EECS 583 – Fall 2020 – Midterm Exam

Open: Monday, November 2, 2020 10:30AM EST
Close: Monday, November 3, 2020 10:30AM EST
Time constraint: 3 hours
Open book, open notes

Name: _____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 12 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, just use additional sheets of paper. Just be sure to clearly mark additional sheets with the problem number.

Part I: Short Answer
      6 questions, 30 pts total   Score:_____

Part II: Medium Problems
      6 questions, 70 pts total   Score:_____

Total (100 possible): _____

Part I. Short Answer (Questions 1-6) (30 pts)

1) Profiling offers many advantages including identification of hot code or frequent behaviors in an application. But, it also suffers from some potential *pitfalls*, name one of those pitfalls. (5 pts).

If profile information cannot represent the actual inputs, it may mislead optimization decisions and lead to performance loss. Also, since profiling overhead can be costly, the benefit from the optimization should be larger than profiling overhead. (e.g., dynamic compilation)

2) It is often possible to improve the performance of a loop limited by *RecMII* by adding resources to the processor. Is the preceding statement True or False? Justify your answer. (5 pts)

False. Dependence cycles that cause recurrences cannot be resolved by additional resources. The only way to reduce RecMII is to transform the code to break one or more dependences that are on a critical recurrence path.

3) Suppose that you have defined the GEN, KILL, IN, and OUT sets for a *top-down any-path* dataflow analysis. What is the primary change required to convert this analysis to a *top-down all-path*? (5 pts)

Need to change the meet function from union to intersection. So, change the any-path: IN = union(OUT from predecessors) to all-path:IN = intersect(OUT from predecessors).

4) Name one performance advantage of loop unrolling. (5 pts)

There are multiple answers to this question including reduce the number of branches, increase instruction level parallelism to achieve more compact schedules, and relax loop-carried dependencies which opens up new opportunities to latter optimizations.

5) Can an instruction with a smaller Estart be scheduled after an instruction with a larger Estart? Justify your answer. (5 pts)

Yes, as long the larger Estart instruction is not a dependence successor of the smaller Estart instruction. Instructions get scheduled later than their Estart times due to resource constraints. An instruction with a smaller Estart time can be delayed due to resource constraints while on a separate dependence chain, the larger Estart instruction gets scheduled earlier (but equal or after its own Estart time) due to resource availability.
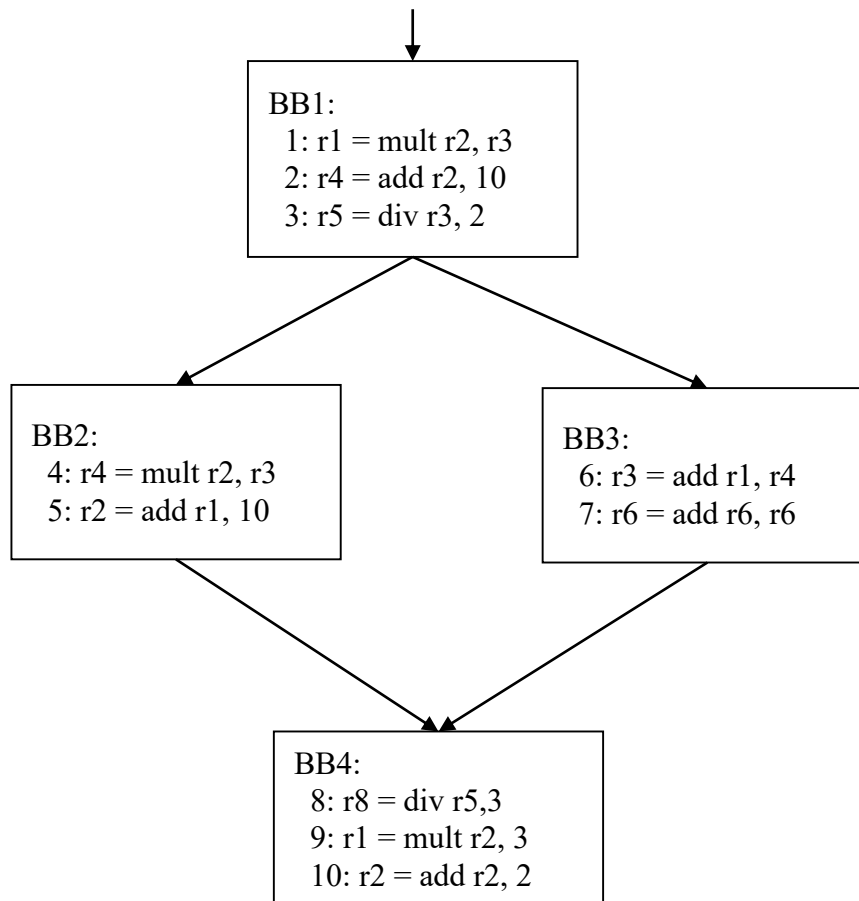
6) A backedge is defined as a control flow edge where the target basic block dominates the source. Can backedge be equivalently defined as a control flow edge where the source basic block post dominates the target? Briefly explain your answer. (5 pts)

No, because the post dominance test does not guarantee a single entry loop. In other words, it identifies a cycle, but the resultant loop may have multiple entrances, hence the loop cannot be optimized using traditional optimizations like LICM and loop unrolling. Thus, it is not considered a "loop". As an example, see the figure below where x post dominates y, but y does not dominates x (i.e. there is a path from z to x), hence the loop has 2 entrances.
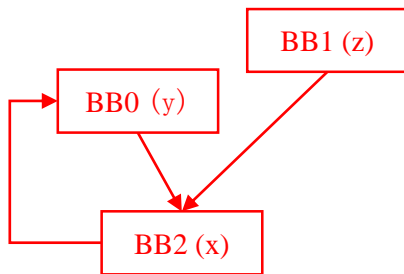
Part II. Medium Problems (Questions 7-12) (70 pts)

7) Compute the Available Expression GEN/KILL/IN/OUT sets at BB4. Assume r2, r3, r6 are properly initialized before entering BB1. (10 pts)

```
BB1:
  1: r1 = mult r2, r3
  2: r4 = add r2, 10
  3: r5 = div r3, 2
```

```
BB2:
  4: r4 = mult r2, r3
  5: r2 = add r1, 10
```

```
BB3:
  6: r3 = add r1, r4
  7: r6 = add r6, r6
```

```
BB4:
  8: r8 = div r5,3
  9: r1 = mult r2, 3
  10: r2 = add r2, 2
```
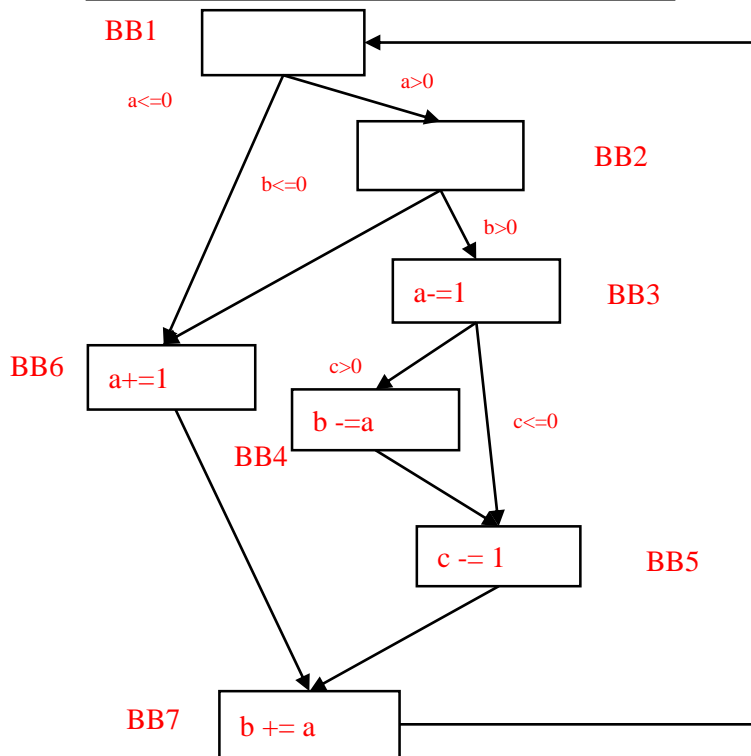
GEN(BB4) = {8}
KILL(BB4) = {1,2,4,5,6,9,10}

IN(BB4) = { }
OUT(BB4) = {8}

8) Draw the control flow graph (CFG) and determine the *minimum* number of predicates required to if-convert the following code. Justify your answers. (10 pts)

```
do{
    if (a>0 && b>0){
        a -= 1;
        if(c>0){
            b -= a;
        }
        c -= 1;
    }else{
        a += 1
    }
    b += a;
}while(b>10);
```

BB1

a<=0   a>0

BB2

b<=0

b>0

BB3   a-=1

BB6   a+=1

c>0

BB4   b -=a

c<=0

BB5   c -= 1

BB7   b += a

BB	Pdom	ipdom	CD
1	1,7	7	-
2	2,7	7	+1
3	3,5,7	5	+2
4	4,5,7	5	-3
5	5,7	7	+2
6	6,7	7	-1, -2
7	7	7	-

4 unique CD sets, thus 4 predicates are needed

9) Satisfy static single assignment (SSA) form by **filling in the blanks with variables from the below pool**. The result and arguments of a Phi node must be all the same variable (i.e., a1 = Phi(a0, a4)). Solving by inspection is fine. (15 pts)

Variable pool: a0, a2, a4, a5, a8; b3, b4, b5, b6, b7 (*All variables can only be used once*)

BB0
```
a0 = 1
b0 = 2
```

BB1
```
a1 = Phi(a0, a8)
b1 = Phi(b0, b7)
a2 = a1 + b1
```

BB2
```
a3 = Phi(a2 a6)
b2 = Phi(b1, b4)
b3 = a3 * b2
```

BB3
```
a4= a3 + 1
```

BB4
```
a5 = a3+ 2
```

BB6
```
b5 = b1 + 2
```

BB5
```
a6 = Phi(a4, a5)
b4 = b3 + 1
```

BB7
```
a7 = Phi(a6, a2)
b6= Phi(b4, b5)
a8= a7+ 1
b7 = a8 + b6
```
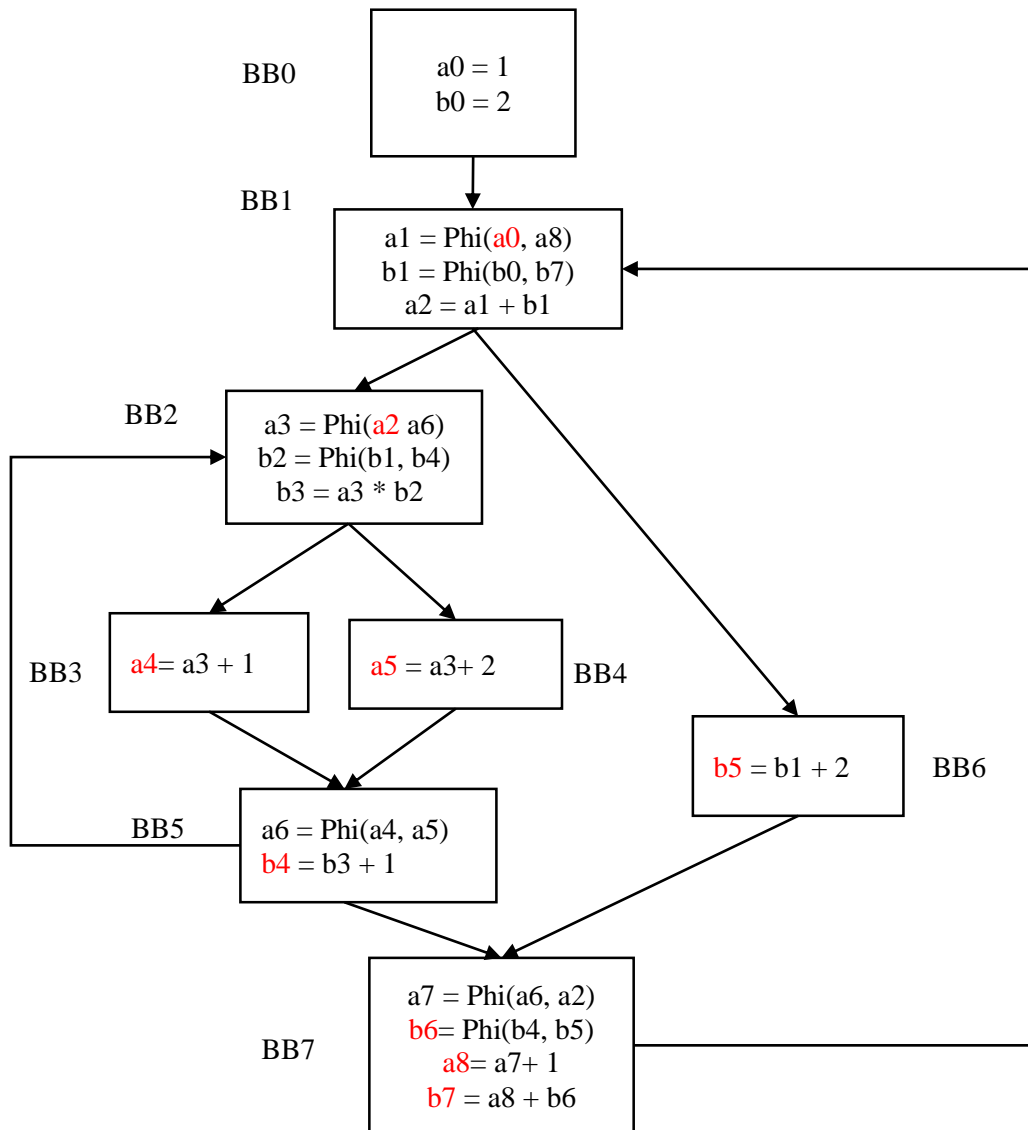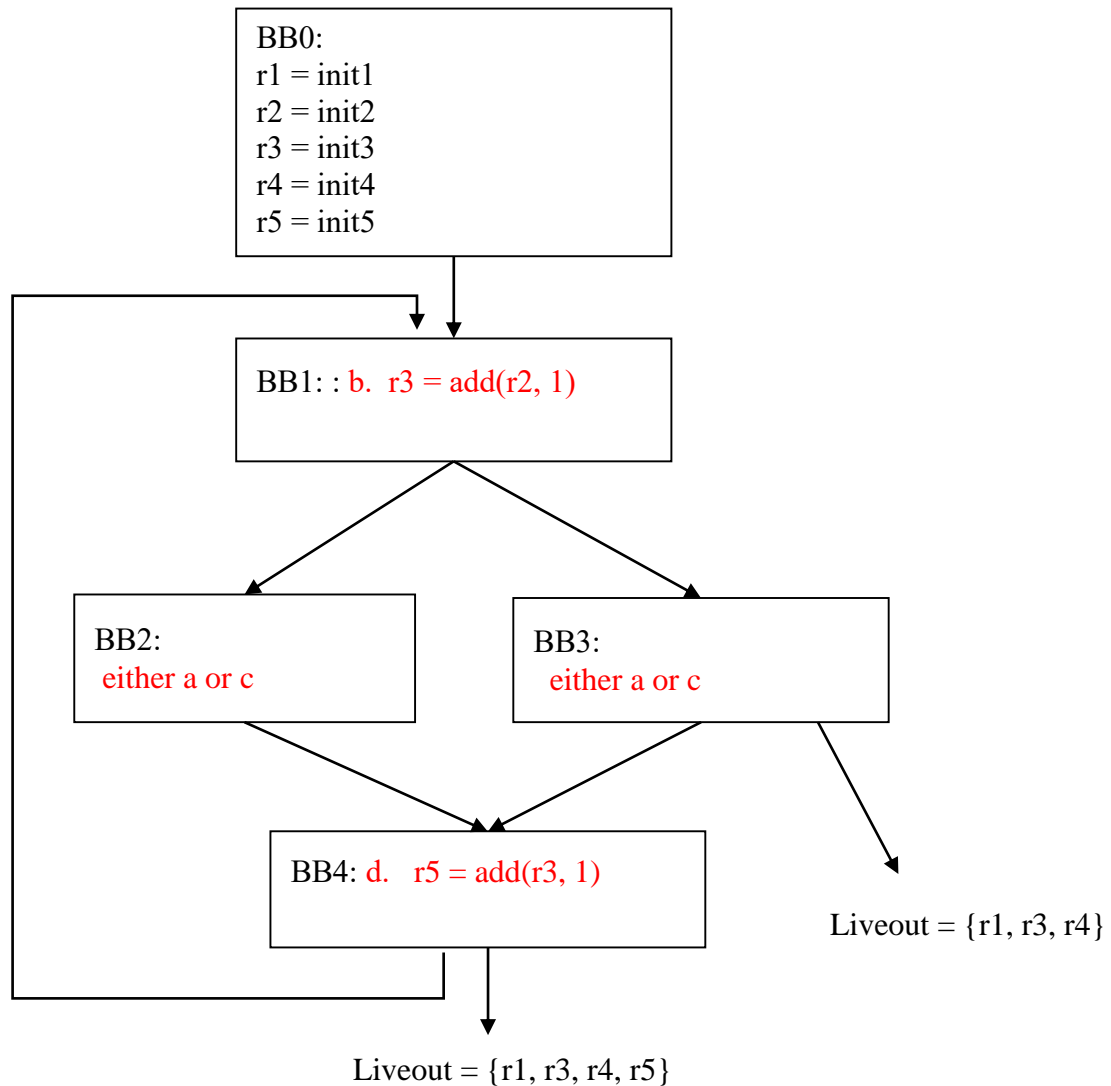
10) You developed a new LICM optimization in your compiler and want to create a testcase to validate your optimization. For your testcase, you will use the loop below with 4 additional instructions that you will insert such that 2 of the instructions can be hoisted to BB0 as invariant and the other 2 cannot be hoisted because they violate one of the LICM conditions.

The new instructions are:
  a. r1 = add(r1, 1)
  b. r3 = add(r2, 1)
  c. r4 = add(r2, 1)
  d. r5 = add(r3, 1)

Each new instruction must go to a *separate BB* (i.e., one instruction in BB1, one in BB2, one in BB3 and one in BB4). Show where they should be inserted on the diagram below such that the code before and after LICM yields the same results. Hint: Remember to pay attention to live-out registers. (10 pts.)
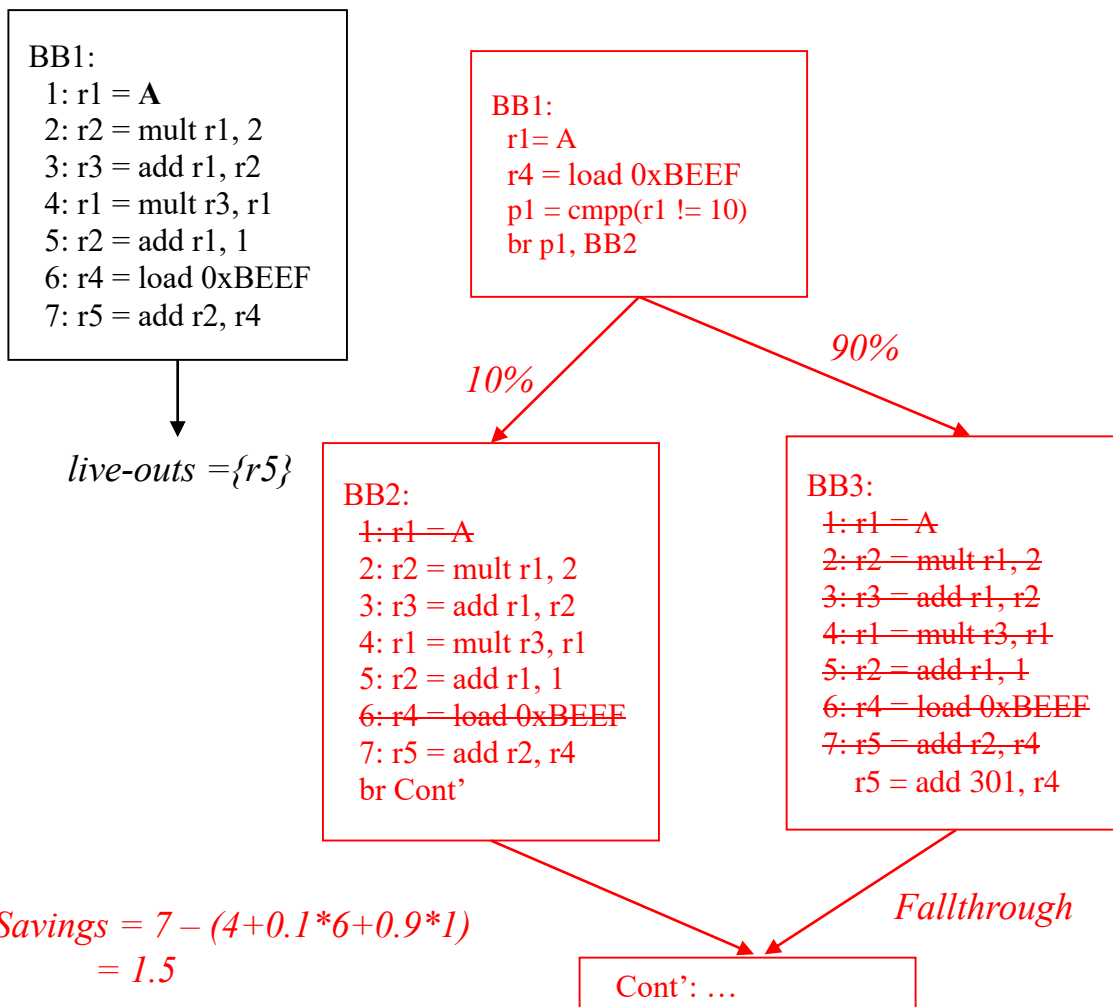

BB0:
r1 = init1
r2 = init2
r3 = init3
r4 = init4
r5 = init5

BB1: : b. r3 = add(r2, 1)

BB2:
 either a or c

BB3:
  either a or c

BB4: d. r5 = add(r3, 1)

Liveout = {r1, r3, r4}

Liveout = {r1, r3, r4, r5}

11) You are building a compiler where the goal is to extend conventional constant
propagation and folding by speculatively applying them using profiling data similar
to Homework 2.   Speculative constant propagation propagates statistically likely
constants to subsequent uses and enables speculative simplification using constant
folding. (10 pts)

Your primary testcase is the program segment below that initializes r1 with a constant
value, A. The profiling data shows that *A=10 with 90% probability*.
   a.   How should the compiler transform the code below to apply speculative
        constant propagation/folding? Show the optimized assembly and repair code.
        (7 pts)
   b.   What is the expected instruction savings for this example? (3 pts)

Note, *when counting saved instructions, DO NOT consider operation type but be sure*
to include any checking/repair instructions. For example, if you could save 10 loads
and 5 add instructions by inserting 1 comparison followed by 1 branch, your expected
saving would be 10+5-1-1=13.

BB1:
  1: r1 = **A**
  2: r2 = mult r1, 2
  3: r3 = add r1, r2
  4: r1 = mult r3, r1
  5: r2 = add r1, 1
  6: r4 = load 0xBEEF
  7: r5 = add r2, r4

live-outs ={r5}

BB1:
  r1= A
  r4 = load 0xBEEF
  p1 = cmpp(r1 != 10)
  br p1, BB2

10%   *90%*

BB2:
  1: r1 = A
  2: r2 = mult r1, 2
  3: r3 = add r1, r2
  4: r1 = mult r3, r1
  5: r2 = add r1, 1
  6: r4 = load 0xBEEF
  7: r5 = add r2, r4
  br Cont'

BB3:
  1: r1 = A
  2: r2 = mult r1, 2
  3: r3 = add r1, r2
  4: r1 = mult r3, r1
  5: r2 = add r1, 1
  6: r4 = load 0xBEEF
  7: r5 = add r2, r4
  r5 = add 301, r4

Fallthrough

Savings = 7 – (4+0.1*6+0.9*1)
*   = 1.5*

Cont': …

12) Given the modulo scheduled loop on the left that achieved II=MII, answer the following questions. Note that the scheduled code is just listed in linear order even though multiple instructions may execute each cycle. (15 pts)

```
Loop:
  1: r4[-1] = load(r2[0]) if p1[0]
  2: r6[-1] = load(r3[0]) if p1[0]
  3: r2[-1] = add r2[0], 4 if p1[0]
  4: r3[-1] = add r3[0], 4 if p1[0]
  5: r5[-1] = mult r4[-1], 30  if p1[0]
  6: r7[-1] = add r5[-1], r6[-1] if p1[1]
  7: store (r2[-1], r7[-1]) if p1[1]
  remap r1,r2,r3,…
  8: brlc Loop
```

Processor model

Resources
 4 issue, 4 fully pipelined units: 2 ALUs, 2 MEMs. **Note, branch (Instr 8) uses ALUs.**

Delays
 mult =3 cycles
 add = 1 cycle
 load = 1 cycles
 store = 1 cycle
 br = 1 cycle

* store(X, Y) saves value of Y to memory location X.

(a) Determine MII and explain whether the given code is resource constrained or recurrence constrained. (10 pts)

ResMII = MAX(5/2, 3/2) = 3

Circuits
  3->3 : 1/1 = 1
  4->4 : 1/1 = 1
  3->1->3 : (1+0)/(1+0) = 1
  4->2->4 : (1+0)/(1+0) = 1
  1->5->6->7->1 = (1+3+1+1)/(0+0+0+1) = 6
     ⇨ Since load (1) is getting value that is updated by store (7), there exists a cross-iteration memory dependency.

RecMII = MAX(1,1,1,1, 6) = 6
MII = MAX(ResMII, RecMII) = 6
Recurrence constrained

(b) How many cycles would it take to finish the loop? Assume LC = 99 (i.e., 100 iterations). (5 pts)

There was a typo in the given schedule that makes the rotating predicates wrong.
➔ 7 was supposed to be store(r2[0], r7[-1]) if p1[1]

With current version of exam, we will accept following two answers.

(1) If you get the number of stages from rotating predicates,
    (WRONG, BUT ACCEPTED)
        (N+ # stages -1)*II = (100+2-1)*6 = 606

(2) If you schedule by yourself and ignore rotating predicates, (CORRECT)

Assume lower instruction id has higher priority.
Unrolled schedule

	ALU	ALU	MEM	MEM
0	3	4	1	2
1		5		
2				
3				
4		6		
5	7	8		

Rolled schedule

	ALU	ALU	MEM	MEM
0	3	4	1	2
1		5		
2				
3				
4		6		
5	7	8		

Thus, # stages = 1
(N+ # stages -1)*II = (100+1-1)*6 = 600