x86 Assembly

CS350 – Undergraduate Operating Systems
02/10/2021

Sasan Golchin

Today's agenda

- Hints about the Primer
- x86 Assembly
  - Architecture Overview
  - The AT&T Operand Format
  - Data Movement Instructions
  - Arithmetic Instructions
  - Bitwise Logical Instructions
  - Control Flow Instructions
  - GNU Inline Assembly

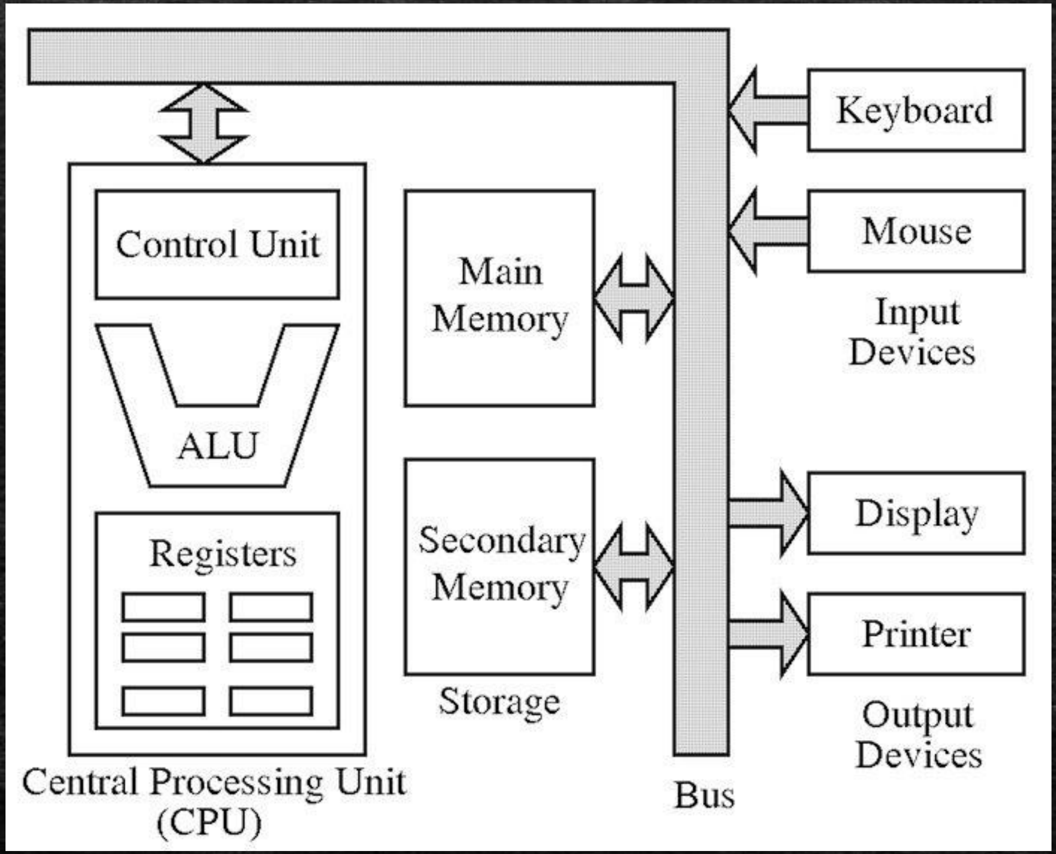
How to go about the Keyboard Driver?

- Start as early as you can!
- Start by extending the provided IOCTL module/test code
- Polling method: You need to disable the native driver (i8042).
- Interrupt Based: Look for Linux kernel functions to define and install your own ISR. Google it!
- At this point, you will get double characters as you are co-existing with the native i8042! Look for a way to bypass that.
- Distinguish key-press from key-release events. Look at the bit definition of the data port (0x60). Read this and then this!
- Work out how to manipulate wait queues in Linux. You need to make the caller task block until there's a keyboard event!

x86 Architecture Overview

Von Neumann Architecture

- Unified memory for Instructions and Data, i.e., the Main Memory
- Move data between Main Memory and Registers
- System Bus:
  - Data Bus: 32-bit wide
  - Address Bus: 32-bit wide
  - Control Bus: R/W, Memory/IO



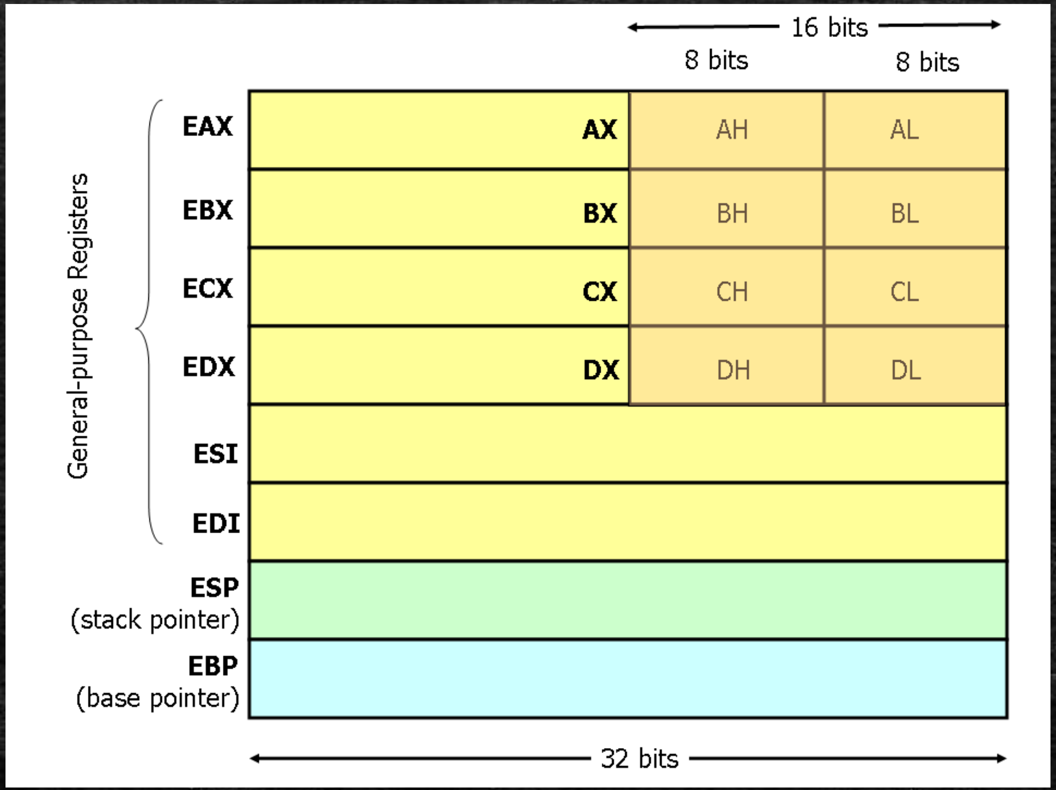
x86 Registers

General Purpose Registers

- EAX, EBX, ECX, EDX
- AX, BX, CX, DX
- AH, AL, BH, BL, …
- ESI, EDI

Special Purpose Register

- ESP: Stack pointer
- EBP: Base pointer
- EIP: Instruction Pointer
- And a lot more: Control Registers, Model-Specific Registers and so on….



Program Organization (1)

- A program is a collection of Data and Code spread over different sections.
- We use GNU-AS to compile our x86 assembly code.
- Some useful GNU-AS directives to organizing your program:
  - **Definition of sections:**
    - .data, .rodata, .bss, .text, .section [name-of-custom-section]
  - **Definition of labels:** Labels are name of constants, variables, functions and anything that can be addressed in the program!
    - [name-of-label]:
  - **Data definition directives:**
    - .byte, .short, .long, .zero, .string, .space, .float, .double

Program Organization (2)

- Examples data section of an assembly program

```
.data /* initialized global variables */
my_byte_arr: /* define an array of bytes labeled my_byte_arr. */
    .byte 64, 0x10, 0xFF
x: /* define a 2-byte integer variable labeled x initialized to 42 */
    .short 42
y: /* define a 4-byte integer variable labeled y = 0x1234ABCD */
    .long 0x1234ABCD
s1: /* define a null-terminated string initialized to "Hello World" */
    .string "Hello World"

.bss /* Uninitialzed global variables */
buf: /* Reserve 256 bytes in a buffer labeled buf */
    .space 256

.text
/* This is where our code (x86 instructions) goes! */
```

An Introduction to x86 Instruction Set

- Data Movement Instructions
  - mov, push, pop, lea, in, out
- Arithmetic Instructions
  - add, sub, inc, dec, imul, idiv
- Logical Instructions
  - and, or, xor, not, neg, shl, shr, sar
- Control-Flow (Branching) Instructions
  - jmp, je, jne, jz, jnz, jg, jge, jl, jle
  - call, ret
  - int
- Many more that we can’t possibly cover in this lab.

The AT&T Syntax - Operands

- The AT&T syntax for instructions w/ more than one operand:

  INSTR src, dst or **INSTR src1, src2, dst**

- Let's consider the MOV instruction. It is used to move data between the registers and the main memory.

- The source and destination operands can be one of the following:
  1. **Registers**: **%[register name]** e.g., `mov %eax, %ebx`
  2. **Immediate**: **$[constant value]** e.g., `mov $0x10, %eax`
  3. **Memory Location**: Follows the format **Offset(Base, Index, Scale)**
     - Offset is an immediate (Constant number or a label)
     - Base and Index are x86 registers
     - Scale is an immediate (Constant number or a label)
     - Memory Location = Offset + (Base + Index*Scale)

The AT&T Syntax - Operands

- Sometimes the instruction operands cannot unequivocally specify how many bytes should the instruction operate on. E.g.,:
- Copying some content from address X in the memory into EAX: Should the CPU copy 1, 2 or 4 bytes from X?
- The following suffixes are attached to the name of the instruction to clarify the size of the operand:
  - **`b`**: One byte
  - **`w`**: A word, i.e., Two bytes
  - **`l`**: A long word, i.e., Four bytes
- Then the instruction mov has four forms: `mov`, `movb`, `movw`, `movl`

Data Movement Instructions (1)

Move!

- Syntax
  - mov <reg>, <reg>
  - mov <reg>, <mem>
  - mov <mem>, <reg>
  - mov <imm>, <reg>
  - mov <imm>, <mem>
- Remember: The first operand is the source and the last one is the destination.
- Also remember that the <mem> operands follow the format: Offset(Base, Index, Scale)

MOV Examples

1. **Direct Memory**: `MOVb var(,1), %ecx`
   - Write the 1-byte value at memory address of the label var into ECX. The suffix b means “a byte”! Label are translated into address values by the linker.
2. **Indirect Memory**: MOVw (%ebx), %eax
   - Write the 2-byte value at memory address stored in EBX into EAX – i.e., Dereference EBX into EAX. The suffix w means “a word” that is 2 bytes.
3. **Indexed Memory**: `MOVl 8(%ebx,%esi,4), %edx`
   - Move the 4-bytes value at address `8+(EBX+ESI*4)` into EDX. The suffix l means “a long word” that is 4 bytes
   - Assume you want to address the 5th element of an array of 4-byte integers located at offset 8 of a data-structure whose base address is at 0x100000

```
mov $0x100000, %ebx          /* Set the base register */
mov $5, %esi                 /* Set the index register */
movl 8(%ebx, %esi, 4), %edx /* access the main memory */
```

Data Movement Instructions (2)

LEA (Load Effective Address): Computes the absolute value of a memory location specified in the **Offset**(**Base**, **Index**, **Scale**) format.

Think of MOV as dereferencing a pointer and LEA as reading the address in a pointer.

Syntax

```
– lea <mem>, <reg32>
```

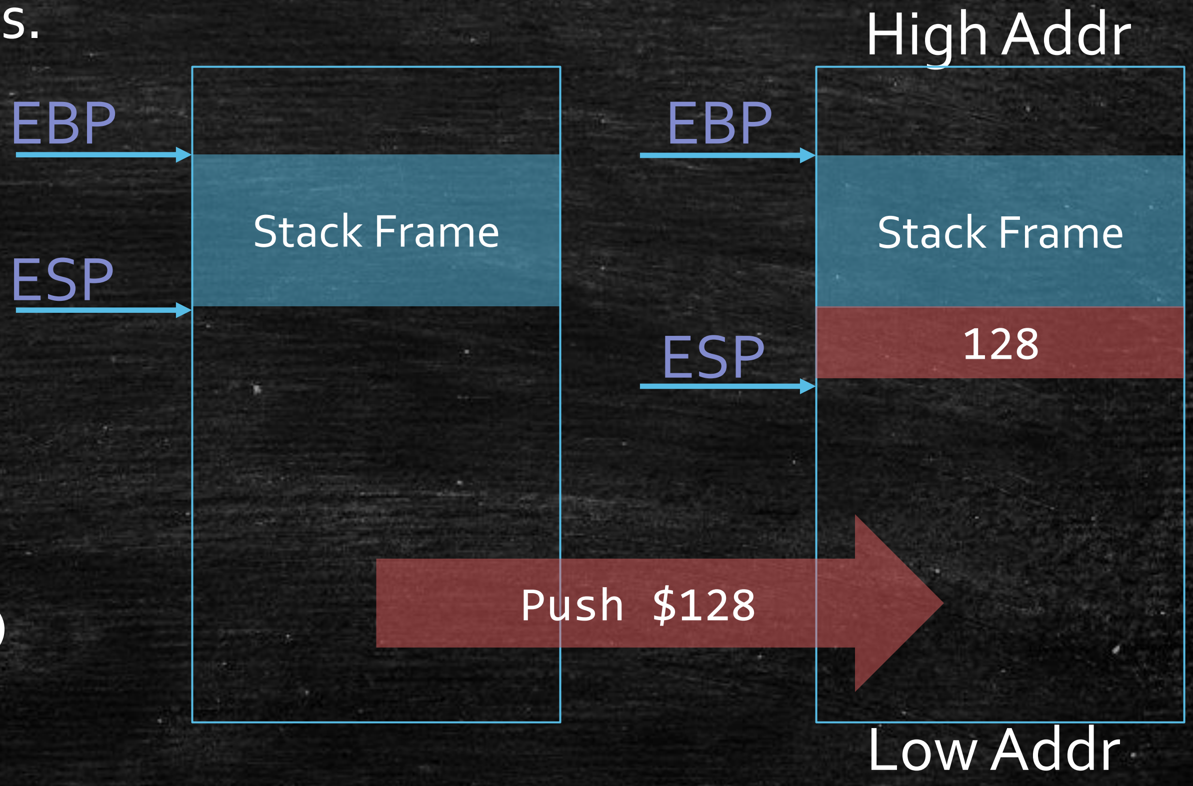
- Examples

```
– lea (%ebx,%esi,8), %edi /* EDI <- EBX+8*ESI */
– lea val(,1), %eax       /* EAX  <- val  */
```

Data Movement Instructions (3)

PUSH: Places its operand onto the top of the hardware supported stack in memory, where ESP points.

- Syntax
  - push <reg>
  - push <mem>
  - push <imm>
- It's Equivalent to:
  - Decrement ESP by 4
  - movl <operand>, (%esp)

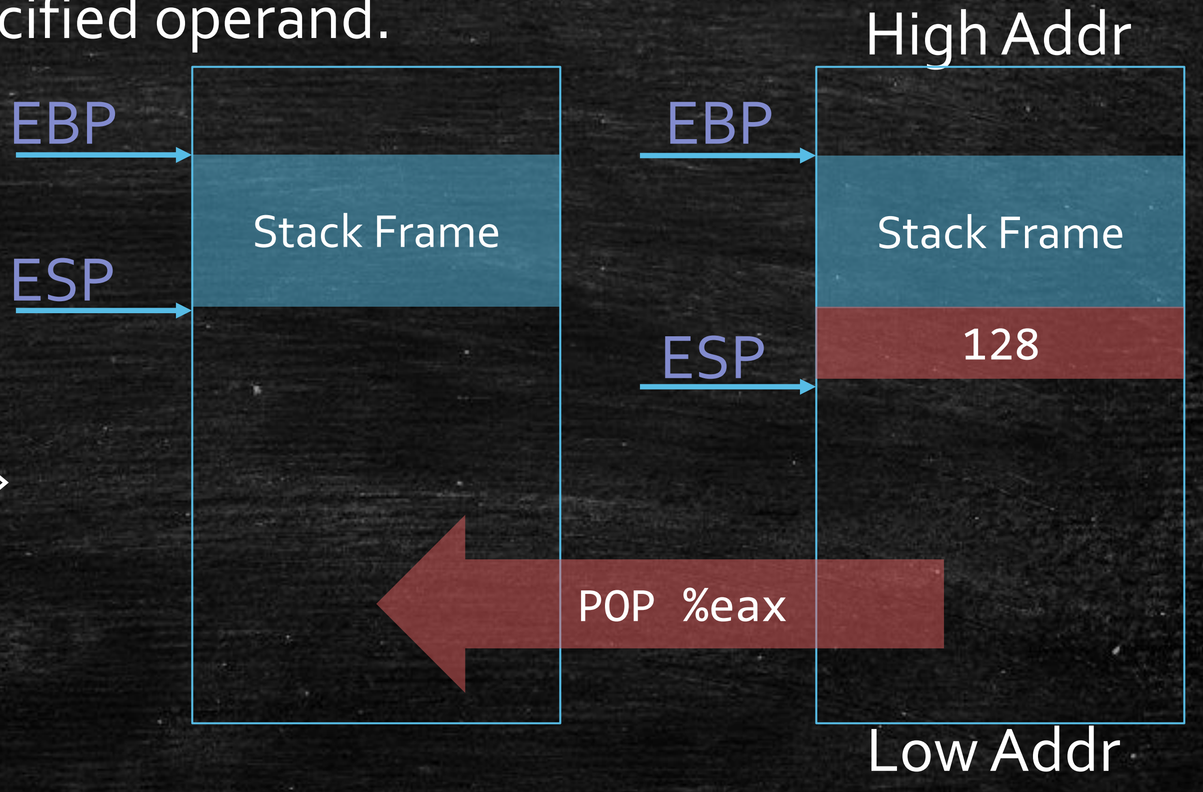


Data Movement Instructions (4)

POP: Removes the 4-byte data element from the top of the hardware-supported stack into the specified operand.

Syntax
- push <reg>
- push <mem>

- It's Equivalent to:
  - movl (%esp), <operand>
  - Increase ESP by 4



Data Movement Instructions (5)

Reading and Write Hardware I/O Ports

- Syntax
  - in <imm8>, <AL, AX, or EAX>
  - in DX , <AL, AX, or EAX >
  - out <AL, AX, or EAX >, <imm8>
  - out <AL, AX, or EAX >, DX
- Examples
  - `inb $0x64, %al /* Read one byte from port# 0x64 into AL */`
  - `inw %dx, %ax /* Read two bytes from port# in DX into AX */`
  - `outb %al, %dx /* Write the byte in AL into the port in DX */`

Arithmetic Instructions (1)

Integer Addition

- Syntax
  - add <reg>, <reg>
  - add <mem>, <reg>
  - add <reg>, <mem>
  - add <imm>, <reg>
  - add <imm>, <mem>
- Examples
  - `add $10, %eax  /* EAX is set to EAX + 10 */`
  - `addb $10, (%eax) /* add 10 to the single byte stored at memory address stored in EAX */`

Arithmetic Instructions (2)

Integer Subtraction

- Syntax
  - `sub <reg>, <reg>`
  - `sub <mem>, <reg>`
  - `sub <reg>, <mem>`
  - `sub <imm>, <reg>`
  - `sub <imm>, <mem>`
- Examples
  - `sub %ah, %al  /* AL <- AL - AH */`
  - `sub $55, %eax /* EAX <- EAX - 55 */`

Arithmetic Instructions (3)

Increment, Decrement

- Syntax
  - `inc <reg>`
  - `inc <mem>`
  - `dec <reg>`
  - `dec <mem>`
- Examples

```
– dec %eax      /* subtract 1 from the contents of EAX */
– incl var(,1) /* add 1 to the 32-bit int. at location var */
```

Arithmetic Instructions (4)

Integer Multiplication

- Syntax
  - `imul <reg32>, <reg32>`
  - `imul <mem>, <reg32>`
  - `imul <con>, <reg32>, <reg32>`
  - `imul <con>, <mem>, <reg32>`
- Examples
  - `imul (%ebx), %eax /* EAX <- EAX * 32-bit value @ mem[EBX] */`
  - `imul $25, %edi, %esi /* ESI <- EDI * 25 */`

Arithmetic Instructions (5)

Integer Division

- Syntax
  - `idiv <reg32>`
  - `idiv <mem>`
- Divides the contents of the 64-bit integer EDX:EAX by the specified operand value. The quotient result of the division is stored into EAX, while the remainder is placed in EDX.
- Examples
  - `idiv %ebx     /* EDX:EAX / EBX */`
  - `idivw (%ebx) /* EDX:EAX / <16-bit value at mem[EBX]> */`

Logical Instructions (1)

Bitwise AND, OR, XOR

- Syntax
  - and <reg>, <reg>
  - and <mem>, <reg>
  - and <reg>, <mem>
  - and <imm>, <reg>
  - and <imm>, <mem>
  - Similar syntaxes for OR and XOR
- Examples
  - `and $0x0F, %eax /* clear all but the last 4 bits of EAX. */`
  - `xor %edx, %edx  /* set the contents of EDX to zero. */`

Logical Instructions (2)

Bitwise Logical NOT, 2’s Complement Negation

- Syntax
  - not <reg>
  - not <mem>
  - neg <reg>
  - neg <mem>
- Examples
  - not %eax /* flip all the bits of EAX */
  - neg %eax /* EAX is set to (- EAX) */

Logical Instructions (3)

Shift Left (SHL) and Right (SHR, SAR)

- Syntax
  - shl <imm8>, <reg>
  - shl <imm8>, <mem>
  - shl %cl, <reg>
  - shl %cl, <mem>
  - Similar syntaxes for SHR and SAR
- Examples
  - shl $1, %eax /* EAX = EAX << 2 ≡ EAX *= 2 (if the most significant bit is 0) */
  - shr %cl, %ebx /* EBX = EBX >> CL (LOGICAL SHIFT TO RIGHT) */
  - sar %cl, %ebx /* EBX = EBX >> CL ≡ EBX <- floor(EBX/(2^CL)) */

Control Flow Instructions (1)

Modify the Instruction Pointer (EIP) based on result of the last arithmetic state

- Syntax
  - cmp <reg>, <reg> (Equivalent to sub <reg>, <reg> but does not change the destination value)
  - cmp <mem>, <reg>
  - cmp <reg>, <mem>
  - cmp <imm>, <reg>
  - je <label> (jump when equal)
  - jne <label> (jump when not equal)
  - jz <label> (jump when last result was zero)
  - jnz <label> (jump when last result was non-zero)
  - jg <label> (jump when greater than)
  - jge <label> (jump when greater than or equal to)
  - jl <label> (jump when less than)
  - jle <label> (jump when less than or equal to)
  - jmp <label> (unconditional jump)

Control Flow Instructions (2)

- Example - A count-up FOR loop:

```
begin:
    xor %ecx, %ecx /* Zero out the counter register */
    mov (%esi), %eax /* Store the final count in EAX */
loop:
    /* DO SOMETHING HERE */
    inc %ecx
    cmp %eax, %ecx
    jl loop /* if counter < final_count then jump to loop */
```

- Example - A count-down FOR loop:

```
begin:
    mov (%esi), %ecx /* Store the count in ECX */
1:
    /* DO SOMETHING HERE */
    dec %ecx
    jnz 1b /* if ECX != 0 then jump to label 1 before this line */
```

Control Flow Instructions (3)

- Call/Ret : Call a function/Return from a function
- Syntax
  - call <label>
  - ret
- call is equivalent to

```
push %eip /* Save the address of the next instruction on top of the stack */
jmp foo   /* jump to the label foo, i.e., name of the destination function */
```

- ret is equivalent to

```
pop %eip    /* Retrieve the return address from the stack-top */
```

- More on x86 Calling Convention: Here

GNU Inline Assembly

- Can’t do everythin in C! E.g., How to make a system call (INT 0x80) in C? :/
- So, what to do now?
  1. Make object files from Assembly and link them with your C code!
  2. Use inline Assembly!

GNU Inline Assembly

- Basic syntax: You can have something like the following in your C functions:
  - asm (“assembly instruction”);
  - asm (“assembly instruction 1\n\t”
    “assembly instruction 2\n\t”);
- Example

```
asm ("pushl %eax\n\t“
     "movl $0, %eax\n\t“
     "popl %eax");
```

GNU Inline Assembly

- Extended syntax: We can let the compiler decide what registers to use and transfer data between registers and our C variables!
  - `asm (“statements” : outputs : inputs : clobbered);`
- Format of outputs and inputs:
  - `“flags”(variable_name), “flags”(variable_name), …`
- Flags:
  - `“r” or “q”: Use a register for as an input operand`
  - `“=r” or “=q”: Use a register as an output operand`
  - `“m”, “=m”: Memory input/output operand`
  - `“a”, “b”, “c”, “d”, “S”, “D”, “N”: Use registers EAX, EBX, ECX, EDX, ESI, EDI, and 0-255 immediate value, respectively`

GNU Inline Assembly

- Example 1: x_times_5 = x + (4*x)
  - `asm ("leal (%1,%1,4), %0"  : "=r"(x_times_5) : "r"(x) );`
  - `asm ("leal (%%ebx,%%ebx,4), %%ebx" : "=b"(x) : "b"(x) );`
- Example 2: Read one byte from an I/O port
  - `asm ("inb %1,%0" : "=a"(value) : "Nd"(port_no));`
- More info? Here
- Still need more info? Here

References

- X86 Assembly Guide by the Flint Group @ Yale
- Brennan's Guide to Inline Assembly
- GCC-Inline-Assembly-HOWTO by Sandeep.S