CS 415 Compilers: Problem Set 1

Spring 2022

Due date: Wednesday, February 9, 11:59pm

Problem 1 – ILOC code shape

Appendix A in our textbook (EaC) discusses ILOC, a linear assembly code for a simple abstract RISC machine. Here, you will also be able to use an additional instruction that allows you to print out a variable value: outputAI *register*, *constant* (print MEM[CONTENT(register) + constant]).

The code that you are writing follows the memory layout as discussed in class. Byte addresses smaller than 1024 are reserved and should not be used for program variables. Program variables are addressed through offsets from address 1024. The reserved register r_0 needs to contain this value.

This problem is about code shape. The general goal is to keep values of variables in registers if possible while preserving specific conditions such as the notion of "memory consistency". Let's define memory consistency as the property of a program that if you set a break-point between two instructions and inspect the memory content of all program variables at that point, the values of the variables in memory have to reflect all memory operations before the breakpoint. For example, if you view the memory state between instructions S2 and S3, you should be able to observe MEM(a) = 2 and MEM(b) = 3, assuming that a and b are not aliased. The values of the other variables should be undefined, or have arbitrary values.

Please provide correct ILOC code for the following conditions and the listed basic block. Do not perform any optimizations such as constant propagation. Clearly, the printed value of d could be computed at compile time.

1. All variables may be aliased.
2. Variables a and b, and c and d may be aliased. Howver, both a and b are not aliased with c or d. Memory consistency has to be preserved.
3. No two variables are aliased. Memory consistency has to be preserved.
4. No two variables are aliased. Memory consistency may not be preserved.

```
S1: a := 2;
S2: b := 3;
S3: c := a + b;
S4: d := a * b + a * c;
S5: d := a + c + d;
S6: PRINT d;
```

You can execute your ILOC code using the ILOC simulator *sim*. Execute your ILOC code in file "test.i" by saying "*./sim < test.i*". The simulator is available on the ilab cluster at (~uli/cs415/ILOC_Simulator).

Instead of copying the *sim* simulator executable to your personal cs415 directory, I recommend to use a softlink to the "installed", i.e., our provided simulator executable. If we need to update the executable, you will have immediate access to the updated version.

Problem 2 – Anti-Dependencies

```
a   loadI 1024 => r0
b   loadI 2 => r1
c   storeAI r1 => r0, 4
d   loadI 3 => r1
e   storeAI r1 => r0, 8
f   loadAI r0, 4 => r1
g   loadAI r0, 8 => r2
h   add r1, r2 => r3
i   storeAI r3 => r0, 12
j   outputAI r0, 12
```

There is an anti-dependence from statement c to statement d, and statement e to statement f.

1. What is the number of cycles needed to run this code assuming the latencies used in class (see lecture 3)? Do not reorder the instructions.

2. Can you remove the anti-dependencies? If so, give the code. What is the number of cycles needed to run the modified code without anti-dependences using latencies as above. Do not reorder or eliminate any instructions.

3. What are the advantages and disadvantages of removing anti-dependencies?

Problem 3 – Instruction scheduling

Perform forward list scheduling for the following ILOC code:

```
a   loadI 1024  => r0
b   loadI 0  => r1
c   storeAI r1  => r0, 0
d   loadI 63  => r3
e   storeAI r3  => r0, 4
f   loadI 5  => r5
g   loadAI r0, 0  => r6
h   add r5, r6  => r7
i   storeAI r7  => r0, 8
j   loadAI r0, 8  => r3
k   loadI 9  => r10
l   sub r3, r10  => r11
m   storeAI r11  => r0, 12
n   loadAI r0, 4  => r13
o   loadI 3  => r14
p   mult r13, r14  => r15
q   storeAI r15  => r0, 16
r   loadAI r0, 16  => r3
s   loadI 7  => r18
t   mult r3, r18  => r4
u   storeAI r4  => r0, 20
v   loadAI r0, 12  => r21
w   loadAI r0, 20  => r22
x    add r21, r22  => r23
y   storeAI r23  => r0, 24
z   loadAI r0, 24  => r25
aa   storeAI r25  => r0, 28
bb   outputAI r0, 28
```

There are many possible variants of the basic forward list scheduling algorithm.

1. Show the assignment S(n) of instruction issue times to instructions when no list scheduling is performed. How many cycles does the program take?

2. Show the dependence graph for the basic block. All true, anti, and output dependences needed to ensure the correct order of execution. You may omit dependences that are "covered" by other dependences in the graph.

3. Label the nodes in the dependence graph based on the longest latency-weighted path (see our class notes). Use the latencies as we discussed in class (anti-dependencies have full latency).

4. Show the result of forward list scheduling, i.e., S(n) using the longest latency-weighted path heuristic. How many cycles does the program take?