ARM Simulation using C++ and Multithreading

Suresh Babu S, Channabasappa Baligar

Abstract: - This project is to be produced a software simulation of an ARM processor. A hardware simulator is a piece of software that emulates specific hardware devices, enabling execution of software that is written and compiled for those devices, on alternate systems. Aim of this project is to develop an ARM simulator using C++ and Multithreading, the same is tested with 'GDB' tool in Linux 2.6.37.4. The main feature of the project is the implementation of the ARM simulation with multi-threading. The analysis phase of the project involved detailed studies of different ARM architectures and ARM assembly language. Most of the decisions about hardware components to include in the simulation and assembly instructions to support were to be made during this stage. This phase also involved identifying the requirements of the simulator. The next stage was design, in which the major parts are identified to develop the simulation part of an ARM processor. The implementation phase involved turning the major parts into code, following the design as closely as possible. C++ programming language is to be used as it is object oriented programming language to implement the project. Multithreading concept is to be adopted to execute decoding function and execute function, so that execution will become faster. GDB is to be used to debug the project.

Index Terms: ARM, Simulation, thumb, multithreading

I. INTRODUCTION

The first ARM processor was designed by Acorn Computer Ltd, Cambridge, United Kingdom in 1985. In 1990 Advanced RISC Machines (ARM) was founded and VLSI Technology became their first licensee. Today ARM licenses high-performance, low-cost, power- efficient RISC processors, peripherals, and system-chip designs to leading international electronics company [1].

A. Problem Definition

The ARM hardware operations some time cannot be checked practically. The hardware operations some time consumes more time. Using hardware devices in all the applications yields cost effective. In certain applications, hard ware devices can be replaced by an effective simulation model. The act of simulating something first requires that a model be developed; this model represents the key characteristics or 56 behaviours of the selected physical or abstract system or process [2], the model represents the system itself, whereas the simulation represents the operation of the system over time.

B. Objective

The objective of this project is to imitate the ARM processor. The ARM thumb instruction simulation is implemented using C++ programming language and multithreading concepts. Simulation of ARM thumb instruction is implemented using modularity.

Revised Manuscript Received on November 2014.

Suresh Babu S, M. Tech., (VLSI Design & Emd. Sys), UTL Technologies, Bangalore, India.

Channabasappa Baligar, M. Tech., (VLSI Design & Emd. Sys), Professor, UTL Technologies, Bangalore, India.

This project consists of three important modules which are fetching, decode and execute modules. The advantages of implementing this project using modularity are debugging becomes easier and easy to understand. These modules are to be implemented using C++ programming language, because it has excellent features which are binding, encapsulations, polymorphism and etc[l][2]. The objective is to propose system with accuracy and speed. Since this project is to be implemented using modularity, the important modules are fetch decode and execute. Pipeline concept is to be used to speed up the fetch, decode and execute cycles. Multithreading concept is also to be used to speed up the decode and execute operations further.

C. Methodology

The methodologies adopted for the design & implementation of ARM simulation are :

1 Literature survey to understand the ARM simulation design concepts
2 Adapt accurate and faster execution concepts to the design
3 Implement the project using modularity, so that debugging and understanding becomes easier
4 Apply pipeline concept to fetch, decode and execute to make execution faster[l]
5 Using Advanced C++ programming language to make encapsulation, binding which yields the security
6 Multithreading concept is to be applied to make execution still faster[6]

II. DESIGN APPROACH

This section describes the main issues considered when designing the ARM Simulator system. It is assumed that the reader have basic knowledge of Object Orientated features like objects, interfaces, inheritance, polymorphism and the use of abstract classes and methods[2]. The first section discusses initial design issues, describing the approach taken to create a framework for the ARM Simulator. The last section describes the final design of the ARM Simulator. The project's design is split into three major sections:

Simulator design:

1 Microprocessor
2 Memory
3 Instruction set
4 User Input
5 I/O file

Assembly language:

1 Syntax
2 Compiler

A. Designing the instruction objects

The instruction objects make up the core of the system. Within the ARM Simulator a instruction is responsible of executing itself, set results and tell the system what instruction to execute next.

A great effort was put down in designing these objects and their connection with the main system[2]. Each instruction object had to be capable of holding all its attributes. In addition each instruction should be capable of executing itself. The first problem encountered was how a set of instructions could be stored within a data structure, and be easily accessed at a later time. If all the objects within a data structure are of a different type, the object needs to be identified before it can be accessed, and all possible objects will need to be instantiated. Below elements are taken care while designing the simulator.

1 Instruction reading from the text file
2 Read instruction is decoded to get the actual 16 bit value to get the instruction.
3 After decoding the instruction, as per the pipeline stage, the instruction is executed to get the desired output.

B. Iterative & Incremental Model

The Incremental development model is a cyclic software development process. The main idea is to develop a system through repeated cycles (iterative) in small portions (incremental). Development in this model starts with a developing small version of the program. The purpose of this is to have a working program even though it does not have all the required functions, which are developed gradually through cycles. At any time a working program can be presented to the customer for feedback. This has a beneficial effect on the overall development as alteration can be done immediately without a need for altering a huge amount of cod [6].

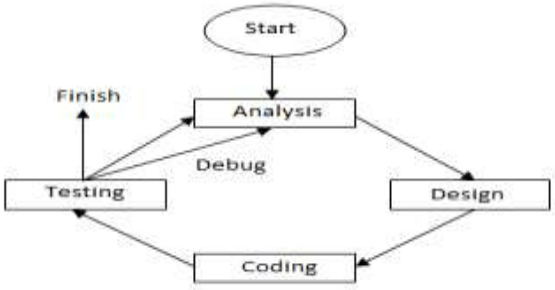

Figure 2.1: SDLC cycle

This model is divided into four major stages

1 Analysis - decision over changes to be implemented in the next stage. Only small portions are implemented at a time
2 Design - changes that need to be done before a new feature can be coded if any.
3 Coding - compiling the code to make sure that the coding is correct. The code is often recompiled after writing a few lines of code
4 Testing - involves running the program by using test data if it is requested. If an error occurs that the compiler has not recognized (typically memory leaks), the program must be debugged and either coding or design should be altered. However, only small portions are implemented through a cycle. It is then generally very easy to define the source of the problem.

The development continues around this loop until the program is finished (Figure given below). The Iterative & Incremental approach means that there is a running program at the end of each iteration. It obviously does not contain all the features that have been proposed to be implemented. This gives an opportunity to evaluate the program after each cycle and possibly a feedback gathered from users before it is fully completed.

Advantages:

1 Generates working software quickly and early during the software life cycle.
2 More flexible - less costly to change scope and requirements.
3 Easier to test and debug during a smaller iteration.
4 Each iteration is an easily managed milestone.

Disadvantages:

1 Each phase of an iteration is rigid and do not overlap each other.
2 Problems may arise pertaining to system architecture because not all requirements are gathered up front for the entire software life cycle can be a costly model to use.
3 Project's success is highly dependent on the risk analysis phase

C. Simulator Design

The simulator design consists of designing the overall functionality of the simulator, the way it input file is read. The simulator recognizes the file format and parses the instructions to the microprocessor. Figure shown below describes the high level architecture of the operation of the simulator.

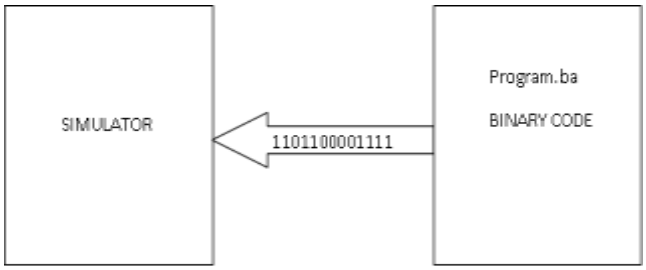

Figure 2.2: High-level operation of the simulator

It is important to ascertain the two terms "microprocessor" and "simulator". The term simulator refers to the implemented software that reads the input file and handles the user input commands. Whereas the microprocessor is a part of the simulator. The simulator passes the instructions to the microprocessor which then moves them to the memory and executes. The simulator accepts a binary file which contains binary coded instructions. With regards to the real machine, the binary file is not a real binary file. It is a common text file that contains a string of zeros and ones (binary). The file however appears as binary to the simulator [2]. The file is read by the simulator and split into groups of 16 bits (define in ISA). Each group is then specifically assigned memory space starting from the address OxOOOh. This part of the memory should not be accessed or altered explicitly by the assembly script. It should be considered as protected and thus OxOOXX is reserved for the program instruction. The simulator accepts a binary file which contains binary coded instructions. With regards to the real machine, the binary file is not a real binary file.

It is a common text file that contains a string of zeros and ones (binary). The file however appears as binary to the simulator. The file is read by the simulator and split into groups of 16 bits (define in ISA). Each group is then specifically assigned memory space starting from the address OxOOOh. This part of the memory should not be accessed or altered explicitly by the assembly script. It should be considered as protected and thus OxOOXX is reserved for the program instruction [1][2].

D. Instruction Set Architecture

Instruction set architecture defines data types, instructions, registers addressing modes and memory architecture.

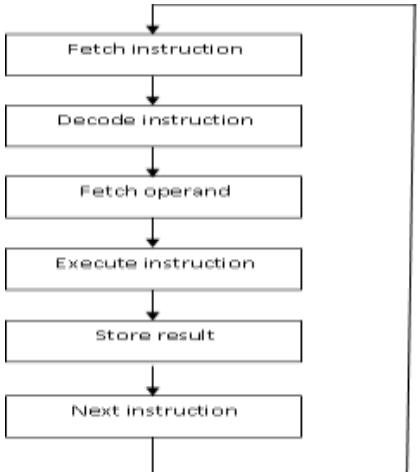

Figure 2.3: Instruction Execution Cycle

Instruction set architecture can be described by five different categories:

1 Operand storage - determines the location where operands can be held apart from memory.
2 Number of explicit named operands - determines the number of operands in a a typical instruction.
3 Operand location - determines whether operands can be accessed from memory directly or must be transferred to internal CPU registers for ALU operations.
4 Operations - operations provided by the ISA.
5 Type and size of operands - determines size and type of each operand.

E. Thumb instruction execution in ARM

Thumb encodes a subset of the 32-bit ARM instructions into a 16-bit instruction set space. Since Thumb has higher performance than ARM on a processor with a 16-bit data bus, but lower performance than ARM on a 32-bit data bus, use Thumb for memory-constrained systems[l]. Thumb has higher code density—the space taken up in memory by an executable program— than ARM. For memory-constrained embedded systems, for example, mobile phones and PDAs, code density is very important. Cost pressures also limit memory size, width, and speed. On average, a Thumb implementation of the same code takes up around 30% less memory than the equivalent ARM implementation. As an example, Figure given below shows the same divide code routine implemented in ARM and Thumb assembly code. Even though the Thumb implementation uses more instructions, the overall memory footprint is reduced. Code density was the main driving force for the Thumb instruction set. Because it was also designed as a compiler target, rather than for hand-written assembly code, Thumb-targeted code is designed using a high-level language like C++. The simulation flow of an ARM processor is shown below.

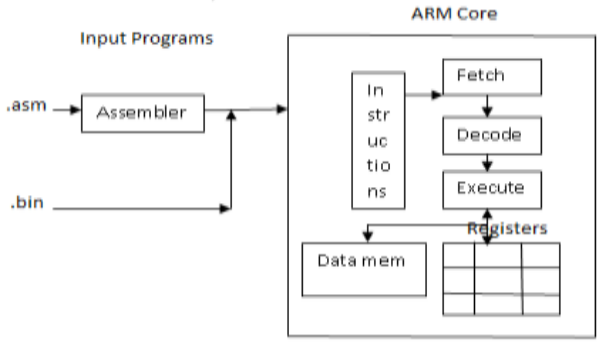

Figure 2.4: Simulation Flow [2]

F. Threading

Threading is one of the main features of the project. Multithreading or free-threading is the ability of an operating system to concurrently run programs that have been divided into subcomponents, or threads. The ability of an OS is to support multiple, concurrent paths of execution within a single process.

G. Features and Benefits of Threads

Mutually exclusive tasks, such as gathering user input and background processing can be managed with the use of threads. Threads can also be used as a convenient way to structure a program that performs several similar or identical tasks concurrently [6]. One of the advantages of using the threads is that you can have multiple activities happening simultaneously. Another advantage is that a developer can make use of threads to achieve faster computations by doing two different computations in two threads instead of serially one after the other. Threads support concurrent operations. For example,

1 Server applications can handle multiple clients by launching a thread to deal with each client.
2 Long computations or high-latency disk and network operations can be handled in the background without disturbing foreground computations or screen updates. Threads often result in simpler programs.
3 In sequential programming, updating multiple displays normally requires a big while-loop that performs small parts of each display update. Unfortunately, this loop basically simulates an operating system scheduler. In Java, each view can be assigned a thread to provide continuous updates.
4 Programs that need to respond to user-initiated events can set up service routines to handle the events without having to insert code in the main routine to look for these events.
5 Threads provide a high degree of control.
6 Imagine launching a complex computation that occasionally takes longer than is satisfactory. A "watchdog" thread can be activated that will "kill" the computation if it becomes costly, perhaps in favour of an alternate, approximate solution. Note that sequential programs must muddy the computation with termination code, whereas, a Java program can use thread control to non-intrusively supervise any operation.

Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication

7 Threaded applications exploit parallelism.
8 A computer with multiple CPUs can literally execute multiple threads on different functional units without having to simulating multi-tasking ("time sharing").
9 On some computers, one CPU handles the display while another handles computations or database accesses, thus, providing extremely fast user interface response

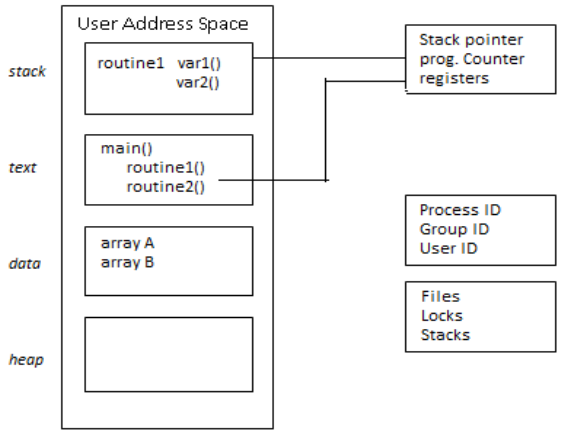

Figure 2.5: Routines without threads

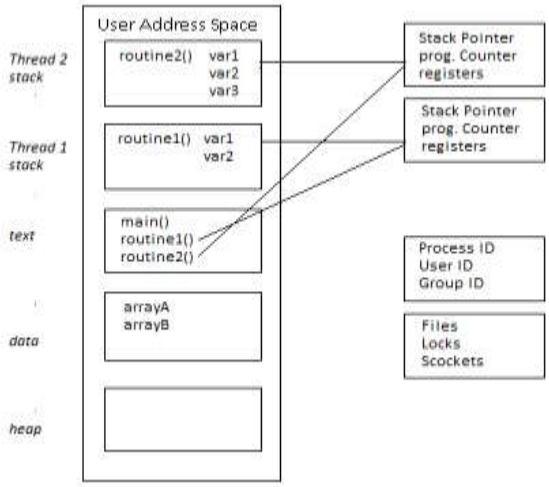

Figure 2.6: Routines with threads

Figure 2.5: Multiple threads execution in separate stack

In the case of "without threading", when the routine is called, the process is happening as shown above. Both the routine won't execute at the same time, instead the first, routine () is PUSHED to the stack and executed, then the routine is POPED. Then the same is done for routine2 ()[6] as shown above. In the case of "with threading", when the routine is called, the process is happening as shown above. Both the routine will execute at the same time, the first routine, routinel() is PUSHED to in the Thread 1 stack and executed, then the routine is POPED. Then the same is done for routine2 () in the Thread2 stack[6].

H. Thread Safety

Threads are an extremely useful programming tool, but can render programs difficult to follow or even non-functional. Consider that threads share the same data address space and, hence, can interfere with each other by interrupting critical sections (so-called "atomic operations" that must execute to completion). For example, consider that a write operation to a database must not be interrupted by a read operation because the read operation would find the database in an incorrect state. Or, perhaps more graphically, consider the analogy of two trains that need to share the same resource (run on the same stretch of track). Without synchronization, a disaster would surely occur the next time both trains arrived at nearly the same time.

The state of a running program is essentially the value of all data at a given instant. The history of a program's execution then is the sequence of states entered by a program. For single threaded application there is exactly one history, but multithreaded applications have many possible histories-some of which may contain a number of incorrect states due to interference of threads. The prevention of interference is called mutual exclusion and results primarily from synchronization, which constrains all possible histories to only those containing exclusively good states. Condition synchronization refers to threads waiting for a condition to become true before continuing; in other words, waiting for the state to become valid or correct before proceeding. In Java language terms, synchronized methods or code blocks have exclusive access to the member variables of an object resulting in atomic code blocks. A safe program is one that has been synchronized to prevent interference, but at the same time avoids deadlock.

III. IMPLEMENTATION

This section describes some of the main issues carried out during implementation. High priority has been given in building the simulator as accurate as possible, all down to binary representation of the instructions. To achieve this lot of effort has been put in functionality that is not directly visible to the user.

A. Simulator Operation

The simulator is or a container that contains the implemented microprocessor and the input/output communication. The simulator accepts a binary file with the b extension. It is assumed that the file contains a string of 0's and l's which defines the instructions. These instructions are passed to the microprocessor which decodes and executes each instruction. All programs written for the simulator must either be written in machine code, or in a symbolic form which requires the symbolic form to be compiled to produce the equivalent machine code [2].

The microprocessor operates in the manner:

1 fetch an instruction from the memory unit
2 read the instruction
3 decode and execute instruction

The instruction fetching continues until the terminating instruction is read. The execution is then halted by the simulator.

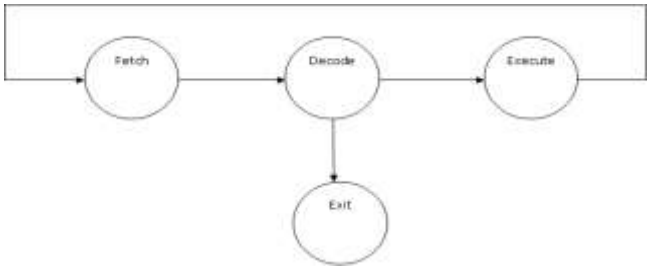

Figure 3.1: Instruction Execution Process

Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication

Each of the steps is executed at a certain speed. However, this speed by default would be the speed of the computer itself, the speed of simulation has to be decreased. The rate at which the execution is performed is determined by the clock and timing of the implemented microprocessor.
Opcodes are inputs to the ARM simulation. 16 bit opcodes are found in an input file from where the decoding of the instruction is performed separating the instructions with register information as explained in the table above. Block diagram of ARM Simulation using Software is shown below

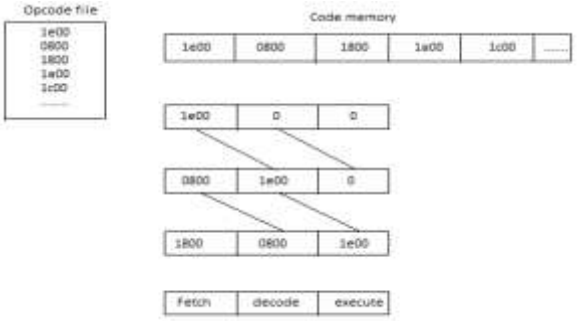

Figure 3.2: Block diagram of ARM Simulation

IV. TESTING AND RESULTS

This section describes the testing which is to be carried out on the ARM Simulator application.. Some of the test methods used are Module testing, refers to the testing of classes. This method has been carried out through the implementation, testing each new part before integrating with the full system. State boundary testing' refers to testing the system with boundary values. 'System testing' refers to over all systems testing. These two have been used in the testing phase of the project.

```
*****************************************
-------------ENTER----------
-------<1> or <2> or <3>--------
*****************************************


*****************************************
1-> Simulator with pipeline
*****************************************

*****************************************
2->Simulator with pipeline and threading
*****************************************

*****************************************
3->Performance difference between two
*****************************************
*****************************************
4-> Exit
*****************************************
ENTER THE CHOICE:
```

Figure 4.1: Simulator User Interface

```
pipe line stages
-----------------------------------------
fetch stage             decode stage            execute stage
-----------------------------------------------------------------
    48ff                    0                       0

pipe line stages
-----------------------------------------
fetch stage             decode stage            execute stage
-----------------------------------------------------------------
    2200                48ff                        0

pipe line stages
-----------------------------------------
fetch stage             decode stage            execute stage
-----------------------------------------------------------------
    200a                2200                    48ff

BEFORE EXECUTION
GENERAL PURPOSE REGISTERS
                        R0:     0
                        R1:     0
                        R2:     0
                        R3:     0
                        R4:     0
                        R5:     0
                        R6:     0
                        R7:     0
SPECIAL REGISTERS
                        SP:   0
                        LR:   0
                        PC:   0
                        CPSR:   0
```

Figure 4.2: Fetch, decode & execute stages

```
BEFORE EXECUTION
GENERAL PURPOSE REGISTERS
                        R0:         a
                        R1:         0
                        R2:         8
                        R3:         0
                        R4:         0
                        R5:         0
                        R6:         0
                        R7:         0
SPECIAL REGISTERS
                        SP:     0
                        LR:     0
                        PC:     c
                        CPSR:4000
MOVRI instruction.....................

AFTER EXECUTION.......................
GENERAL PURPOSE REGISTERS
                        R0:         a
                        R1:         a
                        R2:         8
                        R3:         0
                        R4:         0
                        R5:         0
                        R6:         0
                        R7:         0
SPECIAL REGISTERS
                        SP:     0
                        LR:     0
                        PC:     e
                        CPSR:4000
```

Figure 4.3: Register value before and after execution

```
MOVRI instruction.....................

AFTER EXECUTION.......................
GENERAL PURPOSE REGISTERS
                        R0:         a
                        R1:         a
                        R2:         8
                        R3:         9
                        R4:         0
                        R5:         0
                        R6:         0
                        R7:         0
SPECIAL REGISTERS
                        SP:     0
                        LR:     0
                        PC:    12
                        CPSR:4000

FILE end..............
System time taken(pipeline+thread):0.53

FILE end..............
System time taken(pipeline)  :0.55
```

Figure 4.4: Time difference between with thread and without thread

V. COCLUSION

The aim of this project is to produce a simulation of an ARM Processor thumb instructions, with fast decode and execute stages of the pipeline stages
The following list summaries the main achievements of the software:

1 Thumb instructions can be simulated through software
2 Software is implemented using C++ programming so that binding and encapsulation between the objects create security
3 Pipeline concept is used to increase the speed of instructions execution
4 Threads are used to execute fetch, decode and execute methods, so that execution still become faster

REFERENCES

1. ARM System Developer's Guide by Andrew N SLOSS, Dominic SYMESS, Chris WRIGHT
2. Alpa Shah, Columbia University, ARM Simulator, Proceedings of the IEEE International Conference
3. Alpa Shah, Columbia University, ARMSim, An Instruction Set Simulator for the ARM Processor, Proceedings of the IEEE International Conference.[ajs248@cs.columbia.edu]
4. M7TDMI ARM Processors User's Manual, Advanced Risc Machines Ltd
5. ARM System Developer's Guide, Designing and Optimizing System Software, by Andrew N Sloss, Dominic Symes and Chris Wright
6. POSIX Threads Programming, Author: Blaise Barney, Lawrence Livermore National Laboratory

Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication