Problem Set 4

Fall 23

Due: Sunday, October 29th.

1. Optimization Short Answers

   Answer the following questions:

   (a) Doing a poor job of alias analysis (i.e., the compiler thinks too many variables could alias each other) can make register allocation worse. Why?

Solution:

When aliased variables are allocated to different registers, extra operations (e.g., save dirty registers to memory or free invalid registers) are needed to maintain the consistency between the variables. These operations are unnecessary unless the variables indeed alias each other.

   (b) Better common subexpression elimination can make register allocation worse. Why?

Solution:

The variable used to memoize a common subexpression becomes live, no matter whether it was live before the CSE. The new live variables put more pressure on register allocation and may cause extra spill code.

2. Common Subexpression Elimination

   for the following sub-problems, consider the following piece of three-address code:

```
READ(A)
READ(B)
C = A + B
D = A + C
B = A + B
D = A + B
A = A + C
T1 = C + A
T2 = A + D
T3 = A + C
WRITE(T3)
```

   (a) Show the result of performing Common Subexpression Elimination (CSE) on the above code. Rather than generating assembly, just give new 3AC. If you're reusing the results of an expression, just generate code that looks like X = Y, where Y is the variable/temporary that held the result of the original calculation.

(b) Suppose C and A were aliased. How would that change the results of CSE?

Solution:

(a)
```
READ(A)
READ(B)
C = A + B
D = A + C
B = C // A + B
D = A + B
A = A + C
T1 = C + A
T2 = A + D
T3 = T1 // A + C
WRITE(T3)
```

(b)
```
READ(A)
READ(B)
C = A + B
D = A + C
B = A + B
D = A + B
A = A + C
T1 = C + A
T2 = A + D
T3 = T1 // A + C
WRITE(T3)
```

3. Register Allocation

For the following sub-problems, we will perform register allocation using bottom-up register allocation as presented in the notes. READ(X) defines the value of X (i.e., it assigns to X). WRITE(X) reads the value of X to write it out to the screen. Consider the following piece of three-address code:

```
34  READ(A)
35  READ(B)
36  C = A * B
37  D = C + B
38  C = C + D
39  E = B * B
40  A = E + C
41  F = A + E
42  B = A + F
43  WRITE(B)
```

(a) Perform liveness analysis on this piece of code, assuming that it is the entire code of the program. Show which variables are live after each of the statements in the program.

(b) Show what code would be generated for each 3AC instruction. Use LOAD X Rx to load from a variable/temporary into a register. STORE Rx X to store from a register into a variable/temporary, Rx = Ry + Rz for addition, and Rx = Ry * Rz for multiplication.

When choosing registers to allocate, always allocate the lowest-numbered register available. When choosing registers to spill, choose the non-dirty register that will be used farthest in the future. In case all registers are dirty, choose the register that will be used farthest in the future. In case of a tie, choose the lowest-numbered register.

If a LOAD or STORE is the result of a spill (kicking a value out of a register earlier than required, or loading one of those values back into the register), indicate that.

- Perform register allocation for a machine with 4 registers.
- Perform register allocation for a machine with 3 registers.

(c) Show the interference graph for this piece of code.

(d) Perform register coloring, assuming three registers. If you reach an uncolorable graph, spill arbitrary variable (assume that spills do not require any extra registers– you can use three registers for unspilled variables).

Solution:

(a) The live sets are shown below.

```
1   READ(A) // {A}
2   READ(B) // {B, A}
3   C = A * B // {C, B}
4   D = C + B // {D, C, B}
5   C = C + D // {C, B}
6   E = B * B // {E, C}
7   A = E + C // {A, E}
8   F = A + E // {F, A}
9   B = A + F // {B}
10  WRITE(B)
```

(b) *With 4 registers:*

When we start out, none of the registers are allocated.

```
R1:   R2:   R3:   R4:
```

For each 3AC instruction, we will explain the steps of register allocation and show the assembly instructions that are generated.

READ(A) There are no source operands in this instruction, so we do not need to ensure that anything is in a register. Then we find a register for A, R1:

```
GETI R1 // R1: A* R2:     R3:   R4:
```

READ(B) We put B in R2:

```
GETI R2 // R1: A* R2: B*  R3:   R4:
```

C = A * B We ensure that A and B are in register (which they are), then put C in R1:

```
R1 = R1 * R2 // R1: C* R2: B*  R3:   R4:
```

D = C + B We ensure that C and B are in registers and put D in R3:

```
R3 = R1 + R2 // R1: C* R2: B*  R3: D* R4:
```

C = C + D We ensure that C and D are in registers, and update R3 and free R1, because D is dead. Note that we do not generate a store from this free, because even though D is dirty, it is not live.

```
R1 = R3 + R1 // R1: C* R2: B*  R3:    R4:
```

E = B * B We ensure that B is in register and put E in R2:

```
R2 = R2 * R2 // R1: C* R2: E*  R3:    R4:
```

A = E + C We ensure that E and C are in registers, and then put A in R1.

```
R1 = R2 + R1 // R1: A* R2: E*  R3:    R4:
```

F = A + E We ensure that A and E are in registers and put F in R2:

```
R2 = R1 + R2 // R1: A* R2: F*  R3:    R4:
```

B = A + F We ensure that A and F are in registers, then free them and put B in R1:

```
R1 = R1 + R2 // R1: B* R2:     R3:    R4:
```

WRITE(B) We write out R1!

```
PUTI R1
```

(c) *With 3 registers:* a When we start out, none of the registers are allocated.

```
R1:   R2:   R3:
```

For each 3AC instruction, we will explain the steps of register allocation and show the assembly instructions that are generated.

READ(A) There are no source operands in this instruction, so we do not need to ensure that anything is in a register. Then we find a register for A, R1:

```
GETI R1 // R1: A* R2:   R3:
```

READ(B) We put B in R2:

```
GETI R2 // R1: A* R2: B*  R3:
```

C = A * B We ensure that A and B are in register (which they are), then put C in R1:

```
R1 = R1 * R2 // R1: C* R2: B*  R3:
```

D = C + B We ensure that C and B are in registers and put D in R3:

```
R3 = R1 + R2 // R1: C* R2: B*  R3: D*
```

C = C + D We ensure that C and D are in registers, and update R1 and free R3, because D is dead. Note that we do not generate a store from this free, because even though D is dirty, it is not live.

```
R1 = R3 + R1 // R1: C* R2: B*  R3:
```

E = B * B We ensure that B is in register and put E in R2:

```
R2 = R2 * R2 // R1: C* R2: E*  R3:
```

A = E + C We ensure that E and C are in registers, and then put A in R1.

```
R1 = R2 + R1 // R1: A* R2: E*  R3:
```

F = A + E We ensure that A and E are in registers and put F in R2:

```
R2 = R1 + R2 // R1: A* R2: F*  R3:
```

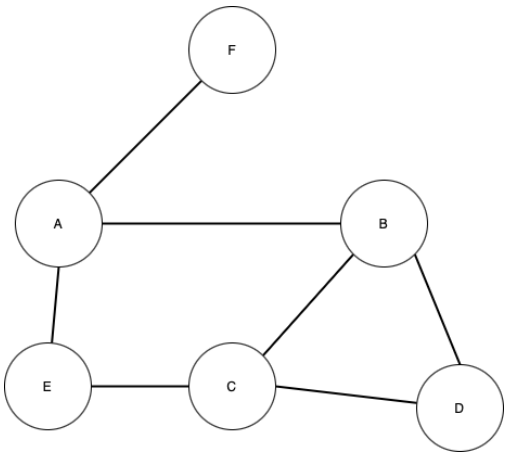
B = A + F We ensure that A and F are in registers, then free them and put B in R1:

```
R1 = R1 + R2 // R1: B* R2:  R3:
```

WRITE(B) We write out R1!

```
PUTI R1
```

(d) Interference graph:



(e) The graph can be simplified by removing F, A, and E first. Then we color B, C, and D different colors, then add back nodes one at a time until we reach our original graph.