Midterm Exam Solution (3 questions; 75 points)
(CS 4240: Compilers, Spring 2019)

1) Intermediate Code, Control Flow Graphs, Redundancy Elimination (25 points)

Consider the following source code that uses the standard do-while, if, and break control flow constructs, available in many standard programming languages, including C and Java.

```
1.    // Search for target in arr[0] ... arr[n-1]
2.    // If found, print index of first occurrence. Otherwise, print n.
3.    int i = 0;
4.    for (i = 0; i < n; i++) {
5.       if (arr[i] == n*n) break;
6.    } // for
7.    print i; // Output = n implies that target was not found
```

1a. Show possible intermediate representation (IR) code for the above source code, using the following instructions only. You may introduce as many new temporary variables as you like. In the instructions below, x, y, and z can refer to scalar variables or array elements, such as arr[i]. **(10 points)**

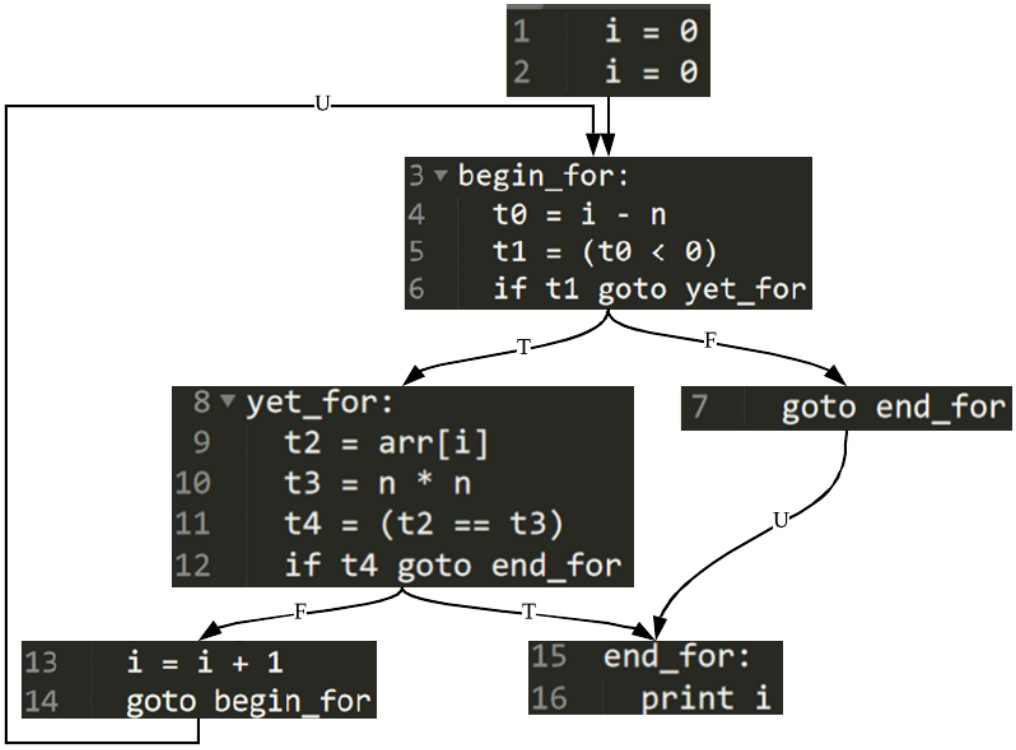
Binary Operator Assignment	x = y *op* z, for *op* in { +, − , ∗, /, >, <, ==, != }
Copy	x = y
Unconditional Jump	goto *label*
Conditional Jump	if x goto *label*
Label	*label*:
Print Operation	*print* x

Sample Answer

```
1    i = 0
2    i = 0
3  begin_for:
4    t0 = i - n
5    t1 = (t0 < 0)
6    if t1 goto yet_for
7    goto end_for
8  yet_for:
9    t2 = arr[i]
10   t3 = n * n
11   t4 = (t2 == t3)
12   if t4 goto end_for
13   i = i + 1
14   goto begin_for
15 end_for:
16   print i
```

1b. Show the control flow graph (CFG) for the intermediate code in part 1a above. You can choose whatever granularity of basic blocks you prefer, so long as you clearly show the instructions in each basic block. **(5 points)**

Sample Answer (Assuming maximal basic blocks)



1c. What opportunities for partial/total redundancy elimination do you see in this program? Explain your answer, and indicate how the CFG in part 1b can be modified to implement the optimization, i.e., which instructions need to be removed/inserted and where? **(10 points)**

- Value of n is never changed inside the for-loop. Therefore, it is redundant to calculate n*n every iteration of the loop. Moving line 10 ("t3 = n * n") to the entry basic block (containing lines 1~2) can remove the partial redundancy of calculating "n*n" inside the for-loop.
- Variable 'i' is initialized to 0 twice at the beginning of the program. Line 2 can be deleted.

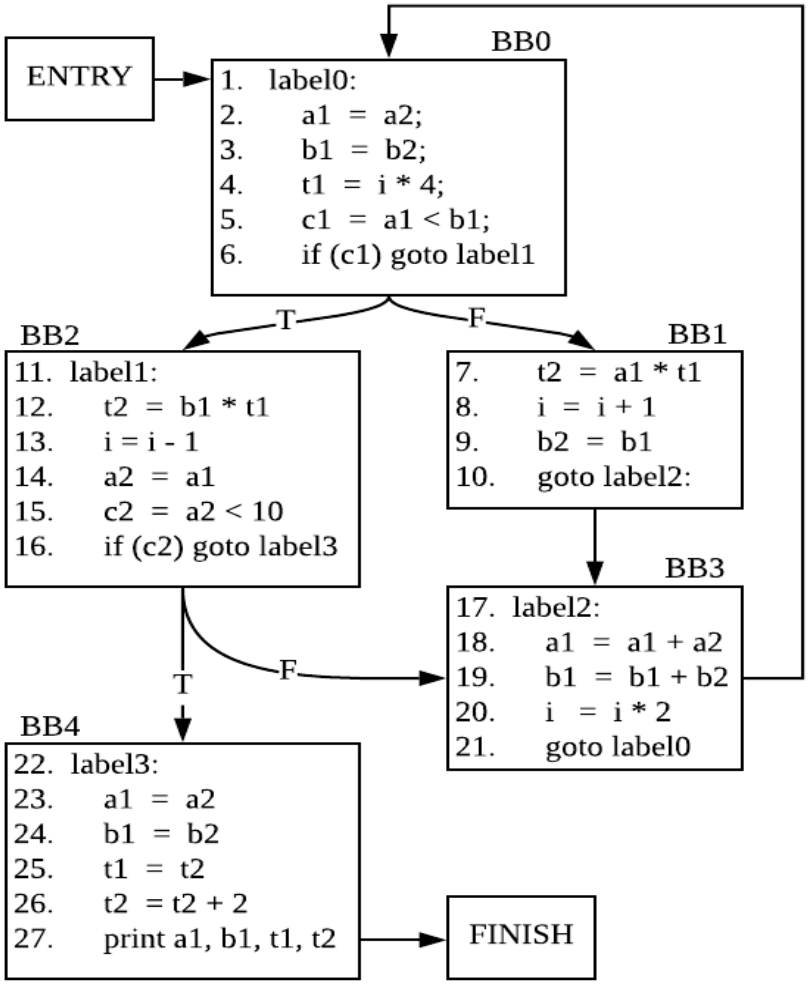
2) Liveness Analysis and Register Allocation (35 points)

Consider the following IR program, and assume that IR variables have been initialized as needed before reaching this code to ensure that no unitialized value is read.

```
1.    label0:
2.    a1 = a2;
3.    b1 = b2;
4.    t1 = i * 4;
5.    c1 = a1 < b1
6.    if (c1) go to label1
7.    t2 = a1 * t1;
8.    i = i + 1;
9.    b2 = b1;
10.   goto label2;
11.   label1:
12.   t2 = b1 * t1;
13.   i = i - 1;
14.   a2 = a1;
15.   c2 = a2 < 10
16.   if (c2) go to label3;
17.   label2:
18.   a1 = a1 + a2;
19.   b1 = b1 + b2;
20.   i = i * 2;
21.   go to label0;
22.   label3:
23.   a1 = a2;
24.   b1 = b2;
25.   t1 = t2;
26.   t2 = t2 + 2;
27.   print a1, b1, t1, t2
```

2a. Show the interference graph for this program, in which each vertex corresponds to a local IR variable (symbolic register) – *a1*, *a2*, *b1*, *b2*, *c1*, *c2*, *i*, *t1*, *t2* -- and an undirected edge between two variables indicates that their LIVE sets have a non-empty intersection. You do not need to provide the LIVE sets. Explain your answer by indicating one program point at which interference occurs, for each edge in the graph. **(10 points)**

Control Flow Graph of the given program.



Assuming that each program points are the entry point of an instruction,

Live(a1) = [3, 18] ∪ [24, 27]

Live(a2) = [1, 10] ∪ [15, 23]

Live(b1) = [4, 19] ∪ [25, 27]

Live(b2) = [1, 6] ∪ [10, 24]

Live(c1) = {6}

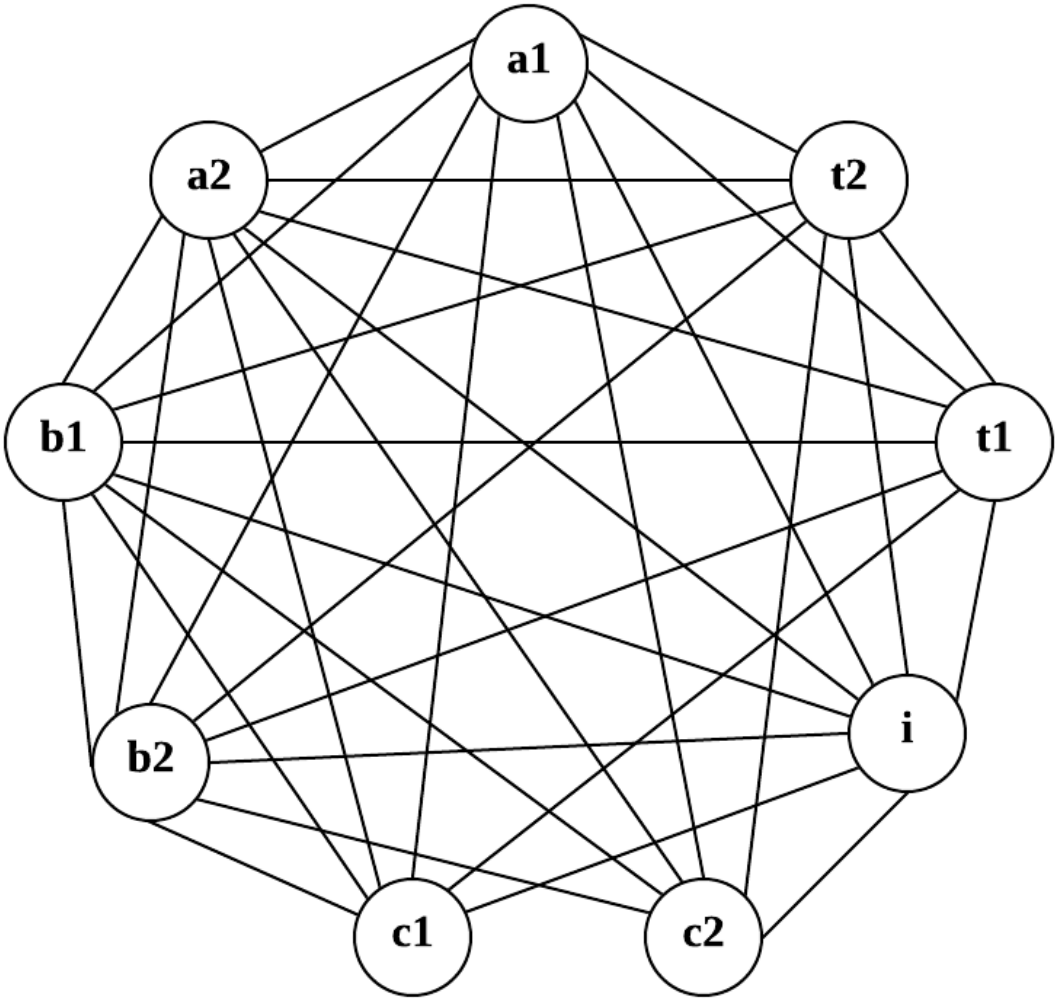
Live(c2) = {16}

Live(i) = [1, 21]

Live(t1) = [5, 7] ∪ [11, 12] ∪ [26, 27]

Live(t2) = [13, 16] ∪ [22, 27]

Interfernce graph of the given program
The graph has 33 edges in total.
(3 edges removed from a fully connected graph with 9 nodes; c1 – c2, c1 – t2, c2 – t1).



Grading Rubric for 2.a

- +5: correct interference graph
- +5: At least 10 correct interference points
- -0.2: each incorrect edge (missing/extra)

- -0.5: each incorrect(or missing) interference points
  (no deductions if student gave 10 correct interference points)

2b. Perform a spill-free register allocation for the above program, using the minimum number of physical registers (colors). Show the mapping of IR variables to physical registers. Your allocation should not assign the same physical register to more than one live variable at any program point. **(10 points)**

Answer : It is possible to do register allocation with 7 registers (minimum) as below.

a1 : R0

a2 : R1

b1 : R2

b2 : R3

c1 : R4, t2 : R4

c2 : R5, t1 : R5

i : R6

Grading Rubric for 2b

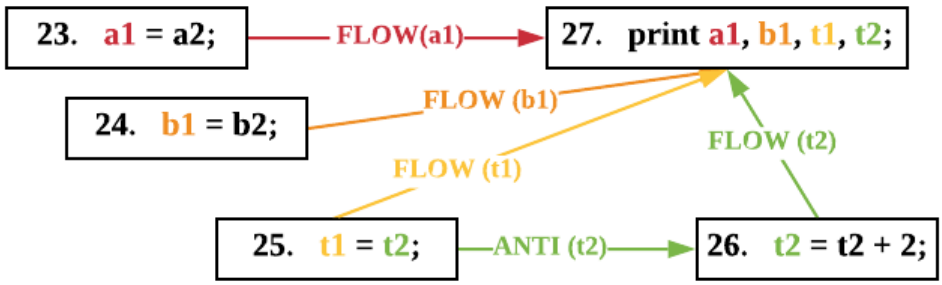
- +10: correct mapping (7 colors)
- +5: correct mapping but not the best
- 2: #colors < 7

- 0: empty answer

2c. Consider the following basic block starting at label3. Assume that these five instructions – *i23, i24, i25, i26, i27* – correspond directly to machine instructions. Show the dependence graph for these five instructions, assuming that they use the original virtual registers shown below (not the physical registers from part 2b), and label each dependence graph edge as flow/anti/output. Your graph should include transitive dependences as well, if any. **(5 points)**

```
23.      a1 = a2;                // Execution time = 1 cycle
24.      b1 = b2;                // Execution time = 1 cycle
25.      t1 = t2;                // Execution time = 1 cycle
26.      t2 = t2 + 2;            // Execution time = 2 cycles
27.      print a1, b1, t1, t2    // Execution time = 5 cycles
```

Sample Answer



2d. Provide an instruction schedule for the dependence graph in part 2c by specifying START(i), the cycle time at which an instruction i starts, and END(i), the cycle time at which instruction i ends, for each of the five instructions. Your schedule must obey the constraints listed below, and minimize the completion time, which is the maximum END(i) value over all instructions, i. Assume that there are sufficient hardware resources to ensure that the constraints below are the only constraints that need to be obeyed. **(10 points)**

- Schedule constraints for all instructions i, j
  - START(i) >= 0
  - If i != j then START(i) != START(j), i.e., at most one instruction can start in a given cycle
  - END(i) = START(i) + TIME(i), where TIME(i) is the execution time shown above
  - If there is a dependence edge from instruction i to instruction j, then START(j) >= END(i). This applies to all dependences – flow, anti, and output.

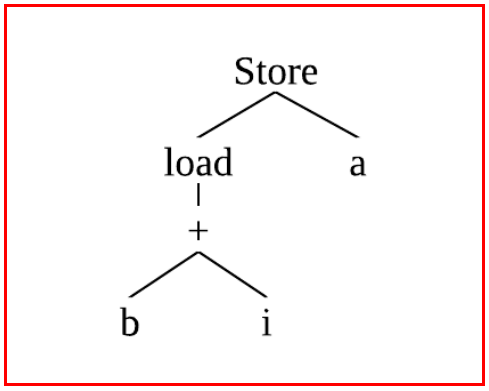
Instruction, i	START(i)	END(i)
i25	0	1
i26	1	3
i23	2	3
i24	3	4
i27	4	9

3) Instruction Selection (15 points)

Consider the following IR code, which uses virtual registers a, b, i, t0, t1:

```
1. t0 = b + i
2. t1 = load *t0
3. store *t1 = a
```

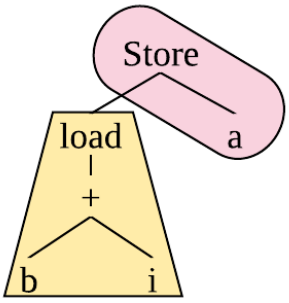
3a. Show the dependency tree for this IR program. Each leaf node of the tree should be labeled with a virtual register that is used but not defined, i.e., a, b, or i. Each internal node of the tree should be labeled with the operation that performs the computation. **(5 points)**



3b. Consider a hypothetical processor that provides the following multiple instructions for implementing compositions of addition, load and store operations on registers, along with their costs. Give an instruction selection (tiling) of the dependency tree from part 3a with minimum cost. **(10 points)**

Machine Instruction	**Cost**
R0 = R1 + R2 // Add registers R1 and R2, and place result in register R0	10
R0 = load *R1 // Load contents of address R1 from memory, and place in R0	20
store *R0 = R1 // Store contents of R1 in address R0 in memory	20
R0 = load *(R1+R2) // Load contents of address R1+R2 from memory, and place in R0	25
store *R0 = load *R1 // Move contents of address R1 from memory into address R0 in memory	30

Minimum tiling cost = 20 (store *R0 = R1 **) + 25 (** R0 = load *(R1+R2) **) = 45**



The tile of cost 30 (store *R0 = load *R1**) is not applicable to the given IR code, since the instruction for that tile stores the loaded value to memory but the given IR code is storing the value of 'a' to the address loaded from memory.**