Machine and Assembly Language Principles

- Assembly language instruction is synonymous with a machine instruction.
- Therefore, need to understand machine instructions and on what they operate - the architecture.

A machine instruction contains:

- The main operation code (OPCODE)
- Optionally (depending on opcode) one or more OPERAND SPECIFIERS.

Examples (80X86):

```
Machine code      Assembly language code

   90              nop

6601D8             add eax, ebx
```

Defining the Architecture

1. What kinds of operations?
2. What kinds of operands (registers, memory, other)?
3. How are operands specified (addressing modes)?
4. What is the instruction format?
5. What resources can be operated on (register set)?

Instruction format is the machine code, seeminly irrelevant to assembly language programmer...

However, significant impact exists - helps understand many seemingly wierd "features" of machine (e.g. why is it that conditional jump is only at most 128 bytes?)

Register Sets

CPU manufacturers differ widely on method of designing the register set.

- Motorola (680X0) has 32 bit registers:
  - PC (program counter)
  - PSW
  - Data registers D0-D7
  - Address registers A0-A7

    Can also directly access lower 8 bits or lower 16 bits of data registers.
  - Additional registers available in SUPERVISOR mode.

- Intel (80X86) has 32 bit registers:
  - IP (instruction pointer, i.e. program counter)
  - Flags register
  - EAX, EBX, ECX, EDX ("general purpose" registers)
  - ESI, EDI ("index" registers)
  - ESP, EBP (stack control registers)

    Can also directly access lower 8 bits or lower 16 bits of data registers, e.g. AL, AX.

    For historical reasons, can also access next-to-lowest 8 bits, e.g. AH.
  - 16 bit SEGMENT registers (CS, DS, ES, SS, GS, FS)
  - Additional registers available in KERNEL mode.

Note: registers cannot be referred to as part of main memory, thus no "pointer to a register".

Instruction Format

Widely different for different CPUs. Can have great variability even for the SAME CPU (especially Intel 80X86)!

In principle, works as follows:

- Some specific bits (e.g. 6 most significant bits) determine main operation type.
- Based on above, rest of bits are further operation specifiers or operand specifiers.
- In variable-length instruction sets, additional bytes/words used to specify operands.

Scheme can be rather complicated (especially in Intel 80X86, where instruction length ranges from 1 to 17 bytes, and backward compatibility necessitates an “escape” opcode scheme).

In assembly language, operation denoted by a MNEMONIC. These menmonics are MACHINE DEPENDENT and VENDOR DEPENDENT... Examples:

1. NOP means “no operation”
2. INC means “increment”
3. SUB means “subtract”
4. AND means “and” (bitwise)

Operands in assembly language come after the operation mnemonic, usually separated by commas. Examples (80X86):

```
90                  nop

6601D8              add eax, ebx

6667298C43E46F0500
          sub [eax*2+ebx+0x56fe4], ecx
```

Note: “little endian” encoding!

Assembly Language (continued)

In addition to mnemonic specification of operations and operands, major feature of assembly language is LABELS and other symbols.

Labels defined in source program, used to symbolically specify locations in memory.

```
my_string:  db "Hello world!", 0

my_var:     dw 0

printf:     push ebp
            mov  esp, ebp
```

Labels can be refered to, and are converted into addresses (i.e. numbers) by the assembler (and linker).

```
            push my_string
            call printf
```

Other symbols types available for compile time, using EQU, and macro definitions.

Operands and Addressing Modes

Number of Operands

For instructions requiring operands, need to specify operands either IMPLICITLY or EXPLICITLY.

Example 1: Intel "push" instruction works on memory implicitly specified by ESP, but also has an explicit operand (what to push).

```
push 4
```

Example 2: "add" instruction in principle needs 3 operands: which two numbers to add (source, or "src" for short) and where to put the result (destination, or "dst" for short).
In some processors (e.g. 80X86) one of the sources is same as destination.

```
addl3  r1, r2, r3  ;  r1 + r2 -> r3
(VAX)

add    eax, ebx    ;  ebx + eax -> eax
(80X86, note REVERSE operand spec.)
```

Size of Operands

Specifying the size of the operand - in machine code done by a bit or 2 in instruction. In the assembler, vendor dependent.

- By mnemonic opcode suffix or extension:

```
(VAX)      addl2   r1, r2
(680X0)    ADD.L   D0, D1
(80X86)    movsb
```

- By default (e.g. "push" operand size is 4 bytes unless specified otherwise).
- By operand name (Intel):

```
add eax, ebx
add al, [my_string]
```

- By explicit specification:

```
inc byte [1000]
```

NOTE: in instruction, size of operands is (almost always) equal -
NO AUTOMATIC CONVERSION!

Addressing Modes

Where is the data to load (or store)? Specified in instruction in one of many possible, machine dependent, ADDRESSING MODES.

Note: names of modes - vendor dependent.

- Data could be either in registers or in memory.
- Not all addressing modes, and not every mix is allowed for every instruction!
- **Example**: for most 80X86 instructions, CANNOT have all operands in memory: one must be register or "immediate" (i.e. part of the instruction).

Addressing modes we will examine: register, immediate, direct (absolute), register indirect, displacement (offset), relative, indexed, indirect, auto-increment/auto decrement, others.

Each addressing mode is useful for some type of COMMON DATA ACCESS, also shown.

Memory Organization

- Most data is stored in MAIN MEMORY.
- Must understand MEMORY ORGANIZATION of the computer,

Questions you should be able to answer:

1. What is the basic addressible unit?
   - Memory is byte-addressible for 80X86, VAX, 680X0.
   - Only word-addressible (NOVA minicomputer, CDC mainframes).
2. How are words mapped into bytes?
   - Little-endian in 80X86, VAX, etc.
   - Big-endian in 680X0, etc.
3. What is the address space (or spaces?)
   - Flat 32 bit addressible space (80X86/linux).
   - Segmented (80X86 in other modes...)

Register Addressing

Application: operate on a variable or intermediate variable.

Machine code: A few bits in instruction.

Assembly language: specify register name.

Machine code	Assembler
`40`	`inc eax`
`43`	`inc ebx`

Immediate Addressing

Application: operate on a CONSTANT.

In **machine code**: operand is part of the instruction.

In **assembly language**: specify the number.

Machine code	Assembler
`050145`	`add ax, 0x4501`

Again, observe "little endian" encoding of constant in instruction.

Absolute (Direct) Addressing

In **machine code**: memory address of operand is part of the instruction.

In **assembly language**: specify the address as a number or label (within square brackets in NASM, but without the brackets in other assemblers, e.g. MASM...)

```
Machine code        Assembler
 FF06[1000]       inc word [my_string]
 FF060010         inc word [0x1000]
```

Application: simple ("global") variables. For example, the C sequence:

```
   int x = 0;
   x++;
```

MIGHT be written in assembler:
(ignoring sections...)

```
x: dd  0   ; NOT a variable definition!!
   inc dword [x]
```

Register Indirect Addressing

Operand is in memory at an address contained in the specified register. **Assembler**:

```
inc byte [ebx]
```

In the example, if the value of ebx is 0x104, the byte residing at memory address 0x104 is incremented.

Application: access through or de-referencing a pointer.
(The pointer here resides in the register.)

Example: the above could be used to implement the C code:

```
*p = *p + 1;
```

(Assuming the pointer p is in register ebx.)

Note: address of operand is called the effective address (EA). In defining the addressing mode, documentation may state effective address equation (e.g. EA=[ebx] here).

Displacement Addressing

Specifies an operand in memory at an address displaced by a constant from an address specified by a register **(EA=[reg]+disp)**.

Assembler: specify a register and a displacement (offset) in bytes.

```
inc byte [ebx+0x10]
```

If the value of ebx is 0x104, the byte residing at memory address 0x114 is incremented.

Application: structure element access.

For example, suppose we have the C code:

```
struct foo {int x; int y; char z;} ;
p = & foo;   /* assume correct type */
p->foo ++;
```

We could implement the last two lines in assembler:

```
mov  ebx, [p]
inc  byte [ebx+8]
```

Relative Addressing

Same as displacement addressing, except that register is the program counter (instruction pointer).

EA=[PC]+disp

Assembler: specify displacement as a number or use a label.

```
Machine code    Assembler
7502                jnz next ; = jnz $+4
6640                inc eax
66F7D8        next: neg eax
```

Application: relative jumps, position-independent code.

Indexed Addressing (with Displacement)

Application: access an array element.
Assembler: specify a register, a multiplier (element size in bytes), and a basis address.

```
inc dword [myarray + ebx*4]
```

If the value of ebx is 5, the dword residing at memory address myarray+20 is incremented.
Note: does not really allow multiplication - multiplier limited to 1, 2, or 4 in 80X86.
Variants:
a) With no displacement

```
inc dword [ebx*4]
```

b) Indexed plus indirect

```
inc dword [ebx*4+eax]
```

c) Indexed plus indirect with displacement

```
inc dword [myarray + ebx*4+eax]
```

Other Addressing Modes

Auto-increment (post-increment) and **auto-decrement** (pre-decrement) modes:

```
(680X0)    MOVI.B  #5, -(A7)
  ; Dec. pointer then move immediate 5.
(680X0)    MOVI.B  (A7)+, D0
  ; Increment pointer then move to D0.
```

Used to advance pointer after or before access, such as in C code:

```
      c=*p++;
```

Also to implement **stacks** - the following are equivalent:

```
(80X86)         push        5
(fake 80X86)    mov         -[esp], 5
```

(But the latter is not directly available in 80X86.)

Double indirect addressing modes - operand at address specified by data in memory to which register points:

```
(VAX)    inc *(r1)
```

Other, more complicated indirection modes exist, such as unlimited indirection using indirect bit (NOVA mini-processor).

Limited absolute addressing modes: current page addressing, page-k addressing, was used in early microprocessors and controllers.