INSTRUCTION SET ARCHITECTURE

Classes of ISA

Two popular versions

- Register-Memory ISA

  eg:80x86

- Load-store ISA

  eg:MIPS

Memory Addressing

Use byte addressing to access memory operands

Addressing Modes

specifies the address of an operand/memory object

Types and sizes of operands

- MIPS and 80x86 support operand sizes of 8 bit,16bit,32bit,64 bit
- 80x86 supports 80 bit floating point(extended double precision)

Operations

MIPS is a simple and easy-to-pipeline ISA, 80x86 has a much larger and richer set of operations.

Encoding ISA

- Fixed length

  Eg:MIPS (32 bit)

- Variable length

  Eg:80x86(1 to 18 bytes)

Classifying ISA

There are mainly **four classes of instruction set**

- **Stack**

  operands are implicitly on the top of the stack- operations are carried out there.

- **Accumulator**

  one operand is implicitly in the accumulator- other operand is specified along with the instruction.

- **Register-Memory**

  only explicit operands- one operand is in register and the other one in memory.

- **Register-Register**

  load-store architecture, only load and store instructions have memory operands.

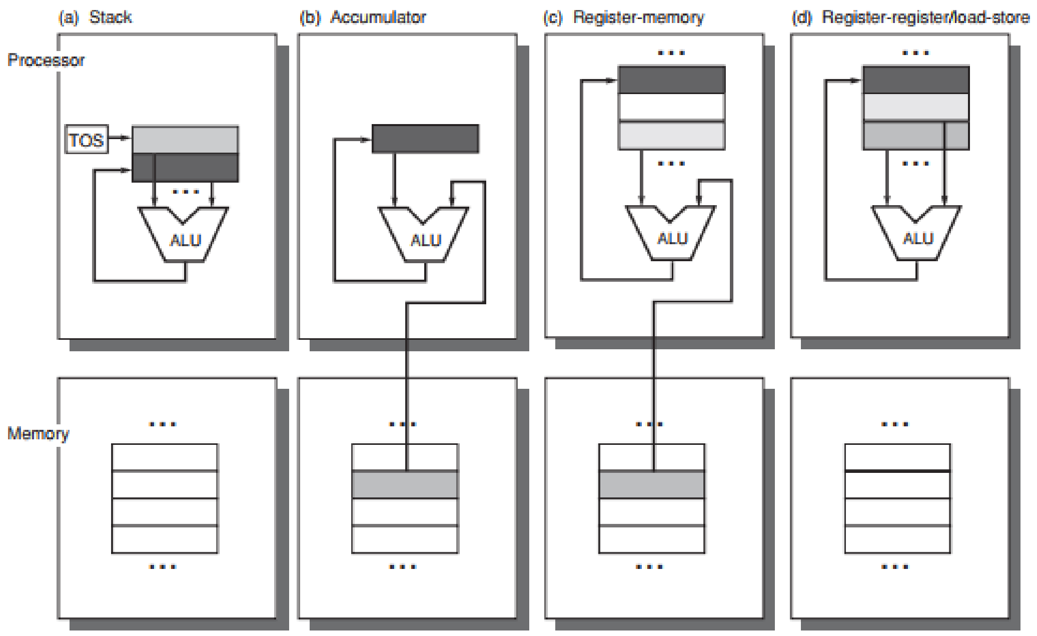
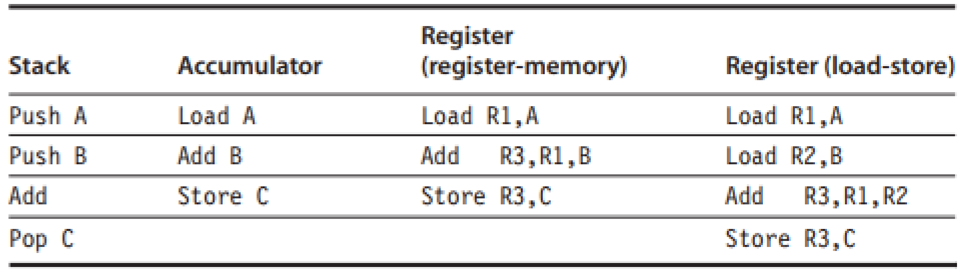
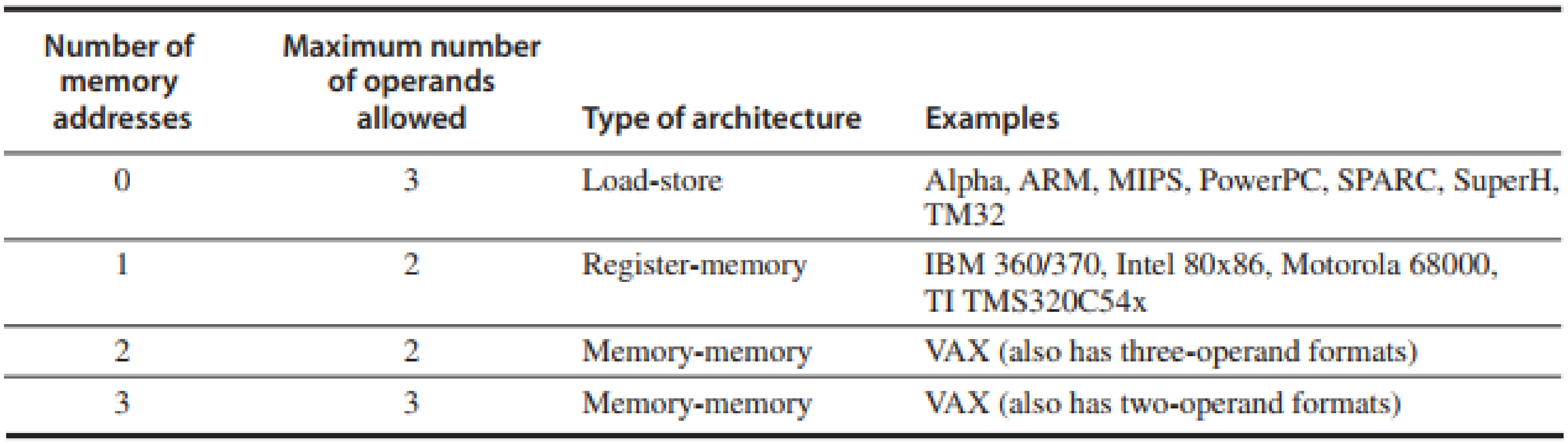


Figure B.1 Operand locations for four instruction set architecture classes. The arrows indicate whether the operand is an input or the result of the ALU operation, or both an input and result. Lighter shades indicate inputs, and the dark shade indicates the result. In (a), a Top Of Stack register (TOS), points to the top input operand, which is combined with the operand below. The first operand is removed from the stack, the result takes the place of the second operand, and TOS is updated to point to the result. All operands are implicit. In (b), the Accumulator is both an implicit input operand and a result. In (c), one input operand is a register, one is in memory, and the result goes to a register. All operands are registers in (d) and, like the stack architecture, can be transferred to memory only via separate instructions: push or pop for (a) and load or store for (d).



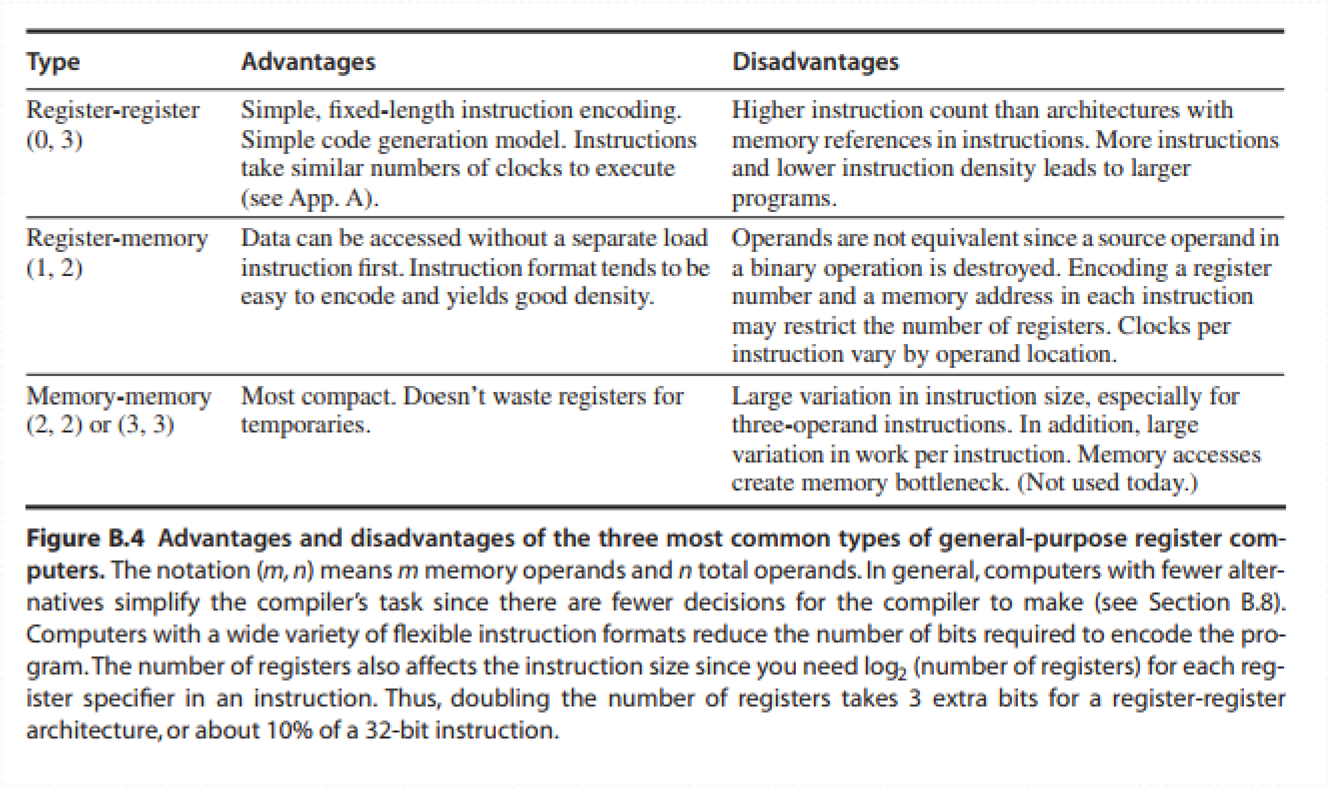
Stack	Accumulator	Register (register-memory)	Register (load-store)
`Push A`	`Load A`	`Load R1,A`	`Load R1,A`
`Push B`	`Add B`	`Add R3,R1,B`	`Load R2,B`
`Add`	`Store C`	`Store R3,C`	`Add R3,R1,R2`
`Pop C`			`Store R3,C`

Figure B.2 The code sequence for C = A + B for four classes of instruction sets. Note that the **Add** instruction has implicit operands for stack and accumulator architectures, and explicit operands for register architectures. It is assumed that A, B, and C all belong in memory and that the values of A and B cannot be destroyed. Figure B.1 shows the **Add** operation for each class of architecture.



Number of memory addresses	Maximum number of operands allowed	Type of architecture	Examples
0	3	Load-store	Alpha, ARM, MIPS, PowerPC, SPARC, SuperH, TM32
1	2	Register-memory	IBM 360/370, Intel 80x86, Motorola 68000, TI TMS320C54x
2	2	Memory-memory	VAX (also has three-operand formats)
3	3	Memory-memory	VAX (also has two-operand formats)

Figure B.3 Typical combinations of memory operands and total operands per typical ALU instruction with examples of computers. Computers with no memory reference per ALU instruction are called load-store or register-register computers. Instructions with multiple memory operands per typical ALU instruction are called register-memory or memory-memory, according to whether they have one or more than one memory operand.



Type	Advantages	Disadvantages
Register-register (0, 3)	Simple, fixed-length instruction encoding. Simple code generation model. Instructions take similar numbers of clocks to execute (see App. A).	Higher instruction count than architectures with memory references in instructions. More instructions and lower instruction density leads to larger programs.
Register-memory (1, 2)	Data can be accessed without a separate load instruction first. Instruction format tends to be easy to encode and yields good density.	Operands are not equivalent since a source operand in a binary operation is destroyed. Encoding a register number and a memory address in each instruction may restrict the number of registers. Clocks per instruction vary by operand location.
Memory-memory (2, 2) or (3, 3)	Most compact. Doesn't waste registers for temporaries.	Large variation in instruction size, especially for three-operand instructions. In addition, large variation in work per instruction. Memory accesses create memory bottleneck. (Not used today.)

Figure B.4 Advantages and disadvantages of the three most common types of general-purpose register computers. The notation (*m*, *n*) means *m* memory operands and *n* total operands. In general, computers with fewer alternatives simplify the compiler's task since there are fewer decisions for the compiler to make (see Section B.8). Computers with a wide variety of flexible instruction formats reduce the number of bits required to encode the program. The number of registers also affects the instruction size since you need $\log_2$ (number of registers) for each register specifier in an instruction. Thus, doubling the number of registers takes 3 extra bits for a register-register architecture, or about 10% of a 32-bit instruction.

Memory Addressing

The architecture must define how memory addresses are interpreted and how they are specified.

Memory consists of storage cells, each of which can store one bit of information(0 or 1)

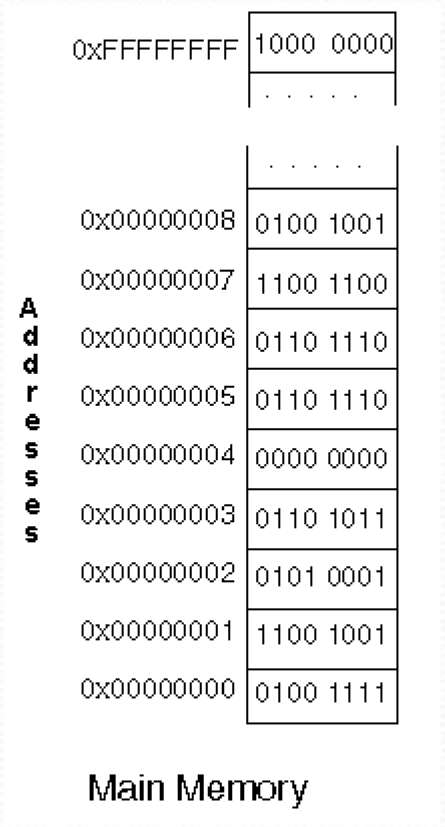
Memory is organized so that a **group of n bits** can be stored or retrieved in a single basic operation.

- Each group of n bits is referred to as a **word** of information, and n is called **word length.**
- Modern computers have word length ranges from **16** to **64** bits.
- Accessing the memory to store or retrieve a single item of information , requires distinct addresses for each item location.

It is customary to use numbers through
0 to 2^k-1 for some suitable values of **k.**

2^k addresses constitute the address space of the computer.

Byte addressability



Byte addressability

- Most practical assignment is to have successive address refer to successive byte locations in the memory (byte addressable memory)
- Byte locations have addresses 0,1,2,……
- If the word length of the machine is **32 bits**, successive words are located at the addresses **0,4,8**, …. With each consisting of 4 bytes.

Big Endian and Little Endian Assignments

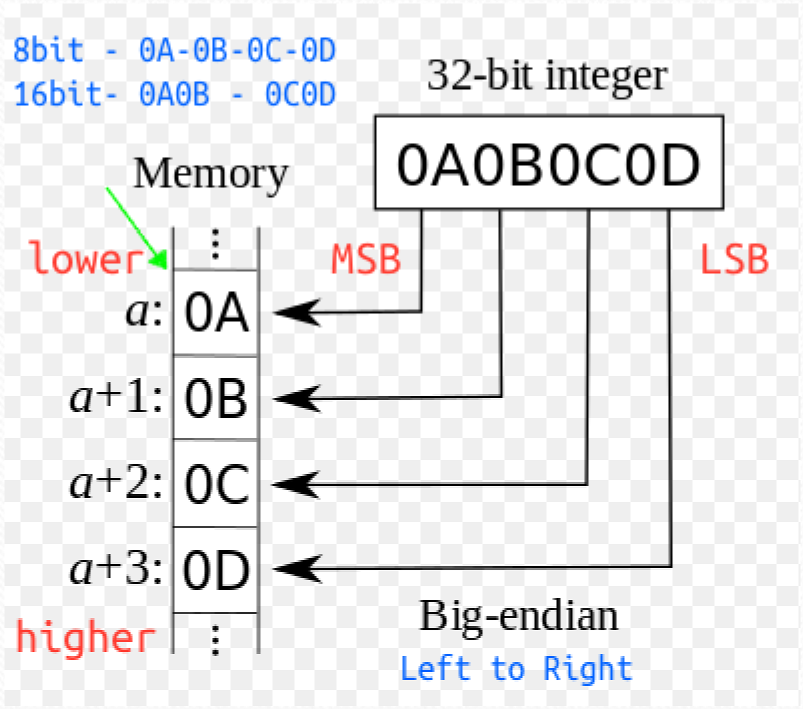
- **Big Endian** is used when lower byte addresses are used for the Most Significant Byte(leftmost byte)of the word

Eg: ARM,PowerPC,MIPS

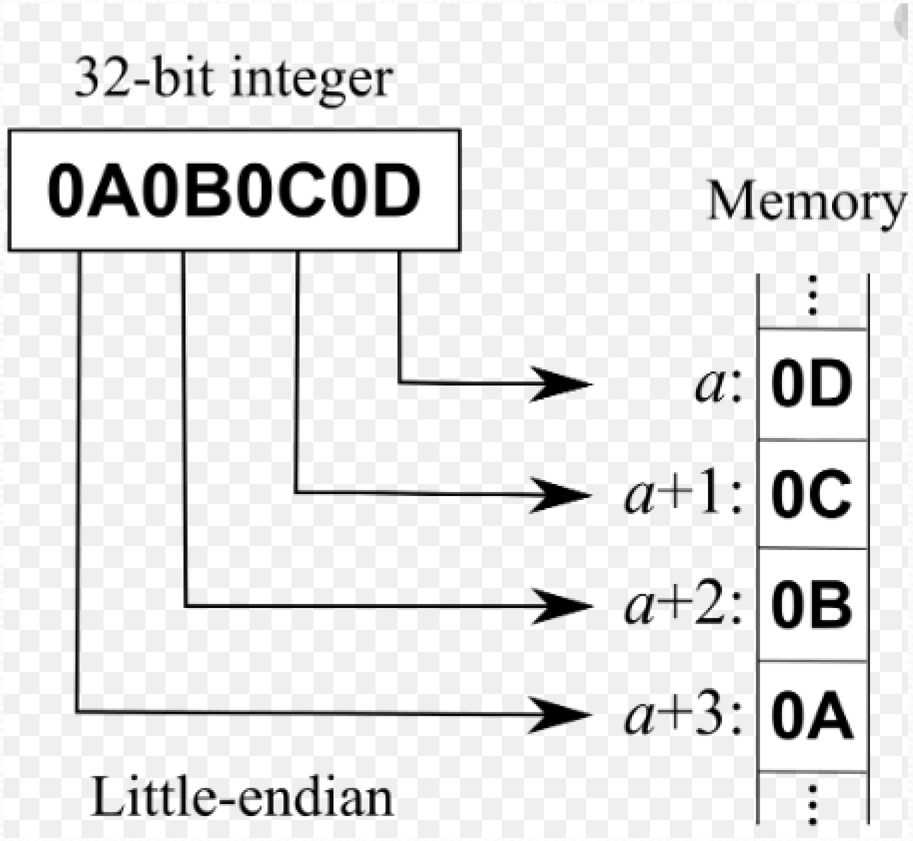
- **Little Endian** is used when lower byte addresses are used for the Least Significant Byte (rightmost bytes) of the word

Eg: x86, VAX

Big Endian

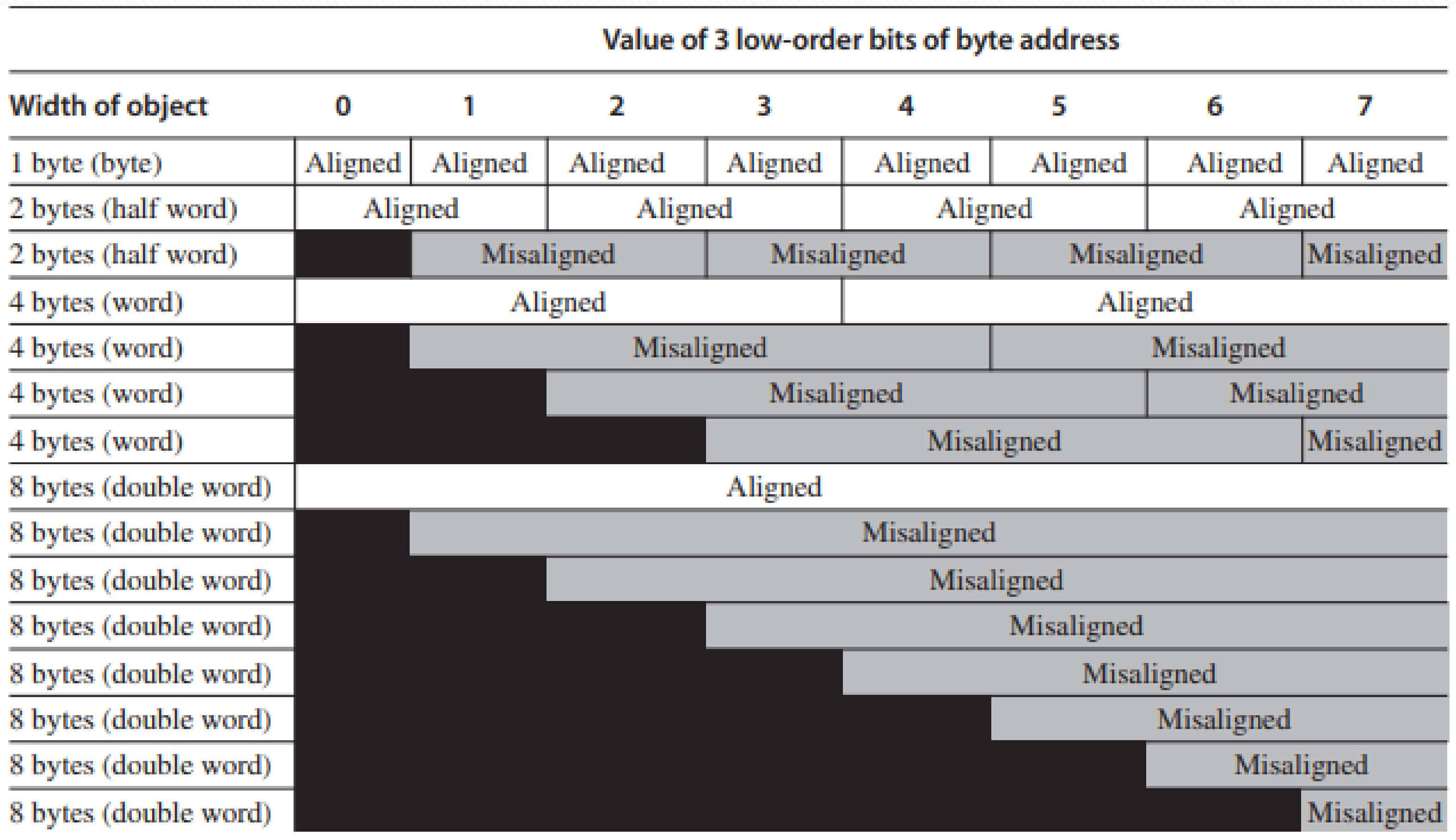

8bit - 0A-0B-0C-0D
16bit- 0A0B - 0C0D
32-bit integer
0A0B0C0D
Memory
lower
MSB
LSB
a: 0A
a+1: 0B
a+2: 0C
a+3: 0D
higher
Big-endian
Left to Right

Little Endian



Aligned and misaligned addresses

- Accesses to objects larger than a byte must be aligned, as misalignment causes hardware complications.
- An access to object of size **s bytes** at byte **address A** is aligned if **A mod s = 0**



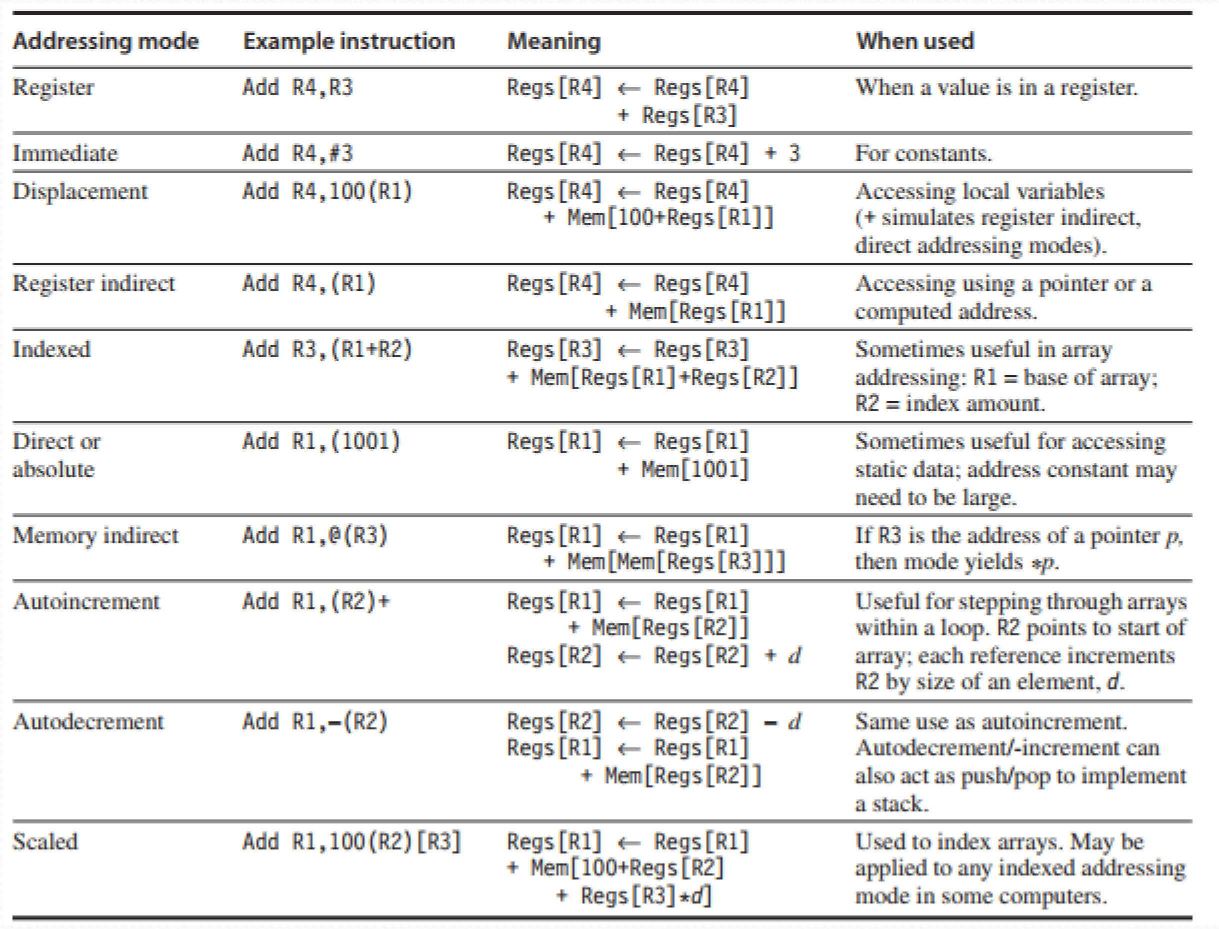
	Value of 3 low-order bits of byte address							
Width of object	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
1 byte (byte)	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned
2 bytes (half word)	Aligned		Aligned		Aligned		Aligned	
2 bytes (half word)	■	Misaligned		Misaligned		Misaligned		Misaligned
4 bytes (word)	Aligned				Aligned			
4 bytes (word)	■	Misaligned				Misaligned		
4 bytes (word)	■	■	Misaligned				Misaligned	
4 bytes (word)	■	■	■	Misaligned				Misaligned
8 bytes (double word)	Aligned							
8 bytes (double word)	■	Misaligned						
8 bytes (double word)	■	■	Misaligned					
8 bytes (double word)	■	■	■	Misaligned				
8 bytes (double word)	■	■	■	■	Misaligned			
8 bytes (double word)	■	■	■	■	■	Misaligned		
8 bytes (double word)	■	■	■	■	■	■	Misaligned	
8 bytes (double word)	■	■	■	■	■	■	■	Misaligned

Figure B.5 Aligned and misaligned addresses of byte, half-word, word, and double-word objects for byte-addressed computers. For each misaligned example some objects require two memory accesses to complete. Every aligned object can always complete in one memory access, as long as the memory is as wide as the object. The figure shows the memory organized as 8 bytes wide. The byte offsets that label the columns specify the low-order 3 bits of the address.

Addressing Modes

- Programs are written in high level language, which enables the programmer to use constants, local and global variables, pointers and arrays.
- When translating a high level language program to assembly language , the compiler must be able to implement these constructs using the facilities provided in the instruction set of the computer in which the program will be run.

- The different ways in which the **location of an operand** is specified in an instruction are referred as **addressing modes.**
- It specify the address of an object they will access- it specify constants and registers in addition to locations in memory.
- When a memory location is used, the actual memory address specified by the addressing mode is called the **effective address.**



Addressing mode	Example instruction	Meaning	When used
Register	`Add R4,R3`	`Regs[R4] ← Regs[R4] + Regs[R3]`	When a value is in a register.
Immediate	`Add R4,#3`	`Regs[R4] ← Regs[R4] + 3`	For constants.
Displacement	`Add R4,100(R1)`	`Regs[R4] ← Regs[R4] + Mem[100+Regs[R1]]`	Accessing local variables (+ simulates register indirect, direct addressing modes).
Register indirect	`Add R4,(R1)`	`Regs[R4] ← Regs[R4] + Mem[Regs[R1]]`	Accessing using a pointer or a computed address.
Indexed	`Add R3,(R1+R2)`	`Regs[R3] ← Regs[R3] + Mem[Regs[R1]+Regs[R2]]`	Sometimes useful in array addressing: R1 = base of array; R2 = index amount.
Direct or absolute	`Add R1,(1001)`	`Regs[R1] ← Regs[R1] + Mem[1001]`	Sometimes useful for accessing static data; address constant may need to be large.
Memory indirect	`Add R1,@(R3)`	`Regs[R1] ← Regs[R1] + Mem[Mem[Regs[R3]]]`	If R3 is the address of a pointer *p*, then mode yields **p*.
Autoincrement	`Add R1,(R2)+`	`Regs[R1] ← Regs[R1] + Mem[Regs[R2]]` `Regs[R2] ← Regs[R2] + d`	Useful for stepping through arrays within a loop. R2 points to start of array; each reference increments R2 by size of an element, *d*.
Autodecrement	`Add R1,–(R2)`	`Regs[R2] ← Regs[R2] – d` `Regs[R1] ← Regs[R1] + Mem[Regs[R2]]`	Same use as autoincrement. Autodecrement/-increment can also act as push/pop to implement a stack.
Scaled	`Add R1,100(R2)[R3]`	`Regs[R1] ← Regs[R1] + Mem[100+Regs[R2] + Regs[R3]*d]`	Used to index arrays. May be applied to any indexed addressing mode in some computers.

Register

ex: `ADD r1, r2, r3`
means: `r1 ← r2 + r3`
comment:used when a value is in a register.

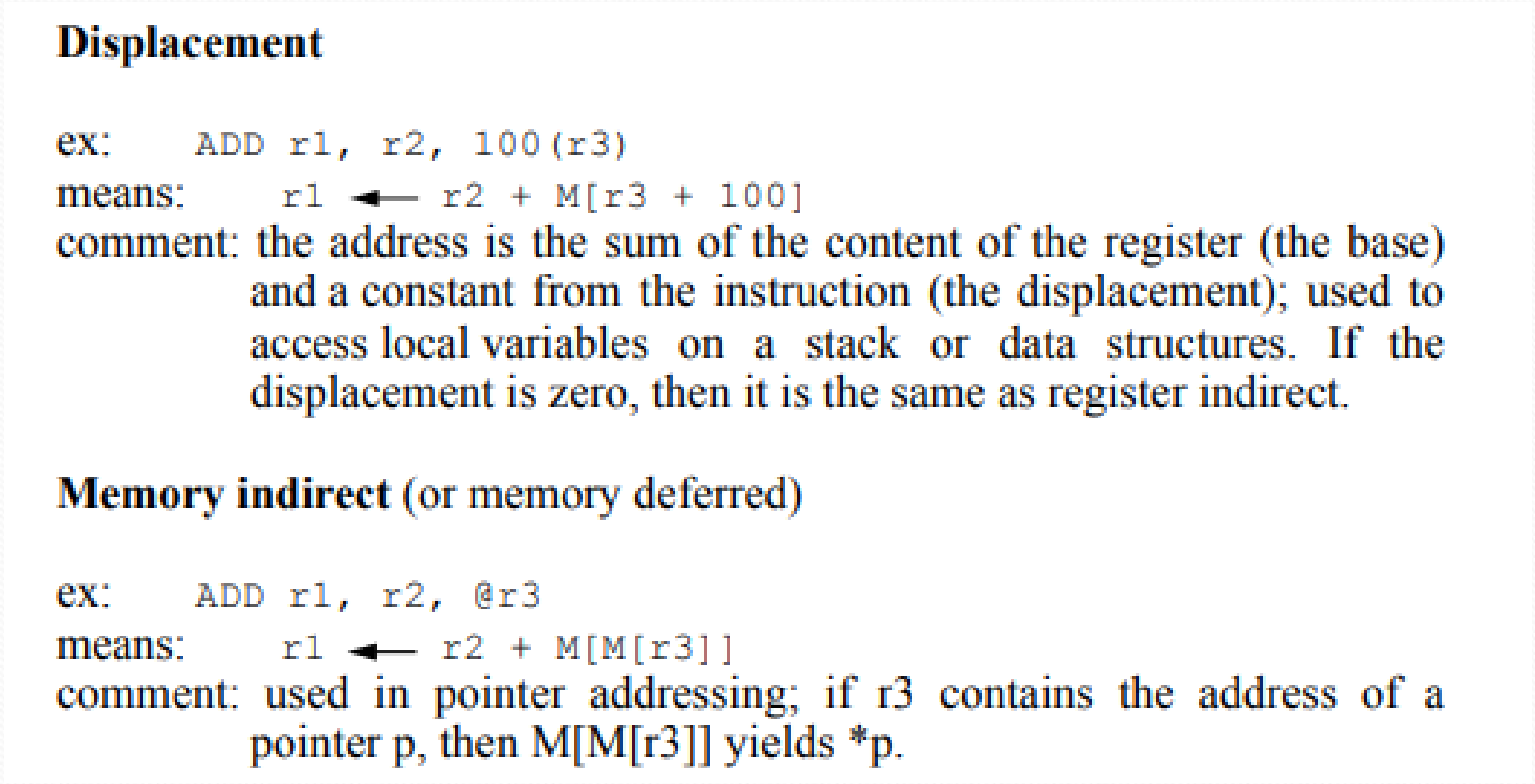
Immediate (or literal)

ex: `ADD r1, r2, 1`
means: `r1 ← r2 + 1`
comment:used when a constant is needed.

Direct
ex: `ADD r1, r2, (100)`
means: `r1 ← r2 + M[100]`
comment: used to access static data; the address of the operand is include in the instruction; space must be provided to accommodate a whole address.

Register indirect (or register deferred)

ex: `ADD r1, r2, (r3)`
means: `r1 ← r2 + M[r3]`
comment: the register (r3 in this example) contains the address of a memory location.



Displacement

ex: `ADD r1, r2, 100(r3)`

means: `r1` $\leftarrow$ `r2 + M[r3 + 100]`

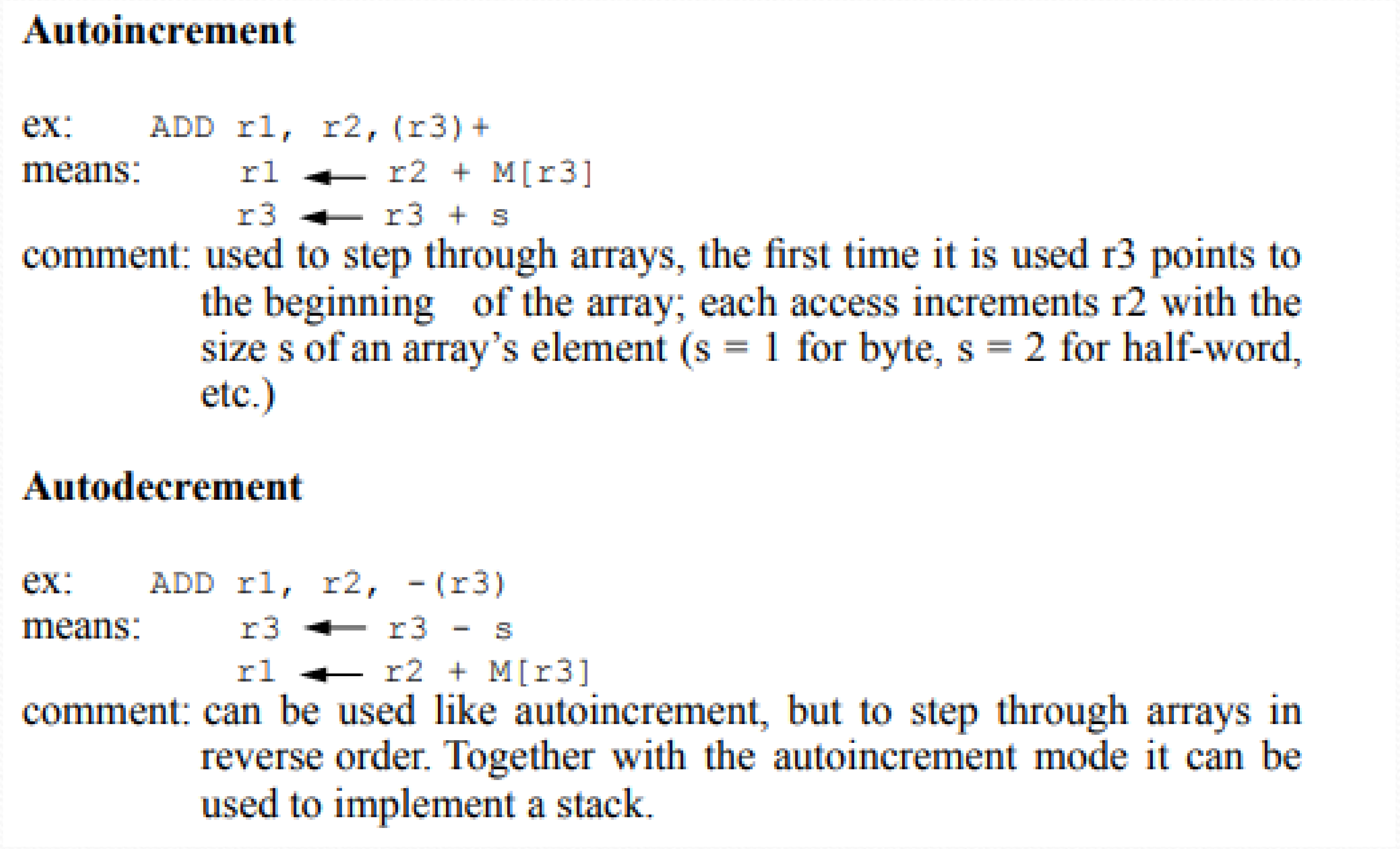
comment: the address is the sum of the content of the register (the base) and a constant from the instruction (the displacement); used to access local variables on a stack or data structures. If the displacement is zero, then it is the same as register indirect.

Memory indirect (or memory deferred)

ex: `ADD r1, r2, @r3`

means: `r1` $\leftarrow$ `r2 + M[M[r3]]`

comment: used in pointer addressing; if r3 contains the address of a pointer p, then M[M[r3]] yields *p.



Autoincrement

ex: `ADD r1, r2,(r3)+`

means: `r1 ← r2 + M[r3]`

`r3 ← r3 + s`

comment: used to step through arrays, the first time it is used r3 points to the beginning of the array; each access increments r2 with the size s of an array's element (s = 1 for byte, s = 2 for half-word, etc.)

Autodecrement

ex: `ADD r1, r2, -(r3)`

means: `r3 ← r3 - s`

`r1 ← r2 + M[r3]`

comment: can be used like autoincrement, but to step through arrays in reverse order. Together with the autoincrement mode it can be used to implement a stack.

Encoding an instruction set

Encoding an instruction into binary representation affects,

- The size of the compiled program
- Implementation of the processor

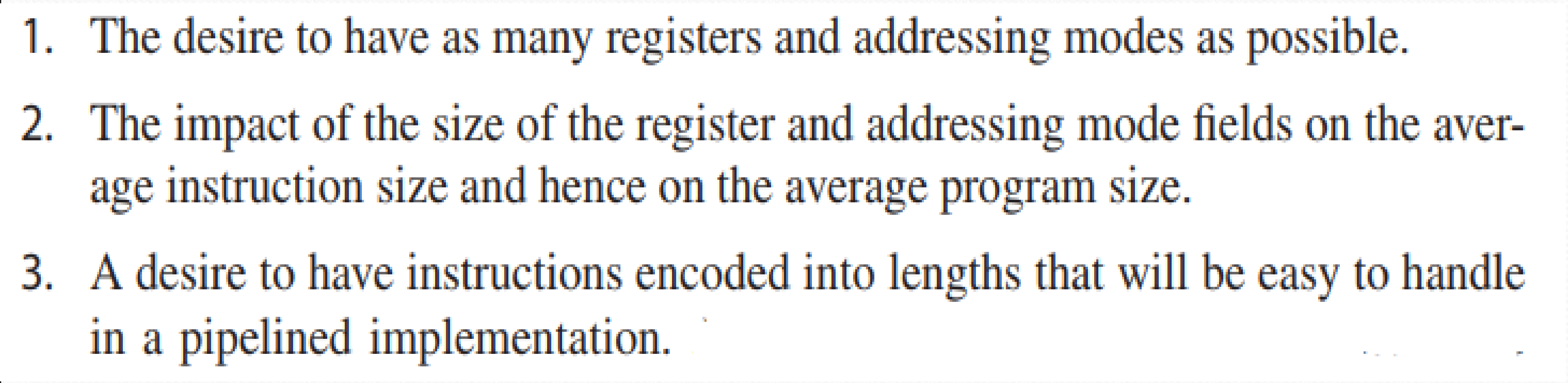
The operation is typically specified in one field, called **opcode**

- The important decision is how to encode the addressing modes with operations.
- The decision depends on the range of addressing modes and the degree of independence between opcodes and modes.

- When encoding the instructions, the number of registers and the number of addressing modes have a significant impact on the size of the instruction.

The computer architect must balance several competing forces when encoding the instruction set:

The computer architect must balance several competing forces when encoding the instruction set:



1. The desire to have as many registers and addressing modes as possible.
2. The impact of the size of the register and addressing mode fields on the average instruction size and hence on the average program size.
3. A desire to have instructions encoded into lengths that will be easy to handle in a pipelined implementation.

Three popular choices for encoding the instruction set:

- **VARIABLE**

It allows virtually all addressing modes to be with all operations.

- **FIXED**

It combines the operation and addressing mode into the opcode.

- **HYBRID**

It has multiple formats specified by the opcode.

Operation and no. of operands	Address specifier 1	Address field 1	• • •	Address specifier *n*	Address field *n*

(a) Variable (e.g., Intel 80x86, VAX)

Operation	Address field 1	Address field 2	Address field 3

(b) Fixed (e.g., Alpha, ARM, MIPS, PowerPC, SPARC, SuperH)

Operation	Address specifier	Address field

Operation	Address specifier 1	Address specifier 2	Address field

Operation	Address specifier	Address field 1	Address field 2

(c) Hybrid (e.g., IBM 360/370, MIPS16, Thumb, TI TMS320C54x)

Variable format

- More complex, harder to decode
- More compact, efficient use of memory
- Variable format is best when there are many addressing modes to be with all operations.
- It generally enables the smallest code representation, as no need to include unused fields.

Fixed format

- Simple, easily decoded
- The fixed format always has the same number of operands , with addressing mode specified as part of the opcode
- Fixed encoding will have only single size for all instructions.
- It works best when there are few addressing modes and operations.
- It generally results in he largest code size.

Variable Vs Fixed

- The architect who interested in **code size** than performance will choose **variable encoding**.
- The architect who is concerned about **performance** than code size will choose **fixed encoding.**