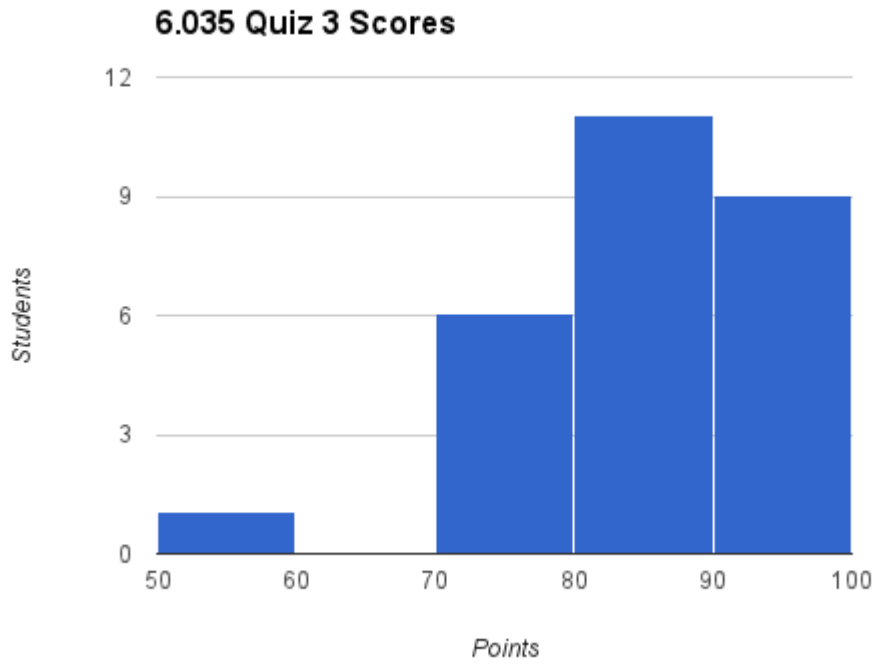
Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.035 Fall

Test III Solutions



Average: 84.9 Median: 86 StDev: 9.1

I Register Allocation

In this problem you will perform register allocation for the following Decaf program code. Note that each instruction is numbered. The functions `get_int()` and `printf()` are callout functions.

```
 void main() {
    int x, y, z;

1:  x = get_int()
2:  y = get_int()
3:  while (x > 0) {
4:    x = x - y;
    }
5:  z = x * y;
6:  if (z > -32) {
7:     z = z - y;
    } else {
8:     z = z + y;
    }
9:  printf (z);
 }
```

1. [6 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.

Solution:

`x` : $(1,3), (1,4), (1,5), (4,3), (4,4), (4,5)$

`y` : $(2,4), (2,5), (2,7), (2,8)$

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

2. [6 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. Label each web with a name, $w_1, \ldots, w_n$.

Solution:

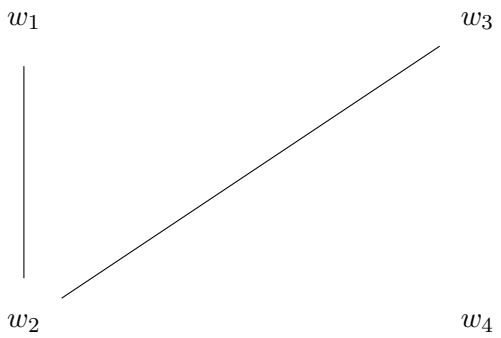
`x` : $w_1 \rightarrow \{(1,3),(1,4),(1,5),(4,3),(4,4),(4,5)\}$

`y` : $w_2 \rightarrow \{(2,4),(2,5),(2,7),(2,8)\}$

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

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. [6 points]: What is the minimum number of registers that we can use to store all variables without spilling? Justify your answer.

Solution:

2 registers, since the interference graph from the previous question is two-colorable.

5. [3 points]: Assuming that you compile this Decaf program to the x86 architecture, which register would you select for the variable `z` to minimize the number of the generated assembly instructions?

Solution:

`%rdi` since this is the first parameter of `printf`.

6. [3 points]: Assuming that you compile this Decaf program to the x86 architecture, which register would you select for the variable `y` to minimize the number of the generated assembly instructions?

Solution:

`%rax` since this is the register that contains the return value of `get_int`.

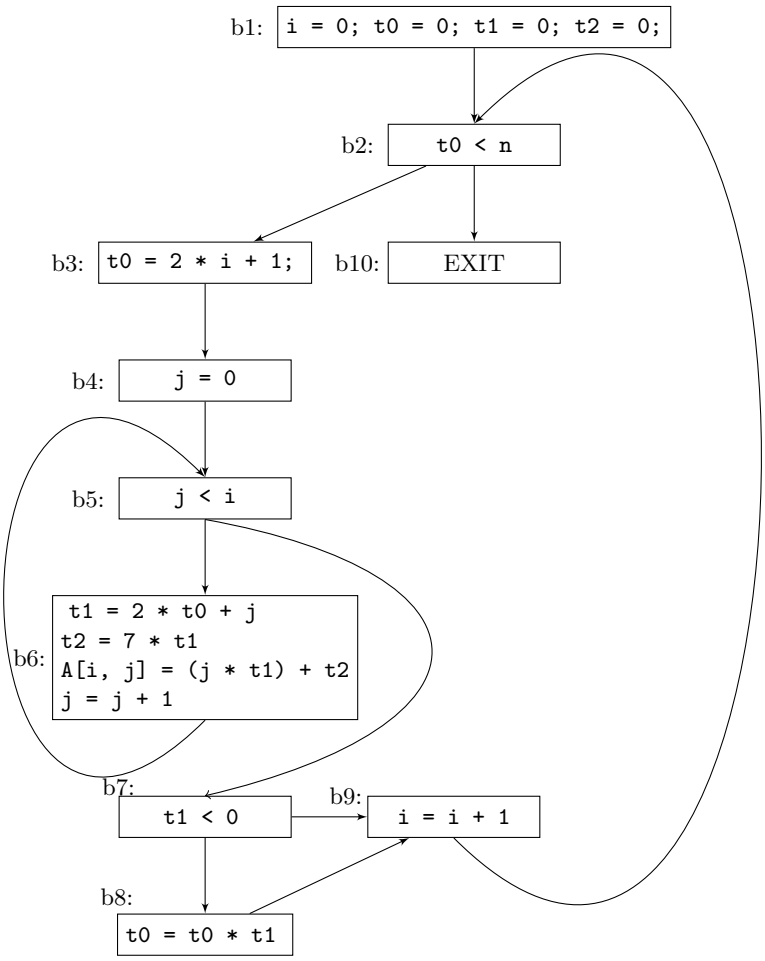
II Loop Identification and Induction Variables

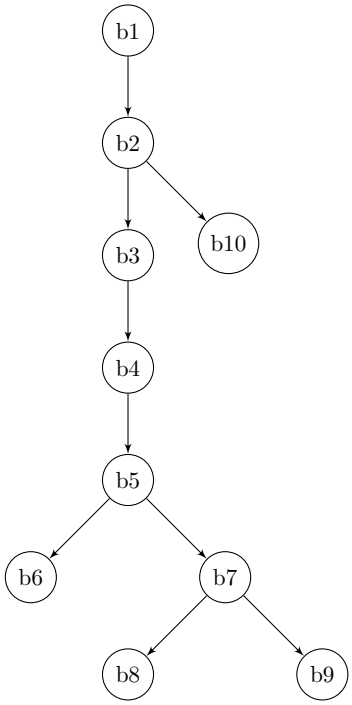
Consider the following snippet of code (`n` is a constant, `A` is a global array; assume that the local variables are not used after the loops):

```
 1:   i = 0; t0 = 0; t1 = 0; t2 = 0;

 2:   while ( t0 < n ) {
 3:      t0 = 2 * i + 1;

 4:      for j = 0 to i {
 5:        t1 = 2 * t0 + j;
 6:        t2 = 7 * t1;
 7:        A[i, j] = (j * t1) + t2;
         }

 8:      if (t1 < 0) {
 9:        t0 = t0 * t1;
         }
10:      i = i + 1;
     }
```

7. [8 points]: Draw the control-flow graph and the dominator tree of this code side-by-side below. Mark the nodes that contain the statements with the corresponding numbers.

Solution:


b1:
i = 0; t0 = 0; t1 = 0; t2 = 0;
b2:
t0 < n
b3:
t0 = 2 * i + 1;
b10:
EXIT
b4:
j = 0
b5:
j < i
b6:
t1 = 2 * t0 + j
t2 = 7 * t1
A[i, j] = (j * t1) + t2
j = j + 1
b7:
t1 < 0
b9:
i = i + 1
b8:
t0 = t0 * t1


b1
b2
b10
b3
b4
b5
b6
b7
b8
b9

8. [2 points]: On the previous CFG, mark the header node of the outer loop with `HO` and the header node of the inner loop with `HI`. Explain below why these nodes represent headers.

Solution:

These are the nodes `b2` and `b6`. From the dominator tree, we can see that the destination node of a back-edge dominates the source node.

9. [5 points]: Identify the basic and the derived induction variables for the inner loop.

Solution:

Basic: `j`

Derived: `t1 (<j, 1, 2*t0>)` and `t2 (<j, 7, 14*t0>)`

10. [5 points]: Is `t0` an induction variable in the outer loop? Explain why or why not. If not, is there a way to transform the program to make `t0` an induction variable?

Solution:

It is not an induction variable, because it is also updated by a non-constant value in the conditional branch (Lines 8-9).

To make it an induction variable, run first the sign analysis to ensure that the variable `t1` cannot be negative, and therefore the condition `t1 < 0` can be replaced with `false`. Then run dead code elimination to remove the conditional.

11. [5 points]: Find a loop invariant code in either the inner or the outer loop. For each such piece of code, specify its line. Write below the optimized loop, after applying the loop-invariant code movement transformation.

Solution:

Loop invariant code: The expression `2*t0` on Line 5.

Transform the code by adding a new variable `t3`:

```
4':     t3 = 2 * t0;
4:      for j = 0 to i {
5:        t1 = t3 + j;
6:        t2 = 7 * t1;
7:        A[i, j] = (j * t1) + t2;
        }
```

12. [5 points]: Can we use the result of the induction variable analysis to perform strength reduction in the inner loop? If yes, write the optimized loop body. If no, explain why.

Solution:

```
4a:     t1 = 2 * t0 - 1;
4b:     t2 = 14 * t0 - 7;
4:      for j = 0 to i {
5:        t1 = t1 + 1
6:        t2 = t2 + 7;
7:        A[i, j] = (j * t1) + t2;
        }
```

III Instruction Scheduling

Consider a simple machine that executes the instructions as follows:

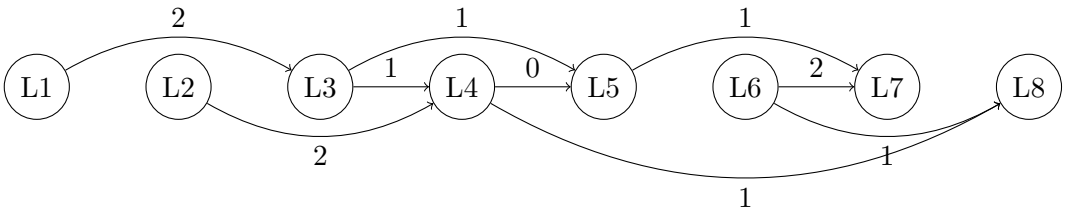
- ALU instructions are executed within one clock cycle
- Memory load instructions are executed within two clock cycles
- Memory store instructions are executed within one clock cycle

Consider the following snippet of assembly code (we use local variables to represent stack locations):

```
1:     mov 0(%rbp), %r0
2:     mov -8(%rbp), %r1
3:     mul $3, %r0
4:     add %r0, %r1
5:     sub $1, %r0
6:     mov -16(%rbp), %r2
7:     add %r0, %r2
8:     mov %r1, -16(%rbp)
```

13. [7 points]: Draw the dependence DAG for this code. Label each edge with the latency between the instructions.

Solution:



14. [4 points]: Consider the machine with a single memory and a single ALU unit. Assume that the CPU executes without instruction scheduling. Fill in the table with the labels `L1` to `L8` for the instructions from the previous page that execute in each cycle. In how many cycles does the base execution complete?

Solution:

	Clock													
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Memory Unit	L1	L1	L2	L2	.	.	.	L6	L6	.	L8			
ALU	.	.	.	.	L3	L4	L5	.	.	L7				

Base execution takes the total of 11 cycles.

15. [7 points]: Consider the machine with a single memory and a single ALU unit. Use list scheduling algorithm to schedule the instructions and fill in the table below (use labels `L1` to `L8` for the corresponding instructions). What is the speedup compared to the base execution?

Solution:

	Clock									
	1	2	3	4	5	6	7	8	9	10
Memory Unit	L1	L1	L2	L2	L6	L6	L8			
ALU	.	.	L3	.	L4	L5	L7			

Speedup: 1.57 (11/7)

16. [7 points]: Consider now the machine with *two* stages memory unit (which can issue one memory operation in each cycle) and a single ALU unit. Use list scheduling algorithm to schedule the instructions and fill in the table below (use labels `L1` to `L8` for the corresponding instructions). What is the speedup compared to the base execution?

Solution:

	Clock									
	1	2	3	4	5	6	7	8	9	10
Memory Unit 1	L1	L2	L6	.	L8	.				
Memory Unit 2	.	L1	L2	L6	.	.				
ALU	.	.	L3	L4	L5	L7				

Speedup: 1.83 (11/6)

IV Loop Optimization

Consider the following loop (`A` is a global array, `y` and `t` are local variables, `%` is the modulo operator):

```
for i = 1 to n {

  t = A[i] / A[i-1];

  if ( i%2 == 1 ) { t = -t; }

  y = y + t;

}
```

17. [10 points]: Unroll the loop body two times. Write down the semantically equivalent unrolled loop (use `BODY(J)` to denote the three statements in the body of the loop for an induction variable `J`).

Solution:

```
i = 1;
while (i <= n-1)  {
  t = A[i] / A[i-1];
  if ( i%2 == 1 ) { t = -t; }
  y = y + t;
  i = i + 1;

  t = A[i] / A[i-1];
  if ( i%2 == 1 ) { t = -t; }
  y = y + t;
  i = i + 1;
}

if (i <= n) {
     t = A[i] / A[i-1];
     if ( i%2 == 1 ) { t = -t; }
     y = y + t;
     i = i + 1;
}
```

18. [5 points]: Outline the analysis and the transformation that would remove unnecessary `if` conditionals from the body of the unrolled loop.

Solution:

First, run an analysis that keeps track of the parity of variables. Since the initial value of `i` is known at the beginning of the loop execution, the analysis should be able to tell that the condition of the first `if` in the unrolled loop is always `true` and the condition of the second `if` in the unrolled loop is always `false`.