1 An Introduction to Processor Design

Summary of chapter contents

The design of a general-purpose processor, in common with most engineering endeavours, requires the careful consideration of many trade-offs and compromises. In this chapter we will look at the basic principles of processor instruction set and logic design and the techniques available to the designer to help achieve the design objectives.

Abstraction is fundamental to understanding complex computers. This chapter introduces the abstractions which are employed by computer hardware designers, of which the most important is the logic gate. The design of a simple processor is presented, from the instruction set, through a register transfer level description, down to logic gates.

The ideas behind the *Reduced Instruction Set Computer* (RISC) originated in processor research programmes at Stanford and Berkeley universities around 1980, though some of the central ideas can be traced back to earlier machines. In this chapter we look at the thinking that led to the RISC movement and consequently influenced the design of the ARM processor which is the subject of the following chapters.

With the rapid development of markets for portable computer-based products, the power consumption of digital circuits is of increasing importance. At the end of the chapter we will look at the principles of low-power high-performance design.

1.1 Processor architecture and organization

All modern general-purpose computers employ the principles of the *stored-program digital computer.* The stored-program concept originated from the Princeton Institute of Advanced Studies in the 1940s and was first implemented in the 'Baby' machine which first ran in June 1948 at the University of Manchester in England.

Fifty years of development have resulted in a spectacular increase in the performance of processors and an equally spectacular reduction in their cost. Over this period of relentless progress in the cost-effectiveness of computers, the principles of operation have changed remarkably little. Most of the improvements have resulted from advances in the technology of electronics, moving from valves (vacuum tubes) to individual transistors, to integrated circuits (ICs) incorporating several bipolar transistors and then through generations of IC technology leading to today's very large scale integrated (VLSI) circuits delivering millions of field-effect transistors on a single chip. As transistors get smaller they get cheaper, faster, and consume less power. This win-win scenario has carried the computer industry forward for the past three decades, and will continue to do so at least for the next few years.

However, not all of the progress over the past 50 years has come from advances in electronics technology. There have also been occasions when a new insight into the way that technology is employed has made a significant contribution. These insights are described under the headings of computer architecture and computer organization, where we will work with the following interpretations of these terms:

Computer architecture

• Computer architecture describes the user's view of the computer. The instruction set, visible registers, memory management table structures and exception handling model are all part of the architecture.

Computer organization

• Computer organization describes the user-invisible implementation of the architecture. The pipeline structure, transparent cache, table-walking hardware and translation look-aside buffer are all aspects of the organization.

Amongst the advances in these aspects of the design of computers, the introduction of virtual memory in the early 1960s, of transparent cache memories, of pipelining and so on, have all been milestones in the evolution of computers. The RISC idea ranks amongst these advances, offering a significant shift in the balance of forces which determines the cost-effectiveness of computer technology.

What is a processor?

A general-purpose processor is a finite-state automaton that executes instructions held in a memory. The state of the system is defined by the values held in the memory locations together with the values held in certain registers within the processor itself (see Figure 1.1 on page 3; the hexadecimal notation for the memory addresses is explained in Section 6.2 on page 153). Each instruction defines a particular way the total state should change and it also defines which instruction should be executed next.

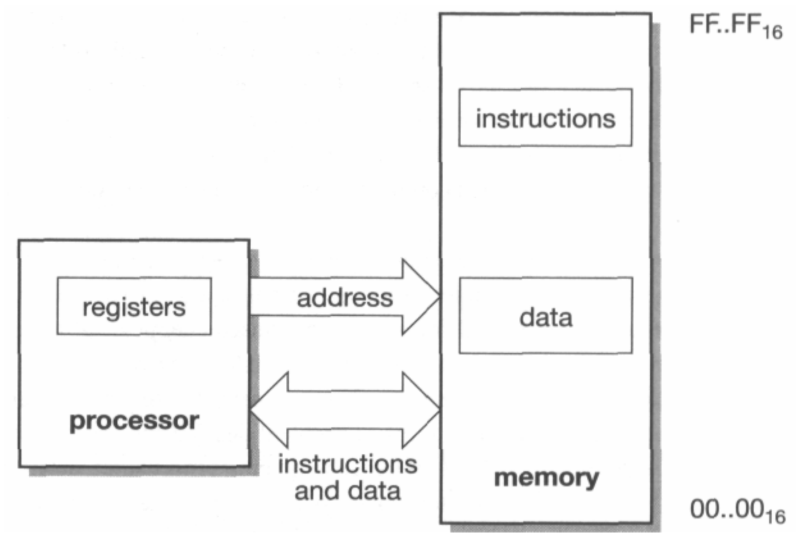


Figure 1.1 The state in a stored-program digital computer.

The Stored-program Computer

The **stored-program** digital computer keeps its instructions and data in the same memory system, allowing the instructions to be treated as data when necessary. This enables the processor itself to generate instructions which it can subsequently execute. Although programs that do this at a fine granularity **(self-modifying** code) are generally considered bad form these days since they are very difficult to debug, use at a coarser granularity is fundamental to the way most computers operate. Whenever a computer loads in a new program from disk (overwriting an old program) and then executes it the computer is employing this ability to change its own program.

Computer applications

Because of its programmability a stored-program digital computer is **universal,** which means that it can undertake any task that can be described by a suitable algorithm. Sometimes this is reflected by its configuration as a desktop machine where the user runs different programs at different times, but sometimes it is reflected by the same processor being used in a range of different applications, each with a fixed program. Such applications are characteristically embedded into products such as mobile telephones, automotive engine-management systems, and so on.

1.2 Abstraction in hardware design

Computers are very complex pieces of equipment that operate at very high speeds. A modern microprocessor may be built from several million transistors each of which can switch a hundred million times a second. Watch a document scroll up the screen on a desktop PC or workstation and try to imagine how a hundred million million transistor switching actions are used in each second of that movement. Now consider that every one of those switching actions is, in some sense, the consequence of a deliberate design decision. None of them is random or uncontrolled; indeed, a single error amongst those transitions is likely to cause the machine to collapse into a useless state. How can such complex systems be designed to operate so reliably?

Transistors

A clue to the answer may be found in the question itself. We have described the operation of the computer in terms of transistors, but what is a transistor? It is a curious structure composed from carefully chosen chemical substances with complex electrical properties that can only be understood by reference to the theory of quantum mechanics, where strange subatomic particles sometimes behave like waves and can only be described in terms of probabilities. Yet the gross behaviour of a transistor can be described, without reference to quantum mechanics, as a set of equations that relate the voltages on its terminals to the current that flows though it. These equations **abstract** the essential behaviour of the device from its underlying physics.

Logic gates

The equations that describe the behaviour of a transistor are still fairly complex. When a group of transistors is wired together in a particular structure, such as the CMOS (Complementary Metal Oxide Semiconductor) NAND gate shown in Figure 1.2, the behaviour of the group has a particularly simple description.

If each of the input wires *(A* and B) is held at a voltage which is either near to *Vdd* or near to *Vss,* the output will will also be near to *Vdd* or *Vss* according to the following rules:

- If *A* and *B* are both near to *Vdd,* the output will be near to *Vss.*
- If either *A* or *B* (or both) is near to *Vss,* the output will be near to *Vdd.*

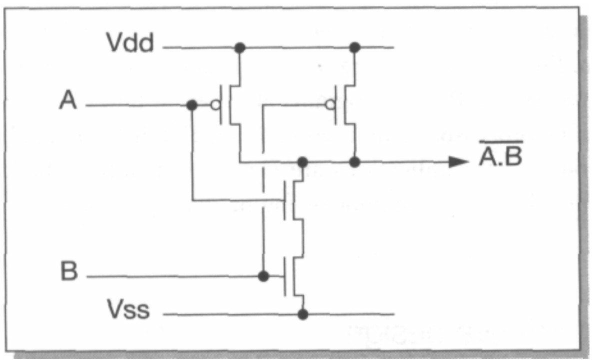


Figure 1.2 The transistor circuit of a static 2-input CMOS NAND gate.

With a bit of care we can define what is meant by 'near to' in these rules, and then associate the meaning **true** with a value near to *Vdd* and **false** with a value near to *Vss*. The circuit is then an implementation of the NAND Boolean logic function:

$$output = \text{—}(A\text{^}B) \qquad \text{Equation 1}$$

Although there is a lot of engineering design involved in turning four transistors into a reliable implementation of this equation, it can be done with sufficient reliability that the logic designer can think almost exclusively in terms of logic gates. The concepts that the logic designer works with are illustrated in Figure 1.3, and consist of the following 'views' of the logic gate:

Logic symbol

- A logic symbol.

  This is a symbol that represents a NAND gate function in a circuit schematic; there are similar symbols for other logic gates (for instance, removing the bubble from the output leaves an AND gate which generates the opposite output function; further examples are given in 'Appendix: Computer Logic' on page 399).

Truth table

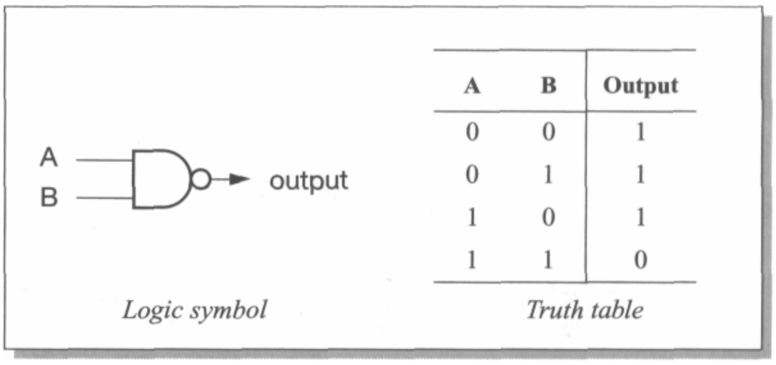
- A truth table.

  This describes the logic function of the gate, and encompasses everything that the logic designer needs to know about the gate for most purposes. The significance here is that it is a lot simpler than four sets of transistor equations.

  (In this truth table we have represented 'true' by '1' and 'false' by '0', as is common practice when dealing with Boolean variables.)

The gate abstraction

The point about the gate abstraction is that not only does it greatly simplify the process of designing circuits with great numbers of transistors, but it actually



A	B	Output
0	0	1
0	1	1
1	0	1
1	1	0

Figure 1.3 The logic symbol and truth table for a NAND gate.

removes the need to know that the gate is built from transistors. A logic circuit should have the same logical behaviour whether the gates are implemented using field-effect transistors (the transistors that are available on a CMOS process), bipolar transistors, electrical relays, fluid logic or any other form of logic. The implementation technology will affect the *performance* of the circuit, but it should have no effect on *its function.* It is the duty of the transistor-level circuit designer to support the gate abstraction as near perfectly as is possible in order to isolate the logic circuit designer from the need to understand the transistor equations.

Levels of abstraction

It may appear that this point is being somewhat laboured, particularly to those readers who have worked with logic gates for many years. However, the principle that is illustrated in the gate level abstraction is repeated many times at different levels in computer science and is absolutely fundamental to the process which we began considering at the start of this section, which is the management of complexity.

The process of gathering together a few components at one level to extract their essential joint behaviour and hide all the unnecessary detail at the next level enables us to scale orders of complexity in a few steps. For instance, if each level encompasses four components of the next lower level as our gate model does, we can get from a transistor to a microprocessor comprising a million transistors in just ten steps. In many cases we work with more than four components, so the number of steps is greatly reduced.

A typical hierarchy of abstraction at the hardware level might be:

1. transistors;
2. logic gates, memory cells, special circuits;
3. single-bit adders, multiplexers, decoders, flip-flops;
4. word-wide adders, multiplexers, decoders, registers, buses;
5. ALUs (Arithmetic-Logic Units), barrel shifters, register banks, memory blocks;
6. processor, cache and memory management organizations;
7. processors, peripheral cells, cache memories, memory management units;
8. integrated system chips;
9. printed circuit boards;
10. mobile telephones, PCs, engine controllers.

The process of understanding a design in terms of levels of abstraction is reasonably concrete when the design is expressed in hardware. But the process doesn't stop with the hardware; if anything, it is even more fundamental to the understanding of software and we will return to look at abstraction in software design in due course.

Gate-level design

The next step up from the logic gate is to assemble a library of useful functions each composed of several gates. Typical functions are, as listed above, adders, multiplexers, decoders and flip-flops, each 1-bit wide. This book is not intended to be a general introduction to logic design since its principal subject material relates to the design and use of processor cores and any reader who is considering applying this information should already be familiar with conventional logic design.

For those who are not so familiar with logic design or who need their knowledge refreshing, 'Appendix: Computer Logic' on page 399 describes the essentials which will be assumed in the next section. It includes brief details on:

- Boolean algebra and notation;
- binary numbers;
- binary addition;
- multiplexers;
- clocks;
- sequential circuits;
- latches and flip-flops;
- registers.

If any of these terms is unfamiliar, a brief look at the appendix may yield sufficient information for what follows.

Note that although the appendix describes these circuit functions in terms of simple logic gates, there are often more efficient CMOS implementations based on alternative transistor circuits. There are many ways to satisfy the basic requirements of logic design using the complementary transistors available on a CMOS chip, and new transistor circuits are published regularly.

For further information consult a text on logic design; a suitable reference is suggested in the 'Bibliography' on page 410.

1.3 MU0 - a simple processor

A simple form of processor can be built from a few basic components:

- **a program counter** (PC) register that is used to hold the address of the current instruction;
- a single register called an **accumulator** (ACC) that holds a data value while it is worked upon;
- **an arithmetic-logic unit** (ALU) that can perform a number of operations on binary operands, such as add, subtract, increment, and so on;
- **an instruction register** (IR) that holds the current instruction while it is executed;
- instruction decode and control logic that employs the above components to achieve the desired results from each instruction.

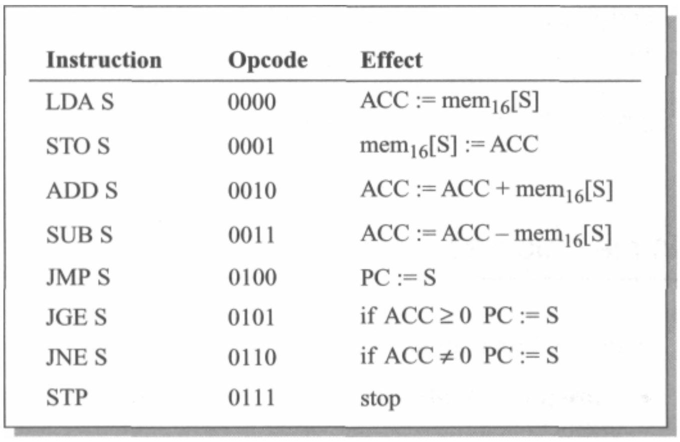
This limited set of components allows a restricted set of instructions to be implemented. Such a design has been employed at the University of Manchester for many years to illustrate the principles of processor design. Manchester-designed machines are often referred to by the names MUn for $1 < n < 6$, so this simple machine is known as MU0. It is a design developed only for teaching and was not one of the large-scale machines built at the university as research vehicles, though it is similar to the very first Manchester machine and has been implemented in various forms by undergraduate students.

The MU0 instruction set

MU0 is a 16-bit machine with a 12-bit address space, so it can address up to 8 Kbytes of memory arranged as 4,096 individually addressable 16-bit locations. Instructions are 16 bits long, with a 4-bit operation code (or **opcode)** and a 12-bit address field (S) as shown in Figure 1.4. The simplest instruction set uses only eight of the 16 available opcodes and is summarized in Table 1.1.

An instruction such as 'ACC := ACC + mem_{16}[S]' means 'add the contents of the (16-bit wide) memory location whose address is S to the accumulator'. Instructions are fetched from consecutive memory addresses, starting from address zero, until an instruction which modifies the PC is executed, whereupon fetching starts from the new address given in the 'jump' instruction.

Table 1.1 The MU0 instruction set.



Instruction	Opcode	Effect
LDA S	0000	ACC := mem_{16}[S]
STO S	0001	mem_{16}[S] := ACC
ADD S	0010	ACC := ACC + mem_{16}[S]
SUB S	0011	ACC := ACC − mem_{16}[S]
JMP S	0100	PC := S
JGE S	0101	if ACC $\geq$ 0 PC := S
JNE S	0110	if ACC $\neq$ 0 PC := S
STP	0111	stop

MU0 logic design

To understand how this instruction set might be implemented we will go through the design process in a logical order. The approach taken here will be to separate the design into two components:

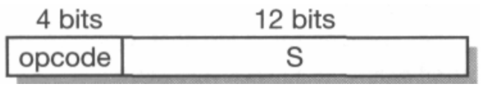


Figure 1.4 The MU0 instruction format.

- The datapath.

  All the components carrying, storing or processing many bits in parallel will be considered part of the datapath, including the accumulator, program counter, ALU and instruction register. For these components we will use a register transfer level **(RTL)** design style based on registers, multiplexers, and so on.

- The control logic.

  Everything that does not fit comfortably into the datapath will be considered part of the control logic and will be designed using a finite state machine (FSM) approach.

Datapath design

There are many ways to connect the basic components needed to implement the MU0 instruction set. Where there are choices to be made we need a guiding principle to help us make the right choices. Here we will follow the principle that the memory will be the limiting factor in our design, and a memory access will always take a clock cycle. Hence we will aim for an implementation where:

- Each instruction takes exactly the number of clock cycles defined by the number of memory accesses it must make.

Referring back to Table 1.1 we can see that the first four instructions each require two memory accesses (one to fetch the instruction itself and one to fetch or store the operand) whereas the last four instructions can execute in one cycle since they do not require an operand. (In practice we would probably not worry about the efficiency of the STP instruction since it halts the processor for ever.) Therefore we need a datapath design which has sufficient resource to allow these instructions to complete in two or one clock cycles. A suitable datapath is shown in Figure 1.5.

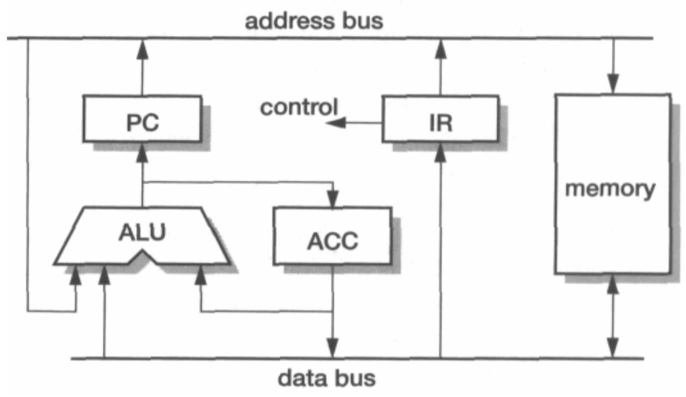


Figure 1.5 MU0 datapath example.

(Readers who might expect to see a dedicated PC incrementer in this datapath should note that all instructions that do not change the PC take two cycles, so the main ALU is available during one of these cycles to increment the PC.)

Datapath operation

The design we will develop assumes that each instruction starts when it has arrived in the instruction register. After all, until it is in the instruction register we cannot know which instruction we are dealing with. Therefore an instruction executes in two stages, possibly omitting the first of these:

1. Access the memory operand and perform the desired operation.

   The address in the instruction register is issued and either an operand is read from memory, combined with the accumulator in the ALU and written back into the accumulator, or the accumulator is stored out to memory.

2. Fetch the next instruction to be executed.

   Either the PC or the address in the instruction register is issued to fetch the next instruction, and in either case the address is incremented in the ALU and the incremented value saved into the PC.

Initialization

The processor must start in a known state. Usually this requires a *reset* input to cause it to start executing instructions from a known address. We will design MU0 to start executing from address 000_{16}. There are several ways to achieve this, one of which is to use the reset signal to zero the ALU output and then clock this into the PC register.

Register transfer level design

The next step is to determine exactly the control signals that are required to cause the datapath to carry out the full set of operations. We assume that all the registers change state on the falling edge of the input clock, and where necessary have control signals that may be used to prevent them from changing on a particular clock edge. The PC, for example, will change at the end of a clock cycle where *PCce* is ' 1' but will not change when *PCce* is '0'.

A suitable register organization is shown in Figure 1.6 on page 11. This shows enables on all of the registers, function select lines to the ALU (the precise number and interpretation to be determined later), the select control lines for two multiplexers, the control for a tri-state driver to send the ACC value to memory and memory request *(MEMrq)* and read/write *(RnW)* control lines. The other signals shown are outputs from the datapath to the control logic, including the opcode bits and signals indicating whether ACC is zero or negative which control the respective conditional jump instructions.

Control logic

The control logic simply has to decode the current instruction and generate the appropriate levels on the datapath control signals, using the control inputs from the datapath where necessary. Although the control logic is a finite state machine, and therefore in principle the design should start from a state transition diagram, in this case the FSM is trivial and the diagram not worth drawing. The implementation requires only two states, 'fetch' and 'execute', and one bit of state *(Ex/ft)* is therefore sufficient.

The control logic can be presented in tabular form as shown in Table 1.2 on page 12. In this table an 'x' indicates a *don't care* condition. Once the ALU function select codes have been assigned the table may be implemented directly as a PLA (programmable logic array) or translated into combinatorial logic and implemented using standard gates.

A quick scrutiny of Table 1.2 reveals a few easy simplifications. The program counter and instruction register clock enables *(PCce* and *IRce)* are always the same. This makes sense, since whenever a new instruction is being fetched the ALU is computing the next program counter value, and this should be latched too. Therefore these control signals may be merged into one. Similarly, whenever the accumulator is driving the data bus *(ACCoe* is high) the memory should perform a write operation *(Rn W* is low), so one of these signals can be generated from the other using an inverter.

After these simplifications the control logic design is almost complete. It only remains to determine the encodings of the ALU functions.

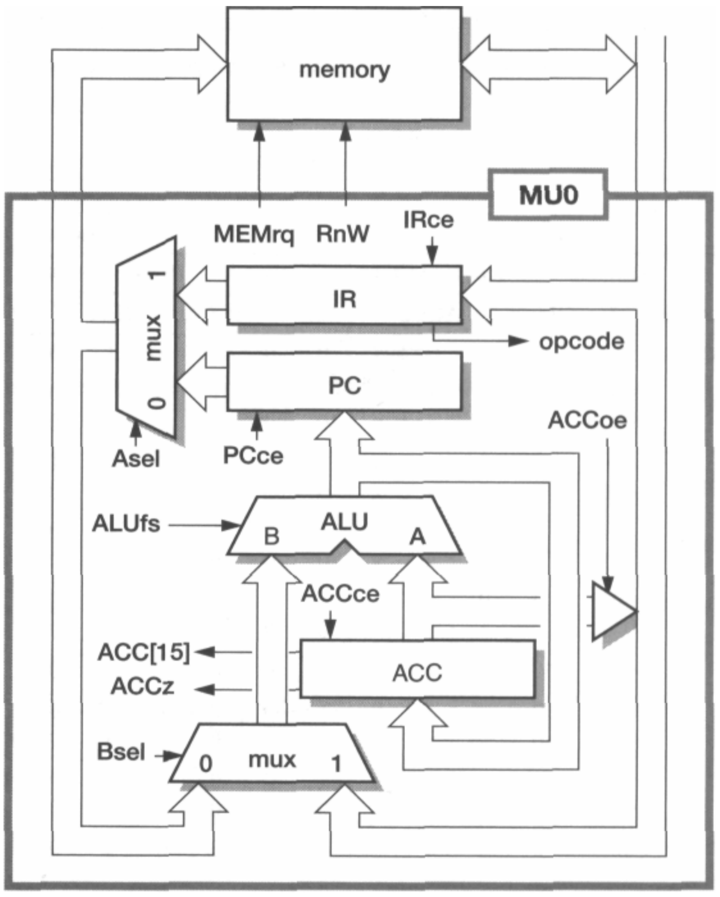


Figure 1.6 MU0 register transfer level organization.

The control logic can be presented in tabular form as shown in Table 1.2 on page 12. In this table an 'x' indicates a *don't care* condition. Once the ALU function select codes have been assigned the table may be implemented directly as a PLA (programmable logic array) or translated into combinatorial logic and implemented using standard gates.

A quick scrutiny of Table 1.2 reveals a few easy simplifications. The program counter and instruction register clock enables *(PCce* and *IRce)* are always the same. This makes sense, since whenever a new instruction is being fetched the ALU is computing the next program counter value, and this should be latched too. Therefore these control signals may be merged into one. Similarly, whenever the accumulator is driving the data bus *(ACCoe* is high) the memory should perform a write operation *(RnW* is low), so one of these signals can be generated from the other using an inverter.

After these simplifications the control logic design is almost complete. It only remains to determine the encodings of the ALU functions.

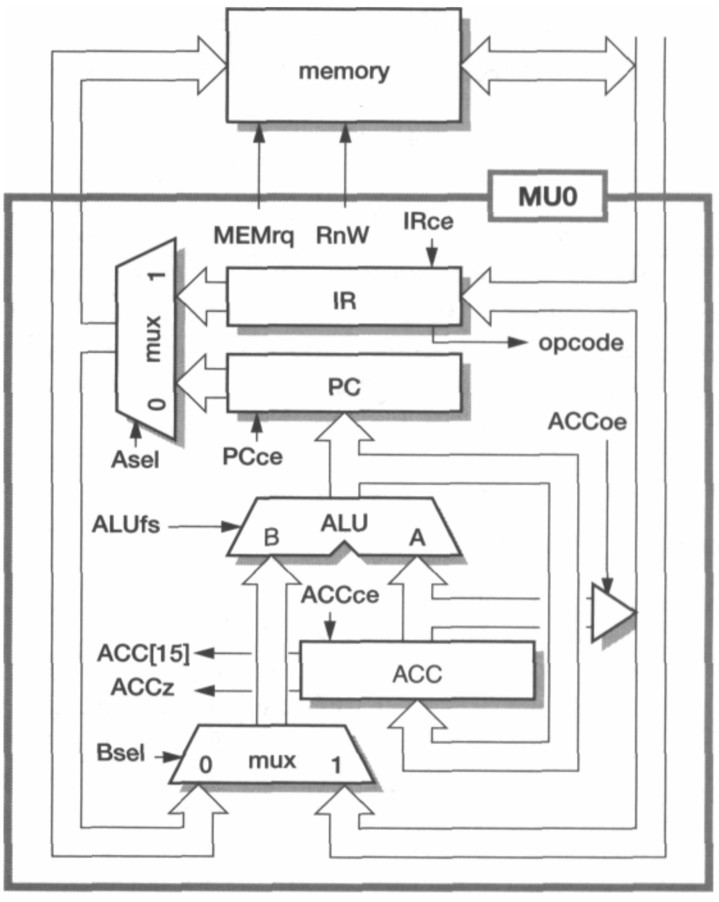


Figure 1.6 MU0 register transfer level organization.

ALU design

Most of the register transfer level functions in Figure 1.6 have straightforward logic implementations (readers who are in doubt should refer to 'Appendix: Computer Logic' on page 399). The MU0 ALU is a little more complex than the simple adder described in the appendix, however.

The ALU functions that are required are listed in Table 1.2. There are five of them (*A* + *B*, *A* — *B*, *B*, *B* + 1,0), the last of which is only used while reset is active. Therefore the reset signal can control this function directly and the control logic need only generate a 2-bit function select code to choose between the other four. If the principal ALU inputs are the *A* and *B* operands, all the functions may be produced by augmenting a conventional binary adder:

- *A* + *B* is the normal adder output (assuming that the carry-in is zero).
- *A* — *B* may be implemented as *A* + *B* + 1, requiring the *B* inputs to be inverted and the carry-in to be forced to a one.
- *B* is implemented by forcing the *A* inputs and the carry-in to zero.
- *B* + 1 is implemented by forcing *A* to zero and the carry-in to one.

Table 1.2 MU0 control logic.

Inputs						Outputs									
Instruction	Opcode	Reset	Ex/ft	ACCz	ACC15	Asel	Bsel	ACCce	PCce	IRce	ACCoe	ALUfs	MEMrq	RnW	Ex/ft
Reset	xxxx	1	x	x	x	0	0	1	1	1	0	= 0	1	1	0
LDA S	0000	0	0	x	x	1	1	1	0	0	0	= B	1	1	1
	0000	0	1	x	x	0	0	0	1	1	0	B+1	1	1	0
STO S	0001	0	0	x	x	1	x	0	0	0	1	x	1	0	1
	0001	0	1	x	x	0	0	0	1	1	0	B+1	1	1	0
ADD S	0010	0	0	x	x	1	1	1	0	0	0	A+B	1	1	1
	0010	0	1	x	x	0	0	0	1	1	0	B+1	1	1	0
SUB S	0011	0	0	x	x	1	1	1	0	0	0	A–B	1	1	1
	0011	0	1	x	x	0	0	0	1	1	0	B+1	1	1	0
JMP S	0100	0	x	x	x	1	0	0	1	1	0	B+1	1	1	0
JGE S	0101	0	x	x	0	1	0	0	1	1	0	B+1	1	1	0
	0101	0	x	x	1	0	0	0	1	1	0	B+1	1	1	0
JNE S	0110	0	x	0	x	1	0	0	1	1	0	B+1	1	1	0
	0110	0	x	1	x	0	0	0	1	1	0	B+1	1	1	0
STP	0111	0	x	x	x	1	x	0	0	0	0	x	0	1	0

The gate-level logic for the ALU is shown in Figure 1.7. *Aen* enables the *A* operand or forces it to zero; *Binv* controls whether or not the *B* operand is inverted. The carry-out *(Cout)* from one bit is connected to the carry-in *(Cin)* of the next; the carry-in to the first bit is controlled by the ALU function selects (as are *Aen* and *Binv),* and the carry-out from the last bit is unused Together with the multiplexers, registers, control logic and a bus buffer (which is used to put the accumulator value onto the data bus), the processor is complete. Add a standard memory and you have a workable computer.

MU0 extensions

Although MU0 is a very simple processor and would not make a good target for a high-level language compiler, it serves to illustrate the basic principles of processor design. The design process used to develop the first ARM processors differed mainly in complexity and not in principle. MU0 designs based on microcoded control logic have also been developed, as have extensions to incorporate indexed addressing. Like any good processor, MU0 has spaces left in the instruction space which allow future expansion of the instruction set.

To turn MU0 into a useful processor takes quite a lot of work. The following extensions seem most important:

- Extending the address space.
- Adding more addressing modes.
- Allowing the PC to be saved in order to support a subroutine mechanism.
- Adding more registers, supporting interrupts, and so on...

Overall, this doesn't seem to be the place to start from if the objective is to design a high-performance processor which is a good compiler target.

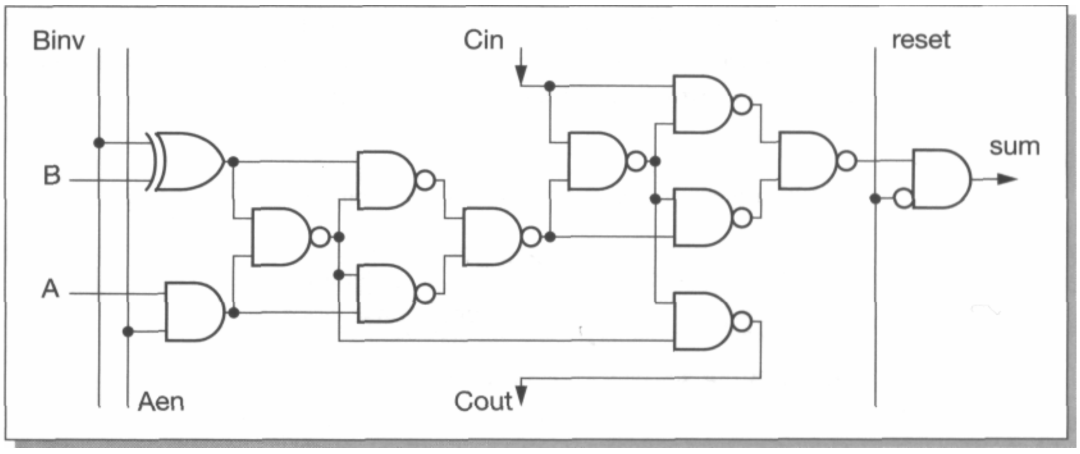


Figure 1.7 MU0 ALU logic for one bit.

1.4 Instruction set design

If the MU0 instruction set is not a good choice for a high-performance processor, what other choices are there?

Starting from first principles, let us look at a basic machine operation such as an instruction to add two numbers to produce a result.

4-addreSS instructions

In its most general form, this instruction requires some bits to differentiate it from other instructions, some bits to specify the operand addresses, some bits to specify where the result should be placed (the **destination),** and some bits to specify the address of the next instruction to be executed. An assembly language format for such an instruction might be:

```
ADD    d,   s1,   s2,   next_i   ;  d  :=  s1  +
s2
```

Such an instruction might be represented in memory by a binary format such as that shown in Figure 1.8. This format requires *4n +f* bits per instruction where each operand requires *n* bits and the opcode that specifies 'ADD' requires/bits.

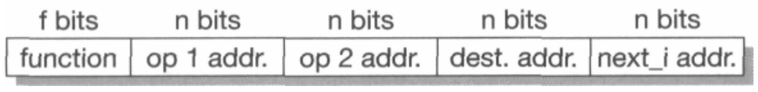


Figure 1.8 A 4-address instruction format.

3-addresS instructions

The first way to reduce the number of bits required for each instruction is to make the address of the next instruction implicit (except for branch instructions, whose role is to modify the instruction sequence explicitly). If we assume that the default next instruction can be found by adding the size of the instruction to the PC, we get a 3-address instruction with an assembly language format like this:

```
ADD    d,   s1,   s2     ;  d  :=  s1  +  s2
```

A binary representation of such an instruction is shown in Figure 1.9.

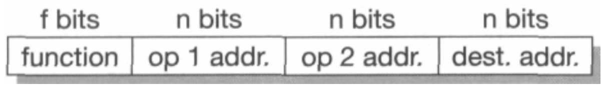


Figure 1.9 A 3-address instruction format.

2-addreSS instructions

A further saving in the number of bits required to store an instruction can be achieved by making the destination register the same as one of the source registers. The assembly language format could be:

```
ADD    d, s1   ; d := d + s1
```

The binary representation now reduces to that shown in Figure 1.10.

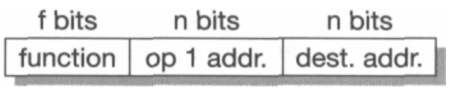


Figure 1.10 A 2-address instruction format.

1-address instructions

If the destination register is made implicit it is often called the **accumulator** (see, for example, MU0 in the previous section); an instruction need only specify one operand:

```
ADD   s1     ; accumulator := accumulator + s1
```

The binary representation simplifies further to that shown in Figure 1.11.

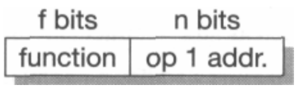


Figure 1.11 A 1-address (accumulator) instruction format.

0-address instructions

Finally, an architecture may make all operand references implicit by using an evaluation stack. The assembly language format is:

```
ADD      ;    top_of_stack    :=    top_of_stack +
next_on_stack
```

The binary representation is as shown in Figure 1.12.

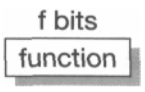


Figure 1.12 A 0-address instruction format.

Examples of n-address use

All these forms of instruction have been used in processor instruction sets apart from the 4-address form which, although it is used internally in some microcode designs, is unnecessarily expensive for a machine-level instruction set. For example:

- The Inmos transputer uses a 0-address evaluation stack architecture.
- The MU0 example in the previous section illustrates a simple 1 -address architecture.

- The Thumb instruction set used for high code density on some ARM processors uses an architecture which is predominantly of the 2-address form (see Chapter 7).
- The standard ARM instruction set uses a 3-address architecture.

Addresses

An address in the MU0 architecture is the straightforward 'absolute' address of the memory location which contains the desired operand. However, the three addresses in the ARM 3-address instruction format are register specifiers, not memory addresses. In general, the term '3-address architecture' refers to an instruction set where the two source operands and the destination can be specified independently of each other, but often only within a restricted set of possible values.

Instruction types

We have just looked at a number of ways of specifying an 'ADD' instruction. A complete instruction set needs to do more than perform arithmetic operations on operands in memory. A general-purpose instruction set can be expected to include instructions in the following categories:

- Data processing instructions such as add, subtract and multiply.
- Data movement instructions that copy data from one place in memory to another, or from memory to the processor's registers, and so on.
- Control flow instructions that switch execution from one part of the program to another, possibly depending on data values.
- Special instructions to control the processor's execution state, for instance to switch into a privileged mode to carry out an operating system function.

Sometimes an instruction will fit into more than one of these categories. For example, a 'decrement and branch if non-zero' instruction, which is useful for controlling program loops, does some data processing on the loop variable and also performs a control flow function. Similarly, a data processing instruction which fetches an operand from an address in memory and places its result in a register can be viewed as performing a data movement function.

Orthogonal instructions

An instruction set is said to be **orthogonal** if each choice in the building of an instruction is independent of the other choices. Since add and subtract are similar operations, one would expect to be able to use them in similar contexts. If add uses a 3-address format with register addresses, so should subtract, and in neither case should there be any peculiar restrictions on the registers which may be used.

An orthogonal instruction set is easier for the assembly language programmer to learn and easier for the compiler writer to target. The hardware implementation will usually be more efficient too.

Addressing modes

When accessing an operand for a data processing or movement instruction, there are several standard techniques used to specify the desired location. Most processors support several of these **addressing modes** (though few support all of them):

1. Immediate addressing: the desired value is presented as a binary value in the instruction.
2. Absolute addressing: the instruction contains the full binary address of the desired value in memory.
3. Indirect addressing: the instruction contains the binary address of a memory location that contains the binary address of the desired value.
4. Register addressing: the desired value is in a register, and the instruction contains the register number.
5. Register indirect addressing: the instruction contains the number of a register which contains the address of the value in memory.
6. Base plus offset addressing: the instruction specifies a register (the **base)** and a binary offset to be added to the base to form the memory address.
7. Base plus index addressing: the instruction specifies a base register and another register (the **index)** which is added to the base to form the memory address.
8. Base plus scaled index addressing: as above, but the index is multiplied by a con stant (usually the size of the data item, and usually a power of two) before being added to the base.
9. Stack addressing: an implicit or specified register (the **stack pointer)** points to an area of memory (the **stack)** where data items are written **(pushed)** or read **(popped)** on a last-in-first-out basis.

Note that the naming conventions used for these modes by different processor manufacturers are not necessarily as above. The list can be extended almost indefinitely by adding more levels of indirection, adding base plus index plus offset, and so on. However, most of the common addressing modes are covered in the list above.

Control flow instructions

Where the program must deviate from the default (normally sequential) instruction sequence, a control flow instruction is used to modify the program counter (PC) explicitly. The simplest such instructions are usually called 'branches' or 'jumps'. Since most branches require a relatively short range, a common form is the 'PC-relative' branch. A typical assembly language format is:

```
          B
          ..
LABEL     ..
```

Here the assembler works out the displacement which must be added to the value the PC has when the branch is executed in order to force the PC to point to LABEL. The maximum range of the branch is determined by the number of bits allocated to the displacement in the binary format; the assembler should report an error if the required branch is out of range.

Conditional branches

A Digital Signal Processing (DSP) program may execute a fixed instruction sequence for ever, but a general-purpose processor is usually required to vary its program in response to data values. Some processors (including MU0) allow the values in the general registers to control whether or not a branch is taken through instructions such as:

- Branch if a particular register is zero (or not zero, or negative, and so on).
- Branch if two specified registers are equal (or not equal).

Condition code register

However, the most frequently used mechanism is based on a condition code register, which is a special-purpose register within the processor. Whenever a data processing instruction is executed (or possibly only for special instructions, or instructions that specifically enable the condition code register), the condition code register records whether the result was zero, negative, overflowed, produced a carry output, and so on. The conditional branch instructions are then controlled by the state of the condition code register when they execute.

Subroutine calls

Sometimes a branch is executed to call a subprogram where the instruction sequence should **return** to the calling sequence when the subprogram terminates. Since the subprogram may be called from many different places, a record of the calling address must be kept. There are many different ways to achieve this:

- The calling routine could compute a suitable return address and put it in a stand ard memory location for use by the subprogram as a return address before exe cuting the branch.
- The return address could be pushed onto a stack.
- The return address could be placed in a register.

Subprogram calls are sufficiently common that most architectures include specific instructions to make them efficient. They typically require to jump further across memory than simple branches, so it makes sense to treat them separately. Often they are not conditional; a conditional subprogram call is programmed, when required, by inserting an unconditional call and branching around it with the opposite condition.

Subprogram return

The return instruction moves the return address from wherever it was stored (in memory, possibly on a stack, or in a register) back into the PC.

System calls

Another category of control flow instruction is the system call. This is a branch to an operating system routine, often associated with a change in the privilege level of the executing program. Some functions in the processor, possibly including all the input and output peripherals, are protected from access by user code. Therefore a user program that needs to access these functions must make a system call.

System calls pass through protection barriers in a controlled way. A well-designed processor will ensure that it is possible to write a multi-user operating system where one user's program is protected from assaults from other, possibly malicious, users. This requires that a malicious user cannot change the system code and, when access to protected functions is required, the system code must make thorough checks that the requested function is authorized.

This is a complex area of hardware and software design. Most embedded systems (and many desktop systems) do not use the full protection capabilities of the hardware, but a processor which does not support a protected system mode will be excluded from consideration for those applications that demand this facility, so most microprocessors now include such support. Whilst it is not necessary to understand the full implications of supporting a secure operating system to appreciate the basic design of an instruction set, even the less well-informed reader should have an awareness of the issues since some features of commercial processor architectures make little sense unless this objective of potentially secure protection is borne in mind.

Exceptions

The final category of control flow instruction comprises cases where the change in the flow of control is not the primary intent of the programmer but is a consequence of some unexpected (and possibly unwanted) side-effect of the program. An attempt to access a memory location may fail, for instance, because a fault is detected in the memory subsystem. The program must therefore deviate from its planned course in order to attempt to recover from the problem.

These unplanned changes in the flow of control are termed **exceptions.**

1.5 Processor design trade-offs

The art of processor design is to define an instruction set that supports the functions that are useful to the programmer whilst allowing an implementation that is as efficient as possible. Preferably, the same instruction set should also allow future, more sophisticated implementations to be equally efficient.

The programmer generally wants to express his or her program in as abstract a way as possible, using a high-level language which supports ways of handling concepts that are appropriate to the problem. Modern trends towards functional and object-oriented languages move the level of abstraction higher than older imperative languages such as C, and even the older languages were quite a long way removed from typical machine instructions.

The **semantic gap** between a high-level language construct and a machine instruction is bridged by a **compiler,** which is a (usually complex) computer program that translates a high-level language program into a sequence of machine instructions. Therefore the processor designer should define his or her instruction set to be a good compiler target rather than something that the programmer will use directly to solve the problem by hand. So, what sort of instruction set makes a good compiler target?

Complex Instruction Set Computers

Prior to 1980, the principal trend in instruction set design was towards increasing complexity in an attempt to reduce the semantic gap that the compiler had to bridge. Single instruction procedure entries and exits were incorporated into the instruction set, each performing a complex sequence of operations over many clock cycles. Processors were sold on the sophistication and number of their addressing modes, data types, and so on.

The origins of this trend were in the minicomputers developed during the 1970s. These computers had relatively slow main memories coupled to processors built using many simple integrated circuits. The processors were controlled by microcode ROMs (Read Only Memories) that were faster than main memory, so it made sense to implement frequently used operations as microcode sequences rather than them requiring several instructions to be fetched from main memory.

Throughout the 1970s microprocessors were advancing in their capabilities. These single chip processors were dependent on state-of-the-art semiconductor technology to achieve the highest possible number of transistors on a single chip, so their development took place within the semiconductor industry rather than within the computer industry. As a result, microprocessor designs displayed a lack of original thought at the architectural level, particularly with respect to the demands of the technology that was used in their implementation. Their designers, at best, took ideas from the minicomputer industry where the implementation technology was very different. In particular, the microcode ROM which was needed for all the complex routines absorbed an unreasonable proportion of the area of a single chip, leaving little room for other performance-enhancing features.

This approach led to the single-chip Complex Instruction Set Computers (CISCs) of the late 1970s, which were microprocessors with minicomputer instruction sets that were severely compromised by the limited available silicon resource.

The RISC revolution

Into this world of increasingly complex instruction sets the Reduced Instruction Set Computer **(RISC)** was born. The RISC concept was a major influence on the design of the ARM processor; indeed, RISC was the ARM's middle name. But before we look at either RISC or the ARM in more detail we need a bit more background on what processors do and how they can be designed to do it quickly.

If reducing the semantic gap between the processor instruction set and the high-level language is not the right way to make an efficient computer, what other options are open to the designer?

What processors do

If we want to make a processor go fast, we must first understand what it spends its time doing. It is a common misconception that computers spend their time computing, that is, carrying out arithmetic operations on user data. In practice they spend very little time 'computing' in this sense. Although they do a fair amount of arithmetic, most of this is with addresses in order to locate the relevant data items and program routines. Then, having found the user's data, most of the work is in moving it around rather than processing it in any transformational sense.

At the instruction set level, it is possible to measure the frequency of use of the various different instructions. It is very important to obtain dynamic measurements, that is, to measure the frequency of instructions that are executed, rather than the static frequency, which is just a count of the various instruction types in the binary image. A typical set of statistics is shown in Table 1.3; these statistics were gathered running a print preview program on an ARM instruction emulator, but are broadly typical of what may be expected from other programs and instruction sets.

Table 1.3 Typical dynamic instruction usage.

Instruction type	Dynamic usage
Data movement	43%
Control flow	23%
Arithmetic operations	15%
Comparisons	13%
Logical operations	5%
Other	1%

These sample statistics suggest that the most important instructions to optimise are those concerned with data movement, either between the processor registers and memory or from register to register. These account for almost half of all instructions executed. Second most frequent are the control flow instructions such as branches and procedure calls, which account for another quarter. Arithmetic operations are down at 15%, as are comparisons.

Now we have a feel for what processors spend their time doing, we can look at ways of making them go faster. The most important of these is pipelining. Another important technique is the use of a cache memory, which will be covered in Section 10.3 on page 272. A third technique, super-scalar instruction execution, is very complex, has not been used on ARM processors and is not covered in this book.

What processors do

If we want to make a processor go fast, we must first understand what it spends its time doing. It is a common misconception that computers spend their time computing, that is, carrying out arithmetic operations on user data. In practice they spend very little time 'computing' in this sense. Although they do a fair amount of arithmetic, most of this is with addresses in order to locate the relevant data items and program routines. Then, having found the user's data, most of the work is in moving it around rather than processing it in any transformational sense.

At the instruction set level, it is possible to measure the frequency of use of the various different instructions. It is very important to obtain dynamic measurements, that is, to measure the frequency of instructions that are executed, rather than the static frequency, which is just a count of the various instruction types in the binary image. A typical set of statistics is shown in Table 1.3; these statistics were gathered running a print preview program on an ARM instruction emulator, but are broadly typical of what may be expected from other programs and instruction sets.

Table 1.3 Typical dynamic instruction usage.

Instruction type	Dynamic usage
Data movement	43%
Control flow	23%
Arithmetic operations	15%
Comparisons	13%
Logical operations	5%
Other	1%

These sample statistics suggest that the most important instructions to optimise are those concerned with data movement, either between the processor registers and memory or from register to register. These account for almost half of all instructions executed. Second most frequent are the control flow instructions such as branches and procedure calls, which account for another quarter. Arithmetic operations are down at 15%, as are comparisons.

Now we have a feel for what processors spend their time doing, we can look at ways of making them go faster. The most important of these is pipelining. Another important technique is the use of a cache memory, which will be covered in Section 10.3 on page 272. A third technique, super-scalar instruction execution, is very complex, has not been used on ARM processors and is not covered in this book.

Pipelines

A processor executes an individual instruction in a sequence of steps. A typical sequence might be:

1. Fetch the instruction from memory (fetch).
2. Decode it to see what sort of instruction it is (dec).
3. Access any operands that may be required from the register bank (reg).
4. Combine the operands to form the result or a memory address (ALU).
5. Access memory for a data operand, if necessary (mem).
6. Write the result back to the register bank (res).

Not all instructions will require every step, but most instructions will require most of them. These steps tend to use different hardware functions, for instance the ALU is probably only used in step 4. Therefore, if an instruction does not start before its predecessor has finished, only a small proportion of the processor hardware will be in use in any step.

An obvious way to improve the utilization of the hardware resources, and also the processor throughput, would be to start the next instruction before the current one has finished. This technique is called **pipelining,** and is a very effective way of exploiting concurrency in a general-purpose processor.

Taking the above sequence of operations, the processor is organized so that as soon as one instruction has completed step 1 and moved on to step 2, the next instruction begins step 1. This is illustrated in Figure 1.13. In principle such a pipeline should deliver a six times speed-up compared with non-overlapped instruction execution; in practice things do not work out quite so well for reasons we will see below.

Pipeline hazards

It is relatively frequent in typical computer programs that the result from one instruction is used as an operand by the next instruction. When this occurs the pipeline operation shown in Figure 1.13 breaks down, since the result of instruction 1 is not available at the time that instruction 2 collects its operands. Instruction 2 must therefore stall until the result is available, giving the behaviour shown in Figure 1.14 on page 23. This is a **read-after-write** pipeline hazard.

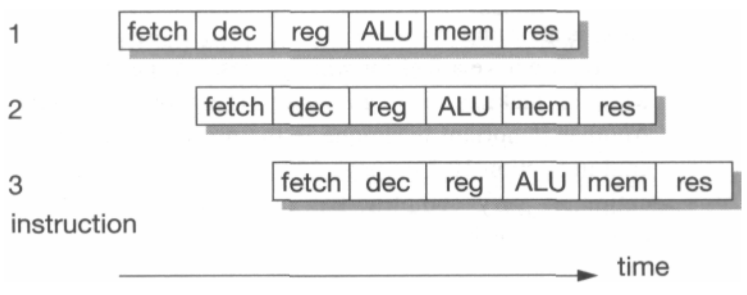


Figure 1.13 Pipelined instruction execution.

Branch instructions result in even worse pipeline behaviour since the fetch step of the following instruction is affected by the branch target computation and must therefore be deferred. Unfortunately, subsequent fetches will be taking place while the branch is being decoded and before it has been recognized as a branch, so the fetched instructions may have to be discarded. If, for example, the branch target calculation is performed in the ALU stage of the pipeline in Figure 1.13, three instructions will have been fetched from the old stream before the branch target is available (see Figure 1.15). It is better to compute the branch target earlier in the pipeline if possible, even though this will probably require dedicated hardware. If branch instructions have a fixed format, the target may be computed **speculatively** (that is, before it has been determined that the instruction *is a* branch) during the 'dec' stage, thereby reducing the branch latency to a single cycle, though note that in this pipeline there may still be hazards on a conditional branch due to dependencies on the condition code result of the instruction preceding the branch. Some RISC architectures (though not the ARM)

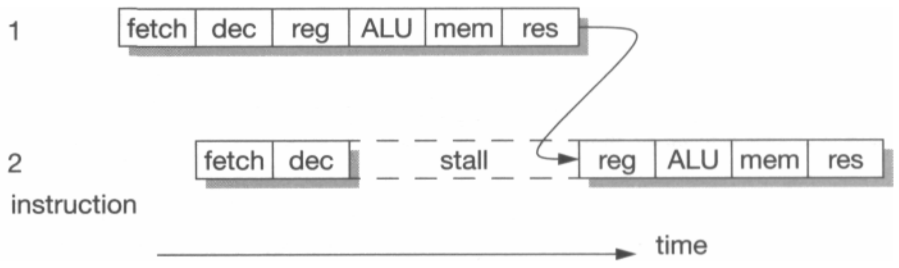


Figure 1.14 Read-after-write pipeline hazard.

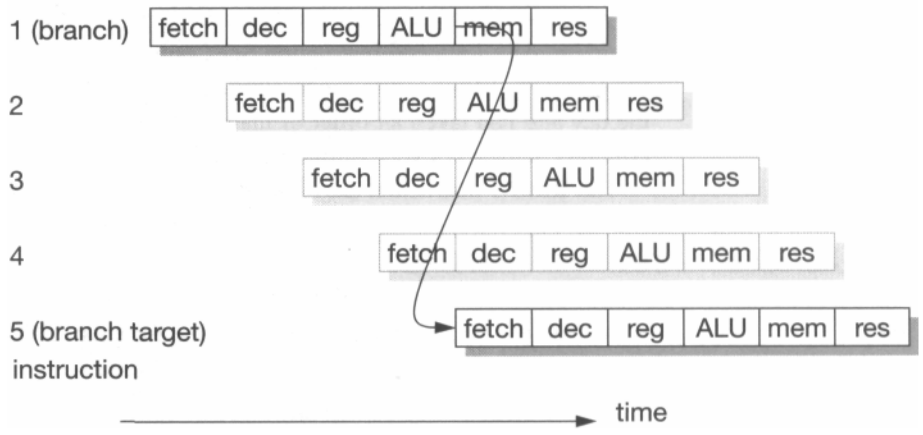


Figure 1.15 Pipelined branch behaviour.

define that the instruction following the branch is executed whether or not the branch is taken. This technique is known as the **delayed branch.**

Pipeline efficiency

Though there are techniques which reduce the impact of these pipeline problems, they cannot remove the difficulties altogether. The deeper the pipeline (that is, the more pipeline stages there are), the worse the problems get. For reasonably simple processors, there are significant benefits in introducing pipelines from three to five stages long, but beyond this the law of diminishing returns begins to apply and the added costs and complexity outweigh the benefits.

Pipelines clearly benefit from all instructions going through a similar sequence of steps. Processors with very complex instructions where every instruction behaves differently from the next are hard to pipeline. In 1980 the complex instruction set microprocessor of the day was not pipelined due to the limited silicon resource, the limited design resource and the high complexity of designing a pipeline for a complex instruction set.

1.6 The Reduced Instruction Set Computer

In 1980 Patterson and Ditzel published a paper entitled 'The Case for the Reduced Instruction Set Computer' (a full reference is given in the bibliography on page 410). In this seminal work they expounded the view that the optimal architecture for a single-chip processor need not be the same as the optimal architecture for a multi-chip processor. Their argument was subsequently supported by the results of a processor design project undertaken by a postgraduate class at Berkeley which incorporated a Reduced Instruction Set Computer (RISC) architecture. This design, the Berkeley RISC I, was much simpler than the commercial CISC processors of the day and had taken an order of magnitude less design effort to develop, but nevertheless delivered a very similar performance.

The RISC I instruction set differed from the minicomputer-like CISC instruction sets used on commercial microprocessors in a number of ways. It had the following key features:

RISC architecture

- A fixed (32-bit) instruction size with few formats; CISC processors typically had variable length instruction sets with many formats.
- A load-store architecture where instructions that process data operate only on registers and are separate from instructions that access memory; CISC processors typically allowed values in memory to be used as operands in data processing instructions.

- A large register bank of thirty-two 32-bit registers, all of which could be used for any purpose, to allow the load-store architecture to operate efficiently; CISC register sets were getting larger, but none was this large and most had different registers for different purposes (for example, the *data* and *address* registers on the Motorola MC68000).

These differences greatly simplified the design of the processor and allowed the designers to implement the architecture using organizational features that contributed to the performance of the prototype devices:

RISC organization

- Hard-wired instruction decode logic; CISC processors used large microcode ROMs to decode their instructions.
- Pipelined execution; CISC processors allowed little, if any, overlap between con secutive instructions (though they do now).
- Single-cycle execution; CISC processors typically took many clock cycles to complete a single instruction.

By incorporating all these architectural and organizational changes at once, the Berkeley RISC microprocessor effectively escaped from the problem that haunts progress by incremental improvement, which is the risk of getting stuck in a local maximum of the performance function.

RISC advantages

Patterson and Ditzel argued that RISC offered three principal advantages:

- A smaller die size.

  A simple processor should require fewer transistors and less silicon area. Therefore a whole CPU will fit on a chip at an earlier stage in process technology development, and once the technology has developed beyond the point where either CPU will fit on a chip, a RISC CPU leaves more die area free for performance-enhancing features such as cache memory, memory management functions, floating-point hardware, and so on.

- A shorter development time.

  A simple processor should take less design effort and therefore have a lower design cost and be better matched to the process technology when it is launched (since process technology developments need be predicted over a shorter development period).

- A higher performance.

  This is the tricky one! The previous two advantages are easy to accept, but in a world where higher performance had been sought through ever-increasing complexity, this was a bit hard to swallow.

  The argument goes something like this: smaller things have higher natural frequencies (insects flap their wings faster than small birds, small birds faster than large birds, and so on) so a simple processor ought to allow a high clock rate. So let's design our complex processor by starting with a simple one, then add complex instructions one at a time. When we add a complex instruction it will make some high-level function more efficient, but it will also slow the clock down a bit for all instructions. We can measure the overall benefit on typical programs, and when we do, all complex instructions make the program run slower. Hence we stick to the simple processor we started with.

These arguments were backed up by experimental results and the prototype processors (the Berkeley RISC II came shortly after RISC I). The commercial processor companies were sceptical at first, but most new companies designing processors for their own purposes saw an opportunity to reduce development costs and get ahead of the game. These commercial RISC designs, of which the ARM was the first, showed that the idea worked, and since 1980 all new general-purpose processor architectures have embraced the concepts of the RISC to a greater or lesser degree.

RISC in retrospect

Since the RISC is now well established in commercial use it is possible to look back and see more clearly what its contribution to the evolution of the microprocessor really was. Early RISCs achieved their performance through:

- Pipelining.

  Pipelining is the simplest form of concurrency to implement in a processor and delivers around two to three times speed-up. A simple instruction set greatly simplifies the design of the pipeline.

- A high clock rate with single-cycle execution.

  In 1980 standard semiconductor memories (DRAMs - Dynamic Random Access Memories) could operate at around 3 MHz for random accesses and at 6 MHz for sequential (page mode) accesses. The CISC microprocessors of the time could access memory at most at 2 MHz, so memory bandwidth was not being exploited to the full. RISC processors, being rather simpler, could be designed to operate at clock rates that would use all the available memory bandwidth.

Neither of these properties is a feature of the architecture, but both depend on the architecture being simple enough to allow the implementation to incorporate it. RISC architectures succeeded because they were simple enough to enable the designers to exploit these organizational techniques. It was entirely feasible to implement a fixed-length instruction load-store architecture using microcode, multi-cycle execution and no pipeline, but such an implementation would exhibit no advantage over an off-the-shelf CISC. It was *not* possible, at that time, to implement a hard-wired, single-cycle execution pipelined CISC. But it is now!

Clock rates

As footnotes to the above analysis, there are two aspects of the clock rate discussion that require further explanation:

- 1980s CISC processors often had higher clock rates than the early RISCs, but they took several clock cycles to perform a single memory access, so they had a lower memory access rate. Beware of evaluating processors on their clock rate alone!
- The mismatch between the CISC memory access rate and the available bandwidth appears to conflict with the comments in 'Complex Instruction Set Computers' on page 20 where microcode is justified in an early 1970s minicomputer on the grounds of the slow main memory speed relative to the processor speed. The resolution of the conflict lies in observing that in the intervening decade memory technology had become significantly faster while early CISC microprocessors were slower than typical minicomputer processors. This loss of processor speed was due to the necessity to switch from fast bipolar technologies to much slower NMOS technologies to achieve the logic density required to fit the complete processor onto a single chip.

RISC drawbacks

RISC processors have clearly won the performance battle and should cost less to design, so is a RISC all good news? With the passage of time, two drawbacks have come to light:

- RISCs generally have poor code density compared with CISCs.
- RISCs don't execute x86 code.

The second of these is hard to fix, though PC emulation software is available for many RISC platforms. It is only a problem, however, if you want to build an IBM PC compatible; for other applications it can safely be ignored.

The poor code density is a consequence of the fixed-length instruction set and is rather more serious for a wide range of applications. In the absence of a cache, poor code density leads to more main memory bandwidth being used for instruction fetching, resulting in a higher memory power consumption. When the processor incorporates an on-chip cache of a particular size, poor code density results in a smaller proportion of the working set being held in the cache at any time, increasing the cache miss rate, resulting in an even greater increase in the main memory bandwidth requirement and consequent power consumption.

ARM code density and Thumb

The ARM processor design is based on RISC principles, but for various reasons suffers less from poor code density than most other RISCs. Its code density is still, however, not as good as some CISC processors. Where code density is of prime importance, ARM Limited has incorporated a novel mechanism, called the Thumb architecture, into some versions of the ARM processor. The Thumb instruction set is a 16-bit compressed form of the original 32-bit ARM instruction set, and employs dynamic decompression hardware in the instruction pipeline. Thumb code density is better than that achieved by most CISC processors. The Thumb architecture is described in Chapter 7.

Beyond RISC

It seems unlikely that RISC represents the last word on computer architecture, so is there any sign of another breakthrough which will render the RISC approach obsolete? There is no development visible at the time of writing which suggests a change on the same scale as RISC, but instruction sets continue to evolve to give better support for efficient implementations and for new applications such as multimedia.

1.7 Design for low power consumption

Since the introduction of digital computers 50 years ago there has been sustained improvement in their cost-effectiveness at a rate unparalleled in any other technical endeavour. As a side-effect of the route taken to increased performance, the power consumption of the machines has reduced equally dramatically. Only very recently, however, has the drive for minimum power consumption become as important as, and in some application areas more important than, the drive for increased performance. This change has come about as a result of the growing market for battery-powered portable equipment, such as digital mobile telephones and lap-top computers, which incorporate high-performance computing components.

Following the introduction of the integrated circuit the computer business has been driven by the win-win scenario whereby smaller transistors yield lower cost, higher performance and lower power consumption. Now, though, designers are beginning to design specifically for low power, even, in some cases, sacrificing performance to achieve it.

The ARM processor is at the centre of this drive for power-efficient processing. It therefore seems appropriate to consider the issues around design for low power.

Where does the power go?

The starting point for low-power design is to understand where the power goes in existing circuits. CMOS is the dominant technology for modern high-performance digital electronics, and has itself some good properties for low-power design, so we start by looking at where the power goes in a CMOS circuit.

A typical CMOS circuit is the static NAND gate, illustrated in Figure 1.2 on page 4. All signals swing between the voltages of the power and ground rails, *Vdd* and *Vss,* Until recently a 5 volt supply was standard, but many modern CMOS processes require a lower supply voltage of around 3 volts and the latest technologies operate with supplies of between 1 and 2 volts, and this will reduce further in the future.

The gate operates by connecting the output either to *Vdd* through a pull-up network of p-type transistors, or to *Vss* through a pull-down network of n-type transistors. When the inputs are both close to one rail or the other, then one of these networks is conducting and the other is effectively not conducting, so there is no path through the gate from *Vdd* to *Vss.* Furthermore, the output is normally connected to the inputs of similar gates and therefore sees only capacitive load. Once the output has been driven close to either rail, it takes no current to hold it there. Therefore a short time after the gate has switched the circuit reaches a stable condition and no further current is taken from the supply.

This characteristic of consuming power only when switching is not shared by many other logic technologies and has been a major factor in making CMOS the technology of choice for high-density integrated circuits.

CMOS power components

The total power consumption of a CMOS circuit comprises three components:

- Switching power.

  This is the power dissipated by charging and discharging the gate output capacitance *CL,* and represents the useful work performed by the gate.

  The energy per output transition is:

$$E_t = \frac{1}{2} \cdot C_L \cdot Vdd^2 \approx 1\,\text{picojoule} \qquad \text{Equation 2}$$

- Short-circuit power.

  When the gate inputs are at an intermediate level both the p- and n-type networks can conduct. This results in a transitory conducting path from *Vdd* to *Vss.* With a correctly designed circuit (which generally means one that avoids slow signal transitions) the short-circuit power should be a small fraction of the switching power.

- Leakage current.

  The transistor networks do conduct a very small current when they are in their 'off' state; though on a conventional process this current is very small (a small fraction of a nanoamp per gate), it is the only dissipation in a circuit that is powered but inactive, and can drain a supply battery over a long period of time. It is generally negligible in an active circuit.

In a well-designed active circuit the switching power dominates, with the short-circuit power adding perhaps 10% to 20% to the total power, and the leakage current being significant only when the circuit is inactive. However, the trend to lower voltage operation does lead to a trade-off between performance and leakage current as discussed further below, and leakage is an increasing concern for future low-power high-performance designs.

CMOS circuit power

The total power dissipation, P_c, of a CMOS circuit, neglecting the short-circuit and leakage components, is therefore given by summing the dissipation of every gate *g* in the circuit C:

$$P_C = \frac{1}{2} \cdot f \cdot Vdd^2 \cdot \sum_{g \in C} A_g \cdot C_L^g \qquad \text{Equation 3}$$

where/is the clock frequency, A_g is the gate **activity factor** (reflecting the fact that not all gates switch every clock cycle) and C/ is the gate load capacitance. Note that within this summation clock lines, which make two transitions per clock cycle, have an activity factor of 2.

Low-power circuit design

The typical gate load capacitance is a function of the process technology and therefore not under the direct control of the designer. The remaining parameters in Equation 3 suggest various approaches to low-power design. These are listed below with the most important first:

1. Minimize the power supply voltage, *Vdd.*

   The quadratic contribution of the supply voltage to the power dissipation makes this an obvious target. This is discussed further below.
2. Minimize the circuit activity, *A.*

   Techniques such as clock gating fall under this heading. Whenever a circuit function is not needed, activity should be eliminated.
3. Minimize the number of gates.

   Simple circuits use less power than complex ones, all other things being equal, since the sum is over a smaller number of gate contributions.
4. Minimize the clock frequency, *f.*

   Avoiding unnecessarily high clock rates is clearly desirable, but although a lower clock rate reduces the power consumption it also reduces performance, having a neutral effect on power-efficiency (measured, for example, in MIPS - Millions of Instructions Per Second - per watt). If, however, a reduced clock frequency allows operation at a reduced *Vdd,* this will be highly beneficial to the power-efficiency.

Reducing *Vdd*

As the feature size on CMOS processes gets smaller, there is pressure to reduce the supply voltage. This is because the materials used to form the transistors cannot withstand an electric field of unlimited strength, and as transistors get smaller the field strength increases if the supply voltage is held constant.

However, with increasing interest in design specifically for low power, it may be desirable for the supply voltage to be reduced faster than is necessary solely to prevent electrical breakdown. What prevents very low supply voltages from being used now?

The problem with reducing *Vdd* is that this also reduces the performance of the circuit. The saturated transistor current is given by:

$$I_{sat} \propto (Vdd - V_t)^2 \qquad \text{Equation 4}$$

where V_t is the transistor threshold. The charge on a circuit node is proportional to *Vdd,* so the maximum operating frequency is given by:

$$f_{max} \propto \frac{(Vdd - V_t)^2}{Vdd} \qquad \text{Equation 5}$$

Therefore the maximum operating frequency is reduced as *Vdd* is reduced. The performance loss on a sub-micron process may not be as severe as Equation 5 suggests since the current at high voltage may be limited by velocity saturation effects, but performance will be lost to some extent. Equation 5 suggests that an obvious way to ameliorate the performance loss would be to reduce V_t. However the leakage current depends strongly on V_t:

$$I_{leak} \propto \exp\left(-\frac{V_t}{35\ \text{mV}}\right) \qquad \text{Equation 6}$$

Even a small reduction in V_t can significantly increase the leakage current, increasing the battery drain through an inactive circuit. There is therefore a trade-off to be struck between maximizing performance and minimizing standby power, and this issue must be considered carefully by designers of systems where both characteristics are important.

Even where standby power is not important designers must be aware that maximizing performance by using very low threshold transistors can increase the leakage power to the point where is becomes comparable with the dynamic power, and therefore leakage power must be taken into consideration when selecting packaging and cooling systems.

Low-power strategies

To conclude this introduction to design techniques for low power consumption, here are some suggested strategies for low-power applications.

- Minimize *Vdd.*

  Choose the lowest clock frequency that delivers the required performance, then set the power supply voltage as low as is practical given the clock frequency and the requirements of the various system components. Be wary of reducing the supply voltage so far that leakage compromises standby power.

- Minimize off-chip activity.

  Off-chip capacitances are much higher than on-chip loads, so always minimize off-chip activity. Avoid allowing transients to drive off-chip loads and use caches to minimize accesses to off-chip memories.

- Minimize on-chip activity.

  Lower priority than minimizing off-chip activity, it is still important to avoid clocking unnecessary circuit functions (for example, by using gated clocks) and to employ sleep modes where possible.

• Exploit parallelism.

Where the power supply voltage is a free variable parallelism can be exploited to improve power-efficiency. Duplicating a circuit allows the two circuits to sustain the same performance at half the clock frequency of the original circuit, which allows the required performance to be delivered with a lower supply voltage.

Design for low power is an active research area and one where new ideas are being generated at a high rate. It is expected that a combination of process and design technology improvements will yield considerable further improvement in the power-efficiency of high-speed digital circuits over the next decade.

1.8 Examples and exercises

(The more practical exercises will require you to have access to some form of hardware simulation environment.)

Example 1.1 **Design a 4-bit binary counter using logic gates and a 4-bit register.**

If the register inputs are denoted by *D[0]* to *D[3]* and its outputs are denoted by *Q[0]* to *Q[3],* the counter may be implemented by building combinatorial logic that generates *D[3:0]* = *Q[3:0]* + 1. The logic equations for a binary adder are given in the Appendix (Equation 20 on page 401 gives the sum and Equation 21 the carry). When the second operand is a constant these equations simplify to:

$$D[0] = \overline{Q[0]} \qquad \text{Equation 7}$$

$$D[i] = Q[i]\cdot\overline{C[i-1]} + \overline{Q[i]}\cdot C[i-1] \qquad \text{Equation 8}$$

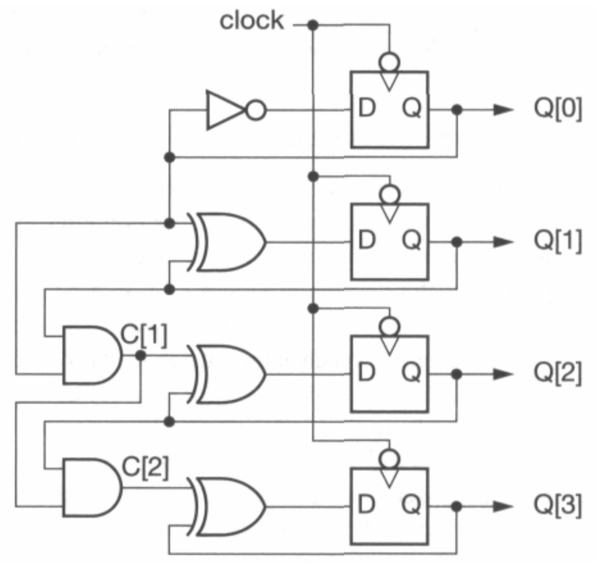
$$C[i] = Q[i]\cdot C[i-1] \qquad \text{Equation 9}$$

for $1 < i < 3$, and *C[0]* = 1. *(C[3]* is not needed.) These equations may be drawn as the logic circuit shown on page 33, which also includes the register.

Exercise 1.1.1 Modify the binary counter to count from 0 to 9, and then, on the next clock edge, to start again at zero. (This is a **modulo 10** counter.)

Exercise 1.1.2 Modify the binary counter to include a **synchronous clear** function. This means adding a new input ('clear') which, if active, causes the counter output to be zero after the next clock edge whatever its current value is.

Exercise 1.1.3 Modify the binary counter to include an **up/down** input. When this input is high the counter should behave as described in the example above; when it is low the counter should count down (in the reverse sequence to the *up* mode).



Example 1.2

Add indexed addressing to the MU0 instruction set.

The minimum extension that *is* useful here is to introduce a new 12-bit index register (X) and some new instructions that allow it to be initialized and used in load and store instructions. Referring to Table 1.1 on page 8, there are eight unused opcodes in the original design, so we could add up to eight new instructions before we run out of space. The basic set of indexing operations is:

```
LDX S LDA        ; X := mem16[S]
S, X STA S,      ; ACC := mem16[S+X]
X                ; mem16[S+X] := ACC
```

An index register is much more useful if there is some way to modify it, for instance to step through a table:

```
INX DEX          ;   X   :=   X
                 +   1 ;
                 X   :=   X   -
                 1
```

This gives the basic functionality of an index register. It would increase the usefulness of X to include a way to store it in memory, then it could be used as a temporary register, but for simplicity we will stop here.

Exercise 1.2.1 Modify the RTL organization shown in Figure 1.6 on page 11 to include the X register, indicating the new control signals required.

Exercise 1.2.2

Modify the control logic in Table 1.2 on page 12 to support indexed addressing. If you have access to a hardware simulator, test your design. (This is non-trivial!)

Example 1.3

Estimate the performance benefit of a single-cycle delayed branch.

A delayed branch allows the instruction following the branch to be executed whether or not the branch is taken. The instruction after the branch is in the 'delay slot'. Assume the dynamic instruction frequencies shown in Table 1.3 on page 21 and the pipeline structure shown in Figure 1.13 on page 22; ignore register hazards; assume all delay slots can be filled (most single delay slots can be filled).

If there is a dedicated branch target adder in the decode stage, a branch has a 1-cycle delayed effect, so a single delay slot removes all wasted cycles. One instruction in four is a branch, so four instructions take five clock cycles without the delay slot and four with it, giving 25% higher performance.

If there is no dedicated branch target adder and the main ALU stage is used to compute the target, a branch will incur three wasted cycles. Therefore four instructions on average include one branch and take seven clock cycles, or six with a single delay slot. The delay slot therefore gives 17% higher performance (but the dedicated branch adder does better even without the delay slot).

Exercise 1.3.1

Estimate the performance benefit of a 2-cycle delayed branch assuming that all the first delay slots can be filled, but only 50% of the second delay slots can be filled.

Why is the 2-cycle delayed branch only relevant if there is no dedicated branch target adder?

Exercise 1.3.2

What is the effect on code size of the 1- and 2-cycle delayed branches suggested above? (All unfilled branch delay slots must be filled with no-ops.)

2 The ARM Architecture

Summary of chapter contents

The ARM processor is a *Reduced Instruction Set Computer* (RISC). The RISC concept, as we saw in the previous chapter, originated in processor research programmes at Stanford and Berkeley universities around 1980.

In this chapter we see how the RISC ideas helped shape the ARM processors. The ARM was originally developed at Acorn Computers Limited of Cambridge, England, between 1983 and 1985. It was the first RISC microprocessor developed for commercial use and has some significant differences from subsequent RISC architectures. The principal features of the ARM architecture are presented here in overview form; the details are postponed to subsequent chapters.

In 1990 ARM Limited was established as a separate company specifically to widen the exploitation of ARM technology, since when the ARM has been licensed to many semiconductor manufacturers around the world. It has become established as a market-leader for low-power and cost-sensitive embedded applications.

No processor is particularly useful without the support of hardware and software development tools. The ARM is supported by a toolkit which includes an instruction set emulator for hardware modelling and software testing and benchmarking, an assembler, C and C++ compilers, a linker and a symbolic debugger.

2.1 The Acorn RISC Machine

The first ARM processor was developed at Acorn Computers Limited, of Cambridge, England, between October 1983 and April 1985. At that time, and until the formation of Advanced RISC Machines Limited (which later was renamed simply ARM Limited) in 1990, ARM stood for Acorn **RISC Machine.**

Acorn had developed a strong position in the UK personal computer market due to the success of the BBC (British Broadcasting Corporation) microcomputer. The BBC micro was a machine powered by the 8-bit 6502 microprocessor and rapidly became established as the dominant machine in UK schools following its introduction in January 1982 in support of a series of television programmes broadcast by the BBC. It also enjoyed enthusiastic support in the hobbyist market and found its way into a number of research laboratories and higher education establishments.

Following the success of the BBC micro, Acorn's engineers looked at various microprocessors to build a successor machine around, but found all the commercial offerings lacking. The 16-bit CISC microprocessors that were available in 1983 were slower than standard memory parts. They also had instructions that took many clock cycles to complete (in some cases, many hundreds of clock cycles), giving them very long interrupt latencies. The BBC micro benefited greatly from the 6502's rapid interrupt response, so Acorn's designers were unwilling to accept a retrograde step in this aspect of the processor's performance.

As a result of these frustrations with the commercial microprocessor offerings, the design of a proprietary microprocessor was considered. The major stumbling block was that the Acorn team knew that commercial microprocessor projects had absorbed hundreds of man-years of design effort. Acorn could not contemplate an investment on that scale since it was a company of only just over 400 employees in total. It had to produce a better design with a fraction of the design effort, and with no experience in custom chip design beyond a few small gate arrays designed for the BBC micro.

Into this apparently impossible scenario, the papers on the Berkeley RISC I fell like a bolt from the blue. Here was a processor which had been designed by a few postgraduate students in under a year, yet was competitive with the leading commercial offerings. It was inherently simple, so there were no complex instructions to ruin the interrupt latency. It also came with supporting arguments that suggested it could point the way to the future, though technical merit, however well supported by academic argument, is no guarantee of commercial success.

The ARM, then, was born through a serendipitous combination of factors, and became the core component in Acorn's product line. Later, after a judicious modification of the acronym expansion to **Advanced RISC Machine,** it lent its name to the company formed to broaden its market beyond Acorn's product range. Despite the change of name, the architecture still remains close to the original Acorn design.

2.2 Architectural inheritance

At the time the first ARM chip was designed, the only examples of RISC architectures were the Berkeley RISC I and II and the Stanford MIPS (which stands for **Microprocessor without Interlocking Pipeline** Stages), although some earlier machines such as the Digital PDP-8, the Cray-1 and the IBM 801, which predated the RISC concept, shared many of the characteristics which later came to be associated with RISCs.

Features used

The ARM architecture incorporated a number of features from the Berkeley RISC design, but a number of other features were rejected. Those that were used were:

- a load-store architecture;
- fixed-length 32-bit instructions;
- 3-address instruction formats.

Features rejected

The features that were employed on the Berkeley RISC designs which were rejected by the ARM designers were:

- Register windows.

  The register banks on the Berkeley RISC processors incorporated a large number of registers, 32 of which were visible at any time. Procedure entry and exit instructions moved the visible 'window' to give each procedure access to new registers, thereby reducing the data traffic between the processor and memory resulting from register saving and restoring.

  The principal problem with register windows is the large chip area occupied by the large number of registers. This feature was therefore rejected on cost grounds, although the shadow registers used to handle exceptions on the ARM are not too different in concept.

  In the early days of RISC the register window mechanism was strongly associated with the RISC idea due to its inclusion in the Berkeley prototypes, but subsequently only the Sun SPARC architecture has adopted it in its original form

- Delayed branches.

  Branches cause pipelines problems since they interrupt the smooth flow of instructions. Most RISC processors ameliorate the problem by using delayed branches where the branch takes effect *after* the following instruction has executed.

  The problem with delayed branches is that they remove the atomicity of individual instructions. They work well on single issue pipelined processors, but they do not scale well to super-scalar implementations and can interact badly with branch prediction mechanisms.

On the original ARM delayed branches were not used because they made exception handling more complex; in the long run this has turned out to be a good decision since it simplifies re-implementing the architecture with a different pipeline.

- Single-cycle execution of all instructions.

  Although the ARM executes most data processing instructions in a single clock cycle, many other instructions take multiple clock cycles.

  The rationale here was based on the observation that with a single memory for both data and instructions, even a simple load or store instruction requires at least two memory accesses (one for the instruction and one for the data). Therefore single cycle operation of all instructions is only possible with separate data and instruction memories, which were considered too expensive for the intended ARM application areas.

  Instead of single-cycle execution of all instructions, the ARM was designed to use the minimum number of cycles required for memory accesses. Where this was greater than one, the extra cycles were used, where possible, to do something useful, such as support auto-indexing addressing modes. This reduces the total number of ARM instructions required to perform any sequence of operations, improving performance and code density.

Simplicity

An overriding concern of the original ARM design team was the need to keep the design simple. Before the first ARM chips, Acorn designers had experience only of gate arrays with complexities up to around 2,000 gates, so the full-custom CMOS design medium was approached with some respect. When venturing into unknown territory it is advisable to minimize those risks which are under your control, since this still leaves significant risks from those factors which are not well understood or are fundamentally not controllable.

The simplicity of the ARM may be more apparent in the hardware organization and implementation (described in Chapter 4) than it is in the instruction set architecture. From the programmer's perspective it is perhaps more visible as a conservatism in the ARM instruction set design which, while accepting the fundamental precepts of the RISC approach, is less radical than many subsequent RISC designs.

The combination of the simple hardware with an instruction set that is grounded in RISC ideas but retains a few key CISC features, and thereby achieves a significantly better code density than a pure RISC, has given the ARM its power-efficiency and its small core size.

2.3 The ARM programmer's model

A processor's instruction set defines the operations that the programmer can use to change the state of the system incorporating the processor. This state usually comprises the values of the data items in the processor's visible registers and the system's memory. Each instruction can be viewed as performing a defined transformation from the state before the instruction is executed to the state after it has completed. Note that although a processor will typically have many invisible registers involved in executing an instruction, the values of these registers before and after the instruction is executed are not significant; only the values in the visible registers have any significance. The visible registers in an ARM processor are shown in Figure 2.1.

When writing user-level programs, only the 15 general-purpose 32-bit registers (r0 to r!4), the program counter (r15) and the current program status register (CPSR) need be considered. The remaining registers are used only for system-level programming and for handling exceptions (for example, interrupts).

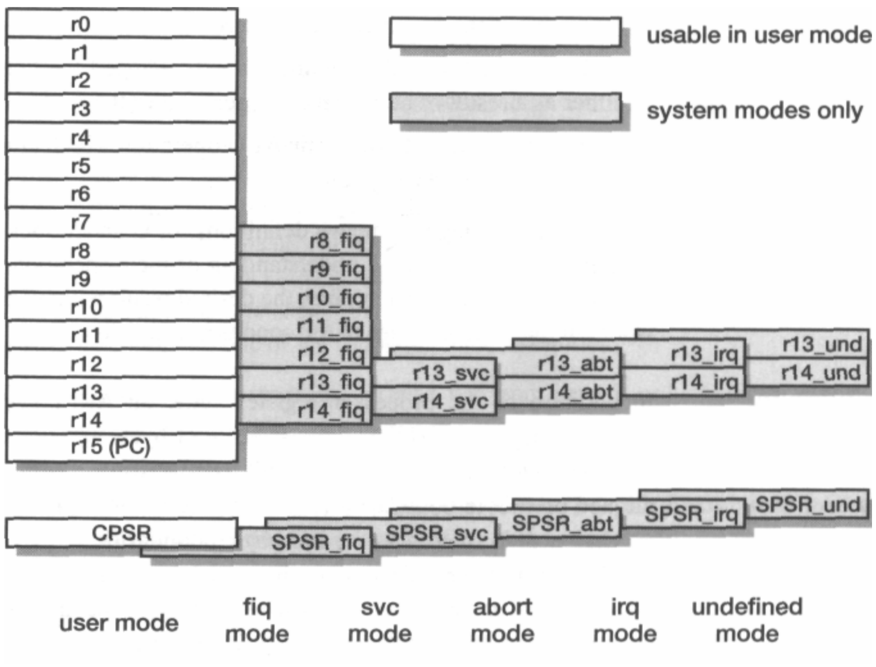


Figure 2.1 ARM's visible registers.

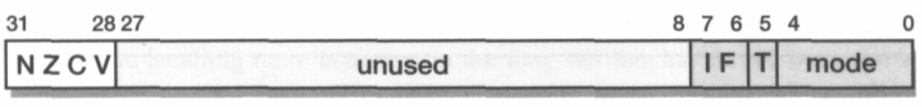


Figure 2.2 ARM CPSR format.

The Current Program Status Register (CPSR)

The CPSR is used in user-level programs to store the condition code bits. These bits are used, for example, to record the result of a comparison operation and to control whether or not a conditional branch is taken. The user-level programmer need not usually be concerned with how this register is configured, but for completeness the register is illustrated in Figure 2.2. The bits at the bottom of the register control the processor mode (see Section 5.1 on page 106), instruction set ('T', see Section 7.1 on page 189) and interrupt enables ('I' and 'F', see Section 5.2 on page 108) and are protected from change by the user-level program. The condition code flags are in the top four bits of the register and have the following meanings:

- N: Negative; the last ALU operation which changed the flags produced a negative result (the top bit of the 32-bit result was a one).
- Z: Zero; the last ALU operation which changed the flags produced a zero result (every bit of the 32-bit result was zero).
- C: Carry; the last ALU operation which changed the flags generated a carry-out, either as a result of an arithmetic operation in the ALU or from the shifter.
- V: oVerflow; the last arithmetic ALU operation which changed the flags generated an overflow into the sign bit.

Note that although the above definitions for C and V look quite complex, their use does not require a detailed understanding of their operation. In most cases there is a simple condition test which gives the desired result without the programmer having to work out the precise values of the condition code bits.

The memory system

In addition to the processor register state, an ARM system has memory state. Memory may be viewed as a linear array of bytes numbered from zero up to 2^{32}-1. Data items may be 8-bit bytes, 16-bit half-words or 32-bit words. Words are always aligned on 4-byte boundaries (that is, the two least significant address bits are zero) and half-words are aligned on even byte boundaries.

The memory organization is illustrated in Figure 2.3 on page 41. This shows a small area of memory where each byte location has a unique number. A byte may occupy any of these locations, and a few examples are shown in the figure. A word-sized data item must occupy a group of four byte locations starting at a byte address which is a multiple of four, and again the figure contains a couple of examples. Half-words occupy two byte locations starting at an even byte address.

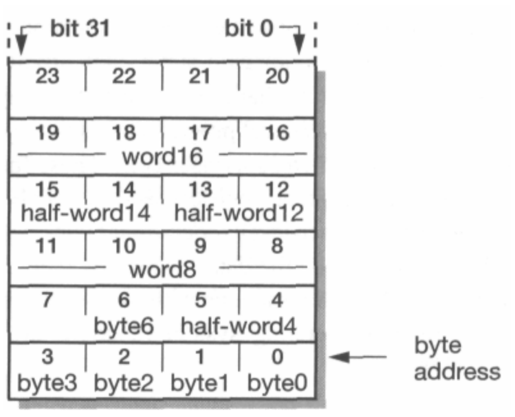


Figure 2.3 ARM memory organization.

(This is the standard, 'little-endian', memory organization used by the ARM. ARM can also be configured to work with a 'big-endian' memory organization; we will return to this issue in Chapter 5.)

Load-store architecture

In common with most RISC processors, ARM employs a load-store architecture. This means that the instruction set will only process (add, subtract, and so on) values which are in registers (or specified directly within the instruction itself), and will always place the results of such processing into a register. The only operations which apply to memory state are ones which copy memory values into registers (load instructions) or copy register values into memory (store instructions).

CISC processors typically allow a value from memory to be added to a value in a register, and sometimes allow a value in a register to be added to a value in memory. ARM does not support such 'memory-to-memory' operations. Therefore all ARM instructions fall into one of the following three categories:

1. Data processing instructions. These use and change only register values. For example, an instruction can add two registers and place the result in a register.
2. Data transfer instructions. These copy memory values into registers (load instructions) or copy register values into memory (store instructions). An addi tional form, useful only in systems code, exchanges a memory value with a reg ister value.
3. Control flow instructions. Normal instruction execution uses instructions stored at consecutive memory addresses. Control flow instructions cause execution to switch to a different address, either permanently (branch instructions) or saving a return address to resume the original sequence (branch and link instructions) or trapping into system code (supervisor calls).

Supervisor mode

The ARM processor supports a protected supervisor mode. The protection mechanism ensures that user code cannot gain supervisor privileges without appropriate checks being carried out to ensure that the code is not attempting illegal operations. The upshot of this for the user-level programmer is that system-level functions can only be accessed through specified supervisor calls. These functions generally include any accesses to hardware peripheral registers, and to widely used operations such as character input and output. User-level programmers are principally concerned with devising algorithms to operate on the data 'owned' by their programs, and rely on the operating system to handle all transactions with the world outside their programs. The instructions which request operating system functions are covered in 'Supervisor calls' on page 67.

The ARM instruction set

All ARM instructions are 32 bits wide (except the compressed 16-bit Thumb instructions which are described in Chapter 7) and are aligned on 4-byte boundaries in memory. Basic use of the instruction set is described in Chapter 3 and full details, including the binary instruction formats, are given in Chapter 5. The most notable features of the ARM instruction set are:

- The load-store architecture;
- 3-address data processing instructions (that is, the two source operand registers and the result register are all independently specified);
- conditional execution of every instruction;
- the inclusion of very powerful load and store multiple register instructions;
- the ability to perform a general shift operation and a general ALU operation in a single instruction that executes in a single clock cycle;
- open instruction set extension through the coprocessor instruction set, including adding new registers and data types to the programmer's model;
- a very dense 16-bit compressed representation of the instruction set in the Thumb architecture.

To those readers familiar with modern RISC instruction sets, the ARM instruction set may appear to have rather more formats than other commercial RISC processors. While this is certainly the case and it does lead to more complex instruction decoding, it also leads to higher code density. For the small embedded systems that most ARM processors are used in, this code density advantage outweighs the small performance penalty incurred by the decode complexity. Thumb code extends this advantage to give ARM better code density than most CISC processors.

The I/O system

The ARM handles I/O (input/output) peripherals (such as disk controllers, network interfaces, and so on) as memory-mapped devices with interrupt support. The internal registers in these devices appear as addressable locations within the ARM's memory map and may be read and written using the same (load-store) instructions as any other memory locations.

Peripherals may attract the processor's attention by making an interrupt request using either the normal interrupt *(IRQ)* or the fast interrupt *(FIQ)* input. Both interrupt inputs are level-sensitive and maskable. Normally most interrupt sources share the IRQ input, with just one or two time-critical sources connected to the higher-priority FIQ input.

Some systems may include direct memory access (DMA) hardware external to the processor to handle high-bandwidth I/O traffic. This is discussed further in Section 11.9 on page 312.

Interrupts are a form *of exception* and are handled as outlined below.

ARM exceptions

The ARM architecture supports a range of interrupts, traps and supervisor calls, all grouped under the general heading of exceptions. The general way these are handled is the same in all cases:

1. The current state is saved by copying the PC into *rl4_exc* and the CPSR into SPSR_exc (where *exc* stands for the exception type).
2. The processor operating mode is changed to the appropriate exception mode.
3. The PC is forced to a value between 00_{16} and $1C_{16}$, the particular value depending on the type of exception.

The instruction at the location the PC is forced to (the *vector address)* will usually contain a branch to the exception handler. The exception handler will use rl3_exc, which will normally have been initialized to point to a dedicated stack in memory, to save some user registers for use as work registers.

The return to the user program is achieved by restoring the user registers and then using an instruction to restore the PC and the CPSR atomically. This may involve some adjustment of the PC value saved in *rl4_exc* to compensate for the state of the pipeline when the exception arose. This is described in more detail in Section 5.2 on page 108.

2.4 ARM development tools

Software development for the ARM is supported by a coherent range of tools developed by ARM Limited, and there are also many third party and public domain tools available, such as an ARM back-end for the *gcc C* compiler.

Since the ARM is widely used as an embedded controller where the target hardware will not make a good environment for software development, the tools are intended for **cross-development** (that is, they run on a different architecture from the one for which they produce code) from a platform such as a PC running Windows or a suitable UNIX workstation. The overall structure of the ARM cross-development toolkit is shown in Figure 2.4. C or assembler source files are compiled or assembled into ARM object format *(.aof)* files, which are then linked into ARM image format *(.aif)* files. The image format files can be built to include the debug tables required by the ARM symbolic debugger (ARMsd which can load, run and debug programs either on hardware such as the ARM Development Board or using a software emulation of the ARM (the ARMulator). The ARMulator has been designed to allow easy extension of the software model to include system features such as caches, particular memory timing characteristics, and so on.

The ARM C compiler

The ARM C compiler is compliant with the ANSI (American National Standards Institute) standard for C and is supported by the appropriate library of standard functions. It uses the ARM Procedure Call Standard (see Section 6.8 on page 175) for all externally available functions. It can be told to produce assembly source output instead of ARM object format, so the code can be inspected, or even hand optimized, and then assembled subsequently. The compiler can also produce Thumb code.

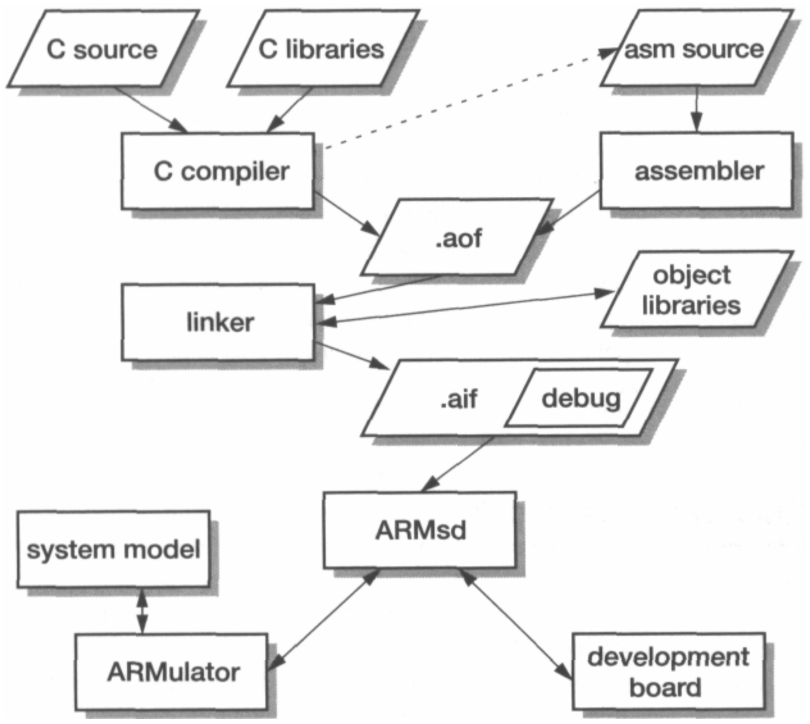


Figure 2.4 The structure of the ARM cross-development toolkit.

The ARM assembler

The ARM assembler is a full macro assembler which produces ARM object format output that can be linked with output from the C compiler.

Assembly source language is near machine-level, with most assembly instructions translating into single ARM (or Thumb) instructions. ARM assembly language programming is introduced in the next chapter, and the full details of the ARM instruction set, including assembly language formats, are given in Chapter *5.* The Thumb instruction set and assembly language formats are given in Chapter 7.

The linker

The linker takes one or more object files and combines them into an executable program. It resolves symbolic references between the object files and extracts object modules from libraries as needed by the program. It can assemble the various components of the program in a number of different ways, depending on whether the code is to run in RAM (Random Access Memory, which can be read and written) or ROM (Read Only Memory), whether overlays are required, and so on.

Normally the linker includes debug tables in the output file. If the object files were compiled with full debug information, this will include full symbolic debug tables (so the program can be debugged using the variable names in the source program). The linker can also produce object library modules that are not executable but are ready for efficient linking with object files in the future.

ARMsd

The ARM symbolic debugger is a front-end interface to assist in debugging programs running either under emulation (on the ARMulator) or remotely on a target system such as the ARM development board. The remote system must support the appropriate remote debug protocols either via a serial line or through a JTAG test interface (see Section 8.6 on page 226). Debugging a system where the processor core is embedded within an application-specific system chip is a complex issue that we will return to in Chapter 8.

At its most basic, ARMsd allows an executable program to be loaded into the ARMulator or a development board and run. It allows the setting of breakpoints, which are addresses in the code that, if executed, cause execution to halt so that the processor state can be examined. In the ARMulator, or when running on hardware with appropriate support, it also allows the setting of watchpoints. These are memory addresses that, if accessed as data addresses, cause execution to halt in a similar way.

At a more sophisticated level ARMsd supports full source level debugging, allowing the C programmer to debug a program using the source file to specify breakpoints and using variable names from the original program.

ARMulator

The ARMulator *(ARM emulator)* is a suite of programs that models the behaviour of various ARM processor cores in software on a host system. It can operate at various levels of accuracy:

- *Instruction-accurate* modelling gives the exact behaviour of the system state without regard to the precise timing characteristics of the processor.
- *Cycle-accurate* modelling gives the exact behaviour of the processor on a cycle-by-cycle basis, allowing the exact number of clock cycles that a program requires to be established.
- *Timing-accurate* modelling presents signals at the correct time within a cycle, allowing logic delays to be accounted for.

All these approaches run considerably slower than the real hardware, but the first incurs the smallest speed penalty and is best suited to software development.

At its simplest, the ARMulator allows an ARM program developed using the C compiler or assembler to be tested and debugged on a host machine with no ARM processor connected. It allows the number of clock cycles the program takes to execute to be measured exactly, so the performance of the target system can be evaluated.

At its most complex, the ARMulator can be used as the centre of a complete, timing-accurate, C model of the target system, with full details of the cache and memory management functions added, running an operating system.

In between these two extremes the ARMulator comes with a set of model prototyping modules including a rapid prototype memory model and coprocessor interfacing support. (There is more detail on this in Section 8.5 on page 225.)

The ARMulator can also be used as the core of a timing-accurate ARM behavioural model in a hardware simulation environment based around a language such as VHDL. (VHDL is a standard, widely supported hardware description language.) A VHDL 'wrapper' must be generated to interface the ARMulator C code to the VHDL environment.

ARM development board

The ARM Development Board is a circuit board incorporating a range of components and interfaces to support the development of ARM-based systems. It includes an ARM core (for example, an ARM7TDMI), memory components which can be configured to match the performance and bus-width of the memory in the target system, and electrically programmable devices which can be configured to emulate application-specific peripherals. It can support both hardware and software development before the final application-specific hardware is available.

Software Toolkit

ARM Limited supplies the complete set of tools described above, with some support utility programs and documentation, as the 'ARM Software Development Toolkit'. The Toolkit CD-ROM includes a PC version of the toolset that runs under most versions of the Windows operating system and includes a full Windows-based project manager. The toolkit is updated as new versions of the ARM become available.

The ARM Project Manager is a graphical front-end for the tools described above. It supports the building of a single library or executable image from a list of files that make up a particular project. These files may be:

- source files (C, assembler, and so on);
- object files;
- library files.

The source files may be edited within the Project Manager, a dependency list created and the output library or executable image built. There are many options which may be chosen for the build, such as:

- Whether the output should be optimized for code size or execution time.
- Whether the output should be in debug or release form.

  (Code compiled for source-level debugging cannot be fully optimized since the mapping from the source to fully optimized output is too obscure for debugging purposes.)
- Which ARM processor is the target (and, particularly, whether it supports the Thumb instruction set).

The CD-ROM also contains versions of the tools that run on a Sun or HP UNIX host, where a command-line interface is used. All versions have on-line help available.

JumpStart

The JumpStart tools from VLSI Technology, Inc., include the same basic set of development tools but present a full X-windows interface on a suitable workstation rather than the command-line interface of the standard ARM toolkit. There are many other suppliers of tools that support ARM development.

2.5 Example and exercises

Example 2.1 **Describe the principal features of the ARM architecture.**

The main features of the ARM architecture are:

- a large set of registers, all of which can be used for most purposes;
- a load-store architecture;
- 3-address instructions (that is, the two source operand registers and the result reg ister are all independently specified);
- conditional execution of every instruction;
- the inclusion of very powerful load and store multiple register instructions;
- the ability to perform a general shift operation and a general ALU operation in a single instruction that executes in a single clock cycle;
- open instruction set extension through the coprocessor instruction set, including adding new registers and data types to the programmer's model.

If the Thumb instruction set is considered part of the ARM architecture, we could also add:

- a very dense 16-bit compressed representation of the instruction set in the Thumb architecture.

Exercise 2.1.1 Which features does ARM have in common with many other RISC architectures?

Exercise 2.1.2 Which features of the ARM architecture are not shared by most other RISCs? Which

Exercise 2.1.3 features of most other RISC architectures are not shared by the ARM?

3 ARM Assembly Language Programming

Summary of chapter contents

The ARM processor is very easy to program at the assembly level, though for most applications it is more appropriate to program in a high-level language such as C or C++.

Assembly language programming requires the programmer to think at the level of the individual machine instruction. An ARM instruction is 32 bits long, so there are around 4 billion different binary machine instructions. Fortunately there is considerable structure within the instruction space, so the programmer does not have to be familiar with each of the 4 billion binary encodings on an individual basis. Even so, there is a considerable amount of detail to be got right in each instruction. The assembler is a computer program which handles most of this detail for the programmer.

In this chapter we will look at ARM assembly language programming at the user level and see how to write simple programs which will run on an ARM development board or an ARM emulator (for example, the ARMulator which comes as part of the ARM development toolkit). Once the basic instruction set is familiar we will move on, in Chapter 5, to look at system-level programming and at some of the finer details of the ARM instruction set, including the binary-level instruction encoding.

Some ARM processors support a form of the instruction set that has been compressed into 16-bit Thumb' instructions. These are discussed in Chapter 7.

3.1 Data processing instructions

ARM data processing instructions enable the programmer to perform arithmetic and logical operations on data values in registers. All other instructions just move data around and control the sequence of program execution, so the data processing instructions are the only instructions which modify data values. These instructions typically require two operands and produce a single result, though there are exceptions to both of these rules. A characteristic operation is to add two values together to produce a single result which is the sum.

Here are some rules which apply to ARM data processing instructions:

- All operands are 32 bits wide and come from registers or are specified as literals in the instruction itself.
- The result, if there is one, is 32 bits wide and is placed in a register.

  (There is an exception here: long multiply instructions produce a 64-bit result; they are discussed in Section 5.8 on page 122.)
- Each of the operand registers and the result register are independently specified in the instruction. That is, the ARM uses a '3-address' format for these instructions.

Simple register operands

A typical ARM data processing instruction is written in assembly language as shown below:

```
                r0,    r1,    r2 ADD        r0,    r1,
                                 r2         ;   r0   :=
        r1   +   r2
```

The semicolon in this line indicates that everything to the right of it is a comment and should be ignored by the assembler. Comments are put into the assembly source code to make reading and understanding it easier.

This example simply takes the values in two registers (r1 and r2), adds them together, and places the result in a third register (r0). The values in the source registers are 32 bits wide and may be considered to be either unsigned integers or signed 2's-complement integers. The addition may produce a carry-out or, in the case of signed 2's-complement values, an internal overflow into the sign bit, but in either case this is ignored.

Note that in writing the assembly language source code, care must be taken to write the operands in the correct order, which is result register first, then the first operand and lastly the second operand (though for commutative operations the order of the first and second operands is not significant when they are both registers). When this instruction is executed the only change to the system state is the value of the destination register r0 (and, optionally, the N, Z, C and V flags in the CPSR, as we shall see later).

The different instructions available in this form are listed below in their classes:

• Arithmetic operations.

These instructions perform binary arithmetic (addition, subtraction and reverse subtraction, which is subtraction with the operand order reversed) on two 32-bit operands. The operands may be unsigned or 2's-complement signed integers; the carry-in, when used, is the current value of the C bit in the CPSR.

```
ADD     r0, r1, r2          ; r0 := r1 + r2
ADC     r0, r1, r2          ; r0 := r1 + r2 + C
SUB     r0, r1, r2          ; r0 := r1 - r2
SBC     r0, r1, r2          ; r0 := r1 - r2 + C - 1
RSB     r0, r1, r2          ; r0 := r2 - r1
RSC     r0, r1, r2          ; r0 := r2 - r1 + C - 1
```

'ADD' is simple addition, 'ADC' is add with carry, 'SUB' is subtract, 'SBC' is subtract with carry, 'RSB' is reverse subtraction and 'RSC' reverse subtract with carry.

• Bit-wise logical operations.

These instructions perform the specified Boolean logic operation on each bit pair of the input operands, so in the first case *r0[i]:= r1[i]* AND *r2[i]* for each value of *i* from 0 to 31 inclusive, where *r0[i]* is the *i*th bit of r0.

```
AND     r0, r1, r2          ; r0 := r1 and r2
ORR     r0, r1, r2          ; r0 := r1 or r2
EOR     r0, r1, r2          ; r0 := r1 xor r2
BIC     r0, r1, r2          ; r0 := r1 and not r2
```

We have met AND, OR and XOR (here called EOR) logical operations at the hardware gate level in Section 1.2 on page 3; the final mnemonic, BIC, stands for 'bit clear' where every ' 1' in the second operand clears the corresponding bit in the first. (The 'not' operation in the assembly language comment inverts each bit of the following operand.)

• Register movement operations.

These instructions ignore the first operand, which is omitted from the assembly language format, and simply move the second operand (possibly bit-wise inverted) to the destination.

```
MOV     r0, r2              ; r0 := r2
MVN     r0, r2              ; r0 := not r2
```

The 'MVN' mnemonic stands for 'move negated'; it leaves the result register set to the value obtained by inverting every bit in the source operand.

• Comparison operations.

These instructions do not produce a result (which is therefore omitted from the assembly language format) but just set the condition code bits (N, Z, C and V) in the CPSR according to the selected operation.

```
CMP CMN  r1, r2            ; set cc on r1 - r2
TST TEQ  r1, r2            ; set cc on r1 + r2
         r1, r2            ; set cc on r1 and r2
         r1, r2            ; set cc on r1 xor r2
```

The mnemonics stand for 'compare' (CMP), 'compare negated' (CMN), '(bit) test' (TST) and 'test equal' (TEQ).

Immediate operands

If, instead of adding two registers, we simply wish to add a constant to a register we can replace the second source operand with an immediate value, which is a literal constant, preceded by '#':

```
ADD      r3,    r3,         ;   r3   :=   r3
#1                          +   1
AND      r8,    r7,         ;   r8   :=
#&ff                        r7[7:0]
```

The first example also illustrates that although the 3-address format allows source and destination operands to be specified separately, they are not required to be distinct registers. The second example shows that the immediate value may be specified in hexadecimal (base 16) notation by putting '&' after the '#'.

Since the immediate value is coded within the 32 bits of the instruction, it is not possible to enter every possible 32-bit value as an immediate. The values which can be entered correspond to any 32-bit binary number where all the binary ones fall within a group of eight adjacent bit positions on a 2-bit boundary. Most valid immediate values are given by:

$$immediate = (0 \rightarrow 255) \times 2^{2n}$$

Equation 10

where $0 < n < 12$. The assembler will also replace MOV with MVN, ADD with SUB, and so on, where this can bring the immediate within range.

This may appear a complex constraint on the immediate values, but it does, in practice, cover all the most common cases such as a byte value at any of the four byte positions within a 32-bit word, any power of 2, and so on. In any case the assembler will report any value which is requested that it cannot encode.

(The reason for the constraint on immediate values is the way they are specified at the binary instruction level. This is described in the Chapter 5, and the reader who wishes to understand this issue fully should look there for the complete explanation.)

Shifted register operands

A third way to specify a data operation is similar to the first, but allows the second register operand to be subject to a shift operation before it is combined with the first operand. For example:

```
ADD      r3,    r2,    r1,    LSL    #3 ;    r3    :=    r2    +    8
x   r1
```

Note that this is still a single ARM instruction, executed in a single clock cycle. Most processors offer shift operations as separate instructions, but the ARM combines them with a general ALU operation in a single instruction.

Here 'LSL' indicates 'logical shift left by the specified number of bits', which in this example is 3. Any number from 0 to 31 may be specified, though using 0 is equivalent to omitting the shift altogether. As before, '#' indicates an immediate quantity. The available shift operations are:

- LSL: logical shift left by 0 to 31 places; fill the vacated bits at the least significant end of the word with zeros.
- LSR: logical shift right by 0 to 32 places; fill the vacated bits at the most signifi cant end of the word with zeros.
- ASL: arithmetic shift left; this is a synonym for LSL.
- ASR: arithmetic shift right by 0 to 32 places; fill the vacated bits at the most sig nificant end of the word with zeros if the source operand was positive, or with ones if the source operand was negative.
- ROR: rotate right by 0 to 32 places; the bits which fall off the least significant end of the word are used, in order, to fill the vacated bits at the most significant end of the word.
- RRX: rotate right extended by 1 place; the vacated bit (bit 31) is filled with the old value of the C flag and the operand is shifted one place to the right. With appropriate use of the condition codes (see below) a 33-bit rotate of the operand and the C flag is performed.

These shift operations are illustrated in Figure 3.1 on page 54. It is also possible to use a register value to specify the number of bits the second operand should be shifted by:

```
ADD      r5,    r5,    r3,    LSL   r2 ;    r5    : =    r5    +    r3
x   2^r2
```

This is a 4-address instruction. Only the bottom eight bits of r2 are significant, but since shifts by more than 32 bits are not very useful this limitation is not important for most purposes.

Setting the condition codes

Any data processing instruction can set the condition codes (N, Z, C and V) if the programmer wishes it to. The comparison operations only set the condition codes, so there is no option with them, but for all other data processing instructions a

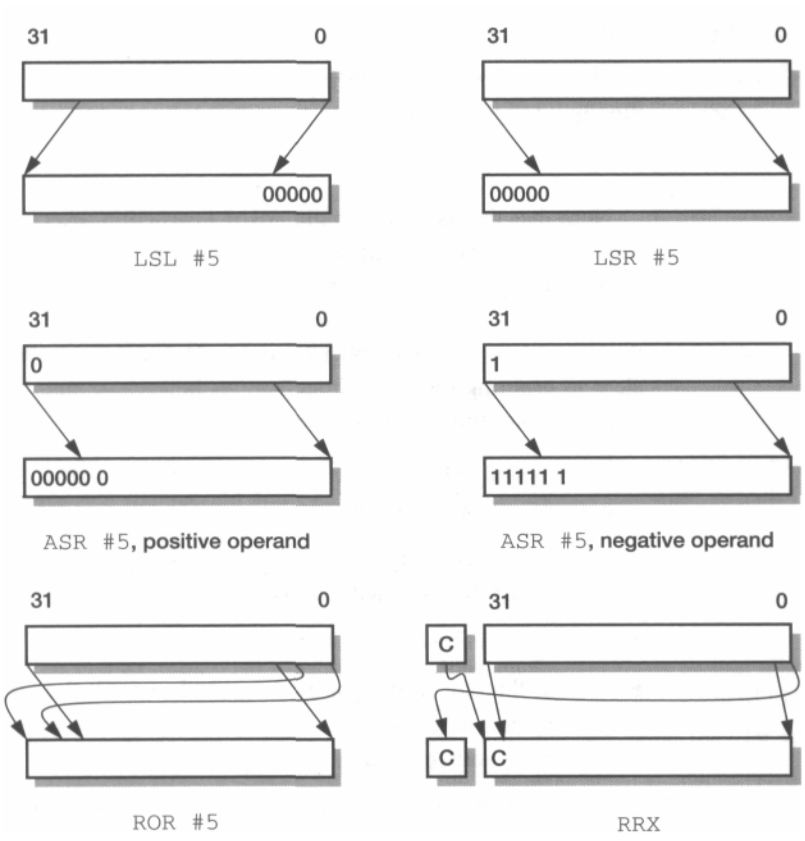


Figure 3.1 ARM shift operations

specific request must be made. At the assembly language level this request is indicated by adding an 's' to the opcode, standing for 'Set condition codes'. As an example, the following code performs a 64-bit addition of two numbers held in r0-r1 and r2-r3, using the C condition code flag to store the intermediate carry:

```
ADDS    r2, r2, r0 ; 32-bit carry out -> C..
ADC     r3, r3, r1 ; .. and added into high word
```

Since the s opcode extension gives the programmer control over whether or not an instruction modifies the condition codes, the codes can be preserved over long instruction sequences when it is appropriate to do so.

An arithmetic operation (which here includes CMP and CMN) sets all the flags according to the arithmetic result. A logical or move operation does not produce a meaningful value for C or V, so these operations set N and Z according to the result but preserve V, and either preserve C when there is no shift operation, or set C to the value of the last bit to fall off the end of the shift. This detail is not often significant.

Use Of the Condition codes

We have already seen the C flag used as an input to an arithmetic data processing instruction. However we have not yet seen the most important use of the condition codes, which is to control the program flow through the conditional branch instructions. These will be described in Section 3.3 on page 63.

Multiplies

A special form of the data processing instruction supports multiplication:

```
MUL     r4,    r3,    r2        ;  r4   :=   (r3   x  r2)[31:0]
```

There are some important differences from the other arithmetic instructions:

- Immediate second operands are not supported.
- The result register must not be the same as the first source register.
- If the ' s' bit is set the V flag is preserved (as for a logical instruction) and the C flag is rendered meaningless.

Multiplying two 32-bit integers gives a 64-bit result, the least significant 32 bits of which are placed in the result register and the rest are ignored. This can be viewed as multiplication in modulo 2^{32} arithmetic and gives the correct result whether the operands are viewed as signed or unsigned integers. (ARMs also support long multiply instructions which place the most significant 32 bits into a second result register; these are described in Section 5.8 on page 122.)

An alternative form, subject to the same restrictions, adds the product to a running total. This is the multiply-accumulate instruction:

```
MLA     r4,    r3,    r2,    r1        ;  r4   :=   (r3  x r2   +  r1)[31:0]
```

Multiplication by a constant can be implemented by loading the constant into a register and then using one of these instructions, but it is usually more efficient to use a short series of data processing instructions using shifts and adds or subtracts. For example, to multiply r0 by 35:

```
ADDRSB  r0, r0,   r0, r0,   r0, r0,   LSL       r0':=     5   x
                                      #2; LSL   r0 r0"   :=   7     (= 35 x r0)
                                      #3;       x   r0'
```

3.2 Data transfer instructions

Data transfer instructions move data between ARM registers and memory. There are three basic forms of data transfer instruction in the ARM instruction set:

- Single register load and store instructions.

  These instructions provide the most flexible way to transfer single data items between an ARM register and memory. The data item may be a byte, a 32-bit word, or a 16-bit half-word. (Older ARM chips may not support half-words.)

- Multiple register load and store instructions.

  These instructions are less flexible than single register transfer instructions, but enable large quantities of data to be transferred more efficiently. They are used for procedure entry and exit, to save and restore workspace registers, and to copy blocks of data around memory.

- Single register swap instructions.

  These instructions allow a value in a register to be exchanged with a value in memory, effectively doing both a load and a store operation in one instruction. They are little used in user-level programs, so they will not be discussed further in this section. Their principal use is to implement semaphores to ensure mutual exclusion on accesses to shared data structures in multi-processor systems, but don't worry if this explanation has little meaning for you at the moment.

It is quite possible to write any program for the ARM using only the single register load and store instructions, but there are situations where the multiple register transfers are much more efficient, so the programmer should be familiar with them.

Register-indirect addressing

Towards the end of Section 1.4 on page 14 there was a discussion of memory addressing mechanisms that are available to the processor instruction set designer. The ARM data transfer instructions are all based around register-indirect addressing, with modes that include base-plus-offset and base-plus-index addressing.

Register-indirect addressing uses a value in one register (the **base** register) as a memory address and either **loads** the value from that address into another register or **stores** the value from another register into that memory address.

These instructions are written in assembly language as follows:

```
LDR    r0, [r1]            ; r0 := mem32[r1] ;
STR    r0, [r1]            mem32[r1] := r0
```

Other forms of addressing all build on this form, adding immediate or register offsets to the base address. In all cases it is necessary to have an ARM register loaded with an address which is near to the desired transfer address, so we will begin by looking at ways of getting memory addresses into a register.

Initializing an address pointer

To load or store from or to a particular memory location, an ARM register must be initialized to contain the address of that location, or, in the case of single register transfer instructions, an address within 4 Kbytes of that location (the 4 Kbyte range will be explained later).

If the location is close to the code being executed it is often possible to exploit the fact that the program counter, r15, is close to the desired address. A data processing instruction can be employed to add a small offset to r15, but calculating the appropriate offset may not be that straightforward. However, this is the sort of tricky calculation that assemblers are good at, and ARM assemblers have an inbuilt 'pseudo instruction', ADR, which makes this easy. A pseudo instruction looks like a normal instruction in the assembly source code but does not correspond directly to a particular ARM instruction. Instead, the assembler has a set of rules which enable it to select the most appropriate ARM instruction or short instruction sequence for the situation in which the pseudo instruction is used. (In fact, ADR is always assembled into a single ADD or SUB instruction.)

As an example, consider a program which must copy data from TABLE1 to TABLE2, both of which are near to the code:

```
COPY    ADR     r1, TABLE1          ; r1 points to TABLE1
        ADR     r2, TABLE2          ; r2 points to TABLE2
        ..
TABLE1                              ; < source of data >
        ..
TABLE2                              ; < destination >
```

Here we have introduced **labels** (COPY, TABLE1 and TABLE2) which are simply names given to particular points in the assembly code. The first ADR pseudo instruction causes r1 to contain the address of the data that follows TABLE1; the second ADR likewise causes r2 to hold the address of the memory starting at TABLE2.

Of course any ARM instruction can be used to compute the address of a data item in memory, but for the purposes of small programs the ADR pseudo instruction will do what we require.

Single register load and store instructions

These instructions compute an address for the transfer using a base register, which should contain an address near to the target address, and an offset which may be another register or an immediate value.

We have just seen the simplest form of these instructions, which does not use an offset:

```
LDR     r0, [r1]            ; r0 := mem32 [r1] ;
STR     r0, [r1]            mem32[r1] := r0
```

The notation used here indicates that the data quantity is the 32-bit memory word addressed by r1. The word address in r1 should be aligned on a 4-byte boundary, so the two least significant bits of r1 should be zero. We can now copy the first word from one table to the other:

```
COPY    ADR     r1, TABLE1          ; r1 points to TABLE1
        ADR     r2, TABLE2          ; r2 points to TABLE2
        LDR     r0, [r1]            ; load first value...
        STR     r0, [r2]            ; and store it in TABLE2
        ..
TABLE1                              ; < source of data >
        ..
TABLE2                              ; < destination >
        ..
```

We could now use data processing instructions to modify both base registers ready for the next transfer:

```
COPY    ADR     r1, TABLE1          ; r1 points to TABLE1
        ADR     r2, TABLE2          ; r2 points to TABLE2
LOOP    LDR     r0, [r1]            ; get TABLE1 1st word
        STR     r0, [r2]            ; copy into TABLE2
        ADD     r1, r1, #4          ; step r1 on 1 word
        ADD     r2, r2, #4          ; step r2 on 1 word
        ???                         ; if more go back to LOOP
        ..
TABLE1                              ; < source of data >
        ..
TABLE2                              ; < destination >
```

Note that the base registers are incremented by 4 (bytes), since this is the size of a word. If the base register was word-aligned before the increment, it will be word-aligned afterwards too.

All load and store instructions could use just this simple form of register-indirect addressing. However, the ARM instruction set includes more addressing modes that can make the code more efficient.

Base plus offset addressing

If the base register does not contain exactly the right address, an offset of up to 4 Kbytes may be added (or subtracted) to the base to compute the transfer address:

```
LDR     r0,             ;  r0  :=  men32[r1
        [r1,#4]         +   4]
```

This is a **pre-indexed** addressing mode. It allows one base register to be used to access a number of memory locations which are in the same area of memory.

Sometimes it is useful to modify the base register to point to the transfer address. This can be achieved by using pre-indexed addressing with **auto-indexing,** and allows the program to walk through a table of values:

```
LDR     r0,             ;  r0  :=  mem32[r1
        [r1,#4]!        +   4] ;  r1   :=   r1
                        +   4
```

The exclamation mark indicates that the instruction should update the base register after initiating the data transfer. On the ARM this auto-indexing costs no extra time since it is performed on the processor's datapath while the data is being fetched from memory. It is exactly equivalent to preceding a simple register-indirect load with a data processing instruction that adds the offset (4 bytes in this example) to the base register, but the time and code space cost of the extra instruction are avoided.

Another useful form of the instruction, called **post-indexed** addressing, allows the base to be used without an offset as the transfer address, after which it is auto-indexed:

```
          LDR       r0, [r1], #4          r0 := mem32 [r1]
                                          r1 := r1 + 4
```

Here the exclamation mark is not needed, since the only use of the immediate offset is as a base register modifier. Again, this form of the instruction is exactly equivalent to a simple register-indirect load followed by a data processing instruction, but it is faster and occupies less code space.

Using the last of these forms we can now improve on the table copying program example introduced earlier:

```
COPY    ADR     r1, TABLE1          ; r1 points to TABLE1
        ADR     r2, TABLE2          ; r2 points to TABLE2
LOOP    LDR     r0, [r1], #4        ; get TABLE1 1st word
        STR     r0, [r2], #4        ; copy into TABLE2
        ???                         ; if more go back to LOOP
        ..
TABLE1                              ; < source of data >
        ..
TABLE2                              ; < destination >
```

The load and store instructions are repeated until the required number of values has been copied into TABLE2, then the loop is exited. Control flow instructions are required to determine the loop exit; they will be introduced shortly.

In the above examples the address offset from the base register was always an immediate value. It can equally be another register, optionally subject to a shift operation before being added to the base, but such forms of the instruction are less useful than the immediate offset form. They are described fully in Section 5.10 on page 125.

As a final variation, the size of the data item which is transferred may be a single unsigned 8-bit byte instead of a 32-bit word. This option is selected by adding a letter B onto the opcode:

```
          LDRB      r0, [r1]              ; r0 := mem8[r1]
```

In this case the transfer address can have any alignment and is not restricted to a 4-byte boundary, since bytes may be stored at any byte address. The loaded byte is placed in the bottom byte of r0 and the remaining bytes in r0 are filled with zeros.

(All but the oldest ARM processors also support **signed** bytes, where the top bit of the byte indicates whether the value should be treated as positive or negative, and signed and unsigned 16-bit half-words; these variants will be described when we return to look at the instruction set in more detail in Section 5.11 on page 128.)

Multiple register data transfers

Where considerable quantities of data are to be transferred, it is preferable to move several registers at a time. These instructions allow any subset (or all) of the 16 registers to be transferred with a single instruction. The trade-off is that the available addressing modes are more restricted than with a single register transfer instruction. A simple example of this instruction class is:

```
LDMIA   r1, {r0,r2,r5}      ; r0 := mem32[r1]
                            ; r2 := mem32[r1 + 4]
                            ; r5 := mem32[r1 + 8]
```

Since the transferred data items are always 32-bit words, the base address (r1) should be word-aligned.

The transfer list, within the curly brackets, may contain any or all of r0 to r15. The order of the registers within the list is insignificant and does not affect the order of transfer or the values in the registers after the instruction has executed. It is normal practice, however, to specify the registers in increasing order within the list.

Note that including r15 in the list will cause a change in the control flow, since r15 is the PC. We will return to this case when we discuss control flow instructions and will not consider it further until then.

The above example illustrates a common feature of all forms of these instructions: the lowest register is transferred to or from the lowest address, and then the other registers are transferred in order of register number to or from consecutive word addresses above the first. However there are several variations on how the first address is formed, and auto-indexing is also available (again by adding a '!' after the base register).

Stack addressing

The addressing variations stem from the fact that one use of these instructions is to implement stacks within memory. A stack is a form of last-in-first-out store which supports simple dynamic memory allocation, that is, memory allocation where the address to be used to store a data value is not known at the time the program is compiled or assembled. An example would be a recursive function, where the depth of recursion depends on the value of the argument. A stack is usually implemented as a linear data structure which grows up (an **ascending** stack) or down (a **descending** stack) memory as data is added to it and shrinks back as data is removed. A **stack pointer** holds the address of the current top of the stack, either by pointing to the last valid data item pushed onto the stack (a **full** stack), or by pointing to the vacant slot where the next data item will be placed (an **empty** stack).

The above description suggests that there are four variations on a stack, representing all the combinations of ascending and descending full and empty stacks. The ARM multiple register transfer instructions support all four forms of stack:

- Full ascending: the stack grows up through increasing memory addresses and the base register points to the highest address containing a valid item.

- Empty ascending: the stack grows up through increasing memory addresses and the base register points to the first empty location above the stack.
- Full descending: the stack grows down through decreasing memory addresses and the base register points to the lowest address containing a valid item.
- Empty descending: the stack grows down through decreasing memory addresses and the base register points to the first empty location below the stack.

Block copy addressing

Although the stack view of multiple register transfer instructions is useful, there are occasions when a different view is easier to understand. For example, when these instructions are used to copy a block of data from one place in memory to another a mechanistic view of the addressing process is more useful. Therefore the ARM assembler supports two different views of the addressing mechanism, both of which map onto the same basic instructions, and which can be used interchangeably. The block copy view is based on whether the data is to be stored above or below the address held in the base register and whether the address incrementing or decrementing begins before or after storing the first value. The mapping between the two views depends on whether the operation is a load or a store, and is detailed in Table 3.1 on page 62.

The block copy views are illustrated in Figure 3.2 on page 62, which shows how each variant stores three registers into memory and how the base register is modified if auto-indexing is enabled. The base register value before the instruction is r9, and after the auto-indexing it is r9'.

To illustrate the use of these instructions, here are two instructions which copy eight words from the location r0 points to to the location r1 points to:

```
LDMIA  r0!, {r2-r9}
STMIA  r1,  {r2-r9}
```

After executing these instructions r0 has increased by 32 since the '!' causes it to auto-index across eight words, whereas r1 is unchanged. If r2 to r9 contained useful values, we could preserve them across this operation by pushing them onto a stack:

```
STMFD   r13!,   {r2-r9}        save regs onto stack
LDMIA   r0!,    {r2-r9}
STMIA   r1,     {r2-r9}
LDMFD   r13!,   {r2-r9}       ;   restore    from
                              stack
```

Here the 'FD' postfix on the first and last instructions signifies the full descending stack address mode as described earlier. Note that auto-indexing is almost always specified for stack operations in order to ensure that the stack pointer has a consistent behaviour.

The load and store multiple register instructions are an efficient way to save and restore processor state and to move blocks of data around in memory. They save code space and operate up to four times faster than the equivalent sequence of single

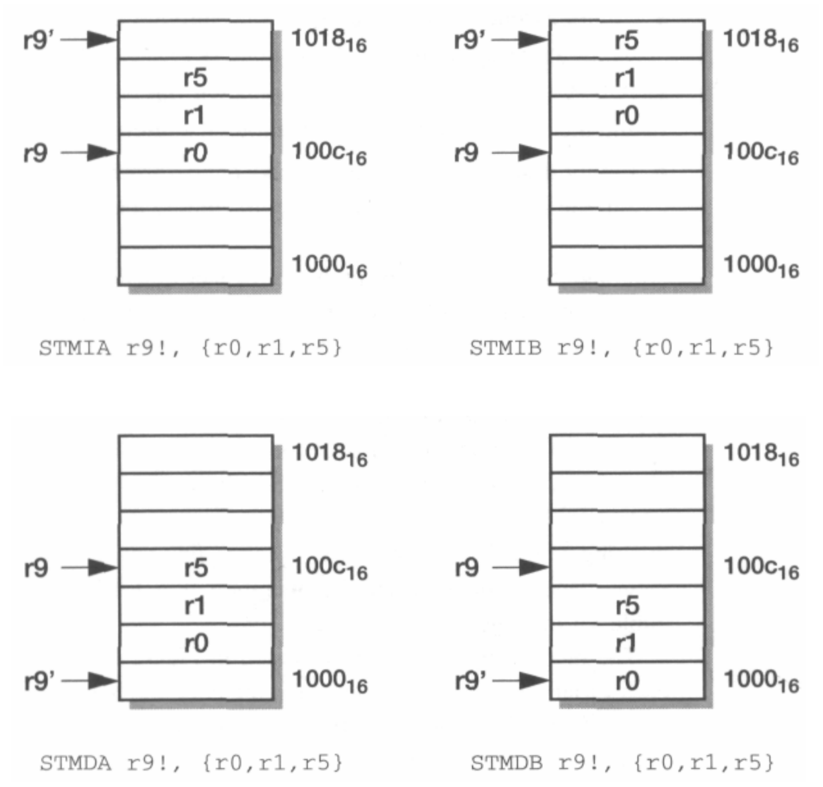


Figure 3.2 Multiple register transfer addressing modes.

Table 3.1 The mapping between the stack and block copy views of the load and store multiple instructions.

		Ascending		Descending	
		Full	**Empty**	**Full**	**Empty**
Increment	**Before**	STMIB STMFA			LDMIB LDMED
	After		STMIA STMEA	LDMIA LDMFD	
Decrement	**Before**		LDMDB LDMEA	STMDB STMFD	
	After	LDMDA LDMFA			STMDA STMED

register load or store instructions (a factor of two due to improved sequential behaviour and another factor of nearly two due to the reduced instruction count). This significant advantage suggests that it is worth thinking carefully about how data is organized in memory in order to maximize the potential for using multiple register data transfer instructions to access it.

These instructions are, perhaps, not pure 'RISC' since they cannot be executed in a single clock cycle even with separate instruction and data caches, but other RISC architectures are beginning to adopt multiple register transfer instructions in order to increase the data bandwidth between the processor's registers and the memory.

On the other side of the equation, load and store multiple instructions are complex to implement, as we shall see later.

The ARM multiple register transfer instructions are uniquely flexible in being able to transfer any subset of the 16 currently visible registers, and this feature is powerfully exploited by the ARM procedure call mechanism which is described in Section 6.8 on page 175.

3.3 Control flow instructions

This third category of instructions neither processes data nor moves it around; it simply determines which instructions get executed next.

Branch instructions

The most common way to switch program execution from one place to another is to use the branch instruction:

```
        B       LABEL
        ..
        ..
LABEL
```

The processor normally executes instructions sequentially, but when it reaches the branch instruction it proceeds directly to the instruction at LABEL instead of executing the instruction immediately after the branch. In this example LABEL comes after the branch instruction in the program, so the instructions in between are skipped. However, LABEL could equally well come before the branch, in which case the processor goes back to it and possibly repeats some instructions it has already executed.

Conditional branches

Sometimes you will want the processor to take a decision whether or not to branch. For example, to implement a loop a branch back to the start of the loop is required, but this branch should only be taken until the loop has been executed the required number of times, then the branch should be skipped.

The mechanism used to control loop exit is conditional branching. Here the branch has a condition associated with it and it is only executed if the condition codes have the correct value. A typical loop control sequence might be:

```
        MOV     r0, #0          ; initialize counter
LOOP
        ADD     r0, r0, #1      ; increment loop counter
        CMP BNE r0, #10         ; compare with limit
                LOOP            ; repeat if not equal
                                ; else fall through
```

This example shows one sort of conditional branch, BNE, or 'branch if not equal'. There are many forms of the condition. All the forms are listed in Table 3.2, along with their normal interpretations. The pairs of conditions which are listed in the same row of the table (for instance BCC and BLO) are synonyms which result in identical binary code, but both are available because each makes the interpretation of the assembly source code easier in particular circumstances. Where the table refers to signed or unsigned comparisons this does not reflect a choice in the comparison instruction itself but supports alternative interpretations of the operands.

Table 3.2 Branch conditions.

Branch	Interpretation	Normal uses
B BAL	Unconditional Always	Always take this branch Always take this branch
BEQ	Equal	Comparison equal or zero result
BNE	Not equal	Comparison not equal or non-zero result
BPL	Plus	Result positive or zero
BMI	Minus	Result minus or negative
BCC BLO	Carry clear Lower	Arithmetic operation did not give carry-out Unsigned comparison gave lower
BCS BHS	Carry set Higher or same	Arithmetic operation gave carry-out Unsigned comparison gave higher or same
BVC	Overflow clear	Signed integer operation; no overflow occurred
BVS	Overflow set	Signed integer operation; overflow occurred
BGT	Greater than	Signed integer comparison gave greater than
BGE	Greater or equal	Signed integer comparison gave greater or equal
BLT	Less than	Signed integer comparison gave less than
BLE	Less or equal	Signed integer comparison gave less than or equal
BHI	Higher	Unsigned comparison gave higher
BLS	Lower or same	Unsigned comparison gave lower or same

Conditional execution

An unusual feature of the ARM instruction set is that conditional execution applies not only to branches but to all ARM instructions. A branch which is used to skip a small number of following instructions may be omitted altogether by giving those instructions the opposite condition. For example, consider the following sequence:

```
        CMP     r0, #5
        BEQ     BYPASS              ; if (r0 != 5) {
        ADD     r1, r1, r0          ;    r1 := r1 + r0 - r2
        SUB     r1, r1, r2          ; }
BYPASS  ..
```

This may be replaced by:

```
        CMP     r0, #5              ; if (r0 != 5) {
        ADDNE   r1, r1, r0          ;    r1 := r1 + r0 - r2
        SUBNE   r1, r1, r2          ; }
```

The new sequence is both smaller and faster than the old one. Whenever the conditional sequence is three instructions or fewer it is better to exploit conditional execution than to use a branch, provided that the skipped sequence is not doing anything complicated with the condition codes within itself.

(The three instruction guideline is based on the fact that ARM branch instructions typically take three cycles to execute, and it is only a guideline. If the code is to be fully optimized then the decision on whether to use conditional execution or a branch must be based on measurements of the dynamic code behaviour.)

Conditional execution is invoked by adding the 2-letter condition after the 3-letter opcode (and before any other instruction modifier letter such as the 's' that controls setting the condition codes in a data processing instruction or the 'B' that specifies a byte load or store).

Just to emphasize the scope of this technique, note that every ARM instruction, including supervisor calls and coprocessor instructions, may have a condition appended which causes it to be skipped if the condition is not met.

It is sometimes possible to write very compact code by cunning use of conditionals, for example:

```
; if ((a==b) && (c==d)) e++;

CMP r0, r1 CMPEQ r2,
r3 ADDEQ r4, r4, #1
```

Note how if the first comparison finds unequal operands the second is skipped, causing the increment to be skipped also. The logical 'and' in the if clause is implemented by making the second comparison conditional.

Branch and link instructions

A common requirement in a program is to be able to branch to a subroutine in a way which makes it possible to resume the original code sequence when the subroutine has completed. This requires that a record is kept of the value of the program counter just before the branch is taken.

ARM offers this functionality through the branch and link instruction which, as well as performing a branch in exactly the same way as the branch instruction, also saves the address of the instruction following the branch in the link register, r14:

```
        BL      SUBR            ; branch to SUBR
        ..                      ; return to here
SUBR    ..                      ; subroutine entry point
        MOV     pc, r14         ; return
```

Note that since the return address is held in a register, the subroutine should not call a further, nested, subroutine without first saving r14, otherwise the new return address will overwrite the old one and it will not be possible to find the way back to the original caller. The normal mechanism used here is to push r14 onto a stack in memory. Since the subroutine will often also require some work registers, the old values in these registers can be saved at the same time using a store multiple instruction:

```
        BL      SUB1

SUB1   STMFD   r13!, {r0-r2,r14} BL    save work & link regs
        SUB2


        ..
SUB2
        ..
```

A subroutine that does not call another subroutine (a **leaf** subroutine) need not save r14 since it will not be overwritten.

Subroutine return instructions

To get back to the calling routine, the value saved by the branch and link instruction in r14 must be copied back into the program counter. In the simplest case of a leaf subroutine (a subroutine that does not call another subroutine) a MOV instruction suffices, exploiting the visibility of the program counter as r15:

```
SUB2
        MOV     pc, r14 ; copy r14 into r15 to return
```

In fact the availability of the program counter as r15 means that any of the data processing instructions can be used to compute a return address, though the 'MOV' form is by far the most commonly used.

Where the return address has been pushed onto a stack, it can be restored along with any saved work registers using a load multiple instruction:

```
SUB1    STMFD  r13!, {r0-r2,r14}; save work regs & link BL
        SUB2


        ..
        LDMFD   r13!, {r0-r2,pc} ; restore work regs & return
```

Note here how the return address is restored directly to the program counter, not to the link register. This single restore and return instruction is very powerful. Note also the use of the stack view of the multiple register transfer addressing modes. The same stack model (in this case 'full descending', which is the most common stack type for ARM code) is used for both the store and the load, ensuring that the correct values will be collected. It is important that for any particular stack the same addressing mode is used for every use of the stack, unless you really know what you are doing.

Supervisor calls

Whenever a program requires input or output, for instance to send some text to the display, it is normal to call a supervisor routine. The supervisor is a program which operates at a privileged level, which means that it can do things that a user-level program cannot do directly. The limitations on the capabilities of a user-level program vary from system to system, but in many systems the user cannot access hardware facilities directly.

The supervisor provides trusted ways to access system resources which appear to the user-level program rather like special subroutine accesses. The instruction set includes a special instruction, SWI, to call these functions, (SWI stands for 'Software Interrupt', but is usually pronounced 'Supervisor Call'.)

Although the supervisor calls are implemented in system software, and could therefore be totally different from one ARM system to another, most ARM systems implement a common subset of calls in addition to any specific calls required by the particular application. The most useful of these is a routine which sends the character in the bottom byte of r0 to the user display device:

```
        SWI     SWI_WriteC          ;   output   r0[7:0]
```

Another useful call returns control from a user program back to the monitor program:

```
        SWI     SWI_Exit            ; return to monitor
```

The operation of SWIs is described in more detail in Section 5.6 on page 117.

Jump tables

Jump tables are not normally used by less experienced programmers, so you can ignore this section if you are relatively new to programming at the assembly level.

The idea of a jump table is that a programmer sometimes wants to call one of a set of subroutines, the choice depending on a value computed by the program. It is clearly possible to do this with the instructions we have seen already. Suppose the value is in r0. We can then write:

```
          BL      JUMPTAB

JUMPTAB   CMP     r0, #0
          BEQ     SUB0 r0,
          CMP     #1 SUB1
          BEQ     r0, #2
          CMP     SUB2
          BEQ
```

However, this solution becomes very slow when the list of subroutines is long unless there is some reason to think that the later choices will rarely be used. A solution which is more efficient in this case exploits the visibility of the program counter in the general register file:

```
         BL      JUMPTAB

         ..

JUMPTAB ADR      r1, SUBTAB             r1 -> SUBTAB
         CMP     r0, #SUBMAX            check for overrun.. . . if OK,
         LDRLS   pc, [r1,r0,LSL #2]      table jump .. otherwise
         B       ERROR                   signal error
SUBTAB  DCD      SUB0                   table of subroutine
         DCD     SUB1                         entry points
         DCD     SUB2
```

The 'DCD' directive instructs the assembler to reserve a word of store and to initialize it to the value of the expression to the right, which in these cases is just the address of the label.

This approach has a constant performance however many subroutines are in the table and independent of the distribution of frequency of use. Note, however, that the consequences of reading beyond the end of the table are likely to be dire, so checking for overrun is essential! Here, note how the overrun check is implemented by making the load into the PC conditional, so the overrun case skips the load and falls into the branch to the error handler. The only performance cost of checking for overrun is the comparison with the maximum value. More obvious code might have been:

```
         CMP     r0,   #SUBMAX          ;    check   for overrun..
         BHI     ERROR                  ;       ..    if overrun call
         error
         LDR     pc,      [r1,r0,LSL    #2]     ;       ..    else    table
         jump
```

but note that here the cost of conditionally skipping the branch is borne every time the jump table is used. The original version is more efficient except when overrun is detected, which should be infrequent and, since it represents an error, performance in that case is not of great concern.

An alternative, less obvious way to implement a jump table is discussed in 'switches'on page 171.

3.4 Writing simple assembly language programs

We now have all the basic tools for writing simply assembly language programs. As with any programming task, it is important to have a clear idea of your algorithm before beginning to type instructions into the computer. Large programs are almost certainly better written in C or C++, so we will only look at small examples of assembly language programs.

Even the most experienced programmers begin by checking that they can get a very simple program to run before moving on to whatever their real task is. There are so many complexities to do with learning to use a text editor, working out how to get the assembler to run, how to load the program into the machine, how to get it to start executing and so on. This sort of simple test program is often referred to as a *Hello World* program because all it does is print 'Hello World' on the display before terminating. Here is an ARM assembly language version:

```
        AREA    HelloW,CODE,READONLY ; declare code area
SWI_WriteC      EQU     &0          ; output character in r0
SWI_Exit        EQU     &11         ; finish program
        ENTRY                       ; code entry point
START   ADR     r1, TEXT            ; r1 -> "Hello World"
LOOP    LDRB    r0, [r1], #1        ; get the next byte
        CMP     r0, #0              ; check for text end
        SWINE   SWI_WriteC          ; if not end print ..
        BNE     LOOP                ;  .. and loop back
        SWI     SWI_Exit            ; end of execution
TEXT    =       "Hello World",&0a,&0d,0
        END                         ; end of program source
```

This program illustrates a number of the features of the ARM assembly language and instruction set:

- The declaration of the code 'AREA', with appropriate attributes.
- The definitions of the system calls which will be used in the routine. (In a larger program these would be defined in a file which other code files would reference.)
- The use of the ADR pseudo instruction to get an address into a base register.
- The use of auto-indexed addressing to move through a list of bytes.
- Conditional execution of the SWI instruction to avoid an extra branch.

Note also the use of a zero byte to mark the end of the string (following the line-feed and carriage return special characters). Whenever you use a looping structure, make sure it has a terminating condition.

In order to run this program you will need the following tools, all of which are available within the ARM software development toolkit:

- A text editor to type the program into.
- An assembler to turn the program into ARM binary code.
- An ARM system or emulator to execute the binary on. The ARM system must have some text output capability. (The ARM development board, for example, sends text output back up to the host for output onto the host's display.)

Once you have this program running you are ready to try something more useful. From now on, the only thing that changes is the program text. The use of the editor, the assembler, and the test system or emulator will remain pretty similar to what you have done already, at least up to the point where your program refuses to do what you want and you can't see why it refuses. Then you will need to use a debugger to see what is happening inside your program. This means learning how to use another complex tool, so we will put off that moment for as long as possible.

For the next example, we can now complete the block copy program developed partially earlier in the text. To ensure that we know it has worked properly, we will use a text source string so that we can output it from the destination address, and we will initialize the destination area to something different:

```
        AREA    BlkCpy,CODE,READONLY
SWI_WriteC      EQU     &0          ; output character in r0
SWI_Exit        EQU     &11         ; finish program
        ENTRY                       ; code entry point
        ADR     r1, TABLE1          ; r1 -> TABLE1
        ADR     r2, TABLE2          ; r2 -> TABLE2
        ADR     r3, T1END           ; r3 -> T1END
LOOP1   LDR     r0, [r1], #4        ; get TABLE1 1st word
        STR     r0, [r2], #4        ; copy into TABLE2
        CMP     r1, r3              ; finished?
        BLT     LOOP1               ; if not, do more
        ADR     r1, TABLE2          ; r1 -> TABLE2
LOOP2   LDRB    r0, [r1], #1        ; get next byte
        CMP     r0, #0              ; check for text end
        SWINE   SWI_WriteC          ; if not end, print ..
        BNE     LOOP2               ; .. and loop back
        SWI     SWI_Exit            ; finish
TABLE1  =       "This is the right string!", &0a, &0d, 0
T1END
        ALIGN                       ; ensure word alignment
TABLE2  =       "This is the wrong string!", 0
        END
```

This program uses word loads and stores to copy the table, which is why the tables must be word-aligned. It then uses byte loads to print out the result using a routine which is the same as that used in the 'Hello World' program.

Note the use of 'BLT' to control the loop termination. If TABLE1 contains a number of bytes which is not a multiple of four, there is a danger that r1 would step past T1END without ever exactly equalling it, so a termination condition based on 'BNE' might fail.

If you have succeeded in getting this program running, you are well on the way to understanding the basic operation of the ARM instruction set. The examples and exercises which follow should be studied to reinforce this understanding. As you attempt more complex programming tasks, questions of detail will arise. These should be answered by the full instruction set description given in Chapter 5.

Program design

With a basic understanding of the instruction set, small programs can be written and debugged without too much trouble by just typing them into an editor and seeing if they work. However, it is dangerous to assume that this simple approach will scale to the successful development of complex programs which may be expected to work for many years, which may be changed by other programmers in the future, and which may end up in the hands of customers who will use use them in unexpected ways.

This book is not a text on program design, but having offered an introduction to programming it would be a serious omission not to point out that there is a lot more to writing a useful program than just sitting down and typing code.

Serious programming should start not with coding, but with careful design. The first step of the development process is to understand the requirements; it is surprising how often programs do not behave as expected because the requirements were not well understood by the programmer! Then the (often informal) requirements should be translated into an unambiguous specification. Now the design can begin, defining a program structure, the data structures that the program works with and the algorithms that are used to perform the required operations on the data. The algorithms may be expressed in **pseudo-code,** a program-like notation which does not follow the syntax of a particular programming language but which makes the meaning clear.

Only when the design is developed should the coding begin. Individual modules should be coded, tested thoroughly (which may require special programs to be designed as 'test-harnesses') and documented, and the program built piece by piece.

Today nearly all programming is based on high-level languages, so it is rare for large programs to be built using assembly programming as described here. Sometimes, however, it may be necessary to develop small software components in assembly language to get the best performance for a critical application, so it is useful to know how to write assembly code for these purposes.

3.5 Examples and exercises

Once you have the basic flavour of an instruction set the easiest way to learn to write programs is to look at some examples, then attempt to write your own program to do something slightly different. To see whether or not your program works you will need an ARM assembler and either an ARM emulator or hardware with an ARM processor in it. The following sections contain example ARM programs and suggestions for modifications to them. You should get the original program working first, then see if you can edit it to perform the modified function suggested in the exercises.

Example 3.1 **Print out r1 in hexadecimal.**

This is a useful little routine which dumps a register to the display in hexadecimal (base 16) notation; it can be used to help debug a program by writing out register values and checking that algorithms are producing the expected results, though in most cases using a debugger is a better way of seeing what is going on inside a program.

```
        AREA    Hex_Out,CODE,READONLY
SWI_WriteC      EQU    &0           output character in r0
SWI_Exit        EQU    &11          finish program
        ENTRY                       code entry point
        LDR     r1, VALUE           get value to print
        BL      HexOut              call hexadecimal output
        SWI     SWI_Exit            finish
VALUE  DCD      &12345678           test value
HexOut MOV      r2, #8              nibble count = 8
LOOP   MOV      r0, r1, LSR #28     get top nibble
        CMP     r0, #9              0-9 or A-F?
        ADDGT   r0, r0, #"A"-10     ASCII alphabetic
        ADDLE   r0, r0, #"0"        ASCII numeric
        SWI     SWI_WriteC          print character
        MOV     r1, r1, LSL #4      shift left one nibble
        SUBS    r2, r2, #1          decrement nibble count
        BNE     LOOP                if more do next nibble
        MOV     pc, r14             return
        END
```

Exercise 3.1.1 Modify the above program to output r1 in binary format. For the value loaded into r1 in the example program you should get:

00010010001101000101011001111000 Use HEXOUT as the basis of

Exercise 3.1.2 a program to display the contents of an area of memory.

Example 3.2 **Write a subroutine to output a text string immediately following the call.**

It is often useful to be able to output a text string without having to set up a separate data area for the text (though this is inefficient if the processor has separate data and instruction caches, as does the StrongARM; in this case it is better to set up a separate data area). A call should look like:

```
BL      TextOut
=       "Test string",&0a,&0d,0
ALIGN
..                          ; return to here
```

The issue here is that the return from the subroutine must not go directly to the value put in the link register by the call, since this would land the program in the text string. Here is a suitable subroutine and test harness:

```
        AREA    Text_Out,CODE,READONLY
SWI_WriteC      EQU     &0          ; output character in r0
SWI_Exit        EQU     &11         ; finish program
        ENTRY                       ; code entry point
        BL      TextOut             ; print following string
        =       "Test string",&0a,&0d,0
        ALIGN
        SWI     SWI_Exit            ; finish
TextOut LDRB    r0, [r14], #1       ; get next character
        CMP     r0, #0              ; test for end mark
        SWINE   SWI_WriteC          ; if not end, print..
        BNE     TextOut             ;  .. and loop
        ADD     r14, r14, #3        ; pass next word boundary
        BIC     r14, r14, #3        ; round back to boundary
        MOV     pc, r14             ; return
        END
```

This example shows r14 incrementing along the text string and then being adjusted to the next word boundary prior to the return. If the adjustment (add 3, then clear the bottom two bits) looks like slight of hand, check it; there are only four cases.

Exercise 3.2.1 Using code from this and the previous examples, write a program to dump the ARM registers in hexadecimal with formatting such as:

```
r0 = 12345678 r1
= 9ABCDEF0
```

Exercise 3.2.2 Now try to save the registers you need to work with before they are changed, for instance by saving them near the code using PC-relative addressing.

4 ARM Organization and Implementation

Summary of chapter contents

The organization of the ARM integer processor core changed very little from the first 3 micron devices developed at Acorn Computers between 1983 and 1985 to the ARM6 and ARM7 developed by ARM Limited between 1990 and 1995. The 3-stage pipeline used by these processors was steadily tightened up, and CMOS process technology reduced in feature size by almost an order of magnitude over this period, so the performance of the cores improved dramatically, but the basic principles of operation remained largely the same.

Since 1995 several new ARM cores have been introduced which deliver significantly higher performance through the use of 5-stage pipelines and separate instruction and data memories (usually in the form of separate caches which are connected to a shared instruction and data main memory system).

This chapter includes descriptions of the internal structures of these two basic styles of processor core and covers the general principles of operation of the 3-stage and 5-stage pipelines and *a* number of implementation details. Details on particular cores are presented in Chapter 9.

4.1 3-stage pipeline ARM organization

The organization of an ARM with a 3-stage pipeline is illustrated in Figure 4.1 on page 76. The principal components are:

- The register bank, which stores the processor state. It has two read ports and one write port which can each be used to access any register, plus an additional read port and an additional write port that give special access to r15, the program counter. (The additional write port on r15 allows it to be updated as the instruc tion fetch address is incremented and the read port allows instruction fetch to resume after a data address has been issued.)
- The barrel shifter, which can shift or rotate one operand by any number of bits.
- The ALU, which performs the arithmetic and logic functions required by the instruction set.
- The address register and incrementer, which select and hold all memory addresses and generate sequential addresses when required.
- The data registers, which hold data passing to and from memory.
- The instruction decoder and associated control logic.

In a single-cycle data processing instruction, two register operands are accessed, the value on the B bus is shifted and combined with the value on the A bus in the ALU, then the result is written back into the register bank. The program counter value is in the address register, from where it is fed into the incrementer, then the incremented value is copied back into rl5 in the register bank and also into the address register to be used as the address for the next instruction fetch.

The 3-stage pipeline

ARM processors up to the ARM? employ a simple 3-stage pipeline with the following pipeline stages:

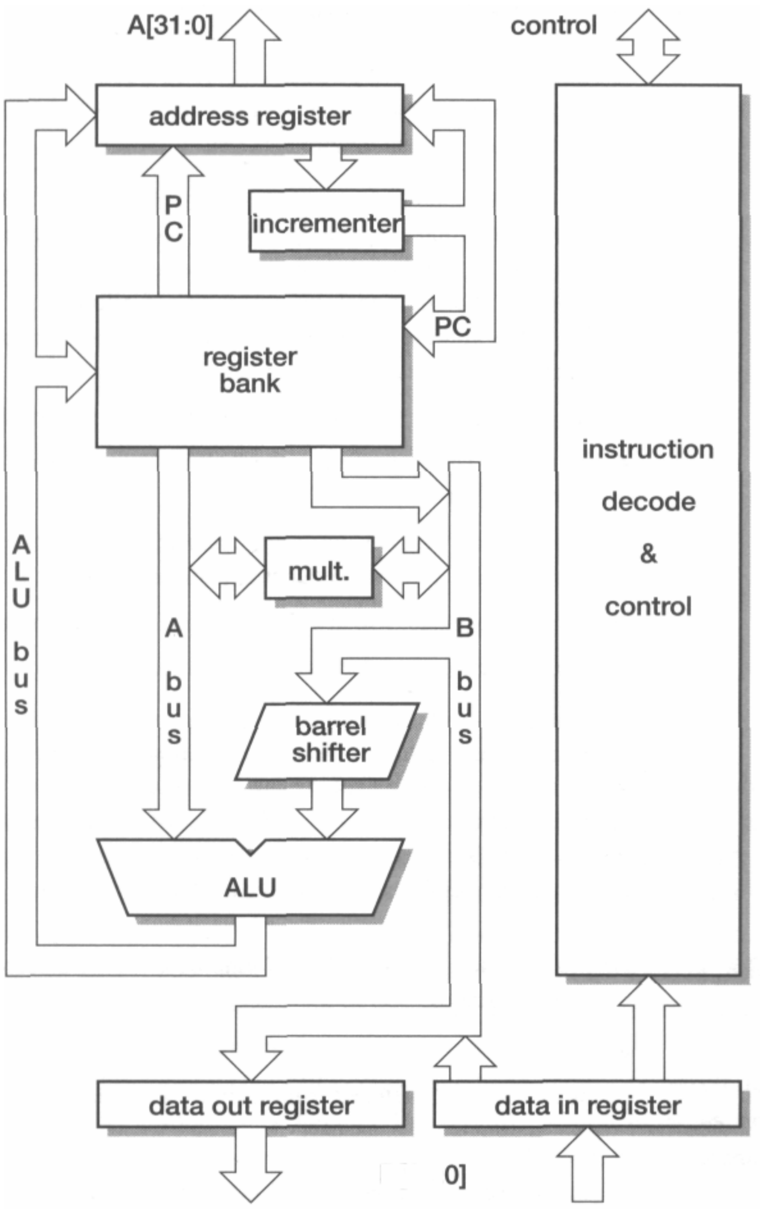
- Fetch;

  the instruction is fetched from memory and placed in the instruction pipeline.
- Decode;

  the instruction is decoded and the datapath control signals prepared for the next cycle. In this stage the instruction 'owns' the decode logic but not the datapath.
- Execute;

  the instruction 'owns' the datapath; the register bank is read, an operand shifted, the ALU result generated and written back into a destination register.

At any one time, three different instructions may occupy each of these stages, so the hardware in each stage has to be capable of independent operation.



D[31: Figure

4.1 3-stage pipeline ARM organization.

When the processor is executing simple data processing instructions the pipeline enables one instruction to be completed every clock cycle. An individual instruction takes three clock cycles to complete, so it has a three-cycle latency, but the throughput is one instruction per cycle. The 3-stage pipeline operation for single-cycle instructions is shown in Figure 4.2 on page 77.

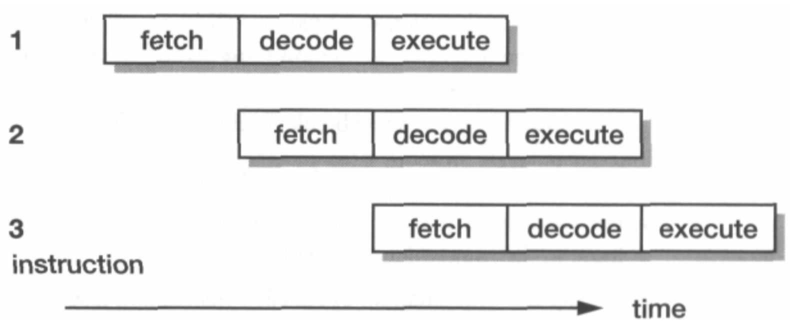


Figure 4.2 ARM single-cycle instruction 3-stage pipeline operation.

When a multi-cycle instruction is executed the flow is less regular, as illustrated in Figure 4.3. This shows a sequence of single-cycle ADD instructions with a data store instruction, STR, occurring after the first ADD. The cycles that access main memory are shown with light shading so it can be seen that memory is used in every cycle. The datapath is likewise used in every cycle, being involved in all the execute cycles, the address calculation and the data transfer. The decode logic is always generating the control signals for the datapath to use in the next cycle, so in addition to the explicit decode cycles it is also generating the control for the data transfer during the address calculation cycle of the STR.

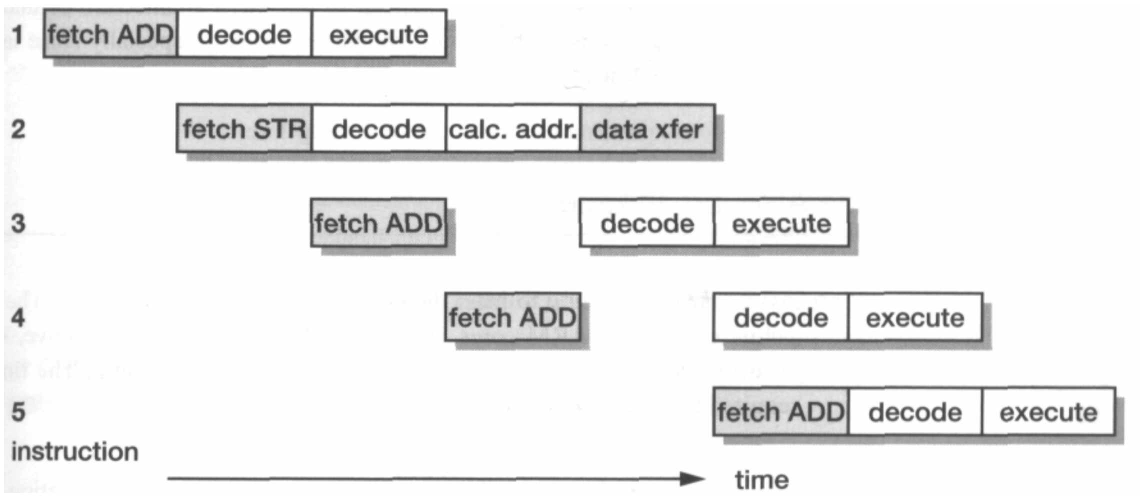


Figure 4.3 ARM multi-cycle instruction 3-stage pipeline operation.

Thus, in this instruction sequence, all parts of the processor are active in every cycle and the memory is the limiting factor, denning the number of cycles the sequence must take.

The simplest way to view breaks in the ARM pipeline is to observe that:

- All instructions occupy the datapath for one or more adjacent cycles.
- For each cycle that an instruction occupies the datapath, it occupies the decode logic in the immediately preceding cycle.
- During the first datapath cycle each instruction issues a fetch for the next instruc tion but one.
- Branch instructions flush and refill the instruction pipeline.

PC behaviour

One consequence of the pipelined execution model used on the ARM is that the program counter, which is visible to the user as r!5, must run ahead of the current instruction. If, as noted above, instructions fetch the next instruction but one during their first cycle, this suggests that the PC must point eight bytes (two instructions) ahead of the current instruction.

This is, indeed, what happens, and the programmer who attempts to access the PC directly through r!5 must take account of the exposure of the pipeline here. However, for most normal purposes the assembler or compiler handles all the details.

Even more complex behaviour is exposed if r!5 is used later than the first cycle of an instruction, since the instruction will itself have incremented the PC during its first cycle. Such use of the PC is not often beneficial so the ARM architecture definition specifies the result as 'unpredictable' and it should be avoided, especially since later ARMs do not have the same behaviour in these cases.

4.2 5-stage pipeline ARM organization

All processors have to develop to meet the demand for higher performance. The 3-stage pipeline used in the ARM cores up to the ARM? is very cost-effective, but higher performance requires the processor organization to be rethought. The time, *T* , required to execute a given program is given by:

$$T_{prog} = \frac{N_{inst} \times CPI}{f_{clk}}, \qquad \text{Equation 11}$$

where N_{inst} is the number of ARM instructions executed in the course of the program, *CPI* is the average number of clock cycles per instruction and f_{clk} is the processor's clock frequency. Since N_{inst} is constant for a given program (compiled with a given compiler using a given set of optimizations, and so on) there are only two ways to increase performance:

- Increase the clock rate, f_{clk}.

  This requires the logic in each pipeline stage to be simplified and, therefore, the number of pipeline stages to be increased.

- Reduce the average number of clock cycles per instruction, *CPI.*

  This requires either that instructions which occupy more than one pipeline slot in a 3-stage pipeline ARM are re-implemented to occupy fewer slots, or that pipeline stalls caused by dependencies between instructions are reduced, or a combination of both.

Memory bottleneck

The fundamental problem with reducing the CPI relative to a 3-stage core is related to the von Neumann bottleneck - any stored-program computer with a single instruction and data memory will have its performance limited by the available memory bandwidth. A 3-stage ARM core accesses memory on (almost) every clock cycle either to fetch an instruction or to transfer data. Simply tightening up on the few cycles where the memory is not used will yield only a small performance gain. To get a significantly better CPI the memory system must deliver more than one value in each clock cycle either by delivering more than 32 bits per cycle from a single memory or by having separate memories for instruction and data accesses.

As a result of the above issues, higher performance ARM cores employ a 5-stage pipeline and have separate instruction and data memories. Breaking instruction execution down into five components rather than three reduces the maximum work which must be completed in a clock cycle, and hence allows a higher clock frequency to be used (provided that other system components, and particularly the instruction memory, are also redesigned to operate at this higher clock rate). The separate instruction and data memories (which may be separate caches connected to a unified instruction and data main memory) allow a significant reduction in the core's CPI.

A typical 5-stage ARM pipeline is that employed in the ARM9TDMI. The organization of the ARM9TDMI is illustrated in Figure 4.4 on page 81.

The 5-stage pipeline

The ARM processors which use a 5-stage pipeline have the following pipeline stages:

- Fetch;

  the instruction is fetched from memory and placed in the instruction pipeline.

- Decode;

  the instruction is decoded and register operands read from the register file. There are three operand read ports in the register file, so most ARM instructions can source all their operands in one cycle.

- Execute;

  an operand is shifted and the ALU result generated. If the instruction is a load or store the memory address is computed in the ALU.

- Buffer/data;

  data memory is accessed if required. Otherwise the ALU result is simply buffered for one clock cycle to give the same pipeline flow for all instructions.

- Write-back;

  the results generated by the instruction are written back to the register file, including any data loaded from memory.

This 5-stage pipeline has been used for many RISC processors and is considered to be the 'classic' way to design such a processor. Although the ARM instruction set was not designed with such a pipeline in mind, it maps onto it relatively simply. The principal concessions to the ARM instruction set architecture in the organization shown in Figure 4.4 on page 81 are the three source operand read ports and two write ports in the register file (where a 'classic' RISC has two read ports and one write port), and the inclusion of address incrementing hardware in the execute stage to support load and store multiple instructions.

Data forwarding A major source of complexity in the 5-stage pipeline (compared to the 3-stage pipeline) is that, because instruction execution is spread across three pipeline stages, the only way to resolve data dependencies without stalling the pipeline is to introduce *forwarding* paths.

Data dependencies arise when an instruction needs to use the result of one of its predecessors before that result has returned to the register file. (This issue was discussed previously under 'Pipeline hazards' on page 22.) Forwarding paths allow results to be passed between stages as soon as they are available, and the 5-stage ARM pipeline requires each of the three source operands to be forwarded from any of three intermediate result registers as shown in Figure 4.4 on page 81.

There is one case where, even with forwarding, it is not possible to avoid a pipeline stall. Consider the following code sequence:

```
LDR  rN, [ . . ]            ; load rN from somewhere
ADD    r2, r1, rN         ; and use it immediately
```

The processor cannot avoid a one-cycle stall as the value loaded into rN only enters the processor at the end of the buffer/data stage and it is needed by the following instruction at the start of the execute stage. The only way to avoid this stall is to encourage the compiler (or assembly language programmer) not to put a dependent instruction immediately after a load instruction.

Since the 3-stage pipeline ARM cores are not adversely affected by this code sequence, existing ARM programs will often use it. Such programs will run correctly

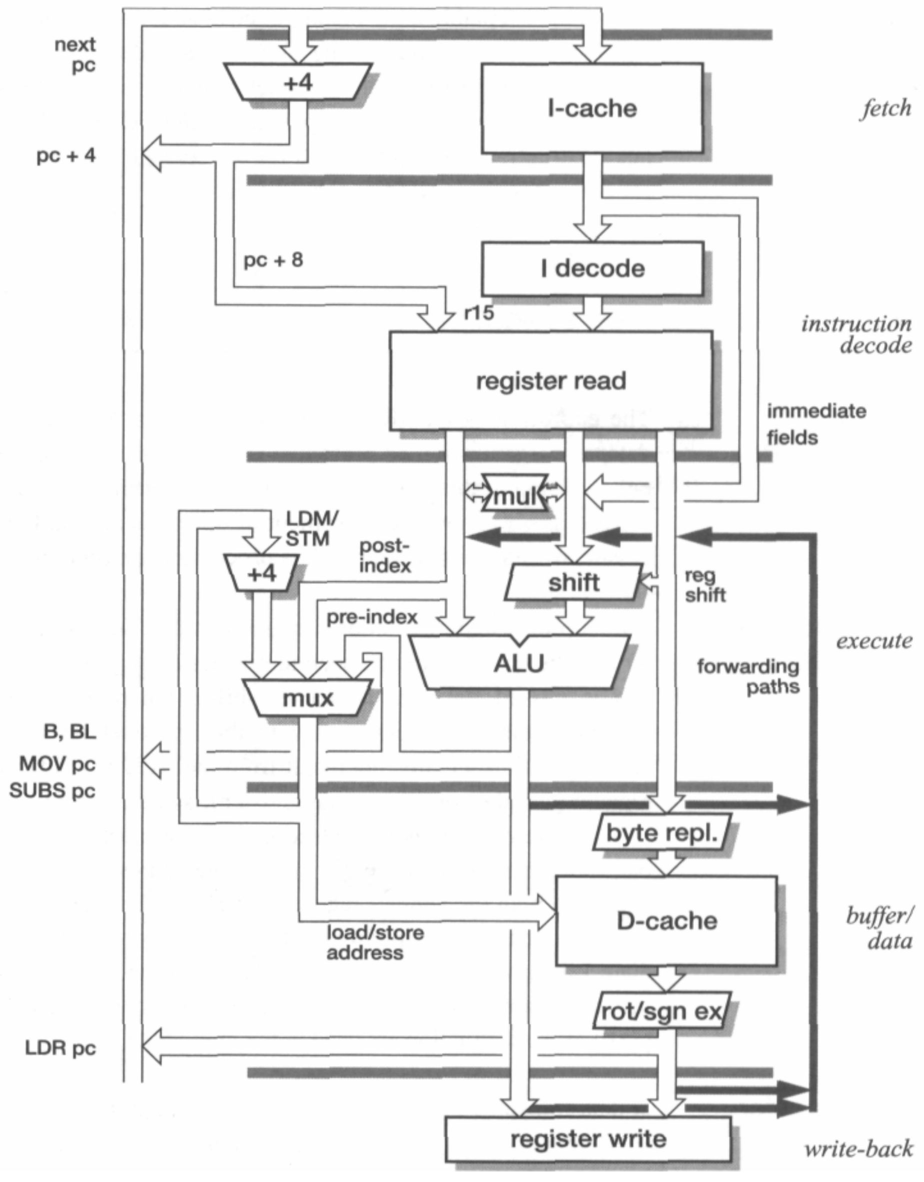


Figure 4.4 ARM9TDMI 5-stage pipeline organization.

on 5-stage ARM cores, but could probably be rewritten to run faster by simply reordering the instructions to remove these dependencies.

PC generation

The behaviour of r15, as seen by the programmer and described in 'PC behaviour' on page 78, is based on the operational characteristics of the 3-stage ARM pipeline. The 5-stage pipeline reads the instruction operands one stage earlier in the pipeline, and would naturally get a different value (PC+4 rather than PC+8). As this would lead to unacceptable code incompatibilities, however, the 5-stage pipeline ARMs all 'emulate' the behaviour of the older 3-stage designs. Referring to Figure 4.4, the incremented PC value from the fetch stage is fed directly to the register file in the decode stage, bypassing the pipeline register between the two stages. PC+4 for the next instruction is equal to PC+8 for the current instruction, so the correct r15 value is obtained without additional hardware.

4.3 ARM instruction execution

The execution of an ARM instruction can best be understood by reference to the datapath organization as presented in Figure 4.1 on page 76. We will use an annotated version of this diagram, omitting the control logic section, and highlighting the active buses to show the movement of operands around the various units in the processor. We start with a simple data processing instruction.

Data processing instructions

A data processing instruction requires two operands, one of which is always a register and the other is either a second register or an immediate value. The second operand is passed through the barrel shifter where it is subject to a general shift operation, then it is combined with the first operand in the ALU using a general ALU operation. Finally, the result from the ALU is written back into the destination register (and the condition code register may be updated).

All these operations take place in a single clock cycle as shown in Figure 4.5 on page 83. Note also how the PC value in the address register is incremented and copied back into both the address register and r15 in the register bank, and the next instruction but one is loaded into the bottom of the instruction pipeline *(i. pipe).* The immediate value, when required, is extracted from the current instruction at the top of the instruction pipeline. For data processing instructions only the bottom eight bits (bits [7:0]) of the instruction are used in the immediate value.

Data transfer instructions

A data transfer (load or store) instruction computes a memory address in a manner very similar to the way a data processing instruction computes its result. A register is used as the base address, to which is added (or from which is subtracted) an offset which again may be another register or an immediate value. This time, however, a 12-bit immediate value is used without a shift operation rather than a shifted 8-bit value. The address is sent to the address register, and in a second cycle the data transfer takes place. Rather than leave the datapath largely idle during the data transfer cycle, the ALU holds the address components from the first cycle and is available to compute an auto-indexing modification to the base register if this is required. (If auto-indexing is not required the computed value is not written back to the base register in the second cycle.)

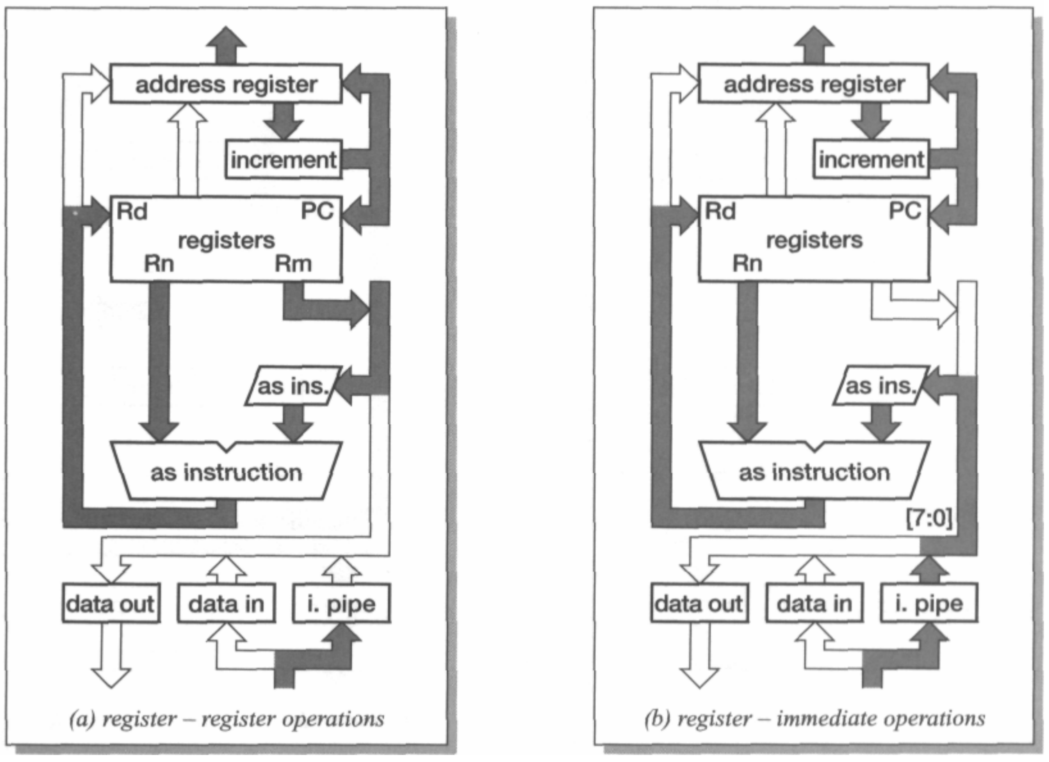


Figure 4.5 Data processing instruction datapath activity.

The datapath operation for the two cycles of a data store instruction (SIR) with an immediate offset are shown in Figure 4.6 on page 84. Note how the incremented PC value is stored in the register bank at the end of the first cycle so that the address register is free to accept the data transfer address for the second cycle, then at the end of the second cycle the PC is fed back to the address register to allow instruction prefetching to continue.

It should, perhaps, be noted at this stage that the value sent to the address register in a cycle is the value used for the memory access in *the following* cycle. The address register is, in effect, a pipeline register between the processor datapath and the external memory.

(The address register can produce the memory address for the next cycle a little before the end of the current cycle, moving responsibility for the pipeline delay out into the memory when this is desired. This can enable some memory devices to operate at higher performance, but this detail can be postponed for the time being. For now we will view the address register as a pipeline register to memory.)

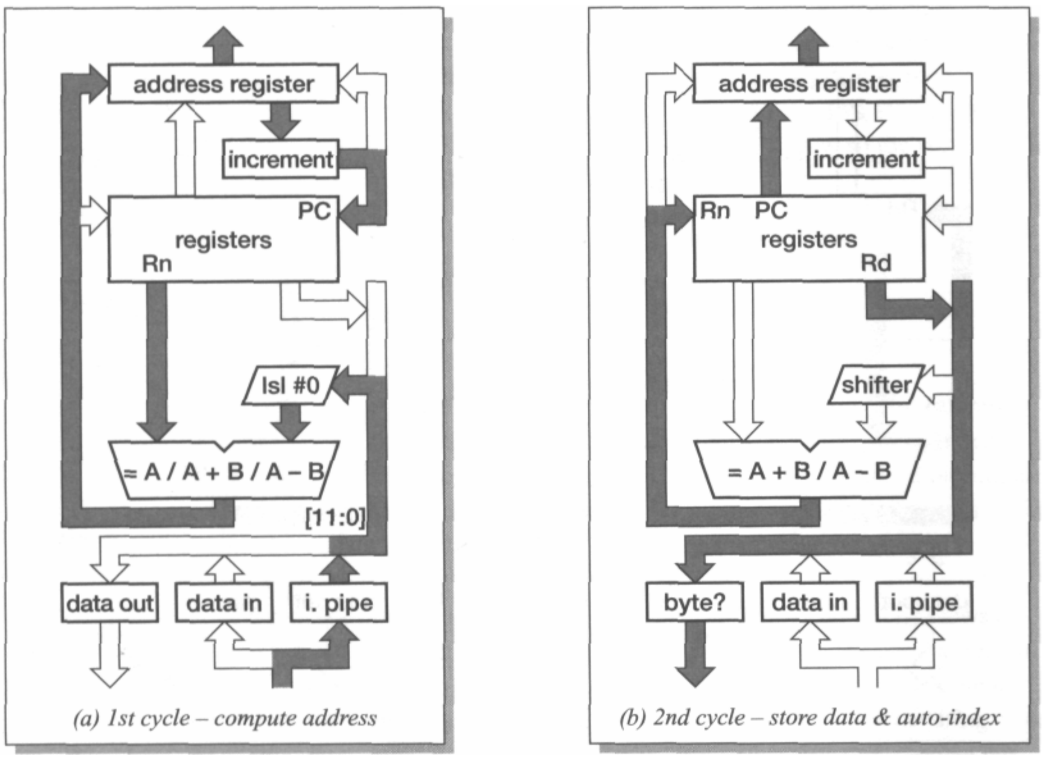


Figure 4.6 SIR (store register) datapath activity.

When the instruction specifies the store of a byte data type, the 'data out' block extracts the bottom byte from the register and replicates it four times across the 32-bit data bus. External memory control logic can then use the bottom two bits of the address bus to activate the appropriate byte within the memory system.

Load instructions follow a similar pattern except that the data from memory only gets as far as the 'data in' register on the second cycle and a third cycle is needed to transfer the data from there to the destination register.

Branch instructions

Branch instructions compute the target address in the first cycle as shown in Figure 4.7 on page 85. A 24-bit immediate field is extracted from the instruction and then shifted left two bit positions to give a word-aligned offset which is added to the PC. The result is issued as an instruction fetch address, and while the instruction pipeline refills the return address is copied into the link register (r14) if this is required (that is, if the instruction is a 'branch with link').

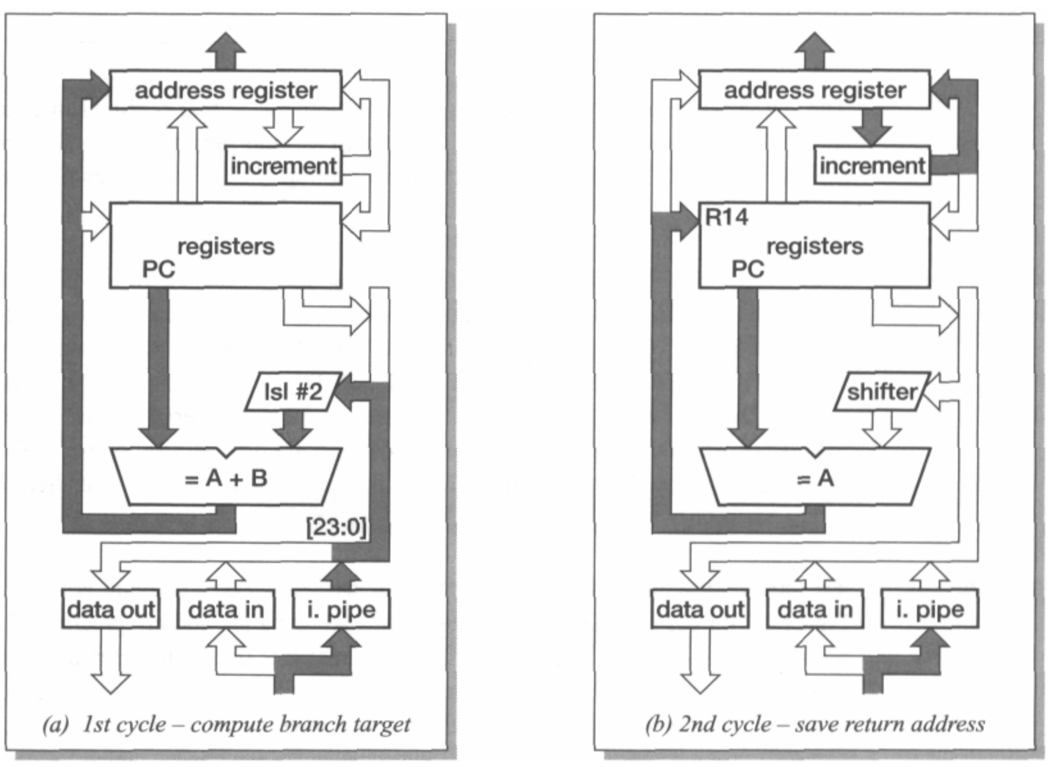


Figure 4.7 The first two (of three) cycles of a branch instruction.

The third cycle, which is required to complete the pipeline refilling, is also used to make a small correction to the value stored in the link register in order that it points directly at the instruction which follows the branch. This is necessary because r15 contains pc + 8 whereas the address of the next instruction is pc + 4 (see 'PC behaviour' on page 78).

Other ARM instructions operate in a similar manner to those described above. We will now move on to look in more detail at how the datapath carries out these operations.

4.4 ARM implementation

The ARM implementation follows a similar approach to that outlined in Chapter 1 for MU0; the design is divided into a datapath section that is described in *register transfer level* (RTL) notation and a control section that is viewed as *a finite state machine* (FSM).

Clocking scheme

Unlike the MU0 example presented in Section 1.3 on page 7, most ARMs do not operate with edge-sensitive registers; instead the design is based around 2-phase non-overlapping clocks, as shown in Figure 4.8, which are generated internally from a single input clock signal. This scheme allows the use of level-sensitive transparent latches. Data movement is controlled by passing the data alternately through latches which are open during phase 1 and latches which are open during phase 2. The non-overlapping property of the phase 1 and phase 2 clocks ensures that there are no race conditions in the circuit.

Datapath timing

The normal timing of the datapath components in a 3-stage pipeline is illustrated in Figure 4.9 on page 87. The register read buses are dynamic and are precharged during phase 2 (here 'dynamic' means that they are sometimes undriven and retain their logic values as electrical charge; charge-retention circuits are used to give pseudo-static behaviour so that data is not lost if the clock is stopped at any point in its cycle). When phase 1 goes high, the selected registers discharge the read buses which become valid early in phase 1. One operand is passed through the barrel shifter, which also uses dynamic techniques, and the shifter output becomes valid a little later in phase 1.

The ALU has input latches which are open during phase 1, allowing the operands to begin combining in the ALU as soon as they are valid, but they close at the end of phase 1 so that the phase 2 precharge does not get through to the ALU. The ALU then continues to process the operands through phase 2, producing a valid output towards the end of the phase which is latched in the destination register at the end of phase 2.

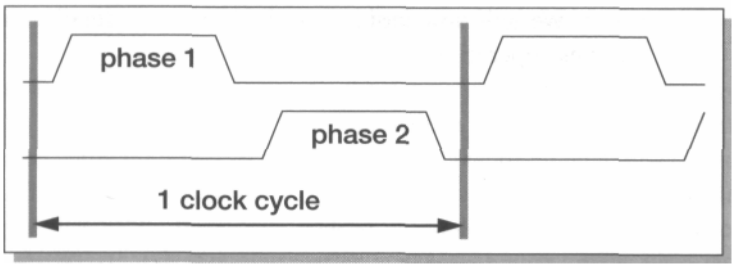


Figure 4.8 2-phase non-overlapping clock scheme.

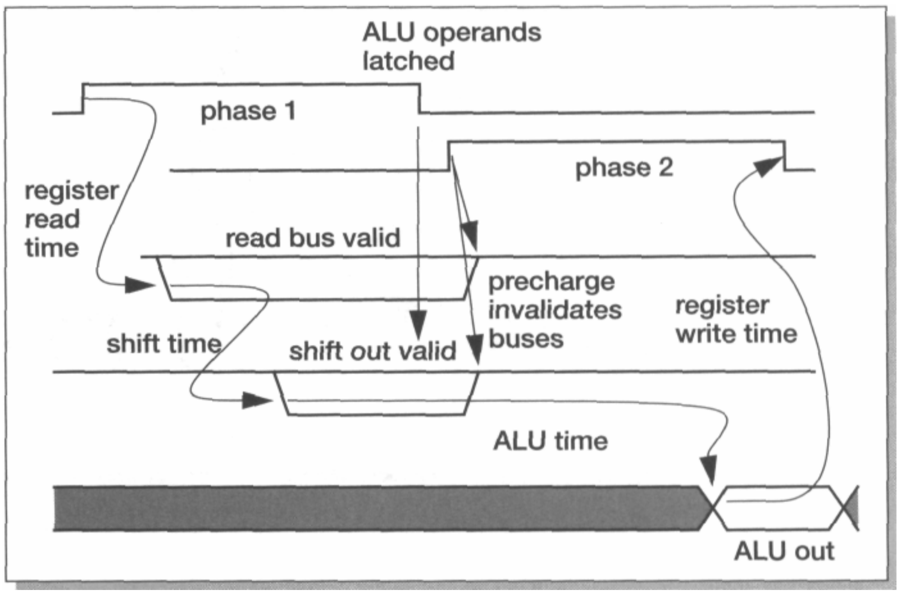


Figure 4.9 ARM datapath timing (3-stage pipeline).

Note how, though the data passes through the ALU input latches, these do not affect the datapath timing since they are open when valid data arrives. This property of transparent latches is exploited in many places in the design of the ARM to ensure that clocks do not slow critical signals.

The minimum datapath cycle time is therefore the sum of:

- the register read time;
- the shifter delay;
- the ALU delay;
- the register write set-up time;
- the phase 2 to phase 1 non-overlap time.

Of these, the ALU delay dominates. The ALU delay is highly variable, depending on the operation it is performing. Logical operations are relatively fast, since they involve no carry propagation. Arithmetic operations (addition, subtraction and comparisons) involve longer logic paths as the carry can propagate across the word width.

Adder design

Since the 32-bit addition time has a significant effect on the datapath cycle time, and hence the maximum clock rate and the processor's performance, it has been the focus of considerable attention during the development of successive versions of the ARM processor.

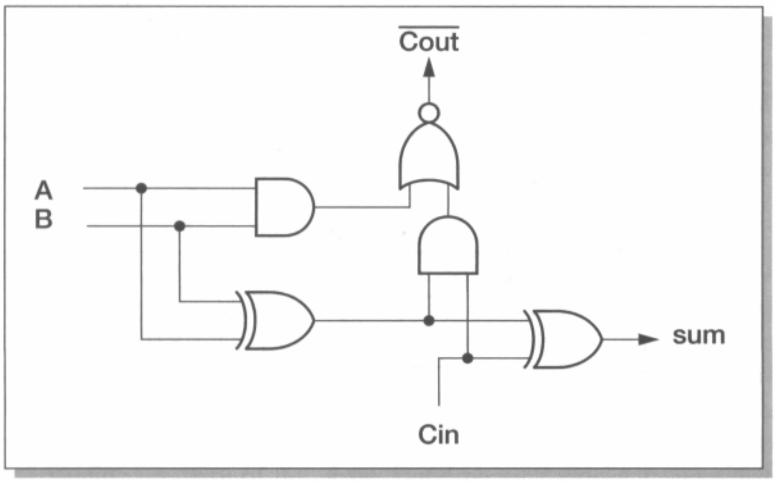


Figure 4.10 The original ARM1 ripple-carry adder circuit.

The first ARM processor prototype used a simple ripple-carry adder as shown in Figure 4.10. Using a CMOS AND-OR-INVERT gate for the carry logic and alternating AND/OR logic so that even bits use the circuit shown and odd bits use the dual circuit with inverted inputs and outputs and AND and OR gates swapped around, the worst-case carry path is 32 gates long.

In order to allow a higher clock rate, ARM2 used a 4-bit carry look-ahead scheme to reduce the worst-case carry path length. The circuit is shown in Figure 4.11 on page 89. The logic produces carry generate (G) and propagate (P) signals which control the 4-bit carry-out. The carry propagate path length is reduced to eight gate delays, again using merged AND-OR-INVERT gates and alternating AND/OR logic.

ALU functions

The ALU does not only add its two inputs. It must perform the full set of data operations defined by the instruction set, including address computations for memory transfers, branch calculations, bit-wise logical functions, and so on.

The full ARM2 ALU logic is illustrated in Figure 4.12 on page 89. The set of functions generated by this ALU and the associated values of the ALU function selects are listed in Table 4.1 on page 90.

The ARM6 carry-select adder

A further improvement in the worst-case add time was introduced on the ARM6 by using a carry-select adder. This form of adder computes the sums of various fields of the word for a carry-in of both zero and one, and then the final result is selected by using the correct carry-in value to control a multiplexer. The overall scheme is illustrated in Figure 4.13 on page 90.

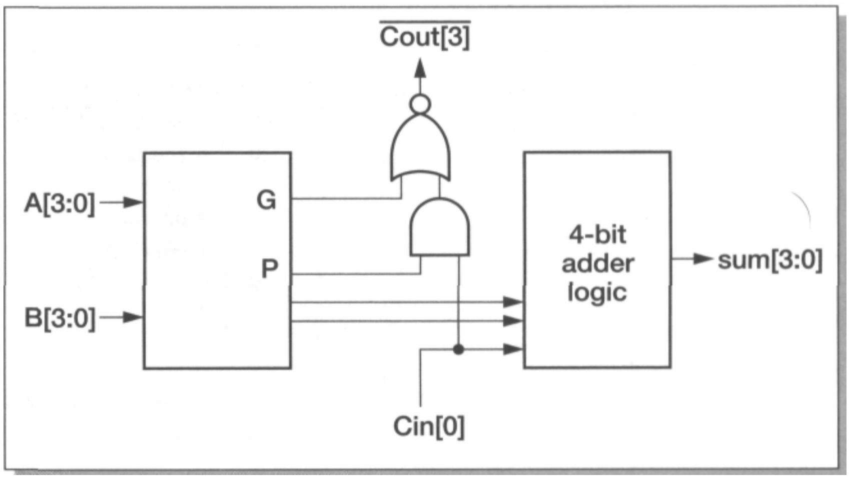


Figure 4.11 The ARM2 4-bit carry look-ahead scheme.

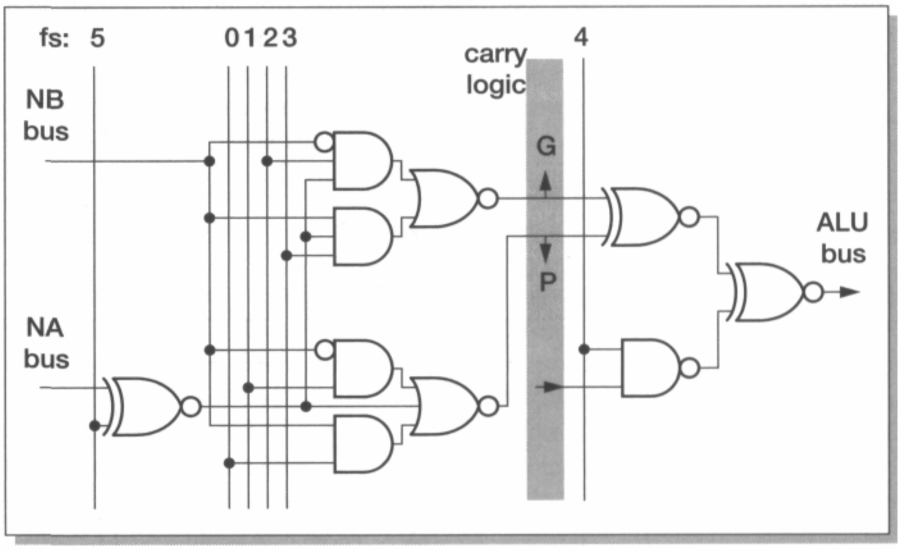


Figure 4.12 The ARM2 ALU logic for one result bit.

The critical path is now O(log$_2$[word width]) gates long, though direct comparison with previous schemes is difficult since the fan-out on some of these gates is high. However, the worst-case addition time is significantly faster than the 4-bit carry look-ahead adder at the cost of significantly increased silicon area.

Table 4.1 ARM2 ALU function codes.

fs5	fs4	fs3	fs2	fs1	fs0	ALU output
0	0	0	1	0	0	A and B
0	0	1	0	0	0	A and not B
0	0	1	0	0	1	A xor B
0	1	1	0	0	1	A plus not B plus carry
0	1	0	1	1	0	A plus B plus carry
1	1	0	1	1	0	not A plus B plus carry
0	0	0	0	0	0	A
0	0	0	0	0	1	A or B
0	0	0	1	0	1	B
0	0	1	0	1	0	not B
0	0	1	1	0	0	zero

ARM6 ALU structure

The ARM6 carry-select adder does not easily lead to a merging of the arithmetic and logic functions into a single structure as was used on ARM2. Instead, a separate logic unit runs in parallel with the adder, and a multiplexer selects the output from the adder or from the logic unit as required.

The overall ALU structure is shown in Figure 4.14 on page 91. The input operands are each selectively inverted, then added and combined in the logic unit, and finally the required result is selected and issued on the ALU result bus.

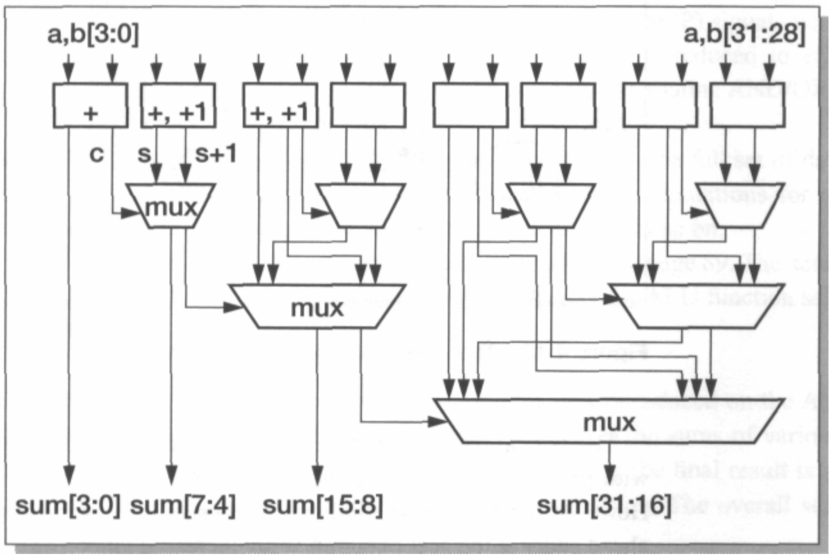


Figure 4.13 The ARM6 carry-select adder scheme.

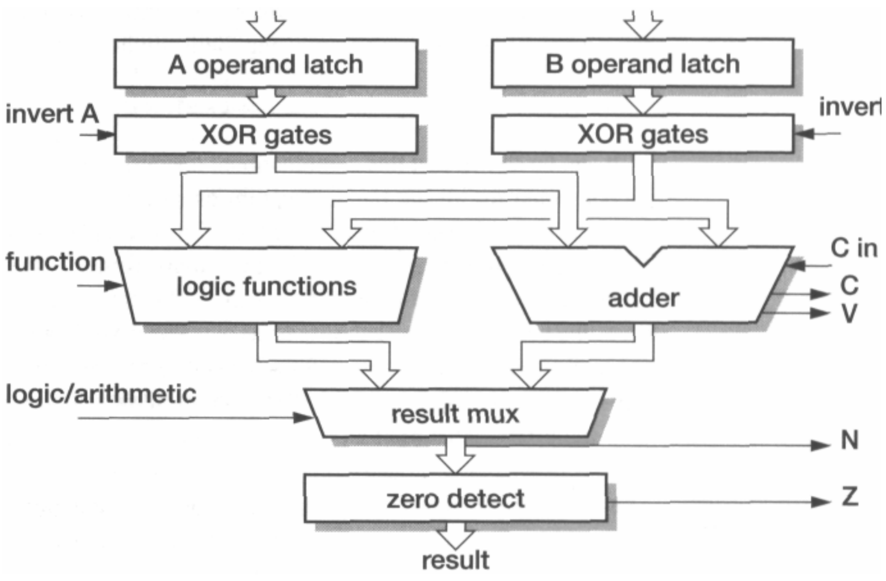


Figure 4.14 The ARM6 ALU organization.

The *C* and *V* flags are generated in the adder (they have no meaning for logical operations), the *N* flag is copied from bit 31 of the result and the *Z* flag is evaluated from the whole result bus. Note that producing the Z flag requires a 32-input NOR gate and this can easily become a critical path signal.

Carry arbitration adder

The adder logic was further improved on the ARM9TDMI, where a 'carry arbitration' adder is used. This adder computes all intermediate carry values using a 'parallel-prefix' tree, which is a very fast parallel logic structure.

The carry arbitration scheme recedes the conventional propagate-generate information in terms of two new variables, *u* and v. Consider the computation of the carry out, C, from a particular bit position in the adder with inputs A and B. Before the carry in is known, the information available is as shown in Table 4.2, which also shows how

Table 4.2 ARM9 carry arbitration encoding.

A	B	C	u	v
0	0	0	0	0
0	1	unknown	1	0
1	0	unknown	1	0
1	1	1	1	1

this information is encoded by *u* and v. This information can be combined with that from a neighbouring bit position using the formula:

$$(u,v) \bullet (w',v') = (v + u \bullet u',v + u \bullet v') \qquad \text{Equation 12}$$

It can be shown that this combinational operator is associative and hence *u* and v can be computed for all the bits in the sum using a regular parallel prefix tree. The logic required to implement Equation 12 can combine up to 4 pairs of inputs in a single CMOS gate, producing the new *u* and v outputs from a single transistor structure.

Furthermore, it can be seen that *u* gives the carry-out if the carry-in is one, and v gives the carry-out if the carry-in is zero, *u* and v can therefore be used to generate the *(Sum, Sum+1)* values required for a hybrid carry arbitration/carry select adder, resulting in a number of possible designs which allow a trade-off between performance, area and power consumption.

The barrel shifter

The ARM architecture supports instructions which perform a shift operation in series with an ALU operation, leading to the organization shown in Figure 4.1 on page 76. The shifter performance is therefore critical since the shift time contributes directly to the datapath cycle time as shown in the datapath timing diagram in Figure 4.9 on page 87.

(Other processor architectures tend to have the shifter in parallel with the ALU, so as long as the shifter is no slower than the ALU it does not affect the datapath cycle time.)

In order to minimize the delay through the shifter, a cross-bar switch matrix is used to steer each input to the appropriate output. The principle of the cross-bar switch is illustrated in Figure 4.15, where a 4 x 4 matrix is shown. (The ARM processors use a

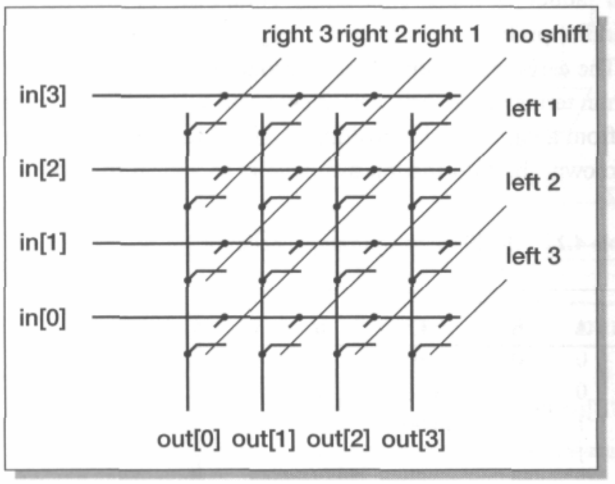


Figure 4.15 The cross-bar switch barrel shifter principle.

32 x 32 matrix.) Each input is connected to each output through a switch. If pre-charged dynamic logic is used, as it is on the ARM datapaths, each switch can be implemented as a single NMOS transistor.

The shifting functions are implemented by wiring switches along diagonals to a common control input:

- For a left or right shift function, one diagonal is turned on. This connects all the input bits to their respective outputs where they are used. (Not all are used, since some bits 'fall off the end'.) In the ARM the barrel shifter operates in negative logic where a ' 1' is represented as a potential near ground and a '0' by a potential near the supply. Precharging sets all the outputs to a logic '0', so those outputs that are not connected to any input during a particular switching operation remain at '0' giving the zero filling required by the shift semantics.
- For a rotate right function, the right shift diagonal is enabled together with the complementary left shift diagonal. For example, on the 4-bit matrix rotate right one bit is implemented using the 'right 1' and the 'left 3' (3 = 4 - 1) diagonals.
- Arithmetic shift right uses sign-extension rather than zero-fill for the unconnected output bits. Separate logic is used to decode the shift amount and discharge those outputs appropriately.

Multiplier design

All ARM processors apart from the first prototype have included hardware support for integer multiplication. Two styles of multiplier have been used:

- Older ARM cores include low-cost multiplication hardware that supports only the 32-bit result multiply and multiply-accumulate instructions.
- Recent ARM cores have high-performance multiplication hardware and support the 64-bit result multiply and multiply-accumulate instructions.

The low-cost support uses the main datapath iteratively, employing the barrel shifter and ALU to generate a 2-bit product in each clock cycle. Early-termination logic stops the iterations when there are no more ones in the multiply register.

The multiplier employs a modified Booth's algorithm to produce the 2-bit product, exploiting the fact that x3 can be implemented as x(-l)+ x4 . This allows all four values of the 2-bit multiplier to be implemented by a simple shift and add or subtract, possibly carrying the x 4 over to the next cycle.

The control settings for the Nth cycle of the multiplication are shown in Table 4.3 on page 94. (Note that the x 2 case is also implemented with a subtract and carry; it could equally well use an add with no carry, but the control logic is slightly simplified with this choice.)

Since this multiplication uses the existing shifter and ALU, the additional hardware it requires is limited to a dedicated two-bits-per-cycle shift register for the multiplier and a few gates for the Booth's algorithm control logic. In total this amounts to an overhead of a few per cent on the area of the ARM core.

Table 4.3 The 2-bit multiplication algorithm, Nth cycle.

Carry-in	Multiplier	Shift	ALU	Carry-out
0	×0	LSL #2N	A + 0	0
	×1	LSL #2N	A + B	0
	×2	LSL #(2N + 1)	A – B	1
	×3	LSL #2N	A – B	1
1	×0	LSL #2N	A + B	0
	×1	LSL #(2N + 1)	A + B	0
	×2	LSL #2N	A – B	1
	×3	LSL #2N	A + 0	1

High-speed multiplier

Where multiplication performance is very important, more hardware resource must be dedicated to it. In some embedded systems the ARM core is used to perform real-time digital signal processing (DSP) in addition to general control functions. DSP programs are typically multiplication intensive and the performance of the multiplication hardware can be critical to meeting the real-time constraints.

The high-performance multiplication used in some ARM cores employs a widely-used redundant binary representation to avoid the carry-propagate delays associated with adding partial products together. Intermediate results are held as partial sums and partial carries where the true binary result is obtained by adding these two together in a carry-propagate adder such as the adder in the main ALU, but this is only done once at the end of the multiplication. During the multiplication the partial sums and carries are combined in **carry-save** adders where the carries only propagate across one bit per addition stage. This gives the carry-save adder a much shorter logic path than the carry-propagate adder, which may have to propagate a carry across all 32 bits. Therefore several carry-save operations may be performed in a single clock cycle which can only accommodate one carry-propagate operation.

There are many ways to construct carry-save adders, but the simplest is the 3-input 2-output form. This accepts as inputs a partial sum, a partial carry and a partial product, all of the same binary weight, and produces as outputs a new partial sum and a new partial carry where the carry has twice the weight of the sum. The logic function for each bit is identical to a conventional full adder as used in a ripple-carry carry-propagate adder (see Figure 4.10 on page 88), but the structure is different. Figure 4.16 on page 95 illustrates the two structures. The carry-propagate adder takes two conventional (irredundant) binary numbers as inputs and produces a binary sum; the carry-save adder takes one binary and one redundant (partial sum and partial carry) input and produces a sum in redundant binary representation.

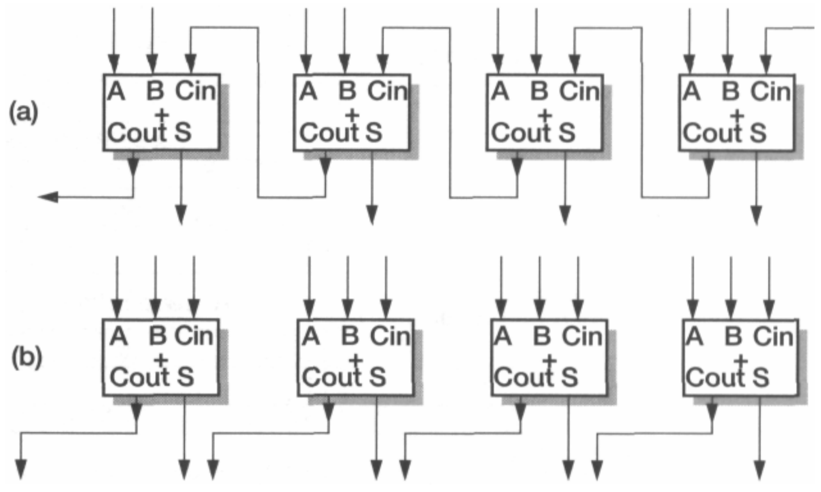


Figure 4.16 Carry-propagate (a) and carry-save (b) adder structures.

During the iterative multiplication stages, the sum is fed back and combined with one new partial product in each iteration. When all the partial products have been added, the redundant representation is converted into a conventional binary number by adding the partial sum and partial carry in the carry-propagate adder in the ALU.

High-speed multipliers have several layers of carry-save adder in series, each handling one partial product. If the partial product is produced following a modified Booth's algorithm similar to the one described in Table 4.3 on page 94, each stage of carry-save adder handles two bits of the multiplier in each cycle.

The overall structure of the high-performance multiplier used on some ARM cores is shown in Figure 4.17 on page 96. The register names refer to the instruction fields described in Section 5.8 on page 122. The carry-save array has four layers of adders, each handling two multiplier bits, so the array can multiply eight bits per clock cycle. The partial sum and carry registers are cleared at the start of the instruction, or the partial sum register may be initialized to the accumulate value. As the multiplier is shifted right eight bits per cycle in the 'Rs' register, the partial sum and carry are rotated right eight bits per cycle. The array is cycled up to four times, using early termination to complete the instruction in fewer cycles where the multiplier has sufficient zeros in the top bits, and the partial sum and carry are combined 32 bits at a time and written back into the register bank. (When the multiply terminates early some realignment of the partial sum and carry is required; this is not shown in Figure 4.17.)

The high-speed multiplier requires considerably more dedicated hardware than the low-cost solution employed on other ARM cores. There are 160 bits of shift register and 128 bits of carry-save adder logic. The incremental area cost is around 10% of the simpler processor cores, though a rather smaller proportion of the higher-performance cores such as ARMS and StrongARM. Its benefits are that it speeds up multiplication

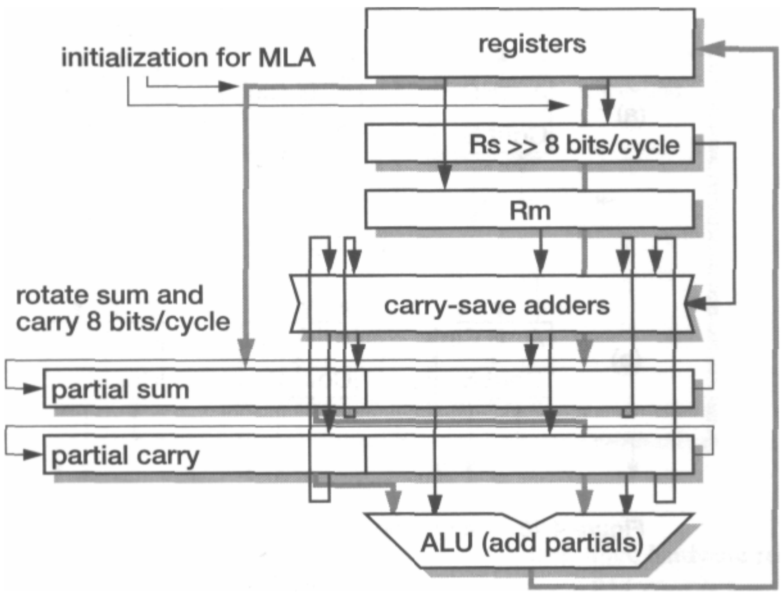


Figure 4.17 ARM high-speed multiplier organization.

by a factor of around 3 and it supports the added functionality of the 64-bit result forms of the multiply instruction.

The register bank

The last major block on the ARM datapath is the register bank. This is where all the user-visible state is stored in 31 general-purpose 32-bit registers, amounting to around 1 Kbits of data altogether. Since the basic 1-bit register cell is repeated so many times in the design, it is worth putting considerable effort into minimizing its size.

The transistor circuit of the register cell used in ARM cores up to the ARM6 is shown in Figure 4.18 on page 97. The storage cell is an asymmetric cross-coupled pair of CMOS inverters which is overdriven by a strong signal from the ALU bus when the register contents are changed. The feedback inverter is made weak in order to minimize the cell's resistance to the new value. The *A* and *B* read buses are precharged to *Vdd* during phase 2 of the clock cycle, so the register cell need only discharge the read buses, which it does through n-type pass transistors when the read-lines are enabled.

This register cell design works well with a 5 volt supply, but writing a ' 1' through the n-type pass transistor becomes difficult at lower supply voltages. Since a low supply voltage gives good power-efficiency, ARM cores since the ARM6 have either used a full CMOS transmission gate (with a p-type transistor in parallel with the n-type pass transistor in the write circuit, requiring complementary write enable control lines) or a more sophisticated register circuit.

These register cells are arranged in columns to form a 32-bit register, and the columns are packed together to form the complete register bank. The decoders for the

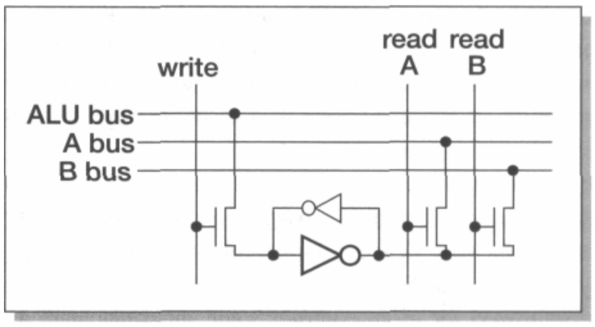


Figure 4.18 ARM6 register cell circuit.

read and write enable lines are then packed above the columns as shown in Figure 4.19, so the enables run vertically and the data buses horizontally across the array of register cells. Since a decoder is logically more complex than the register cell itself, but the horizontal pitch is chosen to suit the cell, the decoder layout can become very tight and the decoders themselves have to be tall and thin.

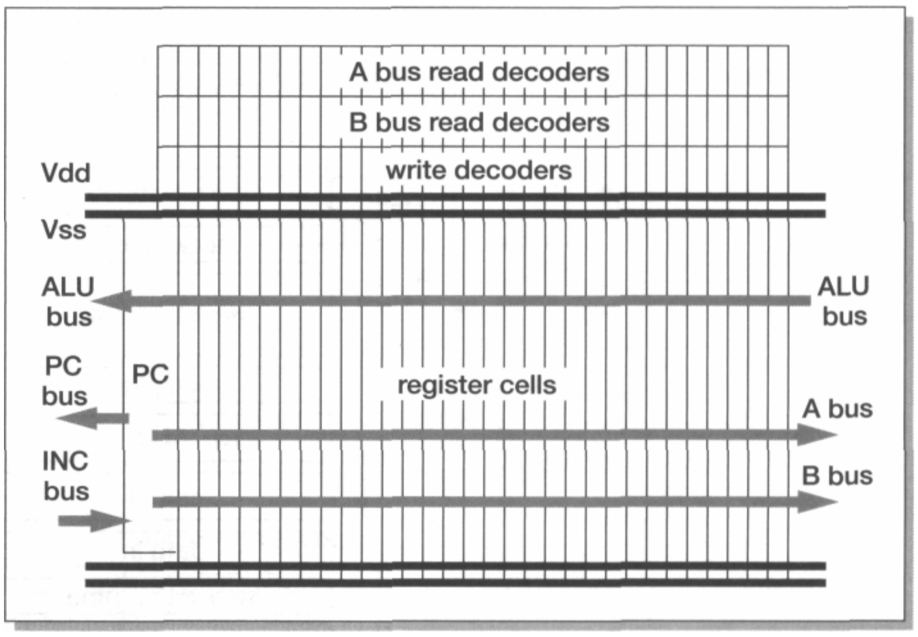


Figure 4.19 ARM register bank floorplan.

The ARM program counter register is physically part of the register bank in the simpler cores, but it has two write and three read ports whereas the other registers have one write and two read ports. The symmetry of the register array is preserved by putting the PC at one end where it is accessible to the additional ports and it can be allowed to have a 'fatter' profile.

The register bank accounts for around one-third of the total transistor count of the simpler ARM cores, but takes a proportionately much smaller share of the silicon area by virtue of its very dense, memory-like structure. It does not match the transistor density of a block of SRAM since it has two read ports and fits on a datapath pitch that is optimized for more complex logic functions such as the ALU. However, it is much denser than those logic functions due to its higher regularity.

Datapath layout

The ARM datapath is laid out to a constant pitch per bit. The pitch will be a compromise between the optimum for the complex functions (such as the ALU) which are best suited to a wide pitch and the simple functions (such as the barrel shifter) which are most efficient when laid out on a narrow pitch.

Each function is then laid out to this pitch, remembering that there may also be buses passing over a function (for example the B bus passes through the ALU but is not used by it); space must be allowed for these. It is a good idea to produce a floor-plan for the datapath noting the 'passenger' buses through each block, as illustrated in Figure 4.20. The order of the function blocks is chosen to minimize the number of additional buses passing over the more complex functions.

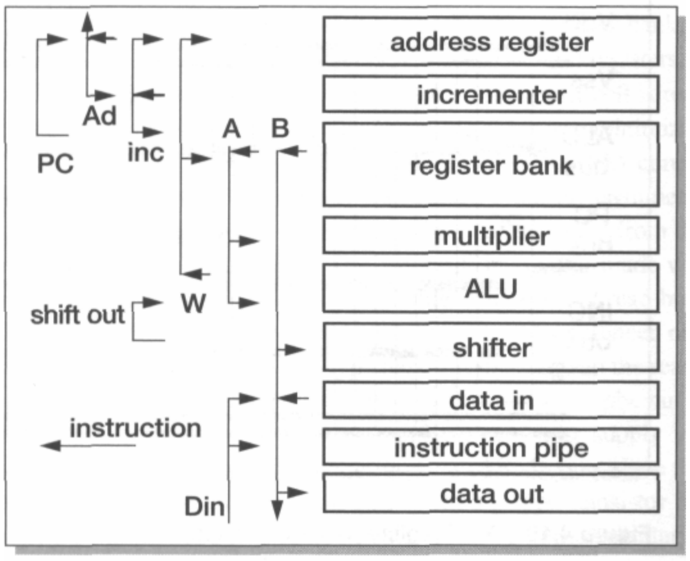


Figure 4.20 ARM core datapath buses.

Modern CMOS processes allow wiring in several metal layers (the early ARM cores used two metal layers). The wiring layers used for power and ground, bus signals along the datapath and control signals across the datapath must be chosen carefully (for example on ARM2 *Vdd* and *Vss* run along both sides of the datapath in metal 2, control wires pass across the datapath in metal 1 and buses run along it in metal 2).

Control structures

The control logic on the simpler ARM cores has three structural components which relate to each other as shown in Figure 4.21.

1. An instruction decoder PLA (programmable logic array). This unit uses some of the instruction bits and an internal cycle counter to define the class of operation to be performed on the datapath in the next cycle.
2. Distributed secondary control associated with each of the major datapath func tion blocks. This logic uses the class information from the main decoder PLA to select other instruction bits and/or processor state information to control the datapath.
3. Decentralized control units for specific instructions that take a variable number of cycles to complete (load and store multiple, multiply and coprocessor opera tions). Here the main decoder PLA locks into a fixed state until the remote con trol unit indicates completion.

The main decoder PLA has around 14 inputs, 40 product terms and 40 outputs, the precise number varying slightly between the different cores. On recent ARM cores it

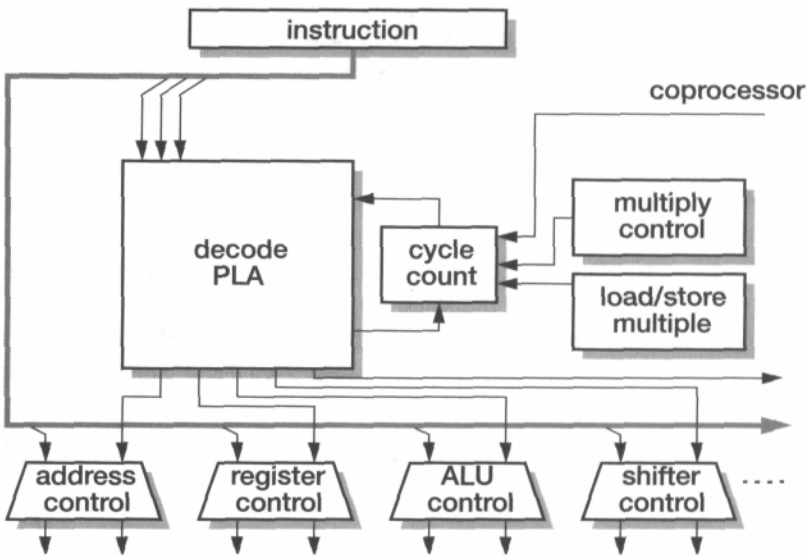


Figure 4.21 ARM control logic structure.

is implemented as two PLAs: a small, fast PLA which generates the time critical outputs and a larger, slower PLA which generates all the other outputs. Functionally, however, it can be viewed as a single PLA unit.

The 'cycle count' block distinguishes the different cycles of multi-cycle instructions so that the decode PLA can generate different control outputs for each cycle. It is, in fact, not simply a counter but a more general finite state machine capable of skipping unneeded cycles and of locking into a fixed state. It determines when the current instruction is about to complete and initiates the transfer of the next instruction from the instruction pipeline, including aborting instructions at the end of their first cycle if they fail their condition test. However, much of the time its behaviour is like that of a simple counter so it is not *too* misleading to think of it as an instruction cycle counter.

Physical design

So far we have been concerned principally with the logic design of an ARM core and said little about its physical implementation on a particular CMOS process.

There are two principal mechanisms used to implement an ARM processor core (or any other core, for than matter) on a particular process:

- a hard macrocell is delivered as physical layout ready to be incorporated into the final design;
- a soft macrocell is delivered as a synthesizable design expressed in a hardware description language such as VHDL.

A hard macrocell can be fully characterized on the target process and can exploit the area advantages of full-custom hand-tuned layout, but it can be used only on the particular process for which it has been designed. The layout must be modified and recharacterized for every new process. A soft macrocell can readily be ported to a new process technology, but after each process change it must still be recharacterized.

Early ARM cores were delivered only as hard macrocells. Their design was based upon full-custom datapaths, with control logic designed at the logic schematic level and converted to layout using automatic place and route tools and a standard cell library. To ease process portability the cores were designed using generic design rules (for both the cell library and the full-custom datapath) that allowed geometrical transformations of the same physical layout to be used to map the same physical layout onto a range of individual processes with similar but not identical design rules.

Recent ARM cores have been available in both hard and soft forms. The hard macrocells increasingly use synthesis for their control logic while retaining hand-drawn full-custom datapaths. The soft macrocells are fully synthesizable from a register transfer level (RTL) description.

Some ARM partners have adopted a middle course, using a gate-level netlist description of an ARM core as their basis for porting to new processes. The porting procedure no longer involves resynthesis, but simply mapping the same netlist (using automatic place and route tools) onto a standard cell library implemented on the new process.

The choice between hard and soft macrocells (or gate-level netlists) is a complex decision. Hard macrocells can clearly give the best area, performance and power-efficiency on a process, but it takes significant time, effort and cost to port them to each process. Soft macrocells and portable netlists are more flexible, and automated tools are now of a quality that means that they come close to hand layout in performance. The portablility of the soft macrocell may mean that the choice is between a soft macrocell on the latest process or a hard macrocell on an older process, and the process technology advantage could easily outweigh the slight loss of optimization.

4.5 The ARM coprocessor interface

The ARM supports a general-purpose extension of its instruction set through the addition of hardware coprocessors, and it also supports the software emulation of these coprocessors through the undefined instruction trap.

Coprocessor architecture

The coprocessor architecture is described in Section 5.16 on page 136. Its most important features are:

- Support for up to 16 logical coprocessors.
- Each coprocessor can have up to 16 private registers of any reasonable size; they are not limited to 32 bits.
- Coprocessors use a load-store architecture, with instructions to perform internal operations on registers, instructions to load and save registers from and to memory, and instructions to move data to or from an ARM register.

The simpler ARM cores offer the coprocessor interface at board level, so a co-processor may be introduced as a separate component. High clock speeds make board-level interfacing very difficult, so the higher-performance ARMs restrict the coprocessor interface to on-chip use, in particular for cache and memory management control functions, but other on-chip coprocessors may also be supported.

ARM7TDMI coprocessor interface

The ARM7TDMI coprocessor interface is based on 'bus watching' (other ARM cores use different techniques). The coprocessor is attached to a bus where the ARM instruction stream flows into the ARM, and the coprocessor copies the instructions into an internal pipeline that mimics the behaviour of the ARM instruction pipeline. As each coprocessor instruction begins execution there is a 'hand-shake' between the ARM and the coprocessor to confirm that they are both ready to execute it. The handshake uses three signals:

1. *cpi* (from ARM to all coprocessors).

   This signal, which stands for 'Coprocessor Instruction', indicates that the ARM has identified a coprocessor instruction and wishes to execute it.

2. *cpa* (from the coprocessors to ARM).

   This is the 'Coprocessor Absent' signal which tells the ARM that there is no coprocessor present that is able to execute the current instruction.

3. *cpb* (from the coprocessors to ARM).

   This is the 'CoProcessor Busy' signal which tells the ARM that the coprocessor cannot begin executing the instruction yet.

The timing is such that both the ARM and the coprocessor must generate their respective signals autonomously. The coprocessor cannot wait until it sees *cpi* before generating *cpa* and *cpb.*

Handshake outcomes

Once a coprocessor instruction has entered the ARM7TDMI and coprocessor pipelines, there are four possible ways it may be handled depending on the handshake signals:

1. The ARM may decide not to execute it, either because it falls in a branch shadow or because it fails its condition code test. (All ARM instructions are con ditionally executed, including coprocessor instructions.) ARM will not assert *cpi,* and the instruction will be discarded by all parties.
2. The ARM may decide to execute it (and signal this by asserting *cpi),* but no present coprocessor can take it so *cpa* stays active. ARM will take the unde fined instruction trap and use software to recover, possibly by emulating the trapped instruction.
3. ARM decides to execute the instruction and a coprocessor accepts it, but cannot execute it yet. The coprocessor takes *cpa* low but leaves *cpb* high. The ARM will 'busy-wait' until the coprocessor takes *cpb* low, stalling the instruction stream at this point. If an enabled interrupt request arrives while the coprocessor is busy, ARM will break off to handle the interrupt, probably returning to retry the coprocessor instruction later.
4. ARM decides to execute the instruction and a coprocessor accepts it for immedi ate execution, *cpi, cpa* and *cpb* are all taken low and both sides commit to com plete the instruction.

Data transfers

If the instruction is a coprocessor data transfer instruction the ARM is responsible for generating an initial memory address (the coprocessor does not require any connection to the address bus) but the coprocessor determines the length of the transfer; ARM will continue incrementing the address until the coprocessor signals completion. The *cpa* and *cpb* handshake signals are also used for this purpose.

Since the data transfer is not interruptible once it has started, coprocessors should limit the maximum transfer length to 16 words (the same as a maximum length load or store multiple instruction) so as not to compromise the ARM's interrupt response.

Pre-emptive execution

A coprocessor may begin executing an instruction as soon as it enters its pipeline so long as it can recover its state if the handshake does not ultimately complete. All activity must be id em potent (repeatable with identical results) up to the point of commitment.

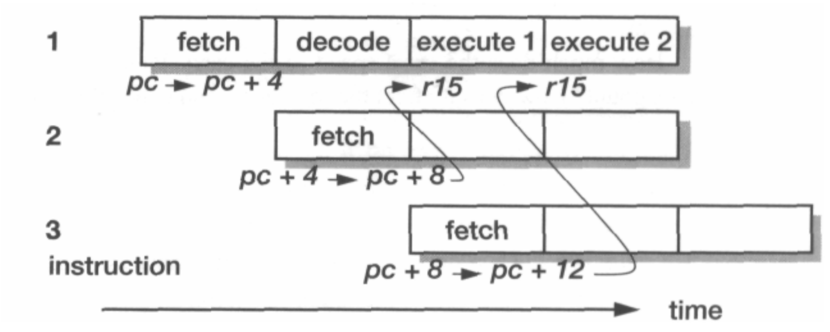
4.6 Examples and exercises

Example 4.1 **Why does r15 give pc + 8 in the first cycle of an instruction and pc + 12 in subsequent cycles on an ARM7?**

This is the ARM pipeline being exposed to the programmer. Referring back to Figure 4.2 on page 77, we can see that the pc value was incremented once when the current instruction (' 1' in the figure below) was fetched and once when its successor ('2') was fetched, giving pc + 8 at the start of the first execute cycle. During the first execute cycle a third instruction ('3') is fetched, giving pc + 12 in all subsequent execute cycles.

While multi-cycle instructions interrupt the pipeline flow they do not affect this aspect of the behaviour. An instruction always fetches the next-instruction-but-one during its first execute cycle, so r15 always progresses from pc + 8 at the start of the first execute cycle to pc + 12 at the start of the second (and subsequent) execute cycle(s).

(Note that other ARM processors do not share this behaviour, so it should never be relied upon when writing ARM programs.)



Exercise 4.1.1 Draw a pipeline flow diagram along the lines of the one above to illustrate the timing of an ARM branch instruction. (The branch target is computed in the first execute cycle of the instruction and issued to memory in the following cycle.)

Exercise 4.1.2 How many execute cycles are there after the branch target calculation and before the instruction at the branch target is ready to execute? What does the processor use these execute cycles for?

Example 4.2 **Complete the ARM2 4-bit carry logic circuit outlined in Figures 4.1 1 and 4. 12 on page 89.**

The 4-bit carry look-ahead scheme uses the individual bit carry generate and propagate signals produced by the logic shown in Figure 4.12. Denoting these by G[3:0] and P[3:0], the carry-out from the top bit of a 4-bit group is given by:

Cout = G[3] + P[3].(G[2] + P[2].(G[1] + P[1].(G[0] + P[0].Cin)))

Therefore the group generate and propagate signals, G4 and P4, as used in Figure 4. 1 1 are given by:

G4 = G[3] + P[3].(G[2] + P[2].(G[1] + P[1].G[0]))

P4 =P[3].P[2].P[1].P[0]

These two signals are independent of the carry-in signal and therefore can be set up ready for its arrival, allowing the carry to propagate across the 4-bit group in just one AND-OR-INVERT gate delay.

Exercise 4.2.1 Write a logic expression for one bit of the ALU output generated by the circuit shown in Figure 4.12 in terms of the inputs and the function select lines, and hence show how all the ALU functions listed in Table 4. 1 on page 90 are generated.

Exercise 4.2.2 Estimate the gate count for the ripple-carry adder and for the 4-bit carry look-ahead adder, basing both designs on the circuit in Figure 4.12 and varying only the carry scheme.

How much does the extra speed of the carry look-ahead scheme cost in terms of gate count? How does it affect the regularity, and hence the design cost, of the adder?