Lab7: Pipelined Implementation of RISC-V

Objectives

The objective of this lab is to complete the implementation of the RISC-V pipelined datapath prior to introducing data or control hazards handling mechanisms, and to test it on the Nexys A7 board.

This lab is organized as follows:

1. An introduction
2. Experiments to be conducted
3. Lab Report Requirements

Introduction

In the previous labs, you should have completed the Verilog modeling and simulation of the single cycle RISC-V processor according to Figure 1 assuming that we only need to support the following RV32I ISA subset:

- Memory reference: LW (I-Format), SW (S-Format)
- Arithmetic/logical: ADD, SUB, AND, OR (R-Format)
- Control transfer: BEQ (SB-Format)

The goal of this lab is to model the pipelined RISC-V datapath supporting the same instruction subset following the design you studied in the lectures and shown in Figure 2.

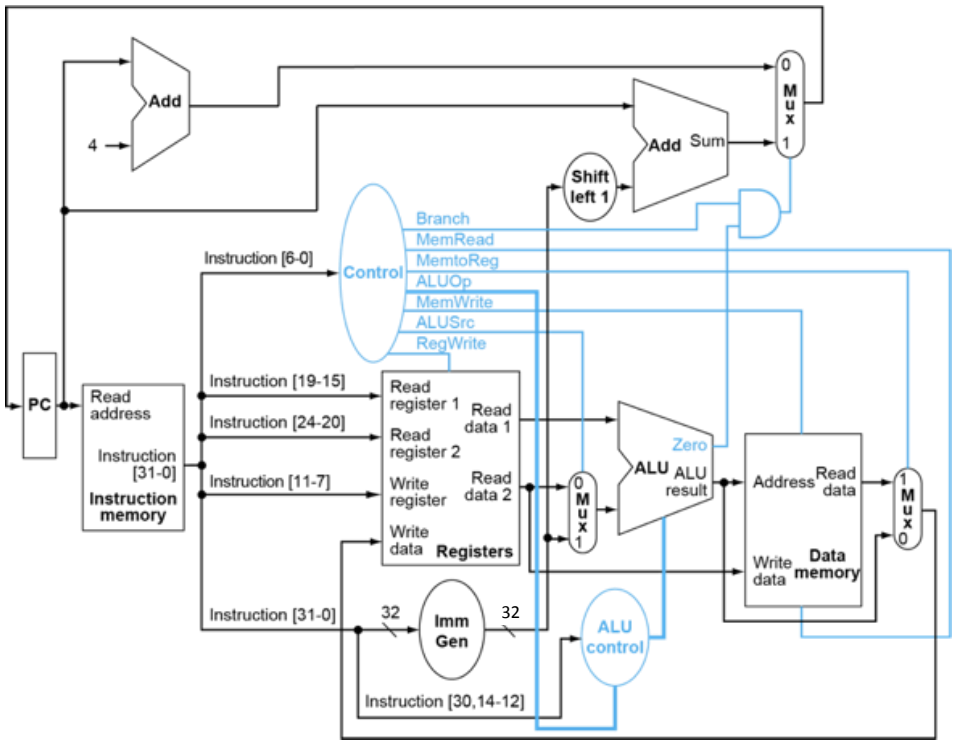

Figure 1 RISC-V Single Cycle Datapath

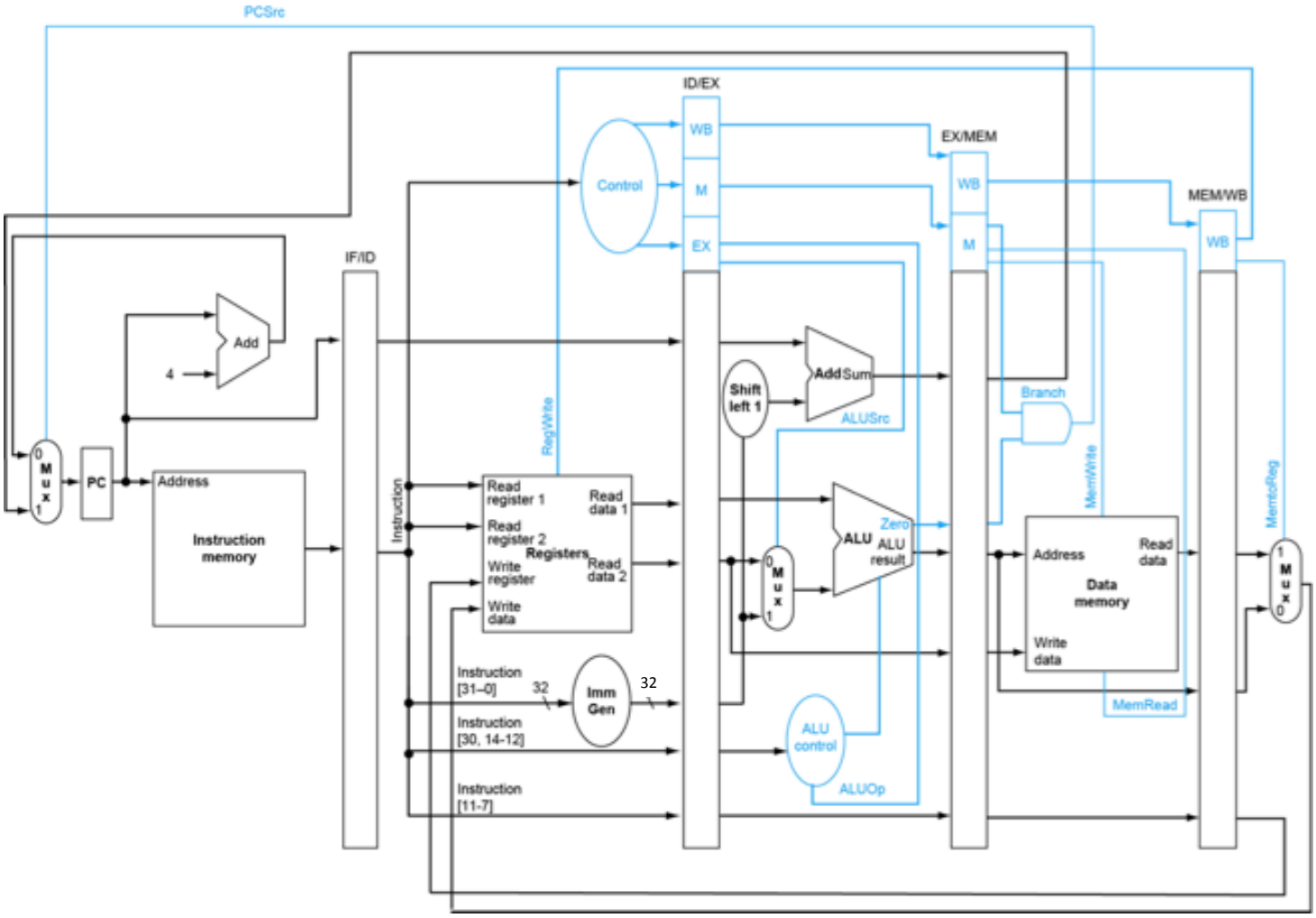

Figure 2 RISC-V Pipelined Datapath (ignoring data and control hazards)

In this lab we are going to conduct the following experiments:

1. Implement the pipelined datapath.
2. Test the pipelined datapath on the Nexys A7 trainer board.

Experiments

Experiment 1: Implement the pipelined datapath

Given the single cycle datapath you implemented, you can easily extend the design of Figure 1 by adding the IF/ID, ID/EX, EX/MEM, and MEM/WB registers whose sizes can be inferred from the circuit diagram in Figure 2.

The following incomplete Verilog code illustrates one possible (structural) implementation of the top-level RISC-V pipelined datapath that corresponds to Figure 2 (and thus ignores data and control hazards). You need to complete its missing parts.

```
module RISCV_pipeline ( .... );

	// wires declarations

	// the module "Register" is an n-bit register module with n as a parameter
	// and with I/O's (clk, rst, load, data_in, data_out) in sequence
	wire [...] IF_ID_PC, IF_ID_Inst;
	Register #(...) IF_ID (clk,rst,1'b1,
				{....},
				{IF_ID_PC,IF_ID_Inst} );

	wire [...] ID_EX_PC, ID_EX_RegR1, ID_EX_RegR2, ID_EX_Imm;
	wire [...] ID_EX_Ctrl;
	wire [...] ID_EX_Func;
	wire [...] ID_EX_Rs1, ID_EX_Rs2, ID_EX_Rd;
	Register #(...) ID_EX (clk,rst,1'b1,
				{....},
				{ID_EX_Ctrl,ID_EX_PC,ID_EX_RegR1,ID_EX_RegR2,
				 ID_EX_Imm, ID_EX_Func,ID_EX_Rs1,ID_EX_Rs2,ID_EX_Rd} );
	// Rs1 and Rs2 are needed later for the forwarding unit

	wire [...] EX_MEM_BranchAddOut, EX_MEM_ALU_out, EX_MEM_RegR2;
	wire [...] EX_MEM_Ctrl;
	wire [...] EX_MEM_Rd;
	wire EX_MEM_Zero;
	Register #(...) EX_MEM (clk,rst,1'b1,
				{....},
				{EX_MEM_Ctrl, EX_MEM_BranchAddOut, EX_MEM_Zero,
				 EX_MEM_ALU_out, EX_MEM_RegR2, EX_MEM_Rd} );

	wire [...] MEM_WB_Mem_out, MEM_WB_ALU_out;
	wire [...] MEM_WB_Ctrl;
	wire [...] MEM_WB_Rd;
	Register #(...) MEM_WB (clk,rst,1'b1,
				{....},
				{MEM_WB_Ctrl,MEM_WB_Mem_out, MEM_WB_ALU_out,
				 MEM_WB_Rd} );

	// all modules instantiations

	// LED and SSD outputs case statements

endmodule
```

Three important notes:

1. Not all signals monitored come from the same stage which means that during a program's execution, each group of signals will be associated with a different instruction in the pipeline.
2. The description above includes 2 extra fields in the ID/EX pipeline register which are the Rs1 and Rs2. These are needed by the forwarding unit which will be implemented in the next experiment.
3. As studied, we will assume that the write back stage is complete in the first half of the clock cycle, which means that writing in the register file should be activated by the falling edge of the RISC-V clock instead of its rising edge.

Experiment 2: Test the pipelined datapath on Nexys trainer Board

Here is a sample set of data items to initialize the data memory:

```
initial begin
        mem[0]=32'd17;
        mem[1]=32'd9;
        mem[2]=32'd25;
end
```

And below is a sample program that uses the above data that can be used to initialize the instruction memory.

To avoid any hazards, we added 3 no operation instructions between every two instructions in the following program.

No operation instructions are written in blue "add x0,x0,x0". Why is it considered a no operation instruction?

Note: the canonical NOP instruction in the RISC-V specification is "`addi x0, x0, 0`". We just use `add` not `addi` here since our current implementation doesn't support `addi` instruction.

```
initial begin
      mem[0]=32'b0000000_00000_00000_000_00000_0110011 ; //add x0, x0, x0
      //added to be skipped since PC starts with 4 after reset
      mem[1]=32'b000000000000_00000_010_00001_0000011 ; //lw x1, 0(x0)
      mem[2]=32'b0000000_00000_00000_000_00000_0110011 ; //add x0, x0, x0
      mem[3]=32'b0000000_00000_00000_000_00000_0110011 ; //add x0, x0, x0
      mem[4]=32'b0000000_00000_00000_000_00000_0110011 ; //add x0, x0, x0
      mem[5]=32'b000000000100_00000_010_00010_0000011 ; //lw x2, 4(x0)
      mem[6]=32'b0000000_00000_00000_000_00000_0110011 ; //add x0, x0, x0
      mem[7]=32'b0000000_00000_00000_000_00000_0110011 ; //add x0, x0, x0
      mem[8]=32'b0000000_00000_00000_000_00000_0110011 ; //add x0, x0, x0
      mem[9]=32'b000000001000_00000_010_00011_0000011 ; //lw x3, 8(x0)
      mem[10]=32'b0000000_00000_00000_000_00000_0110011 ; //add x0, x0, x0
      mem[11]=32'b0000000_00000_00000_000_00000_0110011 ; //add x0, x0, x0
      mem[12]=32'b0000000_00000_00000_000_00000_0110011 ; //add x0, x0, x0
      mem[13]=32'b0000000_00010_00001_110_00100_0110011 ; //or x4, x1, x2
      mem[14]=32'b0000000_00000_00000_000_00000_0110011 ; //add x0, x0, x0
      mem[15]=32'b0000000_00000_00000_000_00000_0110011 ; //add x0, x0, x0
      mem[16]=32'b0000000_00000_00000_000_00000_0110011 ; //add x0, x0, x0
      mem[17]=32'b0_000001_00011_00100_000_0000_0_1100011; //beq x4, x3, 16
      mem[18]=32'b0000000_00000_00000_000_00000_0110011 ; //add x0, x0, x0
      mem[19]=32'b0000000_00000_00000_000_00000_0110011 ; //add x0, x0, x0
      mem[20]=32'b0000000_00000_00000_000_00000_0110011 ; //add x0, x0, x0
      mem[21]=32'b0000000_00010_00001_000_00011_0110011 ; //add x3, x1, x2
      mem[22]=32'b0000000_00000_00000_000_00000_0110011 ; //add x0, x0, x0
      mem[23]=32'b0000000_00000_00000_000_00000_0110011 ; //add x0, x0, x0
      mem[24]=32'b0000000_00000_00000_000_00000_0110011 ; //add x0, x0, x0
      mem[25]=32'b0000000_00010_00011_000_00101_0110011 ; //add x5, x3, x2
      mem[26]=32'b0000000_00000_00000_000_00000_0110011 ; //add x0, x0, x0
      mem[27]=32'b0000000_00000_00000_000_00000_0110011 ; //add x0, x0, x0
      mem[28]=32'b0000000_00000_00000_000_00000_0110011 ; //add x0, x0, x0
```

```
        mem[29]=32'b0000000_00101_00000_010_01100_0100011; //sw x5, 12(x0)
        mem[30]=32'b0000000_00000_00000_000_00000_0110011 ; //add x0, x0, x0
        mem[31]=32'b0000000_00000_00000_000_00000_0110011 ; //add x0, x0, x0
        mem[32]=32'b0000000_00000_00000_000_00000_0110011 ; //add x0, x0, x0
        mem[33]=32'b000000001100_00000_010_00110_0000011 ; //lw x6, 12(x0)
        mem[34]=32'b0000000_00000_00000_000_00000_0110011 ; //add x0, x0, x0
        mem[35]=32'b0000000_00000_00000_000_00000_0110011 ; //add x0, x0, x0
        mem[36]=32'b0000000_00000_00000_000_00000_0110011 ; //add x0, x0, x0
        mem[37]=32'b0000000_00001_00110_111_00111_0110011 ; //and x7, x6, x1
        mem[38]=32'b0000000_00000_00000_000_00000_0110011 ; //add x0, x0, x0
        mem[39]=32'b0000000_00000_00000_000_00000_0110011 ; //add x0, x0, x0
        mem[40]=32'b0000000_00000_00000_000_00000_0110011 ; //add x0, x0, x0
        mem[41]=32'b0100000_00010_00001_000_01000_0110011 ; //sub x8, x1, x2
        mem[42]=32'b0000000_00000_00000_000_00000_0110011 ; //add x0, x0, x0
        mem[43]=32'b0000000_00000_00000_000_00000_0110011 ; //add x0, x0, x0
        mem[44]=32'b0000000_00000_00000_000_00000_0110011 ; //add x0, x0, x0
        mem[45]=32'b0000000_00010_00001_000_00000_0110011 ; //add x0, x1, x2
        mem[46]=32'b0000000_00000_00000_000_00000_0110011 ; //add x0, x0, x0
        mem[47]=32'b0000000_00000_00000_000_00000_0110011 ; //add x0, x0, x0
        mem[48]=32'b0000000_00000_00000_000_00000_0110011 ; //add x0, x0, x0
        mem[49]=32'b0000000_00001_00000_000_01001_0110011 ; //add x9, x0, x1
    end
```

Simulating the top-level module first is strongly recommended. You can test your design by writing a testbench and checking on significant intermediate signals like writedata, writeregister and RegWrite signals entering the register file and the current PC.

In order to be able to monitor the various values on each stage of the datapath, we use the same design we used for lab5&6. Switches are used to select the display of important signals on the Nexys A7 LEDs and seven segment display based on selection inputs ledSel and ssdSel,

On the seven segment Display, add one more option to view the 5-bit write register signal entering the register file when ssdSel = 1100.

Deliverables

General Report Submission Notes:

- Each student is required to submit an individual report.
- The submission should be composed of **two files only**:
  1. YourID_Report.pdf/doc:
     A single pdf/word file having the solution of the lab report questions in correct sequence.
  2. YourID_labwork.zip/rar/…:
     A zipped folder containing your codes for all the lab experiments organized in sequence.
- **Submission deadline**: half an hour before the start of next week's session.
- **Submission method**: On Blackboard

Lab Report [10 pts]

1. [0 pts] Your name and student ID.
2. [2 pts] All Verilog code of experiment 1.
3. [2 pts] Show the final output of the sample program of experiment 2 using screenshots of the simulation.
4. [2 pts] Provide a table for the sample program of experiment 2 that displays for each instruction the expected values of PC output, PC input, writedata of the register file, and writedata of the data memory. You can ignore the NOP instructions.
5. [2 pts] Modify the sample program in experiment 2 to minimize the number of NOP instructions as long as it will not mess up with the correct operation of the program. Identify for each NOP the type of hazard that caused this NOP to be added.
6. [2 pts] You previously created your own RISC-V program in **Lab 5&6 Experiment 5**. Add to that program the minimum number of NOP instructions that will not mess up with the correct operation of the program using our current pipelined implementation. Identify for each NOP the type of hazard that caused this NOP to be added.
