EECS 583 – Fall 2023 – Midterm Exam

Friday, November 3, 2023
Exam duration: 1 hr 45 min
Open book, open notes

Name: _____SOLUTION KEY_____

Please sign indicating that you have upheld the Engineering Honor Code at the University of Michigan.

"I have neither given nor received aid on this examination."

Signature: _____

There are 10 questions divided into 2 sections. The point value for each question is specified with that question. Please show your work unless the answer is obvious. If you need more space, use the back side of the exam sheets.

Part I: Short Answer
5 questions, 25 pts total Score:_____

Part II: Long Answer
5 questions, 75 pts total Score:_____

Total (100 possible): _____

Part I. Short Answer (Questions 1-5) (25 pts)

1) A new compiler identifies possibly uninitialized variables by examining the USE-DEF chains for each variable where an empty chain (e.g., no DEFs that reach the use) indicates an uninitialized variable. Would this method identify *all* uninitialized variables? Briefly explain why or why not. (5 pts)

   False. This method will not catch variables that are conditionally initialized, e.g., a variable is initialized on the "then" path, and then consumed after the "if-then-else", so there is no initialization on the else path. This occurs because reaching defs used to calculate USE-DEF chains is any-path (Union meet function).

2) Profile information can be used for a variety of purposes in a compiler. Name a way it can be used to optimize instruction cache performance. Briefly explain. (5 pts)

   Profile data can be used to co-locate hot code blocks, which reduces instruction footprints and increases **Icache performance**. Examples include **trace selection, superblock formation**, etc. Another acceptable answer is to **code expanding optimizations** like loop unrolling and function inlining to only the most important segments of code so as to reduce the size of binary growth and indirectly improve Icache performance.

3) When scheduling a basic block, all instructions must be scheduled at their Estart (earliest start) time to ensure the basic block finishes in the fewest cycles. Is this statement True or False? Briefly explain. (5 pts)

   False. Instructions must be scheduled by their Lstart to ensure the basic block finishes in the fewest possible cycles. The range of [Estart-Lstart] provides a scheduling range for each instruction such that the block finishes in the minimal cycles, thus scheduling at the Estart time is not necessary.

4) Is it possible to unroll the following loop, for (i=0; i<100; i+=X) { … } where X is a value input by the user immediately before the for loop statement? *You may assume there are no breaks or continues in the loop and X is not modified*. Briefly explain. (5 pts)

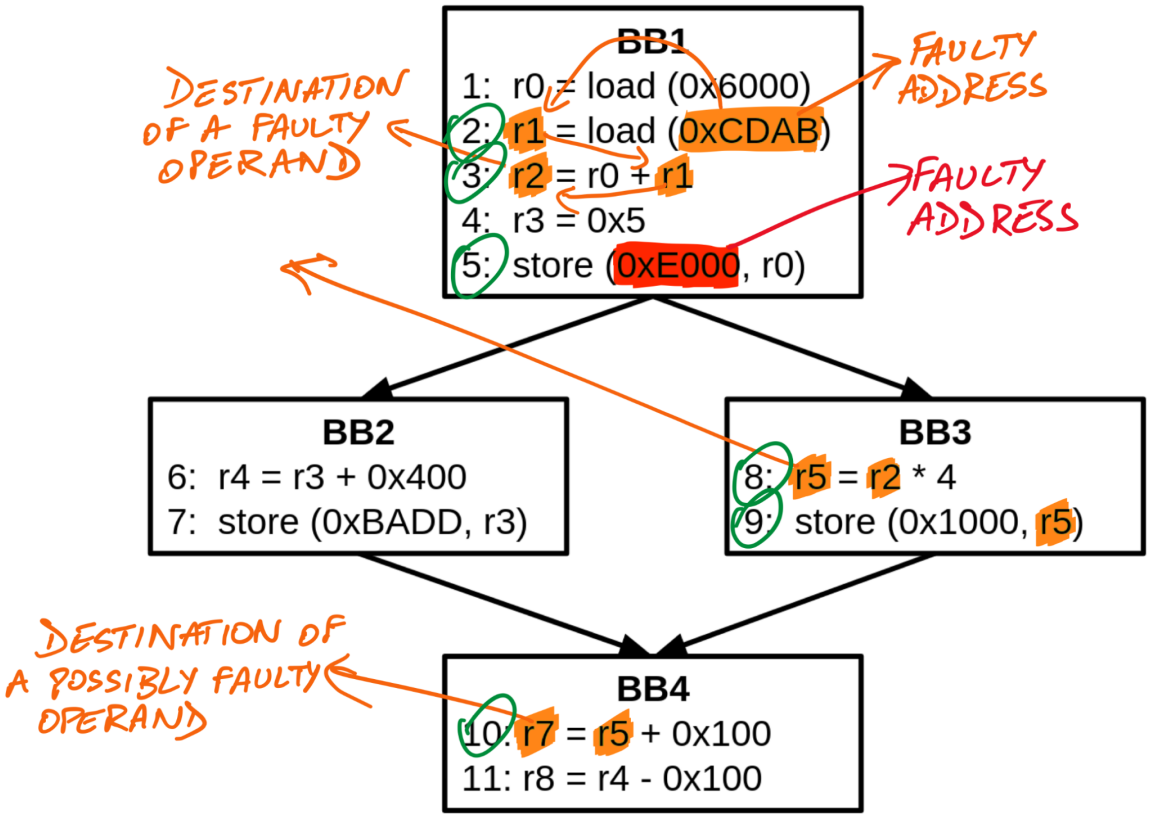
   Yes. This can be done by creating a **Preloop** for the non-multiple of unroll factor iterations. The number of iterations spent in the Preloop and unrolled body will be computed at run time, hence the fact that X is unknown is fine. The number of iterations just needs to be known before the loop is invoked (e.g., the loop is a counted loop). Another acceptable answer is to say the loop is unrolled by keeping the loop back branches in the loop with the reverse condition to conditionally exit the loop. Such unrolling is less effective because fewer branches are eliminated, but still creates a larger loop body.

5) For graph coloring based register allocation, all nodes in the interference graph with degree >= N (the number of registers) will be spilled to achieve a successful allocation. Is this statement True or False? Briefly explain. (5 pts)

False. When a node is selected for spilling, the node and all of its edges are removed from the graph. Hence, the degree of nodes with degree >= N can be reduced by spilling its neighbors which can eventually lead to the node having degree < N which will make it colorable.

Part II. Longer Problems (Questions 6-10) (75 pts)

6) You are designing a dataflow analysis for a processor with unreliable memory. Specifically, data in memory between addresses **0xC000 - 0xEFFF** gets lost. Loads from this address range get garbage values and stores to this address range result in data loss. Such loads and stores are considered as faulty. Any subsequent instructions which use the faulty loaded values (arithmetic, memory, or control) are also deemed faulty. Your dataflow analysis needs to identify such faulty instructions. (15 points)
Note: store (A, B) implies that the value B is being stored at address A in memory. Assume that all registers are initialized properly before BB1 outside the faulty range.



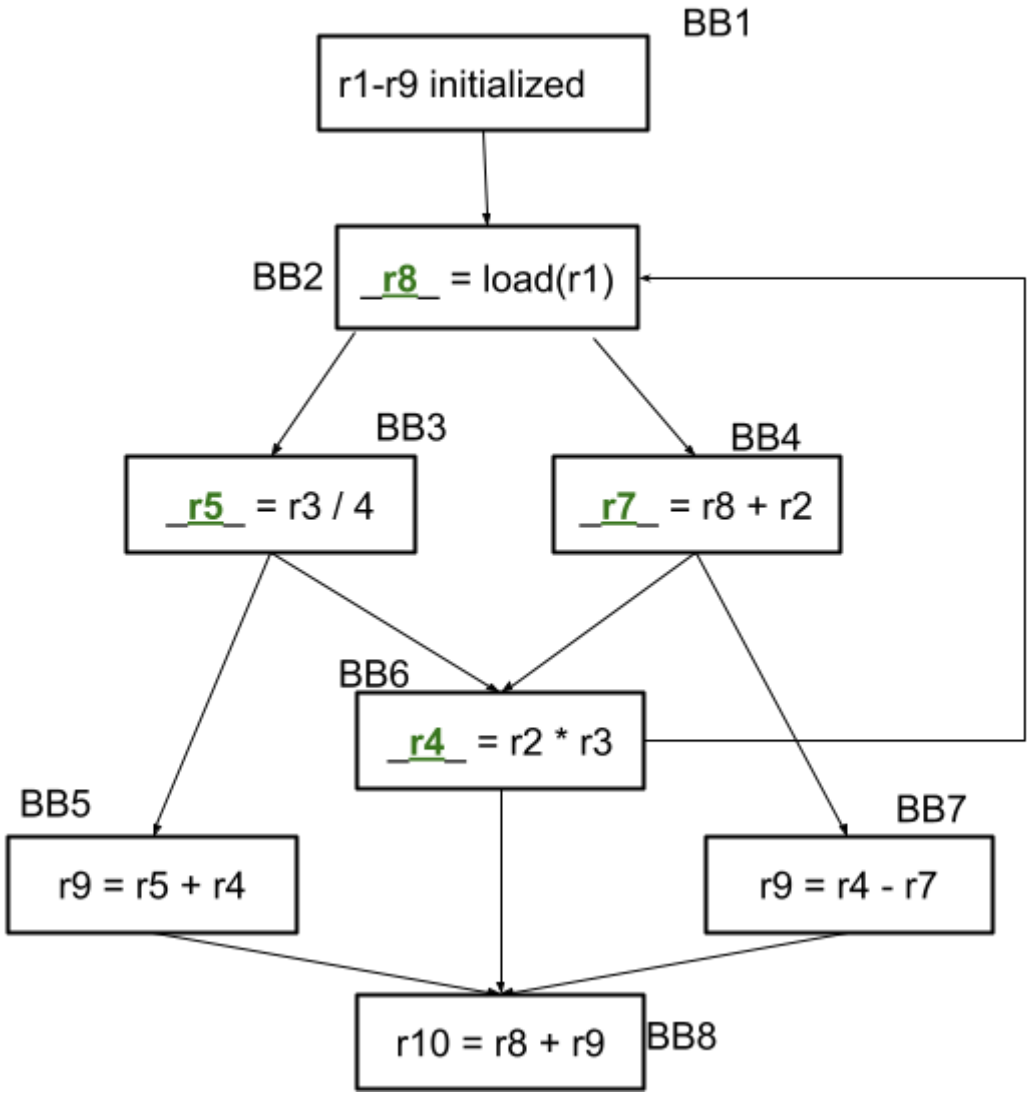
a) Is this a top-down or bottom-up dataflow analysis problem? ____**top-down**_____

b) Is this an all-path or any-path dataflow analysis problem? ____**any-path**_____

c) Write the GEN and KILL sets for each basic block (just specify the contents of each set for each BB, no need to define the algorithm).

	BB1	BB2	BB3	BB4
GEN	**2, 3, 5**	**-**	**8, 9**	**10**
KILL	**1, 4**	**6, 7**	**-**	**11**

7) Fill in the blanks using r4, r5, r7, and r8, so that a maximum number of instructions from BB2, BB3, BB4, and BB6 become eligible for hoisting via LICM. (10 pts)



r4 and r8 are liveout on all edges out of the loop (BB5, BB7, BB8) and hence can only be placed in BB2 to be hoisted correctly. Hence, only one of the two can be hoisted successfully. Observe that by placing r8 in BB2 and allowing it to be hoisted, we also make the instruction in BB4 invariant and able to be hoisted. Hence r8 is preferred over r4, for placement in BB2.
r5 is liveout on BB5 and hence can only be placed in BB2 or BB3, but since we want r8 to be in BB2, we place it in BB3.
r7 is liveout on BB7 and hence can only be placed in BB2 or BB4, but since we want r8 to be in BB2, we place it in BB4.
r4 has to be placed in BB6 and will not be hoisted by the LICM optimization. r8, r5, and r7 will be hoisted.

8) For the given code, (15 points)

a) Compute the number of predicates required to if-convert the code.

b) To profile the code, we ran this function **100** times, we found that:

i) The loop back-edge was never taken.

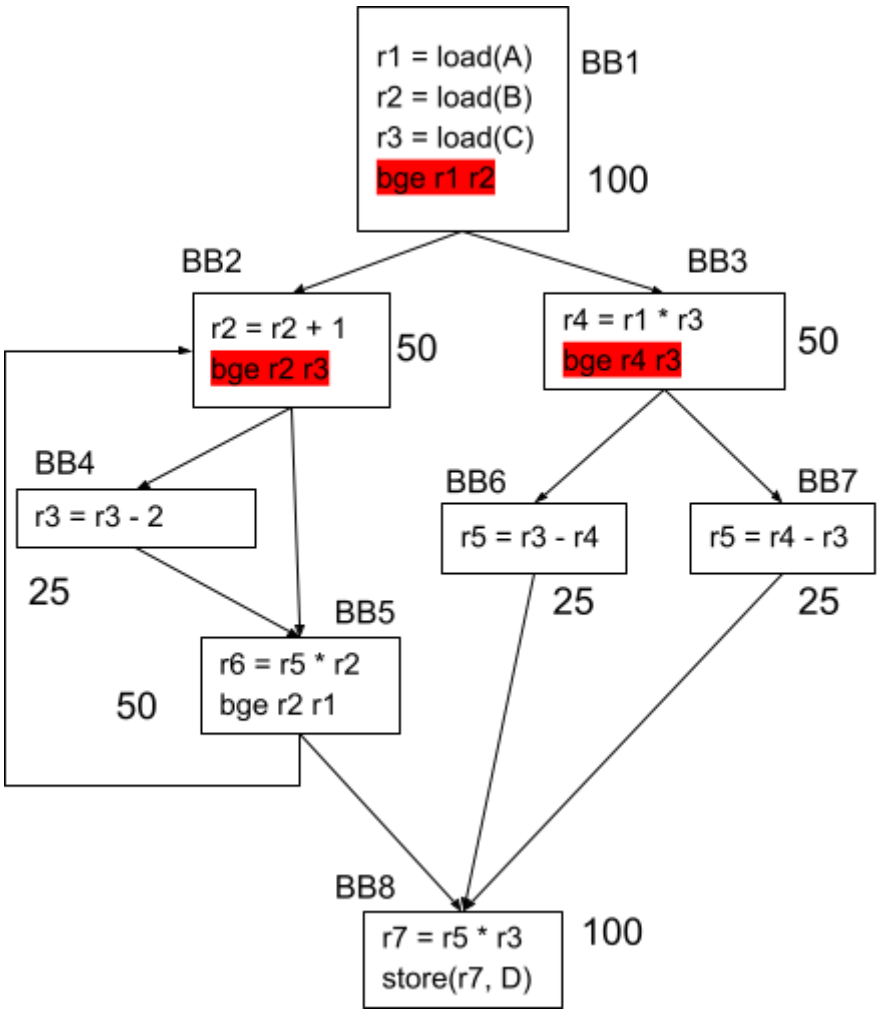
ii) All other conditional branches were taken exactly 50% of the time.

iii) All branch probabilities were independent.

iv) Each instruction takes 1 unit of time to execute, except branches which take 3 units each.

Based on this information, will if-conversion of all eligible branches to utilize predicated execution (no other optimizations) make the code run faster on the same profile test cases? Justify your answer.

You may assume that there are no mispredictions, and that a CMPP instruction can compute up to 2 predicates for every condition.



(BB1 and BB8 are always executed. BB2 and BB5 will have the same control dependence set. All others are unique. Remember that back-edges are nuked when computing these)

Time taken for normal code = #non-branch + 3*#branch =(300+50+50+25+50+25+25 +200) + 3(100+50+50+50) =1475 units of time

Time taken for if-pred code 3 branches (except loop) get replaced by CMPP. = 1400 + 300 = 1700 units of time.

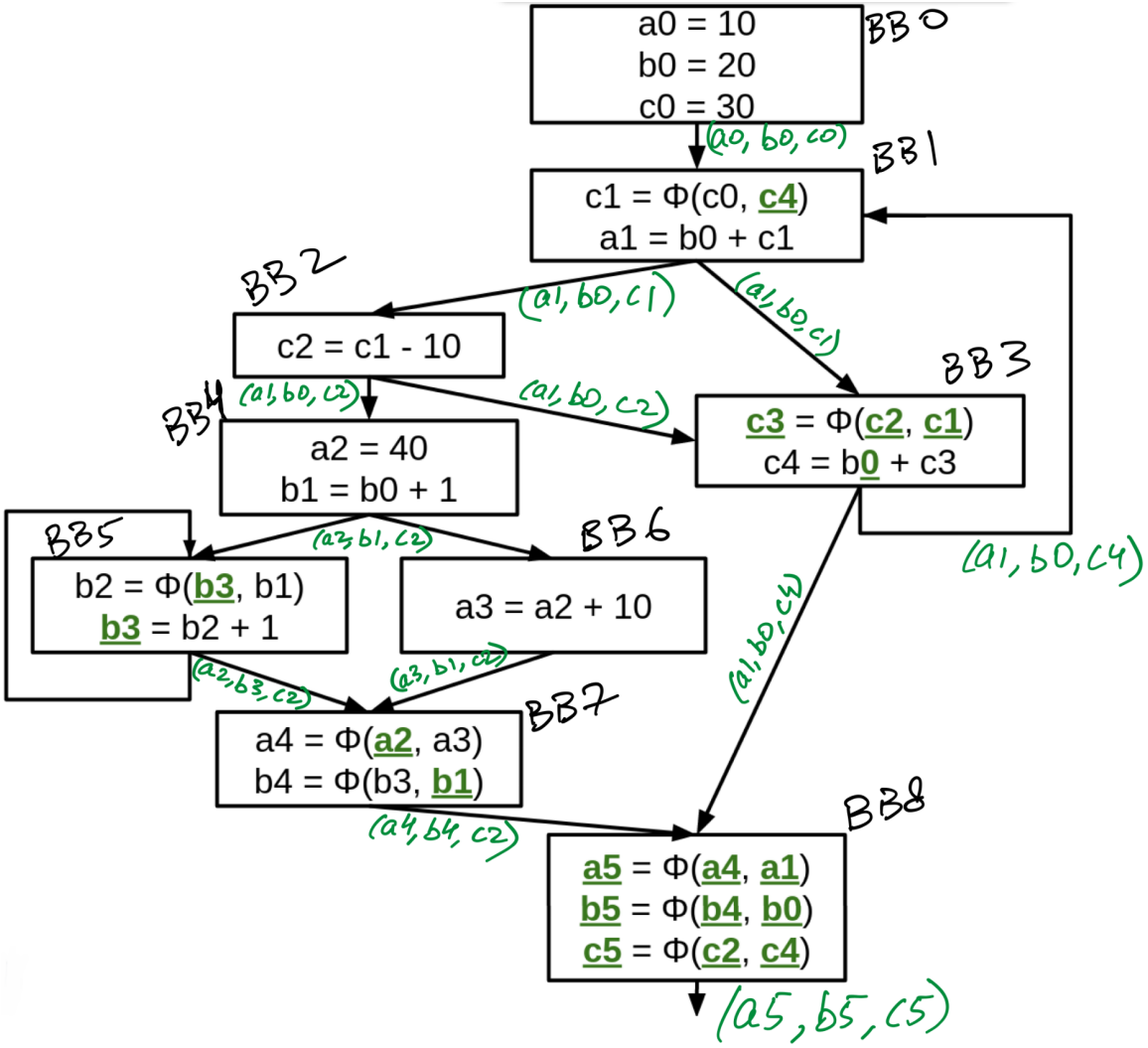
Remember that code correctness still needs to be maintained, so the loop backedge branch has to be reinserted, and will take 3 cycles.

(a) Number of predicates required = 5

(b) If-predicated code will take longer (See calculations above) Partial credit will be given.

9) Satisfy static single assignment (SSA) form by filling in the blanks in the code segment below. Solving by inspection is fine. (15 points)

- The result and arguments of a phi node must be different instances of the same variable (e.g. a1 = Φ(a2, a3)). Further, all variables with the same letter referred to a single variable in the original program.
- Choose operands between a0 - a5, b0 - b5, c0 - c5. Repetition is allowed.
- There are no unnecessary phi nodes.

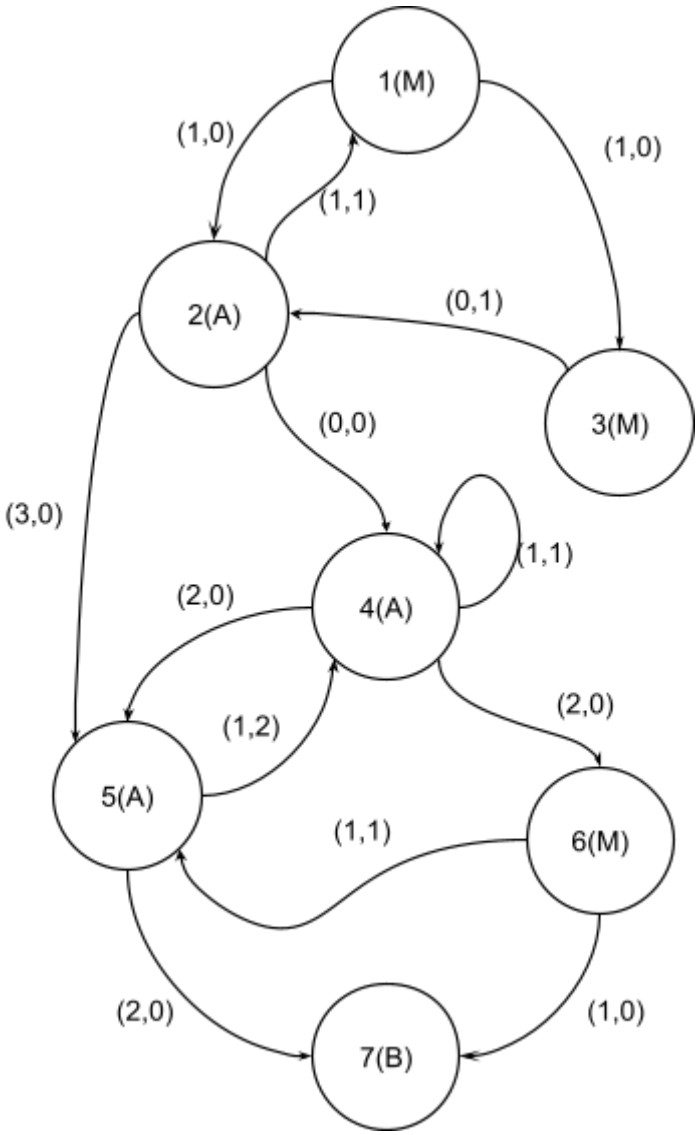


10) Given below is a loop dependence graph and a processor model. (M), (A), and (B) refer to memory, ALU, and branch instructions respectively. The memory instructions use the memory units and the ALU and branch instructions use the ALU units. (20 points)

a. Determine the MII. Show your work. (5 points)

b. Generate the rolled and unrolled schedules using this MII. Lower instruction numbers will have a higher priority, i.e. instruction 1 has the highest priority. (15 points)

Processor Model
4 fully pipelined function units
– 2 ALU units
– 2 MEM units



Unrolled Schedule (may contain extra rows)

	ALU1	ALU2	MEM1	MEM2
0			1	
1		2	3	
2	4			
3				
4		5		6
5				
6				
7	7			
8				

Rolled Schedule (may contain extra rows)

	ALU1	ALU2	MEM1	MEM2
0	4	5	1	6
1	7	2	3	
2				
3				

MII = 2

Cycles: 1→2→1, 1→3→2→1, 4→ 4, 4→5→4, 4→ 6→5→4
ResMII = MAX(4/2, 3/2) = 2; RecMII = MAX(2/1,2/2, 1/1, 3/2, 4/3) = 2

Scheduling Process:

1. **Schedule branch instruction (7) in Rolled schedule first. Reserving Rolled(1, ALU1).**
2. **Since instructions are numbered in priority order, we schedule them one by one, keeping track of which instructions have already been scheduled in the Rolled and Unrolled tables:**
    a. **Instruction 1: Memory instruction. Has no scheduled predecessors. So we schedule in cycle 0. Reserve Rolled(0, MEM1), Schedule Unrolled(0, MEM1).**
    b. **Instruction 2: ALU instruction. Needs to wait for 1 cycle after Instruction 1. So we can schedule in cycle 1. Cannot reserve Rolled(1, ALU1) because Instruction 7 has done so. So instead we Reserve Rolled(1, ALU2) and Schedule Unrolled(1, ALU2).**
    c. **Instruction 3: Memory instruction. Needs to wait 1 cycle after Instruction 1. So we can schedule in cycle 1. Reserve Rolled(1, MEM2) and Schedule Unrolled(1, MEM2).**
    d. **Instruction 4: ALU instruction. Can be scheduled at the same time as Instruction 2 (i.e. Cycle 1). But can't reserve either ALU at that cycle because ALU1 is reserved by Instruction 7 and ALU2 is reserved by Instruction 2. So we push it to Cycle 2. Reserve Rolled(0, ALU1) and Schedule Unrolled(2, ALU1).**
    e. **Instruction 5: ALU instruction. Needs to be scheduled 3 cycles after Instruction 2 and 2 cycles after Instruction 4, so the earliest it can be scheduled is cycle 4. Rolled(0, ALU1) is reserved by Instruction 4, so we Reserve Rolled(0, ALU2) and Schedule Unrolled(4, ALU2).**
    f. **Instruction 6: Memory instruction. Needs to be scheduled at least 2 cycles after instruction 4, i.e. Cycle 4. Rolled(0, MEM1) is reserved by Instruction 1, so we Reserve Rolled(0, MEM2) and Schedule Unrolled(4, MEM2).**
    g. **Instruction 7: Branch instruction. Needs to be scheduled at least 2 cycles after Instruction 5 and 1 cycle after Instruction 6, so it cannot be scheduled before cycle 6. However, in step 1, we have already reserved Rolled(1, ALU1), so we need to push it to cycle 7 instead. So Schedule Unrolled(7, ALU1)**

Switching around MEM0 and MEM1 or ALU0 and ALU1 is okay as long as the rolled and unrolled schedules are consistent.

Common mistakes during scheduling:

1. **Rolled and unrolled schedules are inconsistent.**
2. **Forgetting to reserve a slot for branch instruction in the rolled schedule before beginning scheduling.**
3. **Scheduling 6 at cycle 3 because calculated estart is 3. Remember that estart is just the earliest possible time to schedule an instruction. Since a previous instruction (i.e. Instruction 4) got pushed to the next cycle, every instruction that depended on it could also be delayed.**