

CSE 141 Lab NOTES

- Turn in paper ISA design TODAY!
 - Now, on front table
- Lab 1 Part B Due in 2 weeks (Jan 24)
- Holiday next Monday – no class
 - Office hours that week:
- Webboard – Should be up today or tomorrow!
- LOAD/STORE – CAN'T DO ANYTHING ELSE!

How we'll grade Part A

- Check
 - Things look *generally* OK
 - Not promising you this is the BEST design, but it seems workable
- Frowny Face
 - Serious concerns
 - Not completely designed
 - Not appropriately explained
 - Seems unlikely to function
- Worth 10% of Lab 1 Grade (You get 10% by turning it in)
- If you were not enrolled in lab last Monday, you will not be penalized, but you must show me documentation

What you can do for Part B

- Join a group
- Redesign your ISA
 - You MUST re-turnin a full textual description of your ISA

Part B Information

- Check web site for COPIOUS information
- Look at the “turn in instructions” soon
 - Check out flags your codes must support
- Webboard is the place for questions
 - Once it's up

Part B Deliverables

- (1) The complete description of your ISA, and each instruction.
- (2) An assembler, that consumes your assembly program and outputs the binary format of the program to be executed.
- (3) An emulator that reads in the binary format of your program and an input string, and emulates the execution of your program outputting the correct string, and prints it out.
- (4) The assembly language program from your parser to implement the text formatting algorithm.
- (5) Answers to all the questions asked in the Lab 1 handout

First, write your program in your ISA

- **graphics.asl**
- Your assembly language program (in ASCII text form) to implement the graphics algorithm.
- File format:
 - The assembly code must be heavily commented
 - Each instruction of your program must be on a separate line.
 - Example: ... ld r2, 0(r1) ; load the current character into r2
dec r0 ; reduce the number of characters processed by 1
bnz r0, loop ; if all characters have not been processed go back to the start of the loop ...

Write an assembler in C or Java

- Read in .asl file produce a “binary” file with hexadecimal notation of your program
- 00111100
- 00001111
- 3C
- 0F

What does an assembler do?

- Read input line from text assembly file
- Parse the opcode (usually the 1st word from the input line).
 - Once you know the opcode, you know **part** of the 8-bits you need to create the final 8-bits for the instruction in the binary.
- Based on the opcode, you know if the next words in the input line are registers, addresses, immediate, offsets, etc... Parse these final fields from the input line. At this point you have all of the 8-bits you need to create the instruction.
- Write the 8-bits for this instruction to the binary (a text file, really) in as a 2 character hex number.
- Go back to step 1, until all instructions are processed.

Assembler Details

- The 1st instruction in your program's assembly input should be the 1st instruction that needs to be executed in your program.
- Branch instructions need special mention. They can typically be handled in one of two ways:
 - When writing your assembly input program to execute the text filter program, you can manually (by hand) figure out the branch distances and write the distance as part of the assembly. Note, anytime you add or delete an instruction from your assembly program will require you to go through and recalculate all of these constants.
 - A different option is to use labels. You can have your assembly file consist of labels (e.g., L1) for the destination of the branch. This may require you to "pre-parse" the assembly file once to (A) find all of the labels, and (B) determine how many instructions away the destination is from each branch.
- See the turn in instructions for the list of flags your assembler needs to support (to indicate input and output files, etc).

Write an emulator to “run” your binary

- Read in the .bin file (with hexadecimal notations of your ISA instructions) and “perform” that code (in C or Java)
- BASICALLY:
 - Read in instructions to execute
 - Initialize
 - For each instruction in .bin file, “Do” (in C or Java) whatever that instruction says
 - Dump the complete “data memory” at the end to make sure your program worked correctly.

Initialize

- Note: These won't be done in exactly this way in your final processor design -- more on that later. But your processor will already have the instruction and data memories "loaded" with the program binary and data, respectively. We'll do that here in the emulator by loading the instruction and data "memories" from files.
- Load the hex binary into your instruction memory starting at instruction memory location 0.
- Load the program input into data memory starting at data memory location 10.
- Initialize all constant registers and "implicit" registers as needed to their appropriate values.
- Set the start PC to address 0.

What is instruction and data memory?

- You are “emulating” a processor. A “real” processor would have space to store the program instructions and the data values
 - This is an “array” of space
 - Make an array that can hold data for Imem and Dmem
- Also, a real processor has registers
 - You need to provide storage for them as well.
 - Implicit, explicit
- Processing power
 - Use Java statements

Branches

- Branches might do some math, but they control what instruction to execute next.
- Program Counter!
 - A register
- Make sure that you can “fetch” the next appropriate instruction

Executing

- **Fetch** - fetch/read the 8-bits at the current PC
- **Decode** - look at the correct bits for your ISA to determine the opcode. Once the opcode is determined the rest of the bits are used as needed to determine what registers, immediate, or offsets to use.
- **Execute** - execute the instruction reading the corresponding register and memory data structures.
- **Mem/Reg Writeback** - after you are done executing the instruction update/write the register and data memory as required for that instruction.
- **Set Next PC** - If instruction is NOT a branch set PC to PC+1. If the instruction is a branch set PC to PC+1 if not-taken, otherwise set PC to target instruction of the branch.
- Go back to step 1, until you hit the halt instruction in your binary.

Ending Execution

- To check whether your emulator performed correctly, you'll want to dump the entire contents of your memory to a file
 - For inspection

Details on Emulator

- Remember that the instruction and data memory are **separate** memories. So address 0 of instruction memory is different than address 0 of data memory.
- You can only read or write (not both) 8-bits to data memory per cycle (so also per instruction).
- See the turn in instructions for the list of flags your emulator needs to support (to indicate input and output files, etc).

Check web for turnin instructions

- Cycle count: Performance
 - Your emulator should keep a dynamic count of number of instructions executed.
- IENG 9 is where the code **MUST** run.
 - Am getting IENG9 accounts today or tomorrow for those who don't have them.

Prepping for the next lab

- Install xilinx
 - Link to student download site coming soon!
 - Also, lab in 2444.
- Stay up to date in class
 - Chapter 5 will be the key
 - Single Cycle processor design