
x86 Assembly Review

Revised February 24, 2021

General Purpose Registers

- EAX (AL, AH, AX) Stores return value
- EBX (BL, BH, BX)
- ECX (CL, CH, CX) Loop counter
- EDX (DL, DH, DX) Used with EAX in multiplication, division

More General Purpose Registers

- ESI Source pointer
- EDI Destination pointer
- ESP Stack pointer
- EBP Base pointer

Other Registers

- EIP Instruction pointer

- EFLAGS Status register
 - ZF Zero Flag
 - CF Carry Flag
 - OF Overflow Flag

MOV

- MOV EAX, EBX
- MOV EAX, 0x0
- MOV EAX, [0x400000]
- MOV EAX, [EBX + ESI * 4]

LEA

- “Load Effective Address”
- Moves a pointer into a register, does not dereference

- `LEA EAX, [EBX + 8]`
 - Puts `EBX + 8` into `EAX`

- `MOV EAX, [EBX + 8]`
 - Dereferences `EBX + 8` and puts value into `EAX`

LEA vs MOV

```
_start:  mov     ebx, message
         lea     eax, [ebx]
         mov     ecx, [ebx]

        section .data

message: db     "Hello, World", 10
```

Arithmetic Instructions

- ADD EAX, 0x10
- SUB EAX, EBX
- INC EAX
- DEC EAX

More Arithmetic Instructions

- `MOV EAX, 0x2`
- `MUL 0x4`
 - Multiplies EAX by 4, stores upper 32 bits in EDX and lower 32 bits in EAX

- `MOV EDX, 0x0`
- `MOV EAX, 0x9`
- `DIV 0x3`
 - Divides EDX:EAX by 3, stores result in EAX and remainder in EDX

- CDQ is also used for division!

Logical Operator Instructions

- `XOR EAX, EAX`
 - What does this do?

- `AND EAX, 0xFF`
 - What does this do?

- `OR EAX, EBX`

Bit Shifting Instructions

- SHL EAX, 0x2
- SHR EAX, EBX
- ROL EAX, 0x4
- ROR EAX, EBX

Conditional Instructions

- `CMP EAX, EBX`
- `TEST EAX, 0x10`
- `TEST EAX, EAX`

Rep Instructions


- **REPE CMPSB**
 - Compare ESI and EDI buffers
- **REP STOSB**
 - Initialize all bytes of EDI buffer to the value stored in AL
- **REP MOVSB**
 - Copy ESI to EDI
- **REPNE SCASB**
 - Search EDI for the byte in AL

PUSH in Assembly Language

- What does PUSH actually do?


- **PUSH myVal**

- **SUB ESP, 4**



Subtract 4 from the stack pointer
("make room" on the stack)

- **MOV [ESP], myVal**



Copy the value into that
new space on the stack

POP in Assembly Language

- What does POP actually do?

- **POP myRegister**

- **MOV myRegister, [ESP]**

Copy the value off the stack into the register

- **ADD ESP, 4**

Add 4 to the stack pointer
(move the stack back “up”)

CALL in Assembly Language


- What does CALL actually do?

- **CALL myFunc**

- **PUSH &nextInstruction**


- SUB ESP, 4

- MOV [ESP], &nextInstruction



Push the address in memory you'll want to return to

- **JMP myFunc**



Jump to where the function you're calling resides in memory

RET in Assembly Language

- What does RET actually do?

- **RET**

- **POP EIP**



Pop the return address into EIP

- Trusting that whatever's at the top of the stack is the return address
 - When you execute the next instruction it looks at EIP to see what to do next

What is Cdecl?

- The calling convention for the C programming language
- Calling conventions determine
 - Order in which parameters are placed onto the stack
 - Which registers are used/preserved for the caller
 - How the stack in general is handled

Simple Cdecl Example – Code

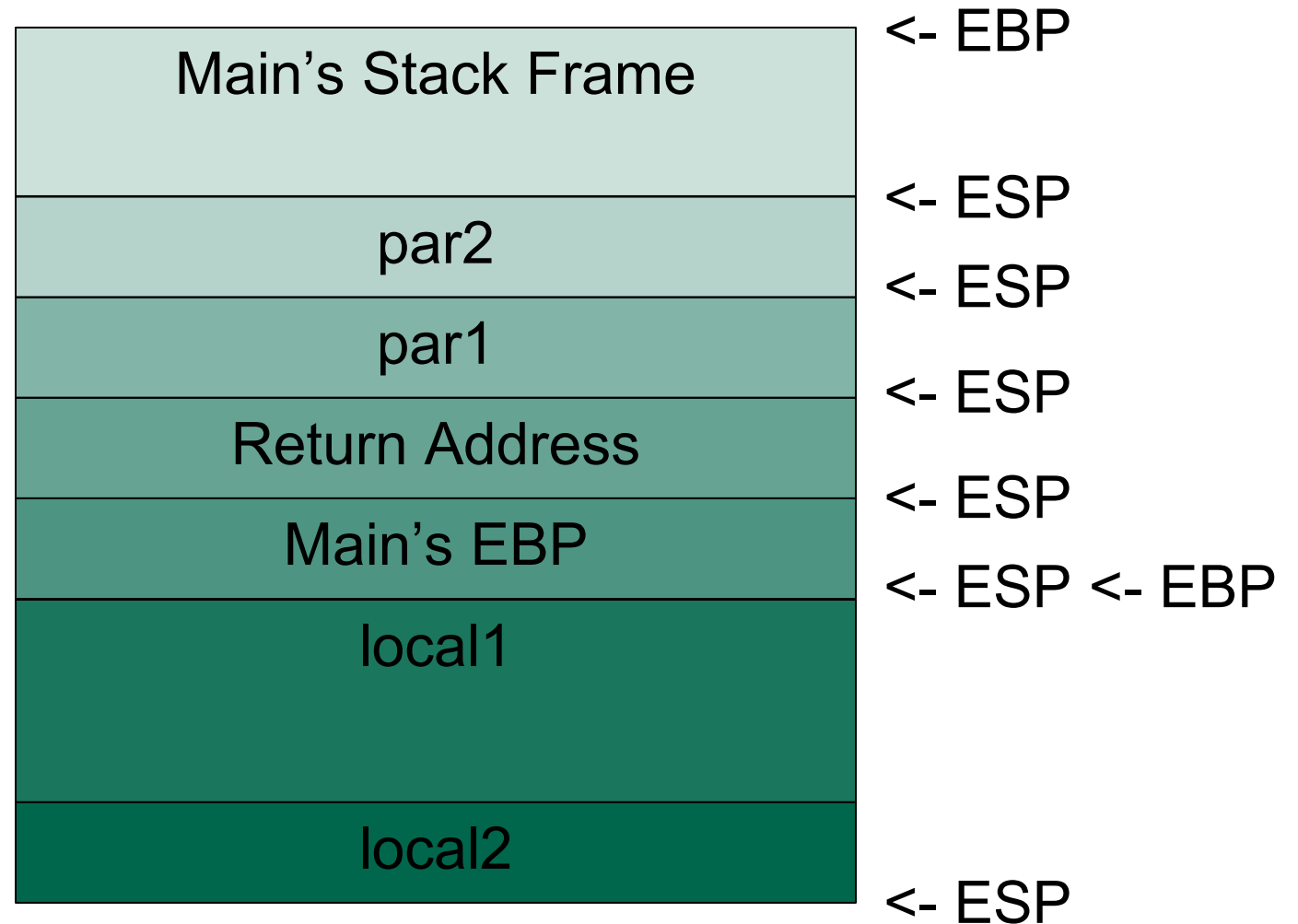
```
int myFunc(char *par1, int par2)
{
    char local1[64];
    int local2;
    return 0;
}
```

```
int main(int argc, char **argv)
{
    myFunc(argv[1], atoi(argv[2]));
    return 0;
}
```

- What actually happens on the stack when this program is run?
 - What variables are allocated first?
 - How does the stack grow?

Simple Cdecl Example – Calling

- PUSH par2
- PUSH par1
- CALL myFunc
- PUSH EBP
- MOV EBP, ESP
- SUB ESP, 68



Simple Cdecl Example – Returning

- MOV ESP, EBP
- POP EBP
- RET

