ARM Processor

Shahriar Attar 810100186, Shahnam Feyzian 810100197

Computer Architecture Lab, Electrical and Computer Engineering Department, University of Tehran

***Abstract*— This document gives detailed instructions on ARM processor implementation in the Computer Architecture Laboratory at the ECE department.**

Keywords*— *ARM, Pipeline, Hazard-Unit, Forwarding-Unit, SRAM, Cache Memory

I. Introduction

The ARM processor is a fundamental component in the computing industry, revered for its exceptional energy efficiency and architecture predominantly utilized in mobile devices, tablets, and embedded systems. The processor's design is based on a Reduced Instruction Set Computing (RISC) philosophy, which simplifies the instruction set to improve processing speed and decrease power consumption. ARM processors are celebrated for their ability to perform complex computations while maintaining a low power profile, making them ideal for battery-powered devices where longevity and performance are paramount. We are going to build a step-down model of ARM (based on this file) using Verilog HDL and then simulate it using ModelSim and then using Quartus we synthesize it to see results on the FPGA board. It's going to support 13 commands and we are going to perform bubble sorts and see results. Datapath can be seen here. For implementing in a pipeline manner after each stage ({stage}.v) there are multiple registers ({stage}_Reg.v).

II. Pipeline

The First thing we did was adding registers and stages so we could have a pipeline structure. The PC value was getting updated by the adder inside the IF module, the reason for adding PC with 4 is that every instruction is 32 bits. The Instruction Module is a simple *case statement* that stores a simple bubble sort program (with few other instructions) implementations in ARM assembly, each line has a comment that explains what that line is for. Cause at first we just wanted to implement the pipeline the instructions were less than the final version, and in all modules, we only moved the PC value, and in the top module, we connected the modules. You can see that version of the code here. The result was a wave-like change in PC values inside stages that could be found here. The problem we encountered in this step was that at first, we didn't import the pin assignments needed for the FPGA board and the 7-segment lights had a weird shape. Another problem that we faced was that the code was compiled and simulated fine in ModelSim but in Quartus when we wanted to synthesize it we got errors in registers, the reason was that in *always statements* we should first check for *rst* signals and then *clk* signals. You can learn more about synthesizable verilog syntax in Reference #1 and #2.

III. Instruction-Fetch Stage

The Instruction-Fetch Stage is the first stage in the ARM pipeline. It fetches the instruction from memory. This is done by generating a new virtual address. This address can either be a branch target address provided by a branch prediction for a previous instruction or if there is no prediction made this cycle, the next address will be calculated sequentially from the fetch address used in the previous cycle, so we need a multiplexer to help us choose different inputs for PC register when we have predicted a branch and for when we simply want to continue instructions. The implementation of this stage can be found here. Also, the Instruction Module, which was explained earlier, and the instructions can be found in that module, and the program that we try to simulate is a simple bubble sort.

IV. Instruction-Decode Stage

The Instruction-Decode Stage in the ARM pipeline is where the fetched instruction is interpreted. This involves the control unit of the CPU deciphering the instruction. A high-level overview of what happens during this stage:

A. Instruction Identification

The fetched instruction is identified. This involves determining the type of instruction and the operations it performs, done by dividing instruction into parts based on Table #2 in ARM_ISA, and the file given to us. Line 20-55 of this file.

B. Instruction Decoding

The instruction is decoded. Instructions are typically stored in a format that requires some translation (decoding) to become the control signals for instruction execution. This decoding presents an abstraction layer between the hardware and the software, allowing the control signals used for execution to be changed without breaking binary compatibility.

C. Control Unit

The Control Unit Module in combination with the Condition Check Module (which is instanced from lines 57-86 of ID) enables us to have a conditional processor. The Condition Check Module works based on Table #3 in ARM-0 Description and the Control Unit Module follows the rules in Table #5 of ARM-0 for determining the Executive Command for the ALU module and also other needed signals such as *WB_EN*, *MEM_R_EN*, *MEM_W_EN*, *B* and *S*. All of them are enabled/disabled based on what kind of operation we are running. If conditions (the output of Condition Check Module are not met) or we have encountered a hazard all of these signals would change into 0 which indicates a NOP (No Operation).

D. Operand Fetching

The operands for the instruction are fetched. This involves reading values from registers or memory, lines 89-106 of this. The Register File Module writes in sync with the clock but reads in an async manner. Inputs are based on the instruction decoding we have done earlier but the multiplexer for the second input of this module is for commands like *STR* that use *src2*, but other commands use *Rm* for the second input). Also, in addition to this, for better results and not reading unnecessary registers from the Register File Module we check whether or not it should, if not the register file would return z (high impedance) instead.

E. Hazard Detection

The decode pipeline stage can also detect various data and structural hazards relative to previous instructions and take mitigating actions (e.g., using result forwarding) or restraining actions (e.g., stalling the pipeline). This part will be explained shortly in the future.

F. Results

Then we ran the first 18 instructions and checked signals issued by the Control Unit Module. The results are here, and for validation, we can see that for instance only for the *LDR* commands the *MEM_R_EN* signal is issued. Other signals are only issued in commands when they are needed, like in the fourth instruction the ALU Module needs to add the two inputs, and as can be seen the *EXE_CMD* signal verifies that, Or for the fifteenth command we need to execute the *ADDEQ* command the *cond* signal is 0000 to check it for running instructions.

This version of the code corresponds to things we have done till this part.

V. Execution Stage

The Execution Stage in the ARM pipeline is where the actual computation defined by the instruction takes place. This involves the control unit of the CPU passing the decoded information as a sequence of control signals to the relevant functional units of the CPU. These actions could include reading values from registers, and passing them to the ALU to perform mathematical or logic functions on them

A. Val2Generator

The Val2Genrator Module in an ARM processor is a Verilog module that generates the second operand for the ALU Module based on various inputs. There are three ways to generate the second value

1) *Memory Mode:* generates the output based on the signed-extended value of the 12-bit offset. We can determine this by *S* signal which is 1 if either *MEM_R_EN* or *MEM_W_EN* are enabled.

2) *32-bit Immediate:* generates the output by circularly shifting the 8 LSBs by 2 times the 4 MSBs of *shift operand*.

3) *Immediate Shift:* generates the output based on the 4 LSBs of *shift operand* which determines the number of register which would be our *valRmIn* in this module, and the 5th and 6th bits of the *shift operand* will show the type; namely, *LSL*, *LSR*, *ASR* or *ROR*.

B. ALU

This well-known module is the core of our ARM processor and supports multiple commands such as *MOV*, *MVN*, *ADD*, *ADC*, *SUB*, *SBC*, *AND*, *ORR*, and *EOR* and can also support more commands. The type of command that it should execute is determined by the *EXE_CMDIn* signal which is extracted in the Instruction-Decode Stage. Other than the first value which comes from the Register File Module and the second value which comes from the Val2Genrator Module, it has another input which is *StatusCarryIn,* and is used for *ADC* and *SBC*. Also, this module will update the value of the Status Register by determining the value of 4 bits which are *z* (zero), *v* (overflow), *n* (negative), and *c* (carry). Later the value of this register is used for checking conditions and also it's recursively used for future computations.

C. Adder

This module is used to compute the branch address for branch commands. It simply adds the sign-extended value of 24 LSBs of Instruction and *PC* value.

VI. Memory Stage

The processor interacts with the memory system in the Memory Stage. The processor reads data from memory for load instructions and writes it back to a register. For store instructions, the processor writes data to memory. At first, we implemented this stage using the Data Memory Module but then we switched to a more advanced approach using SRAM and Cache. The Data Memory Module is just a simple array of registers that takes address and value as input and then either it writes it with the falling edge of the clock or reads it asynchronously. The only thing worth mentioning here is that since it's byte-addressable for writing 32-bit values we put two zeros behind the address. SRAM and Cache are explained in their respective sections.

VII. Write-Back Stage

The Write-Back Stage in the ARM pipeline is where the results of the executed instruction are saved. If the instruction has produced a result, this result is written back to the register file. This could be the result of a computation, or data that has been fetched from memory. The Write-Back Stage is the final stage in the pipeline, and once it is completed, the instruction has been fully executed.

VIII. Top-Level Module

Now it's time to execute instructions. We created instances of all stages and connected them here. The results of the instructions when simulating on ModelSim are here, and as can be seen, the storing sorting, and then loading them are properly executed. The synthesis result of this code can be found here and also here are the results of the simulation on the FPGA board. The Dapath and connection between modules are based on this picture.

IX. Hazard-Unit

This component helps managing hazards that can occur during the pipelined execution of instructions. Hazards are situations that prevent the next instruction from executing in the following clock cycle. They can potentially lead to incorrect computation results.

There are three common types of hazards: data hazards, structural hazards, and control hazards. Data hazards occur when instructions that exhibit data dependence modify data in different stages of a pipeline. Structural hazards occur when a required resource is busy. Control hazards, also known as branching hazards, occur when the flow of instructions in the pipeline is altered.

The Hazard-Unit Module's role is to detect these hazards and take appropriate action to prevent them from affecting the correct execution of instructions., this can involve stalling the pipeline, inserting bubbles (NOPs), or using techniques like operand forwarding or out-of-order execution.

The Hazard-Unit Module can work in two different modes, forward enabled and without forwarding. The one-bit *forwardENIn* input sets

this working mode which we give as input from the Testbench module in simulation or *SW[1]* switch on FPGA for synthesis. Now consider the situation where the instruction located in the Execution Stage or the Memory Stage wants to change the value of some register and the instruction located in the Instruction-Decode Stage wants to fetch exactly the value of the same register, in these scenarios, data hazards occur. In the without forwarding mode, the Hazard-Unit Module stalls the instructions in the Instruction-Fetch Stage and the Instruction-Decode Stage, and also flushes the Instruction-Decode Stage Reg pipeline registers to generate NOP after the Instruction-Decode Stage and continues this until the said scenarios disappear or, in other words, the value inside the register is updated. But in the forward-enabled mode, it ignores these situations and allows the wrong data to be read. Still, we deliver the correct data from further stages to the Execution Stage using Forwarding-Unit that will be explained in the next section, except when we want to read the data from the memory and store it in the register, in this case, we must issue the hazard signal for one clock cycle, with doing this we have the written data from memory in Write-Back Stage and can deliver the data to the Execution Stage.

The *Hazard* signal is issued here whenever we haven't updated a value fully and we want to retrieve its value, in this case, we should use forwarding or if not possible (or not enabled) stalling stages until the correct value comes.

The control hazard is handled by *Branch_Tacken* signal and structural hazard never happens in this architecture until the SRAM Module is added. The reason for structural hazard occurrence and how to handle it are explained in the *SRAM* section.

X. Forwarding-Unit

The Forwarding Unit Module, also known as the Data Forwarding Unit, is a key component that helps to minimize the delay caused by data hazards in the pipelined execution of instructions. Data hazards occur when an instruction depends on the result of a previous instruction.

The Forwarding Unit Module's role is to bypass the result of a computation directly to the next instruction that depends on this result, without waiting for the standard sequence of writing the result back to the register file and then reading it back in the next instruction. This is known as operand forwarding or data forwarding.

As you can see in the Forwarding Unit Module, we recognize the dependency between instructions by comparing the destination signal in the Memory Stage and Write-Back Stage with *src1* and *src2* in the Execution Stage, the write-back enable signal must be issued otherwise we do not update values so there would be no data hazards. After considering the dependency, the Forwarding Unit Module sets the *selSrc1Out* and *selSrc2Out,* these two signals are used as selecting signals in *mux1* and *mux2* components in the Execution Stage that select the input values of ALU, these multiplexers were added based on this picture. Inputs of *mux1* and *mux2* multiplexers are the ALU's result in Memory Stage, Write-Back Stage, and Register File's output so we choose between them as ALU inputs when we have data hazards.

Now with these changes in the Execution Stage and Forwarding Unit, we can deliver the right data from to the Execution Stage and we don't need to wait for the changes to be applied to the register file. After adding this module the synthesis went from this to this which shows the increase in logic elements. Other ways to improve the processor's throughput are using cache (which we adopted) and branch prediction and also preventing out-of-order execution. The results of this simulation in ModelSim are here and also the simulation of this code using forwarding gave us these results, which were pretty impressive since the execution of the same bubble sort program takes about 66 percent (197 cycles / 297 cycles) of the time when we didn't use forwarding, so we have about 37 percent improvement.

XI. SRAM

It's a type of memory that retains its content as long as power is supplied. Unlike Dynamic RAM (DRAM), it doesn't need to be refreshed periodically. This makes SRAM faster, but also more expensive due to its complex internal circuitry.

For using the physical SRAM Module which is located on the board we implemented the SRAM Controller that is responsible for communicating

with SRAM and issuing its signals. We also wrote the SRAM Module to simulate the physical SRAM functionality. SRAM Module is only used for simulation and debugging.

SRAM has a 16-bit data bus and since our data is 32-bit we store the low 16-bit in one SRAM cell and the high 16-bit in the next cell, so at least we need two clock cycles for reading and writing on SRAM. An 18-bit bus for addressing is also built for the SRAM. In addition to these basses, there are also control signals in SRAM, which we only deal with *SRAM_WE_N* which shows whether the input data should be written on SRAM or not. It should be noted that all SRAM control signals are active-low.

As mentioned before, SRAM Controller is an interface between SRAM and other modules, for example, we give the 32-bit data and address to SRAM Controller and it separates data, changes and generates addresses, issues *readyOut* signals, and stores them to SRAM. SRAM Controller has six states, the IDLE state represents that SRAM is ready to work. At the *DATA_LOW* and *DATA_HIGH* states we perform memory tasks on low 16-bit and high 16-bit in order. According to the *wrEnIn* and *rdEnIn* signals, the tasks can be stored or read from SRAM. At the *DATA_UP_LOW* and *DATA_UP_HIGH* states, we read the second word from the SRAM like the two previous states, if our task is storing we don't do anything in these two states but in reading data during our cache architecture that will be explained soon we read two words from SRAM, reading the second word occurs in this two states. The last state is the *DONE* state which shows our task is done by issuing a *readyOut* signal. The *readyOut* signal shows the SRAM not performing any task and the processor can continue to perform instructions, if the *readyOut* signal is zero all registers will be stalled and the processor will stop until the memory task is finished. After using this approach the synthesis results were like this and the output was the previous states and is depicted here.

XII. Cache Memory

Cache Memory is a small-sized volatile computer memory that provides high-speed data access to a processor and stores frequently used computer programs, applications, and data. It is the fastest memory in a computer and is typically integrated into the motherboard and directly embedded in the processor or RAM.

Cache Memory provides faster data storage and access by storing instances of programs and data routinely accessed by the processor. Thus, when a processor requests data that already has an instance in the Cache Memory, it does not need to go to the main memory or the hard disk to fetch the data.

In this lab, we implemented the two-way associates cache. Each word has 32 bits and we have two words in each block, also we have 2 blocks in each row and a total of 64 rows in the cache, so in total 32*2*2*64=8192 bits or 1KB memory for caching data in this design. Each block has two words and our data is word accessible, so the first three bits of address will be the offset bits. Since our address space is 19 bits and we have 64 rows and 3 offset bits we need 10 bits for tag. In the end, we need one bit for each block named valid bit. The valid bit shows whether the data in the block is valid or not, first, we initialize these valid bits to zero, next, after reading some data from memory and storing them in the cache, we change the value of the corresponding valid bit to one. Also while writing data on memory we set this bit to zero again, of course, this procedure has changed in the advanced cache section, which we will discuss soon.

The next step of cache implementation is determining the presence or absence of data in the cache and the logic of finding that data. To do that we separate the input address into three *offset*, *index,* and *tag* parts. The location in the cache to be checked is found from the index bits. After that we check the corresponding cache tag bit with address cache bits, if they are equal our data is hit and we can return it from the cache, so to select the first or second word in the found block, we use the third bit of offset bits.

In the situation that both blocks of one row of the cache are full and to put new data in that row, we need a data replacement policy. The least recently used policy known as LRU is a common policy used in these situations, more information about this and other methods are available in #3 Reference. To implement this policy we assign one

bit to each row that shows which way was used recently. At each access to the cache, we update this bit and when replacing data, we keep the path that we used most recently and sacrifice the other.

As the last test in the cache section, we add to register named *readCnt* and *hitCnt,* on each access to the cache we increase *readCnt,* and on each hit we increase *hitCnt,* so we can calculate the cache hit rate that is equal to 23/32, you can see these number on this picture. Now we change the cache functionality. On each write to memory, we update the cache and write the new value to the cache instead of turning its valid bit to zero. By applying these changes to cache the hit rate is increased to 29/32. In this picture, you can see the result of synthesis with cache memory, and also you can compare the results from when we didn't use cache memory and when we did here and here, which shows about 30 percent improvement (1 - 296 / 421). Also, the result of the enhanced cache module both for synthesis and outcome can be found here.

You can find more information about cache memory in the ARM processor in #4 Reference.

Acknowledgment

This report was prepared by Shahriar Attar and Shahnam Feyzian for the Computer Architecture Laboratory at the ECE department in the 1401-1402 Fall semester.

References

[1] Writing synthesizable Verilog [Online]. Available: https://www.jameswhanlon.com/writing-synthesizable-verilog.html

[2] Synthesizable Verilog [Online]. Available: https://www.muchen.ca/documents/CPEN311/SS3.html

[3] Cache Replacement Policy [Online] Availabe: https://en.wikipedia.org/wiki/Cache_replacement_policies

[4] Cache Memory [Online] Available: https://blog.feabhas.com/2020/10/introduction-to-the-arm-cortex-m7-cache-part-1-cache-basics/