Al-Azhar UNIVERSITY

Faculty of Engineering

Computers and Systems Engineering Department

EXPERIMENT 5 – ARM Processor

OBJECTIVES

- To understand the architectural details of the ARM processor and intricacies of its instruction set
- To appreciate the various generations of advancement in ARM families
- A road map of the register sets and processor modes to enable ARM programming

MATERIALS/EQUIPMENT NEEDED

1. ARM Simulator

INTRODUCTION

An ARM processor is one of a family of CPUs based on the RISC (reduced instruction set computer) architecture developed by Advanced RISC Machines (ARM). ARM makes 32-bit and 64-bit RISC multi-core processors. RISC processors are designed to perform a smaller number of types of computer instructions so that they can operate at a higher speed, performing more millions of instructions per second (MIPS). By stripping out unneeded instructions and optimizing pathways, RISC processors provide outstanding performance at a fraction of the power demand of CISC (complex instruction set computing) devices.

ARM processors are extensively used in consumer electronic devices such as smartphones,tablets, multimedia players and other mobile devices, such as wearables. Because of their reduced instruction set, they require fewer transistors, which enables a smaller die size for the integrated circuitry (IC). The ARM processor‘s smaller size, reduced complexity and lower power consumption makes them suitable for increasingly miniaturized devices.

ARM processor features include:

- Load/store architecture.
- An orthogonal instruction set.
- Mostly single-cycle execution.
- Enhanced power-saving design.
- 64 and 32-bit execution states for scalable high performance.
- Hardware virtualization support.

The simplified design of ARM processors enables more efficient multi-core processing and easier coding for developers. While they don't have the same raw compute throughput as the products of x86 market leader Intel, ARM processors sometimes exceed the performance of Intel processors for applications that exist on both architectures.

The head-to-head competition between the vendors is increasing as ARM is finding its way into full size notebooks. Microsoft, for example, offers ARMbased versions of Surface computers. The cleaner code base of Windows RT versus x86 versions may be also partially responsible -- Windows RT is more streamlined because it doesn‗t have to support a number of legacy hardwares.

ARM is also moving into the server market, a move that represents a large change in direction and a hedging of bets on performance-per-watt over raw compute power. AMD offers 8-core versions of ARM processors for its Opteron series of processors. ARM serversrepresent an important shift in server-based computing. A traditional x86-class server with 12, 16, 24 or more cores increases performance by scaling up the speed and sophistication of each processor, using brute force speed and power to handle demanding computing workloads.

In comparison, an ARM server uses perhaps hundreds of smaller, less sophisticated, low-power processors that share processing tasks among that large number instead of just a few higher-capacity processors. This approach is sometimes referred to as ―scaling out,‖ in contrast with the ―scaling up‖ of x86-based servers.

About the ARM processor

The ARM architecture has been designed to allow very small, yet highperformance implementations. The architectural simplicity of ARM processors leads to very small implementations, and small implementations allow devices with very low power consumption. The ARM is a Reduced Instruction Set Computer (RISC), as it incorporates these typical RISC architecture features:

- A large uniform register file
- A load/store architecture, where data-processing operations only operate
on
- register contents, not directly on memory contents
- Simple addressing modes, with all load/store addresses being
determined
- from register contents and instruction fields only
- Uniform and fixed-length instruction fields, to simplify instruction
- decode.

In addition, the ARM architecture gives you:

- Control over both Arithmetic Logic Unit (ALU) and shifter in every data-processing instruction to maximize the use of an ALU and a shifter
- Load and Store multiple to maximize data throughput.

These enhancements to a basic RISC architecture allow ARM processors to achieve a good balance of high performance, low code sizeand low power consumption.

ARM Block diagram

The main parts of the ARM processor are:
1. Register file: The processor has a total of 37 registers made up of 31 general 32 bit registers and 6 status registers
2. Booth Multiplier
3. Barrel shifter
4. Arithmetic Logic Unit (ALU)
5. Control Unit.

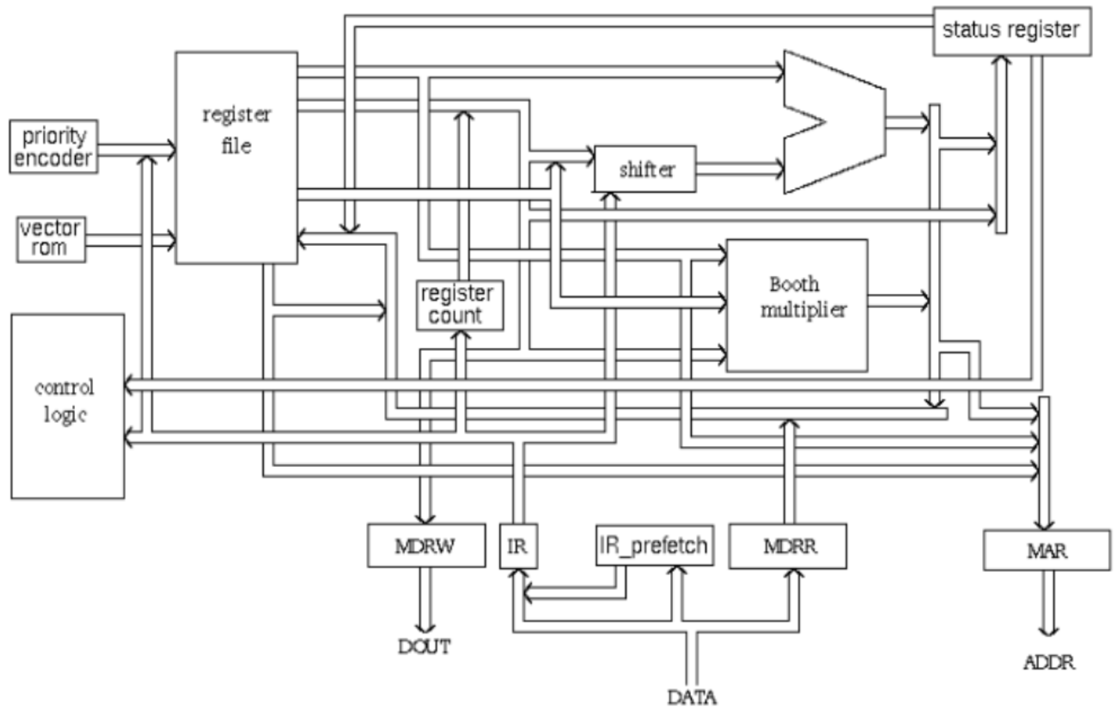


Figure 1 ARM Block Diagram

ARM Assembly Language

ARMSim# is a desktop application running in a Windows environment. It allows users to simulate the execution of ARM assembly language programs on a system based on the ARM7TDMI processor. ARMSim# includes both an assembler and a linker; when a file is loaded, the simulator automatically assembles and links the program. The window of ARMSim# program can be divided into three parts; code view (displaying code), stack view (displaying stacks) and register view (displaying 16 general purpose registers). There is a list of instructions that can be used to implement different tasks in assembly language such as LDR, STR, ADD, MUL etc.

Your first assembly program adds two numbers. The numbers, 3 and 5, are assumed to be in register 0 and register 1, respectively, while the Result is to be stored in the memory location whose address is in register 3.

1. Follow the steps below to run the above program in ARMSim#:
   (i) Write the program in note or text pad and save as .s (source) or .o (object) extension.
   (ii) Open the simulator and select File.

(iii) Choose Load by specifying the location of your file and the file will
be executed automatically.
(iii) You can view the content of the registers that you used in the program in the left side of your working window or register view area. What are the values of registers R0, R1, R2, and R3, before and after the
execution of the program?
2. Write the sample program to multiply two numbers in ―Getting Started
with ARMSim#‖. Run the program using the different running options,
run, step into and step over.
3. Write a program using ARM assembly language which adds two numbers, in variable A and B, and store the result in variable C.

@ A simple ARM assembly program for adding two numbers.

```
MOV R0,#3 @ Move the value 3 into register R0
MOV R1,#5 @ Move the value 5 into register R1
MOV R3,#0x3000 @ Move the address of the result into R3
ADD R2, R0, R1 @ Add the content of R0 and R1, keep result in R2
STR R2,[R3] @ Store the content of R2 at the address in R3
.END
```

@ A simple ARM assembly program for multiply two numbers.

```
MOV r2, #10 @ Load the value 10 into register r2
MOV r3, #2 @ Load the value 2 into register r3
MUL r1, r2, r3 @ Compute r2*r3 and store in r1 (10*2 = 20)
MOV r0, #1 @ Load 1 into register r0 (stdout handle)
SWI 0x6b @ Print integer in register r1 to stdout
SWI 0x11 @ Stop program execution
```

2. A simple program: Adding numbers

Let's start our introduction using a simple example. Imagine that we want to add the numbers from 1 to 10. We might do this in C as follows.

```
int total;
int i;
total = 0;
for (i = 10; i > 0; i--) {
    total += i;
}
```

The following translates this into the instructions supported by ARM's ISA.

```
        MOV  R0, #0          ; R0 accumulates total
        MOV  R1, #10         ; R1 counts from 10 down to 1
again   ADD  R0, R0, R1
        SUBS R1, R1, #1
        BNE  again
halt    B    halt            ; infinite loop to stop computation
```

BNE, will check the Z flag. If the Z flag is not set (i.e., the previous subtraction gives a nonzero result)
B, always branches back to the named instruction

2.2. Another example: Hailstone sequence

Pseudo code:

```
$iters \leftarrow 0$
while $n \neq 1$:
    $iters \leftarrow iters + 1$
    if $n$ is odd:
        $n \leftarrow 3 \cdot n + 1$
    else:
        $n \leftarrow n / 2$
```

Equivalent ARM assembly code:

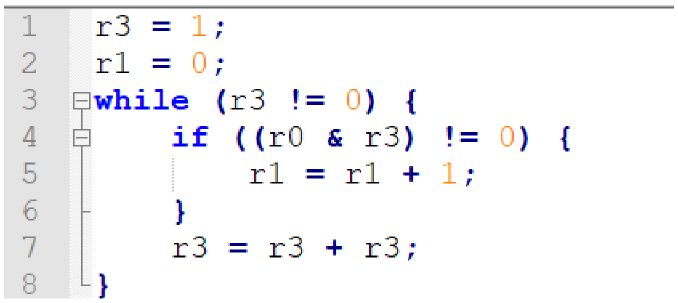
```
        MOV  R0, #5          ; R0 is current number
        MOV  R1, #0          ; R1 is count of number of iterations
again   ADD  R1, R1, #1      ; increment number of iterations
        ANDS R0, R0, #1      ; test whether R0 is odd
        BEQ  even
        ADD  R0, R0, R0, LSL #1 ; if odd, set R0 = R0 + (R0 << 1) + 1
        ADD  R0, R0, #1      ; and repeat (guaranteed R0 > 1)
        B    again
even    MOV  R0, R0, ASR #1  ; if even, set R0 = R0 >> 1
        SUBS R7, R0, #1      ; and repeat if R0 != 1
        BNE  again
halt    B    halt            ; infinite loop to stop computation
```

the **ANDS** instruction to perform a bitwise AND with 1. The **ANDS** instruction sets the Z flag based on whether the result is 0. If the result is 0, then this means that the 1's bit of n is 0, and so n is even.

ASR, arithmetic shift right

POST-LAB

1- Translate the below C code for counting the 1 bits in `R0` into ARM assembly code, using the registers indicated by the variable names.



```
r3 = 1;
r1 = 0;
while (r3 != 0) {
    if ((r0 & r3) != 0) {
        r1 = r1 + 1;
    }
    r3 = r3 + r3;
}
```