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)

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

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

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
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)

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

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)

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)
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

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

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
”**: ASCII string
- **symbolic constants: identifier = expression**
  - Ex: **SIZE = 1024**
  - **_WRITE = 4**

Example

```
.EXIT = 1
_WRITE = 4
_STDOUT= 1

.start:
    MOV CX, de-hw ← de-hw=12 ← length of string, move to CX
    push 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 ← begin program

.now trace code again
```

- **sys**: system constants
- **begin program**: now trace code again
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)

---

Example: HlloLoop

```assembly
Example: HlloLoop

- SUCCESS = 1 | 4 ← new constant
- .SECT .TEXT | 5
- start: | 6
    - MOV CX,de-hw | 7 ← CX is loop counter (12)
    - PUSH 1 | 8 ← # characters to output
    - PUSH hw | 9
    - MOV BP,SP | 10 ← BP points to next char address (on stack)
    - PUSH _STDOUT | 11 ← push destination (standard output)
    - PUSH _WRITE | 12 ← push command
    - JNE 1f | 13 ← # characters to output successful (1 char)
    - SYS AX,SUCCESS | 14 ← issue sys call
    - INC (BP) | 15 ← if not then exit loop
    - INC (BP) | 16 ← point to next char
    - LOOP 1b | 17 ← decrement CX and continue loop if not done
    - ADD SP,8 | 18 ← pop the stack (8 bytes)
    - PUSH AX | 19 ← set up EXIT system call
    - PUSH _EXIT | 20
    - JMP 1b | 21
    - JMP 5f | 22
- .SECT .DATA | 23
- hw: | 24
    - .ASCIZ "Hello World\n" | 25
- .SPACE n | 26 ← reserve n bytes of space
- .ALIGN 2 | 27 ← align next data on word boundary
```

---

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 | 16 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

---

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 (decemented)
  - _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.
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)

Example: vecprod

vecmul:

```
BP=SP
BP
...
BP
...
BP
...
```

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 DL,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
```

Example: vecprod

```
 منتخب .SECT .TEXT
instart:
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

... ← ...
```

Example: vecprod

```
منتخب .SECT .TEXT
instart:
CALL vecmul ← store result in data segment
PUSH AX ← push result on stack
PUSH fmt ← push format statement
PUSH _PRINTF ← push print command
SYS ← print
ADD SP,12 ← clean stack
PUSH 0 ← set up for exit
PUSH _EXIT ← SYS
... ← ...

.SEG .DATA
.Format: ASCIZ "Inner product is %d\n"
.ALIGN 2
vec1: .WORD 3,4,7,11,3
vec2: .WORD 2,6,3,1,0
.SEG .BSS
inprod: .SPACE 2

... ← ...
```

Example: vecprod

```
منتخب .SECT .TEXT
instart:
CALL vecmul ← ...
MOV [inprod],AX ← ...
PUSH AX ← ...
PUSH fmt ← ...
PUSH _PRINTF ← ...
PUSH _EXIT ← ...
SYS ← ...

.SEG .DATA
fmt: ASCIZ "Inner product is %d\n"
.ALIGN 2
vec1: .WORD 3,4,7,11,3
vec2: .WORD 2,6,3,1,0
.SEG .BSS
inprod: .SPACE 2

... ← ...
```
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

Example: Dispatch Table

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

jumpstr:

1: PUSH _GETCHAR
   SYS
   CMP AX,5
   JL 8f
   CMPB AL,'7'
   JL 1b
   CMPB AL,'9'
   JL 9f
   MOV AL,'9'+1
   CALL tbl(BX)
   JMP 1b
   MOV 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,0F
   ← 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
SYS

Example: Dispatch Table

- each routine prints a different message:
- load message addr to AX, go to print it, and return

rout0: MOV AX,mes0   ← each routine prints a different message:
       JMP 9f
rout1: MOV AX,mes1
       JMP 9f
rout8: MOV AX,mes8
       JMP 9f
erout: MOV AX,emes
       9:   PUSH AX
             PUSH _PRINTF
             ADD SP,4
             RET

tbl: .WORD rout0,rout1,...,rout7,rout8,rout9,erout
     ← addresses of routines (2B each)
mes0: .ASCIZ "This is a zero.\n"
mes1: .ASCIZ "This is a digit. Try again.\n"
mes8: .ASCIZ "This digit is not octal.\n"
emes: .ASCIZ "This is not a digit. Try again.\n"

Multiway jump from program:

CALL tbl(BX)

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
Example: input/output

```assembly
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
    CMP AX,EOF    ← AL = next input char
    JE 9f         ← if end of input then
    STOSB AL,asciinl
    CMPB AL,asciinl
    JNE 1b        ← store char from AL to (DI), i.e., STR, and increment DI
    JMP 1b        ← if not new line char

    MOVB (DI),0    ← then go back for more input
    PUSH STR       ← else, write trailing string byte (0) into STR
    PUSH _PRINTF
    SYS
    ADD SP,4       ← print string
    MOV DI,STR
    JMP 1b         ← pop stack and start over

  9: ...           ← exit

.SECT .BSS
STR: .SPACE 80    ← pop stack and start over

Lubomir Bic
```