Authors: Assad Ebrahim (assad.ebrahim@alum.swarthmore.edu)

Objective. This worksheet introduces the Intel 8086 microprocessor and DEBUG as part of a learning journey in Computer Architecture and Programming. Discovery is facilitated by guided explorations and deep dives into illustrative examples.

Why Debug? Debug is essentially a monitor program, and similar programs/utilities will be the first thing you will look for when programming / interacting with any computer, especially a microcontroller (MCU), an embedded system, or robotics. (see Ch.17 of 8051/8052 MCU, Steiner, 2005)

Contents. Appendix 1 will get you started getting debug (the program) and DOSBox (a dos emulator for x86). DOSBox is required to run debug on recent Windows operating systems (Win7 or later). Appendix 3 is a reference for Debug commands.

1. What is Debug, and why are we using it? [3]

Debug is an interactive assembler, disassembler, and tracer written by Tim Paterson in 1980 to diagnose/maintain the QDOS (quick dos) 16-bit operating system he was building for Seattle Computer Products targeted to the Intel 8086 microprocessor chip with translation compatibility to Gary Kildall's flagship operating system CP/M. As it is interactive, Debug is a great way to learn/teach elementary computer architecture, and introduce assembly language programming.

Why did Debug have such a long life? The DOS operating system and the Intel 8086/8 chips it ran on became synonymous with computing in the age of microcomputers, with IBM choosing the 8088 and MS-DOS for its flagship microcomputer IBM 5150. When Patterson moved to Microsoft in 1981, he brought QDOS (then called 86-DOS) program with him, and this was included in MS-DOS 1.00. It has been part of Windows through Windows XP a remarkable 28-year life (1981-2009) for a utility! Debug finally disappeared from Windows in 2009 when Windows 7 replaced WinXP widely.

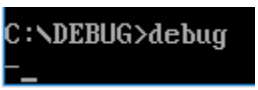
Both Intel and Microsoft have continued to build on these foundations, maintaining backwards compatibility with legacy software and extending the value provided this architecture, even as additional capabilities were added.

Additional Reading

[1] The Chip that Changed Computing: https://plus.google.com/u/0/+AssadEbrahim/posts/HGHFHqW5KK4
[2] Saving Windows from the OS/2 Bulldozer: https://plus.google.com/u/0/+AssadEbrahim/posts/ADq2eexcQzx
[3] The Inside Story of how TI (TMS9900) and Motorola (68000) lost the Microprocessor Great Race. https://plus.google.com/u/0/+AssadEbrahim/posts/WATrWkCm4WD
[4] CP/M (Gary Kildall, Digital Research, formerly from Intel), QDOS (Tim Paterson, Seattle Computer Products), and MS-DOS (Bill Gates, Tim Paterson, Microsoft) – the story of how Microsoft went from selling an MS-BASIC runtime for IBMs first Micro to selling an operating system to IBM that it did not have, but which went on to crush both CP/M and OS/2. https://www.theregister.co.uk/2007/07/30/msdos_paternity_suit_resolved/
[5] Origins of DOS, Patterson, 1983 http://www.patersontech.com/dos/byte%E2%80%93inside-look.aspx
[6] The Roots of DOS & Tim Paterson, 1983, Softalk, http://www.patersontech.com/DOS/softalk.aspx

Debug – Getting Started

Start debug. If you're on a Windows machine newer than WinXP, you'll need to run DOSBox emulator, and then use the WinNT copy of debug.exe (see Appendix 1 for DOSBox setup and Appendix 2 to download debug).
At the command prompt, type debug.
You're now in debug with a – prompt.



Authors: Assad Ebrahim (assad.ebrahim@alum.swarthmore.edu)

Learn by Doing - Tutorial #1 Follow the instructions provided below and work through the tutorial.

In this section you will use Debug to learn about how your computer works, inspect and trace program code, and write and debug small machine language / assembly language programs.

What is a programmable computer?

A programmable computer is a machine that takes instructions from a user (a program) and executes them. The machine itself is an electric/electronic collection of registers, memory, input/output capability, and processing unit (ALU or CPU) that work together to achieve the result, coordinated by the user instructions (program).

Debug is a program that runs on a machine with an MS-DOS operating system, or within an MS-DOS emulator such as DOSBox. It allows you to interface intimately with your machine, inspecting its registers, memory, input/output, and ALU, whilst providing or modifying the coordinating instructions.

1. Viewing and changing register values directly

r Show the x86 chip registers and the values they are holding (e.g. AX=0000).

```
-       r
AX=0000  BX=0000  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0100   NV UP EI PL NZ NA PO NC
073F:0100 0000          ADD     [BX+SI],AL                      DS:0000=CD
```

An 8086 machine (emulated by DOSBox) has 4 general purpose registers (AX, BX, CX, DX), 2 stack registers (SP, BP), 2 index registers (SI, DI), 4 segment registers (DS, ES, SS, CS), 1 instruction pointer (IP), and a flags register holding 16 flags of which 8 are visible to the user.

Each digit in debug is a hexadecimal number, i.e. has value from 0 to F (15), represented in binary by 4-bits (nybble) with 2^4 = 16 values (0 to F). Register AX (=0000) is 16-bits so can hold 4x 4-bit hex digits, or 2x 8-bit bytes, a high and a low byte. The 8086 has an 8-bit (byte-wide) data bus even though it has a 16-bit instruction set (confirm). The high and low bytes can be accessed independently (AH, AL).

rax Show and set the value for register AX to 0x0A01

```
-rax
AX 0000
:0A01
-r
AX=0A01  BX=0000  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0100   NV UP EI PL NZ NA PO NC
073F:0100 0000          ADD     [BX+SI],AL                      DS:0000=CD
```

Observe this is done.

2. Incrementing the register value by specifying then tracing through code

Let's add 1 to AX. To do this we will need to give the chip an instruction. We'll use the INC command to increment the register.

a100 Start assembling user provided instructions beginning from memory location 100 (segment, offset?)

inc al increment the low byte of AX

u100 102 disassemble (view) memory from location 100 to 102

r show the registers and the next instruction (at rip) that will be executed if you trace through one step.

t trace one step (execute next instruction)

Authors: Assad Ebrahim (assad.ebrahim@alum.swarthmore.edu)

```
-a100
073F:0100 inc al
073F:0102
-u100 104
073F:0100 FEC0          INC     AL
073F:0102 0000          ADD     [BX+SI],AL
073F:0104 0000          ADD     [BX+SI],AL
-r
AX=0A01  BX=0000  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0100   NV UP EI PL NZ NA PO NC
073F:0100 FEC0          INC     AL
-t

AX=0A02  BX=0000  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0102   NV UP EI PL NZ NA PO NC
073F:0102 0000          ADD     [BX+SI],AL                      DS:0000=CD
```

Notice AX is now holding 0x0A02, and the instruction pointer has advanced one byte to 0x0102. The inc command has worked.

3. Assembling an infinite looping counter program

a100
add al,1
jmp 0100

u100 102
r

```
AX=0A02  BX=0000  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0100   NV UP EI PL NZ NA PO NC
073F:0100 0401          ADD     AL,01
-t

AX=0A03  BX=0000  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0102   NV UP EI PL NZ NA PE NC
073F:0102 EBFC          JMP     0100
-t

AX=0A03  BX=0000  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0100   NV UP EI PL NZ NA PE NC
073F:0100 0401          ADD     AL,01
-t

AX=0A04  BX=0000  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0102   NV UP EI PL NZ NA PO NC
073F:0102 EBFC          JMP     0100
```

Observe that the jmp command (goto) sets the instruction pointer back to 0100.
Each time through the loop raises AX by another 1.

4. Entering a program in machine language

e100

Type in the following machine code (in hex). You can see this by looking at a small COM program in a hex editor (JujuEdit, or Notepad++ in Hex mode).

Authors: Assad Ebrahim (assad.ebrahim@alum.swarthmore.edu)

```
00000000  B2 48 B4 02 CD 21 B2 65 B4 02 CD 21 B2 6C B4 02  ²H´.Í!²e´.Í!²l´.
00000010  CD 21 B2 6C B4 02 CD 21 B2 6F B4 02 CD 21 B2 20  Í!²l´.Í!²o´.Í!²
00000020  B4 02 CD 21 B2 57 B4 02 CD 21 B2 6F B4 02 CD 21  ´.Í!²W´.Í!²o´.Í!
00000030  B2 72 B4 02 CD 21 B2 6C B4 02 CD 21 B2 64 B4 02  ²r´.Í!²l´.Í!²d´.
00000040  CD 21 B2 21 B4 02 CD 21 CD 20 B4 4C CD 21        Í!²!´.Í!Í
```

The correction with (BC 4C CD 21) MOV AH, 4C; INT 21; provides the correct program terminating instruction sequence for an EXE, replacing (CD 20) INT 20 which was valid when DEBUG was able to run in early days but is not any longer.

Press space to move to the next byte. Press Enter to complete entry.

```
-e100
073F:0100  00.B2   00.48   00.B4   00.02   00.CD   00.21   00.B2   00.65
073F:0108  00.B4   00.02   00.CD   00.21   00.B2   00.6C   AE.B4   FE.02
073F:0110  00.CD   F0.21   46.B2   74.6C   00.B4   00.02   B2.CD   00.21
073F:0118  B2.B2   16.6F   99.B4   00.02   2E.CD   07.21   2E.B2   07.20
073F:0120  00.B4   00.02   00.CD   00.21   00.B2   00.57   00.B4   00.02
073F:0128  00.CD   00.21   00.B2   00.6F   00.B4   00.02   00.CD   00.21
073F:0130  00.B2   00.72   00.B4   00.02   00.CD   00.21   00.B2   00.6C
073F:0138  00.B4   00.02   00.CD   00.21   00.B2   00.64   00.B4   00.02
073F:0140  00.CD   00.21   00.B2   00.21   00.B4   00.02   00.CD   00.21
073F:0148  CD.CD   CD.20_  00:B4   00:4C   00:CD   00:21
```

Unassemble it to see the instructions and find out what this progam does:

```
-u100 120
073F:0100 B248        MOV     DL,48
073F:0102 B402        MOV     AH,02
073F:0104 CD21        INT     21
073F:0106 B265        MOV     DL,65
073F:0108 B402        MOV     AH,02
073F:010A CD21        INT     21
073F:010C B26C        MOV     DL,6C
073F:010E B402        MOV     AH,02
073F:0110 CD21        INT     21
073F:0112 B26C        MOV     DL,6C
073F:0114 B402        MOV     AH,02
073F:0116 CD21        INT     21
073F:0118 B26F        MOV     DL,6F
073F:011A B402        MOV     AH,02
073F:011C CD21        INT     21
073F:011E B220        MOV     DL,20
073F:0120 B402        MOV     AH,02
```

```
-u122 148
073F:0122 CD21        INT     21
073F:0124 B257        MOV     DL,57
073F:0126 B402        MOV     AH,02
073F:0128 CD21        INT     21
073F:012A B26F        MOV     DL,6F
073F:012C B402        MOV     AH,02
073F:012E CD21        INT     21
073F:0130 B272        MOV     DL,72
073F:0132 B402        MOV     AH,02
073F:0134 CD21        INT     21
073F:0136 B26C        MOV     DL,6C
073F:0138 B402        MOV     AH,02
073F:013A CD21        INT     21
073F:013C B264        MOV     DL,64
073F:013E B402        MOV     AH,02
073F:0140 CD21 00:B4  INT     21
073F:0142 B221    00:4C MOV     DL,21
073F:0144 B402 00:CD    MOV     AH,02
073F:0146 CD21    00:21 INT     21
073F:0148 CD20        INT     20
```

Now you see the reason for the repetitive entries – there is a lot of repetition in this code: 2 of every 3 lines is repeated.

mov DL, 0x48 (all numbers in debug are shown in hex)

mov AH, 0x02

int 21

Exercise: Can we do this more efficiently?

Yes. Loop through a string displaying characters. This would be size efficient code but not speed efficient as the number of instructions executed remains the same. We can also call a string printing function in BIOS, which pushes the efficiency to the BIOS implementation.

5. Saving and Loading Files

Authors: Assad Ebrahim (assad.ebrahim@alum.swarthmore.edu)

Before we go further, we should save what we've done, in case something goes wrong and we have to retype the entries…

Saving a file

nfilename.com

rbx make sure the bx register is 0

0

rcx set the cx register with the **HEX** number hh of bytes the program takes (e.g. 74d for Hello.com = 4Ah)

(You can use the calculator to figure out hex numbers for decimal ones)

hh

w writes the program to a command file

```
AX=0000  BX=0000  CX=0056  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=075A  CS=075A  IP=0100   NV UP EI PL NZ NA PO NC
075A:0100 B248          MOV     DL,48
-w
Writing 00056 bytes
```

Loading a file

nfilename.com

l

Success is silent, i.e. there is no prompt that the file is loaded.

You can see the program by pressing u (which lists from 0100 in memory).

6. Tracing and Running through a file

What does this code do?

Tracing instruction at a time.

Tracing (t) steps through a program 1 instruction at a time. Notice that the IP register (instruction pointer) advances to the next instruction. If a JMP is processed, then IP moves to the indicated instruction. If an INT (interrupt) is processed, IP moves to the interrupt service routine (ISR) which is in a different part of memory. If you keep tracing, you will in a few instructions be returned back to your code through IRET xx (interrupt return).

Note – commands like putting characters to the screen happen within the interrupt, so you see them in debug just before the IRET instruction. (To get debug to step over code would be very handy; it would show the result of the interrupt code, which in this case is to print a character to the console.)

Authors: Assad Ebrahim (assad.ebrahim@alum.swarthmore.edu)

```
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0100   NV UP EI PL NZ NA PO NC
073F:0100 B248          MOV     DL,48
-t

AX=0000  BX=0000  CX=0000  DX=0048  SP=00FD  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0102   NV UP EI PL NZ NA PO NC
073F:0102 B402          MOV     AH,02
-t

AX=0200  BX=0000  CX=0000  DX=0048  SP=00FD  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0104   NV UP EI PL NZ NA PO NC
073F:0104 CD21          INT     21
-t
```

Notice the e after the third –t from the top? That's the 2nd character output from Hello World!

```
AX=0248  BX=0000  CX=0056  DX=0065  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=075A  CS=075A  IP=010A   NV UP EI PL NZ NA PO NC
075A:010A CD21          INT     21
-t

AX=0248  BX=0000  CX=0056  DX=0065  SP=FFF8  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=075A  CS=F000  IP=14A0   NV UP DI PL NZ NA PO NC
F000:14A0 FB            STI
-t

AX=0248  BX=0000  CX=0056  DX=0065  SP=FFF8  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=075A  CS=F000  IP=14A1   NV UP EI PL NZ NA PO NC
F000:14A1 FE38          ???     [BX+SI]                         DS:0000=CD
-t
e
AX=0265  BX=0000  CX=0056  DX=0065  SP=FFF8  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=075A  CS=F000  IP=14A5   NV UP EI PL NZ NA PO NC
F000:14A5 CF            IRET
-t

AX=0265  BX=0000  CX=0056  DX=0065  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=075A  CS=075A  IP=010C   NV UP EI PL NZ NA PO NC
075A:010C B26C          MOV     DL,6C
```

Tracing several steps at a time.

tN traces N instructions at a time, showing the register bank after each instruction.

In the above code, as example, t6 will process one character at a time in the above program, 2 assembly instructions, 3 interrupt instructions, 1 IRET instruction.

Note that the u (unassembled) command, shows the next 16 commands from the current instruction pointer.

You can change what instruction will be run next by setting the instruction pointer yourself (a manual jump) using rip and then HHH for the memory location.

Exercise: How efficient/expensive is this program? How to compare to the use of BIOS string print function.

Running the program to a breakpoint

gHHH runs the program to the HHH address in memory (line number, memory location)

Running the program

g runs the program until termination (mov ah 4C; int 21)

Authors: Assad Ebrahim (assad.ebrahim@alum.swarthmore.edu)

```
-                              r
AX=FFFF  BX=0000  CX=0058  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=075A  CS=075A  IP=0100   NV UP EI PL NZ NA PO NC
075A:0100 B248          MOV     DL,48
-g
Hello World!

Program terminated normally
-r
AX=FFFF  BX=0000  CX=0058  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=075A  CS=075A  IP=0100   NV UP EI PL NZ NA PO NC
075A:0100 B248          MOV     DL,48
-_
```

Further Reading & Next Steps

Debug is useful. There are many things that it can do. Follow the references below to learn more.

A more powerful debug program is sst.

To assemble EXE programs directly from assembler, use NASM

Forth is an excellent low level programming language that allows seamless embedding of assembly. F-PC runs within DOSBox and allows exploring x86 assembly language fully from within the Forth programming environment.

Application areas are: 1) microcontroller and embedded systems programming, 2) low level graphics programming (fast graphics), 3) higher level C language programming, and 4) scientific/numerical programming (simulation, fast processing, etc.)

[1] Robert Lafore, 1991, Assembly Language Primer for the IBM PC and XT, The Waite Group, 512pps.
https://www.amazon.co.uk/Assembly-Language-Primer-Plume-computer/dp/0452257115
[2] MS-Debug Command Description, Daniel B. Sedory
https://thestarman.pcministry.com/asm/debug/debug2.htm
[2] Starman's Guide to Debug, 2004-2017, Daniel B. Sedory
http://www.starman.vertcomp.com/asm/debug/debug.htm
[3] Debug (command)
https://en.wikipedia.org/wiki/Debug_(command)
[4] MS-Debug History (1981-2009)
http://www.kerrywong.com/2009/05/08/ms-debug-1981-2009/
[5] Debug Tutorial
https://jakash3.wordpress.com/2010/02/08/debug-exe-tutorial/
[6] Debug Tutorial - Detailed
http://www.armory.com/~rstevew/Public/Tutor/Debug/debug-manual.html
[7] Tutorial with graphics mode
http://www.davidwills.us/cmis310/8086/debug.html
[8] Assembly language tutorial x86 in AT&T syntax
http://flint.cs.yale.edu/cs421/papers/x86-asm/asm.html
[9] Starman's Realm
http://www.starman.vertcomp.com/
[10] Starman's Assembly 101
http://www.starman.vertcomp.com/asm/index.html
[11] Starman's Comprehensive References page
http://www.starman.vertcomp.com/asm/index.html#REF

Authors: Assad Ebrahim (assad.ebrahim@alum.swarthmore.edu)

FREQUENTLY ASKED QUESTIONS

FAQ1: Why does INT 20 not work in DOSBox as program termination, but it used to work in Windows in WinXP or earlier?

Note: Using INT 20 as program termination within DOSBox will cause debug to crash. Fix this by changing the exit code to MOV AL, 4C; INT 21. You can load the program in DOSBox, use the a command to assemble at the INT 20 memory location and overwrite it with the correct exit code. Then use the w command to save the modified program.

Here's the explanation: https://stackoverflow.com/questions/12591673/whats-the-difference-between-using-int-0x20-and-int-0x21-ah-0x4c-to-exit-a-16

FAQ2: What assembly languages does debug understand?

As a Microsoft product, debug uses Intel syntax not the UNIX/Linux convention of AT&T / Motorola syntax. Examples of both:

Intel:

MOV AH, 4; load literal value 4 in low byte of AX register

MOV AX, BX; copy 2 bytes to AX from register BX

AT&T equivalent:

mov 4, ah; load literal value 4 to register AH.

mov bx, ax; copy 2 bytes in BX to register AX.

FAQ3. What is the register set in 8086 and what do they do?

AX, BX, CX, DX are the standard registers.

BP/SP are pointers to the base and top of stack respectively (see Part Two for understanding the stackframe, stack pointers, and stack instructions (PUSH/POP).

SI/DI are index pointers.

DS/ES/SS/CS are segment pointers (see Part Three for understanding these).

IP is instruction pointer --- we have covered that in this paper.

The rest (NV, UP, EI, PL, NZ, NA, PO, NC) are flags. (see Part Four for understanding these).

```
C:\DEBUG>debug
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0100   NV UP EI PL NZ NA PO NC
073F:0100 5A            POP     DX
```

FAQ4. What happens when a file greater than 65,536 bytes (=2^16) is loaded into debug (e.g. edit.com = 69,886 bytes)?

FAQ5. What is the bloat factor in file sizes when going from 8088 (8-bit word), 8086 (16-bit word) or 80686 (32-bit word) to 64-bit machine? X6, X4, or X2 because every e.g. 8-bit word for an 8088 processor now occupies 64-bits on disk.

Authors: Assad Ebrahim (assad.ebrahim@alum.swarthmore.edu)

APPENDICES

Appendix 1. Running MS-DEBUG in a modern Windows machine within DOSBox a DOS Emulator.

Windows removed support for 16-bit DOS and all 16-bit applications when it moved to 64-bits (Win7 onward), so to run such programs now you will need a DOS Emulator. The two leading DOS emulators for Windows are DOSBox (specialized for Gaming and other graphics, sound card emulation) and vDOS/vDOSPlus (specialized for business software performance, integration with Windows, printing, zoom in/out resizing of fonts, etc.) [1,2,3]. **DOSBox** has been stable at version 0.74 since 2013 and emulates 80286/80386 and its peripherals. [1] **vDOSPlus** has a Portable version from Nov 2015, comes with DosZip Commander v2.55 (Hjort Nidudsson), and 4DOS command prompt. (**vDOS** continues to be worked on with full source code through 2017, and updates through 2018.)

Note: DOSBox requires debugNT.exe. vDOSplus requires debug98.exe

Download DOSBox DOS emulator from the DOSBox project page: http://www.dosbox.com/download.php?main=1 or directly from SourceForge: https://sourceforge.net/projects/dosbox/files/dosbox/0.74/DOSBox0.74-win32-installer.exe/download

Install to the c:\ drive instead of to the default Program Files location, e.g. to c:\totalcmd\dosbox\

Use the -noconsole switch to start DOSbox cleanly (suppresses a dummy console window that does nothing)

Set a virtual drive to the path where you have your dos files, e.g.:

```
mount c c:/totalcmd/dosprogs
```

If you want DOSBox to point to this path when it launches, you can set the mount drive instruction to run automatically at startup by copying it to the DOSBox configuration file:

```
run Doskey Options.bat
it will write the config file to
"c:\Documents and Settings\ebraha01\Local Settings\Application Data\DOSBox\dosbox-0.74.conf"
modify it there, save (no .txt extension!)
```

To AUTO-MOUNT:

```
add the mount instructions to [autoexec] command in configuration file:
        http://ipggi.wordpress.com/2008/02/17/dosbox-beginners-newbie-and-first-timers-guide/
```

To relocate the config file to the dos box folder:

```
start dos box with the switch
        -conf c:\totalcmd\dosBox-0.74\dosbox-0.74.conf
```

Debug can be downloaded from the MathSciTech downloads page:

```
http://www.mathscitech.org/downloads/debug.exe
```

References:

[1] What DOSBox emulates. https://www.dosbox.com/status.php?show_status=1
[2] vDOS. http://www.vdos.info/
[3] vDOSplus http://vdosplus.org/

Appendix 2. Debug commands

DEBUG COMMANDS

==========

Authors: Assad Ebrahim (assad.ebrahim@alum.swarthmore.edu)

? displays debug help menu
command list for debug: https://thestarman.pcministry.com/asm/debug/debug2.htm

cmd (brings up a command window, DOS box, console window)
On machines newer than WinXP, you need to use DOSbox (DOS emulator) and run debug.exe from WinNT.
debug (puts the console into debug mode
- the prompt
q ; quit
a100 ; input assembly commands starting at 0x0d22:0100, [return] to end input mode
u100 ; disassemble commands starting at 0x0100
d100 ; dump (show) bytes beginning at 0x0100
e100 ; replace ONE byte at the specified address
r ; show registers
t ; trace (execute) one statement
t=addr n ; trace n instructions beginning at addr
[not true] 24 ; this is the opcode for terminating the program, so that you don't go "forever"

WORKING WITH MEMORY
r show all registers
notice that debug is for the 16-bit 8086 chip.
notice code, data, extra, and stack segments come up: 073F.
notice instruction pointer starts at: 0100.
notice base pointer, source and destination index pointers are null: 0000.
notice stack pointer starts at: 00FD, 3 bytes below 0100.
remember: the stack extends DOWNWARD in memory as you push stuff on it...
the stack is 16-bits (2 bytes), so each push subtracts 2 bytes, and each pop addes 2 bytes
r<reg> show register reg, with a chance to set its value (can only show/modify 16-bit registers) e.g. rax rbx, etc.
rip set instruction pointer register
rf show all flag registers. You can then type in the two digit code to set the corresponding flag.

CONVENTIONS
80x86/8088 are "little-endian" as follows:
0x1234 is stored as 34 12 in debug, i.e. the LSB is in the LEFT memory locations, and the MSB is in the RIGHT memory locations, which provides a nice pnemonic.
The Intel processor is LITTLE-ENDIAN, i.e. the little end of the number goes into the pipe first, so that LSB is on the left with MSB on the right... not quite what we do when we read... and you see that in the BX+9A00, the 9A is the MSB, 00 is the LSB

WORKING WITH INSTRUCTIONS
d100 dumps from 100h IN THE DATA SEGMENT --- this is typically the same as the code segment for COM files, but will typically be different for OMF EXEs.
d100 200 dumps from 100h to 200h
d100 should see B0 01 (machine code)
dHHHH:hhhh dumps from segment HHHH offset hhhh

Authors: Assad Ebrahim (assad.ebrahim@alum.swarthmore.edu)

u100 unassemble (disassemble) from 100h
u100 200 unassemble (disassemble) from 100h to 200h

f100 200 ff fills from 100h to 200h with ff

e100 allows editing memory at 100h, 1 byte at a time. Press space to move on to next byte. Format is curval.newval Useful if you want to enter a machine language code in machine language, instead of assembling it. You are entering machine language op-codes and arguments, i.e. you are coding directly in the machine language.
a100 allows assembling program from line 100h. Press enter to return to the command prompt. You are entering assembly code mnemonics and debug is assembling them into the corresponding machine language. Example: type mov al,01 (intel x86 8-bit assembly language)

RUNNING PROGRAMS IN MEMORY
HEX convention - all numbers are hex
r show registers. at the bottom of the printout is the command that the IP now points to...
t trace (one command at a time). Completing a trace shows the registers (r) and the next command, both dumped and disassembled --
-- NOTE: DON'T trace an interrupt! Either skip over it, or NOP it out using azzz nop or ezzz 90 (90 is the opcode for nop)
g go : runs the program from memory, until...?
ghh run till arrive at the line hh, i.e. hh becomes a breakpoint

LOADING A COMMAND FILE (.com) INTO MEMORY
nfilename.com sets name variable to the given file (has to be .com?)
l loads the program into memory (beginning at 0100h)

Note: rcx will hold the number (in hex) of bytes read in from the file. Useful if you want to save it with another name.

SAVING A COMMAND FILE
nfilename.com
rbx make sure the bx register is 0
0
rcx set the cx register with the **HEX** number hh of bytes the program takes (e.g. 74d for Hello.com = 4Ah) (You can use the calculator to figure out hex numbers for decimal ones)
hh
w writes the program to a command file
===CHECK===
nfilename.com sets name variable to the given command program (has to be com)
l loads the command program

LOADING EXECUTABLE FILE (.exe) INTO MEMORY
Run debug hello.exe, where hello.exe is a simple compiled Hello World! program. Debug will load you up where the program is about to start... but there seems to be a limit to how large a program debug can handle...

Authors: Assad Ebrahim (assad.ebrahim@alum.swarthmore.edu)

You can do the same with sst: sst hello.exe But it appears that SST has a different length window. More is cut off!

Another way to disassembler a .exe is to NASM's ndisasm.exe

Authors: Assad Ebrahim (assad.ebrahim@alum.swarthmore.edu)

Objective. This worksheet continues the learning journey in Computer Architecture and Programming. DEBUG is used to discover key points through guided exploration and deep dives into illustrative examples. In this section, we explore (A) memory and pointers, and (B) the stackframe, stack pointers SP/BP, and stack instructions: PUSH, POP, and (C) memory segments.

Computer Memory and the von Neumann Architecture

The revolutionary aspect of modern computer design is that memory is the same whether storing instructions or data.

Debug and Memory

Debug starts up in the tiny memory model (see FAQ1 for description of 8086 memory models), in which CS=SS=DS, i.e. code, stack, and data all share the same 64k byte segment. All general purpose, stack, and index registers are 0, and the program enters the Tiny Memory Model (DS=ES=SS=CS). IP points to 0100h (which could contain any random data) interpreted as a command. The flags are set to defaults (No Overflow, Up Direction, Enable Interrupts, Plus, Nonzero, No auxiliary (nibble) carry, Parity Odd, No Carrry). See FAQ2 for Debug state after loading a file.

debug in clean state at startup --- use command r to see the register bank and next instruction.

```
C:\DEBUG>DEBUG.EXE
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0100   NV UP EI PL NZ NA PO NC
073F:0100 0000          ADD     [BX+SI],AL                      DS:0000=CD
```

The Registers (AX, BX, CX, DX)

AX, BX, CX, DX are the standard 16-bit registers on an 8086. Each has a high and low byte which can be accessed independently (e.g. AH, AL).

In code, register contents are changed using MOV (move, copy, load, or store, depending on context), and mathematical or logical operators INC (increment), DEC (decrement), AND, OR, NOT, ADD, SUB, MUL, DIV, etc.

The registers each have their own superpowers – capabilities which are optimized in hardware for them, and which result in smaller code or faster execution.

AX is optimized as an **accumulator** for arithmetic and logical operations. So adding an immediate to AX or AL is a smaller instruction than to any other register because AX/AL have short code for this: 0x04hh (add al, hh) and 0x05hhhh (add ax,hhhh), 2 bytes vs. 3 bytes, or 3 bytes vs. 4 bytes.

```
-a100
073F:0100 add al,a0
073F:0102 add bl,a0
073F:0105 add cl,a0
073F:0108 add dl,a0
073F:010B add ax,00a0
073F:010E add bx,00a0
073F:0112 add cx,00a0
073F:0116 add dx,00a0
073F:011A
-u100
073F:0100 04A0          ADD     AL,A0
073F:0102 80C3A0        ADD     BL,A0
073F:0105 80C1A0        ADD     CL,A0
073F:0108 80C2A0        ADD     DL,A0
073F:010B 05A000        ADD     AX,00A0
073F:010E 81C3A000      ADD     BX,00A0
073F:0112 81C1A000      ADD     CX,00A0
073F:0116 81C2A000      ADD     DX,00A0
```

Authors: Assad Ebrahim (assad.ebrahim@alum.swarthmore.edu)

BX is optimized as a **memory pointer** for memory access instructions such as MOV, ADD, SUB, in memory mode.

CX is optimized as a **counter** for use with iteration instructions as LOOP, REPZ, etc.

The Flags (of=NV,OV, df=UP,DN, if=DI,EI, sf=PL,NG, zf =NZ,ZR, af=NA,AC, pf=PO,PE, cf=NC,CY) – more details in Part 4.

The eight 8086 flags are bits that provide signals for decisions to be taken as a result of processing (e.g. overflow, sign, parity, zero outcome, or carry) or are control bits (e.g. set memory direction, enable interrupt).

```
This is an 8-bit register that holds 1-bit of data for each
of the eight "Flags" which are to be interpreted as follows:

Textbook abbrev. for Flag Name =>   of df if sf zf af pf cf
If the FLAGS were all SET (1),      -- -- -- -- -- -- -- --
they would look like this...   =>   OV DN EI NG ZR AC PE CY
If the FLAGS were all CLEARed (0),
they would look like this...   =>   NV UP DI PL NZ NA PO NC
```

The Stackframe, Stack Segment and Stack Pointers (SS, BP, SP) – we cover this in Part 2

The stackframe sits in the stack segment (SS). The memory available within the stack (in bytes) at any point is what sits between the base pointer (BP) which points to the bottom of the stack (SS:BP), and the stack pointer (SP), which points to the top of stack (SS:SP). All PUSH and POP instructions operate on the top of stack. The convention in the 8086 is that the stack grows DOWNWARD (i.e. to smaller memory addresses). Note that SP points to the LAST item that is on the stack, so a PUSH must first advance (go down) by W bytes (W being the size of the data word being pushed), and then store to the stack, and POP needs only go up by W bytes to point to the 2nd last item. It is not possible to tell how many items are on the stack, just how much space is available to use. In Figure 1, BP=0000 and SP=00FD so there are 0XFD (251d) bytes available on the stack frame.

Stack segment can be changed in code by pushing an address onto the stack and POP SS.

Base and stack pointers are registers and can be used as desired using MOV and other register operations (ADD, OR, etc.)

Code Segment and Instruction Pointer (CS, IP)

The code segment (CS) is where operating memory is stored. In the tiny memory model, the SS=CS and consists of the first 0x100 bytes of memory (0000 to 00FF). Coded instructions begin at 0x100, and this is where the instruction pointer (IP) begins. From 0x0000 to 0x0100 lives the stack frame.

Within Debug, CS and IP can be changed (rds and rip commands).

In code, IP is changed by JMP, and undocumented pop cs instruction (0F).

Data and Extra Segments (DS, ES)

Data Segment (DS) for all references to data, Extended Segment (ES) for string operations.

Index Pointers (SI, DI)

These registers are source/destination indices used as offsets into data/extra segments for mass movement of data (repz, stosw, movsw, etc) or by convention in manual movements of data chunks.

Segmented Memory Model – we cover this in more detail in Part 3

The 8086 has a 20-bit memory bus able to physically (linearly) address 2^20 = 100,000h bytes (1MB). But the 8086 has a 16-bit word length, so can only address 2^16 = 10,000h bytes (64KB). Each 64KB chunk is called a segment.

Authors: Assad Ebrahim (assad.ebrahim@alum.swarthmore.edu)

Exercise: How many distinct segments are there in a natural partition of physical memory? 16.

The key point is that the natural syntax of the 8086 works within a memory segment.

The particular segment being referred to is based on a referencing convention for each memory operation.

Data operations (MOV, etc.) refer to DS. Stack operations (PUSH, POP, etc.) refer to SS. String operations (REPZ, etc.) refer to ES.

The tiny memory model is simple: DS=ES=SS=CS --- the entire program with all its memory lives within 64KB.

Given Chuck Moore's claim that a Forth program should be no more than 1000 instructions long, this would be a 4KB program (assuming 4 bytes per instruction). If the quote was for 1000 lines long then assuming 5 Forth instructions per line, equivalent 4 bytes per instruction, this is 20KB program. Debug itself is a 20k program.

Numbers and Literals – more detail in Part X.

Literals are specific values, called immediates (abbreviated IMM). All numbers are assumed to be in HEX, but the interpretation of whether it is a signed or unsigned integer, or fixed point or floating point number depends on the interpretation context. So 0xFFFE is -2 if signed integer or 65,534d if unsigned.

Numbers > 255 (one byte) are little endian, i.e. going from left to right we have the least significant byte (LSB) first. So 0xFFFE (-1) is stored as FE FF in memory (with FE in the first or lower numbered memory).

Translation from Hex to Decimal

Decimal, Binary, Hex, and Octal are just numbers with different bases (10, 16, 2, and 8). The meaning of numbers as a sequence of digits remains the same in all bases: numbers are power series with the given base. Examples:

140d = 1*10^2 + 4^10*1 + 0 = 100 + 40 + 0 (common in standard arithmetic)

140h = 1*16^2 + 4^16*1 + 0 = 256 + 64 + 0 = 320d (common with 16-bit computers)

140h in binary = 0001 0100 0000 (translate each of the hex digits directly into binary and concatenate)

o140 in octal (common with 8-bit computers) = 1*8^2+4*8^1+0 = 64 + 32 = 96

Exercise: 140h in octal? 600

Memory, Addresses, and Pointers – we cover this in more detail in Part 4

We have seen that a computer consists of a sequence of memory cells (latches) that can hold values based on our interpretation of consecutive bits as numbers base 2.

So we need to distinguish between literal/immediate values such as 0xFFFE and memory address [0xFFFE]. Memory addresses are always specified in [square brackets]. This address label is called a pointer (or indirect reference).

Exercise 1: Let's look at an example:

```
-a100
073F:0100 add [bx+si],al
073F:0102 add [bx+di],al
073F:0104 add [bp+si],al
073F:0106 add [bp+di],al
073F:0108 add [si],al
073F:010A add [di],al
073F:010C add [2000],al
073F:0110 add [bx],al
073F:0112
-u100 110
073F:0100 0000          ADD     [BX+SI],AL
073F:0102 0001          ADD     [BX+DI],AL
073F:0104 0002          ADD     [BP+SI],AL
073F:0106 0003          ADD     [BP+DI],AL
073F:0108 0004          ADD     [SI],AL
073F:010A 0005          ADD     [DI],AL
073F:010C 00060020      ADD     [2000],AL
073F:0110 0007          ADD     [BX],AL
```

Authors: Assad Ebrahim (assad.ebrahim@alum.swarthmore.edu)

We'll initialize all the registers: BX=1000; BP=1200; SI=500; DI=600; AX=00FD;

```
-rbx
BX 1000
:1000
-rsi
SI 0500
:500
-rdi
DI 0000
:600
-rbp
BP 0000
:1200
```

```
-rax
AX 0000
:FD
-r
AX=00FD  BX=1000  CX=0000  DX=0000  SP=00FD  BP=1200  SI=0500  DI=0600
DS=073F  ES=073F  SS=073F  CS=073F  IP=0100   NV UP EI PL NZ NA PO NC
073F:0100 0000          ADD     [BX+SI],AL                      DS:1500=00
```

Stepping through the code 8 times (command t for trace), will put FD at the respective memory slots, which we can see by dumping the relevant memory:

```
-       dff0 100f
073F:0FF0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
073F:1000  FD 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
```

TIP: it is useful to be able to use DEBUG as a calculator in order to stay within it. Press ? for help.

H 1 1 (adds and subtracts)

Now it is straightforward to specify the two rows of memory to be dumped, 0xF before and 0xF after.

```
-h ff0 500
14F0  0AF0
-h 100f 500
150F  0B0F
-d14f0 150f
073F:14F0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
073F:1500  FD 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
-d15f0 160f
073F:15F0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
073F:1600  FD 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
-d16f0 170f
073F:16F0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
073F:1700  FD 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
-d17f0 180f
073F:17F0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
073F:1800  FD 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
-d1ff0 200f
073F:1FF0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
073F:2000  FD 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
```

Sure enough, the bytes are stored there.

Exercise 2: Taking advantage of DEBUG's memory view

First paragraph. Notice the SS:1700=FF at the right? This is providing the physical address of the pointer [BP+SI] and showing what is in it before the AND instruction executes. Brilliant and very handy.

Authors: Assad Ebrahim (assad.ebrahim@alum.swarthmore.edu)

Notice that the default interpretation is this is a NEAR pointer into SS. Why? Because BP has been used. If BX had been used, it would have been DS. If

Second paragraph. Notice how the baseline segment reference (CS) is explicitly stated in code?

Third paragraph. Observe the SS:1700 location

```
-r
AX=FFFF  BX=0000  CX=E688  DX=0000  SP=FFFE  BP=1200  SI=0500  DI=0600
DS=075A  ES=075A  SS=075A  CS=075A  IP=012A   NV UP EI NG NZ NA PO NC
075A:012A 2032          AND     [BP+SI],DH                      SS:1700=FF
-t

AX=FFFF  BX=0000  CX=E688  DX=0000  SP=FFFE  BP=1200  SI=0500  DI=0600
DS=075A  ES=075A  SS=075A  CS=075A  IP=012C   NV UP EI PL ZR NA PE NC
075A:012C 2E            CS:
075A:012D 3030          XOR     [BX+SI],DH                      CS:0500=4C
-d16f0 170f
075A:16F0  FF FF FF FF 01 00 04 52-45 50 54 01 3F 3D 00 FF   .......REPT.?=..
075A:1700  00 FF FF 00 00 04 52 45-50 5A 01 0F F3 02 FF FF   ......REPZ......
-d4f0 50f
075A:04F0  03 43 4C 49 01 0F FA 00-00 04 00 04 01 00 04 43   .CLI...........C
075A:0500  4C 54 53 01 04 06 00 00-04 00 04 01 00 03 43 4D   LTS...........CM
-t

AX=FFFF  BX=0000  CX=E688  DX=0000  SP=FFFE  BP=1200  SI=0500  DI=0600
DS=075A  ES=075A  SS=075A  CS=075A  IP=012F   NV UP EI PL NZ NA PO NC
075A:012F 206F72        AND     [BX+72],CH                      DS:0072=00
-d4f0 50f
075A:04F0  03 43 4C 49 01 0F FA 00-00 04 00 04 01 00 04 43   .CLI...........C
075A:0500  4C 54 53 01 04 06 00 00-04 00 04 01 00 03 43 4D   LTS...........CM
```

Exercise 3. Verifying the baseline segments for each of the memory modes

```
-u100 110
075A:0100 0000          ADD     [BX+SI],AL
075A:0102 0001          ADD     [BX+DI],AL
075A:0104 0002          ADD     [BP+SI],AL
075A:0106 0003          ADD     [BP+DI],AL
075A:0108 0004          ADD     [SI],AL
075A:010A 0005          ADD     [DI],AL
075A:010C 00060020      ADD     [2000],AL
075A:0110 0007          ADD     [BX],AL
```

```
-r
AX=FFFF  BX=0000  CX=E688  DX=0000  SP=FFFE  BP=1200  SI=0500  DI=0600
DS=075A  ES=075A  SS=075A  CS=075A  IP=0100   NV UP EI PL NZ NA PO NC
075A:0100 0000          ADD     [BX+SI],AL                      DS:0500=4C
-d04f0 50f
075A:04F0  03 43 4C 49 01 0F FA 00-00 04 00 04 01 00 04 43   .CLI...........C
075A:0500  4C 54 53 01 04 06 00 00-04 00 04 01 00 03 43 4D   LTS...........CM
```

[BX+hh] as a NEAR pointer is relative to DS regardless of hh. Before the instruction executes, it holds 4C.

```
-t

AX=FFFF  BX=0000  CX=E688  DX=0000  SP=FFFE  BP=1200  SI=0500  DI=0600
DS=075A  ES=075A  SS=075A  CS=075A  IP=0102   NV UP EI PL NZ AC PE CY
075A:0102 0001          ADD     [BX+DI],AL                      DS:0600=45
-d04f0 50f
075A:04F0  03 43 4C 49 01 0F FA 00-00 04 00 04 01 00 04 43   .CLI...........C
075A:0500  4B 54 53 01 04 06 00 00-04 00 04 01 00 03 43 4D   KTS...........CM
```

After executing the statement, the location holds 4B. Why? The low byte of 4C+FF=014B is 4B.

```
-h 4c ff
014B  FF4D
```

Authors: Assad Ebrahim (assad.ebrahim@alum.swarthmore.edu)

Next. [BP+hh] as a NEAR pointer is relative to SS regardless of hh.

```
AX=FFFF  BX=0000  CX=E688  DX=0000  SP=FFFE  BP=1200  SI=0500  DI=0600
DS=075A  ES=075A  SS=075A  CS=075A  IP=0104   NV UP EI PL NZ AC PE CY
075A:0104 0002          ADD     [BP+SI],AL                      SS:1700=00
```

Next. [SI], [DI], and [hh] as NEAR pointers are also all relative to DS.

```
AX=FFFF  BX=0000  CX=E688  DX=0000  SP=FFFE  BP=1200  SI=0500  DI=0600
DS=075A  ES=075A  SS=075A  CS=075A  IP=0108   OV UP EI PL NZ NA PO CY
075A:0108 0004          ADD     [SI],AL                         DS:0500=4B
-t

AX=FFFF  BX=0000  CX=E688  DX=0000  SP=FFFE  BP=1200  SI=0500  DI=0600
DS=075A  ES=075A  SS=075A  CS=075A  IP=010A   NV UP EI PL NZ AC PO CY
075A:010A 0005          ADD     [DI],AL                         DS:0600=44
-t

AX=FFFF  BX=0000  CX=E688  DX=0000  SP=FFFE  BP=1200  SI=0500  DI=0600
DS=075A  ES=075A  SS=075A  CS=075A  IP=010C   NV UP EI PL NZ AC PO CY
075A:010C 00060020      ADD     [2000],AL                       DS:2000=00
```

Exercise 4: Reverse Engineering the Opcodes

ONE instruction (0x00HH) add mem/reg8, reg8***

	0000-0007	add mem*, al	
1	0000	add [bx+si], al	*rel. to DS*
2	0001	add [bx+di], al	*rel. to DS*
3	0002	add [bp+si], al	*rel. to SS*
4	0003	add [bp+di], al	*rel. to SS*
5	0004	add [si], al	*rel. to DS*
6	0005	add [di], al	*rel. to DS*
7	0006	add [hhhh], al	*rel. to DS*
8	0007	add [bx], al	*rel. to DS*

memory reference pattern is

baseReg, indexReg, constant

* means iteration over 8 patterns

8086 Instructions

	Machine Codes	Assembly	
	0000-00BF	**add mem**, reg8***	***C0h***
1	0000-0007	add mem*, al	*8h*
2	0008-000F	add mem*, cl	*8h*
3	0010-001F	add mem*, dl/bl	*10h*
4	0020 - 003F	add mem*, ah/ch/dh/bh	*20h*
5	0040xx - 007Fxx	add mem**+xx, reg8*	*40h*
6	0080xxxx-00BFxxxx	add mem**+xxxx, reg8*	*40h*

mem*+xx is e.g. [bx+si+xx], xx is constant offset

** is inner iteration, * is outer iteration

reg8 is 8-bit registers AL, CL, DL, BL, AH, CH, DH, BH

reg16 are 16-bit registers AX, CX, DX, BX, SP, BP, SI, DI

Authors: Assad Ebrahim (assad.ebrahim@alum.swarthmore.edu)

Iteration Table

	mem**	mem**+xx	mem**+xxxx	reg8*	reg16*
1	[bx+si]	[bx+si+xx]	[bx+si+xxxx]	al	ax
2	[bx+di]	[bx+di+xx]	[bx+di+xxxx]	cl	cx
3	[bp+si]	[bp+si+xx]	[bp+si+xxxx]	dl	dx
4	[bp+di]	[bp+di+xx]	[bp+di+xxxx]	bl	bx
5	[si]	[si+xx]	[si+xxxx]	ah	sp
6	[di]	[di+xx]	[di+xxxx]	ch	bp
7	[imm=hhhh]	[imm=hhhh+xx]	[imm=hhhh+xxxx]	dh	si
8	[bx]	[bx+xx]	[bx+xxxx]	bh	di
	8h	*8h*	*8h*	*8h*	*8h*
	18h (24d)			*10h (16d)*	
	180h (384d = 24d x 16d)				

	00C0-00FF	add reg8**,reg8*
1	00C0	add al, al
2	00C1	add cl, al
3	00C2	add dl, al
4	00C3	add bl, al
5	00C4	add ah, al
6	00C5	add ch, al
7	00C6	add dh, al
8	00C7	add bh, al
9	**00C8-00CF**	**add reg8*, cl**
10	**00D0-00DF**	**add reg8*, dl/bl**
11	**00E0-00FF**	**add reg8*, ah/ch/dh/bh**

reg8 is 8-bit registers AL, CL, DL, BL, AH, CH, DH, BH

The rest of the pattern:

	Instructions 0x00, 0x01, 0x02, 0x03			*400h (1024d)*
1	0000-00BF	add mem**, reg8*	*C0h*	*100h (256d)*
2	00C0-00FF	add reg8**,reg8*	*40h*	
3	0100-01BF	add mem**,reg16*	*C0h*	*100h (256d)*
4	01C0-01FF	add reg16**, reg16*	*40h*	
5	0200-02BF	add reg8*, mem**	*C0h*	*100h (256d)*
6	02C0-02FF	add reg8*, reg8**	*40h*	
7	0300-03BF	add reg16*, mem**	*C0h*	*100h (256d)*
8	03C0-03FF	add reg16*, reg16**	*40h*	

** is inner iteration, * is outer iteration
reg8 is 8-bit registers AL, CL, DL, BL, AH, CH, DH, BH
reg16 are 16-bit registers AX, CX, DX, BX, SP, BP, SI, DI

Other memory/register instructions look a lot like this:

Exercise 3. Test the overriding of default stack

Using CS: and ES: overrides.

Exercise 2. Discovering the Stack (PUSH, POP)

Only registers and memory can be pushed, not literals.

Valid: push ax; push [bx]; Invalid: push 0xff (Try it, i.e. try to assemble push ax and push ah or push al.)

Let's reverse engineer how the 8086 stack works by experimenting with push and pop.

Authors: Assad Ebrahim (assad.ebrahim@alum.swarthmore.edu)

Assemble the following instructions into memory location 0x0100 (a100 command)

mov ax, 0xFE; store 0xFE into register AX
push ax; AX value to stack
dec ax; decrement AX
push ax; AX value to stack.

Let's trace this.

Observe: at clean startup, SP points to 0xFD. And SS=CS, i.e. the code and stack segments are the same (see FAQ2 for when they divege,

```
C:\DEBUG>debug
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0100   NV UP EI PL NZ NA PO NC
```

After first push, SP goes to 0xFB. Pushing a 16-bit (2 byte) value from AX onto the stack advanced the stack pointer by 2 bytes – makes sense.

```
AX=00FE  BX=0000  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0103   NV UP EI PL NZ NA PO NC
073F:0103 50            PUSH    AX
-t

AX=00FE  BX=0000  CX=0000  DX=0000  SP=00FB  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0104   NV UP EI PL NZ NA PO NC
```

After second push, SP goes to 0xF9.

```
-r
AX=00FE  BX=0000  CX=0000  DX=0000  SP=00FB  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0104   NV UP EI PL NZ NA PO NC
073F:0104 48            DEC     AX
-t

AX=00FD  BX=0000  CX=0000  DX=0000  SP=00FB  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0105   NV UP EI PL NZ NA PO NC
073F:0105 50            PUSH    AX
-t

AX=00FD  BX=0000  CX=0000  DX=0000  SP=00F9  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0106   NV UP EI PL NZ NA PO NC
```

Let's inspect the stack in debug to see the values we have pushed.

uf8 u105

Notice: 0x00FD (second value pushed) is stored in cells 0xF9 and 0xFA in little endian order, i.e. FD 00 (going from low to high).

0x00FE (first value pushed) is stored in cells 0xFB and 0xFC.

The stack pointer SP=0xF9 points to the last valid entry on the stack.

```
-uf8 105
073F:00F8 01FD          ADD     BP,DI
073F:00FA 00FE          ADD     DH,BH
073F:00FC 0000          ADD     [BX+SI],AL
073F:00FE 0000          ADD     [BX+SI],AL
073F:0100 B8FE00        MOV     AX,00FE
073F:0103 50            PUSH    AX
073F:0104 48            DEC     AX
073F:0105 50            PUSH    AX
-r
AX=00FD  BX=00FD  CX=00FE  DX=0000  SP=00F9  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0106   NV UP EI PL NZ NA PO NC
```

Outside the tiny memory model, i.e. when the stack segment SS is different from the code segment CS, use the extended notation SS:xxxx to see the stackframe. This works with both u and d commands.

Example: u0800:0070 0080 / ues:0070 0080 or d0800:0070 / des:0070

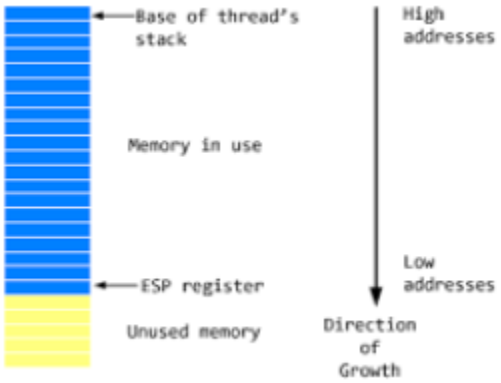
Authors: Assad Ebrahim (assad.ebrahim@alum.swarthmore.edu)

```
-res
ES 073F
:0800
-des:0070 0080
0800:0070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0800:0080  00                                                .
-ues:0070 0080
0800:0070 0000          ADD     [BX+SI],AL
0800:0072 0000          ADD     [BX+SI],AL
0800:0074 0000          ADD     [BX+SI],AL
0800:0076 0000          ADD     [BX+SI],AL
0800:0078 0000          ADD     [BX+SI],AL
0800:007A 0000          ADD     [BX+SI],AL
0800:007C 0000          ADD     [BX+SI],AL
0800:007E 0000          ADD     [BX+SI],AL
0800:0080 0000          ADD     [BX+SI],AL
-r
AX=00FD  BX=1000  CX=0000  DX=0000  SP=00FD  BP=1200  SI=0500  DI=0600
DS=073F  ES=0800  SS=073F  CS=073F  IP=0112   NV UP EI NG NZ NA PO NC
073F:0112 0000          ADD     [BX+SI],AL                   DS:1500=FD
```

A few take-aways:

1) In the tiny memory model, stack memory and instruction memory are contiguous – instructions start at 0x0100. The stack grows downward from 0xFD (which was holding 0x0000).
2) A push operation will first decrement SP by 2 bytes and then write the value to the pointer.
3) A pop operation will first read from SP and then increment SP by 2 bytes so that it points to the next valid entry in the stack.
4) The endian-ness of x86 makes sense when observing that the stack grows downwards, i.e. the low byte appears in lower memory (i.e. first when reading from low to high).

Key concept: the stack in x86 grows downward into memory toward the base pointer.



Stack errors:

Overflow is where SP goes past the BP. Underflow is where it moves below where it was supposed to. In the below example, the last pop instruction underflows the stack, i.e. removes a value we did not put on. This an important point. Assembly language coding requires you to know what you are doing. There are no safeties. Nothing prevents you from corrupting the stack with underflow errors, for example.

Exercise: Setting IP in code.

Push an address onto the stack.

Issue instruction RET (0xC3). This pops the value on the stack into IP.

Authors: Assad Ebrahim (assad.ebrahim@alum.swarthmore.edu)

```
-a1AC
075A:01AC mov ax, 0102
075A:01AF push ax
075A:01B0 inc ax
075A:01B1 push ax
075A:01B2
-e1b2
```

```
-e1b2
075A:01B2  C1.c1   FF.fe   FE.ff   C3.c3

-u1ac 1b8
075A:01AC B80201        MOV     AX,0102
075A:01AF 50            PUSH    AX
075A:01B0 40            INC     AX
075A:01B1 50            PUSH    AX
075A:01B2 C1            DB      C1
075A:01B3 FEFF          ???     BH
075A:01B5 C3            RET
075A:01B6 D2C0          ROL     AL,CL
075A:01B8 D300          ROL     WORD PTR [BX+SI],CL
```

```
-       rip
IP 01B2
:1ac
-r
AX=0103  BX=0000  CX=0000  DX=0000  SP=FFFA  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=075A  CS=075A  IP=01AC   NV UP EI PL NZ NA PE NC
075A:01AC B80201        MOV     AX,0102
```

```
-t

AX=0102  BX=0000  CX=0000  DX=0000  SP=FFFA  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=075A  CS=075A  IP=01AF   NV UP EI PL NZ NA PE NC
075A:01AF 50            PUSH    AX
-t

AX=0102  BX=0000  CX=0000  DX=0000  SP=FFF8  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=075A  CS=075A  IP=01B0   NV UP EI PL NZ NA PE NC
075A:01B0 40            INC     AX
-t

AX=0103  BX=0000  CX=0000  DX=0000  SP=FFF8  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=075A  CS=075A  IP=01B1   NV UP EI PL NZ NA PE NC
075A:01B1 50            PUSH    AX
-t

AX=0103  BX=0000  CX=0000  DX=0000  SP=FFF6  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=075A  CS=075A  IP=01B2   NV UP EI PL NZ NA PE NC
075A:01B2 C1            DB      C1
```

```
-t

AX=0103  BX=0000  CX=0000  DX=0000  SP=FFF6  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=075A  CS=075A  IP=01B5   NV UP EI PL ZR AC PE NC
075A:01B5 C3            RET
-t

AX=0103  BX=0000  CX=0000  DX=0000  SP=FFF8  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=075A  CS=075A  IP=0103   NV UP EI PL ZR AC PE NC
075A:0103 1033          ADC     [BP+DI],DH                      SS:0000=CD
```

Authors: Assad Ebrahim (assad.ebrahim@alum.swarthmore.edu)

Debug seems to skip over C1 and the next word (i.e. 3 bytes).
But RET (C3) works as expected.

Exercise 3: Let's deliberately create an underflow.

```
073F:0106 pop bx
073F:0107 pop cx
073F:0108 pop dx
073F:0109
```

```
AX=00FD  BX=0000  CX=0000  DX=0000  SP=00F9  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0106   NV UP EI PL NZ NA PO NC
073F:0106 5B            POP     BX
-t

AX=00FD  BX=00FD  CX=0000  DX=0000  SP=00FB  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0107   NV UP EI PL NZ NA PO NC
073F:0107 59            POP     CX
-t

AX=00FD  BX=00FD  CX=00FE  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0108   NV UP EI PL NZ NA PO NC
073F:0108 5A            POP     DX
```

```
073F:0108 5A            POP     DX
-t

AX=00FD  BX=00FD  CX=00FE  DX=0000  SP=00FF  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0109   NV UP EI PL NZ NA PO NC
```

Exercise 4: Create an overflow.

Exercise 5: single byte push/pops?
Not possible. Why? Because the 8086 has a 16-bit (2 byte) word, so the cells in the stack frame are 2 bytes wide. (Try it: push ax; push al; push ah;)

Exercise 6: How far down does the stack frame go?
To the BP=0x0000 upon startup there were FD (=253d) cells available.

Segmented Memory – we cover this in Part 3
How do we view data in a particular segment? We specify in S:O format, where S=seg register and O=offset register.
u1000:0000 shows segment 0x1000 from offset 0x0000.
u1000 shows offset 0x1000 from whichever segment was last specified.

Exercise: What happens during an overlow?
In debug, u passes through a segment and then wraps around the beginning. Check: u0000 then uff00 and run forward with u u u. Observe that this wraps back to u0000.

Exercise: How do the segments overlap? Observe 1695:0020 = 1696:0010, and the two windows are identical except offset by 0x10 in offset space.

Authors: Assad Ebrahim (assad.ebrahim@alum.swarthmore.edu)

```
-u1696:0008 28
1696:0008 0010          ADD     [BX+SI],DL
1696:000A 3C13          CMP     AL,13
1696:000C 41            INC     CX
1696:000D 0F            DB      0F
1696:000E 52            PUSH    DX
1696:000F 42            INC     DX
1696:0010 8BE8          MOV     BP,AX
1696:0012 8CC0          MOV     AX,ES
1696:0014 051000        ADD     AX,0010
1696:0017 0E            PUSH    CS
1696:0018 1F            POP     DS
1696:0019 A30400        MOV     [0004],AX
1696:001C 03060C00      ADD     AX,[000C]
1696:0020 8EC0          MOV     ES,AX
1696:0022 8B0E0600      MOV     CX,[0006]
1696:0026 8BF9          MOV     DI,CX
1696:0028 4F            DEC     DI
```

```
-u1695:0018 38
1695:0018 0010          ADD     [BX+SI],DL
1695:001A 3C13          CMP     AL,13
1695:001C 41            INC     CX
1695:001D 0F            DB      0F
1695:001E 52            PUSH    DX
1695:001F 42            INC     DX
1695:0020 8BE8          MOV     BP,AX
1695:0022 8CC0          MOV     AX,ES
1695:0024 051000        ADD     AX,0010
1695:0027 0E            PUSH    CS
1695:0028 1F            POP     DS
1695:0029 A30400        MOV     [0004],AX
1695:002C 03060C00      ADD     AX,[000C]
1695:0030 8EC0          MOV     ES,AX
1695:0032 8B0E0600      MOV     CX,[0006]
1695:0036 8BF9          MOV     DI,CX
1695:0038 4F            DEC     DI
```

We conclude:
1695:0010 = 1696:0000
1695:0020 = 1696:0010
1695:0100 = 1696:00F0
1695:1000 = 1696:FFF0

Then
1695:0020 = 1696:0010 = 1697:0000
1694:0030 = 1695:0020 = … = 1697:0000
1690:0010 = 1691:0000
1690:0100 = 16A0:0000
1600:0100 = 1610:0000
1600:1000 = 1700:0000
1000:1000 = 1100:0000
1000:FFF0 = 1FFF:0000
1000:FFFF = 1FFF:000F

Authors: Assad Ebrahim (assad.ebrahim@alum.swarthmore.edu)

Conclusion: Segmented Memory Model is based on 0xFFFF (16-bit) addressing of a 20-bit address space (0x10000). It works by taking two 16-bit addresses and overlapping them: the last 3 hex digits of the high address are equivalent to the first 3 hex digits of the low address.
1xxx:xxx0

We are correct:
https://en.wikipedia.org/wiki/X86_memory_segmentation#Real_mode

Exercise: Can DEBUG handle a huge memory model?
So – let's find the start of the edit.com file that we loaded into memory.
Debug edit.com
What are the return values in the registers?

```
C:\DEBUG>debug EDIT.COM
-r
AX=FFFF  BX=0001  CX=10FE  DX=0000  SP=0080  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=16C2  CS=1696  IP=0010   NV UP EI PL NZ NA PO NC
1696:0010 8BE8          MOV     BP,AX
```

The size of the file read is AX+CX, i.e. FFFF+10FE = 65,536+4350 = 69,886 which is exactly the size of edit.com

10F60 lines of code where debug starts
Debug 1696:0000 starts at line 0x10F4F (10F4:000F = 1000:0F4F)
So 0696:0000 should see somewhere around line 0F4F

Experiment: used a Hex Editor to chop down edit.com to FF00 lines (to fill one segment).
Let's see if then debug loads the program correctly. **It doesn't, but that is because it is an EXE (see above).**

How do we get memory allocated for a use outside the tiny memory model, e.g. a separate DS or ES segment?

(See Moore's comments about 1000 lines of code… - and he meant in decimal) Already, this is 16x as big…)
[6] 1x Code: Thoughts on Forth (1999)
http://www.ultratechnology.com/1xforth.htm

```
C:\DEBUG>DEBUG.EXE
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0100   NV UP EI PL NZ NA PO NC
073F:0100 0000          ADD     [BX+SI],AL                      DS:0000=CD
-neditL.com
-l
-r
AX=FFFF  BX=0000  CX=FEEC  DX=0000  SP=0080  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=16C2  CS=1696  IP=0010   NV UP EI PL NZ NA PO NC
1696:0010 0000          ADD     [BX+SI],AL                      DS:0000=CD
```

That didn't work.

Let's try just F000 lines of code.

What about loading any file? Hello1.com is 88 bytes.

Authors: Assad Ebrahim (assad.ebrahim@alum.swarthmore.edu)

```
C:\DEBUG>DEBUG.EXE
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0100   NV UP EI PL NZ NA PO NC
073F:0100 5A            POP     DX
-nhello1.com
-l
-r
AX=FFFF  BX=0000  CX=0058  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=075A  CS=075A  IP=0100   NV UP EI PL NZ NA PO NC
075A:0100 B248          MOV     DL,48
```

Note: CS = SS and SP is given 65k bytes of space. So the code grows up and the stack grows down.

https://en.wikipedia.org/wiki/Memory_segmentation
https://en.wikipedia.org/wiki/X86_memory_segmentation

FREQUENTLY ASKED QUESTIONS

FAQ1. What are the memory models of the 8086?

Memory Models

8086 has 6 memory models.

Tiny memory model has everything in one 64k segment, i.e. CS=SS=DS=ES. Note this only works in MS-DOS.

Small memory model has one code segment and one data segment, but they could be different from each other. The key is that all pointers are NEAR pointers by default, i.e. offsets are baselined by convention. Typically SS=CS. Typically ES=DS.

Then we have two intermediate models: multiple data segments (DS and ES) but one code segment (compact model), or multiple code segments and one data segment (medium model).

Finally we have multiple code and data segments (large model).

And then the huge model is where code or data does not fit and extends across multiple segments.

Pointers within a segment are NEAR pointers (16-bits). Pointers beyond a segment can be NEAR if they are offset pointers with an implicit segment baseline, otherwise they must be FAR pointers (32-bits).

References:

[1] http://www.c-jump.com/CIS77/ASM/Directives/D77_0030_models.htm

[2] http://www.scit.wlv.ac.uk/~in8297/CP4044/cbook/chap6/chap6.msdos.memory.html

FAQ2. What happens to the memory state when Debug loads a file?

Loading hello1.com (88 bytes) keeps debug in the tiny memory model.

```
-nhello1.com
-l
-r
AX=FFFF  BX=0000  CX=0058  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=075A  CS=075A  IP=0100   NV UP EI PL NZ NA PO NC
075A:0100 B248          MOV     DL,48
```

Loading up tasm.com (59,016 bytes, 0xE688 bytes) still keeps debug in tiny memory model.

Authors: Assad Ebrahim (assad.ebrahim@alum.swarthmore.edu)

```
-ntasm.com
-l
-r
AX=FFFF  BX=0000  CX=E688  DX=0000  SP=FFFE  BP=1200  SI=0500  DI=0600
DS=075A  ES=075A  SS=075A  CS=075A  IP=0100   NV UP EI NG NZ NA PO NC
075A:0100 B430          MOV     AH,30
```

Loading up an EXE such as edit.com (only way to tell its an EXE is to look at first few lines with a text editor – it has the MZ magic numbers at the start). Because it's an EXE it will adjust the segments etc. as EXEs don't run in tiny memory model. edit.com also is >64k bytes. (X bytes).

```
C:\DEBUG>DEBUG.EXE
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0100   NV UP EI PL NZ NA PO NC
073F:0100 0000          ADD     [BX+SI],AL                      DS:0000=CD
-nedit.com
-l
-r
AX=FFFF  BX=0001  CX=10FE  DX=0000  SP=0080  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=16C2  CS=1696  IP=0010   NV UP EI PL NZ NA PO NC
1696:0010 8BE8          MOV     BP,AX
```

The system switches out of tiny model – all segment registers are given new memory to point to, CS=SS is retained, i.e. code and stack segments continue to be together, and data / extra segments are moved elsewhere to make room.

FAQ3. What happens when a file greater than 65,536 bytes (=2^16) is loaded into debug (e.g. edit.com = 69,886 bytes)?

First, be careful. Existing files, even though stated as .com, may not be. Inspect them with a text editor. If they are EXE files they will start with MZ. Debug will try to load them using the EXE information in their header that sets segments, etc. Edit.com is an EXE so will not load as you expect (see Appendix 1 for details on the EXE header).

```
C:\DEBUG>debug
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0100   NV UP EI PL NZ NA PO NC
073F:0100 5A            POP     DX
```

References:

Raymond Chen

https://blogs.msdn.microsoft.com/oldnewthing/20110314-00/?p=11233

https://blogs.msdn.microsoft.com/oldnewthing/20080324-00/?p=23033/

FAQ4. What makes an EXE file executable? Vs. a COM file?

FAQ5. Does every DOS program get allocated 64KB memory space to operate in? How do programs request more? Was MS-DOS every multi-tasking? Multi-application? What about when Windows started this?

FAQ5. What is the stack useful for?

The stack is an area of memory that is available for fast access – storing to and loading from.

It provides for convenient storage for intermediate data, since the number of registers are often restricted (e.g. 4 primary registers in 8086, AX to DX)

Authors: Assad Ebrahim (assad.ebrahim@alum.swarthmore.edu)

The stack is used in subroutines to provide a clean memory without the function having to deal with the contents of the calling routine. So the function implementation can store the register values upon being launched, and restore them upon exit, thereby cleaning up after itself.

The LIFO design of a stack allows for nesting such operations, i.e. a sub-subroutine can store its calling context onto the stack, use the stack itself, clean up its own usage, return and restore the calling context, which can continue, and as long as it also cleans what it has added, then it can be sure that when it returns, the original context will be restored intact.

This is a very powerful programming paradigm.

In another direction, stacks are in fact all that one needs --- one does not need memory registers at all.

Forth is a language designed by Chuck Moore (b1938) on this concept. It creates a machine abstraction in software that has only a stack memory.

A Forth chip is a chip that implements this concept.

There are a few historical Forth chips that were used in the 1960s space exploration context and worked very well. Currently, Forth chips are part of massively parallel, low power, inexpensive array processing (see GreenArrays, 144-core asynchronous chip, consuming 100nW when idle).

Author: Assad Ebrahim
assad.ebrahim@alum.swarthmore.edu

Objective. This worksheet uses DEBUG to explore, illustrate, discover key points about Computer Architecture and Programming. This continues from Part Two which explores the x86 stack, stack pointers (SP/BP), and stack instructions (PUSH/POP). This worksheet explores the memory segments.

Examining a real Machine Language program (deliverable: present it as a structured assembly language program, with comments)

```
--
  B013CD1033C0BFB001B9007DF3ABBAC803EE42FEC980/~~~~~~~~\       The Bitripper
FB3C7305803C04EB0880FF3C73038C7048AC3EE8AC7/'           `\~~~\
EE32C0EEE2E3B1C88106AC01E9628006AC01628116AE`\           /'   )
  011936A1AE0133D2BB4001F7F38BF2FE8C707DE2DDBE`~~/   \~~'    / Coders do it
   F102BFB17EB162BA3E018A9CC0FE8A44FF03D88A4401/`     )      `\  bit by bit
  03D88A84400103D8C1EB02881D46474A75E246464747/       /        )
 E2D9BEB27EBFB201B97E3E5157F3A55E6800A007BF02(_______/        /
          7D59F3A51E07B401CD16748CB80300CD10C3c4urself`.____/'
--
```

Type the following 8086 machine code into debug using the e100 instruction…

```
B0 13 CD 10 33 C0 BF B0   01 B9 00 7D F3 AB BA C8
03 EE 42 FE C9 80 FB 3C   73 05 80 C3 04 EB 08 80
FF 3C 73 03 80 C7 04 8A   C3 EE 8A C7 EE 32 C0 EE
E2 E3 B1 C8 81 06 AC 01   E9 62 80 06 AC 01 62 81

16 AE 01 19 36 A1 AE 01   33 D2 BB 40 01 F7 F3 8B
F2 FE 8C 70 7D E2 DD BE   F1 02 BF B1 7E B1 62 BA
3E 01 8A 9C C0 FE 8A 44   FF 03 D8 8A 44 01 03 D8
8A 84 40 01 03 D8 C1 EB   02 88 1D 46 47 4A 75 E2

46 46 47 47 E2 D9 BE B2   7E BF B2 01 B9 7E 3E 51
57 F3 A5 5E 68 00 A0 07   BF 02 7D 59 F3 A5 1E 07
B4 01 CD 16 74 8C B8 03   00 CD 10 C3
```

Source: http://www.starman.vertcomp.com/asm/fire/Fire.html

e100

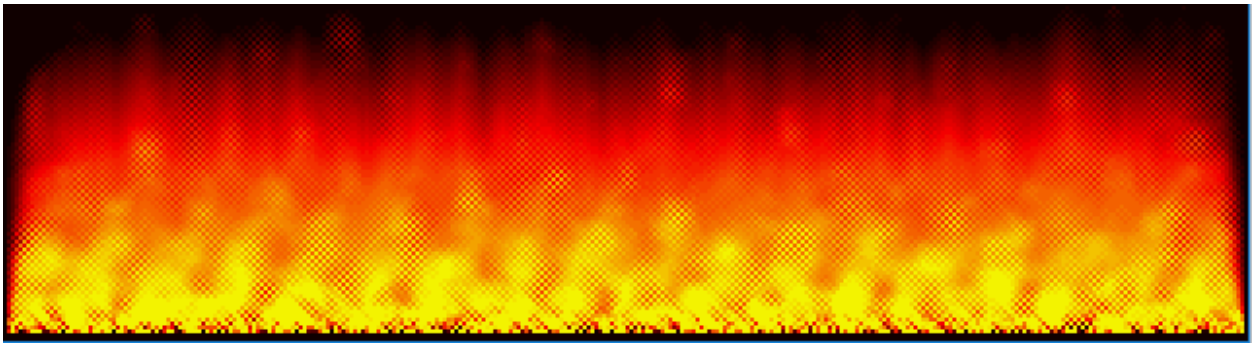
Dump the code memory (command d)

```
-       d100 1ab
073F:0100  B0 13 CD 10 33 C0 BF B0-01 B9 00 7D F3 AB BA C8
073F:0110  03 EE 42 FE C9 80 FB 3C-73 05 80 C3 04 EB 08 80
073F:0120  FF 3C 73 03 80 C7 04 8A-C3 EE 8A C7 EE 32 C0 EE
073F:0130  E2 E3 B1 C8 81 06 AC 01-E9 62 80 06 AC 01 62 81
073F:0140  16 AE 01 19 36 A1 AE 01-33 D2 BB 40 01 F7 F3 8B
073F:0150  F2 FE 8C 70 7D E2 DD BE-F1 02 BF B1 7E B1 62 BA
073F:0160  3E 01 8A 9C C0 FE 8A 44-FF 03 D8 8A 44 01 03 D8
073F:0170  8A 84 40 01 03 D8 C1 EB-02 88 1D 46 47 4A 75 E2
073F:0180  46 46 47 47 E2 D9 BE B2-7E BF B2 01 B9 7E 3E 51
073F:0190  57 F3 A5 5E 68 00 A0 07-BF 02 7D 59 F3 A5 1E 07
073F:01A0  B4 01 CD 16 74 8C B8 03-00 CD 10 C3
```

Author: Assad Ebrahim
assad.ebrahim@alum.swarthmore.edu

```
-       r
AX=0000  BX=0000  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0100   NV UP EI PL NZ NA PO NC
073F:0100 B013          MOV     AL,13
```

Run it (g command).



Exercise: can use u command to unassemble the code

`B0 13 = mov al, 13h`
CD 10 = call interrupt 10h
33 = xor?

```
-U100
073F:0100 B013          MOV     AL,13
073F:0102 CD10          INT     10
073F:0104 33C0          XOR     AX,AX
073F:0106 BFB001        MOV     DI,01B0
073F:0109 B9007D        MOV     CX,7D00
073F:010C F3            REPZ
073F:010D AB            STOSW
073F:010E BAC803        MOV     DX,03C8
073F:0111 EE            OUT     DX,AL
073F:0112 42            INC     DX
073F:0113 FEC9          DEC     CL
073F:0115 80FB3C        CMP     BL,3C
073F:0118 7305          JNB     011F
073F:011A 80C304        ADD     BL,04
073F:011D 3B08          CMP     CX,[BX+SI]
073F:011F 80FF3C        CMP     BH,3C
```

```
-U
073F:0122 7303          JNB     0127
073F:0124 80C704        ADD     BH,04
073F:0127 8AC3          MOV     AL,BL
073F:0129 EE            OUT     DX,AL
073F:012A 8AC7          MOV     AL,BH
073F:012C EE            OUT     DX,AL
073F:012D 32C0          XOR     AL,AL
073F:012F EE            OUT     DX,AL
073F:0130 E2E3          LOOP    0115
073F:0132 B1C8          MOV     CL,C8
073F:0134 8106AC01E962  ADD     WORD PTR [01AC],62E9
073F:013A 8006AC0162    ADD     BYTE PTR [01AC],62
073F:013F 8116AE011936  ADC     WORD PTR [01AE],3619
```

Author: Assad Ebrahim
assad.ebrahim@alum.swarthmore.edu

```
-U
073F:0145 A1AE01        MOV     AX,[01AE]
073F:0148 33D2          XOR     DX,DX
073F:014A BB4001        MOV     BX,0140
073F:014D F7F3          DIV     BX
073F:014F 8BF2          MOV     SI,DX
073F:0151 FE8C707D      DEC     BYTE PTR [SI+7D70]
073F:0155 E2DD          LOOP    0134
073F:0157 BEF102        MOV     SI,02F1
073F:015A BFB17E        MOV     DI,7EB1
073F:015D B162          MOV     CL,62
073F:015F BA3E01        MOV     DX,013E
073F:0162 8A9CC0FE      MOV     BL,[SI+FEC0]
```

```
-U
073F:0166 8A44FF        MOV     AL,[SI-01]
073F:0169 03D8          ADD     BX,AX
073F:016B 8A4401        MOV     AL,[SI+01]
073F:016E 03D8          ADD     BX,AX
073F:0170 8A844001      MOV     AL,[SI+0140]
073F:0174 03D8          ADD     BX,AX
073F:0176 C1            DB      C1
073F:0177 EB02          JMP     017B
073F:0179 881D          MOV     [DI],BL
073F:017B 46            INC     SI
073F:017C 47            INC     DI
073F:017D 4A            DEC     DX
073F:017E 75E2          JNZ     0162
073F:0180 46            INC     SI
073F:0181 46            INC     SI
073F:0182 47            INC     DI
073F:0183 47            INC     DI
073F:0184 E2D9          LOOP    015F
```

```
-U
073F:0186 BEB27E        MOV     SI,7EB2
073F:0189 BFB201        MOV     DI,01B2
073F:018C B97E3E        MOV     CX,3E7E
073F:018F 51            PUSH    CX
073F:0190 57            PUSH    DI
073F:0191 F3            REPZ
073F:0192 A5            MOVSW
073F:0193 5E            POP     SI
073F:0194 68            DB      68
073F:0195 00A007BF      ADD     [BX+SI+BF07],AH
073F:0199 027D59        ADD     BH,[DI+59]
073F:019C F3            REPZ
073F:019D A5            MOVSW
073F:019E 1E            PUSH    DS
073F:019F 07            POP     ES
073F:01A0 B401          MOV     AH,01
073F:01A2 CD16          INT     16
073F:01A4 748C          JZ      0132
```

Note, line 0x194 – this is where DEBUG unassembler doesn't get this write, but stepping through the program does.
68 00 A0 pushes 0xA000 onto stack

Author: Assad Ebrahim
assad.ebrahim@alum.swarthmore.edu

07 pop es
BF 027D mov di, 0x7d02
59 pop cx

```
073F:01A6 B80300        MOV     AX,0003
073F:01A9 CD10          INT     10
073F:01AB C3            RET
```

AC bytes (0x100 to 0x1ab)

Advanced Tracing
Some tracing skills will be valuable
T steps once
TN steps N steps forward, showing each one e.g. T10
Gxxxx run to breakpoint

Write to file.
nFIRE.COM
rbx
0
rcx
AC
W

FIRE.COM

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	B0	13	CD	10	33	C0	BF	B0	01	B9	00	7D	F3	AB	BA	C8
00000010	03	EE	42	FE	C9	80	FB	3C	73	05	80	C3	04	3B	08	80
00000020	FF	3C	73	03	80	C7	04	8A	C3	EE	8A	C7	EE	32	C0	EE
00000030	E2	E3	B1	C8	81	06	AC	01	E9	62	80	06	AC	01	62	81
00000040	16	AE	01	19	36	A1	AE	01	33	D2	BB	40	01	F7	F3	8B
00000050	F2	FE	8C	70	7D	E2	DD	BE	F1	02	BF	B1	7E	B1	62	BA
00000060	3E	01	8A	9C	C0	FE	8A	44	FF	03	D8	8A	44	01	03	D8
00000070	8A	84	40	01	03	D8	C1	EB	02	88	1D	46	47	4A	75	E2
00000080	46	46	47	47	E2	D9	BE	B2	7E	BF	B2	01	B9	7E	3E	51
00000090	57	F3	A5	5E	68	00	A0	07	BF	02	7D	59	F3	A5	1E	07
000000A0	B4	01	CD	16	74	8C	B8	03	00	CD	10	C3				

Exercise: How to get this to load from a file?
When loading this from file --- it doesn't work…
What's the bug? It is this: the registers are not cleared by the program. So in Debug after loading the file, AX and CX are messy. Setting in DEBUG AX=0000 gets it running but with a monochrome yellow palette.
What's the long-term fix?
If instead we replace MOV AL,13 with MOV AX,13 that should work, right? Wrong. MOV AX,13 is 3 byte command (B8 13 00) because the parameter is 0x0013. Whereas MOV AL,13 is a 2 byte command B0 13.
Ok, what about using a Hex editor insert an extra byte after the 2nd byte?

Author: Assad Ebrahim
assad.ebrahim@alum.swarthmore.edu

That works --- to a point. The rest of the program is shifted down by a byte, so any JMP or data references will be off! This illustrates another issue with machine language. It is very difficult to maintain because everything is hardcoded. (NOP instruction 90 doesn't help here as it just adds extra padding – I don't think word boundaries is the problem.)

Exercise: Translate this to assembly language and compile as a flat file (use NASM) --- this preserves the size, but presents it in a maintainable manner.
This is what assembly did for machine languages (1GL: https://en.wikipedia.org/wiki/First-generation_programming_language) – made it human readable, editable, maintainable but still tied to a particular CPU (processing unit) (2GL: https://www.techopedia.com/definition/24305/second-generation-programming-language-2gl). This required additional tools to assemble (and possibly link) the code.
Exercise2: Translate this to Forth --- this now makes it portable, because Forth, like all early higher level languages (FORTRAN, ALGOL, COBOL, BASIC, and C – these were the first 6/big6 3GLs - https://en.wikipedia.org/wiki/Third-generation_programming_language)

https://en.wikipedia.org/wiki/Fourth-generation_programming_language

https://en.wikipedia.org/wiki/History_of_programming_languages

What do we learn from understanding the fire.com code?

Structured programming:
Use Labels instead of jump memory for relocatable code hooks instead of manually having to update all links and jumps and relative references by hand.

Concept: asserts
Write the assertions into comments in code.

Simplification
Chuck Moore – factor. To do that you have to understand what is wanted to be done and then ask whether there is a more optimal way

Author: Assad Ebrahim
assad.ebrahim@alum.swarthmore.edu

Simpler Code

```
073F:0100 B81300          MOV     AX,0013
073F:0103 CD10            INT     10
073F:0105 FC              CLD
073F:0106 31C0            XOR     AX,AX
073F:0108 BFB001          MOV     DI,01B0
073F:010B B9007D          MOV     CX,7D00
073F:010E F3              REPZ
073F:010F AB              STOSW
073F:0110 BAC803          MOV     DX,03C8
073F:0113 EE              OUT     DX,AL
073F:0114 42              INC     DX
073F:0115 FEC9            DEC     CL
073F:0117 3C3C            CMP     AL,3C
073F:0119 7302            JNB     011D
073F:011B 0404            ADD     AL,04
073F:011D EE              OUT     DX,AL
073F:011E EE              OUT     DX,AL
073F:011F 50              PUSH    AX
```

```
073F:0120 30C0            XOR     AL,AL
073F:0122 EE              OUT     DX,AL
073F:0123 58              POP     AX
073F:0124 E2F1            LOOP    0117
073F:0126 B1C8            MOV     CL,C8
073F:0128 8106AC01E962    ADD     WORD PTR [01AC],62E9
073F:012E 8006AC0162      ADD     BYTE PTR [01AC],62
073F:0133 8116AE011936    ADC     WORD PTR [01AE],3619
073F:0139 A1AE01          MOV     AX,[01AE]
073F:013C 31D2            XOR     DX,DX
073F:013E BB4001          MOV     BX,0140
```

```
073F:0141 F7F3            DIV     BX
073F:0143 89D6            MOV     SI,DX
073F:0145 FE8C707D        DEC     BYTE PTR [SI+7D70]
073F:0149 E2DD            LOOP    0128
073F:014B BEF102          MOV     SI,02F1
073F:014E BFB17E          MOV     DI,7EB1
073F:0151 B162            MOV     CL,62
073F:0153 BA3E01          MOV     DX,013E
073F:0156 8A9CC0FE        MOV     BL,[SI+FEC0]
073F:015A 8A44FF          MOV     AL,[SI-01]
073F:015D 01C3            ADD     BX,AX
073F:015F 8A4401          MOV     AL,[SI+01]
```

Author: Assad Ebrahim
assad.ebrahim@alum.swarthmore.edu

```
073F:0162 01C3          ADD     BX,AX
073F:0164 8A844001      MOV     AL,[SI+0140]
073F:0168 01C3          ADD     BX,AX
073F:016A C1            DB      C1
073F:016B EB02          JMP     016F
073F:016D 881D          MOV     [DI],BL
073F:016F 46            INC     SI
073F:0170 47            INC     DI
073F:0171 4A            DEC     DX
073F:0172 75E2          JNZ     0156
073F:0174 46            INC     SI
073F:0175 46            INC     SI
073F:0176 47            INC     DI
073F:0177 47            INC     DI
073F:0178 E2D9          LOOP    0153
073F:017A BEB27E        MOV     SI,7EB2
073F:017D BFB201        MOV     DI,01B2
073F:0180 B9733E        MOV     CX,3E73
073F:0183 51            PUSH    CX
073F:0184 57            PUSH    DI
073F:0185 F3            REPZ
073F:0186 A5            MOVSW
073F:0187 5E            POP     SI
073F:0188 BF027D        MOV     DI,7D02
073F:018B 59            POP     CX
073F:018C 68            DB      68
073F:018D 00A007F3      ADD     [BX+SI+F307],AH
073F:018F 07            POP     ES
073F:0190 F3            REPZ
073F:0191 A5            MOVSW
073F:0192 1E            PUSH    DS
073F:0193 07            POP     ES
073F:0194 B401          MOV     AH,01
073F:0196 CD16          INT     16
073F:0198 748C          JZ      0126
073F:019A B80300        MOV     AX,0003
073F:019D CD10          INT     10
073F:019F 68            DB      68
073F:01A0 0001          ADD     [BX+DI],AL
073F:01A2 C3            RET
```

END

Code segment (new) – 0x100 to 0x1A2 bytes, i.e. 0xA3 bytes (=163 bytes)

```
-d100 1a2
073F:0100  B8 13 00 CD 10 FC 31 C0-BF B0 01 B9 00 7D F3 AB
073F:0110  BA C8 03 EE 42 FE C9 3C-3C 73 02 04 04 EE EE 50
073F:0120  30 C0 EE 58 E2 F1 B1 C8-81 06 AC 01 E9 62 80 06
073F:0130  AC 01 62 81 16 AE 01 19-36 A1 AE 01 31 D2 BB 40
073F:0140  01 F7 F3 89 D6 FE 8C 70-7D E2 DD BE F1 02 BF B1
073F:0150  7E B1 62 BA 3E 01 8A 9C-C0 FE 8A 44 FF 01 C3 8A
073F:0160  44 01 01 C3 8A 84 40 01-01 C3 C1 EB 02 88 1D 46
073F:0170  47 4A 75 E2 46 46 47 47-E2 D9 BE B2 7E BF B2 01
073F:0180  B9 73 3E 51 57 F3 A5 5E-BF 02 7D 59 78 00 A0 07
073F:0190  F3 A5 1E 07 B4 01 CD 16-74 8C B8 03 00 CD 10 68
073F:01A0  00 01 C3
```

Author: Assad Ebrahim
assad.ebrahim@alum.swarthmore.edu

```
Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000   B8 13 00 CD 10 FC 31 C0 BF B0 01 B9 00 7D F3 AB
00000010   BA C8 03 EE 42 FE C9 3C 3C 73 02 04 04 EE EE 50
00000020   30 C0 EE 58 E2 F1 B1 C8 81 06 AC 01 E9 62 80 06
00000030   AC 01 62 81 16 AE 01 19 36 A1 AE 01 31 D2 BB 40
00000040   01 F7 F3 89 D6 FE 8C 70 7D E2 DD BE F1 02 BF B1
00000050   7E B1 62 BA 3E 01 8A 9C C0 FE 8A 44 FF 01 C3 8A
00000060   44 01 01 C3 8A 84 40 01 01 C3 C1 EB 02 88 1D 46
00000070   47 4A 75 E2 46 46 47 47 E2 D9 BE B2 7E BF B2 01
00000080   B9 73 3E 51 57 F3 A5 5E BF 02 7D 59 78 00 A0 07
00000090   F3 A5 1E 07 B4 01 CD 16 74 8C B8 03 00 CD 10 68
000000A0   00 01 C3
```

Color Palette Generator

Exercise: Write a simulator in Forth

Generator.

```
075A:01A0
        E9 62 00 00                 .b..
-t

AX=003C  BX=0000  CX=00C8  DX=03C9  SP=F
FFE  BP=0000  SI=0000  DI=FBB0
DS=075A  ES=075A  SS=075A  CS=075A  IP=0
133   NV UP EI PL NZ NA PE CY
075A:0133 8116AE011936  ADC     WORD PTR
 [01AE],3619                DS:01AE=0000
-d01ac 01af
075A:01A0
        4B 62 00 00                 Kb..
-       t

AX=003C  BX=0000  CX=00C8  DX=03C9  SP=F
FFE  BP=0000  SI=0000  DI=FBB0
DS=075A  ES=075A  SS=075A  CS=075A  IP=0
139   NV UP EI PL NZ NA PO NC
075A:0139 A1AE01          MOV       AX,[01AE
]                          DS:01AE=361A
-d01ac 01af
075A:01A0
        4B 62 1A 36                 Kb.6
-
```

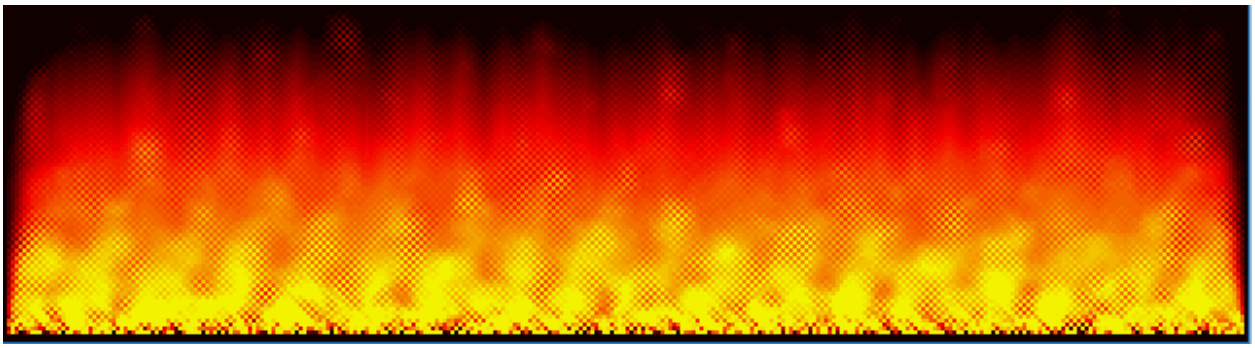
```
AX=002B  BX=0140  CX=00C8  DX=005A  SP=F
FFE  BP=0000  SI=005A  DI=FBB0
DS=075A  ES=075A  SS=075A  CS=075A  IP=0
145   NV UP EI PL ZR NA PE NC
075A:0145 FE8C707D        DEC       BYTE PTR
 [SI+7D70]                  DS:7DCA=00
-
```

Reconstructing the palette… bit.fs (scripts/binary)

```
fire-palette
16    32    48    64    80    96    112   128   144   160   176   192
208   224   240   240   240   240   240   240   240   240   240   240
240   240   240   240   240   240   240   240   240   240   240   240
```

Author: Assad Ebrahim
assad.ebrahim@alum.swarthmore.edu

?? How do you get this then?



There is no blue – confirmed.
But much of this is red/orange/yellow, which is red @ 3C and green varying from 0 to 3C

The Fire-Generator

Exercise: Write a simulator in Forth

```
> table(d$V2)

  2   3   4   7  10  12  15  16  17  18  22  23  24  26  29  30  32  34  36  37
  1   1   1   1   3   1   1   1   1   1   1   1   2   1   1   1   1   1   1   1
 38  43  44  45  49  50  51  53  57  60  62  63  64  66  69  70  71  72  76  78
  2   1   1   2   1   2   1   1   2   2   1   1   1   1   1   1   1   2   1   1
 79  80  84  85  87  90  91  93  96  97  99 100 103 105 107 111 112 113 114 118
  1   1   2   2   1   1   1   2   1   1   1   2   1   2   1   2   1   1   1   1
119 122 123 125 126 127 131 132 134 137 138 140 145 146 147 152 154 155 158 159
  2   1   1   1   1   1   1   1   2   1   1   3   1   1   1   3   1   1   1   1
161 162 165 166 167 169 171 173 174 179 181 182 185 187 188 190 191 192 194 195
  1   1   1   1   1   1   1   1   2   2   1   1   1   1   1   1   1   1   1   1
199 200 201 202 206 207 208 209 213 214 216 221 223 225 226 228 229 233 235 240
  1   1   1   1   1   1   1   1   2   2   1   2   1   1   1   1   2   1   2   1
241 242 243 247 249 250 253 254 255 256 260 261 263 266 268 270 275 276 278 280
  2   1   1   2   1   1   1   1   1   1   1   1   2   1   3   1   2   1   1   2
282 283 285 287 288 290 291 294 295 297 301 302 303 309 310 311 312 315 316 318
  1   1   1   1   1   1   1   1   1   2   1   2   2   1   1   1   1   2   1   2
```

Author: Assad Ebrahim
assad.ebrahim@alum.swarthmore.edu

```
> stem(d$V2)

  The decimal point is 1 digit(s) to the right of the |

   0 | 234700025678
   2 | 2344690246788
   4 | 34559001377
   6 | 00234690122689
   8 | 0445570133679
  10 | 00355711234899
  12 | 23567124478
  14 | 0005672224589
  16 | 125679134499
  18 | 12578012459
  20 | 012678933446
  22 | 11356899355
  24 | 0112377903456
  26 | 0133688805568
  28 | 0023578014577
  30 | 12233901255688
```

First 10 residues are

```
1   5A,90,<2> 624B 361A
2   B3,179,<2> C596 6C33
3   10C,268,<2> 28E1 A24C
4   26,38,<2> 8B2C D866
5   BF,191,<2> EE77 E7F
6   118,280,<2> 51C2 4498
7   32,50,<2> B40D 7AB2
8   8C,140,<2> 1658 B0CC
9   E5,229,<2> 79A3 E6E5
10  3E,62,<2> DCEE 1CFE
```

```
-
-h 7d70 26
7D96   7D4A
-h 7d70 32
7DA2   7D3E
-d 7d90 7dAF
075A:7D90   00 00 00 00 00 00 FF 00-00 00
  00 00 00 00 00 00     ..........
075A:7DA0   00 00 FF 00 00 00 00 00-00 00
  00 00 00 00 00 00     ..........
-
```

Let's fast forward to the end of the loop

```
AX=002B  BX=0140  CX=00C8  DX=005A  SP=FFFE  BP=0000  SI=005A  DI=FBB0
DS=075A  ES=075A  SS=075A  CS=075A  IP=0149   NV UP EI NG NZ AC PE NC
075A:0149 E2DD          LOOP    0128
-g 14b

AX=0036  BX=0140  CX=0000  DX=0055  SP=FFFE  BP=0000  SI=0055  DI=FBB0
DS=075A  ES=075A  SS=075A  CS=075A  IP=014B   NV UP EI NG NZ NA PO NC
075A:014B BEF102        MOV     SI,02F1
```

Notice this matches the last numbers in our simulator:

Residue = 0x55

And 0x36 * 0x140 + 0x55 [ax x bx + dx] = 0x43D5

Author: Assad Ebrahim
assad.ebrahim@alum.swarthmore.edu

```
190   17,23,<2> 68AA 26D7
191   70,112,<2> CBF5 5CF0
192   CA,202,<2> 2E40 930A
193   123,291,<2> 918B C923
194   3C,60,<2> F4D6 FF3C
195   D6,214,<2> 5721 3556
196   12F,303,<2> BA6C 6B6F
197   48,72,<2> 1DB7 A188
198   A2,162,<2> 8002 D7A2
199   13C,316,<2> E24D DBC
200   55,85,<2> 4598 43D5
```

By the time we're done with the loop, we have 152d (=98h) occurring 3 times… there's the FD (from 00 to FF, FE, FD is 3 steps…)

```
-h 7d70 98
7E08  7CD8
-d7e00 7e1f
075A:7E00  00 FF FF FF 00 00 00 00-FD 00
 FF FF 00 00 FF FF  ................
075A:7E10  00 FF FF 00 00 FF FF FF-00 FF
 00 FF 00 FF FE 00  ................
```

I think this is the bottom row of the VGA screen – 7D70, and all the residues will appear yellow (doesn't matter how many times they are hit, as the upper end of the palette is all yellow), and the ones NOT hit will be black, which will be the flicker on the fire.

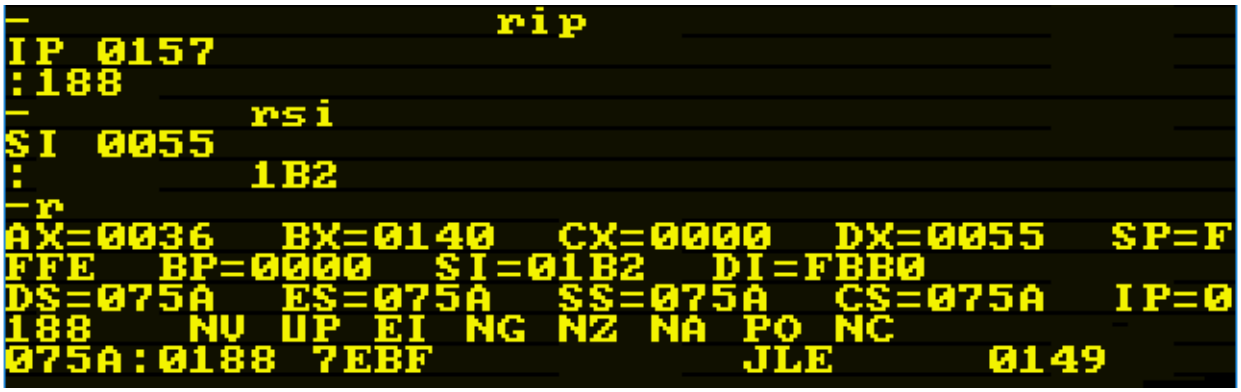
Let's test by short-circuiting the code…
We will pop 0x1B2 onto SI and then start at IP=0x188.
Because it's assembly code, we don't have to worry about loops not completed etc.

```
073F:0187 5E            POP     SI
073F:0188 BF027D        MOV     DI,7D02
073F:018B 59            POP     CX
073F:018C 68            DB      68
073F:018D 00A007F3      ADD     [BX+SI+F307],AH
```

```
073F:018F 07            POP     ES
073F:0190 F3            REPZ
073F:0191 A5            MOVSW
073F:0192 1E            PUSH    DS
073F:0193 07            POP     ES
073F:0194 B401          MOV     AH,01
073F:0196 CD16          INT     16
073F:0198 748C          JZ      0126
073F:019A B80300        MOV     AX,0003
073F:019D CD10          INT     10
073F:019F 68            DB      68
073F:01A0 0001          ADD     [BX+DI],AL
073F:01A2 C3            RET
```

END

Author: Assad Ebrahim
assad.ebrahim@alum.swarthmore.edu



Diamond Filter

To step through the diamond filter run command td0 to go 10x through the loop (13d=0x0d instructions per loop)

in first pass through diamond filter, everything stays 0 until row 99 (N-1) when filter touches fire gen row, and propagates fire upward by 1 row

each successive loop through diamond filter, propagates fire further upwards by 1 row, also merging flames with neighbours flames, so more likely to be yellow

Exercise: Write a diamond-filter in Forth to simulate/visualize/verify what's happening to generate the image.

```
95  \ create last row of fire (row 99, 0-indexed)
96  : tuck ( a b -- b a b ) swap over ;
97  1 constant cells
98  : c!+ ( val ptr -- ptr' ) tuck c! 1 + ;      \ store and increment pointer
99  : c@+ ( ptr -- ptr' val ) dup 1 + swap c@ ;  \ fetch and increment pointer
100 : --c! ( ptr -- ptr ) dup c@ 1 - swap c! ;     \ decrement value in memory
101 : ++c! ( ptr -- ptr ) dup c@ 1 + swap c! ;     \ increment value in memory
102 \ marker free-all      \ saves current place in heap with action to later restore heap to this place when forget is called
103
104 create r99   $13e allot
105 : init-row ( row -- ) $c8 0 do 0 over c! 1 + loop drop ;
106 : start-fire ( -- ) 0 0 $c8 0 do fire-gen r99 + --c! loop 2drop ;
107 : show-fire-row ( row -- ) cr $c8 0 do c@+ ., i 8xCR loop drop ;
108
109 : simulate-fire-starting  ( -- )  \ populate base row of the fire
110     r99 init-row
111     r99 show-fire-row
112     start-fire
113     r99 show-fire-row ;
```

Author: Assad Ebrahim
assad.ebrahim@alum.swarthmore.edu

```
start-fire  ok
  ok
r99 show-fire-row
r99 show-fire-row
0,0,FF,FF,FF,0,0,FF,
0,0,FD,0,FF,0,0,FF,
FF,FF,FF,0,0,0,FF,FF,
FE,0,FF,0,0,FF,FF,0,
FF,0,FF,0,FF,FF,FE,0,
0,0,0,FF,FF,FE,0,0,
0,FF,FE,FF,0,FF,0,0,
0,FE,0,0,FE,0,FF,FF,
FF,0,FF,0,0,FF,FF,FF,
FE,0,0,0,FF,0,FF,FF,
FF,0,0,0,FE,FE,0,FF,
0,0,FF,FF,0,FE,0,0,
FF,FF,0,FF,FE,0,0,FF,
0,FE,0,FF,0,0,0,FE,
FF,FF,FF,0,0,0,FF,FE,
0,0,FF,FF,0,FF,FF,FF,
0,0,0,FF,FF,0,FE,0,
0,FF,FF,0,FD,0,0,0,
0,FF,FF,FF,0,0,0,0,
FD,0,FF,FF,0,0,FF,FF,
0,FF,FF,0,0,FF,FF,FF,
0,FF,0,FF,0,FF,FE,0,
0,0,0,FE,0,FF,FF,0,
0,FF,0,FF,FF,0,FF,FF,
FF,0,FF,FF,0,0,0,FF,
 ok
```

Notice the first row matches up with the table: 0, 1 no hits, 2,3,4 have 1 hit each, 5,6 no hits, 7 jas 1 hit.

But how do the rest of the fire rows get formed?
The next pass from top to bottom should encounter all zeroes until row N-2 (row 98)… but how to verify that in the debugger? Two loops / inner and out. Would take too long to step through, even with two actions (gxxx and t5 to get inside. There is another option. After checking

Write-up

Main outline (the * are the interesting bits)

- switch screen driver to vga 18-bit color mode, 320x200 (=64k bytes). In hex $200x$140 (=$F0000 bytes)
- clear (to 0s) memory buffer for vga data storage (will be two half screens, of $8000 = 32k bytes each, primary screen, and transformation screen
  - ??? can this be optimized further??? because its a diamond filter moving downwards in memory, the raster connection is tied only to two rows, so only need to save two rows, and then copy these back... (more complicated, but would save space, and time)

* select the vga color palette: 256 x 18-bit colors (6 bits each of R,G,B -- rshift each by 2 bits to see as 24-bit RGB)

- ??? in what sequence are they loaded? sequence matters for the fire shading/gradient.

* run fire animation in a loop, checking for a keypress, at which point stop

- restore screen driver to text mode

Main outline in Forth, with * highlighting the key words

```
: fire  ( -- ) switchto-graphics  clear-mem-buf  choose-fire-pallette*
          BEGIN
            make-fire*  keypress?
          UNTIL
         switchto-text ;
: make-fire* ( -- )  stoke-firebase*  diamond-filter-topdown*  bit-blit-to-screen ;
: stoke-firebase*  ( -- )  xwidth 0 DO  pseudo-random-residue  adjust-color LOOP ;
: diamond-filter-topdown*  ( -- )
```

Author: Assad Ebrahim
assad.ebrahim@alum.swarthmore.edu

```
yheight 2 – DO
   xwidth 2 – DO
     diamond-avg attenuate store-to-row
   LOOP
   restore-row-minus2*
LOOP
restore-lasttwo-rows ;
```

Related Reading:
For even more hair-raising adventures in reverse engineering silicon, see;
http://adamsblog.aperturelabs.com/2013/01/fun-with-masked-roms.html

Novix NC4016 was a 16-bit Forth microprocessor from Chuck Moore's silicon startup Novix Inc. in 1985 that was enhanced by Harris Semiconductor to the radiation hardened RTX2000RH. The latter was used by NASA in various satellites and space vehicles.

Author: Assad Ebrahim
assad.ebrahim@alum.swarthmore.edu

Objective. This worksheet uses DEBUG to explore, illustrate, discover key points about Computer Architecture and Programming. This continues from Parts 1,2,3. This worksheet explores the x86 instruction set using debug.

Flags (Processor Status) Register

The flags register is a bank of 8 bits, each of which signals a particular state that the processor may be in after performing an arithmetic (inc, dec, add, sub, mul, imul, div, idiv), logic (and, or, not, xor, neg), or comparison (test, cmp) instruction. Note, the states are independent and not mutually exclusive, i.e. a single computation may set/clear multiple flags. Example: xor ax,ax will set ZR (zero) and PE (even parity).

```
-rf
OV UP EI NG NZ AC PO CY
```

Grouped logically, the flags are as below. Note that debug shows the state not whether the value is 1 or 0, to avoid error, and to emphasize that this are state not numerical variables.

```
This is an 8-bit register that holds 1-bit of data for each
of the eight "Flags" which are to be interpreted as follows:

Textbook abbrev. for Flag Name =>   of df if sf zf af pf cf
If the FLAGS were all SET (1),      -- -- -- -- -- -- -- --
they would look like this...   =>   OV DN EI NG ZR AC PE CY
If the FLAGS were all CLEARed (0),
they would look like this...   =>   NV UP DI PL NZ NA PO NC
```

Auxilliary (AF) = set to auxiliary carry (AC=1) if unsigned overflow happens on the low nibble, else no-auxilliary-carry (NA=0)

Carry (CF) = set to carry (CY=1, use stc) if unsigned overflow happens on the indicated byte or word (eg inc al, or inc ax), else no-carry (NC=0, use clc))

Overflow (OF) = set to signed overflow (OV=1), e.g. when 100+50 exceeds +128, else no-overflow (NV=0) when calculation is within signed range. What about -100-50? (underflow?)

Sign (SF) = set to negative (NG=1) if result is negative, else plus (PL=0) when computation is positive What if computation is zero – is the flag unchanged?

Parity (PF) = set to 1 (PE) if lower byte has even number of 1 bits else 0 (PO) when LSB has odd number of 1 bits

Zero (ZF) = set to zero (ZR=1) if computation results in a 0, else non-zero (NZ=0)

Direction (DF) = set to down (DN=1, use std) if processing data chain works down from high memory to low memory, else works up (UP=0, use cld) in the opposite direction; (note the convention in 8086 is to grow down in memory)

Interrupt (IF) = set to enable interrupts (EI=1, use sti) if interrupts are enabled, else disable interrupts (DI=0, use cli) when interrupts are disabled

How to toggle them in Debug? Use rf and then the flag value to be set/cleared. Separate flags with spaces to specify multiple states at the same time.

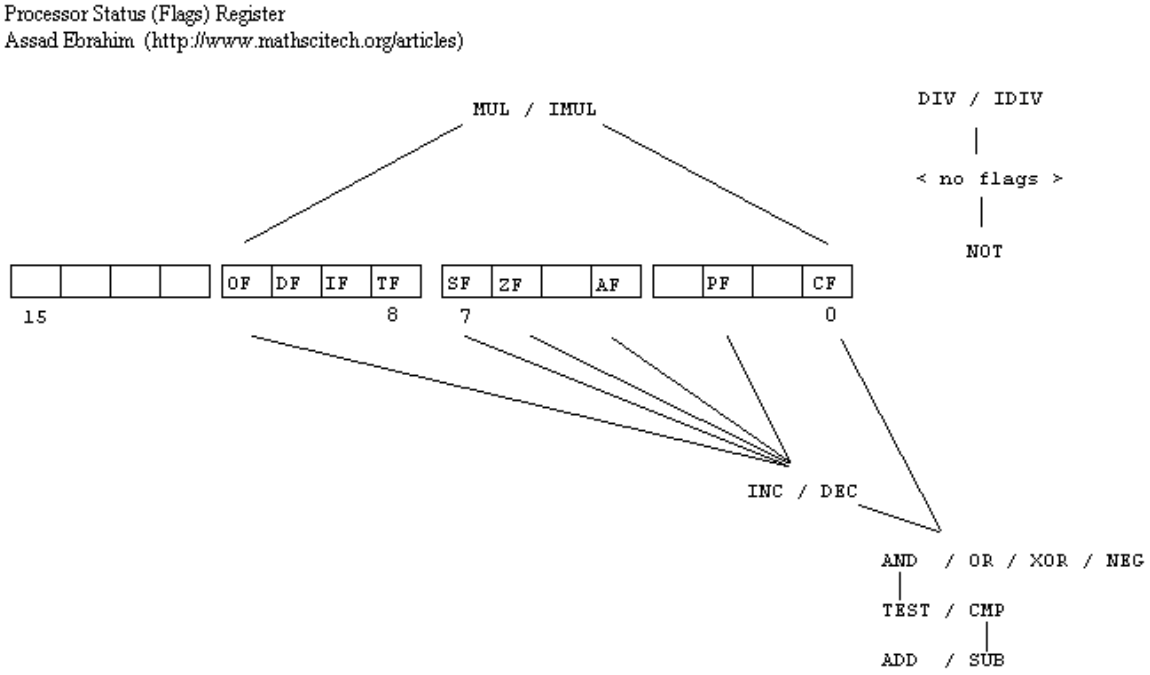
How to toggle them in code? How to view them in code? Via a bitmask? Hex to bin? **See Section X & exercises**

How to ensure the flags start in a fresh state for a piece of code before it runs (both in debug and in code!) What about for a subroutine – are the flags put on the stack and then restored?

Author: Assad Ebrahim
assad.ebrahim@alum.swarthmore.edu

```
AX=0890  BX=0000  CX=7D00  DX=0000  SP=00FD  BP=0000  SI=0000  DI=01B0
DS=073F  ES=073F  SS=073F  CS=073F  IP=0100   OV UP DI NG ZR AC PO CY
073F:0100 F3            REPZ
073F:0101 AB            STOSW
-g102

AX=0890  BX=0000  CX=7CFF  DX=0000  SP=00FD  BP=0000  SI=0000  DI=01B2
DS=073F  ES=073F  SS=073F  CS=073F  IP=0100   OV UP DI NG ZR AC PO CY
073F:0100 F3            REPZ
073F:0101 AB            STOSW
```



Reverse engineering instructions using DEBUG

Xor

Exercise: What does XOR do? Verify the truth function. What is XOR(AX,AX) = ?

```
xor ax, ax    ;     ax = 0x0000
```

Repz; stosw;

Exercise: Explain how this piece of code works and what it does? Verify it in debug.

```
Mov ax, FFFE  ;
mov di, 0900   ; offset (ES:DI)
mov cx, 006   ; counter
repz           ; repeat until zero the paired instruction (stosw) decrementing CX counter. If zero (separate from zero flag ZF) stop loop and continue rest of code; else run paired instr & iterate
stosw          ; store single word (2 bytes) from AX to memory location ES:DI; increment/decrement by 2 bytes depending as direction flag (DF) is DN/UP;
```

Author: Assad Ebrahim
assad.ebrahim@alum.swarthmore.edu

Write tester code (test harness) -

```
-a100
073F:0100 mov ax, fffe
073F:0103 mov di, 0900
073F:0106 mov cx, 0006
073F:0109 repz
073F:010A stosw
073F:010B
```

Inspect the unassembled code

```
-u100 10a
073F:0100 B8FEFF        MOV     AX,FFFE
073F:0103 BF0009        MOV     DI,0900
073F:0106 B90600        MOV     CX,0006
073F:0109 F3            REPZ
073F:010A AB            STOSW
```

Exercise why do lines 103 and 106 look backwards? (Ans. 8086 is little-endian...)

Set computer into known state:

Set IP to 0100 and registers to 0 (ax, bx, cx, dx, si, di)

Set 8 flags in a known state (nv dn ei pl nz na po nc)

```
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0100   NV DN EI PL NZ NA PO NC
073F:0100 B8FEFF        MOV     AX,FFFE
-rf
NV DN EI PL NZ NA PO NC  -nv dn ei pl nz na po nc
```

Check the code and stack memories

a) **Because we're in the tiny memory model (CS=SS=DS) we shouldn't use memory near 0x0100 (below is stack, above is code).**

```
-d cs:98 10f
073F:0090                       00 00 00 00 00 00 00 00
073F:00A0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
073F:00B0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
073F:00C0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
073F:00D0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
073F:00E0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
073F:00F0  00 00 00 FE FF 00 00 09-01 3F 07 A3 01 00 00 00
073F:0100  B8 FE FF BF 00 09 B9 06-00 F3 AB 00 00 00 00 00
```

Check the memory window expected to be used (extra segment for string processing)

```
-d es:8f0 90f
073F:08F0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
073F:0900  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
```

Trace through first 3 instructions

a) **Notice the three registers are set**

b) **Notice mov does not change flags**

c) **Notice that REPZ/STOSW are treated as a single instruction and pointed to together at IP=0109h**

```
AX=FFFE  BX=0000  CX=0010  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0900
DS=073F  ES=073F  SS=073F  CS=073F  IP=0109   NV DN EI PL NZ NA PO NC
073F:0109 F3            REPZ
073F:010A AB            STOSW
```

Now trace first time the repz/stosw and afterwards inspect the ES memory window. Notice:

a) **IP does not move**

Author: Assad Ebrahim
assad.ebrahim@alum.swarthmore.edu

b) **AX (value to copy) did not change**
c) **CX counter decremented by 1**
d) **Notice DI (offset into ES to copy to) decremented by two (to 0x08FE) since DN (down) flag is set**
e) **Notice no flags changed state, in particular ZF flag stayed NZ (nonzero)**
f) **Notice the AX data is copied into memory in little endian order (0xFFFE written FE FF in memory, writing downward in memory, according to 8086 convention)**

```
-t

AX=FFFE  BX=0000  CX=0005  DX=0000  SP=00FD  BP=0000  SI=0000  DI=08FE
DS=073F  ES=073F  SS=073F  CS=073F  IP=0109   NV DN EI PL NZ NA PO NC
073F:0109 F3            REPZ
073F:010A AB            STOSW
-
-d es:8f0 90f
073F:08F0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
073F:0900  FE FF 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
```

Now trace through 5 more times.

a) **Notice we now see the behaviour of REPZ/STOSW in practice: REPZ repeats until CX is zero. So each iteration means CPU decrements CX counter; test if zero; go past next instruction if zero and stop loop;**
b) **else run next instr & then return**

```
-t

AX=FFFE  BX=0000  CX=0004  DX=0000  SP=00FD  BP=0000  SI=0000  DI=08FC
DS=073F  ES=073F  SS=073F  CS=073F  IP=0109   NV DN EI PL NZ NA PO NC
073F:0109 F3            REPZ
073F:010A AB            STOSW
-d es:8f0 90f
073F:08F0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 FE FF   ..........
073F:0900  FE FF 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ..........
```

```
AX=FFFE  BX=0000  CX=0003  DX=0000  SP=00FD  BP=0000  SI=0000  DI=08FA
DS=073F  ES=073F  SS=073F  CS=073F  IP=0109   NV DN EI PL NZ NA PO NC
073F:0109 F3            REPZ
073F:010A AB            STOSW
-d es:8f0 90f
073F:08F0  00 00 00 00 00 00 00 00-00 00 00 00 FE FF FE FF   ..........
073F:0900  FE FF 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ..........
```

```
AX=FFFE  BX=0000  CX=0002  DX=0000  SP=00FD  BP=0000  SI=0000  DI=08F8
DS=073F  ES=073F  SS=073F  CS=073F  IP=0109   NV DN EI PL NZ NA PO NC
073F:0109 F3            REPZ
073F:010A AB            STOSW
-d es:8f0 90f
073F:08F0  00 00 00 00 00 00 00 00-00 00 FE FF FE FF FE FF   ..........
073F:0900  FE FF 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ..........
```

```
-t

AX=FFFE  BX=0000  CX=0001  DX=0000  SP=00FD  BP=0000  SI=0000  DI=08F6
DS=073F  ES=073F  SS=073F  CS=073F  IP=0109   NV DN EI PL NZ NA PO NC
073F:0109 F3            REPZ
073F:010A AB            STOSW
-d es:8f0 90f
073F:08F0  00 00 00 00 00 00 00 00-FE FF FE FF FE FF FE FF   ..........
073F:0900  FE FF 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ..........
```

Author: Assad Ebrahim
assad.ebrahim@alum.swarthmore.edu

```
-r
AX=FFFE  BX=0000  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=08F4
DS=073F  ES=073F  SS=073F  CS=073F  IP=010B   NV DN EI PL NZ NA PO NC
073F:010B 0000          ADD     [BX+SI],AL                      DS:0000=CD
-d es:8f0 90f
073F:08F0  00 00 00 00 00 00 FE FF-FE FF FE FF FE FF FE FF   ................
073F:0900  FE FF 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
```

Notice:

- cx decremented to zero but the NZ flag did not move… ???

Exercise: Reset the loop and set the direction flag to UP. See if the STOSW command now uses DI in the opposite direction.

```
-rip
IP 010B
:100
-rf
NV DN EI PL NZ NA PO NC  -UP
-r
AX=FFFE  BX=0000  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=08F4
DS=073F  ES=073F  SS=073F  CS=073F  IP=0100   NV UP EI PL NZ NA PO NC
073F:0100 B8FEFF        MOV     AX,FFFE
```

It does

```
-t

AX=FFFE  BX=0000  CX=0005  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0902
DS=073F  ES=073F  SS=073F  CS=073F  IP=0109   NV UP EI PL NZ NA PO NC
073F:0109 F3            REPZ
073F:010A AB            STOSW
```

After two:

```
AX=FFFE  BX=0000  CX=0004  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0904
DS=073F  ES=073F  SS=073F  CS=073F  IP=0109   NV UP EI PL NZ NA PO NC
073F:0109 F3            REPZ
073F:010A AB            STOSW
-d es:8f0 90f
073F:08F0  00 00 00 00 00 00 FE FF-FE FF FE FF FE FF FE FF   ..........
073F:0900  FE FF FE FF 00 00 00 00-00 00 00 00 00 00 00 00   ..........
```

After the rest:

```
AX=FFFE  BX=0000  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=090C
DS=073F  ES=073F  SS=073F  CS=073F  IP=010B   NV UP EI PL NZ NA PO NC
073F:010B 0000          ADD     [BX+SI],AL                      DS:0000=CD
-d es:8f0 90f
073F:08F0  00 00 00 00 00 00 FE FF-FE FF FE FF FE FF FE FF   ................
073F:0900  FE FF FE FF FE FF FE FF-FE FF FE FF 00 00 00 00   ................
```

Remarks: CISC approach to computing – hindrance or a help?

REPZ/STOSW illustrate the CISC (complex instruction set computer) approach: the above is a common programmer desire: filling up a block of memory with a value (usually 0x0 to clear it out). Rather than have the programmer also have to write code to do this, why not provide a hardware capability (function) to do it? At once a convenience thing and a marketing thing --- novel, unique, advanced, programmer friendly, raises a barrier against leaving the product (code incompatibility, and need greater programmer skill to leave).

Author: Assad Ebrahim
assad.ebrahim@alum.swarthmore.edu

True, on the one hand, but MISC (Chuck Moore) would say this adds additional burden. Look at the above subtleties. One has to understand now the specific implementation in order not to be caught by details such as assuming that the ZF will be set when CX reaches 0. Or making sure that the direction flag is set correctly.

The equivalent loop in assembly code would be more transparent.

Exercise: Timing
Is the repz/stosw instruction faster than the equivalent assembly code?
Write the equivalent assembly code.

Setting flags in code

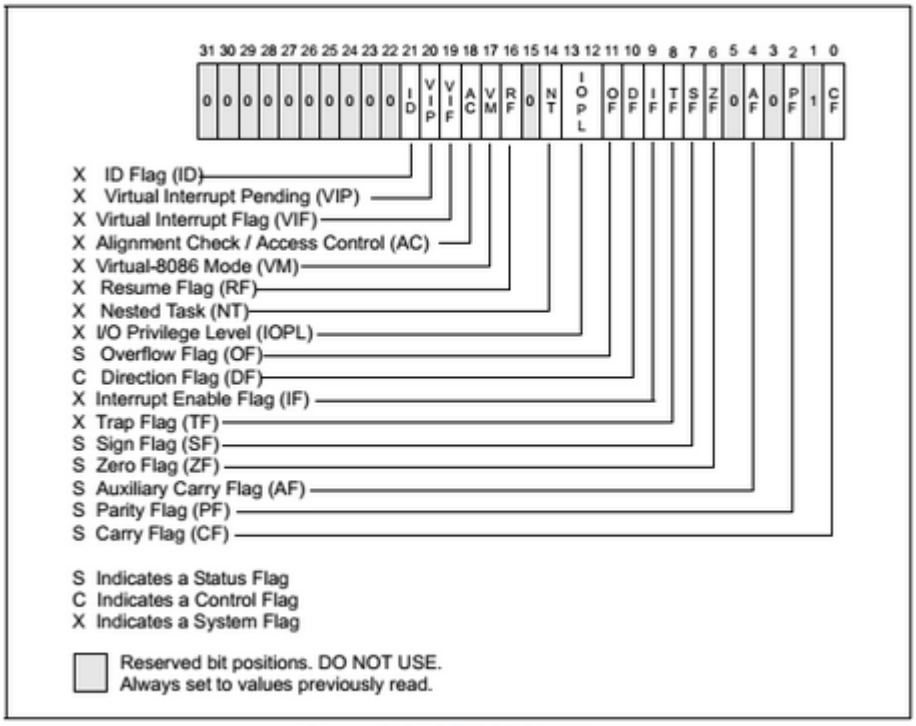


Figure 3-8. EFLAGS Register

https://stackoverflow.com/a/39816897/181638

Note that only direction flag (DF) is considered a control flag, and so has specific instructions to set/clear it (std and cld). The rest are status flags. But even so there are ways to reset the flags using lahf (load flags into ah) and sahf (store flags from ah). If one wants to work with the stack, pushfd (push flags data onto stack) and popfd (pop flags data from stack).

Other flag manipulating commands:
clc (clear carry), stc (set carry), cmc (?), cmovcc, adc, setcc
LAHF

Interrupt Enable (IF) can be manipulated with cli and sti.

Author: Assad Ebrahim
assad.ebrahim@alum.swarthmore.edu

Objective. This worksheet uses DEBUG to explore, illustrate, discover key points about Computer Architecture and Programming. This continues from Parts 1,2,3. This worksheet explores the x86 instruction set using debug.

Numbers

Binary
0001 0x1
0010
0011
…
1000 0x8
1001
…
1111 0xF

Binary coded decimal
Using binary to encode a decimal digit. This is wasteful as codes 10-15 (6 codes out of 16, or 38%) are not used.
It is one way to work with decimal numbers on a 4-bit machine (early 4004 and 4040 were this way). Early calculators were this way.
[1] Busicom 141-PF and Intel 4004
http://www.vintagecalculators.com/html/busicom_141-pf_and_intel_4004.html

Exercise: write these early Intel chips in binary coded decimal: 4004, 4040, 8008, 8080, 8086, 8088
0100 0000 0000 0100
0100 0000 0100 0000
1000 0000 0000 1000
1000 0000 1000 0000
1000 0000 1000 0110
1000 0000 1000 1000

Hex is binary compressed and concatenated

8086 is little endian

Size matters (nybbles, bytes, words, double-words, and quad-words)
Context is byte (8 bits) or word (16 bits).

Sign matters (unsigned and signed arithmetic)
Unsigned (positive only) number: 0xFF (255=2^8) or 0xFFFF (65,536=2^16)
Signed (positive or negative) number: -1 = 0xFF. Highest bit (msb) set means negative. So 0111 1111b is max positive number (+127). 1000 0000b is max negative number (-128), and 1111 1111b is -1, i.e. this is a wrap-around counting. 0000 0000b is 0. This scheme is called two's complement.
Exercise: How to figure out what 1011 1011b is?

Author: Assad Ebrahim
assad.ebrahim@alum.swarthmore.edu

Fixed Point

Floating Point

Comparison instructions

Comparison instructions are the heart of computing decisions.

Conditional jump instructions are a logical soup of multiple ways to say the same thing.
JMP – standard goto, spaghetti code if used badly, Wirth and others on structured programming (subroutines, functions, etc.)

Numeric Comparison

	Direct		Negated	
	Unsigned	Signed	Unsigned	Signed
A>B	JA	JG	JNBE	JNLE
A<B	JB	JL	JNAE	JNGE
A>=B	JAE	JGE	JNB	JNL
A<=B	JBE	JLE	JNA	JNG
A=B	JE		JNE	

JG A,B – Jump if A > B (signed), i.e. if ZF=0 and SF=0
JA A,B – Jump if A > B (unsigned), i.e. if ZF=0 and CF=0
JC – Jump if Carry (8 or 16-bit) is 1

Status Control

Direct	Negated
JZ	JNZ
JC	JNC
JO	JNO
JP	JNP
JS	JNS
JPE	
JPO	

Register Control

JCXZ

Author: Assad Ebrahim
assad.ebrahim@alum.swarthmore.edu

Objective. This worksheet uses DEBUG to explore, illustrate, discover key points about Computer Architecture and Programming. This continues from Parts 1-4. This worksheet explores the x86 instruction set connecting computer to input/output ports.

Input/Output Ports

It is not possible to do anything without on a computer without interfacing with the outside world. Every computer requires input, performs some processing, and then must take some action (produce some output), whether this is displaying something on a screen, or some other more targeted action.

8086 and similar computers had several peripheral chips that supported them. Later, single chip computers incorporated all elements onto a single die.

In addition, separate peripheral devices can be connected to computers (keyboard, mouse, joystick, data acquisition equipment, sensors, monitor, hard disk, etc.) Each of these typically has a device driver that allows accessing and working with the device.

Excellent:
https://mysite.du.edu/~etuttle/electron/elect51.htm

http://physweb.bgu.ac.il/COURSES/SignalNoise/interrupts.pdf

https://www.electronicshub.org/types-of-computer-ports/

Graphics mode: http://www.delorie.com/djgpp/doc/ug/graphics/vga.html

Historical cards: CGA, EGA

VGA mode 0x13 (320x200, 256 colors, VGA compatible graphics card)

DOS text mode (normal) 0x03

To draw something on the screen, you draw to the graphics card's memory, i.e. memory mapped address:

VGA memory is located at physical address 0xA0000 through to 0xAFFFF (64k bytes, or 320x200 bytes)
In 8086 segmented memory model this would be A000:0000
A pixel is displayed by giving x,y coordinates (0-319, 0-199) and a color (0-255)
The coordinates are mapped onto memory as follows:
0xA0000 + x + 320*y
Exercise: explain why this formula makes sense.

Two ways to create graphics: 1) poke them directly into the memory card memory, pixel at a time; 2) create a memory buffer and copy (bitblt) it to the memory card.
This takes more memory, as one replicates a 320x200 byte framebuffer in memory (64k bytes).
Exercise: Put output onto the memory card directly, and with a memory buffer copy.

What colors can you choose? Colors are controlled by a hardware component called the palette, which is a table listing the actual color values for each of the 256 color values that you can display. When you first select a video mode the first 16 entries in the palette (colors zero to fifteen) will be set to the standard DOS text mode colors (black, blue, green, cyan, red, magenta, brown, light grey, dark grey, pale blue, pale green, pale cyan, pale red, pale magenta, yellow, and white), but the other 240 colors may be set to different values depending on the machine. In

Author: Assad Ebrahim
assad.ebrahim@alum.swarthmore.edu

order to use anything more than those 16 default colors, you must set the palette to some new values of your own, which is done by writing a palette index to hardware port 0x3C8 followed by three color values to port 0x3C9, eg:

Port 0x3DA bit 3 holds the retrace period of the display function.

Other graphics resources:
x2ftp - ftp://x2ftp.oulu.fi/pub/msdos/programming/

The most comprehensive collection of DOS graphics coding material on the net.

PCGPE - ftp://x2ftp.oulu.fi/pub/msdos/programming/gpe/

A good clear introduction to many VGA and general graphics coding techniques, although all the example programs are in Pascal.

Abrash in DDJ - ftp://x2ftp.oulu.fi/pub/msdos/programming/docs/graphpro.lzh

The "legendary" articles written by Michael Abrash and published in DDJ between 91/93, covering mode-X, polygon rasterisation, and many other fascinating topics.

Author: Assad Ebrahim
assad.ebrahim@alum.swarthmore.edu

Objective. This worksheet uses DEBUG to explore, illustrate, discover key points about Computer Architecture and Programming. This continues from Parts 1-4. This worksheet explores the x86 instruction set connecting computer to input/output ports.

Pointers & Memory

Thus far, we've been able to store and manipulate data in registers (mov) and store and fetch data from the stack (push, pop). Many computing operations will require more memory than the few hundred bytes available this way.

Memory is a contiguous row of bytes, sort of like a row of lockers. Each byte has an address. You can store or fetch from the address.

On an 8086, there are 20 bits worth of addressable memory, or 2^20 = 1,048,576 bytes, or 1MB. But the word length, and therefore the address length, is 16 bits, able to address 2^16 = 65.536 bytes, or 64KB at a time. Hence, 8086 uses a segmented memory model. Memory can be partitioned into 2^4 = 16 discontinuous segments, each of 64KB. What are these segments?
In hex:
0:0000, 1:0000, 2:0000, …, E:0000, F:0000
But in practice the segments have been done so that there are overlaps, i.e. the 64KB window can be positioned to start anywhere (within 10h bytes).
So we have:
0000:0000, where 0xxx:xxx0 is the full memory, and the middle bits overlap (represent the same part of the absolute address).

Tiny memory model: everything runs within a single segment (DS=ES=SS=CS)

```
AX=FFFE  BX=0000  CX=00FF  DX=0000  SP=00FD  BP=0000  SI=0000  DI=4100
DS=073F  ES=073F  SS=073F  CS=073F  IP=0100   NV UP EI NG NZ AC PE NC
```

The Stack runs from 0x0 to 0xFF. Stack pointer (SP) here is at 0xFD, and the stack will grow downwards in memory (toward smaller values).
Code runs from 0x100 onward – however much is needed. IP points to 0x100 at the start. Let's reserve 4000h bytes (or 3 x 4096d, or 12KB) of memory for code (Moore's 1000 instructions, each (simple) instruction taking 3 bytes of memory, one byte for the instruction, and two bytes for the parameter word).
So we can start our offset into memory from DI=4100h.
Let's store 0xFFFE into DS:DI, i.e. 073F:4100.

Author: Assad Ebrahim
assad.ebrahim@alum.swarthmore.edu

```
-a100
073F:0100 mov DS:4100, ax
               ^ Error
073F:0100 mov 4100, ax
                 ^ Error
073F:0100 mov [4100],ax
073F:0103 mov [DS:4100]
              ^ Error
073F:0103 mov [DS+4100],ax
              ^ Error
073F:0103 mov [073F:4100],ax
                  ^ Error
073F:0103 mov [073F+4100],ax
073F:0106
-u100 106
073F:0100 A30041        MOV     [4100],AX
073F:0103 A33F48        MOV     [483F],AX
```

```
AX=FFFE  BX=0000  CX=00FF  DX=0000  SP=00FD  BP=0000  SI=0000  DI=4100
DS=073F  ES=073F  SS=073F  CS=073F  IP=0100   NV UP EI NG NZ AC PE NC
073F:0100 A30041        MOV     [4100],AX                       DS:4100=0000
-t
```

```
-d40F0 410F
073F:40F0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
073F:4100  FE FF 00 00 00 00 00 00-00 00 00 00 00 00 00 00
```

```
AX=FFFE  BX=0000  CX=00FF  DX=0000  SP=00FD  BP=0000  SI=0000  DI=4100
DS=073F  ES=073F  SS=073F  CS=073F  IP=0103   NV UP EI NG NZ AC PE NC
073F:0103 A33F48        MOV     [483F],AX                       DS:483F=0000
-t
```

```
-d4830 484F
073F:4830  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 FE
073F:4840  FF 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
```

Notice:

1) both accepted instructions worked according to the current (DS) segment by default.
2) [ ] specifies a number as an address (AX is a named address, [4100] is a numbered address, just like houses in the UK can have named or numbered addresses).
3) OK to do arithmetic (+ only) within [ ], but it treats the arithmetic linearly, not as segments.
4) Segment notation : is not recognized, and * is not allowed within [ ]

Let's test #1. If we switch DS and re-run the program.

```
-rds
DS 073F
:083F
```

```
-rip
IP 0106
:100
-r
AX=FFFE  BX=0000  CX=00FF  DX=0000  SP=00FD  BP=0000  SI=0000  DI=4100
DS=083F  ES=073F  SS=073F  CS=073F  IP=0100   NV UP EI NG NZ AC PE NC
073F:0100 A30041        MOV     [4100],AX                       DS:4100=0000
-t
```

Author: Assad Ebrahim
assad.ebrahim@alum.swarthmore.edu

```
-d40F0 410F
083F:40F0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
083F:4100  FE FF 00 00 00 00 00 00-00 00 00 00 00 00 00 00
```

This confirms that by default DEBUG references short memory based on the DS (data segment) pointer.

Notice at the far right, DEBUG shows the full memory address for the short memory move, i.e. DS:483F

```
AX=FFFE  BX=0000  CX=00FF  DX=0000  SP=00FD  BP=0000  SI=0000  DI=4100
DS=083F  ES=073F  SS=073F  CS=073F  IP=0103   NV UP EI NG NZ AC PE NC
073F:0103 A33F48        MOV     [483F],AX                       DS:483F=0000
```

Can we move from all registers to memory? All except IP.

```
-a100
073F:0100 mov [4100], AX
073F:0103 mov [4200], BX
073F:0107 mov [4300], CX
073F:010B mov [4400], DX
073F:010F
-a10F
073F:010F mov [4500], SP
073F:0113 mov [4600], BP
073F:0117 mov [4700], SI
073F:011B mov [4800], DI
073F:011F mov [4900], DS
073F:0123 mov [4A00], ES
073F:0127 mov [4B00], SS
073F:012B mov [4C00], CS
073F:012F mov [4D00], IP
                      ^ Error
```

```
-u100 12F
073F:0100 A30041        MOV     [4100],AX
073F:0103 891E0042      MOV     [4200],BX
073F:0107 890E0043      MOV     [4300],CX
073F:010B 89160044      MOV     [4400],DX
073F:010F 89260045      MOV     [4500],SP
073F:0113 892E0046      MOV     [4600],BP
073F:0117 89360047      MOV     [4700],SI
073F:011B 893E0048      MOV     [4800],DI
073F:011F 8C1E0049      MOV     [4900],DS
073F:0123 8C06004A      MOV     [4A00],ES
073F:0127 8C16004B      MOV     [4B00],SS
073F:012B 8C0E004C      MOV     [4C00],CS
```

What do you observe? The 8086 is tuned for moving from AX to memory – this only takes a 1 byte instruction. All the others are 2 byte instructions (e.g. 891E --- or 0x1E89 – to move from BX to memory, MOV [x], BX).

How do we specify in code a different segment, say the ES segment?
Use different instructions that know to be based on the ES segment and offsets into it.

How do we set the CS, ES, and DS segments in code?
Code segment cannot be set in code --- that would be a huge security hole as CS could be set to execute to places in memory.
DS and ES can be set through the stack (pop).

Can we move a byte register to memory?

Can we move a literal word to memory? A literal byte?

Author: Assad Ebrahim
assad.ebrahim@alum.swarthmore.edu

When moving registers, the size of the operand (parameter) has been unambiguous – 2 bytes if AX, 1 byte if AL or AH for example. When moving literals to memory we'll need some way to specify this.
The processor needs to know how many bytes to read off the code stream.
FFFE is less ambiguous (if we assume the maximum size of a parameter is 2 bytes). FF is ambiguous: it could be 0x00FF or 0xFF.

Moving data between segments.
MOVSB
MOVSW
LODSB
LODSW
LEA
LES
LDS
STOSB
STOSW
XCHG
XLATB

Segment	Offset Registers	Function
CS	IP	Address of the next instruction
DS	BX, DI, SI	Address of data
SS	SP, BP	Address in the stack
ES	BX, DI, SI	Address of destination data (for string operations)

Author: Assad Ebrahim
assad.ebrahim@alum.swarthmore.edu

Objective. This worksheet uses DEBUG to explore, illustrate, discover key points about Computer Architecture and Programming. This continues from Parts 1-4. This worksheet reverse engineers machine language for the x86 using debug, and comments on CISC vs. MISC.

References:
[1] Intel x86 Assembler Instruction Set Opcode Table http://sparksandflames.com/files/x86InstructionChart.html

Summary
Using debug to systematically explore the machine code of x86 chip shows the true complexity behind a CISC chip like the x86.
There are 0xBF varieties of add <mem>, <8-bit reg>, and 100h-0xc0 varieties of add <reg>, <reg> (8-bits).
There are an equal number of varieties of add <mem>, <16-bit reg>.
And then an equal number of add <8-bit reg>, <mem>, and then add <16-bit reg>, <mem>.
Then OR, then sub.
The number of individual instructions is high, and the number of coding varieties is a combinatorial explosion.
Imagine the number of gates in the chip required to implement such a design.

This leads to an over-riding desire – surely it should not be this hard to program a computer? Surely the design of the computer itself should not be that complex?

And you reach the line of thinking of Chuck Moore, the inventory of Forth, and thereafter of simple hardware.

So one should study x86 to get a feel for what CISC looks like, learn it well enough to do something useful in it, and then study Forth.

And then the Forth chips … in history, in reality, in FPGA form. See Appendix 1.

Redundancy: 31C0 and 33C0 both map to XOR ax,ax
Does it matter? Not really. Both get the job done. An assembler will probably pick a standard form (perhaps lower numbers). The mapping table may be larger but it is symmetric which is a benefit.

```
-a100
073F:0100 xor ax,ax
073F:0102
-e102
073F:0102  00.33   00.C0

-u100 102
073F:0100 31C0          XOR     AX,AX
073F:0102 33C0          XOR     AX,AX
```

A larger response to a similar observation: http://www.mlsite.net/blog/?p=76
Hello MXH. Just a note on your "Redundant Opcodes" article. Some of the redundancies you list are desirable so as not to break the symmetry of the operations table. E.g. 31C0 and 33C0 both give xor ax,ax, but are both required in order that their respective tables are complete 31 is xor *,reg16 while 33 is xor reg16,* where the * denotes iteration.

Author: Assad Ebrahim
assad.ebrahim@alum.swarthmore.edu

Another example is based on size. AX and AL are optimized for arithmetic operations. E.g. 04HH is add al, imm-byteHH. The redundant operation you list 80COHH is again needed to complete the symmetry of the 80 op-code table, and is 1 byte longer.

A third example is a different kind of symmetry where the redundancy arises from a processing distinction. Example opcode 82 has the same output as opcode 80 but is handled differently by the processor which sign-extends the immediate byte to word for 0x82 and does not sign extend for 0x80. This is an immaterial distinction for byte literals but a material one for word literals (81 and 83) so is retained for symmetry.

The key question, which is not addressed in your article, is whether there is a downside of these redundant opcodes? In terms of silicon or micro-code to decode the instructions, I wouldn't expect any, but someone more knowledgeable would need to confirm.

The only downside I can see is the philosophical one which you express in your article. But this is in contradiction with established practice in mathematics as well, where, for example, in Pascal's triangle, you will find that C(N,1) = C(N,N-1), and many other identities where one side is “redundant”. The point in mathematics is that one values the symmetry because it allows automation without having to have all sorts of specific logic tests because the redundancy has been manually pruned out. Setting aside the opcodes which lead to smaller code (these should not be classed as redundant), I expect the same argument will hold for the symmetry preserving ones.

Hope the perspective helps.

There are tricks for more compressed code.

To zero out the ax register you can use: xor ax,ax which is 2 bytes (31C0) or mov ax,0000 which is 3 bytes (B80000) because the 16-bit immediate value needs to be specified. (Exercise: Prove that the xor instruction does this.)

```
-r
AX=FFFF  BX=0000  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0100   NV UP EI PL NZ NA PO NC
073F:0100 31C0          XOR     AX,AX
-t

AX=0000  BX=0000  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0102   NV UP EI PL ZR NA PE NC
073F:0102 B80000        MOV     AX,0000
```

(setup: rax FFFF)

Many variations: there are 2050 variations of ADD instruction on the x86.

The first 1024 variations provide mem/reg and reg/reg capability. The second 1024 variations provide mem/imm and reg/imm capability for 8-bit and 16-bit immediates, both signed and unsigned. The last 2 provide optimized (short code) versions of reg/imm for use with the accumulator register AX or AL. All other registers will use the longer forms (longer by 1 byte).

```
073F:0100 8000FF        ADD     BYTE PTR [BX+SI],FF
073F:0103 8100FEFF      ADD     WORD PTR [BX+SI],FFFE
073F:0107 8200FE        ADD     BYTE PTR [BX+SI],FE
073F:010A 8300FE        ADD     WORD PTR [BX+SI],-02
```

(Note: opcode 82 has the same output as opcode 80 but is handled differently by the processor which sign-extends the immediate byte to word for 0x82 and does not sign extend for 0x80. This is an immaterial distinction for byte literals but a material one for word literals (81 and 83).)

Author: Assad Ebrahim
assad.ebrahim@alum.swarthmore.edu

https://github.com/aquynh/capstone/issues/238

```
073F:0100 80C001        ADD     AL,01
073F:0103 81C00100      ADD     AX,0001
073F:0107 82C001        ADD     AL,01
073F:010A 83C001        ADD     AX,+01
```

This is huge amount of complexity.
Why would this have been a design decision? Remember that Stephen Morse, the designer of the Intel 8086 was a software engineer and was looking to make things efficient for software developers.
The answer is that Intel was looking to provide an assembly language interface that would, in some sense, abstract away the machine. A programmer could perform any function in (almost) any possible mode with (almost) any possible registers, memory, and immediate values as targets.
reg/lit
reg/mem
reg/reg
mem/lit
mem/reg
even mem/mem is possible using string operations
A key reason is space – by having a single machine instruction (of 2-4 bytes) complete a single abstract operation, the end user program was not only easier to write, but also substantially smaller in machine code. This was important given the 64K memory limit on 16-bit computers.
Compiled code would be similarly smaller and compressed.
An additional reason is the simplicity for the programmer and/or compiler writer as abstract 3G languages could be mapped much more easily one-to-one with machine instructions, simplifying the adoption of a processor by making it easier to create the suite of development tools it would require.

Exercise: how to implement such a situation?
Brute force would be if all of these are implemented in silicon, i.e. using gates.
CISCs break complex instructions into micro instructions (microcode)
Introducing microcode as a way to save silicon.
Introducing Forth as a simple microcode.
References:
The bible are Agner Fog optimization manuals [1] which contain quite a detailed description of the microarchitecture of intel and AMD CPUs from the pentium era till today. They are based on the extensive reverse engineering done by the author.
David Kanter microarchitecture articles at RWT [2] are also quite good.
Intel manuals are quite detailed as well.
[1] http://www.agner.org/optimize/
[2] http://www.realworldtech.com/cpu/

You can see btw how much time/effort goes into understanding a particular chip.

Exercise: using debug to read machine code:
B0 = mov al, ? = mov bl, ? = mov cl, ? = mov dl

Author: Assad Ebrahim
assad.ebrahim@alum.swarthmore.edu

? = mov ah, ? = mov bh, ? = mov ch, ? = mov dh
? = mov ax, ? = mov bx, B9 = mov cx, BA = mov dx
BF = mov di, ? = mov si,
? = push ax, ? = push bx, ? = push cx, ? = push dx
? = pop ax, ? = pop bx, ? = pop cx, ? = pop dx

Not allowed
? = mov ds, ? = mov ss, ? = mov es
Given one cannot use mov to set DS, SS, ES registers, how to change these in code?
Use pop instructions and the stack.

Note that the machine language instructions 0x26 ES: and 0x2E CS: are segment override prefixes… examples of usage

What are segment override prefixes, and how are they used?
Which segment register that is used in the address calculation depends on the register that is used for `baseReg`. The DS register is assumed for the segment unless `baseReg` is the register BP, in which case SS is assumed. However, any segment register can be explicitly specified using what is called a *segment override* prefix (discussed below). Also, some special instructions may assume other segment registers
https://ece425web.groups.et.byu.net/stable/labs/8086Assembly.html

A segment override prefix allows any segment register (DS, ES, SS, or CS) to be used as the segment when evaluating addresses in an instruction. An override is made by adding the segment register plus a colon to the beginning of the memory reference of the instruction as in the following examples:

```
mov     ax, [es:60126]          ; Use es as the segment
mov     ax, [cs:bx]             ; Use cs as the segment
mov     ax, [ss:bp+si+3]        ; Use ss as the segment
```

EE = out dx, al (value in AL, port number in DX if over 0xff)

```
-a100
073F:0100 mov al,1
073F:0102 mov bl,2
073F:0104 mov cl,3
073F:0106 mov dl,4
073F:0108 mov ah,5
073F:010A mov bh,6
073F:010C mov ch,7
073F:010E mov dh,8
073F:0110 mov ax,9
073F:0113 mov bx,A
073F:0116 mov cx,B
073F:0119 mov dx,C
073F:011C mov di,DD
073F:011F mov si,EE
```

Author: Assad Ebrahim
assad.ebrahim@alum.swarthmore.edu

```
073F:0100 B001          MOV     AL,01
073F:0102 B302          MOV     BL,02
073F:0104 B103          MOV     CL,03
073F:0106 B204          MOV     DL,04
073F:0108 B405          MOV     AH,05
073F:010A B706          MOV     BH,06
073F:010C B507          MOV     CH,07
073F:010E B608          MOV     DH,08
073F:0110 B80900        MOV     AX,0009
073F:0113 BB0A00        MOV     BX,000A
073F:0116 B90B00        MOV     CX,000B
073F:0119 BA0C00        MOV     DX,000C
073F:011C BFDD00        MOV     DI,00DD
073F:011F BEEE00        MOV     SI,00EE
```

Exercise: What instructions are BC and BD?

```
-e12c
073F:012C  00.BC   00.FF   00.BD   00.10

-u12c
073F:012C BCFFBD        MOV     SP,BDFF
```

```
-e12f
073F:012F  BD.BD
073F:0130  10.10   00.11

-u12f
073F:012F BD1011        MOV     BP,1110
```

Exercise: what are the push machine code instructions?

```
073F:0122 push ax
073F:0123 push bx
073F:0124 push cx
073F:0125 push dx
073F:0126 pop ax
073F:0127 pop bx
073F:0128 pop cx
073F:0129 pop dx
```

```
073F:0122 50            PUSH    AX
073F:0123 53            PUSH    BX
073F:0124 51            PUSH    CX
073F:0125 52            PUSH    DX
073F:0126 58            POP     AX
073F:0127 5B            POP     BX
073F:0128 59            POP     CX
073F:0129 5A            POP     DX
```

Exercise: What instructions are 54,55,56,57; 5C,5D,5E,5F

Exercise: What machine instructions are pushf, popf?

```
073F:012A pushf
073F:012B popf
073F:012A 9C            PUSHF
073F:012B 9D            POPF
```

Appendix 1 – Reverse Engineering x86 machine language

Author: Assad Ebrahim
assad.ebrahim@alum.swarthmore.edu

Note:
e command starts by default in the DS data segment.
a and u commands starts in the CS code segment.

Exercise: Reverse Engineer Machine Language x86 using the e command.

```
-                    e100
073F:0100  00.00   00.04   00.00   01.05   00.00   02.06   00.00   03.07
073F:0108  08.00   09.08   0A.00   0B.09   0C.00   0D.0A   0E.00   0F.0B
073F:0110  26.00   00.0C   45.00   89.0D   2E.     00.00   46.0F

-u100
073F:0100 0004          ADD     [SI],AL
073F:0102 0005          ADD     [DI],AL
073F:0104 00060007      ADD     [0700],AL
073F:0108 0008          ADD     [BX+SI],CL
073F:010A 0009          ADD     [BX+DI],CL
073F:010C 000A          ADD     [BP+SI],CL
073F:010E 000B          ADD     [BP+DI],CL
073F:0110 000C          ADD     [SI],CL
073F:0112 000D          ADD     [DI],CL
```

Observe the complexity in the CISC x86 processor! Every possible combination of registers and memory access types for each of the many different basic instructions (add, sub, etc.)
File: reveng.com has 0040ff to 0200, i.e. three instruction families: 00 add, 01 add, 02 add

ADD (a lot of classes) 1026 variants, 2 basics (0x04, 0x05) and 1024 flavors of advanced…
OR (a lot of classes) same with OR.
Note --- the exploration of the first 0x00FF instructions (0x100, 256d instructions) is by hand in debug. Once the pattern of the next sets are verified in the first few instructions in debug, one uses a hex editor to hand edit the 01 instructions (next 256), and then to copy paste the the 0x01 instructions and search-replace the 01 first with 02 (watch out for 0x0101 which should be 0x0201), and then repeat for 03 instructions.
Then do the same for the OR instructions (08, 09, 0A, 0B) – another 1024 instructions.

This gets us to 2048+2+2+2+1 = 2055 instructions up to 0x0E.
The file to illustrate these instructions is runs to 085d (75d bytes of memory since we start at 0x100h, which is 1885d instructions).
(It is useful to have a hex calculator)

See Toolbox for:
HexEditor (HxD)
Hex Calculator (Lite)

Why is the sequence the way it is? Bit patterns – 3 bits to encode 8 selections.
https://www-user.tu-chemnitz.de/~heha/viewchm.php/hs/x86.chm/x64.htm

Author: Assad Ebrahim
assad.ebrahim@alum.swarthmore.edu

```
-u100
073F:0100 0040FF        ADD     [BX+SI-01],AL
073F:0103 0041FF        ADD     [BX+DI-01],AL
073F:0106 0048FF        ADD     [BX+SI-01],CL
073F:0109 0049FF        ADD     [BX+DI-01],CL
073F:010C 0050FF        ADD     [BX+SI-01],DL
073F:010F 0051FF        ADD     [BX+DI-01],DL
073F:0112 0058FF        ADD     [BX+SI-01],BL
073F:0115 0059FF        ADD     [BX+DI-01],BL
073F:0118 0060FF        ADD     [BX+SI-01],AH
073F:011B 0061FF        ADD     [BX+DI-01],AH
073F:011E 0068FF        ADD     [BX+SI-01],CH
-u
073F:0121 0069FF        ADD     [BX+DI-01],CH
073F:0124 0070FF        ADD     [BX+SI-01],DH
073F:0127 0078FF        ADD     [BX+SI-01],BH
073F:012A 0080FF00      ADD     [BX+SI+00FF],AL
073F:012E 0088FFFF      ADD     [BX+SI+FFFF],CL
073F:0132 0090FFFF      ADD     [BX+SI+FFFF],DL
073F:0136 0098FFFF      ADD     [BX+SI+FFFF],BL
073F:013A 00A0FFFF      ADD     [BX+SI+FFFF],AH
073F:013E 00A8FFFF      ADD     [BX+SI+FFFF],CH
073F:0142 00A8FFFF      ADD     [BX+SI+FFFF],CH
073F:0146 00B0FFFF      ADD     [BX+SI+FFFF],DH
073F:014A 00B8FFFF      ADD     [BX+SI+FFFF],BH
073F:014E 00C0          ADD     AL,AL
073F:0150 00C1          ADD     CL,AL
073F:0152 00C2          ADD     DL,AL
073F:0154 00C3          ADD     BL,AL
073F:0156 00C4          ADD     AH,AL
073F:0158 00C5          ADD     CH,AL
073F:015A 00C6          ADD     DH,AL
073F:015C 00C7          ADD     BH,AL
073F:015E 00C8          ADD     AL,CL
073F:0160 00C9          ADD     CL,CL
073F:0162 00CA          ADD     DL,CL
073F:0164 00CB          ADD     BL,CL
073F:0166 00CC          ADD     AH,CL
073F:0168 00CD          ADD     CH,CL
073F:016A 00CE          ADD     DH,CL
073F:016C 00CF          ADD     BH,CL
073F:016E 00D0          ADD     AL,DL
073F:0170 00D1          ADD     CL,DL
073F:0172 00D8          ADD     AL,BL
073F:0174 00D9          ADD     CL,BL
073F:0176 00E0          ADD     AL,AH
073F:0178 00EF          ADD     BH,CH
073F:017A 00F0          ADD     AL,DH
073F:017C 00F8          ADD     AL,BH
073F:017E 00FF          ADD     BH,BH
```

Author: Assad Ebrahim
assad.ebrahim@alum.swarthmore.edu

Direction Flag

CLD (0xFC) clears direction flag (df) which is direction UP.

STD (0xFD) sets direction flag (df) which is direction DOWN.

```
-a100
073F:0100 cld
073F:0101 std
073F:0102
-u100 102
073F:0100 FC            CLD
073F:0101 FD            STD
073F:0102 FEE9          JMP     FAR CL
-                       r
AX=0000  BX=0000  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0100   NV UP EI PL NZ NA PO NC
073F:0100 FC            CLD
-t

AX=0000  BX=0000  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0101   NV UP EI PL NZ NA PO NC
073F:0101 FD            STD
-t

AX=0000  BX=0000  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0102   NV DN EI PL NZ NA PO NC
```

Conditional Jumps

```
-u100 120
073F:0100 70FF          JO      0101
073F:0102 71FF          JNO     0103
073F:0104 72FF          JB      0105
073F:0106 73FF          JNB     0107
073F:0108 74FF          JZ      0109
073F:010A 75FF          JNZ     010B
073F:010C 76FF          JBE     010D
073F:010E 77FF          JA      010F
073F:0110 78FF          JS      0111
073F:0112 79FF          JNS     0113
073F:0114 7AFF          JPE     0115
073F:0116 7BFF          JPO     0117
073F:0118 7CFF          JL      0119
073F:011A 7DFF          JGE     011B
073F:011C 7EFF          JLE     011D
073F:011E 7FFF          JG      011F
```

Notice that 0xFF (-1 signed) specifes a 1 byte move forward, so the formula is IP = IP* + REL, where IP* is where IP would have pointed after the jump is processed.

So at address 0x100: JO FF (0x70ff) means IP*=102, REL=-1, so IP=101.

Whereas at 0x100: JO 01 (0x7001) means IP*=102, REL=+1, so IP=103.

Author: Assad Ebrahim
assad.ebrahim@alum.swarthmore.edu

```
-u100 120
073F:0100 7001          JO      0103
073F:0102 7101          JNO     0105
073F:0104 7201          JB      0107
073F:0106 7301          JNB     0109
073F:0108 7401          JZ      010B
073F:010A 7501          JNZ     010D
073F:010C 7601          JBE     010F
073F:010E 7701          JA      0111
073F:0110 7801          JS      0113
073F:0112 7901          JNS     0115
073F:0114 7A01          JPE     0117
073F:0116 7B01          JPO     0119
073F:0118 7C01          JL      011B
073F:011A 7D01          JGE     011D
073F:011C 7E01          JLE     011F
073F:011E 7F01          JG      0121
```

Optimizations

Mov ax, 0000 3 bytes vs. Xor ax,ax 2 bytes

Out port

Can only use AX as the target

Out dx,ax not out dx,bx etc.

Undocumented Instruction

What does C1 do?

Let's check:

```
075A:014D C1            DB      C1
075A:014E EB02          JMP     0152
075A:0150 FF00          INC     WORD PTR [BX+SI]
075A:0152 00FF          ADD     BH,BH
```

Tracing the C1 instruction consumed C1EB02 (3 bytes) and advanced IP to FF00

```
-r
AX=0098  BX=0032  CX=3E7E  DX=0000  SP=FFFE  BP=0000  SI=01B2  DI=7D02
DS=075A  ES=A000  SS=075A  CS=075A  IP=014D   NV UP EI NG NZ NA PO CY
075A:014D C1            DB      C1
-t

AX=0098  BX=000C  CX=3E7E  DX=0000  SP=FFFE  BP=0000  SI=01B2  DI=7D02
DS=075A  ES=A000  SS=075A  CS=075A  IP=0150   NV UP EI PL NZ AC PE CY
075A:0150 FF00          INC     WORD PTR [BX+SI]                    DS:01BE=0000
-
```

Setting all registers to 0.

Author: Assad Ebrahim
assad.ebrahim@alum.swarthmore.edu

```
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=075A  CS=075A  IP=014D   NV UP EI PL ZR AC PE NC
075A:014D C1            DB      C1
-u14d 157
075A:014D C1            DB      C1
075A:014E EB02          JMP     0152
075A:0150 C1            DB      C1
075A:0151 EB02          JMP     0155
075A:0153 C1            DB      C1
075A:0154 EB02          JMP     0158
075A:0156 C1            DB      C1
075A:0157 EB02          JMP     015B
```

Tracing through 4 successive commands – no change.

Setting AX=10h – no change

Setting BX=20h --- change! (same whether AX=10h or 0h, hypoth indep of AX)

Param: 0x02eb

BX: 20h -> 08h -> 02h -> 00

```
AX=0000  BX=0020  CX=0000  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=075A  CS=075A  IP=014D   NV UP EI PL ZR AC PE CY
075A:014D C1            DB      C1
-t

AX=0000  BX=0008  CX=0000  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=075A  CS=075A  IP=0150   NV UP EI PL NZ AC PO NC
075A:0150 C1            DB      C1
-t

AX=0000  BX=0002  CX=0000  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=075A  CS=075A  IP=0153   NV UP EI PL NZ AC PO NC
075A:0153 C1            DB      C1
-t

AX=0000  BX=0000  CX=0000  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=075A  CS=075A  IP=0156   NV UP EI PL ZR AC PE CY
075A:0156 C1            DB      C1
```

Starting from BX=200h

BX: 200h -> 80h -> 20h -> 8h -> 2h -> 0h

0010 0000 0000 -> 1000 0000 -> 0010 0000 -> 1000 -> 0010 -> 0000

Looks like right shift two. So perhaps C1 EB is the command, and 02 is the parameter

Author: Assad Ebrahim
assad.ebrahim@alum.swarthmore.edu

```
-r
AX=0000  BX=0200  CX=0000  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=075A  CS=075A  IP=014D   NV UP EI PL ZR AC PE CY
075A:014D C1            DB      C1
-t

AX=0000  BX=0080  CX=0000  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=075A  CS=075A  IP=0150   NV UP EI PL NZ AC PO NC
075A:0150 C1            DB      C1
-t

AX=0000  BX=0020  CX=0000  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=075A  CS=075A  IP=0153   NV UP EI PL NZ AC PO NC
075A:0153 C1            DB      C1
-t

AX=0000  BX=0008  CX=0000  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=075A  CS=075A  IP=0156   NV UP EI PL NZ AC PO NC
075A:0156 C1            DB      C1
-t

AX=0000  BX=0002  CX=0000  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=075A  CS=075A  IP=0159   NV UP EI PL NZ AC PO NC
075A:0159 0A00          OR      AL,[BX+SI]                      DS:0
```

Then BX=0000

Test: 0x0C = 1100b.

Sure enough: 0x0c -> 0x03 -> 0x0 because 1100b -> 0011b -> 0000b

```
-r
AX=0000  BX=000C  CX=0000  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=075A  CS=075A  IP=014D   NV UP EI PL ZR AC PE CY
075A:014D C1            DB      C1
-t

AX=0000  BX=0003  CX=0000  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=075A  CS=075A  IP=0150   NV UP EI PL NZ AC PE NC
075A:0150 C1            DB      C1
-       t

AX=0000  BX=0000  CX=0000  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=075A  CS=075A  IP=0153   NV UP EI PL ZR AC PE CY
075A:0153 C1            DB      C1
```

Test: 0x06 = 0110b.
Sure enough: 0x6 0> 0x1 -> 0x0 because 0110b -> 0001b -> 0000b

Test: 1011 0000b 0xB0
Hypoth: 0010 1100 -> 0000 1011 -> 0000 0010 -> 0000 0000, i.e. B0 -> 2c -> 0b -> 2 -> 0
Exactly right!

Author: Assad Ebrahim
assad.ebrahim@alum.swarthmore.edu

```
-r
AX=0000  BX=00B0  CX=0000  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=075A  CS=075A  IP=014D   NV UP EI PL ZR AC PE NC
075A:014D C1            DB      C1
-t

AX=0000  BX=002C  CX=0000  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=075A  CS=075A  IP=0150   NV UP EI PL NZ AC PO NC
075A:0150 C1            DB      C1
-t

AX=0000  BX=000B  CX=0000  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=075A  CS=075A  IP=0153   NV UP EI PL NZ AC PO NC
075A:0153 C1            DB      C1
-t

AX=0000  BX=0002  CX=0000  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=075A  CS=075A  IP=0156   NV UP EI PL NZ AC PO CY
075A:0156 C1            DB      C1
-t

AX=0000  BX=0000  CX=0000  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=075A  CS=075A  IP=0159   NV UP EI PL ZR AC PE CY
075A:0159 0A00          OR      AL,[BX+SI]                      DS:00
```

Test parameters: Hypoth, the 02 is the number of places to right shift. So we will do right shift by 3, then 2, then 1.

```
075A:014D C1            DB      C1
075A:014E EB03          JMP     0153
075A:0150 C1            DB      C1
075A:0151 EB02          JMP     0155
075A:0153 C1            DB      C1
075A:0154 EB01          JMP     0157
```

Hypoth: 0xb0. Shift by 3 right, 2 right, then 1 right.
1011 0000 -> 0001 0110 -> 0000 0101 -> 0000 0010
B0 -> 16 -> 5 -> 2
Exactly right!

```
AX=0000  BX=00B0  CX=0000  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=075A  CS=075A  IP=014D   NV UP EI PL ZR AC PE CY
075A:014D C1            DB      C1
-t

AX=0000  BX=0016  CX=0000  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=075A  CS=075A  IP=0150   NV UP EI PL NZ AC PO NC
075A:0150 C1            DB      C1
-t

AX=0000  BX=0005  CX=0000  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=075A  CS=075A  IP=0153   NV UP EI PL NZ AC PE CY
075A:0153 C1            DB      C1
-t

AX=0000  BX=0002  CX=0000  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=075A  CS=075A  IP=0156   NV UP EI PL NZ AC PO CY
075A:0156 EBEB          JMP     0143
```

Author: Assad Ebrahim
assad.ebrahim@alum.swarthmore.edu

Conclusion: C1 EB is right shift by hh bits.
I suspect that C1 will match D1, but with literal number of bits hh
Then C1 is rol*16, EB is shr bx, hh is number of bits.

Suspect C0 matches D0 with literal number of bits hh. **Appears correct.**

```
AX=0000  BX=00B0  CX=0000  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=075A  CS=075A  IP=014D   NV UP EI PL ZR AC PE CY
075A:014D C0            DB      C0
-t

AX=0000  BX=0016  CX=0000  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=075A  CS=075A  IP=0150   NV UP EI PL NZ AC PO NC
075A:0150 C0            DB      C0
-t

AX=0000  BX=0005  CX=0000  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=075A  CS=075A  IP=0153   NV UP EI PL NZ AC PE CY
075A:0153 C0            DB      C0
-t

AX=0000  BX=0002  CX=0000  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=075A  CS=075A  IP=0156   NV UP EI PL NZ AC PO CY
075A:0156 C0            DB      C0
-t

AX=0000  BX=0001  CX=0000  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=075A  CS=075A  IP=0159   NV UP EI PL NZ AC PO NC
075A:0159 C0            DB      C0
```

Eb 02 is 0x02EB = 747d
BX went from 0032 (=50d) to 000C (=12d); NG->PL; NA->AC; PO->PE; all else stayed the same.
747d % 50d = 47d = -13d…
We can test this out.

```
075A:014F C1            DB      C1
075A:0150 EB02          JMP     0154
075A:0152 C1            DB      C1
075A:0153 EB02          JMP     0157
075A:0155 C1            DB      C1
075A:0156 64            DB      64
075A:0157 00680A        ADD     [BX+SI+0A],CH
075A:015A 005BC1        ADD     [BP+DI-3F],BL
075A:015D DD00                  FLD     QWORD PTR [BX+SI]
075A:015F 68            DB      68
075A:0160 0A00          OR      AL,[BX+SI]
075A:0162 5B            POP     BX
075A:0163 C1            DB      C1
075A:0164 64            DB      64
075A:0165 008A44FF      ADD     [BP+SI+FF44],CL
```

Author: Assad Ebrahim
assad.ebrahim@alum.swarthmore.edu

```
-r
AX=0098  BX=0032  CX=3E7E  DX=0000  SP=FFFE  BP=0000  SI=01B2  DI=7D02
DS=075A  ES=A000  SS=075A  CS=075A  IP=014F   NV UP EI PL NZ AC PE CY
075A:014F C1            DB      C1
-t

AX=0098  BX=000C  CX=3E7E  DX=0000  SP=FFFE  BP=0000  SI=01B2  DI=7D02
DS=075A  ES=A000  SS=075A  CS=075A  IP=0152   NV UP EI PL NZ AC PE CY
075A:0152 C1            DB      C1
-t

AX=0098  BX=0003  CX=3E7E  DX=0000  SP=FFFE  BP=0000  SI=01B2  DI=7D02
DS=075A  ES=A000  SS=075A  CS=075A  IP=0155   NV UP EI PL NZ AC PE NC
075A:0155 C1            DB      C1
-t

AX=0098  BX=0003  CX=3E7E  DX=0000  SP=FFFE  BP=0000  SI=01B2  DI=7D02
DS=075A  ES=A000  SS=075A  CS=075A  IP=0159   NV UP EI PL ZR AC PE NC
075A:0159 0A00          OR      AL,[BX+SI]                      DS:01B5=00
```

```
-r
AX=0098  BX=000A  CX=3E7E  DX=0000  SP=FFFE  BP=0000  SI=01B2  DI=7D02
DS=075A  ES=A000  SS=075A  CS=075A  IP=015C   NV UP EI PL ZR AC PE NC
075A:015C C1            DB      C1
-t

AX=0098  BX=000A  CX=3E7E  DX=0000  SP=FFFE  BP=0000  SI=01B2  DI=7D02
DS=075A  ES=A000  SS=075A  CS=075A  IP=015F   NV UP EI PL ZR AC PE NC
075A:015F 68            DB      68
-t

AX=0098  BX=000A  CX=3E7E  DX=0000  SP=FFFC  BP=0000  SI=01B2  DI=7D02
DS=075A  ES=A000  SS=075A  CS=075A  IP=0162   NV UP EI PL ZR AC PE NC
075A:0162 5B            POP     BX
-t

AX=0098  BX=000A  CX=3E7E  DX=0000  SP=FFFE  BP=0000  SI=01B2  DI=7D02
DS=075A  ES=A000  SS=075A  CS=075A  IP=0163   NV UP EI PL ZR AC PE NC
075A:0163 C1            DB      C1
-t

AX=0098  BX=000A  CX=3E7E  DX=0000  SP=FFFE  BP=0000  SI=01B2  DI=7D02
DS=075A  ES=A000  SS=075A  CS=075A  IP=0167   NV UP EI PL ZR AC PE NC
075A:0167 44            INC     SP
```

747d / 10d = 74 R 7. BX then holds 3, which is 10-7.
747d / 50d = 14 R 47. BX then holds 12 which is 50-

Appendix 1. Forth Chips

http://www.forth.org/cores.html
Jeff Fox: http://www.ultratechnology.com/chips.htm
J1 Forth CPU (James Bowman): http://www.excamera.com/sphinx/fpga-j1.html
https://hackaday.com/2010/12/01/j1-a-small-fast-cpu-core-for-fpga/

Author: Assad Ebrahim
assad.ebrahim@alum.swarthmore.edu

Multi-computer (parallel Forth chip): Green Arrays GA144: http://www.greenarraychips.com/

(In UMR I studied super-computers with multiple cores. Well, here is now GA144 with 144 cores.)

Green arrays school: http://school.arrayforth.com/

(Andrew Holme) Homemade GPS Receiver: http://www.aholme.co.uk/GPS/Main.htm

(Andrew Holme) Mark 1 pure-TTL Forth Computer: http://www.aholme.co.uk/Mk1/Architecture.htm

(Richard Jones) JonesForth: https://rwmj.wordpress.com/2010/08/07/jonesforth-git-repository/

RTX2000, Novix chip, then shboom, then F18,

RTX2000: According to a couple of ads I kept from back then, the RTX2000 ran more than one Forth instruction per cycle, typically about 16MIPS @ 12MHz, and peaked out at 50MIPS. I posted a scan of one of the ads at http://wilsonminesco.com/stacks/RPN_efficiency.html . You can see by the chart in that ad that as long as you didn't need floating-point, the RTX2000 dramatically outperformed the '386

https://en.wikipedia.org/wiki/RTX2010

Facebook group: https://www.facebook.com/groups/PROGRAMMINGFORTH/

Forth starter: Forth is great. Close to the Hardware. If you want to read about the basiscs – see A Start With Forth at https://wiki.forth-ev.de/doku.php/en:projects:a-start-with-forth:start0 or my current Forth Bookshelf at https://www.amazon.co.uk/Juergen-Pintaske/e/B00N8HVEZM you can build your own CDP1802 there and the EP32 https://www.amazon.co.uk/FIG-Forth-Manual-Documentation-Test-1802-ebook/dp/B01N42VLJE/ref=asap_bc?ie=UTF8 and https://www.amazon.co.uk/EP32-RISC-Processor-Description-Implementation-ebook/dp/B071D3XMPS/ref=asap_bc?ie=UTF8 for 1802 see as well the Core at https://wiki.forth-ev.de/doku.php/projects:fig-forth-1802-fpga:start

Starting Forth: https://www.forth.com/starting-forth/

Thinking Forth:

Footsteps Empty Valley: Tings Footsteps book explains about the beginnings of Forth on Gates https://www.amazon.co.uk/Footsteps-Empty-Valley-issue-3-ebook/dp/B06X6JGM5L/ref=asap_bc?ie=UTF8

> > A Forth Story
> > Allen Cekorich
> > Walnut Creek, California
> > Forth Dimensions, July/August 1995

Arduino, Cortex-M, PIC24 (16-bit), 8-bit Atmel, PIC16 (8-bit)

Author: Assad Ebrahim
assad.ebrahim@alum.swarthmore.edu

Objective. This worksheet uses DEBUG to design and write a program, and sees how Forth is a natural evolution of assembly language.

Context A computer works with numbers. Humans need to see those numbers and provide input from a keyboard or input device that works with digits. A fundamental interface is the conversion between REGISTER/MEMORY VALUES (16-bits) to ASCII codes for input/output, i.e. NUMERIC TO ASCII CONVERSION.

Going from number to asci, we have binihex (probably better as hex2asc). Going from ascii to number, we have asc2hex.

Program: binhex (3Dh bytes (62d bytes), less than 40h bytes = 64d bytes)

C7 06 00 02 00 F0 B1 04 53 5B 53 8B 16 00 02 21
D3 B0 04 FE C9 F6 E1 51 88 C1 D3 EB B1 04 D3 EA
89 16 00 02 B2 30 80 FB 0A 7C 02 B2 37 00 DA B4
02 CD 21 59 83 F9 00 77 D0 5B B4 4C CD 21

```
-u100
073F:0100 C706000200F0  MOV     WORD PTR [0200],F000
073F:0106 B104          MOV     CL,04
073F:0108 53            PUSH    BX
073F:0109 5B            POP     BX
073F:010A 53            PUSH    BX
073F:010B 8B160002      MOV     DX,[0200]
073F:010F 21D3          AND     BX,DX
073F:0111 B004          MOV     AL,04
073F:0113 FEC9          DEC     CL
073F:0115 F6E1          MUL     CL
073F:0117 51            PUSH    CX
073F:0118 88C1          MOV     CL,AL
073F:011A D3EB          SHR     BX,CL
073F:011C B104          MOV     CL,04
073F:011E D3EA          SHR     DX,CL
073F:0120 89160002      MOV     [0200],DX
073F:0124 B230          MOV     DL,30
073F:0126 80FB0A        CMP     BL,0A
073F:0129 7C02          JL      012D
073F:012B B237          MOV     DL,37
073F:012D 00DA          ADD     DL,BL
073F:012F B402          MOV     AH,02
073F:0131 CD21          INT     21
073F:0133 59            POP     CX
073F:0134 83F900        CMP     CX,+00
073F:0137 77D0          JA      0109
073F:0139 5B            POP     BX
073F:013A B44C          MOV     AH,4C
073F:013C CD21          INT     21
```

Entering

Author: Assad Ebrahim
assad.ebrahim@alum.swarthmore.edu

```
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0100   NV UP EI PL NZ NA PO NC
073F:0100 0000          ADD     [BX+SI],AL                      DS:0000=CD
-e100
073F:0100  00.C7   00.06   00.00   00.02   00.00   00.F0   00.B1   00.04
073F:0108  00.53   00.5B   00.53   00.8B   00.16   00.00   00.02   00.21
073F:0110  00.D3   00.B0   00.04   00.FE   00.C9   00.F6   00.E1   00.51
073F:0118  00.88   00.C1   00.D3   00.EB   34.B1   00.04   2E.D3   07.EA
073F:0120  00.89   00.16   00.00   00.02   00.B2   00.30   00.80   00.FB
073F:0128  00.0A   00.7C   00.02   00.B2   00.37   00.00   00.DA   00.B4
073F:0130  00.02   00.CD   00.21   00.59   00.83   00.F9   00.00   00.77
073F:0138  00.D0   00.5B   00.B4   00.4C   00.CD   00.21

-rbx
BX 0000
:F1E2
-r
AX=0000  BX=F1E2  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0100   NV UP EI PL NZ NA PO NC
073F:0100 C706000200F0  MOV     WORD PTR [0200],F000            DS:0200=0000
```

Save it to file
Nbinihex2.com
Rbx
0000
Rcx
3e (63d)
W

When loading to run
Nbinihex2.com
L
Rsp
00fd
Rbx
F1e2
G

Pushing onto the stack: sub SP, 2 then mov word data to sp

```
-r
AX=0000  BX=F1E2  CX=0004  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0108   NV UP EI PL NZ NA PO NC
073F:0108 53            PUSH    BX
-t

AX=0000  BX=F1E2  CX=0004  DX=0000  SP=00FB  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0109   NV UP EI PL NZ NA PO NC
073F:0109 5B            POP     BX
-d00f0 00ff
073F:00F0  00 00 00 00 00 09 01 3F-07 A3 01 E2 F1 00 00 73   .......?...
```

Author: Assad Ebrahim
assad.ebrahim@alum.swarthmore.edu

There's the first F printed

```
AX=020C  BX=000F  CX=0004  DX=0F46  SP=00F3  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=F000  IP=14A1   NV UP EI PL NZ AC PO NC
F000:14A1 FE38          ???     [BX+SI]                         DS:000F=03
-t
F
```

Then the 1 printed

```
AX=0208  BX=0001  CX=0004  DX=0031  SP=00F3  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=F000  IP=14A1   NV UP EI PL NZ NA PO NC
F000:14A1 FE38          ???     [BX+SI]                         DS:0001=20
-t
1
```

Then the E printed

```
AX=0204  BX=000E  CX=0004  DX=0045  SP=00F3  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=F000  IP=14A1   NV UP EI PL NZ AC PO NC
F000:14A1 FE38          ???     [BX+SI]                         DS:000E=8A
-t
E
```

Then the 2 printed

```
AX=0200  BX=0002  CX=0004  DX=0032  SP=00F3  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=F000  IP=14A1   NV UP EI PL NZ NA PO NC
F000:14A1 FE38          ???     [BX+SI]                         DS:0002=3E
-t
2
```

So this works.

What does this look like in Assembly?
See binihex_v2.asm

What would this look like in Forth?

```
: adjust-mask ( mask -- mask' ) 4 rshift ;
: print-char ( digit -- ) dup $A < if $30 else $37 then + emit ;
: binihex ( val -- ) $f000 4 0 do 2dup and 3 i - 4 * rshift print-char adjust-mask loop 2drop ;
```

Further reflections and program of work

06-Aug-2018 (Mon 11:00)
Reflections on Assembly, C, Forth, and Chip Design
What do we learn from studying the fire.com program?

It is possible to do a surprising amount with very little! The key here is utilizing the complex instructions that 8086 provides (e.g. repz/stosw or repz/movsw), though with a bit of expansion one could make this into a loop with a different processor (more bytes, possibly slower).

Trying to understand a program from hand-optimized machine code is tough but do-able; doing the same from

Author: Assad Ebrahim
assad.ebrahim@alum.swarthmore.edu

automatically generated machine code by an assembler is harder; from a compiler like GCC it is almost impossible. Why?

- machine code sends machine instructions with no additional context; you know *what* but you don't know *why*

- assemblers (in theory) attempt a one-to-one mapping between assembler instructions (mnemonics) and machine instructions (opcodes), using directives to guide the choice of the right opcode. The assembly code should be well commented --- this is what should provide the context.

C (3GL) provided structured programming by trading off on time and space costs because the subroutines it introduced added space (additional code) and execution time to preserve the processor state and then restore it. It provided portability by targeting a tightly specified "generic" processor, against which interface then targeted compilers could be written to optimize for various things, be it execution time, size, whatever. So the toolset expands to knowledge of this generic processor and how a compiler will translate to the real underlying machine. Most programmers won't need to know about the particular processor.
This is the principle of abstraction layers and interfaces (design architecture).

The principle is that information hiding through abstraction and interfaces makes it possible for teams to work in parallel, and the work of each is complementary to the work of the other. The better and tighter the compiler writers are, the better the eventual code is. The non-shifting target for the application designers frees up more of their time and thought to writing applications instead of keeping up with processor changes. The cost of providing (basic) compilers can be passed on to the chip developers who are incentivized to provide correct and good tools so that their chips are adopted. The cost of R&D to develop better compiler techniques is passed on to the professional compiler writers who are incentivized to make theirs much better than the ordinary ones provided by the chip makers, to justify costs of getting these professional tools.
There is a lot in the above design that follows modern economic thinking: specialization for greater productivity, incentives for innovation, competition for continuous improvement, and the belief that all of this is better for humanity as a whole.

Assembly (2GL) did this without portability, and without requiring then the complex syntax parsing needing a compiler. An assembler is a simpler thing because of the mapping between mnemonics and opcodes. It can be simpler because it requires the programmer to understand very well his hardware. I.e. the programmer is programming the hardware directly, and using the assembler to take out the tedium of counting bits, bytes, maintaining references by hand, and coding the structures manually.
What do assembler procs do? Do they preserve registers?
What is the view with Forth?
Forth (4GL) aims at the same ends as C (although in a lot less friendly format) and adds a further advantage: dynamic expansion of the language itself. I.e. in Forth, the language itself is not static, and the programmer is simultaneously writing his own compiler. What makes it great is that Forth does not surrender the territory of any adjacent areas, and does not exact as high a trade-off cost on either time or space. The way it does this is by an ingenious invention --- the Forth engine.
This is what gets entered to allow Forth to run on a targeted chip.
To write this takes some effort, but once it is done, it is small enough that one could in principle enter this by hand, boot strapping it.

The other view that Forth then takes is to ask why are the processors so complicated? Do we need such complex processors?

Author: Assad Ebrahim
assad.ebrahim@alum.swarthmore.edu

In C, this is not asked because the space and time trade-offs and the heaviness of the compilers and toolsets mean that it becomes a tenet that faster and more memory is better and this is sold to those on the other side, who see this as allowing them to not have to spend time optimizing (premature optimization mantra...)

Now this is a very interesting viewpoint.
Imagine a Forth chip and coding in Forth.
Two highly targeted systems. You wouldn't need much else. And the rest would be all application energy.
But what would you need to make this happen?
Vision and dictat. I.e. the free hand of economics would have to be controlled for the overall interest.

So this is this unfortunate limitation of economics --- it searches for local optimizations initially, but as competition gets more fierce, sub-optimal positions are taken and energy is spent maintaining those because of the power of the vested interests. So it is neither locally optimal not globally optimal.
Regulation can try to go for local optimality, but it can never get global optimality if this is in a different part of the design space. Not without dictat.
That is then why we have nations and tribes -- and why there is revolution, evolution, and overthrowing of old paradigms by new ones. It takes disruption.
Who does the disrupting?
Someone with the discipline and vision to make it apparent to all that there is a better way, i.e. the peaceable demonstration.

So it is not about converting through preaching, or legislation, or mandate, but DO IT, make it work, demonstrate it, and it if really is better, then it will disrupt.
Sometimes you need a killer app to do that.
This ends the DEBUG module of the course.
(The overall course has 4 modules: starts with DEBUG, moves to Assembly, then C, and finally Forth)