Unit 1

An Introduction to Microprocessor

This chapter covers

- Processor architecture and organization
- Abstraction in hardware design
- MU0 - a simple processor
- Instruction set design
- Processor design tradeoffs
- The Reduced Instruction Set Computer
- Cortex-M0 Technical Overview
- Implementation Features
- **Self-Study:**
  - **System Features**
  - **Debug Features**
  - **Advantages**

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.

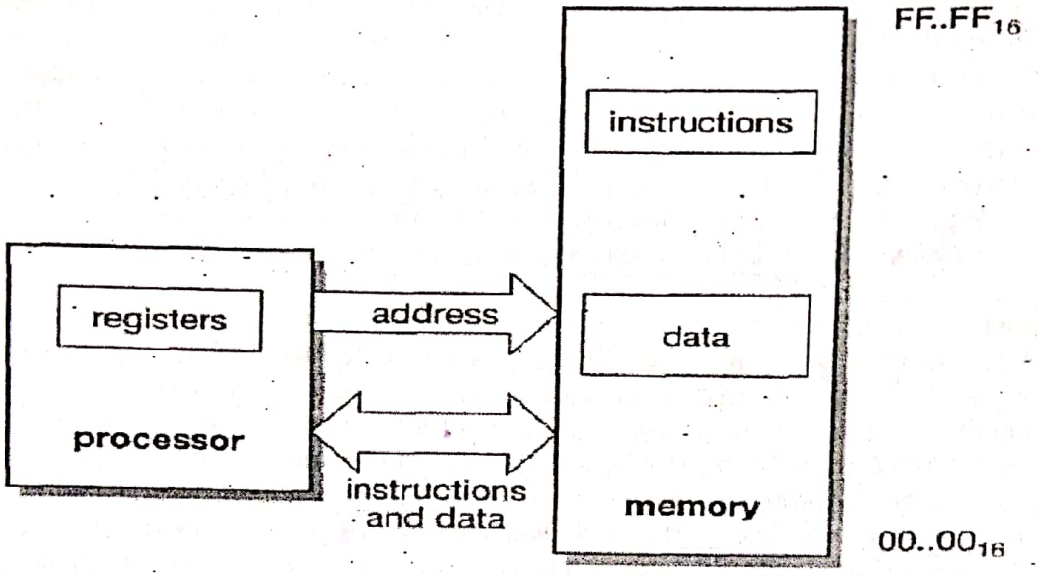


Figure 1.1 The state in a stored-program digital computer.

The Stored Computer-

The stored-program digital computer keeps its instructions and data in the same program 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.

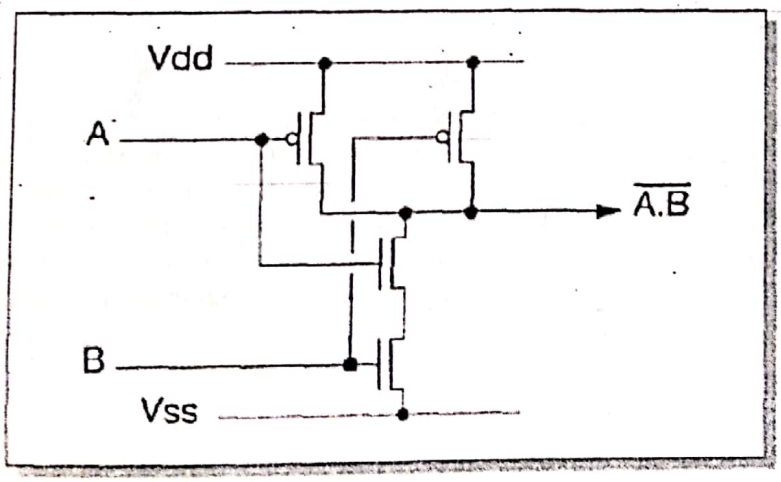


Figure 1.2 The transistor circuit of a static 2-input CMOS NAND gate.

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 = —(A^B)

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:

A logic symbol. (Imp of logic gates)

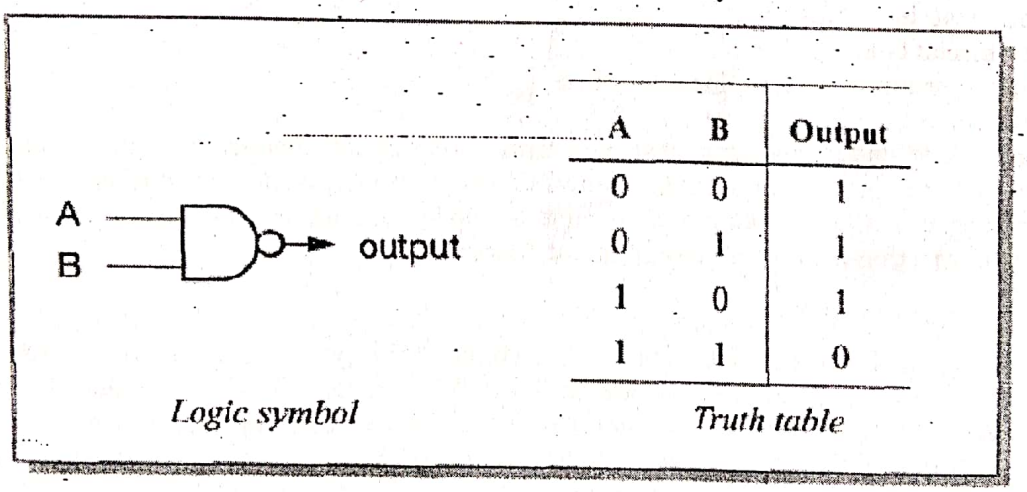
This is a symbol that represents a NAND gate function in a circuit schematic; there are similar symbols for other logic gates (for instance, removing the bubble from the output leaves an AND gate which generates the opposite output function; further examples are given in 'Appendix: Computer Logic' on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

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

Truth table

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 + mem16[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 ≥ 0 PC := S
JNE S	0110	if ACC ≠ 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:

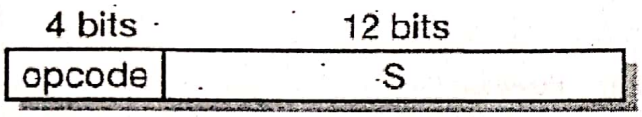


Figure 1.4 The MU0 instruction format.

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.

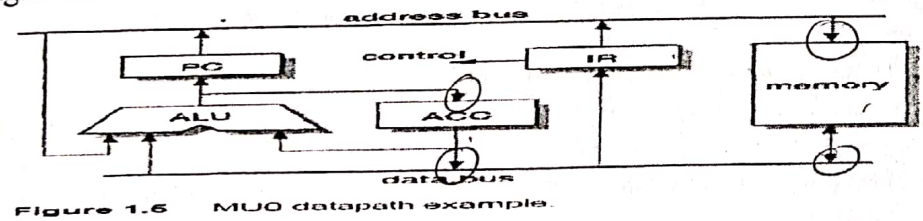


Figure 1.5 MU0 datapath example

(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

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 00016. 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.

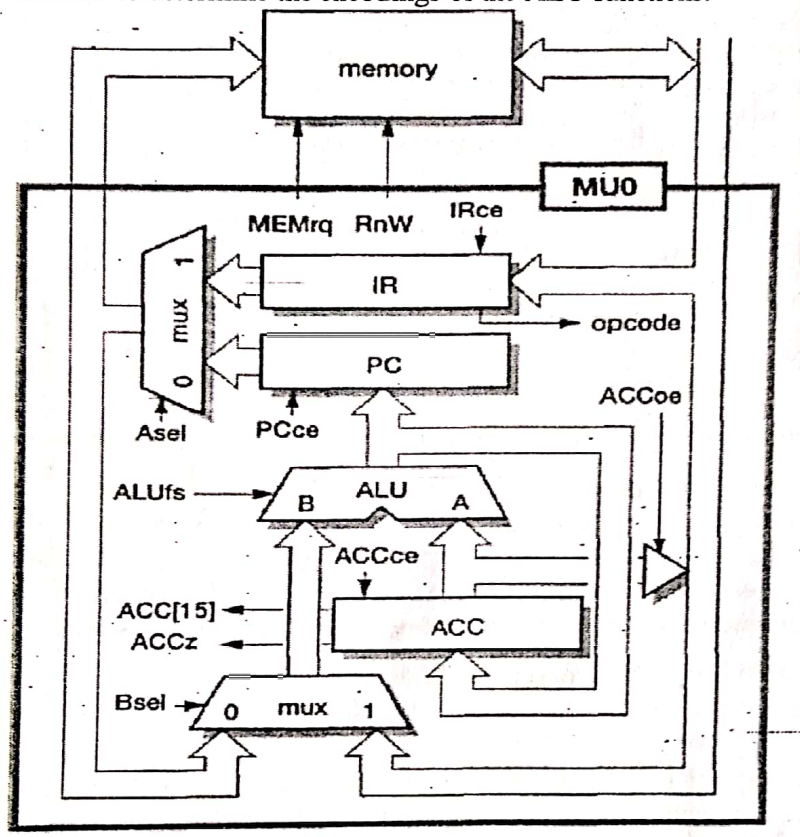


Figure 1.6 MU0 register transfer level organization.

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.

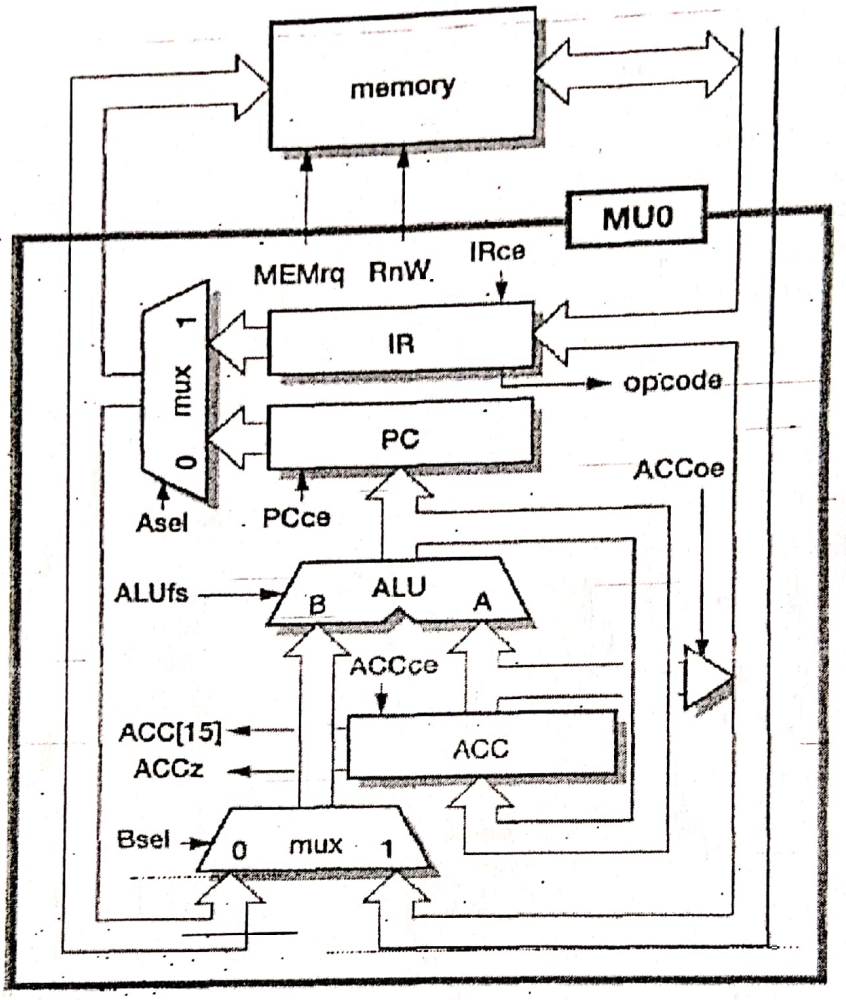


Figure 1.6 MU0 register transfer level organization.

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	ACC15 ACCz		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.

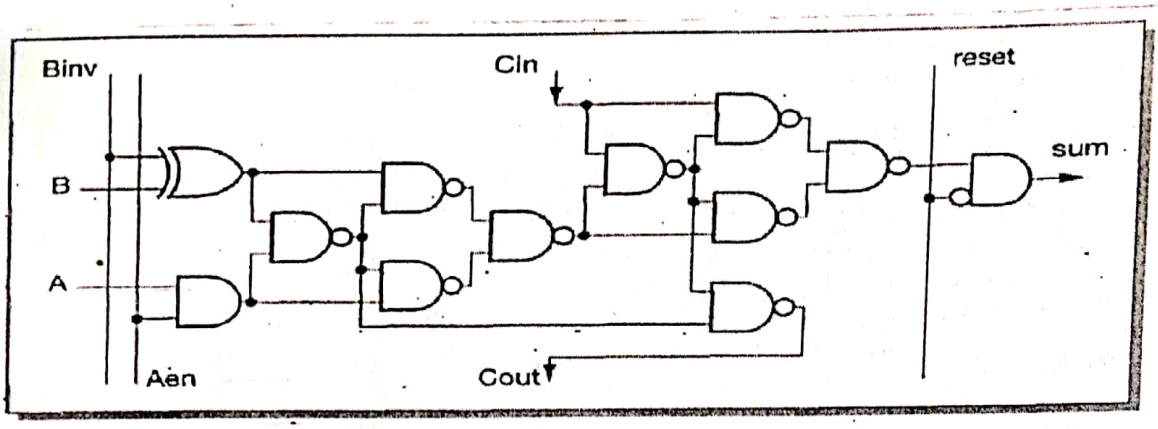


Figure 1.7 MU0 ALU logic for one bit.

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 Instruction

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.

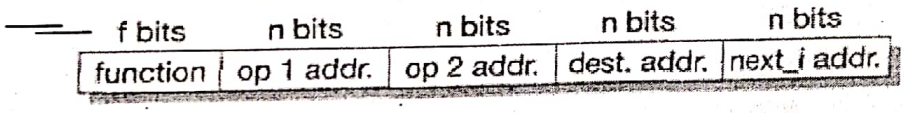


Figure 1.8 A 4-address instruction format.

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.

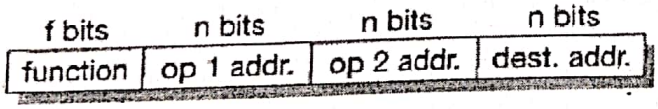


Figure 1.9 A 3-address instruction format.

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.

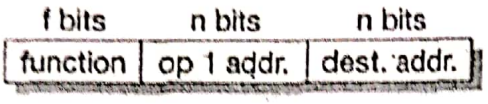


Figure 1.10 A 2-address instruction format.

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.

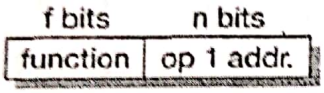


Figure 1.11 A 1-address (accumulator) instruction format.

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 constant (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 standard 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.

Pipelining
Cache memory
Super-scalar inst execution.

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.

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.

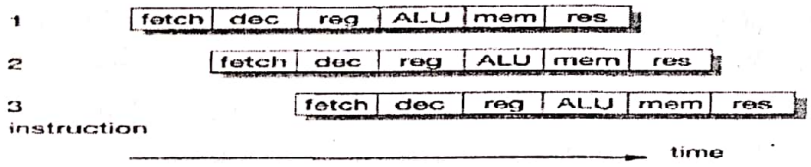


Figure 1.13 Pipelined instruction execution.

Pipeline is h/w sol^{n} for ILP pgm.

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)

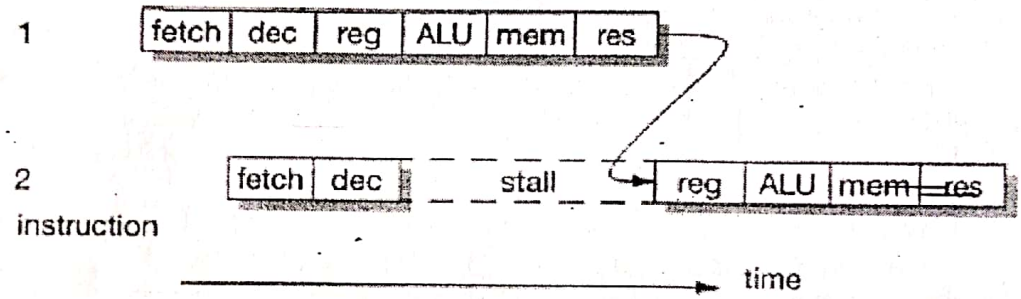


Figure 1.14 Read-after-write pipeline hazard.

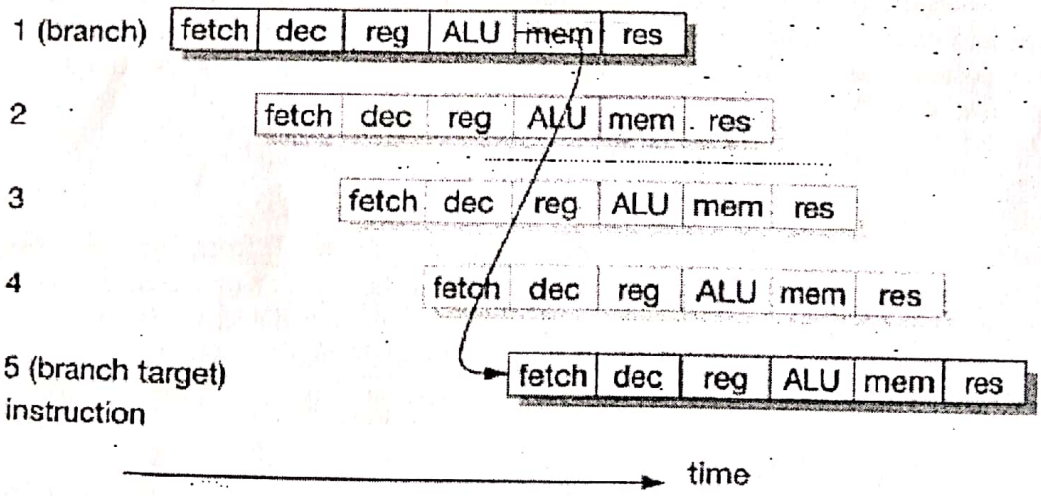


Figure 1.15 Pipelined branch behaviour.

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.

CISC failure — limited design resource
high complexity design of pipeline.

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 Cortex-M0 Technical Overview

General Information on the Cortex-M0 Processor

The Cortex-M0 processor is a 32-bit Reduced Instruction Set Computing (RISC) processor with a von Neumann architecture (single bus interface). It uses an instruction set called Thumb, which was first supported in the ARM7TDMI processor; however, several newer instructions from the ARMv6 architecture and a few instructions from the Thumb-2 technology are also included. Thumb-2 technology extended the previous Thumb instruction set to allow all operations to be carried out in one CPU state. The instruction set in Thumb-2 included both 16-bit and 32-bit instructions; most instructions generated by the C compiler use the 16-bit instructions, and the 32-bit instructions are used when the 16-bit version cannot carry out the required operations. This results in high code density and avoids the overhead of switching between two instruction sets.

In total, the Cortex-M0 processor supports only 56 base instructions, although some instructions can have more than one form. Although the instruction set is small, the Cortex- M0 processor is highly capable because the Thumb instruction set is highly optimized. Academically, the Cortex-M0 processor is classified as load-store architecture, as it has separate instructions for reading and writing to memory, and instructions for arithmetic or logical operations that use registers.

A simplified block diagram of the Cortex-M0 is shown in Figure 2.1.

The processor core contains the register banks, ALU, data path, and control logic. It is a three stage pipeline design with fetch stage, decode stage, and execution stage. The register bank has sixteen 32-bit registers. A few registers have special usages.

The Nested Vectored Interrupt Controller (NVIC) accepts up to 32 interrupt request signals and a non maskable interrupt (NMI) input. It contains the functionality required for comparing priority between interrupt requests and the current priority level so that nested interrupts can be handled automatically. If an interrupt is accepted, it communicates with the processor so that the processor can execute the correct interrupt handler.

The Wakeup Interrupt Controller (WIC) is an optional unit. In low-power applications, the microcontroller can enter standby state with most of the processor powered down. In this situation, the WIC can perform the function of interrupt masking while the NVIC and the

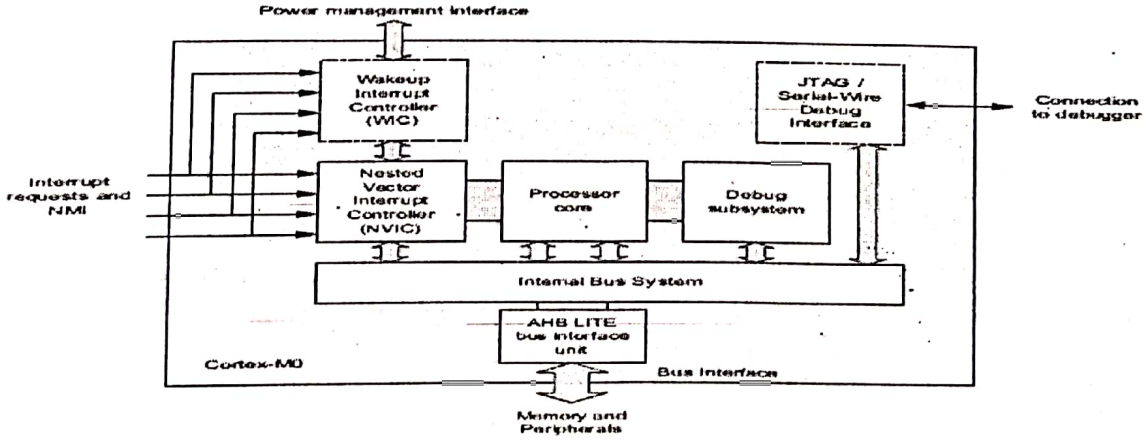


Figure 2.1:
Simplified block diagram of the Cortex-M0 processor.

processor core are inactive. When an interrupt request is detected, the WIC informs the power management to power up the system so that the NVIC and the processor core can then handle the rest of the interrupt processing.

The debug subsystem contains various functional blocks to handle debug control, program breakpoints, and data watchpoints. When a debug event occurs, it can put the processor core in a halted state so that embedded developers can examine the status of the processor at that point.

The JTAG or serial wire interface units provide access to the bus system and debugging functionalities. The JTAG protocol is a popular five-pin communication protocol commonly used for testing. The serial wire protocol is a newer communication protocol that only requires two wires, but it can handle the same debug functionalities as JTAG.

The internal bus system, the data path in the processor core, and the AHB LITE bus interface are all 32 bits wide. AHB-Lite is an on-chip bus protocol used in many ARM processors. This bus protocol is part of the Advanced Microcontroller Bus Architecture (AMBA) specification, a bus architecture developed by ARM that is widely used in the IC design industry.

The ARM Cortex-M0 Processor Features

The ARM Cortex-M0 processor contains many features. Some are visible system features, and others are not visible to embedded developers.

System Features

- Thumb instruction set. Highly efficient, high code density and able to execute all Thumb instructions from the ARM7TDMI processor.
- High performance. Up to 0.9 DMIPS/MHz (Dhrystone 2.1) with fast multiplier or 0.85 DMIPS/MHz with smaller multiplier.
- Built-in Nested Vectored Interrupt Controller (NVIC). This makes interrupt configuration and coding of exception handlers easy. When an interrupt request is taken, the corresponding interrupt handler is executed automatically without the need to determine the exception vector in software.
- Interrupts can have four different programmable priority levels. The NVIC automatically handles nested interrupts.
- Deterministic exception response timing. The design can be set up to respond to exceptions (e.g., interrupts) with a fixed number of cycles (constant interrupt latency arrangement) or to respond to the exception as soon as possible (minimum 16 clock cycles).
- Nonmaskable interrupt (NMI) input for safety critical systems.
- Architectural predefined memory map. The memory space of the Cortex-M0 processor is architecturally predefined to make software porting easier and to allow easier optimization of chip design. However, the arrangement is very flexible. The memory space is linear and there is no memory paging required like in a number of other processor architectures.
- Easy to use and C friendly. There are only two modes (Thread mode and Handler mode). The whole application, including exception handlers, can be written in C without any assembler.
- Built-in optional System Tick timer for OS support. A 24-bit timer with a dedicated exception type is included in the architecture, which the OS can use as a tick timer or as a general timer in other applications without an OS.

• SuperVisor Call (SVC) instruction with a dedicated SVC exception and PendSV (Pendable Supervisor service) to support various operations in an embedded OS.
• Architecturally defined sleep modes and instructions to enter sleep. The sleep features allow power consumption to be reduced dramatically. Defining sleep modes as an architectural feature makes porting of software easier because sleep is entered by a specific instruction rather than implementation defined control registers.
• Fault handling exception to catch various sources of errors in the system.

Implementation Features

• Configurable number of interrupts (1 to 32)
• Fast multiplier (single cycle) or small multiplier (for a smaller chip area and lower power, 32 cycles)
• Little endian or big endian memory support
• Optional Wakeup Interrupt Controller (WIC) to allow the processor to be powered down during sleep, while still allowing interrupt sources to wake up the system
• Very low gate count, which allows the design to be implemented in mixed signal semiconductor processes

Debug Features

• Halt mode debug. Allows the processor activity to stop completely so that register values can be accessed and modified. No overhead in code size and stack memory size.
• CoreSight technology. Allows memories and peripherals to be accessed from the debugger without halting the processor. It also allows a system-on-chip design with multiple processors to share a single debug connection.
• Supports JTAG connection and serial wire debug connections. The serial wire debug protocol can handle the same debug features as the JTAG, but it only requires two wires and is already supported by a number of debug solutions from various tools vendors.
• Configurable number of hardware breakpoints (from 0 to maximum of 4) and watchpoints (from 0 to maximum of 2). The chip manufacturer defines this during implementation.
• Breakpoint instruction support for an unlimited number of software breakpoints.
• All debug features can be omitted by chip vendors to allow minimum size implementations.

Others

• Programmer's model similar to the ARM7TDMI processor. Most existing Thumb code for
the ARM7TDMI processor can be reused. This also makes it easy for ARM7TDMI users, as there is no need to learn a new instruction set.
• Compatible with the Cortex-M1 processor. This allows users of the Cortex-M1 processor to migrate their FPGA designs to an ASICs easily.
• Forward compatibility with the ARM Cortex-M3 and Cortex-M4 processors. All instructions
supported in the Cortex-M0 processor are supported on the Cortex-M3 processor, which allows an easy upgrade path.
• Easy porting from the ARM Cortex-M3/M4. Because of the similarities between the architectures, many C applications for the Cortex-M3/M4 can be ported to the Cortex-M0 processor easily. This is great news for middleware vendors and embedded OS vendors, as it is straightforward to port their existing software products for Cortex-M3

microcontrollers to Cortex-M0 microcontrollers.

- Supported by various development suites including the ARM Keil Microcontroller Development Kit (MDK), the ARM RealView Development Suite (RVDS), the IAR C compiler, and the open source GNU C compiler, including tool chains based on gcc (e.g., CodeSourcery Gþþ development suite).
- Support of various embedded operating systems (OSs). A number of OS for the Cortex-M0 processor are available, including some free OSs. For example, the Keil MDK toolkit includes a free embedded OS called the RTX kernel. Examples of using the RTX are covered in Chapter 18.

Advantages of the Cortex-M0 Processor

With all these features on the Cortex-M0 processor, what does it really mean for an embedded developer? And why should embedded developers moved from 8-bit and 16-bit architectures?

Energy Efficiency

The most significant benefit of the Cortex-M0 processor over other 8-bit and 16-bit processors is its energy efficiency. The Cortex-M0 processor is about the same size as a typical 16-bit processor and possibly several times bigger than some of the 8-bit processors. However, it has much better performance than 16-bit and 8-bit architectures. As a result, you can put the processor into sleep mode for the majority of the time to reduce power to a minimum, yet you will still be able to get the processing task done.

For comparison, the DMIPS figures of some popular architectures are shown in Figure 2.2 and Table 2.1.

Note: You might wonder why the Dhrystone 2.1 is used for comparison while there are other well-established benchmarks like the EEMBC. However, the EEMBC has restrictions on the use of its benchmark results and therefore cannot be openly published.

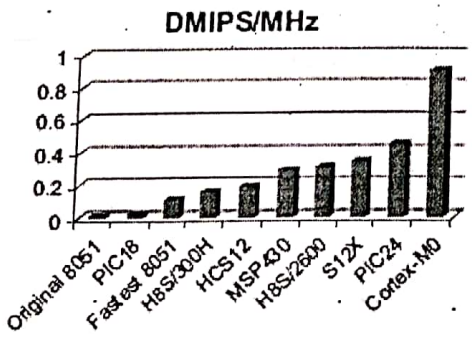


Figure 2.2:
Dhrystone comparison.

Table 2.1: Dhrystone Performance Data Based on Information Available on the Internet

Architecture	Estimated DMIPS/MHz with Dhrystone 2.1
Original 80C51	0.0094
PIC18	0.01966
Fastest 8051	0.113
H8S/300H	0.16
HCS12	0.19
MSP430	0.288
H8S/2600	0.303
S12X	0.34
PIC24	0.445
Cortex-M0	0.896 (if a small multiplier is used, the performance is 0.85)

As you can see, the Cortex-M0 processor is significantly faster than all popular 16-bit microcontrollers and eight times faster than the fastest 8051 implementation. This advantage can be used in conjunction with the sleep mode feature in the Cortex-M0 processor so that an embedded system can stay in low-power mode more often to reduce the average power consumption without losing performance. For example, Figure 2.3 illustrates that in an interrupt-driven application, the Cortex-M0 processor can have much lower average power consumption compared to 8-bit and 16-bit microcontrollers.

Although some 8-bit microcontrollers having a very low gate count, which can reduce the sleep mode current consumption, the average current consumed by the processor can be much larger than that for the Cortex-M0. The comparison is even more significant at the chip level, when including the power consumption of the memory system and the peripherals.

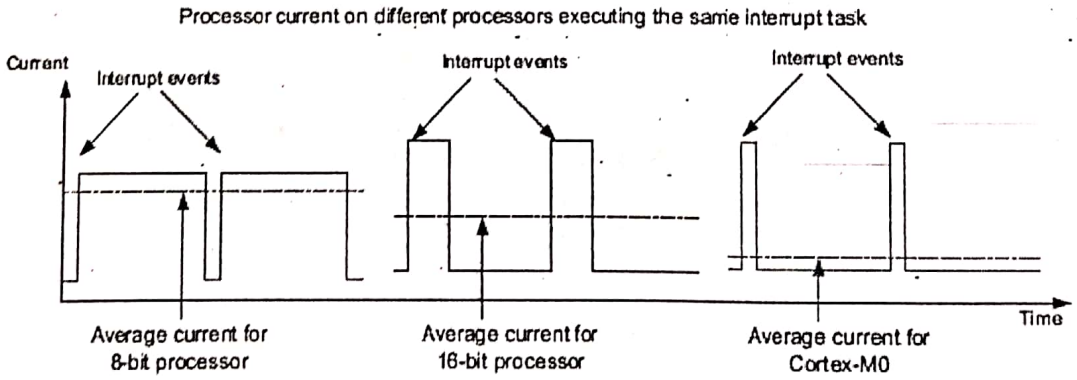


Figure 2.3:
The Cortex-M0 provides better energy saving at the same processing performance.

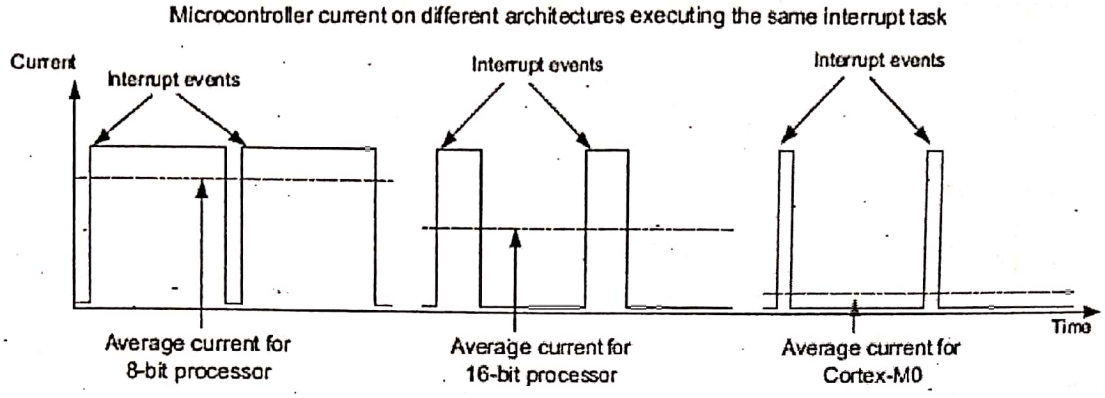


Figure 2.4:
At the chip level, the duty cycle of processor activity becomes more significant.

In a microcontroller design, the processor core only takes a small amount of the chip area, whereas a large portion of the power is consumed by other parts of the chip. As a result, the duty cycle (portion of time where the processor is active) dominates the power calculation at chip component level, as shown in Figure 2.4.

When running other applications that are not interrupt driven, the clock frequency for the Cortex-M0 processor can be reduced significantly, compared to 8-bit/16-bit processors, to lower the power consumption. Even if an 8-bit or 16-bit microcontroller has a lower operating current than the Cortex-M0 at the same clock frequency, you can still achieve lower power consumption on the Cortex-M0 by reducing the clock speed without losing the performance level compared to 8-bit/16-bit solutions (Figure 2.5).

Although other 32-bit microcontrollers are available that some of them have a higher performance than the Cortex-M0, their processor sizes are a number of times larger than the Cortex-M0 processor. As a result, the average power consumptions of these microcontrollers are higher than the Cortex-M0 microcontrollers.

Limitations in 8-Bit and 16-Bit Architectures

Another important reason to use the 32-bit Cortex-M0 processor rather than the traditional 16-bit or 8-bit architectures is that it does not have many architectural limitations found in these architectures.

The first obvious limitation of 8-bit and 16-bit architectures is memory size. Whereas program size and data RAM size can directly limit the capability of an embedded product, other less obvious limitations like stack memory size (e.g., 8051 stack is located in the internal RAM, which is limited to 256 bytes, including the register bank space) can also affect what you can

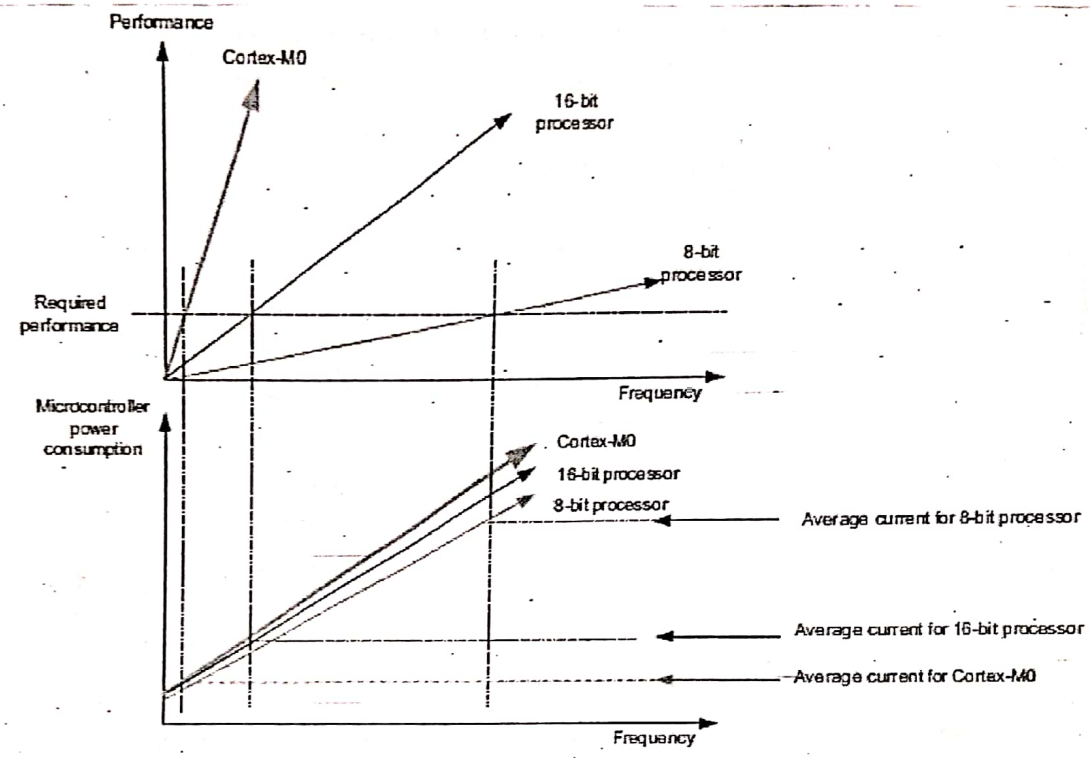


Figure 2.5:
The Cortex-M0 can provide lower power consumption by running at lower clock frequencies.

develop.With the ARM architecture, the memory space is much larger and the stack is located in system memory, making it much more flexible.

Many 8-bit and 16-bit microcontrollers allow access to a larger memory range by dividing memory space into memory pages. By doing so, development of software can become difficult because accessing addresses in a different memory page is not straightforward. It also increases code size and reduces performance because of the overhead in switching memory pages. For example, a processing task with a program size larger than one memory page might need pageswitching code to be inserted within it, or it might need to be partitioned into multiple parts. ARM microcontrollers use 32-bit linear addresses and do not require memory paging; therefore, they are easier to use and provide better efficiency.

Another limitation of 8-bit microcontroller architectures can be the limitations of their instruction sets. For example, 8051 heavily relies on the accumulator register to handle data processing and memory transfers. This increases the code size because you need to keep transferring data into the accumulator and taking it out before and after operations. For instance, when processing integer multiplications on an 8051, a lot of data transfer is required to move data in and out of the ACC (Accumulator) register and B register.

Addressing modes account for another factor that limits performance in many 8-bit and 16-bit microcontrollers. A number of addressing modes are available in the Cortex-M0, allowing better code density and making it easier to use.

The instruction set limitations on 8-bit and 16-bit architectures not only reduce the

performance of the embedded system, but they also increase code size and hence increase power consumption, as a larger program memory is required.

Easy to Use, Software Portability

When compared to other processors, including many 32-bit processors, the ARM Cortex microcontrollers are much easier to use. All the software code for the ARM Cortex microcontrollers can be written in C, allowing shorter software development time as well as improving software portability. Even if a software developer decided to use assembly code, the instruction set is fairly easy to understand. Furthermore, because the programmer's model is similar to ARM7TDMI, those who are already familiar with ARM processors will quickly become familiar with the Cortex microcontrollers.

The architecture of the Cortex-M0 also allows an embedded OS to be implemented efficiently. In complex applications, use of an embedded OS can make it easier to handle parallel tasks.

Wide Range of Choices

Because ARM operates as an intellectual property (IP) supplier, and ARM processors are adopted by most of the microcontroller vendors, you can easily find the right ARM microcontroller for your application. Also, you do not need to change your development tools if you change the target microcontroller between different vendors.

Apart from the hardware, you can also find a wide range of choices of embedded OS, code libraries, development tools, and other resources. This ecosystem allows you to focus on product development and get your product ready faster.

Low-Power Applications

One of the key targets of the Cortex-M0 processor is low power. The result is that the processor consumes only 12mW/MHz with a 65nm semiconductor process or 85mW/MHz with a 180nm semiconductor process. This is very low power consumption for a 32-bit processor. How was this target achieved?

ARM put a lot of effort into various areas to ensure the Cortex-M0 processor could reach its low power-consumption target. These areas included the following:

- Small gate count
- High efficiency
- Low-power features (sleep modes)
- Logic cell enhancement

Let us take a look at these areas one by one.

Small Gate Count

The Cortex-M0 processor's small gate count characteristic directly reduces the active current and leakage current of the processor. During the development of the Cortex-M0 processor, various design techniques and optimizations were used tomake the circuit size as small as possible. Each part of the design was carefully developed and reviewed to ensure that the circuit size is small (it is a bit like writing an application programin assembly to achieve the best optimization). This allows the gate count to be 12k gates at minimum configuration. Typically, the gate count could be 17k to 25k gates when including more features. This is about

the same size or smaller than typical 16-bit microprocessors, with more than double the system performance.

High Efficiency

By having a highly efficient architecture, embedded system designers can develop their product so that it has a lower clock frequency while still being able to provide the required performance, reducing the active current of the product. With a performance of 0.9DMIPS/MHz, despite not being very high compared with some modern 32-bit processors, the Dhrystone benchmark result of Cortex-M0 is still higher than the older generation of 32-bit desktop processors like the 80486DX2 (0.81DMIPS/MHz), and it is a lot smaller. The high efficiency of the Cortex-M0 processor is mostly due to the efficiency of the Thumb instruction set, as well as highly optimized hardware implementation.

Low-Power Features

The Cortex-M0 processors have a number of low-power features that allow embedded product developers to reduce the product's power consumption. First, the processor provides two sleep modes and they can be entered easily with "Wait-for-Interrupt" (WFI) or "Waitfor- Event" (WFE) instructions. The power management unit on the chip can use the sleep status of the processor to reduce the power consumption of the system. The Cortex-M0 processor also provides a "Sleep-on-Exit" feature, which causes the processor to run only when an interrupt service is required. In addition, the Cortex-M0 processor has been carefully developed so that some parts of the processor, like the debug system, can be switched off when not required.

Apart from these normal sleep features, the Cortex-M0 processor also supports a unique feature called theWakeup Interrupt Controller (WIC). This allows the processor to be powered down while still allowing interrupt events to power up the system and resume operation almost instantaneously when required. This greatly reduces the leakage current (static power consumption) of the system during sleep.

Logic Cell Enhancement

In recent years, there have been enhancements in logic cell designs. Apart from pushing logic gate designs to smaller transistor sizes, the Physical IP (intellectual property) division in ARM has also been working hard to find innovative ways to reduce power consumption in embedded systems. One of the major developments is the introduction of the Ultra Low Leakage (ULL) logic cell library. The first ULL cell library has been developed with a 0.18um process. Apart from reducing the leakage current, the new cell library also supports special state retention cells that can hold state information while the rest of the system is powered down. ARM also works with leading EDA tools vendors to allow chip vendors to make use of these new technologies in their chip designs.

Cortex-M0 Software Portability

The Cortex-M0 is the third processor released from the Cortex-M family. The Cortex-M processors are developed to target microcontroller products and other products that require a processor architecture that is easy to use and has flexible interrupt support. The first Cortex-M processor released was the Cortex-M3 processor, a high-performance processor with many advanced features. The second processor released was the Cortex-M1, a processor developed for FPGA applications. Despite being developed for different types of applications, they all have a

consistent architecture, similar programmer's models, and use a compatible instruction set. Both Cortex-M0 and Cortex-M1 processors are based on the ARMv6-Marchitecture. Therefore, they have exactly the same instruction set and programmer's model. However, they have different physical characteristics like instruction timing and have different system features.

The Cortex-M3 processor is based on the ARMv7-M architecture, and its Thumb-2 instruction set is a superset of the instruction set used in ARMv6-M. The programmer's model is also similar to ARMv6-M. As a result, software developed for the Cortex-M0 can run on the Cortex-M3 processor without changes (Figure 2.6).

The similarity between the Cortex-M processors provides various benefits. First, it provides better software portability. In most cases, C programs can be transferred between these processors without changes. And binary images from Cortex-M0 or Cortex-M1 processors can run on a Cortex-M3 processor because of its upward compatibility.

The second benefit is that the similarities between Cortex-M processors allow development tool chains to support multiple processors easily. Apart from similarities on the instruction set and programmer's model, the debug architecture is also similar.

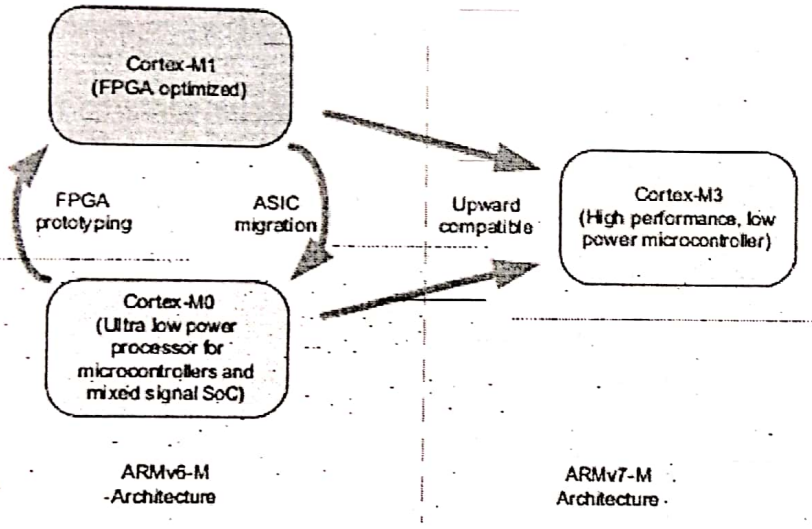


Figure 2.6:
Cortex-M0 compatibility.

The consistency of instruction set and programmer's model also make it easier for embedded programmers to migrate between different products and projects without facing a sharp learning curve.