Pipelined Processor

冯硕 11911736

This is the lab report of Pipelined Processor. In the last project, I have built up a Single-Cycle ARM Processor. Below will shows the structure of Pipelined ARM Processor.

Analysis Report

In the last report, I have built up a Single-Cycle Processor, whose over all structure of Single-Cycle ARM Processor is shown in Fig. 1.

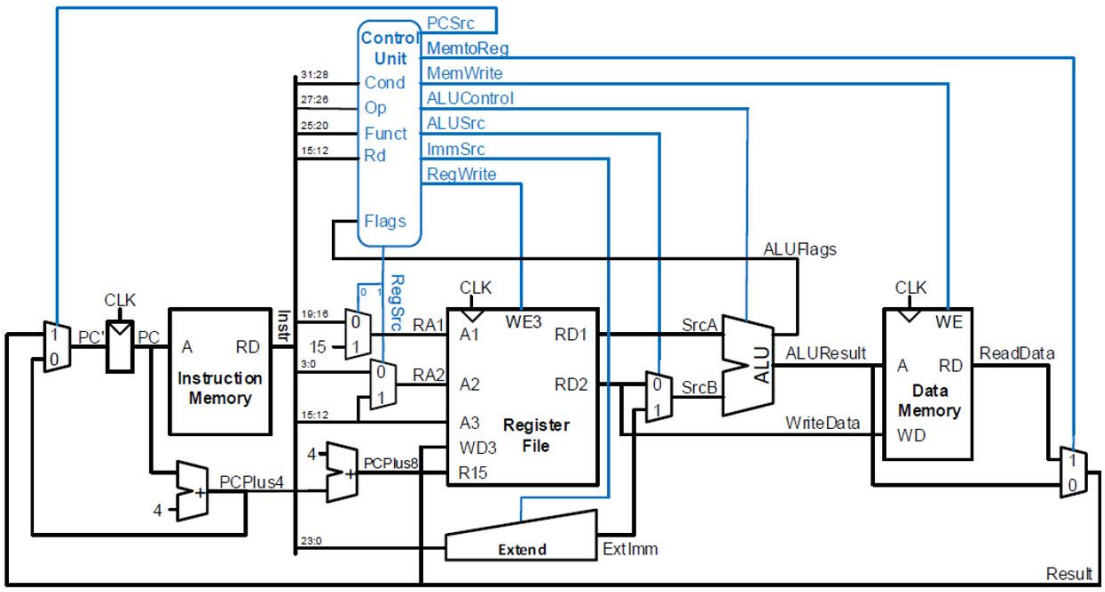
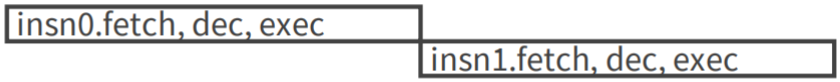


Fig. 1: Single-Cycle ARM Processor with control.

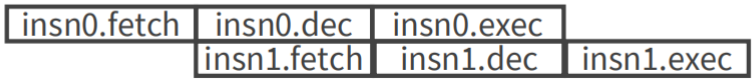
3 type of Processor Comparisons

type	Advantages	Disadvantages
Single Cycle Processor	Single cycle per instruction make logic and clock simple	Since instructions take different time to finish, memory and functional unit are not efficiently utilized; Cycle time is the longest delay
Multi Cycle Processor	Better MIPS and smaller clock period (higher clock frequency)	Higher CPI than single cycle processor
Pipelined Processor	want small CPI (close to 1) with high MIPS and short CLK period	Hazards!

Single-cycle



Pipelined



For the wanted Pipelined Processor, the overall structure is shown in Fig. 2. We notice that Hazard Unit is added in this target processor.

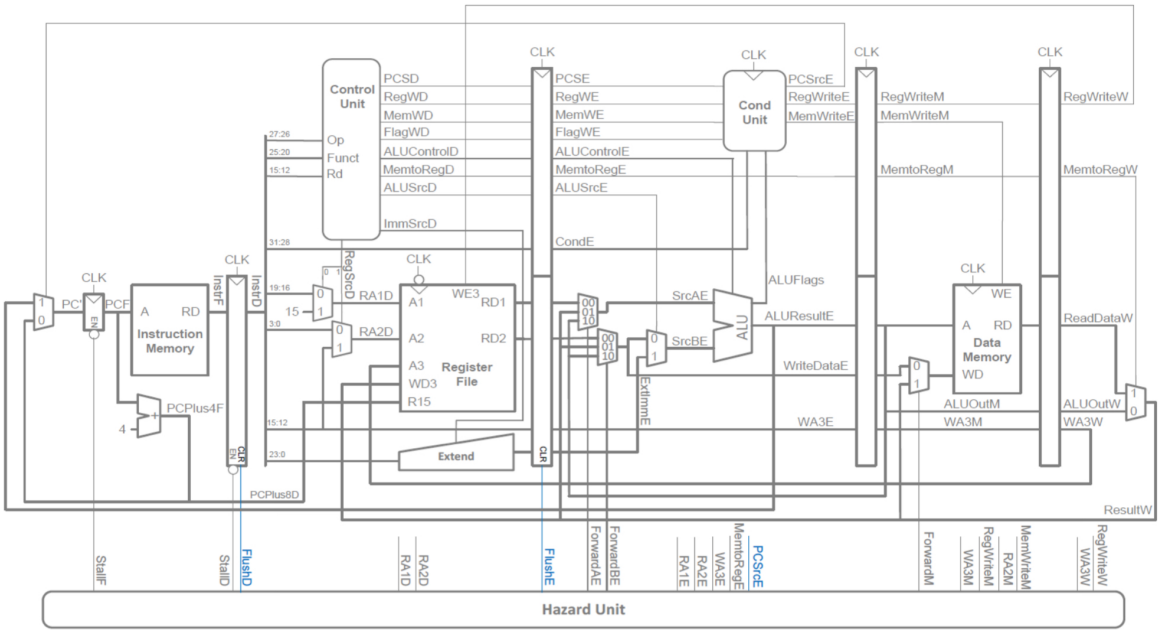


Fig. 2: the Pipelined ARM Processor with control signal.

Pipeline Stages

Stage	Perform Functionality	Latch values of interest
Fetch	Use PC to index Program Memory, increment PC	Instruction bits (to be decoded); PC + 4 (to compute branch targets)
Decode	Decode instruction, generate control signals, read register file	Control information, Rd index, immediates, offsets, register values (Ra, Rb), PC+4 (to compute branch targets)
Execute	Perform ALU operation Compute targets (PC+4+offset, etc.) in case this is a branch, decide if branch taken	Control information, Rd index, etc. Result of ALU operation, value in case this is a store instruction
Memory	Perform load/store if needed, address is ALU result	Perform load/store if needed, address is ALU result
Writeback	Select value, write to register file	

Instruction Fetch (IF)

Stage 1: Instruction Fetch

Fetch a new instruction every cycle. Current PC is index to instruction memory, increment the PC at end of cycle (assume no branches for now). Also, write values of interest to pipeline register (IF/ID) , Instruction bits (for later decoding) .

Decode (D)

Stage 2: Instruction Decode

On every cycle, will read IF/ID pipeline register to get instruction bits, decode instruction, generate control signals; read from register file; write values of interest to pipeline register (ID/EX) ; control information, Rd index, immediates, offsets, ... ; contents of RA1, RA2.

Execute (EX)

Stage 3: Execute

On every cycle: Read ID/EX pipeline register to get values and control bits; Perform ALU operation; Compute targets (PC+4+offset, etc.) in case this is a branch; Decide if jump/branch should be taken; Write values of interest to pipeline register (EX/MEM) ; Control information, Rd index, ... ; Result of ALU operation ; Value in case this is a memory store instruction.

Memory (M)

Stage 4: Memory

On every cycle: Read EX/MEM pipeline register to get values and control bits ; Perform memory load/store if needed - address is ALU result ; Write values of interest to pipeline register (MEM/WB) ; Control information, Rd index, ... ; Result of memory operation ; Pass result of ALU operation.

WriteBack (W)

Stage 5: Write-back

On every cycle: Read MEM/WB pipeline register to get values and control bits ; Select value and write to register file.

Modules in CPU (Pipelined_Processor.v)

What's new in this pipelined processor: the ControlUnit.v is divided into ControlUnit.v (with Decoder.v) and CondUnit.v; and the HazardUnit.v is a newly added module.

Module	Instance	Description
ProgrammCounter.v	PC0	gives the instruction address
instr_mem.v	InstructionMem0	only need 7 bit to specify the address, capacity is 128 words
ControlUnit.v	ControlUnit0	gives the inter-media control signals: **PCSD**, **MemtoRegD**, **MemWD**, **ALUControlD**, **ALUSrcD**, **ImmSrcD**, **RegWD**.
CondUnit.v *(new)*	CondUnit0	gives the signals to judge whether to execute, **PCSrcE**, **RegWriteE**, **MemWriteE**
RegisterFile.v	RegisterFile0	2^4 registers, each register stores 32-bit data
Extend.v	Extend0	extends the immediate number to 32-bit **ExtImm**
ALU.v	ALU0	gives the arithmetic logic results **ALUResult**
data_mem.v	DataMem0	gives **ReadData** and can write data in data memory, 128 words capacity, 32-bit for each data
HazardUnit.v *(new)*	HazardUnit0	gives control signals for 4 registers in 5 stages to handle the Data Hazard and Control Hazard.

State Registers

There are in total 5 stages: fetch (F), decode (D), execute (E), memory (M) and write back (WB). Among each stage, there are 4 registers:

Fetch / Decode Register: Reg_F_D

```
/*************************************************
-----------------------------------------------------------
Register Reg_F_D: connect Fetch Stage and Decode Stage
-----------------------------------------------------------
*************************************************/
    always @ (posedge CLK) begin
        if(FlushD) InstrD <= 32'd0;
        else if(~StallD) InstrD <= InstrF;
        else InstrD <= InstrD;
    end
```

Note that: ***FlushD*** from Hazard Unit gives the ***Clear*** signal of this register; ***StallD*** from Hazard Unit gives the ***Enable*** signal of this register, (active low); ***CLK*** from input signal gives the ***Clock*** signal of the register.

Decode / Execute Register: Reg_D_E

```
/*************************************************
-----------------------------------------------------------
*Register Reg_D_E: connect Decode Stage and Execute Stage
-----------------------------------------------------------
*************************************************/
    always @ (posedge CLK) begin//
        if(FlushE) begin
            PCSE <= 1'b0;
            RegWE <= 1'b0;
            MemWE <= 1'b0;
            FlagWE <= 1'b0;
            end
        else begin
            PCSE <= PCSD;
            RegWE <= RegWD;
            MemWE <= MemWD;
            FlagWE <= FlagWD;
            ALUControlE <= ALUControlD;
            MemtoRegE <= MemtoRegD;
            ALUSrcE <= ALUSrcD;
            CondE <= CondD;
            RD1E <= RD1D;
            RD2E <= RD2D;
            ExtImmE <= ExtImmD;
            WA3E <= WA3D;
            RA1E <= RA1D;
            RA2E <= RA2D;
        end
    end
```

Note that: ***FlushE*** from Hazard Unit gives the ***Clear*** signal of this register; ***CLK*** from input signal gives the ***Clock*** signal of the register.

Execute / Memory Register: Reg_E_M

```
/*************************************************
-----------------------------------------------------------
*Register Reg_E_M: connect Execute Stage and Memory Stage
-----------------------------------------------------------
*************************************************/
    always @ (posedge CLK) begin
            RegWriteM <= RegWriteE;
            MemWriteM <= MemWriteE;
            MemtoRegM <= MemtoRegE;
            ALUResultM <= ALUResultE;
            WriteDataM <= WriteDataE;
            WA3M <= WA3E;
            RA1M <= RA1E;
            RA2M <= RA2E;
    end
```

Note that: ***CLK*** from input signal gives the ***Clock*** signal of the register.

Memory / Writeback Register: Reg_M_W

```
/************************************************
------------------------------------------------------------
*Register Reg_M_W: connect Memory Stage and Write Stage
------------------------------------------------------------
************************************************/
    always @ (posedge CLK) begin
            RegWriteW <= RegWriteM;
            MemtoRegW <= MemtoRegM;
            ReadDataW <= ReadDataM;
            ALUResultW <= ALUResultM;
            WA3W <= WA3M;
    end
```

Note that: ***CLK*** from input signal gives the ***Clock*** signal of the register.

Elaborate Design of Test

Design of Instructions

The carefully designed instruction flows to test my CPU core are shown below in **Verilog**.

```
//------------------------------------------------------------------
// Instruction Memory
//------------------------------------------------------------------
integer i;
initial begin
        //initialize R0 to R10
            INSTR_MEM[0] = 32'hE2000000;    //R0 = 0
            //4 NOPs to fill the pipeline at the beginning.
            INSTR_MEM[1] = 32'h00000000;    //ANDEQ R0, R0, R0
            INSTR_MEM[2] = 32'h00000000;    //ANDEQ R0, R0, R0
            INSTR_MEM[3] = 32'h00000000;    //ANDEQ R0, R0, R0
            INSTR_MEM[4] = 32'h00000000;    //ANDEQ R0, R0, R0
            //continue to initialize RF:
            INSTR_MEM[5] = 32'hE2801001;    //R1 = 1
            INSTR_MEM[6] = 32'hE2802002;    //R2 = 2
            INSTR_MEM[7] = 32'hE2803003;    //R3 = 3
            INSTR_MEM[8] = 32'hE2804004;    //R4 = 4
            INSTR_MEM[9] = 32'hE2805005;    //R5 = 5
            INSTR_MEM[10] = 32'hE2806006; //R6 = 6
            INSTR_MEM[11] = 32'hE2807007; //R7 = 7
            INSTR_MEM[12] = 32'hE2808008; //R8 = 8
            INSTR_MEM[13] = 32'hE2809009; //R9 = 9
            INSTR_MEM[14] = 32'hE280A00A; //R10= 10

        //Instruction Flows to check Hazard handling
            //Data Hazard elimination by Data Forwarding
            INSTR_MEM[15] = 32'hE0841005; //ADD R1, R4, R5  ; R1 will be Written
            INSTR_MEM[16] = 32'hE0018003; //AND R8, R1, R3  ; R1 RAW, MEM to EXE
forwarding
            INSTR_MEM[17] = 32'hE1869001; //ORR R9, R6, R1  ; R1 RAW, WB to EXE
forwarding
            INSTR_MEM[18] = 32'hE041A007; //SUB R10, R1, R7 ; R1 RAW, RF write
in different edge
            INSTR_MEM[19] = 32'hE5941000; //LDR R1, [R4]; R1 will be Witten
```

```
            INSTR_MEM[20] = 32'hE5831000; //STR R1, [R3]; R1 RAW, MEM(WB) to MEM
forwarding

            //Data Hazard elimination by Stall and Flush
            INSTR_MEM[21] = 32'hE5941001; //LDR R1, [R4, #1]
            INSTR_MEM[22] = 32'hE0835001; //ADD R5, R3, R1      ; R1 Load and
use
            INSTR_MEM[23] = 32'hE0019008; //AND R9, R1, R8      ; R1 Load and
use
            INSTR_MEM[24] = 32'hE041A007; //SUB R10, R1, R7

            //Control Hazard elimination by early BTA and flush
            INSTR_MEM[25] = 32'hEA000005; //B BTA
            INSTR_MEM[26] = 32'hE2800001; //ADD R0, R0, #1      ; in
            INSTR_MEM[27] = 32'hE2811001; //ADD R1, R1, #1      ; in
            INSTR_MEM[28] = 32'hE2422001; //SUB R2, R2, #1      ; not execute
            INSTR_MEM[29] = 32'hE2833001; //ADD R3, R3, #1      ; not execute
            INSTR_MEM[30] = 32'hE2801001; //ADD R1, R0, #1      ; not execute
            INSTR_MEM[31] = 32'hE2802002; //ADD R2, R0, #2      ; not execute
            //BTA
            INSTR_MEM[32] = 32'hE2803003; //ADD R3, R0, #3   , R3=R0+3=4
            INSTR_MEM[33] = 32'hE2804004; //ADD R4, R0, #4   , R4=R0+4=5
            INSTR_MEM[34] = 32'hE2805005; //ADD R5, R0, #5   , R5=R0+5=6
            INSTR_MEM[35] = 32'hE2806006; //ADD R6, R0, #6   , R6=R0+6=7
            INSTR_MEM[36] = 32'hE2807007; //ADD R7, R0, #7   , R7=R0+7=8
            INSTR_MEM[37] = 32'hE2808008; //ADD R8, R0, #8   , R8=R0+8=9
            for(i = 38; i < 128; i = i+1) begin
                INSTR_MEM[i] = 32'h0;
            end
end
```

Initialization and Filling the Pipeline

This part corresponding to *INSTR_MEM[0] ~ INSTR_MEM[14]*. After these instruction flows, R0 ~ R10 stored the value of 0 ~ 10, respectively.

Data Hazard elimination by Data Forwarding

This part corresponding to *INSTR_MEM[15] ~ INSTR_MEM[20]*. Without handling, R1 will has the data hazard for the following 4 cases, shown in Fig. 3:

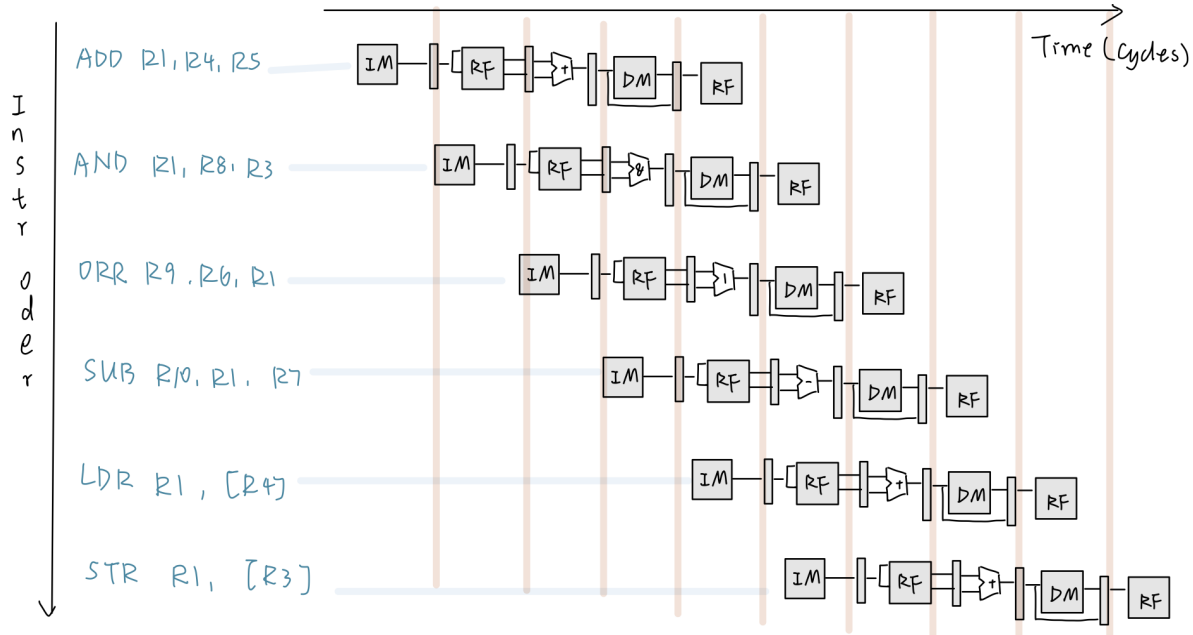


Fig. 3: the data hazard instruction flows in pipeline, handled by data forwarding and different edges reading and writing.

① handling by *Memory* forwarding to *Execute* stage.

② handling by *Writeback* forwarding to *Execute* stage.

③ handling by setting different clk edge, reading in the second half of the cycle and writing in the first half.

④ handling by *Writeback* forwarding to *Memory* stage.

Data Hazard elimination by Stalling and Flushing

This part corresponding to *INSTR_MEM[21] ~ INSTR_MEM[24]*. The instruction flows has the hazard of "Load and Use" data hazard. Without handling, R1 is where the hazard has. The following Fig. 4 shows the detailed data flow and handling method.

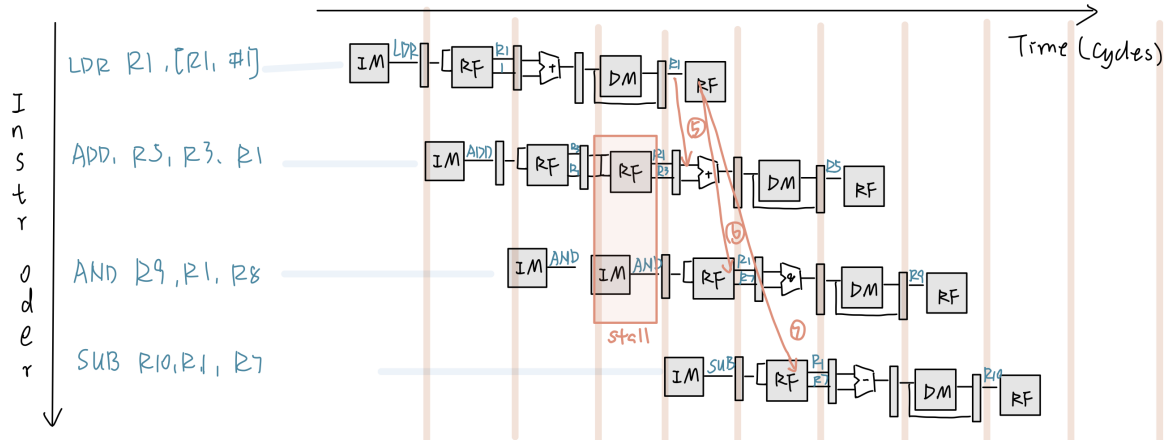


Fig. 4: the data hazard instruction flows in pipeline, handled by data "stalling and flushing" and "different edges reading and writing".

⑤ handling by a. stalling and b. *Writeback* forwarding to *Execute* stage.

⑥ handing by a. stalling and b. setting different clk edge, reading in the second half of the cycle and writing in the first half.

⑦ handing by stalling.

Control Hazard elimination by early BTA and flushing

This part corresponding to *INSTR_MEM[25] ~ INSTR_MEM[37]*. The instruction flows contains the control hazard at *INSTR_MEM[26], INSTR_MEM[27]*. With early BTA, the hazard can only happens in two cycles, which minimizes the cycle consumption. Example is shown in Fig. 5.

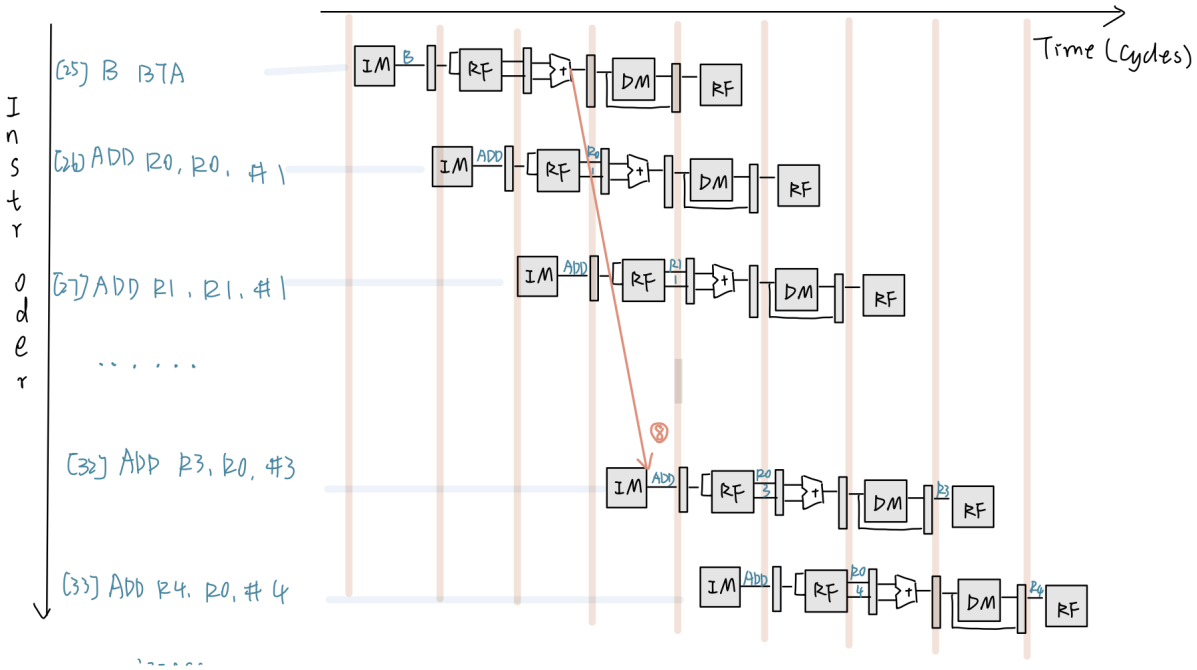


Fig. 5: the control hazard instruction flows in pipeline, handled by early BTA and flushing(not shown in flows).

⑧ handling by a. early BTA and b. flushing.

Simulations

The overall simulation results are shown in Fig. 6, which clarify the signals of Register File and Data Memory. The first 16 lines combine as Register File; The last several lines refer to part of Data Memory. To check whether it is correct, more analysis will be given in the following picture.

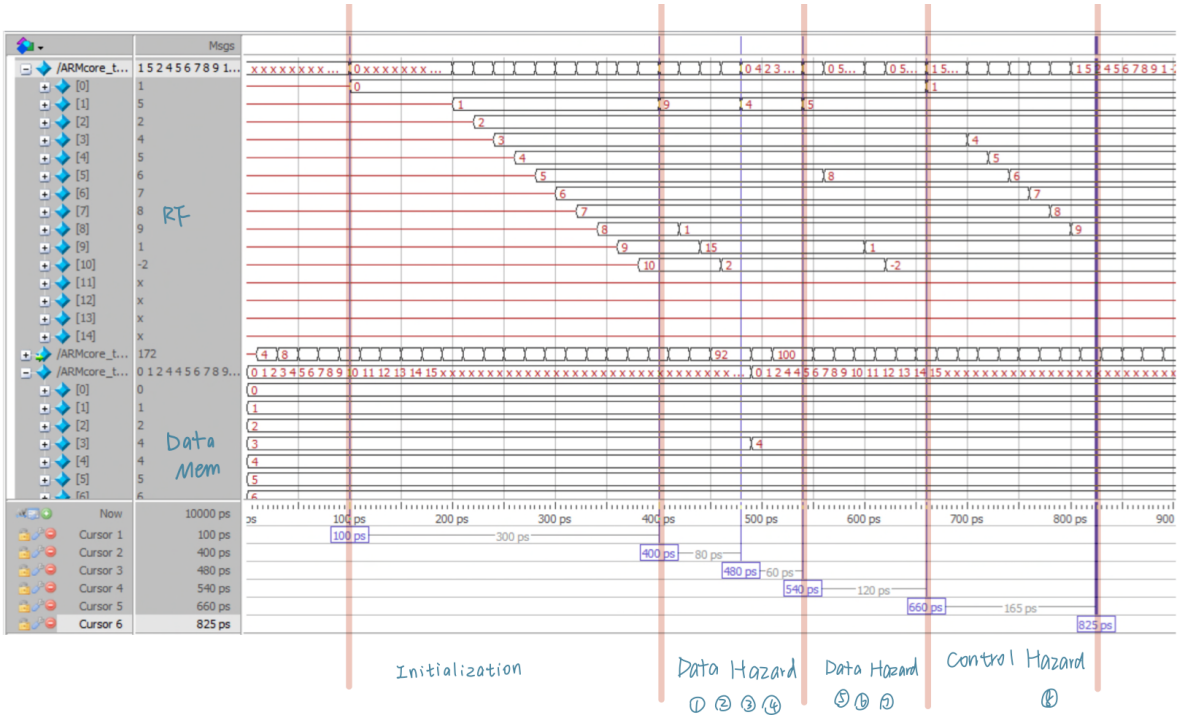


*Fig. 6: Overall simulation results for the designed instruction flow, which is shown in **Design of Instructions** part.*

The corresponding label has been given in the lower part of the picture. It is clear that this Pipelined processor works correctly, and can deal with the "Data Hazard" and "Control Hazard" with specific methods.

Source Codes

Note that: *These "*.v" files are also in Code Folder.*

Top Module

```
/*************************************************************
*This is top module of the Pipelined ARM core.
*There are 9 modules and some multipliers for connections.
*File        : Pipelined_Processor.v
*Author      : Shuo Feng
*Class       : SME 309
*DateTime    : 2021.12.21(V1.0)
*************************************************************/

module Pipelined_Processor(
    input wire CLK,
    input wire Reset    );


//Fetch Stage
    //connect to PC
    wire [31:0] current_PCF;
    wire [31:0] PC_Plus_4F;
        //connect to instr_mem and Reg_F_D
    wire [31:0] InstrF;


//Decode Stage
    //connect from Reg_F_D
    reg [31:0]  InstrD;
    //connect to Control Unit
    wire    [1:0] Op;     //instrD[27:26]
    wire    [5:0] Funct;     //instrD[25:20]
    wire    [3:0] Rd;        //instrD[15:12]
    wire            PCSD;
    wire            RegWD;
    wire            MemWD;
    wire    [1:0] FlagWD;
    wire  [1:0] ALUControlD;
    wire            MemtoRegD;
    wire            ALUSrcD;
    wire    [1:0] ImmSrcD;
    wire  [1:0] RegSrcD;
    //connect to RF
    wire [3:0] RA1D;
    wire [3:0] RA2D;
    wire [31:0] PC_Plus_8D;
    wire [31:0] RD1D;    //from RegisterFile RD1
    wire [31:0] RD2D;   //from RegisterFile RD2
    //connect to Extend
    wire [31:0] ExtImmD;//from Extend
    //connect to Reg_D_E
    wire [3:0] CondD;
    wire [3:0] WA3D;
```

```
//Execution Stage
    //connect to Cond Unit
    reg         PCSE;
    reg         RegWE;
    reg         MemWE;
    reg [1:0] FlagWE;
    reg   [3:0] CondE;
    wire    [31:0]ALUResultE;
    wire            PCSrcE;
    wire            RegWriteE;
    wire            MemWriteE;
    //connect to ALU
    reg   [1:0] ALUControlE;
    reg  [31:0] SrcAE;
    wire [31:0] SrcBE;
    //intermedia signal for ALU
    reg         ALUSrcE;
    reg [31:0]ExtImmE;
    reg   [31:0]RD1E;
    reg   [31:0]RD2E;
    wire    [3:0] ALUFlags;  //from ALU
    //connect to Reg_E_W
    reg   [3:0] WA3E;
    reg     [3:0] RA1E;
    reg [3:0] RA2E;
    reg         MemtoRegE;//reg?
    reg     [31:0]WriteDataE;



//Memory Stage
    reg         RegWriteM;
    reg         MemWriteM;
    reg         MemtoRegM;
    reg [31:0]ALUResultM;
    reg [31:0]WriteDataM;
    reg   [3:0] WA3M;//reg?
    wire    [31:0]ReadDataM;
    wire  [31:0]WDM;
    reg   [3:0] RA1M;
    reg [3:0] RA2M;



//Write Back Stage
    reg     [31:0]ReadDataW;
    reg [31:0]ALUResultW;//reg?
    reg   [3:0] WA3W;//reg?
    wire  [31:0]ResultW; //equals ReadData or ALUResult, controled by MemtoReg
    reg         MemtoRegW;
    reg         RegWriteW;


//Hazard Unit Signals
    wire [1:0] ForwardAE;
    wire [1:0] ForwardBE;
    wire          ForwardM;
    wire          FlushE;
```

```
    wire          StallD;
    wire          StallF;
    wire          FlushD;


//generate signals Op, Funct, Rd for Control Unit
    assign Op = InstrD[27:26];
    assign Funct = InstrD[25:20];
    assign Rd = InstrD[15:12];



//generete signal for Cond Unit
    assign CondD = InstrD[31:28];



//generate signals RA1D, RA2D and PC_Plus_8 for Register File
    assign RA1D = (RegSrcD[0])?4'b1111:InstrD[19:16];
    assign RA2D = (RegSrcD[1])?InstrD[15:12]:InstrD[3:0];
    assign WA3D = InstrD[15:12];
    assign PC_Plus_8D = PC_Plus_4F;



//generate signal SrcAE, SrcBE for ALU
    always@(*) begin
        case(ForwardAE)
            2'b00 : SrcAE = RD1E;
            2'b01 : SrcAE = ResultW;
            2'b10 : SrcAE = ALUResultM;
            default: SrcAE = 32'hxxxxxxxx;
        endcase
    end

    always@(*) begin
        case(ForwardBE)
            2'b00 : WriteDataE = RD2E;
            2'b01 : WriteDataE = ResultW;
            2'b10 : WriteDataE = ALUResultM;
            default: WriteDataE = 32'hxxxxxxxx;
        endcase
    end
    assign SrcBE = (ALUSrcE)?ExtImmE:WriteDataE;



//generate signal WDM for DataMem
    assign WDM = (ForwardM)?ResultW:WriteDataM;


//generate signal Result for PC and RF
    assign ResultW = (MemtoRegW)?ReadDataW:ALUResultW;



/***************************************************
```

```
*Program Counter: Fetch Stage, gives the instruction address
************************************************/
    ProgramCounter PC0(
        //input
        .CLK(CLK),
        .Reset(Reset),
        .PCSrc(PCSrcE),  // from Control Unit
        .Result(ALUResultE),//ALUResult from ALU or ReadData from data_mem
        .StallF(StallF),

        //output
        .current_PC(current_PCF),//output reg [31:0]
        .PC_Plus_4(PC_Plus_4F)//output [31:0]
);




/************************************************
*Instruction Memory: Fetch Stage, gives 32-bit Instruction to Control Unit
************************************************/
    instr_mem InstructionMem0(
        //input
        .PC(current_PCF),//input wire [31:0],from ProgramCounter

        //output
        .Instr(InstrF)//output wire [31:0]
);



/************************************************
------------------------------------------------------------------
Register Reg_F_D: connect Fetch Stage and Decode Stage
------------------------------------------------------------------
************************************************/
    always @ (posedge CLK) begin
        if(FlushD) InstrD <= 32'd0;
        else if(~StallD) InstrD <= InstrF;
        else InstrD <= InstrD;
    end



/************************************************
*Control Unit: Decode Stage, gives the control signals: PCsrc, MemtoReg,
MemWrite,
*ALUControl, ALUSrc, ImmSrc, RegWrite.
************************************************/
    ControlUnit ControlUnit0(
        //input
        .CLK(CLK),
        .Op(Op),
        .Funct(Funct),
        .Rd(Rd),
```

```
        //output
        .MemtoRegD(MemtoRegD),
        .MemWD(MemWD),
        .ALUSrcD(ALUSrcD),
        .ImmSrcD(ImmSrcD),
        .RegWD(RegWD),
        .RegSrcD(RegSrcD),
        .ALUControlD(ALUControlD),
        .PCSD(PCSD)
    );



/************************************************
*Register File: Decode Stage, 2^4 registers, each register stores 32-bit data
************************************************/
    RegisterFile RegisterFile0 (
        //input
        .CLK(~CLK),
        .WE3(RegWriteW),//high active, from Control Unit
        .A1(RA1D),//input [3:0], Read index1
        .A2(RA2D),//input [3:0], Read index2
        .A3(WA3W),   //input [3:0], from Instr Write index
        .WD3(ResultW),//input [31:0], Write data
        .R15(PC_Plus_8D),//[31:0], R15 Data in

        //output
        .RD1(RD1D),//Read data1,output reg [31:0]
        .RD2(RD2D) //Read data2,output reg [31:0]
    );



/************************************************
*Extend: Decode Stage, gives 32-bit ExtImm
************************************************/
    Extend Extend0(
        //input
        .ImmSrc(ImmSrcD),//input [1:0], from Control Unit
        .InstrImm(InstrD[23:0]),//input [23:0], from Instr

        //output
        .ExtImm(ExtImmD)// [31:0]
    );



/************************************************
------------------------------------------------------------
*Register Reg_D_E: connect Decode Stage and Execute Stage
------------------------------------------------------------
************************************************/
    always @ (posedge CLK) begin//
        if(FlushE) begin
            PCSE <= 1'b0;
```

```
            RegWE <= 1'b0;
            MemWE <= 1'b0;
            FlagWE <= 1'b0;
            end
        else begin
            PCSE <= PCSD;
            RegWE <= RegWD;
            MemWE <= MemWD;
            FlagWE <= FlagWD;
            ALUControlE <= ALUControlD;
            MemtoRegE <= MemtoRegD;
            ALUSrcE <= ALUSrcD;
            CondE <= CondD;
            RD1E <= RD1D;
            RD2E <= RD2D;
            ExtImmE <= ExtImmD;
            WA3E <= WA3D;
            RA1E <= RA1D;
            RA2E <= RA2D;
        end

    end




/*************************************************
*Conditional Unit: Exectue Stage
*************************************************/
    CondUnit CondUnit0(
        //input
        .CLK(CLK),
        .PCSE(PCSE),
        .RegWE(RegWE),
        .MemWE(MemWE),
        .FlagWE(FlagWE),
        .CondE(CondE),
        .ALUFlags(ALUFlags),

        //output
        .PCSrcE(PCSrcE),
        .RegWriteE(RegWriteE),
        .MemWriteE(MemWriteE)
    );




/*************************************************
*ALU: Execute Stage, gives ALUresult
*************************************************/
    ALU ALU0(
        //input
        .A(SrcAE),
        .B(SrcBE),
        .ALUControl(ALUControlE),//[1:0], from Control Unit

        //output
```

```
        .ALUResult(ALUResultE),//output reg [31:0]
        .ALUFlags(ALUFlags)//output [3:0], N, Z, C, V,
    );



/***********************************************
------------------------------------------------------------
*Register Reg_E_M: connect Execute Stage and Memory Stage
------------------------------------------------------------
***********************************************/
    always @ (posedge CLK) begin
            RegWriteM <= RegWriteE;
            MemWriteM <= MemWriteE;
            MemtoRegM <= MemtoRegE;
            ALUResultM <= ALUResultE;
            WriteDataM <= WriteDataE;
            WA3M <= WA3E;
            RA1M <= RA1E;
            RA2M <= RA2E;
    end



/***********************************************
*Data Memory: Memory Stage, gives ReadData and can write data.
***********************************************/
    data_mem DataMem0(
        //input
        .CLK(CLK),
        .Address(ALUResultM),//input wire   [31:0], from ALUResult
        //input write port
        .WE(MemWriteM),      //Write Enable, from Control Unit
        .WD(WDM),  //Write Data, [31:0], from RF

        //output read port
        .ReadData(ReadDataM)//output wire [31:0]
    );



/***********************************************
------------------------------------------------------------
*Register Reg_M_W: connect Memory Stage and Write Stage
------------------------------------------------------------
***********************************************/
    always @ (posedge CLK) begin
            RegWriteW <= RegWriteM;
            MemtoRegW <= MemtoRegM;
            ReadDataW <= ReadDataM;
            ALUResultW <= ALUResultM;
            WA3W <= WA3M;
    end
```

```
/************************************************
*Hazard Unit: detects the Hazard.
************************************************/
    HazardUnit HazardUnit0(
        //input
        .RA1E(RA1E),
        .RA1D(RA1D),
        .RA2E(RA2E),
        .RA2D(RA2D),
        .RA2M(RA2M),
        .WA3M(WA3M),
        .WA3W(WA3W),
        .WA3E(WA3E),
        .RegWriteM(RegWriteM),
        .RegWriteW(RegWriteW),
        .MemWriteM(MemWriteM),
        .MemtoRegE(MemtoRegE),
        .PCSrcE(PCSrcE),
        .MemtoRegW(MemtoRegW),
        .RegWriteE(RegWriteE),
        .Reset(Reset),
        .CLK(CLK),

        //output
        .ForwardAE(ForwardAE),
        .ForwardBE(ForwardBE),
        .ForwardM(ForwardM),
        .FlushE(FlushE),
        .StallD(StallD),
        .StallF(StallF),
        .FlushD(FlushD)
    );
endmodule
```

Top Module Test Bench

```
/************************************************************
*This is the test bench of the top module of the ARM core.
*File        : ARMcore_top_tb.v
*Author      : Shuo Feng
*Class       : SME 309
*DateTime    : 2021.11.16(V1.0)
************************************************************/

module ARMcore_top_tb();

    reg CLK;
    reg Reset;

    initial begin
        CLK = 1'b0;
        forever #10 CLK = ~CLK;//unit is ps
    end

    initial begin
        #0  Reset = 1'b1;
```

```
        #25 Reset = 1'b0;
    end

    Pipelined_Processor PipelinedARMCore0(
        .CLK(CLK),
        .Reset(Reset)
    );

    endmodule
```

Modules in CPU

HazardUnit.v

```
/*****************************************************************
*This is a design of Hazard detection unit
*File     : HazardUnit.v
*Author   : Shuo Feng
*Class    : SME 309
*DateTime : 2021.12.11 (V1.0)
*DateTime : 2021.12.20 (V2.0)
*What's new: I changed some of control signals from sequential logic
*to combinatorial logic.
*****************************************************************/

module HazardUnit(
    input [3:0] RA1E,
    input [3:0] RA1D,
    input [3:0] RA2E,
    input [3:0] RA2D,
    input [3:0] RA2M,
    input [3:0] WA3M,
    input [3:0] WA3W,
    input [3:0] WA3E,
    input RegWriteM,
    input RegWriteW,
    input MemWriteM,
    input MemtoRegE,
    input PCSrcE,
    input MemtoRegW,
    input RegWriteE,
    input Reset,
    input CLK,


    //output
    output reg [1:0] ForwardAE,
    output reg [1:0] ForwardBE,
    output reg ForwardM,
    output reg FlushE,
    output reg StallD,
    output reg StallF,
    output reg FlushD

    );

    //Data Forwarding
```

```
    wire Match_1E_M;
    wire Match_2E_M;
    wire Match_1E_W;
    wire Match_2E_W;
    //Stalling
    wire Match_12D_E;
    wire Idrstall;


    //Data Forwarding: Memory to Execute
    assign Match_1E_M = (Reset)?1'b0:(RA1E == WA3M);
    assign Match_2E_M = (Reset)?1'b0:(RA2E == WA3M);

    //Data Forwarding: Writeback to Execute
    assign Match_1E_W = (Reset)?1'b0:(RA1E == WA3W);
    assign Match_2E_W = (Reset)?1'b0:(RA2E == WA3W);

    //Data Forwarding for SrcA
    always@(*)begin
        if        (Match_1E_M & RegWriteM)  ForwardAE = (Reset)?2'b0:2'b10;//MEM
to EXE
        else if (Match_1E_W & RegWriteW) ForwardAE = (Reset)?2'b0:2'b01;//WB to
EXE
        else                                        ForwardAE = (Reset)?
2'b0:2'b00;//Normal Path

    end

    //Data Forwarding for SrcB
    always@(*)begin
        if        (Match_2E_M & RegWriteM)  ForwardBE = (Reset)?2'b0:2'b10;//MEM
to EXE
        else if (Match_2E_W & RegWriteW) ForwardBE = (Reset)?2'b0:2'b01;//WB to
EXE
        else                                        ForwardBE = (Reset)?
2'b0:2'b00;//Normal Path

    end


    //Data Forwarding: Mem (Writeback) to Memory, happends when LDR -> STR
    always @(posedge CLK) begin
        ForwardM <= (Reset)?1'b0:((RA2M == WA3W) & MemWriteM & MemtoRegW &
RegWriteW);
    end


    //Stalling: Load and Use Hazard
    assign Match_12D_E =(Reset)?1'b0:( (RA1D == WA3E) || (RA2D == WA3E));
    assign Idrstall = (Reset)?1'b0:(Match_12D_E & MemtoRegE & RegWriteE);
    always @(posedge CLK) begin
        StallF <=(Reset)?1'b0:Idrstall;
    end
    always @(posedge CLK) begin
        StallD <=(Reset)?1'b0:Idrstall;
    end
    always @(posedge CLK) begin
        FlushE <=(Reset)?1'b0:(Idrstall || PCSrcE);
```

```
    end


    //Flushing: Control Hazard
    always @(posedge CLK) begin
    FlushD <= (Reset)?1'b0:PCSrcE;
    end


endmodule
```

ProgramCounter.v

```
/*************************************************************
*This is a design of program counter
*File     : ProgramCounter.v
*Author   : Shuo Feng
*Class    : SME 309
*DateTime : 2021.12.20 (V1.0)
*************************************************************/

module ProgramCounter(
    input CLK,
    input Reset,
    input PCSrc,  //control signal
    input [31:0] Result,
    input StallF,

    output reg [31:0] current_PC,
    output [31:0] PC_Plus_4
);

assign PC_Plus_4=current_PC + 32'd4;

    always@(posedge CLK)
        begin

        if(Reset)
        current_PC<=32'b0;
        else if(StallF)
        current_PC<=current_PC;
        else if(PCSrc)
        current_PC<=Result;
        else
        current_PC<=PC_Plus_4;
        end


endmodule
```

instr_mem.v

```
/*************************************************************
*This is the Instruction Memory block, which is a part of the processor,
*and this instr_mem.v contains instruction flows with data hazard and
*control hazard. (What's more, they can be handled by hazard unit!)
*File        : instr_mem.v
```

```
*Author      : Shuo Feng
*Class       : SME 309
*DateTime    : 2021.12.20(V1.0)
**************************************************************/

module instr_mem(
    input wire  [31:0]  PC,
    output wire [31:0]  Instr
);

    reg [31:0]  INSTR_MEM[0:127];//only need 7 bit to specify the address, so pc


//-----------------------------------------------------------------
// Instruction Memory
//-----------------------------------------------------------------
integer i;

initial begin
        //initialize R0 to R10
            INSTR_MEM[0] = 32'hE2000000;    //R0 = 0
            //4 NOPs to fill the pipeline at the beginning.
            INSTR_MEM[1] = 32'h00000000;    //ANDEQ R0, R0, R0
            INSTR_MEM[2] = 32'h00000000;    //ANDEQ R0, R0, R0
            INSTR_MEM[3] = 32'h00000000;    //ANDEQ R0, R0, R0
            INSTR_MEM[4] = 32'h00000000;    //ANDEQ R0, R0, R0

            INSTR_MEM[5] = 32'hE2801001;    //R1 = 1
            INSTR_MEM[6] = 32'hE2802002;    //R2 = 2
            INSTR_MEM[7] = 32'hE2803003;    //R3 = 3
            INSTR_MEM[8] = 32'hE2804004;    //R4 = 4
            INSTR_MEM[9] = 32'hE2805005;    //R5 = 5
            INSTR_MEM[10] = 32'hE2806006; //R6 = 6
            INSTR_MEM[11] = 32'hE2807007; //R7 = 7
            INSTR_MEM[12] = 32'hE2808008; //R8 = 8
            INSTR_MEM[13] = 32'hE2809009; //R9 = 9
            INSTR_MEM[14] = 32'hE280A00A; //R10= 10

        //Instruction Flows to check Hazard handling
            //Data Hazard elimination by Data Forwarding
            INSTR_MEM[15] = 32'hE0841005; //ADD R1, R4, R5      ; R1 will be
Written
            INSTR_MEM[16] = 32'hE0018003; //AND R8, R1, R3      ; R1 RAW, MEM to
EXE forwarding
            INSTR_MEM[17] = 32'hE1869001; //ORR R9, R6, R1      ; R1 RAW, WB to
EXE forwarding
            INSTR_MEM[18] = 32'hE041A007; //SUB R10, R1, R7     ; R1 RAW, RF
write in different edge
            INSTR_MEM[19] = 32'hE5941000; //LDR R1, [R4]        ; R1 will be
Witten
            INSTR_MEM[20] = 32'hE5831000; //STR R1, [R3]        ; R1 RAW,
MEM(WB) to MEM forwarding

            //Data Hazard elimination by Stall and Flush
            INSTR_MEM[21] = 32'hE5941001; //LDR R1, [R4, #1]
            INSTR_MEM[22] = 32'hE0835001; //ADD R5, R3, R1      ; R1 Load and
use
```

```
            INSTR_MEM[23] = 32'hE0019008; //AND R9, R1, R8      ; R1 Load and
use
            INSTR_MEM[24] = 32'hE041A007; //SUB R10, R1, R7

            //Control Hazard elimination by early BTA and flush
            INSTR_MEM[25] = 32'hEA000005; //B BTA
            INSTR_MEM[26] = 32'hE2800001; //ADD R0, R0, #1      ; in
            INSTR_MEM[27] = 32'hE2811001; //ADD R1, R1, #1      ; in
            INSTR_MEM[28] = 32'hE2422001; //SUB R2, R2, #1      ; not execute
            INSTR_MEM[29] = 32'hE2833001; //ADD R3, R3, #1      ; not execute
            INSTR_MEM[30] = 32'hE2801001; //ADD R1, R0, #1      ; not execute
            INSTR_MEM[31] = 32'hE2802002; //ADD R2, R0, #2      ; not execute

            //BTA
            INSTR_MEM[32] = 32'hE2803003; //ADD R3, R0, #3  , R3=R0+3=4
            INSTR_MEM[33] = 32'hE2804004; //ADD R4, R0, #4  , R4=R0+4=5
            INSTR_MEM[34] = 32'hE2805005; //ADD R5, R0, #5  , R5=R0+5=6
            INSTR_MEM[35] = 32'hE2806006; //ADD R6, R0, #6  , R6=R0+6=7
            INSTR_MEM[36] = 32'hE2807007; //ADD R7, R0, #7  , R7=R0+7=8
            INSTR_MEM[37] = 32'hE2808008; //ADD R8, R0, #8  , R8=R0+8=9
            for(i = 38; i < 128; i = i+1) begin
                INSTR_MEM[i] = 32'h0;
            end
end


    assign Instr = ( (PC >= 32'h00000000) & (PC <= 32'h000001FC) ) ? // To check
if PC is in the valid range, assuming 128 word memory.
                INSTR_MEM[PC[8:2]] : 32'h00000000 ; //7 bit, no problem

endmodule
```

ControlUnit.v

```
/*****************************************************************
*This is Control Unit block for Pipelined Processor,different from Sigle-Cycle
Processor, which contains Decoder and Conditional Logic.
*This instantiates one Decoder (Decoder1) and one CondLogic (CondLogic1)
*File        : ControlUnit.v
*Author      : Shuo Feng
*Class       : SME 309
*DateTime    : 2021.12.11(V1.0)
*****************************************************************/

module ControlUnit(
    input CLK,           //for CondLogic
    input [1:0] Op,      //instrD[27:26]
    input [5:0] Funct,  //instrD[25:20]
    input [3:0] Rd,     //instrD[15:12]

    output  MemtoRegD,
    output  MemWD,          //to CondUnit
   output  ALUSrcD,
   output  [1:0] ImmSrcD,
   output  RegWD,           //to CondUnit
   output  [1:0] RegSrcD,
   output  [1:0] ALUControlD,
```

```
    output  PCSD,            //to CondUnit
     output [1:0] FlagWD //to CondUnit
     );

     Decoder Decoder1(
     //input
      .Op(Op),
      .Funct(Funct),
      .Rd(Rd),

      //output
      .MemtoReg(MemtoRegD),
     .MemW(MemWD),
     .ALUSrc(ALUSrcD),
     .ImmSrc(ImmSrcD),
     .RegW(RegWD),
     .RegSrc(RegSrcD),
     .ALUControl(ALUControlD),
     .FlagW(FlagWD),
     .PCS(PCSD)
     );

endmodule




/*****************************************************************
*This is a Decoder
*File         : Decoder.v
*Author       : Shuo Feng
*Class        : SME 309
*DateTime     : 2021.10.19(V1.0) for single cycle processor
*               2021.12.11(V2.0) for pipelined processor
*****************************************************************/

//Cases of MainD
`define DP_reg1 4'b0001
`define DP_reg2 4'b0000
`define DP_imm1 4'b0010
`define DP_imm2 4'b0011
`define STR1 4'b0100
`define STR2 4'b0110
`define LDR1 4'b0101
`define LDR2 4'b0111
`define B1 4'b1000
`define B2 4'b1001
`define B3 4'b1010
`define B4 4'b1011

//Cases of ALUD
`define NotDP1 5'b00000
`define NotDP2 5'b00001
`define NotDP3 5'b00010
`define NotDP4 5'b00011
`define NotDP5 5'b00100
```

```
`define NotDP6 5'b00101
`define NotDP7 5'b00110
`define NotDP8 5'b00111
`define NotDP9 5'b01000
`define NotDP10 5'b01001
`define NotDP11 5'b01010
`define NotDP12 5'b01011
`define NotDP13 5'b01100
`define NotDP14 5'b01101
`define NotDP15 5'b01110
`define NotDP16 5'b01111
`define ADD 5'b10100
`define SUB 5'b10010
`define AND 5'b10000
`define ORR 5'b11100


module Decoder(
     input [1:0] Op,
     input [5:0] Funct,
     input [3:0] Rd,


    output reg MemtoReg,
    output reg MemW,
    output reg ALUSrc,
    output reg [1:0] ImmSrc,
    output reg RegW,
    output reg [1:0] RegSrc,
    output reg [1:0] ALUControl,
    output reg [1:0] FlagW,
    output reg PCS
    );

   reg ALUOp ;
   reg Branch ;

    wire [3:0] MainD;
    wire S;//Funct0
    reg [4:0] ALUD;
    assign MainD = {Op,Funct[5],Funct[0]};
    assign S=Funct[0];


    //Main Decoder
    always @(*)begin
        case(MainD)
            `DP_reg1: begin Branch = 1'b0; MemtoReg = 1'b0; MemW = 1'b0; ALUSrc
= 1'b0;
                      ImmSrc = 2'bxx; RegW = 1'b1; RegSrc = 2'b00; ALUOp =
1'b1;
                      end
            `DP_reg2: begin Branch = 1'b0; MemtoReg = 1'b0; MemW = 1'b0; ALUSrc
= 1'b0;
                      ImmSrc = 2'bxx; RegW = 1'b1; RegSrc = 2'b00; ALUOp =
1'b1;
                      end
```

```
            `DP_imm1: begin
                        Branch = 0; MemtoReg = 0; MemW = 0; ALUSrc = 1;
                        ImmSrc = 2'b00; RegW = 1; RegSrc = 2'bx0; ALUOp = 1'b1;
                        end
            `DP_imm2: begin
                        Branch = 0; MemtoReg = 0; MemW = 0; ALUSrc = 1;
                        ImmSrc = 2'b00; RegW = 1; RegSrc = 2'bx0; ALUOp = 1'b1;
                        end

            `STR1   : begin
                        Branch = 0; MemtoReg = 1'bx; MemW = 1; ALUSrc = 1;
                        ImmSrc = 2'b01; RegW = 0; RegSrc = 2'b10; ALUOp = 1'b0;
                        end
            `STR2   : begin
                        Branch = 0; MemtoReg = 1'bx; MemW = 1; ALUSrc = 1;
                        ImmSrc = 2'b01; RegW = 0; RegSrc = 2'b10; ALUOp = 1'b0;
                        end

            `LDR1   : begin
                        Branch = 0; MemtoReg = 1; MemW = 0; ALUSrc = 1;
                        ImmSrc = 2'b01; RegW = 1; RegSrc = 2'bx0; ALUOp = 1'b0;
                        end
            `LDR2   : begin
                        Branch = 0; MemtoReg = 1; MemW = 0; ALUSrc = 1;
                        ImmSrc = 2'b01; RegW = 1; RegSrc = 2'bx0; ALUOp = 1'b0;
                        end

            `B1     : begin
                        Branch = 1; MemtoReg = 0; MemW = 0; ALUSrc = 1;
                        ImmSrc = 2'b10; RegW = 0; RegSrc = 2'bx1; ALUOp = 1'b0;
                        end
            `B2     : begin
                        Branch = 1; MemtoReg = 0; MemW = 0; ALUSrc = 1;
                        ImmSrc = 2'b10; RegW = 0; RegSrc = 2'bx1; ALUOp = 1'b0;
                        end
            `B3     : begin
                        Branch = 1; MemtoReg = 0; MemW = 0; ALUSrc = 1;
                        ImmSrc = 2'b10; RegW = 0; RegSrc = 2'bx1; ALUOp = 1'b0;
                        end
            `B4     : begin
                        Branch = 1; MemtoReg = 0; MemW = 0; ALUSrc = 1;
                        ImmSrc = 2'b10; RegW = 0; RegSrc = 2'bx1; ALUOp = 1'b0;
                        end

            default:begin
                        Branch = 1'bx; MemtoReg = 1'bx; MemW = 1'bx; ALUSrc =
1'bx;
                        ImmSrc = 2'bxx; RegW = 1'bx; RegSrc = 2'bxx; ALUOp =
1'bx;
                    end
    endcase
    ALUD = {ALUOp, Funct[4:1]};

    //ALU Decoder
    case(ALUD)
            `NotDP1: begin
                ALUControl = 2'b00; FlagW = 2'b00; end
            `NotDP2: begin
```

```
                ALUControl = 2'b00; FlagW = 2'b00; end
            `NotDP3: begin
                ALUControl = 2'b00; FlagW = 2'b00; end
            `NotDP4: begin
                ALUControl = 2'b00; FlagW = 2'b00; end
            `NotDP5: begin
                ALUControl = 2'b00; FlagW = 2'b00; end
            `NotDP6: begin
                ALUControl = 2'b00; FlagW = 2'b00; end
            `NotDP7: begin
                ALUControl = 2'b00; FlagW = 2'b00; end
            `NotDP8: begin
                ALUControl = 2'b00; FlagW = 2'b00; end
            `NotDP9: begin
                ALUControl = 2'b00; FlagW = 2'b00; end
            `NotDP10: begin
                ALUControl = 2'b00; FlagW = 2'b00; end
            `NotDP11: begin
                ALUControl = 2'b00; FlagW = 2'b00; end
            `NotDP12: begin
                ALUControl = 2'b00; FlagW = 2'b00; end
            `NotDP13: begin
                ALUControl = 2'b00; FlagW = 2'b00; end
            `NotDP14: begin
                ALUControl = 2'b00; FlagW = 2'b00; end
            `NotDP15: begin
                ALUControl = 2'b00; FlagW = 2'b00; end
            `NotDP16: begin
                ALUControl = 2'b00; FlagW = 2'b00; end
            `ADD  : begin
                ALUControl = 2'b00;
                if(S) FlagW = 2'b11; else FlagW = 2'b00; end
            `SUB  : begin
                ALUControl = 2'b01;
                if(S) FlagW = 2'b11; else FlagW = 2'b00; end
            `AND  : begin
                ALUControl = 2'b10;
                if(S) FlagW = 2'b10; else FlagW = 2'b00; end
            `ORR  : begin
                ALUControl = 2'b11;
                if(S) FlagW = 2'b10; else FlagW = 2'b00; end
            default: begin
                ALUControl = 2'bxx; FlagW = 2'bxx; end
        endcase
    end


    //PC Logic
    always@(*)
    PCS = ((Rd == 4'b1111) & RegW) | Branch;


endmodule
```

CondUnit.v

```
/*****************************************************************
*This is a conditional logic block, which is a part of Control Unit
*File        : CondUnit.v
*Author      : Shuo Feng
*Class       : SME 309
*DateTime    : 2021.10.26(V1.0) for single cycle
*              2021.12.11(V2.0) for pipelined
*****************************************************************/

`define EQ 4'b0000
`define NE 4'b0001
`define CS_HS 4'b0010
`define CC_LO 4'b0011
`define MI 4'b0100
`define PL 4'b0101
`define VS 4'b0110
`define VC 4'b0111
`define HI 4'b1000
`define LS 4'b1001
`define GE 4'b1010
`define LT 4'b1011
`define GT 4'b1100
`define LE 4'b1101
`define AL 4'b1110



module CondUnit(
    input CLK,
    input PCSE,
    input RegWE,
    input MemWE,
    input [1:0] FlagWE,
    input [3:0] CondE,
    input [3:0] ALUFlags,

    output PCSrcE,
    output RegWriteE,
    output MemWriteE
    );

    reg CondEx;  //whether to exacute
    reg N = 0, Z = 0, C = 0, V = 0;
    wire [1:0] FlagWrite;

    //1 define the output of CondLogic
    assign PCSrcE = PCSE & CondEx;
    assign RegWriteE = RegWE & CondEx;
    assign MemWriteE = MemWE & CondEx;
    assign FlagWrite[1] = FlagWE[1] & CondEx;
    assign FlagWrite[0] = FlagWE[0] & CondEx;


    //2 flags register update
    always @(posedge CLK) begin
```

```
        if(FlagWrite[1])
            {N,Z} <= ALUFlags[3:2];
        if(FlagWrite[0])
            {C,V} <= ALUFlags[1:0];
    end

    //CondEx generate, case statement
    always @(*) begin
        case(CondE)
            `EQ   : CondEx = Z;
            `NE   : CondEx = ~Z;
            `CS_HS: CondEx = C;
            `CC_LO: CondEx = ~C;
            `MI   : CondEx = N;
            `PL   : CondEx = ~N;
            `VS   : CondEx = V;
            `VC   : CondEx = ~V;
            `HI   : CondEx = (~Z) & C;
            `LS   : CondEx = Z | (~C);
            `GE   : CondEx = ~(N^V);
            `LT   : CondEx = N ^ V;
            `GT   : CondEx = (~Z)&(~(N^V));
            `LE   : CondEx = Z|(N^V);
            `AL   : CondEx = 1'b1;
          default: CondEx = 1'bx;
          endcase
    end

endmodule
```

RegisterFile.v

```
/*****************************************************************
*This is a design of register file
*File     : RegisterFile_tb.v
*Author   : Shuo Feng
*Class    : SME 309
*DateTime : 2021.12.11 (V1.0)
*****************************************************************/

module RegisterFile (
    input CLK,
    input WE3,//high active
    input [3:0] A1,//Read index1
    input [3:0] A2,//Read index2
    input [3:0] A3,//Write index
    input [31:0] WD3,//Write data
    input [31:0] R15,//R15 Data in
    output reg [31:0] RD1,//Read data1
    output reg [31:0] RD2//Read data2
    );

    reg [31:0] RegBankCore[0:14];

    //read
    always@(*) begin
    if(A1 == 4'd15)
```

```
    RD1 = R15;
    else
    RD1 = RegBankCore[A1];
    if(A2 == 4'd15)
    RD2 = R15;
    else
    RD2 = RegBankCore[A2];
    end

    //write
    always@(posedge CLK)
    begin
    if(WE3)
    RegBankCore[A3] <= WD3;
    end

endmodule
```

Extend.v

```
/*****************************************************************
*This is an extend block, which is a part of the processor
*File        : Extend.v
*Author      : Shuo Feng
*Class       : SME 309
*DateTime    : 2021.11.16(V1.0)
*****************************************************************/

module Extend (
    input [1:0] ImmSrc,
    input [23:0] InstrImm,

    output reg [31:0] ExtImm
    );

    always@(*) begin
        case(ImmSrc)
        2'b00   : ExtImm = {24'b0,InstrImm[7:0]};//DP
        2'b01   : ExtImm = {20'b0,InstrImm[11:0]};//LDR/STR
        2'b10   : ExtImm = {{6{InstrImm[23]}},InstrImm[23:0],{2{1'b0}}};//B
        default: ExtImm = 32'hxxxxxxxx;
        endcase
    end

endmodule
```

ALU.v

```
/*****************************************************************
*This is an ALU (arithmetic logic unit).
*File        : ALU.v
*Author      : Shuo Feng
*Class       : SME 309
*DateTime    : 2021.11.2(V1.0)
*****************************************************************/
`define ADD 2'b00
`define SUB 2'b01
```

```
`define AND 2'b10
`define ORR 2'b11
module ALU(
    input [31:0] A,
    input [31:0] B,
    input [1:0] ALUControl,

    output reg [31:0] ALUResult,
    output [3:0] ALUFlags//N, Z, C, V, connect to the Conditional Logic Unit
    );

    reg Cin;
    wire Cout;
    wire [31:0] Sum;
    wire [31:0] And;
    wire [31:0] Orr;

    reg [31:0] Btem;

    //Arithmetic Operation
    //get 32-bit Sum
    Adder32bit Adder0(.a(A), .b(Btem), .Cin(Cin), .Cout(Cout), .Sum(Sum));
    //get 32-bit And
    assign And = A & B;
    //get 32-bit Orr
    assign Orr = A | B;

    //ALUFlags Generation
    always@(*) begin
    if(ALUControl[0]) begin Btem=~B; Cin = 1'b1; end
    else begin Btem=B; Cin = 1'b0; end
    end

    always@(*) begin
    case(ALUControl)
        `ADD : ALUResult = Sum;
        `SUB : ALUResult = Sum;
        `AND : ALUResult = And;
        `ORR : ALUResult = Orr;
        default: ALUResult = 32'hxxxxxxxx;
    endcase
    end

    //Z
    assign ALUFlags[2] = &(~ALUResult[31:0]);
    //N
    assign ALUFlags[3] = ALUResult[31];
    //C
    assign ALUFlags[1] = (~ALUControl[1]) & Cout;
    //V
    assign ALUFlags[0] = ((~(ALUControl[0]^A[31]^B[31]))&(A[31]^Sum[31])&
(~ALUControl[1]));

endmodule


/*************************************************************
*This is a 32-bit Ripple Carry Adder, whole upper module is ALU.v
```

```
*THere are 32 inctances of FullAdder.v
*File        : Adder32bit.v
*Author      : Shuo Feng
*Class       : SME 309
*DateTime    : 2021.11.2(V2.0)
*V1.0 is duplicated 32 full adders made up 32-bit adder.
**************************************************************/
module Adder32bit(
    input [31:0] a,
    input [31:0] b,
    input Cin,

    output Cout,
    output [31:0] Sum
    );

    wire [31:0] c;
    assign Cout=c[31];

    fa fa0(.a(a[0]),.b(b[0]),.cin(Cin),.cout(c[0]),.sum(Sum[0]));

    genvar i;
    generate
        for (i=1; i<32; i=i+1) begin: generate_block_identifier // <-- example
block name

        fa whatever_fa(.a(a[i]),.b(b[i]),.cin(c[i-1]),.cout(c[i]),.sum(Sum[i]));

        end
    endgenerate

//This is version 1, duplicated 32 full adders made up 32-bit adder

//  fa fa0(.a(a[0]),.b(b[0]),.cin(cin),.cout(c[0]),.s(s[0]));
//  fa fa1(.a(a[1]),.b(b[1]),.cin(c[0]),.cout(c[1]),.s(s[1]));
//  fa fa2(.a(a[2]),.b(b[2]),.cin(c[1]),.cout(c[2]),.s(s[2]));
//  fa fa3(.a(a[3]),.b(b[3]),.cin(c[2]),.cout(c[3]),.s(s[3]));
//  fa4 ~ fa30 .........
//  fa fa31(.a(a[31]),.b(b[31]),.cin(c[30]),.cout(c[31]),.s(s[31]));

endmodule




/**************************************************************
*This is a 1-bit full adder, whole upper module is Adder32bit.v
*File        : fa.v
*Author      : Shuo Feng
*Class       : SME 309
*DateTime    : 2021.11.2(V1.0)
**************************************************************/
module fa(
    input a,
    input b,
    input cin,

    output cout,
```

```
    output sum
    );

    assign sum = a ^ b ^ cin;
    assign cout = (a & b) | cin & (a^b);

endmodule
```

data_mem.v

```
/*************************************************************
*This is the Data Memory block, which is a part of the processor
*File        : data_mem.v
*Author      : Shuo Feng
*Class       : SME 309
*DateTime    : 2021.11.16(V1.0)
*************************************************************/

module data_mem(
        input wire         CLK,
        input wire  [31:0] Address,

        // write port
        input wire         WE,     //Write Enable
        input wire  [31:0] WD,     //Write Date

        // read port
        output wire [31:0] ReadData
    );



    reg [31:0] DATA_MEM [0:127];

    initial $readmemh (
"C:/intelFPGA/18.1/project/microcontroller/Files/data_mem(example).txt",
DATA_MEM ); //you should write your own path here



//mem write
    always @(posedge CLK) begin
        if (WE)
            DATA_MEM[Address] <= WD;
        else
            DATA_MEM[Address] <=DATA_MEM[Address];
    end

//mem read

    assign ReadData=DATA_MEM[Address][31:0];

endmodule
```