>
</tr>
<tr>
<td>A 1</td>
<td>8S in hexadecimal</td>
<td>MOV 8S, EAX</td>
</tr>
<tr>
<td>B 0</td>
<td>Opcode for ADDL</td>
<td>ADD EAX, [EBX+8]</td>
</tr>
<tr>
<td>C 4</td>
<td>Short operands 16-bits</td>
<td>ADD EAX, [EBX+8]</td>
</tr>
<tr>
<td>0 1</td>
<td>Immediate 16-bits</td>
<td>ADD EAX, [EBX+8]</td>
</tr>
<tr>
<td>A 1</td>
<td>Index prefix 8S</td>
<td>MOV EAX, [EBX+8]</td>
</tr>
<tr>
<td>B 0</td>
<td>Indirect addressing displacement from PC</td>
<td>MOV EAX, [EBX+8]</td>
</tr>
<tr>
<td>C 4</td>
<td>Amount of displacement from PC relative to location A</td>
<td>MOV EAX, [EBX+8]</td>
</tr>
</tbody>
</table>

### Pentium Instruction Format

```
Instruction prefixes | Opcode | Mod/R/M | SIB | Displacement | Immediate

7 6 5 4 3 2 1 0
```

```
Mod Reg/Opcode R/M
```

```
Scale Index Base
```

```
7 6 5 4 3 2 1 0
```
# PowerPC Instruction Formats (1)

<table>
<thead>
<tr>
<th>Branch</th>
<th>Long Immediate</th>
<th>A</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit Conditional</td>
<td>Options</td>
<td>CR Bit</td>
<td>Branch Displacement</td>
</tr>
<tr>
<td>Bit Conditional</td>
<td>Options</td>
<td>CR Bit</td>
<td>Indirect through Link or Copied Register</td>
</tr>
</tbody>
</table>

(a) Branch instructions

<table>
<thead>
<tr>
<th>CR</th>
<th>Dest Bit</th>
<th>Source Bit</th>
<th>Source Bit</th>
<th>Add, OR, XOR, etc.</th>
<th>/</th>
</tr>
</thead>
</table>

(b) Condition register logical instructions

<table>
<thead>
<tr>
<th>Load/Store Indirect</th>
<th>Dest Register</th>
<th>Base Register</th>
<th>Displacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load/Store Indirect</td>
<td>Dest Register</td>
<td>Base Register</td>
<td>Index Register</td>
</tr>
<tr>
<td>Load/Store Indirect</td>
<td>Dest Register</td>
<td>Base Register</td>
<td>Displacement</td>
</tr>
</tbody>
</table>

(c) Load/store instructions

# PowerPC Instruction Formats (2)

<table>
<thead>
<tr>
<th>Load/Store Indirect</th>
<th>Dest Register</th>
<th>Base Register</th>
<th>Displacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load/Store Indirect</td>
<td>Dest Register</td>
<td>Base Register</td>
<td>Index Register</td>
</tr>
<tr>
<td>Load/Store Indirect</td>
<td>Dest Register</td>
<td>Base Register</td>
<td>Displacement</td>
</tr>
</tbody>
</table>

(c) Load/store instructions

<table>
<thead>
<tr>
<th>Arithmetic</th>
<th>Dest Register</th>
<th>Src Register</th>
<th>Src Register</th>
<th>O</th>
<th>Add, Sub, etc.</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add, Sub, etc.</td>
<td>Dest Register</td>
<td>Src Register</td>
<td>Signed Immediate Value</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logical</td>
<td>Src Register</td>
<td>Dest Register</td>
<td>Dest Register</td>
<td>ADD, OR, XOR, etc.</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>AND, OR, etc.</td>
<td>Src Register</td>
<td>Dest Register</td>
<td>Undefined Immediate Value</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rotate</td>
<td>Src Register</td>
<td>Dest Register</td>
<td>Shift Amount</td>
<td>Mask: Right</td>
<td>Mask: Left</td>
<td>R</td>
</tr>
<tr>
<td>Rotate</td>
<td>Src Register</td>
<td>Dest Register</td>
<td>Shift Amount</td>
<td>Mask</td>
<td>XO</td>
<td>SR</td>
</tr>
<tr>
<td>Rotate</td>
<td>Src Register</td>
<td>Dest Register</td>
<td>Shift Amount</td>
<td>Mask</td>
<td>XO</td>
<td>R</td>
</tr>
<tr>
<td>Shift</td>
<td>Src Register</td>
<td>Dest Register</td>
<td>Shift Type or Mask</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(d) Integer arithmetic, logical, and shift/rotate instructions

<table>
<thead>
<tr>
<th>Bit</th>
<th>Dest Register</th>
<th>Src Register</th>
<th>Src Register</th>
<th>Shift Width</th>
<th>R</th>
</tr>
</thead>
</table>

(e) Floating-point arithmetic instructions

- \( A = \) Absolute or PC relative
- \( t = \) Link or exception
- \( g = \) Type of shift amount field
- \( o = \) Overflow or sign extension
- \( X = \) OPO or shift amount field
- \( s = \) 64-bit implementation only
- \( r = \) Result condition in CRI