built on 2021/09/14 at 15:31:29 ***due:*** *Wednesday, October 6th @ 11:59pm*

Now that you have finish the single-cycle version of our simulator. We are going to improve the performance of our design through two concepts we learned in class: Pipelining and Data Forwarding.

Introduction

For this assignment, you will extend your Lab 1 simulator so that they can support pipelining and data forwarding. In this exercise, we will assume our processor support the same set of instruction (See Table 1 for your reference.)

J	JAL	BEQ	BNE	BLEZ	BGTZ
ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI
XORI	LUI	LB	LH	LW	LBU
LHU	SB	SH	SW	BLTZ	BGEZ
BLTZAL	BGEZAL	SLL	SRL	SRA	SLLV
SRLV	SRAV	JR	JALR	ADD	ADDU
SUB	SUBU	AND	OR	XOR	NOR
SLT	SLTU	MULT	MFHI	MFLO	MTHI
MTLO	MULTU	DIV	DIVU	SYSCALL	

Table 1: List of all the MIPS instructions to be implemented

The simulator will process an input file that contains a MIPS program. Each line of the input file corresponds to a single MIPS instruction written as a hexadecimal string. For example, `2402000a` is the hexadecimal representation of `addiu` $v0$,`zero, 10`. We will provide several input files. But you should also create additional input files in order to test your simulator more comprehensively.

The Shell

Similar to Lab 1, the shell supports the following commands:

1. `go`: simulates the program until it indicates that the simulator should halt. (As we de

   ne below, this is when a SYSCALL instruction is executed when the value in `$v0` (register 2) is equal to `0x0A`.)

2. `run` <n>: simulates the execution of the machine for n instructions.

3. `mdump` <low> <high>: dumps the contents of memory, from location low to location high to the screen and the dump file (`dumpsim`).

4. `rdump`: dump the current instruction count, the contents of `R0` – `R31`, and the PC to the screen and the dump file (`dumpsim`).

5. input `reg_num reg_val`: set general purpose register `reg_num` to value `reg_val`.

6. `high value`: set the HI register to value.

7. `low value`: set the LO register to value.

8. `?`: print out a list of all shell commands.

9. `quit:` quit the shell.

1 The Simulation Routine

With your single-cycle implementation, we now ask you to separate the processor so that they executed instructions in a pipelined manner (with data forwarding).

1.1 Pipelining

The simulator will read new instructions and feed new instructions to the fetch unit whenever it can proceed to fetch the next instruction (one instruction at a time). After each instruction retires through the writeback stage, the simulator will modify the MIPS architectural state: values stored in registers and memory. The simulator is partitioned into two main sections: the (1) shell and the (2) simulation routine. Your job is to implement the simulation routine. *Please note that the architectural states are not updated until after the writeback stage even with data forwarding. Data forwarding only forward operands values to future instructions in the pipeline.*

To support pipelining, your simulator should breaks operations based on what they do into 5 different pipeline stages:

1. FETCH: handle reading the content of the current instructions (based on the PC), and update PC.
2. DECODE: handle reading the operation and sending out all the control signals throughout the design.
3. EXECUTE: handle the actual operation (within the scope of the ALU) for compute-related instructions as well as control instructions that require results from the ALUs.
4. MEMORY: handle all memory-related instructions.
5. WRITEBACK: Handle all the modification to the register files as well as the PC values.

2 Your Tasks

2.1 Pipeline Your Design

With your single-cycle implementation, your first task in this lab is to pipeline all the operations across the 5-stage pipeline we discussed above. This can be done by first creating the pipeline registers that can temporarily store intermediate values required for the instructions (so that the design can hold these values across the pipeline stages).

Once the pipeline registers are created, your next task is to connect all the components to the pipeline registers such that all parts can correctly process the instructions step-by-step through the pipeline.

When you have a functioning pipeline with all the pipeline registers, you will then need to properly stall the pipeline based on the stall condition (i.e., data dependency).

2.2 Data Forwarding

Once you are done with your pipelined design with stalling, it is now time to implement data forwarding. To do this, operands that becomes available and is needed by any follow-up instructions should be forwarded to its corresponding pipeline stages.

To actually perform this task, the following steps is provided as hints.

1. Draw out your design (this should actually be done regardless)
2. Based on the stalling condition, check what source information is missing (that cause the stall)

3. Then, figure out what unit first produce the data that stall your pipeline
4. If you can forward the data (i.e., data becomes ready), forward the data to the demanding source and resume execution.

2.3 MIPS Instruction-level Simulator (with Pipelining and Data Forwarding)

With your code from Lab 1, the function `process_instruction()` in `sim.c` should be extended to handle its corresponding pipelined operations. **Feels free to add any separate files to handle all the pipeline registers and data forwarding**.

Similar to Lab 1, the `process_instruction()` function should be able to simulate the instruction-level execution of the following MIPS instructions:

Note that for the SYSCALL instruction, you only need to implement the following behavior: if the register `$v0` (register 2) has value `0x0A` (decimal 10) when SYSCALL is executed, then the `go` command should stop its simulation loop and return to the simulator shell's prompt. If `$v0` has any other value, the instruction should have no effect. No registers are modified in either case, except that PC is incremented to the next instruction as usual. The `process_instruction()` function that you write should cause the main simulation loop to terminate by setting the global variable RUN BIT to 0.[1]

The accuracy of your simulator is your main priority. Specifically, make sure the architectural state is correctly updated after the execution of each instruction. We will test your simulator with many input programs (some provided with the handout, some not) in order to ensure that each instruction is simulated correctly. In order to test that your simulator is working correctly, you should run the input programs we provide you with and also write one or more programs using all of the required MIPS instructions that are listed in the table above, and execute them one instruction at a time (run 1). You can use the `rdump` command to verify that the state of the machine is updated correctly after the execution of each instruction.

While the table appears to have many instructions, there are actually only a few unique instruction behaviors with a number of minor variations. You should tackle the instructions in groups: R-type ALU, I-type ALU, LW, SW, Jump, Branch, and so on. The MIPS R4000 User Manual (provided in the tarball) contains the official definition for each instruction in this table (except for SYSCALL, for which we provide a restricted definition above). Please implement only the 32-bit behavior of the instructions (the R4000 also has a 64-bit mode, which we can ignore for the purpose of this and subsequent labs.)

Branch Delay Slot. Unlike the manual, we will implement our architecture without the branch delay slots, which will be covered later in our lecture. For the purposes of Lab 1, this means that branch instructions can update `NEXT_STATE.PC` directly to the branch target when the branch is taken. Furthermore, "jump-and-link" instructions (JAL, JALR, BLTZAL, BGEZAL) store `PC + 4` in `R31`, rather than `PC + 8` as specified in the manual in these instructions' descriptions.

In this lab, you do not need to handle overflow exceptions that can be raised by certain arithmetic instructions (e.g., ADDI). Finally, note that your simulator does not have to handle instructions that we do not include in the table above, or any other invalid instructions. We will only test your simulator with valid code that uses the instructions listed above.

Lab Files

Instead of the starter code, you will continue with your code from Lab 1. However, we will now provide you with more assembly files to test your design. These files are available in the lab handout folder on Canvas.

[1] This SYSCALL behavior is consistent with SPIM, which defines a set of syscalls, which you can find on the MIPS 4000 Manual.

Hand-in Instructions

To submit this assignment, please follow the steps below:

1. Compress all the files into a tarball or zip format with your name clearly indicated
2. Submit the file on Canvas.
3. We will do a live demo showing the functionality of your code.