

ENCODING OF MACHINE INSTRUCTIONS

We have introduced a variety of useful instructions and addressing modes. These instructions specify the actions that must be performed by the processor circuitry to carry out the desired tasks. We have often referred to them as machine instructions. Actually, the form in which we have presented the instructions is indicative of the form used in assembly languages, except that we tried to avoid using acronyms for the various operations, which are awkward to memorize and are likely to be specific to a particular commercial processor. To be executed in a processor, an instruction must be encoded in a compact binary pattern. Such encoded instructions are properly referred to as machine instructions. The instructions that use symbolic names and acronyms are called assembly language instructions, which are converted into the machine instructions using the assembler program.

We have seen instructions that perform operations such as add, subtract, move, shift, rotate, and branch. These instructions may use operands of different sizes, such as 32-bit and 8-bit numbers or 8-bit ASCII-encoded characters. The type of operation that is to be performed and the type of operands used may be specified using an encoded binary pattern referred to as the OP code for the given instruction. Suppose that 8 bits are allocated for this purpose, giving 256 possibilities for specifying different instructions. This leaves 24 bits to specify the rest of the required information.

Let us examine some typical cases. The instruction

Add R1, R2

Has to specify the registers R1 and R2, in addition to the OP code. If the processor has 16 registers, then four bits are needed to identify each register. Additional bits are needed to indicate that the Register addressing mode is used for each operand.

The instruction

Move 24(R0), R5

Requires 16 bits to denote the OP code and the two registers, and some bits to express that the source operand uses the Index addressing mode and that the index value is 24.

The shift instruction

LShiftR #2, R0

And the move instruction

Move #\$3A, R1

Have to indicate the immediate values 2 and #\$3A, respectively, in addition to the 18 bits used to specify the OP code, the addressing modes, and the register. This limits the size of the immediate operand to what is expressible in 14 bits.

Consider next the branch instruction

Branch >0 LOOP

Again, 8 bits are used for the OP code, leaving 24 bits to specify the branch offset. Since the offset is a 2's-complement number, the branch target address must be within 223 bytes of the location of the branch instruction. To branch to an instruction outside this range, a different addressing mode has to be used, such as Absolute or Register Indirect. Branch instructions that use these modes are usually called Jump instructions.

In all these examples, the instructions can be encoded in a 32-bit word. Depicts a possible format. There is an 8-bit Op-code field and two 7-bit fields for specifying the source and destination operands. The 7-bit field identifies the addressing mode and the register involved (if any). The "Other info" field allows us to specify the additional information that may be needed, such as an index value or an immediate operand.

But, what happens if we want to specify a memory operand using the Absolute addressing mode? The instruction

Move R2, LOC

(a) One-word instruction

Opcode	Source	Dest	Other info
--------	--------	------	------------

(b) Two-Word instruction

Opcode	Source	Dest	Other info
Memory address/Immediate operand			

(c) Three-operand instruction

Op code	Ri	Rj	Rk	Other info
---------	----	----	----	------------

Requires 18 bits to denote the OP code, the addressing modes, and the register. This leaves 14 bits to express the address that corresponds to LOC, which is clearly insufficient.

And # $\$FF000000$. R2

In which case the second word gives a full 32-bit immediate operand.

If we want to allow an instruction in which two operands can be specified using the Absolute addressing mode, for example

Move LOC1, LOC2

Then it becomes necessary to use two additional words for the 32-bit addresses of the operands.

This approach results in instructions of variable length, dependent on the number of operands and the type of addressing modes used. Using multiple words, we can implement quite complex instructions, closely resembling operations in high-level programming languages. The term complex instruction set computer (CISC) has been used to refer to processors that use instruction sets of this type.

The restriction that an instruction must occupy only one word has led to a style of computers that have become known as reduced instruction set computer (RISC). The RISC approach introduced other restrictions, such as that all manipulation of data must be done on operands that are already in processor registers. This restriction means that the above addition would need a two-instruction sequence

Move (R3), R1
Add R1, R2

If the Add instruction only has to specify the two registers, it will need just a portion of a 32-bit word. So, we may provide a more powerful instruction that uses three operands

Add R1, R2, R3

Which performs the operation

$R3 \leftarrow [R1] + [R2]$

A possible format for such an instruction is shown in fig c. Of course, the processor has to be able to deal with such three-operand instructions. In an instruction set where all arithmetic and logical operations use only register operands, the only memory references are made to load/store the operands into/from the processor registers.

RISC-type instruction sets typically have fewer and less complex instructions than CISC-type sets. We will discuss the relative merits of RISC and CISC approaches in Chapter 8, which deals with the details of processor design.