CSCI 565 - Compiler Design

Spring 2011

Second Test

April 27, 2011 at 3.30 PM in Room RTH 115

Duration: 2h 30 min.

Please label all pages you turn in with your name and student number.

Name: ____ **Number:** ___________________

Grade:

Problem 1 [25 points]:

Problem 2 [35 points]:

Problem 3 [40 points]:

Total:

Instructions:

1. This is a closed book Exam.
2. The test booklet contains four (4) pages including this cover page.
3. Clearly label all pages you turn in with your name and student ID number.
4. Append, by stapling or attaching your answer pages.
5. Use a black or blue pen (not a pencil).

Problem 1. Code Generation and Run-Time Environment [25 points]

In this problem we will explore a code generation scheme for concurrency generation and termination spawn and wait constructs respectively. The spawn construct conceptually creates a new thread of execution that begins after the invocation of the spawn completes. The arguments include the address of a function (returning void) and a selected set of arguments. The spawn construct returns a handle to the executing threads, which can be passed around to a wait construct. The wait construct will block until the corresponding thread completes execution.

```
/* this is the spawning parent context */l
…
tid = spawn(myFunc, v1, v2)
…
wait (tid)
…

void myFunc(int p1, int p2){
  …
 return;
}
```

In terms of code generation this means that the enclosed thread, i.e., the one within which the code executes the spawn function, needs to create an execution environment. Similarly, the thread executing the wait primitive (the parent thread) needs to dispose of the created execution context (the child thread). Termination of the spawn is done via the return instruction. Notice also that while the spawned thread executes the spawning threads (or parent thread) will continue its execution possibly spawning other threads and/or execution sequentially and invoking other functions sequentially before attempting to synchronize with the spawned thread(s).

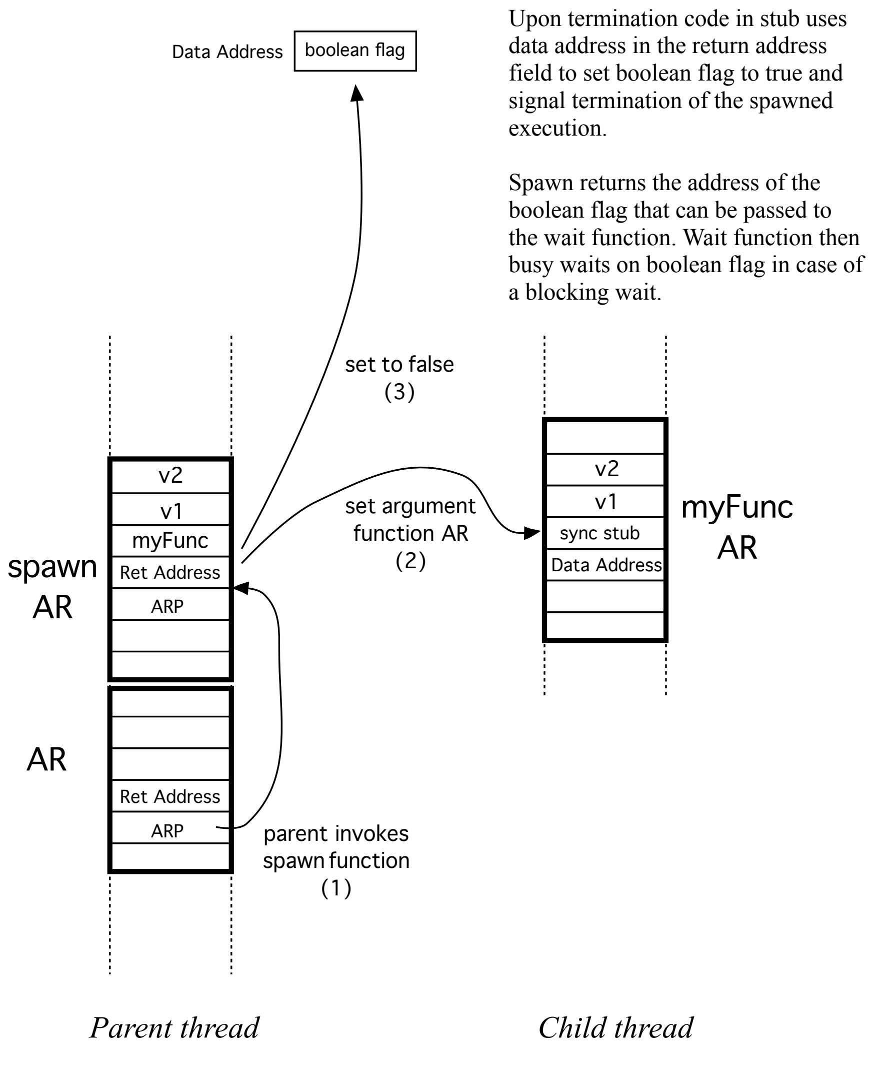
Discuss in broad terms how to create and maintain the execution environment for the spawned threads and establish the synchronization between the spawning thread and the spawned threads. Be specific about when and where to allocate the AR for the spawned threads, how to pass the corresponding arguments and how to synchronize on termination of the spawned thread.

Solution:

There are several issues when spawning a thread when the caller – the parent thread creates the executing context of the called function. This can be accomplished by having the spawn construct be implemented as a simple function that creates an activation records for its argument function and passes along the parameters used when the called did invoke the spawn.

Inside the spawn implementation the code can create yet another AR off on a different stack (a cactus stack) and write in the place of the argument to the spawned function. The spawn function then signals another processor or process at the OS level and returns to the calling context with an identifier. There can be a specific data structure that holds the return status of the spawned function. The caller (the function that executed the spawn construct) can then query this specific memory location for termination of the spawned function. This is the implementation of the wait function. As to the spawned function, it initiates its execution and the compiler includes in the return address on the newly created AR on the cactus stack a code stub that uses the identifier passed to the caller, so that on a return or exit the code can signal the spawning threads that the execution terminated.

Figure 1 below depicts the various steps of this implementation highlighting a possible implementation of the auxiliary structure used for termination synchronization.


Data Address
boolean flag
Upon termination code in stub uses data address in the return address field to set boolean flag to true and signal termination of the spawned execution.
Spawn returns the address of the boolean flag that can be passed to the wait function. Wait function then busy waits on boolean flag in case of a blocking wait.
set to false (3)
v2
v1
myFunc
Ret Address
ARP
spawn AR
set argument function AR (2)
sync stub
Data Address
myFunc AR
AR
Ret Address
ARP
parent invokes spawn function (1)
Parent thread
Child thread

Problem 2. Live Variables Analysis and Register Allocation [35 points]

In this problem we will explore the use of a global graph-coloring based register allocation and assignment and the simpler top-down register allocation algorithm based on the frequency of occurrence of each variable used in the code. For this problem consider the following 3-address instruction sequence below where you need not be concerned about the code that precedes the line 8 of the code nor the code afterwards. Assume the first basic block where line 8 is located is basic block labeled BB0 and the last basic block where line 23 is located is basic block BB3.

```
8:          t1 = 4
9:          t6 = 0
10:  L1:    t1 = t1 – 1
11:         t2 = t1 * 4
12:         t3 = GP + t2
13:         t4 = 0
14:         *t3 = t4
15:         if t1 = 0 goto L2
16:         t4 = 0
17:         t5 = ARP + 8
18:         *t5 = t4
19:         goto L1
20:  L2:    t6 = t6 + 1
21:         t7 = ARP + t1
22:         *t7 = t6
23:         goto L3
```

a. [10 points] **Recognize** the basic blocks of this code. Under the assumption that t6 is live at the end of basic block containing line 23 **compute the live ranges** of the variables in this code segment. You need not to show the iterative data-flow analysis algorithm steps.

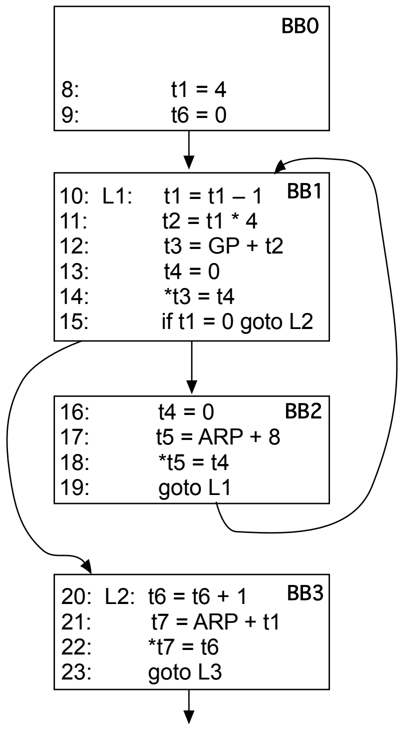
b. [10 points] Derive the interference graphs (or table) for these variables using both conflict definitions described in class.

c. [10 points] Can you color the resulting interference graphs with 3 colors? Why or why not? If not suggest a way to split one of the webs so that the resulting interference graph is 3-colorable.

d. [5 points] Would the top-down algorithm described in class work well for this example? Justify.

Solution:

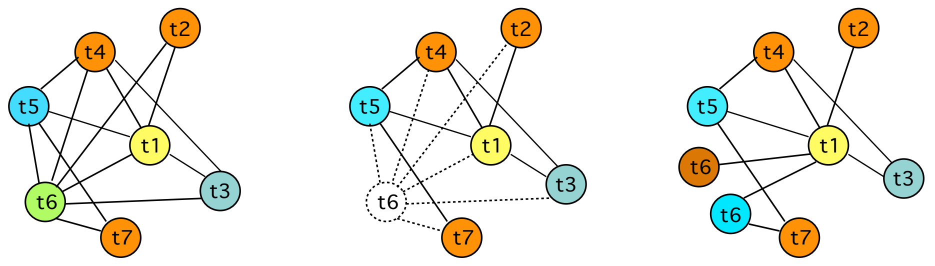
a. [10 points] The Control-Flow Graph (CFG) is as shown below.



```
BB0
8:          t1 = 4
9:          t6 = 0

BB1
10:  L1:    t1 = t1 – 1
11:         t2 = t1 * 4
12:         t3 = GP + t2
13:         t4 = 0
14:         *t3 = t4
15:         if t1 = 0 goto L2

BB2
16:         t4 = 0
17:         t5 = ARP + 8
18:         *t5 = t4
19:         goto L1

BB3
20:  L2:  t6 = t6 + 1
21:         t7 = ARP + t1
22:         *t7 = t6
23:         goto L3
```

b. [10 points] The live ranges are indicated in the table below (left). Here we present only the line number where each variable is active. The first interference graph on the right, however, does indicate the interference taking into account the last and first use of each variable in a specific instruction. For example variables t1 and t7 do not interfere although they are active in the same instruction in line 21.

Live Ranges	Instruction Ranges
t1	[8 - 21]
t2	[11 - 12]
t3	[12 - 14]
t4	[13 - 14 + 16 - 18]
t5	[17 - 18]
t6	[9 - 22]
t7	[21 - 22]



As can be seen this interference graph does include a graph clique with 4 nodes, respectively t1, t4, t5 and t6. As such it cannot be colored using only 3 colors and thus we cannot use only 3 registers. The first option is to color the graph with 4 colors as shown in the first graph. The alternative as discussed in class is to either not color at all one of the nodes (as is the case of t6 shown in the graph in the center) or split the corresponding web. The second alternative, of splitting the web, one could select the web with the longest inactivity section. In this case a good candidate would be t6. The last interference graph shows the resulting interference graph when we split the web of t6 split into two webs. Now the graph can be easily colored with 3 colors and thus we can use 3 registers.

For this particular example using the top-down register allocation would lead to very different results as illustrated in the figure below. The first table presents the heuristic weight of the various basic blocks where BB1 and BB2 have a weight of 10 as they are nested inside a loop. Taking the number of accesses to each variable as the metric of interest, the algorithm ranks the variables as shown in the last table on the right-hand side.

BB ID	00	01	02	03
Frequency	1	10	10	1

Variable	00	01	02	03
t1	1	4	0	1
t2	0	2	0	0
t3	0	2	0	0
t4	0	2	2	0
t5	0	0	2	0
t6	1	0	0	3
t7	0	0	0	2

Variable	Metric
t1	42
t2	20
t3	20
t4	40
t5	20
t6	4
t7	2

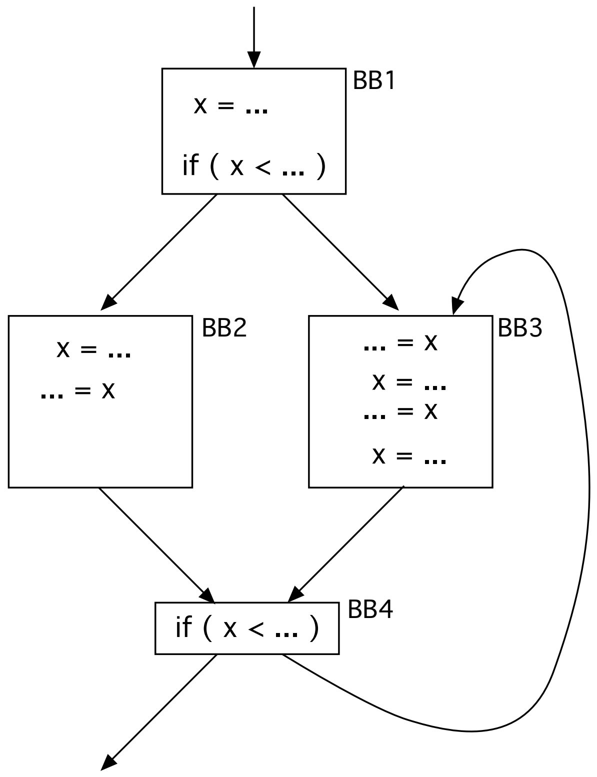
As can be seen there are a lot of variables that rank very closely. If only three registers are available, the algorithm will only assign registers to t1, t4 and (arbitrarily) t2 resulting in a very different implementation than the one derived using the graph-coloring algorithm,

Problem 3: Analysis and Code Representation [40 points]

There are several compiler passes that rely on the information about which variables are defined and used. Register allocation is such a case whose information can be derived directly from live variable analysis or by *def-use* (defined-used) chains. In this problem you are asked to describe the *def-use* (DU) data-flow analysis in detail and discuss its use to derive information for program enhancing transformations. Specifically address the following questions.

a. [20 points] Formulate the *def-use* iterative data-flow analysis problem indicating the structure of the lattice, the meet operator; the transfer functions and the initialization values for the nodes in the program assumed to be the nodes in the CFG corresponding to the program's basic blocks. Justify the choice of initial value in terms of precision and safety of the initial and the resulting final solution it leads to.

b. [10 points] Using the formulation you have developed in a. above apply it to the CFG structure depicted below where you can assume that the initial values for the data-flow abstractions are empty and that they do not change over the various iterations of your algorithm. Moreover, assume there are no other definitions to the x variable other than the ones depicted here. Show the intermediate values of the IN and OUT abstractions for each basic block.



c. [10 points] Use the solution you have found in b. above to convert the instructions in the basic blocks for the snippet of code in b. into SSA form. While a very efficient algorithm for the placement of the phi-function is exceedingly sophisticated, suggest a simple algorithm that uses the information in the DU-chains to place (and in many program instance correctly) the phi-function and thus transform a 3-address representation to SSA form.

Solution:

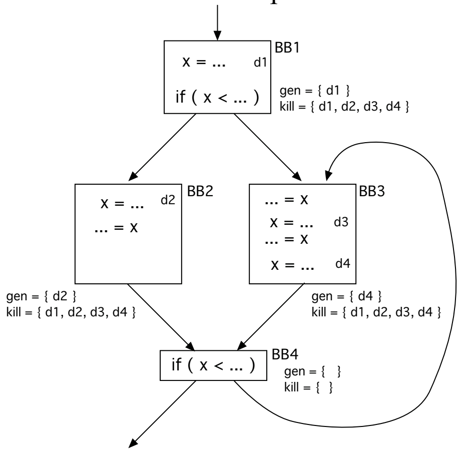
a. [20 points] We use the same formulation as described in class using the common abstractions of In and Out to represent the definitions that reach the input and the output of each basic block in the program's CFG. As such the lattice will consist of the set of definitions for each variable and it closed under the subset relationship. The top element of the lattice (T) is the set of all definitions and the bottom element (⊥), and safest, is the empty set. The Gen set for each basic block is the set of definitions that are downward exposed or downward available and correspond to the definition set in the current basic block for variables that are not later redefined in the same basic block. The Kill set for each basic block corresponds to the definition whose variables are defined in the basic block. Note that the Kill set does include definition in other basic blocks as well as the definition in the basic lock to which it corresponds. The meet operator is in this case the set union and the transfer function for the basic block is defined by the equation

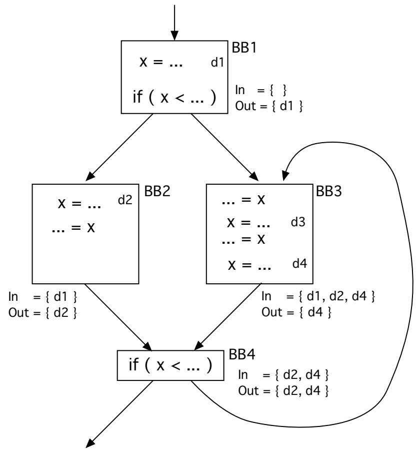
$$\text{In}(n) = \bigcup \text{Out}(p) \text{ for all predecessors nodes p of n}$$
$$\text{Out}(n) = \text{Gen}(n) \cup (\text{In}(n - \text{Kill}(n)))$$

The lattice being the power-set of the definition in the program is of finite height (as well as width) and the set-union is commutative, distributive and associative. This means that not only does the iterative work-list algorithm does converge, but that also the MOP is the same solution as the MFP computed by this algorithm.

As to the initialization, all sets In and Out should be initialized to empty sets. This is the safest assumption to the reaching definitions where we claim that no definition reaches any point of the program.

b. [10 points] The picture below depicts the various values for the iterative data-flow analysis algorithm for this DU-chain problem for the inputs CFG.





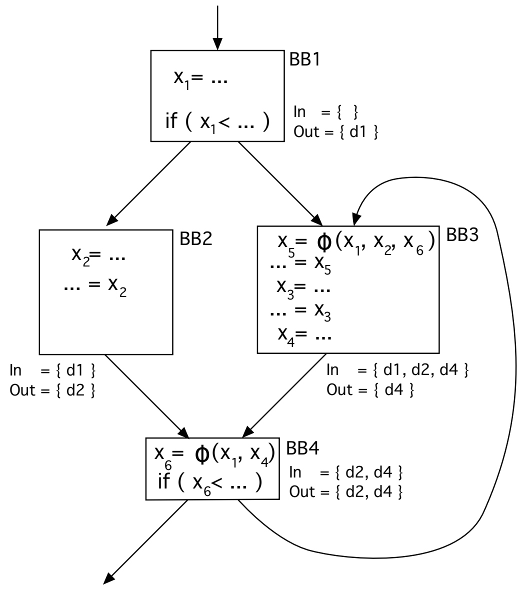
	Iteration 0		Iteration 1		Iteration 2		Iteration 3	
BB	In	Out	In	Out	In	Out	In	Out
1	{ }	{ }	{ }	{ d1 }	{ }	{ d1 }	{ }	{ d1 }
2	{ }	{ }	{ d1 }	{ d2 }	{ d1 }	{ d2 }	{ d1 }	{ d2 }
3	{ }	{ }	{ d1 }	{ d4 }	{ d1 }	{ d4 }	{ d1, d2, d4 }	{ d4 }
4	{ }	{ }	{ d2, d4 }	{ d2, d4 }	{ d2, d4 }	{ d2, d4 }	{ d2, d4 }	{ d2, d4 }

Order = BB1 BB2, BB3, BB4

c. [10 points] The insertion of the phi-function and the subsequent labeling of the various definitions and uses of the variables in SSA-form is very tricky. A possible (and not very efficient nor completely correct algorithm) could be as follows:

```
step1: Insert phi-functions
  for each basic block n and all variables v do
      if multiple definitions of the variable v are in the IN(n) then
        create a new variable v(k) where k is a per-variable counter;
        assign v(k) to a phi-function having as inputs the DU chain ;
         (needs to be revised as the names might have changed – see step 2)
      end if
  end for

step 2: Relabel variables
for each basic block n do
    propagate the new v(k) names inserted at the top of the block to the
    upwards exposed reads of v;
    if there are no assignments to v in n then
       replace v(k) in Out(n)
       propagate v(k) to the phi-functions in all other basic blocks;
    end if
end for
```

Applying this "algorithm" to our example we have to introduce the new variables x_5 and x_6 as at the input of basic blocks BB3 and BB4 we have more than one definition reaching that basic block. The subsequent step 2 will propagate these symbolic values to the statements in their basic blocks. Notice that in the case of BB4 the symbolic x_6 need to be propagated as one of the inputs of the phi-function that defines x_5 as well.


BB1
x1= ...
if (x1< ...)
In = { }
Out = { d1 }
BB2
x2= ...
... = x2
In = { d1 }
Out = { d2 }
BB3
x5= ϕ(x1, x2, x6)
... = x5
x3= ...
... = x3
x4= ...
In = { d1, d2, d4 }
Out = { d4 }
BB4
x6= ϕ(x1, x4)
if (x6< ...)
In = { d2, d4 }
Out = { d2, d4 }