EE573 Fall 2002, Exam 3

open book, open notes. if a question seems ambiguous, ask me to clarify the question. If my answer doesn't satisfy you, please state your assumptions. Look at the black board occasionally to make sure I haven't added information about a question as a result of another students inquiries. If you can't see the blackboard, stand up occasionally and check it. **Tests will be taken up at 4:25**

1: Syllabus question. Prof. Midkiff's office hours are typically held two days a week.
TRUE
FALSE

2: Circle all that are valid peephold optimizations.

a. MUL *op1*,2, *res* becomes SHIFTL 2, *op1* what if no- res ≠ op1?

b. ADD1, *op2*, *op2* becomes INC *op2*

c. MOVE *lit1*, *res1*
MOVE *lit1*+*lit2*, *res3* becomes MOVE *lit1*, *res1*
ADD *lit2*, *res2*, *res3*

d. JEQ *L1*
JMP *L2* ⟶ JNE *L2*
L1

d. SUB 1, *op2*, *op2* ⟶ DECR *op2*

3: consider the arrays declared:

a array b[0:100,0:50]

b array c[1:100,0:50]

c arrayint d[4:100,8:50] Broken question

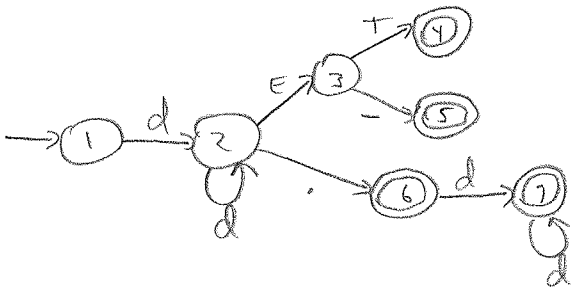
Let **A** be the base of an array's storage. Assume row-major storage. In the table below, place the array name next to the expressions that could address element $b(i,j)$, $c(i,j)$ or $d(i,j)$. Leave spaces blank that are beside addressing expressions that are invalid for arrays *b, c* or *d*.

------	$\mathbf{A} + i \times j \times 100 + 50$	------	$\mathbf{A} + (i-2) \times 100 + j$
------	$\mathbf{A} + i \times 100 + j$	------	$\mathbf{A} + (i-4) \times 100 + (j-8) \times 8$
------	$\mathbf{A} + (i-1) \times 100 + j$	------	$\mathbf{A} + (i-2) \times 100 + j$
------	$\mathbf{A} + (i-4) \times 100 + j \times 8$	------	$\mathbf{A} + (i-4) \times 100 + j$

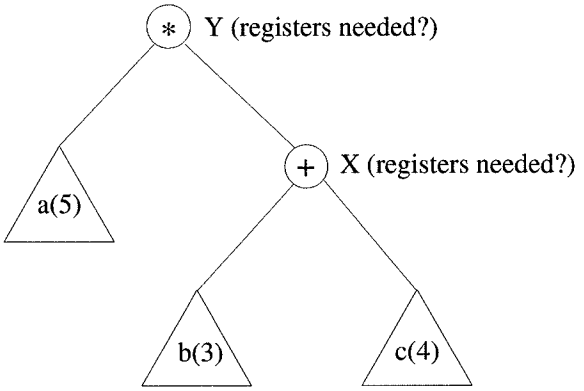
4: write a deterministisc finite state automata that accepts the regular expression

$$d^{+}\ (\ \mathrm{E}\ \{\ +\ |\ -\ \}\ )\ |(\ .\ )d^{*}$$

Note that "(", ")", "{", "}" and "|" are meta-characters (i.e. are used to define the regular expression, but are not symbols in the regular expression.) $\{string\}$ denotes that *string* is optional. *dddE+dd*, *dddE* and, *ddd.d* are three strings in the language defined by this regular expression.



A compiler uses *context sensitive code generation* to generate code for expression trees, and is now ready to generate code for the tree:



Triangles represent expression trees, the label "a (5)" means the name of the tree (for the purpose of refering to it in this question) is "a", and the expression takes 5 registers. The trees formed by the "*" and "+" are labeled "Y" and "X" respectively. Put *a, b, c, Y* or *X* in each dash, with leftmost dash containing the name of the subtree evaluated first.

5: Give the order that code will be generated for the expression trees a, b, c, Y and X.

----------, ----------, ----------, ----------, ----------.

6: How many registers are needed to generate expressions:

X: ----------

Y: ----------

A compiler uses list scheduling within basic blocks. Arithmetic operations take 1 cycles. Loads and stores of variables take 2 cycles. The compiler does top down list scheduling. When two operations (load or arithmetic) can be scheduled, the one with the longest delay is scheduled.

7: Shown below is the code in a basic block. Draw a precedence graph for the basic block (one node for each operation) and label it with the labels (a), (b), (c), ... (g) given for each instruction in the code. Write the delay for the node inside of each node.

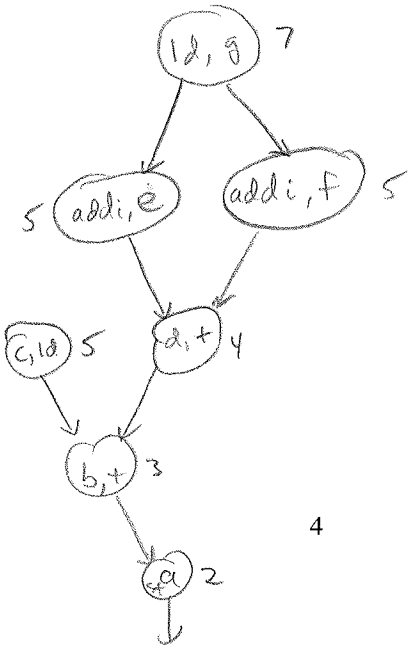
```
(g)  load   i,   r3
(f)  addi   1,   r3   r4
(e)  addi   7,   r3,  r6
(d)  add    r6,  r4,  r7
(c)  load   c,   r8
(b)  add    r7,  r8,  r9
(a)  store  r9,  a
```

The next page is blank if you need the space for your precedence graph.

8: Give the instruction schedule for the basic block (i.e. put one of a, b, c, d, e, f, g, or h on each line.) (1) gets the first instruction to execute in the basic block, (2) the second, and so forth.

(1) g (2) c (3) e (or 1) (4) f (or e)

(5) d (6) b (7) a



Put your precedence graph here if you don't have enough room on the previous page.

The following program is used in questions 9 and 10. A program originally looks like:

```
do i = 1, n
    do j = 1, n
        a(j,i) = ...
    end do
    do j = 1, n
        ... = a(j,i)
    end do
end do
```

and is transformed to

```
do j = 1, n
    do i = 1, n, 2
        a(j,i) = ...
        ... = a(j,i)
        a(j,i+1) = ...
        ... = a(j,i+1)
    end do
    if (i mod 2 == 1) then
        a(j,n) = ...
        ... = a(j,n)
    end if
end do
```

9: Circle the transformations that have been performed on at least one of the loops in the orginal loop nest to create the second loop nest.

a tiling

b prefetching

c interchange

d fusion

e software pipelining

f subroutine inline expansion

g unrolling

h register allocation

10: Assuming the a array is laid out in row-major order, these optimizations have the following effect on the program execution:

a cache locality is improved

b cache locality is harmed

c there are more possibilies of instruction level parallelism

d debugging is made easier

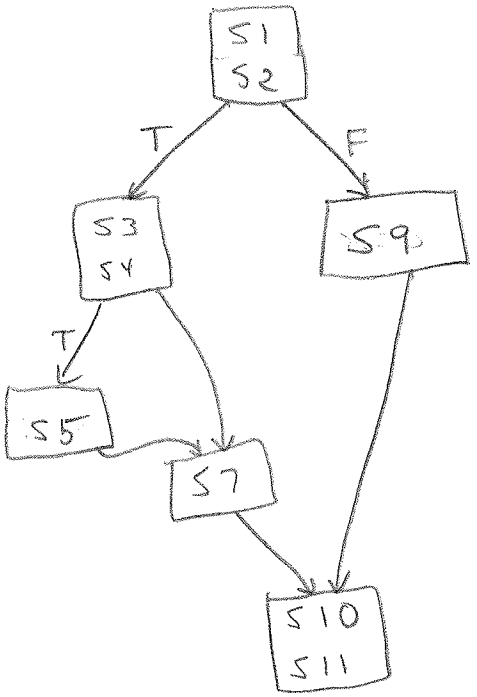
e fewer arithmetic operations are performed

f just as many operations are performed, but because of strength reduction they are cheaper.

11: Given the program fragment

```
S1 :  read(a,b)
S2 :  if (a ≤ b) then {
S3 :        x = 1
S4 :        if (a = b) then {
S5 :              x = 2
S6 :        end if
S7 :        y = 5
S8 :  else
S9 :        x = 3
S10 end if
S11 : write(x,y)
```

draw the control flow graph. Label **true** branches with a "T", and **false** branches with an "F".



You have gotten a job with the Vaporware Compiler Company to design analysis routines for their compiler, in particular a *not very busy register* routine. A register R_i in some block B_j is not very busy if some block $B_k, B_k \in Succ(B_j)$ reads the value in R_i, but not every block $B_k, B_k \in Succ(B_j)$ reads the value. That is, the register R_i is not very busy in some block B_i if some block that immediately follows it in the program execution uses the value in R_i that is in R_i in B_j, but not every block does.

11: A reference to R_i becomes a member of the set $GEN(B_k)$ if
(circle the most correct answer)

a. R_i is read in B_k

b. R_i is written in B_k

c. R_i is read in block B_k before any writes to R_i in B_k

d. R_i is written in block B_k before any reads to R_i in B_k

Note: "before" and "after" refer to the order of a read an write in an execution of the basic block, not the order they might be visited during a dataflow analysis of the basic block.

12: A reference to R_i should become a member of the set $KILL(B_k)$ if
(circle the most correct answer)

a. R_i is read in B_k

b. R_i is written in B_k

c. R_i is read in block B_k before any writes to R_i in B_k

d. R_i is written in block B_k before any reads to R_i in B_k

Note: "before" and "after" refer to the order of a read an write in an execution of the basic block, not the order they might be visited during a dataflow analysis of the basic block.

continued on next page

You have gotten a job with the Vaporware Compiler Company to design analysis routines for their compiler, in particular a *not very busy register* routine. A register R_i in some block B_j is not very busy if some block $B_k, B_k \in Succ(B_j)$ reads the value in R_i, but not every block $B_k, B_k \in Succ(B_j)$ reads the value. That is, the register R_i is not very busy in some block B_i if some block that immediately follows it in the program execution uses the value in R_i that is in R_i in B_j, but not every block does.

13: Let

$$XXX(B_j) = Gen(B_j) \cup (YYY(B_j) - Killed(B_j) \quad (1)$$

$$T_1 = \oplus ZZZ(B_k), B_k \in WWW(B_j) \quad (2)$$

$$T_2 = \odot ZZZ(B_k), B_k \in WWW(B_j) \quad (3)$$

$$YYY(B_j) = T_2 - T_1 \quad (4)$$

Circle the correct function/operator in each line below.

WWW is *Succ* *Pred*

XXX is *In* *Out*

YYY is *In* *Out*

ZZZ is *In* *Out*

$\oplus$ is $\cup$ $\cap$

$\odot$ is $\cup$ $\cap$