Introduction to Assembly

Slides by Marius Wiik

Based on PDF by Erlend Helland Graff

Introduction

- We will focus on 32 bits x86 ISA (Instruction Set Architecture)
  - An ISA enables multiple different hardware to implement the same functionality in different ways. That is why you can play your games on both AMD and Intel chips.

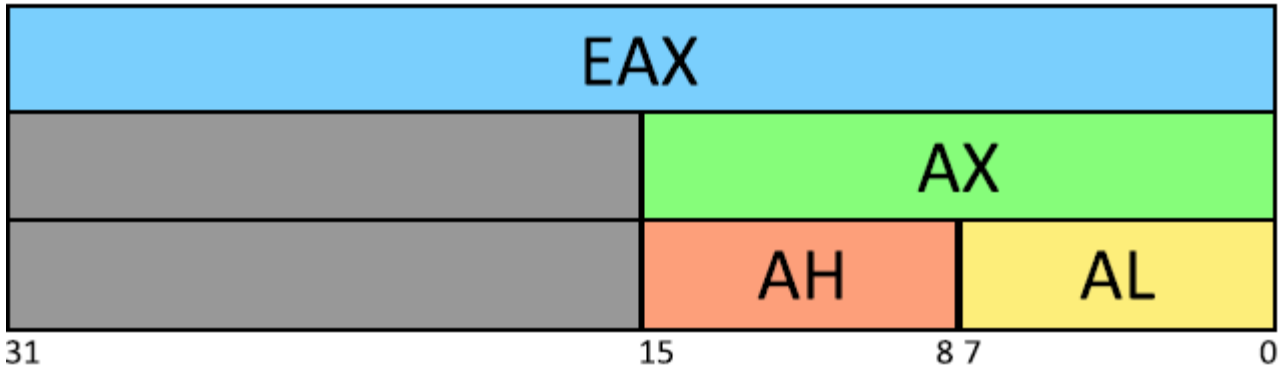
- Memory can be viewed as a collection of bits
  - This ISA can only access groups of bits at a time
    - 1 byte, 2 bytes or 4 bytes at a time.
  - 32 bit architecture means we have a 32bit address space → We can address 2^{32} addresses
    - Means we can access address 0x00000000 – 0xFFFFFFFF (4 GB)

Introduction (2)

- On a low level, programming is all about moving numbers around
- Assembly can be considered as human readable binary.
- Assembly does not have variables in the traditional sense!
  - We use temporary, very quick storage units called *registers.*
  - We will only use integer registers.
- Moving numbers between registers is extremely quick, only a few clock cycles
  - Accessing memory is slow in comparison, several hundred clock cycles
- **The x86 ISA does not provide a way to move numbers from one memory location to another, so we need to use the registers.**
- We want to use as few memory accesses as possible, so utilize those registers! ☺

Introduction (3)

- x86 has eight GPRs (general purpose registers)
- These are 16 bit integer registers
  - There is also an extended version. Prefixed with “E”
    - “EAX, ECX etc”
- On the registers ending with “X” we can access each of the two bytes separately
  - Switch the X with H (high) or L (low)



Name	Legacy Usage
AX	Accumulator register
CX	Counter register
DX	Data register
BX	Base register
SP	**Stack pointer**
BP	**Base pointer**
SI	Source index register
DI	Destination index register

Some assembly

- Two syntaxes are used (Intel assembly and AT&T / GAS)
  - We will use AT&T / GAS
  - Be careful when looking up resources online. Check which syntax the examples are using
  - An easy way to check is to see if the registers are prefixed with “%”, for example %eax. Then it is AT&T / GAS
- The instruction INC is used to increment the value of a destination operand. The letter that comes after the instruction “INC” decides which data size we’re working with
  - b for byte, w for word (2 bytes) l for long (4 bytes)

```
1   incb %al      # AL++
2   incw %bx      # BX++
3   incl %ecx     # ECX++
```

Some assembly (2)

- Some instruction examples
- Immediate values are prefixed with «$» and comments starts with «#»

```
 9   add $2, %al      # AL += 2
10   addw $4, %bx     # GX += 4
11   addl %edx, %ecx # ECX += EDX
```

```
13   subb $2, %al     # AL -= 2
14   subw $4, %bx     # BX -= 4
15   subl %edx, %ecx # ECX -= EDX
```

Instruction	Effect
INC	Increment value
DEC	Decrement value
ADD	Add source to destination
SUB	Subtract source from destination

Some assembly (3)

- Another important instruction is “MOV”. Used to move data into a register, memory space, between two registers or between a register and memory
  - **Remember: You can not move data between two memory locations**

```
17  # EDX contains the memory address
18  # of a long int (32 bit)
19  movl $4, %eax          # EAX = 4
20  movl %eax, %ebx        # EBX = EAX = 4
21  movl %eax, (%edx)      # *(EDX) = EAX
```

- On the last line we copy the contents of EAX into the **MEMORY ADDRESS** that resides in EDX.

Some assembly (4)

- As in C, we can also manipulate pointers in assembly
  - Remember if you incremented a pointer in C it incremented a multiple of the size of the data type it pointed to?
  - Assembly doesn't. And that is easier to relate to (kinda).
  - Assembly increments as if it was an integer (as far as the assembler knows it is)
  - Denoted by having parenthesis around the register containing the memory address

```
1  movl %eax, (%ebx)              # *(EBX) = EAX
2  movl (%ebx), %eax              # EAX = *(EBX)
3  movl %eax, 6(%ebx)             # *(EBX + 6) = EAX
4  movl -6(%ebx), %eax            # EAX = *(EBX - 6)
5  movl %eax, -4(%ebx, %ecx)      # *(EBX + ECX - 4) = EAX
6  movl %eax, 8(%ebx, %ecx, 4)    # *(EBX + ECX * 4 + 8) = EAX
7  movl %eax, 12(, %ecx, 8)       # *(ECX * 8 + 12) = EAX
```

Some assembly (5)

- Instructions are read and executed sequentially from memory
- The ISA guarantees that the effect of one instruction will be active before the next instruction is executed.
- A register called the program counter (PC) or instruction pointer (IP) is used to know the memory address of the current instruction being executed.
  - In x86 this register is called EIP
  - We cannot access this register directly but we can change the value using jump instructions
- The jump instruction is used to break the sequential execution flow
  - If you do a function call, use loops or conditional statements.
  - We can define places to jump by using labels

Some assembly (6)

- Two types of jump: conditional or unconditional
- Can use CMP instruction to create conditional jumps

Instruction	Jump if	Meaning
JE	Equal	A == B
JNE	Not equal	A != B
JB	Below (unsigned)	A < B
JA	Above (unsigned)	A > B
JBE	Below or equal (US)	A <= B
JAE	Above or equal (US)	A >= B
JL	Less (signed)	A < B
JG	Greater (signed)	A > B
JLE	Less or equal (S)	A <= B
JGE	Greater or equal (S)	A >= B

```
1  first:
2      movl %eax, %ebx
3      jmp second
4  third:
5      incl %ecx
6      jmp last
7  second:
8      movl %ebx, %ecx
9      jmp third
10 last:
11     decl %ecx
```

Some assembly (7)

- The CMP instruction compares two operands and updates an internal EFLAGS-register based on the comparison. The conditional JMP-instructions will read that information and act based on it

- If the instruction is CMP op1, op2
- The jump will happen if:
  - Op2 JAE op1
  - Reversed of what you might think?

```
1  movl $4, %eax    # EAX = 4
2  movl $10, %ebx   # EBX = 10
3  cmpl %eax, %ebx  # Compare operands
4  jae jump_here    # Jump if EBX >= EAX
5
6  #Code below the jump statement will
7  #not be executed
8
9
10 jump_here:
11     #Code here will be executed
```

Two examples

```
1  movl $1, %ebx # b = 1;
2  cmpl $5, %eax # compare a with 5...
3  jne after     # ..and if a != 5, jump to .after
4  # Else, if a == 5...
5  addl $2, %ebx # b += 2
6  .after:
7  incl %ebx     # b++
```

```
1  int b = 1;
2  if (a == 5)
3  {
4      b += 2;
5  }
6  b++;
```

```
 1  movl $0, %eax # int a = 0;
 2  movl $0, %ecx # int i = 0;
 3  loop:
 4      cmpl $10, %ecx # if ECX...
 5      jge end; # ...is greater or equal 10, jump to end
 6      addl %ecx, %eax # a += i;
 7      incl %ecx # i++;
 8      jmp loop # goto loop;
 9  end:
10      incl %eax # a++;
```

```
1  int i, a = 0;
2  for (i = 0; i < 10; i++)
3      a += i;
4
5  a++;
```

The stack

- The registers ESP and EBP mentioned earlier are used to maintain the stack. This is a very important data structure provided by the operating system and is used to store auto variables (variables that are local to a function), temporary data, function arguments and return addresses for function calls.
- The stack grows downwards in memory from a high address to a low
  - ESP (Stack pointer) always points to the top of the stack (lowest memory address) and will therefore always contain the memory address of the last byte put on the stack.
  - Data below ESP is considered garbage.
- When we enter a function we create a new stack frame. A limited portion of the total program stack that is local for this particular function
- EBP is used to keep track of the bottom of the stack, so that the stack is limited from the base pointer (highest address) to the stack pointer (lowest address)

The stack (2)

- The ISA defines two operations, push and pop to make working with the stack easier.
  - Push adds something to the stack
  - Pop removes something from the stack

```
1  pushw $5      # Push the number 5 to stack
2  pushl %eax    # Push the contents of eax to stack
3  popl %ebx     # Pop the last pushed value and store it in EBX
4  popw %cx      # Pop the number 5 and store it in CX
```

- Note that pushb or popb is not allowed as the stack is 16-bit aligned.
- **It is important to maintain symmetry between push and pop operations**
  - **Remove as much as you have added!**

The stack (3)

- The stack operations affect the stack pointer (ESP), since it always points to the top of the stack

- Hopefully, you will not have to deal with endianness, but note this:
  - X86 is a little-endian architecture.
- Lets say the register EAX has the value 0x01234567 and we want to move the value into memory address 0xABCD0000
  - How will the four bytes that is the value of EAX be located in relation to the address?

Address	Byte
0xABCD0000	0x01
0xABCD0001	0x23
0xABCD0002	0x45
0xABCD0003	0x67

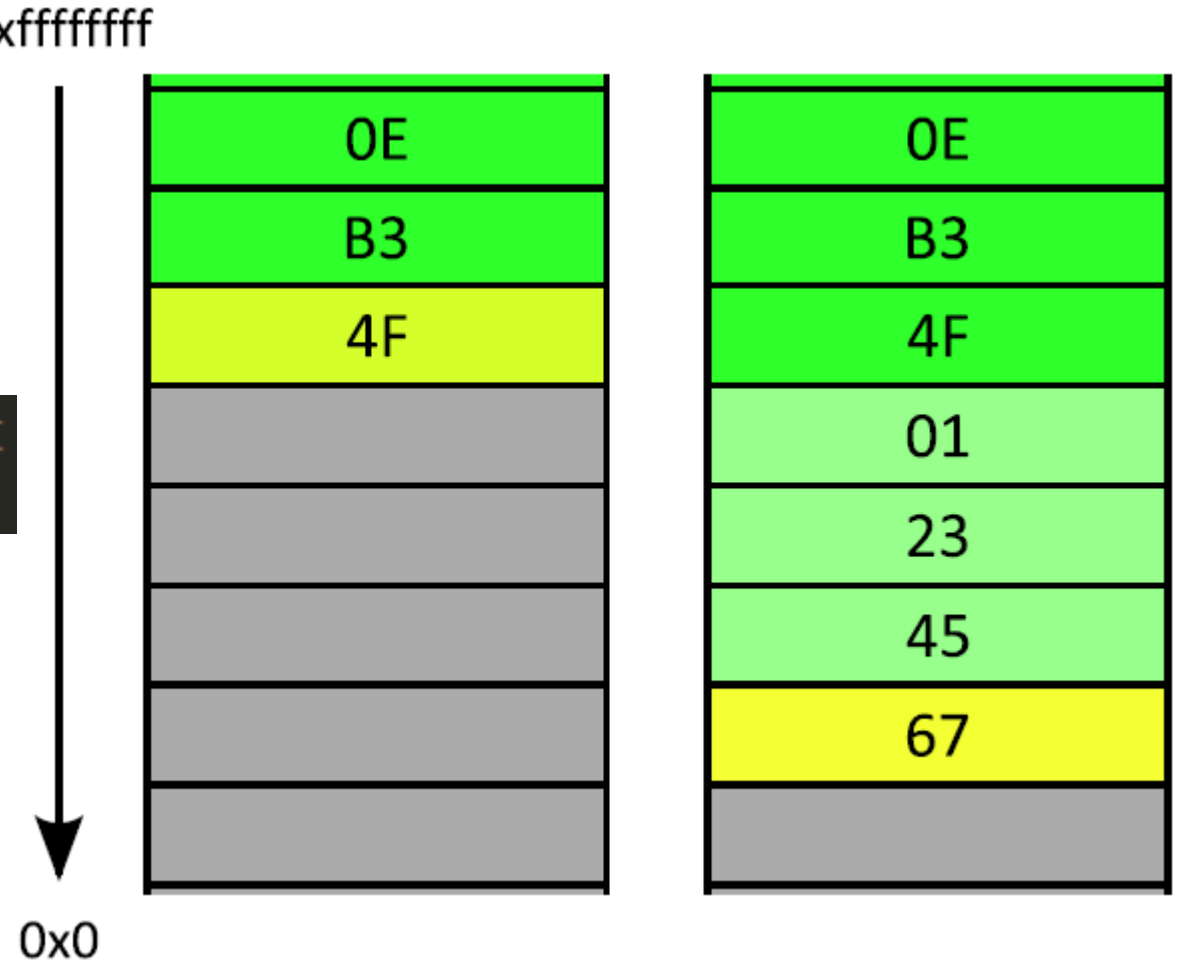
Address	Byte
0xABCD0000	0x67
0xABCD0001	0x45
0xABCD0002	0x23
0xABCD0003	0x01

The stack (4)

- This shows the stack before and after
- The yellow fields represents the ESP register
- At any time, you can remove data you have pushed by adding to the ESP register

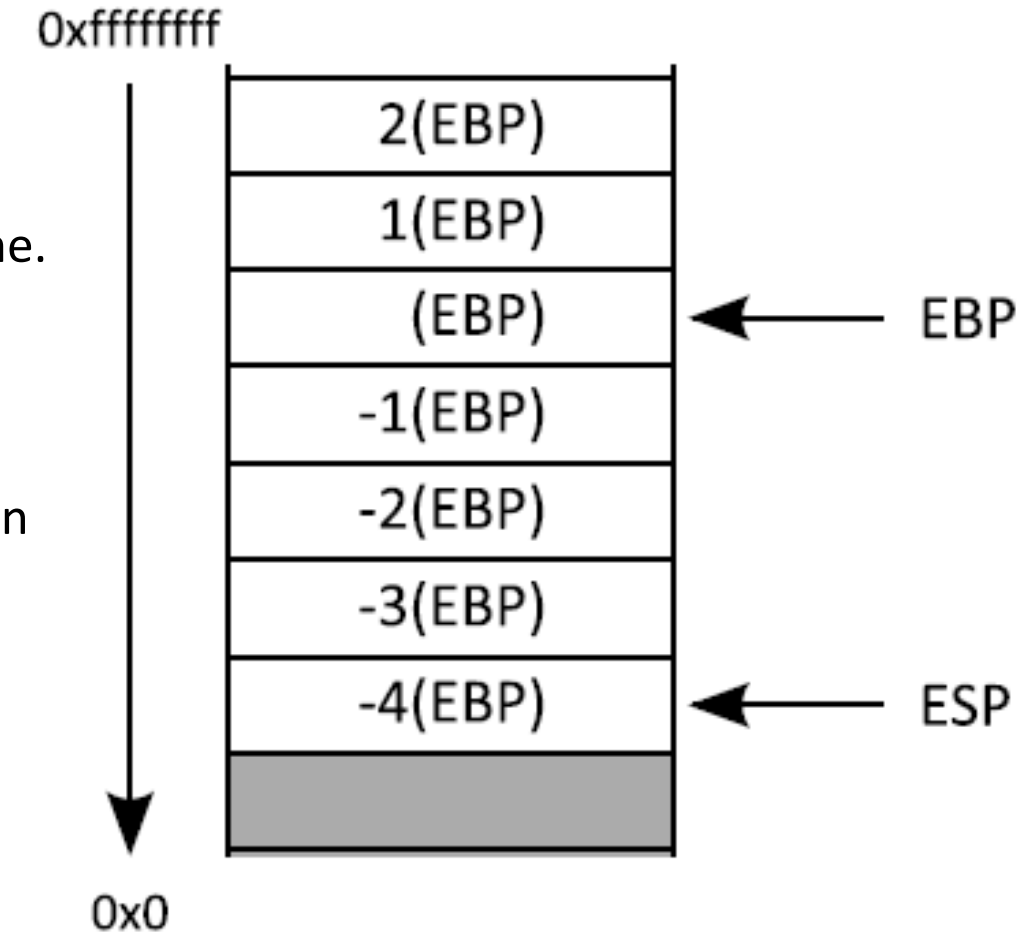
```
1  #Count 4 bytes up from the stack
2  addl $4, %esp
```

- Can also reserve space on the stack by subtracting



The stack (5)

- With all the adding, subtracting, pushing and popping the ESP will be in constant movement
- The base pointer will however never move within a stack frame.
- To create a new stack frame you need to:
  - Store the old value of EBP on the stack (push)
  - Copy the value of stack pointer into the base pointer
- This is generally the first thing you do when entering a function
- The last thing? Pop that EBP! ☺



Functions

- Different calling conventions for functions. We are going to use C declaration(CDELC)
- This means there are rules to follow when calling and creating functions.
- All arguments are put on the stack from right to left.
- CALL is almost the same as an unconditional jump, but CALL will first push EIP on the stack.
  - This means that the function can return to where it was called from.
  - RET returns from a function after popping four bytes of the stack and into EIP
    - This really illustrates why the push/pop symmetry needs to be followed. What happens if the bytes popped into EIP is some random number you have stored on your stack? ☺

```
1  int my_add(int first, int second){
2      return first + second
3  }
```

```
1  pushl $15      # Push "second" argument onto stack
2  pushl $27      # Push "first" argument onto stack
3  call my_add    # Call the function 'my_add'
4  #This code executes when the function returns
5  addl $8, %esp # Pop arguments from stack
```

Functions (2)

- CEDCL also states that the registers EAX, ECX and EDX is caller saved. That means that a function can freely use these registers without thinking about the contents
  - Remember: Your entire program shares the same registers
- It also means that if you want to use EBX, ESI or EDI you first have to store the old values before touching them, and restore the values before returning (push, pop)
  - If you don’t you will probably mess up something for the caller!
- It is also defined that the **return value** of a function should always be stored in EAX before returning (Does not apply for floating point numbers, but don’t think about that)

Example

```
1  int my_add(int first, int second){
2      return first + second
3  }
```

```
 1  #Calling the function
 2  #Save the caller saved registers
 3  pushl %eax
 4  pushl %ecx #Only if they are important
 5  pushl %edx
 6  #Push arguments
 7  pushl $15
 8  pushl $27
 9  call my_add #call function
10  #After returning, pop arguments
11  addl $8, %esp #Can also use popl instruction
12  #And restore registers
13  popl %edx
14  popl %ecx
15  popl %eax
```

Example – The function

```
17  my_add:
18      pushl %ebp              #Preserve old stackframe
19      movl %esp, %ebp         # Create a new stack frame
20      pushl %ebx
21      pushl %esi              #Preserve callee saved registers so
22      pushl %edi              #We can use them (even though we dont)
23      movl 8(%ebp), %eax      # EAX = 1st argument (int first)
24      movl 12(%ebp), %ecx     # ECX = 2nd argument (int second)
25      addl %ecx, %eax         # EAX += ECX
26      popl    %edi
27      popl    %esi            # Restore callee saved registers
28      popl    %ebx
29      popl    %ebp            # Restore caller's stack frame
30      ret                     # Return value is in EAX
```

Resources

- https://en.wikipedia.org/wiki/X86_calling_conventions
- https://en.wikibooks.org/wiki/X86_Assembly/GAS_Syntax
- https://www.cs.swarthmore.edu/~newhall/cs31/resources/IA32_Cheat_Sheet.pdf

Questions?