EE 361L Fall 2017
Pipelined LEGLite

October 24, 2017

1 Introduction

LEGLite-Pipe is pipelined version of LEGLite-Single, as described in Homework 6. It has five pipeline stages just as the pipelined LEGv8 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. You can use parts from LEGLite-Single (homework 6) including Parts.V, IM1.V, and IM2.V.

The project has two parts:

- **Part 1**: You will implement a simpler version of LEGLite-Pipe we refer to as LEGLite-Pipe0. This computer has the same datapath as LEGLite-Pipe 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 (they are stalled) until this instruction is completely processed.
- **Part 2**: You will implement LEGLite-Pipe. This computer will pipeline its instructions. It uses static branch prediction, and in particular "branch-untaken".

Each part is organized into a sequence of subprojects.

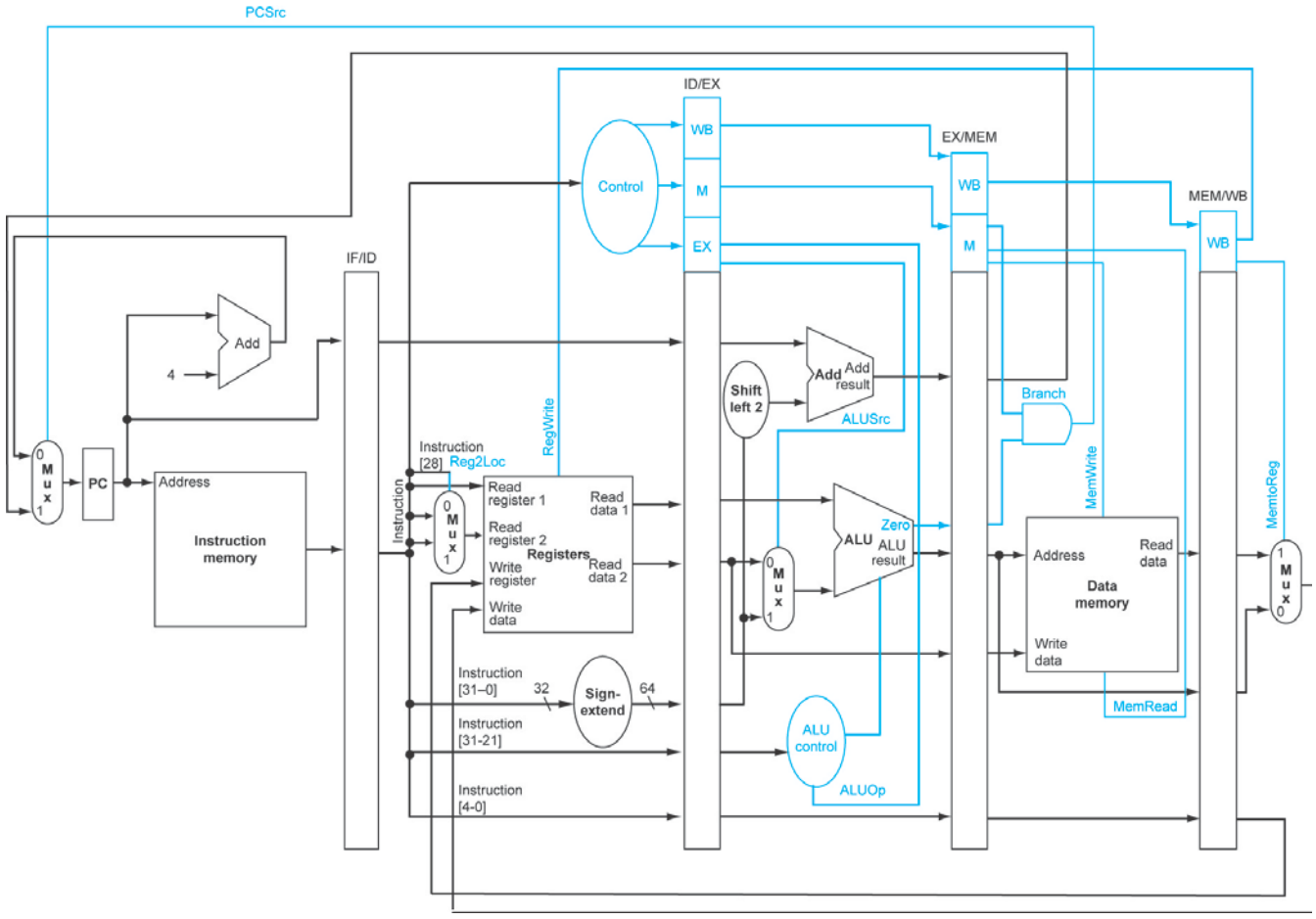


Figure 1. Five-stage Pipeline LEGv8 (from textbook).

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 LEGLite-Pipe:
This part has three subprojects.

Subproject 1: Implement LEGLite-Pipe0 in verilog so that it can run the program in IM1.V from Homework 6. Thus, it must be able to execute "ADD", "ADDI", and "CBZ". For this project, use the files parts verilog file from Homework 6. Also use the attached files in a folder Subproject 1:

- LEGLitePipe0.V: This is a verilog file for the LEGLite-Pipe0. It's very incomplete and may be buggy.
- testbenchLEGLitePipe0.V: This is the testbench for the processor.

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

A description of LEGLite-Pipe0 controller is given in Section 4.

Before writing any verilog code,

- Draw a block diagram of your computer. It should be similar to Figure 1 but for LEGLite-Pipe0.
- Understand completely how the computer should operate. The controller is usually the most difficult component to understand.

Helpful hint is to note that the Data Memory isn't used in IM1.V. So you could design a proto-LEGLite-Pipe0 without the data memory. After correctly building this, you can modify it to LEGLite-Pipe0.

Subproject 2: Modify the LEGLite-Pipe0 in verilog so that it can also execute LD and ST. It should be able to run the program in IM2.V from Homework 6.

There is a folder for Subproject 2, which has a testbench: testbenchSubproject2. Recall that program IM2.V 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".

You can also find in this folder a file "Lab5-Stage2-ExampleTrace" which is an example trace for this version of the computer.

Subproject 3: Configure the LEGLite-Pipe0 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 LEGLite-Pipe (Under construction)

This part is under construction. It is to implement the pipelined version of the LEGLite. There will be branch untaken prediction, and stalls to avoid data and control hazards.

Stage 4

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

- Controller avoids data and control hazards by stalling
- 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, and CBZ. Note that the jump instruction j is not involved in any data hazards. Thus, only three instructions (ADDI, ADD, CBZ) may be involved with data hazards.

The instructions ADD, ADDI, and CBZ 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.

Stage 5

Upgrade the processor so that it can execute IM2.V. This means it has to execute LD and ST instructions.

3. Grading

Total points for this lab is 40

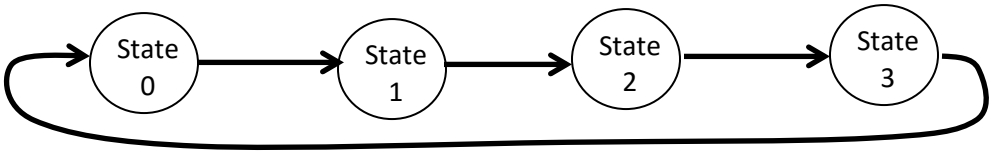
- Writing Style: 20 points for the written report. No revisions.
- LEGLite-Pipe Implementation: 20 points depends on how far the student gets
    - Subproject 1 = 14 points
    - Subprojects 1 and 2 = 16 points
    - Subprojects 1, 2, and 3 = 18 points
    - Subprojects 1,..., 4 = 20 points
    - Subprojects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

4. LEGLite-Pipe0 Controller

For the LEGLite-Pipe0 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)	

Below is a state diagram of the controller



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

- Figure 3 has PC is initially 100 and the instructions in memory are
  100 ADDI
  102 ANDI
  104 ADD

- Figures 4 and 5 are the case when PC is initially 100 and the instructions in memory are
  100 CBZ
  102 ANDI
  104 ADD
  ...
  200 ADDI
  202 LD

  Figure 4 is the subcase where the branch condition is false, so the branch is not taken. CBZ is located at address 100, while ADDI is located at address 102. Figure 5 is the subcase where the branch condition is true, so the branch is taken.

The figures show how the pipeline evolves over time. They all start with clock cycle 0, and progress through subsequent clock cycles. The figure also shows the "state" of the computer, which corresponds to one of the four states of the controller. Note that output of the controller is a "bubble" for states 0, 2, and 3. However, in state 1, the output depends on the instruction.

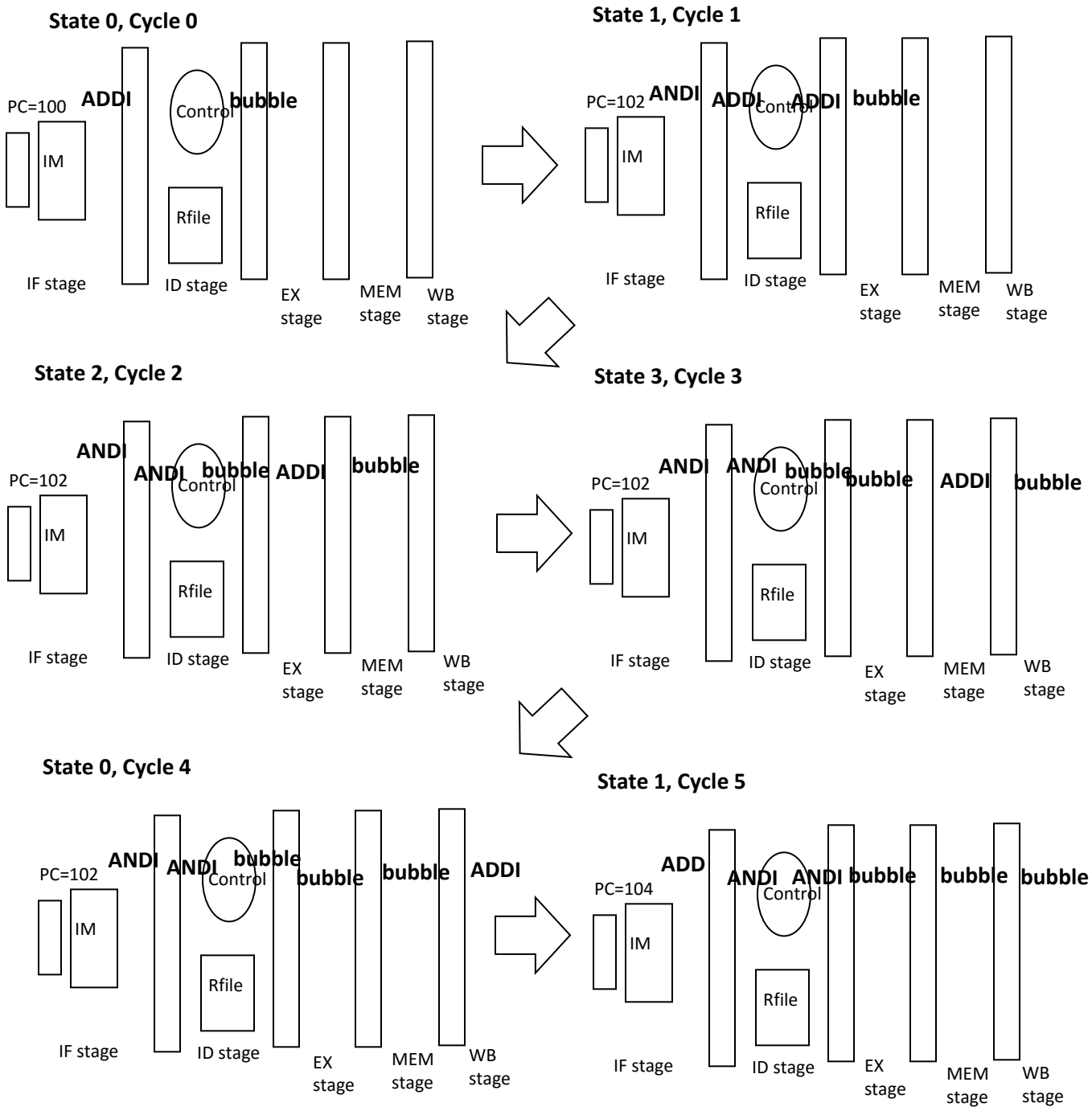


Figure 3. The progression of ADDI followed ANDI and ADD in the datapath.

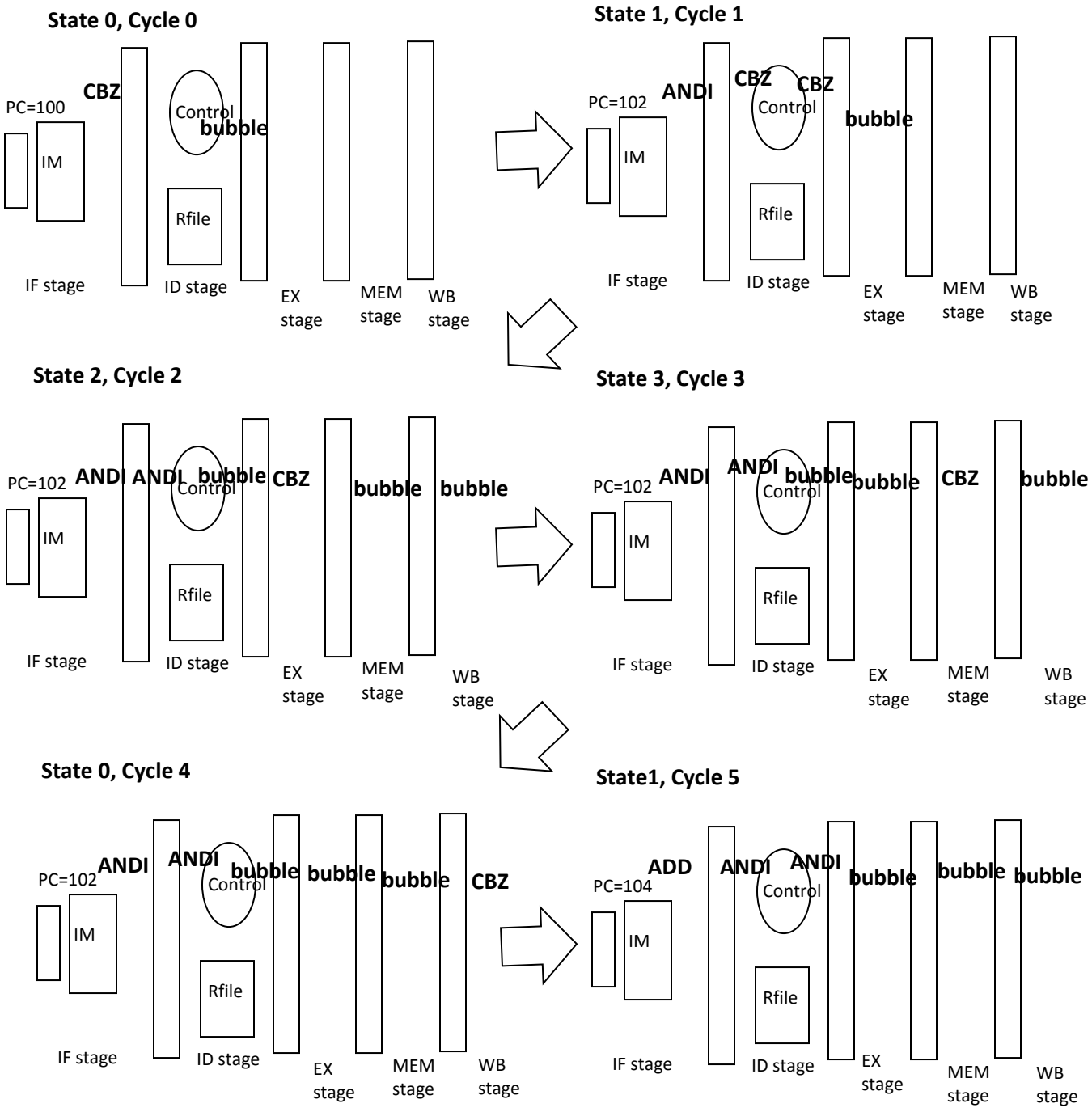


Figure 4. The progression of CBZ followed by ADDI in the datapath when the branch is not taken.

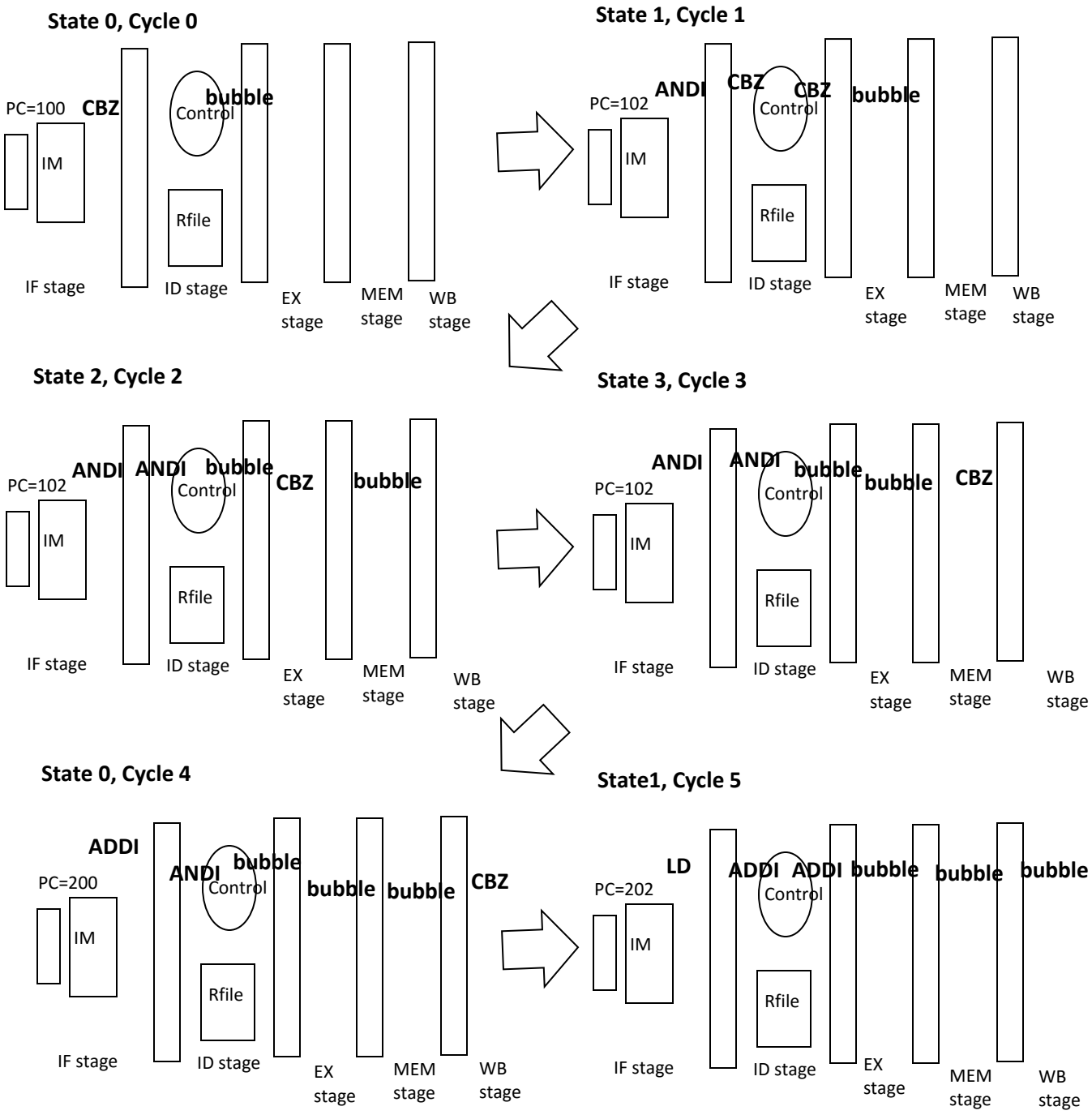


Figure 5. The progression of beq followed by addi in the datapath when the branch is taken to location 200, which has the instruction sub.

PC: Note that the PC 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.