**Chapter 13: Instruction Sets – Addressing Modes and Formats**

Table of Contents

[13.1 Addressing Modes 4](#_Toc48999487)

[Immediate Addressing 6](#_Toc48999488)

[Direct Addressing 7](#_Toc48999489)

[Indirect Addressing 7](#_Toc48999490)

[Register Addressing 8](#_Toc48999491)

[Register Indirect Addressing 9](#_Toc48999492)

[Displacement Addressing 10](#_Toc48999493)

[Relative Addressing 10](#_Toc48999494)

[Base-Register Addressing 11](#_Toc48999495)

[Indexing 11](#_Toc48999496)

[Stack Addressing 13](#_Toc48999497)

[13.3 Instruction Formats 14](#_Toc48999498)

[Instruction Length 14](#_Toc48999499)

[Allocation of Bits 15](#_Toc48999500)

[PDP-8 17](#_Toc48999501)

[PDP-10 19](#_Toc48999502)

[Variable Length Instructions 20](#_Toc48999503)

[PDP-11 21](#_Toc48999504)

[VAX 22](#_Toc48999505)

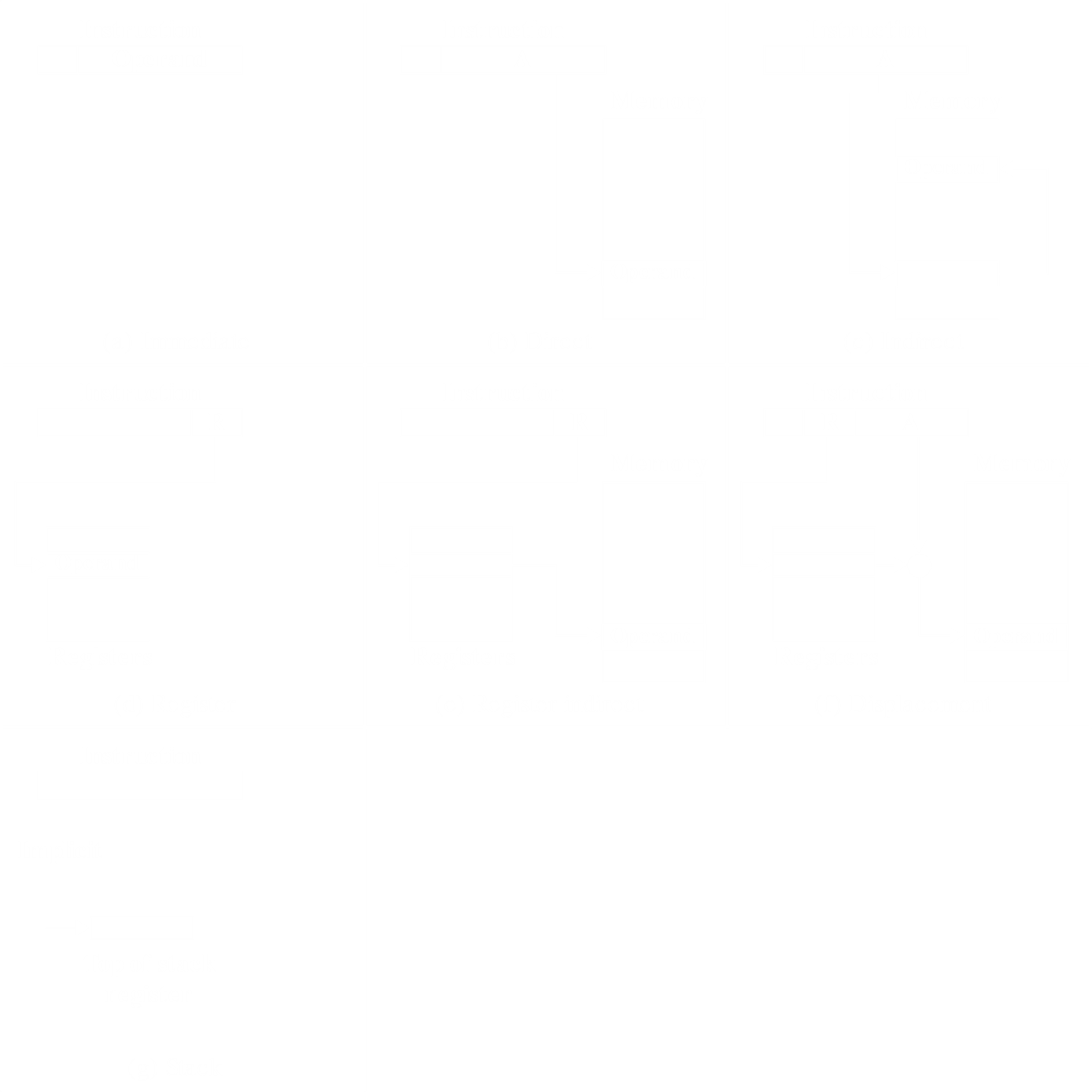
[13.5 Assembly Language 25](#_Toc48999506)

In the previous chapter, we looked at what an instruction does. We looked at the types of operands and operations that may be specified by machine instructions. Here, we will look at how the operands and operations of instructions are specified.

## 13.1 Addressing Modes

The address field or fields in a typical instruction format are relatively small. We want to reference a large range of locations in main memory, or even virtual memory. To achieve this, a variety of addressing techniques are used. All of the techniques have some trade-off between address range and/or addressing flexibility on one hand, and the number of memory refences in the instruction and/or the complexity of address calculation on the other hand. The most common addressing techniques are:

* Immediate
* Direct
* Indirect
* Register
* Register Indirect
* Displacement
* Stack



In the figures above, A is the contents of an address field in the instruction, R is the contents of an address field in the instruction that refers to a register, EA is the actual (effective) address of the location containing the references operand, and X is the contents of the memory location X or register X.

|  |  |  |  |
| --- | --- | --- | --- |
| **Mode** | **Algorithm** | **Principle Advantage** | **Principle 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 refences |
| 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 |

The table above shows the address calculation performed for each addressing mode.

Before beginning, we need to know two things. First, almost all computers have more than one addressing modes. This means the processor needs to determine which addressing mode is being used in a particular instruction. There are several approaches to this. We could use different op-codes for different addressing modes, or use one or more bits in the instruction format as a mode field to determine which addressing mode to use.

The second point is about the interpretation of the effective address (EA). In a system without virtual memory, the effective address is a main memory address or a register. In a system with virtual memory, the effective address is a virtual memory address or a register. The actual mapping to a physical address is done by the memory management unit (MMU) and is not visible to the programmer.

### Immediate Addressing

Immediate addressing is the simplest form of addressing. The value of the operand is present in the instruction. This mode can be used to define and use constant or set initial values for variables. Typically, the number is stored in twos complement form, with the leftmost bit of the operand field being used as a sign bit. When the operand is loaded into a data register, the sign bit is extended to the full data word size. In some cases, the immediate binary value is interpreted as an unsigned non-negative integer.

The advantage of immediate addressing is that no memory reference other than the instruction fetch is needed to retrieve 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 to the word length.

### Direct Addressing

Another simple addressing form is direct addressing, in which the address field contains the effective address of the operand. The technique was common in earlier computers, but is not common now. It needs only one memory reference and no other calculations. The obvious limitation is 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 called indirect addressing.

The use of parentheses indicates ‘contents of’.

The advantage of this approach is that for a word length of is now available. The disadvantage is that the instruction execution requires two memory references to get the operand.

Although the number of words that can be addressed is now , 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 cascade indirect addressing.

In this case, one bit of a full-word address is an indirect flag , then another level of indirection is invoked. There does not seem to be any particular advantage to this approach, and the disadvantage is that there are more memory references.

### Register Addressing

Register addressing is like direct addressing, except that the address field refers to a register rather than a main memory address.

If the content of a register address field is should be checked and the operand value contained there is what we want. Typically, an address field that references registers will have from 3 to 5 bits, so that a total of 8 to 32 general-purpose registers can be referenced.

The advantage of register addressing is that only a small address field is needed in the instruction and no time-consuming memory references are made. The memory access time for a register internal to the processor is much less than that for a main memory address. The disadvantage is that the address space is very limited.

If register addressing is heavily used in an instruction set, this means the processor registers will be heavily used. Since there are very few registers (compared to memory locations), their use in this manner only makes sense if they are employed efficiently. If every operand is brough into a register from memory, operated on just once, and then returned to main memory, then we have a wasteful intermediate step. If the operand in a register is used for multiple operations, then a real savings has been achieved.

The programmer or compiler must decide which values should stay in registers and which should stay in main memory. Most modern processors use multiple general-purpose registers, placing a burden for efficient execution on the assembly language programmer (compiler writer).

### Register Indirect Addressing

Register indirect addressing is like indirect addressing, except that the address field refers to a register instead of a memory location.

The advantages and limitations of register indirect addressing are the same as those for indirect addressing. In both cases, the address space limitation 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.

### Displacement Addressing

Displacement addressing is a very powerful mode of addressing that combines the capabilities of direct addressing and register indirect addressing. It has a few other names depending on context, but the basic mechanism is the same.

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

Three of the most common uses of displacement addressing are:

* Relative Addressing
* Base-Register Addressing
* Indexing

#### Relative Addressing

For relative addressing, or PC-relative addressing, the implicitly referenced register is the program counter (PC). This means the next instruction address is added to the address field to produce the EA. Typically, the address field is treated as a twos complement number. Thus, the effective address is a displacement relative to the address of the instruction.

Relative addressing uses the concept of locality. If most memory references are relatively near to the instruction being executed, the use of relative addressing saves address bits in the instruction.

#### Base-Register Addressing

For base-register addressing, the references register contains a main memory address, and the address field contains a displacement (also an address; 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 way to implement segmentation. In some implementations, a single segment-base register is employed and 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 the latter case, if the length of the address field is words.

#### Indexing

For indexing, the address field references a main memory address, and the referenced register contains a positive displacement from that address. Notice that this usage is the opposite of the base-register addressing. Because the address field is considered to be a memory address, it generally contains more bits than a comparable base-register instruction. There are also some refinements to indexing that would not be as useful in the base-register context. Even so, the method to calculate the EA is the same as for base-register addressing, and in both cases the register reference may be explicit or implicit.

An important use of indexing is to provide an efficient mechanism for performing iterative operations. Suppose we want to add .

Because index registers are commonly used for such iterative tasks, and there is a need to increment or decrement the index register after each reference to it. Some systems do this automatically as part of the instruction cycle in a process called autoindexing. If certain registers are devoted exclusively to indexing, then autoindexing can be invoked implicitly or automatically. If general-purpose registers are used, autoindexing needs to be signalled by a bit in the instruction.

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. Either the indexing is performed before the indirection or after.

If the indexing is performed after the indirection, it is called post-indexing.

First, the contents of the address field are used to access a memory location with a direct address. This address is then indexed by the register value. This is useful for accessing one or a number of blocks of data of a fixed format. For example, the OS 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 address of the instructions that references the block could point to a location containing a variable pointer to the start of a process control block. The index register contains the displacement within the block.

If indexing is performed before the indirection, it is called pre-indexing.

An address is calculated as in simple indexing. However, in this case, the calculated address does not contain 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 stating at a location . By indexing into this table, the required location can be found.

Typically, an instruction set will not have both pre-indexing and post-indexing.

### Stack Addressing

A stack is a linear array of locations. It is a reserved block of locations. Items are appended to the top of the stack, so that at any 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 two top 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 references, but implicitly operate on the top of the stack.

## 13.3 Instruction Formats

An instruction format defines the layout of the bits of an instruction, in terms of its constituent fields. It must have an opcode and, either implicitly or explicitly, zero or more operands. Each explicit operand is referenced using one of the addressing modes we have discussed, which must also be implicitly or explicitly indicated in the instruction format. For most instruction sets, more than one instruction format is used.

### Instruction Length

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

There is a trade-off here between the desire for a powerful instruction arsenal and a need to save space. Having more opcodes, more operands, more addressing modes and larger address ranges makes life easier for programmers, more opcodes and operands means programs written to accomplish a given task will be shorter. More addressing modes means greater flexibility in implementing certain functions, like table manipulation and multiple-way branching. And of course, since main memory size is expanding and virtual memory is being increasingly used, programmers want to address larger memory ranges. However, each of these things require extra bits and make the instruction longer. However, longer instruction can be wasteful. A 64-bit instruction takes up twice the space of a 32-bit instruction, but it unlikely to be twice as useful.

Beyond this, there are other things to consider. Either the instruction must be equal to the memory-transfer length (or the data-bus length in a bus system), or one must be the multiple of the other. Otherwise, we will not be able to retrieve an integral number of instructions during a fetch cycle. Another consideration is the memory transfer rate, which is not as high as processor speeds. This can cause bottlenecks if the processor can execute instructions faster than they can be fetched. One solution is to use cache memory, while another is to use shorter instructions. Thus, 16-bit instructions can be fetched at twice the rate of 32-bit instructions, but are unlikely to be executed twice as fast.

Instruction lengths also need to be multiples of character length, which is usually 8 bits., and of the length of fixed-point numbers. This is due to the word length. The word length of memory is the ‘natural’ unit size of organization, and it determines the size of fixed-point numbers, usually being equal. Word size is also typically equal to, or at least related to, the memory transfer size. Because character data is a very common form of data, we would want a word to store an integral number of characters, since otherwise there would be wasted bits.

### Allocation of Bits

There are complex trade-offs in the decision on how to allocate bits in the chosen instruction format. For a given instruction length, there is a clear trade-off between the number of op-codes and the power of the addressing capability. More op-codes means more bits in the op-code field, leaving less for addressing.

There is an interesting refinement to this trade-off, which is to use variable-length op-codes. This has a minimum op-code length, but for some op-codes, additional operations may be specified with additional bits in the instruction. For a fixed-length instruction, this leaves fewer bits for addressing. Thus, this feature is used for instructions that need fewer operands and/or less powerful addressing.

The following factors go into the decision of how to use addressing bits:

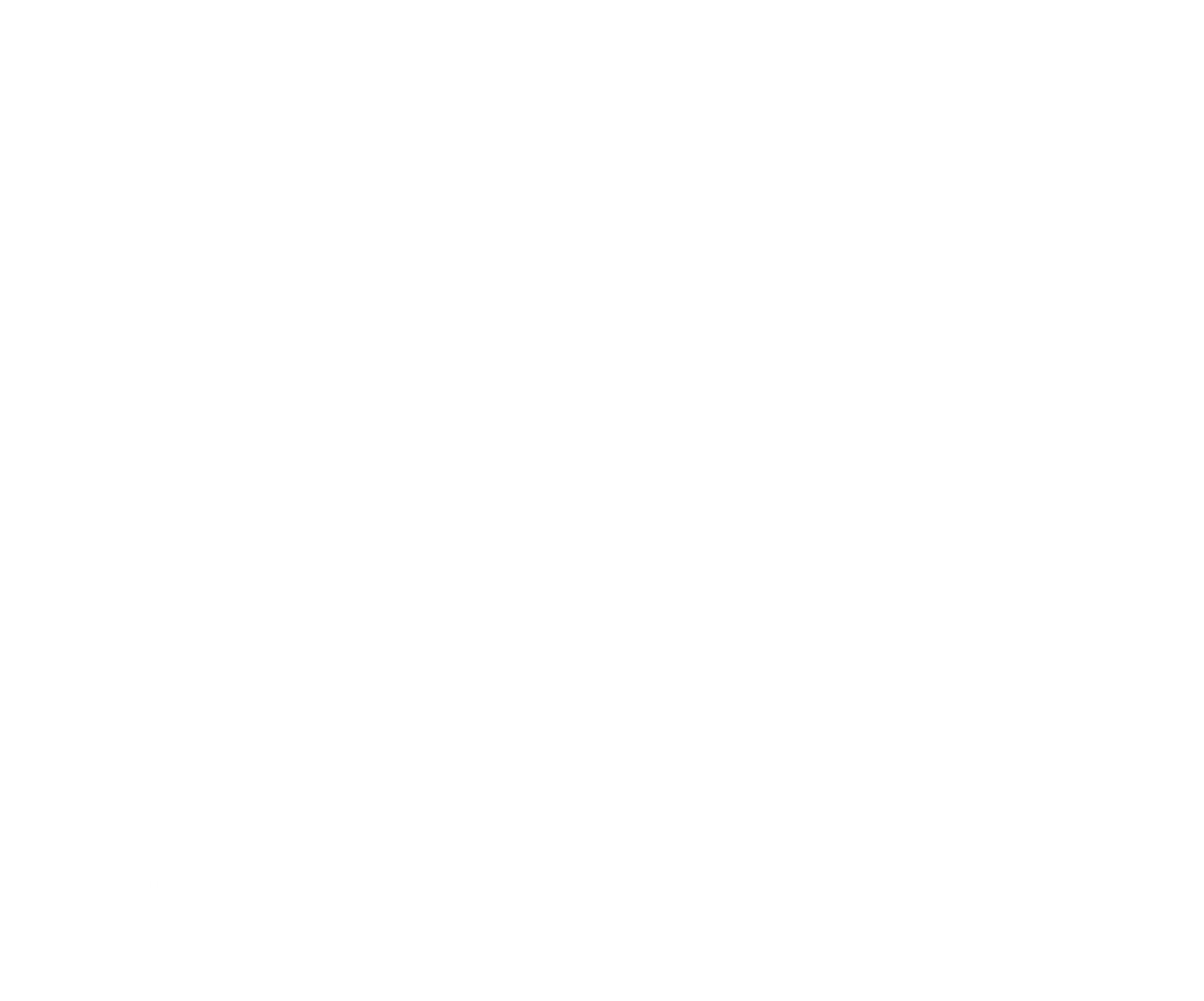
* **Number of Addressing Modes**: Sometimes, an addressing mode can be indicated implicitly. For example, certain op-codes may always use indexing. In other cases, the addressing mode must be explicit, which uses one or more mode bits.
* **Number of Operands**: We have seen that fewer addresses lead to longer, awkward programs. Typical instruction formats on modern machines have two operands. Each operand address may need its own mode indicator, or the mode indicator could be limited to just one of the address fields.
* **Register Versus Memory**: A machine has to have registers so data can be brought into the processor to be processed. With a single, user-visible register (usually called the accumulator), one operand address is implicit and does not use up an instruction bit. Single-register programming is awkward though, and needs many instructions. Even with multiple registers, only a few bits are needed to specify a register. The more that registers can be used to specify operands, the fewer bits are needed. Many studies have concluded that having 8 to 32 user-visible registers is optimal. Fewer is not enough and more does not provide additional benefits.
* **Number of Register Sets**: Most contemporary machines have one set of general-purpose registers, typically with 32 or more registers. These registers can store data or address for displacement addressing. Some architectures have two or more specialized sets. One advantage to this is that fewer bits are use in the instruction since the function of the register is implicit. For example, with two sets of eight registers, only 3 bits are needed to identify a register since the op-code determines which set of registers is being referenced.
* **Address Range**: For addresses that reference memory, the range of addresses that can be referenced is related to the numbed of address bits. Due to the severe limitation this imposes, direct addressing is rarely used. With displacement addressing, the range is increased to the length of the address register. Even then, it is convenient to allow large displacements from the register address, which needs a large number of address bits.
* **Address Granularity**: For addresses that reference memory instead of registers, address granularity becomes important. In a system with 16- or 32-bit words, an address can reference a word or a byte, at the designer’s choice. Byte addressing is better for character manipulation, but needs more address bits, for a fixed-size memory.

The designer must deal with all of these factors and balance them. It is unclear how critical each factor is.

### PDP-8

PDP-8 is one of the simplest instruction designs for a general-purpose computer. It uses 12-bit instructions and operates on 12-bit words. There is a single, general-purpose register, the accumulator.

Despite the limitations of this design, the addressing is pretty flexible. Each memory reference is for 7 bits plus two 1-bit modifiers. The memory is divided into fixed-length pages of are auto index ‘registers’. When an indirect reference is made to one of these locations, pre-indexing occurs.



There are 3-bit opcodes and three types of instructions. Op-codes 0 through 5 use a single address memory reference instruction, with a page bit and an indirect bit. Thus, there are only 6 basic operations. To enlarge the group or operations, op-code 7 defines a register reference or microinstruction. In this format, the remaining bits are used to encode additional operations. In general, each bit defines a specific operation (e.g. clear accumulator) and these bits can be combined in a single instruction. Op-code 6 is the I/O operation where 6 bits are used to select one of 64 devices, and 3 bits are used to specify an I/O command.

The PDP-8 instruction format is very efficient. It supports indirect addressing, displacement addressing and indexing. With the use of op-code extension, approximately 35 instructions are supported. Given the constraints of a 12-bit instruction, the designers did a great job.

### PDP-10

PDP-10 was designed to be a large-scale, time-shared system, with an emphasis on making the system easy to program, even at the cost of additional hardware. Among the design principles used in designing the instruction set were:

* **Orthogonality**: Orthogonality is a principle by which two variables are independent of each other. Thus, the elements of an instruction are not determined by the op-code. This means that the address is always calculated in the same way. This is in contrast to many machines, where the addressing mode sometimes depends implicitly on the operator being used.
* **Completeness**: Each arithmetic data type (integer, fixed-point, floating-point) should have a complete and identical set of operations.
* **Direct Addressing**: Base plus displacement addressing, which places a memory organization burden on the programmer, was avoided in favour of direct addressing.

Each of these principles helps make programming easier.



PDP-10 uses a 36-bit word length and a 36-bit instruction length. The op-code is of 9 bits, allowing up to 512 operations. 365 different instructions are defined. Most instructions have two addresses, one of which is one of the 16 general purpose registers, which needs 4 bits to be referenced. The other operand reference starts with an 18-bit memory address field. This can be used as an immediate operand, or a memory address. In the latter usage, both indexing and indirect addressing is allowed. The same general-purpose registers are also used as index registers.

A 36-bit instruction length is extremely convenient, and the 9-bit op-code is more than adequate. Addressing is also straightforward. An 18-bit address field makes direct addressing desirable, while indirection is used for memory sizes greater than . For programming ease, indexing is also provided for table manipulation and iterative programs. The 18-bit operand field also makes immediate addressing attractive. The designers made a conscious choice to make the task of the programmer or compiler easier at the expense of an inefficient utilization of space. This cannot be counted as a fault in the design.

### Variable Length Instructions

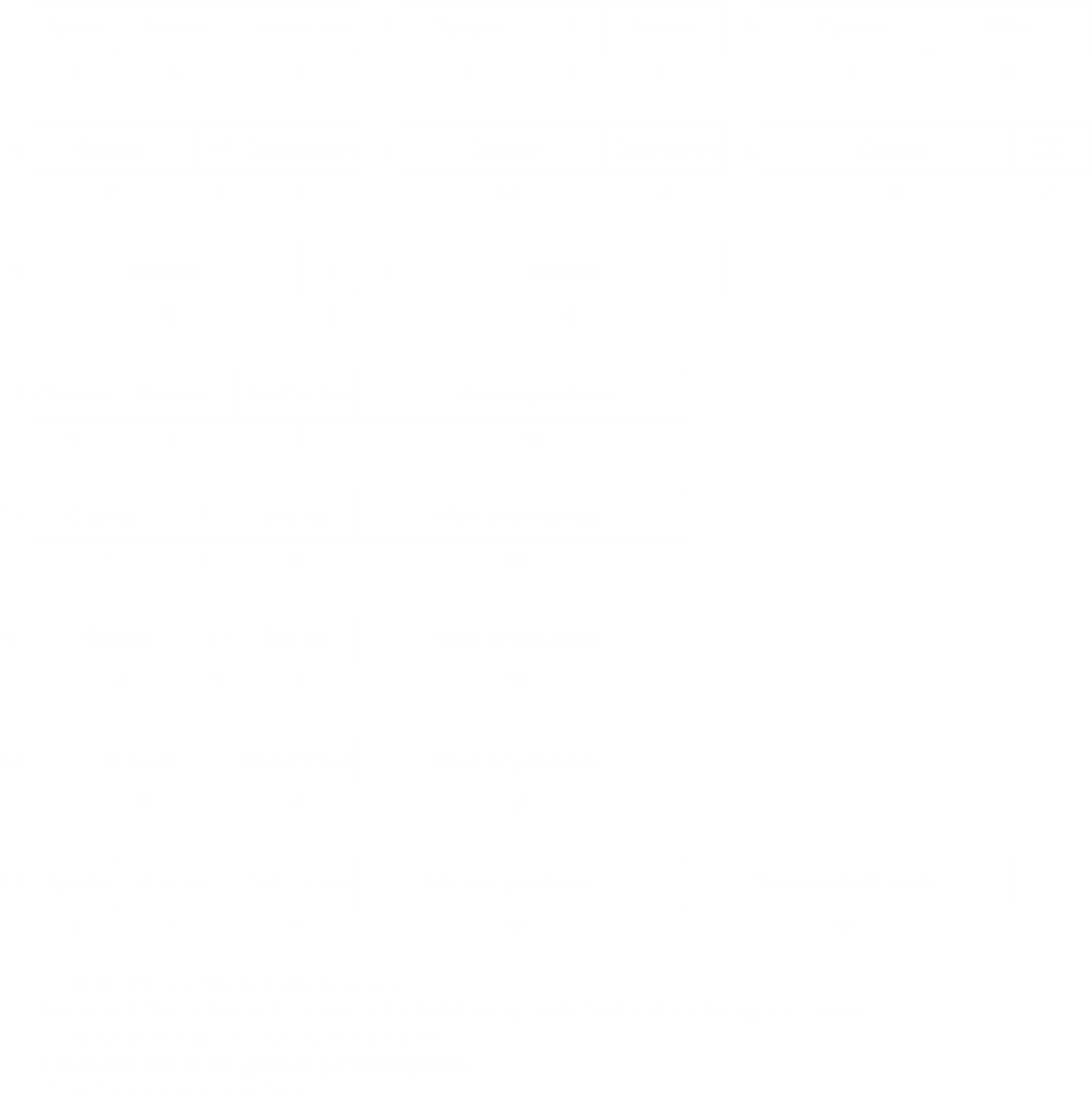
The examples we have seen so far used a single, fixed instruction length. However, the designer may want to provide a variety of instruction formats of different lengths. This makes it easier to provide a large repertoire of op-codes with different op-code lengths. Addressing is also more flexible, with various combinations of register and memory references plus addressing modes. With variable length instructions, these many variations can be provided efficiently and compactly.

The principle price to pay for variable-length instructions is an increase in the complexity of the processor. Falling hardware prices, microprogramming and a general increase in the understanding of processor design makes this easier to deal with. However, RISC and superscalar machines are able to exploit the use of fixed-length instructions to provide improved performance.

Instruction lengths still need to be integrally related to word length. Since the processor does not know the length of the next instruction to be fetched, a typical strategy is to fetch a number of bytes or words equal to at least the longest possible instruction. This means that multiple instructions might be fetched. However, as we will see in the next chapter, this is a good strategy in any case.

#### PDP-11

The PDP-11 was designed to provide a powerful and flexible instruction set within the constraints of a 16-bit minicomputer. It uses a set of 8 16-bit general-purpose registers. One of these is used as a stack pointer for special-purpose stack operations and another is used as the program counter and contains the address of the next instruction.



13 different formats are used, ranging over zero-, one- and two-address instructions. The op-code can be between 4 and 16 bits in length. Register references are 6 bits, 3 to identify the register and 3 to identify the addressing modes. One advantage of linking the addressing mode to the operand instead of the op-code is that any addressing mode can be used with any op-code (orthogonality).

Instructions are usually one word (16 bits) long. One or two memory addresses can be appended for some instructions, leading to 32- and 48-bit instructions. This provides additional flexibility in addressing.

The instruction set and addressing capabilities of PDP-11 are complex, which leads to higher hardware cost and programming complexity. The advantage is that more efficient and compact programs can be developed.

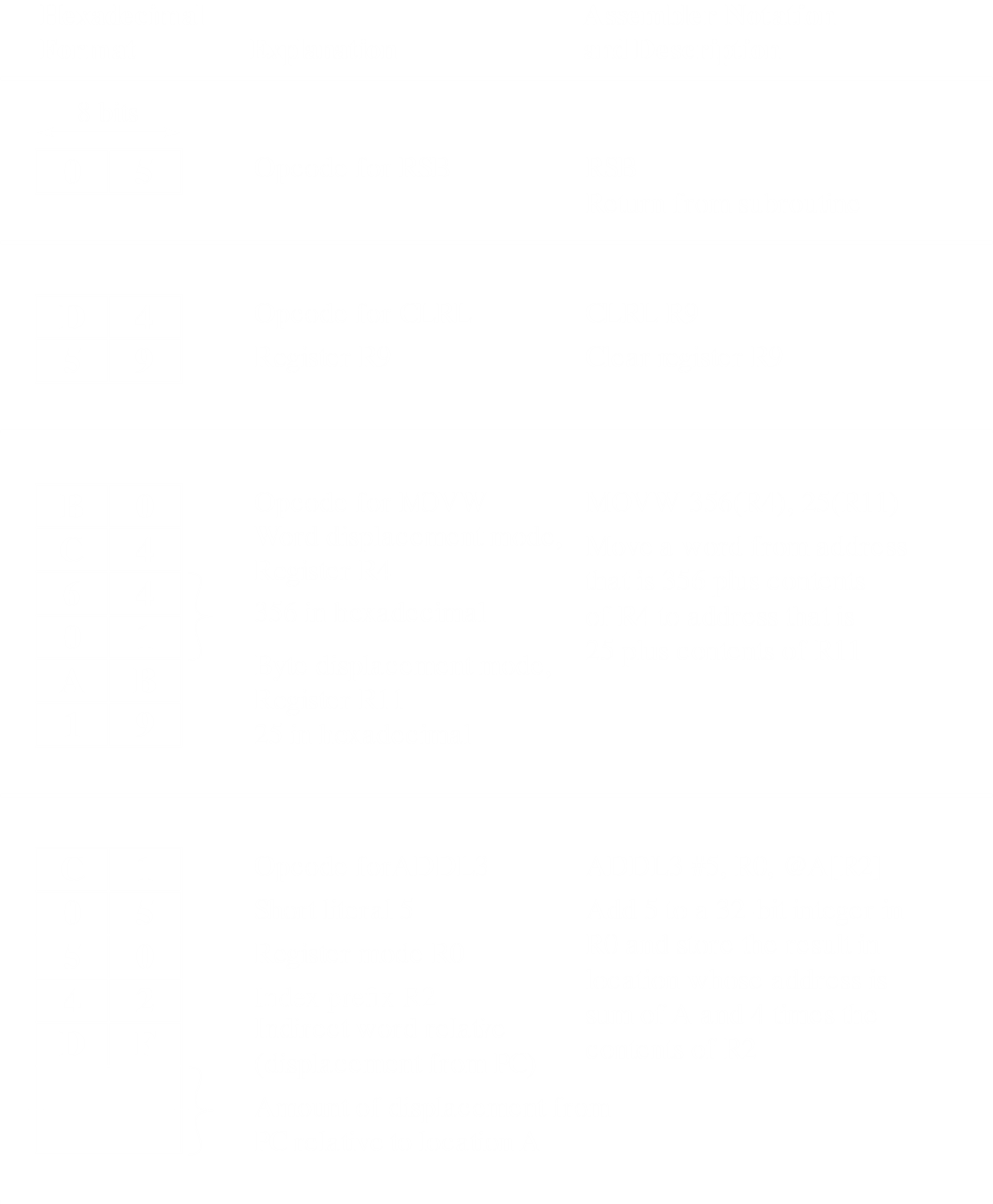
#### VAX

Most architectures provide a relatively small number of fixed instruction formats. This can cause two problems. First, the addressing mode and op-code are not orthogonal. For example, for a given operation, one operand must come from the register and another from memory, or both from registers and so on. Second, only a limited number of operands can be accommodated, typically two or three. Since some operations need more operands, different strategies must be used to achieve the result using more instructions in the process.

To avoid these problems, the VAX instruction format was made following two criteria:

1. All instructions should have the ‘natural’ number of operands
2. All operands should have the same generality in specification

This resulted in a highly variable instruction format. An instruction consists of a 1- or 2-byte op-code and between 0 to 6 operand specifiers, depending on the op-code. Instructions can be from 1 to 37 bytes.



The instruction begins with a 1-byte opcode, which is sufficient for most VAX instructions. However, since there are more than 300 instructions, 8 bits are not enough. The hex codes FD and FF indicate an extended op-code, with the actual op-code being specified in the second byte.

The rest of the instruction consists of up to 6 operand specifiers, each of which is at least 1-byte. The leftmost 4 bits specify the address mode. The only exception is the literal mode, which has 00 in the leftmost 2 bits, leaving 6 bits empty for the literal. Due to this exception, a total of 12 different addressing modes can be specified.

The rightmost 4 bits of the operand specifier specifies one of the 16 general-purpose registers. The length of the operand specifier can be increased in one of two ways. First, a constant value of one or more bytes may immediately follow the first byte of the operand specifier. An example of this is the displacement mode, where 8-, 16- and 32-bit displacement is used. Second, an index mode of addressing could be used. In this case, the first byte of the operand specifier consists of the 4-bit addressing mode code 0100 and a 4-bit index register identifier. The remained of the operand specifier consist of the base address specifier, which may itself be one or more bytes.

It may be surprising that an instruction may need 6 different operands, but VAX has many such instructions. Consider ADDP6 OP1, OP2, OP3, OP4, OP5, OP6. This instruction adds two packed decimal numbers. OP1 and OP2 specify the length and starting address of one decimal string. While OP3 and OP4 specify another. The result of the addition is stored in the decimal string with length and location specified by OP5 and OP6.

The VAX instruction set provides a wide variety of operations and addressing modes. This gives the programmer a very powerful and flexible tool for developing programs. In theory, this should give us efficient machine-language compilations of high-level language programs and, in general, effective and efficient use of processor resources. The penalty for the benefits is the increased complexity of the processor resources.

## 13.5 Assembly Language

A processor can understand machine instructions, which are stored in binary format. If a programmer wanted to write instructions directly in machine language, it would be necessary to do so in binary format.

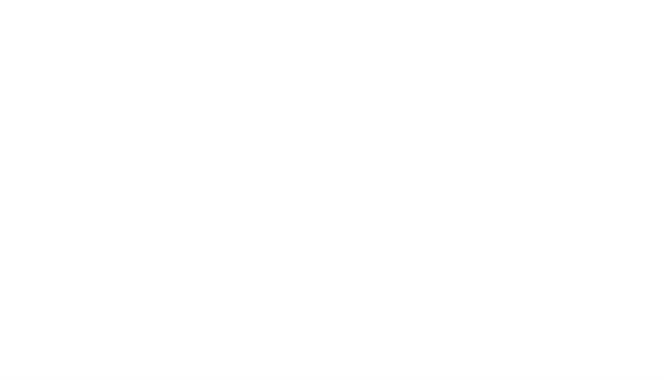
Consider the following statement:

N = I + J + K

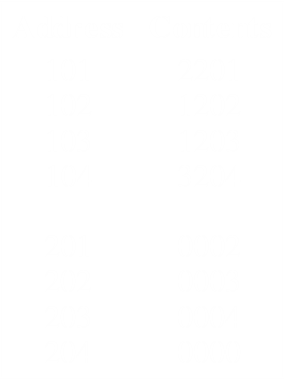
Say we want to execute this statement in machine language and also initialize I, J, and K to 2, 3 and 4 respectively. Say the program starts at location 101 (hexadecimal) and the four variables are stored consecutively starting at 201. The program has 4 instructions:

1. Load the contents of location 201 into the AC
2. Add the contents of location 202 to the AC
3. Add the contents of location 203 to the AC
4. Store the contents of the AC in location 204

This process is shown below:

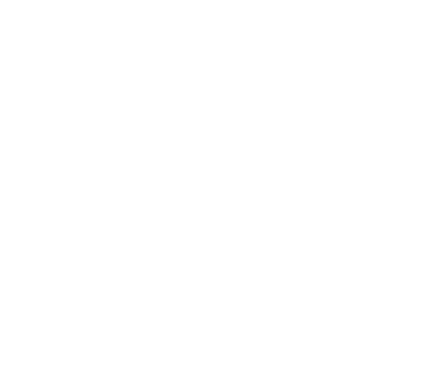


Clearly, this is tedious and error-prone. A slight improvement is to write the program in hexadecimal format rather than binary notation. This is shown below:



Each line contains the address of the memory location and a number indicating the operation. Now we need a program that will accept this as input, translate it into a binary number and run it.

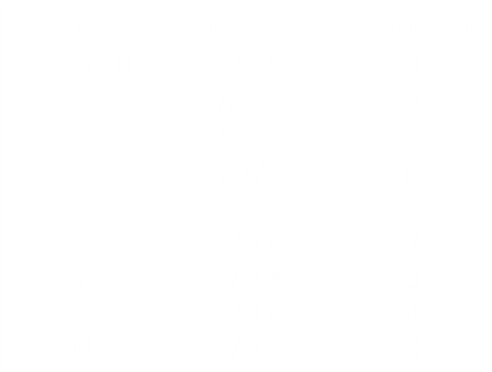
For even more improvement, we could use a symbolic name or mnemonic for each instruction. This results in a symbolic program, shown below.



The three-letter symbol is a representation of the op-code for the instruction. The symbol DAT means that the value is to be stored at this location in memory. This type of input needs an even more complex program to convert the code into binary format.

The symbolic program is convenient, but still awkward. In particular, we need to give an absolute address for each word. This means the program and data can be loaded into only one place in memory, and we must know that place ahead of time. Worse, if we want to insert something in between two pieces of data, the addresses of everything beneath it changes, breaking the program.

A much better system, and one that is commonly used, is to use symbolic address as shown below.



Here, we have used a symbol in place of an actual numeric address. Some lines have no address, implying that its address is one more than the line before it.

This is called an assembly language program. Assembly language programs are translated into machine language by an assembler. The assembler must do the symbolic translation and also assign memory addresses to the symbolic addresses.

Assembly language programs were the first step to the high-level languages that we use now. Although not many programs use assembly language nowadays, almost all machines provide it. If they are used at all, it is for systems programs like compilers and I/O routines.