

Assembly Language Level

- **mnemonic** version of machine code
 - symbolic names for opcodes and operands
 - symbolic addresses (labels)
 - pseudoinstructions to define/organize data
- **why not high-level language?**
 - access to all **hardware** (regs, flags, device controllers, ...)
 - **performance**
 - critical parts of code must be optimized (10-90 rule)
 - smaller **size** (important for embedded systems)

Lubomir Bic

1

Assembly Language Programming

- AL statement consists of:
 - label (optional)
 - opcode
 - zero or more operands
 - comments (optional)
- **Examples:**

```
START:  ADD CX, 20    !add constant 20 to register CX
        JMP  START    !jump to label START
```

Lubomir Bic

2

The 8088 Development Environment

- **tracer:** simulation systems for Intel 8088 processor
- **development process:**
 - write AL program, test.S, using editor (e.g. Notepad)
 - assemble it: `as88 test`
 - this produces several files
 - run it: `s88 test`
 - or trace it: `t88 test`
- **demo of tracer**

Lubomir Bic

3

8088 Programming

- programming can only be done incrementally by doing and experimenting
- **overview:**
 - addressing modes
 - memory organization
 - registers
 - instruction types
- examples to study, modify, or develop

Lubomir Bic

4

Addressing

- any machine instruction has 0 or more operands
- an operand can be: a **constant**, a **register**, an **address**
- **immediate**
 - `ADD CX, 20`
 - add constant 20 to register CX
 - operand, 20, is part of instruction
- **register**
 - `ADD CX, 20`
 - operand is in register CX
- **direct**
 - `ADD CX, (20)`
 - operand is in memory at address 20
 - note the difference between 20 and (20) !!

Lubomir Bic

5

Addressing

- **register indirect**
 - `ADD CX, (SI)`
 - content of SI is interpreted as a memory address
 - operand is in memory at address (SI)
- **register indirect with displacement** (aka indexed)
 - `ADD CX, LABEL(SI)`
 - operand is in memory at address LABEL+(SI)
- **register indirect with index** (aka based-indexed)
 - `ADD CX, (BX)(SI)`
 - operand is in memory at address (BX)+(SI)
- **register indirect with index and displacement**
 - `ADD CX, LABEL(BX)(SI)`
 - operand is in memory at address LABEL+(BX)+(SI)

Lubomir Bic

6

Memory Organization

- memory is divided into **segments**
- **code** segment: contains binary program
- **data** segment: contains all initial data
- **stack** segment: local variables, intermediate results
- **extra** segment: additional data, as needed
- starting address of each segment is in a **register**
 - CS, DS, SS, BS -- each register is 16 bits long
 - every memory reference uses one of the registers
 - if none specified, DS is assumed

Lubomir Bic

7

Registers

- **general**
 - AX, [AH, AL]: used as **accumulator**
 - BX, [BH, BL]: used as **base** register (reg indirect addressing)
 - CX, [CH, CL]: **counter**, used in loops
 - DX, [DH, DL]: **data** register, used with AX for **double** length
- **pointer registers**
 - SP: top of stack
 - BP: typically the base of current frame
- **index registers**
 - SI, DI: used in conjunction with BX (data) or BP (stack)

Lubomir Bic

8

Registers

- **IP: instruction pointer** (program counter)
 - modified automatically after each instruction
- **SF: condition codes**
 - Z: result is zero
 - S: result is negative
 - V: overflow
 - C: carry

Lubomir Bic

9

Basic Instruction Types

- **move data**
 - `MOV AX, 20` ! move constant 20 to AX
 - `PUSH 20` ! push constant 20 on stack
- **arithmetic**
 - `ADD SI, 2` ! add 2 to register SI
 - `ADDB AH, AL` ! add low-order byte of AX to high-order

Lubomir Bic

10

Pseudoinstructions

- `.SECT .TEXT` following lines go into code segment
- `.SECT .DATA` following lines go into data segment
- `.ASCII "str"` store str as ASCII string
 - Ex: `.ASCII "Hello World\n"`
- **symbolic constants: identifier = expression**
 - Ex: `SIZE = 1024`
`_WRITE = 4`

Lubomir Bic

11

Example

```

_EXIT = 1      ← system constants
_WRITE = 4
_STDOUT = 1
.SECT .TEXT    ← begin program
start:
  MOV  CX, de-hw ← de-hw=12 -- length of string, move to CX
                ← prepare system call: push args, then call SYS
  PUSH CX       ← push CX=12 on stack (# of chars)
  PUSH hw       ← push address of string on stack
  PUSH _STDOUT  ← push destination (standard output)
  PUSH _WRITE   ← push command
  SYS          ← issue sys call
  ADD  SP, 8    ← pop the stack (8 bytes)
  SUB  CX, AX   ← set up EXIT system call
  PUSH CX
  PUSH _EXIT
  SYS
.SECT .DATA    ← begin data section
hw:
.ASCII "Hello World\n"
de:

```

- now trace code again

Lubomir Bic

12

More Instruction Types

- labels
 - **global**: unique alphanumeric id followed by colon
 - Ex: START: ...
 JMP START
 - **local**: single digit within code followed by colon, not unique
 - Ex: 1: ...
 JMP 1b !jump backward to nearest label 1:
 JMP 5f !jump forward to nearest label 5:
- loops
 - LOOP L1 !decrement CX; jump to L1 if positive
- jumps
 - JNE L1 !jump to L1 if not equal zero (ZF=0)

Lubomir Bic

13

Example: HlloLoop

```

...
SUCCESS      = 1      ! 4 ← new constant
.sect .text
start:
MOV CX,de-hw  ! 7 ← CX is loop counter (12)
PUSH 1        ! 9 ← # characters to output
PUSH hw       ! 10 ← push address of character on stack
MOV BP,SP     ! 11 ← BP points to next char address (on stack)
PUSH _STDOUT  ! 12 ← push destination (standard output)
PUSH _WRITE   ! 13 ← push command
1: SYS        ! 14 ← issue sys call
SUB AX,SUCCESS ! 15 ← test if output successful (1 char)
JNE 1f        ! 16 ← if not then exit loop
INC (BP)      ! 17 ← point to next char
LOOP 1b       ! 18 ← decrement CX and continue loop if not done
1: ADD SP,8    ! 19 ← pop the stack (8 bytes)
PUSH AX       ! 20 ← set up EXIT system call
PUSH _EXIT
SYS
.sect .data
hw:
.ASCII "Hello World\n"
de:

```

Note **errors** in original code:

1. delete line 8
2. change SUB to ADD
3. change \\n to \n

Lubomir Bic

14

More Instructions/Pseudoinstr.

- arithmetic/logic
 - MUL (BX) !multiply AX by (BX), store in AX:DX
 - SHR CX,1 !shift right = divide CX by 2
- .WORD !assemble args as words (2B each)
 - Ex: .WORD 2, 10, 625 !6 bytes of data
 - integers are in "little-endian" representation:
 low-order byte stored in low-order address
 - 2, 10, 625 in Hex: 00 02, 00 0a, 02 71
 stored as: 02 00, 0a 00, 71 02
 - tracer does not show leading zeros: 2 0 a 0 71 2
- .SPACE n !reserve n bytes of space
- .ALIGN 2 !align next data on word boundary

Lubomir Bic

15

More Instruction Types

- string operations
 - LODS !load string (2B) into AX from (SI), incr. SI by 2
 !both AX and SI are implicit operands
 !SI is auto-incremented (decremented)
 - _PRINTF !system call, printf from Unix/Linux
 !print using a format: string with %d
 !format and values are all pushed on stack
 - Ex: .ASCIZ "The result is %d in decimal.\n"
 The result is 256 in decimal.

Lubomir Bic

16

More Instruction Types

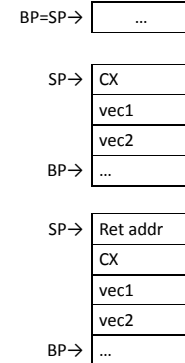
- subroutine calls
 - CALL L1 !push return address (PC) on stack
!jump to subroutine at label L1
!parameters needed by subroutine are
!pushed on stack before call
 - RET !return from subroutine using address on stack
- subroutine must create new stack frame:
 - save BP at entering the subroutine
 - start new stack frame: move BP to top of stack
 - restore BP upon return (discard stack frame)

Lubomir Bic

17

Example: vecprod

```
.SECT .TEXT
inpstart:
  MOV BP,SP      ← point BP to SP
  PUSH vec2      ← push vector addresses on stack
  PUSH vec1
  MOV CX,vec2-vec1 ← CX=vector length
  SHR CX,1       ← divide vector length by 2 and
  PUSH CX        ← push on stack
  CALL vecmul    ← call subroutine
  ...
  ...
.SECT .DATA
.ALIGN 2
vec1: .WORD 3,4,7,11,3
vec2: .WORD 2,6,3,1,0
```

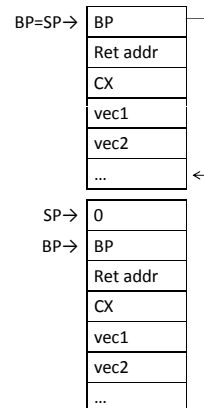


Lubomir Bic

18

Example: vecprod -- subroutine

```
vecmul:
  PUSH BP      ← save old BP
  MOV BP,SP    ← point BP to SP: start new frame
  MOV CX,4(BP) ← CX=vec length from stack
  MOV SI,6(BP) ← SI=starting address of vec1
  MOV DI,8(BP) ← DI=starting address of vec2
  PUSH 0       ← initial value of vector product
1: LODS        ← AX=(SI)=vec1[i], increment SI
  MUL (DI)     ← AX=AX*(DI)=vec1[i]*vec2[i]
  ADD -2(BP),AX ← add AX to stack at BP-2
  ADD DI,2     ← increment DI
  LOOP 1b      ← decrement CX and repeat
  POP AX       ← AX=vector product
  POP BP       ← Restore old BP
  RET         ← return
```

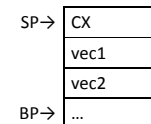


Lubomir Bic

19

Example: vecprod

```
.SECT .TEXT
inpstart:
  ...
  CALL vecmul
  MOV (inprod),AX ← store result in data segment
  PUSH AX         ← push result on stack
  PUSH pfmt       ← push format statement
  PUSH _PRINTF    ← push print command
  SYS             ← print
  ADD SP,12       ← clean stack
  PUSH 0          ← set up for exit
  PUSH _EXIT
  SYS
  ...
.SECT .DATA
pfmt: .ASCIZ "Inner product is %d!\n"
.ALIGN 2
vec1: .WORD 3,4,7,11,3
vec2: .WORD 2,6,3,1,0
.SECT .BSS
inprod: .SPACE 2
```



Lubomir Bic

20

More Instructions/Pseudoinstr.

- arithmetic/logic
 - SAL BX,1 !multiply by 2 (shift left)
 - CMP AX,5 !compare AX to constant 5:
!computes 5-AX, sets condition flags
 - CMPB AL,'7' !compare byte to ASCII character
- jumps
 - JL L1 !jump to L1 on less than 0
 - JLE L1 !jump to L1 on less than or equal to 0
- system call
 - _GETCHAR !get byte from input into AL

Lubomir Bic

21

Example: Dispatch Table

- input octal digits; print message for each
- print error message if not 0-7

jumpstr:

```

...
PUSH _GETCHAR
1: SYS          ← AX = next character (AH=0, AL=char)
CMP AX,5       ← if less than 5 (ascii chars 00-04 = EOF) then
JL 8f          ← go to exit
CMPB AL,'0'    ← if less than char '0' then
JL 1b          ← ignore
CMPB AL,'9'    ← if greater than '9'
JLE 2f
MOVB AL,'9'+1  ← then set to char ':', i.e. 1 above '9'
2: MOV BX,AX   ← copy AX to BX (BX has ascii 30, 31, 32, ..., 39, 3A)
AND BX,0xf    ← mask out high-order byte, 3 (BX has digits 0-A)
SAL BX,1      ← multiply by 2 (by shifting left)
CALL tbl(BX)  ← tbl has list of routines indexed by BX: 0-20 in decimal
JMP 1b        ← start over
8: PUSH 0     ← exit
PUSH _EXIT
SYS

```

Lubomir Bic

22

Example: Dispatch Table

```

rout0: MOV AX,mes0 ← Each routine prints a different message:
        JMP 9f      Load message addr to AX, go to print it, and return
rout1: MOV AX,mes1
        JMP 9f

```

```

...
rout8: MOV AX,mes8
        JMP 9f

```

```

erout: MOV AX,emes
9:  PUSH AX
    PUSH _PRINTF
    SYS
    ADD SP,4
    RET

```

Multiway jump from program:
CALL tbl(BX)

```

tbl: .WORD rout0,rout1,...,rout7,rout8,rout8,erout ← addresses of routs (2B each)
mes0: .ASCIZ "This is a zero.\n"
mes1: .ASCIZ "How about a one.\n"
...
mes8: .ASCIZ "This digit is not octal.\n"
emes: .ASCIZ "This is not a digit. Try again.\n"

```

Lubomir Bic

23

More Instructions

- more string operations
 - LODS !load string (2B) into AX from (SI), incr. SI by 2
 - LODSB !load string (1B) into AL from (SI), incr. SI by 1
 - STOS !store string (2B) from AX into (SI), incr. SI by 2
 - STOSB !store string (1B) from AX into (SI), incr. SI by 1
- direction depends on flag D in SF register:
 - if D=0 (shown as ">" in tracer) then SI is incremented
 - if D=1 (shown as "<" in tracer) then SI is decremented
 - D is set/reset with STD/CLD
 - default is 0 (increment)
- next example shows use of STOSB for char I/O

Lubomir Bic

24

Example: input/output

```

...
asciinl = 10      ← 'new line' char (ascii 0A)
EOF = '.'         ← define some end of file marker, e.g. the period
.SECT             .TEXT
start:
  MOV  DI,STR      ← DI points to STR
  PUSH _GETCHAR
1: SYS             ← AL = next input char
  CMP  AX,EOF      ← if end of input then
  JE   9f           ← go to exit
  STOSB            ← store char from AL to (DI), i.e., STR, and increment DI
  CMPB AL,asciinl  ← if not new line char
  JNE  1b           ← then go back for more input
  MOVB (DI),0      ← else, write trailing string byte (0) into STR
  PUSH STR         ← print string
  PUSH _PRINTF
  SYS
  ADD  SP,4        ← pop stack and start over
  MOV  DI,STR
  JMP  1b
9: ...            !exit
.SECT             .BSS
STR:             .SPACE 80

```

Lubomir Bic

25