Available at www.ijsred.com

RESEARCH ARTICLE OPEN ACCESS

Implementation of a Pipelined Core with Hazard Detection and Data Forwarding Circuitry

Sowdamini Jayanthi[1]

[1]Student, Department of VLSI Engineering, The Veda Educational Society, MRP Towers, Vidyanagar 1st Lane, Guntur, Andhra Pradesh, India.

sowdaminivit@gmail.com

--************************----------------------------------

Abstract:

Performance is the main constraint for any processor. Having number of features added, but if it takes more time to execute, then it's a drawback. So, pipelining is a concept that is employed to reduce the overall CPU time and increase the throughput of the core. Pipelining is the technique where more number of instructions is overlapped and gets executed simultaneously. In this process, there are chances to get various types of hazards, while the instructions are traversing through pipeline stages. It is necessary to observe these hazards and provide solution by either stalling or forwarding the data when it is available. So, hazard detection unit is designed to check for occurrence of hazards and provide stall cycles when necessary. Data forwarding unit provides the necessary data to the next stages of pipeline even before the updating the actual location.

Keywords --- **ISA – Instruction Set Architecture, HDL – Hardware Description Language, MUX – Multiplexer.**

--************************----------------------------------

I. INTRODUCTION

Performance of a computer system depends on many factors and can be defined as how efficiently it produces results for an operation. The main factors that affect performance are instruction count, cycles per instruction and cycle time. Instruction set architecture (ISA) and compiler actually determines the number of instructions required i.e., instruction count, whereas processor implementation decides the clock cycle time and cycles required per instruction. Performance is inversely proportional to the execution time. For the improvement of these features concept of pipelining is introduced in the implementation of processor. Pipelining reduces the execution time, so performance gets improved.

Pipelining allows the instructions to be executed in parallel. This is made possible by dividing the entire operation into different stages and by building separate hardware for each of these stages, so that more instructions can be operated at the same time in different stages. All these stages are connected to form a pipe like structure. Number of stages in a pipeline is decided by the resources available and also by the possibility to split the operation between the stages.

By using the concept of pipelining in processor, we can get output of one instruction for each clock cycle (happens only after the completion of operation of first instruction). Thereby, increasing the overall throughput of the computer system. So, performance of the system increases. If there is more number of stages, more number of instructions can be executed in parallel and performance is improved. But, there is a limit for the division of stages based on operation. The Fig.1 shows the improvement in CPU time by use of pipelining.

 ©IJSRED: All Rights are Reserved

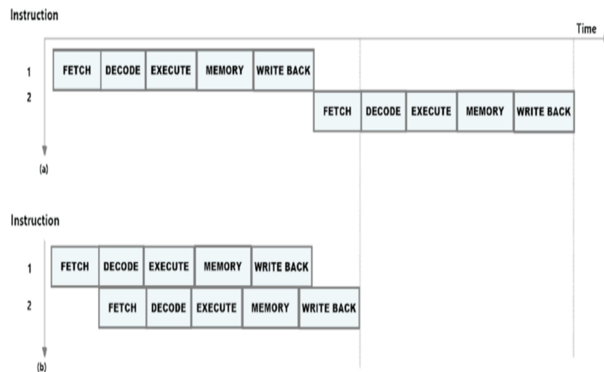

Fig.1 Comparison of processor operation with and without pipeline

The figure consists of five stages which are fetching, decoding, executing, data memory and write back. The figure shows both pipelined and non-pipelined example. Let us assume 1 clock cycle time is required for one stage. Then, CPU time for the processor without pipeline for completing 2 instructions operation is 10 clock cycles and CPU time for the processor with pipeline for the same operation requires only 6 clock cycles. So, overall time is reduced and performance is improved.

In this paper, designing of a pipelined core using Verilog HDL is proposed and this core also included with hazard detection and data forwarding circuitry for much more efficiency.

II. DESIGN ARCHITECTURE

There are five stages of pipeline, i.e., fetch, decode, execute, data memory and write back. Each of these stages require one clock cycle time for their completion and completing the entire execution of instruction in maximum 5 clock cycles assuming that no hazards that cause stalls are present. The functionality of each of these stages, their internal circuit details and their operation are explained below in this section.

Instruction Fetch Stage: In this stage, instruction is fetched from the instruction memory using the address given by the PC (program counter). The first block in the Fig.2 shows adder blocks which is used to generate the next address by adding 4 to the current address. This address is given to the multiplexer which is used to select between branch address and normal next address.

This selection is done by the PCSrc signal which is made high when there is branch instruction and a zero output from ALU. Branches require the use of the ALU output to determine the next instruction address, which comes either from the adder (where the PC and branch offset are summed) or from an adder that increments the current PC by four.

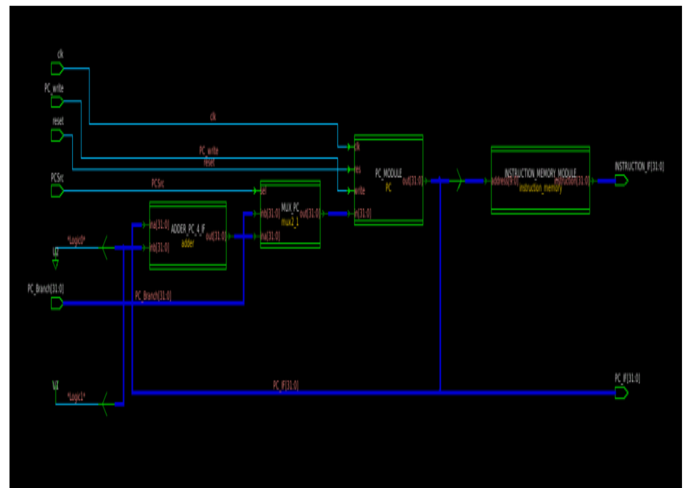

Fig.2 Fetch Block

Instruction Decode Stage: In this stage, instruction is divided into parts to get the register addresses or immediate values or branch addresses or offsets and is given to register file module, from where the operands are read as shown in Fig.3. Register file is a state element that consists of a set of registers that can be read and written by supplying a register number to be accessed. The register file contains the register state of the computer. R-format instructions have three register operands, so two data words are to be read from the register file and write one data word into the register file for each instruction.

 ©IJSRED: All Rights are Reserved

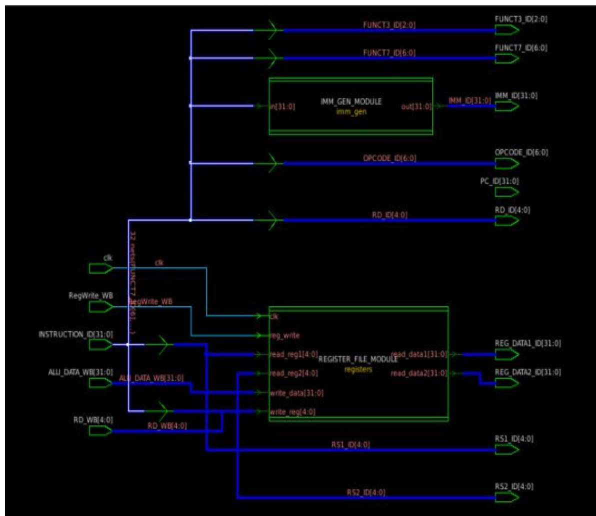

Fig.3 Decode Block

For load register and store register instructions, which have the general form ld x1, offset(x2) or sd x1, offset(x2). These instructions compute a memory address by adding the base register, which is x2, to the 12-bit signed offset field contained in the instruction. If the instruction is a store, the value to be stored must also be read from the register file where it resides in x1. If the instruction is a load, the value read from memory must be written into the register file in the specified register, which is x1.

Execute Stage: All instruction classes use the arithmetic-logical unit (ALU) after reading the registers. The memory-reference instructions use the ALU for an address calculation, the arithmetic logical instructions for the operation execution, and conditional branches for the equality test.

In Fig.4, there are three multiplexers used in this stage, out of which two are 4:1 MUX's for selecting the two operands of the execution unit. These operands may be from direct register file contents (for normal operation) or maybe from pipeline registers either from ex_mem stage or mem_wb stage (for data forwarding case), so these two MUX outputs provides the data operands. There is one 2:1 MUX used for selecting the second operand to be a data from register file or from immediate formed in the decode stage.

There is specific adder used for branch address calculation and ALU unit is used for performing all other operations. ALU_control is a special block which generates a 4-bit operation code for ALU. This is formed by classifying the type of operation to be performed.

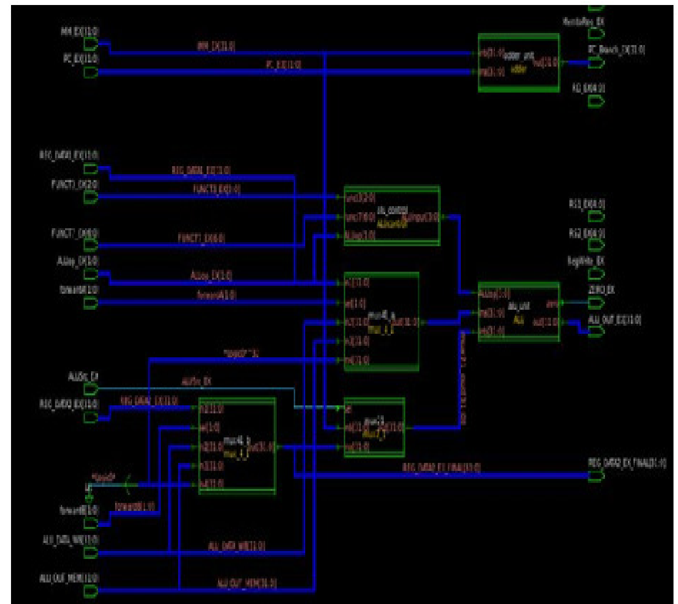
Fig.4 Execute Block

Data Memory Stage: In this stage, there is one memory which is called as data memory and each word is of 32-bit wide. The operation performed here is either reading or writing from/to memory respectively. This stage is required only for memory access instructions like load and store, the address from which data need to be taken or written is provided from the previous ALU stage output.

Write Back Stage: This stage is present for instructions which need the register to be updated like after add, sub, load, or, nor, etc.,(all arithmetic and logical operations need this stage). Here, MUX is present to select the data from memory stage for load operation or from the execute stage for all other instructions that require data to be written back to the register file and the selection is done by the Mem_to_Reg control signal, which is generated by the control unit.

 ©IJSRED: All Rights are Reserved

Control Unit: This unit is responsible for generating all the control signals like Mem_to_Reg, MemRead, MemWrite, etc.,

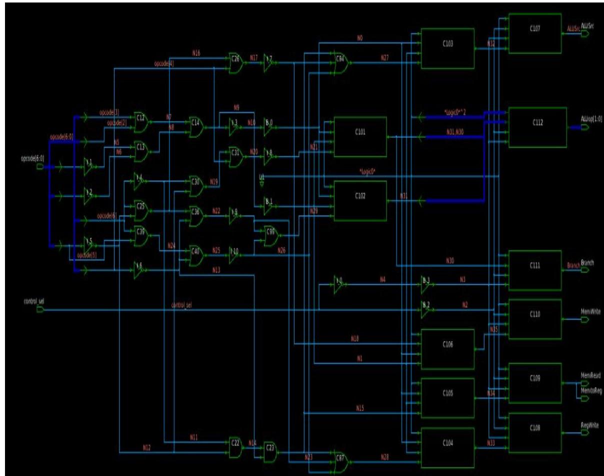

Fig.5 Control Unit

For generating these control signals, it makes use of opcode that is obtained from the instruction and control_sel signal obtained from the hazard detection unit. When there is a chance of hazard, then pipeline is stalled to avoid fetching and executing of invalid instructions, it is notified to control unit by control_sel signal and hence control signals are generated as all bits with 0's irrespective of the opcode.

Hazard Detection Unit: This circuit inserts a stall cycle for halting the pipeline from fetching next instruction. This is done to avoid flushing the pipeline later. It uses the current source register addresses and past destination register address and the MemRead control signal for generating the signals indicating hazard. When the previous operation is load to a specific register and the current instruction has the same register in its source position, then 1 stall cycle is required, since the value present in that register is obtained only after memory stage(4th cycle) but the current instruction requires that data at least by execute stage (3rd cycle).

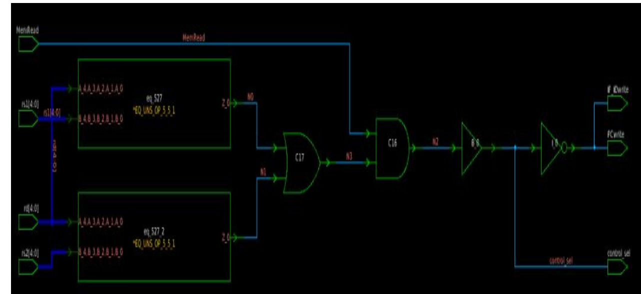

Fig.6 Hazard Detection Unit

Data Forwarding Unit: This is used to supply the data by using temporary pipeline registers for the next instructions even before the actual register update, so that improper output can be avoided.

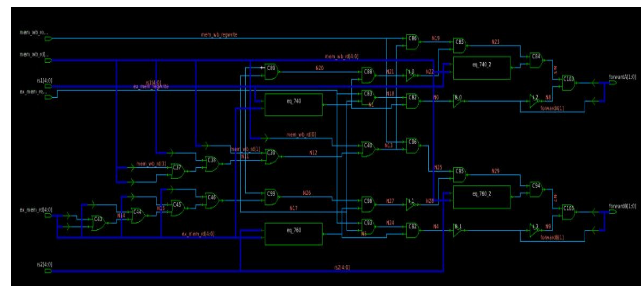

Fig.7 Data Forwarding Unit

This unit uses current source registers and destination register at memory and write back stages to generate forward control signals which are given to the 4:1 MUX's of execute stage as selection signals. Two forward signals forwardA and forwardB are generated depending on destination same as source1 or source2 respectively. These are 2-bit signals and combinations re used as follows:

2'b00 => corresponding operands from register file.

2'b01 => corresponding operands from pipeline register between memory and write back stage..

2'b10 => corresponding operands from pipeline register between execute and memory stage .

 ©IJSRED: All Rights are Reserved

2’b11 => corresponding operands are 32’d0.

III. RESULTS AND WAVEFORMS

This project is tested by applying some instructions and simulating it using Synopsys VCS tool and obtained the waveforms that are presented here.

Set of Instructions Applied: The instructions used are very limited and are used to sample all the required outputs.

```
00008133  //add x2,x1,x0
00108093  //addi x1,x1,1
0020F1B3  //and x3,x1,x2
0010E213  //ori x4,x1,1
0042A223  //sw x4,4(x5)
00802603  //lw x12,8(x0)
00160633  //add x12,x12,x1(hazard)
04090E63  //beq x18,x0,5c
0042A583  //lw x11,4(x5)
```

Waveforms: In Fig.8, it can be seen that first instruction starts fetching in cycle1 and completes its operation in 5 clock cycles itself.

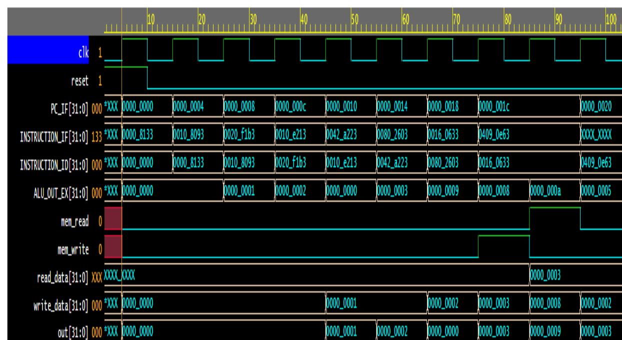

Fig.8 Waveform to show number of cycles required for a normal operation.

In Fig.9, hazard is detected by hazard detection unit in 8th clock cycle, indicated by making control_sel signal high and also pipeline_stall is made high and the INSTRCUTION_ID signal indicates that pipeline is stalled by staying in the same instruction for 2 cycles.

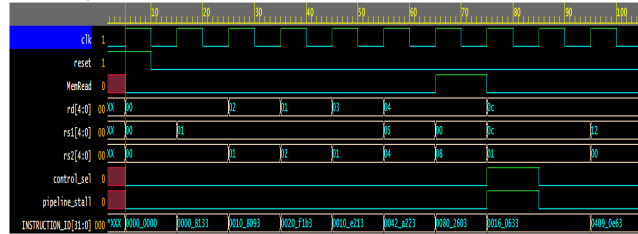

Fig.9 Waveform showing hazard is detected.

In Fig.10, waveform showing data is forwarded by forwarding unit is shown. It can be seen in 5th cycle rs1 and destination register at memory stage and rs2 and destination register at write back stage are equal, which results in generation of forwardA and forwardB signals.

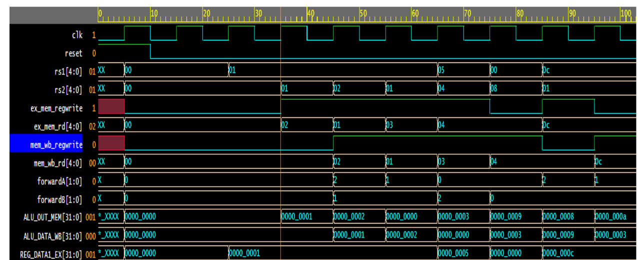

Fig.10 Waveform showing that data is forwarded.

Table 1 shows the various properties of pielined and non-pipelined core.

	Without pipeline	With pipeline and without forwarding and hazard circuitry	With hazard detection and forwarding circuitry
Number of instructions executed	9	9	9
Clock cycles required	45	14	14
Chances of obtaining invalid data	No	Yes	No

Table 1 Differences between pipelined and non-pipelined core

IV CONCLUSION

This project describes the Verilog modelling of a 32-bit pipelined core with hazard detection and data forwarding circuitry. Some set of instructions are

 ©IJSRED: All Rights are Reserved

given such that functionality of every unit could be covered and produced the simulated waveforms here. Comparisons drawn between pipelined and non-pipelined cores are also mentioned. Number of clock cycles required are less for a pipelined processor which makes its execution time lesser, thereby increasing performance (Performance is inversely proportional to execution time).

ACKNOWLEDGMENT

I express my sincere thanks to Industry guide Mr. Murali Mohan Kilari, SMTS, Invecas Pvt. Ltd, and VEDA IIT guide Mr. Srinivasa Rao Pokuri, Assistant Professor, VEDA IIT, Amaravati for helping me to carry out the Project Work.

I convey my deep sense of gratitude to Prof. K.Malakondaiah, Director, VEDAIIT Amaravati, and Mrs. Madhuri Nallapaneni, Director, Invecas, Guntur, for providing me with an opportunity for carrying out this Project Work.

REFERENCES

[1] "Computer Organization and Design –RISC-V Edition" by David A Patterson and John L. Hennessy.
[2] "Study of data hazard and control hazard resolution techniques in a simulated five stage pipelined RISC processor" available at https://ieeexplore.ieee.org/document/7824864.
[3] Samir Palnitkar, "Verilog HDL: A Guide to Digital Design and Synthesis", Second Edition.
[4] "Simulating a pipelined RISC processor" available at https://ieeexplore.ieee.org/document/7824854
[5] "Design and Implementation of MIPS Processor" available at http://ijsrd.com/Article.php?manuscript=IJSRDV6I60007
[6] http://www.testbench.co.in.
[7] http://www.doulos.com/knowhow/verilog
[8] "MIPS-A-Microprocessor Architecture" article available at https://os.ecci.ucr.ac.cr/ci1310/articulos/Mips/Hennessy.pdf
[9] "Design & Analysis of 16 bit RISC Processor Using low Power Pipelining" available at http://ieeexplore.ieee.org/document/7148575/
[10] " Review on 32-bit MIPS RISC Processor using VHDL" available at https://www.semanticscholar.org/paper/Review-on-32-bit-MIPS-RISC-Processor-using-VHDL-Ritpurkar-Thakare/9634c0b31be5490319a2aa0adb0d0ffd6d6ced12
[11] Reference manual of design compiler tool from Synopsys.

 ©IJSRED: All Rights are Reserved