CISC 340 – Project 3
Due: 11:59 pm, Monday, April 20
(15 points)

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 UST-3400 complete with data forwarding and simple branch prediction.
You will write procedural C.
You will use GCC on the departmental linux system (rusty).
You will do this project with a partner.

2. UST-3400 (Rip Van saWinkle) Pipelined Implementation

For this project we will use the datapath from class as a basis. This is similar to the one from Chapter 4 of Patterson and Hennessy (Page 287). Of course, since the MIPS and UST-3400 architectures are slightly different, we had to make a few minor changes to the book's datapath:

1) Instead of a "4" input in the PC's adder, we will use a "1", since the UST-3400 is word- addressed instead of byte addressed.
2) The instruction bit fields have to be modified to suit the UST-3400 ISA.
3) The "shift left 2" component is not necessary, since both offsetfield for branches and the PC use word addressing.

The main difference between the Project 3 design and the pipelining done in the book is that we will add a pipeline register AFTER the write-back stage (the WB/END pipeline register). This will be used to simplify data forwarding so that the register file does not have to do any internal forwarding.

To follow the pipelining done in class as closely as possible, we will use the MIPS clocking scheme (e.g., register file and memory writes require the data to be present for the whole cycle). This should give us the pipelined datapath we discussed in class.

2.2 JALR

You will not implement the jalr instruction from the UST-3400. Taking out jalr eliminates several dependencies/hazards.

2.3 Memory

The memory unit can be accessed directly as an array (similar to Project 1).

Note in the "example" of the machine state 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'll have the same contents in the beginning). 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 via a coherency scheme.)

2.4 Pipeline Registers

To simplify the project and make the output formats uniform, you will use the following code and structs to hold the contents of the pipeline register. Note that the instruction should get passed down the pipeline in its entirety. (For simplicity in reporting.)

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

#define ADD 0
#define NAND 1
#define LW 2
#define SW 3
#define BEQ 4
#define JALR 5 /* JALR – not implemented in 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 readreg;
} 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 */
        int fetched;      /* Total number of instructions fetched */
        int retired;      /* Total number of completed instructions */
        int branches;  /* Total number of branches executed */
        int mispreds;  /* Number of branch mispredictions*/
} statetype;
```

3. Problem

3.1 Basic Structure

Your task is to write a **cycle-accurate** pipelined simulator for the UST-3400. I recommend you start with the Project 1 simulator. The main modification will have to do with <u>when</u> your Project 1 operations happen.

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).

Your primary function should be a loop, where each iteration through the loop executes one clock cycle. At the beginning of the cycle, print the complete state of the machine (I've included sample code and sample output for the "printstate" function at the end of this document). In the body of the loop, you will figure out what the new state of the machine (memory, registers, pipeline registers) will be at the end of the cycle. Conceptually all stages of the pipeline compute their new state simultaneously. Since statements execute sequentially in most programming languages, rather than simultaneously, you will need two state variables: state and newstate. 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 current cycle. Each state of the pipeline will modify the newstate variable using the current values in the state variable. E.g., in the ID stage, you will have a statement like

```
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 done in the textbook and the execution of multiple instruction should be overlapped. The ID stage should be the ONLY stage that reads the register file; the other stages must get their values from a pipeline register. *<u>If your project violates these criteria, your submission will get a **ZERO**. Seriously.</u>*

Here's the beginnings of the primary loop. **<u>Please include these exact comments in your own code, so that I can discern what is going on easily!</u>**

```
while(1){
        printstate(&state);

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

```
            printf("total of %d instructions retired\n", state.retired);
            printf("total of %d branches executed\n", state.branches);
            printf("total of %d branch mispredictions\n", state.mispreds);
            exit(0);
        }
        newstate = state;
        newstate.cycles++;

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

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

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

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

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

        state = newstate;       /* this is the last statement before the end of the loop.
                                   It marks the end of the cycle and updates the current
                                   state with the values calculated in this cycle
                                   – AKA "Clock Tick". */
}
```

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 functionality 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 means putting a number of noops in an assembly-language program after a branch and a number of noops in an assembly language program before a dependent data operation (it's 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.

Use data forwarding to resolve most data hazards. I.e., the ALU should be able to take its inputs from any pipeline register (instead of just the IDEX register). There is no need for forwarding within the register file (as the textbook has on page 311). 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 as alluded to in the text)**.

You will need to stall for one type of data hazard: a lw followed by an instruction that uses the register being loaded. Implement this stall as shown on page 315 of the text.

Use branch-not-taken prediction to resolve most branch hazards, and decide whether or not to branch in the MEM stage (Figure 4.62 on page 320 of the text). 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, special handling for short forward branches, etc.).

4. Running Your Program

Your simulator should be run using the same command format specified in Project 1, and be compatible with the same machine-code inputs.

I recommend using the solution simulator from Project 1 as a starting point. You should use the solution assembler from Project 1 to create the machine-code file that your simulator will run (since that's how I'll test it). If you doubt the accuracy of your assembler, an "official" assembler will be posted on Canvas for your use. If you doubt the accuracy of your simulator, an "official" simulator will be posted on Canvas for your use.

5. Test Cases

An integral (and graded) part of writing your pipeline simulator will be to write a suite of test cases to validate any UST-3400 pipeline simulator. This is common practice in the real world – software (and hardware) companies maintain a suite of test cases for their programs and use this suite to check the program's correctness after a change. Writing a comprehensive suite of test cases will deepen your understanding of the project specification and your program, and it will help you a lot as you debug your program.

The test cases for this project will be short assembly-language programs that, after being assembled into machine code, serve as input to a simulator. You will submit your suite of test cases together with your simulator, and we will grade your test suite according to how thoroughly it exercises an UST-3400 simulator. Each test case may execute at most 100 cycles on a correct simulator, and your test suite may contain up to 20 test cases. These limits are much (MUCH) larger than needed for full credit.

Writing good test cases for this project will require more thinking than the test suites in Project 1. A pipeline simulator is much (**MUCH**) more complex than the behavioral simulator, and the bugs that should be tested for are correspondingly more complex. Randomly choosing a few instructions is unlikely to expose many pipelining bugs. Think about how to test systematically for pipeline-specific conditions, such as data forwarding, branching, and stalling. As you write the code for your simulator, keep notes on what different conditions you've tested for (e.g., forwarding from different stages).

6. Grading and formatting

I will grade primarily on functionality. In particular, I will run your program on various assembly-language programs and check the contents of your memory, registers, and pipeline registers at each cycle. Most of these assembly-language programs will have hazards; a few will be hazard-free. Since I'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 simulator.

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, etc.). **Accuracy of these details is important!**

The best way to debug your program is to generate your own test cases, figure out the correct answers, and compare your program's output to the correct answers. This is also one of the best ways to learn the concepts in the project.

The student suite of test cases for the simulator will be graded according to how thoroughly they test a UST-3400 pipeline simulator. I will judge thoroughness of the test suite by well it exposes potential bugs in a pipeline simulator.

Some suggestions of details to keep an eye on (and ideas to help your professor keep his sanity while grading):

1) Have your "printstate" function print out EXACTLY in the format of the example!

2) Make sure to initialize all values correctly:
   a. state.numMemory should be set to the number of memory words in the machine-code file.
   b. state.cycles/fetched/retired/branches/mispreds 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 set to the noop instruction (0x1c00000).
3) Check your program's output on the sample assembly-language program at the end of this document, and compare it with the output provided.
4) Pay particular attention to what stage various operations are done in! For example, PC is incremented in the IF stage, so the IFID registers should contain the value PC+1. Also, the sign-extender is in the ID stage, so the IDEX register should contain the value of offsetfield AFTER calling signextend().

7. Turning in the Project

The project will be turned in via Canvas. The following components (**and structure!**) are necessary for submission:

- code for your simulator
- a Makefile to compile your code
- Sub-Folder of assembler test files (assembly code and machine!) named tests
- A README file with instructions and explanations of files. Ensure that you are clear about the purpose of each file of your test suite in your explanations
- An overview document (pdf) telling me how your simulator works. This document should also communicate any difficulties you had in developing each program, as well as any shortcomings that remain (e.g., did not finish implementing load-stalls).

Place all of this stuff inside a directory named “project3_username1_username2” using whatever your login names are. You can then zip this directory using the linux command “zip -r project3_username1_username2.zip project3_username1_username2”. Submit the zip file via Canvas. Only one submission per group!
NOTE: When I unzip your submission I should have a single directory, not a whole bunch of files that I have to clean up.

8. Output format

Here's the code for printstate and associated functions to help in understanding the instruction flow through the pipeline.

```
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) == NAND) {
         strcpy(opcodestring, "nand");
   } 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");
    }

   if(opcode(instr) == ADD || opcode(instr) == NAND){
        printf("%s %d %d %d\n", opcodestring, field2(instr), field0(instr), field1(instr));
   }else if(0 == strcmp(opcodestring, "data")){
        printf("%s %d\n", opcodestring, signextend(field2(instr)));
   }else{
        printf("%s %d %d %d\n", opcodestring, field0(instr), field1(instr),
            signextend(field2(instr)));
   }
}

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\treadreg %d\n", stateptr->EXMEM.readreg);
    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);
}
```

9. Sample Assembly-Language Program and Output

Here is a sample assembly-language program:

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

and its corresponding output is below. 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). Do you know where the add 12345 0 0 instruction came from?

```
@@@
state before cycle 0 starts
        pc 0
        data memory:
                datamem[ 0 ] 8912898
                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 0
        IDEX:
                instruction noop 0 0 0
                pcplus1 0
                readregA 0
                readregB 0
                offset 0
        EXMEM:
                instruction noop 0 0 0
                branchtarget 0
                aluresult 0
                readreg 0
        MEMWB:
                instruction noop 0 0 0
                writedata 0
        WBEND:
                instruction noop 0 0 0
                writedata 0
```

@@@
state before cycle 1 starts
	pc 1
	data memory:
		datamem[0] 8912898
		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 1 0 2
		pcplus1 1
	IDEX:
		instruction noop 0 0 0
		pcplus1 0
		readregA 0
		readregB 0
		offset 0
	EXMEM:
		instruction noop 0 0 0
		branchtarget 0
		aluresult 0
		readreg 0
	MEMWB:
		instruction noop 0 0 0
		writedata 0
	WBEND:
		instruction noop 0 0 0
		writedata 0

@@@
state before cycle 2 starts
	pc 2
	data memory:
		datamem[0] 8912898
		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 1 0 2
pcplus1 1
readregA 0
readregB 0
offset 2
EXMEM:
instruction noop 0 0 0
branchtarget 0
aluresult 0
readreg 0
MEMWB:
instruction noop 0 0 0
writedata 0
WBEND:
instruction noop 0 0 0
writedata 0

@@@
state before cycle 3 starts
pc 3
data memory:
datamem[0] 8912898
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 12345 0 0
pcplus1 3
IDEX:
instruction halt 0 0 0
pcplus1 2
readregA 0
readregB 0

offset 0
EXMEM:
instruction lw 1 0 2
branchtarget 3
aluresult 2
readreg 0
MEMWB:
instruction noop 0 0 0
writedata 0
WBEND:
instruction noop 0 0 0
writedata 0

@@@
state before cycle 4 starts
pc 4
data memory:
datamem[0] 8912898
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 12345 0 0
pcplus1 3
readregA 0
readregB 0
offset 12345
EXMEM:
instruction halt 0 0 0
branchtarget 2
aluresult 0
readreg 0
MEMWB:
instruction lw 1 0 2
writedata 12345
WBEND:
instruction noop 0 0 0
writedata 0

@@@
state before cycle 5 starts
	pc 5
	data memory:
		datamem[0] 8912898
		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 12345 0 0
		branchtarget 12348
		aluresult 0
		readreg 0
	MEMWB:
		instruction halt 0 0 0
		writedata 0
	WBEND:
		instruction lw 1 0 2
		writedata 12345
machine halted
total of 5 cycles executed
total of 2 instructions fetched
total of 2 instructions retired
total of 0 branches executed
total of 0 branch mispredictions