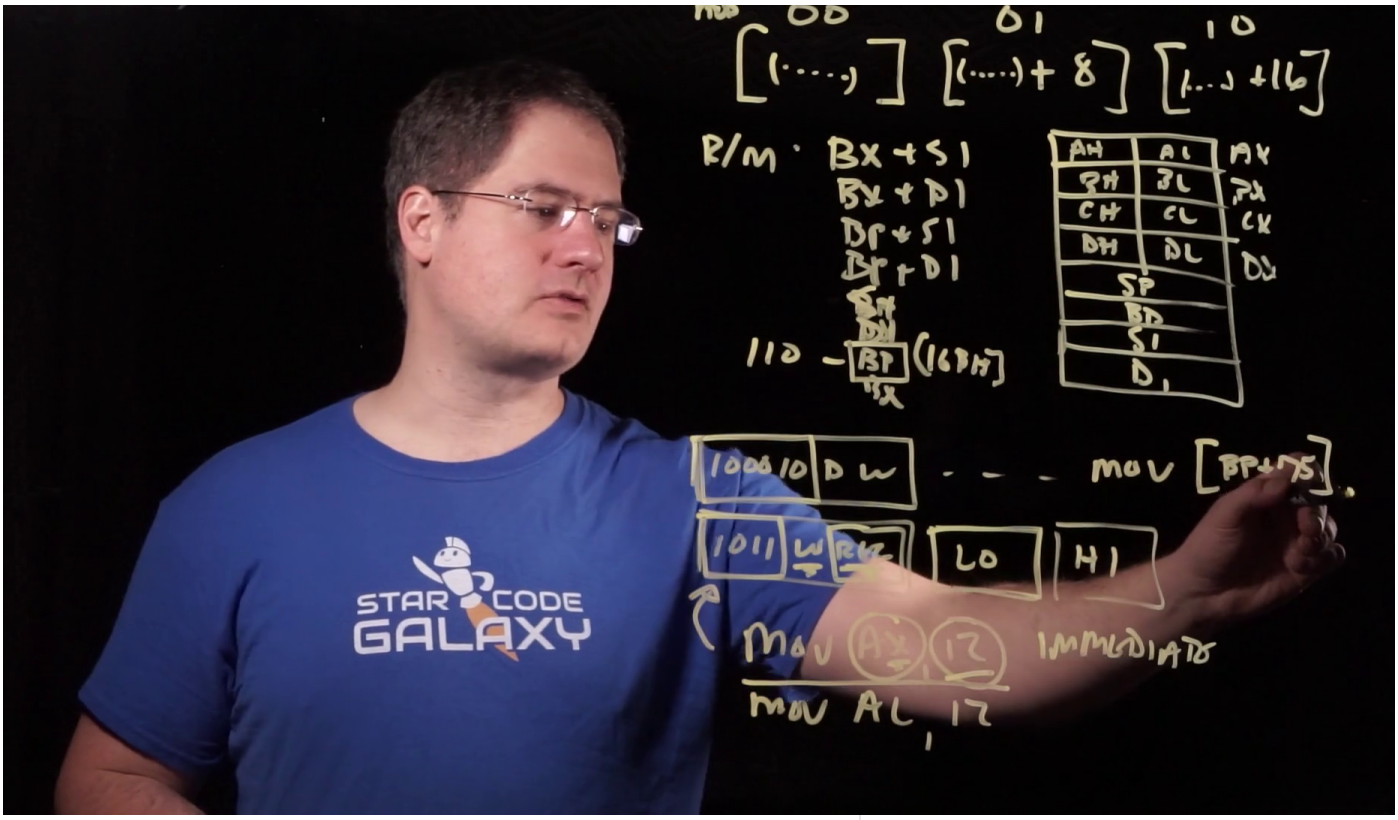


Decoding Multiple Instructions and Suffixes

Things get more complicated when different opcodes and variable instruction lengths are added to the mix!

CASEY MURATORI
MAR 05, 2023 · PAID

Programming Courses

A series of courses on programming topics.

Authors

Casey Muratori

This is the second video in Part 1 of the Performance-Aware Programming series. Please see the Table of Contents to quickly navigate through the rest of the course as it is updated weekly. Homework data is available on the github. A lightly-edited transcript of the video appears below.

In the previous video I gave you some homework where I asked you to decode a register-to-register move instruction on the original 8086. I hope it wasn't too difficult! I tried to pick something simple for the first assignment, but at the end of the day, nothing is ever really that simple in the land of 8086. Even a simple instruction is kind of tricky to decode, as you may have noticed!

I hope it went okay, but if you struggled with it, that's perfectly fine! At the end of these instruction decode videos, I will go over the code that I wrote as a reference, and you can see potential solutions to any problems you may have had. For now, what I'd like to do is add a little bit more to the problem. I'd like to give you a chance to work out a few

more details of 8086 instruction decode, just to get you in the mindset of a CPU and what it really does.

Last time we talked about the two-byte encoding for register-to-register move:

In this encoding, we relied on the MOD field to tell us it was a register-to-register move — when the MOD field was 11, that meant register-to-register.

But in the homework assignment, you could have just skipped that check because I was only asking you asking you to decode this specific kind of instruction. You didn't really have to check the MOD field, and honestly you didn't really even have to check the 100010 opcode because every instruction was going to have those two things set the same way.

What I'd like to do now is expand the decoding so that you do have to check everything. This is just to really make sure you've got it all down, and also so you can see how the actual decoding process changes dramatically from instruction to instruction even when you're looking at the *same* assembly language mnemonic. Even if we just stick with MOV, as you'll see, the assembler may generate *radically* different instruction encodings depending on what is being copied where.

You already saw a preview of this in the 8086 manual, where it listed several possible encodings under MOV in the encoding table. But even with just the instruction we already decoded — register-to-register move — we only decoded a *subset* of that particular encoding because when the MOD field is *not* 11, that exact same encoding might indicate a move to or from *memory*.

So the first thing I'd like us to do is expand our handling of that one

instruction encoding to decode *all* the possible things it can encode. This means handling the other possible values of the MOD field, and also parsing the optional bytes that we say in that table, the ones labeled "DISP-LO" and "DISP-HI":

The other values of MOD are there to encode *memory-to-register* moves (loads) and *register-to-memory* moves (stores). And some combinations of values will require parsing the two optional DISP bytes.

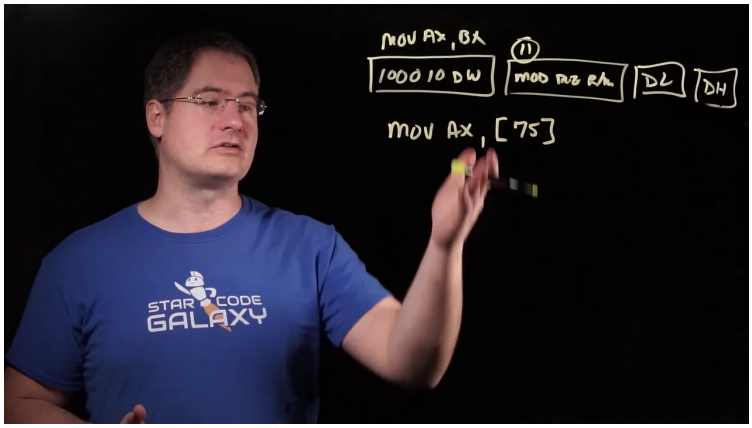
Before I go any further, I really want to make sure you understand how significant what I'm saying is in terms of how a CPU has to process these instructions in the 8086 lineage[1]: if the mod field is 11, we don't have these optional bytes. If the mod field is 01, we

have one extra byte, and if it's 10, we have two extra bytes. So not only does the CPU have to look at the first byte of this instruction, check the opcode, and then decide to read a second byte for the MOD REG R/M fields, but then it *also* has to check the *second* byte to see what the MOD field (and potentially R/M, as we'll see) to find out whether to read a third and fourth byte!

It really is a nasty, dependency-chain process that the CPU has to do to decode these instructions. This actually causes a lot of headaches in modern CPU design, as you might imagine, because if a CPU needs to decode four, five, six instructions per clock, well, how does it do that if it has to decode the first one to know where the second starts? It's a much harder problem than in an instruction set where the size of instructions is fixed.

But thankfully, that's not our problem, that's Intel and AMD's problem! But you're quickly gaining some insight into the kind of stuff their chip designers have to contend with.

Anyway, back to memory moves. First of all, how do we notate a memory-to-register or register-to-memory move in assembly language? One way to write it is if we want to do a load, we begin with something like "mov ax," like we did before, but instead of a second register for a source, we use *brackets* to enclose a *memory address* to read from:



The memory address can be as simple as just a constant, like 75. That would mean we're going to load the byte from address 75 and move it into the ax register — and since ax is a 16-bit register, it'll actually move byte 76 into the high bits of ax, too.

Now, because I am a bit sneaky, I picked "ax" here because I wanted to foreshadow a challenge problem in the homework. If you actually tried to

assemble this as-is, you might be shocked to find that many assemblers will *not* produce the memory-to-register MOV encoding! What has happened? Well, if that piques your interest, the challenge problems in the homework are for you.

I'll leave that little teaser there for now, and just slyly change *ax* to *bx* and continue on:

So if we wrote something like this, how is this going to be encoded? This is saying we will load 16 bits into bx from memory address 75 (and for the high byte, 76). It's a 16-bit move, so we know that the W bit will be set to 1, just like with the register-to-register move.

At this point, I should mention that if you are coming from a language that

doesn't have pointers, and you're uncomfortable with how memory addresses work or things like that, this memory move instruction might be confusing to you. But it doesn't really have to be for purposes of understanding the 8086, or honestly even for understanding pointers at all. Pointers and memory are actually *way simpler* than people make them out to be!

If you imagine you just had an array of bytes called *Memory*, and it was 65536 bytes lone — 64k worth of bytes — that is exactly what we the CPU is thinking of when you do a memory access on 8086. The [75] is jus like looking up index 75 in an array of bytes, and getting out the value. That's all the CPU is doing. A "memory address" or a "pointer" is nothing more than the *index* in the array of memory where you are going to look for some data. There's seriously nothing more to it than that!

In this case, because we're reading 16 bits, we're just accessing *two* elements

of the array — [75], which goes into the low 8 bits of bx, and [76], which goes into the high 8 bits. That's it.

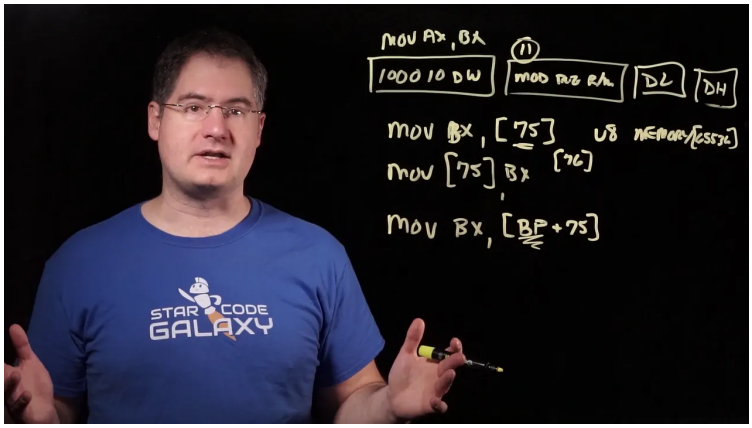
So that's how we write a *load* — a copy from memory into a register — in 8086 assembly. We could also do a *store* — a copy from a register back to memory. We write it the exact same way, but with the operands reversed:

How do we encode these loads and stores in the instruction stream? We use the MOD and R/M fields.

Unlike register-to-register moves, where MOD was 11, when a move involves memory the MOD field will be 00, 01, or 10. That's our first step in decoding the memory address: if the MOD field is not 11, then we know we have to decode something more

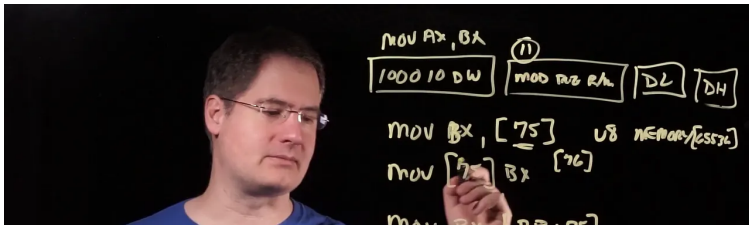
complicated than just a register name.

The effective address calculation is what we call resolving the address specified by the expression within the brackets. When it's something as simple as 75, it doesn't *look* like an expression, but it is. And we can use a more complex expression if we want:



For example, we can use [bp + 75], which says to take the value of the bp register and add 75 to it.

But our options aren't unlimited. We have to choose from a few preset expressions. For example, we could *not* write [2ax + bx + 7]:



That's not one of the possible patterns.

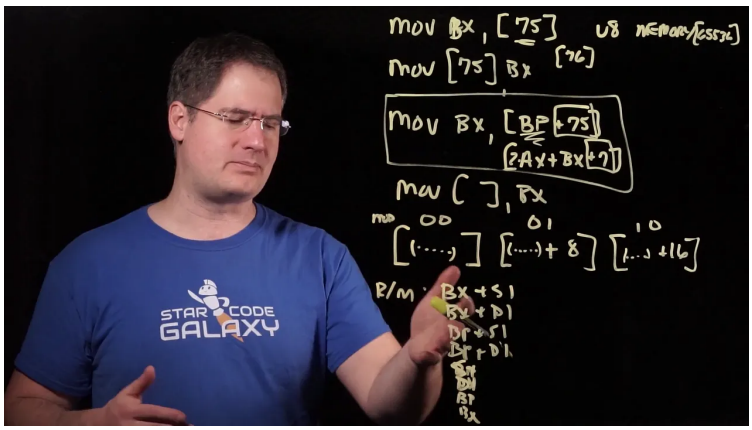
Why are there only certain patterns? The answer becomes obvious once we look at how these expressions are *encoded*. They are derived directly from the two bits in MOD and the three bits in R/M — which, as the small number of total bits would indicate, can only encode a small number of expression patterns.

First, the MOD field specifies if there is a *displacement*. In the expression [bp + 75], the constant 75 is the displacement. If there is no displacement, MOD is 00. If there is an 8-bit displacement, MOD is 01, and that "DISP-LO" byte we talked about earlier comes after the instruction. And as you might guess, is MOD is 10, there's a 16-bit displacement, and both the "DISP-LO" and "DISP-HI" bytes will follow the instruction.

So that's the first part of the decoding:

check the MOD field, and read the extra bytes if they're there. But there's a catch — if you want to do the *challenge* homework, you'll need to watch out for a special case with MOD 00! More on that in a second.

After looking at the MOD field, we need to look at the R/M field to figure out what expression pattern we will use. It's another table which maps the 3 bits in R/M to eight different expressions:



Those eight expressions are the *only* expressions we get! They are assorted combinations of bx, si, bp, and di — no other register can appear!

So that's all we get. We get those eight expressions, and then using the MOD field we can say whether we want to add an 8-bit or 16-bit constant to the

end. Now for the catch!

The expression pattern when R/M is 110 is *usually* just [bp]. If MOD is 01, it will be [bp + 8-bit constant], and if MOD is 10, it will be [bp + 16-bit constant]. But if MOD is 00, instead of the expression being [bp] — which is what you would expect — it *changes* to being *just a 16-bit constant*. This is called *direct addressing*, and it's actually how we would encode the bare [75] expression I showed earlier! It's actually an oddity of the table.

So that's how you encode the address you're reading from. And you'll notice I said reading from. What about *writing to*? Do we have to do something special to encode a *write* instead of a read?

Thankfully, we don't. Remember the D field? Well this is why that field exists in the first place! If the D field is set, then we know the REG parameter encodes the destination register. So it will be set when we are doing a read from memory. But if the D field is *zero*, that means REG encodes the *source* register, right? So *that's* how we know which

way the mov goes! The D field tells us whether the memory expression — which is always decoded from MOD and R/M — is the first operand or the second operand in our ASM.

Now you may have noticed, when you did the homework last time, that there are more than just the ax, bx, cx, and dx registers. Because if you did the homework, you had to decode the REG field, and that field encoded some more registers — some of which we've used in the effective address table.

And you may also have noticed that ax, bx, cx, and dx all had *l* and *h* variants, so you could *specifically* talk about the high bits or low bits of those registers. But there were *no such variant*s for the other four registers in the table, sp, bp, si, and di. Why is this?

Well, again, x86 has a reputation for being a crazy instruction set for a reason, and this is just one of those reasons. Even with just these eight registers, you can already see *significant limitations* on what you can and can't do with each type of register.

We've only looked at *one instruction* so far and we've already seen two *major* differences!

The first major difference is that some registers can have their low and high bytes targeted separately: ax, bx, cx, and dx. The second major difference is that some registers can appear in memory address expressions: bx, bp, si, di. Can you see why this instruction set would be hard to program well? Or to write a good compiler for? Think of all the constraints you have to consider when trying to write the instruction streams!

From our perspective, since we're trying to learn about CPUs, it's actually kind of fun. If you're like me, you like seeing this kind of complexity and learning about it. So from the perspective of a fun homework problem, this is kinda cool! But if you think about how this must have been when it was your job to ship reliable, efficient code generators for this ISA... yikes.

OK, so that's the first part of the

homework. The second part of the homework is decoding a *second instruction opcode*.

Thus far, we've only talked about decoding *one opcode*, which is 100010. You didn't really even have to check it for the previous homework, because all the instructions had that pattern in the top bits of the first byte.

Now, I'd like to add a second opcode, 1011. This is the opcode for an *immediate-to-register* move.

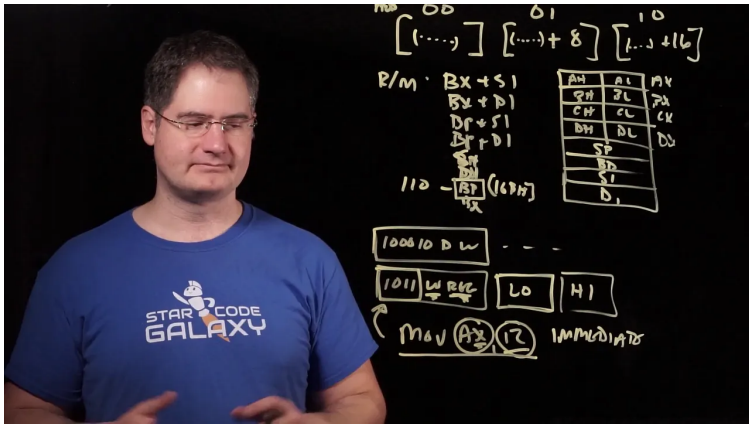
We haven't really talked about immediates yet. It's a somewhat weird term, but it's a very simple concept: an immediate is a data value that is encoded directly into the instruction stream, so that it can be fetched when the instruction stream is fetched rather than requiring a separate move instruction to read.

In a way, the *displacement* value we just saw was an "immediate". It was a constant value we were using to offset a memory access, and instead of it coming from a register, it came *directly*

from the instruction stream.

The main reason for these immediates is, as the name implies, to be *more efficient*. Since the instruction stream is being fetched in order, if we know we're always going to need some constant value for an operation, we can simply encode it directly into the stream and save the CPU the trouble of decoding an entire instruction just to read the constant separately.

Moving an immediate to a register has a similar syntax to all the other moves. If we want to move the value 12 into the register ax, we just write "mov ax, 12":



How is *this* encoded?

As I said before, the opcode for this kind of mov is 1011. So the high bits of

the first instruction byte will be 1011. Following that, we will have the same W field we had before: if it's a 16-bit register, it will be 1, if it's 8-bit, 0. Note that the *meaning* of this bit is the same, but it's *location* is different! It used to be the bottom bit, now it's the fourth bit, so watch out for that.

But that's not the only thing the moved. The REG field, which encodes the register name, is *now in the first byte*. It used to be in the second byte! Why all this shuffling around? Well, remember, memory was expensive back then, and fetching bytes from it was slow. So common instructions like this one were made as compact as possible! If a byte could be saved, it was.

So this instruction takes up *only one byte* of control information, unlike the previous instruction which took two. But it *does* have more bytes, of course: if W is 0, then there will be a second byte to encode the 8-bit value to move into the register. If W is 1, then there will also be a *third* byte to encode the 8-bit value to move into the high bits of

a 16-bit register.

That's everything for today's homework. It gets really complicated really fast, doesn't it? Can you see why I wanted you to get a feel for how much intricacy there is in x86 instruction decoding? With just *two opcodes*, we've already seen all sorts of different bits packed in all sorts of different ways! And our instructions can be one, two, three, or four bytes long.

Before we take a look on the computer and go over the listings, I did want to mention one more thing if you're planning on doing the *challenge* homework. Yes, there is *challenge* homework this time because... well, why not? It's optional, don't worry.

In assembly language there is an extra syntactic element you use when the assembler *wouldn't be able to automatically tell* what the W bit should be in the encoded instruction. If you are going to do something like move an immediate *directly to memory*, without going through register, then there wouldn't be any "ax" or "al" or "ah" to

tell the assembler whether to use 8 bits or 16 bits. In this case, you must add the "byte" or "word" keyword to the immediate value to tell the assembler whether it should be one byte or two:

So, "byte 12" means a 12 encoded in 8 bits, whereas "word 12" means a 12 encoded in 16 bits. Obviously all the top bits are 0 in both cases, but it changes *how much memory is written*. In the byte case, it's one byte, and in the word case, it's two.

In a way, it's almost like a cast — it's telling the assembler exactly what size something should be, so it doesn't have to guess.

Okay, let's go over to the computer now and I'll walk you through the homework assignment and the parts of the 8086

manual you'll need. You'll probably remember most of this from last time, but I just want to refresh your memory.

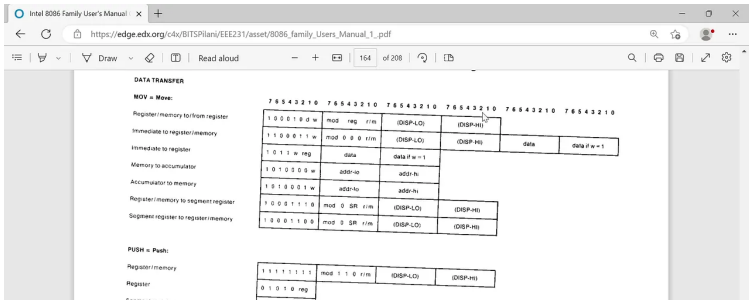
Here is listing thirty-nine, which is the new homework assignment:

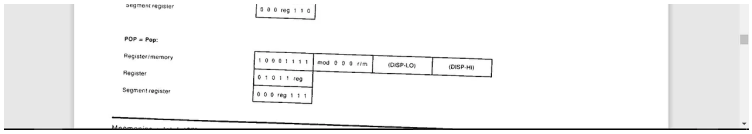
```
;
========================================
========================================
==
; LISTING 39
;
========================================
========================================
==

bits 16

; Register-to-register
mov si, bx
mov dh, al

; 8-bit immediate-to-register
mov cl, 12
mov ch, -12

; 16-bit immediate-to-register
mov cx, 12
mov cx, -12
mov dx, 3948
mov dx, -3948

; Source address calculation
mov al, [bx + si]
mov bx, [bp + di]
```

```
mov dx, [bp]

; Source address calculation plus
8-bit displacement
mov ah, [bx + si + 4]

; Source address calculation plus
16-bit displacement
mov al, [bx + si + 4999]

; Dest address calculation
mov [bx + di], cx
mov [bp + si], cl
mov [bp], ch
```

And you can see that this time I've put in comments to label what the different lines are testing. Unlike last time, where everything was the same kind of mov, now we're testing different forms of the instruction so I wanted to make sure you could tell which was which.

If we look back at the Intel manual, you may recall table 4-12 where we found the instruction encodings for all the mov instructions:





That top instruction, 100010, is the one we decoded last time. And this time, we need to decode more of it... but we *also* have to decode the third one, "immediate to register", which is 1011. So you have to handle *both* those encodings, not just one... and if you plan on doing the challenge homework, well, you'll have to handle even more than that! But I don't want to spoil the fun.

If we look on the previous page, we can see the tables for decoding MOD and R/M into an effective address calculation. Table 4-8 explains the values of MOD:

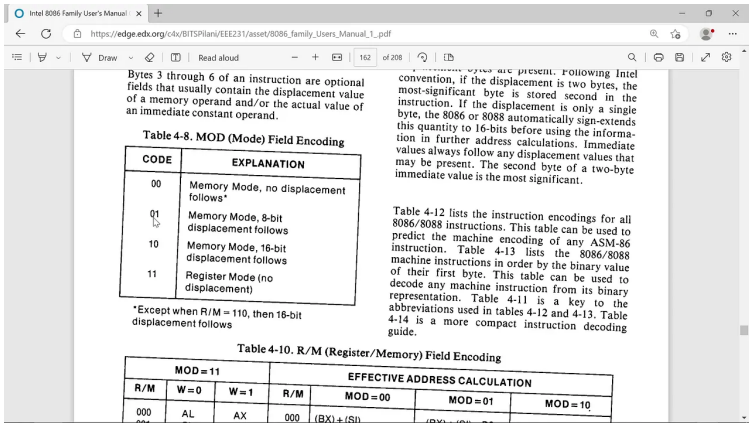
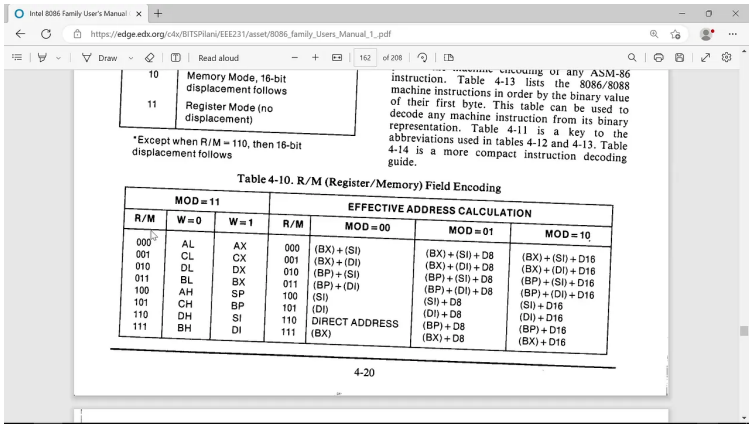


Table 4-10 explains the values of R/M,

and how they combine with MOD to form the full expression:



10 Memory Mode, 16-bit displacement follows

11 Register Mode (no displacement)

*Except when R/M = 110, then 16-bit displacement follows

instruction. Table 4-13 lists the 8086/8088 machine instructions in order by the binary value of their first byte. This table can be used to decode any machine instruction from its binary representation. Table 4-11 is a key to the abbreviations used in tables 4-12 and 4-13. Table 4-14 is a more compact instruction decoding guide.

Table 4-10. R/M (Register/Memory) Field Encoding

MOD = 11			EFFECTIVE ADDRESS CALCULATION			
R/M	W = 0	W = 1	R/M	MOD = 00	MOD = 01	MOD = 10
000	AL	AX	000	(BX) + (SI)	(BX) + (SI) + D8	(BX) + (SI) + D16
001	CL	CX	001	(BX) + (DI)	(BX) + (DI) + D8	(BX) + (DI) + D16
010	DL	DX	010	(BP) + (SI)	(BP) + (SI) + D8	(BP) + (SI) + D16
011	BL	BX	011	(BP) + (DI)	(BP) + (DI) + D8	(BP) + (DI) + D16
100	AH	SP	100	(SI)	(SI) + D8	(SI) + D16
101	CH	BP	101	(DI)	(DI) + D8	(DI) + D16
110	DH	SI	110	DIRECT ADDRESS	(BP) + D8	(BP) + D16
111	BH	DI	111	(BX)	(BX) + D8	(BX) + D16

4-20

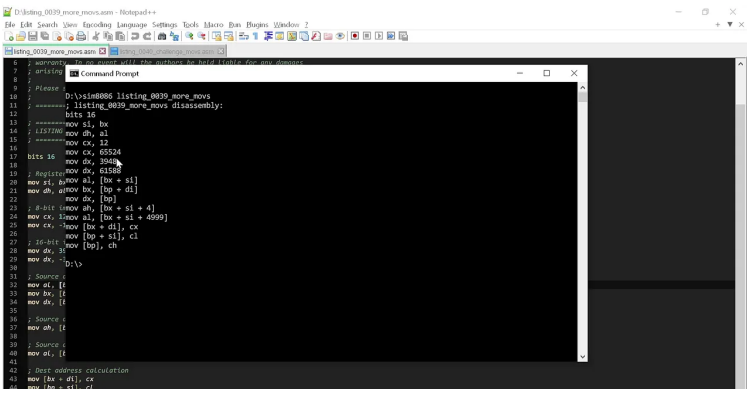
You can see the “special case” I mentioned there in table 4-10 under the “MOD = 00” heading. See that “DIRECT ADDRESS” entry? That’s where [BP] *would* have gone, as you can see from the “MOD = 01” and “MOD = 10” columns. But it’s the slot they chose for the direct memory address encoding, so that’s what MOD=11 R/M=110 decodes to instead.

If we go back to the listing, I'd like to point out something to watch out for: if the assembly language programmer put in a *signed* immediate, like the -12 in the listing, *how are you supposed to know that*? Remember, there were no bits in the encoding to say whether something was signed or not! Why not?

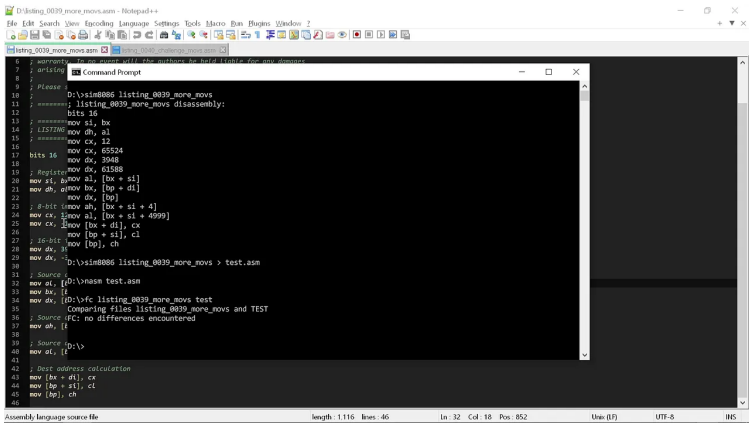
Well, it's because CPUs simply don't care about whether something is signed or unsigned until they actually do sign-sensitive operations on them. So CPUs don't think of registers as signed or unsigned in the first place. They just store bits, and if you want them to interpret those bits as a signed value, well, they'll do that *when you tell them to do a signed instruction.*

So when you move an immediate, there is actually no difference between a signed value like -12 and its unsigned counterpart, 65524. They're the same bits to the CPU, and they'll look *exactly the same* in the instruction stream.

When you disassemble this machine code, then, don't worry if you input -12 but you get back 65524! That's exactly what my reference disassembler does too:



How do we know this is correct? We use the trick I showed last time! If we just *reassemble* the disassembly, we can compare the binaries and be *sure* that we produced a valid disassembly indistinguishable from the original:



So that's it. That's all you *have* to do for homework. But for those of you who want more of a challenge, I am totally happy to provide such a challenge.

Listing forty is *much* harder than listing thirty-nine:

```
;
========================================
========================================
==
; LISTING 40
;
```

```
;========================================
;========================================
;

bits 16

; Signed displacements
mov ax, [bx + di - 37]
mov [si - 300], cx
mov dx, [bx - 32]

; Explicit sizes
mov [bp + di], byte 7
mov [di + 901], word 347

; Direct address
mov bp, [5]
mov bx, [3458]

; Memory-to-accumulator test
mov ax, [2555]
mov ax, [16]

; Accumulator-to-memory test
mov [2554], ax
mov [15], ax
```

The signed displacements listed here are the exact same instructions that appear in listing thirty-nine, but they include displacements are *negative*. To understand how to get that right in both the 8-bit and the 16-bit cases, you'll have to go read the "Machine Instruction Encoding and Decoding"

section of the 8086 manual carefully. Sign extension for displacements is specifically specified in the text, and you have to obey that specification.

I also added more types of mov instructions to this listing. There are explicit size tests, which actually tests two things: first, it tests whether your disassembly properly adds the “byte” and “word” keywords when necessary. Second, it tests whether you can decode this *additional* opcode for mov, since it is *not* the same opcode as the two we discussed on the lightboard!

This listing also contains those *direct address* moves I warned you about. These are the ones where MOD is 00 but RM is 110. If you didn’t handle that special case, you won’t be able to correctly disassemble this listing.

Finally, we have two special opcodes that actually don’t do anything you couldn’t already do with opcode 100010. These are memory reads and writes, but they can be encoded more compactly *specifically when the register is ax*.

Why do these instruction exists if they are redundant? Well, again, space was precious. A smaller instruction stream took up less of the expensive memory to store, and it took less of the expensive cycles to fetch, so instruction sets of this era were always looking for ways to be more efficient. This is just one way 8086 chose to try to be more compact.

So that's the challenge homework. You only have to do listing thirty-nine — that will be plenty! The point of this course is not to learn how to disassemble 8086 assembly, so once you've got the gist of it, that's good enough. As long as you understand how the process works in general, it is not necessary to know all the specifics.

However, it seemed like people were enjoying the homework, so, I wanted to make sure that people who wanted to play around with 8086 disassembly could keep on playing with it! That's what listing forty is for, so if you're enjoying these little exercises, that one is for you.

So that's it! I hope everyone enjoys the homework, and I'll be back here next time to discuss a few more things we should learn to decode.

---

1 And even though it has gone through multiple major revisions, it *still* retains this property where it is impossible to know how many bytes an instruction requires until you've read and tested potentially *multiple* bit sequences!

301 Likes · **4 Restacks**

176 Comments

Write a comment...

Kevin Howard Mar 5, 2023

Liked by Casey Muratori

Interesting that technically that one instruction is really,

mov dx, [bp + 0]

LIKE (16) REPLY (2) SHARE

Gaurav Gautam Gaurav's Substack Mar 13, 2023

When I first saw your comment I didn't get it. But then my program generated it too lol

LIKE (1) REPLY (1) SHARE

James Gibson Jun 7

+1

LIKE REPLY SHARE

rizoma0x00 Mar 6, 2023

SPOILER!!!

LIKE REPLY SHARE

Chris Hinton Mar 6, 2023 *Edited*

Liked by Casey Muratori

I had a bit of a laugh when my code was printing "mov dx, [bp + 0]" and realized that because of the exception for direct addresses, the encoding must include a displacement when loading from bp, even if it is zero.

LIKE (10) REPLY SHARE

Lee Mar 7, 2023 *Edited*

Liked by Casey Muratori

Implementation of latest listings in Jai.

https://github.com/schlag/sim8086

LIKE (9) REPLY (1) SHARE

dechichi Apr 3, 2023

Hope you don't mind me asking, but how do you got access to the private beta? I've been interested in trying Jai

LIKE (4) REPLY (1) SHARE

Aske BV Nov 13, 2023

In case other people are wondering, I got it through emailing language at thekla dot com describing my background, intentions and clearly emphasizing that I understand that it's

not supposed to be stable (even though it mostly is). And then be patient after that.

LIKE (4) REPLY SHARE

Andrew Saraev Mar 5, 2023 *Edited*

What a fun exercise. And an amazing course so far!

First time using C for me, trying to crack listing 40 now...

Spent some time searching for a separate sign control for displacement because the manual states at page 4-21 that (DISP-LO) and (DISP-HI) are bytes of an unsigned displacement...

But apparently they aren't? As the page 4-20 mentions "sign-extending" 8-bit displacement quantity to 16-bit.

LIKE (7) REPLY (5) SHARE

Christopher Galpin Weirdist Mar 6, 2023 *Edited*

I found these but got really confused for a while; reinforced by Casey saying it was "tricky". I was even searching for a MOV that used the S field (like ADD does lower in the manual), etc. Then I re-watched and realized Casey was talking about sign-extension ITSELF being tricky, not WHETHER to sign-extend here being tricky. I interpreted him wrong because brain in non-statically-typed-Python mode, doh. 🐍🧠💪

LIKE (3) REPLY SHARE

Paolo Mar 5, 2023

For 16 bit displacements, I suspect that within a 64k memory segment it makes no difference if you move by a negative offset

or if you move by the positive offset you get if you interpret the negative number as unsigned. This assumes that the memory index will roll over after 2^16-1 back to 0 within the segment... if that is indeed true then it may just be a convention to interpret the 16-bit displacement as signed for ease of readability.

LIKE (2) REPLY (1) SHARE

Andrew Saraev Mar 6, 2023

Huh, that is an interesting suggestion!

I just finished reading Game Engine Black Book: Wolfenstein 3D and it was interesting to find out about all the problems with addressing and arithmetics you have to deal with with 16-bit registers. I wonder if we'll have to implements some of it during the course, with all the segmentation.

LIKE REPLY SHARE

Connor A. Haskins Button Pusher Mar 11, 2023

So is the documentation just wrong? Confused as to why it would explicitly state that they are unsigned...

LIKE (1) REPLY (1) SHARE

Paolo Mar 11, 2023 *Edited*

Not necessarily. If indeed a 16 bit displacement in a memory segment acts as a circular buffer (i.e. if you are at address n and add 2^16 to you address you land back at address n), then it would not matter if you consider it signed or unsigned, as you would end up at the same location.

For example a displacement of -1 signed is the same as 2^16-1 unsigned, thus generating an displaced address of n-1 = n+(2^16-1) = (n + 2^16) -1 = n-1.

LIKE (6) REPLY SHARE

Gaurav Gautam Gaurav's Substack Mar 12, 2023

Ahh.... I am stuck guys. After reading the 8bit value that is the disp8 how do I know it is a signed number? Or is it always supposed to be a signed number? I'm not sure how to make it signed in javascript either..... yeahhhhhh Maybe I can used a typed array somehow or maybe Ill have to write a wasm function just for this. But is it that we just read all displacements as signed bits?

LIKE REPLY (1) SHARE

Andrew Saraev Mar 13, 2023

I interpreted them all as signed and it looks correct. So for the 8-bit case it's a [-128, 127] number. For the 16-bit case two bytes together represent a [-32768, 32767] number.

(Btw, this situation was also the first time the endianness showed itself to me plainly...)

I don't know what JavaScript allows you to do. As a last resort I guess you could just check if the number is greater than 127 and then subtract 256 (or greater than 32767 and then subtract 65536 for the 16-bit case).

LIKE (2) REPLY (1) SHARE

Gaurav Gautam Gaurav's Substack Mar 14, 2023

I got it working. In javascript you can cast integer types if you have the bits by putting it inside a typed array. I used an unsigned int8 typed array to read the raw bytes from the file and then wehenver I need to cast it I either put a single byte into a length 1 int8 array or I 'OR' two bytes after changing the sequence and then put it in a int16 array. Here is my implementation: https://gautam1168.github.io/Part2-8086/disassemble.html

LIKE (1) REPLY SHARE

Andrew Saraev Mar 6, 2023 *Edited*

Here's my current solution in C.

https://github.com/AndrewSaraev/perf-aware-homework/blob/08162a8a607d7dc08a7792017627e08fdf7be314/part_1/asm_decoder.c

LIKE REPLY SHARE

Tom Montgomery Mar 5, 2023

In listing 39, the "8-bit immediate-to-register" examples will generate a 16-bit mov as the register is cx. Still a good test that even though 12 will fit into a byte, it will move a word, but people might want to test a "mov cl, 12" as well.

LIKE (7) REPLY (1) SHARE

Casey Muratori Mar 5, 2023

Author

Yes, sorry about that. I totally forgot to include an 8-bit register mov :(That was my intention but I didn't do it. I will update

the files to include it!

- Casey

LIKE (4) REPLY (1) SHARE

James McKinney Mar 7, 2023

Another good test to add is "mov al, [8]" (my code had assumed the accumulator was always ax).

LIKE REPLY (1) SHARE

Valentin Ignatev Jun 6, 2023

Hah, I actually handled al/ax right away and only then realized that the test code doesn't have it :)

LIKE REPLY SHARE

David Larsson Mar 5, 2023

Argh! Opcodes can be different widths? Ah, well, will be interesting to work on. :)

LIKE (6) REPLY (1) SHARE

Casey Muratori Mar 5, 2023

Author

Welcome to x86 :) If you can parse this instruction stream, you can parse _any_ instruction stream!

- Casey

LIKE (13) REPLY SHARE

TJ Mar 5, 2023

Liked by Casey Muratori

Well, I didn't think that I would have so much fun doing these assignments.

Here is my solution https://pastebin.com/jqazmWCT in python.

Thank you for making this!

LIKE (1) REPLY SHARE

Nicolas Vidal Mar 5, 2023 *Edited*

Very fun exercises, my (ugly and naive) solutions there in Rust : https://github.com/NicolasVidal/rust_computer_enhance_exercises

Anyway, I was wondering, can we assume in the real world, the printing on your T-shirt is printed backwards ? ;)

LIKE (5) REPLY (3) SHARE

photobaric Mar 7, 2023

Here's another one in Rust, making use of macros and with zero allocations:

https://gist.github.com/hzuo/e274212ab1fc066889ff13759c9fde70

LIKE (2) REPLY (2) SHARE

James McKinney Mar 7, 2023 *Edited*

Nice! This helped me catch a bug in my code (listing 40 didn't have an accumulator like mov al, [8]).

Edit: It inspired me to eliminate some allocations (with a tiny bit more duplication) – but I've kept the allocations in my MOD logic as I didn't want to duplicate so much 😅

LIKE (1) REPLY SHARE

Nicolas Vidal Mar 7, 2023

Nice one, quite clean :)

LIKE (1) REPLY SHARE

Dmitry Mar 6, 2023

Nice one! I've desided to try out bitfields in Rust, and while it works I'd prefer not to

tackle this again https://gist.github.com/9dadb14ecb91682d6b8447a0f928b1cb

LIKE REPLY (1) SHARE

James McKinney Mar 6, 2023

Ah, yeah, it seems simpler to just use bit operations and &str array constants.

LIKE REPLY (1) SHARE

Dmitry Mar 6, 2023

Yeah, I'd probably still leave enums & Display there (probably would join the registers, etc), but modular bitfields hadn't delivered as much as I hoped. Still nice. I remember writing microOS in the university and didn't enjoy how C doesn't support not-byte-aligned bitfield structs yet this one supports those. That would've helped with all the paging.

LIKE REPLY SHARE

James McKinney Mar 6, 2023

Here's mine in Rust: https://github.com/jpmckinney/computer_enhance/blob/main/src/main.rs

LIKE REPLY SHARE

Ameen Mar 5, 2023

I can't find where the manual talks about the signed displacement? The only thing I found is the fact it sign-extends 8bit displacement to 16bit.

Which means I just need to cast it to signed 8bit/16bit before printing it.

LIKE (4) REPLY (1) SHARE

Casey Muratori Mar 5, 2023

Author

That is correct! That is all you needed to catch.

- Casey

LIKE (6) REPLY (3) SHARE

Gaurav Gautam Gaurav's Substack Mar 12, 2023

Update: I figured it out. I just needed to create a signed typed array and put the byte/bytes inside it and take it back out. It would probably be best to lean into webassembly now but I want to see how far I can push javascript.

I made the assignment solution by just taking all the numbers from 0 to max of unsigned 16bit int and treating them as instructions to generate a table. Its here: https://github.com/gautam1168/gautam1168.github.io/blob/dev/Part2-8086/decodetable.txt

Is this not what you intended? Everyone seems to be decoding the instructions and I feel maybe there is a reason I shouldn't be using a lookup table? (I only generated it for MOV so far, going to do the rest now).

LIKE (1) REPLY (1) SHARE

Ethan Mar 16, 2023

I hadn't thought to use a lookup table like that. Honestly that's probably cleaner than the code I

wrote

LIKE REPLY (1) SHARE

Gaurav Gautam Gaurav's Substack Mar 16, 2023 *Edited* ···

I have it almost working along with the challenge homework of listing 41. It turned out I needed 2 lookup tables instead of 1. One for 0 to 255 (1 byte) and another one for the rest. This is because in many instructions the leading 0's are meaningful so you cannot just left pad a 1 byte instruction to be 2 bytes. It causes collisions that are hard to keep straight. But 2 tables is not that much worse. The updated implementation is here: https://gautam1168.github.io/Part3-8086/disassemble.html

and tables are here: https://github.com/gautam1168/gautam1168.github.io/blob/main/Part3-8086/decodetable1byte.txt

LIKE REPLY SHARE

Gaurav Gautam Gaurav's Substack Mar 12, 2023

Also, I just copied the asm files from your repo. One other person asked this but please tell us in monday qna if its not correct to put those files in our repos. Ill remove it if so.

LIKE REPLY SHARE

Gaurav Gautam Gaurav's Substack Mar 12, 2023

how do i even do this in javascript!

LIKE REPLY SHARE

Dmitry Mar 6, 2023

Man 8086 is annoying. Why accumulator-memory mov has 7th bit being "implied reg IS NOT the destination" while in memreg-memreg one it is "reg IS the destination"?

LIKE (3) REPLY (2) SHARE

Casey Muratori Mar 7, 2023

Author

I had that exact same question. It's like they just wanted to make us add one more case for no reason :)

I assume there is a hardware reason for it (like it is easier to implement the circuits that way, or something), but it's certainly weird when looking at it purely in software!

- Casey

LIKE (2) REPLY SHARE

Cory Duplantis Mar 6, 2023

What helped me is to think of "memory to accumulator" and "accumulator to memory" as two separate opcodes.

"Memory to accumulator" means "ax = [mem]" and it has the opcode 0b1010000

(no d flag).

"Accumulator to memory" means "[mem] = ax" and it has the opcode 0b1010001 (no d flag).

It is a little confusing coming off of the first "register/memory to/from register" operation that has a bunch of other bitflags.

LIKE REPLY (1) SHARE

James McKinney Mar 6, 2023

Liked by Casey Muratori

For the accumulator opcodes, I treated them as a single opcode and invented an e flag that has the opposite meaning of d ;)

LIKE (1) REPLY (1) SHARE

Dmitry Mar 7, 2023

Yeah, I did the same and was so suprised by the inversion that I renamed the bit to reg_is_dst and reg_is_src.

LIKE REPLY SHARE

not-quite-country Mar 5, 2023

next round of pain: decoding all the (non-segment) MOVs -- the hard way.

https://github.com/not-quite-country/pap/blob/main/1_2.asm

Passes listings 39 and 40 byte-for-byte assemble/disassemble round trip. Super janky hex output routines, but gets the job done for this round.

LIKE (3) REPLY SHARE

Yakvi Mar 7, 2023 *Edited*

It was quite surprising to discover that "mov dx, [bp]" is encoded as MOD 1 by nasm. Shouldn't it be mode 0 with special handling as per instruction manual? My silly implementation in C# (listing 39 compatible): https://github.com/Yakvi/sim8086

LIKE (2) REPLY (1) SHARE

not-quite-country Mar 7, 2023

The issue is that Mod = 00b, R/M = 110b, which you would expect to be [BP], is special cased to encode a 16-bit absolute address. So there isn't a way to encode [BP] without a displacement, and [BP+0] with 8-bit displacement is as close as you can get. That BP is chosen to suffer this fate makes more sense if you consider the traditional use of BP as a frame pointer, which makes [BP+0] a rarely needed effective address. i.e. mosts uses of BP in it's intended role are going to be [BP-off].

LIKE (4) REPLY (1) SHARE

Yakvi Mar 8, 2023

Ah that makes sense, thanks for clarifying!

LIKE REPLY SHARE

JoeTDC Mar 5, 2023

Anyone else spotted the error in the 8086 manual on page 4-32? I think B3 should be 1011 0011, but is printed as 1011 1011

LIKE (2) REPLY (1) SHARE

Cory Duplantis Mar 6, 2023

Someone above also saw that instruction A3 is also wrong. "mov MEM16, al" should

be "mov MEM16, ax"

LIKE (1) REPLY SHARE

Marcel Mikołajko Marcel's Substack Mar 5, 2023

Here's my version of the homework: https://github.com/OrangeLightning219/pap/blob/main/8086/main.jai

This was very fun to do :)

LIKE (2) REPLY SHARE

Mateusz Radomski Mar 5, 2023

It's interesting that places like https://www.felixcloutier.com/x86/mov treat for example the 0b100010 6bit opcode as different instructions depending on the value of D and W bits. Which is quite useful to simplify the dynamic nature of those prefixes I guess

LIKE (2) REPLY SHARE

Gaurav Gautam Gaurav's Substack Mar 5, 2023 *Edited*

Download link of mp4 if anyone needs it:
[Removed because substack doesn't want it to be posted]

LIKE (2) REPLY (1) SHARE

Casey Muratori Mar 5, 2023 *Edited*

Author

I don't think Substack wants you posting those, just FYI. Those are CDN keys and they have to pay for that bandwidth (so they do not want plain links being posted that non-subscribers can use to download videos, etc.)

- Casey

LIKE REPLY (2) SHARE

Gaurav Gautam Gaurav's Substack Mar 5, 2023 *Edited* ···

I see. Ill remove it then. If anyone wants the link, its easy enough:

1. Open the post whose video you wish to download and open the network tab in the browser.

2. Add a filter `hls` in the network pane and then reload the page. You will see one network call like this https://www.computerenhance.com/api/v1/video/upload/<uuid>/src?override_publication_id=<pubid>&type=hls

3. Change the type parameter at the end of this from hls to mp4 and paste that link in a new tab, which should redirect you to the mp4 link

You can also just paste that hls link into a new tab to get a .m3u8 file which you can then open in vlc media player. If you google a little bit, you will be able to figure out how to save that video as mp4 in your disk.

LIKE (6) REPLY (2) SHARE

Casey Muratori Mar 5, 2023 ···

Author

Actually no, I think it is OK to tell subscribers how to get the link themselves. Because non-subscribers won't be able to use that. So that seems fine to me? I don't think they would mind that.

- Casey

LIKE (1) REPLY (1) SHARE

Gaurav Gautam Gaurav's Substack Mar 5, 2023

alright Ill add it back then

LIKE REPLY SHARE

Ilija Tatalovic Mar 5, 2023

My tip: After download, immediately re-encode (Handbrake/H.265 preset). It cuts size to almost 1/3 without noticeable decrease in quality.

LIKE REPLY SHARE

Gaurav Gautam Gaurav's Substack Mar 5, 2023

Why do you sign your name like that in posts? Its like Captain Holt in Brooklyn 99 lol

LIKE (3) REPLY SHARE

Spencer Gehin Mar 16, 2023 *Edited*

Not going to bother posting all of my work on the homework since I'm playing catch-up, but I am rather proud of the common lisp format monstrosity I've created to handle the general case for effective address printing (mode 3 and direct addressing are handled in a real branch because I'm not that insane):

(format nil "[~[BX + SI~;BX + DI~;BP + SI~;BP + DI~;SI~;DI~;BP~;BX~]~:[~;~[~; + #o~3,'0O~; + #x~2,'0X~]~]]" r/m (not (zerop disp)) mode disp)

LIKE (1) REPLY SHARE

kroveit Mar 12, 2023

Shouldn't it be MOD=00 instead of MOD=11 in "But it's the slot they chose for the direct memory address encoding, so that's what MOD=11 R/M=110 decodes to instead."?

LIKE (1) REPLY SHARE

Gaurav Gautam Gaurav's Substack Mar 11, 2023

Why don't we just make a huge lookup table so we don't have to parse anything except the Displacements and Immediates?

For example the mov instruction from the manual

MOV -> 100010dw, mod/reg/rm, disp-lo, disp-hi

can be broken down to

100010 0 0, 00 000 000 -> MOV AL, [BX+SI]

100010 0 1, 00 000 000 -> MOV AX, [BX+SI]

100010 1 0, 00 000 000 -> MOV [BX + SI], AL

100010 1 1, 00 000 000 -> MOV [BX + SI], AX

100010 0 0, 01 000 000, D8 -> MOV AL, [BX + SI + D8]

100010 0 1, 01 000 000, D8 -> MOV AX, [BX + SI + D8]

100010 1 0, 01 000 000, D8 -> MOV [BX + SI + D8], AL

100010 1 1, 01 000 000, D8 -> MOV [BX + SI + D8], AX

100010 0 0, 10 000 000, D16-1, D16-2 -> MOV AL, [BX + SI + D16]

100010 0 1, 10 000 000, D16-1, D16-2 -> MOV AX, [BX + SI + D16]

100010 1 0, 10 000 000, D16-1, D16-2 -> MOV [BX + SI + D16], AL

100010 1 1, 10 000 000, D16-1, D16-2 -> MOV [BX + SI + D16], AX

and so on...

And this way we don't have to decode the MOD, REG, R/M etc. All we read are the displacements

and immediates etc. I see in the next video it was shown that there are repeating patterns in the first and second bytes and it was hinted that those can be exploited to get more for free. However, am I missing something here? Can't I just treat the first 2 bytes as indexes into a table that tells me what is the instruction?

LIKE (1) REPLY (2) SHARE

Gaurav Gautam Gaurav's Substack Mar 12, 2023

Update: So I was able to generate this table. I just generated an array of integers from 0 to 65536 and then split the 2 bytes. Then I can just decode the bytes as if they are parts of the MOV instruction. And I put the generated assembly template like this into a `translation key`. Once this process is completed I can just go through the bytes in a binary and read off the assembly template from the `translation key`. I still haven't completed the part that will read off the next bytes for displacements and immediates but Im working on that. I am behind on the homework but I think Ill get all the instructions in one swoop once I get through this.

LIKE REPLY SHARE

Gaurav Gautam Gaurav's Substack Mar 11, 2023

Ok I tried doing this. I made a file to represent the table and I added about 200 entries in it. But for the whole thing its 16bits so 65536 possible rows. Which will take me more time than I want to spend on this. But maybe I can try to generate that table using another program first and then use it to decode. Ill go and try that now.

LIKE REPLY (1) SHARE

Rase Apr 13, 2023

I made something similar in the first homework until I realized how unmaintainable it was. In the real world though it could work since you have a constant amount of instructions you have to decode that you know in advance but for the sake of doing the homework one by one it was too much to maintain. https://github.com/52617365/intel8086-decoder/blob/master/src/main.rs

LIKE REPLY (1) SHARE

Rase Apr 13, 2023

Also, I tried to make it into a lookup table instead of a huge branching function but I didn't know how to do that in rust.

LIKE REPLY (1) SHARE

Rase Apr 13, 2023

I guess I could of had a massive array and index into it with the byte casted into a usize. That way I could of avoided all branching.

LIKE REPLY (1) SHARE

Gaurav Gautam Gaurav's Substack Apr 13, 2023

i got it to work in order to do the challenge homework with all the instructions. But I didn't take it any further after the reference decoder was available.

LIKE REPLY SHARE

David Larsson Mar 8, 2023

Updated my gist to work for both listing 39 and 40 (and for previous listings): https://gist.github.com/DavidLarsson/a92e92a1d27236b4c0152d0e89031fcd

Was a bit tricky to realize that the 16-bit displacement codes to the same thing whether it's signed or unsigned. But as someone else commented, it makes sense once you realize that it's a wraparound in the 16-bit memory. Still not entirely sure I got the sign-extension correct.

LIKE (1) REPLY SHARE

Pierre Mar 7, 2023

I can feel my code spaghettifying before my eyes! Had to do one initial refactor to achieve correctness, but really probably should do more.

I'm trying to iterate 1 byte at a time inside a loop, but starting to think this was not a good approach. I need to maintain information about the state seen so far, which adds complexity. I wonder if there is a good way of doing it like

this. I originally did this to avoid ever parsing a byte that didn't exist in the file, but now I'm not sure if this was a good trade-off.

Is it just is always better to do bytes[i], bytes[i+1] etc (with some kind of bounds check to make sure that you don't go outside of the bytes array in cases of a malformed file)

LIKE (1) REPLY (1) SHARE

Gordon Mar 7, 2023

I read one byte a time into a 6 byte buffer (max instruction length). My logic is basically:

while true:

1. Read 2 bytes (return if no bytes left)

2. Figure out if we need to read more bytes by parsing opcode, and, if applicable, mod and w.

3. Read any extra bytes we need into the buffer

4. Print the assembly instruction, parsing whatever data from the buffer is needed and hasnt been parsed yet

As you mentioned, some vars that are assigned in 2 need to be referred to in 4.

LIKE (1) REPLY (1) SHARE

Pierre Mar 8, 2023

Hmm, yeah that's an interesting approach. On similar lines, I was wondering if it might be a good plan to slide a window of 6 bytes along the array (would need some padding to deal with the edges). Something like:

while pos < endOfBuffer:

1. Read 6 bytes from current pos

2. Do whatever parsing you can on that 6 byte area

3. If there was some parsing problem (e.g. it's a 6 byte instruction but only 4 bytes were in the file), bail.

4. Increment current pos by whatever size the instruction turned out to be

Maybe that would handle the sort of thing Casey spoke about in his Q&A, where you could parallelise it by speculatively parsing several different increments of 'pos' at once and then throwing away the ones that didn't turn out to be at the right offsets,

LIKE REPLY (1) SHARE

Gordon Mar 10, 2023

Interesting - I am guessing if you just drop into the middle of the binary without the context of what proceeded you can't definitively find an instruction boundary. Perhaps you could do an initial pass to find all the instruction boundaries and then do the rest of the parse in parallel.

LIKE REPLY SHARE

Lorenz Mar 6, 2023

I was watching the videos as they came out but only read the articles just now - I like to read in bulk. I took note of a couple typos as I went, this is what I noticed:

Instructions Per Clock:

- Usually loops do a more operations

Mutlithreading:

- You already saw me to do things

Python Revisited:

- guide your decisions are the high level

- learning what's the fast could would be

Decoding Multiple Instructions and Suffixes:

- The [75] is jus like looking up

I hope I don't come off as rude - just pointing these out so they can be fixed. Typos are always easier to find for a third party than the original author - at least that's how I feel about my own writing.

LIKE (1) REPLY (1) SHARE

Casey Muratori Mar 7, 2023

Author

Always helpful, yes - I haven't had a time to go back and re-edit the old articles to remove typos yet, but I appreciate that people have kept track of them! I do plan to do it soon.

- Casey

LIKE REPLY SHARE

James McKinney Mar 6, 2023 *Edited*

Can the assembly ever construct an "immediate to register/memory" with mod 11, or will that always be more compact as an "Immediate to register"? I noticed some solutions that didn't handle the first case, but I can't get nasm to generate it. I can manually create a test file with the bytes 0b11000110, 0b11000000, 0b00000001 (equivalent to "mov al, byte 1", though if I give nasm that as input, it generates the equivalent

of "mov al, 1", which saves one byte).

LIKE (1) REPLY SHARE

Cory Duplantis Mar 6, 2023

Starting the 8086 emulator with the assumption we'll be emulating instructions in the future. Lifting the instructions into a form that would make it easy to emulate and print the disassembly. Writing emulators is just a blast, so I'm glad we're starting here!

https://github.com/ctfhacker/performance_aware_programming/blob/master/emu8086/src/decoder.rs#L24

LIKE (1) REPLY SHARE

iobt Mar 6, 2023

With an underlying theme being that instruction decoding can be tricky and requires some twiddling, does the "promised" performance in a CPU spec (i.e. the clock cycles, 4.7GHz for instance) exclude the overhead of decoding these instructions? Or is instruction decoding these days a more trivially solved problem and hence has negligible overhead?

LIKE (1) REPLY SHARE

Christopher Galpin Weirdist Mar 6, 2023 *Edited*

Semantically compressed Python (3.10) Listing 40 and 39 https://gist.github.com/CodeOptimist/fe665af42f80df648e114df0a79b7a52

LIKE (1) REPLY SHARE

Pulak Malhotra Mar 5, 2023

My solutions for listing 39 and 40 using typescript (with Deno).

https://github.com/MalhotraPulak/

Performance-Aware-Programming/blob/main/Assignments/Assignment2/solution.ts

Thank you so much for the course Casey!

LIKE (1) REPLY SHARE

Seth Mar 5, 2023 *Edited*

My solution for Listing 39:

https://pastebin.com/tdauWrAA

Now that we see that opcodes take up different bits, I got rid of the structs that I was using before that are no longer very useful, since the structure of the data changes between opcodes.

Also got rid of the idea of using the d flag as an array index to adjust the source / destination since I found it better to be clear than to avoid the if statement.

The biggest challenge for me was combining two 8 bit ints into one 16 bit int. Did not expect to need the use of a bitmask on the lower value to do that properly.. To be honest I'm still a little confused what kinds of situations I have to be extra careful and use a 0b11111111 bit mask.. It seems like sometimes I don't need it, and sometimes I do.

Update: Here's my solution for listing 40:

https://pastebin.com/cRYrCZXd

This was much harder! Converting two 8 bits

Expand full comment

LIKE (1) REPLY (1) SHARE

Seth Mar 6, 2023

This course is really bringing out some obsessiveness in me!

Updated: https://pastebin.com/4pPXT2ZL

I've come to embrace special types from stdint.h to develop a stronger mental model of what's going on with the bytes. I see now that the issue comes when assigning 8 bits to a 16 bit variable. When the number is negative, due to two's complement, you end up affecting all 16 bits. So to stop this from happening you have to & 0x00ff, to cut off those bits, so you can drop in the other byte that's supposed to live there.

Of course there are plenty of times when we only need the 8 bits, in which case it's best to declare variables that are just 8 bits. But often we've got to go with 16 bits for code simplicity, since the "w" flag can increase the width, and it's nice to continue to use the same variable for the 16 bit result, just adding the bits on to the other 8 bits. So 16 by default then..

LIKE REPLY SHARE

Jeroen Mar 5, 2023

My solution for listing 39: https://pastebin.com/B3n5kNJ0 (C++)

Decoding is getting pretty complicated really quickly! Since you write listing 40 is optional and much more difficult, I'm happy to skip it :)

LIKE (1) REPLY SHARE

TatriX Mar 5, 2023 *Edited*

```
; 8-bit immediate-to-register

mov cx, 12

mov cx, -12
```

It looks like `w` bit is set for these instructions. I tried both yasm and nasm. Is it how it is supposed to be or am I missing something?

edit: based on Tom Montgomery's comment

```

; 8-bit immediate-to-register

mov cl, 12

mov ch, -12

```

These will generate `w` = 0.

edit2: here's listing-39 https://github.com/TatriX/pap/blob/e185bd1f39fda9789947adf1bb7e31dc9dee1dc7/c8086/c8086.c#L93

edit3: listing-40: https://github.com/TatriX/pap/

LIKE (1) REPLY (1) SHARE

Mateusz Radomski Mar 5, 2023

From what I understand cx is a 16bit register so it's always going to encode the 'w' bit, try changing those to cl or ch to get 'w' bit set to zero

LIKE REPLY SHARE

Duck Dodgers Mar 5, 2023

Hm, I wonder why they decided to list "Accumulator to Memory" and "Memory to Accumulator" as different things in the manual, when it seems to me that the 7th bit works exactly like the D flag in the previous ones.

LIKE (1) REPLY (2) SHARE

Casey Muratori Mar 5, 2023

Author

When I first read the table I thought that too, and naively parsed it the same way, and got an error. Then I looked closer and realized they had _inverted_ the meaning of the D bit just for that instruction! WHY?? I have no idea why they would do that. But it ends up meaning you can't rely on the same D bit parsing as all the other movs :(

- Casey

LIKE (3) REPLY SHARE

Tom Montgomery Mar 5, 2023 *Edited*

Edited as made a mistake before:

The 'd' field in the manual says that if true the 'reg' field is the destination, but for this op, the bit that would be 'd' is true if the memory is the destination, so I guess that's why. Hopefully I've not got this mixed up again :).

LIKE REPLY (1) SHARE

Ameen Mar 5, 2023 *Edited*

Thanks, you are correct! The bit here is opposite to 'd'.

But now I'm wondering why it is opposite :)

LIKE (1) REPLY (1) SHARE

Tom Montgomery Mar 5, 2023

Yeah, I'd be interested to find out if there was a good reason for it, or they just didn't think to give it the same meaning as the 'd' field.

LIKE REPLY SHARE

Todor Nov 30

Amazing, I made a number of assumptions

while doing the previous homework, and all of them were wrong! Everything was table-driven af, and there were no conditionals except for I/O. And now I'm like... do I add exponentially more tables, or do I turn it into a Balatro-style forest of if-else? And now imagine you gotta do 32 and 64 bit instructions.. oh my!

LIKE REPLY SHARE

Casey Coleman Nov 5 *Edited*

Listings 37-40 in Common Lisp:

https://gitlab.com/performance-aware-design/pad/-/releases/pad-listing-40-final

The trickiest part was finding an extensible approach, but once I had a workable solution to listing 39, listing 40 was free.

LIKE REPLY SHARE

Andrés Sep 20

Hi, i don't seem to understand why the instruction

mov cx, 12 gets assembled to:

10111001 00001100 00000000

In my understanding, de first four bits tell us that its an immediate to register. Then, as the w bit is set to one, we should take the following two bytes, which would mean that the immediate es 3072, nonetheless it's like it only reads the first following byte (12). We have the w bit turned on and a complete register (CX).

LIKE REPLY SHARE

Anthony Verdi Aug 3

Really enjoyed this HW assignment + challenge, here is my solution in python: https://github.com/vrdiy/8086_decoder/blob/main/

decode_8086.py I also added tests so I can automate checking my decoder against your provided binaries :)

LIKE REPLY SHARE

Ankit Pariyar Jun 29

He mentioned that -12 and 12 should be the same when assembled. However, my assembly results seem different.

For example, consider the following assembly code:

```

mov cx, 12

mov cx, -12

mov dx, 3948

mov dx, -3948
```

When I read the assembled file into a `Vec<u8>` in Rust, I got the following bytes:

```

[185, 12, 0, 185, 244, 255, 186, 108, 15, 186, 148, 240]
```

But if I remove the "-" sign, I got this:

```

[185, 12, 0, 185, 12, 0, 186, 108, 15, 186, 108, 15]
```

LIKE REPLY (1) SHARE

jee zee Jul 2

What he was saying is that you will get 244 instead of -12 and those are same in binary 0b11110100. Converting negative binary value to signed decimal representation is a

bit more than removing leftmost bit. Look it up. Also for larger numbers that take up 2 bytes you have to take in account endianness (yeah, I also just learned that word :)

LIKE (1) REPLY (1) SHARE

Ankit Pariyar Jul 2

Thank you, I am really struggling on parsing negative numbers. I think I do understand it now, hope I can implement on code too

LIKE (1) REPLY SHARE

jee zee Jun 28

My solution in JS https://github.com/half-cto/computer_enhance/blob/main/part_1/002-more_movs/8086_decoder_v2.js

Couple bonus things learned - how signed numbers are stored in binary, endianness.

LIKE REPLY SHARE

Alejandro Guillamon Weekly Wander | KNblog Jun 24 *Edited*

Just started the class and loving it.

Here is listing 39 and 40 in golang.

https://github.com/alexguillamon/computer_enhance/blob/main/perfaware/homework/0002_instruction_decoding/main.go

LIKE REPLY SHARE

disusdev disusdev Jun 14

This bit pattern looks sus: https://ibb.co/BZXqf1R

LIKE REPLY SHARE

ari zablozki May 30

Doing these first few Homeworks feels like programming to me(and it's fun as can be), Is this what programming is supposed to feel like ?

LIKE REPLY SHARE

Rafael Áquila May 4

Does someone knows a production level 8086 disassembler made with C to compare?

LIKE REPLY SHARE

Rafael Áquila May 4

Loved! I'm going to go back to listing 40 soon for sure.

LIKE REPLY SHARE

Pedro Apr 11 *Edited*

My solution has way too many If statements... is this normal?

LIKE REPLY SHARE

Gilles Ramstein Feb 29

My solutions in Nim:

https://github.com/GillesRamstein/Performance-aware-programming

LIKE REPLY SHARE

Pablo Sanchez Dec 9, 2023

Watching this course I cannot avoid to think about Margaret Hamilton and her work on the Apollo program... Damn ,that's legendary!

LIKE REPLY SHARE

Daniel Sep 11, 2023

Doing this homework was my first clash with the c type system sign shenanigans, sure was an experience

LIKE REPLY SHARE

ahrav Sep 3, 2023

My Go solution: https://gist.github.com/ahrav/6f586cf5ff7f896dddf9db74ba64f70a

LIKE REPLY SHARE

Anshul Sanghi Anshul Sanghi Aug 31, 2023 *Edited*

Was finally able to finish the homework for this after a couple of days. I wrote the solution in rust and I also added some additional test cases based on Tom Montgomery's & James McKinney's comments about some cases not covered in original listings. My solution also covers both listing 39 and listing 40.

https://github.com/anshap1719-performance-aware/instruction-decoding-8086

LIKE REPLY SHARE

Clyde Williams Aug 8, 2023 *Edited*

My solution including challenges (C++): https://pastebin.com/sEYpUdMg

This was pretty fiddly, but I'm pretty happy with the API I have going here so far; it's quite easy to add new logic for each instruction, and I can merge the similar parts together into smaller functions as I go. We'll see how it holds up when we get past just MOVs but I'm satisfied so far.

LIKE REPLY SHARE

gbjn - brenno Jun 13, 2023

My attempt in JavaScript, doesn't handle listing 40 yet https://github.com/gbjnewman/lab/blob/main/computerEnhance/partOne/decoding_multiple_instructions_and_suffixes/sim8086.js

LIKE REPLY SHARE

Stephanie Jura Jun 7, 2023 *Edited*

Coming in awfully late, but I'm either doing something wrong (entirely possible), or the third entry in 39 wasn't updated in the assembled version on github. I'm feeding it to my decoder, and getting the initial byte of that instruction as 10110001. The opcode and reg fields are correct there, but w is still set to 0.

EDIT: Oh, I see the problem. The version in the post says the register is cx, so when I kept getting cl - which is what it's shown as even in the .asm version - I got confused.

LIKE REPLY SHARE

xDahl May 27, 2023

Wrote my implementation in C, in a fairly hacky way. But it was a LOT of fun!

https://github.com/xDahl/computerenhance_hw_8086_mov

Would have loved to have done it in Odin, but I'm still too new to it.

LIKE REPLY SHARE

Christian May 18, 2023

Catching up :) Zig solution to Homework 2:

https://github.com/xtian/performance-aware/blob/21cfc60217eddc00b90ff4f1a8cd12278718155e/sim8086/src/main.zig

LIKE REPLY SHARE

Alex V May 18, 2023

Great fun. My favourite bug was solved by figuring out where bit-shifting fits inside the order of operations. Which as it turns out, was

not where I thought it was...

LIKE REPLY SHARE

Soumitra Goswami Apr 21, 2023

Enjoying the Homeworks so far. Here is HW2

https://github.com/som1990/ComputerEnhance/blob/master/HW2/SG_HW2.py

LIKE REPLY SHARE

Soumitra Goswami Apr 21, 2023

Implementation using Python. Finally had some time to do some Homeworks.

https://github.com/som1990/ComputerEnhance/blob/ecdf396afb687aa67d12f9e4d16ccd74805fa08d/HW1/SG_HW1.py

LIKE REPLY SHARE

Christopher Bentley Apr 19, 2023 *Edited*

Using this as a github

https://pastebin.com/2LwHvseP

LIKE REPLY SHARE

Rase Apr 16, 2023 *Edited*

My rust solution :) (I'm just printing the results and comparing it manually) https://github.com/52617365/intel8086-decoder/blob/listing_0039/src/main.rs

LIKE REPLY SHARE

psygo Apr 14, 2023

Please offer the code listings with bigger fonts, it's just too small.

LIKE REPLY SHARE

Vlad Pavliuk Apr 12, 2023 *Edited*

So weird, when I try to compile:

mov [si - 300], cx

I get an instruction that starts with opcode:

11000010

Based on documentation it's the opcode for "Within seg adding immed to SP"

But if I change -300 to anything else (like -299 or -301) I get the expected mov instruction.

Does anybody know what's going on?? Did anyone else had the same issue?

LIKE REPLY (2) SHARE

Vlad Pavliuk Apr 13, 2023

Finally made it! My solution is probably the worst (500 lines of code), but it includes the tough part as well =)

https://github.com/VladPavliuk/JustCppHomework

LIKE REPLY SHARE

Vlad Pavliuk Apr 12, 2023

Oh, it seems that I found a bug in Notepad++ HexViewer plugin, lol.

Maybe it works like: binary file -> text (converts with error) -> binary on screen.

I wasted around 5 hours on this =(

LIKE REPLY SHARE

dechichi Apr 3, 2023

Sharing my solution in C# (listing 39 and 40): https://github.com/gabrieldechichi/programming-studies/blob/main/computer-enhance/perfaware/listing-39-40/Decoder8086.cs

LIKE REPLY SHARE

CJ Apr 1, 2023

Finally got some time to do the homework, really fun! my solution in Go https://github.com/cj1128/perfaware-review/blob/master/03.05/sim0086.go

I added a `-check` mode so it can automatically check the correctness by invoking `nasm` and then compare the binary.

LIKE REPLY SHARE

Kyle Mar 16, 2023

Finished mine: https://gist.github.com/kpence/ac634db2184741853df8f16cfb324d9f

LIKE REPLY SHARE

Gaurav Gautam Gaurav's Substack Mar 13, 2023 *Edited*

When generating the asm for the challenge listing in listing 40, how did you get the thing to ouput the `byte` or `word` in front of the immediate? If I just output the prefix whenever its the 2 byte immediate then i will always get word even if it was not specified in the source. Is that what we have to do?

I'm going to move on for now to the next episode. Here is my javascript implementation for 39 and 49: https://gautam1168.github.io/Part2-8086/disassemble.html

LIKE REPLY SHARE

kroveit Mar 12, 2023

Without handling a special case for MOD=00 and R/M=110, I am still able to get a disassembled version which when run in nasm produces an identical content. Anyone could tell me where is the mistake in my solution

https://pastebin.com/5LPzAKRM?

LIKE REPLY SHARE

SMK Mar 11, 2023

Here's my solution in Elixir (includes the extra challenge): https://gist.github.com/seemk/0ef3b0b063e571a82e946df61a29b5c9

LIKE REPLY SHARE

Jim Mar 10, 2023 *Edited*

Took me more hours than I care to admit, but I finally got it working for listings 39 and 40!

There were a few cases (like with the explicit sizes) where I wasn't 100% sure how to handle things, so I ended up writing my own one-instruction assembly files, seeing what would and wouldn't assemble, and then comparing my binaries to the homework binaries by hand. This helped me a lot!

https://github.com/zyusouken/perfaware_homework/blob/main/hw02%20(Decoding%20Multiple%20Instructions%20and%20Suffixes)/hw02.c

Now off to watch the new video.

LIKE REPLY SHARE

Connor A. Haskins Button Pusher Mar 10, 2023

Heads up to anyone who might encounter an error from NASM that looks like the following:

```

.\test.asm:1: warning: label alone on a line without a colon might be in error [-w+label-orphan]

.\test.asm:31: error: label or instruction expected at start of line

.\test.asm:212: error: invalid directive line
```

```

My console output seemed great, so I just redirected the console output to a file like Casey and everything seemingly worked great. The file looked perfectly as expected. However, when I ran NASM, I was getting these ridiculous errors. I did some snooping around and realized that my .asm files were oddly about twice as big as they should be. I eventually turned my attention to the fact that I was using powershell. I cannot tell you what exactly powershell is doing, but running "main.exe > test.asm" in powershell was creating a file of 682 bytes. While running the same command in

LIKE REPLY SHARE

**Said Al Attrach** Mar 10, 2023

omg was this exhausting, but totally fun and worth it.. maybe.. I think..

https://github.com/Said6289/perf_aware_course/blob/main/instruction_decode/decode_mov.asm

LIKE REPLY SHARE

**JoeTDC** Mar 9, 2023

Decided to continue on decoding other instructions and I just found out about the Segment Override Prefix, good lord.

LIKE REPLY SHARE

**Steven Ulin** Mar 9, 2023 *Edited*

Here's my cpp solution (listing 39). Things get pretty convoluted quickly but I think I did a good job containing the complexity.

https://github.com/SteveUlin/performance-aware-programming/blob/89481cc91b919f2b431c3713d836b199237be5
```

21/main.cpp

LIKE REPLY SHARE

Thom Mar 8, 2023

https://github.com/tgaldi/ComputerEnhance/tree/master/projects/part1/src

LIKE REPLY SHARE

Akmubi Lin Mar 8, 2023

A bit late on homework. https://github.com/akmubi/decoder8086

LIKE REPLY SHARE

Bruno Mar 8, 2023

Yatah! I did it ... in Java:

https://gist.github.com/3nigm4/c93d4c14800391e6d211f5482b7f99d8

LIKE REPLY SHARE

Fabián Mar 8, 2023 *Edited*

Listing 40 in Dlang: https://pastebin.com/PpgeBUqZ.

note: I forgot to remove some unused things like the OpCodeId enum.

I just did a bunch of different switch statements for all the instructions grouped by their bit length from the shortest to the longest, though I'm not sure if that could cause collisions with other instructions if I were to extend the code to things other than MOV.

LIKE REPLY SHARE

bolt Mar 7, 2023 *Edited*

My arguably rather dodgy approach using C

https://codeberg.org/bolt/perfaware-course/src/branch/main/part_01

Any comments or criticism (constructive, brutal or just downright cruel) welcome.

LIKE REPLY SHARE

bolt Mar 7, 2023

I wonder why the memory to accumulator and it's counterpart instructions have a w-bit but don't actually make use of it.

The address passed is always 16-bit afaict.

And as has been touched on before, I'm not really sure why their equivalent of a d-bit is not simply documented, even if it is the opposite to the usage for other mov instructions.

LIKE REPLY (1) SHARE

bolt Mar 7, 2023

Hmm, I imagine it's just because the examples in listing 40 explicitly use ax.

I presume one could use al or ah instead and that would give the w-bit purpose.

LIKE REPLY (1) SHARE

bolt Mar 7, 2023

Well, that seems correct for al, but using ah appears to generate a mov reg r/m instruction instead.

LIKE REPLY (1) SHARE

Casey Muratori Mar 8, 2023

Author

That's because there is no way to specify ah with the implicit accumulator mov. The w bit is either 0 or 1 for the implicit accumulator mov, so there are only two possible registers. They chose al and ax. They could have

chosen ah and ax, but then al wouldn't have been possible!

- Casey

LIKE REPLY (1) SHARE

bolt Mar 8, 2023

Cool, yep, totally makes sense.

My brain has been spinning at times over the last few days as I've been trying to mentally parse the instruction variations.

I'd be lying if I said I haven't enjoyed it though.

LIKE REPLY SHARE

rizoma0x00 Mar 7, 2023

Here's my challenge dirty one in C:

https://pastebin.com/L691qcXi

Prints stuff like mov ax, [bx + di + -37] and I didn't bother to fix it since it assembles fine, in nasm... Maybe next time!

I relied a bit too much on the debugger this time... I feel like I brute forced the thing, but this stuff is really nasty!

LIKE REPLY SHARE

Hasen Judi Hasen Judi Mar 7, 2023 *Edited*

alright, I did it like I said I would in my previous comment.

The decoder now decodes instructions into a structured object that we can then print. This should allow me to extend this and actually 'execute' the instructions against a virtual cpu.

Odin's tagged unions come in really handy here!

I tested it against listings 37-40 and it seems to work for all of them.

One thing I'm not sure about: when the address is 'direct' I assume it can't be negative, so I structured the code that way, but I'm not sure if that's to spec (hard to imagine they would allow a negative memory address).

https://gist.github.com/hasenj/57b866a742e7b7506d49f785fb4a8254

LIKE REPLY SHARE

Jonathan Pentecost Mar 7, 2023

This was very fun. Here is mine in Go: https://github.com/vishen/perfaware

LIKE REPLY SHARE

Cypress Mar 7, 2023

Here's my attempt at listing 0039 in rust https://github.com/cypressf/computer_enhance/blob/main/perfaware/part1/src/main.rs

Looking at the work others have done is making me consider using lookup tables instead of match expressions, although there's certainly something nice about pattern matching in rust, this just might not be the right time for it. Usually I like pattern matching because it guarantees you catch all the cases, but when comparing small, 3-bit or 1-bit flags, there are not many cases from the u8 type I'm matching on.

LIKE REPLY SHARE

TheGag96 Mar 7, 2023

Casey, I think your microphone is slightly panned to the left and has been since the series started. Is this intentional?

LIKE REPLY (1) SHARE

Casey Muratori Mar 7, 2023

Author

Yes, since I am on the left side of the screen.

- Casey

LIKE (2) REPLY SHARE

Tristan Burgess Mar 6, 2023 *Edited*

I haven't put much thought into structuring the code, but got all tests captured and passing! Wahoo!

https://github.com/tristanburgess/performance_aware_programming/tree/v0.1/hw/sim8086

LIKE REPLY SHARE

Finalspace Mar 6, 2023 *Edited*

Finally i finished listing 39 and 40 and it produces the same asm output ;-)

Most of it is table driven now, so adding more stuff gets easier.

https://github.com/f1nalspace/8086sim/blob/main/CPU8086/8086.cs

Note that there is a bug in the 8086 documentation on page 174 for instruction $A3.

The source register of AL is incorrect, it is AX instead because its a wide instruction!

LIKE REPLY (1) SHARE

Cory Duplantis Mar 6, 2023

Oh neat! Took me a second to realize what page 174 was, but I see this on page "4-32" in the manual itself. Is this what you were referencing? The offending line (I think):

A3 | 1010 0011 | ADDR-LO | ADDR-HI | MOV MEM16, AL

should be

A3 | 1010 0011 | ADDR-LO | ADDR-HI | MOV MEM16, AX

LIKE REPLY (1) SHARE

Finalspace Mar 6, 2023

Correct, thats the description which is wrong

LIKE REPLY SHARE

Gordon Mar 6, 2023 *Edited*

Got listing 39 working in Zig. Haven't looked at 40 yet (but confirmed it doesn't work with my code so far).

Edit: 40 is working now too

https://gist.github.com/g-cassie/9702ec3e88c437677f095de77a7890ad

LIKE REPLY (1) SHARE

Mikhail Turkeev Mar 7, 2023

I am also using zig. High five:)

https://gist.github.com/mixtur/c215058a91f43da672694f9aac504762

Just statred learning it though.

LIKE (1) REPLY (1) SHARE

Gordon Mar 7, 2023

Nice - I am also new to it. I was curious about using packed structs so it was cool to see how you did it.

LIKE REPLY SHARE

The Sandvich Maker Makin' Sandviches Mar 5, 2023

Challenge taken!

https://github.com/TheSandvichMaker/ComputerEnhance-Homework/tree/main/2%20-%20Decoding%20Multiple%20Instructions%20and%20Suffixes

LIKE REPLY SHARE

Antonio Martinez Arias Pounded Earth and Pundits Mar 5, 2023 *Edited*

Here's my code for the first part of this exercise (solves listing_39.asm): https://pastebin.com/WsbUSwgL

I felt that the instructions want to be decoded in a particular sequence:

1. What is the instruction type (e.g. "oh, this is a MOV")

2. Where do the operands go and what kind are they (immediate, displacement, register for memory or not) - this is very easy for some instructions :)

3. If there is a mode, do whatever needs to happen based on the table

4. If there is an immediate value, get it

5. Output the instruction using the information above

Of course, my code doesn't do this exactly because I only figured out some of the steps in retrospect, but I will be updating it for the challenge :)

LIKE REPLY (1) SHARE

Antonio Martinez Arias Pounded Earth and Pundits Mar 6, 2023

Here's the new code for the challenge listing: https://pastebin.com/5bdBf6um

I'm sure there's a bug somewhere because I only tested it out with the listings and a

few other test cases for the encodings I wasn't sure about.

LIKE REPLY SHARE

Ameen Mar 5, 2023

Here is my solution. I implemented all MOV opcodes including mov to/from segment registers:

https://github.com/aameen951/computer-enhance-8086-decoder/blob/master/main.cpp

LIKE REPLY SHARE

Alexander Keller Mar 5, 2023

Loving this series!

Something I've seen but is tricky to google for - is there a difference between eg. "mov byte [address], imm" or "mov [address], byte imm"?

LIKE REPLY (1) SHARE

Paolo Mar 6, 2023 *Edited*

Is exactly the same meaning, but different assemblers may prefer (or require) one way or the other

LIKE REPLY SHARE

TechnicBeam TechnicBeam Mar 5, 2023

Two typos in the same paragraph: If you imagine you just had an array of bytes called Memory, and it was 65536 bytes **long**/(~lone) — 64k worth of bytes — that is exactly what we the CPU is thinking of when you do a memory access on 8086. The [75] is **just**(~jus) like looking up index 75 in an array of bytes, and getting out the value.

LIKE REPLY SHARE

mnemonix Mar 5, 2023 *Edited*

Thanks, Casey! Here is my homework: https://pastebin.com/UAnqjZYf

EDIT: Forgot sign extension for displacement. New version: https://pastebin.com/EVSBBiNy

LIKE REPLY SHARE

Mikael Nilsson Mikael's Substack Mar 5, 2023

That was a fun exercise. My solution for listing 40: https://pastebin.com/kCn1cykm (C)

LIKE REPLY (2) SHARE

Robert Hildebrandt Robert's Substack Mar 6, 2023

Wow C supports binary literals? I wish I had known that earlier^^

LIKE REPLY (2) SHARE

Mikael Nilsson Mikael's Substack Mar 6, 2023

It is a gcc extension not standard c

LIKE (1) REPLY (1) SHARE

Robert Hildebrandt Robert's Substack Mar 7, 2023

I've just took a look at the list of GCC C extensions. There's some crazy stuff in there!

My new two favorites (besides binary literals):

- Statement blocks as expressions! I need to check which macros I can simplify with them.

- Vectorized types (apparently with the standard C operators supported). Will be interesting to analyze the generated binaries when we arrive at SIMD in this

course.

LIKE (1) REPLY SHARE

Ameen Mar 6, 2023

Not in C unfortunately. C++14 finally added it.

LIKE (1) REPLY SHARE

Ameen Mar 5, 2023

You code looks nice and clean.

Two notes:

1. You're not checking `instruction_stream` for overflow before incrementing.

2. You can just write `-DISP_16` instead of `(~DISP_16 + 1)`.

LIKE REPLY (1) SHARE

Craft Links Mar 6, 2023

Telling someone's code is "clean" almost reads like an insult after watching Casey's previous video ;-)

LIKE (1) REPLY (1) SHARE

Seth Mar 6, 2023

I know right? Thems fighting words! :)

LIKE REPLY SHARE

A21 Mar 5, 2023 *Edited*

Thanks, Casey!

Updated my homework (Listing 39): https://github.com/sigmawq/8086/blob/master/src/main.odin

LIKE REPLY SHARE

Lothar Narins Mar 5, 2023 *Edited*

This was fun! My solution in Odin, including the challenge listing: https://github.com/MooOfDoom/sim8086/tree/115d69e691bfe92b9ac8b52cee9dc5f363b6d921

LIKE REPLY SHARE