ARM Assembly Programming

Module 1

Template & Setup

- GitHub - https://github.com/drmilhous/templateMake
- Pre-req libraries - *sudo apt install nasm gcc-multilib vim*
- *Creating Own Templates*
  - *genMake.sh MyProject*
- Creating a program
  - Create a project -
    - $ genMake Hello
  - Cd projects/Hello
  - Add a string at the top -
    - hi db "hi",10,0 (note
  - Add a call to print string
    - mov eax, hi
    - Call print string
  - Compile, run, repeat
    - Make
    - ./Hello

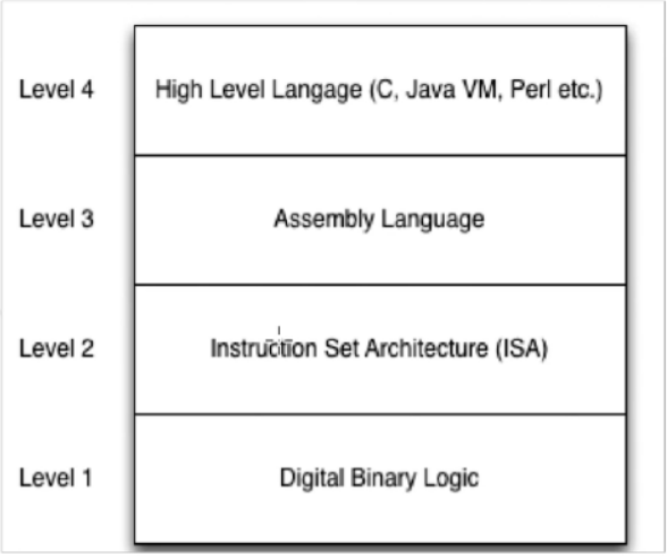
Introduction to Assembly

- Assembler – NASM
- Level 2 – ISA
  - Defined by processor makers
  - INTEL, ARM & SPARC
- Level 3 – Assembly
  - Assembly instructions transformed to ISA
- Data Representation – Binary (1 or 0)
- Converting binary to decimal – ADD respective INT

128	64	32	16	8	4	2	1
0	1	1	0	0	0	1	0

- Hexadecimal (Base 16)
  - 0 to 9 & A to F
- 2's Complement
  - Represents –ve numbers.
  - Reverse bits and add '1'.

Dec	Hex	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

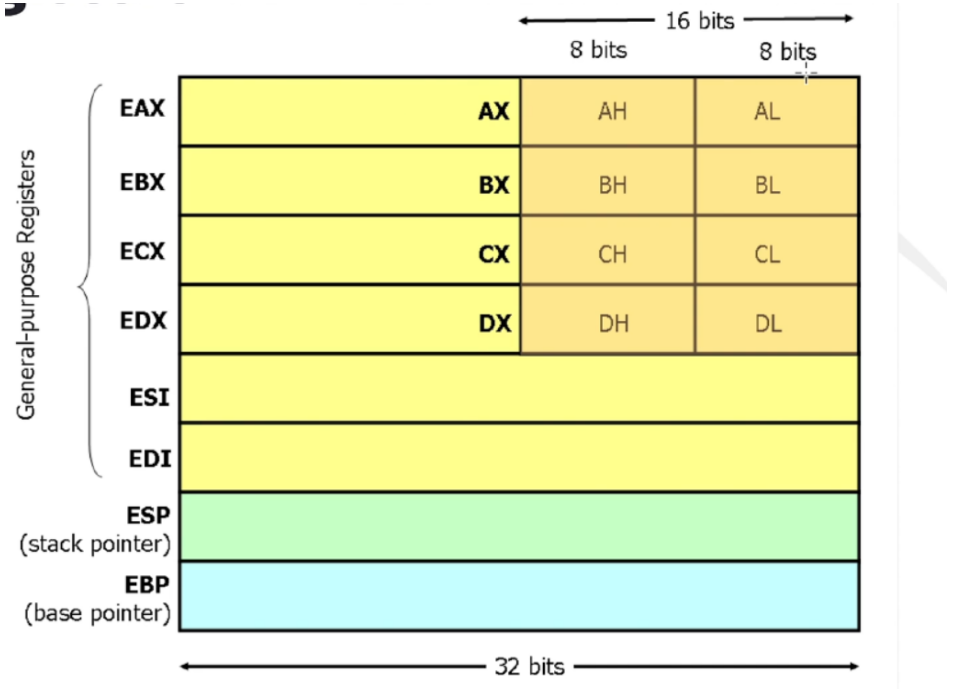


ASCII – for american english
- 8 bits for characters
- For letters/numbers MSB = 0
- ASSEMBLY can convert from char to byte

Unicode – for other languages
- UTF-8, UTF-16 & UTF-32

Computer Organization

- Computer Composition – Registers, Flags, Memory – Memory Addresses, IO Function & Computing Units.
- Basic unit of memory – 1 byte.
- CPU
  - CPU executes machine code – Fetch, Decode, Execute & Store
  - ALU
  - FLU
- Registers
  - Larger registers are broken into multiple parts
  - EAX – 32 bits
  - AX – Low 16 bits of EAX
  - AL – Low 8 bits of AX or EAX
  - AH – High 8 bits of AX
- Register Usage -
  - EAX for Accumulator and returns & ECX for counter
  - ESP for Extended Stack Pointer
  - ESI – for Source, EDI – for Destination : DATA TRANSFER
  - EBP for frame pointer & EDX : EAX mul and divide
  - IP/EIP - points to the current instruction to be executed. Cannot be modified directly



Flags -

- Carry, Overflow, Sign, Zero, Auxiliary Carry & Parity.
- Store info about previous instruction.

Segment Registers -

- Store info of where elements are stored
- Segments – Code (CS) , Data (DS), Stack (SS) & ES, FS, GS.

Processor Arch Family, Registers and Real/Protected Mode

- 8088, 8086 – 16-bit registers; 80286 – Protected Mode; 80386 – 32-bit Mode; 80486 – Faster; MMX – Instr Multi-Media & Modern – 64-bit & AES.
- Real Mode -
  - To address memory – 20 bits for address, --> Two 16-bit numbers, --> Selector & Offset, --> Shift to get address.
  - Wastes memory and address are not unique – NO VIRTUAL MEMORY.
  - Example – shift 4 bit and add a 16-bit offset --> 8000:0250 --> 80000h + 250h = 80250h read address.
- Protected Mode -
  - Implements virtual memory – load memory as needed.
  - 16-bit selector is index in descriptor table.
  - 32-bit – more memory around 3 gig. Use page instead of segments.
  - 64-bit – Address 16 Exabytes.
  - Example – 32 bit address --> '0' to 'FF FF FF FF' = 4GB --> Segment Registers – Point to descriptor tables, Global Descriptor Table & Local Descriptor Tables.
- Paging – Not all memory is loaded
  - Page Fault – Process is suspended --> page is loaded --> page is loaded.
- Interrupts
  - Hardware, Software like illegal memory access & timer and Errors/Traps - div by zero, illegal instruction and illegal memory access.

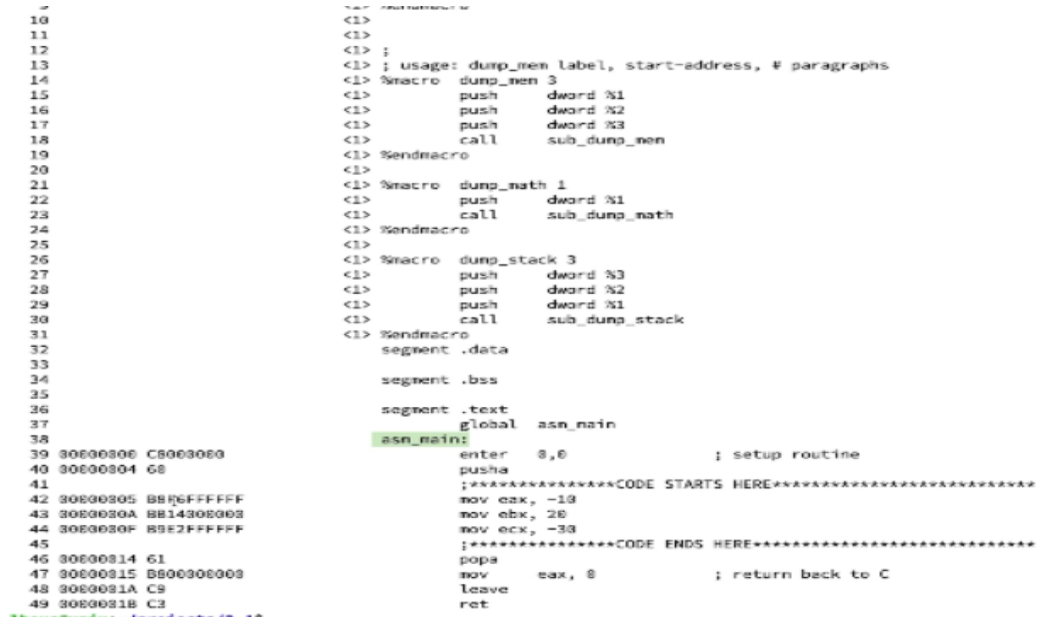
Data Representation

- Binary – 1's/0's - Each BIT represents the power two.
- 2's Complement - MSB is sign bit; Reverse and ADD '1' to it.
- Convert Decimal to 2's Complement
  - If Positive – Convert from decimal like before.
  - If Negative – Convert to binary --> Flip bits --> ADD '1'
- Convert 2's Complement to Decimal
  - If MSB is '0' - Convert from decimal as before.
  - If MSB is '1' - Convert to binary --> Flip all the bits --> ADD '1'.
- Hexadecimal (HEX) Base 16 – '0' to '9' & 'A' to 'F'
- In HEXADECIMAL - (-42) --> Convert 42 to binary --> Flip BITS --> ADD '1'

Dec	Hex	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Assembly Language Basics

- Machine Code --> Assembly Language for Higher Abstraction. Assembler – 1-to-1 assembly codes and machine dependent.
- Compiler – Generates many machine/assembly instructions.
- Operands – Registers, Memory Locations, Immediate & Implied.
- [label:] mnemonic [operands][;comment]
- Directives – Tell assembler something to do like set the Size of the stack, Define memory & Define constants.
  - Ex - %define SIZE 10
- Data Directives - 'db' : Define byte, 'dw' : Define word, & 'dd' : Define double word.
- Identifiers – like $, %, ^, &
- Listing – Gives information such as offset information and Binary code for commands. Useful for finding errors.
- Generating LISTING.
  - NASM –F ELF –L FRED.TXT 2.1.ASM
- 'xxd finename' - SHOWS exe file
- 'objdump –disassembler-options=intel a.out"
  - Shows dissassemble of the program.
  - Coverts program to assembly.



Logical Operators & Memory Hierarchy

- XOR - '1' if both bits are different.
- X86 Processor Memory Hierarchy
  - Data Bus – Shared communication medium
  - Regusters – Small & Fast
  - Clock – Cycle is smallest amount of time to execute a single instruction.
  - Branch Prediction
  - ALU& FPU
- Instruction Execution Cycle
  - FETCH : Load From – Cache, Main Memory RAM, Hard Disk & Internet.
  - DECODE :
  - EXECUTE :
  - STORE : Cache, RAM, HD & Internet
- Programs – Usually stored on DISK initially
  - Stored on a disk : OS searches the path --> If program exists --> Load into RAM --> Create process with PID --> Execute Process --> OS handles Resources line interrupts, Task switching and so on.
- TASK Switch – OS switches rapidly b/w processes like processes may be waiting and does prioritize other processes in those intervals.
- Save Context – Load new context and commence running.

INTEL is CISC – Less power per instruction.
ARM is RISC

Segments and Functions

- Segments – Different segments have different meanings.
  - DATA : Used for string & global variables. Initialized Read/Write.
    - VarD db "Hello World", 0 – assembler will take care of it.
  - BSS : Uninitialized variables. Read/Writes.
  - TEXT : CODE. Read/Execute.
- Executing Functions - Use call mnemonic to call a function. Operand is the function name.
- Example – Define a string, Put that address in eax & Call the print_string function.
  - Null terminated string and that string pointed to by eax is printed.
- Some of the functions:
  - Read_int : Read integer from the user and result is stored in EAX.
  - Print_int : Print the integer in EAX.
  - Print_nl : Print a newline.
  - Read_char : Read a single char from the user store in eax (or al)

```
%include "asm_io.inc"
segment .data
prompt db "Please Enter a number:", 0
result db "", 0
segment .bss

segment .text
        global  asm_main
asm_main:
        enter   0,0               ; setup routine
        pusha
        ;***************CODE STARTS HERE***************************
        mov eax, prompt
        call print_string
        call read_int
        mov ebx, eax; number 1 in ebx

        mov eax, prompt
        call print_string
        call read_int
        mov ecx, eax; number 2 in ecx

        mov eax,

        add eax, ebx
        call print_int
        call print_nl
        ;***************CODE ENDS HERE*****************************
        popa
        mov     eax, 0            ; return back to C
        leave
        ret
~
~
~
```

Arithmetic Manipulation

- Changing Sizes – Make something smaller
  - Move to smaller register
  - Example – 1) $ mov ax, 1234h, 2) $mov cl, ah
- Increase the Size? Take care of sign bit
  - Example - -7 =0xF9 --> mov bl, -7 --> movsx – sign extend or movzx – zero extend
- Unsigned Multiplication
  - 'mul' for unsigned numbers.
  - 8-bit --> AX = AL * src
  - 16-BIT --> DX:AX = AX*src
- Signed Multiplication
  - Operands can be register, immediate & memory.
  - Imul eax, ebx;
- Division div & idiv
  - Quotient – How many times.
  - Remainder – What is left.
  - 8-bit register --> AX/source; Quotient in AL & Rem in AH.
  - 16-bit register --> DX:AX/source; Quotient in AX & REM in DX.

Movzx

```
mov bl, -7 ; F9
movzx eax, bl; fill 0's 0 = 0000
movzx ax, bl same as above
mov bl, 7; 07
movzx eax, bl; fill 0's
```

Movsx

```
mov bl, -7 ; F9
movsx eax, bl; fill F's as top bit is 1
F = 1111
movsx ax, bl same as above
mov bl, 7; 07
movsx eax, bl; fill 0's as top bit is 0
```

```
mov eax, 0Ah
mov ebx, 2
mov ecx, 3
mov edx, 0FEEDBEEFh
mul bl
dump_regs 1

mov eax, 0Ah
mov ebx, 2
mov ecx, 3
mov edx, 0FEEDBEEFh
mul cx
dump_regs 2

mov eax, 0Ah
mov ebx, 2
mov ecx, 3
mov edx, 0FEEDBEEFh
mul ebx
dump_regs 3
```

```
millermj@unk.edu@unix ~/p/3.1> ./3.1
Register Dump # 1
EAX = 00000014 EBX = 00000002 ECX = 00000003 EDX = FEEDBEEF
ESI = F76E3000 EDI = F76E3000 EBP = FFB40FE8 ESP = FFB40FC8
EIP = 080484C8 FLAGS = 0286       SF       PF
Register Dump # 2
EAX = 0000001E EBX = 00000002 ECX = 00000003 EDX = FEED0000
ESI = F76E3000 EDI = F76E3000 EBP = FFB40FE8 ESP = FFB40FC8
EIP = 080484E6 FLAGS = 0286       SF       PF
Register Dump # 3
EAX = 00000014 EBX = 00000002 ECX = 00000003 EDX = 00000000
ESI = F76E3000 EDI = F76E3000 EBP = FFB40FE8 ESP = FFB40FC8
EIP = 08048503 FLAGS = 0286       SF       PF
```

Control Structures & Looping

- Compare --> "cmp reg1, reg2"
  - Does subtraction Reg1 – Reg2
  - SET flags based on that : Carry, Overflow & Zero
    - If reg1 = reg2 --> Zero flag is set; If reg1 > reg2 --> ZF = 0 & cf = 0; iF REG1 < REG2 --> ZF = 0 & CF = 1. Carry flag as borrow occurred.
  - One operand must be register and other can be memory or immediate
- Branching --> Unconditional Jump & Conditional Jump – Only jump when the flag is SET.
- Looping -->
  - 1) loop label – jump to label if != 0
  - 2) loope or loopz – jumps if ECX != 0 and ZF = 1
  - 3) loopne or loonz – jumps if ECX != 0 & ZF = 0

Example Loop

```
sum = 0;
for( i=10; i >0; i-- )
  sum += i;
```

could be translated into assembly as:

```
      mov    eax, 0          ; eax is sum
      mov    ecx, 10         ; ecx is i
loop_start:
      add    eax, ecx
      loop   loop_start
```

JZ	branches only if ZF is set
JNZ	branches only if ZF is unset
JO	branches only if OF is set
JNO	branches only if OF is unset
JS	branches only if SF is set
JNS	branches only if SF is unset
JC	branches only if CF is set
JNC	branches only if CF is unset
JP	branches only if PF is set
JNP	branches only if PF is unset

ble 2.3: Simple Conditional Branches

If, Looping, Shifts & Rotations

- Loop construct
  - Decrements ECX
  - Checks for zero, if not zero, jump to top
- SHIFT
  - Logical Shift : Move bits left/right and fill with zeros.
  - Discarded bits will SET the carry flag.
  - "shl" - logical left, "shr" - logical right/
  - "AX" & "CF"
- Arithemetic Shift & Double Precision
  - Signed Shift
  - Same as logical shift except the last gets into carry flag.
    - Sal --> Shift Arithemetic Left
    - Sar --> Shift Arithemetic Right
  - Double Precision Shifts
    - Shrd dest, src, cnt
    - Shld dest, src, cnt
    - Modifies flags : CF PF SF ZF (OF, AF undefined)

Example

- Loop to sum numbers read from the user
  - Print prompts for inputs
- Stop if they enter -1

Example

- Prompt user for 2 numbers
  - Amount to shift
  - Number to shift
- Print the result of an arithmetic shift

Example if template

```
 cmp reg1, reg2
       jz labela
       ;code if reg1 != reg2
       jmp endLabel
labela:
       ;code if reg1 == reg2
endlabel:
```

Double Precision Arithmetic Shift

- **SHLD/SHRD** - Double Precision Shift (386+)
  - Usage:
    - `SHLD dest, src, count`
    - `SHRD dest, src, count`
  - Modifies flags: CF PF SF ZF (OF,AF undefined)
- SHLD shifts dest to the left count times and the bit positions opened are filled with the most significant bits of src. SHRD shifts dest to the right count times and the bit positions opened are filled with the least significant bits of the second operand. Only the 5 lower bits of count are used.[2]

Module 1 Review

- Mnemonic – Instruction to be executed.
- Operand – A parameter to the instruction.
- Names of 32-bit registers ? Check – EAX TO EDX,….
- Loop mnemonic register – ECX
- Shift to the right
- Shift to the left
- Comparison and Looping.
- Write code to loop ebx times and calculate the power of 2^ebx and store in EAX

Module 2 – Indirect Addressing Stack, Arrays and Strings

Indirect Addressing

- Point to memory (RAM)
  - Hello db "This is a string",0
  - Mov eax, hello; this is a pointer
  - Call print_string
- Variables – Defined in the .bss or .data
  - Data section includes initialised variables
  - BSS section includes non-initialised variables with just space allocated.
- In the example besides, first call print_int prints only the address and the second call print_int

Prints the value the address pointing to.

Variable example

- Load and save data to and from a memory location

```
.data
fred dd 17 ; define dword
. . .
mov eax, [fred]
inc eax
mov [fred], eax
```

Addresses

```
segment .data
var1 dd 0
. . .
mov eax, var1
call print_int
call print_nl
mov eax, [var1]
call print_int
call print_nl
```

String example

```
segment .data
string db "Car Rocket Horse",0
mov eax, string
call print_string
call print_nl
mov eax, string
add eax, 4
call print_string
call print_nl
```

```
mov eax, string
add eax, 11
call print_string
call print_nl
```

```
millermj@unk.edu@unix ~/p/6.1>
Car Rocket Horse
Rocket Horse
Horse
millermj@unk.edu@unix ~/p/6.1>
```

Notation

- Brackets mean value at the location pointed to by a variable

```
.data
var db “Hello Class!”,0
. . .
mov eax, var ; move the address to eax
mov al, [var] ; move the character ‘H’ to al
mov ax, [var]; move ‘eH’ to ax “Little Endian”
mov eax, [var]; mov ‘lleH’ to ax
```

STACK

- What is the stack?
  - Stack is a region of RAM pointed to by register ESP.
  - Push and Pop things onto and off of the stack.
  - Last In First Out Operation
  - SS register points to stack memory
- Used for?
  - Stores local variables
  - Stores parameter
  - Allows for function context to be saved
  - Temporary Space
- Usage of POP and PUSH
- Operations -
  - Call function_Name - Push the address of the next instruction onto the stack -->
  - Move EIP to the new functions location address.
  - RET – Return from a function --> Pop the return address off the stack
  - --> Move EIP to that address
- "AND esp, 0FFFFFFF0h" - Make sure that esp is on an even byte boundary. - SECURITY

```
push eax
push ebx
pop ebx
add esp, 20h
sub esp, 20h
```

- push X
  - Subtracts 4 from esp
    - Copies that data to the location pointed at by esp
  - X is a register or Immediate value
- pop X
  - Adds 4 to esp
    - Stores result in the register X given in pop instruction

- pushad
  - Push EAX, ECX, EDX, EBX, original ESP, EBP, ESI,and EDI
- popad
  - Pop the registers in the reverse order, original ESP is restored.

Operations

- sub esp, X
  - Subtract X bytes from esp
    - used when entering a function
  - Space is used for local variables
- add esp, X
  - Add X bytes to esp
    - used when leaving a function
- and esp, 0FFFFFFF0h
  - Make sure that esp is on an even byte boundary

STACK USAGE

Usage

- If you push something onto the stack
  - Then you must either
    - Pop it back off
    - Or add 4 to esp for every variable pushed on
  - Then you must either
    - Pop it back off
    - Or add 4 to esp for every variable pushed on

```
push eax
push ebx
pop  ebx
pop  eax

push eax
push eax
add  esp, 8
```

Saving Data

- Enter
  - Push data in
- Exit
  - Pop data in reverse order

```
;Enter
push eax
push ecx
push edx
push esi
```

```
;Exit
pop edi
pop edx
pop ecx
pop eax
```

printf

- printf
  - push args on in reverse order
  - string format will be last

Quiz

- When exiting a function what operation do we preform on esp
  - add esp, X; where X is how much we allocated when entering the function

2.6/2.7 Functions

- Creating a FUNCTION,
  - LABEL is function name & RET at the end.
  - CALL instruction calls a function – 1) PUSHES return address of the next instruction onto the stack, 2) JUMPS to the address of the function – EIP sets to function address.
  - RET instruction will return from a function – 1) POPS off the address on the top of the stack, 2) SETS EIP to that address.
- DEBUGGING using GDB,
  - wget -q -O- https://github.com/hugsy/gef/raw/master/scripts/gef.sh | sh
- Function Prologue – Setup entering a function and SAVE the ebp register
  - EBP – Extended Base Pointer.
  - Gives you a constant point of reference.

Code

```
push ebp ; save the old version of ebp onto the stack
mov ebp, esp ;set ebp to esp
```

Simple Example

```
        call clearEAX
        dump_regs 0
        ret; return from the main function

clearEAX:
        mov eax, 0; set eax to 0
  ret
```

Stack Diagram

```
push 0beefbeefh
push 0cafecafeh
call function
add esp, 8
function:
        push ebp
        mov ebp, esp
        mov eax, 0 ←
```

```
Top of Stack → old ebp          ← ebp
               ret address
               0xcafecafe
               0xbeefbeef
```

2.7 Function Prolog

- A function prolog is the initial set of instructions executed when a function is called.
- It sets up the function's stack frame and prepares the environment for the function to execute.
  - Save the return address – The address to return after function completes is saved, usually on the stack.
  - Save the caller's frame pointer - The current frame pointer (base pointer) is saved to keep track of the caller's stack frame.
  - Set up the new frame pointer – The frame pointer is updated to point to the current stack frame.
  - Allocate space for local variables – Space is reserved on the stack for the function's local variables.
- Function Prologue – It’s a process sets by the compiler when the callers calls the function.
  - Save the EBP register first.

```
push ebp ; save the old version of ebp onto the stack
mov ebp, esp ; set ebp to esp
```

- Example :

```
push 0beefbeefh
push 0cafecafeh
call function
add esp, 8
function:
        push ebp
        mov ebp, esp
        mov eax, 0 ←
```

```
Top of Stack → old ebp        ← ebp
               ret address
               0xcafecafe
               0xbeefbeef
```

2.7 Function Prolog Example

```
asm_main:
        enter   0,0
        pusha
                push 0beefbeefh
                push 0cafecafeh
                call function
                add esp, 8
        popa
        mov     eax, 0
        leave
        ret
```

```
function:
        push ebp
        mov ebp, esp
        mov eax, 0


        mov esp, ebp
        pop ebp
        ret
```

2.8 Function Epilog

- Restore the stack,

```
mov esp, ebp ; restore esp
pop ebp ; restore old version of ebp
ret; return from function
```

Stack Diagram

```
push 0beefbeefh
push 0cafecafeh
call function
add esp, 8
function:
        push ebp
        mov ebp, esp
        mov eax, 0
        mov esp, ebp ←
        pop ebp
        ret
```

```
Top of Stack → old ebp        ← ebp
               ret address
               0xcafecafe
               0xbeefbeef
```

2.8 Example

```
asm_main:
        enter   0,0
        pusha
                push 0beefbeefh
                push 0cafecafeh
                call function
                add esp, 8
        popa
        mov     eax, 0
        leave
        ret
```

```
function:
        push ebp
        mov ebp, esp
        mov eax, 0


        mov esp, ebp
        pop ebp
        ret
```

2.1 Indirect Addressing

- Indirect Addressing : Point to memory (RAM).
- Allocate mem in RAM : "hello db "This is a string",0
- Moving that RAM value into EAX - "mov eax, hello" this is a pointer
- Creating variables : Defined in the .bss or .data
- Example :
  - Data Section – We can give values.
    - .data
    - "Hello db 0" - 'hello' is variable, 'db' denotes double bytes & '0' indicates the initial value.
  - BSS Section - don’t assign value just allocate space.
    - .bss
    - "Resb world"

Addresses

```
segment .data
var1 dd 0
. . .
mov eax, var1
call print_int
call print_nl
mov eax, [var1]
call print_int
call print_nl
```

```
milhous@unix:~/projects/6.1$ ./6.1
134520872
0
milhous@unix:~/projects/6.1$
```

Thank You