Department of Electrical and Computer Engineering

The University of Texas at Austin

ECE460N, Fall 2024

Lab Assignment 6

Due: Monday, December 9th, 5:00PM

- **Submission instructions**
- Control store sheet (Excel spreadsheet)
- Sample simulator runs to help you in debugging your simulator (Each of the hex files were simulated cycle by cycle using the "run 1" command and an "idump" was performed after each cycle. *.dump files show the cycle by cycle output of idump. *.state files summarize the contents of the pipeline latches. *.timeline shows a timeline of the execution of the program in the pipeline):
    - example0.asm, example0.hex, example0.dump, example0.state, example0.timeline
    - example1.asm, example1.hex, example1.dump, example1.state, example1.timeline
    - example2.asm, example2.hex, example2.dump, example2.state, example2.timeline
- Note that these test cases are not meant to be exhaustive. You should write your own test cases to make sure that your simulator is working for every instruction and program.

Introduction

For this assignment, you will write a simulator for the pipelined LC-3b. The simulator will take two input files:

1. A file entitled **`ucode`** which holds the control store that is located in the DE stage of the pipeline.
2. A file entitled **`isaprogram`** holding an assembled LC-3b program.

The simulator will execute the input LC-3b program using the control store and the code you write inside the simulator functions to direct the simulation of the datapath and memory components of the LC-3b.

Note: The file **`isaprogram`** is the output you generated in Lab Assignment 1. This file should consist of 4 hex characters per line. Each line of 4 hex characters should be prefixed with '0x'. For example, the instruction `NOT R1, R6` would have been assembled to `1001001110111111`. This instruction would be represented in the isaprogram file as `0x93BF`. The file **`ucode`** is an ASCII file that consists of 64 rows and 23 columns of zeros and ones.

Similar to the simulator you used in Lab 3, the simulator for this lab is partitioned into two main sections, the shell, which allows a user to control the simulator, and the simulation routines, which carry out the simulation.

The Shell

The purpose of the shell is to provide the user with commands to control the execution of the simulator. In order to extract information from the simulator, a file named **`dumpsim`** will be created to hold information requested from the simulator. The shell supports the following commands:

1. go – simulate the program until a HALT instruction is executed.
2. run <n> – simulate the execution of the machine for n cycles
3. mdump <low> <high> – dump the contents of memory, from location low to location high, to the screen and the dump file.
4. rdump – dump the current cycle count, the contents of R0–R7, PC, condition codes to the screen and the dump file.
5. idump – dump the current cycle count, the contents of R0–R7, PC, condition codes, and the state of the pipeline latches to the screen and the dump file. This function also dumps the values of stall signals.
6. ? – print out a list of all shell commands.
7. quit – quit the shell

Do not change any of the code related to the shell. We will be using these functions to grade your program and our grading script expects the output format provided by these functions.

Important!!!

lab 6 is quite sensitive for any modification of the shell code and that is why we specify that you should not change any part of the shell code.

For example, it will cause issues if you move the left bracket of icache_access and dcache_access down a line. Be very careful, because VSCode and other IDEs are likely to create autoformatting spacing.

The correct icache_access and dcache_access function declaration should look like this:

```
void icache_access(int icache_addr, int *read_word, int *icache_r) {

void dcache_access(int dcache_addr, int *read_word, int write_word, int *dcache_r,
            int mem_w0, int mem_w1) {
```

If you happen to change it to:

```
void icache_access(int icache_addr, int *read_word, int *icache_r)
{

void dcache_access(int dcache_addr, int *read_word, int write_word, int *dcache_r,
            int mem_w0, int mem_w1)
{
```

Please change it back. The grader will not like it.

The Simulation Routines

The simulator simulates one processor cycle via the `cycle()` function. This function calls the following functions, each of which corresponds to one of the stages in the pipeline:

1. `FETCH_stage();`
2. `DE_stage();`
3. `AGEX_stage();`
4. `MEM_stage();`
5. `SR_stage();`

The `SR_stage()` function has been written for you to get you started. Your job is to write the remaining four functions. Note: For each stage, your code must fully implement the structures in that stage of the pipeline. Refer to the description of the pipeline, the figures of the pipeline stages, and the list of control signals to figure out what to implement in each pipeline stage. The simulator code provided contains two structures that hold the state of the pipeline latches: `PS` and `NEW_PS`. `PS` contains the state of the pipeline latches during the current clock cycle. `NEW_PS` contains the new values of the pipeline latches that will be latched at the end of the current clock cycle. At the end of each cycle, `NEW_PS` is assigned to `PS` (`PS = NEW_PS`) to simulate the latching of data values into the pipeline registers. You need to make use of these structures while writing the simulation routines. When you need to read a value from a pipeline

latch, you need to read it from the `PS` structure. When you need to update the value in a pipeline latch, you should update it in the `NEW_PS` structure. In other words, the `PS` structure should always be used in the right-hand side of an assignment statement, and the `NEW_PS` structure should always be used on the left-hand side of an assignment statement. Please carefully examine the code related to these two structures as you have to use them in the code you write.

We have also provided you with interfaces to the instruction and data caches: `icache_access` and `dcache_access` functions. You must use these functions to perform accesses to the I-Cache and the D-Cache.

What To Do

First, read and understand the documentation for the pipelined version of LC-3b you are going to implement. You may download it from the link below:

lab6_documentation.pdf

Your job is to implement the pipelined LC-3b microarchitecture exactly as it is described in the above documentation. Some of the logic blocks in the pipeline are left for you to implement. It is advisable that you design these logic blocks on paper before you start writing the simulator code.

The shell has been written for you. From your ECE LRC account, copy the following file to your work directory: **lc3bsim6.c**

At present, the shell reads in the control store and input program and initializes the machine. It is your responsibility to write the correct control store file and to augment the shell with the simulation routines that simulate the activity of the pipelined LC-3b. In particular, you are to write the code to perform the activities of the first four pipeline stages as described above.
Add your code to the end of the shell code. Do not modify the shell code.

The accuracy of your simulator is your main priority. It is suggested that you start out by writing a one instruction program and simulating the execution of this program cycle by cycle using the idump command to verify that the instruction propagates correctly through the pipeline. We suggest that you start out by making sure a simple instruction, like an ADD, flows correctly through the pipeline. Then you can move on to more complicated instructions like loads and branches. Try instructions one by one to make sure each works as it is supposed to. After you get the memory and control instructions working correctly, you can try more complicated programs that contain dependencies and test whether or not your pipeline stalls correctly.

You are not responsible for implementing the RTI instruction. You are also not required to support exception/interrupt handling, although you are encouraged to think about the issues related to the handling of exceptions and interrupts on a pipelined microarchitecture.

Since we will be evaluating your code on linux, you must be sure that your code compiles on one of the ECE linux machines using gcc with the -std=c99 flag. This means that you need to write your code in C such that it conforms to the C99 standard. You should also make sure that your code runs correctly on grader2 machine.

If the value of a control signal is a don't care, you should set that signal to 0. We will be checking the state of the internal pipeline latches and architectural state when testing your simulator. **To receive full credit, the values stored in these latches by your simulator should exactly match the values stored in these latches by a correct simulator.**

What To Turn In

Please submit your lab assignment electronically on grader1. You will submit the following:

1. **`lc3bsim6.c`** – adequately documented source code of your simulator.
2. **`ucode6`** – microcode file.

Optional: Support for Exceptions

For this lab, you may optionally implement support for exceptions and the RTI instruction. **Note that implementing these optional additions will receive no extra credit; this is merely for those students who wish to challenge themselves as an additional learning experience.** Before implementing exception support into the pipeline, please first complete the basic pipeline functionality.

As a starting point for exception support in the LC-3b pipeline, we recommend making many simplifying assumptions. First, begin with support for only two kinds of exceptions: unknown opcode and unaligned access (LDW/STW only initially). The unknown opcode exception should use exception vector x01, and the unaligned access exception should use exception vector x02.

Assume that all programs will only execute in Supervisor Mode with PSR[15]=0. Thus, no support for two levels of privilege will be required. As a consequence, assume that R6 will always contain the Supervisor Stack Pointer; thus, no switching of stack pointers will be necessary.

Note that both RTI and the steps to initiate the exception handler are complex processes: they each require multiple memory accesses, stack pointer modification, and change in PC. The current LC-3b pipeline resolves control instructions in the MEM stage of the pipeline. Because all exceptions can be detected by this stage, modifying the pipeline to support exceptions will require an additional exception state machine in the MEM stage. We can add additional bits to the MEM stage Pipeline Registers to keep track of whether an exception has been detected, and which state of exception processing the MEM stage is currently carrying out in the case of a detected exception. RTI can be implemented with a similar state machine in the decode stage of the pipeline. These state machines will stall the pipeline until they complete.
To support the stack operations required by the RTI instruction and the exception handler setup steps, it is easiest to assume the MEM stage has access to direct R6 read and write ports from the Register File. Appropriate dependency stall logic will need to be included to wait on previous instructions' potential writes to R6. Similar design principles can be applied to the CC (which need to be saved/restored onto the System Stack as part of the PSR).
Both the RTI instruction and the exception handler setup steps will require additional logic to ensure earlier stages of the pipeline are properly invalidated; this can be implemented similarly to the BR.STALL mechanism used by all control instructions currently in the pipeline. One challenge to consider is that while an RTI can be detected at the same time as other control instructions, memory exceptions cannot be detected until AGEX.

Please contact the TA for additional details if you wish to implement support for exceptions.

Lab Assignment 6 Clarifications

NOTE: FAQ's for this semester will be posted here. Please check back regularly.

1. We will check the values of the internal pipeline latches generated by your simulator, so make sure you follow these conventions:

- If the data to be loaded/stored is a byte, set `DATA.SIZE` to 0. Set `DATA.SIZE` to 1 if the data to be loaded/stored is a word.
- A load enable signal (LD.REG or LD.CC) should be set to 1, if the instruction is supposed to write to the structure which is load-enabled by that signal. If the instruction does not write to a structure, the load-enable signal associated with that structure should be set to 0.
- BR.OP should be set to 1, if the instruction's opcode is BR. UNCON.OP should be set to 1, if the opcode of the instruction is JMP, RET, JSR, or JSRR. TRAP.OP should be set to 1 if the opcode of the instruction is TRAP.
- ALUK signals should be 00 for ADD, 01 for AND, 10 for XOR, 11 for PASSB. PASSB lets the ALU pass the B input to the output undisturbed.
- BR.STALL should be set to 1 if the instruction is a control instruction.
- LSHF1 is 1, if the output of the ADDR2MUX should be left-shifted by 1.
- DCACHE.EN should be set to 1, if the instruction is supposed to access data memory.
- DCACHE.RW is 0, if the memory access is a read. It is 1, if the memory access is a write.
- For mux control signals, follow the encodings shown in the figures for the pipeline stages.
- As mentioned in the handout, if the value of a control signal does not matter, set the signal to 0.

2. If the D-Cache is not enabled (V.DCACHE.EN signal is 0), the data output by the D-Cache should be set to 0x0000.

3. Control store entries corresponding to invalid opcodes (opcodes 1010 and 1011) should be set to all 0.

4. Will you check the values in pipeline latches even if the latches are invalid?

If the pipeline latches in a stage are invalid, our grading script is still going to check the values in the latch. The datapath of a stage performs calculations regardless of whether or not the instruction in that stage is valid (unless the valid bit is explicitly input to some logic blocks to gate the calculations). For example, even if AGEX.V is 0, the address generation logic, shifter, and the ALU will still perform calculations based on the data values in AGEX latches and control signals in AGEX.CS latch. At the end of the cycle, calculated outputs of these units will be latched into the MEM latches. Concurrently, AGEX.V is propagated to MEM.V.

5. Because all the TRAP instruction encodings start with 1111 0000, the microinstructions in states 62 and 63 should be don't cares. However, please fill the same microinstructions into states 62 and 63 as the ones in 60 and 61. JMP/RET is similar to the TRAP instruction. It always starts with 1100 000, thus the microinstructions in states 49, 50 and 51 should be don't cares. However, please fill the same microinstructions into states 49, 50 and 51 as the ones in 48 for the same reason as the TRAP instruction. The JSRR should be treated as the same above. Please fill the microinstruction in state 16 into state 17.

6. The shifter in AGEX stage should perform a left shift if IR[5:4] is 10.