CHAPTER 3 (CPUs)

1

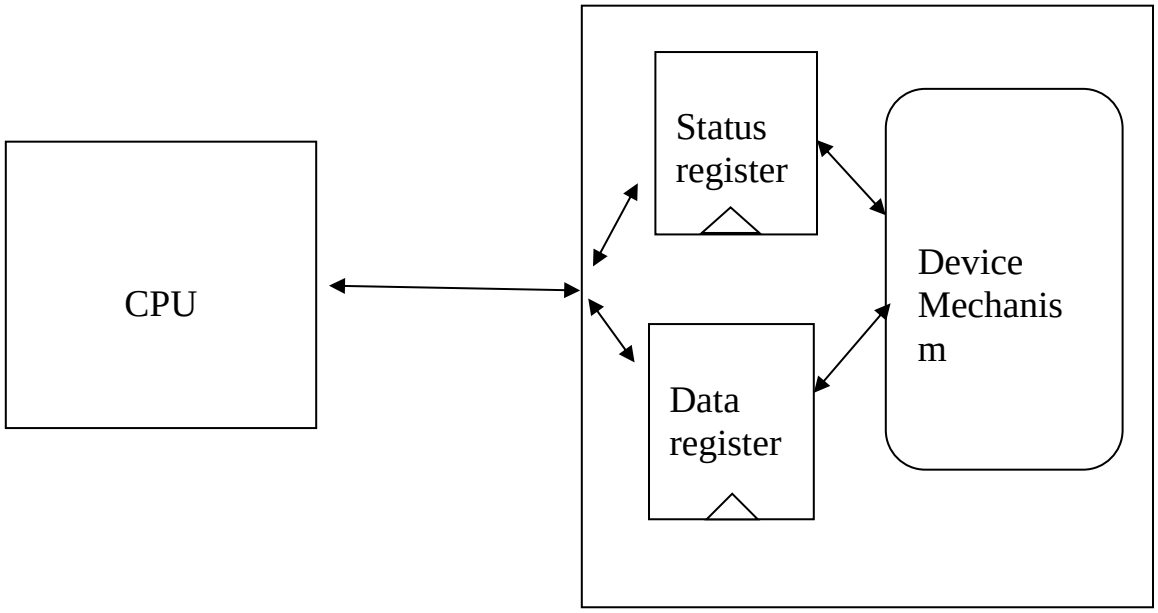


FIGURE 3.1
Structure of typical I/O device.

Input and Output Devices

The structure of a typical I/O device and its relationship to the CPU. The interface between the CPU and the device's internals (e.g., the rotating disk and read/write electronics in a disk drive) is a set of registers. The CPU talks to the device by reading and writing the registers. Devices typically have several registers:

■ Data registers hold values that are treated as data by the device, such as the data read or written by a disk.

■ Status registers provide information about the device's operation, such as whether the current transaction has completed.

2. Input and Output Primitives

Microprocessors can provide programming support for input and output in two ways: I/O instructions and memory-mapped I/O.

Example 3.1
Memory-mapped I/O on ARM

We can use the EQU pseudo-op to define a symbolic name for the memory location of our I/O device:

```
DEV1 EQU 0x1000
```

Given that name, we can use the following standard code to read and write the device register:

```
LDR r1, #DEV1    ;    set up device address
LDR r0, [r1]     ;    read DEV1
LDR r0, #8       ;    set up value to write
STR r0, [r1]     ;    write 8 to device
```

How can we directly write I/O devices in a high-level language like C? When we define and use a variable in C, the compiler hides the variable's address from us. But we can use pointers to manipulate addresses of I/O devices. The traditional names for functions that read and write arbitrary memory locations are peek and poke. The peek function can be written in C as:

```
int peek(char *location) {
        return *location;     /* de-reference location pointer */
}
```

The argument to peek is a pointer that is de-referenced by the C * operator to read the location. Thus, to read a device register we can write:

```
#define DEV1 0x1000
...
dev_status = peek(DEV1);          /* read device register */
```

The poke function can be implemented as:

```
void poke(char *location, char newval) {
```

```
    (*location) = newval;       /* write to location */
    }
```

To write to the status register, we can use the following code: poke(DEV1,8); /* write 8 to device register */ These functions can, of course, be used to read and write arbitrary memory locations, not just devices.

3. Busy-Wait I/O

The most basic way to use devices in a program is busy-wait I/O. Devices are typically slower than the CPU and may require many cycles to complete an operation. If the CPU is performing multiple operations on a single device, such as writing several characters to an output device, then it must wait for one operation to complete before starting the next one.

Example 3.2
Busy-wait I/O programming

We want to write a sequence of characters to an output device. The device has two registers: one for the character to be written and a status register. The status register's value is 1 when the device is busy writing and 0 when the write transaction has completed. We will use the peek and poke functions to write the busy-wait routine in C. First, we define symbolic names for the register addresses:

```
#define OUT_CHAR 0x1000          /* output device character register */ #define
OUT_STATUS 0x1001          /* output device status register */


char  *mystring = "Hello, world."          /* string to write */
char  *current_char;                       /* pointer to current position in
                                           string*/
current_char  =  mystring;                 /* point to head of string */
while (*current_char != `\ 0') {           /*  until  null  character  */
      poke(OUT_CHAR,*current_char);        /* send character to device */
      while (peek(OUT_STATUS) != 0);       /* keep checking status */
      current_char++;                      /* update character pointer */
}
```

>Peek - get the byte located at the specified memory address.
>Poke - set the memory byte at the specific address.

4. Interrupts Basics

The CPU could do useful work in parallel with the I/O transaction, such as:

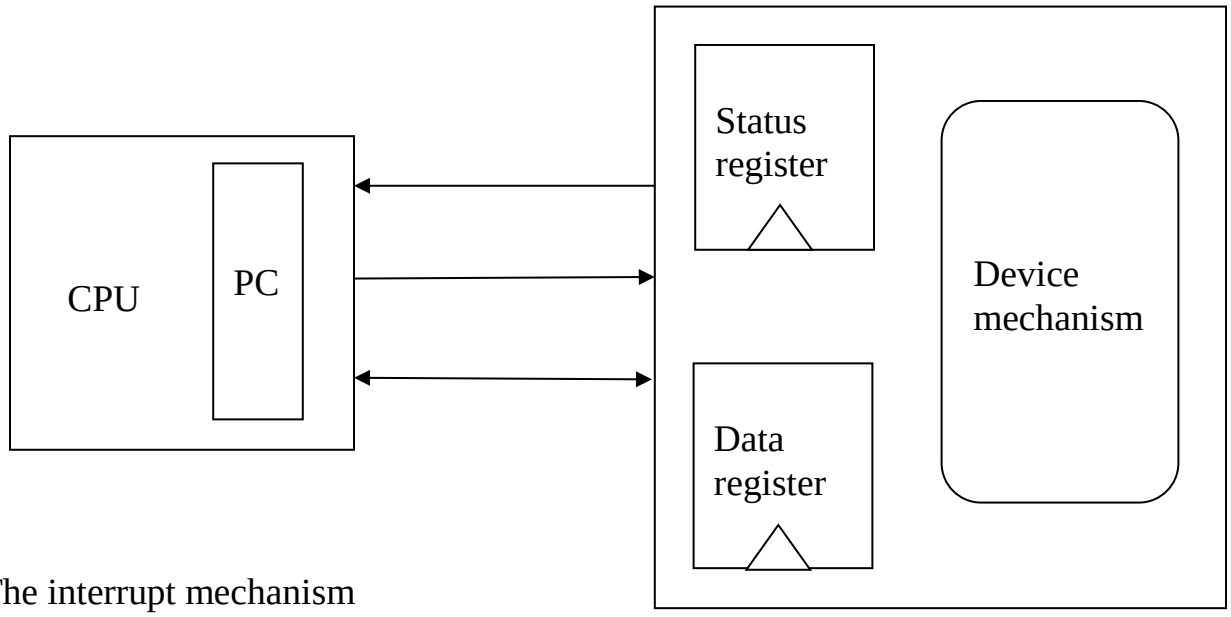
- computation, as in determining the next output to send to the device or processing the last input received, and
- control of other I/O devices.

To allow parallelism, we need to introduce new mechanisms into the CPU. The interrupt mechanism allows devices to signal the CPU and to force execution of a particular piece of code.

The interface between the CPU and I/O device includes the following signals for interrupting:

- the I/O device asserts the interrupt request signal when it wants service from the CPU; and
- the CPU asserts the interrupt acknowledge signal when it is ready to handle the I/O device's request.

The CPU may not be able to immediately service an interrupt request because it may be doing something else that must be finished first—for example, a program that talks to both a high-speed disk drive and a low-speed keyboard should be designed to finish a disk transaction before handling a keyboard interrupt.



The interrupt mechanism

5. Example 3.4

Copying characters from input to output with basic interrupts

We repeatedly read a character from an input device and write it to an output device. We assume that we can write C functions that act as interrupt handlers. Those handlers will work with the devices in much the same way as in busy-wait I/O by reading and writing status and data registers. The main difference is in handling the output—the interrupt signals that the character is done, so the handler does not have to do anything.

We will use a global variable achar for the input handler to pass the character to the foreground program. Because the foreground program doesn't know when an interrupt occurs, we also use a global Boolean variable, got char, to signal when a new character has been received. The code for the input and output handlers follows:

```
void  input_handler() {          /* get a character and put in global */
achar = peek(IN_DATA);          /* get character */
gotchar = TRUE;                  /* signal to main program */
poke(IN_STATUS,0);               /* reset status to initiate next transfer */
}
void output_handler() {          /* react to character being sent */
        /* don't have to do anything */
}
```

The main program is reminiscent of the busy-wait program. It looks at gotchar to check when a new character has been read and then immediately sends it out to the output device

```
main() {
        while (TRUE) {                          /* read then write forever */
                if (gotchar) {                          /* write a character */
                        poke(OUT_DATA,achar);           /* put character in device
                                                        */
                        poke(OUT_STATUS,1);             /* set status to initiate
                                                        write */
                        gotchar = FALSE;                /* reset flag */
}
        }
}
```

6. Priorities and Vectors

- **Interrupt priorities** allow the CPU to recognize some interrupts as more important than others, and

- **Interrupt vectors** allow the interrupting device to specify its handler.

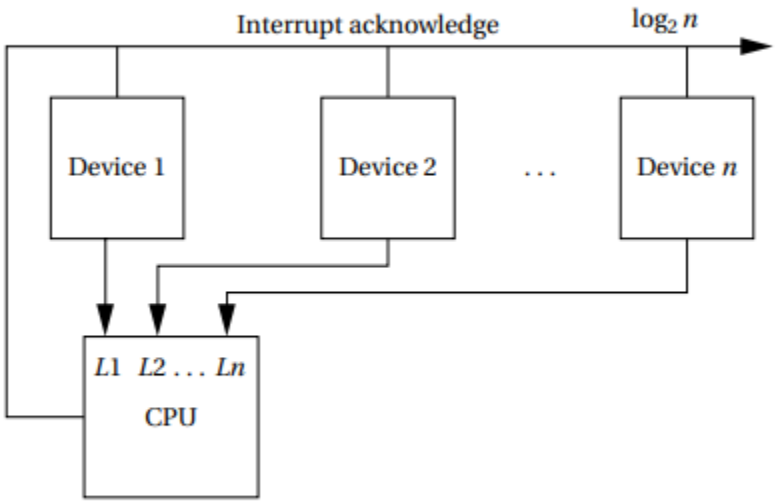


FIGURE 3.3
Prioritized device interrupts.

Prioritized device interrupts

7.Interrupt vectors

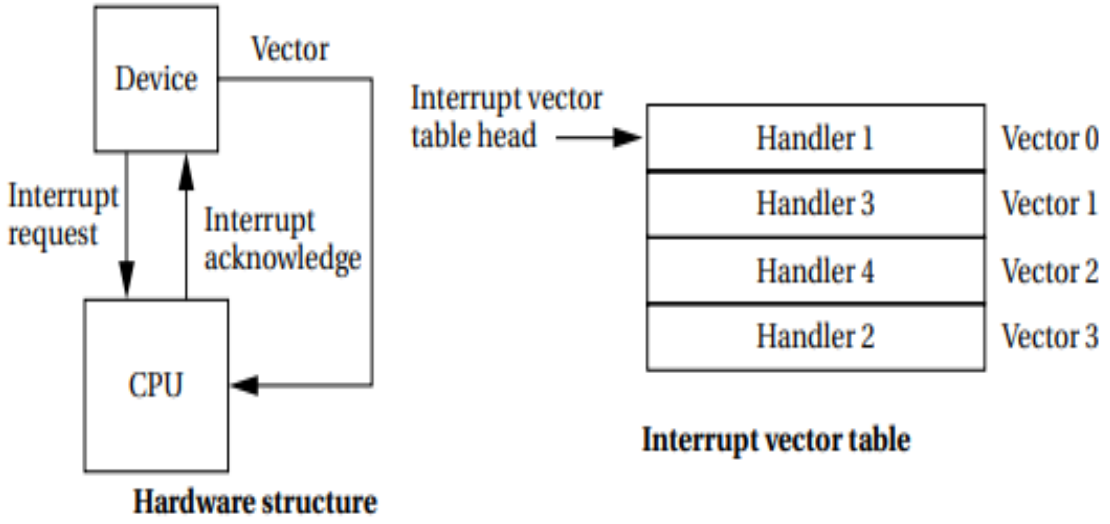


FIGURE 3.5

Interrupt vectors.

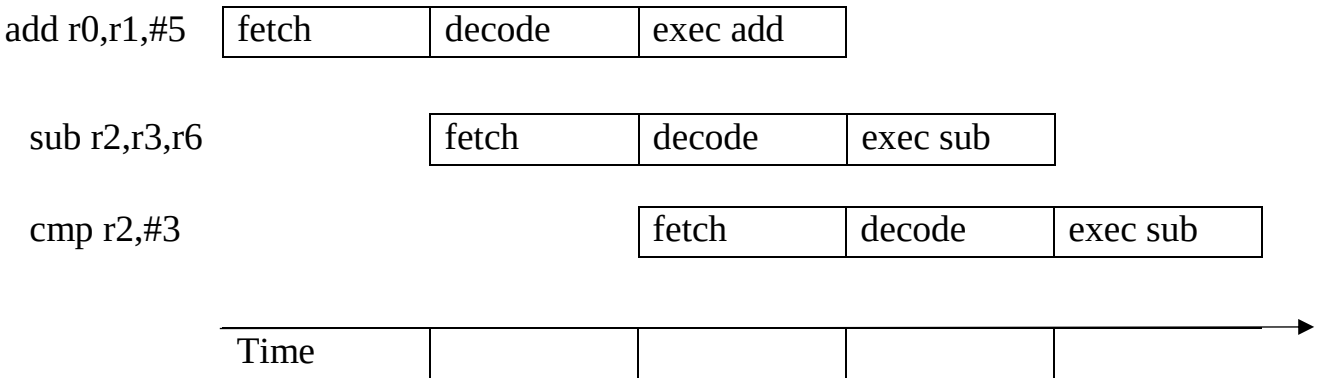
The complete interrupt handling process

1. **CPU** - The CPU checks for pending interrupts at the beginning of an instruction. It answers the highest-priority interrupt.
2. **Device** - The device receives the acknowledgment and sends the CPU its interrupt vector.
3. **CPU** - The CPU looks up the device handler address in the interrupt vector table using the vector as an index.
4. **Software** - The device driver may save additional CPU state. It then performs the required operations on the device. It then restores any saved state and executes the interrupt return instruction.
5. **CPU** - The interrupt return instruction restores the PC and other automatically saved states to return execution to the code that was interrupted.

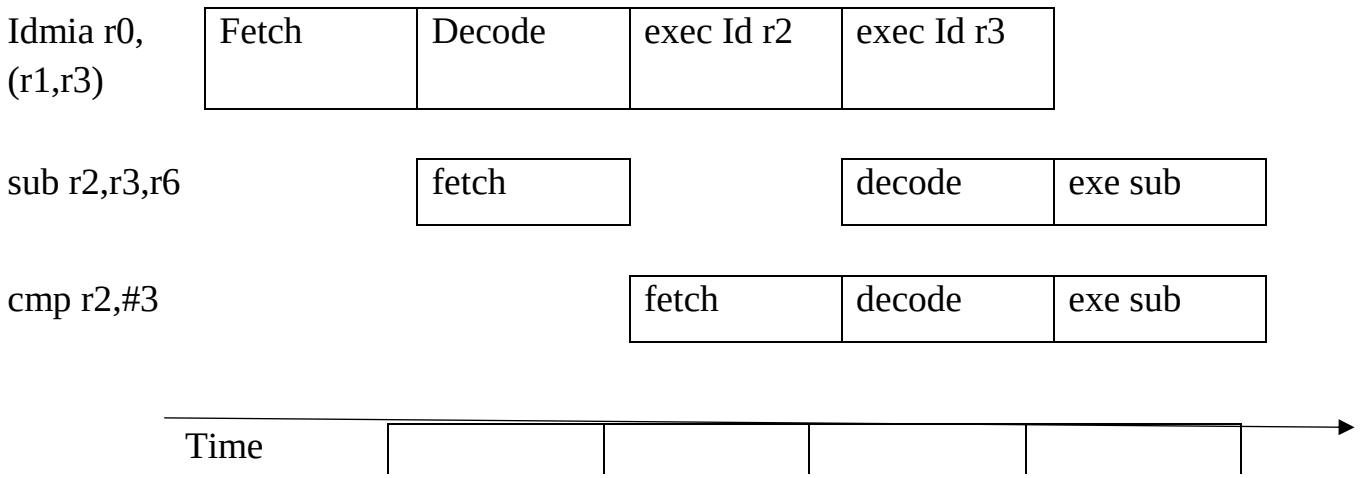
8. Pipeline

The ARM7 has a three-stage pipeline:

- Fetch the instruction is fetched from memory.
- Decode the instruction's opcode and operands are decoded to determine what function to perform.
- Execute the decoded instruction is executed.



Pipeline execution of ARM instructions



Pipelined execution of multicycle ARM instruction

9. Example 3.9

Execution time of a for loop on the ARM

We will use the C code for the FIR filter of Application Example 2.1:

```
for (i = 0, f = 0; i < N; i++)
        f=f+ c[i] * x[i];
```

We repeat the ARM code for this loop:

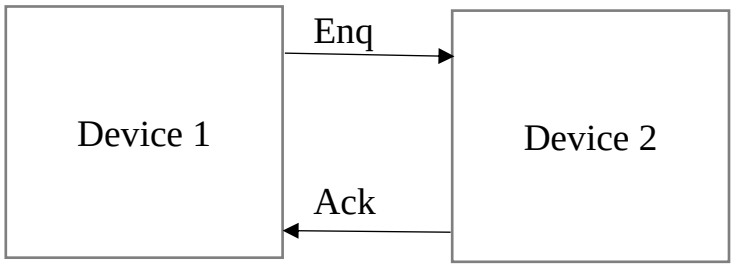
```
                loop initiation code
MOV r0,#0 ;             use r0 for i, set to 0
MOV r8,#0 ;             use a separate index for arrays
ADR r2,N ;              get address for N
LDR r1,[r2] ;           get value of N for loop termination test
MOV r2,#0 ;             use r2 for f, set to 0
ADR r3,c ;              load r3 with address of base of c array
ADR r5,x ;              load r5 with address of base of x array ; loop body loop
LDR r4,[r3,r8] ;        get value of c[i]
LDR r6,[r5,r8] ;        get value of x[i]
MUL r4,r4,r6 ;          compute c[i]*x[i]
ADD r2,r2,r4 ;          add into running sum f ; update loop counter and array
                        index
ADD r8,r8,#4 ;          add one word offset to array index
ADD r0,r0,#1 ;          add 1 to i ; test for exit
CMP r0,r1
BLT loop ;              if i < N, continue loop loopend...
```

Chapter (4)

1. Four-cycle handshake

The handshake ensures that when two devices want to communicate, one is ready to transmit and the other is ready to receive. The handshake uses a pair of wires dedicated to the handshake: **enq** (meaning enquiry) and **ack** (meaning acknowledge). Extra wires are used for the data transmitted during the handshake. The four cycles are described below.

1. Device 1 raises its output to signal an enquiry, which tells device 2 that it should get ready to listen for data. .
2. When device 2 is ready to receive, it raises its output to signal an acknowledgment. At this point, devices 1 and 2 can transmit or receive.
3. Once the data transfer is complete, device 2 lowers its output, signaling that it has received the data.
4. After seeing that ack has been released, device 1 lowers its output.



Structure

The four-cycle handshake

2. Typical microprocessor bus

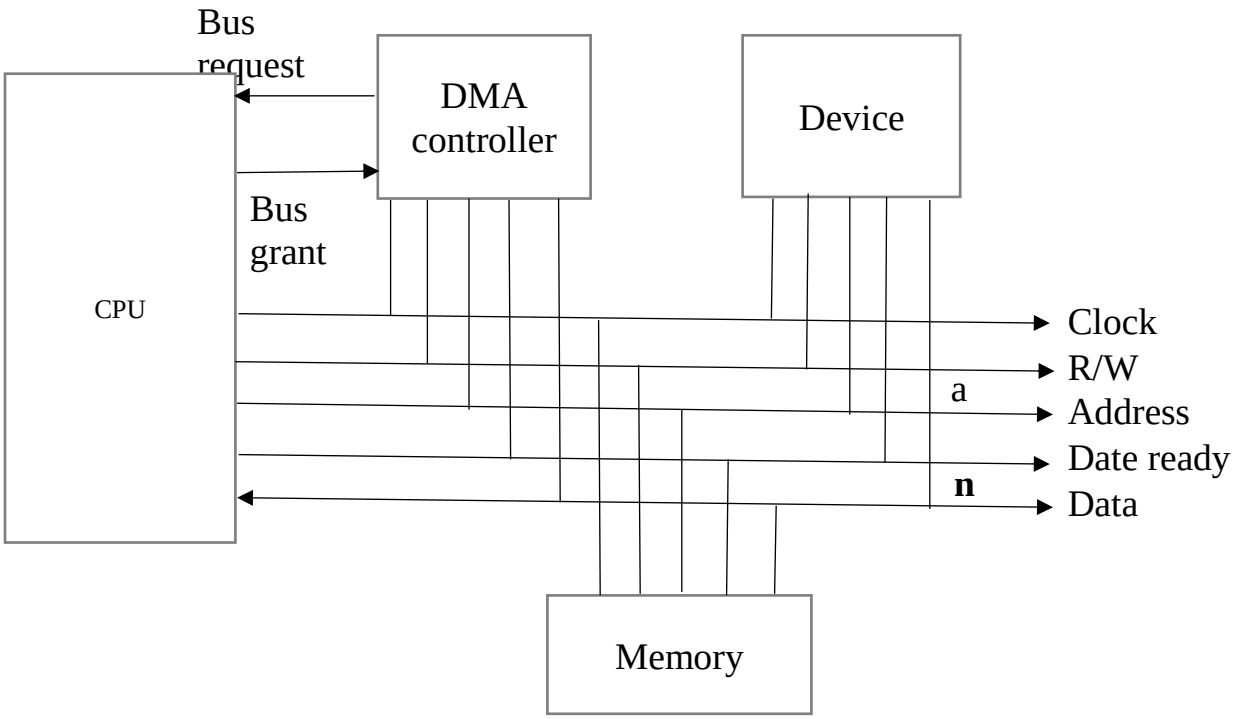
- Clock provides synchronization to the bus components,
- R/W is true when the bus is reading and false when the bus is writing,
- Address is an a-bit bundle of signals that transmits the address for an access,
- Data is an n-bit bundle of signals that can carry data to or from the CPU, and
- Data ready signals when the values on the data bundle are valid.

DRAW FIGURE 4.2

3. Direct memory access (DMA)

Direct memory access (DMA) is a bus operation that allows reads and writes not controlled by the CPU. A DMA transfer is controlled by a DMA controller, which requests control of the bus from the CPU. After gaining control, the DMA controller performs read and write operations directly between devices and memory.

- The bus request is an input to the CPU through which DMA controllers ask for ownership of the bus.
- The bus grant signals that the bus has been granted to the DMA controller.



A bus with a DMA controller

4. The CPU controls the DMA operation through registers in the DMA controller. A typical DMA controller includes the following three registers.

- A starting address register specifies where the transfer is to begin.
- A length register specifies the number of words to be transferred.
- A status register allows the DMA controller to be operated by the CPU.

5. Multiple bus system

- Higher-speed buses may provide wider data connections.
- A high-speed bus usually requires more expensive circuits and connectors. The cost of low-speed devices can be held down by using a lower-speed, lower-cost bus.
- The bridge may allow the buses to operate independently, thereby providing some parallelism in I/O operations.

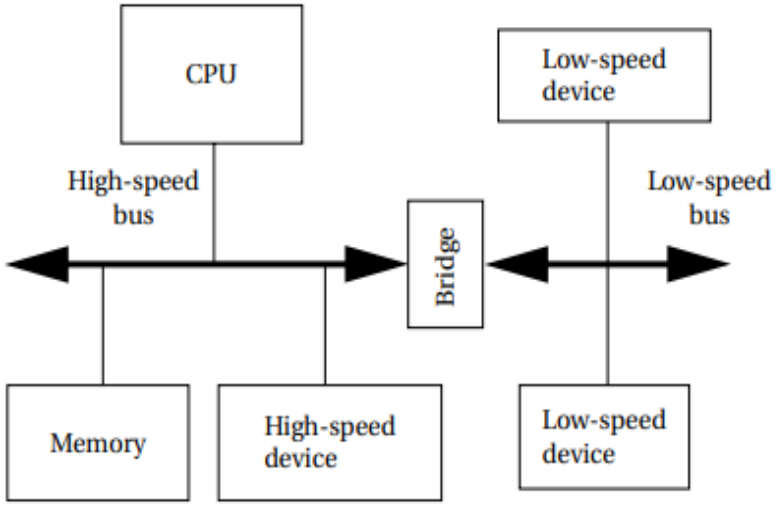


FIGURE 4.12

A multiple bus system.

9. Sequential and parallel schedules in a bus-based system

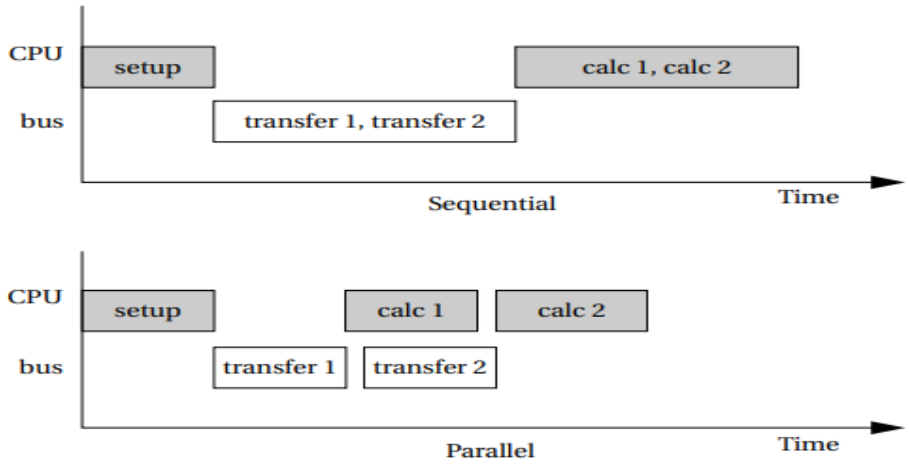
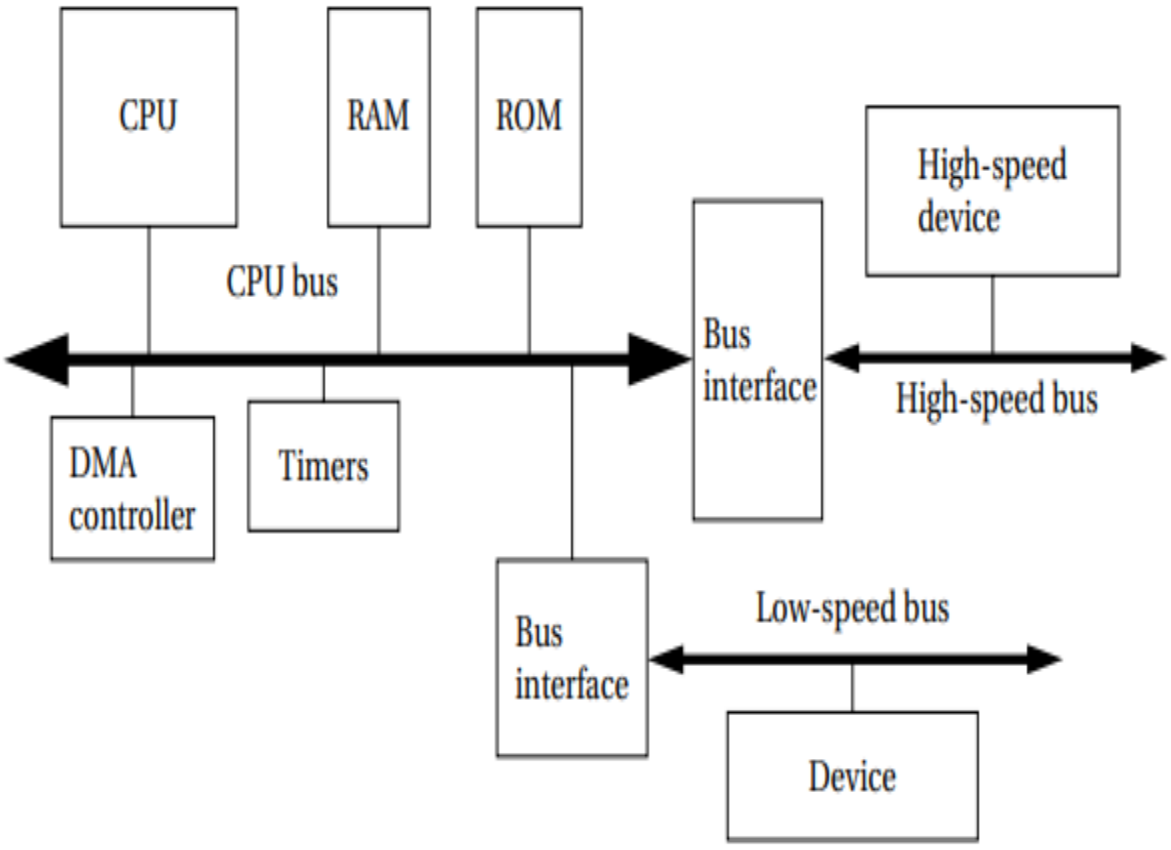


FIGURE 4.33

Sequential and parallel schedules in a bus-based system.

6. Hardware architecture of a typical PC

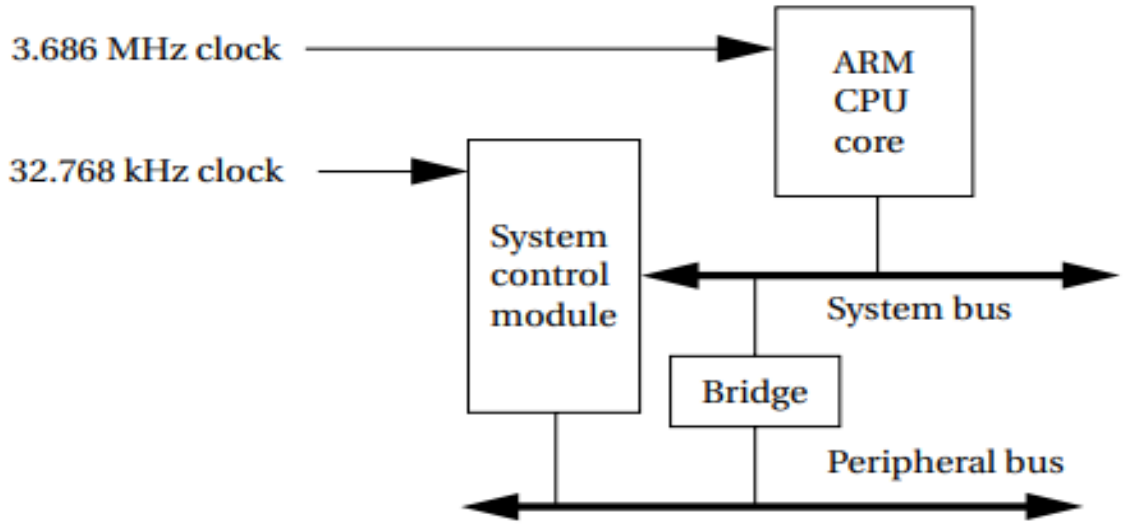
- The CPU provides basic computational facilities.
- RAM is used for program storage. ROM holds the boot program.
- A DMA controller provides DMA capabilities.
- Timers are used by the operating system for a variety of purposes.
- A high-speed bus, connected to the CPU bus through a bridge, allows fast devices to communicate efficiently with the rest of the system.
- A low-speed bus provides an inexpensive way to connect simpler devices and may be necessary for backward compatibility as well.



Hardware architecture of a typical PC

7. **Application Example 4.1**

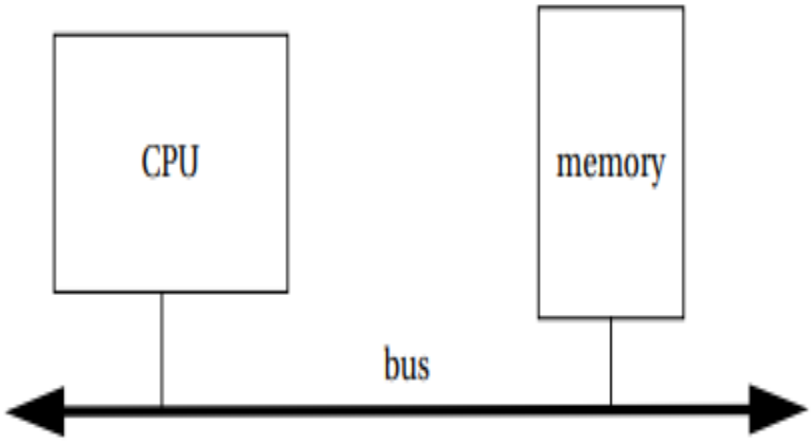
System organization of the Intel StrongARM SA-1100 and SA-1111
The StrongARM SA-1100 provides a number of functions besides the ARM CPU:



8. **Example 4.3**

Performance bottlenecks in a bus-based system.

Consider a simple bus-based system:



We want to transfer data between the CPU and the memory over the bus. We need to be able to read a 320*240 video frame into the CPU at the rate of 30 frames/s, for a total of 612,000 bytes/s. Which will be the bottleneck and limit system performance: the bus or the memory?

Let's assume that the bus has a 1-MHz clock rate (period of 10^{-6} sec) and is 2 bytes wide, with D =1 and O =3. This gives a total transfer time of

T_{basic} = (1 + 3) *612,000/ 2 = 1,224,000 cycles,

$$t = T_{basic} \cdot P = 1{,}224{,}000 \cdot 1 \times 10^{-6} = 1.224 \text{ sec.} \quad (4.6)$$

Since the total time to transfer one second's worth of frames is more than 1 s, the bus is not fast enough for our application.

The memory provides a burst mode with $B = 4$ but is only 4 bits wide, giving $W = 0.5$. For this memory, $D = 1$ and $O = 4$. The clock period for this memory is 10^{-7} s. Then

$$T_{mem} = (4 \cdot 1 + 4)\frac{612{,}000}{4 \cdot 0.5} = 2{,}448{,}000 \text{ cycles,} \quad (4.7)$$

$$t = T_{mem} \cdot P = 2{,}448{,}000 \cdot 1 \times 10^{-7} = 0.2448 \text{ sec} \quad (4.8)$$

The memory requires <1 s to transfer the 30 frames that must be transmitted in 1 s, so it is fast enough.

One way to explore design trade-offs is to build a spreadsheet:

Bus		Memory	
Clock period	1.00E – 06	Clock period	1.00E – 08
W	2	*W*	0.5
D	1	*D*	1
O	3	*O*	4
		B	4
N	612000	*N*	612000
T_{basic}	1224000	T_{mem}	2448000
t	1.22E + 00	*t*	2.45E – 02

Chapter (5)

1. MODELS OF PROGRAMS

Our fundamental model for programs is the control/data flow graph (CDFG).CDFG has constructs that model both data operations (arithmetic and other computations) and control operations (conditionals). Part of the power of the CDFG comes from its combination of control and data constructs.

```
w = a + b;
x = a − c;
y = x + d;
x = a + c;
z = y + e;
```

FIGURE 5.2
A basic block in C.

```
w = a + b;
x1 = a − c;
y = x1 + d;
x2 = a + c;
z = y + e;
```

FIGURE 5.3
The basic block in single-assignment form.

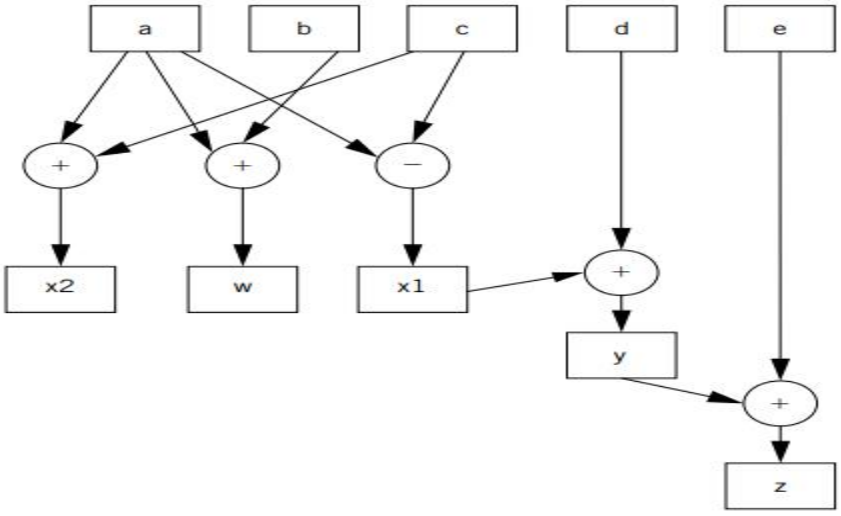


FIGURE 5.4
An extended data flow graph for our sample basic block.

2. Control/Data Flow Graphs

A CDFG uses a data flow graph as an element, adding constructs to describe control. In a basic CDFG, we have two types of nodes: decision nodes and data flow nodes. A data flow node encapsulates a complete data flow graph to represent a basic block.

```
for (i = 0; i < N; i++) { loop_body(); }
is equivalent to
i = 0;
while (i < N) { loop_body(); i++; }
if (cond1) basic_block_1();
else basic_block_2();
basic_block_3();
switch (test1) {
case c1: basic_block_4(); break;
case c2: basic_block_5(); break;
case c3: basic_block_6():break;}
```

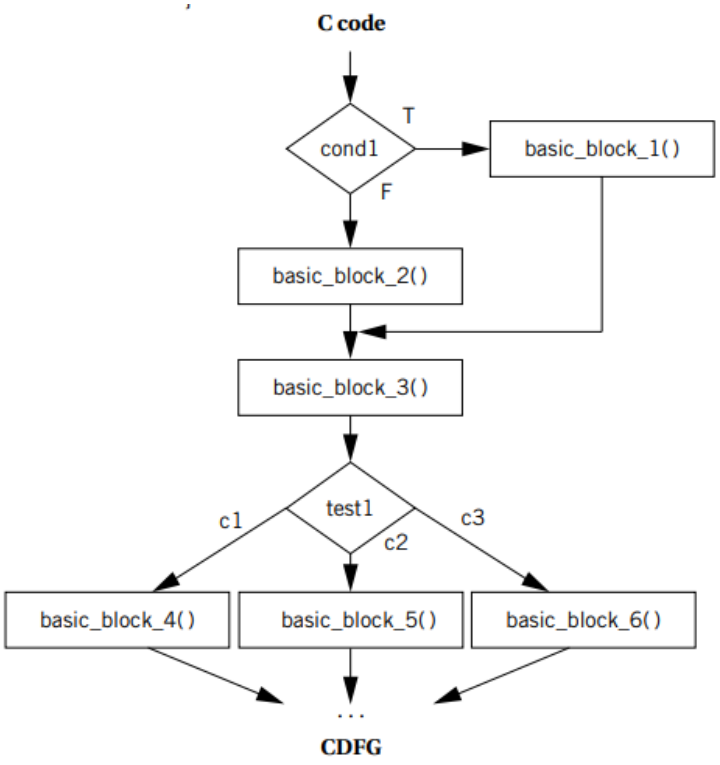


FIGURE 5.6
C code and its CDFG.

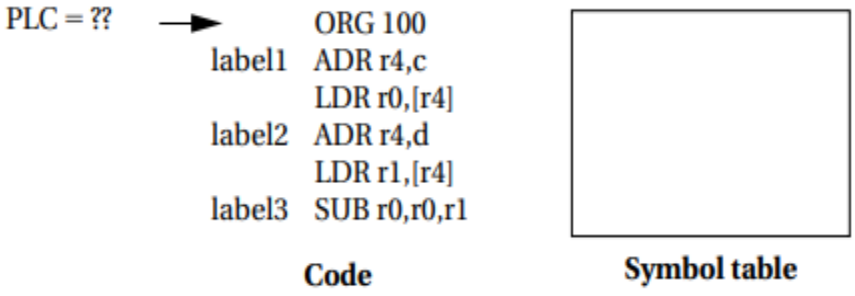
3. **Example 5.1**

Generating a symbol table

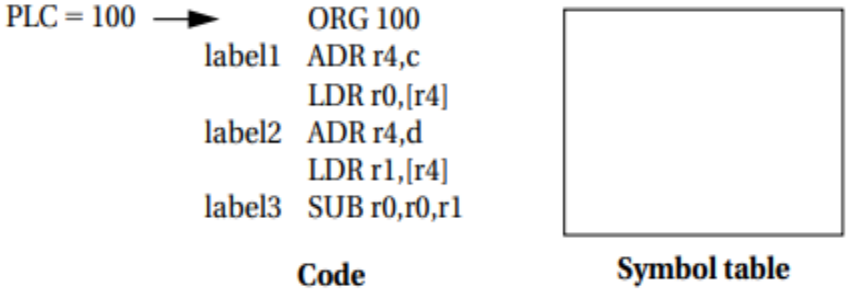
Let's use the following simple example of ARM assembly code:

```
ORG 100 label1
ADR r4, c
LDR r0, [r4] label2
ADR r4, d
LDR r1, [r4] label3
SUB r0, r0, r1
```

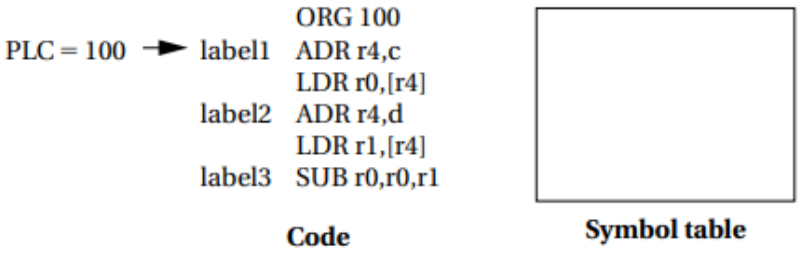
The initial ORG statement tells us the starting address of the program. To begin, let's initialize the symbol table to an empty state and put the PLC at the initial ORG statement.



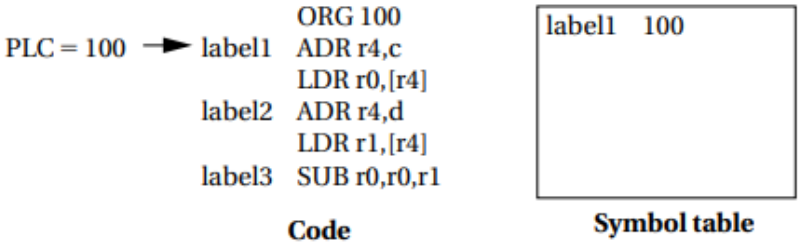
The PLC value shown is at the beginning of this step, before we have processed the ORG statement. The ORG tells us to set the PLC value to 100.



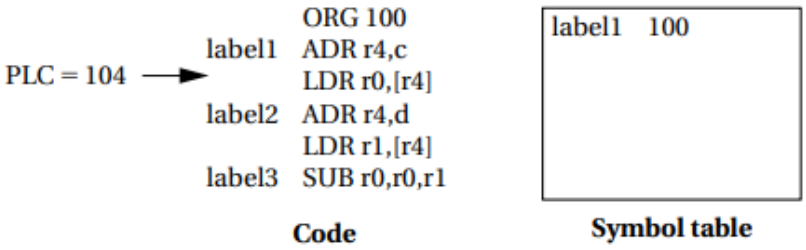
To process the next statement, we move the PLC to point to the next statement. But because the last statement was a pseudo-op that generates no memory values, the PLC value remains at 100.



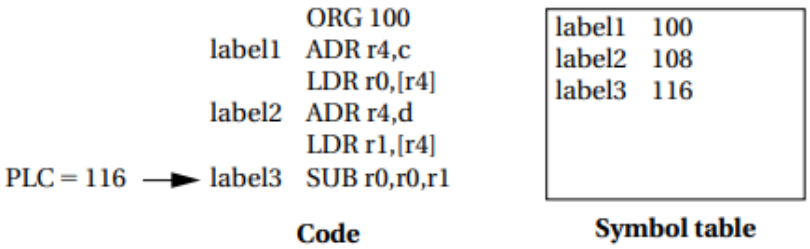
Because there is a label in this statement, we add it to the symbol table, taking its value from the current PLC value.



To process the next statement, we advance the PLC to point to the next line of the program and increment its value by the length in memory of the last line, namely, 4.



We continue this process as we scan the program until we reach the end, at which the state of the PLC and symbol table are as shown below.

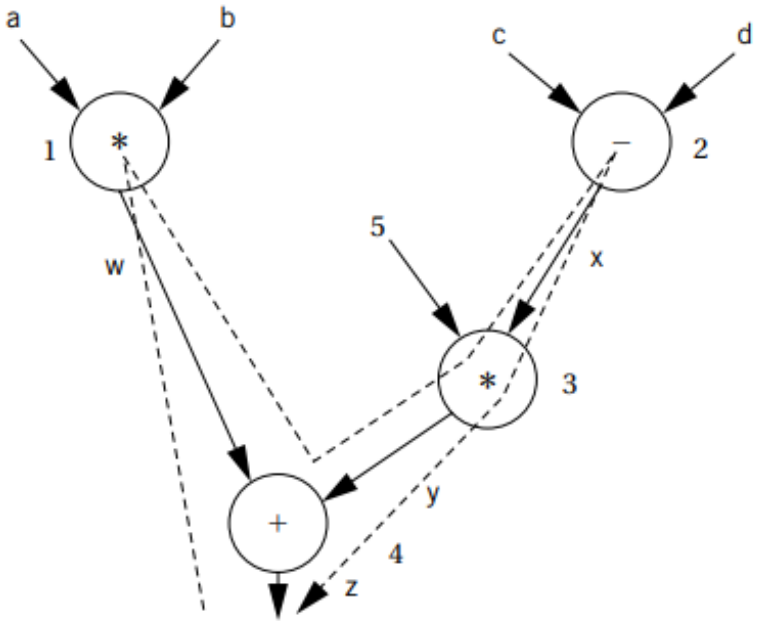


4. Example 5.2

Compiling an arithmetic expression

In the following arithmetic expression,

a * b + 5*(c – d)

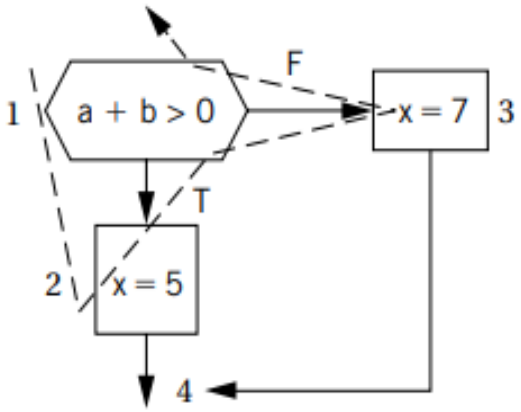



```
; operator  1  (+)
ADR r4, a          ; get address for a
MOV r1, [r4]       ; load a
ADR r4, b          ; get address for b
MOV r2, [r4]       ; load b
ADD r3, r1, r2     ; put w into r3; operator 2 (–)
ADR r4, c          ; get address for c
MOV r4, [r4]       ; load c
ADR r4, d          ; get address for d
MOV r5, [r4]       ; load d
SUB r6, r4, r5     ; put x into r6; operator 3 (*)
MUL r7, r6, #5     ; operator 3, puts y into r7; operator 4 (+)
ADD r8, r7, r3     ; operator 4, puts z into r8
```

5.Example 5.3
Generating code for a conditional

Consider the following C statement:

if (a + b > 0) x = 5; else x = 7;




```
ADR r5, a          ;     get address for a
LDR r1, [r5]       ;     load a
ADR r5, b          ;     get address for b
LDR r2, b          ;     load b
ADD r3, r1, r2
BLE label3         ;     true condition falls through branch;
; true case
        LDR r3, #5 ;     load constant
        ADR r5, x
        STR r3, [r5] ;   store value into x
        B stmtend  ;     done with the true case
;false case label3
LDR r3, #7         ;     load constant
ADR r5, x          ;     get address of x
STR r3, [r5]       ;     store value into x
Stmtend   ...
```

6.Program generation from compilation through loading

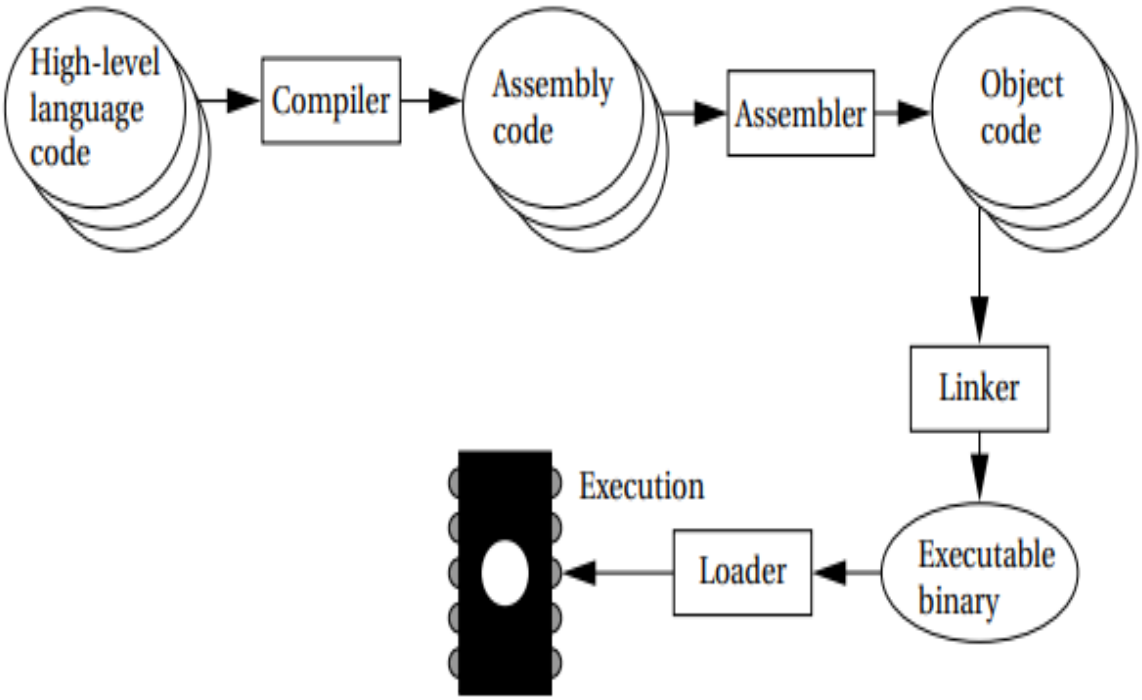


FIGURE 5.8

Program generation from compilation through loading.

Chapter (1)

1. What is an embedded computer system?

It is any device that includes a programmable computer but is not itself intended to be a general-purpose computer. Thus, a PC is not itself an embedded computing system, although PCs are often used to build embedded computing systems. But a fax machine or a clock built from a microprocessor is an embedded computing system.

This means that embedded computing system design is a useful skill for many types of product design. Automobiles, cell phones, and even household appliances make extensive use of microprocessors. Designers in many fields must be able to identify where microprocessors can be used, design a hardware platform with I/O devices that can support the required tasks, and implement software that performs the required processing.

2. Major level of abstraction in the design process

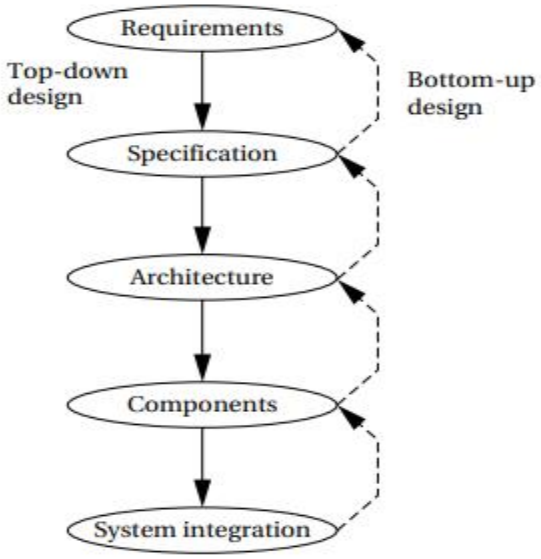


FIGURE 1.1
Major levels of abstraction in the design process.

3. A model train control system

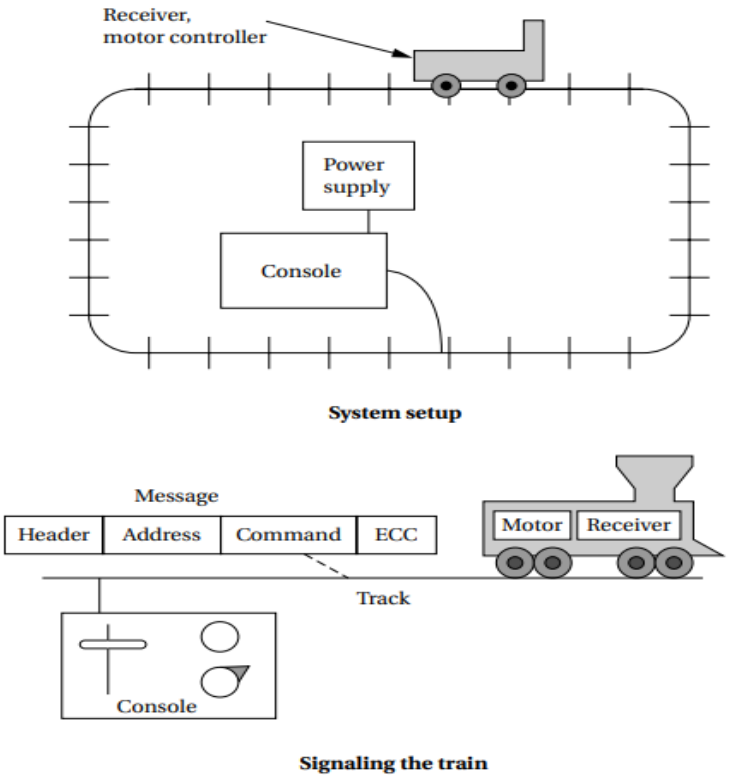


FIGURE 1.14
A model train control system.

Name	Model train controller
Purpose	Control speed of up to eight model trains
Inputs	Throttle, inertia setting, emergency stop, train number
Outputs	Train control signals
Functions	Set engine speed based upon inertia settings; respond to emergency stop
Performance	Can update train speed at least 10 times per second
Manufacturing cost	$50
Power	10W (plugs into wall)
Physical size and weight	Console should be comfortable for two hands, approximate size of standard keyboard; weight <2 pounds

Chapter (2)

1. Assembly Language

ARM assembly code to remind us of the basic features of assembly languages. Assembly languages usually share the same basic features:

- One instruction appears per line.
- Labels, which give names to memory locations, start in the first column.
- Instructions must start in the second column or after to distinguish them from labels.
- Comments run from some designated comment character (; in the case of ARM) to the end of the line.

Assembly language follows this relatively structured form to make it easy for the assembler to parse the program and to consider most aspects of the program line by line.

2. little-endian and big-endian

The ARM processor can be configured at power-up to address the bytes in a word in either **little-endian** mode (with the lowest-order byte residing in the low-order bits of the word) **or big-endian mode** (the lowest-order byte stored in the highest bits of the word).

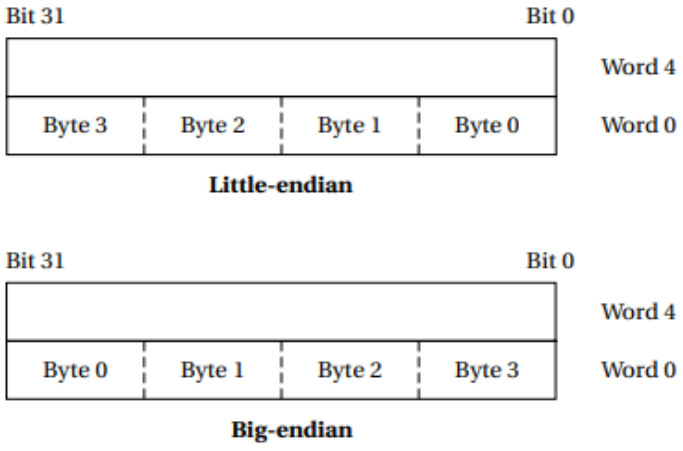


FIGURE 2.6
Byte organizations within an ARM word.

3. Instruction Sets

ADD	Add
ADC	Add with carry
SUB	Subtract
SBC	Subtract with carry
RSB	Reverse subtract
RSC	Reverse subtract with carry
MUL	Multiply
MLA	Multiply and accumulate

Arithmetic

AND	Bit-wise and
ORR	Bit-wise or
EOR	Bit-wise exclusive-or
BIC	Bit clear

Logical

LSL	Logical shift left (zero fill)
LSR	Logical shift right (zero fill)
ASL	Arithmetic shift left
ASR	Arithmetic shift right
ROR	Rotate right
RRX	Rotate right extended with C

Shift/rotate

FIGURE 2.9

ARM data instructions.

CMP	Compare
CMN	Negated compare
TST	Bit-wise test
TEQ	Bit-wise negated test

FIGURE 2.10

ARM comparison instructions.

MOV	Move
MVN	Move negated

FIGURE 2.11

ARM move instructions.

LDR	Load
STR	Store
LDRH	Load half-word
STRH	Store half-word
LDRSH	Load half-word signed
LDRB	Load byte
STRB	Store byte
ADR	Set register to address

FIGURE 2.12

ARM load-store instructions and pseudo-operations.

4. Example 2.2 -C assignments in ARM instructions

x = (a + b) - c;

```
ADR r4, a            ;      get address for a
LDR r0, [r4]         ;      get value of a
ADR r4, b            ;      get address for b, reusing r4
LDR r1, [r4]         ;      load value of b
ADD r3, r0, r1       ;      set intermediate result for x to a + b
ADR r4, c            ;      get address for c
LDR r2, [r4]         ;      get value of c
SUB r3, r3, r2       ;      complete computation of x
ADR r4, x            ;      get address for x
STR r3, [r4]         ;      store x at proper location
```

y = a ∗ (b + c)

```
ADR r4, b            ;      get address for b
LDR r0, [r4]         ;      get value of b
ADR r4, c            ;      get address for c
LDR r1, [r4]         ;      get value of c
ADD r2, r0, r1       ;      compute partial result of y
ADR r4, a            ;      get address for a
LDR r0, [r4]         ;      get value of a
MUL r2, r2, r0       ;      compute final value of y
ADR r4, y            ;      get address for y
STR r2, [r4]         ;      store value of y at proper location
```

z = (a <<2) | (b & 15)

```
ADR r4, a            ;      get address for a
LDR r0, [r4]         ;      get value of a
MOV r0, r0, LSL 2 ;         perform shift
ADR r4, b            ;      get address for b
LDR r1, [r4]         ;      get value of b
AND r1, r1, #15      ;      perform logical AND
ORR r1, r0, r1       ;      compute final value of z
ADR r4, z            ;      get address for z
STR r1, [r4]         ;      store value of z
```

5. **Example 2.3**

Implementing an if statement in ARM

We will use the following if statement as an example:

if (a < b) { x = 5; y = c + d; } else x = c – d;

```
; compute and test the condition
        ADR r4,a        ; get address for a
        LDR r0,[r4]     ; get value of a
        ADR r4,b        ; get address for b
        LDR r1,[r4]     ; get value of b
        CMP r0, r1      ; compare a < b
        BGE fblock      ; if a >= b, take branch
; the true block follows
        MOV r0,#5       ; generate value for x
        ADR r4,x        ; get address for x
        STR r0,[r4]     ; store value of x
        ADR r4,c        ; get address for c
        LDR r0,[r4]     ; get value of c
        ADR r4,d        ; get address for d
        LDR r1,[r4]     ; get value of d
        ADD r0,r0,r1    ; compute c + d
        ADR r4,y        ; get address for y
        STR r0,[r4]     ; store value of y
        B after         ; branch around the false block
; the false block follows
fblock ADR r4,c         ; get address for c
        LDR r0,[r4]     ; get value of c
        ADR r4,d        ; get address for d
        LDR r1,[r4]     ; get value of d
        SUB r0,r0,r1    ; compute c - d
        ADR r4,x        ; get address for x
        STR r0,[r4]     ; store value of x
after ... ; code after the if statement
```

6. **Example 2.5**

An FIR filter for the ARM

The C code for the FIR filter of Application Example 2.1 follows:

```
for (i = 0, f = 0; i < N; i++)
f=f+ c[i] * x[i];
```

```
i = 0; f = 0;
while (i < N) {
f=f+ c[i]*x[i]; i++; }
```

```
      ; loop initiation code
      MOV r0,#0       ; use r0 for i, set to 0
      MOV r8,#0       ; use a separate index for arrays
      ADR r2,N        ; get address for N
      LDR r1,[r2]     ; get value of N for loop termination test
      MOV r2,#0       ; use r2 for f, set to 0
      ADR r3,c        ; load r3 with address of base of c array
      ADR r5,x        ; load r5 with address of base of x array
      ; loop body
loop  LDR r4,[r3,r8]    ; get value of c[i]
      LDR r6,[r5,r8]    ; get value of x[i]
      MUL r4,r4,r6    ; compute c[i]*x[i]
      ADD r2,r2,r4    ; add into running sum f
      ; update loop counter and array index
      ADD r8,r8,#4      ; add one word offset to array index
      ADD r0,r0,#1      ; add 1 to i
      ; test for exit
      CMP r0,r1
      BLT loop        ; if i < N, continue loop
loopend...
```