EE 361L Fall 2015
Pipelined MIPS-L (PMIPS-L)

1 Introduction

PMIPS-L is pipelined version of MIPSL, as described in Homework 9. It has five pipeline stages just as the pipelined MIPS in Chapter 4 in the textbook as shown in Figure 1. This project is to design this computer using verilog and to implement it in an FPGA. The project has two parts:

- **Part 1**: You will implement a simpler version of PMIPS-L we refer to as PMIPS-L0. This computer has the same datapath as PMIPS-L. However, only one instruction is executed at any time. Thus, the computer doesn't pipeline instructions. An instruction is issued into the pipeline if the datapath is empty. Then subsequent instructions must wait until this instruction is completely processed.
- **Part 2**: You will implement PMIPS-L. This computer will pipeline its instructions. It uses static branch prediction, and in particular the prediction of branch-untaken.

Each part is organized into a sequence of subprojects.

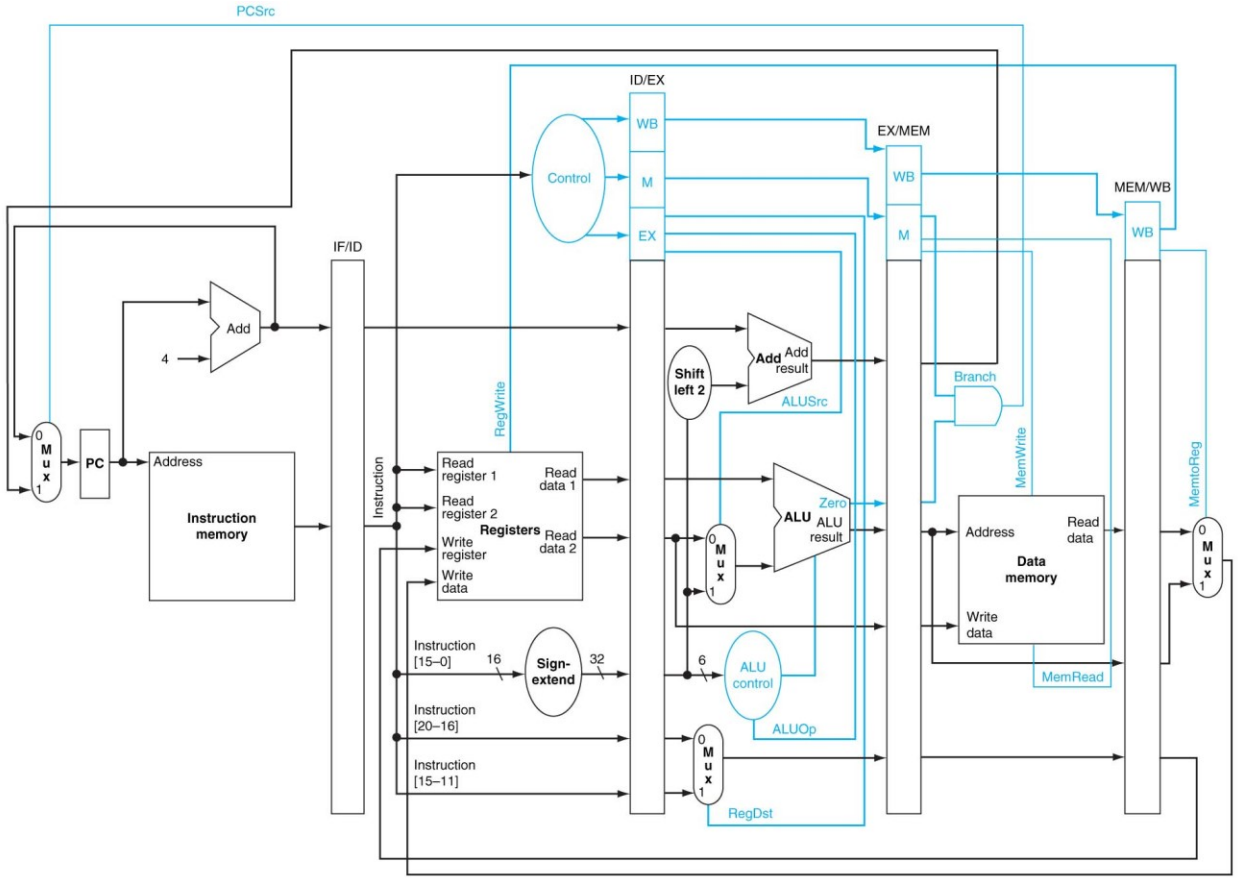

Figure 1. Five-stage Pipeline MIPS (from textbook).

All the sequential components are synchronized with the positive clock edge except the register file, which is synchronized with the negative clock edge.

2 Assignment

Part 1: Implement PMIPS-LO:
This part has three subprojects.

Subproject 1: Implement PMIPS-LO in verilog so that it can run the program in IM1.V from Homework 9. Thus, it must be able to execute "add", "addi", "beq", and "j". For this project, use the files parts verilog file from Homework 9. Also use the attached files (which are in a folder Subproject 1):

- PMIPSL0.V: This is a verilog file for the PMIPS-L0. It's very incomplete and may be buggy.
- testbenchPMIPSL0.V: This is the testbench for the processor.

You have to design your own controller, and PC logic.

Subproject 2: Modify the PMIPS-L0 in verilog so that it can also execute lw and sw.

There is a folder for Subproject 2, which has a testbench: testbenchSubproject2. It can run the program IM2.V from Homework 9. Recall that this program will constantly check switch 0 of the I/O to check if it's 0 or 1. If it's 0 then the 7-segment display is set to display "0". If the switch is 1 then the 7-segment display is set to display "1".

Subproject 3: Configure the PMIPS-L0 from Subproject 2 into the FPGA in the Digilent Basys board.

The basic difference between this subproject from the previous subproject is that there is no verilog testbench. Instead you must create a verilog module FPGADevice.V that has as its components the instruction memory, data memory and I/O, and CPU. There is a folder for Subproject 3 which shows what this verilog module should look like. It may be buggy. There is also a testbench for it.

Once you have the module synthesized, connect its input port to a sliding switch of the Basys board, and its output port to a 7 segment display. Then you can download the bit file into the FPGA.

Part 2: Implement PMIPS-L

This part is under construction. It is to implement the pipelined version of the PMIPS-L. There will be branch untaken prediction, and stalls to avoid data hazards. Also, the jump instruction is done in the IF stage of the pipeline.

3. Grading

Total points for this lab is 40

- Writing Style: 20 points for the written report. No revisions.
- MIPS-L Implementation: 20 points depends on how far the student gets
    - Subproject 1 = 14 points
    - Subprojects 1 and 2 = 16 points
    - Subprojects 1, 2, and 3 = 18 points
    - Subprojects 1,..., 4 = 20 points
    - Subprojects 1, ..., 5 = 20 points + 4 bonus points

2. PMIPS-L0

For this processor, only one instruction is being processed at a time, each instruction taking *exactly* four clock cycles. The following are the states:

State	Controller output to the ID/EX register	PC updates	Description of stage
0	Bubble	PC = PC+2	Instruction fetch stage
1	Control signals that are dependent on the opcode in IF/ID	PC = PC (hold)	Instruction decode stage
2	Bubble	PC = PC (hold)	
3	Bubble	PC = Target branch address if the branch condition is true. Otherwise, PC = PC (hold)	

Figures 3, 4, and 5 shows different examples of instructions traversing the datapath.

- Figure 3 has two consecutive instructions: addi followed by sub. addi is located at address 100, while sub is located at address 102.
- Figure 4 has two consecutive instructions: beq followed by addi, where the branch condition is false, so the branch is not taken. beq is located at address 100, while addi is located at address 102.
- Figure 5 has two consecutive instructions: beq followed by addi, but where the branch condition is false, so the branch is taken. The branch is to address 200, which is the location of instruction sub.

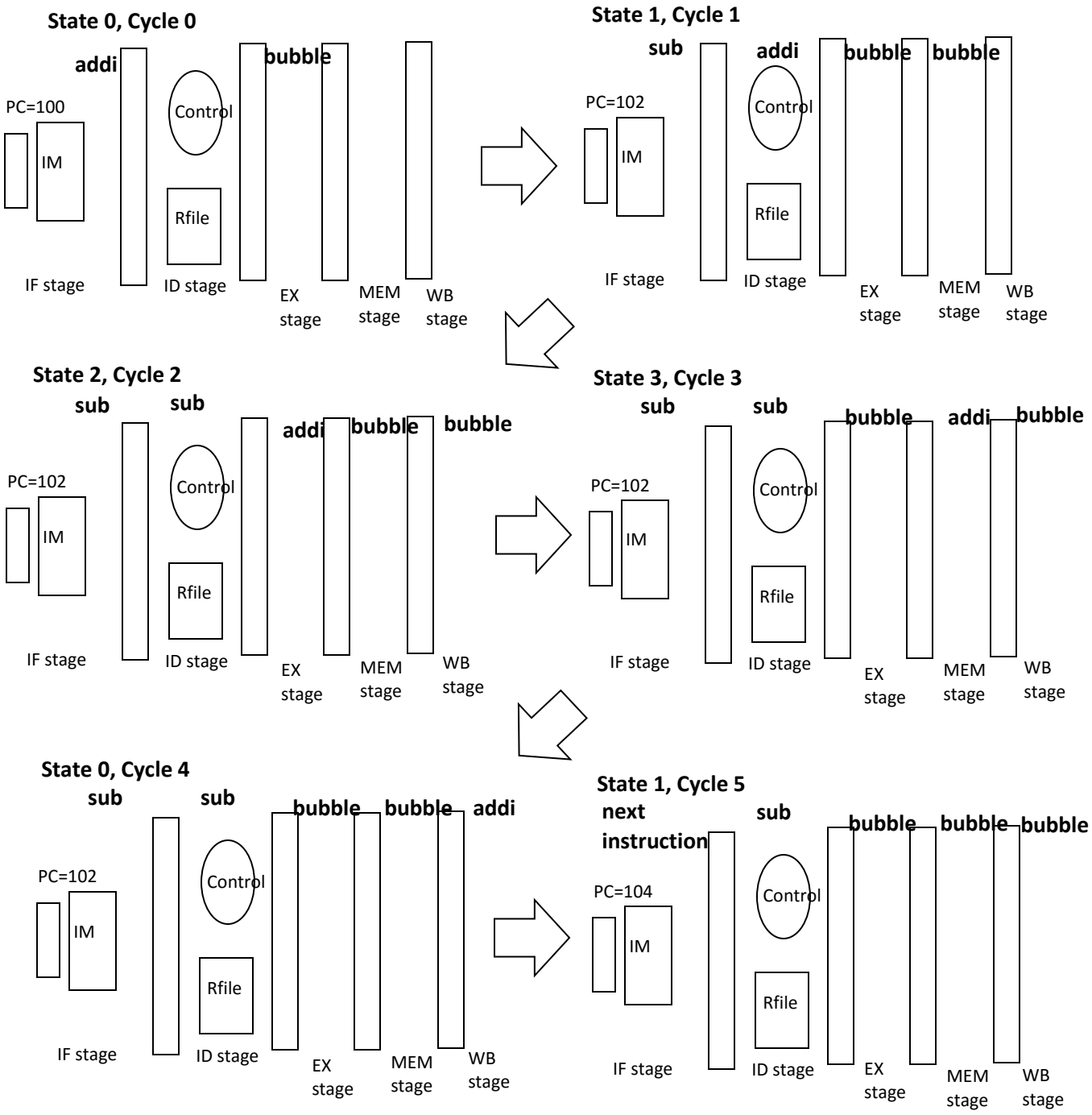

Figure 3. The progression of addi followed by sub in the datapath.

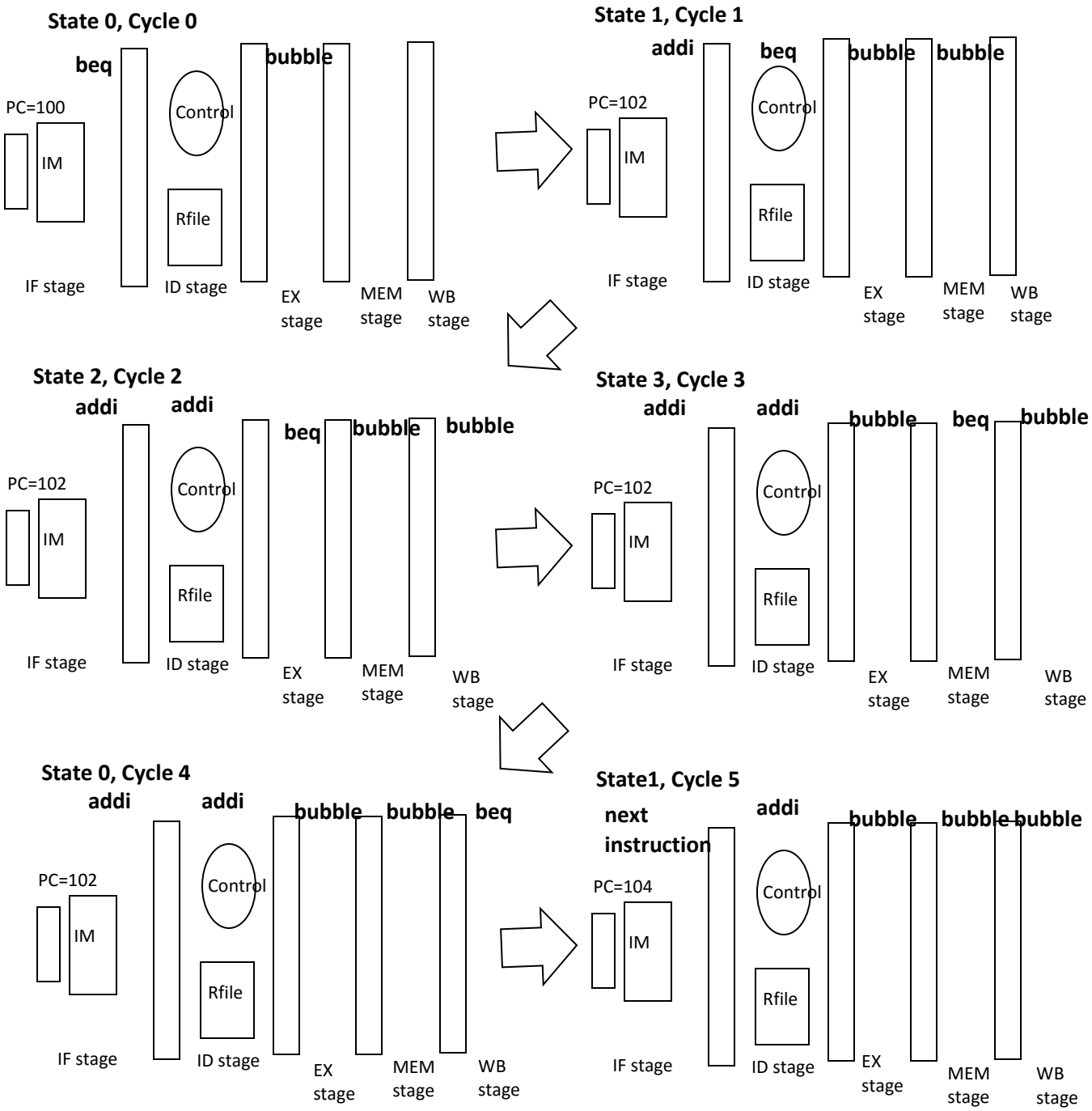

Figure 4. The progression of sub followed by addi in the datapath when the branch is not taken.

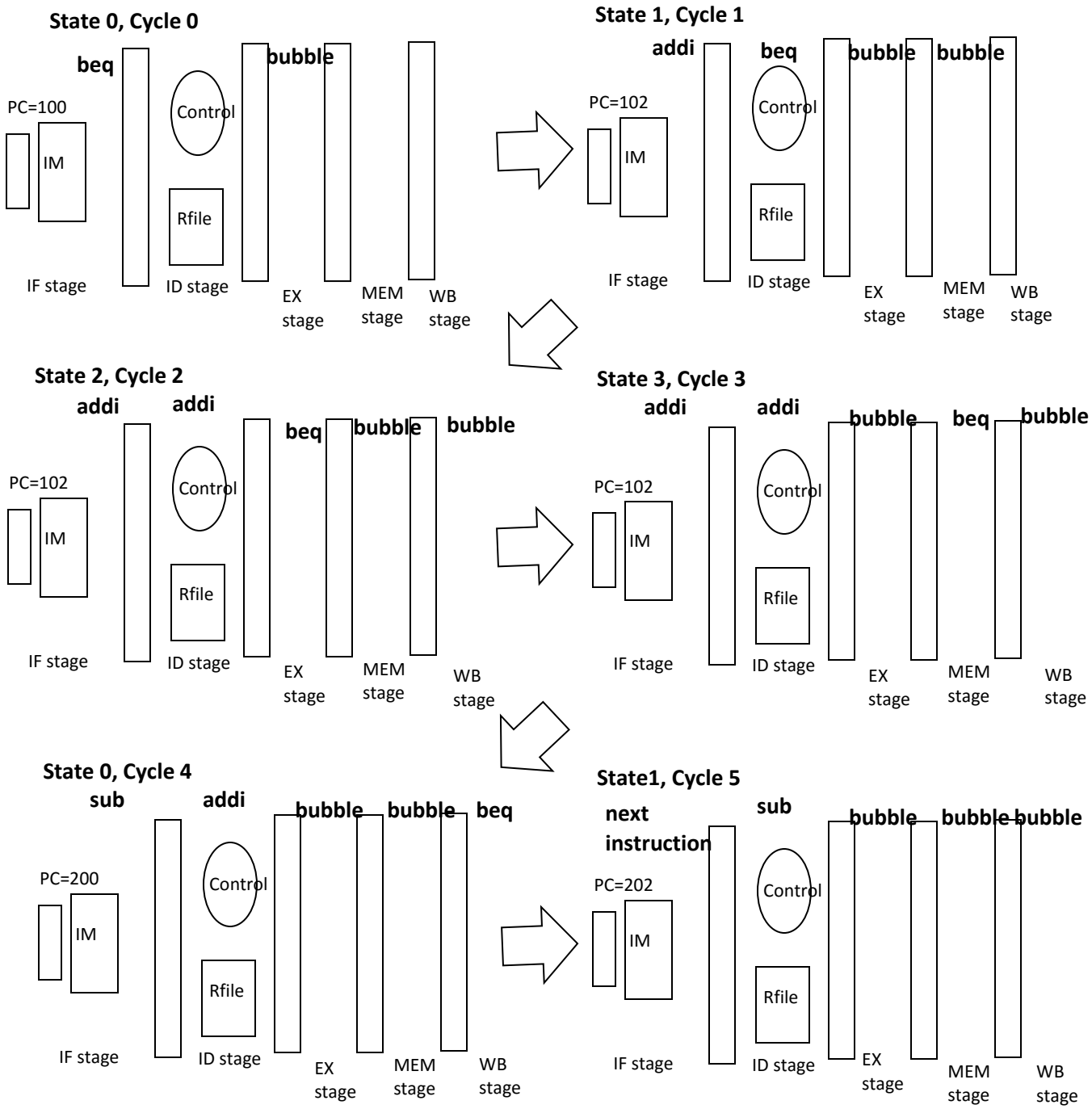

Figure 5. The progression of beq followed by addi in the datapath when the branch is taken to location 200, which has the instruction sub.

PC Logic: Note that the PC logic must be able to load the target branch address into the PC while the PC is in the stalled state.

Controller: This is a sequential circuit with four states. It repeatedly goes through states 0, 1, 2, 3, 0, ... The controller inputs the opcode of the instruction from the IF/ID register.

When the controller inserts a bubble, it must disable all the control lines, and in particular the ones that control the write-enables for registers and memory, as well as loads to the PC. Thus, to insert a bubble means that the register-

write and memory-write must be disabled. To prevent the PC from being loaded inadvertently, the PC should be set to a stalled state. Then the only way the PC can be inadvertently set to the wrong value is due to an inadvertent branch. To prevent this, set Branch= 0.

Pipeline Registers: They are basically an array of D flip flops.

As an example, the ID/EX register has the following fields:

- WB: {RegWrite,MemtoReg}
- M: {MemRead,MemWrite,Branch}
- EX: {ALUSrc,ALUOp,RegDst}
- 16-bit output of the sign extension
- Outputs of the read data from the register file
- Two register fields (3-bits each) from the instruction

There are two memories: instruction memory that has the program, and data memory which has RAM. The RAM portion of the data memory has address 0 up to 254.

For EE 361L lab 5, you will implement PMIPS-L0 in an FPGA. Also, your computer will need Input/Output (IO). The IO will be connected to other circuits on the Digilent Basys boards. In particular, the IO of your computer will be attached to a seven segment delay and sliding switches. Your computer will need two IO ports, one for the display and the other for the switches. The port for the seven segment display is 0xfffa, and the port for the sliding switches is 0xfff0. These addresses are in the address space of data memory. So you will access the IO using the lw and sw instructions. The seven segment display can be written to, and the sliding switches can be read from. The sliding switches are labeled SW1 and SW0 and are the last two bits of the data read from 0xfff0.

3. PMIPS-L

For Stage 4, is to implement a little bit of pipelining:

- Controller avoids data hazards
- Branching is done by stall on branch. This means that when a branch instruction is put into the pipeline, there is a stall until the branch is resolved.

Your processor will run the multiply program of IM1.V. So it only needs to execute four instructions: addi, add, beq, and j. Note that the jump instruction j is not involved in any data hazards. Thus, only three instructions (addi, add, beq) may be involved with data hazards.

The jump instruction is resolved in the Instruction Fetch (IF) stage. This means that in the IF stage, the output of the instruction memory is attached to a circuit that detects if the instruction is a "jump" or not (easily checked by comparing the opcode field with the value 8). If it is then the PC equals the jump address in the next clock cycle.

The instructions add, addi, and beq may have to be stalled to avoid data hazards. In particular, it has to check if the regwrite signal is enabled downstream in the pipeline. It also checks if the destination register for the regwrite signal is a source register for the instruction. If this is the case then it stalls the instruction.

Appendix C shows how the instructions of IM1.V are processed in this pipelined processor.

The controller will be a combinational circuit. This is different from earlier versions (Stages 1, 2, and 3) where the controller was a state machine with four states. Instead the controller for this Stage 4 will input values from pipeline stage registers as well as the opcode of the instruction in the instruction decode stage. Then it decides to insert the current instruction into the pipe or insert a bubble.

Note that reset must also reset the pipeline registers so that they basically function as nop operations. This means that all the control signals do not write into the register file, data memory, or registers (e.g., PC register).

Appendix C

```
0:      L0      addi    $2,$0,3         // addi1
2:              add     $3,$0,$0
4:      L1      beq     $2,$0,L0        // beq
6:              addi    $3,$3,5         // addi2
8:              addi    $2,$2,-1        // addi3
10:             j       L1              // j
```

PC	I-Memory Output	IF/ID (src reg1, src reg2)	ID/EX [DataHaz, DstReg]	EX/MEM [DataHaz, DstReg]	MEM/ WB	Comments: Actions by controller and datapath
0	addi1					
2	add	addi1 (0,)				
4	beq	add (0,0)	addi1 [1,2]			
6	addi2	beq(2,0)	add [1,4]	addi1 [1,2]		Stall: hazard via $2
6	addi2	beq (2,0)	Bubble	add [1,4]	addi1	
8	addi2	Bubble	beq [,]	Bubble	add	
10	addi2	Bubble	Bubble	beq [,]	Bubble	Not branching
12	addi3	addi2	Bubble	Bubble	beq	
14	j	addi3	addi2	Bubble	Bubble	
4	beq	j	addi3	addi2	Bubble	
6	addi2	beq	j	addi3	addi2	
6	addi2	beq	Bubble	j	addi3	
8	add3	add2	beq	Bubble	j	