INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

Stamp / Signature of the Invigilator

EXAMINATION (End Semester)	SEMESTER (Autumn)

Roll Number										Section		Name	

Subject Number	C	S	3	1	0	0	3	Subject Name	*Compilers*

Department / Center of the student		Additional Sheets	

Important Instructions and Guidelines for Students

1. You must occupy your seat as per the Examination Schedule/Sitting Plan.
2. Do not keep mobile phones or any similar electronic gadgets with you even in the switched off mode.
3. Loose papers, class notes, books or any such materials must not be in your possession, even if they are irrelevant to the subject you are taking examination.
4. Data book, codes, graph papers, relevant standard tables/charts or any other materials are allowed only when instructed by the paper-setter.
5. Use of instrument box, pencil box and non-programmable calculator is allowed during the examination. However, exchange of these items of any other papers (including question papers) is not permitted.
6. Write on both sides of the answer script and do not tear off any page. **Use last page(s) of the answer script for rough work.** Report to the Invigilator if the answer script has torn or distorted page(s).
7. It is your responsibility to ensure that you have signed the Attendance Sheet. Keep your Admit Card/Identity Card on the desk for checking by the invigilator.
8. You may leave the examination hall for wash room or for drinking water for a very short period. Record your absence from the Examination Hall in the register provided. Smoking and the consumption of any kind of beverages are strictly prohibited inside the Examination Hall.
9. Do not leave the Examination Hall without submitting your answer script to the invigilator. **In any case, you are not allowed to take away the answer script with you.** After the completion of the examination, do not leave the seat until the invigilators collect all the answer scripts.
10. During the examination, either inside or outside the Examination Hall, gathering information from any kind of sources or exchanging Information with others or any such attempt will be treated as '**unfair means**'. Do not adopt unfair means and do not indulge in unseemly behavior.

Violation of any of the above instructions may lead to severe punishment.

Signature of the Student

To be filled in by the examiner

Question Number	1	2	3	4	5	6	7	8	9	10	Total
Marks obtained											

Marks Obtained (in words)	Signature of the Examiner	Signature of the Scrutineer

CS31003 COMPILERS
AUTUMN 2026 – 2027
END-SEMESTER EXAMINATION
20-NOVEMBER-2026, 3:00PM – 5:00PM
MAXIMUM MARKS: 100

Instructions to students

- Write your answers in the question paper itself.
- Answer **all** questions.
- Write only in the blank spaces provided in the question paper. Use the empty pages at the end for rough work. Please avoid taking or using supplementary sheets. If you continue some answer to the Rough-Work pages, please supply an appropriate pointer (with the rough-page number) in the space of the corresponding question.
- Do not write anything on this page. Questions start from the next page (Page 3).
- If you need to make any assumptions in some questions, write them clearly in your respective answers.

1. [*Backpatching*]

A programming language uses the following two types of statements.

$S \rightarrow$ **do** S **unless** B ;
$S \rightarrow$ **do** S **until** B ;

Here, **`do`**, **`unless`**, and **`until`** are keywords, S is a statement, and B is a boolean condition. In the **`unless`** branching statement, S is carried out if and only if the condition B evaluates to *false*. The **`until`** statement is a loop that runs so long as the condition B remains *false*. Unlike do-while loops in C (which are post-test loops), the **`until`** loop is a pre-test loop, that is, the condition B is evaluated at the beginning of every iteration (including the first). To sum up, the statements are equivalent to the following usual statements.

do S **unless** B ;	is equivalent to	**if** (! B) S
do S **until** B ;	is equivalent to	**while** (! B) S

You are required to generate three-address codes on the fly for these two types of statements using backpatching. Continue to use the same backpatching mechanism as given in the Dragon Book (also covered in the class), that is, a boolean expression has two synthesized lists *truelist* and *falselist*, whereas a statement has a synthesized list *nextlist*. Use the functions *makelist*(), *merge*(), and *backpatch*() in the same manner as we have done for boolean expressions, and for branching (if and if-else) and looping (while) statements. Moreover, use the marker non-terminals M and N as earlier, with the following semantic actions. Let *nextinstr* denote the next instruction number.

$M \rightarrow \epsilon$ { $M.instr = nextinstr$; }
$N \rightarrow \epsilon$ { $N.nextlist = makelist(nextinstr)$; gen (' goto – ') ; }

In the two parts below, write your syntax-directed translation (SDT) schemes for only the two new statements. **[5 + 5]**

(a) Updated S-production for **unless** branching (add the necessary markers):

$S \rightarrow$ **do** N_1 M_1 S_1 **unless** N_2 M_2 B

SDT for the updated production:

$backpatch$ ($N_1.nextlist$, $M_2.instr$) ;
$backpatch$ ($B.falselist$, $M_1.instr$) ;
$S.nextlist = merge$ ($S_1.nextlist$, $merge$ ($N_2.nextlist$, $B.truelist$)) ;

(b) Updated S-production for **until** looping (add the necessary markers):

$S \rightarrow$ **do** N M_1 S_1 **until** M_2 B

SDT for the updated production:

$backpatch$ ($B.falselist$, $M_1.instr$) ;
$backpatch$ ($N.nextlist$, $M_2.instr$) ;
$backpatch$ ($S_1.nextlist$, $M_2.instr$) ; /* needed because there may be jumps from within the code of S_1 */
$S.nextlist = B.truelist$;

2. [*Run-time environment*]

Consider the following recursive implementation of binary search for a key `x` in a sorted array `A[]`.

```
int b ( int A[], int start , int end , int x )
{
    int mid, r;

    if (start > end) r = -1;
    else {
        mid = (start + end) / 2;
        if (x == A[mid]) r = mid;
        else if (x < A[mid]) r = b(A, start, mid - 1, x);     /* recursive call */
        else r = b(A, mid + 1, end, x);                       /* recursive call */
    }
    return r;
}
```

Assume the following.

- The size of `int` is 4 bytes, and the size of a memory address is 8 bytes.
- The control stack begins at address 1,000,000 and grows *downward* (that is, toward lower memory addresses).
- The stack pointer *SP* points to the beginning of the activation record last added to the control stack.
- The activation record of a function consists of the following items in the given sequence from higher to lower memory addresses (because the control stack grows toward lower memory addresses): (i) the formal parameters (an array is passed as its base address), (ii) the return value, (iii) the control link (for returning to the caller; a memory address in the code area), and (iv) the local variables appearing in the same sequence as they are declared. The computation in the body of `b()` requires some temporaries. There are sufficiently many registers, so these temporaries need not be stored in the activation record of `b()`.
- The activation record of the `main()` function is of size 4000 bytes.
- The return address for the outermost call of `b()` from `main()` is 1234, whereas the return addresses for the two recursive calls in the body of `b()` are respectively 2990 and 3010.

(a) Determine the size of an activation record for `b()`. Show your calculations. **[5]**

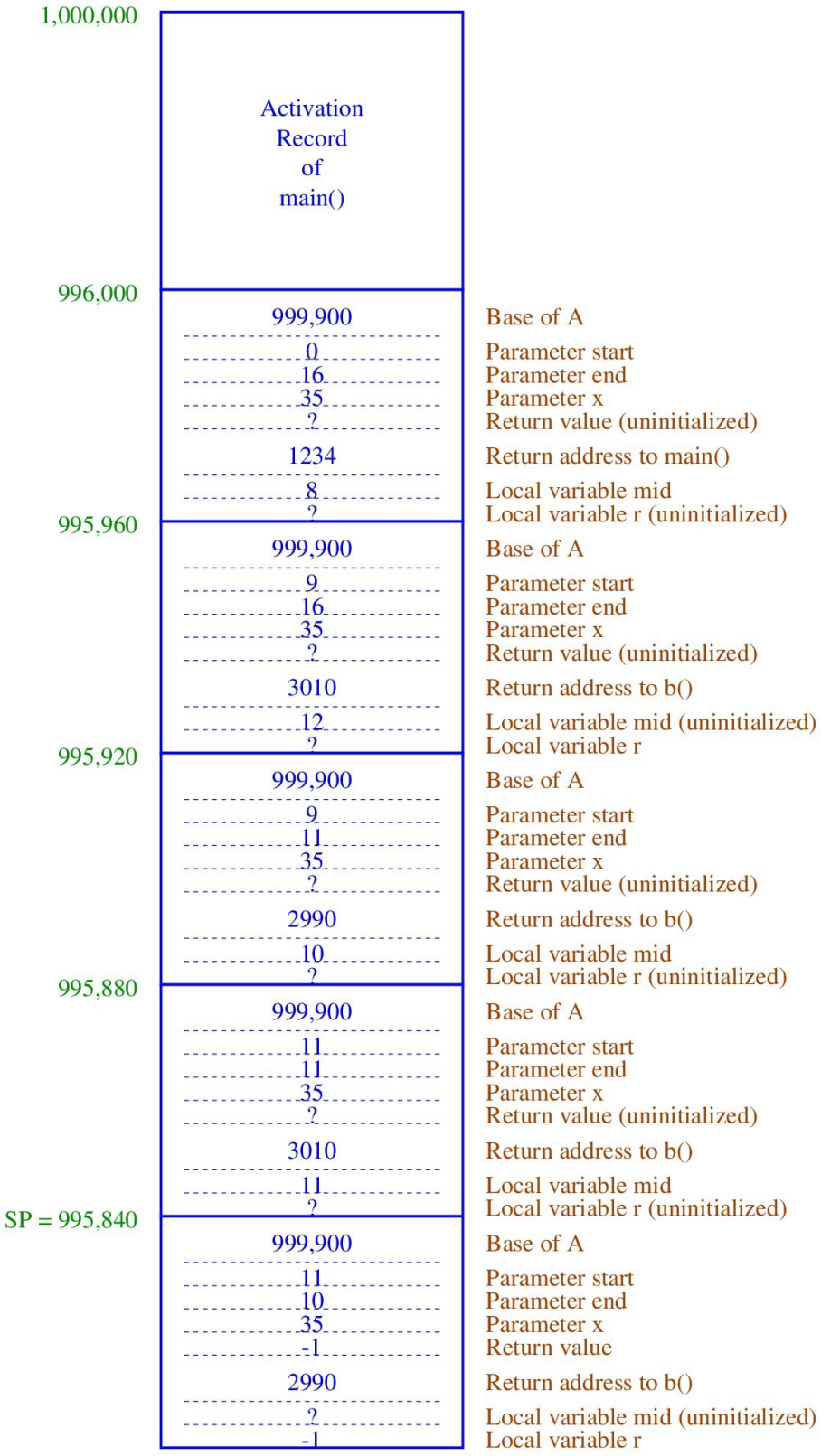
Formal parameters:	`A` (8 bytes), `start`, `end`, and `x` (3 × 4 = 12 bytes)
Return value:	`int` (4 bytes)
Control link:	An address (8 bytes)
Local variables:	`mid` and `r` (int, 4 + 4 bytes)

Total = 8 + 12 + 4 + 8 + 8 = 40 bytes

(b) You call `b(A,0,16,35)` from the `main()` function, where

`A` = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59]

is a local array in the `main()` function, the base of which is at a (negative) offset of 1000 relative to the start of the activation record of `main()` in the control stack. The search key 35 is not present in `A[]`, so the recursive function will return `-1`. Consider the time when the basis call that sees `start > end` is going to return `-1`. *Draw* the control stack at that instant showing the complete contents of all the current activation records of `b()`. Show the activation record of `main()` too but without the contents. Uninitialized values are to be shown as ?. Also show the beginning addresses of all the activation records currently at the control stack, and the address that *SP* is pointing to at that moment. **[10]**


1,000,000
Activation
Record
of
main()
996,000
999,900
Base of A
0
Parameter start
16
Parameter end
35
Parameter x
?
Return value (uninitialized)
1234
Return address to main()
8
Local variable mid
?
Local variable r (uninitialized)
995,960
999,900
Base of A
9
Parameter start
16
Parameter end
35
Parameter x
?
Return value (uninitialized)
3010
Return address to b()
12
Local variable mid (uninitialized)
?
Local variable r
995,920
999,900
Base of A
9
Parameter start
11
Parameter end
35
Parameter x
?
Return value (uninitialized)
2990
Return address to b()
10
Local variable mid
?
Local variable r (uninitialized)
995,880
999,900
Base of A
11
Parameter start
11
Parameter end
35
Parameter x
?
Return value (uninitialized)
3010
Return address to b()
11
Local variable mid
?
Local variable r (uninitialized)
SP = 995,840
999,900
Base of A
11
Parameter start
10
Parameter end
35
Parameter x
-1
Return value
2990
Return address to b()
?
Local variable mid (uninitialized)
-1
Local variable r

3. [*Register allocation*]

Consider the following basic block where `a`, `b`, `c`, `d` are user-defined variables, and `t1`, `t2`, `t3`, `t4`, `t5`, `t6` are temporaries. The instructions are numbered (1) through (10).

```
(1)    t1 = a * b
(2)    t2 = b * c
(3)    t3 = a * c
(4)    t4 = c - a
(5)    c = t4
(6)    t5 = t2 - t3
(7)    t6 = t1 - t5
(8)    b = b + t4
(9)    d = t5 % t6
(10)   if d == 0 goto L
```

(a) In the following table, show for each of the above ten instructions the next-use information of the variables and the temporaries. The next-use information can be "Not live", "Other block", or "(*k*)" (*k* is an instruction number). **[8]**

	a	b	c	d	t1	t2	t3	t4	t5	t6
(1) t1 = a * b	(1)	(1)	(2)	Not live	Not live	Not live	Not live	Not live	Not live	Not live
(2) t2 = b * c	(3)	(2)	(2)	Not live	(7)	Not live	Not live	Not live	Not live	Not live
(3) t3 = a * c	(3)	(8)	(3)	Not live	(7)	(6)	Not live	Not live	Not live	Not live
(4) t4 = c - a	(4)	(8)	(4)	Not live	(7)	(6)	(6)	Not live	Not live	Not live
(5) c = t4	Other block	(8)	Not live	Not live	(7)	(6)	(6)	(5)	Not live	Not live
(6) t5 = t2 - t3	Other block	(8)	Other block	Not live	(7)	(6)	(6)	(8)	Not live	Not live
(7) t6 = t1 - t5	Other block	(8)	Other block	Not live	(7)	Not live	Not live	(8)	(7)	Not live
(8) b = b + t4	Other block	(8)	Other block	Not live	Not live	Not live	Not live	(8)	(9)	(9)
(9) d = t5 % t6	Other block	Other block	Other block	Not live	Not live	Not live	Not live	Not live	(9)	(9)
(10) if d == 0 goto L	Other block	Other block	Other block	(10)	Not live	Not live	Not live	Not live	Not live	Not live

(b) A machine has four registers R1, R2, R3, and R4 for allocating to this basic block. At the beginning of the basic block, all registers are empty, and at the end of the basic block, you should clear all the registers. The variables `a`, `b`, and `c` are live on exit from the basic block, whereas the variable `d` is not. No temporaries are used outside this basic block. For register allocation, we use the following *getReg*() function (instead of the one used in the Dragon Book and covered in the class). By *value*, we refer to a user-defined variable or a temporary. The following steps are used for choosing a register, first for the two operands, and then for the result. Step 3 is called the *farthest fit* (*FF*) strategy.

1. If the value is already stored in a register, use that register.
2. If some register is empty, use that register for loading a value or for storing the result.
3. Compute the score of each register R as the *minimum* of the current next-use information of all values v stored in R (without considering the future use of these values). If v is not live, take its next use to be ∞. If v is live on exit from the basic block, its next use is larger than the largest instruction number in the basic block (in our example, we can take any fixed value > 10). If v has a next use in the block itself, then its next use is the instruction number of its current next use. A register with the *largest score* is chosen for loading an operand or for storing a result. If the result is the same as one of the operands, store the result in the register already chosen for that operand.

For a copy instruction $v = w$, a register is first chosen for w using the above three steps. The same register is used for v. If v already resides in some *other* register R, remove R from the current locations of v (this may make R free).

In Steps 2 and 3, ties are broken by taking the candidate register with the smallest number. Using the table of Part (a), determine the register allocation and the target code generated, for each three-address instruction in the basic block. Also show the register-descriptor table (RDT) and the address-descriptor table (ADT). Mark all the *spills* (that is the store instructions needed when a register is chosen for allocation). Temporaries may also be spilled. If a latest value already resides in the memory, it is not needed to be stored again in the memory. **[16]**

Initialization

	R1	R2	R3	R4						
RDT										
ADT	a	b	c	d						
	a	b	c	d	t1	t2	t3	t4	t5	t6

Target code generated: No code generated

(1) t1 = a * b

	R1	R2	R3	R4						
RDT	a	b	t1							
ADT	a, R1	b, R2	c	d	R3					
	a	b	c	d	t1	t2	t3	t4	t5	t6

Target code generated:

```
LD R1, a
LD R2, b
MUL R3, R1, R2
```

(2) t2 = b * c

	R1	R2	R3	R4						
RDT	a	b	t2	c						
ADT	a, R1	b, R2	c, R4	d	t1	R3				
	a	b	c	d	t1	t2	t3	t4	t5	t6

Target code generated:

```
LD R4, c
ST t1, R3
MUL R3, R2, R4
```

(3) t3 = a * c

	R1	R2	R3	R4						
RDT	a	t3	t2	c						
ADT	a, R1	b	c, R4	d	t1	R3	R2			
	a	b	c	d	t1	t2	t3	t4	t5	t6

Target code generated:

```
MUL R2, R1, R4
```

(4) t4 = c - a

	R1	R2	R3	R4						
RDT	a	t4	t2	c						
ADT	a, R1	b	c, R4	d	t1	R3	t3	R2		
	a	b	c	d	t1	t2	t3	t4	t5	t6

Target code generated:

```
ST t3, R2
SUB R2, R4, R1
```

(5) c = t4

	R1	R2	R3	R4						
RDT	a	c, t4	t2							
ADT	a, R1	b	R2	d	t1	R3	t3	R2		
	a	b	c	d	t1	t2	t3	t4	t5	t6

Target code generated

No code generated

(6) t5 = t2 - t3

	R1	R2	R3	R4						
RDT	t5	c, t4	t2	t3						
ADT	a	b	R2	d	t1	R3	t3, R4	R2	R1	
	a	b	c	d	t1	t2	t3	t4	t5	t6

Target code generated

```
LD R4, t3
SUB R1, R3, R4
```

(7) t6 = t1 - t5

	R1	R2	R3	R4						
RDT	t5	c, t4	t1	t6						
ADT	a	b	R2	d	t1, R3		t3	R2	R1	R4
	a	b	c	d	t1	t2	t3	t4	t5	t6

Target code generated

```
LD R3, t1
SUB R4, R3, R1
```

(8) b = b + t4

	R1	R2	R3	R4						
RDT	t5	c, t4	b	t6						
ADT	a	R3	R2	d	t1		t3	R2	R1	R4
	a	b	c	d	t1	t2	t3	t4	t5	t6

Target code generated

```
LD R3, b
ADD R3, R3, R2
```

(9) d = t5 % t6

	R1	R2	R3	R4						
RDT	t5	d	b	t6						
ADT	a	R3	c	R2	t1		t3		R1	R4
	a	b	c	d	t1	t2	t3	t4	t5	t6

Target code generated

```
ST c, R2
REM R2, R1, R4
```

(10) if d == 0 goto L

	R1	R2	R3	R4						
RDT	t5	d	b	t6						
ADT	a	R3	c	R2	t1		t3		R1	R4
	a	b	c	d	t1	t2	t3	t4	t5	t6

Target code generated

(use the instruction BZ)

```
BZ R2, L
```

What store instruction(s) should be generated at the end of the given basic block? **[1]**

```
ST b, R3
```

4. [*Optimization*]

A famous unsolved problem in number theory is the Collatz Conjecture which goes as follows.

Take any positive integer `n`. If `n` is even, divide it by `2`, whereas if `n` is odd, multiply it by `3` and add `1` to the product. Repeat the process over and over again with each new result as `n`. The conjecture states that no matter what positive integer `n` you start with, the sequence will always eventually reach `1`.

The following C code simulates the steps of this computation. The code uses a two-dimensional array `ST[100][3]` of integers. (i) `ST[i][0]` stores the total number of steps for the number `n` (initialized to `i + 1`) to reach 1, (ii) `ST[i][1]` stores the total number of steps `n = 3n+1` when `n` is odd (for the given `i`) and (iii) `ST[i][2]` stores the total number of steps `n = n/2` when `n` is even (for the given `i`). Assume that `0 ≤ m < 100`. All basic variables are of type `int`.

```
for ( i = 0 ; i < m ; ++i ) {
   n = i + 1;
   ST[i][0] = 0; ST[i][1] = 0; ST[i][2] = 0;
   while ( n > 1 ) {
      ST[i][0] = ST[i][0] + 1;
      if ( n % 2 == 1 ) {
         n = 3 * n + 1;
         ST[i][1] = ST[i][1] + 1;
      }
      else {
         n = n / 2;
         ST[i][2] = ST[i][2] + 1;
      }
   }
}
```

(a) Generate the three-address code for the above snippet using fall-through optimization that reduces the number of `goto` statements (use `ifFalse` in conditional jumps). Do not use any other optimization in this part. Assume that the size (width) of each `int` is `4` bytes. Use symbolic labels (`L1`, `L2`, `L3`, and so on) instead of instruction numbers. Name the temporaries as `t1`, `t2`, `t3`, and so on. Identify the basic blocks by enclosing them in rectangles. Name the basic blocks as B_1, B_2, B_3, and so on. **[10]**

```
    +--------------------------+
    | i = 0                    | Block B1
    +--------------------------+
L1: | ifFalse i < m goto L8    | Block B2
    +--------------------------+
L2: | t1 = i + 1               | Block B3
    | n = t1                   |
    | t2 = i * 12              |
    | t3 = 0 * 4               |
    | t4 = t2 + t3             |
    | ST[t4] = 0               |
    | t5 = i * 12              |
    | t6 = 1 * 4               |
    | t7 = t5 + t6             |
    | ST[t7] = 0               |
    | t8 = i * 12              |
    | t9 = 2 * 4               |
    | t10 = t8 + t9            |
    | ST[t10] = 0              |
    +--------------------------+
L3: | ifFalse n > 1 goto L7    | Block B4
    +--------------------------+
L4: | t11 = i * 12             | Block B5
    | t12 = 0 * 4              |
    | t13 = t11 + t12          |
    | t14 = ST[t13]            |
    | t15 = t14 + 1            |
    | ST[t13] = t15            |
    | t16 = n % 2              |
    | ifFalse t16 == 1 goto L6 |
    +--------------------------+
L5: | t17 = 3 * n              | Block B6
    | t18 = t17 + 1            |
    | n = t18                  |
    | t19 = i * 12             |
    | t20 = 1 * 4              |
    | t21 = t19 + t20          |
    | t22 = ST[t21]            |
    | t23 = t22 + 1            |
    | ST[t21] = t23            |
    | goto L3                  |
    +--------------------------+
L6: | t24 = n / 2              | Block B7
    | n = t24                  |
    | t25 = i * 12             |
    | t26 = 2 * 4              |
    | t27 = t25 + t26          |
    | t28 = ST[t27]            |
    | t29 = t28 + 1            |
    | ST[t27] = t29            |
    | goto L3                  |
    +--------------------------+
L7: | t30 = i + 1              | Block B8
    | i = t30                  |
    | goto L1                  |
    +--------------------------+
L8:
```

Note: Other correct variations are given due credit.

(b) Hand-optimize each basic block individually, by locating local common sub-expressions, by using constant propagation/folding and algebraic identities (including strength reduction by applying distributivity), and by eliminating dead code. You do not have to use the DAG representation of the basic blocks. But give clear justifications (in plain English) for every optimization step that you use. Do not perform any global optimization here. Do not renumber the temporaries. **[10]**

The optimization steps are discussed block by block. The basic blocks that cannot be optimized are omitted in the discussion.

Block B_3:
Optimizations Applied:
(i) Constant Propagation/Folding: `t3 = 0 * 4` → `0`, `t6 = 1 * 4` → `4`, `t9 = 2 * 4` → `8`.
(ii) Algebraic Identity: `t4 = t2 + 0` simplifies directly to `t4 = t2`.
(iii) Common Sub-expression Elimination (CSE): `t5` and `t8` compute the exact same value as `t2 = i * 12`. They are replaced with references to `t2`.
Updated Intermediate Code:

```
L2: t1 = i + 1
    n = t1
    t2 = i * 12
    ST[t2] = 0
    t7 = t2 + 4
    ST[t7] = 0
    t10 = t2 + 8
    ST[t10] = 0
```

Block B_5:
Optimizations Applied:
(i) Constant Folding: `t12 = 0 * 4` → `0`.
(ii) Algebraic Identity: `t13 = t11 + 0` simplifies to `t13 = t11`.
Updated Intermediate Code:

```
L4: t11 = i * 12
    t14 = ST[t11]
    t15 = t14 + 1
    ST[t11] = t15
    t16 = n % 2
    ifFalse t16 == 1 goto L6
```

Block B_6:
Optimizations Applied:
(i) Constant Folding: `t20 = 1 * 4` → `4`.
(ii) Constant propagation: `t21 = t19 + t20` simplifies to `t21 = t19 + 4`, making the computation of `t20` dead code.
Updated Intermediate Code:

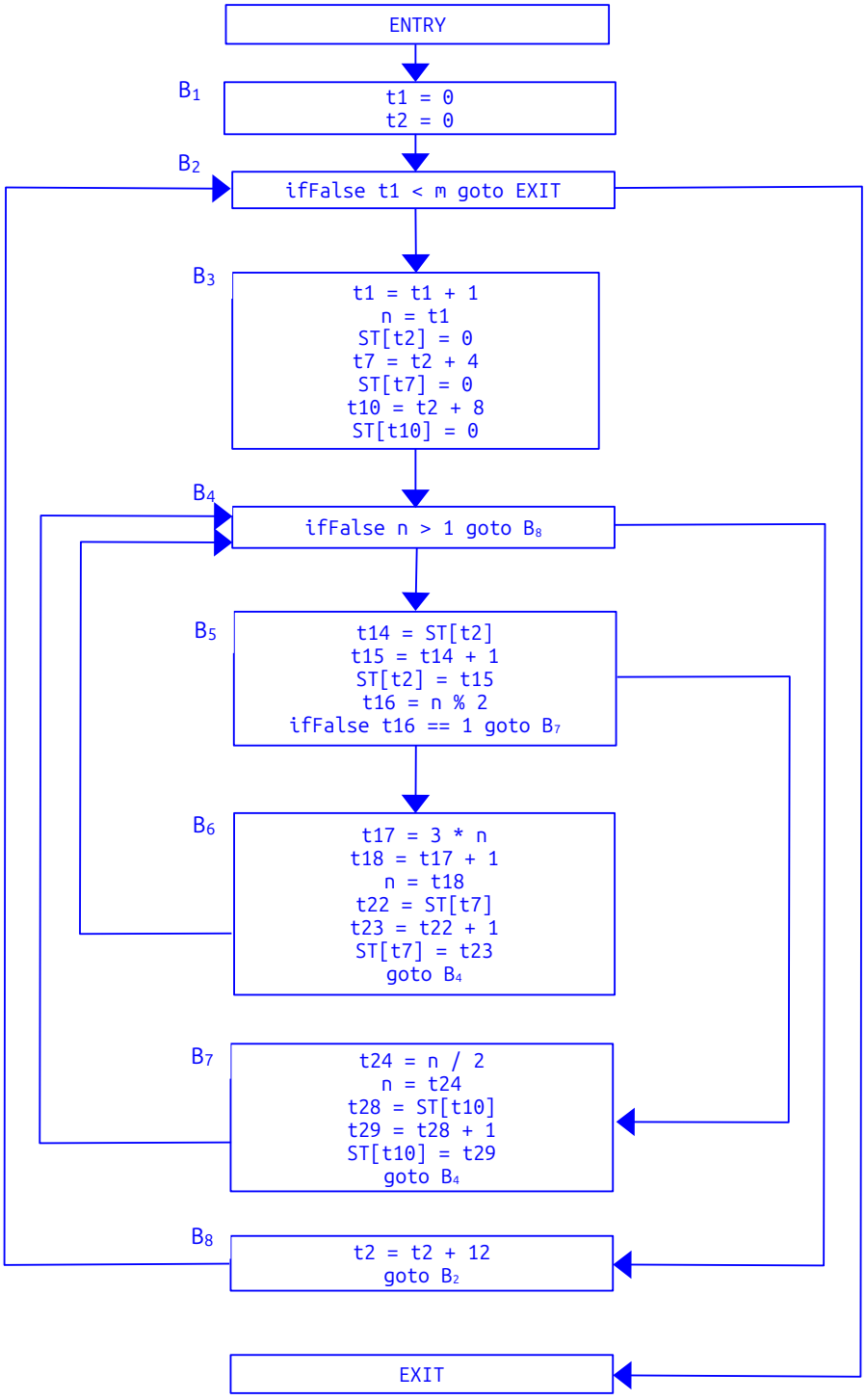
```
L5: t17 = 3 * n
    t18 = t17 + 1
    n = t18
    t19 = i * 12
    t21 = t19 + 4
    t22 = ST[t21]
    t23 = t22 + 1
    ST[t21] = t23
    goto L3
```

Block B_7:
Optimizations Applied:
(i) Constant Folding: `t26 = 2 * 4` → `8`.
(ii) Constant propagation: `t27 = t25 + t26` simplifies to `t27 = t25 + 8`, making the computation of `t26` dead code.
Updated Intermediate Code:

```
L6: t24 = n / 2
    n = t24
    t25 = i * 12
    t27 = t25 + 8
    t28 = ST[t27]
    t29 = t28 + 1
    ST[t27] = t29
    goto L3
```

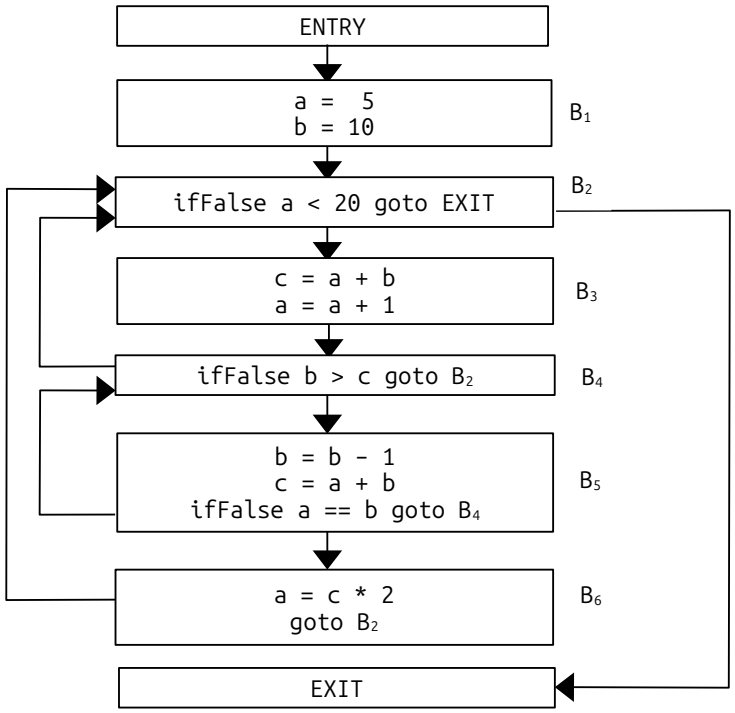
(c) Hand-optimize the code globally by finding global common sub-expressions, by dead-code elimination, and by using induction variables and reduction in strength. Clearly indicate where you use what. Draw the control-flow graph on the optimized basic blocks (your drawing format should be as illustrated in Q5). Write the three-address instructions inside each optimized block. The instructions remaining in the optimized blocks should be in the same sequence as in the original blocks. Retain the old numbering of the temporaries (although some temporaries are not computed, and some use modified formulas). Assume that the variable `i` is not live on exit from the given code snippet. **[10]**

By moving `i * 12` out of the inner loop, we save up to three multiplications per single loop iteration inside the while structure. Since `t2 + 4` is a (global) common subexpression between blocks B_3 and B_6, we can use `t7` for `t21` (and do not compute `t21` at all). Likewise `t2 + 8` is common to B_3 and B_7, so we do not need to compute `t27` and use `t10` for it instead. We can get rid of `i` (not live on exit) as follows. The temporary `t1` always stores `i + 1`, so the loop on `i` may be on `t1`. Moreover, `i + 1` is a common subexpression between B_3 and B_8, so the computation of `t30` can be eliminated altogether. Finally, `t2` (always equal to `12 * i`) is an induction variable. The elimination of `i` requires introducing the necessary changes to `t2`.



5. [*Data-flow analysis*]

Consider the following intermediate code. The basic blocks (denoted by B_i) and the data-flow graph are shown below. You are supposed to perform live variable analysis over this.



(a) Write below the sets $def[B_k]$ and $use[B_k]$ for all basic blocks B_k, where $1 \leq k \leq 6$. **[6]**

$def[B_1] = \{ a, b \}$

$def[B_2] = \varnothing$

$def[B_3] = \{ a, c \}$

$def[B_4] = \varnothing$

$def[B_5] = \{ b, c \}$

$def[B_6] = \{ a \}$

$use[B_1] = \varnothing$

$use[B_2] = \{ a \}$

$use[B_3] = \{ a, b \}$

$use[B_4] = \{ b, c \}$

$use[B_5] = \{ a, b \}$

$use[B_6] = \{ c \}$

(b) For a basic block B_k, present the generic transfer equations for computing $IN[B_k]$ and $OUT[B_k]$ for live variable analysis. **[2]**

$OUT[B_k]$ = Union of $IN[A]$ over all blocks A connected from the output of B_k

$IN[B_k]$ = $use[B_k] \cup (OUT[B_k] - def[B_k])$

(c) In order to compute the live variables, that is, $IN[B_k]$ and $OUT[B_k]$ for all basic blocks B_k ($1 \le k \le 6$), we initialize $OUT[B_k] = \varnothing$ (empty set) for all basic blocks B_k (including ENTRY and EXIT). We then carry out a sequence of iterations for updating $IN[B_k]$ and $OUT[B_k]$ for all basic blocks B_k. Show the iterations. In all iterations, use the update sequence as given below for the first iteration. Use the format { … } instead of bitmaps for writing the sets. Also write the reason/criterion for stopping the updating loop. Show your calculations. **[12]**

Iteration 1

$OUT[B_6] = \varnothing$

$OUT[B_5] = \varnothing$

$OUT[B_4] = \varnothing$

$OUT[B_3] = \varnothing$

$OUT[B_2] = \varnothing$

$OUT[B_1] = \varnothing$

$IN[B_6] = use[B_6] \cup (OUT[B_6] - def[B_6]) = \{ c \} \cup (\varnothing - \{ a \}) = \{ c \}$

$IN[B_5] = use[B_5] \cup (OUT[B_5] - def[B_5]) = \{ a, b \} \cup (\varnothing - \{ b, c \}) = \{ a, b \}$

$IN[B_4] = use[B_4] \cup (OUT[B_4] - def[B_4]) = \{ b, c \} \cup (\varnothing - \varnothing) = \{ b, c \}$

$IN[B_3] = use[B_3] \cup (OUT[B_3] - def[B_3]) = \{ a, b \} \cup (\varnothing - \{ a, c \}) = \{ a, b \}$

$IN[B_2] = use[B_2] \cup (OUT[B_2] - def[B_2]) = \{ a \} \cup (\varnothing - \varnothing) = \{ a \}$

$IN[B_1] = use[B_1] \cup (OUT[B_1] - def[B_1]) = \varnothing \cup (\varnothing - \{ a, b \}) = \varnothing$

Show the remaining iterations on the next page(s).

Iteration 2

$OUT[B_6] = IN[B_2] = \{ a \}$ ← CHANGE

$OUT[B_5] = IN[B_4] \cup IN[B_6] = \{ b, c \} \cup \{ c \} = \{ b, c \}$ ← CHANGE

$OUT[B_4] = IN[B_2] \cup IN[B_5] = \{ a \} \cup \{ a, b \} = \{ a, b \}$ ← CHANGE

$OUT[B_3] = IN[B_4] = \{ b, c \}$ ← CHANGE

$OUT[B_2] = IN[B_3] \cup IN[EXIT] = \{ a, b \} \cup \varnothing = \{ a, b \}$ ← CHANGE

$OUT[B_1] = IN[B_2] = \{ a \}$ ← CHANGE

$IN[B_6] = \{ c \} \cup (\{ a \} - \{ a \}) = \{ c \}$ ← STABLE

$IN[B_5] = \{ a, b \} \cup (\{ b, c \} - \{ b, c \}) = \{ a, b \}$ ← STABLE

$IN[B_4] = \{ b, c \} \cup (\{ a, b \} - \varnothing) = \{ a, b, c \}$ ← CHANGE

$IN[B_3] = \{ a, b \} \cup (\{ b, c \} - \{ a, c \}) = \{ a, b \}$ ← STABLE

$IN[B_2] = \{ a \} \cup (\{ a, b \} - \varnothing) = \{ a, b \}$ ← CHANGE

$IN[B_1] = \varnothing \cup (\{ a \} - \{ a, b \}) = \varnothing$ ← STABLE

Iteration 3

$$
\begin{aligned}
OUT[B_6] &= \{ a, b \} && \leftarrow \text{CHANGE} \\
OUT[B_5] &= \{ a, b, c \} \cup \{ c \} = \{ a, b, c \} && \leftarrow \text{CHANGE} \\
OUT[B_4] &= \{ a, b \} \cup \{ a, b \} = \{ a, b \} && \leftarrow \text{STABLE} \\
OUT[B_3] &= \{ a, b, c \} && \leftarrow \text{CHANGE} \\
OUT[B_2] &= \{ a, b \} \cup \varnothing = \{ a, b \} && \leftarrow \text{STABLE} \\
OUT[B_1] &= \{ a, b \} && \leftarrow \text{CHANGE}
\end{aligned}
$$

$$
\begin{aligned}
IN[B_6] &= \{ c \} \cup (\{ a, b \} - \{ a \}) = \{ b, c \} && \leftarrow \text{CHANGE} \\
IN[B_5] &= \{ a, b \} \cup (\{ a, b, c \} - \{ b, c \}) = \{ a, b \} && \leftarrow \text{STABLE} \\
IN[B_4] &= \{ b, c \} \cup (\{ a, b \} - \varnothing) = \{ a, b, c \} && \leftarrow \text{STABLE} \\
IN[B_3] &= \{ a, b \} \cup (\{ a, b, c \} - \{ a, c \}) = \{ a, b \} && \leftarrow \text{STABLE} \\
IN[B_2] &= \{ a \} \cup (\{ a, b \} - \varnothing) = \{ a, b \} && \leftarrow \text{STABLE} \\
IN[B_1] &= \varnothing \cup (\{ a, b \} - \{ a, b \}) = \varnothing && \leftarrow \text{STABLE}
\end{aligned}
$$

Iteration 4

$$
\begin{aligned}
OUT[B_6] &= \{ a, b \} && \leftarrow \text{STABLE} \\
OUT[B_5] &= \{ a, b, c \} \cup \{ b, c \} = \{ a, b, c \} && \leftarrow \text{STABLE} \\
OUT[B_4] &= \{ a, b \} \cup \{ a, b \} = \{ a, b \} && \leftarrow \text{STABLE} \\
OUT[B_3] &= \{ a, b, c \} && \leftarrow \text{STABLE} \\
OUT[B_2] &= \{ a, b \} \cup \varnothing = \{ a, b \} && \leftarrow \text{STABLE} \\
OUT[B_1] &= \{ a, b \} && \leftarrow \text{STABLE}
\end{aligned}
$$

$$
\begin{aligned}
IN[B_6] &= \{ c \} \cup (\{ a, b \} - \{ a \}) = \{ b, c \} && \leftarrow \text{STABLE} \\
IN[B_5] &= \{ a, b \} \cup (\{ a, b, c \} - \{ b, c \}) = \{ a, b \} && \leftarrow \text{STABLE} \\
IN[B_4] &= \{ b, c \} \cup (\{ a, b \} - \varnothing) = \{ a, b, c \} && \leftarrow \text{STABLE} \\
IN[B_3] &= \{ a, b \} \cup (\{ a, b, c \} - \{ a, c \}) = \{ a, b \} && \leftarrow \text{STABLE} \\
IN[B_2] &= \{ a \} \cup (\{ a, b \} - \varnothing) = \{ a, b \} && \leftarrow \text{STABLE} \\
IN[B_1] &= \varnothing \cup (\{ a, b \} - \{ a, b \}) = \varnothing && \leftarrow \text{STABLE}
\end{aligned}
$$

We **STOP** as there are no more updates/changes to any of $IN[B_k]$ or $OUT[B_k]$ for all $1 \leq k \leq 6$.

Note: Other correct variations are given due credit.