



-IDE WORLD'S MDSL POPULAR 3 2 - BIT EMBEDDED PROCESSOR

- HHCHITECTURE 111 HSSEMBLV

In this chapter, you will learn

o The history of the ARM processor

O The features and architecture of ARM

o Hie instruction set of ARM

o Assembly language programming for

ARM

o The addressing modes of ARM

O How to use subroutines without a stack o How to generate 32-bit constants using

the rotation scheme O The concept of literal pools O I low to access R/W and Read Only memory o The use of different types of stacks

Introduction

This chapter gives an introduction to ARM, the very popular 32-bit processor, with a short account of its history, followed by details of where it stands in the embedded processor market now. ARM stands for'Advanced RISC Machine'. The name explicitly states its characteristic of being a RISC processor. The first ARM processor actually was meant to be the 'Acorn RISC Machine' as it was manufactured by Acorn Computers Ltd., Cambridge, England, in 1985.

10.1 History of the ARM Processor

In 1985, Acorn Computers Ltd. was in search of a new processor to put up in the desktop market. While the technocrats were contemplating various design options, they came across a few papers published by a set of students in the University of Berkley (USA) outlining a very simple processor design based on RISC principles.The computer architects of Acorn Computers found the design very attractive and decided to build

Chapter-opening image: An ARM7 LPC2140 board.

a new processor using some of these principles. This led to the development of ARMl, which had less than 25,000 transistors, and operated at 6 MHz.

This was followed by ARM2 (in 1987) with 30,000 transistors. Comparing this to an Intel/Motorola's processor of that time having 70,000 transistors, this was a beauty in terms of a smaller die size and lower power dissipation. This was thus, the first ARM processor which was produced in bulk. It had a 32-bit data bus, a 26-bit address space and sixteen 32-bit registers and was clocked at 8 to 12 MHz. It dissipated much less power, and performed much better than Intel's 80286 which came up around the same time (but focused on the desktop market).

ARM3, ARM4 and ARM5 were also designed, but never produced, because around this rime, in 1990, Acorn Computers teamed up with Apple Computers and VLSI Technology group to form a company named Advanced RISC Machines Ltd.This com­pany continued with ARM6, ARM7, etc. Hie latter was the processor which became very popular and led to ARM being used in exotic products such as mobile phones, PDAs, IPods, computer hard disks, etc. After this, ARM made rapid strides in the 32-bit embedded market, accounting for a very high percentage of applications in the high-end embedded systems market.

As of 2011, ARM processors account for approximately 90 per cent of all embedded 32-bit RISC processors. ARM processors are used extensively in consumer electronics, including PDAs, mobile phones, digital media and music players, handheld game con­soles, calculators and computer peripherals such as hard drives and routers, etc.

The subsequent and more advanced processors of the ARM family (ARM9, ARM 10, ARMll, Cortex) have been built on the success of the ARM7 processor, which is still the most popular and widely used member of the ARM family.

Over the years, many advanced features have been added to the ARM processor, but the core has remained more or less the same.

10.1.1 The ARM Core

What is meant by the core'? The core is the 'processing unit'or the 'computing engine' which has all the computing power, and this aspect is decided by the architecture, which represents the basic design of the processor.

One special and unique feature of ARM as a company is that it designs the core and licenses this IP (Intellectual Property) to others. This simply means that the company does not 'fabricate'the chip, but sells only the design. This design is taken by the licensee, who may or may not add more features (usually peripherals) to the design. Sometimes the buyer can also modify the basic design to a minor extent. The buyer company fabri­cates the design and sells it/uses it for its products.

There are various ways in which ARM sells its IP. It could be in the form of a soft IP. In this case, the design is sold as RTL (VITDL/Verilog code), and this allows the buyer to modify the design to a certain extent. If the design is sold as a hard IP, it means the buyer gets only the layout or the net list (connection of nets or electronic wires). Tlius, the buyer can add only peripherals to the 'black box'design he has purchased.

We can thus understand that ARM the company does not 'fabricate' ARM chips. (In contrast, Intel fabricates its processors and sells them as chips.) It is because of this, that we have ARM chips and boards of various companies—Samsung, Philips, Atmel, Texas Instruments, ST Microelectronics and so on—the list is very long.

10.1.2 The ARM Microcontroller

ARM has been designated as a 'microprocessor' and indeed it is a processor which has very high computing capabilities. It has a rich set of features for handling complex computations.

I Iowcver, for using it as an embedded processor, it needs many more capabilities and these come in the forms of on-chip peripherals. To the ARM core, peripherals are added and thus it becomes a 'microcontroller'or an MCU (microcontroller unit), rather than an MPU (micro processor unit). Figure 10.1 shows the ARM MCU. The number and kind of peripherals added, depends on the requirements of the buyer of the IR It is because of this that we have varying number of peripherals for ARM processors sup­plied by different companies. It could be obvious that to support more peripherals, the core has to be more powerful. That is why we generally find more peripherals around an ARM 9 core rather than around an ARM7 core. But as a rule, users have to spell out their requirements for the peripherals of an MCU.

When a chip has the core and the necessary peripherals to perform as a system, it is called a System on Chip (SoC)—and the term 'ARM SoC'is a very commonly used— understandably it has some version of the ARM core and a large set of peripherals.

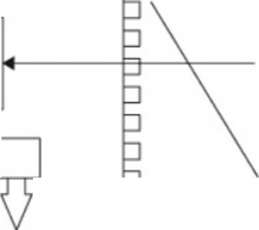
10.1.3 RISC vs CISC

The differences between these two schools of thought in computer architecture have been discussed in Section 0.3.

But to put the idea in a proper perspective in the context of ARM, some specific features of RISC arc listed herein. These apply to most of the instructions of ARM, but not necessarily to all.

1. Instructions are of the same size, that is, 32 bits
2. Instructions are executed in one cycle
3. Only the load and store instructions access memory

**nnnnnnnnnnnnnnnnn**



Developed by ARM

Chip developed by licensees and chip manufacturers

ARM Core

Internal Bus

5

|  |  |  |
| --- | --- | --- |
| Peripherals | r-- + | Memory |
| • |  |  |
| Clock |  | GPIO |
|  |

c c c c c c c c c c c c c

(= ----I

uuuuuuuuuuuuuuuuu

Figure 10.1 | ARM SoC—core with peripherals

Due to these simple guidelines in the design of the ISA (Instruction Set Architecture), the outstanding features of this RISC processor arc as follows:

* 1. Hie number of transistors needed is much less than that of a CISC processor of comparable computational power.
  2. The die size is less because of the reduced hardware involved.

iii) Due to these aspects (and a few others, which will soon be elaborated), power dis­sipation is very low.

10.1.4 Advanced Features

Once the basic ARM core was designed, later members of the family kept on having more and more features added. Over the years, some of these became 'standard' and some arc still optional. To specify what features arc available with a particular ARM core, naming conventions were adopted, but which have had to be changed over the years. Let us take a look into some of these features. But if reading this section seems cumbersome, you can skip it now, but make sure you read it later.

* + 1. Thumb: A new 16-bit instruction set called 'Thumb'was made available.The logic of having this less powerful instruction set is that all applications do not need the full power of 32-bit ARM instructions. For such cases, the 16-bit Thumb set (which is a compressed form of the ARM instruction set) will be enough and the advantage obtained is that of high 'code density'.

But what is code density?

The higher the amount of code that can be contained in unit area of memory, the higher is the code density. Thus, when available memory is limited, it may be suf­ficient to use Thumb instructions, if the application is light. There is also present, the facility for mixing ARM and THUMB instructions, this is called ARM THUMB interworking'.

* + 1. MMU and MPU: These are two aspects related to memory. One is the 'memory management unit' and the other is the 'memory protection unit'. Such units arc mandatorily available in all advanced desktop processors (like Pentium), but for embedded systems, the necessity of such units is dictated by the product for which the processor is to be used. Ill us, we have some ARM processors with both MPU and MMU, and others with one or neither of them.
    2. Cache: The first ARM processor with a cache was ARM3. It had an on-chip cache of 4 KB. ARM 7 had a cache of 8 KB which was improved in ways other than just the size. Current ARM processors have cache as a standard component.
    3. Debug interface: There is an on chip unit for testing called the JTAG interface. JTAG stands for 'Joint Test Action Group' and defines a set of standards for testing the functionality of hardware. For any chip/systcm there is a set of scan cells loeated at the boundaries and there are specific signals designed to enable ctesting' of the device. Such a unit is called the JTAG debug interface, and some ARM chips have this facility.
    4. Embedded ICE macrocelhThe current hardware trend is to design a system as'mac- rocclls', which is a hardware unit.Hie ARM core could be considered as a macrocell, while other units (peripheral units as well) may also be added as 'macrocells'. Some processors have an embedded ICE (In Circuit Emulator) macrocell to enable test­ing. This unit is powered by breakpoint and watch point registers and control and status registers. All this together can work to halt the ARM core to read status and thus do active debugging.
    5. Fast muliplier: Even though ARM is a RISC processor, there are many features in it which do not conform exactly to the RISC philosophy. Having dedicated hardware for complex operations is one such deviation. Multiplication is a complex operation, and for fast multiplication, there may be a fast multiplier unit.
    6. Enhanced instructions: Most advanced embedded systems require DSP opera­tions, and for that a DSP unit with complex arithmetic operations, may be made available on the chip.
       1. Jazelle DBX (Direct Bytecode eXecution): allows some ARM processors to exe­cute Java bytecode in hardware as a third execution state along with the existing ARM and Thumb mode. This is useful to increase the execution speed of Java ME games and applications. ARM claims that such Java applications get run in hard­ware (rather than software) so that 'more speed'is achieved.
       2. Vector floating point unit: This implies hardware support for floating point computation.

ix) Synthesizable: If an ARM processor is synthesizable, it means that its RTL code is available with the licensee, using which extensions and modifications are possible to the basic core.

Sec TablelO.l which summarizes the early naming conventions of ARM processors. ( Ihe { } notation means 'optional').

From Table 10.1, let's try to decipher what ARM7TDMI indicates. It is based on the ARM7 core, and has the Thumb instruction set (T), JTAG debugger (D), fast multiplier (M) and the embedded ICE macrocell(I). If the designation is ARM7TDMI-S, it means it is synthesizable. (Design available as VI IDL/Verilog code.) Figure 10.2 shows two ARM cores which uses this naming convention.

Table 10.1 | Early Naming Conventions for ARM

ARM {xHyHzHTHDHMHIHEHJHFHHHS}

x Family (7,8,9,10,11,...)

y Memory management/protection unit

z Cache

T Thumb 16-bit decoder

D JTAG debug

M Fast Multiplier

I EmbeddedlCE macrocell

E DSP Enhanced instructions (assumes TDMI)

J Jazelle

F Vector Floating-point Unit

S Synthesizable Version

ARM7TDMI Embedded ICE-RT ETM7 Interface ARM V4T ARM-7 Core Thumb

ARM920T MMU Dual 16K Caches Embedded ICE ETM9 Interface ARM V4T ARM-9 Core

Thumb ASB Interface

Figure 10.2 | Two ARM cores

Subsequently, it was decided to do away with these complex naming schcmcs, as the features corresponding toTDMI were expected to be mandatorily available in all ARM processors. But some numbers were added to imply the presence of memory interfaces, cache, tightly coupled memory and so on. For example, ARM with cache and MMU are now given the suffix 26 or 36, whereas processors with MPUs are suffixed with 46. Over the years, this type of naming convention has also changed. Refer to Table 10.2 for some more variants of ARM.

1. Architecture Versions

Over the years, the architectural features have also been enhanced. Ihus, later versions of the architecture are more powerful Versions v4 and v4T are the early versions, later versions arc v5, v5E, v6 and v7. Table 10.3 lists various architecture variants of ARM.

1. ARM CORTEX

ARM has come a long way from ARM2, which was the first one to be commercially produced. ARM7 was a resounding success which made ARM the dominant player in the 32-bit embedded processor market. ARM7 was followed by ARM9, ARM10 and ARMll, all of which boasted of more and more computing powers. Ihe latest in the sequence is the CORTEX series which has the architecture v7 version. To make this series cater to well-defined application sets, the following three profiles have been defined:

i) The A profile: This profile which has the ARMv7-A architecture is meant for high end applications. It is meant to handle complex applications with high-end embedded operating systems, and typical applications requiring such a profile are mobile phones and video systems.

Table 10.2 | Variants of the ARM Processor

|  |  |  |  |
| --- | --- | --- | --- |
| Processor | Architecture | Memory Management | Other |
| Name | Version | Features | Features |
| ARM7TDMI | ARMv4T |  |  |
| ARM7TDMI-S | ARMv4T |  |  |
| ARM7EJ-S | ARMv5E |  | DSR Jazelle |
| ARM920T | ARMV4T | MMU |  |
| ARM922T | ARMv4T | MMU |  |
| ARM926EJ-S | ARMv5E | MMU | DSP, Jazelle |
| ARM946E-S | ARMv5E | MPU | DSP |
| ARM966E-S | ARMv5E | DSP |  |
| ARM968E-S | ARMv5E |  | DMA, DSP |
| ARM966HS | ARMv5E | MPU (optional) | DSP |
| ARM1020E | ARMv5E | MMU | DSP |
| ARM1022E | ARMvSE | MMU | DSP |
| ARM1026EJ-S | ARMv5E | MMU or MPU | DSP, Jazelle |
| ARM1136J(F)-S | ARMv6 | MMU | DSP, Jazelle |
| ARM1176JZ(F)-S | ARMv6 | MMU+TrustZone | DSP, Jazelle |
| ARM11 MPCore | ARMv6 | MMU+Multiprocessor | DSP, Jazelle |
|  |  | Cache Support |  |
| ARM1156T2(F)-S | ARMv6 | MPU | DSP |
| Cortex-MO | ARMv6-M |  | NVIC |
| Cortex-Mi | ARMv6-M | FPGATCM interface | NVIC |
| Cortex-M3 | ARMv7-M | MPU (optional) | NVIC |

(Courtesy: The Definitive Guide to ARM Cortex-M3 by Joseph Liu, Newnes Publications)

Table 10.3 | Features of the Architecture Variants of ARM

Architecture Features Versions

v4 ARM instructions only

v4T THUMB instructions also added

v5 More advanced ARM and THUMB instructions

v5E Advanced ARM instructions and enhanced DSP instructions

v6 Advanced ARM and THUMB. SIMD and memory support

instructions added

v7 THUMB-2 technology, in which both 16-bit and 32-

bit instructions are supported, and there is no need to switching between ARM and THUMB instruction sets

* 1. The R profile: This profile which has the ARMv7-R architecture has been designed for high-end applications which require real-time capabilities. Typical applications are automatic braking systems and other safety critical applications.
  2. The M profile: This profile which has the ARMv7-M architecture has been designed for deeply embedded microcontroller type systems. This is to be used in industrial control applications where a large number of peripherals may have to be handled and controlled.

10.1.7 The Features of ARM Which Makes It'Special'

Now that we have done a survey of the range of ARM processors, let's discuss the

features which have made ARM a very popular processor in the high-end embedded

market.

* + 1. Data bus width: The processor has a 32-bit data bus width, which means that it can read and write 32 bits in one cycle. For high end applications, having a wide data bus corresponds to a high data bandwidth and is very important. When ARM first made its entry into the field, there were very few embedded processors which had such a wide bus width.
    2. Computational capability: The instruction set of ARM has been cleverly designed to facilitate very good computational capability. Many unique and new methods of fast computation without the necessity of extensive hardware is used.The design of the processor used the RISC approach, but over the years, this philosophy has been diluted to enable the addition of specialized hardware for computationally intensive tasks. In essence, ARM is a RISC processor which has a few CISC features as well.
    3. Low power: In the embedded field, power saving is very important, because a large number of devices operate on battery power. Designing lower power processor cores is thus a matter of high priority. I low is it that a processor is designed to have low power capability? Embedded processors operate at low clock frequencies compared to desk top processors. While 3.3GHz is commonly used in the desktop processor field, ARM operates at relatively low frequencies from 60 MHz to at the most 1 GHz.

The other techniques in low-power design are explained in Section 2.4.

* + 1. Pipelining: Pipelining is a fundamental idea in computer architecture, for increas­ing the speed of operation. The idea is to get many activities to be done in tandem, by dividing the whole instruction processing stage into sub stages. The basic task that any processor does is 'fetch, decode and execute'. In the simplest form of pipe­lining (3 stage), all the three stages are active all the time. While the first stage is fetching an instruction, the next stage, that is,, the decode stage, is busy with the decoding of the previously fetched instruction, and the execute stage is execut­ing the instruction which had been previously decoded. Thus at any time, there are three instructions simultaneously present in the pipeline, at different levels of processing.

If the processor clock frequency is f, the clock period (T) of the processor is divided by 3 to give a time of T/3 for each of the stages. In this sub-cycle (of period T/3), one instruction each is obtained as a throughput, which is essentially 3 instructions in the periodT. It means that the processing speed is multiplied by 3.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Fetch | - | Decode | —\*■ | Execute |

Figure 10.3a | A three stage pipeline

Cycle 1

Operation INSTR 1

INSTR 2

INSTR 3



Figure 10.3b | The three stage pipeline with 3 instructions in operation

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Fetch | —\*■ | Decode | —► | Execute | —► | Buffer | —► | Write |

Figure 10.4 | A five-stage pipeline

Figure 10.3a shows a three stage pipeline, while Figure 10.3b shows three instructions in the pipeline. Any instruction needs three sub cycles to come out of the pipeline, which translates to a throughput of three instructions per clock period (T).

ARM7 has a 3-stage pipeline, while ARM9 has a 5-stage pipeline with more finely quantized stages (Figure 10.4), which are 'fetch, decode, execute, buffer data and write back'. As a general rule, more advanced processors have more pipeline stages for example. ARM10 has 6 stages.

Pipelining is a great idea, but it has the drawback that when a branch instruc­tion appears, the instructions following it are no longer needed to be executed in the normal sequence. So the instructions in the previous stage/stages have to be discarded, or we say that the pipeline is to be flushed. This creates a loss of speed, and the penalty is higher for pipelines with more number of stages.

* + 1. Multiple register instructions: Since ARM is a RISC processor, it has instruc­tions which process data which are in registers only — this simply means that data processing instructions do not use of addressing modes in which one operand is in memory. But there are instructions which access memory and load data into mul­tiple registers - also, contents of multiple registers can be stored in memory, with a single instruction.
    2. DSP enhancements: Our processor has RISC as its basic policy, but the more advanced members of the family have DSP (Digital Signal Processing) instructions as an enhanced feature. This is where ARM departs from its RISC philosophy, but is necessary for surviving in the embedded market. These DSP enhancements are signified by an 'E'in the name as of the ARMv5TE and ARMv5TEJ architectures.

10.2 ARM Architecture

With this background, let us get started on the more intricate details of the processor.

1. Instruction Set Architecture

It is likely that you have heard the term 'Instruction Set Architecture'(ISA) mentioned in some context or the other. The term implies the user's i.e. the programmer's view of the processor, which constitute the instruction set, addressing modes, registers, etc. ISA is the assembly programmer's or compiler designer's view of the processor. We will base most of our discussions on ARM7 which was the first and still the most popular of the ARM processors. Advanced versions may have more enhancements, but the basic archi­tecture is more or less the same.

1. Operating Modes

ARM has seven operating modes which arc listed here. It is not important to understand the exact functions of each mode right now. But keep in mind that the user mode cor­responds to the simplest mode, with least privileges, but is the mode under which most application programs run. The system mode is a highly privileged mode. This mode is used by operating systems to manipulate and control the activities of the processor. The other modes arc entered on the occurrence of exceptions or rather, they arc interrupt modes. See the list of the operating modes of ARM.

* 1. User: Unprivileged mode under which most tasks run
  2. FIQ\_(Fast Interrupt Request): Entered on a high priority (fast) interrupt request
  3. IRQ\_(Interrupt Request): Entered on a low priority interrupt request
  4. Supervisor: Entered on reset and when a software interrupt instruction (SWI) is executed
  5. Abort: Used to handle memory access violations
  6. Undef: Used to handle undefined instructions
  7. System: Privileged mode using the same registers as user mode

1. Register Set

ARM has 37 registers each of which is 32 bits long. They are listed as follows:

* 1. 1 dedicated program counter (PC)
  2. 1 dedicated current program status register (CPSR)
  3. 5 dedicated saved program status registers (SPSR)
  4. 30 general purpose registers

Now, let's go into the details of the listed registers 10.2.3.1 General Purpose Registers

There are 30 of them, but they are distributed among different modes.

To understand this feature, see the case of one particular mode, say the user mode. In this mode, the registers act as shown in Table 10.4.

ARM—THE WORLD'S MOST POPULAR 32-BIT EMBEDDED PROCESSOR Table 10.4 | Registers in the User Mode

Register Numbers

R0-R12 R13 R14 R15

Designations

General purpose registers Stack pointer (SP) Link register (LR) Program counter (PC)

RO

R1

User and

System

R2

R3

R4

R5

R6

R7

Fast

Interrupt

Request

R8

R8\_FIQ

R9

R9 FIQ

R10

R10 FIQ

R11

R11\_FIQ

Interrupt Request

Abort

R12

R12 FIQ

Supervisor Undefined

|  |  |
| --- | --- |
| R13. | .IRQ |
| R14\_ | .IRQ |

R13SP

R13 FIQ

R14LR

R14 FIQ

|  |  |
| --- | --- |
| R13. | .SVC |
| R14\_ | .SVC |

|  |  |
| --- | --- |
| R13. | .Undef |
| R14\_ | .Undef |

|  |  |
| --- | --- |
| R13. | .ABT |
| R14 | \_ABT |

R15PC

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| CPSR |  | | |  | |  | |  | |  |
| - |  | SPSR\_FIQ |  | SPSRJRQ |  | SPSR\_SVC |  | SPSRJJndef |  | SPSR\_ABT |

Figure 10.5 | Register set of ARM

Figure 10.5 shows the whole set of registers available for the processor. Look at the set of registers titled as 'user and system'. Let's discuss the specific functions of each of them.

R0-R12 arc general purpose registers, or what may be designated as scratch pad' registers. These are the registers into which data and address are loaded. They are also 'the' registers used in computations.

R13 is the pointer to the stack, and is the stack pointer (SP).

R15 acts as the program counter (PC), which, like in any other processor, is the register which sequences instructions as they arc fetched from memory.

Rl4 is the link register (LR), a special register. It is used whether there is a procedure call or an interrupt, that is, branching to a location. When branching becomes necessary, the value of PC is saved in the link register, and PC takes on the new branch address. When returning to the original sequence, the PC value can be retrieved from the link register. This is a very convenient option, because the necessity to push the PC value to the stack is avoided. The stack is a memory area, and saving and retrieving from stack is time consuming. Having such a register, that is, the LR, to store return addresses helps to reduce the delay associated with procedure calls and interrupts.

10.2.4 Mode Switching

We know that there are seven modes for the processor, which implies that it can be switched to different modes, as decided by the requirement. When the processor switches, say, from the user to another mode, some of the user mode registers are replaced by another set of registers. See the FIQ\_mode, for example, in this mode, R8 to Rl4 are replaced by another set of registers, and the names of these registers arc suffixed by FIQj like Rl4\_FIQtRl2\_FIQ\_and so on.

Why Is it thatFIQ uses another set of registers?

Note that this mode is entered on a 'fast interrupt'which means it requires fast action. One action during interrupts would be to save the contents of the currently used rcgis- ters.This 'saving'takes some time.To ensure fast operation, in the case of being switched to the FIQ^inode, new registers are used. No time is spent on saving the contents of register R8 to Rl4 of the user mode. Once the FIQjnode is entered, those registers are just swapped out, and replaced by a set of new registers. Note also, that all registers are not swapped out, however.

Now look at Figure 10.5 once again to note the IRQjnode. Here only R13 and R14 are replaced by new registers. In the IRQ\_mode, the response is not expected to be, as fast as in the FIQjnode. Thus, there is sufficient time to allow the contents of most of the registers to be saved, before mode switching is done. This also applies to the modes 'undef, supervisor and abort'. In these modes too, only two registers are swapped out and replaced with new ones.

CPSR

The CPSR (Current Program Status Register) is a very important register, and there is only one such register for the processor. Figure 10.6 and Table 10.5 gives its details.

Ihe CPSR contains the information about the current state of the processor. It has bits which specify the mode, control bits to enable/disable interrupts, and also specifies whether the Thumb or ARM mode is currently in use.

31 28 27 24 23 16 15 8 7 6 5 4 0

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| N Z C V  I I I | Q |  | J | Undefined | Undefined | 1 F I | T | mode i i I i |
|  | | | |  |  |  | | |

Figure 10.6 | Current Program Status Register (CPSR) bit configuration

Table 10.5 | CPSR Bits

Bit Nos.

0 to4

Notation

Mode

Interpretation

Specifies the current mode of operation

5

T

F

6

7

Specifies whether in ARM(T =) or Thumb(T = 0) state Disables (F = 1)FIQ Disables (I = 1)IRQ

Undefined J

Q

V,C, Z, N

8 to 23,25 to 26

24 27

28 to 31

In Jazelle state (J= 1) Sticky overflow flag Conditional flags

Bits 0 to 4 specify the current mode of operation. Since there are only 7 modes of operation, only seven mode numbers arc valid.

The J bit is for indicating whether the Jazelle state is valid or not. The T bit specifies whether the current operation is in the ARM or Thumb mode.

The contents of this register can be modified only in the highly privileged system mode. It also contains the condition flag bits. Most of you are likely to know the rel­evance of the conditional flag bits. But for those who might be new to the concept of flags, here is a concise description.

10.2.5 Conditional Flags

N: Negative Flag This flag indicates the status of the MSB of the result of an opera­tion. If we arc dealing with signed number N = 1 means that the sign bit = 1, which is a negative result.

C: Carry Flag This bit is set if there is an overflow from the MSB of the data being manipulated; this can happen in additions, shifts, rotates etc. It is also set when the result of subtraction is positive. If R1-R2 gives a positive result, C = 1, indicates that Rl is greater than R2.To be precise, let's say that A carry occurs if the result of an add, sub­tract or compare is greater than or equal to 2or as the result of an inline barrel shifter operation in a move or logical instruction'.

Z:Zero Flag If the result of an arithmetic or logical operation is zero, then Z = 1.

V: Overflow Flag This is the overflow flag, which is relevant only for signed operations. It indicates that the sign bit has possibly been corrupted bccausc the result has gone out of the range.

When signed numbers are used, only 31 bits are available for the magnitude of the numbers. With 32 bits, overflow occurs if the result of an add, subtract or compare is greater than or equal to (231-1) or less than — 2J1,which is the maximum range available for signed numbers.

To cite an example, say two positive numbers are added, and the magnitude of the sum becomes greater than 31 bits. There will be an overflow into the sign bit, which will change the MSB to T'and get wrongly interpreted as a negative number. This overflow into the sign bit (MSB) with no overflow out of the MSB causes the overflow (V) bit to be set.

0: Sticky Overflow Flag This flag indicates overflow itself, but it is 'sticky' in the sense that it remains set until explicitly cleared.

Saved Program Status Registers (SPSR) There are five 'Saved Program Status Registers', that is, one for each of the'exception'modes of operation. When an exception, that is, an interrupt occurs, the corresponding SPSR saves the current CPSR value into it (so as to be able to retrieve it on returning to the previous mode). Hie system mode and user modes do not have SPSRs because they arc not entered through the mechanism of interrupts.

10.3 Interrupt Vector Table

We have seen that ARM has a number of exception modes. Exceptions are a class of interrupts which are internally generated due to the occurrence of some specific condi­tions. For example, when an undefined instruction is detected, the processor can't process it.The solution for such an undesired situation is to make the processor switch to another mode and generate an interrupt. This interrupt takes control to an interrupt service rou­tine (ISR i.e. interrupt handler) residing in a specific location in memory. This specific location is termed the 'Interrupt Vector'corresponding to this exception.

Besides 'exceptions', the processor can be interrupted by instructions and this is called a software interrupt (SWI). There are hardware interrupts as well, which are acti­vated by FICLor IRQ.

The aforesaid discussion is just to clarify the fact that associated with all exceptions, hardware and software interrupts, there is a fixed interrupt vector which leads to the ISR or the interrupt handler.

See Table 10.6 which shows the pre-defined interrupt vectors.

Table 10.6 | List of Interrupt Vectors

Exception Shorthand Vector Address

RESET

UNDEF

SWI

PABT

DABT

0x00000000 0x00000004 0x00000008 0x0000000c 0x00000010 0x00000014 0x00000018 0x0000001c

Reset

IRQ FIQ

Undefined instruction Software interrupt Prefetch abort Data abort Reserved Interrupt request Fast interrupt request

Note that the first entry in the table is 'Reset'. All processors have a address, termed 'reset vector'which is the location to which control branches to, when it is first powered on, or when reset in the midst of processor activity. For ARM, this is 0x0000 0000. Since this location is always fixed, RESET is usually included in the class of vectored interrupts.

10.4 Programming the ARM Processor

Now that we have had a look at the concepts regarding the instruction set architecture (ISA) of ARM, we are in a position to understand it better by programming. Writing, running and testing programs is the key to understanding any processor. By doing pro­gramming, we become capable of understanding almost everything about how registers, memory and flags act on data. In short, we get a total feel about the processing activity done inside the processor.

To get to this, we need a programming environment, that is, an Integrated Development Environment (IDE). There are many IDEs available for ARM, some of which are free of cost (and freely downloadable) and some of which arc proprietary and thus have to be paid for. However for students, an evaluation version is available which is freely downloadable and available from the website wvw.keil.com. I Iere, we will use the Keil IDE also called the RVDK (Real View Development Kit), which is very popular and easy to use. This version can be used for testing programs and for simulation also. We will do all our learning using this IDE. The step-by-step procedure for using this, is detailed in Appendix A. In this part of the chapter, we will assume that you have this IDE and also that you have already browsed through Appendix A.

10.4.1 Programming—Assembly vsC

Programming can be done in assembly as well in high level languages. In the embedded design world, high level languages are used in product design, and C is a very popular language. As such we will also do C programming (in the next chapter). But before that, let's have a stint in assembly programming. Our approach will be such that to understand the ARM core, that is, to use its registers, do memory access and so on, we will do assem­bly programming.This ensures that we get a good grip on the ARM core architecture. In this context, it will turn out that we focus on the computational capabilities of the core.

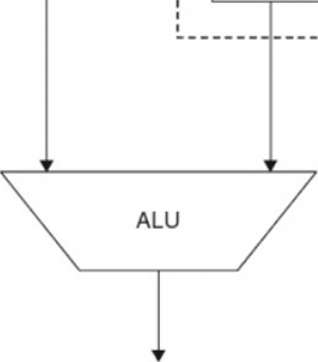
And when we start using ARM as a microcontroller, i.e. the core with a number of peripherals, we use C programming. This will allow us to use the processor in various practical applications involving peripherals and interaction with the external world.This part will be discussed in Chapter 11.

10.5 ARM Assembly Language

As mentioned earlier, the ARM instruction set has been cleverly designed to get more than one operation to be done in a single instruction. Let's list out some features of the ARM instruction set.

z

Barrel Shifter



Result

Operand 1 Operand 2

Figure 10.7 | Data processing unit

1. ARM is a RISC processor, in which every instruction has a maximum size of 32 bits. Instructions are expected to be executed in one cycle. This is true for most instruc­tions, but not for all.Therefore it is better to say that ARM is a RISC processor with a few CISC type instructions as well.
2. Another feature of RISC and therefore of ARM, is that it is a load-store architec­ture. This means that all computations are register based, that is, the operands arc to be brought to registers from memory, using a load instruction. After computation, the result is to be stored in memory. For the user, this means that there is no data processing instructions in which one of the operands is in memory. All operands are to be available in registers before computation can be done.
3. A third feature of ARM is that its ALU has a barrel shifter (Figure 10.7) associated with one of its operands. A barrel shifter is a unit that can perform more than one bit of shift/rotation, to the right or to the left on an operand. As we will soon see, the barrel shifter adds some clever processing techniques to data processing and allows shifting and an arithmetic operation to be combined in the same instruction.
4. 'Conditions'can be appended to instructions: this implies that we can choose to 'do or not do'a particular operation based on a status of a condition flag, For most other processors, only branching operations depend on flag status. Here we will see that data movement and data processing instructions can be made 'conditional'.

10.5.1 Data Types

ARM can operate on 32-bit data, which is termed a word, 16-bit data called a half word and also on byte operands. The processing tools offer the option of storing data as 'little endian', or 'big endian'. To clarify this concept, follow the forthcoming discussion, and observe Figure 10.8

|  |  |
| --- | --- |
| Address | Data |
| 0x00001200 | A3 |
| 0x00001201 | 90 |
| 0x00001202 | 47 |
| 0x00001203 | 0E |

Figure 10.8a | The little endian format

|  |  |
| --- | --- |
| Address | Data |
| 0x00001200 | 0E |
| 0x00001201 | 47 |
| 0x00001202 | 90 |
| 0x00001203 | A3 |

Figure 10.8b | The big endian format

A 32-bit data stored in memory needs 4 bytes of space which means 4 consecutive addresses are required, as one address can store only one byte. When the lowest byte of the 32-bit word is stored in the lowest of these four addresses, it is called the 'little endian'format. Otherwise, it is the 'big endian'format. See Figure 10.8. The 32-bit data word is 0 = 0xE4790A3.The storage addresses are from 0x00001200 onwards.

In the processor industry, both formats arc used. Intel prefers the little endian format, while Motorola uses the big endian format. ARM allows both formats (can be fixed up by software, in the initialization stage). In this book, we assume the little endian format.

10.5.2 Data Alignment

Storing (and loading also) of 4 bytes in memory can be done in one cycle, because the processor has a 32-bit data bus. When 32-bit data is stored in memory, four addresses are needed. But we need to specify only one address in our instruction; but there is an aspect called 'alignment'. For 32-bit data,'alignment'implies that the last two bits of this address are zero. For example, the address 0x00001200 is an aligned address. When this address is used to store 32-bit data, this address and the next three addresses arc auto­matically accessed. This is because of the way memory is organized, as four banks (see Figure 10.9).

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | Bank 3 |  | Bank 2 |  | Bank 1 |  | Bank 0 |
| 0x1204 |  | 0x1205 |  | 0x1206 |  | 0x1207 |  |
| 0x1200 |  | 0x1201 |  | 0x1202 |  | 0x1203 |  |
|  |  |  |  |  |  |  |  |

D31 D24 D23 D16 D15 D8 D7 DO

4 16 Bits 1\ [« 16 Bits ►

4 32 Bits ►

Figure 10.9 | Memory banks

If the address of a 32-bit number is given as 0x1200, the accessed addresses are 0x1200,0x1201,0x1202 and 0x1203.The 4 bytes in these addresses arc considered to be in the same row, that is, aligned. In this case, one byte each from each bank is accessed and only one memory cycle is needed to access an aligned word.

For unaligned data, one more cycle is necessary. Think of the address 0x1201. The locations to be accessed will be 0x1201,0x1202,0x1203 and 0x1204. Note that the first three bytes will be in the same row, while the last will be in a different row (bank), and so one more cycle of access will be required.

We summarize the conditions for'aligned data'as follows:

* For word (32-bit) data, the specified address should have its least significant two bits asO.
* For half word (16-bit) accesses, the specified address should have the LSB equal to 0.

Most of the tools for ARM ensure that data is stored in aligned locations, so as to avoid unnecessary extra cycles of operation.

10.5.3 Assembly Language Rules

An assembly language line has four fields, namely, label, opcode, operand and comment. A label is positioned at the left of a line and is the symbol for the memory address which stores that line of information. There are certain rules regarding labels that arc allowed under the type of assembler being used. The manual of the specific assembler should be referred, to get this clear. The second field is the opcode or instruction field. The third is the operand field, and the last is the comment field which starts with a semicolon. The use of comments is advised for making programs more readable. A typical assembly language statement is

BOSE ADD Rl, R2, R3 ; add R2 and R3 and copy the sum to Rl.

The label is BOSE, the opcode is ADD, the operands are Rl, R2 and R3 and the line after the semicolon is the comment. While writing programs, make sure you don't write instructions at the extreme left of the page—that part is the 'label' field in this book. We will use the assembler which is part of the RVDK supplied by Keil.The steps in using it have been clearly described in Appendix A. More details are available in the 'Real view assembly guide'.

10.6 ARM Instruction Set

We will now discuss the ARM instruction set, and gradually move on to writing programs.

The instruction set can be broadly classified as follows:

* 1. Data processing instructions
  2. Load store instructions—single register, multiple register
  3. Branch instructions
  4. Status register access instructions

The last set moves the contents of the CPSR or an SPSR to or from a general purpose register and are used only in privileged modes. We will discuss the first three sets in detail.

10.6.1 Data Processing Instructions

ARM is a RISC processor, one of the features of which is that it processes, i.e., performs computations, on data which arc in registers only. There are instructions which move data from one register to another. Such instructions have only two operands, that is, the source and the destination. Instructions which perform arithmctic/logical computations have three operands—two source operands and one destination operand.

**10.6.1.1** MOV and MVN

The 'MOV'instruction is a 'register to register'data movement instruction with the for­mat MOV destination, source where both the source and destination have to be registers.

The mnemonic 'MVN'stands for 'move negated'which implies moving the comple­mented value of the source to the destination.

Registers Rl to R12 can be used for data movement as they are general purpose registers. The registers Rl3, R14 and R15, which are the stack pointer, link register and the program counter respectively, can also use the MOV instructions, but this must be done carefully and only for specific purposes.

Examples

MOV R11, R2 ;copy the contents of R2 to R11 MOV R12, RIO ;copy the contents of RIO to R12 MVN RO, R9 ;move the complemented value of R9 to RO ;if R9 = OxFFFOOOOO, RO = OxOOOFFFFFF

Note Here we have discussed only the case of the MOV instruction used for moving data between registers. The MOV instruction is also used for copying immediate data into registers. That will be discussed in Section 10.17.

**10.6.1.2** The Barrel Shifter

Now, refer to Figure 10.7. We see that there is a barrel shifter associated with data processing. The figure shows two register operands, one of which can optionally be acted upon by a barrel shifter, before being admitted to the ALU. The barrel shifter can do shifting and rotation. Let us first have a general discussion on shifts and rotations.

10.6.2 Shift and Rotate

Two types of shifts are possible: logical and arithmetic. 10.6.2.1 Logical Shift Left (LSL)

CF

Register

Figure 10.10 | Logical shift left

Logical Shift Left of a (say) 32-bit number causes it to shift left, (a specified number of times) and the vacant bits on the right are filled with zeros. See Figure 10.10. The last bit shifted out from the left is copied to the carry flag. Keep in mind that a left shift by one bit position corresponds to multiplication by 2. An LSL of 5 implies multiplication by 32.

1. **|** Logical Shift Right (LSR)

Logical Shift Right does a similar thing. The vacant bit positions on the left are filled with zeros, and the last bit shifted out is retained in the carry flag. This is shown in Figure 10.11. Shifting right by one, divides the number by 2. Two right shifts cause a division by 4.

1. Arithmetic Shift Right (ASR)

Arithmetic Shift Right is different in the sense that the vacant bit positions on the left are filled with the MSB of the original number. See Figure 10.12. This type of shift has the function of doing 'sign extension'of data, bccausc for positive numbers the MSB is 0, and for negative numbers, the MSB is 1.There is no instruction for arithmetic shift left, because of not having an application for it.

1. Rotate Right (ROR)

In this, the data is moved right, and the bits shifted out from the right arc inserted back through the left. See Figure 10.13. The last bit rotated out is available in the carry flag. There is no 'rotate left\* instruction, because left rotation by n times can be achieved by rotating to the right (32 - n) times. For example, rotating 4 times to the left is achieved by rotating 32 — 4 = 28 times to the right.

1. Rotate Right Extended (RRX)

This corresponds to rotating right through the carry bit, meaning that the bit that drops off from the right side is moved to C and the carry bit enters through the left of the data. This should be obvious from Figure 10.14.

CF

Register

Figure 10.11 | Logical shift right



Figure 10.12 | Arithmetic shift right

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  | Register |  |  | CF |
|  |  | |

Figure 10.13 | Rotate right

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | |
|  | Register |  | CF |  |
|  |  |  |

Figure 10.14 | Rotate right extended

10.6.3 Format of Shift and Rotate Instructions

The number of bit positions by which shifts and rotations are to be done may be specified by a constant or may be indicated in another register.

Examples

LSL R2, #4 ;shift left logically, the content of R2 by 4 bit positions

ASR R5, #8 ;shift right arithmetically, the content of R2 by 4 bit positions

ROR Rl, R2 ;rotate the content of Rl, by the number specified in R2

Example 10.1

The content of some of the registers are given as:

Rl = 0xEF00DEl2, R2 = 0x0456123F, R5 = 4, R6 = 28. Find the result (in the destination register), when the following instructions are executed.

* 1. LSL Rl, #8
  2. ASR Rl, R5
  3. ROR R2, R6
  4. LSR R2, #5

Solution

i) Shifting Rl left 8 times causes 8 zeros in the 8 positions on the right. Rl now con­tains OxOOD El 200

* + 1. R5 contains 4. Arithmetically right shifting Rl 4 times, causes the MSB (1, for the given number) to be replicated 4 times on the left, thus causing a sign extension of the shifted number. Rl now contains OxFEFOODEl.
    2. R6 contains 28. Rotating R2 28 times to the right is equivalent to rotating it 32-28 = 4 times, to the left. After rotation, R6 contains 0x456123F0.
    3. Here, R2 is logically shifted right 5 times, and so 5 zeros enter through the left. R2 now has the value 0x0022B091.

10.6.4 Combining the Operations of Move and Shift

Recollect the barrel shifter which is an integral part of the data processing unit of the processor. This allows shifting and data processing to be done in the same instruction cycle. We will first see how moving and shifting can be combined in one instruction itself.

MOV Rl, R2, LSL #2

MOV Rl, R2, LSR R3 In both the above instructions, Rl is the destination register. In the first instruction, the source operand, that is, the content of R2 is logically shifted twice and then moved to the destination register Rl. In the second, the amount of'shifting'is specified in register R3. After the shifting is done, the result is moved to Rl.

Example 10.2

Find the content of the destination registers after the execution of each of the given instructions, given that the content of R5 = 0x72340200 and R2 = 4.

* + - 1. MOV R3, R5, LSL #3
      2. MOV R6, R5, ASR R2

Solution

The results here are similar to Example 10.1, except that the source and destination reg­isters arc not the same after execution of the instructions.

* + - * 1. MOV R3, R5, LSL #3.

The content of R5 is shifted left 3 times, and moved to R3. R3 now contains 0x72340200

* + - * 1. MOV R6, R5, ASR R2

R2 = 4, and so R5 is arithmetically shifted right 4 times. Since the MSB of the number in R5 is 0, when right shifting, this bit is replicated 4 times at the left of the number. After execution, R6 contains 0x07234020

10.7 Conditional Execution

ARM has another interesting feature which can be designated as 'conditional execution'. This means that instructions are executed only if a specified condition is true, and here the important thing is that it is not branch instructions alone that are meant—any data processing instruction can be used in this way.

In general, all arithmetic and logic instructions are expected to affect conditional flags. But for ARM, wc must suffix the instruction by S for this to happen. Otherwise the flags are unaffected. It is the S suffix on a data processing instruction that causes the flags in the CPSR to be updated.

In Example 10.2, in the instruction MOV R3, R5, LSL #3, there is a logical opera­tion involved, that is, the left shift operation.This should cause the carry flag and N flag to be set. But since the MOV instruction is not appended with the suffix, 'S', the flags remain unaffected, that is, reset.The MOV instruction can be made conditional by writ­ing it as MOVS R3, R5, LSL #3. After this is executed, wc find the N and C flags to be­set. This flag setting can be used to make an instruction following it, to be 'conditional'. We will soon see more aspects of this.

Figure 10.15 shows the format of a typical ARM instruction. In the instruction code, four bits are allotted for the condition under which the instruction is to be exe­cuted. If no condition is indicated, these bits assume the 'always'condition.

Table 10.7 lists the conditions, condition codes and the flag statuses for these condi­tions. We will discuss the use of condition codes for instructions.

Note that the conditions used for signed numbers and unsigned numbers are dif­ferent. For unsigned numbers, we use the mnemonic 'higher'or 'lower', while for signed numbers, the conditions are specified as 'greater than'or 'lower than'. The flag settings are also different. The logic of this is very simple, that is, we know that 6 is higher than 3,

31 28 27 20 19 16 15 12 11 43 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| COND | OPCODE | Fin | Rd | Other Info | Rm |

Figure 10.15 Format of a typical instruction

Table 10.7 | List of Conditions, Codes and Corresponding Flag Status

|  |  |  |  |
| --- | --- | --- | --- |
| Cond | Mnemonic | Meaning | Condition Flag State |
| 0000 | EQ | Equal | Z = 1 |
| 0001 | NE | Not Equal | z = o |
| 0010 | CS/HS | Carry set/unsigned > = | C = 1 |
| 0011 | CC/LO | Carry clear/unsigned < | C = 0 |
| 0100 | Ml | Minus/Negative | N = 1 |
| 0101 | PL | Plus/Positive or Zero | N = 0 |
| 0110 | VS | Overflow | 0=1 |
| 0111 | VC | No overflow | 0 = 0 |
| 1000 | HI | Unsigned higher | C =1&z = o |
| 1001 | LS | Unsigned lower or same | C = 01Z = 1 |
| 1010 | GE | Signed > = | N = = V |
| 1011 | LT | Signed < | N! = V |
| 1100 | GT | Signed > | Z = = 0, N = =V |
| 1101 | LE | Signed < = | Z = = 1 or N! = V |
| 1110 | AL | Always |  |
| 1111 | (NV) | Unpredictable |  |

but -3 is greater than -6. Thus, it is clear that unsigned and signed numbers have to be dealt with differently.

10.8 Arithmetic Instructions

Now let's get a feel of the arithmetic instructions of ARM and the special ways in which they can be used.

10.8.1 Addition and Subtraction

Addition and subtraction are three operand instructions.The destination is always a regis­ter. The source operands may both be registers or one of them may be an immediate data. There arc some issues in using immediate data greater than 8 bits (Ref Section 10.17).

See Table 10.8 which gives examples of how the different addition and subtraction instructions work. Any of the general purpose registers may be used as operands, though in the table, only R3, R4 and R5 have been mentioned.

Table 10.8 | List of Arithmetic Instructions

Operation

Add

Add with carry Subtract

Subtract with carry Reverse subtract Reverse subtract with carry

Instruction

ADD R3, R4, R5 ADC R3, R4, R5 SUB R3, R4, R5 SBC R3, R4, R5 RSB R3, R4, R5 RSC R3, R4, R5

Calculation

R3 = R4 + R5

R3 = R4 + R5 + C R3 = R4 - R5 R3 = R4 - R5 - C R3 = R5 - R4 R3 = R5 - R4 - C

Remember the concept of suffixing data processing instructions. This can be used ingenuously for making operations conditional. For example, the add instructions (just as any other data processing instruction) does not affect the conditional flags unless it is suffixed by S. Following such an ADD instructions, we can have instructions with conditions appended to it.lhe set of possible conditions are listed in Table. 10.7. For any instruction, the upper 4 bits arc used to specify the condition (Figure 10.15).

Consider these program lines

SUBS Rl, R2, R3 ;the suffix 'S' has been used MOVEQ\_R2, Rl ;the EQjiotation tests the Z = 1 condition

Here the move instruction is executed only if the result of the subtraction produces a zero and sets the zero flag. The condition EQjmplics the setting of the zero flag (Refer Table 10.7)). Let's use this concept in a simple example.

Example 10.3

It is required to compare two numbers which are in registers Rl and R2. The bigger number is to be placed in R10. If the two numbers are equal, then the number is to be moved to R9.

Solution

Here we use the subtraction operation to do the comparison.

SUBS R3, Rl, R2 ;R3 = Rl - R2

MOVEQ.R9, Rl ;If Rl and R2 are equal (Z = 1) move Rl to R9

MOVHI R10, Rl ;if Rl >R2, C = 1, Rl is moved to R10

MOV R10, R2 ;otherwise move R2 to R10

The salient points of this program are as follows:

First the operation, R1-R2 is performed and the result is placed in R3.

Since the SUB instruction has been appended with S, the flags will be set accordingly.

If the two numbers are equal, the zero flag gets set and the instruction MOVEQ\_ will get executed. Otherwise it becomes a NOP (no operation) instruction. Here one of the numbers (Rl) is to moved to R9 (as both numbers are equal).

The next line checks whether the carry flag has been set. If Rl >R2, the carry flag is set (C = 1) and the MOVIII (move if high) instruction gets executed. Otherwise this also becomes a NOP. The move instruction gets the bigger number into R10.

v) If the carry flag is not set, and the Z flag is also not set, it means that R2 is bigger. This is moved to RlO.

Note The last line does not need a condition. It can simply be MOV. If the other two conditions arc not satisfied it is obvious that the last one will be.

Example 10.3 might seem much too simple to need such a lot of explanation, but the intention is to make this idea (of conditional execution) very clear, so as to enable you to tackle more difficult problems with case.

One question that may come to your mind is 'why'such conditional execution?

The answer is that with this, the use of branch instructions can be avoided in many instances. This is a very great saving, as branching causes stalling of the pipeline (Section 10.2). It allows very dense code, without many branches. Not executing some of the conditional instructions does affect the speed, but the penalty is less than the over­head due to a branch.

Example 1 0.4

Find the result of the following instructions. What do these instructions accomplish?

ADD Rl, R2, R2, LSL #3

RSB R3, R3, R3, LSL #3

RSB R3, R2, R2, LSL #4

SUB RO, RO, RO, LSL #2

RSB R2, Rl, #0

Solution

ADD Rl, R2, R2, LSL #3

One source operand is R2, LSL #3. Left shifting 3 times accomplishes multiplica­tion by 23 = 8

The result of the whole operation is Rl = R2 + 8R2 - 9R2

RSB R3, R3, R3, LSL #3 R3 = 8R3 - R3 = 7 R3

RSB R3, R2, R2, LSL #4 R3 = 16R2 - R2 = 15R2

SUB RO, RO, RO LSL #2 RO = RO - 4R0 = -3R0

RSB R2, Rl, #0

We get R2 = 0 — Rl = -Rl. i.e., we get the negative value of Rl

10.9 Logical Instructions

Now, we will see the logical instructions of the processor. They also need to be suffixed with 'S'to have the flags updated. See Table 10.9.

Table 10.9 | List of Logical Instructions

Instruction Operation

AND R3, R4, R5 Logical AND of 32 bit values

ORR R3, R4, R5 Logical OR of 32 bit values

EOR R3, R4, R5 Logical XOR of 32 bit values

BIC R3, R4, R5 Logical bit clear

Logical Result

R3 = R4 AND R5 R3 = R4 OR R5 R3 = R4 XOR R5

R3 = R4 (AND NOT) R5

Example 1 0.5

Given the contents of R3 and R4 as, R3 = OxOFFOOFFO, R4 = OxOFFOOFFO. and RO = 0. Find the values in Rl, R2 and R5 at the end of the sequence of instructions shown.

EORS Rl, R3, R4

ANDS R5, R3,

Solution

The content of the destination register and the affected flag is shown alongside the executed instruction

EORSR1.R3, R4 ;R1 = 0x00000000, Z = 1

ANDS R5, R3, R0 ;R5 = 0x00000000 Z = 1

Note One of the source operands may he 8-bit immediate data as well. Refer to Section 10.17 for details of how to handle data bigger than 8 bits.

10.10 Compare Instructions

This instruction compares two operands and causes the conditional flags to be affected, but neither the destination nor the source changes. Comparison is done by a subtraction operation, and the flags are set/reset according to the result of this. (ARM has four types of compare instructions as shown in Table 10.10). However, only two flags really matter and they arc the zero flag and the carry flag. Refer to Table 10.11 to get an idea of the flag settings after a compare instruction.

Note Since the compare instructions explicitly affect the flags, the suffix S is not required for them.

Comparison is a very important operation, and we will use it very frequently. A number of programs using this instruction will be discussed subsequently.

Table 10.10 | List of'Compare'Instructions

CMP R3, R4 Compare

CMN R3, R4 Compare negated

TST R3, R4 Test

TEQ R3, R4 Test Equivalence

R3 - R4, but only flags affected R3 + R4, but only flags affected

R3 AND R4 but only flags affected R3 OR R4 but only flags affected

ARM—THE WORLD'S MOST POPULAR 32-BIT EMBEDDED PROCESSOR Table 10.11 | Flag Settings After a Compare Instruction

If C Z

R3 > R4 1 0

R3 < R4 0 0

[R3 = R4 1 1](#bookmark67)

What Is the use of the TST instruction?

TST is an instruction similar to compare, but it does ANDing and then sets conditional flags. If the result of ANDing is a zero, the Z flag is set. It can be used to verify if at least one of the bits of a data word is set or not. For that, 'test' the number with another one in which the required bit position has a '1'. For example, let's say we need to know if the LSB of the content of Rl is set or not. Use the instruction TST Rl, #01 and verify the status of the Z flag. If Z = 1, it implies that the LSB of Rl is not set, because the AND operation for that bit, has produced a 0, not a 1.

What is the use of the TEQ instruction?

TEQ\_does exclusive ORing which tests for equality. If both the operands arc equal, the Z flag is set. It verifies if the value in a register is equal to a specified number. The instruc­tion TEQ\_Rl, #45 verifies whether the content of Rl is 45.

10.11 Multiplication

Multiplication is a complex operation which needs specialized hardware and takes more than one cycle to execute. ARM has a number of multiplication instructions, which uses this hardware. Let's examine how these instructions are used.

10.11.1 Multiply

The format of the multiply instruction is MUL Rd, Rm, Rs

where Rd is the destination register. Rm and Rs are source registers. A number of points are to be kept in mind when these instructions are used.Table 10.12 lists different types of multiplication instructions.

Table 10.12 | List of Multiply Instructions

Instruction Operation Calculation

SMLAL RO, R1, R2, R3 Signed multiply and [RO, Rl] = [R0, Rl] + R2 \* R3

accumulate

SMULL RO, Rl, R2, R3 Signed multiply [RO, Rl] = R2 • R3

UMLAL RO, Rl, R2, R3 Unsigned multiply and [RO, Rl] = [RO, Rl] + R2 \* R3

accumulate

UMULL RO, Rl, R2, R3 Unsigned multiply [R0, Rl] = R2 • R3

1. The source and destination registers are 32 bits in length. If the product is longer than 32 bits, only the lower bits arc preserved in the destination register.
2. Immediate data cannot be used as a source operand.
3. If the multiplicand and multiplier are signed numbers, it is up to the programmer to identify a logic to interpret the sign of the product.
4. The instruction can be made conditional.

Example

MUL Rl, R2, R3 ;Rl = R2 x R3 MULS Rl, R2, R3 ;R1 = R2 X R3 and flags arc also set MULSEQ. R3, R2, Rl ;R1 = R2 X R3 is done only if the Z = 1

;(because of the EQ\_sutfix) ;because of the S suffix, flags are updated MULEQ\_ R4, R3, R5 ;if Z = 1, R4 = R3 x R5

1. Multiply and Accumulate

The format of this instruction is

MLA Rd, Rm, Rs, Rn ;Rd = (Rm \* Rs) + Rn

This instruction does multiplication and accumulation (addition) as seen above. All the conditions specified for the MUL instruction arc applicable here, as well.

Example

MLA RO, Rl, R2, R3 ;R0 = Rl xR2 + R3

1. Long Multiply/Long Multiply and Accumulate

In this, when 32-bit data are multiplied to get 64 bit results, the upper 32 bits are saved in a specified register. For signed data, the sign bit is also preserved in the upper register. The format is

Instruction <RdLo>, <RdIIi>, <Rm>, <Rs> Examples

Note That in all the above cases, two registers function as the destination Since multiplication is a complex instruction, it takes many cycles for execution. So, it is best to realize multiplication using shifting and adding, rather than using any of the mul­tiply instructions. Table 10.12 lists the available 'multiply and accumulate'instructions.

10.12 Division

Division is another complex instruction requiring specialized hardware and extra clock cycles. As a policy, basic ARM architecture does not have a 'divide' instruction. Division can be realized using repeated subtraction. Compilers arc given the responsibility of accomplishing division using the simple instructions of the processor.

10.13 Starting Assembly Language Programming

If you have any previous experience of assembly language programming, you will know that there are two items used therein—instructions and directives—the former are exe­cutable statements which arc 'executed' by the processor. The latter, that is, directives arc non-executable statements relating to the assembler.They are used to give the assembler necessary information to perform the assembly process smoothly. For some processors, directives are also called pseudo instructions. For ARM, pseudo instructions are special directives issued to the processor which causes certain instructions to be executed. Thus, they arc also executable statements. Thus for ARM, an assembly language line will con­tain an instruction, directive or pseudo instruction.

Writing and testing a program for ARM is done in a computer, usually a PC, which is called the host computer. The host computer should have the program development tools for ARM. Since the program written in ARM assembly language is assembled in a PC which has a different processor (usually some version of Pentium), the process is called 'cross assembly'. After the program in tested, it is converted to a hex file and burned into our processor (i.e. ARM).

Hie output of an assembly or compilation process has atleast two areas.

* 1. A code area. This is usually a read-only area.
  2. A data area. This is usually a read-write area.

The default area for code is Read-Only and for data it is Read-Write. Let's understand some fundamental directives first.

1. The AREA Directive

The first thing we do when we start assembly language programming is to define an area.There is a directive named 'AREA'for this.This directive names the area and sets its attributes. The attributes are placed after the name, separated by commas.

Example

AREA SORT, CODE, READ ONLY

AREA TABLE, DATA

The first area defined above is given the name SORT; it contains a code, and is read only.The word 'read only'is optional. The second AREA directive has the name TABLE and it contains data and though not mentioned will correspond to the Read Write area (as it is a data area).

1. The ENTRY Directive

The ENTRY directive marks the first instruction to be executed within an application. Because an application cannot have more than one entry point, the ENTRY directive can appear in only one of the source modules.

10.13.3 | The END Directive

This directive tells the assembler to stop reading. Anything written after the END directive will be ignored by the assembler. So every assembly language source module must finish with an END directive, on a line by itself.

10.14 General Structure of an Assembly Language Line

The general form of source lines in assembly language is:

{label}{instruction|directive|pseudo-instruction}{/comment}

Some points to keep in mind are listed as follows:

* 1. Instructions, pseudo-instructions and directives must be preceded by a white space, such as a space or a tab, even if there is no label. This means that they should not be written in the label space (extreme left of the line).
  2. Instruction mnemonics and register names can be written in uppercase or lowercase, but not mixed.

iii) Labels are symbols that represent addresses. The address given by a label is calcu­lated during assembly. The assembler calculates the address of a label relative to the origin of the area where the label is defined. Assigning labels eases the programmer's burden as he does not have to concern himself with numerical values. The location countcr in the assembler keeps on incrementing as labels arc cncountcrcd.

Typical assembly language lines are

NOO MOV Rl, R2, LSL #2 ;copy the content of R2 left shifted, to Rl NUMS DCW 2354,5678 jdefine two data half words

In the above two lines, NOO and NUMS arc the labels, MOV (with operands) is an instruction and DCW is a directive, which is explained in the following section.

10.14.1 Directives for Defining Data

Before we go deep into programming, we need to understand a few directives of the assembler which define and describe different kinds of data. Data which is used in a program can be bytes or words or half words. We define data and assign labels to their corresponding addresses. Defining data implies allocating space for data. The space we allocate corresponds to memory addresses, which are identified by labels. Data, when stored in memory is defined accordingly, using directives.

DCB defines data byte, DCW defines 16 bits or a half word and DCD defines a word (32 bits).

Examples

NUMS DCB 9,82,71

NUMB DCW 0x6787,0x4564

NUMBR DCD 0x00000123,0x67890900

In the above, the first line shows data which are bytes. The first byte 9, has the address NUMS, 82 has the address NUMS + 1, and 71 has the address NUMS + 1.

The sccond line has address NUMB for the first half word, and NUMB + 2 for the second half word. In the last case, the addresses of the words are NUMBR and NUMBR + 4.

Keep in mind that a byte needs only one address, half word needs two, and a word requires four memory addresses.

1. The EQU Directive

This is a frequently used directive, and is used to equate a numeric constant to a label. The constant may be data or address. Examples are as follows:

FACTR EQU 35

BASE\_ADDR EQU 0x40000000

1. Constants Allowed

The constants that can be used are numbers (decimals, hex or having any other base), characters, strings and Boolean

* Decimal., say, 346, 6748, etc.
* I Iexadecimal. For example, 0x12345678,0xFCE45, etc.
* n\_xxx where: n is a base between 2 and 9 and xxx is a number in that base
* Characters: They are to be enclosed within single quotes, 'e', 'R\ etc.
* Strings: They arc characters enclosed within double quotes "mine", "non", etc.
* Boolean: TRUE or FALSE

1. The RN Directive

The names of the general purpose registers have been introduced as R0, Rl, R2, etc.

When we use them for loading operands, it is possible that we have a confusion as to which data has been loaded into which register. To ease out this problem, there is a way for giving variable names to registers. Suppose we need to use R0 for loading the value of X, and Rl for loading Y, we use the directive RN as follows:

X RN 0 Y RN 1

This method can be used for any of the registers.

DATl RN 8 DET RN 10

10.15 Writing Assembly Programs

Now, that we have got used to writing some instructions, let's get down to writing a complete program. This will let us get a feel of the programming process, after which we can learn more important instructions and write bigger and better programs.

Example 1 0.6

|  |  |  |  |
| --- | --- | --- | --- |
| Write a program to find the | sum of 3X + 4Y + 9Z, where X = 2, Y = | 3 and Z |  |
| Solution |  |  |  |
| AREA SUMM, | CODE, READONLY |  |  |
| X RN 1 | /register Rl is | named | X |
| Y RN 2 | /register R2 is | named | Y |
| Z RN 3 | /register R3 is | named | Z |

ENTRY

into register Rl into register R2 into register R3

MOV X,#2 MOV Y,#3 MOV Z,#4

3X+4Y 3X+4Y+9Z

**ADD** Rl,Rl,Rl,LSL#1 MOV R2,R2,LSL#2 **ADD** R3,R3,R3,LSL#3 ADD Rl,Rl,R2 ADD Rl,Rl,R3

B STOP END

;load X = 2 ;load Y = 3 ;load Z = 4 ;R1 = 3X ;R2 = 4Y ;R3 = 9Z

;R1 = R1+R2 i.e. ;R1 = R1+R3 i.e.

STOP

/continue branching at STOP ;end of the assembly file

Since this is the first complete program we are writing, it is important to make some

observations regarding it.

* 1. SUMM is the name of the code AREA defined. The term 'Read only'is optional, as by default, a code area corresponds to the Read only memory only.
  2. As assembly language line has the label field at the left, and the opcode field to its right. For the Keil assembler, you may find that writing the word AREA' in the label field will generate an error message. But the directive RN is to be positioned in the label field itself. No instructions should be in the label field.
  3. The ENTRY directive should be followed by an instruction or pseudo instruction.
  4. Hie program involves multiplication and addition. Since the multiply instruction is a complex' one involving the use of special hardware (more power dissipation and more clock cycles), it is not used. Instead multiplication is achieved by the use of shift and add instructions.
  5. The last instruction is an unconditional branch instruction (mnemonic 'B') and it continually branches to the same label STOR This is done so that control does not go any instruction beyond this location. Any code has to finally be burned into ROM. Many embedded programs have their last line as this kind of self branching, since we don't want the next memory locations in code memory to be accessed.

10.16 Branch Instructions

For any processor, branching is a very important operation. The power to change the sequence of execution is obtained by branching, which may be conditional or

Table 10.13 | List of Branch Instructions

Mnemonic

B

BL BX BLX

Instruction

Branch

Branch and link Branch and Exchange Branch Exchange with link

31

28 27 26 25 24 23

COND

1 0 1

Signed-immed\_24

Figure 10.16 Format of a branch instruction

unconditional. Most processors have 'jump' and 'call\* instructions for changing the sequence of execution. ARM does all this by different forms of a 'branch' instruction. It has the mnemonic 'B'for branch. Hie four different forms of branch instruction are given in Table 10.13

Let's see the usage of each of them.

Branching implies transferring control to a new memory location which is expressed as a 'label'. Hence the format of any branch instruction is B label. Branching is made conditional by appending the mnemonic B with the necessary condition.

Examples

B NEW transfers control unconditionally to location NEW

STOP B STOP ;continually branches to its own label STOP

BNE NOO ;branch to NOO if Z flag is not set

BHI LUX ;branch if high, i.e., if C = 1

The format of a branch instruction is as shown in Figure 10.16. Target addresses are 'relative'. What this means is that when a branch instruction is taken up, the PC (program counter) value and the value specified by the instruction are algebraically added.

The target is specified as a 24-bit signed number; this number is shifted left (logically) twice (so that the two LSB bits are zero. This makes all target address to be 'aligned' (Ref Secion 10.5.2). The left shifting also multiplies the number by 4. This makes the target to have 26 bits, that is, the maximum range is between +/— 225 (one bit is for sign, remember). This number is added to the PC value. In short, what is done with the 24-bit immediate number, by the instruction is that it shifts it left by two bits, sign extends it to 32 bits, and adds it to PC.Thus, the maximum range for branching is only +/- 32 MB. (225 = 220 x25;220= 1 MB, 25 = 32). For branch addresses beyond this range, the PC can be directly loaded with the target address. Now see this simple program which calculates the factorial of 10.

; define the code area ;entry point ;R1 = 10

AREA FACTO, CODE ENTRY

MOV Rl, #10

Example 1 0.7

MOV R2, #1 ;R2 = 1

REPT MUL R2, Rl, R2 ;R2 = R2 xR2

SUBS Rl, Rl, #1 ;Rl = Rl- 1

BNE REPT ;branch to REPT if Z! = 0

STOP B STOP ;last line

END

This is a very simple program which finds the factorial of 10. It can be used to find the factorial of any other number (except 0), provided the factorial does not exceed 32 bits in size.The technique is to multiply the number with the 'number-1' recursively. Meanwhile, a counter also decrements by l(which is done by subtraction), and when the counter is 0, the Z flag is set.'Ihe multiplication is then stopped.'Ihe factorial is avail­able in the register R2. The branch instruction used is a conditional one, that is, BNE which tests the Zero flag.The instruction before it, that is, SUB has been appended with the 'S' sutfix to ensure the setting of flags.

Now let's see another example which uses conditional branching.This program per­forms division by repeated subtraction.

Example 1 0.8

AREA DIV, CODE ENTRY

MOV Rl, #500 MOV R2, #16 MOV R3, #0 MOV R4, Rl SUBS R4, R4, R2 ADDPL R3, R3, #1 BPL REPT

ADDMI R4, R4, R2

B STOP

END

;Move the dividend to Rl ;Move the divisor to R2 ;R3 = 0

REPT

STOP

;copy the dividend to R4 ; subtract and set flags ;add if N = 1 i.e. MSB of R$ is +ve ;repeat the loop if the MSB is +ve ;if MSB of R4 is -ve, add R2 to R4

This program performs division by repeated subtraction. Here 500 is to be divided by 16.

The method is to subtract 16 from 500 repeatedly until the result becomes negative. The branch instruction BPL REPT means Branch to label REPT if plus (PL), i.e., if N = 0.

Besides conditional branching, there are the ADD and SUB instructions also, which arc conditional—the condition used is the status of the sign flag N. Ihe steps of the program are as follows:

i) Subtract 16 from 500, and check if the result is +ve or —ve. This can be verified by checking the N flag which corresponds to the MSB of the resultant number. The condition flags arc updated by the subtraction operation (using the suffix S). ii) If the number (in R4) is +ve, it means that subtraction can be repeated unhindered. Each time this is verified, the quotient register (R3) is incremented by 1.

1. When the result of subtraction becomes —ve, (the condition 'Ml'for minus), add the divisor to this negative number (in R3).
2. In this problem, when 16 is subtracted 31 (0x1 F) times from 500, the value in R4 is+ve. One more subtraction makes the N flag to be set, and the number R4 to be negative.
3. To this -ve number add the divisor. This makes it equal to the remainder which is 4, in this case.
4. Thus, we get 31 (0x1 F) as the quotient (in R3) and 4 as the remainder (in R4) 10.16.1 Subroutines/Procedures

In Table 10.13, there is another form of the branch instruction which is BL standing for 'Branch and Link\*. Recollect that a procedure (also called subroutines, functions, etc.) means that a new program sequence is taken up, but control returns to the original point after that. Most processors (including ARM) use stacks to store the return addresses and return instructions to handle procedure calls. ARM has an additional feature to handle procedures in a simpler manner. Recollect a register named the 'Link Register'. When a BL instruction is encountered, the PC value is changed to that of the target, but the old PC value is copied to the LR register. At the end of the procedure, the LR value can be copied back to the PC. Now let's write a program which calls a procedure.

Examplel 0.9

Write a program to calculate 3x2 + 5Y\*, where X = 8 and Y = 5 Solution

; to calculate 3X: +5Y: ;call the SQUARE procedure ; 3X

;R2 = 5

;call the SQUARE procedure

;5Y2

;R4 = R1+R0 i.e 3X^ +5Y' ;last line in the execution

;the SQUARE procedure ;return LR back to PC

AREA PROCED,CODE ENTRY

MOV R2,#8 BL SQUARE

ADD R1,R3,R3,LSL #1 MOV R2,#5 BL SQUARE

ADD RO,R3,R3,LSL #2 ADD R4,R1,R0 STOP B STOP

SQUARE MUL R3,R2,R2 MOV PC,LR END

The salient points of this program are as follows:

i) A procedure named SQUARE has been used. This procedure uses the multiply instruction to find the square of any number. The number to be squared is passed to the procedure using the register R2.1he square of the number is returned to the main program in R3.

* 1. There are two numbers, X and Y, whose squares are to be found. Calling the pro­cedure amounts to just writing the instruction BL SQUARE. This instruction will cause a branching to the procedure named SQUARE. It also copies the current PC value to the link register (LR).
  2. The procedure has only two instructions: one to perform squaring, and the other to copy the LR content back to PC.The second instruction causes a return to the main program.
  3. We need two multiplications, in addition to the squaring operation.These two, that is, 3X2 and 5Y2 are achieved by shifting and adding.The MUL instruction is used as little as possible because it takes more time, and causes higher power dissipation.
  4. The last step is adding 3X\* and 5YJ which are now in Rl and RO.The sum is avail­able in R4.
  5. Note that the last program line to be executed is STOP B STOP, even though it is not the last line in the assembly file.

In Table 10.13 there are two more forms for the branch instruction. BX stands for Branch and Exchange. BLX is for Branch Link and Exchange. The Exchange feature is applicable when ARM and THUMB instructions are being used, and it is needed to switch from one set to another.

10.17 Loading Constants

How is it different for ARM ?

An important addressing mode for any processor is the 'immediate'mode. In this, a con­stant which is specified in the instruction itself is to be copied into a register, or is used as one operand in any arithmetic or logic operations.

Examples

MOV Rl, #0x7867 ADD Rl, R2, # 567

This seems very obvious and direct. For CISC machines, this is fine, because the imme­diate data can be another byte or word. But ARM has the limitation that its instruction size should nor exceed 32 bits, which means that the constant should fit in the word length of 32 bits along with the opcode, condition code, register code and other infor­mation that the instruction should carry. It is thus apparent that we can't have a 32-bit constant embedded in the instruction.

So then what is the maximum size of the constant that can be used in the immediate mode?

We would like to be able to use immediate constants as large as 32 bits. How is this done? ARM uses an ingenious technique, the idea being the use of rotation of a small number to generate a large number. We have already seen that there is a harrel shifter in the ALU. Any data processing instruction has a format as shown in Figure 10.17a. The data processing instruction format has 12 bits available for operand 2.

Figure 10.17b shows the instruction format which has been modified for using the immediate mode.

31 3029 28272625 242322 21 20 19 18 17 16 1514 13 12 11 109 87 6 5432 1 0

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Cond | 0 | 0 | 1 | Opcode | S | Rn | Rd | Shifter\_operand |

Figure 10.17a | Format of a typical data processing instruction

11

87

ROT

IMMED-8

x2

ROR

32-Bit Constant

Figure 10.17b

Modification of the'shifter operand'

When the immediate mode is needed, the 12-bit field is modified such that there is an 8-bit immediate constant which is subject to a ROR (rotate right operation). The rotate operator can use only 4 bits. But since the maximum rotation possible is 32 bits, the four bit 'rotate' operand is multiplied by 2 and then rotated. Hence, it becomes a ease of'8 bits shifted by an even number of bit positions'. The rotated 8-bit number will become a 32-bit number during the data processing. Let's try to understand how this is done.

10.17.1 Generating a 32-bit Constant Using Rotation

Consider the steps in rotating to the right by 2, the number OxFO after expanding it to fill 32 bits space

Case 1

The original 8-bit number is 1111 0000 Expanding it to fill 32 bits makes it

00000000 00000000 00000000 11110000 Rotating it right by 2 makes it

00000000 00000000 00000000 00111100 i.e. 0x3 C

Case 2

We can also use the MVN instruction for generating new numbers. If 0 is loaded into a register and moved into another or the same register after using the MVN instruction, we get OxFFFFFFFF

You can try this code to verify.

Table 10.14 | The Range of Constants That Can Be Generated By the Rotation Scheme

Decimal Values

0-255

256,260,264 1020

4096,4160,4224 16320 0x1000 - 0x3fc0 64

1024, 1040, 1056 4080

Equivalent Hexadecimal

O-Oxff

Ox 100 - 0x3fc

0x400 - OxffO

Step Between Values

1

4 16

Rotate

No rotate Right by 30 bits Right by 28 bits Right by 26 bits

MOV Rl, #0x0 MVN Rl, Rl

Let's summarize the points regarding the generation of constants using the ARM rotation scheme.

* + 1. A class of constants can be generated by this scheme, (Table 10.14) but all constants cannot be generated.Those that cannot be, will have to loaded directly into memory, by using the concept of'literal pools'. We will come to that soon. (Table 10.14 shows the range of constants that can be generated by the rotation scheme)
    2. To generate the constant needed, the programmer need not specify the 8-bit imme­diate number, and the number of rotations to be done. lie just has to write an instruction in the immediate mode. The assembler converts this instruction to the required scheme. When a constant 0x200000002 is needed, the assembler converts it to the instructions.

MOV Rl,#Ox22

MOV Rl, Rl, ROR#4, which creates the require 32-bit constants for us. iii) The processor does not have an instruction for rotation to the left. But, rotating n times to the left is achieved by rotating (32-n) times to the right.

Example 1 0.9

Find the 32-bit constant generated by each of the following rotations

* + - 1. Rotate 0x40, to the right 30 times
      2. Rotate 0x56, to the left 12 times
      3. Rotate 0x6D, to the right 16 times
      4. Rotate 0x05, to the right 6 times
      5. Rotate OxFC, to the right 2 times

Solution

* + - * 1. The 8-bit number is 01000000.

00000000 00000000 00000000 01000000-, the 8-bit number in 32-bit format 00000000 00000000 00000001 00000000; the number after rotation Thus, the constant obtained is 0x100

* + - * 1. The 8-bit number is 01010110

00000000 00000000 000000000 01010110-, the 8-bit number in 32-bit format. Rotating 12 times to the left is equivalent to rotating 20 times to the right 00000000 00000101 01100000 00000000; the number after rotation The constant generated is 0x56000

Table 10.15

ROR

Constant

8-Bit Number

0x6D 0x05 0x3E

16 6 2

0x6D0000

0x14000000

0x8000000F

in

IV

v

The answers for iii, iv and v are in Table 10.15

From Example 10.9, we note that many 32-bit numbers can be generated by the ARM rotation scheme, but there are constants which cannot be obtained by this method. For example, the number 0x11111111 cannot be generated by rotation.

How then are such constants obtained for use in the immediate mode of addressing? **10.17.2 Literal Pools**

In computer science, specifically in compiler and assembler design, a literal pool is a lookup table used to hold literals during assembly and execution. But first, what exactly is a literal? In programming, a literal is a value written exactly as it is meant to be interpreted. A literal can be a number, a character or a string. For example, in the expression, x = 145, x is a variable, and 145 is a literal. Thus, literals are constants for ARM. When it is required to load a constant in a register, the assembler can help by creating a space in memory and then placing this constant in the space. From this memory space, the processor can take it and use it using load instructions. But assemblers arc not guided by instructions; they use what arc called pscudo instructions. In this case of making a literal pool, and taking a constant from the pool, there is a specific pseudo

instruction

LDR Rd, = const

This pseudo instruction can construct any 32-bit numeric constant. Suppose we need to the constant 0x33333333, it is likely that we write an instructions MOV Rl, # 0x33333333. With this, the assembler will give an error message that such a constant cannot be generated.To avoid such a situation, we write

LDR Rl, = 0x33333333.

This is a pseudo instruction (don't confuse it with the LDR instruction, we will soon come to, there is a difference in format between the two). This will cause the assembler to check one of the following possibilities.

Can the constant be constructed with MOV or MVN instruction combined with rotation? If rhis is possible, the assemhler generates the appropriate instruction, that is, an 8-bit number is rotated appropriately to get the constant in question.

If the constant cannot be constructed this way, the assembler places the value in a literal pool and generates an LDR (load register) instruction with a program-relative address that reads the constant from this literal pool.

Example 10.10a

AREA PROG1,CODE,READONLY ENTRY

LDR Rl, = #0x12400000 LDR R2, = 0X00555555 ADD R3,R1,R2 STOP B STOP END

In ExamplelO.lOa, two constants are needed. If you run this program and check the disassembly file, you will find two interesting facts, which relate to the different ways in which these two constants are generated. The assembler realizes that the first constant can be obtained by the rotation scheme, but the sccond one cannot. So a literal pool is created just after the last instruction, and the constant 0x00555555 is placed therein. Then a'load register'instruction is generated to load the constant into register.

How Is the literal pool accessed?

The literal we need is accessed from the literal pool using a PC relative mode. In this mode, only 12 bits are allowed for the 'relative number'which can be positive or negative. Thus, the literal in the pool has to be within +/ 4KB of the current PC value.

Where should the literal pool be placed?

Normally the literal pool is placed just after the END directive, which means just after the end of the program area. This is okay for normal size programs. But some­times, programs arc very large and if the literal pool is placed after the end of the program, it may be out of range (of +/— 4KB) of the LDR instruction. Such a situ­ation implies that there should be the flexibility of placing literal pools anywhere in memory. This is done by the LTORG directive which allows us to define the origin of a literal pool.

When a pseudo instruction LDR Rd = const is encountered, the assembler checks if the constant is available and addressable in the nearest literal pool. If it is so, it takes it from the pool. Otherwise, it attempts to place the constant in the next literal pool. If the next literal pool is out of range, the assembler generates an error message. In this case, the LTORG directive is to be used to place an additional literal pool in the code. Place the LTORG directive after the failed LDR pseudo instruction, and within 4KB memory space.

Literal pools arc to be placed in locations where the processor docs not attempt to execute them as instructions. It is best to place them after unconditional branch instruc­tions, or after the return instruction at the end of a subroutine. Let us see an example of a case where the LTORG directive bccomcs necessary.

Examplel 0.10b

AREA PROG1,CODE,READONLY ENTRY

LDR Rl, = 0x12400000 LDR R2, = 0x00555555 ADD R3,Rl,R2 SPACE 4400 STOP B STOP END

This is a modified version of Example 10.10a. Recollect that we need to have a literal pool for the constant 0x00555555. Here, a directive called SPACE 4400 has been inserted. This directive creates an empty area of 4400 byes. Because of this, the total space occu­pied by the program becomes large (greater than 4400 bytes, anyway). Tlie literal pool is usually after the program area. In this case, this will make the literal pool to be beyond the range (greater than 4KB) of the LDR R2, = 0x00555555. Hence, on assembling, the following message is seen.

error: A12H4F.: Literal pool too distant, use LTORG to assemble it within 4KB In the program an error message indicating this will be obtained as above. To avoid such a situation, we can place the literal closer to the instruction which needs the constant. See the modified version of the program.

Example 10.10c

AREA PROG1,CODE,READONLY ENTRY

LDR Rl, = 0x12400000 LDR R2, = 0X00555555 ADD R3,R1,R2 LTORG

SPACE 4400 STOP B STOP END

Now the program runs without error, because a literal pool has been created before the 'free space'of 4400 bytes. By the use of the LTORG directive, the required constant is found to have been placed in this pool, by the assembler. Thus, we see that the directive LTORG can be used to place literal pools wherever we want. This will become useful as programs become larger.

10.18 Load and Store Instructions

ARM is a RISC architecture, and one of the features of RISC is that of being a 'load store' architecture. Loading is the process of getting data from memory into a register, and storing is just the reverse process. In ARM, data is brought into registers using a load instruction, and only then can it be used for data processing. After computation, the result can be 'stored' in memory. The memory in question is 'RAM' which is the read/write memory. RAM is volatile and is used for temporary storage of data in the course of computations. The only instructions which access RAM are 'load' and 'store\*. All registers can be accessed using these instructions, but programmers arc advised to exercise caution when accessing critical registers like the PC, SP, etc. The syntax for load or store is

LDR/STR {<cond>}<Rd>, <addressing modc>

Rd is the source register for store and destination register for load. The addressing mode gives us the necessary information to get the 'effective address', which is the actual memory address to be accessed. The addressing mode is indirect because the memory address is not to be specified directly in the instruction, rather a base register is mandatorily used. For the simplest case, an example of LOAD and STORE instructions are as follows:

LDR Rl, [R2J ;copy into Rl the content of memory specified in R2

STR Rl, fR2] ;storc the content of Rl into the memory address specified in R2

This implies that the load/store instruction must be preceded by an instruction which copies the address into R2. We will soon get to know how this is done. There are various ways of specifying the effective address. The barrel shifter can be part of the address specifying mechanism.

Example 10.11

I low is the effective memory address calculated in the following load and store instructions?

1. LDR R3, fR2, LSL #2]
2. STR R9, [Rl, R2, ROR #2]
3. LDR R4, [R3, R2J
4. STR R5, [R4, R3, ASL #4]

Solution

* 1. LDR R3, [R2, LSL #2]

In this the effective address is the content of R2 left shifted by 2, i.e. multiplied by 4

* 1. STR R9, [Rl, R2, ROR #2]

Here, the effective address is specified by Rl, R2 and a right rotation. To calculate it, the content of R2 is rotated twice by 2, and then added to the contcnt of Rl.

* + 1. LDR R4, [R3, R2]

The effective address here is the sum of R3 and R2.

* + 1. STR R5, [R4, R3, ASL #4]

The effective address is the sum of the content of R4 and the arithmetically left shifted (by 4) contcnt of R3.

10.18.1 Bytes, Half Words and Words

Now, let's see another aspect of load and store instructions. ARM has instructions to transfer specifically a word (32 bits), half word (16 bits) or a byte (8 bits) between

ARM—THE WORLD'S MOST POPULAR 32-BIT EMBEDDED PROCESSOR Table 10.16 | List of Load and Store Instructions

|  |  |  |  |
| --- | --- | --- | --- |
| LDR | Load Word | STR | Store Word |
| LDRH | Load Half Word | STRH | Store Half Word |
| LDRSH | Load Signed Half Word |  |  |
| LDRB | Load Byte | STRB | Store Byte |
| LDRSB | Load Signed Byte |  |  |

memory and registers.There are also instructions which differentiate between signed and unsigned data.

There are instructions which clearly indicate the kind of data to be moved. See Table 10.16. From the table, wc understand that we can load and store parts of a 32-bit word by using B for byte and 11 for half word, along with the load and store instructions.

If a memory location contains a 32-bit word, wc can move the LSB (assuming little endian format) into a register by using LDRB, or the lower half of the word by using LDRH. Let's clarify this by an example.

Example 10.12

Two memory areas are being referenced and two registers are used as pointers: Rl= 0x00000100 R2 = 0x40001200

Figures 10.18a and b show the data addresses and corresponding data. Show the content of memory, after the execution of the following instructions:

|  |  |
| --- | --- |
| Address | Byte Stored |
| 0x00000100 | 56 |
| 0x00000101 | 23 |
| 0x00000102 | 0D |
| 0x00000103 | AE |

Figure 10.18a | Address and data

|  |  |
| --- | --- |
| Address | Byte Stored |
| 0x40001200 | 00 |
| 0x40001201 | 00 |
| 0x40001202 | 00 |
| 0x40001203 | 00 |

Figure 10.18b Address and data

* + - 1. LDR R3, [Rl]
      2. LDRB R3, [Rl]
      3. LDRIIR3, [Rl]
      4. STRB R3, [R2J given that R3 = 0xAE0D2356 For this case, show half word and word storage as well.

Solution

* + - * 1. LDR R3,[Rl]

In this, the complete 32-bit data in the address pointed to by Rl is copicd to R3. So R3 = 0xAE0D2356

* + - * 1. LDRB R3,[RlJ

In this, the byte (LSB) of the word alone is copied to R3. Since it is an unsigned byte, the remaining bytes of R3 contain 0. So R3 = 0x00000056 Hi) LDRII R3,[Rl]

In this, the half word (lower two bytes) of the address is copied to R3. R3 = 0x00002356 iv) STRB R3,[R2] given that R3 = 0xAE0D2356

In this, the byte corresponding to the LSB of the data in R3 is copied to the address pointed by R2. See Tables 10.17a, b and c for byte, half word storage and word storage as well.

Table 10.17a, b and c

|  |  |
| --- | --- |
| STRB R3, [R2] | |
| 0x40001200 | 56 |
|  | 00 |
|  | 00 |
|  | 00 |

|  |  |
| --- | --- |
| STRH R3, [R2] | |
| 0x40001200 | 56 |
|  | 23 |
|  | 00 |
|  | 00 |

|  |  |
| --- | --- |
| STR R3, [R2] | |
| 0x40001200 | 56 |
|  | 23 |
|  | 0D |
|  | AE |

10.18.2 Loading Signed Numbers

Signed numbers are those whose MSB is the sign bit. For positive numbers, the sign bit is '0', whereas negative numbers are in the two's complement form and have their MSBs to be '1', When a 32-bit number is available in memory, it can be loaded into registers as signed bytes and signed half words. In these cases, the MSB of the byte part or the half word part is checked, and sign extension is done while loading it into registers

Consider the case of a word 0xCDEF8204 in memory. Let R7 be used as a pointer to that memory location. Then, observe the result of execution of the following instruc­tions, as given in the comments column.

LDR

LDRSH

LDRSB

Rl, [R7] ;Rl = 0xCDEF8204 Case 1

R2, R7] ;R2 = 0xFFFF8204 Case 2

R3, [R7J ;R3 = 0x00000004 Case 3

For case 1, the 32 bits are copied to Rl. For case 2,only the lower 16 bits arc to be copied. The MSB of the 16-bit half word is '1', and this is extended to 32 bits while copying to R2. That's how the upper 16 bits of R2 become FFFF.

For case 3, the lowest byte alone is copied. Its MSB is 0. As such, the rest of R3 is filled with zeros, i.e., the sign bit '0' is extended to fill the upper 24 bits. You can also observe in Table 10.16 that there arc no store instructions for signed bytes or signed half words.This is because storing simply means placing numbers in memory.These numbers may be signed, unsigned data or code—it is only when the user brings it to a register, is the processing on that number done. Only then it is necessary for that number to be interpreted as signed or unsigned.

10.18.3 Indexed Addressing Modes

In this mode, the effective address calculation can be done before a load/store is executed or afterwards. Let's see what it is all about.

**10.18.3.1** Pre-indexed Addressing Mode

Observe the instruction LDR R0, f R7, #4], I Iere R7 is the base register and the effective address is R7 + 4. The data at this effective address is copicd to R7.

Next, see the instruction STR Rl, [R5, R6, LSL #2].The effective address = R5 + R6 left shifted twice.

In the above two instructions, there is a notable feature, however. After the load/ store is done, the base address content remains unchanged, that is, the effective address is not copied to the base register. But if we want the base address to contain the effective address, just suffix the instruction by the character'!' and then 'write back'occurs.

Consider the instruction LDR R2, [R6, #-8] !. In this, after the loading operation is done, R6 has the effective address written back into it.

Example 10.13

Calculate the effective addresses and explain what each instruction does.

STRB R2, | R6, R7, #0x24)!

LDRSII R4, [R10, Rl 1, ASR #4]

Solution

STRB R2,[R6, R7, #0x24]!

The ctfcctivc address is the sum of the contents of R6, R7 and the number 0x24. The content of R2 is stored in the effective address. After that, the effective address is copicd to R6.

LDRSI I R4, | R10, Rl 1, ASR #4]

Here, the effective address is the sum of the contents of R10 and Rll after arith­metically shifting it right by 4 positions. Hie half word in this address is loaded to R4. The contents of the base register remains unchanged.

**10.18.3.2** Post-indexed Addressing Mode

In this mode, the effective address calculation is done after the execution of the specific instruction has been done.

Take the case of the instruction LDR RO, [R4], #4

I Icrc the data pointed by the content of R4 is first copied to RO. After that, the content of R4 is changed to R4 + 4/Ihere is no need of the '['operation because that is exactly what post-indexing does.

Example 1 0.14

Let's add 10 numbers which are in memory. The numbers are 16-bit long, that is, half words, and use two byte spaces. The prc-indexcd mode of addressing with write back is used to index the half words which have addresses with a spacing of 2 between them. The instruction LDRII R2, IR7, #2j! does the indexing of the 16-bit numbers.

Solution

AREA DADD, CODE, READONLY ENTRY

LDR R7, = TABLE MOV RO,#9 LDRH Rl,[R7] LDRH R2, [R7,#2]! ADD Rl,Rl,R2 SUBS RO,RO,#1 BNE REPT B STOP

STRT

REPT

STOP TABLE

;copy the address of Table to R7 ;R0 = 9

;load l2t number from memory to Rl **;pre-indexed with writeback** ;R1 = R1+R2 ;RO = R0-1

;repeat the addition until RO = 0 ;last line

DCW 3456,7859,1234,9876,3452,3214,7864,0987,2032 END

Let's examine the salient features of this program.

The numbers to be added are stored in code memory, just after the last line of the program.

There arc 10 numbers to be added. The first number is loaded into register Rl and the rest are loaded one by one into R2.

R0 is used as a counter to the numbers. In a general case, if there are N numbers to be added, R0 = N-l. Here N = 10.

The loading of the numbers to R2 is done using a loop. Since the numbers are half words, their addresses are to be incremented by2.'Ihis is done very efficiently by the pre-indexed addressing with write back scheme. After one half word is accessed, the effective address is written back to the base register R7 in readiness for accessing the next half word.

The address corresponding to TABLE is a 32-bit constant. It is calculatcd by the assembler, and loaded into R7 using the techniques mentioned in Section 10.17.

10.19 Readonly and Read/Write Memory

The two memory areas defined by the compiler arc 'Readonly'for code, and 'Read/write' for data. Usually this corresponds to ROM and RAM in a physical system. RAM is used for intermediate results, for temporary storage, etc., as this is volatile memory. We can store data permanently in the readonly memory, process it and copy it in RAM. In the readonly memory, data is written using directives like DCD, DCW, etc. From there, it is copied to rcadwritc memory using load and store instructions.

Example 10.15

AREA FIRST, CODE, READONLY ENTRY

LDR R7, = NUMS LDR R8, = NUMS1 LDR R9, = NUMS2 LDR Rl,[R7] STR Rl,[R9] STR Rl,[R8] ;load the address of NUMS in R7

;load the address of NUMS1 in R8

;load the address of NUMS2 in R9

;load the word to Rl

;store the word in Rl in NUMS2 ;store the word in Rl in NUMS1

STOP B STOP NUMS DCD 653451134

DATA, READWRITE

AREA SECOND, NUMS2 SPACE 60 NUMS1 DCD 0 END

In Example 10.15, three memory areas have been defined: one in readonly, and two in rcadwritc memory. What is accomplished is just the transfer of a word from readonly memory to readwrite memory. In readwrite memory, one part is a space of 60 bytes. The next is a word space which is initialized to 0. After the execution of the program, the number 653451134 is copied to both these spaces.

Example 1 0.16

AREA STRIN1, CODE, READONLY ENTRY

STRT LDR Rl, LDR R0, BL COPY STOP B STOP

SOURCE /pointer to source string DESTIN /pointer to destination string /call procedure for copying /last line of execution

COPY LDRB R2, [Rl],#l /Load byte and update address.

STRB R2, [R0],#1 CMP R2, #0 BNE COPY MOV PC,LR

/Store byte and update address. /Check for 0

/repeat until the string is over /return to calling program

SOURCE DCB "I am sam",0

AREA STRIN2,DATA,READWRITE DESTIN DCB 0 END

Example 10.16 uses many of the programming aspects that we have been discussing so far. Let's have a look at the important features of this program.

There is an ASCII string written in readonly memory using the DCB directive. Such a string is enclosed in double quotes and each character is a byte.

One readonly and one read/write memory areas have been defined.

After the ASCII string, a 0 is used as a terminating character. The arrival of this 0 in R2 is used to check whether the required transfer of the string is done.

The instructions for loading and storing are suffixed by 'B'which indicates that only a byte is to be transferred.

Post-indexed mode of addressing is used for load and store. The addresses need to be incremented only by 1, as only a byte is transferred.

The instructions for loading and storing are in a procedure named COPY. The pro­cedure is called'by the BL instruction which does branching and also copies the current PC to the link register. The last line of the procedure is copying the LR back to PC.This constitutes the 'return'to the main program.

10.20 Multiple Register Load and Store

We have seen in Section 10.18 the LDR and STR instructions, which transfer data (in the form of bytes, halfwords or words) between a register and memory. Now, let's see an advanced (or let's say an extended) form of loading and storing, wherein multiple registers are involved. But only data in the form of words (32 bits) can be handled by these instruction. The mnemonic of multiple load and store is LDM/STM.

10.20.1 The LDM Instruction

Let's talk about LDM first—it has the syntax

LDM{cond}address-mode R»{!},r<£-/»/{A}

Rn is the base register for the load operation.The address stored in this register is the starting address for the load operation. There can be a number of modes for specifying the address, register-list is a comma-delimited list of symbolic register names and register ranges enclosed in braces. There must be at least one register in the list. Register ranges are specified with a dash. For example, {R0—R5, R9} is a list. The ! option is for 'write back' and the A option is relevant for interrupts. We will not discuss the second option here. Write back is not to be specified if the base register Rw is in register-list.

i •

■ •

■ i

■ \*

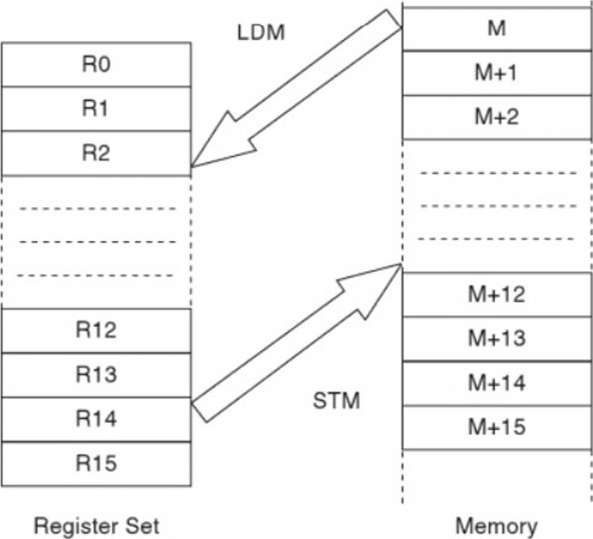


Figure 10.19 The LDM and STM instructions

Multiple register load means that multiple memory locations are to be acccssed, and loaded into multiple registers. There is a 'base register' acting as a pointer for the first memory location to be accessed. This register is then incremented or decremented to point to the next memory addresses. There are four options for handling this. The base register can be incremented or decremented by 4 (one word needs four addresses) for cach register in the operation, and the increment or decrement can occur before or after the operation.The suffixes for these options are as follows:

IA - increment after IB - increment before DA — decrement after DB - decrement before

Consider the instruction LDM DA RO, {R4-R9}

The base register here is RO. Let us assume it holds the number 0x45000000. The opera­tion of this instruction is that the 32-bit word at that address is pointed by R0, and that word copied to R4. Then the address is decremented to point to the next word. So the new address is [R0-4J, and this word is copied to R5. The sequence of decrementing the address and loading data from memory is done for the registers R4, R5, R6, R7, R8 and R9.

It is obvious that a single instruction replaces six LDR instructions. Is there any advantage in this? As far as execution is concerned, it is 'No'. All the six load operations have to be done. But note that only one 'instruction fetch\*cycle is needed for the six load operations together. So there is definitely some savings in terms of time.

What Is the difference in operation of the following instruction?

LDMIA R10,{ R9, Rl - R5}

Here the base address is in R10, and after each data transfer, it is incremented by 4. In the destination register list, R9 is specified first, but the processor has a particular way of handling the list. The lowest register will always be loaded from the lowest address in memory, and the highest register from the highest address. I Iere Rl gets the data in the address pointed by R10.

10.20.2 The STM instruction

This has the same format as the LDM instruction. Consider the instruction

STMIA Rl, {R2-R4} This will be equivalent to the instructions

STR R2, [Rl] STR R3, [Rl. #4] STR R4,[R1.#8J]

After the sequences of four stores are over, the base content does not vary, however. If you need it to be changed to that of the final address, the writeback operator '!'is to be used. So write the instruction as STMIA Rl!,[R2-R4}

Now let's use the LDM and STM instructions to simply Example 10.16 which transfers bytes from one portion of memory (Readonly) to another portion (Read/ write). But the multiple load/store instructions can be used only for words (32 bits). So Example 10.17 has been modified and used to move 6 words.

Example 10.17

AREA STRIN1, CODE, READONLY ENTRY

LDR Rl, = SOURCE /pointer to source

LDR RO, = DESTIN /pointer to destination

LDMIA Rl,{R2-R8} ;Load six words to R2-R8

STMIA R0,{R2-R8) ;Store six words in destination STOP B STOP

SOURCE DCD 0x675889,0x1234568,0x9876543,0x2345678,0x8907653

AREA STRIN2 ,DATA,READWRITE /define the R/W memory area DESTIN DCD 0 END

Here 6 words from the source memory have been copied to six registers using just one instruction. In the next instruction, these six words are stored in the destination memory

Note how simple, the program is.

Example 10.17 illustrates the idea that block data transfers can be simplified using the multiple register instructions. But their real importance is for stack implementation. Stacks are a necessity for any processor; stacks are needed for storing data temporarily and also for storing return addresses and register values during procedure calls. We will see this now. For those who are not very familiar with the concept of stacks, here is a brief review.

10.20.3 Stack

A stack is an area in memory, the accessing of which is done in a special way. Most stacks arc Last-In First-Out (LIFO) type stacks. This means that the last data that was stored is the first one that can be taken out. It is sequential access that is done, and not random acccss.Two operations arc defined for a stack, that is, the PUSH, in which data is written into the stack, and POP in which data is read out and loaded into registers.The stack has a pointer to its top which is called the Stack pointer (SP). For ARM, this is register Rl3. This means that the address of the top of the stack is to be available in SP.

**10.20.3.1 *Types of Stacks*** Ascending/Descending and Empty/Full

An ascending stack grows upwards. It starts from a low memory address and, as items arc pushed onto it, progresses to higher memory addresses. A descending stack grows downwards. It starts from a high memory address, and as items are pushed onto it, it progresses to lower memory addresses.

In an empty stack, the stack pointer points to the next free (empty) location on the stack, i.e., to the place where the next item to be pushed, will be stored. In a full stack, the stack pointer points to the topmost item in the stack, that is the location of the last item pushed onto the stack. In practice, stacks are almost always full and descending. Most stacks are 'Full descending' types.

Let's consider a descending stack in which SP is first decremented and then data is pushed in.The reverse occurs for the POP operation. Stacks allow data to be pushed or popped only as words (32 bits for ARM). Consider that SP = 0x50002000, and the contents of Rl and R2 are pushed in At the end of the operation we find that SP = SP-8 = 0x50001 FF8. ARM does not have a mnemonic for PUSH, instead it uses the STM instruction. To simplify the use of the STM/LDM instructions corresponding to PUSH and POP for different types of stacks, Table 10.18 can be referred to.

For the kind of stack that we are talking about now, what is the instruction we can use for pushing the contents of registers Rl to R3?

The answer is STMDB SP! {R1-R3). We need SP to be used as the base register. For pushing in, SP is first decremented, and then storing is done. So we use the suffix 'DB' along with SP.The operator '!\*is used such the decremented value is available in SP.

See this simple program (Example 10.18) in which SP is initialized to 0x40000200. Some values arc loaded into registers Rl to R3. Using the STMDB instruction, the con­tent of the three registers, that is, 3 words are pushed to the stack, and will be available in memory. At the end of the program, SP will be found to have the value of 0x400001 F4.

Table 10.18 | Types of Stacks and Corresponding Instructions to be Used

Push

Pop

Stack Type

STMFD (DB) STM FA (IB) STM ED (DA) STMEA (IA)

LDMFD (IA) LDMFA (DA) LDMED (IB) LDMEA (DB)

Full Descending Full Ascending Empty Descending Empty Ascending

Example 10.18

AREA STCK, CODE, READONLY ENTRY

LDR SP, = 0x40000200 MOV Rl,#1 MOV R2,#2 MOV R3,#3 STMDB SP!,{R1-R3} STOP B STOP END

Now, in this program, if the STM instruction is changed to STMIA, the stack becomes an ascending stack, and the value of SP will be 0x4000200C, after program execution. Thus, it is obvious that a stack is a data structure which can be defined by software.

10.20.4 Stacks and Subroutines/Procedures

For most processors, procedures use a stack to store the return address. A procedure is taken up by a 'CALL' instruction. This causes the action of pushing the current value of PC onto the stack. The procedure ends with a 'RETURN'instruction. This causes the PC value to be popped back.

For ARM, so far (Section 10.16.1) we have used procedures without the necessity of a stack. That is because the Link Register (LR) keeps the return address, when a proce­dure is called. But think of the case of nested procedures. There is only one link register for a mode, and a new procedure will overwrite the existing link register which stores the details of the previous procedure, and very soon things may go out of hand.

In such cases, a stack is a necessity. Each time a procedure is called, the PC value is saved in the LR, as is the usual case. When a nested procedure comes in, the content of the link register is pushed on to the stack, and popped out from the stack when exit­ing the procedure. Figure 10.20 shows the sequence of actions needed to take care of a nested procedure.

Now, let's try to understand the sequence of actions indicated by Figure 10.20. In the main program, we define a stack by giving a value to the stack pointer (SP).

Main Program PROC1 PROC2

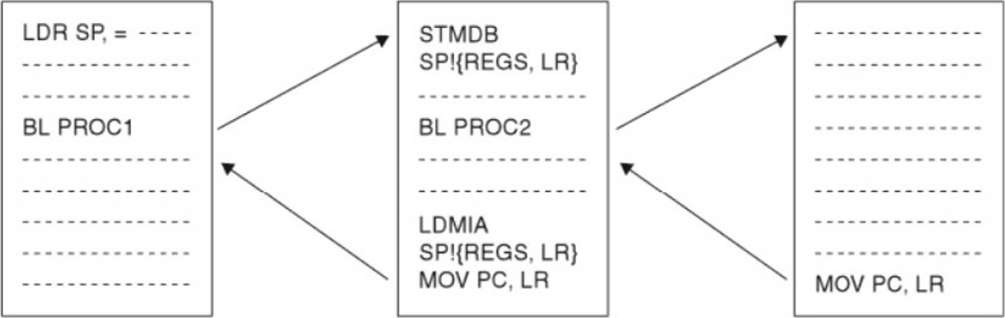


Figure 10.20 | Sequence of actions needed for a nested procedure

The main program has a procedure named PROCl which is called by the instruction BL PROCl. This instruction causes the current PC value to be copied to LR. In PROCl, since we anticipate a nested procedure, we push LR and the working registers to stack using the instruction STMDB SP!,{REGS,LR}. Thus the content of LR is safely stored in the stack.

In the procedure PROCl, another procedure is called by the instruction BL PROC2.This instruction causes the copying of the present PC to LR. In PROC2, there is the instruction MOV PC, LR at the end. This will get the PC value back from LR, and thus execution goes back to PROCl.

At the end of PROCl, there is the stack based instruction LDMLA. SP!, {REGS, LR}. This retrieves the contents of LR.This is given back to PC by the instruction MOV PC, LR.Thus, execution goes back to the main program.

Aiv part of memory can be defined as a stack, by simply defining the content of the stack pointer register. Lets write a procedure using a stack.

Example 1 0.19

AREA NESTED, CODE, READONLY ENTRY

;define SP

;Call PROCl ;load R6

;save registers and LR on stack

LDR R7, = 0X40000000 LDR SP, = 0x40000210 MOV Rl,#1 MOV R2,#2 MOV R3,#3 BL PROCl LDR R6,[R7] STOP B STOP

;call PROC2 ;store R5

;retrieve registers from stack ;copy LR to PC

;the nested procedure PROC2 ;go back to PROCl

PROCl STMDB SP!,{LR,R1-R3} MOV Rl,#0x34 MOV R2,#0x45 MOV R3,#0xDC BL PROC2 STR R5,[R7] LDMIA SP•,{R1-R3,LR} MOV PC,LR

PROC2 ADD R4,R2,Rl

ADD R5,R4,R3

MOV PC,LR END

Example 10.19 shows the instance of a nested procedure and the use of the stack. The example is in tune with the sequence outlined by Figure 10.20. Nothing very important is achieved by the program, But it shows how any nested procedure can be written. PROCl changes the contents of registers Rl, R2 and R3, but since they have already been saved on the stack by the STMDB instruction, their contents can be retrieved while returning to the main program.

PROC2 adds the new contents of Rl, R2 and R3, and returns. In PROCl, the sum in R5 is stored in the memory location pointed by R7. Later, in the main program, this content is loaded to R6.

Example 1 0.20

Write a program which arranges numbers (stored in readonly memory) in ascending order, and place them in the RAV memory.

AREA NUM,DATA,READONLY

ARRAY DCD 2,7,4,5,11,17,3,15,8,6,9,19,10,23,20 AREA COD,CODE

ENTRY

LDR RO, = ARRAY /load ARRAY to RO

LDMIA RO, {R1-R10} /load 10 numbers to Rl to R10

MOV SP,#0x40000000 /location of R/W memory

STMIA SP,{R1-R10} /store the 10 numbers

ADD SP,#40

ADD R0,#40

LDMIA R0, {R1-R5} /load next set of 5 numbers STMIA SP,{R1-R5} /store them

MOV SP,#0x40000000/address of Read-write memory MOV Rl,#0 /Initialize counter Rl to zero

MOV R3,SP

LOOP1 MOV R2, #0 /outer loop, counter from one end

MOV R4,SP

LOOP2 CMP R2,#14

|  |  |  |
| --- | --- | --- |
| BEQ | OUTER | /branch to OUTER |
| ADD | R2,#l | /increment the counter |
| LDR | R0,[R4] | /stored as 4 bytes |
|  |  | /hence a jump of 4 |
| LDR | R5,[R4,#4] |  |
| ADD | R4 , #4 |  |
| CMP | R0,R5 | /comparing nearby values |
| BLT | LOOP 2 |  |
| MOV | R6, R0 |  |
| MOV | R0, R5 | /swapping and storing them |
| MOV | R5,R6 |  |
| STR | R5,[R4] |  |
| SUB | R4 , #4 |  |
| STR | R0,[R4] |  |
| ADD | R4 , #4 |  |
| B LOOP2 | |  |

ADD Rl,#1 CMP Rl,#15 BNE LOOPI

STOP B STOP END

This program uses the concept of'bubble sorting'. First the 15 numbers stored in the vari­able ARRAY are loaded, in two different steps. After that, two loops are taken in which the first loop uses a counter from 0 to 14. The first loop traverses from one end of the array to other, and the second loop is used to compare nearby values and swap them according to which value is lesser than the other. Hence, by each iteration of the outer loop, the low­est value in the array slowly comes to the first element, and this process continues.

Conclusion

With this, we come to the end of our discussion on the architecture and assembly lan­guage programming for ARM. There are a few more instructions, pseudo instructions and directives that haven't been dealt with, but that can be learned by referring to a book fully dedicated to this processor.

KEY POINTS OF THIS CHAPTER

o ARM is the most popular of the 32-bit processors in the market.

o ARM, the company does not fabricate chips—instead it sells the design as 'Intellectual Property'.

o The ARM family consists of members ARM7,9,10,11 and Cortex versions.

o Lower power dissipation and good computational capability are the chief attributes of the ARM processor.

o The processor has a large set of registers, and operates in seven modes.

o It can be programmed in assembly and one of the IDEs available is Keil RVDK.

o The barrel shifter in the ALU has a lot of relevance, as it simplifies computations.

O ARM can use data processing instructions conditionally, by suffixing S to it.

o There is a link register (LR) for simplifying procedure calls.

o It has a special mechanism for handling immediate data which is bigger than 8 bits,

o It has multiple register load and store instructions .

o Stacks are needed when nested procedures come .

QUESTIONS

1. List out the important features that make ARM ideal for embedded applications.
2. Name two aspects in the design of ARM which has made it a processor with 'low-power dissipation'.
3. What is the use of a cache for any processor?
4. What does the acronym ISA mean to you?
5. What are the advantages and disadvantages of'pipelining'?
6. What is the penalty incurred in the case of'unaligned'data?
7. How is'rotation to the left'achieved in ARM?
8. How is the instruction LDR different from the pseudo instruction LDR?
9. Why is it that compare instructions don't need the suffix'S'?

10. How is the write back operator used in the'pre-indexed'mode of addressing?

EXERCISES

1 Write instructions for the following, without using any CISC type instruction.

* 1. move into R7, a byte multiplied by 8
  2. move into R6, a word multiplied by 17
  3. move into R5, a number divided by 8
     1. What do the following instructions mean and what is accomplished?
        1. ANDEQ Rl, R2, R4
        2. ADDHI R2, R4, R2
        3. MOVAL R7, R5
        4. SUBME R1, R2, R7
        5. CMP Rl, R2
        6. TEST Rl, R3
        7. MOVGT R2, R5
        8. ADDLT R5, R6, R7
     2. Write assembly language programs for the following.
        1. Find the factorial of any number (the factorial should fit in a 32-bit register)
        2. Do division using repeated subtraction
        3. Find the sum of the first 100 natural numbers. Save the result in memory
        4. Find the sum of 10 numbers stored in Readonly memory the result should be in Read/ write memory
        5. Store 15 numbers in memory and arrange them in
* descending order
* ascending order
  1. Write a program with a procedure call using
* without using stack
* using stack