CDA 4205L Lab #10: ISA Assembler Design (Part 3)

University of South Florida

Lab Date: March 26

Report due: One week after the lab section, see Canvas

***This is a group lab**. One submission per group is sufficient.*

Welcome to CDA 4205L Lab #10! This is the third portion of the ISA lab that injects in the middle of Lab #4. The goal of this lab is for you to understand how to recognize and remove data hazards from the assembly code created in part 1. This step will rearrange the sequencing of the assembly code such that the computation and output remains the same, but data hazards are removed.

Prelab

Pipeline Hazards

In a pipelined datapath, a "hazard" is a condition that prevents the normal execution of the code. Hazards include structural, data, and control. Structural hazards refer to conditions where the same hardware unit is needed by more than one instruction at the same time. For example, if there is one ALU, and two instructions in the pipeline both need that ALU at the same time, there would be a structural hazard, and one instruction would have to wait. Data hazards, also called "read before write" hazards, refer to situations where one instruction attempts to use a value in a source register that has not yet been written back. It may have already been computed or loaded, but registers are not updated until the final stage of the pipeline. Finally, control hazards are conditions where there may be a change in control flow in the code, e.g., a branch (if/else). Branches are not evaluated until the second stage of the pipeline, and hence, the CPU may load the wrong next instruction.

The RISC-V pipeline discussed in class does not have structural hazards. However, data and control hazards are common and must be addressed. In hardware, the CPU will detect hazards and try to either mitigate them (e.g., by using forwarding or branch prediction). In some cases, forwarding can "solve" a hazard by allowing values computed in the EX stage to "bypass" the register (at least, temporarily) and be passed along directly to the instruction that requires it. Sometimes, however, a stall – waiting for data to be ready – is inevitable.

A good compiler can also detect potential hazards and try to reduce the number of stalls needed for the correct execution of the code. This can be achieved by reordering some of the instructions. For example, if a dependency exists between instruction A and instruction B that would otherwise require a stall cycle, a completely unrelated instruction could be inserted between them so no time is wasted.

Stall Cycles

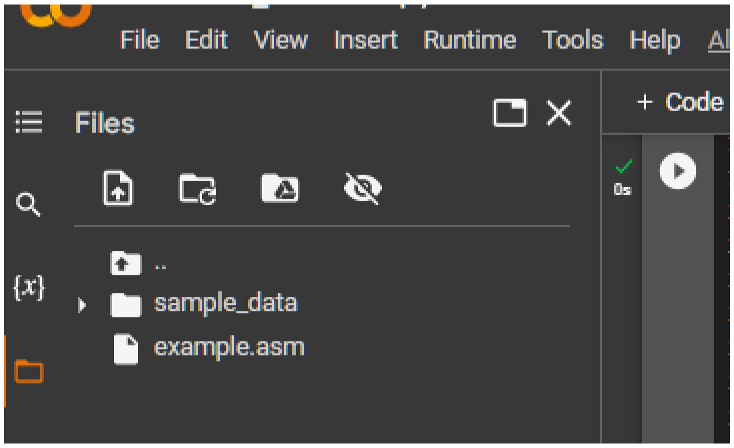
This lab will focus on three scenarios for data hazards that, even with forwarding, would require stall cycles. These situations are as follows:

1. A hazard in which a load (e.g. lw, ld) instruction precedes an arithmetic instruction. Loads occur in stage 4 of the pipeline (MEM), and arithmetic instructions require input to the ALU at the beginning of stage 3. This "dependency backwards in time" means that even if we forward the output of the MEM stage to the beginning of the EX stage, the arithmetic instruction still must wait for one cycle.
2. A hazard in which an arithmetic instruction is immediately followed by a branch instruction. Note that the branch instruction itself represents a control hazard, but if the comparison (e.g. beq, bne, etc.) source operand is read from the destination of a previous instruction, it requires one cycle.
3. A hazard in which a load instruction is immediately followed by a branch instruction. The load occurs in stage 4, but the branch decision is made in stage 2. Hence, this requires two stall cycles.

If possible, other unrelated instructions can be inserted in the required stall cycles such that no time is wasted. Note that this should not impact the functionality of the code.

Lab

1. Open the following link to access the python notebook in Google Collab.
2. Download 'example.asm' from Canvas (Files > Labs > Lab 10) and upload it in the Files pane.



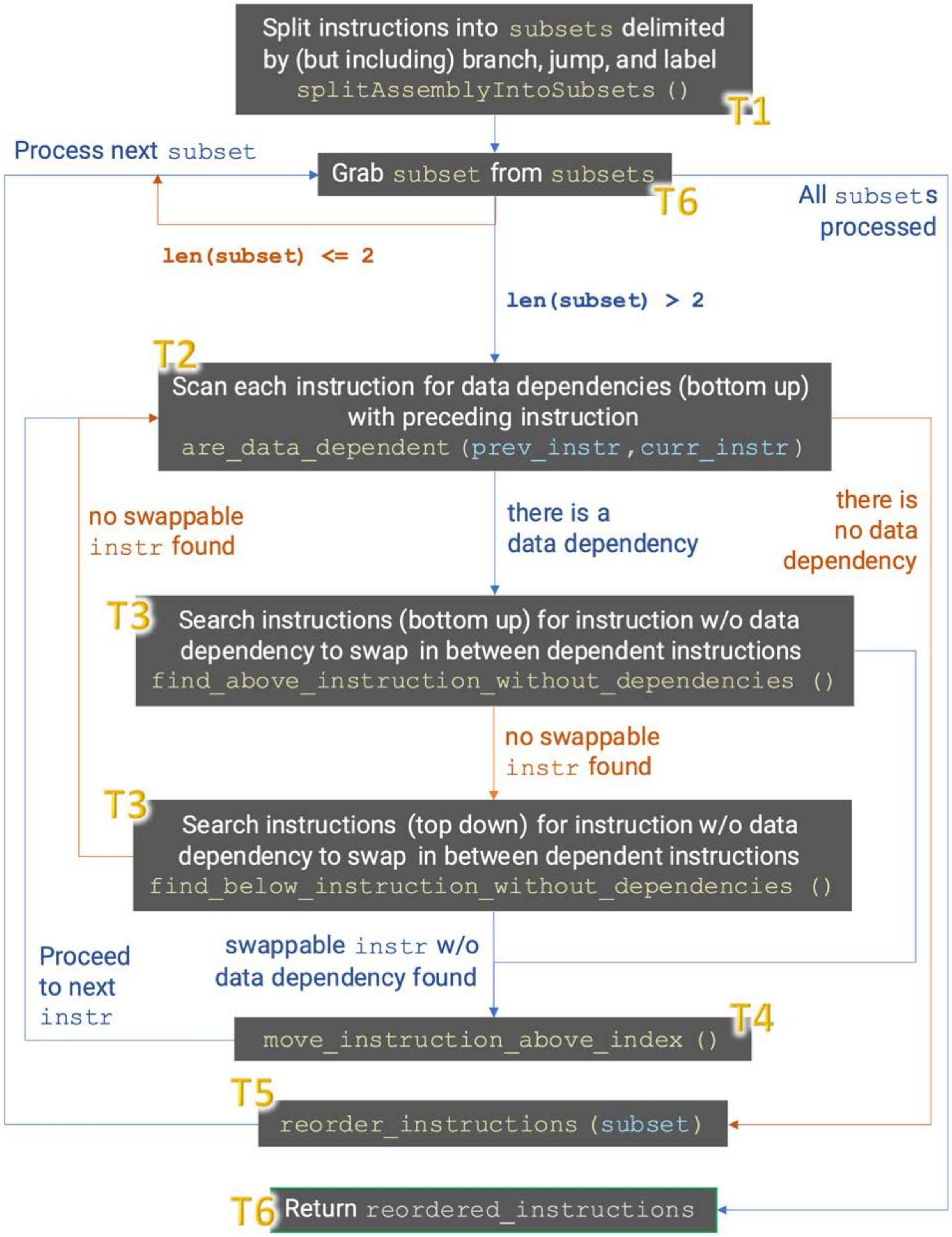
3. Complete Tasks 1-6, as described in the python notebook.
   Note: The last page contains a flow chart displaying a high-level overview of the algorithm of the tasks to be implemented in this lab.

T1-6: Take screenshots of outputs from Tasks 1 - 6 and include those in your report.

T7: What are pipeline hazards? Give one example and provide a sample of assembly code that would cause such a hazard.

T8: How can CPU hardware deal with hazards? How can a compiler help?

In-class work - Complete **T1** and **T2**, show it to the TA for review and submit it on Canvas before the lab session ends. Submit your in-class work as a Word document (.docx) format.
Final Report - Submit your report .pdf with answers to all questions and screenshots. Also, submit the final `.ipynb` file. Submit just one .zip file per group.


Split instructions into subsets delimited by (but including) branch, jump, and label splitAssemblyIntoSubsets ()
T1
Process next subset
Grab subset from subsets
T6
All subsets processed
len(subset) <= 2
len(subset) > 2
T2
Scan each instruction for data dependencies (bottom up) with preceding instruction are_data_dependent (prev_instr,curr_instr)
no swappable instr found
there is a data dependency
there is no data dependency
T3
Search instructions (bottom up) for instruction w/o data dependency to swap in between dependent instructions find_above_instruction_without_dependencies ()
no swappable instr found
T3
Search instructions (top down) for instruction w/o data dependency to swap in between dependent instructions find_below_instruction_without_dependencies ()
Proceed to next instr
swappable instr w/o data dependency found
move_instruction_above_index ()
T4
T5
reorder_instructions (subset)
T6
Return reordered_instructions