Chapter 10
Instruction Sets:
Characteristics and Functions
**Instruction Set** = The complete collection of instructions that are recognized by a CPU.

Representations:
- Machine code (Binary/Hex)
- Assembly code (Mnemonics)

Example (IAS – see ch.2):
00000101 00000000011       ADD M(3)
Elements of an Instruction

• Operation code (opcode)
  — Do this

• Source operand(s) reference(s)
  — To this

• Result operand reference
  — Put the answer here

• Next instruction reference
  — When you’ve finished, go here to find out what to do next...

00000101 0000000011  ADD M(3)
Where are the operands stored?

- In the instruction itself
  - immediate addressing
- Main memory
  - or virtual memory or caches, but this is transparent to the CPU
- CPU register
- I/O device
  - If memory-mapped, it’s just another address

00000101 0000000011       ADD M(3)
Instruction Cycle State Diagram (ch.3)
In machine code each instruction (incl. operands!) has a unique bit pattern

In assembly code:
- ADD M(3)
- ADD A,B
- etc.
High-level language and assembly language

\[ X = X + Y \]

compiles into:

- Load register A with contents of memory address 513
- Add reg. A w/contents of mem. addr. 514
- Store reg. A into mem. addr. 513

Conclusion: assembly is closer to the hardware.
Instruction Types

- Data processing (ALU)
- Data storage (main memory)
- Data movement (I/O)
- Program flow control
Number of Addresses (a)

• 3 addresses
  — Operand 1, Operand 2, Result
  — $a = b + c$;
  — There may be a forth – the next instruction! (usually implicit)
  — Not common
  — Needs very long words to hold everything
Number of Addresses (b)

- 2 addresses
  - One address doubles as operand and result
  - PDP-11 had **ADD B,A** (p.72)
  - Reduces length of instruction
  - Requires some extra work or specialized hardware: Temporary storage to hold some results ... although synchronization with a CLK usually solves the problem elegantly:

![Block diagram of a serial adder](http://web.njit.edu/~gilhc/ECE394/ECE394-V.htm)
Number of Addresses (c)

- 1 address
  - Implicit second address
  - Usually a register (accumulator)
  - Common on early machines
Number of Addresses (d)

- 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
### Temporary memory location (a.k.a. variable)

#### Three-address instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUB Y, A, B</td>
<td>Y ← A − B</td>
</tr>
<tr>
<td>MPY T, D, E</td>
<td>T ← D × E</td>
</tr>
<tr>
<td>ADD T, T, C</td>
<td>T ← T + C</td>
</tr>
<tr>
<td>DIV Y, Y, T</td>
<td>Y ← Y ÷ T</td>
</tr>
</tbody>
</table>

#### Two-address instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVE Y, A</td>
<td>Y ← A</td>
</tr>
<tr>
<td>SUB Y, B</td>
<td>Y ← Y − B</td>
</tr>
<tr>
<td>MOVE T, D</td>
<td>T ← D</td>
</tr>
<tr>
<td>MPY T, E</td>
<td>T ← T × E</td>
</tr>
<tr>
<td>ADD T, C</td>
<td>T ← T + C</td>
</tr>
<tr>
<td>DIV Y, T</td>
<td>Y ← Y ÷ T</td>
</tr>
</tbody>
</table>

#### One-address instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD D</td>
<td>AC ← D</td>
</tr>
<tr>
<td>MPY E</td>
<td>AC ← AC × E</td>
</tr>
<tr>
<td>ADD C</td>
<td>AC ← AC + C</td>
</tr>
<tr>
<td>STOR Y</td>
<td>Y ← AC</td>
</tr>
<tr>
<td>LOAD A</td>
<td>AC ← A</td>
</tr>
<tr>
<td>SUB B</td>
<td>AC ← AC − B</td>
</tr>
<tr>
<td>DIV Y</td>
<td>AC ← AC ÷ Y</td>
</tr>
<tr>
<td>STOR Y</td>
<td>Y ← AC</td>
</tr>
</tbody>
</table>

**Figure 10.3** Programs to Execute \[ Y = \frac{A - B}{C + (D \times E)} \]
<table>
<thead>
<tr>
<th>Number of Addresses</th>
<th>Symbolic Representation</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>OP A, B, C</td>
<td>A ← B OP C</td>
</tr>
<tr>
<td>2</td>
<td>OP A, B</td>
<td>A ← A OP B</td>
</tr>
<tr>
<td>1</td>
<td>OP A</td>
<td>AC ← AC OP A</td>
</tr>
<tr>
<td>0</td>
<td>OP</td>
<td>T ← (T - 1) OP T</td>
</tr>
</tbody>
</table>

AC = accumulator
T = top of stack
(T - 1) = second element of stack
A, B, C = memory or register locations
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, i.e. how do we get to the operands (later...)
- RISC v CISC
10.2 Types of Operands

- Addresses
- Numbers
  - Integer/floating point
- Characters
  - EBCDIC (old!) on IBM mainframes
  - IRA (International Reference Alphabet), a.k.a ASCII etc.
  - UNICODE
- Logical Data
  - Bits or flags, a.k.a. bitwise operations
(Aside: Is there any difference between numbers and characters? Ask a C programmer!)
Review / reading assignment  
(will also cover in this week’s lab)

Ch.9 sections:

- 9.1
- 9.2

Please follow all examples with pencil and paper (notebook!)
Write programs to compute \( \frac{A + B \cdot C}{D - E \cdot F} \) and store result in \( X \), on machines with instructions featuring:

- 0 addresses
- 1 address
- 2 addresses
- 3 addresses
<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>Byte, word (16 bits), doubleword (32 bits), quadword (64 bits), and double quadword (128 bits) locations with arbitrary binary contents.</td>
</tr>
<tr>
<td>Integer</td>
<td>A signed binary value contained in a byte, word, or doubleword, using two's complement representation.</td>
</tr>
<tr>
<td>Ordinal</td>
<td>An unsigned integer contained in a byte, word, or doubleword.</td>
</tr>
<tr>
<td>Unpacked binary coded decimal (BCD)</td>
<td>A representation of a BCD digit in the range 0 through 9, with one digit in each byte.</td>
</tr>
<tr>
<td>Packed BCD</td>
<td>Packed byte representation of two BCD digits; value in the range 0 to 99.</td>
</tr>
<tr>
<td>Near pointer</td>
<td>A 16-bit, 32-bit, or 64-bit effective address that represents the offset within a segment. Used for all pointers in a nonsegmented memory and for references within a segment in a segmented memory.</td>
</tr>
<tr>
<td>Far pointer</td>
<td>A logical address consisting of a 16-bit segment selector and an offset of 16, 32, or 64 bits. Far pointers are used for memory references in a segmented memory model where the identity of a segment being accessed must be specified explicitly.</td>
</tr>
<tr>
<td>Bit field</td>
<td>A contiguous sequence of bits in which the position of each bit is considered as an independent unit. A bit string can begin at any bit position of any byte and can contain up to 32 bits.</td>
</tr>
<tr>
<td>Bit string</td>
<td>A contiguous sequence of bits, containing from zero to (2^{32} - 1) bits.</td>
</tr>
<tr>
<td>Byte string</td>
<td>A contiguous sequence of bytes, words, or doublewords, containing from zero to (2^{32} - 1) bytes.</td>
</tr>
<tr>
<td>Floating point</td>
<td>See Figure 10.4.</td>
</tr>
<tr>
<td>Packed SIMD (single instruction, multiple data)</td>
<td>Packed 64-bit and 128-bit data types</td>
</tr>
</tbody>
</table>
x86 Data Types - General

- Recap: Byte (8), word (16), double word (32), quad word (64), double quad word (128)
- Addressing is by 8 bit unit
- Words do not need to align at even-numbered address
- Data accessed across 32 bit bus:
  - Units of double word read at addresses divisible by 4
  - The bus controller only accepts starting addr. that are multiples of 4
    - Faster this way!
  - The CPU must convert “misaligned” requests
Memory (mis-)alignment

OK!

Possible, but with extra work – see next slide
Memory (mis-)alignment

Source: http://www.songho.ca/misc/alignment/dataalign.html
Memory (mis-)alignment – what to do?

- Design data types to have sizes multiples of 4 Bytes as often as possible, e.g. in C:
  - int → 4, long long int → 8B
  - float → 4B, double → 8B, long double → 16B
  - short → 2B, char → 1B

- Compilers are alignment-aware → they try to assign memory space for variables such that memory access is optimized, a.k.a. **packing**

- But sometimes it’s not clear what the best way is to do it, e.g. structures, unions and classes in C/C++
void main() {
    struct stru {
      char m1;     // 1-byte
      // padding 3-byte space here!
      int m2;      // 4-byte
      double m3;   // 8-byte
    };
    struct stru s;

    printf("s.m1 starts at %p\n", &s.m1);
    printf("s.m2 starts at %p\n", &s.m2);
    printf("s.m3 starts at %p\n", &s.m3);
}
MSVC example

Packing unit becomes 1 Byte

```c
void main()
{
    #pragma pack(push, 1)
    struct stru {
        char m1; // 1-byte
        // padding 3-byte space here!
        int m2; // 4-byte
        float m3; // 8-byte
    }
    #pragma pack(pop)
    struct stru s;

    printf("s.m1 starts at %p\n", &s.m1);
    printf("s.m2 starts at %p\n", &s.m2);
    printf("s.m3 starts at %p\n", &s.m3);
}
```

Packing unit is reset to default value (8 Bytes)
**x86 Data Types**

**Endianness 😊**

- What is the problem?
- A multi-Byte register can be stored in memory two ways

Source: http://en.wikipedia.org/wiki/Endianness
x86 Data Types – Endianness

- All Intel 80x86 CPUs are little endian
  — Also Z80 and PDP-11!
- All Motorola 6800 CPUs are big endian
  — Also IBM System/360 and the early SPARCs
- Bi-endian CPUs: PowerPC and ARM

Read Cohen’s original 1981 article
“On Holy Wars and a Plea for Peace”
http://www.ietf.org/rfc/ien/ien137.txt
Bit-level endianness

- When transmitting data bit-by-bit over a communication link, the order of the bits becomes important!
- The Internet is big-endian!
- WinSock provides the functions `htoi` and `itoh` (Host to Internet & Internet to Host) to convert
  - Note: If the host machine is big endian, the functions don’t do anything, but they are left in the code for portability!
x86 Data Types – Endianness

Big endian, little endian - why stop here? 😊

On the PDP-11 (16-bit little-endian) for example, the compiler stored 32-bit values with the 16-bit halves swapped from the expected little-endian order. This ordering is known as PDP-endian.

The ARM architecture can also produce this format when writing a 32-bit word to an address 2 bytes from a 32-bit word alignment.

Source: http://en.wikipedia.org/wiki/Endianness
Read and take notes:
Appendix 10B
App. 10B Combined example: packing and endianness

```c
struct {
    int    a; //0x1112_1314    word
    int    pad; //
    double b; //0x2122_2324_2526_2728   doubleword
    char*  c; //0x3132_3334    word
    char   d[7]; // 'A'.'B'.'C'.'D'.'E'.'F'.'G'   byte array
    short  e; //0x5152    halfword
    int    f; //0x6161_6364   word
} s;
```

**Big-endian address mapping**

<table>
<thead>
<tr>
<th>Byte Address</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>00</th>
<th>01</th>
<th>02</th>
<th>03</th>
<th>04</th>
<th>05</th>
<th>06</th>
<th>07</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>08</td>
<td>08</td>
<td>09</td>
<td>0A</td>
<td>0B</td>
<td>0C</td>
<td>0D</td>
<td>0E</td>
<td>0F</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>'A'</td>
<td>'B'</td>
<td>'C'</td>
<td>'D'</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>'E'</td>
<td>'F'</td>
<td>'G'</td>
<td>51</td>
<td>52</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>61</td>
<td>62</td>
<td>63</td>
<td>64</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Little-endian address mapping**

<table>
<thead>
<tr>
<th>Byte Address</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>00</th>
<th>01</th>
<th>02</th>
<th>03</th>
<th>04</th>
<th>05</th>
<th>06</th>
<th>07</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>08</td>
<td>0F</td>
<td>0E</td>
<td>0D</td>
<td>0C</td>
<td>0B</td>
<td>0A</td>
<td>09</td>
<td>08</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>'D'</td>
<td>'C'</td>
<td>'B'</td>
<td>'A'</td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>51</td>
<td>52</td>
<td>'G'</td>
<td>'F'</td>
<td>'E'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>61</td>
<td>62</td>
<td>63</td>
<td>64</td>
<td>23</td>
<td>22</td>
<td>21</td>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
X86 Data Types – packed BCD

Read and take notes: p.357
Problem 10.24: Draw the memory map for the following C-style variables, assuming big endian and little endian layouts:

a.  double i = 0x1112131415161718;

b.  struct{
    int i;    //0x11121314
    int j;    //0x15161718
} s2;
Problem 10.24: Draw the memory map for the following C-style variables, assuming big endian and little endian layouts:

c. struct{
    short i;     //0x1112
    short j;     //0x1314
    short k;     //0x1516
    short l;     //0x1718
}s3;
Similar to Problem 10.24: Draw the memory map for the following C-style variables, assuming big endian and little endian layouts. **Packing is done on 4-byte boundaries!**

```
struct{
    short i;       //0x1112
    int  j;        //0x13141516
    char  c;       //0x17
    float l;       //0x18192021
}s4;
```
Infix and postfix (a.k.a. reverse Polish) notations

\[ a + b \rightarrow a \ b \ + \]
\[ a + (b \times c) \rightarrow a \ b \ c \ x \ + \]
\[ (a + b) \times c \rightarrow a \ b \ + \ c \ x \]

Do we really need these parentheses?

Let’s try some conversions by hand:

\[ (a + b) \times (c + d) \rightarrow \]
\[ (a + (b \times (c + d))) \rightarrow \]

The Pole Jan Łukasiewicz invented (prefix) Polish notation in the 1920s.
Lab week 9 --- Appendix 10A STACKS

Infix to postfix using a stack!

Dijkstra’s “shunting yard” algorithm
Dijkstra’s “shunting yard” algorithm:

1. Examine the next input element (L to R)
2. If operand, write it to output (L to R)
3. If opening parenthesis, PUSH
4. If operator:
   a. If top of stack is opening parenthesis, PUSH
   b. If it has higher precedence than top of stack (or stack is empty), PUSH
   c. Else POP and write to output and goto 4 with same operator
5. If closing parenthesis, POP and write to output until open parenthesis encountered. POP and discard parenthesis.
6. If there’s more input, goto 1
7. If no more input, POP all remaining operands and write to output
Let's try it out on our examples:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>a + b</td>
<td>a b +</td>
</tr>
<tr>
<td>a + (b x c)</td>
<td>a b c x +</td>
</tr>
<tr>
<td>(a + b) x c</td>
<td>a b + c x</td>
</tr>
<tr>
<td>(a + b) x (c + d)</td>
<td></td>
</tr>
<tr>
<td>(a + (b x (c + d)))</td>
<td></td>
</tr>
</tbody>
</table>

Individual work: Figure 10.17
Why is postfix notation important?

Because it allows to evaluate expression using “only” a stack!

Scan L to R, following these rules:

- **If operand, PUSH**
- **If operator:**
  - POP twice
  - apply operator
  - PUSH result

At the end, the result will be on the top of the stack!
Let’s try it out on our examples:

\[ \begin{align*}
  a + b & \rightarrow a \ b \ + \\
  a + (b \times c) & \rightarrow a \ b \ c \ x \ + \\
  (a + b) \times c & \rightarrow a \ b \ + \ c \ x \\
  (a + b) \times (c + d) & \rightarrow \\
  (a + (b \times (c + d))) & \rightarrow \\
\end{align*} \]

Individual work: Evaluate the output from Figure 10.17

End of week 9 Lab
Using the stack for “0-address” instructions

Table 10.1 Utilization of Instruction Addresses (Nonbranching Instructions)

<table>
<thead>
<tr>
<th>Number of Addresses</th>
<th>Symbolic Representation</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>OP A, B, C</td>
<td>A ← B OP C</td>
</tr>
<tr>
<td>2</td>
<td>OP A, B</td>
<td>A ← A OP B</td>
</tr>
<tr>
<td>1</td>
<td>OP A</td>
<td>AC ← AC OP A</td>
</tr>
<tr>
<td>0</td>
<td>OP</td>
<td>T ← (T − 1) OP T</td>
</tr>
</tbody>
</table>

AC = accumulator  
T = top of stack  
(T − 1) = second element of stack  
A, B, C = memory or register locations

Q: How exactly does the compiler generate the correct sequence of instructions?  
A: It converts the expression to postfix notation (Dijkstra’s “shunting yard”), then evaluates postfix using stack.
Show how it’s done for

\[ f = \frac{a - b}{c + (d \times e)} \]

**Step 0:** How is \( f \) written in source code?

**Step 1:** Convert to postfix

**Step 2:** Evaluate using stack
### Quiz answer

<table>
<thead>
<tr>
<th>Stack</th>
<th>General Registers</th>
<th>Single Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push a</td>
<td>Load R1, a</td>
<td>Load d</td>
</tr>
<tr>
<td>Push b</td>
<td>Subtract R1, b</td>
<td>Multiply e</td>
</tr>
<tr>
<td>Subtract</td>
<td>Load R2, d</td>
<td>Add c</td>
</tr>
<tr>
<td>Push c</td>
<td>Multiply R2, e</td>
<td>Store f</td>
</tr>
<tr>
<td>Push d</td>
<td>Add R2, c</td>
<td>Load a</td>
</tr>
<tr>
<td>Push e</td>
<td>Divide R1, R2</td>
<td>Subtract b</td>
</tr>
<tr>
<td>Multiply</td>
<td>Store R1, f</td>
<td>Divide f</td>
</tr>
<tr>
<td>Add</td>
<td></td>
<td>Store f</td>
</tr>
<tr>
<td>Divide</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pop f</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of instructions</th>
<th>10</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory access</td>
<td>10 op + 6 d</td>
<td>7 op + 6 d</td>
<td>8 op + 8 d</td>
</tr>
</tbody>
</table>

Figure 10.15 Comparison of Three Programs to Calculate \[ f = \frac{a - b}{c + (d \times e)} \]
X86 Data Types – SMID

SIMD = Single Instruction Multiple Data

- What other options? SISD, MISD, MIMD

- Read about Flynn’s taxonomy

Basic idea: Multimedia data consists of large numbers of small “chunks”. It is advantageous to process several chunks in parallel.

Examples:

- Audio samples are typically quantized on 16 bit
- Image pixels are represented on 3 Bytes (RGB)
Several “packed” data types are defined:

- **Packed byte and packed byte integer**
  - Bytes packed into 64-bit quadword or 128-bit double quadword

- **Packed word and packed word integer**
  - 16-bit words packed into 64-bit quadword or 128-bit double quadword

- **Packed doubleword and packed doubleword integer**
  - 32-bit doublewords packed into 64-bit quadword or 128-bit double quadword

- **Packed quadword and packed quadword integer**
  - Two 64-bit quadwords packed into 128-bit double quadword
X86 Data Types – SMID

Several “packed” data types are defined:

Source: http://arecorlib.sourceforge.net/theory.html
How are the “packed” data types used?

- Unpack → Process in parallel → Pack
- The MMX instruction set has specialized instructions for each of the above (Section 10.5)
- Take a look at Fig. 10.11/382 for a general sense of this process

Source: http://arecorlib.sourceforge.net/theory.html
FYI only: ARM Data Types

- 8 (byte), 16 (halfword), 32 (word) bits
- Halfword and word accesses should be word aligned
- Nonaligned access alternatives
  - Default
    - Treated as truncated
    - Bits[1:0] treated as zero for word
    - Bit[0] treated as zero for halfword
    - Load single word instructions rotate right word aligned data transferred by non word-aligned address one, two or three bytes
  - Data abort signal indicates alignment fault for attempting unaligned access
  - Unaligned access
    - Processor uses one or more memory accesses to generate transfer of adjacent bytes transparently to the programmer
- Unsigned integer interpretation supported for all types
- Two’s-complement signed integer interpretation supported for all types
- Majority of implementations do not provide floating-point hardware
  - Saves power and area
  - Floating-point arithmetic implemented in software
  - Optional floating-point coprocessor
  - Single- and double-precision IEEE 754 floating point data types
ARM Endian Support

- E-bit in system control register
- Under program control
10.4 Types of Operations

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

See tables 10.3 and 10.4 for overview
Data Transfer

- Specify
  - Source
  - Destination
  - Amount of data
- May be different instructions for different movements
  - Lingo: LOAD STORE MOVE
  - e.g. IBM 370 – see next table
- Or one instruction (MOV) and different addresses
  - e.g. in VAX, the operand contains the information about whether it’s in memory or register (not shown)
<table>
<thead>
<tr>
<th>Operation Mnemonic</th>
<th>Name</th>
<th>Number of Bits Transferred</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>L</strong></td>
<td>Load</td>
<td>32</td>
<td>Transfer from memory to register</td>
</tr>
<tr>
<td><strong>LH</strong></td>
<td>Load Halfword</td>
<td>16</td>
<td>Transfer from memory to register</td>
</tr>
<tr>
<td><strong>LR</strong></td>
<td>Load</td>
<td>32</td>
<td>Transfer from register to register</td>
</tr>
<tr>
<td><strong>LER</strong></td>
<td>Load (Short)</td>
<td>32</td>
<td>Transfer from floating-point register to floating-point register</td>
</tr>
<tr>
<td><strong>LE</strong></td>
<td>Load (Short)</td>
<td>32</td>
<td>Transfer from memory to floating-point register</td>
</tr>
<tr>
<td><strong>LDR</strong></td>
<td>Load (Long)</td>
<td>64</td>
<td>Transfer from floating-point register to floating-point register</td>
</tr>
<tr>
<td><strong>LD</strong></td>
<td>Load (Long)</td>
<td>64</td>
<td>Transfer from memory to floating-point register</td>
</tr>
<tr>
<td><strong>ST</strong></td>
<td>Store</td>
<td>32</td>
<td>Transfer from register to memory</td>
</tr>
<tr>
<td><strong>STH</strong></td>
<td>Store Halfword</td>
<td>16</td>
<td>Transfer from register to memory</td>
</tr>
<tr>
<td><strong>STC</strong></td>
<td>Store Character</td>
<td>8</td>
<td>Transfer from register to memory</td>
</tr>
<tr>
<td><strong>STE</strong></td>
<td>Store (Short)</td>
<td>32</td>
<td>Transfer from floating-point register to memory</td>
</tr>
<tr>
<td><strong>STD</strong></td>
<td>Store (Long)</td>
<td>64</td>
<td>Transfer from floating-point register to memory</td>
</tr>
</tbody>
</table>

Table 10.5  Examples of IBM EAS/390 Data Transfer Operations
Arithmetic

- Add, Subtract, Multiply, Divide
- Signed Integer
- Optional: Floating point
- May include
  - Increment (a++)
  - Decrement (a--)
  - Negate (-a)
Logical: Shift and Rotate

(a) Logical right shift

(b) Logical left shift

(c) Arithmetic right shift

(d) Arithmetic left shift

(e) Right rotate

(f) Left rotate
Logical

- Bitwise operations
- AND, OR, NOT
Conversion

- Example: Binary to BCD
- Complex example: the IBM EAS/390 TR instruction for converting between EBCDIC and ASCII
  — Pp.368-9
  — It involves an entire conversion table!

**TR R1 (4), R2**

- Instr. name TRanslate
- Register R1 holds the starting addr. of the block of bytes to translate
- Register R2 holds the starting addr. of the translation table
- # of bytes
Input/Output

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

- Privileged instructions, typically reserved for OS use
- CPU needs to be in a specific state
  - Ring 0 on 80386+
  - Kernel mode
- Examples:
  - Read/write control registers
  - Read/write memory protection key (e.g. IBM EAS/390)
  - Read/write of PCBs (process control blocks) in multiprogramming systems
Transfer of Control

Default: PC is incremented → sequential execution

In high-level languages we have flow control:
  — Decisions
  — Loops
  — Modularity

• Branch
• Skip
• Subroutine call
Transfer of Control - Skip

E.g. increment and skip if zero = ISZ

Before loop starts, R1 is loaded with the negative of the # of iterations!

Unconditional branch, a.k.a. **jump**

A note on labels and symbols …
Transfer of Control - Branch

<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>203</td>
<td></td>
</tr>
<tr>
<td>210</td>
<td>BR 202</td>
</tr>
<tr>
<td>211</td>
<td></td>
</tr>
<tr>
<td>225</td>
<td>BRE R1, R2, 235</td>
</tr>
<tr>
<td>235</td>
<td></td>
</tr>
</tbody>
</table>

Unconditional Branch

Conditional Branch
Transfer of Control - Procedure Calls

(a) Calls and returns

(b) Execution sequence

Function, subroutine, etc.
We have covered pp.359-372 of the text.

To do in notebook for next time:

- Answer review questions 1 – 14
- Solve the “0 Address” part of problem 6.
A Byte of memory contains the following bits:

1011 1010

Show the content after the following instructions have been applied (start with the original Byte for each):

— Arithmetic shift left, once
— Arithmetic shift left, twice
— Logical shift left, twice
— Arithmetic shift right, 4 times
— Logical shift right, 4 times
— Rotation right, 3 times
— Rotation left, 3 times
Transfer of Control - Procedure Calls

- A procedure may be called from multiple locations.
- Procedures may be nested (arbitrary depth, limited only by the maximum stack size!)
- Each procedure must (eventually) return to its caller. Where to store the return address?
  - In a dedicated register
  - At the start of the memory area used by the procedure itself
  - On the stack → elegantly solves:
    - reentrance (a.k.a. recursion) problem (e.g. factorial)
    - variable # of parameters problem (e.g. printf())
Use of Stack
Use of stack

Most modern CPUs have these 3 registers for managing the stack:

- SP
- FP (for variable # of parameters)
- BP (for indirect addressing, see Ch.11)
Stack Frame Growth Using Sample Procedures P and Q, each with locals.

(a) P is active

(b) P has called Q
Stack example with function parameters

Can you see why it make sense for the parameters to be stacked below (before) the return address?

Source: http://upload.wikimedia.org/wikipedia/commons/e/e7/Call_stack_layout.png
## 10.5 Intel and ARM operation types

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data Movement</strong></td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td>Move operand, between registers or between register and memory.</td>
</tr>
<tr>
<td>PUSH</td>
<td>Push operand onto stack.</td>
</tr>
<tr>
<td>PUSHA</td>
<td>Push all registers on stack.</td>
</tr>
<tr>
<td>MOVsx</td>
<td>Move byte, word, dword, sign extended. Moves a byte to a word or a word to a doubleword with two’s-complement sign extension.</td>
</tr>
<tr>
<td>LEA</td>
<td>Load effective address. Loads the offset of the source operand, rather than its value to the destination operand.</td>
</tr>
<tr>
<td>XLAT</td>
<td>Table lookup translation. Replaces a byte in AL with a byte from a user-coded translation table. When XLAT is executed, AL should have an unsigned index to the table. XLAT changes the contents of AL from the table index to the table entry.</td>
</tr>
<tr>
<td><strong>Arithmetic</strong></td>
<td></td>
</tr>
<tr>
<td>IN, OUT</td>
<td>Input, output operand from I/O space.</td>
</tr>
<tr>
<td>ADD</td>
<td>Add operands.</td>
</tr>
<tr>
<td>SUB</td>
<td>Subtract operands.</td>
</tr>
<tr>
<td>MUL</td>
<td>Unsigned integer multiplication, with byte, word, or double word operands, and word, doubleword, or quadword result.</td>
</tr>
<tr>
<td>IDIV</td>
<td>Signed divide.</td>
</tr>
</tbody>
</table>

Table 10.8 x86 Operation Types (1/4)
### Logical

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>AND operands.</td>
</tr>
<tr>
<td>BTS</td>
<td>Bit test and set. Operates on a bit field operand. The instruction copies the current value of a bit to flag CF and sets the original bit to 1.</td>
</tr>
<tr>
<td>BSF</td>
<td>Bit scan forward. Scans a word or doubleword for a 1-bit and stores the number of the first 1-bit into a register.</td>
</tr>
<tr>
<td>SHL/SHR</td>
<td>Shift logical left or right.</td>
</tr>
<tr>
<td>SAL/SAR</td>
<td>Shift arithmetic left or right.</td>
</tr>
<tr>
<td>ROL/ROR</td>
<td>Rotate left or right.</td>
</tr>
<tr>
<td>SETcc</td>
<td>Sets a byte to zero or one depending on any of the 16 conditions defined by status flags.</td>
</tr>
</tbody>
</table>

### Control Transfer

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMP</td>
<td>Unconditional jump.</td>
</tr>
<tr>
<td>CALL</td>
<td>Transfer control to another location. Before transfer, the address of the instruction following the CALL is placed on the stack.</td>
</tr>
<tr>
<td>JE/JZ</td>
<td>Jump if equal/zero.</td>
</tr>
<tr>
<td>LOOPE/LOOPZ</td>
<td>Loops if equal/zero. This is a conditional jump using a value stored in register ECX. The instruction first decrements ECX before testing ECX for the branch condition.</td>
</tr>
<tr>
<td>INT/INTO</td>
<td>Interrupt/Interrupt if overflow. Transfer control to an interrupt service routine</td>
</tr>
</tbody>
</table>

See example after table.
### String Operations

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVS</td>
<td>Move byte, word, dword string. The instruction operates on one element of a string, indexed by registers ESI and EDI. After each string operation, the registers are automatically incremented or decremented to point to the next element of the string.</td>
</tr>
<tr>
<td>LODS</td>
<td>Load byte, word, dword of string.</td>
</tr>
</tbody>
</table>

### High-Level Language Support

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENTER</td>
<td>Creates a stack frame that can be used to implement the rules of a block-structured high-level language.</td>
</tr>
<tr>
<td>LEAVE</td>
<td>Reverses the action of the previous ENTER.</td>
</tr>
<tr>
<td>BOUND</td>
<td>Check array bounds. Verifies that the value in operand 1 is within lower and upper limits. The limits are in two adjacent memory locations referenced by operand 2. An interrupt occurs if the value is out of bounds. This instruction is used to check an array index.</td>
</tr>
</tbody>
</table>

### Flag Control

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STC</td>
<td>Set Carry flag.</td>
</tr>
<tr>
<td>LAHF</td>
<td>Load AH register from flags. Copies SF, ZF, AF, PF, and CF bits into A register.</td>
</tr>
</tbody>
</table>

---

Read *Status Flags and Condition Codes*, p.378
<table>
<thead>
<tr>
<th><strong>Segment Register</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>LDS</td>
</tr>
<tr>
<td>HLT</td>
</tr>
<tr>
<td>LOCK</td>
</tr>
<tr>
<td>ESC</td>
</tr>
<tr>
<td>WAIT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Protection</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>SGDT</td>
</tr>
<tr>
<td>LSL</td>
</tr>
<tr>
<td>VERR/VERW</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Cache Management</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>INVD</td>
</tr>
<tr>
<td>WBINVD</td>
</tr>
<tr>
<td>INVLPG</td>
</tr>
</tbody>
</table>

“Systems control” category, for the OS only! 

Remember Section 4.4
x86 case study: Procedure call and return

Explain the following sequence of instructions:

CALL proc1

.....................

proc1:
PUSH EBP
MOV EBP, ESP
SUB ESP, bytes_for_locals
.....................
MOV ESP, EBP
ADD ESP, bytes_for_locals
RET
x86 case study: Procedure call and return

CALL proc1

..................

proc1:
PUSH EPB
MOV EPB, ESP
ENTER bytes_for_locals, 0
SUB ESP, bytes_for_locals
..................
MOV ESP, EBP
SUB ESP, bytes_for_locals
LEAVE
RET

C calling convention!
# x86 – SIMD / MMX instructions

<table>
<thead>
<tr>
<th>Category</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>PADD [B, W, D]</td>
<td>Parallel add of packed eight bytes, four 16-bit words, or two 32-bit doublewords, with wraparound.</td>
</tr>
<tr>
<td></td>
<td>PADDUS [B, W]</td>
<td>Add unsigned with saturation</td>
</tr>
<tr>
<td></td>
<td>PSUB [B, W, D]</td>
<td>Subtract with wraparound.</td>
</tr>
<tr>
<td></td>
<td>PSUBUS [B, W]</td>
<td>Subtract unsigned with saturation</td>
</tr>
<tr>
<td></td>
<td>PMULHW</td>
<td>Parallel multiply of four signed 16-bit words, with high-order 16 bits of 32-bit result chosen.</td>
</tr>
<tr>
<td></td>
<td>PMULLW</td>
<td>Parallel multiply of four signed 16-bit words, with low-order 16 bits of 32-bit result chosen.</td>
</tr>
<tr>
<td></td>
<td>PMADDWD</td>
<td>Parallel multiply of four signed 16-bit words; add together adjacent pairs of 32-bit results.</td>
</tr>
<tr>
<td>Comparison</td>
<td>PCMPEQ [B, W, D]</td>
<td>Parallel compare for equality; result is mask of 1s if true or 0s if false.</td>
</tr>
<tr>
<td></td>
<td>PCMPGT [B, W, D]</td>
<td>Parallel compare for greater than; result is mask of 1s if true or 0s if false.</td>
</tr>
<tr>
<td>Conversion</td>
<td>PACKUSWB</td>
<td>Pack words into bytes with unsigned saturation.</td>
</tr>
<tr>
<td></td>
<td>PACKSS [WB, DW]</td>
<td>Pack words into bytes, or doublewords into words, with signed saturation.</td>
</tr>
<tr>
<td></td>
<td>PUNPCKH [BW, WD, DQ]</td>
<td>Parallel unpack (interleaved merge) high-order bytes, words, or doublewords from MMX register.</td>
</tr>
<tr>
<td></td>
<td>PUNPCKL [BW, WD, DQ]</td>
<td>Parallel unpack (interleaved merge) low-order bytes, words, or doublewords from MMX register.</td>
</tr>
<tr>
<td>Logical</td>
<td>PAND</td>
<td>64-bit bitwise logical AND</td>
</tr>
<tr>
<td></td>
<td>PNDN</td>
<td>64-bit bitwise logical AND NOT</td>
</tr>
<tr>
<td></td>
<td>POR</td>
<td>64-bit bitwise logical OR</td>
</tr>
<tr>
<td></td>
<td>PXOR</td>
<td>64-bit bitwise logical XOR</td>
</tr>
<tr>
<td>Shift</td>
<td>PSLL [W, D, Q]</td>
<td>Parallel logical left shift of packed words, doublewords, or quadword by amount specified in MMX register or immediate value.</td>
</tr>
<tr>
<td></td>
<td>PSRL [W, D, Q]</td>
<td>Parallel logical right shift of packed words, doublewords, or quadword.</td>
</tr>
<tr>
<td></td>
<td>PSRA [W, D]</td>
<td>Parallel arithmetic right shift of packed words, doublewords, or quadword.</td>
</tr>
<tr>
<td>Data Transfer</td>
<td>MOV [D, Q]</td>
<td>Move doubleword or quadword to/from MMX register.</td>
</tr>
</tbody>
</table>
x86 – SIMD / MMX instructions

- Saturation arithmetic
- Example: fade-in fade-out effect (Fig. 10.11)
- Another example: weather report!

Source: http://www.tommesani.com/MMXExamples.html

PCMPEQ (packed compare for equality) is performed on the weathercaster and blue-screen images, yielding a bitmask that traces the outline of the weathercaster.

This bitmask image is PANDned (packed and not) with the weathercaster image, yielding the first intermediate image: now the weathercaster has no background behind her.

The same bitmask image is PANDed (packed and) with the weather map image, yielding the second intermediate image.

The two intermediate images are PORed (packed or) together, resulting in final composite of the weathercaster over weather map.
SKIP ARM Operation Types

Homework for Ch.10 (Due Thu, Nov 11):

• 16
• 18 a, b
• 19 a, b
• 20
• 23
• 28