Module #3: X86 Assembly Language

Möbius Strip Reverse Engineering

April 19, 2018

X86 Assembly Language

Overview

$$\underbrace{\texttt{lock}}_{\text{Prefix}}\quad\underbrace{\texttt{add}}_{\text{Mnemonic}}\quad\underbrace{\texttt{word ptr [eax]}}_{\text{Operand \#1}}\quad,\quad\underbrace{\texttt{bx}}_{\text{Operand \#2}}$$

An X86 instruction consists of:

1. Zero or more **prefixes** (dictating sizes, repetition (REP), atomicity behavior (LOCK), etc.).
2. A **mnemonic**, which gives a human-friendly name to the operation being performed by the processor.
3. Zero to three **operands**, the "arguments".

X86 Assembly Language

Operands

X86 instruction operands fall into five broad categories:

1. Registers
2. Constant values
3. Control flow (`jmp`/`call`) destinations
4. Absolute addresses (segment:offset)
5. Memory expressions

X86 Assembly Language Operands

Registers

Category	Size	Values
Gb	8	al, cl, dl, bl, ah, ch, dh, bh
Gw	16	ax, cx, dx, bx, sp, bp, si, di
Gd	32	eax, ecx, edx, ebx, esp, ebp, esi, edi
SegReg	16	es, cs, ss, ds, fs, gs
ControlReg	32	cr0, cr1, cr2, cr3, cr4, cr5, cr6, cr7
DebugReg	32	dr0, dr1, dr2, dr3, dr4, dr5, dr6, dr7
FPUReg	80	st0, st1, st2, st3, st4, st5, st6, st7
MMXReg	64	mm0, mm1, mm2, mm3, mm4, mm5, mm6, mm7
XMMReg	128	xmm0, xmm1, xmm2, xmm3, xmm4, xmm5, xmm6, xmm7

Each register is associated with a number 0-7, in the order shown.

X86 Assembly Language Operands

Immediates

Category	Size	Example Values
`Ib`	8	`12h`, `0FFh`
`Iw`	16	`1234h`, `0FFFFh`
`Id`	32	`12345678h`, `0FFFFFFFFh`

Immediates are constant values of a specified size.

X86 Assembly Language Operands

Control-Flow Targets

```
.text:00401024 jnz loc_401050    JccTarget(0x401250,0x401026)
.text:00401024 call loc_401050   JccTarget(0x401250,0x401026)
.text:00401024 jmp loc_401050    JccTarget(0x401250,0x401026)
```

`JccTarget` records the destination, and:

- For conditional jumps, the not-taken (i.e. next) address.
- For `call`, the return (i.e. next) address.
- For `jmp`, a bogus value (the next address).

X86 Assembly Language Operands

Segment:Offset Memory Locations

```
call 1234h:12345678h   AP32(0x1234,0x12345678)
jmp 1234h:5678h        AP16(0x1234,0x5678)
```

- Some instructions allow memory locations specified by segment and offset.

X86 Assembly Language Operands

Memory Expressions, 32-Bit

$$\underbrace{\texttt{byte ptr}}_{\text{Operand Size}}\ \underbrace{\texttt{ds:}}_{\text{Segment}}\ \texttt{[}\ \underbrace{\texttt{eax}}_{\text{Base Register}}\ \texttt{+}\ \underbrace{\texttt{ebx}}_{\text{Index Register}}\ \texttt{*}\ \underbrace{\texttt{4}}_{\text{Scale Factor}}\ \texttt{+}\ \underbrace{\texttt{18h}}_{\text{Displacement}}\ \texttt{]}$$

```
Mem32(DS,Mb,Eax,Ebx,4,0x18)
```

32-bit memory expressions contain:

1. A 32-bit general base register
2. A 32-bit general index register and a 2-bit scale factor
3. A 32-bit immediate displacement

- All parts are optional, but at least one must be present.

X86 Assembly Language Operands

Memory Expressions, 16-Bit

$$\underbrace{\texttt{byte ptr}}_{\text{Operand Size}}\ \underbrace{\texttt{ds:}}_{\text{Segment}}\ [\ \underbrace{\texttt{bx}}_{\text{Base Register}} + \underbrace{\texttt{si}}_{\text{Index Register}} + \underbrace{\texttt{1234h}}_{\text{Displacement}}\ ]$$

```
Mem16(DS,Mb,Bx,Si,0x1234)
```

16-bit memory expressions contain:

1. A 16-bit general base register
2. A 16-bit general index register
3. A 16-bit immediate displacement

- All parts are optional, but at least a base register or displacement must be present.
- Only certain base/index register combinations are legal.

X86 Assembly Language

Prefixes

Prefix	Non-Prefixed Instruction	Prefixed Instruction
	Prefix Group #1	
LOCK	`add ecx, [eax]`	`lock add ecx, [eax]`
REP	`movsb`	`rep movsb`
REPNZ	`scasb`	`repnz scasb`
	Prefix Group #2	
SEGMENT	`mov edx, ds:[0]`	`mov edx, fs:[0]`
	Prefix Group #3	
OPSIZE	`xor eax, eax`	`xor ax, ax`
	Prefix Group #4	
ADDRSIZE	`add ecx, [eax]`	`add ecx, [bx+si]`

- Adding prefixes can modify an instruction.

X86 Assembly Language

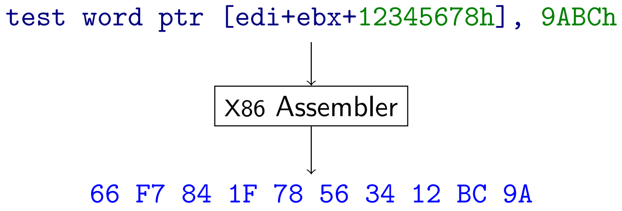
Group #1 Prefixes

Prefix	Non-Prefixed Instruction	Prefixed Instruction
Prefix Group #1		
LOCK	`add ecx, [eax]`	`lock add ecx, [eax]`
REP	`movsb`	`rep movsb`
REPNZ	`scasb`	`repnz scasb`

- LOCK is used for inter-processor synchronization. It is valid on few mnemonics, and only meaningful for memory operands.
  - `adc`, `add`, `and`, `btc`, `btr`, `bts`, `cmpxchg`, `cmpxchg8b`, `dec`, `inc`, `neg`, `not`, `or`, `sbb`, `xadd`, `xchg`, `xor`
- REP (also REPZ) modifies a string instruction to repeat (while the `zf` flag is set).
- REPNZ modifies a string instruction to repeat while the `zf` flag is not set.

X86 Machine Code

Overview



```
test word ptr [edi+ebx+12345678h], 9ABCh
```

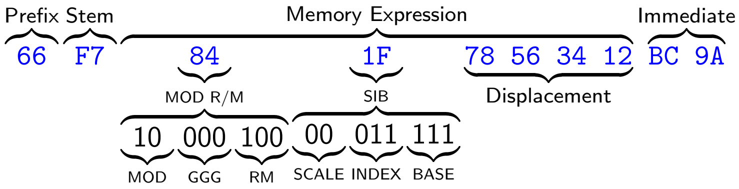
```
66 F7 84 1F 78 56 34 12 BC 9A
```

Textual X86 instructions must be encoded into **machine code** (bytes) before the processor can interpret them.

X86 Machine Code

Overview

```
66 F7 84 1F 78 56 34 12 BC 9A
test word ptr [edi+ebx+12345678h], 9ABCh
```



X86 machine code instructions consist of four parts:

1. Optional **prefixes**
2. An **instruction stem**
3. An optional **memory expression**, consisting of:
    3.1 A **MOD R/M** byte
    3.2 An optional **SIB** byte
    3.3 An optional **displacement** (1, 2, or 4 bytes)
4. Zero, one, or two **immediate** (constant) values

X86 Machine Code

Prefixes

Group #1		Group #2		Group #3		Group #4	
lock	F0	cs	2E	OPSIZE	66	ADDRSIZE	67
rep	F3	ss	36				
repz	F2	ds	3E				
repnz	F2	es	26				
		fs	64				
		gs	65				

- Each prefix is associated with some particular byte.

X86 Machine Code

Stems

- The **stem** follows any prefix bytes, and dictates which instruction (i.e., mnemonic) should execute.
- X86 has more than 256 instructions, and uses variable-length instruction encodings, so more than one byte may be required.
- X86 uses **escape bytes** to encode some instructions.
  - `0F xx` designates 256 two-byte instruction stems.
  - `0F 38 xx` designates 256 three-byte instruction stems.
  - `0F 3A xx` designates 256 three-byte instruction stems.
- Sometimes, the mnemonic selected by a stem depends on:
  1. Prefixes, and/or
  2. The contents of the MOD R/M (described next).

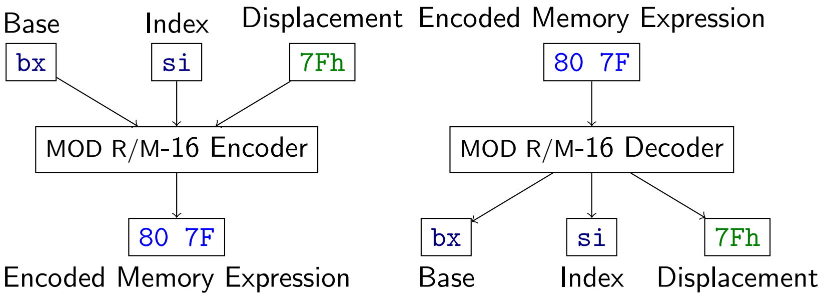
X86 Machine Code

MOD R/M Encodings

- X86 allows for complex memory access modes.
  - 32-bit examples: `[eax]`, `[eax*8]`, `[ebx+eax]`, `[ebx+eax*8]`, `[eax+12345678h]`, `[eax*8+12345678h]`, `[ebx+eax+12345678h]`, `[ebx+eax*8+12345678h]`, `gs:[eax]`, `[12345678h]`
  - 16-bit examples: `[bx]`, `[bx+si]`, `[1234h]`
- In 32-bit mode, 32-bit expressions are used by default.
  - The ADDRSIZE prefix causes a 16-bit expression to be used.
- Memory expressions are encoded using the MOD R/M scheme.
  - 16-bit memory expressions use MOD R/M-16.
  - 32-bit memory expressions use MOD R/M-32.

X86 MOD R/M Encodings

MOD R/M-16



We begin with MOD R/M-16 for simplicity.

X86 MOD R/M Encodings

Subdivision, Specification of Registers or Memory

$$\overbrace{\underbrace{11}_{\text{MOD}}\;\underbrace{000}_{\text{GGG}}\;\underbrace{\mathbf{001}}_{\text{RM}}}^{\overbrace{\text{MOD R/M}}^{\texttt{C1}}}$$

- MOD R/M bytes are divided into three fields.
  - We ignore GGG for now.
- They specify either a register or a memory location.
  - If MOD is 11 (i.e., 3), RM is used as a register number.
    - Recall that each register is associated with a number $0 - 7$.
    - The instruction stem dictates the register family.
  - If MOD is not 11, a memory expression is specified.
    - We assume in the following material that MOD is not 11.

X86 MOD R/M-16 Encodings

MOD R/M Memory Encodings

- MOD R/M-16.RM selects the base and index registers, as in:

000	001	010	011
bx si	bx di	bp si	bp di

100	101	110	111
si	di	bp	bx

- These are the only valid combinations of base and index registers allowed by MOD R/M-16.
  - I.e., `[ax+sp]` is an invalid 16-bit memory expression.

X86 MOD R/M-16 Encodings

MOD and Displacements

- A displacement may follow the MOD R/M-16 byte, depending upon the value of MOD.

MOD	Size of displacement
00	None
01	8-bit displacement, sign-extended to 16-bits
10	16-bit

MOD	MOD R/M-16	Memory Expression
00	$\underbrace{\texttt{04}}_{\text{MOD R/M}}$ $\overbrace{\underbrace{00}_{\text{MOD}}\ \underbrace{000}_{\text{GGG}}\ \underbrace{100}_{\text{RM}}}$	`[si]`
01	$\underbrace{\texttt{45}}_{\text{MOD R/M}}\ \underbrace{\texttt{80}}_{\text{Displacement}}$	`[di+0FF80h]`
10	$\underbrace{\texttt{81}}_{\text{MOD R/M}}\ \underbrace{\texttt{34 12}}_{\text{Displacement}}$	`[bx+di+1234h]`

X86 MOD R/M-16 Encodings

8-Bit Displacements

An 8-bit encoding for displacement `d` may be used when:

- Take the low 8 bits of the displacement: `low = d&0xFF`.
- Sign extend `low` to 16-bits: `ext = sign_extend_8_16(low)`.
- If `d == ext`, the result is the same as the original displacement.

d	low	ext	d==ext	d	low	ext	d==ext
00h	00h	00h	✓	0FF7Eh	7Eh	7Eh	✗
01h	01h	01h	✓	0FF7Fh	7Fh	7Fh	✗
	...			0FF80h	80h	0FF80h	✓
7Eh	7Eh	7Eh	✓	0FF81h	81h	0FF81h	✓
7Fh	7Fh	7Fh	✓		...		
80h	80h	0FF80h	✗	0FFFEh	0FEh	0FFFEh	✓
81h	81h	0FF81h	✗	0FFFFh	0FFh	0FFFFh	✓

- I.e., 8-bit encoding can be used when `0FF80h <= d <= 7Fh`.

X86 MOD R/M-16 Encodings

Special Case

MOD R/M-16	Memory Expression
$\underbrace{04}_{\text{MOD R/M}} \quad \underbrace{34\ \ 12}_{\text{Displacement}}$ $\overbrace{\underbrace{\mathbf{00}}_{\text{MOD}}\ \underbrace{000}_{\text{GGG}}\ \underbrace{\mathbf{110}}_{\text{RM}}}$	`[1234h]`

- If MOD is **00** and RM is **110** (otherwise representing `[bp]`), the memory expression is just the 16-bit displacement following the MOD R/M-16 byte.
  - Consequently, it is impossible to use `bp` as a base register without specifying a displacement.

X86 MOD R/M-16 Encodings

r8			AL	CL	DL	BL	AH	CH	DH	BH
r16			AX	CX	DX	BX	SP	BP	SI	DI
r32			EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
mm			MM0	MM1	MM2	MM3	MM4	MM5	MM6	MM7
xmm			XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7
ymm			YMM0	YMM1	YMM2	YMM3	YMM4	YMM5	YMM6	YMM7
sreg			ES	CS	SS	DS	FS	GS		
ccc			CR0	CR1	CR2	CR3	CR4	CR5	CR6	CR7
ddd			DR0	DR1	DR2	DR3	DR4	DR5	DR6	DR7
digit			0	1	2	3	4	5	6	7
ggg			000	001	010	011	100	101	110	111
effective address	mod	R/M	value of mod R/M byte (hex)							
[BX+SI]	00	000	00	08	10	18	20	28	30	38
[BX+DI]		001	01	09	11	19	21	29	31	39
[BP+SI]		010	02	0A	12	1A	22	2A	32	3A
[BP+DI]		011	03	0B	13	1B	23	2B	33	3B
[SI]		100	04	0C	14	1C	24	2C	34	3C
[DI]		101	05	0D	15	1D	25	2D	35	3D
[sword]		110	06	0E	16	1E	26	2E	36	3E
[BX]		111	07	0F	17	1F	27	2F	37	3F
[BX+SI+sbyte]	01	000	40	48	50	58	60	68	70	78
[BX+DI+sbyte]		001	41	49	51	59	61	69	71	79
[BP+SI+sbyte]		010	42	4A	52	5A	62	6A	72	7A
[BP+DI+sbyte]		011	43	4B	53	5B	63	6B	73	7B
[SI+sbyte]		100	44	4C	54	5C	64	6C	74	7C
[DI+sbyte]		101	45	4D	55	5D	65	6D	75	7D
[BP+sbyte]		110	46	4E	56	5E	66	6E	76	7E
[BX+sbyte]		111	47	4F	57	5F	67	6F	77	7F
[BX+SI+sword]	10	000	80	88	90	98	A0	A8	B0	B8
[BX+DI+sword]		001	81	89	91	99	A1	A9	B1	B9
[BP+SI+sword]		010	82	8A	92	9A	A2	AA	B2	BA
[BP+DI+sword]		011	83	8B	93	9B	A3	AB	B3	BB
[SI+sword]		100	84	8C	94	9C	A4	AC	B4	BC
[DI+sword]		101	85	8D	95	9D	A5	AD	B5	BD
[BP+sword]		110	86	8E	96	9E	A6	AE	B6	BE
[BX+sword]		111	87	8F	97	9F	A7	AF	B7	BF
AL/AX/EAX/MM0/XMM0/YMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
CL/CX/ECX/MM1/XMM1/YMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
DL/DX/EDX/MM2/XMM2/YMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
BL/BX/EBX/MM3/XMM3/YMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
AH/SP/ESP/MM4/XMM4/YMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
CH/BP/EBP/MM5/XMM5/YMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
DH/SI/ESI/MM6/XMM6/YMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
BH/DI/EDI/MM7/XMM7/YMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

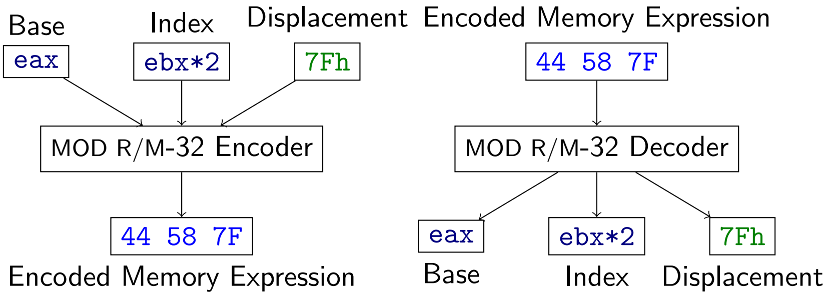
X86 MOD R/M-16 Encodings

Segment Rules

- If a segment prefix is specified before the instruction stem, the memory expression addresses that segment.
- Otherwise, if the base register is `bp`, use `ss`.
- Otherwise, use `ds`.

X86 MOD R/M-32

Overview



- MOD R/M-32 is a binary coding scheme for 32-bit memory expressions, similar to MOD R/M-16.
- Index registers require an additional SCALE-INDEX-BASE (SIB) byte after the MOD R/M and before any displacement.
- More flexible than MOD R/M-16, but more complex.
  - I.e., more special cases.

X86 MOD R/M Encodings

Subdivision, Specification of Registers or Memory

$$\overbrace{\underbrace{11}_{\text{MOD}}\ \underbrace{000}_{\text{GGG}}\ \underbrace{\mathbf{001}}_{\text{RM}}}^{\overbrace{\text{MOD R/M}}^{\texttt{C1}}}$$

As before:

- MOD R/M bytes are divided into three fields.
- They can specify either a register or a memory location.
- If MOD is 11 (i.e., 3), RM is used as a register number.
  - The register family is specified by the instruction stem.
- If MOD is not 11, a memory expression is specified.
  - We assume in the following material that MOD is not 11.

X86 MOD R/M-32 Encodings

MOD R/M Memory Encodings

As before:

- MOD R/M-32.RM selects the base register, as in:

000	001	010	011	100	101	110	111
`eax`	`ecx`	`edx`	`ebx`	N/A	`ebp`	`esi`	`edi`

- `esp` cannot be used as a MOD R/M base register.
  - It triggers a special case, SIB, described later.

X86 MOD R/M-32 Encodings

MOD and Displacements

- As before, a displacement may follow the MOD R/M-32 byte:

MOD	Size of displacement
00	None
01	8-bit displacement, sign-extended to 32-bits
10	32-bit

MOD	MOD R/M-32	Memory Expression
00	$\underbrace{00}_{\text{MOD R/M}}$ $\overbrace{\underbrace{00}_{\text{MOD}}\ \underbrace{000}_{\text{GGG}}\ \underbrace{001}_{\text{RM}}}$	`[ecx]`
01	$\underbrace{42}_{\text{MOD R/M}}\ \underbrace{80}_{\text{Displacement}}$	`[edx+0FFFFFF80h]`
10	$\underbrace{83}_{\text{MOD R/M}}\ \underbrace{78\ 56\ 34\ 12}_{\text{Displacement}}$	`[ebx+12345678h]`

X86 MOD R/M-32 Encodings

Special Case #1: Displacement Only

MOD R/M-32	Memory Expression
$\underbrace{05}_{\text{MOD R/M}} \quad \underbrace{78\ 56\ 34\ 12}_{\text{Displacement}}$ $\overbrace{\underbrace{\mathbf{00}}_{\text{MOD}}\ \underbrace{000}_{\text{GGG}}\ \underbrace{\mathbf{101}}_{\text{RM}}}$	`[12345678h]`

- Similarly to before, if MOD is **00** and RM is **101** (otherwise representing `[ebp]`), the memory expression is just the 32-bit displacement following the MOD R/M-32 byte.
  - Consequently, it is impossible to use `ebp` as a base register without specifying a displacement.

X86 MOD R/M-32 Encodings

Special Case #2: SIB

- We discussed expressions with base registers, but no index register. The latter require SCALE-INDEX-BASE (SIB) bytes.

MOD R/M-32 with SIB	Memory Expression
$\overbrace{\underbrace{01}_{\text{MOD}}\ \underbrace{000}_{\text{GGG}}\ \underbrace{\mathbf{100}}_{\text{RM}}}^{\underbrace{44}_{\text{MOD R/M}}}\ \overbrace{\underbrace{01}_{\text{SCALE}}\ \underbrace{011}_{\text{INDEX}}\ \underbrace{000}_{\text{BASE}}}^{\underbrace{58}_{\text{SIB}}}\ \underbrace{78}_{\text{Displacement}}$	`[eax+ebx*2+78h]`

- If RM is **100**, then a SIB byte follows the MOD R/M.
  - If not for the special case, this would specify `[esp+disp]`.
- If a displacement is required, it is encoded after the SIB.

X86 MOD R/M-32/SIB Encodings

Subdivision and Ordinary Cases

$$\overbrace{\underbrace{00}_{\text{SCALE}}\ \underbrace{011}_{\text{INDEX}}\ \underbrace{111}_{\text{BASE}}}^{\underbrace{\texttt{1F}}_{\text{SIB}}}$$

- Like MOD R/M, SIB bytes are divided into three fields.
- SCALE specifies the **scale factor**, i.e., the `4` in `ebx*4`.
  - 00: `1`, 01: `2`, 10: `4`, 11: `8`
- INDEX specifies the **index register** number.
- BASE specifies the **base register** number.
- A displacement may follow, depending upon MOD R/M.MOD.

X86 MOD R/M-32/SIB Encodings

Special Case #1: No Index Register

MOD R/M-32 with SIB	Memory Expression
$\overbrace{\underbrace{01}_{\text{MOD}}\ \underbrace{000}_{\text{GGG}}\ \underbrace{100}_{\text{RM}}}^{\underbrace{44}_{\text{MOD R/M}}}\ \overbrace{\underbrace{11}_{\text{SCALE}}\ \underbrace{\mathbf{100}}_{\text{INDEX}}\ \underbrace{000}_{\text{BASE}}}^{\underbrace{\texttt{E0}}_{\text{SIB}}}\ \underbrace{78}_{\text{Displacement}}$	`[eax+78h]`

- If a INDEX of **100** (`esp`) is specified, there is no index register.
    - I.e., `esp` cannot be used as an index register.

X86 MOD R/M-32/SIB Encodings

Special Case #2: Base Register is `ebp`

MOD R/M-32 with SIB	Memory Expression
$\overbrace{\underbrace{01}_{\text{MOD}}\;\underbrace{000}_{\text{GGG}}\;\underbrace{100}_{\text{RM}}}^{\underbrace{\texttt{44}}_{\text{MOD R/M}}}\;\overbrace{\underbrace{11}_{\text{SCALE}}\;\underbrace{000}_{\text{INDEX}}\;\underbrace{\mathbf{101}}_{\text{BASE}}}^{\underbrace{\texttt{E7}}_{\text{SIB}}}\;\underbrace{\texttt{78}}_{\text{Displacement}}$	`[ebp+eax*8+78h]`

- If a BASE of **101** (`ebp`) is specified, the memory expression and its displacement depends on MOD.

MOD	Resulting memory expression
00	`[index*scale+dword]`
01	`[ebp+index*scale+byte]`
10	`[ebp+index*scale+dword]`

X86 MOD R/M-32 Encodings

r8			AL	CL	DL	BL	AH	CH	DH	BH
r16			AX	CX	DX	BX	SP	BP	SI	DI
r32			EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
mm			MM0	MM1	MM2	MM3	MM4	MM5	MM6	MM7
xmm			XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7
ymm			YMM0	YMM1	YMM2	YMM3	YMM4	YMM5	YMM6	YMM7
sreg			ES	CS	SS	DS	FS	GS		
eee			CR0	CR1	CR2	CR3	CR4	CR5	CR6	CR7
eee			DR0	DR1	DR2	DR3	DR4	DR5	DR6	DR7
digit			0	1	2	3	4	5	6	7
reg=			000	001	010	011	100	101	110	111
effective address	mod	R/M	value of mod R/M byte (hex)							
[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
[sib]		100	04	0C	14	1C	24	2C	34	3C
[sdword]		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+sbyte]	01	000	40	48	50	58	60	68	70	78
[ECX+sbyte]		001	41	49	51	59	61	69	71	79
[EDX+sbyte]		010	42	4A	52	5A	62	6A	72	7A
[EBX+sbyte]		011	43	4B	53	5B	63	6B	73	7B
[sib+sbyte]		100	44	4C	54	5C	64	6C	74	7C
[EBP+sbyte]		101	45	4D	55	5D	65	6D	75	7D
[ESI+sbyte]		110	46	4E	56	5E	66	6E	76	7E
[EDI+sbyte]		111	47	4F	57	5F	67	6F	77	7F
[EAX+sdword]	10	000	80	88	90	98	A0	A8	B0	B8
[ECX+sdword]		001	81	89	91	99	A1	A9	B1	B9
[EDX+sdword]		010	82	8A	92	9A	A2	AA	B2	BA
[EBX+sdword]		011	83	8B	93	9B	A3	AB	B3	BB
[sib+sdword]		100	84	8C	94	9C	A4	AC	B4	BC
[EBP+sdword]		101	85	8D	95	9D	A5	AD	B5	BD
[ESI+sdword]		110	86	8E	96	9E	A6	AE	B6	BE
[EDI+sdword]		111	87	8F	97	9F	A7	AF	B7	BF
AL/AX/EAX/MM0/XMM0/YMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
CL/CX/ECX/MM1/XMM1/YMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
DL/DX/EDX/MM2/XMM2/YMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
BL/BX/EBX/MM3/XMM3/YMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
AH/SP/ESP/MM4/XMM4/YMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
CH/BP/EBP/MM5/XMM5/YMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
DH/SI/ESI/MM6/XMM6/YMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
BH/DI/EDI/MM7/XMM7/YMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

X86 MOD R/M-32/SIB Encodings

r32			EAX	ECX	EDX	EBX	ESP	[*]	ESI	EDI
digit			0	1	2	3	4	5	6	7
base=			000	001	010	011	100	101	110	111
scaled index	SS	Index	value of SIB byte (hex)							
[EAX*1]	00	000	00	01	02	03	04	05	06	07
[ECX*1]		001	08	09	0A	0B	0C	0D	0E	0F
[EDX*1]		010	10	11	12	13	14	15	16	17
[EBX*1]		011	18	19	1A	1B	1C	1D	1E	1F
none		100	20	21	22	23	24	25	26	27
[EBP*1]		101	28	29	2A	2B	2C	2D	2E	2F
[ESI*1]		110	30	31	32	33	34	35	36	37
[EDI*1]		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

X86 MOD R/M-32 Encodings

Segment Rules

Similarly to MOD R/M-16:

- If a segment prefix is specified before the instruction stem, the memory expression addresses that segment.
- Otherwise, if the base register is `ebp` or `esp`, use `ss`.
- Otherwise, use `ds`.

The X86 Instruction Set
Overview

- Not every combination of prefixes, mnemonics, and operands is valid.

Purported Instruction	Reason for Invalidity
`add eax`	Too few operands
`add eax, al`	Mismatched register types
`add eax, ebx, ecx`	Too many operands
`add 1, al`	Invalid destination
`add [eax], [ebx]`	Illegal operand combination

- Each legal instruction conforms to some **encoding**.

The X86 Instruction Set

Opcode Maps

	0	1	2	3	4	5	6	7
0	ADD						PUSH	POP
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	ES	ES
1	ADC						PUSH	POP
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	SS	SS
2	AND						SEG=ES	DAA
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	(Prefix)	
3	XOR						SEG=SS	AAA
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	(Prefix)	
4	INC general register							
	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI
5	PUSH general register							
	rAX	rCX	rDX	rBX	rSP	rBP	rSI	rDI
6	PUSHA/ PUSHAD	POPA/ POPAD	BOUND Gv, Ma	ARPL Ew, Gw	SEG=FS (Prefix)	SEG=GS (Prefix)	Operand Size (Prefix)	Address Size (Prefix)
7	Jcc, Jb - Short-displacement jump on condition							
	O	NO	B/NAE/C	NB/AE/NC	Z/E	NZ/NE	BE/NA	NBE/A
8	Immediate Grp 1				TEST		XCHG	
	Eb, Ib	Ev, Iz	Eb, Ib	Ev, Ib	Eb, Gb	Ev, Gv	Eb, Gb	Ev, Gv

- The Intel opcode maps describe the X86 instruction set, i.e., the legal **encodings** for each instruction stem.
  - There may be multiple for a given stem; see the **group** in 80.
- Legal operands are specified by **abstract operand types**.
  - E.g. `OEb`, `OGb`, `OEv`, `OGv`, `OAL`, `OIb`, `OIz`, `OrAX`, `OES`, `OSS`, `OeAX`, . . .

X86 Instruction Encodings

Encoding Table

Table: All Encodings for `add`

Encoding Method	Stem Bytes	Abstract Operand Types		Example X86 Instruction
Ordinary	00	OEb	OGb	`add bl, al`
Ordinary	01	OEv	OGv	`add [eax], eax`
Ordinary	02	OGb	OEb	`add bl, al`
Ordinary	03	OGv	OEv	`add eax, [eax]`
Ordinary	04	OAL	OIb	`add al, 12h`
Ordinary	05	OrAX	OIz	`add ax, 1234h`
MOD R/M Group #0	80	OEb	OIb	`add al, 1`
MOD R/M Group #0	81	OEv	OIz	`add [ecx], 1`
MOD R/M Group #0	83	OEv	OIbv	`add ebx, -1`

- For each mnemonic, we keep a table of its valid encodings.

X86 Instruction Encodings

Encoding Method

Encoding Method	Description
Ordinary	Nothing special needs to be done.
Size Prefix	Instruction requires an OPSIZE prefix.
MOD R/M Group #n	Set MOD R/M.GGG to n.

- The **encoding method** describes any extra information required to encode a particular instruction.

X86 Instruction Encoding Methods

Size Prefix Required

Mnemonic	Encoding Method	Stem Bytes	Abstract Operand Types
`pushaw`	Size Prefix	60	

`66 60 pushaw` $\underbrace{\texttt{66}}_{\text{OPSIZE Prefix}} \underbrace{\texttt{60}}_{\text{Stem}}$

- The Size Prefix encoding dictates that an OPSIZE prefix must be present.

X86 Instruction Encoding Methods

MOD R/M Groups

Mnemonic	Encoding Method	Stem Bytes	Abstract Operand Types
`add`	MOD R/M Group **#0**	80	`OEb` `OIb`
`or`	MOD R/M Group **#1**	80	`OEb` `OIb`

`80 C0 7F add al, 7Fh`   `80 C8 7F or al, 7Fh`

$$\underbrace{\texttt{80}}_{\text{Stem}} \quad \underbrace{\texttt{C0}}_{\overbrace{\underbrace{11}_{\text{MOD}}\ \underbrace{\mathbf{000}}_{\text{GGG}}\ \underbrace{000}_{\text{RM}}}^{\text{MOD R/M}}} \quad \underbrace{\texttt{7F}}_{\text{Immediate } \texttt{OIb}} \qquad \underbrace{\texttt{80}}_{\text{Stem}} \quad \underbrace{\texttt{C8}}_{\overbrace{\underbrace{11}_{\text{MOD}}\ \underbrace{\mathbf{001}}_{\text{GGG}}\ \underbrace{000}_{\text{RM}}}^{\text{MOD R/M}}} \quad \underbrace{\texttt{7F}}_{\text{Immediate } \texttt{OIb}}$$

- For instruction groups, the MOD R/M.GGG is set to the **value** indicated in the encoding.
  - `al` is encoded as MOD = 11, RM = 000.
  - `7Fh` is encoded as an immediate.
- Groups can be more complex; we discuss them again later.

X86 Instruction Encodings
Abstract Operand Types

Encodings for `add`, Truncated

Encoding Method	Stem Bytes	Abstract Operand Types	Example X86 Instruction
Ordinary	00	`OEb` `OGb`	`add [eax], al`
Ordinary	02	`OGb` `OEb`	`add bl, al`
Ordinary	04	`OAL` `OIb`	`add al, 12h`
MOD R/M Group #0	80	`OEb` `OIb`	`add al, 1`

- The **abstract operand types** (AOTs), to be detailed subsequently, describe which operands are valid for a particular encoding.

`OGb`	8-bit register	`OIb`	8-bit immediate value
`OAL`	The register `al`	`OEb`	8-bit register or memory location

Meaning of Abstract Operand Types Used Above

X86 Abstract Operand Types

Overview

- The Intel opcode maps use over 100 abstract operand types.
- Encoding and decoding operands is the primary source of complexity in X86 assemblers and disassemblers.
- We simplify our X86 library dramatically by:
  - Dividing the AOTs into eight categories.
  - Representing them with a simple abstract operand type description language, AOTDL.

X86 Abstract Operand Types
Exact Operand

Exact AOTs (this and subsequent tables are partial listings)

OAL	al	OCL	cl	OAX	ax	ODX	dx
OYb	byte ptr es:[edi]			OYw	word ptr es:[edi]		

- Some AOTs specify precisely one operand.

Mnemonic	Encoding Method	Stem Bytes	Abstract Operand Types
out	Ordinary	EE	ODX OAL

$$\underbrace{\texttt{EE}}_{\text{Stem}}\ \texttt{out dx, al}$$

- These are implicit to a given stem, and hence these operands are not encoded explicitly.

X86 Abstract Operand Types

Operand Description Language: Exact

Representing Exact AOTs in AOTDL

OAL	Exact(Gb(Al))
OCL	Exact(Gb(Cl))
OAX	Exact(Gw(Ax))
ODX	Exact(Gw(Dx))
OYb	Exact(Mem32(ES,Mb,Edi,None,0,None))
OYw	Exact(Mem32(ES,Mw,Edi,None,0,None))

- The AOTDL element `Exact` is used when the AOT specifies an operand exactly.

X86 Abstract Operand Types

Exact Memory Location, With Flexible Segment

`0Xb`	`byte ptr ds:[esi]`	`0Xw`	`word ptr ds:[esi]`
`0Xd`	`dword ptr ds:[esi]`		

- Some AOTs specify a precise memory location, whose segment may vary.
  - I.e., `byte ptr fs:[esi]` is admissible for `0Xb`.

Mnemonic	Encoding Method	Stem Bytes	Abstract Operand Types
`stosb`	Ordinary	`AA`	`0Xb`

$\underbrace{\texttt{AA}}_{\text{Stem}}$ `stosb ds:[esi]`

$\underbrace{\texttt{64}}_{\texttt{fs}\text{ Prefix}}$ $\underbrace{\texttt{AA}}_{\text{Stem}}$ `stosb fs:[esi]`

- These operands are encoded implicitly, though they may require a segment prefix.

X86 Abstract Operand Types

Operand Description Language: `ExactSeg`

Representing Exact AOTs with Flexible Segments in AOTDL

`OXb`	`ExactSeg(Mem32(DS,Mb,Esi,None,0,None))`
`OXw`	`ExactSeg(Mem32(DS,Mw,Esi,None,0,None))`
`OXd`	`ExactSeg(Mem32(DS,Md,Esi,None,0,None))`

- The AOTDL element `ExactSeg` is used when the AOT specifies an exact memory location whose segment may vary.

X86 Abstract Operand Types

Immediate Operands

`OIb`	Any 8-bit constant	`OIw`	Any 16-bit constant
`OOb`	A raw memory address	`OJz`	A `jmp` displacement

- Some AOTs specify immediate operands, i.e., encoded after the prefixes, stem, and MOD R/M.

Mnemonic	Encoding Method	Stem Bytes	Abstract Operand Types
`add`	Ordinary	04	`OAL` `OIb`

$$\underbrace{\texttt{04}}_{\text{Stem}}\quad\underbrace{\texttt{7F}}_{\text{Immediate }\texttt{OIb}}\qquad \texttt{add al, 7Fh}$$

X86 Abstract Operand Types

Operand Description Language: `ImmEnc`

Representing Immediate AOTs in AOTDL

`OIb`	`ImmEnc(Ib)`
`OIw`	`ImmEnc(Iw)`
`OOb`	`ImmEnc(MemExpr(DS,Mb))`
`OJz`	`ImmEnc(JccTarget)`

- The AOTDL element `ImmEnc(t)` is used when the AOT specifies an operand encoded as an immediate.
- Its parameter `t` specifies the type of the operand.

X86 Abstract Operand Types

Sign-Extended Immediate Operands

`OJb`	An 8-bit `jmp` displacement
`OIbv`	An 8-bit immediate, sign-extended

- Some AOTs specify 8-bit immediate operands that are sign-extended to a larger size.

Mnemonic	Encoding Method	Stem Bytes	Abstract Operand Types
`jmp`	Ordinary	EB	`OJb`

$$\texttt{.text:00401024}\ \underbrace{\texttt{EB}}_{\text{Stem}}\quad \underbrace{\texttt{FE}}_{\text{Immediate }\texttt{OJb}}\quad \texttt{jmp loc\_401024}$$

- The immediate `FE` is sign-extended and added to the next instruction's address to produce the jump target `loc_401024`.

X86 Abstract Operand Types

Operand Description Language: `SignedImm`

Representing Sign-Extended Immediate AOTs in AOTDL

`OJb`	`SignedImm(JccTarget)`

- The AOTDL element `SignedImm(t)` is used when the AOT specifies an operand encoded as an 8-bit immediate.
- Its parameter `t` specifies the type of the operand.

X86 Abstract Operand Types

Register from a Family

`OGb`	An 8-bit register	`OGw`	A 16-bit register
`OGd`	A 32-bit register	`OSw`	A segment register
`OCd`	A control register	`ODd`	A debug register
`OPd`	An MMX register	`OVq`	An XMM register

- Some AOTs specify a family of registers.
- The specific register is selected by MOD R/M.GGG.

Mnemonic	Encoding Method	Stem Bytes	Abstract Operand Types
`xor`	Ordinary	32	`OGb` `OEb`

$$\texttt{32 C3 add } \underbrace{\texttt{al}}_{\texttt{OGb}\ (\text{GGG})} \texttt{ , bl}$$

$$\overbrace{\text{Stem}}^{32} \qquad \overbrace{\underbrace{11}_{\text{MOD}}\ \underbrace{\mathbf{000}}_{\text{GGG}}\ \underbrace{011}_{\text{RM}}}^{\overbrace{\text{MOD R/M}}^{\text{C3}}}$$

X86 Abstract Operand Types

Operand Description Language: GPart

Representing Register Family AOTs in AOTDL

`OGb`	`GPart(Gb)`
`OGw`	`GPart(Gw)`
`OGd`	`GPart(Gd)`
`OSw`	`GPart(SegReg)`
`OCd`	`GPart(ControlReg)`
`ODd`	`GPart(DebugReg)`
`OPd`	`GPart(MMXReg)`
`OVq`	`GPart(XMMReg)`

- The AOTDL element `GPart(t)` is used when the AOT specifies a register family selected by MOD R/M.GGG.
- Its parameter `t` specifies the type of the operand.

X86 Abstract Operand Types

Register or Memory Expression

Some Register or Memory Expression AOTs

Operand Type	Register Type	Memory Access Size	Operand Type	Register Type	Memory Access Size
`OEb`	8-bit	8-bit	`OEw`	16-bit	16-bit
`OEd`	32-bit	32-bit	`OQq`	MMX	64-bit
`OWdq`	XMM	128-bit	`ORw`	16-bit	ILLEGAL
`OMb`	ILLEGAL	8-bit	`ORdMb`	32-bit	8-bit

- Some AOTs specify either a family of registers, or a memory expression of a certain access size.
- This information is contained in the MOD R/M.
- Examples follow on the next slide.

X86 Abstract Operand Types

Register or Memory Expression

Mnemonic	Encoding Method	Stem Bytes	Abstract Operand Types	
xor	Ordinary	32	OGb	OEb
mov	Ordinary	0F 20	ORd	OCd
movlps	Ordinary	0F 13	OMq	OVq
pmovsxbd	Size Prefix	0F 21	OVdq	OUdq_Md

$$\overbrace{\overbrace{\underbrace{11}_{\text{MOD}}\ \underbrace{010}_{\text{GGG}}\ \underbrace{111}_{\text{RM}}}^{\text{MOD R/M \#1}}}^{\text{D7}}$$

$$\overbrace{\overbrace{\underbrace{00}_{\text{MOD}}\ \underbrace{010}_{\text{GGG}}\ \underbrace{111}_{\text{RM}}}^{\text{MOD R/M \#2}}}^{\text{17}}$$

Instructions Above Encoded Using · · ·

MOD R/M #1		MOD R/M #2	
32 D7	xor dl, bh	32 17	xor dl, [edi]
0F 20 D7	mov edi, cr2	0F 20 17	ILLEGAL
0F 13 D7	ILLEGAL	0F 13 17	movlps [edi], xmm2
66 0F 21 D7	pmovsxbd xmm2, xmm7	66 0F 21 17	pmovsxbd xmm2, [edi]

X86 Abstract Operand Types

Operand Description Language: `RegOrMem`

Operand Type	Register Type	Memory Access Size	AOTDL
`OEb`	8-bit	8-bit	`RegOrMem(Gb,Mb)`
`OEw`	16-bit	16-bit	`RegOrMem(Gw,Mw)`
`OEd`	32-bit	32-bit	`RegOrMem(Gd,Md)`
`OQq`	MMX	64-bit	`RegOrMem(MMXReg,Mq)`
`OWdq`	XMM	128-bit	`RegOrMem(XMMReg,Mdq)`
`ORw`	16-bit	ILLEGAL	`RegOrMem(Gw,None)`
`OMb`	ILLEGAL	8-bit	`RegOrMem(None,Mb)`
`ORdMb`	32-bit	8-bit	`RegOrMem(Gd,Mb)`

- The AOTDL element `RegOrMem(r,m)` is used when the AOT may specify a register or memory location.
  - Its parameter `r` specifies the type of the register operand.
  - Its parameter `m` specifies the size of the memory operand.
  - Illegal operands are represented as `None`.

X86 Abstract Operand Types

Type Depends Upon Size Prefix

Operand Type	OPSIZE	No OPSIZE
`OeAX`	`ax`	`eax`
`OGv`	16-bit register	32-bit register
`OIv`	16-bit immediate	32-bit immediate
`OEv`	16-bit register or memory access	32-bit register or memory access

- Some AOTs differ if an OPSIZE prefix is specified.
- The behavior depends on the processor's current operating mode.
- An example is on the next slide.

X86 Abstract Operand Types

Type Depends Upon Size Prefix

Operand		No
Type	OPSIZE	OPSIZE
OeAX	ax	eax

	Encoding	Stem	Abstract
Mnemonic	Method	Bytes	Operand Types
inc	Ordinary	40	OeAX

$\underbrace{\texttt{40}}_{\text{Stem}}$ `inc eax`

$\underbrace{\texttt{66}}_{\text{OPSIZE Prefix}}\ \underbrace{\texttt{40}}_{\text{Stem}}$ `inc ax`

- The presence of OPSIZE chooses `ax` or `eax`.

X86 Abstract Operand Types

Operand Description Language: `SizePrefix`

Representing OPSIZE-Variant AOTs in AOTDL

Operand Type	AOTDL
`OeAX`	`SizePrefix(Exact(Gw(Ax)),Exact(Gd(Eax)))`
`OGv`	`SizePrefix(GPart(Gw),GPart(Gd))`
`OIv`	`SizePrefix(ImmEnc(Iw),ImmEnc(Id))`
`OEv`	`SizePrefix(RegOrMem(Gw,Mw),RegOrMem(Gd,Md))`

- The AOTDL element `SizePrefix(yes,no)` is used when the AOT depends upon the OPSIZE prefix.
  - `yes`: is the AOTDL to use when the prefix is present.
  - `no`: is the AOTDL to use when the prefix is absent.

X86 Abstract Operand Types

Type Depends Upon Address Size Prefix

Some AOTs Depending upon ADDRSIZE

Operand Type	ADDRSIZE	No ADDRSIZE
`OM`	`word`-sized memory access	`dword`-sized memory access

- Some AOTs differ if an ADDRSIZE prefix is specified.
- The behavior depends on the processor's current operating mode.
- An example is on the next slide.

X86 Abstract Operand Types

Type Depends Upon Address Size Prefix

Operand Type	ADDRSIZE	No ADDRSIZE
`OM`	`word`-sized memory access	`dword`-sized memory access

Mnemonic	Encoding Method	Stem Bytes	Abstract Operand Types
`lea`	Ordinary	`8D`	`OGv` `OM`

$$\underbrace{\texttt{8D}}_{\text{Stem}}\ \underbrace{\texttt{00}}_{\text{MOD R/M}}\quad \texttt{lea eax, }\underbrace{\texttt{dword ptr [eax]}}_{\texttt{OM}}$$

$$\underbrace{\texttt{67}}_{\text{ADDRSIZE Prefix}}\ \underbrace{\texttt{8D}}_{\text{Stem}}\ \underbrace{\texttt{00}}_{\text{MOD R/M}}\quad \texttt{lea eax, }\underbrace{\texttt{word ptr [bx+si]}}_{\texttt{OM}}$$

- ADDRSIZE chooses a `word` or `dword`-sized access.

X86 Abstract Operand Types

Operand Description Language: `AddrPrefix`

Representing ADDRSIZE-Variant AOTs in AOTDL

Operand Type	AOTDL
`OM`	`AddrPrefix(RegOrMem(None,Mw),RegOrMem(None,Md))`

- The AOTDL element `AddrPrefix(yes,no)` is used when the AOT depends upon the ADDRSIZE prefix.
    - `yes` the AOTDL to use when the prefix is present.
    - `no` the AOTDL to use when the prefix is absent.

X86 Abstract Operand Types

Language Definition

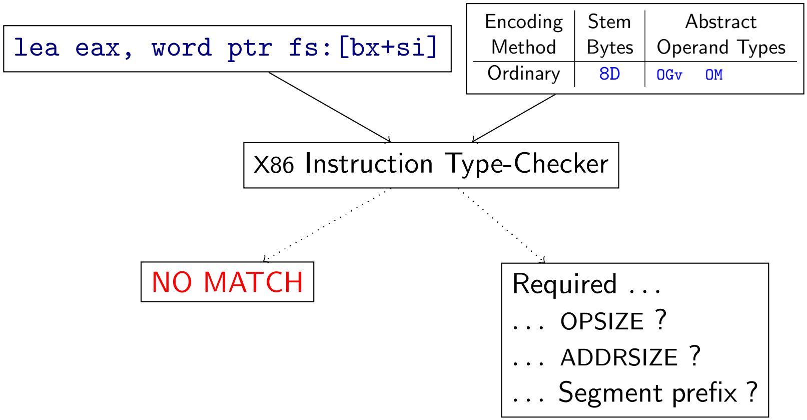
- Every AOT has a translation into AOTDL.

Language Element					Meaning
aotdl	:=	Exact	x86op		x86op exactly.
	|	ExactSeg	x86op		x86op memory expression whose segment may vary.
	|	Immediate	x86op		Immediate of type x86op.
	|	SignedImm	x86op		Sign-extended immediate of type x86op.
	|	GPart	regtype		Register from family regtype.
	|	ModRM	regtype	memsize	Register regtype or memory memsize. Either may be None.
	|	SizePrefix	aotdl	aotdl	aotdl with and without OPSIZE.
	|	AddrPrefix	aotdl	aotdl	aotdl with and without ADDRSIZE.

- This massively reduces the complexity of our X86 library.

X86 Type Checking

Overview



Encoding Method	Stem Bytes	Abstract Operand Types
Ordinary	8D	OGv OM

- The **instruction type-checker** decides whether a purported X86 instruction matches a given encoding.
- Additionally, it reports any prefixes required to encode it.

X86 Type Checking

Example

```
lea   eax   , word ptr fs:[bx+si]
       ▲
```

Encoding Method	Stem Bytes	Abstract Operand Types	
Ordinary	8D	OGv ▲	OM

Operand Type	AOTDL
OGv	SizePrefix(GPart(Gw),GPart(Gd)) ◀
OM	AddrPrefix(RegOrMem(None,Mw),RegOrMem(None,Md))

- Check the first operand against OGv.

X86 Type Checking

Example

```
lea   eax   , word ptr fs:[bx+si]
       ▲
```

Encoding Method	Stem Bytes	Abstract Operand Types	
Ordinary	8D	OGv ▲	OM

Operand Type	AOTDL
OGv	SizePrefix(GPart(Gw)◀,GPart(Gd)) ◀
OM	AddrPrefix(RegOrMem(None,Mw),RegOrMem(None,Md))

- Check the first operand against `GPart(Gw)`.
  - No match.

X86 Type Checking
Example

`lea eax , word ptr fs:[bx+si]`
▲

Encoding Method	Stem Bytes	Abstract Operand Types	
Ordinary	8D	OGv ▲	OM

Operand Type	AOTDL
OGv	`SizePrefix(GPart(Gw),GPart(Gd)◀) ◀`
OM	`AddrPrefix(RegOrMem(None,Mw),RegOrMem(None,Md))`

- Check the first operand against `GPart(Gd)`.
  - Matches. OPSIZE possible, not required: `SizePFX(False)`.

X86 Type Checking
Example

`lea eax , word ptr fs:[bx+si]`
▲

Encoding Method	Stem Bytes	Abstract Operand Types	
Ordinary	8D	OGv	OM
			▲

Operand Type	AOTDL
OGv	SizePrefix(GPart(Gw),GPart(Gd))
OM	AddrPrefix(RegOrMem(None,Mw),RegOrMem(None,Md)) ◀

- Check the first operand against `GPart(Gd)`.
  - Matches. OPSIZE possible, not required: `SizePFX(False)`.
- Check the second operand against `OM`.

X86 Type Checking

Example

`lea   eax   , word ptr fs:[bx+si]`
▲

Encoding Method	Stem Bytes	Abstract Operand Types	
Ordinary	8D	OGv	OM
			▲

Operand Type	AOTDL
OGv	SizePrefix(GPart(Gw),GPart(Gd))
OM	AddrPrefix(RegOrMem(None,Mw)◀,RegOrMem(None,Md)) ◀

- Check the first operand against `GPart(Gd)`.
  - Matches. OPSIZE possible, not required: `SizePFX(False)`.
- Check the second operand against `RegOrMem(None,Mw)`.

X86 Type Checking

Example

`lea eax , word ptr fs:[bx+si]`
▲

Encoding Method	Stem Bytes	Abstract Operand Types	
Ordinary	8D	OGv	OM ▲

Operand Type	AOTDL
OGv	`SizePrefix(GPart(Gw),GPart(Gd))`
OM	`AddrPrefix(RegOrMem(None◀,Mw)◀,RegOrMem(None,Md))` ◀

- Check the first operand against `GPart(Gd)`.
  - Matches. OPSIZE possible, not required: `SizePFX(False)`.
- Check the second operand against `None`.
  - No match.

X86 Type Checking
Example

```
lea   eax   , word ptr fs:[bx+si]
                      ▲
```

Encoding Method	Stem Bytes	Abstract Operand Types	
Ordinary	8D	OGv	OM ▲

Operand Type	AOTDL
OGv	`SizePrefix(GPart(Gw),GPart(Gd))`
OM	`AddrPrefix(RegOrMem(None,Mw◀)◀,RegOrMem(None,Md))` ◀

- Check the first operand against `GPart(Gd)`.
  - Matches. OPSIZE possible, not required: `SizePFX(False)`.
- Check the second operand against `Mw`.
  - Matches. ADDRSIZE possible, required: `AddrPFX(True)`.
  - `fs` prefix required: `SegPFX(FS)`.

X86 Type Checking

Mutual Compatibility

`xor eax, bx`

Encoding Method	Stem Bytes	Abstract Operand Types	
Ordinary	33	`OGv`	`OEv`

- Checking `eax` against `OGv` returns `SizePFX(False)`.
  - I.e., it only matches if OPSIZE is absent.
- Checking `ax` against `OEv` returns `SizePFX(True)`.
  - I.e., it only matches if OPSIZE is present.
- These operands are not mutually compatible.
  - This instruction is not valid for this encoding.

X86 Type Checking by Variety

- We use pattern-matching to describe type-checking concisely.

a	x	Type-Checking Logic	Return Value
Exact(o)	x	x == o	MATCHES
ExactSeg(o)	MemExpr(_)	x == o	MATCHES
ExactSeg(o)	MemExpr(_)	x == o(x.Seg)	SegPFX(x.Seg)
ImmEnc(Immediate(i))	x	type(i) == type(x)	MATCHES
ImmEnc(JccTarget(_))	JccTarget(_)		MATCHES
SignedImm(JccTarget(_))	JccTarget(_)		MATCHES
SignedImm(Id(_))	Id(i)	0xFFFFFF80 <= i <= 0x7F	MATCHES
SignedImm(Iw(_))	Iw(i)	0xFF80 <= i <= 0x7F	MATCHES
GPart(g)	x	type(g) == type(x)	MATCHES
RegOrMem(r,m)	Register(y)	type(r) == type(y)	MATCHES

typecheck_x86(a,x), Part 1

X86 Type Checking by Variety

Memory Expressions

a	x	Type-Checking Logic	Return Value
ImmEnc(FarTarget(_))	AP16(_)		AddrPFX(False)
ImmEnc(FarTarget(_))	AP32(_)		AddrPFX(True)
ImmEnc(MemExpr(_))	Mem32(_)	x.BaseReg == None and x.IndexReg == None	AddrPFX(False)
ImmEnc(MemExpr(_))	Mem16(_)	x.BaseReg == None and x.IndexReg == None	AddrPFX(True)
RegOrMem(r,m)	Mem32(_)	m.size == x.size	AddrPFX(False)
RegOrMem(r,m)	Mem16(_)	m.size == x.size	AddrPFX(True)

typecheck_x86(a,x), Continued

- 16-bit memory expressions require ADDRSIZE prefixes in 32-bit mode.

X86 Type Checking by Variety, Concluded

SizePrefix, AddrPrefix

typecheck_x86(a,x), for a = SizePrefix(y,n)

```
# Check prefix required AOTDL
yr = typecheck_x86(a.yes,x)
if yr & MATCHES:
  return SizePFX(True)  | yr

# Check prefix absent AOTDL
nr = typecheck_x86(a.no,x)
if nr & MATCHES:
  return SizePFX(False) | nr

return NOMATCH
```

- For a prefix-dependent AOTDL a, check both possibilities.
  - AddrPrefix(y,n) is as above, with AddrPFX instead of SizePFX.
- Return NOMATCH, or:
  1. An indication that the prefix was possible, and
  2. Whether the prefix was required.

Encoding X86 Instructions

Retrieve Encodings

$$\begin{tikzcd}
\texttt{add bx, 1234h} \arrow[d] \\
\boxed{\text{X86 Encoding Table}} \arrow[d] \\
{}
\end{tikzcd}$$

Encoding Method	Stem Bytes	Abstract Operand Types
Ordinary	00	`OEb` `OGb`
Ordinary	01	`OEv` `OGv`
MOD R/M Group #0	80	`OEb` `OIb`
MOD R/M Group #0	81	`OEv` `OIz`

Encoding Table for `add` (Partial)

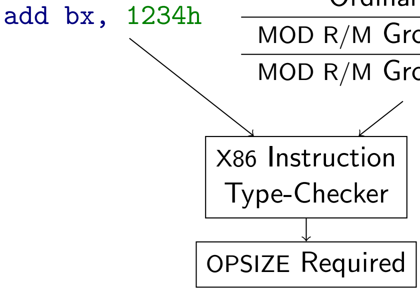
- First, look up the mnemonic's encodings.

Encoding X86 Instructions

Finding a Suitable Encoding

Encoding Table for `add` (Partial)

Ordinary	00	`OEb`	`OGb`	
Ordinary	01	`OEv`	`OGv`	
MOD R/M Group #0	80	`OEb`	`OIb`	
MOD R/M Group #0	81	`OEv`	`OIz`	◀



- Type-check the instruction against every encoding.
- Find the first one that matches ◀.
- Return it and the prefix information.

Encoding X86 Instructions
The Encoding Context

`add bx, 1234h` MOD R/M Group #0 | 81 | `0Ev` `0Iz`

Required Prefixes	Stem	MOD R/M	Immediates
OPSIZE	81		

Encoding Context

- Allocate an **encoding context** for subsequent use.
- Type-checking indicated OPSIZE was required.
- The stem is 81.

Encoding X86 Instructions

Attend to Encoding Method

`add bx, 1234h`   MOD R/M Group #0 | 81 | `0Ev`   `0Iz`

Required Prefixes	Stem	MOD R/M	Immediates
OPSIZE	81	$\overbrace{\underbrace{}_{\text{MOD}}\underbrace{000}_{\text{GGG}}\underbrace{}_{\text{RM}}}^{\underbrace{}_{}\text{MOD R/M}}$	

Encoding Method	Action
Ordinary	Do nothing.
Size Prefix	Set OPSIZE.
MOD R/M Group #n	Set MOD R/M.GGG to n.

- Perform the action required by the encoding method.

Encoding X86 Instructions

Emit Operands

`add bx , 1234h` MOD R/M Group #0 | 81 | `OEv OIz`

Required Prefixes	Stem	MOD R/M	Immediates
OPSIZE	81	$\overbrace{\underbrace{\quad\quad}_{\text{MOD}}\underbrace{000}_{\text{GGG}}\underbrace{\quad\quad}_{\text{RM}}}^{\text{MOD R/M}}$	

Operand Type	AOTDL
`OEv`	`SizePrefix(RegOrMem(Gw,Mw),RegOrMem(Gd,Md))`
`OIz`	`SizePrefix(ImmEnc(Iw),ImmEnc(Id))`

- Encode the first operand using `OEv`.
  - OPSIZE present, so look at prefixed part.

Encoding X86 Instructions

Emit Operands

```
add  bx  , 1234h      MOD R/M Group #0 | 81 | 0Ev  0Iz
     ▲                                        ▲
```

Required Prefixes	Stem	MOD R/M	Immediates
OPSIZE	81	$\overbrace{\underbrace{11}_{\text{MOD}}\;\underbrace{000}_{\text{GGG}}\;\underbrace{}_{\text{RM}}}^{\underbrace{}_{\text{MOD R/M}}}$	

Operand Type	AOTDL
`0Ev`	`SizePrefix(RegOrMem(Gw,Mw)`◀`,RegOrMem(Gd,Md))` ◀
`0Iz`	`SizePrefix(ImmEnc(Iw),ImmEnc(Id))`

- Encode the first operand using `RegOrMem(Gw,Mw)`.
  - Operand is a register, so set MOD R/M.MOD to 11.

Encoding X86 Instructions

Emit Operands

`add bx , 1234h` MOD R/M Group #0 | 81 | `OEv OIz`

▲ ▲

Required Prefixes	Stem	MOD R/M	Immediates
OPSIZE	81	$\underbrace{\text{C3}}_{\text{MOD R/M}}$ $\overbrace{\underbrace{11}_{\text{MOD}}\ \underbrace{000}_{\text{GGG}}\ \underbrace{011}_{\text{RM}}}$	

Operand Type	AOTDL
`OEv`	`SizePrefix(RegOrMem(Gw◀,Mw),RegOrMem(Gd,Md))` ◀
`OIz`	`SizePrefix(ImmEnc(Iw),ImmEnc(Id))`

- Encode the first operand using `Gw`.
    - Set MOD R/M.RM to 011 (`bx`'s register number).

Encoding X86 Instructions

Emit Operands

`add bx , 1234h`   MOD R/M Group #0 | 81 | `OEv` `OIz`

▲ (under `1234h`) ▲ (under `OIz`)

Required Prefixes	Stem	MOD R/M	Immediates
OPSIZE	81	$\underbrace{\texttt{C3}}_{\text{MOD R/M}}$ $\overbrace{\underbrace{11}_{\text{MOD}}\ \underbrace{000}_{\text{GGG}}\ \underbrace{011}_{\text{RM}}}$	

Operand Type	AOTDL
`OEv`	`SizePrefix(RegOrMem(Gw,Mw),RegOrMem(Gd,Md))`
`OIz`	`SizePrefix(ImmEnc(Iw),ImmEnc(Id))` ◀

- Encode the first operand using `Gw`.
  - Set MOD R/M.RM to 011 (`bx`'s register number).
- Encode the second operand using `OIz`.
  - OPSIZE present, so look at prefixed part.

Encoding X86 Instructions

Emit Operands

`add bx , 1234h` MOD R/M Group #0 | 81 | `OEv` `OIz`

▲ ▲

Required Prefixes	Stem	MOD R/M	Immediates
OPSIZE	81	$\underbrace{\texttt{C3}}_{\text{MOD R/M}}$ $\overbrace{\underbrace{11}_{\text{MOD}}\ \underbrace{000}_{\text{GGG}}\ \underbrace{011}_{\text{RM}}}$	34 12◀

Operand Type	AOTDL
`OEv`	`SizePrefix(RegOrMem(Gw,Mw),RegOrMem(Gd,Md))`
`OIz`	`SizePrefix(ImmEnc(Iw)`◀`,ImmEnc(Id))` ◀

- Encode the first operand using `Gw`.
  - Set MOD R/M.RM to 011 (`bx`'s register number).
- Encode the second operand using `ImmEnc(Iw)`.
  - Encode as an immediate ◀.

Encoding X86 Instructions

Producing X86 Machine Code

`add bx, 1234h` MOD R/M Group #0 | 81 | `OEv` `OIz`

Required Prefixes	Stem	MOD R/M	Immediates
OPSIZE (66)	81	$\underbrace{\text{C3}}_{\text{MOD R/M}}$ $\overbrace{\underbrace{11}_{\text{MOD}}\ \underbrace{000}_{\text{GGG}}\ \underbrace{011}_{\text{RM}}}$	34 12

66 81 C3 34 12

- Concatenate all fields together to produce X86 machine code.

Encoding X86 Operands by Variety

- Pattern-matching describes operand encoding concisely.

a	x	Encoder Logic
`Exact(o)`	`_`	
`ExactSeg(o)`	`_`	
`ImmEnc(Immediate(_))`	`Id(i)`	`immediates.append(i,4)`
`ImmEnc(Immediate(_))`	`Iw(i)`	`immediates.append(i,2)`
`ImmEnc(FarTarget(_))`	`AP32(s,o)`	`immediates.append(s,2)` `immediates.append(o,4)`
`ImmEnc(FarTarget(_))`	`AP16(s,o)`	`immediates.append(s,2)` `immediates.append(o,2)`
`ImmEnc(MemExpr(_))`	`Mem32(_)`	`immediates.append(x.Disp,4)`
`ImmEnd(MemExpr(_))`	`Mem16(_)`	`immediates.append(x.Disp,2)`
`SignedImm(Immediate(_))`	`_`	`immediates.append(x.value,2)`
`GPart(g)`	`_`	`ModRM.GGG = x.IntValue()`
`RegOrMem(r,m)`	`Register(y)`	`ModRM.RM = y.IntValue()` `ModRM.MOD = 3`
`RegOrMem(r,m)`	`Mem32(_)`	`EncodeModRM32(x,m)`
`RegOrMem(r,m)`	`Mem16(_)`	`EncodeModRM16(x,m)`

`encode_x86(a,x)`, Part 1

Encoding X86 Operands by Variety, Continued

`SizePrefix`, `AddrPrefix`

`a`	`x`	Encoder Logic
`SizePrefix(y,n)`	`_`	`if sizepfx: encode_x86(y,x)` `else: encode_x86(n,x)`
`AddrPrefix(y,n)`	`_`	`if addrpfx: encode_x86(y,x)` `else: encode_x86(n,x)`

`encode_x86(a,x)`, Continued

- For prefix-dependent AOTDL elements:
  - Check the encoder context to see whether the prefix is present.
  - Call `encode_x86` on `y` if it is, or `n` if it is not.

Encoding X86 Operands by Variety, Concluded

X86 Operand JccTarget

a	x	Encoder Logic
`ImmEnc(JccTarget(t,f))`	`JccTarget(_)`	`immediates.append(t - next_addr,4)`
`SignedImm(JccTarget(t,f))`	`JccTarget(_)`	`immediates.append(t - next_addr,4)`

`encode_x86(a,x)`, Concluded

- For simplicity, we encode all relative jumps as 32-bit immediates.

X86 Decoder Table

- Varities
  - Direct
  - Invalid
  - Depends Upon OPSIZE
  - Depends Upon ADDRSIZE
  - Depends Upon MOD R/M.GGG
  - Depends Upon MOD R/M.MOD
  - Depends Upon MOD R/M.RM
  - Depends Upon SSE Prefixes
- Decoder Entry Description Language

X86 Decoder Entries

Overview

	0	1	2	3	4	5	6	7
0	ADD						PUSH ES	POP ES
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
1	ADC						PUSH SS	POP SS
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
2	AND						SEG=ES (Prefix)	DAA
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
3	XOR						SEG=SS (Prefix)	AAA
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
4	INC general register							
	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI
5	PUSH general register							
	rAX	rCX	rDX	rBX	rSP	rBP	rSI	rDI
6	PUSHA/ PUSHAD	POPA/ POPAD	BOUND Gv, Ma	ARPL Ew, Gw	SEG=FS (Prefix)	SEG=GS (Prefix)	Operand Size (Prefix)	Address Size (Prefix)
7	Jcc, Jb - Short-displacement jump on condition							
	O	NO	B/NAE/C	NB/AE/NC	Z/E	NZ/NE	BE/NA	NBE/A
8	Immediate Grp 1				TEST		XCHG	
	Eb, Ib	Ev, Iz	Eb, Ib	Ev, Ib	Eb, Gb	Ev, Gv	Eb, Gb	Ev, Gv

- As we had an encoder table, we have a **decoder table**.
- We represent the opcode maps as a table of **decoder entries** using a decoder language DECDL, like AOTDL but simpler.
  - Fidelity to the manuals, maintainability, ease of programming.

X86 Decoder Entries

`Direct`

	0	1	2	3	4	5	6	7
0	ADD						PUSH	POP
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	ES	ES

```
Direct(Add,[OEb,OGb])
Direct(Add,[OEv,OGv])
Direct(Add,[OGb,OEb])
Direct(Add,[OGv,OEv])
Direct(Add,[OAL,OIb])
Direct(Add,[OrAX,OIz])
Direct(Push,[OES])
Direct(Pop,[OES])
```

- Most decoder entries are of type `Direct(mnem,oplist)`.
  - `mnem`: Mnemonic
  - `oplist`: Abstract operand types

X86 Decoder Entries

`Invalid`

- Many instruction stems have no legal encodings.
  - These entries are blank in the Intel opcode maps.
- We represent them using the element `Invalid`.

X86 Decoder Entries

PredOpSize

- Some instruction stems specify different decoder entries if OPSIZE is present.

```
9C | PredOpSize(Direct(Pushfw,[]),Direct(Pushfd,[]))
9D | PredOpSize(Direct(Popfw,[]), Direct(Popfd,[]))
A5 | PredOpSize(Direct(Movsw,[OXv,OYv]),Direct(Movsd,[OXv,OYv]))
A7 | PredOpSize(Direct(Cmpsw,[OXv,OYv]),Direct(Cmpsd,[OXv,OYv]))
AB | PredOpSize(Direct(Stosw,[OYv]),Direct(Stosd,[OYv]))
AD | PredOpSize(Direct(Lodsw,[OXv]),Direct(Lodsd,[OXv]))
AF | PredOpSize(Direct(Scasw,[OYv]),Direct(Scasd,[OYv]))
```

X86 Decoder Entries

PredAddrSize

- Some instruction stems specify different decoder entries if ADDRSIZE is present.

E3	`PredAddrSize(Direct(Jcxz,[OJb]),Direct(Jecxz,[OJb]))`

Instruction Groups

- The X86 instruction features **instruction groups**, with multiple opcodes encoded under one stem.
- These stems each take a MOD R/M, and use various parts of the MOD R/M to choose the opcode:
    1. GGG
    2. GGG and MOD
    3. GGG, MOD, and RM
    4. GGG, MOD, RM, and prefixes

Instruction Groups

Distinguishment Based on GGG

- In some groups, MOD R/M.GGG selects the opcode.

				GGG							
Opcode	Group	MOD	prefix	000	001	**010**	011	100	**101**	110	111
80-83	1			ADD	OR	ADC	SBB	AND	SUB	XOR	CMP

`80 D0 7F adc al, 7Fh`

$$\underbrace{\texttt{80}}_{\text{Stem}} \quad \underbrace{\texttt{D0}}_{\text{MOD R/M}} \quad \underbrace{\texttt{7F}}_{\text{Immediate}}$$

$$\overbrace{\underbrace{11}_{\text{MOD}}\ \underbrace{\mathbf{010}}_{\text{GGG}}\ \underbrace{000}_{\text{RM}}}$$

`80 E9 7F sub cl, 7Fh`

$$\underbrace{\texttt{80}}_{\text{Stem}} \quad \underbrace{\texttt{E9}}_{\text{MOD R/M}} \quad \underbrace{\texttt{7F}}_{\text{Immediate}}$$

$$\overbrace{\underbrace{11}_{\text{MOD}}\ \underbrace{\mathbf{101}}_{\text{GGG}}\ \underbrace{001}_{\text{RM}}}$$

X86 Decoder Entries

Group

				GGG							
Opcode	Group	MOD	prefix	000	001	**010**	011	100	**101**	110	111
80	1			ADD	OR	ADC	SBB	AND	SUB	XOR	CMP

GGG	Group() Parameter #GGG
0	`Direct(Add,[OEb,OIb])`
1	`Direct(Or, [OEb,OIb])`
2	`Direct(Adc,[OEb,OIb])`
3	`Direct(Sbb,[OEb,OIb])`
4	`Direct(And,[OEb,OIb])`
5	`Direct(Sub,[OEb,OIb])`
6	`Direct(Xor,[OEb,OIb])`
7	`Direct(Cmp,[OEb,OIb])`

- The element `Group` takes eight parameters, one decoder entry for each value of MOD R/M.GGG (in order).

Instruction Groups

Distinguishing Based on MOD

- In some groups, different opcodes are selected depending upon whether MOD = 11, i.e. whether or not it specifies a register.

				GGG							
Opcode	Group	MOD	prefix	000	001	010	011	100	101	110	111
0F 18	16	mem		prefetch NTA	prefetch T0	prefetch T1	prefetch T2				
		11b									

`0F 18 00 prefetchnta [eax]`

$$\underbrace{\texttt{0F 18}}_{\text{Stem}} \qquad \underbrace{\texttt{00}}_{\text{MOD R/M}}$$

$$\overbrace{\underbrace{\mathbf{00}}_{\text{MOD}}\ \underbrace{000}_{\text{GGG}}\ \underbrace{000}_{\text{RM}}}$$

`0F 18 C0 (invalid)`

$$\underbrace{\texttt{0F 18}}_{\text{Stem}} \qquad \underbrace{\texttt{C0}}_{\text{MOD R/M}}$$

$$\overbrace{\underbrace{\mathbf{11}}_{\text{MOD}}\ \underbrace{000}_{\text{GGG}}\ \underbrace{000}_{\text{RM}}}$$

X86 Decoder Entries

PredMOD

				GGG							
Opcode	Group	MOD	prefix	000	001	010	011	100	101	110	111
0F 18	16	mem		prefetch NTA	prefetch T0	prefetch T1	prefetch T2				
		11b									

```
g = Group(...)
```

GGG	Group() Parameter #GGG
0	`Direct(Prefetchnta,[OEv])`
1	`Direct(Prefetcht0, [OEv])`
2	`Direct(Prefetcht1, [OEv])`
3	`Direct(Prefetcht2, [OEv])`
4	`Invalid`
5	`Invalid`
6	`Invalid`
7	`Invalid`

- The element `PredMOD(m,r)` has two parameters:
  - `m`: the decoder entry to use if MOD R/M.MOD specifies memory.
  - `r`: the decoder entry to use for registers.
- This group is represented by `PredMOD(g,Invalid)`.
  - `g` is shown to the left.

Instruction Groups

Distinguishing Based on RM

- In some groups, when the MOD specifies a register, the RM instead is used to specify an opcode (and not a register).

				GGG							
Opcode	Group	MOD	prefix	000	001	010	011	100	101	110	111
C6	11	mem		MOV Eb, Ib							
		11b									XABORT (000) Ib

`C6 F8 7F xabort 7Fh`

$$\underbrace{\texttt{C6}}_{\text{Stem}}\quad \underbrace{\texttt{F8}}_{\text{MOD R/M}}\quad \underbrace{\texttt{7F}}_{\text{Immediate}}$$

$$\overbrace{\underbrace{11}_{\text{MOD}}\ \underbrace{111}_{\text{GGG}}\ \underbrace{\mathbf{000}}_{\text{RM}}}$$

`C6 F9 7F (invalid)`

$$\underbrace{\texttt{C6}}_{\text{Stem}}\quad \underbrace{\texttt{F9}}_{\text{MOD R/M}}\quad \underbrace{\texttt{7F}}_{\text{Immediate}}$$

$$\overbrace{\underbrace{11}_{\text{MOD}}\ \underbrace{111}_{\text{GGG}}\ \underbrace{\mathbf{001}}_{\text{RM}}}$$

X86 Decoder Entries

RMGroup

				GGG							
Opcode	Group	MOD	prefix	000	001	010	011	100	101	110	111
C6	11	mem		MOV Eb, Ib							
		11b									XABORT (000) Ib

l = RMGroup(...)   m = Group(...)

g = Group(...)

i	RMGroup(i)
0	Direct(Xabort,[OIb])
1	Invalid
2	Invalid
3	Invalid
4	Invalid
5	Invalid
6	Invalid
7	Invalid

i	Group(i)
0	Invalid
1	Invalid
2	Invalid
3	Invalid
4	Invalid
5	Invalid
6	Invalid
7	l

i	Group(i)
0	Direct(Mov,[OEb,OIb])
1	Invalid
2	Invalid
3	Invalid
4	Invalid
5	Invalid
6	Invalid
7	Invalid

- We represent this group as PredMOD(m,g).

Instruction Groups

Distinguishing Based on Prefix

- In some groups, different opcodes are selected depending upon whether certain prefixes are present.

				GGG							
Opcode	Group	MOD	prefix	000	001	010	011	100	101	110	111
0F AE	15	mem		fxsave	fxrstor	ldmxcsr	stmxcsr	XSAVE	XRSTOR	XSAVEOPT	clflush
		11b							lfence	mfence	sfence
			F3	RDFSBASE Ry	RDGSBASE Ry	WRFSBASE Ry	WRGSBASE Ry				

`0F AE F8 sfence`

`F3 0F AE C8 rdgsbase eax`

$$\underbrace{\texttt{0F AE}}_{\text{Stem}} \quad \underbrace{\texttt{F8}}_{\text{MOD R/M}}$$

$$\overbrace{\underbrace{11}_{\text{MOD}}\ \underbrace{\mathbf{111}}_{\text{GGG}}\ \underbrace{000}_{\text{RM}}}$$

$$\underbrace{\texttt{F3}}_{\text{Prefix}}\ \underbrace{\texttt{0F AE}}_{\text{Stem}} \quad \underbrace{\texttt{C8}}_{\text{MOD R/M}}$$

$$\overbrace{\underbrace{11}_{\text{MOD}}\ \underbrace{\mathbf{001}}_{\text{GGG}}\ \underbrace{000}_{\text{RM}}}$$

- We represent these with `SSE`, which we discuss next.

SSE Encoding

pfx	0F D0	0F D1	0F D2	0F D3	0F D4	0F D5	0F D6	0F D7
		psrlw Pq, Qq	psrld Pq, Qq	psrlq Pq, Qq	paddq Pq, Qq	pmullw Pq, Qq		pmovmskb Gd, Nq
66	vaddsubpd Vpd, Hpd, Wpd	vpsrlw Vx, Hx, Wx	vpsrld Vx, Hx, Wx	vpsrlq Vx, Hx, Wx	vpaddq Vx, Hx, Wx	vpmullw Vx, Hx, Wx	vmovq Wq, Vq	vpmovmskb Gd, Ux
F3							movq2dq Vdq, Nq	
F2	vaddsubps Vps, Hps, Wps						movdq2q Pq, Uq	

- Several SSE instructions are encoded under the same stem.
- The OPSIZE, REP, and REPNZ prefixes select a decoder entry.
    - I.e., SSE encodings are like a group, but the decoder entry is selected by prefixes instead of MOD R/M.GGG.

X86 Decoder Entries

SSE

pfx	0F D0	0F D1	0F D2	0F D3	0F D4	0F D5	0F D6	0F D7
		psrlw Pq, Qq	psrld Pq, Qq	psrlq Pq, Qq	paddq Pq, Qq	pmullw Pq, Qq		pmovmskb Gd, Nq
66	vaddsubpd Vpd, Hpd, Wpd	vpsrlw Vx, Hx, Wx	vpsrld Vx, Hx, Wx	vpsrlq Vx, Hx, Wx	vpaddq Vx, Hx, Wx	vpmullw Vx, Hx, Wx	vmovq Wq, Vq	vpmovmskb Gd, Ux
F3							movq2dq Vdq, Nq	
F2	vaddsubps Vps, Hps, Wps						movdq2q Pq, Uq	

Decoder Entry for `0F D0`

Prefix	`SSE()` Parameter
NONE	`Invalid`
OPSIZE	`Direct(Vaddsupbd,[OVpd,OHpd,OWpd])`
REP	`Invalid`
REPNZ	`Direct(Vaddsupbs,[OVps,OHps,OWps])`

Decoder Entry for `0F D7`

Prefix	`SSE()` Parameter
NONE	`Direct(Pmovmskb, [OGd,ONq])`
OPSIZE	`Direct(Vpmovmskb,[OGq,OUx])`
REP	`Invalid`
REPNZ	`Invalid`

X86 Decoder Entries

DECDL Language

Language Element					Meaning
decdl	:=	Direct	mnem	[aotdl]	Mnemnonic mnem, operands [aotdl].
	|	Invalid			Illegal instruction.
	|	PredMOD	decdl	decdl	decdl for MOD $\neq 3$ and MOD $= 3$.
	|	PredOpSize	decdl	decdl	decdl with and without OPSIZE.
	|	PredAddrSize	decdl	decdl	decdl with and without ADDRSIZE.
	|	Group	[decdl]*8		One decdl entry for each MOD R/M.GGG.
	|	RMGroup	[decdl]*8		One decdl entry for each MOD R/M.RM.
	|	SSE	[decdl]*4		One decdl entry for each SSE prefix.

- We represent the Intel opcode maps as an array of elements of this description language.

Decoding X86 Instructions

Overview: Decode Prefixes and Stem

Given a stream of bytes:

```
66 6B 84 1F 78 56 34 12 9A
```

Decoding X86 Instructions

Overview: Decode Prefixes and Stem

Given a stream of bytes:

1. Consume prefixes.
   - 66 is OPSIZE.

```
66 6B 84 1F 78 56 34 12 9A
```

$\underbrace{66}_{\text{Prefix}}$

Decoding X86 Instructions

Overview: Decode Prefixes and Stem

Given a stream of bytes:

1. Consume prefixes.
   - `66` is OPSIZE.
2. Consume stem, retrieve corresponding decoder entry.
   - Entry is `Direct(Imul,[OGv,OEv,OIb])`.

`66 6B 84 1F 78 56 34 12 9A`

$\underbrace{\texttt{66}}_{\text{Prefix}}\ \underbrace{\texttt{6B}}_{\text{Stem}}$

Decoding X86 Instructions

Overview: Decode Prefixes and Stem

Given a stream of bytes:

1. Consume prefixes.
   - `66` is OPSIZE.
2. Consume stem, retrieve corresponding decoder entry.
   - Entry is `Direct(Imul,[OGv,OEv,OIb])`.
3. Process decoder entry.
   - Decode operands (shown on next slide).

```
66 6B 84 1F 78 56 34 12 9A
```

$\underbrace{\texttt{66}}_{\text{Prefix}}\ \underbrace{\texttt{6B}}_{\text{Stem}}$

Decoding X86 Instructions

Overview: Decode Operands

imul | OGv OEv OIb
▲

Operand Type	AOTDL
OGv	SizePrefix(GPart(Gw),GPart(Gd)) ◀
OEv	SizePrefix(RegOrMem(Gw,Mw),RegOrMem(Gd,Md))
OIb	ImmEnc(Ib)

- Decode the first operand using OGv.
    - OPSIZE present, so look at prefixed part.

66 6B 84 1F 78 56 34 12 9A

$\underbrace{\text{66}}_{\text{Prefix}}\ \underbrace{\text{6B}}_{\text{Stem}}$

Decoding X86 Instructions

Overview: Decode Operands

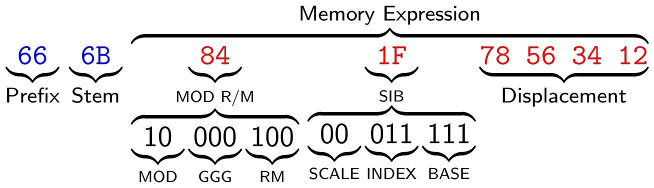
`imul` | `OGv` `OEv` `OIb`

▲

Operand Type	AOTDL
`OGv`	`SizePrefix(GPart(Gw)`◀`,GPart(Gd))` ◀
`OEv`	`SizePrefix(RegOrMem(Gw,Mw),RegOrMem(Gd,Md))`
`OIb`	`ImmEnc(Ib)`

- Decode the first operand using `GPart(Gw)`
  - Decode MOD R/M, return `OGw` register numbered GGG.

`66 6B 84 1F 78 56 34 12 9A`



Decoding X86 Instructions

Overview: Decode Operands

```
imul | OGv   OEv   OIb
              ▲
```

Operand Type	AOTDL
`OGv`	`SizePrefix(GPart(Gw),GPart(Gd))`
`OEv`	`SizePrefix(RegOrMem(Gw,Mw),RegOrMem(Gd,Md))` ◀
`OIb`	`ImmEnc(Ib)`

- Decode the second operand using `OEv`.
  - First operand is `ax`.
  - OPSIZE present, so look at prefixed part.

`66 6B 84 1F 78 56 34 12 9A`

$$\underbrace{66}_{\text{Prefix}}\ \underbrace{6B}_{\text{Stem}}\ \overbrace{\underbrace{84}_{\substack{\text{MOD R/M}\\ \overbrace{\underbrace{10}_{\text{MOD}}\ \underbrace{000}_{\text{GGG}}\ \underbrace{100}_{\text{RM}}}}}\ \underbrace{1F}_{\substack{\text{SIB}\\ \overbrace{\underbrace{00}_{\text{SCALE}}\ \underbrace{011}_{\text{INDEX}}\ \underbrace{111}_{\text{BASE}}}}}\ \underbrace{78\ 56\ 34\ 12}_{\text{Displacement}}}^{\text{Memory Expression}}$$

Decoding X86 Instructions

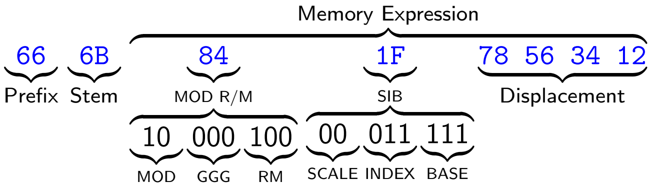
Overview: Decode Operands

imul | OGv OEv OIb
▲

Operand Type	AOTDL
OGv	SizePrefix(GPart(Gw),GPart(Gd))
OEv	SizePrefix(RegOrMem(Gw,Mw) ◀,RegOrMem(Gd,Md)) ◀
OIb	ImmEnc(Ib)

- Decode the second operand using RegOrMem(Gw,Mw).
  - First operand is ax.
  - MOD R/M specifies a memory location.

66 6B 84 1F 78 56 34 12 9A



Decoding X86 Instructions

Overview: Decode Operands

`imul | OGv  OEv  OIb`

▲

Operand Type	AOTDL
`OGv`	`SizePrefix(GPart(Gw),GPart(Gd))`
`OEv`	`SizePrefix(RegOrMem(Gw,Mw◀),RegOrMem(Gd,Md))` ◀
`OIb`	`ImmEnc(Ib)`

- Decode the second operand using `Mw`.
  - First operand is `ax`.
  - Decode a `word`-sized memory expression.

`66 6B 84 1F 78 56 34 12 9A`

$$\underbrace{66}_{\text{Prefix}}\ \underbrace{6B}_{\text{Stem}}\ \overbrace{\underbrace{84}_{\text{MOD R/M}}\ \underbrace{1F}_{\text{SIB}}\ \underbrace{78\ 56\ 34\ 12}_{\text{Displacement}}}^{\text{Memory Expression}}$$

$$\overbrace{\underbrace{10}_{\text{MOD}}\ \underbrace{000}_{\text{GGG}}\ \underbrace{100}_{\text{RM}}}\ \overbrace{\underbrace{00}_{\text{SCALE}}\ \underbrace{011}_{\text{INDEX}}\ \underbrace{111}_{\text{BASE}}}$$

Decoding X86 Instructions

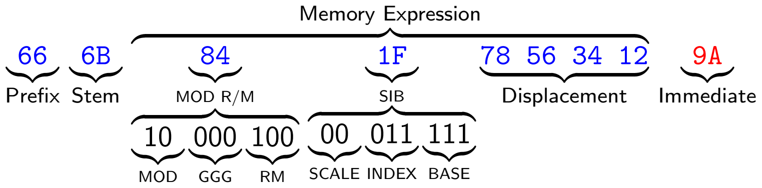
Overview: Decode Operands

imul | OGv OEv OIb

Operand Type	AOTDL
OGv	SizePrefix(GPart(Gw),GPart(Gd))
OEv	SizePrefix(RegOrMem(Gw,Mw),RegOrMem(Gd,Md))
OIb	ImmEnc(Ib) ◀

- Decode the third operand using `ImmEnc(Ib)`.
  - First operand is `ax`.
  - Second operand is `word ptr [edi+ebx+12345678h]`.
  - Retrieve a byte from the instruction stream.

`66 6B 84 1F 78 56 34 12 9A`



Decoding X86 Instructions

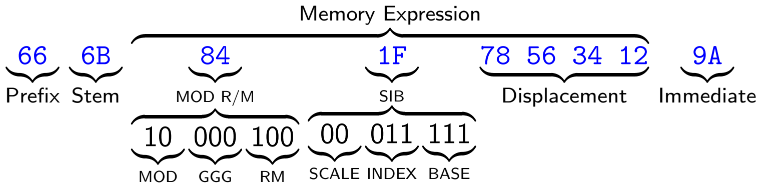
Overview: Decode Operands

`imul` | `OGv` `OEv` `OIb`

Operand Type	AOTDL
`OGv`	`SizePrefix(GPart(Gw),GPart(Gd))`
`OEv`	`SizePrefix(RegOrMem(Gw,Mw),RegOrMem(Gd,Md))`
`OIb`	`ImmEnc(Ib)`

- Decode the third operand using `ImmEnc(Ib)`.
  - First operand is `ax`.
  - Second operand is `word ptr [edi+ebx+12345678h]`.
  - Third operand is `9Ah`.

`66 6B 84 1F 78 56 34 12 9A`



Instruction is `imul ax, word ptr [edi+ebx+12345678h], 9Ah`

Decoding X86 Instructions

Consume Prefixes

Group #1		Group #2		Group #3		Group #4	
lock	F0	cs	2E	OPSIZE	66	ADDRSIZE	67
rep	F3	ss	36				
repz	F2	ds	3E				
repnz	F2	es	26				
		fs	64				
		gs	65				

- Consume bytes from the stream until the first non-prefix.
- Keep track of the order of Group #1 prefixes.
    - SSE decoding requires this.

Decoding X86 Instructions

Consume Stem

Stem Bytes	Table Index
`aa`	`0xAA`
`0F bb`	`0x1BB`
`0F 38 cc`	`0x2CC`
`0F 3A dd`	`0x3DD`

- Consume up to three bytes and return an integer from `0x000` to `0x3FF`.
- Retrieve that decoder entry from our decoder table.

Decoding X86 Instructions

Process DECDL Decoder Entry

DECDL	Condition	Action
Direct(m,o)		Decode operands
Invalid		raise InvalidInstruction
PredMOD(m,r)	ModRM.MOD != 3	decode(m)
PredMOD(m,r)		decode(r)
PredOpSize(y,n)	sizepfx	decode(y)
PredOpSize(y,n)		decode(n)
PredAddrSize(y,n)	addrpfx	decode(y)
PredAddrSize(y,n)		decode(n)
Group(l)		decode(l[ModRM.GGG])
RMGroup(l)		decode(l[ModRM.RM])
SSE(n,r,s,z)		(Complex logic)

- DECDL makes instruction decoding very simple.

Decoding X86 Instructions

SSE for Multiple Prefixes

pfx	0F D0	0F D1	0F D2	0F D3	0F D4	0F D5	0F D6	0F D7
		psrlw Pq, Qq	psrld Pq, Qq	psrlq Pq, Qq	paddq Pq, Qq	pmullw Pq, Qq		pmovmskb Gd, Nq
66	vaddsubpd Vpd, Hpd, Wpd	vpsrlw Vx, Hx, Wx	vpsrld Vx, Hx, Wx	vpsrlq Vx, Hx, Wx	vpaddq Vx, Hx, Wx	vpmullw Vx, Hx, Wx	vmovq Wq, Vq	vpmovmskb Gd, Ux
F3							movq2dq Vdq, Nq	
F2	vaddsubps Vps, Hps, Wps						movdq2q Pq, Uq	

Behavior for multiple prefixes is not well-documented by Intel.

1. If `rep` (`F3`) or `repe` (`F2`) were present:
    - For the prefix closest to the stem, if that DECDL entry is not `Invalid`, use it.
    - Otherwise, look at the next prefix further from the stem.
2. If OPSIZE was present and that DECDL entry is not `Invalid`, use it.
3. Otherwise, use the unprefixed entry.

Decoding X86 Instructions

SSE for Multiple Prefixes

pfx	0F D0	0F D1	0F D2	0F D3	0F D4	0F D5	0F D6	0F D7
		psrlw Pq, Qq	psrld Pq, Qq	psrlq Pq, Qq	paddq Pq, Qq	pmullw Pq, Qq		pmovmskb Gd, Nq
66	vaddsubpd Vpd, Hpd, Wpd	vpsrlw Vx, Hx, Wx	vpsrld Vx, Hx, Wx	vpsrlq Vx, Hx, Wx	vpaddq Vx, Hx, Wx	vpmullw Vx, Hx, Wx	vmovq Wq, Vq	vpmovmskb Gd, Ux
F3							movq2dq Vdq, Nq	
F2	vaddsubps Vps, Hps, Wps						movdq2q Pq, Uq	

```
F2 0F D6 C0          movdq2q     mm0, xmm0
66 F2 0F D6 C0       movdq2q     mm0, xmm0
F2 66 0F D6 C0       movdq2q     mm0, xmm0
F3 F2 66 0F D6 C0    movdq2q     mm0, xmm0
F2 F3 66 0F D6 C0    movq2dq     xmm0, mm0
F2 66 0F D7 C0       vpmovmskb   eax, xmm0
66 F2 0F D7 C0       vpmovmskb   eax, xmm0
```

Decoding X86 Operands by Variety

- Pattern-matching concisely describes operand decoding.

a	Condition	Return Value
`Exact(o)`		`return o`
`ExactSeg(o)`	`segpfx == None`	`return o`
`ExactSeg(o)`		`return o(segpfx)`
`ImmEnc(Immediate(Id))`		`return Id(Dword())`
`ImmEnc(Immediate(Iw))`		`return Iw(Word())`
`ImmEnc(Immediate(Ib))`		`return Ib(Byte())`
`ImmEnc(FarTarget(_))`	`addrpfx`	`return AP16(Word(),Word())`
`ImmEnc(FarTarget(_))`		`return AP32(Word(),Dword())`
`SignedImm(Id(_))`		`return Id(sign_extend(Byte()))`
`SignedImm(Iw(_))`		`return Iw(sign_extend(Byte()))`
`SizePrefix(y,n)`	`sizepfx`	`return decode(y)`
`SizePrefix(y,n)`		`return decode(n)`
`AddrPrefix(y,n)`	`addrpfx`	`return decode(y)`
`AddrPrefix(y,n)`		`return decode(n)`

`decode(a)`, Part 1

Decoding X86 Operands, by Variety

a	Condition	Return Value
`ImmEnc(MemExpr(m))`	`addrpfx`	`return Mem16(segpfx,m.Size,None,None,Word())`
`ImmEnc(MemExpr(m))`		`return Mem32(segpfx,m.Size,None,None,0,Dword())`
`GPart(g)`		`return g(ModRM.GGG)`
`RegOrMem(r,m)`	`ModRM.MOD == 3`	`return r(ModRM.RM)`
`RegOrMem(r,m)`	`addrpfx`	`return DecodeModRM16(m)`
`RegOrMem(r,m)`		`return DecodeModRM32(m)`

`ImmEnc(JccTarget(_))`		`displ = Dword()`
		`nextaddr = stream.ea()`
		`return JccTarget(nextaddr+displ,nextaddr)`

`SignedImm(JccTarget(_))`		`displ = sign_extend(Byte())`
		`nextaddr = stream.ea()`
		`return JccTarget(nextaddr+displ,nextaddr)`

`decode(a)`, Part 1

Decoding X86 Instructions

Create Flow Information

Mnemonic	Operand	Flow Type
`call`	`JccTarget(t,f)`	`FlowCallDirect(t,f)`
`call`	`GeneralReg(_)`	`FlowCallIndirect(retaddr)`
`call`	`FarTarget(_,_)`	`FlowCallIndirect(retaddr)`
`jmp`	`JccTarget(t,_)`	`FlowJmpUnconditional(t)`
`jmp`	`GeneralReg(_)`	`FlowJmpIndirect`
`jmp`	`FarTarget(_,_)`	`FlowJmpIndirect`
`jo`, `jno`, `jb`, `jae`, `jz`, `jnz`, `jbe`, `ja`, `js`, `jns`, `jp`, `jnp`, `jl`, `jge`, `jle`, `jg`, `loopnz`, `loopz`, `loop`, `jcxz`, `jecxz`	`JccTarget(t,f)`	`FlowJmpConditional(t,f)`
`ret`, `retf`, `iretd`, `iretw`	N/A	`FlowReturn`
Anything Else	N/A	`FlowOrdinary(nextaddr)`

- After decoding, determine the instruction's control flow behaviors from its mnemonic and operands.