Department of Electrical and Computer Engineering

The University of Texas at Austin

EE 460N, Spring 2021
Lab Assignment 4
Code Due: Sunday, April 11th, 11:59 pm
Documentation Due: 5 pm of the day after which you submitted the lab on Gradescope

Make sure your Lab 3 implementation is correct before starting Lab 4. You may use the **submit-ee360n** command on grader2 to test your Lab 3 program. Submit the lc3bsim3.c and ucode3 files first with submit-ee360n -late, then run submit-ee360n -lategrade. You can run submit-ee360n to see all available options. You have unlimited attempts for late grade. See here for more details.

In order to encourage you to fix your lab 3, we will use your late lab 3 grade as a 40% component of your final lab 3 grade. The other 60% component is your initial lab 3 grade. **Take advantage of this policy to improve your lab 3 grade!** See here for more details.

Note: Passing all the tests does not mean that your Lab 3 is correct! An incorrect program may just happen to behave correctly on our test cases. It is your responsibility to ensure that your Lab 3 is correct.

- **Submission instructions**
- **The control store template in Excel and PDF formats.**
- **You can use the following pdf files to show the changes you made to the datapath, state diagram and microsequencer. Please note that you can submit hand drawn diagrams.**
  - **datapath pdf file**
  - **state machine pdf file**
  - **microsequencer file pdf file**

Introduction

The goal of this lab assignment is to extend the LC-3b simulator you wrote in Lab 3 to handle interrupts and exceptions. You will augment the existing LC-3b microarchitecture to support detection and handling of one type of interrupts (timer) and three types of exceptions (protection, unaligned access, and unknown opcode).
The inputs to this new simulator will be:

1. A file named **`ucode4`** which will hold the expanded microcontrol store.
2. A file holding an LC-3b user program that will be loaded into memory starting at memory location `x3000`.
3. A file holding data for the user program. This data will be loaded into memory starting at memory location `xC000`.

4. A file holding the interrupt/exception vector table that will be loaded into memory starting at memory location `x0200`. You should form the contents of this table based on the vector of each interrupt/exception and starting address of the service routine for each interrupt/exception.
5. A file holding an LC-3b system program that will be loaded into memory starting at memory location `x1200`. This will be the timer interrupt service routine.
6. A file holding an LC-3b system program that will be loaded into memory starting at memory location `x1600`. This will be the protection exception handler.
7. A file holding an LC-3b system program that will be loaded into memory starting at memory location `x1A00`. This will be the unaligned access exception handler.
8. A file holding an LC-3b system program that will be loaded into memory starting at memory location `x1C00`. This will be the unknown opcode exception handler.

The simulator will start executing the LC-3b program loaded at `x3000`. The timer interrupt will occur at cycle 300. You are supposed to provide microarchitectural support for interrupt handling. The interrupt handler you write should increment location (word-sized access) `x4000` by 1. You will also need to implement the RTI instruction so that the interrupt handler can transfer control back to the user program.
You will have to provide microarchitectural support for handling exceptions. Most of this support is similar to the support you will add for interrupts. If the user program accesses a memory location that can only be accessed in supervisor mode, a protection exception will occur. The exception service routine you will write should simply halt the machine. If the user program makes an unaligned memory access in either supervisor mode or user mode, an unaligned access exception will occur. The exception service routine again should simply halt the machine. If the user program tries to use an unknown opcode (1010 or 1011), an unknown opcode exception will occur. The exception service routine again should simply halt the machine. **(Important note: we should be able to swap out your exception routines with a routine that returns from an exception. Therefore, make sure you implement the exceptions correctly.)**

New shell code

A new shell code has been written for you:

lc3bsim4.c

You will need to copy and paste the code you wrote for Lab 3 into this new shell code.
To run the simulator, type:

```
lc3bsim4 <micro_code_file> <program_file_1> <program_file_2> ...
```

The first parameter is the microcode file as before. The second parameter is the user program you will write and will be executed by the simulator. The rest of the parameters are program or data files that will be loaded into memory.

Specifications

This Lab consists of 2 parts, which we will describe in detail:

1. Adding support for timer interrupts
2. Adding support for protection, unaligned access, and unknown opcode exceptions

Adding support for timer interrupts

Timer Interrupt

In this lab, 1 timer interrupt will occur at cycle 300. When the timer generates an interrupt, the microarchitecture may be in the middle of executing an instruction. You need to decide exactly when the interrupt is detected and add the necessary microarchitectural support to handle interrupts. This will involve augmenting the state diagram of the LC-3b with additional states, augmenting the microsequencer with additional logic to sequence these new states, and extending the existing microinstructions with additional bits for both the microsequencer and the datapath.

When the interrupt is detected the following actions will be taken by the processor:

1. Compare the incoming interrupt priority level, if that is greater than the current process's priority level (stored in PSR[10:8]), proceed to the next step. Otherwise, do nothing. See Appendix A.4 for more information.
2. The privilege mode (most significant bit of the PSR, program status register) is set to 0, which indicates supervisor-level privilege. The interrupt service routine will be executed with supervisor-level privilege.
3. The priority level (PSR[10:8]) should be updated with the interrupt priority. The priority of the timer interrupt is 3. Note the user program runs at the lowest priority level, which is 0.
4. R6 is set to the supervisor stack pointer if that is not already the case.
5. The old PSR (PSR before the update) and PC are pushed onto the supervisor stack. On a push operation, the supervisor stack pointer is **decremented**, and then the data is written (stored) on the stack. Note that the supervisor stack is different from the user stack. Interrupt service routines can access the supervisor stack using R6. The shell code initializes the supervisor stack pointer to address `x3000`. Note that R6 will refer to the user stack while a user program is running. It will refer to the supervisor stack while the interrupt service routine is being executed. If the system is in user mode when an interrupt is detected, the microarchitecture should transparently switch R6 so that it points to the supervisor stack. You need to implement this "stack switching" in microcode. You will need to modify the datapath, add new states to the state machine, and possibly add new control signals to support this operation. You may add registers to the datapath to save and restore user and supervisor stack pointers.
6. When preparing for an interrupt or exception handler, it is the hardware's responsibility only to preserve R6. Preserving the register values for R0-R5 and R7 must be done by the interrupt/exception handler itself and should not be done by the hardware.
7. The interrupting event supplies its 8-bit interrupt vector (INTV). The interrupt vector for the timer interrupt is `x01`.
8. The processor left-shifts the interrupt vector one bit, yielding `x02`, and adds it to the base address of the interrupt/exception vector table (`x0200`), yielding the address of the

memory location (x0202) that contains the starting address of the interrupt service routine.
9. The contents of memory location x0202, which should be x1200 for this assignment, are read and loaded into the PC.
10. The processor begins execution of the interrupt service routine.

The first step in adding support for interrupts is to determine how to modify the state diagram of the LC-3b to handle interrupts. You will have to augment the microsequencer with additional logic to sequence these new states, and extend the existing microinstructions with additional bits for both the microsequencer and the datapath. You may augment current microinstruction fields and add new fields. You may also add new logic to the datapath. You are free to implement this as you wish, but you must document your method.

RTI Instruction

Next, you have to implement an instruction for returning from an interrupt. This will be used in the interrupt service routine to transfer control back to the interrupted program. This instruction, called RTI (return from interrupt), has the opcode 1000, and pops the old PC and PSR off the supervisor stack. By looking at the old PSR's most significant bit, you can find out if you are returning to user-level. If the RTI transfers control back to a user-level program, then the hardware should R6 to its original value (see Appendix A for details on the RTI instruction).

Adding support for exceptions

Exception handling is very similar to interrupt handling as described above with an important difference, in almost all cases and certainly in the three you will implement, the exception-causing instruction should not be allowed to complete before the exception is handled. Hence, the memory access that causes the protection exception or unaligned exception also should not be allowed to complete. When you detect an exception, you must first return the system to a consistent state by undoing any changes that were made to architectural registers during the processing of the instruction that caused the exception. In your case, this just means decrementing the PC, because no other architectural registers are modified between any possible exception and the start of the instruction that caused it. You will need to change the mode of the machine to supervisor, switch to the supervisor stack, push the old PSR (PSR before PSR[15] is set to 0) and the decremented PC on the supervisor stack and load the PC with the address of the exception service routine. You do not need to change the priority level because the exception handler will be executed on the same priority level as the current running process. The exception causing event supplies its own 8-bit exception vector (EXCV). You can store the appropriate exception vector in a separate register and add this register left-shifted by one to the interrupt/exception vector table base register to get the address of the location that contains the starting address of the exception service routine. You are free to implement this as you wish, but keep in mind the possibility of combining the states used for initiating the interrupt service routine and those used for initiating the exception service routine.

Protection Exception

Protection exceptions occur only when the machine is in user mode, and a memory location in system space (locations x0000 – x2FFF) is accessed by the user program. The exception vector for a protection exception is x02.

You will also write the exception service routine for the protection exception. This routine should start at memory location x1600. For the purposes of this assignment, the exception service routine will simply halt the machine. However, don't rely on this. **We can test your simulator by replacing your exception routine with our routine which returns from the exception**

handler using the RTI instruction instead of halting the machine. Upon return from the exception handler, the instruction that caused the exception should be re-executed.

Unaligned Access Exception

An unaligned access exception occurs whenever a word-size memory access to an unaligned memory address is attempted. The exception vector for an unaligned access exception is x03. You will write the exception service routine for the unaligned access exception. This routine should start at memory location x1A00. For purposes of this assignment, the exception service routine will simply halt the machine. Again, do not rely on this as we can test your simulator by replacing your exception routine with our own routine.
In the event of an unaligned access to a memory location in system space (locations x0000 – x2FFF), you should give priority to the protection exception. For example, if a STW instruction accesses x0001, the protection exception handler should be called rather than the unaligned access exception handler. Upon return from the exception handler, the instruction that caused the exception should be re-executed.

Unknown Opcode Exception

Unknown opcode exceptions occur when the program attempts to execute an instruction with opcode 1010 or 1011. The exception vector for an unknown opcode exception is x04.
You will also write the exception service routine for the unknown opcode exception. This routine should start at memory location x1C00. For the purposes of this assignment, the exception service routine will simply halt the machine. However, don't rely on this. We can test your simulator by replacing your exception routine with our routine which returns from the exception handler using the RTI instruction instead of halting the machine. Upon return from the exception handler, the instruction that caused the exception should be re-executed.

Tips on getting started

When designing the mechanisms to support interrupts and exceptions keep in mind that they are handled very similarly. In your microcode, states for pushing the PSR and PC on stack and loading the address of handler routines could be shared for both exception and interrupt handling.

Writing Code

The user program loaded into x3000 should do the following: initialize memory location x4000 to 1 and calculate the sum of the first 20 bytes stored in the memory locations beginning with xC000. This sum should first be stored at xC014. The next step would depend upon whether you are testing for a protection exception or an unaligned access exception. If you want to cause a protection exception, the user program should next store the sum at x0000. If you want to cause an unaligned access exception, the user program should store the sum at xC017. Finally, if you wish to test for an unknown opcode exception, you can simply use a .FILL pseudo op to create an instruction with an unimplemented opcode (1010 or 1011).

Note: While submitting the user program, only submit the version which tests for protection exception.

The following numbers should be stored at locations xC000:

`x12, x11, x39, x23, x02, xF6, x12, x23, x56, x89, xBC, xEF, x00, x01, x02, x03, x04, x05, x06, x07.`

The interrupt service routine must increment memory location `x4000`.
The protection exception handler, the unaligned access exception handler, and the unknown opcode exception handler that you will submit should simply halt the machine using the `TRAP x25` instruction.

What To Submit to grader

1. Adequately documented source code of your simulator called **`lc3bsim4.c`**.
2. The assembly code for the interrupt service routine, the interrupt/exception vector table, the protection exception handler, the unaligned access exception handler, the unknown opcode exception handler, the user program, and the data for locations `xC000` – `xC013`, called **`int.asm`**, **`vector_table.asm`**, **`except_prot.asm`**, **`except_unaligned.asm`**, **`except_unknown.asm`**, **`add.asm`**, and **`data.asm`**, respectively.
3. The new microcode called **`ucode4`**.
4. A file called **`dumpsim`**. To generate this file, simulate the LC-3b assembly programs by following the instructions at the end of this section.

What to Submit in on Gradescope

1. A **`readme`** file that explains how you implemented this lab assignment. This readme file should describe the following:
    1. Changes you made to the state diagram. Include a picture similar to the state machine that shows the new states you added (only show your changes or mark your changes in a new state diagram). This picture should include the encodings of new states you added. Clearly show where each state fits in the current state diagram. Describe what happens in each new state.
    2. Changes you made to the datapath. Clearly show the new structures added, along with the control signals controlling those structures. Describe the purpose of each structure.
    3. New control signals you added to each microinstruction. Briefly explain what each control signal is used for.
    4. Changes you made to the microsequencer. Draw a logic diagram of your new microsequencer and describe why you made the changes.
    5. *Please submit this pdf file electronically to Gradescope by 5pm of the day after which you submitted the code*

Note this document is not a full lab report. You do not need to document everything you did for the lab. What you do need is clear diagrams of what you added to the datapath, how does each state transition from one to another (based on what signal), how does the microsequencer support those new transitions. Drawing a box and labeling it with "Logic" is ok as long as the input output relationship is clear. (You do not need to do gate level implementation.)

How to generate the dumpsim file

1. Run the simulator with the protection exception causing version of the user program. Dump location `x4000` and the registers once right before the 300th cycle (299th cycle), once after the ISR is done, once after the protection exception halts the execution of the program.
2. Rename the **`dumpsim`** file to another name (say **`dumpsim_temp`**)
3. Rerun the simulator with the unaligned access exception causing version of user program. Dump location `x4000` and the registers after the unaligned access exception halts the execution of the program.
4. Open the **`dumpsim`** file and the **`dumpsim_temp`** file with a text editor. Copy and paste the contents of **`dumpsim`** to the end of **`dumpsim_temp`**.
5. Rename **`dumpsim_temp`** into **`dumpsim`**.

Things To Consider

1. The user stack should start at address `xFDFF` (i.e. the initial user stack pointer value should be `xFE00`) and the supervisor stack should start at address `x2FFF` (i.e. the initial supervisor stack pointer value should be `x3000`).
2. The microcode used for starting the exception service routine is very similar to the microcode that is used for starting the interrupt service routine. Are there any differences? With these differences in mind perhaps you can combine the states used for interrupt and exception handling.
3. The interrupt service routine should not change any register values which are being used by the main program. We should be able to test the service routine you wrote with a main program (and a simulator) written by us and both programs should work correctly. Therefore, if your interrupt service routine destroys any registers, it should save the original values of those registers before destroying them and restore those values before returning control to the main program. Note that this saving/restoring of general purpose registers could also be supported by microcode. Is this a good idea? A related issue is the saving and restoring of condition codes. This has to be done in microcode, it cannot be done by the interrupt service routine. Why?
4. Make sure you check Appendix A for the description of PSR, the RTI instruction, privilege mode, user stack, supervisor stack, and more information on interrupt processing.
5. The functionality of your modifications to support interrupts and exceptions should not depend on the interrupt and exception handlers you write for this assignment. We should be able to replace your interrupt and exception handlers with handlers written by us, and your code should still work correctly. Conversely, the functionality of your interrupt and exception handlers should not depend on the user program you wrote for this assignment. We should be able to use your exception and interrupt handlers in our simulator running a user program different from what you wrote.
6. Basic LC-3b microarchitecture described in Appendix C makes use of 31 of the available 64 states. This leaves you with 33 states to accommodate your changes to support interrupts and exceptions. This is more than enough. Try to avoid expanding the size of the control store. You can avoid this by careful thinking and design before starting coding. Leaving around 10 states empty is more than enough for lab5.
7. You need to support nested interrupts and exceptions for this assignment.

8. **Important note:** We are aware that there are solutions to this lab available in various textbooks and on the internet. That material is "off limits" as far as doing this lab is concerned. Reading such material is not allowed. If you find anything that you think could be not allowed, please check with the instructor or one of the TAs before reading it.

Lab Assignment 4 Clarifications

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

1. Are we changing the datapath/state diagram to *detect* or *generate* the interrupt or exception?

As part of this lab, you are changing the datapath and state diagram to detect all exceptions and the timer interrupt. However, you are only simulating the generation of the timer interrupt.

2. Should INTV/EXCV be left-shifted by 1 before adding to the interrupt/exception vector base register?

Yes. INTV/EXCV is an index into the interrupt/exception vector table and as each entry stores an address, INTV/EXCV has to be left-shifted by one to get the correct offset in this table.

3. You should generate the timer interrupt only once – at cycle 300. You must also take care of setting the global variable INTV to the correct interrupt vector at this time.

4. What kind of file format should we write for the readme file?

Please submit a PDF on Gradescope.

5. You need to change in the shell code which we provided to you the enum `CS_BITS` and the struct `System_Latches`.

6. Your simulator should generate the timer interrupt once at cycle 300. You can implement this by using the `CYCLE_COUNT` variable we provided.

7. What should PSR[15] be initialized to?

You should initialize the privilege mode of the processor (PSR[15]) to 1 (user mode) in your simulator.

8. Can a user program execute the RTI instruction?

In general, no. But for the purpose of this assignment, you can assume that RTI will never be used by a user program. You are not responsible for generating an exception if an RTI instruction is encountered in user mode.

9. Who initializes the user stack pointer?

The user stack pointer is initialized in the user program (if the user program makes use of the stack). The supervisor stack pointer is initialized by the shell code we already provided.

10. What is the size of the data elements our user program is supposed to add?

The elements your user program is supposed to add are *bytes*. You should store the sum of the 20 8-bit two's complement integers as a *16-bit word* at location `xC014`.

11. Can I execute one cycle of the memory access (for example in state 33) before handling the exception?

No. You should not access memory if the access causes an exception.

12. You may declare registers you need as part of the System_Latches struct.

13. TRAP should be able to execute in user mode, i.e. it must be able to access the trap vector table at `x0000` to `x01FF` without causing a protection exception. For this assignment, TRAP **does not** need to change the processor state to execute in system mode.

14. You may add a line to the `initialize()` function in the shell code to set the initial value of the PSR and any other registers that you add to the datapath.

15. Please note that LEA no longer sets condition codes.

16. What hardware support is required for timer interrupts?

Microarchitecture support needs to be added only for detecting interrupts, not generating them. Since you are simulating an interrupt at cycle 300, you can have an if() statement in the cycle() function that sets the interrupt vector (INTV) to the appropriate value using the CYCLE_COUNT variable. Unlike exceptions, you do not have to deal with the interrupt immediately, you may choose to deal with the interrupt whenever it is convenient for you.