EE345

Instruction Encoding

Instructions

- Remember what an instruction conveys
  - Operation
  - Operands (source and destination = “src” and “dst”)
- Need to encode these into binary
- Architecture Defines
  - Instruction “formats”: patterns for encoding
  - Rules on how instructions can sit in memory
    - E.g., instruction address must be 4-byte aligned
    - E.g., instructions are little-endian
  - How many bytes (length) instructions can use

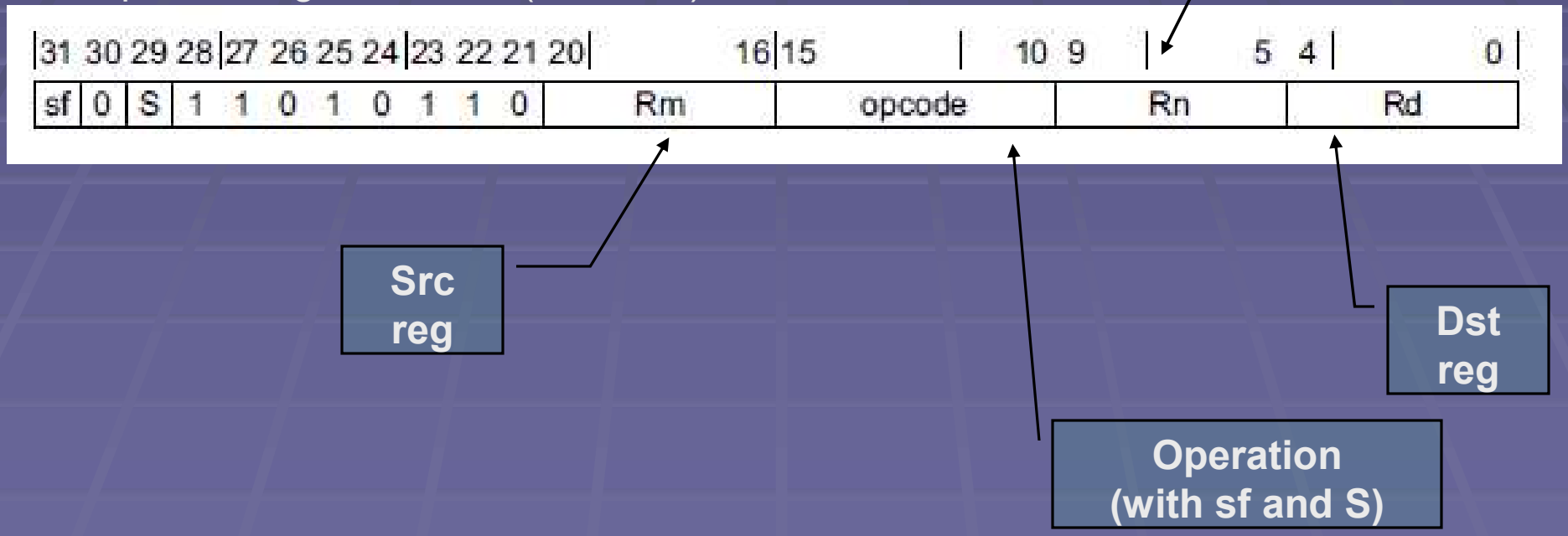
Instruction Encoding Length

- Fixed Length
  - All instructions are the same number of bytes
    - Typically two or four bytes, depending on ISA
  - CPU designers like these the best ☺
  - We're only going to deal with these for EE345 hardware
- Small number of lengths
  - E.g., 4B or 8B (Intel i960)
  - E.g., 2B or 4B (ARM Thumb)
- Variable length
  - E.g., 1B…15B (X86)
  - E.g., 68020 had a 22B (!) instruction
  - Hardest to deal with

Instruction Formats

- ISA defines patterns for encoding
  - positions for items in instructions
- May have a few formats ☺, may have many ☹

From ARM Architecture Reference Manual: ARMv8
Data-processing instruction (2 source)



Number of Operands

- 1-operand
  - Implied (single) register = accumulator
  - E.g., `ADD B ; A = A + B`
- 2-operand
  - One src, other operand is both src and dst
  - E.g., `ADD A, B ; A = A op B`
- 3-operand
  - Two src, one dst
  - E.g., `ADD A, B, C ; A = B + C`

Common Instruction Components

- Often called a “field” in the encoding
- Register ID’s
- Immediate values
- Branch targets
- Data-address specifiers

Register ID's

- Typically an N-bit group of bits
  - Denoting one of 2^N registers
- E.g., 5b field to indicate one of 32 registers
- Instructions can have multiple register ID's
  - Destination, source1, etc.
- Tradeoff on number of registers
  - More registers = wider reg ID's, instruction encoding becomes congested
  - Fewer registers = slower execution

Immediate Values

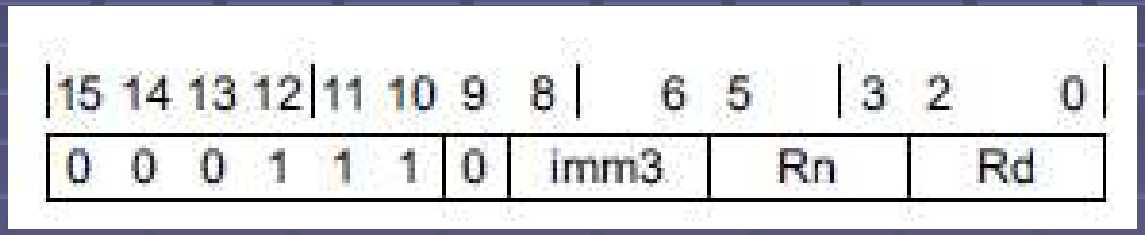
- Integer encoded directly into an instruction
- Used for things like…

```
ADD EAX, 2  ; X86: add 2 to reg
```

- Fixed-width instruction means…
  - Can’t fit wide immediate values
  - Use a narrow value (like 16b) and sign-extend
- Variable width…
  - Immediates can be wide (32b, 64b, whatever)
  - Can fit more than one immediate in instruction

Register ID and Immediate in ARM

- ADDS: ARM “Thumb” ISA (16b instructions)




```
0001c  1c9b adds        r3,r3,#2  ; r3 = r3 + 2
```

1C9B = 0 0 0 1 1 1 0 0 1 0 0 1 1 0 1 1

operation | #2 | R3 (src) | R3 (dst)

Branch Targets

- When code wants to “branch”
  - Execute at another place in memory
  - Need to specify where that place *is*
  - That’s a branch target – it’s an address
- Absolute address
  - Encode the address into the instruction
  - Or some bits of address (no room for all!)
- Relative address
  - Encode an offset in the instruction
  - Target is related to current address + offset

Relative Branch Target

- You'll also hear it called "PC-relative"
  - PC is Program Counter = address of next instruction
  - X86 calls it the IP: instruction pointer
- Rules to compute target are architecture-specific
  - E.g., (PC + offset + 4) or similar is common
  - Offset usually sign-extended to allow backward branch

Addr	Instr	Note
0004	br +3	PC-relative branch
0005	add	Jump over this
0006	sub	Jump over this
0007	xor	Target of br

Relative branch in X86

JMP—Jump

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
EB *cb*	JMP *rel8*	D	Valid	Valid	Jump short, RIP = RIP + 8-bit displacement sign extended to 64-bits

A relative offset (*rel8, rel16*, or *rel32*) is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed 8-, 16-, or 32-bit immediate value. This value is added to the value in the EIP register. (Here, the EIP register contains the address of the instruction following the JMP instruction). When using relative offsets, the opcode (for short vs. near jumps) and the operand-size attribute (for near relative jumps) determines the size of the target operand (8, 16, or 32 bits).

- Branch Target is sum of
  - RIP (Instruction pointer in 64b mode, EIP in 32b)
  - Sign-extended rel8 (one-byte field in instruction)
  - 2 (length of the instruction, in bytes)

Example of X86 JMP

- Branching around a 7B instruction

```
00007FF6B7BB1730 EB 07                    jmp main+49h
00007FF6B7BB1732 C7 45 04 00 00 00 00 mov dword ptr [r],0
00007FF6B7BB1739 8B 45 04                 mov eax,dword ptr [r]
```

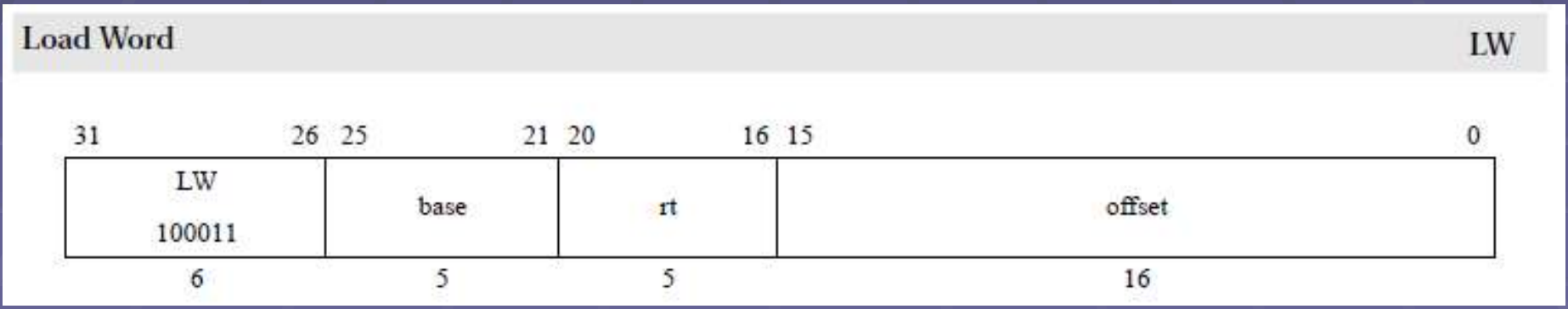
Branch at address (ending in)	730
Branch offset	07
Bytes in branch instruction	02
Target of branch	739

Data-Address Specifiers

- To access memory – need an address
- Address is formed from pieces
  - Registers
  - Offsets
  - Operations on them (like shifts and adds)
- Instruction encodes all these pieces
- Combine pieces according to architectural rules: the “effective address”

MIPS Effective Addresses

- Effective address =
  - Contents of register +
  - Sign-extended immediate
- We like how simple this is!



- Base: a register (0-31)
- Offset: 16b to add

X86 Effective Address

- Here's just one table from the spec

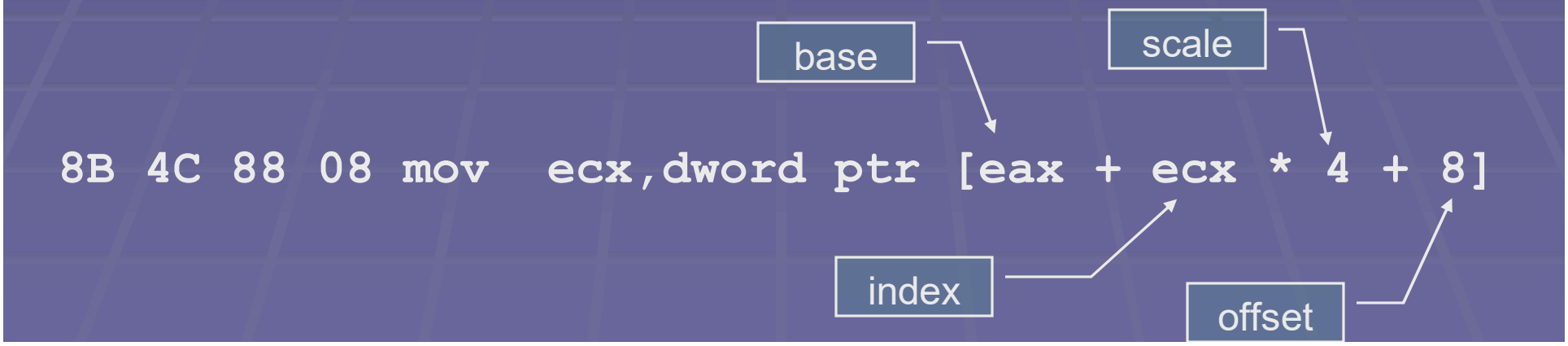
Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte

r8(/r) r16(/r) r32(/r) mm(/r) xmm(/r) (In decimal) /digit (Opcode) (In binary) REG =			AL AX EAX MM0 XMM0 0 000	CL CX ECX MM1 XMM1 1 001	DL DX EDX MM2 XMM2 2 010	BL BX EBX MM3 XMM3 3 011	AH SP ESP MM4 XMM4 4 100	CH BP EBP MM5 XMM5 5 101	DH SI ESI MM6 XMM6 6 110	BH DI EDI MM7 XMM7 7 111
Effective Address	**Mod**	**R/M**	**Value of ModR/M Byte (in Hexadecimal)**							
[EAX] [ECX] [EDX] [EBX] [--][--][1] disp32[2] [ESI] [EDI]	00	000 001 010 011 100 101 110 111	00 01 02 03 04 05 06 07	08 09 0A 0B 0C 0D 0E 0F	10 11 12 13 14 15 16 17	18 19 1A	20 21 22	28 29 2A	30 31 32	38 39 3A
[EAX]+disp8[3] [ECX]+disp8 [EDX]+disp8 [EBX]+disp8 [--][--]+disp8 [EBP]+disp8 [ESI]+disp8 [EDI]+disp8	01	000 001 010 011 100 101 110 111	40 41 42 43 44 45 46 47	48 49 4A 4B 4C 4D 4E 4F	50 51 52 53 54 55 56 57					
[EAX]+disp32 [ECX]+disp32 [EDX]+disp32 [EBX]+disp32 [--][--]+disp32 [EBP]+disp32 [ESI]+disp32 [EDI]+disp32	10	000 001 010 011 100 101 110 111	80 81 82 83 84 85 86 87	88 89 8A 8B 8C 8D 8E 8F	90 91 92 93 94 95 96 97	98	A0	A8	B0	B8

X86 Effective Address

- Can specify:
  - Base register
  - Index register
  - Scale (multiplier) for the index register
  - Offset (various sizes)

8B /r	MOV r32,r/m32	RM	Valid	Valid	Move r/m32 to r32.



Instruction Formats (MIPS)

- For integer instructions: just three formats
  - Fixed width = 32b
  - (from MIPS32 architecture reference)

Figure 4-1 Immediate (I-Type) CPU Instruction Format

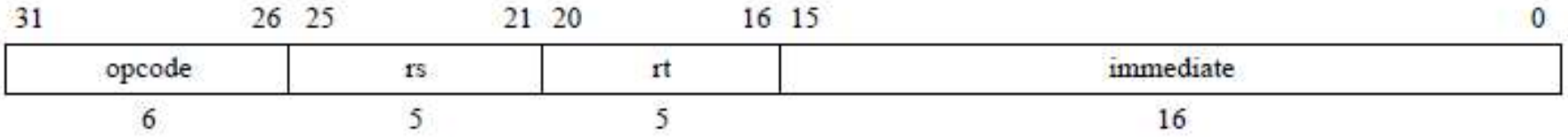


Figure 4-2 Jump (J-Type) CPU Instruction Format

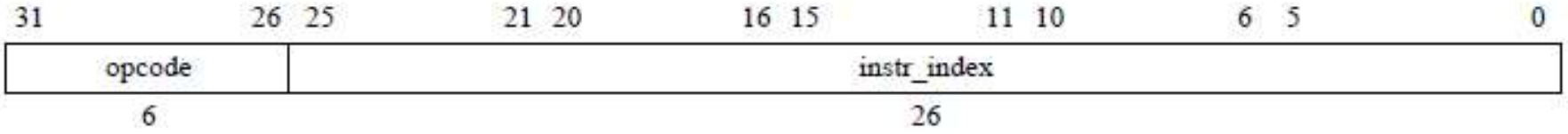
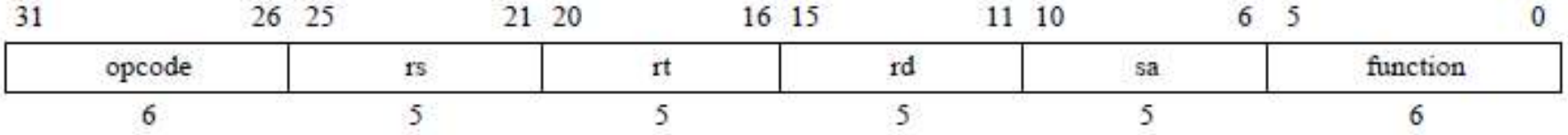


Figure 4-3 Register (R-Type) CPU Instruction Format



Instruction Formats (X86)

- Variable width
- Much more complicated to decode
- (from Intel 64 and IA-32 Architecture SW Developer's Manual)

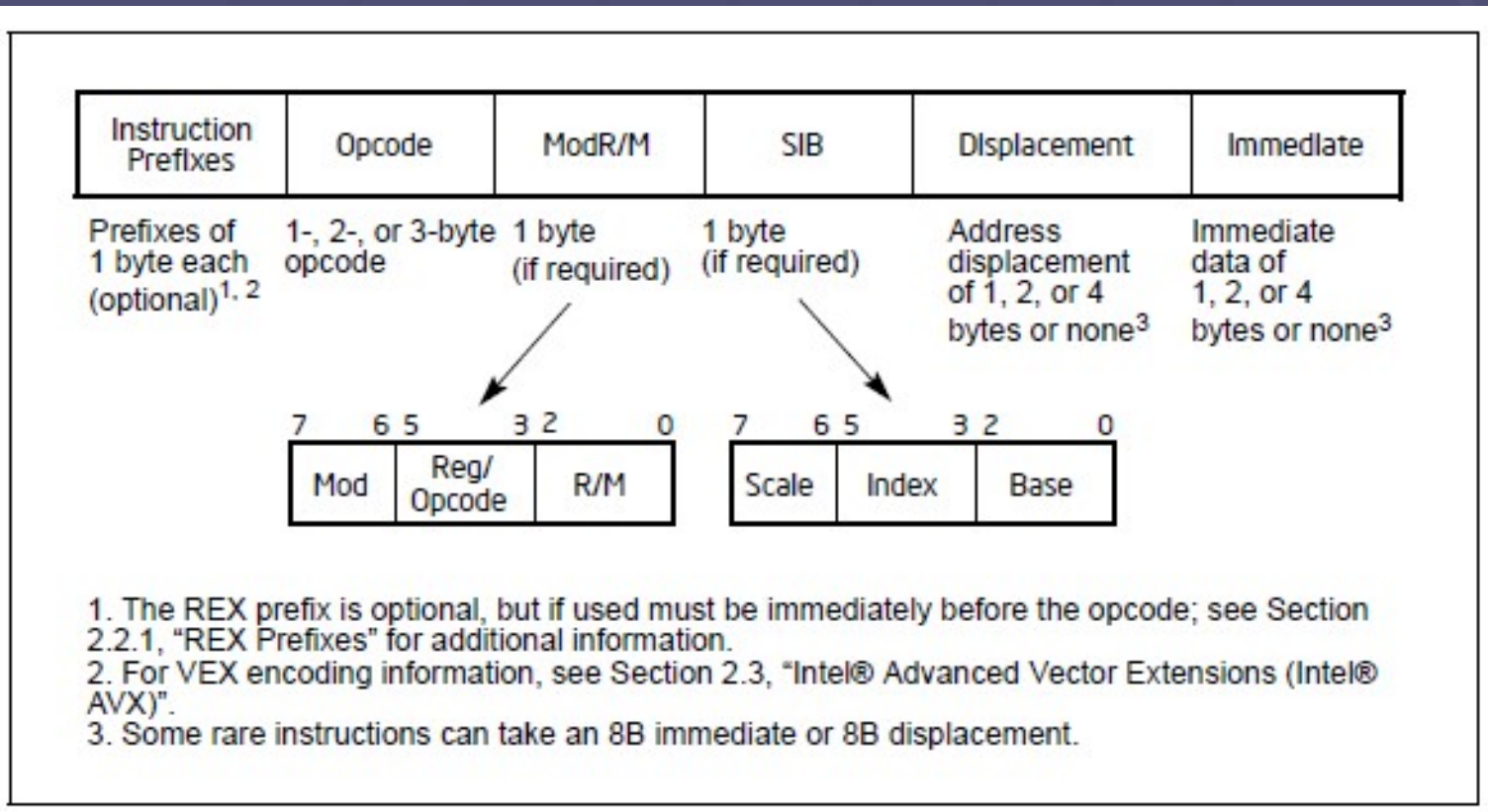


Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format