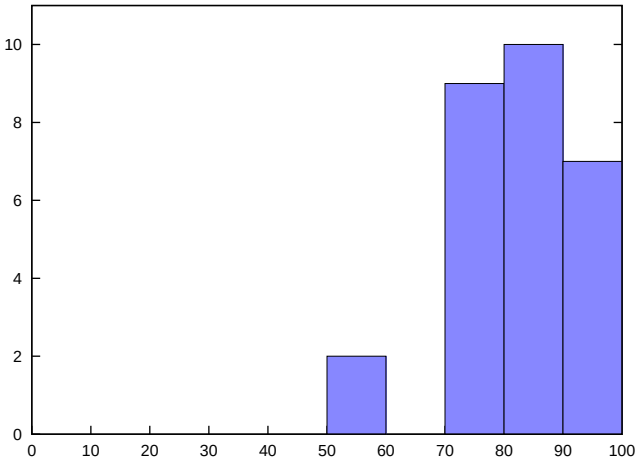
Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.035 Spring 2013

Test III Solutions

Mean 81 Median 83 Std. dev 11.23



I Register Allocation

In this problem you will perform register allocation for the following code. Note that each instruction is numbered. Also, assume that the variables are not used after instruction `9`.

```
1:  a = get_int()
2:  b = get_int()
3:  if (b < 0) {
4:    a = a + b
5:    c = 1
    } else {
6:    c = get_int()
    }
7:  d = c + 1
8:  c = d * a
9:  print(c + 5)
```

1. [8 points]: Write the set of def-use chains for each variable in the program. Write each def-use chain as number pair (d, u) where d is the number of an instruction that defines the variable and u is the number of an instruction that uses that definition. Be sure to label each set with the variable it corresponds to.

Solution:

`a` : $(1,4), (1,8), (4,8)$

`b` : $(2,3), (2,4)$

`c` : $(5,7), (6,7), (8,9)$

`d` : $(7,8)$

2. [8 points]: Write the set of webs for each variable in the program. Write each web as the set of def-use chains that belong to the web. Be sure to label each web with a name – for example $w_1, \ldots, w_n$ – and to label each set with the variable it corresponds to.

Solution:

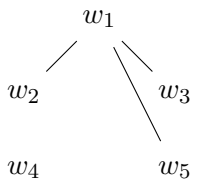
`a` : $w_1 \rightarrow \{(1,4), (1,8), (4,8)\}$

`b` : $w_2 \rightarrow \{(2,3), (2,4)\}$

`c` : $w_3 \rightarrow \{(5,7), (6,7)\}, w_4 \rightarrow \{(8,9)\}$

`d` : $w_5 \rightarrow \{(7,8)\}$

3. [6 points]: Draw the interference graph for the webs that you have identified. Specifically, each node in your graph should represent one web and there should be an edge between two nodes if the two webs interfere. Label each node with the name of the web it represents.



Solution:

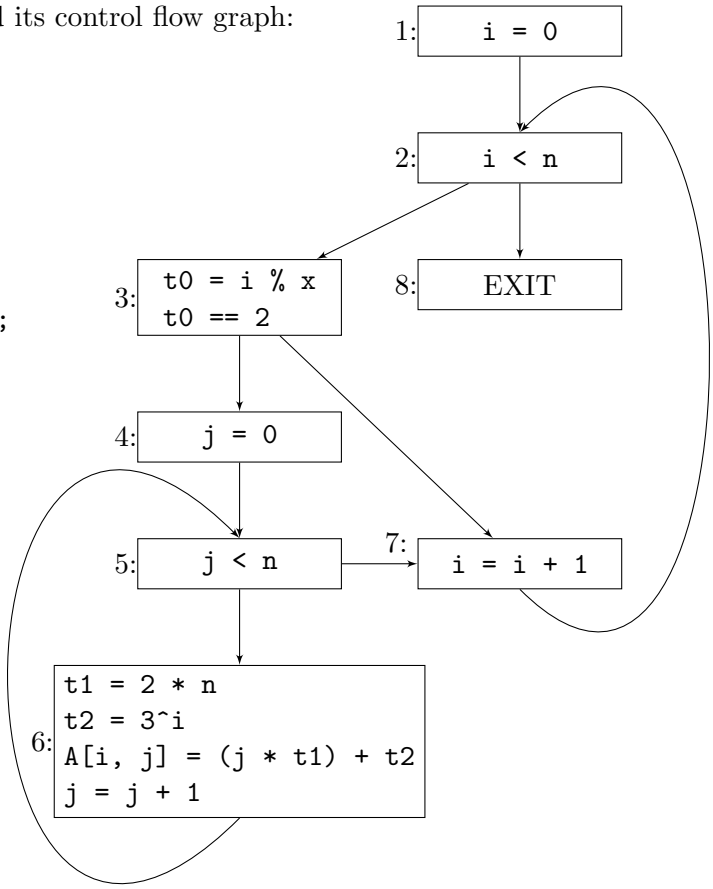
4. [5 points]: Suppose that the architecture we are targeting for compilation has two registers. Can we place all the variables in this code in registers? Explain why or why not using your interference graph as part of your justification.

Solution: Yes, we can color the interference graph with two colors.

II Loop Optimization: Code Hoisting and Strength Reduction

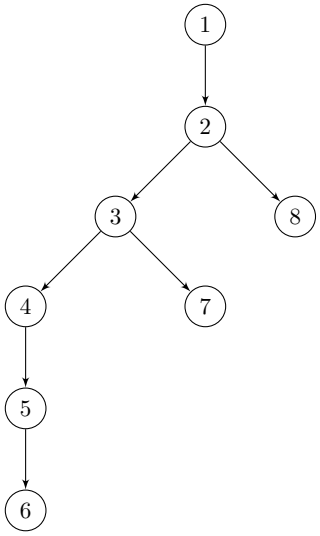
Consider the following snippet of code and its control flow graph:

```
for i = 0 to n {
(a) t0 = i % x;
    if (t0 == 2) {
        for j = 0 to n {
(b)         t1 = 2 * n;
(c)         t2 = 3^i;
(d)         A[i, j] = (j * t1) + t2;
        }
    }
}
```



5. [6 points]: Draw the dominator tree for your control flow graph. Each node should correspond to a basic block in your control flow graph. Label each node with the corresponding label from the control flow graph.

Solution:



6. [6 points]: Which of the labeled statements (`a`, `b`, `c`, `d`) are loop invariant? For each invariant statement explain why it is invariant. Be sure to state in which loop (or loops) it is invariant. Assume that only `i`, `j`, and `A` are live after the loops.

Solution:

- `b` is loop invariant in both the loop for `i` and `j` because it has one reaching definition from outside both loops.
- `c` is loop invariant in the loop for `j` because all of its reaching definitions are outside of that loop. It is not loop invariant in the loop for `i` because its operand has a reaching definition from within the loop and a reaching definition from outside the loop.

7. [4 points]: Perform loop invariant code motion on the code. Use the space below to state to which basic block each labeled statement (if any) should move. You do not need to provide the full code. As before, assume that only `i`, `j`, and `A` are live after the loops.

Solution: We can move statement `b` to block 1 and we can move statement `c` to block 4.

8. [8 points]: The code snippet contains a potentially expensive exponential operation (`t1 = 3^i`). Is it possible to perform strength reduction on this statement? If so, either explain how to perform the transformation on the given control flow graph or provide the resulting control flow graph (or code). If not, explain why not.

Solution: Yes, we can perform strength reduction by changing the statement to `t2 = 3 * t2` and adding the statement `t2 = 1` to block 1. However, we must also be sure to move `t2 = 3 * t2` to ensure that `t2` is updated on each iteration of the loop on `i`. We can either move it to the beginning (or middle) of block 3 and guard it by the condition `i != 0` (i.e. `if (i != 0) { t2 = 3 * t2 }` . Alternatively, we could move `t2 = 3 * t2` to block 7.

III Loop Optimization: Instruction Scheduling

Consider a CPU that has 1) a 2-stage pipelined memory unit (MU) that can issue one memory instruction (load or store) per clock and 2) an ALU that can issue two arithmetic instructions (e.g., addition) per clock. Memory instructions take 2 cycles to complete and arithmetic instructions take 1 cycle to complete.

Consider the following loop and a fragment of x64 code from its body:

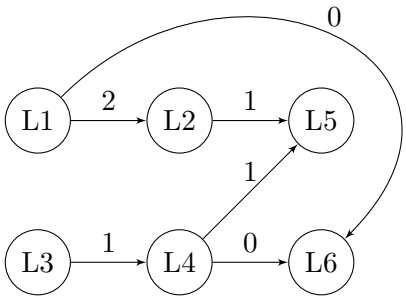
```
for i = 0 to n
  A[i + 2] = A[i] + 1
```

```
L1:     mov (%rdi,%rax), %r11  // t1 = A[i]
L2:     add $1, %r11           // t2 = t1 + 1
L3:     mov $16, %r10
L4:     add %rax, %r10
L5:     mov %r11, (%rdi, %r10) // A[i + 2] = t2
L6:     add $8, %rax
```

Here `mov` *`src`*`,` *`dst`* copies a value from *`src`* to *`dst`* and `add` *`src`*`,` *`dst`* means *`dst`* `+=` *`src`*.

9. [10 points]: Draw the dependence DAG for the above x64 code. Be sure to use directed edges and to label each edge with the latency required between each instruction.

Solution:



10. [10 points]: Use your dependence DAG to complete the following schedule. Fill in the schedule diagram with the remaining instructions. Note that `L3` is scheduled on the ALU because it loads an immediate value into a register and does not access memory.

Solution:

		Clock															
		1		2		3		4		5		6		7		8	
MU	Stage 1	L1						L5									
	Stage 2			L1						L5							
ALU		L3		L4	L6	L2											

11. [0 points]: (BONUS - 5 points)

We cannot use the distance vector method to parallelize this loop even though it is clear that there are no dependencies between even and odd iterations.

If we unroll and register rename a second iteration of the loop, can instruction scheduling achieve full parallelization of the adjacent iterations on our single CPU? Here full parallelization means executing two iterations in the same number of cycles as one iteration.

If yes, provide a schedule. If no, explain why not.

Solution: No, it takes at least 5 cycles to execute `L1`, `L2`, and `L5`. Because the CPU only has one memory unit, one of the iterations can't start this sequence until clock 2 at the earliest, meaning that the schedule ends in 6 cycles at the earliest. This is greater than our original 5 cycle schedule.

IV Parallelization

In class we discussed how compilers use loop transformations like scalar privatization and loop skewing to eliminate or change the dependencies of a loop so that the resulting loop is easily parallelizble. In this question, we will consider two other transformations that compilers often use.

- **Loop Splitting.** Loop splitting splits the iteration space of a loop into a set of disjoint ranges and creates a new loop (with the same body as the original loop) for each of the ranges. For example, loop splitting may transform

  ```
  for i = 0 to 10
    f(i);
  ```

  into

  ```
  for i = 0 to 5
    f(i);
  for i = 5 to 10
    f(i);
  ```

- **Loop Fission.** Loop fission splits the body of a loop into disjoint sets of statements and creates a new loop (with the same iteration space) for each set of statements. For example, loop fission may transform

  ```
  for i = 0 to n
    f(i);
    g(i);
  ```

  into

  ```
  for i = 0 to n
    f(i);
  for i = 0 to n
    g(i);
  ```

For each of the loops on the next several pages, perform the following:

A. Provide the distance vectors for the original loop.

B. Identify if either loop splitting or loop fission can be used to create at least one parallelizable loop. If yes, provide or describe the transformed code and identify which loops – including nested loops – are parallelizable and why. If no, state why neither transformation can create a parallelizable loop.

(continues on next page...)

12. [10 points]:

```
for i = 0 to n
    A[i] = A[i] + A[0];
```

A. Solution: The distance vectors are [0] and [*].

B. Solution: A compiler can use loop splitting to produce the following code:

```
i = 0;
if (0 < n) {
    A[0] = A[0] + A[0];
    for i = 1 to n
      A[i] = A[i] + A[0];
}
```

or

```
for i = 0 to 1
   A[i] = A[i] + A[0];
for i = 1 to n
   A[i] = A[i] + A[0];
```

The distance vector for the new loop is now just [0], which can be parallelized.

13. [7 points]:

```
for i = 0 to n
    A[i] = A[i + 1] + A[0];
```

A. Solution: The distance vectors are [1] and [*].

B. Solution: Loop fission doesn't apply as there is only one statement in the loop. Loop splitting also doesn't work here because even if we did remove the first iteration to eliminate the [*] distance vector, the remaining distance vector is [1], which is not parallelizable.

14. [12 points]:

```
for i = 1 to n
    for j = 1 to n
        A[i, j] = A[i - 1, j + 1] + c;
        B[i, j + 1] = B[i, j] + d;
```

A. Solution: The distance vectors are $\begin{bmatrix} 1 \\ -1 \end{bmatrix}$ and $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$.

B. Solution: A compiler can use loop fission to separate the dependence vectors and create the following code:

```
for i = 1 to n
    for j = 1 to n
        A[i, j] = A[i - 1, j + 1] + c;
for i = 1 to n
    for j = 1 to n
        B[i, j + 1] = B[i, j] + d;
```

The first loop can be parallelized on the inner loop over `j` because its distance vector is now $\begin{bmatrix} 1 \\ -1 \end{bmatrix}$. The second loop can be parallelized on the outer loop over `i` because its distance vector is now $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$.