Lecture 4: ISA Tradeoffs (Continued) and Single-Cycle Microarchitectures

ISA-level Tradeoffs: Instruction Length

- Fixed length: Length of all instructions the same
  - + Easier to decode single instruction in hardware
  - + Easier to decode multiple instructions concurrently
  - -- Wasted bits in instructions (Why is this bad?)
  - -- Harder-to-extend ISA (how to add new instructions?)
- Variable length: Length of instructions different (determined by opcode and sub-opcode)
  - + Compact encoding (Why is this good?)
    - Intel 432: Huffman encoding (sort of).
  - -- More logic to decode a single instruction
  - -- Harder to decode multiple instructions concurrently
- Tradeoffs
  - ❑ Code size (memory space, bandwidth, latency) vs. hardware complexity
  - ❑ ISA extensibility and expressiveness vs. hardware complexity
  - ❑ Performance? Smaller code vs. ease of decode. Energy?

ISA-level Tradeoffs: Uniform Decode

- Uniform decode: Same bits in each instruction correspond to the same meaning
  - Opcode is always in the same location
  - Ditto operand specifiers, immediate values, ...
  - Many “RISC” ISAs: Alpha, MIPS, SPARC

  + Easier decode, simpler hardware

  + Enables parallelism: generate target address before knowing the instruction is a branch

  -- Restricts instruction format (fewer instructions?) or wastes space

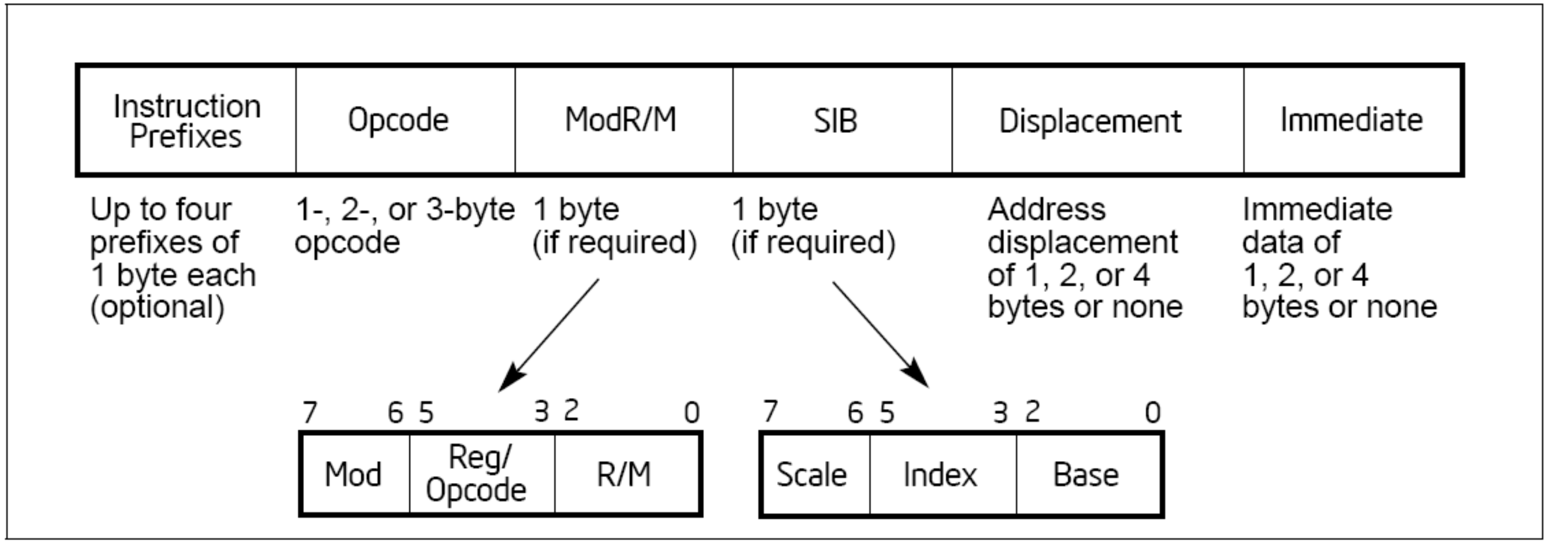
- Non-uniform decode
  - E.g., opcode can be the 1st-7th byte in x86

  + More compact and powerful instruction format

  -- More complex decode logic

x86 vs. Alpha Instruction Formats

- x86:



Instruction Prefixes | Opcode | ModR/M | SIB | Displacement | Immediate

Instruction Prefixes: Up to four prefixes of 1 byte each (optional)

Opcode: 1-, 2-, or 3-byte opcode

ModR/M: 1 byte (if required)

SIB: 1 byte (if required)

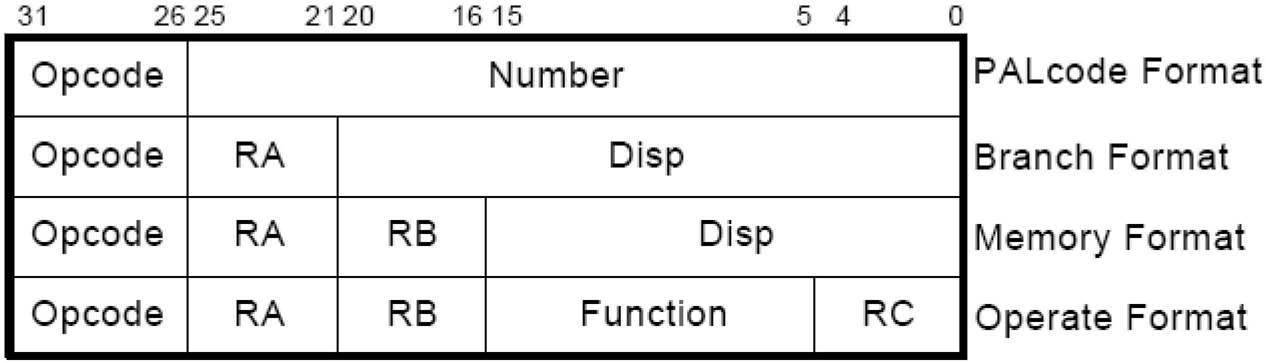
Displacement: Address displacement of 1, 2, or 4 bytes or none

Immediate: Immediate data of 1, 2, or 4 bytes or none

ModR/M: Mod (bits 7–6) | Reg/Opcode (bits 5–3) | R/M (bits 2–0)

SIB: Scale (bits 7–6) | Index (bits 5–3) | Base (bits 2–0)

- Alpha:



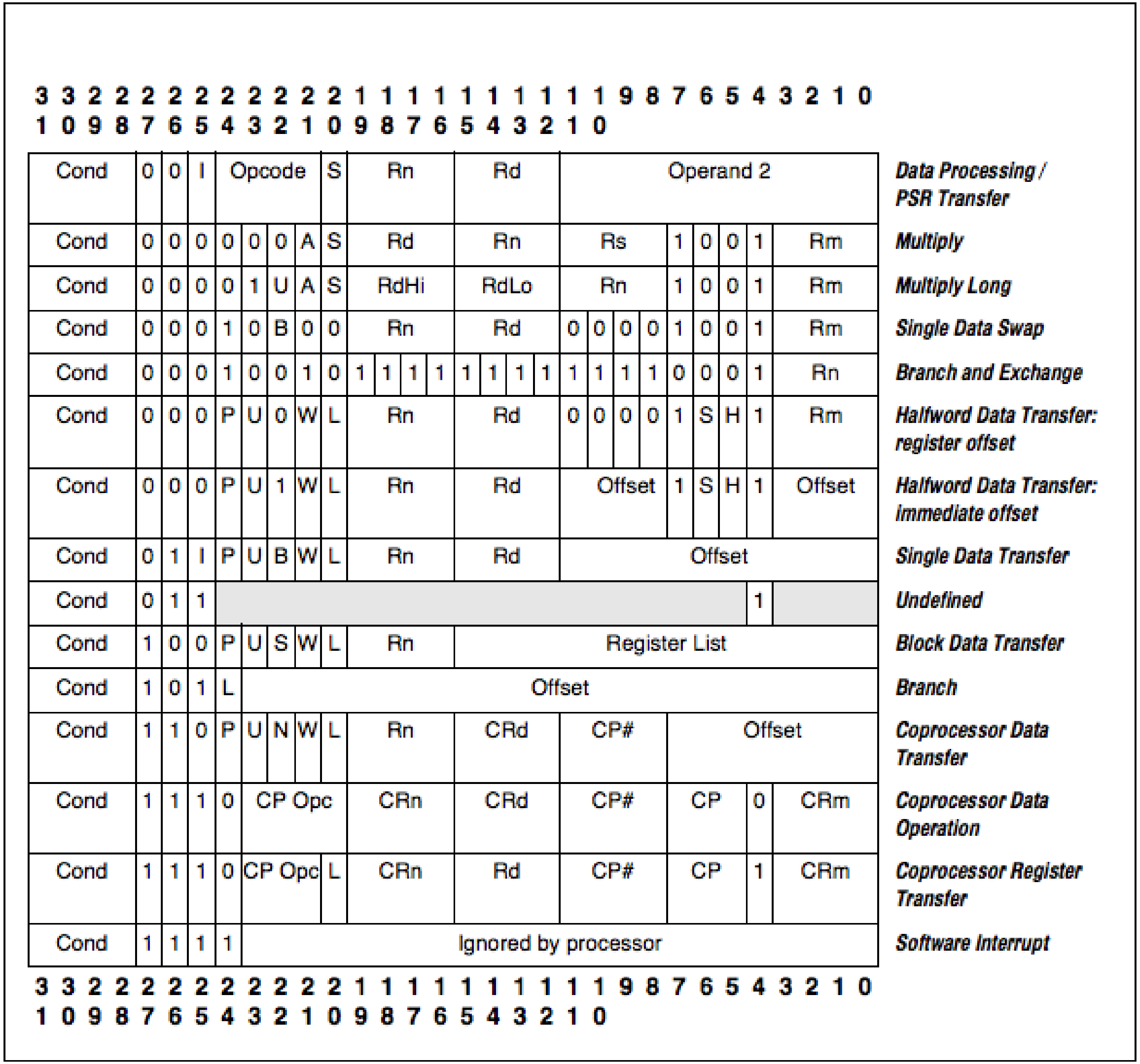
Opcode | Number — PALcode Format

Opcode | RA | Disp — Branch Format

Opcode | RA | RB | Disp — Memory Format

Opcode | RA | RB | Function | RC — Operate Format

ARM



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	
Cond	0	0	I	Opcode				S	Rn	Rd	Operand 2						*Data Processing / PSR Transfer*
Cond	0	0	0	0	0	0	A	S	Rd	Rn	Rs	1	0	0	1	Rm	*Multiply*
Cond	0	0	0	0	1	U	A	S	RdHi	RdLo	Rn	1	0	0	1	Rm	*Multiply Long*
Cond	0	0	0	1	0	B	0	0	Rn	Rd	0 0 0 0	1	0	0	1	Rm	*Single Data Swap*
Cond	0	0	0	1	0	0	1	0	1 1 1 1	1 1 1 1	1 1 1 1	0	0	0	1	Rn	*Branch and Exchange*
Cond	0	0	0	P	U	0	W	L	Rn	Rd	0 0 0 0	1	S	H	1	Rm	*Halfword Data Transfer: register offset*
Cond	0	0	0	P	U	1	W	L	Rn	Rd	Offset	1	S	H	1	Offset	*Halfword Data Transfer: immediate offset*
Cond	0	1	I	P	U	B	W	L	Rn	Rd	Offset						*Single Data Transfer*
Cond	0	1	1												1		*Undefined*
Cond	1	0	0	P	U	S	W	L	Rn	Register List							*Block Data Transfer*
Cond	1	0	1	L	Offset												*Branch*
Cond	1	1	0	P	U	N	W	L	Rn	CRd	CP#	Offset					*Coprocessor Data Transfer*
Cond	1	1	1	0	CP Opc				CRn	CRd	CP#	CP			0	CRm	*Coprocessor Data Operation*
Cond	1	1	1	0	CP Opc			L	CRn	Rd	CP#	CP			1	CRm	*Coprocessor Register Transfer*
Cond	1	1	1	1	Ignored by processor												*Software Interrupt*
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	

Figure 4-1: ARM instruction set formats

A Note on RISC vs. CISC

- Usually, ...

- RISC
  - Simple instructions
  - Fixed length
  - Uniform decode
  - Few addressing modes

- CISC
  - Complex instructions
  - Variable length
  - Non-uniform decode
  - Many addressing modes

ISA-level Tradeoffs: Number of Registers

- Affects:
  - Number of bits used for encoding register address
  - Number of values kept in fast storage (register file)
  - (uarch) Size, access time, power consumption of register file

- Large number of registers:

  + Enables better register allocation (and optimizations) by compiler → fewer saves/restores

  -- Larger instruction size

  -- Larger register file size

ISA-level Tradeoffs: Addressing Modes

- Addressing mode specifies how to obtain an operand of an instruction
  - Register
  - Immediate
  - Memory (displacement, register indirect, indexed, absolute, memory indirect, autoincrement, autodecrement, ...)

- More modes:

  + help better support programming constructs (arrays, pointer-based accesses)

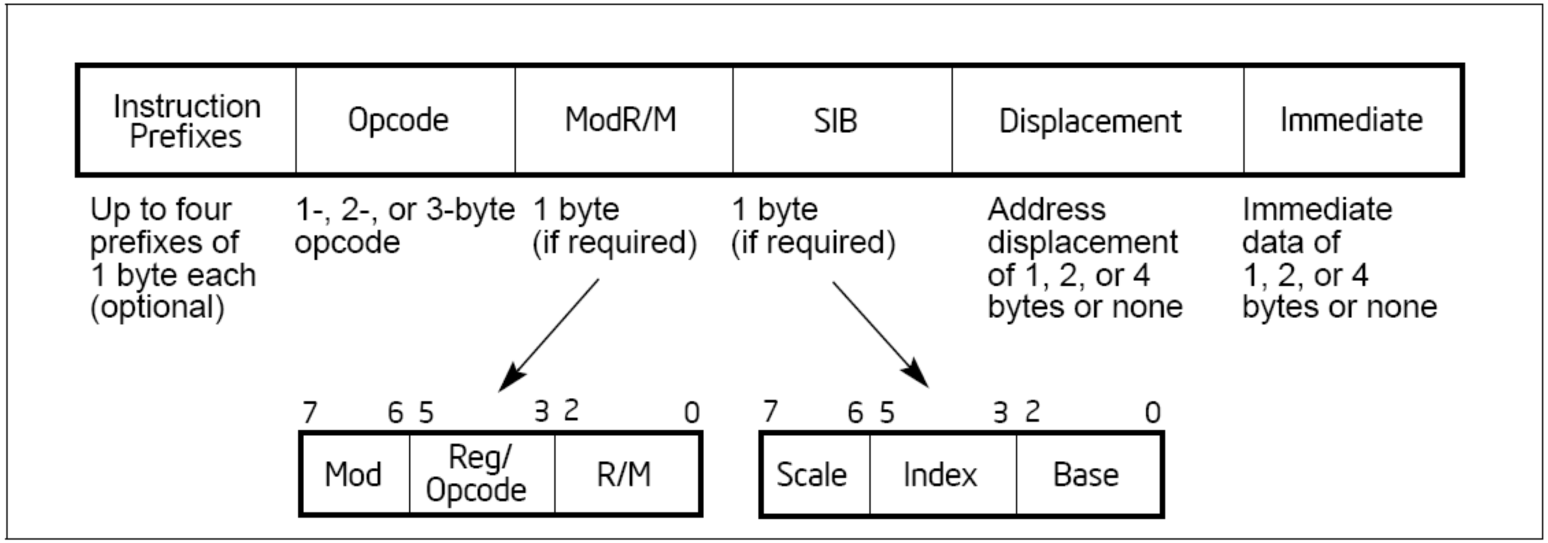
  -- make it harder for the architect to design

  -- too many choices for the compiler?

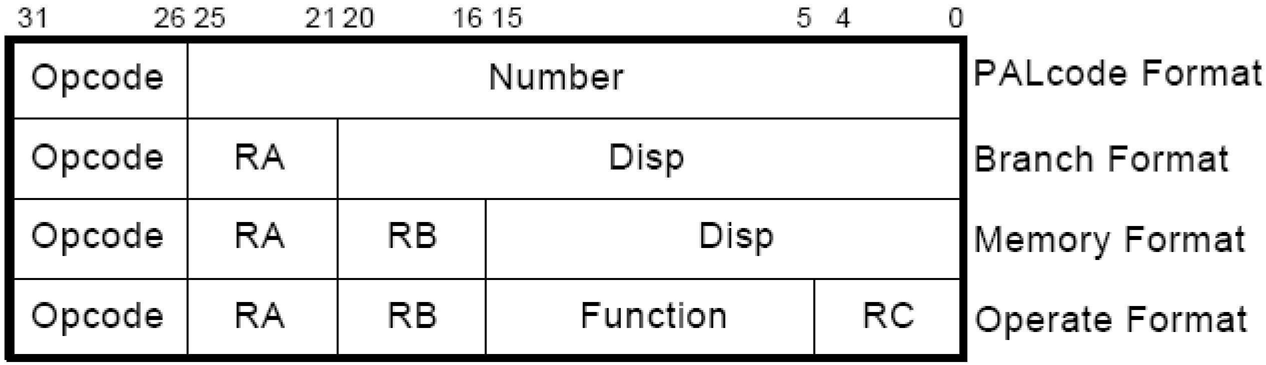
  - Many ways to do the same thing complicates compiler design
  - *Wulf, "Compilers and Computer Architecture," IEEE Computer 1981*

x86 vs. Alpha Instruction Formats

- x86:



- Alpha:



x86

register indirect

absolute

register + displacement

register

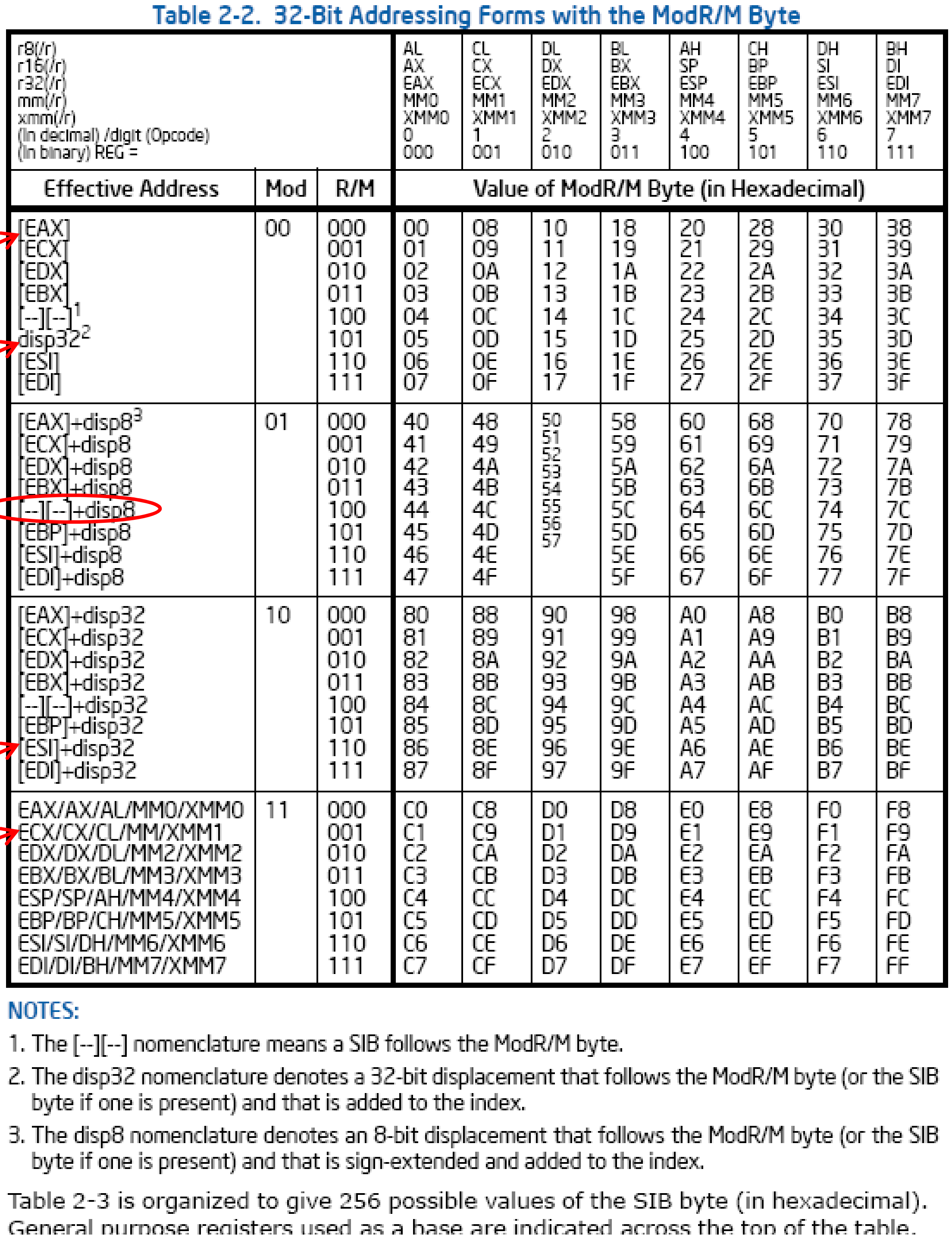


Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte

r8(/r)
r16(/r)
r32(/r)
mm(/r)
xmm(/r)
(In decimal) /digit (Opcode)
(In binary) REG =
			AL
			AX
			EAX
			MM0
			XMM0
			0
			000
				CL
				CX
				ECX
				MM1
				XMM1
				1
				001
					DL
					DX
					EDX
					MM2
					XMM2
					2
					010
						BL
						BX
						EBX
						MM3
						XMM3
						3
						011
							AH
							SP
							ESP
							MM4
							XMM4
							4
							100
								CH
								BP
								EBP
								MM5
								XMM5
								5
								101
									DH
									SI
									ESI
									MM6
									XMM6
									6
									110
										BH
										DI
										EDI
										MM7
										XMM7
										7
										111

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

NOTES:

1. The [--][--] nomenclature means a SIB follows the ModR/M byte.
2. The disp32 nomenclature denotes a 32-bit displacement that follows the ModR/M byte (or the SIB byte if one is present) and that is added to the index.
3. The disp8 nomenclature denotes an 8-bit displacement that follows the ModR/M byte (or the SIB byte if one is present) and that is sign-extended and added to the index.

Table 2-3 is organized to give 256 possible values of the SIB byte (in hexadecimal). General purpose registers used as a base are indicated across the top of the table.

x86

indexed
(base +
index)

scaled
(base +
index*4)

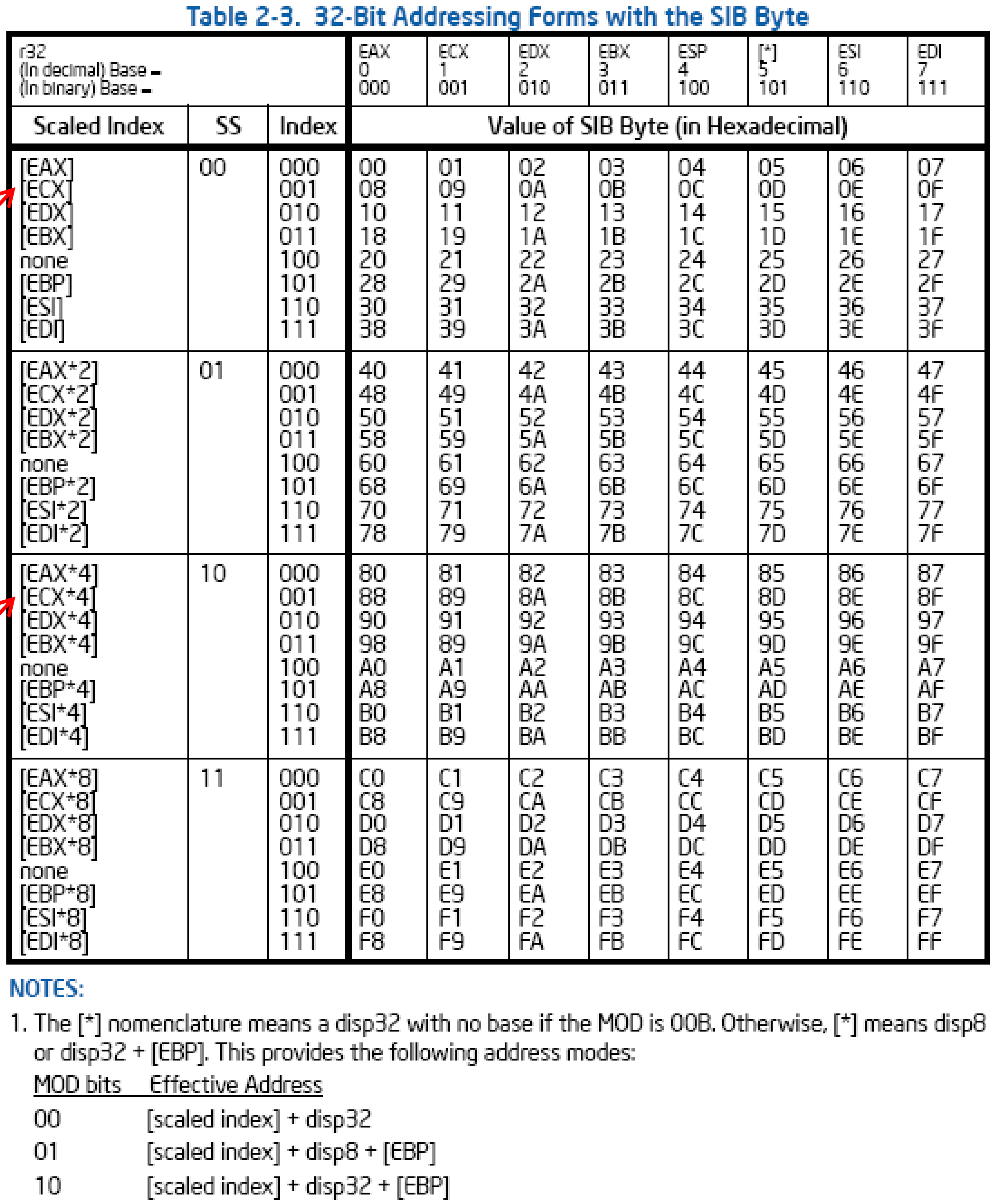


Table 2-3. 32-Bit Addressing Forms with the SIB Byte

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

NOTES:

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

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

X86 SIB-D Addressing Mode

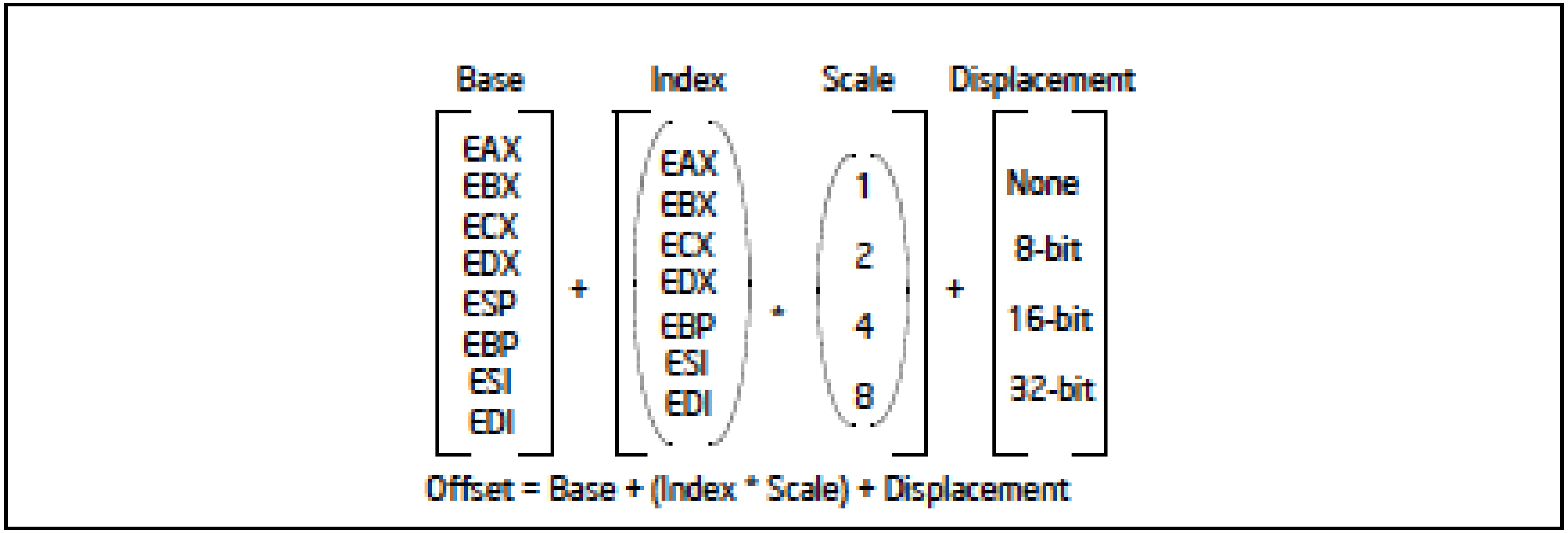
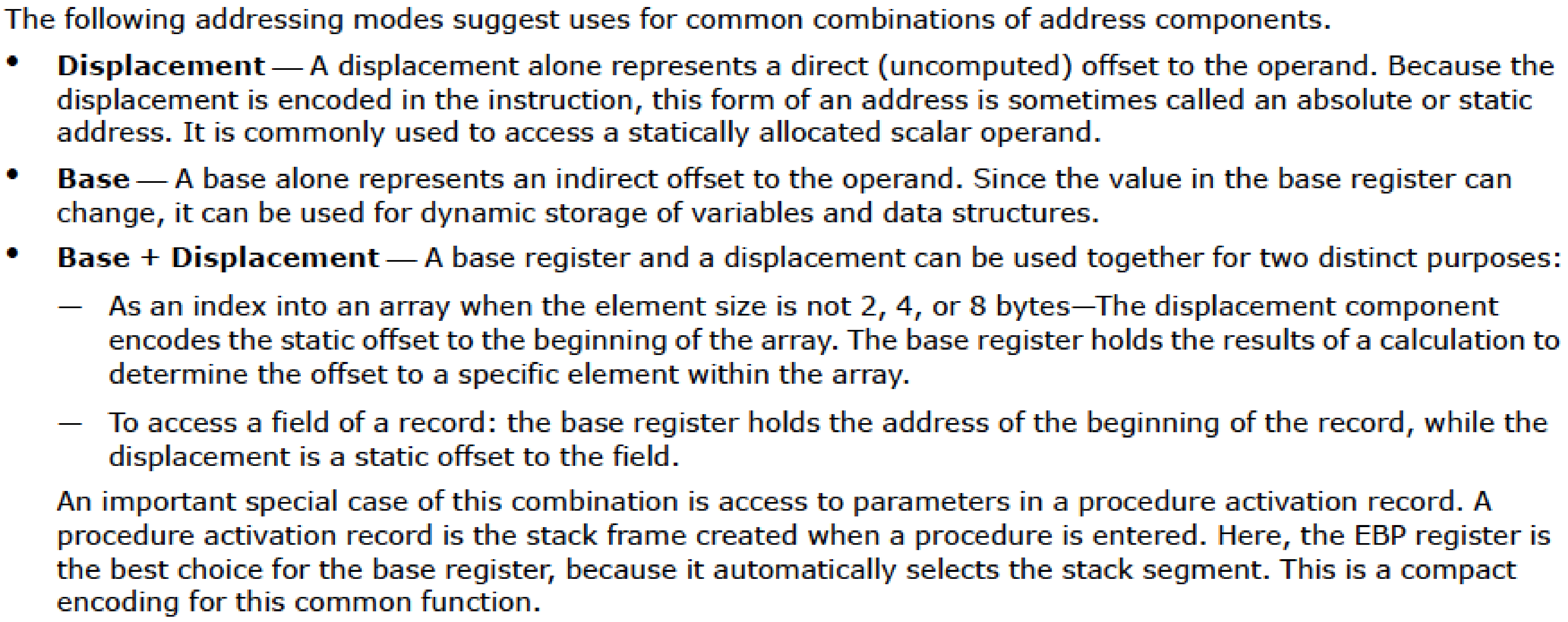


Figure 3-11. Offset (or Effective Address) Computation

x86 Manual Vol. 1, page 3-22
Also, see Section 3.7.3 and 3.7.5

X86 Manual: Suggested Uses of Addressing Modes



The following addressing modes suggest uses for common combinations of address components.

- **Displacement** — A displacement alone represents a direct (uncomputed) offset to the operand. Because the displacement is encoded in the instruction, this form of an address is sometimes called an absolute or static address. It is commonly used to access a statically allocated scalar operand.
- **Base** — A base alone represents an indirect offset to the operand. Since the value in the base register can change, it can be used for dynamic storage of variables and data structures.
- **Base + Displacement** — A base register and a displacement can be used together for two distinct purposes:
  - As an index into an array when the element size is not 2, 4, or 8 bytes—The displacement component encodes the static offset to the beginning of the array. The base register holds the results of a calculation to determine the offset to a specific element within the array.
  - To access a field of a record: the base register holds the address of the beginning of the record, while the displacement is a static offset to the field.

  An important special case of this combination is access to parameters in a procedure activation record. A procedure activation record is the stack frame created when a procedure is entered. Here, the EBP register is the best choice for the base register, because it automatically selects the stack segment. This is a compact encoding for this common function.

Other Example ISA-level Tradeoffs

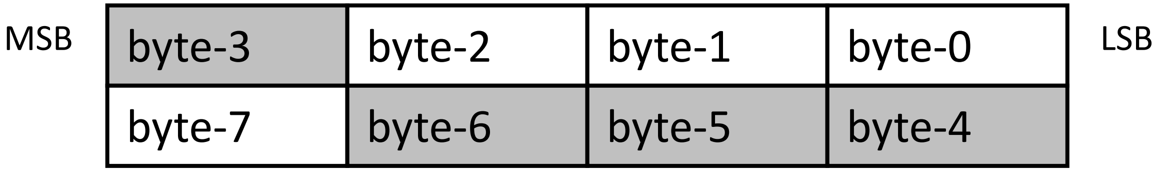
- Condition codes vs. not
- VLIW vs. single instruction
- Precise vs. imprecise exceptions
- Virtual memory vs. not
- Unaligned access vs. not
- Hardware interlocks vs. software-guaranteed interlocking
- Software vs. hardware managed page fault handling
- Cache coherence (hardware vs. software)
- ...

Back to Programmer vs. (Micro)architect

- Many ISA features designed to aid programmers
- But, complicate the hardware designer’s job

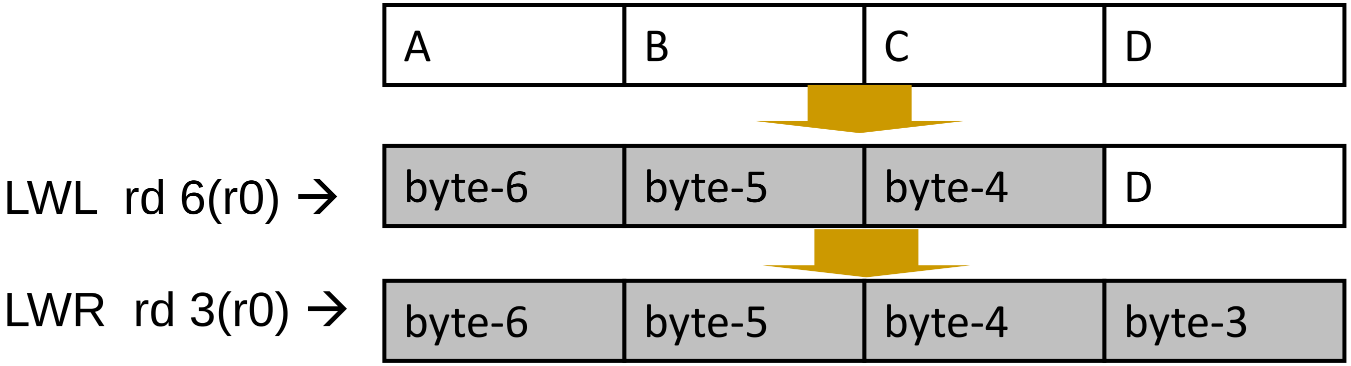
- Virtual memory
  - vs. overlay programming
  - Should the programmer be concerned about the size of code blocks fitting physical memory?
- Addressing modes
- Unaligned memory access
  - Compile/programmer needs to align data

MIPS: Aligned Access



MSB					LSB
	byte-3	byte-2	byte-1	byte-0	
	byte-7	byte-6	byte-5	byte-4	

- LW/SW alignment restriction: 4-byte word-alignment
  - not designed to fetch memory bytes not within a word boundary
  - not designed to rotate unaligned bytes into registers
- Provide separate opcodes for the “infrequent” case

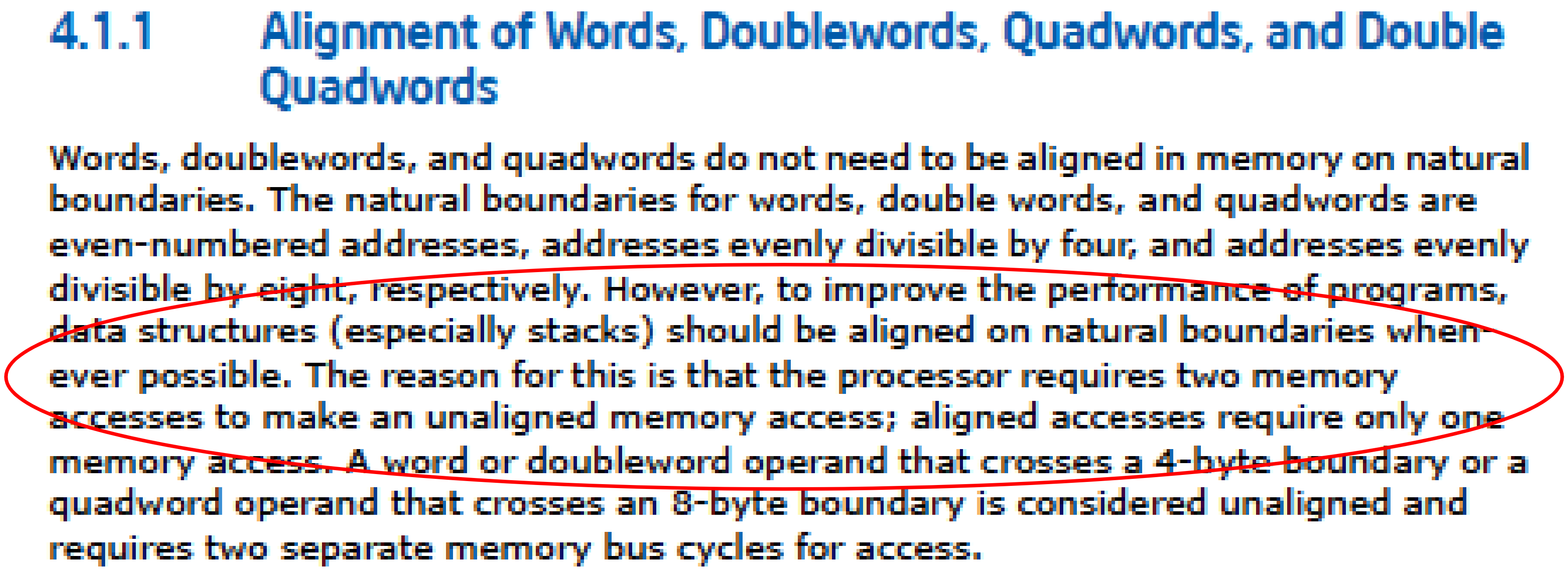


	A	B	C	D
LWL rd 6(r0) →	byte-6	byte-5	byte-4	D
LWR rd 3(r0) →	byte-6	byte-5	byte-4	byte-3

  - LWL/LWR is slower
  - Note LWL and LWR still fetch within word boundary

X86: Unaligned Access

- LD/ST instructions automatically align data that spans a “word” boundary
- Programmer/compiler does not need to worry about where data is stored (whether or not in a word-aligned location)



4.1.1 Alignment of Words, Doublewords, Quadwords, and Double Quadwords

Words, doublewords, and quadwords do not need to be aligned in memory on natural boundaries. The natural boundaries for words, double words, and quadwords are even-numbered addresses, addresses evenly divisible by four, and addresses evenly divisible by eight, respectively. However, to improve the performance of programs, data structures (especially stacks) should be aligned on natural boundaries whenever possible. The reason for this is that the processor requires two memory accesses to make an unaligned memory access; aligned accesses require only one memory access. A word or doubleword operand that crosses a 4-byte boundary or a quadword operand that crosses an 8-byte boundary is considered unaligned and requires two separate memory bus cycles for access.

X86: Unaligned Access

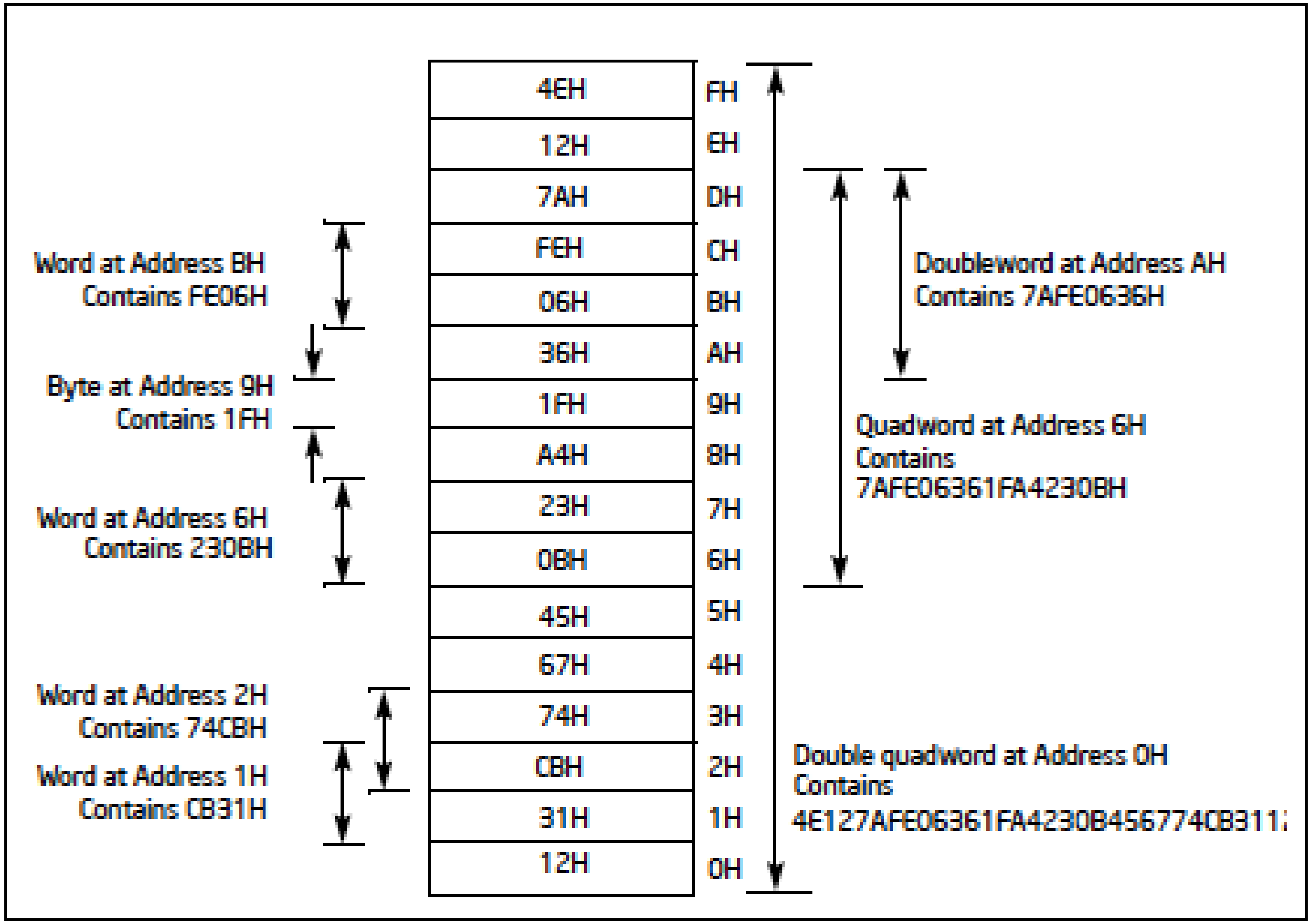

4EH
12H
7AH
FEH
06H
36H
1FH
A4H
23H
0BH
45H
67H
74H
CBH
31H
12H
FH
EH
DH
CH
BH
AH
9H
8H
7H
6H
5H
4H
3H
2H
1H
0H
Word at Address BH Contains FE06H
Byte at Address 9H Contains 1FH
Word at Address 6H Contains 230BH
Word at Address 2H Contains 74CBH
Word at Address 1H Contains CB31H
Doubleword at Address AH Contains 7AFE0636H
Quadword at Address 6H Contains 7AFE06361FA4230BH
Double quadword at Address 0H Contains 4E127AFE06361FA4230B456774CB311

Figure 4-2. Bytes, Words, Doublewords, Quadwords, and Double Quadwords in Memory

Aligned vs. Unaligned Access

- Pros of having no restrictions on alignment

- Cons of having no restrictions on alignment

- Filling in the above: an exercise for you...

Implementing the ISA: Microarchitecture Basics

How Does a Machine Process Instructions?

- What does processing an instruction mean?
- Remember the von Neumann model

A = Architectural (programmer visible) state before an instruction is processed

↓

Process instruction

↓

A’ = Architectural (programmer visible) state after an instruction is processed

- Processing an instruction: Transforming A to A’ according to the ISA specification of the instruction

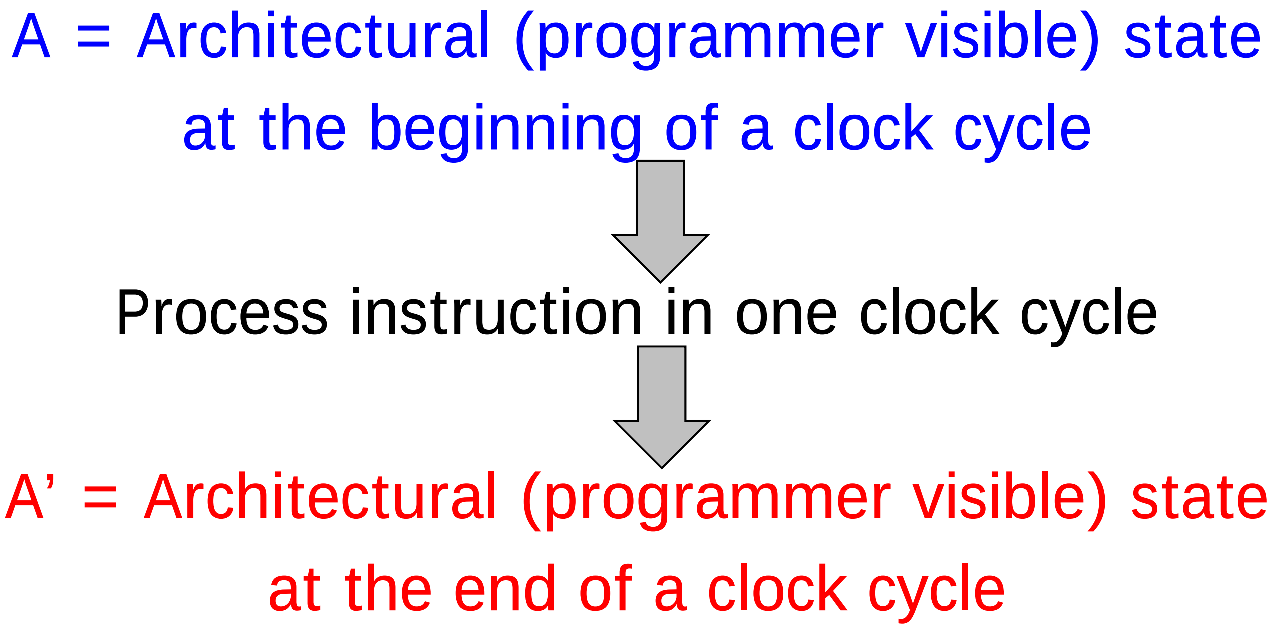
The "Process instruction" Step

- ISA specifies abstractly what A' should be, given an instruction and A
    - It defines an abstract finite state machine where
        - State = programmer-visible state
        - Next-state logic = instruction execution specification
    - From ISA point of view, there are no "intermediate states" between A and A' during instruction execution
        - One state transition per instruction

- Microarchitecture implements how A is transformed to A'
    - There are many choices in implementation
    - We can have programmer-invisible state to optimize the speed of instruction execution: multiple state transitions per instruction
        - Choice 1: A → A' (transform A to A' in a single clock cycle)
        - Choice 2: A → A+MS1 → A+MS2 → A+MS3 → A' (take multiple clock cycles to transform A to A')

A Very Basic Instruction Processing Engine

- Each instruction takes a single clock cycle to execute
- Only combinational logic is used to implement instruction execution
  - *No intermediate, programmer-invisible state updates*



A = Architectural (programmer visible) state

at the beginning of a clock cycle

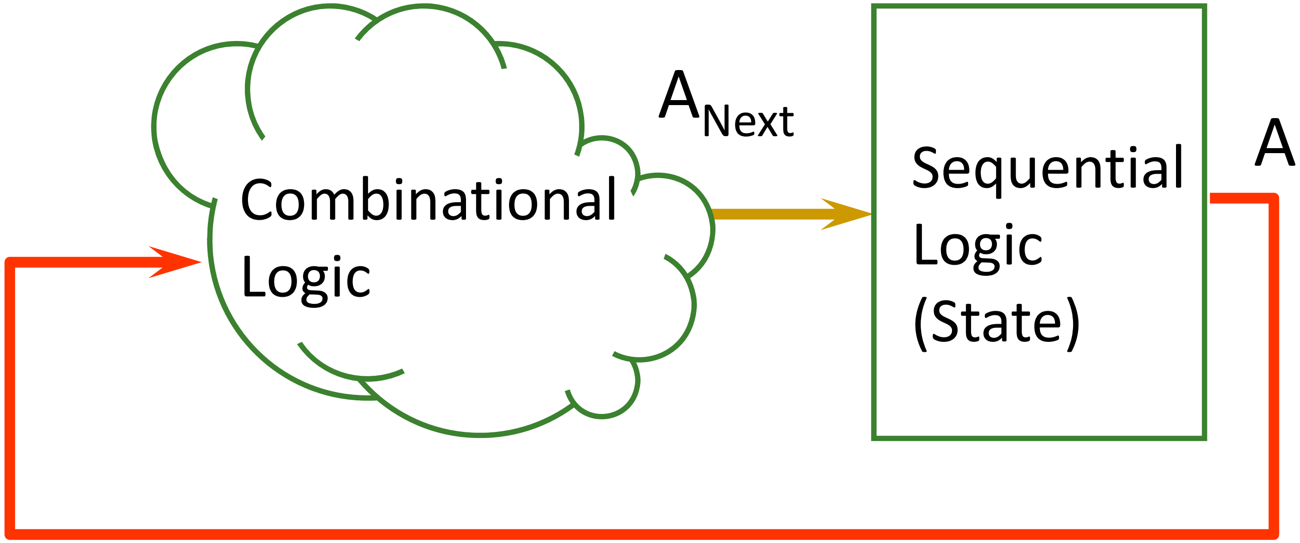
Process instruction in one clock cycle

A' = Architectural (programmer visible) state

at the end of a clock cycle

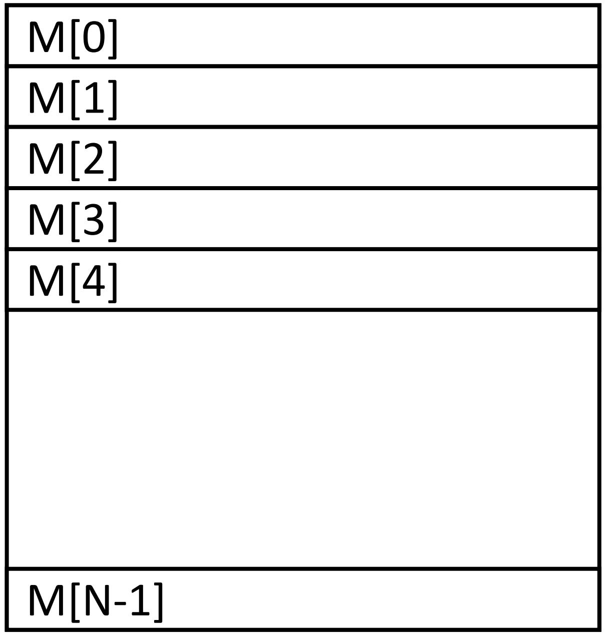
A Very Basic Instruction Processing Engine

- Single-cycle machine

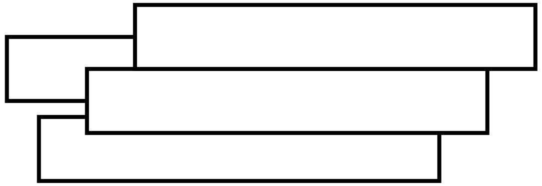


- What is the *clock cycle time* determined by?
- What is the *critical path* of the combinational logic determined by?

Remember: Programmer Visible (Architectural) State



Memory
array of storage locations
indexed by an address



Registers
- given special names in the ISA
  (as opposed to addresses)
- general vs. special purpose

Program Counter
memory address
of the current instruction

Instructions (and programs) specify how to transform
the values of programmer visible state

Single-cycle vs. Multi-cycle Machines

- Single-cycle machines
  - Each instruction takes a single clock cycle
  - All state updates made at the end of an instruction's execution
  - Big disadvantage: The slowest instruction determines cycle time → long clock cycle time

- Multi-cycle machines
  - Instruction processing broken into multiple cycles/stages
  - State updates can be made during an instruction's execution
  - Architectural state updates made only at the end of an instruction's execution
  - Advantage over single-cycle: The slowest "stage" determines cycle time

- Both single-cycle and multi-cycle machines literally follow the von Neumann model at the microarchitecture level

Instruction Processing "Cycle"

- Instructions are processed under the direction of a "control unit" step by step.
- Instruction cycle: Sequence of steps to process an instruction
- Fundamentally, there are six phases:

- Fetch
- Decode
- Evaluate Address
- Fetch Operands
- Execute
- Store Result

- Not all instructions require all six stages (see P&P Ch. 4)

Instruction Processing "Cycle" vs. Machine Clock Cycle

- Single-cycle machine:
  - All six phases of the instruction processing cycle take a *single machine clock cycle* to complete

- Multi-cycle machine:
  - All six phases of the instruction processing cycle can take *multiple machine clock cycles* to complete
  - In fact, each phase can take multiple clock cycles to complete

Instruction Processing Viewed Another Way

- Instructions transform Data (AS) to Data’ (AS’)
- This transformation is done by functional units
  - Units that “operate” on data
- These units need to be told what to do to the data

- An instruction processing engine consists of two components
  - Datapath: Consists of hardware elements that deal with and transform data signals
    - functional units that operate on data
    - hardware structures (e.g. wires and muxes) that enable the flow of data into the functional units and registers
    - storage units that store data (e.g., registers)
  - Control logic: Consists of hardware elements that determine control signals, i.e., signals that specify what the datapath elements should do to the data

Single-cycle vs. Multi-cycle: Control & Data

- Single-cycle machine:
  - Control signals are generated in the same clock cycle as data signals are operated on
  - Everything related to an instruction happens in one clock cycle

- Multi-cycle machine:
  - Control signals needed in the next cycle can be generated in the previous cycle
  - Latency of control processing can be overlapped with latency of datapath operation

- We will see the difference clearly in *microprogrammed multi-cycle microarchitecture*

Many Ways of Datapath and Control Design

- There are many ways of designing the data path and control logic

- Single-cycle, multi-cycle, pipelined datapath and control
- Single-bus vs. multi-bus datapaths
  - See your homework 2 question
- Hardwired/combinational vs. microcoded/microprogrammed control
  - Control signals generated by combinational logic versus
  - Control signals stored in a memory structure

- Control signals and structure depend on the datapath design

Flash-Forward: Performance Analysis

- Execution time of an instruction
  - {CPI} x {clock cycle time}
- Execution time of a program
  - Sum over all instructions [{CPI} x {clock cycle time}]
  - {# of instructions} x {Average CPI} x {clock cycle time}

- Single cycle microarchitecture performance
  - CPI = 1
  - Clock cycle time = long
- Multi-cycle microarchitecture performance
  - CPI = different for each instruction
    - Average CPI → hopefully small
  - Clock cycle time = short

**Now, we have
two degrees of freedom
to optimize independently**