EECS 583 – Class 16
Finish Modulo Scheduling
Exam Review

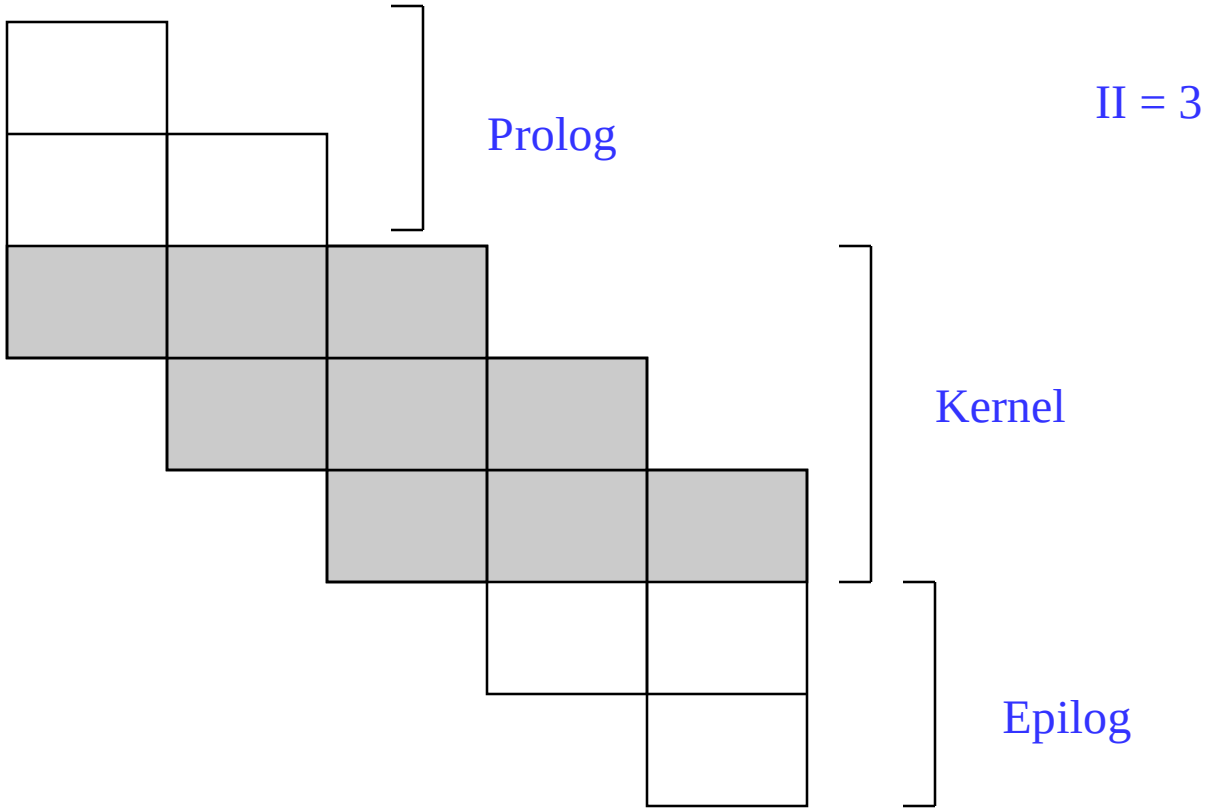
University of Michigan

November 11, 2019

Announcements

- Exam – 2 options, you can choose
  - Option 1: Wed Nov 13 10:40-12:30 – 2246 SRB (during class)
  - Option 2: Fri Nov 15 10:40-12:30 – 220 Chrysler
- If there is a notable difference in mean score between Wed and Fri results
  - We will consider this fact when determining final grades and make adjustments
- Covers through modulo scheduling, no register allocation

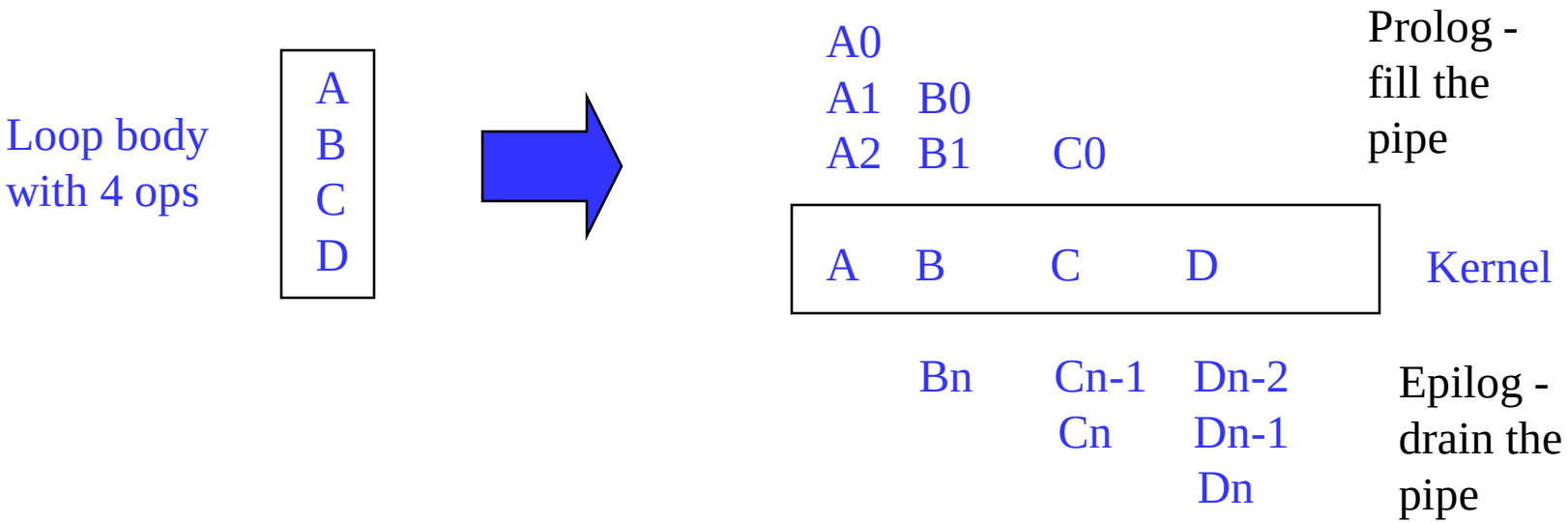
Loop Prolog and Epilog



Only the kernel involves executing full width of operations

Prolog and epilog execute a subset (ramp-up and ramp-down)

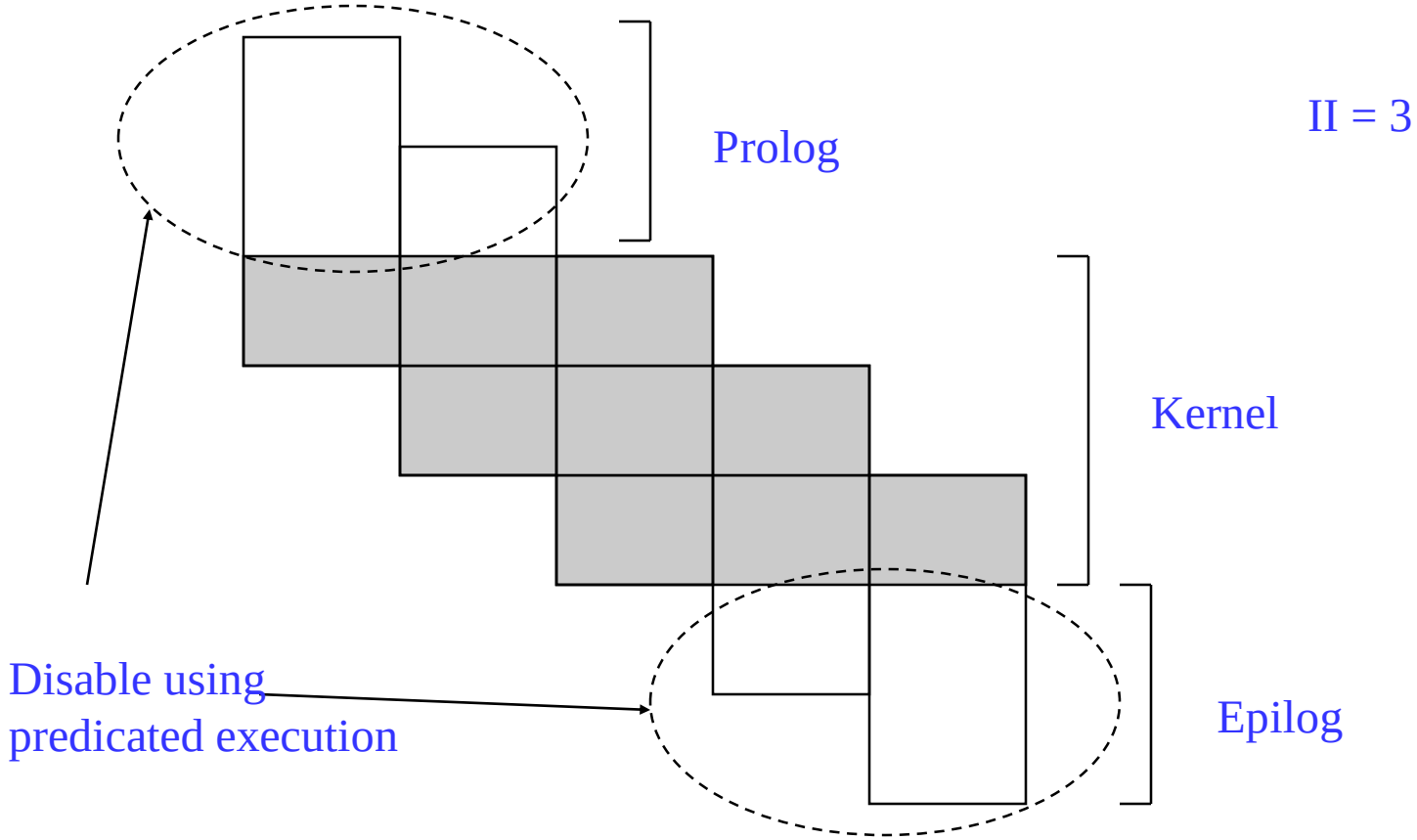
Separate Code for Prolog and Epilog



Generate special code before the loop (preheader) to fill the pipe and special code after the loop to drain the pipe.

Peel off II-1 iterations for the prolog. Complete II-1 iterations in epilog

Removing Prolog/Epilog

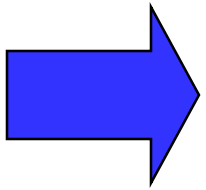


Execute loop kernel on every iteration, but for prolog and epilog selectively disable the appropriate operations to fill/drain the pipeline

Kernel-only Code Using Rotating Predicates

A0
A1 B0
A2 B1 C0

A	B	C	D



A if P[0] B if P[1] C if P[2] D if P[3]

Bn Cn-1 Dn-2
Cn Dn-1
Dn

P referred to as the staging predicate

P[0]	P[1]	P[2]	P[3]
1	0	0	0
1	1	0	0
1	1	1	0
1	1	1	1
…			
0	1	1	1
0	0	1	1
0	0	0	1

A	-	-	-
A	B	-	-
A	B	C	-
A	B	C	D
…			
-	B	C	D
-	-	C	D
-	-	-	D

Modulo Scheduling Architectural Support

- Loop requiring N iterations
  - Will take N + (S – 1) where S is the number of stages
- 2 special registers created
  - LC: loop counter (holds N)
  - ESC: epilog stage counter (holds S)
- Software pipeline branch operations
  - Initialize LC = N, ESC = S in loop preheader
  - All rotating predicates are cleared
  - SWP-BR
    - While LC > 0, decrement LC and RRB, P[0] = 1, branch to top of loop
      - This occurs for prolog and kernel
    - If LC = 0, then while ESC > 0, decrement RRB and write a 0 into P[0], and branch to the top of the loop
      - This occurs for the epilog

Execution History With LC/ESC

LC = 3, ESC = 3 /* Remember 0 relative!! */
Clear all rotating predicates
P[0] = 1

A if P[0]; B if P[1]; C if P[2]; D if P[3]; P[0] = BRF.B.B.F;

LC	ESC	P[0]	P[1]	P[2]	P[3]				
3	3	1	0	0	0	A			
2	3	1	1	0	0	A	B		
1	3	1	1	1	0	A	B	C	
0	3	1	1	1	1	A	B	C	D
0	2	0	1	1	1	-	B	C	D
0	1	0	0	1	1	-	-	C	D
0	0	0	0	0	1	-	-	-	D

4 iterations, 4 stages, II = 1, Note 4 + 4 –1 iterations of kernel executed

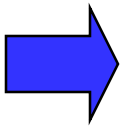
Modulo Scheduling Example

resources: 4 issue, 2 alu, 1 mem, 1 br
latencies: add=1, mpy=3, ld = 2, st = 1, br = 1

```
for (j=0; j<100; j++)
   b[j] = a[j] * 26
```

Step1: Compute to loop into form that uses LC

```
Loop: 1: r3 = load(r1)
      2: r4 = r3 * 26
      3: store (r2, r4)
      4: r1 = r1 + 4
      5: r2 = r2 + 4
      6: p1 = cmpp (r1 < r9)
      7: brct p1 Loop
```



LC = 99

```
Loop: 1: r3 = load(r1)
      2: r4 = r3 * 26
      3: store (r2, r4)
      4: r1 = r1 + 4
      5: r2 = r2 + 4
      7: brlc Loop
```

Example – Step 11

Step11: calculate ESC, SC = ceiling(max unrolled sched length / ii)
unrolled sched time of branch = rolled sched time of br + (ii*esc)

SC = 6 / 2 = 3, ESC = SC – 1
time of br = 1 + 2*2 = 5

LC = 99

Loop:

```
1: r3[-1] = load(r1[0])
2: r4[-1] = r3[-1] * 26
3: store (r2[0], r4[-1])
4: r1[-1] = r1[0] + 4
5: r2[-1] = r2[0] + 4
remap r1, r2, r3, r4
7: brlc Loop
```

Rolled Schedule

0	1 2 4
1	7 3 5

Unrolled Schedule

0	1 4
1	
2	2
3	
4	
5	3 5 7
6	

MRT

	alu0	alu1	mem	br
0	X	X	X	
1	X		X	X

Example – Step 12

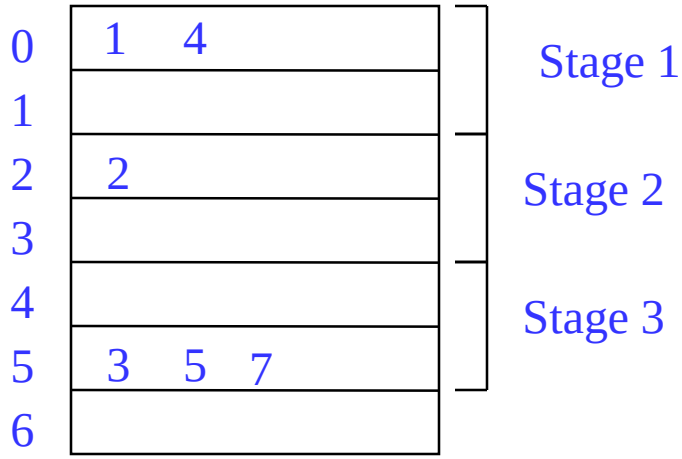
Finishing touches - Sort ops, initialize ESC, insert BRF and staging predicate, initialize staging predicate outside loop

LC = 99
ESC = 2
p1[0] = 1

Staging predicate, each successive stage increment the index of the staging predicate by 1, stage 1 gets px[0]

```
Loop:  1: r3[-1] = load(r1[0]) if p1[0]
       2: r4[-1] = r3[-1] * 26 if p1[1]
       4: r1[-1] = r1[0] + 4 if p1[0]
       3: store (r2[0], r4[-1]) if p1[2]
       5: r2[-1] = r2[0] + 4 if p1[2]
       7: brlc Loop if p1[2]
```

Unrolled Schedule



Example – Dynamic Execution of the Code

LC = 99
ESC = 2
p1[0] = 1

Loop:
1: r3[-1] = load(r1[0]) if p1[0]
2: r4[-1] = r3[-1] * 26 if p1[1]
4: r1[-1] = r1[0] + 4 if p1[0]
3: store (r2[0], r4[-1]) if p1[2]
5: r2[-1] = r2[0] + 4 if p1[2]
7: brlc Loop if p1[2]

Total time = II(num_iteration + num_stages – 1)
= 2(100 + 3 – 1) = 204 cycles

time: ops executed

0: 1, 4
1:

2: 1,2,4
3:

4: 1,2,4
5: 3,5,7

6: 1,2,4
7: 3,5,7

…

198: 1,2,4
199: 3,5,7

200: 2
201: 3,5,7

202: -
203 3,5,7

Homework Problem

latencies: add=1, mpy=3, ld = 2, st = 1, br = 1

```
for (j=0; j<100; j++)
  b[j] = a[j] * 26
```

```
      LC = 99
Loop: 1: r3 = load(r1)
      2: r4 = r3 * 26
      3: store (r2, r4)
      4: r1 = r1 + 4
      5: r2 = r2 + 4
      7: brlc Loop
```

How many resources of each type are required to achieve an II=1 schedule?

If the resources are non-pipelined, how many resources of each type are required to achieve II=1

Assuming pipelined resources, generate the II=1 modulo schedule.

Homework Problem – Answers in Red

latencies: add=1, mpy=3, ld = 2, st = 1, br = 1

```
for (j=0; j<100; j++)
  b[j] = a[j] * 26
```

```
      LC = 99
Loop: 1: r3 = load(r1)
      2: r4 = r3 * 26
      3: store (r2, r4)
      4: r1 = r1 + 4
      5: r2 = r2 + 4
      7: brlc Loop
```

How many resources of each type are required to achieve an II=1 schedule?
For II=1, each operation needs a dedicated resource, so: 3 ALU, 2 MEM, 1 BR

If the resources are non-pipelined, how many resources of each type are required to achieve II=1
Instead of 1 ALU to do the multiplies, 3 are needed, and instead of 1 MEM to do the loads, 2 are needed.
Hence: 5 ALU, 3 MEM, 1 BR

Assuming pipelined resources, generate the II=1 modulo schedule.
See next few slides

Problem continued

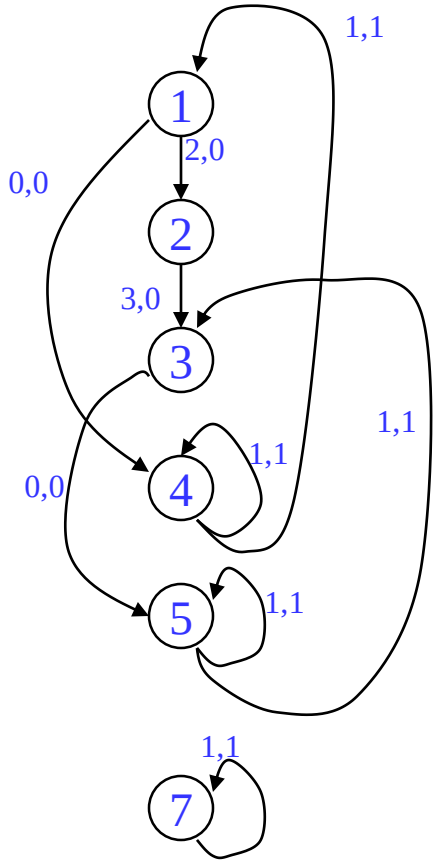
Assume II=1 so resources are: 3 ALU, 2 MEM, 1 BR

DSA converted code below (same as example in class)

LC = 99

```
Loop:  1: r3[-1] = load(r1[0])
       2: r4[-1] = r3[-1] * 26
       3: store (r2[0], r4[-1])
       4: r1[-1] = r1[0] + 4
       5: r2[-1] = r2[0] + 4
       remap r1, r2, r3, r4
       7: brlc Loop
```

Dependence graph (same as example in class)



RecMII = 1
RESMII = 1
MII = MAX(1,1) = 1

Priorities
1: H = 5
2: H = 3
3: H = 0
4: H = 4
5: H = 0
7: H = 0

Problem continued

resources: 3 alu, 2 mem, 1 br
latencies: add=1, mpy=3, ld = 2, st = 1, br = 1

LC = 99

Loop:

```
1: r3[-1] = load(r1[0])
2: r4[-1] = r3[-1] * 26
3: store (r2[0], r4[-1])
4: r1[-1] = r1[0] + 4
5: r2[-1] = r2[0] + 4
remap r1, r2, r3, r4
7: brlc Loop
```

Rolled Schedule

0	7 1 4 2 3 5

Unrolled Schedule

0	1 4	stage 1
1		stage 2
2	2	stage 3
3		stage 4
4		stage 5
5	3 5 7	stage 6
6		

Scheduling steps:

Schedule brlc at time II-1
Schedule op1 at time 0
Schedule op4 at time 0
Schedule op2 at time 2
Schedule op3 at time 5
Schedule op5 at time 5
Schedule op7 at time 5

MRT

	alu0	alu1	alu2	m1	m2	br
0	X	X	X	X	X	X

Problem continued

The final loop consists of a single MultiOp containing 6 operations, each predicated on the appropriate staging predicate. Note register allocation still needs to be performed.

```
    LC = 99
Loop:
    r3[-1] = load(r1[0]) if p1[0]; r4[-1] = r3[-1] * 26 if p1[2]; store (r2[0], r4[-1]) if p1[5]; r1[-1] = r1[0] + 4 if p1[0]; r2[-1] = r2[0] + 4 if p1[5]; brf Loop
```

Exam Review

Midterm Exam

- Where
  - This room on Wednes
  - OR Chrysler Auditorium on Fri
  - Start at 10:40 promptly
- What to expect
  - Open notes (bring whatever you like), but no laptops
  - Apply techniques we discussed in class
  - Reason about solving compiler problems – how/why things are done
  - A couple of thinking problems
  - No LLVM code
  - Reasonably long so don’t get stuck on a single problem

Midterm Exam – Continued

- 3 exams (F12-F13, F18) are posted on the course website
  - Note – Past exams may not accurately predict future exams!!
- Office hours
  - Armand: Mon (Today) 1-3pm
  - Sung: Tue 11-1
  - Scott: Tue 2:00-3:00
- Studying
  - Yes, you should study even though its open notes
    - Lots of material that you have likely forgotten from early this semester
    - Refresh your memories
    - No memorization required, but you need to be familiar with the material to finish the exam
  - Go through lecture notes, especially the examples!
  - If you are confused on a topic, go through the reading
  - Go through the practice exams (Don't look at the answer) as the final step

Exam Topics

- Control flow analysis
  - Control flow graphs, Dom/pdom, Loop detection
  - Trace selection, superblocks
- Predicated execution
  - Control dependence analysis, if-conversion
- Dataflow analysis
  - Liveness, reaching defs, DU/UD chains, available defs/exprs
  - Static single assignment
- Optimizations
  - Classical: Dead code elim, constant/copy prop, CSE, LICM, induction variable strength reduction
  - ILP optimizations - unrolling, tree height reduction, induction/accumulator expansion – Just understand the concepts
  - Speculative optimization – like HW2

Exam Topics - Continued

- Acyclic scheduling
  - Dependence graphs, Estart/Lstart/Slack, list scheduling
  - Code motion across branches, speculation, exceptions
  - Can ignore sentinel scheduling (delayed exceptions)
- Software pipelining
  - DSA form, ResMII, RecMII, modulo scheduling
  - Make sure you can modulo schedule a loop!
  - Execution control with LC, ESC
- Can ignore register allocation
- Can ignore automatic parallelization

Question 3 – Fall 2018

- When a compiler scheduler wants to speculate an instruction, name one issue that it must consider to preserve correctness of the resulting code.

Question 6 – Fall 2018

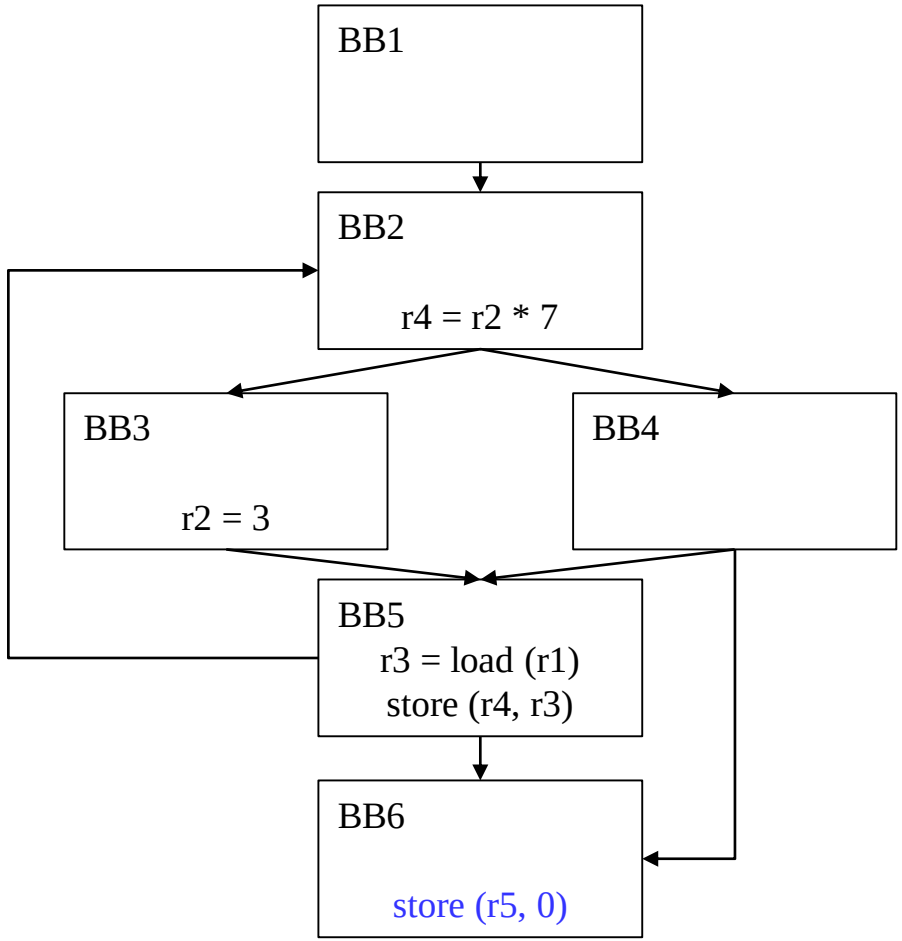
- Draw a control flow graph (CFG) consisting of 5 nodes (A, B, C, D, and E) that satisfies the 4 properties below. (10 pts)
  - A dominates all nodes.
  - B only dominates C and itself.
  - D only post dominates C and itself.
  - E post dominates all nodes

Question 9 – Fall 2018

- In the following control flow graph (CFG), place the 4 instructions (I-IV) such that the following conditions are met: at most 1 instruction is added to each basic block (BB); each instruction is placed at the beginning of a BB; after correctly placing all the instructions, the following optimizations should be applicable to the resulting code at least one time each: a) Forward Copy Propagation, b) Common Subexpression Elimination, c) Loop-Invariant Code Motion, and d) Dead Code Elimination. Do not worry about the impact of where you place the instructions on the execution results of the code segment. You may assume any relevant registers are properly initialized and the loads/stores are known to go to different addresses.

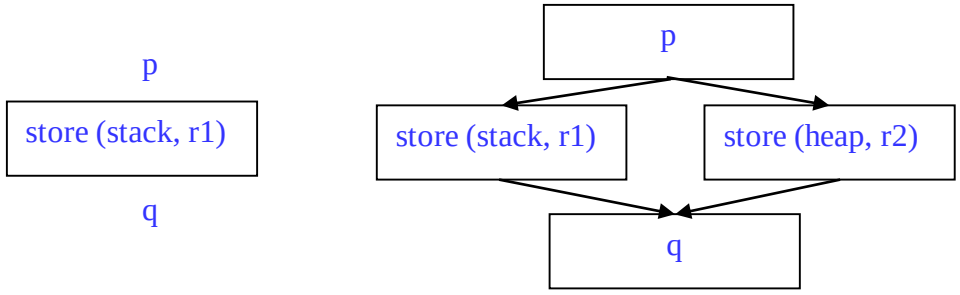
Question 9 – Fall 2018 (Continued)

I. r1 = r1 + 1
II. r5 = r8 * 7
III. r8 = r2
IV. r2 = r4 + 1



Question 11 – Fall 2018

- You are building a new compiler where you will have no memory dependence analysis capabilities. The only intelligence that you have is that you can differentiate stack/heap accesses. To enable some optimizations of loads (i.e., CSE or LICM), you decide to build a dataflow analysis pass to summarize the presence of heap/stack store instructions to avoid scanning basic blocks repeatedly. Your dataflow is defined as follows: *A heap (stack) store is present if there exists at least one store to the heap (stack) starting from p and ending at q*. When no heap (stack) store is present, you will be able to freely optimize loads between p and q.



Question 11 (continued)

- (a) Is this a forward or backward dataflow analysis problem?
- (b) Is this an all-path or any path dataflow analysis problem?
- (c) Define GEN and KILL sets to compute store presence. *Hint: Consider the use of special variables: heap and stack*