COMP 8551 Advanced Games Programming Techniques

Borna Noureddin, Ph.D.

British Columbia Institute of Technology

Assembly Language

Assembly Language

- Human-readable notation (second-generation language) for machine language (first-generation language)
- High-level languages (FORTRAN, C, C++, BASIC, etc.) are third-generation languages, while languages that produce high-level language code (e.g., visual tools) are considered fourth-generation languages
- With the move to interpretive code, frameworks, etc., that terminology is not commonly used anymore

Assembly Language

Bits: 1011000001100001

Turns series of transistors on/off

Indicates to CPU (or microcode) to:

“move value 0x61 to the register at location 1”

Assembly language for this might be something like:

“MOV 061h, R1”

Assembler: assembly → machine language

Disassembler: machine language → assembly

Assembly Language

Common types of instructions:

- Move
  - set register to fixed constant value
  - move data between memory location and register
  - read/write data to/from hardware devices
- Compute
  - add/subtract/multiply/divide values of two registers (result placed in register)
  - perform bitwise operations
  - compare two values in registers

Assembly Language

Common types of instructions:

- Program flow
  - jump to another location in program (address)
  - jump to another location if a certain condition holds
  - jump to another location, but save location of current next instruction (function call – may also save other information on a "stack")

Assembly Language

Common types of instructions:

- Complex instructions
  - save many registers on the stack at once
  - move large blocks of memory
  - complex and/or floating-point arithmetic (sine, cosine, square root, etc.)
  - perform atomic test-and-set instruction
  - combine ALU with an operand from memory rather than a register
  - SIMD instructions are a good example

Assembly Language

Common usage

- Historically: entire programs
  - Lotus 123
  - Console games from 1990s (Sega, Super NES, etc.)
  - Only way to write games for early PCs and game consoles (e.g., “high-res” games for Sinclair computers, Commodore, Adam, Intellivision, Atari, etc.)
- Debate still open whether modern compilers obviate need entirely for assembly language (although there are far fewer cases where it is worth it, especially given complexity of modern CPUs)

Assembly Language

Common usage

- More current applications still requiring assembly language:
  - device drivers
  - O/S kernel code
  - system BIOS
  - firmware
  - embedded systems
    - robotics
    - industrial control systems
    - security systems
    - sensors
    - medical equipment
    - flight navigation systems

Assembly Language

Common usage

- More current applications still requiring assembly language:
  - new or specialized processor for which good compiler does not yet exist
  - self-modifying code (e.g., function loader)
  - compilers
  - real-time 3D graphics applications that are <1MB and run on 1MHz system (Commodore64!)
- Although shading languages are not strictly assembly language, they follow the same basic concept (closer to the hardware, instructions rather than statements, etc.)

Assembly Language: x86

Variables

```
myvar1          DB 3
```

Sets aside single byte of memory and initializes it to 3

Can then be referred to by name myvar1

myvar1 represents address of memory

```
anothervar      DW 03FAh
```

Sets aside a word of data (2 consecutive bytes) containing value

```
someval         DD 721099

repeatvar       DB 7 dup(12,28)
```

Makes use of dup operator to set aside 14 bytes of data and initialize it to 7 copies of two bytes 12, 28

Useful for declaring arrays of bytes, words, etc., initialized to 0 (e.g., `myarr DD 100 dup 0`*)*

```
string1         DB 'This is a string'
```

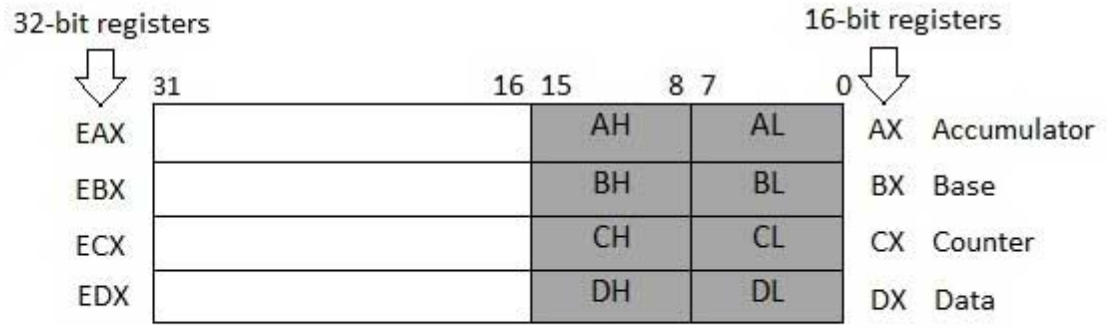
Sets aside 16 bytes of data and sets contents to be equal to ASCII values corresponding to letters of given string

Assembly Language: x86

Registers – Data Registers

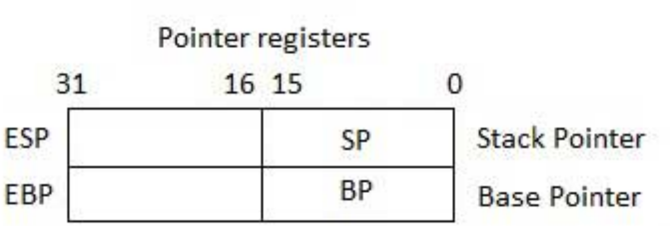
- Four 32-bit registers: EAX, EBX, ECX, EDX
- Can also access lower 16-bits: AX, BX, CX, DX
- Can access each 8-bit segment of each register:

AH, AL, BH, BL, CH, CL, DH, DL



Assembly Language: x86

Registers – Data Registers



Pointer registers

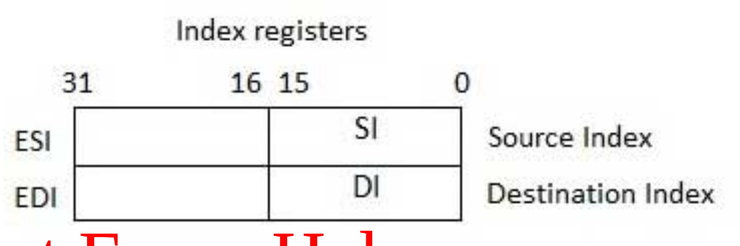
	31	16 15	0	
ESP		SP		Stack Pointer
EBP		BP		Base Pointer

- Instruction Pointer (IP)
  - Stores offset address of next instruction to be executed

- Stack Pointer (SP)
  - Provides offset within program stack
- Base Pointer (BP)
  - Helps in referencing parameter variables passed to subroutine

Assembly Language: x86

Registers – Index Registers



- Source Index (SI)
  - Source index for string operations
- Destination Index (DI)
  - Destination index for string operations

Assembly Language: x86

Instructions – examples

```
INC COUNT         ; Increment the memory variable COUNT

MOV TOTAL, 48     ; Transfer the value 48 in the
                  ; memory variable TOTAL

ADD AH, BH        ; Add the content of the
                  ; BH register into the AH register

AND MASK1, 128    ; Perform AND operation on the
                  ; variable MASK1 and 128

ADD MARKS, 10     ; Add 10 to the variable MARKS

MOV AL, 10        ; Transfer the value 10 to the AL register
```

Assembly Language: x86

Instructions – MOV

- Can use same instruction to move data from memory to registers and vice versa
- Cannot move data from memory to memory with MOV instruction
- Move data at byte memory location called **`myvar`** into **`AH`**:

```
MOV AH,[myvar]
```

- Note: square brackets means move actual data into **`AH`**, not address of data

Assembly Language: x86

Instructions – MOV

- Source and destination must be of matching sizes
  - E.g., cannot move data from variable declared as byte of data into 16 or 32 bit register
- But can be easily overridden: e.g., if `myvar1` is byte variable location:

```
MOV word AX,[myvar1]
```

will move byte at address `myvar1` and next byte into `AX`

- Similar overrides for moving byte and double word of data (denoted byte and dword respectively)

Assembly Language: x86

Instructions – MOV

- Can also do reverse (move data from register to memory):

```
MOV [myvar1],CH
```

- To move address of variable myvar2 into EAX register:

```
MOV EAX,myvar2
```

- EAX register now a *pointer* to myvar2 (does not contain *contents* of myvar2, but the *address* of myvar2)

Assembly Language: x86

Instructions – MOV Example

- Once moved address into 32 bit register, can move it into double word variable for storage

- EAX has been loaded with address of some memory location storing byte of data

- `mypoint` is double word variable to store address

```
MOV [mypoint],EAX
```

Assembly Language: x86

Instructions – MOV Example

- What if we wanted to load contents of memory location now pointed to by `mypoint` into CH register?

- First retrieve address from storage:

```
MOV EBX,[mypoint]
```

- Now EBX points to desired location.
- To retrieve byte of data at that location:

```
MOV CH,[EBX]
```

- Here square brackets do not denote contents of EBX itself but rather contents of location *pointed to* by EBX

Additional Reading

http://www.computernostalgia.net/articles/assembly.htm

http://en.wikipedia.org/wiki/Assembly_language#Current_usage

https://software.intel.com/en-us/articles/optimizing-the-rendering-pipeline-of-animated-models-using-the-intel-streaming-simd-extensions

http://en.wikipedia.org/wiki/SIMD

https://www.tutorialspoint.com/assembly_programming/index.htm

END