UNIVERSITY OF TEHRAN

JANUARY 2024

Computer Architecture Laboratory

Report

PREPARED BY

Mohammadreza Shokouhi and
Mohammadmehdi Abdolhoseini

Contents

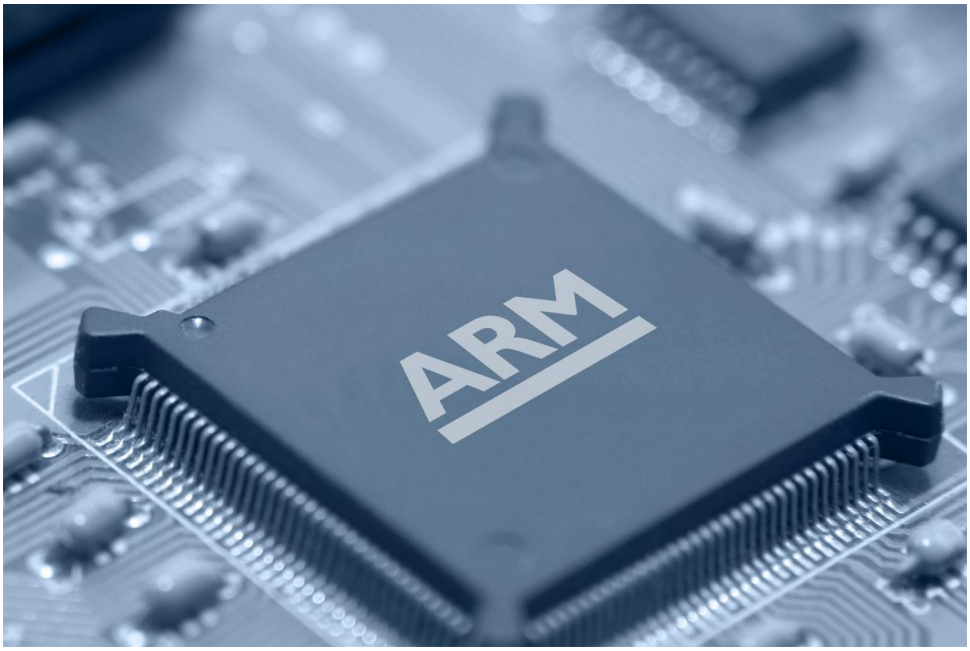


Figure (1.1) *ARM processor*

1 Introduction

1.1 Why ARM?

Advanced RISC Machine (ARM) Processor is considered to be a family of Central Processing Units that are used in music players, smartphones, wearables, tablets, and other consumer electronic devices.

Advanced RISC Machines create the ARM processor architecture, hence the name ARM. This needs very few instruction sets and transistors. It has very small in size. This is the reason that it is a perfect fit for small-size devices. It has less power consumption along with reduced complexity in its circuits.

The applications of the ARM Process start with getting knowledge of the ARM Processor's history. Before ARM, x86 processors were used, which were launched in 1978. Whenever we remove the predefined instructions like complex instructions and hard-to-implement instructions, the remaining instructions take less power and pace and run faster, this is called Reduced Instruction Set Computer (RISC) Architecture. x86 is a Complex Instruction Set Architecture (CISC).

One of the most common electronic architectural designs in the market is Advanced RISC Machine Architecture, even better than x86, which is very common in the server market. ARM Architecture is widely used in smartphones, normal phones, and also in laptops. Though x86 processors have optimized performance ARM Processor gives cost-effective processors with small size, takes less power to run, and also gives better battery life.

ARM Processor is not only limited to mobile phones but is also used in Fugaku, the world's fastest supercomputer. ARM Processor also gives more feasibility to designs of hardware designers and also gives control to designer's supply chains.

Table (1.1) *Main instructions of our ARM processor*

1.2 What we have done

During the course, we implemented *ARM968E-S* from families *ARM9* and *ARM9E* as described below:

- It has 13 main instructions.
- At each stage, codes are primarily written in *ModelSim*. Then they get debugged and verified in the same environment as long as they meet our needs and run as desired. Results are shown in the report subsequently. At this point, a top-level test bench is required since we are just doing pure simulations.
- When codes are verified, they are transferred to *Quartus II* which is an intermediate software that programs our FPGA board *Cyclone II-EP2C35F672C6*. Here, some changes must be applied on the codes. For instance, virtual clock and reset command are replaced with real clock of the FPGA and a button on the board respectively.
- When our processor is completed, we try to enhance its performance by the operand forwarding method. As we'll see, the internal memory of our processor is not large enough for many purposes. To overcome this problem, one needs to utilize an external memory of the board. We will do this in *SRAM* section.

2 Instruction Fetch Stage

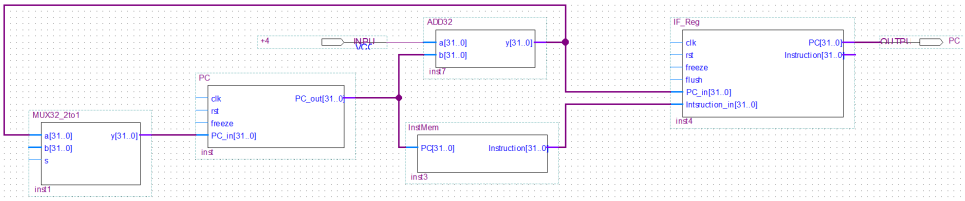
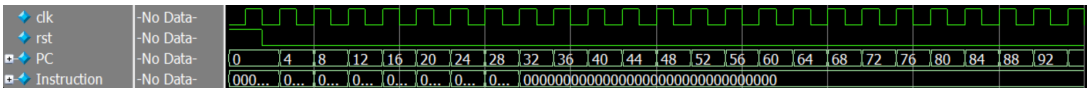
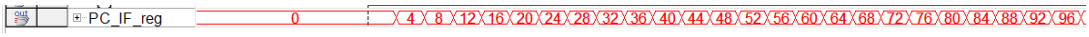


Figure (2.1) *Implementation of IF Stage*

In this stage, instructions are fetched from the instruction memory as shown in figure 2.1. *PC* is a 32-bit register that points to the instruction at hand. The 32-bit adder is used for counting the instructions. It adds +4 to the number registered in *PC* which is equivalent to reading the next instruction from the memory. At this point, the multiplexer does nothing as we have'nt implemented other stages yet. But, basically, it is used for branching. Note that the value of *freeze*, *Branch_taken* and *BranchAddr* are assigned to zero.



(a) *ModelSim*



(b) *SignalTab*

Figure (2.2) *Outputs*

As it can be seen, *PC* reads instructions from the memory one by one and goes ahead. Flow summary of this implementation is shown in figure 2.3.

Flow Summary	
Flow Status	Successful - Mon Oct 23 10:26:53 2023
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	ARM
Top-level Entity Name	ARM
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	74 / 33,216 (< 1 %)
Total combinational functions	48 / 33,216 (< 1 %)
Dedicated logic registers	69 / 33,216 (< 1 %)
Total registers	69
Total pins	66 / 475 (14 %)
Total virtual pins	0
Total memory bits	0 / 483,840 (0 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

Figure (2.3) *Flow summary of the simulation(IF)*

3 Instruction Decode Stage

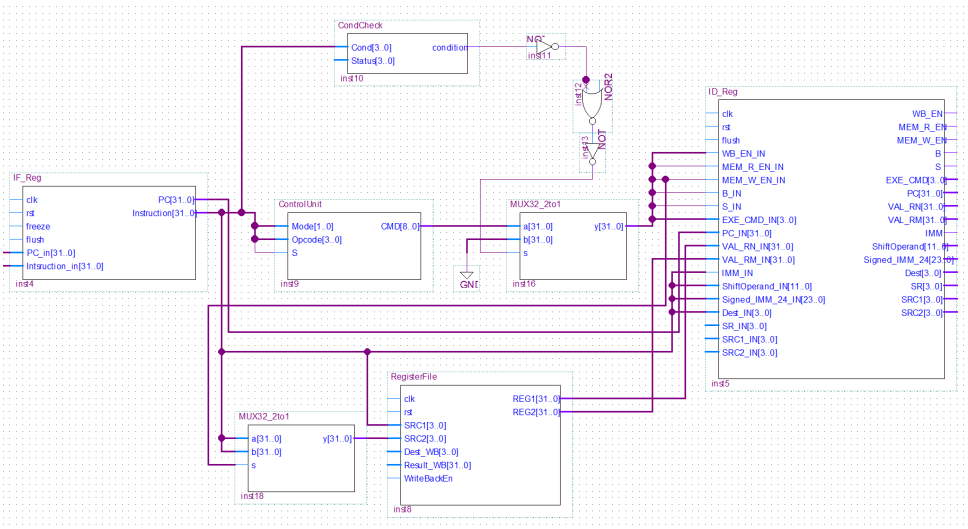


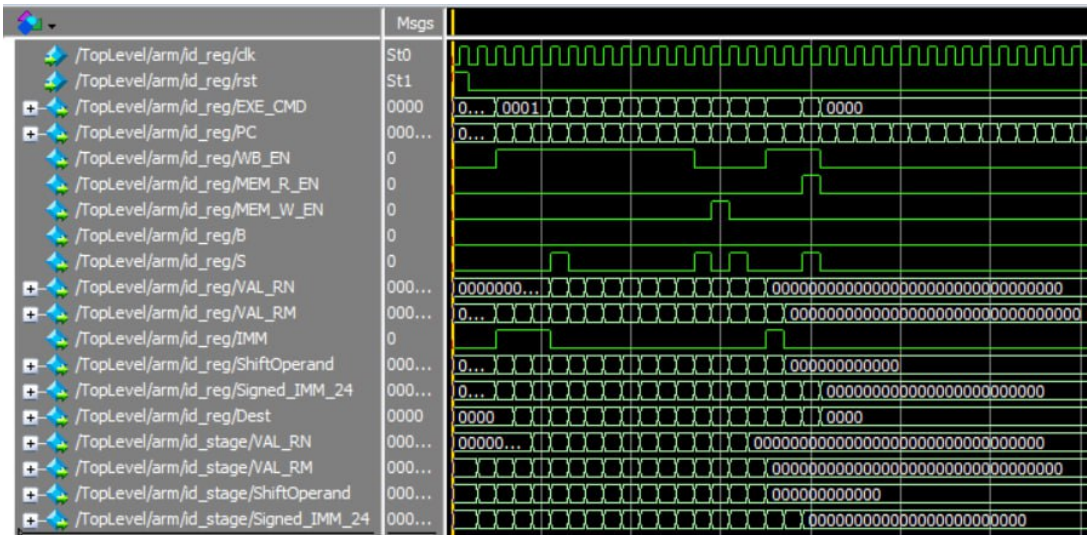
Figure (3.1) *Implementation of ID stage*

In this stage, instructions are decoded into several components to be identified by the controller unit. Then the corresponding commands are generated and transferred to the next stage.
The controller unit generates the following signals:

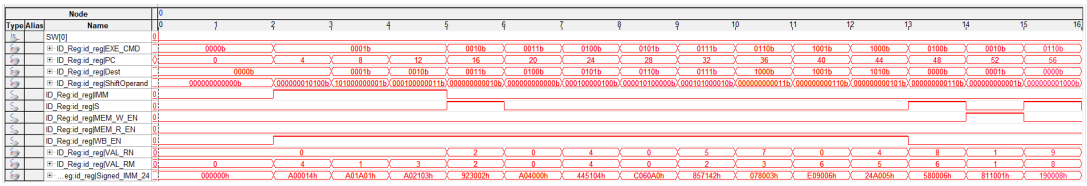
- *Execution Command*: This command is mainly used for computational sub-instructions that are performed by arithmetic logic unit (*see table 4.1*).
- *mem_read* and *mem_write*: These signals enable reading and writing from the main memory.
- *WB_En*: This signal aims to write back data to the register file.
- *B*: Used for conditional and branching instructions.
- *S*: Updates the status register. Note that, this signal is zero for all computational instructions with *mode*=0.

Note that, these signals are generated if and only if the *condition_check* module verifies the satisfaction of the condition (please note the multiplexer after this module). For instance, in assembly language, *ADDEQ* means "add if the equal condition is satisfied" and the key components of this condition are generated by the status register. Since it has'nt been implemented yet, we expect the instructions to be performed unconditionally.
The register file consists of 16 32-bit registers used for general purpose registers. Note that the *PC* register is in fact the 16^{th} register of the register file, but for the sake of simplicity, it is placed in the instruction fetch stage. Read and write operations are performed by 2 asynchronous ports being sensitive to the negative edge of the clock. Note that, instruction *STR* takes the destination register as the 2^{nd} port of the register file otherwise *Rm* is propagated.

The results of simulation are displayed in figure 3.2. Due to the given test bench, non-memory reference instructions are decoded correctly at the same frequency of the clock (since hazards are ignored). The flow summary of the simulation are shown in figure 3.3.



(a) *ModelSim*



(b) *SignalTab*

Figure (3.2) *Outputs*

Flow Summary	
Flow Status	Successful - Mon Nov 13 10:48:32 2023
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	ARM
Top-level Entity Name	ARM
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	3,250 / 33,216 (10 %)
Total combinational functions	1,325 / 33,216 (4 %)
Dedicated logic registers	2,930 / 33,216 (9 %)
Total registers	2930
Total pins	19 / 475 (4 %)
Total virtual pins	0
Total memory bits	18,816 / 483,840 (4 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

Figure (3.3) *Flow summary of the simulation(IF+ID)*

4 Execution and Write-back Stages

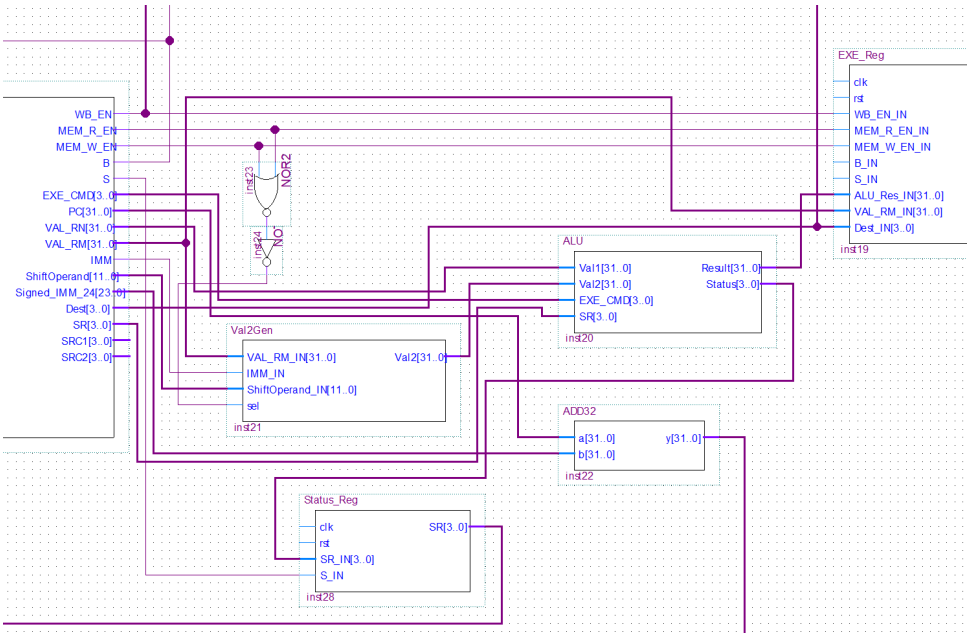


Figure (4.1) *Implementation of EXE stage*

In execution stage, as its name implies, commands are executed. The computational core of the processor, *ALU* is located here. It takes two inputs, the execution command generated by the controller unit and contents of the status register and returns the computed result and the modified status of the status register[1]. Its first input is simply the contents of register *Rn* in register file whereas its second input is generated by the *Val2Gen* module.

Table (4.1) *Sub-instructions of ALU*

Instruction	ALU Command	Operation
MOV	0001	in_2
MVN	1001	$\bar{in_2}$
ADD	0010	$in_1 + in_2$
ADC	0011	$in_1 + in_2 + C$
SUB	0100	$in_1 - in_2$
SBC	0101	$in_1 - in_2 - \bar{C}$
AND	0110	$in_1 \wedge in_2$
ORR	0111	$in_1 \vee in_2$
EOR	1000	$in_1 \oplus in_2$
CMP	0100	$in_1 - in_2$
TST	0110	$in_1 \wedge in_2$
LDR	0010	$in_1 + in_2$
STR	0010	$in_1 + in_2$
BBB	-	-

[1] Its contents is updated only if $S = 1$.

Val2Gen module takes 3 inputs: *Val_Rm*, *imm* and *ShiftOperand*. *ShiftOperand* is made up of two components *immed_8* and *rotate_imm* which are $ShiftOperand[7:0]$ and $ShiftOperand[11:8]$ respectively. This module works as follows: if $imm = 1$ the output is *immed_8* rotated by $2 \times rot\_imm$ otherwise *Val_Rm* shifted due to *shift_imm*. This is when *mem_r_en* and *mem_w_en* are not enabled. In other cases, *ShiftOperand* is simply sign-extended to 32-bit.

Table (4.2) *Immediate shifts*

Shift	Description	Value
LSL	logical shift left	00
LSR	logical shift right	01
ASR	arithmetic shift right	10
ROR	rotate right	11

Write-back stage is nothing but a multiplexer as shown in figure 4.2. If *mem_r_en* is low, the result of the ALU is propagated otherwise the data read from the main memory (which is not implemented, yet).

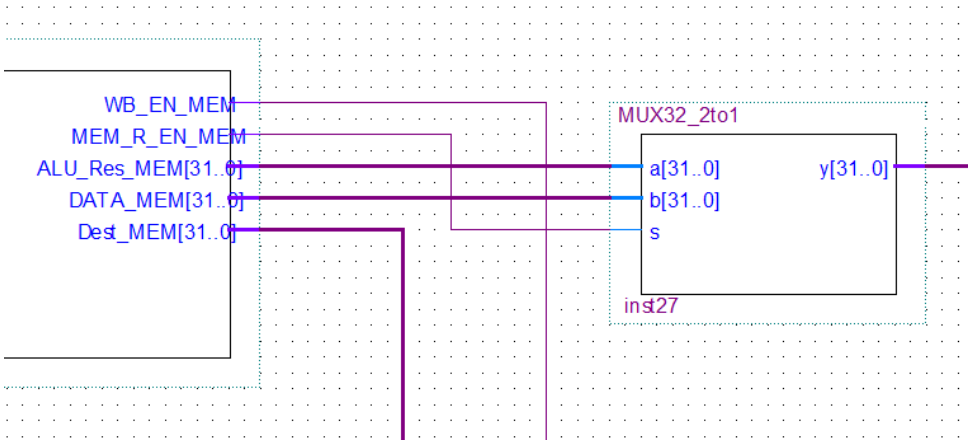
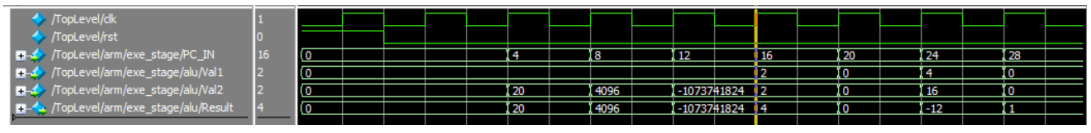
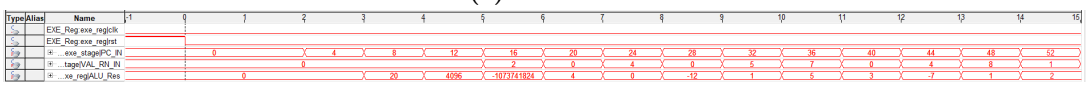


Figure (4.2) *Implementation of WB stage*

At this point, we expect non-memory reference instructions to be carried out properly. In figure 4.3, the operational functionality of ALU for some instructions is demonstrated. The corresponding flow summary of the simulation is shown in figure 4.4.



(a) *ModelSim*



(b) *SignalTab*

Figure (4.3) *Outputs*

Flow Summary	
Flow Status	Successful - Mon Nov 20 11:26:10 2023
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	ARM
Top-level Entity Name	ARM
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	3,309 / 33,216 (10 %)
Total combinational functions	1,616 / 33,216 (5 %)
Dedicated logic registers	2,663 / 33,216 (8 %)
Total registers	2663
Total pins	19 / 475 (4 %)
Total virtual pins	0
Total memory bits	16,768 / 483,840 (3 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

Figure (4.4) *Flow summary of the simulation(IF+ID+EXE+WB)*

5 Memory Stage and Hazard Detection Unit

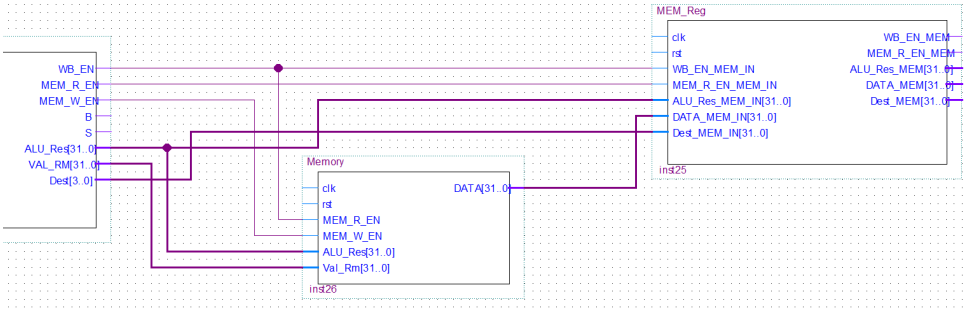


Figure (5.1) *Implementation of MEM stage*

Memory stage consists of only the main memory. The main memory is made up of 64 32-bit registers, hence *64 Kbit*. Reading and writing are performed combinationally and sequentially respectively. Note that, addressing starts from 1024 since the instruction memory is in fact a part of the main memory. As it can be seen, the input address is calculated by the ALU.
To address hazards, we primarily need to know what they actually are:

- **Structural Hazard**: A structural hazard arises when two (or more) pipelined instructions need the same resource. Instructions must therefore be carried out in series rather than parallel for a segment of the pipeline. To overcome this issue, one must sensitize the register file to the negative edge of the clock, which is already done.

- **Control Hazard**: A control hazard, also known as a branch hazard, is a type of pipeline hazard that occurs when the pipeline incorrectly predicts a branch, leading to unnecessary instructions entering the pipeline. To resolve this issue we have already added *flush* signals to the processor.

- **Data Hazard (RAW)**: Occurs when an instruction depends on the result of previous instruction and that result of instruction has not yet been computed. Whenever two different instructions use the same storage. The location must appear as if it is executed in sequential order. Here, we only deal with read-after-right data hazards. To handle this type of hazards, we have implemented a hazard detection unit shown in figure 5.2. It simply performs the following command:

$$
\begin{aligned}
hazard = \\
&((SRC1 == Dest\_EXE) \wedge WB\_EN\_EXE) \vee \\
&((SRC1 == Dest\_MEM) \wedge WB\_EN\_MEM) \vee \\
&((SRC2 == Dest\_EXE) \wedge WB\_EN\_EXE \wedge Two\_SRC) \vee \\
&((SRC2 == Dest\_MEM) \wedge WB\_EN\_MEM \wedge Two\_SRC) ? 1'b1 : 1'b0
\end{aligned}
$$

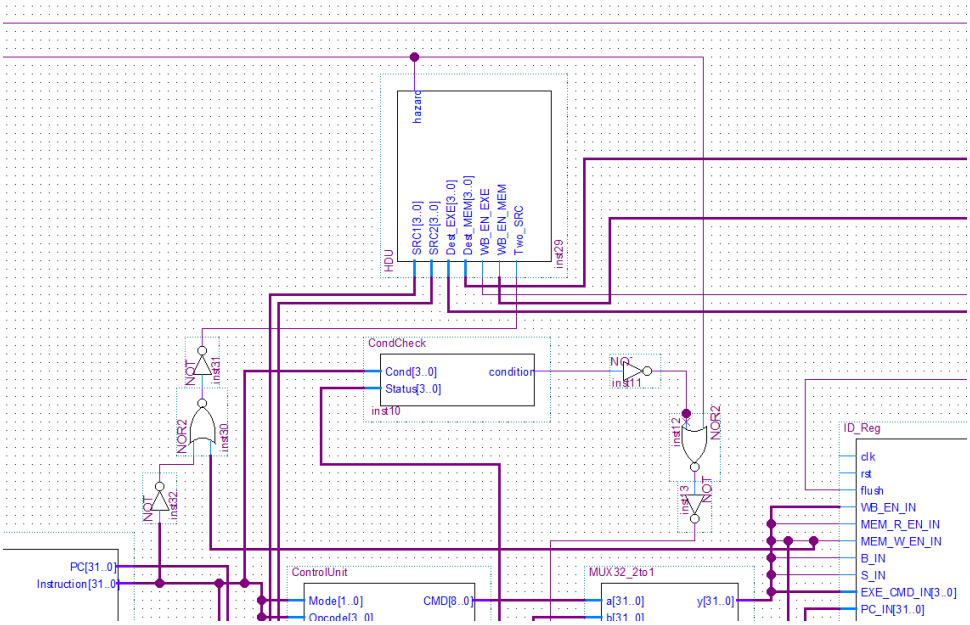
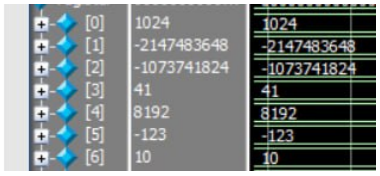


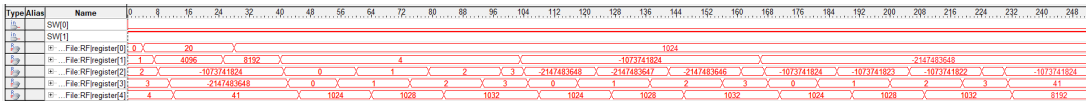
Figure (5.2) *Implementation of hazard detection unit*

By putting everything together, we complete our *ARM* processor. Results of the simulations are shown in figure 5.3 that are fully in agreement with the given test bench.

```
40.  32'b1011_10_1_0_11111111111111111111110011   ; //BLT      #-13
41.  32'b1110_01_0_0100_1_0000_0001_000000000000; //LDR        R1 ,[R0],#0     //R1 = -2147483648
42.  32'b1110_01_0_0100_1_0000_0010_000000000100; //LDR        R2 ,[R0],#4     //R2 = -1073741824
43.  32'b1110_01_0_0100_1_0000_0011_000000001000; //LDR        R3 ,[R0],#8     //R3 = 41
44.  32'b1110_01_0_0100_1_0000_0100_000000001100; //LDR        R4 ,[R0],#12    //R4 = 8192
45.  32'b1110_01_0_0100_1_0000_0101_000000010000; //LDR        R5 ,[R0],#16    //R5 = -123
46.  32'b1110_01_0_0100_1_0000_0110_000000010100; //LDR        R6 ,[R0],#20    //R4 = 10
```



(a) *ModelSim*



(b) *SignalTab*

Figure (5.3) *Outputs*

Flow Summary	
Flow Status	Successful - Mon Dec 18 12:05:28 2023
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	ARM
Top-level Entity Name	ARM
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	5,423 / 33,216 (16 %)
Total combinational functions	3,216 / 33,216 (10 %)
Dedicated logic registers	3,997 / 33,216 (12 %)
Total registers	3997
Total pins	19 / 475 (4 %)
Total virtual pins	0
Total memory bits	168,960 / 483,840 (35 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

Figure (5.4) *Flow summary of the simulation(ARM)*

6 Operand Forwarding Optimization

As observed in previous section, a data hazard can lead to a pipeline stall when the current operation has to wait for the results of an earlier operation which has not yet finished. Obviously, this method causes performance deficits. In forwarding method, we try to transfer the results from the earlier operation to be used for the current operation. Generally speaking, this is done by adding some simple components to the processor. As shown in figure 6.1, we only need 2 multiplexers to select the proper contents from registers and a forwarding unit to control the data flow of the operations.

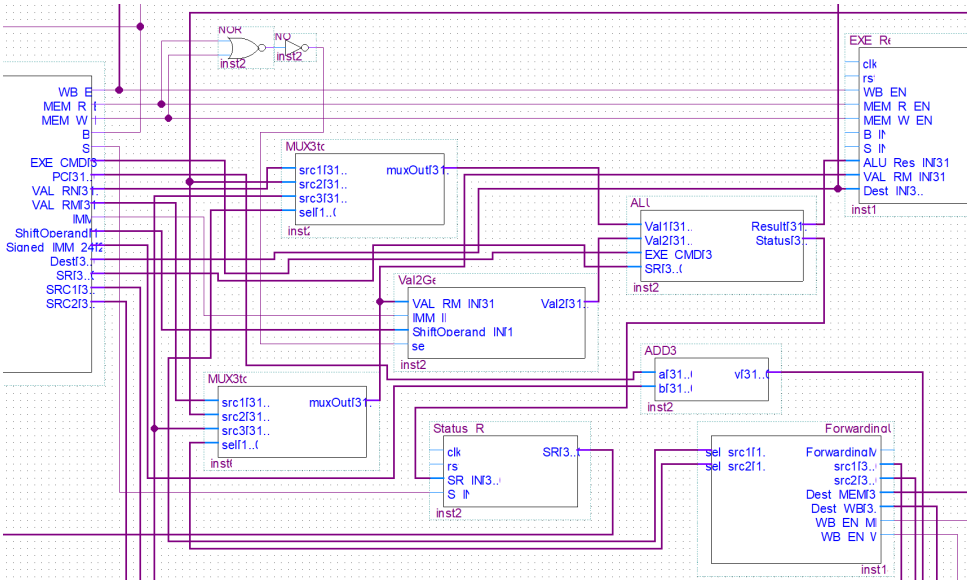


Figure (6.1) *Block diagram of the modified EXE stage*

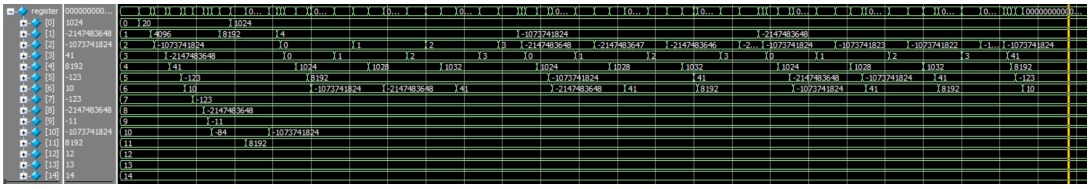
Clearly, data after the execution stage are hold in the execution register, then they are ready to use for further operations. To do so, we directly transfer the data from the memory stage to the execution stage. These data would be on the second port of the 3-to-1 multiplexers. Moreover, sometimes, we need to make a use of the data that are to be written in the register file. These data are often get ready in the write-back stage and can be used for the current operation in the execution stage instead of stalling the pipelines for a clock cycle. Hence, the third port of the multiplexers is dedicated to these data.

To have control over data, we have implemented a control unit named *forwarding unit* that checks the possibility use of the forwarding method. It takes the *forwarding mode*, sources of the register file, destinations and write-back signals in the memory and the write-back stages. Logically, it works as follows:

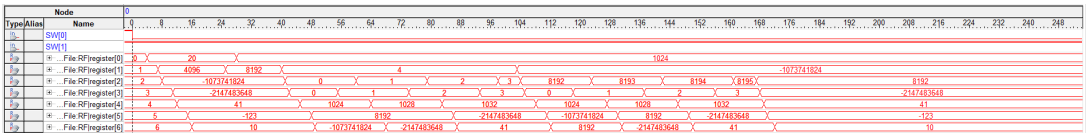
$$sel\_src_1 = (ForwardingMode == 0)?\,00 :$$
$$((src1 == Dest\_MEM \wedge WB\_EN\_MEM == 1)?\,01 :$$
$$((src1 == Dest\_WB \wedge WB\_EN\_WB == 1)?\,10 : 00))$$

The same is done for the second source.

Now our processor is able to work in two different modes: *non-forwarding* and *forwarding* modes. Results of the simulation are shown in figure 6.2. As it can be seen, the processor has become *1.35* times more efficient. We will evaluate the results more precisely later.



(a) *ModelSim*



(b) *SignalTab*

Figure (6.2) *Outputs*

Flow Summary	
Flow Status	Successful - Mon Dec 25 09:24:43 2023
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	ARM
Top-level Entity Name	ARM
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	6,539 / 33,216 (20 %)
Total combinational functions	3,572 / 33,216 (11 %)
Dedicated logic registers	5,066 / 33,216 (15 %)
Total registers	5066
Total pins	19 / 475 (4 %)
Total virtual pins	0
Total memory bits	118,272 / 483,840 (24 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

Figure (6.3) *Flow summary of the simulation (Forwarding)*

7 Use of Static Random-access Memory

As per our knowledge, FPGAs have small on-chip memory storage and in our processor, the size of the internal memory is *64 Kbit* which is not sufficient for most applications. To address this problem, we are to use an external memory on the board that is of size *512 Kbit*. Note that the address bus is 18-bit. To do so, the main memory is moved out of the processor as shown in figure 7.1. Obviously, one needs an unit to control the data flow of the memory (*MemoryController*). Reading and writing are performed

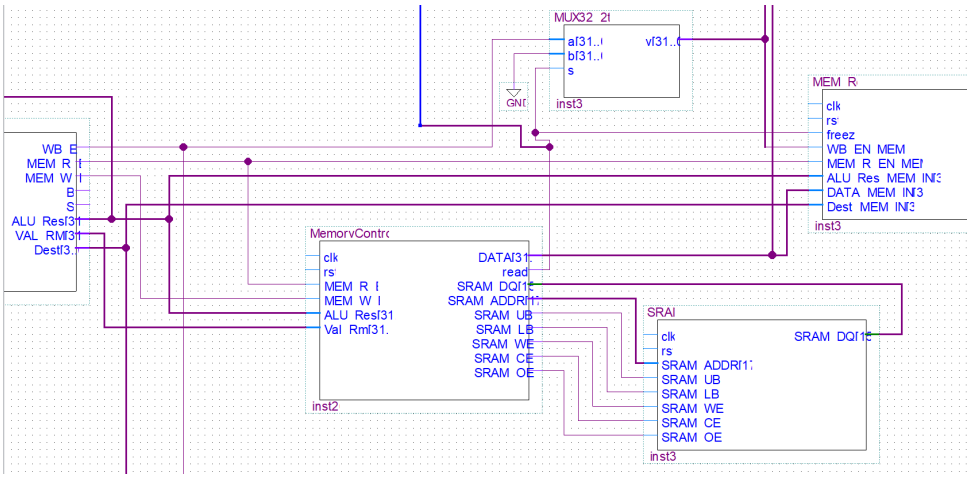


Figure (7.1) *Block diagram of the modified MEM stage*

synchronously. These operations can be executed in a clock cycle, but we take the memory access time to be 6 clock cycles. There are 5 control signals as written in table 7.1. For the sake of simplicity, *SRAM_CE_N*, *SRAM_LB_N*, *SRAM_UB_N* and *SRAM_OE_N* are always enabled.

Table (7.1) *Control signals of the SRAM*

Verilog name	Pin	Description
SRAM_WE_N	PIN_AE_10	SRAM Write Enable
SRAM_OE_N	PIN_AD_10	SRAM Output Enable
SRAM_UB_N	PIN_AF9	SRAM High-byte Data Mask
SRAM_LB_N	PIN_AE9	SRAM Low-byte Data Mask
SRAM_CE_N	PIN_AC11	SRAM Chip Enable

Writing into the memory is done as follows:

1. On the first rising edge of the clock, low-byte data and low-byte address are put on the port of the SRAM. Here, SRAM_WE_N is low.
2. The same is done for the high-byte components on the next rising edge of the clock.
3. Nextly, SRAM_WE_N is set to 1. Here, the write operation is done.
4. In all the first five clock cycles, *ready* is low and all the intermediate registers are stalled. In the sixth clock cycle, *ready* is set to 1.

To read from the memory, the following procedure must be carried out:

1. In the first clock cycle, low-byte address is put on the port of the SRAM. Moreover, SRAM_WE_N is set to 1.

2. Then low-byte data is read from the memory in the next cycle. Here, high-byte address is given to the SRAM.

3. In the next cycle, high-byte data is extracted.

4. Note that, in this operation, SRAM_DQ is high-impendence.

Please note the multiplexer placed in this stage. This component is used to enable write-back operation if the *ready* signal is high.

Results are shown in figure 7.2. As it can be seen, the efficiency of the processor is significantly decreased as a result of the use of an external memory. The same simulation is done in forwarding mode (*see figure 7.3*). It is worth noting that, when we were trying to simulate the processor in *Quartus*, we faced some incorrect results shown in figure 7.4. Nevertheless, by increasing the size of the memory the problem was resolved.

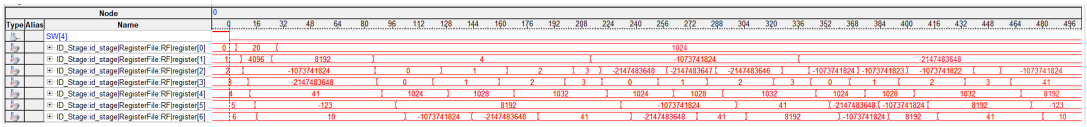
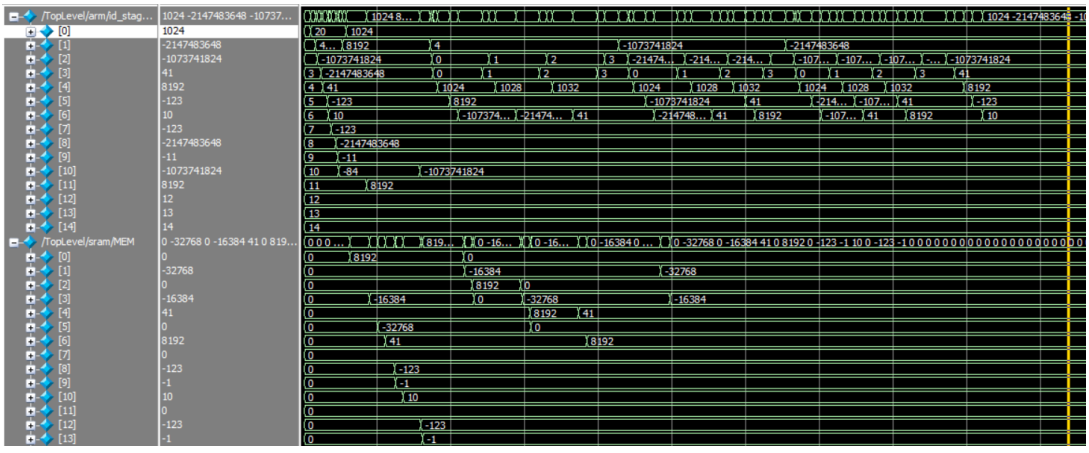
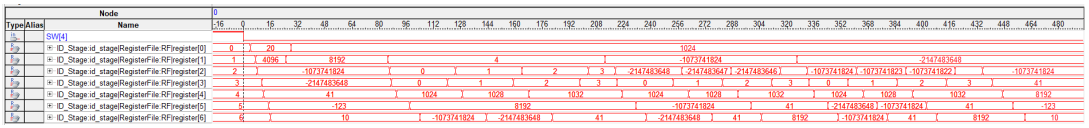


Figure (7.2) *SignalTab output (without forwarding)*



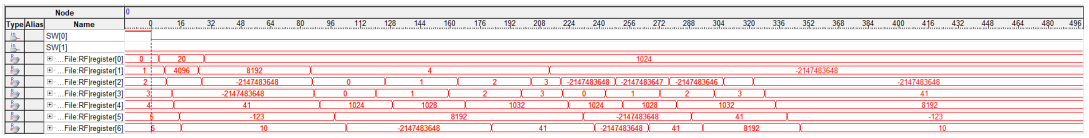
(a) *ModelSim*



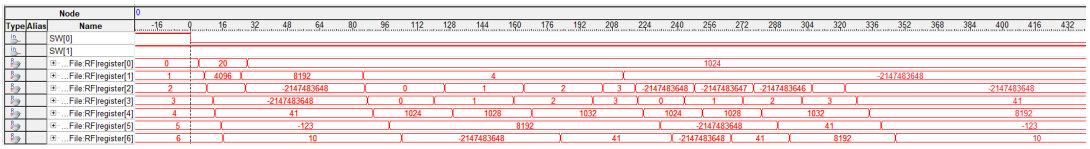
(b) *SignalTab*

Figure (7.3) *Outputs (with forwarding)*

(a) *Try 1*



(b) *Try 2*



(c) *Try 3*

Figure (7.4) *Unsuccessful tries*

Flow Summary	
Flow Status	Successful - Mon Jan 08 10:49:43 2024
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	ARM
Top-level Entity Name	ARM
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	7,034 / 33,216 (21 %)
Total combinational functions	4,103 / 33,216 (12 %)
Dedicated logic registers	5,062 / 33,216 (15 %)
Total registers	5062
Total pins	59 / 475 (12 %)
Total virtual pins	0
Total memory bits	462,848 / 483,840 (96 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

(a) *Without forwarding*

Flow Summary	
Flow Status	Successful - Mon Jan 08 09:56:52 2024
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	ARM
Top-level Entity Name	ARM
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	7,561 / 33,216 (23 %)
Total combinational functions	4,169 / 33,216 (13 %)
Dedicated logic registers	5,963 / 33,216 (18 %)
Total registers	5963
Total pins	58 / 475 (12 %)
Total virtual pins	0
Total memory bits	232,448 / 483,840 (48 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

(b) *With forwarding*

Figure (7.5) *Flow summary of the simulation (SRAM)*

8 Conclusion

Overall results from the simulations are summarized in the table below:

Features		**Completion Time**	**Combinational**	**Dedicated Logic**	**Total Logic**
Forwarding	SRAM	(ns)	**Functions**	**Registers**	**Elements**
✗	✗	232	3216	3997	5423
✓	✗	172	3572	5066	6539
✗	✓	480	4103	5062	7034
✓	✓	460	4169	5963	7561

As it can be seen, utilizing the forwarding method improves the functionality of the processor by factor of *1.35*. But the total hardware cost is increased *20%*. Consequently, the performance per cost is increased by factor of *1.12*.

Clearly, using the SRAM, lowers the performance of the processor significantly (by factor of *0.48*), and using the forwarding method does'nt improve it remarkably. Moreover, the hardware cost is increased by *1.3*. To resolve these deficiencies, one way is to use *cache*, but we have'nt gone through it in this course.