Simulating a Pipelined RISC Processor

Amit Pandey*, *Member, IEEE*
Department of Computer Science,
Sunrise University,
Alwar, India
amit.pandey@live.com

***Abstract*— Most of the processors available now days have a pipelined architecture. The pipelined processors are designed in a way that they can process more than one instructions in the same clock pulse. To achieve this phenomenon of multiple processing, the processors are designed to have multiple stages. These stages work together as single unit under common clock pulse to generate the effect of multiple processing. Generally we have Instruction fetch, Instruction decode, Execution, Memory and Wright back stages in a five stage pipelined RISC processor. Each of these stages works as a sub processor connected in the sequence. Such that with every clock pulse each of them process one instruction and forwards it to the next stage in the sequence. To understand the implementation of these concepts, in this paper we have materialized a working simulation of a pipelined RISC processor.**

Keywords— Pipelined processor, Pipelined architecture, Processor simulation, Pipelined RISC architecture.

I. INTRODUCTION

There are Instruction fetch, Instruction Decode, Execution, Memory and Write back stages in any five stage pipelined RISC processor. This stages operate in a synchronized manner under single clock cycle and forward one instruction from one stage to the next stage in the sequence [1]. *(See Fig. 1)*

The architecture described above seems to be sound and perfect. But in reality there exist few hazards that may be present in the design. These hazards are mainly of three types Data hazards, Control hazards and Structural hazards [1].

Data hazards are caused when an instruction is trying to read a register in general purpose register set, whose value is being evaluated in the preceding instruction but not yet updated in the general purpose register set. These hazards can be resolved by forwarding values directly between the stages of the processor. While resolving these hazards in some cases empty cycles or stalls are also induced between the instructions. There are eight different cases of Data hazards that are possible in a Pipelined RISC processor [1].

Control hazards are caused because it is not decided which instruction to fetch next after any branch instruction leaves the Instruction fetch stage. As the branch instructions are evaluated at the end of Instruction decode stage, so by the time branch instruction enters the Instruction decode stage it is not known which instruction to fetch next in instruction fetch stage[2][3][4]. The easiest way to resolve this hazard is to induce a stall after the branch instruction. But this approach is quite expensive in terms of processor efficiency. Hence some better approaches like static and dynamic branch predictions were incorporated in practice to resolve these hazards [5][6][7][8].

Also, the structural hazards are caused because of any designing flaw in the data-path of the processor.

II. SIMULATING THE PIPELINED PROCESSOR

We have used Logic works software to design the working simulation for the pipelined RISC processor. The first step is to design the instruction set for our processor *(See Table I)*, which will guide us through the further process. *(See Fig. 2)*

TABLE I. Instruction Set.

ALU Instructions	
OP.Code	Instruction
0000	No Operation
0001	Subtraction
0010	Addition
0011	Add Immediate
0100	Shift Right
0101	Shift Left
0110	Logical And
0111	Logical OR
1000	Logical NOR
1001	Logical NAND
1010	Logical XOR
Load - Store Instructions	
OP.Code	Instruction
1011	Load
1100	Store
Branch - Jump Instructions	
OP.Code	Instruction
1101	Branch Equal
1110	Branch Not Equal
1111	Jump

* PH.D Scholar at Sunrise University, Alwar, India

Opcode	RD	RS1	RS2

2a) R Type Instruction Format: For ALU operations

Opcode	RD	RS1	Immediate

2b) I Type Instruction Format: For Load, Store, Immediate and Branch operations.

Opcode	Offset Value

2c) J Type Format: For Jump operation

Fig. 2. Types of Instruction Formats.

A. *The Instruction Fetch Stage*

Instruction fetch is the first stage in the pipelined RISC processor. It works as the entry point for the incoming instructions. It has Program counter *(See Fig. 3)* and Instruction memory as its main components. This stage is responsible for calculating the next instruction address and then forwarding it to the next stage.

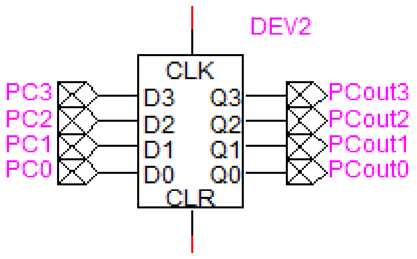


Fig. 3. Program Counter Register.

B. *The Instruction Decode Stage*

Instruction decode stage contains the General purpose register set. Whenever the value of any register is altered then it must be updated here in the General purpose register set. Further this stage is also responsible for detecting and inducing the stalls between the instructions. *(See Fig. 4)*

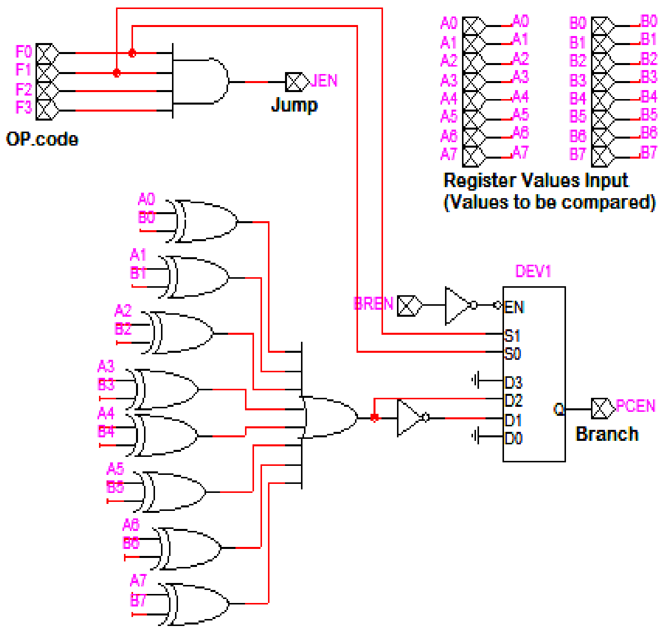


Fig. 5. Branch Detector Circuit.

The branch detector is also part of this stage, Which compares the register values and detects whether the branch will be taken or not. *(See Fig. 5)*

C. *The Execution Stage*

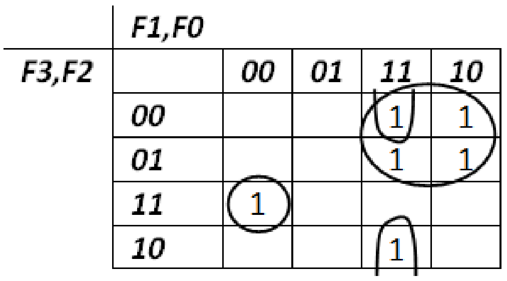
Execution stage deals with the evaluation of all arithmetic and logic operations. The ALU is part of Execution stage in a pipelined RISC processor. There are two multiplexers, used to forward the selected values for each of its operands, in this stage.

We have used the 74181 chip provided with the Logic works toolkit to design our ALU. Initially we have to designed the table which maps the operations in our proposed instruction set to the operations performed by the 74181 chip. *(See Table II)*

TABLE II. Mapping of Operations With 74181 Chip.

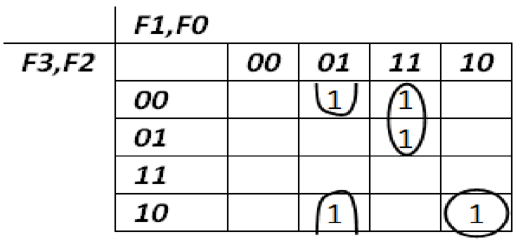
Proposed Opcode				**Operations in 74181**						**Operation Performed**
F3	*F2*	*F1*	*F0*	*M*	*S3*	*S2*	*S1*	*S0*	*C_{-1}'*	
0	0	0	0	0	0	0	0	0	1	No Operation
0	0	0	1	0	0	1	1	0	0	A - B
0	0	1	0	0	1	0	0	1	1	A + B
0	0	1	1	0	1	1	1	1	1	Add Immediate
0	1	0	0	0	0	0	0	0	1	Circular Right Shift
0	1	0	1	0	0	0	0	0	1	Circular Left Shift
0	1	1	0	1	1	0	1	1	x	A AND B
0	1	1	1	1	1	1	1	0	x	A OR B
1	0	0	0	1	0	0	0	1	x	A NOR B
1	0	0	1	1	0	1	0	0	x	A NAND B
1	0	1	0	1	0	1	1	0	x	A XOR B
1	0	1	1	0	1	0	0	1	1	Load
1	1	0	0	0	1	0	0	1	1	Store
1	1	0	1	0	0	0	0	0	1	Branch equal
1	1	1	0	0	0	0	0	0	1	Branch Not Equal
1	1	1	1	0	0	0	0	0	1	Jump

Now, we have to generate Boolean expressions for each of the columns of the " Operations in 74181" in terms of our Opcode inputs. For this purpose we will use K-Maps. *(See Fig. 6)*



S3 = F3'.F1 + F2'.F1.F0 + F3.F2.F1'.F0'

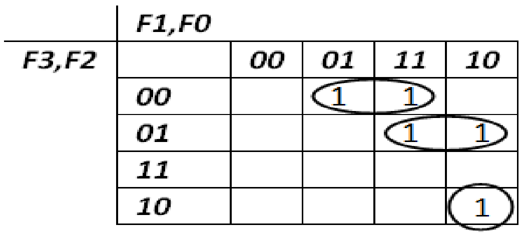
6a) Generating expression for S3.



F3,F2 \ F1,F0	00	01	11	10
00		1	1	
01			1	
11				
10		1		1

$$S2 = F2'.F1'.F0 + F3'.F1.F0 + F3.F2'.F1.F0'$$

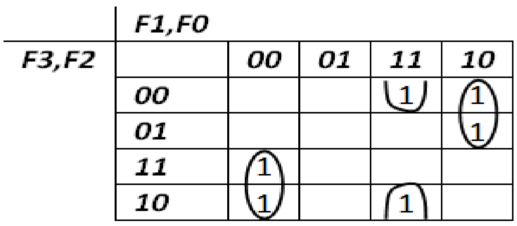
6b) Generating expression for S2.



F3,F2 \ F1,F0	00	01	11	10
00		1	1	
01			1	1
11				
10				1

$$S1 = F3'.F2'.F0 + F3'.F2.F1 + F3.F2'.F1.F0'$$

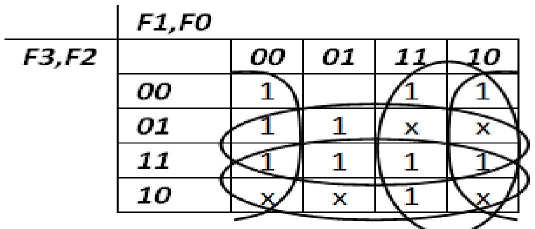
6c) Generating expression for S1.



F3,F2 \ F1,F0	00	01	11	10
00			1	1
01				1
11	1			
10	1		1	

$$S0 = F3.F1'.F0' + F2'.F1.F0 + F3'.F1.F0'$$

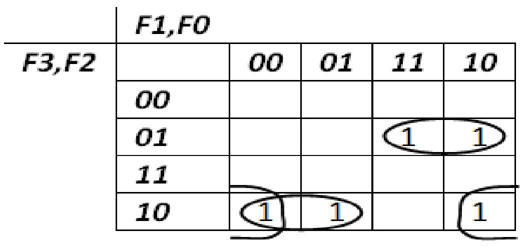
6d) Generating expression for S0.



F3,F2 \ F1,F0	00	01	11	10
00	1		1	1
01	1	1	x	x
11	1	1	1	1
10	x	x	1	x

$$(C_{-1})' = F0' + F1 + F2 + F3$$

6e) Generating expression for carry $(C_{-1})'$.



F3,F2 \ F1,F0	00	01	11	10
00				
01			1	1
11				
10	1	1		1

6f) Generating expression for Mode Select M.

Fig. 6. Generating the Boolean expressions for ALU.

Now using these expressions and 74181 chip design the circuit for ALU. *(See Fig. 7)*

D. *The Memory Stage*

The job of this stage is to load values in to memory or to get values from memory and save them in to registers. We have used a inbuilt wizard in Logic works to create a memory chip for our design *(See Fig. 8)*. All the instructions except the Load and Save are bypassed by this stage [1].

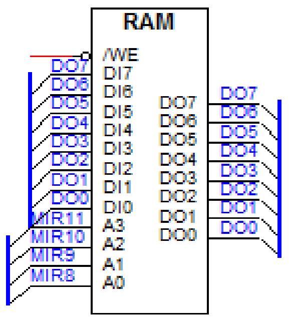


Fig. 8. The Memory chip.

E. *The Write back Stage*

The job of Write back stage is to update the final results of the instructions in to respective registers in the general purpose register set.

III. Results

We have executed the code given below on the simulated processor to confirm its validity. (Code I).

```
LOAD  RS1, #3

LOAD  RS2, #1

ADD RS3, RS1, RS2

ADD RS4, RS1, RS3

SUBSTRACT  RS5, RS4, RS1
```

There is a data dependency between the second Load operation and the third ALU operation. To resolve this dependency a stall will be induced between the instructions. This stall can also be seen in the results as a empty cycle. *(See Fig. 9)*

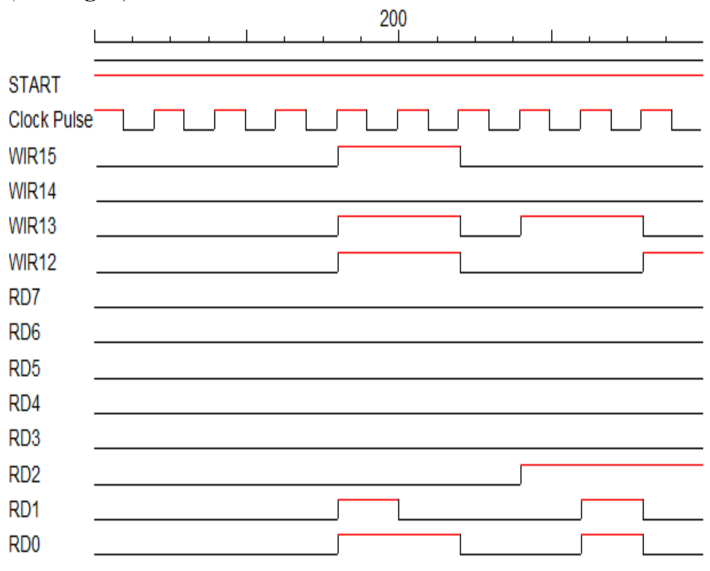


Fig. 9. Output of the code I.

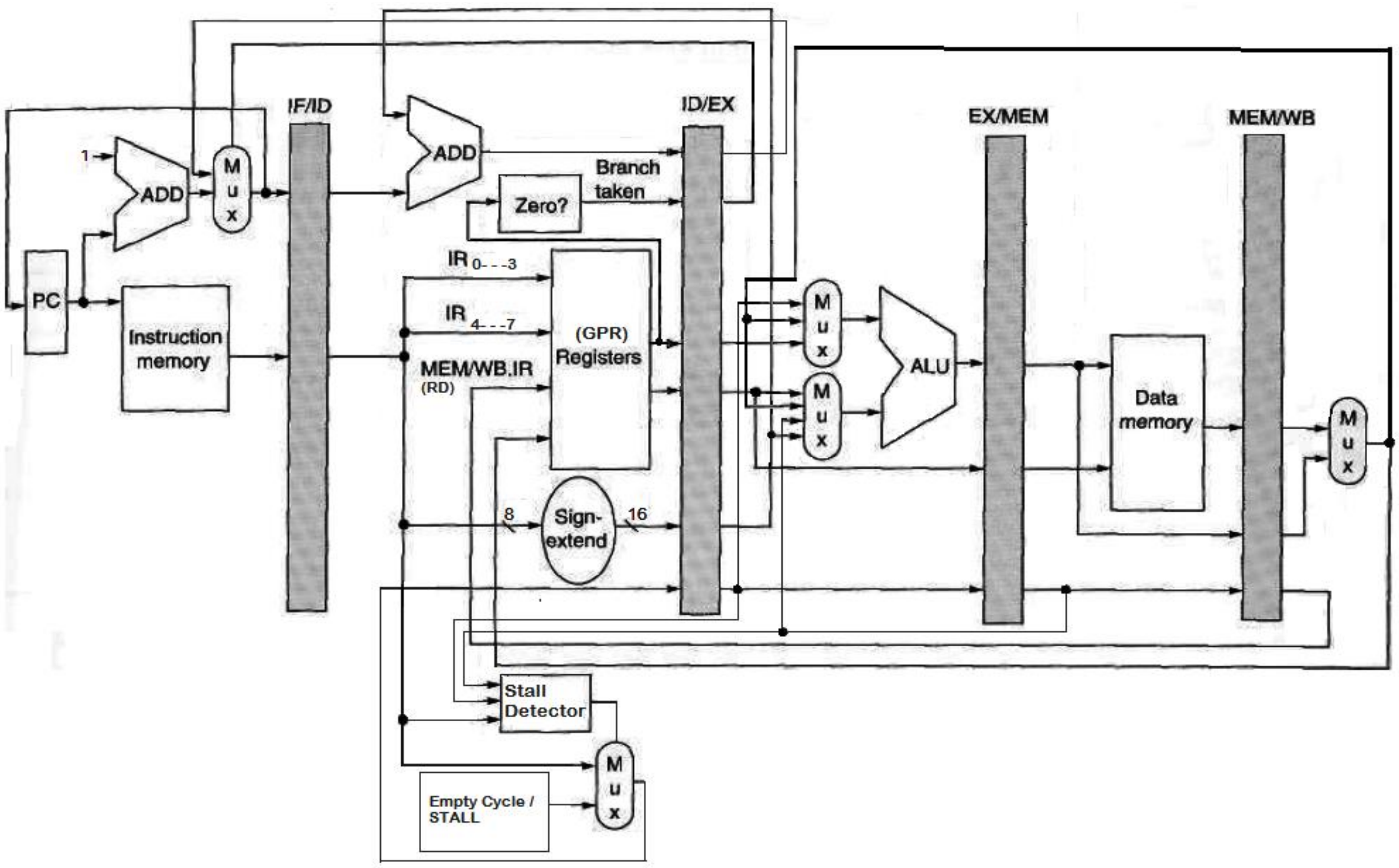


Fig. 1. Conceptual Data path design for a five stage pipelined RISC processor.

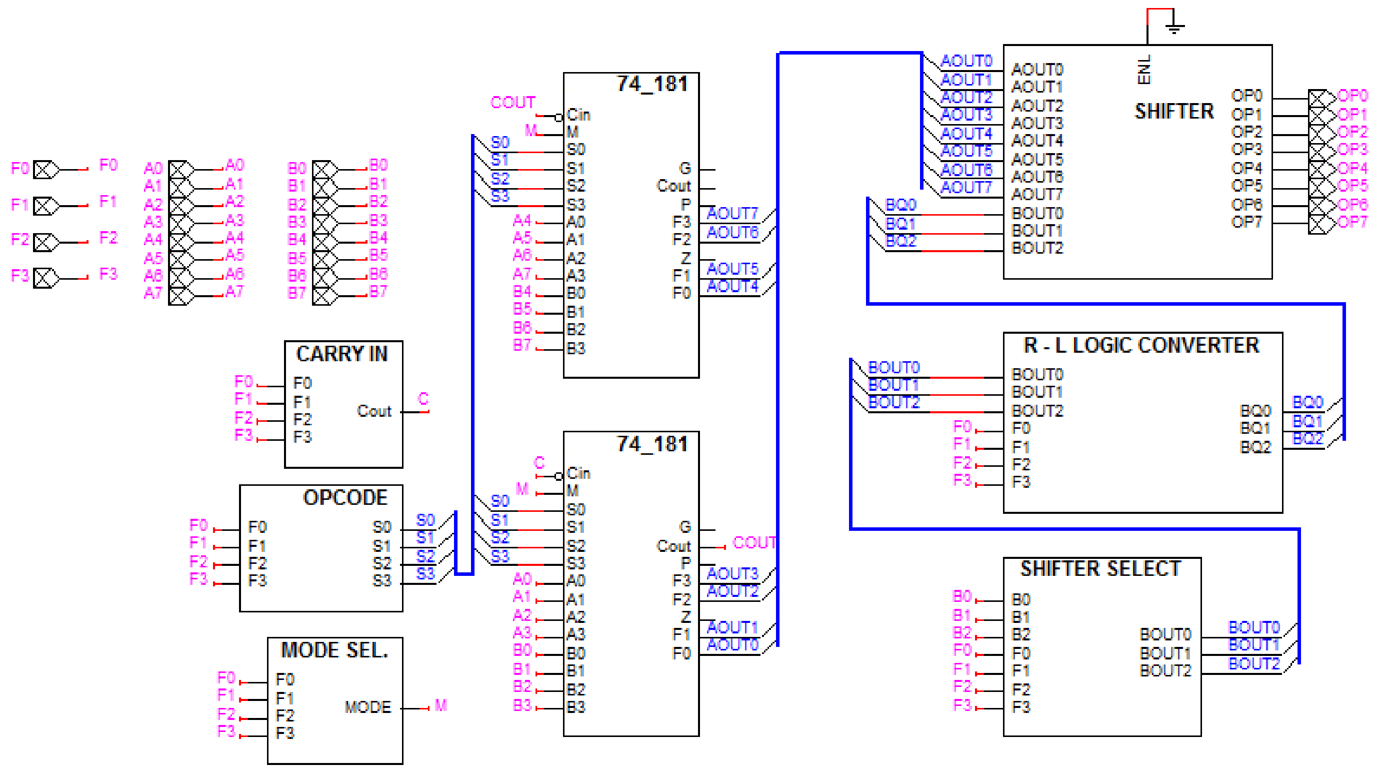


Fig. 7. ALU Circuit

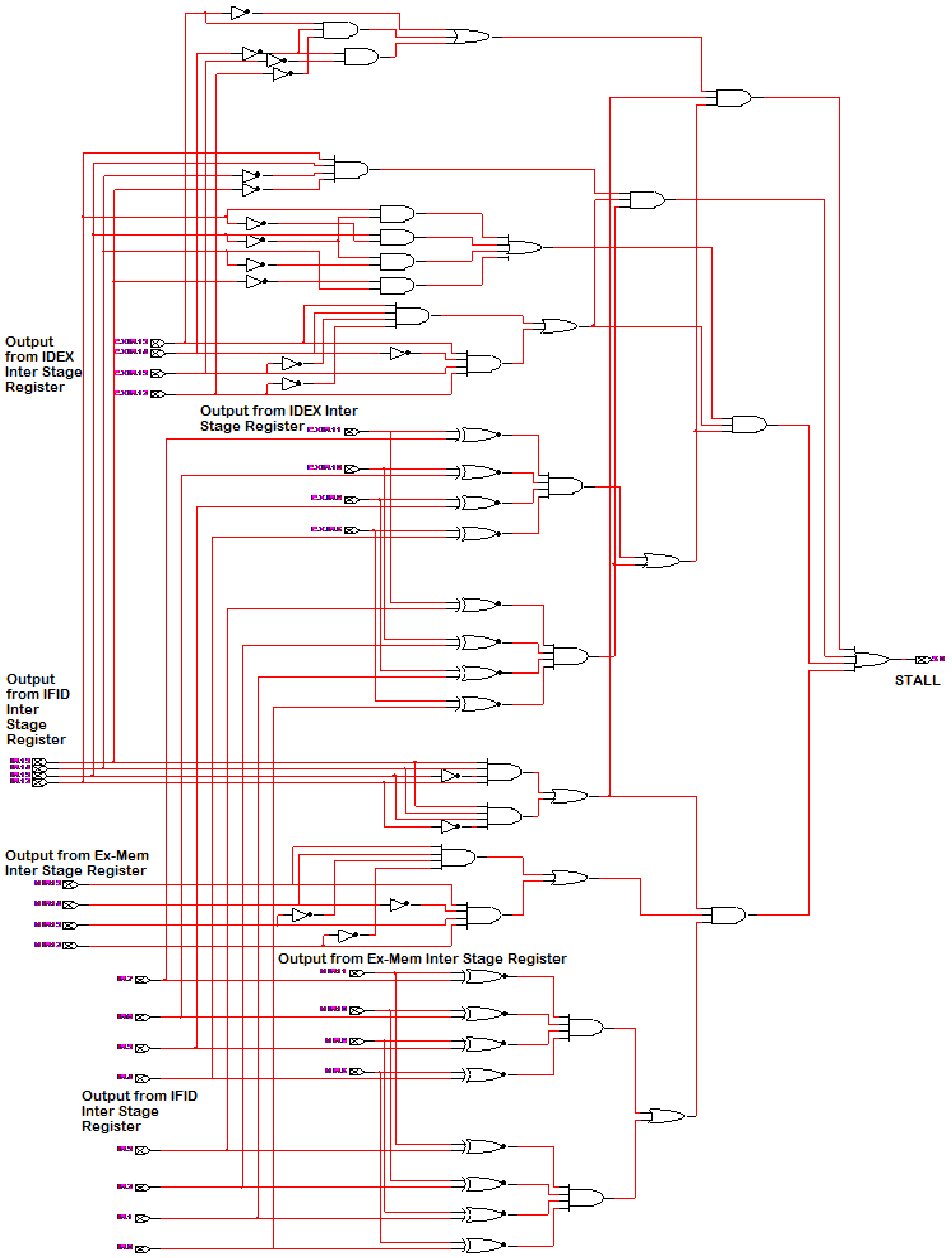

Output from IDEX Inter Stage Register
EXIR15
EXIR14
EXIR13
EXIR12
Output from IDEX Inter Stage Register
Output from IFID Inter Stage Register
IR15
IR14
IR13
IR12
Output from Ex-Mem Inter Stage Register
Output from Ex-Mem Inter Stage Register
Output from IFID Inter Stage Register
STALL

Fig. 4. STALL detection circuit

Further we have considered a second example with control hazard to assure the validity of the processor. (Code II).

```
LOAD  RS1, #3
LOAD  RS2, #1
ADD  RS3, RS1, RS2
BRANCH EQUAL  RS1, RS2, #3
```

In this example, there is a data hazard between the second Load operation and third ALU operation, which will be resolved by inducing a single stall between those instructions. Also, there is a control hazard as there is a branch instruction. This hazard will also be resolved by inducing a single stall before the branch instruction. *(See Fig. 10)*

The figure shows the result obtained after executing the code on a simulated pipelined processor. Here WIR15 to WIR12 shows the operation (Opcode), where WIR15 is the most significant bit and RD7 to RD0 shows the results obtained for each operation, where RD7 is the most significant bit.

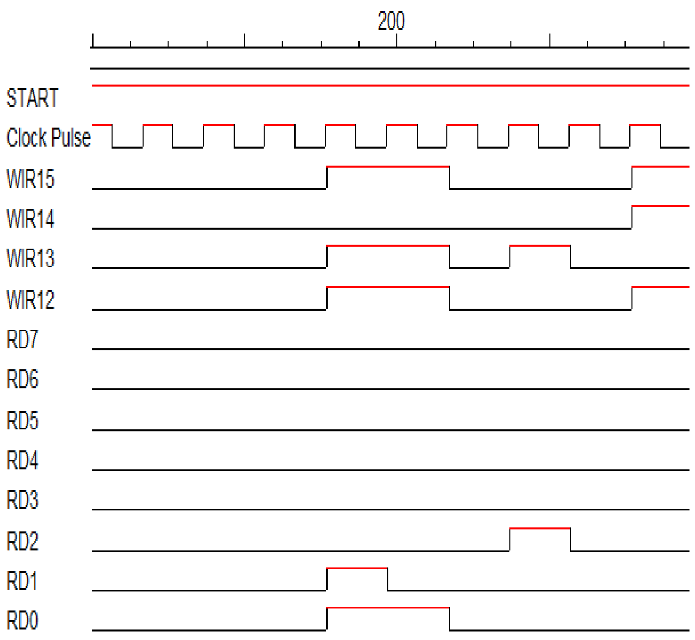


Fig. 10. Output of code II.

IV. Conclusion

In the current study we have designed a working simulation of sixteen bit, five stage pipelined RISC processor that is capable of resolving Data hazards by data forwarding between the stages and handles the Control hazards by inducing stall between the instructions.

References

[1] J.L. Hennessy, and D.A. Patterson, Computer architecture: a quantitative approach. Elsevier: 2011.

[2] J. Lee and A. Smith, Branch prediction strategies and branch target buffer design. In Instruction-level parallel processors . IEEE Computer Society Press.March 1995, pp. 83-99.

[3] Z. Su and M. Zhou, "A comparative analysis of branch prediction schemes." University of California at Berkeley, Computer Architecture Project, 1995.

[4] T. Ball andJ.R. Larus, "Branch prediction for free". ACM, Vol. 28, No. 6, 1993, pp. 300-313.

[5] J.B. Chen, M.D. Smith,C. Young and N. Gloy, "An analysis of dynamic branch prediction schemes on system workloads". 23rd Annual International Symposium on Computer Architecture. IEEE, 1996, May, pp. 12-12.

[6] D.A. Jiménez, and C. Lin, "Dynamic branch prediction with perceptrons. In High-Performance Computer Architecture". The Seventh International Symposium on HPCA. IEEE, 2001, pp. 197-206.

[7] T.Y. Yeh, and Y.N. Patt, "Two-level adaptive training branch prediction". In Proceedings of the 24th annual international symposium on Microarchitecture. ACM,1991, September, pp. 51-61.

[8] C.C. Cheng, "The schemes and performances of dynamic branch predictors".2000.