COM/BLM 376 Computer Architecture

Chapter 13 Instruction Sets: Adddressing Modes and Formats

Asst. Prof. Dr. Gazi Erkan BOSTANCI

ebostanci@ankara.edu.tr

Slides are mainly based on

Computer Organization and Architecture: Designing for Performance by William Stallings, 9th Edition, Prentice Hall

Outline

1. Addressing Modes
2. Instruction Formats

This chapter turns to the question of *how* to specify the operands and operations of instructions. Two issues arise:

- First, how is the address of an operand specified, and
- Second, how are the bits of an instruction organized to define the operand addresses and operation of that instruction?

ADDRESSING MODES

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 systems, virtual memory.

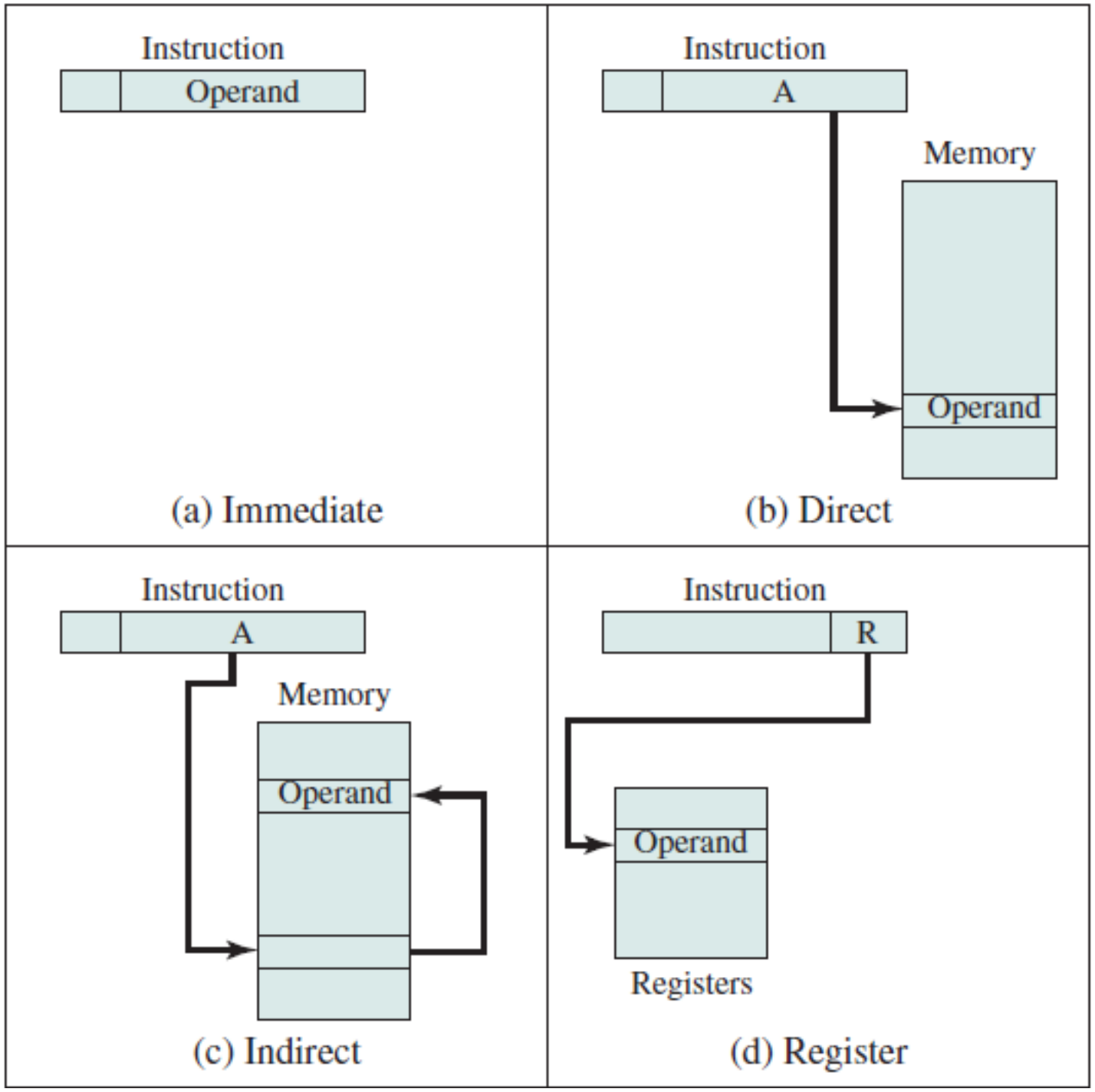
To achieve this objective, a variety of addressing techniques has been employed. They all involve some trade-off between address range and/or addressing flexibility, on the one hand, and the number of memory references in the instruction and/or the complexity of address calculation, on the other.

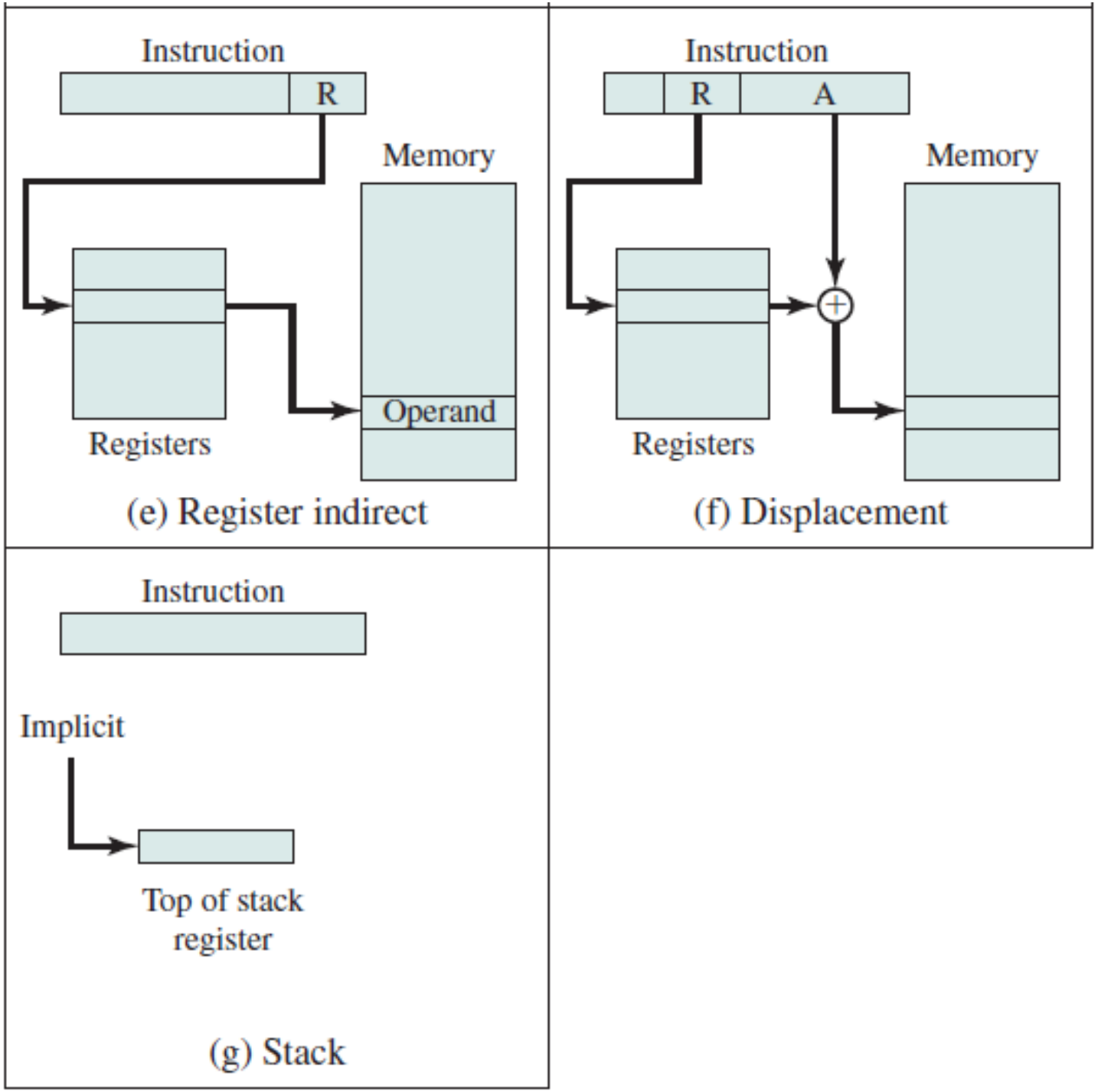
We examine the most common addressing techniques, or modes:

- Immediate
- Direct
- Indirect
- Register
- Register indirect
- Displacement
- Stack

The following notation is used:

- 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




Instruction
R
Memory
Operand
Registers
(e) Register indirect
Instruction
R
A
Memory
Registers
(f) Displacement
Instruction
Implicit
Top of stack
register
(g) Stack

The table indicates the address calculation performed for each addressing mode.

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	EA = R	No memory reference	Limited address space
Register indirect	EA = (R)	Large address space	Extra memory reference
Displacement	EA = A + (R)	Flexibility	Complexity
Stack	EA = top of stack	No memory reference	Limited applicability

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.
- 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.

Immediate Addressing

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

$$\text{Operand} = A$$

This mode can be used to define and use constants or set initial values of variables. Typically, the number will be stored in two's complement form; the leftmost bit of the operand field is used as a sign bit. When the operand is loaded into a data register, the sign bit is extended to the left to the full data **word** size. In some cases, the immediate binary value is interpreted as an unsigned nonnegative integer.

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.

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.

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 N, an address space of 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 2^N, the number of different effective addresses that may be referenced at any one time is limited to 2^K, where K is the length of the address field.

Typically, this is not a burdensome restriction, and it can be an asset. In a virtual memory environment, all the effective address locations can be confined to page 0 of any process. Because the address field of an instruction is small, it will naturally produce low-numbered direct addresses, which would appear in page 0. (The only restriction is that the page size must be greater than or equal to 2^K.)

When a process is active, there will be repeated references to page 0, causing it to remain in real memory. Thus, an indirect memory reference will involve, at most, one page fault rather than two.

A rarely used variant of indirect addressing is multilevel or cascaded indirect addressing:

$$EA = (\ldots(A)\ldots)$$

In this case, one bit of a full-word address is an indirect flag (I). If the I bit is 0, then the word contains the EA. If the I bit is 1, then another level of indirection is invoked.

There does not appear to be any particular advantage to this approach, and its disadvantage is that three or more memory references could be required to fetch an operand.

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.

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.

We will describe three of the most common uses of displacement addressing:

- Relative addressing
- Base-register addressing
- Indexing

RELATIVE ADDRESSING For relative addressing, also called PC-relative addressing, the implicitly referenced register is the program counter (PC).

That is, the next instruction address is added to the address field to produce the EA. Typically, the address field is treated as a two's complement number for this operation. Thus, the effective address is a displacement relative to the address of the instruction.

Relative addressing exploits the concept of locality that was discussed earlier. If most memory references are relatively near to the instruction being executed, then the use of relative addressing saves address bits in the instruction.

BASE-REGISTER ADDRESSING For **base-register addressing**, the interpretation is the following: The referenced register contains a main memory address, and the address field contains a displacement (usually an unsigned integer representation) from that address. The register reference may be explicit or implicit.

Base-register addressing also exploits the locality of memory references. It is a convenient means of implementing segmentation.

In some implementations, a single segment-base register is employed and is used implicitly.

In others, the programmer may choose a register to hold the base address of a segment, and the instruction must reference it explicitly. In this case, if the length of the address field is *K* and the number of possible registers is *N*, then one instruction can reference any one of *N* areas of 2^K words.

INDEXING For indexing, the interpretation is typically the following: The address field references a main memory address, and the referenced register contains a positive displacement from that address. Note that this usage is just the opposite of the interpretation for base-register addressing.

Of course, it is more than just a matter of user interpretation. Because the address field is considered to be a memory address in indexing, it generally contains more bits than an address field in a comparable base-register instruction.

Also, we shall see that there are some refinements to indexing that would not be as useful in the base-register context. Nevertheless, the method of calculating the EA is the same for both base-register addressing and indexing, and in both cases the register reference is sometimes explicit and sometimes implicit (for different processor types).

An important use of indexing is to provide an efficient mechanism for performing iterative operations. Consider, for example, a list of numbers stored starting at location A.

Suppose that we would like to add 1 to each element on the list. We need to fetch each value, add 1 to it, and store it back. The sequence of effective addresses that we need is A, A + 1, A + 2, . . . , up to the last location on the list.

With indexing, this is easily done. The value A is stored in the instruction's address field, and the chosen register, called an *index register*, is initialized to 0. After each operation, the index register is incremented by 1.

Because index registers are commonly used for such iterative tasks, it is typical that there is a need to increment or decrement the index register after each reference to it. Because this is such a common operation, some systems will automatically do this as part of the same instruction cycle. This is known as **autoindexing**.

If certain registers are devoted exclusively to indexing, then autoindexing can be invoked implicitly and automatically. If general-purpose registers are used, the autoindex operation may need to be signaled by a bit in the instruction. Autoindexing using increment can be depicted as follows.

$$EA = A + (R)$$

$$(R) \leftarrow (R) + 1$$

In some machines, both indirect addressing and indexing are provided, and it is possible to employ both in the same instruction. There are two possibilities: the indexing is performed either before or after the indirection.

If indexing is performed after the indirection, it is termed **postindexing**:

$$EA = (A) + (R)$$

First, the contents of the address field are used to access a memory location containing a direct address. This address is then indexed by the register value. This technique is useful for accessing one of a number of blocks of data of a fixed format.

For example, it was described earlier that the operating system needs to employ a process control block for each process. The operations performed are the same regardless of which block is being manipulated.

Thus, the addresses in the instructions that reference the block could point to a location (value = A) containing a variable pointer to the start of a process control block. The index register contains the displacement within the block.

With **preindexing**, the indexing is performed before the indirection:

$$EA = (A + (R))$$

An address is calculated as with simple indexing. In this case, however, the calculated address contains not the operand, but the address of the operand.

An example of the use of this technique is to construct a multiway branch table. At a particular point in a program, there may be a branch to one of a number of locations depending on conditions. A table of addresses can be set up starting at location A. By indexing into this table, the required location can be found.

Typically, an instruction set will not include both preindexing and postindexing.

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.

INSTRUCTION FORMATS

An instruction format defines the layout of the bits of an instruction, in terms of its constituent fields. An instruction format must include an opcode and, implicitly or explicitly, zero or more operands.

The format must, implicitly or explicitly, indicate the addressing mode for each operand. For most instruction sets, more than one instruction format is used.

The design of an instruction format is a complex art, and an amazing variety of designs have been implemented.

Instruction Length

The most basic design issue to be faced is the instruction format length. This decision affects, and is affected by, memory size, memory organization, bus structure, processor complexity, and processor speed. This decision determines the richness and flexibility of the machine as seen by the assembly-language programmer.

The most obvious trade-off here is between the desire for a powerful instruction repertoire and a need to save space. Programmers want more opcodes, more operands, more addressing modes, and greater address range.

- More opcodes and more operands make life easier for the programmer, because shorter programs can be written to accomplish given tasks.
- Similarly, more addressing modes give the programmer greater flexibility in implementing certain functions, such as table manipulations and multiple-way branching.
- And, of course, with the increase in main memory size and the increasing use of virtual memory, programmers want to be able to address larger memory ranges.

All of these things (opcodes, operands, addressing modes, address range) require bits and push in the direction of longer instruction lengths. But longer instruction length may be wasteful. A 64-bit instruction occupies twice the space of a 32-bit instruction but is probably less than twice as useful.

Beyond this basic trade-off, there are other considerations. Either the instruction length should be equal to the memory-transfer length (in a bus system, data bus length) or one should be a multiple of the other. Otherwise, we will not get an integral number of instructions during a fetch cycle.

A related consideration is the memory transfer rate. This rate has not kept up with increases in processor speed. Accordingly, memory can become a bottleneck if the processor can execute instructions faster than it can fetch them.

One solution to this problem is to use cache memory; another is to use shorter instructions. Thus, 16-bit instructions can be fetched at twice the rate of 32-bit instructions but probably can be executed less than twice as rapidly.

A seemingly mundane but nevertheless important feature is that the instruction length should be a multiple of the character length, which is usually 8 bits, and of the length of fixed-point numbers. To see this, we need to make use of that unfortunately ill-defined word, *word*.

The word length of memory is, in some sense, the “natural” unit of organization. The size of a word usually determines the size of fixed-point numbers (usually the two are equal). Word size is also typically equal to, or at least integrally related to, the memory transfer size.

Because a common form of data is character data, we would like a word to store an integral number of characters. Otherwise, there are wasted bits in each word when storing multiple characters, or a character will have to straddle a word boundary.