Computer Organization and Architecture

Chapter 10
Instruction Set Characteristics and Functions

What is an Instruction Set?

- The complete collection of instructions that are understood by a CPU
- Can be considered as a functional spec for a CPU
  - Implementing the CPU in large part is implementing the machine instruction set
- Machine Code is rarely used by humans
  - Binary numbers / bits
  - Machine code is usually represented by human readable assembly codes
  - In general, one assembler instruction equals one machine instruction

Elements of an Instruction

- Operation code (Op code)
  - Do this
- Source Operand reference
  - To this
- Result Operand reference
  - Put the result here
- Next Instruction Reference
  - When you have done that, do this...
  - Next instruction reference often implicit (sequential execution)

Operands

- Main memory (or virtual memory or cache)
  - Requires address
- CPU register
  - Can be an implicit reference (e.g., x87 FADD) or explicit operands (add eax, ecx)
- I/O device
  - Several forms:
    - Specify I/O module and device
    - Specify address in I/O space
    - Memory-mapped I/O just another memory address

Instruction Cycle State Diagram

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

### Instruction Types
- Data processing
  - Arithmetic and logical instructions
- Data storage (main memory)
- Data movement (I/O)
- Program flow control
  - Conditional and unconditional branches
  - Call and Return

### Design Issues
- These issues are still in dispute
- Operation Repertoire
  - How many operations and how complex should they be? Few operations = simple silicon; many ops makes it easier to program
- Data types
- Instruction Format and Encoding
  - Fixed or variable length? How many bits? How many addresses? This has some bearing on data types
- Registers
  - How many can be addressed and how are they used?
- Addressing Modes
  - Many of the same issues as Operation Repertoire

### Number of Addresses (a)
- 3 addresses
  - Operand 1, Operand 2, Result
  - \( a = b + c; \)
  - \( \text{add ax, bx, cx} \)
  - May be a fourth address - next instruction (usually implicit) [not common]

### Number of Addresses (b)
- 2 addresses
  - One address doubles as operand and result
  - \( a = a + b \)
  - \( \text{add ax, bx} \)
  - Reduces length of instruction over 3-address format
  - Requires some extra work by processor
    - Temporary storage to hold some results

### Number of Addresses (c)
- 1 address
  - Implicit second address
  - Usually a register (accumulator)
  - Common on early machines
  - Used in some Intel x86 instructions with implied operands
    - \( \text{mul ax} \)
    - \( \text{idiv ebx} \)
Number of Addresses (d)

- 0 (zero) addresses
  - All addresses implicit
  - Uses a stack. X87 example $c = a + b$:
    \[
    \text{fld } a \quad \text{;push a}  \\ 
    \text{fld } b \quad \text{;push b}  \\ 
    \text{fadd } \quad \text{;st(1) <- a+b, pop stack}  \\ 
    \text{fstop } c \quad \text{;store and pop c}
    \]
  - Can reduce to 3 instructions:
    \[
    \text{fld } a \quad \text{;push a}  \\ 
    \text{fld } b \quad \text{;push b}  \\ 
    \text{faddp } c \quad \text{;add and pop}
    \]

Computation of $Y = \frac{(a-b)}{(c + (d \times e))}$

- Three address instructions
  - \text{sub y,a,b}  \\  
  - \text{mul t,d,e}  \\  
  - \text{add t,t,c}  \\  
  - \text{div y,y,t}  \\  
- Two address instructions
  - \text{mov y,a}  \\  
  - \text{sub y,b}  \\  
  - \text{mov t,d}  \\  
  - \text{mul t,e}  \\  
  - \text{add t,c}  \\  
  - \text{div y,t}

How Many Addresses?

- More addresses
  - More complex instructions
  - More registers
  - Inter-register operations are quicker
  - Fewer instructions per program
  - More complexity in processor
- Fewer addresses
  - Less complex instructions
  - More instructions per program
  - Faster fetch/execution of instructions
  - Less complexity in processor
  - One address format however limits you to one register

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

Design Decisions (2)

- Registers
  - Number of CPU registers available
  - Which operations can be performed on which registers?
- Addressing modes (later...)
- RISC v CISC
Types of Operands

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

Pentium (32 bit) Data Types

- 8 bit Byte (unsigned or signed)
- 16 bit word (unsigned or signed)
- 32 bit double word (unsigned or signed)
- 64 bit quad word (unsigned or signed)
  - multiplication and division only in the integer processor plus a few obscure instructions such as cmpxchg8b
- Integer unit has instruction set support for both packed and unpacked BCD
- Memory is byte-addressable
- But dwords should be aligned on 4-byte address boundaries to avoid multiple bus cycles for single operands

Pentium FPU data types

- Pentium FPU (x87) supports IEEE 32, 64, and 80 bit floats
- All numbers internally are 80 bit floats
- Can load from and store to integer formats (signed) in word, dword, and qword formats
- Can load from and store to a tenbyte packed BCD format (18 digits + one byte for sign)

Pentium Numeric Data Formats

Pentium Byte Strings

- X86 processors have a set of 5 instructions called “string” instructions that can manipulate blocks of memory up to $2^{32} - 1$ bytes in length
- Blocks of memory can be manipulated as bytes, words, or dwords
- Operations:
  - CMPS: mem-to-mem compare
  - MOVS: mem-to-mem copy
  - SCAS: scan memory for match to value in accumulator
  - STOS: store accumulator to memory
  - LODS: load accumulator from memory

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
ARM Data Types

- ARM Processors support
  - Byte 8 bits
  - Halfword 16 bits
  - Word 32 bits
- Word access should be word aligned (memory address is divisible by 4) and halfword access should be halfword aligned
- Three alternatives for unaligned access
  - Default (truncate address; set I.O. bits to 0)
  - Alignment check: fault on unaligned access
  - Unaligned access: allow unaligned access transparently to the programmer; may require more than one memory access to load the operand

ARM Data Types

- ARM processors support both signed 2’s complement and unsigned interpretations of integers
- Most implementations do not provide floating point hardware
  - FP arithmetic can be implemented in software
  - ARM specs also support an optional FP co-processor with IEEE 754 singles and doubles

Endian Support

- The system control register E-bit can be set and cleared using the SETEND instruction

Types of Operations

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

Data Transfer

- Specify
  - Source
  - Destination
  - Amount of data
- May be different instructions for different movements source, destination, size
  - e.g. IBM mainframes
- Or one instruction and different addresses
  - e.g. VAX, Pentium (MOV)

Example IBM EAS/390 Operations

<table>
<thead>
<tr>
<th>Operation Character</th>
<th>Name</th>
<th>Number of Bits Transferred</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>Load</td>
<td>16</td>
<td>Transfer from memory to register</td>
</tr>
<tr>
<td>LB</td>
<td>Load HalfWord</td>
<td>16</td>
<td>Transfer from memory to register</td>
</tr>
<tr>
<td>LR</td>
<td>Load</td>
<td>32</td>
<td>Transfer from register to register</td>
</tr>
<tr>
<td>LDR</td>
<td>Load (Long)</td>
<td>32</td>
<td>Transfer from floating-point register to floating-point register</td>
</tr>
<tr>
<td>LB</td>
<td>Load (Long)</td>
<td>64</td>
<td>Transfer from memory to floating-point register</td>
</tr>
<tr>
<td>LD</td>
<td>Load (Long)</td>
<td>64</td>
<td>Transfer from memory to floating-point register</td>
</tr>
<tr>
<td>ST</td>
<td>Store</td>
<td>32</td>
<td>Transfer from register to memory</td>
</tr>
<tr>
<td>SSH</td>
<td>Store HalfWord</td>
<td>36</td>
<td>Transfer from register to memory</td>
</tr>
<tr>
<td>SEC</td>
<td>Store Character</td>
<td>8</td>
<td>Transfer from register to memory</td>
</tr>
<tr>
<td>SNE</td>
<td>Store (Short)</td>
<td>32</td>
<td>Transfer from floating-point register to memory</td>
</tr>
<tr>
<td>STD</td>
<td>Store (Long)</td>
<td>64</td>
<td>Transfer from floating-point register to memory</td>
</tr>
</tbody>
</table>
Types of data transfer operations

<table>
<thead>
<tr>
<th>Operation Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move (transfer)</td>
<td>Transfer word or block from source to destination</td>
</tr>
<tr>
<td>Store</td>
<td>Transfer word from processor to memory</td>
</tr>
<tr>
<td>Load (fetch)</td>
<td>Transfer word from memory to processor</td>
</tr>
<tr>
<td>Exchange</td>
<td>Swap contents of source and destination</td>
</tr>
<tr>
<td>Clear (reset)</td>
<td>Transfer word of 0s to destination</td>
</tr>
<tr>
<td>Set</td>
<td>Transfer word of 1s to destination</td>
</tr>
<tr>
<td>Push</td>
<td>Transfer word from source to top of stack</td>
</tr>
<tr>
<td>Pop</td>
<td>Transfer word from top of stack to destination</td>
</tr>
</tbody>
</table>

Arithmetic Operations

- Add: Compute sum of two operands
- Subtract: Compute difference of two operands
- Multiply: Compute product of two operands
- Divide: Compute quotient of two operands
- Absolute: Replace operand by its absolute value
- Negate: Change sign of operand
- Increment: Add 1 to operand
- Decrement: Subtract 1 from operand

Comparison is usually considered an arithmetic operation – subtraction without storage of results

Shift and Rotate Operations

Logical Operations

- AND: Performs logical AND
- OR: Performs logical OR
- NOT (complement): Performs logical NOT
- Exclusive OR: Performs logical XOR
- Test: Test specified condition; set flag(s) based on outcome

Translation and Conversion

- Conversion: convert one representation to another. Ex: x87 load and store instructions perform conversions from any numeric format to IEEE and vice versa
- Translate: table based translation
  - S/390 translate instruction TR R1, R2, L
    - R2 has address of a table of 8-bit codes
    - L bytes at addr R1 are replaced by byte at table entry in R2 indexed by that byte
    - Typically used for character code conversions
  - Intel XLATE instruction has implied operands
    - Bx points to the table
    - AL contains the index and is replaced by BX[al]

Input/Output

- May be specific instructions (Pentium IN and OUT)
- May be done using data movement instructions (memory mapped)
- May be done by a separate controller (DMA)
Systems Control

- Privileged instructions
- CPU needs to be in specific state
  - Ring 0 on 80386+
  - Kernel mode
- For operating systems use
- Examples: Pentium
  - LGDTR (Load Global Descriptor Table Register)
  - LAR (Load Access Rights)
  - MOV to/from Control Register

Transfer of Control

- Branch (JMP in x86)
  - Unconditional == goto
  - Conditional
    - Can be based on flags (condition codes) or other tests
    - I address format can have instructions such as
      `bra r1,r2, dest ; branch to dest if r1=r2`
- Skip instructions
  - Skip the next instruction if condition is true
  - Implied address
  - May include other ops
- Examples: Pentium
  - LGDTR (Load Global Descriptor Table Register)
  - LAR (Load Access Rights)
  - MOV to/from Control Register

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>BRZ 211</td>
</tr>
<tr>
<td>202</td>
<td></td>
</tr>
<tr>
<td>203</td>
<td></td>
</tr>
<tr>
<td>210</td>
<td>BR 202</td>
</tr>
<tr>
<td>211</td>
<td></td>
</tr>
<tr>
<td>212</td>
<td></td>
</tr>
<tr>
<td>213</td>
<td></td>
</tr>
<tr>
<td>225</td>
<td>BRE R1, R2, 235</td>
</tr>
<tr>
<td>226</td>
<td></td>
</tr>
<tr>
<td>227</td>
<td></td>
</tr>
<tr>
<td>228</td>
<td></td>
</tr>
</tbody>
</table>

Unconditional Branch

Conditional Branch

Procedure (Function) Calls

- Two instructions: CALL and RETURN
- Most machines use a stack mechanism:
  - CALL pushes return address on stack
  - RETURN pops stack into program counter
- Other approaches:
  - Store return address in special register
  - Store return address at start of procedure
  - Berkeley RISC: register windows
- These approaches all suffer from problems with re-entrancy

Software Interrupts

- Many architectures provide a software interrupt
  - Intel x86 provides for 224 possible software interrupts
- A software interrupt is an instruction, not an interrupt at all
- Effect is similar to a CALL instruction
- Return address and other data are pushed on the stack
- Address to which transfer is passed is located using the same vectoring mechanism that hardware interrupts use
- Intel x86 provides the IRET instruction for executing a RETURN from an interrupt

Use of Stack

0 4000 4001 4002 4003 4004 4005 4006 4007
0 4000 4001 4002 4003 4004 4005 4006 4007
Nested Procedure Calls

- Most high-level languages make a syntactic distinction between Procedures (no return value) and Functions (returns a value as a result of the call)
- At the machine level this distinction does not exist
- We have only a CALL with RETURN to the instruction after the call

Stack Frames and Parameters

- Stacks provide a very flexible mechanism for parameter passing
- To call a procedure, first push parameters on the stack
- Issue the CALL instruction
- The procedure may also use the stack for storage of local (volatile) variables
- The entire structure (parameters, frame pointer, return address, local storage) is called a Stack Frame

Frame pointer

- Intel architecture uses a semi-dedicated register as a frame pointer
- EBP (or BP) is SS-relative (same as ESP or SP)
- All other address registers are DS relative except for EIP (CS relative)

Other Specialized Instructions

- Many architectures provide specialized instructions designed for operating system use
- Intel x86 provides process synchronization instructions XADD, CMPXCHG, CMPXCHG8B
- Also bus control/synch operations ESC, LOCK, HALT, WAIT
**Example: XADD**

**RAX, REX Exchange and add [496]**

**Purpose:** Takes 2 operands, exchanges their contents and writes the sum of the two operands into the first operand.

**Syntax:**

```
XADD dest1, dest2
```

**Semantic:**

dest2 ← dest1; dest1 ← dest2 + dest1

**Flags:**

O: modified for sum

**Operator:**

reg, reg, reg, reg

**Note:**

This bizarre instruction can be used to write the world’s fastest Fibonacci calculator:

```
cal = 0, m = 1
```

**Example:**

```
loop fib500p
```

**Examples: HLT, WAIT, LOCK**

**HLT (Halt)**

**Purpose:**

Halt the processor until one of these events occurs: 1) pt interrupt; 2) a hardware reset or 3) DMA operation completes. Normally only used in a program to synchronize with a hardware device.

**Syntax:**

```
HLT
```

**Wait**

**Purpose:**

Monitor the TEST pin on the 8259 (later renamed to 8259A in later processors).

- If TEST = 0 there is no effect.
- If TEST = 1 then the processor halts until TEST = 0.

The TEST pin is usually connected to the TEST pin of the coprocessor, such as the 80386; thus the coprocessor can wake up the processor.

**Syntax:**

```
WAIT
```

**Note:**

Often inserted automatically by assemblers following a coprocessor instruction.

**SMM prefix**

**Purpose:**

Given the lock pin on the processor and disables external bus master, thus preventing access to memory or other resources, used to lock memory during critical operations.

**Syntax:**

```
lock
```

**MMX Instruction Set**

- MMX (Multi-Media eXtensions)
- Optimized instruction set for multimedia tasks
- Instructions are SIMD (Single-Issue, Multiple Data)
- Allow parallel operations on 2-8 operands in one clock cycle

**MMX Registers**

- 8 64-bit registers mm0 - mm7
- Unfortunately these were aliased to the X87 ST(0) - ST(7)
- Cannot intermingle floating point instructions and MMX instructions
- Pentium III added SSE (Streaming SIMD Extensions) along with 8 128-bit registers xmm0 - xmm7

**Saturation Arithmetic**

- When 2’s complement or unsigned operations are performed, normal arithmetic results in “wrap-around” with resulting overflow
  - e.g., in 8 bits 125+4 = -127
- In saturation arithmetic results are clamped to min and max values of the representation
  - 125+4 = 127
  - 125+125 = 127
- This is useful in digital signal processor, color arithmetic and other multimedia applications

**Saturation arithmetic 2**

- Because 2’s complement and unsigned representations have different min and max values, the instruction set provides signed and unsigned instructions for saturated arithmetic
- Audio uses signed arithmetic (positive and negative amplitudes)
- Color processing uses unsigned arithmetic (e.g., 32-bit color R,G,B, Alpha channels range from 0-255)
Saturation Arithmetic Example: Fade

- Video fade-out, fade-in effect
  - One scene gradually dissolves into another
- Two images are combined, pixel-by-pixel, using a weighted average:

\[
\text{Result}_\text{pixel} = \text{pixel}_A \times \text{fade} + \text{pixel}_B \times (1 - \text{fade})
\]

- A series of frames is produced, gradually changing the fade value from 0 to 1 (scaled to 8-bit integers)

Video Fade Algorithm

1. Extract low 8-bit pixel component from image A, and B
2. Subtract image B from image A
3. Multiply result by fade value
4. Add image B pixels
5. Pack new combined pixels back into image A

Video Fade Algorithm

- In the MMX code sequence the 8-bit pixels are converted to 16-bit elements to accommodate the 16-bit multiplication capability
- Assuming 640x480 resolution, and use of all 255 fade values the total number of instructions executed by the mmx code is ~525,000,000
- Same calculation without MMX code requires ~1,400,000,000 instructions

Video Fade

- MMX Instruction Set Overview

<table>
<thead>
<tr>
<th>Category</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical</td>
<td>PAND</td>
<td>66-bit binary logical AND</td>
</tr>
<tr>
<td></td>
<td>PNDN</td>
<td>66-bit binary logical NOT</td>
</tr>
<tr>
<td></td>
<td>OR</td>
<td>66-bit binary logical OR</td>
</tr>
<tr>
<td></td>
<td>XOR</td>
<td>66-bit binary logical XOR</td>
</tr>
<tr>
<td></td>
<td>PUBL [W, D, Q]</td>
<td>Parallel logical left shift of packed words, doublewords, or quadwords, or quadrant</td>
</tr>
<tr>
<td></td>
<td>PAPL [W, D, Q]</td>
<td>Parallel logical right shift of packed words, doublewords, or quadwords</td>
</tr>
<tr>
<td></td>
<td>MOVQ (64-bit)</td>
<td>Move quadword or quadword elements between registers</td>
</tr>
</tbody>
</table>

Video Fade

- MMX Code

```
// mm7, mm8
mov qword ptr [edi], [esi] ; Load the value from memory
movss xmm0, [edi] ; Load the single-precision floating point value
movss xmm1, [edi + 8] ; Load the second single-precision floating point value
movsd xmm2, [edi + 16] ; Load the double-precision floating point value
movd quadword ptr [edi + 24], xmm0 ; Store the value in memory
movd quadword ptr [edi + 32], xmm1 ; Store the second value in memory
movd quadword ptr [edi + 40], xmm2 ; Store the third value in memory
```

```
// mm7, mm8
// Datapaths
movq xmm0, [edi] ; Load the value from memory
movss xmm1, [edi + 8] ; Load the single-precision floating point value
movsd xmm2, [edi + 16] ; Load the double-precision floating point value
movd quadword ptr [edi + 24], xmm0 ; Store the value in memory
movd quadword ptr [edi + 32], xmm1 ; Store the second value in memory
movd quadword ptr [edi + 40], xmm2 ; Store the third value in memory
```
ARM Operation Types

- Principal categories
  - Load and Store
  - Branch
  - Data Processing
  - Multiply
  - Parallel Addition and Subtraction
  - Status Register Access

Load and Store

- ARM Load and Store are used to access memory
- All arithmetic and logical operations are performed on registers; may use immediate values but not memory operands
- Two broad types of load / store:
  - Load/store a 32-bit word or an 8-bit unsigned byte
  - Load/store a 16-bit unsigned halfword or load and sign extend a 16-bit halfword or an 8-bit byte

Branch

- Unconditional branch forwards or backwards up to 32MB
- The program counter R15 is addressable as a register so writing a value to R15 executes a branch
- Branch with Link (BL) can be used for function calls.
  - It preserves the return address in the Link Register (LR or R14)
- Conditional branches are determined by a 4-bit condition code in the instruction (discussed below)

Data Processing and Multiply

- Includes the usual add, subtract, test, compare and Boolean AND, OR, NOT instructions
  - Add/subtract also have saturated versions
- Integer multiply instructions operate on halfword or word operands and can produce results with upper part discarded or long results (2N bits)
- There are some complex multiply and accumulate instructions
  - multiply two registers and add the contents of a third to the result
  - Note that there is no divide instruction!

Parallel Addition and Subtraction

- Similar to MMX, can use two halfword operands or four byte operands
- Example ADD16 adds halfwords of two registers separately for two halfword results in a third
- Other operations include Halfword-wise exchange, add, subtract; unsigned sum of absolute differences

Condition Codes

- ARM defines four condition flags that are stored in the program status register:
  - N Negative  Z Zero
  - C Carry  V Overflow

- Equivalent to x86 S, Z, C, V flags
- Conditions test various combinations of these flags
Conditional Execution

- ALL instructions have a 4-bit condition code field so virtually all can be conditionally executed
- Any combination of bits except 1110 or 1111 designates conditional execution
- All arithmetic and logical instructions have an S-bit that indicates whether flags are to be updated or not
- Conditional execution is very useful for pipelined processors
  - Avoids need to flush pipeline when a branch is taken

Conditional Tests

Code is condition code field in instruction

<table>
<thead>
<tr>
<th>Code</th>
<th>Opcode</th>
<th>Condition Code</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>G2</td>
<td>2 = 0</td>
<td>Eq</td>
</tr>
<tr>
<td>001</td>
<td>G2</td>
<td>2 = 0</td>
<td>Noteq</td>
</tr>
<tr>
<td>010</td>
<td>G2</td>
<td>C = 0</td>
<td>Carry set/zero in case</td>
</tr>
<tr>
<td>011</td>
<td>G2</td>
<td>2 &lt; 0</td>
<td>Less than or equal</td>
</tr>
<tr>
<td>101</td>
<td>G2</td>
<td>2 &gt; 0</td>
<td>Greater than or equal</td>
</tr>
<tr>
<td>111</td>
<td>1111</td>
<td>1111</td>
<td>illegal</td>
</tr>
</tbody>
</table>