FACULTY OF ENGINEERING AND TECHNOLOGY
DEPARTMENT OF COMPUTER ENGINEERING

Computer Architecture - ENCS4370

Pipelined Processor

Prepared by: Rawan Yassin 1182224 Alaa Zuhd 1180865
Instructor: Dr. Aziz Qaroush

Section: 1

BIRZEIT
June 3, 2021

1 ABSTRACT

In this project a pipelined 16-bit processor was designed and tested with the help of the Logisim simulator.
A RISC processor with eight 16-bit general purpose registers and a 16-bit program counter register was implemented, supporting 16 bits instructions with three different formats (R-type, I-type, and J-type) and different addressing modes.
A five-stage pipeline-datapath and its control logic were implemented completely from scratch. Detection of data dependencies among different instruction was implemented, data forwarding was achieved successfully and the minimum number of stall cycles was provided. Following, all details implemented and encountered in the process of implementing this datapath along with its logic will be illustrated in details and supported by various test cases.

Contents

2 THEORY

In this section, the datapath which is a critical component in building any computer is described briefly as well as an explanation of what is meant by a pipeline architecture.

2.1 Datapath

A datapath is a set of functional units that handle data processing tasks, such as arithmetic logic units or multipliers. It is part of the central processing unit(CPU), along with the control unit. Multiplexers can be used to combine numerous datapaths together to create a larger datapath. Due to the latter, the final datapath for any architecture will be implemented in an incremental approach by considering at each step a set of the instruction and adapting the datapath to support it. Also, more components can be added into the datapath as we are moving from an architecture to another, like moving from the single cycle architecture into the pipeline architecture. However the main components at any datapath are: the PC, the Instruction memory and the ALU as well as to the register file. Theses components are enough for the R-type instruction, and by moving into the J-type and the I-type instruction, we further need data memory and sign extenders. Moving to the pipeline architecture requires more components, mainly the buffers between each two stages, forwarding units and hazard detection units.

2.2 Pipelinging Architecture

Instruction pipelining is a technique of the instruction-level parallelism, that divides the datapath as well as the instructions into many stages (in our case 5 stages) and keep every stage always busy in implementing a part of an instruction (at the same time every stage will be working on a sub-instruction of different instructions), hence the benefit of this architecture is to finish one instruction at each cycle, with ignoring the fill cycle, which makes the ideal CPI equal to 1. As illustrated above, moving to the pipelining architecture requires adding more components, like the buffers between each two stages to store the values of the current instruction from the previous stage and moving them to the next stage, as well as the forwarding units that forwards the needed data from one stage into another without the needs to wait for these data to be written into the register file, and also hazard detection unit that detect when we need to stall the pipelining because the forwarding is not possible to minimize the stall cycles.

3 PROJECT BACKGROUND

This project aims to implement and test a 16-bit pipelined processor. It was built starting by building the basic components followed by connecting these different components together and then optimizing it.
Our design is characterised by a file register with 8 general purpose registers, each with 16-bit, with the first in the register file (R0) being hardwired. It also has a 16-bit PC register. The instruction set architecture supports three different format types as follows, in which each instruction is 16-bit obtained as a full word from the instruction memory :

1. R-type Format: the instruction here is divided into 4 main parts, 4-bit opcode (Op), 3-bit register numbers (Rs, Rt, and Rd), and 3-bit function field (funct). The Rs and Rt specify the two source register numbers, and Rd specifies the destination register number. 0 is reserved as the opcode for this type with the function field specifying at most 8 different functions.

2. I-type Format: the instruction is divided into 4-bit opcode (Op), 3-bit register number (Rs and Rt), and 6-bit signed immediate constant. Rs specifies a source register number, and Rt can be a second source or a destination register number.

3. J-type Format: the instruction is divided into 4-bit opcode (Op) and 12-bit immediate constant.

For our datapath, the following instruction are being supported.

Table 1: Instruction Encoding

Instr	Meaning	Encoding				
AND	Reg(Rd) = Reg(Rs) & Reg(Rt)	Op = 0000	Rs	Rt	Rd	f = 111
OR	Reg(Rd) = Reg(Rs) | Reg(Rt)	Op = 0000	Rs	Rt	Rd	f = 110
CAS	Reg(Rd) = Max[Reg(Rs) , Reg(Rt)]	Op = 0000	Rs	Rt	Rd	f = 101
Lws	Reg(Rd) = Mem[Reg(Rs) + Reg(Rt)]	Op = 0000	Rs	Rt	Rd	f = 100
ADD	Reg(Rd) = Reg(Rs) + Reg(Rt)	Op = 0000	Rs	Rt	Rd	f = 011
SUB	Reg(Rd) = Reg(Rs) – Reg(Rt)	Op = 0000	Rs	Rt	Rd	f = 010
SLT	Reg(Rd) = Reg(Rs) < Reg(Rt)	Op = 0000	Rs	Rt	Rd	f = 001
JR	PC = Reg(Rs)	Op = 0000	Rs	000	000	f = 000
ANDI	Reg(Rt) = Reg(Rs) & $Immediate^6$	Op = 0001	Rs	Rt	$Immediate^6$	
ORI	Reg(Rt) = Reg(Rs) | $Immediate^6$	Op = 0010	Rs	Rt	$Immediate^6$	
ADDI	Reg(Rt) = Reg(Rs) + $Immediate^6$	Op = 0011	Rs	Rt	$Immediate^6$	
Lw	Reg(Rt) = Mem(Reg(Rs) + Imm^6)	Op = 0100	Rs	Rt	$Immediate^6$	
Lbu	Reg(Rt [7:0]) = Mem(Reg(Rs) + Imm^6) Reg(Rt [15:8]) = 00000000	Op = 0101			$Immediate^6$	
Lb	Reg(Rt [7:0]) = Mem(Reg(Rs) + Imm^6) Reg(Rt [15:8]) = sign bit	Op = 0110			$Immediate^6$	
Sw	Mem(Reg(Rs) + Imm^6) = Reg(Rt)	Op = 0111	Rs	Rt	$Immediate^6$	
Sb	Mem(Reg(Rs) + Imm^6) = Reg(Rt [7:0])	Op = 1000			$Immediate^6$	
BEQ	Branch if (Reg(Rs) == Reg(Rt))	Op = 1001	Rs	Rt	$Immediate^6$	
J	PC = PC + Immediate	Op = 1010	$Immediate^{12}$			
JAL	R7 = PC + 1, PC = PC + Immediate	Op = 1011	$Immediate^{12}$			
LUI	R1 = $Immediate^{12}$ << 4	Op = 1100	$Immediate^{12}$			

A five-stage pipeline supporting all mentioned above instructions was designed and implemented. The control units for detecting hazards and data dependencies were added. Stall cycles were reduced to zero in most of the cases and to one as a worst scenario with the help of the different implemented forwarding units.

4 DESIGN AND IMPLEMENTATION

In this section, each component in the implemented datapath will be illustrated in details as well as all the needed control units.

4.1 Combinational Elements

In this part: ALU, comparator, used multiplexers as well as the used extenders will be illustrated.

4.1.1 ALU

Referring to the given table of instructions, 7 different ALU operations are needed in order to execute all the supported instructions. These are:

1. Anding: implementing simply with an AND gate, accepting two 16-bit inputs.
2. Oring:implementing simply with an OR gate, accepting two 16-bit inputs.
3. Max: implemented simply with this logic, max = (A. b_out)+(B.b_out), as if A (first input) is greater, the b_out (borrow out signal of subtraction) would be zero, else it would be one.
4. Addition: implemented with the help of the adder provided by Logisim, accepting two inputs each 16-bit and a zero carry-in bit.
5. Subtraction: implemented with the help of a subtractor provided by Logisim, accepting two inputs each 16-bit and a zero borrow-in bit.
6. Set On Less Than: it is the zero-extention of the borrow out signal generated by the subtractor, which is 1 if the first input is less than the second.
7. Shifting: implemented with the help of the ready shifter provided by Logisim, 4 bits is the needed amount for the supported logical left shift.

Following is ALU unit implemented circuit in Logisim with the mentioned above functions

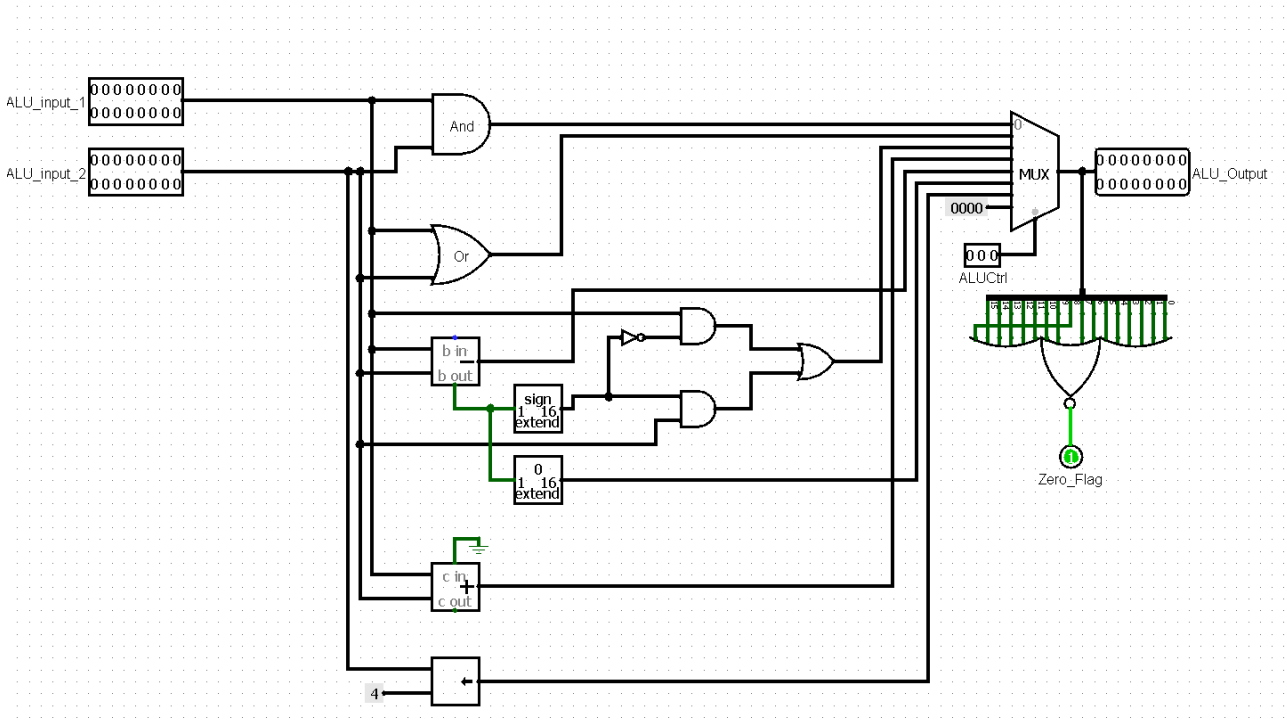

Figure 1: ALU Unit

4.1.2 Extender_6

This block was implemented to extend the 6-bit immediate, obtained from the I-type instruction, into a 16-bit number with the capability for doing the extension with the sign extension or the zero extension based on the ExtOP signal, which is derived from the control unit. Following is the implemented extender unit, in which a multiplexer is used to choose between the sign extended or zero extended immediate with the help of the ExtOP signal.

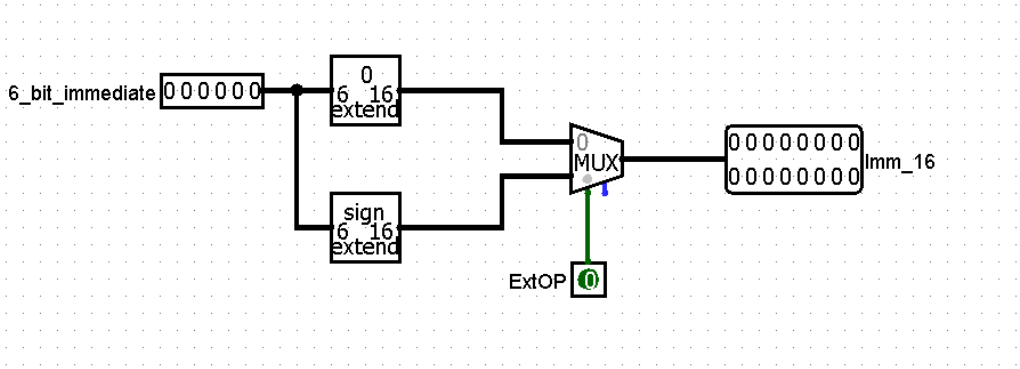

Figure 2: Extender_6 Unit

4.1.3 Extender_12

This block was implemented to extend the 12-bit number, obtained from the J-Type instruction, into a 16-bit number by using the sign extention since all the J-Type instructions require a sign extension.

Figure 3: Extender_12 Unit

4.1.4 Destination_Register Multilplexer

This multiplexer accepts four main inputs, the value of one, seven, the register number of Rt in the decoded instruction, and the register number of Rd in the decoded instruction. It has the RegDst signal generated by the control unit as the selection signal. It selects the needed register for the instruction being executed. It is noticed in our path, that the output of this multiplexer is followed by another multiplexer, having zero as the first input and the output of the first mux as its second input and The regWrite as its selection signal. The importance of the second multiplexer is noticed when the stall cycle is introduced, at which the RegDes singal is zero and the RegWrite is also zero, yielding that the operation result is to be stored in register zero, which is hardwired and as a result the stall is implemented as a concept.

4.1.5 JAL_Multiplexeer

This multiplexer chooses the correct result for the current instruction in the execute stage; so either it takes the result of the ALU unit or the Incremented PC value in case of a JAL instruction as R7 = PC+1, so the selection signal for this mux is JAL2 signal.

4.1.6 MemoryResult_ALUResult Multiplexer

This multiplexer chooses the final result that need to be written into the Register File which can either be the ALU result or the memory result, and the selection signal for this multiplexer is the MemToReg4 signal.

4.1.7 Is_It_LBU_LB unit

This unit is used mainly for the LB and LBU instructions, which aims to load the a byte from the memory and extend it with a zero extension or a signed extension to become a 16-bit number and store it in the specified register. It does this by using two muxes; the first to select either the 16-bit memory result or the 8-bit memory result that was extended using a zero extension into a 16-bit with the LBU as the selection line for this mux, and the second mux chooses either the result of the previous mux or the 8-bit memory result extended into a 16-bit using sign extension with the LB signal as it's selection signal.

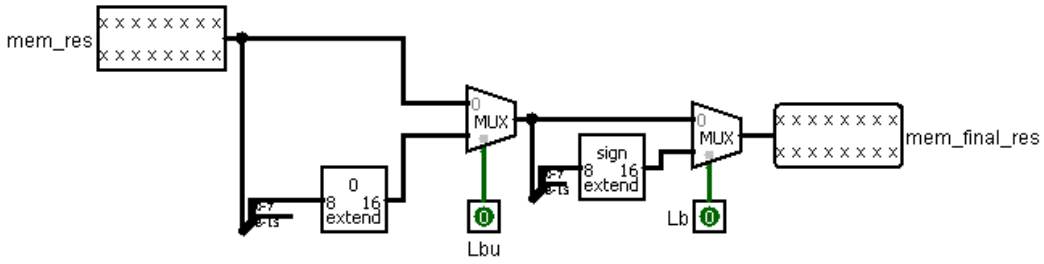

Figure 4: Is_It_LBU_LB Unit

4.2 Storage Elements

4.2.1 Data Memory

The data memory is of size 64KX16, for this memory an existence model in the RAM library was used, with a Data interface of type Separate load and store ports, so it have 8 ports which they are as the following:

1. A port: this port is specified for the address bus, which is of 16-bit.
2. D_in port: this port is specified for the data to be written into the memory, which is also of 16-bit.
3. D_out port: this port is specified for the data read from the memory, which is also of 16-bit.
4. str port: this port is controlling the store operation, so whenever it's 1 then the data in the d_in bus will be stored in the memory at the specified address.
5. ld port: this port is controlling the load operation, so whenever it's 1 then the D_out bus will be loaded by the the data in the memory of the specified address.

6. clk port: this port synchronises the write operation, so whenever there is rising edge and the str port is on, the write operation will be done.
7. sel port: this port responsible for disabling the component whenever it's value is 1, for this project there is no need for it.
8. clr port: this port responsible for the clearing of the memory, and in this project there is no need for it.

4.2.2 Instruction Memory

The instruction memory is of size 64KX16, for this memory an existence model in the ROM library was used, it has two main ports, an input for the 16-bit address and an output for the data_out fetched from the specified address. The reading operation is done asynchronously whenever and address is ready.

4.2.3 INC PC

This component is responsible for incremenating the PC by 2 at each clock cycle by using a D-Flip-Flop that transfer the value at its input to its output at the rising edge of the clock. A wire from its output is marked as an output of this component with the name (PCToInstMem), and another wire from its output is connected into an adder that adds 2 to the input resulting in an output called IncPC having the address of the next instruction to be fetched. Following is INC PC circuit implemented in Logisim.

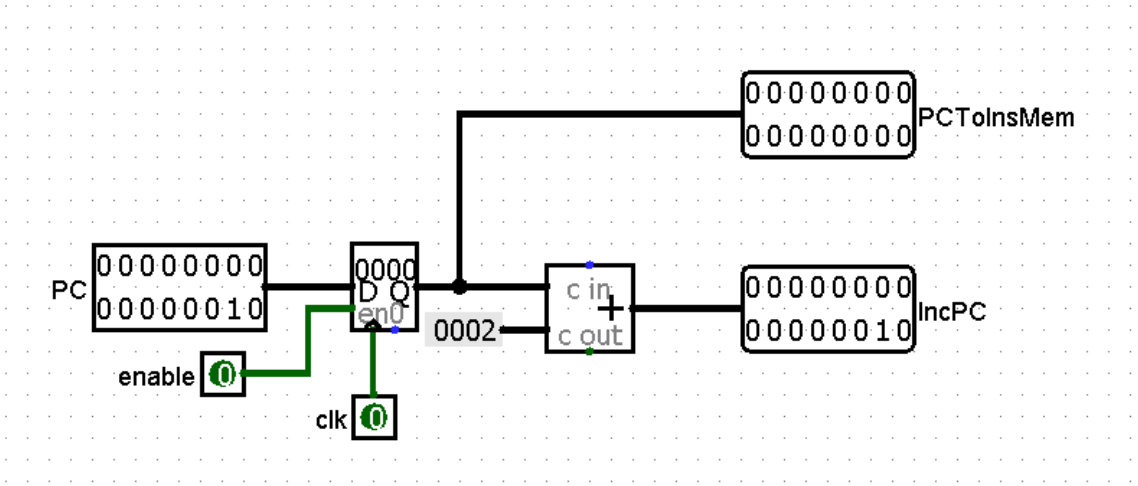

Figure 5: INC PC Unit

Considering the INC PC in a wider view, its input is taken in our datapath from a mux with a PCSrc signal as the selection signal. The PCSrc signal will be illustarted in details below,

but shortly, it is a signal set to one when there is a taken branch,jump,JAL or JR instructions. Having it as one, sets the input to our INC PC circuit as the address calculated in the NEXT_PC circuit illustrated below, else, if it is zero, the INC PC receives at its input, the last incremented PC.
Following is the related connection of the INC PC circuit and the multiplexer.

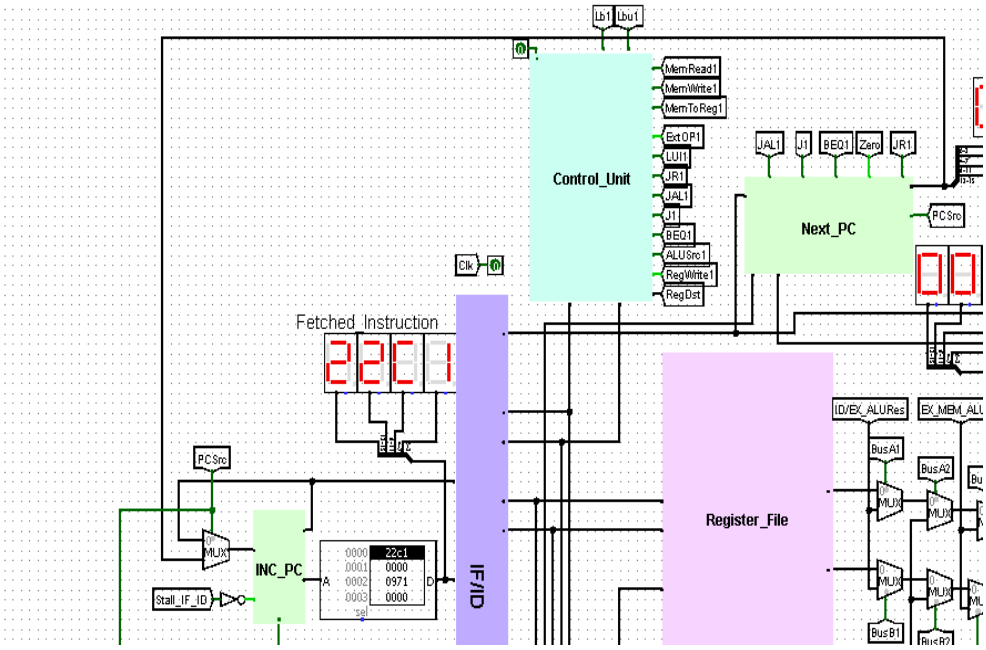

Figure 6: INC PC - Wider View

4.2.4 Next_PC

Next_PC unit, is mainly used to implement the jumping and branching instructions correctly. It takes as inputs some of the generated signals of the Control_Unit, namely: **JAL, J, BEQ, JR**, the **Zero Flag** generated by the comparator, the value of the **RS register** of the instruction in the decode stage, **least significant 12-bit of the instruction** in the decode stage, and finally the **incremented PC** of the instruction in the decode stage.
This circuit has mainly two outputs found as follows:

1. The Jumping_Branching Address: calculated as follows:
   (a) If a Branch instruction: shift left by one the sign-extended immediate added to the entered incremented PC.
   (b) If a Jump or JAL instruction: Most_significant_three_bits of entered PC concatenated with the 12_bit immediate concatenated with a zero.

(c) If JR: the entered value of the RS register.

2. The PCSrc signal: it is implemented as (JAL+JR+J+BEQ.Zero_Flag). Having this signal a one, changes the value of the PC to the Jumping_Branching address.

Following is Next_PC unit implemented with the above illustrated details, with the help of multiplexers coordinating the different calculated addresses.

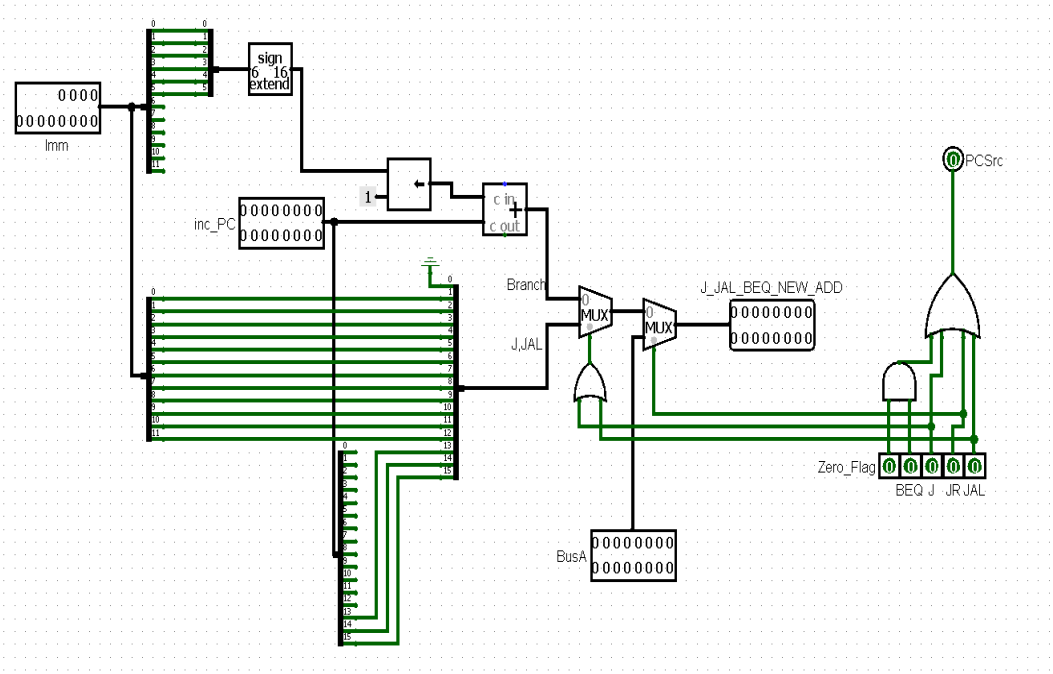

Figure 7: Next_PC Unit

4.2.5 Buffers Between Stages

For pipelining purpose, we need to have a buffer between each two stages, so we have 4 buffers which are IF/ID, ID/EX, EX/MEM and MEM/WB. All of them are implemented in the same way which is by using registers for each value to be stored in the buffer and all of them are triggered at the rising edge of the clock, the following illustrate the signals at each buffer:

1. IF/ID: we have 7 signals which they are: funct, Rd, Rt, Rs all of 3-bit, OP of 4-bit, imm_6 of 6-bit and imm_12 12-bit and all these signals were ontained from the Fetched instruction by using splitters. The last signal is the IncPC which is obtained from the IncPC resulted from the INC_PC component and it's 16-bit.

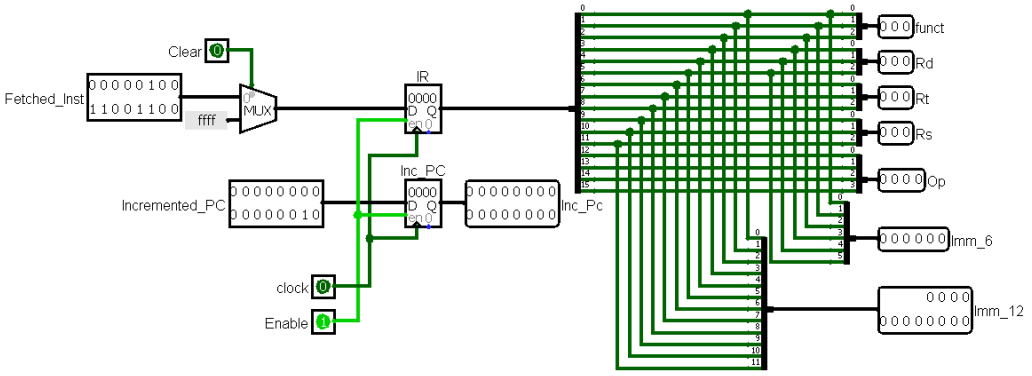

Figure 8: IF/ID Buffer

2. ID/EX: we have 20 signals which are: BUSA and BUSB of 16-bit representing the values of Reg(RS) and Reg(Rt) obtained from either the register file or from the forwarding units, also we have imm_IType and imm_JType of 16-bit which they are obtained from the Extender_6 and Extender_12 respectively, also we have Rt, Rd, Rs, OP, funct and IncPC signals obtained from the first buffer, as well as to the follwoing signals obtained from the control_unit MemToReg, RegWrite, MemRead, MemWrite, ALUSrc, JAL, LUI, LB and LBU and all of them are of 1-bit, and the last signal is the RegDst signal which is of 2-bit obtained from the control_unit.

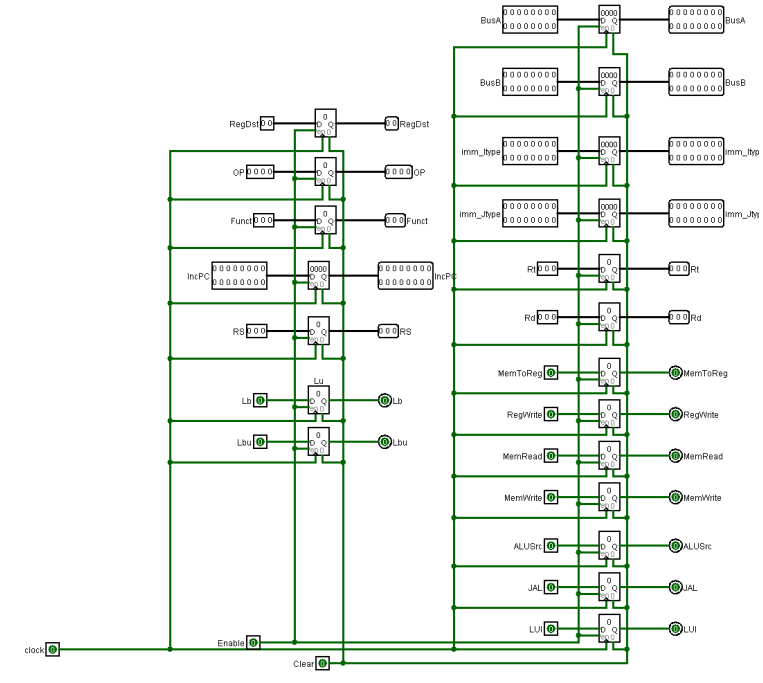

Figure 9: ID/EX Buffer Unit

3. EX/MEM: We have 11 signals which they are as the following: IncPC, BUSB, MemToReg, RegWrite, MemRead, MemWrite, Rt, Lb, LBu which they are obtained from the second buffer, also we have a signal for the result of the ALU which is ALUResult, as well as to RW signal that contains the address of the destination to write the result on.

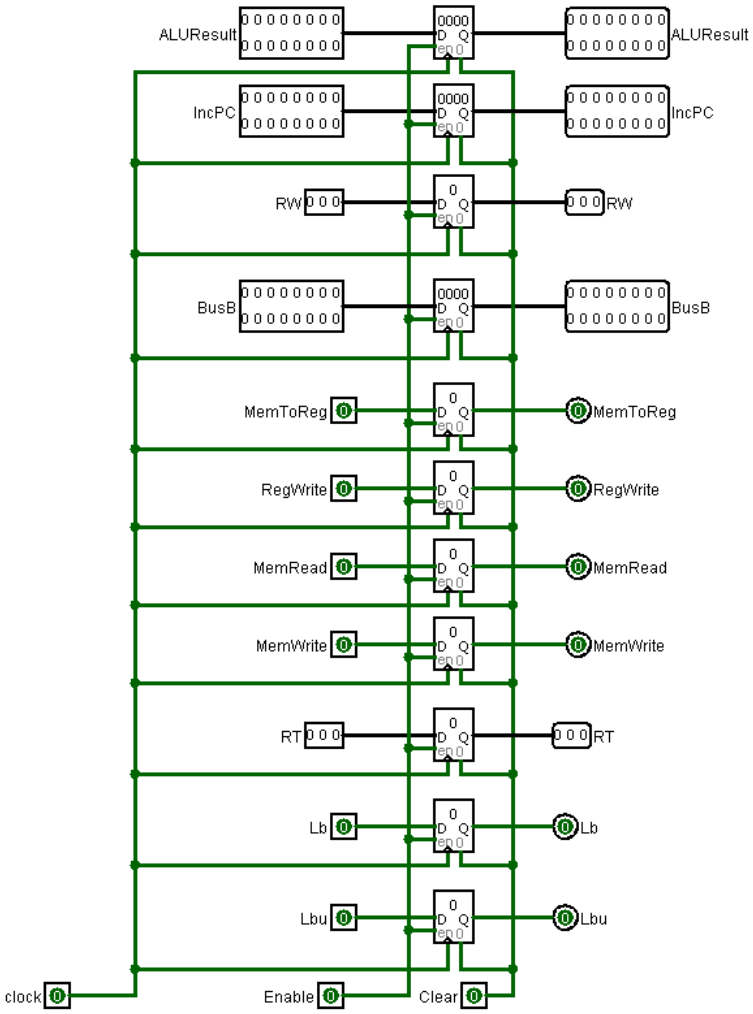

Figure 10: EX/MEM Buffer Unit

4. MEM/WB: we have 6 signals which they are ALUResult, IncPC, MemToreg, RegWrite and RW obtained from the previuous buffer as well as to MemoryData signal that resulted from the Memory stage.

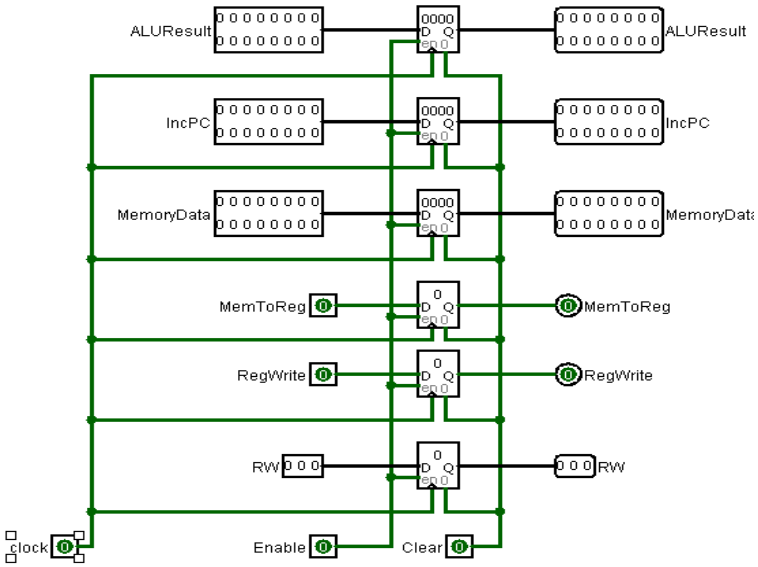

Figure 11: MEM/WB Buffer Unit

4.2.6 Register File

Register file consists of 8 16-bit registers, and it has 8 ports which are illustrated as the following:

1. BUSA And BUSB: 16-bit output buses for reading two registers.
2. BUSW: 16-bit input bus to write its data into the specified register when the needed conditions are applied as it will be illustrated later.
3. RA: 3-bit input bus that select the register to be read on BUSA.
4. RB: 3-bit input bus that select the register to be read on BUSB.
5. RW: 3-bit input bus that select the register to loaded with the value in bUSW.
6. RegWrite: 1-bit input that selects if the current action is read or write (1 means write, 0 means read).
7. Clock: The clock input is used only during write operation. During read, register file behaves as a combinational logic block RA or RB valid => BusA or BusB valid after access time.

The implementation of the register file done through two stages the first by implementing the write circuit and the second is by implementing the read circuit, for the write circuit it

was done by using a 3-8_Decoder with RW signal as it's input signal and each output of the decoder was Anding with the RegWrite signal and connecting to one of the registers enable in the register file (in-order mapping, Output0 to R0, Output1 to R1 and so on), expect the the first register since the first output of the decoder was Anding with '0' to disable the writing operation on that register (R0 always '0'), so in this way just one register can be written in any time with its address specified in the RW Bus and when RegWrite is '1'. For the read Circuit it was done by using two 8X1Muxs, each have 8 inputs comes from the output of each register in the register file, and the selection of the first MUX is RA and for the second is RB and connecting BUSA and BUSB with the output of each mux respectively.

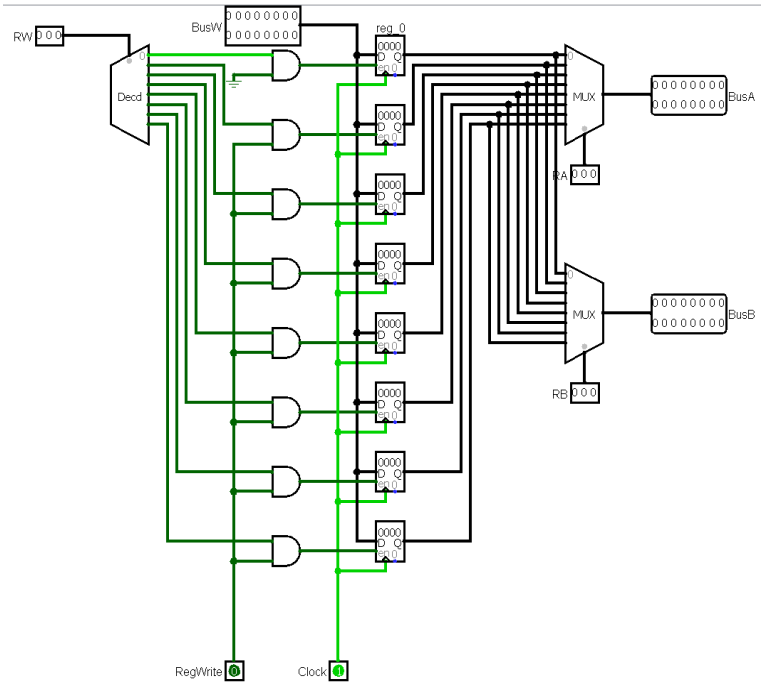

Figure 12: Register File Unit

4.3 Control Units and Hazard Detection

4.3.1 ALU_control Unit

Following is the ALU Control truth table

Table 2: ALU Control Truth Table

OP	Funct	ALU Operation	Encoding
0000	111	AND	000
0000	110	OR	001
0000	101	MAX	010
0000	100	ADD (LWS)	011
0000	011	ADD	011
0000	010	SUB	100
0000	001	(SLT)	101
0001	X	AND	000
0010	X	OR	001
0011	X	ADD	011
0100	X	ADD (LW)	011
0101	X	ADD (LBU)	011
0110	X	ADD (LB)	011
0111	X	ADD (SW)	011
1000	X	ADD (SB)	011
1001	X	SUB (BEQ)	100
1100	X	Shift (LUI)	110

Following are the logic equations for the ALU operations

1. AND = OP(0000).Funct(111) + OP(0001).
2. OR = OP(0000).Funct(110) + Op(0010).
3. MAX = OP(0000).Funct(101)
4. ADD = OP(0000).Funct(100) + OP(0000).Funct(011) + OP(0011) + OP(0100) + OP(0110) + OP(0111) + OP(1000).
5. SUB = OP(0000).Funct(010) + OP(1001)
6. SLT = OP(0000).Funct(001)
7. Shift = OP(1100).

Following is ALU control unit implemented circuit in Logisim with the help of the declared above functions.

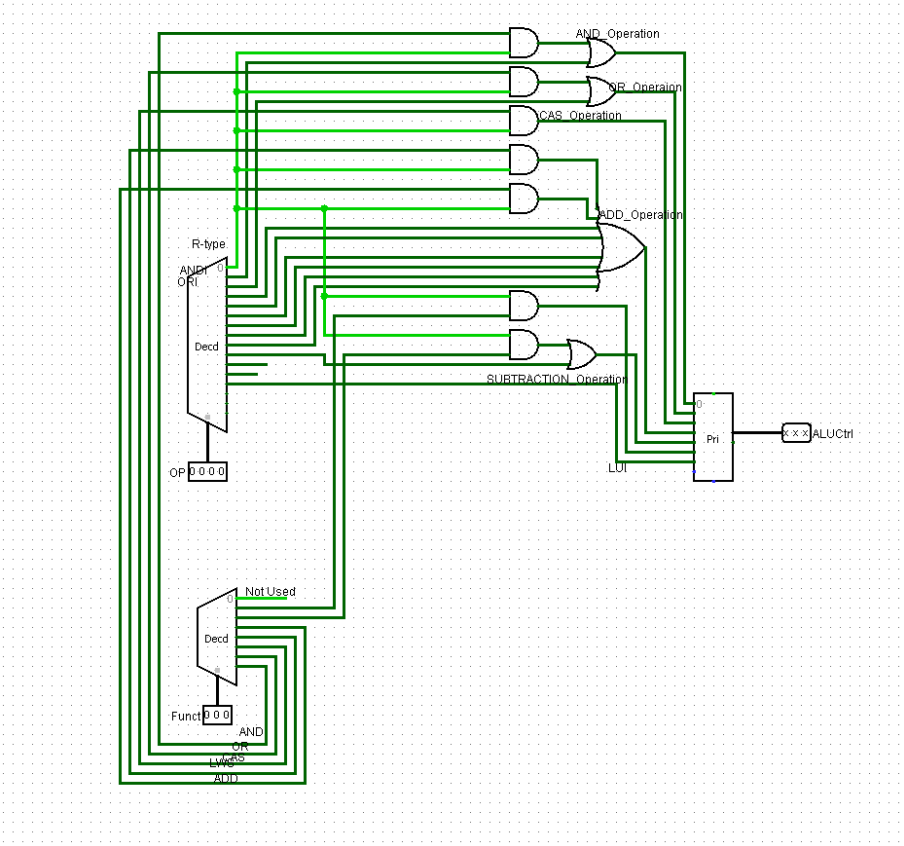

Figure 13: ALU Control Unit

4.3.2 Main_Control Unit

The Main_Control Unit, is a core unit in the implemented datapath, it is responsible for generating various signals responsible for the correct coordination of all used multiplexers, the inputs to different units and some of the work done by different units. It takes as input, the opcode and function fields of the instruction and them performs its work accordingly. Following is the Control unit truth table indicating the generated signals.

Table 3: Main Control Unit Truth Table

OP	Funct	RegDst (2-bits)	Ext OP	RegWrite	ALU Src	Beq	J	JAL	JR	LUI	Mem Read	Mem Write	Mem toReg
R-type	All except 000	11=Rd	X	1	0	0	0	0	0	0	0	0	0
JR	000	X	X	0	X	0	0	0	1	0	0	0	X
Lws	000	11=Rd	X	1	0	0	0	0	0	0	1	0	1
ANDI	X	10=Rt	1	1	1	0	0	0	0	0	0	0	0
ORI	X	10=Rt	1	1	1	0	0	0	0	0	0	0	0
ADDI	X	10=Rt	1	1	1	0	0	0	0	0	0	0	0
LW	X	10=Rt	1	1	1	0	0	0	0	0	1	0	1
LBU	X	10=Rt	0	1	1	0	0	0	0	0	1	0	1
LB	X	10=Rt	1	1	1	0	0	0	0	0	1	0	1
SW	X	X	1	0	1	0	0	0	0	0	0	1	X
SB	X	X	1	0	1	0	0	0	0	0	0	1	X
BEQ	X	X	X	0	0	1	0	0	0	0	0	0	X
J	X	X	X	0	X	0	1	0	0	0	0	0	X
JAL	X	01=R7	X	1	X	0	0	1	0	0	0	0	0
LUI	X	00=R1	X	1	1	0	0	0	0	1	0	0	0

Following are the logic equations for the control all generated signals by the control unit.

1. RegWrite = (JR + SW + SB + BEQ + J)
2. ExtOP = LBU
3. ALUSrc = (R-type + BEQ)
4. BEQ = BEQ
5. J = J
6. JAL = JAL
7. JR =R-type.FUNCTION(000)

8. LUI = LUI
9. MemRead = LW + LBU + LB + LWs
10. MemWrite = SW + SB
11. MemToReg = LW + LBU + LB + LWs
12. LBU = LBU
13. LB = LB.
14. RegDst : encoder.

Following is the implemented Control unit after having the truth table and the expression for each signal ready.

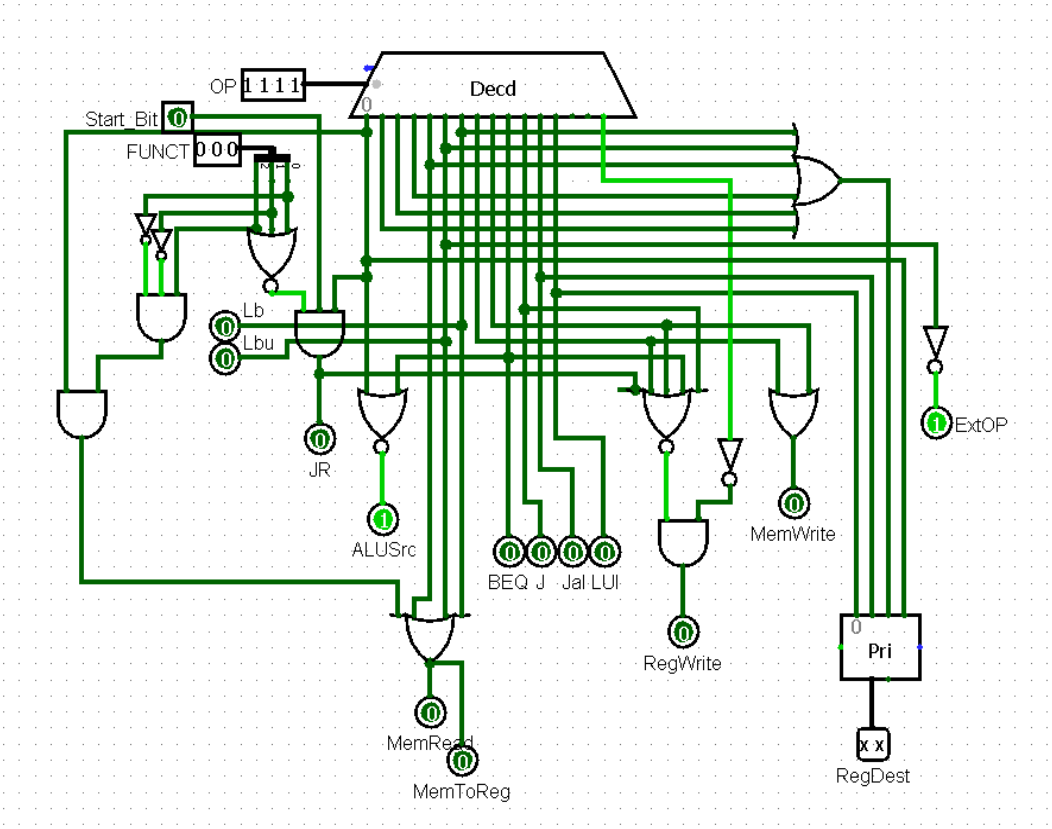

Figure 14: Main_Control Unit

4.3.3 Hazard_Detection Unit

This unit was implemented in order to detect when to stall the pipeline by one cycle when needed, meaning at the cases that the forwarding fails to forward the data at the right time, and by analyzing all the cases, the forwarding fails and a stall cycle is needed at the two following scenarios:

1. When there is a load instruction followed by an instruction that depends on the data resulted from the load instruction (R/W dependency), which can be:
   (a) Load followed by on R-type or a Branch instruction that depends on the data resulted from the load (RS or Rt in the R-type or the Branch instruction need Rd from the load instruction).
   (b) Load instruction followed by a store or JR instruction that depends on the loaded data to calculate the address of the desired memory location in case of Store instruction or depends on the loaded data to update the value of the PC in case of JR instruction (dependency between Rs in the store or JR instruction and Rt in the load instruction).
2. When there is an update of the PC because of J, JAL, JR and BEQ instructions.

How to detect the stall cycles:

1. The logic to detect the stall cycle in case of a load instruction followed by an instruction that depends on it (as illustrated above) is as the following:
   If
   (ID/EX.MEMRead and
   (ID/EX.RegisterRD != 0) and
   (ID/EX.RegisterRd = IF/ID.RegisterRs or
   (not IF/ID.MemWrite and
   ID/EX.RegisterRd = IF/ID.RegisterRt)))
2. The logic to detect the stall cycle in case of a misspredicition in BEQ, or J, JAL and JR is as the following: **if(**
   PCSrc and
   PC1 != PC2)
   where PC1 is the one generated from the Next_PC Component and the second is the IncPC.

What we need to do to stall the piplining:

1. For the first kind of stall (R/W dependency because of the Load instruction), we need to to disable the INC_PC component and the IF/ID buffer using the signal generated from

the detection logic illustrated above, also we need to clear (flush) the second buffer for the first half cycle of the next cycle (after detecting the stall) by using a D-FlipFlop triggering at the falling edge and connecting its output with an and gate with the clock signal, so the resulted signal will be the one that controls the flush operation of the second buffer.

2. For the second kind of stall, we need to flush the first buffer (IF/ID buffer) when we detect the stall to prevent it from passing to the decode stage and processing the wrong instruction, and the way we used to control the stall in this case is by using a D-FlipFlop triggering at the falling edge of the clock to pass the signal generated from the detected logic of the second type, and anding its output with the not of the clock, so in this way the flush for the first buffer will be activated during the second half cycle of the detection cycle.
Note: Flush (clear) here means to pass a nop instruction into the decoding stage (which includes seting all the bits in the instruction to be passed into the decoding stage to 1) that's because clearing the buffer by resetting it will result in a JR instruction and returns the execution of the program to the first instruction.

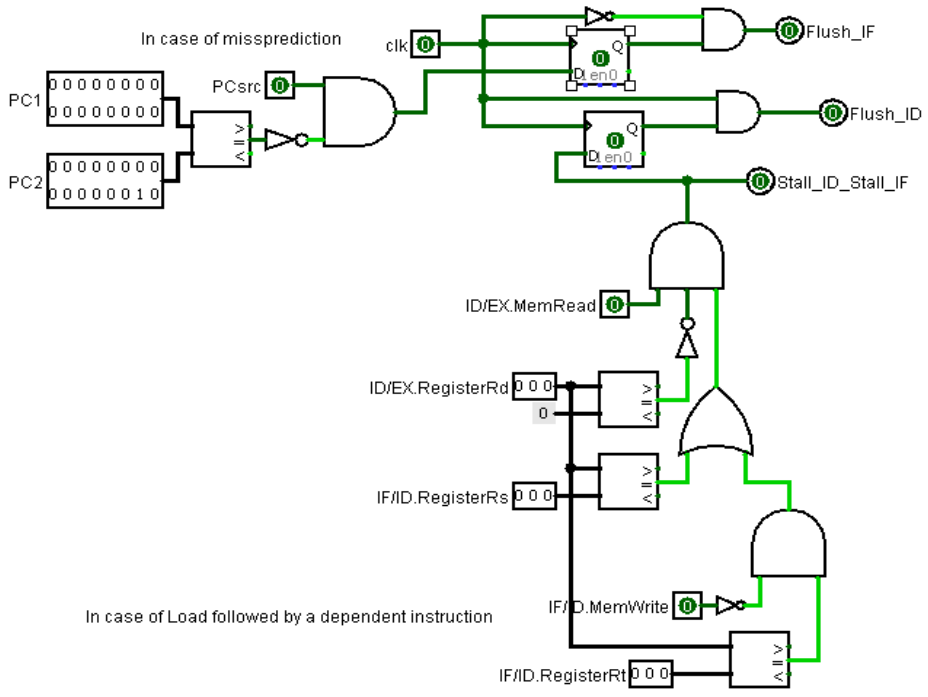

Figure 15: Hazard Detection Unit

4.4 Forwarding Units

In this section, all implemented forwarded units are to be illustrated. A forwarding unit is a way of optimising the pipelined path when a stall is needed; as some operations need results provided by earlier not yet finished operations.
Our introduced datapath has the following forwarding unit:

4.4.1 Write-Back_to_Memory Forwarding Unit

This unit, aims to forward the final result in the Write Back stage, to be written into the register file, to the memory stage, when there is a read after write hazard between the instruction in the memory stage and the instruction in the write back stage. It has the following logic:

If (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (MEM/WB.RegisterRd = EX/MEM.RegisterRt))
then WB_to_MEM = 1
Then, the produced signal is used as a selection signal in the multiplexer used before the Data port of the Data_Memory

Following is the implemented circuit for the unit.

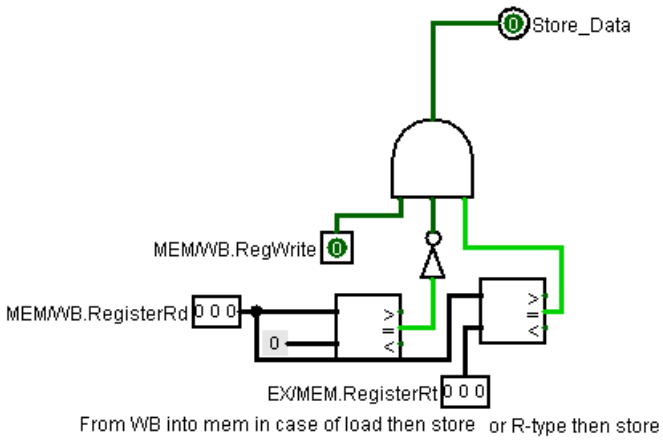

Figure 16: Write-Back_to_Memory Forwarding Unit

4.4.2 Memory_to_Execute Forwarding Unit

This unit, aims to forward the result of the ALU presented in the memory stage to the execution stage, when there is a read after write hazard between the instruction in the execute and

the instruction in the memory stage. It has the following logic:

If (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
then MEM_to_EX_ALU1 = 1

If (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
then MEM_to_EX_ALU2 = 1

Then, the two produced signals are used as selection signals in the two multiplexers used in deciding the ALU inputs.
Following is the implemented circuit for the unit.

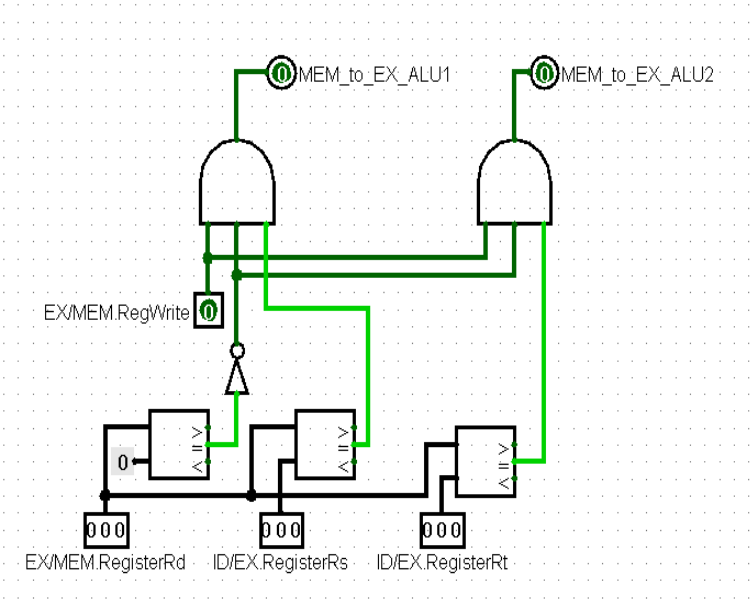

Figure 17: Memory_to_Execute Forwarding Unit

4.4.3 Write-back_to_Execute Forwarding Unit

This unit forwards the result to be written in the register file in the write-back stage to the execution stage when dependency is detected with the following logic:

If (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)

and (EX/MEM.RegisterRd != ID/EX.RegisterRs)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
then WB_to_EX_ALU1 = 1

If (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (EX/MEM.RegisterRd != ID/EX.RegisterRt)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
then WB_to_EX_ALU2 = 1

Following is the implemented circuit for the unit.

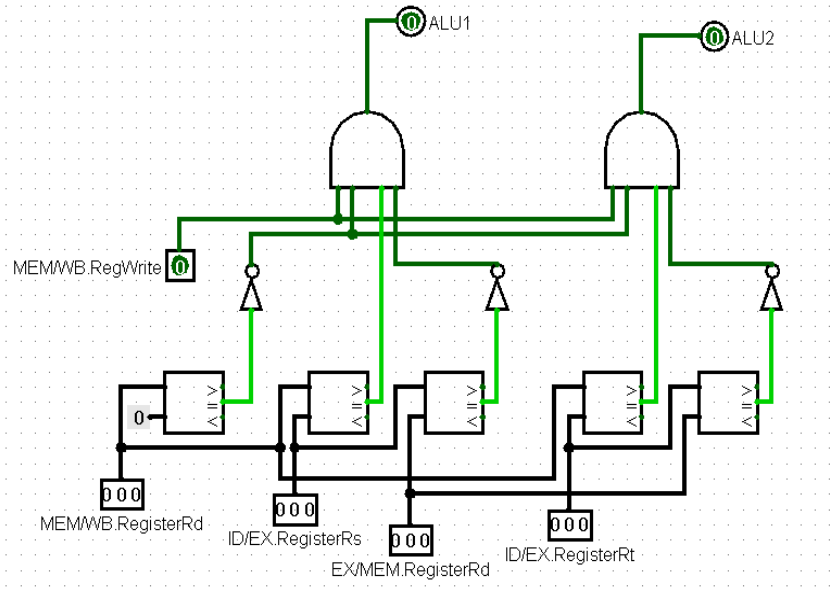

Figure 18: Write-back_to_Execute Forwarding Unit

4.4.4 Memory_to_Decode Forwarding Unit

This section has two main forwarding unit, the first one forwards the loaded value from memory in the memory stage to the decode stage in case of a read after write dependency is detected between the instruction in the decode stage and the one in the memory stage. This forwarding unit is concerned mainly with a load instruction followed by any instruction then an instruction that reads the loaded value. The detecting of the hazard is done with the help of the following logic:

If (EX/MEM.RegWrite
and (EX/MEM.MEMRead)

and (EX/MEM.RegisterRd != 0)
and (ID/EX.RegisterRd != IF/ID.RegisterRs)
and (EX/Mem.RegisterRd = IF/ID.RegisterRs))
then MEM_to_ID_BusA = 1

If (EX/MEM.RegWrite
and (EX/MEM.MEMRead)
and (EX/MEM.RegisterRd != 0)
and (ID/EX.RegisterRd != IF/ID.RegisterRt)
and (EX/MEM.RegisterRd = IF/ID.RegisterRt))
then MEM_to_ID_BusB = 1

Following is the implemented circuit for the unit.

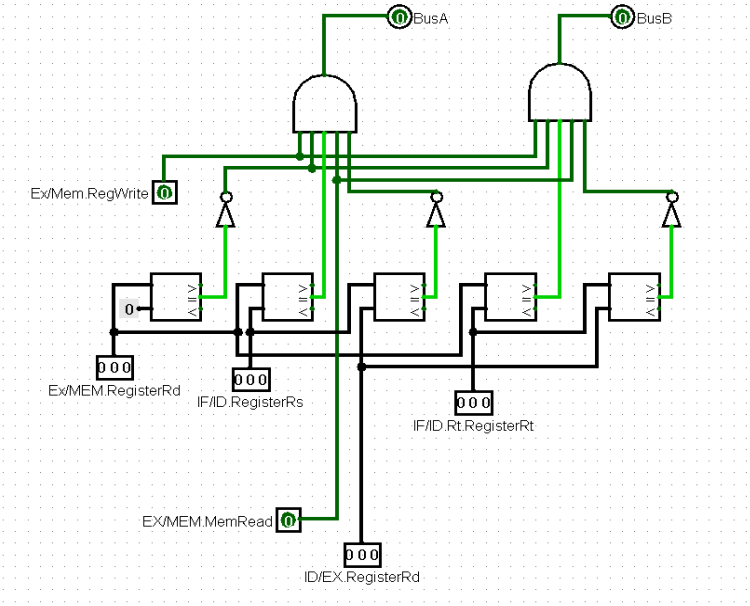

Figure 19: Memory_to_Decode Forwarding Unit

The second unit in this section forwards the value of the ALU that is in the memory stage to the dependent instruction reading this value in the decode stage. The detection of dependency is done with the following logic:

If (EX/MEM.RegWrite
and (not EX/MEM.MEMRead)
and (EX/MEM.RegisterRd != 0)
and (IF/ID.RegisterRs != ID/EX.RegisterRd)

and (EX/MEM.RegisterRd = IF/ID.RegisterRs))
then MEM_ALURes_to_ID_BusA = 1

If (ID/EX.RegWrite
and (not EX/MEM.MEMRead)
and (EX/MEM.RegisterRd != 0)
and (IF/ID.RegisterRt != ID/EX.RegisterRd)
and (EX/MEM.RegisterRd == IF/ID.RegisterRt))
then MEM_ALURes_to_ID_BusB = 1

Following is the implemented circuit for the unit.

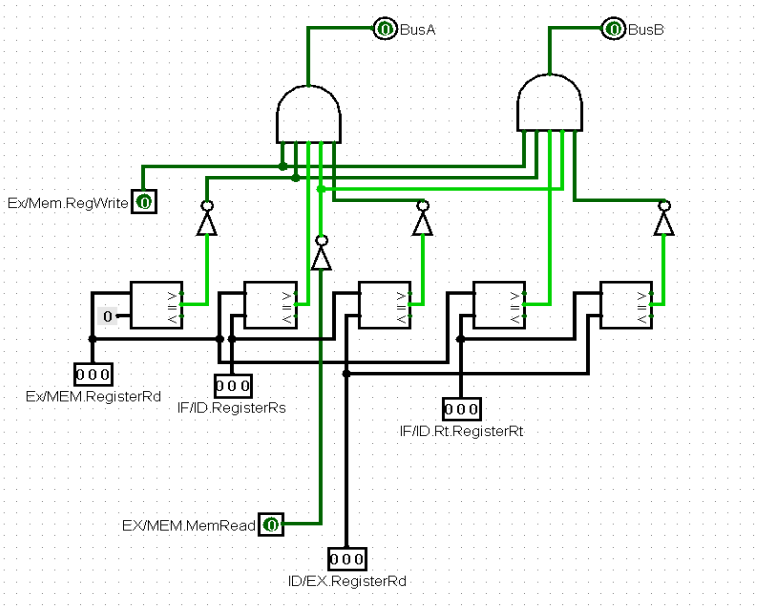

Figure 20: Memory_to_Decode Forwarding Unit

4.4.5 Execute_to_Decode Forwarding Unit

This units forwards the result of ALU in the execution stage to the decode stage in case of a dependency detected by the following logic:

If (ID/EX.RegWrite
and (ID/EX.RegisterRd != 0)
and (ID/EX.RegisterRd = IF/ID.RegisterRs)
and (not ID/EX.MEMRead))
then ALU_to_ID_RS = 1

If (ID/EX.RegWrite
and (ID/EX.RegisterRd != 0)
and (ID/EX.RegisterRd = IF/ID.RegisterRt)
and (not ID/EX.MEMRead))
then ALU_ALURes_to_ID_RT = 1

Following is the implemented circuit for the unit.

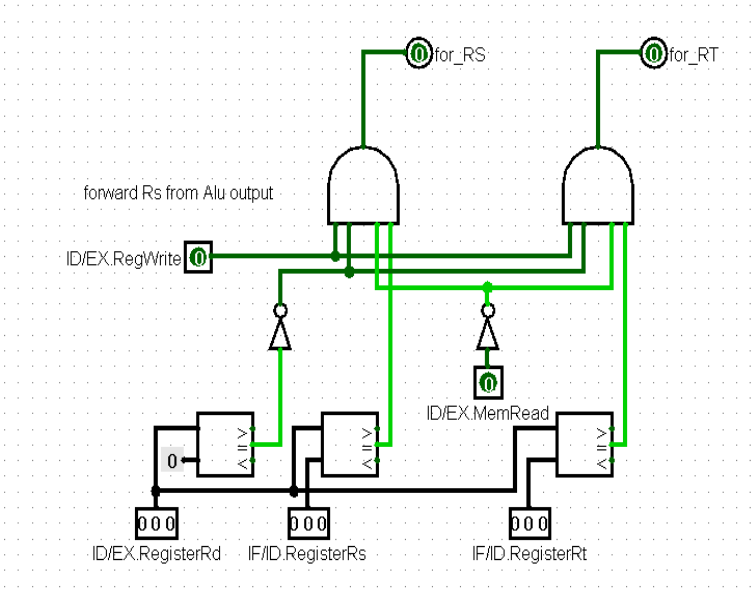

Figure 21: Execute_to_Decode Forwarding Unit

5 SIMULATION AND TESTING

In this section, many test cases are provided, starting with a basic flow demonstrating our correct implementation of all components, followed by testing the implemented forwarding units as well as hazard detection units and finally the correct implementation of the reduced number of introduced stall cycles.

5.1 Basic Flow Testing

The following instructions were written in the instruction memory:

1. 0X3285 = 0011 001 010 000101 : ADDI R2,R1,5.
2. 0X28C6 = 0010 100 011 000110 : ORI R3,R4,6.
3. 0X0BBB = 0000 101 110 111 011 : ADD R7,R56,R5.
4. 0XC006 = 1100 000000000110 : LUI 6.

The register file was initialized with the following values:

1. R1 = 1.
2. R5 = 5.
3. R6 = 6.
4. others = 0.

In this basic flow, no dependencies were introduced, nor any forwarding unit or hazard detection were needed. This flow demonstrates a full functional pipelined datapath at first place.
In the following figures, we aimed to capture the pipeline in different clock cycles, hex-digit displays were added in each stage so that the result of each separate stage can be tested and verified clearly.
The following first figure captures the third clock cycle, in which the first instruction(ADDI R2,R1,5) is in the third stage (Execution) and we notice that the ALU works properly as the result is 6. The second instruction(ORI R3,R4,6) is in the decode stage, and the read values from the register file are zeros as R3 and R4 were initialized by zero. The third instruction (ADD R7,R56,R5) is in the fetch stage and so we notice that the instruction (0X0BBB) is clear in the digit display connected to the instruction memory.

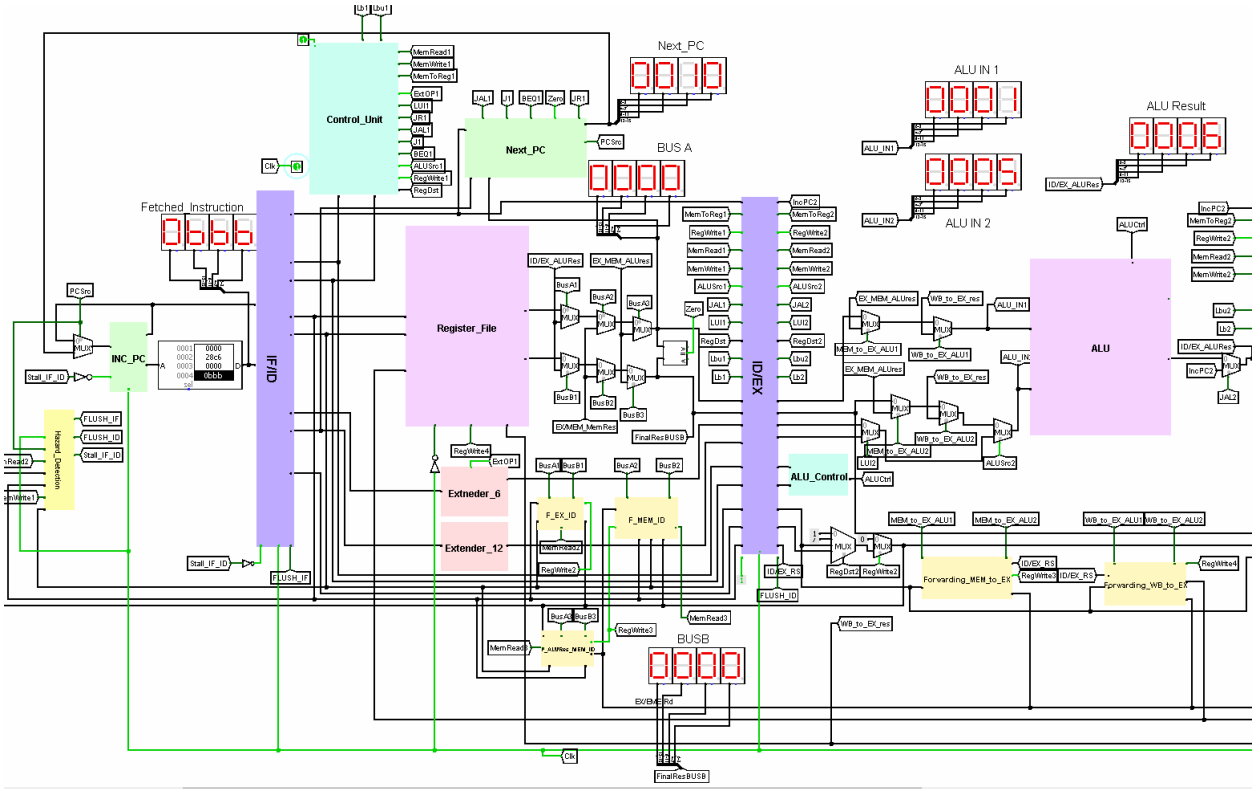

Figure 22: Third Clock Cycle

In the next two figures, the implemented pipelined path is captured at the fifth clock cycle.

In the first following figure, it is noticed that the (LUI 6) is in the decode stage and the LUI signal is generated from the Control_Unit as noticed. Also, the instruction to be fetched now is in address 0X8 and it is 0X0000 instruction.

In the second following figure, we are capturing the (ADD R7,R56,R5) in the execution stage, and we notice a correct result of the ALU as 5+6=b. We also notice the (ORI R3,R4,6) in the memory stage, in which its ALU_result indicated in the above display is 6 as 0 or 6 = 6. And in the writeback stage there is the first fetched instruction (ADDI R2,R1,5) aiming to write its result (6) back to the register file.

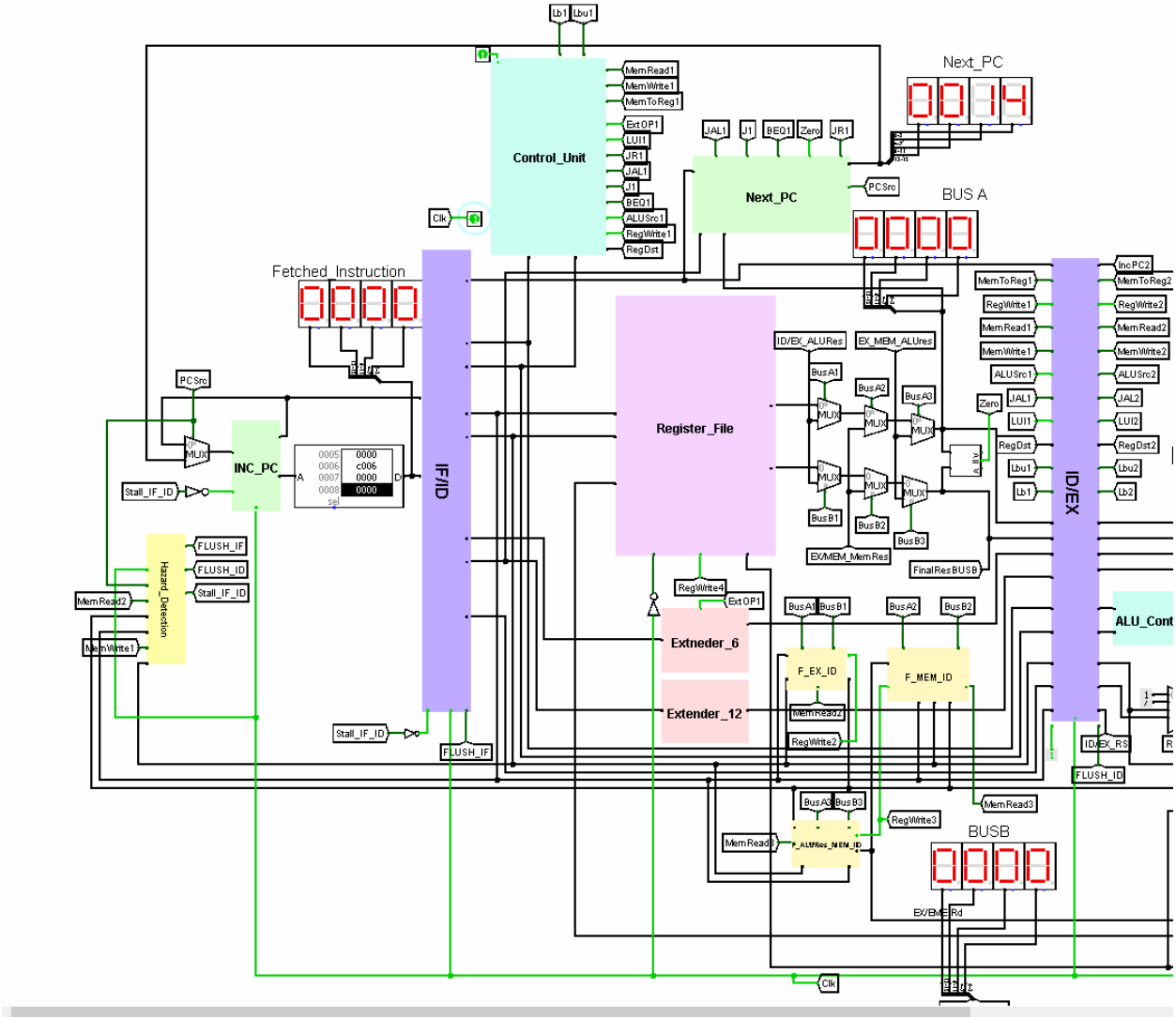

Figure 23: Fifth Clock Cycle

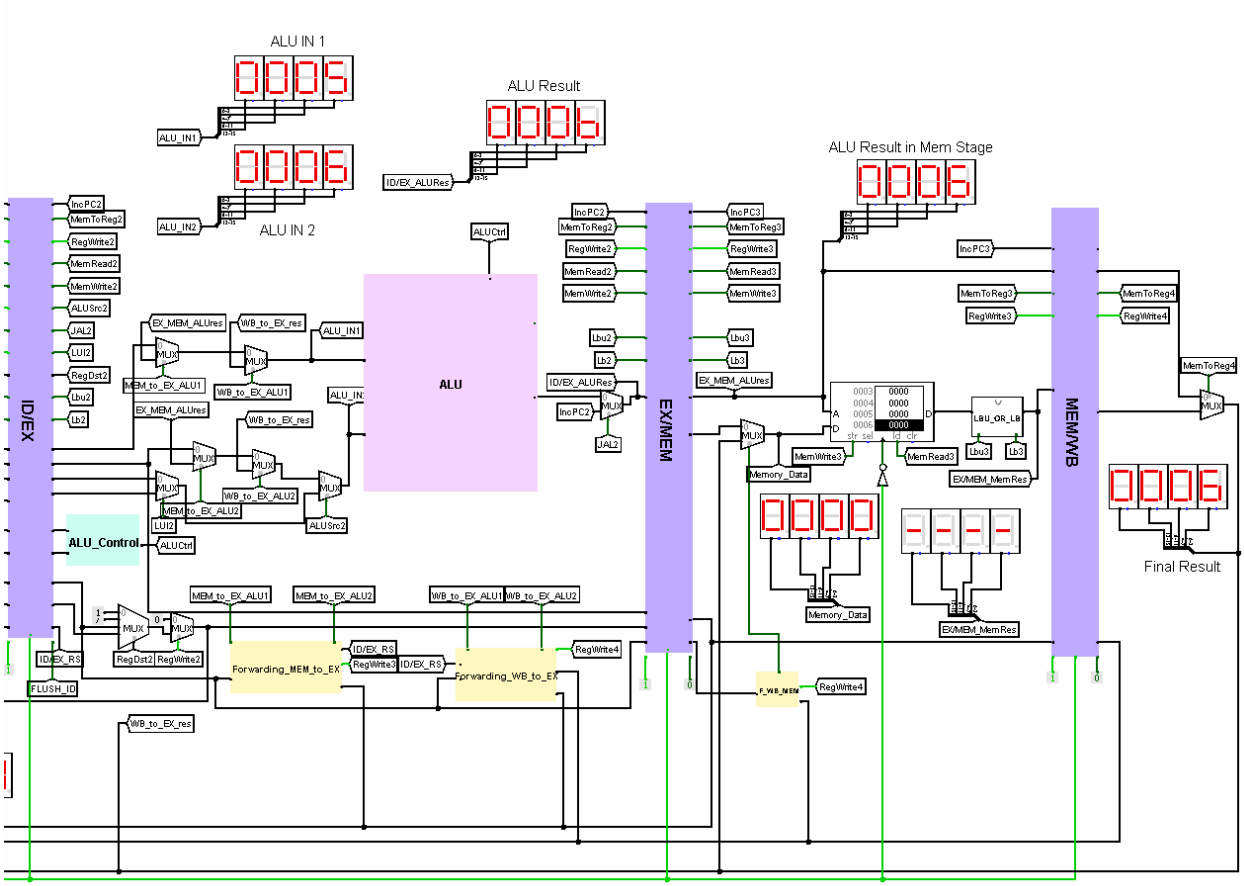

Figure 24: Fifth Clock Cycle

In the following figure, the pipelined is captured at the sixth cycle, in which the (LUI 6) instruction is now in the execute stage and the result is 6, indicating a successful ALU control implementation, as LUI instruction would shift the immediate 4 steps logically to the left, and so 0X6 would be now 0X60. It is important to notice that having the LUI signal set to one when executing this instruction, selects the 16-bit extended immediate (that was fetched as 12-bit), also, in this stage the RegDst singal would be 00, allowing the multiplexer that decides the destination register to have the value of 1.
It is also noticed that the (ADD R7,R6,R5) is now in the memory stage with its right ALU_result passed correctly. And in the write_back stage is the result of the Oring function.

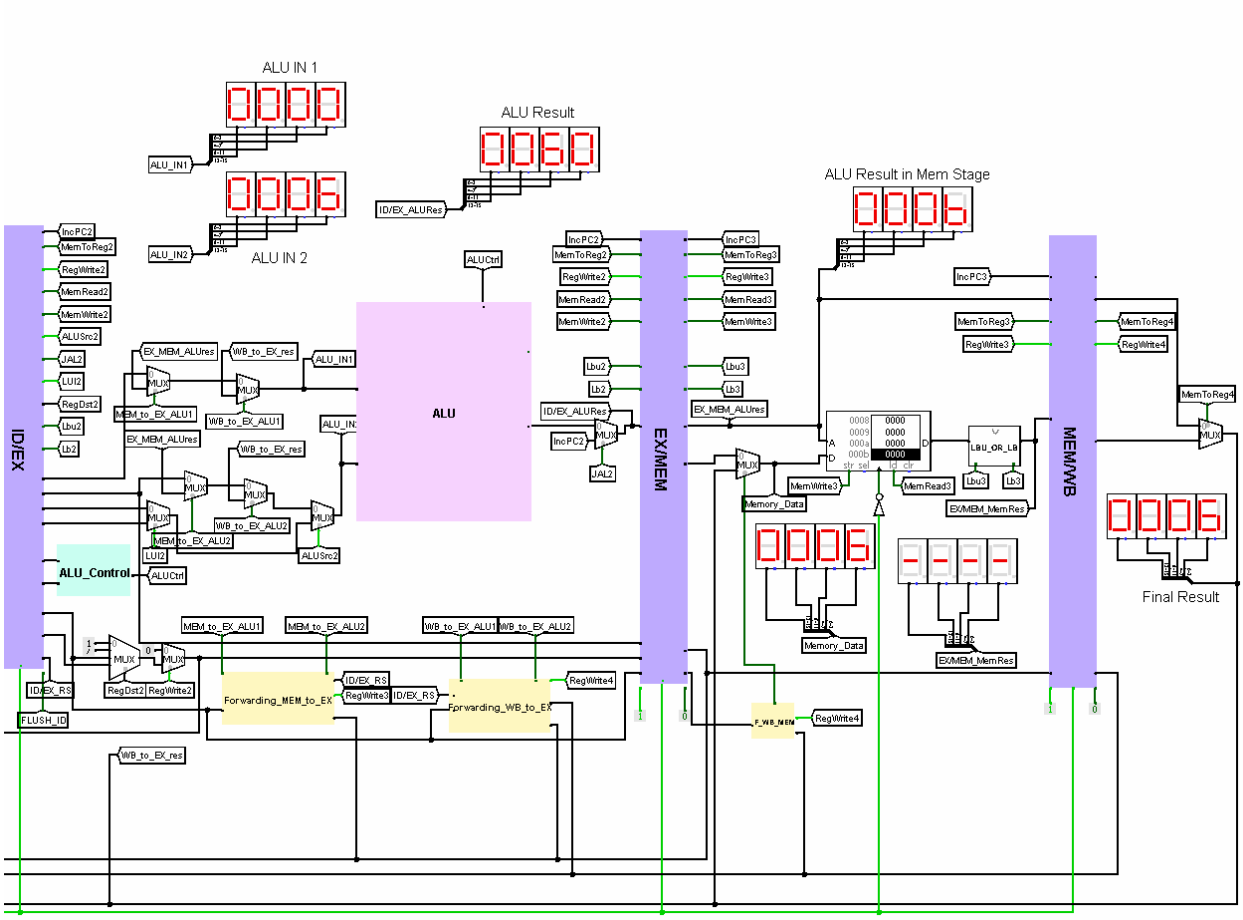

Figure 25: Sixth Clock Cycle

Finally, in this last figure, the register file after implementing all the above instruction, the correct result are there in our register file:

1. R1 = 0X60 = result of LUI 6, as the defualt resigter for this function is R1
2. R2 = 0X6 = result of ADDI R2, R1, R2 = 1 + 5 = 6.
3. R3 = 0X6 = result of ORI R3, R4, R3 = 0 OR 6 = 6.
4. R4 = 0X0, since there isn't any instruction that overwrites it.
5. R5 = 0X5, since there isn't any instruction that overwrites it.
6. R6 = 0X6, since there isn't any instruction that overwrites it.
7. R7 = 0X7, since there isn't any instruction that overwrites it.

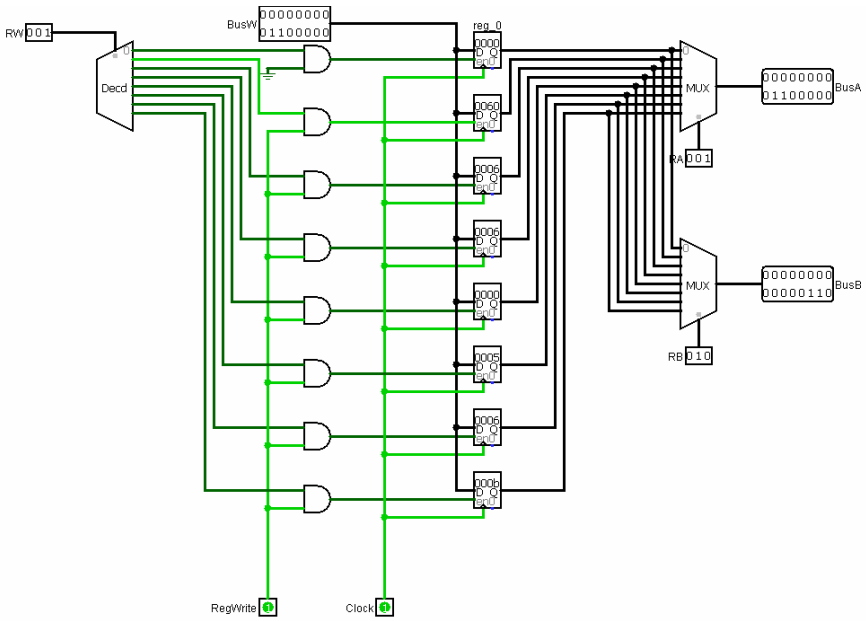

Figure 26: Register File

5.2 Testing Forwarding Units and Stalls- Part 1

In this section, the **write_back to execute** forwarding unit, as well as the **memory to execute** forwarding unit are to be tested with the following instructions, the jump is also added for testing introducing a stall, as we aim to test a maximum number of instructions as possible:

1. 0X04CE = 0000 010 011 001 110 : OR R1,R2,R3.
2. 0X0975 = 0000 100 101 110 101 : CAS R6,R4,R5.
3. 0X03BB = 0000 001 110 111 011 : ADDI R7,R1,R6.
4. 0XA005 = 1010 000000000101 : Jump 5.

In the above instructions, it is noticed that the third instruction (ADDI R7,R1,R6) has a read after write dependencies for both the instructions (OR R1,R2,R3 and CAS R6,R4,R5), and as a result and to reduce the needed stall cycles to zero, the forward unit was implemented and proved its effectiveness as shown in the example below. The register file was initialized with the following values:

1. R2 = 2.
2. R3 = 3.
3. R4 = 4.

4. R5 = 5.

5. others = 0.

In the following figure, the pipelined was captured in the third clock cycle, in which the (OR R1,R2,R3) instruction is in the execution stage and the Oring is done right between 2 and 3. In the decode stage, the (CAS R6,R4,R5) is to be decoded, and so the obtained values of registers 4 and 5 are to be written in the digit display, reading the right values stored in both registers. Also, it is noticed that the (ADDI R7,R1,R6) is in the fetch stage.

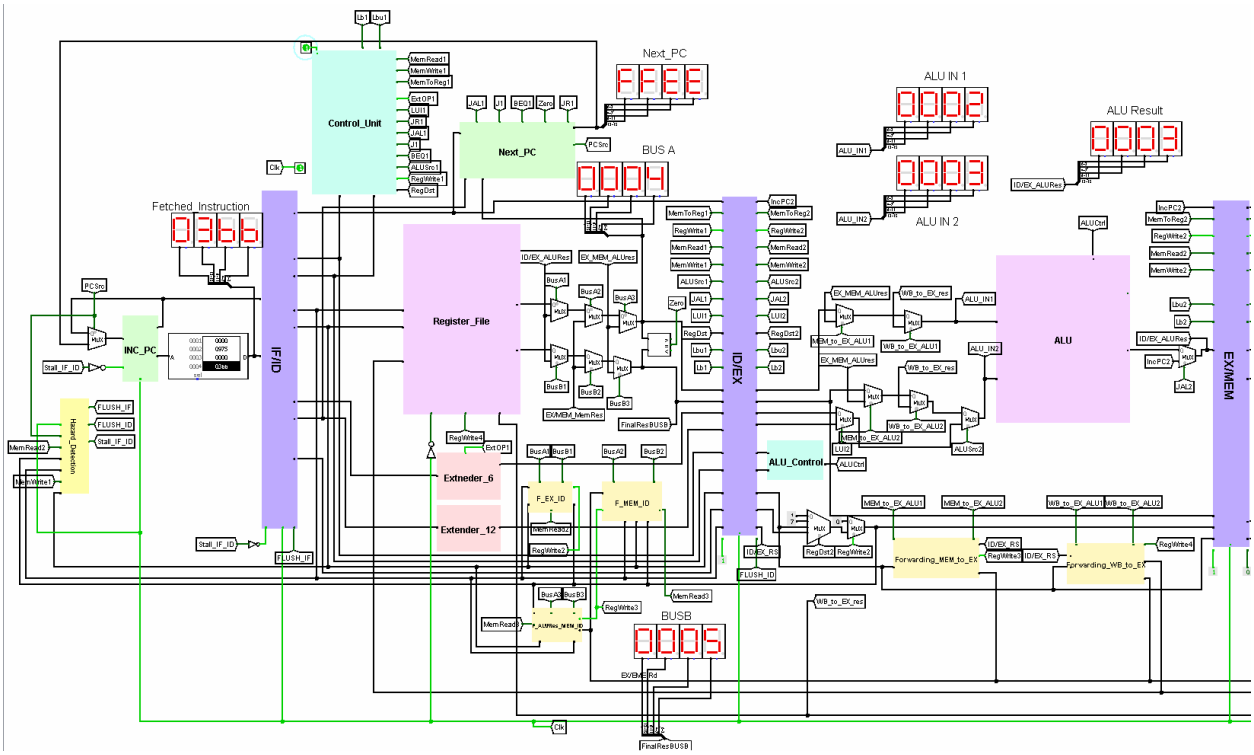

Figure 27: Third Clock Cycle

In the following two figures, the datapath is captured in the fifth clock cycle. Here, many important points are to be mentioned:

1. The program counter has the value of 0008, which is the address of the next instruction to be fetched if the jump is not in the decode.

2. The decode stage has the jump instruction, it is noticed that the J signal generated from the Control_Unit is set to one. In addition, the address written in the next_PC display is to be considered now and would be written in the program counter. It is 0XA as the 12-bit immediate is 5 and the next PC is calculated with the formula (MSB(3-bits) of

incremented PC Concatenated with 12-bit immediate concatenated with zero). and so A is the right new PC.

3. In the execution stage, the (ADDI R7,R1,R6) is being executed, and the role of the forwarding unit is now considered. Referring to the logic implemented in the forwarding units and as explained in the implementation section;

   (a) The **write_back to execute** forwarding unit would signal the multiplexer in the first ALU source to consider the forwarded result from the Write_back stage (R1 result).

   (b) The **memory to execute** forwarding unit would signal the multiplexer in the second ALU source to consider the forwarded ALU_result presented in the memory stage (R6 result).

   (c) It is clear that the forwarding is right as R1(first ALU source) is 3 which is the result of Oring 2 and 3, and as R6(second ALU source) is 5 which the max between 4 and 5. All proving that the ALU previously was working correct depending on the signals generated from the ALU_Control Unit.

   (d) The (CAS R6,R4,R5) instruction is in the memory stage.

   (e) The (OR R1,R2,R3) is in the Write_back stage.

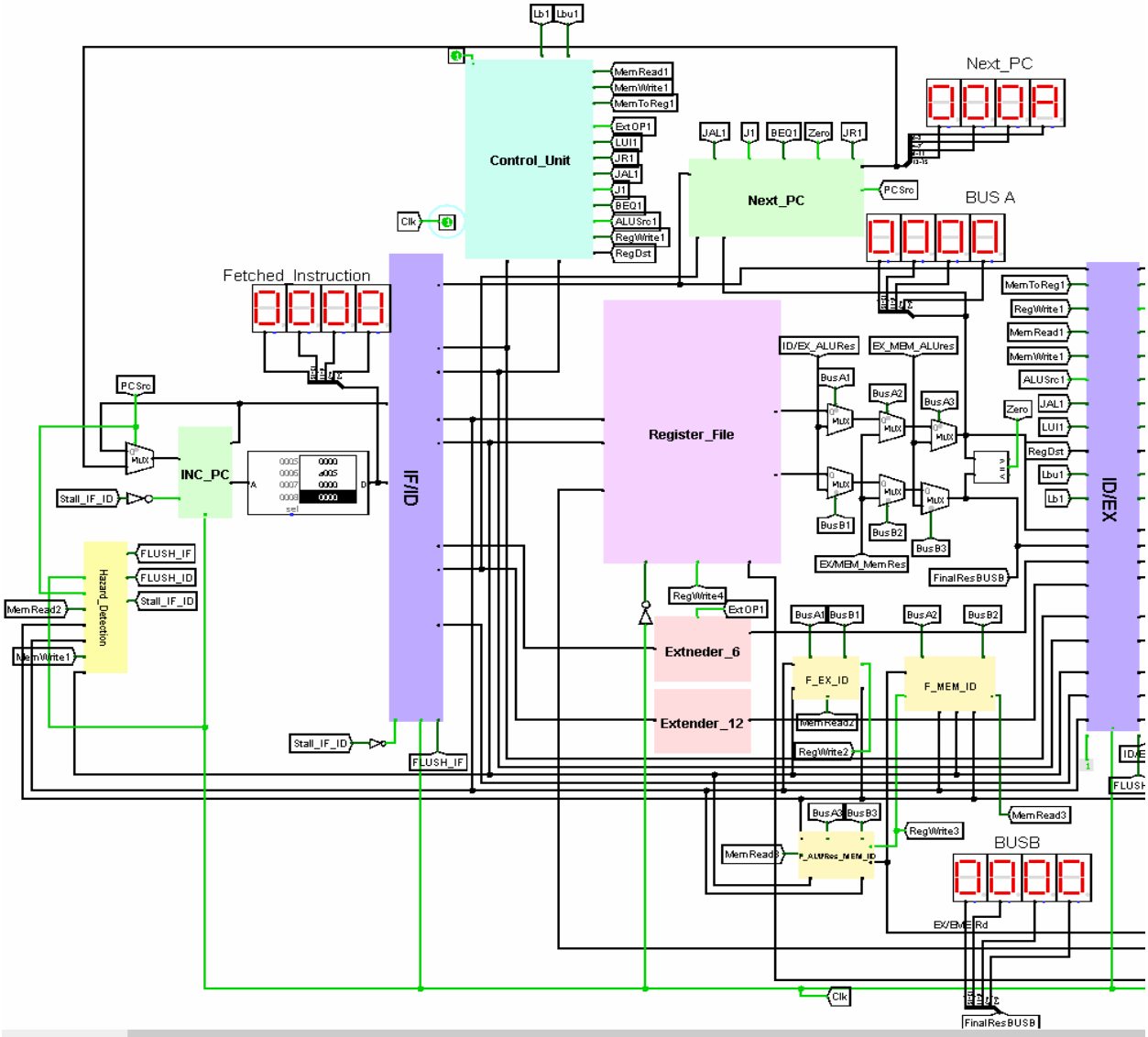

Figure 28: Fifth Clock Cycle

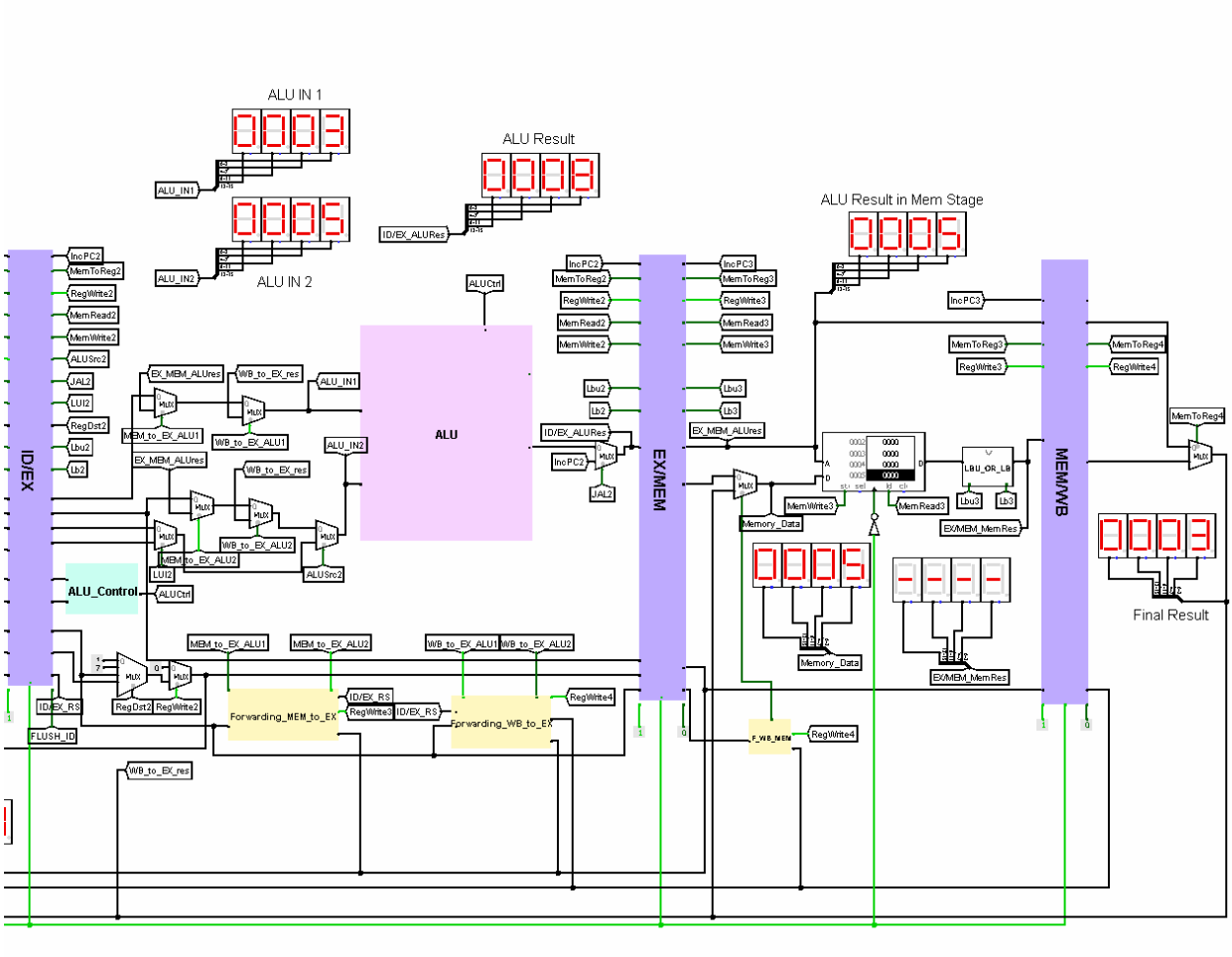

Figure 29: Fifth Clock Cycle

In the following figure, we would be considering the correct implementation of the stall cycle.
In the fetch, the value of PC is 0XA, which is the value introduced by the jump instruction. In the decode, the passed instruction is 0XFFFF, as this what we have implemented for stalls. After another clock cycle, the stall is moved further to the execution with RegWrite and MemWrite singals set to zero, and the instruction in address 0XA is now in the decode (fetched after a stall is introduced). Aslo, the PC now is 0XC which is the expected value of the next PC after 0XA.

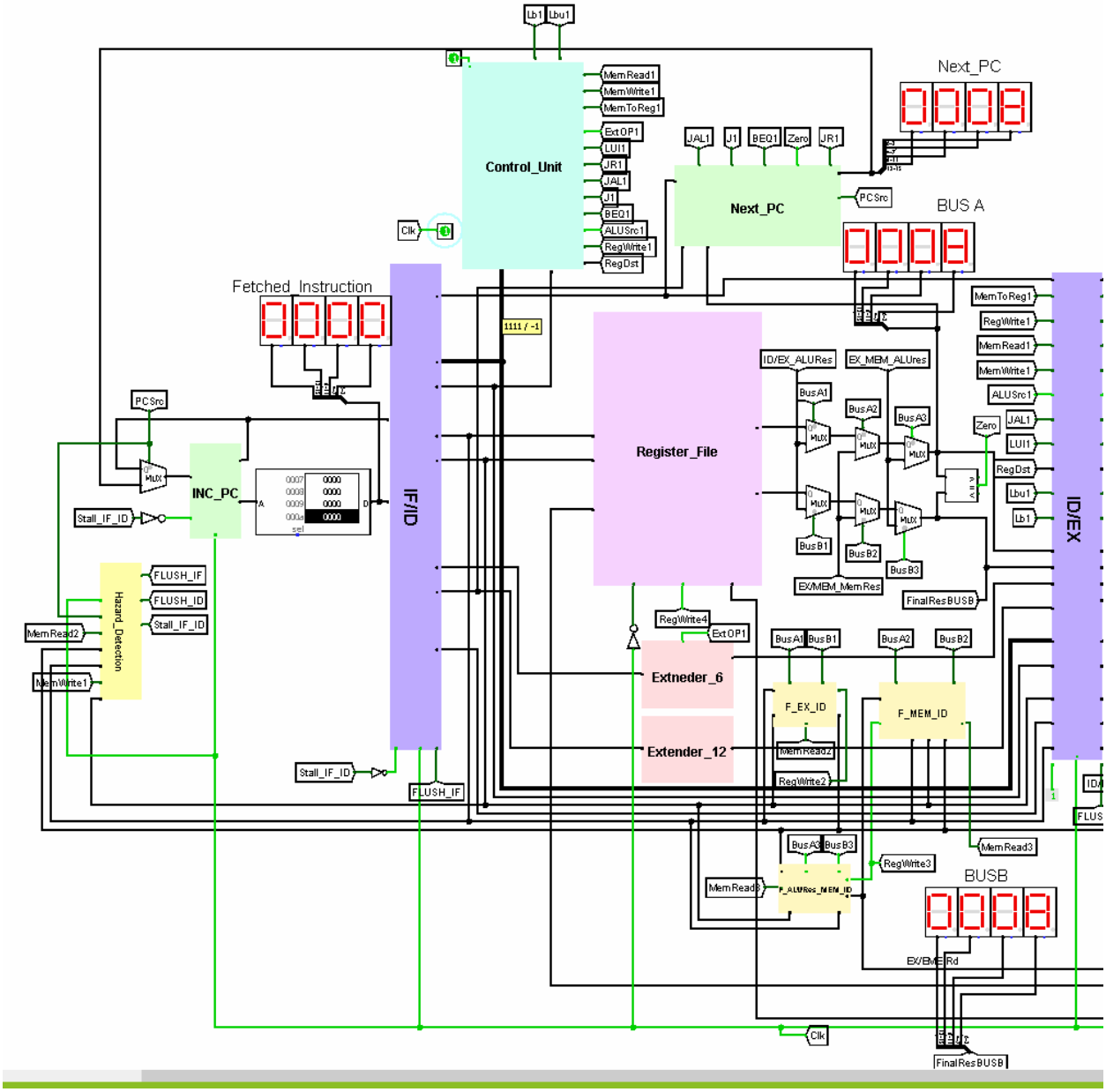

Figure 30: Introducing a Stall

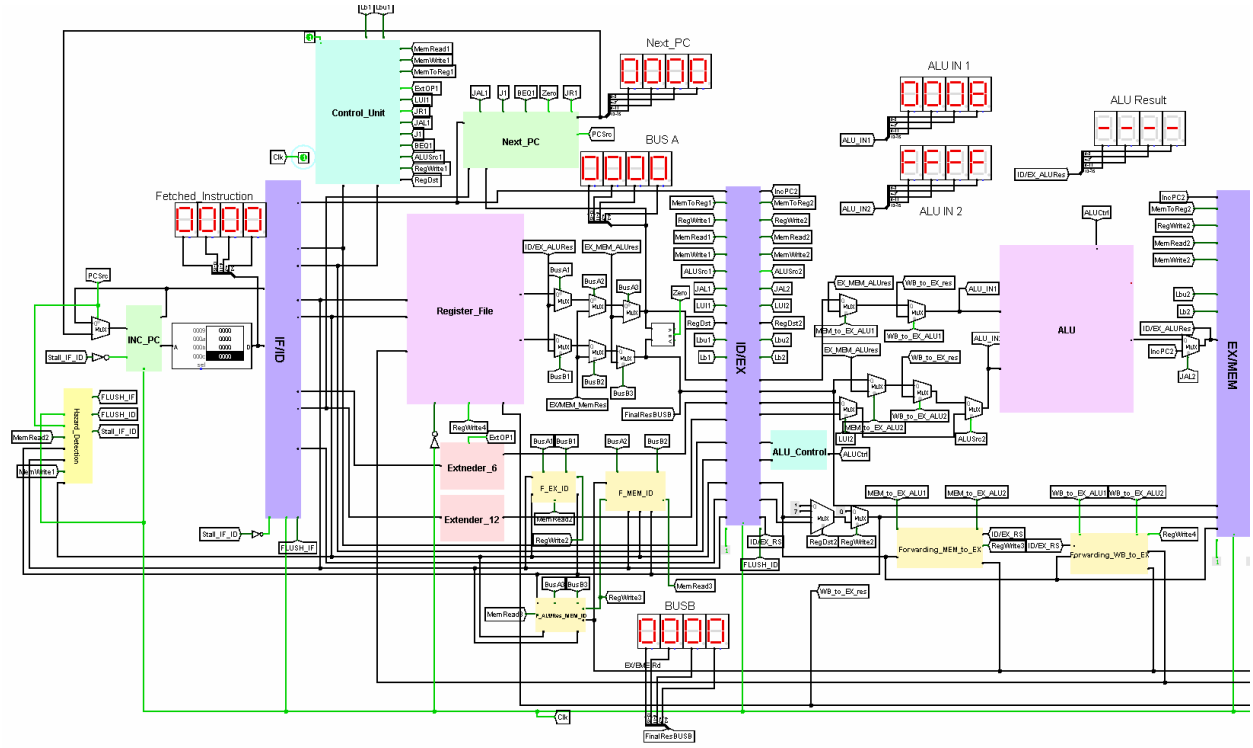

Figure 31: A Clock After the Stall

Finally, in this last figure, the register file after implementing all the above instructions, the correct results are there in our register file:

1. R1 = 0X3 = result of OR R1,R2,R3, R1 = 2 OR 3 = 3.
2. R2 = 0X2, since there isn't any instruction that overwrites it.
3. R3 = 0X3, since there isn't any instruction that overwrites it.
4. R4 = 0X4, since there isn't any instruction that overwrites it.
5. R5 = 0X5, since there isn't any instruction that overwrites it.
6. R6 = 0X5, result of CAS R6,R4,R5, R6 = max(4,5) = 5.
7. R7 = 0X8, result of ADDI R7,R1,R6, R7 = 3 + 5 = 8.

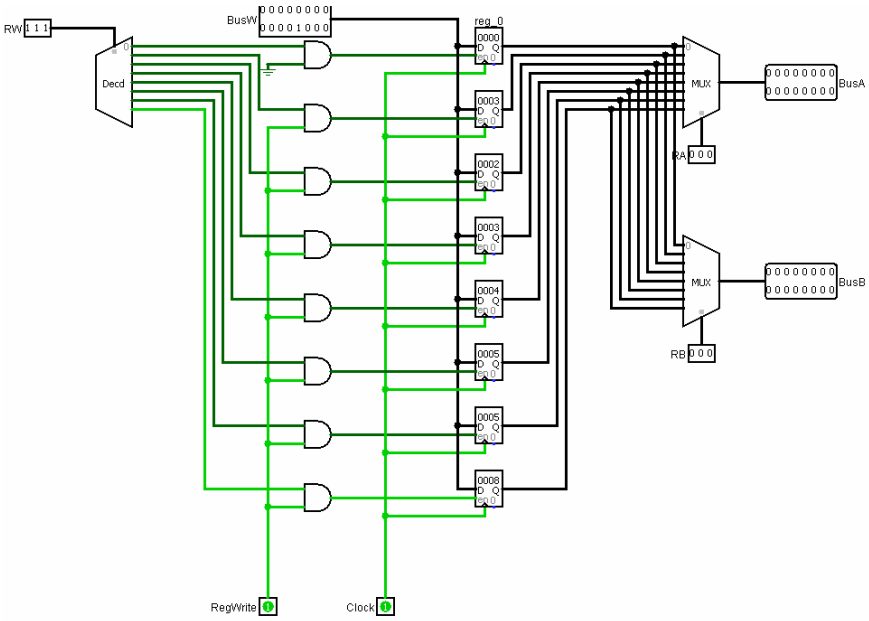

Figure 32: Register File

5.3 Testing Forwarding Units and Stalls- Part2

In this section the **Write_back to Memory** forwarding unit is to be tested by the following instructions, it is role is noticeable when a load is followed by a store in which the value to be stored is in the register in which the loaded data is to be written in:

1. 0X4443 = 0100 010 001 000011 : LW R1,3(R2) -> Loading R1 with the value in memory address R2+3.

2. 0X7844 = 0111 100 001 000100 : SW R1,4(R4) -> Storing R1 in memory address R4+4.

The register file was initialized with the following values:

1. R4 = 7.

The data memory was initialized with the following values:

1. address 3 = 0XC.

In the following figure, the store in the decode stage is captured, the MemWrtie is set to one by the Control_Unit. The load instruction is in the execution, with MemRead signal set to one along with the RegWrite signal and the address found with the help of ALU as: 0(value in R2)+3 = 3.

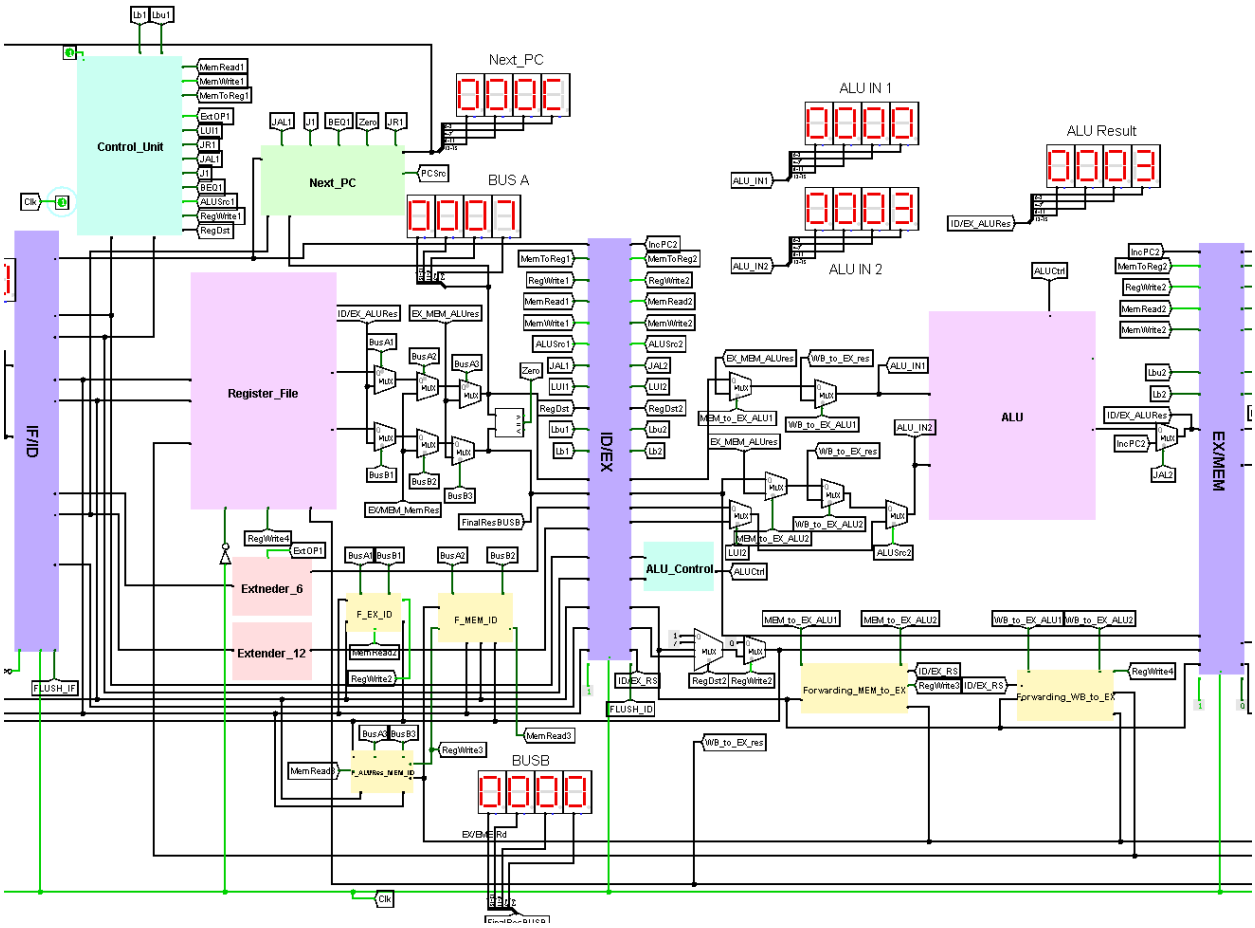

Figure 33: Third Clock Cycle

In the following figure, the store is in the execution stage, with the address to store the value on as: R4 value(7 as read in the decode) + 4 = 0XB. And the load instruction is in memory, with the loaded value shown in the leftmost segment display as C (which is the correct value in address 3).

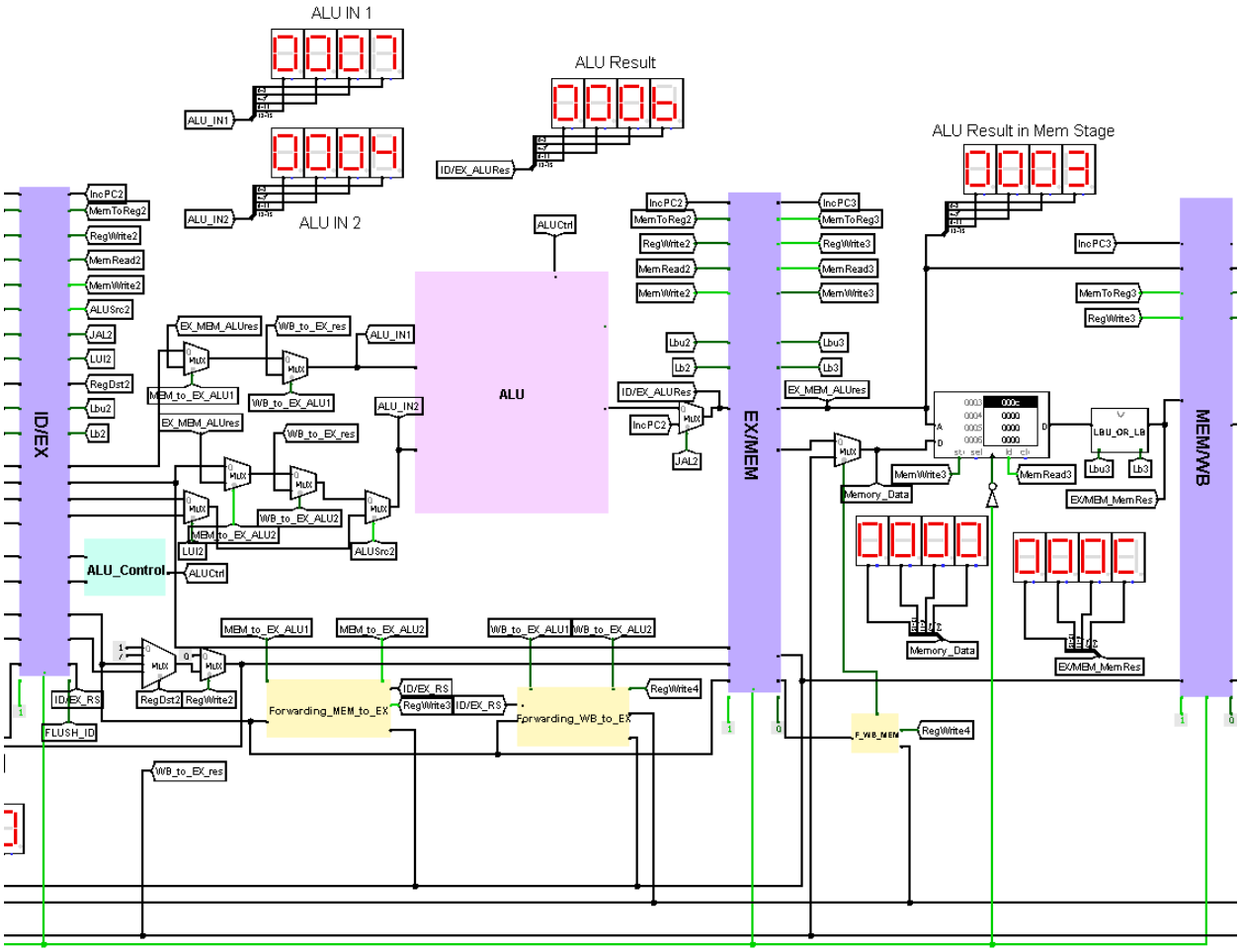

Figure 34: Fourth Clock Cycle

Here, the **Write_back to Memory** forwarding unit is activated after the detection of R1 in both Memory and Write_back stages, and so the multiplexer is singled to pass the value forwarded from the Write_back stage as an input to data memory.

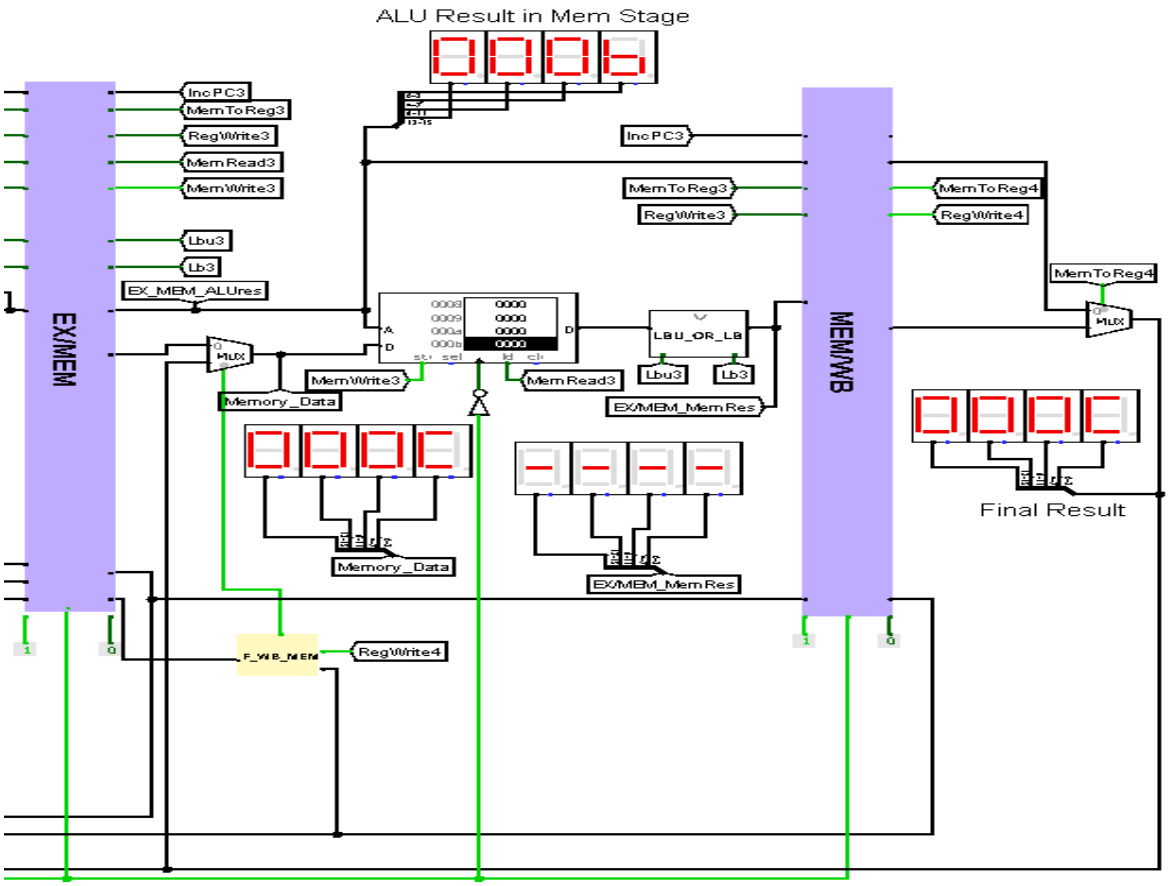

Figure 35: Fifth Clock Cycle

Finally, the last two figures illustrate the register file and the data memory after implementing all the above instructions, the correct results are there in our register file and data memory:

1. R1 = 0XC, result of LD R1,3(R2), R1 = Mem(3+0) = 0XC.
2. The rest registers remain the same as in the beginning of the program since there isn't any instruction that overwrites them.

For the data memory there was an update on it, which is Mem(0Xb) = 0XC, the result of SW R1,4(R4), Mem(4+7)=0Xc.

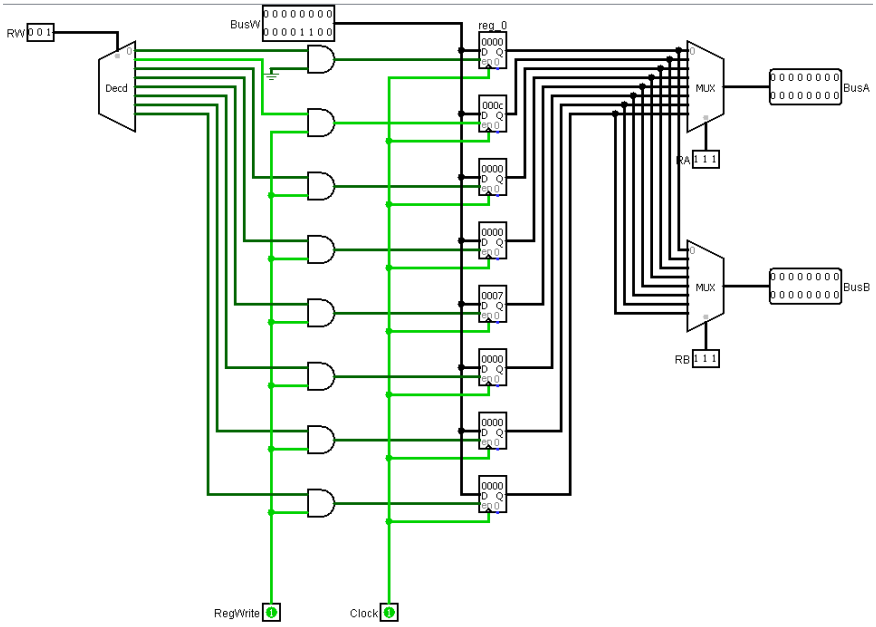

Figure 36: Register File

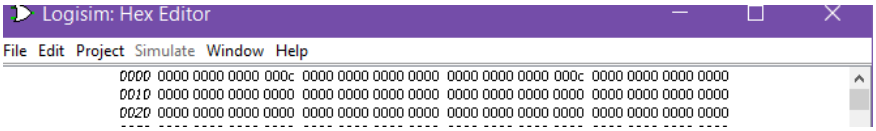

Figure 37: Data Memory

5.4 Testing Forwarding Units and Stalls- Part3

In this section the **execution to decode** forwarding unit and the **memory to decode** forwarding unit are to be tested, also the branch will be taken introducing the need for a stall cycle. The following instructions were used:

1. 0X22C1 = 0010 001 011 000001 : ORI R3,R1,1.
2. 0X0971 = 0000 100 101 110 001 : SLT R6,R4,R5.
3. 0X9782 = 1001 011 110 000010 : BEQ R3,R6,2.
4. 0X22C1 = 0010 001 011 000001 : ORI R3,R1,1 (Stored in the Branch address).

The register file was initialized with the following values:

1. R4 = 4.

2. R5 = 5.

3. others = 0.

In the following two figures, the the fourth clock cycle of the datapath is being captured, with the following:

1. The ORI instruction is in the memory stage, with correct result of ORing the immediate 1 with the value of 0 stored in R1.

2. The SLT instruction is in the execution stage, with the result of the ALU set to 1 as the value in R4(4) is less than the value of R5(5) and so R6 is to be set to the value of 1.

3. The BEQ instruction is in the decode stage, having both the execution_to_decode and memory_to_execution forwarding units both activated. The BEQ instruction needs the values of both R3 and R6, R3 is now forwarded from the memory stage to first operand of the branch and R6 is forwarded from the execution stage to the second operand of the branch.

4. The address that the branch would write on the PC in case it is taken would be as illustrated in the implmenetation section; the sign extended 6-bit immediate, shift lefted one bit and added to the value of the incremented PC. The immediate in our case is 2, shifting it one bit to the left and adding the value of imcremented PC (6 in this case) results in a 10 or 0XA - shown in the upper digit display in the decode stage.

5. The value of the PC in the decode stage is currently 6.

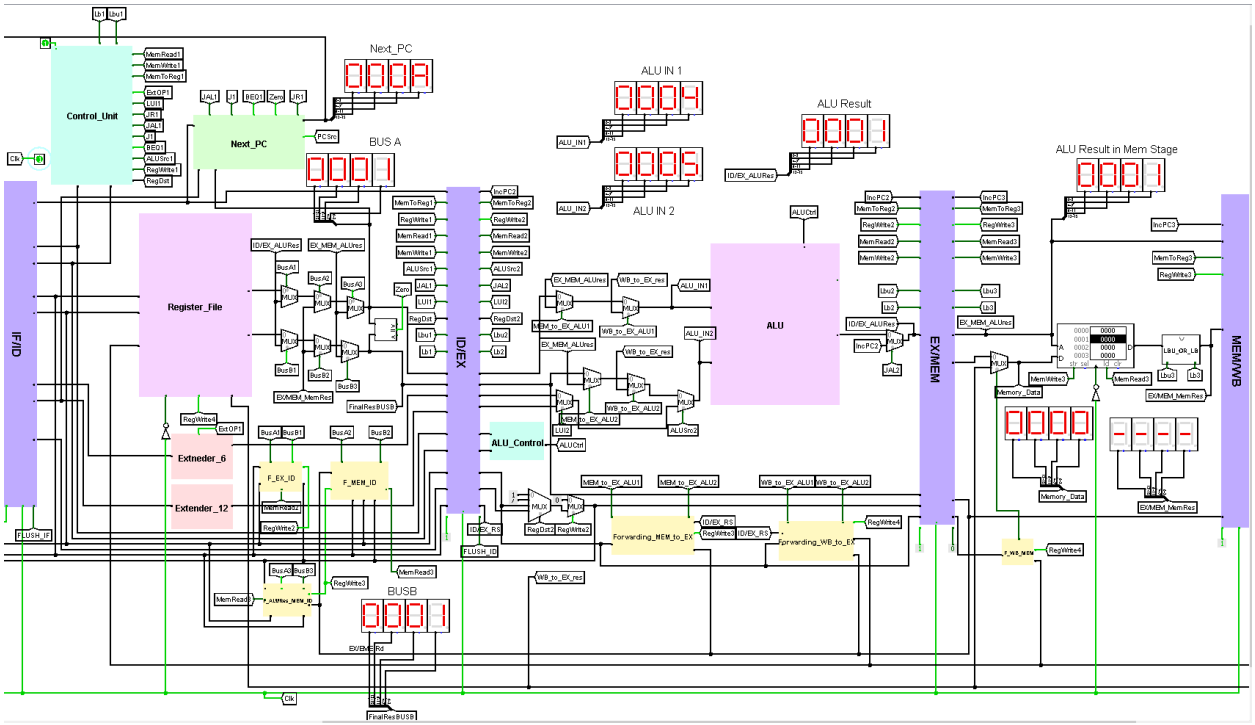

Figure 38: Fourth Clock Cycle - Part1

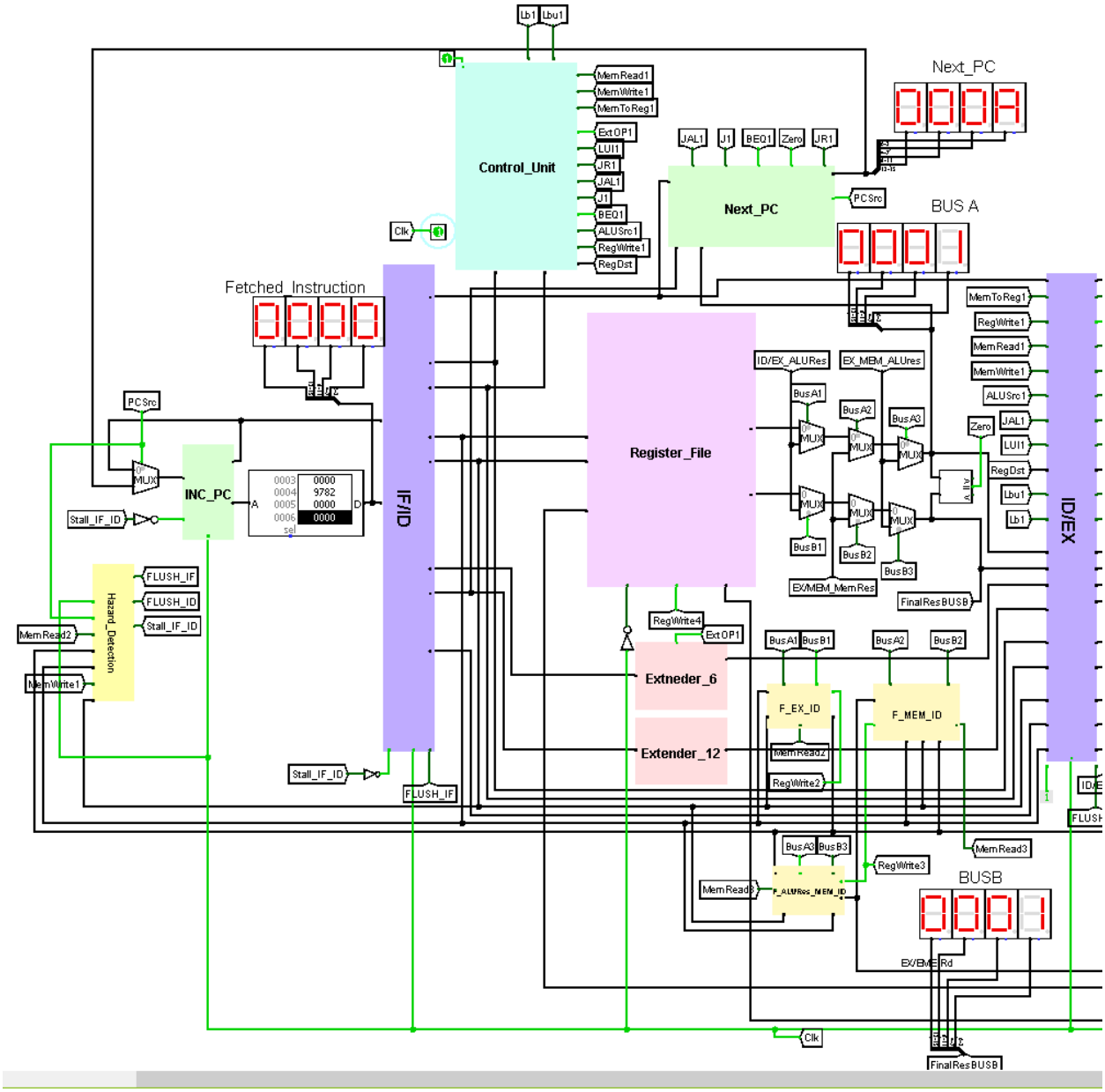

Figure 39: Fourth Clock Cycle - Part2

After another clock cycle, and after deducing that the branch is taken, the value of the PC is modified to the value calculated due to the branch instruction and a stall cycle is introduced in the decode stage(1111 as op code is now in the decode, declaring a stall cycle).

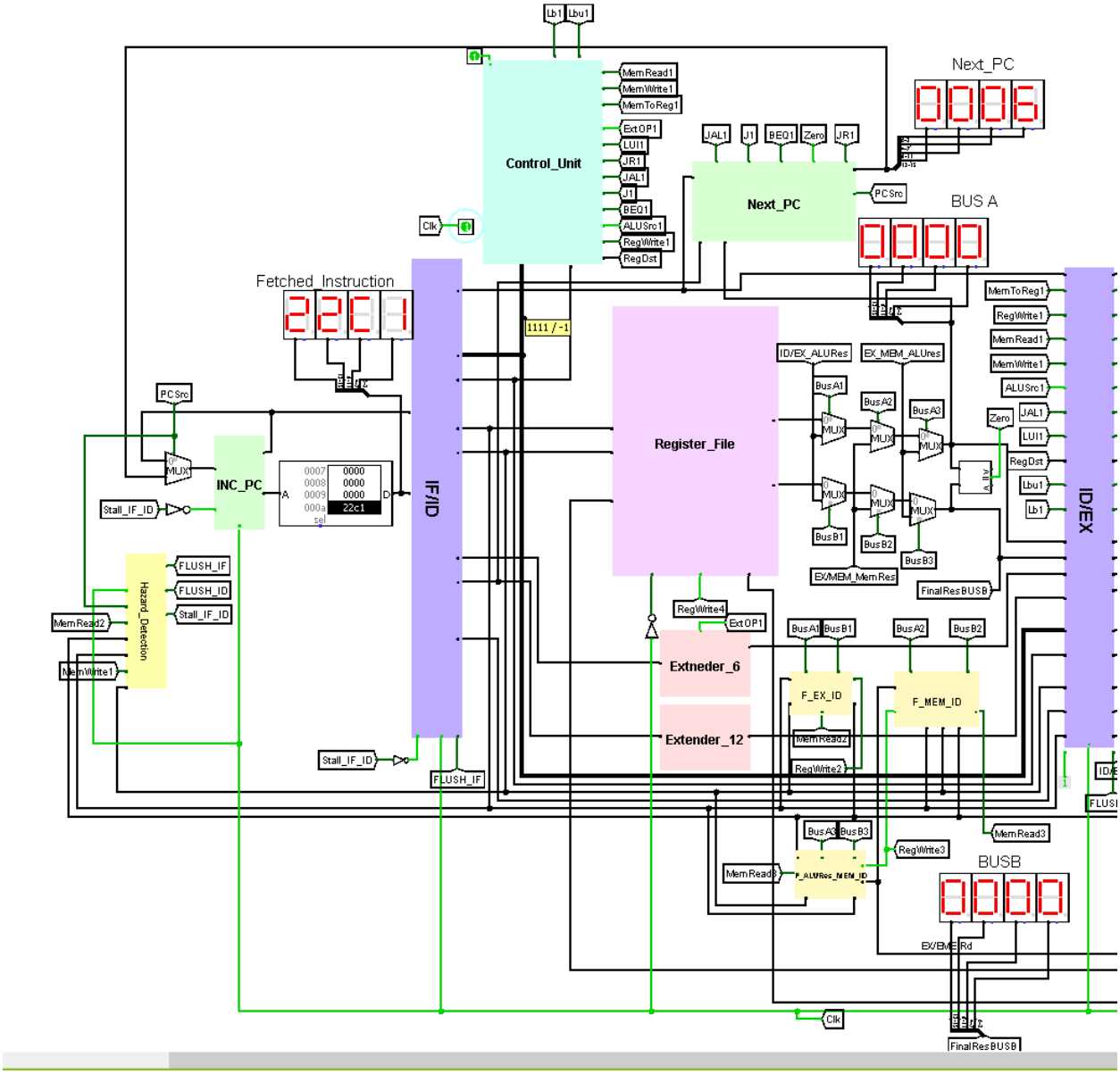

Figure 40: Fifth Clock Cycle

The following figure, aims to make it clear that our smart datapath is working properly when a branch is taken, and so the ORI R3,R1,1 instruction is now in the decode stage and the PC is moved further to the next instruction.

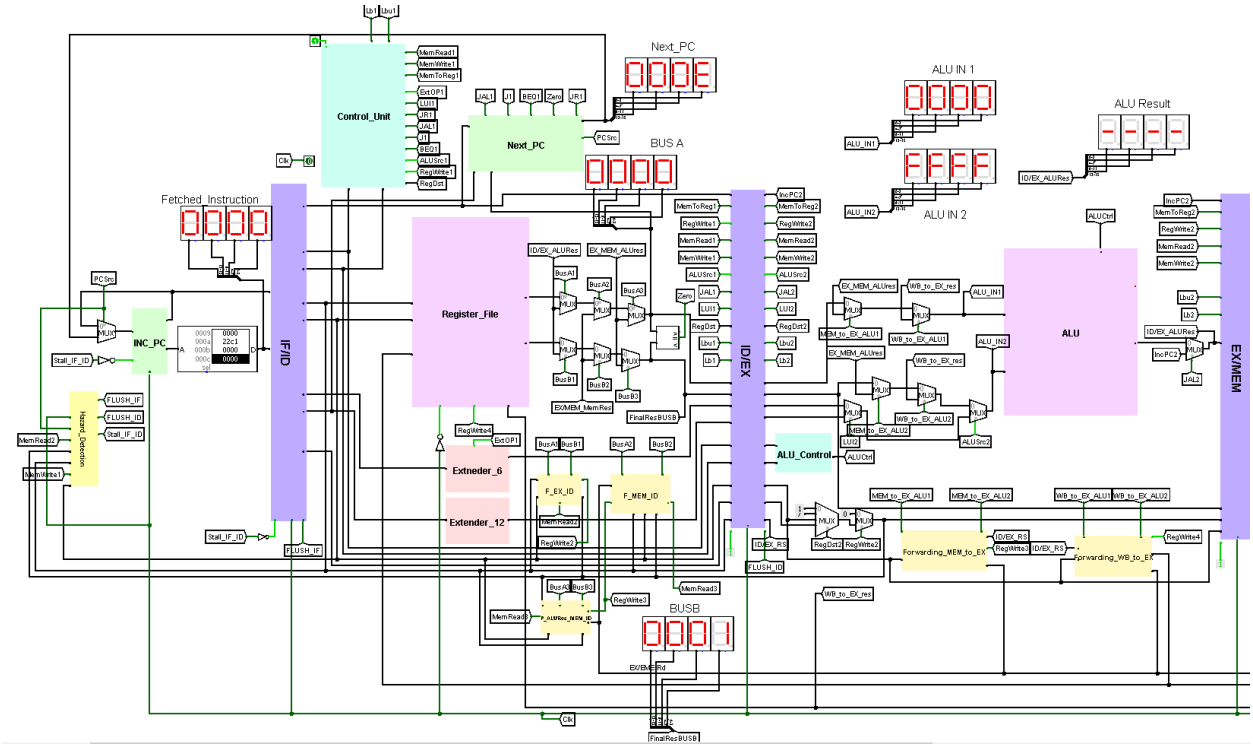

Figure 41: Sixth Clock Cycle

Finally, in this last figure, the register file after implementing all the above instructions, the correct results are there in our register file:

1. R3 = 0X1, result of ORI R3,R1,1, R3 = 0 OR 1 = 1.
2. R6 = 0X1, result of SLT R6,R4, R5 = 4 < 5 = 1.
3. The rest registers remain the same as in the beginning of the program since there isn't any instruction that overwrites them.

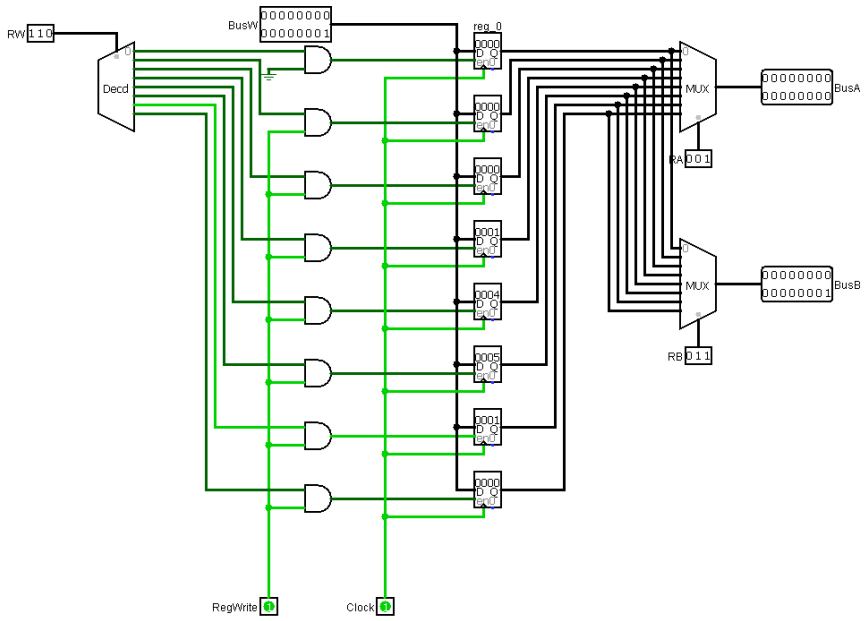

Figure 42: Register File

5.5 Testing Forwarding Units and Stalls- Part4

In this section a load instruction is followed by an R-type instruction with dependency, in which a stall is needed and introduced as follows:
The following two instructions were used:

1. 0X544B = 0101 010 001 001011 : LBU R1,B(R2) -> Loading R1 with the zero-extended-byte in memory address R2+B.
2. 0X031B = 0000 001 100 011 011 : ADD R3,R1,R4.

The register file was initialized with the following values:

1. R4 = 3.
2. Other = 0.

The data memory was initialized with the following values:

1. address 0XB = 0XAB.

In the following figure, the pipeline is captured with the load instruction in the execution stage(with the address to load from is B as shown in the digit display, which is the summation of R2 (zero value)+immediate(B)) and the add instruction is in the decode stage(with the value of R1 fetched as 0 and the value of R4 fetched as 3). Now, with the help of the Hazard Detection

unit, it was detected that a stall is needed as a load is followed by an R-type instruction as shown in the second figure.

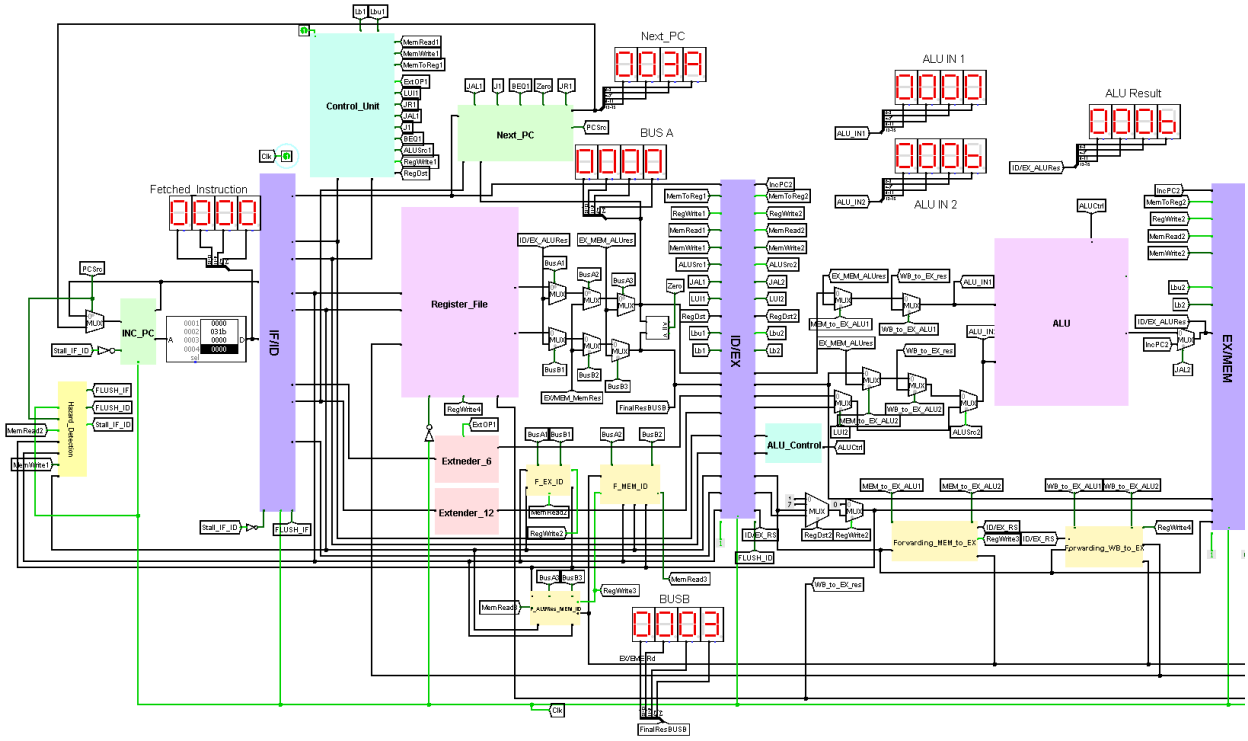

Figure 43: Third Clock Cycle

Here and after detecting the Hazard, a stall is needed between the load AND the execute stage and so introduced. It is noticed that the add instruction is still in the decode stage, and a stall is now in the execute with all control signals implementing a write to either a register file or memory are disabled. This stall was introduced after detection, by flushing the buffer between the decode/execute and disabling both the incrementing of PC value and the buffer between the fetch and decode stages. It is also noticed that the memory to forwarding unit is enabled here, as the load is done to R1 which is Rs in the add instruction. An important note to consider for this case is that without the help of the memory to forwarding unit, at least two cycles would have been needed.

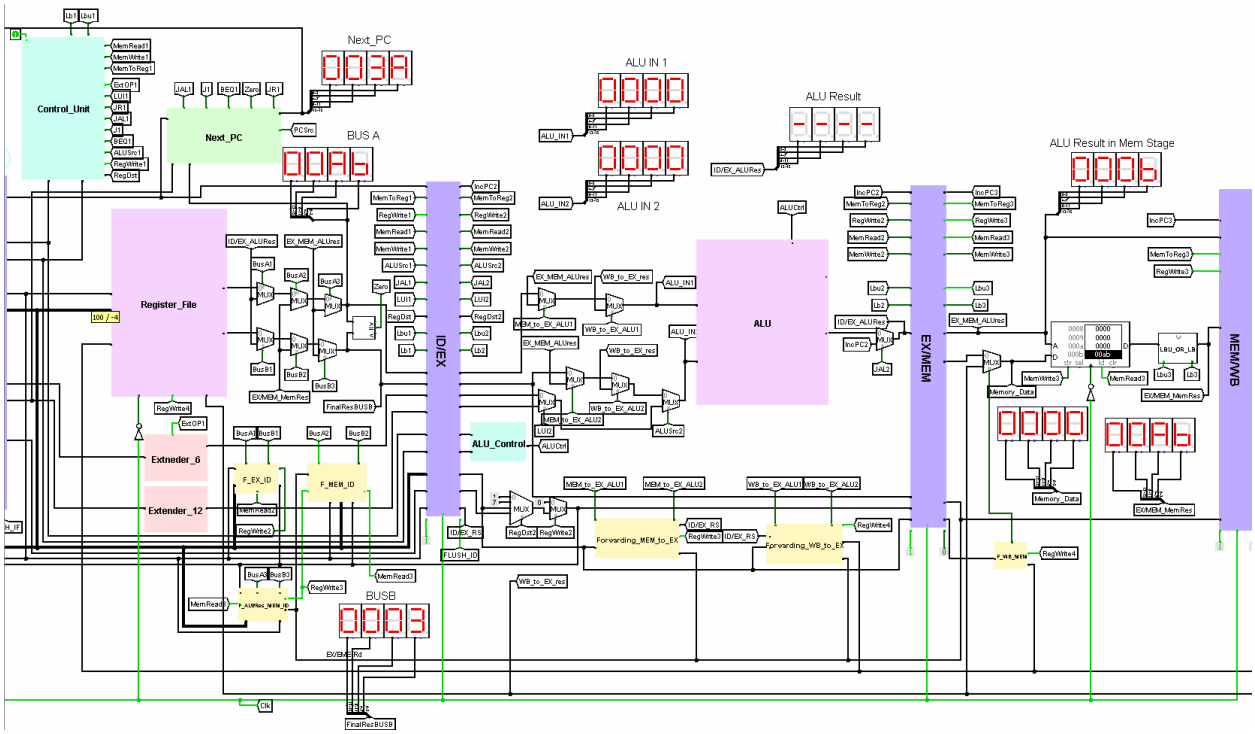

Figure 44: Fourth Clock Cycle- Introducing a Stall

Finally, in this last figure, the register file after implementing all the above instructions, the correct results are there in our register file:

1. R1 = 0XAB, result of LBU R1,B(R2), R1 = Mem(11+R2)=Mem(11).
2. R3 = 0XAE, result of ADD R3,R1,R4, R3 = 0XAB + 3 = 0XAE.
3. The rest registers remain the same as in the beginning of the program since there isn't any instruction that overwrites them.

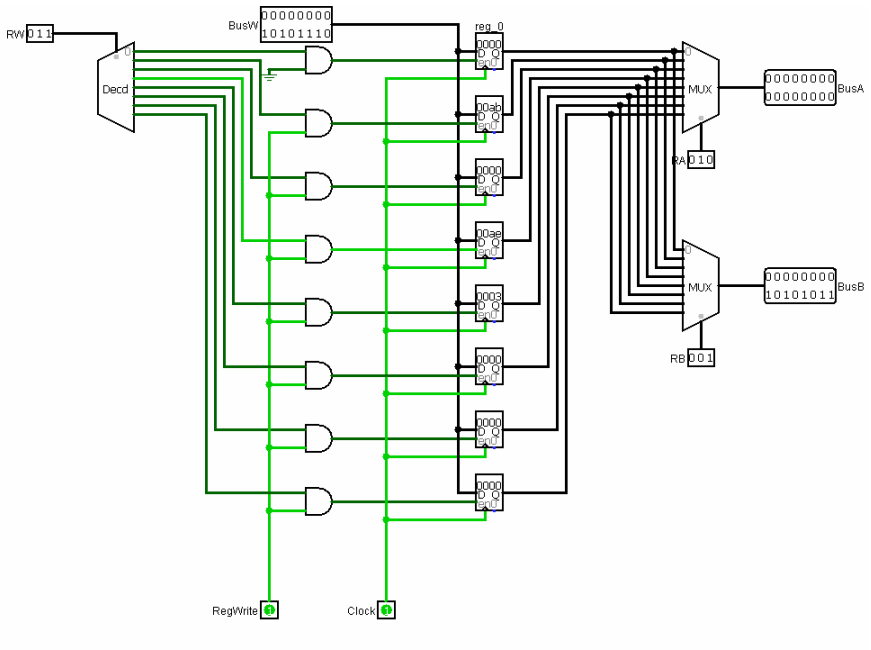

Figure 45: Register File

5.6 Testing Stalls- Part5

In this section a load instruction (LWS) is followed by a JAL instruction. This case consider the correct and complete implementation of the JAL instruction.
The following two instructions were used:

1. 0X04cc = 0000 010 011 001 100 : LWS R1, R2, R3 -> Loading R1 with the value in memory address R2+R3.

2. 0Xb004 = 1011 000000000100 : JAL 4.

The register file was initialized with the following values:

1. R2 = 2.

2. R3 = 3.

3. Other registers = 0.

The data memory was initialized with the following values:

1. address 0X5 = 0X7.

In the following figure, the pipeline is captured in the third clock cycle. The LWS instruction is in the execution stage, with the address to load from is the summation of R2 and

R3, and is shown correctly in the digit display. The JAL instruction is in the decode stage, with the control unit detecting it as a JAL correctly. The value of the incremented PC is 4 (it will be written at R7 at the end of the execution of this instruction). And the value of the new address to which the PC will be pointing next is 8, calculated as follows: MSB(3-bits) of incremented PC(000) Concatenated with 12-bit immediate(000000000100) concatenated with zero; producing an 8.

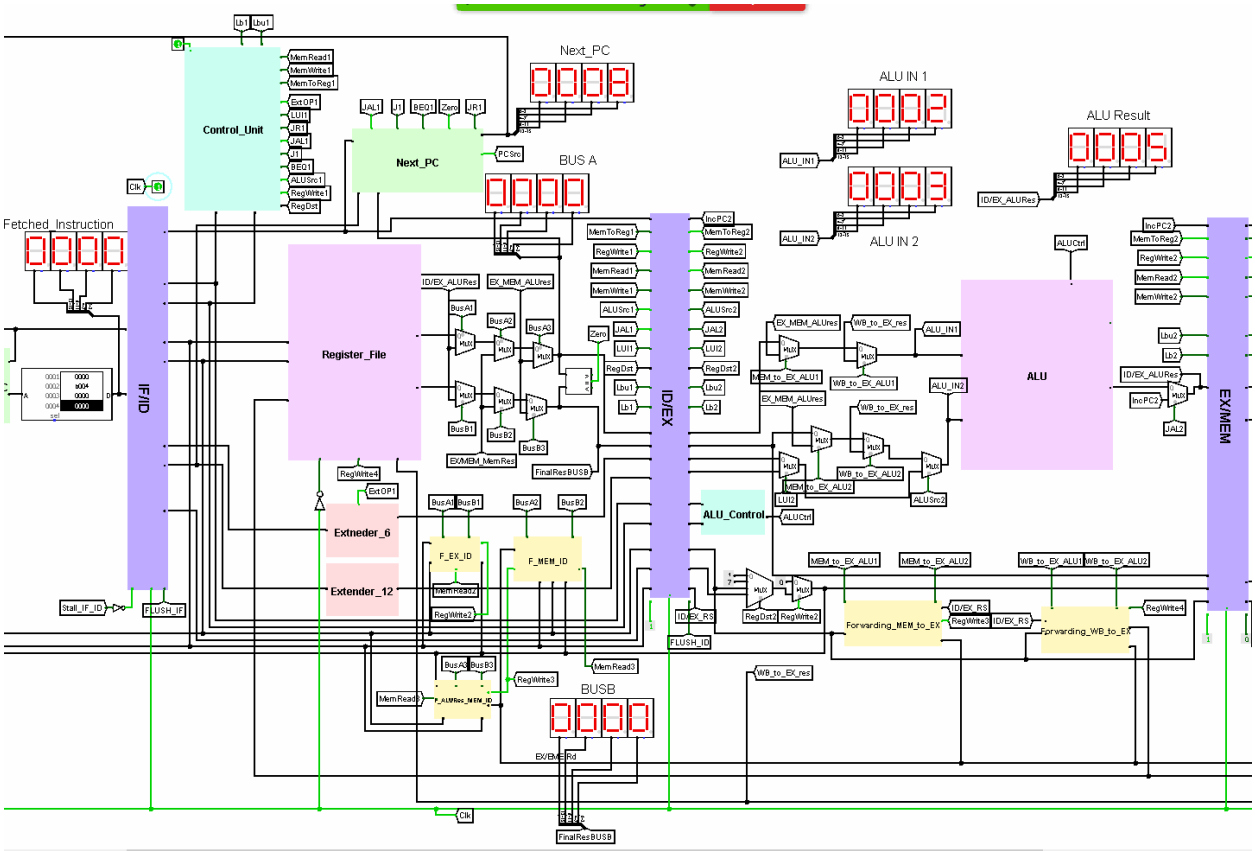

Figure 46: Third Clock Cycle

After a clock cycle, the PC will be now pointing to 8 (the value calculated from the JAL) and a stall is introduced in the decode stage as shown (the op code is 1111 used to indicate a stall). The stall is introduced here by flushing the buffer between the fetch/decode stage and then fetching the new correct instruction.
The second following figure is for the same clock cycle, capturing the LWS instruction in the memory stage with the correct value from the correct address being loaded. And the JAL in the execution stage.

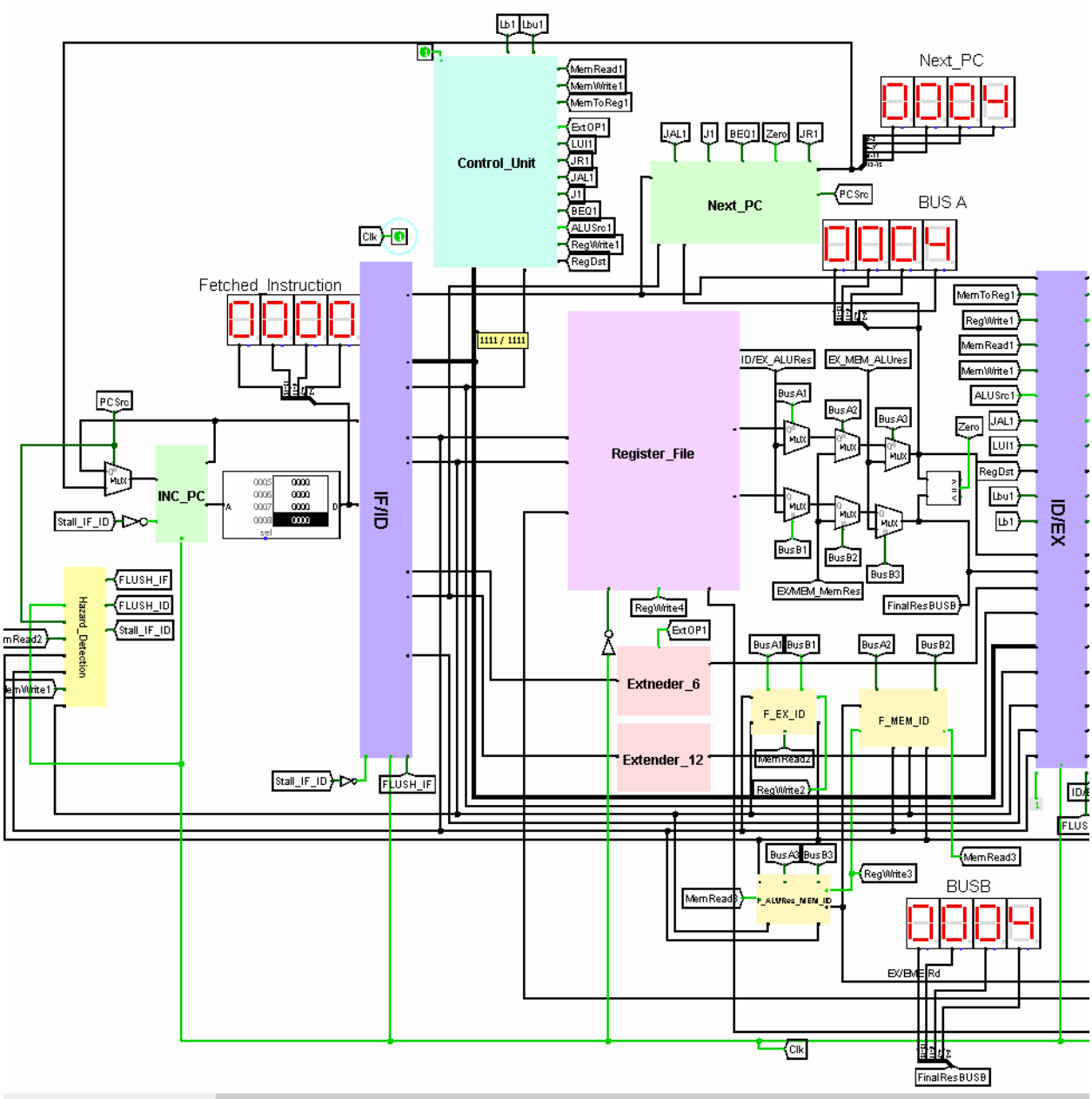

Figure 47: Fourth Clock Cycle - Part1

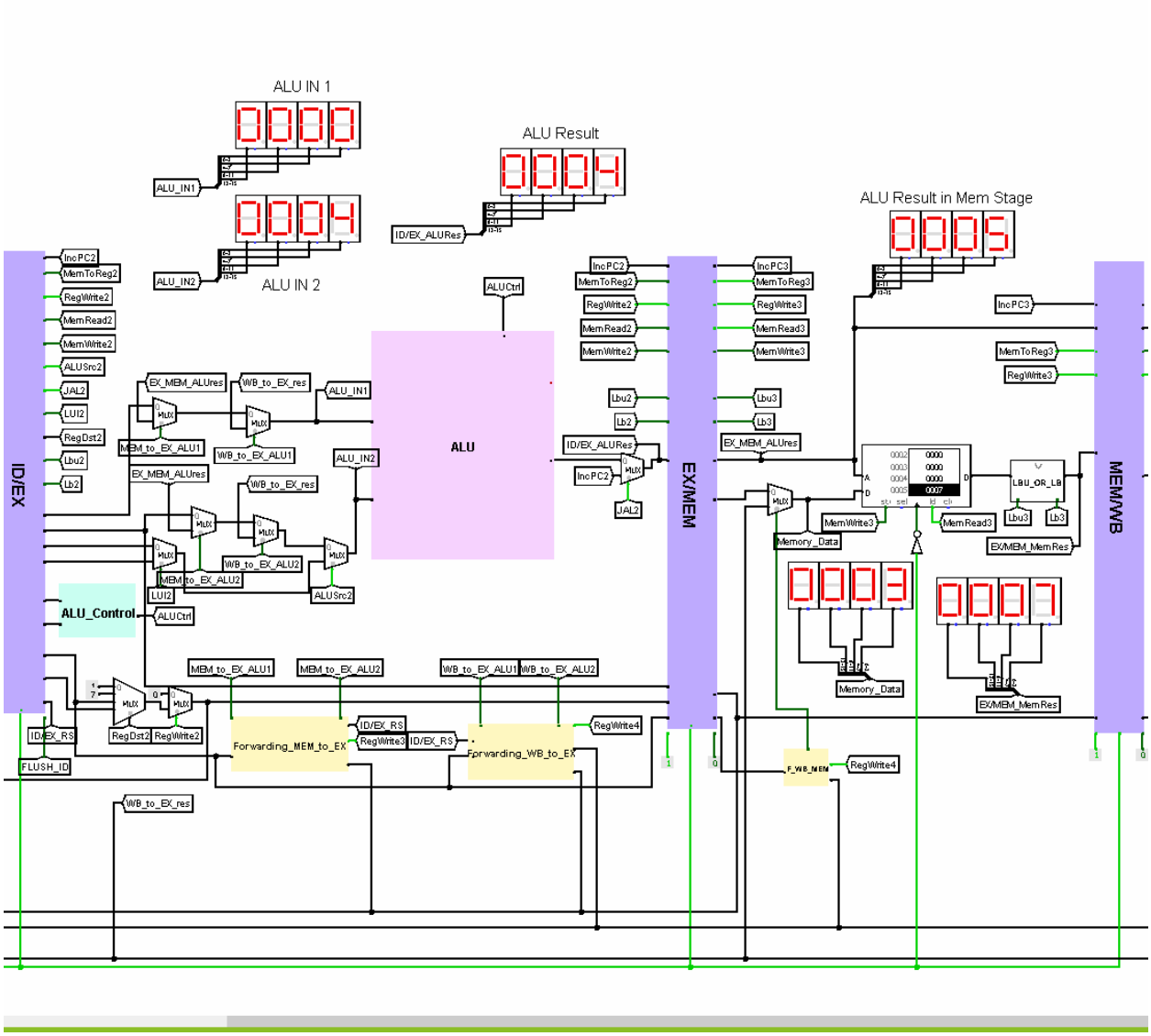

Figure 48: Fourth Clock Cycle - Part4

After another clock cycle, the stall is moved further to the execution stage, and the instruction in address 0X0008 is in the decode stage, with the PC now with the Value of 0X000A as expected with a normal increment. It is noticed that the instruction in the decode stage is a JR as the signal JR is activated with the help of the control unit.
For the same clock cycle, considering the memory and write back stages. It is clear that the JAL is now in the memory with no values being loaded or stored in memory as (MemRead and MemWrite signals are deactivated) also that the load instruction is currently in the write back stage aiming to write the loaded value to the R1 register.

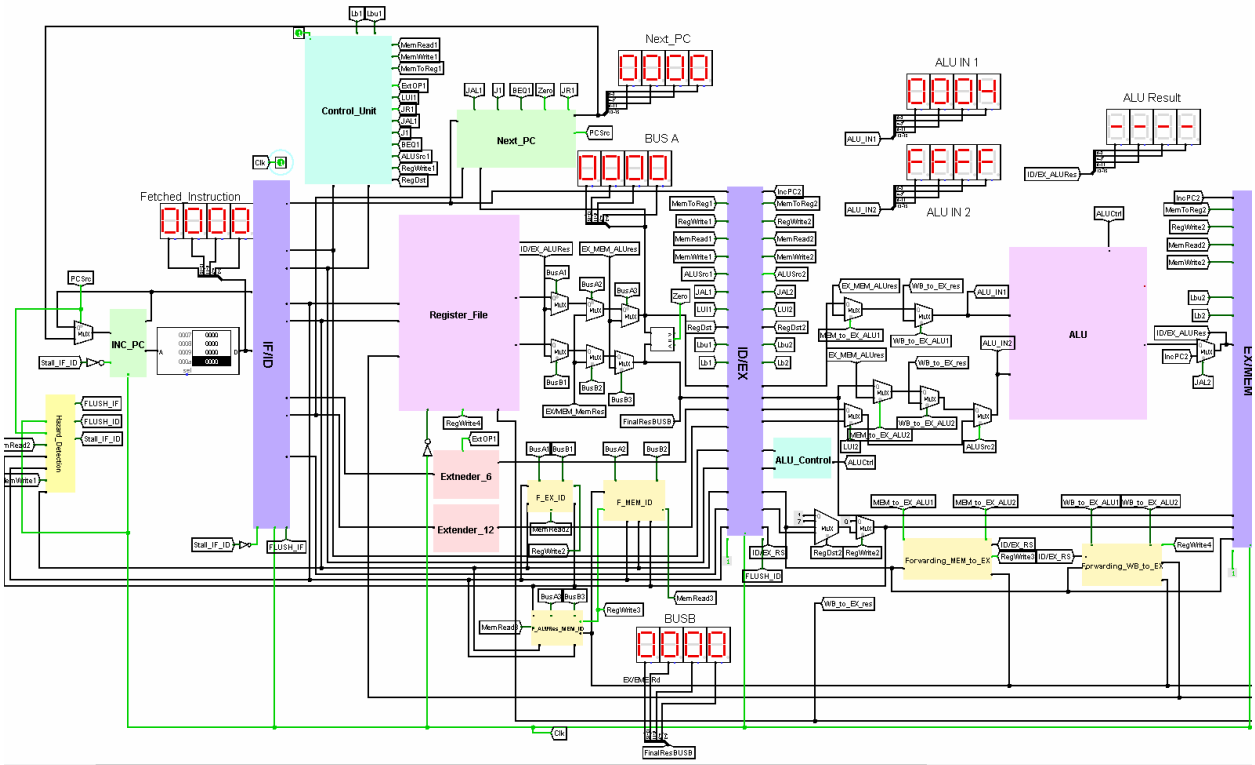

Figure 49: Fifth Clock Cycle

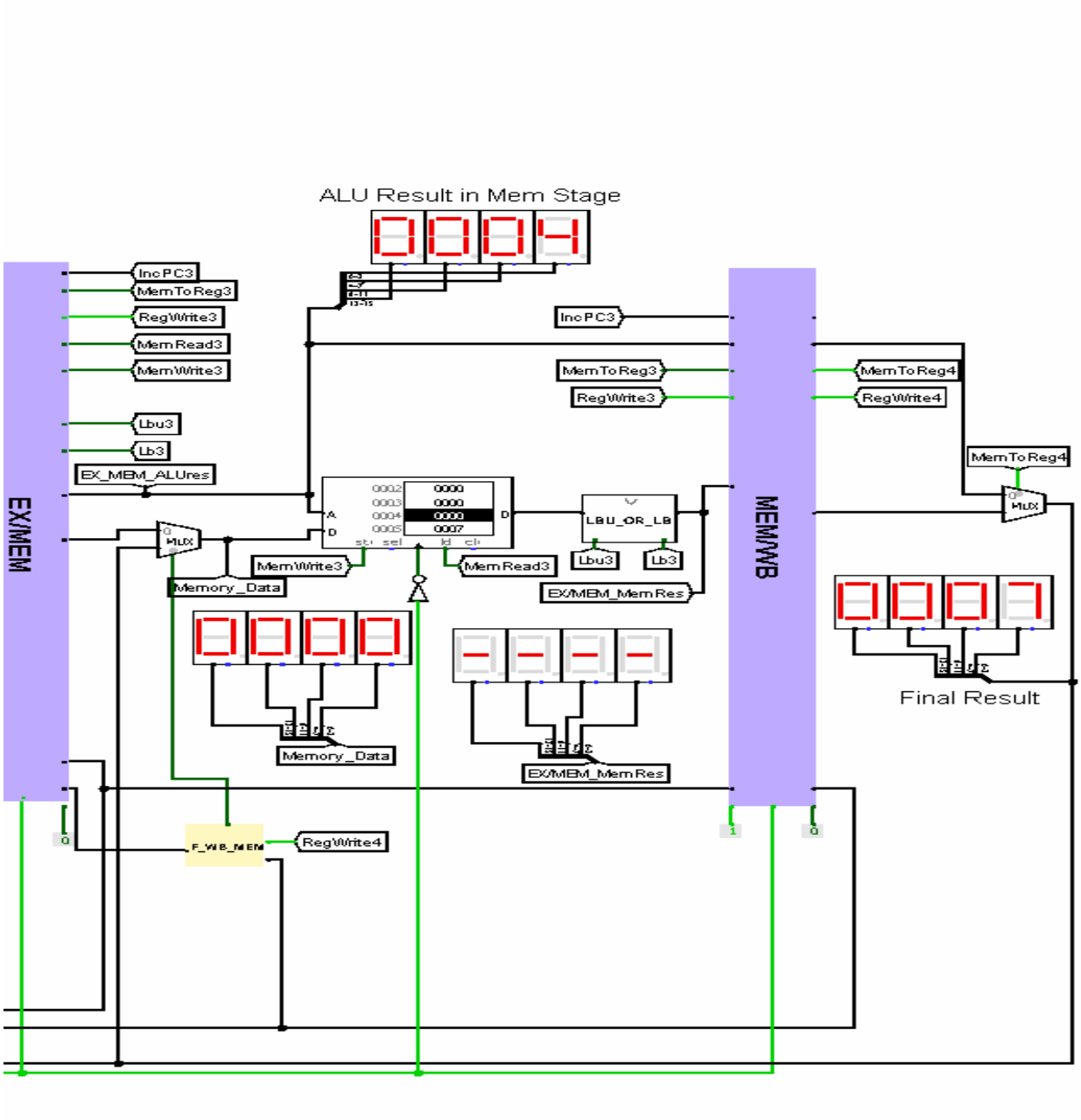

Figure 50: Register File

Finally, in this last figure, the register file after another implementing the instructions, the correct results are there in our register file:

1. R1 = 0X7, result of LWS R1, R2, R3, R1 = Mem(R2+R3) = Mem(5) = 0x7.
2. R7 = result of Jal 4, R7 = PC+1 = incemented PC then = 4.

3. The rest registers remain the same as in the beginning of the program since there isn't any instruction that overwrites them.

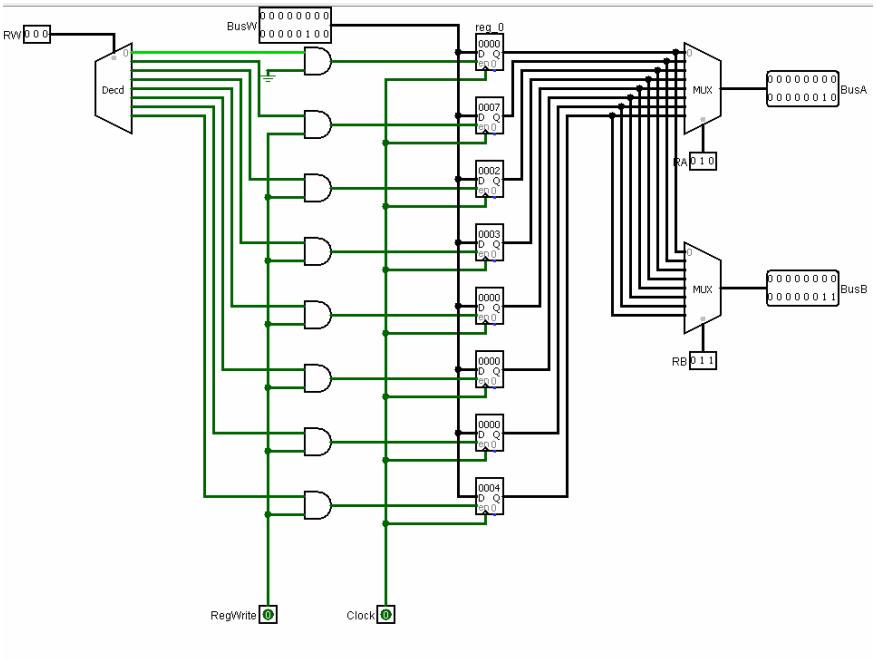

Figure 51: Register File

6 CONCLUSION AND FUTURE WORK

In this project we have learned a lot about the pipelined structure and the implementation of it. We have succeed in implementing a pipelined 16-bit processor using Logisim simulator with all the aims that were required in this project like implementing the forwarding technique, and getting the minimum possible stall cycles which is 1 stall cycle in case of load followed by an instruction that depend on it (R/W) or a miss prediction in case of a branch or jump instruction. As a future plan, this project encourages us to dive more and more in computer architecture techniques and how to optimize the possible designs that exist nowadays.

7 REFERENCES

1. https://www.cise.ufl.edu/~mssz/CompOrg/CDA-proc.html#:~:text=Datapath%20is%20the%20hardware%20that,movement%20between%20ALU%20components%2C%20etc.
[Accessed on 1/June/2021 at 1:00 pm]