SUMMER PROJECT REPORT

Course: MEL G642 **Date**: 15th July 2023
Name: VLSI Architecture Design **Sem**: 2nd / HD
Submitter Detail:

Bharathi Shrinivasan T R
2022H1400182P (M.E. Embedded Systems)

Objective: Understand and comprehend the design philosophy of RISC-V Processor architecture.

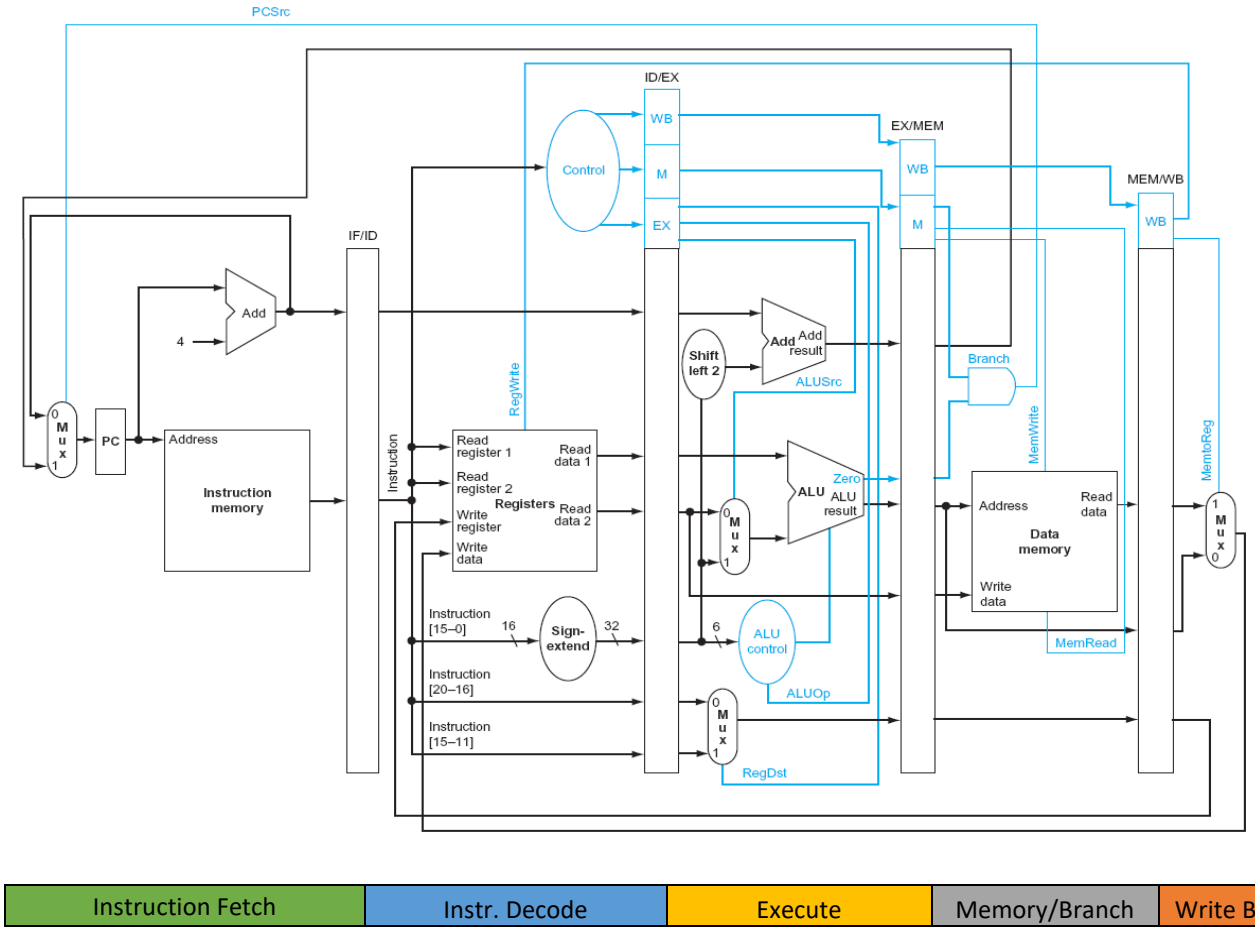
Reference: "*Computer Organisation and Design*" – David A. Patterson, John Hennessy

Project:

1. Learn conceptual design strategy of RISC processor architecture. Covering execution stages, pipelining, etc.,
2. Develop a custom 32-bit MIPS (**M**icroprocessor without **I**nterlocked **P**ipeline **S**tages) RISC-V architecture prototype using HDL modelling of the logic circuits and implementing same on Xilinx's ZedBoard (Zynq 7000 series FPGA).
3. Machine-code a program to compute Fibonacci series using the implemented RISC-V processor.

Project database links:

1. Complete Vivado (2020.2) source code file: Drive Link
2. Simulation results and report: Drive Link



1.1 Block diagram of a complete RISC-V architecture

Table of Contents

Design Workflow carried out during this project

The Design of RISC based microprocessor involves majorly 5 phases –

1. Defining custom instruction format and its micro-operations at each stage respectively.
2. Designing of Datapath and Local controlling unit for each stage.
3. Designing Pipelining strategy across each stage.
4. Designing Hazard detection and addressing (I.e., Port-forwarding) unit.
5. Testing and verification.

This section shows the water-flow model of the work carried in this complete design.

Define Programmer's Instruction set format and stages operation desired. List all supporting instructions, its Opcode and anatomy.

In a spreadsheet, List out all the instructions and define its related control signals applicable at each stage of its operation. Use this information to design the Local control module for each stage.

Design Data path and Local Control for all stages

(i) IF Stage - Program memory (ROM) module, PC address branching module.

(ii) ID Stage - Register File module, Register decoder module.

(iii) EX Stage - ALU, Local control to steer ALU operation.

(iv) MEM Stage - Data memory (RAM) module, Brancher module, Local control module to steer branching and memory write operation.

(v) WB Stage - Result Selector module, Local control module to steer register write operation.

Pipeline Register Mapping - In spreadsheet declear all the necessary I/Os that that to be sent in the pipe at different stages. Design the adequate pipeline register array and integrate the same between each stage modules.

Devise adequate **clocking strategy** (clock phase) to individual modules and pipeline registers.

Test the design for all comitted instructions. Verify its operation, if any modification/insertion of new control signal/instruction then revise the design from step-3 (Designing of datapath)

Design **Hazard detection and prevention** module - Port-forwarding unit, Hazard Addresser(stalling) unit.

Integrate the same with the design.

Test the design post integration for all instructions. Simulate hazards i.e, RAW, Immediate-LW scenario and check if its response is as desired.

Develop a simple user program (Fibonnaci), hard-code it in Program memory ROM and Run. Test the result.

Programmer's Instruction format and list of supporting instructions

This section details the supported list of programmer's instruction along with custom assigned Op-Code, and function code. The anatomy of our custom instruction set format is as below.

Instruction	31:28	27:24	23:20	19:16	15:12	11:8	7:4	3:0	Operation
R-type	opcode	source	2nd operand	destination				Function	
ADD Rs,Rt,Rd	1	Rs	Rt	Rd	0	0	0	0	Rd<-Rs+Rt
SUB Rs,Rt,Rd	1	Rs	Rt	Rd	0	0	0	1	Rd<-Rs-Rt
AND Rs,Rt,Rd	1	Rs	Rt	Rd	0	0	0	2	Rd<-Rs&Rt
OR Rs,Rt,Rd	1	Rs	Rt	Rd	0	0	0	3	Rd<-Rs|Rt
XOR Rs,Rt,Rd	1	Rs	Rt	Rd	0	0	0	4	Rd<-Rs^Rt
I-type	opcode	base	dest/Src	Address OFFSET					
LW Rb,Rd,#OFFSET	2	Rb	Rd	#OFFSET					Rd<-[Rb+#OFFSET]
SW Rb,Rs,#OFFSET	3	Rb	Rsw	#OFFSET					Rs->[Rb+#OFFSET]
J-type	opcode	reg1	reg2	Address OFFSET					
BEQ R1,R2,#PC-offset	4	R1	R2	#OFFSET					if(R1==R2) GOTO PC+1+#OFFSET
	opcode	Direct Address							
JUMP #PROGLINE	5	#PROGLINE							GOTO #PROGLINE
NOP	0	0	0	0	0	0	0	0	Nothing

In this project,

- We have taken instructions of each that fall in wide category i.e., R-type, I-type, J-type, direct addressing, Indirect plus base addressing, PC-relative addressing, pseudo-direct addressing. Each is 32 bit long.
- The size of register field is reserved 4-bit as it accesses registers from R0-R15.
- Opcode is reserved 4-bit, this project requires 6 unique function type. Having multiple of 4bit is always advantageous as it helps to group it as nibble each.
- Branch conditional (i.e., BEQ) is PC-relative addressing mode instruction, and we have allocated 5 nibble / 20-bit. It can be programmed to branch in range of PC ± 2^{19} instructions.
- Jump instruction is pseudo-direct addressing mode, and can directly branch the program to 0 - $(2^{28} - 1)$ instructions.

Note, throughout the project, the ROM and RAM memory units are 32-bit organized cells each. Hence program address increment by 1 not 4 (conventionally memory byte organized).

Reference –

MIPS Fields

MIPS fields are given names to make them easier to discuss:

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Here is the meaning of each name of the fields in MIPS instructions:

- *op:* Basic operation of the instruction, traditionally called the **opcode**.
- *rs:* The first register source operand.
- *rt:* The second register source operand.
- *rd:* The register destination operand. It gets the result of the operation.
- *shamt:* Shift amount. (Section 2.6 explains shift instructions and this term; it will not be used until then, and hence the field contains zero in this section.)
- *funct:* Function. This field, often called the *function code*, selects the specific variant of the operation in the op field.

Few examples of instructions and it's opcode –

Instruction	Machine code
ADD R1,R2,R3	32'h11230000
XOR R7,R9,R13	32'h179D0004
LW R1,R4,#55	32'h21400037
SW R1,R7,#55	32'h31700037
JUMP #12	32'h5000000C

The Machine code- 8st nibble corresponds to the opcode and based on the opcode type the addressing mode will differ. In case of R-type instructions opcode=1, the 7nd, 6rd, 5th nibble corresponds to Source Register, Second Operand Register, Destination Register respectively. Then 1st nibble function code. The detailed anatomy is given in the above table.

Instructions and its operation at each stage -

Instruction	Operation	STAGE -1	STAGE -2	STAGE -3	STAGE -4	STAGE -5
R-type		IF	ID	EX	MEM	WB
ADD Rs,Rt,Rd	Rd<-Rs+Rt	Instruction fetch, Increment PC	Read registers - Rs, Rt	Rs OP Rd	None	Write Rd
SUB Rs,Rt,Rd	Rd<-Rs-Rt					
AND Rs,Rt,Rd	Rd<-Rs&Rt					
OR Rs,Rt,Rd	Rd<-Rs|Rt					
XOR Rs,Rt,Rd	Rd<-Rs^Rt					
I-type						
LW Rb,Rd,#OFFSET	Rd<-[Rb+#OFFSET]	Instruction fetch, Increment PC	Read register Rb	Rb + #OFFSET	Read	Write Rd
SW Rb,Rs,#OFFSET	Rs->[Rb+#OFFSET]				Write	None
J-type						
BEQ R1,R2,#PC-offset	if(R1==R2) GOTO PC+1+#OFFSET	Instruction fetch, Increment PC	Read register R1, R2	R1 - R2	PC+#OFFSET Jump if EQ	None
JUMP #PROGLINE	GOTO #PROGLINE	Instruction fetch, Increment PC	None	None	Jump	None
NOP	Nothing	Instruction fetch, Increment PC	None	None	None	None

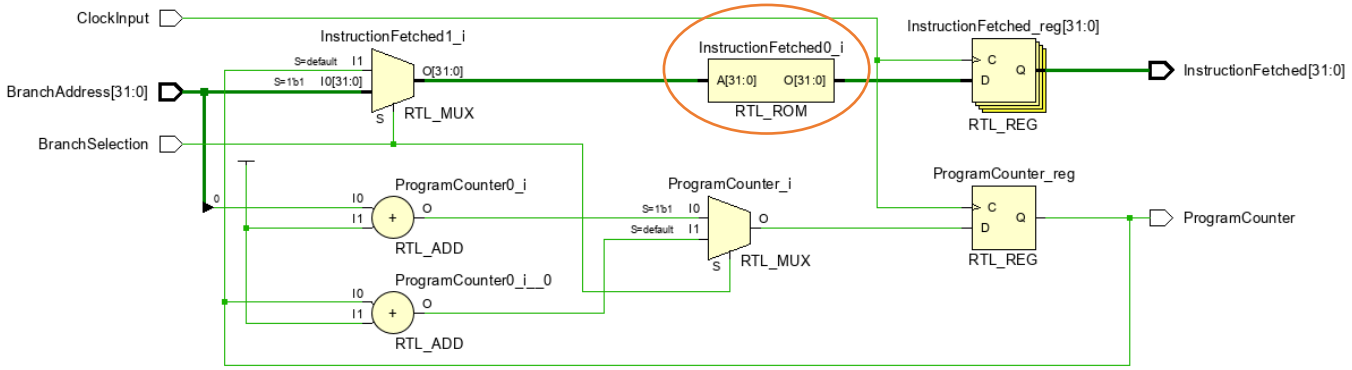
This is very useful to get insight on design of data-path and control requirement at each stage. At each stage we can see what different operations/similar it has to perform to accommodate all instructions. Based on a local controller at each stage the desired operation can be performed for fed opcode. Example, in EX stage, if the opcode is R-type the ALU operation depends on OP field, if opcode is I-type then the ALU operation is + etc., the EX stage data-path should be capable to do all desired operation and exact operation for the fed opcode shall be evoked using control signals locally.

In next section, we shall start building data-path for each stage. While do so, the control signals required shall be noted, which later can be used to build local controller logic.

STAGE-1 Instruction Fetch Datapath and Control

The section details the IF stage data path. Instruction fetch stage has to do –

1. Get the next address for fetching. Either from PC (which holds the address of next instruction) or from branch address in case of any branching signaled.
2. Read the Program memory/ROM for the next address, and thus fetch the instruction for processing.



The Instruction decoder is developed in vivado HDL. The PC is incremented to either its own or from branch address. The new address is then queried in the program memory ROM, which is them sampled at the InstructionFetch register. Reference –

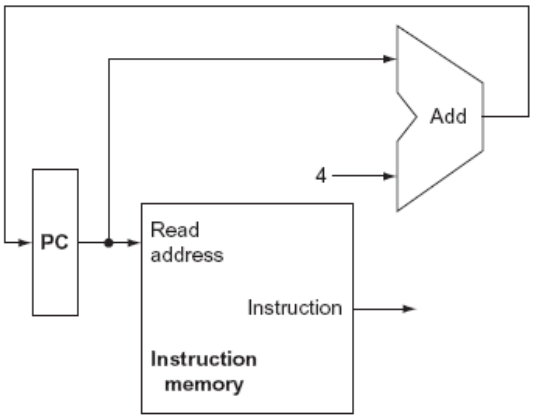


FIGURE 4.6 A portion of the datapath used for fetching instructions and incrementing the program counter. The fetched instruction is used by other parts of the datapath.

The reference image doesn't have the branching logic. But in the later stage the branching unit is added. Also note, for hazard addressing, the IF datapath is further modified to accommodate Stalling option, which shall be detailed in a separate section under Hazard detection and prevention techniques.

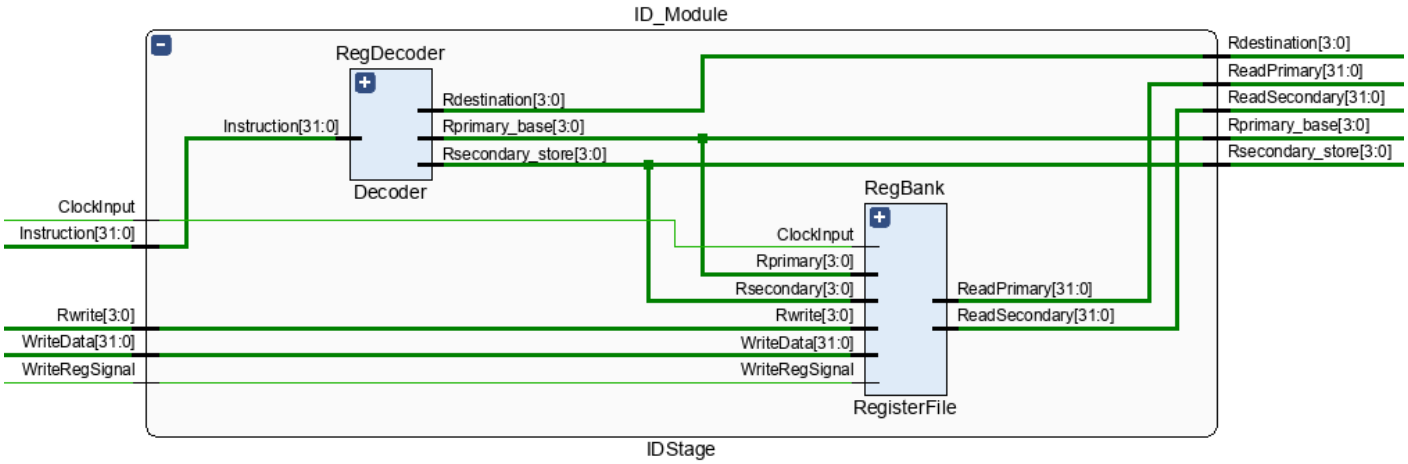
Control signal –

The only control to the datapath is whether the next PC is taken from incrementing current PC or from branching address. In other words, whether the next instruction is at PC location or different branch address. This is signaled by MEM stage (brancher unit), with branch address. Upon the signaling of branch, the Instruction fetched would be altered to read branching instruction. Therefore, there is not separate Local control module for IF stage.

STAGE-2 Instruction Decoder Datapath and Control

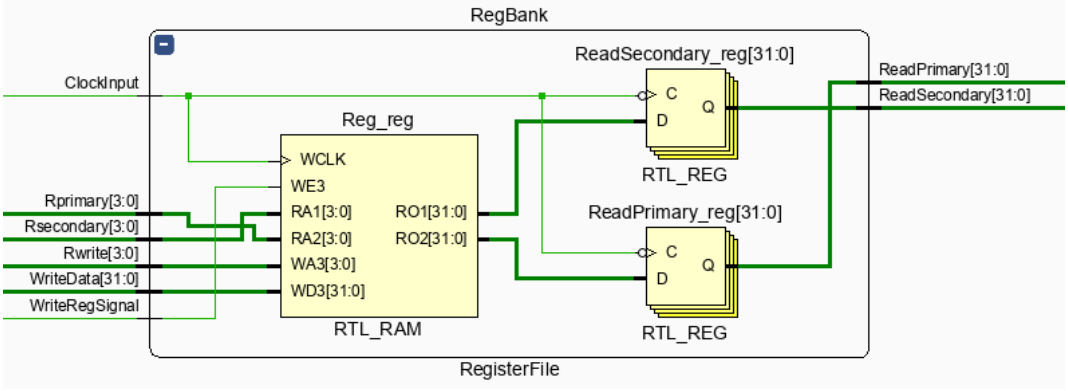
Instruction Decoder is responsible for -

1. Decoding the register details for primary operand and optional secondary operand.
2. Reading the primary and secondary register contents from Register File (which holds R0-R15 32-bit registers used by programmers to hold data and manipulate data).
3. Decoding the Destination register (if in case any register write is applicable) and passing the information to the pipeline.



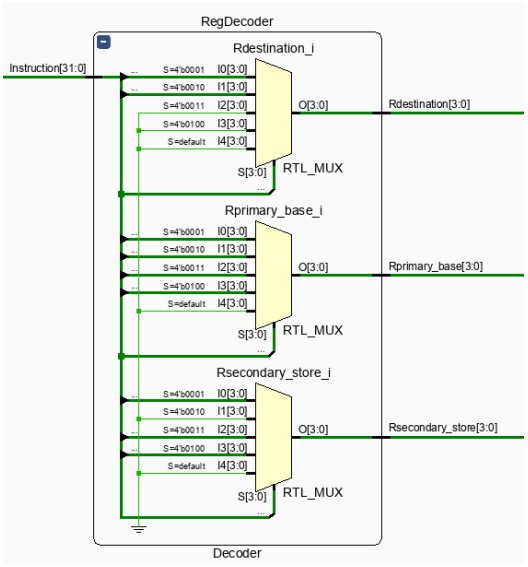
The ID module is packed with Programmer's register Bank containing 16 (32-bit) registers, named R0-R15. Depending on the selected RPrimary, RSecondary, two independent register content read can be done. Note, The ID stage also has some data and control signal from Write-back stage, as WB stage also accesses Register Bank to write the operated value into a destination register of choice. The detailed register writing will be dealt in WB stage. Below we shall be detail the child modules that encompasses the ID stage.

Register File module



The Register file / Register bank is a RAM memory allocated to hold 16 registers of each 32-bit. Used by programmer to manipulate, store data. This memory registers are within the CPU, the access rate and write cycle are very fast. They are simple 31-bit PIPO register.

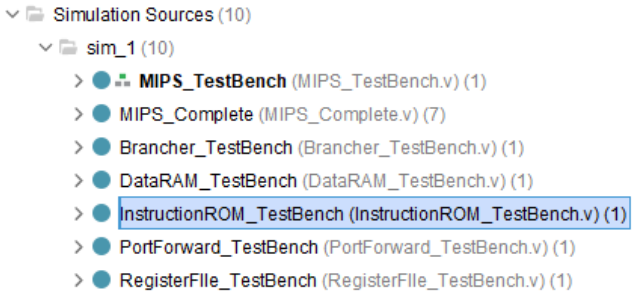
Registers Decoder:



The decoder is responsible for decoding the instruction opcode, extract the register no. for source, optional second operand, thirdly the destination register.

Testing Data path using vivado simulation –

Usually, the best practice is to test each built modules before building something that inherits on this foundation module. Testing screenshots are attached in the appendix section.

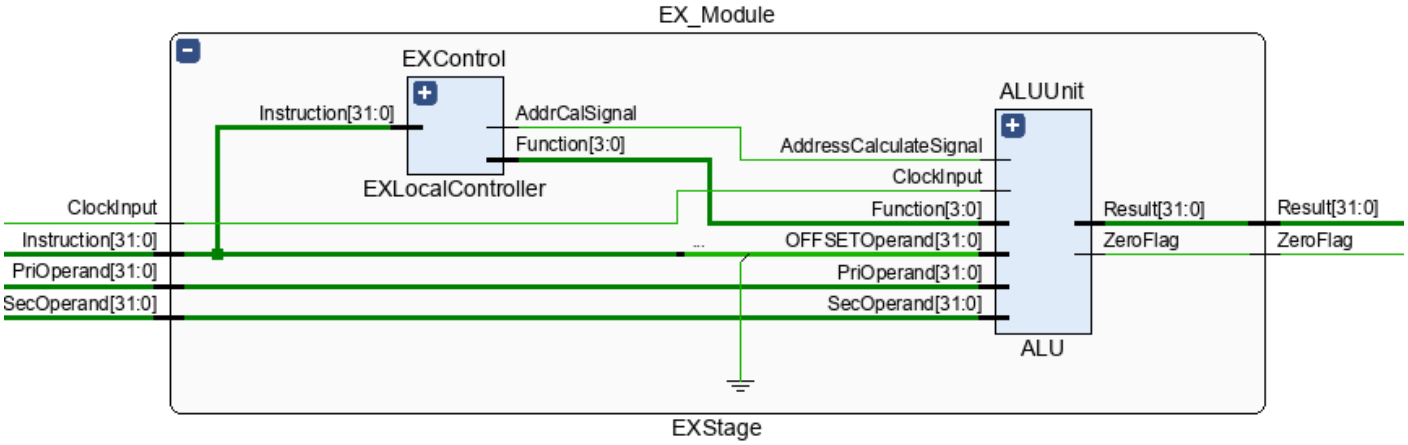


STAGE-3 Execution Datapath and Control

Execution stage is one of the crucial stage, where the actual computation/programmer's required operation takes place. The roles of this stage include –

1. Operating on the primary and secondary operands. Operation based on opcode, based on the programmer specified function code. Like in case of memory read instruction, the ALU performs the address calculation from Base and displacement request as its operands.
2. The local control unit will steer the ALU to perform intended operation. Depending on the opcode, it signals the ALU unit to do address calculation or data and accordingly the second operand is chosen by the ALU, i.e., in case of address calculation the OFFSET component is added with the primary operand (as it contains the base address).
3. The local control unit extracts the OFFSET component in case of memory-based instruction.

The EX stage encompasses ALU Unit, and Local control Unit. ALU Unit does the actual operation, where else the local control unit steers the ALU to what operation it has to do and upon which operand. Below is the block model of the EX stage as developed in vivado.



Reference –

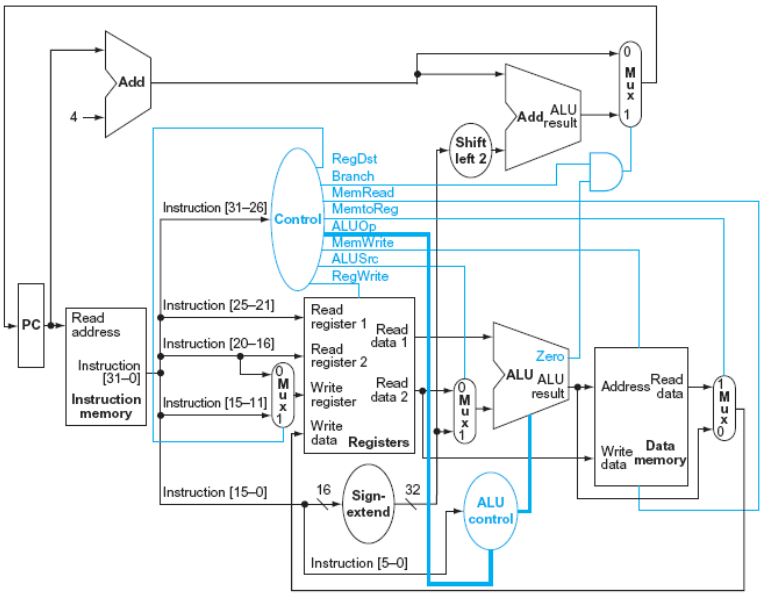
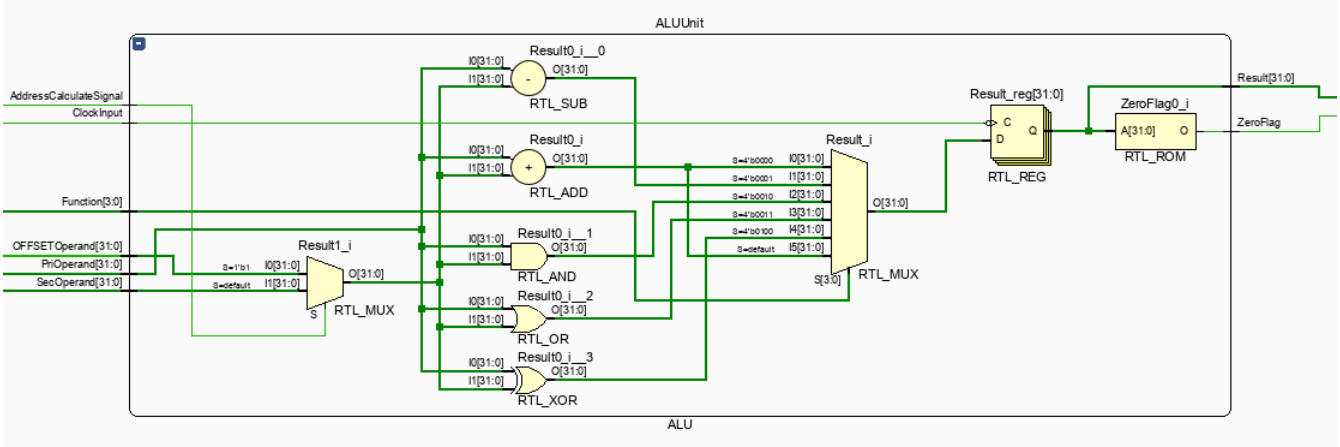


FIGURE 4.17 The simple datapath with the control unit. The input to the control unit is the 6-bit opcode field from the instruction.

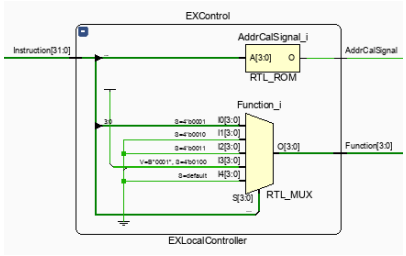
The idea of local controller / decentralized controlling is inspired from the above reference, this way the designer no need to know the complete control strategy before starting to actually build. But later in the LC any new/modification in control can be incorporated wisely.

Arithmetic Logic Unit module



The simple ALU unit, which does the computation. Note, in the project we required only few operations and only those are implemented. Later at the custom requirement raised new operation can be added. For now the operations available are simple – 32-bit unsigned addition, subtraction, bit-wise AND, OR, XOR. Also, the condition codes or result flags are very limited only to Zero flag. As this is used by BEQ operation for finding if two operands are same by subtraction.

Local Control module

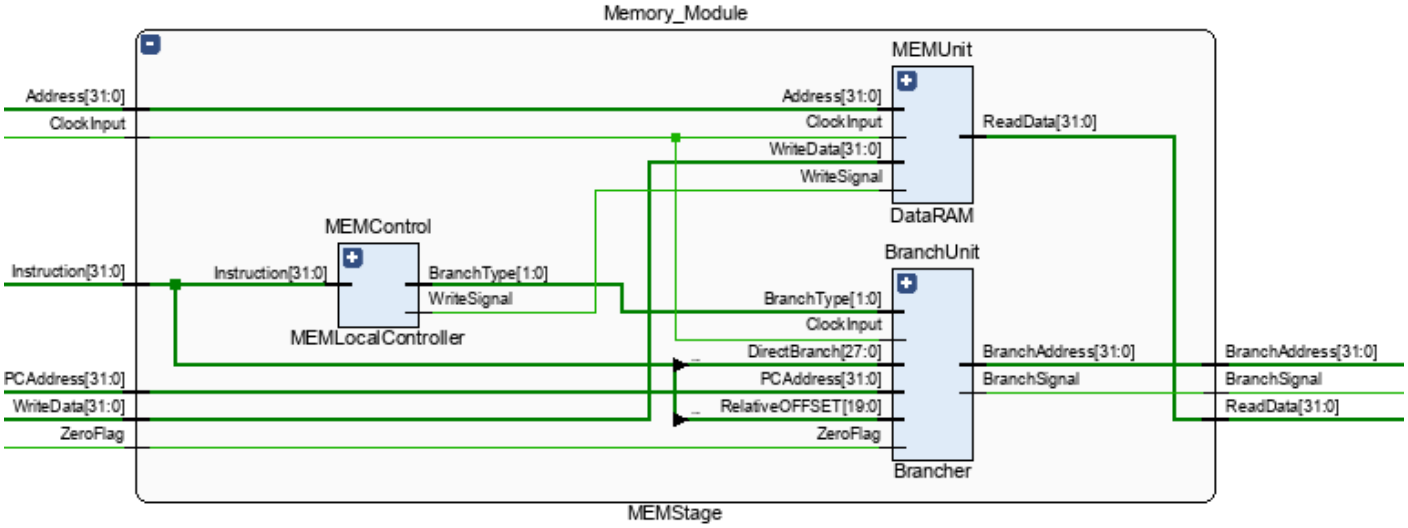


		EX	
Instruction	**Operation**	**AddressCalculateSignal**	**ALU Function**
R-type		0=SecOprd, 1=address	0= +, -, &, |, ^
ADD Rs,Rt,Rd	Rd<-Rs+Rt	0	0
SUB Rs,Rt,Rd	Rd<-Rs-Rt	0	1
AND Rs,Rt,Rd	Rd<-Rs&Rt	0	2
OR Rs,Rt,Rd	Rd<-Rs|Rt	0	3
XOR Rs,Rt,Rd	Rd<-Rs^Rt	0	4
I-type			
LW Rb,Rd,#OFFSET	Rd<-[Rb+#OFFSET]	1	0
SW Rb,Rs,#OFFSET	Rs->[Rb+#OFFSET]	1	0
J-type			
BEQ R1,R2,#PC-offset	if(R1==R2) GOTO PC+1+#OFFSET	0	1
JUMP #PROGLINE	GOTO #PROGLINE	0/X	0/X
NOP	Nothing	0/X	0/X

STAGE-4 Memory and Branch, Datapath and Control

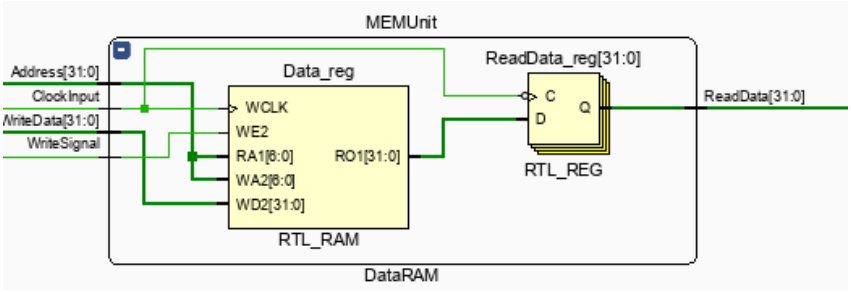
This Stage is responsible for two major things – Data memory reading/writing and Branching.

1. Responsible for reading from Data memory / writing into as 32-bit. Instructions like SW, LW have their work here.
2. Responsible for checking if branching to be taken, and calculating the branch address to where the next program flow has to go.



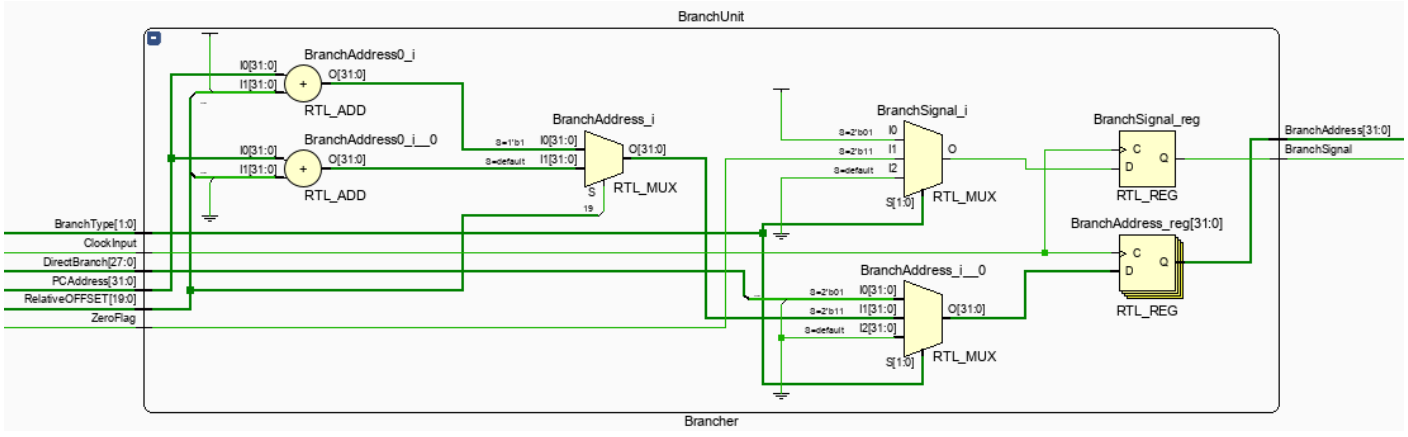
The Data memory could be an external if the address bus and data bus can be stretched outside the CPU unit. For now, in this project, we have allocated the memory within the FPGA (project space).

Data Memory module



This RAM memory could be extended to external memory segment, I/O peripherals. For now, in this project, we have allocated the memory within the FPGA (project space). It's a simple stack of 32-bit registers of 128 counts.

Brancher module



This module calculates the branch address – in case of BEQ the addressing is PC-relative, so the branch address is calculated as,

Branch address = PC + {#OFFSET sign extended to 32 bits}

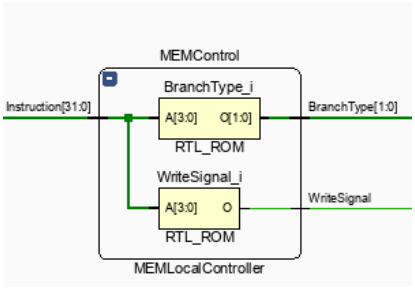
For OFFSET the instruction code have allocated 5 nibble / 20-bit. It can be programmed to branch in range of PC ± 2^19 instructions. The negative numbers (in case of backward branching is achieved by supplying 2's complement value as the offset). The addition is a 32-bit, so the 20-bit is sign extended (arithmetic) before adding.

In case of JUMP instruction is pseudo-direct addressing mode, and can directly branch the program to 0 – $(2^{28} - 1)$ instructions. The branch address is calculated as,

Branch address = {#OFFSET sign extended to 32 bits}

Note, the project doesn't have any dedicated branch predictor, so an Always no-branching decision is taken and next instructions are fed into the pipeline. Further control hazard detection and prevention shall be discussed in later section.

Local Control module



The MEM local control is responsible for generating control signals i.e., Branch Type and Memory write signal.

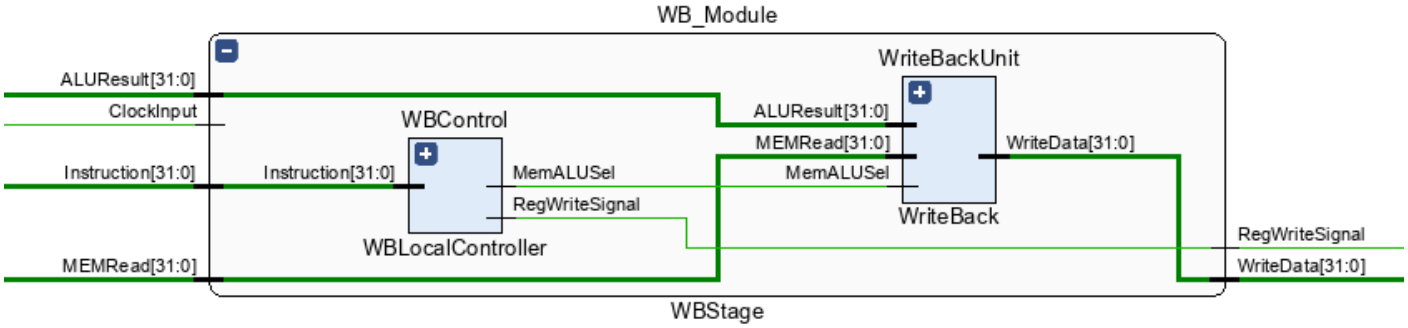
		MEM	
Instruction	**Operation**	**BranchType**	**MEM WriteSignal**
R-type		0=none 1=Direct 3=Conditional/relative	1=write, 0=none
ADD Rs,Rt,Rd	Rd<-Rs+Rt	0	0
SUB Rs,Rt,Rd	Rd<-Rs-Rt	0	0
AND Rs,Rt,Rd	Rd<-Rs&Rt	0	0
OR Rs,Rt,Rd	Rd<-Rs|Rt	0	0
XOR Rs,Rt,Rd	Rd<-Rs^Rt	0	0
I-type			
LW Rb,Rd,#OFFSET	Rd<-[Rb+#OFFSET]	0	0
SW Rb,Rs,#OFFSET	Rs->[Rb+#OFFSET]	0	1
J-type			
BEQ R1,R2,#PC-offset	if(R1==R2) GOTO PC+1+#OFFSET	3	0
JUMP #PROGLINE	GOTO #PROGLINE	1	0
NOP	Nothing	0	0

STAGE-5 Write Back

This stage is the last stage where the calculated/result is stored in the register. The stage is responsible for –

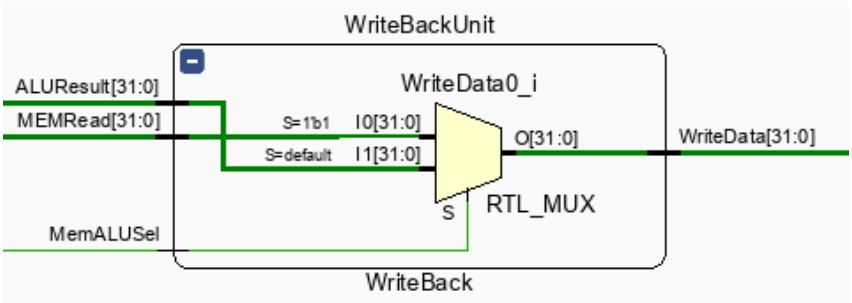
1. Writing the result into the destination register by accessing Register Bank at ID stage.
2. The result is either ALU result or Memory read (in case of LW instruction). The WB local control will steer the selection mux to either ALU or Memory.

The overview of write back is as below-

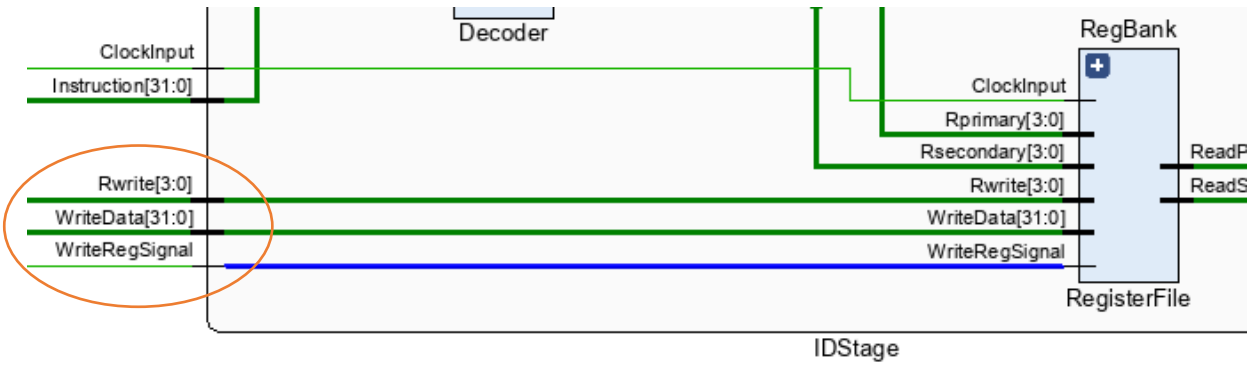


Some of the encompassed sub-modules are elaborated below.

Write-back module

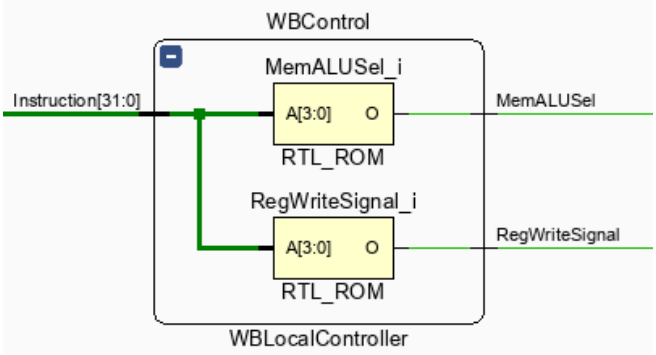


The dathpath is very simple Multiplexer which either selects the ALU or Memory depending on the local control signal. Note the WB stage isn't complete with this, the datapath extends till the Register Bank at ID stage. Since the Register bank is at ID stage, the Write data and destination register is sent all the way to Register Bank at ID. As shown in the below diagram.



A fact to be appreciated here is – Always be it register bank, memory, the write operation is performed prior to the read operation. As this way it ensures latest value is read every time.

Local control module



		WB	
Instruction	**Operation**	**MemALUSel**	**RegisterWriteSignal**
R-type		0=ALU, 1=MEM	
ADD Rs,Rt,Rd	Rd<-Rs+Rt	0	1
SUB Rs,Rt,Rd	Rd<-Rs-Rt	0	1
AND Rs,Rt,Rd	Rd<-Rs&Rt	0	1
OR Rs,Rt,Rd	Rd<-Rs|Rt	0	1
XOR Rs,Rt,Rd	Rd<-Rs^Rt	0	1
I-type			
LW Rb,Rd,#OFFSET	Rd<-[Rb+#OFFSET]	1	1
SW Rb,Rs,#OFFSET	Rs->[Rb+#OFFSET]	0	0
J-type			
BEQ R1,R2,#PC-offset	if(R1==R2) GOTO PC+1+#OFFSET	0/X	0
JUMP #PROGLINE	GOTO #PROGLINE	0/X	0
NOP	Nothing	0/X	0

Overall control signal

This section illustrates the overall control signal at each stage and the controlling technique implemented in this project.

		Control Signal list						Various stages and its operation				
		EX		MEM		WB						
Instruction	**Operation**	**AddressCalculateSignal**	**ALU Function**	**BranchType**	**MEM WriteSignal**	**MemALUSel**	**RegisterWriteSignal**	**STAGE -1**	**STAGE -2**	**STAGE -3**	**STAGE -4**	**STAGE -5**
R-type		0=SecOprd, 1=address	0= +, -, &, |, ^	0=none 1=Direct 3=Conditional/relative	1=write, 0=none	0=ALU, 1=MEM		IF	ID	EX	MEM	WB
ADD Rs,Rt,Rd	Rd<-Rs+Rt	0	0	0	0	0	1	Instruction fetch, Increment PC	Read registers - Rs, Rt	Rs OP Rd	None	Write Rd
SUB Rs,Rt,Rd	Rd<-Rs-Rt	0	1	0	0	0	1					
AND Rs,Rt,Rd	Rd<-Rs&Rt	0	2	0	0	0	1					
OR Rs,Rt,Rd	Rd<-Rs|Rt	0	3	0	0	0	1					
XOR Rs,Rt,Rd	Rd<-Rs^Rt	0	4	0	0	0	1					
I-type												
LW Rb,Rd,#OFFSET	Rd<-[Rb+#OFFSET]	1	0	0	0	1	1	Instruction fetch, Increment PC	Read register Rb	Rb + #OFFSET	Read	Write Rd
SW Rb,Rs,#OFFSET	Rs->[Rb+#OFFSET]	1	0	0	1	0	0				Write	None
J-type												
BEQ R1,R2,#PC-offset	if(R1==R2) GOTO PC+1+#OFFSET	0	1	3	0	0/X	0	Instruction fetch, Increment PC	Read register R1, R2	R1 - R2	PC+#OFFSET Jump if EQ	None
JUMP #PROGLINE	GOTO #PROGLINE	0/X	0/X	1	0	0/X	0	Instruction fetch, Increment PC	None	None	Jump	None
NOP	Nothing	0/X	0/X	0	0	0/X	0	Instruction fetch, Increment PC	None	None	None	None

Reference -

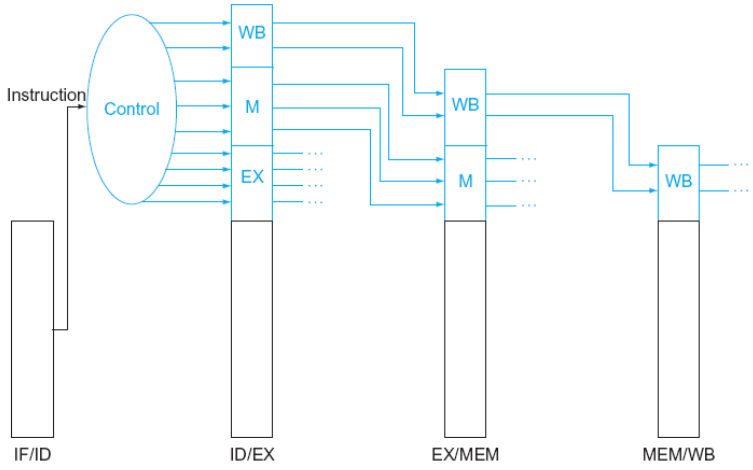
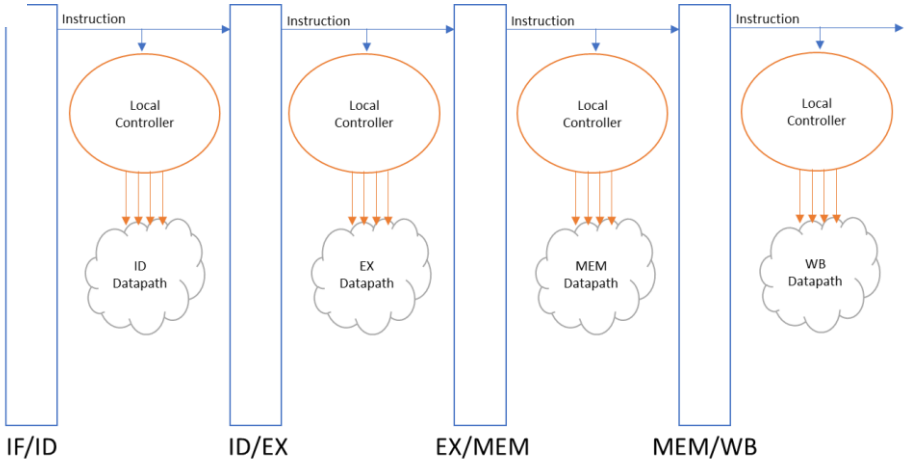


FIGURE 4.50 The control lines for the final three stages. Note that four of the nine control lines are used in the EX phase, with the remaining five control lines passed on to the EX/MEM pipeline register extended to hold the control lines; three are used during the MEM stage, and the last two are passed to MEM/WB for use in the WB stage.

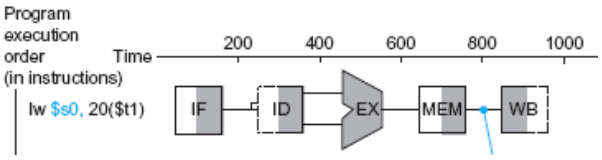


This diagram is the actual implementation in this project. The decentralised local controlling.

Pipeline-ing the stages

One of the Eight Great ideas in Computer Architecture: *Performance elevation via Pipelining*.

Pipelining helps us to que up the instructions and flow throw each stage as they progress to completion, there by efficient utilization of stage. As at every instance of time all the stages are occupied with instructions.



This idea differentiates RISC from CISC. In CISC the micro-codes will perform at any instant one of either Addressing > Execution > Housekeeping of the instruction, before yielding the result. There by every instruction latency and throughput is sum of all cycles utilized in all three stages.

Time -->	Cycle 0	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9	Cycle 10	Cycle 11	
Addressing	*	Instr -1	*	*	Instr -2	*	*	Instr -3	*	*	Instr -4	*	*
Execution	*	*	Instr -1	*	*	Instr -2	*	*	Instr -3	*	*	Instr -4	*
Housekeeping	*	*	*	Instr -1	*	*	Instr -2	*	*	Instr -3	*	*	Instr -4

In RISC, the stark difference is that the number of stages is fixed (say 5 as in this RISC-V project) and all instructions will pass each stage and partially reach to completion status in a que. This opens an opportunity to utilize un-utilized stages for next instruction processing. This que-ing is achieved by pipeline registers, which is a FIFO buffer. Therefore, RISC though the latency remains same for all instructions, the throughput is almost reduced to 1-instruction each cycle.

$$\text{Time} = \text{Seconds/Program} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

The CISC works in reducing no. of instructions in a program. RISC has fewer instruction set (compared to CISC) but latency is way improved, therefore RISC works in reducing no. of cycles per instruction (aka increasing throughput). As illustrated below, every cycle will take new instruction inside, and spit out result of one instruction. At every instant (after the que is filled / 5cycles later) all the 5 stages are occupied with some instruction processing.

Time -->	Cycle 0	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9	Cycle 10	Cycle 11	
IF	*	IF-1	IF-2	IF-3	IF-4	IF-5	IF-6	IF-7	IF-8				
ID	*	*	ID-1	ID-2	ID-3	ID-4	ID-5	ID-6	ID-7	ID-8			
EX	*	*	*	EX-1	EX-2	EX-3	EX-4	EX-5	EX-6	EX-7	EX-8		
MEM	*	*	*	*	MEM-1	MEM-2	MEM-3	MEM-4	MEM-5	MEM-6	MEM-7	MEM-8	
WB	*	*	*	*	*	WB-1	WB-2	WB-3	WB-4	WB-5	WB-6	WB-7	WB-8

In RISC, each stage has a data-path and control, forming a Combinational logic (or state + combination within) both the input and output are from/to a state element i.e., Pipeline registers. Each pipeline register is a synchronous D-FF of 64bit/128bit size. It has features like Chip-enable (used to stall the next input/output by setting it to false), Reset (used to purge/bubble a stage). These are used to address hazards prevention techniques.

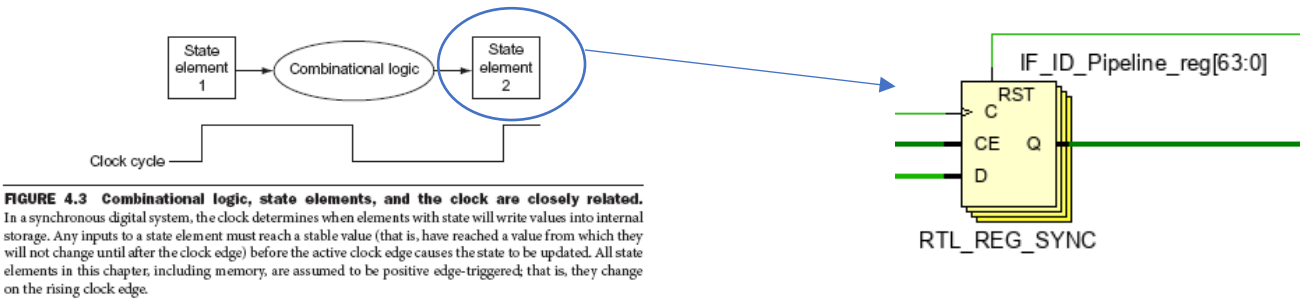
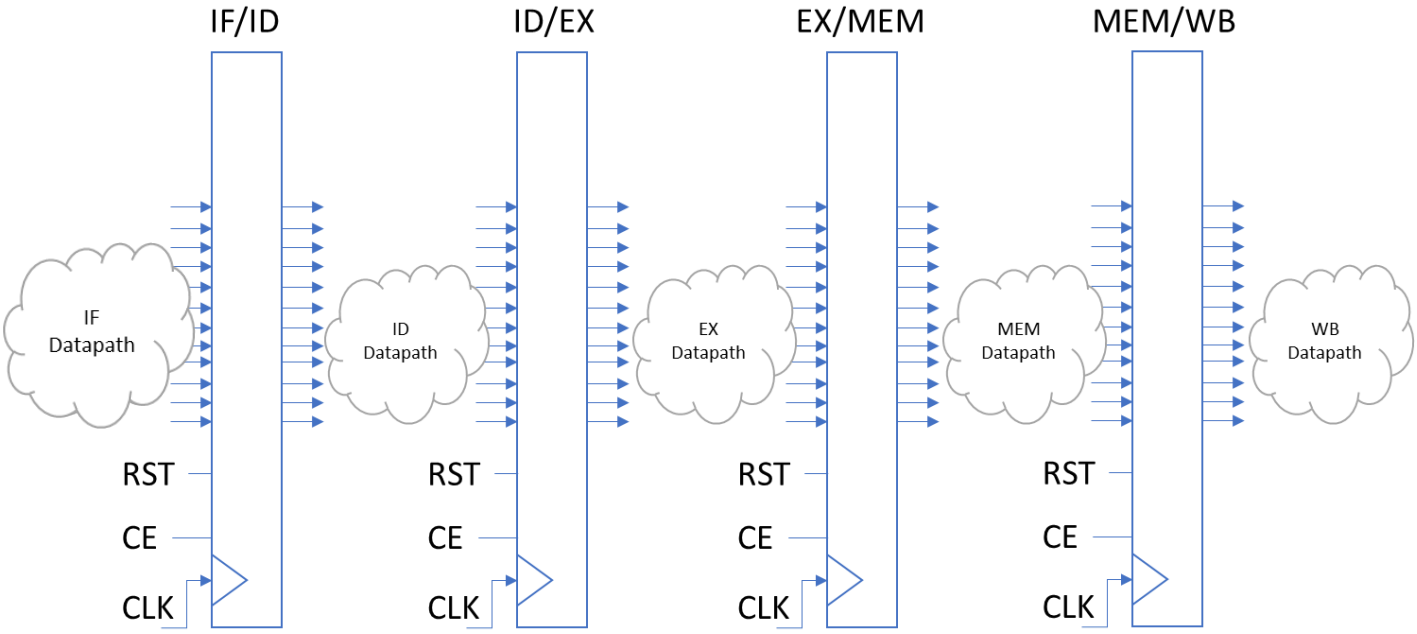


FIGURE 4.3 Combinational logic, state elements, and the clock are closely related. In a synchronous digital system, the clock determines when elements with state will write values into internal storage. Any inputs to a state element must reach a stable value (that is, have reached a value from which they will not change until after the clock edge) before the active clock edge causes the state to be updated. All state elements in this chapter, including memory, are assumed to be positive edge-triggered; that is, they change on the rising clock edge.

Pipeline register Mapping

The pipeline registers are bit mapped to each output values a stage wanted to supply to next stage.



At each stage the output is packed and sourced to pipeline register. Later in the next stage the same is unpacked and used to perform further operation, this takes place sequentially until its complete/finishes WB stage.

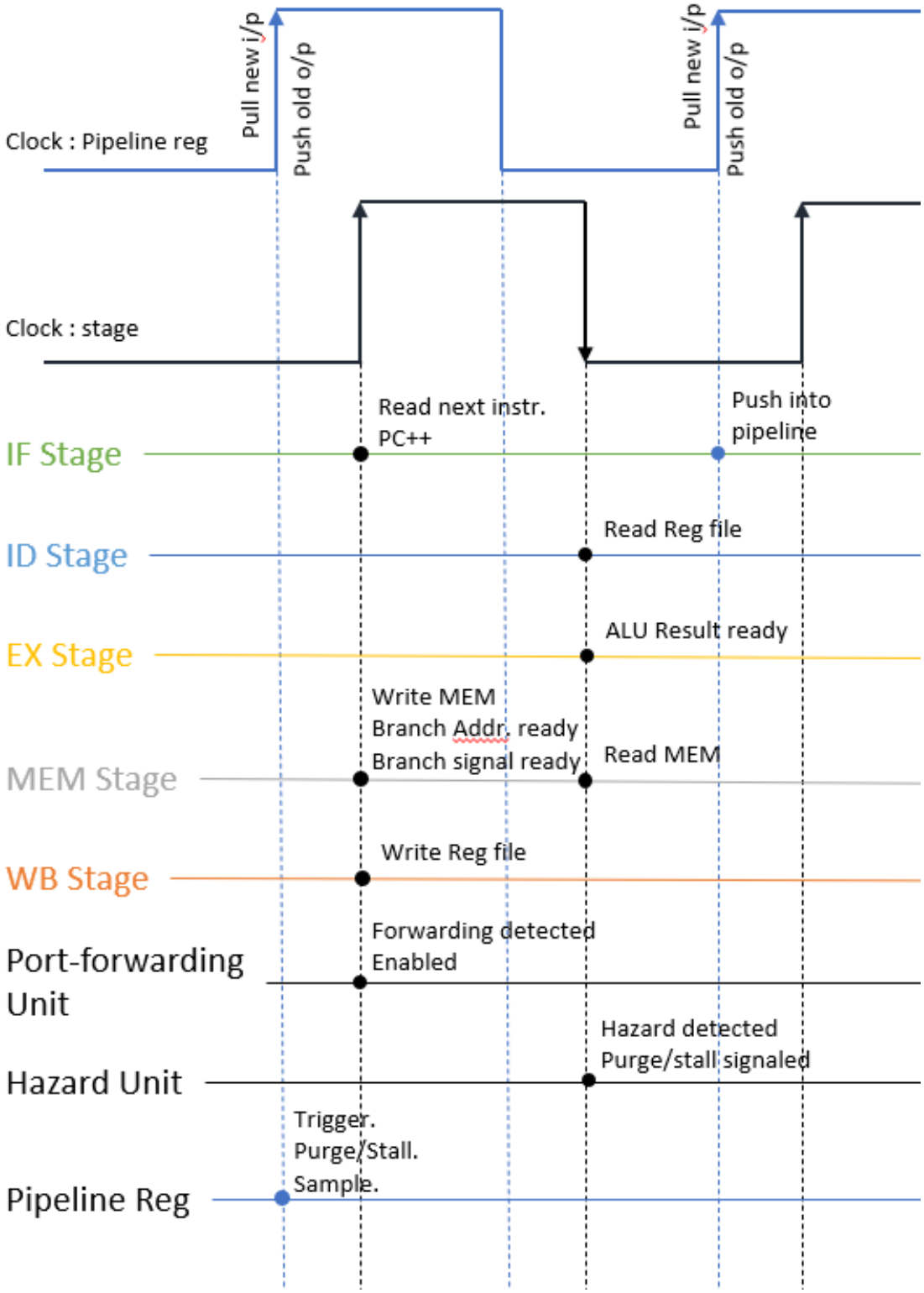
In this project the mapping is as follows -

Pipeline register Mapping

Index`	IF/ID	Index`	ID/EX	Index`	EX/MEM	Index`	MEM/WB
31:0	Instruction	31:0	Instruction	31:0	Instruction	31:0	Instruction
63:32	ProgramCounter	63:32	ProgramCounter	63:32	ProgramCounter	35:32	Rdestination
		67:64	Rdestination	67:64	Rdestination	67:36	Result
		99:68	ReadSecondary	99:68	ReadSecondary	99:68	MemReadData
		131:100	ReadPrimary	131:100	Result		
				132	ZeroFlag		
Total:	**64**		**132**		**133**		**100**

Clocking and Timing

In this project we are using 2 phase clocks, one used for pipeline register triggering, other used by datapath of each stage. Some stages operate sequentially i.e., ID stage – The Write operation (from WB is performed first then Read operation. Various clock trigger and task completion is marked in the below diagram.



Integration with pipeline and testing the design (without Hazard Detection)

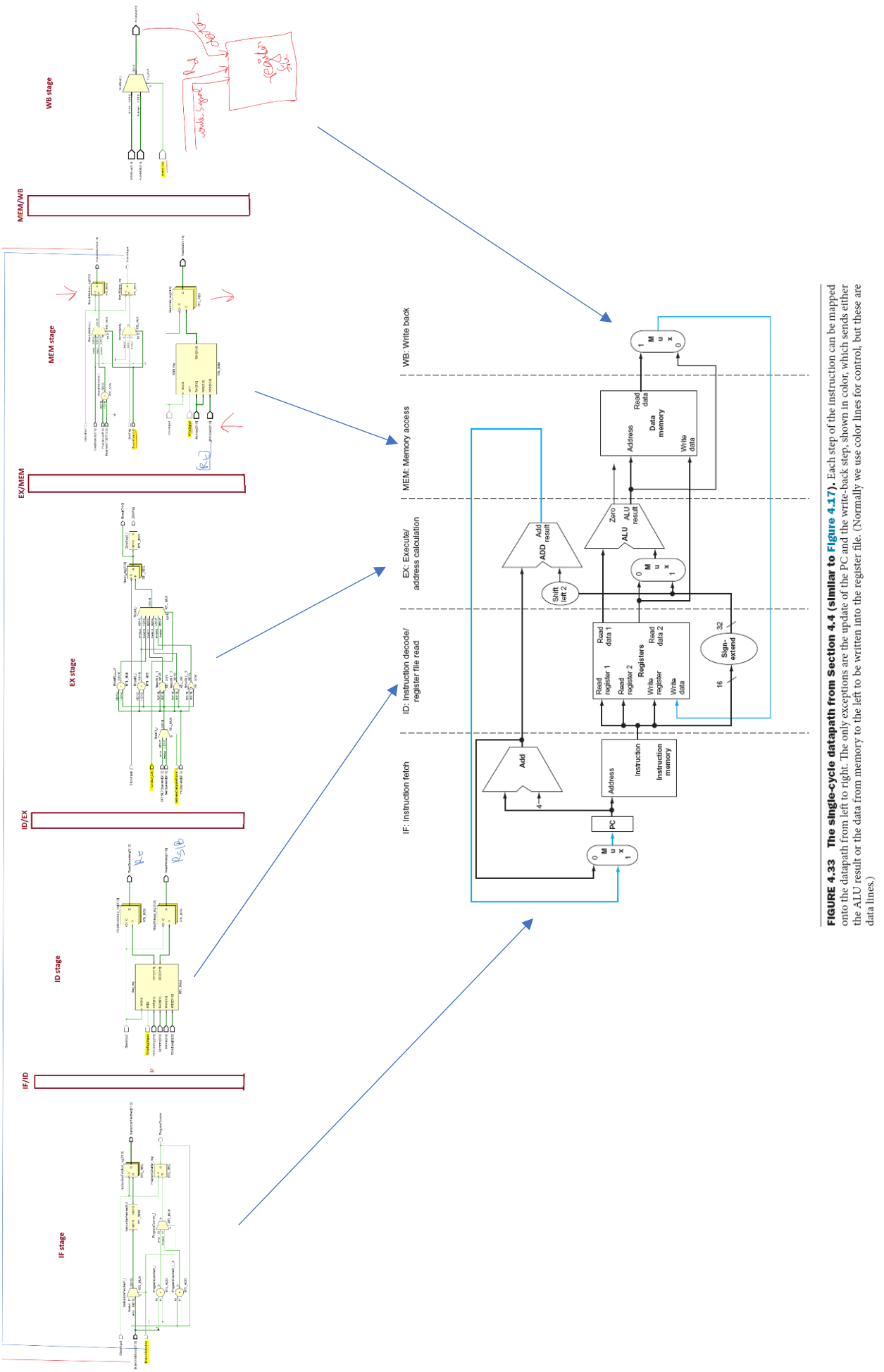


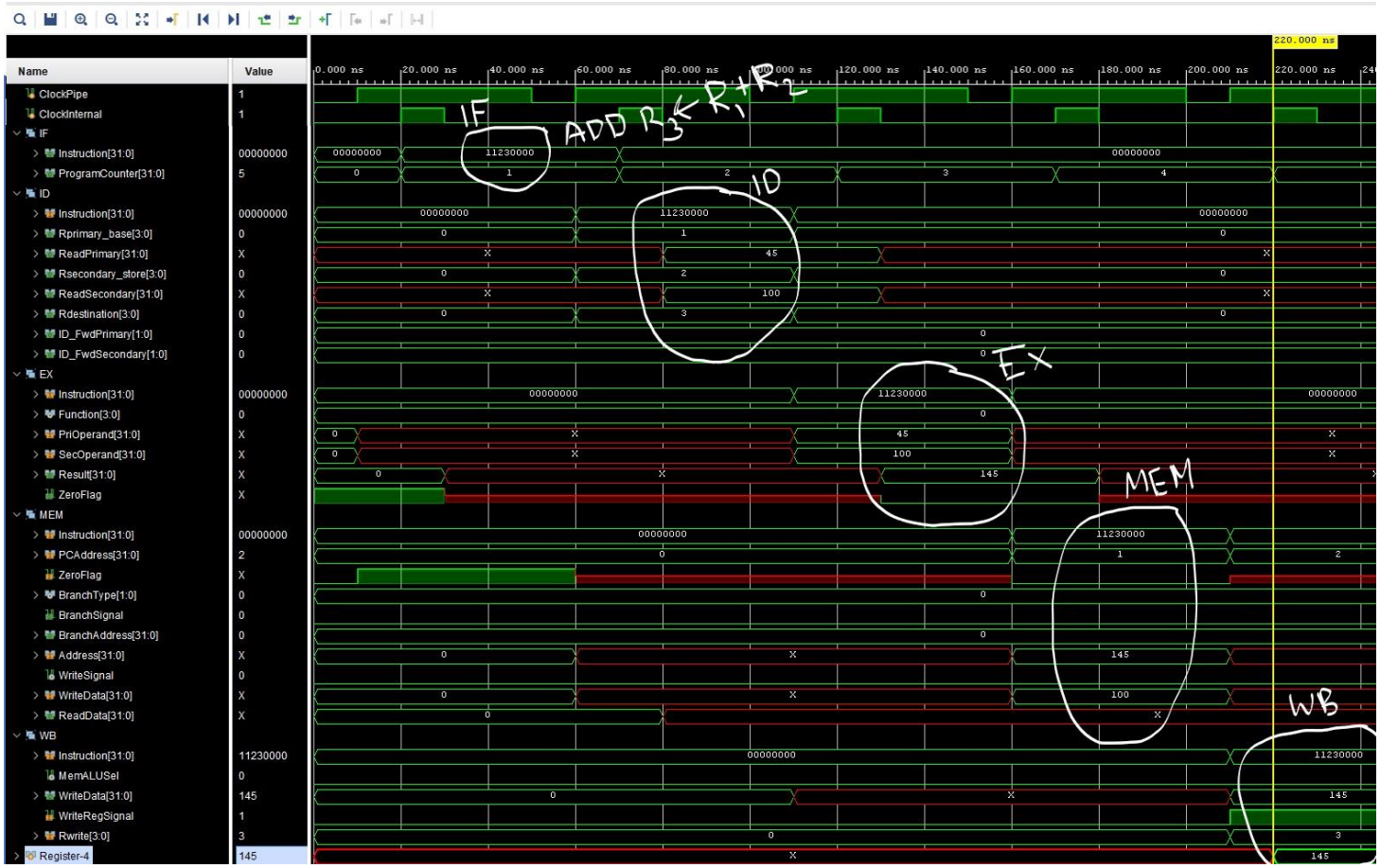
FIGURE 4.33 The single-cycle datapath from Section 4.4 (similar to Figure 4.17). Each step of the instruction can be mapped onto the datapath from left to right. The only exceptions are the update of the PC and the write-back step, shown in color, which sends either the ALU result or the data from memory to the left to be written into the register file. (Normally we use color lines for control, but these are data lines.)

Testing individual instructions

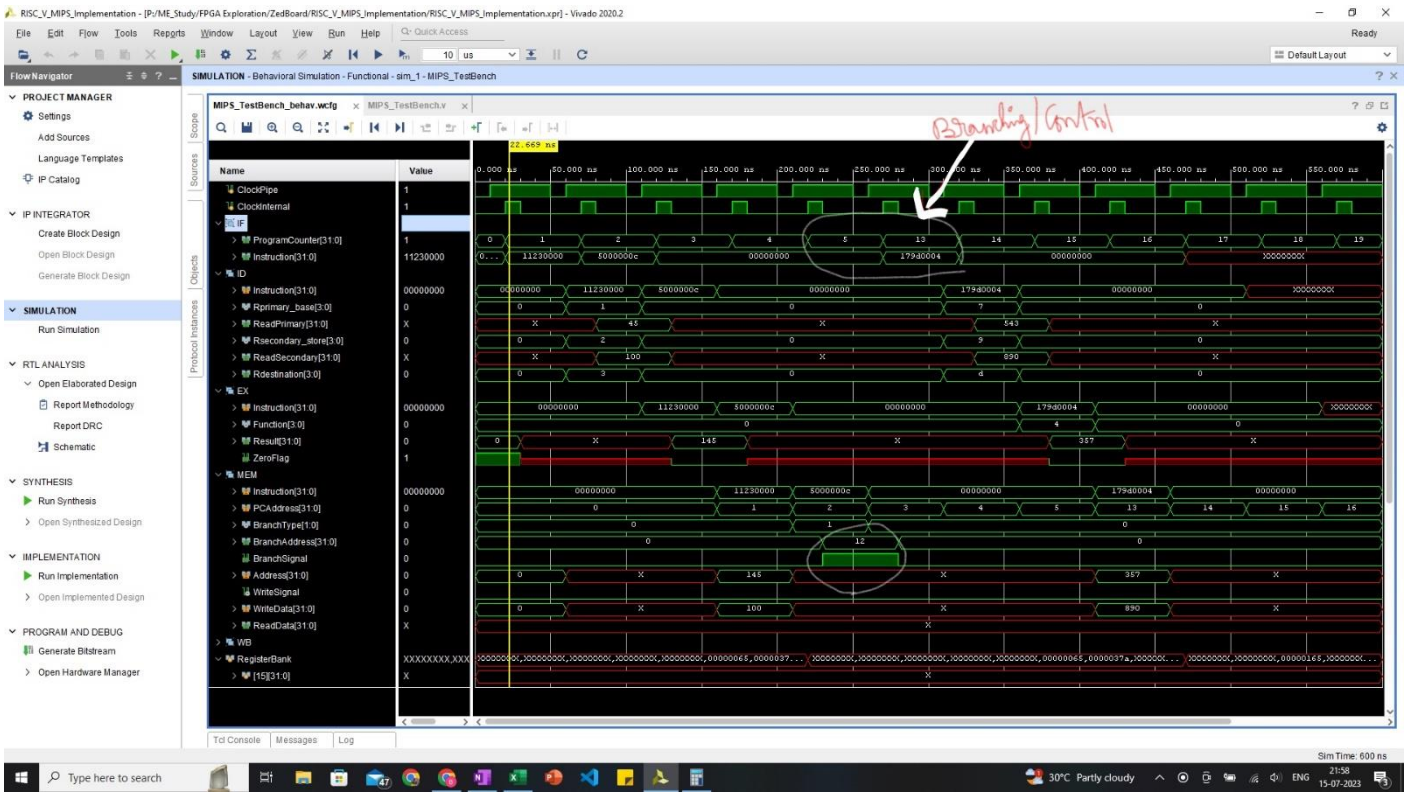
After integration (MIPS_Complete testbench) each instruction and its operation is verified. The test program is stored in ROM memory (at IF stage, Program memory) and simulation is verified. Below is the test slip.

R-type	Address	Instruction	Opcode	Result after WB	Test
ADD Rs,Rt,Rd	0	ADD R1,R2,R3	32'h11230000	R3=145	PASS
SUB Rs,Rt,Rd					
AND Rs,Rt,Rd					
OR Rs,Rt,Rd					
XOR Rs,Rt,Rd	0	XOR R7,R9,R13	32'h179D0004	R13=357	PASS
I-type					
LW Rb,Rd,#OFFSET	1	LW R1,R4,#55	32'h21400037	R4=543	PASS
SW Rb,Rs,#OFFSET	0	SW R1,R7,#55	32'h31700037		PASS
JUMP #PROGLINE	0	JUMP #12	32'h5000000C		PASS
NOP					
J-type	0	ADD R1,R2,R3	32'h11230000	ProgramMEM[0]=32'h11230000;	PASS
BEQ R1,R2,#PC-offset	1	BEQ R2,R10,#10	32'h42A0000A	ProgramMEM[1]=32'h42A0000A;	
	2	NOP	0	ProgramMEM[2]=0;	
	3	NOP	0	ProgramMEM[3]=0;	
Initialise the Reg Bank	4	NOP	0	ProgramMEM[4]=0;	
R1=45;	5	NOP	0	ProgramMEM[5]=0;	
R2=100	6	NOP	0	ProgramMEM[6]=0;	
R7=543	7	NOP	0	ProgramMEM[7]=0;	
R9=890	8	NOP	0	ProgramMEM[8]=0;	
R10=100	9	NOP	0	ProgramMEM[9]=0;	
	10	NOP	0	ProgramMEM[10]=0;	
	11	NOP	0	ProgramMEM[11]=0;	
Data memory	12	XOR R7,R9,R13	32'h179D0004	ProgramMEM[12]=32'h179D0004;	
mem[100]=543;	13	NOP	0	ProgramMEM[13]=0;	
	14	NOP	0	ProgramMEM[14]=0;	
	15	NOP	0	ProgramMEM[15]=0;	

Below is the simulation result of ADD R1,R2,R3 operation – You can see the instruction traversing in the pipeline every cycle till it reaches WB stage and writes the result value (R3=R1+R2; // 145).



Below is the screenshot of testing the branching instruction- the branch signal and the PC change can be seen.



Hazard Detection and Prevention Unit

In RISC, at each cycle next/new instructions are taken into processing, there may be scenarios were -

1. The new instruction might be needing to read a register content which is in process of getting updated by earlier instructions in pipeline. Thus, if not addressed, it would read un-updated register value leading to wrong result.
2. The older instruction tells to change the program flow (branching). If not addressed, the un-branched next-instructions will also be processed undesirably.

The above scenarios are called as Hazards. And they are of broadly classified as – Data Hazard and Control Hazard. The 1st scenario is Data hazard and 2nd is Control hazard.

Some scenarios and prevention technique -

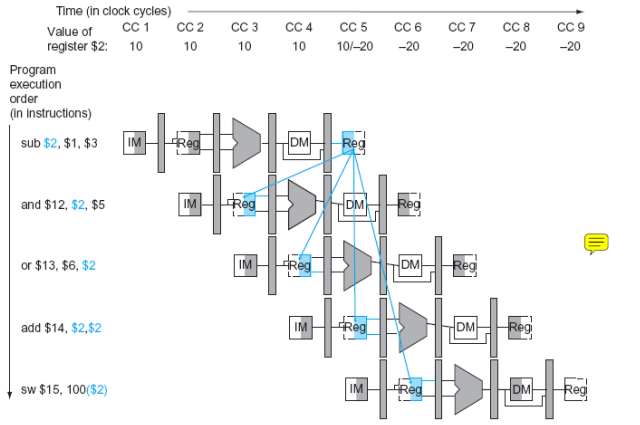


FIGURE 4.52 Pipelined dependences in a five-instruction sequence using simplified datapaths to show the dependences. All the dependent actions are shown in color, and "CC 1" at the top of the figure means clock cycle 1. The first instruction writes into $2, and all the following instructions read $2. This register is written in clock cycle 5, so the proper value is unavailable before clock cycle 5. (A read of a register during a clock cycle returns the value written at the end of the first half of the cycle, when such a write occurs.) The colored lines from the top datapath to the lower ones show the dependences. Those that must go backward in time are *pipeline data hazards*.

A data hazard scenario where RAW dependency following LOAD (LW) i.e.,

1. LW R0, R3, #100; //R3<-Mem[R0+100]
2. SUB R3, R4, R5;// R5<-R3-R4

In such case, the 1st instruction can't finish ID before 2nd instruction finishes MEM. So, to prevent such case pipeline is stalled (inserting a bubble/nop) for one cycle, thus next cycle the MEM result is forwarded to ID stage. Same is explained in the right diagram.

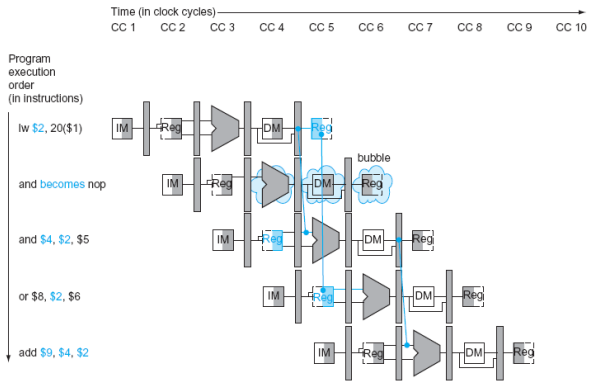


FIGURE 4.59 The way stalls are really inserted into the pipeline. A bubble is inserted beginning in clock cycle 4, by changing the and instruction to a nop. Note that the and instruction is really fetched and decoded in clock cycles 2 and 3, but its EX stage is delayed until clock cycle 5 (versus the unstalled position in clock cycle 4). Likewise the OR instruction is fetched in clock cycle 3, but its ID stage is delayed until clock cycle 5 (versus the unstalled clock cycle 4 position). After insertion of the bubble, all the dependences go forward in time and no further hazards occur.

A data hazard scenario where the current instruction requires information/result of previous instruction i.e., RAW (Read after write dependency)

1. ADD R1, R2, R3; //R3<-R1+R2
2. SUB R3, R4, R5; //R5<-R3-R4

In such case, the 1st instruction can't finish ID before 2nd instruction finishes EX. So, to prevent such case port forwarding technique is used. Same is explained in the left diagram.

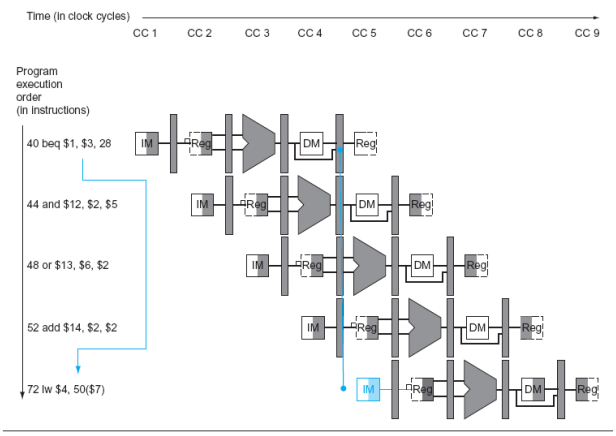


FIGURE 4.61 The impact of the pipeline on the branch instruction. The numbers to the left of the instruction (40, 44, ...) are the addresses of the instructions. Since the branch instruction decides whether to branch in the MEM stage—clock cycle 4 for the beq instruction above—the three sequential instructions that follow the branch will be fetched and begin execution. Without intervention, those three following instructions will begin execution before beq branches to lw at location 72. (Figure 4.31 assumed extra hardware to reduce the control hazard to one clock cycle; this figure uses the nonoptimized datapath.)

Control hazard scenario, the branch is taken in contrary to prediction. Then the enrolled three instructions has to be flushed out of the pipe before starting the program flow to new branch. This flushing is achieved by purging the pipe (or synchronously resetting the pipeline register). Same is explained in the left diagram.

In the coming section the detailed design of hazard detection and prevention implementation is explained.

RAW Detection and Prevention / Port forwarding

This data hazard can be detected at the ID stage, check if the source register or second operand register is same as the destination register of next stages with write register signal (i.e., R-type, LW). If so, then RAW is detected.

Logic –

```
//Check in EX unit as its latest
if(EX_OpCode==1 && EX_Rdestination==ID_RPrimary && (|ID_RPrimary)) ID_FwdPrimary<=FwdEX_ALUResult;
else begin //Check in MEM unit
    if(MEM_OpCode==1 && MEM_Rdestination==ID_RPrimary && (|ID_RPrimary))ID_FwdPrimary<=FwdMEM_ALUResult;
    else begin
        if(MEM_OpCode==2 && MEM_Rdestination==ID_RPrimary && (|ID_RPrimary))ID_FwdPrimary<=FwdMEM_MemoryRead;
        else ID_FwdPrimary<=0;
    end
end
```

Same logic is duplicated for Rsecondary (2nd operand) also. Note the R0 register is ignored (such as Rd!=0) as during nop/bubble the destination register is reset to R0 default.

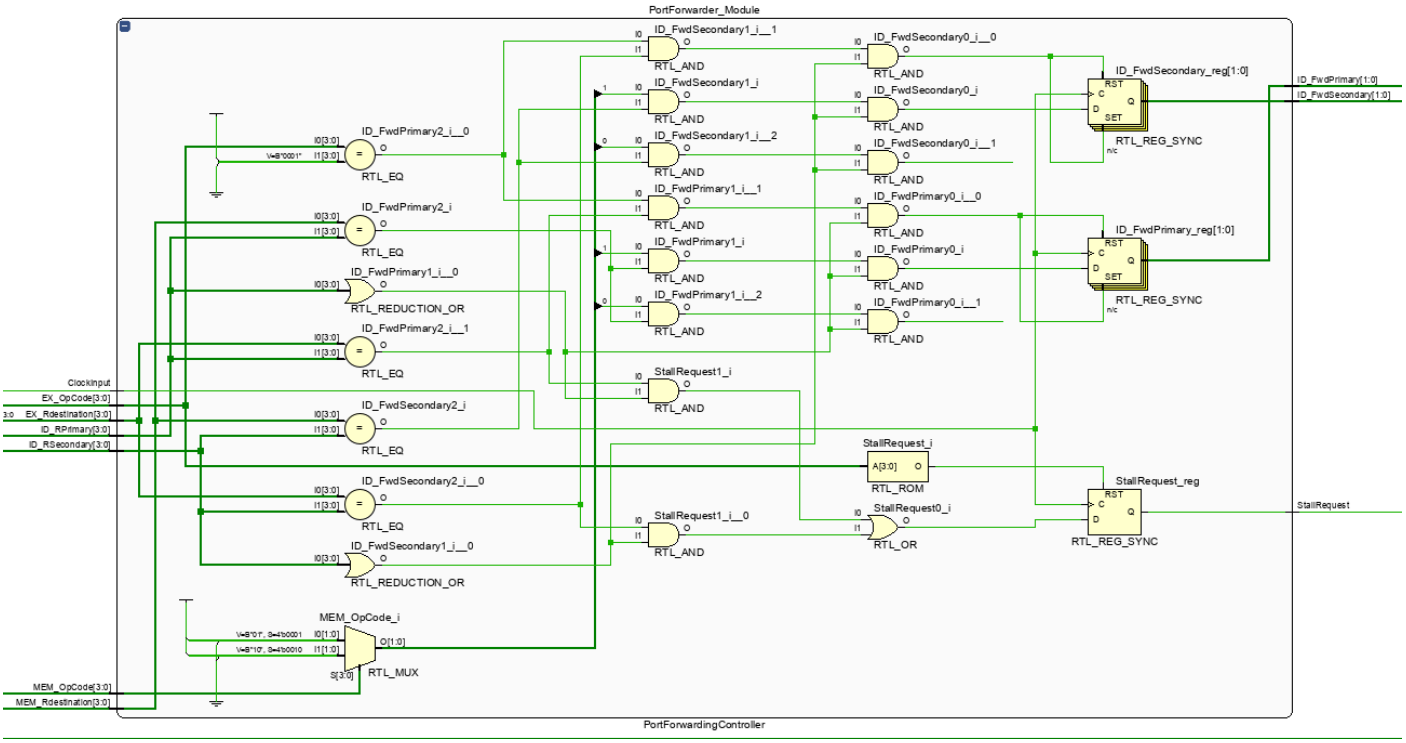
Reference –

```
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠  0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs)) ForwardA = 10

if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠  0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt)) ForwardB = 10
```

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠  0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠  0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01
```

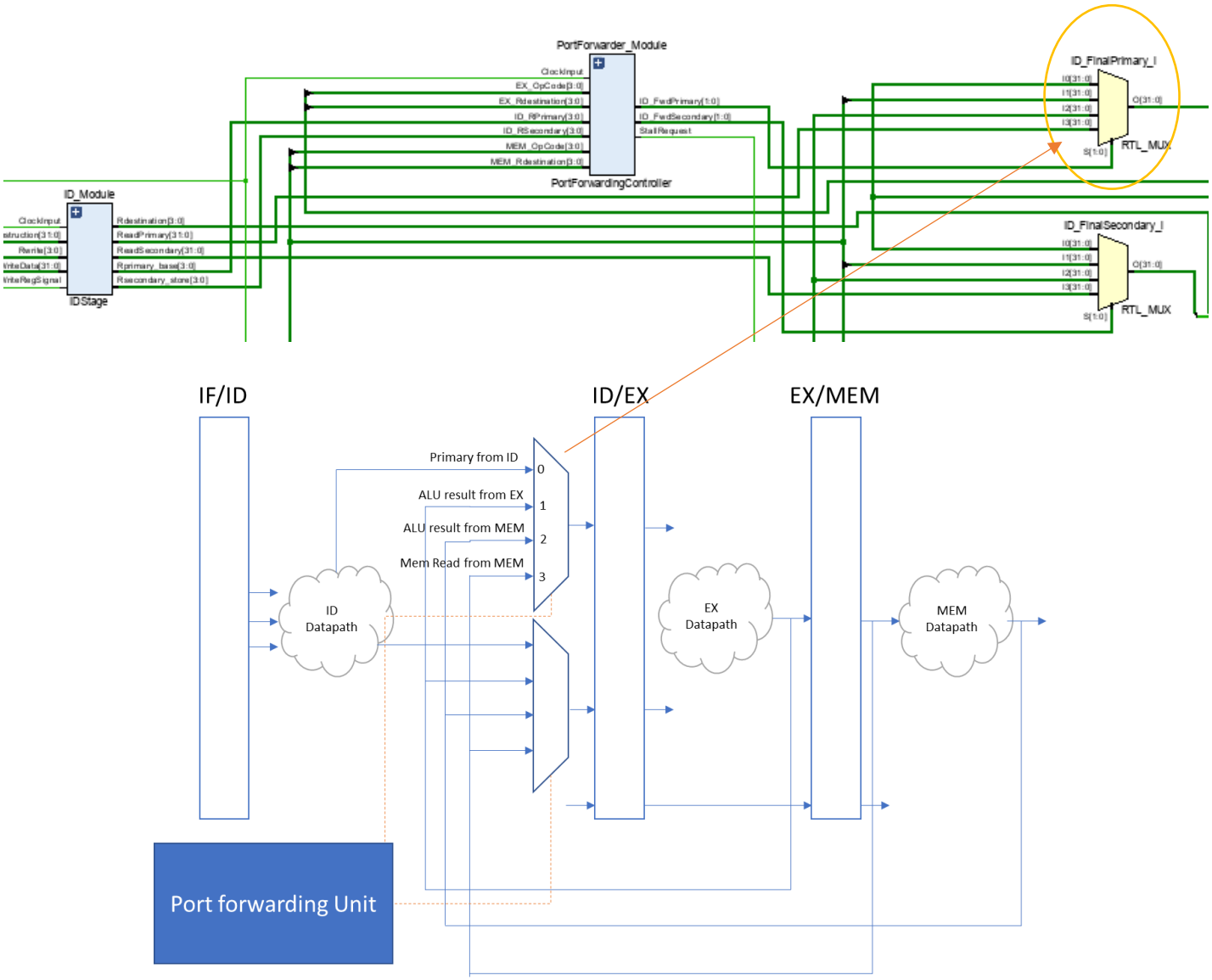


The above is the vivado RTL schematic for the port forwarding unit. This unit will only detect the data hazard i.e., RAW. And upon detection it will signal the forwarding MUX which is shown next page as a complete hazard control unit.

The LOAD following RAW detection is also embedded in this unit, and it signals the hazard addresser to stall the system for one cycle.

Logic –

```
if(EX_OpCode==2)begin//LW
    if(((EX_Rdestination==ID_RPrimary)&&(|ID_RPrimary)) || ((EX_Rdestination==ID_RSecondary)&&(|ID_RSecondary)))begin
        StallRequest<=1;
    end
    else StallRequest<=0;
end
else StallRequest<=0;
end
```



Reference –

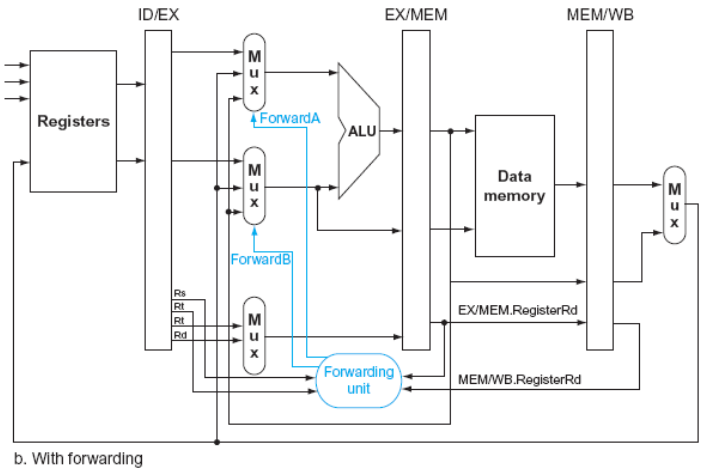
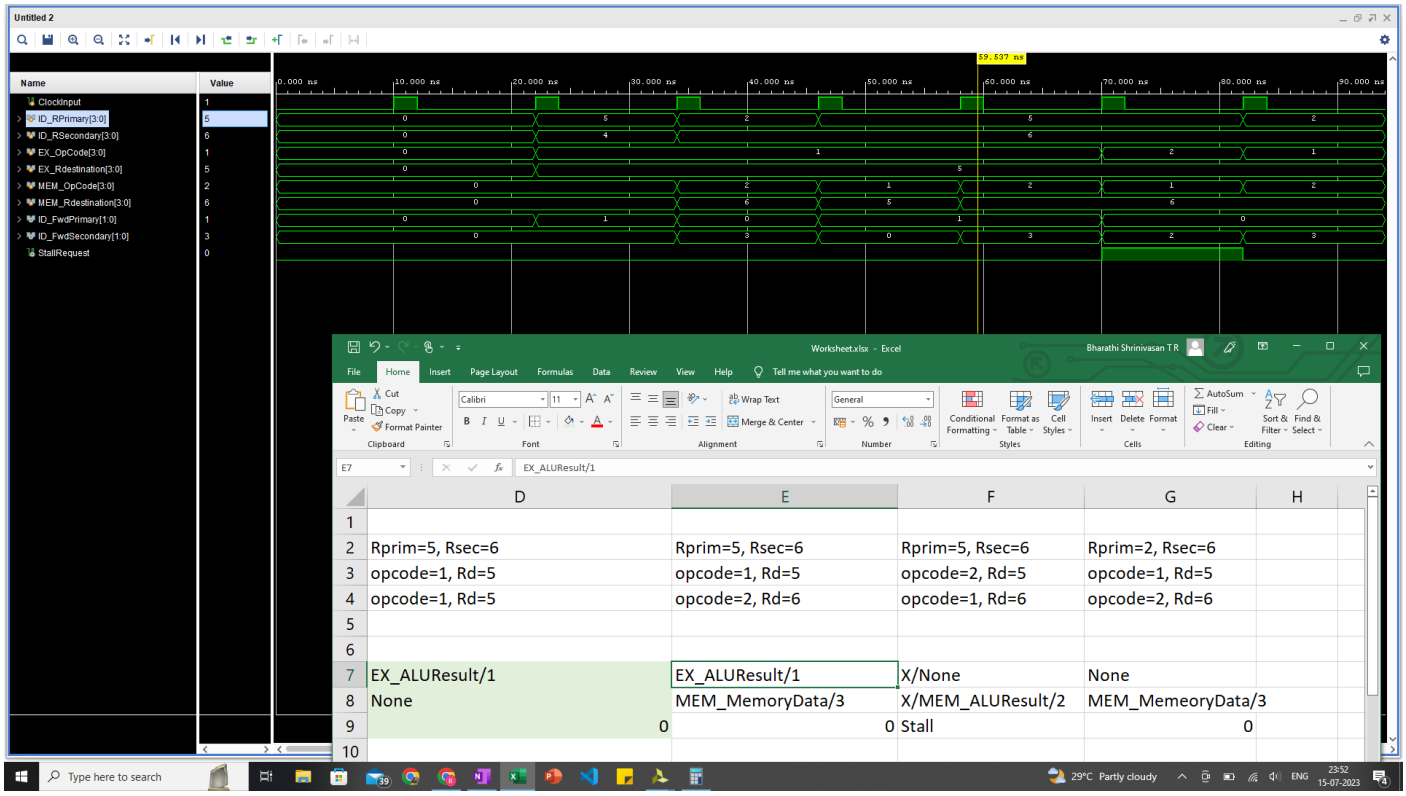


FIGURE 4.54 On the top are the ALU and pipeline registers before adding forwarding. On the bottom, the multiplexors have been expanded to add the forwarding paths, and we show the forwarding unit. The new hardware is shown in color. This figure is a stylized drawing, however, leaving out details from the full datapath such as the sign extension hardware. Note that the ID/EX.RegisterRt field is shown twice, once to connect to the Mux and once to the forwarding unit, but it is a single signal. As in the earlier discussion, this ignores forwarding of a store value to a store instruction. Also note that this mechanism works for `slt` instructions as well.

Note, unlike the refence, in the project the MUX is put in ID stage not in EX stage and ALU result forwarding from MEM stage is also included in the design.

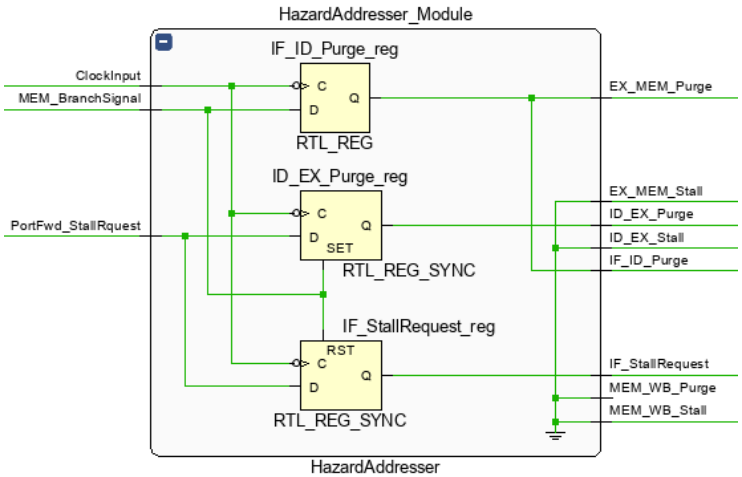
Testing Port forwarding module

Condition	1	2	3	4	5	6
IF	Rprim=5, Rsec=4	Rprim=2, Rsec=6	Rprim=5, Rsec=6	Rprim=5, Rsec=6	Rprim=5, Rsec=6	Rprim=2, Rsec=6
EX	opcode=1, Rd=5	opcode=1, Rd=5	opcode=1, Rd=5	opcode=1, Rd=5	opcode=2, Rd=5	opcode=1, Rd=5
MEM	opcode=0, Rd=0	opcode=2, Rd=6	opcode=1, Rd=5	opcode=2, Rd=6	opcode=1, Rd=6	opcode=2, Rd=6
Expected						
FWDpri	EX_ALUResult/1	None	EX_ALUResult/1	EX_ALUResult/1	X/None	None
FWDSec	None	MEM_MemeoryData/3	None	MEM_MemoryData/3	X/MEM_ALUResult/2	MEM_MemeoryData/3
StallReq	0	0	0	0	Stall	0
Status	PASS	PASS	PASS	PASS	PASS	PASS



Control Hazard Prevention / Purging

During branching, the decision of branch/not-to-branch is available at end of EX stage and eventually at MEM stage the branch signal is generated and branch address is calculated. Note, since we don't have any dedicated branch predictor unit, an Always no-branch prediction is taken and next instructions are fed into the pipeline. These 3 next instructions have to be bubbled in order to start



Condition	Action
Branching	Purge - IF/ID, ID/EX, EX/MEM
Load following read RAW	Purge - ID/EX Stall - IF/ID and IF unit
Other RAW	None. Port forwarding

Integrating and testing the design under hazard conditions

The final schematic diagram is in the appendix.

Reference -

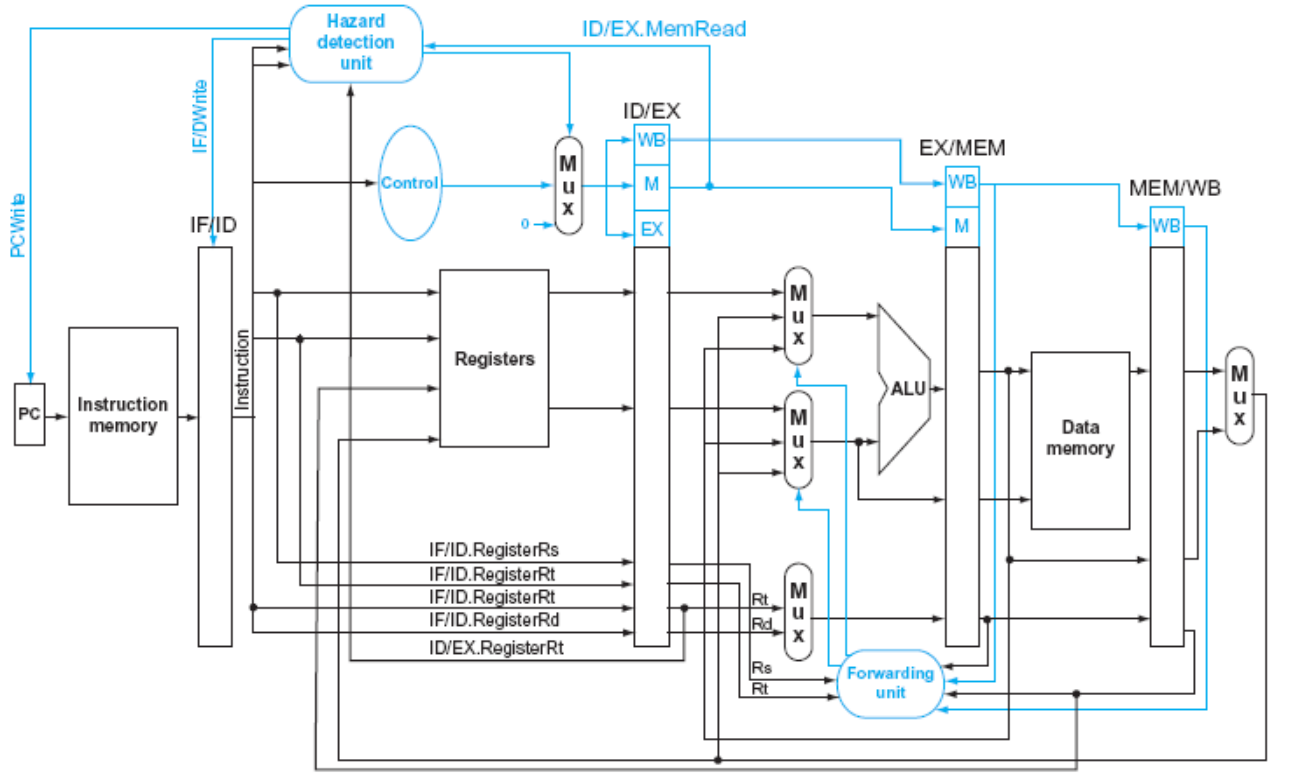


FIGURE 4.60 Pipelined control overview, showing the two multiplexors for forwarding, the hazard detection unit, and the forwarding unit. Although the ID and EX stages have been simplified—the sign-extended immediate and branch logic are missing—this drawing gives the essence of the forwarding hardware requirements.

The testing table and screenshots are in the appendix.

Fibonacci Series program run in this MIPS microprocessor

Final application program - Fibonacci Program								
PC	Address		Instruciton	Comment		Opcode		INIT
1	0	START:	LW R0,R1,#0	R1<-[R0+0] OR 0		32'h20100000	ProgramMEM[0]=32'h20100000;	R0=0
2	1		LW R0,R2,#1	R2<-[R0+1] OR 1		32'h20200001	ProgramMEM[1]=32'h20200001;	R4=0x1A6D
3	2	LOOP:	ADD R1,R2,R3			32'h11230000	ProgramMEM[2]=32'h11230000;	MEM[0]=0
4	3		BEQ R3,R4,START	-4		32'h434FFFFC	ProgramMEM[3]=32'h434FFFFC;	MEM[1]=1
5	4		ADD R2,R0,R1			32'h12010000	ProgramMEM[4]=32'h12010000;	
6	5		ADD R3,R0,R2			32'h13020000	ProgramMEM[5]=32'h13020000;	
7	6		JUMP LOOP			32'h50000002	ProgramMEM[6]=32'h50000002;	
8	7		NOP			32'h0	ProgramMEM[7]=32'h0;	
9	8		NOP			32'h0	ProgramMEM[8]=32'h0;	
10	9		NOP					

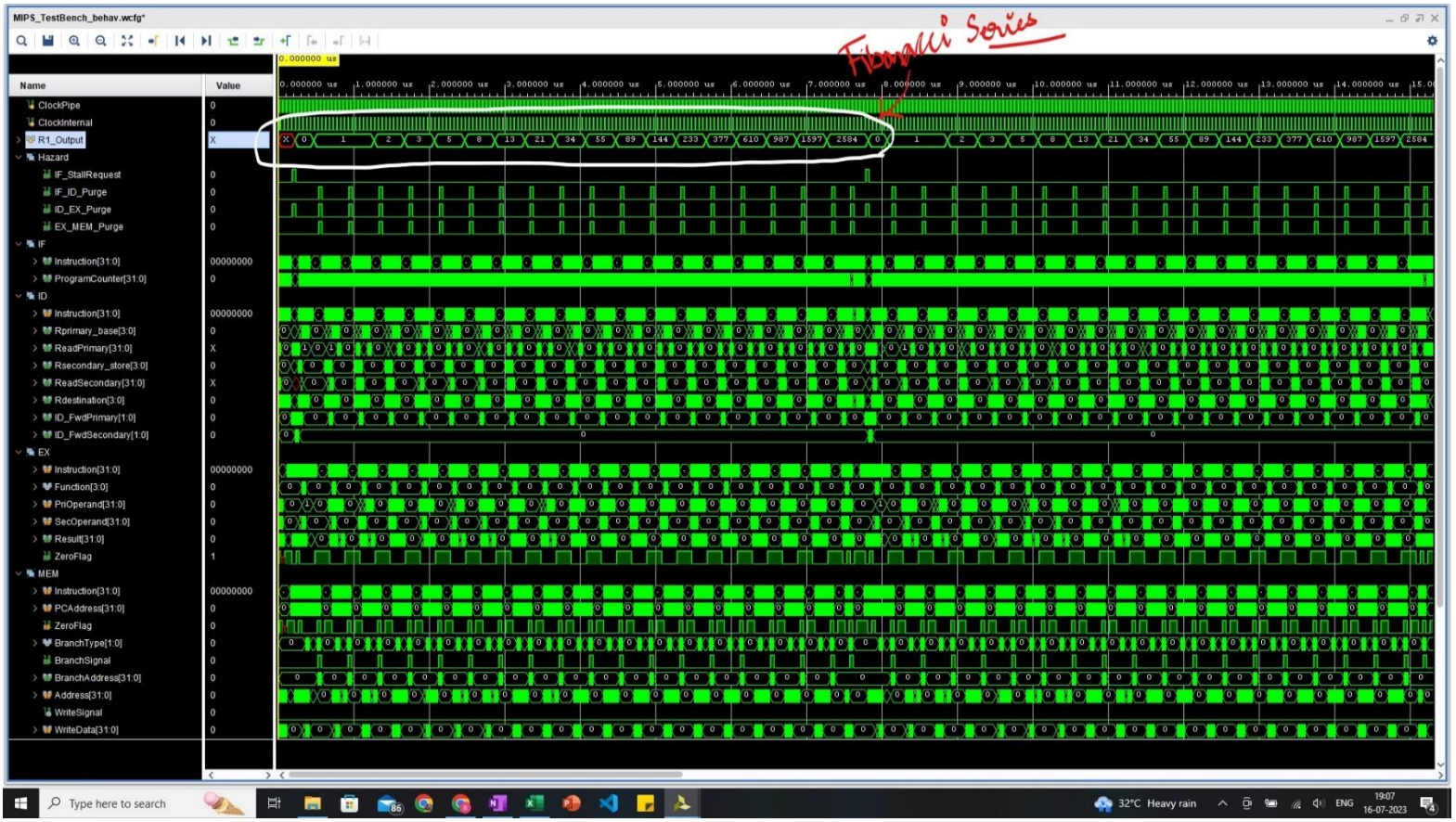
Project Summary | InstructionROM.v

P:/ME_Study/FPGA Exploration/ZedBoard/RISC_V_MIPS_Implemental

```
                    end
                end
            end

            initial begin // Write your Program here
                ProgramMEM[0]=32'h20100000;
                ProgramMEM[1]=32'h20200001;
                ProgramMEM[2]=32'h11230000;
                ProgramMEM[3]=32'h434FFFFC;
                ProgramMEM[4]=32'h12010000;
                ProgramMEM[5]=32'h13020000;
                ProgramMEM[6]=32'h50000002;
                ProgramMEM[7]=32'h0;
                ProgramMEM[8]=32'h0;


                ProgramMEM[8]=0;
                ProgramMEM[9]=0;
                ProgramMEM[10]=0;
                ProgramMEM[11]=0;
                ProgramMEM[12]=0;
                ProgramMEM[13]=0;
                ProgramMEM[14]=0;
                ProgramMEM[15]=0;
                ProgramMEM[16]=0;
                ProgramMEM[17]=0;
            end
        endmodule
```



Appendix –

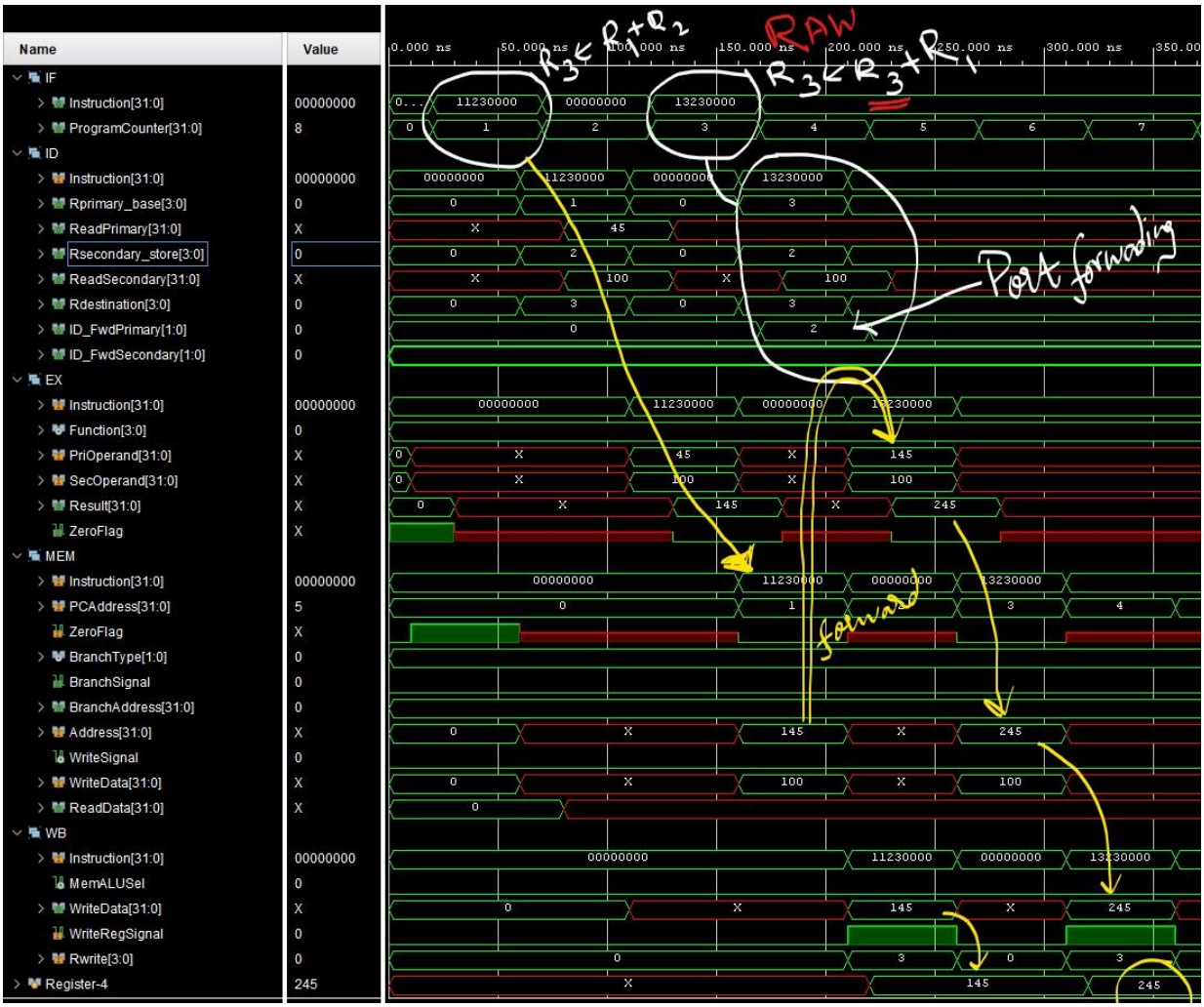
Hazard condition simulation and verification table -

init	Hazard		Test porting - FwdALUResult	PASS		
R1=45;						
R2=100		0	ADD R1,R2,R3	32'h11230000	ProgramMEM[0]=32'h11230000;	R3=145
R7=543	RAW	1	ADD R3,R2,R3	32'h13230000	ProgramMEM[1]=32'h13230000;	R3=245
R9=890		2	NOP	0	ProgramMEM[2]=0;	
R10=100		3	NOP	0	ProgramMEM[3]=0;	
		4	NOP	0	ProgramMEM[4]=0;	
		5	NOP	0	ProgramMEM[5]=0;	
		6	NOP	0	ProgramMEM[6]=0;	
		7	NOP	0	ProgramMEM[7]=0;	
mem[100]=543		8				
			Test porting - FwdMEM_AluResult	PASS		
		0	ADD R1,R2,R3	32'h11230000	ProgramMEM[0]=32'h11230000;	
		1	NOP	0	ProgramMEM[1]=0;	
	RAW	2	ADD R3,R2,R3	32'h13230000	ProgramMEM[2]=32'h13230000;	
		3	NOP	0	ProgramMEM[3]=0;	
		4	NOP	0	ProgramMEM[4]=0;	
		5	NOP	0	ProgramMEM[5]=0;	
		6	NOP	0	ProgramMEM[6]=0;	
		7	NOP	0	ProgramMEM[7]=0;	
			Test porting - FwdMEM_Read	PASS		
		0	SW R1,R7,#55	32'h31700037	ProgramMEM[0]=32'h31700037;	MEM[100]=543
		1	NOP	0	ProgramMEM[1]=0;	
		2	LW R1,R2,#55	32'h21200037	ProgramMEM[2]=32'h21200037;	R2=543
		3	NOP	0	ProgramMEM[3]=0;	
	RAW	4	ADD R1,R2,R3	32'h11230000	ProgramMEM[4]=32'h11230000;	R3=588
		5	NOP	0	ProgramMEM[5]=0;	
		6	NOP	0	ProgramMEM[6]=0;	
		7	NOP	0	ProgramMEM[7]=0;	
			Test hazard stall	PASS		
		0	SW R1,R7,#55	32'h31700037	ProgramMEM[0]=32'h31700037;	MEM[100]=543
		1	NOP	0	ProgramMEM[1]=0;	
		2	LW R1,R2,#55	32'h21200037	ProgramMEM[2]=32'h21200037;	R2=543
	RAW	3	ADD R1,R2,R3	32'h11230000	ProgramMEM[3]=32'h11230000;	R3=588
		4	XOR R1,R2,R3	32'h11230004	ProgramMEM[4]=32'h11230004;	R3=562
		5	NOP	0	ProgramMEM[5]=0;	
		6	NOP	0	ProgramMEM[6]=0;	
		7	NOP	0	ProgramMEM[7]=0;	

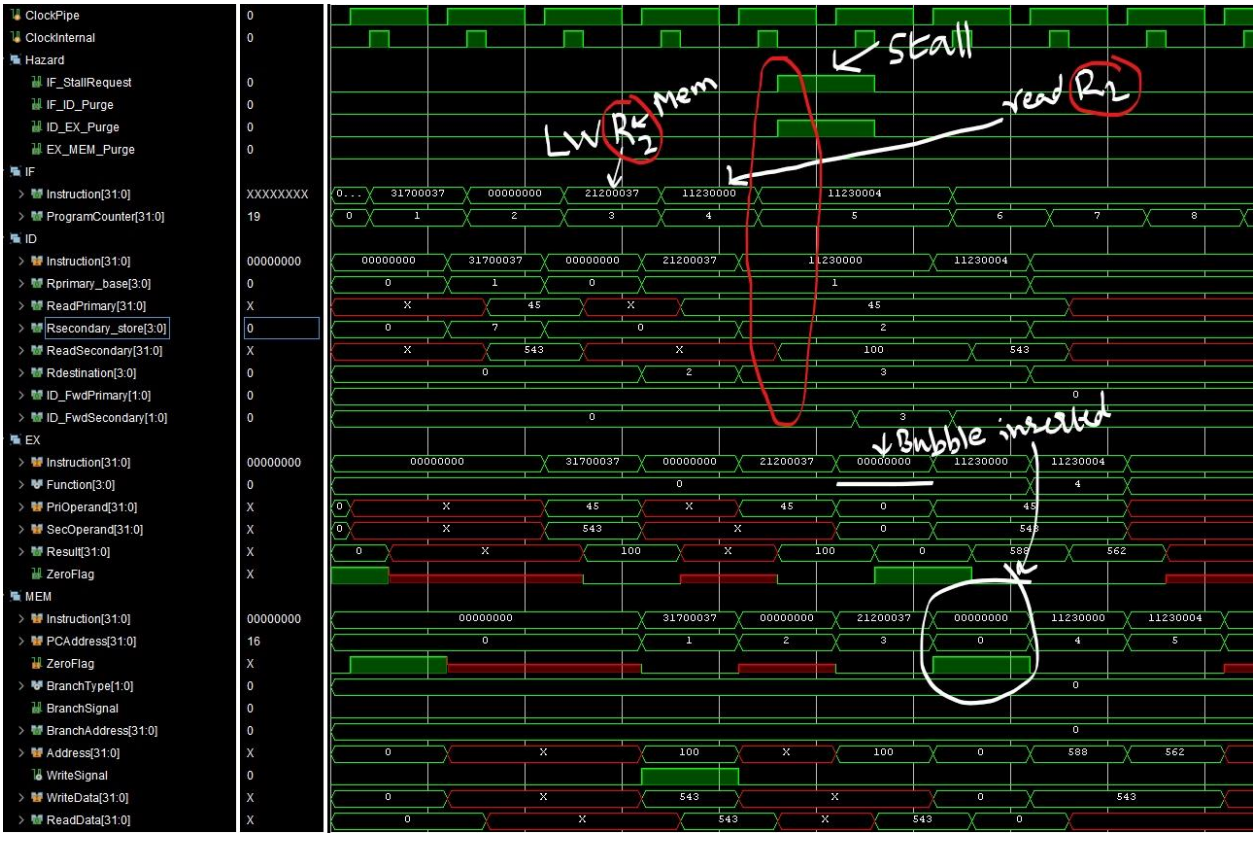
Simulation shown in next page

Simulation shown in next page

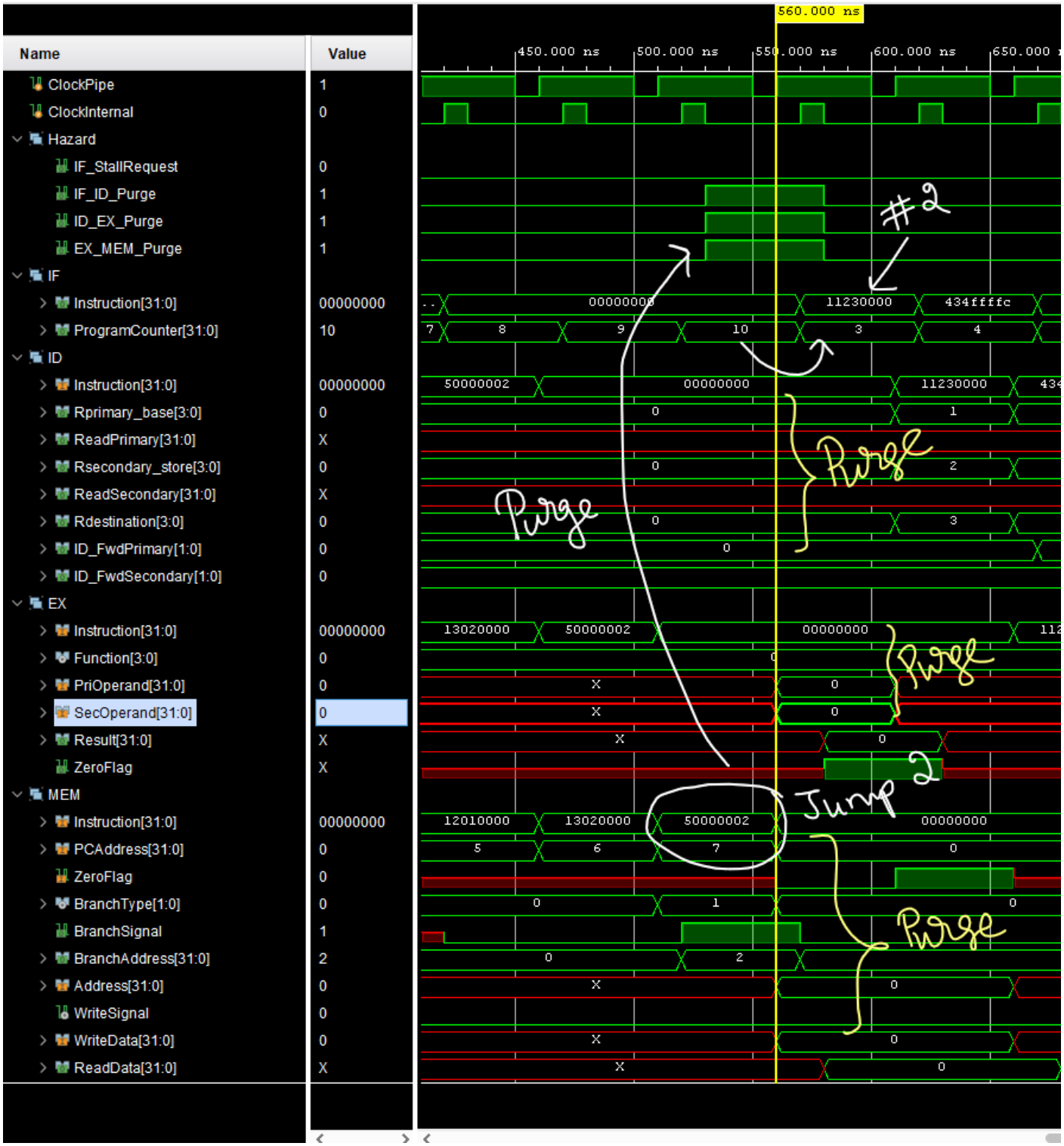
RAW simulation testbench – PORT Forwarding



RAW following Load simulation testbench- Stalling



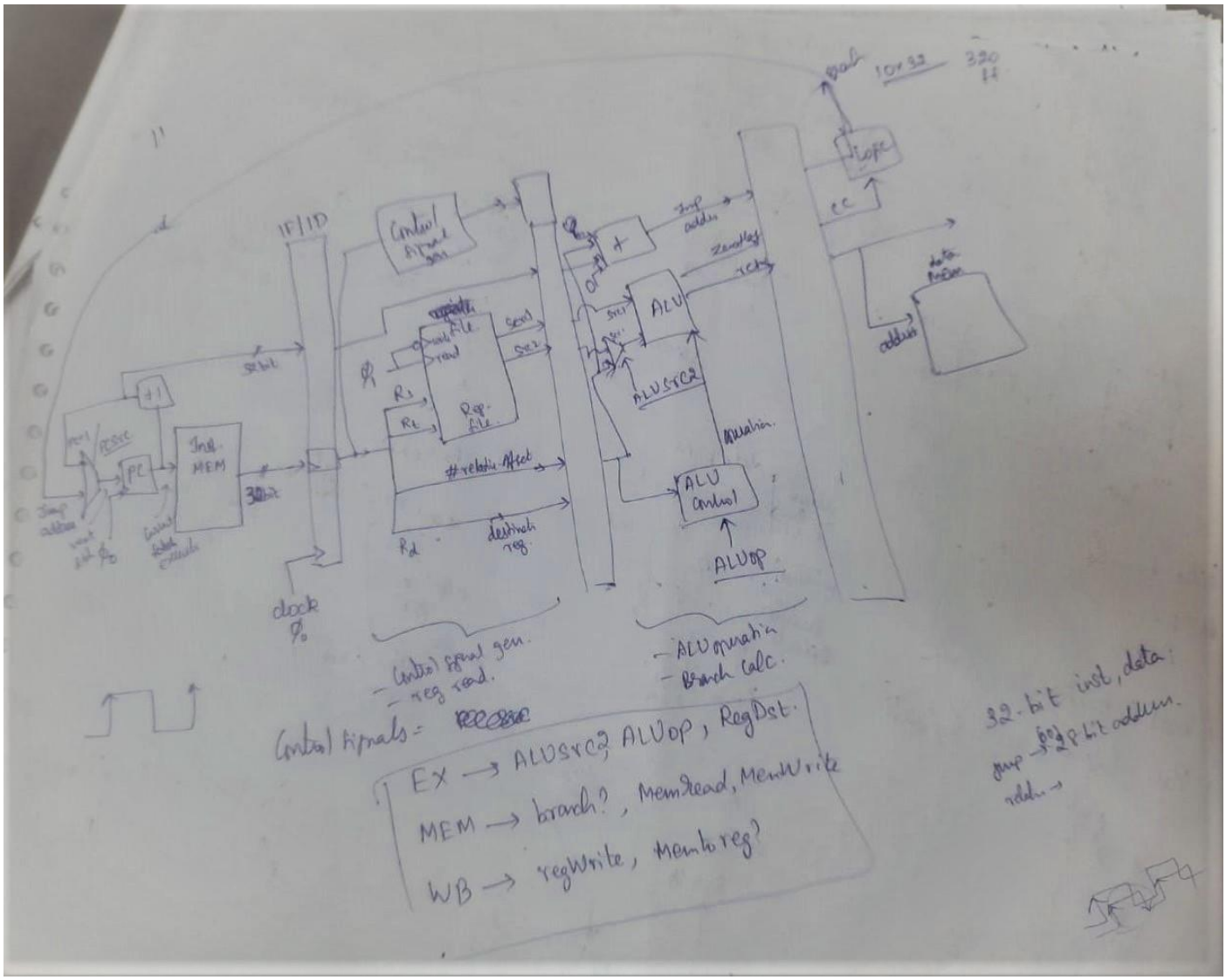
Branching simulation testbench – Purging

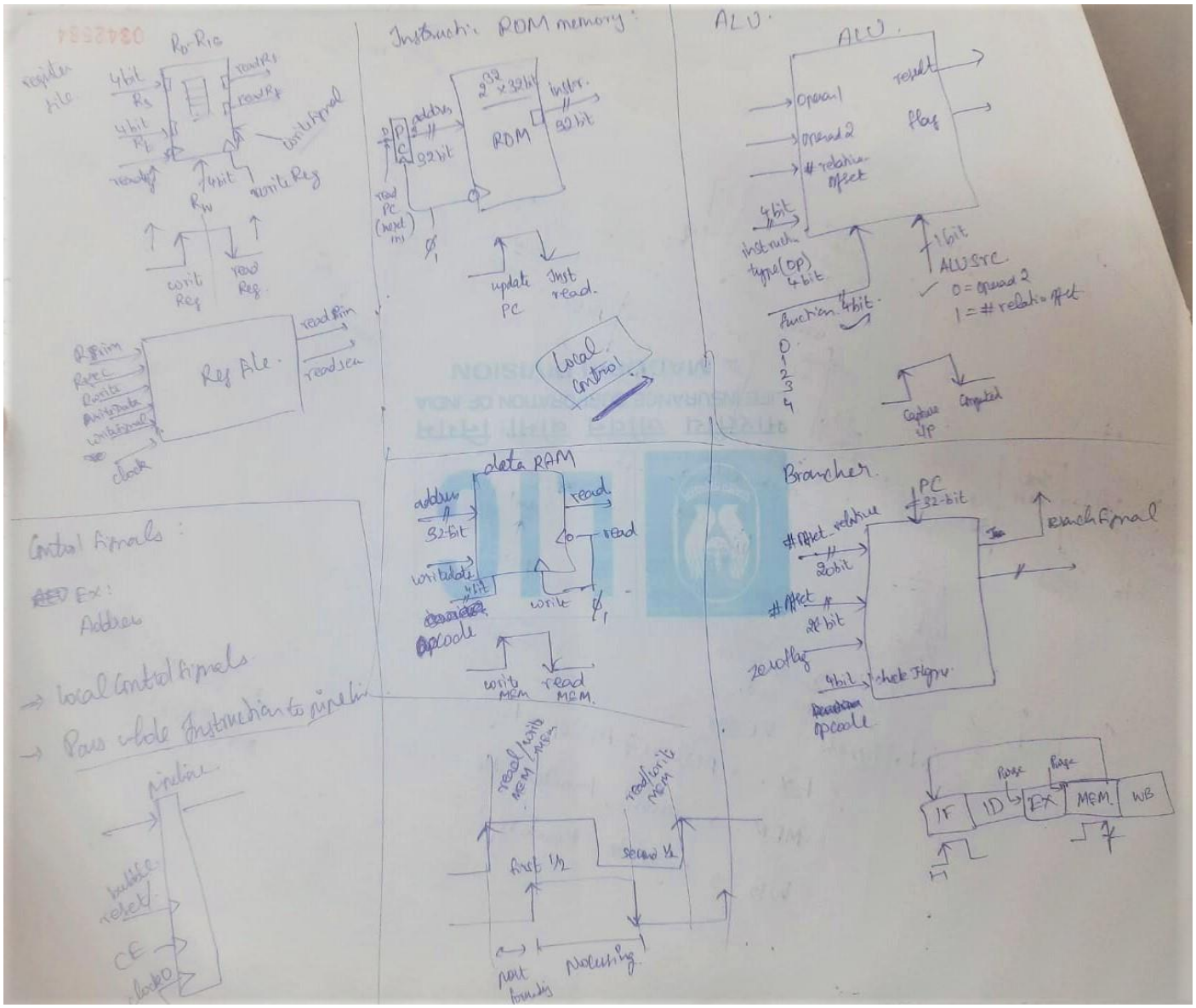


Project design source organization -

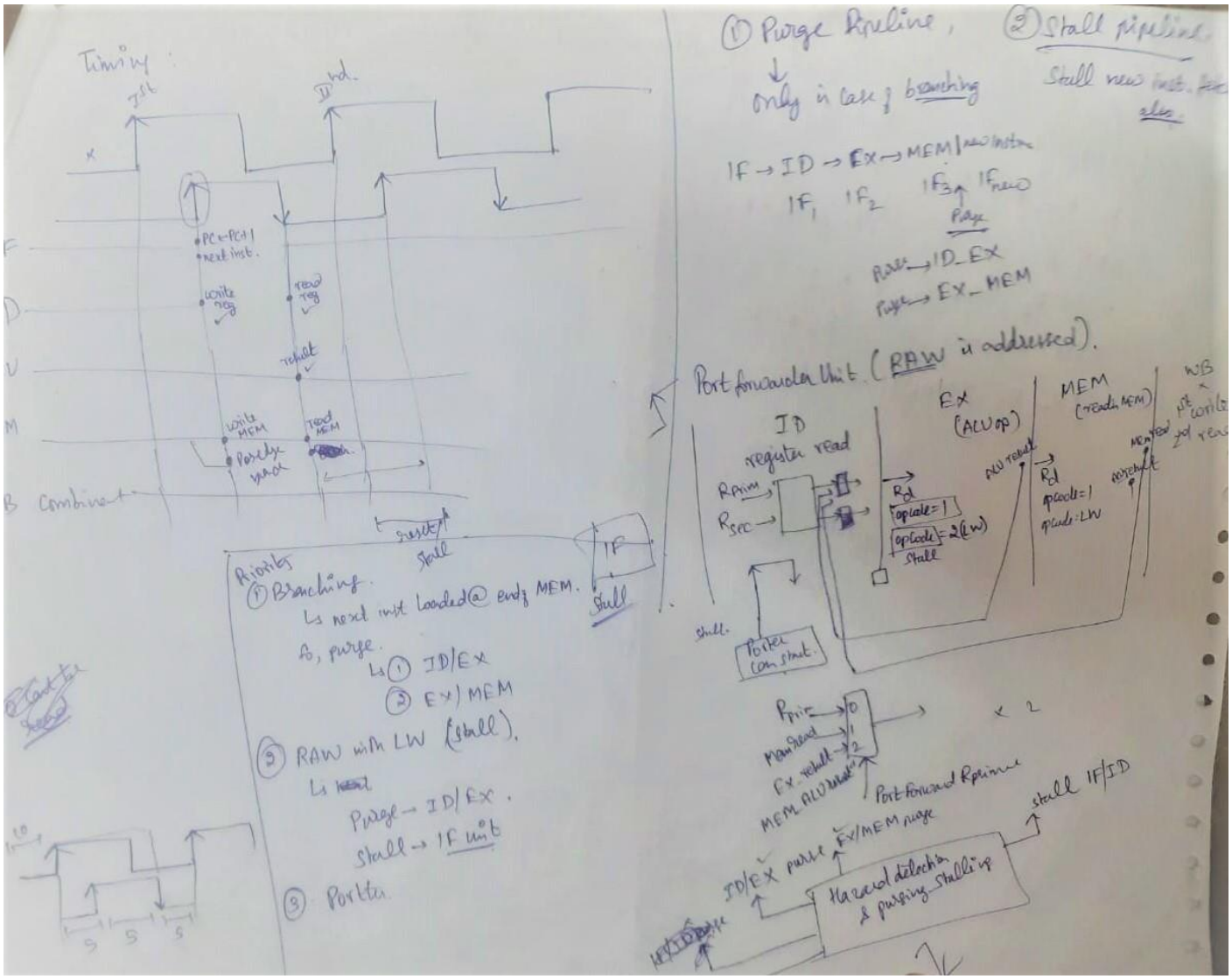
- Design Sources (3)
  - **MIPS_Complete** (MIPS_Complete.v) (7)
    - IF_Module : IFStage (IFStage.v) (1)
      - ProgramMEM : InstructionROM (InstructionROM.v)
    - ID_Module : IDStage (IDStage.v) (2)
      - RegBank : RegisterFile (RegisterFile.v)
      - RegDecoder : Decoder (Decoder.v)
    - EX_Module : EXStage (EXStage.v) (2)
      - ALUUnit : ALU (ALU.v)
      - EXControl : EXLocalController (EXLocalController.v)
    - Memory_Module : MEMStage (MEMStage.v) (3)
      - MEMUnit : DataRAM (DataRAM.v)
      - BranchUnit : Brancher (Brancher.v)
      - MEMControl : MEMLocalController (MEMLocalController.v)
    - WB_Module : WBStage (WBStage.v) (2)
    - PortForwarder_Module : PortForwardingController (PortForwardingControl
    - HazardAddresser_Module : HazardAddresser (HazardAddresser.v)

Rough works



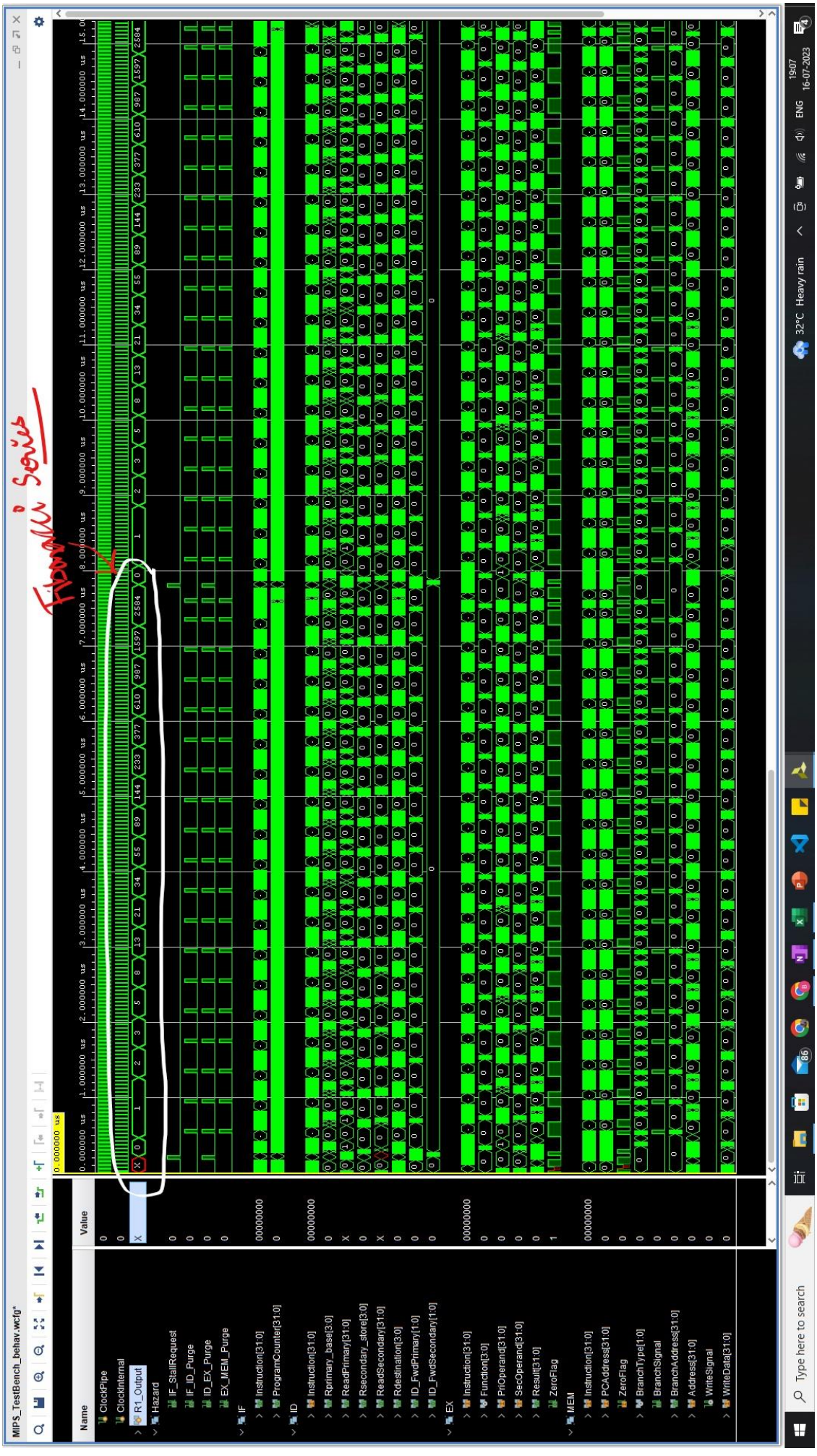


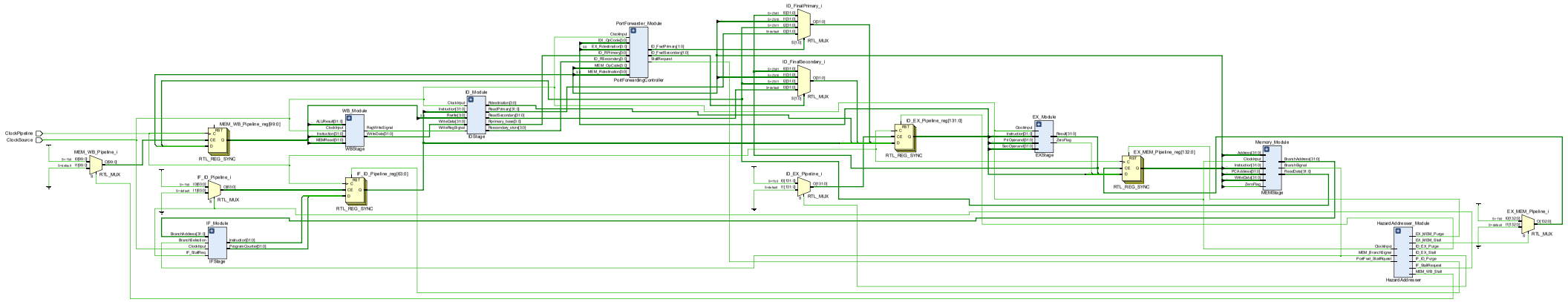
Individual stages design rough work.



Timing related discussion (left). Porting and purging related discussion (right).

Final Schematic and Result




ClockPipeline
ClockSource
MEM_WB_Pipeline_i
RTL_MUX
MEM_WB_Pipeline_reg[89:0]
RTL_REG_SYNC
WB_Module
WBStage
ID_Module
IDStage
PortForwarder_Module
PortForwardingController
ID_FinalPrimary_i
ID_FinalSecondary_i
ID_EX_Pipeline_i
ID_EX_Pipeline_reg[131:0]
EX_Module
EXStage
EX_MEM_Pipeline_reg[132:0]
Memory_Module
MEMStage
IF_ID_Pipeline_i
IF_ID_Pipeline_reg[63:0]
IF_Module
IFStage
Hazard Addresser_Module
HazardAddresser
EX_MEM_Pipeline_i