Advanced x86:
BIOS and System Management Mode Internals
Reset Vector

Xeno Kovah && Corey Kallenberg

LegbaCore, LLC

All materials are licensed under a Creative Commons “Share Alike” license.

http://creativecommons.org/licenses/by-sa/3.0/

You are free:

to Share — to copy, distribute and transmit the work

to Remix — to adapt the work

Under the following conditions:

Attribution — You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

Share Alike — If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

Attribution condition: You must indicate that derivative work
"Is derived from John Butterworth & Xeno Kovah’s ’Advanced Intel x86: BIOS and SMM’ class posted at http://opensecuritytraining.info/IntroBIOS.html”

Reset Vector
Execution Environment

Real-Address Mode (Real Mode)

- The original x86 operating mode
- Referred to as “Real Mode” for short
- Introduced way back in 8086/8088 processors
- Was the only operating mode until Protected Mode (with its "virtual addresses") was introduced in the Intel 286
- Exists today solely for compatibility so that code written for 8086 will still run on a modern processor
  - Someday processors will boot into protected mode instead
- In the BIOS’ I have looked at, the general theme seems to be to get out of Real Mode as fast as possible
- Therefore we won’t stay here long either

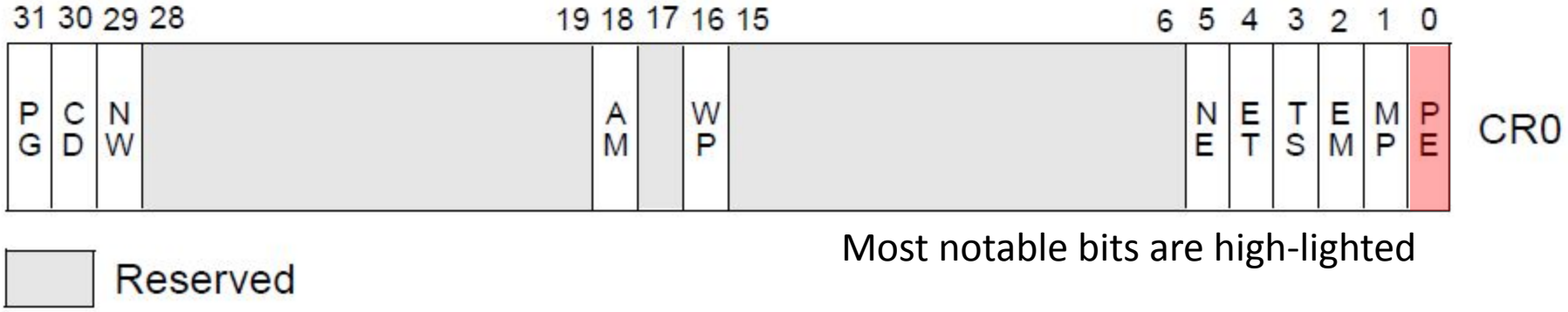
Processor State After Reset

E6400 Registers at Reset

Name	Value
EAX	00000000
EBX	00000000
ECX	00000000
EDX	00010676*
EBP	00000000
ESI	00000000
EDI	00000000
ESP	00000000
CS	F000
DS	0000
SS	0000
ES	0000
FS	0000
GS	0000
EIP	0000FFF0
EFLAGS	00000002

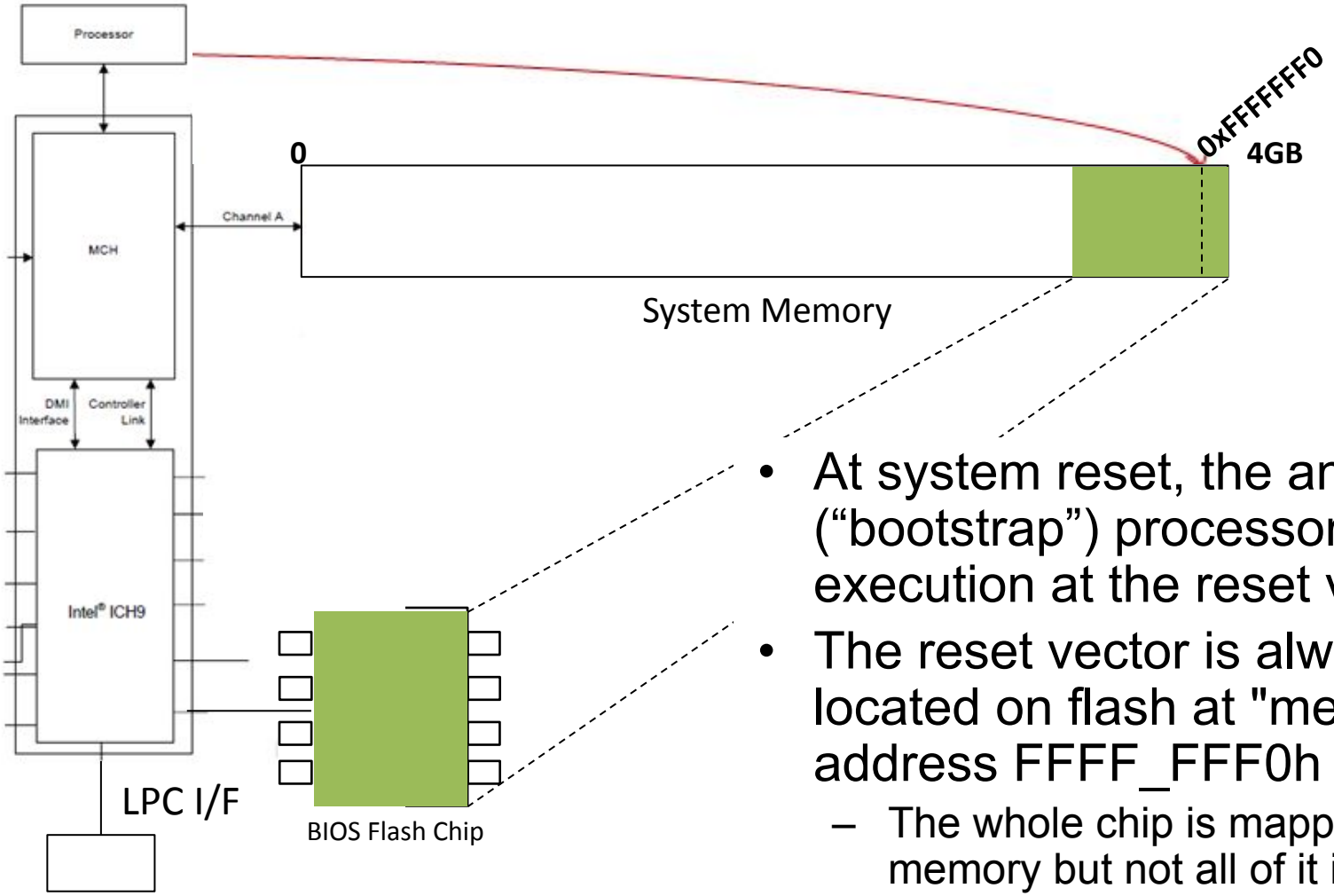
- EAX, EBX, ECX, EBP, ESI, EDI, ESP are all reset to 0
- EDX contains the CPU stepping identification information
  - Same info returned in EAX when CPUID is called with EAX initialized to '1'
  - *This will vary of course, the value in the table to the left corresponds to the Core2Duo inside the E6400
- The base registers are 0 with the exception of CS which is initialized with F000
- EIP (or IP since it's 16-bit mode) is initialized with (0000)FFF0
  - CS:IP = F:FFF0h
- EFLAGS is 00000002h
  - Only hard-coded bit 1 is asserted
  - If I were sitting at a breakpoint at the entry vector, then bit 16 (resume flag) would be asserted indicating that debug exceptions (#DB) are disabled.

Processor State After Reset: Control Registers (CRs)



- Control registers CR2, CR3, and CR4 are all 0
- CR0 is 6000_0010h (likely since Pentium)
- Paging (bit 31) is disabled
  - All linear addresses are treated as physical addresses
- Protection Enable (bit 0) is 0
  - *0 indicates that we are in Real Mode*
  - 1 indicates we are in Protected Mode
- All the other bits are 0

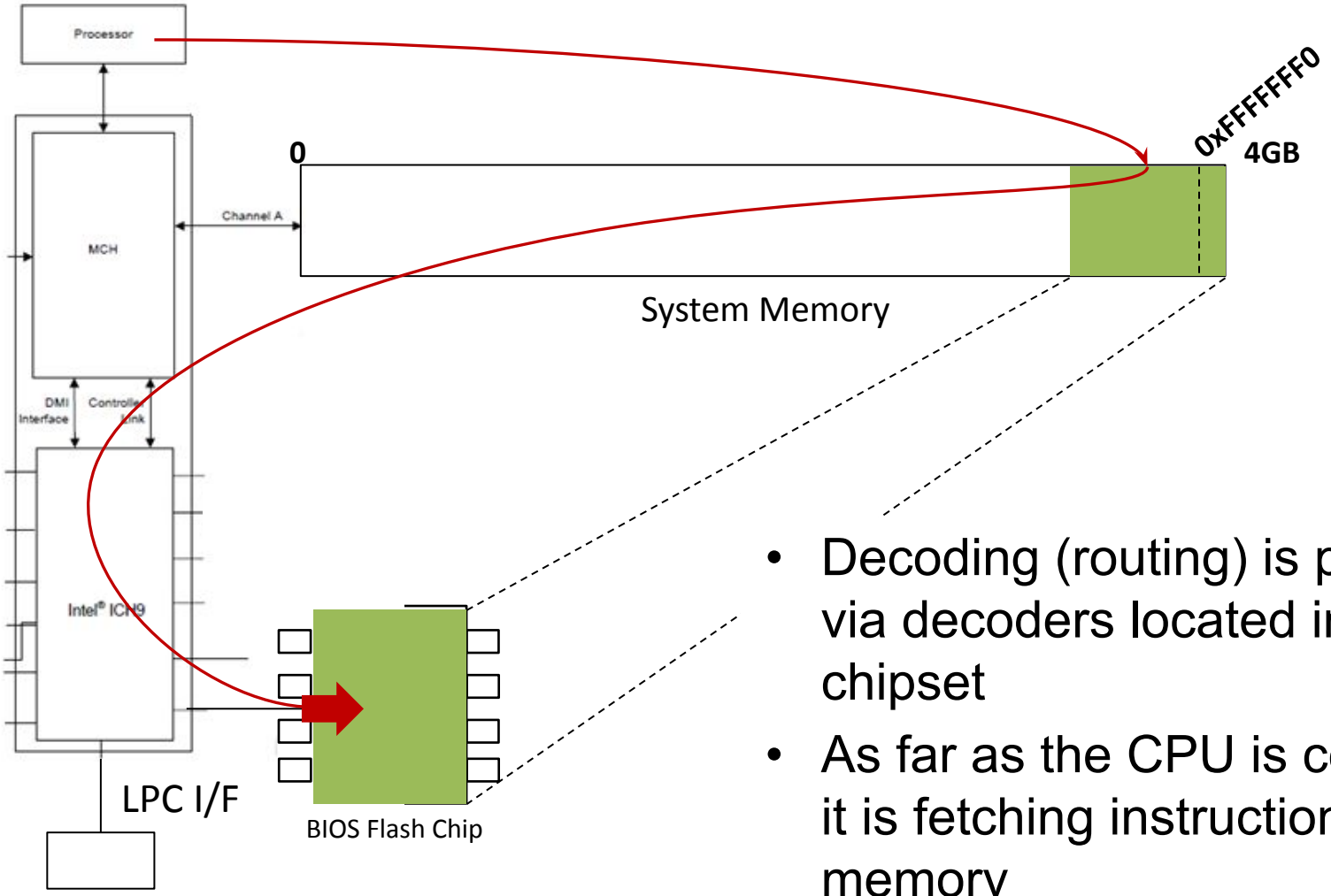
Reset Vector



- At system reset, the an initial ("bootstrap") processor begins execution at the reset vector
- The reset vector is always located on flash at "memory" address FFFF_FFF0h
  - The whole chip is mapped to memory but not all of it is readable due to protections on the flash device itself

www.intel.com/.../**datasheet/io-controller-hub-9-datasheet**.pdf

Reset Vector Decoding



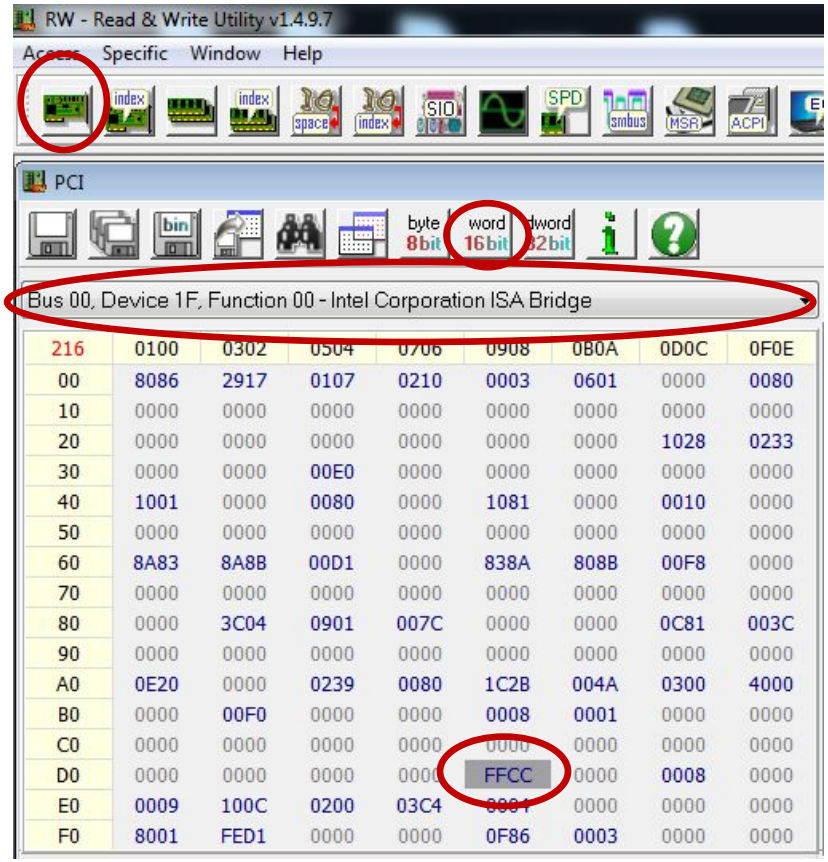
- Decoding (routing) is performed via decoders located in the chipset
- As far as the CPU is concerned it is fetching instructions from memory
- But in fact it's from the SPI flash

www.intel.com/.../**datasheet/io-controller-hub-9-datasheet**.pdf

Aside: Forensics People

- If the top of memory always contains a memory-mapped copy of part of the SPI flash chip, that means it should theoretically show up in memory forensic dumps (e.g. those given out by memory forensic challenges)
- I've never had time to test this, but you should see if you can go grab some memory forensics dumps and determine whether there is a complete copy of the BIOS in the memory dump, or only a partial copy (and if partial, where it ends)
  - Probably should start by testing on a system you have known BIOS dump for
  - As I mentioned before, virtual machines have virtual BIOSes, so you could also determine if the dump was taken off a virtual machine by comparing against some virtual BIOSes
- Let me know what you find! :)
  - A volatility plugin to carve BIOS out of memdumps would be cool ☺️
    - IIRC someone might have done this now, but I can't find the link again…

Mini-Lab: BIOS Flash Decoding



- Let's look at some of the decoding (routing) of the BIOS to memory
- Open RW Everything and click on the PCI tab to open up the PCI window
- Click the drop-down tab and select Bus 00, Device 1F, Function 00
- This is the LPC device
- Click on the Word 16 bit button to arrange the PCI configuration registers into 16-bit words
- Notice word offset D8-D9h

Mini-Lab: BIOS Flash Decoding

FWH_DEC_EN1—Firmware Hub Decode Enable Register (LPC I/F—D31:F0)

Offset Address: D8h–D9h Attribute: R/W, RO
Default Value: FFCFh Size: 16 bits

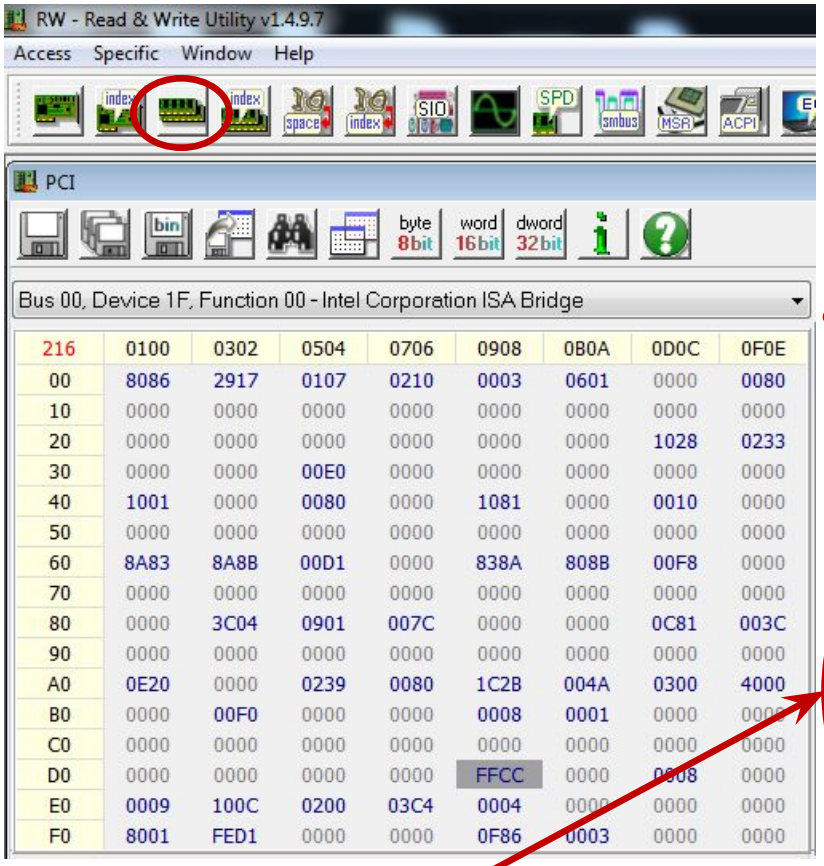
Bit	Description
15	**FWH_F8_EN** — RO. This bit enables decoding two 512-KB Firmware Hub memory ranges, and one 128-KB memory range. 0 = Disable 1 = Enable the following ranges for the Firmware Hub FFF80000h – FFFFFFFFh FFB80000h – FFBFFFFFh
14	**FWH_F0_EN** — R/W. This bit enables decoding two 512-KB Firmware Hub memory ranges. 0 = Disable. 1 = Enable the following ranges for the Firmware Hub: FFF00000h – FFF7FFFFh FFB00000h – FFB7FFFFh
	FWH_E8_EN — R/W. This bit enables decoding two 512-KB Firmware Hub memory ranges.

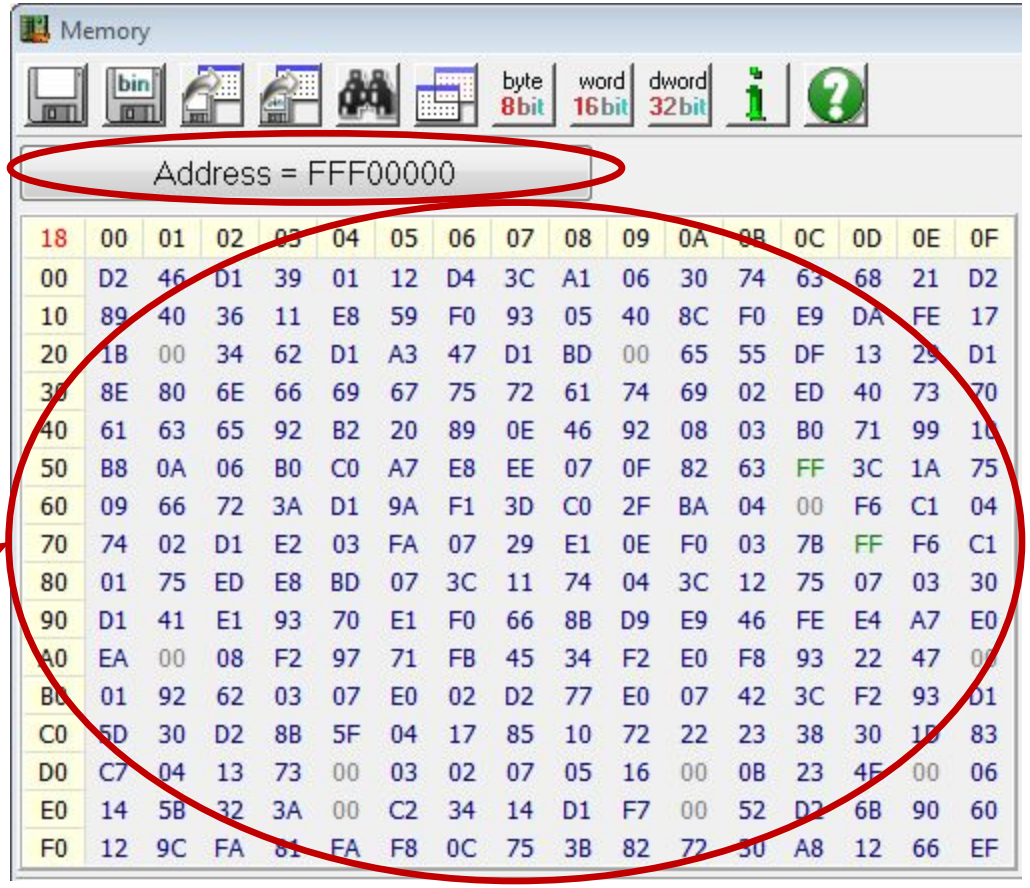
Note: "FWH" is substituted with "BIOS" in the above in the newer datasheets

- Offset D8-D9h is FWH_DEC_EN1
- As stated, this controls the decoding of ranges to the FWH
- If your system uses SPI and not a Firmware Hub (and it does since FWH is very rare), it still decodes to the SPI BIOS
- We want bit 14 which decodes FFF0_0000h – FFF7_FFFFh

Mini-Lab: BIOS Flash Decoding

- Click Memory button and type address FFF00000





- Therefore, with FWH_DEC_EN bit 14 asserted, we're decoding to a portion of BIOS binary

Mini-Lab: BIOS Flash Decoding

- This memory range is still read-only
- This example is to help provide a picture of the initial boot environment

PCI

byte 8bit | word 16bit | dword 32bit

Bus 00, Device 1F, Function 00 - Intel Corporation ISA Bridge

216	0100	0302	0504	0706	0908	0B0A	0D0C	0F0E
00	8086	2917	0107	0210	0003	0601	0000	0080
10	0000	0000	0000	0000	0000	0000	0000	0000
20	0000	0000	0000	0000	0000	0000	1028	0233
30	0000	0000	00E0	0000	0000	0000	0000	0000
40	1001	0000	0080	0000	1081	0000	0010	0000
50	0000	0000	0000	0000	0000	0000	0000	0000
60	8A83	8A8B	00D1	0000	838A	808B	00F8	0000
70	0000	0000	0000	0000	0000	0000	0000	0000
80	0000	3C04	0901	007C	0000	0000	0C81	003C
90	0000	0000	0000	0000	0000	0000	0000	0000
A0	0E20	0000	0239	0080	1C2B	004A	0300	4000
B0	0000	00F0	0000	0000	0008	0001	0000	0000
C0	0000	0000	0000	0000	0000	0000	0000	0000
D0	0000	0000	0000	0000	BFCC	0000	0008	0000
E0	0009	100C	0200	03C4	0004	0000	0000	0000
F0	8001	FED1	0000	0000	0F86	0003	0000	0000

Hardware

Memory

byte 8bit | word 16bit | dword 32bit

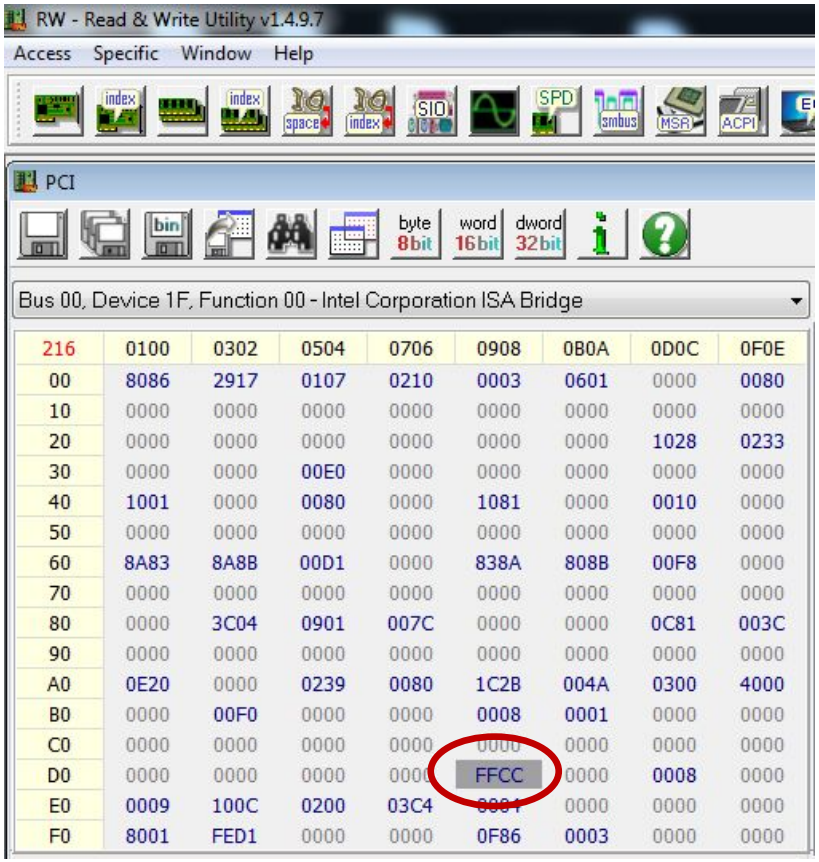
Address = FFF00000

18	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
10	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
20	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
30	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
40	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
50	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
60	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
70	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
80	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
90	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
A0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
B0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
C0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
D0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
E0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
F0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF

Hardware

- De-assert bit 14 (set to 0xBFCC)
- Decoded to memory now

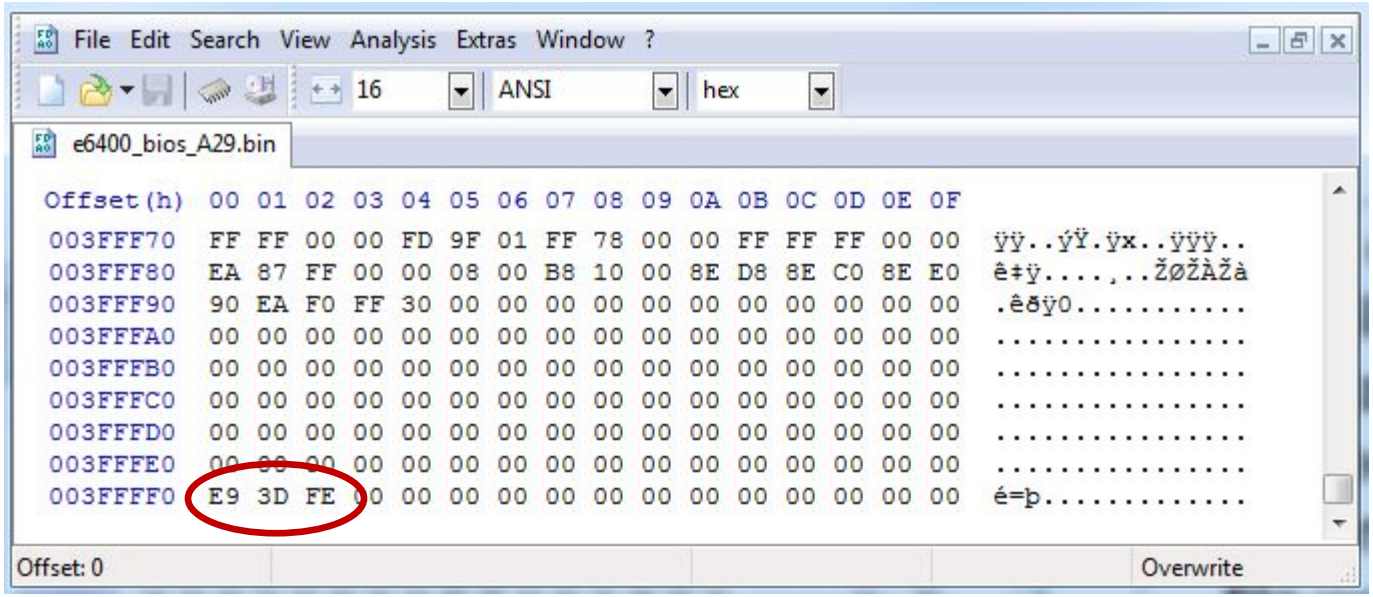
Mini-Lab: BIOS Flash Decoding



216	0100	0302	0504	0706	0908	0B0A	0D0C	0F0E
00	8086	2917	0107	0210	0003	0601	0000	0080
10	0000	0000	0000	0000	0000	0000	0000	0000
20	0000	0000	0000	0000	0000	0000	1028	0233
30	0000	0000	00E0	0000	0000	0000	0000	0000
40	1001	0000	0080	0000	1081	0000	0010	0000
50	0000	0000	0000	0000	0000	0000	0000	0000
60	8A83	8A8B	00D1	0000	838A	808B	00F8	0000
70	0000	0000	0000	0000	0000	0000	0000	0000
80	0000	3C04	0901	007C	0000	0000	0C81	003C
90	0000	0000	0000	0000	0000	0000	0000	0000
A0	0E20	0000	0239	0080	1C2B	004A	0300	4000
B0	0000	00F0	0000	0000	0008	0001	0000	0000
C0	0000	0000	0000	0000	0000	0000	0000	0000
D0	0000	0000	0000	0000	FFCC	0000	0008	0000
E0	0009	100C	0200	03C4	0004	0000	0000	0000
F0	8001	FED1	0000	0000	0F86	0003	0000	0000

- Reset it back to 0xFFCC
- Couple of notes:
- Your original values may differ since BIOS flips them on and off as the developers decided necessary
- Bit 15 is Read Only and <u>always</u> asserted

Mini-data-collection Lab: Reset Vector in BIOS Binary



- If we dump the BIOS and look at it in a hex editor, at the end of the file we will see a jump instruction (near, relative jump)
- The chipset aligns the flash so that the limit of the BIOS region (always either the only/last region on the flash) aligns with address FFFF_FFF0h
- The CPU executes these instructions in 16-bit Real Mode

Real Mode Memory

- 16-bit operating mode
- Segmented memory model
- When operating in real-address mode, the default addressing and operand size is 16 bits
- An address-size override can be used in real-address mode to enable access to 32-bit addressing (like the extended general-purpose registers EAX, EDX, etc.)
- However, the maximum allowable 32-bit linear address is still 000F_FFFFH (2^{20} -1)
- So how can it address FFFF_FFF0h?
  - We'll answer that in a bit

Real Mode Addressing: Segment Registers

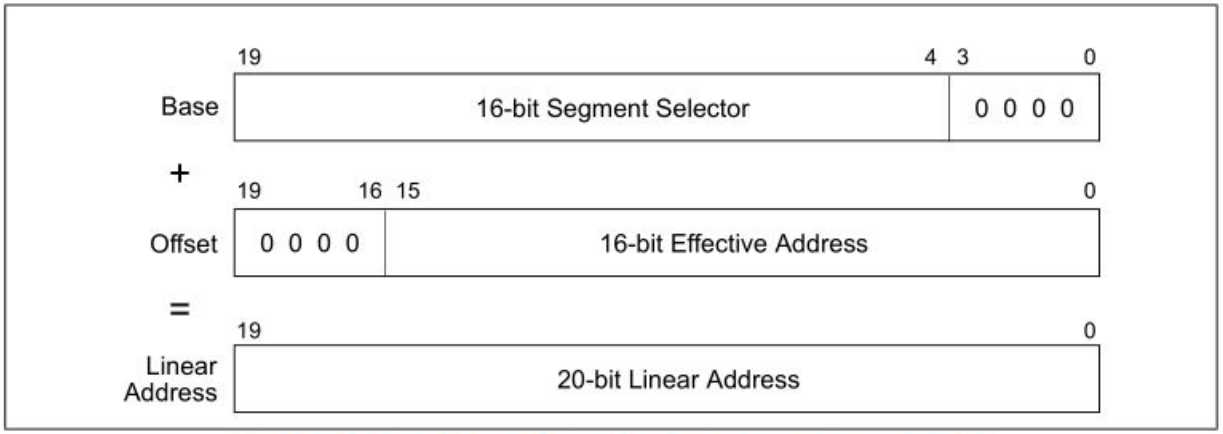


Figure 20-1. Real-Address Mode Address Translation

- CS, DS, SS, ES, FS, GS
- Only six segments can be active at any one time
- 16-bit segment selector contains a pointer to a memory segment of 64 Kbytes (max)
- 16-bit Effective address can access up to 64KB of memory address space
- Segment Selector combines with effective address to provide a 20-bit linear address
- So an application running in real mode can access an address space of up to 384 KB at a time (including stack segment) without switching segments

Real Mode Addressing

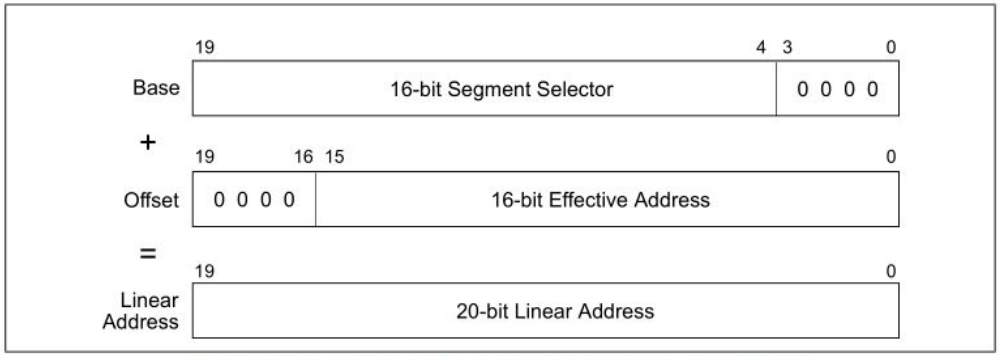


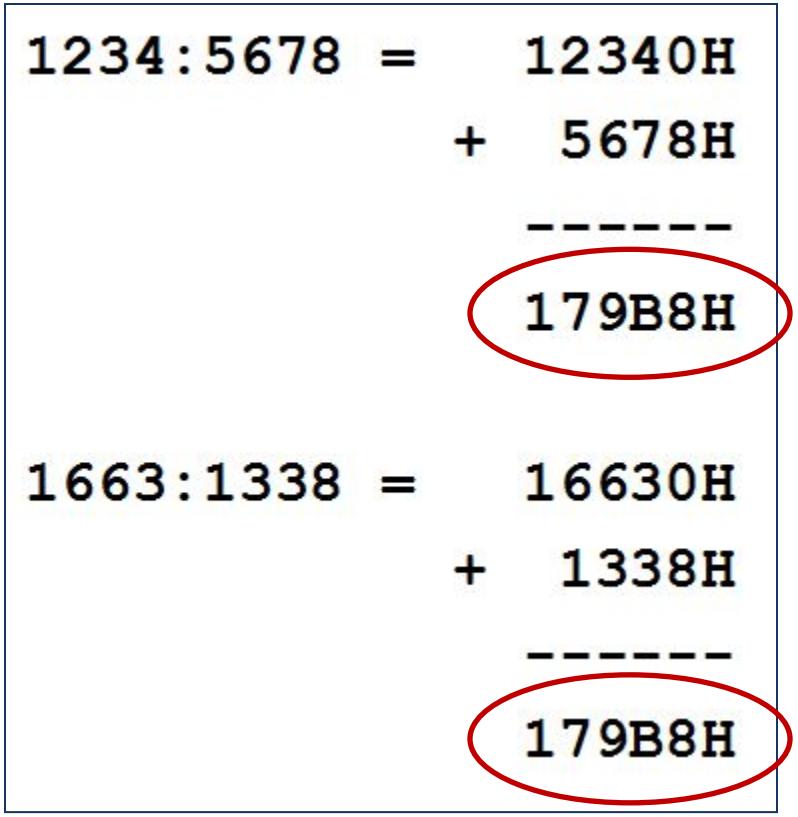
Figure 20-1. Real-Address Mode Address Translation

```
1234:5678 =   12340H
          +    5678H
          ------
           179B8H
```

- As shown in Figure 20-1 in the Intel SW Developers guide
- The Segment Selector (CS, DS, SS, etc.) is left-shifted 4 bits
- The 16-bit Segment Selector is then added to a 16-bit effective address (or offset if you will) within the segment
- Remember, upon entry into the BIOS, all linear addresses are translated as physical (per CR0)

Intel Developers Manual, 20.1.1

Real Mode Addressing Problem: Overlap



- Addresses in different segments can overlap
- Given such a limited environment it's no wonder we want to choose a different operating mode as soon as possible

Intel Developers Manual, 20.1.1

F:FFF0 != FFFF:FFF0

- Every segment register has a “visible” part and a “hidden” part.
- Intel sometimes refers to the “hidden part” as the “descriptor cache”
- It’s called “cache” because it stores the descriptor info so that the processor doesn’t have to resolve it each time a memory address is accessed

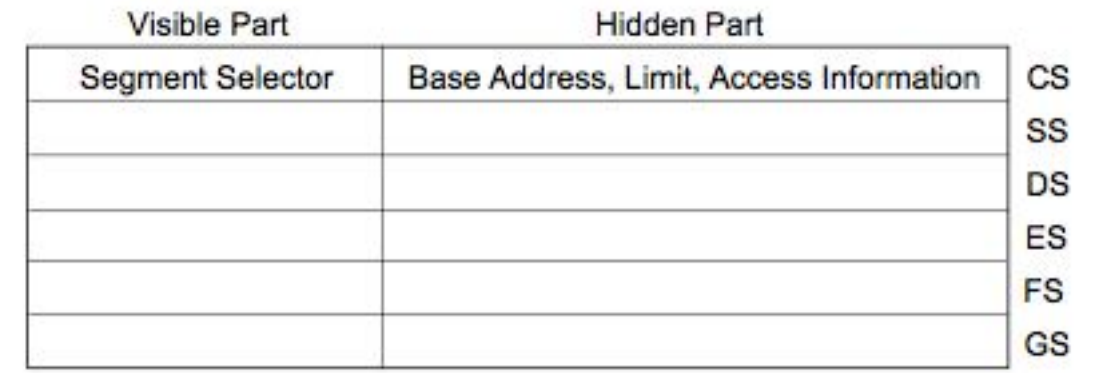


Figure 3-7. Segment Registers

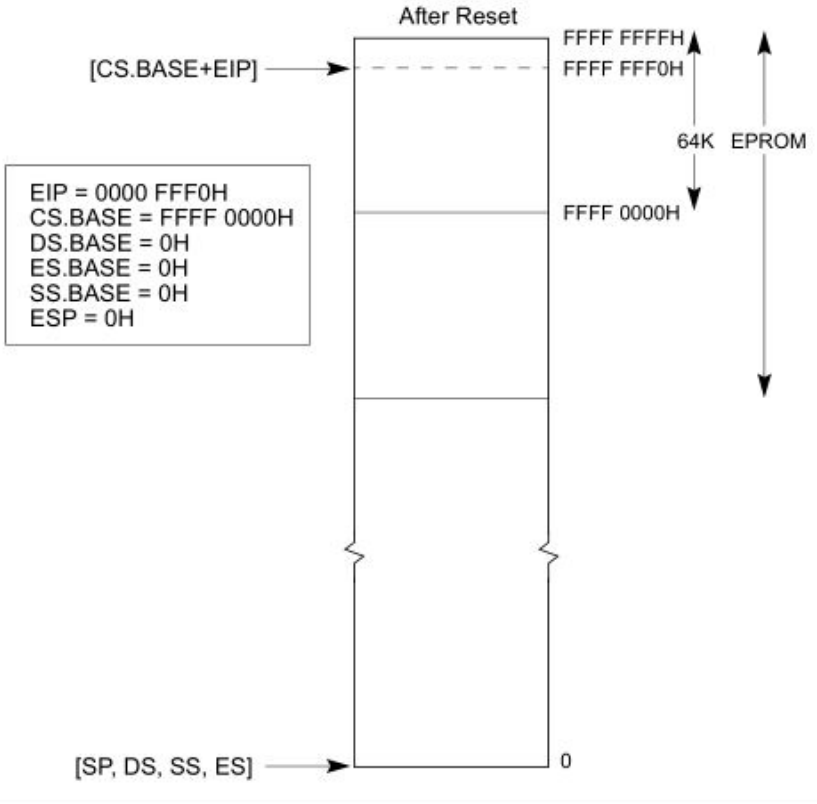
Descriptor Cache

- “When a segment selector is loaded into the visible part of a segment register, the processor also loads the hidden part of the segment register with the base address, segment limit, and [access information] from the segment descriptor pointed to by the segment selector.”
- Real Mode doesn’t have protected mode style access-control so the [access information] part is ignored
- This means that the hidden part isn’t modified until after a value is loaded into the segment selector
- So the moment CS is modified, the CS.BASE of FFFF_0000H is replaced with the new value of CS (left shifted 4 bits)

Intel SW Dev, Vol 3, Sec 3.4.3

CS.BASE + EIP

- *CS.BASE is pre-set to FFFF_0000H upon CPU reset/power-up*
- EIP set to 0000_FFF0H
- So even though CS is set to F000H, CS.BASE+EIP makes FFFF_FFF0H
- So when you see references to CS:IP upon power-up being equal to F:FFF0h, respectively, now you know how what it really means and how it equates to an entry vector at FFFF_FFF0h

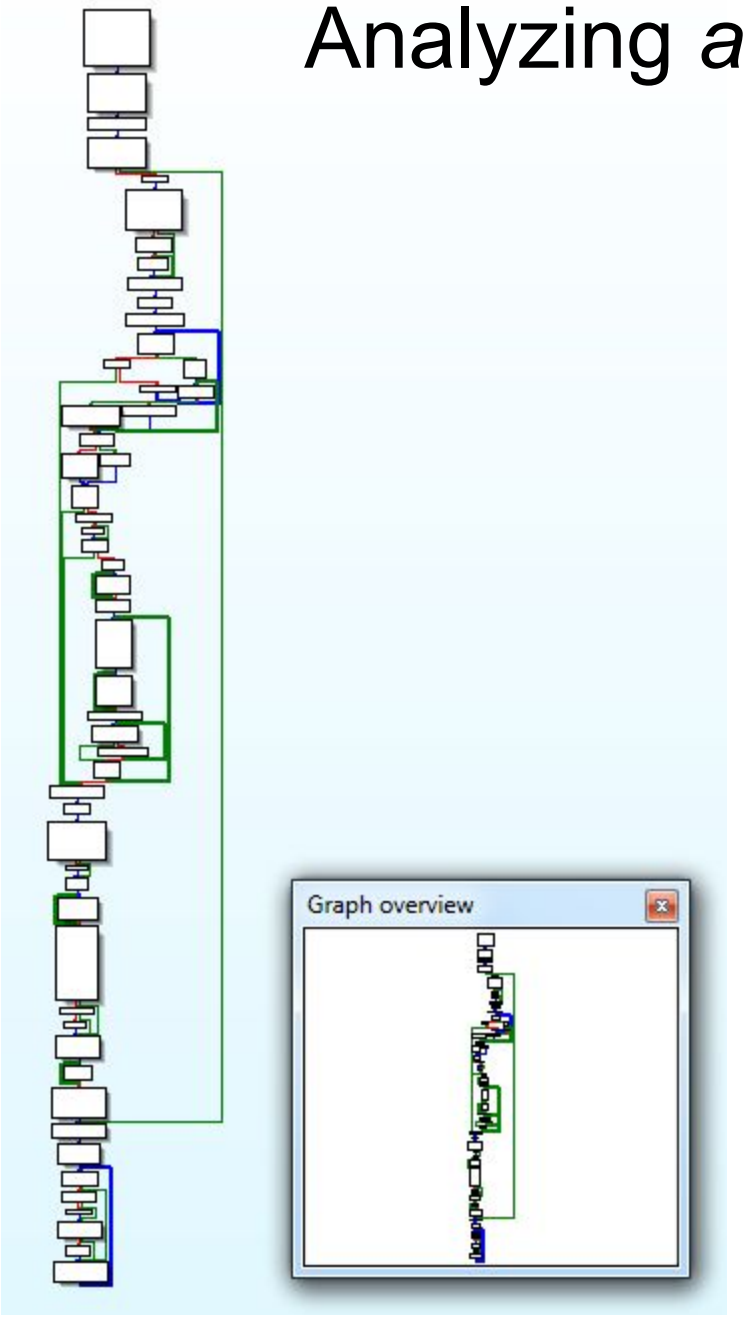


Vol. 3, Figure 9-3

Reset Vector

- So upon startup, while the processor stays in Real Mode, it can access only the memory range FFFF_0000h to FFFF_FFFFh.
- If BIOS were to modify CS while still in Real Mode, the processor would only be able to address 0_0000h to F_FFFFh.
  - PAM0 helps out by mapping this range to high memory (another decoder)
- So therefore if your BIOS is large enough that it is mapped below FFFF_0000H and you want to access that part of it, you best get yourself into Protected Mode ASAP.
  - And this is typically what they do

Analyzing *any* x86 BIOS Binary

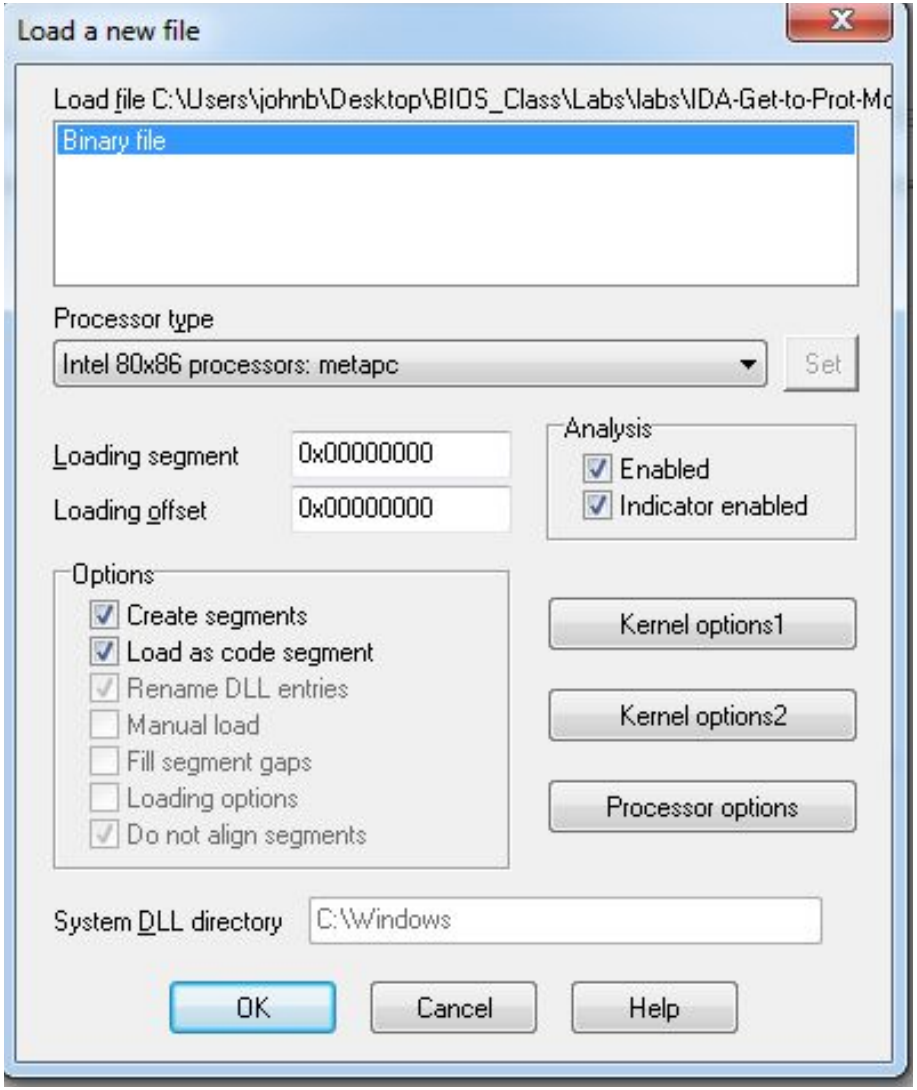


- With UEFI we can usually skip straight to analyzing code we care about.
- But what if you want to analyze a legacy BIOS, or some other non-UEFI x86 BIOS like CoreBoot?
- In that case you may need to do as the computer does, and really read starting from the first instruction
- The subsequent slides provide the generic process to do that

A dream deferred

- We're going to hold off on the rest of the entry vector analysis for now, and go back to it later if we have time.
  - We never have time ;)
- I left the slides in here for if you want to try to go through an equivalent process
  - Note: I know the slides are a little hard to follow and occasionally make jumps in intuition. I've been wanting to clean these up from John's version, but haven't had time

1: Disassemble the BIOS Binary

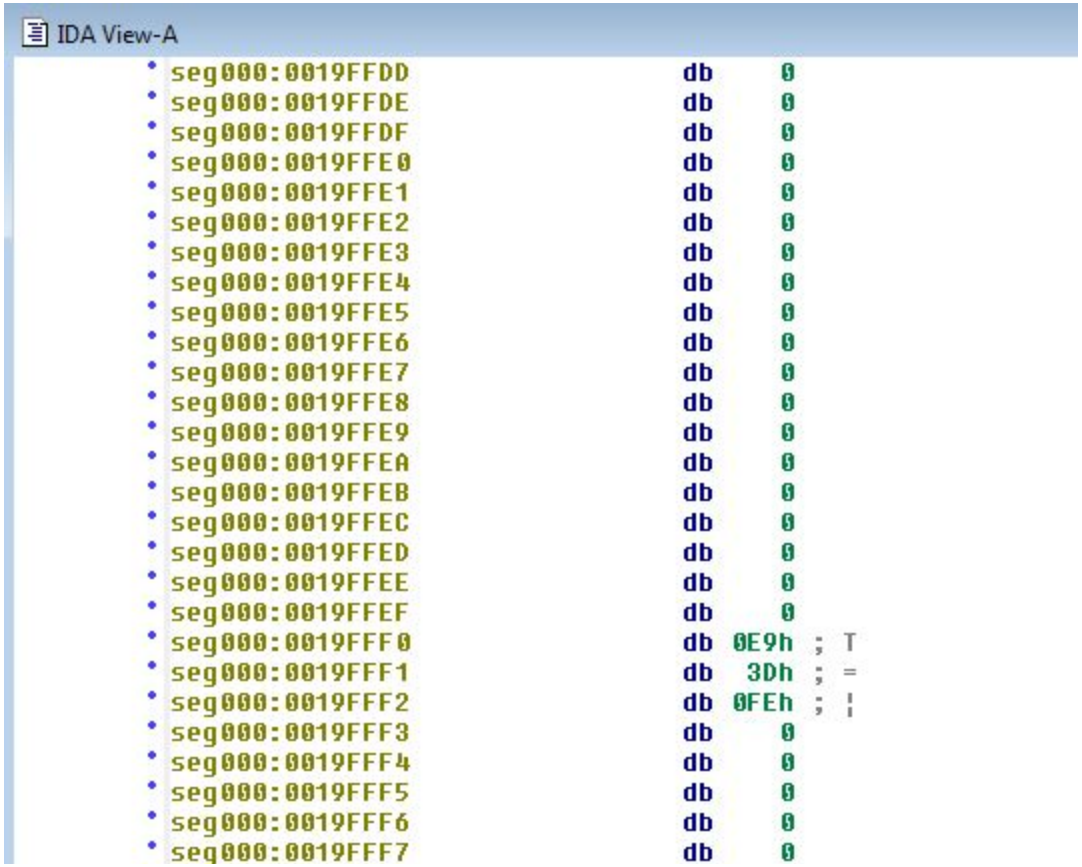


- Acquire a dump of the BIOS flash from a tool like Flashrom or Copernicus and open it in IDA
- Intel 80x86 metapc setting is fine regardless of IDA version
- Choose to disassemble in 32-bit mode
- Not a typo, most BIOS' jump into 32-bit protected mode as soon as possible
  - If your BIOS is much older, just edit the segment to 16-bit
- I have the full version of IDA Pro but am using Free version 5.0 to show you that this works with that version
- Other debuggers like OllyDbg should also work

FIXME

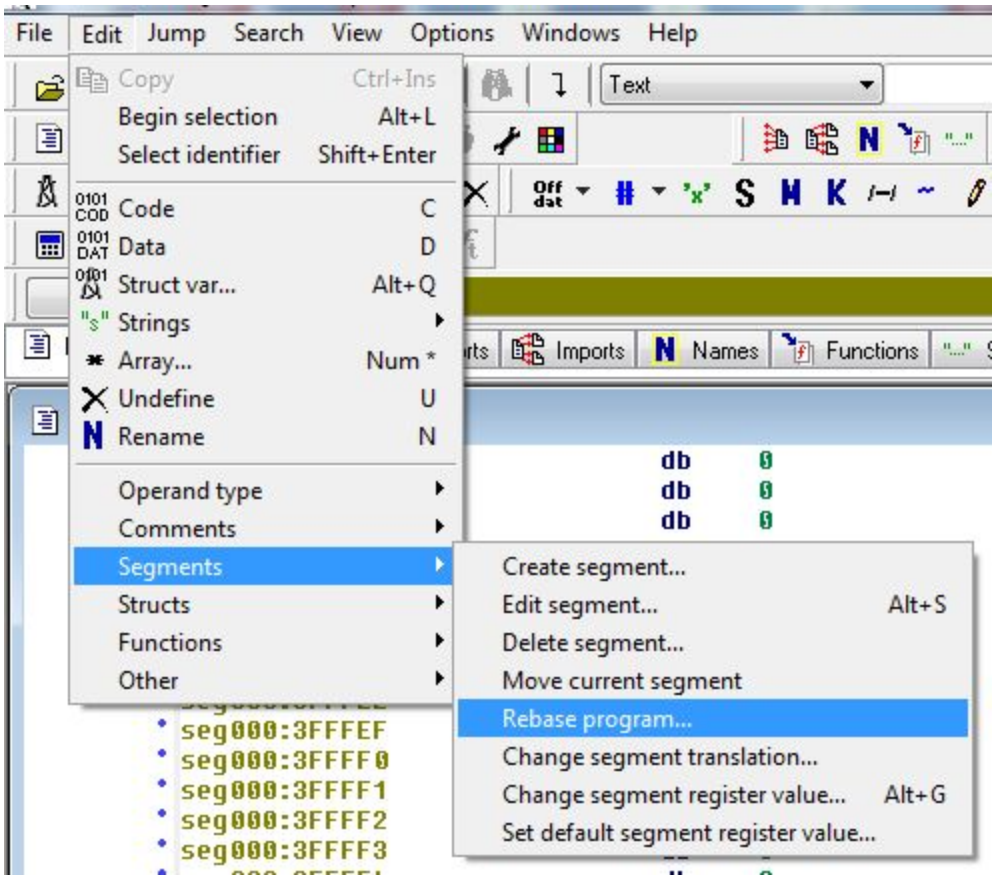
- Update procedure for new IDA demo 6.6

2: Rebase the Program



- First thing we're going to do is rebase the program
- We know the entire image of this BIOS is mapped to memory so that its upper address boundary is at FFFF_FFFFh with the entry vector at FFFF_FFF0h
- Let's touch these up to reflect this

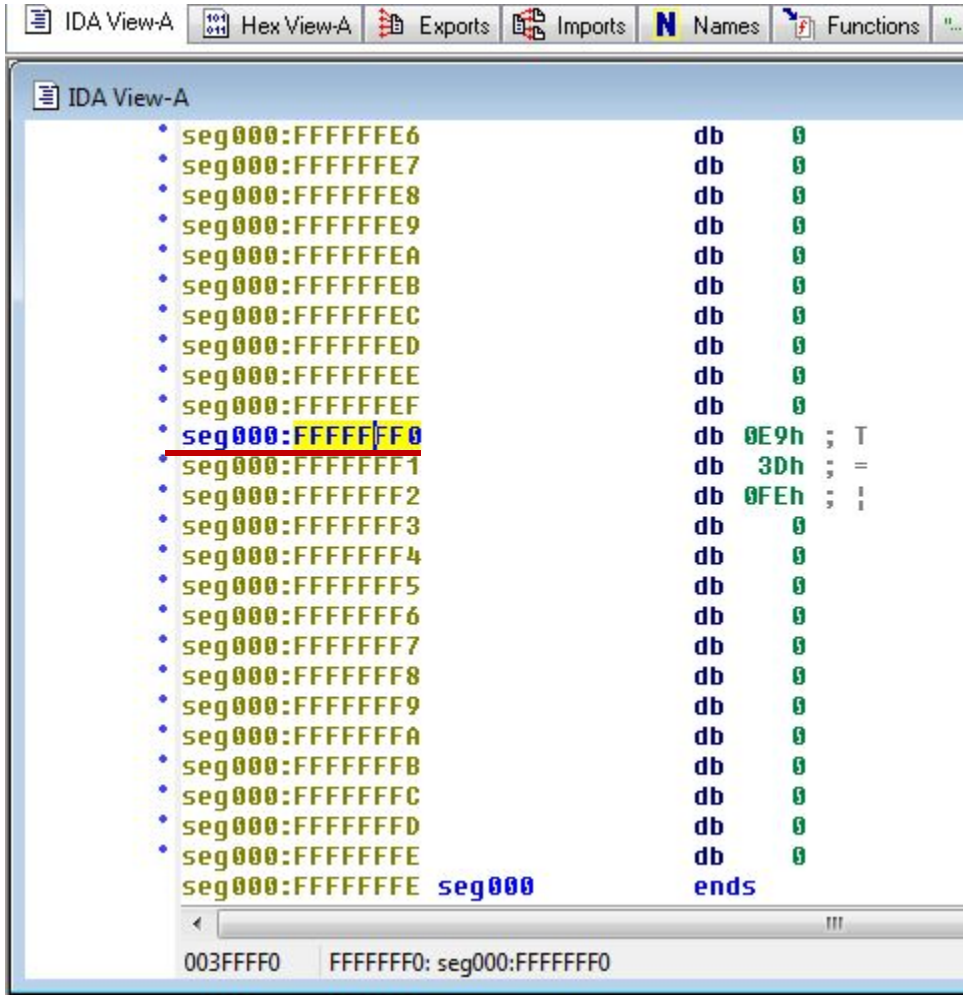
2.1: Rebase the Program



- In this lab our file contains only the BIOS portion of the flash.
- The value to enter is:
- 4 GB – (Size of BIOS Binary)
- For this lab it is 0xFFE60000
  - (for BIOS Length 1A0000h)
- Example: If you had a 2 MB BIOS binary you would rebase the program to FFE0_0000h
- The idea is for the entry vector at FFFF_FFF0h in memory to be displayed in IDA at linear address FFFF_FFF0h

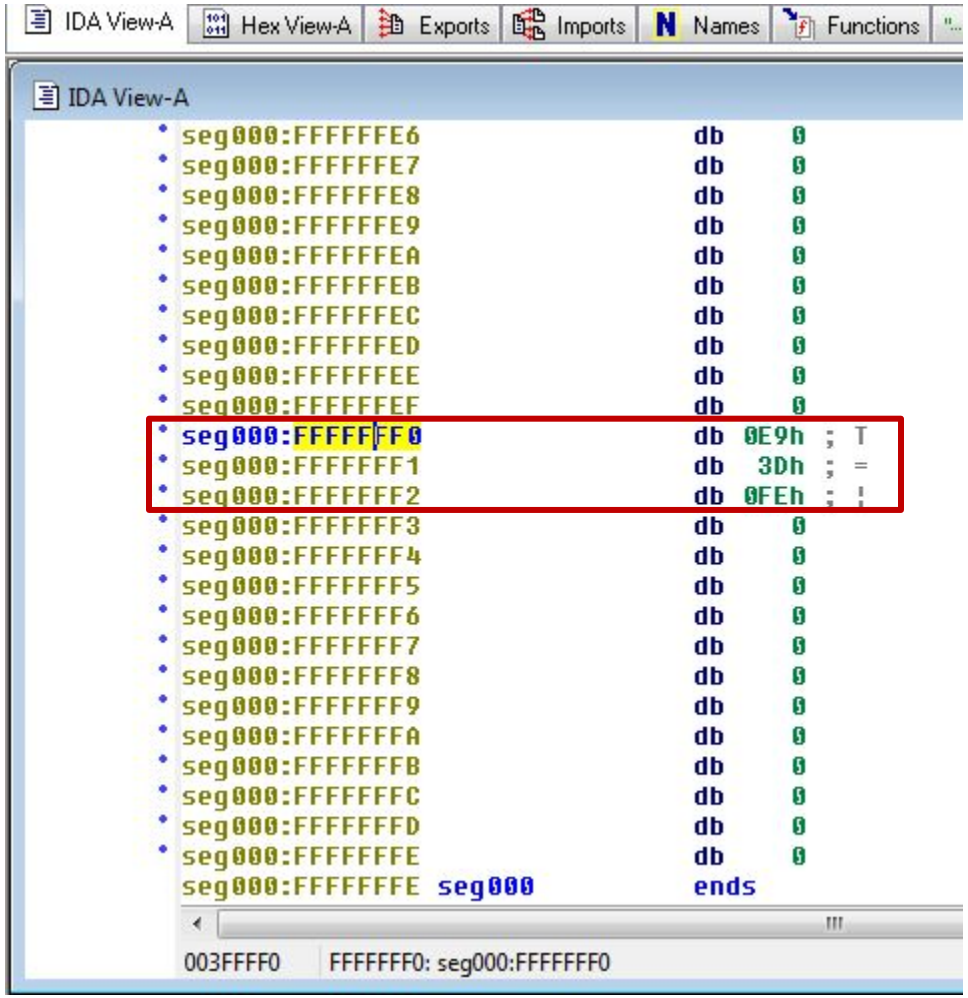
! If you encounter a size-related error, open the binary file with a hex editor (like HxD) and delete the last byte. Then re-open the binary in IDA and rebase it. Still treat it like it were its original size.

2.2: Rebase the Program



- You know you have done it right when you see executable instructions at FFFF_FFF0h, such as:
- E9 3D FE
- E9 is a relative JMP instruction (JMP FE3Dh)
- Note: The JMP instruction may be preceded by a WBINVD instruction or a couple NOP instructions
  - In this case, these instructions will be at FFFF_FFF0h instead of the JMP
- There always will be a JMP here following those

3. Determine IDA Segments: Manually Analyze the Reset Vector JMP



- So now we want to create some IDA segments to help us (and IDA) interpret the disassembly
- One goal is to keep the 16-bit segment that contains the entry vector as small as possible
  - From experience, BIOS takes a FAR JMP away from here after entering protected mode
- JMP FE3Dh is relative to the address following the JMP:
- FFFF_FFF3h, in this case

3.1: JMP rel16

JMP—Jump

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
EB *cb*	JMP *rel8*	D	Valid	Valid	Jump short, RIP = RIP + 8-bit displacement sign extended to 64-bits
E9 *cw*	JMP *rel16*	D	N.S.	Valid	Jump near, relative, displacement relative to next instruction. Not supported in 64-bit mode.
E9 *cd*	JMP *rel32*	D	Valid	Valid	Jump near, relative, RIP = RIP + 32-bit displacement sign extended to 64-bits

- The address following our JMP instruction is FFFF_FFF3h
  - We'll treat it like a 64KB segment (FFF3h) for easier readability
  - Technically it is a 64KB segment so we don't have to worry about this assumption throwing off our calculation
- Take the 2's compliment of the operand in the JMP FE3Dh instruction:
  1. (FE3Dh – 1) = FE3Ch
  2. ~FE3Ch = 01C3h
- Subtract this displacement from the address following the JMP instruction to find the destination:
- FFF3h – 01C3h = **FE30h**

Intel SW Developers Guide, Vol. 2, Intel Instruction Set Reference

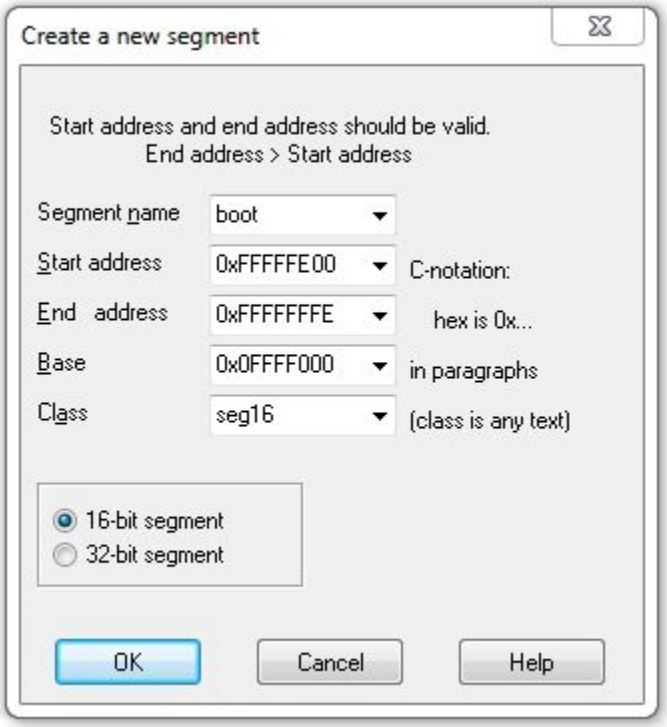
3.2: Determine Segment Boundary

```
seg000:FFFFFDFD                 db 0FFh
seg000:FFFFFDFE                 db 0FFh
seg000:FFFFFDFF                 db 0FFh
seg000:FFFFFE00                 db  44h ; D
seg000:FFFFFE01                 db  65h ; e
seg000:FFFFFE02                 db  6Ch ; l
seg000:FFFFFE03                 db  6Ch ; l
seg000:FFFFFE04                 db  20h
seg000:FFFFFE05                 db  53h ; S
seg000:FFFFFE06                 db  79h ; y
seg000:FFFFFE07                 db  73h ; s
seg000:FFFFFE08                 db  74h ; t
seg000:FFFFFE09                 db  65h ; e
seg000:FFFFFE0A                 db  6Dh ; m
seg000:FFFFFE0B                 db  20h
seg000:FFFFFE0C                 db  4Ch ; L
seg000:FFFFFE0D                 db  61h ; a
seg000:FFFFFE0E                 db  74h ; t
seg000:FFFFFE0F                 db  69h ; i
seg000:FFFFFE10                 db  74h ; t
seg000:FFFFFE11                 db  75h ; u
seg000:FFFFFE12                 db  64h ; d
seg000:FFFFFE13                 db  65h ; e
seg000:FFFFFE14                 db  20h
```

- So we know the destination of the JMP at the entry vector is FFFF_FE30h
- We can now make an assumption that the address FFFF_FE00h can serve as a segment boundary for us
- Our goal is to keep the segment containing the entry JMP as small as possible
- The assumption is that code will be aligned and will take a far JMP to a lower address space
- This assumption is based on experience, but could vary
- Remember these are segments to help IDA translate our disassembly, not necessarily mimic the system

4: Create Initial 16-bit Segment

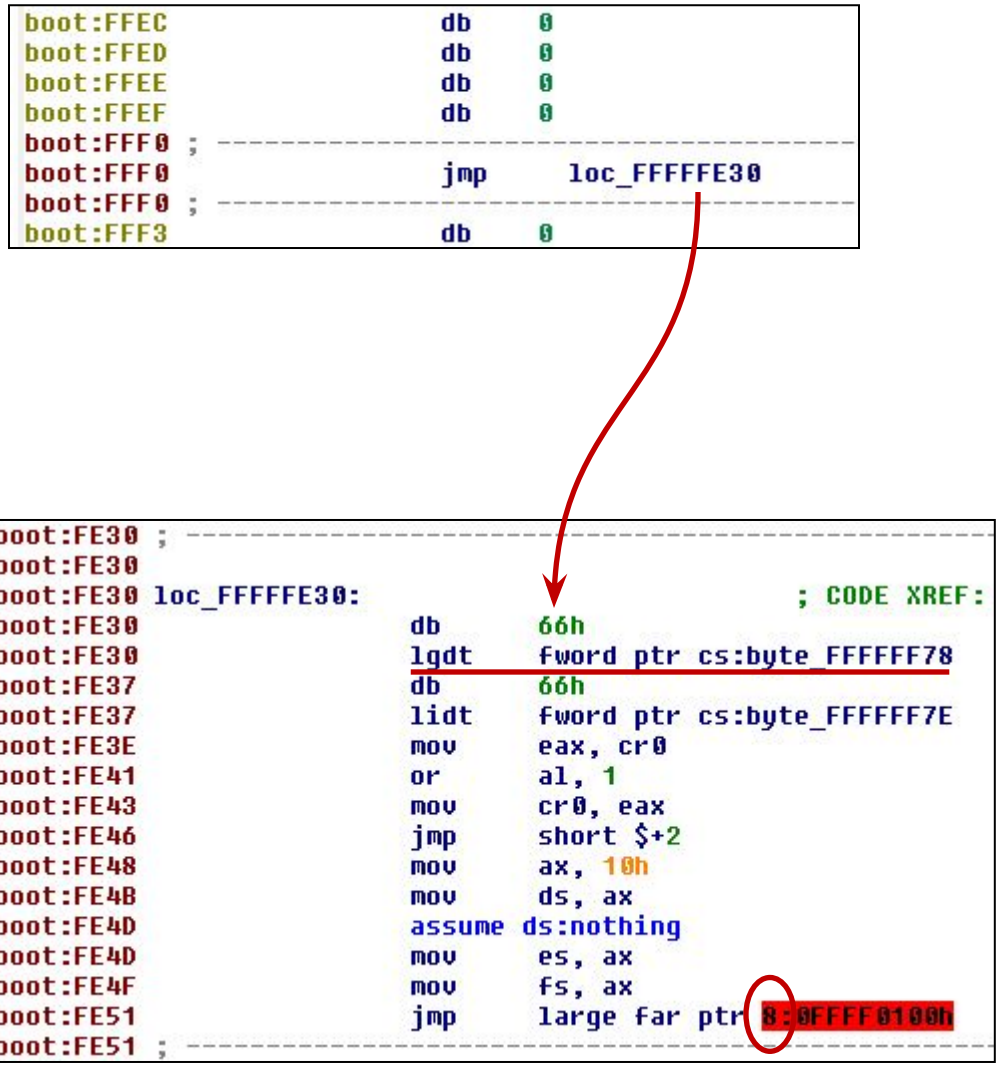
```
db    0
db    0
db    0
db    0
db    0
db    0
db    0
db    0
db    0
db 0E9h ; T
db  3Dh ; =
db 0FEh ; ¦
db    0
db    0
db    0
db    0
db    0
db    0
db    0
db    0
db    0
db    0
db    0
db    0

db    0
ends
```



- Edit –> Segments –> Create Segment
- Pick any segment name you want
- Class can be any text name
- 16-bit segment
- Start Address = 0xFFFFFE00
- End Address = 0xFFFFFFFE
  - Remember: IDA Does not like the address FFFFFFFF (-1) !!
  - Actually, according to IDA documentation, the 32-bit version of IDA doesn't "like" any address at or above FF00_0000h ☹
- Base = 0x0FFFF000
  - CS.BASE = FFFF_0000h on boot

```
VirtualAddress = LinearAddress - (Base << 4)
          FFF0         FFFF:FFF0 - (Base << 4)
```

5: Identify Memory Model



- Once this segment is created, IDA "automagically" recognizes the destination of the entry vector jump
- What we see here is the BIOS preparing to enter protected mode
- Likely it will be using a flat memory model
- Note the '8' in the far jump operand
- That references the entry at offset 8 in the GDT
- Now let's look at that LGDT instruction

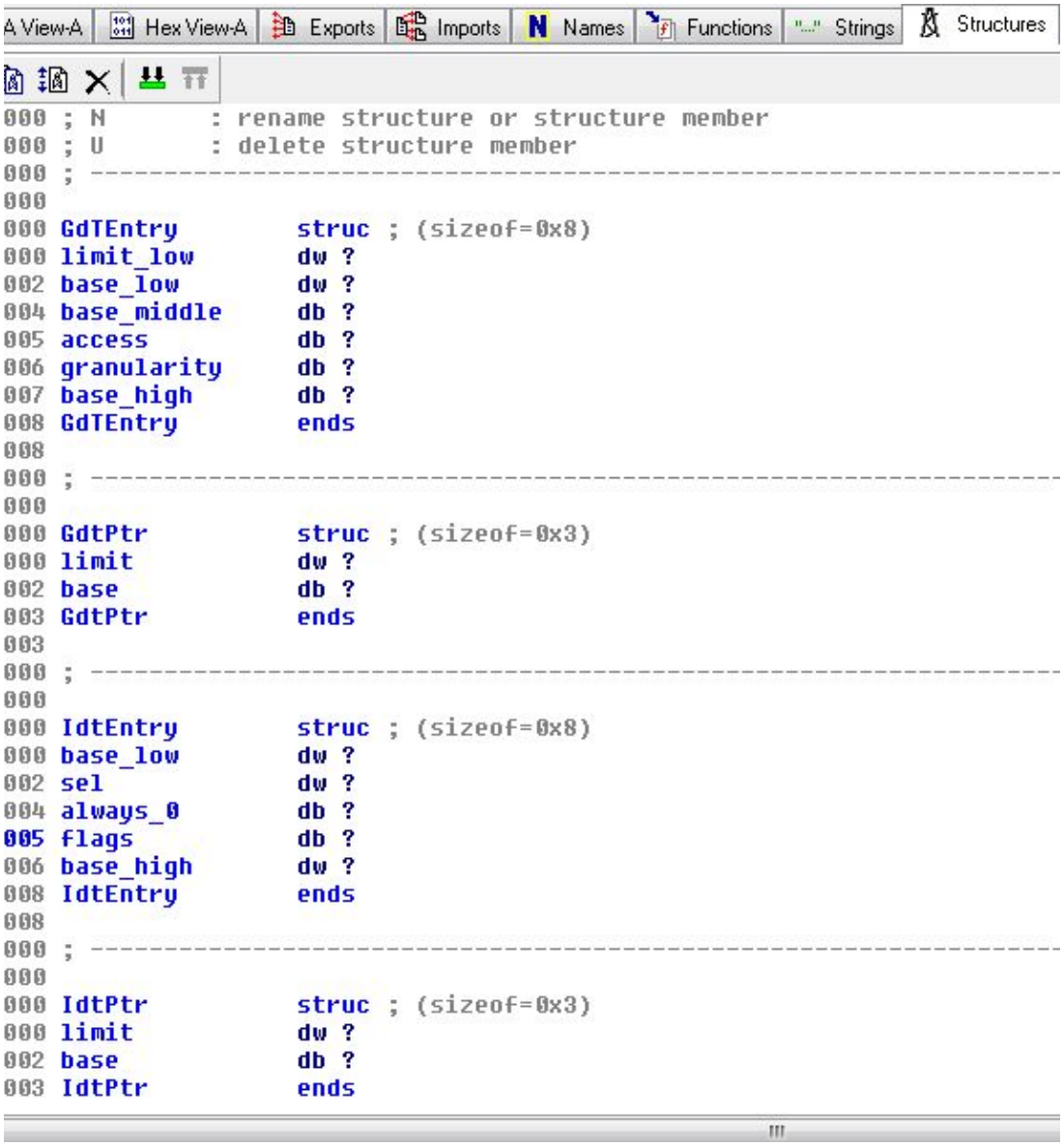
5.1: LGDT Instruction

LGDT/LIDT—Load Global/Interrupt Descriptor Table Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 01 /2	LGDT *m16&32*	M	N.E.	Valid	Load *m* into GDTR.
0F 01 /3	LIDT *m16&32*	M	N.E.	Valid	Load *m* into IDTR.
0F 01 /2	LGDT *m16&64*	M	Valid	N.E.	Load *m* into GDTR.
0F 01 /3	LIDT *m16&64*	M	Valid	N.E.	Load *m* into IDTR.

- LGDT loads the values in the source operand into the global descriptor table register (GDTR)
- The operand specifies a 6-byte structure containing the size of the table (2-bytes) and a 4-byte pointer to the location of the table data
- The table data contains segment bases, limits, access rights
- More than likely it will be a single base of 0000_0000h and a limit of FFFF_FFFFh
- If this is true, then they are using a Flat Memory Model
  - And you shall rejoice!
  - Really there is no point in not using the flat memory model, you can generally just assume they are

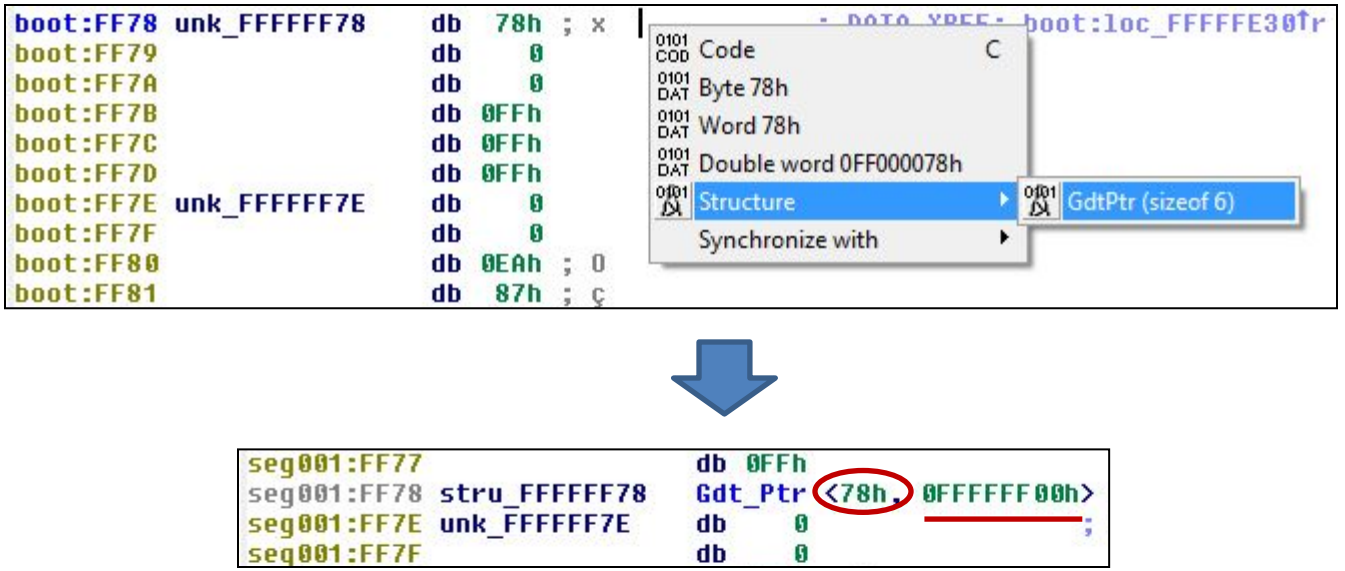
5.2: Import GDT/IDT Structures



- You can import these structures into IDA by parsing the file “descriptors.h”
- Screenshot included so you can enter them manually if necessary
- IDT structures are also provided
- Importing structures like this is very useful for analyzing BIOS
- Legacy BIOS is filled with proprietary structure definitions
- Contrasted with UEFI structures which are defined in a publically-released standard

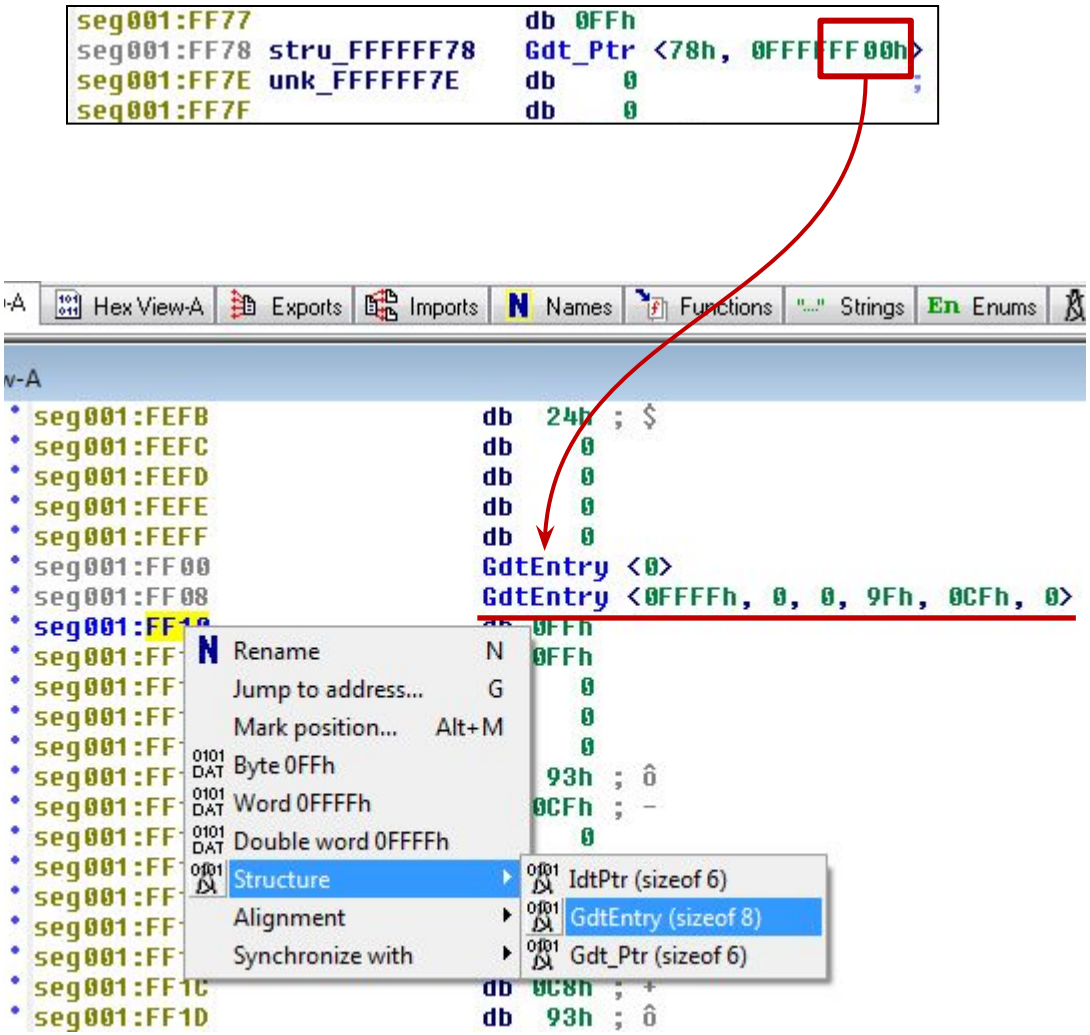
http://www.jamesmolloy.co.uk/tutorial_html/4.-The%20GDT%20and%20IDT.html

5.3: Define GdtPtr



- Go to the address referenced by the operand to the LGDT instruction
- IDA will have already tried to interpret this and failed, undefine that
- Now define it as structure of type GdtPtr
- As per the structure definition, the first member is the size of the GDT table and the second is a pointer to the location of the GDT entries
- That pointer won't translate properly for us, but we can tell where the entries are defined just by looking at the value

5.4: Define GDT Entries



- We know it's location is in our 16-bit segment
- Manually go there by jumping to seg:FF00
- This is where the GDT entries are defined
- Look at the structure definition in peewee.h to interpret
- The table size is 0x78 bytes, but we only want the second entry into the table at offset 8:
- BASE = 0000_0000h
- LIMIT = FFFF_FFFFh
- This is the flat memory model
- These descriptors will be used by the subsequent code so you can fill out the rest as needed

*There may be a superior way to set up our segments so that it all "just works" but I have not found it yet. Also, disregard the different segment names.

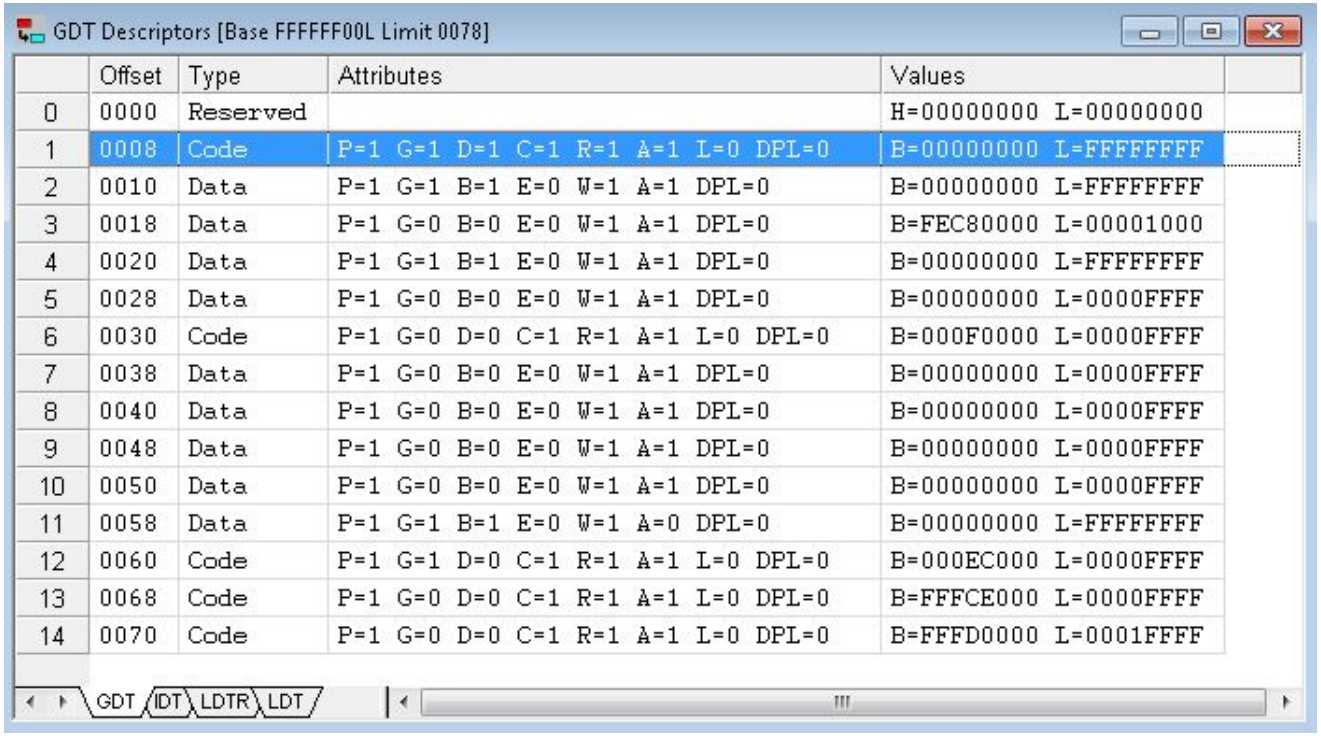
5.5: Full GDT

```
struct GdtEntry {
    uint16 limit_low;           // The lower 16 bits of the limit.
    uint16 base_low;            // The lower 16 bits of the base.
    uint8  base_middle;         // The next 8 bits of the base.
    uint8  access;              // Access flags
    uint8  granularity;
    uint8  base_high;           // The last 8 bits of the base.
} Gdt_Entry;
```

```
GdtEntry <0>
GdtEntry <0FFFFh, 0, 0, 9Fh, 0CFh, 0>
GdtEntry <0FFFFh, 0, 0, 93h, 0CFh, 0>
GdtEntry <1000h, 0, 0C8h, 93h, 0, 0FEh>
GdtEntry <0FFFFh, 0, 0, 93h, 0CFh, 0>
GdtEntry <0FFFFh, 0, 0, 93h, 0, 0>
GdtEntry <0FFFFh, 0, 0Fh, 9Fh, 0, 0>
GdtEntry <0FFFFh, 0, 0, 93h, 0, 0>
GdtEntry <0FFFFh, 0, 0, 93h, 0, 0>
GdtEntry <0FFFFh, 0, 0, 93h, 0, 0>
GdtEntry <0FFFFh, 0, 0, 93h, 0, 0>
GdtEntry <0FFFFh, 0, 0, 92h, 0CFh, 0>
GdtEntry <0Fh, 0C000h, 0Eh, 9Fh, 80h, 0>
GdtEntry <0FFFFh, 0E000h, 0FCh, 9Fh, 0, 0FFh>
GdtEntry <0FFFFh, 0, 0FDh, 9Fh, 1, 0FFh>
```

- The GdtEntry structure definition in peewee.h can be used to interpret the GDT entries
- Each structure is 8 bytes in size
- The FAR JMP is referencing the second entry (offset 8)
- Base 0, Limit FFFF_FFFFh

5.5: Full GDT



GDT Descriptors [Base FFFFFF00L Limit 0078]

	Offset	Type	Attributes	Values
0	0000	Reserved		H=00000000 L=00000000
1	0008	Code	P=1 G=1 D=1 C=1 R=1 A=1 L=0 DPL=0	B=00000000 L=FFFFFFFF
2	0010	Data	P=1 G=1 B=1 E=0 W=1 A=1 DPL=0	B=00000000 L=FFFFFFFF
3	0018	Data	P=1 G=0 B=0 E=0 W=1 A=1 DPL=0	B=FEC80000 L=00001000
4	0020	Data	P=1 G=1 B=1 E=0 W=1 A=1 DPL=0	B=00000000 L=FFFFFFFF
5	0028	Data	P=1 G=0 B=0 E=0 W=1 A=1 DPL=0	B=00000000 L=0000FFFF
6	0030	Code	P=1 G=0 D=0 C=1 R=1 A=1 L=0 DPL=0	B=000F0000 L=0000FFFF
7	0038	Data	P=1 G=0 B=0 E=0 W=1 A=1 DPL=0	B=00000000 L=0000FFFF
8	0040	Data	P=1 G=0 B=0 E=0 W=1 A=1 DPL=0	B=00000000 L=0000FFFF
9	0048	Data	P=1 G=0 B=0 E=0 W=1 A=1 DPL=0	B=00000000 L=0000FFFF
10	0050	Data	P=1 G=0 B=0 E=0 W=1 A=1 DPL=0	B=00000000 L=0000FFFF
11	0058	Data	P=1 G=1 B=1 E=0 W=1 A=0 DPL=0	B=00000000 L=FFFFFFFF
12	0060	Code	P=1 G=1 D=0 C=1 R=1 A=1 L=0 DPL=0	B=000EC000 L=0000FFFF
13	0068	Code	P=1 G=0 D=0 C=1 R=1 A=1 L=0 DPL=0	B=FFFCE000 L=0000FFFF
14	0070	Code	P=1 G=0 D=0 C=1 R=1 A=1 L=0 DPL=0	B=FFFD0000 L=0001FFFF

GDT IDT LDTR LDT

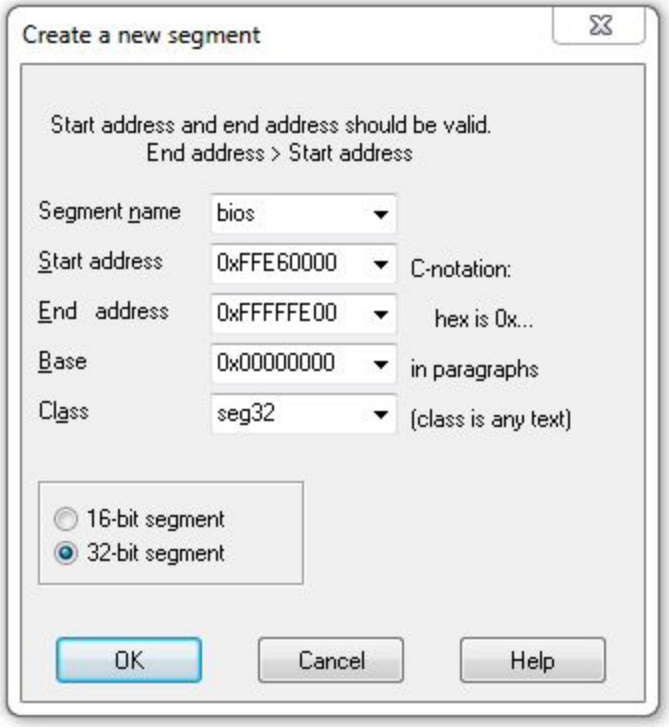
- Here is the entire GDT for reference. You don’t need an expensive debugger to analyze BIOS (but it does save a lot of time)

6: Create the 32-bit BIOS segment

Copernicus_Log.txt

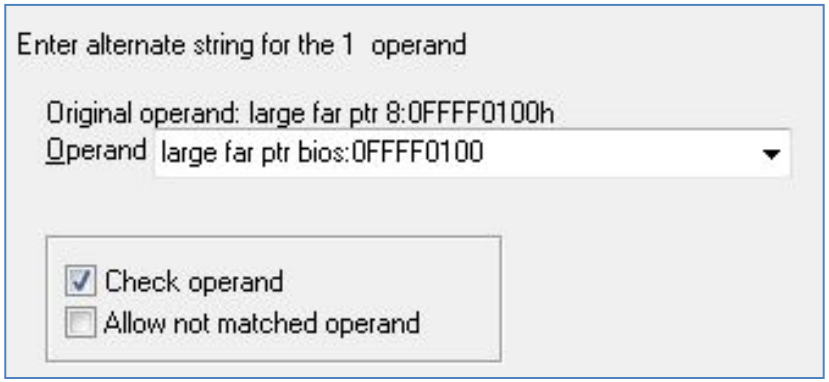
```
Determining size of SPI flash chip
  SPI Region 0 (Flash Descriptor) base = 00000000, limit = 00000fff
  SPI Region 1 (BIOS) base = 00260000, limit = 003fffff
  SPI Region 2 (Management Engine) base = 0000b000, limit = 0025ffff
  SPI Region 3 (Gigabit Ethernet) base = 00001000, limit = 00002fff
  SPI Region 4 (Platform Data) base = 00003000, limit = 0000afff
  SPI Flash chip size = 0x00400000
```

- Now create the 32-bit segment
- Start address is FFFF_FFFFh - <size of the BIOS region> + 1
  - FFFF_FFFFh – 1A_0000h in this example
  - SPI regions will be explained more during BIOS flash portion of the course
- End Address is our segment boundary Address
  - FFFF_FE00h in this example
- Base Address matches that of the GDT table, entry 8 (0000_0000h)



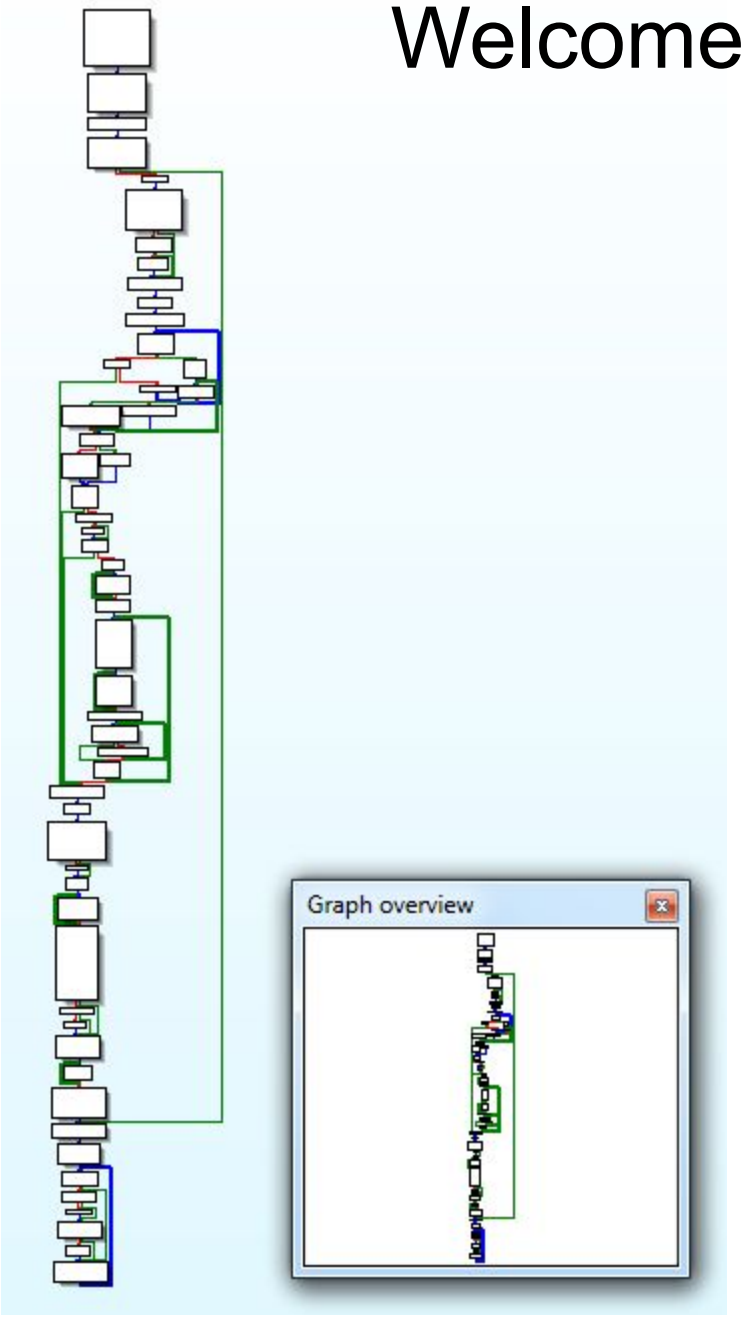
7: Touch up the Far Jump

```
seg001:FE30 ; Segment type: Regular
seg001:FE30 seg001          segment byte public '16bit' use16
seg001:FE30                 assume cs:seg001
seg001:FE30                 ;org 0FE30h
seg001:FE30                 assume es:nothing, ss:nothing, ds:nothi
seg001:FE30
seg001:FE30 loc_FFFFFE30:                           ; CODE XREF: se
seg001:FE30                 db      66h
seg001:FE30                 lgdt    fword ptr cs:byte_FFFFFF78
seg001:FE37                 db      66h
seg001:FE37                 lidt    fword ptr cs:byte_FFFFFF7E
seg001:FE3E                 mov     eax, cr0
seg001:FE41                 or      al, 1
seg001:FE43                 mov     cr0, eax
seg001:FE46                 jmp     short $+2
seg001:FE48                 mov     ax, 10h
seg001:FE4B                 mov     ds, ax
seg001:FE4D                 assume ds:nothing
seg001:FE4D                 mov     es, ax
seg001:FE4F                 assume es:nothing
seg001:FE4F                 mov     fs, ax
seg001:FE51                 assume fs:nothing
seg001:FE51                 jmp     large far ptr 8:0FFFF0100h
seg001:FE51 ; ---------------------------------------------------
```



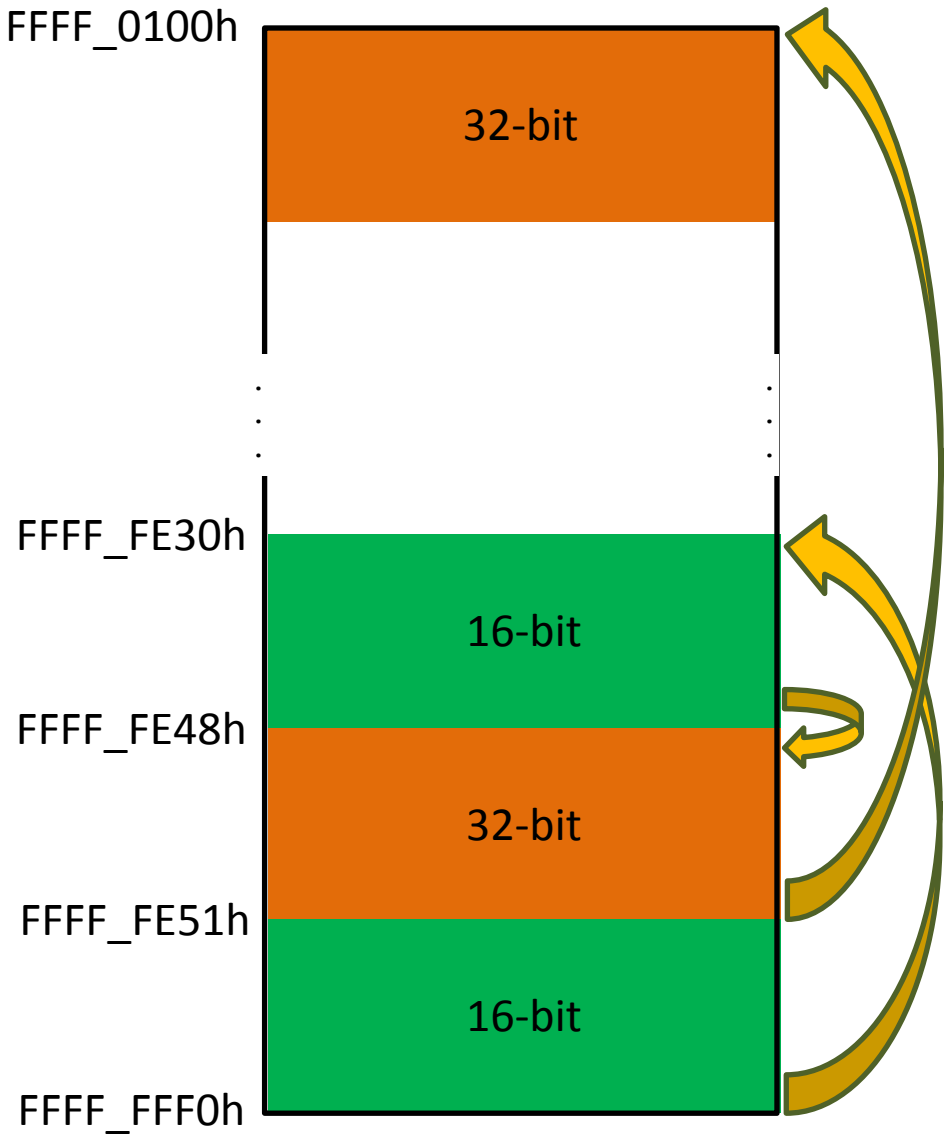
- So we know that this is loading the descriptor entry at offset 8 in the GDT
- We can visually inspect the operand of this JMP to see that it's going to FFFF_0100h
- We can manually fix this operand
- Right click the operand and select 'Manual'
- Change it to:
- bios:FFFF0100h
- Uncheck 'Check Operand'
- A little ugly

Welcome to BIOS Analysis



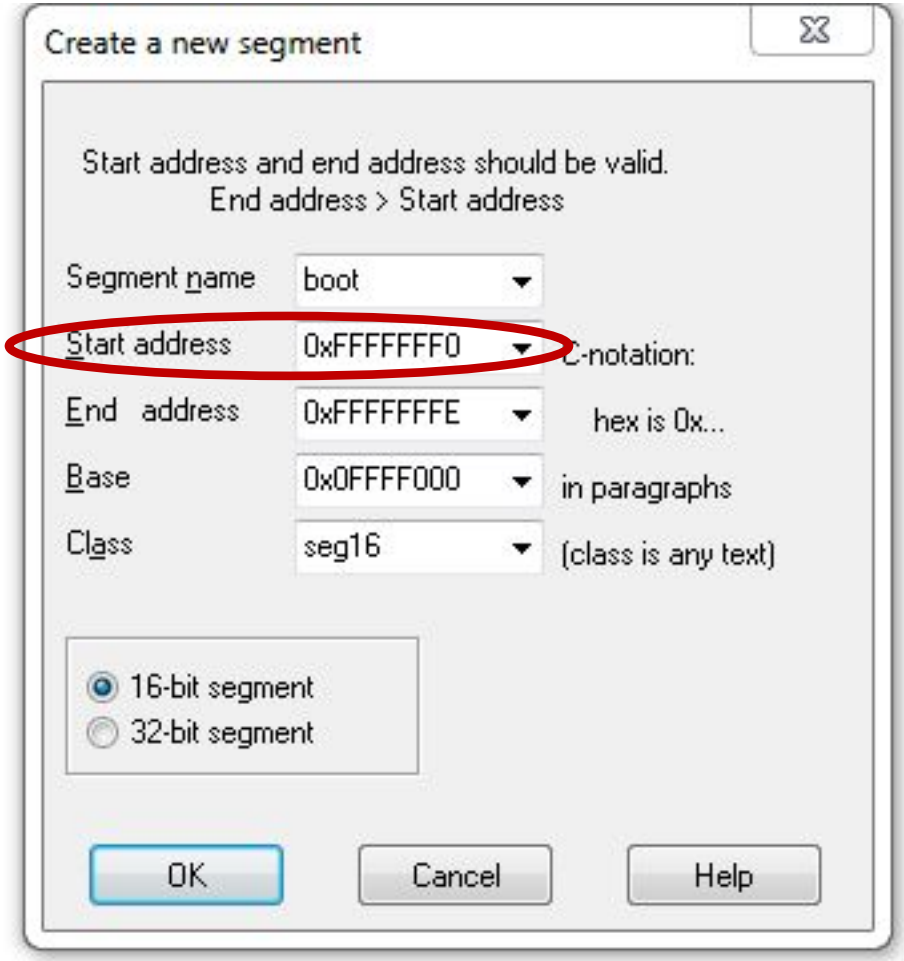
- Converting the binary at FFFF_0100h to code provides you the entry point to the real BIOS initialization
- Up until this point everything we covered is pretty standard across many BIOSes
  - This applies to UEFI BIOS too
  - Even really old BIOS will basically follow the path we took, perhaps staying in real mode longer though
- From here on though, if legacy, it's completely proprietary to the OEM (data structures, etc.)
- By contrast, UEFI is standardized from head to toe

Why so Ugly? IDA Segments



- IDA can't combine 16-bit and 32-bit instructions in the same segment
- We could have created another 32-bit segment to account for the processor entering 32-bit protected mode
- But then we'd have to create 4 segments
- Not really necessary since we can visually inspect it and determine what's going on
- Fudging it is okay since the important stuff happens after all this

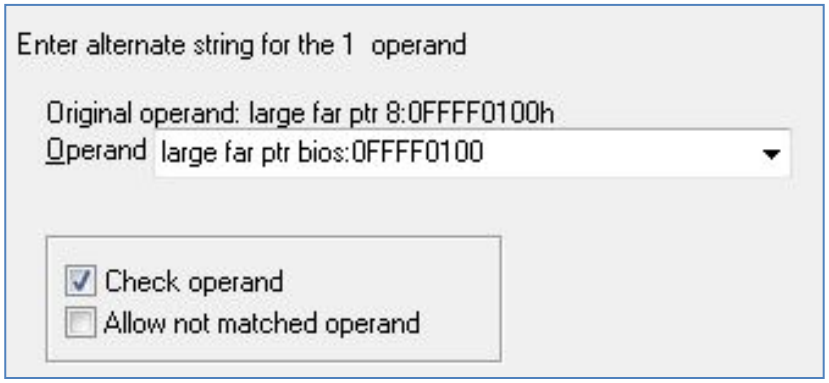
BIOS Reset Vector Analysis: Short Cut 1



- You can likely skip a few of the steps and make some assumptions to get to the initialization code faster:
- Open your BIOS binary file in IDA same as before
- Rebase the program, same as before
- Don't bother analyzing the entry vector JMP, just create a 16-bit segment the exact same as before, except:
  - Start Address: 0xFFFFFFF0
  - We can count on IDA being smart enough to interpret this properly even though it makes our segment a little odd

BIOS Reset Vector Analysis: Short Cut 2

```
seg001:FE30 ; Segment type: Regular
seg001:FE30 seg001          segment byte public '16bit' use16
seg001:FE30                 assume cs:seg001
seg001:FE30                 ;org 0FE30h
seg001:FE30                 assume es:nothing, ss:nothing, ds:nothi
seg001:FE30
seg001:FE30 loc_FFFFFE30:                           ; CODE XREF: se
seg001:FE30                 db      66h
seg001:FE30                 lgdt    fword ptr cs:byte_FFFFFF78
seg001:FE37                 db      66h
seg001:FE37                 lidt    fword ptr cs:byte_FFFFFF7E
seg001:FE3E                 mov     eax, cr0
seg001:FE41                 or      al, 1
seg001:FE43                 mov     cr0, eax
seg001:FE46                 jmp     short $+2
seg001:FE48                 mov     ax, 10h
seg001:FE4B                 mov     ds, ax
seg001:FE4D                 assume ds:nothing
seg001:FE4D                 mov     es, ax
seg001:FE4F                 assume es:nothing
seg001:FE4F                 mov     fs, ax
seg001:FE51                 assume fs:nothing
seg001:FE51                 jmp     large far ptr 8:0FFFF0100h
seg001:FE51 ; ---------------------------------------------------
```



- Follow the entry JMP
  - Notice that IDA automagically modified our segment so it begins at seg:FE30
- Manually touch up the FAR JMP same as before
- We could optionally create a 32-bit segment here just to ensure it has a base of 0h
  - Assume a flat memory model
- Now we can go to the real BIOS initialization code entry, just like before!
- This shortcut doesn't always work

Lab: Scratch the surface

- Repeat the process we just did for the E6400 BIOS on each of your BIOS dumps
- We'll see if there are any where it leads to early confusion