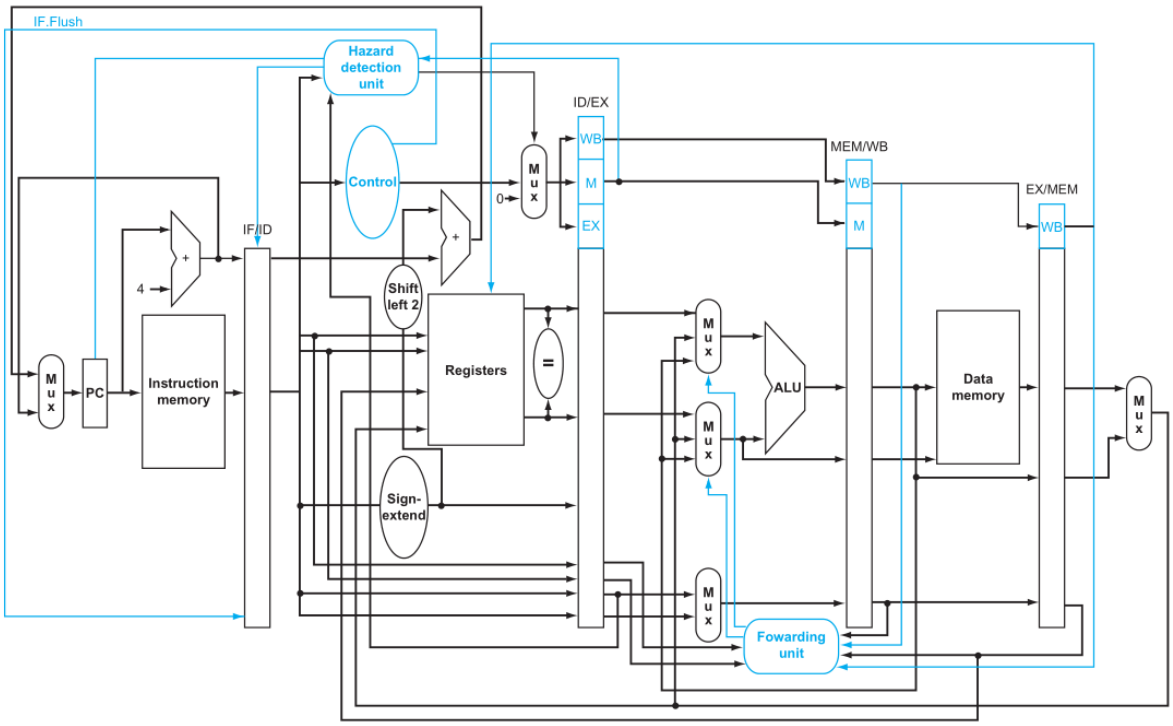
PIPELINED IMPLEMENTATION OF MIPS ARCHITECTURE

Submitted by
Amit Kshirsagar
2018AAPS0381G

The design of the processor implemented is based on the following diagram. (modifications have been made as per need.)



Modules Build:

1. **Instruction_Fetch –** This module takes Program counter (PC) and Reset as the input, and gives the Instruction fetched as the output. This module has the Instruction Memory that is of 1024 bits size. The instruction memory is an array consisting of 128 eight bits register used to store 32 bit instruction. The 32 bit instructions as stored in consecutive instruction memory locations at an offset of 4 bytes (since 4bytes = 32 bits). When Reset equals 0, the instruction memory is written by some default instruction that are supposed to be executed. Whenever PC gets updated a new instruction is fetched and given as output.

2. **PC_Adder1 –** This module calculates PC+4.

3. **Mux_PCSrc –** This module works like a Mux and selects the value that is to be updated in PC, it decides the appropriate value of PC on the basis of the PCSrc control line.
   If PCSrc control line is 1 then PC is updated with the value from the adder used for branched instructions.
   If PCSrc control line is 0 then PC is updated by PC+4 value, for non-branch instructions.

4. **Control_Unit –** This is the control unit of the main processor that generates all the control lines. This module takes the 6bit opcode (Instruction[31:26]), and depending upon the task the control unit generates the following control lines:
   i. **RegDst –** This control line decides which instruction field corresponds to the destination register, because the destination register is specified by different instruction fields in case of R-format and Load word instructions.
   RegDst = 1 means destination reg is specified by Instruction[15:11]
   RegDst = 0 means destination reg is specified by Instruction[20:16]

   ii. **Branch –** This control line is used to select proper value for the Program counter (PC) in case of BEQ and BNE instruction.
   Branch = 1 means the Instruction is a branch instruction.
   Branch = 0 means the Instruction is not a branch instruction.

   iii. **MemRead –** This control line tells whether any data is being read from the Data Memory or not.
   MemRead = 1 means data will be read from the data memory.
   MemRead = 0 means data will not be read from the data memory.

   iv. **MemtoReg –** This control line tells which data will be written in the register file.
   MemtoReg = 1 means the data read from Data Memory will be written in register file.
   MemtoReg = 0 means the data read result from ALU Operation will be written in register file.

   v. **MemWrite –** This control line tells whether any data will be written in the Data Memory or not.
   MemWrite = 1 means data will be written into the data memory.
   MemWrite = 0 means data will not be written into the data memory.

   vi. **RegWrite –** This control line tells whether any data will be written in the Register file or not.
   RegWrite = 1 means data will be written in the register file.

RegWrite = 0 means data will not be written in the register file.

vii. **ALUOp –** This control line is used to generate Alu_control that tells the ALU what function is to be performed. The different ALUOp codes used are as follows:
   1. **00-** LW, SW
   2. **01-** BEQ
   3. **10-** R-type add, sub, and, or, slt

viii. **ALUSrc –** This control line decide what is the 2nd input of the ALU.
ALUSrc = 1 means sign extended instruction is the input (in case of LW, SW instruction).
ALUSrc = 0 means data Read from 2nd source register is the input.

ix. **IF_Flush –** This control line flushes the IF/ID pipeline registers when a branch is being taken.
IF_Flush = 1 means branch is not being taken and the IF/ID pipeline register need not be flushed
IF_Flush = 0 means branch is being taken and the IF/ID pipeline register need to be flushed

5. **Hazard_detection_unit –** This module represents the hazard detection unit used for testing the load use hazard. This module checks the following hazard detection condition:

```
if (ID/EX.MemRead and
   ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
     (ID/EX.RegisterRt = IF/ID.RegisterRt)))
     stall the pipeline
```

When the condition is true it makes the value of the control_line, IF_ID_Write, and PCWrite to be 0. These are then used to create a stall in the case when a hazard is detected.

i. **Control_line –** This wire is give as the input to the mux which takes the input from the control unit. In case there is a hazard the output of the Mux is given as 0.(all control signals made 0).
Control_line = 0, means hazard has occurred.
Control_line = 1, means no hazard has occurred.

ii. **IF_ID_Write –** This wire is used to preserve the values of the IF/ID pipeline registers in case a hazard is detected.
IF_ID_Write = 0, means hazard has occurred.
IF_ID_Write = 1, means no hazard has occurred.

iii. **PCWrite –** This wire is used to preserve the value of PC in case of a hazard.
PCWrite = 0, means hazard has occurred.
PCWrite = 1, means no hazard has occurred.

6. **Mux_Control –** This module does the work of the Mux. It takes all the control signals as the input along with the control_line. When a hazard has occurred control_line is 0 and thus the output of Mux is 0, i.e. all the control lines are made 0, for nop. When there is no hazard detected control_line is 1 and the output of the Mux is the original values of control lines generated by control unit.

7. **Register_file –** This module takes in the control signal RegWrite, and depending on it value data may be written in the register file in the register specified by Write_Reg. This module also reads the data from the source registers present in the register file that is specified by the different instruction fields that are given in input as Read_Reg_1, Read_Reg_2. This module also has 1024 bits of memory that is used as register file. There are in total 32 registers of 32 bits size each, out of which some of them have specific tasks.
   0^{th} reg is the $zero register that is hard wired to ground.
   8^{th} – 14^{th} are the temporary registers.
   16^{th} – 23^{rd} correspond to $s registers
   The register file unit also outputs a Zero control signal, in case when both the data read from the register file is equal/same. This Zero control line is used for executing Branch instruction, and moving the branch instruction in the ID stage.

8. **Sign_Extend –** This module takes in 16 bits of address specified in the instruction in case of BEQ and sign extends it to 32 bit address.

9. **Shift_left_2 –** This module takes the 32 bit address and shifts it left by 2 i.e. multiplies the address by 4.

10. **PC_Adder_2 –** This module takes PC+4 as one of the input and then adds the left shifted value to it. This value is used for updating the PC in case when a branch is taken.

11. **PCSrc_And –** This module performs logical AND operation on Branch and Zero line, and if the output is 1 it means that the BEQ condition is true and the PC must be changed according to the offset given in instruction.

12. **Forwarding_unit –** This module represents the forwarding unit used for forwarding the data in case of data hazard. This module checks the following condition for data hazard:

Type 1 hazard

```
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
```

Type 2 hazard

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠  0)
and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠  0)
       and (EX/MEM.RegisterRd ≠  ID/EX.RegisterRs))
and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠  0)
and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠  0)
       and (EX/MEM.RegisterRd ≠  ID/EX.RegisterRt))
and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01
```

The forwarding unit sets the forwardA, and forwardB values according to the above condition.

13. **Mux_forwardA and Mux_forwardB –** This module represent the Mux added for forwarding. Depending upon the values of forwardA, and forwardB control lines the output of this Mux is decided. The table tells how the output is decided on the basis of forward control lines:

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

14. **Mux_ALUSrc –** This module takes in ALUSrc control line and decides the 2nd input of ALU. if ALUSrc = 1, output is sign extended value, if ALUSrc = 0, the output is Data read from the forwardB Mux.

15. **ALU_Control –** This module generates the ALU control line, that helps the ALU decide what type of arithmetic is to be implemented.
ALU_control_op = 0 → and operation

ALU_control_op = 1 → or operation
ALU_control_op = 2 → add operation
ALU_control_op = 6 → sub operation
ALU_control_op = 7 → set on less than (slt) operation

16. **ALU –** This module takes in the ALU_control_op generated by the ALU_Control and performs the arithmetic specified by ALU_control_op.

17. **Mux_RegDst –** This module takes in instruction fields as input and depending upon the value of RegDst give the output that corresponds to the destination register.

18. **Data_Memory –** This module takes in the MemWrite and MemRead control lines and depending on their values the ALU_Result may be written in Data Memory, and the data may be read from the Data Memory. The Data Memory has 4096 bits i.e. 4Kilo bits of memory, that is created using 512 registers of 8 bits reach.

19. **Mux_MemtoReg –** This module give data as output that will be written in the register file. It decides the output by using the MemtoReg control line.

20. **MIPS_Pipelined_Processor –** This is the main module where all the above mentioned modules have been called. This module takes Clk and Reset as input. Whenever Reset = 0 the processor is brought back to its default by loading the instructions in the instruction memory, and by making PC =00. After the Reset has been set the value of PC is updated at the positive edge of the input clock, which makes sure that only 1 instruction is fetched in one clock cycle. At the end of each clock cycle the new value of PC is calculated that will be used to fetch the next instruction in queue.
    All Pipeline register have been defined whenever their need has arised, and all the pipeline register are updated in the always block on the positive edge, this makes sure that 5 different stages are maintained, and the processor has 5 pipeline stages.
    The different control lines generated by hazard detection unit have also been used here. These lines include the PCWrite, IF_ID_Write. The IF_Flush is also checked for flushing the values of IF/ID values in case when a branch is taken.

Assumptions:

1. Instructions to be fetched are preloaded in the instruction memory.
2. The size of DataMemory is assumed to be 4096 bits.
3. The size of Instruction Memory is assumed to be 1024 bits.
4. It is assumed that only 32 registers are there.
5. Fowarding unit has been added to detect the following data hazard:
    a. EXE/MEM and ID/EXE for both Rt and Rs registers.
    b. MEM/WB and ID/EXE for both Rt and Rs registers.
6. Hazard detection unit has been added to detect hazard in case of load instruction and the next instruction having data dependency, and a stall is added in case of a hazard.
7. The Branch instruction implementation has been optimised by checking the branch condition in ID stage. By this only 1 cycle is stalled in case when a branch is taken.
8. No separate/additional forwarding unit has been added for handing the data hazard in case of branch instruction.
9. The processor designed for this submission can implement only the following 8 instructions :-

Add	Sub	And	Or
Beq	Lw	Sw	Slt

Note: Any other instruction if tried to implement may or may not work properly, some hazard condition may occur which might not have been tested.

TestBench:

The following testbench has been used:

```
module test;
   // Inputs
   reg Clk;
   reg Reset;

   // Instantiate the Unit Under Test (UUT)
   MIPS_Pipelined_Processor uut (
      .Clk(Clk),
      .Reset(Reset)
   );

   initial begin
      Clk = 0;
      Reset = 0;
      forever #10 Clk = ~Clk;
   end
   initial begin
      #20 Reset = 1;// nothing executed until now as reset was equal to 0 before
      #10       // only 1 clock cycle completed until now, PC is taken to be 0,instruction is fetched
      #20       //2nd clock cycle, ID stage, PC=4
      #20       //3rd clock cycle, EX stage, PC=8
      #20       //4th clock cycle, MEM stage PC=12
      #20       //5th, last clock cycle, WB stage
      //5 clock cycle ends first instructions is executed
      #20   // after 6 clock cycles first 2 instructions are executed
      //#20 // after 7 clock cycles first 3 instructions are executed

      $dumpfile("dump.vcd");
      $dumpvars;
      $finish;
   end
endmodule
```

For first 20ns Reset = 0 so that the processor is initialized to its base conditions.

After that when the simulation is carried for 10ns i.e. only 1 clock cycle, only the value of PC will be fetched and instruction will be fetched. The rest of the pipeline registers are 0 as they are in different stage.

After another 20ns seconds, PC will be updated and new instruction will be fetched in IF stage, while the ID stage will be executed for the instruction fetched in previous clock cycle. The rest of the pipeline registers are 0 as they are in different stage.

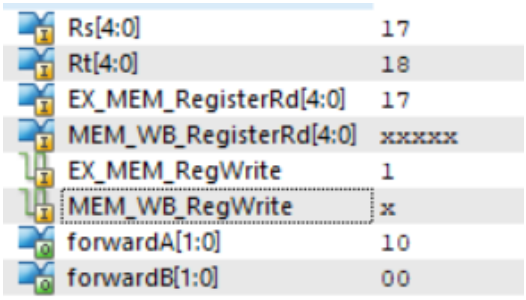
The simulation time can be increased so that all the instructions in the queue(stored in the instruction memory) can be executed. Since it is pipelined implementation of MIPS architecture the CLK period is chosen such that the longest stage is executed within that time period. So 20ns has been chosen as the time period.

Instructions Implemented and their simulation outcome:

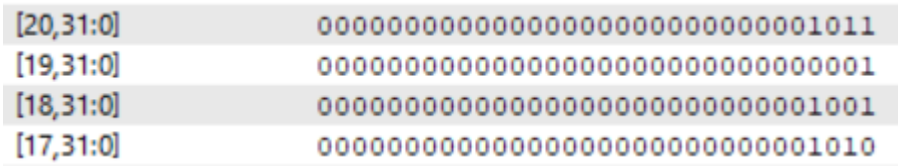
- All the instructions implemented here have been included in the instruction memory, and can be checked from the MIPS_Pipelined_Processor.v file
- All the values mentioned for the register before the instructions are mentioned in decimal

1. **Type 1 hazard**
   Add s1 s2 s3
   Or s4 s1 s2
   Values before Instruction execution: s1=0, s2=9, s3=1,s4=0
   running it for 6 clock cycles.

   Values of forwarding unit.



   Register file after instruction execution



2. **Type 2 hazard**
   Add s1,s2,s3
   Sub s5,s2,s4
   Sub s6,s1,s2
   Values before Instruction execution: s1=0, s2=9, s3=1, s4=1, s5=0,s6=0
   Running it for 7 clock cycles

Values of forwarding unit.

Rs[4:0]	17
Rt[4:0]	18
EX_MEM_RegisterRd[4:0]	21
MEM_WB_RegisterRd[4:0]	17
EX_MEM_RegWrite	1
MEM_WB_RegWrite	1
forwardA[1:0]	01
forwardB[1:0]	00

Register file after instruction execution

[22,31:0]	00000000000000000000000000000001
[21,31:0]	00000000000000000000000000001000
[20,31:0]	00000000000000000000000000000001
[19,31:0]	00000000000000000000000000000001
[18,31:0]	00000000000000000000000000001001
[17,31:0]	00000000000000000000000000001010

3. **Type 1 and 2 hazard**

Add s1 s2 s3
Or s4 s1 s2
Sub s5,s2,s1
Values before Instruction execution: s1=0, s2=9, s3=1, s4=1, s5=0
Running it for 7 clock cycles

Values of forwarding units.

Rs[4:0]	17
Rt[4:0]	18
EX_MEM_RegisterRd[4:0]	17
MEM_WB_RegisterRd[4:0]	xxxxx
EX_MEM_RegWrite	1
MEM_WB_RegWrite	x
forwardA[1:0]	10
forwardB[1:0]	00

Rs[4:0]	18
Rt[4:0]	17
EX_MEM_RegisterRd[4:0]	20
MEM_WB_RegisterRd[4:0]	17
EX_MEM_RegWrite	1
MEM_WB_RegWrite	1
forwardA[1:0]	00
forwardB[1:0]	01

Register file after instruction execution

[21,31:0]	11111111111111111111111111111111
[20,31:0]	00000000000000000000000000001011
[19,31:0]	00000000000000000000000000000001
[18,31:0]	00000000000000000000000000001001
[17,31:0]	00000000000000000000000000001010

4. **Hazard detection for load instruction**

Lw s1 0x2(s2)
Or s4,s1,s2
Values before Instruction execution: s1=0, s2=9, s3=0, s4=0
data in mem=0x08080808
Running it for 7 clock cycles (1 cycles is stalled due to data hazard)

Values of hazard detection unit.

IF_ID_RegisterRs[4:0]	17
IF_ID_RegisterRt[4:0]	18
ID_EX_RegisterRt[4:0]	17
ID_EX_MemRead	1
control_line	0
IF_ID_Write	0
PCWrite	0

Values of forwarding unit.

Rs[4:0]	17
Rt[4:0]	18
EX_MEM_RegisterRd[4:0]	17
MEM_WB_RegisterRd[4:0]	xxxxx
EX_MEM_RegWrite	1
MEM_WB_RegWrite	x
forwardA[1:0]	10
forwardB[1:0]	00

Register file after instruction execution

[20,31:0]	00001000000010000000100000001001
[19,31:0]	00000000000000000000000000000000
[18,31:0]	00000000000000000000000000001001
[17,31:0]	00001000000010000000100000001000