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

Computer Architecture

Second year

Dr. Salah Al-Obaidi

Lecture #6: Operations in ISA

Spring 2025

Contents

6. Operations in the Instruction Set

The number of different opcodes varies widely from machine to machine. However, the same general types of operations are found on all machines. A useful and typical categorization is the following:

- Data transfer
- Arithmetic
- Logical
- Conversion
- I/O
- System control
- Transfer of control

Figure 6.1 lists common instruction types in each category.

Data Transfer

The most fundamental type of machine instruction is the data transfer instruction. The data transfer instruction must specify several things:

1. **First,** the location of the source and destination operands must be specified. Each location could be memory, a register, or the top of the stack.

Type	Operation Name	Description
Data transfer	Move (transfer)	Transfer word or block from source to destination
	Store	Transfer word from processor to memory
	Load (fetch)	Transfer word from memory to processor
	Exchange	Swap contents of source and destination
	Clear (reset)	Transfer word of 0s to destination
	Set	Transfer word of 1s to destination
	Push	Transfer word from source to top of stack
	Pop	Transfer word from top of stack to destination
Arithmetic	Add	Compute sum of two operands
	Subtract	Compute difference of two operands
	Multiply	Compute product of two operands
	Divide	Compute quotient of two operands
	Absolute	Replace operand by its absolute value
	Negate	Change sign of operand
	Increment	Add 1 to operand
	Decrement	Subtract 1 from operand
Logical	AND	Perform logical AND
	OR	Perform logical OR
	NOT	(complement) Perform logical NOT
	Exclusive-OR	Perform logical XOR
	Test	Test specified condition; set flag(s) based on outcome
	Compare	Make logical or arithmetic comparison of two or more operands; set flag(s) based on outcome
	Set Control Variables	Class of instructions to set controls for protection purposes, interrupt handling, timer control, etc.
	Shift	Left (right) shift operand, introducing constants at end
	Rotate	Left (right) shift operand, with wraparound end
Transfer of control	Jump (branch)	Unconditional transfer; load PC with specified address
	Jump Conditional	Test specified condition; either load PC with specified address or do nothing, based on condition
	Jump to Subroutine	Place current program control information in known location; jump to specified address
	Return	Replace contents of PC and other register from known location
	Execute	Fetch operand from specified location and execute as instruction; do not modify PC
	Skip	Increment PC to skip next instruction
	Skip Conditional	Test specified condition; either skip or do nothing based on condition
	Halt	Stop program execution
	Wait (hold)	Stop program execution; test specified condition repeatedly; resume execution when condition is satisfied
	No operation	No operation is performed, but program execution is continued

Figure 6.1: Common Instruction Set Operations.

2. **Second,** the length of data to be transferred must be indicated.
3. **Third,** as with all instructions with operands, the mode of addressing for each operand must be specified.

In terms of processor action, data transfer operations are perhaps the simplest type. If both source and destination are registers, then the processor simply causes data to be transferred from one register to another; this is an operation internal to the processor. If one or both operands are in memory, then the processor must perform some or all of the following actions:

1. Calculate the memory address, based on the address mode.
2. If the address refers to virtual memory, translate from virtual to real memory address.
3. Determine whether the addressed item is in cache.
4. If not, issue a command to the memory module.

Arithmetic

Most machines provide the basic arithmetic operations of add, subtract, multiply, and divide. These are invariably provided for signed integer (fixed- point) numbers. Often they are also provided for floating-point and packed decimal numbers.

Other possible operations include a variety of single-operand instructions; for example:

- Absolute: Take the absolute value of the operand.
- Negate: Negate the operand.
- Increment: Add 1 to the operand.
- Decrement: Subtract 1 from the operand.

The execution of an arithmetic instruction may involve data transfer operations to position operands for input to the ALU, and to deliver the output of the ALU.

Logical

Most machines also provide a variety of operations for manipulating individual bits of a word or other addressable units, often referred to as "**bit twiddling**". They are based upon Boolean operations.

Some of the basic logical operations that can be performed on Boolean or binary data are shown in Table 6.1. The NOT operation inverts a bit. AND, OR, and Exclusive-OR (XOR) are the most common logical functions with two operands. EQUAL is a useful binary test.

Table 6.1: Basic Logical Operations.

P	Q	NOT P	P AND Q	P OR Q	P XOR Q	P=Q
0	0	1	0	0	0	1
0	1	1	0	1	1	0
1	0	0	0	1	1	0
1	1	0	1	1	0	1

These logical operations can be applied bitwise to n-bit logical data units. Thus, if two registers contain the data

(R1) = 10100101

(R2) = 00001111

then

(R1) AND (R2) = 00000101

As another example, if two registers contain (R1) = 10100101

(R2) = 11111111

then

(R1) XOR (R2) = 01011010

With one word set to all 1s, the XOR operation inverts all of the bits in the other word (ones complement).

In addition to bitwise logical operations, most machines provide a variety of shifting and rotating functions. The most basic operations are illustrated in Figure 6.2. With a **logical shift**, the bits of a word are shifted left or right. On one end, the bit shifted out

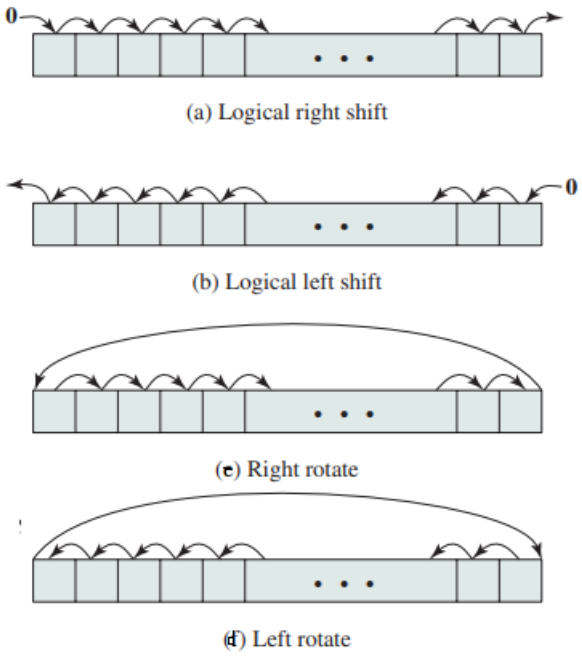

Figure 6.2: Shift and Rotate Operations.

is lost. On the other end, a 0 is shifted in. Logical shifts are useful primarily for isolating fields within a word. The 0s that are shifted into a word displace unwanted information that is shifted off the other end.

Conversion

Conversion instructions are those that change the format or operate on the format of data. An example is converting from decimal to binary. An example of a more complex editing instruction is the EAS/390 Translate (TR) instruction. This instruction can be used to convert from one 8-bit code to another, and it takes three operands:

TR R1 (L), R2

The operand R2 contains the address of the start of a table of 8-bit codes. The **L** bytes starting at the address specified in **R1** are translated, each byte being replaced by the contents of a table entry indexed by that byte.

Input/Output

There are a variety of Input/output instructions, including isolated programmed I/O, memory-mapped programmed I/O, DMA, and the use of an I/O processor. Many implementations provide only a few I/O instructions, with the specific actions specified by parameters, codes, or command words.

System Control

System control instructions are those that can be executed only while the processor is in a certain privileged state or is executing a program in a special privileged area of memory. Typically, these instructions are reserved for the use of the operating system.

Transfer of Control

For all of the operation types discussed so far, the next instruction to be performed is the one that immediately follows, in memory, the current instruction. However, a significant fraction of the instructions in any program have as their function changing the sequence of instruction execution. For these instructions, the operation performed by the processor is to update the program counter to contain the address of some instruction in memory.

There are a number of reasons why transfer- of- control operations are required. Among the most important are the following:

1. In the practical use of computers, it is essential to be able to execute each instruction more than once and perhaps many thousands of times. It may require thousands or perhaps millions of instructions to implement an application. This would be unthinkable if each instruction had to be written out separately. If a table or a list of items is to be processed, a program loop is needed. One sequence of instructions is executed repeatedly to process all the data.

2. Virtually all programs involve some decision making. We would like the computer to do one thing if one condition holds, and another thing if another condition holds.

For example, a sequence of instructions computes the square root of a number. At the start of the sequence, the sign of the number is tested. If the number is negative, the computation is not performed, but an error condition is reported.

3. To compose correctly a large or even medium- size computer program is an exceedingly difficult task. It helps if there are mechanisms for breaking the task up into smaller pieces that can be worked on one at a time.

We can distinguish four different types of transfer of control operations found in instruction sets:

- branches
- Skips
- Procedure calls
- Procedure returns

BRANCH INSTRUCTIONS: A branch instruction, also called a jump instruction, has as one of its operands the address of the next instruction to be executed. Most often, the instruction is a **conditional branch** instruction. That is, the branch is made (update program counter to equal address specified in operand) only if a certain condition is met. Otherwise, the next instruction in sequence is executed (increment program counter as usual). A branch instruction in which the branch is always taken is an **unconditional branch**.

SKIP INSTRUCTIONS: Another form of transfer- of- control instruction is the skip instruction. The skip instruction includes an implied address. Typically, the skip implies that one instruction be skipped; thus, the implied address equals the address of the next instruction plus one instruction length. Because the skip instruction does not require a destination address field, it is free to do other things. A typical example is the increment-and-skip-if-zero (ISZ) instruction. Consider the following program fragment:

```
301
.
.
.
309 ISZ R1
310 BR 301
311
```

In this fragment, the two transfer- of- control instructions are used to implement an iterative loop. R1 is set with the negative of the number of iterations to be performed. At the end of the loop, R1 is incremented. If it is not 0, the program branches back to the beginning of the loop. Otherwise, the branch is skipped, and the program continues with the next instruction after the end of the loop.

PROCEDURE CALL INSTRUCTIONS: Perhaps the most important innovation in the development of programming languages is the procedure. A procedure is a self contained computer program that is incorporated into a larger program. At any point in the program the procedure may be invoked, or called. The processor is instructed to go and execute the entire procedure and then return to the point from which the call took place.

The procedure mechanism involves two basic instructions: a call instruction that branches from the present location to the procedure, and a return instruction that returns from the procedure to the place from which it was called. Both of these are forms of branching instructions.

Figure 6.3 illustrates the use of procedures to construct a program. In this example, there is a main program starting at location 4000. This program includes a call to procedure PROC1, starting at location 4500. When this call instruction is encountered, the processor suspends execution of the main program and begins execution of PROC1 by fetching the next instruction from location 4500. Within PROC1, there are two calls to PROC2 at location 4800. In each case, the execution of PROC1 is suspended and PROC2 is executed. The RETURN statement causes the processor to go back to the calling program and continue execution at the instruction after the corresponding CALL instruction.

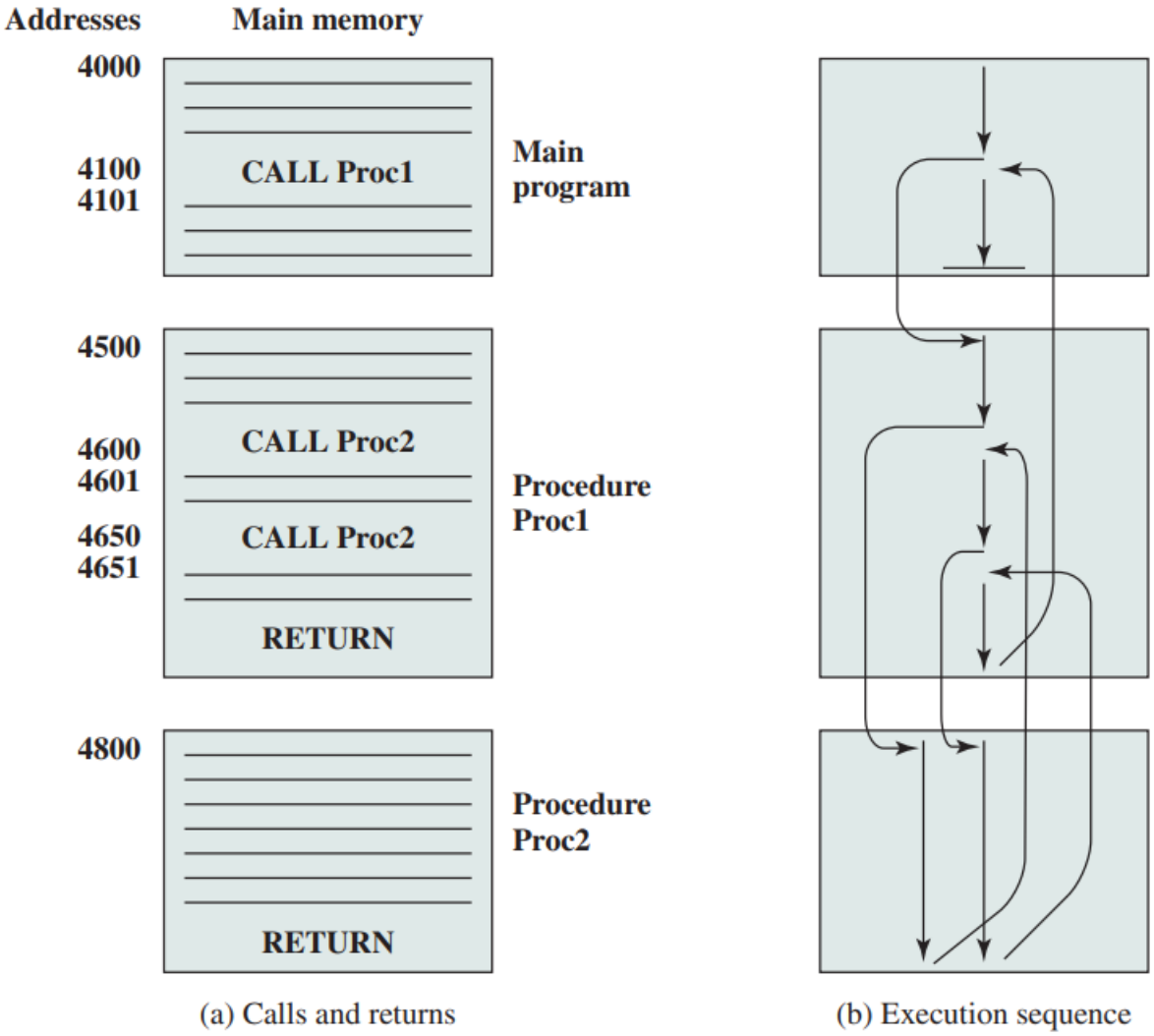

Figure 6.3: Nested Procedures .

7. Encoding an Instruction Set

Clearly, the choice of operations will affect how the instructions are encoded into a binary representation for execution by the processor. This representation affects not only the size of the compiled program; it affects the implementation of the processor, which must decode this representation to quickly find the operation and its operands. The operation is typically specified in one field, called the ***opcode***. An instruction format must include an opcode and, implicitly or explicitly, zero or more operands. Each explicit operand is referenced using one of the addressing modes. The format must, implicitly or explicitly, indicate the addressing mode for each operand. For most instruction sets, more than one instruction format is used. As we shall see, the important decision is how to encode the addressing modes with the operations.

This decision depends on the range of addressing modes and the degree of independence between opcodes and modes. Some older computers have one to five operands with 10 addressing modes for each operand. For such a large number of combinations, typically a separate ***address specifier*** is needed for each operand: The address specifier tells what addressing mode is used to access the operand. At the other extreme are load-store computers with only one memory operand and only one or two addressing modes; obviously, in this case, the addressing mode can be encoded as part of the opcode.

When encoding the instructions, the number of registers and the number of addressing modes both have a significant impact on the size of instructions, as the register field and addressing mode field may appear many times in a single instruction. In fact, for most instructions many more bits are consumed in encoding addressing modes and register fields than in specifying the opcode. The 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. As a minimum, the architect wants instructions to be in multiples of bytes, rather than an arbitrary bit length. Many desktop and server architects have chosen to use a fixed-length instruction to gain implementation benefits while sacrificing average code size.

Figure 7.1 shows three popular choices for encoding the instruction set:

- The first we call ***variable***, since it allows virtually all addressing modes to be with all operations. This style is best when there are many addressing modes and operations.

- The second choice we call ***fixed***, since it combines the operation and the addressing mode into the opcode. Often fixed encoding will have only a single size for all instructions; it works best when there are few addressing modes and operations. The trade-off between variable encoding and fixed encoding is size of programs versus ease of decoding in the processor. Variable tries to use as few bits as possible to represent the program, but individual instructions can vary widely in both size and the amount of work to be performed. Let's look at an 80x86 instruction to see an example of the variable encoding:

  add EAX,1000(EBX)

  The name add means a 32-bit integer add instruction with two operands, and this opcode takes 1 byte. An 80x86 address specifier is 1 or 2 bytes, specifying the source/destination register (EAX) and the addressing mode (displacement in this case) and base register (EBX) for the second operand. This combination takes 1 byte to specify the operands. When in 32-bit mode, the size of the address field is either 1 byte or 4 bytes. Since 1000 is bigger than 2^8, the total length of the instruction is

$1 + 1 + 4 = 6$ bytes

The length of 80x86 instructions varies between 1 and 17 bytes. 80x86 programs are generally smaller than the RISC architectures, which use fixed formats.

- The third alternative immediately springs to mind: Reduce the variability in size and work of the variable architecture but provide multiple instruction lengths to reduce code size. This ***hybrid*** approach is the third encoding alternative.

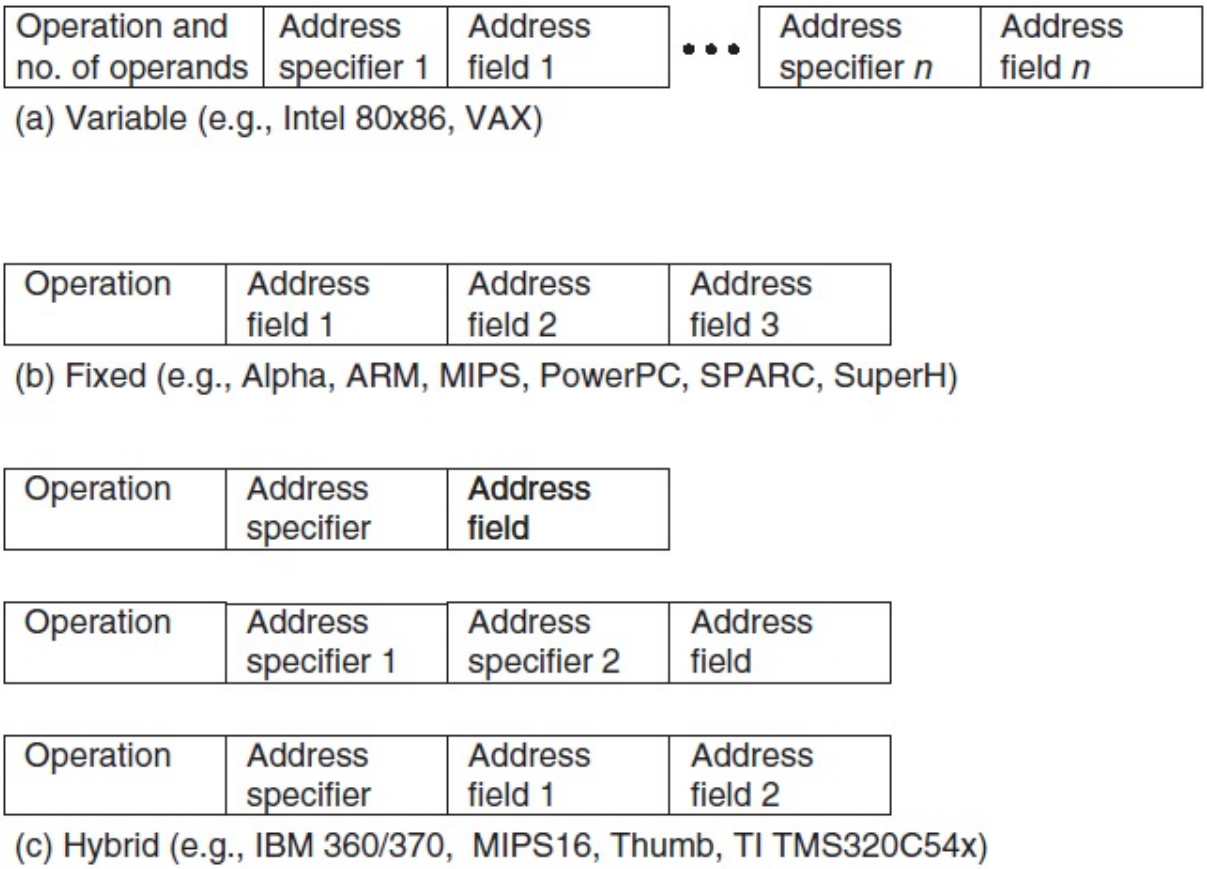

Figure 7.1: Three basic variations in instruction encoding: variable length, fixed length, and hybrid. The variable format can support any number of operands, with each address specifier determining the addressing mode and the length of the specifier for that operand. It generally enables the smallest code representation, since unused fields need not be included. The fixed format always has the same number of operands, with the addressing modes (if options exist) specified as part of the opcode. It generally results in the largest code size. Although the fields tend not to vary in their location, they will be used for different purposes by different instructions. The hybrid approach has multiple formats specified by the opcode, adding one or two fields to specify the addressing mode and one or two fields to specify the operand address.