CSC 462/562 Computer Architecture
Homework #6 answer key

1.
   a. True: a from S1 to S2 and S1 to S3, and b from S2 to S1 (loop-carried)
      Anti: a from S2 to S3
      Output: a from S1 to S3
      Parallelizable by rearranging instructions
   b. True: x from S3 to S1 and S2 (loop carried) and x from S3 to S4
      Anti: x from S2 to S1 and y from S4 to S1
      Output: x from S2 to S3 (loop carried) might be considered an output dependence
      Not parallelizable because x[i] and x[i+1] both appear in the same loop iteration on the left and right hand side of assignment statements.

2.
   a. a = 5, b = -1, c = 3, d = 1, test gives us (1 – (-1)) / GCD(3, 5) = 2 / 1, so there is a loop carried dependence (for instance when i=1)
   b. a = 3, b = 3, c = 1, d = -1, test gives us (3 – (-1)) / GCD(3, 1) = 4 / 1, so there is a loop carried dependence (for instance when i=0 and i=4)
   c. a = 24, b = 1, c = 9, d = -1, test gives us (1 – (-1)) / GCD(24, 9) = 2 / 3, so the test does not tell us anything but we can infer there will never be a dependence because c[24*i+1] will always be 1 greater than a multiple of 3 and c[9*i-1] will always be 1 less than a multiple of 3.
   d. a = 2, b = -1, c = 4, d = 0, test gives us (0 - -1) / GCD(2, 4) = 1 / 2, so the test does not tell us anything but we can infer there will never be a dependence because 2*i-1 is always odd and 4*i is always even.

3.
   a. Original (no speculation):

```
Loop:   lw      x1, 10000(x9)
        slt     x2, x7, x1
        beq     x2, x0, else
        addiw   x7, x7, 1
        j       next
else:   addiw   x1, x1, 1
        sw      x1, 10000(x9)
next:   addiw   x9, x9, 4
        addiw   x6, x6, 1
        bne     x6, x8, Loop
```

   b. Speculated code:

```
Loop:   lw      x1, 10000(x9)
        slt     x2, x7, x1
        addiw   x7, x7, 1
        bne     x2, x0, next
        addiw   x1, x1, 1
        sw      x1, 10000(x9)
        subiw   x7, x7, 1
next:   addiw   x9, x9, 4
        addiw   x6, x6, 1
        bne     x6, x8, Loop
```

c. The original code executes 11 cycles for the if clause and 10 cycles for the else clause, taking 1000 * (11 * .80 + 10 * .20) = 10800 cycles. The revised code executes 9 cycles for the if clause and 12 cycles for the else clause, taking 1000 * (9 * .80 + 12 * .20) = 9600 cycles. Speedup = 10800 / 9600 = 1.125.

d. If each clause is equally likely, the original code runs in 1000 * (11 * .50 + 10 * .50) = 10500 cycles and the revised code in 1000 * (9 * .50 + 12 * .50) = 105000 cycles, so no speedup at all.

e. If we want to speculate the else clause, we can increment x1 but we wouldn't want to write the result back to memory until we were sure of the speculation for two reasons. First, if we speculated the else clause, incremented x1 and wrote it back to a[i] and then were incorrect, we would have to decrement x1 and write back to a[i] a second time. So, "undoing" the speculation would take 2 operations instead of 1. Second, memory references can cause exceptions (although incrementing a register might also cause an exception, it is far less likely). So, speculating the else clause creates a greater risk of having an exception that, if we miss-speculated, should not have arisen, and this requires extra mechanisms to handle.

4. Here are the two sets of code without speculation:

```
        slt     x6, x1, x2                  slt     x6, x1, x2
        beq     x6, x0, else                beq     x6, x0, else
        addw    x3, x4, x0                  addw    x3, x5, x0
        j       next                        j       next
else:   addw    x3, x5, x0          else:   addw    x4, x5, x0
next:                               next:
```

Here are the two sets of code with speculation:

```
        addw    x3, x4, x0                  addw    x3, x5, x0
        slt     x6, x1, x2                  slt     x6, x1, x2
        bne     x6, x0, next                bne     x6, x0, next
        addw    x3, x5, x0                  addw    x4, x5, x0
next:                                       lw      x3, …
```

The reason the second if-else is harder to speculate is that the if and else clause are modifying different variables. Should we miss-speculate, undoing our speculation is harder because we have to restore x3 to its original value. In the case of the first if-else statement, we store a result in x3 no matter what, so undoing or miss-speculation is just a matter of replacing the value in x3 (x4) with the correct value (x5). For the second if-else statement, restoring x3 has to be done either by loading the value from memory or copying it from some temp register. Either way, it is more work.

5. An if statement, when compiled into RISC-V, has one instruction to test a condition (e.g., slt x1, x2, x3) and a second instruction to branch around the if clause. A predicated instruction tests the condition and if true, performs the action which would have been in the if clause and if the condition is false, the instruction turns into a no-op (that is, the action is canceled). Thus, there is no branch instruction. In order for this to work, the condition and action must be executable in one cycle of an EX stage (or possibly in two cycles, in the ID stage and the EX stage). Therefore, the condition must be simple (e.g., x == 0, x != 0) and the action must be simple (e.g., x++ or x = y). Further, to keep the instruction within 32 bits, we must confine ourselves to no more than 3 operands. The condition must operate on a single register (compared to say 0 or not 0) and the action must operate on only two operands, or the condition might be able to operate on two registers while the action must be limited to a single operand, say to clear or set a register or increment or decrement a register. Here are some examples of infeasible predicate instructions.

```
if(x == y) a = b;                -- four operands
if(x > 0 && y > 0) x = 0;        -- condition cannot be tested in 1 cycle
if(a > b) c++;                   -- this may be doable but we usually limit our condition
                                    to == or !=
```

6. There are 4 sets of instructions to unroll, two flds, the fmult.d, the fadd.d and the fsd. Thus, each symbolically unrolled loop will be 4 iterations worth. The adjustments we need to make are: arrange the instructions as fsd for the 1st iteration, fadd.d for 2nd, fmult.d for 3rd and two flds for 4th. We need to modify the array offset of the fsd by 4 iterations worth (32). The two addiw instructions can be used to remove other stalls (the addiw for x2 can cause a RAW hazard stall, so we move it up, and the addiw for x1 can fill the branch delay slot). In moving one of the addiw's to the branch delay slot, we have to adjust an array offset.

```
Loop:   fsd     f4, -32(x1)
        fadd.d  f4, f2, f3
        fmult.d f2, f0, f1
        addiw   x2, x2, 8
        fld     f0, 0(x1)
        fld     f1, -8(x2)          // -8 because we added 8 above
        bne     x2, x3, Loop
        addiw   x1, x1, 8
```

7. In the code below, I'm pairing up the two flds for one iteration in one bundle.

Template	Slot 0	Slot 1	Slot 2
8	fld	fld	
8	fld	fld	
14	fld	fld	fmult.d
14	fld	fld	fmult.d
14	fld	fld	fmult.d
12		addiw	fmult.d
12		addiw	fmult.d
12		addiw	fadd.d
12		fadd.d	
12		fadd.d	
14	fsd		fadd.d
14	fsd		fadd.d
14	fsd		
18	fsd	bne	
0	fsd		

CPI = 15 / 29 = .517 (not very good compared to the ideal of .333). Part of the reason for this is that we are dealing with 2 FP operations that must be done in sequence, so there isn't a lot of available instruction to schedule 3 at a time. If you unroll this fewer than 5 times, you wind up having more slots of 2 or 1 instruction reducing the CPI even more. Unrolling more times provides a slight improvement to CPI but not by much.