DESIGN OF A 6-STAGE PIPELINED PROCESSOR,IITB- RISC

213070077 Omkar chavare

Master Of Technology

Department of Electrical Engineering

Indian Institute of Technology , Bombay

Mumbai 400 076

Chapter 1

Introduction to Pipelining

Pipelining is an implementation technique in which multiple instructions are overlapped in execution.

RISC-VI instructions classically take six steps:

1. Fetch instruction from memory.
2. decode the instruction.
3. Read Register.
4. Execute the operation or calculate an address.
5. Access an operand in data memory (if necessary).
6. Write the result into a register (if necessary).

If the stages are perfectly balanced, then the time between instructions on the pipelined processor—assuming ideal conditions—is equal to

$$\mathit{Time\ between\ instructions}\ (\mathit{pipelined}) = \frac{\mathit{Time\ between\ instructions\ (non{-}pipelined)}}{\mathit{No.\ of\ Stages}}$$

Under ideal conditions and with a large number of instructions, the speed-up from pipelining is approximately equal to the number of pipe stages; a six-stage pipelineis nearly six times faster.

However, the stages may be imperfectly balanced. Moreover, pipelining involves some overhead from sources like data forwarding and branch prediction.Thus, the time per instruction in the pipelined processor will exceed the minimum possible, and speed-up will be less than the number of pipeline stages.

Chapter 2

Pipelined Datapath and Control

The division of an instruction into six stages means a six-stage pipeline, which in turn means that up to 6 instructions will be in execution during any single clock cycle. The 6-stages of instruction execution are :

1. IF: Instruction fetch
2. ID: Instruction decode and register file read
3. RG: Register read.
4. EX: Execution or address calculation
5. MEM: Data memory access
6. WB: Write back

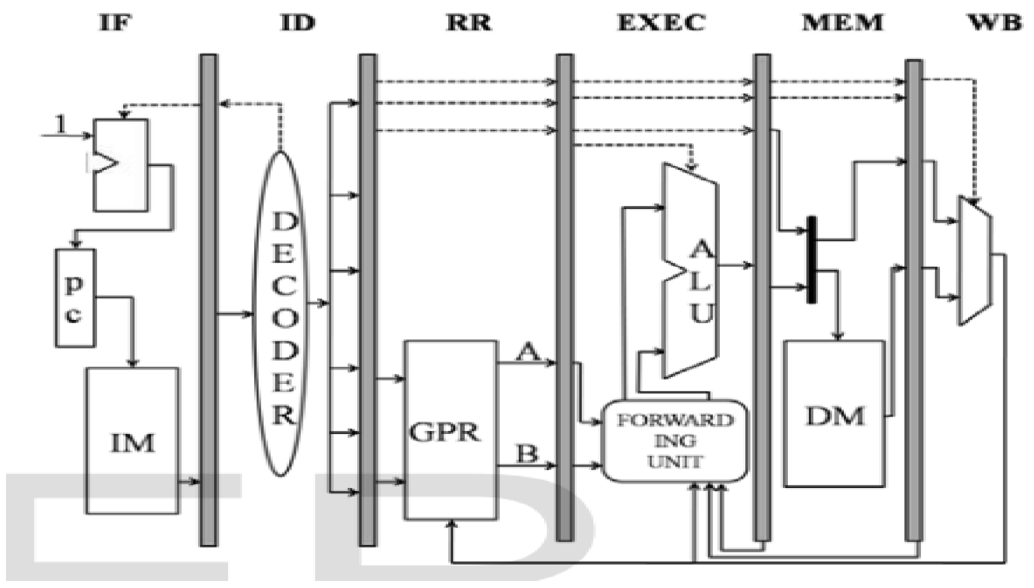


Figure 2.1: The single-cycle pipelined datapath2

Instructions and data move generally from left to right through the 6 stagesas they complete execution. There are, however, two exceptions to this left-to-right flow of instructions:

- The write-back stage, which places the result back into the register file in the middle of the datapath.
- The selection of the next value of the PC, choosing between the incremented PC and the branch address from the MEM stage.

Note : Data flowing from right to left do not affect the current instruction; these reverse data movements influence only later instructions in the pipeline.

- The first right-to-left flow of data can lead to data hazards
- The second leads to control hazards.

To retain the value of an individual instruction for its next stages, we must place registers wherever there are dividing lines between stages in Fig 2.1.

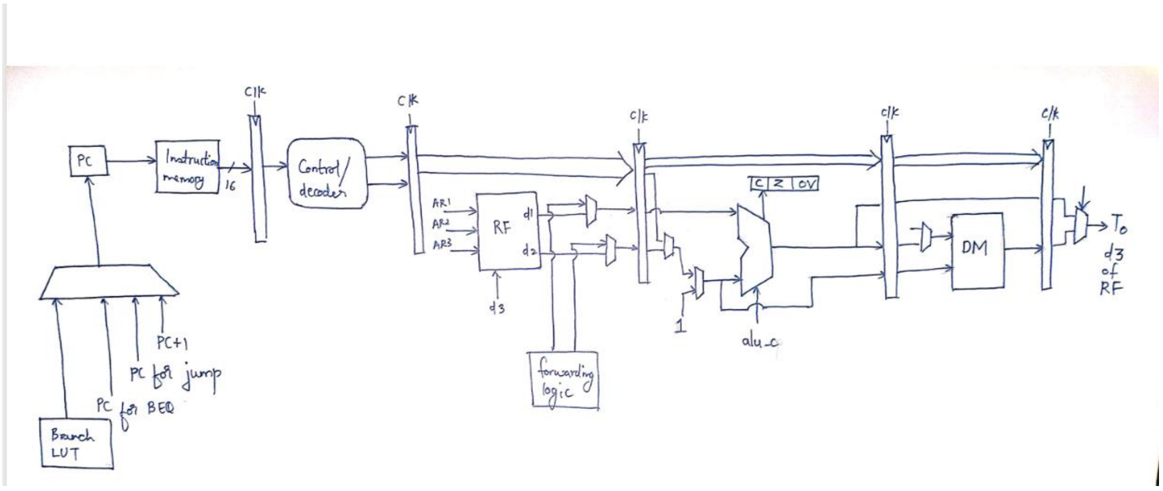


Figure 2.2: Pipelined datapath with the pipeline registers highlighted

All instructions advance during each clock cycle from one pipeline register to the next. The registers are named for the two stages separated by that register. For example, the pipeline register between the IF and ID stages is called IF/ID.The registers must be wide enough to store all the data corresponding to the lines that go through them. All instructions must update some state in the processor—the register file, memory, or the PC.

2.1 Pipeline Datapath :

The Six stages are the following:

1. Instruction fetch:UNIT In the IF stage instructions are fetched one by one from the instruction memory according to the PC value. Program counter (PC) keeps the track of instruction that is being fetched. Instructions are fetched at every clock cycle from instruction memory.

2. INSTRUCTION DECODE This unit read instruction from instruction register and decodes the operands according to operation. The 16 bit instruction will be divided into several parts.

3. REGISTER READ UNIT In this unit we calculate the address of read register and this address is send to the GPR. Data from GPR (general purpose register) is read and forwarded to the next unit. GPR is dual ports RAM in which read and write operations can be done simultaneously at different address.

4. EXECUTION UNIT The main function of this stage is arithmetic calculation. This unit contain ALU unit. The inputs to the ALU are selected by forwarding unit. It decides that from where data is forwarded whether from EXEC-to-EXEC unit forwarding or MEM-toEXEC forwarding or WB-to-EXEC forwarding.

5. DATA MEMORY If there is any data to be written or read from the data memory then this unit is used. So this stage is only used for loading and storing instruction, which read and writes the data memory respectively. For other Instruction this unit is not used. The result of the ALU can be directly stored in the data memory. This unit interface with the data memory.

6. WRITE BACK This stage is used when we need to write back to the GPR. It is used for writing any data from instruction or storing result of the ALU to the GPR

2.2 Pipeline Control :

To specify control for the pipeline, we need only set the control values during each pipeline stage. Because each control line is associated with a component active in only a single pipeline stage, we can divide the control lines into five groups according to the pipeline stage.

1. Instruction fetch: The control signals to read instruction memory and to write the PC are always asserted, so there is nothing special to control in this pipeline stage.

2. Instruction decode: Fetched instruction passed to the decoder which generates instruction specific control signals. These control signals passed to pipelined register from there, it is given to desired blocks.

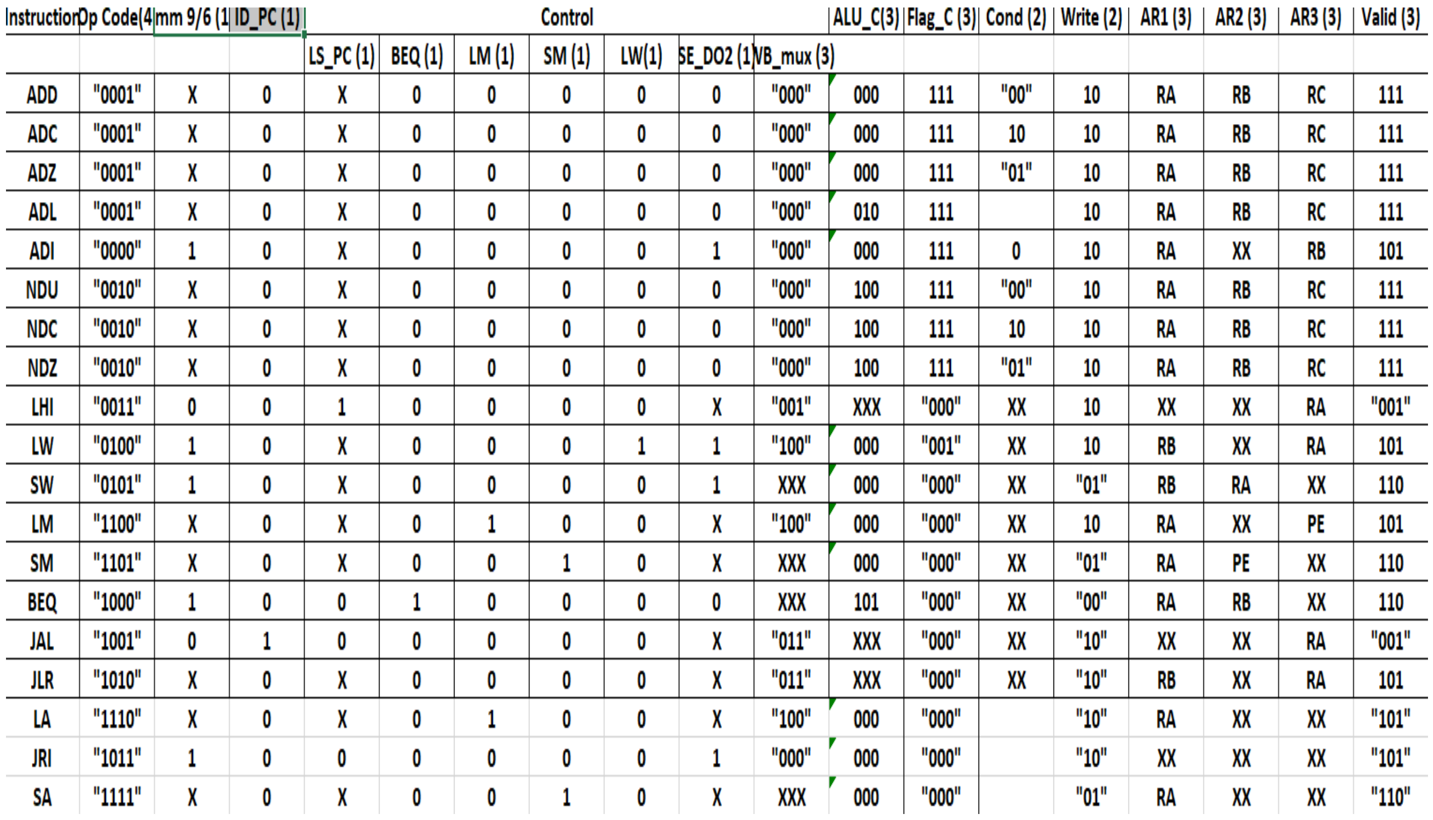
3. Register read: - : The two source registers are always in the same location in the RISC-VI instruction formats, so there is nothing special to control in this pipeline stage.

4. Execution/address calculation: The signals to be set are ALUOp and ALUSrc. The signals select the ALU operation and either Read data 2 or a sign-extended immediate as inputs to the ALU.

5. Memory access: The conrol lines set in this stage are Branch, MemRead, and MemWrite. The branch if equal, load, and store instructions set these signals, respectively. PCSrc selects the next sequential address unless control asserts Branch and the ALU result was 0.

6. Write-back: The two control lines are MemtoReg, which decides between send-ing the ALU result or the memory value to the register file, and RegWrite, which writes the chosen value.

Control signal used per instruction:



Instructior	Op Code(4	mm 9/6 (1	ID_PC (1)				Control				ALU_C(3)	Flag_C (3)	Cond (2)	Write (2)	AR1 (3)	AR2 (3)	AR3 (3)	Valid (3)
				LS_PC (1)	BEQ (1)	LM (1)	SM (1)	LW(1)	SE_DO2 (1	WB_mux (3)								
ADD	"0001"	X	0	X	0	0	0	0	0	"000"	000	111	"00"	10	RA	RB	RC	111
ADC	"0001"	X	0	X	0	0	0	0	0	"000"	000	111	10	10	RA	RB	RC	111
ADZ	"0001"	X	0	X	0	0	0	0	0	"000"	000	111	"01"	10	RA	RB	RC	111
ADL	"0001"	X	0	X	0	0	0	0	0	"000"	010	111		10	RA	RB	RC	111
ADI	"0000"	1	0	X	0	0	0	0	1	"000"	000	111	0	10	RA	XX	RB	101
NDU	"0010"	X	0	X	0	0	0	0	0	"000"	100	111	"00"	10	RA	RB	RC	111
NDC	"0010"	X	0	X	0	0	0	0	0	"000"	100	111	10	10	RA	RB	RC	111
NDZ	"0010"	X	0	X	0	0	0	0	0	"000"	100	111	"01"	10	RA	RB	RC	111
LHI	"0011"	0	0	1	0	0	0	0	X	"001"	XXX	"000"	XX	10	XX	XX	RA	"001"
LW	"0100"	1	0	X	0	0	0	1	1	"100"	000	"001"	XX	10	RB	XX	RA	101
SW	"0101"	1	0	X	0	0	0	0	1	XXX	000	"000"	XX	"01"	RB	RA	XX	110
LM	"1100"	X	0	X	0	1	0	0	X	"100"	000	"000"	XX	10	RA	XX	PE	101
SM	"1101"	X	0	X	0	0	1	0	X	XXX	000	"000"	XX	"01"	RA	PE	XX	110
BEQ	"1000"	1	0	0	1	0	0	0	0	XXX	101	"000"	XX	"00"	RA	RB	XX	110
JAL	"1001"	0	1	0	0	0	0	0	X	"011"	XXX	"000"	XX	"10"	XX	XX	RA	"001"
JLR	"1010"	X	0	X	0	0	0	0	X	"011"	XXX	"000"	XX	"10"	RB	XX	RA	101
LA	"1110"	X	0	X	0	1	0	0	X	"100"	000	"000"		"10"	RA	XX	XX	"101"
JRI	"1011"	1	0	0	0	0	0	0	1	"000"	000	"000"		"10"	XX	XX	XX	"101"
SA	"1111"	X	0	X	0	0	1	0	X	XXX	000	"000"		"01"	RA	XX	XX	"110"

Chapter 3

Pipeline Hazards :

There are situations in pipelining when the next instruction cannot execute in the following clock cycle. These events are called hazards, and there are three different types.

3.1.1 Control Hazard

CONTROL HAZARD Control hazard mainly occur when the branch instruction comes and the instruction in the IF and ID are to be rewritten [7]. And the instructions that are in this stage have to be flushed out. But this type of hazard does not occur in this design because in instruction set we are not using any branch instruction.

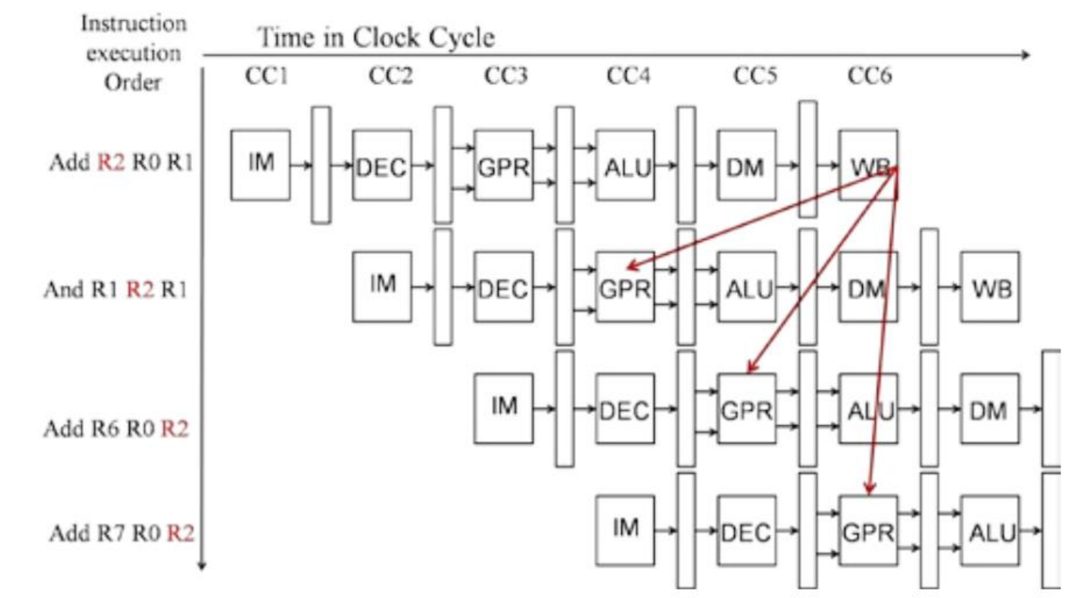


Fig. 4. *Data dependencies in pipelining.*

3.1.1 Assume Branch Not Taken :

One improvement over branch stalling is to predict that the conditional branch will not be taken and thus continue execution down the sequential instruction stream. If the conditional branch is taken, the instructions that are being fetched and decoded must be discarded. Execution continues at the branch target. If conditional branches are not-taken half the time, and if it costs little to discard the instructions, this optimization halves the cost of control hazards.

To discard instructions, we merely change the original control values to 0s. We must also change the three instructions in the IF, ID, and EX stages when the branch reaches the MEM stage. Discarding instructions, then, means we must be able to flush instructions in the IF, ID, and EX stages of the pipeline.

3.1.2 Reducing the Delay of Branches:

Till now we have assumed the next PC for a branch is selected in the MEM stage, but if we move the conditional branch execution earlier in the pipeline, then fewer instructions need be flushed. Moving the branch decision up requires two actions to occur earlier: computing the branch target address and evaluating the branch decision. The easy part of this change is to move up the branch address calculation. We already have the PC value and the immediate field in the IF/ID pipeline register, so we just move the branch adder from the EX stage to the ID stage; of course, the address calculation for branch targets will be performed for all instructions, but only used when needed.

The harder part is the branch decision itself. For branch if equal, we would compare two register reads during the ID stage to see if they are equal. Equality can be tested by XORing individual bit positions of two registers and ORing the XORed result. (A zero output of the OR gate means the two registers are equal.)

3.1.2 Dynamic Branch Prediction:

One approach is to look up the address of the instruction to see if the conditional branch was taken the last time this instruction was executed, and, if so, to begin fetching new instructions from the same place as the last time. This technique is called dynamic branch prediction.

One implementation of that approach is a branch prediction buffer or branch history table. A branch prediction buffer is a small memory indexed by the lower portion of the address of the branch instruction. The memory contains a bit thatsays whether the branch was recently taken or not.

This prediction uses the simplest sort of buffer. Prediction is just a hint that we hope is accurate so fetching begins in the predicted direction. If the hint turns out to be wrong, the incorrectly predicted instructions are deleted, the prediction bit is inverted and stored back, and the proper sequence is fetched and executed.

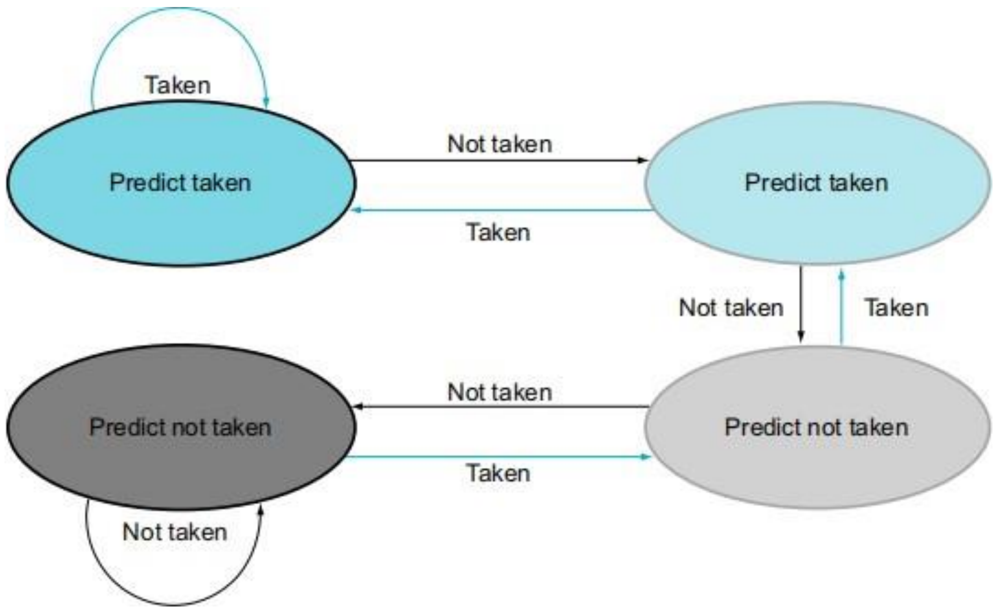


Figure 3.7: The states in a 2-bit prediction scheme.

This simple 1-bit prediction scheme has a performance shortcoming: even if a conditional branch is almost always taken, we can predict incorrectly twice, rather than once, when it is not taken.Ideally, the accuracy of the predictor would match the taken branch frequency for these highly regular branches. To remedy this weakness, we can use more prediction bits. In a 2-bit scheme, a prediction must be wrong twice before it is changed. Fig 3.7 shows the finite-state machine for a 2-bit prediction scheme.

By using 2 bits rather than 1, a branch that strongly favors taken or not taken—as many branches do—will be mispredicted only once. The 2 bits are used to en- code the four states in the system. The 2-bit scheme is a general instance of acounter-based predictor, which is incremented when the prediction is accurate and decremented otherwise, and uses the mid-point of its range as the division between taken and not taken.

3.2 Data Hazard

This type of hazard commonly comes when an instruction depends on the result of previous instruction or any data which is not yet generated. In Fig.4 one instruction will generate a result which will written in the R2 in Clock Cycle 6(CC6) but next instruction need this result in CC4 and a false data will read from R2 in CC4, to remove such problem we have to wait until the result will be written to the register R2 in CC6. Pipelining is stall for some clock cycle and it will decrease the performance of the processor [7].Fig.4 shows the example pipelined data dependencies and Fig.5 and Fig.6 shows how these dependencies are resolved

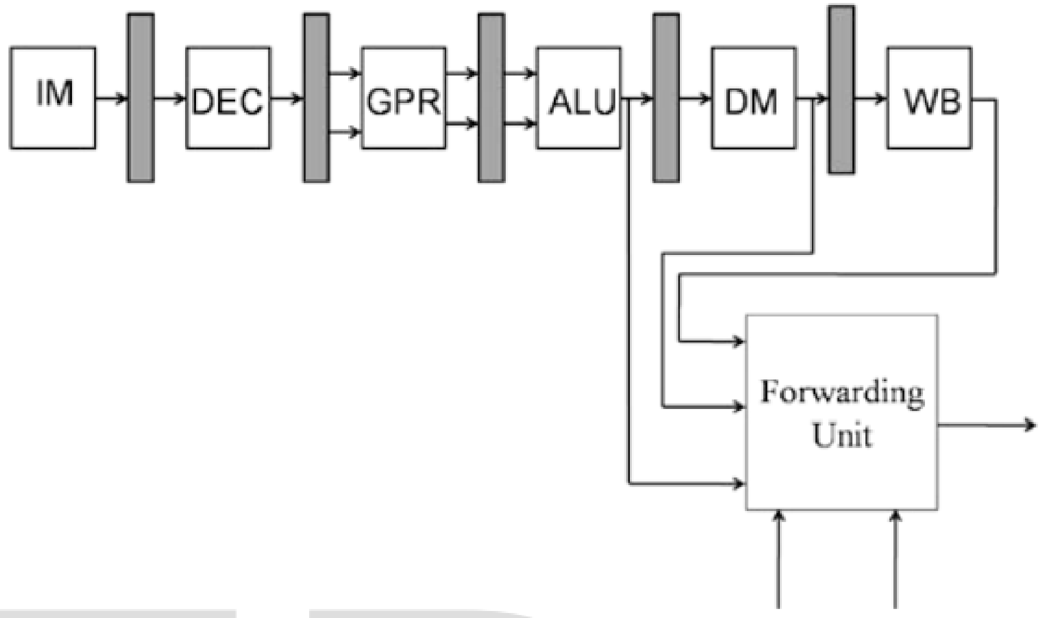


Fig. 5. *Forwarding Unit.*

To remove such type of hazard data forwarding technique is used shown in Fig 5. Hazard unit Compare the addresses of the Source registers of Current instruction decoded in ID unit with the address of destination register of the previous 3 instruction, according to these address Hazard unit send control signal to the forwarding unit. Current result of ALU and previous result of ALU which are in next pipelined stage are forwarded to Data forwarding unit so that according to control signal forwarding unit send data to ALU.

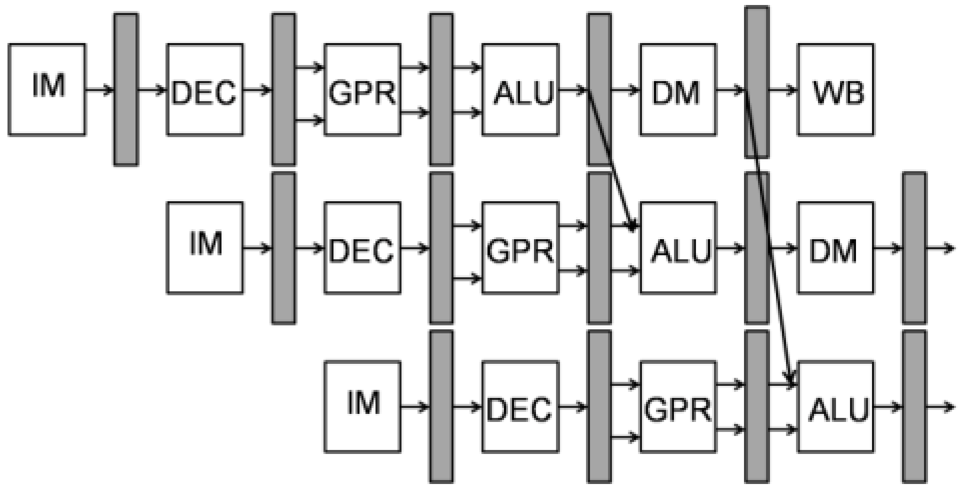


Fig. 6. *Data forwarding.*

Hazard unit Compare the addresses of the Source registers of Current instruction decoded in ID unit with the address of destination register of the previous 3 instruction, according to these address Hazard unit send control signal to the forwarding unit. Current result of ALU and previous result of AlU which are in next pipelined stage are forwarded to Dataforwarding unit so that according to control signal forwarding unit send data to ALU Show in Fig.5. In this case if result of the first instruction is generated by the ALU then this result is feed back to the input of the ALU. And there is no need to wait for result written in the R2. A forwarding unit is used for data forwarding, it forward the result to EX stage from EX, MEM and WB

Chapter 4

Implementation

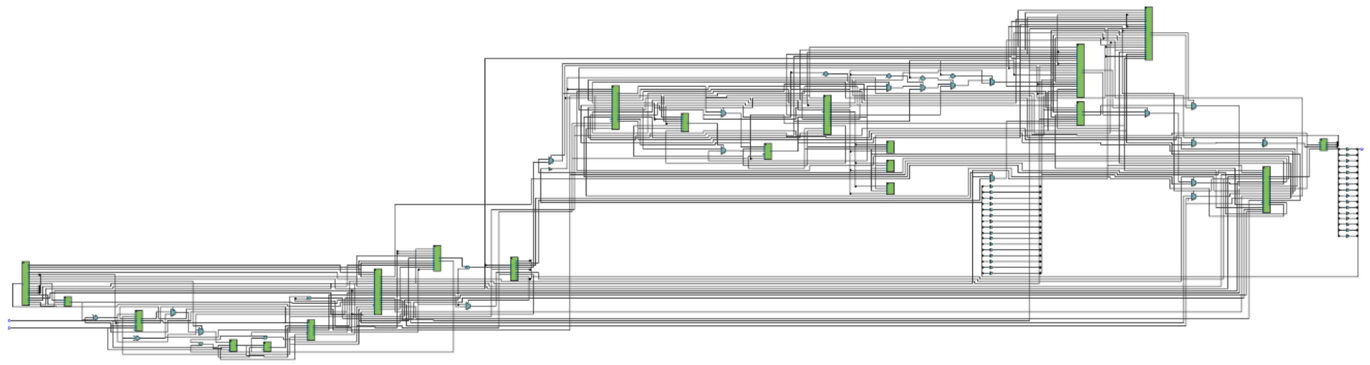
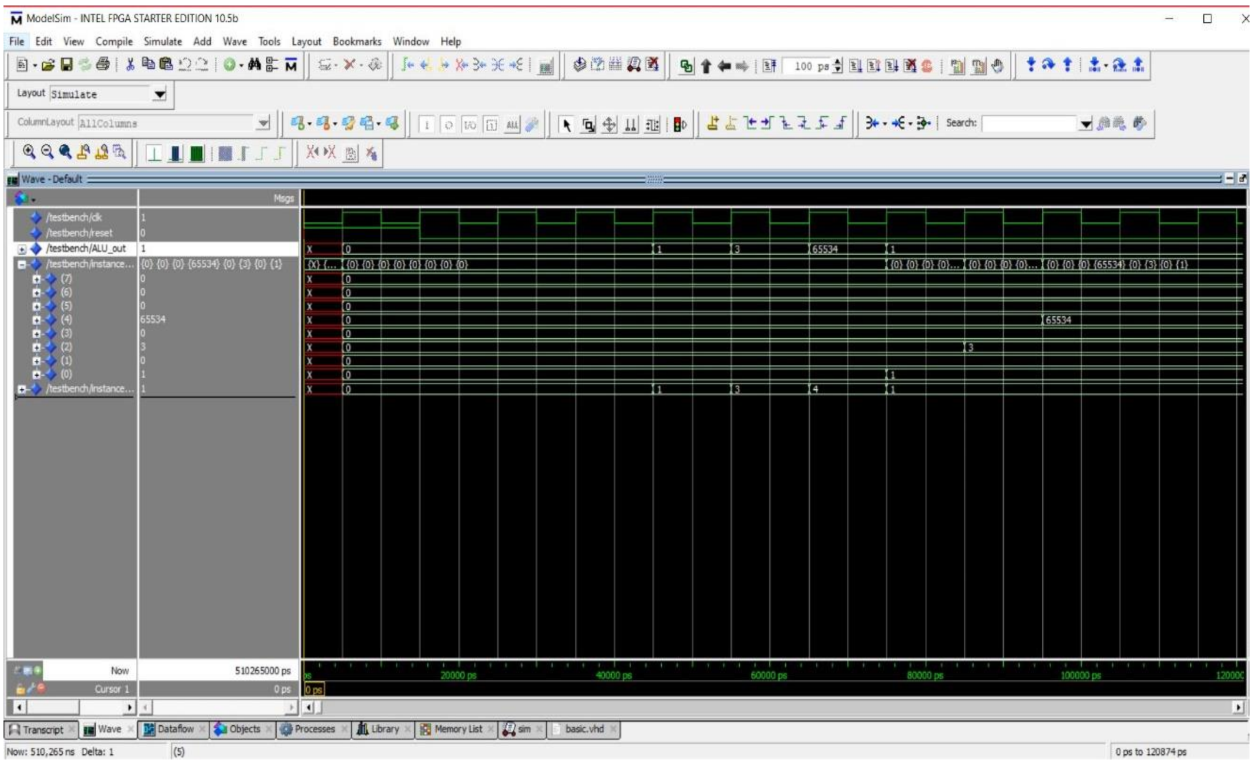


Figure 4.1: Implementation : RTL View

Chapter 5

Results Of Simulation

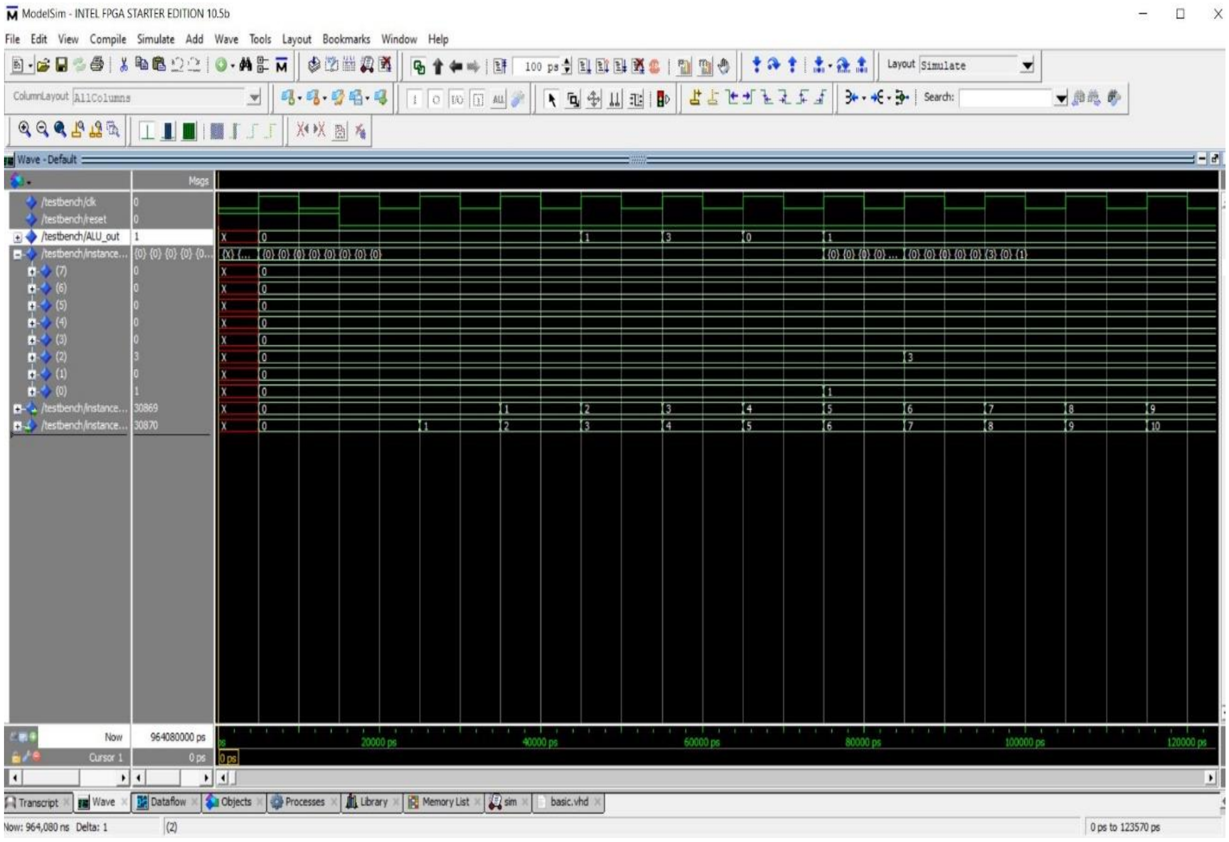


Results : Waveform of Add and ADI instruction

adi \$3 \$0 1
adi \$1 \$2 3
add \$0 \$2 \$4

register R0 and R2 gets loaded by 1,3 respectively as R3 and R1 contains 0 value.. Their addition is done in alu which can be seen at last row in simulation. But, during simulation latch is forming at output; because of that output of ALU changes to 65534 which gets stored in R4.

Waveform for LM instruction
LM $1 0 11111111
As we can see R1 contains 0 value which is the address of memory and it will load the content of the memory of location 0 to 7 to register R0 to R7; as all are given high in the immediate field.



	Msgs
/testbench/clk	0
/testbench/reset	0
/testbench/ALU_out	1
/testbench/instance...	{0} {0} {0} {0} {0...
(7)	0
(6)	0
(5)	0
(4)	0
(3)	0
(2)	3
(1)	0
(0)	1
/testbench/instance...	30869
/testbench/instance...	30870

```
Now   964080000 ps
Cursor 1   0 ps
Now: 964,080 ns  Delta: 1     (2)     0 ps to 123570 ps
```

Chapter 6

Conclusion

IITB-RISC is a 16-bit very simple computer developed for the teaching that is based on the Little Computer Architecture. The IITB-RISC is an 8-register, 16-bit computer system. It has 8 general-purpose registers (R0 to R7). Register R7 is always stores Program Counter. All addresses are short word addresses (i.e. address 0 corresponds to the first two bytes of main memory, address 1 corresponds to the second two bytes of main memory, etc.). This architecture uses condition code register which has two flags Carry flag (C) and Zero flag (Z).

The IITB-RISC is very simple, but it is general enough to solve complex prob- lems. The architecture allows predicated instruction execution and multiple load and store execution. There are three machine-code instruction formats (R, I, and J type) and a total of 14 instructions.

We successfully designed and try to implement a 6 stage pipelined processor, IITB-RISC, whose instruction set architecture is provided to us. IITB-RISC is a 16-bit very simple computer developed for the teaching that is based on the Little Computer Architecture. The IITB-RISC is an 8-register, 16-bit computer system. It follows the standard 6 stage pipelines (Instruction fetch, instruction decode and register read, execute, memory access, and write back). The architecture is optimized for performance, i.e., it includes hazard mitigation techniques like forwarding and branch prediction technique.

Chapter 7

References

- Computer Organization and Design
  THE HARDWARE AND SOFTWARE INTERFACE (RISC-V EDITION)
  AUTHOR:- David A. Patterson, John L. Hennessy
  SERIES:-The Morgan Kaufmann Series in Computer Architecture And Design

- PROCESSOR DESIGN CLASS NOTES
  AUTHOR -: PROF VIRENDRA SINGH