University of Babylon, College of science for women
Dept. of Computer science

Computer Architecture

Second year

Dr. Salah Al-Obaidi

Lecture #4: Memory Addressing Spring 2025

Contents

4. Memory Addressing

Independent of whether the architecture is load-store or allows any operand to be a memory reference, it must define how memory addresses are interpreted and how they are specified.

4.1 Interpreting Memory Addresses

How is a memory address interpreted? That is, what object is accessed as a function of the address and the length? The computers deal with instructions that are byte addressed and provide access for bytes (8 bits), half words (16 bits), and words (32 bits). Most of the computers also provide access to double words (64 bits).

There are two different conventions for ordering the bytes within a larger object:

- **Little Endian** byte order puts the byte whose address is "x . . . x000" at the least significant position in the double word (the little end). The bytes are numbered:

- **Big Endian** byte order puts the byte whose address is "x . . . x000" at the most significant position in the double word (the big end). The bytes are numbered:

When operating within one computer, the byte order is often unnoticeable— only programs that access the same locations as both, say, words and bytes, can notice the difference. Byte order is a problem when exchanging data among computers with different

orderings, however. Little Endian ordering also fails to match the normal ordering of words when strings are compared. Strings appear "**SDRAWKCAB**" (backwards) in the registers.

A second memory issue is that in many computers, accesses to objects larger than a byte must be aligned. An access to an object of size **s** bytes at byte address **A** is aligned if **A mod s = 0**. Figure 4.1 shows the addresses at which an access is aligned or misaligned.

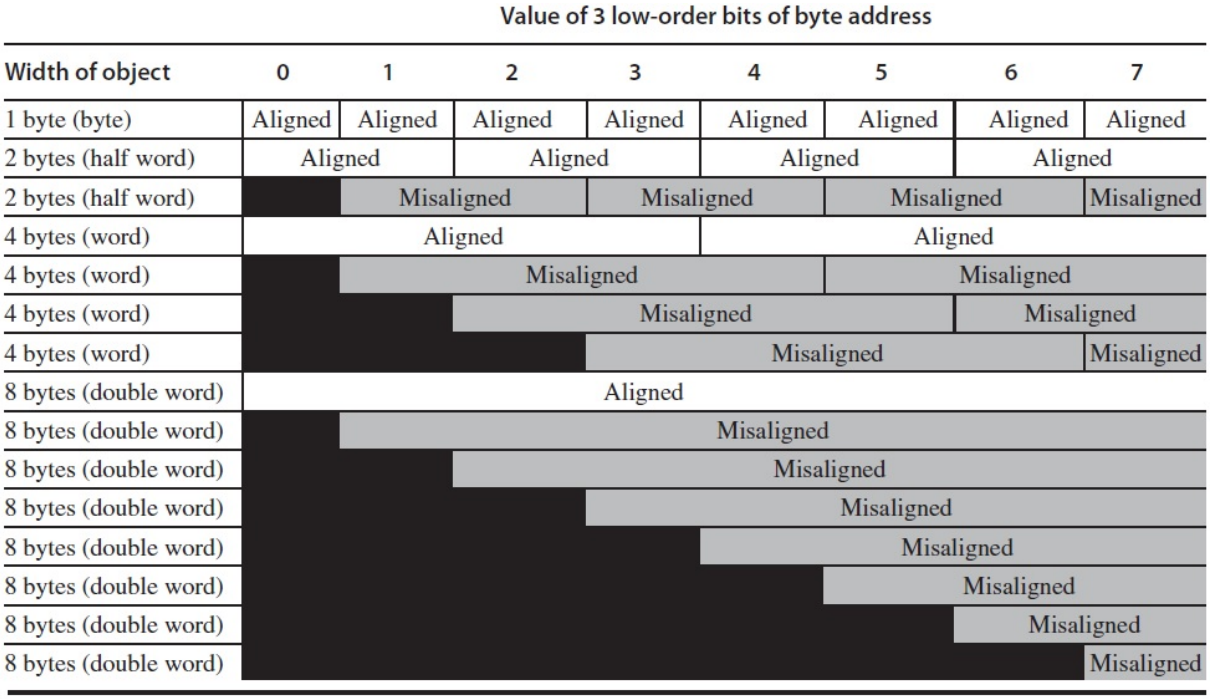

Figure 4.1: **Aligned and misaligned addresses of byte, half-word, word, and double-word objects for byte addressed computers.** For each misaligned example some objects require two memory accesses to complete. Every aligned object can always complete in one memory access, as long as the memory is as 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.

Why would someone design a computer with alignment restrictions and avoid misalignment?:

- Misalignment causes hardware complications, since the memory is typically aligned on a multiple of a word or double-word boundary.
- A misaligned memory access may, therefore, take multiple aligned memory references.

Thus, even in computers that allow misaligned access, programs with aligned accesses run faster.

Now that we have discussed alternative interpretations of memory addresses, we can discuss the ways addresses are specified by instructions, called **addressing modes**.

4.2 Addressing Modes

In this section, we will look at addressing modes—*how architectures specify the address of an object they will access.* **Addressing modes 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.* Figure 4.2 shows all the data addressing modes that have been used in recent computers.

These addressing modes are only useful when the elements being accessed are adjacent in memory. RISC computers use displacement addressing to simulate register indirect with 0 for the address and to simulate direct addressing using 0 in the base register. In our measurements, we use the first name shown for each mode.

Figure 4.2 shows the most common names for the addressing modes, though the names differ among architectures. In this figure, the left arrow (←) is used for assignment. We also use the array Mem as the name for main memory and the array Regs for registers. Thus, **Mem**[**Regs**[**R1**]] refers to the contents of the memory location whose address is given by the contents of register 1 (**R1**).

Addressing modes can significantly reduce instruction counts; they also add to the complexity of building a computer and may increase the average clock cycles per instruction (**CPI**) of computers that implement those modes. Thus, **the usage of various addressing modes is quite important in helping the architect choose what to include**.

4.3 Memory Addressing: More details

The address field or **fields** in a typical instruction format are relatively small. We would like to be able to reference a large range of locations in main memory or, for some

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.

Figure 4.2: **Selection of addressing modes with examples, meaning, and usage.** In autoincrement/-decrement and scaled addressing modes, the variable d designates the size of the data item being accessed (i.e., whether the instruction is accessing 1, 2, 4, or 8 bytes).

systems, virtual memory. To achieve this objective, a variety of addressing techniques has been employed. They all involve some trade-off between *address range* and/or *addressing flexibility*, on the one hand, and *the number of memory references* in the instruction and/or *the complexity of address calculation*, on the other hand.

These modes are illustrated in Figure 4.3. In this section, we use the following notation:

- **A** = contents of an address field in the instruction.
- **R** = contents of an address field in the instruction that refers to a register.
- **EA** = actual (effective) address of the location containing the referenced operand.

- **(X)** = contents of memory location X or register X.

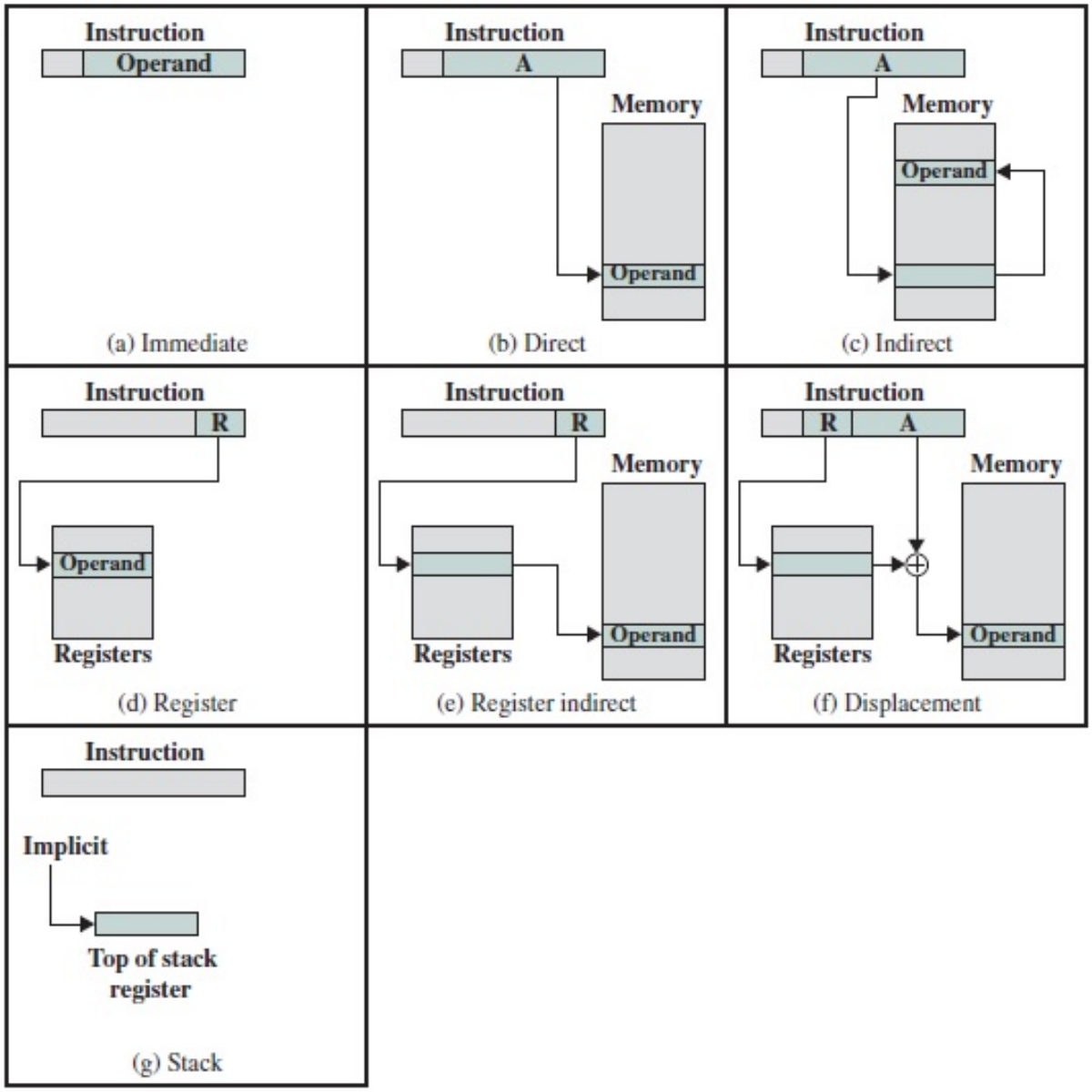

Figure 4.3: Addressing Modes.

Table 4.1 indicates the address calculation performed for each addressing mode. Before beginning this discussion, two comments need to be made. **First**, virtually all computer architectures provide more than one of these addressing modes. The question arises as to how the processor can determine which address mode is being used in a particular instruction. Several approaches are taken. Often, different opcodes will use different

addressing modes. Also, one or more bits in the instruction format can be used as a **mode field**. The value of the mode field determines which addressing mode is to be used.

Table 4.1: Basic Addressing Modes.

Mode	Algorithm	Principal Advantage	Principal Disadvantage
Immediate	Operand = A	No memory reference	Limited operand magnitude
Direct	EA = A	Simple	Limited address space
Indirect	EA = (A)	Large address space	Multiple memory references
Register Operand	LA = R	No memory reference	Limited address space
Register indirect	EA = (R)	Large address space	Extra memory reference
Displacement	LA = (R) + A	Flexibility	Complexity
Stack	EA = top of stack	No memory reference	Limited applicability

The second comment concerns the interpretation of the effective address (EA). In a system without virtual memory, **the effective address** will be either a main memory address or a register. In a virtual memory system, **the effective address** is a virtual address or a register. The actual mapping to a physical address is a function of *the memory management unit* (MMU) and is invisible to the programmer.

4.3.1 Immediate Addressing

The simplest form of addressing is immediate addressing, in which the operand value is present in the instruction

$$Operand = A$$

This mode can be used to define and use constants or set initial values of variables. Typically, the number will be stored in twos complement form; the leftmost bit of the operand field is used as a sign bit.

The advantage of immediate addressing is that *no memory reference other than the instruction fetch is required to obtain the operand, thus saving one memory or cache cycle in the instruction cycle.* The disadvantage is that *the size of the number is restricted to the size of the address field, which, in most instruction sets, is small compared with the word length.*

4.3.2 Direct Addressing

A very simple form of addressing is direct addressing, in which the address field contains the effective address of the operand:

$$EA = A$$

The technique was common in earlier generations of computers but is not common on contemporary architectures. **It requires only one memory reference and no special calculation**. The obvious limitation is that *it provides only a limited address space.*

4.3.3 Indirect Addressing

With direct addressing, the length of the address field is usually less than the word length, thus limiting the address range. One solution is to have the address field refer to the address of a word in memory, which in turn contains a full-length address of the operand. This is known as **indirect addressing**:

$$EA = (A)$$

As defined earlier, the parentheses are to be interpreted as meaning contents of. The obvious advantage of this approach is that *for a word length of* $\boldsymbol{N}$*, an address space of* $\boldsymbol{2^N}$ *is now available.* The disadvantage is that *instruction execution requires two memory references to fetch the operand: one to get its address and a second to get its value.*

Although the number of words that can be addressed is now equal to $\boldsymbol{2^N}$, the number of different effective addresses that may be referenced at any one time is limited to $\boldsymbol{2^K}$, where $\boldsymbol{K}$ is the length of the address field.

4.3.4 Register Addressing

Register addressing is similar to direct addressing. The only difference is that the address field refers to a register rather than a main memory address:

$$EA = R$$

To clarify, if the contents of a register address field in an instruction is 5, then register R5 is the intended address, and the operand value is contained in R5. Typically, an address field that references registers will have from 3 to 5 bits, so that a total of from 8 to 32 general-purpose registers can be referenced.

The advantages of register addressing are that

1. only a small address field is needed in the instruction
2. no time-consuming memory references are required. The memory access time for a register internal to the processor is much less than that for a main memory address.

The disadvantage of register addressing is that *the address space is very limited.*

4.3.5 Register Indirect Addressing

Just as register addressing is analogous to direct addressing, **register indirect addressing** is analogous to indirect addressing. In both cases, the only difference is whether the address field refers to a memory location or a register. Thus, for **register indirect address**,

$$EA = (R)$$

The advantages and limitations of register indirect addressing are basically the same as for indirect addressing. In both cases, the address space limitation (limited range of addresses) of the address field is overcome by having that field refer to a word-length location containing an address. In addition, register indirect addressing uses one less memory reference than indirect addressing.

4.3.6 Displacement Addressing

A very powerful mode of addressing combines the capabilities of direct addressing and register indirect addressing. It is known by a variety of names depending on the context

of its use, but the basic mechanism is the same. We will refer to this as **displacement addressing**:

$$EA = A + (R)$$

Displacement addressing requires that the instruction have two address fields, at least one of which is explicit. The value contained in one address field (**value = A**) is used directly. The other address field, or an implicit reference based on opcode, refers to a register whose contents are added to **A** to produce the effective address.

4.3.7 Stack Addressing

The final addressing mode that we consider is **stack addressing**. A stack is a linear array of locations. It is sometimes referred to as a *pushdown list* or *last-in-first-out queue*. The stack is a reserved block of locations. Items are appended to the top of the stack so that, at any given time, the block is partially filled. Associated with the stack is a pointer whose value is the address of the top of the stack. Alternatively, the top two elements of the stack may be in processor registers, in which case the stack pointer references the third element of the stack. The stack pointer is maintained in a register. Thus, references to stack locations in memory are in fact register indirect addresses.

The stack mode of addressing is a form of *implied addressing*. The machine instructions need not include a memory reference but implicitly operate on the top of the stack.