CSE 401/M501 24sp Final Exam 6/4/24 Sample Solution

Question 1. (15 points) A bit of x86-64 coding. Here is a small C function that returns the cube of the absolute value of an element an integer array.

```
/* return cube of the absolute value of a[i] */
int absEltCubed (int a[], int i) {
  int val;
  val = a[i];
  if (val < 0) {
    val = -val;
  }
  return cube(val);
}
```

You should assume that the function `cube(n)` already exists and that it return the value *n***n***n*.

On the next page, translate this function into x86-64 assembly language. You should use the standard x86-64 runtime conventions for parameter passing, register usage, and so forth that we used in the MiniJava project, including using `%rbp` as a stack frame pointer in the function prologue code. Note that this is simple C code, not a Java method, so there is no `this` pointer or method vtable involved.

Reference and ground rules for x86-64 code, (same as for the MiniJava project and other x86-64 code):

- All values, including pointers and `int`s, are 64 bits (8 bytes) each, as in MiniJava
- You must use the Linux/gcc assembly language, and must follow the x86-64 function call, register, and stack frame conventions:
  - Argument registers: `%rdi, %rsi, %rdx, %rcx, %r8, %r9` in that order
  - Called function must save and restore `%rbx, %rbp,` and `%r12-%r15` if these are used in the function
  - Function result returned in `%rax`
  - `%rsp` must be aligned on a 16-byte boundary when a `call` instruction is executed
  - `%rbp` must be used as the base pointer (frame pointer) register for this question
- The full form of a memory address is *constant*(*%rbase*,*%rindex*,*scalefactor*), which references memory address *%rbase*+*%rindex***scalefactor*+*constant*. *scalefactor* must be 0, 2, 4, or 8.
- Your x86-64 code must implement all of the statements in the original function including allocating space for and using the local variable `val`, specifically updating the memory location allocated for `val` every time `val` is updated in the original code. You may *not* rewrite the code into a different form that produces equivalent results (i.e., restructuring or reordering the code or eliminating function calls). Other than that, you can use any reasonable x86-64 code that follows the standard function call and register conventions – you do not need to mimic the code produced by your MiniJava compiler.
- Please include *brief* comments in your code to help us understand what the code is supposed to be doing (which will help us assign partial credit if it doesn't do exactly what you intended.)

Question 1. (cont.) Write your x86-64 translation of function `absEltCubed` below. Remember to read and follow the above ground rules carefully, including managing registers properly and using the correct argument registers for function calls, and creating a local stack frame to hold `val`. Your code should include a translation of all of the code in the original function. Brief comments are appreciated. Original code repeated below for convenience:

```
/* return cube of the absolute value of a[i] */
int absEltCubed(int a[], int i) {
  int val;
  val = a[i];
  if (val < 0) {
    val = -val;
  }
  return cube(val);
}
```

```
absEltCubed:
        pushq   %rbp            ; function prologue
        movq    %rsp,%rbp
        subq    $16,%rsp        ; allocate stack frame
        movq    0(%rdi,%rsi,8),%rax   ; load a[i]
        movq    %rax,-8(%rbp)   ; store in val
        cmpq    $0,%rax         ; jump if a[i]-0 >= 0
        jge     else
        negq    %rax            ; set val = -val
        movq    %rax,-8(%rbp)
else:
        movq    -8(%rbp),%rdi   ; call cube(val)
        call    cube            ; function result is in %rax
        movq    %rbp,%rsp       ; unwind stack and return
        popq    %rbp
        ret
```

Notes: The code to load `a[i]` from memory could add `8*%rsi` to `%rdi` to compute the address of `a[i]` instead of using the indexed addressing mode shown here. It would be possible to use `tstq` to test whether `a[i]` is negative instead of `cmpq`. There are many other details that could be different and still yield a correct assembly language program.

Question 2. (24 points) Compiler hacking. Some of our customers have complained that it's hard to debug MiniJava programs. To help them we would like to add an `assert` statement to MiniJava. Our new `assert` statement will be basically the same as in languages like C and Java, except that when an assertion fails, we will not be able to print a string message describing the failure because MiniJava does not contain string values. Instead, we will include an integer code in the new `assert` statement to be printed if the assert fails.

The syntax of the new `assert` statement is

`assert` *exp1* `:` *exp2* `;`

The meaning of this new statement is first evaluate *exp1*. If *exp1* evaluates to true, then execution continues after the assert statement and *exp2* is not evaluated. If *exp1* evaluates to false, then *exp2* is evaluated, a message that includes the value of the integer *exp2* is printed saying that the `assert` failed, and execution of the MiniJava program is terminated.

You should assume that the following function has been added to `boot.c`. The compiled code for `assert` should call this function when an assertion fails and use it to print a message that includes the value of *exp2* from the `assert` statement that failed, and then terminate the program.

```
/* Print a message reporting a failed assertion with the given
 * integer msg code, then terminate the program */
void assertFailed(int64_t msg) {
  fprintf(stderr, "assert failed; msg = %" PRId64 "\n", msg);
  exit(1);
}
```

Answer the questions below about how this `assert` statement would be added to a MiniJava compiler. There is likely way more space than you will need for some of the answers. The full MiniJava grammar is attached at the end of the exam if you need to refer to it.

(a) (2 points) What new lexical tokens, if any, need to be added to the scanner and parser of our MiniJava compiler to add this new `assert` statement to the original MiniJava language? Just describe any necessary changes and new token(s) needed and their name(s). You don't need to give JFlex or CUP specifications or code in this part of the question, but you will need to use any token name(s) you write here in a later part of this question.

We need two new tokens: ASSERT for the new keyword, and COLON for the : character.

(continued on next page)

Question 2. (cont.) (b) (6 points) Complete the following new AST class to define an AST node type for this new `assert` statement. You only need to define instance variables and the constructor. Assume that all appropriate package and import declarations are supplied, and don't worry about visitor code.

(Hint: recall that the AST package in MiniJava contains the following key classes: `ASTNode`, `Exp extends ASTNode`, and `Statement extends ASTNode`. Also remember that each AST node constructor has a `Location` parameter, and the supplied `super(pos);` statement at the beginning of the constructor below is used to properly initialize the superclass with this information.)

```
public class Assert extends Statement {
  // add any needed instance variables below
```

```
    Exp e1;
    Exp e2;
```

```
  // constructor - add parameters and method body below

  public Assert (   Exp e1, Exp e2, Location pos              ){
```

```
    super(pos);   // initialize location information in superclass
    this.e1 = e1;
    this.e2 = e2;
```

```
  }
}
```

(continued on next page)

Question 2. (cont.) (c) (5 points) Complete the CUP specification below to define a production for this new `assert` statement, including associated semantic action(s) needed to parse the new expression and create an appropriate AST node (as defined in part (b) above). You should use any new lexical tokens defined in your answer to part (a) as needed. Use reasonable names for any other lexical tokens that already would exist in the compiler scanner and parser if you need them. We have added additional code to the parser rule for `Statement` below so the CUP specification for the `assert` statement can be written as an independent grammar rule with separate semantic actions.

Hint: recall that the `Location` of an item `foo` in a CUP grammar production can be referenced as `fooxleft`.

```
Statement ::= ...
      | AssertStmt:s  {: RESULT = s; :}
     ...
      ;
AssertStmt ::=
```

```
ASSERT:a Expression:e1 COLON Expression:e2 SEMICOLON
   {: RESULT = new Assert(e1, e2, axleft); :}


  /* any location from a, e1, or e2 could be used
     for the third parameter */
```

(d) (4 points) Describe the checks that would be needed in the semantics/type-checking part of the compiler to verify that a program containing this new `assert` statement is legal. You do not need to give code for a visitor method or anything like that – just describe what language rules (if any) need to be checked for this new statement to verify it is used correctly.

Need to verify:

- **exp1 has type Boolean**
- **exp2 has type int**

(continued on next page)

Question 2. (cont.) (e) (7 points) Describe the x86-64 code shape for this new `assert` statement that would be generated by a MiniJava compiler. Your answer should be similar in format to the descriptions we used in class for other language constructs. If needed, you should assume that the code generated for an expression will leave the value of that expression in `%rax`, as in our MiniJava project.

Reminder: the generated code for `assert` *exp1* `:` *exp2* `;` should call this new `boot.c` function to print the value of *exp2* and terminate the program if *exp1* is false.

```
/* Print a message reporting a failed assertion with the given
 * integer msg code, then terminate the program */
void assertFailed(int64_t msg) {
  fprintf(stderr, "assert failed; msg = %" PRId64 "\n", msg);
  exit(1);
}
```

Use Linux/gcc x86-64 instructions and assembler syntax when needed. If you need to make any additional assumptions about code generated by the rest of the compiler you should state them. Reminder: The argument registers for a function call are `%rdi,%rsi,%rdx,%rcx,%r8,` and `%r9` in that order.

Hint: be sure that your code follows the described operation and semantics of the new `assert` statement precisely, including evaluating *exp2* only when needed.

```
        Generate code to evaluate exp1 and leave the result (0 false or 1 true) in %rax
        cmpq    1,%rax          ; check exp1 and branch if true
        je      done
        Generate code to evaluate exp2 and leave the result in %rax
        movq    %rax,%rdi       ; call assertFailed(exp2)
        call    assertFailed
done:                           ; resume execution here if assert succeeds
```

Note: Other solutions, particularly ones that evaluate exp1 and generate a direct "jump if true" instead of storing a 0 or 1 in %rax would also be correct. There are also other correct ways of writing the code, but one important thing is to ensure that exp2 is not evaluated if exp1 is true.

Interlude. A few short-answer questions before heading on to control flow graphs and other things.

Question 3. (12 points) Compiler phases. For each of the following situations, indicate where the situation would normally be discovered or handled in a production compiler. Assume that the compiler is a conventional one that generates native code for a single target machine (say, x86-64), and assume that the source language is standard Java (if it matters). Use the following abbreviations for the stages:

scan – scanner
parse – parser
sem – semantics/type check
opt – optimization (dataflow/ssa analysis; code transformations)
instr – instruction selection & scheduling
reg – register allocation
run – runtime (i.e., when the compiled code is executed)
can't – can't always be done during either compilation or execution

__run__ Detect integer division by 0 and terminate the program

__sem__ Report that the + operator cannot be used with operand expressions of type boolean

parse Report an error when a reserved keyword like `class` or `if` is used as an identifier in a variable declaration (e.g. `int class = 0;`)

__opt__ Evaluate expressions with constant values and replace them with the computed results in the generated code

__scan__ Report an unclosed string literal (e.g. `"this is an unclosed string`)

__sem__ Report that an appropriate superclass constructor is not available and cannot be called by `super(...)` in a subclass constructor.

__reg__ Decide which values to store in registers and which ones to spill to memory at a particular point in the program

__run__ Report that a method call causes a stack overflow

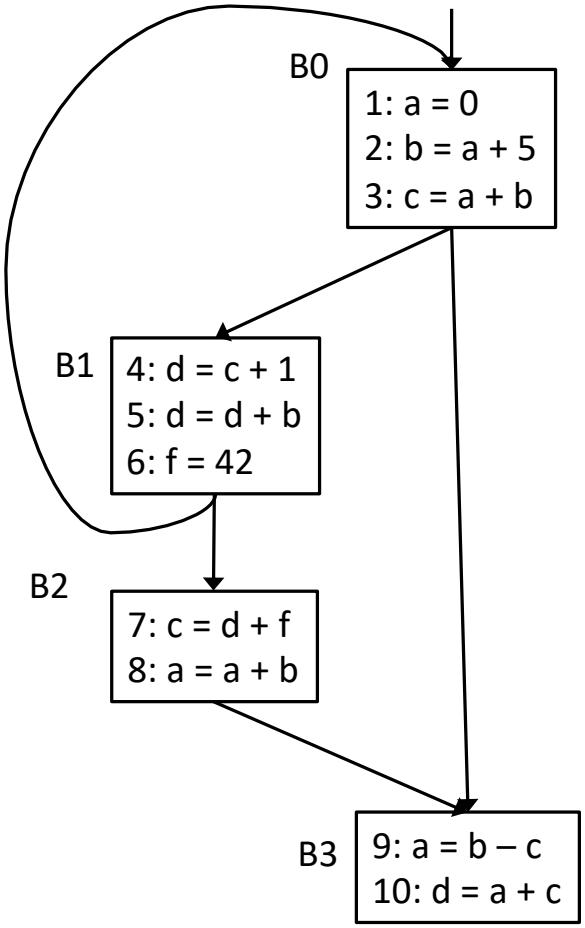
__sem__ Report that a non-abstract class implementing an interface does not provide implementations for all methods declared in the interface

__instr__ Reordering LOAD instructions to minimize execution latency

__instr__ Use a LEAQ instruction to compute an integer expression of the form x+4*y

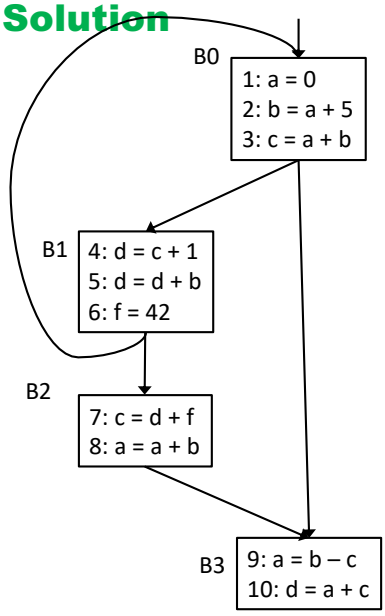
__opt__ Report that a particular variable might be uninitialized when it is used in the program

The next two questions concern the following control flow graph:



Each of the assignment statements in this flow graph defines a variable like b = 0. Each assignment statement is preceded by a number since we will be interested in a dataflow problem to discover which assignment statements reach other points in the program (precise description below). So, for instance, in the first block, the entry 2: b = a+5 means that the assignment b = a+5 is assignment statement number 2 in the program. Each assignment statement in the entire program has a unique number.

(continued on next page)



Question 4. (18 points) Dataflow analysis – reaching definitions. A useful analysis during code optimization is to examine a program to find all the places in the program where values are assigned to a particular variable, and then discover where in the program the values stored by those assignments to that variable could be used. In particular, if we have a statement x = y + 1, we would like to discover all the assignment statements in the program that could have stored the value in variable y that is used in this statement. Given a statement that stores a value in y, as long as there is at least one path in the control flow graph from that statement to the x = y + 1 statement, and as long as no other statement along that particular path changes the value of y, we say that this statement that stores the value in y *reaches* the point in the program where we read the value of y in the expression y+1. Of course, there might be many places in the program that store a value in *y* that also reach the x = y + 1 statement, so the set of reaching definitions could include several statements that could set the value of *y* that is used to compute *y*+1, and those statements could store different values in *y*.

We would like to use dataflow analysis to discover all of the *reaching definitions* in the program. More precisely, we would like to know which assignment statements reach the beginning of each basic block along some possible path. (Note that there might be other paths to the same basic block that alter the value of the same variable, but as long as there is at least one path from an assignment to the start of the basic block that does not contain other assignments to that variable, we say that the assignment (definition) reaches the block because there is at least one path where it could do so.)

To solve this problem, we define the following sets for each basic block *b*:

DEFOUT(*b*) – the set of variable definitions *v* = *value* in *b* that reach the end of *b* without variable *v* being subsequently redefined later in block *b*.

SURVIVED(*b*) – the set of all variable definition statements (assignments) in the entire program that are not obscured by an assignment in *b* to the same variable when *b* is executed.

REACHES(*b*) – the set of variable definitions (assignment statements) that reach the beginning of block *b*.

These sets are related by the following equation for each block *b*:

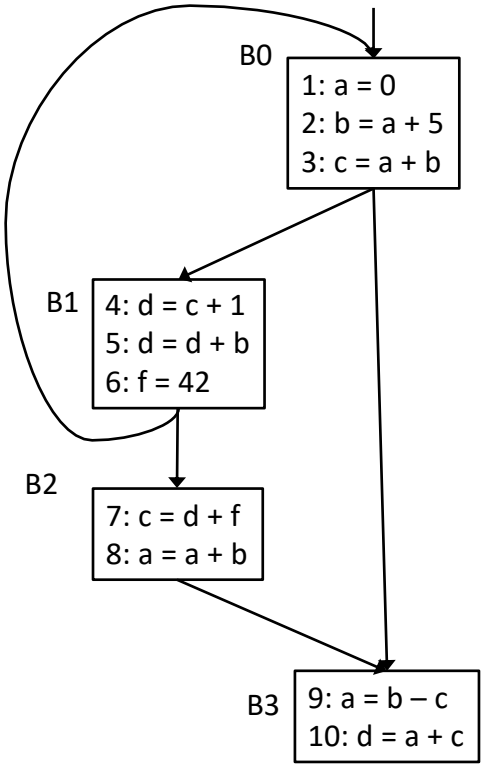
$$\text{REACHES}(b) = \bigcup_{p \in \text{predecessors}(b)} (\text{DEFOUT}(p) \cup (\text{REACHES}(p) \cap \text{SURVIVED}(p)))$$

For this problem, each of these sets will be a collection of statement numbers, possibly empty, from the flow graph shown above and on the previous page.

Note that DEFOUT(*b*) and SURVIVED(*b*) depend only on information in a single block *b*, so they can be computed once for each block before doing an iterative solution to compute the REACHES(*b*) sets for all blocks.

(continued on next page)

Question 4. (cont) Complete the following table to solve the reaching definitions problem for the flowgraph. You should fill in the DEFOUT and SURVIVED values for each block, then use one or more iterations to compute the REACHES sets for each block until no changes occur after updating the REACHES sets.



For full credit (and to help award partial credit if needed) **you must show your work**, i.e., the contents of all sets, including the REACHES sets for each iteration of the algorithm until no further changes occur. You can write "same" in a REACHES column if the entry is the same as in the previous column.

Sets and equations repeated for convenience:

DEFOUT(*b*) – the set of variable definitions v = value in *b* that reach the end of *b* without variable v being subsequently redefined later in block *b*.

SURVIVED(*b*) – the set of all variable definition statements (assignments) in the entire program that are not obscured by an assignment in *b* to the same variable when *b* is executed.

REACHES(*b*) – the set of variable definitions (assignments) that reach the beginning of block *b*.

$$\text{REACHES}(b) = \bigcup_{p \in \text{predecessors}(b)} (\text{DEFOUT}(p) \cup (\text{REACHES}(p) \cap \text{SURVIVED}(p)))$$

The DEFOUT (but not SURVIVED or REACHES) set for block B0 is provided to help you get started.

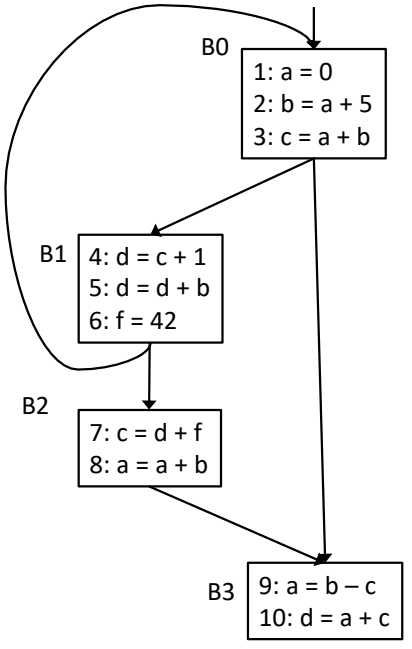
	DEFOUT	SURVIVED	REACHES (1)	REACHES (2)	REACHES (3)	REACHES(4)
B0	1, 2, 3	**4, 5, 6, 10**	**5, 6**	**1, 2, 3, 5, 6**	**Same**	
B1	**5, 6**	**1, 2, 3, 7, 8, 9**	**1, 2, 3, 5, 6**	**1, 2, 3, 5, 6**	**Same**	
B2	**7, 8**	**2, 4, 5, 6, 10**	**1, 2, 3, 5, 6**	**1, 2, 3, 5, 6**	**Same**	
B3	**9, 10**	**2, 3, 6, 7**	**1, 2, 3, 5, 6, 7, 8**	**1, 2, 3, 5, 6, 7, 8**	**Same**	

Question 5. (18 points) Dominators and SSA. Here are the basic definitions of dominators and related concepts we have seen previously in class:

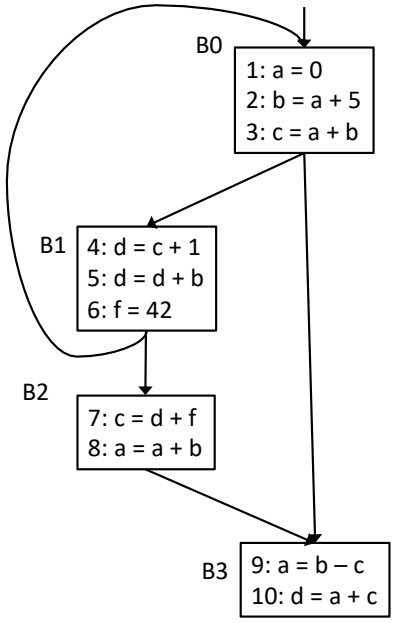
- Every control flow graph has a unique **start node** s.
- Node *x* **dominates** node *y* if every path from s to *y* must go through x.
  - A node *x* dominates itself.
- A node *x* **strictly dominates** node *y* if *x* dominates *y* and $x \neq y$.
- The **dominator set** of a node *x* is the set of nodes *dominated by x*.
  - | Dom(x) | ≥ 1
  - (note: sometimes the definition of Dom(x) is given as the set of all nodes that dominate *x*. For SSA it is more convenient to keep track of the set of nodes that *x* dominates.)
- An **immediate dominator** of a node *y*, idom(*y*), has the following properties:
  - idom(*y*) strictly dominates *y* (i.e., dominates *y* but is different from *y*)
  - idom(*y*) does not dominate any other strict dominator of *y*
- The **dominator tree** of a control flow graph is a tree where there is an edge from every node *x* to its immediate dominator idom(*x*).
- The **dominance frontier** of a node *x* is the set of all nodes *w* such that
  - *x* dominates a predecessor of *w*, but
  - *x* does not strictly dominate *w*

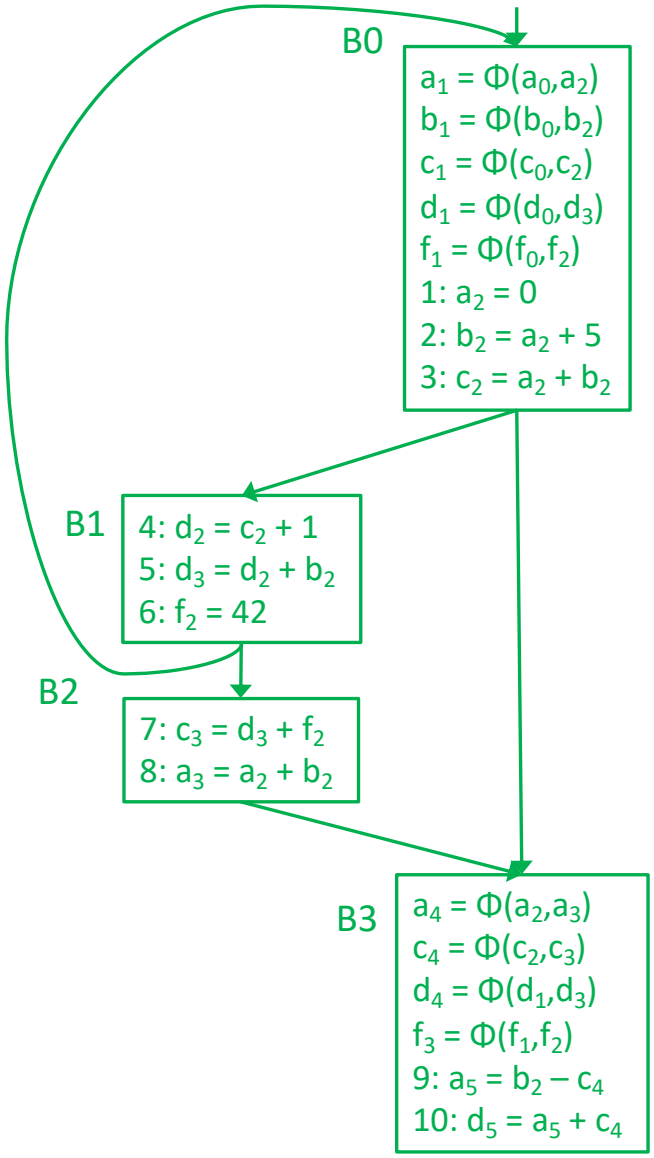
(a) (8 points) Using the same control flow graph from the previous problem, complete the following table. List for each node: the node(s) that it dominates, successor nodes to the dominated nodes, and the nodes that are in its dominance frontier (if any):

Node	Nodes dominated by this node	Successor(s) to nodes dominated by this node	Dominance Frontier of this node
B0	B0, B1, B2, B3	B0, B1, B2, B3	B0
B1	B1, B2	B0, B2, B3	B0, B3
B2	B2	B3	B3
B3	B3	--	--



Question 5. (cont.) (b) (10 points) Now redraw the flowgraph in SSA (static single-assignment) form. You need to insert all Φ-functions that are required by the dominance frontier criteria, even if some of the variables created by those functions are not used later. Once that is done, add appropriate version numbers to all variables that are assigned in the flowgraph. You do not need to trace the steps of any particular algorithm to place the Φ-functions as long as you add them to the flowgraph in appropriate places. Answers that have a couple of extraneous Φ-functions will receive appropriate partial credit, but answers that, for example, use a maximal-SSA strategy of placing Φ-functions for all variables at the beginning of every block will not be looked on with favor.





The next two questions concern register allocation and instructions scheduling. For both of these questions, assume that we're using the same hypothetical machine that was presented in lecture and in the textbook examples for list scheduling.

The instructions on this example machine are assumed to take the following numbers of cycles each:

Operation	Cycles
LOAD	3
STORE	3
ADD	1
MULT	2
SHIFT	1

Our instruction selection algorithm has been modified so it does not re-use registers, but instead just creates temporaries and leaves register selection for later. Here is a sequence of instructions produced by the code generator for the statement `x = 3 * (a + b) + c[1]`. We assume that the elements of `c` each occupy 8 bytes. We also assume that one of the operands of a MULT or ADD instruction can be an integer constant.

```
a.  LOAD   t1 <- a              // t1 = a
b.  LOAD   t2 <- b              // t2 = b
c.  ADD    t3 <- t1, t2         // t3 = a + b
d.  MULT   t4 <- t3, 3          // t4 = 3 * (a + b)
e.  LOAD   t5 <- c              // t5 = pointer to array c
f.  ADD    t6 <- t5, 8          // t6 = pointer to c[1]
g.  LOAD   t7 <- MEM[t6]        // t7 = value of c[1]
h.  ADD    t8 <- t4, t7         // t8 = (3 * (a + b)) + c[1]
i.  STORE  x <- t8              // x = (3 * (a + b)) + c[1]
```

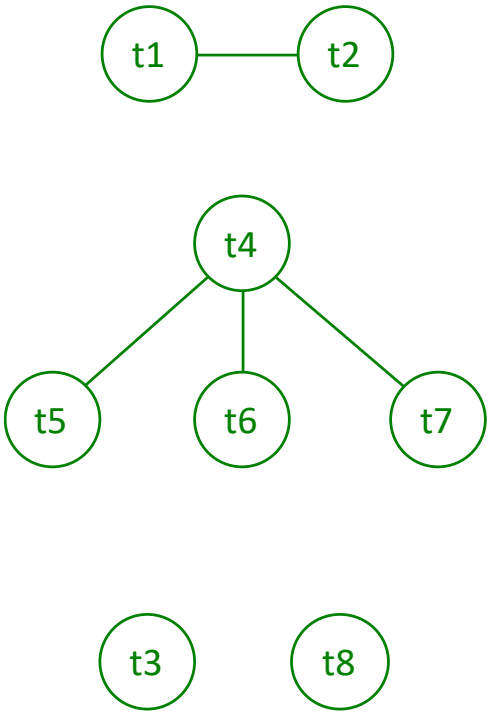
In a real compiler we would first use list scheduling to pick a (possibly) better order for the instructions, then use graph coloring to assign temporaries (t1-t8) to actual registers. But for this exam we're going to ask those two questions separately so the answers don't depend on each other, which will make it much easier to assign credit fairly (☺).

Answer the questions about this sequence of code on the next pages.

Question 6. (15 points) Register allocation/graph coloring.

(a) (9 points) Draw the interference graph for the temporary variables (t1-t8) in the code on the previous page (repeated here for convenience). You should assume that the code is executed in the sequence given and not rearranged before assigning registers.

```
a.  LOAD   t1 <- a
b.  LOAD   t2 <- b
c.  ADD    t3 <- t1, t2
d.  MULT   t4 <- t3, 3
e.  LOAD   t5 <- c
f.  ADD    t6 <- t5, 8
g.  LOAD   t7 <- MEM[t6]
h.  ADD    t8 <- t4, t7
i.  STORE  x <- t8
```



(b) (6 points) Give an assignment of groups of temporary variables to registers that uses the minimum number of registers possible based on the information in the interference graph. Use R1, R2, R3, … for the register names.

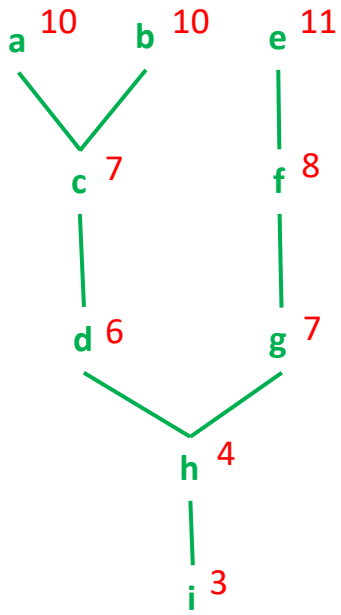
Two registers are needed. Here are two possible assignments, there are many others that obey the constraints of the interference graph:

R1: t1, t4
R2: t2, t3, t5, t6, t7, t8

R1: t1, t3, t4, t8
R2: t2, t5, t6, t7

Question 7. (16 points) Forward list scheduling. (a) (8 points) Given the original sequence of instructions shown below, draw the precedence graph showing the dependencies between these instructions. Label each node (instruction) in the graph with the letter identifying the instruction (a-i) and its latency – the number of cycles between the beginning of that instruction and the end of the graph on the shortest possible path that respects the dependencies.

```
a.  LOAD   t1 <- a
b.  LOAD   t2 <- b
c.  ADD    t3 <- t1, t2
d.  MULT   t4 <- t3, 3
e.  LOAD   t5 <- c
f.  ADD    t6 <- t5, 8
g.  LOAD   t7 <- MEM[t6]
h.  ADD    t8 <- t4, t7
i.  STORE  x <- t8
```



Operation	Cycles
LOAD	3
STORE	3
ADD	1
MULT	2
SHIFT	1

(continued on next page)

Question 7. (cont) (b) (8 points) Rewrite the instructions in the order they would be chosen by forward list scheduling. If there is a tie at any step when picking the best instruction to schedule next, pick one of them arbitrarily. Label each instruction with its letter and instruction operation (LOAD, ADD, etc.) from the original sequence above and the cycle number on which it begins execution (you do not need to write all of the instruction operands, just the letter and opcode). The first instruction begins on cycle 1. You do not need to show your bookkeeping or trace the algorithm as done in class, although if you leave these clues about what you did, it could be helpful if we need to figure out how to assign partial credit.

1. **e LOAD**
2. **a LOAD**
3. **b LOAD**
4. **f ADD**
5. **g LOAD**
6. **c ADD**
7. **d MULT**
8. **---**
9. **H ADD**
10. **I STORE**

The two load instructions a and b on cycles 2 and 3 could be swapped with b on cycle 2 and a on cycle 3. All of the other operations must be listed in the given order on the specified cycles.

Question 8. (2 free points – all answers get the free points)

Draw a picture of something you are planning to this summer!
– or –
Draw a picture of something you think one or more of your TAs will do this summer!

Have a great summer and best wishes for the future!
The CSE 401/M501 staff