Introduction to x86 disassembly

Defcon 24

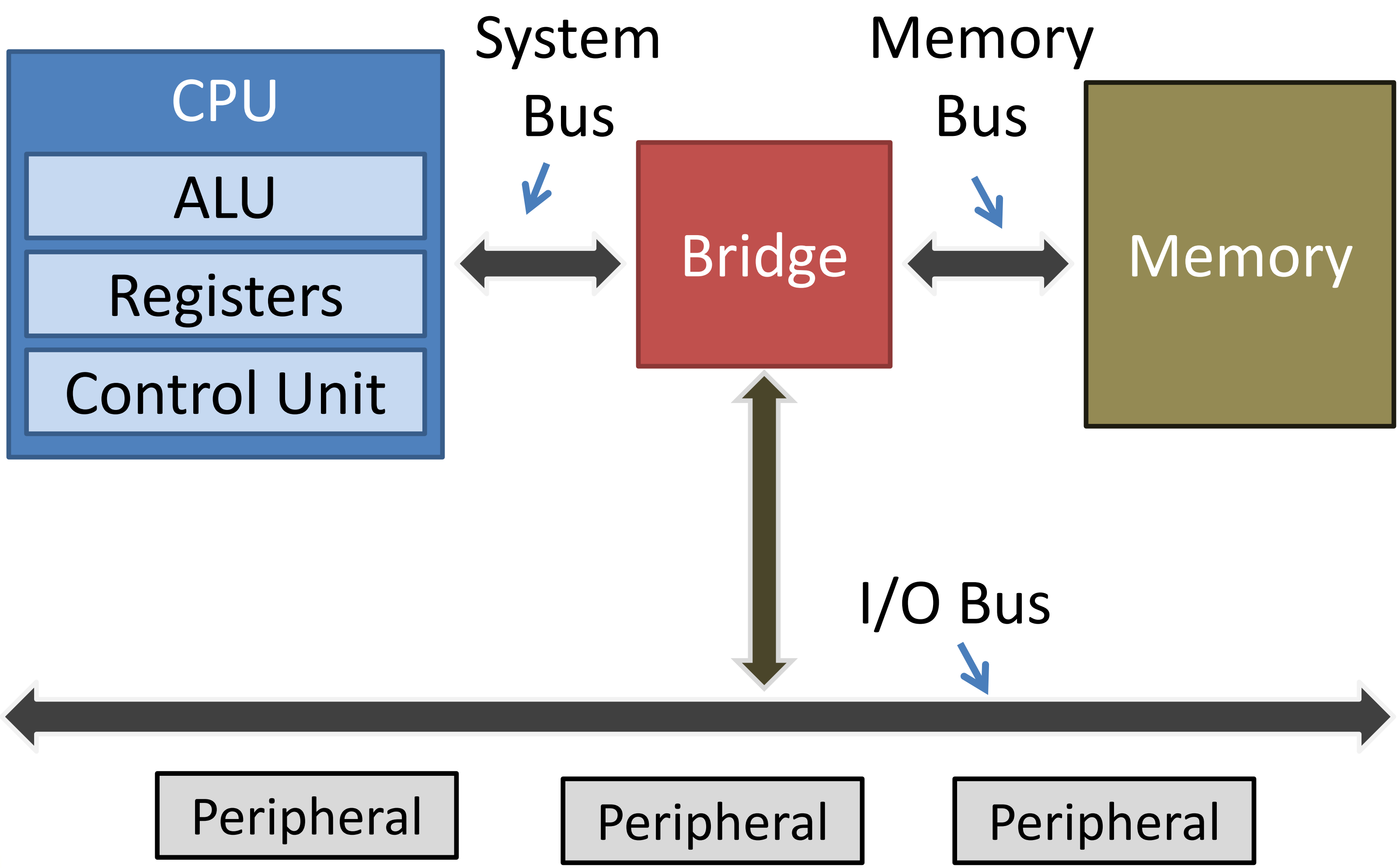
2016

DazzleCat Duo

github - dazzlecatduo

Architecture Overview

Computer Architecture



Computer Architecture

- Memory
  - Stores data (moderately fast)
  - A linear array of bytes, accessed via their address
- Bridges
  - Coordinate communication between buses
- Buses
  - Transfer information
- Peripherals
  - Communicate with the outside world
  - Do anything else the system might need

Computer Architecture

- CPU (Central Processing Unit)
  – Processes information
  – ALU (Arithmetic logic unit)
    - Does math
  – Registers
    - Store data (very fast)
    - Register size: 1 word
    - Generally *named*, rather than *addressed*
  – Control unit
    - Executes code

Computer Architecture

- Registers vs. Memory
- Registers serve the same purpose as memory
  - They store data
  - Memory
    - Moderate access speed
    - Cheap
    - Lots
  - Registers
    - Fast
    - Expensive
    - Few
- Your program/data/etc sit in memory, while registers are used to process very small pieces at a time

Abstractions

- All of this is normally abstracted away from the programmer
- The Operating System manages…
  - Processes
    - Makes it look like your program has control of the processor
  - Memory
    - Makes it look like your process has it
  - Files
    - Makes them look like a sequence of bytes

Abstractions

- But none of these things are true
- Goal of learning assembly is to start seeing the world as it really is

Source: http://www.einstorm-xp1-home-biz.com/Morpheus.jpg

Assembly

- Everything the CPU does is through digital logic
  - On/Off, 1/0
- Including running your program
- The series of bits that control the CPU is *machine code*
  - A bunch of numbers
  - Define a set of instructions to run

Assembly

- The machine code for a standard “hello world” :

```
55 89 e5 83 e4 f0 83 ec 10 b8 b0 84 04 08 89 04
24 e8 1a ff ff ff b8 00 00 00 00 c9 c3 90
```

- This is a series of instructions for the processor to execute
- It flips the right transistors to calculate information, fetch data from memory, send signals to the system buses, communicate with the graphics card, and print out “hello world”
  – With help from additional machine code

Assembly

- Machine code controls the processor on the most detailed possible level
  - Moves information in and out of memory
  - Moves information to and from registers
  - Controls the system bus
  - Controls the ALU, control unit, etc

Assembly

- We want to directly control the CPU to leverage its full power
  - But we don’t want to write a bunch of numbers that we can’t hope to understand
- *Assembly* is a shorthand, more legible version of machine code
  - Uses mnemonics to save us from memorizing which numbers do what
  - “sub” (subtract) instead of 0x83
  - “add” (add) instead of 0x81

Assembly

Machine Code	Assembly
`55`	`push %ebp`
`89 e5`	`mov %esp,%ebp`
`83 e4 f0`	`and $0xfffffff0,%esp`
`83 ec 10`	`sub $0x10,%esp`
`b8 b0 84 04 08`	`mov $0x80484b0,%eax`
`89 04 24`	`mov %eax,(%esp)`
`e8 1a ff ff ff`	`call 80482f4`
`b8 00 00 00 00`	`mov $0x0,%eax`
`c9`	`leave`
`c3`	`ret`
`90`	`nop`

Assembly

- Writing in pure machine code is fun, and has its uses, but is difficult and uncommon
- Much more practical to write in assembly
- An *assembler* is a tool that translates from assembly to machine code; this process is called *assembling*
- A *disassembler* is a tool that translates from machine code to assembly; this process is called *disassembling*

Assembly

- C code:
  - `int x=1, y=2, z=x+y;`
- Assembly code:
  - `mov   [ebp-4], 0x1`
  - `mov  [ebp-8], 0x2`
  - `mov    eax, [ebp-8]`
  - `mov    edx, [ebp-4]`
  - `lea    eax, [edx+1*eax]`
  - `mov    [ebp-0xc], eax`
- Machine code:

```
c7 45 fc 01 00 00 00 c7 45 f8 02 00 00
00 8b 45 f8 8b 55 fc 8d 04 02 89 45 f4
```

Compilation Process

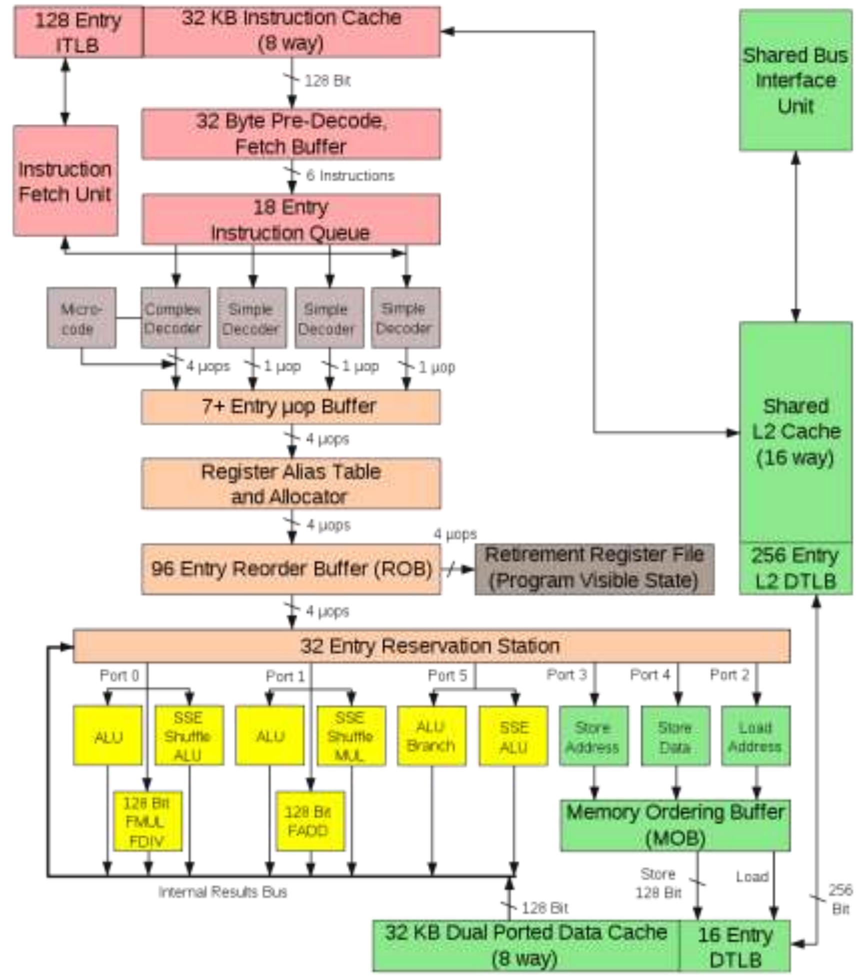
- Source code is compiled into assembly code
- Assembly code is assembled into machine code
- Compilers have been doing all this for you

Instruction Set Architecture

- The *Instruction Set Architecture (ISA)* defines
  - Processor registers
    - One register, or 200? 8 bits, or 128?
  - Address and data format
    - Do I grab a byte from memory at a time? Or 500?
  - Machine instructions
    - Can I add and subtract? Check for equality? Halt?
- Indirectly defines the *assembly language*
  - What low level instructions we have available, what those instructions do

Microarchitecture

- A *microarchitecture* is the way a given instruction set is implemented on a processor



Intel Core 2 Architecture

Source: http://en.wikipedia.org/wiki/File:Intel_Core2_arch.svg

Computer Architecture

- Collectively, the instruction set architecture and microarchitecture define the *computer architecture*

Computer Architecture

- There are…
  - Thousands of instruction set architectures
  - Thousands of microarchitectures
  - Thousands of computer architectures

Computer Architecture

- Architectures can usually be broadly divided into two categories
  - *Reduced Instruction Set Computing (RISC)*
  - *Complex Instruction Set Computing (CISC)*

RISC vs. CISC

- RISC
  - Small set of simple instructions
  - Generally…
    - Cheaper to create
    - Easier to design
    - Lower power consumption
    - Physically smaller
- CISC
  - Large set of powerful instructions
  - Generally…
    - More expensive
    - Hard to design
    - Higher power requirements
    - Pysically larger

RISC vs. CISC

- Hypothetical example
  – Multiply by 5, RISC vs. CISC
- CISC:
  – mul [100], 5
- RISC:
  – load r0, [100]
  – mov r1, r0
  – add r1, r0
  – add r1, r0
  – add r1, r0
  – add r1, r0
  – mov [100], r1

RISC vs. CISC

- Neither RISC nor CISC is better or worse than the other
  - Both have advantages, and disadvantages
  - A CISC instruction may take 100 RISC instructions to implement
  - But a CISC instruction may run at 1/200th the speed of the RISC instructions
  - Or consume 1000x the power
  - Or take a year to design

(Some of) The Major Players

- RISC
  - ARM (examples: phones, tablets)
  - MIPS (examples: embedded systems, routers)
  - PowerPC (examples: original Macs, Xbox)
- CISC
  - x86 (examples: consumer computers)
  - Motorola 68k (examples: early PCs, consoles)

Introduction to x86

Introduction to x86

- Why x86?
  - Can build, run, and play with on your own computer
  - Extremely popular, billions of systems, market dominance
  - Core of familiar operating systems (Windows, Mac, Linux)

x86

- Your laptops, desktops, workstations, servers, etc, all use the x86 architecture
- When you buy a new processor to upgrade your computer, that's an x86 processor
- Makes it an ideal choice for studying assembly and computer architecture

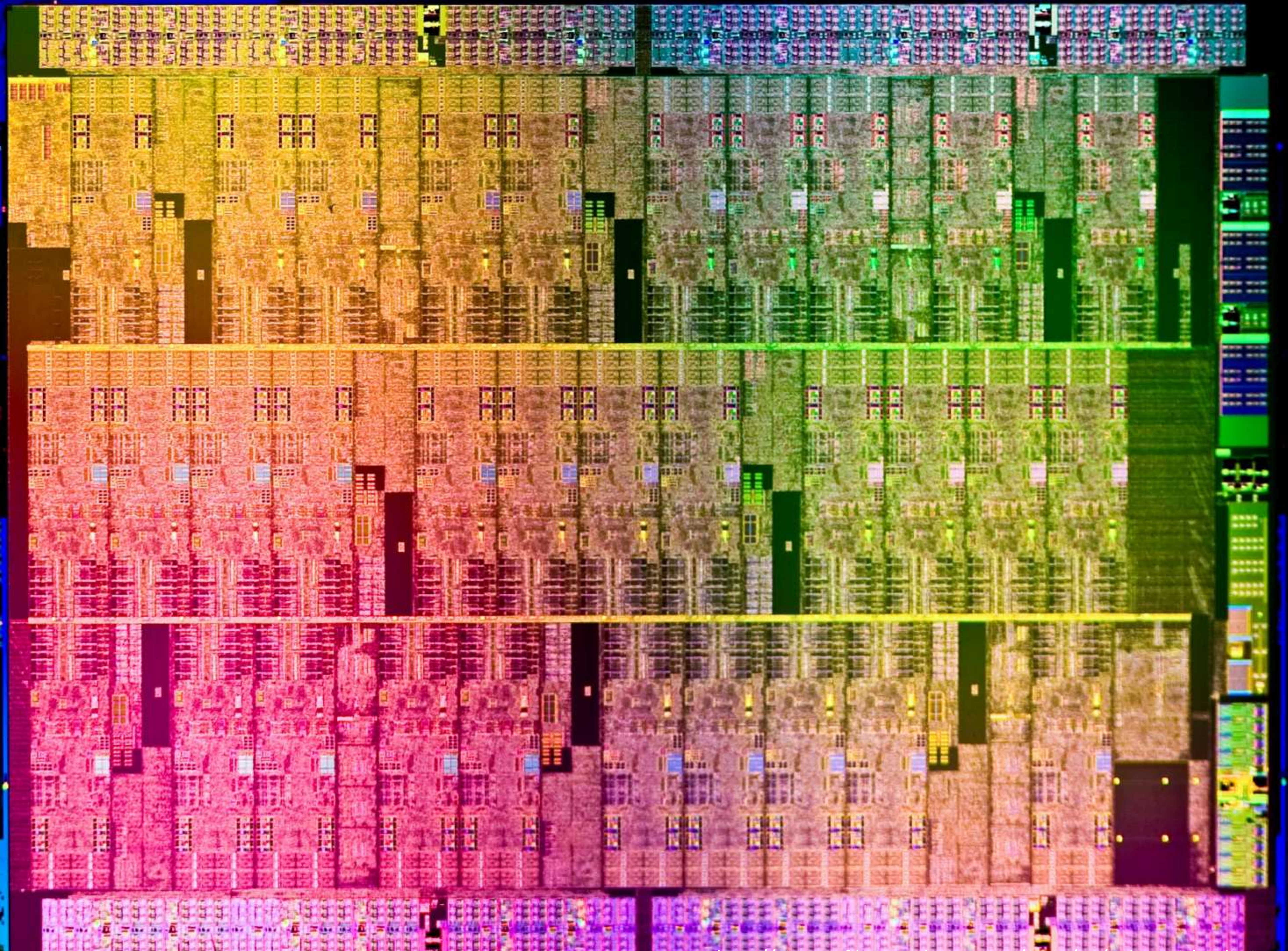
History of x86

- Intel 8080
  - 8 bit microprocessor, introduced in 1974
- Intel 8086
  - 16 bit microprocessor, introduced in 1978
- Intel 80386
  - 32 bit microprocessor, introduced in 1985
- Intel Prescott, AMD Opteron and Athlon 64
  - 64 bit microprocessor, introduced in 2003/2004

Source: http://en.wikipedia.org/wiki/File:KL_Intel_D8086.jpg

History of x86

- Goal of design: backwards compatibility
  - Every generation adds new features
    - But doesn’t break or remove any of the old
    - Even when the old features were later determined to be useless/broken/etc
  - Code that runs on the original 8086 processor can run unmodified on the latest 9th generation architectures
- Has resulted in an immense, complex, interesting architecture



Source: http://gigaom.files.wordpress.com/2010/06/aubrey_isle_die.jpg

A Complex Architecture

- Intel Software Developer's manual…
  - http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf
- 4000 pages, doesn't even begin to scratch the surface
- Goal in class: give you the basics

x86

- Today, “x86” generally refers to all architectures based off of the original 8086
  - The 8086, which contains the 16 bit architecture
  - The 80286, which contains the 32 bit architecture and the 16 bit architecture
  - The 80886, which contain a 64 bit architecture, 32 bit architecture, and 16 bit architecture
- The term “x64” refers specifically to the 64 bit version of the x86 architecture
- We will study the 32 bit x86, since it is the most universal

x86

- CISC
- Little Endian

Assembly Syntax

Assembly Syntax

- The ISA defines registers, data format, machine instructions, etc
- But it doesn't actually define what code should look like
  - It might define a "multiply" instruction, and how it works
  - But it doesn't say anything about how we would *write* such an instruction in assembly
    - "multiply", "mul", "MUL", etc

Assembly Syntax

- There is no standard syntax for assembly
- Not even a standard syntax for a particular architecture's assembly language
- Entirely defined by the *assembler*
- Hundreds of variations

Rivals

- Two main branches of x86 syntax
  - AT&T
    - Used by gcc
  - Intel
    - Used by Intel
- They both have their pros and cons
- Then hundreds of smaller variations specific to an assembler

Assembler Syntax

- In this class:
  - The assembler is NASM
    - The “netwide assembler”
    - Extremely popular
    - Very powerful
    - Very flexible
  - So we’ll teach NASM’s x86 syntax
    - Uses Intel syntax

Assembler Syntax

- Almost universally true in assembly, and with NASM
  - Lines do not end in a semi-colon
  - Semi-colons are used to start a single line comment
  - instruction ; comment

x86 Registers

Registers

- Registers are how the processor stores information
- The processor can access memory, but since the system's memory is not part of the actual processor, this is *extremely* slow
- Registers are contained in the actual processor, they are very fast (access at the same speed as the processor)

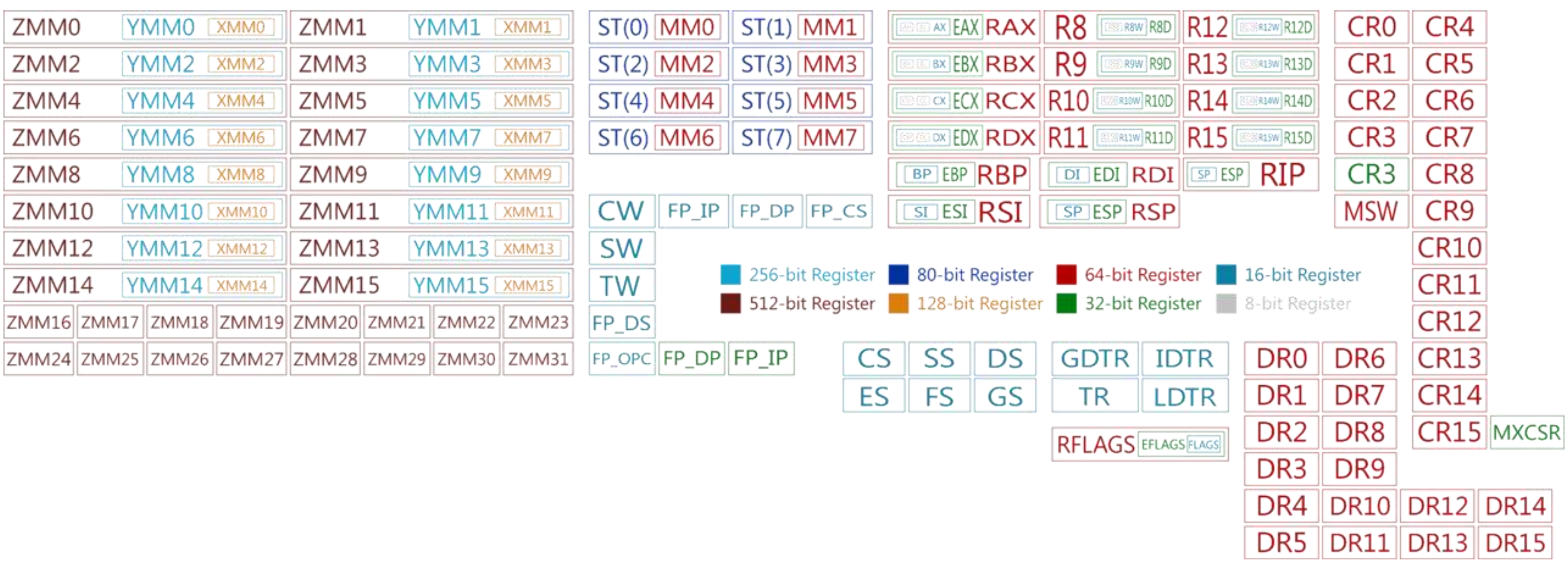
Registers

- You can think of registers as 32 bit variables
  - Each register has its own name
  - Can be modified, etc
- But there are a very limited number of registers
  - They must be shared by the whole program
  - When they run out, they need to store their information back to memory
  - Typical execution:
    - Fetch data from memory, store in registers
    - Work with data
    - Save data back to memory
    - Repeat

Registers

- Registers are generally divided into two categories
  - *General Purpose Registers (GPRs)*
    - Used for “general” things
    - Store data, addresses, etc
  - *Special Purpose Registers (SPRs)*
    - Store program state

x86 Registers



Source: http://en.wikipedia.org/wiki/File:Table_of_x86_Registers.png

x86 Registers

- Fortunately, you do not need to know all those
- The ones you will need to know…
- x86 GPRs:
  - eax, ebx, ecx, edx, esi, edi, ebp, esp
- x86 SPRs:
  - eip, eflags

x86 Registers

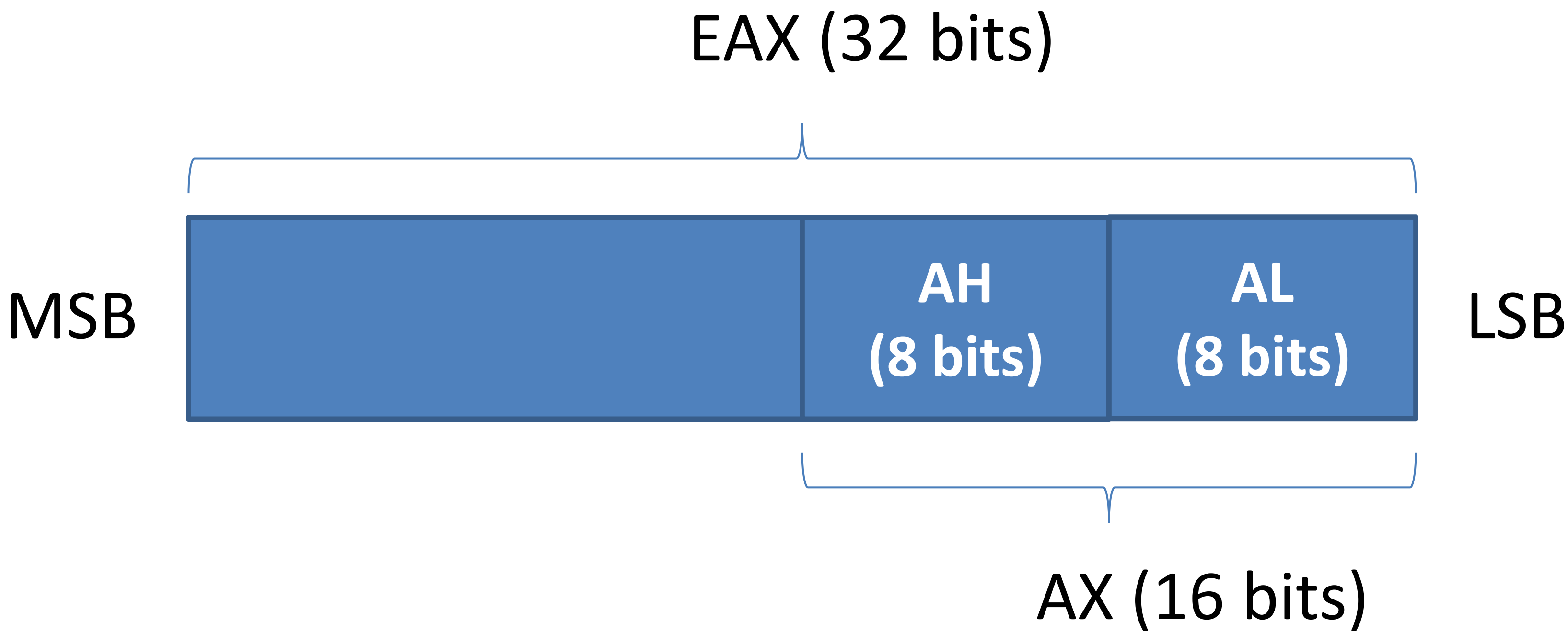
- The registers we will discuss are 32 bit registers
- Notice that these register names begin with "e"
  - This is for "extended" – the latest 32 bit processors "extended" their 16 bit predecessors

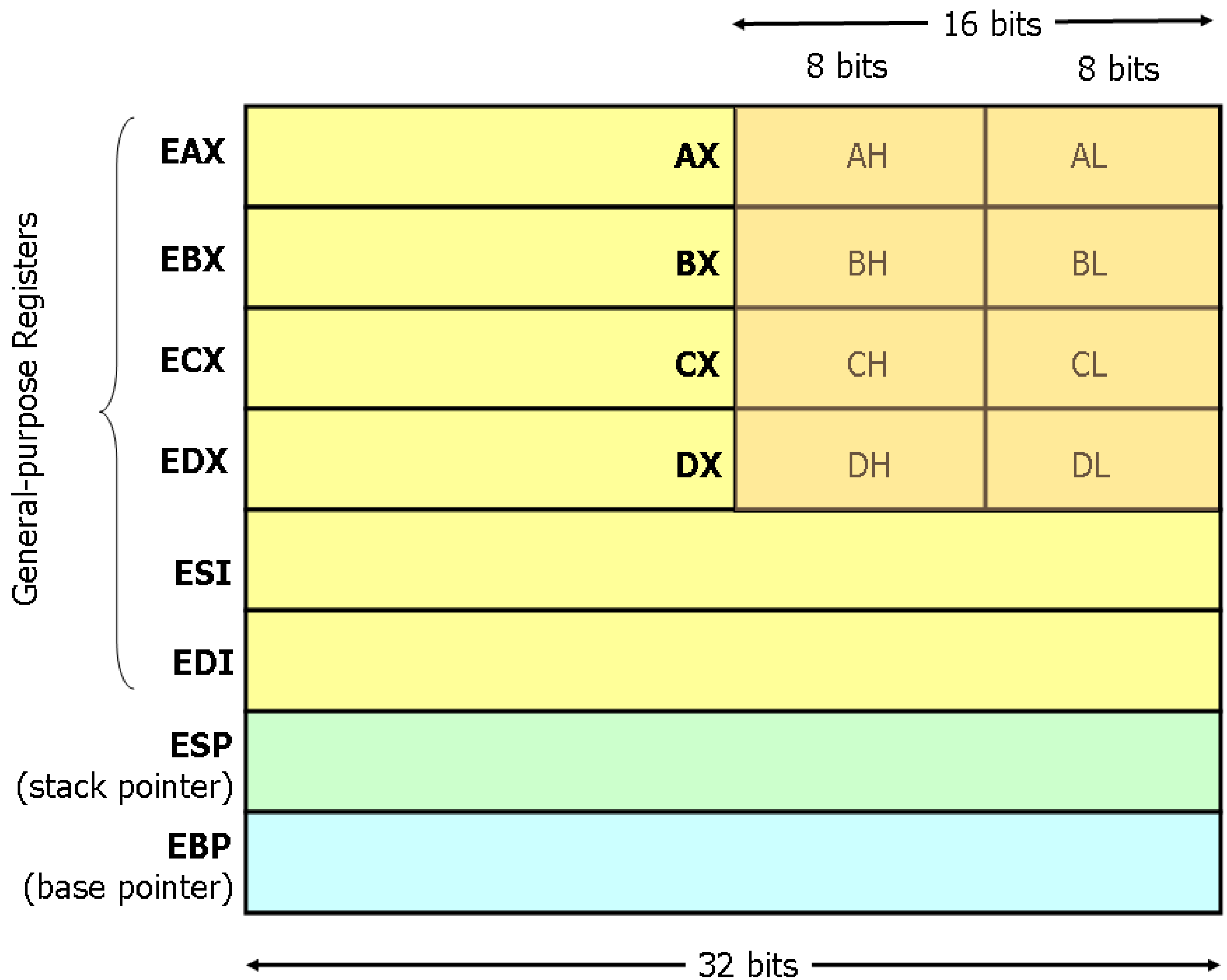
x86 Registers

- You can access the *low order* 16 bits of the register by removing the “e” from the register name (for example, “ax” is the low 16 bit of “eax”)
- For the register names that end in “x” (eax, ebx, ecx, edx), you can access the low order 8 bits of the 16 bit register using “l” (al, bl, cl, dl), and the high order 8 bits of the 16 bit register using “h” (ah, bh, ch, dh)

x86 Registers

- Example, accessing pieces of the eax register





github - dazzlecatduo

x86 Registers

- eax, ebx, ecx, edx
- Registers can be accessed in parts
- e*r*x – Refers to 32 bit register
- *r*x – Referes to the lower 16 bits of e*r*x
- *r*h – Refers to the top 8 bits of the *r*x bit register
- *r*l – Refers to the lower 8 bits of the *r*x register

EAX

- 32 bit GPR
- The “accumulator” register
  – Traditionally used to accumulate results of arithmetic operations
  – e.g. eax+=ebx; eax+=ecx; eax+=edx;
- ax: low 16 bits of eax
- al: low 8 bits of ax
- ah: high 8 bits of ax

EBX

- 32 bit GPR
- The "base" register
  - Traditionally used to store the base of an address
  - e.g. accessing array index 5: [ebx + 5]
- bx: low 16 bits of ebx
- bl: low 8 bits of bx
- bh: high 8 bits of bx

ECX

- 32 bit GPR
- The “counter” register
  - Traditionally used to count
  - e.g. “for (i=0; i<10; i++)” – i might be assembled to the ecx register
- cx: low 16 bits of ecx
- cl: low 8 bits of cx
- ch: high 8 bits of cx

EDX

- 32 bit GPR
- The “data” register
  – Traditionally used to store and work with data
  – e.g. sub edx, 7
- dx: low 16 bits of edx
- dl: low 8 bits of dx
- dh: high 8 bits of dx

EBP

- 32 bit GPR
- The “base pointer” register
- Stores the address of the base of the stack frame
- bp: low 16 bits of ebp

ESP

- 32 bit GPR
- The "stack pointer" register
- Stores the address of the top of the stack frame
- sp: low 16 bits of esp

EBP/ESP

- Intel classifies EBP and ESP as GPRs
- But many people would consider them SPRs
- GPRs are used in arithmetic, memory accesses, etc
- SPRs have some other special purpose
- EBP/ESP control the stack, so they have another special purpose
- You would generally not modify them like you would EAX/EBX/ECX/EDX

EIP

- 32 bit SPR
- The “instruction pointer” register
- Stores the address of the next instruction to execute
- ip: low 16 bits of eip

EFLAGS

- 32 bit SPR
- The “flags” register
- Stores “flags” (bits specifically indicating true/false) about system state and information about the results of previously executed instructions
- flags: low 16 bits of eflags

Accessing Registers

- When you write C code…

```c
int x = 5;
int y = 2;
int z = x + y;
```

- … x, y, and z are variables stored in memory. But to work with them, they need to be moved into registers first. The compiler chooses which registers to use for this.

Accessing Registers

- The EFLAGS and EIP registers cannot be accessed directly this way
  - This is because they store and track system state for you, you are not supposed to need to set them yourself
  - mov eip, 1 ; does not assemble

Initializing Registers

- *registers are not initialized to any specific values* when your function begins
- You must first set them to be the values you need
- Be careful:
  - Setting low bits (e.g. al) does not initialize the high bits

x86 Memory Access

Accessing Memory

- Registers used in this class:
  - eax, ebx, ecx, edx, esi, edi, esp, ebp, eip, eflags
- ebp, esp track the stack, and shouldn’t be used (in this class) for computation
- eip and eflags are special purpose registers that can’t be used for general computation
- That leaves only eax, ebx, ecx, edx, esi, edi
- This isn’t enough to do much
- At some point, the program needs to access memory

Accessing Memory

- In assembly, memory is accessed using [ ] notation
- Examples:
  - [0x12345678]
    - Access the value stored at memory address 0x12345678
  - [eax]
    - Access the value stored at the memory pointed to by eax

Accessing Memory

- [0x12345678]
  - I am accessing memory at address 0x12345678
  - But how much am I accessing?
    - A byte? A word? A double word?
- In some cases, the size of the access is implicit
  - mov eax, [0x1234567]
  - Since I am moving memory into the eax register, and eax is 32 bits, I must be accessing 32 bits of memory
- But in other cases, it is not
  - mov [0x1234567], 1
  - Am I trying to set a byte, word, or doubleword?

Accessing Memory

- If the size of the memory access is not implied by the instruction, it must be explicitly specified with either “byte”, “word”, or “dword”
- Examples:
  - byte [100]
  - Access the single byte at address 100
  - dword [ax]
  - Access the doubleword pointed to by ax

x86 Word Size

- A quirk…
  - Traditionally, the "word" size of an architecture is the size of data that architecture is natively built for
    - A 32 bit architecture like x86 is designed to work with 32 bit data – this would be its word size
  - But the original x86 was 16 bits
    - Had a 16 bit word
    - We still use this definition
  - Even though the architecture works with 32 bit data, 32 bit registers, etc
    - At least when writing assembly, we say a "word" is 16 bits

x86 Word Size

- byte: 8 bits
- word: 16 bits
- dword: 32 bits
- qword: 64 bits
- fword: (not what you think) 48 bits
- tword: 80 bits

Only 3 you need for this class

Accessing Registers & Memory

- Most x86 instructions take operands
- The instruction mnemonic indicates the operation the processor is supposed to perform
- The instruction operands indicate what is used in the operation
- Example: “add eax, ebx”
  - Add ebx to eax
  - add is the mnemonic
  - eax and ebx are the operands

Accessing Registers & Memory

- Instructions (mnemonics) typically accept 0, 1, 2, or 3 operands (depends on the instruction)
- In general, x86 instructions can access any number of registers at once, but at most one memory location at once
- add eax, ebx
  - Accesses two registers at once, valid
- add eax, [0x12345678]
  - Accesses one register, and one memory address, valid
- add [0x12345678], [0x87654321]
  - Accesses two memory addresses at once, not valid

x86 Instructions

x86 Instructions

- Arithmetic
  - add
  - sub
  - mul
  - inc
  - dec
  - and
  - or
  - xor
  - Not
- Stack
  - call
  - return
  - push
  - pop
- Data movement:
  - mov
- Execution flow
  - jmp
  - Conditional jumps
- Comparison
  - test
  - cmp
- Other
  - lea
  - nop

x86 Instructions

- There are hundreds more, but those are the basics we need for this class
- Even this might seem like a lot, but when you think of all the operators (+, -, *, /, %, &&, ||, &, |, ^, !, ~, <, >, >=, <=, ==, ., ->, etc) and keywords (if, else, switch, while, do, case, break, continue, for, etc) you know for any other language, this is trivial

mov

- Move data from one location (memory, register, etc) to another
- Syntax: `mov destination, source`

mov Examples

- `mov eax, 5`
  - Store the value 5 into eax
- `mov eax, [ 1 ]`
  - Copy the 32 bit value at memory address 1 into eax
- `mov dx, [ 0x100 ]`
  - Copy the 16 bit value at memory address 0x100 into dx
- `mov ecx, eax`
  - Copy the contents of eax into ecx
- `mov [ 1984 ], bl`
  - Store the 8 bit value in bl to memory address 1984
- `mov [ eax ], cx`
  - Store the 16 bit value in cx to the memory pointed to by eax; e.g. if eax is 0x777, store cx to location 0x777 in memory

inc, dec

- Increment, decrement by 1
- Syntax:
  - `inc register`
  - `inc [ memory ]`
  - `dec register`
  - `dec [ memory ]`

inc, dec Examples

- `inc eax`
  - Increment the eax register by 1
- `dec dx`
  - Decrement the dx register by 1
- `dec dword [ 0x11223344 ]`
  - Decrement the 32 bit value at 0x11223344 by 1
- `inc word [ ecx ]`
  - Increment the 16 bit value pointed to by ecx by 1

add, sub

- Add and subtract
- Syntax
  - `add destination, value`
  - `sub destination, value`
- Destination can be a register or memory
- Value can be a register, memory, or immediate
- Note: operands must all be same size
  - `add eax, bx` is invalid

add, sub Examples

- `add eax, ebx`
  - Add ebx to eax, store result in eax
- `sub ecx, [ 100 ]`
  - Subtract the 32 bit value at address 100 from ecx, store the result in ecx
  - Note that the memory access is implied to be 32 bits, there is no need to specify “dword”
- `add dword [ edx ], 100`
  - Add 100 to the 32 bit value pointed by edx
  - Note that the *address* is implied to be 32 bits (edx), but the data size must be specified

mul

- Multiply eax by operand, store result in edx:eax
  - edx: high 32 bits of result
  - eax: low 32 bits of result
- Syntax:
  - `mul [ memory ]`
  - `mul register`
- mul *always* uses the eax register as a source
- And *always* stores the result in edx:eax

mul Examples

- `mul eax`
  - edx:eax = eax * eax; (Square eax)
- `mul ebx`
  - edx:eax = eax * ebx;
- `mul dword [ 0x555 ]`
  - edx:eax = eax * (32 bit value at address 0x555)
- `mul byte [ 0x123 ]`
  - edx:eax = eax * (8 bit value at address 0x123)

and, or, xor

- Binary AND, OR, and XOR
- Syntax:
  - `and destination, source`
  - `or destination, source`
  - `xor destination, source`
- Destination can be a register or memory address
- Source can be a register, memory address, or immediate

and, or, xor Examples

- `or eax, 0xffffffff`
  - Set eax to all 1's
- `and dword [ 0xdeadbeef ], 0x1`
  - Mask off low bit of 32 bit value at 0xdeadbeef
- `xor ecx, eax`
  - ecx = ecx ^ eax
  - Evaluate exclusive or of bits in ecx and eax, store result in ecx

and, or, xor Examples

- `xor eax, eax`
  - Fastest way to clear a register in x86
  - Other ways
    - mov eax, 0
    - and eax, 0
    - sub eax, eax
  - Involve extra computation or longer machine encodings, which slow them down

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

not

- Binary NOT
- Syntax:
  - `not register`
  - `not [ memory ]`
- Retrieves the value of the operand, computes its one's complement, and stores it back to the operand

not Examples

- `not ch`
  - Inverts all the bits of ch
- `not dword [ 2020 ]`
  - Inverts all the bits of 32 bit value at address 2020

nop

- “No operation”
- Literally does nothing
- Syntax: `nop`
- Compiles to exactly one byte in machine code (0x90)
- Commonly used for…
  - Timing
  - Memory alignment
  - Hazard prevention
  - Branch delay slot (RISC architectures)
  - A placeholder to be replaced later
  - Hacking (nop sleds)
  - Cracking (nop outs)

lea

- Load Effective Address
- Syntax: `lea destination, [ source ]`
- Computes the address of the source operand, and places it in the destination operand
- Similar to the & operator in C
- Often used for simple math, rather than anything to do with addresses

lea examples

- lea eax, [100]
  - Computes the effective address of [100] (which is 100) and stores it in eax
- lea ecx, [ebx]
  - Computes the effective address of [ebx] (which is ebx) and stores it in ecx

Examples

- Evaluate 0x13 * 0x100 + 0x37 using assembly

```
mov eax, 0x13
mov ecx, 0x100
mul ecx
add eax, 0x37
```

Multiplies eax by ecx, saving result in edx:eax. Could not use immediate value in multiplication, needed a scratch register

Conclusion

- Always a similar pattern
  - Load data from memory into registers
  - Work with the data
  - Store back to memory

x86 reference

- One of my favorite x86 references
- http://ref.x86asm.net/coder32.html

x86

- 8 32 bit registers
- General Purpose Registers
  - eax
  - ebx
  - ecx
  - edx
  - esi
  - edi
- Stack Register
  - esp
- Base Register
  - ebp

Conditional Codes

- Eflags register contains the current state of flags AKA conditional codes
- There are 9 conditional codes on x86
- Flags are used to track the outcome of operations
- Flags are used to conditional execute code

- CF, PF, ZF, SF, OF, AF, TF, IF, DF

Condition Flags

- The most useful two:
  - CF – Carry – Last arithmetic resulted in a carry
  - ZF – Zero – Last arithmetic/logical operation resulted in a zero

x86 Instructions

<reg32> Any 32-bit register (EAX, EBX, ECX, EDX, ESI, EDI, ESP, or EBP)

<reg16> Any 16-bit register (AX, BX, CX, or DX)

<reg8> Any 8-bit register (AH, BH, CH, DH, AL, BL, CL, or DL)

<reg> Any register

<mem> A memory address (e.g., [eax], [var + 4], or dword ptr [eax+ebx])

<con32> Any 32-bit constant

<con16> Any 16-bit constant

<con8> Any 8-bit constant

<con> Any 8-, 16-, or 32-bit constant

Data Movement

- *mov destination, source*
- Move data from source to destination
- *Syntax*
  mov <reg>,<reg>
  mov <reg>,<mem>
  mov <mem>,<reg>
  mov <reg>,<const>
  mov <mem>,<const>
- *Examples*
  `mov eax, ebx` — copy the value in ebx into eax

Data Movement

- *lea – Load Effective Address*
- loads the address of operand2 into operand1
- *Syntax*
  lea <reg32>, <mem>
- *Examples*
  `lea eax, [var]` – address of var is placed into eax

lea examples

- lea eax, [100]
  - Computes the effective address of [100] (which is 100) and stores it in eax
- lea ecx, [ebx]
  - Computes the effective address of [ebx] (which is ebx) and stores it in ecx
- lea eax, [ebx + ecx + 5]
  - Computes the effective address of [ebx + ecx + 5] (which is ebx + ecx + 5) and stores it in eax

lea examples

- Why is this useful?
  - Variables are often stored at offsets from a register
- Example: char s[5];
  - eax may contain the address of s
  - lea ebx, [eax + 2] gives me the address of element 2
  - We could do that with
    - mov ebx, eax
    - add ebx, 2
  - But this is an extra instruction

lea examples

- Why is this useful?
  - Variables are often stored at offsets from a register
- Example: char s[5];
  - eax may contain the address of s
  - lea ebx, [eax + 2] gives me the address of element 2
  - We could do that with
    - mov ebx, eax
    - add ebx, 2
  - But this is an extra instruction

Data Movement

- *leave*
- Sets stack pointer to the base frame address
- *Syntax*
  leave
- *Examples*

```
leave –equivalent to:
                 mov esp,ebp
                 pop ebp
```

Arithmetic and Logic

- *add op1, op2*
- adds together the two operands and stores result in the first operand
- *Flags*
  - *o..szapc*
- *Syntax*
  add <reg>,<reg>
  add <reg>,<mem>
  add <mem>,<reg>
  add <reg>,<con>
  add <mem>,<con>
  Examples
  `add eax, 10` — add 10 to the current value in eax, and store result in eax

Arithmetic and Logic

- *sub op1, op2*
- subtracts the two operands and stores result in the first operand
- *Flags*
  - *o..szapc*
- *Syntax*
  sub <reg>,<reg>
  sub <reg>,<mem>
  sub <mem>,<reg>
  sub <reg>,<con>
  sub <mem>,<con>
- *Examples*
  `sub al, ah` — AL ← AL - AH
  `sub eax, 216` — subtract 216 from the value stored in EAX

Arithmetic and Logic

- *inc op1*
- increments contents of operand1 by 1
- *Flags*
  - *o..szap.*
- *Syntax*
  inc <reg>
  inc <mem>
- *Examples*
  `inc eax` - adds one to the contents of eax
  `inc DWORD PTR [var]` – add one to the 32-bit integer stored at location *var*

Arithmetic and Logic

- *dec op1*
- decrements contents of operand1 by 1
- *Flags*
  - o..szap.
- *Syntax*
  dec <reg>
  dec <mem>
- *Examples*
  `dec eax` – subtracts one from the contents of eax
  `dec DWORD PTR [var]` – subtracts one from the 32-bit integer stored at location *var*

Arithmetic and Logic

- *imul*
  - 2 operand – multiplies op1 and op2 together and stores result in op1. op1 must be a register
  - 3 operand – multiplies op2 and op3 together and stores results in op1. op1 must be a register, op3 must be a constant
- *Flags*
  - o..szap.
- *Syntax*
  imul <reg32>,<reg32>
  imul <reg32>,<mem>
  imul <reg32>,<reg32>,<con>
  imul <reg32>,<mem>,<con>
- *Examples*
  `imul eax, [var]` - multiply the contents of EAX by the 32-bit contents of the memory location *var*. Store the result in EAX.

  `imul esi, edi, 25` - ESI = EDI * 25

Arithmetic and Logic

- *idiv*
- Divides the contents of the 64 bit integer EDX:EAX by op1. Quotient stored in EAX, remained in EDX
- *Flags*
  - o..szapc
- *Syntax*
  idiv <reg32>
  idiv <mem>
- *Examples*
  `idiv ebx` – divide the contents of EDX:EAX by the contents of EBX.
- `idiv DWORD PTR [var]` - divide the contents of EDX:EAX by the 32-bit value stored at memory location *var*.

Arithmetic and Logic

- *and op1, op2*
- bitwise and, save results in op1
- *Flags*
  - *o..szapc*
- *Syntax*
  and <reg>,<reg>
  and <reg>,<mem>
  and <mem>,<reg>
  and <reg>,<con>
  and <mem>,<con>
- *Examples*
  `and eax, 0fH` — clear all but the last 4 bits of EAX

Arithmetic and Logic

- *or op1, op2*
- bitwise or, save results in op1
- *Flags*
  - *o..szapc*
- *Syntax*
  or <reg>,<reg>
  or <reg>,<mem>
  or <mem>,<reg>
  or <reg>,<con>
  or <mem>,<con>
- *Examples*
  `or eax, 0fH` — set the last 4 bits of EAX

Arithmetic and Logic

- *xor op1, op2*
- bitwise xor, save results in op1
- *Flags*
  - o..szapc
- *Syntax*
  xor <reg>,<reg>
  xor <reg>,<mem>
  xor <mem>,<reg>
  xor <reg>,<con>
  xor <mem>,<con>
- *Examples*
  `xor eax, eax` – set eax to 0

Arithmetic and Logic

- *not op1*
- bitwise not of op1, save results in op1
- *Syntax*
  not <reg>
  not <mem>
- *Examples*
  `not BYTE PTR [var]` — negate all bits in the byte at the memory location *var*.

Arithmetic and Logic

- *neg op1*
- twos complement on op1, save results in op1
- *Flags*
  - o..szapc
- *Syntax*

  neg <reg>

  neg <mem>
- *Examples*

  `neg eax` — EAX → - EAX

Arithmetic and Logic

- *shl op1, op2*
- logical shift left op1, op2 times
- *Flags*
  - o..szapc
- *Syntax*
  shl <reg>,<con8>
  shl <mem>,<con8>
  shl <reg>,<cl>
  shl <mem>,<cl>
- *Examples*
  `shl eax, 1` — Multiply the value of EAX by 2

Arithmetic and Logic

- *sal op1, op2*
- arithmetic shift left op1, op2 times
- *Flags*
  - o..szapc
- *Syntax*
shl <reg>,<con8>
shl <mem>,<con8>
shl <reg>,<cl>
shl <mem>,<cl>
- *Examples*
`sal eax, 1` — shift the value of EAX by 1

Arithmetic and Logic

- *shr op1, op2*
- logical shift right op1, op2 times
- *Flags*
  - o..szapc
- *Syntax*
  shr <reg>,<con8>
  shr <mem>,<con8>
  shr <reg>,<cl>
  shr <mem>,<cl>
- *Examples*
  `shr eax, 2` — Divide the value of EAX by 4 bits

Arithmetic and Logic

- *sar op1, op2*
- arithmetic shift right op1, op2 times
- *Flags*
  - o..szapc
- *Syntax*
  sar<reg>,<con8>
  sar<mem>,<con8>
  sar<reg>,<cl>
  sar<mem>,<cl>
- *Examples*
  `sar eax, 1` — shift eax right 1, duplicating the sign bit with each shift

Arithmetic and Logic

- *test op1, op2*
- logical and of op1 and op2, result is discarded
- *Flags*
  - *o..szapc*
- *Syntax*
  test <reg>,<reg>
  test <con>,<reg>
  test <reg>,<mem>
  test <con>,<mem>
- *Examples*
  `test ax, 5 – sets ZF, PF, and SF to appropriate state based on value in ax`

Arithmetic and Logic

- *cmp op1, op2*
- subtracts op2 from op1, result is discarded
- *Flags*
  - o..szapc
- *Syntax*
  cmp <reg>,<reg>
  cmp <reg>,<con>
  cmp <reg>,<mem>
  cmp <mem>,<mem>
- cmp <mem>,<reg>
- cmp <mem>,<con>
- *Examples*
  `cmp ax, 5 – sets ZF, OF, PF, and SF to appropriate state based on value in ax`

Examples

- Rewrite the following C code in assembly:
  - int i = 7; char j = 5; int k = i + j;
- Assume:
  - i is at address 100
  - j is at address 200
  - k is at address 300

```
int i = 7; char j = 5; int k = i + j;
```

```
mov dword [ 100 ], 7  ; set i
mov byte [ 200 ], 5   ; set j

mov eax, [ 100 ]      ; load i into eax
xor ebx, ebx          ; zero ebx
mov bl, [ 200 ]       ; load j into ebx

add eax, ebx          ; add ebx to eax, store in eax

mov [ 300 ], eax      ; save result to k
```

Examples

- Rewrite the following C code in assembly:
  – int i = 7; char j = 5; int k = i * i + j * j;
- Assume:
  – i is at address 100
  – j is at address 200
  – k is at address 300

```
int i = 7; char j = 5; int k = i * i + j * j;
```

```
mov dword [ 100 ], 7   ; set i
mov byte [ 200 ], 5    ; set j

mov ecx, [ 100 ]       ; load i into ecx
xor ebx, ebx           ; zero ebx
mov bl, [ 200 ]        ; load j into ebx

mov eax, ecx           ; copy ecx into eax (eax = ecx = i)
mul ecx                ; multiply ecx by eax, store result in eax
mov ecx, eax           ; save result back to ecx to free up eax

mov eax, ebx           ; copy ebx into eax (eax = ebx = j)
mul ebx                ; multiply ebx by eax, store result in eax

add eax, ecx           ; add ecx to eax, store result in eax
mov [ 300 ], eax       ; save final value to k
```

Examples

- None of the following lines will assemble
  - Determine why

```
mov [ bl ], 0xf              ; cannot address with bl
mov [0xabcd], 1337           ; ambiguous memory size
mov word [0xabcd], eax       ; incorrect memory size
mov byte [1], byte [2]       ; memory to memory
mov sl, al                   ; no sl register
mov 0x1234, eax              ; immediate destination
mov eax, dx                  ; size mismatch
```

Writing your own

- name your code file `.asm`
- in Linux

//assemble the code into an object file

```
nasm -f elf myasm.asm
```

//link the object file

```
ld -melf_i386 myasm.o -o
                    myasm.out
```

//running the output

```
./myasm.out
```

Writing your own

- assemble the code into an object file

```
nasm -f elf myasm.asm
```

- This uses the nasm assembler to translate the assembly code into machine code, with additional information that can later be used to create an executable file

Writing your own

- link the object file

```
ld -melf_i386 myasm.o -o
                     myasm.out
```

  - This creates an executable file called “myasm.out” in your directory.
  - -melf_i386 tells ld to link for a x86 elf

Writing your own

- Disassembling
  - `objdump -D –Mintel myasm.out`
  - `dumps code section & data section`
  - `-Mintel tells it to use intel syntax`

Assembly Makefile

```
all: myasm.o

myasm.o: myasm.asm
	nasm -f elf myasm.asm
	ld -melf_i386 myasm.o


clean:
	rm myasm.o a.out
```

Makefile

```
all: printreg-shift.o

printreg-shift.o: printreg-shift.asm
	nasm -f elf32 -g printreg-shift.asm
	ld -melf_i386 -g printreg-shift.o -o printreg-
shift.out

clean:
	rm printreg-shift.o printreg-shift.out
```

Add debugging symbols

- Once we get into more complicated assembly you wont be able to do much debugging if you don’t include extra debug symbols

NASM sections

```
section       .text;Section for all
code
global     _start ;Exports start
method
_start:    ;Linker entry point. ld by
              default looks for _start


........ CODE HERE


section .data ;Section for global
data
```

Declaring Variables

- General form: `Name <granularity> <initial value>`
- db = 1 byte
- dw = 2 bytes
- dd = 4 bytes
- dq = 8 bytes

```
section      .data
v1 db    0x55               ; just the byte 0x55
v2 db    0x55,0x56,0x57     ; three bytes in succession
v3 db    'a',0x55           ; character constants are
OK
v4 db    'hello',13,10,'$'  ; so are string constants
v5 dw    0x1234             ; 0x34 0x12
v6 dw    'a'              ; 0x61 0x00
v7 dw    'ab'             ; 0x61 0x62
```

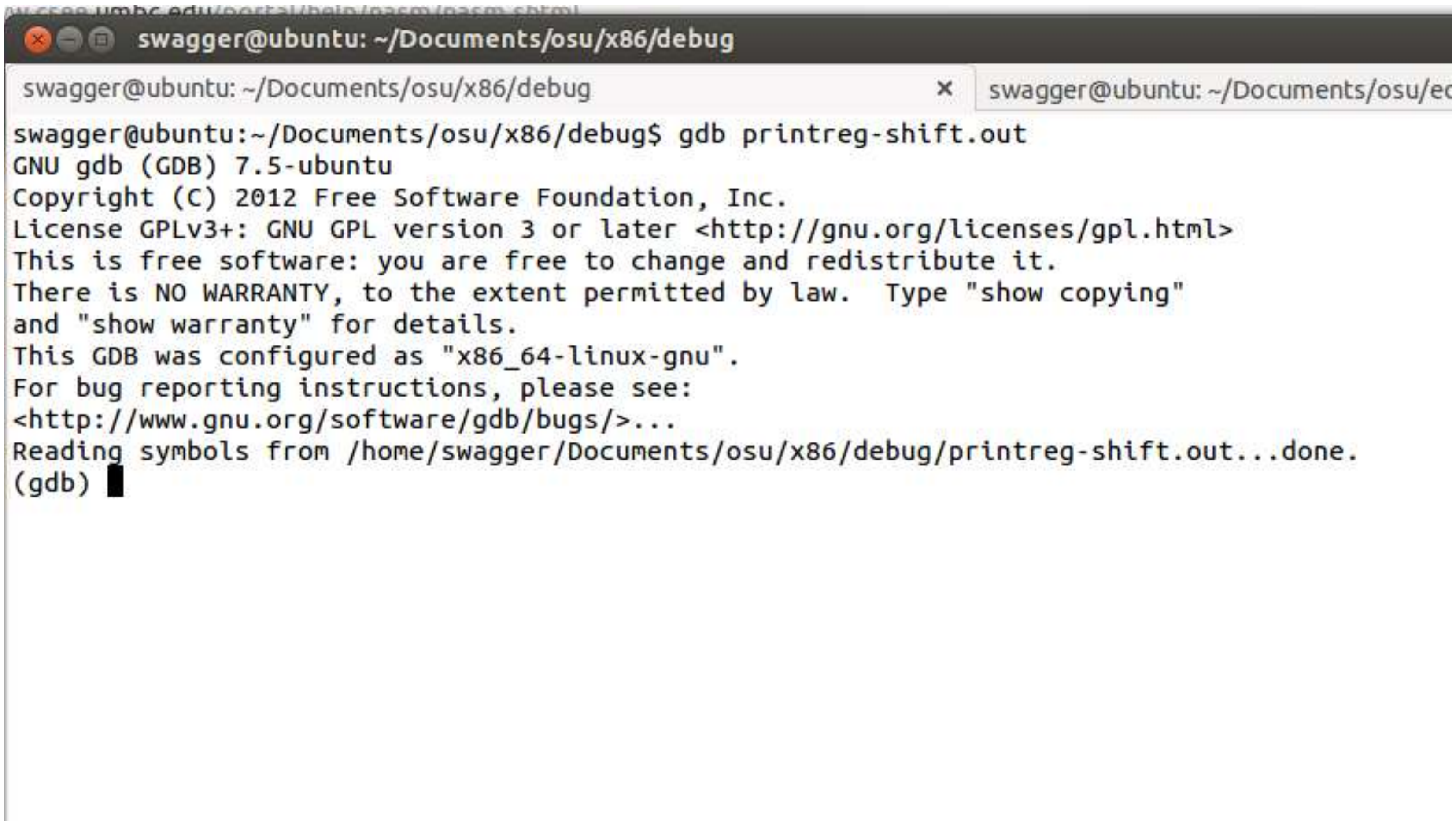
Declaring Variables

- General form: Name <granularity> <initial value>
- db = 1 byte
- dw = 2 bytes
- dd = 4 bytes
- dq = 8 bytes

```
section      .data
v8 dw    'abc'              ; 0x61 0x62 0x63 0x00 (string)
v9 dd    0x12345678            ; 0x78 0x56 0x34 0x12
v10 dd    1.234567e20          ; floating-point constant
v11 dq    0x123456789abcdef0  ; eight byte constant
v12 dq    1.234567e20          ; double-precision float
v13 dt    1.234567e20          ; extended-precision float
```

Debugging x86 with GDB

- Run “gdb <executable_name>”




```
swagger@ubuntu:~/Documents/osu/x86/debug$ gdb printreg-shift.out
GNU gdb (GDB) 7.5-ubuntu
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/swagger/Documents/osu/x86/debug/printreg-shift.out...done.
(gdb) 
```

Debugging x86 with GDB

- Tell gdb we want to look at intel assembly
- `(gbd) set disassembly-flavor intel`

swagger@ubuntu: ~/Documents/osu/x86/debug

swagger@ubuntu: ~/Documents/osu/x86/debug | swagger@ubuntu: ~/Documents/osu/ec

```
swagger@ubuntu:~/Documents/osu/x86/debug$ gdb printreg-shift.out
GNU gdb (GDB) 7.5-ubuntu
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/swagger/Documents/osu/x86/debug/printreg-shift.out...done.
(gdb) set disassembly-flavor intel
(gdb)
(gdb)
(gdb) 
```

Debugging x86 with GDB

- Show the different parts of the file
- `(gdb) info files`

```
(gdb) info files
Symbols from "/home/swagger/Documents/osu/x86/a.out".
Unix child process:
        Using the running image of child process 61165.
        While running this, GDB does not access memory from...
Local exec file:
        `/home/swagger/Documents/osu/x86/a.out', file type elf32-i386.
        Entry point: 0x80480d1
        0x08048080 - 0x080480dd is .text
        0x080490e0 - 0x080490e5 is .data
(gdb) █
```

Debugging x86 with GDB

- disassemble
  - disassembles where IP currently is
- disassemble address
  - `disassemble 0x8048080`
- disassemble label
  - `disassemble loop`
  - `disassemble main`

- add '+ number' to print number instructions
  - `disassemble main +50`

```
loop:
    mov edx, eax    ;copy the value into edx for us to do manipulations on
    mov ecx, ebx
    shl ecx, 2  ;multiply by 4
    shr edx, cl
    and edx, 0xf    ;get rid of all but the bottom nibble

    cmp dl, 10      ;check if the remainder is less than 10
    jge _ascii_to_hex   ;if it was greater or equal to 10 then we know its A-F
    add dl, '0'     ; its a numeric digit, add '0' to conver to ascii
    jmp _ascii_to_end
_ascii_to_hex:
    add dl,'7'      ;its A-F, add 0x55 which how to convert to a letter
_ascii_to_end:
    dec ebx
    mov [byteToPrint], dl; store the result into memory
    ;save our values
    push eax
    push ebx
    ;print it
    mov eax, 4                  ; system call #4 = sys_write
    mov ebx, 1                  ; file descriptor 1 = stdout
                                                        37,1-4
```

```
(gdb) disassemble loop
Dump of assembler code for function loop:
   0x080483f1 <+0>:     mov    edx,eax
   0x080483f3 <+2>:     mov    ecx,ebx
   0x080483f5 <+4>:     shl    ecx,0x2
   0x080483f8 <+7>:     shr    edx,cl
   0x080483fa <+9>:     and    edx,0xf
   0x080483fd <+12>:    cmp    dl,0xa
   0x08048400 <+15>:    jge    0x8048407 <_ascii_to_hex>
   0x08048402 <+17>:    add    dl,0x30
   0x08048405 <+20>:    jmp    0x804840a <_ascii_to_end>
```

github - dazzlecatduo

Debugging x86 with GDB

- break address
- break label

```
(gdb) disassemble loop
Dump of assembler code for function loop:
   0x080483f1 <+0>:     mov    edx,eax
   0x080483f3 <+2>:     mov    ecx,ebx
   0x080483f5 <+4>:     shl    ecx,0x2
   0x080483f8 <+7>:     shr    edx,cl
   0x080483fa <+9>:     and    edx,0xf
   0x080483fd <+12>:    cmp    dl,0xa
   0x08048400 <+15>:    jge    0x8048407 <_ascii_to_hex>
   0x08048402 <+17>:    add    dl,0x30
   0x08048405 <+20>:    jmp    0x804840a <_ascii_to_end>
End of assembler dump.
(gdb) break loop
Breakpoint 1 at 0x80483f1: file printreg-shift.asm, line 21.
(gdb)
```

Debugging x86 with GDB

- `(gdb) info register`
  - Show the current values in the x86 registers

```
Starting program: /home/swagger/Documents/osu/x86/debug/printreg-shift.out

Breakpoint 1, loop () at printreg-shift.asm:21
21              mov edx, eax     ;copy the value into edx for us to do manipulations on
(gdb) info register
eax            0xabcdef12       -1412567278
ecx            0xffffd304       -11516
edx            0xffffd294       -11628
ebx            0x7      7
esp            0xffffd268       0xffffd268
ebp            0x0      0x0
esi            0x0      0
edi            0x0      0
eip            0x80483f1        0x80483f1 <loop>
eflags         0x246    [ PF ZF IF ]
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
fs             0x0      0
gs             0x63     99
(gdb) 
```

Flags currently set

Decimal Value

Hex Value

Debugging x86 with GDB

- You can print individual registers
  - *print $reg*

```
(gdb) print $esp
$1 = (void *) 0xffffd260
(gdb) █
```

Debugging x86 with GDB

- Step 1 instruction at a time

```
(gdb) stepi
34              dec ebx
(gdb) stepi
35              mov [byteToPrint], dl; store the result into memory
(gdb) stepi
37              push eax
(gdb) stepi
38              push ebx
(gdb) stepi
40              mov eax, 4                      ; system call #4 = sys_write
(gdb) stepi
41              mov ebx, 1                      ; file descriptor 1 = stdout
(gdb) █
```

Notice how we have our comments? Because we did a debug build those are left in

Debugging x86 with GDB

- If we would have wanted to step OVER a “call” (just like stepping over a function call in C when debugging), we would have used “nexti” instead
- “stepi” will step INTO any function if you call, “nexti” will step OVER it

Debugging x86 with GDB

- See all defined variables in the application
  - *(gdb) info variables*

```
(gdb) info variables
All defined variables:

Non-debugging symbols:
0x080490e0  loop_index
0x080490e4  byteToPrint
0x080490e5  __bss_start
0x080490e5  _edata
0x080490e8  _end
(gdb) 
```

Debugging x86 with GDB

- command x (for "examine") to examine memory
- Format: `x/nfu addr`
- n, f, and u are all optional parameters
  - n - repeat count. How much memory (counting by units u) to display.
  - f - display format – what format to print
    - `s' (null-terminated string),
    - `i' (machine instruction).
    - `x' (hexadecimal) - *DEFAULT*.
- u - the unit size
    - b - Bytes.
    - h - Halfwords (two bytes)
    - w - Words (four bytes) - *DEFAULT*
    - g - Giant words (eight bytes)

Debugging x86 with GDB

- From info variables we know the address of 'byteToPrint'

```
(gdb) info variables
All defined variables:

Non-debugging symbols:
0x080490e0  loop_index
0x080490e4  byteToPrint
0x080490e5  __bss_start
0x080490e5  _edata
0x080490e8  _end
(gdb)
```

- Lets dump memory there
  - print 10 bytes in hex format

```
(gdb) x/16x 0x80490e4
0x80490e4 <byteToPrint>:        0x00000041      0x00000001      0x00210000
0x00000014
0x80490f4:      0x00000001      0x00000064      0x08048080      0x00000000
0x8049104:      0x000a0044      0x08048080      0x00000000      0x000b0044
0x8049114:      0x08048085      0x00000000      0x000c0044      0x0804808a
(gdb)
```

  - Looks like there is an 'A' (0x41) sitting there

Debugging x86 with GDB

- Can use register values with the ‘x’ command
  - ex: dump the stack

```
(gdb) x/10x $esp
0xffffd260:     0x00000006      0xabcdef12      0x0804843b      0xf7e324d3
0xffffd270:     0x00000001      0xffffd304      0xffffd30c      0xf7fda858
0xffffd280:     0x00000000      0xffffd31c
(gdb)
```

  - Dump 10 bytes in hex format at the address in esp

Debugging x86 with GDB

- When wanting to debug your C code you could compile using the debug flag in GCC
  - gcc –g myfile.c
- What if you want to debug something that you cant rebuild with debug symbols?
  - You can debug in assembly!

Debugging with GDB

- keychecker.out
  - written in C, built without debug flag (-g)

```
swagger@ubuntu:~/Documents/osu/ec$ gdb keychecker.out
GNU gdb (GDB) 7.5-ubuntu
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/swagger/Documents/osu/ec/keychecker.out...(no debugging symbols found)...done.
```

Debugging with GDB

```
Entry point: 0x80483a0
0x08048154 - 0x08048167 is .interp
0x08048168 - 0x08048188 is .note.ABI-tag
0x08048188 - 0x080481ac is .note.gnu.build-id
0x080481ac - 0x080481cc is .gnu.hash
0x080481cc - 0x0804823c is .dynsym
0x0804823c - 0x080482a6 is .dynstr
0x080482a6 - 0x080482b4 is .gnu.version
0x080482b4 - 0x080482e4 is .gnu.version_r
0x080482e4 - 0x080482ec is .rel.dyn
0x080482ec - 0x08048314 is .rel.plt
0x08048314 - 0x08048338 is .init
0x08048340 - 0x080483a0 is .plt
0x080483a0 - 0x08048648 is .text
0x08048648 - 0x0804865d is .fini
0x08048660 - 0x080486a9 is .rodata
0x080486ac - 0x080486f0 is .eh_frame_hdr
0x080486f0 - 0x080487f4 is .eh_frame
0x08049f08 - 0x08049f0c is .init_array
0x08049f0c - 0x08049f10 is .fini_array
0x08049f10 - 0x08049f14 is .jcr
0x08049f14 - 0x08049ffc is .dynamic
0x08049ffc - 0x0804a000 is .got
0x0804a000 - 0x0804a020 is .got.plt
0x0804a020 - 0x0804a028 is .data
0x0804a028 - 0x0804a02c is .bss
0xf7fdc114 - 0xf7fdc138 is .note.gnu.build-id in /lib/ld-linux.so.2
0xf7fdc138 - 0xf7fdc1f4 is .hash in /lib/ld-linux.so.2
0xf7fdc1f4 - 0xf7fdc2d4 is .gnu.hash in /lib/ld-linux.so.2
0xf7fdc2d4 - 0xf7fdc494 is .dynsym in /lib/ld-linux.so.2
0xf7fdc494 - 0xf7fdc612 is .dynstr in /lib/ld-linux.so.2
0xf7fdc612 - 0xf7fdc64a is .gnu.version in /lib/ld-linux.so.2
0xf7fdc64c - 0xf7fdc714 is .gnu.version_d in /lib/ld-linux.so.2
0xf7fdc714 - 0xf7fdc77c is .rel.dyn in /lib/ld-linux.so.2
0xf7fdc77c - 0xf7fdc7ac is .rel.plt in /lib/ld-linux.so.2
0xf7fdc7b0 - 0xf7fdc820 is .plt in /lib/ld-linux.so.2
```

```
(gdb) info files
```

Applications built in C have much more overhead and compiler generated sections than programming in straight assembly

Debugging with GDB

```
0xf7ffcff4 - 0xf7ffd000 is .got in /lib/ld-linux.so.2
0xf7ffd000 - 0xf7ffd024 is .got.plt in /lib/ld-linux.so.2
0xf7ffd040 - 0xf7ffd874 is .data in /lib/ld-linux.so.2
0xf7ffd878 - 0xf7ffd938 is .bss in /lib/ld-linux.so.2
0xf7e19174 - 0xf7e19198 is .note.gnu.build-id in /lib/i386-linux-gnu/libc.so.6
0xf7e19198 - 0xf7e191b8 is .note.ABI-tag in /lib/i386-linux-gnu/libc.so.6
0xf7e191b8 - 0xf7e1ce38 is .gnu.hash in /lib/i386-linux-gnu/libc.so.6
0xf7e1ce38 - 0xf7e26158 is .dynsym in /lib/i386-linux-gnu/libc.so.6
0xf7e26158 - 0xf7e2bcef is .dynstr in /lib/i386-linux-gnu/libc.so.6
0xf7e2bcf0 - 0xf7e2cf54 is .gnu.version in /lib/i386-linux-gnu/libc.so.6
0xf7e2cf54 - 0xf7e2d374 is .gnu.version_d in /lib/i386-linux-gnu/libc.so.6
0xf7e2d374 - 0xf7e2d3b4 is .gnu.version_r in /lib/i386-linux-gnu/libc.so.6
0xf7e2d3b4 - 0xf7e2fdec is .rel.dyn in /lib/i386-linux-gnu/libc.so.6
0xf7e2fdec - 0xf7e2fe44 is .rel.plt in /lib/i386-linux-gnu/libc.so.6
0xf7e2fe50 - 0xf7e2ff10 is .plt in /lib/i386-linux-gnu/libc.so.6
0xf7e2ff10 - 0xf7f6469c is .text in /lib/i386-linux-gnu/libc.so.6
0xf7f646a0 - 0xf7f65613 is __libc_freeres_fn in /lib/i386-linux-gnu/libc.so.6
0xf7f65620 - 0xf7f65846 is __libc_thread_freeres_fn in /lib/i386-linux-gnu/libc.so.6
0xf7f65860 - 0xf7f843c8 is .rodata in /lib/i386-linux-gnu/libc.so.6
0xf7f843c8 - 0xf7f843db is .interp in /lib/i386-linux-gnu/libc.so.6
0xf7f843dc - 0xf7f8bac8 is .eh_frame_hdr in /lib/i386-linux-gnu/libc.so.6
0xf7f8bac8 - 0xf7fb843c is .eh_frame in /lib/i386-linux-gnu/libc.so.6
0xf7fb843c - 0xf7fb8964 is .gcc_except_table in /lib/i386-linux-gnu/libc.so.6
0xf7fb8964 - 0xf7fbbe30 is .hash in /lib/i386-linux-gnu/libc.so.6
0xf7fbd1ac - 0xf7fbd1b4 is .tdata in /lib/i386-linux-gnu/libc.so.6
0xf7fbd1b4 - 0xf7fbd1ec is .tbss in /lib/i386-linux-gnu/libc.so.6
0xf7fbd1b4 - 0xf7fbd1c0 is .init_array in /lib/i386-linux-gnu/libc.so.6
0xf7fbd1c0 - 0xf7fbd234 is __libc_subfreeres in /lib/i386-linux-gnu/libc.so.6
0xf7fbd234 - 0xf7fbd238 is __libc_atexit in /lib/i386-linux-gnu/libc.so.6
0xf7fbd238 - 0xf7fbd248 is __libc_thread_subfreeres in /lib/i386-linux-gnu/libc.so.6
```

Debugging with GDB

- We cant view c code

```
(gdb) start
Temporary breakpoint 1 at 0x804848f
Starting program: /home/swagger/Documents/osu/ec/keychecker.out

Temporary breakpoint 1, 0x0804848f in main ()
(gdb) l
No symbol table is loaded.  Use the "file" command.
(gdb) 
```

- There are no symbols!

Debugging with x86

- We can ALWAYS view the disassembly

```
0x08048340 - 0x080483a0 is .plt
0x080483a0 - 0x08048648 is .text
0x08048648 - 0x0804865d is .fini
```

```
(gdb) set disassembly-flavor intel
(gdb) disassemble 0x80483a0
Dump of assembler code for function _start:
   0x080483a0 <+0>:     xor    ebp,ebp
   0x080483a2 <+2>:     pop    esi
   0x080483a3 <+3>:     mov    ecx,esp
   0x080483a5 <+5>:     and    esp,0xfffffff0
   0x080483a8 <+8>:     push   eax
   0x080483a9 <+9>:     push   esp
   0x080483aa <+10>:    push   edx
   0x080483ab <+11>:    push   0x8048640
   0x080483b0 <+16>:    push   0x80485d0
   0x080483b5 <+21>:    push   ecx
   0x080483b6 <+22>:    push   esi
   0x080483b7 <+23>:    push   0x804848c
   0x080483bc <+28>:    call   0x8048380 <__libc_start_main@plt>
   0x080483c1 <+33>:    hlt
   0x080483c2 <+34>:    xchg   ax,ax
   0x080483c4 <+36>:    xchg   ax,ax
   0x080483c6 <+38>:    xchg   ax,ax
   0x080483c8 <+40>:    xchg   ax,ax
   0x080483ca <+42>:    xchg   ax,ax
   0x080483cc <+44>:    xchg   ax,ax
   0x080483ce <+46>:    xchg   ax,ax
End of assembler dump.
(gdb)
```

Debugging with x86

- With c code there are things that happen before your ‘main’ is called.
- The application is setting up the environment for you
- Locate the call to __libc_start_main
- The value pushed on the stack just before that call is the address of OUR main() function in memory

Debugging with x86

- Locate the push right before libc_start
  - Our code starts at 0x804848c

```
End of assembler dump.
(gdb) set disassembly-flavor intel
(gdb) disassemble 0x80483a0
Dump of assembler code for function _start:
   0x080483a0 <+0>:     xor    ebp,ebp
   0x080483a2 <+2>:     pop    esi
   0x080483a3 <+3>:     mov    ecx,esp
   0x080483a5 <+5>:     and    esp,0xfffffff0
   0x080483a8 <+8>:     push   eax
   0x080483a9 <+9>:     push   esp
   0x080483aa <+10>:    push   edx
   0x080483ab <+11>:    push   0x8048640
   0x080483b0 <+16>:    push   0x80485d0
   0x080483b5 <+21>:    push   ecx
   0x080483b6 <+22>:    push   esi
   0x080483b7 <+23>:    push   0x804848c
   0x080483bc <+28>:    call   0x8048380 <__libc_start_main@plt>
   0x080483c1 <+33>:    hlt
   0x080483c2 <+34>:    xchg   ax,ax
   0x080483c4 <+36>:    xchg   ax,ax
   0x080483c6 <+38>:    xchg   ax,ax
   0x080483c8 <+40>:    xchg   ax,ax
   0x080483ca <+42>:    xchg   ax,ax
   0x080483cc <+44>:    xchg   ax,ax
   0x080483ce <+46>:    xchg   ax,ax
End of assembler dump.
(gdb)
```

Debugging with x86

- Now we can disassemble the main

```
shadowfax@ubuntu: ~/Documents/osu/SP2014/x86
End of assembler dump.
(gdb) disassemble 0x804848c
Dump of assembler code for function main:
   0x0804848c <+0>:     push   ebp
   0x0804848d <+1>:     mov    ebp,esp
   0x0804848f <+3>:     and    esp,0xfffffff0
   0x08048492 <+6>:     sub    esp,0x20
   0x08048495 <+9>:     mov    DWORD PTR [esp],0x8048668
   0x0804849c <+16>:    call   0x8048350 <printf@plt>
   0x080484a1 <+21>:    lea    eax,[esp+0x1c]
   0x080484a5 <+25>:    mov    DWORD PTR [esp+0x4],eax
   0x080484a9 <+29>:    mov    DWORD PTR [esp],0x804868d
   0x080484b0 <+36>:    call   0x8048390 <__isoc99_scanf@plt>
   0x080484b5 <+41>:    mov    eax,DWORD PTR [esp+0x1c]
   0x080484b9 <+45>:    mov    DWORD PTR [esp],eax
   0x080484bc <+48>:    call   0x80484eb <is_valid>
   0x080484c1 <+53>:    test   eax,eax
   0x080484c3 <+55>:    je     0x80484d8 <main+76>
   0x080484c5 <+57>:    mov    DWORD PTR [esp],0x8048690
   0x080484cc <+64>:    call   0x8048360 <puts@plt>
   0x080484d1 <+69>:    mov    eax,0x1
   0x080484d6 <+74>:    jmp    0x80484e9 <main+93>
   0x080484d8 <+76>:    mov    DWORD PTR [esp],0x804869a
   0x080484df <+83>:    call   0x8048360 <puts@plt>
   0x080484e4 <+88>:    mov    eax,0x0
   0x080484e9 <+93>:    leave
   0x080484ea <+94>:    ret
End of assembler dump.
(gdb)
```

libc calls

GDB Summary Slide

Using GDB

- Basic commands are usually sufficient
  - Starting and stopping
    - quit, run, kill
  - Breakpoints
    - break, delete
  - Execution
    - stepi, nexti, continue, finish
  - Examining code and data
    - disas, print, x
  - Useful information
    - info, help
  - Listing source code line numbers
    - list

Lab

SharkSim 3000

- In the linux VM (dazzlecat/dazzleme)
- cd ~/training/sharksim
- READ sharksim3000.pdf
  - you've spent thousands of hours crafting the most terrifying, realistic shark simulation the world has ever seen – all in x86 assembly!
  - There's just one problem –the program is seg faulting!!!
  - You'll need to debug the code to save your reputation and unleash the greatest game of all time – Shark Sim 3000!

Control Flow Instructions

- ip register (instruction pointer)
  - Holds the address of the current instruction
- ip register cannot be manipulated directly
- Updated by control flow instructions

- In x86 we use labels to denote locations in program text.
  - label name followed by a colon

```
        mov esi, [ebp+8]
begin:  xor, ecx, ecx
        mov eax, [esi]
```

Control Flow Instructions

- *jmp* op1
- Jump
- Transfers program control flow to the instruction at the memory location indicated by the op1
- *Syntax*
  jmp <label>
- *Example*
  `jmp begin` — Jump to the instruction labeled begin

Jump

- Using the JMP instruction, we can create a infinite loop that counts up from zero using the eax register:

```
       mov eax, 0
loop:  inc eax
       jmp loop
```

Conditional Jumps

- Conditional jumps take into consideration the current state of the flags to determine if a jump is taken or not

Control Flow Instructions

- Jumps
- *Syntax*
  je <label> (jump when equal)
  jne <label> (jump when not equal)
  jz <label> (jump when last result was 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)
- *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 the next instruction.

je

- jge - jump when greater than or equal to
  - Conditions: SF = 0 || ZF = 1
- jl - jump when less than
  - Conditions: SF = 1
- jle - jump when less than or equal to
  - Conditions: SF = 1 || ZF = 1

je

- je - jump equals
  - Conditions: ZF = 1
- jne - jump when not equal
  - Conditions: ZF = 0
- jz - jump when last result was zero
  - Conditions: ZF = 1
- jg - jump when greater than
  - Conditions: SF = 0 && ZF = 0

Arithmetic and Logic

- *test*
- Bitwise AND of op1 and op2, result is discarded
- *Flags*
  - *o..szapc*
- *Syntax*
  test <reg>,<reg>
  test <con>,<reg>
  test <reg>,<mem>
  test <con>,<mem>
- *Examples*
  `test ax, 5 – Check if bits 0 and 2 are set`

test example

- Why is this useful?
- Test can be used to see if a particular bit is set by looking at ZF

```
mov ax, 0x1450
test ax, 0x01 ;check bit 1 is set

0001 0100 0101 0000 &
0000 0000 0000 0001
-------------------
0000 0000 0000 0000

ZF = 1; Means the bit was not set!
```

test example

- Why is this useful?
- Test can be used to see if a particular bit is set by looking at ZF

```
mov ax, 0x1451
test ax, 0x01 ;check bit 1 is set

0001 0100 0101 0001 &
0000 0000 0000 0001
-------------------
0000 0000 0000 0001

ZF = 0; Means the bit was set!
```

Arithmetic and Logic

- *cmp*
- subtracts op2 from op1, result is discarded
- *Flags*
  - o..szapc
- *Syntax*
  cmp <reg>,<reg>
  cmp <reg>,<con>
  cmp <reg>,<mem>
  cmp <mem>,<mem>
- cmp <mem>,<reg>
- cmp <mem>,<con>
- *Examples*

```
cmp ax, 5 – sets ZF, OF, PF, and SF to
appropriate state based on value in ax
```

Compare Example

- Why is this useful?
- Check if value is equal, greater, or less than another value

```
mov eax, 0x100
mov ebx, 0x200
cmp eax, ebx; does eax-ebx

eax is less than ebx
SF = 1, ZF = 0
```

Compare Example

- Why is this useful?
- Check if value is equal, greater, or less than another value

```
mov eax, 0x300
mov ebx, 0x200
cmp eax, ebx; does eax-ebx

eax is greater than ebx
SF = 0, ZF = 0
```

Compare Example

- Why is this useful?
- Check if value is equal, greater, or less than another value

```
mov eax, 0x500
mov ebx, 0x500
cmp eax, ebx; does eax-ebx

eax is equal to ebx
SF = 0, ZF = 1
```

Compare

```
cmp eax, ebx
```

if	SF	ZF
eax > ebx	0	0
eax = ebx	0	1
eax < ebx	1	0

Conditional Jump

- Using conditional jumps we can implement a non infinite loop

```
Loop to count eax from 0 to 5:
        mov   eax, 0
loop:   inc eax
        cmp eax, 5
        jle loop
; conditional jump
;if eax <=  5 then go to loop
```

Example

```
;C if ( a == b ) x = 1;

      cmp ax, bx    ; (ax-bx) == 0
      jne skip ;not equal, so skip
      mov cx, 1;since a == b,x=1
skip:
      nop             ;no operation
```

Example

```
; C  if ( a > b ) x = 1;

     cmp a, b   ; (a-b) > 0
     jle skip   ; skip if a <= b
     mov x, 1
skip:
   … ;stuff
```

Example 4

```
int max(int x, int
y){
      if(x > y)
            return x;
      else
            return y;
}
```

```
int max(int x, int y) {
      int rval = y;
      int ok = (x <= y);
      if(ok)
            goto done;
      rval = x;
done:
      return rval;
```

a GOTO in c is generally considered bad coding practice, but its very close to machine level code

Example 4

```
int gotomax(int x, int y) {
        int rval = y;
        int ok = (x <= y);
        if(ok)
                goto done;
        rval = x;
done:
        return rval;
```

```
mov edx, [8+ebp]      ;edx = x
mov eax, [12+ebp]     ;eax = y
cmp  edx, eax         ; x :  y
jle    L9             ;goto L9
mov  eax, edx         ;eax = x
L9:
ret                   ;done
```

Data Movement

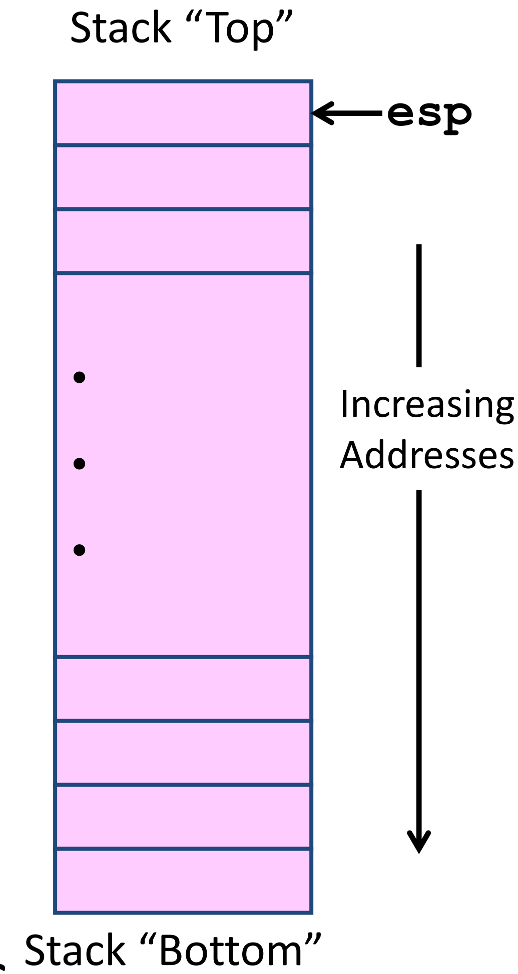
- *push*
- add 4 bytes on top of the stack
- *Syntax*
  push <reg32>
  push <mem>
  push <con32>
- *Examples*
  `push eax` — push eax on the stack

Data Movement

- *pop*
- remove top 4 byte value from stack
- *Syntax*
  pop <reg32>
  pop <mem>
- *Examples*
  `pop eax` — pop 4 bytes from stack into eax

Stack

- Stack is used to store data values contiguously in memory
- Stack is used for temporary storage
- Exists in RAM
- esp holds the address of the top of the stack
- stack grows towards lower addresses



Stack - Push

- Pushing
- Decrement the stack register by 4 then store new data

```
;(1)esp = 0x120
mov eax, 0xFECA8712
push eax
;(2)esp = 0x11C
```

(1)

addr	value
0x11B	
0x11C	
0x11D	
0x11E	
0x11F	
0x120	0x11
0x121	0x22
0x122	0x33

(2)

addr	value
0x11B	
0x11C	*0x12*
0x11D	*0x87*
0x11E	*0xCA*
0x11F	*0xFE*
0x120	0x11
0x121	0x22
0x122	0x33

Stack - Push

```
;(1)esp = 0x120
mov eax, 0xFECA8712
push eax
;(2)esp = 0x11C
```

```
;(1)esp = 0x120
mov eax, 0xFECA8712
sub esp, 4
mov [esp], eax
;(2)esp = 0x11C
```

(1)

addr	value
0x11B	
0x11C	
0x11D	
0x11E	
0x11F	
0x120	0x11
0x121	0x22
0x122	0x33

(2)

addr	value
0x11B	
0x11C	*0x12*
0x11D	*0x87*
0x11E	*0xCA*
0x11F	*0xFE*
0x120	0x11
0x121	0x22
0x122	0x33

Stack - Pop

- Pop
- Take data off the stack, increment the stack register by 4

;(1) esp = 0x11C
pop eax
;(2) esp = 0x120
;eax = 0xFECA8712

(1)

addr	value
0x11B	
0x11C	*0x12*
0x11D	*0x87*
0x11E	*0xCA*
0x11F	*0xFE*
0x120	0x11
0x121	0x22
0x122	0x33

(2)

addr	value
0x11B	
0x11C	
0x11D	
0x11E	
0x11F	
0x120	0x11
0x121	0x22
0x122	0x33

Stack - Pop

;(1) esp = 0x11C
pop eax
;(2) esp = 0x120
;eax = 0xFECA8712

;(1) esp = 0x11C
mov eax, [esp]
add esp, 4
;(2) esp = 0x120
;eax = 0xFECA8712

(1)

addr	value
0x11B	
0x11C	*0x12*
0x11D	*0x87*
0x11E	*0xCA*
0x11F	*0xFE*
0x120	0x11
0x121	0x22
0x122	0x33

(2)

addr	value
0x11B	
0x11C	
0x11D	
0x11E	
0x11F	
0x120	0x11
0x121	0x22
0x122	0x33

Stack

- Use the stack to hold values temporarily
- With limited registers we often need to free them up but don’t want to lose what we had
- Push it on the stack!

```
mov eax, 0x25
mov ebx 0x00C0FFEE
add eax, ebx
push eax;save to free up eax
...          ;do other things
pop eax      ; retrieve my saved eax
```

Stack

- When items are *popp*ed off the stack their value is not removed from memory
- The memory is considered deallocated and its value should not be relied on

Stack

esp	0x120

```
mov eax, 0x00C0FFEE
```

addr	value
0x11B	?
0x11C	?
0x11D	?
0x11E	?
0x11F	?
0x120	?
0x121	?
0x122	?
0x123	?

esp → 0x120

Stack

esp	0x11c

```
mov eax, 0x00C0FFEE
push eax
```

esp → 0x11C

addr	value
0x11B	?
0x11C	*0xEE*
0x11D	*0xFF*
0x11E	*0xC0*
0x11F	*0x00*
0x120	?
0x121	?
0x122	?
0x123	?

Stack

esp	0x120

```
mov eax, 0x00C0FFEE
push eax
pop ebx; ebx=0x00C0FFEE
```

addr	value
0x11B	?
0x11C	~~0xEE~~
0x11D	~~0xFF~~
0x11E	~~0xC0~~
0x11F	~~0x00~~
esp → 0x120	?
0x121	?
0x122	?
0x123	?

Deallocated from stack but still resident in memory

Stack

esp	0x11c

```
mov eax, 0x00C0FFEE
push eax
pop ebx; ebx=0x00C0FFEE
mov ecx, 0xDEADBEEF
push ecx
```

	addr	value
	0x11B	?
esp	0x11C	*0xEF*
	0x11D	*0xBE*
	0x11E	*0xAD*
	0x11F	*0xDE*
	0x120	?
	0x121	?
	0x122	?
	0x123	?

New values have now overwritten
the old ones

Control Flow Instructions

- *call*
- pushes the address of the next instruction onto the stack, then performs an unconditional jump to the code location in op1
- *Syntax*
  call <label>

Control Flow Instructions

- *ret*
- pops an address off of the stack and performs an unconditional jump to the address
- *Syntax*
  ret

Functions

- By combining the functionality of *call* and *ret* instructions we can implement functions in assembly

Calling

- Consider the following:

```
void func(){
   .........
}

void main(){
     ......
     func();
     ......
}
```

Calling

```
;C     func()
0x15423 main:   ← ip
           ......
0x15662 call _func
0x15666 mov eax, 10

0x16745 func:
           ......
0x16768 ret
```

Addr	Value
0xF0	
0xF4	?
0xF8	?
0xFC	?
esp → 0x100	?

esp	0x100
ip	0x15423

Calling

```
;C    func()
0x15423 main:
            ......
0x15662 call _func
0x15666 mov eax, 10

0x16745 func:
            ......
0x16768 ret
```

ip

esp

Addr	Value
0xF0	
0xF4	?
0xF8	?
0xFC	?
0x100	?

esp	0x100
ip	0x15662

Calling

```
;C    func()
0x15423 main:
        ......
0x15662 call _func
0x15666 mov eax, 10

0x16745 func:    <- ip
        ......
0x16768 ret
```

Addr	Value
0xF0	
0xF4	?
0xF8	?
esp → 0xFC	0x15666
0x100	?

esp	0xFC
ip	0x16745

Calling

```
;C    func()
0x15423 main:
            ......
0x15662 call _func
0x15666 mov eax, 10

0x16745 func:
            ......
0x16768 ret  <-- ip
```

Addr	Value
0xF0	
0xF4	?
0xF8	?
0xFC	0x15666
0x100	?

esp → 0xFC

esp	0xFC
ip	0x16768

Calling

```
;C    func()
0x15423 main:
        ......
0x15662 call _func
0x15666 mov eax, 10
```

ip

```
0x16745 func:
        ......
0x16768 ret
```

esp

Addr	Value
0xF0	
0xF4	?
0xF8	?
0xFC	~~0x15666~~
0x100	~~?~~

esp	0x100
ip	0x15666

Calling Convention

- Calling Convention – protocol for how to call and return from routines
- cdecl
  - caller clean-up
    - After call returns remove all pushed parameters from the stack
  - C declaration, used by c compilers
  - Arguments are passed on the stack
    - pushed from right to left
  - Return value is stored in EAX
  - EAX, ECX, EDX are caller-saved
  - Other registers are callee-saved

CDECL

```
void caller()
{
    //does the calling
    callee()
}


void callee()
{
    //was called
}
```

Calling Conventions

- Other calling conventions
- caller clean-up
  - cdecl
  - syscall
  - optlink
- Calle clean-up
  - pascal
  - register
  - stdcall
  - fastcall
  - safecall

CDECL

- Arguments are passed on the stack
  - pushed from right to left

```
void function(int a, int b, int c)
{
      ….
}
```

pushed first

pushed second

pushed third

Passing Parameters

```
;void function(int a, int b, int c)

_start:
    ;assume a is in eax
    ;assume b is in ebx
    ;assume c is in ecx
    push ecx        <- ip
    push ebx
    push eax
    call function
```

addr	value
0x64E8	?
0x64EC	?
0x64F0	?
0x64F4	?
0x64F8	?
0x64FC	?
0x6500	?

esp	0x6500

Passing Parameters

```
;void function(int a, int b, int c)

_start:
    ;assume a is in eax
    ;assume b is in ebx
    ;assume c is in ecx
    push ecx
    push ebx        <- ip
    push eax
    call function
```

addr	value
0x64E8	?
0x64EC	?
0x64F0	?
0x64F4	?
0x64F8	?
0x64FC	c
0x6500	?

esp	0x65FC

Passing Parameters

```
;void function(int a, int b, int c)

_start:
    ;assume a is in eax
    ;assume b is in ebx
    ;assume c is in ecx
    push ecx
    push ebx
    push eax        <- ip
    call function
```

addr	value
0x64E8	?
0x64EC	?
0x64F0	?
0x64F4	?
0x64F8	b
0x64FC	c
0x6500	?

esp	0x65F8

Passing Parameters

```
;void function(int a, int b, int c)

_start:
    ;assume a is in eax
    ;assume b is in ebx
    ;assume c is in ecx
    push ecx
    push ebx
    push eax
    call function
```

ip

addr	value
0x64E8	?
0x64EC	?
0x64F0	?
0x64F4	a
0x64F8	b
0x64FC	c
0x6500	?

esp	0x65F4

Passing Parameters

```
;void function(int a, int b, int c)

function:  <- ip
    ...
```

addr	value
0x64E8	?
0x64EC	?
0x64F0	ret addr
0x64F4	a
0x64F8	b
0x64FC	c
0x6500	?

esp	0x65F0

Parameter Passing

- instead of ‘pushing’ parameters onto the stack, pre-allocate enough space on the stack and moved to appropriate places
- Compiler specific, both ways achieve the same thing

Passing Parameters

```
;void function(int a, int b, int c)

_start:
    ;assume a is in eax
    ;assume b is in ebx
    ;assume c is in ecx
ip → sub esp, 0xC ;12 bytes
    mov [esp], eax
    mov [esp+4], ebx
    mov [esp+8], ecx
    call function
```

addr	value
0x64E8	?
0x64EC	?
0x64F0	?
0x64F4	?
0x64F8	?
0x64FC	?
0x6500	?

esp	0x6500

Passing Parameters

```
;void function(int a, int b, int c)

_start:
    ;assume a is in eax
    ;assume b is in ebx
    ;assume c is in ecx
ip → sub esp, 0xC ;12 bytes
    mov [esp], eax
    mov [esp+4], ebx
    mov [esp+8], ecx
    call function
```

addr	value
0x64E8	?
0x64EC	?
0x64F0	?
0x64F4	?
0x64F8	?
0x64FC	?
0x6500	?

esp	0x64F4

Passing Parameters

```
;void function(int a, int b, int c)

_start:
    ;assume a is in eax
    ;assume b is in ebx
    ;assume c is in ecx
    sub esp, 0xC ;12 bytes
ip → mov [esp], eax
    mov [esp+4], ebx
    mov [esp+8], ecx
    call function
```

addr	value
0x64E8	?
0x64EC	?
0x64F0	?
0x64F4	a
0x64F8	?
0x64FC	?
0x6500	?

esp	0x64F4

Passing Parameters

```
;void function(int a, int b, int c)

_start:
    ;assume a is in eax
    ;assume b is in ebx
    ;assume c is in ecx
    sub esp, 0xC ;12 bytes
    mov [esp], eax
ip → mov [esp+4], ebx
    mov [esp+8], ecx
    call function
```

addr	value
0x64E8	?
0x64EC	?
0x64F0	?
0x64F4	a
0x64F8	b
0x64FC	?
0x6500	?

esp	0x64F4

Passing Parameters

```
;void function(int a, int b, int c)

_start:
    ;assume a is in eax
    ;assume b is in ebx
    ;assume c is in ecx
    sub esp, 0xC ;12 bytes
    mov [esp], eax
    mov [esp+4], ebx
ip  mov [esp+8], ecx
    call function
```

addr	value
0x64E8	?
0x64EC	?
0x64F0	?
0x64F4	a
0x64F8	b
0x64FC	c
0x6500	?

esp	0x64F4

Passing Parameters

```
;void function(int a, int b, int c)

function:
    ...
```

ip

addr	value
0x64E8	?
0x64EC	?
0x64F0	ret addr
0x64F4	a
0x64F8	b
0x64FC	c
0x6500	?

esp	0x65F0

CDECL

- eax, ecx, edx are caller-saved
- Other registers are callee-saved

- What does this mean?
- Upon entering a function you may overwrite eax, ecx, edx
- If altering other registers (ebx, esi, edi) you must save their value and restore it before returning

CDECL

```
;can overwrite eax, ecx, edx
function:
    mov eax, 0x11
    add edx, eax
    mov ecx, [edx]
    ret
```

CDECL

```
;Have to preserve ebx, esi, edi
function:
    push ebx                    ;save
    push esi                    ;save
    push edi                    ;save
    mov ebx, 0x11
    add esi, edi
    mov ebx, [edx]
    pop edi                     ;restore
    pop esi                     ;restore
    pop ebx                     ;restore
    ret
```

CDECL

- Since functions are allowed to overwrite eax, edx, ecx caller **can not** rely on their value after a call
- Since functions have to preserve ebx, esi, edi, caller **can** rely on their value after a call

CDECL

```
caller:
    mov ecx, 0x112233
    mov ebx, 0x445566
    call function
;can rely on ebx still having value 0x445566
    add ebx, 0x12
;cannot rely on ecx still having value 0x112233
```

CDECL

- Return value is in eax

```
//C
int function()
{
    return 0;
}
```

```
//assembly
function:
    mov eax, 0
    ret
```

CDECL

- Return value is in eax

```
//C
int function()
{
    return 1;
}
```

```
//assembly
function:
    mov eax, 1
    ret
```

Callee Rules

```
push ebp
mov ebp, esp
```

- Prior to performing any actions the callee should push the current frame pointer (ebp) then move esp into ebp
- ebp is used as the reference to where all variables for a method begin on the stack
  - Stack (esp) may shrink or grow while in a method, but ebp will always point to the beginning of the parameters for that method
- Before returning restore saved ebp

Mov ebp, esp

```
pop ebp
ret
```

Data Movement

- *leave*
- Sets stack pointer to the base frame address
- *Syntax*

  leave
- *Examples*

```
leave –equivalent to:
                    mov esp,ebp
                    pop ebp
```

cdecl

```
int callee(int, int, int);
 int caller(void)
{
      int ret;
      ret = callee(1, 2,
3);
      ret += 5;
      return ret;
}
```

```
 .global  caller
caller:
        push   ebp
        mov    ebp, esp
        push   3
        push   2
        push   1
        call   callee
        add     esp, 12 ;(1)
        add    eax, 5
        leave
        ret
```

(1) – clean up the stack. Function parameters took up 12 bytes (3x 4 byte parameters).

cdecl – parameter example

- Consider the following:

```
func(uint32_t a, uint32_t b,
                uint32_t c){
   //no internal parameters
   .........
   return 0;
}

void main(){
     ......
     m = func(a,b,c);
     ......
}
```

cdecl – parameter example

```
;C    m = func(a,b,c)
main:
……
push c
push b
push a
call _func ;stack shown here
add esp, 12;cleaning up the stack
mov m, eax        ;save the result
```

esp → 0xF4

Addr	Value
0xF0	
0xF4	retaddr
0xF8	a
0xFC	b
0x100	c

	Addr	Value
esp	0xE4	ebx
	0xE8	esi
	0xEC	edi
ebp	0xF0	caller ebp
	0xF4	retaddr
	0xF8	a
	0xFC	b
	0x100	c

```
;C  uint32_t func(uint32_t a,uint32_t b,uint32_t c)

_func:
      PUSH    EBP
      MOV     EBP, ESP
      PUSH    EDI
      PUSH    ESI
      PUSH    EBX                 ;stack shown here
      .
      . MOV     EBX, [EBP + 12]    ; (+0xc) Load b into EBX
      .
      .
      XOR     EAX, EAX            ; Zero the return value
      POP     EBX                 ; Restore the saved registers
      POP     ESI
      POP     EDI
      LEAVE                       ; Equivalent to  MOV    ESP, EBP
                                  ;                POP    EBP
      RET
```

cdecl example 2

- Consider the following c code

```
int MyFunc1(int a, int b)
{
    return a + b;
}
```

- And the following function call

```
x = MyFunc1(2, 3);
```

cdecl example 2

- Caller

```
;x = MyFunc1(2, 3);

push 3
push 2
call _MyFunc1 ;stack shown here
add esp, 8
```

esp →

address	value
0xF0	
0xF4	
0xF8	ret addr
0xFC	2
0x100	3

cdecl example 2

addr	value
0xF0	
0xF4	caller ebp
0xF8	ret addr
0xFC	2
0x100	3

esp/ebp → 0xF4

- Callee

```
;int MyFunc1(int a, int b) {
;     return a + b;
;}

_MyFunc1:
push ebp
mov ebp, esp  ;stack shown here
mov eax, [ebp + 8]  ;loads 2
mov edx, [ebp + 12] ;loads 3
add eax, edx
pop ebp
ret
```

CDECL

- Parameter Passing is always right to left
- Return address is always pushed
- Callee pushes ebp to preserve caller's value

increasing address

addr	value
ebp	Caller ebp
ebp+4	Return Addr
ebp+8	Argument 1
	…
ebp+4+4n	Argument n
	…

CDECL

```
int function(int a, int b)
    [ebp+8] = a
    [ebp+0xc] = b

int function2(int a, int b, int c)
    [ebp+8] = a
    [ebp+0xc] = b
    [ebp+0x10] = c
```

CDECL - Local Variables

- We’ve established that parameters are passed into functions on the stack
- What about local variables?

- Local variables are allocated space on the stack after setting up stack frame (saving ebp)

CDECL – Local Variables

void function()
{
    int x = 0;
    int y = 1;
}

```
function:
ip --> push ebp
       mov ebp, esp
       sub esp, 4 ; allocate space for x
       sub esp, 4 ; allocate space for y
       mov [esp+4], 0   ;set x= 0
       mov [esp], 1     ;set y = 1
       ret
```

addr	value
0x7FC	
0x800	
0x804	
0x808	
0x80C	ret addr

esp	0x80C

CDECL – Local Variables

```
void function()
{
        int x = 0;
        int y = 1;
}
```

addr	value
0x7FC	
0x800	
0x804	
0x808	caller ebp
0x80C	ret addr

```
function:
        push ebp
ip →    mov ebp, esp
        sub esp, 4 ; allocate space for x
        sub esp, 4 ; allocate space for y
        mov [esp+4], 0   ;set x= 0
        mov [esp], 1     ;set y = 1
        ret
```

esp	0x808

CDECL – Local Variables

addr	value
0x7FC	
0x800	
0x804	?
0x808	caller ebp
0x80C	ret addr

void function()
{
    int x = 0;
    int y = 1;
}

esp	0x804

```
function:
      push ebp
      mov ebp, esp
ip →  sub esp, 4 ; allocate space for x
      sub esp, 4 ; allocate space for y
      mov [esp+4], 0   ;set x= 0
      mov [esp], 1     ;set y = 1
      ret
```

CDECL – Local Variables

addr	value
0x7FC	
0x800	?
0x804	?
0x808	caller ebp
0x80C	ret addr

```
void function()
{
        int x = 0;
        int y = 1;
}
```

esp	0x800

```
function:
        push ebp
        mov ebp, esp
        sub esp, 4 ; allocate space for x
ip →    sub esp, 4 ; allocate space for y
        mov [esp+4], 0   ;set x= 0
        mov [esp], 1     ;set y = 1
        ret
```

CDECL – Local Variables

void function()
{
    int x = 0;
    int y = 1;
}

addr	value
0x7FC	
0x800	?
0x804	0
0x808	caller ebp
0x80C	ret addr

esp	0x800

```
function:
        push ebp
        mov ebp, esp
        sub esp, 4 ; allocate space for x
        sub esp, 4 ; allocate space for y
ip →    mov [esp+4], 0   ;set x= 0
        mov [esp], 1     ;set y = 1
        ret
```

CDECL – Local Variables

addr	value
0x7FC	
0x800	1
0x804	0
0x808	caller ebp
0x80C	ret addr

void function()
{
    int x = 0;
    int y = 1;
}

esp	0x800

```
function:
      push ebp
      mov ebp, esp
      sub esp, 4 ; allocate space for x
      sub esp, 4 ; allocate space for y
      mov [esp+4], 0   ;set x= 0
ip →  mov [esp], 1     ;set y = 1
      ret
```

cdecl example 3

- Consider the following, now with local parameters:

```
func(uint32_t a, uint32_t b,
                uint32_t c){
    uint32_t x, y;//local parameters
    .........
    return 0;
}

void main(){
        ......
        m = func(a,b,c);
        ......
}
```

cdecl –example 3

```
;C    m = func(a,b,c)
main:
……
push c
push b
push a
call _func ;stack shown here
add esp, 12;cleaning up the stack
mov m, eax         ;save the result
```

esp → 0xF4

Addr	Value
0xF0	
0xF4	retaddr
0xF8	a
0xFC	b
0x100	c

```
;C  uint32_t func(uint32_t a,uint32_t b,uint32_t c)

_func:
    PUSH     EBP
    MOV      EBP, ESP    ; 8 bytes for two
    SUB      ESP, 08H    ; local variables.
        ;(Stack shown at this point)

    PUSH     EDI         ; These would only be
    PUSH     ESI         ;  pushed if they were used
    PUSH     EBX         ;  in this function.
    . .
    MOV      EAX, [EBP - 8]     ; Load y into EAX
    MOV      EBX, [EBP + 12]    ; Load b into EBX
    .
    .
    XOR      EAX, EAX             ; Zero the return value
    POP      EBX                  ; Restore saved registers
    POP      ESI
    POP      EDI
    LEAVE                         ; Equivalent to  MOV   ESP, EBP
                                  ;                POP   EBP
    RET
```

	Addr	Value
	0xDC	
	0xE0	
	0xE4	
esp →	0xE8	y
	0xEC	x
ebp →	0xF0	caller ebp
	0xF4	retaddr
	0xF8	a
	0xFC	b
	0x100	c

```
;C  uint32_t func(uint32_t a,uint32_t b,uint32_t c)

_func:
    PUSH    EBP
    MOV     EBP, ESP    ; Allocating 8 bytes of storage
    SUB     ESP, 08H    ;  for two local variables.
    PUSH    EDI         ; These would only be
    PUSH    ESI         ;  pushed if they were used
    PUSH    EBX         ;  in this function.
    . ;(Stack shown at this point)
    .
    MOV     EAX, [EBP - 8]     ; Load y into EAX
    MOV     EBX, [EBP + 12]    ; Load b into EBX
    .
    .
    XOR     EAX, EAX           ; Zero the return value
    POP     EBX                ; Restore saved registers
    POP     ESI
    POP     EDI
    LEAVE                      ; Equivalent to  MOV   ESP, EBP
                               ;                POP   EBP
    RET
```

	Addr	Value
esp	0xDC	ebx
	0xE0	esi
	0xE4	edi
	0xE8	y
	0xEC	x
ebp	0xF0	caller ebp
	0xF4	retaddr
	0xF8	a
	0xFC	b
	0x100	c

Local Parameters

- If the method is simple enough local parameters will simply use a register

Stack Alignment

- Some compilers will enforce 16 byte alignment for entering methods
- What does this look like?
  - When allocating space for local parameters allocate enough to align frame to 16 byte boundary
  - Often leads to unused space in the functions stack frame
- You don’t have to do this in your code

C constructs in x86

C constructs in x86

- if (...) { ... }
- if (...) { ... } else { ... }
- if (...) { ... } else if (...) { ... } else { ... }
- while (...) { ... }
- do { ... } while (...);
- for (...; ...; ...) { ... }
- switch (...) { ... }

C constructs in x86

- All of these can be written using comparisons (cmp) and jumps (jmp, je, jne, jl, jle, jg, jge)
- When compiling your program, the compiler does this translation for you
- When writing your own assembly, you need to figure out the translation

Translation: C to Assembly

- Assembly does not have concepts of code blocks {…} like higher level languages do
  - Everything is just a stream of instructions
- Translation step 1: remove code blocks
  - Requires you to rewrite code using goto statements
- Translation step 2: rewrite as assembly

if (...) { ... }

With blocks:

```
if (condition)
{
    code_if_true;
}
```

Without blocks:

```
if (!condition)
     goto skip_block;

code_if_true;

skip_block:
```

if (...) { ... }

With blocks:

```
if (x==5)
{
    x++;
    y=x;
}
```

Without blocks:

```
if (x!=5)
      goto skip_block;

x++;
y=x;

skip_block:
```

if (...) { ... }

C:

```
if (x!=5)
    goto skip_block;

x++;
y=x;

skip_block:
```

x86:

```
cmp dword [x], 5
jne skip

inc dword [x]
mov eax, [x]
mov [y], eax

skip:
```

if (...) { ... } else { ... }

With blocks:

```
if (condition)
{
    code_if_true;
}
else
{
    code_if_false;
}
```

Without blocks:

```
if (!condition)
    goto false_block;

code_if_true;
goto skip_block;

false_block:
code_if_false;

skip_block:
```

if (...) { ... } else { ... }

With blocks:

```
if (x)
{
    x++;
}
else
{
    x--;
}
```

Without blocks:

```
if (!x)
    goto false_block;

x++;
goto skip_block;

false_block:
x--;

skip_block:
```

if (...) { ... } else { ... }

C:

```c
if (!x)
    goto false_block;

x++;
goto skip_block;

false_block:
x--;

skip_block:
```

x86:

```asm
cmp dword [x], 0
je false_block

inc dword [x]
jmp skip_block

false_block:
dec dword [x]

skip_block:
```

if (...) { ... } else if { ... } else { ... }

With blocks:

```
if (condition_1)
{
    code_if_1;
}
else if (condition_2)
{
    code_if_2;
}
else
{
    code_if_false;
}
```

Without blocks:

```
if (!condition_1)
    goto test_2;
code_if_1;
goto skip_block;

test_2:
if (!condition_2)
    goto false_block;
code_if_2;
goto skip_block;

false_block:
code_if_false;

skip_block:
```

if (…) { … } else if { … } else { … }

With blocks:

```
if (score>70)
{
    grade=‘a’;
}
else if (score>50)
{
    grade=‘b’;
}
else
{
    grade=‘c’;
}
```

Without blocks:

```
if (score<=70)
    goto test_2;
grade=‘a’;
goto skip_block;

test_2:
if (score<=50)
    goto false_block;
grade=‘b’;
goto skip_block;

false_block:
grade=‘c’;

skip_block:
```

if (...) { ... } else if { ... } else { ... }

C:

```
if (score<=70)
    goto test_2;
grade=‘a’;
goto skip_block;

test_2:
if (score<=50)
    goto false_block;
grade=‘b’;
goto skip_block;

false_block:
grade=‘c’;

skip_block:
```

x86:

```
cmp dword [score], 70
jle test_2
mov byte [grade], 'a'
jmp skip_block

test_2:
cmp dword [score], 50
jle false_block
mov byte [grade], 'b'
jmp skip_block

false_block:
mov byte [grade], 'c'

skip_block:
```

do { … } while (…);

With blocks:

```
do
{
    code;
}
while (condition);
```

Without blocks:

```
loop:

code;

if (condition)
    goto loop;
```

do { … } while (…);

With blocks:

```
do
{
    y*=x;
    x--;
}
while (x);
```

Without blocks:

```
loop:

y*=x;
x--;

if (x)
    goto loop;
```

do { … } while (…);

C:

```
loop:

y*=x;
x--;

if (x)
    goto loop;
```

x86:

```
loop:

mov eax, [y]
mul dword [x]
mov [y], eax

dec dword [x]

cmp dword [x], 0
jne loop
```

while (...) { ... }

With blocks:

```
while (condition)
{
    code;
}
```

Without blocks:

```
loop:
if (!condition)
    goto done;

code;
goto loop;

done:
```

while (...) { ... }

With blocks:

```
while (tired)
{
    sleep();
}
```

Without blocks:

```
loop:
if (!tired)
    goto done;

sleep();
goto loop;

done:
```

while (…) { … }

C:

```
loop:
if (!tired)
    goto done;

sleep();
goto loop;

done:
```

x86:

```
loop:
cmp dword [tired], 0
je done

call sleep
goto loop

done:
```

for (...; ...; ...) { ... }

With blocks:

```
for (expr_1; expr_2; expr_3)
{
    code;
}
```

Without blocks:

```
expr_1;

loop:
if (!expr_2)
    goto done;
code;
expr_3;
goto loop;

done:
```

for (…; …; …) { … }

With blocks:

```
for (i=0; i<100; i++)
{
    sum+=i;
}
```

Without blocks:

```
i=0;

loop:
if (i>=100)
    goto done;

sum+=i;
i++;
goto loop;

done:
```

for (...; ...; ...) { ... }

C:

```
i=0;

loop:
if (i>=100)
    goto done;

sum+=i;
i++;
goto loop;

done:
```

x86:

```
mov dword [i], 0

loop:
cmp dword [i], 100
jge done

mov eax,[i]
add [sum],eax
inc dword [i]

jmp  loop

done:
```

A Simple Function

```
int factorial(int n)
{
    if (n <= 1)
        return n;
    else
        return n * factorial(n-1);
}
```

```
factorial:
push  ebp
mov   ebp, esp

mov   eax, [ebp+8]
cmp   eax, 1
jle   done

sub   eax, 1
push  eax
call  factorial
mul   dword [ebp+8]

done:
mov   esp, ebp
pop   ebp
ret
```

```
int f(int n)
{
    if (n <= 1)
        return n;
    else
        return n * f(n-1);
}
```

```
00008000  6655          push ebp
00008002  6689E5        mov ebp,esp
00008005  67668B4508    mov eax,[ebp+0x8]
0000800A  6683F801      cmp eax,byte +0x1
0000800E  7E0E          jng 0x1e
00008010  6683E801      sub eax,byte +0x1
00008014  6650          push eax
00008016  E8E7FF        call word 0x0
00008019  6766F76508    mul dword [ebp+0x8]
0000801E  6689EC        mov esp,ebp
00008021  665D          pop ebp
00008023  C3            ret
```

Run Trace

- Let's evaluate a call to factorial with n = 2

```
push 2
call factorial
```

```
        factorial:
8000    push   ebp
8002    mov    ebp, esp

8005    mov    eax, [ebp+8]
800A    cmp    eax, 1
800E    jle    done

8010    sub    eax, 1
8014    push   eax
8016    call   factorial
8019    mul    dword [ebp+8]

        done:
801E    mov    esp, ebp
8021    pop    ebp
8023    ret
```

eax	??

	Address	Value
	0x0ec	
	0x0f0	
	0x0f4	
	0x0f8	
	0x0fc	
esp →	0x100	Ret. Addr
	0x104	2
ebp →	0x108	??

```
     factorial:
8000 push  ebp
8002 mov   ebp, esp

8005 mov   eax, [ebp+8]
800A cmp   eax, 1
800E jle   done

8010 sub   eax, 1
8014 push  eax
8016 call  factorial
8019 mul   dword [ebp+8]

     done:
801E mov   esp, ebp
8021 pop   ebp
8023 ret
```

eax	??

	Address	Value
	0x0ec	
	0x0f0	
	0x0f4	
	0x0f8	
	0x0fc	
esp →	0x100	Ret. Addr
	0x104	2
ebp →	0x108	??

```
     factorial:
8000 push  ebp
8002 mov   ebp, esp

8005 mov   eax, [ebp+8]
800A cmp   eax, 1
800E jle   done

8010 sub   eax, 1
8014 push  eax
8016 call  factorial
8019 mul   dword [ebp+8]

     done:
801E mov   esp, ebp
8021 pop   ebp
8023 ret
```

eax	??

	Address	Value
	0x0ec	
	0x0f0	
	0x0f4	
	0x0f8	
esp →	0x0fc	0x108
	0x100	Ret. Addr
	0x104	2
ebp →	0x108	??

```
        factorial:
8000    push   ebp
8002    mov    ebp, esp

8005    mov    eax, [ebp+8]
800A    cmp    eax, 1
800E    jle    done

8010    sub    eax, 1
8014    push   eax
8016    call   factorial
8019    mul    dword [ebp+8]

        done:
801E    mov    esp, ebp
8021    pop    ebp
8023    ret
```

eax	??

	Address	Value
	0x0ec	
	0x0f0	
	0x0f4	
	0x0f8	
esp →	0x0fc	0x108
ebp	0x100	Ret. Addr
	0x104	2
	0x108	??

```
      factorial:
8000  push  ebp
8002  mov   ebp, esp

8005  mov   eax, [ebp+8]
800A  cmp   eax, 1
800E  jle   done

8010  sub   eax, 1
8014  push  eax
8016  call  factorial
8019  mul   dword [ebp+8]

      done:
801E  mov   esp, ebp
8021  pop   ebp
8023  ret
```

eax	2

	Address	Value
	0x0ec	
	0x0f0	
	0x0f4	
	0x0f8	
esp →	0x0fc	0x108
ebp	0x100	Ret. Addr
	0x104	2
	0x108	??

```
        factorial:
8000    push   ebp
8002    mov    ebp, esp

8005    mov    eax, [ebp+8]
800A    cmp    eax, 1
800E    jle    done

8010    sub    eax, 1
8014    push   eax
8016    call   factorial
8019    mul    dword [ebp+8]

        done:
801E    mov    esp, ebp
8021    pop    ebp
8023    ret
```

eax	2

	Address	Value
	0x0ec	
	0x0f0	
	0x0f4	
	0x0f8	
esp →	0x0fc	0x108
ebp	0x100	Ret. Addr
	0x104	2
	0x108	??

```
        factorial:
8000    push   ebp
8002    mov    ebp, esp

8005    mov    eax, [ebp+8]
800A    cmp    eax, 1
800E    jle    done

8010    sub    eax, 1
8014    push   eax
8016    call   factorial
8019    mul    dword [ebp+8]

        done:
801E    mov    esp, ebp
8021    pop    ebp
8023    ret
```

eax	2

	Address	**Value**
	0x0ec	
	0x0f0	
	0x0f4	
	0x0f8	
esp →	0x0fc	0x108
ebp	0x100	Ret. Addr
	0x104	2
	0x108	??

```
      factorial:
8000  push  ebp
8002  mov   ebp, esp

8005  mov   eax, [ebp+8]
800A  cmp   eax, 1
800E  jle   done

8010  sub   eax, 1
8014  push  eax
8016  call  factorial
8019  mul   dword [ebp+8]

      done:
801E  mov   esp, ebp
8021  pop   ebp
8023  ret
```

eax	1

	Address	Value
	0x0ec	
	0x0f0	
	0x0f4	
	0x0f8	
esp →	0x0fc	0x108
ebp	0x100	Ret. Addr
	0x104	2
	0x108	??

```
     factorial:
8000 push  ebp
8002 mov   ebp, esp

8005 mov   eax, [ebp+8]
800A cmp   eax, 1
800E jle   done

8010 sub   eax, 1
8014 push  eax
8016 call  factorial
8019 mul   dword [ebp+8]

     done:
801E mov   esp, ebp
8021 pop   ebp
8023 ret
```

eax	1

	Address	Value
	0x0ec	
	0x0f0	
	0x0f4	
esp →	0x0f8	1
ebp →	0x0fc	0x108
	0x100	Ret. Addr
	0x104	2
	0x108	??

```
        factorial:
8000    push    ebp
8002    mov     ebp, esp

8005    mov     eax, [ebp+8]
800A    cmp     eax, 1
800E    jle     done

8010    sub     eax, 1
8014    push    eax
8016    call    factorial
8019    mul     dword [ebp+8]

        done:
801E    mov     esp, ebp
8021    pop     ebp
8023    ret
```

eax	1

	Address	Value
	0x0ec	
	0x0f0	
esp →	0x0f4	0x8019
	0x0f8	1
ebp →	0x0fc	0x108
	0x100	Ret. Addr
	0x104	2
	0x108	??

```
        factorial:
8000    push  ebp
8002    mov   ebp, esp

8005    mov   eax, [ebp+8]
800A    cmp   eax, 1
800E    jle   done

8010    sub   eax, 1
8014    push  eax
8016    call  factorial
8019    mul   dword [ebp+8]

        done:
801E    mov   esp, ebp
8021    pop   ebp
8023    ret
```

eax	1

	Address	Value
	0x0ec	
esp →	0x0f0	0x0fc
	0x0f4	0x8019
	0x0f8	1
ebp →	0x0fc	0x108
	0x100	Ret. Addr
	0x104	2
	0x108	??

```
     factorial:
8000 push  ebp
8002 mov   ebp, esp

8005 mov   eax, [ebp+8]
800A cmp   eax, 1
800E jle   done

8010 sub   eax, 1
8014 push  eax
8016 call  factorial
8019 mul   dword [ebp+8]

     done:
801E mov   esp, ebp
8021 pop   ebp
8023 ret
```

eax	1

	Address	Value
	0x0ec	
esp → ebp	0x0f0	0x0fc
	0x0f4	0x8019
	0x0f8	1
	0x0fc	0x108
	0x100	Ret. Addr
	0x104	2
	0x108	??

```
      factorial:
8000  push  ebp
8002  mov   ebp, esp

8005  mov   eax, [ebp+8]
800A  cmp   eax, 1
800E  jle   done

8010  sub   eax, 1
8014  push  eax
8016  call  factorial
8019  mul   dword [ebp+8]

      done:
801E  mov   esp, ebp
8021  pop   ebp
8023  ret
```

eax	1

	Address	Value
	0x0ec	
esp →	0x0f0	0x0fc
ebp	0x0f4	0x8019
	0x0f8	1
	0x0fc	0x108
	0x100	Ret. Addr
	0x104	2
	0x108	??

```
        factorial:
8000    push   ebp
8002    mov    ebp, esp

8005    mov    eax, [ebp+8]
800A    cmp    eax, 1
800E    jle    done

8010    sub    eax, 1
8014    push   eax
8016    call   factorial
8019    mul    dword [ebp+8]

        done:
801E    mov    esp, ebp
8021    pop    ebp
8023    ret
```

eax	1

esp →
ebp

Address	Value
0x0ec	
0x0f0	0x0fc
0x0f4	0x8019
0x0f8	1
0x0fc	0x108
0x100	Ret. Addr
0x104	2
0x108	??

```
        factorial:
8000    push    ebp
8002    mov     ebp, esp

8005    mov     eax, [ebp+8]
800A    cmp     eax, 1
800E    jle     done

8010    sub     eax, 1
8014    push    eax
8016    call    factorial
8019    mul     dword [ebp+8]

        done:
801E    mov     esp, ebp
8021    pop     ebp
8023    ret
```

eax	1

esp →
ebp

Address	Value
0x0ec	
0x0f0	0x0fc
0x0f4	0x8019
0x0f8	1
0x0fc	0x108
0x100	Ret. Addr
0x104	2
0x108	??

```
        factorial:
8000    push   ebp
8002    mov    ebp, esp

8005    mov    eax, [ebp+8]
800A    cmp    eax, 1
800E    jle    done

8010    sub    eax, 1
8014    push   eax
8016    call   factorial
8019    mul    dword [ebp+8]

        done:
801E    mov    esp, ebp
8021    pop    ebp
8023    ret
```

eax	1

esp →
ebp

Address	Value
0x0ec	
0x0f0	0x0fc
0x0f4	0x8019
0x0f8	1
0x0fc	0x108
0x100	Ret. Addr
0x104	2
0x108	??

```
     factorial:
8000 push  ebp
8002 mov   ebp, esp

8005 mov   eax, [ebp+8]
800A cmp   eax, 1
800E jle   done

8010 sub   eax, 1
8014 push  eax
8016 call  factorial
8019 mul   dword [ebp+8]

     done:
801E mov   esp, ebp
8021 pop   ebp
8023 ret
```

eax	1

	Address	Value
	0x0ec	
esp → ebp	0x0f0	0x0fc
	0x0f4	0x8019
	0x0f8	1
	0x0fc	0x108
	0x100	Ret. Addr
	0x104	2
	0x108	??

```
        factorial:
8000    push   ebp
8002    mov    ebp, esp

8005    mov    eax, [ebp+8]
800A    cmp    eax, 1
800E    jle    done

8010    sub    eax, 1
8014    push   eax
8016    call   factorial
8019    mul    dword [ebp+8]

        done:
801E    mov    esp, ebp
8021    pop    ebp
8023    ret
```

eax	1

	Address	Value
	0x0ec	
	0x0f0	0x0fc
esp →	0x0f4	0x8019
	0x0f8	1
ebp →	0x0fc	0x108
	0x100	Ret. Addr
	0x104	2
	0x108	??

```
        factorial:
8000    push   ebp
8002    mov    ebp, esp

8005    mov    eax, [ebp+8]
800A    cmp    eax, 1
800E    jle    done

8010    sub    eax, 1
8014    push   eax
8016    call   factorial
8019    mul    dword [ebp+8]

        done:
801E    mov    esp, ebp
8021    pop    ebp
8023    ret
```

eax	1

	Address	Value
	0x0ec	
	0x0f0	0x0fc
	0x0f4	0x8019
esp →	0x0f8	1
ebp →	0x0fc	0x108
	0x100	Ret. Addr
	0x104	2
	0x108	??

```
        factorial:
8000    push   ebp
8002    mov    ebp, esp

8005    mov    eax, [ebp+8]
800A    cmp    eax, 1
800E    jle    done

8010    sub    eax, 1
8014    push   eax
8016    call   factorial
8019    mul    dword [ebp+8]

        done:
801E    mov    esp, ebp
8021    pop    ebp
8023    ret
```

eax	1

	Address	Value
	0x0ec	
	0x0f0	0x0fc
	0x0f4	0x8019
esp →	0x0f8	1
ebp →	0x0fc	0x108
	0x100	Ret. Addr
	0x104	2
	0x108	??

```
        factorial:
8000    push   ebp
8002    mov    ebp, esp

8005    mov    eax, [ebp+8]
800A    cmp    eax, 1
800E    jle    done

8010    sub    eax, 1
8014    push   eax
8016    call   factorial
8019    mul    dword [ebp+8]

        done:
801E    mov    esp, ebp
8021    pop    ebp
8023    ret
```

eax	2

	Address	Value
	0x0ec	
	0x0f0	0x0fc
	0x0f4	0x8019
esp →	0x0f8	1
ebp →	0x0fc	0x108
	0x100	Ret. Addr
	0x104	2
	0x108	??

```
        factorial:
8000    push   ebp
8002    mov    ebp, esp

8005    mov    eax, [ebp+8]
800A    cmp    eax, 1
800E    jle    done

8010    sub    eax, 1
8014    push   eax
8016    call   factorial
8019    mul    dword [ebp+8]

        done:
801E    mov    esp, ebp
8021    pop    ebp
8023    ret
```

eax	2

	Address	Value
	0x0ec	
	0x0f0	0x0fc
	0x0f4	0x8019
	0x0f8	1
esp →	0x0fc	0x108
ebp	0x100	Ret. Addr
	0x104	2
	0x108	??

```
      factorial:
8000  push  ebp
8002  mov   ebp, esp

8005  mov   eax, [ebp+8]
800A  cmp   eax, 1
800E  jle   done

8010  sub   eax, 1
8014  push  eax
8016  call  factorial
8019  mul   dword [ebp+8]

      done:
801E  mov   esp, ebp
8021  pop   ebp
8023  ret
```

eax	2

	Address	Value
	0x0ec	
	0x0f0	0x0fc
	0x0f4	0x8019
	0x0f8	1
	0x0fc	0x108
esp →	0x100	Ret. Addr
	0x104	2
ebp →	0x108	??

```
        factorial:
8000    push   ebp
8002    mov    ebp, esp

8005    mov    eax, [ebp+8]
800A    cmp    eax, 1
800E    jle    done

8010    sub    eax, 1
8014    push   eax
8016    call   factorial
8019    mul    dword [ebp+8]

        done:
801E    mov    esp, ebp
8021    pop    ebp
8023    ret
```

eax	2

	Address	Value
	0x0ec	
	0x0f0	0x0fc
	0x0f4	0x8019
	0x0f8	1
	0x0fc	0x108
	0x100	Ret. Addr
esp →	0x104	2
ebp →	0x108	??

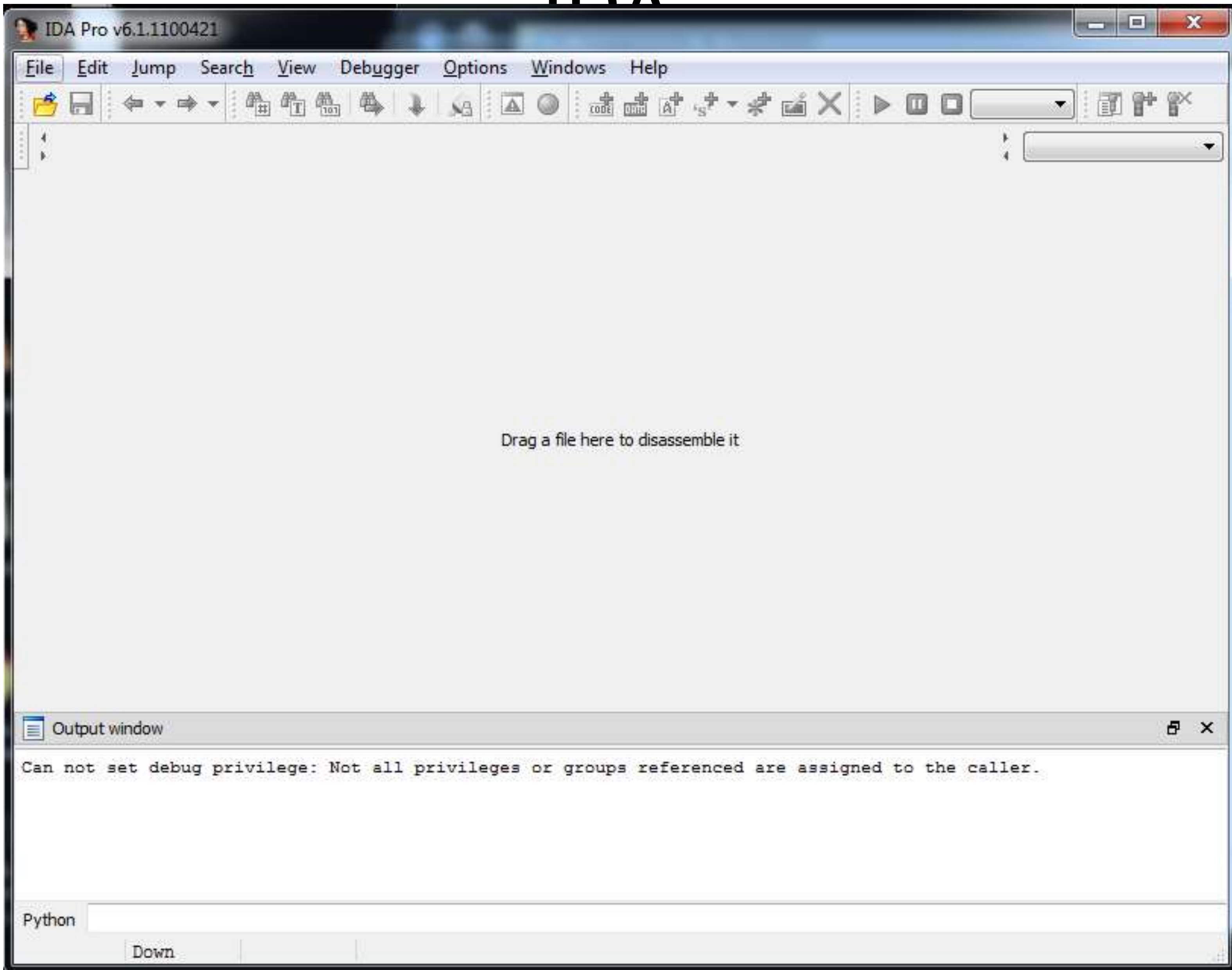
Objdump

- Objdump is a tool that will let you see the assembly code for any application
- syntax: objdump <application> -Mintel –d
  - -Mintel says you want intel assembly syntax
  - -d says to disassemble the code

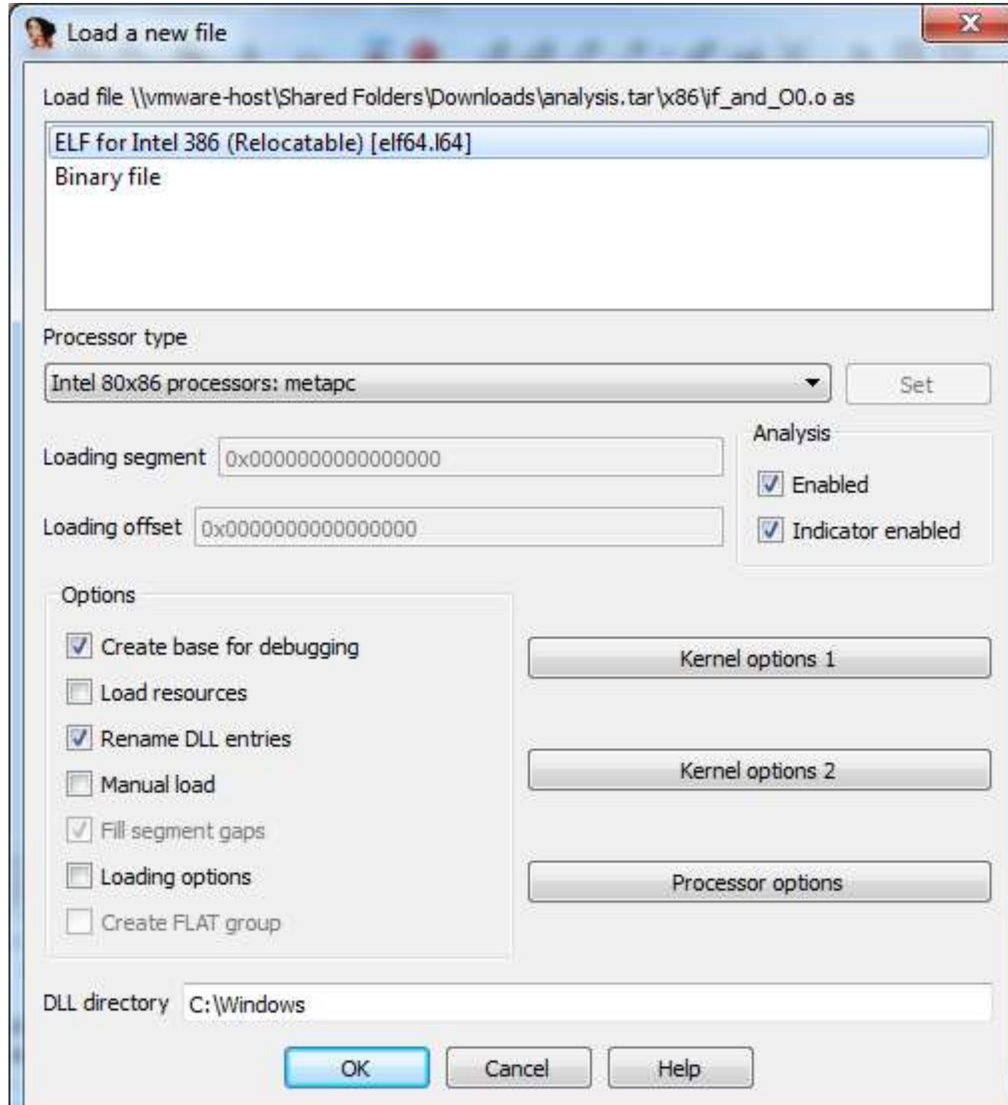
IDA

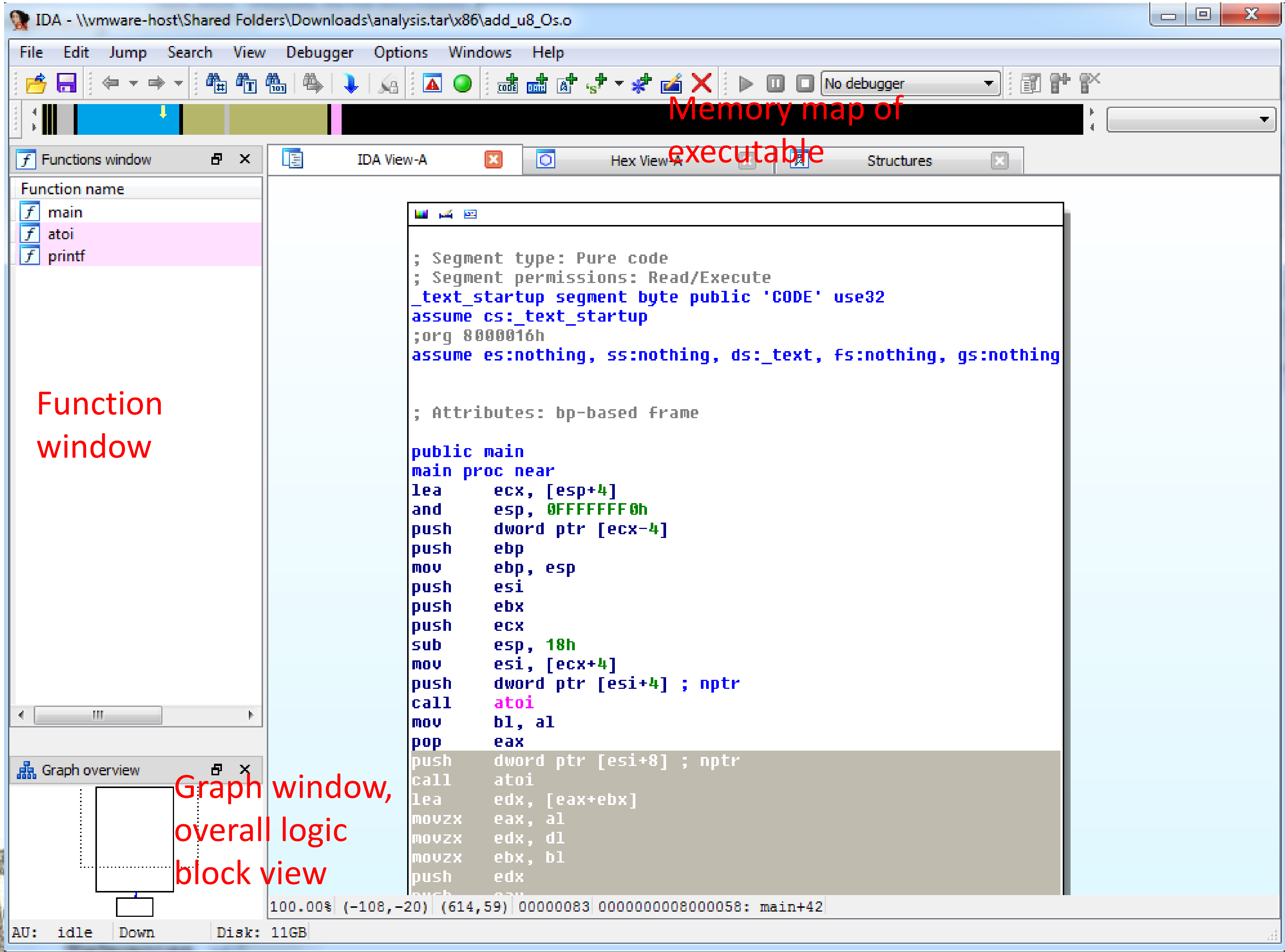
- IDA – Interactive Disassembler
- Allows for binary visualization of disassembly
- About
  – https://hex-rays.com/products/ida/index.shtml
- Download
  – https://hex-rays.com/products/ida/support/download.shtml
- Free version supports x86

IDA


IDA Pro v6.1.1100421
File Edit Jump Search View Debugger Options Windows Help
Drag a file here to disassemble it
Output window
Can not set debug privilege: Not all privileges or groups referenced are assigned to the caller.
Python
Down

- IDA recognizes many common file formats
- If it gets it wrong you can select generic ‘binary file’
- Processor type drop down to change architectures





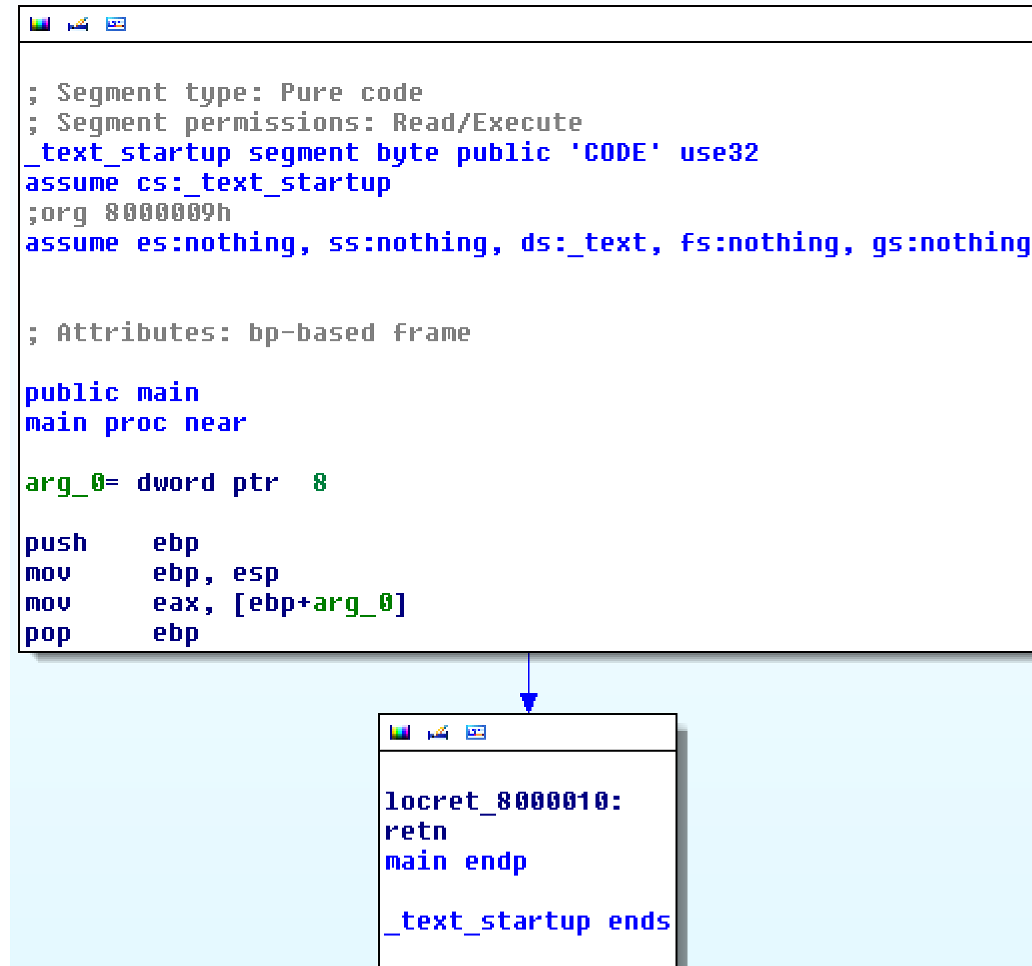
IDA - \\vmware-host\Shared Folders\Downloads\analysis.tar\x86\add_u8_Os.o
File Edit Jump Search View Debugger Options Windows Help
No debugger
Memory map of executable
Functions window
Function name
main
atoi
printf
Function window
IDA View-A
Hex View-A
Structures
; Segment type: Pure code
; Segment permissions: Read/Execute
_text_startup segment byte public 'CODE' use32
assume cs:_text_startup
;org 8000016h
assume es:nothing, ss:nothing, ds:_text, fs:nothing, gs:nothing
; Attributes: bp-based frame
public main
main proc near
lea ecx, [esp+4]
and esp, 0FFFFFFF0h
push dword ptr [ecx-4]
push ebp
mov ebp, esp
push esi
push ebx
push ecx
sub esp, 18h
mov esi, [ecx+4]
push dword ptr [esi+4] ; nptr
call atoi
mov bl, al
pop eax
push dword ptr [esi+8] ; nptr
call atoi
lea edx, [eax+ebx]
movzx eax, al
movzx edx, dl
movzx ebx, bl
push edx
Graph overview
Graph window, overall logic block view
100.00% (-108,-20) (614,59) 00000083 0000000008000058: main+42
AU: idle Down Disk: 11GB

IDA

- IDA shows code in Basic Blocks
- basic block has:
  – one entry point, meaning no code within it is the destination of a jump instruction anywhere in the program;
  – one exit point, meaning only the last instruction can cause the program to begin executing code in a different basic block.

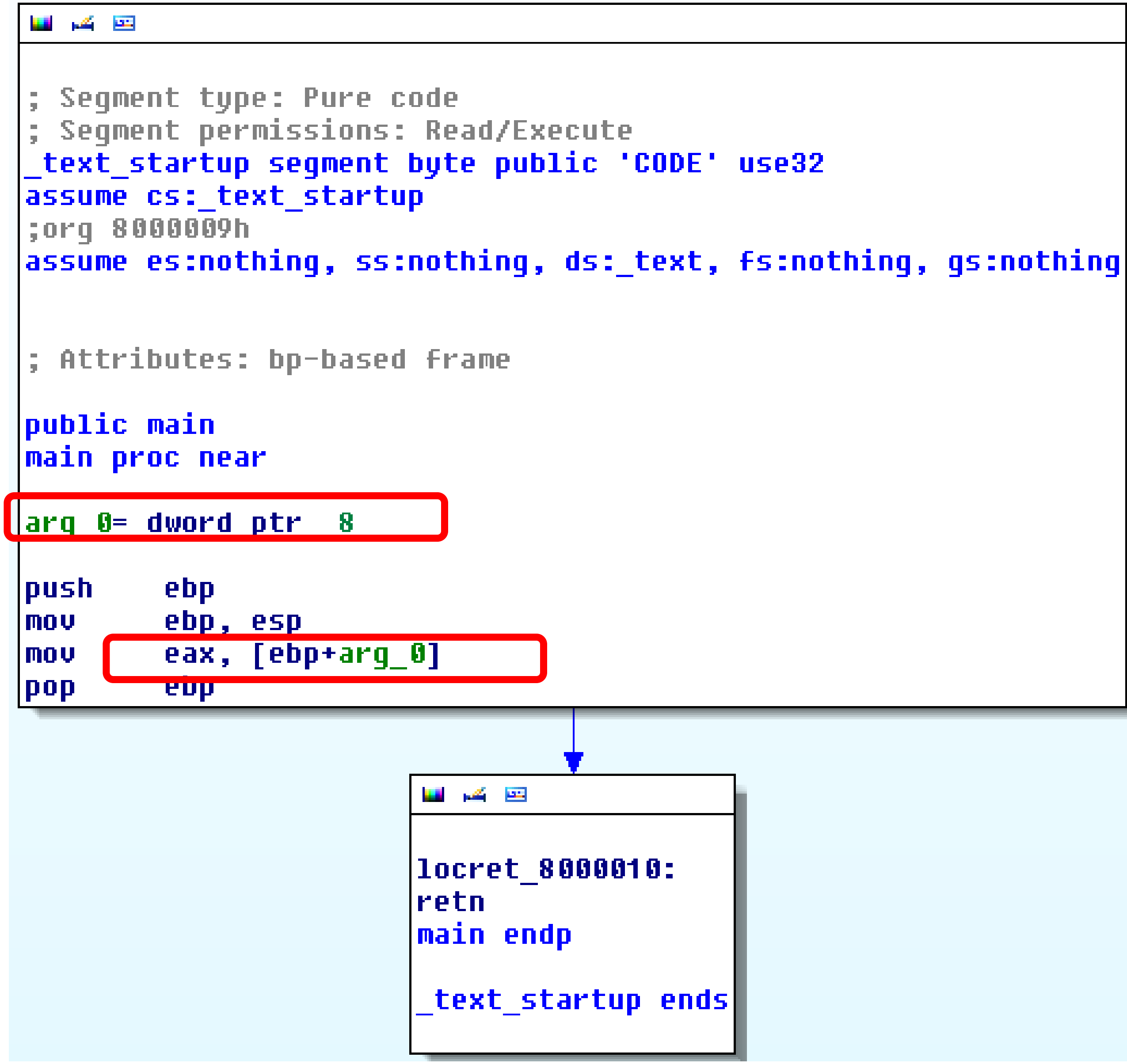
IDA Basic Block

```
int main(int argc, char* argv[])
{
      return argc;
}
```



IDA Basic Block

IDA is smart enough to know that the first argument always starts at ebp+8, so it renames that offset to arg_0 to make it easier to read



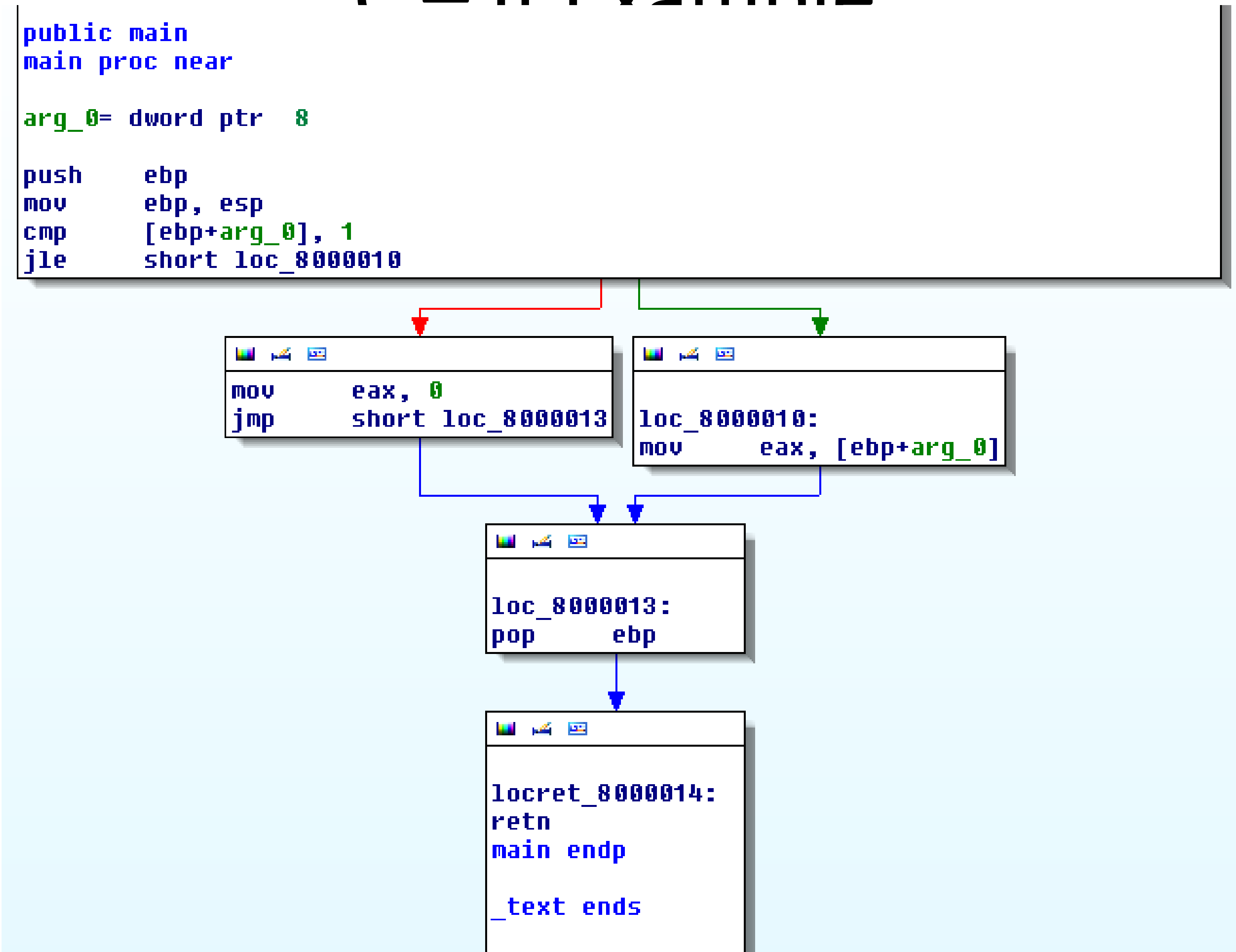
IDA Paths

- IDA shows 3 different types of paths between basic blocks
- RED – Path taken if conditional jump is not taken
- GREEN – Path taken if conditional jump is taken
- BLUE – Guaranteed path

C – If Example

```
int main(int argc, char* argv[])
{
    if (argc > 1)
        return 0;

    return argc;
}
```

C – If Example


public main
main proc near
arg_0= dword ptr 8
push ebp
mov ebp, esp
cmp [ebp+arg_0], 1
jle short loc_8000010
mov eax, 0
jmp short loc_8000013
loc_8000010:
mov eax, [ebp+arg_0]
loc_8000013:
pop ebp
locret_8000014:
retn
main endp
_text ends

C – If Example

```
public main
main proc near

arg_0= dword ptr  8

push     ebp
mov      ebp, esp
cmp      [ebp+arg_0], 1
jle      short loc_8000010
```

Taken if arg_0 greater than 1

Taken if arg_0 less than or equal to 1

```
mov      eax, 0
jmp      short loc_8000013
```

```
loc_8000010:
mov      eax, [ebp+arg_0]
```

```
loc_8000013:
pop      ebp
```

```
locret_8000014:
retn
main endp

_text ends
```

cmp [ebp+arg_0], 1
Compares argc to 1

jle – Jump less than equal

IDA – If And Example

- Lets look at some common C structures in assembly

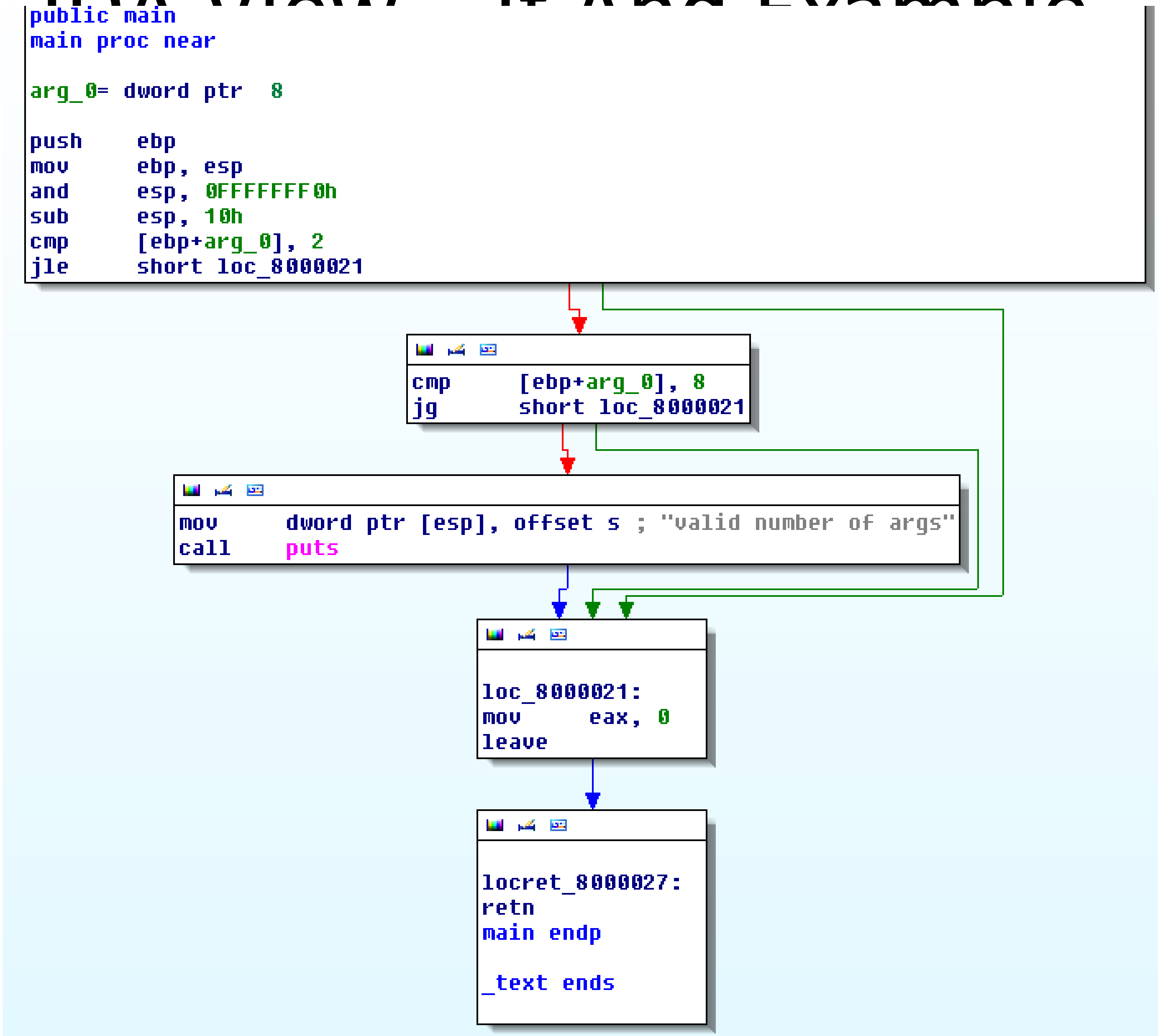
```
#include <stdio.h>

int main(int argc, char* argv[])
{
	if (argc >= 3 && argc <= 8)
	{
		printf("valid number of args\n");
	}

	return 0;
}
```

IDA – If And Example

- Objdump view - Hard to read!

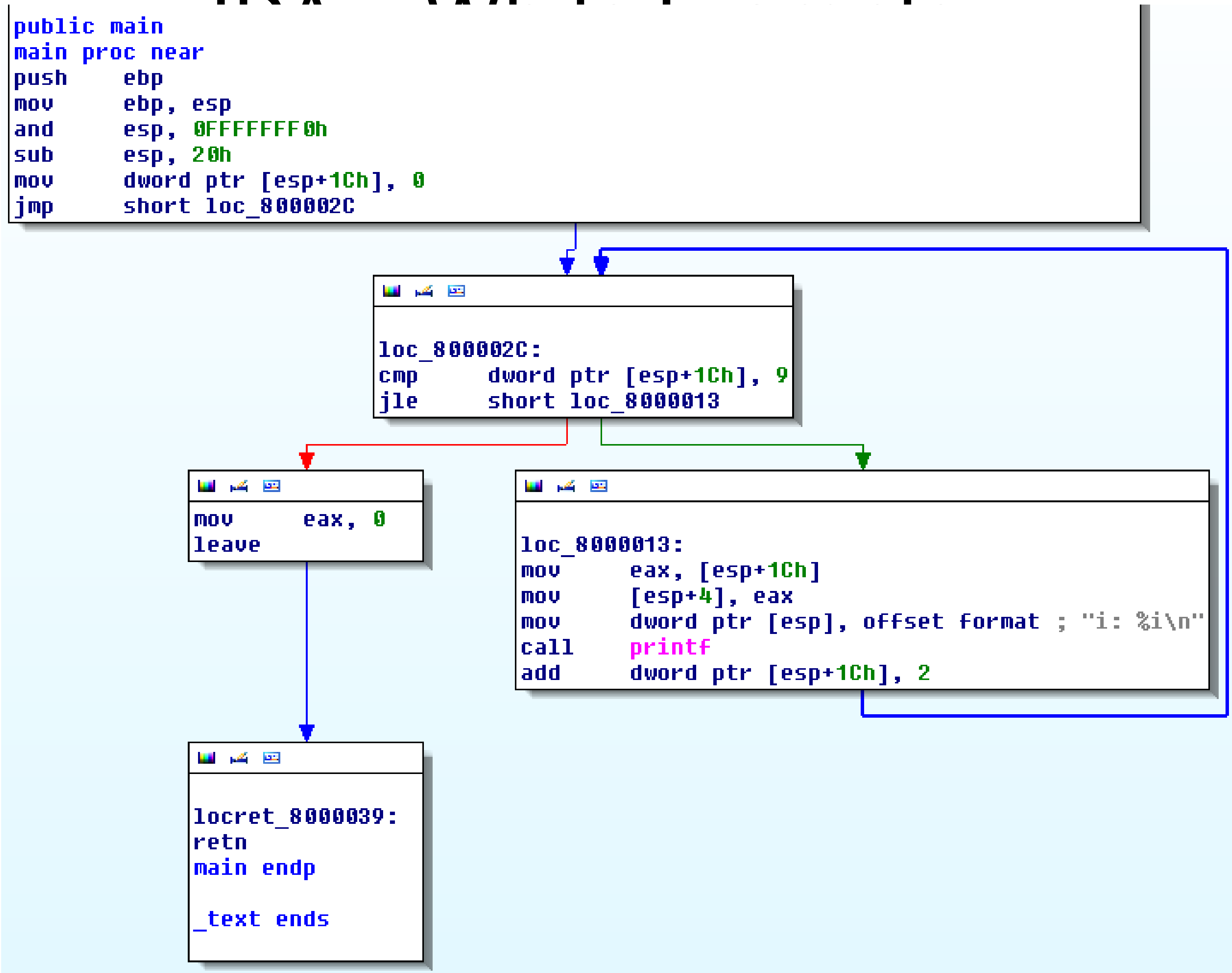
```
000000000040051c <main>:
  40051c:	55                   	push   rbp
  40051d:	48 89 e5             	mov    rbp,rsp
  400520:	48 83 ec 10          	sub    rsp,0x10
  400524:	89 7d fc             	mov    DWORD PTR [rbp-0x4],edi
  400527:	48 89 75 f0          	mov    QWORD PTR [rbp-0x10],rsi
  40052b:	83 7d fc 02          	cmp    DWORD PTR [rbp-0x4],0x2
  40052f:	7e 10                	jle    400541 <main+0x25>
  400531:	83 7d fc 08          	cmp    DWORD PTR [rbp-0x4],0x8
  400535:	7f 0a                	jg     400541 <main+0x25>
  400537:	bf f4 05 40 00       	mov    edi,0x4005f4
  40053c:	e8 af fe ff ff       	call   4003f0 <puts@plt>
  400541:	b8 00 00 00 00       	mov    eax,0x0
  400546:	c9                   	leave
  400547:	c3                   	ret
```

IDA View – If And Example



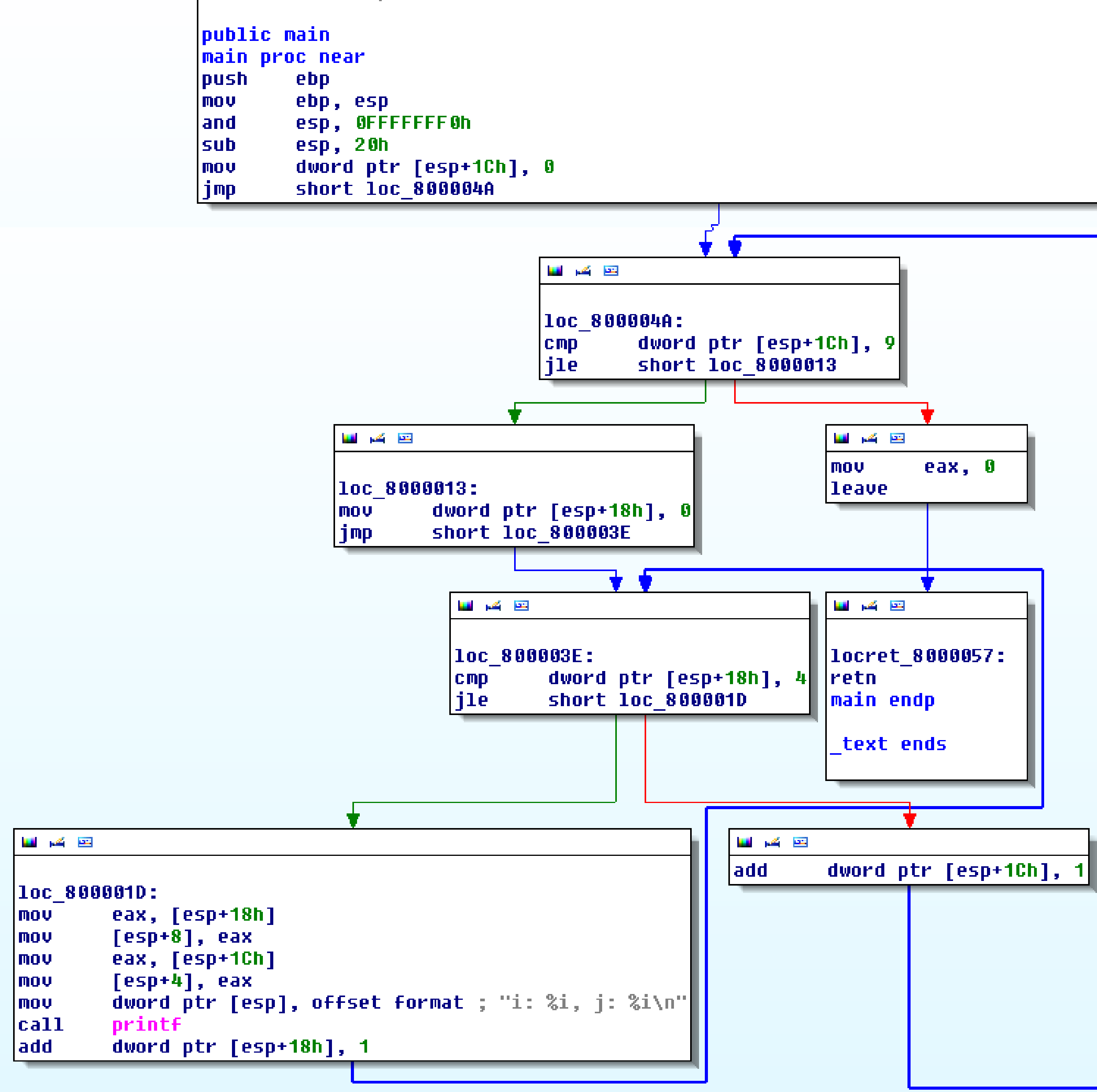
IDA- While Example

```
#include <stdio.h>

int main(int argc, char* argv[])
{
	int i;

	i = 0;
	while (i < 10)
	{
		printf("i: %i\n", i);
		i += 2;
	}

	return 0;
}
```


public main
main proc near
push ebp
mov ebp, esp
and esp, 0FFFFFFF0h
sub esp, 20h
mov dword ptr [esp+1Ch], 0
jmp short loc_800002C
loc_800002C:
cmp dword ptr [esp+1Ch], 9
jle short loc_8000013
mov eax, 0
leave
loc_8000013:
mov eax, [esp+1Ch]
mov [esp+4], eax
mov dword ptr [esp], offset format ; "i: %i\n"
call printf
add dword ptr [esp+1Ch], 2
locret_8000039:
retn
main endp
_text ends

IDA – While/While

```
#include <stdio.h>
int main(int argc, char* argv[]){
	int i, j;
	i = 0;
	while (i < 10)
	{
		j = 0;
		while (j < 5)
		{
			printf("i: %i, j: %i\n", i, j);
			j++;
		}
		i++;
	}
	return 0;
}
```

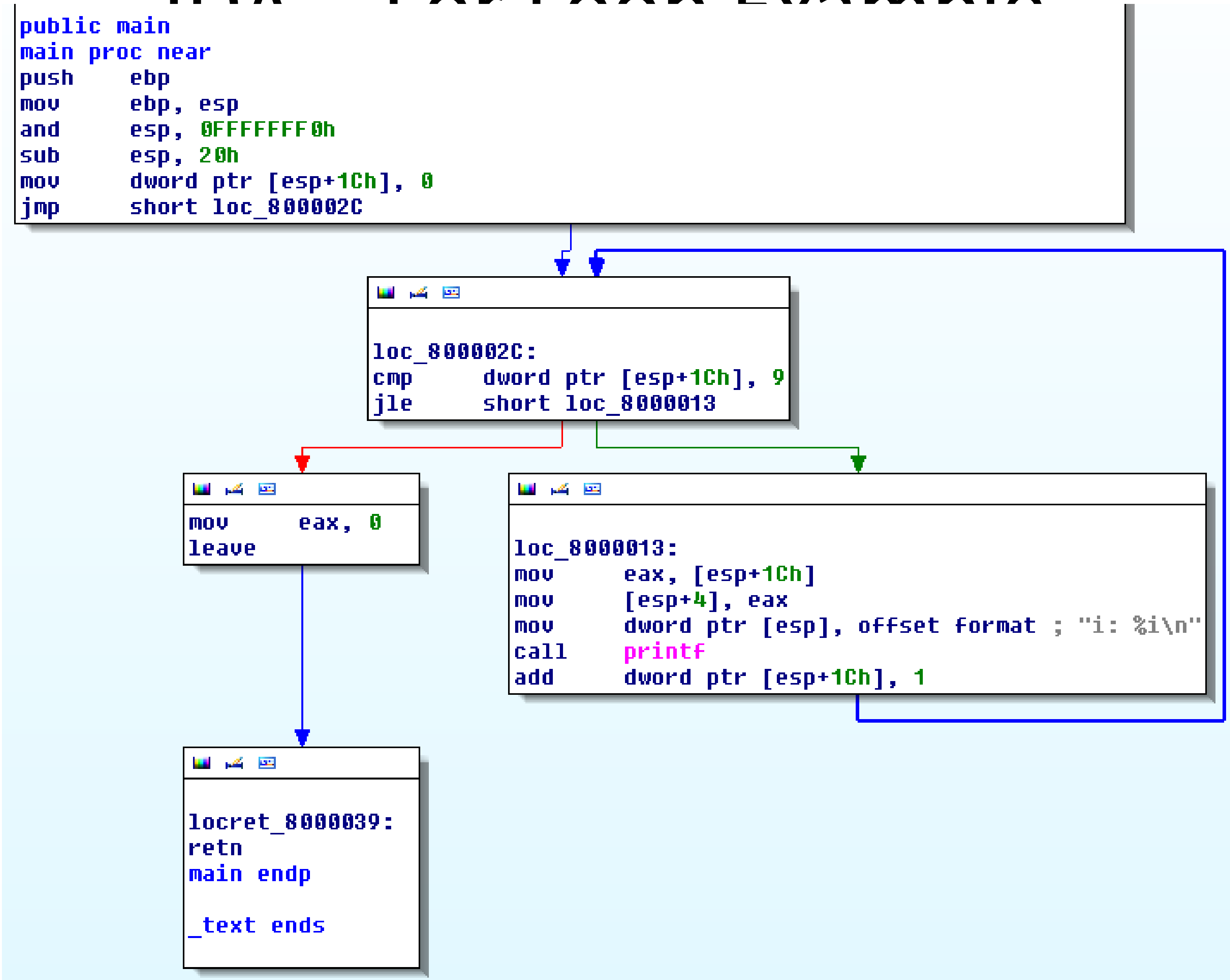

public main
main proc near
push ebp
mov ebp, esp
and esp, 0FFFFFFF0h
sub esp, 20h
mov dword ptr [esp+1Ch], 0
jmp short loc_800004A
loc_800004A:
cmp dword ptr [esp+1Ch], 9
jle short loc_8000013
loc_8000013:
mov dword ptr [esp+18h], 0
jmp short loc_800003E
mov eax, 0
leave
loc_800003E:
cmp dword ptr [esp+18h], 4
jle short loc_800001D
locret_8000057:
retn
main endp
_text ends
loc_800001D:
mov eax, [esp+18h]
mov [esp+8], eax
mov eax, [esp+1Ch]
mov [esp+4], eax
mov dword ptr [esp], offset format ; "i: %i, j: %i\n"
call printf
add dword ptr [esp+18h], 1
add dword ptr [esp+1Ch], 1

atduo

IDA – For Loop Example

```
#include <stdio.h>

int main(int argc, char* argv[])
{
	int i;

	for (i = 0; i < 10; i++)
	{
		printf("i: %i\n", i);
	}

	return 0;
}
```

IDA – For Loop Example



Goodies

- Windows VM (dazzlecat/dazzleme)
- ~Desktop/training/assembly_samples
  - src – Contains simple c programs that incorporate a basic logic flow (if/else/etc)
  - bin – compiled programs of the src with different optimization levels
    - -O0 optimization for compilation time (default)
    - -O2 optimization more for code size and execution time
    - -Os optimization for code size

IDA –Example

- IDA Patching- Demo

dazzlecatduo on github
dazzle.cat.duo@gmail.com

Dazzlecatduo logo source:
http://embed.polyvoreimg.com/cgi/img-thing/size/y/tid/38030333.jpg