CS 50011: Introduction to Systems II

Lecture 4: Introduction to Assembly

Prof. Jeff Turkstra

Lecture 03

- History
- Background
- x86
  - Syntax
  - Operands
  - Addressing modes
  - Data types
  - Instructions

A software hierarchy

Java, Pascal, FORTRAN, etc.

Portable among different computers with some effort; some machine-dependent features are visible; performance

C

Assembly languages: a distinct one for each computer, hardware dependence, just a few abstractions

VAX	X86	MIPS	ARM	• • •

Machine languages: one per computer; NO abstractions

Variable length bit strings	Variable length bit strings	32- & 64-bit strings	32- & 64-bit strings	• • •

Assembly language

- All (somewhat) different
- Many assembly languages share the same fundamental structure
  - Why?
- Typical assembly language statement syntax and corresponding machine code in hex…

```
        label: op result, operand1, operand2
0x004005F9 0x23CC803C
```

- Label is symbolic (an abstraction) for a memory address
- "op" is a mnemonic for the operation

Assembly is two-pass

- Initial pass of assembler resolves memory addresses for all labels
  - Even (especially) forward references
  - Symbol table
- Second pass emits machine code bitstrings
  - Translates mnemonics, register names, etc
  - Uses symbol table to fill in offset bit field
    - Offset = branch_target - current_addr

Why?

- Many languages are one-pass
  - C, for example
    - Have to prototype functions, declare/define
- Would have to manually determine instruction addresses and branch targets
- Changing the code often changes all of the offsets and addresses
- Impractical

Opcodes

- Set of opcode-field bit strings defines what the processor circuit can do
- Different processors have different sets of opcodes
- Assembly language defines a memorable symbolic name of a few characters for each opcode, a mnemonic
- No agreement on opcode mnemonics across assembly languages

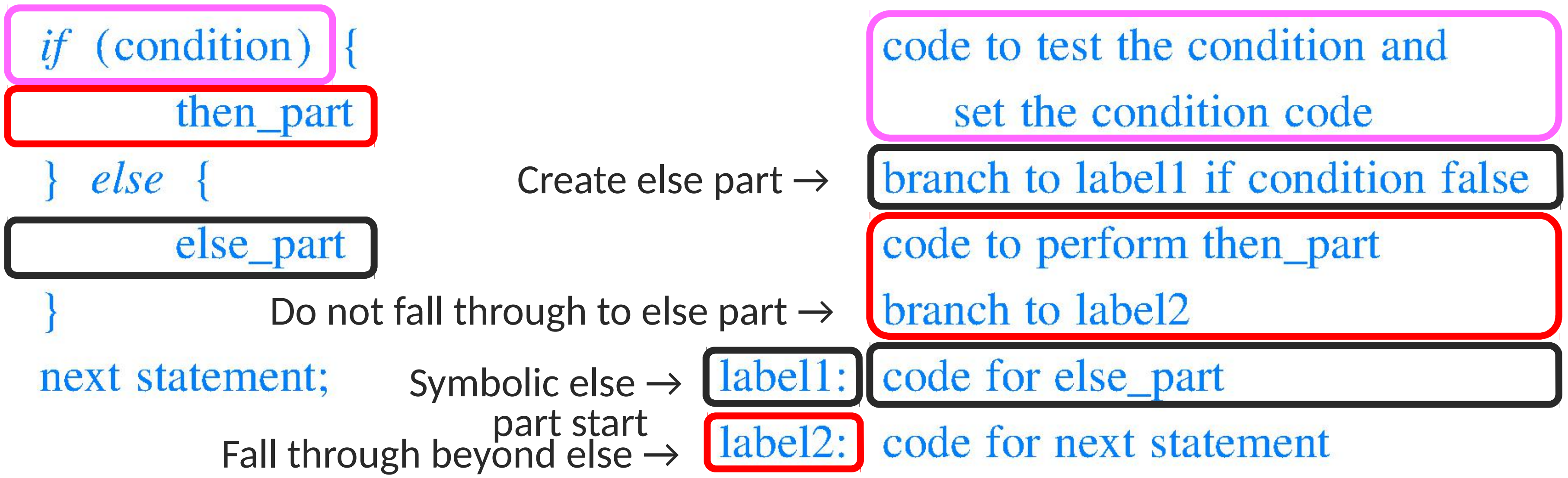
Readability

- Assembly is easy to write but hard to follow
- Comments are essential
  - Block comment – explain the purpose of a section of code, detail the use of registers and memory
  - Line comment – explains each instruction
- Comment usually starts with a delimiter, runs to end of line
- Best strategy: comment every line

Example

```
#####################################################
# Search linked list of free memory blocks to find  #
# a block of size N bytes or greater. Pointer must  #
# be in r3 and N in r4. Code destroys contents of   #
# r5, which is used to walk the list.               #
#####################################################
        ld   r5,r3    # load address of list into r5
loop_1: cmp  r5,0     # test to see if at list end
        bz   notfnd   # if reached end go to notfnd
```

Coding IF-THEN-ELSE in assembly



```
if (condition) {
        then_part
} else {
        else_part
}
next statement;
```

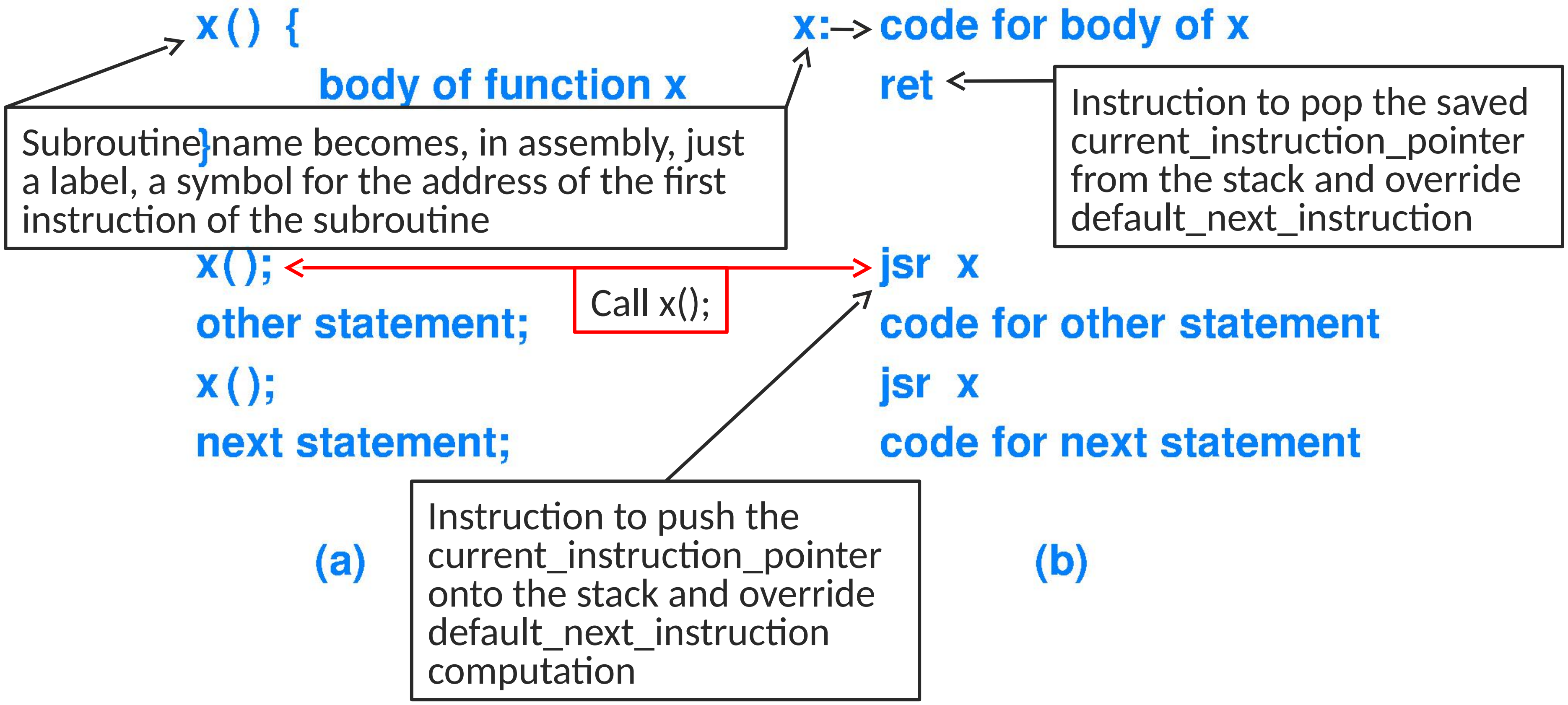
```
        code to test the condition and
            set the condition code
        branch to label1 if condition false
        code to perform then_part
        branch to label2
label1: code for else_part
label2: code for next statement
```

Create else part →
Do not fall through to else part →
Symbolic else → part start
Fall through beyond else →

"Fall through" means to fetch at the default next instruction location; must code two exceptions for if-then-else

Figure 9.2 (a) An *if-then-else* statement used in a high-level language, and (b) the equivalent assembly language code.

Subroutine call in assembly



Subroutine name becomes, in assembly, just a label, a symbol for the address of the first instruction of the subroutine

Instruction to pop the saved current_instruction_pointer from the stack and override default_next_instruction

Instruction to push the current_instruction_pointer onto the stack and override default_next_instruction computation

Figure 9.5 (a) A declaration for procedure *x* and two invocations in a high-level language, and (b) the assembly language equivalent.

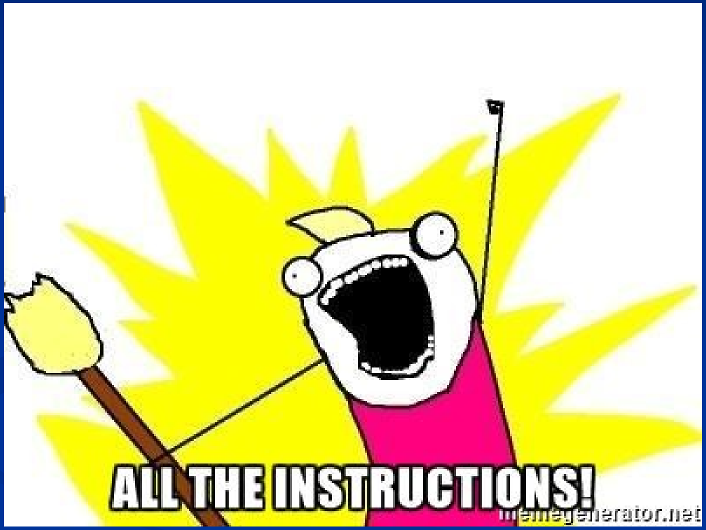
Language specifics

- Documentation
  - Operand order
  - Register naming
  - Syntax
    - Immediate values, register values, memory, etc
- Assembly language does not provide any program control structures, nor enforce any coding style

Intel documentation

- Volume 1: Basic Architecture
  - 482 pages
  - 19 Chapters
  - Includes basic execution environment as well as summary of instructions
    - Groups instructions for programming
      - MMX, SIMD, SSE, etc

- Volume 2: Instruction Set Reference A-Z
  - 2234 pages
  - "Only" 6 chapters
  - Instruction format
  - All of the instructions
  - Safer Mode Extensions



- Volume 3: System Programming Guide
  - 1660 pages
  - 43 Chapters
  - Everything the hardware does to support an OS and how to use it

CPUs have errata

- Ever hear of the original Pentium floating point bug?
  - Could have been errata, but the press picked it up
- Ever find a compiler error?
- Imagine finding a hardware error
  - Probably involves premature baldness
    - Possibly temporary

x86 Assembly

- Unfortunately, x86 is arguably the most complex assembly language around
  - MOV is even Turing complete
- Exposure to most common instructions
  - Focus on ability to read assembled C programs
  - Maybe a little writing
- Differences with x86_64

The Intel Legacy

- Started with 4004
  - 4-bit processor
- 8086, first x86 CPU
  - 16-bits
  - June 8, 1978
  - 5MHz, 8MHz, and 10MHz
- 80186, 80286
- 80386 (SX/DX), 80486 (SX/DX/DX2/etc)

Pentium

- MMX
- SSE, SSE2, SSE3
- X86-64
- AMD-V
- Intel VT-x
- etc
- …and it’s all backwards compatible

Fortunately

- Some analyses claim only 14 instructions account for 90% of compiled code

Assembly is symbolic

- label: mnemonic arg1, arg2, arg3
  - Zero to three args
  - Right is source, left is destination
- Mnemonic may represent different (multiple) opcodes

Remember

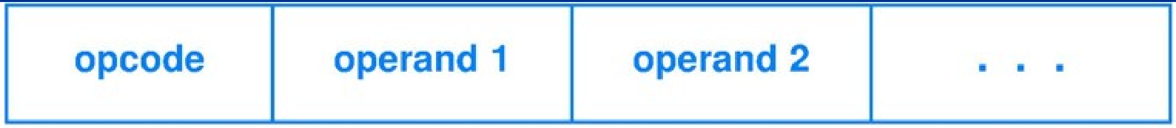
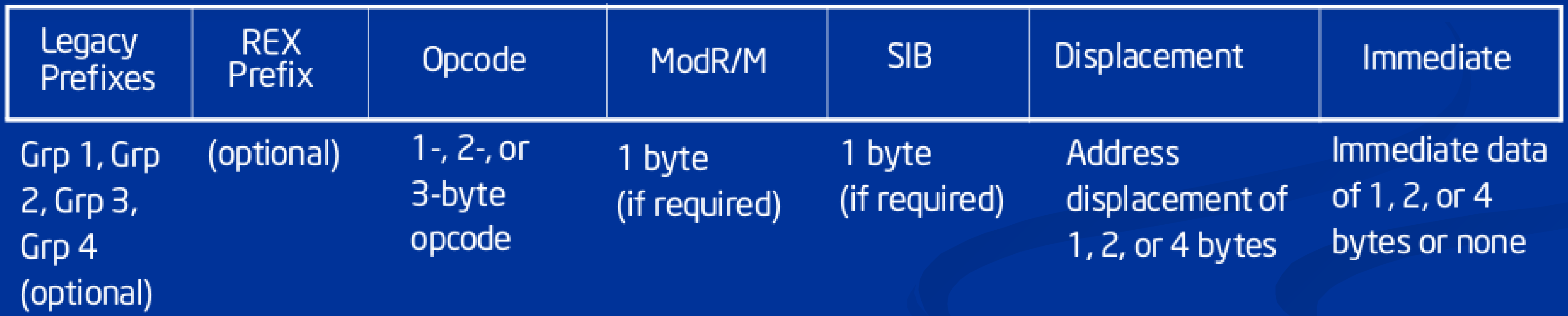


Figure 5.1 The general instruction format that many processors use. The opcode at the beginning of an instruction determines exactly which operands follow.

64-bit prefix ordering

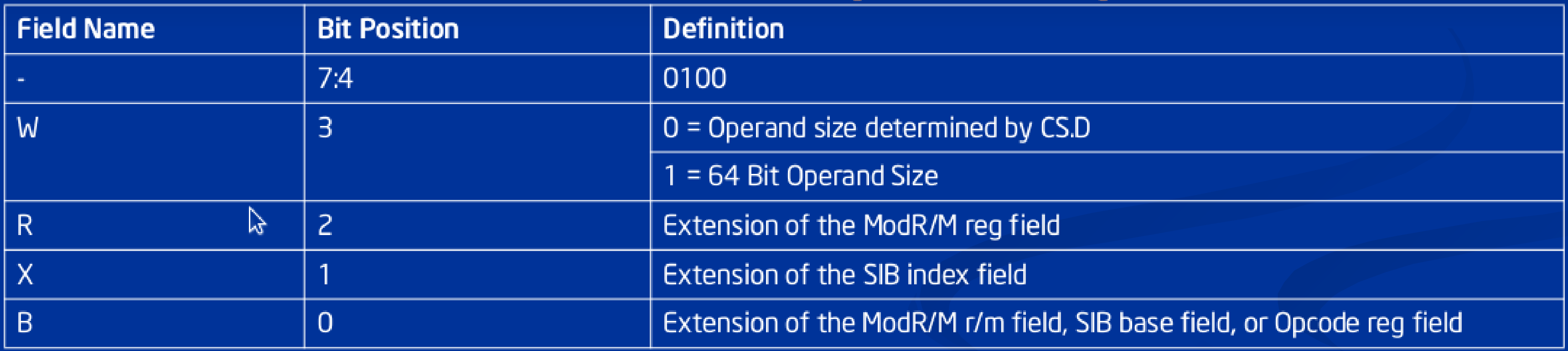


Legacy Prefixes	REX Prefix	Opcode	ModR/M	SIB	Displacement	Immediate
Grp 1, Grp 2, Grp 3, Grp 4 (optional)	(optional)	1-, 2-, or 3-byte opcode	1 byte (if required)	1 byte (if required)	Address displacement of 1, 2, or 4 bytes	Immediate data of 1, 2, or 4 bytes or none

```
mov    rcx,0x4004e0
48 c7 c1 e0 04 40 00
```

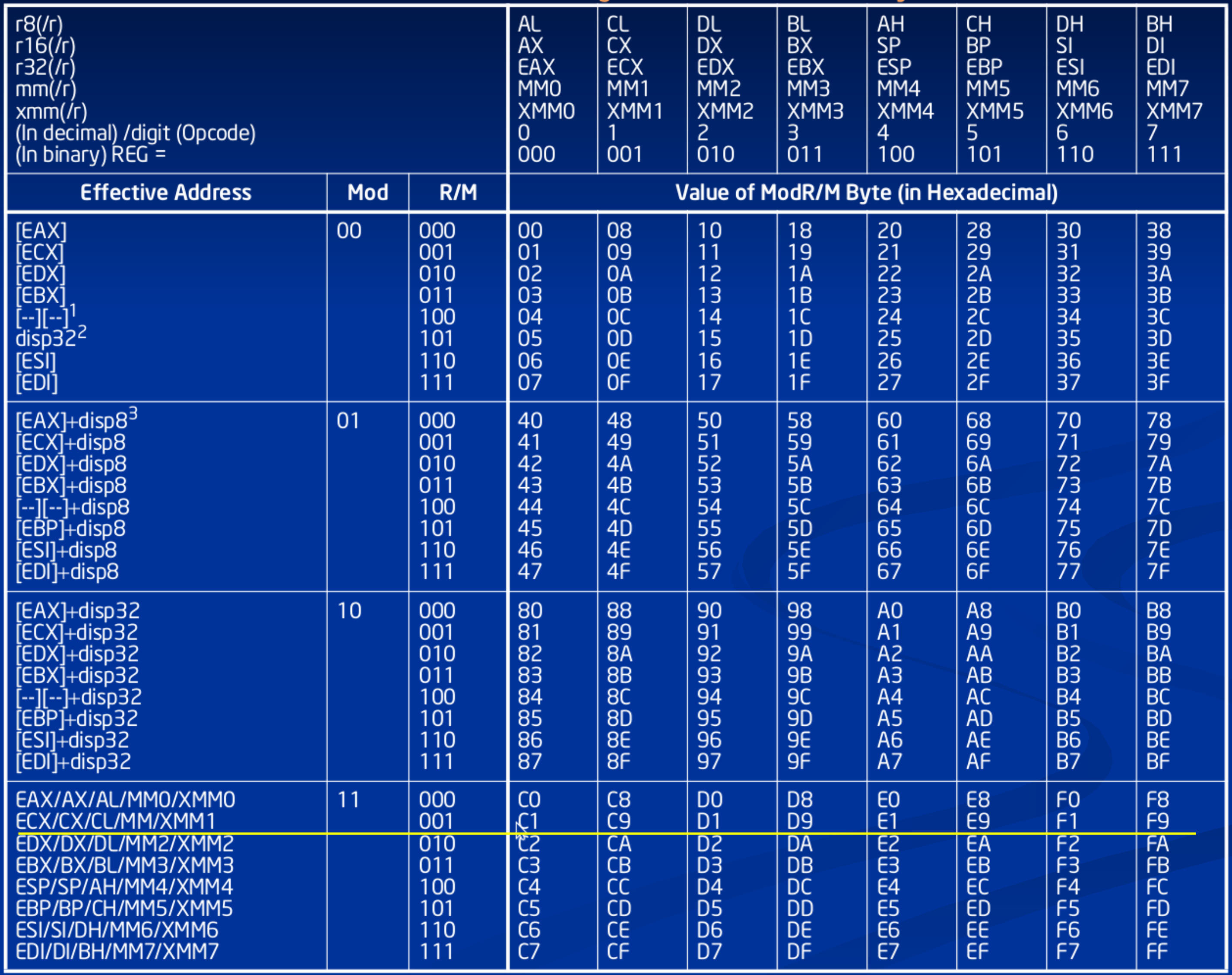
48: REX.W prefix: 64-bit operand
c7: MOV
c1: ecx (but really rcx)
e0044000: 004004e0

REX prefix



Field Name	Bit Position	Definition
-	7:4	0100
W	3	0 = Operand size determined by CS.D
		1 = 64 Bit Operand Size
R	2	Extension of the ModR/M reg field
X	1	Extension of the SIB index field
B	0	Extension of the ModR/M r/m field, SIB base field, or Opcode reg field

Wat?



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

Syntax

- Intel
  - [base + index*scale + disp]

```
call DWORD PTR [rbx+rsi*4-0xe8]
mov rax, DWORD PTR [rbp+0x8]
lea rax, [rbx-0xe8]
```

- AT&T
  - disp(base, index, scale)

```
call *-0xe8(%rbx,%rsi,4)
mov 0x8(%rbp), %rax
lea -0xe8(%rbx), %rax
```

Intel vs. AT&T syntax

- Intel
  - Destination comes first

```
mov rbp, rsp
add rax, 0x14
```

- AT&T
  - Reverse

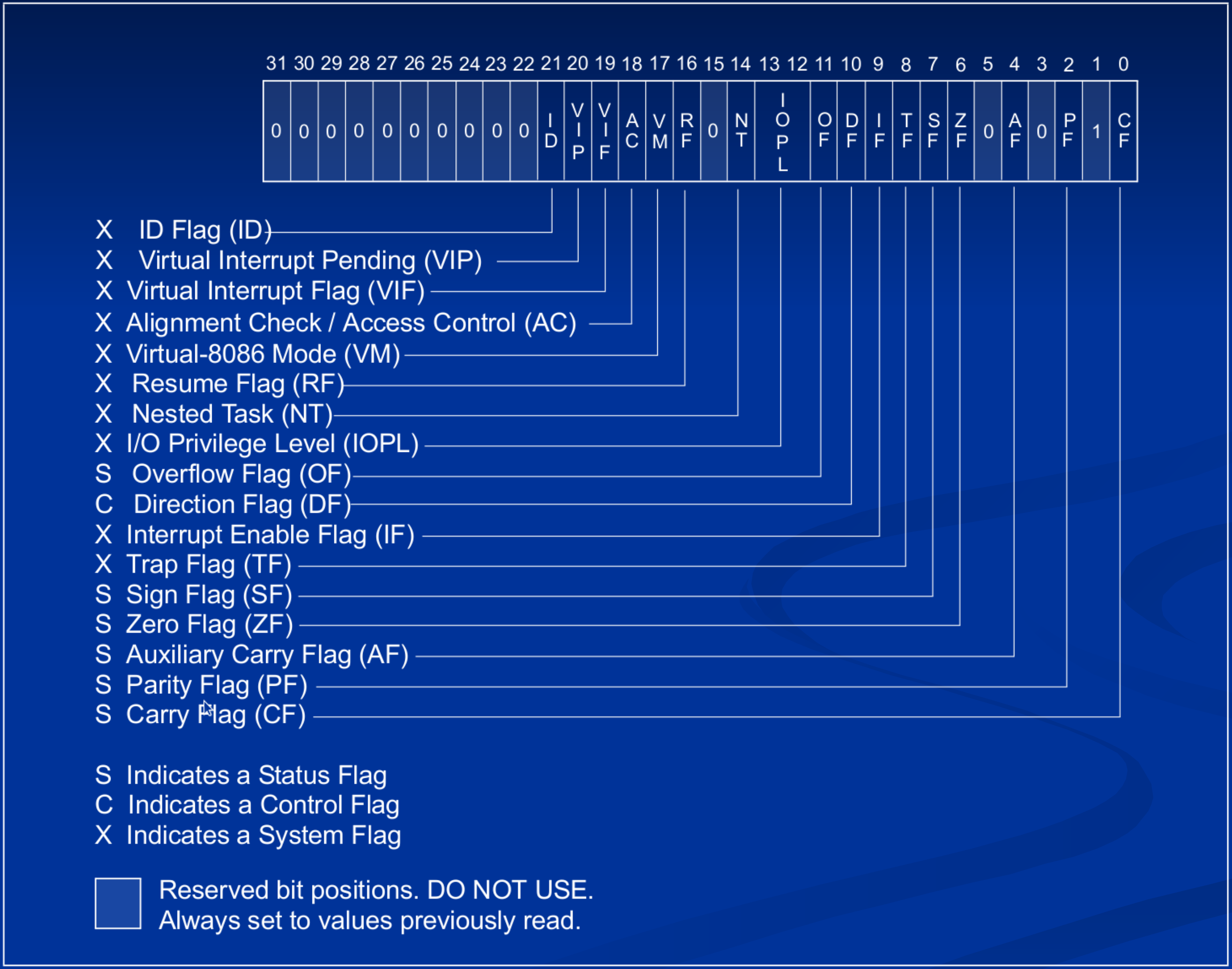
```
mov %rsp, %rbp
add $0x14, %rsp
```

  - Registers prefixed with %, immediate $

Registers

- EIP/RIP
- (E|R)[ABCD]X
  - A: Accumulator
  - B: Base
  - C: Counter
  - D: Data
- ESI, EDI: source and destination pointers for string operations
  - Based off DS in compatibility mode
- ESP, EBP
  - SS segment

EFLAGS/RFLAGS



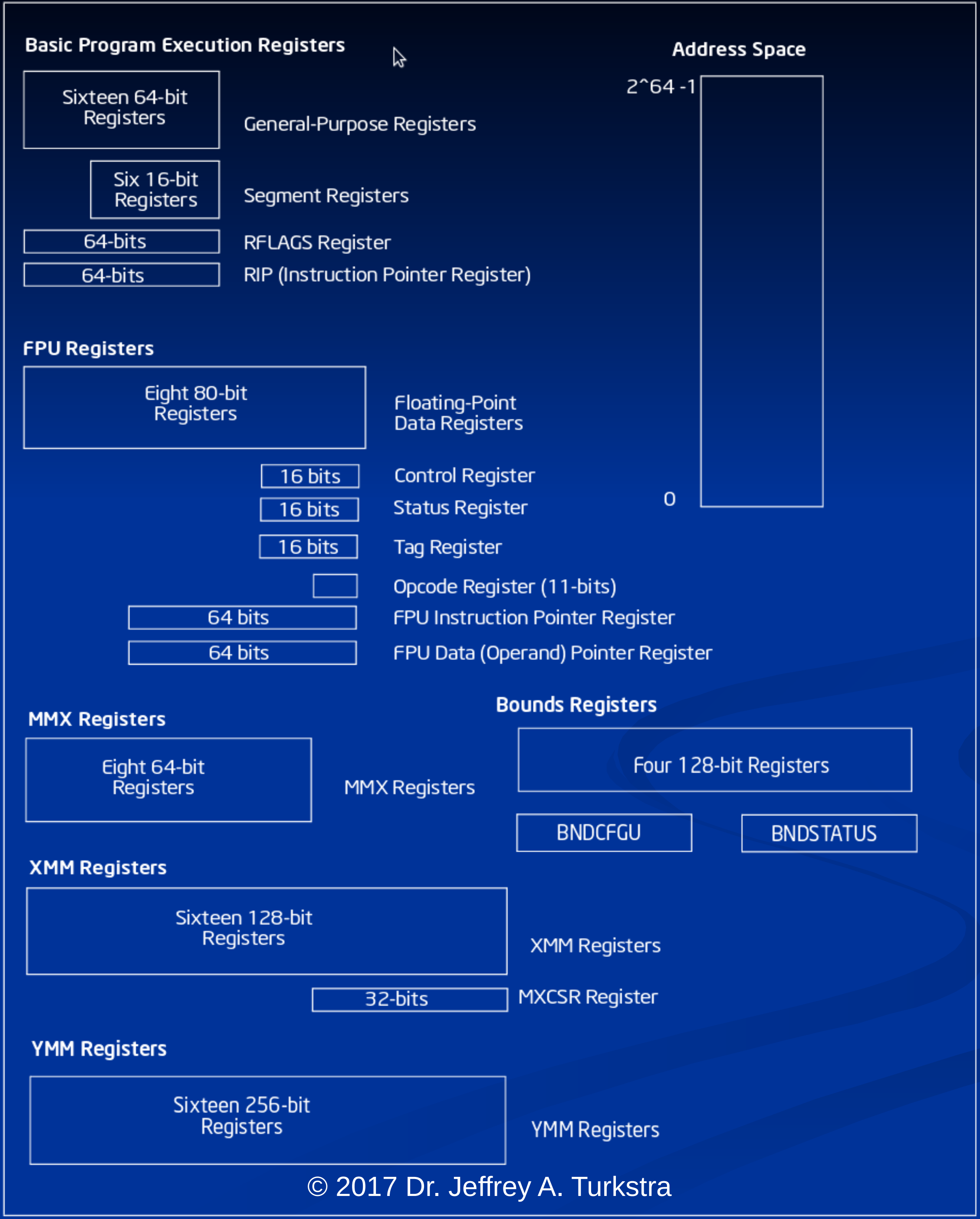
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	ID	VIP	VIF	AC	VM	RF	0	NT	IOPL		OF	DF	IF	TF	SF	ZF	0	AF	0	PF	1	CF

X ID Flag (ID)
X Virtual Interrupt Pending (VIP)
X Virtual Interrupt Flag (VIF)
X Alignment Check / Access Control (AC)
X Virtual-8086 Mode (VM)
X Resume Flag (RF)
X Nested Task (NT)
X I/O Privilege Level (IOPL)
S Overflow Flag (OF)
C Direction Flag (DF)
X Interrupt Enable Flag (IF)
X Trap Flag (TF)
S Sign Flag (SF)
S Zero Flag (ZF)
S Auxiliary Carry Flag (AF)
S Parity Flag (PF)
S Carry Flag (CF)

S Indicates a Status Flag
C Indicates a Control Flag
X Indicates a System Flag

▨ Reserved bit positions. DO NOT USE.
Always set to values previously read.

Figure 3-8. EFLAGS Register


Basic Program Execution Registers
Sixteen 64-bit Registers
General-Purpose Registers
Six 16-bit Registers
Segment Registers
64-bits
RFLAGS Register
64-bits
RIP (Instruction Pointer Register)
Address Space
2^64 -1
0
FPU Registers
Eight 80-bit Registers
Floating-Point Data Registers
16 bits
Control Register
16 bits
Status Register
16 bits
Tag Register
Opcode Register (11-bits)
64 bits
FPU Instruction Pointer Register
64 bits
FPU Data (Operand) Pointer Register
MMX Registers
Eight 64-bit Registers
MMX Registers
Bounds Registers
Four 128-bit Registers
BNDCFGU
BNDSTATUS
XMM Registers
Sixteen 128-bit Registers
XMM Registers
32-bits
MXCSR Register
YMM Registers
Sixteen 256-bit Registers
YMM Registers

Operand Addressing

- Data for a source operand can be found in...
  - The instruction itself (immediate)
  - A register
  - A memory location
  - An I/O port
- A destination operand can be:
  - A register
  - A memory location
  - An I/O port

Immediate operands

- Example: ADD EAX, 14

- All arithmetic instructions permit an immediate source operand.
- Max value varies, never larger than an unsigned doubleword integer (2^{32})

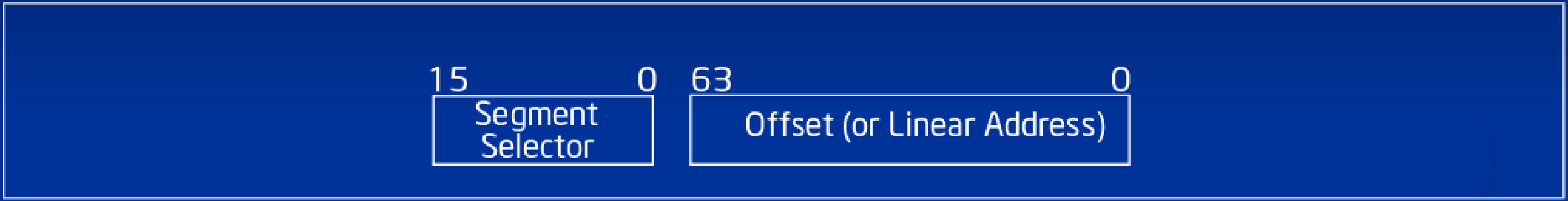
Register operands

- 64-bit general-purpose registers:
  - RAX, RBX, RCX, RDX, RSI, RDI, RSP, RBP, R8-R15
- 32-bit general-purpose registers:
  - EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP, R8D-R15D
- 16-bit general-purpose registers
- 8-bit general-purpose registers
- Segment registers

- RFLAGS
- FPU registers
- MMX, XMM, Control, Debug, and MSR registers
- RDX:RAX register pair (128-bit operand)

Memory operands

- Segment selector and offset

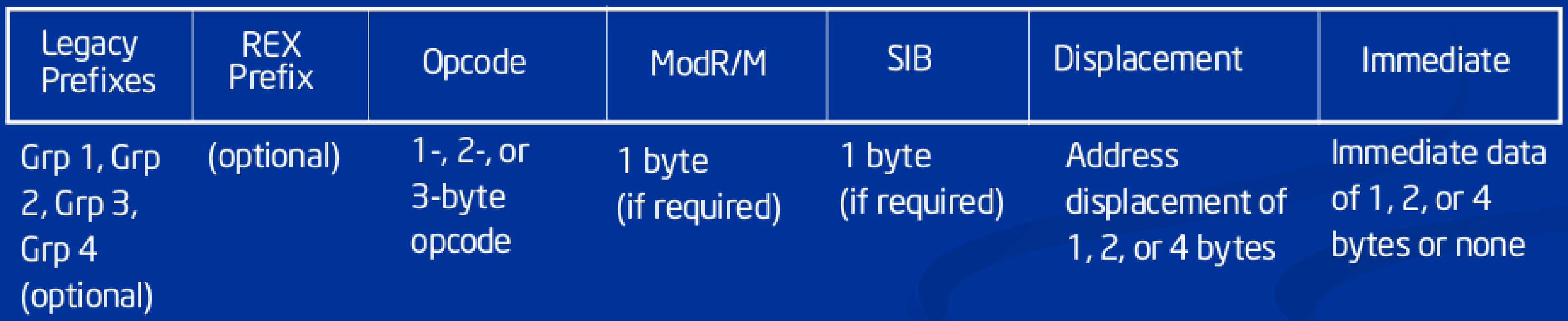


- 64-bit mode segmentation is generally disabled (flat 64-bit linear address space)
  - CS, DS, ES, SS are 0
  - FS and GS can be used as additional base registers

Memory offset

- Displacement: 8, 16, or 32-bits
    - Direct, static value
- Base and Index
    - Values from general-purpose registers
- Scale factor
    - 2, 4, or 8
    - Multiplies Index
- RIP + Displacement
- Result is called an effective address

64-bit prefix ordering

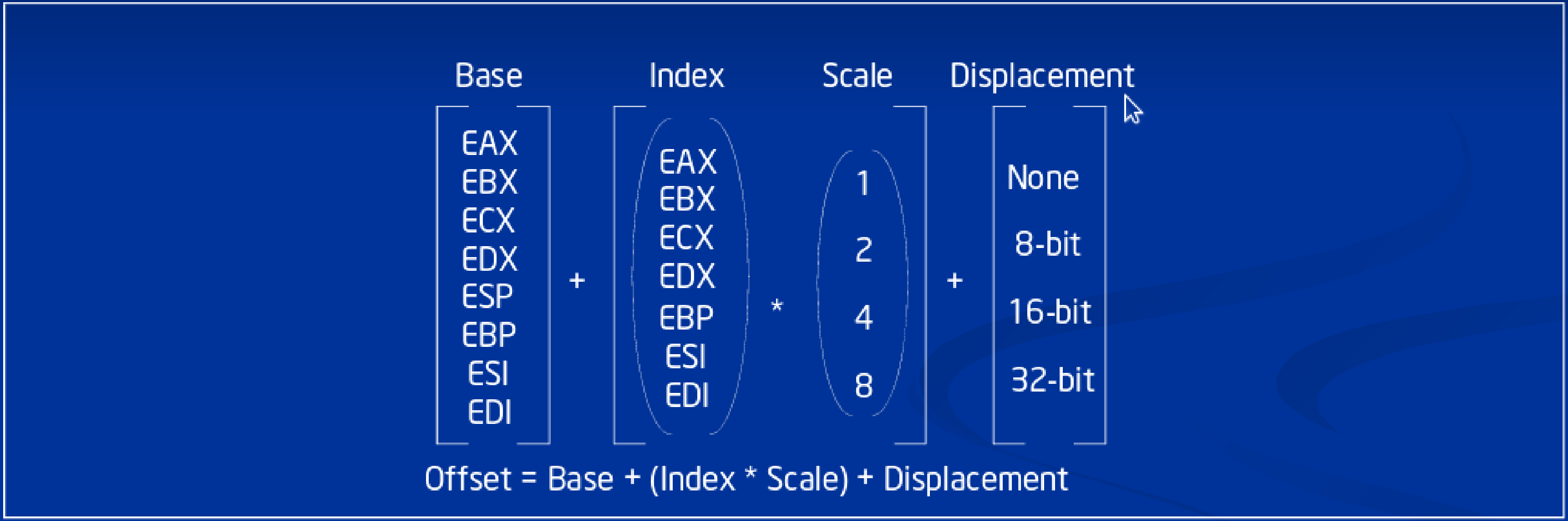


Legacy Prefixes	REX Prefix	Opcode	ModR/M	SIB	Displacement	Immediate
Grp 1, Grp 2, Grp 3, Grp 4 (optional)	(optional)	1-, 2-, or 3-byte opcode	1 byte (if required)	1 byte (if required)	Address displacement of 1, 2, or 4 bytes	Immediate data of 1, 2, or 4 bytes or none

SIB?

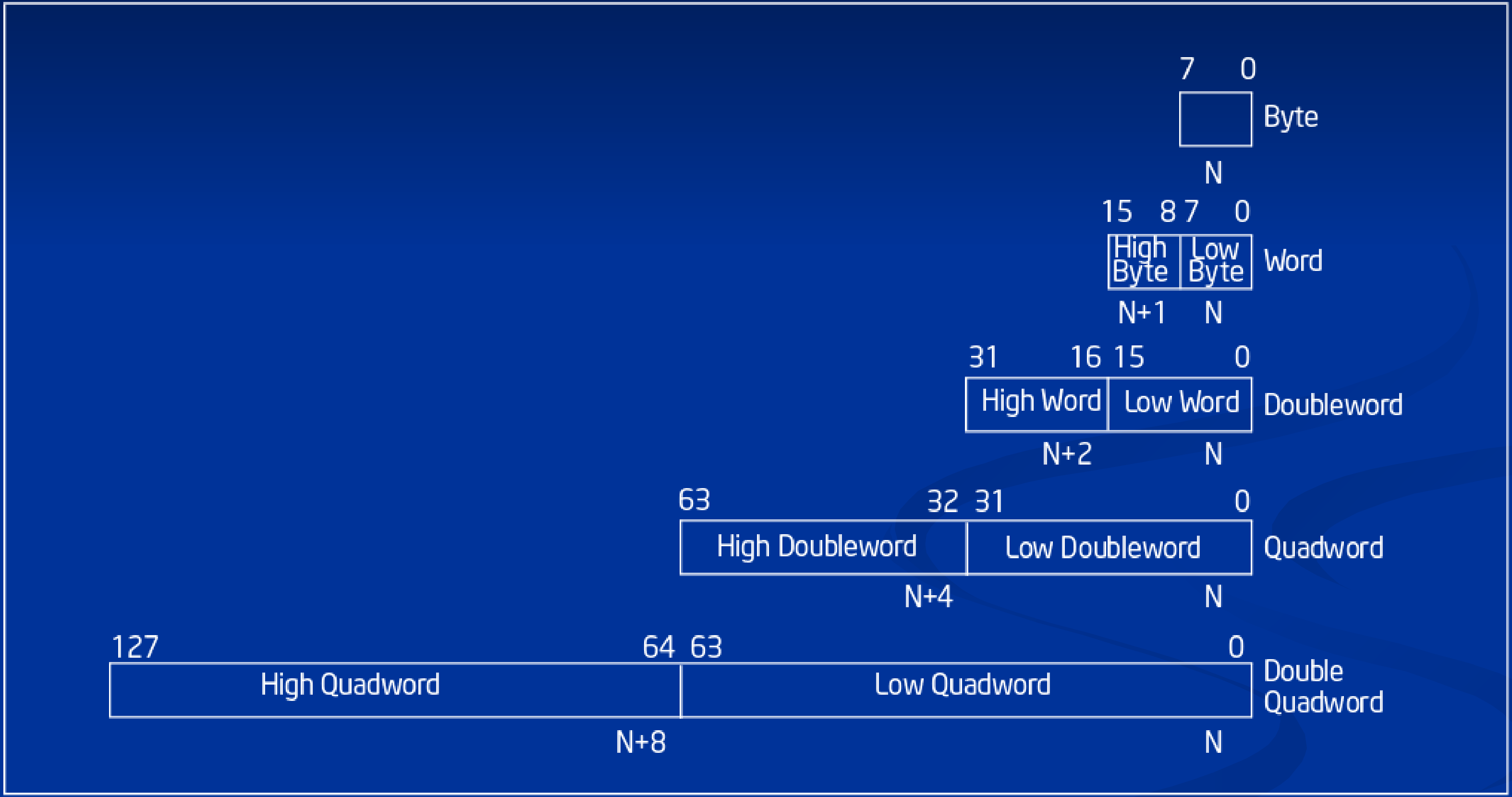
- Scale
- Index
- Base

Effective address computation



Offset = Base + (Index * Scale) + Displacement

Data types


7 0
Byte
N
15 8 7 0
High Byte
Low Byte
Word
N+1 N
31 16 15 0
High Word
Low Word
Doubleword
N+2 N
63 32 31 0
High Doubleword
Low Doubleword
Quadword
N+4 N
127 64 63 0
High Quadword
Low Quadword
Double Quadword
N+8 N

LEA

- LEA, the only instruction that performs memory addressing calculations but doesn't actually address memory. LEA accepts a standard memory addressing operand, but does nothing more than store the calculated memory offset in the specified register, which may be any general purpose register.
- What does that give us? Two things that ADD doesn't provide:
  - the ability to perform addition with either two or three operands, and
  - the ability to store the result in any register; not just one of the source operands.

What about 32-bits

- Many systems now are x86_64
- BUT, they can run a lot of 32-bit software
  - “Compatibility mode”
  - Segment registers actually matter
  - Relies on 32-bit registers/addresses/etc
- x86_64 CPUs can switch in and out of compatibility mode with ease
  - Consider system calls for a 64-bit kernel running a 32-bit program

Instruction set

- Data transfer instructions
- Binary arithmetic
- Decimal arithmetic
- Logical
- Shift and rotate
- Bit and byte
- Control
- String

- Flag control ([ER]FLAG)
- Segment registers
- Miscellaneous

Data transfer instructions

- Move data between memory and registers
    - Can be conditional
    - Includes stack access
- CMOV and friends
- XCHG
- BSWAP
- PUSH, PUSHA
- POP, POPA

MOV

- Register to register
- Memory to register
- Register to Memory
- Never memory to memory
  - Remember DMA?

Binary arithmetic instructions

- Basic binary integer computations
- ADD
- SUB
- IMUL, IDIV
- MUL, DIV
- INC, DEC, NEG
- CMP

Decimal arithmetic instructions

- Manipulate BCD data
- Invalid in 64-bit mode

Logical, shift and rotate instructions

- AND, OR, XOR, NOT
- SAR, SHR, SAL, SHL
- ROR, ROL, RCR, RCL

Bit and byte instructions

- BT
- BTS, BTR
  - Semaphores
- SETE, SETZ and friends
- TEST
- CRC32, POPCNT

Control transfer instructions

- JMP
- JE, JZ, JNE, JNZ
- CALL, RET
- INT, IRET
- ENTER, LEAVE

String instructions

- MOVS, MOVSB
  - B/W/D: byte, word, doubleword
- CMPS, CMPSB

Flag control instructions

- STC, CLC
- STD, CLD
- LAHF, SAHF
- PUSHF, PUSHFD
- POPF, POPFD
- STI
- CLI

Questions?