CS311 Project 3: Simulating Pipelined Execution

Due 11:59pm, December 3rd

1. Overview

This third project is to implement a five-stage pipeline for the MIPS ISA, which supports the same subset of instructions as the second project. The pipeline model will follow the MIPS pipeline in the lecture slides and textbook.

2. Pipeline Implementation

2.1 Pipeline Stages

We will use a simple 5-stage pipeline for this project

- IF : fetch a new instruction from memory.
- ID: decode the fetched instruction and read the register file
- EX: execute an ALU operation
  - Execute arithmetic and logical operations
  - Calculate the addresses for loads and stores
- MEM: access memory for load and store instructions
- WB: write back the result to the register

Between the adjacent pipeline stages, pipeline registers (or pipeline latches) must be modeled. At ID, the control signals must be generated to control the operations at each stage.

2.2 Register Timing at ID and WB

Similarly to the MIPS pipeline, WB writes the register file at the first half of a cycle and ID reads the register file at the second half of a cycle. Therefore, there will be no structural hazard between ID and WB.

2.3 Memory Model

We assume an ideal dual-ported memory, which allows both a read at IF, and a read/write at MEM simultaneously at a cycle. There is no structural hazard for the memory accesses from IF and MEM. You can assume the memory locations where instructions are stored are never updated. Therefore, you do not need to consider the case where a store at MEM updates the same address fetched at IF at the same cycle. Your test programs must not violate this assumption.

2.4 Forwarding

The pipelined architecture must support data forwarding from MEM/WB-to-EX, EX/MEM-to-EX. With the forwarding support, the data hazard occurs for the data dependency from a load to the succeeding instruction which uses the value at EX. MEM/WB-to-MEM is also supported.

2.5 Control Hazard

For unconditional jumps, you will always add a one-cycle stall to the pipeline (J, JAL, JR).

For conditional branches (BEQ, BNE), your simulator must support two modes, with or without branch prediction. i) With branch prediction, it uses a static branch predictor, which always predicts a branch is taken. If a branch miss prediction occurs, the pipeline must be stalled for three cycles. ii) Without branch prediction, every branch must be followed by three bubble cycles.

2.6 Stopping the Pipeline

The simulator must stop after a give number of instructions finishes the WB stage. At cycle 1, the first instruction is fetched from the memory. If the last instruction is in the WB stage at cycle N, the final CYCLE count is N.

2.7 Other Conditions (same as the project 2)

States: the simulator must maintain the system states. The necessary register set (R0-R31, PC) and the memory must be created when the emulation begins.

Loading an input binary: For a given input binary, the loader must identify the text and data section size. The text section must be loaded to the emulated memory from the address 0x400000. The data section must be loaded to the emulated memory from the address 0x10000000.

In this project, the simple loader does not create the stack region.

Initial states: PC:
The initial value of PC is 0x400000
Registers: All values of register0 to 31 are zero.
Memory: You may assume all initial values are zero, except for the loaded text and data sections.

Supported Instruction Set

ADDIU	ADDU	AND	ANDI	BEQ	BNE	J
JAL	JR	LUI	LW	LA*	NOR	OR
ORI	SLTIU	SLTU	SLL	SRL	SW	SUBU

3. Pipeline Register States

You need to add pipeline register states between stages. The followings are possible register contents, but you need to add more states.

IF_ID.Instr : 32-bit instruction
IF_ID.NPC : 32-bit next PC (PC+4)
ID_EX.NPC : 32-bit next PC

ID_EX.REG1 : REG1 value
ID_EX.REG2 : REG2 value
ID_EX.IMM : Immediate value
EX_MEM.ALU_OUT: ALU output
EX_MEM.BR_TARGET: Branch target address
MEM_WB.ALU_OUT: ALU output
MEM_WB.MEM_OUT: memory output

You must carefully design what fields are necessary for each pipeline register, and explain them in "README" file in your submission. You must explain what each field means.

4. Simulator Option and Output

4.1 Options

```
cs311sim [-nobp] [-m addr1:addr2] [-d] [-p] [-n num_instr] inputBinary
```

- -nobp : branch prediction is disabled. All conditional branches must add three-cycle bubbles.
- -m : Dump the memory content at the end of simulation
- -d : print the register file content, and the current PC, at *every cycle*
- The default output is the PC and register file content after the completion of the given number of instructions. If –m option is specified, the memory content from addr1 to addr2 must be printed.
- -p : print the PCs of instructions in each pipeline stage at *every cycle*
  CYCLE N: x3004 | x3003 | x3002 | x3001 | x3000

4.2 Formatting Output

The TAs will provide the skeleton code for displaying output format like lab2.
PC and register content must be printed in addition to the optional memory content.

- Register contents
- CYCLES: the number of cycles to the completion of the input program.
- Print pipeline states every cycle if –p option is enabled. See section 4.1.
- Print register states every cycle if –d option is enabled. See section 4.1

For a given input MIPS binary (the output binary file from the assembler built in the project 1), the simulator must be able to mimic the behaviors of the MIPS ISA execution.
The examples of output format are attached at the next page.

5. Hand in

You should submit the compressed file of the source files through tar or zip program.

In your submission, a "README" file must be included. (See 3. Pipeline Register States)

Please make the compressed file name with your team name. (team_name.tar or team_name.zip)

Ex) If you are team14, you should make the compressed file with “team14.tar” or “team14.zip”
Please send the email to cs311_ta@calab.kaist.ac.kr with attaching the compressed file until the due date.

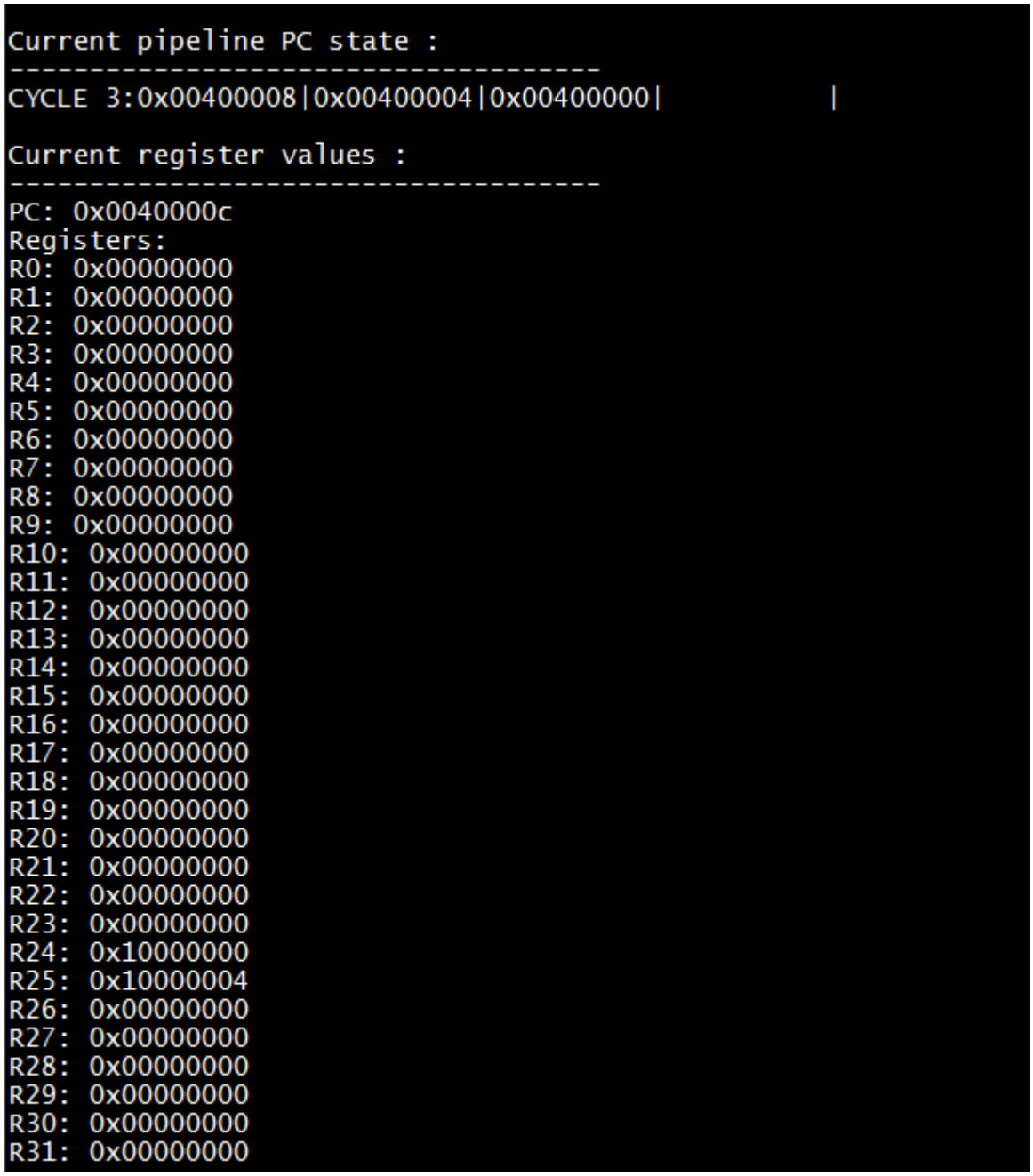

Figure 1. Output format at cycle 3

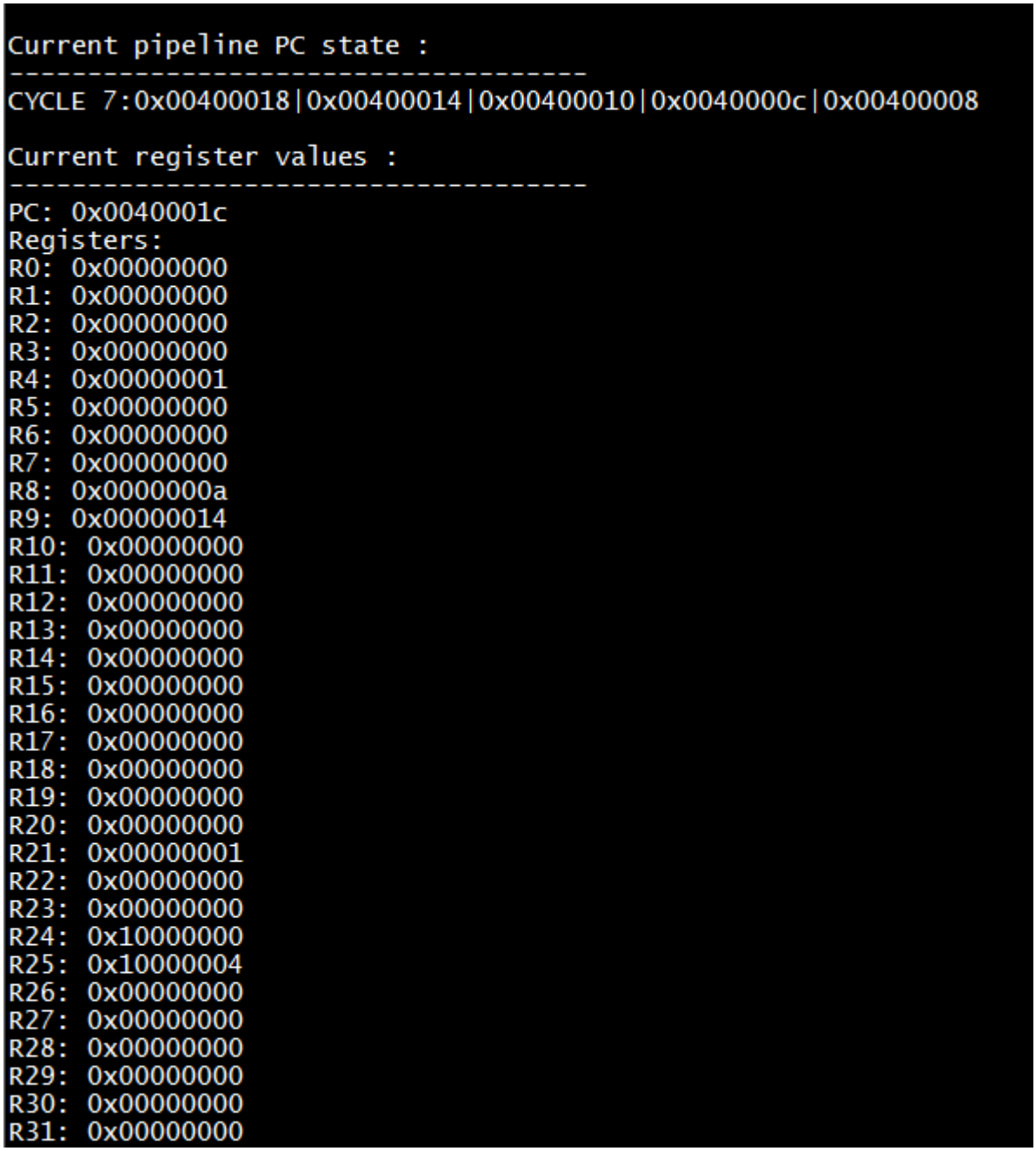

Figure 2. Output format at cycle 7

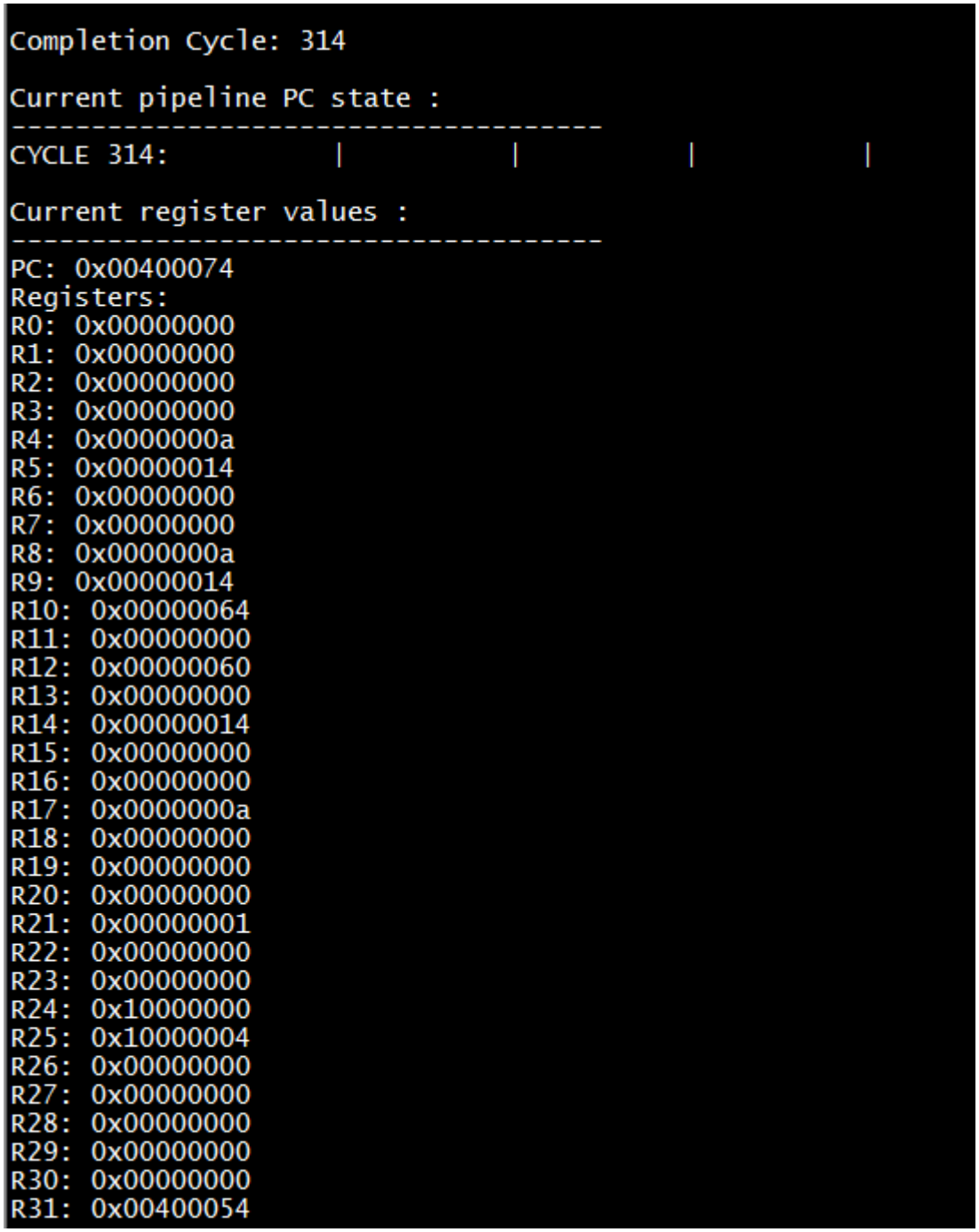

Figure 3. Output format at the program completion