EECS 583 – Winter 2024 – Midterm Exam

Wednesday, Mar 20, 2024
Exam duration: 1 hr 20 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 pts): _____

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

1) Is branch profile information useful to consider when performing LICM? Answer Yes or No and **briefly explain**. (5 pts)

Yes.
Reason1: Not all BBs in a loop are high frequency blocks, and moving an instruction from a BB that is executed less often than the preheader often results in performance loss.
Reason2: LICM may also increase register pressure resulting in more register spills, thus moving low frequency instructions to the preheader may result in a net performance loss.
Note: it is LICM not FPLICM, answering frequent path/speculative execution (with fixup code) benefit will result in -2pts

2) Which two phrases below describe Available Expressions analysis? Circle the **2 best** answers below. (5 pts)

Bottom Up

>>>Top Down

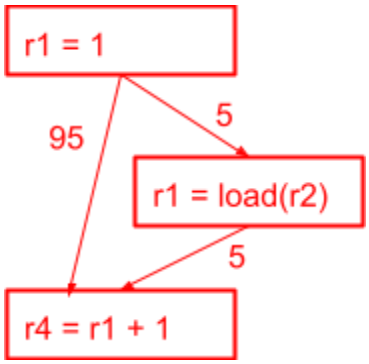
Any Path

>>>All Path

3) Suppose basic block X dominates basic block Y in function run(). When run() is executed, is it ever possible to visit X *after* visiting Y? Answer Yes or No, and **briefly explain**. (5 pts)

Yes, by traversing a loop backedge. A loop backedge is a branch from a BB, say Y, to another BB, X, where BBX dominates BBY

4) Using Homework 2 as inspiration, would frequent-path constant propagation provide any advantages over traditional constant propagation? **Briefly explain with a small code example.** (5 pts)



Frequent-path constant propagation would enable more constant propagations to occur when a register holding a constant is conditionally overwritten on an infrequent path like in the figure. By ignoring the infrequent path, r1=1 could be propagated to r4=r1+1. Fixup code must be inserted to maintain correctness. Fixup would consist of placing a copy of r4=r1+1 after the load instruction and adjusting the control flow out of the basic block containing the load.
If your example does not have a use in the end, it is marked as incomplete.

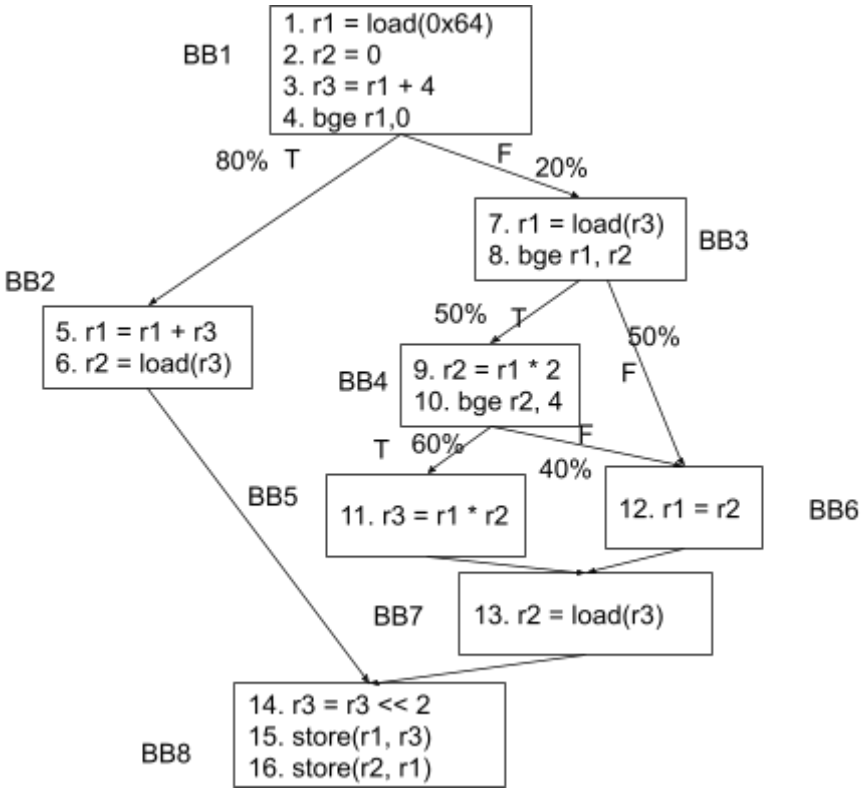
5) Name **one benefit and one cost** of compile time loop unrolling? (5 pts)

Benefit: Remove loop back branches or create larger loop body to facilitate instruction overlap during scheduling.

Cost: Code size increase or increased register pressure which may result in more register spills.

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

6) For the following control flow graph (CFG): (15 pts)



a. Compute the minimum number of predicates needed to if-convert the code. (5 pts)

Minimum Predicates Needed = _____5_____

Basic Block	CD Set
BB1	{}
BB2	{+1}
BB3	{-1}
BB4	{+3}
BB5	{+4}
BB6	{-3,-4}
BB7	{-1}
BB8	{}

b. We profiled the code and found that BB1 was executed **100** times and recorded the edge probabilities of conditional branches ('bge' instruction) and labeled the edges with 'T' for true (branch taken) and 'F' for false (branch not taken). In modern

CPUs, a branch predictor is a unit that will decide which path to follow when encountering a conditional branch. Assuming each non-branch instruction (including CMPP) takes **1** unit of time to execute. Each branch ("bge") instruction takes **1** unit of time when the branch predictor is correct, and takes **12** units of time if it predicts incorrectly. Assume CMPP instructions can only compute 1 predicate at a time (CMPPs do not have 2 destinations). Ignore instructions for predicate initialization and unconditional jumps that are not shown in the CFG.

If our branch predictor **always predicts not taken ('F')**, would if-converting the entire CFG offer performance benefits? Justify your answer **numerically** (10 pts)

It is beneficial to if-convert the code. (2 pts)

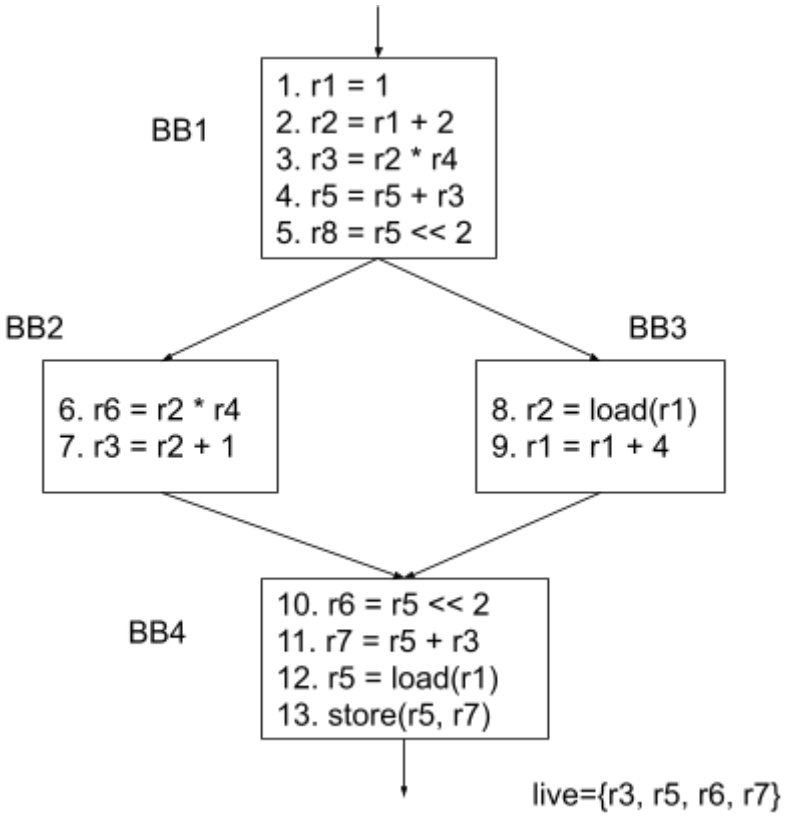
Original code:

non-bge instructions: 300+160+20+10+6+14+20+300 = 830
bge instructions: 12 * (80 +10 + 6) + 1 * (20 + 10 + 4) = 1186
in total: 830 + 1186 = 2016 (4 pts)

BB1-8 merge into a single BB with 13 normal instructions and 6 CMPP instructions, time: 19 * 100 = 1900.
So it is beneficial to if-convert because 1900 < 2016 (4 pts).

1. r1 = load(0x64) if T
2. r2 = 0 if T
3. r3 = r1 + 4 if T
4. **p2 = CMPP.GE.UN(r1,0) if T**
5. **p3 = CMPP.GE.UC(r1,0) if T**
6. r1 = r1 + 3 if p2
7. r2 = load(r3) if p2
8. r1 = load(r3) if p3
9. **p4 = CMPP.GE.UN(r1,r2) if p3**
10. **p6 = CMPP.GE.UC(r1,r2) if p3**
11. r2 = r1 * 2 if b4
12. **p5 = CMPP.GE.UN(r2,4) if p4**
13. **p6 = CMPP.GE.OC(r2,4) if p4**
14. r3 = r1 * r2 if p5
15. r1 = r2 if p6
16. r2 = load(r3) if p3
17. r3 = r3 << 2 if T
18. store(r1,r3) if T
19. store(r2,r1) if T

7) Consider optimizing the code segment below. Assume all registers are initialized before entering BB1 and the liveness after exiting BB4 is listed below BB4.



a. Which instructions can be optimized by Common Subexpression Elimination (CSE)? List the instruction numbers. Do not apply other optimizations (12 pts)

3, 6; 5, 10; (or just write 6 and 10)
Note that 9, 12 or 4, 11 could not be optimized by CSE. Because r1 value is updated in I9, and r3 value is updated in I7.

b. Name one more optimization that can be applied beyond the CSE done in part (a). (3 pts)

Constant Propagation, dead code elimination, copy propagation

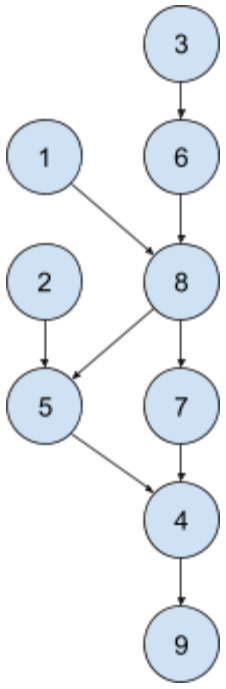
8) There are 9 instructions in a basic block (BB) for which we calculated the Estart and Lstart values using the instruction latencies specified below. However, due to a glitch, the original instruction order got lost and now we want to piece it back together.

a. We remember that the **store instruction** was the last instruction of the BB and has the largest Estart and Lstart values that are specified in the table.
b. Assume all operands used as addresses are valid, but all registers must be defined before being used to ensure correctness (e.g., no registers are live into the BB).
c. 6 of the 9 instructions have slack 0, while the others have slack > 0.

Determine the missing Estart/Lstart values in the table below for the original ordering. Remember, the numbers in the table do not represent the original order. Hint: Try finding the critical path and drawing the dependence graph. (15 pts)

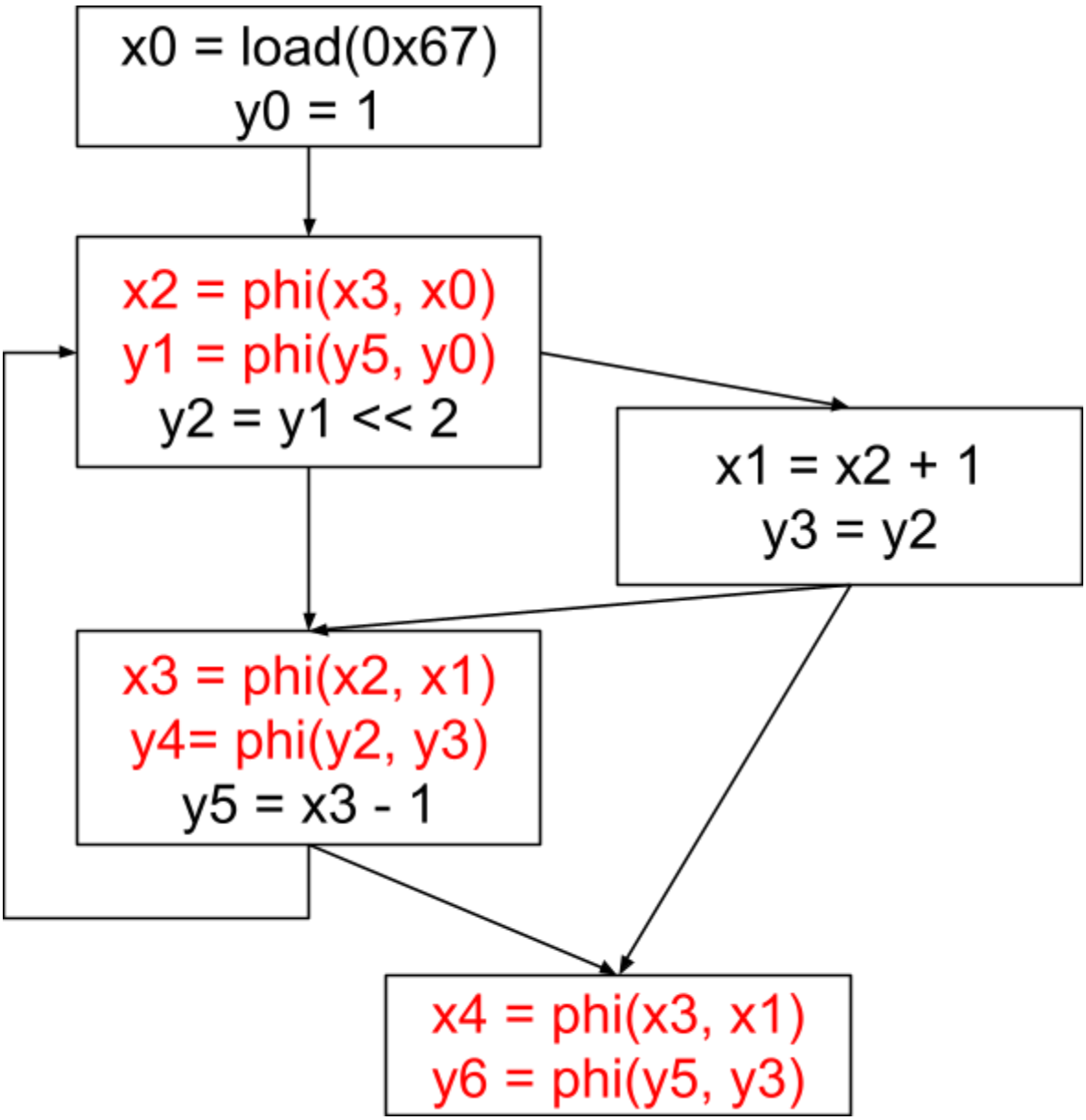
Instruction Latencies

Load/Store: 2
Add: 1
Multiply: 3



No.	Instruction	EStart	LStart
1	r2 = load(0xDF2333)	**0**	**1**
2	r6 = load(0xBF4363)	**0**	**6**
3	r3 = load(0xCF3537)	**0**	**0**
4	r9 = r2 + r3	**9**	**9**
5	r2 = r5 + r6	**6**	**8**
6	r4 = r3 + 6	**2**	**2**
7	r3 = r5 * 3	**6**	**6**
8	r5 = r2 * r4	**3**	**3**
9	store(0xA23455, r9)	**10**	**10**

9) Fill in the blanks below to put the code into the static single assignment (SSA) form. Leave phi nodes blank when they are unnecessary. Choose operands sequentially from the lists x0, x1, … xn, and y0, y1, … ym, using each operand only after all its preceding operands in the sequence have been used. Repetition of registers used is permitted. For phi nodes, place the operand in the **first** position if it originates from the **left** edge, and in the **second** position if it originates from the **right** edge. (15 pts)



Note: y4 = phi(y2,y3) is in fact superfluous. Hence if you omitted only that, you still get full points.

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 unit and the ALU and branch instructions use the ALU units. (15 points)

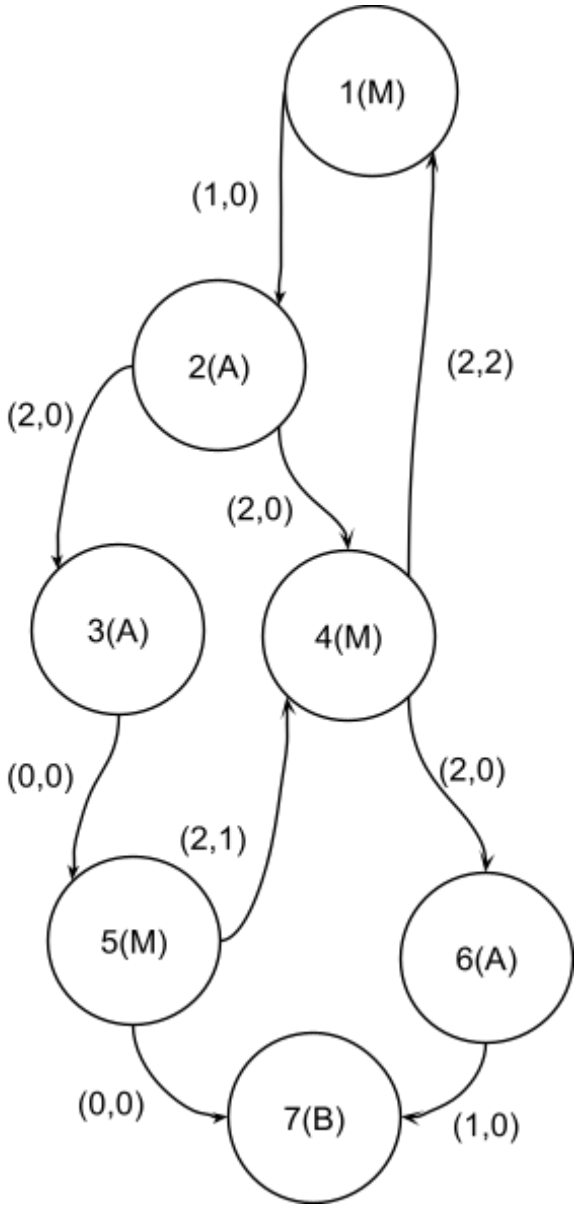
a. Determine the MII. **Show your work and fill out the following:** (5 pts)

Cycles : 1→2→4→1, 1→2→3→5→4→1 ResMII = ceil(max(4/2, 3/1)) = 3

RecMII = ceil(max(4/2, 7/3))=3 MII = max(3,3) = 3

b. Generate the rolled and unrolled schedules using this MII. Assume **lower instruction numbers have a higher priority, i.e. instruction 1 has the highest priority**. (10 pts)

Processor Model
3 fully pipelined functional units
– 2 ALUs
– 1 MEM unit



Unrolled Schedule (may contain extra rows)

	ALU0	ALU1	MEM
0			1
1	2		
2			
3	3		
4			4
5			5
6		6	
7			
8	7		

Rolled Schedule (may contain extra rows)

	ALU0	ALU1	MEM
0	3	6	1
1	2		4
2	7		5
3			

Since we are already given the priorities for the instructions, we just need to schedule them in order as per the algorithm. Remember that we place the branch instruction (7) in the last row of the rolled schedule first, thus reserving its slot there.

R(2,ALU0) ← 7

Now we start with instruction 1, starting at time t=0. Since we have free spots in both schedules in cycle 0, we write:

R(0,MEM) ← 1 and U(0,MEM) ← 1

Next we look at instruction 2. Since it has a dependency of 1 cycle on Instruction 1, the earliest it can be scheduled is cycle 1, which has free spots in both ALU positions in both tables:

R(1, ALU0) ← 2 and U(1, ALU0) ← 2

Next we look at instruction 3. Since it has a dependency of 2 cycles on Instruction 2, the earliest it can be scheduled is cycle 3 (%3 = 0), which has free spots in both ALU positions in both tables:

R(0, ALU0) ← 3 and U(3, ALU0) ← 3

Next we look at instruction 4. Since it has a dependency of 2 cycles on Instruction 2, the earliest it can be scheduled is cycle 3 (%3 = 0), which does not have a free spot in the rolled schedule as (0,MEM) is being used by Instruction 1. Hence, it gets pushed to the next cycle (4%3 = 1) in both schedules:

R(1, MEM) ← 4 and U(4, ALU0) ← 4

Next we look at instruction 5. Since it has a dependency of 0 cycles on Instruction 3, the earliest it can be scheduled is cycle 3 (%3 = 0), which does not have a free spot in the rolled schedule as (0,MEM) is being used by Instruction 1. Hence, it gets pushed to the next cycle (4%3 = 1) in both schedules, but that (1,MEM) is being used by Instruction 4. Hence, it gets pushed to the next cycle (5%3 = 2) in both schedules:

R(2,MEM) ← 5 and U(5, MEM) ← 5

Next we look at instruction 6. Since it has a dependency of 2 cycles on instruction 4, the earliest it can be scheduled is cycle 6 (%3 = 0), which has one free spot in the rolled schedule. Hence, we have:

R(0, ALU1) ← 6 and U(6, ALU1) ← 6

Finally, we need to place instruction 7 in the unrolled schedule. It has a 1 cycle dependence on Instruction 6, hence can be scheduled earliest at cycle 7 (%3 = 1). However, since it is the branch instruction, it needs to correspond to the rolled schedule and must be placed in the next cycle where %3 = 2, which in this case is cycle 8. Hence, we have:

U(8, ALU0)

And that completes the schedule. Note that the choice of ALU0 and ALU1 is arbitrary, so swapping those columns is okay, as long as you are consistent in both schedules.