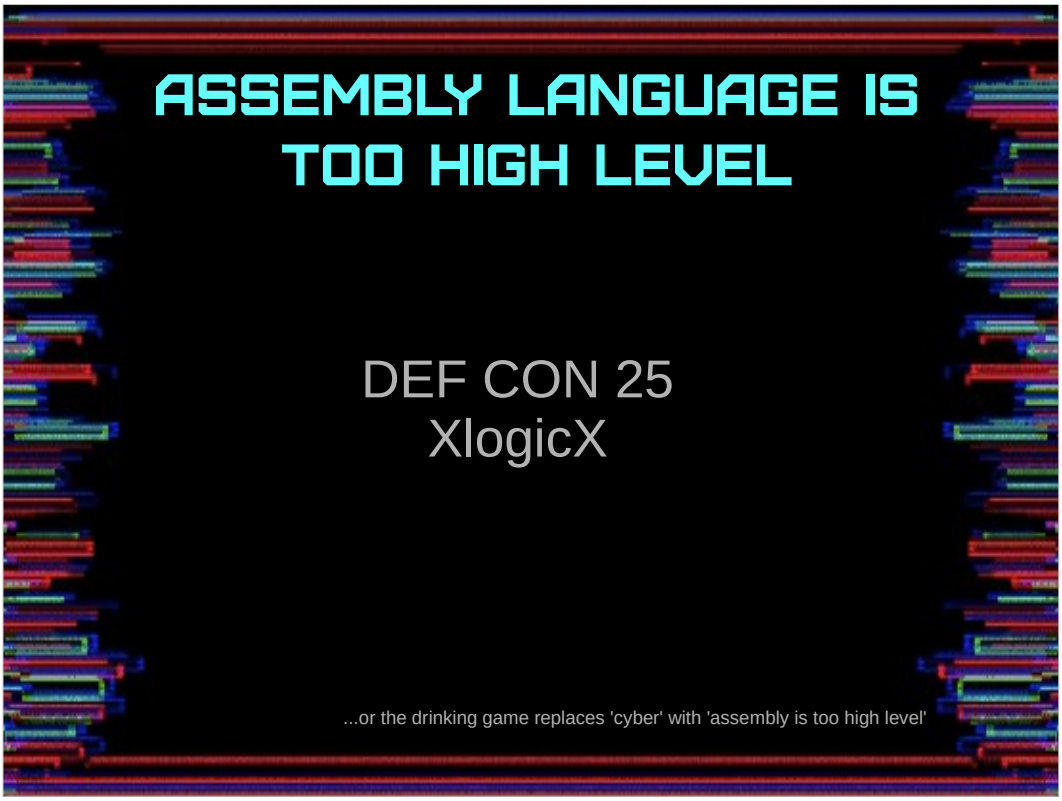
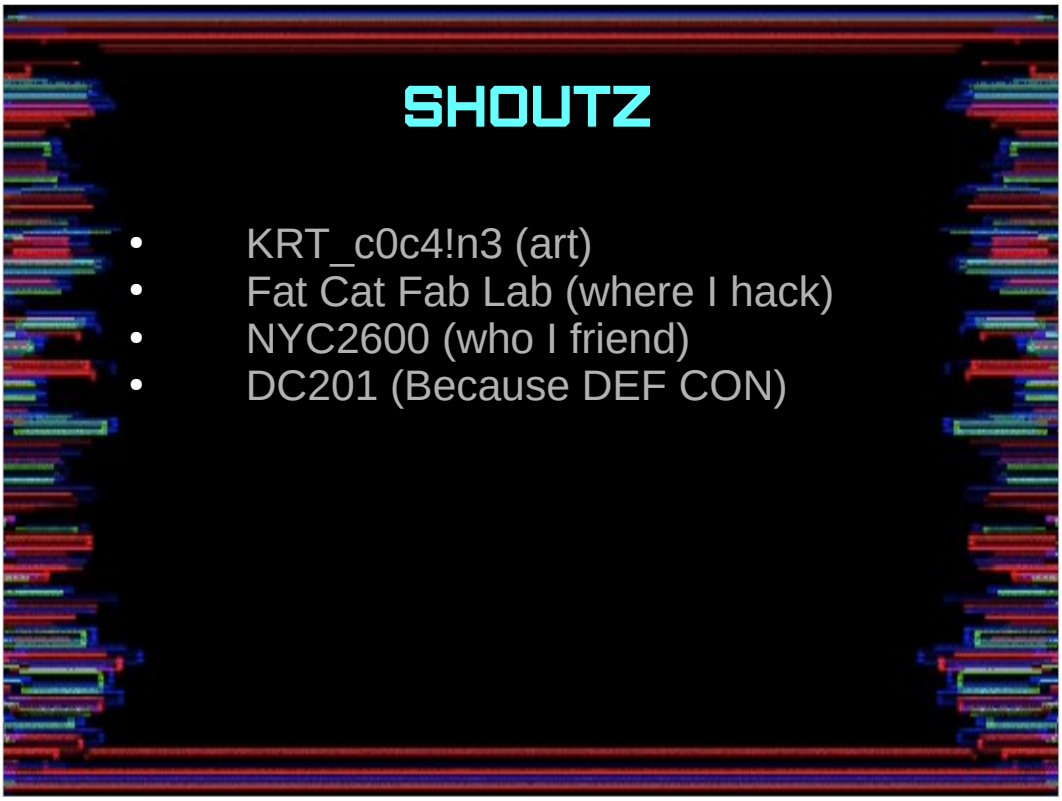
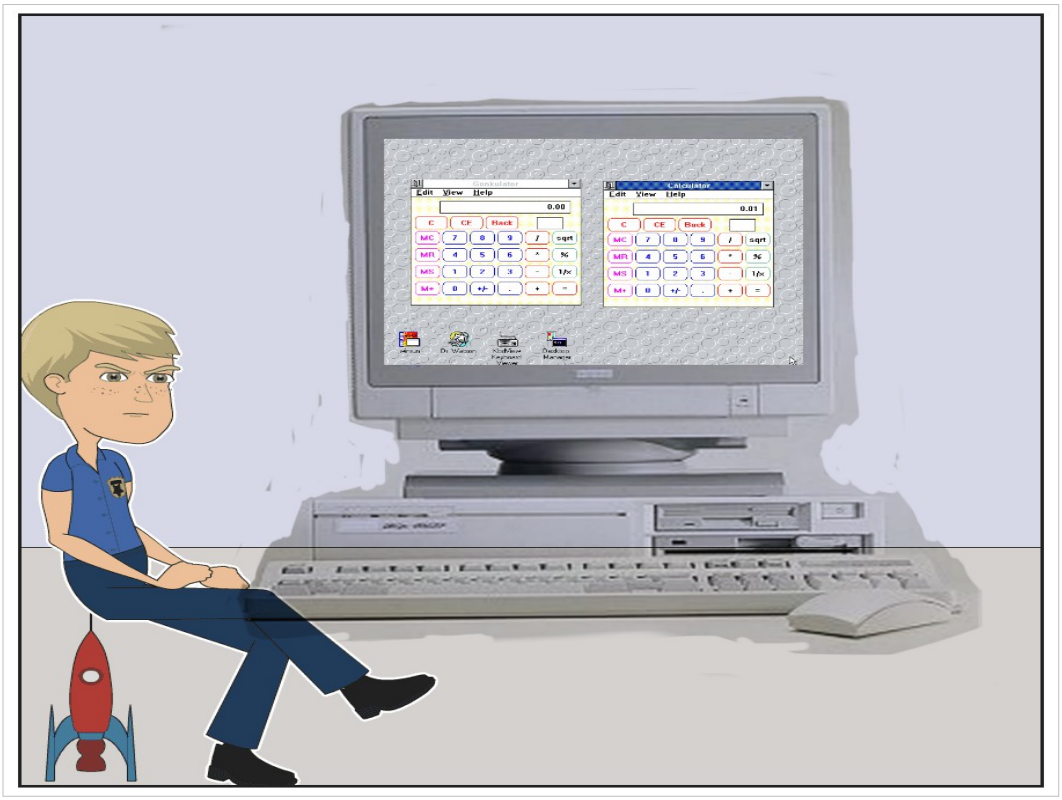
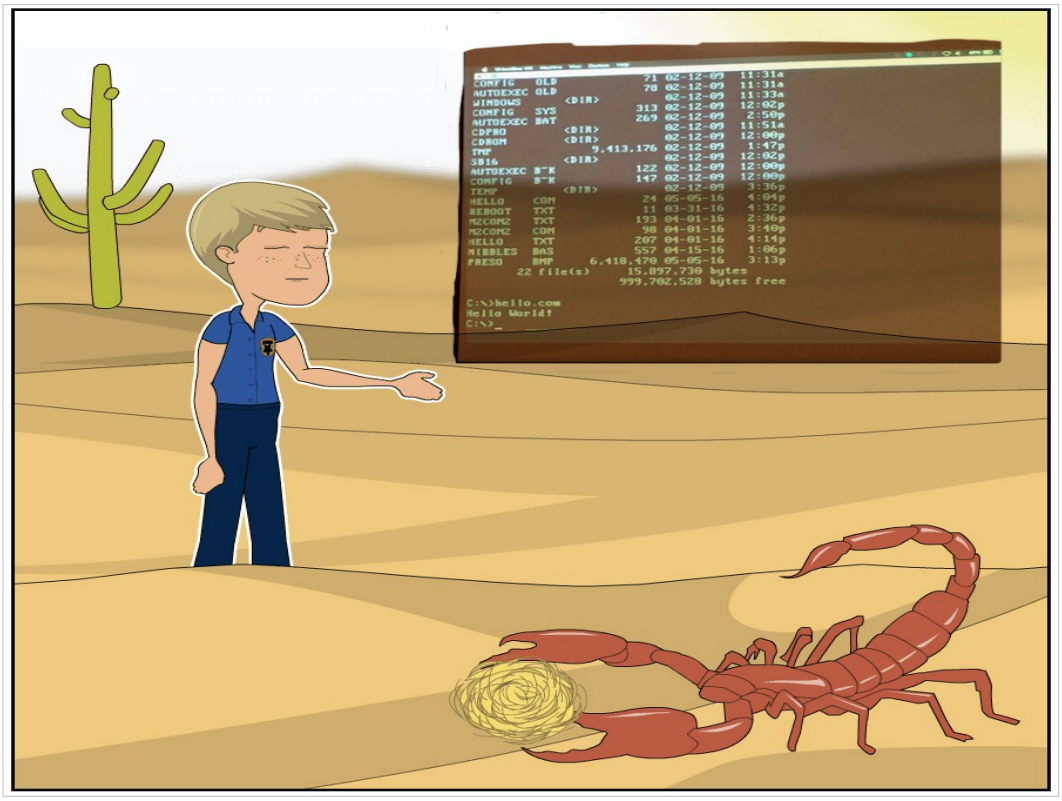

ASSEMBLY LANGUAGE IS TOO HIGH LEVEL
DEF CON 25
XlogicX
...or the drinking game replaces 'cyber' with 'assembly is too high level'



- My girlfriend KRT_c0c4!n3 (art director) did a good portion of the art of these slides
- I worked on most of my code and all of these slides from Fat Cat Fab Lab. It's my favorite hackerspace in the NYC area (West Village)
- NYC2600 is my local 2600 community and where I've made most of the friends I have in NYC
-DC201, because it's the closest active DEF CON group in my area

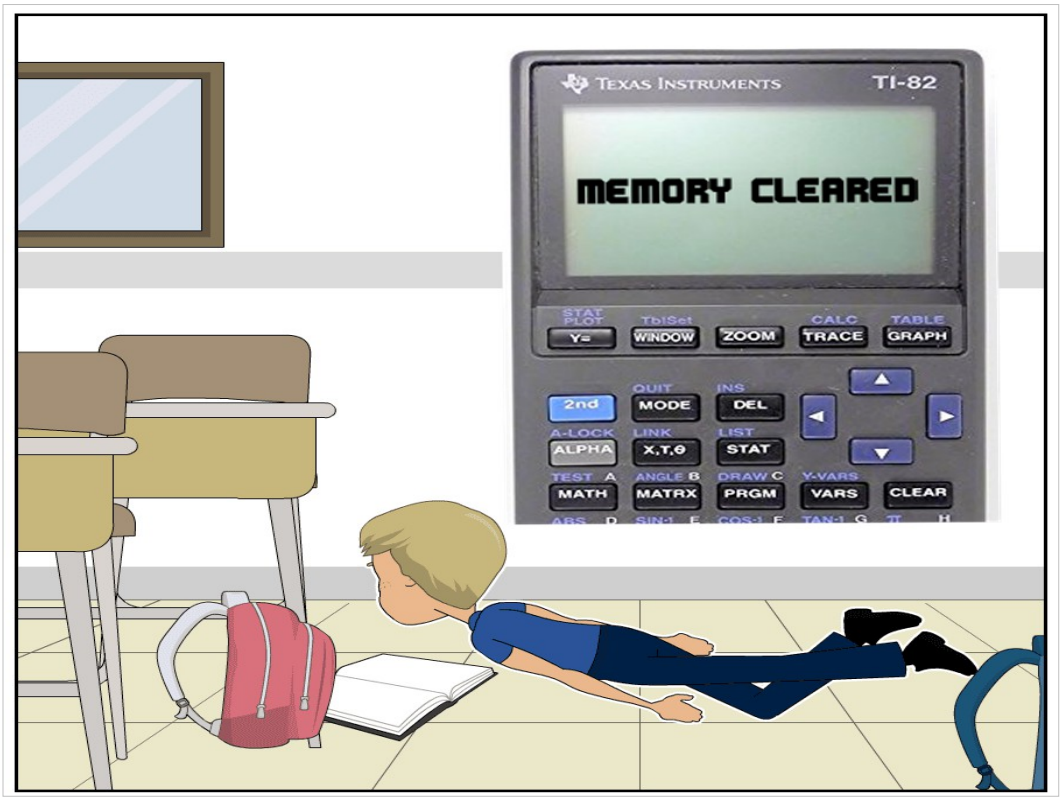


Even as a kid I wanted to do low level programming. I had no access or knowledge of compilers or even major programming languages. I deep down felt like I should be able to type the right binary data into a notepad (or something like it) and run it, but all I had was just some Windows 3.11 and ignorance

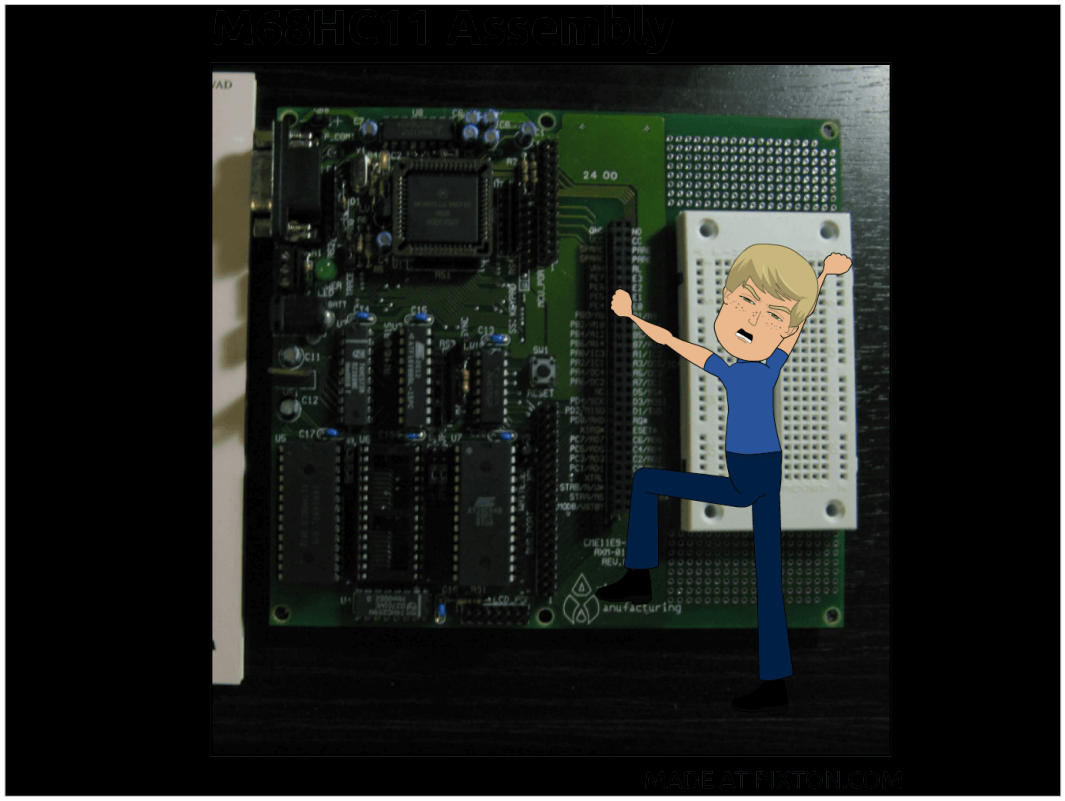


I eventually did end up typing hex into debug (comes with Windows 3.11+) and executed my program live at CactusCon 2016

Deck at http://xlogicx.net/?p=515

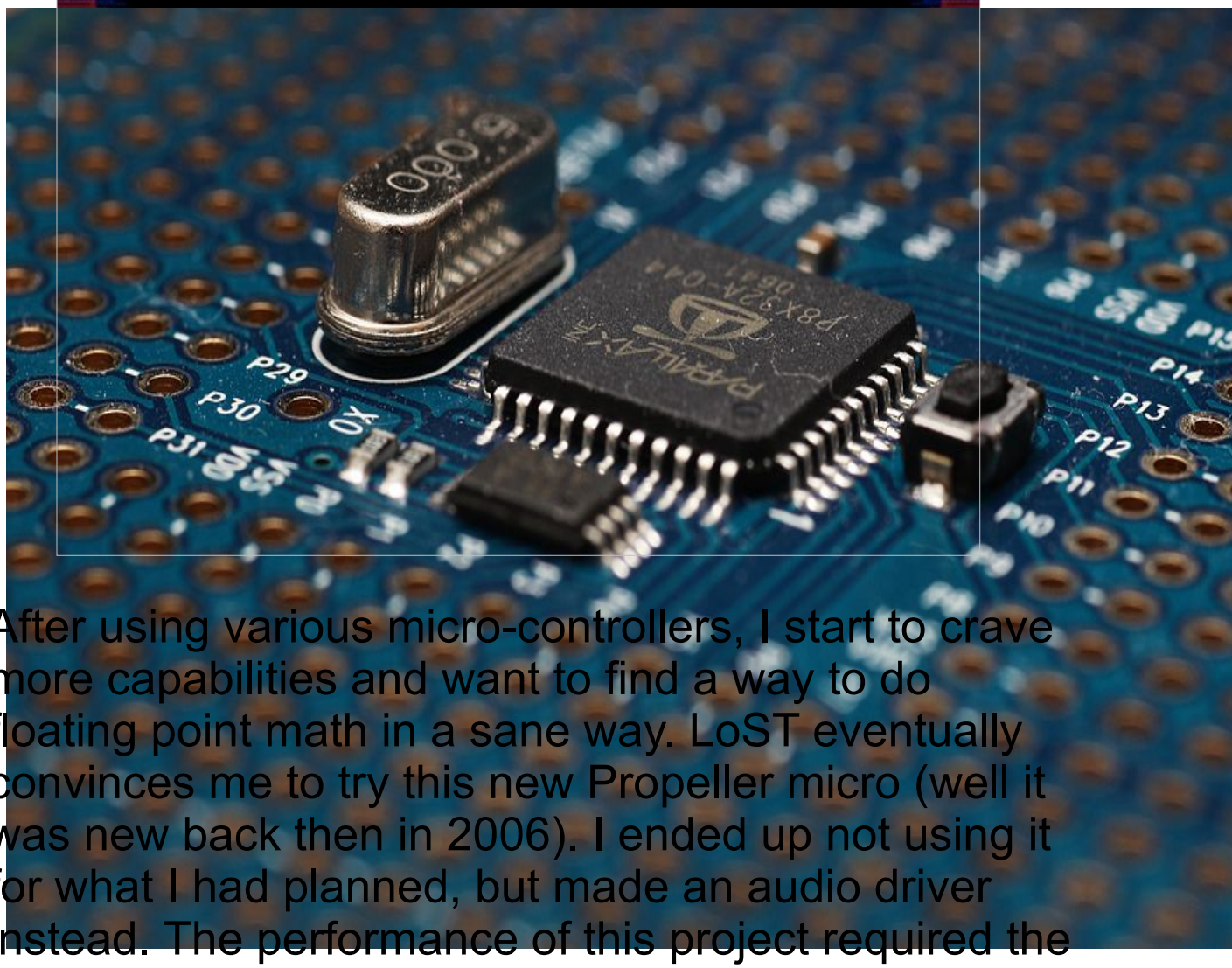


I eventually try to teach myself Z80 assembly. This is because I already had a TI-82 and already tried some sweet games programmed in assembly.
The first program I made was an example program that clears the screen. My first attempt to make my own program cleared the memory. This was unintended...



I then formally learn Assembly for the M68HC11 microcontroller in school. I don't even remember if we had a textbook, but we did have the Motorola manual. This manual listed all of the instructions with the machine code next to the instruction.
I had a lot of fun with this architecture. Inspired by Godel Escher Bach, I attempted to create a program that replicated itself into the next area of memory and executed itself. I learned the importance of needing to understand the abstraction layer of machine code in order to pull this off. Also, the assembly language and machine code for this architecture was relatively one to one.

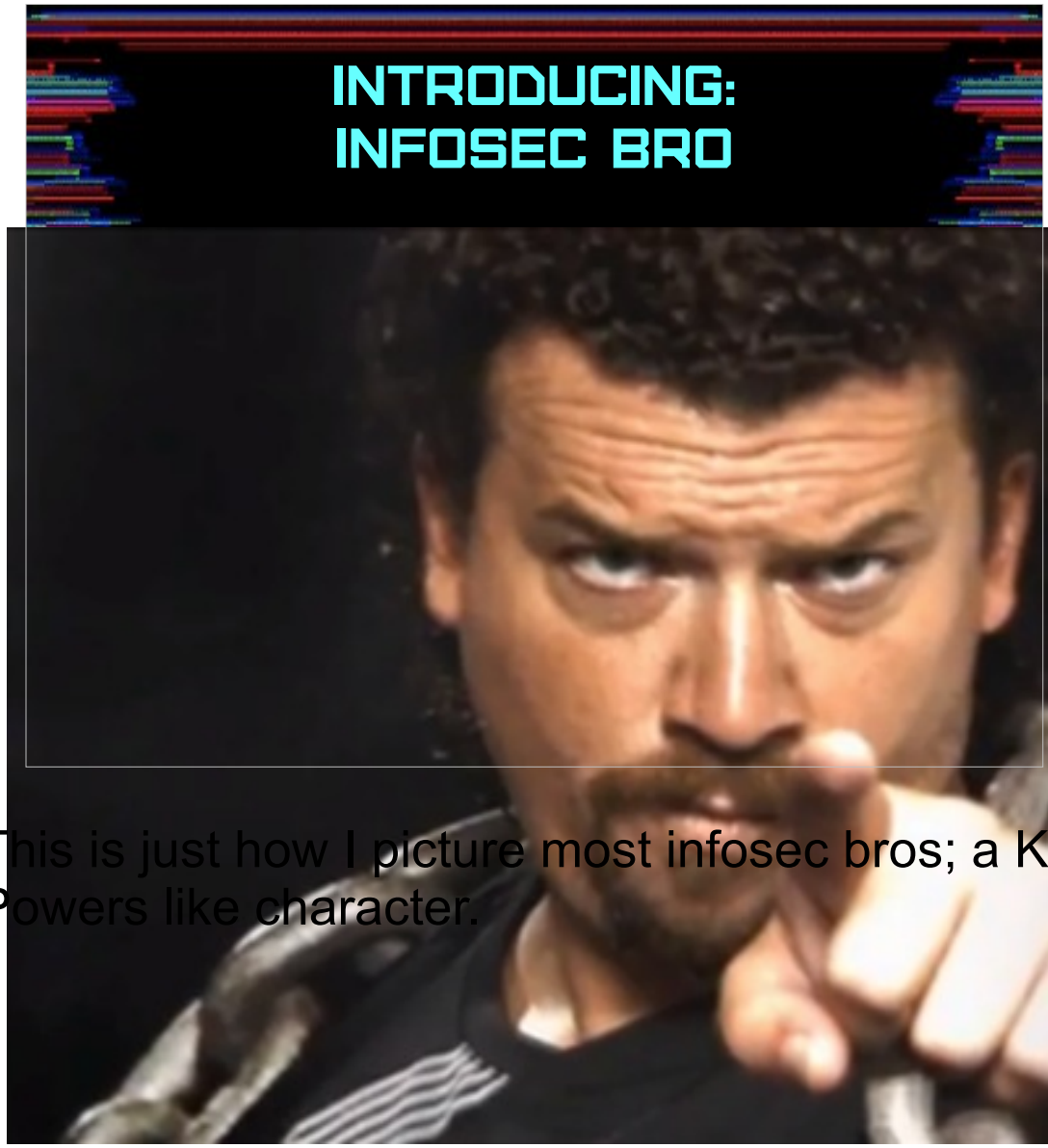
PROPELLER ASSEMBLY



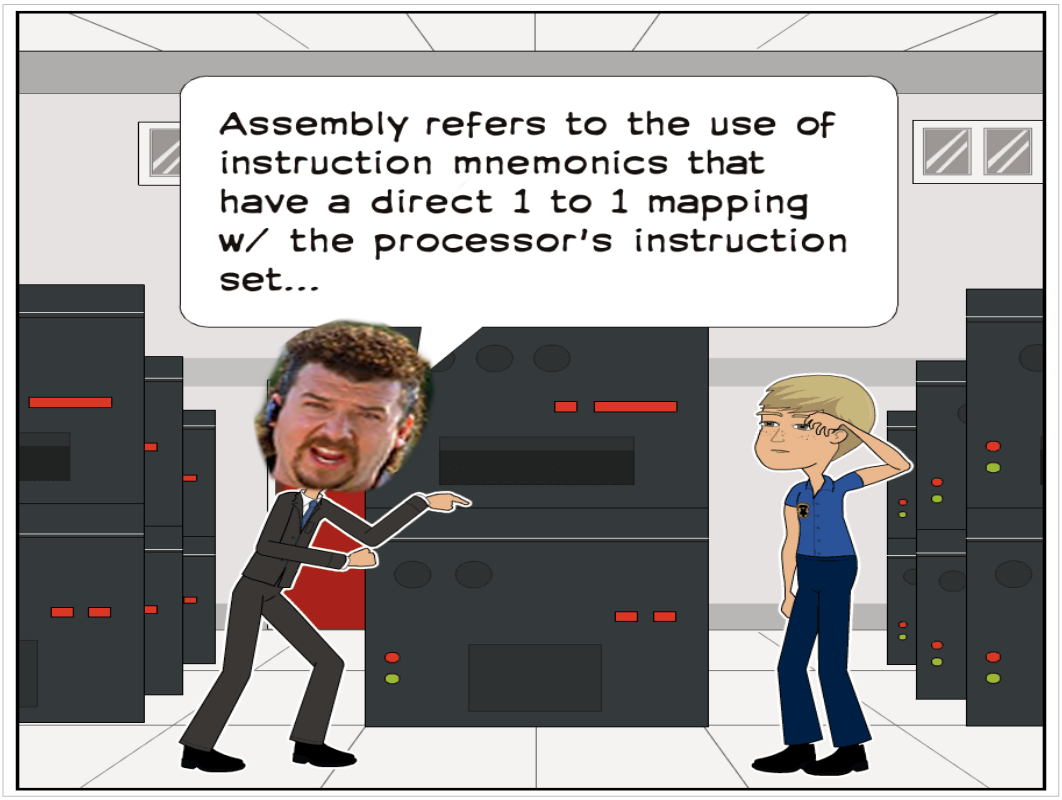
After using various micro-controllers, I start to crave more capabilities and want to find a way to do floating point math in a sane way. LoST eventually convinces me to try this new Propeller micro (well it was new back then in 2006). I ended up not using it for what I had planned, but made an audio driver instead. The performance of this project required the use of Propeller Assembly (instead of the recommended high level language: SPIN). This architecture was pure beauty, and the relationship between the machine-code and assembly language was practically one to one for all intents and purposes. I'm still waiting on Chip Gracey to finish Duke Nukem Forever (...I mean the Propeller II)

Then, a matter of years ago, the company I was working with before voluntold me to take GREM training (GIAC Reverse Engineering Malware). This is the context in which I eventually learned the x86 architecture for assembly language. I learned that the language was the most terrible assembly language I've ever seen up to this point; which made it all that more beautiful.

And those manuals in the screenshot, I've read them all, cover to cover.

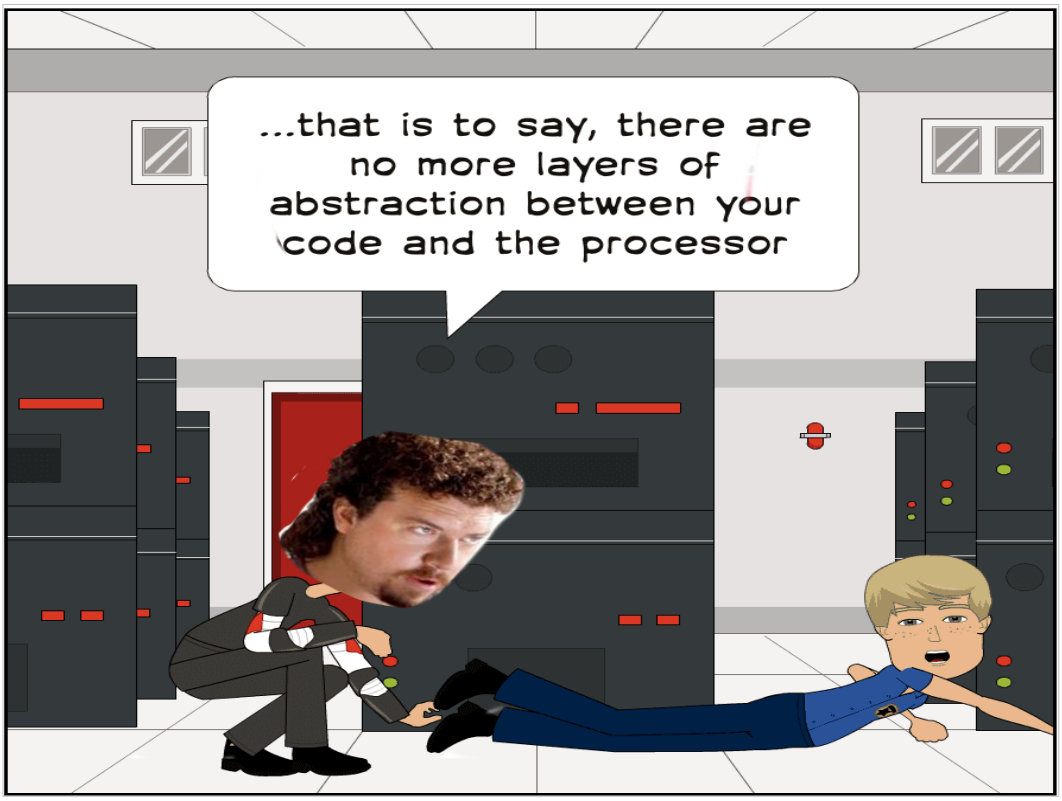


This is just how I picture most infosec bros; a Kenny Powers like character.

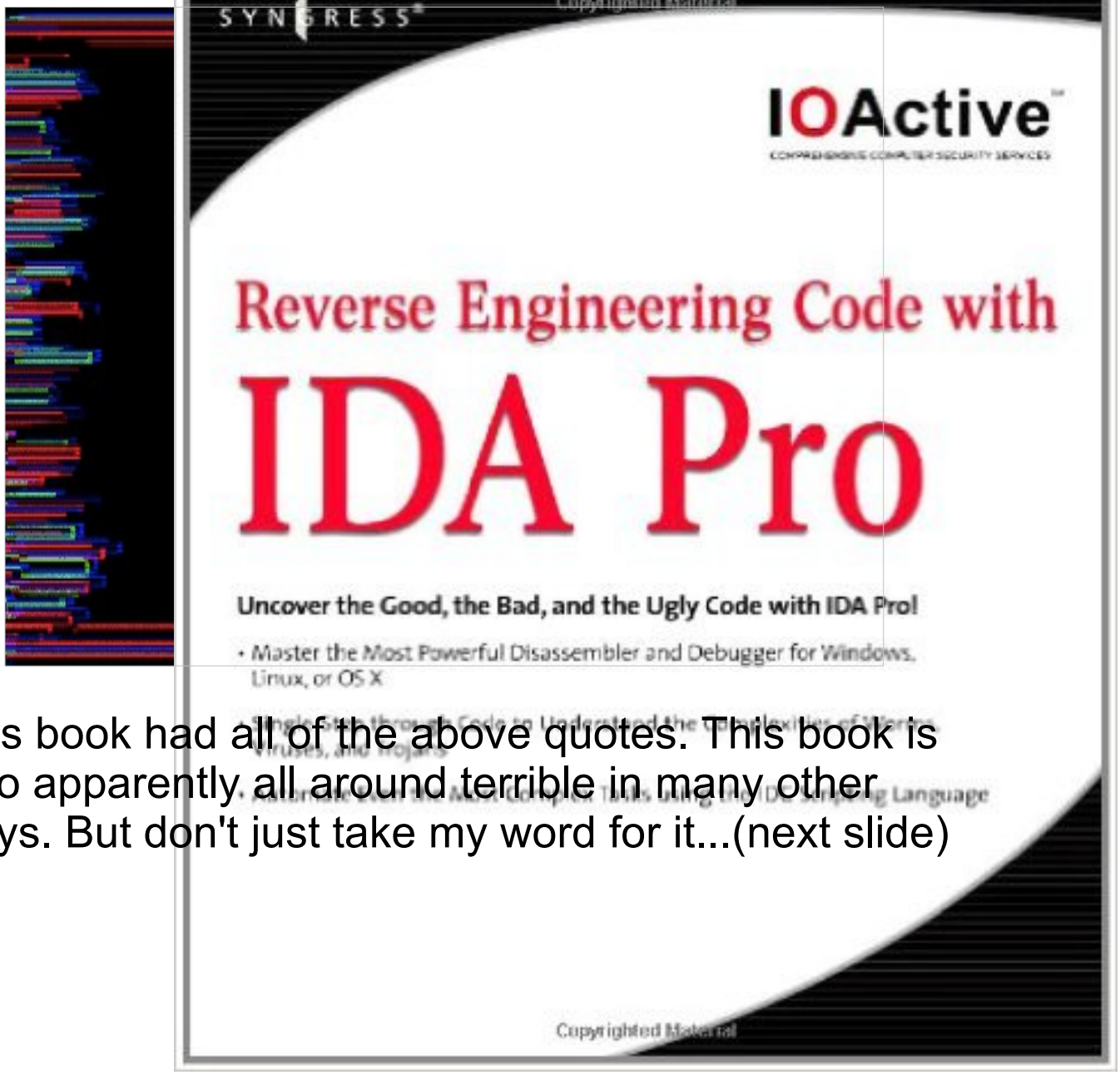


- "Assembly refers to the use of instruction mnemonics that have a direct one-to-one mapping with the processors instruction set"

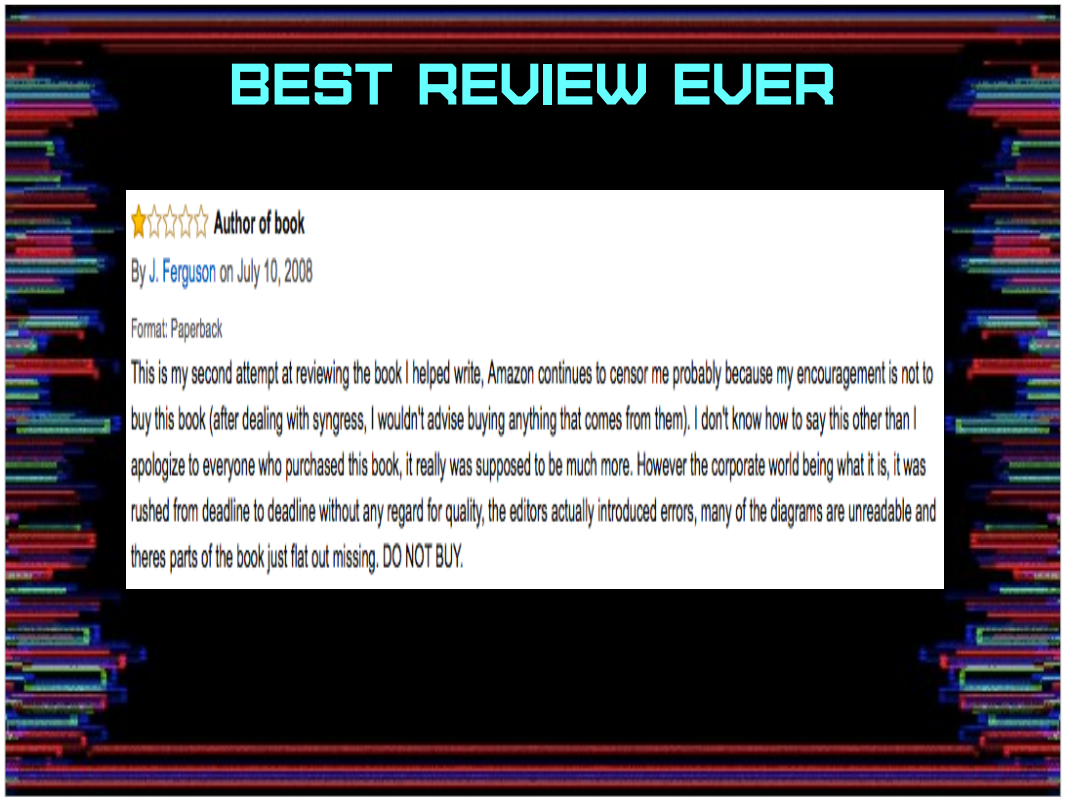
- "However, everything in the end is assembly, and that is just fixed sequences of ones and zeros being sent to the processor"



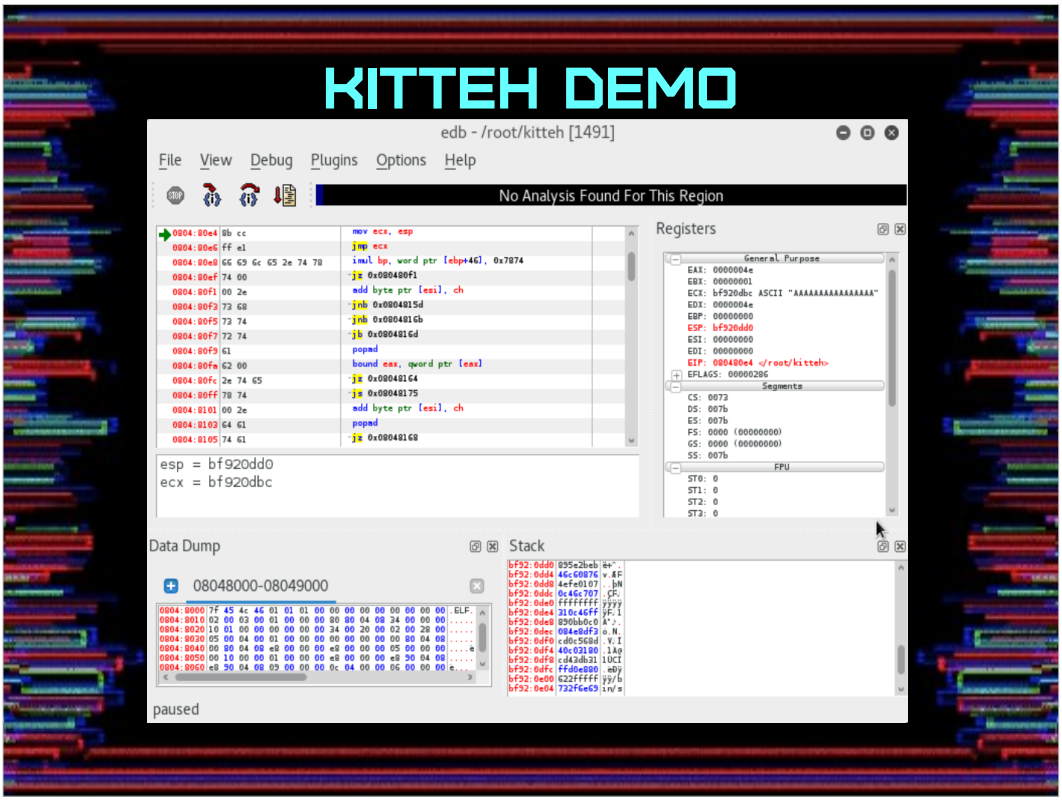
- "...that is to say, there are no more layers of abstraction between your code and the processor"



This book had all of the above quotes. This book is also apparently all around terrible in many other ways. But don't just take my word for it...(next slide)

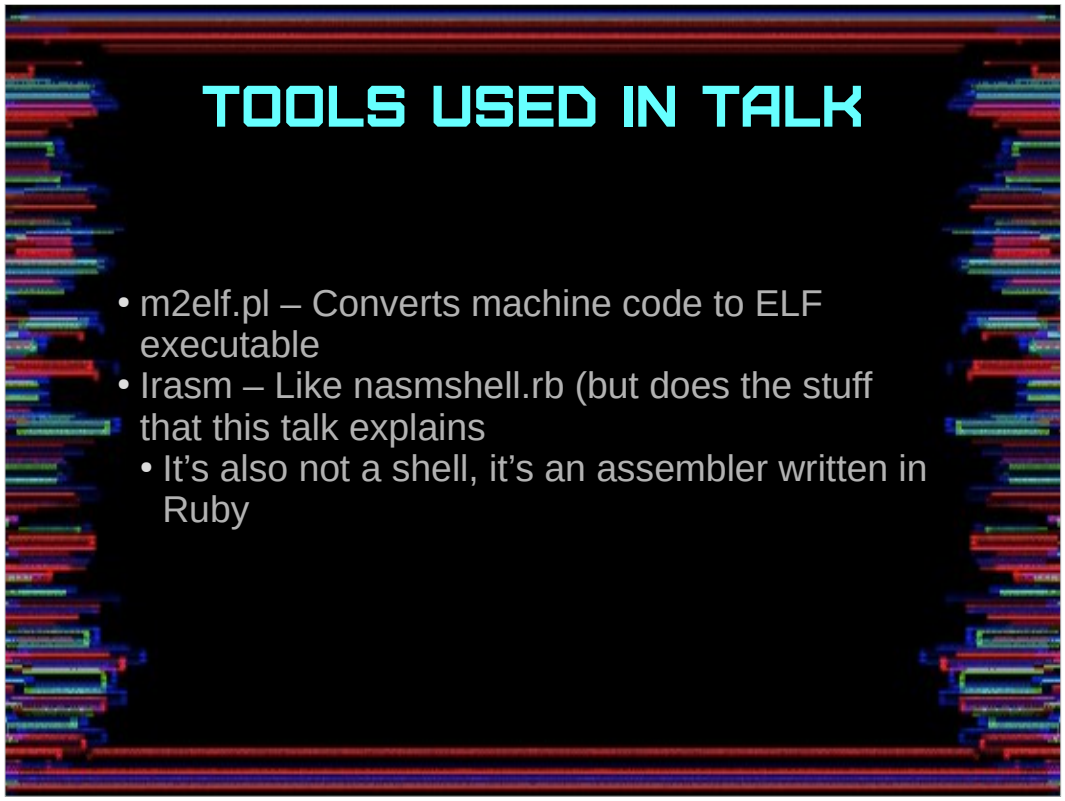


This review was from one of the authors of this book!



Running the demo kitteh program to show what it does
Quickly running through the source to show the vulnerabilities
Exploiting the program to get a 'shell'
Showing the important line of assembly being exploited, and how the actual machine code cannot be produced by nasm_shell

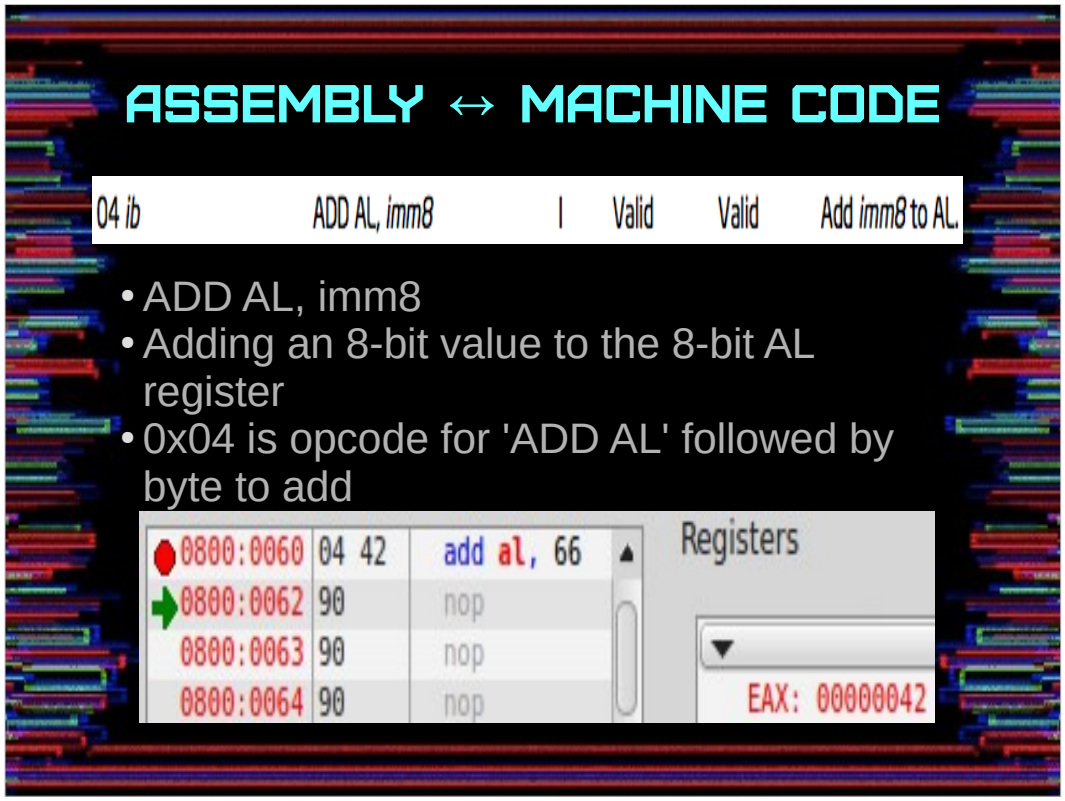
The screenshot in this slide is for the PDF version, it is only a hint at what will be demonstrated



I will likely be flying in and out of these tools during this talk. Not as legitimate full demos, just a few seconds here and there to illustrate the points.

M2elf is a tool that I created that takes hex or binary (1's and 0's) in an input file and converts it into a fully ELF executable. For the purposes of this presentation, I will be running it in 'interactive' mode; it takes machine code input and immediately displays the instruction it represents (instruction by instruction)

Irasm is like nasmshell.rb, only irasm is not a shell, it's an assembler. Instead of just displaying official machine code, it outputs a bunch of redundant machine code as well (as discussed in this talk)

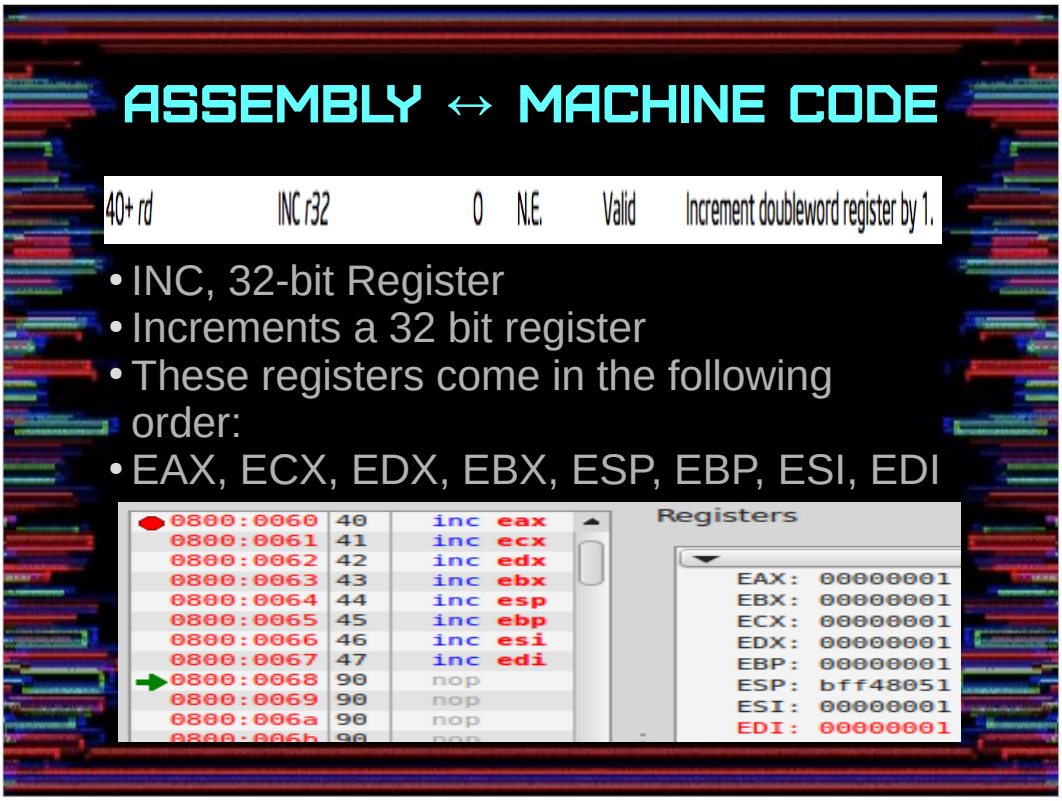


Let's talk about what people are thinking about when they erroneously say that assembly language and machine code have a one to one relationship.

We can say that if we add the byte of 0x42 to the AL register (ADD AL,0x42). The machine code will be 0x0442 (0x04 for ADD and 0x42 is the byte).

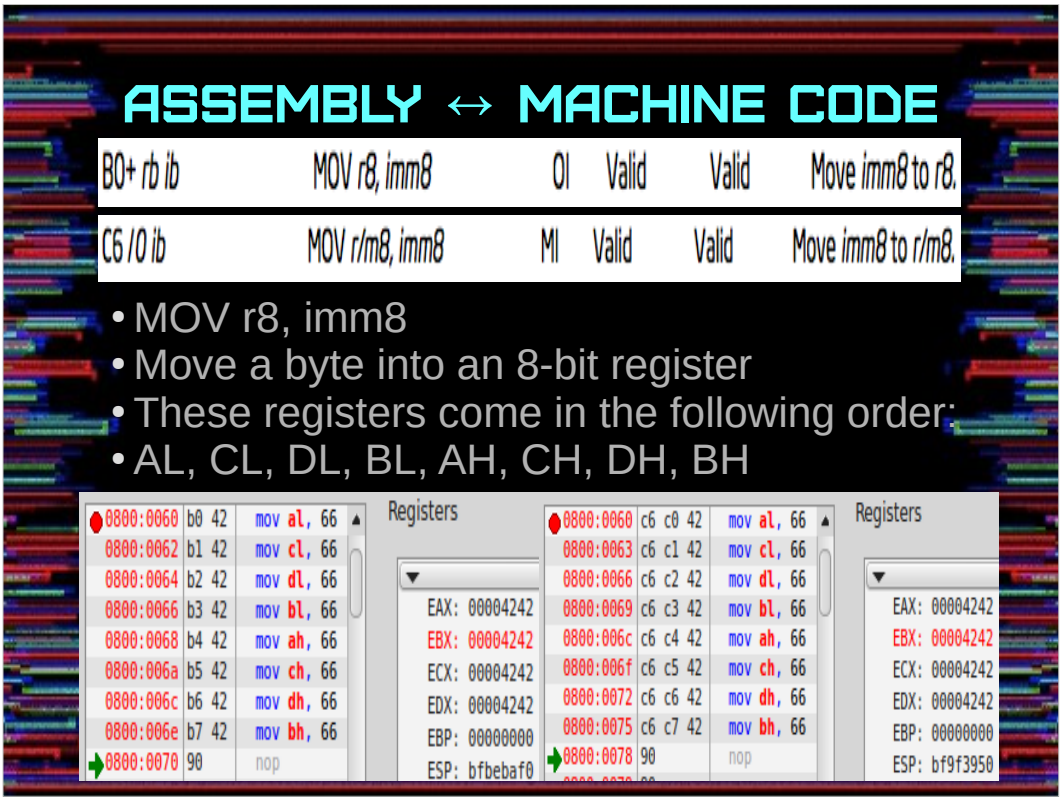
This means that if we wanted to add 0x33 to the AL register, the machine code would be 0x0433

You see the correlation right?



This one is a little more complicated but not that bad. All of this increment (INC) instructions start with a 0x4 nibble, and the next nibble corresponds to the register you want to increment. Since EAX is first, INC EAX is just 0x40.

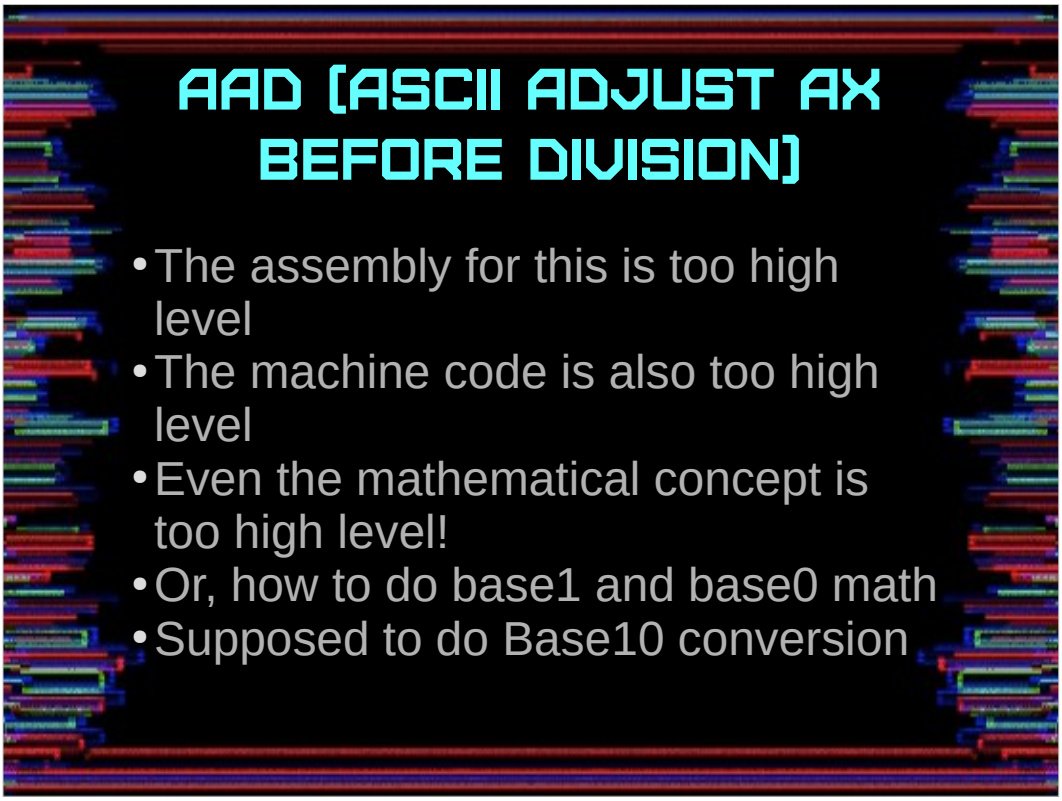
This is unless we are using a 64 bit processor, then the 0x40 is a prefix byte, different story all together though.



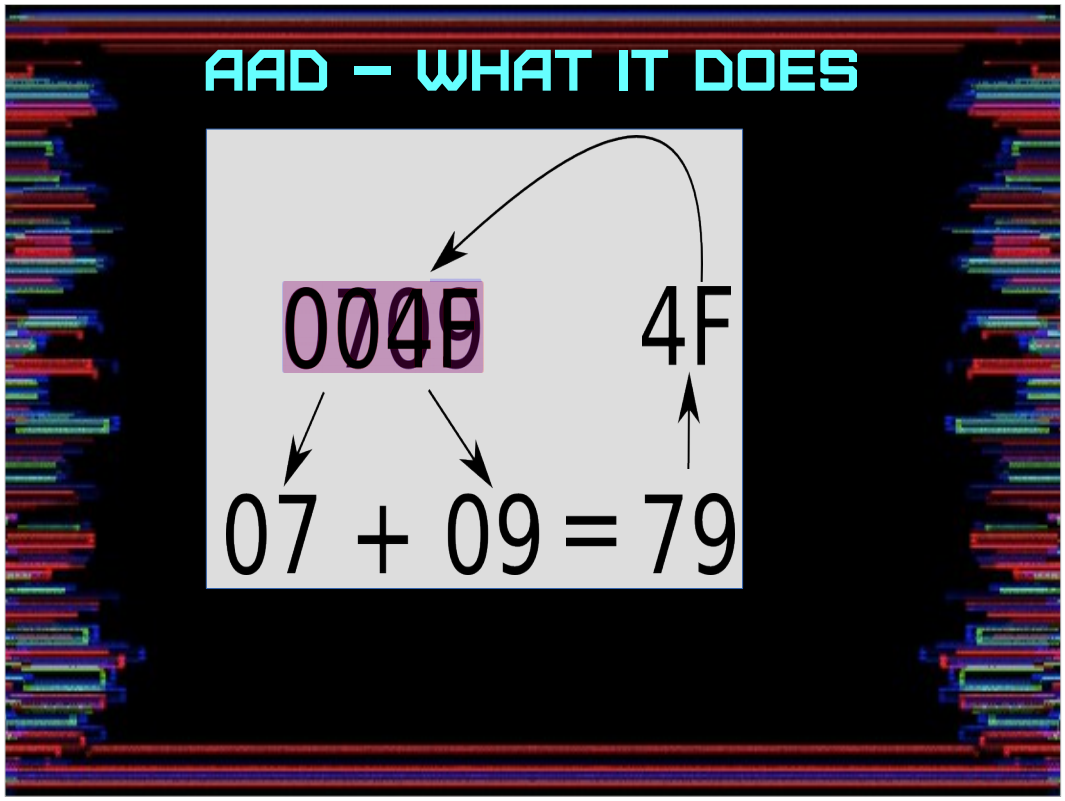
Similar to the last two instructions. This is a group of MOV instructions where 0xB is the first nibble representing MOV, and the next nibble represents the register. Finally, the byte that follows is the byte to be moved to said register.

But wait, there's a 0xC6 format that allows us to add a byte to a more complex data structure that includes memory pointers AND also registers (and because this structure supports registers, we find a redundancy here)

Knowing all of this, if you did: mov al, 0x44
Your assembler (and nasmshell) would output: 0xB042
It wouldn't output 0xC6C042
But the irasm tool will



I love the AAD instruction. It says it does a thing. But the thing it actually does to do the thing it says it does is far more interesting. The next several slides go into depth of these things.



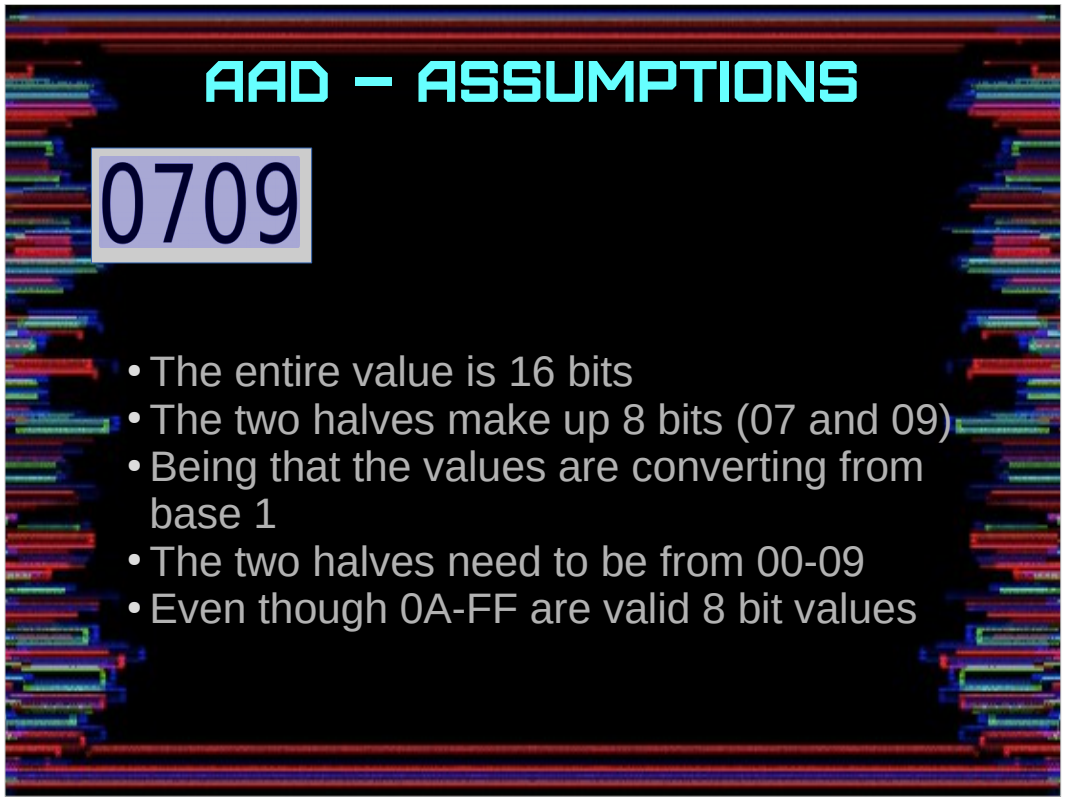
This instruction takes the value of AX (two bytes).

It breaks them out and considers them to be two decimal numbers (base10).

Regardless of the misleading '+' symbol in the slide, it combines the two digits as if the zeros weren't there.

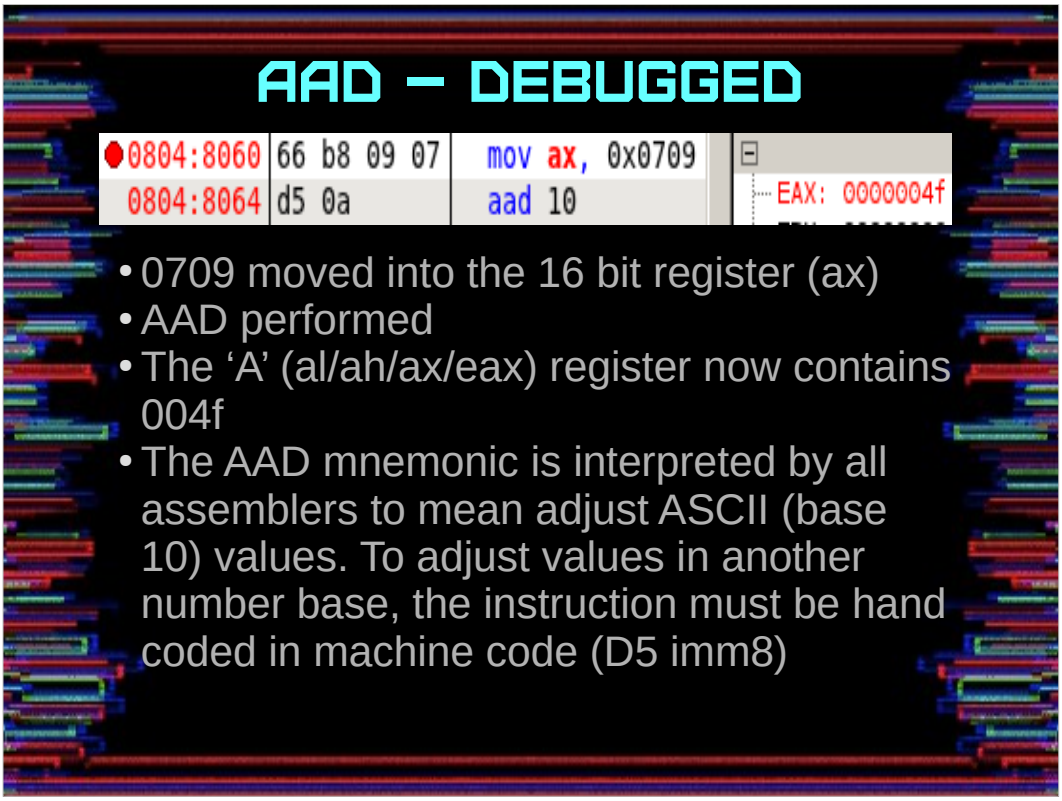
The result is considered a base10 value. It's hexadecimal representation is stored back into AX. This really means that it is stored into AL and AH gets wiped. Because even the largest decimal value of 99 would still fit into AL as hexadecimal.

This style of slides are animated; they will look a little weird in the PDF version.



To think like a hacker for a second, think of the context of what goes wrong when you don't do input validation and the things that could go wrong.

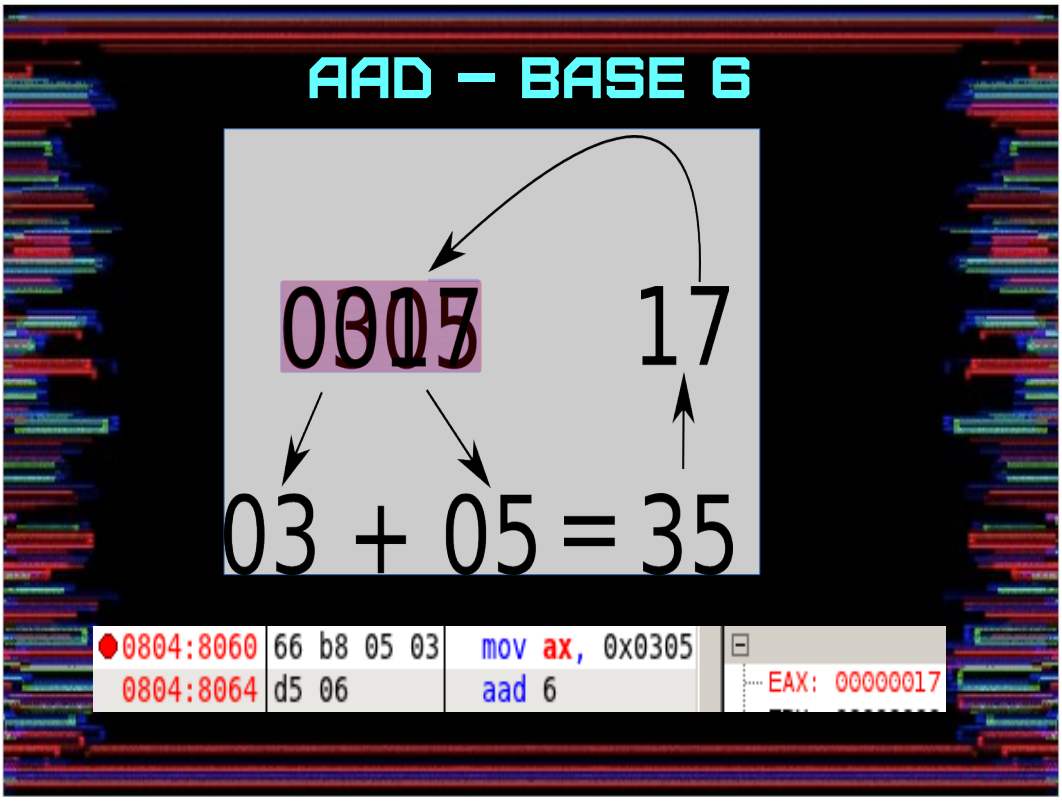
In AX, you're supposed to have a decimal (0-9) value in AH and AL. However, each of these registers could actually be in the range of 0x00-0xFF



The interesting thing here is that the real machine code for the opcode of AAD is just 0xD5, the next byte is actually not part of the opcode; it's an operand. It just defaults to 0x0A (or 10 in decimal). In assembly, you can only type 'aad'; you can't give it the base you want to use because base10 is assumed.

However, if you write this instruction in directly in machine code though, you can actually choose a different base and the high level mathematical concept works out.

Assembly, it's too high level



This is us working through an example of choosing our own arbitrary base of 6.

Our character set for base6 is from 0-5.

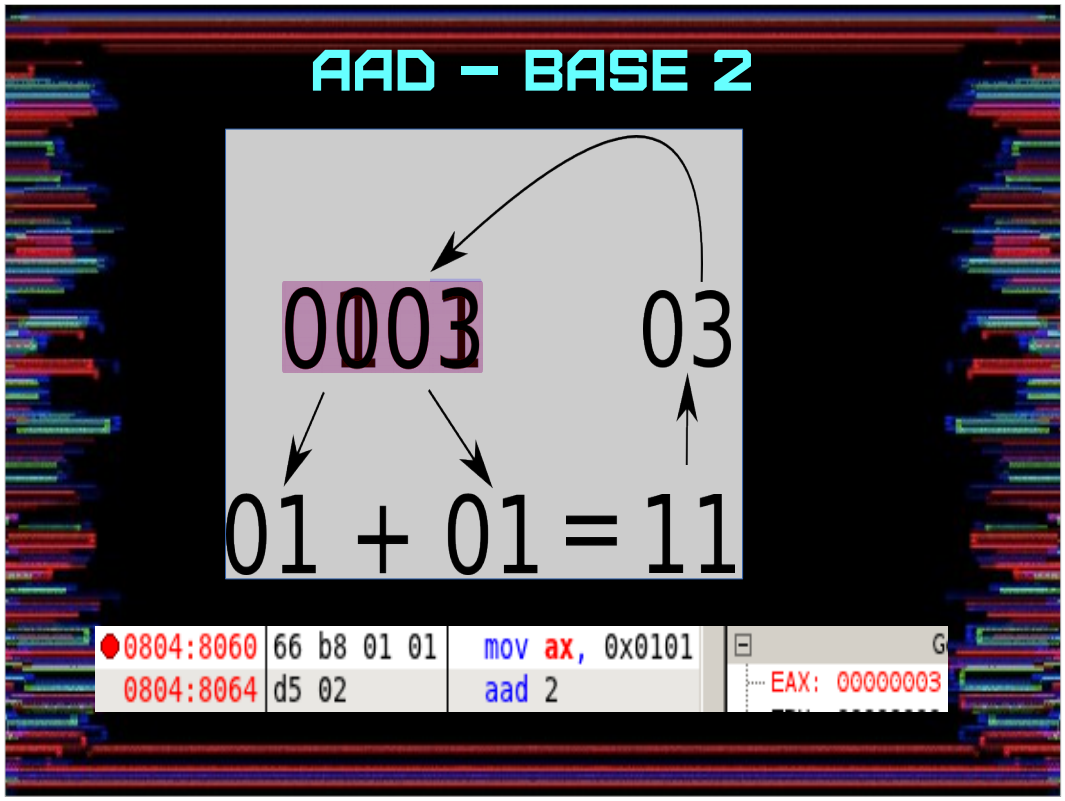
Cramming 3 and 5 together gives us 35.

This instruction needs to convert 35 (base6) to a hexadecimal (base16) value.

35 in base10 is actually 23 = ((3 * 6) + (5 * 1))

23 in hexadecimal is 0x17

It's amazing, it all works out!

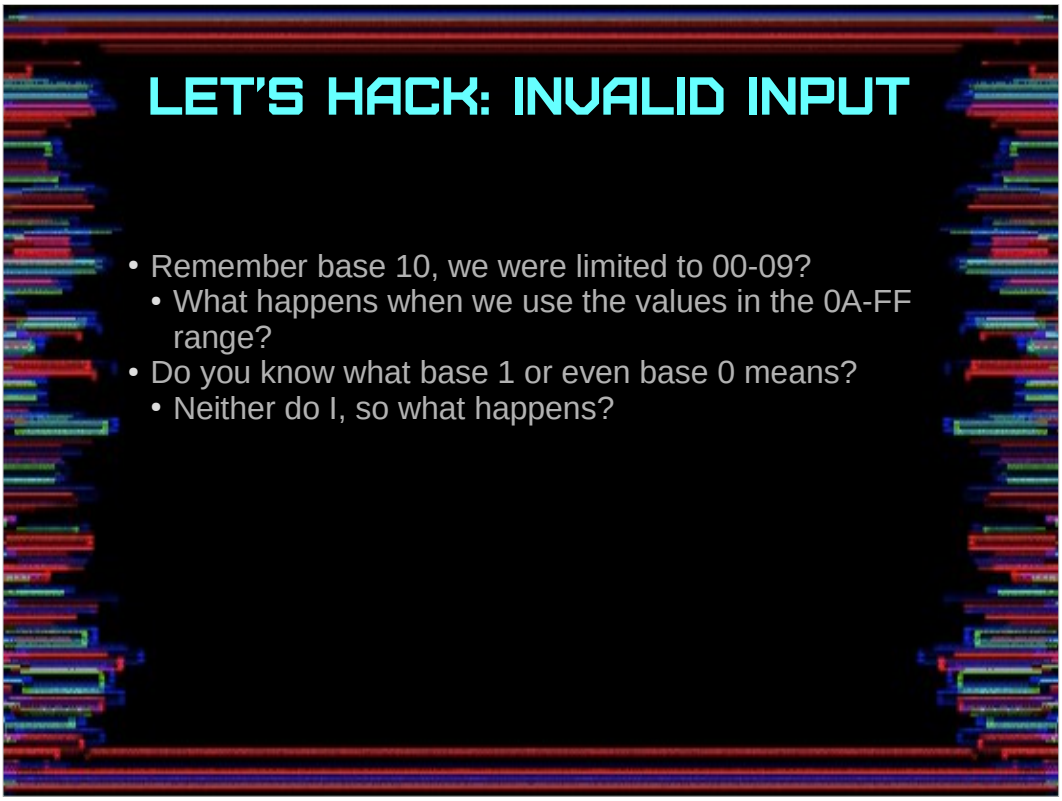


Let's do base2

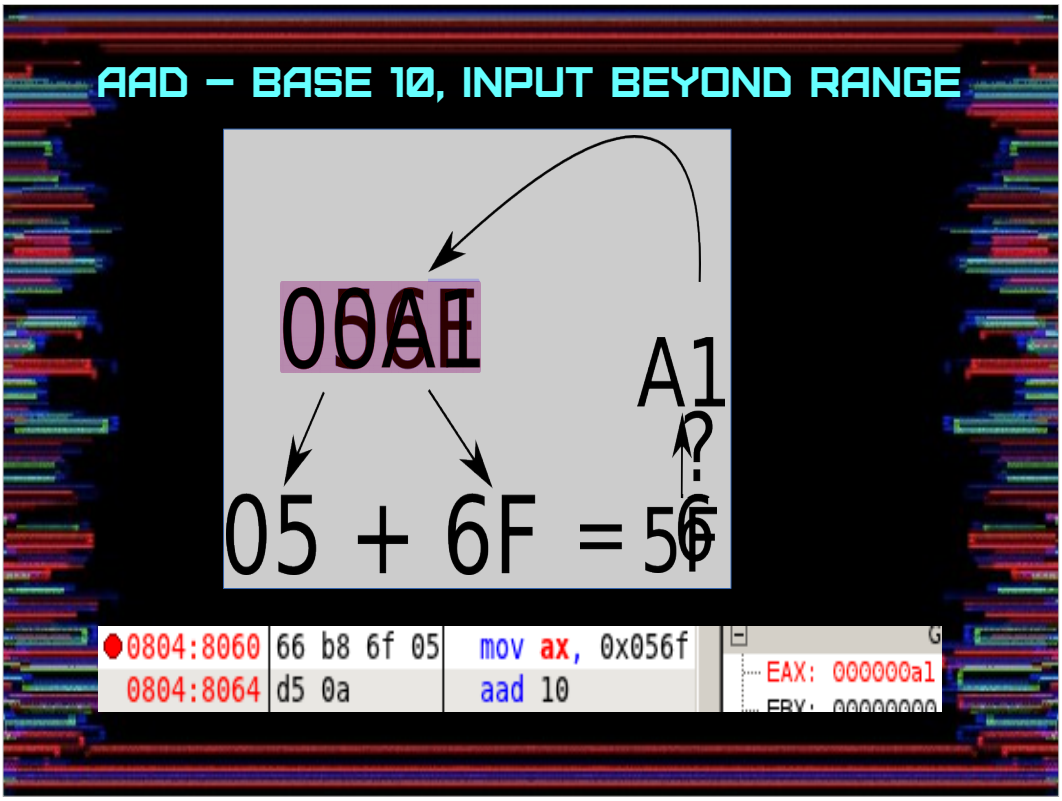
We cram 1 and 1 together and get 11

11 in binary is 3 in decimal which is 0x03 in hexadecimal

So this works too.



This is an introduction slide for us to try some real ignorant things and to attempt to make some meaning out of it

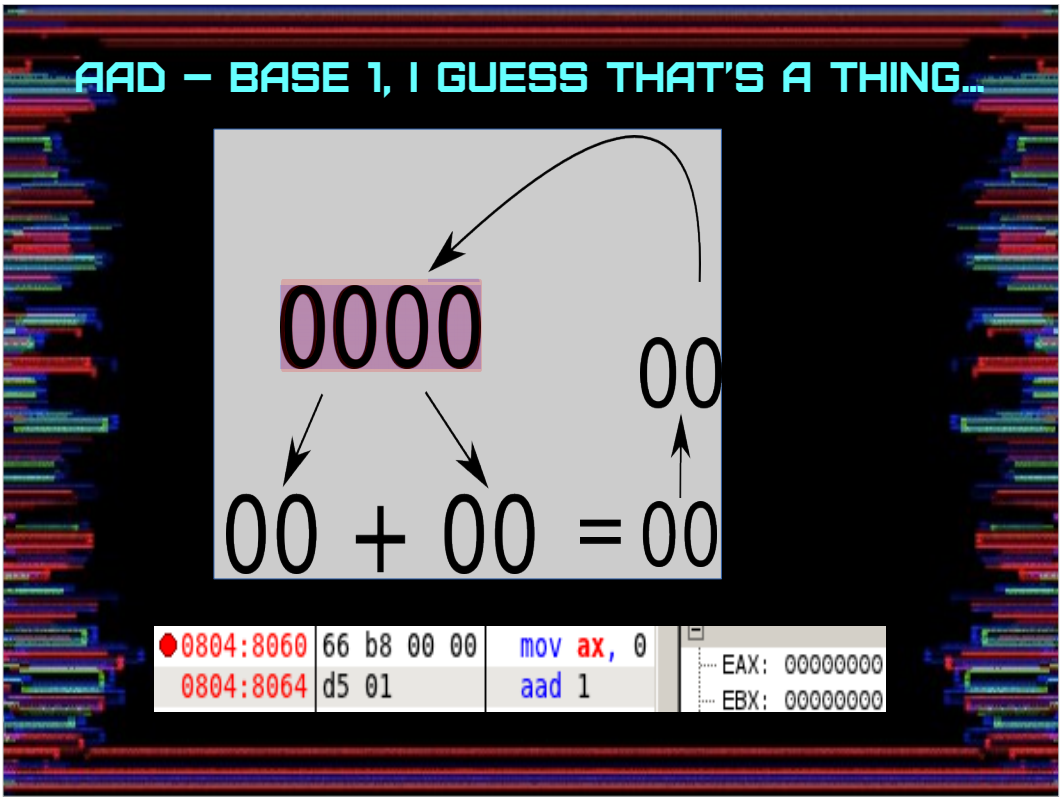


This is us going far above base10 values in AX (AH/AL), but then specifying base10 for the aad instruction.

It's hard to visualize cramming 5 and 6F together, but the slide does it's best to make something of it.

By the process of magic (whatever AAD is actually doing), we get the result of 0xA1.

0xA1 is then stored back into AX

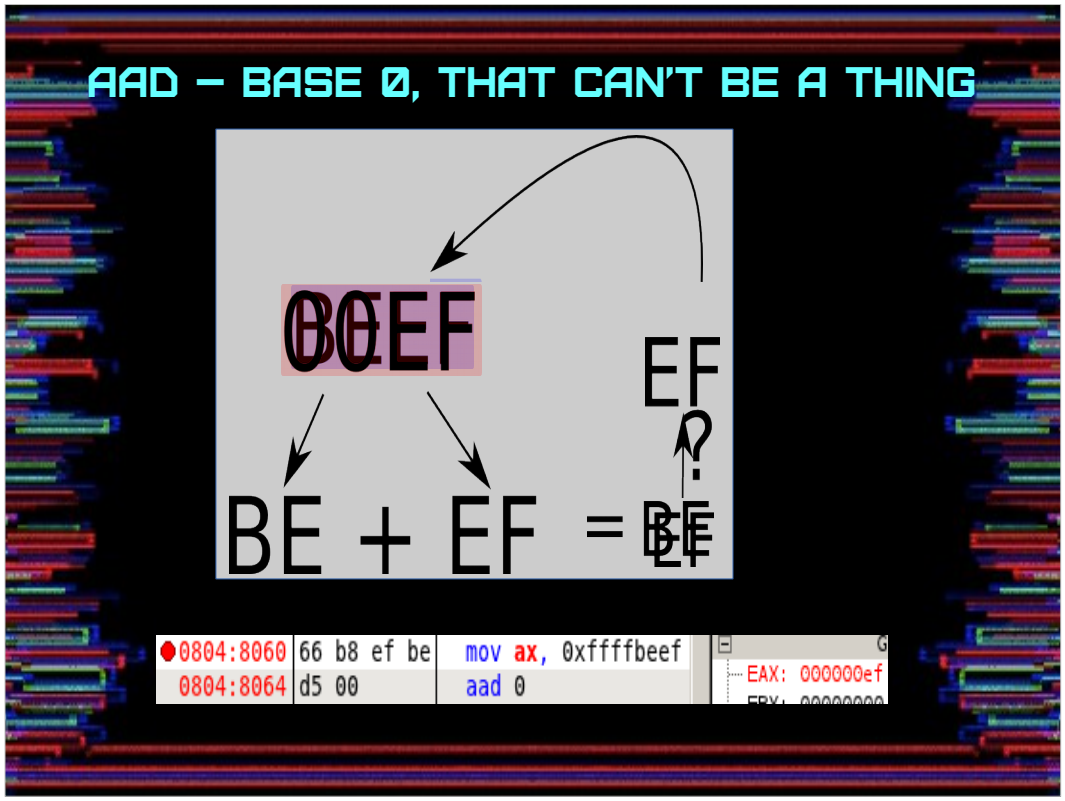


What about base 1?

Well, our only valid character is zero, so:

Cram 0 with 0 to get 0 to convert to 0 and store 0 back into our register that already had 0.

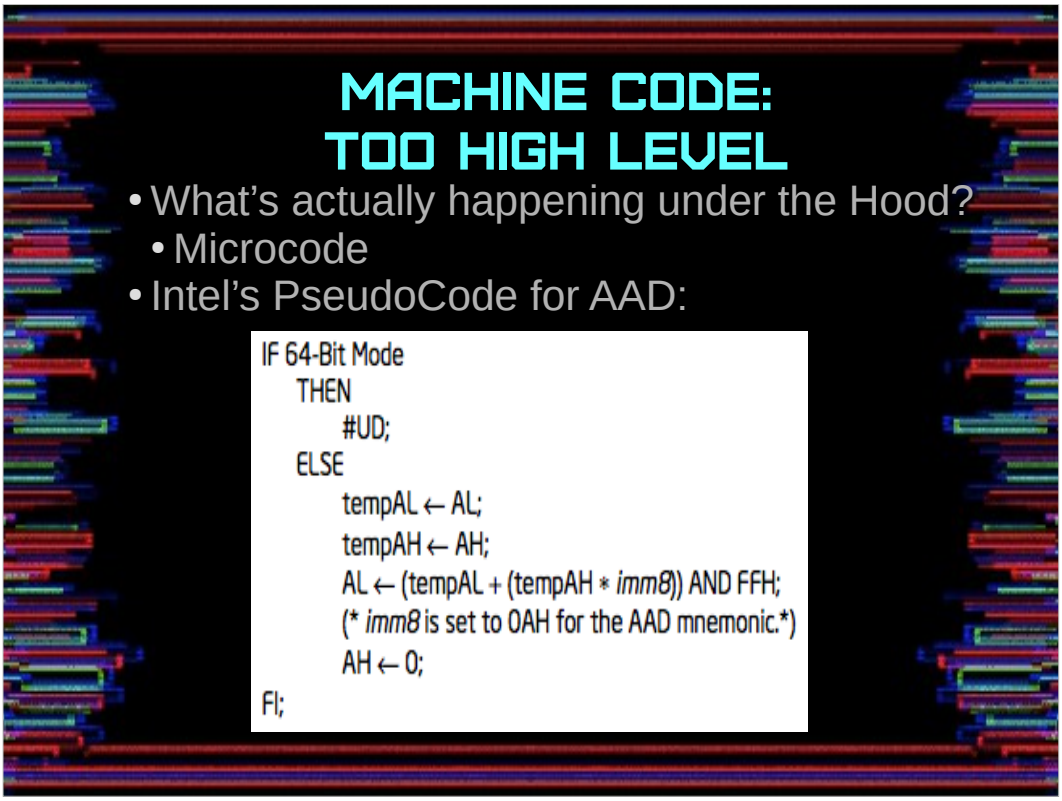
Pointless, but at least it makes sense and we know whats going on here I guess.



Then there's base0. There is really no valid character for this, so I just made AX 0xBEEF.

We cram it together, and by the magical process of AAD we get a result of 0xEF and store it back into AX.

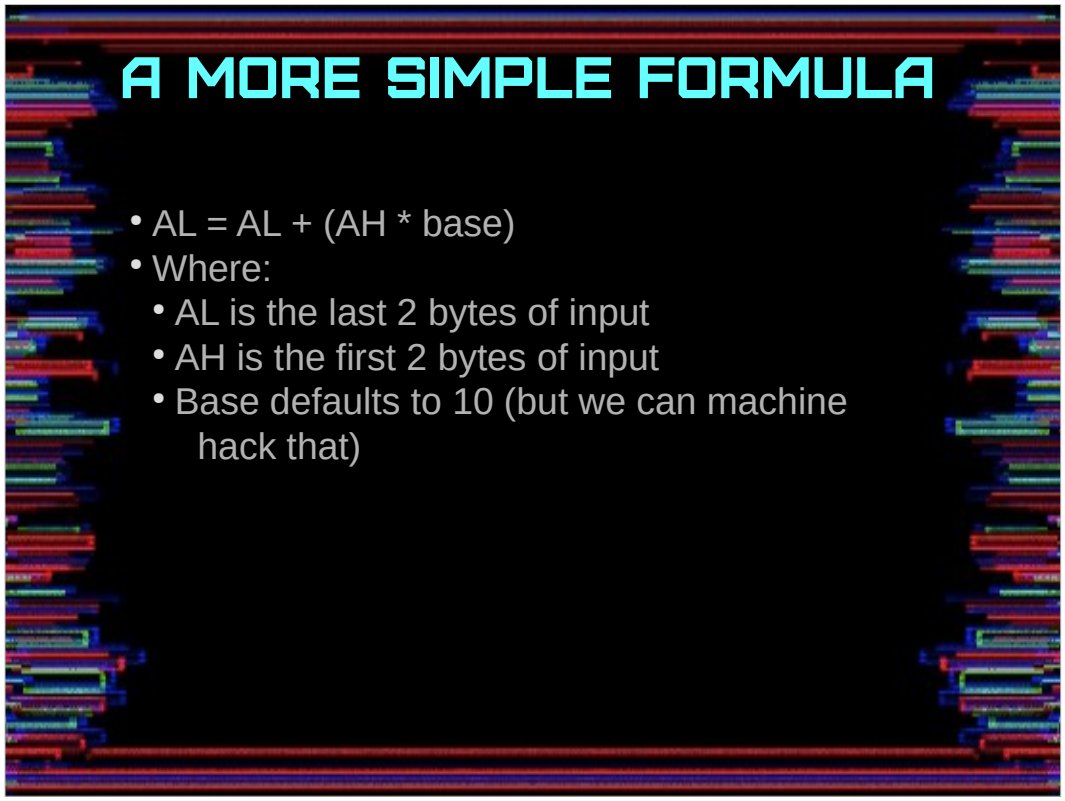

THIS IS
It really is fine though, because microcode



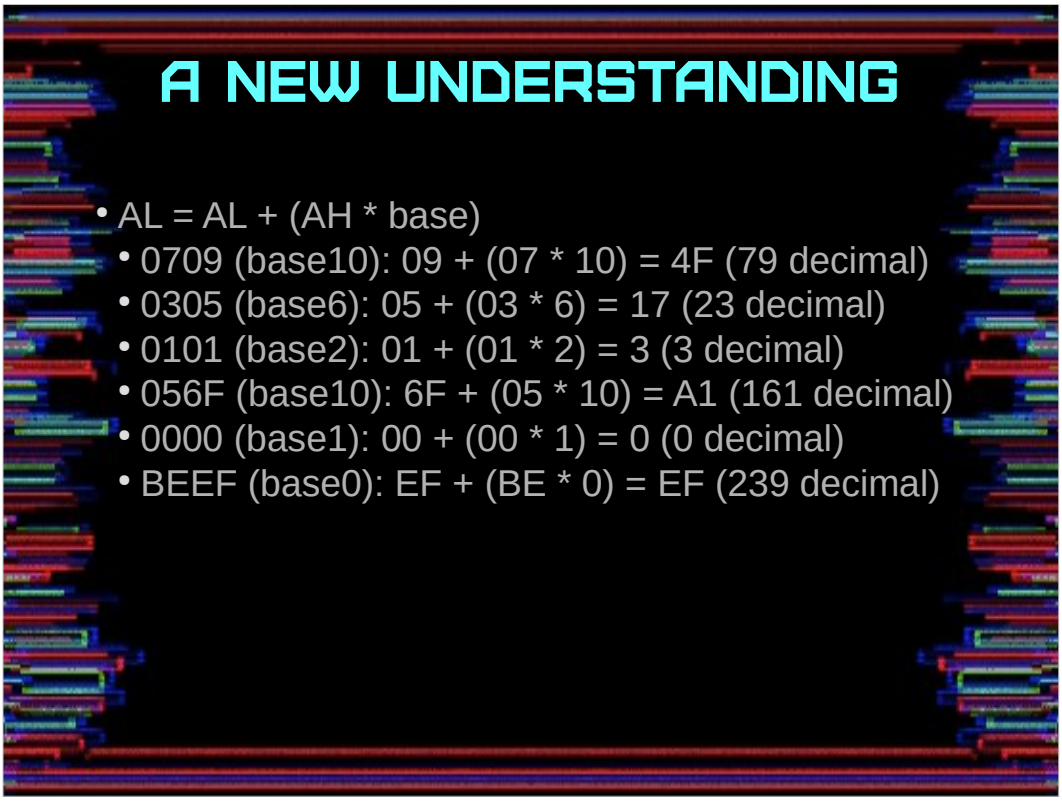
This screenshot from the Intel manual shows what is actually happening under the hood.

It's not literally a base conversion, just some mathematical operations (an 'algorithm') that happen to perform the conversion when you don't feed it garbage.

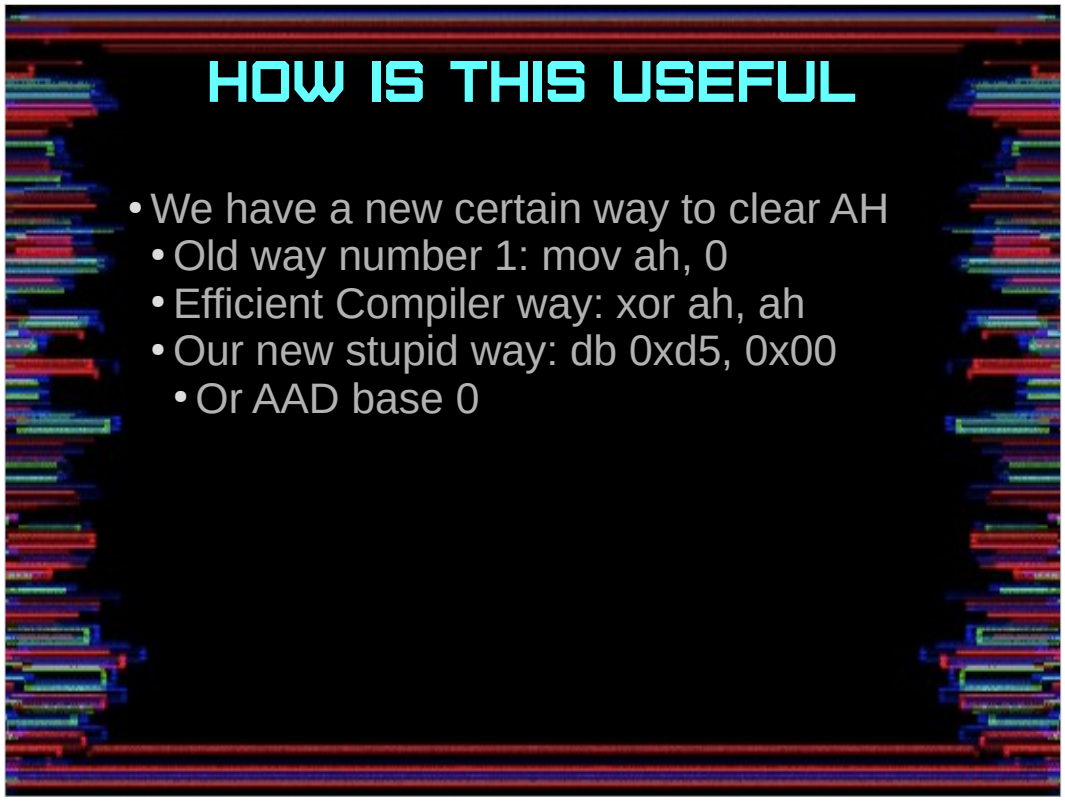
This is fucking profound. Mathematics is not reality, it's just a model for it sometimes. Don't take math too seriously, math is stupid.



This is a better representation of what the Intel pseudo-code is doing. It's actually pretty elegant looking. It's also pretty cool that something so simple can 'convert' 'bases' so easily

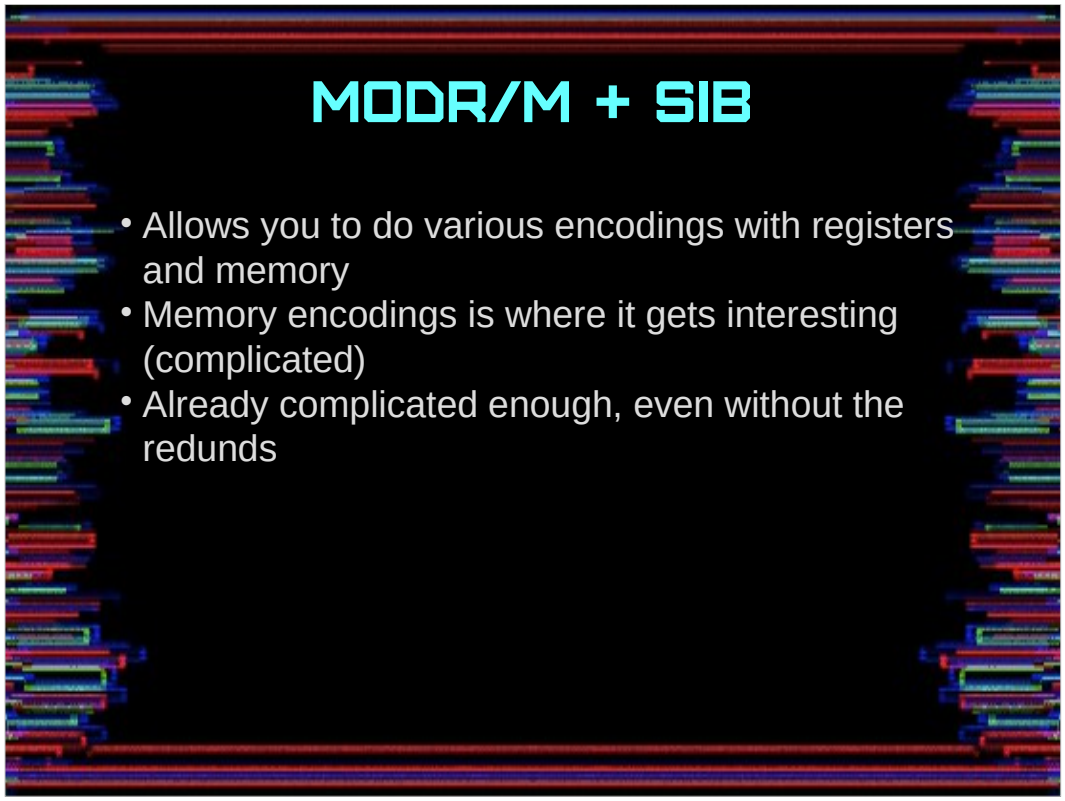


For fun, we use this simple formula to crunch through all of the examples in the previous slides to see that the formula does crunch out the answers that we expect them to.



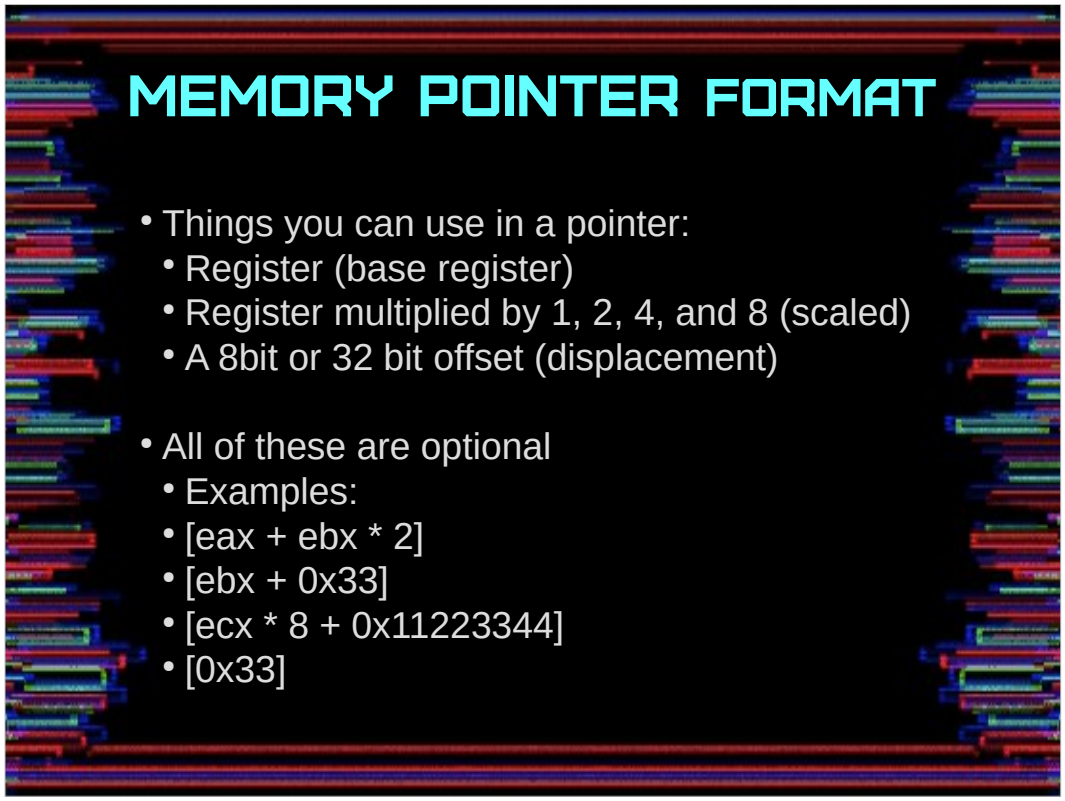
All kidding aside about clearing the AH register, it's cool to know that we can do conversions in obscure bases with one instruction. It's even cooler that the way to implement it is even more obscure: you have to do it in machine code

...because assembly is too high level



This can be some rough terrain right here. Not having to manually do this encoding should make people appreciate assembly language as a super high level language that makes things easier for the programmer. We will be treading this terrain in the next 30ish something slides!

This encoding is used to allow the programmer to use registers and memory pointers as operands



In a memory pointer, you can have a base register, a scaled register, and a displacement. They are all optional, but you at least need to use one of them (otherwise it would be nothing at all)

Of the registers, you have the 8 general purpose ones to choose from (with some major exceptions)

If eax is 0x11223344, XOR [eax], eax will XOR the value of eax with the value in the address of 0x11223344 and store it at that address

You can also add to the address of that pointer with a displacement. [eax + 0x42] would be [0x11223386] (considering what eax originally had above)

MODR/M TABLE

Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte

r8(/r)			AL	CL	DL	BL	AH	CH	DH	BH
r16(/r)			AX	CX	DX	BX	SP	BP	SI	DI
r32(/r)			EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
mm(/r)			MM0	MM1	MM2	MM3	MM4	MM5	MM6	MM7
xmm(/r)			XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7
(In decimal) /digit (Opcode)			0	1	2	3	4	5	6	7
(In binary) REG =			000	001	010	011	100	101	110	111
Effective Address	**Mod**	**R/M**	**Value of ModR/M Byte (in Hexadecimal)**							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[--][--][1]		100	04	0C	14	1C	24	2C	34	3C
disp32[2]		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
[EAX]+disp8[3]	01	000	40	48	50	58	60	68	70	78
[ECX]+disp8		001	41	49	51	59	61	69	71	79
[EDX]+disp8		010	42	4A	52	5A	62	6A	72	7A
[EBX]+disp8		011	43	4B	53	5B	63	6B	73	7B
[--][--]+disp8		100	44	4C	54	5C	64	6C	74	7C
[EBP]+disp8		101	45	4D	55	5D	65	6D	75	7D
[ESI]+disp8		110	46	4E	56	5E	66	6E	76	7E
[EDI]+disp8		111	47	4F	57	5F	67	6F	77	7F
[EAX]+disp32	10	000	80	88	90	98	A0	A8	B0	B8
[ECX]+disp32		001	81	89	91	99	A1	A9	B1	B9
[EDX]+disp32		010	82	8A	92	9B	A2	AA	B2	BA
[EBX]+disp32		011	83	8B	93	9B	A3	AB	B3	BB
[--][--]+disp32		100	84	8C	94	9C	A4	AC	B4	BC
[EBP]+disp32		101	85	8D	95	9D	A5	AD	B5	BD
[ESI]+disp32		110	86	8E	96	9E	A6	AE	B6	BE
[EDI]+disp32		111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL/MM/XMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH/MM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

This is the machine encoding table that makes it all happen (well half of it, the other half is the SIB byte when required).

The MODR/M Table allows for encoding operands as a register, a pointer with one base register, a pointer with a base register and a 8 or 32 bit displacement, or just a 32 bit displacement.

If you want to have a scaled register or mix and match the above with a scaled register, then you need the SIB byte (selectable from this table)

As always, there are many exceptions

XOR EAX, EDX [0X31D0]

Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte

r8(/r)			AL	CL	DL	BL	AH	CH	DH	BH
r16(/r)			AX	CX	DX	BX	SP	BP	SI	DI
r32(/r)			EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
mm(/r)			MM0	MM1	MM2	MM3	MM4	MM5	MM6	MM7
xmm(/r)			XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7
(In decimal) /digit (Opcode)			0	1	2	3	4	5	6	7
(In binary) REG =			000	001	010	011	100	101	110	111
Effective Address	**Mod**	**R/M**	**Value of ModR/M Byte (in Hexadecimal)**							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[--][--][1]		100	04	0C	14	1C	24	2C	34	3C
disp32[2]		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
[EAX]+disp8[3]	01	000	40	48	50	58	60	68	70	78
[ECX]+disp8		001	41	49	51	59	61	69	71	79
[EDX]+disp8		010	42	4A	52	5A	62	6A	72	7A
[EBX]+disp8		011	43	4B	53	5B	63	6B	73	7B
[--][--]+disp8		100	44	4C	54	5C	64	6C	74	7C
[EBP]+disp8		101	45	4D	55	5D	65	6D	75	7D
[ESI]+disp8		110	46	4E	56	5E	66	6E	76	7E
[EDI]+disp8		111	47	4F	57	5F	67	6F	77	7F
[EAX]+disp32	10	000	80	88	90	98	A0	A8	B0	B8
[ECX]+disp32		001	81	89	91	99	A1	A9	B1	B9
[EDX]+disp32		010	82	8A	92	9A	A2	AA	B2	BA
[EBX]+disp32		011	83	8B	93	9B	A3	AB	B3	BB
[--][--]+disp32		100	84	8C	94	9C	A4	AC	B4	BC
[EBP]+disp32		101	85	8D	95	9D	A5	AD	B5	BD
[ESI]+disp32		110	86	8E	96	9E	A6	AE	B6	BE
[EDI]+disp32		111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL/MM/XMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH/MM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

In this slide we work through an example, because we like to explore more than just theory.

In most of our examples, we will use the 0x31 machine opcode for XOR (there are exceptions when we cover redundancies). It's the XOR r/m32, r32 encoding (so first operand can be register or pointer and second operand has to be a register, both 32 bit)

In the table, we line up EAX with EDX to get our 0xD0 value for the operand information for our machine code.

XOR [ECX], EAX [0X3101]

Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte

r8(/r)			AL	CL	DL	BL	AH	CH	DH	BH
r16(/r)			AX	CX	DX	BX	SP	BP	SI	DI
r32(/r)			EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
mm(/r)			MM0	MM1	MM2	MM3	MM4	MM5	MM6	MM7
xmm(/r)			XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7
(In decimal) /digit (Opcode)			0	1	2	3	4	5	6	7
(In binary) REG =			000	001	010	011	100	101	110	111
Effective Address	**Mod**	**R/M**	**Value of ModR/M Byte (in Hexadecimal)**							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[--][--][1]		100	04	0C	14	1C	24	2C	34	3C
disp32[2]		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
[EAX]+disp8[3]	01	000	40	48	50	58	60	68	70	78
[ECX]+disp8		001	41	49	51	59	61	69	71	79
[EDX]+disp8		010	42	4A	52	5A	62	6A	72	7A
[EBX]+disp8		011	43	4B	53	5B	63	6B	73	7B
[--][--]+disp8		100	44	4C	54	5C	64	6C	74	7C
[EBP]+disp8		101	45	4D	55	5D	65	6D	75	7D
[ESI]+disp8		110	46	4E	56	5E	66	6E	76	7E
[EDI]+disp8		111	47	4F	57	5F	67	6F	77	7F
[EAX]+disp32	10	000	80	88	90	98	A0	A8	B0	B8
[ECX]+disp32		001	81	89	91	99	A1	A9	B1	B9
[EDX]+disp32		010	82	8A	92	9A	A2	AA	B2	BA
[EBX]+disp32		011	83	8B	93	9B	A3	AB	B3	BB
[--][--]+disp32		100	84	8C	94	9C	A4	AC	B4	BC
[EBP]+disp32		101	85	8D	95	9D	A5	AD	B5	BD
[ESI]+disp32		110	86	8E	96	9E	A6	AE	B6	BE
[EDI]+disp32		111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL/MM/XMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH/MM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

Next we do a pointer for the first operand. Note we are still starting with the 0x31 encoding for XOR

We are using the pointer of [ECX] for the first operand and EAX for the second operand. All we have to do is line them up to arrive at the 0x01 byte for the machine code byte to encode this. It's just as straight forward as the last example

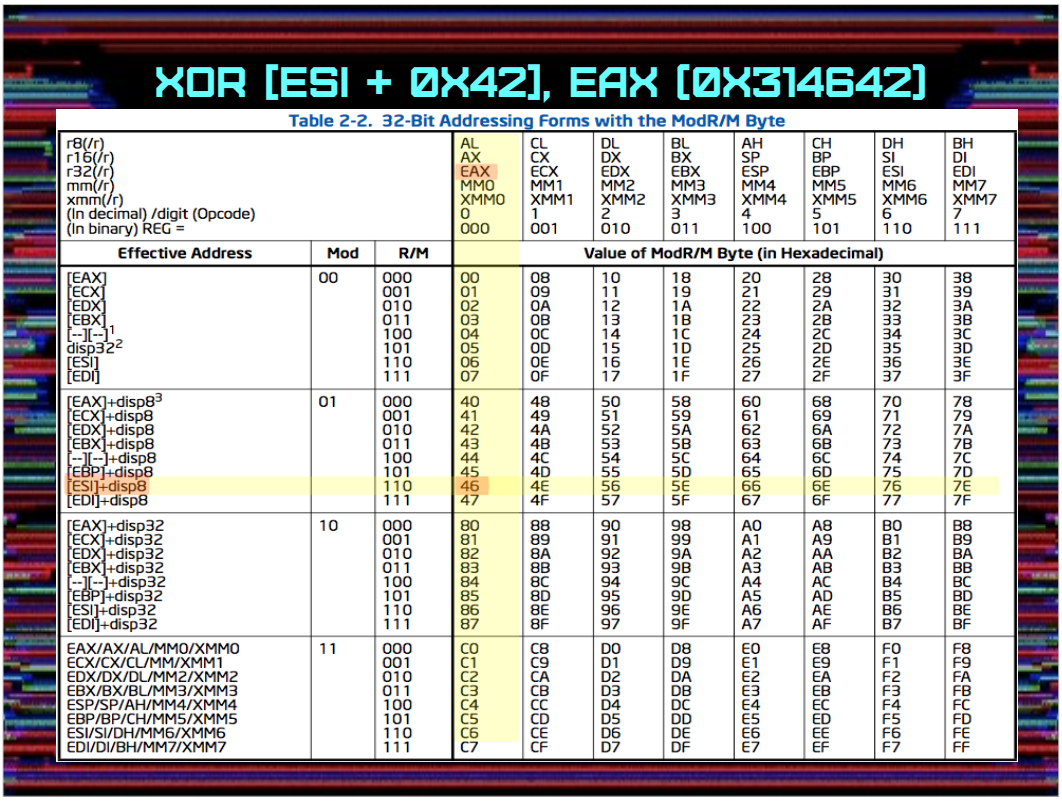


Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte

r8(/r)			AL	CL	DL	BL	AH	CH	DH	BH
r16(/r)			AX	CX	DX	BX	SP	BP	SI	DI
r32(/r)			EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
mm(/r)			MM0	MM1	MM2	MM3	MM4	MM5	MM6	MM7
xmm(/r)			XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7
(In decimal) /digit (Opcode)			0	1	2	3	4	5	6	7
(In binary) REG =			000	001	010	011	100	101	110	111
Effective Address	**Mod**	**R/M**	**Value of ModR/M Byte (in Hexadecimal)**							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[--][--][1]		100	04	0C	14	1C	24	2C	34	3C
disp32[2]		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
[EAX]+disp8[3]	01	000	40	48	50	58	60	68	70	78
[ECX]+disp8		001	41	49	51	59	61	69	71	79
[EDX]+disp8		010	42	4A	52	5A	62	6A	72	7A
[EBX]+disp8		011	43	4B	53	5B	63	6B	73	7B
[--][--]+disp8		100	44	4C	54	5C	64	6C	74	7C
[EBP]+disp8		101	45	4D	55	5D	65	6D	75	7D
[ESI]+disp8		110	46	4E	56	5E	66	6E	76	7E
[EDI]+disp8		111	47	4F	57	5F	67	6F	77	7F
[EAX]+disp32	10	000	80	88	90	98	A0	A8	B0	B8
[ECX]+disp32		001	81	89	91	99	A1	A9	B1	B9
[EDX]+disp32		010	82	8A	92	9A	A2	AA	B2	BA
[EBX]+disp32		011	83	8B	93	9B	A3	AB	B3	BB
[--][--]+disp32		100	84	8C	94	9C	A4	AC	B4	BC
[EBP]+disp32		101	85	8D	95	9D	A5	AD	B5	BD
[ESI]+disp32		110	86	8E	96	9E	A6	AE	B6	BE
[EDI]+disp32		111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL/MM/XMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH/MM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

This one adds one little extra bit of complexity.

We first start with our 0x31 for XOR. Next we have a pointer of [ESI + 0x42] and then EAX.

EAX is easy to line up at the top. For the first operand, we need to find a line that supports ESI plus a 1 byte displacement. It is shown in the screenshot as 0x46

But we aren't done, the processor then expects the next byte of the instruction to actually be that offset, so the 0x42 displacement comes as the next byte

XOR [EBX + 0XFFF31337], ESP [0X31A33713F3FF]

Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte

r8(/r)			AL	CL	DL	BL	AH	CH	DH	BH
r16(/r)			AX	CX	DX	BX	SP	BP	SI	DI
r32(/r)			EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
mm(/r)			MM0	MM1	MM2	MM3	MM4	MM5	MM6	MM7
xmm(/r)			XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7
(In decimal) /digit (Opcode)			0	1	2	3	4	5	6	7
(In binary) REG =			000	001	010	011	100	101	110	111
Effective Address	**Mod**	**R/M**	**Value of ModR/M Byte (in Hexadecimal)**							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[--][--][1]		100	04	0C	14	1C	24	2C	34	3C
disp32[2]		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
[EAX]+disp8[3]	01	000	40	48	50	58	60	68	70	78
[ECX]+disp8		001	41	49	51	59	61	69	71	79
[EDX]+disp8		010	42	4A	52	5A	62	6A	72	7A
[EBX]+disp8		011	43	4B	53	5B	63	6B	73	7B
[--][--]+disp8		100	44	4C	54	5C	64	6C	74	7C
[EBP]+disp8		101	45	4D	55	5D	65	6D	75	7D
[ESI]+disp8		110	46	4E	56	5E	66	6E	76	7E
[EDI]+disp8		111	47	4F	57	5F	67	6F	77	7F
[EAX]+disp32	10	000	80	88	90	98	A0	A8	B0	B8
[ECX]+disp32		001	81	89	91	99	A1	A9	B1	B9
[EDX]+disp32		010	82	8A	92	9A	A2	AA	B2	BA
[EBX]+disp32		011	83	8B	93	9B	A3	AB	B3	BB
[--][--]+disp32		100	84	8C	94	9C	A4	AC	B4	BC
[EBP]+disp32		101	85	8D	95	9D	A5	AD	B5	BD
[ESI]+disp32		110	86	8E	96	9E	A6	AE	B6	BE
[EDI]+disp32		111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL/MM/XMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH/MM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

If the previous example made sense, this one should be just as easy.

We need to find a pointer that supports EBX plus a 32 bit displacement and the register of ESP. When lining this up on the table, we find that it is 0xA4.

The only thing that may appear confusing to those that don't know is that Intel encodes addresses in Little-Endian form. This is just another way to say that bytes are in backward order. So 0xFFF31337 becomes 0x3713F3FF after our machine code of 0x31A3.

This makes the entire instruction: 0x31A33713F3FF

XOR [0X42], EAX [0X3105420000000]

Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte

r8(/r)			AL	CL	DL	BL	AH	CH	DH	BH
r16(/r)			AX	CX	DX	BX	SP	BP	SI	DI
r32(/r)			EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
mm(/r)			MM0	MM1	MM2	MM3	MM4	MM5	MM6	MM7
xmm(/r)			XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7
(In decimal) /digit (Opcode)			0	1	2	3	4	5	6	7
(In binary) REG =			000	001	010	011	100	101	110	111
Effective Address	**Mod**	**R/M**	**Value of ModR/M Byte (in Hexadecimal)**							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[--][--][1]		100	04	0C	14	1C	24	2C	34	3C
disp32[2]		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
[EAX]+disp8[3]	01	000	40	48	50	58	60	68	70	78
[ECX]+disp8		001	41	49	51	59	61	69	71	79
[EDX]+disp8		010	42	4A	52	5A	62	6A	72	7A
[EBX]+disp8		011	43	4B	53	5B	63	6B	73	7B
[--][--]+disp8		100	44	4C	54	5C	64	6C	74	7C
[EBP]+disp8		101	45	4D	55	5D	65	6D	75	7D
[ESI]+disp8		110	46	4E	56	5E	66	6E	76	7E
[EDI]+disp8		111	47	4F	57	5F	67	6F	77	7F
[EAX]+disp32	10	000	80	88	90	98	A0	A8	B0	B8
[ECX]+disp32		001	81	89	91	99	A1	A9	B1	B9
[EDX]+disp32		010	82	8A	92	9A	A2	AA	B2	BA
[EBX]+disp32		011	83	8B	93	9B	A3	AB	B3	BB
[--][--]+disp32		100	84	8C	94	9C	A4	AC	B4	BC
[EBP]+disp32		101	85	8D	95	9D	A5	AD	B5	BD
[ESI]+disp32		110	86	8E	96	9E	A6	AE	B6	BE
[EDI]+disp32		111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL/MM/XMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH/MM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

This is looking at not using any registers for our pointer. This examples just demonstrates a literal displacement of 0x42.

We need to find the horizontal line that encodes for only a displacement and the vertical line for EAX. There are no horizontal lines for just an 8 bit displacement, so we are forced to use the 32 bit one and just pad the first 3 bytes with nulls.

So we have our 0x31 for XOR, 0x05 for the operand encoding from the chart, and 0x42000000 for the displacement data (ordered like that because Little-Endian)

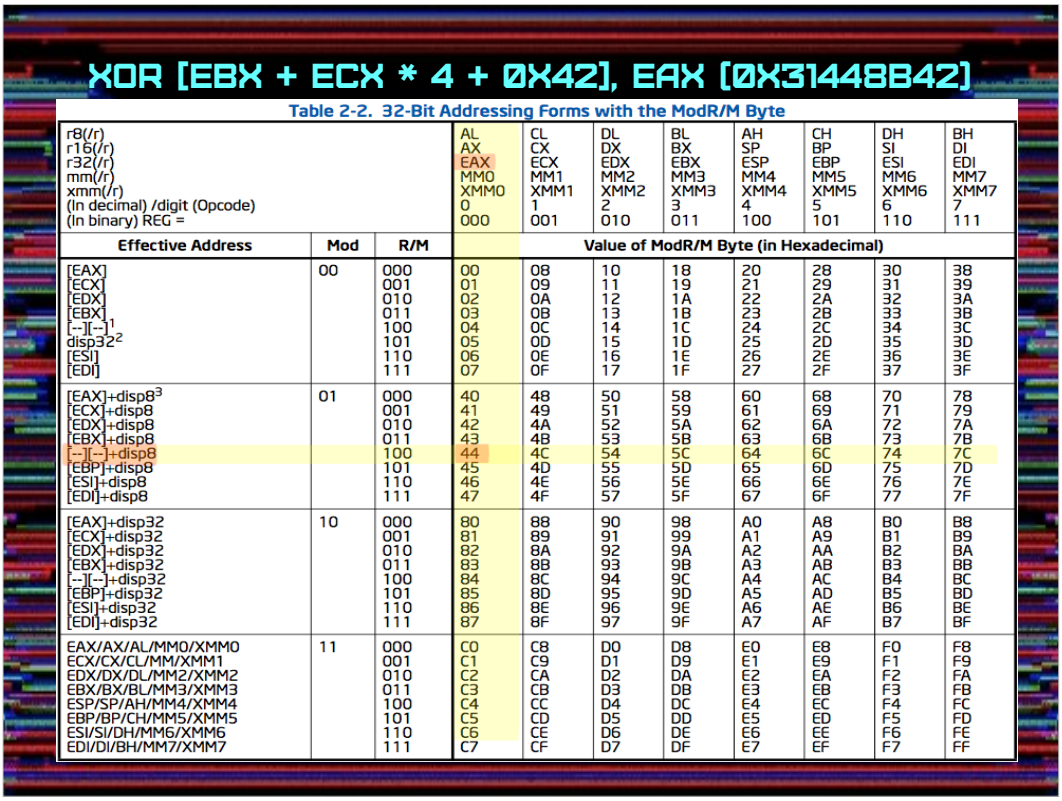


Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte

r8(/r)			AL	CL	DL	BL	AH	CH	DH	BH
r16(/r)			AX	CX	DX	BX	SP	BP	SI	DI
r32(/r)			EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
mm(/r)			MM0	MM1	MM2	MM3	MM4	MM5	MM6	MM7
xmm(/r)			XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7
(In decimal) /digit (Opcode)			0	1	2	3	4	5	6	7
(In binary) REG =			000	001	010	011	100	101	110	111
Effective Address	**Mod**	**R/M**	**Value of ModR/M Byte (in Hexadecimal)**							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[--][--][1]		100	04	0C	14	1C	24	2C	34	3C
disp32[2]		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
[EAX]+disp8[3]	01	000	40	48	50	58	60	68	70	78
[ECX]+disp8		001	41	49	51	59	61	69	71	79
[EDX]+disp8		010	42	4A	52	5A	62	6A	72	7A
[EBX]+disp8		011	43	4B	53	5B	63	6B	73	7B
[--][--]+disp8		100	44	4C	54	5C	64	6C	74	7C
[EBP]+disp8		101	45	4D	55	5D	65	6D	75	7D
[ESI]+disp8		110	46	4E	56	5E	66	6E	76	7E
[EDI]+disp8		111	47	4F	57	5F	67	6F	77	7F
[EAX]+disp32	10	000	80	88	90	98	A0	A8	B0	B8
[ECX]+disp32		001	81	89	91	99	A1	A9	B1	B9
[EDX]+disp32		010	82	8A	92	9A	A2	AA	B2	BA
[EBX]+disp32		011	83	8B	93	9B	A3	AB	B3	BB
[--][--]+disp32		100	84	8C	94	9C	A4	AC	B4	BC
[EBP]+disp32		101	85	8D	95	9D	A5	AD	B5	BD
[ESI]+disp32		110	86	8E	96	9E	A6	AE	B6	BE
[EDI]+disp32		111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL/MM/XMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH/MM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

Now we start to get a little crazier; we are going to use a scaled register.

Lining up the second operand of EAX on the chart is easy. To use a scaled register, we need the SIB byte, which is one of the horizontal options using [--][--].

There are 3 different variations of this SIB option, one without a displacement, one with an 8 bit displacement, and another with a 32 bit displacement. In this case, it's just the 8 bit displacement. So we choose 0x44 in this table, and then look next to our SIB table to pick the actual Base and Scaled register

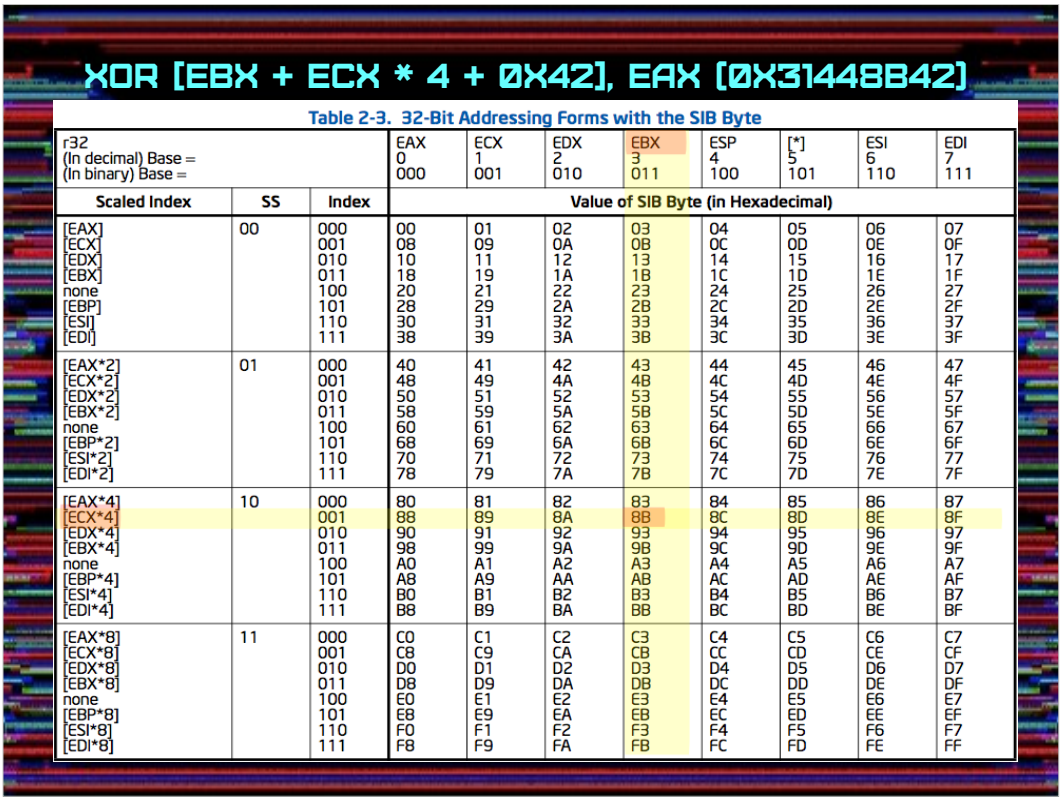


Table 2-3. 32-Bit Addressing Forms with the SIB Byte

r32 (In decimal) Base = (In binary) Base =			EAX 0 000	ECX 1 001	EDX 2 010	EBX 3 011	ESP 4 100	[*] 5 101	ESI 6 110	EDI 7 111
Scaled Index	**SS**	**Index**	**Value of SIB Byte (in Hexadecimal)**							
[EAX]	00	000	00	01	02	03	04	05	06	07
[ECX]		001	08	09	0A	0B	0C	0D	0E	0F
[EDX]		010	10	11	12	13	14	15	16	17
[EBX]		011	18	19	1A	1B	1C	1D	1E	1F
none		100	20	21	22	23	24	25	26	27
[EBP]		101	28	29	2A	2B	2C	2D	2E	2F
[ESI]		110	30	31	32	33	34	35	36	37
[EDI]		111	38	39	3A	3B	3C	3D	3E	3F
[EAX*2]	01	000	40	41	42	43	44	45	46	47
[ECX*2]		001	48	49	4A	4B	4C	4D	4E	4F
[EDX*2]		010	50	51	52	53	54	55	56	57
[EBX*2]		011	58	59	5A	5B	5C	5D	5E	5F
none		100	60	61	62	63	64	65	66	67
[EBP*2]		101	68	69	6A	6B	6C	6D	6E	6F
[ESI*2]		110	70	71	72	73	74	75	76	77
[EDI*2]		111	78	79	7A	7B	7C	7D	7E	7F
[EAX*4]	10	000	80	81	82	83	84	85	86	87
[ECX*4]		001	88	89	8A	8B	8C	8D	8E	8F
[EDX*4]		010	90	91	92	93	94	95	96	97
[EBX*4]		011	98	99	9A	9B	9C	9D	9E	9F
none		100	A0	A1	A2	A3	A4	A5	A6	A7
[EBP*4]		101	A8	A9	AA	AB	AC	AD	AE	AF
[ESI*4]		110	B0	B1	B2	B3	B4	B5	B6	B7
[EDI*4]		111	B8	B9	BA	BB	BC	BD	BE	BF
[EAX*8]	11	000	C0	C1	C2	C3	C4	C5	C6	C7
[ECX*8]		001	C8	C9	CA	CB	CC	CD	CE	CF
[EDX*8]		010	D0	D1	D2	D3	D4	D5	D6	D7
[EBX*8]		011	D8	D9	DA	DB	DC	DD	DE	DF
none		100	E0	E1	E2	E3	E4	E5	E6	E7
[EBP*8]		101	E8	E9	EA	EB	EC	ED	EE	EF
[ESI*8]		110	F0	F1	F2	F3	F4	F5	F6	F7
[EDI*8]		111	F8	F9	FA	FB	FC	FD	FE	FF

The Base register will be the vertical line and the Scaled (multiplied register) will be the horizontal line.

Finding EBX (vertical base register) is the easiest.

For the horizontal line, we must find the item that uses ECX and is also * 4. This is actually not terribly hard to find on the table either.

When you line this up, you get 0x8B for the SIB byte.

Finally, we have the displacement of 0x42 to add to the end of the instruction to get our final result

XOR [ESP], EAX [0X310424]

Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte

r8(/r) r16(/r) r32(/r) mm(/r) xmm(/r) (In decimal) /digit (Opcode) (In binary) REG =			AL AX EAX MM0 XMM0 0 000	CL CX ECX MM1 XMM1 1 001	DL DX EDX MM2 XMM2 2 010	BL BX EBX MM3 XMM3 3 011	AH SP ESP MM4 XMM4 4 100	CH BP EBP MM5 XMM5 5 101	DH SI ESI MM6 XMM6 6 110	BH DI EDI MM7 XMM7 7 111
Effective Address	**Mod**	**R/M**	**Value of ModR/M Byte (in Hexadecimal)**							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[--][--][1]		100	04	0C	14	1C	24	2C	34	3C
disp32[2]		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
[EAX]+disp8[3]	01	000	40	48	50	58	60	68	70	78
[ECX]+disp8		001	41	49	51	59	61	69	71	79
[EDX]+disp8		010	42	4A	52	5A	62	6A	72	7A
[EBX]+disp8		011	43	4B	53	5B	63	6B	73	7B
[--][--]+disp8		100	44	4C	54	5C	64	6C	74	7C
[EBP]+disp8		101	45	4D	55	5D	65	6D	75	7D
[ESI]+disp8		110	46	4E	56	5E	66	6E	76	7E
[EDI]+disp8		111	47	4F	57	5F	67	6F	77	7F
[EAX]+disp32	10	000	80	88	90	98	A0	A8	B0	B8
[ECX]+disp32		001	81	89	91	99	A1	A9	B1	B9
[EDX]+disp32		010	82	8A	92	9A	A2	AA	B2	BA
[EBX]+disp32		011	83	8B	93	9B	A3	AB	B3	BB
[--][--]+disp32		100	84	8C	94	9C	A4	AC	B4	BC
[EBP]+disp32		101	85	8D	95	9D	A5	AD	B5	BD
[ESI]+disp32		110	86	8E	96	9E	A6	AE	B6	BE
[EDI]+disp32		111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL/MM/XMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH/MM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

Now lets dig into some weird exceptions; lets start with using ESP as the base register in a pointer. When looking at the table, ESP isn't an option?

However, we know from the SIB byte that you can choose a Base register, although you have to choose a Scaled register as well. But did you notice from the table on the last slide that 'none' was an option for the Scaled register. That's the hack that assemblers use.

For the MODR/M byte, we line up EAX for the vertical and the [--][--] (SIB) for no displacement. This gives us 0x04 for our MODR/M byte.

Next let's look at what we do with the SIB byte.

XOR [ESP], EAX [0X310424]

Table 2-3. 32-Bit Addressing Forms with the SIB Byte

r32 (In decimal) Base = (In binary) Base =			EAX 0 000	ECX 1 001	EDX 2 010	EBX 3 011	ESP 4 100	[*] 5 101	ESI 6 110	EDI 7 111
Scaled Index	**SS**	**Index**	**Value of SIB Byte (in Hexadecimal)**							
[EAX]	00	000	00	01	02	03	04	05	06	07
[ECX]		001	08	09	0A	0B	0C	0D	0E	0F
[EDX]		010	10	11	12	13	14	15	16	17
[EBX]		011	18	19	1A	1B	1C	1D	1E	1F
none		100	20	21	22	23	24	25	26	27
[EBP]		101	28	29	2A	2B	2C	2D	2E	2F
[ESI]		110	30	31	32	33	34	35	36	37
[EDI]		111	38	39	3A	3B	3C	3D	3E	3F
[EAX*2]	01	000	40	41	42	43	44	45	46	47
[ECX*2]		001	48	49	4A	4B	4C	4D	4E	4F
[EDX*2]		010	50	51	52	53	54	55	56	57
[EBX*2]		011	58	59	5A	5B	5C	5D	5E	5F
none		100	60	61	62	63	64	65	66	67
[EBP*2]		101	68	69	6A	6B	6C	6D	6E	6F
[ESI*2]		110	70	71	72	73	74	75	76	77
[EDI*2]		111	78	79	7A	7B	7C	7D	7E	7F
[EAX*4]	10	000	80	81	82	83	84	85	86	87
[ECX*4]		001	88	89	8A	8B	8C	8D	8E	8F
[EDX*4]		010	90	91	92	93	94	95	96	97
[EBX*4]		011	98	99	9A	9B	9C	9D	9E	9F
none		100	A0	A1	A2	A3	A4	A5	A6	A7
[EBP*4]		101	A8	A9	AA	AB	AC	AD	AE	AF
[ESI*4]		110	B0	B1	B2	B3	B4	B5	B6	B7
[EDI*4]		111	B8	B9	BA	BB	BC	BD	BE	BF
[EAX*8]	11	000	C0	C1	C2	C3	C4	C5	C6	C7
[ECX*8]		001	C8	C9	CA	CB	CC	CD	CE	CF
[EDX*8]		010	D0	D1	D2	D3	D4	D5	D6	D7
[EBX*8]		011	D8	D9	DA	DB	DC	DD	DE	DF
none		100	E0	E1	E2	E3	E4	E5	E6	E7
[EBP*8]		101	E8	E9	EA	EB	EC	ED	EE	EF
[ESI*8]		110	F0	F1	F2	F3	F4	F5	F6	F7
[EDI*8]		111	F8	F9	FA	FB	FC	FD	FE	FF

Since ESP is our Base register, we line that up vertically. We choose the first 'none' horizontal line for the Scaled register to give us 0x24.

So what's the difference between that 'none' and the 3 others. There isn't any in this particular case, hence the next slide

In this slide we see the PoC of using all 4 of the 'none' options in the SIB byte. This is to note that the assembly is the same for any of these

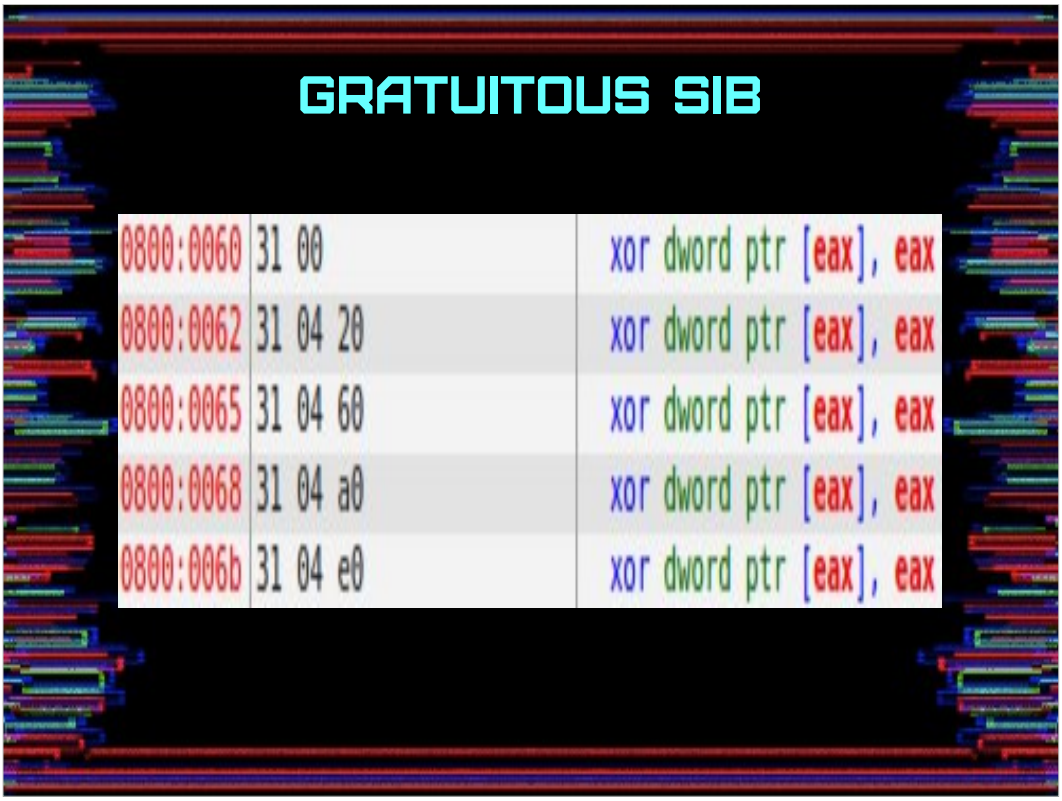
USING SIB WHEN YOU DON'T NEED TO

Table 2-3. 32-Bit Addressing Forms with the SIB Byte

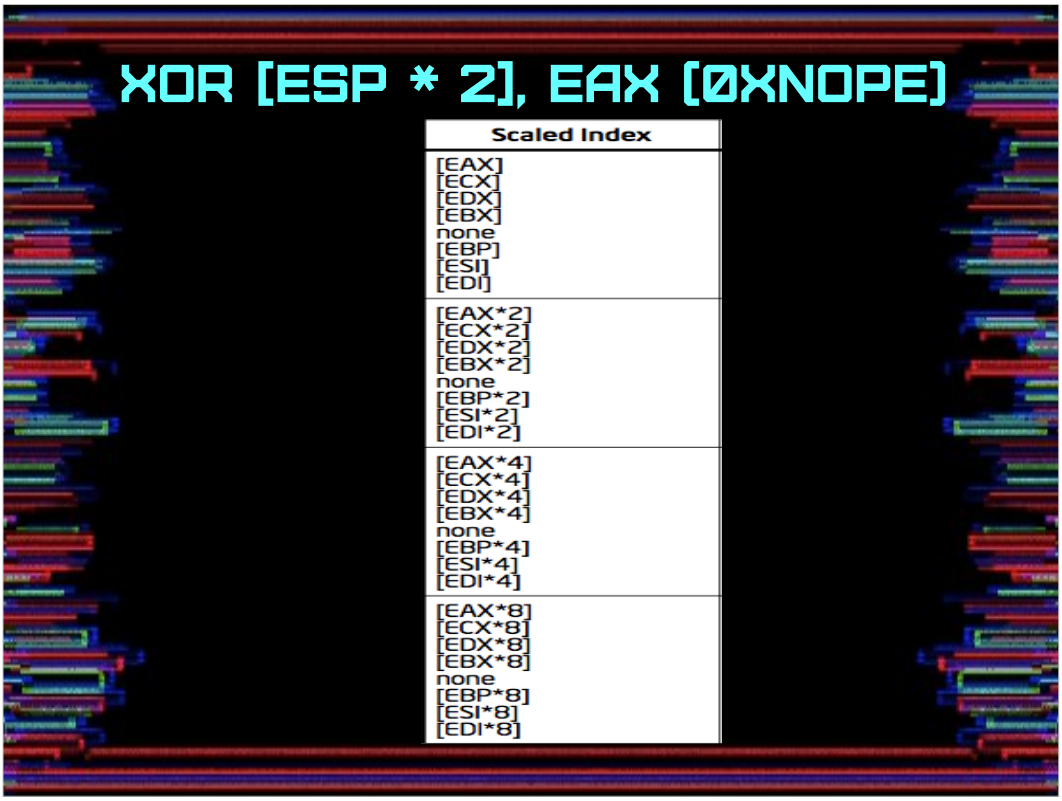
r32 (In decimal) Base = (In binary) Base =			EAX 0 000	ECX 1 001	EDX 2 010	EBX 3 011	ESP 4 100	[*] 5 101	ESI 6 110	EDI 7 111
Scaled Index	**SS**	**Index**	**Value of SIB Byte (in Hexadecimal)**							
[EAX]	00	000	00	01	02	03	04	05	06	07
[ECX]		001	08	09	0A	0B	0C	0D	0E	0F
[EDX]		010	10	11	12	13	14	15	16	17
[EBX]		011	18	19	1A	1B	1C	1D	1E	1F
none		100	20	21	22	23	24	25	26	27
[EBP]		101	28	29	2A	2B	2C	2D	2E	2F
[ESI]		110	30	31	32	33	34	35	36	37
[EDI]		111	38	39	3A	3B	3C	3D	3E	3F
[EAX*2]	01	000	40	41	42	43	44	45	46	47
[ECX*2]		001	48	49	4A	4B	4C	4D	4E	4F
[EDX*2]		010	50	51	52	53	54	55	56	57
[EBX*2]		011	58	59	5A	5B	5C	5D	5E	5F
none		100	60	61	62	63	64	65	66	67
[EBP*2]		101	68	69	6A	6B	6C	6D	6E	6F
[ESI*2]		110	70	71	72	73	74	75	76	77
[EDI*2]		111	78	79	7A	7B	7C	7D	7E	7F
[EAX*4]	10	000	80	81	82	83	84	85	86	87
[ECX*4]		001	88	89	8A	8B	8C	8D	8E	8F
[EDX*4]		010	90	91	92	93	94	95	96	97
[EBX*4]		011	98	99	9A	9B	9C	9D	9E	9F
none		100	A0	A1	A2	A3	A4	A5	A6	A7
[EBP*4]		101	A8	A9	AA	AB	AC	AD	AE	AF
[ESI*4]		110	B0	B1	B2	B3	B4	B5	B6	B7
[EDI*4]		111	B8	B9	BA	BB	BC	BD	BE	BF
[EAX*8]	11	000	C0	C1	C2	C3	C4	C5	C6	C7
[ECX*8]		001	C8	C9	CA	CB	CC	CD	CE	CF
[EDX*8]		010	D0	D1	D2	D3	D4	D5	D6	D7
[EBX*8]		011	D8	D9	DA	DB	DC	DD	DE	DF
none		100	E0	E1	E2	E3	E4	E5	E6	E7
[EBP*8]		101	E8	E9	EA	EB	EC	ED	EE	EF
[ESI*8]		110	F0	F1	F2	F3	F4	F5	F6	F7
[EDI*8]		111	F8	F9	FA	FB	FC	FD	FE	FF

In the last example, we needed to use the 'none' field in the SIB byte because ESP wasn't an option for the base register. However, we can still use this ignorance when the base register is already an option in the MODR/M table.

In this slide, we are showing that we are using this encoding with EAX. Keep in mind that we can still use any of the 4 'none' bytes



In this screenshot we first see how an assembler 'should' encode XOR [EAX], EAX. The last 4 instructions are the various ways we can encode it with the pointless 'none's in the SIB byte



What's the exception to use ESP as a Scaled register? as we didn't notice it as an option in the SIB byte encodings. It's because you can't. You try to write this above instruction and your assembler will give you an error and make you feel bad.

XOR [EBP + EAX * 2], EAX [0X31444500]

Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte

r8(/r)			AL	CL	DL	BL	AH	CH	DH	BH
r16(/r)			AX	CX	DX	BX	SP	BP	SI	DI
r32(/r)			EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
mm(/r)			MM0	MM1	MM2	MM3	MM4	MM5	MM6	MM7
xmm(/r)			XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7
(In decimal) /digit (Opcode)			0	1	2	3	4	5	6	7
(In binary) REG =			000	001	010	011	100	101	110	111
Effective Address	**Mod**	**R/M**	**Value of ModR/M Byte (in Hexadecimal)**							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[--][--][1]		100	04	0C	14	1C	24	2C	34	3C
disp32[2]		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
[EAX]+disp8[3]	01	000	40	48	50	58	60	68	70	78
[ECX]+disp8		001	41	49	51	59	61	69	71	79
[EDX]+disp8		010	42	4A	52	5A	62	6A	72	7A
[EBX]+disp8		011	43	4B	53	5B	63	6B	73	7B
[--][--]+disp8		100	44	4C	54	5C	64	6C	74	7C
[EBP]+disp8		101	45	4D	55	5D	65	6D	75	7D
[ESI]+disp8		110	46	4E	56	5E	66	6E	76	7E
[EDI]+disp8		111	47	4F	57	5F	67	6F	77	7F
[EAX]+disp32	10	000	80	88	90	98	A0	A8	B0	B8
[ECX]+disp32		001	81	89	91	99	A1	A9	B1	B9
[EDX]+disp32		010	82	8A	92	9A	A2	AA	B2	BA
[EBX]+disp32		011	83	8B	93	9B	A3	AB	B3	BB
[--][--]+disp32		100	84	8C	94	9C	A4	AC	B4	BC
[EBP]+disp32		101	85	8D	95	9D	A5	AD	B5	BD
[ESI]+disp32		110	86	8E	96	9E	A6	AE	B6	BE
[EDI]+disp32		111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL/MM/XMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH/MM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

This instruction has a base register of EBP and a scaled register of EAX * 2. Vertically aligning the 2nd operand of EAX is easy. Since we are using a scaled register, we need to find the appropriate [--][--] line horizontally.

One would think that we would pick 0x04, but that is not the case, we need to pick 0x44 due to some EBP base register complications in the SIB byte that we are about to explore on the next slide

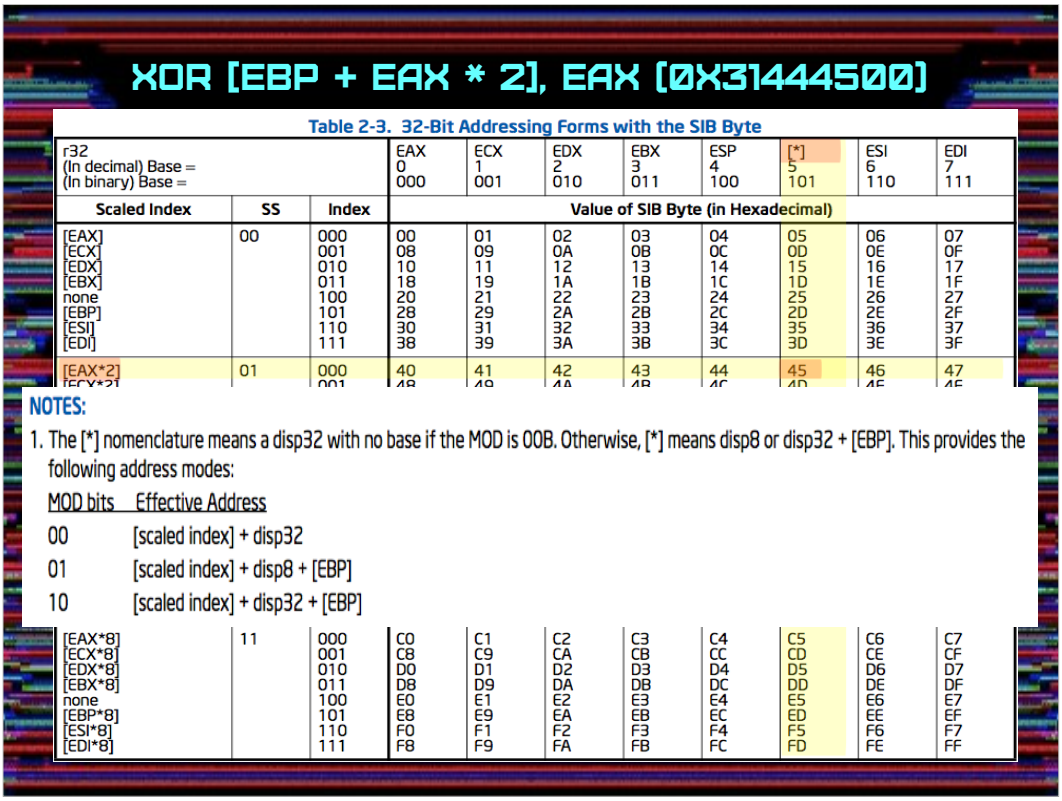


Table 2-3. 32-Bit Addressing Forms with the SIB Byte

r32 (In decimal) Base = (In binary) Base =			EAX 0 000	ECX 1 001	EDX 2 010	EBX 3 011	ESP 4 100	[*] 5 101	ESI 6 110	EDI 7 111
Scaled Index	**SS**	**Index**	**Value of SIB Byte (in Hexadecimal)**							
[EAX]	00	000	00	01	02	03	04	05	06	07
[ECX]		001	08	09	0A	0B	0C	0D	0E	0F
[EDX]		010	10	11	12	13	14	15	16	17
[EBX]		011	18	19	1A	1B	1C	1D	1E	1F
none		100	20	21	22	23	24	25	26	27
[EBP]		101	28	29	2A	2B	2C	2D	2E	2F
[ESI]		110	30	31	32	33	34	35	36	37
[EDI]		111	38	39	3A	3B	3C	3D	3E	3F
[EAX*2]	01	000	40	41	42	43	44	45	46	47
[ECX*2]		001	48	49	4A	4B	4C	4D	4E	4F
[EDX*2]		010	50	51	52	53	54	55	56	57
[EBX*2]		011	58	59	5A	5B	5C	5D	5E	5F
none		100	60	61	62	63	64	65	66	67
[EBP*2]		101	68	69	6A	6B	6C	6D	6E	6F
[ESI*2]		110	70	71	72	73	74	75	76	77
[EDI*2]		111	78	79	7A	7B	7C	7D	7E	7F
[EAX*4]	10	000	80	81	82	83	84	85	86	87
[ECX*4]		001	88	89	8A	8B	8C	8D	8E	8F
[EDX*4]		010	90	91	92	93	94	95	96	97
[EBX*4]		011	98	99	9A	9B	9C	9D	9E	9F
none		100	A0	A1	A2	A3	A4	A5	A6	A7
[EBP*4]		101	A8	A9	AA	AB	AC	AD	AE	AF
[ESI*4]		110	B0	B1	B2	B3	B4	B5	B6	B7
[EDI*4]		111	B8	B9	BA	BB	BC	BD	BE	BF
[EAX*8]	11	000	C0	C1	C2	C3	C4	C5	C6	C7
[ECX*8]		001	C8	C9	CA	CB	CC	CD	CE	CF
[EDX*8]		010	D0	D1	D2	D3	D4	D5	D6	D7
[EBX*8]		011	D8	D9	DA	DB	DC	DD	DE	DF
none		100	E0	E1	E2	E3	E4	E5	E6	E7
[EBP*8]		101	E8	E9	EA	EB	EC	ED	EE	EF
[ESI*8]		110	F0	F1	F2	F3	F4	F5	F6	F7
[EDI*8]		111	F8	F9	FA	FB	FC	FD	FE	FF

NOTES:

1. The [*] nomenclature means a disp32 with no base if the MOD is 00B. Otherwise, [*] means disp8 or disp32 + [EBP]. This provides the following address modes:

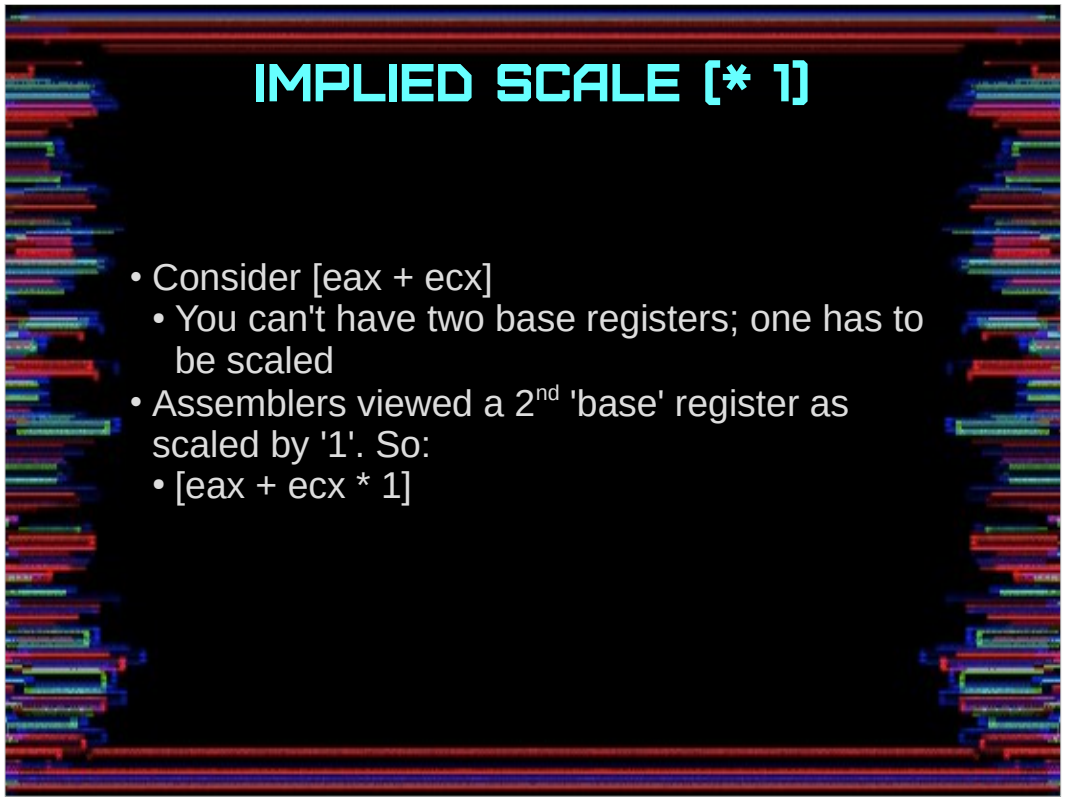
MOD bits	Effective Address
00	[scaled index] + disp32
01	[scaled index] + disp8 + [EBP]
10	[scaled index] + disp32 + [EBP]

Lining up the horizontal line for the scaled register of EAX * 2 is straight forward. However, we don't find an obvious EBP base register on the vertical line. It's the [*] line that actually gives us what we need.

The [*] line is dependent on the displacement option we pick from the MODR/M byte. There are only 3 variations; no displacement, 8-bit displacement, and 32-bit displacement. The results are as follows:
No displacement = [ScaledReg * n + 0x11223344]
Disp8 = [EBP + ScaledReg * n +0x11]
Disp32 = [EBP + ScaledReg * n +0x11223344]

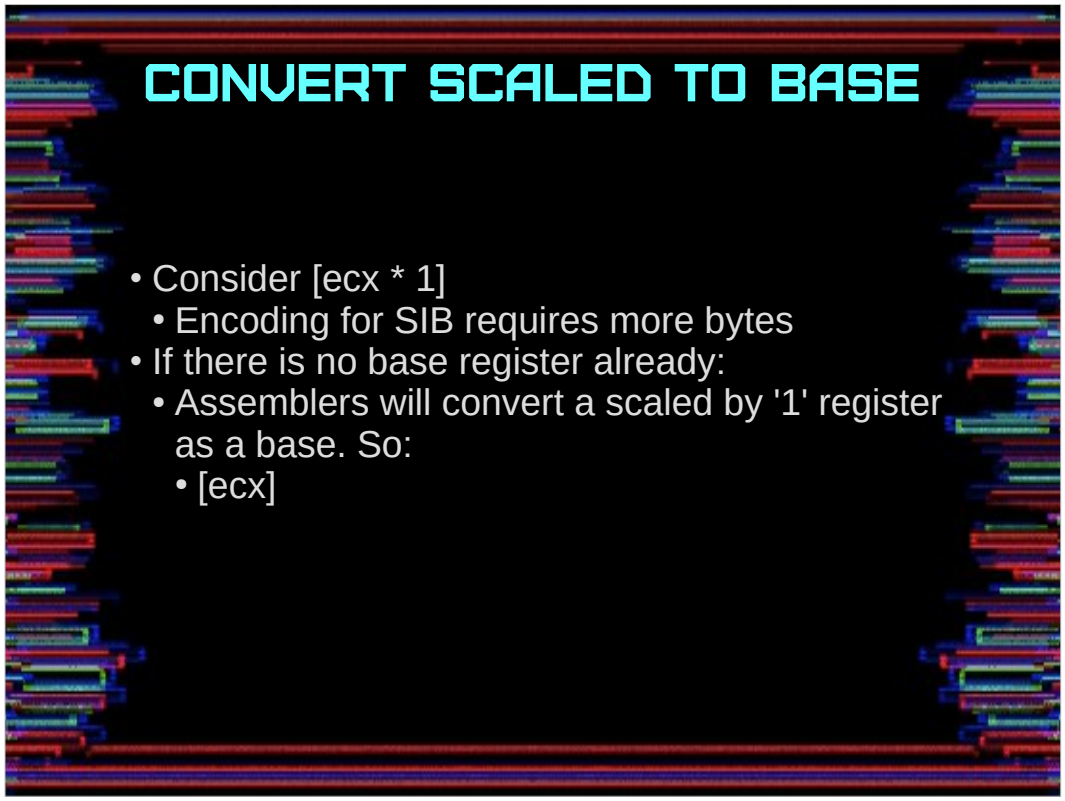
Either of the last 2 options would technically work, but we chose the 8-bit displacement option because it would get encoded in with 3 less bytes.

So finally, we arrive at the 0x45 byte in our table. However, we aren't done until we actually put the 0x00 byte at the end, because this is our 'invisible' displacement This means that our assembly would more literally be interpreted as such: XOR [EBP + EAX * 2 + 0x00], EAX



There are things we take for granted when only writing in a high level language like assembly. If you type a pointer like [eax + ecx], the thing to consider is that there can only be one base register.

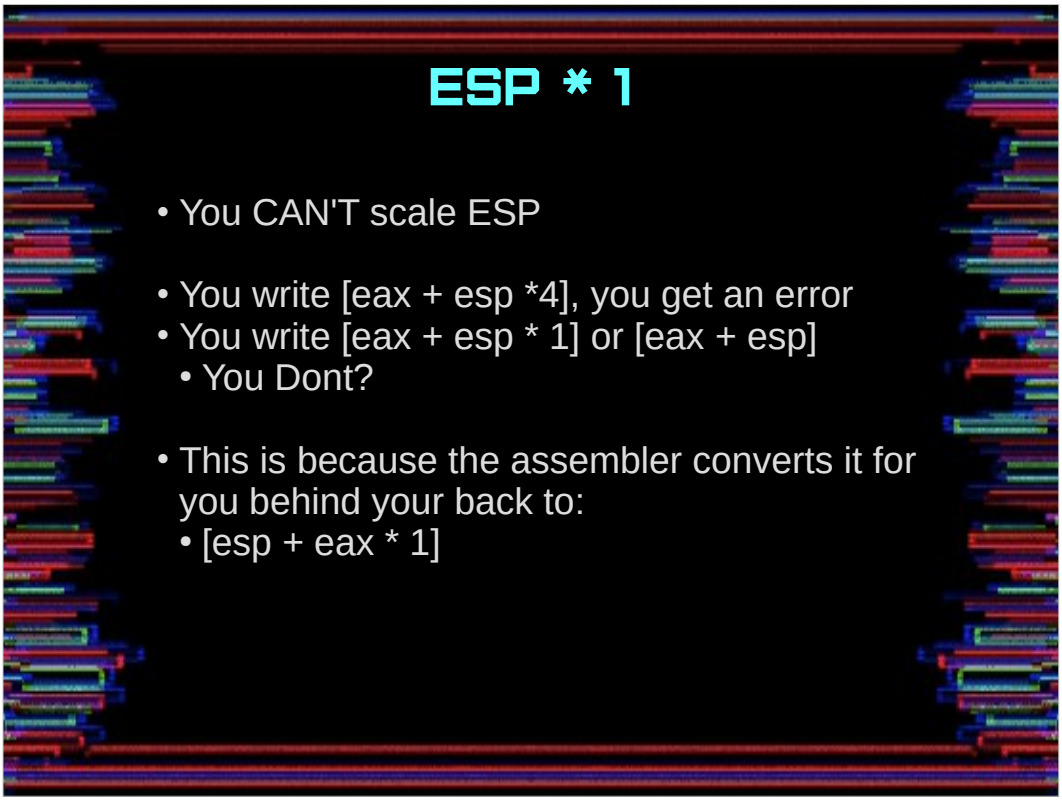
An assembler (like nasm) is going to look to your 2nd register to encode as the scaled register; the assembler will treat [eax + ecx] more literally as [eax + ecx * 1]. Or it will make ecx the scaled register and scale it by 1.



It's one thing to have something like [ecx * 4]. It is unambiguous: there is no base register and we need a scaled register of ecx * 4.

[ecx * 1] on the other hand, assemblers don't do what you asked for here. If you don't pick a base register, and your scaled register is scaled by one, your assembler is just going to make it the base register.

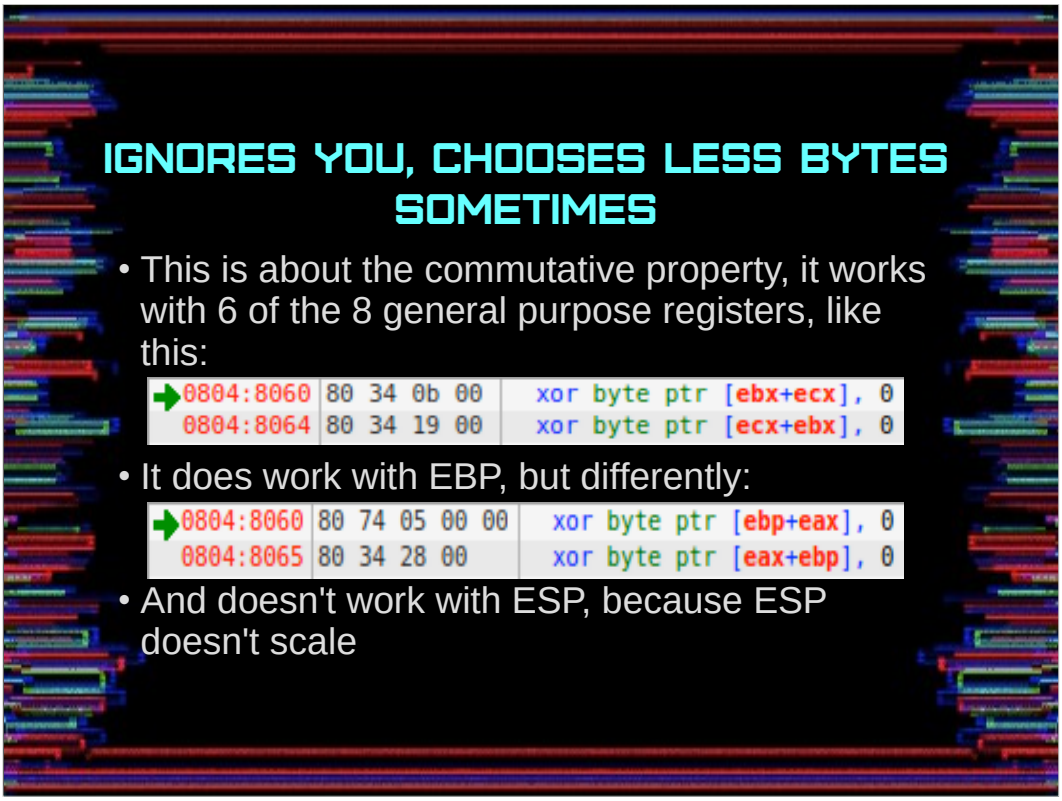
My instinct is to get annoyed with this, as my assembly is being interpreted into machine code that I didn't intend for, as I would have and could have written [ecx] if that's what I wanted. The reason an assembler is going to choose this because it takes less bytes to encode (because it doesn't need the SIB byte).



So we know that we can't use ESP as the scaled register. This is why if we write something like [eax + esp * 4] we will get an error. But why do we not get an error if we write [eax + esp * 1]?

Well, if you were to assemble this and then disassemble it, you would discover that your assembler actually writes this as [esp + eax * 1].

In other words, if esp is scaled by only one, and the base register itself is not also esp, it will make the base register the scaled one so esp can join back in as the base. It logically does the same thing.



Speaking of swapping around the registers, this is the commutative property in mathematics (because addition). We can do this no problem with eax, ecx, edx, ebx, esi, and edi.

esp is a register that can't be swapped, because of its scaling issues as previously discussed.

We also discussed the trade-off that needs to be made when using ebp in the SIB byte, so we do this at the cost of having to add the extra disp8 null.

However, the most interesting part of this is that if you use [ebp+eax] in your assembly, it will take you literally If it did [eax + ebp] (logically the same), it would actually take 1 less byte to encode, but it doesn't opt for less machine code in this case. Just goes to show that sometimes an assembler optimizes for this kind of stuff, but not always

PUT A NULL IN IT

- If a pointer doesn't have a displacement, then put in a displacement of 0x00...same difference right

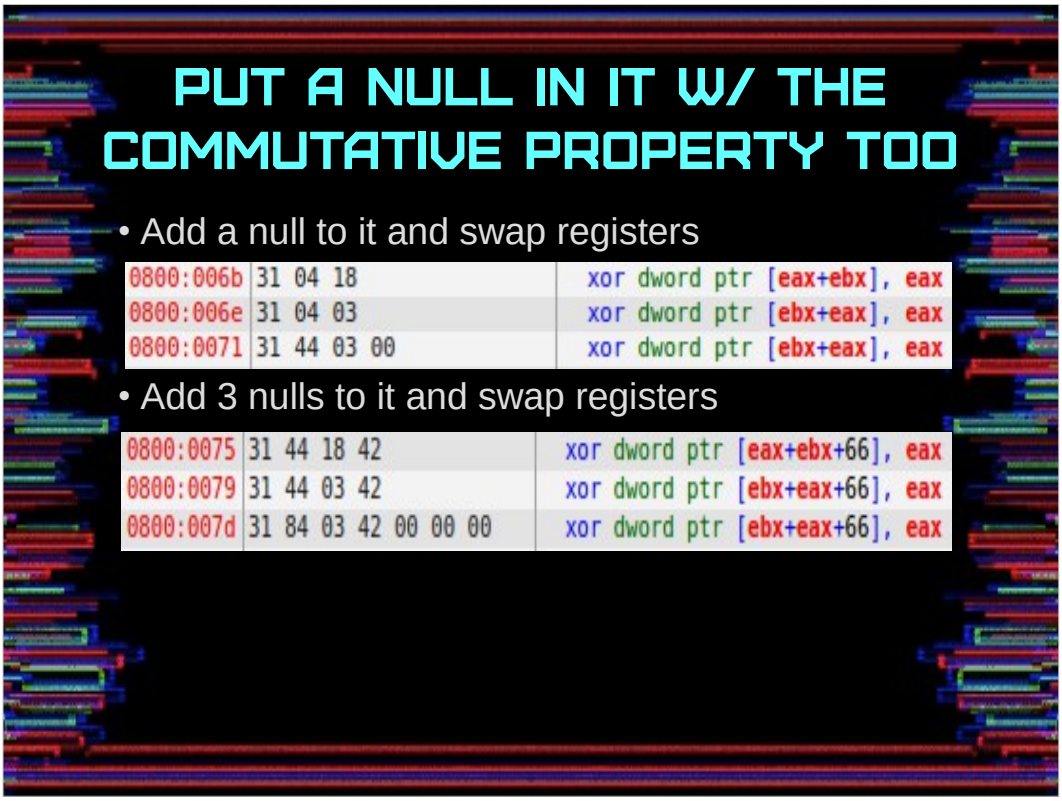
```
0800:0060 31 04 18          xor dword ptr [eax+ebx], eax
0800:0063 31 44 18 00       xor dword ptr [eax+ebx], eax
```

- If there's an 8 bit displacement, make it a 32 bit displacement with 3 bytes of leading nulls

```
0800:0067 31 44 18 42       xor dword ptr [eax+ebx+66], eax
0800:006b 31 84 18 42 00 00 00  xor dword ptr [eax+ebx+66], eax
```

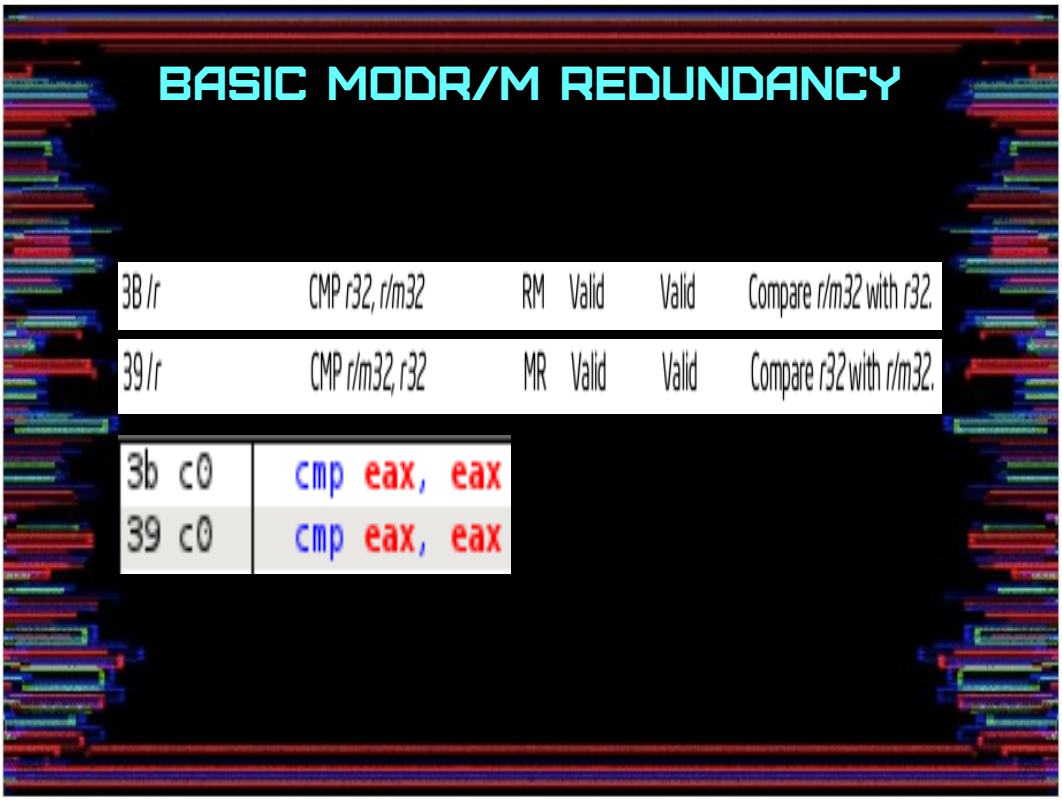
For instructions that don't already have displacements, there's nothing from stopping us from being a troll and adding a displacement of nothing (0x00). We can add an 8-bit or a 32-bit displacement with nothing in it and the memory pointer would be logically the same.

Additionally, if we have an 8-bit displacement, we can 'upgrade' it to 32-bit by padding 3 null bytes in front of it.



Of course you can get creative and mix and match these redundancies.

This slide shows us mixing the 'null upgrade' with the commutative property



This redundancy works because x86 generally has no instructions that allow for both operands to be a memory location in the same instruction.

For instance, if your instruction was 'mov', you could move a value of a register into a memory location, you could also move the value in a memory location into a register, but you could never move the value of a memory location into another memory location (with only one instruction).

Because of this, you need an encoding for each scenario. However, the operand that allows for a memory pointer also allows for it to just be a register as well (allowing register to register).

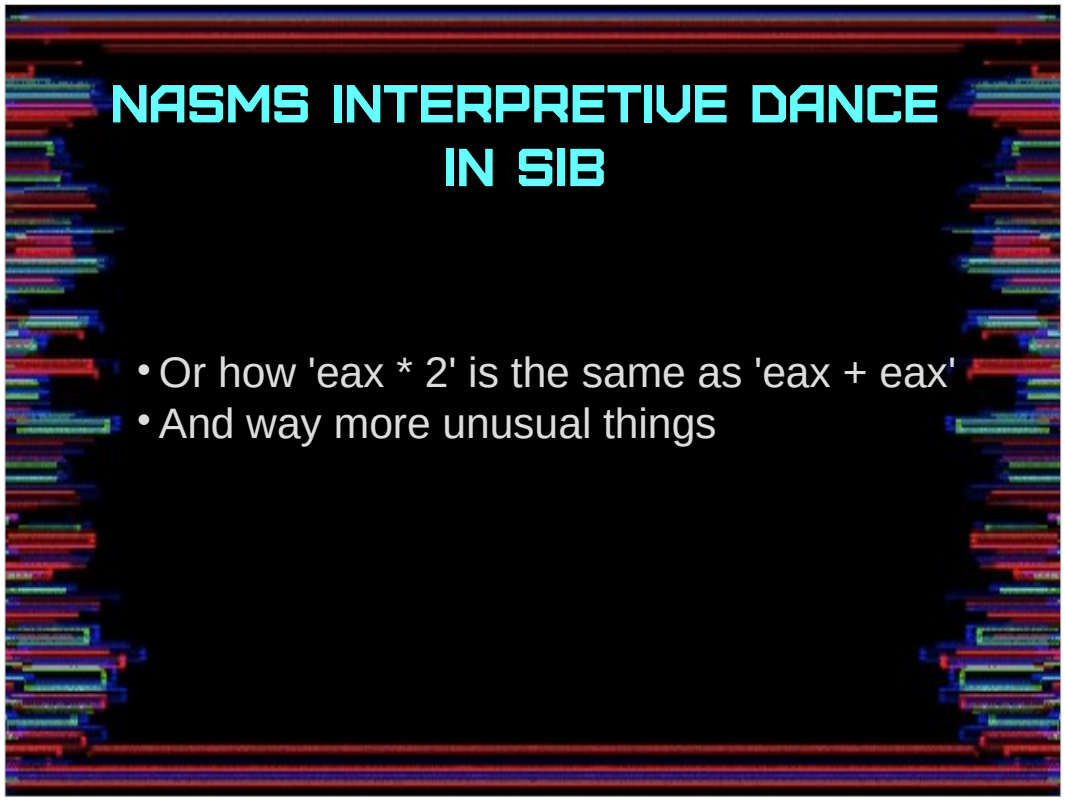
This means that both encodings allow for register to register. This is where the redundancy comes into play and why we can see something like the above screenshot.

BASIC MODR/M REDUNDANCY

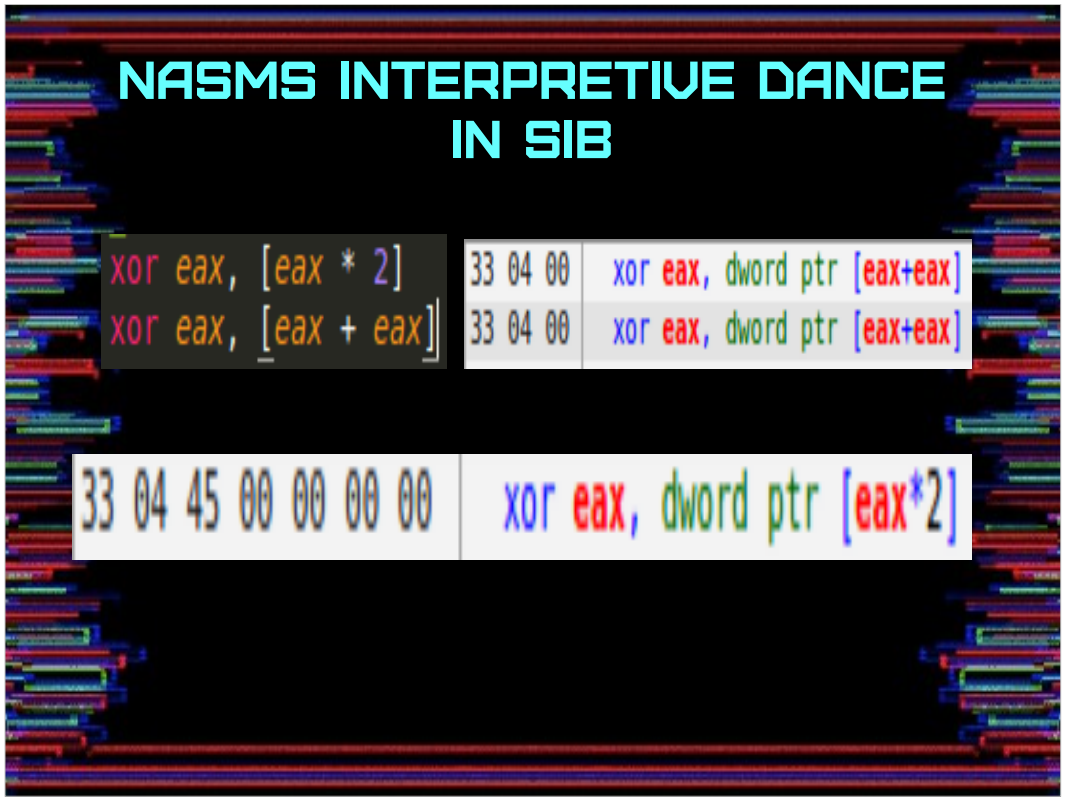
Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte

r8(/r) r16(/r) r32(/r) mm(/r) xmm(/r) (In decimal) /digit (Opcode) (In binary) REG =			AL AX EAX MM0 XMM0 0 000	CL CX ECX MM1 XMM1 1 001	DL DX EDX MM2 XMM2 2 010	BL BX EBX MM3 XMM3 3 011	AH SP ESP MM4 XMM4 4 100	CH BP EBP MM5 XMM5 5 101	DH SI ESI MM6 XMM6 6 110	BH DI EDI MM7 XMM7 7 111
Effective Address	**Mod**	**R/M**	**Value of ModR/M Byte (in Hexadecimal)**							
[EAX] [ECX] [EDX] [EBX] [--][--][1] disp32[2] [ESI] [EDI]	00	000 001 010 011 100 101 110 111	00 01 02 03 04 05 06 07	08 09 0A 0B 0C 0D 0E 0F	10 11 12 13 14 15 16 17	18 19 1A 1B 1C 1D 1E 1F	20 21 22 23 24 25 26 27	28 29 2A 2B 2C 2D 2E 2F	30 31 32 33 34 35 36 37	38 39 3A 3B 3C 3D 3E 3F
[EAX]+disp8[3] [ECX]+disp8 [EDX]+disp8 [EBX]+disp8 [--][--]+disp8 [EBP]+disp8 [ESI]+disp8 [EDI]+disp8	01	000 001 010 011 100 101 110 111	40 41 42 43 44 45 46 47	48 49 4A 4B 4C 4D 4E 4F	50 51 52 53 54 55 56 57	58 59 5A 5B 5C 5D 5E 5F	60 61 62 63 64 65 66 67	68 69 6A 6B 6C 6D 6E 6F	70 71 72 73 74 75 76 77	78 79 7A 7B 7C 7D 7E 7F
[EAX]+disp32 [ECX]+disp32 [EDX]+disp32 [EBX]+disp32 [--][--]+disp32 [EBP]+disp32 [ESI]+disp32 [EDI]+disp32	10	000 001 010 011 100 101 110 111	80 81 82 83 84 85 86 87	88 89 8A 8B 8C 8D 8E 8F	90 91 92 93 94 95 96 97	98 99 9A 9B 9C 9D 9E 9F	A0 A1 A2 A3 A4 A5 A6 A7	A8 A9 AA AB AC AD AE AF	B0 B1 B2 B3 B4 B5 B6 B7	B8 B9 BA BB BC BD BE BF
EAX/AX/AL/MM0/XMM0 ECX/CX/CL/MM/XMM1 EDX/DX/DL/MM2/XMM2 EBX/BX/BL/MM3/XMM3 ESP/SP/AH/MM4/XMM4 EBP/BP/CH/MM5/XMM5 ESI/SI/DH/MM6/XMM6 EDI/DI/BH/MM7/XMM7	11	000 001 010 011 100 101 110 111	C0 C1 C2 C3 C4 C5 C6 C7	C8 C9 CA CB CC CD CE CF	D0 D1 D2 D3 D4 D5 D6 D7	D8 D9 DA DB DC DD DE DF	E0 E1 E2 E3 E4 E5 E6 E7	E8 E9 EA EB EC ED EE EF	F0 F1 F2 F3 F4 F5 F6 F7	F8 F9 FA FB FC FD FE FF

In the previous slide it seemed like magic that we could just swap out the machine opcode and leave the operand data (0xC0) alone. This isn't always the case. With the different encodings, the vertical and horizontal parts of the table get swapped. But in the case of using the same register with itself, it's symmetric enough to not change the value in the table.



This is another byte saving optimization. The next slide will follow the maze of the MODR/M + SIB byte to find out why



So in the top 2 screenshots, we are comparing two different assembly instructions to the machine code nasm outputs on the right. Notably, both instructions are converted to the [eax + eax] form. It is logically the same as [eax * 2], what does nasm have against scaling eax?

It is because of the side effects of not having a base register when using SIB. You can have 'none' for a scaled register, but having 'none' (or [*]) for the base register comes at the cost of having to use a 32-bit displacement. This was covered a few slides back (the 3 options the [*] uses).

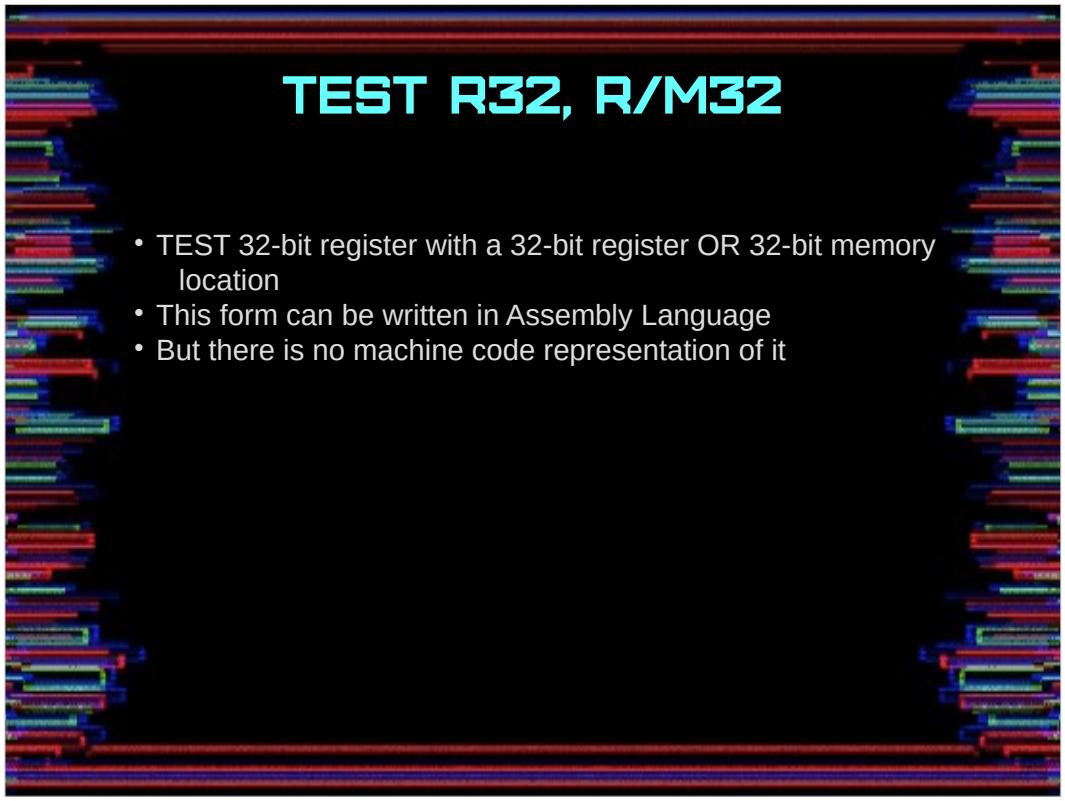
If we take [eax * 2] literally, it doubles our machine code for the instruction. Assemblers do not see this as ideal

But what's really interesting is what kind of bullshit assemblers like nasm will put up with.

First of all, there is no scale of * 5; only 1, 2, 4, and 8. But nasm is smart enough to look at this instruction and decide it is logically the same as eax + eax * 4

Finally, scaling by something non-existant is one thing, but there is no such thing as subtraction in our pointer format, but it is valid assembly to nasm. Nasm is smart enough to look at [eax * 2 – eax] and know that it is pretty much the same thing as just [eax]

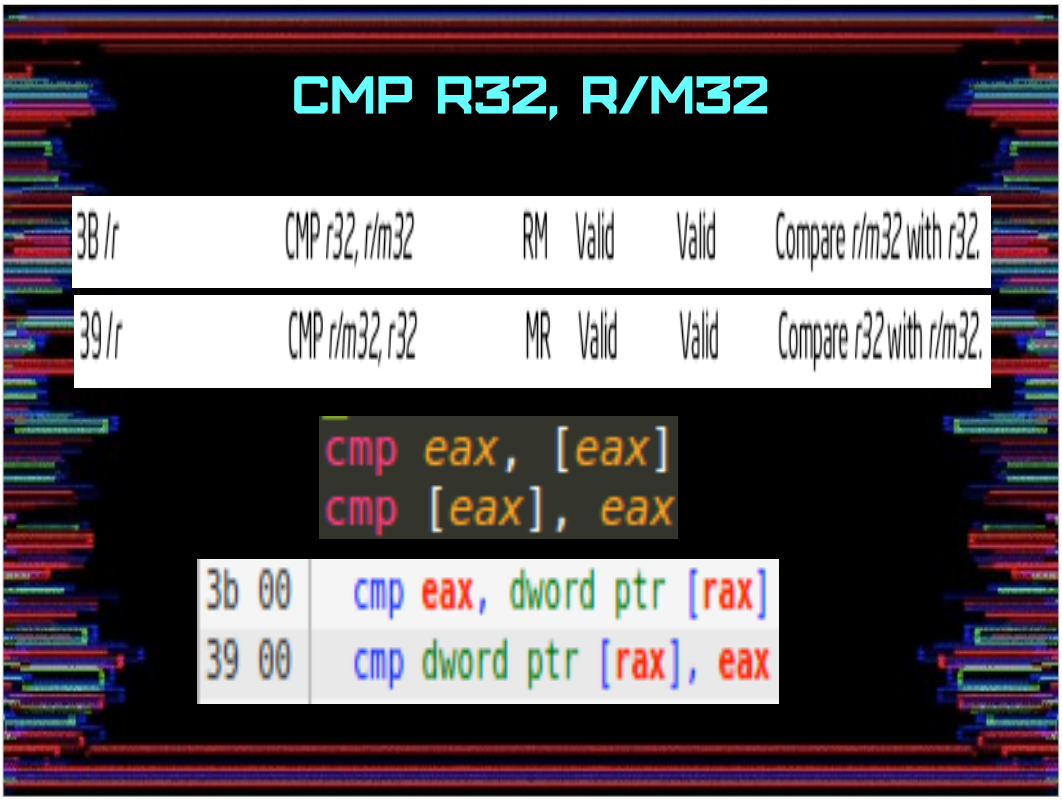
I love nasm



I like this one. This slide is saying that you can write something in assembly like: TEST EAX, [EAX]

The thing is, there is no machine encoding to represent this. We previously discussed how we needed more than one encoding to mitigate being able to use a pointer for the source or destination. So what's going on here?

We will explore in the next couple slides



This slide shows the two different encodings of the cmp instruction with 32bit operands.

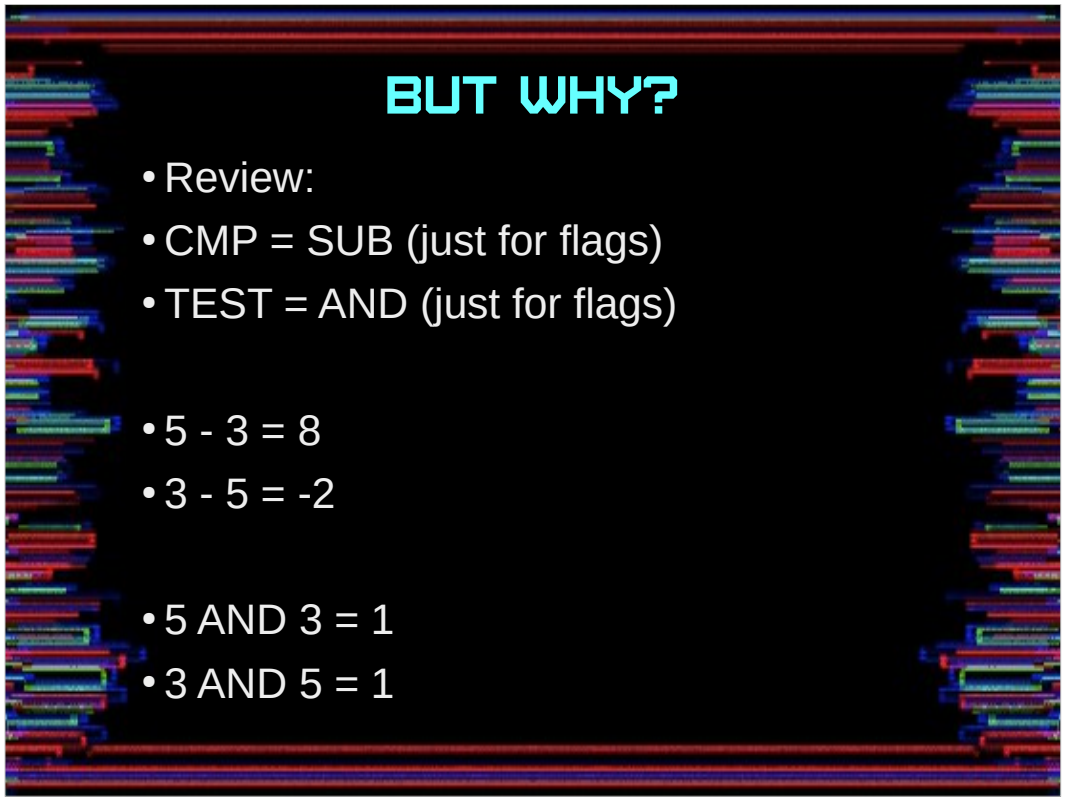
The last 2 screenshots compare the source assembly with the resulting machine-code in a debugger.

If we write the assembly shown on top, we get machine code comparable to the middle image.

What we see here is that the first instruction gets interpreted and converted by swapping the operands around to its only supported encoding. That is, Test r/m32, r32.

We see the encoding for this in the Intel manual (last image). Trust me, there is not corresponding encoding for the operands swapped around like other sane instructions.

So can we swap these operands and logically have the same results?



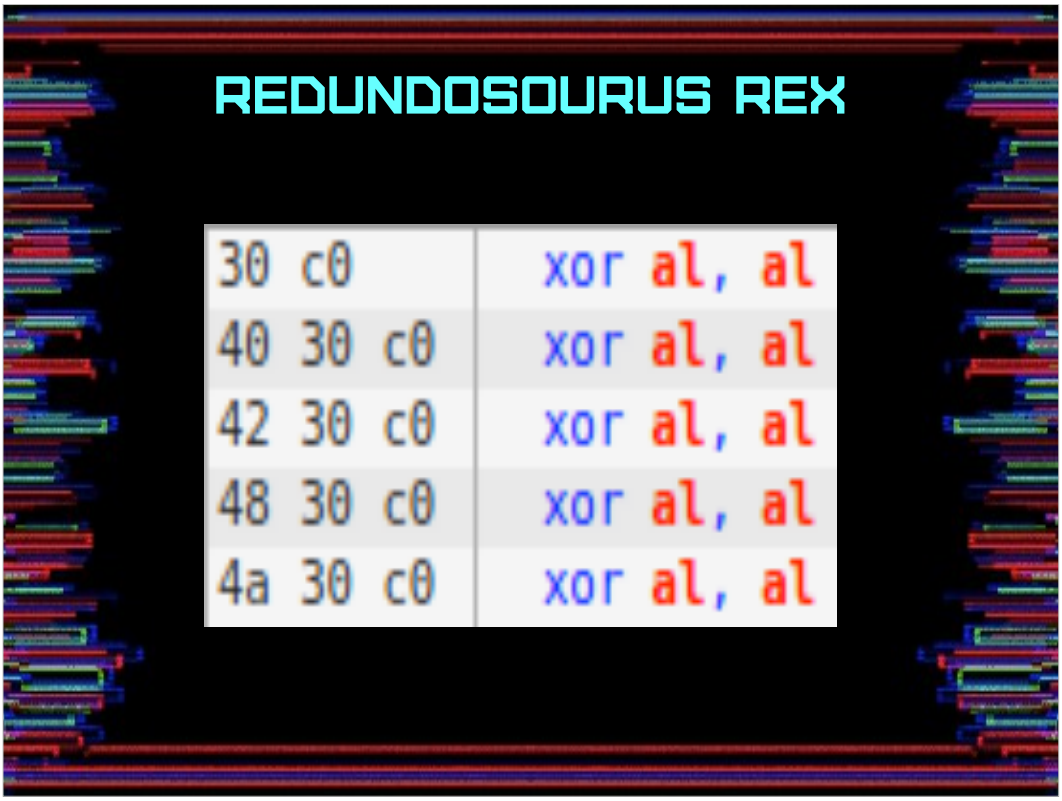
The answer is yes. We compare CMP and TEST to see why.

Both of these instructions act like a math/logic instruction but without storing the result; it just does the instruction for the side effect.

CMP is like subtraction and TEST is like a logical AND. CMP doesn't SUB though, nor does TEST do an AND. They just set the flags so conditional jumps can have more intelligent behavior

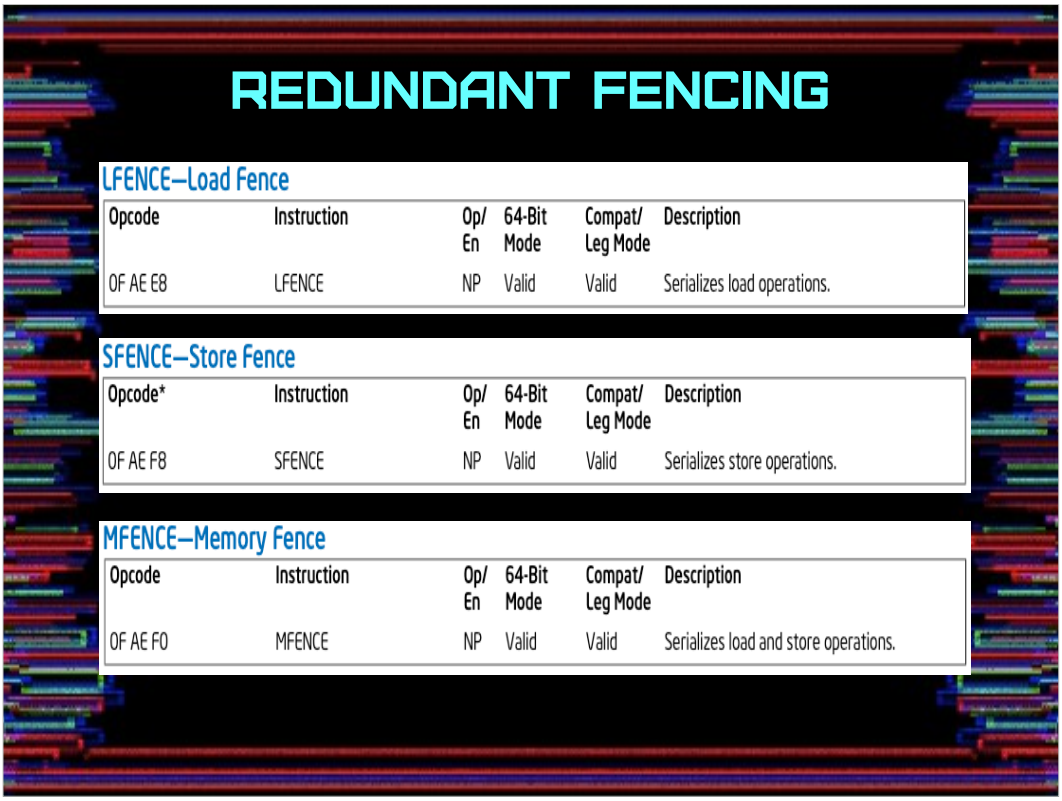
If you try to do some commutative stuff, you see subtraction obviously isn't commutative, swapping the operands gives you different results.

TEST (and AND) on the other hand are commutative, swapping the operands gives the same result. Therefor you only really do need one encoding to represent both orders. So assemblers look at your un-encodable instruction and converts it into something that does the same thing

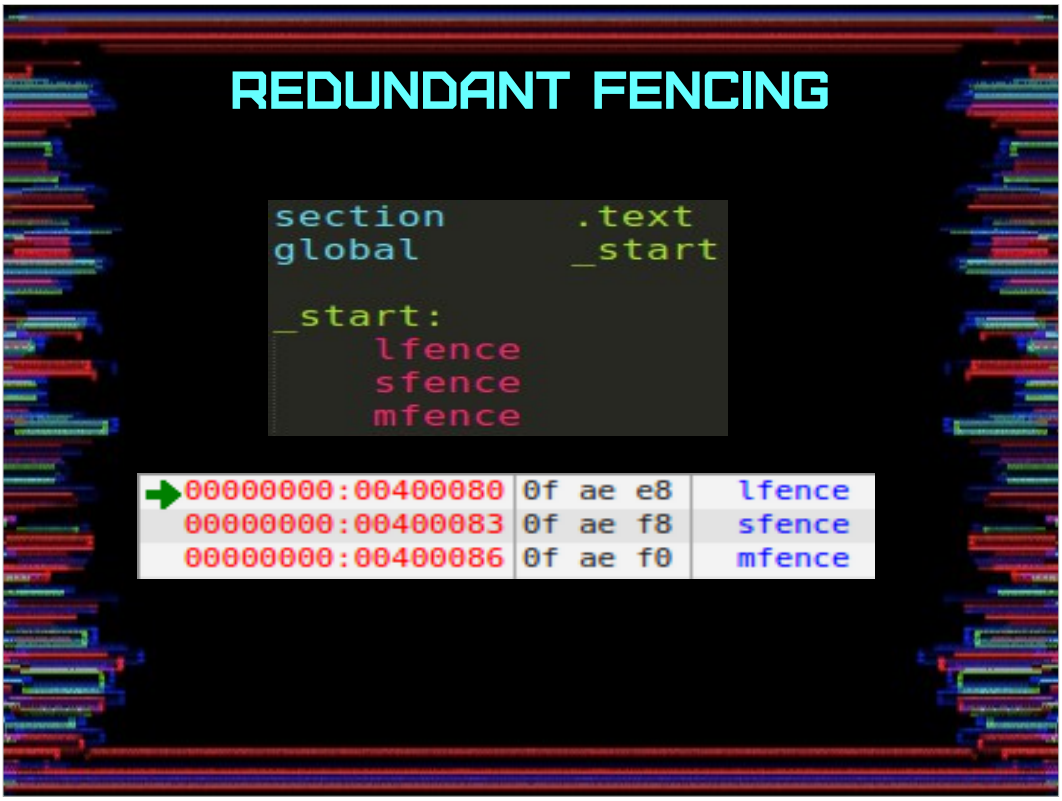


This is just a 64-bit prefix hack. In order to access all of the extra registers that come with 64 bit processors, but also remain backwards compatible, Intel chose to prefix instructions with a byte that would change what the registers end up being.

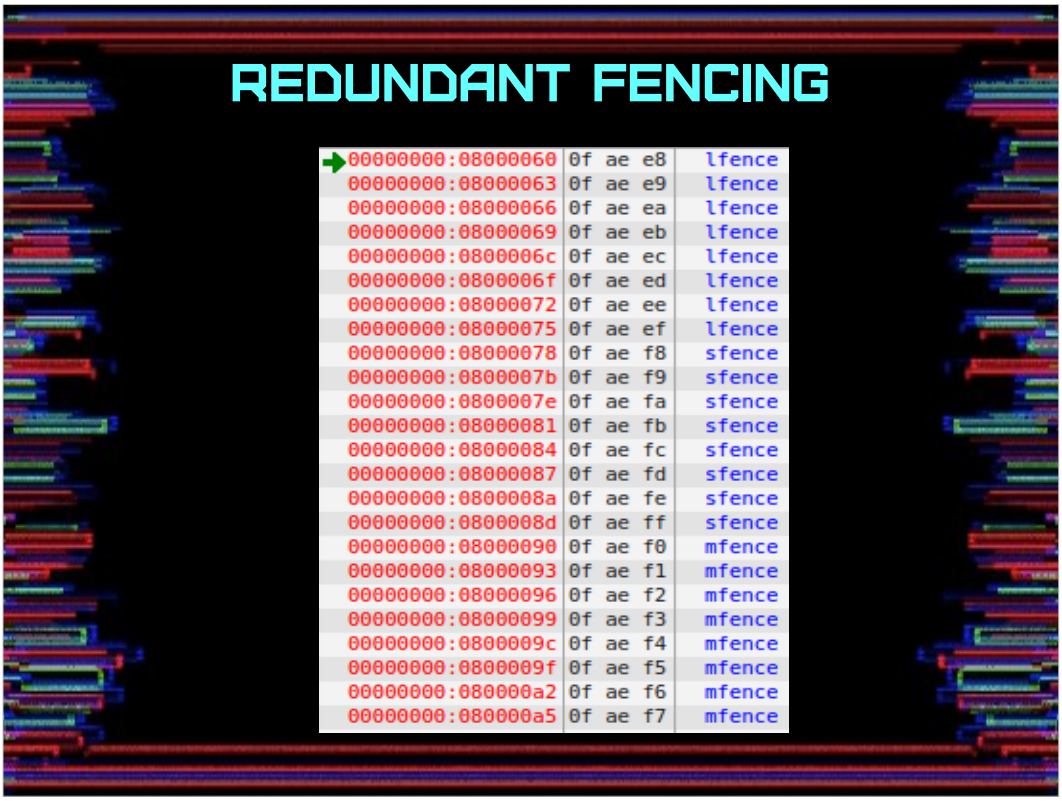
Of course, some of the old registers are also encodable with the prefixes, and of course there are many redundancies to this; as the image of this slide demonstrates.



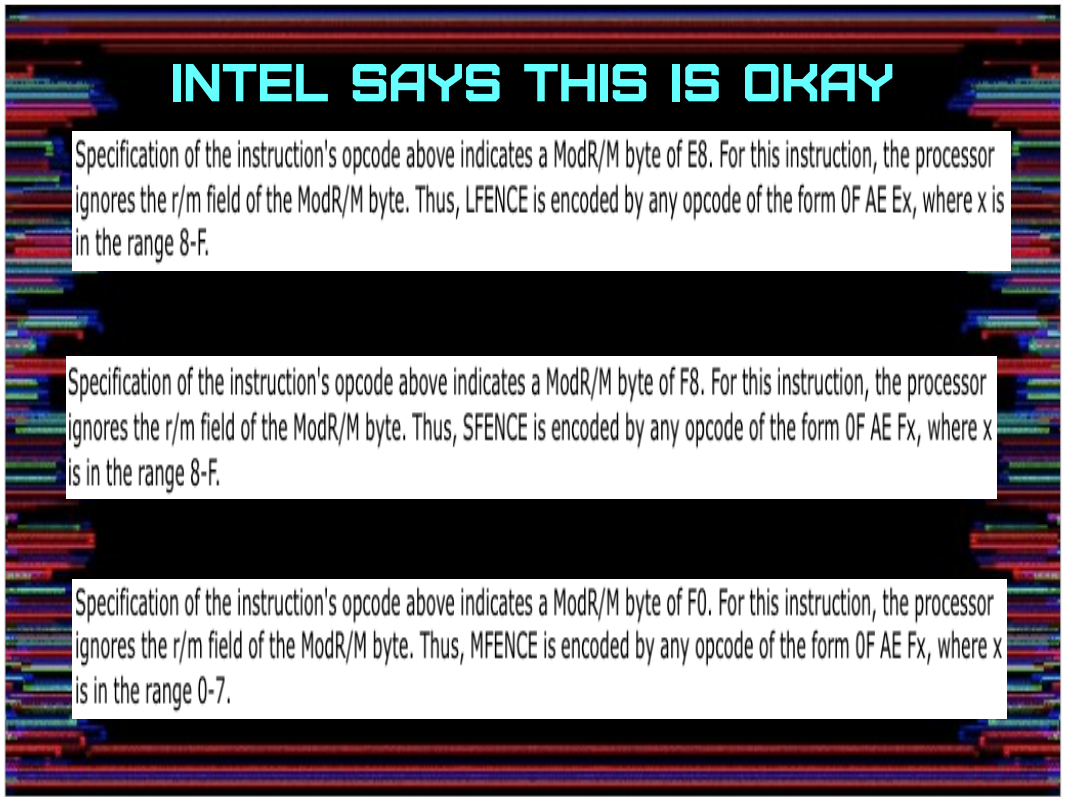
There are 3 different types of 'fence' instructions, each of them have the recommended machine code.



We can see that the suggested machine code is dutifully used when comparing the assembly source and the machine code output from the disassembly

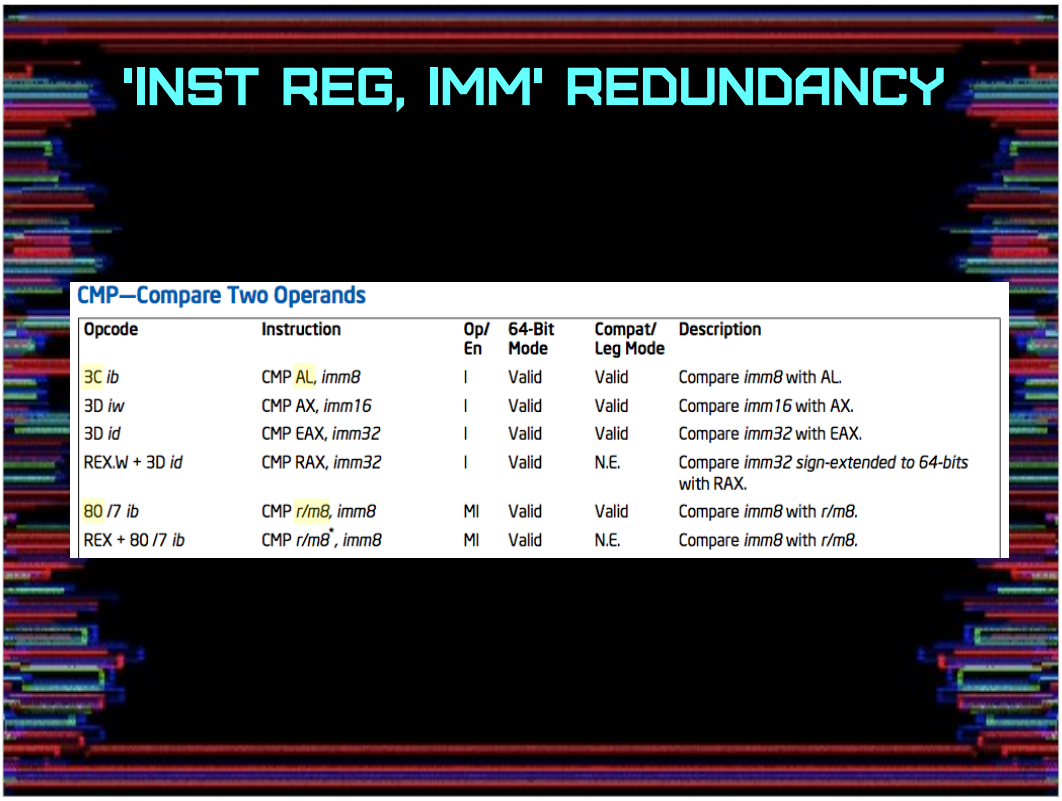


However, there is a lot of redundancy on this one. It so turns out that Intel suggests that this can be done with direct machine code. There's no real benefit to using any of these alternate encodings, however.



This is the part of the Intel manual that suggests you can use the extra 7 other end nibbles for these fence instructions.

'INST REG, IMM' REDUNDANCY

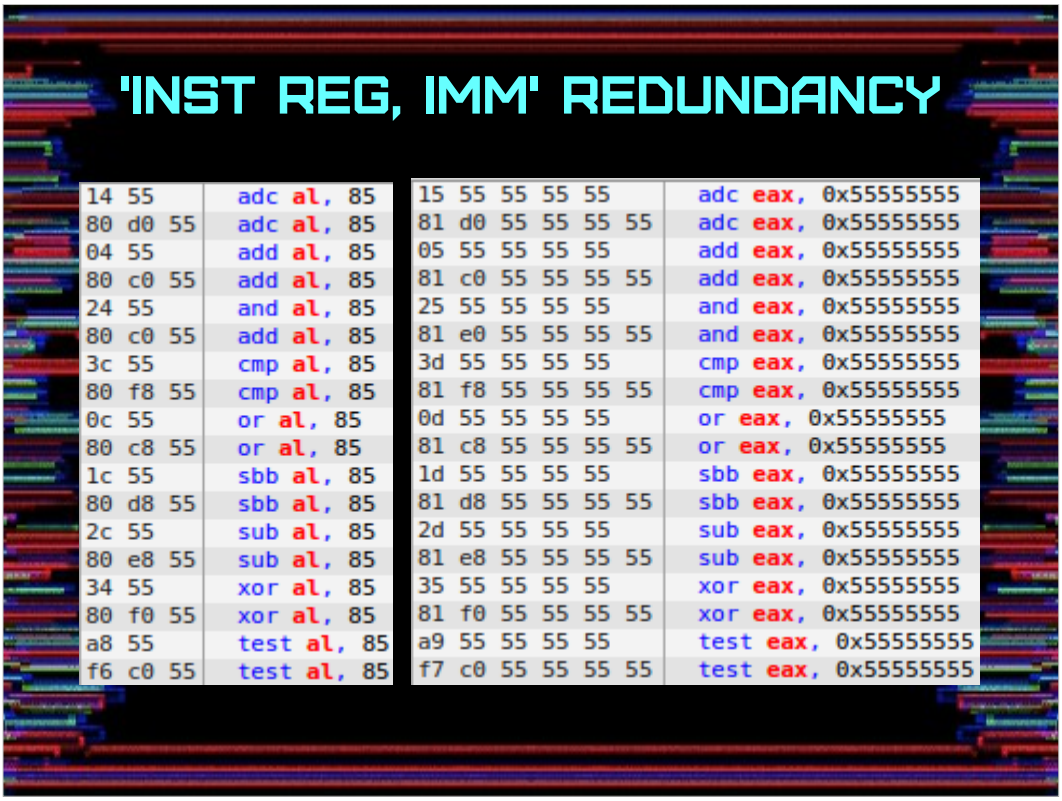


CMP—Compare Two Operands

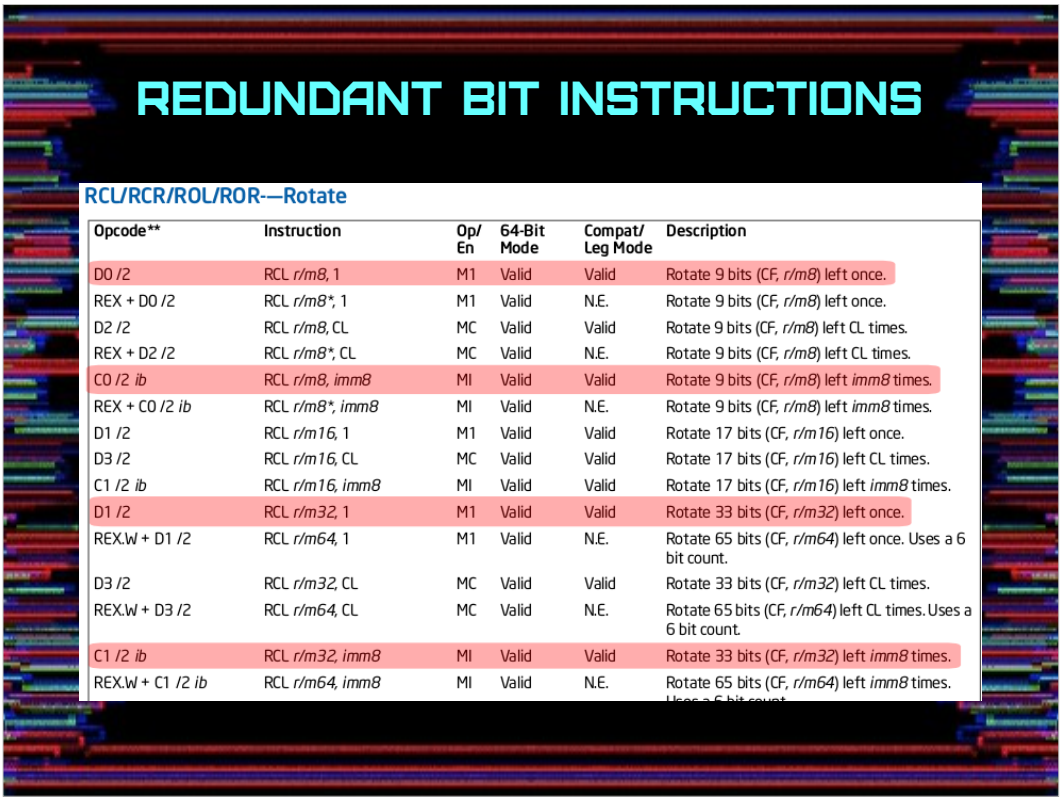
Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
3C *ib*	CMP AL, *imm8*	I	Valid	Valid	Compare *imm8* with AL.
3D *iw*	CMP AX, *imm16*	I	Valid	Valid	Compare *imm16* with AX.
3D *id*	CMP EAX, *imm32*	I	Valid	Valid	Compare *imm32* with EAX.
REX.W + 3D *id*	CMP RAX, *imm32*	I	Valid	N.E.	Compare *imm32 sign-extended to 64-bits* with RAX.
80 /7 *ib*	CMP *r/m8*, *imm8*	MI	Valid	Valid	Compare *imm8* with *r/m8*.
REX + 80 /7 *ib*	CMP *r/m8**, *imm8*	MI	Valid	N.E.	Compare *imm8* with *r/m8*.

In similar fashion to the very first redundancy explored in this presentation, there are many instructions that have an encoding for putting an immediate value into just the AL/AX/EAX register. This is because this register is so common, might as well have reduced machine code for it.

There is also the more generic encoding that allows for putting an immediate value into a MODR/M+SIB encodable operand. The redundancy comes in because AL/AX/EAX can be one of those options.

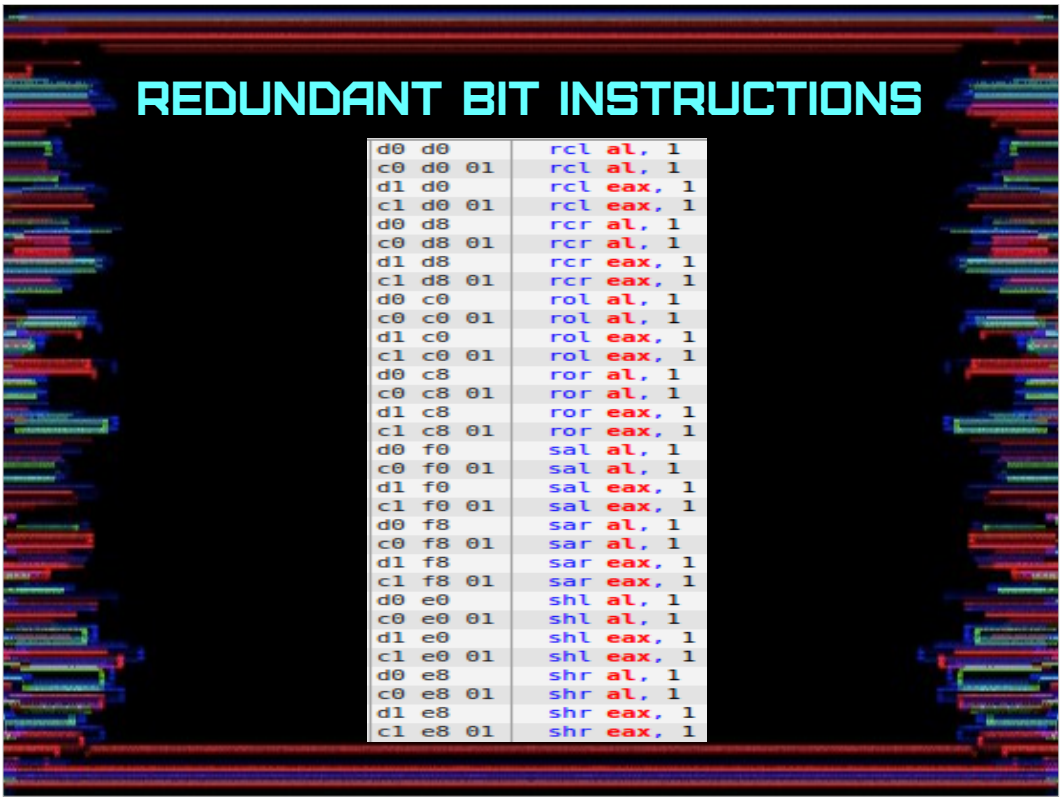


This slide shows all of those redundancies

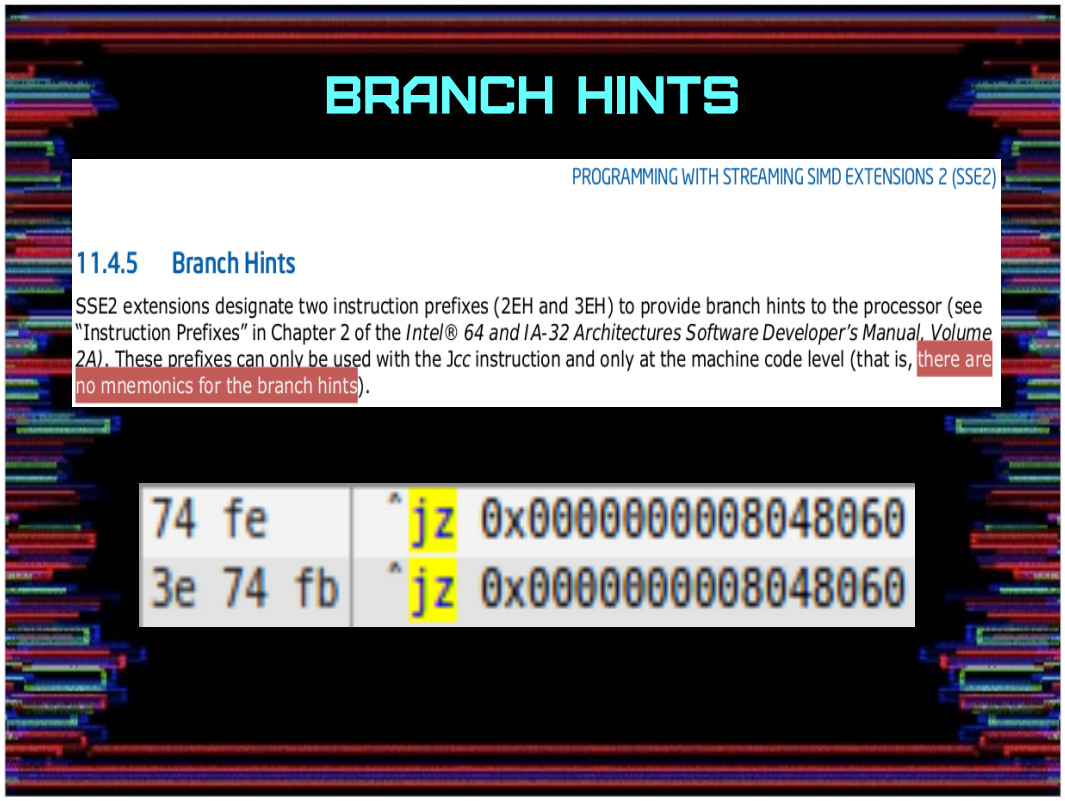


Speaking of doing something so common that Intel provides a direct smaller machine code encoding for it; bitwise instructions like rotating and shifting are often done by just one bit. Because of this, there's a shortcut to have the immediate operand be just '1'.

There is also the more generic 8-bit immediate operand. But obviously '1' is a valid value in this encoding as well.

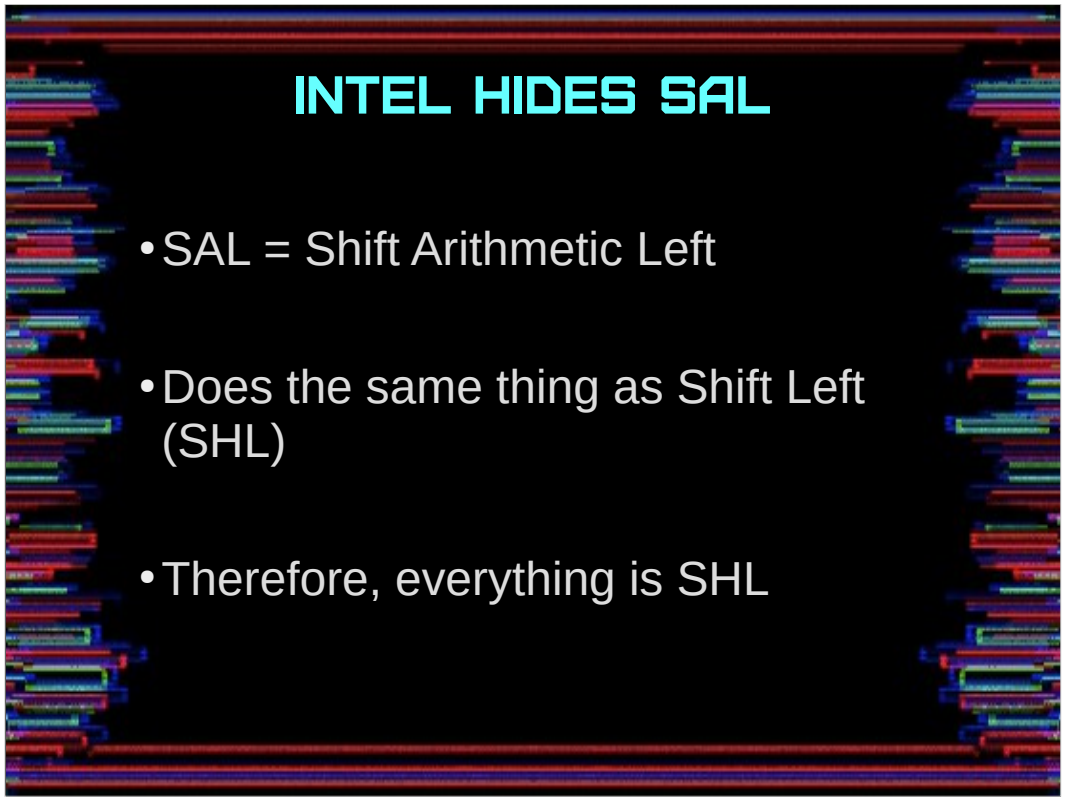


So this is the image of showing all of those redundancies



There's no real good reason to manually use a branch hint. There's also no way to do it directly with assembly.

However, you can manually machine the prefix in front of a branch instruction. It wont really affect much, but hey, you can (when you can't in assembly).



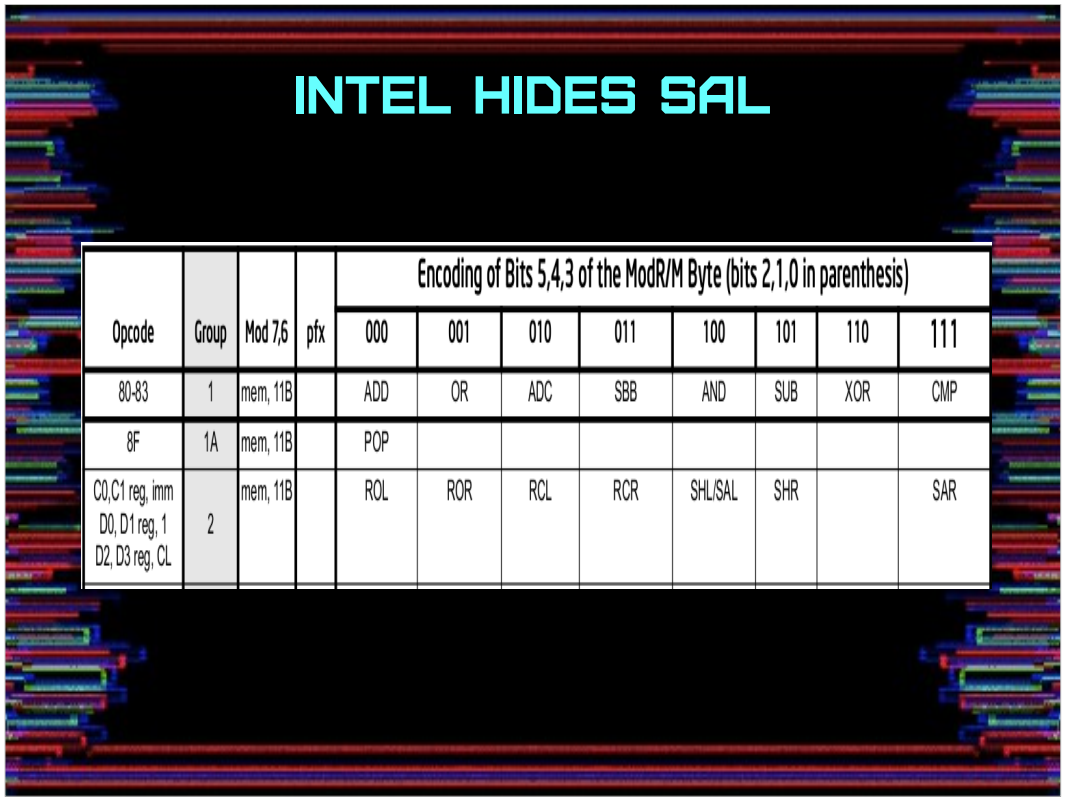
Similar to not having our assembly converting our TEST instruction to a equivalent form; SAL(Shift Arithmetic Left) gets converted to SHL(Shift Left). SAL and SHL are technically equivalent. The Intel manual recommends this and assemblers obey it.

The difference here is that there really is an encoding for SAL, and it is functional.

Here is our assembler converting our SAL instruction in assembly to SHL when it gets to machine code.

Note that even the machine code in the Intel manual is the same for SHL and SAL.

We will get to this next, but the /4 represents the specific instruction, where the D0 represents the group of instructions. For instance, /5 would be SHR (Shift Right).

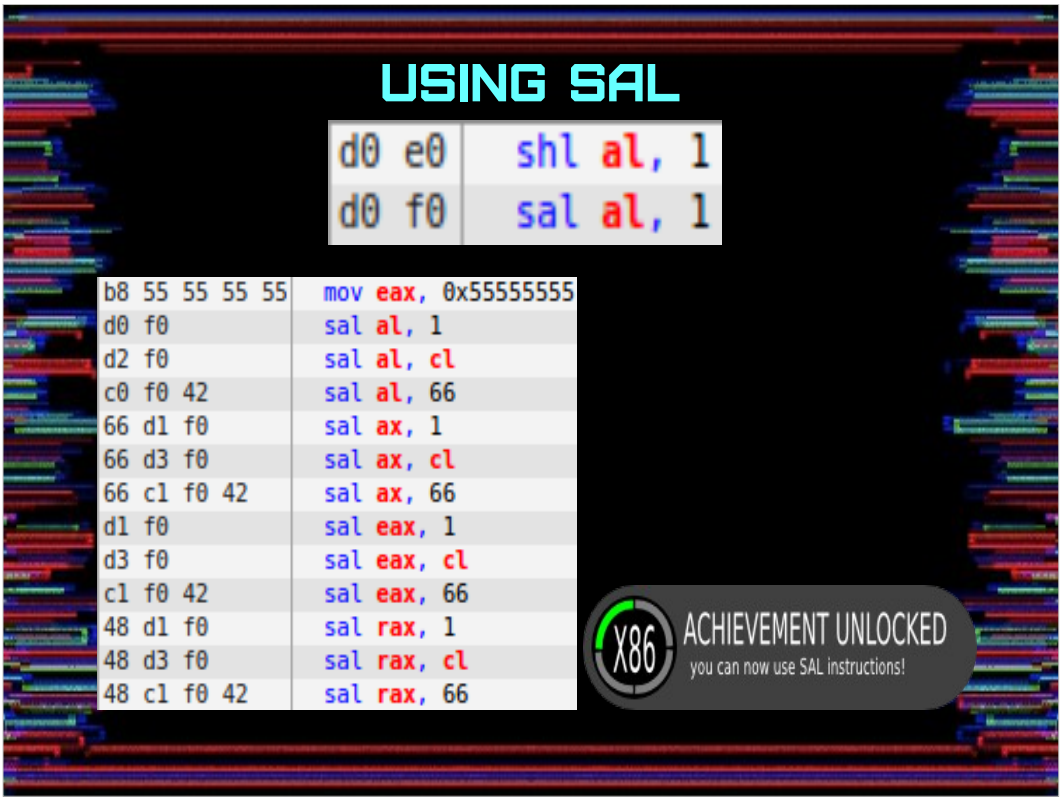


This table shows all of these /n numbers. We see that under '100' or /4, SHL and SAL are combined.

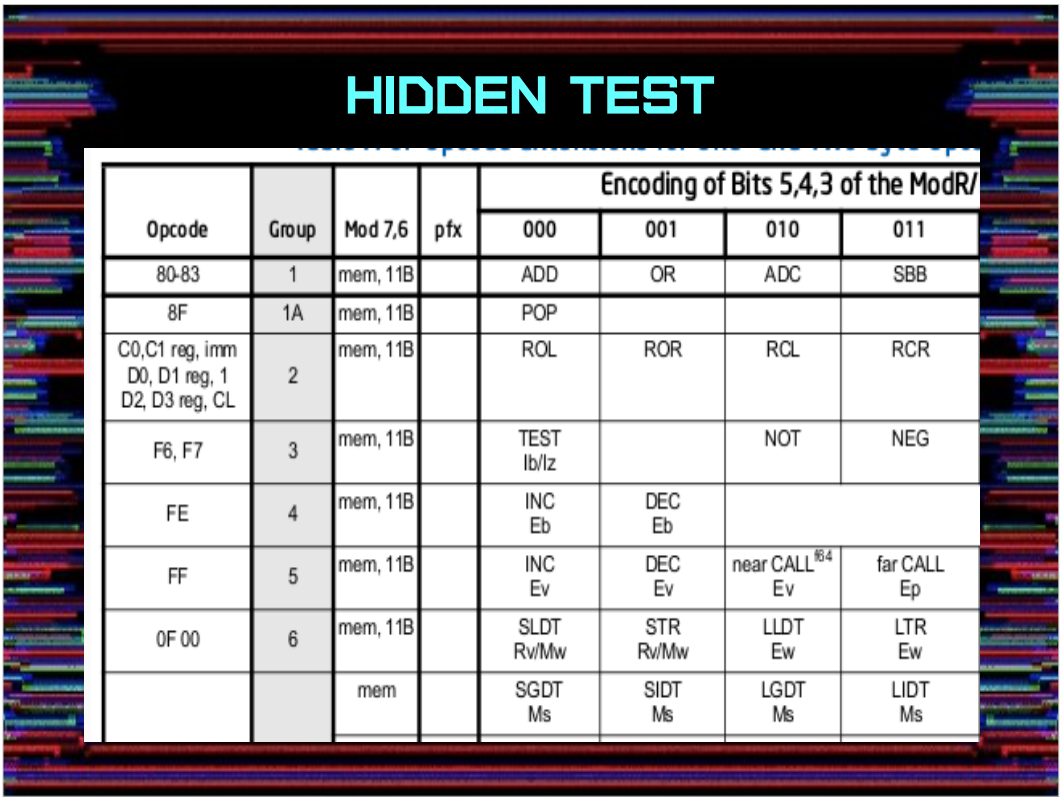
More interestingly, we notice that '110' or /6 is empty.

There is no way to mess around with this in assembly language, but we can do this directly in machine code to see what happens.


Be
4
2-2
tter
Ca
20
2-8-8-2
II
S
AL
13
2-8-3



It is SAL. After testing it, it works. SAL unlocked!



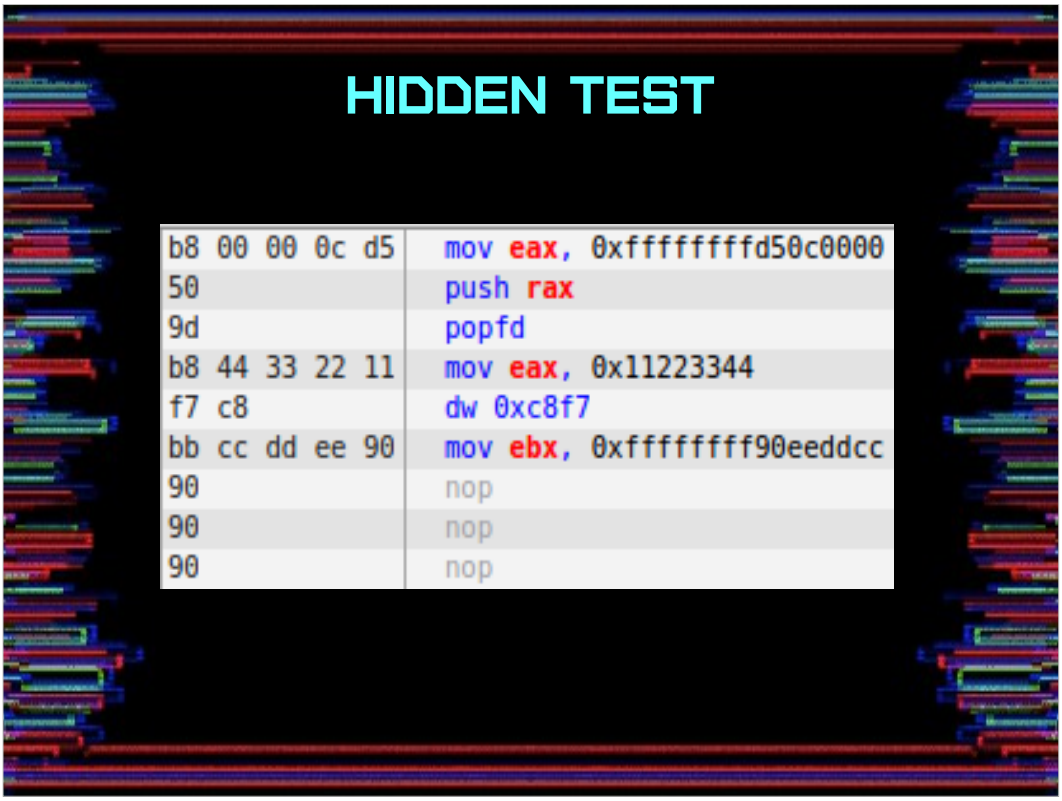
Opcode	Group	Mod 7,6	pfx	Encoding of Bits 5,4,3 of the ModR/			
				000	001	010	011
80-83	1	mem, 11B		ADD	OR	ADC	SBB
8F	1A	mem, 11B		POP			
C0,C1 reg, imm D0, D1 reg, 1 D2, D3 reg, CL	2	mem, 11B		ROL	ROR	RCL	RCR
F6, F7	3	mem, 11B		TEST Ib/Iz		NOT	NEG
FE	4	mem, 11B		INC Eb	DEC Eb		
FF	5	mem, 11B		INC Ev	DEC Ev	near CALL[i64] Ev	far CALL Ep
0F 00	6	mem, 11B		SLDT Rv/Mw	STR Rv/Mw	LLDT Ew	LTR Ew
		mem		SGDT Ms	SIDT Ms	LGDT Ms	LIDT Ms

There's an encoding under the machine code of 0xF6 (8-bit) and 0xF7 (32-bit) for the TEST instruction, as in TEST EAX, 0x11223344.

We will use the 32-bit encoding for this example. This is a /0 encoding, to mean TEST, as in /2 would mean NOT and /3 would mean NEG and so on.

You'll notice there is a blank spot in this table that would have an instruction for /1. It so turns out that this is also a TEST instruction. If you machine encode this, the processor will run this exactly as the /0 test.

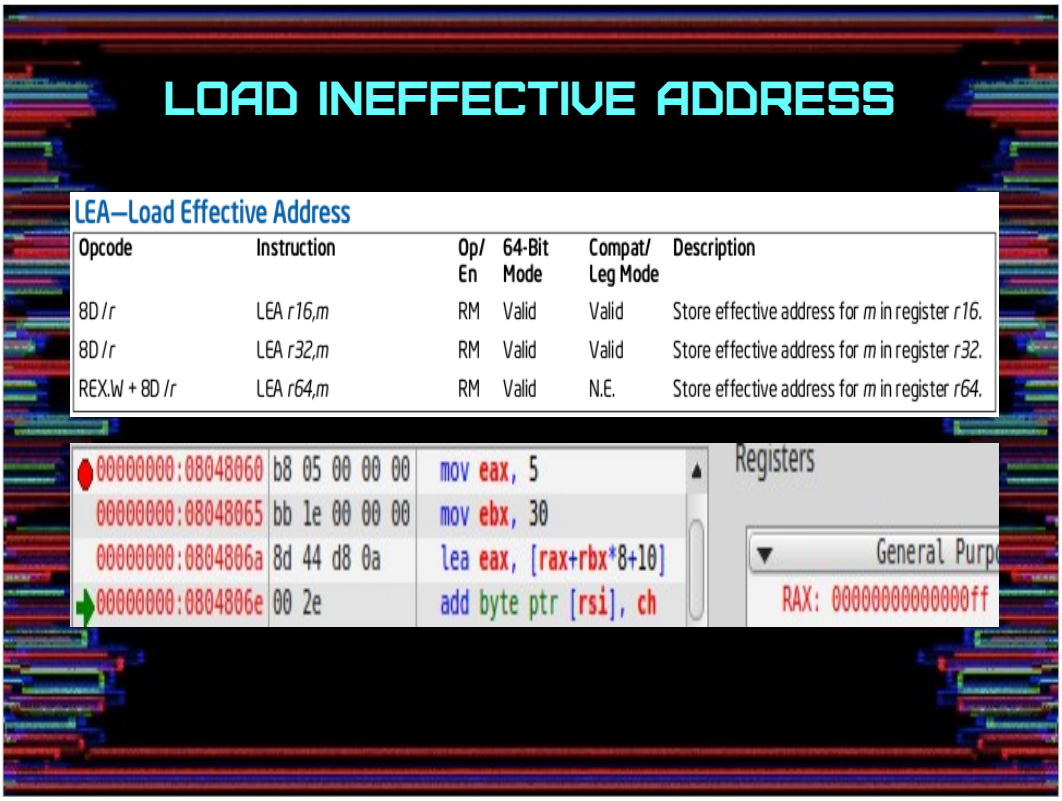
Your mileage will vary depending on the disassembler you use, for whether it tells you it is a TEST instruction or not...



In the case of the EDB (Evans Debugger), the instruction is not disassembled showing the TEST it actually is. We instead see a dw (data word directive) of 0xc8f7 and then a mov instruction.

This 'mov' instruction will never run because it doesn't exist, it is actually part of the operand data of the TEST instruction. This instruction should be:
TEST EAX, 0xeeddccbb

This TEST instruction is what the processor will actually execute



What the Load Effective Address does is stores the pointer address into a register. So not the value of the address into the register, but the actual address that the pointer would point to.

In the above example, we are running: LEA EAX, [RAX + RBX * 8 + 10].

Knowing EAX(RAX) is 5 and EBX(RBX) is 30 (decimal). So [5 + (30 * 8) + 10]. Simplify again to [5 + 240 + 10]. Finally, this simplifies to 255. In hex this is 0xff.

Note that RAX/EAX has 0xff as it's value after we run that LEA instruction. That's what LEA does in a nutshell. Compilers more often use this as a one instruction math hack.

Because of what this instruction does, it only makes sense to have a register as the dest operand and a pointer as the source operand.

However, the Encoding of the LEA instruction uses the MODR/M byte. This means that a register could be encoded with both operands (like and MODR/M based instruction).

If we try to do this in assembly, we get an error that we used an invalid combination of opcode and operands.

That doesn't stop us from directly encoding LEA EAX, EAX (8D C0).

However, all of this is fairly pointless as this instruction IS indeed invalid and will cause an error if it is executed. But in principle, this is a specific error that would be harder to achieve in assembly alone (without being able to machine hack)


PREFIX
ABUSE

BSWAP—Byte Swap

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
0F C8+*rd*	BSWAP *r32*	0	Valid*	Valid	Reverses the byte order of a 32-bit register.
REX.W + 0F C8+*rd*	BSWAP *r64*	0	Valid	N.E.	Reverses the byte order of a 64-bit register.

16-bit register?

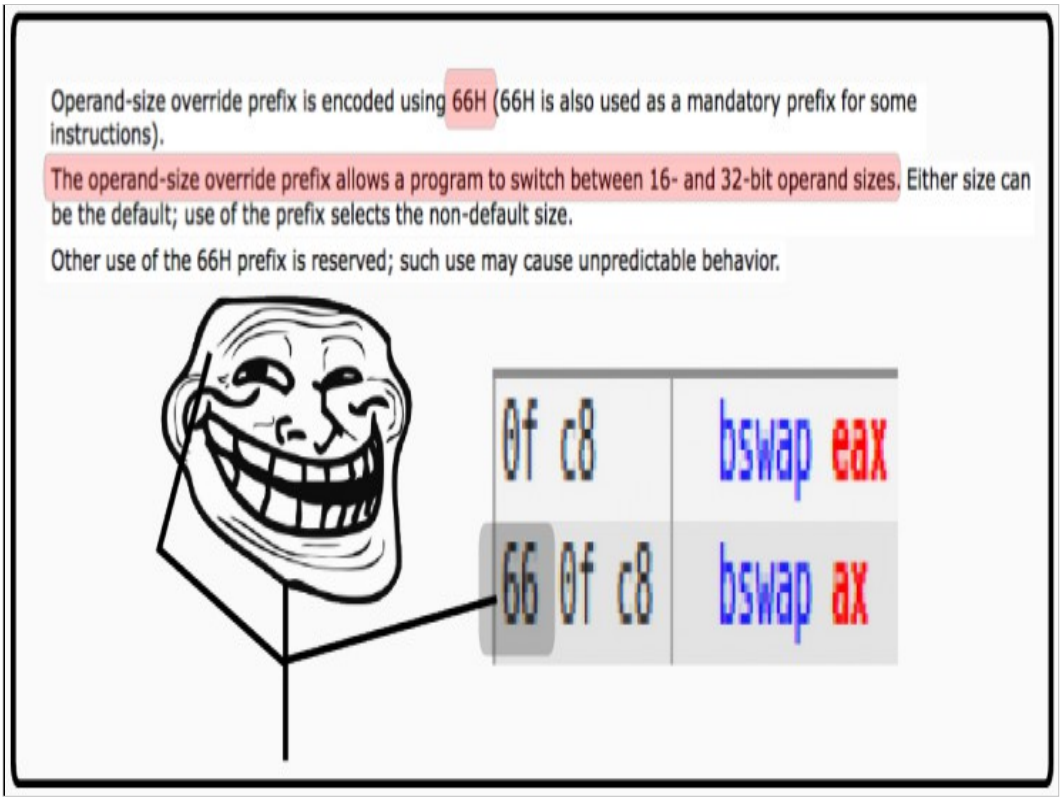
Reverses the byte order of a 32-bit or 64-bit (destination) register. This instruction is provided for converting little-endian values to big-endian format and vice versa. To swap bytes in a word value (16-bit register), use the XCHG instruction. When the BSWAP instruction references a 16-bit register, the result is undefined.

The BSWAP instruction can be used to reverse all of the bytes in a register. Notice that there is only an encoding for 64-bit and 32-bit registers, but not 16-bit registers. Even though 16-bits is enough bits to reverse 2 bytes. Why can't we do this?

Challenge accepted!

This is us in assembly attempting to write an instruction that uses bswap on a 16 bit register: BSWAP AX

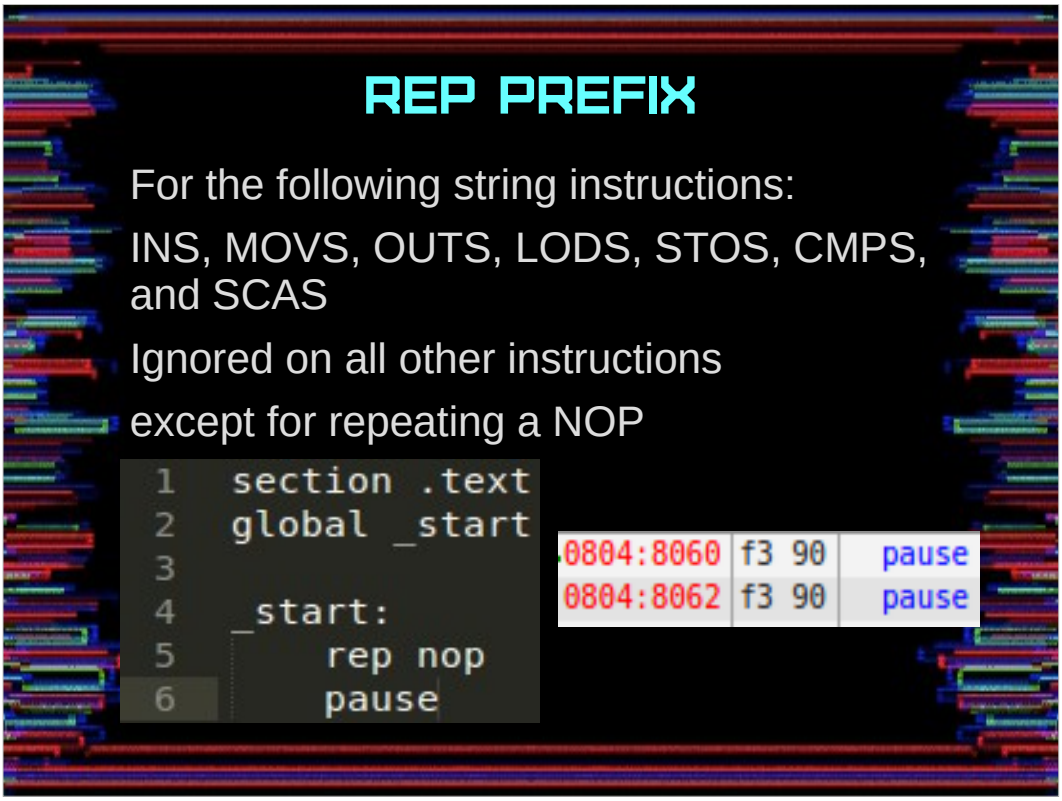
Of course we get an error saying that we used an invalid combination of opcode and operands



In 32-bit x86 (64-bit is similar but not exactly the same), there are prefixes that modify the operand sizes. For many instructions there is no encoding for 16-bit instructions, just an encoding for 8-bit and 32-bit. In order to use a 16-bit encoding, you should use a 0x66 or 0x67 prefix before your instruction (depending on what part of the instruction you wanted to override)

So we put a 0x66 in front of our BSWAP EAX and achieve BSWAP AX.

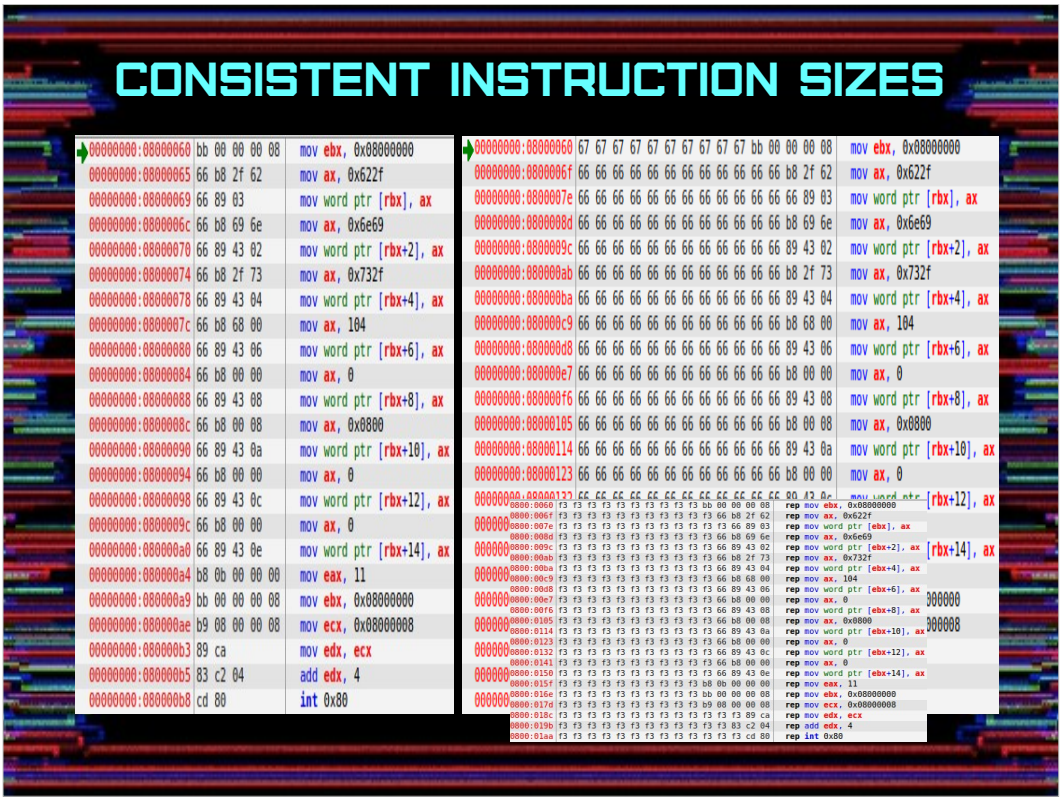
It should be noted however that this instruction doesn't work as intended (in my experience, it just clears the register completely)



The REP prefix can be used to repeat an instruction. This is really only intended to be used for instructions that operate on strings, so it doesn't do anything to any other instruction. The REP prefix byte is 0xF3

But there is one interesting exception, the screenshot shows these two different assembly instructions and how they mean the same thing to the processor.

This is because for whatever reason, the pause instruction is machine encoded as 0xF390.



The cool thing about this prefixes, is considering what would happen if you prefix a prefixed instruction with another of the same prefix. The answer is nothing. There is a limit to how many prefixes you an use; the instruction can be no larger than 15 bytes (you will get an error otherwise).

This screenshot shows some functional shellcode, and a couple of examples of the same code padded with prefixes. These examples make each instruction take the same amount of machine code bytes as every other instruction. I can't think of a reason why this would be useful, but it's still pretty cool.

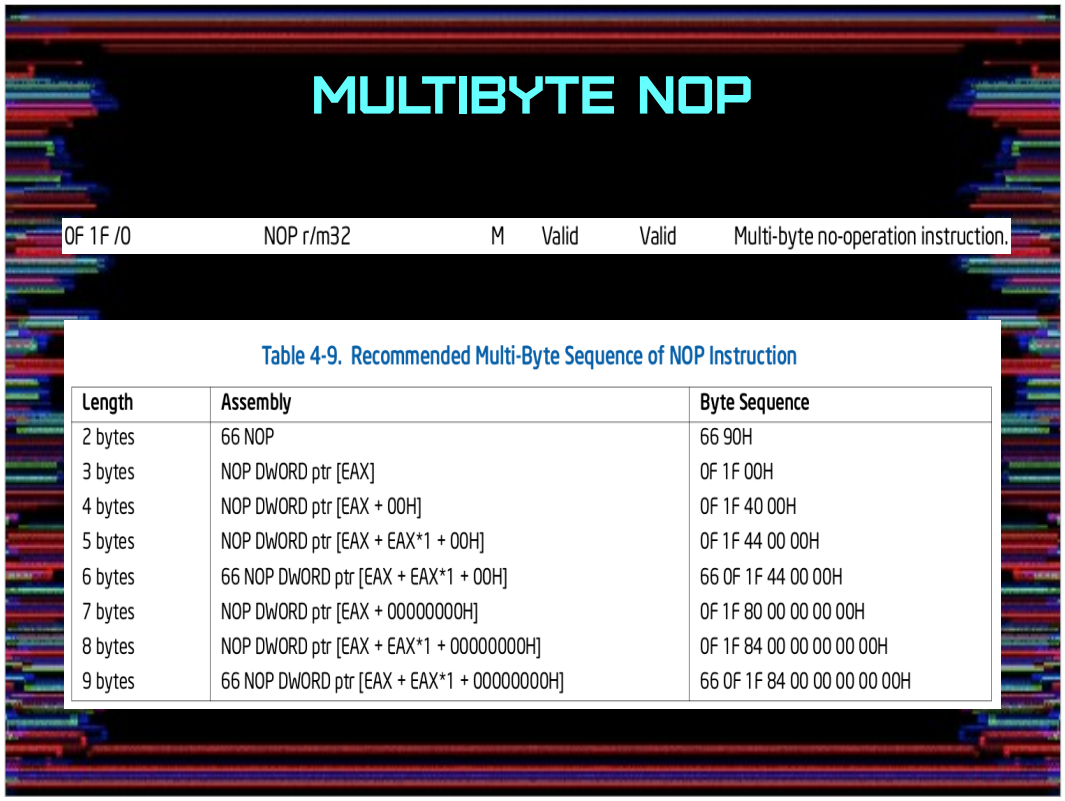
Here's something interesting, looking at the top instruction, the disassembly says that the instruction is xor [rax + rax], eax

However, if we actually type that instruction and assemble it, we get the same disassembly, but different machine code.

What the hell is going on here?

This is just more of nasm's interpretive dance. Obviously we don't want the first instruction, this is just the 'put a null' in it trick. We obviously want the version with less bytes right?


FULL OFFSETS
1 ; Obligitory NASM stuff----------
2 section .text
3 global _start
4
5 _start:
6 ;----------
7 xor [rax+rax*1 + 00000000h], eax
31 04 00 xor dword ptr [rax+rax], eax

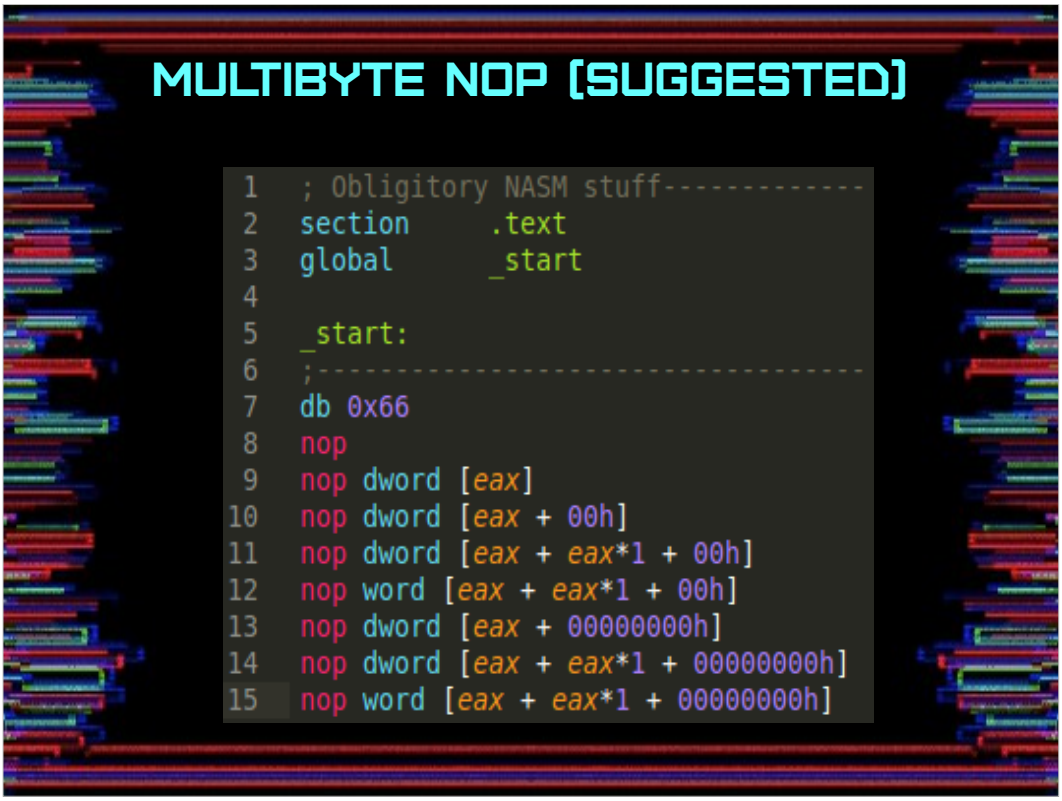


That is unless we don't.

The MultiByte NOP is the argument for not wanting our assembler to interpret our assembly into something optimized.

The MultiByte NOP allows for many different bytes because it takes advantage of how multibyte the MODR/M can be. The MODR/M argument doesn't actually contribute anything to the instruction in any meaningful way, it is just a dummy operand to add to the instruction size in a variable way.

So I'm going to take the suggested assembly in the intel manual and...



...and I'm gonna put it in an assembly source file and assemble it with nasm...

This is our result...

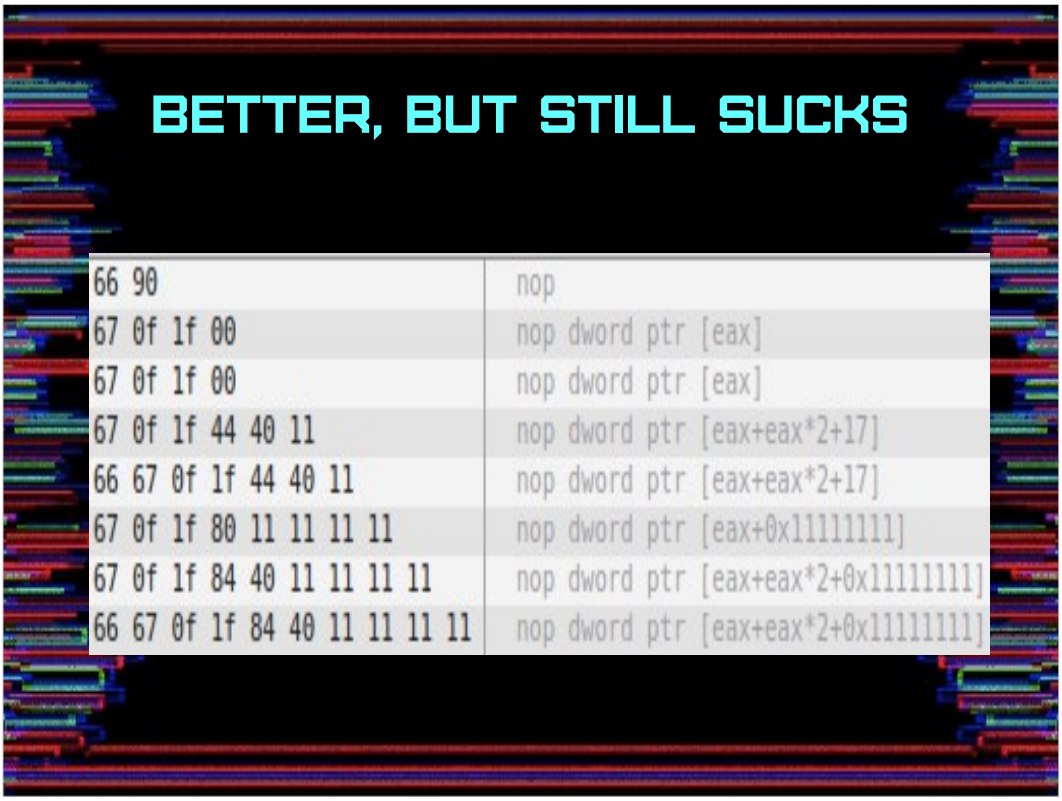
This for sure got an interpretive dance performed on it.

MULTIBYTE NOP (SUGGESTED): W/O NULLS

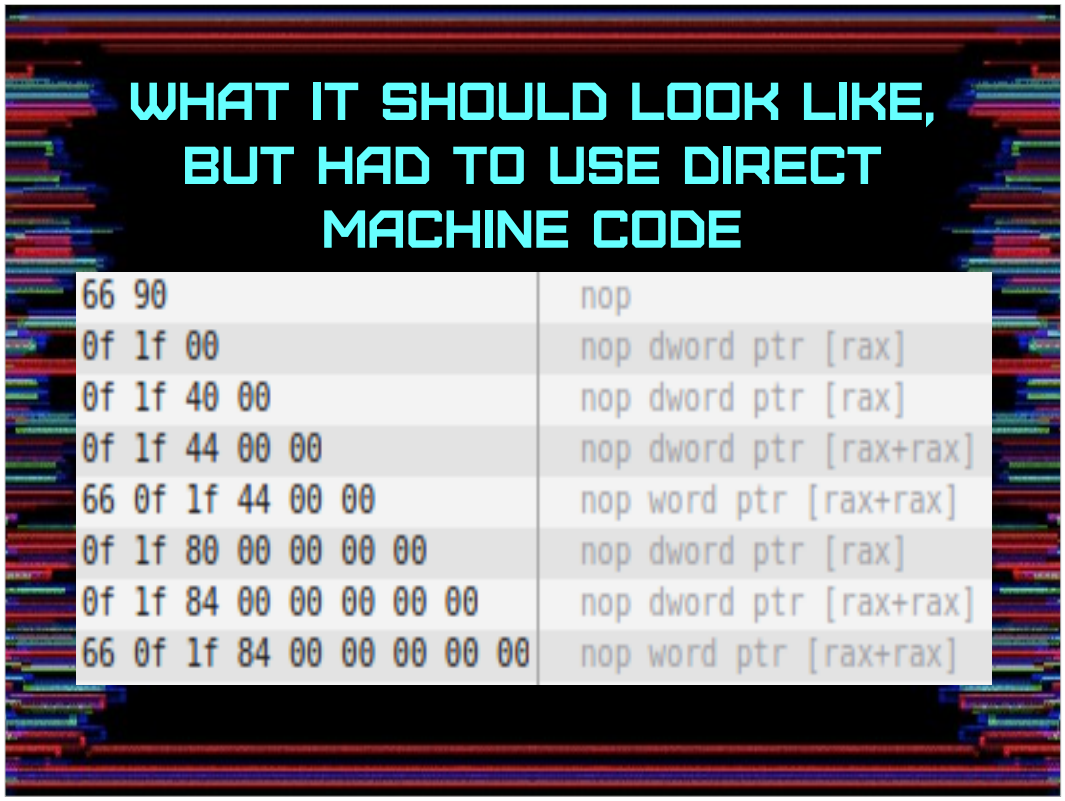
```
1   ; Obligitory NASM stuff-------------
2   section     .text
3   global      _start
4
5   _start:
6   ;-----------------------------------
7   db 0x66
8   nop
9   nop dword [eax]
10  nop dword [eax + 00h]
11  nop dword [eax + eax*2 + 11h]
12  nop word [eax + eax*2 + 11h]
13  nop dword [eax + 11111111h]
14  nop dword [eax + eax*2 + 11111111h]
15  nop word [eax + eax*2 + 11111111h]
```

I next try to mitigate this by putting some non null offsets into the pointers, this prevents the assembler from optimizing them out.

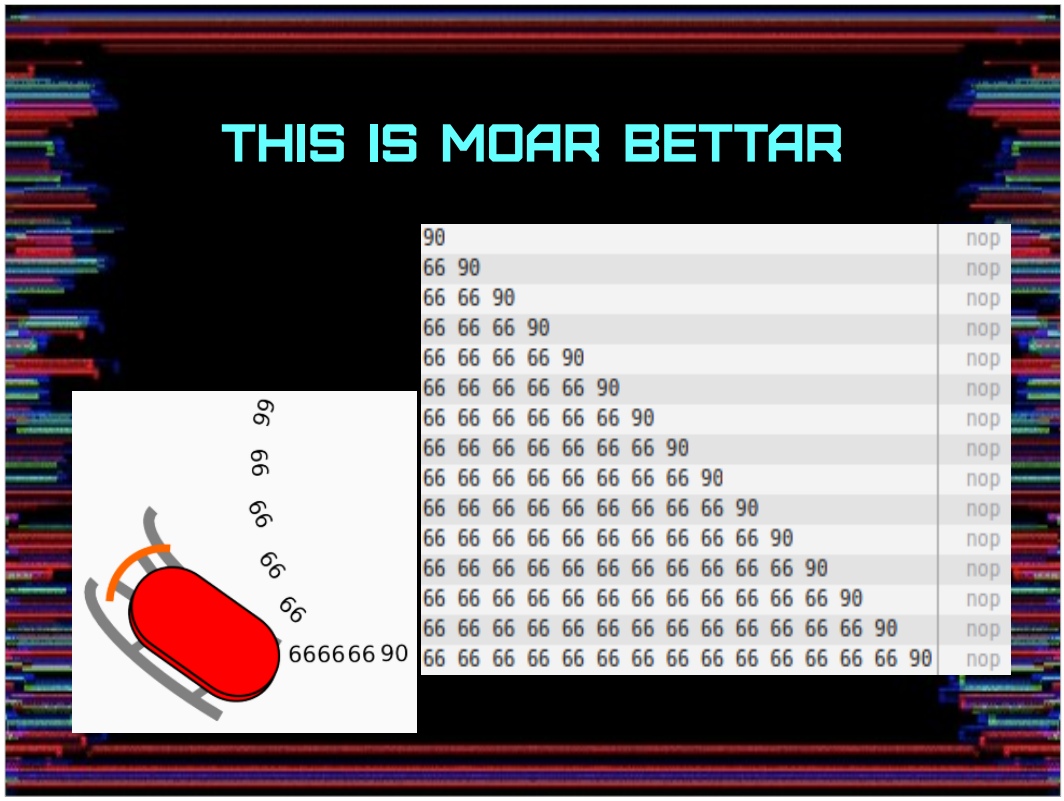
Of course we are misadventuring from what Intel suggests...



...but as you can see, it works a little bit better. But only a little bit.

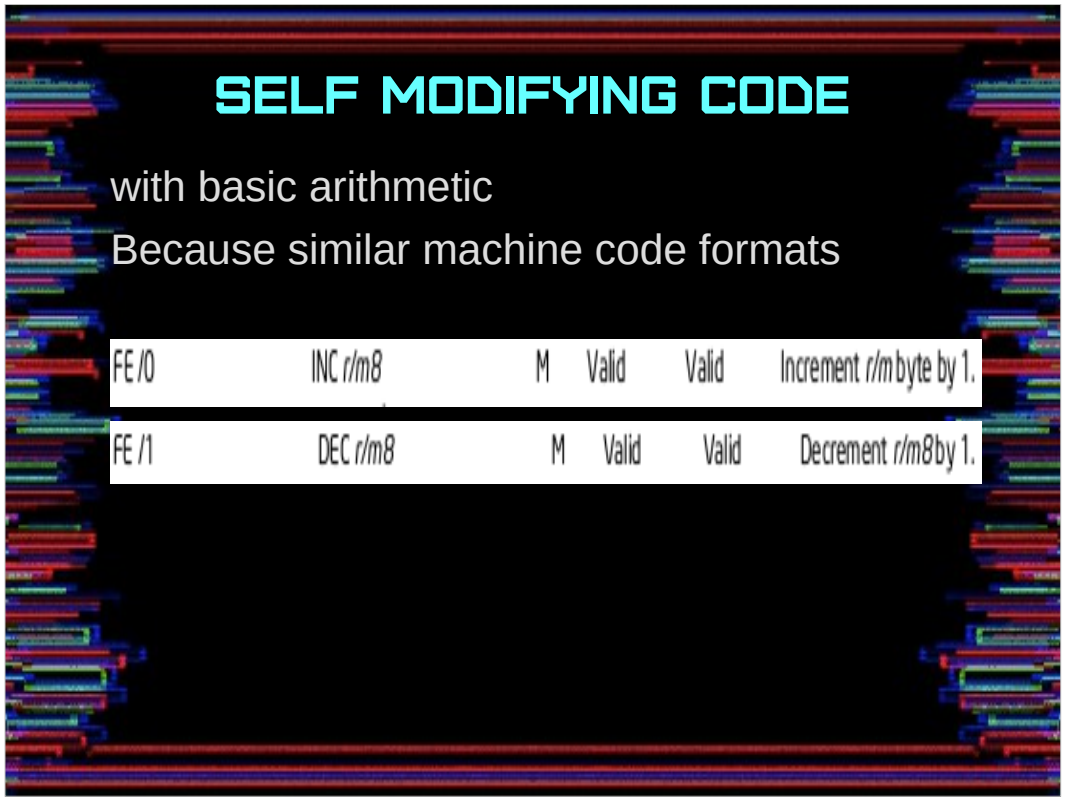


To get the (exact) machine code advertized in the Intel manual, I could find no other way but to manually program this in machine code



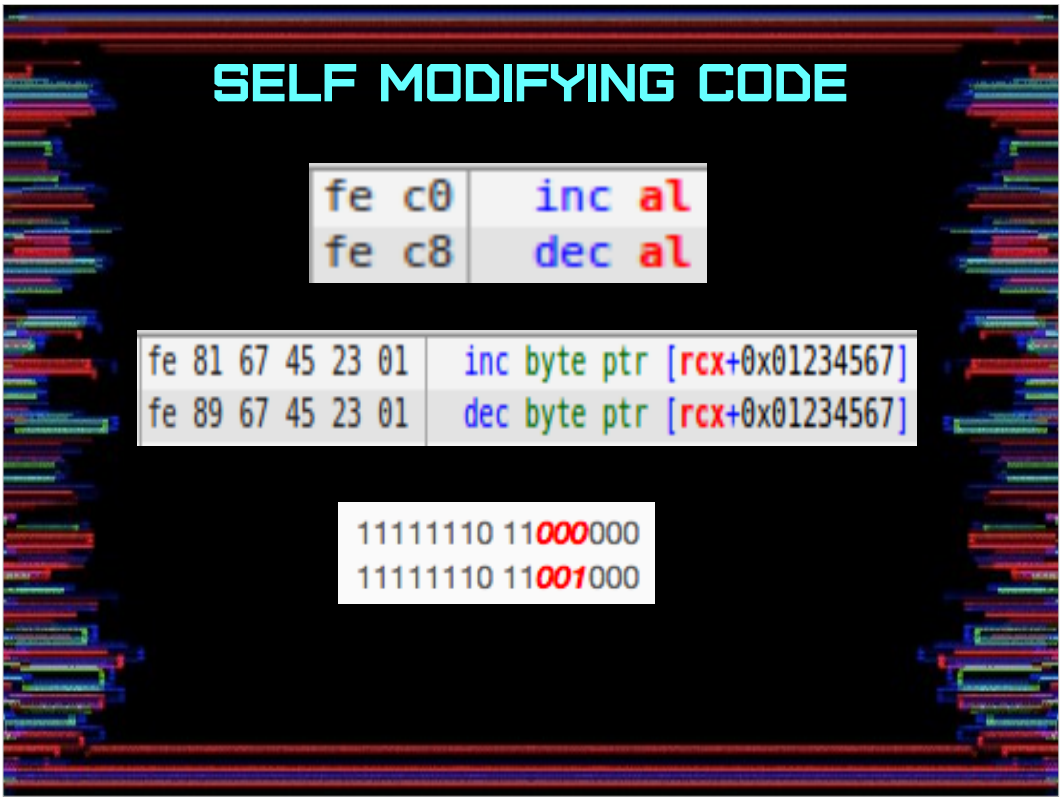
But why go through any of that trouble!

I'd rather just be ignorant and prefix up a normal NOP



Ignoring stuff like exploit development, an understanding of machine code can also be extremely useful for self modifying code. There are MANY different strategies/techniques a programmer could take to achieve cool self modifying code. We will only really explore one PoC example here.

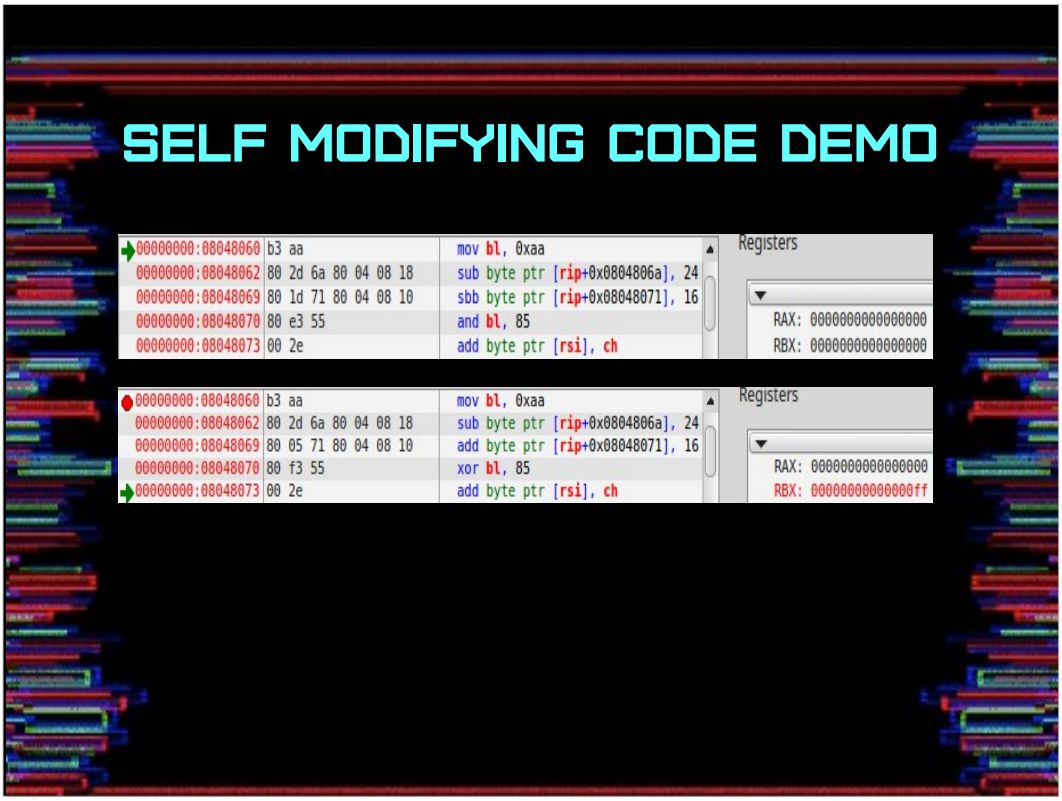
For this example, we can ADD a value to a [pointer] that happens to be the memory location of another instruction. Instructions with the /n format have the instruction itself encoded in the number of /n. For example, INC is 0xFE /0 and DEC is 0xFE /1. If we just added the right number to the right location of the INC instruction, it would be convertible to a DEC instruction.



This slide shows the machine code and assembly tho show the very small differences.

The last image shows the machine code of the first image in binary isolating out the 3 bits that control which instruction it is.

In red, the 000 means INC and the 001 means DEC. The difference to the 2 instruction is just one bit.

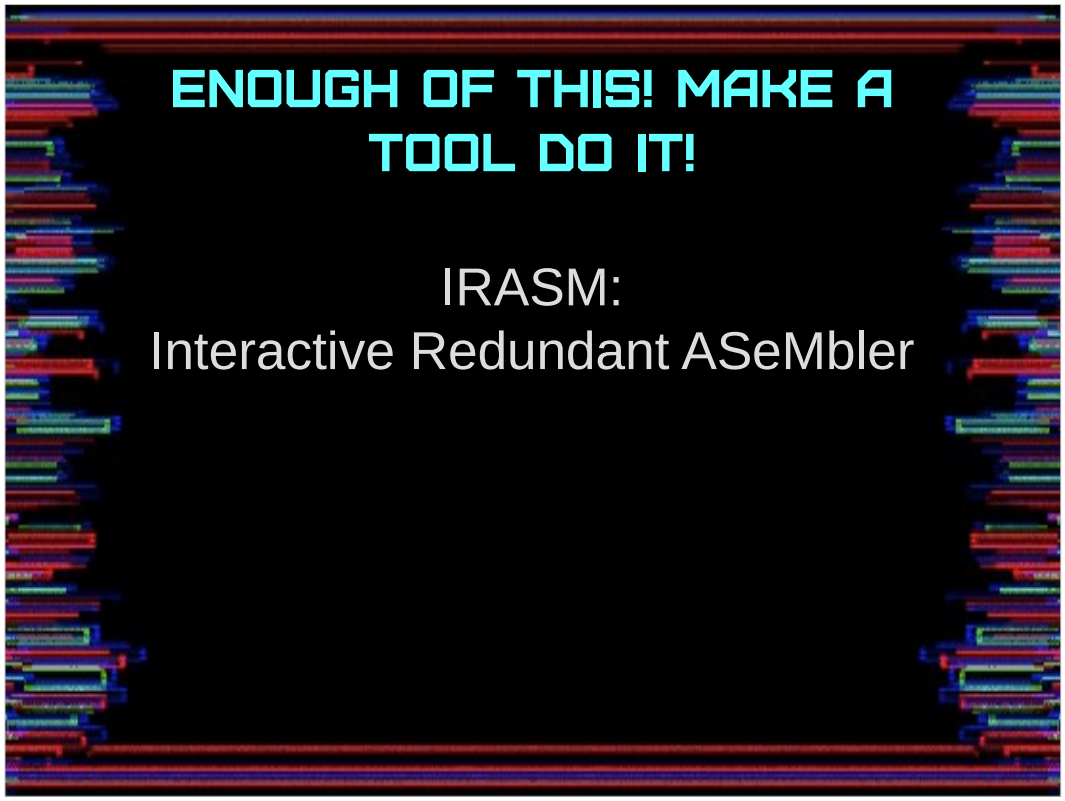


This demo will show a series of 3 instructions that are using this trick. When you get to the 3rd instruction, it isn't the same instruction it looked like before the program ran.

These 2 screenshots are more for the benefit of the PDF version of these slides. Time permitting, a live demo of this will be done during the presentation.

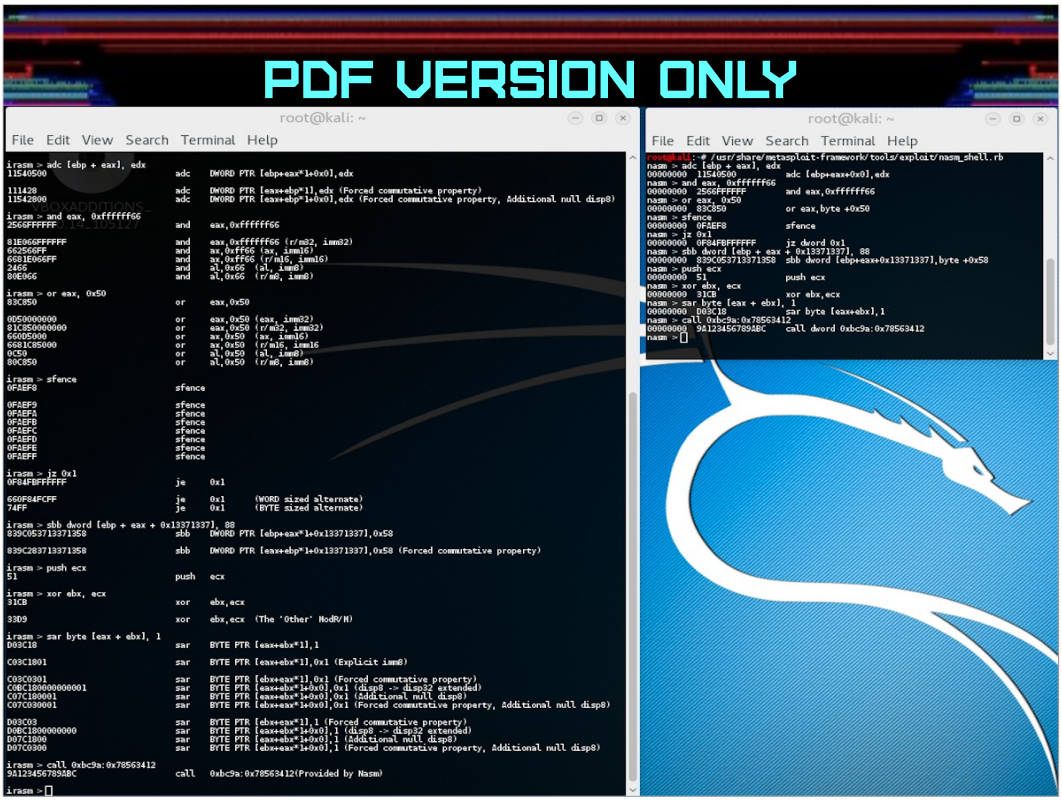
I will be giving a talk at CactusCon 2017 in September called Boot and Play. It is about 512 byte boot sector programs that are games and puzzles.

Self modifying code is a nice trick to have in the bag because it helps get the byte count down. The above trick that I mentioned is a trick that I use in TronSolitaire (https://github.com/XlogicX/tronsolitare)

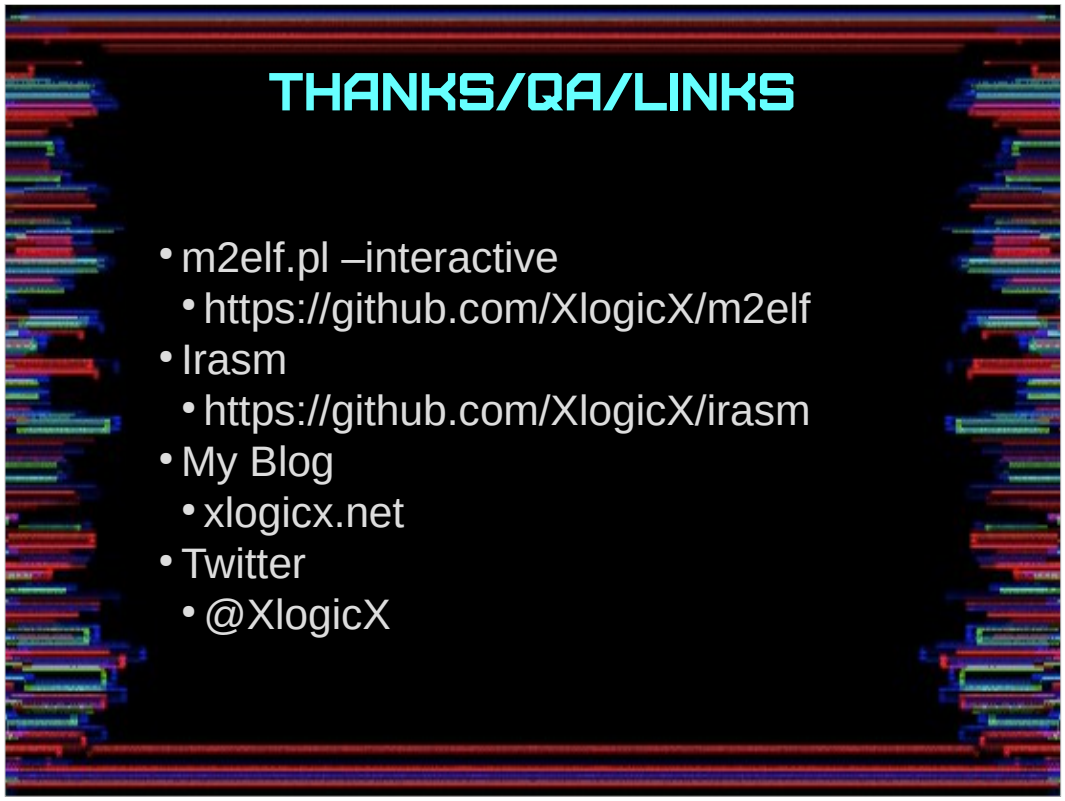


This is another 'demo' slide. This is where I demonstrate what the demo can do.

Hint: it pretty much does everything with the concepts described in this whole talk. It's like nasm_shell, but it outputs many other valid variations of machine code that represents the same assembly input.



This slide wont be displayed in the main presentation, instead I will demo the tool live, but since the PDF version can’t do that, this is a screenshot showing irasm side by side with nasmshell. The same assembly instructions are entered into both, you see the left hand side is more verbose.



I tend to speak fairly quick and am good at time management, so I may have time for questions. It really depends on this years DEF CON policy on Q/A. Regardless, I will make myself available for more in depth Q/A in the hangout room after I deliver the talk.

This slide is more just to leave up the links to the tools and my contact info / blog