2

2 Programs: Instructions in the Computer

Figure 2.1 illustrates the first few processing steps taken as a simple CPU executes a program. The CPU for this example is assumed to have a program counter (PC), an instruction register (IR), a single data register (R0), and a flags register (a few of whose constituent bits are shown individually). Instructions and data values each take up one "word"; addresses are the addresses of words (not of constituent bytes within words).

The program, located in memory starting in word 0, calculates $\sum_{n=1}^{n=3} n$.

memory location		*contents*	
0		LOAD	N
1		COMPARE	3
2		JUMP_IF_GREATER TO	9
3		ADD	SUM
4		STORE	SUM
5		LOAD	N
6		ADD	1
7		STORE	N
8		GOTO	0
9		STOP	
10	N	1	
11	SUM	0	

The program consists of a sequence of instructions occupying memory words 0–9; the data values are stored in the next two memory locations. The data locations are initialized before the program starts.

The calculation is done with a loop. The loop starts by loading the current value of `N`. (Because the imaginary CPU only has one register, instructions like the "load" don't need to specify which register!) Once loaded, the value from variable `N` is checked to see whether it exceeds the required limit (here, 3); the comparison instruction would set one of the "less than" (LT), "equal" (EQ), or "greater than" (GT) bits in the flags register.

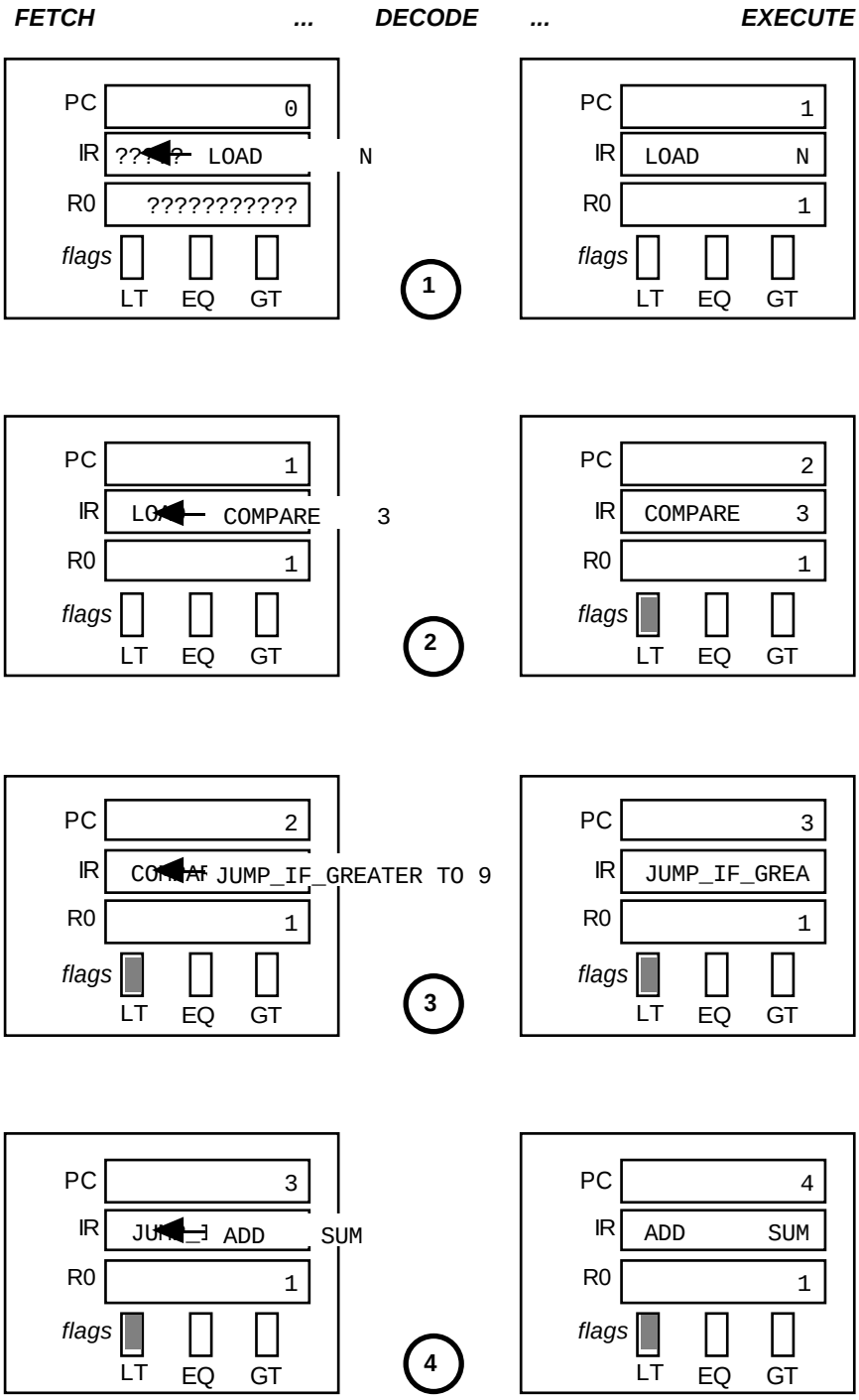


Figure 2.1 Executing a program.

The next instruction, the conditional jump in memory word 2, can cause a transfer to the `STOP` instruction following the loop and located in memory word 9. The jump will occur if the GT flag is set; otherwise the CPU would continue as normal with the next instructions in sequence. These instructions, in words 3 and 4, add the value of `N` into the accumulating total (held in `SUM`). The value of `N` is then incremented by 1 (instructions in words 5-7). Instruction 8 causes the CPU to reset the program counter so that it starts again at instruction 0.

The first few steps are shown in Figure 2.1. Step 1 illustrates the fetch and decode steps for the first instruction, loading value from `N` (i.e. its initial value of 1). As each step is completed, the PC is incremented to hold the address of the next instruction. Steps 2, 3, and 4 illustrate the next few instruction cycles.

If you could see inside a computer (or, at least, a simulator for a computer), this is the sort of process that you would observe.

Of course, if you could really see inside a computer as it ran, you wouldn't see "instructions" written out with names like `COMPARE` or `LOAD`, you wouldn't even see decimal numbers like 3 or 9. All that you would be able to see would be "bit-patterns" --- the sequences of '1's and '0's in the words of memory or the registers of the CPU. So it would be something a little more like illustration Figure 2.2.

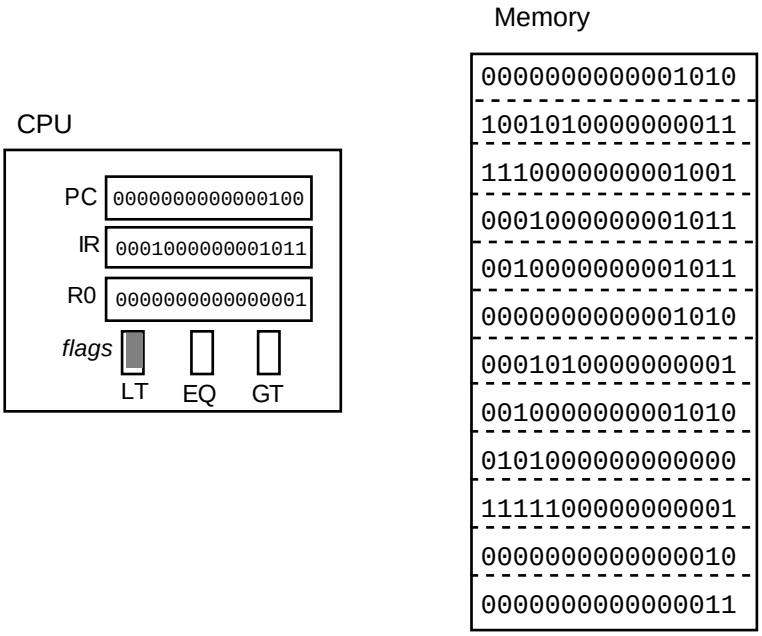


Figure 2.2 View into the machine.

In order to be executed by a computer, a program must end up as a sequence of instructions represented by appropriate bit patterns in the memory of the computer.

There is a very big gap between a statement of some problem that is to be solved by a computer program and the sequence of instruction bit patterns, and data

bit patterns, that must be placed in the computer's memory so that they can be processed by the CPU.

You will be given a problem statement ---

> *"Get the computer to draw a tyrannosaurus rex chasing some corythosaurus plant eating dinosaurs." (Jurassic Park movie)*
> *"Program the computer to apply some rules to determine which bacterium caused this patient's meningitis." (Mycin diagnostic program)*
> *"Write a program that monitor's the Voice of America newswire and tells me about any news items that will interest me."*

and you have to compose an instruction sequence. Hard work, but that is programming.

2.1 PROGRAMMING WITH BITS!

On the very first computers, in the late 1940s, programmers did end up deciding exactly what bit patterns would have to be placed in each word in the memory of their computer!

The programs that were being written were not that complex. They typically involved something like evaluating some equation from physics (one of the first uses of computers was calculating range tables for guns). You may remember such formulae from your studies of physics at school --- for example there is a formula for calculating the speed of an object undergoing uniform acceleration

```
v = speed at time t,   u = initial speed,
       a = acceleration, t = time

 v = u +  a * t
```

(symbol `*` is used to indicate multiplication)

You can see that it wouldn't be too hard to compose a loop of instructions, like the loop illustrated at the start this chapter, that worked out `v` for a number of values of `t`.

The programmer would write out the instruction sequence in rough on paper ...

```
...
load          t
multiply      a
add           u
store         v
...
```

Then it would be necessary to chose which memory word was going to hold each instruction and each data element, noting these address in a table:

```
start of loop @ location 108
end of loop @ location 120
variable t @ 156
```

Given this information it was possible to compose the bit patterns needed for each instruction.

The early computers had relatively simple fixed layouts for their instruction words; the layout shown in Figure 2.3 would have been typical (though the number of operand bits would have varied).

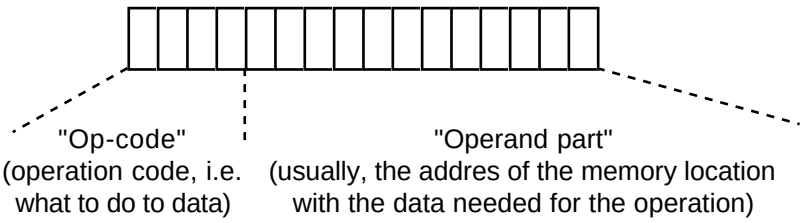


Figure 2.3 Simple layout for an instruction word

The programmer would have had a list of the bit patterns for each of the instructions that could be executed by the CPU

```
LOAD        0000                ADD         0001
STORE       0010                MULTIPLY    0011
SUBTRACT    0100                COMPARE     0101
...
```

The "op-code" part of an instruction could easily be filled in by reference to this table. Working out the "operand" part was a bit more difficult --- the programmer had to convert the word numbers from the address table into binary values. Then, as shown in Figure 2.4, the complete instruction could be "assembled" by fitting together the bits for the opcode part and the bits for the address.

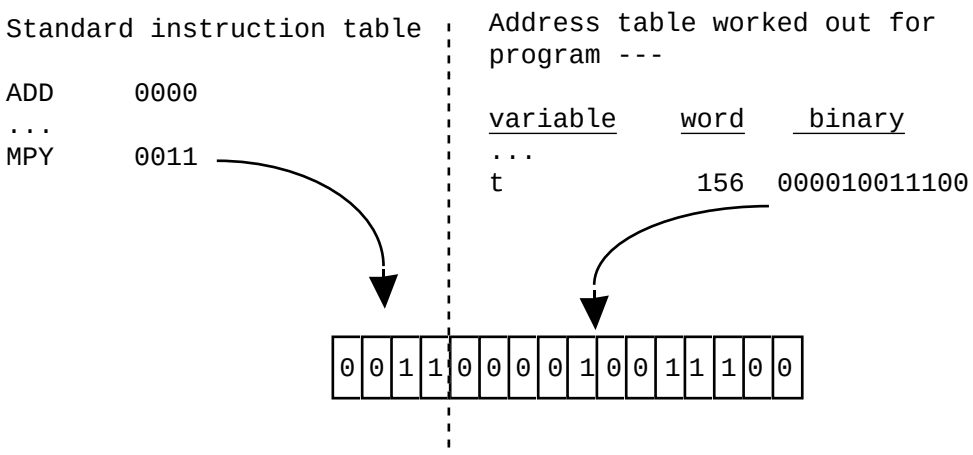
Loading on the switches

The instruction still had to be placed into the memory of the machine. This would have been done using a set of switches on the front of the computer. One set of switches would have been set to represent the address for the instruction (switch down for a 0 bit, switch up for a 1). A second set of switches would have been set up with the bit pattern just worked out for that instruction. Then a "load address" button would have been pressed.

Every instruction in the program had to be worked out, and then loaded individually into memory, in this manner. As you can imagine, this approach to programming a computer was tedious and error prone.

By 1949, bigger computers with more memory were becoming available. These had as many as one thousand words of memory (!) for storing data and programs. Toggling in the bit patterns for a program with several hundred instructions was simply not feasible.

But the program preparation process started to become a little more sophisticated and a bit more automated. New kinds of programs were developed that helped the programmers in their task of composing programs to solve problems. These new software development aids were *loaders* and *symbolic assemblers*.



Coding the instruction

MPY T

Figure 2.4 "Assembling" an instruction.

2.2 LOADERS

A "loader" is a fairly simple program. It reads in bit patterns representing the instructions (and initialized data elements) needed for a program, and stores these bit patterns in the correct locations in memory.

The bit patterns were originally represented on media like punched cards or paper tapes (a hole in the card at particular position might represent a 1 bit, no hole meant a 0 bit). With the bit patterns were punched on cards, a loader program could read these cards and store the bit patterns, starting at some fixed address and filling out successive words. The cards used typically had 12 rows where bits could be punched, and 80 columns. Depending on the word length of the computer, one instruction would be represented by two to four columns of a card; each card could hold the codes for twenty or more instructions.

The "loader" program itself was typically about 10 instructions in length. It would get toggled into memory using the switches on the front of the computer (using the last few locations in memory, or some other area not used by the program being loaded). The code of the loader would start by noting the address where the first instruction was to be placed. Then there would be a loop. In this loop, columns would be read from cards, instruction words built up from the bits read and, when complete would be stored in memory. As each instruction was stored, the loader would update its record of where to store the next. The loader would stop when it read some special end-marker bit pattern from a card. The person running the machine could then set the starting address for their program using the switches and set it running.

2.3 ASSEMBLERS

By 1950, programmers were using "assembler" programs to help create the bit pattern representation of the instructions.

The difficult creative aspect of programming is deciding the correct sequence of instructions to solve the problem. Conversion of the chosen instructions to bit patterns is an essentially mechanical process, one that can be automated without too much difficulty.

If an assembler was available, programmers could write out their programs using the mnemonic instruction names (LOAD, MULTIPLY, etc) and named data elements. Once the program had been drafted, it was punched on cards, one card for each instruction. This process produced the program source card deck, Figure 2.5.

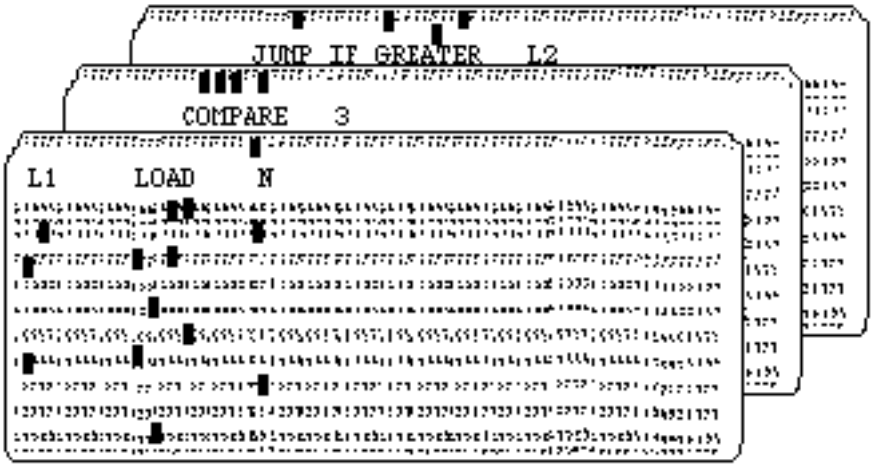


Figure 2.5 Assembly language source deck

This "source" program was converted to binary instruction patterns by an assembler program (it "assembles" instructions). The source cards would be read by the assembler which would generate a binary card deck that could be loaded by the loader.

The assembler program performs a translation process – converting the mnemonic instruction names and variable names into the correct bits. Assembler programs are meant to be fairly small (particularly the early ones that had to fit into a memory with only a few hundred words). Consequently, the translation task must not be complex.

"Assembly languages" are designed to have simple regular structures so that translation is easy. Assembly languages are defined by a few rules that specify the different kinds of instruction and data element allowed. In the early days, further rules specified how an instruction should be laid out on a card so as to make it even easier for the translation code to find the different parts of an instruction.

Syntax rules

Rules that specify the legal constructs in a language are that language's "syntax rules". For assembly languages, these rules are simple; for example, a particular assembly language might be defined by the following rules:

1. One statement per line (card).
2. A statement can be either:
   An optional "label" at start and an instruction
   or
   a "label" (variable name) and a numeric value.
3. A "label" is a name that starts with a letter and has 6 or fewer letters and digits.
4. An instruction is either:
   an input/output instruction
   or
   a data manipulation instruction
   or
   ...
   Instructions are specified using names from a standard table provided with the assembler.
5. Labels start in column 1 of a card, instructions in column 8, operand details in column 15.
6. …

An assembler program uses a table with the names of all the instructions that could be executed by the CPU. The instruction names are shortened to mnemonic abbreviations (with 3 letters or less) ...

```
LOAD             --->  L
ADD              --->  A
STORE            --->  S
GOTO (JUMP)      --->  J
JUMP_IF_GREATER  --->  JGT
MULTIPLY         --->  MPY
STOP (HALT)      --->  STP
COMPARE          --->  C
...
```

Short names of 1--3 characters require less storage space for this table (this was important in the early machines with their limited memories).

"Two-pass" translation process

If an assembly language is sufficiently restricted in this way, it becomes relatively simple to translate from source statements to binary code. The assembler program (translator) reads the text of an assembly language program twice, first working out information that it will need and then generating the bit patterns for the instructions.

The first time the text is read, the assembler works out where to store each instruction and data element. This involves simply counting the cards (assembly language statements), and noting those where labels are defined, see Figure 2.6. The names of the labels and the number of the card where they occurred are stored in a table that the assembler builds up in memory.

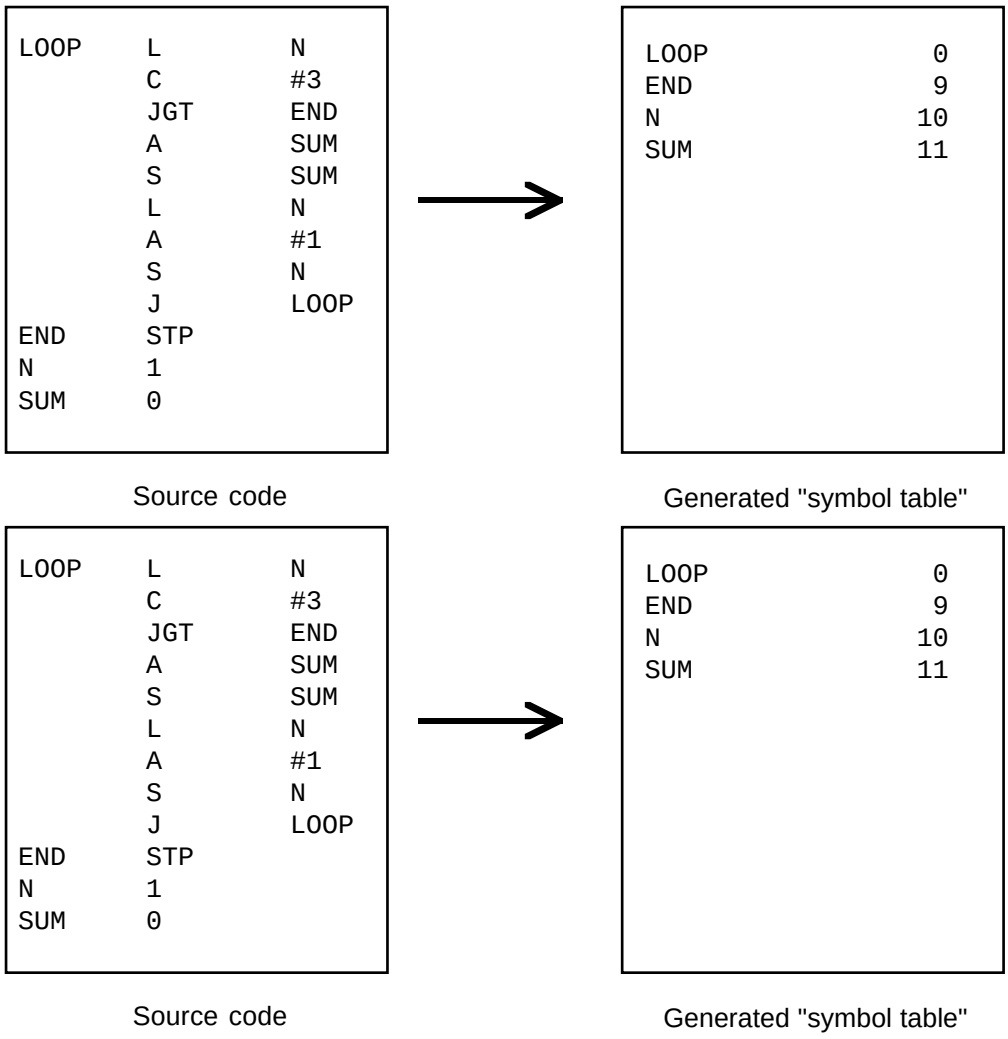


Figure 2.6 Generating a symbol table in "pass 1" of the assembly process.

The second time the source code is read, the assembler works out the bit patterns and saves them (by punching on cards – or in more modern systems by writing the information to a file).

All the translation work is done in this second pass, see Figure 2.7. The translation is largely a matter of table lookup. The assembler program finds the characters that make up an instruction name, e.g. `JGT`, and looks up the translation in the "instructions" table (`1110`). The bits for the translation are copied into the op-code part of the instruction word being assembled. If the operand part of the source instruction involves a named location, e.g. `END`, this name can be looked up in the generated symbol table. (Usually, the two tables would be combined.) Again, the translation as a bit pattern would be extracted from the table and these "address" bits would make up most of the rest of the instruction word.

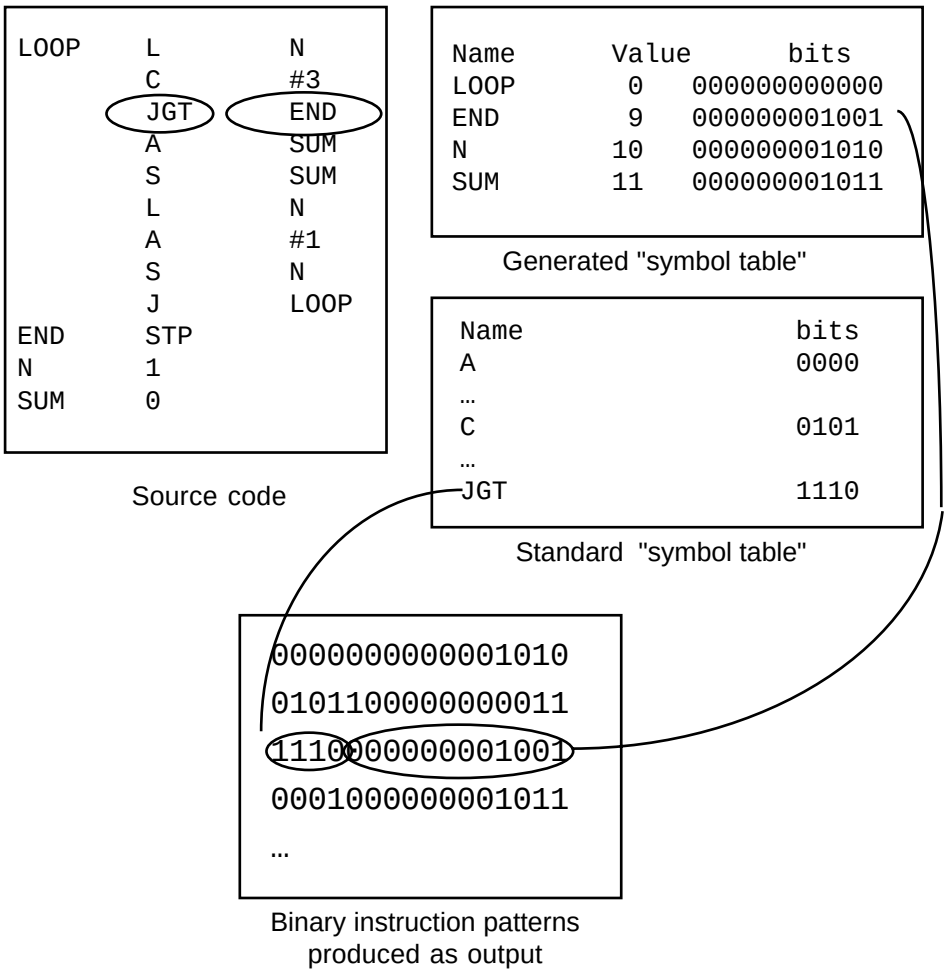


Figure 2.7 Generating code in "pass 2" of the assembly process

Apart from the op-code bits and address bits, the instruction word would have a couple of bits in the operand field for the "addressing mode". These bits would be used to distinguish cases where the rest of the operand bits held the address of the data (e.g. `A SUM`) from cases where the rest of the operand bits held the actual data (e.g. `C #3`).

2. 4 CODING IN ASSEMBLY LANGUAGE

With assemblers to translate "assembly language" source code to bit patterns, and loaders to get the bits properly into memory, the programmers of the later '40s early '50s were able to forget about the bit patterns and focus more on problem solving and coding.

Many of the details of coding were specific to a machine. After all, each different kind of CPU has a slightly different set of circuits for interpreting instructions and so offers a different set of instructions for the programmer to use. But there were basic coding patterns that appeared in slightly different forms in all programs on all machines. Programmers learnt these patterns —— and used them as sort of building blocks that helped them work out an appropriate structure for a complete program.

Coding– sequence

The simplest pattern is the sequence of instructions:

```
L           TIME
MPY         ACCEL
A           U
S           V
```

This kind of code is needed at points in a program where some "formula" has to be evaluated.

Coding – loops

Commonly, one finds places in a program where the same sequence of instructions has to be executed several times. A couple of different code patterns serve as standard "templates" for such loops:

Do some calculations to determine whether can miss out
code sequence, these calculations set a "condition flag"
Conditional jump --- if appropriate flag is set, jump beyond loop

> *Sequence of instructions forming the "body" of loop*
>
> *Jump back to calculations to determine if must*
> *execute "body" again*

Point to resume when have completed sufficient iterations of loop

This is the pattern that was used in the loop in the example at the beginning of this chapter. It is a "while loop" – the body of the loop must be executed again and again while some condition remains true (like the condition `N <= 3` in the first example).

An alternative pattern for a loop is

Start point for loop

> *Sequence of instructions forming the "body" of loop*
>
> *Do some calculations to determine whether need to*

repeat code sequence again, these calculations set a "condition flag"
Conditional jump – if the appropriate flag is set, jump back to the start of the loop
Point to resume when have completed sufficient iterations of loop.

This pattern is appropriate if you know that the instructions forming the body of the loop will always have to be executed at least once. It is a "repeat" loop; the body is repeated until some condition flag gets set to indicate that it has been done enough times.

Coding– choosing between alternative actions

Another very common requirement is to be able to chose between two (or more) alternatives. For example you might need to know the larger of two values that result from some other stage of some calculation:

```
if Velocity1 greater than Velocity2 then
        maxvelocity equals Velocity1
otherwise maxvelocity equals Velocity2
```

Patterns for such "selection code" have to be organized around the CPU's instructions for testing values and making conditional jumps.

A possible code pattern for selecting the larger of two values would be ...

```
    load first value from memory into CPU register
    compare with second value
    jump if "less flag" is set to label L2
           first value is the larger so just save it
    store in place to hold larger
    goto label L3
L2  load second value
    store in place to hold larger
L3  ...    start instruction sequence that uses larger value
```

Similar code patterns would have existed for many other kinds of conditional test.

Coding– "subroutines"

"Sequence of statements", "selections using conditional tests", and "loops" are the basic patterns for organizing a particular calculation step.

There is one other standard pattern that is very important for organizing a complete program – the *subroutine* .

Typically, there will be several places in a program where essentially the same operation is needed. For example, a program might need to read values for several different data elements. Now the data values would be numbers that would have to end up being represented as integers in binary notation, but the actual input would

be in the form of characters typed at a keyboard. It requires quite a lot of work to convert a character sequence, e.g. '1' '3' '4' (representing the number one hundred and thirty four), into the corresponding bit pattern (for 134 this is 0000000010000110). The code involves a double loop --- one loop working through successive digits, inside it there would be a "wait loop" used to read a character from the keyboard. The code for number input would take at least 15 to 20 instructions. It wouldn't have been very practical to repeat the code at each point where a data value was needed.

"Jump to subroutine" and "return from subroutine" instructions

If the code were repeated everywhere it was needed, then all your memory would get filled up with the code to read your data and you wouldn't have any room for the rest of the program (see Figure 2.8 A). Subroutines make it possible to have just one copy of some code, like the number input code (Figure 2.8 B). At places where that code needs to be used, there is a "jump to subroutine" instruction. Executing this instruction causes a jump to the start of the code and, in some machine dependent way, keeps a record of where the main program should be resumed when the code is completed. At the end of the shared code, a "return from subroutine" instruction sets the program counter so as to resume the main program at the appropriate place.

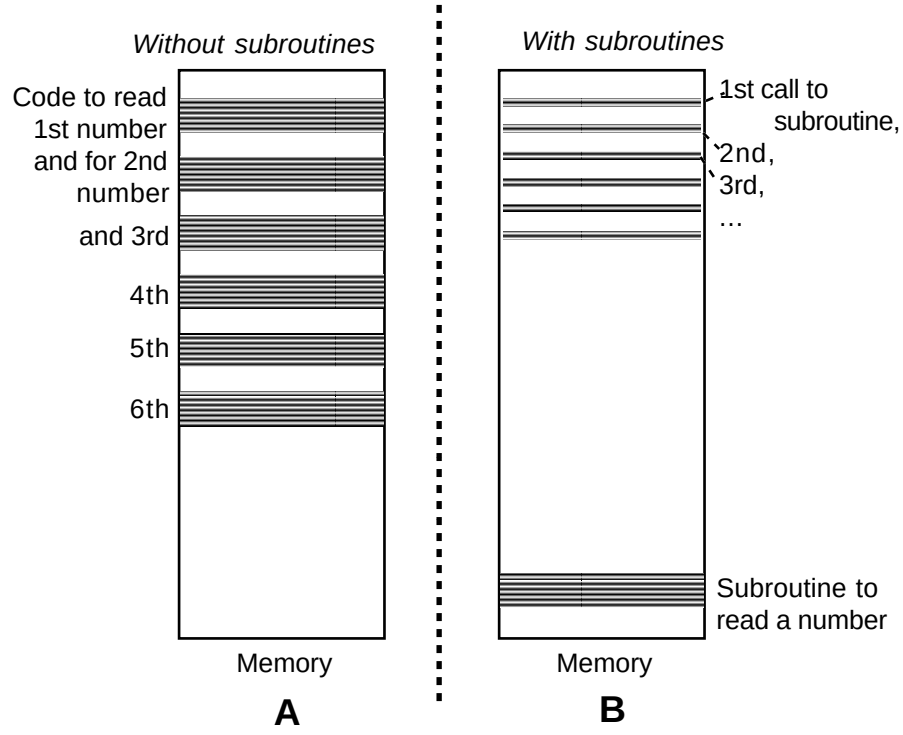


Figure 2.8 Sharing a piece of code as a subroutine.

The form of the code for the "main" program and for the subroutine would be something like the following:

```
Main program, starts by reading data values ...
START       JSR         INPUT
As the subroutine call instruction is decoded, the CPU saves
the address of the next instruction so that the main program
can be repeated when the subroutine finishes
            S           ACCEL
            JSR         INPUT
            S           U
            JSR         INPUT
            S           TFINAL
Main  program can continue with code that does calculations
on input values
LOOP        L           T
            C           TFINAL
            JGT         END
            MPY         ACCEL
            A           U
            S           V
            ...
```

and

```
Subroutine, code  would start by initializing result to
zero, then would have loops  getting input digits
            "CLEAR"     RESULT
            ...
WAIT        "KEYBOARD_READY?"
            JNE         WAIT
            "KEYBOARD_READ"
            ...
            ...

When all digits read, subroutine would put number in CPU
register and return  (RTS = return from subroutine) ...
            L             RESULT
            RTS
```

Different computers have used quite distinct schemes for "*saving the return address*" —— but in all other respects they have all implemented essentially the same subroutine call and return scheme.

Coding– "subroutine libraries"

Subroutines first appeared about 1948 or 1949. It was soon realized that they had benefits quite apart from saving memory space.

Similar subroutines tended to be needed in many different programs. Most programs had to read and print numbers; many needed to calculate values for functions like sine(), cosine(), exponential(), log() etc. Subroutines for numeric input and output, and for evaluating trigonometric functions could be written just once and "put in a library". A programmer needing one of these functions could simply copy a subroutine card deck borrowed from this library. The first book on

programming, published in 1950, explored the idea of subroutines and included several examples.

Coding – "subroutines for program design"

It was also realized that subroutines provided a way of thinking about a programming problem and breaking it down into manageable chunks. Instead of a very long "main line program" with lots of loops and conditional tests, programs could have relatively simple main programs that were comprised mainly of calls to subroutines.

Each subroutine would involve some sequential statements, loops, conditionals – and, possibly, calls to yet other subroutines.

If a code segment is long and complex, with many nested loops and criss-crossing jump (goto) instructions, then it becomes difficult to understand and the chances of coding errors are increased. Such difficulties could be reduced by breaking a problem down through the use of lots of subroutines each having a relatively simple structure.

Coding– data

Assembly languages do not provide much support for programmers when it comes to data. An assembly language will allow the programmer to attach a name to a word, or a group of memory words. The programmer has to chose how many memory words will be needed to represent a data value (one for an integer, two/ four or more for a real number, or with text some arbitrary number of words depending on the number of characters in the text "string".) The programmer can not usually indicate that a memory word (or group of words) is to store integer data or real data, and certainly the assembler program won't check how the data values are used.

In the early days of programming, programmer's distinguished two kinds of data:

Globals
: The programmers would arrange a block of memory to hold those variables that represented the main data used by the program and which could be accessed by the main program or any of the subroutines.

Locals
: Along with the code for each subroutine, the programmer would have allocated any variables needed to hold temporary results etc. These were only supposed to be used in that subroutine and so were "local" to the routine.

The organization of memory in an early computer is shown in Figure 2.9. The "local" data space for each subroutine was often located immediately after the code, so mixing together code and data parts of a program. Many machines were designed around the concept of "global" and "local" data. These machine

architectures reserved specific memory addresses for global data and provided special "addressing modes" for accessing global and local data elements. Different memory organizations were proposed in the early 1960s in association with the development of the "Algol" languages. These alternative ways of arranging memory are now more popular; they are introduced in section 4.8.

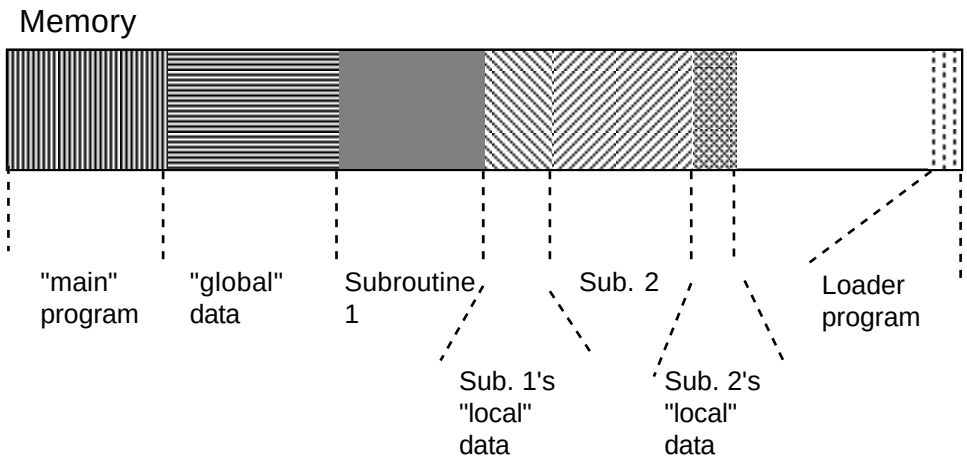


Figure 2.9 Organization of a program in the memory of an early computer.

2.5 FROM ASSEMBLY LANGUAGE TO "HIGH LEVEL" LANGUAGES

In the 1950s, most programs were written in assembly language.

But assembly languages have disadvantages – they involve the programmer in too much detail. The programmer is stuck with thinking about "instructions in the computer". Now sometimes one does want to chose the instruction sequence; people still program in assembly language when dealing with small sections of time critical code. But for most programming tasks, one doesn't really want to get down to the level of detail of choosing individual instructions.

Starting in the mid-50s, "higher level programming languages" began to appear. Initially they did little more than provided packaged versions of those coding patterns that assembly language programmers were already familiar with. Thus a "high level programming language" might provide some kind of "DO- loop" ...

```
      DO 10     IT = 1, 4
      ...
10    CONTINUE
```

Here, the programmer has specified that a sequence of statements (`...`) is to be executed 4 times. It is fairly easy to translate such a high level specification of a loop into assembly language code with all the instructions for comparisons, conditional tests, jumps etc.

When programming in a high level language, the programmer no longer has to bother about individual instructions. Instead, the programmer works at a higher level of abstraction – more remote from machine details. Automatic translation systems expand the high level code into the instruction sequences that must be placed in the computer's memory.

Over time, high level languages have evolved to take on more responsibilities and to do more for the programmer. But in most high level languages, you can still see the basic patterns of "instruction sequence", "loop", "conditional tests for selection", and "subroutines" that have been inherited from earlier assembly language style programming.