Microprocessor System Design

Omid Fatemi

Machine Language Programming

(omid@fatemi.net)

Outline

- **Assembly programming**
- **Instruction set**
- **Simple program**

قبلا به اینجا رسیدیم که ماشین با زبان
اسمبلی یعنی همان زبان ماشین یعنی
همان ۰ و ۱ ها کار می‌کند.

Assembly Programming

- CPU works in binary
- All instructions, data are in binary
- Binary instructions (0, 1) are Machine Language
  - Or even hexadecimal representation
- Assembly language
  - Mnemonics
  - Low level

از آنجایی که نوشتن اعداد ثابت دشوار است به سراغ اسمبلی می‌رویم

Still low level- they get interpreted and not compiled

- Assembler
- Linker

Assembler versus Machine Code

```
ADD   AX, BX      ;AX gets value AX+BX
SUB   AX, BX      ;AX gets value AX-BX
AND   AX, BX      ;AX gets bitwise AND of AX and BX
INC   AX          ;AX gets its original value plus 1
DEC   BX          ;BX gets its original value minus 1
MOV   AX, BX      ;AX gets values in BX
```

turns assembly to binary

ASSEMBLER

```
01 D8
29 D8
21 D8
40
4B
8B C3
```

LINKER

incase there are multiple parts in multiple files they are linked together

```
01 D8
29 D8
21 D8
40
4B
8B C3
```

LOADER

loads from hard-disk into the memory

logical address	*physical memory*	*physical address*
93ee:db1e	01	a19fe
93ee:db1f	D8	a19ff
93ee:db20	29	a1a00
93ee:db21	D8	a1a01
93ee:db22	21	a1a02
93ee:db23	D8	a1a03
93ee:db24	40	a1a04
93ee:db25	4B	a1a05
93ee:db26	8B	a1a06
93ee:db27	C3	a1a07

CPU: fetch-> decode-> execute (based on it’s pc pointer)

Instruction Set (AVR)

Syntax:		Operands:	Program Counter:
ADC	Rd, Rr	$0 \le d \le 31, 0 \le r \le 31$	PC ← PC + 1
ADD	Rd, Rr	$0 \le d \le 31, 0 \le r \le 31$	PC ← PC + 1
AND	Rd, Rr	$0 \le d \le 31, 0 \le r \le 31$	PC ← PC + 1
ANDI	Rd, K	$16 \le d \le 31, 0 \le K \le 255$	PC ← PC + 1
BREQ	k	$-64 \le k \le +63$	PC ← PC + k + 1 PC ← PC + 1, if condition is false
CBI	A, b	$0 \le A \le 31, 0 \le b \le 7$	PC ← PC + 1
CP	Rd, Rr	$0 \le d \le 31, 0 \le r \le 31$	PC ← PC + 1
DEC	Rd	$0 \le d \le 31$	PC ← PC + 1
IN	Rd, A	$0 \le d \le 31, 0 \le A \le 63$	PC ← PC + 1
INC	Rd	$0 \le d \le 31$	PC ← PC + 1
JMP	k	$0 \le k < 4M$	PC ← k
LDI	Rd, K	$16 \le d \le 31, 0 \le K \le 255$	PC ← PC + 1
MOV	Rd, Rr	$0 \le d \le 31, 0 \le r \le 31$	PC ← PC + 1
MUL	Rd, Rr	$0 \le d \le 31, 0 \le r \le 31$	PC ← PC + 1
OUT	A, Rr	$0 \le r \le 31, 0 \le A \le 63$	PC ← PC + 1
RJMP	k	$-2K \le k < 2K$	PC ← PC + k + 1
SBI	A, b	$0 \le A \le 31, 0 \le b \le 7$	PC ← PC + 1

Instruction Set (8088)

Instructions	Description	Notes
ADC	Add with Carry	destination := destination + source + carry_flag
ADD	Add without Carry	(1) r/m += r/imm; (2) r += m/imm;
AND	Logical AND	(1) r/m &= r/imm; (2) r &= m/imm;
JE	Branch if Equal	if (Z = 1) then PC ← PC + k + 1
JL	Branch if Lower	if (C = 1) then PC ← PC + k + 1
JNE	Branch if Not Equal	if (Z = 0) then PC ← PC + k + 1
JGE	Branch if Same or Higher	if (C = 0) then PC ← PC + k + 1
CALL	call Subroutine	push eip;
CLI	Global Interrupt Disable	I ← 0
CMP	Compare	
DEC	Decrement by 1	
DIV	Unsigned divide	DX:AX = DX:AX / r/m; resulting DX = remainder
XOR	Exclusive OR	(1) r/m ^= r/imm; (2) r ^= m/imm;
IN	Input from port	(1) AL = port[imm]; (2) AL = port[DX]; (3) AX = port[DX];
NOT	Negate the operand, logical NOT	r/m ^= -1;
INT	Call to interrupt	
INC	Increment by 1	
IRET	Return from interrupt	
JMP	Jump	
MOV	copies data from one location to another	(1) r/m = r; (2) r = r/m;
MUL	Multiply Unsigned	(1) DX:AX = AX * r/m; (2) AX = AL * r/m;
OR	Logical OR	(1) r/m |= r/imm; (2) r |= m/imm;
OUT	Output to port	(1) port[imm] = AL; (2) port[DX] = AL; (3) port[DX] = AX;
POP	Pop Register from Stack	r/m = *SP++;
PUSH	Push Register on Stack	*--SP = r/m;
RET	Subroutine Return	It will be translated to a RETN or a RETF
RETN	Return from near procedure	
RETF	Return from far procedure	
STI	Global Interrupt Enable	I ← 1
ROR	Rotate right	
ROL	Rotate left	

MOV Instructions

- **MOV instruction**
  - **MOV des, src ; copy source to destination**
  - **Examples:**
    - » **MOV CL,55H** source: عدد ۵۵ هگز
    - » **MOV DL, CL** این بار مبدا هم یک رجیستر است
    - » **MOV AH, DL**
    - » **MOV CX,EF28H**
    - » **MOV AX, CX**
    - » **MOV DI, AX** برای رجیسترهای ۱۶ بیتی هم می‌توان به همین شکل عمل کرد
    - » **MOV BP,DI**
  - **No MOV for flag register**
  - **No immediate load to segment register (only registers)**
  - **Same size (destination and source)**

ADD Instruction

- **ADD instruction**
  - **ADD des, src ; add the source to destination**
    dest += src
  - **Examples:**
    - » **MOV AL,55H**
    - » **MOV CL,23H**
    - » **ADD AL,CL ;** Al = 55H + 23H
    - » **MOV DH,25H**
    - » **ADD DH,34H ; immediate operand**
    - » **MOV CX,345H**
    - » **ADD CX,679H**
  - **Same size (destination and source)**

Debug program

Debug(windows 7)
appendix 30 book

- **R <register name>**
- **A <starting address>**
- **U <start> <end> or U <start> <L number>**
- **G < = starting address> <stop address(es)>**
- **T < = starting address> <number>**
- **F <s> <e> <data> or F <s> <L n> <data>**
- **D <s> <e> or D <s> <L n>**
- **E <address> <data list>**

فرض کنید این کل حافظه‌ی ما است برای مشخص کردن این که مثلا کد ما کجا است به سراغ code segment می‌رویم.
یعنی نمی‌خواهیم کل فضا را فقط به کد اختصاص دهیم. برای هر بخش ۶۴ بایت در نظر می‌گیریم و شروع هر بخش را با
پوینتری به سر آن که در همین سگمنت رجیسترها نگه می‌داریم.

Segmented Memory

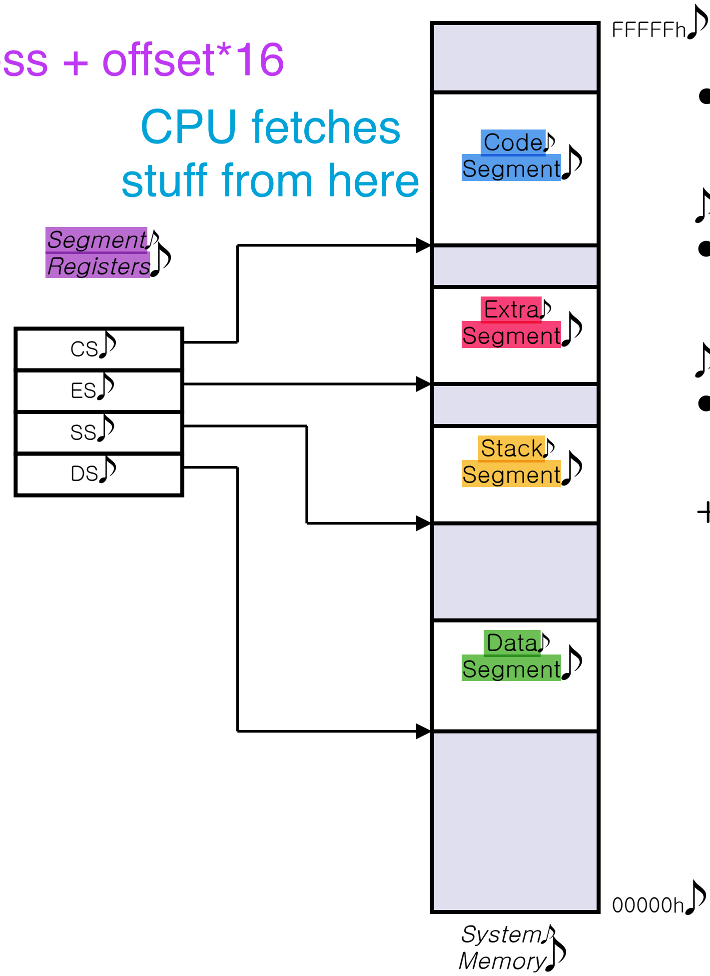
the parts separated by : then calculate the physical address like this:

segment address + offset*16

logical address = segment address: offset

offset = distance from the start of the segment

CPU fetches stuff from here



- Logical, Segmented Address:
  0FE6:012Bh
- Offset, Index Address:
  012Bh
- Physical Address:
  0FE60h → 65120
  + 012Bh → 299
  0FF8Bh → 65149

Program Segments

- Code
- Data
- Stack
- 80x86 segment registers
  - DS, CS, SS, ES
- Logical address, physical address
  - Physical: 20bit
  - Offset: 16 bit
  - Logical: segment+offset
- How to convert?
- Examples of code and data segments

The Stack

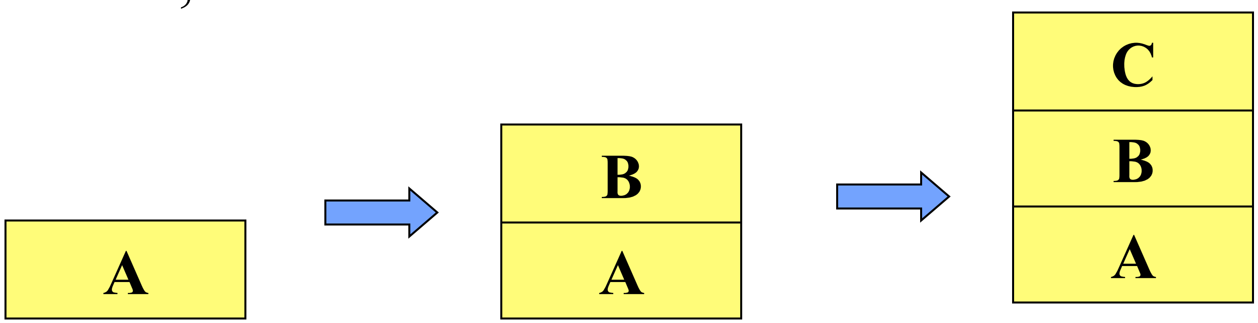
- **The stack is a memory area intended for storing temporary values.**
- **The stack is accessed by the SS:SP segment/ offset combination (StackSegment: StackPointer)**
- **Some instructions make use of the stack area during execution (*push*, *pop*, *call*, *ret*, many others)**
- **If you need to store temporary values in memory, the stack is the best place to do so.**

push: add to stack
pop: remove from the top of stack
call: we save our return address in stack

Data Storage via the Stack

The word '*stack*' is used because storage/retrieval of words in the stack memory area is the same as accessing items from a stack of items.

Visualize a stack of boxes. To build a stack, you place box A, then box B, then box C.



Notice that you only have access to the last item placed on the stack (the Top of Stack – TOS). You retrieve the boxes from the stack in reverse order (C then B then A).

Storing data on X86 stack via PUSH

- The SP (Stack Pointer) register is used to access items on the stack. The SP register points to the LAST value put on the stack.
- The PUSH operation stores a value to the stack:

۱۶ بیتی PUSH AX ; SP= SP-2, M[SP] ← AX

۱۶ بیتی - ۲ بایت - برای همین ۲ تا تغییر می‌دهیم

- The "push AX" instruction is equivalent to:

```
sub  SP, 2        ; decrement SP by 2 for word operation
mov [SP], AX      ; write value to stack.
```

- Stack access only supports 16-bit or 32-bit operations

Visualizing the PUSH operation

before PUSH AX

high memory

offset from ss = 8

lastval ← SP
????
????
????
????
????
????
????
????

stack segment is here

low memory

View memory as being 16 bits wide since stack operations are always 16 bit or 32 bits.

after PUSH AX

high memory

lastval
ahal ← SP
????
????
????
????
????
????
????

(new SP = old SP-2)

low memory

Multiple Pushes

before

high memory

lastval	← SP
????	
????	
????	
????	
????	
????	
????	
????	

low memory

PUSH AX
PUSH BX
PUSH CX

after all pushes

high memory

lastval	
ax	
bx	
cx	← SP
????	
????	
????	
????	
????	

low memory

Reading Data from X86 stack via POP

The POP operation retrieves a value from the stack:

POP AX ; $AX \leftarrow M[SP]$, $SP = SP+2$

The "pop AX" instruction is equivalent to:

```
mov AX, [SP]      ; read value from top of stack
add  sp, 2        ; increment SP by 2 for word operation
```

Visualizing the POP operation

before POP AX

high memory

FF65
23AB ← SP
????
????
????
????
????
????
????

low memory

View memory as being 16 bits wide since stack operations are always 16 bit or 32 bits.

after POP AX

high memory

FF65 ← SP
23AB
????
????
????
????
????
????
????

low memory

AX = 23AB

Visualizing multiple POP operations

before

high memory

FF65
23AB
357F
D21B
38AC ← SP
23F4
????
????
????

low memory

pop AX
pop BX
pop CX

after all POPs

high memory

FF65
23AB ← SP
357F
D21B
38AC
23F4
????
????
????

low memory

AX = 38AC
BX = D21B
CX = 357F

Stack Overflow, Underflow

- If you keep pushing data on the stack without taking data off the stack, then the stack can eventually grow larger than your allocated space
  - Can begin writing to memory area that your code is in or other non-stack data
  - This is called stack OVERFLOW
- If you take off more data than you placed on the stack, then stack pointer can increment past the 'start' of the stack. This is stack UNDERFLOW.
- Bottom line: You should allocate sufficient memory for your stack needs, and pop off the same amount of data as pushed in.

Stack (summary)

- **Temporary storage**
- **Segment and pointer SS:SP**
- **Push and Pop (LIFO)**
- **SP : top of the stack**
- **After push SP is decremented**

Summary

- Programs for 80x86
- Machine language, Assembly, …
- Registers, segments
- Instruction set
- Debug program
- Stack