Project 2 Pipelining: Computer Architecture (Spring 2020)

Worth: 100 points
Assigned: 6/1 (Munday), 2020
Due: 23:59pm, 6/19 (Friday), 2020

1. Purpose

This project is intended to help you understand in detail how a pipelined implementation works. You will write a cycle-accurate behavioral simulator for a pipelined implementation of the proto-type processor from the Project 1, complete with data forwarding and simple branch prediction.

2. Pipelined Implementation

2.1. Datapath

For this project we will use the datapath as shown in the additional slides for this project.
One difference between Project 2 and the pipelined processor described in lecture is that we will add a pipeline register after the write-back stage (the WBEND pipeline register). This will be used to simplify data forwarding so that the register file does not have to do any internal forwarding. It will also mean that 3 noops will be required to avoid hazards instead of the 2 needed as discussed in class.
You will use a clocking scheme mimicking real-life processors (e.g., register file and memory writes require the data to be present for the whole cycle).

2.2. jalr

You will not implement the jalr instruction. Taking out jalr eliminates several dependencies.

2.3. Memory

We are keeping a simple representation of a memory unit. Just access memory directly as an array (similar to Project 1).
Note in the typedef of stateType below that there are two memories: instrMem and dataMem. When the program starts, read the machine-code file into BOTH

instrMem and dataMem (i.e., they will initially have the same contents). During execution, read instructions from instrMem and perform load/stores using dataMem. That is, instrMem will never change after the program starts, but dataMem will change. In a real machine, these two memories would be an instruction and data cache, and they would be kept consistent.

2.4. Pipeline Registers

To simplify the project and make the output formats uniform, you must use the following structures WITHOUT MODIFICATION to hold pipeline register contents. Note that the instruction gets passed down the pipeline in its entirety.

```
#define NUMMEMORY 65536 /* maximum number of data words in memory */
#define NUMREGS 8 /* number of machine registers */

#define ADD 0
#define NOR 1
#define LW 2
#define SW 3
#define BEQ 4
#define JALR 5 /* JALR will not implemented for this project */
#define HALT 6
#define NOOP 7

#define NOOPINSTRUCTION 0x1c00000

typedef struct IFIDStruct {
        int instr;
        int pcPlus1;
} IFIDType;

typedef struct IDEXStruct {
        int instr;
        int pcPlus1;
        int readRegA;
        int readRegB;
        int offset;
} IDEXType;

typedef struct EXMEMStruct {
        int instr;
```

```
        int branchTarget;
        int aluResult;
        int readRegB;
} EXMEMType;

typedef struct MEMWBStruct {
        int instr;
        int writeData;
} MEMWBType;

typedef struct WBENDStruct {
        int instr;
        int writeData;
} WBENDType;

typedef struct stateStruct {
        int pc;
        int instrMem[NUMMEMORY];
        int dataMem[NUMMEMORY];
        int reg[NUMREGS];
        int numMemory;
        IFIDType IFID;
        IDEXType IDEX;
        EXMEMType EXMEM;
        MEMWBType MEMWB;
        WBENDType WBEND;
        int cycles; /* number of cycles run so far */
} stateType;
```

3. Problem

3.1. Basic Structure

Your task is to write a cycle-accurate simulator for the LC-2K. We recommend you start with the Project 1 simulator. The main modifications will be in the run() function.

At the start of the program, initialize the pc and all registers to zero. Initialize the instruction field in all pipeline registers to the noop instruction (0x1c00000).

run() will be a loop, where each iteration through the loop executes one cycle. At the beginning of the cycle, print the complete state of the machine (you must use the printState() function at the end of this handout WITHOUT MODIFICATION). In

the body of the loop, you will figure out what the new state of the machine (memory, registers, and pipeline registers) will be at the end of the cycle. Conceptually, all stages of the pipeline compute their new states simultaneously. Since statements execute sequentially in C rather than simultaneously, you will need two state variables: "state" and "newState". For clarity, "state" and "newState" refer to the C variable names, and state without quotes refers to the English word meaning the condition of the datapath at an instant of time. "state" will be the state of the machine while the cycle is executing; "newState" will be the state of the machine at the end of the cycle. Each stage of the pipeline will modify the "newState" variable using the current values in the "state" variable. For example, in the ID stage, you might have the following statement:

newState.IDEX.instr = state.IFID.instr (to transfer the instruction in the IFID register to the IDEX register)

In the body of the loop, you will use "newState" only as the target of an assignment and you will use "state" only as the source of an assignment (e.g., newState... = state...). "state" should never appear on the left-hand side of an assignment (except for array subscripts), and "newState" should never appear on the right-hand side of an assignment.

Your simulator must be pipelined. This means that the work of carrying out an instruction should be done in different stages of the pipeline as described in lecture. The execution of multiple instructions should be overlapped. The ID stage should be the only stage that reads the register file; the other stages must get the register values from a pipeline register. If it violates these criteria, your program will get a 0 point.

Here's the main loop in run(). Add to this code, but don't otherwise modify it (and leave the comments as is) so we can understand your program more easily.

```
while (1) {

printState(&state);

/* check for halt */
if (opcode(state.MEMWB.instr) == HALT) {
        printf("machine halted\n");
        printf("total of %d cycles executed\n", state.cycles);
        exit(0);
}
```

```
newState = state;
newState.cycles++;

/* --------------------- IF stage --------------------- */

/* --------------------- ID stage --------------------- */

/* --------------------- EX stage --------------------- */

/* --------------------- MEM stage --------------------- */

/* --------------------- WB stage --------------------- */

state = newState; /* this is the last statement before end of the loop.
                     It marks the end of the cycle and updates the
                     current state with the values calculated in this
                     cycle */
}
```

3.2. Halting

At what point does the pipelined computer know to halt? It's incorrect to halt as soon as a halt instruction is fetched because if an earlier branch was actually taken, then the halt instruction could actually have been branched around.
To solve this problem, halt the machine when a halt instruction reaches the MEMWB register. This ensures that previously executed instructions have completed, and it also ensures that the machine won't branch around this halt. This solution is shown above; note how the final printState() call before the check for halt will print the final state of the machine.

3.3. Begin Your Implementation Assuming No Hazards

The easiest way to start is to first write your simulator so that it does not account for data or branch hazards. This will allow you to get started right away. Of course, the simulator will only be able to correctly run assembly-language programs that have no hazards. It is thus the responsibility of the assembly-language programmer to insert noop instructions so that there are no data or branch hazards. This will require putting noops in assembly-language programs after a branch and a number of noops in an assembly-language program before a dependent data operation (it is a good exercise to figure out the minimum number needed in each situation).

3.4. Finish Your Implementation by Accounting for Hazards

Modifying your first implementation to account for data and branch hazards will probably be the hardest part of this assignment.

3.4.1. Handling data hazard in general

Use data forwarding to resolve most data hazards. The ALU should be able to take its inputs from any pipeline register (instead of just the IDEX register). There is no need for internal forwarding within the register file. For this case of forwarding, you'll instead forward data from the WBEND pipeline register. Remember to take the most recent data (e.g., data in the EXMEM register gets priority over data in the MEMWB register). ONLY FORWARD DATA TO THE EX STAGE (not to memory).
You will need to stall for one type of data hazard: a lw followed by an instruction that uses the register being loaded.

3.4.2 Handling branch hazard in general

Use branch-not-taken to resolve most branch hazards, and decide whether or not to branch in the MEM stage. This requires you to discard instructions if it turns out that the branch really was taken. To discard instructions, change the relevant instructions in the pipeline to the noop instruction (0x1c00000). Do not use any other branch optimizations, e.g., resolving branches earlier, more advanced branch prediction, or special handling for short forward branches.

3.4.3 For full credit (Important!!)

You must add data/branch hazard resolving logic. If they can handle any one case to resolve each hazard, it will be okay for full credit. Therefore, you need to submit two your own test assembly codes to prove your hazard resolution with explanation. (1 for data hazard, and 1 for branch hazard)

4. Running Your Program

Your simulator should be run using the same command format specified in Project 1:

```
simulate program.mc > output
```

You should use the solution assembler from Project 1 to create the machine-code file that your simulator will run (since that's how we'll test it). The solution assembler can be found at Portal.

5. Test Cases

You will submit your suite of test cases together with your simulator, and we will grade the test suite. 5 short assembly-language programs will be okay for full credit, and 2 of them must be used to verify your hazard handling logic.

6. Grading and Formatting

We will grade primarily on functionality. In particular, we will run your program on various assembly-language programs and check the contents of your memory, registers, and pipeline registers at each cycle. All these assembly-language programs will be hazard-free.
Since we'll be grading on getting the exact right answers (both at the end of the run and being cycle-accurate throughout the run), it behooves you to spend a lot of time writing test assembly-language programs and testing your program. Events must happen on the right cycle (e.g., stall the exact number of cycles needed, write the branch target into the PC at exactly the right cycle, halt at the exact right cycle, stalling only when needed).
In order to prove that your simulator handle at least one data/branch hazards, at least two of your test cases are required to have data/branch hazards. In the test case assembly files, you must add some explanation about how to efficiently handle data/branch hazards.

For easy and fair grading, you must be careful to follow the exact formatting rules in the project description:

1) Don't modify printState() at all.

2) There should be only ONE call to printState() in your program, which is the printState() shown in Section 3.1. Do not put in any extra printState() calls (you can put these in for debugging, but take them out before submitting the program).

3) Initialize all values correctly.
    a. state.numMemory should be set to the number of memory words in the machine-code file.

b. state.cycles should be initialized to 0.
c. pc and all registers should be initialized to 0.
d. the instruction field in all pipeline registers should be initialized to the noop instruction (0x1c00000).

4) Check your program's output on the sample assembly-language program and output at the end of this handout.

5) Pay particular attention to what stage various operations are done in. For example, PC is incremented in the IF stage, so the IFID register should have PC+1. Also, the sign-extender is in the ID stage, so the IDEX register should contain the value of offsetField AFTER calling convertNum().

6) Don't print the sequence "@@@" anywhere except in printState().

7. Turning in the Project

Use the GITLab to submit your files

Here are the files you should submit:

1) pipeline simulator (part 2)
   a. C program for your simulator (name should end in ".c")
   b. suite of test cases (each test case is an assembly-language program in a separate file and at least 2 files must include detailed explanation about hazard handling in comments)
   c. Submission file name is "simulator.c, testcase1.as, testcase2.as, testcase3.as, testcase4.as, testcase5.as"

example:
simulator.c testcase1.as testcase2.as testcase3.as testcase4.as testcase5.as

Your simulator must be in a single C file. We will compile your program on a Linux workstation using "gcc program.c -lm", so your program should not require additional compiler flags or libraries.

The official time of submission for your project will be the time the last file is sent. If you send in anything after the due date, your project will be considered late.

8. Program Fragment

Here's the code for printState() and associated functions. Don't modify this code at all.

```
void
printState(stateType *statePtr)
{
    int i;
    printf("\n@@@\nstate before cycle %d starts\n", statePtr->cycles);
    printf("\tpc %d\n", statePtr->pc);

    printf("\tdata memory:\n");
        for (i=0; i<statePtr->numMemory; i++) {
            printf("\t\tdataMem[ %d ] %d\n", i, statePtr->dataMem[i]);
        }
    printf("\tregisters:\n");
        for (i=0; i<NUMREGS; i++) {
            printf("\t\treg[ %d ] %d\n", i, statePtr->reg[i]);
        }
    printf("\tIFID:\n");
        printf("\t\tinstruction ");
        printInstruction(statePtr->IFID.instr);
        printf("\t\tpcPlus1 %d\n", statePtr->IFID.pcPlus1);
    printf("\tIDEX:\n");
        printf("\t\tinstruction ");
        printInstruction(statePtr->IDEX.instr);
        printf("\t\tpcPlus1 %d\n", statePtr->IDEX.pcPlus1);
        printf("\t\treadRegA %d\n", statePtr->IDEX.readRegA);
        printf("\t\treadRegB %d\n", statePtr->IDEX.readRegB);
        printf("\t\toffset %d\n", statePtr->IDEX.offset);
    printf("\tEXMEM:\n");
        printf("\t\tinstruction ");
        printInstruction(statePtr->EXMEM.instr);
        printf("\t\tbranchTarget %d\n", statePtr->EXMEM.branchTarget);
        printf("\t\taluResult %d\n", statePtr->EXMEM.aluResult);
        printf("\t\treadRegB %d\n", statePtr->EXMEM.readRegB);
    printf("\tMEMWB:\n");
        printf("\t\tinstruction ");
        printInstruction(statePtr->MEMWB.instr);
        printf("\t\twriteData %d\n", statePtr->MEMWB.writeData);
    printf("\tWBEND:\n");
```

```
        printf("\t\tinstruction ");
        printInstruction(statePtr->WBEND.instr);
        printf("\t\twriteData %d\n", statePtr->WBEND.writeData);
}

int
field0(int instruction)
{
        return( (instruction>>19) & 0x7);
}

int
field1(int instruction)
{
        return( (instruction>>16) & 0x7);
}

int
field2(int instruction)
{
        return(instruction & 0xFFFF);
}

int
opcode(int instruction)
{
        return(instruction>>22);
}

void
printInstruction(int instr)
{

        char opcodeString[10];

        if (opcode(instr) == ADD) {
                strcpy(opcodeString, "add");
        } else if (opcode(instr) == NOR) {
                strcpy(opcodeString, "nor");
        } else if (opcode(instr) == LW) {
                strcpy(opcodeString, "lw");
```

```
        } else if (opcode(instr) == SW) {
                strcpy(opcodeString, "sw");
        } else if (opcode(instr) == BEQ) {
                strcpy(opcodeString, "beq");
        } else if (opcode(instr) == JALR) {
                strcpy(opcodeString, "jalr");
        } else if (opcode(instr) == HALT) {
                strcpy(opcodeString, "halt");
        } else if (opcode(instr) == NOOP) {
                strcpy(opcodeString, "noop");
        } else {
                strcpy(opcodeString, "data");
    }
    printf("%s %d %d %d\n", opcodeString, field0(instr), field1(instr),
                field2(instr));
}
```

9. Sample Assembly-Language Program and Output

Here is a sample assembly-language program:

```
        lw      0       1       data1   $1= mem[data1]
        halt
data1   .fill   12345
```

and its corresponding output. Note especially how halt is done (the add 0 0 0 instructions after the halt are from memory locations after the halt, which were initialized to 0). The add 0 0 12345 instruction comes from the .fill line. When converted to machine code, 12345 coming from a .fill looks the same as add 0 0 12345. This instruction will propagate through the pipeline and the addition will be performed. However, the register file is never modified due to this instruction because when the halt instruction reaches the execution stage, the program exits.

```
memory[0]=8454146
memory[1]=25165824
memory[2]=12345
3 memory words
        instruction memory:
                instrMem[ 0 ] lw 0 1 2
                instrMem[ 1 ] halt 0 0 0
```

```
                instrMem[ 2 ] add 0 0 12345

@@@
state before cycle 0 starts
        pc 0
        data memory:
                dataMem[ 0 ] 8454146
                dataMem[ 1 ] 25165824
                dataMem[ 2 ] 12345
        registers:
                reg[ 0 ] 0
                reg[ 1 ] 0
                reg[ 2 ] 0
                reg[ 3 ] 0
                reg[ 4 ] 0
                reg[ 5 ] 0
                reg[ 6 ] 0
                reg[ 7 ] 0
        IFID:
                instruction noop 0 0 0
                pcPlus1 -12973480
        IDEX:
                instruction noop 0 0 0
                pcPlus1 0
                readRegA 6
                readRegB 1
                offset 0
        EXMEM:
                instruction noop 0 0 0
                branchTarget -12974332
                aluResult -14024712
                readRegB 12
        MEMWB:
                instruction noop 0 0 0
                writeData -14040720
        WBEND:
                instruction noop 0 0 0
                writeData -4262240

@@@
state before cycle 1 starts
```

```
        pc 1
        data memory:
                dataMem[ 0 ] 8454146
                dataMem[ 1 ] 25165824
                dataMem[ 2 ] 12345
        registers:
                reg[ 0 ] 0
                reg[ 1 ] 0
                reg[ 2 ] 0
                reg[ 3 ] 0
                reg[ 4 ] 0
                reg[ 5 ] 0
                reg[ 6 ] 0
                reg[ 7 ] 0
        IFID:
                instruction lw 0 1 2
                pcPlus1 1
        IDEX:
                instruction noop 0 0 0
                pcPlus1 -12973480
                readRegA 0
                readRegB 0
                offset 0
        EXMEM:
                instruction noop 0 0 0
                branchTarget 0
                aluResult -14024712
                readRegB 12
        MEMWB:
                instruction noop 0 0 0
                writeData -14040720
        WBEND:
                instruction noop 0 0 0
                writeData -14040720

@@@
state before cycle 2 starts
        pc 2
        data memory:
                dataMem[ 0 ] 8454146
                dataMem[ 1 ] 25165824
```

```
        dataMem[ 2 ] 12345
    registers:
        reg[ 0 ] 0
        reg[ 1 ] 0
        reg[ 2 ] 0
        reg[ 3 ] 0
        reg[ 4 ] 0
        reg[ 5 ] 0
        reg[ 6 ] 0
        reg[ 7 ] 0
    IFID:
        instruction halt 0 0 0
        pcPlus1 2
    IDEX:
        instruction lw 0 1 2
        pcPlus1 1
        readRegA 0
        readRegB 0
        offset 2
    EXMEM:
        instruction noop 0 0 0
        branchTarget -12973480
        aluResult -14024712
        readRegB 12
    MEMWB:
        instruction noop 0 0 0
        writeData -14040720
    WBEND:
        instruction noop 0 0 0
        writeData -14040720

@@@
state before cycle 3 starts
    pc 3
    data memory:
        dataMem[ 0 ] 8454146
        dataMem[ 1 ] 25165824
        dataMem[ 2 ] 12345
    registers:
        reg[ 0 ] 0
        reg[ 1 ] 0
```

```
                reg[ 2 ] 0
                reg[ 3 ] 0
                reg[ 4 ] 0
                reg[ 5 ] 0
                reg[ 6 ] 0
                reg[ 7 ] 0
        IFID:
                instruction add 0 0 12345
                pcPlus1 3
        IDEX:
                instruction halt 0 0 0
                pcPlus1 2
                readRegA 0
                readRegB 0
                offset 0
        EXMEM:
                instruction lw 0 1 2
                branchTarget 3
                aluResult 2
                readRegB 0
        MEMWB:
                instruction noop 0 0 0
                writeData -14040720
        WBEND:
                instruction noop 0 0 0
                writeData -14040720

@@@
state before cycle 4 starts
        pc 4
        data memory:
                dataMem[ 0 ] 8454146
                dataMem[ 1 ] 25165824
                dataMem[ 2 ] 12345
        registers:
                reg[ 0 ] 0
                reg[ 1 ] 0
                reg[ 2 ] 0
                reg[ 3 ] 0
                reg[ 4 ] 0
                reg[ 5 ] 0
```

```
                reg[ 6 ] 0
                reg[ 7 ] 0
        IFID:
                instruction add 0 0 0
                pcPlus1 4
        IDEX:
                instruction add 0 0 12345
                pcPlus1 3
                readRegA 0
                readRegB 0
                offset 12345
        EXMEM:
                instruction halt 0 0 0
                branchTarget 2
                aluResult 2
                readRegB 0
        MEMWB:
                instruction lw 0 1 2
                writeData 12345
        WBEND:
                instruction noop 0 0 0
                writeData -14040720

@@@
state before cycle 5 starts
        pc 5
        data memory:
                dataMem[ 0 ] 8454146
                dataMem[ 1 ] 25165824
                dataMem[ 2 ] 12345
        registers:
                reg[ 0 ] 0
                reg[ 1 ] 12345
                reg[ 2 ] 0
                reg[ 3 ] 0
                reg[ 4 ] 0
                reg[ 5 ] 0
                reg[ 6 ] 0
                reg[ 7 ] 0
        IFID:
                instruction add 0 0 0
```

```
            pcPlus1 5
        IDEX:
            instruction add 0 0 0
            pcPlus1 4
            readRegA 0
            readRegB 0
            offset 0
        EXMEM:
            instruction add 0 0 12345
            branchTarget 12348
            aluResult 0
            readRegB 0
        MEMWB:
            instruction halt 0 0 0
            writeData 12345
        WBEND:
            instruction lw 0 1 2
            writeData 12345
machine halted
total of 5 cycles executed
```