Recitation 1: Intel 80x86 Instructions and Basic Assembly

COP3402 FALL 2015 – ARYA POURTABATABAIE

FROM EURIPIDES MONTAGNE, FALL 2014

Main Characteristics

Memory

- Composed of 8-bit bytes
- Each byte has a 32-bit label called a physical address
- Addressing is by byte, not by word
- Memory size = 4,294,967,296 (2^32) bytes

Bit Order and Endianness

- *Bits* within *bytes* have the most-significant bit on the *left* ("76543210")
- *Bytes* within *words* have the most-significant byte on the *right* – "little endian" numbering

Registers

- 16 general-purpose registers, along with other specific-purpose registers
- Registers are 16 to 32 bits long

Registers

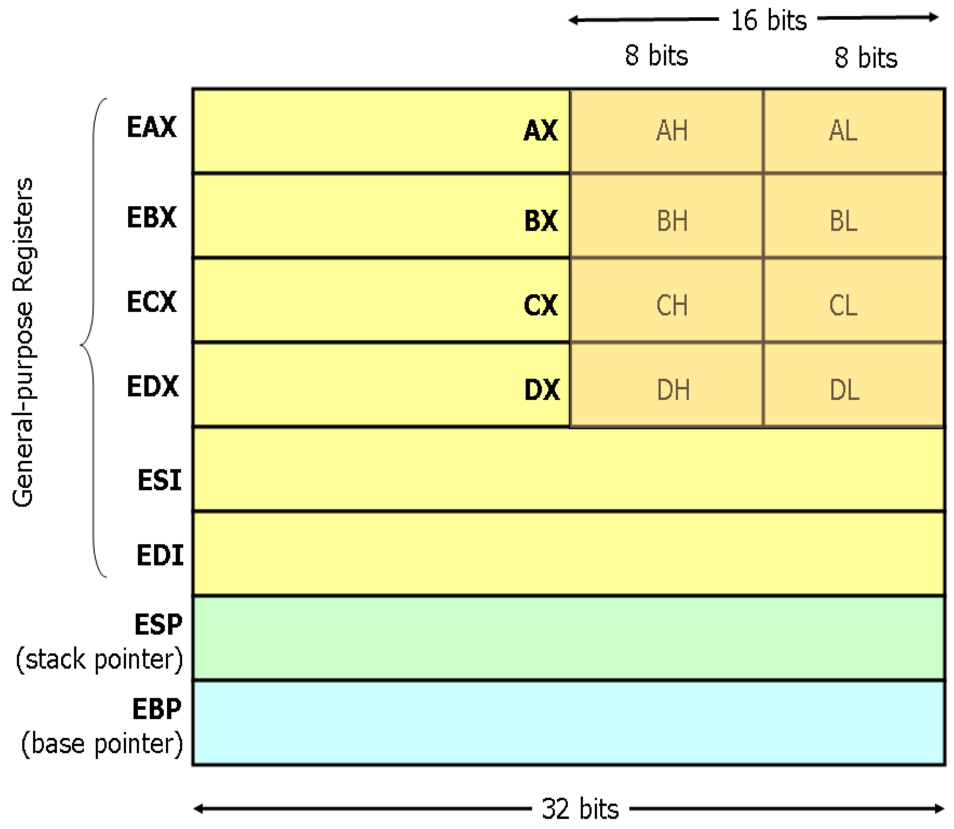
- Most of the register purposes are self-explanatory
- All eight general-purpose registers can be used as *base* registers for base + index addressing
- All but the stack pointer can also be used as the index
- EAX and EDX are considered the primary and secondary accumulators
  - This doesn't matter for most things…
  - …except multiplication and division of large integers, which will still load results directly into them

Mnemonic	Length	Special Use
EAX	32	Accumulator; General purpose
EBX, ECX, EDX	32	General purpose
ESI	32	Source index; source address
EDI	32	Destination index; address of destination
ESP	32	Stack pointer; address of top of stack
EBP	32	Base pointer; address of reference point in stack
CS, DS, ES, SS, FS, GS	16	Selector for "Code Segment", "Data Segment", "Extra Segment", "Stack Segment", and "Additional Segments"
EIP	32	Instruction pointer; next instruction
EFLAGS	32	Collection of flags and status bits (carry, parity, zero, sign, overflow, etc.)

Registers – History and the Present

- Originally, registers on 80x86 were 16-bit
- 32-bit *extended* versions arrived with the 80386
- Some bits of EAX, EBX, ECX and EDX can also be accessed as 16-bit and 8-bit registers

31-24	23-16	15-8	7-0
EAX, EBX, ECX, EDX			
		AX, BX, CX, DX	
		*H	*L



Integers and Strings

Integers are stored as binary numbers in one of four sizes.

- 8 bits (byte)
- 16 bits (word)
- 32 bits (double-word)
- 64 bits (quad-word)

2’s complement representation is used for negative values.

Characters in strings are often stored using 8-bit ASCII or 16-bit Unicode codes.

697_{10} = $0000\ 0010\ 1011\ 1001_2$ (word)
= $00\ 00\ 02\ \mathrm{B9}_{16}$ (dword)

-565_{10} = $1111\ 1101\ 1100\ 1011_2$ (word)
= $\mathrm{FF\ FF\ FD\ CB}_{16}$ (dword)

Floating-Point Numbers

The FPU (floating point unit) is a separate part of the chip that does floating point math.

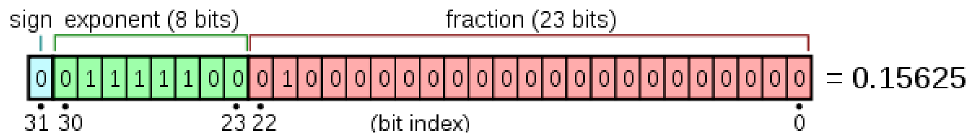
- It has its own registers and operations, separate from integer registers and operations
- *Its architecture is outside the scope of this class*

Floating Point Format:

- Sign bit: 1 bit
- Exponent width: 8-11 bits
- Significand precision/fraction: 24-53 (23-52 explicitly stored)

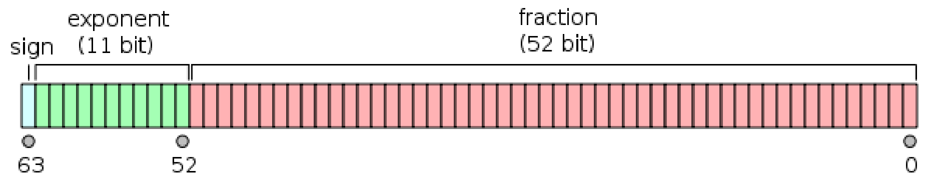
IEEE Single-Precision

- Value = $(-1)^{sign} \times (1._{\text{fraction part}}) \times 2^{e-127}$
- Max value ≈ 3.40×10^{38}, Min value ≈ 1.18×10^{-38}



IEEE Double-Precision

- Value = $(-1)^{sign} \times (1._{\text{fraction part}}) \times 2^{e-1023}$
- Max value ≈ 1.79×10^{308}, Min value ≈ 2.23×10^{-308}



Assembly: Instructions

Assembly language instructions are directly converted to object code (byte code). They typically take the form:

```
    Mnemonic Operand1(tgt), Operand2(src), [Op3], [Op4]
```

Opcodes are typically one-byte, but can be two.

Example:

```
    add eax, 158   ; add 158 to the contents of the EAX register
```

Assembly: Instruction Set and Addressing

INSTRUCTION SET

80x86 has a large number of instructions. Some commonly used mnemonics are:

mov	Copy data
add	Addition
sub	Subtraction
mul	Multiplication
div	Division
jmp	Jump (branch)
cmp	Compare

ADDRESSING

Addressing Modes

- Immediate – Data built into the instruction
- Register – Data in a register
- Memory – Data at a memory address

Memory Modes

- Direct – Location built into the instruction
- Register Indirect – Location in a register

Assembly: A Sample Program

This piece of a program (only a piece, as all the "front matter" is missing) converts a hardcoded temperature from Celsius to Fahrenheit.

It then saves the results into an equally-hardcoded memory location.

We didn't explicitly cover all of the mnemonics, but the process should still be fairly clear.

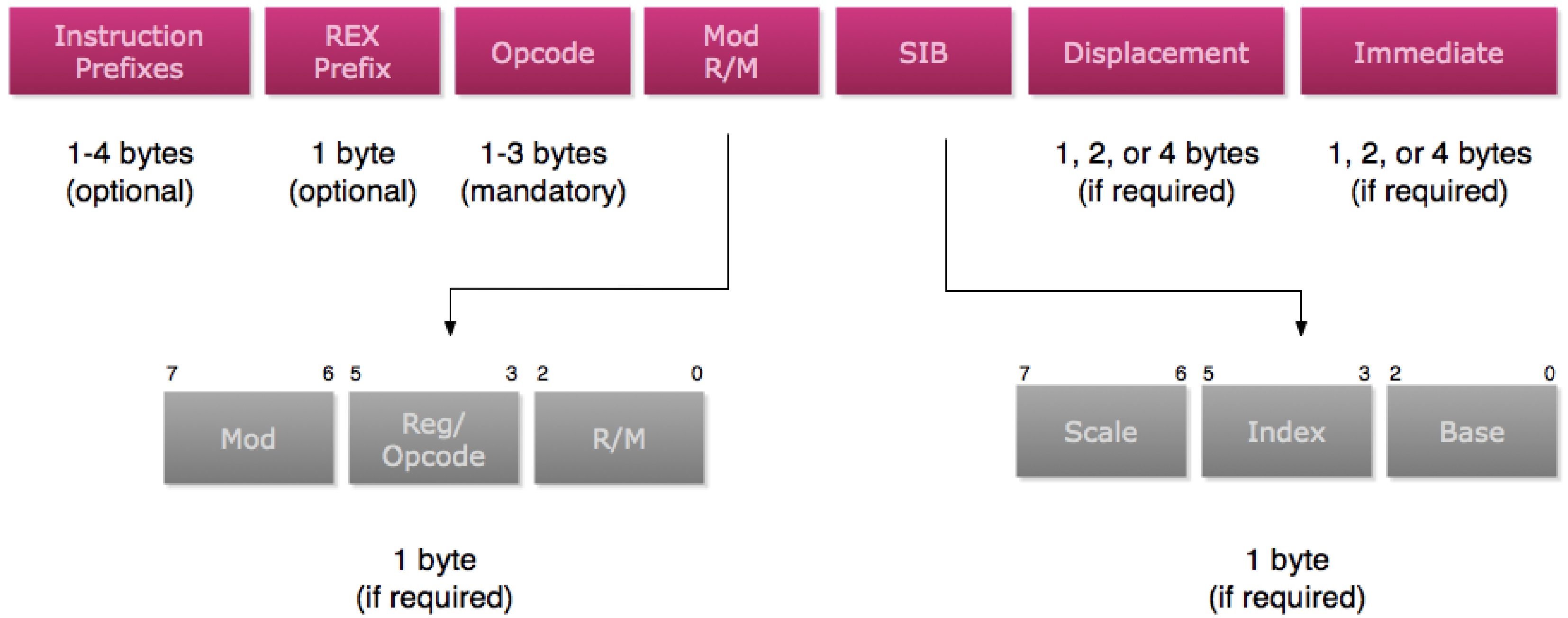
`mov  eax, 36`	; move `36` into `eax` register
`imul eax, 9`	; "immediate" multiple previous line by `9`. `eax = eax×9`
`mov  ebx, 5`	; divisor
`cdq`	; convert `ebx` double to quadword (prepare for division)
`idiv ebx`	; divide (implicit accumulator) by `ebx`. `eax = eax÷5`
`add  eax, 32`	; `eax = eax+32`
`mov  [02CDh], eax`	; save results to memory address $\texttt{02CD}_{16}$

Instruction Format

Here we see all the components of an 80x86 instruction in object code format.

The **Mod Register/Memory** byte determines the addressing mode, which registers are used, and whether memory is involved.

If memory *is* involved, the *direction* of transfer is actually determined by the opcode itself.



Mod, Reg and R/M Tables

Mod	Displacement
`00`	If R/M is `110`, Displacement (32 bits) is address Otherwise, no displacement
`01`	Eight-bit displacement, sign-extended to 32 bits
`10`	32-bit displacement (example: MOV [BX + SI]+ displacement, AL)
`11`	R/M is treated as a second "reg" field

Reg	W=0	W=1	Double word
`000`	AL	AX	EAX
`001`	CL	CX	ECX
`010`	DL	DX	EDX
`011`	BL	BX	EBX
`100`	AH	SP	ESP
`101`	CH	BP	EBP
`110`	DH	SI	ESI
`111`	BH	DI	EDI

- Mod `00`, R/M `110` is special
- Normally, this would be the operand [BP], but instead the 32-bit “displacement” is treated as an absolute address
- To encode [BP], use:
  Mod = `01`, R/M = `110`, 8-bit displacement = `0`.

R/M	Operand Address
`000`	(BX) + (SI) + displacement (0, 1, 2, or 4 bytes long)
`001`	(BX) + (DI) + displacement (0, 1, 2, or 4bytes long)
`010`	(BP) + (SI) + displacement (0, 1, 2, or 4bytes long)
`011`	(BP) + (DI) + displacement (0, 1, 2, or 4bytes long)
`100`	(SI) + displacement (0, 1, 2, or 4bytes long)
`101`	(DI) + displacement (0, 1, 2, or 4bytes long)
`110`	(BP) + displacement unless mod = 00 (see mod table)
`111`	(BX) + displacement (0, 1, 2, or 4 bytes long)

Example 1

`xor CL, [12H]`

XOR the contents of register `CL` (last byte of the `ECX` register) with the contents of address `12H`

The opcode for `xor` is 001100*dw*, with:

- Direction *d* = 1 because the register is the destination
- Word size *w* = 0 because we are using bytes

From there we determine:

- MOD = 00 for simple displacement
- The Reg code for `CL` is 001
- R/M = 110

So the instruction becomes:

- $00110010\ 00001110\ 00010010\ 00000000\ 00000000\ 00000000_2$, or
- $32\ 0E\ 12\ 00\ 00\ 00_{16}$

Example 2

`add AL, BL`

Add the contents of register `BL` to register `AL`.

The opcode for `add` is 000000*dw*, with:

- Direction *d* = 1 because we will use the “Reg” register as the destination
- Word size *w* = 0 because we are using bytes

From there we determine:

- MOD = 11 to use R/M as a register field
- The Reg code for `AL` is 000
- The Reg code for `BL` is 011

So the instruction becomes:

- $00000010\ 11000011_2$, or
- $02\ C3_{16}$