Software Security

Module 1 - x86 Assembly & C Call Stack

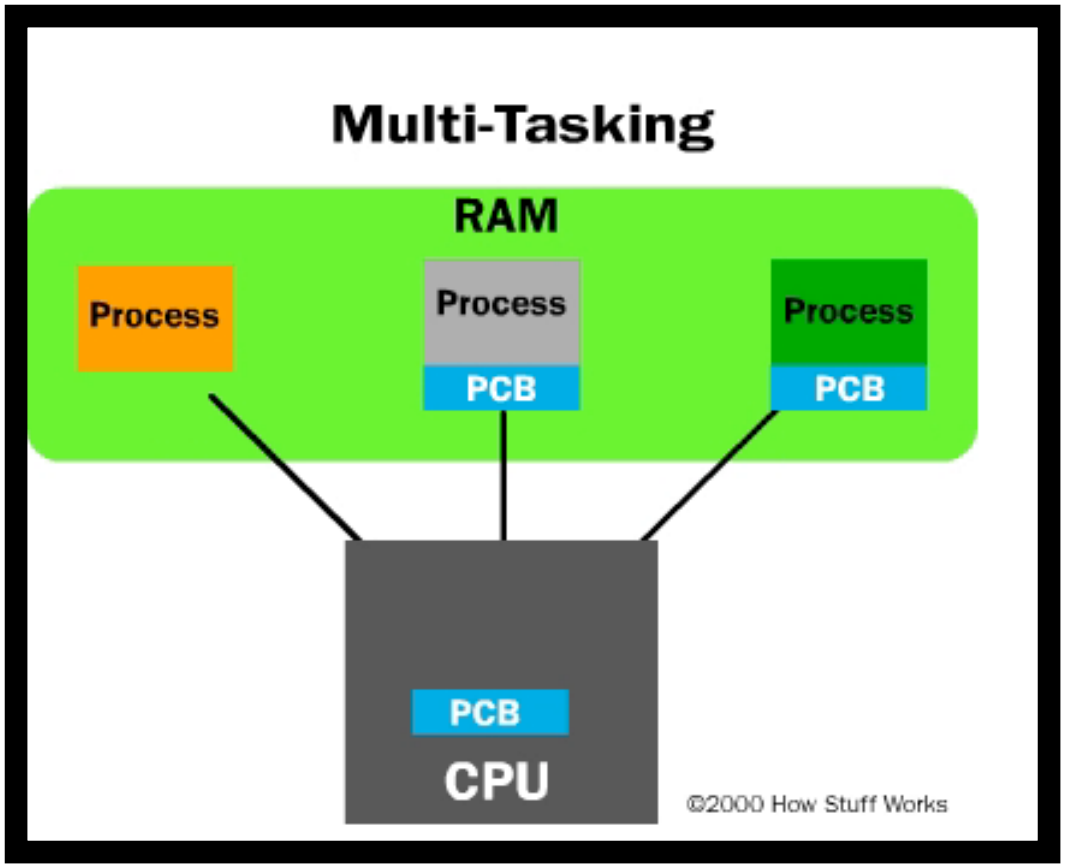
CMPE279
Software Security Technologies
San Jose State University

CPU & The OS

A Quick Review

The Operating System

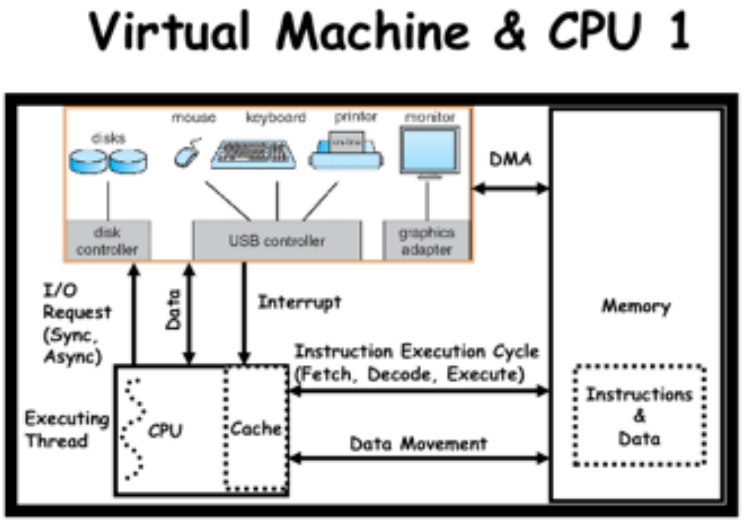
A Quick Review

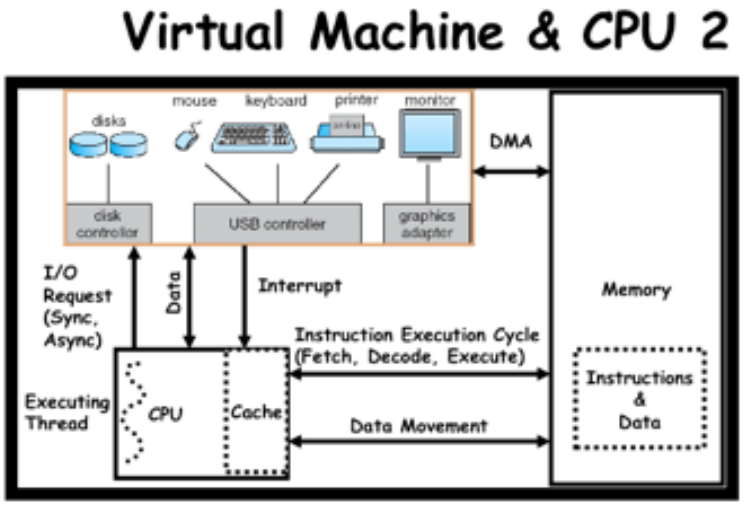


How does an OS (like Unix) allow concurrent programs to use shared resources:

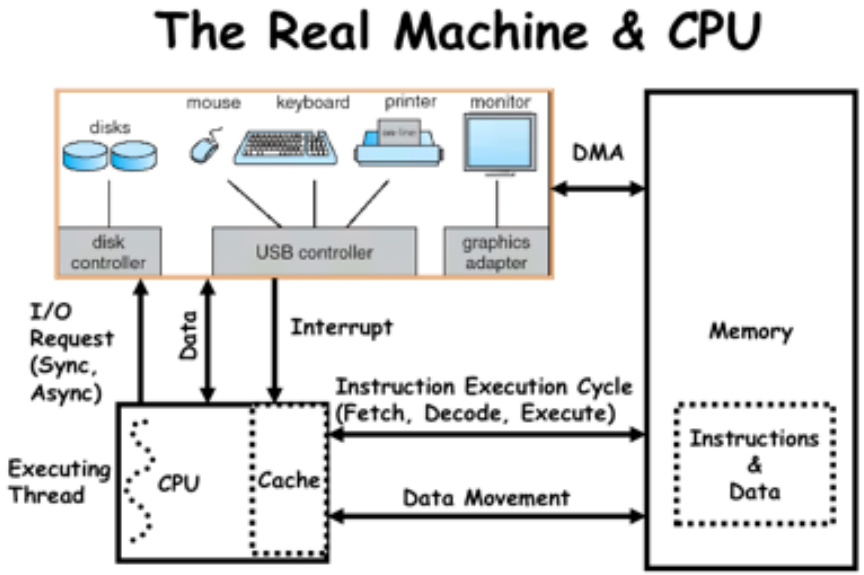
-- CPU
-- Memory
-- Disks

The Process Abstraction

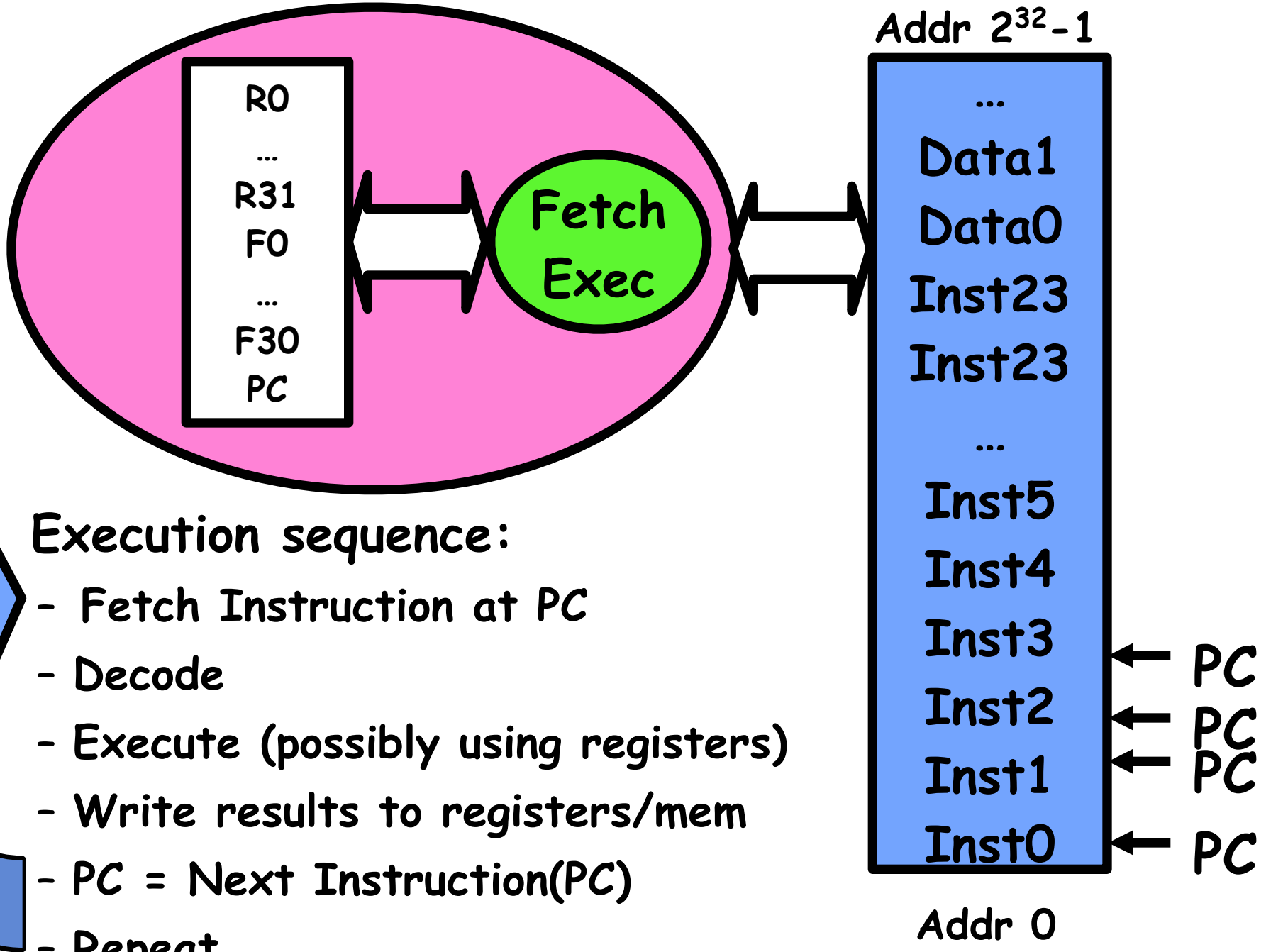




The Context Switch!

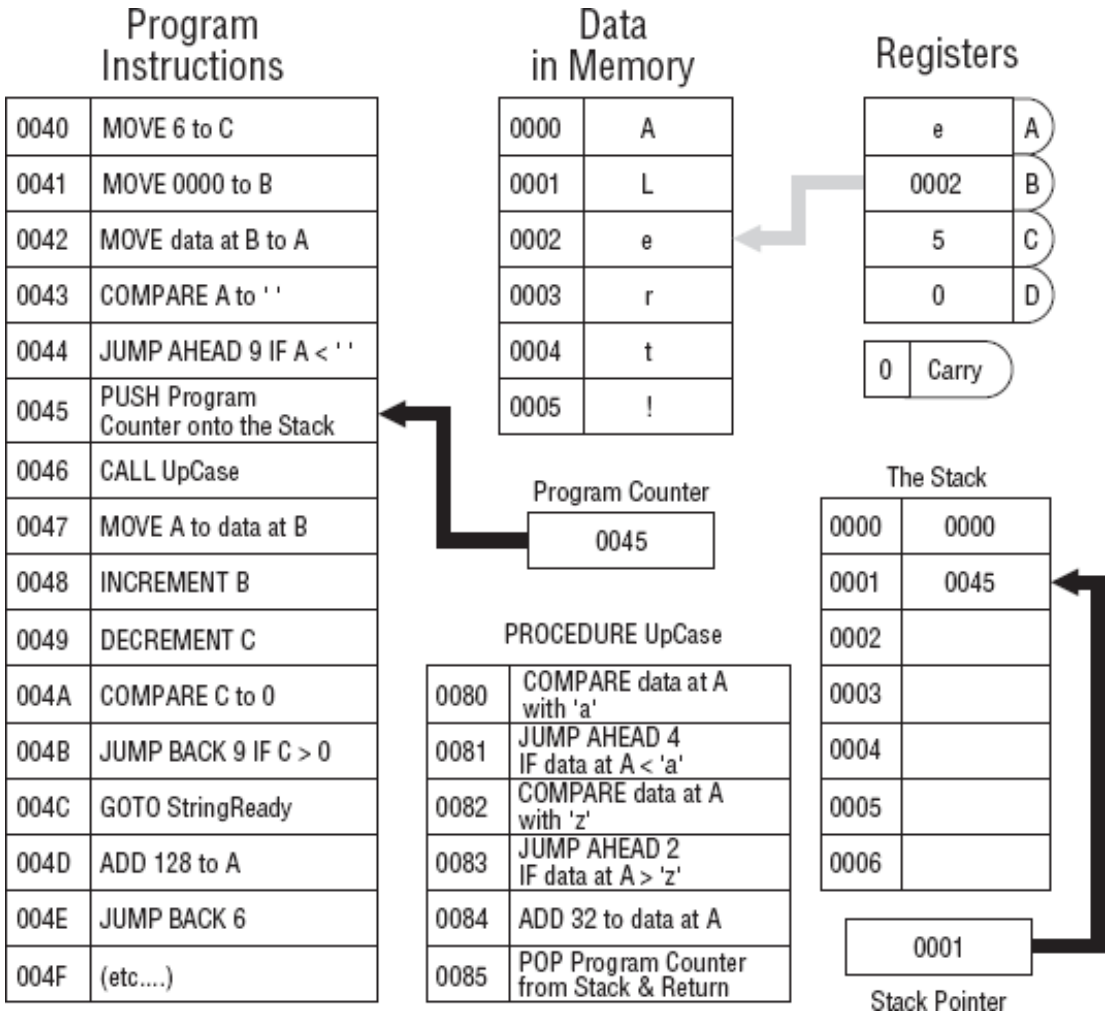


CPU Execution



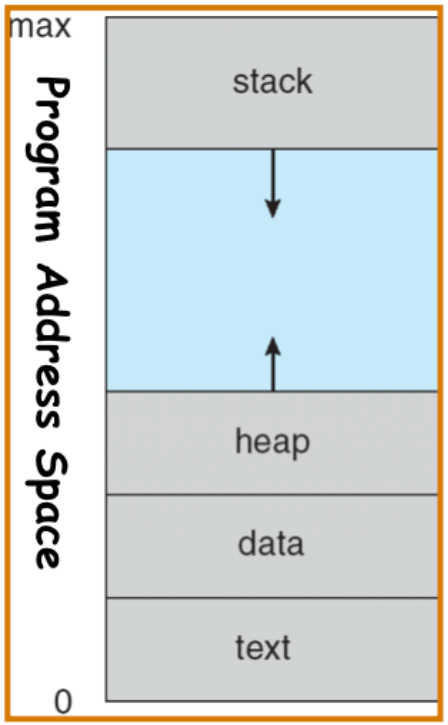
- Execution sequence:
  - Fetch Instruction at PC
  - Decode
  - Execute (possibly using registers)
  - Write results to registers/mem
  - PC = Next Instruction(PC)
  - Repeat

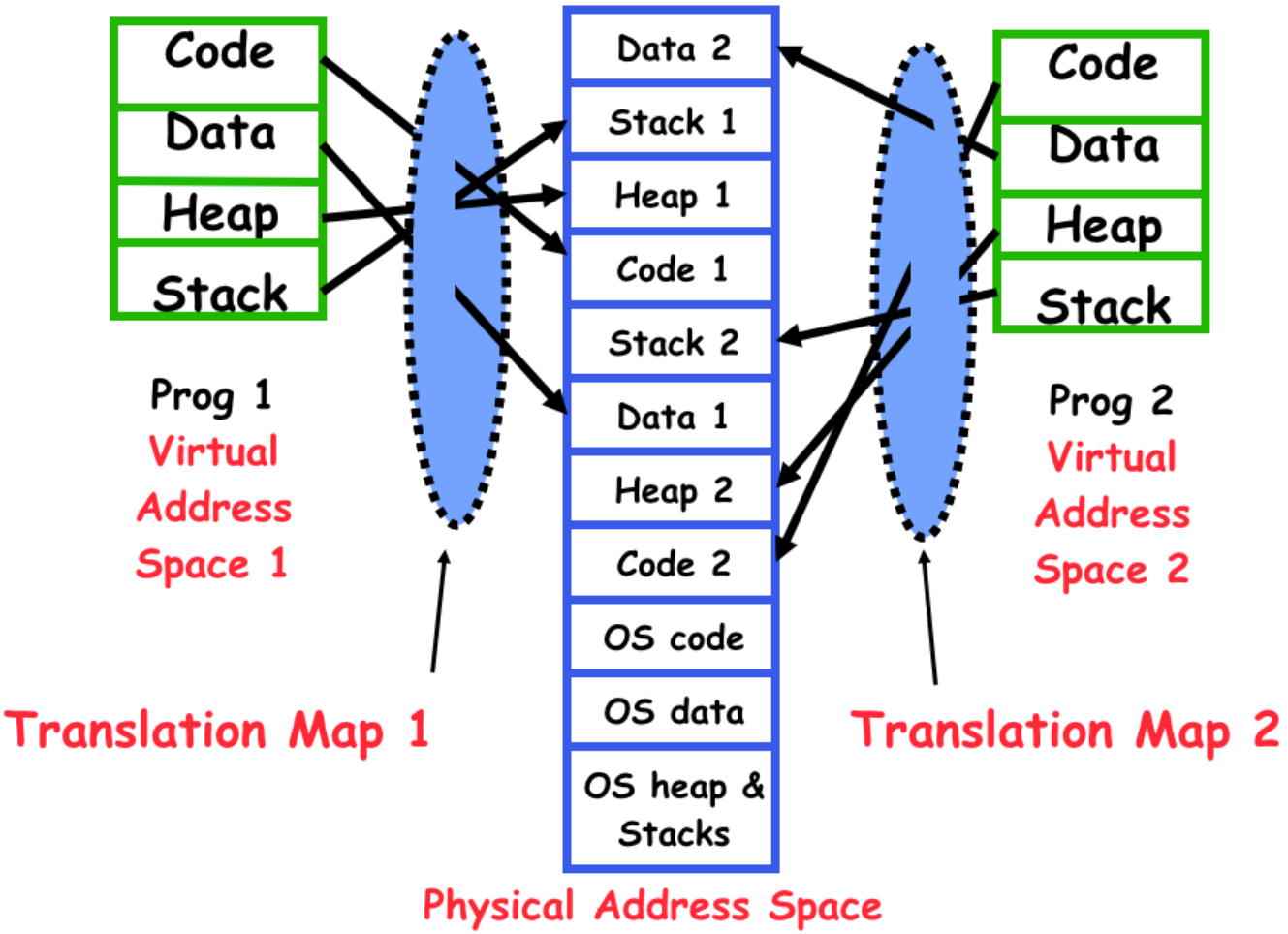
Execution Model



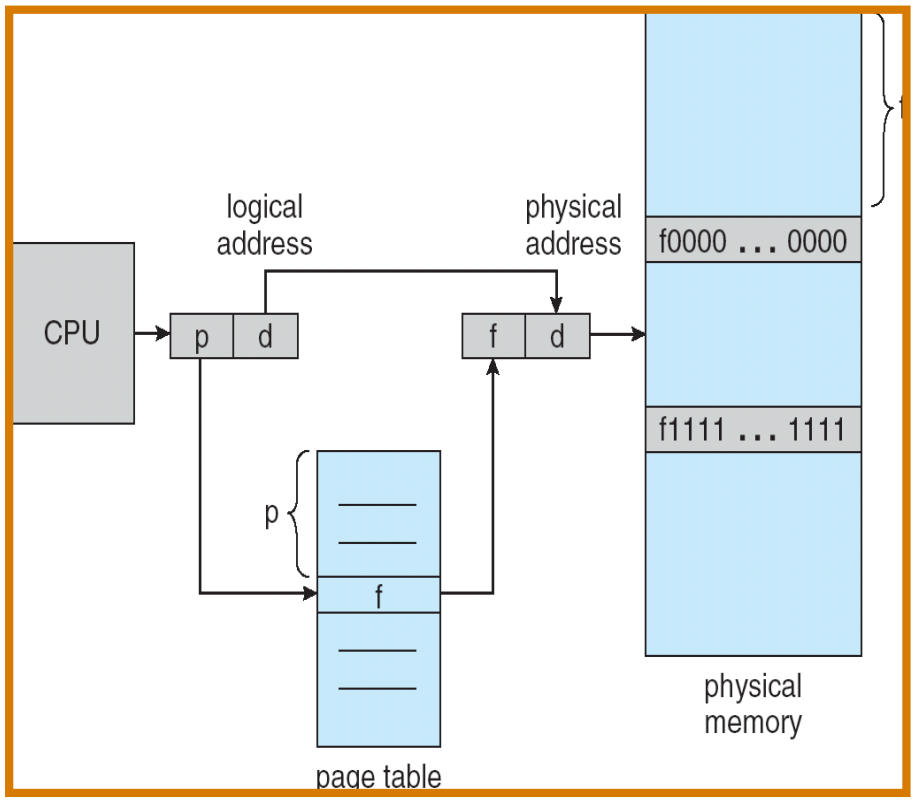
Source: Assembly Language Step-by-Step: Programming with Linux®, Third Edition, Jeff Duntemann

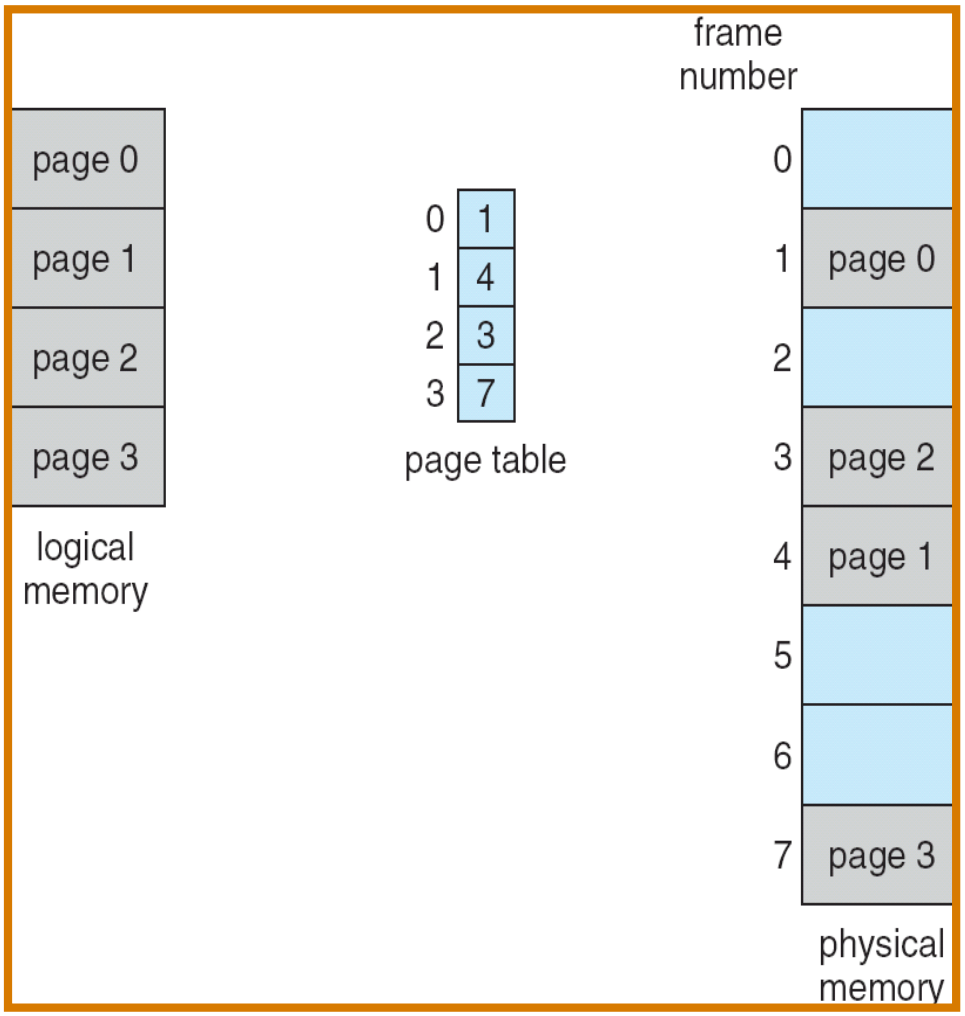
Address Space



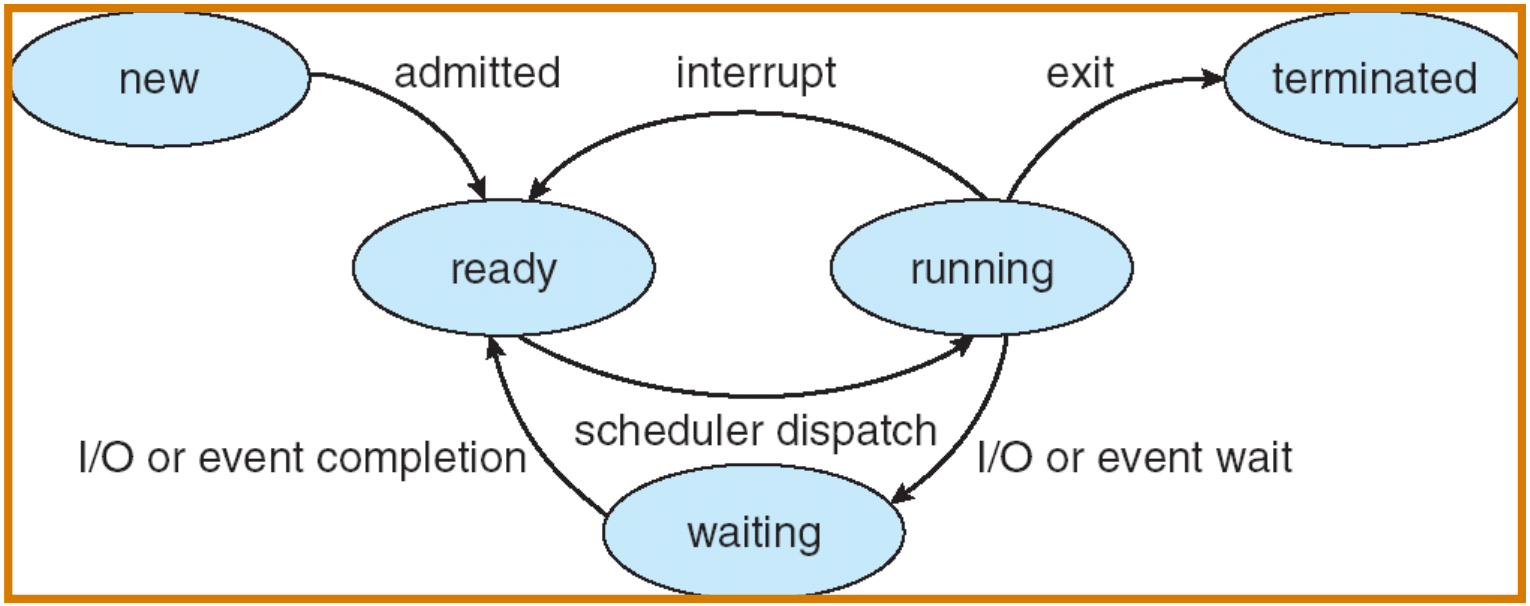


Logical vs. Physical Memory





Process States & Multiplex I/O



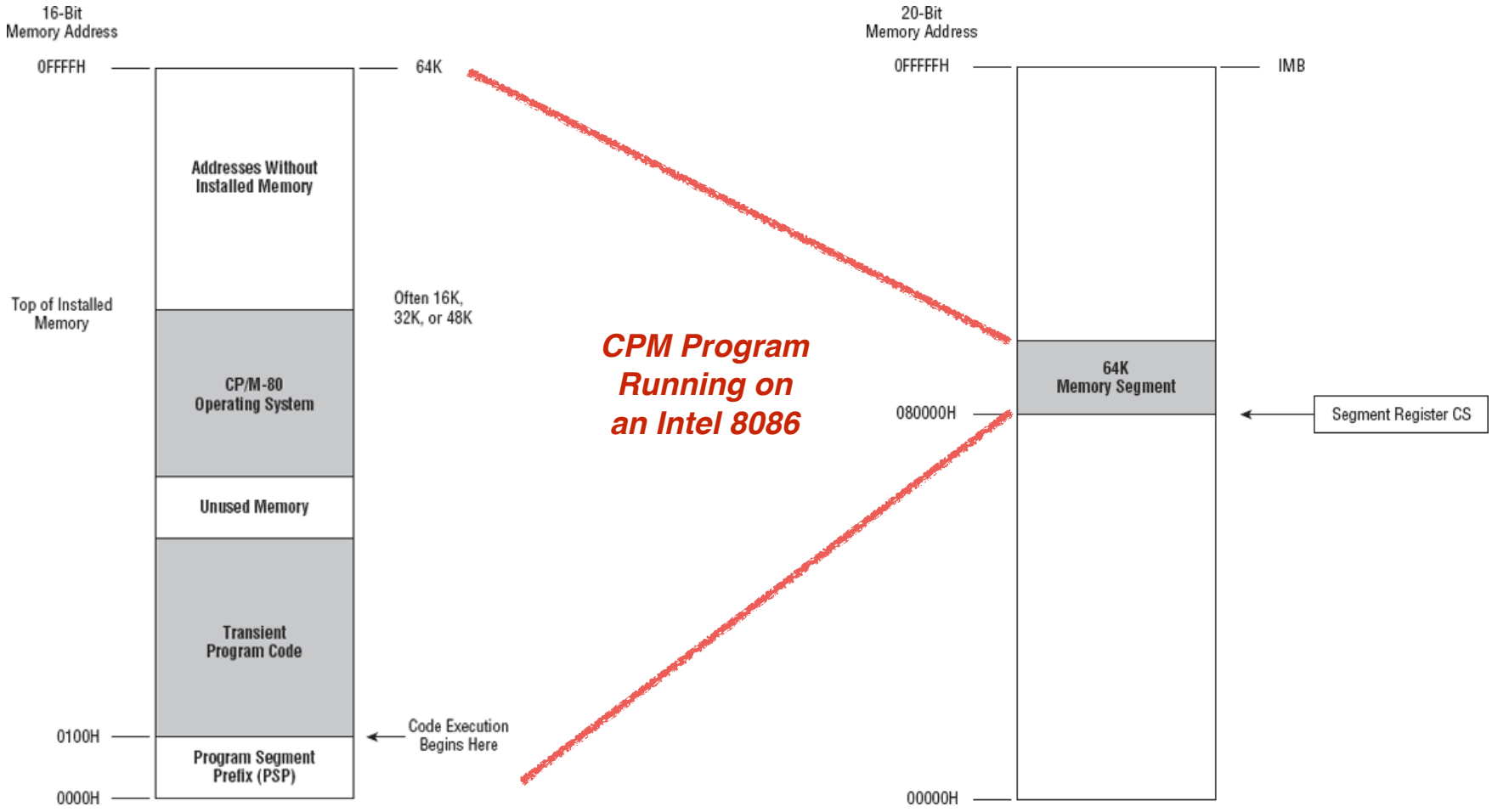
- **As a process executes, it changes state**
  - new: **The process is being created**
  - ready: **The process is waiting to run**
  - running: **Instructions are being executed**
  - waiting: **Process waiting for some event to occur**
  - terminated: **The process has finished execution**

x86 Architecture & Memory Models

A Brief History

Sources:
Assembly Language Step-by-Step: Programming with Linux®, Third Edition, Jeff Duntemann
http://en.wikibooks.org/wiki/X86_Assembly/X86_Architecture#Flat_Memory_Model

Intel 8080 & 8086



8080 - 16-bit address,
64K addressable Memory

8086 - 20-bit address,
1 MB addressable Memory

16-Bit Segment Registers

Segment registers are only used for holding "Segment Addresses"

Register	Usage	CPU's
CS	Code Segment	8088, 8086, 80286
DS	Data Segment	8088, 8086, 80286
SS	Stack Segment	8088, 8086, 80286
ES	Extra Segment	8088, 8086, 80286
FS	Extra Segment	80386 & later
GS	Extra Segment	80386 & later

Segment Registers [edit]

The 6 Segment Registers are:

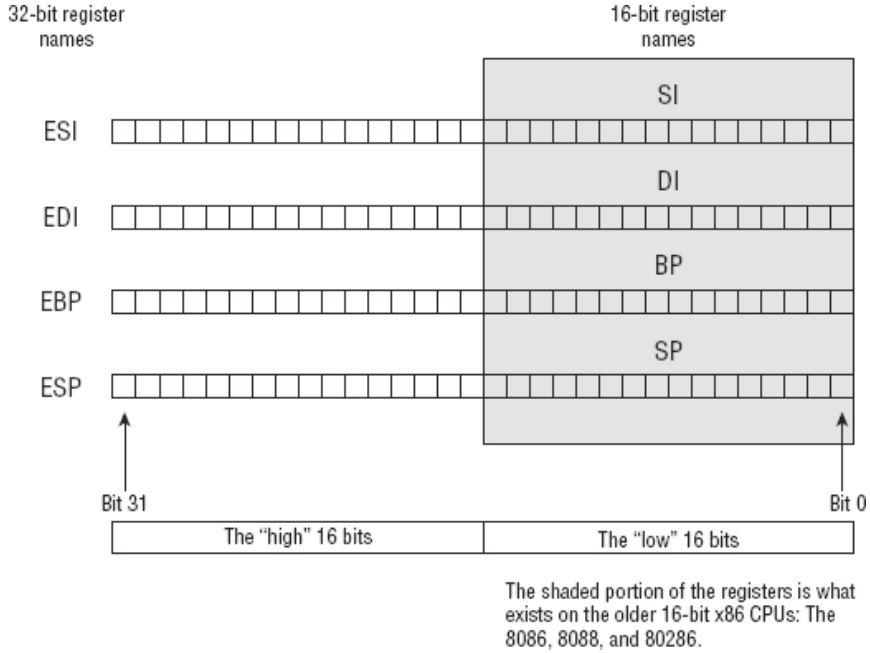
- Stack Segment (SS). Pointer to the stack.
- Code Segment (CS). Pointer to the code.
- Data Segment (DS). Pointer to the data.
- Extra Segment (ES). Pointer to extra data ('E' stands for 'Extra').
- F Segment (FS). Pointer to more extra data ('F' comes after 'E').
- G Segment (GS). Pointer to still more extra data ('G' comes after 'F').

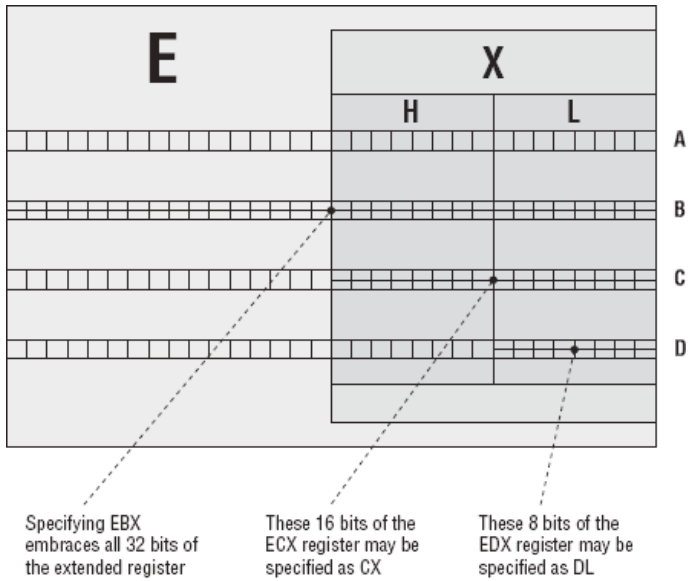
Most applications on most modern operating systems (like FreeBSD, Linux or Microsoft Windows) use a memory model that points nearly all segment registers to the same place (and uses paging instead), effectively disabling their use. Typically the use of FS or GS is an exception to this rule, instead being used to point at thread-specific data.

http://en.wikibooks.org/wiki/X86_Assembly/X86_Architecture#Flat_Memory_Model

General Purpose Registers

Registers	Notes	CPU's
AX, BX, CX, DX, BP, SI, DI, and SP	16-Bits	8088, 8086, 80286
EAX, EBX, ECX, EDX, EBP, ESI, EDI, and ESP	32-Bit Expansions	80386 & later





32-bit Registers, 16-bit Registers & 8-bit Register Halves

x86 Architecture [edit]

The x86 architecture has 8 General-Purpose Registers (GPR), 6 Segment Registers, 1 Flags Register and an Instruction Pointer. 64-bit x86 has additional registers.

Wikipedia has related information at ***Processor register***

General-Purpose Registers (GPR) - 32-bit naming conventions [edit]

The 8 GPRs are:

1. Accumulator register (AX). Used in arithmetic operations.
2. Base register (BX). Used as a pointer to data (located in segment register DS, when in segmented mode).
3. Counter register (CX). Used in shift/rotate instructions and loops.
4. Data register (DX). Used in arithmetic operations and I/O operations.
5. Stack Pointer register (SP). Pointer to the top of the stack.
6. Stack Base Pointer register (BP). Used to point to the base of the stack.
7. Source Index register (SI). Used as a pointer to a source in stream operations.
8. Destination Index register (DI). Used as a pointer to a destination in stream operations.

The order in which they are listed here is for a reason: it is the same order that is used in a push-to-stack operation, which will be covered later.

All registers can be accessed in 16-bit and 32-bit modes. In 16-bit mode, the register is identified by its two-letter abbreviation from the list above. In 32-bit mode, this two-letter abbreviation is prefixed with an 'E' (*extended*). For example, 'EAX' is the accumulator register as a 32-bit value.

Similarly, in the 64-bit version, the 'E' is replaced with an 'R', so the 64-bit version of 'EAX' is called 'RAX'.

It is also possible to address the first four registers (AX, CX, DX and BX) in their size of 16-bit as two 8-bit halves. The least significant byte (LSB), or low half, is identified by replacing the 'X' with an 'L'. The most significant byte (MSB), or high half, uses an 'H' instead. For example, CL is the LSB of the counter register, whereas CH is its MSB.

In total, this gives us five ways to access the accumulator, counter, data and base registers: 64-bit, 32-bit, 16-bit, 8-bit LSB, and 8-bit MSB. The other four are accessed in only three ways: 64-bit, 32-bit and 16-bit. The following table summarises this:

Register	Accumulator	Counter	Data	Base	Stack Pointer	Stack Base Pointer	Source	Destination
64-bit	RAX	RCX	RDX	RBX	RSP	RBP	RSI	RDI
32-bit	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
16-bit	AX	CX	DX	BX	SP	BP	SI	DI
8-bit	AH, AL	CH, CL	DH, DL	BH, BL				

http://en.wikibooks.org/wiki/X86_Assembly/X86_Architecture#Flat_Memory_Model

Special Registers

SP, BP, SI, and DI ***don't have*** "8-bit register halves"

IP (Instruction Pointer) only does "one thing"! It holds the address of the next instruction for CPU execution.

EFLAGS. The Flags Register used by CPU instructions to test for certain conditions.

EFLAGS Register [edit]

The EFLAGS is a 32-bit register used as a collection of bits representing Boolean values to store the results of operations and the state of the processor.

The names of these bits are:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0	0	0	0	0	0	0	0	0	0	ID	VIP	VIF	AC	VM	RF

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	NT	IOPL		OF	DF	IF	TF	SF	ZF	0	AF	0	PF	1	CF

The bits named 0 and 1 are reserved bits and shouldn't be modified.

The different use of these flags are:

0. CF : Carry Flag. Set if the last arithmetic operation carried (addition) or borrowed (subtraction) a bit beyond the size of the register. This is then checked when the operation is followed with an add-with-carry or subtract-with-borrow to deal with values too large for just one register to contain.
2. PF : Parity Flag. Set if the number of set bits in the least significant byte is a multiple of 2.
4. AF : Adjust Flag. Carry of Binary Code Decimal (BCD) numbers arithmetic operations.
6. ZF : Zero Flag. Set if the result of an operation is Zero (0).
7. SF : Sign Flag. Set if the result of an operation is negative.
8. TF : Trap Flag. Set if step by step debugging.
9. IF : Interruption Flag. Set if interrupts are enabled.
10. DF : Direction Flag. Stream direction. If set, string operations will decrement their pointer rather than incrementing it, reading memory backwards.
11. OF : Overflow Flag. Set if signed arithmetic operations result in a value too large for the register to contain.

12-13. IOPL : I/O Privilege Level field (2 bits). I/O Privilege Level of the current process.

14. NT : Nested Task flag. Controls chaining of interrupts. Set if the current process is linked to the next process.
16. RF : Resume Flag. Response to debug exceptions.
17. VM : Virtual-8086 Mode. Set if in 8086 compatibility mode.
18. AC : Alignment Check. Set if alignment checking of memory references is done.
19. VIF : Virtual Interrupt Flag. Virtual image of IF.
20. VIP : Virtual Interrupt Pending flag. Set if an interrupt is pending.
21. ID : Identification Flag. Support for CPUID instruction if can be set.

http://en.wikibooks.org/wiki/X86_Assembly/X86_Architecture#Flat_Memory_Model

Instruction Pointer [edit]

The EIP register contains the address of the **next** instruction to be executed if no branching is done.

EIP can only be read through the stack after a `call` instruction.

Memory [edit]

The x86 architecture is little-endian, meaning that multi-byte values are written least significant byte first. (This refers only to the ordering of the bytes, not to the bits.)

So the 32 bit value $B3B2B1B0_{16}$ on an x86 would be represented in memory as:

Little endian representation			
B0	B1	B2	B3

For example, the 32 bits double word 0x1BA583D4 (the **0x** denotes hexadecimal) would be written in memory as:

Little endian example			
D4	83	A5	1B

This will be seen as `0xD4 0x83 0xA5 0x1B` when doing a memory dump.

Two's Complement Representation [edit]

Two's complement is the standard way of representing negative integers in binary. The sign is changed by inverting all of the bits and adding one.

Two's complement example	
Start:	0001
Invert:	1110
Add One:	1111

0001 represents decimal 1

1111 represents decimal -1

http://en.wikibooks.org/wiki/X86_Assembly/X86_Architecture#Flat_Memory_Model

Addressing Modes

Addressing modes [edit]

The addressing mode indicates the manner in which the operand is presented.

Register Addressing

(operand address R is in the address field)

```
mov ax, bx  ; moves contents of register bx into ax
```

Immediate

(actual value is in the field)

```
mov ax, 1   ; moves value of 1 into register ax
```

or

```
mov ax, 010Ch ; moves value of 0x010C into register ax
```

Direct memory addressing

(operand address is in the address field)

```
.data
my_var dw 0abcdh ; my_var = 0xabcd
.code
mov ax, my_var ; copy my_var content in ax (ax=0xabcd)
mov ax, [my_var] ; idem as above
```

http://en.wikibooks.org/wiki/X86_Assembly/X86_Architecture#Flat_Memory_Model

Addressing Modes (cont.)

Direct offset addressing

(uses arithmetics to modify address)

```
byte_tbl db 12,15,16,22,..... ; Table of bytes
mov al,[byte_tbl+2]
mov al,byte_tbl[2] ; same as the former
```

Register Indirect

(field points to a register that contains the operand address)

```
mov ax,[di]
```

The registers used for indirect addressing are BX, BP, SI, DI

Base-index

```
mov ax,[bx + di]
```

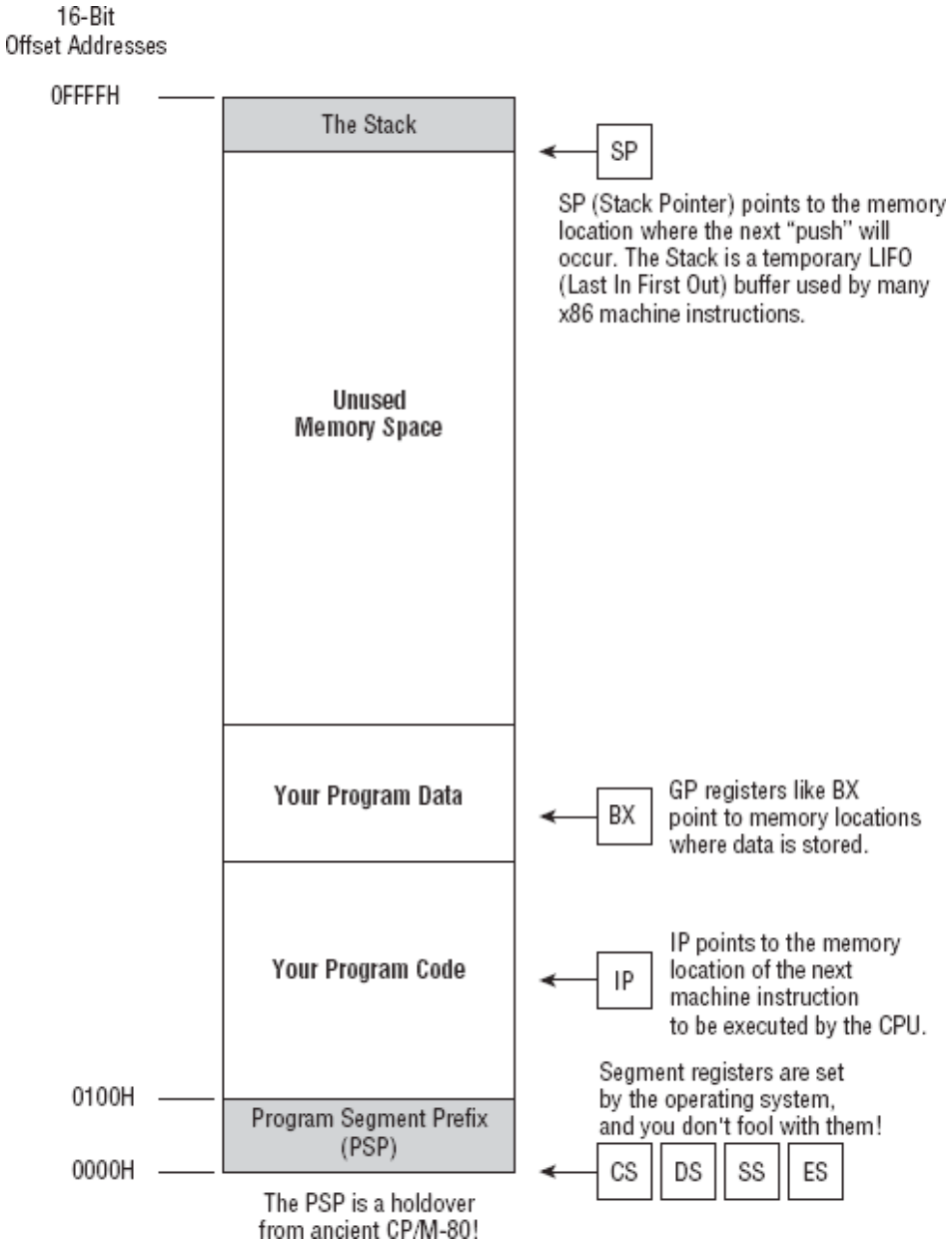
For example, if we are talking about an array, BX contains the address of the beginning of the array, and DI contains the index into the array.

Base-index with displacement

```
mov ax,[bx + di + 10]
```

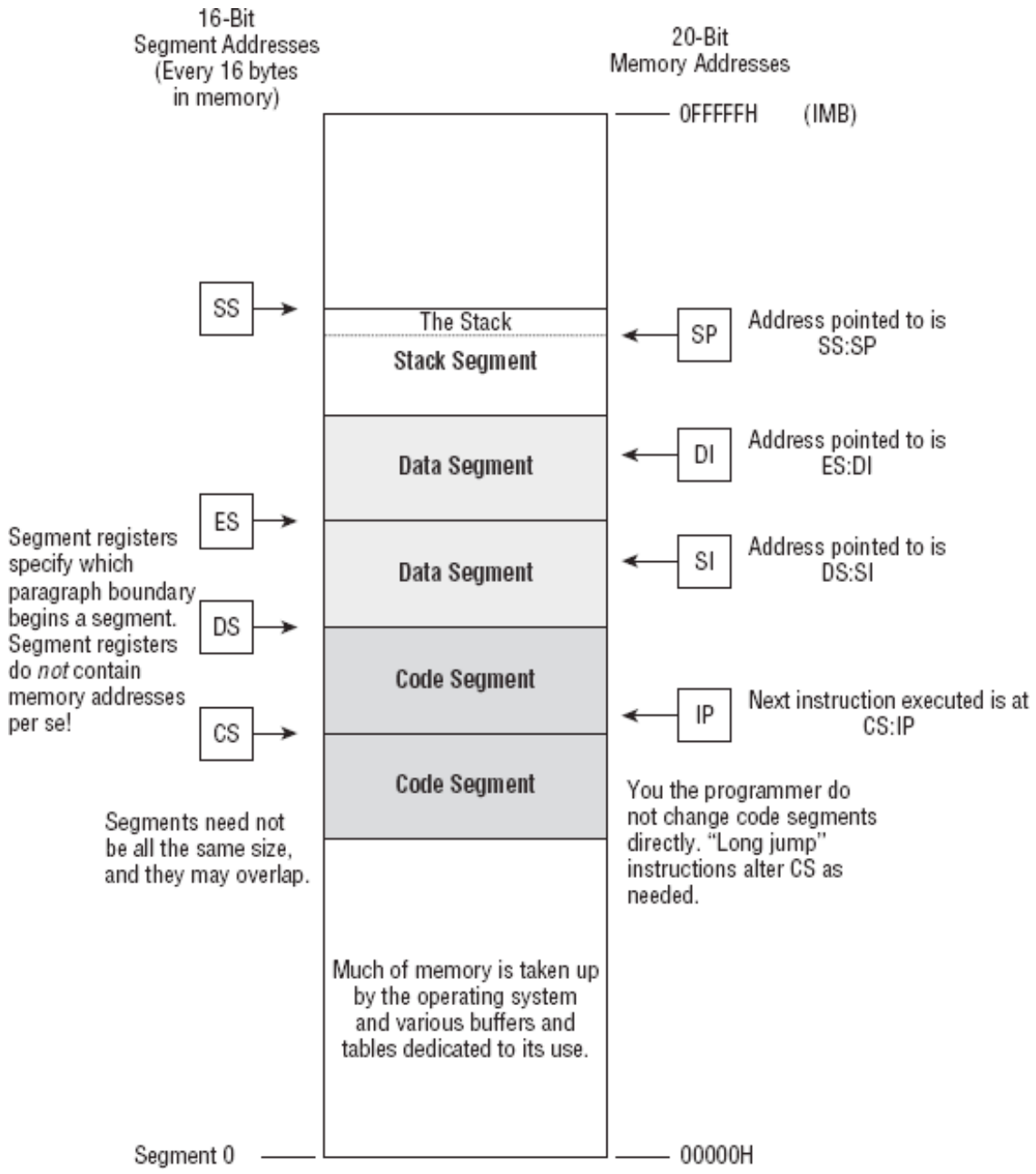
http://en.wikibooks.org/wiki/X86_Assembly/X86_Architecture#Flat_Memory_Model

Real Mode Flat Model



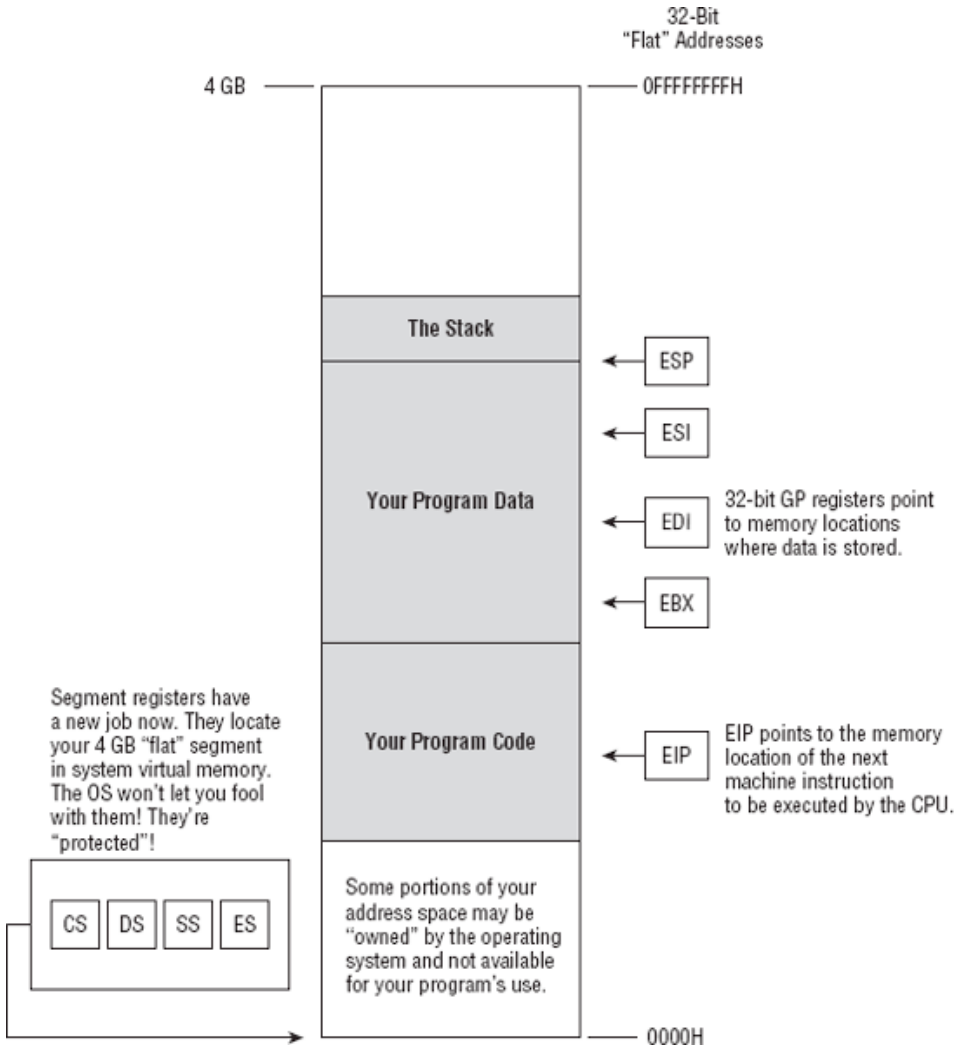
- *1 MB Addressable*
- *Program can use just 64K (16-bit address)*
- *Or, full 1MB via 20-bit address (segment:offset)*

Real Mode Segmented Model



- *1 MB Addressable*
- *Program use full 1MB via*
- *Programs share memory with OS!*

Protected Mode Flat Model Since x386



- *4 GB Addressable*
- *Programs have the "illusion" of full access to 4GB.*
- *OS does this trick using Virtual Memory!*

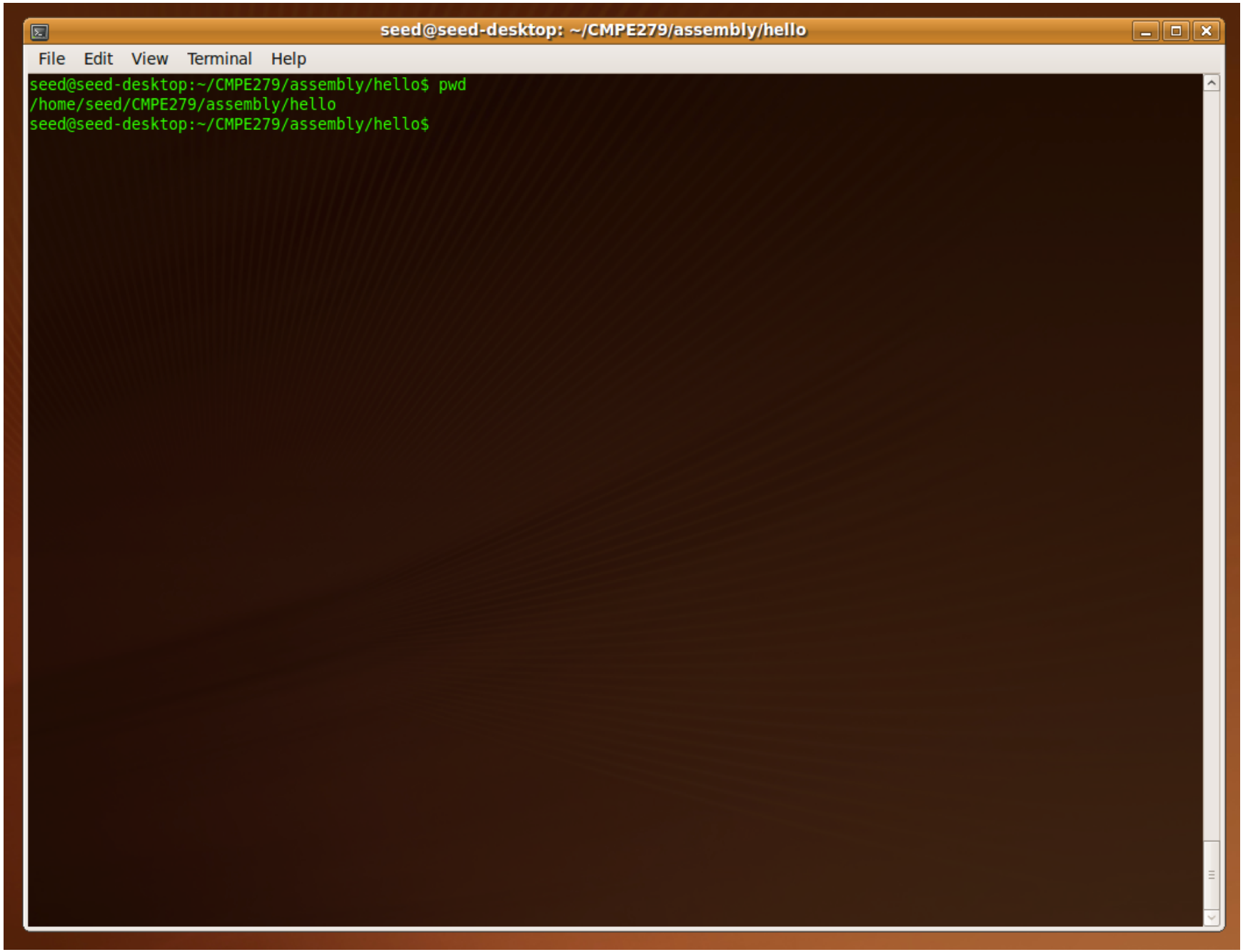
64-Bit "Long Mode" Model
x86-64

- *CPU using 64-bit addresses. Registers are 64-bits.*
- *IA-32 Registers extended to 64-Bits with "R".*
  For example, EAX becomes RAX.
- *64-bit addresses can address: **16 exabytes**.*
- *An exabyte is 260 bytes, billion gigabytes.*

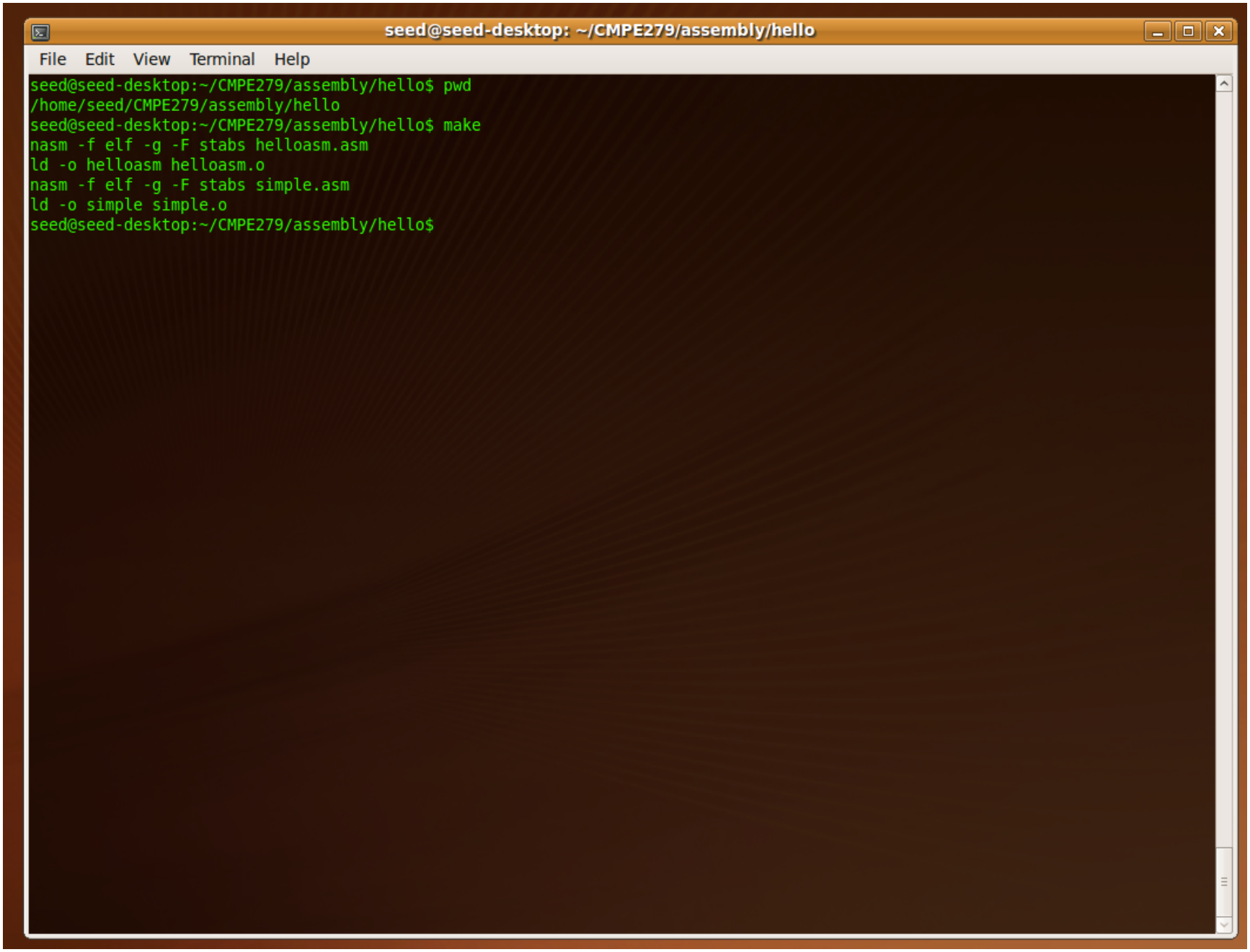
Short Break!

Using KDbg & Insight Debuggers

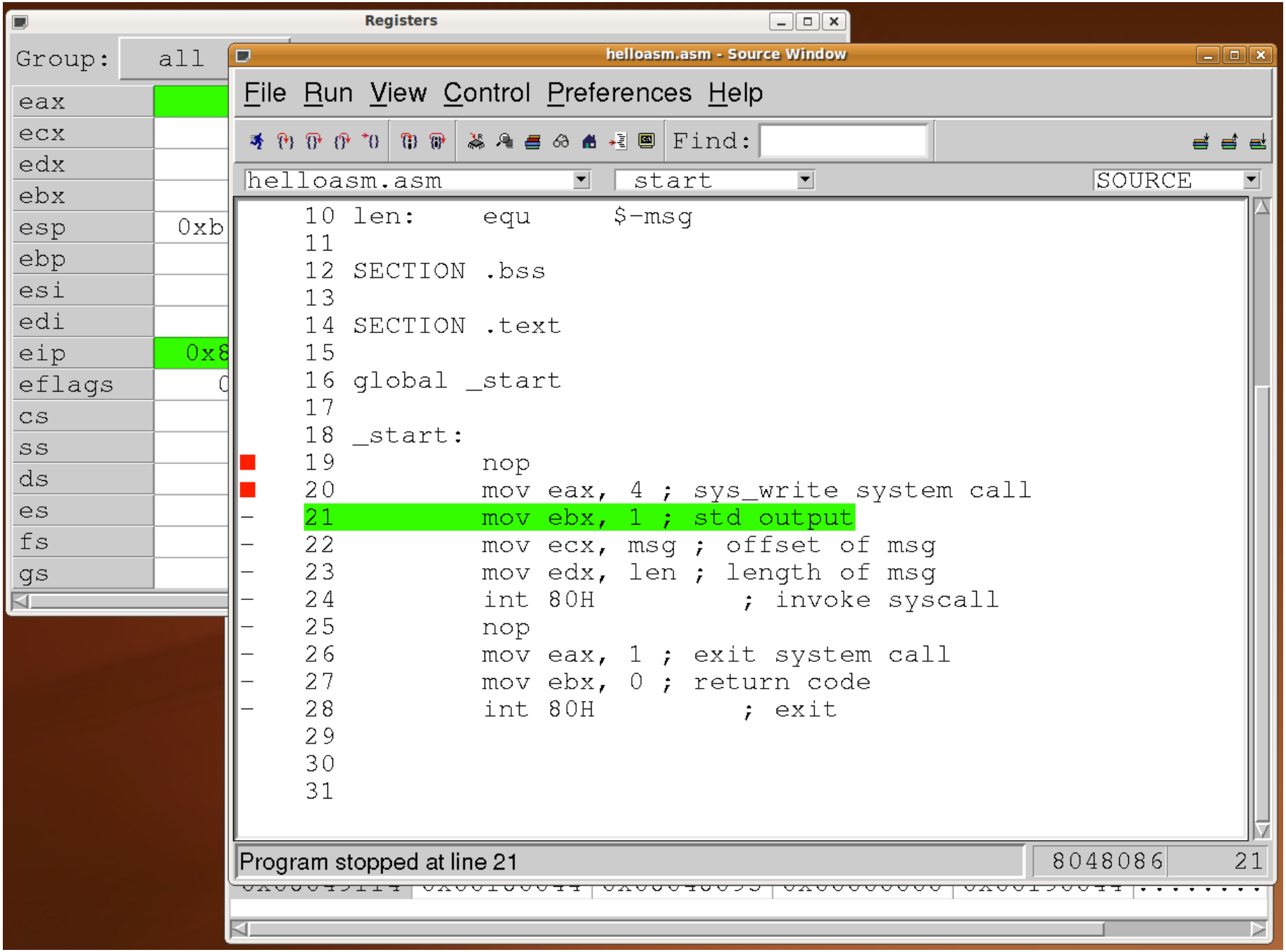
In SeedLab VM, Go to Folder ~/CMPE279/assembly/hello



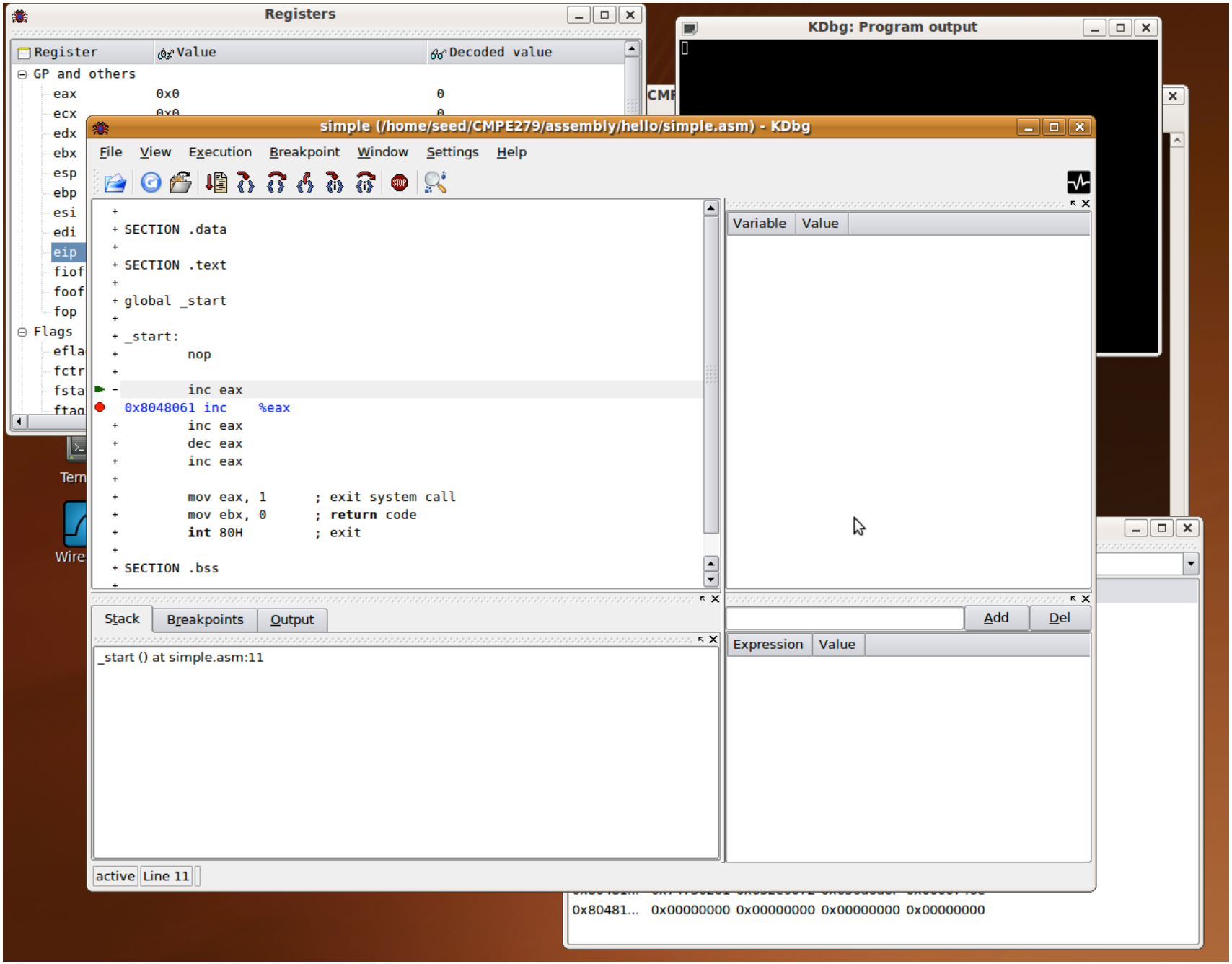
Run “make” to build programs



Run: insight helloasm



Run: kdbg simple



Op Codes

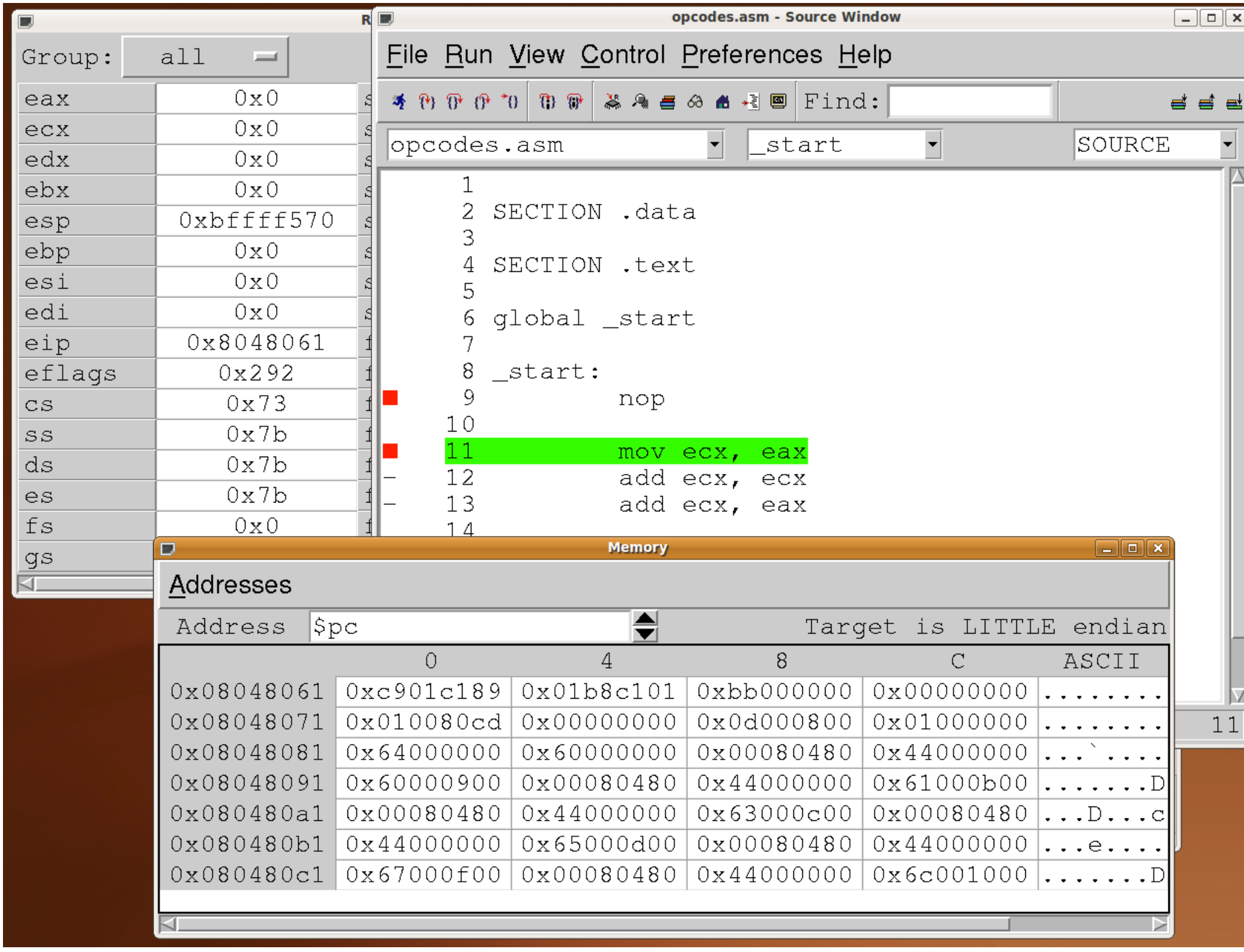
http://ref.x86asm.net/coder32.html

Build the “**opcodes.asm**” program and use Insight to debug it. Use the “test.sh” script to build it. What does test.sh do?

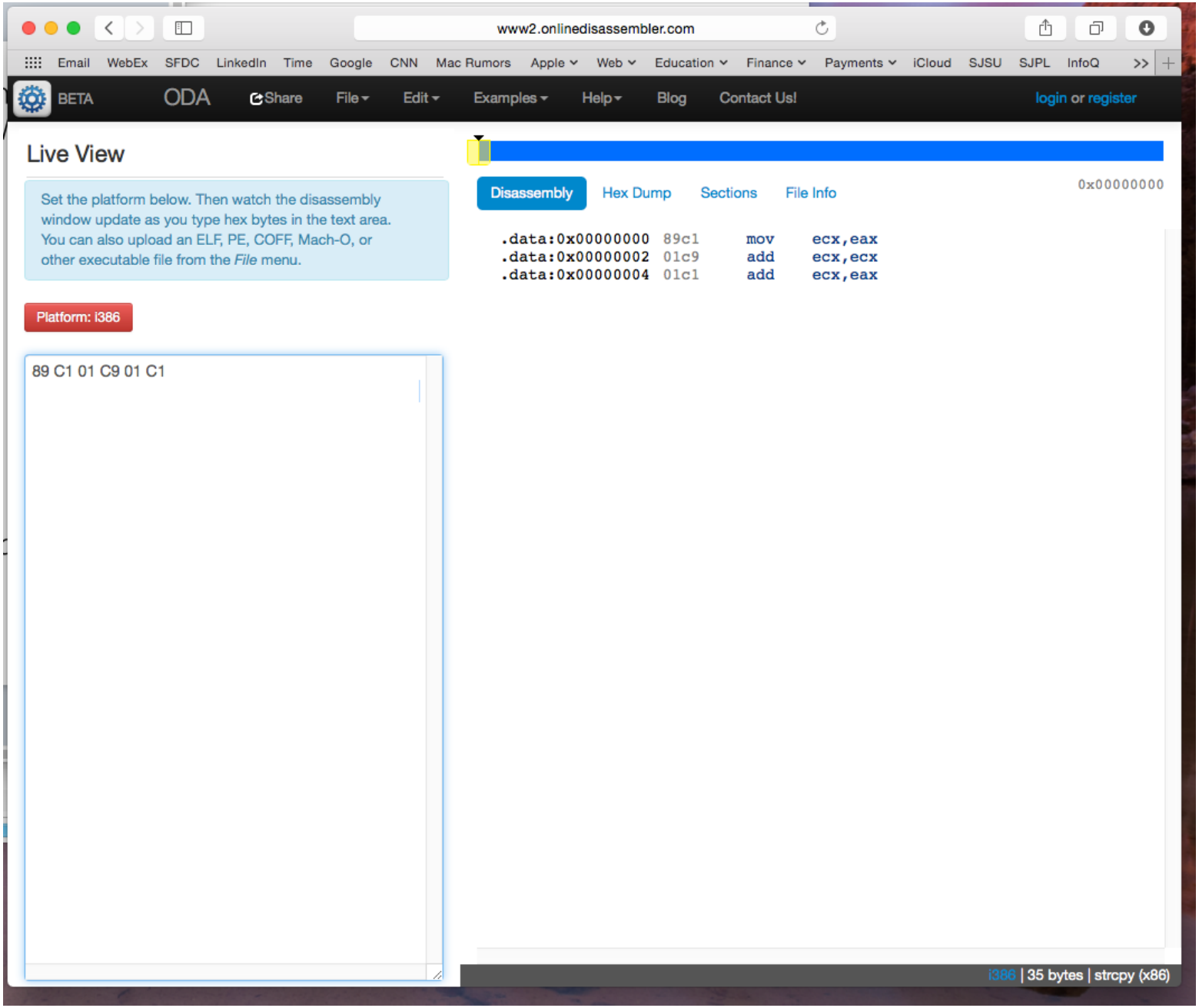
The Opcodes are:
(in Base 16 Hex Digits)

89 C1 // mov ecx, eax
01 C9 // add ecx, ecx
01 C1 // add ecx, eax

How are these stored in memory? Was it what you expected?


Group: all
eax 0x0
ecx 0x0
edx 0x0
ebx 0x0
esp 0xbffff570
ebp 0x0
esi 0x0
edi 0x0
eip 0x8048061
eflags 0x292
cs 0x73
ss 0x7b
ds 0x7b
es 0x7b
fs 0x0
gs
opcodes.asm - Source Window
File Run View Control Preferences Help
Find:
opcodes.asm
_start
SOURCE
1
2 SECTION .data
3
4 SECTION .text
5
6 global _start
7
8 _start:
9 nop
10
11 mov ecx, eax
12 add ecx, ecx
13 add ecx, eax
14
11
Memory
Addresses
Address $pc
Target is LITTLE endian
0 4 8 C ASCII
0x08048061 0xc901c189 0x01b8c101 0xbb000000 0x00000000 ....
0x08048071 0x010080cd 0x00000000 0x0d000800 0x01000000 ....
0x08048081 0x64000000 0x60000000 0x00080480 0x44000000 ...`....
0x08048091 0x60000900 0x00080480 0x44000000 0x61000b00 ....D
0x080480a1 0x00080480 0x44000000 0x63000c00 0x00080480 ...D...c
0x080480b1 0x44000000 0x65000d00 0x00080480 0x44000000 ...e....
0x080480c1 0x67000f00 0x00080480 0x44000000 0x6c001000 ....D

http://www.onlinedisassembler.com/odaweb/



MOV

http://en.wikibooks.org/wiki/X86_Assembly

Build the “**mov1.asm**” program and use Insight to debug it.

Single step the execution for the instructions between the “noop’s” and observe changes to “**eax**”, “**ecx**” and “**edx**” with each step.

Explain the each change.

Instruction	eax	ecx	edx
	0	0	0
mov eax, 3h	0x3	0	0
mov edx, 44h	0x3	0	0x44
mov edx, edx	0x3	0	0x44
mov ecx, edx	0x3	0x44	0x44
mov edx, eax	0x3	0x44	0x3

Build the “**mov2.asm**” program and use Insight to debug it.

Single step the execution the instructions between the “noop’s” and observe changes to “**eax**” and “**ecx**” with each step.

Explain the each change.

Instruction	eax	ecx
	0	0
mov ax, 9cH	0x9c	0
mov eax, 99991234H	0x99991234	0
mov cl, 22H	0x99991234	0x22
mov ah, cl	0x99992234	0x22

ADD

http://en.wikibooks.org/wiki/X86_Assembly

Build the “**add1.asm**” program and use Insight to debug it.

Single step the execution for the instructions between the “noop’s” and observe changes to “**esi**”, “**eax**” and “**ebx**” with each step.

Explain the each change.

Instruction	esi	eax	ebx
	0	0	0
add eax, ex	0	0	0
add eax, eax	0	0	0
mov esi, 0FFFFFFFFH	0xFFFFFFFF	0	0
add ebx, esi	0xFFFFFFFF	0	0xFFFFFFFF
add esi, eax	0xFFFFFFFF	0	0xFFFFFFFF

Did you observe anything else that changed?

Build the “**add2.asm**” program and use Insight to debug it.

Single step the execution for the instructions between the “noop’s” and observe changes to “**edi**”, “**eax**” and “**ecx**” with each step.

Explain the each change. Also, note the Decimal values in each of the three registers.

Instruction	eax	ecx	edi
	0	0	0
add al, ch	0	0	0
add di, cx	0	0	0
mov edi, 0AB29FFFH	0	0	0xAB29FFFF
add edi, ecx	0	0	0xAB29FFFF

Did you observe anything else that changed?

SUB, INC, DEC, MUL, DIV

http://en.wikibooks.org/wiki/X86_Assembly

Single step the execution of

- ***incdec.asm***
- ***sub.asm***
- ***div1.asm***
- ***div2.asm***

…and observe changes to as noted in the “comments” in the file with each step.

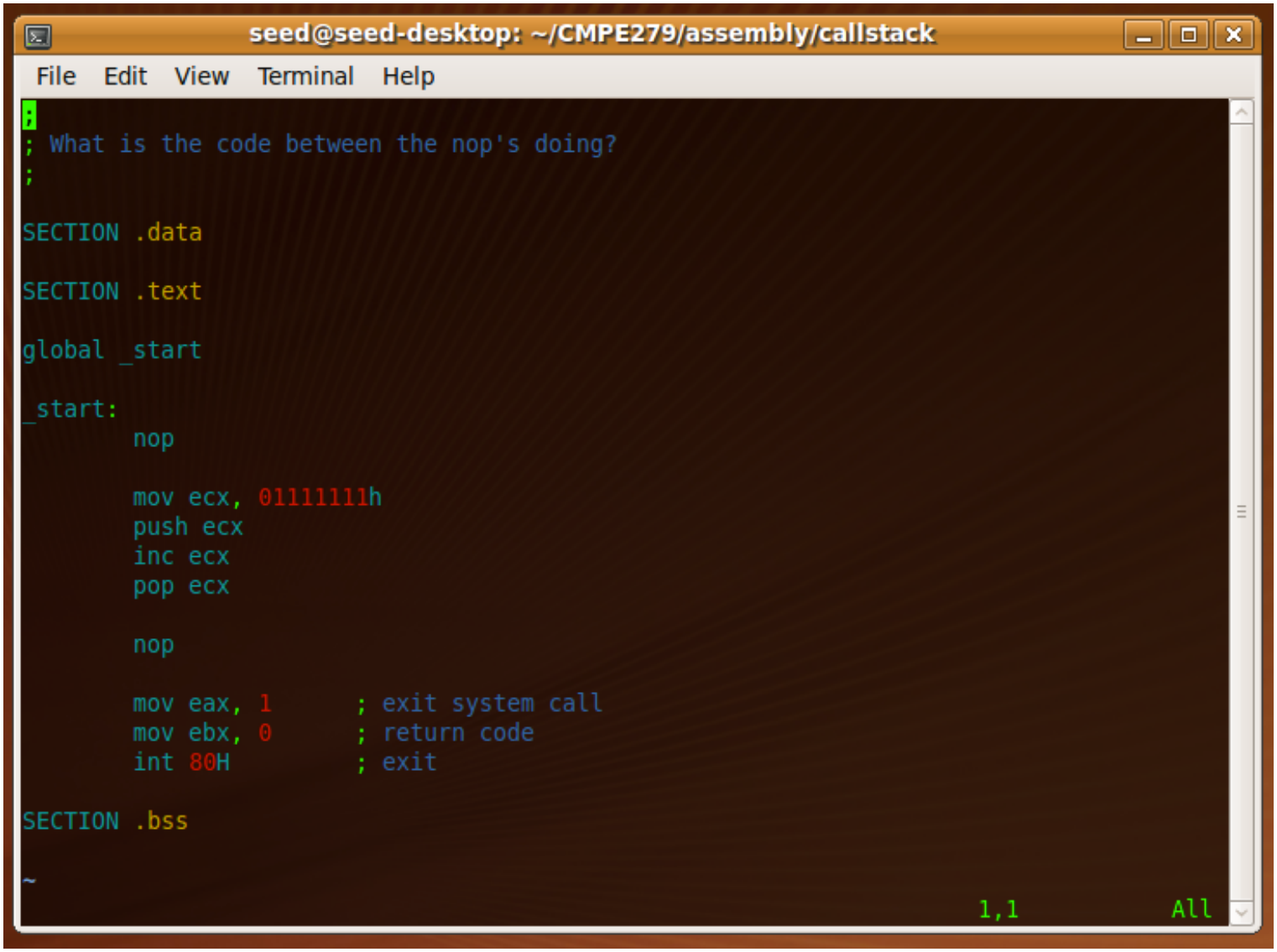
Explain the each change.

Instruction			

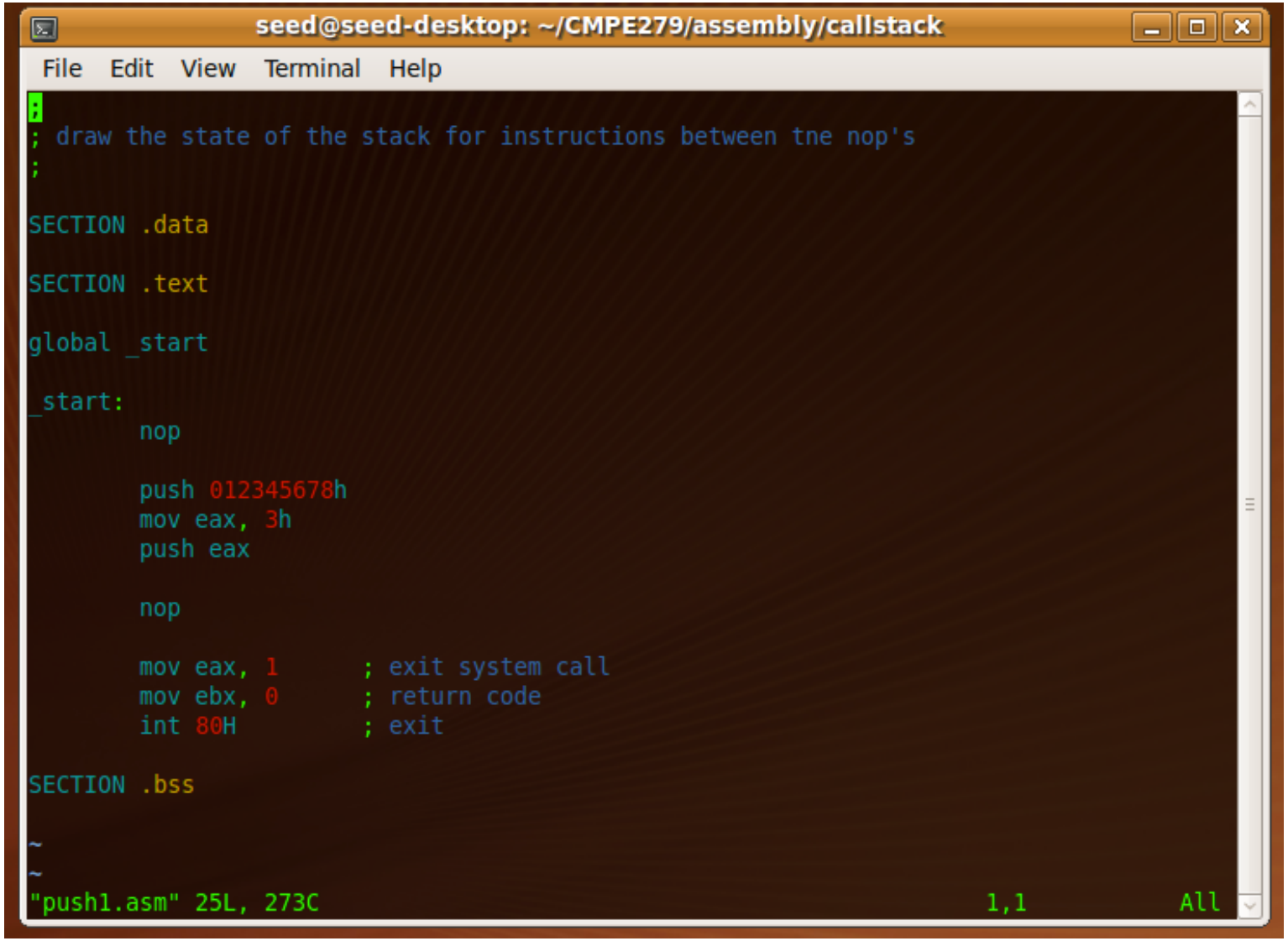
Functions & Call Stack

http://en.wikibooks.org/wiki/X86_Assembly

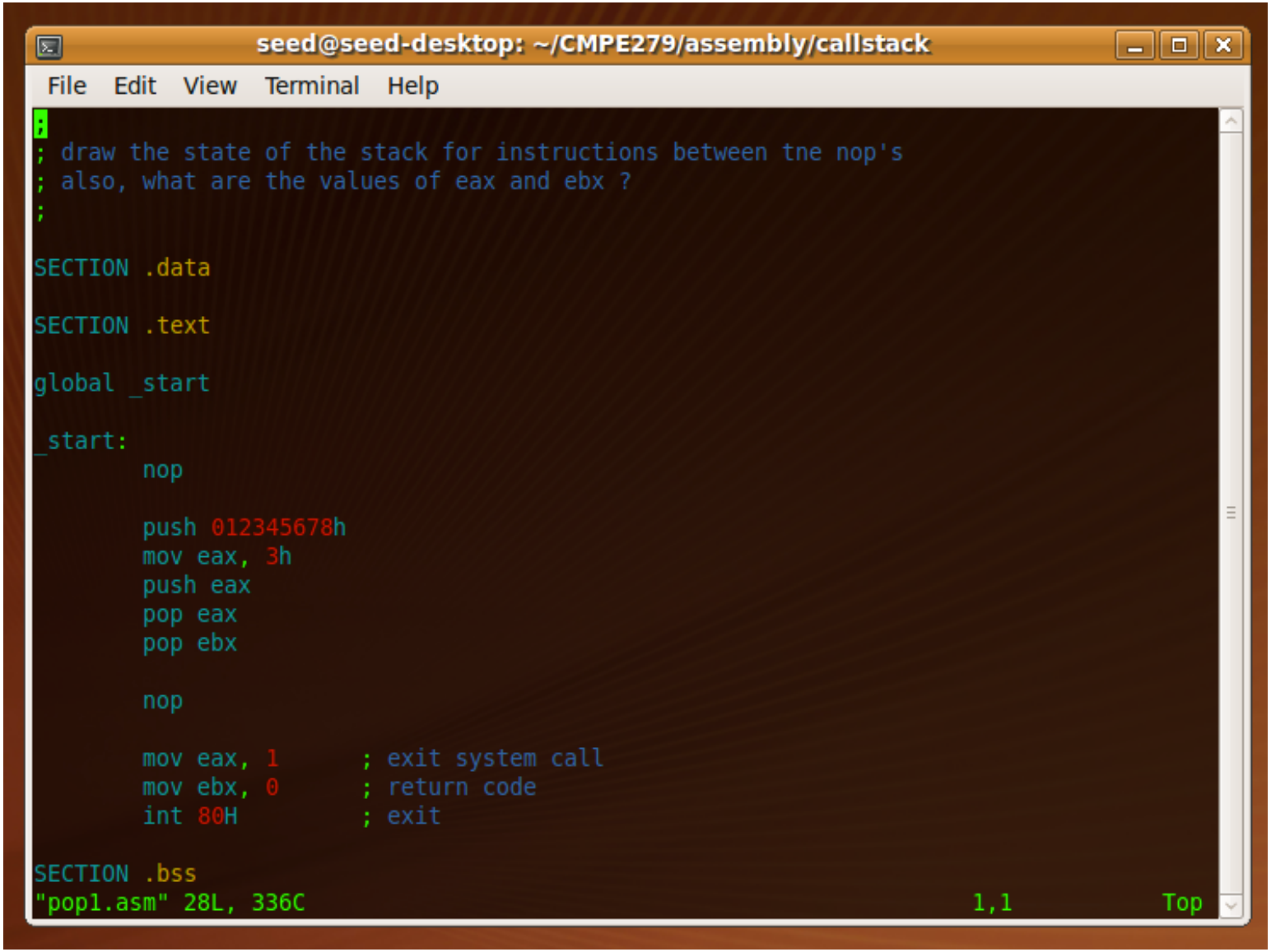
what.asm


seed@seed-desktop: ~/CMPE279/assembly/callstack
File Edit View Terminal Help
;
; What is the code between the nop's doing?
;
SECTION .data
SECTION .text
global _start
_start:
nop
mov ecx, 01111111h
push ecx
inc ecx
pop ecx
nop
mov eax, 1 ; exit system call
mov ebx, 0 ; return code
int 80H ; exit
SECTION .bss
~
1,1 All

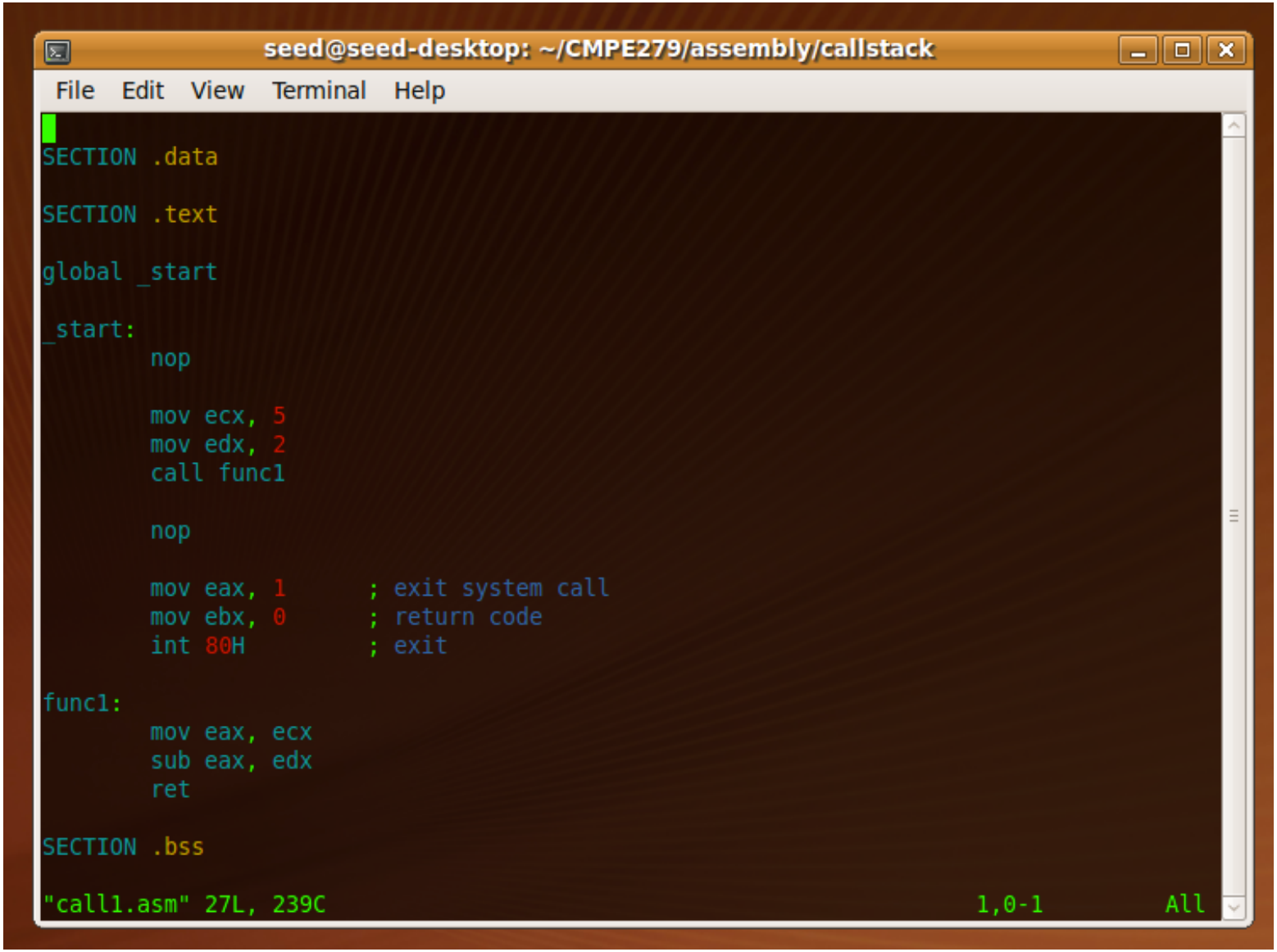
push1.asm - debug & note changes to registers, stack, memory



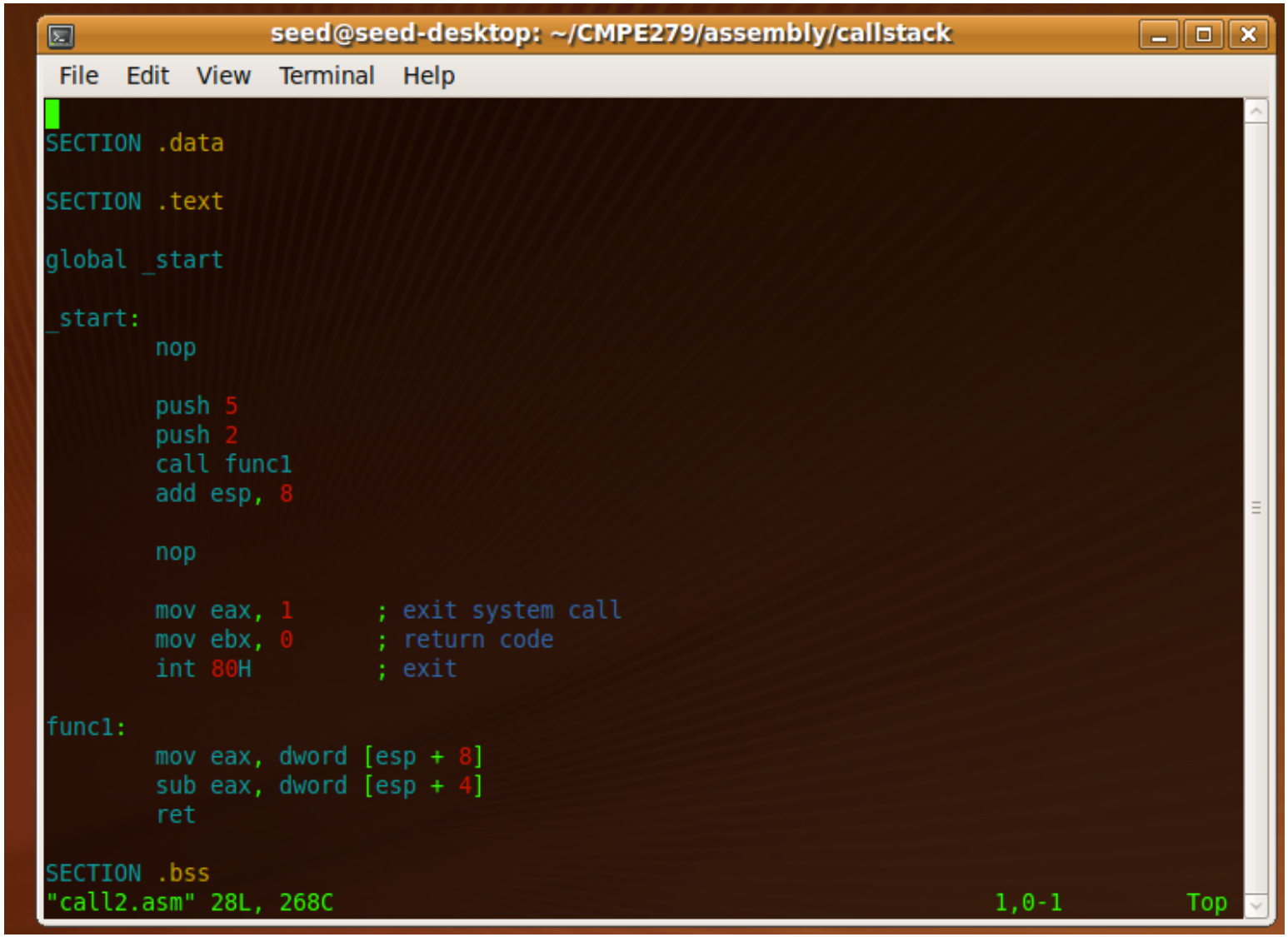
pop1.asm - debug & note changes to registers, stack, memory



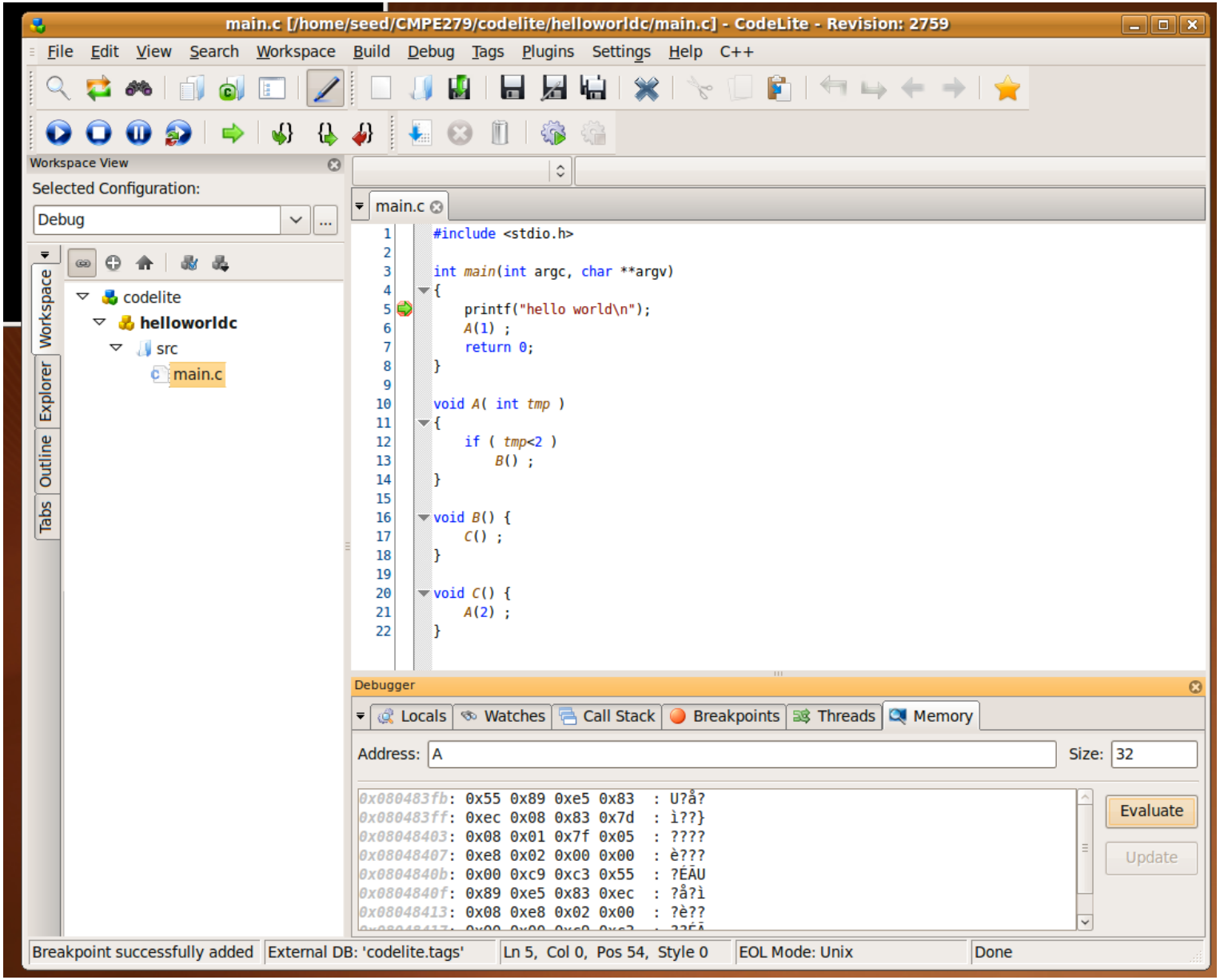
call1.asm - debug & note changes to registers, stack, memory



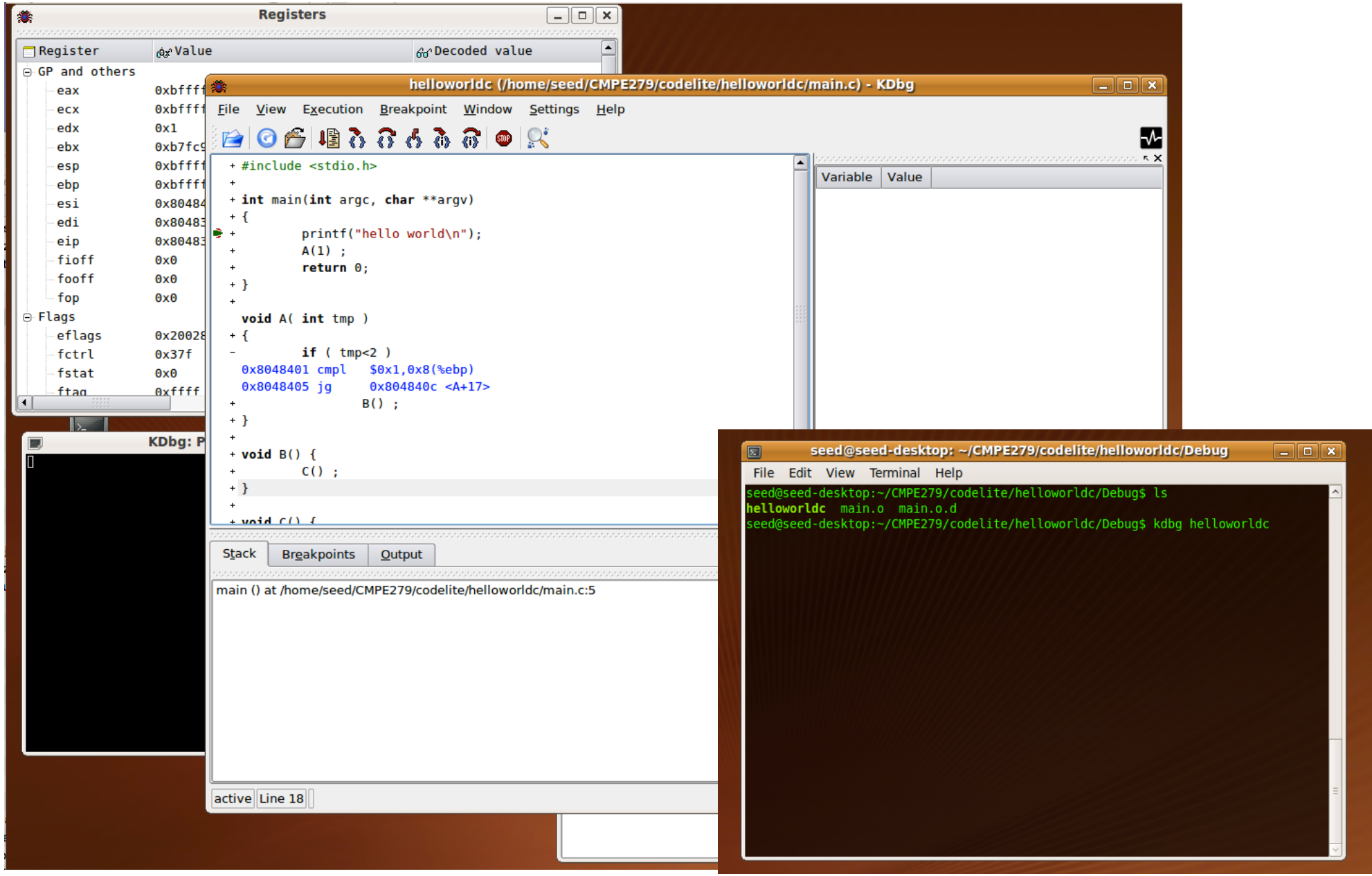
call2.asm - debug & note changes to registers, stack, memory



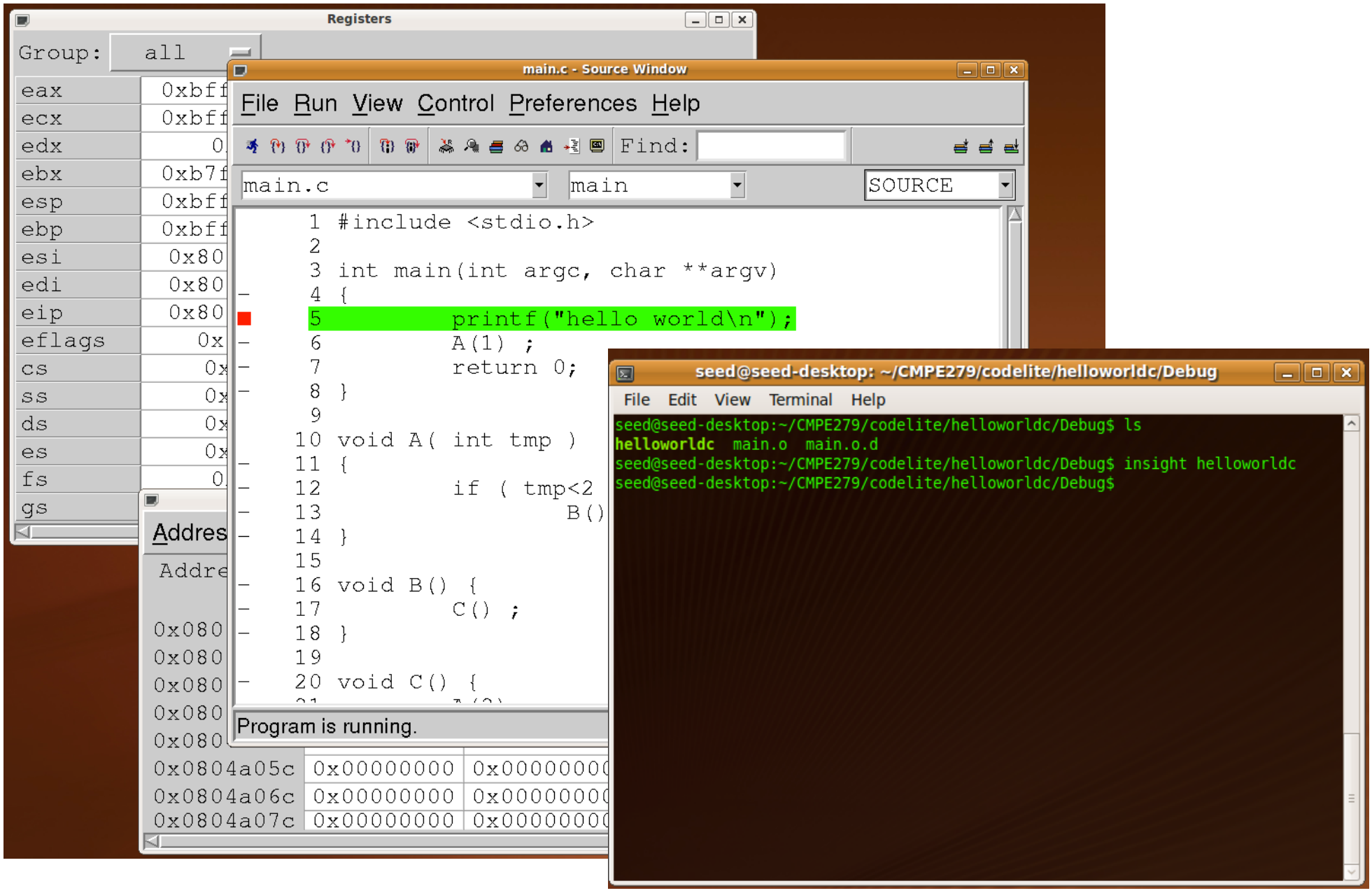
Hello World "C" Program - Codelite just single step thru the program.



Hello World "C" Program - Kdbg
debug & note changes to registers, stack, memory



Hello World "C" Program - Insight debug & note changes to registers, stack, memory



Useful Lookup Tables

http://www.pcguide.com/res/tablesPowers-c.html

n	2^n	CommonStorage Designation (Bytes)
0	1	1 byte
1	2	2 bytes
2	4	4 bytes
3	8	8 bytes
4	16	16 bytes
5	32	32 bytes
6	64	64 bytes
7	128	128 bytes
8	256	256 bytes or 0.25 KB
9	512	512 bytes or 0.5 KB
10	1,024	1 KB
11	2,048	2 KB
12	4,096	4 KB
13	8,192	8 KB
14	16,384	16 KB
15	32,768	32 KB
16	65,536	64 KB
17	131,072	128 KB
18	262,144	256 KB or 0.25 MB
19	524,288	512 KB or 0.5 MB
20	1,048,576	1 MB
32	4,294,967,296	4 GB
33	8,589,934,592	8 GB
34	17,179,869,184	16 GB
35	34,359,738,368	32 GB
36	68,719,476,736	64 GB
37	137,438,953,472	128 GB
38	274,877,906,944	256 GB or 0.25 TB
39	549,755,813,888	512 GB or 0.5 TB
40	1,099,511,627,776	1 TB

http://en.wikipedia.org/wiki/Power_of_two

$2^8 = 256$

The number of values represented by the 8 bits in a byte, more specifically termed as an octet. (The term byte is often defined as a collection of bits rather than the strict definition of an 8-bit quantity, as demonstrated by the term kilobyte.)

$2^{16} = 65{,}536$

The number of distinct values representable in a single word on a 16-bit processor, such as the original x86 processors.[4]

The maximum range of a short integer variable in the C#, and Java programming languages. The maximum range of a **Word** or **Smallint** variable in the Pascal programming language.

http://en.wikipedia.org/wiki/Power_of_two

2^{32} = 4,294,967,296

The number of distinct values representable in a single word on a 32-bit processor. Or, the number of values representable in a doubleword on a 16-bit processor, such as the original x86 processors.[4]

The range of an `int` variable in the Java and C# programming languages.

The range of a `Cardinal` or `Integer` variable in the Pascal programming language.

The minimum range of a long integer variable in the C and C++ programming languages.

The total number of IP addresses under IPv4. Although this is a seemingly large number, IPv4 address exhaustion is imminent.

http://en.wikipedia.org/wiki/Power_of_two

2^{64} = 18,446,744,073,709,551,616

The number of distinct values representable in a single word on a 64-bit processor. Or, the number of values representable in a doubleword on a 32-bit processor. Or, the number of values representable in a quadword on a 16-bit processor, such as the original x86 processors.[4]

The range of a long variable in the Java and C# programming languages.

The range of a **Int64** or **QWord** variable in the Pascal programming language.

The total number of IPv6 addresses generally given to a single LAN or subnet.

One more than the number of grains of rice on a chessboard, according to the old story, where the first square contains one grain of rice and each succeeding square twice as many as the previous square. For this reason the number $2^{64} - 1$ is known as the "chess number".

http://en.wikipedia.org/wiki/Hexadecimal

$0_{hex} = 0_{dec} = 0_{oct}$	0	0	0	0
$1_{hex} = 1_{dec} = 1_{oct}$	0	0	0	1
$2_{hex} = 2_{dec} = 2_{oct}$	0	0	1	0
$3_{hex} = 3_{dec} = 3_{oct}$	0	0	1	1
$4_{hex} = 4_{dec} = 4_{oct}$	0	1	0	0
$5_{hex} = 5_{dec} = 5_{oct}$	0	1	0	1
$6_{hex} = 6_{dec} = 6_{oct}$	0	1	1	0
$7_{hex} = 7_{dec} = 7_{oct}$	0	1	1	1
$8_{hex} = 8_{dec} = 10_{oct}$	1	0	0	0
$9_{hex} = 9_{dec} = 11_{oct}$	1	0	0	1
$A_{hex} = 10_{dec} = 12_{oct}$	1	0	1	0
$B_{hex} = 11_{dec} = 13_{oct}$	1	0	1	1
$C_{hex} = 12_{dec} = 14_{oct}$	1	1	0	0
$D_{hex} = 13_{dec} = 15_{oct}$	1	1	0	1
$E_{hex} = 14_{dec} = 16_{oct}$	1	1	1	0
$F_{hex} = 15_{dec} = 17_{oct}$	1	1	1	1

Done!