Lecture 07

Assembly Language & HW/SW Integration

x86 Registers · Instructions · cdecl Convention · Stack Frames · Boot Sequence

SENG 21213 · Computer Architecture & Operating Systems

Learning Objectives — By the end of this lecture you should be able to:

- 1. Identify all x86-32 general-purpose registers and their sub-registers and primary uses
- 2. Write x86-32 NASM assembly using data movement, arithmetic, logic, and control flow instructions
- 3. Trace the complete cdecl calling convention through 8 steps including stack frame layout
- 4. Draw a stack frame diagram showing saved EBP, return address, arguments, and local variables
- 5. Describe the 10-step boot sequence from CPU reset to kernel_main() execution
- 6. Explain Real Mode, Protected Mode, and Long Mode and the CR0 bit 0 transition mechanism

Section 1 · Assembly Language — x86-32 Fundamentals

📖 Stallings Reference

- Chapter 12: Processor Structure and Function
- Understanding assembly is essential for OS kernel programming, debugger use, and security analysis

1.1 x86-32 Register Set

Register	Size	Primary Use	Sub-registers
EAX	32-bit	Accumulator; return value from functions; arithmetic	AX(16), AH(8, high byte), AL(8, low byte)
EBX	32-bit	Base pointer for data segments; callee-saved	BX, BH, BL
ECX	32-bit	Counter for loops (REP prefix); shift count	CX, CH, CL
EDX	32-bit	Data; high word of MUL/DIV result; I/O port address	DX, DH, DL

Register	Size	Primary Use	Sub-registers
ESI	32-bit	Source index for string operations (MOVS, CMPS)	SI
EDI	32-bit	Destination index for string operations	DI
ESP	32-bit	Stack Pointer — points to top of stack (decrements downward)	SP
EBP	32-bit	Base Pointer — points to base of current stack frame	BP
EIP	32-bit	Instruction Pointer (Program Counter) — not directly writable	IP
EFLAGS	32-bit	Status flags: CF(carry), ZF(zero), SF(sign), OF(overflow), DF(direction), IF(interrupt)	individual bits

1.2 Core x86-32 Instructions (NASM syntax)

```
section .data
    msg db 'SENG OS', 10, 0       ; null-terminated string

section .text
global _start
_start:
    ; ── Data Movement ──────────────────────────────
    mov eax, 42               ; immediate → register
    mov ebx, eax              ; register → register
    mov eax, [0x1000]         ; memory → register (load)
    mov [0x2000], ebx         ; register → memory (store)
    movzx eax, byte [0x500]   ; zero-extend byte to 32-bit
    movsx eax, byte [0x500]   ; sign-extend byte to 32-bit
    xchg eax, ebx             ; atomic swap (implicit LOCK)

    ; ── Arithmetic ─────────────────────────────────
    add  eax, ebx             ; EAX = EAX + EBX; sets ZF, CF, OF, SF
    sub  eax, 10              ; EAX = EAX - 10
    imul ebx                  ; EDX:EAX = EAX * EBX (signed)
    idiv ecx                  ; EAX = EDX:EAX / ECX (signed quotient)
    inc  ecx                  ; ECX = ECX + 1 (does NOT set CF)
    dec  ecx                  ; ECX = ECX - 1
    neg  eax                  ; EAX = 0 - EAX (2's complement negate)

    ; ── Logic & Bit Manipulation ──────────────────
    and  eax, 0xFF            ; mask lower 8 bits; clear upper 24
    or   eax, 0x01            ; set bit 0
    xor  eax, eax             ; EAX = 0 (fastest way to zero a register)
    not  eax                  ; bitwise complement
    shl  eax, 3               ; EAX = EAX * 8 (fast multiply by power of 2)
    shr  eax, 1               ; EAX = EAX / 2 (unsigned right shift)
    sar  eax, 1               ; EAX = EAX / 2 (signed arithmetic right shift)

    ; ── Control Flow ──────────────────────────────
```

```
    cmp  eax, ebx            ; compute EAX-EBX; set flags; discard result
    je   equal               ; jump if ZF=1 (EAX == EBX)
    jne  notequal            ; jump if ZF=0
    jl   less                ; jump if SF≠OF (signed: EAX < EBX)
    jge  greater_or_eq       ; jump if SF=OF (signed: EAX >= EBX)
    jb   below               ; jump if CF=1 (unsigned: EAX < EBX)
    jmp  loop_start          ; unconditional jump
    call my_function         ; PUSH EIP+n; JMP my_function
    ret                      ; POP EIP (return to caller)
```

Section 2 · The Stack and Calling Conventions

2.1 Stack Operation Model

The x86 stack grows DOWNWARD (toward lower addresses). ESP always points to the CURRENT TOP (lowest occupied address). Key invariant: the stack must be 16-byte aligned before a CALL instruction on x86-64 (4-byte aligned on x86-32 cdecl).

Instruction	Pseudo-code	Effect on ESP
PUSH src	ESP ← ESP - 4; Memory[ESP] ← src	ESP decreases by 4
POP dst	dst ← Memory[ESP]; ESP ← ESP + 4	ESP increases by 4
CALL addr	PUSH EIP; JMP addr	ESP decreases by 4 (return address saved)
RET	POP EIP	ESP increases by 4
ENTER n,0	PUSH EBP; MOV EBP,ESP; SUB ESP,n	Creates stack frame; reserves n bytes for locals
LEAVE	MOV ESP,EBP; POP EBP	Tears down stack frame; restores ESP and EBP

2.2 cdecl Calling Convention — 8 Steps

The cdecl (C declaration) convention is the standard for 32-bit C on Linux and Windows. The CALLER is responsible for cleaning up pushed arguments.

1. Caller pushes arguments RIGHT-TO-LEFT: PUSH arg3; PUSH arg2; PUSH arg1.
2. Caller executes CALL func — pushes return address (EIP) and jumps to func.
3. Callee prologue: PUSH EBP; MOV EBP, ESP (save and set frame pointer).
4. Callee allocates locals: SUB ESP, n (n = total bytes for local variables).
5. Callee accesses arguments via positive EBP offsets: [EBP+8]=arg1, [EBP+12]=arg2, etc.
6. Callee accesses locals via negative EBP offsets: [EBP-4]=local1, [EBP-8]=local2.
7. Callee places return value in EAX (32-bit) or EDX:EAX (64-bit value).
8. Callee epilogue: LEAVE (or MOV ESP,EBP; POP EBP); RET. Caller does ADD ESP, n×4.

```
;;; int add(int a, int b) { return a + b; }  ← C source
add:
    push ebp              ; save caller's frame pointer
    mov  ebp, esp         ; EBP = frame pointer for this function
    ; Stack layout at this point:
```

```
;   [EBP - ?]  locals (none in this function)
;   [EBP + 0]  saved EBP
;   [EBP + 4]  return address (pushed by CALL)
;   [EBP + 8]  argument a  (first pushed → highest address)
;   [EBP + 12] argument b  (last pushed → lower address)
mov  eax, [ebp + 8]   ; EAX = a
add  eax, [ebp + 12]  ; EAX = a + b   ← return value in EAX
pop  ebp              ; restore caller's frame pointer
ret                   ; return to caller (cdecl: caller does ADD ESP,8)
```

2.3 Calling Convention Comparison

Convention	Arguments	Stack Cleanup	Return	Used By
cdecl	Right-to-left on stack	Caller (ADD ESP,n)	EAX	C functions, vararg (printf)
stdcall	Right-to-left on stack	Callee (RET n)	EAX	Win32 API functions
fastcall (x86)	ECX, EDX; rest on stack	Callee	EAX	Performance-critical Windows code
System V AMD64 (Linux)	RDI,RSI,RDX,RCX,R8,R9; rest on stack	Caller	RAX	64-bit Linux/macOS
Microsoft x64 (Windows)	RCX,RDX,R8,R9 + 32-byte shadow; rest on stack	Caller	RAX	64-bit Windows

Section 3 · HW/SW Interface — Boot Sequence and CPU Modes

3.1 x86 CPU Modes

Mode	Addressing	Memory Limit	Protection	Use
Real Mode	Segment:Offset (20-bit)	1 MB (2^20 bytes)	None — flat memory	BIOS, bootloader stage 1
Protected Mode (32-bit)	Virtual addresses via GDT/LDT	4 GB (2^32 bytes)	Ring 0–3; segmentation + paging	32-bit OS kernels
Long Mode (64-bit)	64-bit virtual addresses; paging only	128 TB virtual	Ring 0 & 3; paging mandatory	64-bit OS (Linux, Windows 10+)
System Management Mode	Physical addresses	Platform-specific	Separate SMM RAM	BIOS power management; transparent to OS

3.2 Boot Sequence — Power-On to Kernel

9. Power-on: CPU reset vector = 0xFFFFFFF0. CPU executes BIOS/UEFI from ROM.
10. POST (Power-On Self-Test): BIOS tests CPU registers, memory, I/O controller.

11. BIOS finds bootable device. Loads first 512 bytes (Master Boot Record, MBR) to 0x7C00 in RAM.
12. Bootloader executes at 0x7C00 in Real Mode. Loads kernel sectors using INT 0x13 (BIOS disk service).
13. Bootloader sets up the Global Descriptor Table (GDT) — defines code/data segments.
14. Bootloader sets CR0 bit 0 = 1 — switches CPU from Real Mode to Protected Mode.
15. Far jump (JMP 0x08:label) flushes the instruction pipeline and reloads CS with 32-bit code segment selector.
16. Segment registers (DS, ES, SS) loaded with data segment selector.
17. Stack pointer ESP set to the kernel stack top.
18. kernel_main() called — the OS is now in charge of the hardware.

★ OS Assignment — Stage 0 Boot Details

- boot.asm loads 64 disk sectors (32 KB) from LBA using INT 0x13 AH=0x42 (Extended Read).
- The GDT has 3 entries: null descriptor, code segment (base=0, limit=4GB, Ring 0, execute/read), data segment (base=0, limit=4GB, Ring 0, read/write).
- After setting CR0 bit 0, a far jump is mandatory — it reloads CS and forces the CPU to recognise the new 32-bit code segment.
- In kernel_entry.asm: MOV AX, 0x10 (data segment selector); MOV DS,AX; MOV ES,AX; MOV SS,AX; MOV ESP, 0x90000 (top of kernel stack).

Section 4 · Practice Exercises

✎ Exercise 7.1 [14 marks] — Stack Frame Tracing

- (a) [6] A C function int calc(int x, int y, int z) { int tmp = x + y; return tmp * z; } is called with arguments 3, 5, 7. Trace the complete cdecl calling sequence:
- (i) Show the PUSH instructions the caller executes (include values pushed).
- (ii) Draw the stack layout at the start of the function body, showing every stack cell with its address (relative to EBP) and value.
- (iii) Show the assembly instructions for the function body and indicate where EAX is set.
- (b) [4] A buffer overflow occurs in: void vuln() { char buf[16]; gets(buf); }. If the input is 20 bytes of 'A' followed by the 4-byte little-endian address 0x0804ABCD:
- (i) How many bytes of buf are overflowed?
- (ii) What memory cell is overwritten by the last 4 bytes?
- (iii) What happens when the function returns?
- (c) [4] Rewrite the vuln() function in safe assembly using MOV instructions limited to 16 bytes. What modern OS/hardware mechanism would detect this overflow even in the unsafe version?

✎ Exercise 7.2 [10 marks] — Boot Sequence and Modes

- (a) [4] Why is it necessary to execute a FAR JUMP immediately after setting CR0 bit 0 when switching from Real Mode to Protected Mode? What would happen if the jump were omitted? Name the CPU register that must be reloaded and explain why.
- (b) [3] The Real Mode segment:offset addressing computes physical address as segment × 16 +

offset. (i) What physical address is 0x07C0:0x0000? (ii) What physical address is 0x0000:0x7C00? (iii) Are these the same? What does this tell you about MBR loading?

- (c) [3] Your kernel.c implements a VGA text mode driver. VGA text buffer is at physical address 0xB8000. Each character cell is 2 bytes (ASCII + colour). (i) What protected-mode flat address is 0xB8000? (ii) Write a C expression to write the red letter 'A' to row 0, column 0. (iii) What video memory address corresponds to row 2, column 10?