I) *Do bottom-up register allocation with 3 registers. When choosing a register to allocate always choose the lowest numbered one available. When choosing register to spill, choose the non-dirty register that will be used farthest in future. If all registers are dirty, choose the one that is used farthest in future. In case of a tie, choose the lowest numbered register.*

```
1. A = B + C
2. C = A + B
3. T1 = B + C
4. T2 = T1 + C
5. D = T2
6. E = A + B
7. B = E + D
8. A = C + D
9. T3 = A + B
10. WRITE(T3)
```

Ans:

	Instruction	**Live**
1	A = B + C	{ A, B }
2	C = A + B	{ A, B, C }
3	T1 = B + C	{ A, B, C, T1 }
4	T2 = T1 + C	{ A, B, C, T2 }
5	D = T2	{ A, B, C, D }
6	E = A + B	{ C, D, E }
7	B = E + D	{ B, C, D }
8	A = C + D	{ A, B }
9	T3 = A + B	{ T3 }
10	WRITE(T3)	{ }

1. A = B + C

Code generated	Register Map (* indicates dirty register)	Comments
LD B R1 LD C R2 ADD R1 R2 R2	R1: B R2: A*	• ensure(B) returns R1, associates B with R1 and generates LD B R1, • ensure(C) returns R2, associates C with R2 and generates LD C R2 • free(R2) marks R2 as free **(because C is dead)** • allocate(A) returns R2, associates R2 with A • generate code for ADD • mark R2 as dirty

2. C = A + B

Code generated	Register Map (* indicates dirty register)	Comments
ADD R2 R1 R3	R1: B R2: A* R3: C*	• ensure(A) returns R2 • ensure(B) returns R1 • allocate(C) returns R3, associates R3 with C • generate code for ADD • mark R3 as dirty

3. T1 = B + C

Code generated	Register Map (* indicates dirty register)	Comments
ADD R1 R3 R1	R1: T1* R2: A* R3: C*	• ensure(B) returns R1 • ensure(C) returns R3 • allocate(T1) returns R1, associates R1 with T1 **(No register is free. Chooses R1 to free since R1 is non-dirty and is used farthest in future (in instruction 6))** • generate code for ADD • mark R1 as dirty

4. T2 = T1 + C

Code generated	Register Map (* indicates dirty register)	Comments
ADD R1 R3 R1	R1: T2* R2: A* R3: C*	• ensure(T1) returns R1 • ensure(C) returns R3 • free(T1) marks R1 as free **(No Store despite R1 is dirty because T1 is dead)** • allocate(T2) returns R1, associates R1 with C • generate code for ADD • mark R1 as dirty

5. D = T2

Code generated	Register Map (* indicates dirty register)	Comments
	R1: D* R2: A* R3: C*	• ensure(T2) returns R1 • free(T2) marks R1 as free **(No Store despite R1 is dirty because T2 is dead)** • allocate(D) returns R1, associates R1 with D • No code generated • mark R1 as dirty

6. E = A + B

Code generated	Register Map (* indicates dirty register)	Comments
ST R3 C LD B R3 ADD R2 R3 R2	R1: D* R2: E* R3:	• ensure(A) returns R2 • ensure(B) returns R3 **(No register is free. All registers dirty. So, a call to allocate(B) chooses R3 among R1 and R3 to free. R3 is chosen because C is used farthest in future. free(R3) generates store because C is live. Associates R3 with B. Generates Load from B into R3)** • free(A) marks R2 as free **(No Store despite R2 is dirty because A is dead)** • free(B) marks R3 as free **(because B is dead)** • allocate(E) returns R2, associates R2 with E • Generate code for ADD • mark R2 as dirty

7. B = E + D

Code generated	Register Map (* indicates dirty register)	Comments
ADD R2 R1 R2	R1: D* R2: B* R3:	• ensure(E) returns R2 • ensure(D) returns R1 • free(E) marks R2 as free **(No Store despite R2 is dirty because E is dead)** • allocate(B) returns R2, associates R2 with B • Generate code for ADD • mark R2 as dirty

8. A = C + D

Code generated	Register Map (* indicates dirty register)	Comments
LD C R3 ADD R3 R1 R1	R1: A* R2: B* R3:	• ensure(C) returns R3, associates R3 with C, generates load from C into R3 • ensure(D) returns R1 • free(C) marks R3 as free **(C is dead)** • free(D) marks R1 as free **(no store despite R1 being dirty because D is dead)** • allocate(A) returns R1, associates R1 with A • Generate code for ADD • mark R1 as dirty

9. T3 = A + B

Code generated	Register Map (* indicates dirty register)	Comments
ADD R1 R2 R1	R1: T3* R2: R3:	• ensure(A) returns R1 • ensure(B) returns R2 • free(A) marks R1 as free **(A is dead, No store.)** • free(B) marks R2 as free **(B is dead. No store)** • allocate(T3) returns R1, associates R1 with T3 • Generate code for ADD • mark R1 as dirty

10. WRITE(T3)

Code generated	Register Map (* indicates dirty register)	Comments
WRITE(R1)	R1: R2: R3:	• ensure(T3) returns R1 • free(T3) marks R1 as free • Generate code for WRITE

Summarizing:

	Instruction	Live	Registers			Code
			R1	R2	R3	
1	A = B + C	{ A, B }	B	A*		LD B R1 LD C R2 ADD R1 R2 R2
2	C = A + B	{ A, B, C }	B	A*	C*	ADD R2 R1 R3
3	T1 = B + C	{ A, B, C, T1 }	T1*	A*	C*	ADD R1 R3 R1
4	T2 = T1 + C	{ A, B, C, T2 }	T2*	A*	C*	ADD R1 R3 R1
5	D = T2	{ A, B, C, D }	D*	A*	C*	
6	E = A + B	{ C, D, E }	D*	E*		ST R3 C LD B R3 ADD R2 R3 R2
7	B = E + D	{ B, C, D }	D*	B*		ADD R2 R1 R2
8	A = C + D	{ A, B }	A*	B*		LD C R3 ADD R3 R1 R1
9	T3 = A + B	{ T3 }	T3*			ADD R1 R2 R1
10	WRITE(T3)	{ }				WRITE R1

II. *Two ALUs (fully pipelined) and one LD/ST unit (not pipelined) are available. Either of the ALUs can execute ADD (1 cycle). Only one of the ALUs can execute MUL (2 cycles). LDs take up an ALU for 1 cycle and LD/ST unit for two cycles. STs take up an ALU for 1 cycle and LD/ST unit for one cycle. i) Draw reservation tables, ii) DAG for the code shown iii) schedule using height based list scheduling.*

```
1. LD A R1
2. LD B R2
3. LD C R3
4. LD D R4
5. R5 = R1 + R2
6. R6 = R5 * R3
7. R7 = R1 + R6
8. R8 = R6 + R5
9. R9 = R4 + R7
10. R10 = R9 + R8
11. ST R10 E
12. ST R7 F
```

Ans:

(i)

Reservation Tables

ADD

ALU1	ALU2	LD/ST
	✓	

ALU1	ALU2	LD/ST
✓		

ADDs take up either of the ALUs and occupy that ALU for a single cycle. Hence, we show two tables and a single row of occupancy.

MUL

ALU1	ALU2	LD/ST
✓		

MULs can execute on only one of the ALUs. Hence, we show a single table. Here, I assume that only ALU1 can execute MUL. Further, MULs take up two cycles to complete. Hence, we show two rows of occupancy. The ALUs are fully pipelined. This means that while ALU1 is executing the second cycle of `mul1`, it is also available to execute another instruction (`mul2 / add1 / ld1 / st1 `).

LD

ALU1	ALU2	LD/ST
✓		
		✓
		✓

ALU1	ALU2	LD/ST
	✓	
		✓
		✓

LDs take up an ALU (either of the ALUs) for one cycle and LD/ST unit for two cycles. Hence, we show two tables and three rows of occupancy.

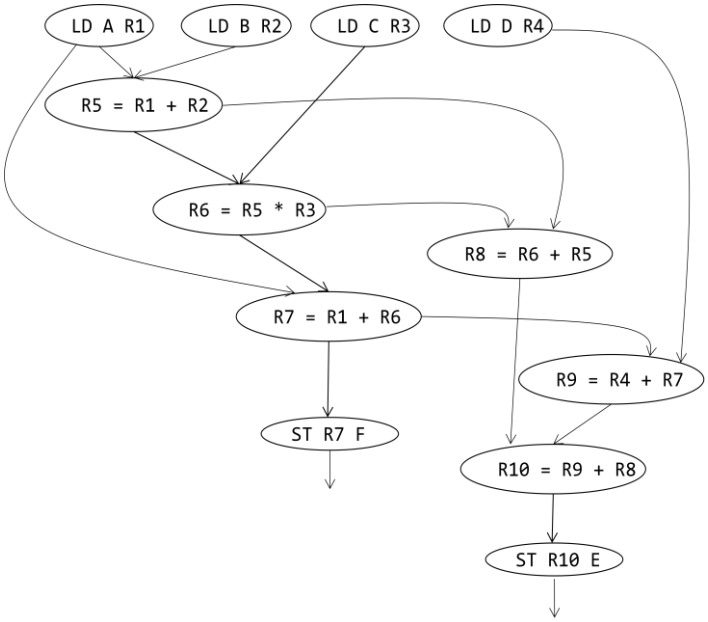
ST

ALU1	ALU2	LD/ST
✓		
		✓

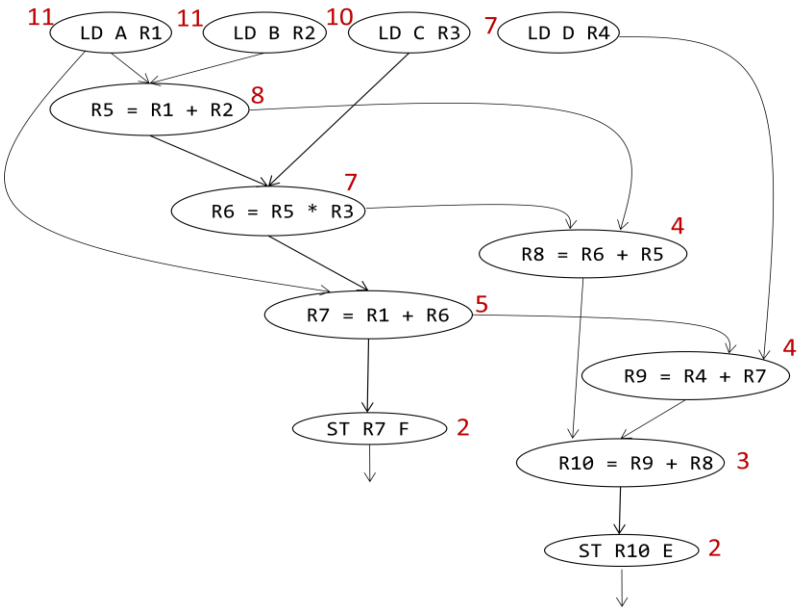
ALU1	ALU2	LD/ST
	✓	
		✓

STs take up an ALU (either of the ALUs) for one cycle and LD/ST unit for one cycle. Hence, we show two tables and two rows of occupancy.

(ii) DAG



(iii) DAG with heights assigned to nodes (height of a leaf node = latency of that instruction. Height of an interior node = maximum of heights of all children + latency of that instruction)



Cycle #	Available Instructions	Scheduled Instructions	Completed Instructions	ALU1	ALU2	LD /ST	Comments
0	1, 2, 3, 4	1	-	1			1 and 2 have max height. 1 (picked at random) scheduled.
1	2, 3, 4	-	-			1	
2	2, 3, 4	2	-	2		1	2 scheduled to utilize available ALUs
3	3, 4	-	1			2	
4	3, 4	3	-	3		2	3 scheduled to utilize available ALUs
5	4, 5	5	2	5		3	5 becomes available as 1 and 2 are complete. 4 has to wait till the last cycle of 3 is executed.
6	4	4	5	4		3	5 finishes. 4 can now be scheduled.
7	6	6	3	6		4	3 finishes. 6 becomes available. Takes two cycles on fully-pipelined ALU1.
8						4	ALU1 is also executing 6 in its pipeline.
9	7, 8	7, 8	4, 6	7	8		4 and 6 finish. As a result, 7 and 8 become available. Both can be scheduled on different ALUs.
10	9, 12	9, 12	7, 8	9	12		7 and 8 finish. As a result, 9 and 12 are available. Both can be scheduled.
11	10	10	9	10		12	9 finishes. 10 becomes available.
12	11	11	10, 12	11			10 and 12 finish. 11 becomes available.
13						11	
14			11				11 finishes.

III.

Assignment 2 – Q3 (8 mins)

```
1. X := 2
2. Label1:
3. Y := X + 1
4. if Z > 8 goto Label2
5. X := 3
6. X := X + 5
7. Y := X + 5
8. X := 2
9. if Z > 10 goto Label1
10.X := 3
11.Label2:
12.Y := X + 2
13.X := 0
14.goto Label3
15.X := 10
16.X := X + X
17.Label3:
18.Y := X + 1
```

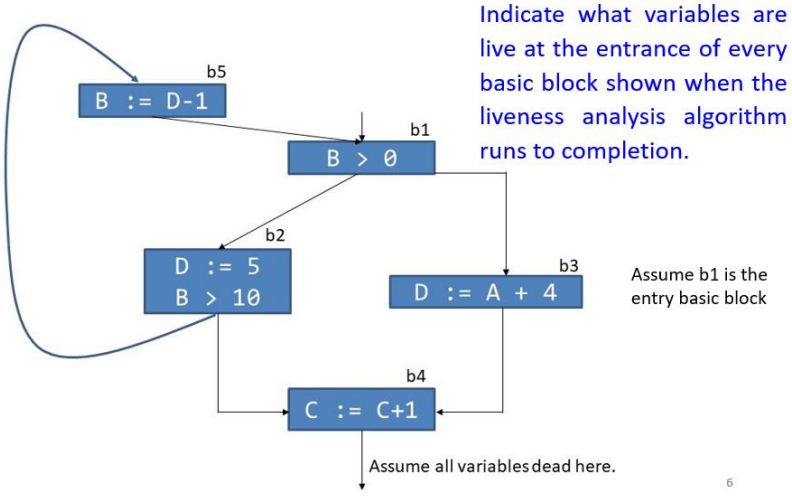
1. Draw a CFG for the code shown using the methods that we discussed in class
   1. Show the set of leaders
   2. Show the set of basic blocks
2. Apply dead-code elimination. How will the CFG change? Indicate in the previously drawn CFG.

Answer (part 1): leaders: {1, 2/3, 5, 10, 11/12, 15, 17/18} acceptable if you write 2 or 3 as part of the leader set but not both. Similarly, 11/12 and 17/18. Basic Blocks: there are seven basic blocks each having instructions with numbers starting from the leader and up to but not including the next leader.

Answer (part 2): After applying dead-code elimination, the CFG will not contain the basic block having instructions 15, 16

IV.

Assignment 2 – Q4 (12 mins)



Indicate what variables are live at the entrance of every basic block shown when the liveness analysis algorithm runs to completion.

Assume b1 is the entry basic block

Answer:

Block	USE	DEF	IN	OUT	IN	OUT
b1	B	{}	A,B,C	A,B,C	A, B, C	A, B, C
b2	B	D	B,C	C	A, B, C	A, C, D
b3	A	D	A,C	C	A, C	C
b4	C	C	C	{}	C	{}
b5	D	B	A,C,D	A,B,C	A, C, D	A, B, C

Order of computation used in computing the table:

IN_b4, OUT_b3, IN_b3, OUT_b2, IN_b2, OUT_b1, IN_b1, OUT_b5, IN_b5

Change in IN_b5 requires the edge b2-b5 to be revisited. Hence compute: OUT_b2, IN_b2.

Change in IN_b2 requires OUT_b1 to be revisited. Hence compute: OUT_b1, IN_b1.

IN and OUT sets for other blocks remain the same as earlier.

IN_b1 and OUT_b1 don't change. Hence, there are no more edges to be processed and

The worklist algorithm halts.

V.

Assignment 2 – Q5 (5 mins)

```
for(i=1;i<10;i++)
   for(j=1;j<10;j++) {
      a[i][j] = b[i][j] + c[i][j];
      d[i-1][j-1] = a[i-1][b-1];
   }
}
```

Name the loops whose iterations can be executed in parallel. If no loops qualify, write 'None'. If one or more loops qualify, for each loop that qualifies explain why the loop iterations can be executed in parallel.

8

Answer: Loop j can be executed in parallel. The dependencies are on previous iteration values.