CSE 401/M501 22au Final Exam 12/13/22 Sample Solution

Question 1. (20 points) A bit of x86-64 coding. Here is a small C function that has two integer parameters and returns -1, 0, or +1 depending on whether the first parameter is less than, equal to, or greater than the second.

```
int comp(int x, int y) {
  int ans;
  if (x < y)
    ans = -1;
  else if (x == y)
    ans = 0;
  else // x > y
    ans = 1;
  return ans;
}
```

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. 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). You should allocate space for the local variable `ans` and generate appropriate load and store instructions when needed so that the copy in memory always has the correct current value of that variable at the end of the x86-64 code for each statement in the original C function. (And, of course, you can allocate additional space in the stack frame as needed to store other data.). 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 `comp` below. Remember to read and follow the above ground rules carefully, including managing registers properly, allocating space for the local variable `ans`, and storing correct values in `ans` when it is assigned a value in the code. Brief comments are appreciated. Original code repeated below for convenience:

```
int comp(int x, int y) {
  int ans;
  if (x < y)
    ans = -1;
  else if (x == y)
    ans = 0;
  else // x > y
    ans = 1;
  return ans;
}
```

```
# on entry: %rdi = x, %rsi = y

comp:   pushq   %rbp            # function prologue
        movq    %rsp,%rbp       # save %rbp and allocate stack frame
        subq    $16,%rsp        #   (size should be multiple of 16)

        cmpq    %rsi,%rdi       # set cond codes by subtracting x-y
        jge     geq             # jump if x >= y
        movq    $-1,-8(%rbp)    # x < y so set ans = -1
        jmp     done

        # jump here if x >= y
geq:    jne     gt              # jump if x != y
        movq    $0,-8(%rbp)     # x == y so set ans = 0
        jmp     done

        # jump here if x > y
gt:     movq    $1,-8(%rbp)     # x > y so set ans = 1

        # exit with result in %rax
done:   movq    -8(%rbp),%rax   # load ans into result register
        movq    %rbp,%rsp       # free stack frame
        popq    %rbp            # restore caller's %rbp
        ret                     # return to caller
```

Notes: there are obviously many ways to write the code, including having multiple conditional jumps after the `cmpq` instruction to branch to the right code to store the result value. The problem did require, however, allocation of a stack frame for the local variable `ans` and storing values there.

Question 2. (24 points) Compiler hacking. As always seems to happen, now that our MiniJava compiler is done, one of our customers would like us to add a new "feature". This time our customer got tired of typing `if`-statements and wants us to add a new "max" expression to the language. To do that, we'll add the following new production for Expression to the existing ones in the grammar:

Expression ::= "max" "(" Expression "," Expression ")"

This new expression has the meaning we would expect: `max(`*e1*`,`*e2*`)` evaluates both expressions *e1* and *e2*, and then the larger value of *e1* or *e2* is the value of the `max` expression. Expressions *e1* and *e2* and the resulting value all have type `int`.

Answer the questions below about how this new `max` expression 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 kind of expression to the original MiniJava language? Just describe any necessary changes and new token name(s) needed. You don't need to give JFlex or CUP specifications or code, but you will need to use any token name(s) you write here in a later part of this question.

Only one new token needed: MAX for the new "max" keyword.

Note: Since "max' is quoted in the grammar rule, we follow the MiniJava conventions of treating it as a reserved keyword.

(continued on next page)

Question 2. (cont.) (b) (5 points) Complete the following new AST class to define an AST node type for the new `max` expression. 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 `Max` constructor below is used to properly initialize the superclass with this information.)

```
public class Max extends Exp {
  // add any needed instance variables below
```

```
    public Exp e1, e2;    // left and right operands
```

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

  public Max ( 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 the new `max` expression including associated semantic action(s) needed to parse an expression containing `max` and create an appropriate `Max` 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 needed that already would exist in the compiler scanner and parser such as COMMA. We have added additional code to the parser rule for `Exp` below so the CUP specification for `max` 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`.

```
Exp ::= ...
      | Max:e  {: RESULT = e; :}
     ...
      ;
Max ::=
```

```
MAX  LPAREN  Exp:e1  COMMA  Exp:e2 RPAREN
      {: RESULT = new Max(e1, e2, e1xleft); :}
```

Note: positions from any of the tokens could have been used for the position argument in the `new Max(...)` object creation call.

(d) (4 points) Describe the checks that would be needed in the semantics/type-checking part of the compiler to verify that an expression containing `max` 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 and any type information that needs to be produced for this expression.

Verify that *e1* and *e2* have type `int`.

Set the type of the `Max` expression node to `int`.

(continued on next page)

Question 2. (cont.) (e) (8 points) Describe the x86-64 code shape for this added `max` expression 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 we did in the MiniJava project.

Use Linux/gcc x86-64 instructions and assembler syntax when needed. However, remember that the question is asking for the code shape for this expression, so using things like `Jfalse`, for example, to indicate control flow, instead of pure x86-64 machine instructions, is fine as long as the meaning is clear. If you need to make any additional assumptions about code generated by the rest of the compiler you should state them.

Assume that we are evaluating max(exp1, exp2):

```
        <code to evaluate exp1 into %rax>

        pushq %rax          (save exp1 value on the stack temporarily)

        <code to evaluate exp2 into %rax>

        popq %rdx           (reload exp1 value)

        cmpq %rdx,%rax      (evaluate exp2-exp1 – if >= 0, exp2 is larger and already in %rax)

        jge  done           (jump if exp2 result in %rax is larger of the two)

        movq  %rdx,%rax     (move larger exp1 value into result register)

done:
```

There are, of course, other possible ways to do this. This solution closely follows the code shape examples from class, but other reasonable and correct solutions received full credit.

Question 3. (8 points) vtables. Suppose we have the following three classes in a MiniJava program.

class Base {	class Derived extends Base {	class Sub extends Derived {
int one(int n) { return n; }	int two() {return 0;}	int fun(int n) { return 1; }
int alpha(int n) { return n; }	int one(int n) { return 0; }	int alpha(int n) { return 1; }
}	int beta() { return 0;}	int beta() { return -1; }
	}	}

(Obviously, the values returned by the methods are of no significance – they were just included above to be sure the code was syntactically correct.)

When class `Base` was compiled, the compiler picked the following vtable layout for the class:

	Vtable layout	offset
`Base$$:`	`.quad 0`	# no superclass
	`.quad Base$alpha`	# +8
	`.quad Base$one`	# +16

Below, show appropriate vtable layouts for classes `Derived` and `Sub`, in the same format used above for class `Base`. Be sure to properly account for inherited methods in the vtable layouts. There is at least one correct solution. If there are different possible solutions, any one is acceptable.

Here is one solution:

	Vtable layout	offset
`Derived$$:`	**`.quad Base$$`**	**# +0 superclass**
	`.quad Base$alpha`	**# +8**
	`.quad Derived$one`	**# +16**
	`.quad Derived$two`	**# +24**
	`.quad Derived$beta`	**# +32**
`Sub$$:`	**`.quad Derived$$`**	**# +0 superclass**
	`.quad Sub$alpha`	**# +8**
	`.quad Derived$one`	**# +16**
	`.quad Derived$two`	**# +24**
	`.quad Sub$beta`	**# +32**
	`.quad Sub$fun`	**# +40**

Notes: The entries for `alpha` and `one` in `Derived$$` must be in the order shown to match the offsets in `Base$$`. The entries for `two` and `beta` in `Derived$$` could be in the order shown above or could be reversed.

The order of entries for functions in offsets +8 to +32 of `Sub$$` must match the ordering in `Derived$$` exactly. The new entry for `fun` must be at the end of `Sub$$`.

CSE 401/M501 22au Final Exam 12/13/22 Sample Solution

Question 4. (15 points) A little optimization. For this question we'd like to perform local constant propagation and folding (compile-time arithmetic), plus copy propagation (reuse values that are already present in another temporary t*i* when possible), strength reduction (replace expensive operations like * with cheaper ones when possible), common subexpression elimination, and dead code elimination.

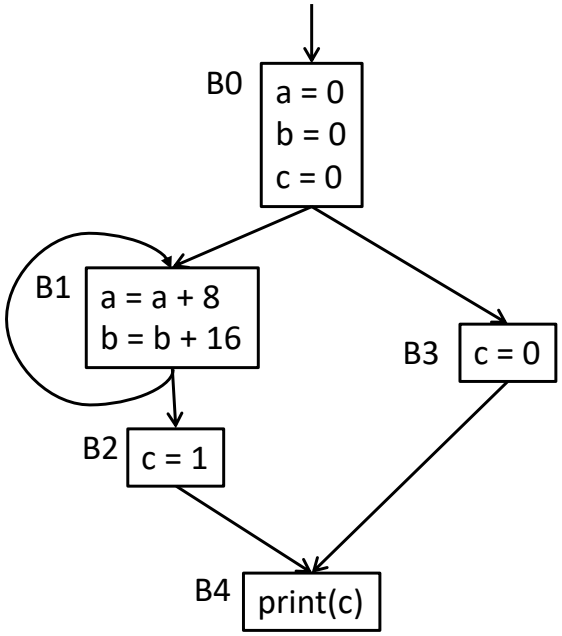
The left column of the table below gives the three-address code generated for this statement: y[i] = m*x[i]+b. All values are assumed to occupy 8 bytes each.

(a) Fill in the middle column with the code from the left column after any changes due to constant propagation and folding, copy propagation, strength reduction, and common subexpression elimination, but before any dead code elimination. You should not attempt any algebraic re-writing of the code to change the order of operations. (Notes: memory reference addresses can use a register (t*i* or `fp`) and a constant offset only – they cannot be more complex. Also note that the arrays `x` and `y` are assumed to be local values stored in the current stack frame, as is possible in C or C++ code, instead of being a pointer to an array allocated elsewhere in memory as in Java.)

(b) In the last column, put an X under "deleted" if the statement would be deleted by dead code elimination after performing the constant propagation/folding, copy, and strength reduction optimizations in part (a).

	Original Code	After constant prop./folding & copy prop., strength reduction, and CSE (copy original code if no change)	"X" if deleted as dead code
a	t1 = *(fp + moffset) // m	**t1 = *(fp + moffset) // m**	
b	t2 = *(fp + ioffset) // i	**t2 = *(fp + ioffset) // i**	
c	t3 = t2 * 8 // i*8	**t3 = t2 <<3 // i*8**	
d	t4 = fp + t3	**t4 = fp + t3**	
e	t5 = *(t4 + xoffset) // x[i]	**t5 = *(t4 + xoffset) // x[i]**	
f	t6 = t1 * t5 // m*x[i]	**t6 = t1 * t5 // m*x[i]**	
g	t7 = *(fp + boffset) // b	**t7 = *(fp + boffset) // b**	
h	t8 = t6 + t7 // m*x[i]+b	**t8 = t6 + t7 // m*x[i]+b**	
i	t9 = *(fp + ioffset) // i	**t9 = t2 // i**	**X**
j	t10 = t9 * 8 // i*8	**t10 = t3 // i*8**	**X**
k	t11 = fp + t10	**t11 = t4**	**X**
l	*(t11 + yoffset) = t8 // y[i]=…	***(t4 + yoffset) = t8 // y[i]=…**	

The next two questions concern the following control flow graph:



Question 5. (20 points) Dataflow – live variables. Recall from lecture that *live-variable* analysis determines for each point *p* in a program which variables are live at that point. A live variable *v* at point *p* is one where there exists a path from point *p* to another point *q* where *v* is used without *v* being redefined anywhere along that path. The sets for the live variable dataflow problem are:

use[*b*] = variables used in block *b* before any definition
def[*b*] = variables defined in block *b*
in[*b*] = variables live on entry to block *b*
out[*b*] = variables live on exit from block *b*

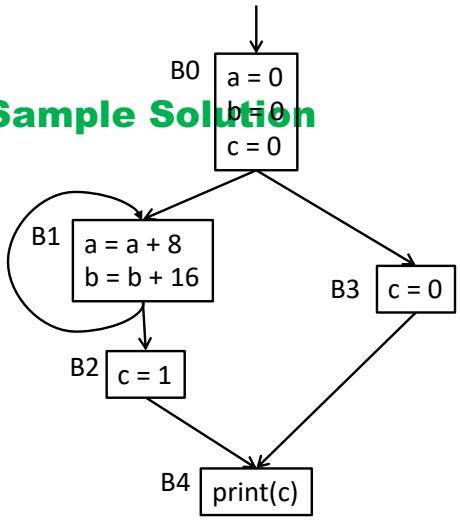
The dataflow equations for live variables are

in[*b*] = use[*b*] ∪ (out[*b*] – def[*b*])
out[*b*] = $\cup_{s \in succ[b]}$ in[*s*]

On the next page, calculate the use and def sets for each block, then solve for the in and out sets of each block. A table is provided with room for the use and def sets for each block and up to three iterations of the main algorithm to solve for the in and out sets. If the algorithm does not converge after three iterations, use additional space below the table for additional iterations.

Then remember to answer the undefined variable question at the bottom of the page.

Hint: remember that live-variables is a backwards dataflow problem, so the algorithm should update the sets from the end of the flowgraph towards the beginning to reduce the total amount of work needed.



Question 5. (cont). Graph and definitions repeated from previous page:

use[*b*] = variables used in block *b* before any definition
def[*b*] = variables defined in block *b*
in[*b*] = variables live on entry to block *b*
out[*b*] = variables live on exit from block *b*

in[*b*] = use[*b*] ∪ (out[*b*] – def[*b*])
out[*b*] = $\cup_{s \in succ[b]}$ in[*s*]

(a) (18 points) Write the results of calculations for live variables in the chart below. If you need extra space for further iterations, please continue on one of the blank pages at the end of the exam and add a note here indicating where your answer continues. (Note: you should write down the details of the individual passes so if there is an error in the final result we will be better able to evaluate what happened.)

Block	use	def	out (1)	in (1)	out (2)	in (2)	out (3)	in (3)
B4	**c**	--	--	**c**	--	**c**		
B3	--	**c**	**c**	--	**c**	--	**no changes**	
B2	--	**c**	**c**	--	**c**	--	**on pass 3**	
B1	**a, b**	**a, b**	--	**a, b**	**a, b**	**a, b**		
B0	--	**a, b, c**	**a, b**	--	**a, b**	--		

(b) (2 points) One use of live variable analysis is to detect potential use of uninitialized variables in a program. Are there any potentially uninitialized variables in this flowgraph, and if so which ones, where, and why? (i.e., justify your answer using the information from the analysis you have done above.)

No. There are no live variables on entry to B0.

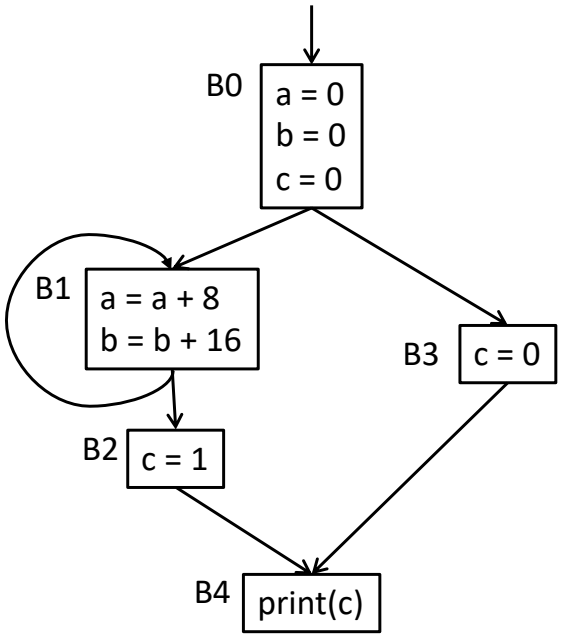
Question 6. (20 points) Dominators and SSA. Here are the basic definitions of dominators and related concepts we have seen previously in class:

- 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 *y* is the set of all nodes *x* that dominate *y*.
- 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*

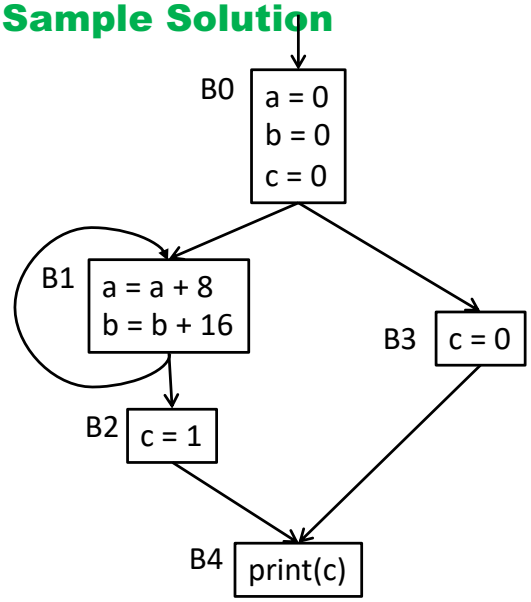
  A node might not have an immediate dominator. A node has at most one immediate dominator.
- The **dominator tree** of a control flow graph is a tree where there is an edge from every node *x* to its immediate dominator idom(*x*).
- The **dominance frontier** of a node *x* is the set of all nodes *w* such that
  - *x* dominates a predecessor of *w*, but
  - *x* does not strictly dominate *w*

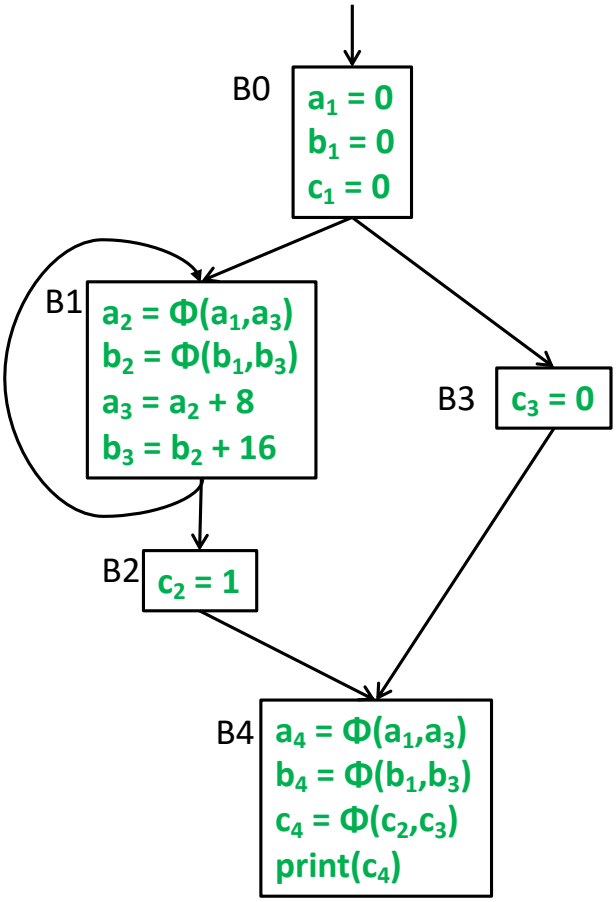
(a) (8 points) Using the same control flow graph from the previous problem, complete the following table. List for each node: the node(s) that it strictly dominates and the nodes that are in its dominance frontier (if any):

Node	Strictly dominates	Dominance Frontier
B0	**B1, B2, B3, B4**	**--**
B1	**B2**	**B1, B4**
B2	**--**	**B4**
B3	**--**	**B4**
B4	**--**	**--**



Question 6 (cont.) (b) (12 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.





Notes: The phi functions must be placed as shown. The subscript numbers for the variables could be different depending on the order in which they are assigned, but should be consistent with the above diagram.

This question concerns register allocation. Assume that we're using the same hypothetical machine that was presented in lecture and in some textbook examples.

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. Given the statement w = x*w + 2*(y+z); here's what the instruction selector generated:

```
a.  LOAD   t1 <- x          // t1 = a
b.  LOAD   t2 <- w          // t2 = w
c.  MULT   t3 <- t1, t2     // t3 = x*w
d.  LOAD   t4 <- y          // t4 = y
e.  LOAD   t5 <- z          // t5 = z
f.  ADD    t6 <- t4, t5     // t6 = y+z
g.  ADD    t7 <- t6, t6     // t7 = 2*(y+z)
h.  ADD    t8 <- t3, t7     // t8 = x*w + 2*(y+z)
i.  STORE  w <- t8          // w = x*w + 2*(y+z)
```

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 question we're only concerned with register allocation so we'll assume that this is the fixed, final order of the instructions.

Answer the questions

about this sequence of code

on

the

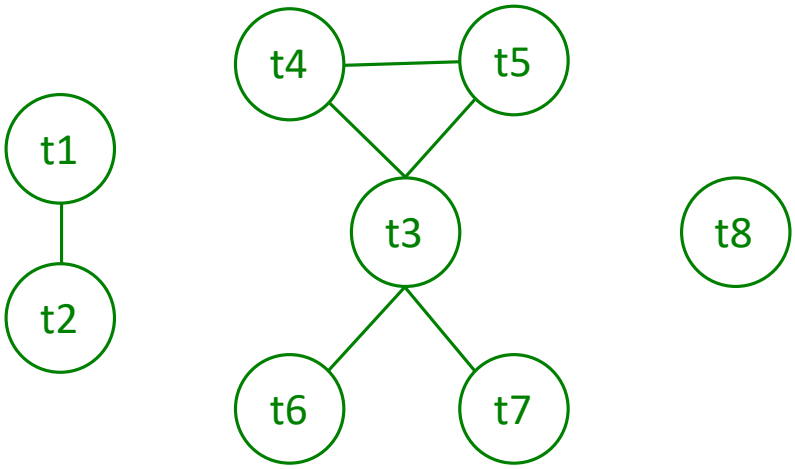
next

page…..

```
a.  LOAD   t1 <- x
b.  LOAD   t2 <- w
c.  MULT   t3 <- t1, t2
d.  LOAD   t4 <- y
e.  LOAD   t5 <- z
f.  ADD    t6 <- t4, t5
g.  ADD    t7 <- t6, t6
h.  ADD    t8 <- t3, t7
i.  STORE  w <- t8
```

Question 7. (15 points) Register allocation/graph coloring.

(a) (10 points) Draw the interference graph for the temporary variables (t1-t8) in this code. You should assume that the code is executed in the sequence given and not rearranged before assigning registers.



(b) (5 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.

Three registers are needed. Here is one possible assignment:

R1: t1, t3, t8
R2: t2, t4, t6, t7
R3: t5

A couple of short-answer questions to finish up. Keep your answers brief and to the point. It should be possible to answer these questions with a few sentences at most.

Question 8. (3 points) *Generational garbage collectors* are widely used in contemporary systems. In these collectors, objects are allocated from only a small part of the total heap space ("new space" or the "nursery") and that part of the heap is collected frequently. The entire heap is only scanned occasionally to reclaim free space. Why is this an effective strategy for reducing the overhead of garbage collection?

Objects allocated on the heap generally have short lifetimes. That means that if we allocate objects in a small part of the heap and collect that part frequently, we are likely to collect a large fraction of the unreachable objects in memory, without the overhead of examining other parts of the heap that contain longer-lived objects that are much less likely to be unreachable.

Question 9. (3 points) x86-64 systems, like most others, have an elaborate set of conventions about how registers are managed when one function calls another. Some registers must be preserved (saved and restored if used) by the called function, other registers can be freely altered and the caller may not assume that their contents will be preserved across a function call. Why this complexity? Why not do something simpler: either require a called function to save and restore all registers that it uses, or else establish the rule that when we call a function, that function might change every one of the registers, with the possible exception of a few basic ones like the stack pointer?

The goal is to reduce the overhead of function calls, particularly the amount of memory loads and stores needed. By allowing some registers to be used freely without saving or restoring, the code for a typical function will be less likely to need to save and restore registers to have enough available registers needed for the generated instructions. Callers will be able to assume that some registers will be unchanged after calling another function and won't need the memory overhead of saving all in-use registers at every call even though it is unlikely that all of those registers would be needed by a typical function (functions are often quite small, requiring few registers).

Question 10. (2 free points – all answers get the free points) Draw a picture of something you are planning to do over winter break!

Have a great holiday break and best wishes for the new year!
The CSE 401/M501 staff