Assembly

Registers

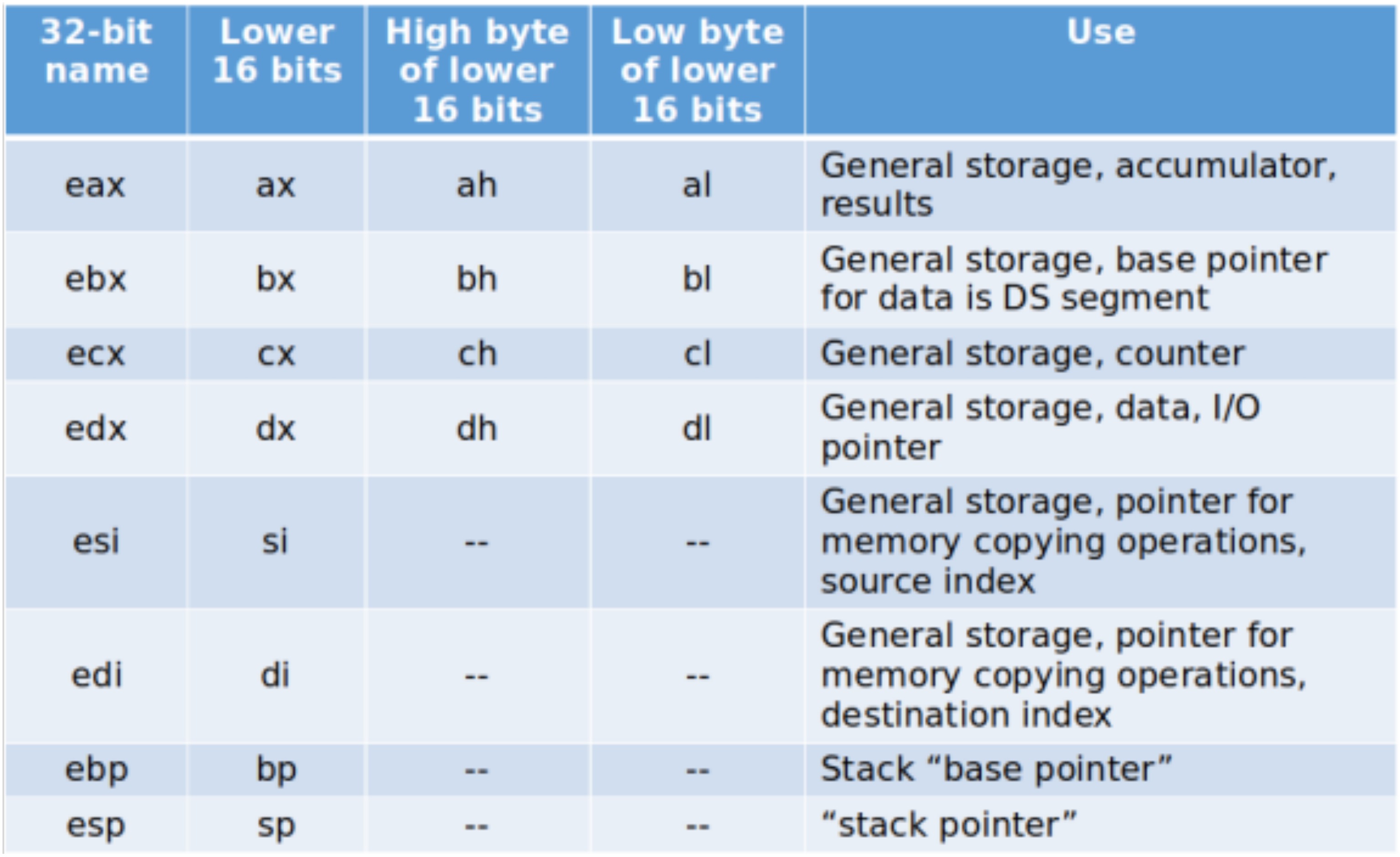
32-bit

Registers for 32-bit:

- There are 10 32-bit registers which can be grouped into 3 categories:
  - General registers
  - Control registers
  - Segment registers

- General registers are further divided into the following groups:
  - Data registers
  - Pointer registers
  - Index registers

General Purpose Registers



32-bit name	Lower 16 bits	High byte of lower 16 bits	Low byte of lower 16 bits	Use
eax	ax	ah	al	General storage, accumulator, results
ebx	bx	bh	bl	General storage, base pointer for data is DS segment
ecx	cx	ch	cl	General storage, counter
edx	dx	dh	dl	General storage, data, I/O pointer
esi	si	--	--	General storage, pointer for memory copying operations, source index
edi	di	--	--	General storage, pointer for memory copying operations, destination index
ebp	bp	--	--	Stack “base pointer”
esp	sp	--	--	“stack pointer”

- DS = Data Segment

Data Registers

Data Registers:

• Four 32-bit data registers are used for arithmetic, logical, & other operations. These can be used in 3 ways:

32-bit	**Lower 16-bit of 32-bit**	**<b><b>Lower & Higher 8-bit of lower 16-bit**
EAX	AX	AH, AL
EBX	BX	BH, BL
ECX	CX	CH, CL
EDX	DX	DH,DL

• **AX** - Primary accumulator, it's used in I/O & most arithmetic instructions. **EX**: In multiplication operation, 1 operand is stored in EAX, AX, or AL register depending on the size of the operand.
• **BX** - The base register which could be used in indexed addressing.
• **CX** - The count register. Both ECX & CX store the loop count in iterative operations (iterating through list, etc).
• **DX** - The data register. also used in I/O operations along with AX for multiply & divide operations involving large values.

Pointer Registers

Pointer Registers:

32-bit registers	**16-bit portion of 32-bit**
EIP	IP
ESP	SP
EBP	BP

• There are 3 categories of pointer registers:
- **Instruction Pointer (IP)** - The 16-bit IP register stores the offset address of the next instruction to be executed, when associated with the CS register (as CS:IP) it can give the complete address of the current instruction in the code segment.
- **Stack Pointer (SP)** - The 16-bit SP register provides the offset value within the program stack, when associated with the SS register (as SS:SP), it refers to the current position of data or address within the program stack.
- **Base Pointer (BP)** - The 16-bit BP register mainly helps in referencing the parameter vars passed to a subroutine. The address in SS register is combined w/offset in BP to get the location of the parameter. BP can also be combined w/DI & SI as base register for special addressing.

Index Registers

Index Registers:

- 32-bit index registers along w/their 16-bit rightmost portions (SI & DI) are used for indexed addressing
    - Occassionally used in addition & subtraction too!

32-bit	**16-bit rightmost of 32-bit**
ESI	SI
EDI	DI

- There are 2 sets of index pointers:
    - **Source Index (SI)** - Used as source index for string operations.
    - **Destination Index (DI)** - Used as destination index for string oeprations

Special Purpose Registers

IA-32 Special Registers:

EIP (Extending Instruction Pointer):
- Program counter
- 32-bit address for the next instruction
- As name suggests, it points to another instruction
- Pointer is altered by special instructions, not directly by coder
    - ▪ Special instructions include:
        - → JMP
        - → Jcc (Jump on condition code cc)
        - → CALL
        - → RET
        - → IRET
            - ⇒ These use 1 or more of the status flags as condition codes, the status flags are defined next.
- Control flow attacks focus on controlling this register.

Status Control Register (EFLAGS):
- Also known as status flags
- Processor status
- Basis for conditional control flow
    - ▪ CF (Carry Flag)
        - → Used w/add with carry (ADC) & subtract with borrow (SBB) instructions in multi-precision arithmetic to cause a carry or borrow from one problem to the next. This flag can indicate an overflow condition for an unsigned-integer too.
    - ▪ PF (Parity Flag)
        - → Set if the least-significant byte of the result contains an even number of 1 bits length
    - ▪ ZF (Zero Flag)
        - → Set if the result is zero
    - ▪ SF (Sign Flag)
        - → Set equal to the most-significant bit of the result which is the sign bit of a signed integer (0 for positive value, 1 for negative value).
    - ▪ OF (Overflow Flag)
        - → Indicates overflow condition for signed-integer arithmetic. Set if the int result is too large (positive #) or too small (negative #) to fit in the destination operand.
- Status flags allow a single arithmetic operation to produce results for 3 different data types:
    - ▪ unsigned ints
    - ▪ signed ints
    - ▪ Binary coded decimal ints (BCD)
- If result is treated as an unsigned int:
    - ▪ CF flag indicates an out-of-range condition (carry or borrow)
- If result is treated as a signed int:
    - ▪ OF flag indicates a carry or borrow
- The SF flag indicates the sign of a signed int
- The ZF flag indicates either a signed or an unsigned integer zero.

Control Registers

Control Registers:

• The combination of **32-bit instruction pointer register** & **32-bit flags register** is called the control register.
• Many instructions involve comparisons & mathematical calculations which change the status of the flags.
• There are some other conditional instructions which test the value of these status flags to take the control flow to another location depending on the status.

Common flag bits include:

Flag Bit	**Function**
Overflow flag (OF)	Indicates overflow of a high-order bit (leftmost bit data after a signed arithmetic operation
Direction Flag (DF)	Determines left or right direction for moving or comparing string data. When set to 0: string opera takes left-to-right direction & when set to 1: string operation takes right-to-left direction
Interrupt Flag (IF)	Determines whether the external interrupts like keyboard entry, etc are to be ignored or processe value is 0, it disables external interrupt. If value is enables interrupts
Trap Flag (TF)	Allows setting the operation of processor in single step mode to allow for stepping through executio instruction at a time. Useful for debugging.
Sign Flag (SF)	Shows the sign of the result of an arithmetic operation, a positive result clears the value of SF and a negative result sets it to 1. The sign is indic by the high-order of leftmost bit.
Zero Flag (ZF)	Indicates result of an arithmetic operation or comparison operation. A nonzero result clears the to 0 and a zero result sets it to 1.
Auxiliary Carry Flag (AF)	Contains the carry from bit 3 to bit 4 following an arithmetic operation; used for specialized arithme The AF is set when 1-byte arithmetic operation ca a carry from bit 3 into bit 4.
Parity Flag (PF)	Indicates total number of 1-bits in the result obtai from an arithmetic operation. An even number of bits clears the parity flag to 0 & an odd number o bits sets the flag to 1.
Carry Flag (CF)	Contains the carry of 0 or 1 from a high-order bit (leftmost) after an arithmetic operation. It also sto the contents of last bit of a shift or rotate operatio

Segment Registers

Segment Registers:

- Segments are specific areas defined in a program for containing data, code, & stack.
- There are 3 main segments:
    - **Code Segment** - Contains all the instructions to be executed. A 16-bit Code Segment register (CS register) stores the starting address of the code segment.
    - **Data Segment** - Contains data, constants, and work areas. A 16-bit Data Segment register (DS Register) stores the starting address of the data segment.
    - **Stack Segment** - Contains data & return addresses of procedures or subroutines. It's implemented as a ‘stack’ data structure. The Stack Segment register (SS Register) stores the starting address of the stack.
- All memory locations within a segment are related to the starting address of the segment which begins in an address evenly divisible by 16.
- To get the exact location of data or instruction within a segment, an offset value (aka displacement value) is required.
    - To reference any memory location in a segment, the processor combines the segment address in the segment register (SS, stores all starting addresses of a segment) w/the offset value of the location.

- Excluding the DS, CS, & SS registers there are other extra segment registers which provide additional segments for storing data & include:
    - ES (Extra segment)
    - FS
    - GS

Segmented Memory Model

Segmented Memory Model:

- Usually not used
- Contains special registers:
  - CS - Code segment
  - DS - Data segments
  - SS - Stack segment
  - ES, FS, GS - Extra data segments
  - in 64-bit mode: CS, DS, SS, & ES are forced to 0
- Segment registers contain 16 bits
  - These are pointers into a segment table which give the start address of a segment while all other addresses are treated as offsets.
  - `mov ss:[edx], es` - moves values stored as SS address + the edx (offset?)

Other Registers

Other Registers:

- Floating point (FPU) registers
  - ST0 to ST7, status word, control word, tag word, data operand pointer, & instruction pointer
- MMX (for vector operations)
  - MM0 to MM7 (64-bit registers)
  - XMM0 to XMM7 (128-bit registers)
- Control registers
  - CR0, CR1 (reserved), CR2, CR3
- Debug registers
  - DR0, DR1, DR2, DR3, DR6, DR7
- System Table Pointer registers
  - GDTR, LDTR, IDTR, task register

• **NOTE:** Some instructions use quadword operands contained in a pair of 32-bit registers. Register pairs have a colon separating them, **EX**: In the register pair **EDX:EAX**, EDX contains the *high* order bits while EAX contains the *low* order bits of a quadword operand.

Sys Calls:

- The following steps are required for using Linux sys calls in assembly programs:
  - ☐ Put the sys call number in the EAX register.
  - ☐ Store the arguments to the sys call in the registers EBX, ECX, (see below for full list)
  - ☐ Call the relevant interrupt (80h).
  - ☐ The result is usually returned in the EAX register
- There are 6 registers which stores the arguments of the sys call used:
  1- EBX
  2- ECX
  3- EDX
  4- ESI
  5- EDI
- The following shows the use of the sys call sys_exit:

```
mov     eax, 1      ; system call number (sys_exit)
int     0x80        ; call kernal
```

- All sys calls & their corresponding numbers are stored in /usr/include/asm/unistd.h, here is a small list of sys calls:

EAX	**Name**	**EBX**	**ECX**
1	sys_exit	int	-
2	sys_fork	struct pt_regs	-
3	sys_read	unsigned int	char *
4	sys_write	unsigned int	const char *
5	sys_open	const char *	int
6	sys_close	unsigned int	-

Data Types

Data Types:

• Byte (8 bits)
• Word (16 bits)
• Double Word (32 bits)
• Quadword (64 bits)
• Double quadword (128 bits)

• To move a quadword from one xmm register to another:
  ▪ `movq xmm1, xmm2`
• Intel doesn't normally differentiate between signed & unsigned integers, except in comparison operations.

Common Instructions

Move

Mov:

• mov (Opcodes: 88, 89, 8A, 8B, 8C, 8E)
• copies data referred to by its second operand (ex: register contents, memory contents, or a constant value) into the location referred to by its first operand (ex: register or memory).

- **EX**: `mov eax, ebx` ← Copies the value in ebx over to eax. This is what it means by data referred to by its second operand, simply the source is the second argument, or operand while the destination is the first argument, or operand.
- **EX**: `mov byte ptr [var], 5` ← stores the value 5 into the byte at location [var].

Syntax:

- mov <reg>, <reg>
- mov <reg>, <mem>
- mov <mem>, <reg>
- mov <reg>, <const>
- mov <mem>, <const>

Push

Push:

- push - Push stack (Opcodes: FF, 89, 8A, 8B, 8C, 8E)
- push places its operand onto the top of the hardware supported stack in memory.
    - It first decrements ESP by 4, then places its operand into the contents of the 32-bit location at address [ESP]. ESP (the stack pointer) is decremented by push since the x86 stack grows down, the stack grows from high addresses to lower addresses.

Syntax:

- push <reg32>
- push <mem>
- push <con32>

Examples:

- `push eax` ← push eax on the stack
- `push [var]` ← push the 4 bytes at address var onto the stack

Pop

Pop:
• pop - Pop stack
• Removes the 4-byte data element from the top of the hardware-supported stack into the specified argument, or operand (**EX**: register or memory location).
    ▪ First, pop moves the 4 bytes located at memory location [SP] into the specified register or memory location, & then increments SP by 4.

Syntax:
• pop <reg32>
• pop <mem>

Examples:
• `pop edi` ← pop the top element of the stack into EDI
• `pop [ebx]` ← pop the top element of the stack into the memory at the 4 bytes starting at location EBX.

LEA

LEA (Load Effective Address):
• places address specified in second argument, or operand into the register specified by its first argument, or operand
    ▪ Only the effective address is computed & placed into the register, the actual contents of the memory location aren't loaded.
• Useful for obtaining a pointer into a memory region.

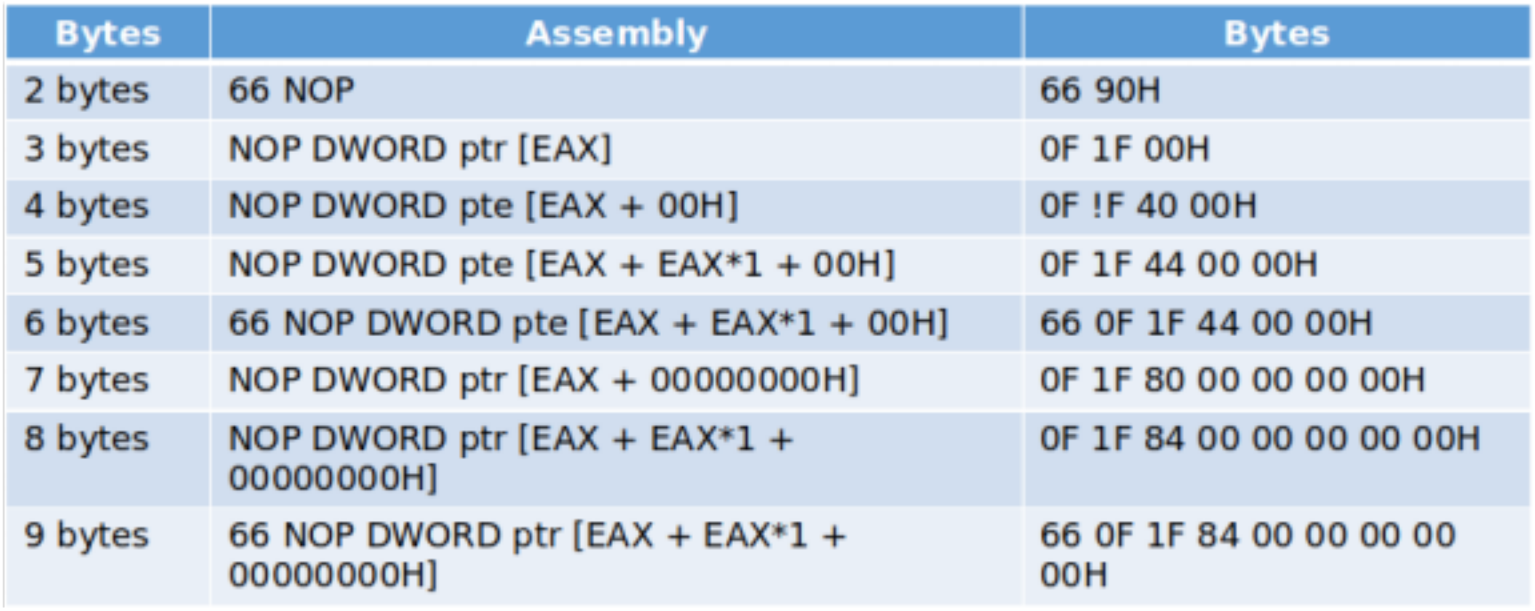
Syntax:
• lea <reg32>, <mem>

Examples:
• `lea edi, [ebx+4*esi]` ← the quantity EBX+4*ESI is placed in EDI
• `lea eax, [var]` ← the value in var is placed in EAX
• `lea eax, [val]` ← the value val is placed in EAX

NOP

NOP:

- nop - no operation (opcode 0x90)
- Does nothing, but a handy placeholder & useful for shellcoding.
- Can exist for alignment of functions



Bytes	Assembly	Bytes
2 bytes	66 NOP	66 90H
3 bytes	NOP DWORD ptr [EAX]	0F 1F 00H
4 bytes	NOP DWORD pte [EAX + 00H]	0F !F 40 00H
5 bytes	NOP DWORD pte [EAX + EAX*1 + 00H]	0F 1F 44 00 00H
6 bytes	66 NOP DWORD pte [EAX + EAX*1 + 00H]	66 0F 1F 44 00 00H
7 bytes	NOP DWORD ptr [EAX + 00000000H]	0F 1F 80 00 00 00 00H
8 bytes	NOP DWORD ptr [EAX + EAX*1 + 00000000H]	0F 1F 84 00 00 00 00 00H
9 bytes	66 NOP DWORD ptr [EAX + EAX*1 + 00000000H]	66 0F 1F 84 00 00 00 00 00H

Arithmetic & Logic Instructions

Addition

Addition:
- add - Integer addition
- Adds together its 2 operands & stores the result in its 1st argument, or operand.
- Both operands may be registers, at most 1 operand may be a memory location.

Syntax:
- add <reg>, <reg>
- add <reg>, <mem>
- add <mem>, <reg>
- add <reg>, <con>
- add <mem>, <con>

Examples:
- `add eax, 10` ← EAX + 10
- `add DWORD PTR [var], 10` ← add 10 to the s32-bit value stored at memory address var

Subtraction

Subtraction:
• sub - Integer subtraction
• subtracts value of second argument, or operand from the value of its first argument, or operand. Result is stored in first argument, or operand.

Syntax:
• sub <reg>, <reg>
• sub <reg>, <mem>
• sub <mem>, <reg>
• sub <reg>, <con>
• sub <mem>, <con>

Examples:
• `sub al, ah` ← AL - AH, result stored in AL
• `sub eax, 216` ← subtract 216 from value stored in EAX

Multiplication

Multiplication:

- imul - integer multiplication
- **two-operand form** multiples its two arguments, or operands together & stores the result in the first argument, or operand. The result (aka the first argument, or operand) must be a register.
- **three operand form** multiplies its second and third arguments, or operands together & stores the result in its first argument, or operand. Still, the result operand must be a register & the third argument, or operand is restricted to being a constant value.

Syntax:

- imul <reg32>, <reg32>
- imul <reg32>, <mem>
- imul <reg32>, <reg32>, <con>
- imul <reg32>, <mem>, <con>

Examples:

- `imul eax, [var]` ← Multiplies contents of EAX by the 32-bit contents of the memory location var, result stored in EAX.
- `imul esi, edi, 25` ← multiplies 25 by contents of EDI & stores result in ESI

Increment & Decrement

Increment & Decrement:
- inc, dec - Increment, Decrement
- inc increments the contents of its arguments, or operand by one. dec instruction decrements the contents of its argument, or operand by one.

Syntax:
- inc <reg>
- inc <mem>
- dec <reg>
- dec <mem>

Examples:
- `dec eax` ← subtract one from the contents of EAX
- `inc DWORD PTR [var]` ← add one to the 32-bit integer stored at location var

Integer Division

Integer Division:
• idiv - Integer division
• divides the contents of the 64 bit integer EDX:EAX (constructed by viewing EDX as the most significant 4 bytes & EAX as the least significant 4 bytes) by the specified argument, or operand value.
  ▪ The quotient result is stored in EAX & the remainder is placed in EDX

Syntax:
• idiv <reg32>
• idiv <mem>

Examples:
• `idiv ebx` ← divide the contents of EDX:EAX by the contents of EBX. Place the quotient in EAX & the remainder in EDX.
• `idiv DWORD PTR [var]` ← divide the contents of EDX:EAX by the 32-bit value stored at memory location var. Place the quotient in EAX & the remainder in EDX.

Bitwise Logic

Bitwise Logic:
• and, or, xor - Bitwise logical and, or, and exclusive or
• Perform the specified logical operation on their arguments, or operands, placing the result in the first argument, or operand location.

Syntax:
• (<op> = and, or, or xor)
• <op> <reg>, <reg>
• <op> <reg>, <mem>
• <op> <mem>, <reg>
• <op> <reg>, <con>
• <op> <mem>, <con>

Examples:
• `and eax, 0fH` ← Clears all excep the last 4 bits of EAX
• `xor eax, eax` ← set the contents of EAX to zero

Bitwise Logical Not

Bitwise Logical Not:

- not - Bitwise Logical Not
- Logically negates the argument, or operand contents (basically, flips all bit values in the argument, or operand)

Syntax:

- not <reg>
- not <mem>

Example:

- `not BYTE PTR [var]` ← negate all bits in the byte at the memory location var.

Negation

Negation:
• neg - Negate
• performs the two's complement negation of the argument, or operand contents. In other words, turns the argument, or operand contents to negative.

Syntax:
• neg <reg>
• neg <mem>

Example:
• neg eax ← EAX turns into -EAX

Shifting

Shifting:
• shl, shr - Shift Left, Shift Right
• shift the bits in their first argument's, or operand's contents left and right, padding the resulting empty bit positions with zeros. The shifted argument, or operand can be shifted up to 31 places.
• The number of bits to shift is specified by the 2nd argument, or operand, which can be either an 8-bit constant or the register CL. In either case, shifts counts of greater than 31 are performed modulo 32.

Syntax:
• <op> <reg>, <con8>
• <op> <mem>, <con8>
• <op> <reg>, <cl>
• <op> <mem>, <cl>

Examples:
• `shr ebx, cli` ← Divides the value of EBX by *2n* where n is the value in CL. The result is stored in EBX.
• `sh1 eax, 1` ← Multiplies value of EAX by 2 (if the most significant bit is 0)

Control Flow Instructions

Jump

Jump:
• jmp - Jump
• Transfer program's control flow to the instruction at the memory location specified by the argument, or operand.

Syntax:
• jmp <label>

Example:
```
mov esi, [ebp+8]
begin: xor ecx, ecx
       mov eax, [esi]
       jmp begin
```
• `jmp begin` ← jump to the instruction labeled begin which then turns contents of ECX into 0

Conditional Jump

Conditional Jump:
- jcondition - Conditional jump
- Jump to label based on a condition in the control status word (EFLAGS)

Syntax:
- je <label> (jump if equal)
- jne <label> (jump if not equal)
- jz <label> (jump if last result is 0)
- jg <label> (jump if greater than)
- jge <label> (jump if greater than or equal to)
- jl <label> (jump if less than)
- jle <label> (jump if less than or equal to)

Example:
```
cmp eax, ebx
jle done
```
- If the contents of EAX are less than or equal to the contents of EBX, jump to the label done. Otherwise continue to next instruction.

Comparison

Comparison:
• cmp - Compare
• Compare the values of the 2 specific arguments, or operands, setting the condition codes in the machine status word appropriately.

Syntax:
• cmp <reg>, <reg>
• cmp <reg>, <mem>
• cmp <mem>, <reg>
• cmp <reg>, <con>

Example:
```
cmp DWORD PTR [var], 10
jeq loop
```
• If the 4 bytes stored at location var are equal to the 4-byte integer constant 10, jump to the location labeled loop.

String Operations

String Operations:

- ins, outs
  - inputs/outputs a string from/to a port
- movs, movsb, movsw, movsd
  - moves data from one string to another
- lods, lodsb, lodsw, lodsd
  - loads data into a string
- stos, stosb, stows, stosd
  - stores data in a string
- cmps, cmpsb, cmpsw, cmpsd
  - Compare strings in memory
- scas, scab, scasw, scads
  - Compare a string (scan a string)
- repm, repe, repz, repne, repnz
  - Repeat string operation until condition is met.

Subroutine Call & Return

Call & Return:

• call, ret - Subroutine call & return
• These instructions implement a subroutine call & return.
• The call instruction first pushes the current code location onto the hardware supported stack in memory (see push instruction for details), & then performs an unconditional jump to the code location indicated by the label argument, or operand.
    ▪ Unlike the simple jmp instructions, the call instruction saves the location to return to when the subroutine completes.
• The ret instruction implements a subroutine return mechanism. ret first pops a code location off the hardware supported in-memory stack (see the pop instruction for details). It then performs an unconditional jmp to the retrieved code location.

Syntax:

• call <label> ← performs unconditional jmp to the location <label>
• ret ← returns to the original saved location before the jmp.

Interrupts

Interrupts:
- int/rti - generate & return from a software interrupt.
- int generates a software interrupt, transferring control to an interrupt handler
- RTI returns from interrupt

Syntax:
- int <con>
- rti <con>

Example:
- int 3H // debugger breakpoint
- int 80H // linux system call
- rti

The Stack

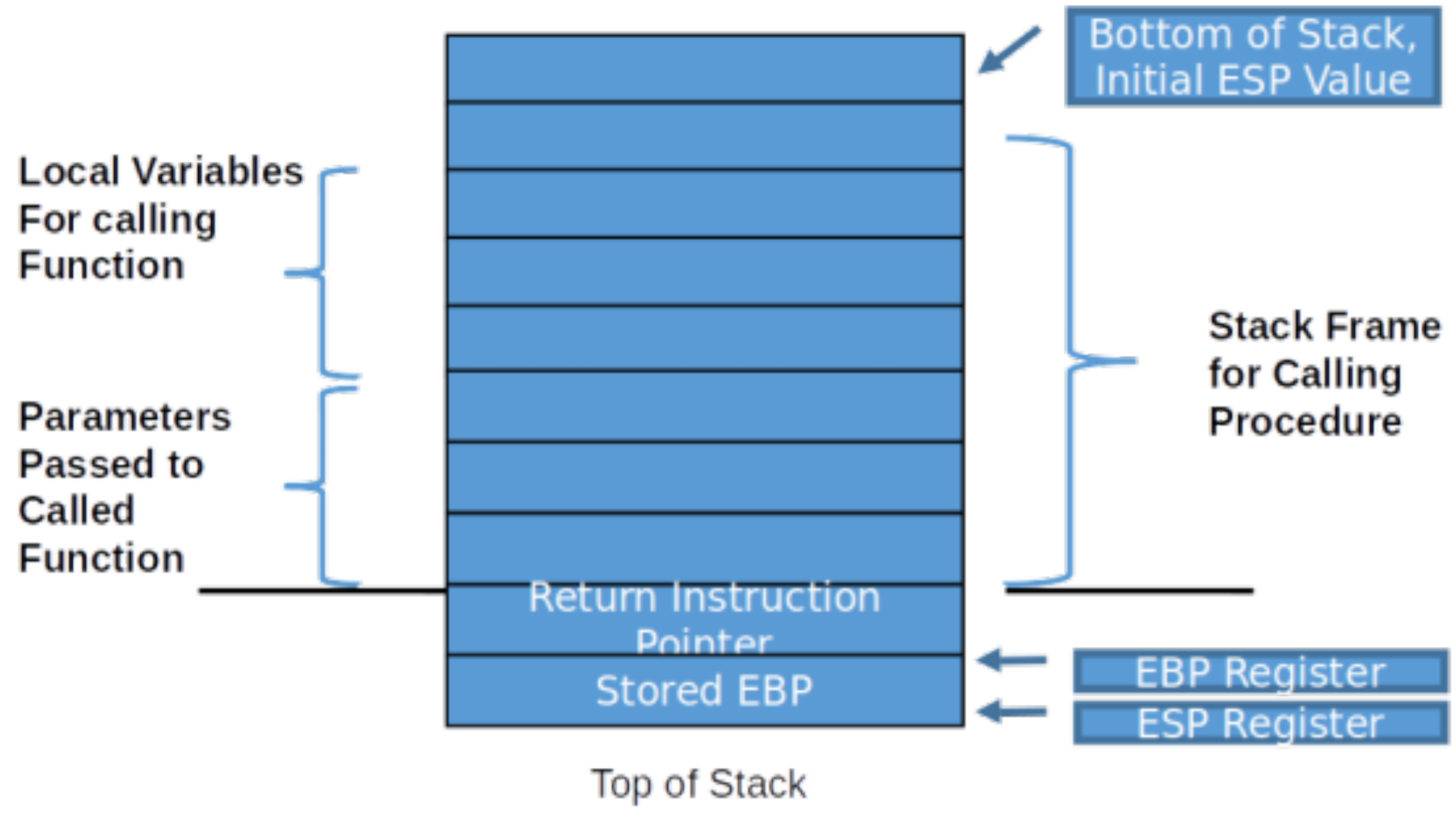
The Stack:

• The Intel Stack is a *LIFO* (last-in-first-out) data structure. The stack grows downward in memory.
• The sp register (stack pointer) indicates the current memory address of the stack. The register is referenced as “SP”, “ESP”, or “RSP” for 16/32/64 bit wide stacks respectively.
    ▪ So for IA-32, we use ESP.
• A PUSH operation decrements ESP & then stores a value at the address referenced by ESP.
    ▪ Intel allows us to reference individual bytes, so a PUSH actually decrements ESP by 4. (4 bytes = 32 bits).

• A POP operation retrieves the value at the address referenced by ESP & then increments ESP & then increments ESP.

• The Stack Segment Register (SS) can be used to specify the base location of the stack in memory.

Diagram



Bottom of Stack, Initial ESP Value
Local Variables For calling Function
Parameters Passed to Called Function
Stack Frame for Calling Procedure
Return Instruction Pointer
Stored EBP
EBP Register
ESP Register
Top of Stack

Stack Instructions

Stack Instructions:

- PUSH
- POP
- PUSHA - push ALL GP registers onto stack
    - EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI ← In this order
- POPA - popa all GP registers from stack
- PUSHF - push EFLAGS
- POPF - pop EFLAGS
- ENTER - enter stack frame
    - push ebp; mov ebp, esp
- LEAVE - exit stack frame
    - mov esp, ebp; pop ebp

Function Calling Conventions (cdel entry)

Function Calling Conventions (cdel entry):

- Cdecl - C-standard
  - Standard entry sequence:
    - → save old base pointer
    - → set new base pointer
    - → allocate space for local vars

```
push ebp
mov ebp, esp
sub esp, x      ; not always present
--OR--
enter
sub esp, x      ; not always present
```

Function Calling Conventions (cdel exit):

- Cdecl - C-standard
  - Standard exit sequence:
    - → reload old stack pointer
    - → reload old base pointer

```
mov esp, ebp
pop ebp
ret
--OR--
leave
ret
```

Calling Conventions

Calling Conventions:

- Cdecl
    - used by GCC, GNU, & standard “c” libs
- Stdcall
    - used by Wing32 API
    - “Called PASCAL” convention
- Fastcall
    - Many implementions although it isn't a standard

Parameter Passing Conventions

Parameter Passing Conventions:

- cdecl:
    - Parameters pushed : last one first
    - EAX, ECX, EDX not preserved
    - return values returned in EAX - floating point in ST0
    - Caller cleans up stack
- stdcall:
    - same as cdecl except callee cleans-up
    - So callee pops values from stack
    - *Pascal convention pushed parameters first one is first*
- fastcall:
    - Parameters usually passed in registers:
        - → **EX**: First in ECX, second in EDX, then rest on stack..

Parameter & Local Variable Access

Parameter & Local Variable Access:

- Parameters usually accessed:
    - With a positive offset from ebp
    - [ebp + 8]
- Local vars usually accepted:
    - with a negative offset from ebp
        - → [ebp - 8]
- Recall EBP points to locations on stack where old EBP is stored:
    - [EBP] = old EBP
    - [EBP + 0x04] = return address
    - [EBP + 0x08] = parameter 1
    - [EBP + 0x0c] = parameter 2
- Sometimes parameters are copied to local vars or registers.
- Sometimes there's no local storage for index vars, just registers.

Addressing Modes:

- Most assembly instructions requires operands
  - ▪ an operand address provides location where data to be processed is stored.
- If 2 operands are required:
  - ▪ 1st = destination which contains data in a register or memory location.
  - ▪ 2nd = source
    → The source contains either data to be delivered (**immediate addressing**) or the address (in register or memory) of the data.
    → Generally source data remains unaltered after the operation.

- The 3 basic modes of addressing are:
  1- **Register Addressing**
  2- **Immediate Addressing**
  3- **Memory Addressing**

Register Addressing

Register Addressing:

• In this addressing mode, a register has the operand or argument
• Depending on the instruction, the register may be the 1st operand (argument) or the 2nd operand (argument):

-
  ```
  MOV DX, TAX_RATE ; Register in 1st operand

  MOV COUNT, CX ; Register in 2nd operand

  MOV EAX, EBX ; Both operands are in registers
  ```

• Since processing data between register sdoesn't involve memory, this mode provides the fastest processing of data.

Immediate Addressing

Immediate Addressing:

- Here, an immediate operand has a constant value or an expression
  - 1st operand may be a register or memory location
  - 2nd operand is an immediate constant
  - The 1st operand defines the length of the data

```
BYTE_VALUE DB 150 ; A byte value is defined

WORD_VALUE DW 300 ; A word value is defined

ADD BYTE_VALUE, 65 ; An immediate operand 65 is added

MOV AX, 45H ; Immediate constant 45H is transferred to AX
```

Direct Memory Addressing

Direct Memory Addressing:

• This method of addressing is slower when processing data
• When operands are specified in memory addressing mode, direct access to main memory (usually to data segment) is required.

  ▪ To locate the exact location of data in memory, you need:
    1. **Segment start address** (typically found in DS register)
    2. Offset value, called **effective address**

• The offset value (or effective address) is specified directly as part of the instruction, usually indicated by a var name
• The assembler calculates the offset & maintains a symbol table that stores the offset values of all vars used in the program.
• In direct mem addressing, 1 of the operands refers to the mem location & the other operand references a register:

```
ADD BYTE_VALUE, DL ; Adds the register in the memory location
MOV BX, WORD_VALUE ; Operand from the memory is added to register
```

Indirect Memory Addressing

Indirect Memory Addressing:

- Generally used for vars containing several elements (such as arrays)
- This addressing mode uses the computer's ability of *Segment:Offset* addressing
    - Base registers (EBX, EBP, BX, BP) & index registers (DI, SI) are used for this purpose

```
MY_TABLE TIMES 10 DW 0 ; Allocates 10 words (2 bytes) each initialized to 0
MOV EBX, [MY_TABLE] ; Effective Address of MY_TABLE in EBX
MOV [EBX], 110 ; MY_TABLE[0] = 110
ADD EBX, 2 ; EBX = EBX +2
MOV [EBX], 123 ; MY_TABLE[1] = 123
```

Variables

Variables:

- NASM provides various **define directives** for reserving storage spsace for vars
    - A define directive can be used to reserve & initialize 1 or more bytes
- For **allocating storage space for initialized data**, syntax:

```
[var-name]   define-directive    initial-value
```

    - 5 basic forms of define directive:

Directive	Purpose	Storage Space
DB	Define Byte	allocates 1 byte
DW	Define Word	allocates 2 bytes
DD	Define Doubleword	allocates 4 bytes
DQ	Define Quadword	allocates 8 bytes
DT	Define Ten Bytes	allocates 10 bytes

→

- For allocating storage space for uninitialized data:
    - **Reserve directives** are used for reserving space here
    - Reserve directives take 1 operand which specifies the number of units of space to be reserved.
    - Each define directive has a related reserve directive:

Directive	Purpose
RESB	Reserve a Byte
RESW	Reserve a Word
RESD	Reserve a Doubleword
RESQ	Reserve a Quadword
REST	Reserve a Ten Bytes

String Instructions

String Instructions:

• There are 5 basic instructions for processing strings:

Instruction	Function
MOVS	Moves 1 Byte, Word, or Doubleword of data from mem location to another
LODS	Loads from mem. If operand is of 1 byte, it's loaded into the AL register. If operand is 1 word, it's loaded into the AX register & a doubleword is loaded into the EAX register
STOS	Stores data from register (AL, AX, or EAX) to mem
CMPS	Compares 2 data items in mem. Data could be of a byte size, word, or doubleword
SCAS	Compares contents of a register (AL, AX, or EAX) w/contents of an item in mem

• For 32-bit segments:
  ▪ String instructions use ESI & EDI registers to point to the source & destination operands respectively
• For 16-bit segments:
  ▪ SI & DI registers are used to point to the source & destination respectively.

• These instructions use the ES:DI (EDI) and DS:SI (ESI) pair of registers
  ▪ The DI & SI registers contain valid offset addresses that refers to bytes stored in mem
  ▪ SI is normally associated with DS (data segment) & DI is always associated w/ES (Extra Segment)
• DS:SI (ESI) & ES:DI (EDI) registers point to the source & destination operands respectively
  ▪ Source operand is at DS:SI (ESI) in mem
  ▪ Destination operand is at ES:DI (EDI) in mem

Repetition Prefixes:

• When the **REP** prefix is set before a string instruction (**EX**: REP MOVSB) causes repetition of the instruction based on a counter placed at the CX register
  ▪ REP executes the instruction & decreases CX by 1 & stops when CX = 0
• The Direction Flag (DF) determines the direction of the operation
  ▪ **CLD** (Clear Direction Flag, DF=0) to make the operation left -> right
  ▪ **STD** (Set Direction Flag (DF=1) to make the operation right -> left
• REP prefix has the following variations:
  ▪ **REP** - unconditional repeat, repeats operation until CX is 0
  ▪ **REPE or REPZ** - Conditional repeat. Repeats the operation while the zero flag indicates equal or equal to zero. Stops repeating when ZF indicates not equal or not zero OR when CX is zero
  ▪ **REPNE or REPNZ** - Conditional repeat. Repeats operation until the zero flag indicates not equal or not zero. Stops when the ZF indicates equal or equal to zero OR when CX is decremented to zero.

Macros

Macros:

- Useful when you need to repeat the same instructions over again
    - Helps prevent having to retype all the instructions all over again
- In NASM, macros are defined with **%macro** & **%endmacro** directives
- Syntax for macro definition:

```
%macro macro_name number_of_params
<macro body>
%endmacro
```

- Macros are invoked by using the macro name along with the necessary parameters

EX:

```
; A macro with two parameters
   %macro write_string 2
      mov   eax, 4
      mov   ebx, 1
      mov   ecx, %1
      mov   edx, %2
      int   80h
   %endmacro

section .text:
   global _start            ;must be declared for using gcc

_start:                     ;tell linker entry point
   write_string msg1, len1
   write_string msg2, len2
   write_string msg3, len3

   mov eax,1                ;system call number (sys_exit)
   int 0x80                 ;call kernel

section .data:
msg1 db 'Hello, programmers!',0xA,0xD
len1 equ $ - msg1

msg2 db 'Welcome to the world of,', 0xA,0xD
len2 equ $- msg2

msg3 db 'Linux assembly programming! '
len3 equ $- msg3
```