Appendix H sample problems

1. Identify the true, anti and output dependences for each of the following loops. For each dependence, indicate from which statement to which statement and on which variable.
   a.
```
for(i=0;i<n;i++) {
        a[i]=b[i]+x;     // S1
        b[i]=c[i]-y;     // S2
        a[i]=d[i]*z;     // S3
        e[i]=a[i]+1;     // S4
}
```
   b.
```
for(i=0;i<n;i++) {
        x[i]=x[i]+1;     // S1
        y[i]=x[i]*2;     // S2
        x[i]=x[i-1]-1;   // S3
}
```
   c.
```
for(i=0;i<n;i++) {
        m[i]=n[i]*o[i];// S1
        o[i]=n[i]*m[i];  // S2
        n[i]=p[i]+o[i];   // S3
        p[i]=m[i]-n[i];   // S4
}
```

   Answer:
   a. true: a from S3 to S4, anti: b from S2 to S1, output: a from S1 to S3
   b. true: x from S1 to S2 and from S3 to S3 (loop carried), anti: x from S2 to S1, note there is no output dependence between S1 and S3 because of the true dependence from S1 to S3
   c. true: m from S1 to S2 and S4, o from S2 to S3, n from S3 to S4, anti: n from S3 to S1 and S2, o from S2 to S1, p from S4 to S3

2. Examine the following loop and answer the questions below.
```
for(i=0;i<n;i++) {
        a[3*i+1]=b[i]+x;
        c[i]=a[i]+y;
}
```
   a. The loop has a loop carried dependence, what is it?
   b. In spite of this, we can parallelize the loop by limiting the number of unrollings to 2. Show what this would look like in high-level language code. NOTE: to accomplish this, you will need to change the start index and precede the loop with some "start-up" code.
   c. Can we similarly unroll the loop 4 times? Explain.

   Answer:
   a. We have a loop carried dependence on a between the first and second assignment statements in the loop. For instance, when i=2, we compute a[7] which is then used several iterations later when i=7.

b. Because the first reference to array a grows by 3 with each iteration, we can unroll the loop twice and get away with it, except for when i=0 because 3*0+1 = 1 so we have a loop carried dependence whose distance is 1 initially before becoming 3. We resolve this by computing a[0] before the loop and then start i at 1 when we unroll the loop. The loop becomes:

```
a[1]=b[0]+x;
c[0]=a[0]+y;
for(i=1;i<n;i+=2) {
        a[3*i+1]=b[i]+x;
        a[3*(i+1)+1]=b[i+1]+x;
        c[i]=a[i]+y;
        c[i+1]=a[i+1]+y;
}
```

Note the adjustments to the loop initialization and step size. We start at 1 as mentioned above, but because we've unrolled the loop, we increment i by 2 instead of 1.

c. No, because the reference to a grows by increments of 3 so we cannot have a loop that references 4 values of a within a single loop iteration.

3. Explain why there is no way to alter the following loop to make it parallelizable.

```
for(i=1;i<n-1;i++) {
        a[i]=b[i+1]*c;
        a[i+1]=b[i]*d;
}
```

Answer: both statements have an array reference with index i and both have one with index i+1. In the powerpoint notes, we saw an example where we could rearrange the two instruction so that the reference of i+1 came first and we changed the other statement's references from i to i+1. We cannot do that here since both instructions have an i and an i+1 reference.

4. Apply the GCD test to determine if there is definitely a dependence in each of the following loops.

a.
```
for(i=1;i<100;i+=4)
    a[3*i-2] = a[5*i+4] + c;
```

b.
```
for(k=0;k<100;k++)
    x[2*k] = x[2*k + 3] + 1;
```

c.
```
for(j=0;j<200;j+=2)
    x[10*j+3] = x[25*j – 2];
```

Answer:

a. a = 3, b = -2, c = 5, d = 4 giving (4 – (-2)) / gcd(3, 5) = 6 / 1, since 6 is evenly divisible by 1, the loop carries a dependence. We can see this with i=4 on the left and i=2 on the right giving us a[14].

b. a = 2, b = 0, c = 2, d = 3 giving (3 – 0) / gcd (2, 2) = 3 / 2, since 3 is not evenly divisible by 2, we cannot prove there is a dependence. However, with a little

analysis, we can see there is no dependence because the leftside index is always even and the rightside index is always odd.

c. a = 10, b = 3, c = 25, d = -2 giving (-2 - 3) / gcd(10, 25) = -5 / 5, so there is a loop carried dependence. This occurs, for instance, when j=7 on the lefthand side and j=3 on the righthand side (index of 73).

5. Use software pipelining to reschedule the loop so that there are no stalls. Provide any start-up and finish-up code. Assume the mulw (integer multiplication) takes 4 cycles to compute.

```
Loop: lw    x2, 0(x1)
      mulw  x4, x2, x3
      lw    x5, 4(x1)
      addw  x7, x5, x6
      sw    x7, 0(x1)
      addiw x1, x1, 8
      bne   x1, x8, Loop
```

Answer: Compiler scheduling would move the two lw's together. If we think of the loop then as having the two lw's, the mulw, the addw, and the sw, we have 4 groups of instructions allowing us to unroll the loop for 4 iterations worth. The loop will consist of the sw for the earliest iteration, the addw for the next iteration, the mulw for the next iteration, and the two lw's for the latest iteration. We move the addiw up so that there is no RAW hazard between it and the bne, and move one lw into the branch delay slot. The loop becomes:

```
Loop: sw    x7, -32(x1)
      addw  x7, x4, x5
      mulw  x4, x2, x3
      addiw x1, x1, 8
      lw    x5, 4(x1)
      bne   x1, x6, Loop
      lw    x2, 0(x1)
```

For the start-up code, we need to do both lw's, mulw and addw of the first iteration, both lw's and the mulw of the second iteration, and both lw's of the third iteration. We have to be careful about our register assignments to make sure that the code flows correctly. We could further schedule this code to be more efficient but that step is omitted here:

```
lw    x10, 0(x1)        // 1st iteration
lw    x11, 4(x1)
mulw  x12, x10, x3
addw  x7, x12, x11
lw    x13, 8(x1)        // 2nd iteration
lw    x5, 12(x1)
mulw  x4, x13, x3
lw    x2, 16(x1)        // 3rd iteration
addiw x1, x1, 24        // reset array offset
```

The finish-up code needs to addw x4 and x5 and store the result, mulw x2 and x3 and add this to x5 and store the result, and sw x7. That code is omitted.

6. For each of the following high-level language statements, come up with (invent) a RISC-V predicated instruction so that the operation can be executed with a single RISC-V instruction without a branch, explaining how your instruction works, or explain why you do not think a predicated instruction would work or is even possible. Assume a, b and c are already stored in registers x1, x2 and x3. Each predicated instruction is only allowed to reference registers and immediate data.
    a. if(a==b) a++;
    b. if(a!=b) c++;
    c. if(a==0) a=b+1;
    d. if(a!=b&&a!=c) a++;
    e. if(a!=b) {temp=a;a=b;b=temp;}
    f. if(a==0) b=0; else c=0;
    g. if(a==0) a=1;

    Answer:
    a. cinceq x1, x1, x2 – conditional increment on equal – increment the first register if the second register equals the third (note we could build this instruction to just use two registers where the first register is compared and also incremented as in cinceq x1, x2, but I chose to use three registers to fit part b)
    b. cincne x3, x1, x2 – same as part a but the opposite condition
    c. caddz x1, x2, 1 – conditional add on zero - this instruction compares the first register to 0 and if equal, sets it to the summation of the second register and the immediate datum; this instruction is doable if we restrict the immediate datum to a reasonable size
    d. although we could potential implement this as cneinc x1, x2, x3, in which if x1 is not equal to either x2 or x3 it is incremented, this would not be a good instruction because the condition is too complex – requiring that two separate pairs of registers be compared and the result ANDed together, not only would this require two condition tester pieces of hardware but also an AND gate (and presumably for other such conditional instructions, a version with an OR gate)
    e. this could be solved with an instruction like cswapne x1, x2, which swaps the two values between the registers if they are not equal, but this would take 3 data movements which we could not do with the current ALU hardware in 1 clock cycle
    f. cclearz x1, x2, x3 – conditional clear on zero - if x1 is 0, clear x2 otherwise clear x3 – this could be done but we have to be careful in how we implement sending the destination register number to the register file because the instruction could operate on one of two registers – this would require a MUX to select between the two destination registers and may be too expensive and complex to implement
    g. cseti x1, 0 – conditional set if the value equals the immediate datum, as with c, we would have to limit the immediate datum's size

7. In the following loop, assume the if-clause executes 80% of the time.
    a. Rewrite the code using RISC-V with no assumptions
    b. Rewrite the code using RISC-V where you apply the if clause automatically and then undo it if the else clause executes

c. Assuming no stalls, how much faster is your code from part b over part a?

d. Assuming 1 cycle stall for each data hazard and a 1 cycle branch penalty for every branch (whether taken or not), how much faster is your code from part b over part a?

```
for(i=0;i<n;i++)
        if(a[i]<b[i]) x++;
        else y++;
```

For the comparison, use a slt instruction. Assume register x1 stores the starting address of a, x2 stores the starting address of b, that x, y, n and i are stored in x10, x11, x12 and x13 respectively with x13 (i) already initialized to 0. Do not store x10 or x11 back to memory in your code, assume that happens later.

Answer:

a.

```
loop:   lw      x3, 0(x1)
        lw      x4, 0(x2)
        slt     x5, x3, x4          // a[i] < b[i] ?
        beq     x5, x0, else
        addiw   x10, x10, 1         // if clause
        j       next
else:   addiw   x11, x11, 1         // else clause
next:   addiw   x13, x13, 1         // i++
        addiw   x1, x1, 4
        addiw   x2, x2, 4
        bne     x13, x12, loop
```

b.

```
loop:   lw      x3, 0(x1)
        lw      x4, 0(x2)
        slt     x5, x3, x4
        addiw   x10, x10, 1         // if clause always executes
        bne     x5, x0, next        // branch around else clause
        subiw   x10, x10, 2         // undo x++
        addiw   x11, x11, 1
next:   addiw   x13, x13, 1
        addiw   x1, x1, 4
        addiw   x2, x2, 4
        bne     x13, x12, loop
```

c. The original code executes 10 instructions if the if clause is taken and 9 instructions if the else clause is taken, or 10 * .8 + 9 * .2 = 9.8 instructions per iteration. The revised code executes 9 instructions if the if clause is taken and 11 instructions if the else clause is taken, or 9 * .8 + 11 * .2 = 8.4. The revised code has a speedup of 9.8 / 9.4 = 1.043.

d. Both sets of code have data hazards with 1 cycle stalls after the second lw and the original code has one after the slt (but not the revised code). The first set of code has 3 branches if the if clause executes and 2 branches if the else clause executes. The revised code has 2 branches no matter which clause is executed. So the original code takes 15 cycles to execute the if clause and 13 cycles to execute the else clause,

or 15 * .8 + 13 * .2 = 14.6 cycles. The revised code takes 12 cycles to execute the if clause and 14 cycles to execute the else clause, or 12 * .8 + 14 * .2 = 12.4 cycles. The revised code offers a 14.6 / 12.4 = 1.177 speedup.

8. When the compiler speculates to reduce the number of branch penalties (as we saw for instance in problem #7 part b), we introduce a new problem in that the speculated instruction may cause an exception. One example of such an exception is shown in the example on page H-30 where a speculated load (sLD) may cause a memory violation and so, in RISC-V, an instruction called SPECCK is introduced.
    a. Explain what the SPECCK instruction does.
    b. Can the speculated code in #7 cause an exception? Explain. If yes, can the SPECCK instruction help us in this case?
    c. Aside from any exception you may have found in part b, and a memory violation that might occur from a speculated load, are there other situations that might cause exceptions from speculated instructions? Explain.

    Answer:
    a. In the code on page H-30, we speculate that we will want to load 0(R2). But the memory access could result in a memory violation. If the speculation were incorrect, that exception would never have arisen. SPECCK tests the memory access to see if it causes an exception. If in fact the speculation is incorrect and SPECCK determines there would be an exception from the load instruction, that exception is ignored. Therefore, it makes the speculated instruction (the load), "safe".
    b. The speculated instruction is addiw x10, x10, 1. The only reason that this could cause an exception is if it overflows x10. However, the speculation itself is not causing that exception, it is just poor programming (that is, the programmer did not test x before incrementing it to see if x could safely be incremented – which is common in programming, we rarely test for overflow). So, literally, yes, the speculated code could cause an exception but the exception would arise anyway the next time the if clause executes. In any event, SPECCK would not help us because this instruction only deals with speculations of memory-related instructions. Had there been an equivalent SPECCK instruction to test for any exception, then yes, it could have prevented the exception on a miss-speculation.
    c. Referring back to table C.27, the sources of exceptions are memory violations and page faults in the IF and MEM stages, illegal/undefined op codes and arithmetic exceptions. So any speculated load or store could result in either a memory violation, misaligned access or page fault, and the SPECCK instruction can handle those for us, and any ALU operation could result in an arithmetic exception. In part a, we had to worry about overflow, but we could also have division by 0, square root of a negative number, or an attempt to put a negative number in an unsigned location. But in general, we are talking about just MEM-exceptions during loads/stores and EX-exceptions during arithmetic operations. We would assume that the compiler never produces an illegal/undefined op code.

9. Assume the fmult.d takes 7 cycles to compute (meaning 6 cycles between the fmult.d and the fsd, assume the fmult.d and fsd do not enter MEM together). Unroll the loop and then schedule it for the EPIC architecture, showing the bundle template type and the instructions in slots 0, 1 and 2. Select the bundle type and schedule to minimize the number of cycles to execute the code (similar to figure H.8b on page H-37). Next, redo this assuming that the fmult.d takes 4 cycles to compute. For both versions, compute the CPI. NOTE: you do not need to include operands in your unrolled and scheduled code.

```
Loop: fld      f0, 0(x1)
      fld      f1, 0(x2)
      fmult.d  f2, f0, f1
      fsd      f2, 0(x3)
      addiw    x1, x1, 8
      addiw    x2, x2, 8
      addiw    x3, x3, 8
      subiw    x4, x4, 1
      bne      x4, 0, Loop
```

Answer: First, we need to identify the type of each instruction to see which slot it can fit into. There are 4 types of instructions, memory access (M), FP (F), integer ALU (A) and branch (B). The A type can go into an I-unit or M-unit, the F type must go into an F-unit, the B type must go into a B-unit and the M type must go into an M-unit for a bundle. Next, we need to unroll the loop, group instructions into bundles and schedule them. Since the fmult.d requires 7 cycles to compute, we need to have enough duplication of instructions to schedule 6 instructions between an fmult.d and its fsd. We will unroll the loop for a total of 7 iterations to fill those cycles. This gives us two cycles of fld instructions until the first data become available and then five cycles of fld instructions and an fmult.d. At this point, there are still two more cycles of fmult.d's to issue, which can also contain at least one integer ALU operation. These are followed by the fsd's and the bne. The initial two cycles are just two flds, using template 8. These are followed by 5 cycles of two flds and the fmult.d, using template 14.

#	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
14	fld	fld	fmult.d
14	fld	fld	fmult.d
15		fmult.d	
15		fmult.d	
16	fsd	addiw	
16	fsd	addiw	
16	fsd	addiw	
16	fsd	subiw	
9	fsd		
18	fsd	bne	

9	fsd	

CPI = 16 / 33 = .485

If the fmult.d only takes 4 cycles to compute, we can still unroll the loop the same number of times, but now the bundles become more concise as we can move the fsd's up.

#	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
14	fld	fld	fmult.d
14	fld	fld	fmult.d
12	fsd	fmult.d	addiw
12	fsd	fmult.d	addiw
16	fsd	addiw	
16	fsd	subiw	
9	fsd		
18	fsd	bne	
9	fsd		

CPI = 14 / 33 = .424 (ideal CPI = .333)