4 MACHINE EMULATION

In Chapter 2 we discussed the use of emulation or interpretation as a tool for programming language translation. In this chapter we aim to discuss hypothetical machine languages and the emulation of hypothetical machines for these languages in more detail. Modern computers are among the most complex machines ever designed by the human mind. However, this is a text on programming language translation and not on electronic engineering, and our restricted discussion will focus only on rather primitive object languages suited to the simple translators to be discussed in later chapters.

4.1 Simple machine architecture

Many CPU (central processor unit) chips used in modern computers have one or more internal **registers** or **accumulators**, which may be regarded as highly local memory where simple arithmetic and logical operations may be performed, and between which local data transfers may take place. These registers may be restricted to the capacity of a single byte (8 bits), or, as is typical of most modern processors, they may come in a variety of small multiples of bytes or machine words.

One fundamental internal register is the **instruction register** (IR), through which moves the bitstrings (bytes) representing the fundamental machine-level instructions that the processor can obey. These instructions tend to be extremely simple - operations such as "clear a register" or "move a byte from one register to another" being the typical order of complexity. Some of these instructions may be completely defined by a single byte value. Others may need two or more bytes for a complete definition. Of these multi-byte instructions, the first usually denotes an operation, and the rest relate either to a value to be operated upon, or to the address of a location in memory at which can be found the value to be operated upon.

The simplest processors have only a few **data registers**, and are very limited in what they can actually do with their contents, and so processors invariably make provision for interfacing to the memory of the computer, and allow transfers to take place along so-called **bus** lines between the internal registers and the far greater number of external memory locations. When information is to be transferred to or from memory, the CPU places the appropriate address information on the address bus, and then transmits or receives the data itself on the data bus. This is illustrated in Figure 4.1.

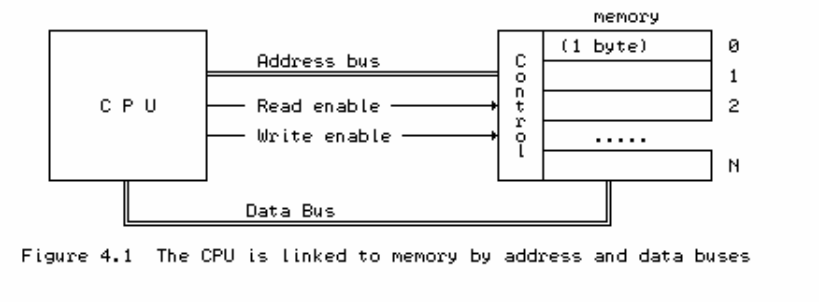

Figure 4.1 The CPU is linked to memory by address and data buses

The memory may simplistically be viewed as a one-dimensional array of byte values, analogous to what might be described in high-level language terms by declarations like the following

```
TYPE
  ADDRESS = CARDINAL [0 .. MemSize - 1];
  BYTES   = CARDINAL [0 .. 255];
VAR
  Mem : ARRAY ADDRESS OF BYTES;
```

in Modula-2, or, in C++ (which does not provide for the subrange types so useful in this regard)

```
typedef unsigned char BYTES;
BYTES Mem[MemSize];
```

Since the memory is used to store not only "data" but also "instructions", another important internal register in a processor, the so-called **program counter** or **instruction pointer** (denoted by PC or IP), is used to keep track of the address in memory of the next instruction to be fed to the processor's instruction register (IR).

Perhaps it will be helpful to think of the processor itself in high-level terms:

```
TYPE
  PROCESSOR =                   struct processor {
    RECORD                        BYTES IR;
      IR,                         BYTES R1, R2, R3;
      R1, R2, R3 : BYTES;         unsigned PC;
      PC : ADDRESS;             };
    END;
VAR                             processor cpu;
  CPU : PROCESSOR;
```

The operation of the machine is repeatedly to *fetch* a byte at a time from memory (along the data bus), place it in the IR, and then *execute* the operation which this byte represents. Multi-byte instructions may require the fetching of further bytes before the instruction itself can be decoded fully by the CPU, of course. After the instruction denoted by the contents of IR has been executed, the value of PC will have been changed to point to the next instruction to be fetched. This **fetch-execute cycle** may be described by the following algorithm:

```
BEGIN
  CPU.PC := initialValue;  (* address of first code instruction *)
  LOOP
    CPU.IR := Mem[CPU.PC]; (* fetch *)
    Increment(CPU.PC);     (* bump PC in anticipation *)
    Execute(CPU.IR);       (* affecting other registers, memory, PC *)
                           (* handle machine interrupts if necessary *)
  END
END.
```

Normally the value of PC alters by small steps (since instructions are usually stored in memory in sequence); execution of branch instructions may, however, have a rather more dramatic effect. So might the occurrence of hardware interrupts, although we shall not discuss interrupt handling further.

A program for such a machine consists, in the last resort, of a long string of byte values. Were these to be written on paper (as binary, decimal, or hexadecimal values), they would appear pretty meaningless to the human reader. We might, for example, find a section of program reading

```
        25  45  21  34  34  30  45
```

Although it may not be obvious, this might be equivalent to a high-level statement like

```
        Price := 2 * Price + MarkUp;
```

Machine-level programming is usually performed by associating *mnemonics* with the recognizable

operations, like `HLT` for "halt" or `ADD` for "add to register". The above code is far more comprehensible when written (with commentary) as

```
        LDA  45   ; load accumulator with value stored in memory location 45
        SHL       ; shift accumulator one bit left (multiply by 2)
        ADI  34   ; add 34 to the accumulator
        STA  45   ; store the value in the accumulator at memory location 45
```

Programs written in an assembly language - which have first to be assembled before they can be executed - usually make use of other named entities, for example

```
MarkUp EQU  34       ; CONST MarkUp = 34;
       LDA  Price    ; CPU.A := Price;
       SHL           ; CPU.A := 2 * CPU.A;
       ADI  MarkUp   ; CPU.A := CPU.A + 34;
       STA  Price    ; Price := CPU.A;
```

When we use code fragments such as these for illustration we shall make frequent use of commentary showing an equivalent fragment written in a high-level language. Commentary follows the semicolon on each line, a common convention in assembler languages.

4.2 Addressing modes

As the examples given earlier suggest, programs prepared at or near the machine level frequently consist of a sequence of simple instructions, each involving a machine-level operation and one or more parameters.

An example of a simple operation expressed in a high-level language might be

```
AmountDue := Price + Tax;
```

Some machines and assembler languages provide for such operations in terms of so-called **three-address code**, in which an *operation* - denoted by a mnemonic usually called the **opcode** - is followed by two *operands* and a *destination*. In general this takes the form

operation destination, operand$_1$*, operand*$_2$

for example

```
ADD      AmountDue, Price, Tax
```

We may also express this in a general sense as a function call

destination := operation(operand$_1$*, operand*$_2$ *)*

which helps to stress the important idea that the *operands* really denote "values", while the *destination* denotes a processor register, or an address in memory where the result is to be stored.

In many cases this generality is restricted (that is, the machine suffers from non-orthogonality in design). Typically the value of one *operand* is required to be the value originally stored at the *destination*. This corresponds to high-level statements like

```
Price := Price * InflationFactor;
```

and is mirrored at the low-level by so-called **two-address code** of the general form

operation destination, operand

for example

```
MUL      Price, InflationFactor
```

In passing, we should point out an obvious connection between some of the assignment operations in C++ and two-address code. In C++ the above assignment would probably have been written

```
Price *= InflationFactor;
```

which, while less transparent to a Modula-2 programmer, is surely a hint to a C++ compiler to generate code of this form. (Perhaps this example may help you understand why C++ is regarded by some as the world's finest assembly language!)

In many real machines even general two-address code is not found at the machine level. One of *destination* and *operand* might be restricted to denoting a machine register (the other one might denote a machine register, or a constant, or a machine address). This is often called **one and a half address code**, and is exemplified by

```
MOV     R1, Value     ; CPU.R1 := Value
ADD     Answer, R1    ; Answer := Answer + CPU.R1
MOV     Result, R2    ; Result := CPU.R2
```

Finally, in so-called *accumulator machines* we may be restricted to **one-address code**, where the destination is always a machine register (except for those operations that copy (store) the contents of a machine register into memory). In some assembler languages such instructions may still appear to be of the two-address form, as above. Alternatively they might be written in terms of opcodes that have the register implicit in the mnemonic, for example

```
LDA     Value        ; CPU.A := Value
ADA     Answer       ; CPU.A := CPU.A + Answer
STB     Result       ; Result := CPU.B
```

Although many of these examples might give the impression that the corresponding machine level operations require multiple bytes for their representation, this is not necessarily true. For example, operations that only involve machine registers, exemplified by

```
MOV     R1, R2       ;  CPU.R1 := CPU.R2
LDA     B            ;  CPU.A := CPU.B
TAX                  ;  CPU.X := CPU.A
```

might require only a single byte - as would be most obvious in an assembler language that used the third representation. The assembly of such programs is be eased considerably by a simple and self-consistent notation for the source code, a subject that we shall consider further in a later chapter.

In those instructions that *do* involve the manipulation of values other than those in the machine registers alone, multi-byte instructions are usually required. The first byte typically specifies the operation itself (and possibly the register or registers that are involved), while the remaining bytes specify the other values (or the memory addresses of the other values) involved. In such instructions there are several ways in which the ancillary bytes might be used. This variety gives rise to what are known as different **addressing modes** for the processor, and whose purpose it is to provide an **effective address** to be used in an instruction. Exactly which modes are available varies tremendously from processor to processor, and we can mention only a few representative examples here. The various possibilities may be distinguished in some assembler languages by the use of different mnemonics for what at first sight appear to be closely related operations. In other assembler languages the distinction may be drawn by different syntactic forms used to specify the registers, addresses or values. One may even find different assembler languages for a common

processor.

In **inherent addressing** the operand is implicit in the opcode itself, and often the instruction is contained in a single byte. For example, to clear a machine register named `A` we might have

```
CLA          or       CLR  A         ;  CPU.A := 0
```

Again we stress that, though the second form seems to have two components, it does not always imply the use of two bytes of code at the machine level.

In **immediate addressing** the ancillary bytes for an instruction typically give the *actual value* that is to be combined with a value in a register. Examples might be

```
ADI  34      or       ADD  A, #34    ;  CPU.A := CPU.A + 34
```

In these two addressing modes the use of the word "address" is almost misleading, as the value of the ancillary bytes may often have nothing to do with a memory address at all. In the modes now to be discussed the connection with memory addresses is far more obvious.

In **direct** or **absolute addressing** the ancillary bytes typically specify the *memory address* of the value that is to be retrieved or combined with the value in a register, or specify where a register value is to be stored. Examples are

```
LDA  34      or       MOV  A, 34     ;  CPU.A := Mem[34]
STA  45               MOV  45, A     ;  Mem[45] := CPU.A
ADD  38               ADD  A, 38     ;  CPU.A := CPU.A + Mem[38]
```

Beginners frequently confuse immediate and direct addressing, a situation not improved by the fact that there is no consistency in notation between different assembler languages, and there may even be a variety of ways of expressing a particular addressing mode. For example, for the Intel 80x86 processors as used in the IBM-PC and compatibles, low-level code is written in a two-address form similar to that shown above - but the immediate mode is denoted without needing a special symbol like #, while the direct mode may have the address in brackets:

```
ADD  AX, 34    ;  CPU.AX := CPU.AX + 34  Immediate
MOV  AX, [34]  ;  CPU.AX := Mem[34]      Direct
```

In **register-indexed addressing** one of the operands in an instruction specifies both an address and also an *index register*, whose value at the time of execution may be thought of as specifying the subscript to an array stored from that address

```
LDX  34      or       MOV  A, 34[X]  ;  CPU.A := Mem[34 + CPU.X]
STX  45               MOV  45[X], A  ;  Mem[45+CPU.X] := CPU.A
ADX  38               ADD  A, 38[X]  ;  CPU.A := CPU.A + Mem[38+CPU.X]
```

In **register-indirect addressing** one of the operands in an instruction specifies a register whose value at the time of execution gives the effective address where the value of the operand is to be found. This relates to the concept of *pointers* as used in Modula-2, Pascal and C++.

```
MOV   R1, @R2     ;  CPU.R1 := Mem[CPU.R2]
MOV   AX, [BX]    ;  CPU.AX := Mem[CPU.BX]
```

Not all the registers in a machine can necessarily be used in these ways. Indeed, some machines have rather awkward restrictions in this regard.

Some processors allow for very powerful variations on indexed and indirect addressing modes. For example, in **memory-indexed** addressing, a single operand may specify two memory addresses - the first of which gives the address of the first element of an array, and the second of which gives

the address of a variable whose value will be used as a subscript to the array.

```
        MOV   R1, 400[100]    ;  CPU.R1 := Mem[400 + Mem[100]]
```

Similarly, in **memory-indirect addressing** one of the operands in an instruction specifies a memory address at which will be found a value that forms the effective address where another operand is to be found.

```
        MOV   R1, @100    ;  CPU.R1 := Mem[Mem[100]]
```

This mode is not as commonly found as the others; where it does occur it directly corresponds to the use of pointer variables in languages that support them. Code like

```
TYPE
  ARROW = POINTER TO CARDINAL;          typedef int *ARROW;
VAR
  Arrow  : ARROW;                       ARROW Arrow;
  Target : CARDINAL;                    int Target;
BEGIN
  Target := Arrow^;                     Target = *Arrow;
```

might translate to equivalent code in assembler like

```
        MOV   AX, @Arrow
        MOV   Target, AX
```

or even

```
        MOV   Target, @Arrow
```

where, once again, we can see an immediate correspondence between the syntax in C++ and the corresponding assembler.

Finally, in **relative addressing** an operand specifies an amount by which the current program count register PC must be incremented or decremented to find the actual address of interest. This is chiefly found in "branching" instructions, rather than in those that move data between various registers and/or locations in memory.

Further reading

Most books on assembler level programming have far deeper discussions of the subject of addressing modes than we have presented. Two very readable accounts are to be found in the books by Wakerly (1981) and MacCabe (1993). A deeper discussion of machine architectures is to be found in the book by Hennessy and Patterson (1990).

4.3 Case study 1 - A single-accumulator machine

Although sophisticated processors may have several registers, their basic principles - especially as they apply to emulation - may be illustrated by the following model of a single-accumulator processor and computer, very similar to one suggested by Wakerly (1981). Here we shall take things to extremes and presume the existence of a system with all registers only 1 byte (8 bits) wide.

4.3.1 Machine architecture

Diagrammatically we might represent this machine as in Figure 4.2.

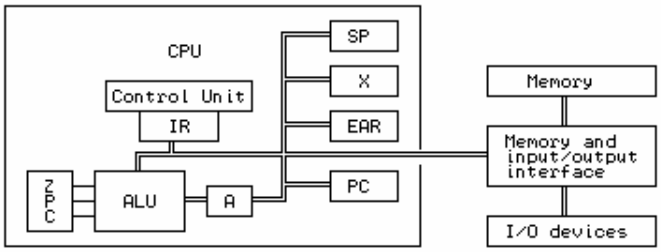

Figure 4.2 A simple single-accumulator CPU and computer

The symbols in this diagram refer to the following components of the machine

> `ALU` is the *arithmetic logic unit*, where arithmetic and logical operations are actually performed.
>
> `A` is the 8-bit *accumulator*, a register for doing arithmetic or logical operations.
>
> `SP` is an 8-bit *stack pointer*, a register that points to an area in memory that may be utilized as a stack.
>
> `X` is an 8-bit *index register*, which is used in indexing areas of memory which conceptually form data arrays.
>
> `Z, P, C` are single bit *condition flags* or *status registers*, which are set "true" when an operation causes a register to change to a zero value, or to a positive value, or to propagate a carry, respectively.
>
> `IR` is the 8-bit *instruction register*, in which is held the byte value of the instruction currently being executed.
>
> `PC` is the 8-bit *program counter*, which contains the address in memory of the instruction that is next to be executed.
>
> `EAR` is the *effective address register*, which contains the address of the byte of data which is being manipulated by the current instruction.

The programmer's model of this sort of machine is somewhat simpler - it consists of a number of "variables" (in the C++ or Modula-2 sense), each of which is one byte in capacity. Some of these correspond to processor registers, while the others form the random access read/write (RAM) memory, of which we have assumed there to be 256 bytes, addressed by the values 0 through 255. In this memory, as usual, will be stored both the data and the instructions for the program under execution. The processor, its registers, and the associated RAM memory can be thought of as though they were described by declarations like

```
TYPE
  BYTES = CARDINAL [0 .. 255];
  PROCESSOR = RECORD
    A, SP, X, IR, PC : BYTES;
    Z, P, C : BOOLEAN;
  END;
```

```
typedef unsigned char bytes;
struct processor {
  bytes a, sp, x, ir, pc;
  bool z, p, c;
};
```

```
TYPE STATUS = (running, finished,              typedef enum { running, finished,
              nodata, baddata,                   nodata, baddata, badop
              badop);                          } status;
VAR
  CPU : PROCESSOR;                             processor cpu;
  Mem : ARRAY BYTES OF BYTES;                  bytes mem[256];
  PS  : STATUS;                                status ps;
```

where the concept of the **processor status** PS has been introduced in terms of an enumeration that defines the states in which an emulator might find itself.

4.3.2 Instruction set

Some machine operations are described by a single byte. Others require two bytes, and have the format

```
Byte 1     Opcode
Byte 2     Address field
```

The set of machine code functions available is quite small. Those marked * affect the P and Z flags, and those marked + affect the C flag. An informal description of their semantics follows:

Mnemonic			Hex opcode	Decimal	Function
NOP			00h	0	No operation (this might be used to set a break point in an emulator)
CLA			01h	1	Clear accumulator A
CLC	+		02h	2	Clear carry bit C
CLX			03h	3	Clear index register X
CMC	+		04h	4	Complement carry bit C
INC		*	05h	5	Increment accumulator A by 1
DEC		*	06h	6	Decrement accumulator A by 1
INX		*	07h	7	Increment index register X by 1
DEX		*	08h	8	Decrement index register X by 1
TAX			09h	9	Transfer accumulator A to index register X
INI		*	0Ah	10	Load accumulator A with integer read from input in decimal
INH		*	0Bh	11	Load accumulator A with integer read from input in hexadecimal
INB		*	0Ch	12	Load accumulator A with integer read from input in binary
INA		*	0Dh	13	Load accumulator A with ASCII value read from input (a single character)
OTI			0Eh	14	Write value of accumulator A to output as a signed decimal number
OTC			0Fh	15	Write value of accumulator A to output as an unsigned decimal number
OTH			10h	16	Write value of accumulator A to output as an unsigned hexadecimal number
OTB			11h	17	Write value of accumulator A to output as an unsigned binary number
OTA			12h	18	Write value of accumulator A to output as a single character
PSH			13h	19	Decrement SP and push value of accumulator A onto stack
POP		*	14h	20	Pop stack into accumulator A and increment SP
SHL	+	*	15h	21	Shift accumulator A one bit left
SHR	+	*	16h	22	Shift accumulator A one bit right
RET			17h	23	Return from subroutine (return address popped from stack)
HLT			18h	24	Halt program execution

The above are all single-byte instructions. The following are all double-byte instructions.

LDA	B	*	19h	25	Load accumulator A directly with contents of location whose address is given as B
LDX	B	*	1Ah	26	Load accumulator A with contents of location whose address is given as B, indexed by the value of X (that is, an address computed as the value of B + X)
LDI	B	*	1Bh	27	Load accumulator A with the immediate value B
LSP	B		1Ch	28	Load stack pointer SP with contents of location whose address is given as B
LSI	B		1Dh	29	Load stack pointer SP immediately with the value B
STA	B		1Eh	30	Store accumulator A on the location whose address is given as B

STX	B		1Fh	31	Store accumulator A on the location whose address is given as B, indexed by the value of X
ADD	B	+ *	20h	32	Add to accumulator A the contents of the location whose address is given as B
ADX	B	+ *	21h	33	Add to accumulator A the contents of the location whose address is given as B,indexed by the value of X
ADI	B	+ *	22h	34	Add the immediate value B to accumulator A
ADC	B	+ *	23h	35	Add to accumulator A the value of the carry bit C plus the contents of the location whose address is given as B
ACX	B	+ *	24h	36	Add to accumulator A the value of the carry bit C plus the contents of the location whose address is given as B, indexed by the value of X
ACI	B	+ *	25h	37	Add the immediate value B plus the value of the carry bit C to accumulator A
SUB	B	+ *	26h	38	Subtract from accumulator A the contents of the location whose address is given as B
SBX	B	+ *	27h	39	Subtract from accumulator A the contents of the location whose address is given as B, indexed by the value of X
SBI	B	+ *	28h	40	Subtract the immediate value B from accumulator A
SBC	B	+ *	29h	41	Subtract from accumulator A the value of the carry bit C plus the contents of the location whose address is given as B
SCX	B	+ *	2Ah	42	Subtract from accumulator A the value of the carry bit C plus the contents of the location whose address is given as B, indexed by the value of X
SCI	B	+ *	2Bh	43	Subtract the immediate value B plus the value of the carry bit C from accumulator A
CMP	B	+ *	2Ch	44	Compare accumulator A with the contents of the location whose address is given as B
CPX	B	+ *	2Dh	45	Compare accumulator A with the contents of the location whose address is given as B, indexed by the value of X
CPI	B	+ *	2Eh	46	Compare accumulator A directly with the value B

These comparisons are done by virtual subtraction of the operand from A, and setting the flags P and Z as appropriate

ANA	B	+ *	2Fh	47	Bitwise AND accumulator A with the contents of the location whose address is given as B
ANX	B	+ *	30h	48	Bitwise AND accumulator A with the contents of the location whose address is given as B, indexed by the value of X
ANI	B	+ *	31h	49	Bitwise AND accumulator A with the immediate value B
ORA	B	+ *	32h	50	Bitwise OR accumulator A with the contents of the location whose address is given as B
ORX	B	+ *	33h	51	Bitwise OR accumulator A with the contents of the location whose address is given as B, indexed by the value of X
ORI	B	+ *	34h	52	Bitwise OR accumulator A with the immediate value B
BRN	B		35h	53	Branch to the address given as B
BZE	B		36h	54	Branch to the address given as B if the Z condition flag is set
BNZ	B		37h	55	Branch to the address given as B if the Z condition flag is unset
BPZ	B		38h	56	Branch to the address given as B if the P condition flag is set
BNG	B		39h	57	Branch to the address given as B if the P condition flag is unset
BCC	B		3Ah	58	Branch to the address given as B if the C condition flag is unset
BCS	B		3Bh	59	Branch to the address given as B if the C condition flag is set
JSR	B		3Ch	60	Call subroutine whose address is B, pushing return address onto the stack

Most of the operations listed above are typical of those found in real machines. Notable exceptions are provided by the I/O (input/output) operations. Most real machines have extremely primitive facilities for doing anything like this directly, but for the purposes of this discussion we shall cheat somewhat and assume that our machine has several very powerful single-byte opcodes for handling I/O. (Actually this is not cheating too much, for some macro-assemblers allow instructions like this which are converted into procedure calls into part of an underlying operating system, stored perhaps in a ROM BIOS).

A careful examination of the machine and its instruction set will show some features that *are* typical of real machines. Although there are three data registers, `A, X` and `SP`, two of them (`X` and `SP`) can only be used in very specialized ways. For example, it is possible to transfer a value from `A` to `X`, but not vice versa, and while it is possible to load a value into `SP` it is not possible to examine the value of `SP` at a later stage. The logical operations affect the carry bit (they all unset it), but, surprisingly, the `INC` and `DEC` operations do not.

It is this model upon which we shall build an emulator in section 4.3.4. In a sense the formal semantics of these opcodes are then embodied directly in the **operational semantics** of the machine (or pseudo-machine) responsible for executing them.

Exercises

4.1 Which addressing mode is used in each of the operations defined above? Which addressing modes are not represented?

4.2 Many 8-bit microprocessors have 2-byte (16-bit) index registers, and one, two, and three-byte instructions (and even longer). What peculiar or restrictive features does our machine possess, compared to such processors?

4.3 As we have already commented, informal descriptions in English, as we have above, are not as precise as semantics that are formulated mathematically. Compare the informal description of the `INC` operation with the following:

```
INC  *  05h   5   A := (A + 1) mod 256;  Z := A = 0;  P := A IN {0 ... 127}
```

Try to express the semantics of each of the other machine instructions in a similar way.

4.3.3 A specimen program

Some examples of code for this machine may help the reader's understanding. Consider the problem of reading a number and then counting the number of non-zero bits in its binary representation.

Example 4.1

The listing below shows a program to solve this problem coded in an ASSEMBLER language based on the mnemonics given previously, as it might be listed by an assembler program, showing the hexadecimal representation of each byte and where it is located in memory.

```
00                  BEG                 ; Count the bits in a number
00    0A            INI                 ; Read(A)
01          LOOP                        ; REPEAT
01    16            SHR                 ;  A := A DIV 2
02    3A 0D         BCC     EVEN        ;  IF A MOD 2 # 0 THEN
04    1E 13         STA     TEMP        ;    TEMP := A
06    19 14         LDA     BITS
08    05            INC
09    1E 14         STA     BITS        ;    BITS := BITS + 1
0B    19 13         LDA     TEMP        ;    A := TEMP
0D    37 01 EVEN    BNZ     LOOP        ; UNTIL A = 0
0F    19 14         LDA     BITS        ;
11    0E            OTI                 ; Write(BITS)
12    18            HLT                 ; terminate execution
13          TEMP    DS      1           ; VAR TEMP : BYTE
14    00    BITS    DC      0           ;     BITS : BYTE
```

```
15                  END
```

Example 4.2 (absolute byte values)

In a later chapter we shall discuss how this same program can be translated into the following corresponding absolute format (expressed this time as decimal numbers):

```
10 22 58 13 30 19 25 20  5 30 20 25 19 55  1 25 20 14 24  0  0
```

Example 4.3 (mnemonics with absolute address fields)

For the moment, we shall allow ourselves to consider the absolute form as equivalent to a form in which the mnemonics still appear for the sake of clarity, but where the operands have all been converted into absolute (decimal) addresses and values:

```
INI
SHR
BCC   13
STA   19
LDA   20
INC
STA   20
LDA   19
BNZ   1
LDA   20
OTI
HLT
0
0
```

Exercises

4.4 The machine does not possess an instruction for negating the value in the accumulator. What code would one have to write to be able to achieve this?

4.5 Similarly, it does not possess instructions for multiplication and division. Is it possible to use the existing instructions to develop code for doing these operations? If so, how efficiently can they be done?

4.6 Try to write programs for this machine that will

(a) Find the largest of three numbers.

(b) Find the largest and the smallest of a list of numbers terminated by a zero (which is not regarded as a member of the list).

(c) Find the average of a list of non-zero numbers, the list being terminated by a zero.

(d) Compute N! for small N. Try using an iterative as well as a recursive approach.

(e) Read a word and then write it backwards. The word is terminated with a period. Try using an "array", or alternatively, the "stack".

(f) Determine the prime numbers between 0 and 255.

(g) Determine the longest repeated sequence in a sequence of digits terminated with

zero. For example, for data reading 1 2 3 3 3 3 4 5 4 4 4 4 4 4 4 6 5 5 report that "4 appeared 7 times".

(h) Read an input sequence of numbers terminated with zero, and then extract the embedded monotonically increasing sequence. For example, from 1 2 12 7 4 14 6 23 extract the sequence 1 2 12 14 23.

(i) Read a small array of integers or characters and sort them into order.

(j) Search for and report on the largest byte in the program code itself.

(k) Search for and report on the largest byte currently in memory.

(l) Read a piece of text terminated with a period, and then report on how many times each letter appeared. To make things interesting, ignore the difference between upper and lower case.

(m) Repeat some of the above problems using 16-bit arithmetic (storing values as pairs of bytes, and using the "carry" operations to perform extended arithmetic).

4.7 Based on your experiences with Exercise 4.6, comment on the usefulness, redundancy and any other features of the code set for the machine.

4.3.4 An emulator for the single-accumulator machine

Although a processor for our machine almost certainly does not exist "in silicon", its action may easily be simulated "in software". Essentially we need only to write an emulator that models the *fetch-execute* cycle of the machine, and we can do this in any suitable language for which we already have a compiler on a real machine.

Languages like Modula-2 or C++ are highly suited to this purpose. Not only do they have "bit-twiddling" capabilities for performing operations like "bitwise and", they have the advantage that one can implement the various phases of translators and emulators as coherent, clearly separated modules (in Modula-2) or classes (in C++). Extended versions of Pascal, such as Turbo Pascal, also provide support for such modules in the form of units. C is also very suitable on the first score, but is less well equipped to deal with clearly separated modules, as the header file mechanism used in C is less watertight than the mechanisms in the other languages.

In modelling our hypothetical machine in Modula-2 or C++ it will thus be convenient to define an interface in the usual way by means of a definition module, or by the public interface to a class. (In this text we shall illustrate code in C++; equivalent code in Modula-2 and Turbo Pascal will be found on the diskette that accompanies the book.)

The main responsibility of the interface is to declare an `emulator` routine for interpreting the code stored in the memory of the machine. For expediency we choose to extend the interface to expose the values of the operations, and the memory itself, and to provide various other useful facilities that will help us develop an assembler or compiler for the machine in due course. (In this, and in other interfaces, "private" members are not shown.)

```
// machine instructions - order is significant
enum MC_opcodes {
  MC_nop, MC_cla, MC_clc, MC_clx, MC_cmc, MC_inc, MC_dec, MC_inx, MC_dex,
  MC_tax, MC_ini, MC_inh, MC_inb, MC_ina, MC_oti, MC_otc, MC_oth, MC_otb,
```

```
    MC_ota, MC_psh, MC_pop, MC_shl, MC_shr, MC_ret, MC_hlt, MC_lda, MC_ldx,
    MC_ldi, MC_lsp, MC_lsi, MC_sta, MC_stx, MC_add, MC_adx, MC_adi, MC_adc,
    MC_acx, MC_aci, MC_sub, MC_sbx, MC_sbi, MC_sbc, MC_scx, MC_sci, MC_cmp,
    MC_cpx, MC_cpi, MC_ana, MC_anx, MC_ani, MC_ora, MC_orx, MC_ori, MC_brn,
    MC_bze, MC_bnz, MC_bpz, MC_bng, MC_bcc, MC_bcs, MC_jsr, MC_bad = 255 };

  typedef enum { running, finished, nodata, baddata, badop } status;
  typedef unsigned char MC_bytes;

  class MC {
    public:
      MC_bytes mem[256];    // virtual machine memory

      void listcode(void);
      // Lists the 256 bytes stored in mem on requested output file

      void emulator(MC_bytes initpc, FILE *data, FILE *results, bool tracing);
      // Emulates action of the instructions stored in mem, with program counter
      // initialized to initpc.  data and results are used for I/O.
      // Tracing at the code level may be requested

      void interpret(void);
      // Interactively opens data and results files, and requests entry point.
      // Then interprets instructions stored in mem

      MC_bytes opcode(char *str);
      // Maps str to opcode, or to MC_bad (0FFH) if no match can be found

      MC();
      // Initializes accumulator machine

  };
```

The implementation of `emulator` must model the typical *fetch-execute* cycle of the hypothetical machine. This is easily achieved by the repetitive execution of a large `switch` or `CASE` statement, and follows the lines of the algorithm given in section 4.1, but allowing for the possibility that the program may halt, or otherwise come to grief:

```
BEGIN
  InitializeProgramCounter(CPU.PC);
  InitializeRegisters(CPU.A, CPU.X, CPU.SP, CPU.Z, CPU.P, CPU.C);
  PS := running;
  REPEAT
    CPU.IR := Mem[CPU.PC]; Increment(CPU.PC)    (* fetch *)
    CASE CPU.IR OF                              (* execute *)
        . . . .
    END
  UNTIL PS # running;
  IF PS # finished THEN PostMortem END
END
```

A detailed implementation of the machine class is given as part of Appendix D, and the reader is urged to study it carefully.

Exercises

4.8 You will notice that the code in Appendix D makes no use of an explicit `EAR` register. Develop an emulator that does have such a register, and investigate whether this is an improvement.

4.9 How well does the informal description of the machine instruction set allow you to develop programs and an interpreter for the machine? Would a description in the form suggested by Exercise 4.3 be better?

4.10 Do you suppose interpreters might find it difficult to handle I/O errors in user programs?

4.11 Although we have required that the machine incorporate the three condition flags `P, Z` and `C`, we have not provided another one commonly found on such machines, namely for detecting

overflow. Introduce V as such a flag into the definition of the machine, provide suitable instructions for testing it, and modify the emulator so that V is set and cleared by the appropriate operations.

4.12 Extend the instruction set and the emulator to include operations for negating the accumulator, and for providing multiplication and division operations.

4.13 Enhance the emulator so that when it interprets a program, a full screen display is given, highlighting the instruction that is currently being obeyed and depicting the entire memory contents of the machine, as well as the state of the machine registers. For example we might have a display like that in Figure 4.3 for the program exemplified earlier, at the stage where it is about to execute the first instruction.

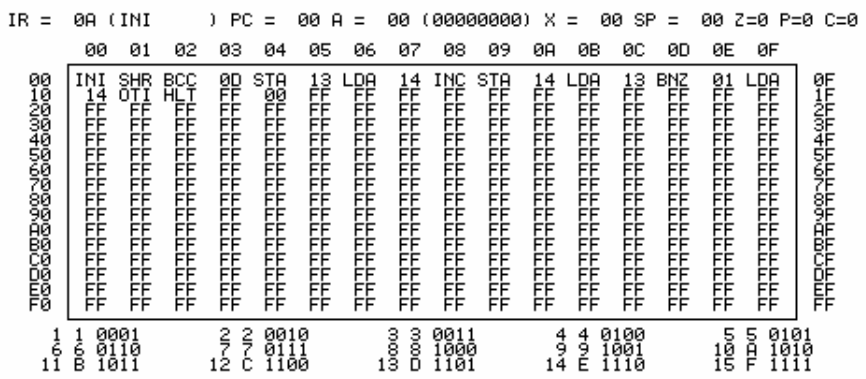

Figure 4.3 Possible display format for an enhanced interpreter

4.3.5 A minimal assembler for the machine

Given the emulator as implemented above, and some way of assembling or compiling programs, it becomes possible to implement a complete load-and-go system for developing and running simple programs. An assembler can be provided through a class with a public interface like

```
class AS {
  public:
    AS(char *sourcename, MC *M);
    // Opens source file from supplied sourcename

    ~AS();
    // Closes source file

    void assemble(bool &errors);
    // Assembles source code from src file and loads bytes of code directly
    // into memory.  Returns errors = true if source code is corrupt
};
```

In terms of these two classes, a load-and-go system might then take the form

```
void main(int argc, char *argv[])
{ bool errors;
  if (argc == 1) { printf("Usage: ASSEMBLE source\n"); exit(1); }
  MC *Machine = new MC();
  AS *Assembler = new AS(argv[1], Machine);
  Assembler->assemble(errors);
  delete Assembler;
  if (errors)
    printf("Unable to interpret code\n");
  else
  { printf("Interpreting code ...\n");
    Machine->interpret();
  }
  delete Machine;
}
```

A detailed discussion of assembler techniques is given in a later chapter. For the moment we note that various implementations matching this interface might be written, of various complexities. The very simplest of these might require the user to hand-assemble his or her programs and would amount to nothing more than a simple loader:

```
AS::AS(char *sourcename, MC *M)
{ Machine = M;
  src = fopen(sourcename, "r");
  if (src == NULL) { printf("Could not open input file\n"); exit(1); }
}

AS::~AS()
{ if (src) fclose(src); src = NULL; }

void AS::assemble(bool &errors)
{ int number;
  errors = false;
  for (int i = 0; i <= 255; i++)
  { if (fscanf(src, "%d", &number) != 1)
      { errors = true; number = MC_bad; }
    Machine->mem[i] = number % 256;
  }
}
```

However, it is not difficult to write an alternative implementation of the `assemble` routine that allows the system to accept a sequence of mnemonics and numerical address fields, like that given in Example 4.3 earlier. We present possible code, with sufficient commentary that the reader should be able to follow it easily.

```
void readmnemonic(FILE *src, char &ch, char *mnemonic)
{ int i = 0;
  while (ch > ' ')
  { if (i <= 2) { mnemonic[i] = ch; i++; }
    ch = toupper(getc(src));
  }
  mnemonic[i] = '\0';
}

void readint(FILE *src, char &ch, int &number, bool &okay)
{ okay = true;
  number = 0;
  bool negative = (ch == '-');
  if (ch == '-' || ch == '+') ch = getc(src);
  while (ch > ' ')
  { if (isdigit(ch))
      number = number * 10 + ch - '0';
    else
      okay = false;
    ch = getc(src);
  }
  if (negative) number = -number;
}

void AS::assemble(bool &errors)
{ char mnemonic[4]; // mnemonic for matching
  MC_bytes lc = 0;  // location counter
  MC_bytes op;      // assembled opcode
  int number;       // assembled number
  char ch;          // general character for input
  bool okay;        // error checking on reading numbers

  printf("Assembling code ... \n");
  for (int i = 0; i <= 255; i++)          // fill with invalid opcodes
    Machine->mem[i] = MC_bad;
  lc = 0;                                 // initialize location counter
  errors = false;                         // optimist!
  do
  { do ch = toupper(getc(src));
    while (ch <= ' ' && !feof(src));      // skip spaces and blank lines
    if (!feof(src))                       // there should be a line to assemble
    { if (isupper(ch))                    // we should have a mnemonic
      { readmnemonic(src, ch, mnemonic);  // unpack it
        op = Machine->opcode(mnemonic);   // look it up
        if (op == MC_bad)                 // the opcode was unrecognizable
          { printf("%s - Bad mnemonic at %d\n", mnemonic, lc); errors = true; }
        Machine->mem[lc] = op;            // store numerical equivalent
```

```
      }
      else                              // we should have a numeric constant
      { readint(src, ch, number, okay);   // unpack it
        if (!okay) { printf("Bad number at %d\n", lc); errors = true; }
        if (number >= 0)                  // convert to proper byte value
          Machine->mem[lc] = number % 256;
        else
          Machine->mem[lc] = (256 - abs(number) % 256) % 256;
      }
      lc = (lc + 1) % 256;              // bump up location counter
    }
  } while (!feof(src));
}
```

4.4 Case study 2 - a stack-oriented computer

In later sections of this text we shall be looking at developing a compiler that generates object code for a hypothetical "stack machine", one that may have no general data registers of the sort discussed previously, but which functions primarily by manipulating a stack pointer and associated stack. An architecture like this will be found to be ideally suited to the evaluation of complicated arithmetic or Boolean expressions, as well as to the implementation of high-level languages which support recursion. It will be appropriate to discuss such a machine in the same way as we did for the single-accumulator machine in the last section.

4.4.1 Machine architecture

Compared with normal register based machines, this one may at first seem a little strange, because of the paucity of registers. In common with most machines we shall still assume that it stores code and data in a memory that can be modelled as a linear array. The elements of the memory are "words", each of which can store a single integer - typically using a 16 bit two’s-complement representation. Diagrammatically we might represent this machine as in Figure 4.4:

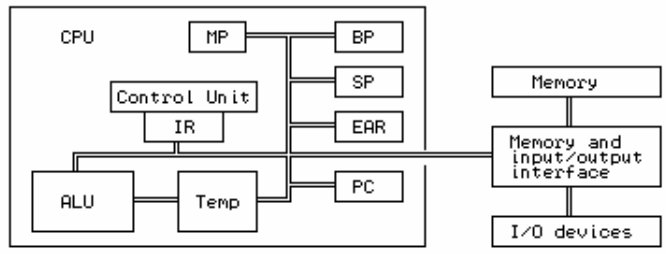

Figure 4.4 A simple stack-oriented CPU and computer

The symbols in this diagram refer to the following components of the machine

ALU is the *arithmetic logic unit* where arithmetic and logical operations are actually performed.

Temp is a set of 16-bit registers for holding intermediate results needed during arithmetic or logical operations. These registers cannot be accessed explicitly.

SP is the 16-bit *stack pointer*, a register that points to the area in memory utilized as the main stack.

BP is the 16-bit *base pointer*, a register that points to the base of an area of memory

within the stack, known as a *stack frame*, which is used to store variables.

`MP` is the 16-bit *mark stack pointer*, a register used in handling procedure calls, whose use will become apparent only in later chapters.

`IR` is the 16-bit *instruction register*, in which is held the instruction currently being executed.

`PC` is the 16-bit *program counter*, which contains the address in memory of the instruction that is the next to be executed.

`EAR` is the *effective address* register, which contains the address in memory of the data that is being manipulated by the current instruction.

A programmer's model of the machine is suggested by declarations like

```
CONST
  MemSize = 512;
TYPE
  ADDRESS = CARDINAL [0 .. MemSize - 1];
  PROCESSOR = RECORD
    IR : OPCODES;
    BP, MP, SP, PC : ADDRESS;
  END;
TYPE STATUS = (running, finished,
               badMem, badData,
               noData, divZero,
               badOP);
VAR
  CPU : PROCESSOR;
  Mem : ARRAY ADDRESS OF INTEGER;
  PS : STATUS;
```

```
const int MemSize = 512;
typedef short address;
struct processor {
  opcodes ir;
  address bp, mp, sp, pc;
};

typedef enum { running, finished,
  badmem, baddata, nodata,
  divzero, badop
} status;

processor cpu;
int mem[MemSize];
status ps;
```

For simplicity we shall assume that the code is stored in the low end of memory, and that the top part of memory is used as the stack for storing data. We shall assume that the topmost section of this stack is a *literal pool*, in which are stored constants, such as literal character strings. Immediately below this pool is the *stack frame*, in which the static variables are stored. The rest of the stack is to be used for working storage. A typical memory layout might be as shown in Figure 4.5, where the markers `CodeTop` and `StkTop` will be useful for providing memory protection in an emulated system.

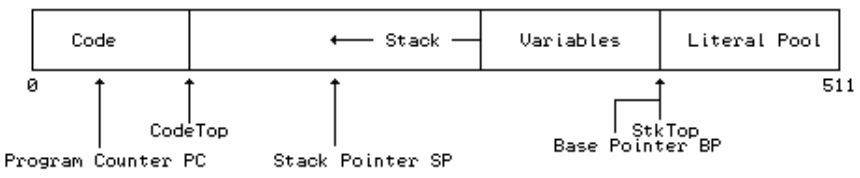

Figure 4.5 Usage of memory in the simple stack-oriented computer

We assume that the program loader will load the code at the bottom of memory (leaving the marker denoted by `CodeTop` pointing to the last word of code). It will also load the literals into the literal pool (leaving the marker denoted by `StkTop` pointing to the low end of this pool). It will go on to initialize both the stack pointer `SP` and base pointer `BP` to the value of `StkTop`. The first instruction in any program will have the responsibility of reserving further space on the stack for its variables, simply by decrementing the stack pointer `SP` by the number of words needed for these variables. A variable can be addressed by adding an offset to the base register `BP`. Since the stack "grows downwards" in memory, from high addresses towards low ones, these offsets will usually have

negative values.

4.4.2 Instruction set

A minimal set of operations for this machine is described informally below; in later chapters we shall find it convenient to add more opcodes to this set. We shall use the mnemonics introduced here to code programs for the machine in what appears to be a simple assembler language, albeit with addresses stipulated in absolute form.

Several of these operations belong to a category known as **zero address** instructions. Even though operands are clearly needed for operations such as addition and multiplication, the addresses of these are not specified by part of the instruction, but are implicitly derived from the value of the stack pointer `SP`. The two operands are assumed to reside on the top of the stack and just below the top; in our informal descriptions their values are denoted by `TOS` (for "top of stack") and `SOS` (for "second on stack"). A binary operation is performed by popping its two operands from the stack into (inaccessible) internal registers in the CPU, performing the operation, and then pushing the result back onto the stack. Such operations can be very economically encoded in terms of the storage taken up by the program code itself - the high density of stack-oriented machine code is another point in its favour so far as developing interpretive translators is concerned.

`ADD`		Pop `TOS` and `SOS`, add `SOS` to `TOS`, push sum to form new `TOS`
`SUB`		Pop `TOS` and `SOS`, subtract `TOS` from `SOS`, push result to form new `TOS`
`MUL`		Pop `TOS` and `SOS`, multiply `SOS` by `TOS`, push result to form new `TOS`
`DVD`		Pop `TOS` and `SOS`, divide `SOS` by `TOS`, push result to form new `TOS`
`EQL`		Pop `TOS` and `SOS`, push 1 to form new `TOS` if `SOS` = `TOS`, 0 otherwise
`NEQ`		Pop `TOS` and `SOS`, push 1 to form new `TOS` if `SOS` # `TOS`, 0 otherwise
`GTR`		Pop `TOS` and `SOS`, push 1 to form new `TOS` if `SOS` > `TOS`, 0 otherwise
`LSS`		Pop `TOS` and `SOS`, push 1 to form new `TOS` if `SOS` < `TOS`, 0 otherwise
`LEQ`		Pop `TOS` and `SOS`, push 1 to form new `TOS` if `SOS` <= `TOS`, 0 otherwise
`GEQ`		Pop `TOS` and `SOS`, push 1 to form new `TOS` if `SOS` >= `TOS`, 0 otherwise
`NEG`		Negate `TOS`
`STK`		Dump stack to output (useful for debugging)
`PRN`		Pop `TOS` and write it to the output as an integer value
`PRS`	`A`	Write the nul-terminated string that was stacked in the literal pool from `Mem[A]`
`NLN`		Write a newline (carriage-return-line-feed) sequence
`INN`		Read integer value, pop `TOS`, store the value that was read in `Mem[TOS]`
`DSP`	`A`	Decrement value of stack pointer `SP` by `A`
`LIT`	`A`	Push the integer value `A` onto the stack to form new `TOS`
`ADR`	`A`	Push the value `BP + A` onto the stack to form new `TOS`. (This value is conceptually the address of a variable stored at an offset `A` within the stack frame pointed to by the base register `BP`.)
`IND`		Pop `TOS` to yield `Size`; pop `TOS` and `SOS`; if `0` <= `TOS` < `Size` then subtract `TOS` from `SOS`, push result to form new `TOS`
`VAL`		Pop `TOS`, and push the value of `Mem[TOS]` to form new `TOS` (an operation we shall call *dereferencing*)
`STO`		Pop `TOS` and `SOS`; store `TOS` in `Mem[SOS]`
`HLT`		Halt
`BRN`	`A`	Unconditional branch to instruction `A`
`BZE`	`A`	Pop `TOS`, and branch to instruction `A` if `TOS` is zero
`NOP`		No operation

The instructions in the first group are concerned with arithmetic and logical operations, those in the second group afford I/O facilities, those in the third group allow for the access of data in memory by means of manipulating addresses and the stack, and those in the last group allow for control of flow of the program itself. The `IND` operation allows for array indexing with subscript range

checking.

As before, the I/O operations are not typical of real machines, but will allow us to focus on the principles of emulation without getting lost in the trivia and overheads of handling real I/O systems.

Exercises

4.14 How closely does the machine code for this stack machine resemble anything you have seen before?

4.15 Notice that there is a `BZE` operation, but not a complementary `BNZ` (one that would branch if `TOS` were non-zero). Do you suppose this is a serious omission? Are there any opcodes which have been omitted from the set above which you can foresee as being absolutely essential (or at least very useful) for defining a viable "integer" machine?

4.16 Attempt to write down a mathematically oriented version of the semantics of each of the machine instructions, as suggested by Exercise 4.3.

4.4.3 Specimen programs

As before, some samples of program code for the machine may help to clarify various points.

Example 4.4

To illustrate how the memory is allocated, consider a simple section of program that corresponds to high-level code of the form

```
X := 8; Write("Y = ", Y);
```

```
                    ; Example 4.4
 0 DSP    2         ; X is at Mem[CPU.BP-1], Y is at Mem[CPU.BP-2]
 2 ADR   -1         ; push address of X
 4 LIT    8         ; push 8
 6 STO              ;        X := 8
 7 STK              ; dump stack to look at it
 8 PRS   'Y = '     ;        Write string "Y = "
10 ADR   -2         ; push address of Y
12 VAL              ; dereference
13 PRN              ;        Write integer Y
14 HLT              ; terminate execution
```

This would be stored in memory as

```
 DSP  2  ADR -1  LIT  8  STO STK PRS 510 ADR -2  VAL PRN HLT
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14

 ...   (Y)   (X)   0   ' '  '='  ' '  'Y'   0
       504   505  506  507  508  509  510  511
```

Immediately after loading this program (and before executing the `DSP` instruction), the program counter `PC` would have the value 0, while the base register `BP` and stack pointer `SP` would each have the value 506.

Example 4.5

Example 4.4 scarcely represents the epitome of the programmer's art! A more ambitious program follows, as a translation of the simple algorithm

```
BEGIN
  Y := 0;
  REPEAT  READ(X);  Y := X + Y   UNTIL X = 0;
  WRITE('Total is ', Y);
END
```

This would require a stack frame of size two to contain the variables X and Y. The machine code might read

```
                    ;  Example 4.5
 0 DSP    2         ;  X is at Mem[CPU.BP-1], Y is at Mem[CPU.BP-2]
 2 ADR   -2         ;  push address of Y (CPU.BP-2) on stack
 4 LIT    0         ;  push 0 on stack
 6 STO              ;  store 0 as value of Y
 7 ADR   -1         ;  push address of X (CPU.BP-1) on stack
 9 INN              ;  read value, store on X
10 ADR   -2         ;  push address of Y on stack
12 ADR   -1         ;  push address of X on stack
14 VAL              ;  dereference - value of X now on stack
15 ADR   -2         ;  push address of Y on stack
17 VAL              ;  dereference - value of Y now on stack
18 ADD              ;  add X to Y
19 STO              ;  store result as new value of Y
20 ADR   -1         ;  push address of X on stack
22 VAL              ;  dereference - value of X now on stack
23 LIT    0         ;  push constant 0 onto stack
25 EQL              ;  check equality
26 BZE    7         ;  branch if X # 0
28 PRS   'Total is' ;  label output
30 ADR   -2         ;  push address of Y on stack
32 VAL              ;  dereference - value of Y now on stack
33 PRN              ;  write result
34 HLT              ;  terminate execution
```

Exercises

4.17 Would you write code anything like that given in Example 4.5 if you had to translate the corresponding algorithm into a familiar ASSEMBLER language directly?

4.18 How difficult would it be to hand translate programs written in this stack machine code into your favourite ASSEMBLER ?

4.19 Use the stack language (and, in due course, its interpreter) to write and test the simple programs suggested in Exercises 4.6.

4.4.4 An emulator for the stack machine

Once again, to emulate this machine by means of a program written in Modula-2 or C++, it will be convenient to define an interface to the machine by means of a definition module or appropriate class. As in the case of the accumulator machine, the main exported facility is a routine to perform the emulation itself, but for expediency we shall export further entities that make it easy to develop an assembler, compiler, or loader that will leave pseudo-code directly in memory after translation of some source code.

```
const int STKMC_memsize = 512;  // Limit on memory

// machine instructions - order is significant
enum STKMC_opcodes {
  STKMC_adr, STKMC_lit, STKMC_dsp, STKMC_brn, STKMC_bze, STKMC_prs, STKMC_add,
  STKMC_sub, STKMC_mul, STKMC_dvd, STKMC_eql, STKMC_neq, STKMC_lss, STKMC_geq,
  STKMC_gtr, STKMC_leq, STKMC_neg, STKMC_val, STKMC_sto, STKMC_ind, STKMC_stk,
  STKMC_hlt, STKMC_inn, STKMC_prn, STKMC_nln, STKMC_nop, STKMC_nul
};
```

```
typedef enum {
  running, finished, badmem, baddata, nodata, divzero, badop, badind
} status;
typedef int STKMC_address;

class STKMC {
  public:
    int mem[STKMC_memsize];  // virtual machine memory

    void listcode(char *filename, STKMC_address codelen);
    // Lists the codelen instructions stored in mem on named output file

    void emulator(STKMC_address initpc, STKMC_address codelen,
                  STKMC_address initsp, FILE *data, FILE *results,
                  bool tracing);
    // Emulates action of the codelen instructions stored in mem, with
    // program counter initialized to initpc, stack pointer initialized to
    // initsp.  data and results are used for I/O.  Tracing at the code level
    // may be requested

    void interpret(STKMC_address codelen, STKMC_address initsp);
    // Interactively opens data and results files.  Then interprets the
    // codelen instructions stored in mem, with stack pointer initialized
    // to initsp

    STKMC_opcodes opcode(char *str);
    // Maps str to opcode, or to STKMC_nul if no match can be found

    STKMC();
    // Initializes stack machine
};
```

The emulator itself has to model the typical *fetch-execute* cycle of an actual machine. This is easily achieved as before, and follows an almost identical pattern to that used for the other machine. A full implementation is to be found on the accompanying diskette; only the important parts are listed here for the reader to study:

```
bool STKMC::inbounds(int p)
// Check that memory pointer p does not go out of bounds.  This should not
// happen with correct code, but it is just as well to check
{ if (p < stackmin || p >= STKMC_memsize) ps = badmem;
  return (ps == running);
}

void STKMC::stackdump(STKMC_address initsp, FILE *results, STKMC_address pcnow)
// Dump data area - useful for debugging
{ int online = 0;
  fprintf(results, "\nStack dump at %4d", pcnow);
  fprintf(results, " SP:%4d BP:%4d SM:%4d\n", cpu.sp, cpu.bp, stackmin);
  for (int l = stackmax - 1; l >= cpu.sp; l--)
  { fprintf(results, "%7d:%5d", l, mem[l]);
    online++; if (online % 6 == 0) putc('\n', results);
  }
  putc('\n', results);
}

void STKMC::trace(FILE *results, STKMC_address pcnow)
// Simple trace facility for run time debugging
{ fprintf(results, " PC:%4d BP:%4d SP:%4d TOS:", pcnow, cpu.bp, cpu.sp);
  if (cpu.sp < STKMC_memsize)
    fprintf(results, "%4d", mem[cpu.sp]);
  else
    fprintf(results, "????");
  fprintf(results, " %s", mnemonics[cpu.ir]);
  switch (cpu.ir)
  { case STKMC_adr:
    case STKMC_prs:
    case STKMC_lit:
    case STKMC_dsp:
    case STKMC_brn:
    case STKMC_bze:
      fprintf(results, "%7d", mem[cpu.pc]); break;
    // no default needed
  }
  putc('\n', results);
}

void STKMC::postmortem(FILE *results, STKMC_address pcnow)
// Report run time error and position
{ putc('\n', results);
```

```
  switch (ps)
  { case badop:    fprintf(results, "Illegal opcode"); break;
    case nodata:   fprintf(results, "No more data"); break;
    case baddata:  fprintf(results, "Invalid data"); break;
    case divzero:  fprintf(results, "Division by zero"); break;
    case badmem:   fprintf(results, "Memory violation"); break;
    case badind:   fprintf(results, "Subscript out of range"); break;
  }
  fprintf(results, " at %4d\n", pcnow);
}

void STKMC::emulator(STKMC_address initpc, STKMC_address codelen,
                     STKMC_address initsp, FILE *data, FILE *results,
                     bool tracing)
{ STKMC_address pcnow;  // current program counter
  stackmax = initsp;
  stackmin = codelen;
  ps = running;
  cpu.sp = initsp;
  cpu.bp = initsp;      // initialize registers
  cpu.pc = initpc;      // initialize program counter
  do
  { pcnow = cpu.pc;
    if (unsigned(mem[cpu.pc]) > int(STKMC_nul)) ps = badop;
    else
    { cpu.ir = STKMC_opcodes(mem[cpu.pc]); cpu.pc++;  // fetch
      if (tracing) trace(results, pcnow);
      switch (cpu.ir)                                 // execute
      { case STKMC_adr:
          cpu.sp--;
          if (inbounds(cpu.sp))
            { mem[cpu.sp] = cpu.bp + mem[cpu.pc]; cpu.pc++; }
          break;
        case STKMC_lit:
          cpu.sp--;
          if (inbounds(cpu.sp)) { mem[cpu.sp] = mem[cpu.pc]; cpu.pc++; }
          break;
        case STKMC_dsp:
          cpu.sp -= mem[cpu.pc];
          if (inbounds(cpu.sp)) cpu.pc++;
          break;
        case STKMC_brn:
          cpu.pc = mem[cpu.pc]; break;
        case STKMC_bze:
          cpu.sp++;
          if (inbounds(cpu.sp))
          { if (mem[cpu.sp - 1] == 0) cpu.pc = mem[cpu.pc]; else cpu.pc++; }
          break;
        case STKMC_prs:
          if (tracing) fputs(BLANKS, results);
          int loop = mem[cpu.pc];
          cpu.pc++;
          while (inbounds(loop) && mem[loop] != 0)
            { putc(mem[loop], results); loop--; }
          if (tracing) putc('\n', results);
          break;
        case STKMC_add:
          cpu.sp++;
          if (inbounds(cpu.sp)) mem[cpu.sp] += mem[cpu.sp - 1];
          break;
        case STKMC_sub:
          cpu.sp++;
          if (inbounds(cpu.sp)) mem[cpu.sp] -= mem[cpu.sp - 1];
          break;
        case STKMC_mul:
          cpu.sp++;
          if (inbounds(cpu.sp)) mem[cpu.sp] *= mem[cpu.sp - 1];
          break;
        case STKMC_dvd:
          cpu.sp++;
          if (inbounds(cpu.sp))
          { if (mem[cpu.sp - 1] == 0)
              ps = divzero;
            else
              mem[cpu.sp] /= mem[cpu.sp - 1];
          }
          break;
        case STKMC_eql:
          cpu.sp++;
          if (inbounds(cpu.sp)) mem[cpu.sp] = (mem[cpu.sp] == mem[cpu.sp - 1]);
          break;
        case STKMC_neq:
          cpu.sp++;
```

```
          if (inbounds(cpu.sp)) mem[cpu.sp] = (mem[cpu.sp] != mem[cpu.sp - 1]);
          break;
        case STKMC_lss:
          cpu.sp++;
          if (inbounds(cpu.sp)) mem[cpu.sp] = (mem[cpu.sp] < mem[cpu.sp - 1]);
          break;
        case STKMC_geq:
          cpu.sp++;
          if (inbounds(cpu.sp)) mem[cpu.sp] = (mem[cpu.sp] >= mem[cpu.sp - 1]);
          break;
        case STKMC_gtr:
          cpu.sp++;
          if (inbounds(cpu.sp)) mem[cpu.sp] = (mem[cpu.sp] > mem[cpu.sp - 1]);
          break;
        case STKMC_leq:
          cpu.sp++;
          if (inbounds(cpu.sp)) mem[cpu.sp] = (mem[cpu.sp] <= mem[cpu.sp - 1]);
          break;
        case STKMC_neg:
          if (inbounds(cpu.sp)) mem[cpu.sp] = -mem[cpu.sp];
          break;
        case STKMC_val:
          if (inbounds(cpu.sp) && inbounds(mem[cpu.sp]))
            mem[cpu.sp] = mem[mem[cpu.sp]];
          break;
        case STKMC_sto:
          cpu.sp++;
          if (inbounds(cpu.sp) && inbounds(mem[cpu.sp]))
            mem[mem[cpu.sp]] = mem[cpu.sp - 1];
          cpu.sp++;
          break;
        case STKMC_ind:
          if ((mem[cpu.sp + 1] < 0) || (mem[cpu.sp + 1] >= mem[cpu.sp]))
            ps = badind;
          else
          { cpu.sp += 2;
            if (inbounds(cpu.sp)) mem[cpu.sp] -= mem[cpu.sp - 1];
          }
          break;
        case STKMC_stk:
          stackdump(initsp, results, pcnow); break;
        case STKMC_hlt:
          ps = finished; break;
        case STKMC_inn:
          if (inbounds(cpu.sp) && inbounds(mem[cpu.sp]))
          { if (fscanf(data, "%d", &mem[mem[cpu.sp]]) == 0)
              ps = baddata;
            else
              cpu.sp++;
          }
          break;
        case STKMC_prn:
          if (tracing) fputs(BLANKS, results);
          cpu.sp++;
          if (inbounds(cpu.sp)) fprintf(results, " %d", mem[cpu.sp - 1]);
          if (tracing) putc('\n', results);
          break;
        case STKMC_nln:
          putc('\n', results); break;
        case STKMC_nop:
          break;
        default:
          ps = badop; break;
      }
    }
  } while (ps == running);
  if (ps != finished) postmortem(results, pcnow);
}
```

We should remark that there is rather more error-checking code in this interpreter than we should like. This will detract from the efficiency of the interpreter, but is code that is probably very necessary when testing the system.

Exercises

4.20 Can you think of ways in which this interpreter can be improved, both as regards efficiency, and user friendliness? In particular, try adding debugging aids over and above the simple stack dump already provided. Can you think of any ways in which it could be made to detect infinite loops in a user program, or to allow itself to be manually interrupted by an irate or frustrated user?

4.21 The interpreter attempts to prevent corruption of the memory by detecting when the machine registers go out of bounds. The implementation above is not totally foolproof so, as a useful exercise, improve on it. One might argue that correct code will never cause such corruption to occur, but if one attempts to write stack machine code by hand, it will be found easy to "push" without "popping" or *vice versa*, and so the checks are very necessary.

4.22 The interpreter checks for division by zero, but does no other checking that arithmetic operations will stay within bounds. Improve it so that it does so, bearing in mind that one has to predict overflow, rather than wait for it to occur.

4.23 As an alternative, extend the machine so that overflow detection does not halt the program, but sets an overflow flag in the processor. Provide operations whereby the programmer can check this flag and take whatever action he or she deems appropriate.

4.24 One of the advantages of an emulated machine is that it is usually very easy to extend it (provided the host language for the interpreter can support the features required). Try introducing two new operations, say `INC` and `PRC`, which will read and print single character data. Then rework those of Exercises 4.6 that involve characters.

4.25 If you examine the code in Examples 4.4 and 4.5 - and in the solutions to Exercises 4.6 - you will observe that the sequences

```
ADR x
VAL
```

and

```
ADR x
(calculations)
STO
```

are very common. Introduce and implement two new operations

```
PSH  A     Push   Mem[CPU.BP + A]   onto stack to form new  TOS
POP  A     Pop   TOS   and assign   Mem[CPU.BP + A] := TOS
```

Then rework some of Exercise 4.6 using these facilities, and comment on the possible advantages of having these new operations available.

4.26 As a further variation on the emulated machine, develop a variation where the branch instructions are "relative" rather than "absolute". This makes for rather simpler transition to relocatable code.

4.27 Is it possible to accomplish Boolean (NOT, AND and OR) operations using the current instruction set? If not, how would you extend the instruction set to incorporate these? If they are not strictly necessary, would they be useful additions anyway?

4.28 As yet another alternative, suppose the machine had a set of condition flags such as `Z` and `P`, similar to those used in the single-accumulator machine of the last section. How would the instruction set and the emulator need to be changed to use these? Would their presence make it

easier to write programs, particularly those that need to evaluate complex Boolean expressions?

4.4.5 A minimal assembler for the machine

To be able to use this system we must, of course, have some way of loading or assembling code into memory. An assembler might conveniently be developed using the following interface, very similar to that used for the single- accumulator machine.

```
class STKASM {
  public:
    STKASM(char *sourcename, STKMC *M);
    // Opens source file from supplied sourcename

    ~STKASM();
    // Closes source file

    void assemble(bool &errors, STKMC_address &codetop,
                  STKMC_address &stktop);
    // Assembles source code from an input file and loads codetop
    // words of code directly into memory mem[0 .. codetop-1],
    // storing strings in the string pool at the top of memory in
    // mem[stktop .. STKMC_memsize-1].
    //
    // Returns
    //    codetop = number of instructions assembled and stored
    //              in mem[0] .. mem[codetop - 1]
    //    stktop  = 1 + highest byte in memory available
    //              below string pool in mem[stktop] .. mem[STK_memsize-1]
    //    errors  = true if erroneous instruction format detected
    // Instruction format :
    //    Instruction  = [Label] Opcode [AddressField] [Comment]
    //    Label        = Integer
    //    Opcode       = STKMC_Mnemonic
    //    AddressField = Integer | 'String'
    //    Comment      = String
    //
    // A string AddressField may only be used with a PRS opcode
    // Instructions are supplied one to a line; terminated at end of input file
};
```

This interface would allow us to develop sophisticated assemblers without altering the rest of the system - merely the implementation. In particular we can write a load-and-go assembler/interpreter very easily, using essentially the same system as was suggested in section 4.3.5.

The objective of this chapter is to introduce the principles of machine emulation, and not to be too concerned about the problems of assembly. If, however, we confine ourselves to assembling code where the operations are denoted by their mnemonics, but all the addresses and offsets are written in absolute form, as was done for Examples 4.4 and 4.5, a rudimentary assembler can be written relatively easily. The essence of this is described informally by an algorithm like

```
BEGIN
  CodeTop := 0;
  REPEAT
    SkipLabel;
    IF NOT EOF(SourceFile) THEN
      Extract(Mnemonic);
      Convert(Mnemonic, OpCode);
      Mem[CodeTop] := OpCode; Increment(CodeTop);
      IF OpCode = PRS THEN
        Extract(String); Store(String, Address);
        Mem[CodeTop] := Address; Increment(CodeTop);
      ELSIF OpCode in {ADR, LIT, DSP, BRN, BZE} THEN
        Extract(Address); Mem[CodeTop] := Address; Increment(CodeTop);
      END;
      IgnoreComments;
    END
  UNTIL EOF(SourceFile)
END
```

An implementation of this is to be found on the source diskette, where code is assumed to be

supplied to the machine in free format, one instruction per line. Comments and labels may be added, as in the examples given earlier, but these are simply ignored by the assembler. Since absolute addresses are required, any labels are more of a nuisance than they are worth.

Exercises

4.29 The assembler on the source diskette attempts some, but not much, error detection. Investigate how it could be improved.

4.30 The machine is rather wasteful of memory. Had we used a byte oriented approach we could have stored the code and the literal strings far more compactly. Develop an implementation that does this.

4.31 It might be deemed unsatisfactory to locate the literal pool in high memory. An alternative arrangement would be to locate it immediately above the executable code, on the lines of Figure 4.6. Develop a variation on the assembler (and, if necessary, the interpreter) to exploit this idea.

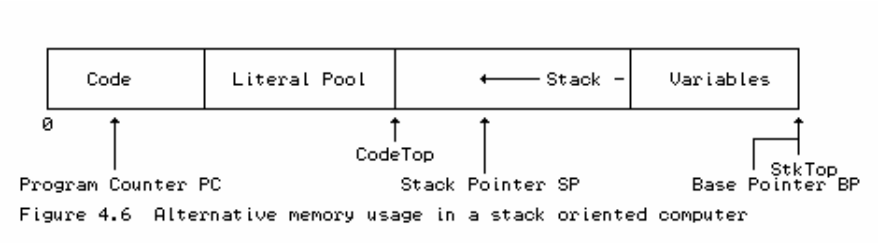

Figure 4.6 Alternative memory usage in a stack oriented computer

---

Further reading

Other descriptions of pseudo-machines and of stack machines are to be found in the books by Wakerly (1981), Brinch Hansen (1985), Wirth (1986, 1996), Watt (1993), and Bennett (1990).

The very comprehensive stack-based interpreter for the Zürich Pascal-P system is fully described in the book by Pemberton and Daniels (1982).