CO PROJECT

Names :

Amira Muhammad Fareed (1)
Basma Saeed Ragab (1)
Eslam Medhat Mahrous (1)
Eman Ashraf Ahmed (1)
Mohamed El-sayed Abdelfatah (3)

Section :

1, 3

Submission Date:

7/12/2017

Contents

CO PROJECT 0
1. Instruction Fetch Stage 2
   1.1. Modules 2
2. Instruction Decode Stage 4
   2.1. Modules 4
3. Excution Stage 8
   3.1. Modules 8
4. Memory Stage 13
   4.1. Modules 13
5. Write Back Stage 14
   5.1. Modules 14
6. Top Module 15
7. Test Bench 17
8. How To Add Instructions File 19
9. Synthesis Report 19
10. Test Cases 19
   10.1. No Hazards 19
   10.2. Data Hazard 19
   10.3. Control Hazard 19
   10.4. All Hazards 19

INSTRUCTION FETCH STAGE

MODULES

```
module Adder ( In1 , In2 , Out ) ;

input signed [31:0] In1 , In2 ;
output reg signed [31:0] Out ;

always @ (In1,In2)
begin
Out <= In1 + In2 ;
end
endmodule
```

```
module PC (inputAddress , clk , reset  , PC_hold, ReadAddress);

input [31:0] inputAddress;
input clk ,reset ,PC_hold;
output reg [31:0] ReadAddress;

always @ (posedge clk or posedge reset)
begin
if(reset)
ReadAddress <= 0;

else if(PC_hold)
ReadAddress  <= inputAddress - 1 ;

else
ReadAddress <= inputAddress ;
end
endmodule
```

```
module MUX_2x1_32bit (In1,In2,Sel,Output);

input signed [31:0] In1 , In2 ;
input Sel;
output reg signed [31:0] Output;

always @(In1 or In2 or Sel)
begin

if(Sel)
Output <= In2 ;

else
Output <= In1 ;

end
endmodule
```

```
module Instruction_Memory (ReadAddress,Instruction);

input [31:0] ReadAddress ;
output reg [31:0] Instruction ;
reg [31:0] iMemory [0:1023];

always@(ReadAddress)
begin
Instruction <= iMemory[ReadAddress];
end

endmodule
```

INSTRUCTION DECODE STAGE

MODULES

```
module Sign_Extend (In,Out);

input signed [15:0] In ;
output reg signed [31:0] Out;

always @(In)
begin
Out <= {{16{In[15]}},In};
end

endmodule
```

```
module Control (OP_Code ,Control_Signals);

input [5:0] OP_Code;
output reg [9:0] Control_Signals;

// 0 => RegWrite , 1 => MemtoReg , 2 => MemWrite , 3 => MemRead , 4 => AluSrc , 5:6 => AluOP , 7 => RegDst
, 8 => branch , 9 => jump

always @(OP_Code)
begin

case(OP_Code)
 /* R format  */
0: Control_Signals <= 10'b 0011000011 ;

 /* jump */
2: Control_Signals <= 10'b 10xxxx00x0 ;

/* beq */
4: Control_Signals <= 10'b 01x01x00x0 ;

/* LW */
35: Control_Signals <= 10'b 0000111001 ;

/* SW */
43: Control_Signals <= 10'b 00x01101x0 ;

endcase
end
endmodule
```

```
module Reg_File ( RR1 , RR2 , WR , WD , RegWrite , clk , RD1 , RD2);

input [4:0] RR1 , RR2 , WR ; // Addresses :)
input [31:0] WD ; // WriteData
input RegWrite ,clk ;  // Control
output [31:0] RD1,RD2 ; // ReadData
reg [31:0] RF [0:31];

always @(RR1 or RR2)
begin
RD1 <= RF[RR1];
RD2 <= RF[RR2];
end
always @(posedge clk)
begin
if(RegWrite)
RF[WR] <= WD;
end
endmodule
```

```
module Sll_32bit (In , Out);

input [31:0] In;
output reg [31:0] Out ;

always @ (In)
begin
Out <= In << 2 ;
end
endmodule
```

```
module Adder ( In1 , In2 , Out ) ;

input signed [31:0] In1 , In2 ;
output reg signed [31:0] Out ;

always @ (In1,In2)
begin
Out <= In1 + In2 ;
end
endmodule
```

```
module HazardDetection_Unit ( Rs_D , Rt_D , Rt_Ex , MemRead_Ex , PC_hold , Instruction_hold , Stall_Control );

input [4:0] Rs_D , Rt_D , Rt_Ex ;
input MemRead_Ex ;
output reg PC_hold , Instruction_hold , Stall_Control ;

always @ (Rs_D or Rt_D or Rt_Ex or MemRead_Ex)
begin

if( MemRead_Ex && (Rs_D == Rt_Ex || Rt_D == Rt_Ex))
begin
PC_hold <= 1;
Instruction_hold <= 1;
Stall_Control <= 1;
end

else
begin
PC_hold <= 0;
Instruction_hold <= 0;
Stall_Control <= 0;
end
end
endmodule
```

```
module MUX_2x1_8bit (In1,In2,Sel,Output);

input signed [7:0] In1 , In2 ;
input Sel;
output reg signed [7:0] Output;

always @(In1 or In2 or Sel)
begin

if(Sel)
Output <= In2 ;

else
Output <= In1 ;

end
endmodule
```

```
module IFID_Reg (Instruction , PC4 , clk , Instruction_hold , PC4_D , Instruction_D );

input [31:0] Instruction , PC4 ;
input  clk , Instruction_hold ;

output reg [31:0] PC4_D , Instruction_D ;

always @ (posedge clk)
begin

PC4_D <= PC4 ;
if (Instruction_hold == 1)
#3 Instruction_D = Instruction_D ;

else
#3 Instruction_D <= Instruction ;

end
endmodule
```

```
module Comparator (In1 , In2 , Output);

input [31:0] In1 , In2 ;
output reg Output ;
reg [31:0] XOR_output ;

always@ (In1 or In2)
begin

XOR_output <= In1 ^ In2;

#5
if(XOR_output == 0)
Output <= 1;

else
Output <= 0;
end
endmodule
```

EXCUTION STAGE

MODULES

```
module IDEX( /*inputs*/readData1, readData2, offset, rs, rt, rd, funct, memCtrl, exctrl, wbctrl, clk,shamt,
    /*outputs*/ IDEX_readData1, IDEX_readData2, IDEX_offset, IDEX_rs, IDEX_rt, IDEX_rd, IDEX_funct,
IDEX_memCtrl, IDEX_exctrl, IDEX_wbctrl , IDEX_shamt );

input [31:0] readData1, readData2, offset;
input [5:0]  funct;
input [4:0] rs, rt, rd ,shamt;
input [3:0] exctrl;
input [1:0] memCtrl, wbctrl;
input  clk;

output reg [31:0] IDEX_readData1, IDEX_readData2, IDEX_offset;
output reg [5:0]  IDEX_funct;
output reg [4:0] IDEX_rs, IDEX_rt, IDEX_rd, IDEX_shamt;
output reg [3:0] IDEX_exctrl;
output reg [1:0] IDEX_memCtrl, IDEX_wbctrl;

always @(posedge clk)

begin

IDEX_readData1<=readData1;
IDEX_readData2<=readData2;
IDEX_offset<=offset;
IDEX_funct<=funct;
IDEX_rs<=rs;
IDEX_rt<=rt;
IDEX_rd<=rd;
IDEX_exctrl<=exctrl;
IDEX_memCtrl<=memCtrl;
IDEX_wbctrl<=wbctrl;
end

endmodule
```

```
module Alu_Control(ALUOP , funct, OP);

input [1:0] ALUOP ;
input [5:0] funct ;
output reg [3:0] OP ;

always@(ALUOP or funct )
begin
case(ALUOP)

0: OP <= 2; // add(lw,sw)
1: OP <= 3; // sub(beq)
2:        begin
          OP <=(funct==36)? 0: //and
         (funct==37)? 1: //or
         (funct==32)? 2: //add
             (funct==34)? 3: //sub
             (funct==0) ? 4: //sll
             (funct==42)? 5:4'bx ;
            end

default: OP <= 4'bx;
endcase
end

endmodule
```

```
module Mux3x1_32( muxOut, in0, in1, in2, muxSel );
input [31:0] in0, in1, in2;
input [1:0] muxSel;

output reg [31:0] muxOut;

always@(in0 or in1 or in2 or muxSel)
begin

muxOut<=( muxSel==0 )? in0 :
      ( muxSel==1 )? in1 :
      ( muxSel==2 )? in2 :
       32'bx;
end
endmodule
```

```
module MUX_2x1_5bit(In1,In2,Sel,Output);

input signed [4:0] In1 , In2 ;
input Sel;
output reg signed [4:0] Output;

always @(In1 or In2 or Sel)
begin

if(Sel)
Output <= In2 ;

else
Output <= In1 ;

end
endmodule
```

```
module MUX_2x1_32bit(In1,In2,Sel,Output);

input signed [31:0] In1 , In2 ;
input Sel;

output reg signed [31:0] Output;

always @(In1 or In2 or Sel)
begin

if(Sel)
Output <= In2 ;

else
Output <= In1 ;

end
endmodule
```

```
module ALU(In1 , In2 , OP ,shamt, Result);

input signed [31:0] In1 , In2 ;
input [3:0] OP ;
input [4:0] shamt ;
output reg signed [31:0] Result ;

always @(In1 or In2 or OP)
begin
  case(OP)
    0 : Result <= In1 & In2 ; // and
    1 : Result <= In1 | In2 ; // or
    2 : Result <= In1 + In2 ; // add
    3 : Result <= In1 - In2 ; // sub
    4 : Result <= In1 << shamt ; //sll
    5 : Result <=(In1 < In2) ? 1 : 0 ; //slt
    default : Result <= 32'b x ;
  endcase
end
endmodule
```

```
module forwardingUnit( forwardA, forwardB, IDEX_rs, IDEX_rt, EXMEM_desReg, EXMEM_regWrite, MEMWB_regWrite, MEMWB_desReg );

input [4:0] IDEX_rs, IDEX_rt, EXMEM_desReg, MEMWB_desReg;
input EXMEM_regWrite, MEMWB_regWrite;

output reg [1:0] forwardA, forwardB;

always @( IDEX_rs or IDEX_rt or EXMEM_desReg or EXMEM_regWrite or MEMWB_regWrite or MEMWB_desReg )
begin

// Rd=RS & instruction before me is writing & instruction before me not writing in register $0
if((EXMEM_regWrite) &&(EXMEM_desReg != 0)  &&(EXMEM_desReg == IDEX_rs) )
begin
forwardA<=2'b10;
end
```

```
// if instruction(-2) is writing but in $0  &&  instruction(-1) in not writing  && destination of instruction(-1) != Rs of instruction(0)  && destination of instruction(-2) == RS
else if((MEMWB_regWrite) &&( MEMWB_desReg != 0 ) &&( MEMWB_desReg == IDEX_rs )  &&  !(( EXMEM_regWrite) &&( EXMEM_desReg != 0 ) &&( EXMEM_desReg != IDEX_rs )) )
begin
forwardA<=2'b01;
end

// no hazards
else
begin
forwardA<=2'b00;
end

// Rd=Rt & instruction before me is writing & instruction before me not writing in register $0
if((EXMEM_regWrite) &&(EXMEM_desReg != 0)  &&(EXMEM_desReg == IDEX_rt) )
begin
forwardB<=2'b10;
end

// if instruction(-2) is writing but in $0  &&  instruction(-1) in not writing  && destination of instruction(-1) != Rt of instruction(0)  && destination of instruction(-2) == Rt
else if((MEMWB_regWrite) &&( MEMWB_desReg != 0 ) && (! EXMEM_regWrite) &&( EXMEM_desReg != 0 ) &&( EXMEM_desReg != IDEX_rt ) &&( MEMWB_desReg == IDEX_rt ) )
begin
forwardB<=2'b01;
end

//no hazards
else
begin
forwardB<=2'b00;
end

end

endmodule
```

MEMORY STAGE

MODULES

```
module Data_Memory (Address , WD , MemRead , MemWrite , clk , RD);

input  [31:0] Address , WD ;
input  MemRead , MemWrite , clk;
output reg [31:0] RD ;
reg [31:0] dMemory [0:1023];

always@(Address or WD or MemRead)
begin
if (MemRead)
RD <= dMemory [Address];
end

always @(negedge clk)
begin
if(MemWrite)
dMemory [Address] <= WD ;
end
endmodule
```

```
module Data_Memory (Address , WD , MemRead , MemWrite , clk , RD);

input [31:0] Address , WD ;
input MemRead , MemWrite , clk;
output reg [31:0] RD ;
reg [31:0] dMemory [0:1023];

always@(Address or WD or MemRead)
begin
if (MemRead)
RD <= dMemory [Address];
end

always @(negedge clk)
begin
if(MemWrite)
dMemory [Address] <= WD ;
end
endmodule
```

WRITE BACK STAGE

MODULES

```
module MUX_2x1_32bit (In1,In2,Sel,Output );
input signed [31:0] In1 , In2 ;
input Sel;
output  reg signed [31:0] Output ;
always @(In1 or In2 or Sel)
begin
if(Sel)   Output  <= In2 ;
else      Output  <= In1 ;
end
endmodule
```

```
module MEM_WB_Reg (RD_M, Alu_Result_M ,WR_M , WB_Control,clk,regWrite ,MemtoReg ,RD, Alu_Result ,WR ) ;
input [4:0] WR_M ;
input [31:0] RD_M , Alu_Result_M ;
input [1:0] WB_Control ;
input clk ;
output  reg [4:0] WR ;
output  reg [31:0] RD , Alu_Result ;
output  reg regWrite , MemtoReg ;
always @(posedge clk)
begin
RD <= RD_M ;
Alu_Result <= Alu_Result_M ;
WR <= WR_M ;
RegWrite <= WB_Control[0];
MemtoReg <= WB_Control[1];
end
endmodule
```

TOP MODULE

```
module Top_Module (clk , reset);

input clk ,reset ;

wire PC_hold , Instruction_hold , RegWrite_WB , ComparatorResult , Stall_Control , MemRead , MemWrite,
 MemtoReg , branch , branch_ornot;
wire [1:0] WB_D , WB_M ,WB_EX , MEM_D , MEM_EX , forwardA, forwardB;
wire [3:0] EX_D , EX_EX , OP;
wire [4:0] Rs_D , Rt_D , Rt_EX , Rs_EX, Rd_D , Rd_EX ,shamt_D , shamt_EX , WR_WB, WR_M , WR_EX;
wire [5:0] funct_EX;
wire [7:0] Control_D ;
wire [9:0] Control_Signals;
wire [31:0] PC4 , PC4_D ,inputAddress, ReadAddress , Instruction , Instruction_D , WD_WB , RD1 , RD2 ,
Immediate_D , branch_offset, branch_Address , RD1_EX, RD2_EX, Immediate_EX ,Alu_In1 , Alu_In2
,Alu_Result_EX,Alu_Result_WB, Alu_Result_M , M3_out , RD ,RD_WB ,WD;

/*/////////////////////////////////////////// Instruction Fetch \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\*/
PC pc (inputAddress , clk ,reset , PC_hold , ReadAddress);

Adder A1 ( ReadAddress , 1 , PC4 ) ;

Instruction_Memory IMemory (ReadAddress , Instruction);

and a (branch_ornot,branch,ComparatorResult);

MUX_2x1_32bit MM (PC4 , branch_Address, branch_ornot ,inputAddress);

/*/////////////////////////////////////////// Instruction Decode \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\*/
IFID_Reg IFID (Instruction , PC4 , clk , Instruction_hold , PC4_D , Instruction_D );

Control control (Instruction_D[31:26] ,Control_Signals);

Reg_File reg_file ( Rs_D , Rt_D , WR_WB , WD_WB , RegWrite_WB , clk , RD1 , RD2);

Comparator comparator ( RD1 , RD2 , ComparatorResult );

Sign_Extend sign_extend (Instruction_D[15:0],Immediate_D);

Sll_32bit sll_32 (Immediate_D , branch_offset);

Adder A2 ( branch_offset , PC4_D , branch_Address) ;

HazardDetection_Unit hazard_detector ( Rs_D , Rt_D , Rt_EX , MEM_EX[1] , PC_hold ,
 Instruction_hold , Stall_Control );

MUX_2x1_8bit M1 (Control_Signals[7:0] , 8'h00 , Stall_Control , Control_D );
```

```
assign Rs_D = Instruction_D[25:21];
assign Rt_D = Instruction_D[20:16];
assign Rd_D = Instruction_D[15:11];
assign shamt_D = Instruction_D[10:6];
assign WB_D = Control_D [1:0] ;
assign MEM_D = Control_D [3:2] ;
assign EX_D = Control_D [7:4] ;
assign branch = Control_D [8] ;

/*/////////////////////////////////////////////// Execution \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\*/

IDEX_Reg IDEX (/*inputs*/ RD1 , RD2, Immediate_D, Rs_D , Rt_D , Rd_D,
            Instruction_D[5:0], MEM_D, EX_D, WB_D,clk,shamt_D,
            /*outputs*/ RD1_EX, RD2_EX, Immediate_EX, Rs_EX, Rt_EX, Rd_EX, funct_EX,
               MEM_EX, EX_EX, WB_EX, shamt_EX );

Alu_Control alu_control (EX_EX[2:1], funct_EX, OP);

ALU alu (Alu_In1 , Alu_In2 , OP ,shamt_EX, Alu_Result_EX);

Mux3x1_32 M2( Alu_In1 ,RD1_EX, WD_WB, Alu_Result_M, forwardA );

Mux3x1_32 M3( M3_out, RD2_EX, WD_WB , Alu_Result_M, forwardB );

MUX_2x1_32bit M4 (M3_out, Immediate_EX, EX_EX[0],Alu_In2);

MUX_2x1_5bit M5 (Rt_EX, Rd_EX,EX_EX[3],WR_EX);

forwardingUnit forwarding_unit ( forwardA, forwardB,
                    Rs_EX, Rt_EX, WR_M , WB_M[0], RegWrite_WB , WR_WB );

/*/////////////////////////////////////////////// Memory  \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\*/

EXMEM_Reg EXMEM (WB_EX , MEM_EX , Alu_Result_EX , RD2_EX ,WR_EX, clk , WB_M , MemRead ,
MemWrite ,Alu_Result_M ,WD , WR_M );

Data_Memory DMemory (Alu_Result_M , WD , MemRead , MemWrite , clk , RD);

/*/////////////////////////////////////////////// Write Back  \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\*/

MEM_WB_Reg MEMWB ( RD , Alu_Result_M , WR_M , WB_M , clk , RegWrite_WB , MemtoReg , RD_WB ,
Alu_Result_WB , WR_WB ) ;

MUX_2x1_32bit M6 (RD_WB , Alu_Result_WB ,MemtoReg,WD_WB);

endmodule
```

TEST BENCH

```
module Test_Bench ;
reg clk , reset ;

Top_Module top_module (clk , reset);

always
begin
#150
clk = ~ clk ;
end
initial
begin

$readmemb("inst_mem.txt" ,top_module.IMemory.iMemory);

top_module.reg_file.RF[0] <= 0; //$zero
top_module.reg_file.RF[1] <= 5;
top_module.reg_file.RF[2] <= 1;
top_module.reg_file.RF[3] <= 2;
top_module.reg_file.RF[4] <= 3;
top_module.reg_file.RF[5] <= 4;
top_module.reg_file.RF[6] <= 5;
top_module.reg_file.RF[7] <= 6;
top_module.reg_file.RF[8] <= 7;   //$t0
top_module.reg_file.RF[9] <= 5;  //$t1
top_module.reg_file.RF[10] <= 4;  //$t2
top_module.reg_file.RF[11] <= 2;  //$t3
top_module.reg_file.RF[12] <= 3;  //$t4
top_module.reg_file.RF[13] <= 2;  //$t5
top_module.reg_file.RF[14] <= 5;  //$t6
top_module.reg_file.RF[15] <= 8; //$t7
top_module.reg_file.RF[16] <= 1; //$s0
top_module.reg_file.RF[17] <= 2; //$s1
top_module.reg_file.RF[18] <= 3; //$s2
top_module.reg_file.RF[19] <= 3; //$s3
top_module.reg_file.RF[20] <= 1; //$s4
top_module.reg_file.RF[21] <= 2; //$s5
top_module.reg_file.RF[22] <= 2; //$s6
top_module.reg_file.RF[23] <= 0; //$s7
```

```
top_module.reg_file.RF[24] <= 2;//$t8
top_module.reg_file.RF[25] <= 3;//$t9


top_module.DMemory.dMemory [0] = 0 ;
top_module.DMemory.dMemory [1] = 1 ;
top_module.DMemory.dMemory [2] = 2 ;
top_module.DMemory.dMemory [3] = 3 ;
top_module.DMemory.dMemory [4] = 4 ;
top_module.DMemory.dMemory [5] = 5 ;
top_module.DMemory.dMemory [6] = 6 ;
top_module.DMemory.dMemory [7] = 50 ;
top_module.DMemory.dMemory [8] = 0 ;
top_module.DMemory.dMemory [9] = 1 ;
top_module.DMemory.dMemory [10] = 2 ;
top_module.DMemory.dMemory [11] = 3 ;
top_module.DMemory.dMemory [12] = 4 ;
top_module.DMemory.dMemory [13] = 5 ;
top_module.DMemory.dMemory [14] = 6 ;
top_module.DMemory.dMemory [15] = 7 ;
top_module.DMemory.dMemory [16] = 8 ;
top_module.DMemory.dMemory [17] = 9 ;
top_module.DMemory.dMemory [18] = 10 ;
top_module.DMemory.dMemory [19] = 11 ;
top_module.DMemory.dMemory [20] = 12 ;
top_module.DMemory.dMemory [21] = 13 ;
top_module.DMemory.dMemory [22] = 14 ;
top_module.DMemory.dMemory [23] = 15 ;
top_module.DMemory.dMemory [24] = 16 ;
top_module.DMemory.dMemory [25] = 17 ;

#50
clk = 0;
reset = 1;
#100 reset = 0;
end
endmodule
```

HOW TO ADD INSTRUCTIONS FILE

1. Save your instructions in Binary , in a inst_mem.txt file .
2. Place your inst_mem.txt file in the Project's directory .
3. Save the Test_bench.v → compile → simulate.

SYNTHESIS REPORT

synthesis-report.txt

TEST CASES

Type	Assembly	Instruction memory	Expected Output	Register file	Data memory
NO HAZARDS	Assembly	Inst_mem	Exepected Output	Reg file	Data Memory
DATA HAZARD	Assembly	inst_mem	Exepected Output	Register FIle	Data Memory
Data Hazard	Assembly	inst_mem	Exepected Output	Register File	
Data Hazard	Assembly	inst_mem	Exepected Output	Register File	
Data Hazard	Assembly	inst mem	Exepected Output	Register FIle	Data Memory
Data Hazard	Assembly	inst mem	Exepected Output	Register FIle	Data Memory
Data Hazard	Assembly	inst mem	Exepected Output	Register FIle	
CONTROL HAZARD	Assembly	inst mem	Exepected Output	Register FIle	
Control Hazard	Assembly	inst mem	Exepected Output	Register FIle	
Control Hazard	Assembly	inst mem	Exepected Output	Register FIle	
ALL HAZARDS	Assembly	inst mem	Exepected Output	Register FIle	