RISC SIX Stage Pipelined Implementation

Design Report

SELF-PROJECT

Submitted by

PRIYANSH KASYAP (213070114)

DATA PATH COMPONENTS

Instruction Fetch

PC	**Stores the PC value of the current instruction**
IM	**Instruction Memory from which the instruction is obtained.**
PC++	**It increments the PC value by 1 for the next possible instruction.**
Brach Prediction Table Block	**Store the already encountered BEQ instruction PCs and the branch address along with the only more recent history of the instruction (whether it was taken or not)**
MUX	**Chooses between the taken address given by the branch prediction table and PC++. The control signal is also generated by the table block itself**

Instruction Decode

SE	It extends the sign of 6-bit or 9-bit immediate data to 16 bit. It also observes the extra LLI bit in which case it only packs the immediate data by zeros.
SE+PC	It adds the SE value and the PC value of the current instruction.
Decoder	It gives the control signals according to the current instruction.
MUX	Steers input between PC+1 and (SE+PC) value. Chooses (SE+PC) when JAL instruction.

Register Read

LS	Left shifts the sign extended value.
RF	Register file which contains the register (R0-R7).
MUX1	It steers inputs between (SE+PC) and LS. SE+PC is chosen during JAL instruction and LS is chosen during LHI instruction.
MUX2	The mux before AR2. It steers inputs between AR2 from IDRR register and the output from LM_SM block.
MUX3	The mux after DO1. It steers inputs between DO1 and the ALU output which gives the DO+1 value during LM_SM instruction.
Hazard Mux	Controlled by Hazard_RR.
Hazard_RR	The Hazard Block present in RR stage. Sends Input to the PC register according to the instruction. a. SE during LLI and R7 is the destination. b. DO1 during the JLR instruction.
LM_SM mux	Choses the input for the LM_SM block between the data after decoder or from the ID_RR pipeline.
LM_SM block	It operates the LM and SM operation.

Execute

ALU	Perform ADD, NAND and Comparator operation.
Flags	Contains the carry, zeros and overflow flags.
Forwarding Blocks	Checks if dependency present between RR_EX pipeline and the pipeline registers in the further stages. It accordingly controls the forward logic muxes.
MUX1	The second mux before ALU second input. It steers inputs according to the arithmetic operation or address calculation.
MUX2	The mux just before the second input of ALU. Controlled by the LM_SM block. Sends 1 during the LM or SM operation to increment the address.
Hazard Mux	Mux controlled by the hazard logic block in execution stage.
Staller	It stalls the pipeline registers when there is an immediate dependency after the load instruction.
Hazard_EX	Controls the hazard when during arithmetic instruction the destination is R7. It loads the PC with the required value controlling the hazard mux. Flushes the pipeline.

Memory access

MEM	Data memory from which the instruction read data or writes data.
MUX1	Mux before address of data memory. Steers inputs between ALU output (address calculation) or DO1 (during LM or SM operation).
Hazard MUX	Mux being controlled by the hazard block in MW stage.
Hazard_MM	Checks the hazard during load instruction when the destination is R7. It suitably loads the PC value by controlling the hazard mux.

Write Back

Flags_user	The flags which are visible to user.
WB_mux	It decides the input to be written in the register file. The inputs according to the instructions are a. ALU output : During arithmetic instructions. b. LS_PC : LHI instruction. c. SE : LLI instruction. d. PC+1 : JAL, JLR instruction. e. Mem_out : LW, LM instruction.
R7_mux	It decides the input to the R7 in the register file. The inputs according to the instructions are a. DO1 : During JLR instruction. b. PC+1 : Normal Program Flow. c. LS_PC : For BEQ instruction when the branch is taken.
Conditional and Hazard Control	Takes care of the following cases a. Conditional arithmetic instruction is not taken and dependency present in previous pipelineregisters. b. Conditional arithmetic instruction when the destination is R7.

	c. JLR instruction when the destination is R7. d. JAL instruction when the destination is R7. e. Check for BEQ instruction if the branch is taken or not. f. Flushes the pipeline if any of the above condition is true.
Hazard Mux	Controlled by the above logic block. Decides the input to PC register. a. (SE+PC) when branch is taken in BEQ. b. PC+1 when JLR and JAL hazard is seen. c. Otherwise, the default value coming at 0 input.

CONTROL SIGNALS FLOW

Instruction Fetch

1	**PC is incremented and sent to PC register**
2	**Instruction is fetched from Memory**
3	**Branch Prediction Table (BPT) is used if possible**

Instruction Decode

Creation of Signals:

1.	**Control Signals for multiplexors in the other pipeline stages**
2.	**Sign Extended value of the Immediate data in the instruction if present**
3.	**The addition of the sign extended value and the current PC value (SE + PC)**
4.	**Control line for the mux which decides if PC+1 must be sent or (SE+PC) value to the PC register (SE+PC)**
5.	**value is sent to PC when decoder encounters with JAL instruction**
6.	**Clear bit for when JAL instruction arrives to clear the pipeline register (IF-ID)**
7.	**Control Signal for SE which decides if to sign extend 6 bit or 9-bit immediate data**
8.	**LLI bit for SE which just packs the immediate data with zeros to make the data 16bit**
9.	**Address of the register file (AR1, AR2, AR3)**
10.	**The LM or SM data to be given to the LM_SM block This contains the data of the registers whose data must be either stored or loaded**
11.	**The ALU control bits which tells the ALU of its operations**
12.	**Flag control bits which enable or disables the flag registers**
13.	**The condition bits which indicate the presence of conditional arithmetic instruction**
14.	**Register write and Memory write signals**
15.	**Control Signal for the BPT telling it whether the instruction is BEQ type or not**
16.	**Branch Prediction Table produces the index of the instruction from the table and information on whether the branch was taken or not**

Register Read

1. **Consumption of Signals:**

1.	The LM or SM data is consumed by the LM_SM block and also LM or SM bit which activates the LM_SM block
2.	AR1 and the AR2 data for the register file
3.	The control signal for the mux which steers inputs between (SE+PC) and the left shifted value of the sign extended immediate data
4.	LM_SM block creates the control signal for the mux before AR2 which steers the input between the original AR2 and the AR2 created by the LM_SM block which sets data bits to zero
5.	LM_SM block also creates the signal for the mux which decides the data to be sent in the DO1 register. The original DO1 value or the auto incremented value of DO1 by the LM_SM block
6.	RR_PC created by the hazard block in RR stage which controls the mux which decides the input to PC should be PC+1 or SE value (when R7 is AR3 in LLI) or left shifted SE (when R7 is AR3 in LHI) or DO1 (when the instruction is JLR)

2. **Creation of Signals**: DO1 and DO2 data created from the register file

Execution

1. **Consumption of Signals**:

1.	Two control signals for the forwarding mux created by the forwarding logic block
2.	The control signal for the mux which steers input between DO2 or the sign extended value DO2 is used for arithmetic operations whereas sign extended value is used for address calculation
3.	The control signal for the mux which is created by the LM_SM block to select 1 such that the required address is always incremented by 1 during the LM or SM instruction
4.	The control signal for the hazard mux
5.	The control signal for the mux created by LM_SM block to store the created address in the register file for storing data during LM from the memory

2. **Creation of Signals:**

1.	The flag register values after the ALU operation
2.	The ALU output data after ALU operation

Data Memory Write/Read

1. **Consumption of Signals:**

1.	Memory write control for the data memory during instructions like SW or SM
2.	The control signal for the mux created by the LM_SM block which steers inputs between the ALU output, or the address calculated by the LM_SM block
3.	The control signal for the hazard mux in the Memory Write stage

2. **Creation of Signals:**

The data after reading from the data memory

Write Back

1. **Consumption of Signals:**

1.	**Flag values to be written in the user flag registers**
2.	**The control signals for the mux which decides the data to be written in the register file according to the instruction (ALU output, left shifted sign extended value, (SE+PC), SE, PC+1, Memory outdata)**
3.	**Flag control bits, Condition bits, Control Signals is sent to the hazard logic block in WB stage**
4.	**The control signal for the mux which decides the input to R7 in register file This created by the hazard block**

The data after WB mux, AR3 and some of the control signals (valid, the control signals for WB mux) is sent to a temporary register This register holds data for the forwarding logic.

The LM andSM

The LM_SM block is responsible generating the control signals to run the Load and Store multiple instructions. Inherently, the instruction is a multi-cycle instruction and thus must be performed by halting the pipeline.

The LM_SM block has the following inputs and outputs:

Inputs

- 8-bit data (from the instruction to be fed to the priority encoder)
  - Note that there is a mux connected to it, since the inputs for LM and SM (which are the same) will be in different clock cycles, the details of which are explained later.
- LM bit and SM bit
  - The LM and SM bits specifically tell when the instruction is LM or SM and are used to activate the block, so that priority encoder can start giving address
- clk and reset

Outputs

- Register Address: AR2 in case of SM, AR3 in case of LM
- Clear and disable for the pipeline registers
- RF_DO1_mux: Controls the mux connected to the input of DO1 register in RR_EX
- ALU2_mux: Controls the mux connected to the second input of ALU, decides when +1 should be the input to the ALU
- AR3_mux: Used in case of LM, this mux controls the data that is stored in AR3 in EX_MM
- AR2_mux: Used in case of SM, this mux controls the input to the register file for AR2
- mem_in_mux: Controls the input to the address of the memory which is usually the output of the ALU exceptin LM or SM where it is DO1

NOTE :– The LM_SM block starts outputting the address one cycle after it is activated

The block has been built using an FSM which goes into different states, depending on whether the instruction is LM or SM.

FSM Logic for LM:

- The block gets activated when the current instruction is in the RF stage. Here it is in the S1 state. The bitsthat control memory input, AR3, and ALU are ‘1’ and are put through the pipeline register.
- It then moves to the S2 state, where the mux connected to input of DO1 now starts accepting the output of ALU (DO1 + 1), and the block starts outputting AR3 addresses, which will be written in the write back stage into the register file. The disable signal is high, since the registers are now disabled (RR_EX, ID_RR, IF_ID and PC are disabled).

    - Note that there is a special enable signal to DO1 in RR_EX since that has to be enabled during the LM_SM process.
- The whole cycle continues till the last bit in the input of PE goes to zero, when the valid signal goes low, and one instruction currently in the RR_EX register has to be disabled. The disable signal goes low as well, meaning the pipeline flow has started again.

FSM Logic for SM

- The block gets activated when the current instruction is in the decode stage.
- In the next clock cycle, the LM_SM block starts outputting the AR2 address and thus, SM begins. The control bits follow the same pattern as LM, except the mux controlling the input to DO1 starts accepting the ALU output one cycle later. When the valid signal goes low, the last set of data is in the RR_EX register and so, in SM, no clear signal is required. The disable signal goes low and pipeline resumes.

PIPELINE REGISTER CONTENTS

Pipeline Register	Components	Length(bits)
IFID	PC	16
	Instruction	16
	PC++	16
IDRR	PC	16
	SEPC	16
	SE	16
	CL(Control Lines)	11
	• LS_PC • BEQ • LM • LW • SE_DO2 • WB_mux • valid	1 1 1 1 1 3 3
	Opcode	4
	ALU Control	2
	Flag Control	3
	Condition bits	2
	Write Bits	2
	AR1	3
	AR2	3
	AR3	3
	PC++	16
	LM input	8
	BLUT(Branch Table)	4
RREX	PC	16
	LSPC	16
	SE	16
	CL(Control Lines)	12

	• BEQ • LW • SE_DO2 • WB_mux • Valid • LM_SM control	1 1 1 3 3 3
	ALU Control	2
	Flag Control	3
	Condition	2
	Write	2
	BLUT	4
	DO1	16
	DO2	16
	AR1	3
	AR2	3
	AR3	3
	PC++	16
EXMM	LSPC	16
	SE	16
	CL(Control Lines)	8
	• BEQ • WB_mux • Valid • LM_SM control	1 3 3 1
	Flag Control	3
	Condition	2
	Write	2
	AR1	3
	AR2	3
	AR3	3
	Flags	3
	DO1	16
	DO2	16
	ALU output	16
MMWB	LSPC	16
	SE	16
	CL(Control Lines)	7
	• BEQ • WB_mux • valid	1 3 3
	Flag Control	3
	Condition	2
	Write	2
	AR1	3
	AR2	3
	AR3	3
	ALU output	16
	Memory output	16
	DO1	16
	PC++	16

HAZARD DETECTION AND MITIGATION

Data Forwarding

Data is forwarded through two multiplexers.

Each mux is controlled by separate forwarding unit which cheeks if any of the operand in execution stage depends on output of any instruction in further stages or current value of PC and provides corresponding data to ALU input.

Each pipeline register holds valid bits corresponding to operand and destination register address of corresponding instruction. Forwarding block considers these bits to determine dependency among instruction

Instruction-Wise Hazards andMitigation

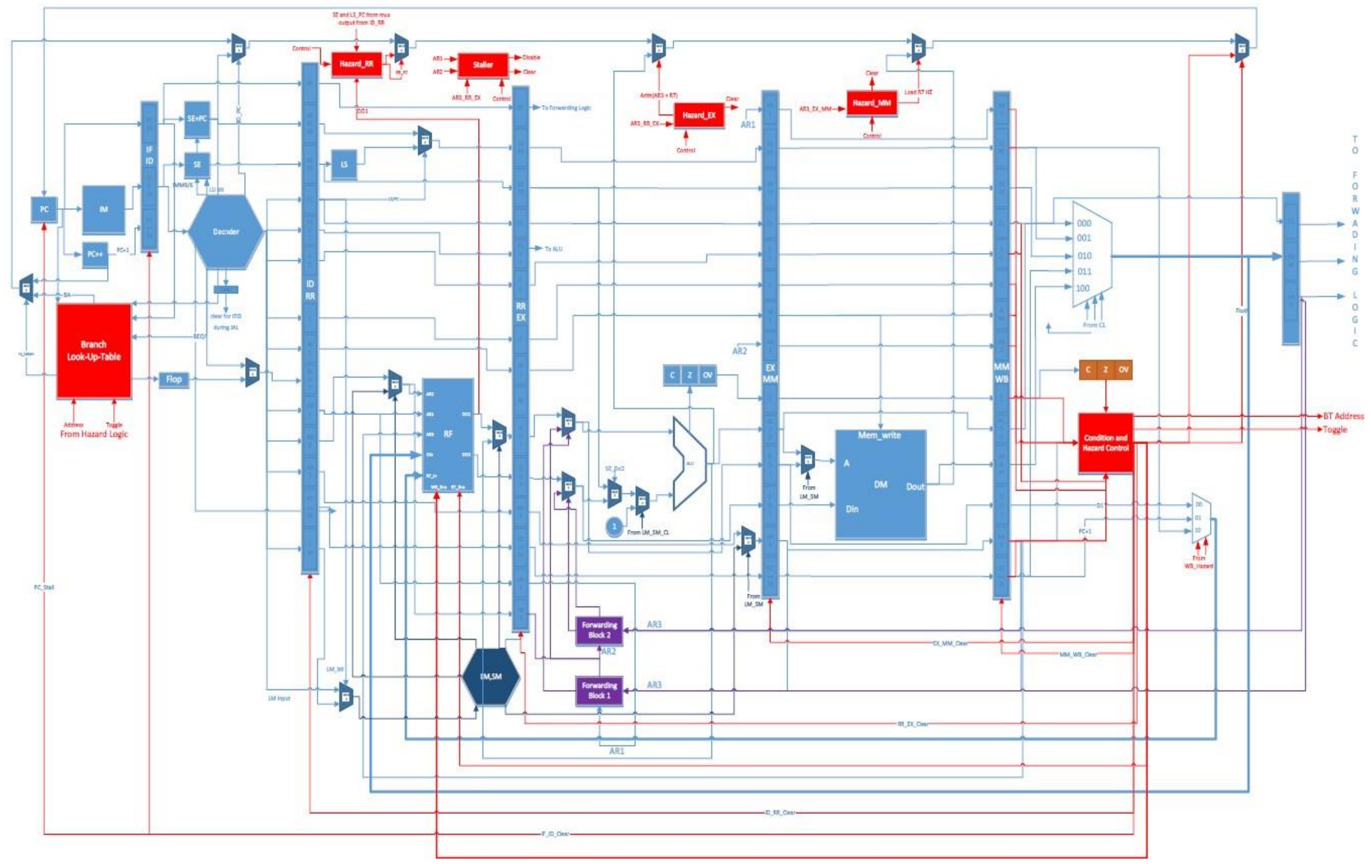
Arithmetic Instructions

1. Arithmetic instructions not having R7 as destination
   a. If instruction is unconditional only data forwarding is sufficient.
   b. If instruction is conditional, assume instruction to be taken and check instruction in writeback stage. If instruction is false and preceding instructions depend on the current instruction, then flush the pipeline. Dependency can be found by `AR1/2` and `Valid` bits in pipeline registers.
2. Arithmetic instructions having R7 as destination
   a. If instruction is unconditional then update PC in execution state and R7 in writeback stage. Flush `[IF-ID]and [ID-RR]` registers.
   b. If instruction is conditional assume it to be taken i.e. update PC in execution stage. If condition becomes false (checked in Writeback stage) then flush the pipeline.

Load Instructions

1. LW: Stall only if instruction in [RR-EX] is LW and instruction in [ID-RR] depends on its output.
2. If instruction in MM stage is load type (both LM and LW) and destination register is R7 then flush previous registers and update PC.

THE DATA PATH



Jump to register

Branch and Jump Instructions

JLR	Update PC in RR stage and clear [IF-ID] and [ID-RR] register.
JAL	Update PC in ID stage and clear [IF-ID] register.
BEQ	BEQ instructions are assumed to be taken/not taken based on the branch prediction table. a. If not found in the table, they are assumed to be not taken as the address to branch to is not known. Condition is checked in WB stage. b. If condition becomes opposite to what was predicted, the table is accordingly edited, the PC updated, and the pipeline is flushed. c. New entries to the table are made in the decode stage
JRI	Jump to memory location whose address is given by RA + immediate data

Hazard Mitigation Blocks

Stalling Block

1. If instruction in [ID-RR] is LW and instruction in [IF-ID] depends on its output, then stall for onecycle.
2. Clear enable of PC, [IF-ID], [ID-RR]. This will create two copies of LW instruction, one in [ID-RR] and one in [RR-EX]. So, pass the same signal through register to clear [ID-RR] in next signal.
3. If instruction in [IF-ID] is SM then delay SM start bit through register and MUX.

EX Stage Block

1. This block updates PC in case of Arithmetic Instructions with R7 as destination.
2. It sends output to PC and clear [IF-ID] and [ID-RR] registers.

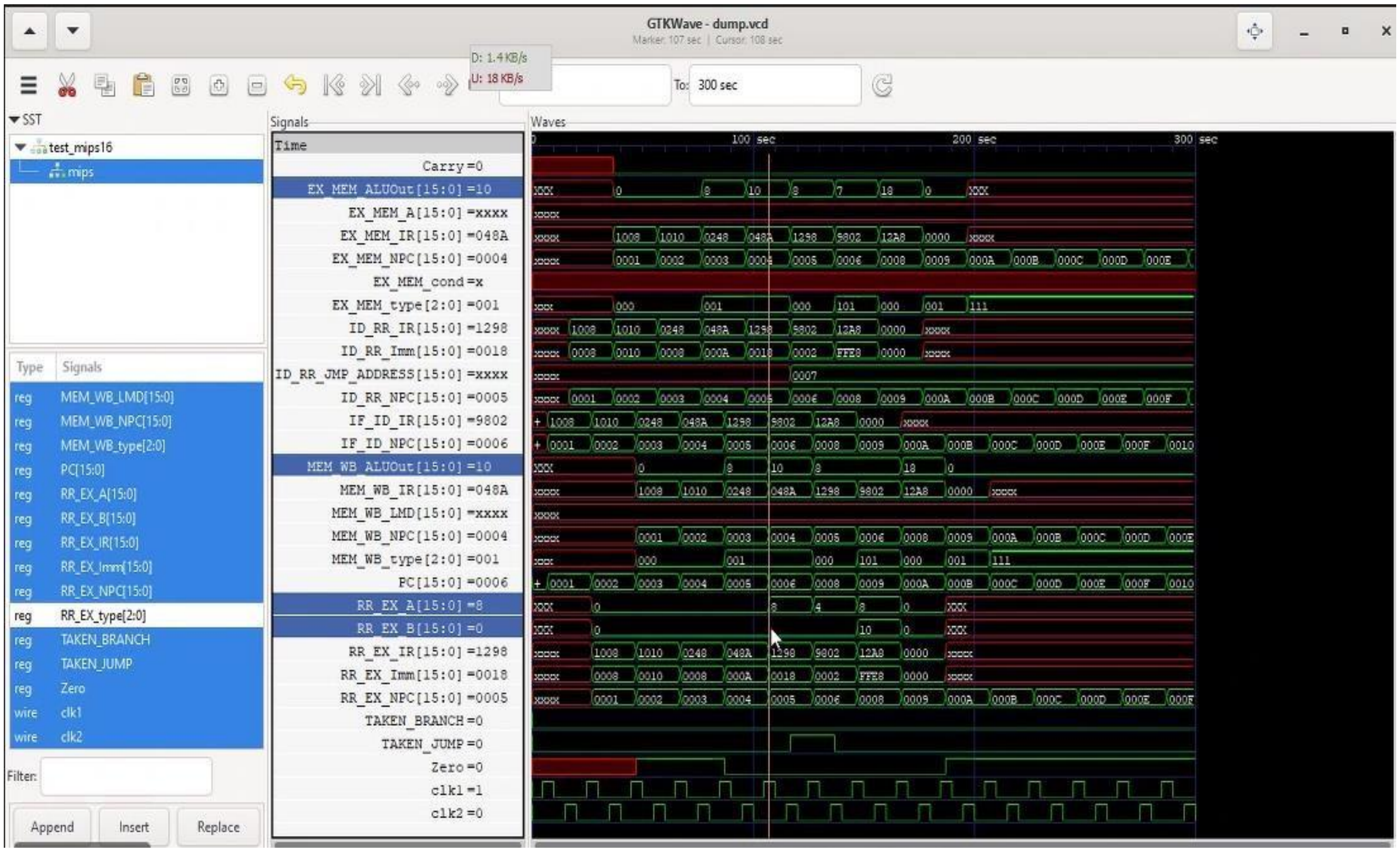
MM Stage Block

1. If any load type instruction has R7 as destination, then update PC in MM stage.
2. Clear [IF-ID], [ID-RR] and [RR-EX] registers.

Condition Control Block

Condition	Action
Conditional Arithmetic Instruction not having R7 as output destination becomes false and preceding instructions depends on it	Clear register write signal and flush pipeline.
Conditional Arithmetic Instruction not having R7 as output destination becomes false	Clear register write signal and flush pipeline. Write PC and R7 with PC+1
(JLR R7 R7)	Flush pipeline and write PC and R7 with PC+1
BEQ becomes true	Flush pipeline and write PC and R7 with (PC+SE)

SIMULATION RESULTS



```
Time
Carry=0
EX_MEM_ALUOut[15:0]=10
EX_MEM_A[15:0]=xxxx
EX_MEM_IR[15:0]=048A
EX_MEM_NPC[15:0]=0004
EX_MEM_cond=x
EX_MEM_type[2:0]=001
ID_RR_IR[15:0]=1298
ID_RR_Imm[15:0]=0018
ID_RR_JMP_ADDRESS[15:0]=xxxx
ID_RR_NPC[15:0]=0005
IF_ID_IR[15:0]=9802
IF_ID_NPC[15:0]=0006
MEM_WB_ALUOut[15:0]=10
MEM_WB_IR[15:0]=048A
MEM_WB_LMD[15:0]=xxxx
MEM_WB_NPC[15:0]=0004
MEM_WB_type[2:0]=001
PC[15:0]=0006
RR_EX_A[15:0]=8
RR_EX_B[15:0]=0
RR_EX_IR[15:0]=1298
RR_EX_Imm[15:0]=0018
RR_EX_NPC[15:0]=0005
TAKEN_BRANCH=0
TAKEN_JUMP=0
Zero=0
clk1=1
clk2=0
```

CODES

```
module pipe_MIPS16 (clk1, clk2);
input clk1, clk2; // Two-phase clock
reg [15:0] PC, IF_ID_IR, IF_ID_NPC, RR_EX_NPC, EX_MEM_NPC, MEM_WB_NPC, RR_EX_IR;
reg [15:0] ID_RR_IR, ID_RR_NPC, RR_EX_A, RR_EX_B, ID_RR_Imm, RR_EX_Imm;
reg [2:0] ID_RR_type, RR_EX_type, EX_MEM_type, MEM_WB_type;
reg [15:0] EX_MEM_IR, EX_MEM_ALUOut, EX_MEM_A;
reg EX_MEM_cond;
reg [15:0] MEM_WB_IR, MEM_WB_ALUOut, MEM_WB_LMD;
reg [15:0] ID_RR_JMP_ADDRESS; // JUMP Address
reg [15:0] Reg [0:7]; // Register bank (16 x 8)
reg [15:0] Mem [0:1023]; // 1024 x 32 memory
reg Carry, Zero; // Carry Flag and Zero Flag

parameter ADDR=4'b0001, ADI=4'b0000, NAND=4'b0010, LW=4'b0100, SW=4'b0101, LM=4'b1100,
SM=4'b1101, LA=4'b1110, SA=4'b1111, BEQ=4'b1000, JAL=4'b1001, JLR=4'b1010, JRI=4'b1011;
parameter RR_ALU=3'b000, RI_ALU=3'b001, LOAD=3'b010, STORE=3'b011, BRANCH=3'b100,
JUMP=3'b101;
```

```
parameter ADD=2'b00, ADC=2'b01, ADZ=2'b10, ADL=2'b11, NDU=2'b00, NDC=2'b01, NDZ=2'b10;
//reg HALTED;  // Set after HLT instruction is completed (in WB stage)
reg TAKEN_BRANCH, TAKEN_JUMP; // Required to disable instructions after branch

always @(posedge clk1) begin // IF (Instruction Fetch) Stage
 //if (HALTED == 0)
// begin
  if ((EX_MEM_IR[15:12] == BEQ) && (EX_MEM_cond == 1)) begin
    IF_ID_IR <= #2 Mem[EX_MEM_ALUOut];
    TAKEN_BRANCH <= #2 1'b1;
//    TAKEN_JUMP <= #2 1'b0;
    IF_ID_NPC <= #2 EX_MEM_ALUOut + 1;
    PC <= #2 EX_MEM_ALUOut + 1;
  end
  //else if (TAKEN_JUMP) begin
  else if ((ID_RR_IR[15:12]==4'b1001)||(ID_RR_IR[15:12]==4'b1010)||(ID_RR_IR[15:12]==4'b1011)) begin
    IF_ID_IR <= #2 Mem[ID_RR_JMP_ADDRESS];
    TAKEN_BRANCH <= #2 1'b0;
//    TAKEN_JUMP <= #2 1'b1;
    IF_ID_NPC <= #2 ID_RR_JMP_ADDRESS + 1;
    PC <= #2 ID_RR_JMP_ADDRESS + 1;
  end
  else begin
    IF_ID_IR <= #2 Mem[PC];
    TAKEN_BRANCH <= #2 1'b0;
//    TAKEN_JUMP <= #2 1'b0;
    IF_ID_NPC <= #2 PC + 1;
    PC <= #2 PC + 1;
  end
//  if ((IF_ID_IR[15:12]==4'b1001) || (IF_ID_IR[15:12]==4'b1011))
//  ID_RR_Imm <= #2 {{7{IF_ID_IR[8]}}, {IF_ID_IR[8:0]}};
//  else ID_RR_Imm <= #2 {{10{IF_ID_IR[5]}}, {IF_ID_IR[5:0]}};
end

always @(posedge clk2) begin // ID (Instruction Decode) Stage
//  if (HALTED == 0)
// begin
//  if (IF_ID_IR[11:9] == 3'b000) ID_RR_A <= 0;
//  else ID_RR_A <= #2 Reg[IF_ID_IR[11:9]]; // "rs"
//  if (IF_ID_IR[8:6] == 3'b000) ID_RR_B <= 0;
//  else ID_RR_B <= #2 Reg[IF_ID_IR[8:6]]; // "rt"
  ID_RR_NPC <= #2 IF_ID_NPC;
  ID_RR_IR <= #2 IF_ID_IR;
  if ((IF_ID_IR[15:12]==4'b1001) || (IF_ID_IR[15:12]==4'b1011))
  ID_RR_Imm <= #2 {{7{IF_ID_IR[8]}}, {IF_ID_IR[8:0]}};
  else ID_RR_Imm <= #2 {{10{IF_ID_IR[5]}}, {IF_ID_IR[5:0]}};

  case (IF_ID_IR[15:12])
    ADDR, NAND: begin
      ID_RR_type <= #2 RR_ALU;
      TAKEN_JUMP <= #2 1'b0;
    end
    ADI: begin
      ID_RR_type <= #2 RI_ALU;
      TAKEN_JUMP <= #2 1'b0;
    end
    LW: begin
      ID_RR_type <= #2 LOAD;
      TAKEN_JUMP <= #2 1'b0;
    end
    SW: begin
      ID_RR_type <= #2 STORE;
      TAKEN_JUMP <= #2 1'b0;
    end
    BEQ: begin
      ID_RR_type <= #2 BRANCH;
      TAKEN_JUMP <= #2 1'b0;
    end
    JAL: begin
      ID_RR_type <= #2 JUMP;
      TAKEN_JUMP <= #2 1'b1;
      ID_RR_JMP_ADDRESS <= #2 (IF_ID_NPC - 1) + {{7{IF_ID_IR[8]}}, {IF_ID_IR[8:0]}};
    end
    JLR: begin
      ID_RR_type <= #2 JUMP;
      TAKEN_JUMP <= #2 1'b1;
      ID_RR_JMP_ADDRESS <= #2 Reg[IF_ID_IR[8:6]];
    end
    JRI: begin
      ID_RR_type <= #2 JUMP;
      TAKEN_JUMP <= #2 1'b1;
      ID_RR_JMP_ADDRESS <= #2 Reg[IF_ID_IR[11:9]] + {{7{IF_ID_IR[8]}}, {IF_ID_IR[8:0]}};
    end
    default: ID_RR_type <= #2 3'b111; // Invalid opcode
  endcase
end

always @(posedge clk1) begin //RR (Register Read) Stage
  RR_EX_NPC <= #2 ID_RR_NPC;
  RR_EX_IR <= #2 ID_RR_IR;
  RR_EX_type <= #2 ID_RR_type;
  RR_EX_Imm <= #2 ID_RR_Imm;
  if (ID_RR_IR[11:9] == 3'b000) RR_EX_A <= #2 0;
  else RR_EX_A <= #2 Reg[ID_RR_IR[11:9]]; // "rs"
  if (ID_RR_IR[8:6] == 3'b000) RR_EX_B <= #2 0;
  else RR_EX_B <= #2 Reg[ID_RR_IR[8:6]]; // "rt"
end

always @(posedge clk2) begin // EX (Execution) Stage
// if (HALTED == 0)
// begin
  EX_MEM_NPC <= #2 RR_EX_NPC;
  EX_MEM_type <= #2 RR_EX_type;
  EX_MEM_IR <= #2 RR_EX_IR;
//  EX_MEM_Imm <= #2 RR_EX_Imm
//  TAKEN_BRANCH <= #2 0;
  case (RR_EX_type)
    RR_ALU: begin
      case (RR_EX_IR[15:12]) // "opcode"
        ADDR: begin
          case (RR_EX_IR[1:0])
            ADD: {Carry,EX_MEM_ALUOut} <= #2 RR_EX_A + RR_EX_B;
            ADC: begin
              if (Carry) {Carry,EX_MEM_ALUOut} <= #2 RR_EX_A + RR_EX_B;
              else {Carry,EX_MEM_ALUOut} <= {Carry,EX_MEM_ALUOut};
            end
            ADZ: begin
              if (Zero) {Carry,EX_MEM_ALUOut} <= #2 RR_EX_A + RR_EX_B;
              else {Carry,EX_MEM_ALUOut} <= {Carry,EX_MEM_ALUOut};
            end
            ADL: {Carry,EX_MEM_ALUOut} <= #2 RR_EX_A + (2*RR_EX_B);
          endcase
        end
        NAND: begin
          case (RR_EX_IR[1:0])
            NDU: EX_MEM_ALUOut <= #2 ~(RR_EX_A & RR_EX_B);
            NDC: begin
              if (Carry) EX_MEM_ALUOut <= #2 ~(RR_EX_A & RR_EX_B);
              else EX_MEM_ALUOut <= EX_MEM_ALUOut;
            end
            NDZ: begin
              if (Zero) EX_MEM_ALUOut <= #2 ~(RR_EX_A & RR_EX_B);
              else EX_MEM_ALUOut <= EX_MEM_ALUOut;
            end
          endcase
        end
        //default: EX_MEM_ALUOut <= #2 16'hxxxxxxxx;
      endcase
    end

    RI_ALU: {Carry,EX_MEM_ALUOut} <= #2 RR_EX_A + RR_EX_Imm; // ADI Instruction

    LOAD, STORE: begin
      EX_MEM_ALUOut <= #2 RR_EX_B + RR_EX_Imm;
      EX_MEM_A <= #2 RR_EX_A;
    end

    BRANCH: begin
      EX_MEM_ALUOut <= #2 RR_EX_NPC + RR_EX_Imm;
      EX_MEM_cond <= #2 (RR_EX_A == RR_EX_B);
    end

    JUMP: begin
      case (RR_EX_IR[15:12])
        JAL: EX_MEM_ALUOut <= #2 (RR_EX_NPC - 1) + RR_EX_Imm;
        JLR: EX_MEM_ALUOut <= #2 RR_EX_B;
        JRI: EX_MEM_ALUOut <= #2 RR_EX_A + RR_EX_Imm;
      endcase
    end

    default: EX_MEM_ALUOut <= #2 16'hxxxx;
  endcase
end

always @(posedge clk1) begin // MEM (Memory Access) Stage
// if (HALTED == 0)
// begin
  MEM_WB_NPC <= #2 EX_MEM_NPC;
  MEM_WB_type <= #2 EX_MEM_type;
  MEM_WB_IR <= #2 EX_MEM_IR;
  case (EX_MEM_type)
    RR_ALU, RI_ALU: begin
      MEM_WB_ALUOut <= #2 EX_MEM_ALUOut;
      if (EX_MEM_ALUOut == 16'h0000) Zero <= #2 1'b1;
      else Zero <= #2 1'b0;
    end
    LOAD: begin
      MEM_WB_LMD <= #2 Mem[EX_MEM_ALUOut];
      if (Mem[EX_MEM_ALUOut]==16'h0000) Zero <= #2 1'b1;
      else Zero <= #2 1'b0;
    end
    STORE: if (TAKEN_BRANCH == 0) Mem[EX_MEM_ALUOut] <= #2 EX_MEM_A; // Disable write
    //JUMP:
  endcase
end

always @(posedge clk2) begin // WB (Writeback) Stage
  if (TAKEN_BRANCH == 0) begin // Disable write if branch taken
    case (MEM_WB_type)
      RR_ALU: Reg[MEM_WB_IR[5:3]] <= #2 MEM_WB_ALUOut; // "rd"
      RI_ALU: Reg[MEM_WB_IR[8:6]] <= #2 MEM_WB_ALUOut; // "rt"
      LOAD: Reg[MEM_WB_IR[11:9]] <= #2 MEM_WB_LMD; // "rt"
      JUMP: if ((MEM_WB_IR[15:12]==4'b1001)||(RR_EX_NPC[15:12]==4'b1010)) Reg[MEM_WB_IR[11:9]]
<= #2 MEM_WB_NPC;
    endcase
  end
end

endmodule
```

TEST BENCH CODE

```
//`timescale 1us/1ns
module test_mips16;
reg clk1, clk2;
integer k;
pipe_MIPS16 mips (clk1, clk2);
initial begin

 clk1 = 0; clk2 = 0;
 repeat (20) // Generating two-phase clock
 begin
 #5 clk1 = 1; #5 clk1 = 0;
 #5 clk2 = 1; #5 clk2 = 0;
 end
 end



initial begin
  for (k=0; k<7; k=k+1) begin
    mips.Reg[k] = k;
    mips.Mem[0] = 16'h1008; // R1 = R0 + R0 = 0
    mips.Mem[1] = 16'h1010; // R2 = R0 + R0 = 0
    mips.Mem[2] = 16'h0248; // R1 = R1 + 8 = 8
    mips.Mem[3] = 16'h048A; // R2 = R2 + 10 = 10
    mips.Mem[4] = 16'h1298; // R3 = R1 + R2 = 18
    mips.Mem[5] = 16'h9802; // R4 = PC + 1= 6, PC = PC + 2 (PC = Address of Current Instruction = 5)
    mips.Mem[6] = 16'h0000; // Dummy
    mips.Mem[7] = 16'h12A8; // R5 = R1 + R2 = 18
    mips.Mem[8] = 16'h0000; // Dummy
  end

    //mips.HALTED = 0;
    mips.PC = 0;
    mips.TAKEN_BRANCH = 0;
    mips.TAKEN_JUMP = 0;
    #280 for (k=0; k<6; k=k+1) begin
      $display ("R%1d - %2d", k, mips.Reg[k]);
    end
end

initial begin
//  $dumpfile ("mips.vcd");
//  $dumpvars (0, test_mips16);
  $dumpvars ();
  #300 $finish;
end

endmodule
```