Chapter 4 sample problems.

1. Explain what each of the following enhancements does to performance beyond having an OOC superscalar. For each, suggest an application or type of problem in which the enhancement helps over a version of the processor without the enhancement. Apply each enhancement on top of the previous enhancement (that is, the enhancement for b also has enhancement a).
    a. Pipelined functional units
    b. Lanes of functional units
    c. Predicate registers and vector-mask control

    Answer:
    a. Multiple instructions of the same type can execute in an overlapped fashion rather than waiting for the previous operation to conclude. For FP operations, particularly multiplications and divisions, this can speed up processing. This in turn allows multiple loop iterations to be active at a time (whether by dynamic or static loop unrolling). An example application that can benefit from this is a simple for loop computation like
        for(i=0;i<n;i++) a[i]=a[i]*b[i];
        If the multiplication takes say 7 cycles to compute, the multiplier is still in use by the time the second iteration comes around.
    b. This enhancement provides the processor with the ability to issue multiple instances of the same operation to be handled in parallel. If we have enough lanes, a vector operation could potentially be executed without a loop because all iterations are issued across the lanes in one cycle. For instance, if in the above loop n is 16 and we have >= 16 lanes, then each lane receives a[i] * b[i] for some i and they all execute in parallel and then there is no need for any loop mechanism at all.
    c. This enhancement allows the lanes to control whether to execute or not based on some condition. For example: for(i=0;i<n;i++) if(b[i]!=0) a[i] = a[i] / b[i];
        Here, the predicate register is set so that p[i] = true if b[i]!=0. Then a lane executes a[i] / b[i] if its predicate value is true. If false, that lane does nothing.

2. Do question 4.9.b and 4.9.c on page 360. Use the notation a_re, b_re, a_im and b_im to indicate the base locations of the four arrays (e.g., use a_re(x1) for an address rather than 500(x1)). Assume the loop iterates a number of times that is a multiple of the mvl so that you do not need to employ strip mining (as mentioned in the problem). Skip the portion of 4.9c on number of clock cycles per complex result.

    Answer:

```
        addiw   x1, x0, 0
Loop:   vld     v1, a_re(x1)
        vld     v2, b_re(x1)
        vmul    v3, v1, v2
        vld     v4, a_im(x1)
        vld     v5, b_im(x1)
        vmul    v6, v4, v5
        vmul    v7, v1, v5
        vsub    v8, v3, v6
        vadd    v9, v3, v6
        vst     v8, c_re(x1)
        vst     v9, c_im(x1)
        addiw   x1, x1, 4
        bne     x1, x2, Loop
```

The convoys consist of instructions that do not share the same functional unit. So we will have to place each vld and vst and each vmul in separate convoys. This leads to the following:

```
vld     (a_re)
vld     (b_re)  vmul (v3)
vld     (a_im)
vld     (b_im)  vmul (v6)
vmul    (v7)    vsub
vadd    vst (c_re)
vst (c_im)
```

Or 7 chimes.

3. Given the following C code, and assuming arrays a, b and c are both arrays of doubles
   a. rewrite the code first in normal RISC-V and then in RV64V. Assume n is 64 and that the vector registers are large enough to support that many doubles at a time. d is a scalar stored in f0, and arrays a, b and c are stored at memory locations which are stored in registers x1, x2 and x3 respectively.

```
for(i=0;i<n;i++)
    a[i] = b[i] * c[i] + a[i] * d;
```

   b. Given the RV64V code from part a, show how the instructions would appear as convoys. Next, can the code be rearranged to reduce the number of convoys, if so, show this and if not, explain why. NOTE that the vsetdcfg and vdisable should not count as part of the code for the convoys.
   c. Given the convoys from part b, and assuming there are 64 elements to the arrays a, b and c from part a, compute the approximate execution time for the RV64V code from part b.

Answer:

a. RISC-V code

```
        addiw   x4, x0, 512     // repeat until x1 equals 512
Loop:   fld     f1, 0(x1)
        fld     f2, 0(x2)
        fld     f3, 0(x3)
        fmult.d f4, f2, f3
        fmult.d f5, f1, f0
        fadd.d  f6, f4, f5
        fsd     f6, 0(x1)
        addiw   x1, x1, 8
        addiw   x2, x2, 8
        addiw   x3, x3, 8
        bne     x1, x4, Loop
```

RV64V

```
vsetdcfg        3*FP64
vld             v1, x1
vld             v2, x2
vld             v3, x3
vmul            v2, v2, v3
vmul.vs         v1, v1, f0          // multiply vector by scalar
vadd            v1, v1, v2
vst             v1, x1
vdisable
```

b. A convoy will include all instructions that do not cause structural hazards. We have 3 vld's, 2 vmul's, a vadd and a vst. The three 3 vld's all require the same hardware, so would all cause structural hazards, as would the two vmul's. This gives us convoys as follows:

```
vld
vld
vld     vmul
vmul    vadd    vst
```

It is unlikely that the second vmul and the vadd would cause a structural hazard but if it were the case, then the vadd and vst would appear on a separate line. We can rewrite the original code as follows (excluding the vsetdcfg and vdisable):

```
vld             v2, x2
vld             v3, x3
vmul            v2, v2, v3
vld             v1, x1
vmul            v1, v1, f0
vadd            v1, v1, v2
vst             v1, x1
```

Giving us the following breakdown of convoys:

```
vld
vld     vmul
vld     vmul    vadd
vst
```

Or we might group the vadd and vst together. Unfortunately, this does nothing to improve our number of convoys. In fact, both give us the minimum number as we have 4 total load/store operations, and each of these must be in a separate convoy. So there is nothing we can do to reduce the number of convoys.

c. We see from part b that there are 4 convoys taking 4 chimes. Given that the operations execute at approximately 1 cycle per array element, the code takes 4 * 64 = 256 clock cycles, not including initialization time. This is in fact misleading because of stalls that will arise between the second vmul and the vadd and between the vlds and their corresponding vmuls.

4. Given the following RISC-V code and assuming 16 lanes, how many clock cycles would be required to run the entire set of code ignoring start-up time? The loop will iterate 60 times. Assume the only stalls arise between the fmult.d and the fsd and it is 2 cycles.

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

Answer: Including the 2 cycle stalls, each loop iteration takes 7 cycles to complete. The 60 loop iterations can execute over the 16 lanes in 4 total iterations. So the total time to execute the code is 7 * 4 = 28 cycles.

5. A vector processor consists of 16 lanes within each is an ALU of pipelined functional units. The maximum size for a vector register is 16 elements. The FP portion of the ALU has the following functional units: FP adder (4 cycles to compute), FP multiplier (7 cycles to compute) and FP divider (10 cycles to compute), there is also a "simple" FP unit that can be used to initialize a value (that

is, be set to an immediate datum) or shift/rotate an FP value's mantissa, or double or halve an FP value (each of these simple operations takes 1 cycle).

a. How many total operations could potentially be running at one time in this processor?
b. A set of code consists of a loop in which an FP array is modified by adding a scalar, d, to each element (e.g., the loop body is a[i]=a[i]+d). Assume the array contains 2000 values. How many loop iterations are required to perform this calculation and what is the number of operations that are performed on the last loop iteration (that is, apply strip mining)?

Answer:

a. With 16 lanes, and with the longest latency being the divider, we could potentially have 10 * 16 or 160 operations running at a time.
b. Each lane will have its own vector register available, whose maximum size is 16 vector elements. With 16 lanes, this means that 16 * 16 = 256 iterations of the C loop can be handled in one iteration of the vector code's loop. With 2000 total elements, this means 2000 / 256 = 7.8125 loop iterations are required. For the first 7 loop iterations, 256 elements are computed. The last iteration then requires .8125 * 256 = 208 operations (the strip mining case).

6. Convert the following C code first into RISC-V code and then into RV64V code using a predicate register to handle the if statement. Next, revise both of your sets of code adding the else clause. Use x1 as the loop index and assume it stores 0, x2 stores 64, x3 is the pointer to a and x4 is the pointer to b, where a and b are int arrays. Assume there is an instruction vpnot which flips every bit in a predicate register.

```
for(i=0;i<64;i++)
   if(a[i]>b[i]) a[i]=a[i]+b[i];
   // else b[i]=b[i]+a[i];
```

Answer:

RISC-V:

```
top:    beq     x1, x2, out
        lw      x5, 0(x3)
        lw      x6, 0(x4)
        slt     x7, x6, x5          // set x7 if b[i] < a[i]
        beq     x7, x0, bottom      // if x7 is 0, skip the if clause
        addiw   x5, x5, x6
        sw      x5, 0(x3)
bottom: addiw   x3, x3, 4
        addiw   x4, x4, 4
        addiw   x1, x1, 1
        j       top
out:
```

RV64V:

```
        vsetdcfg        2*I32
        vsetpcfgi       1           // we need 1 predicate register
        vld             v0, x3
        vld             v1, x4
        vplt            p0, v1, v0  // set p[i] if b[i] < a[i]
        vadd            v0, v0, v1  // for those where p[i] is 1, do a[i] = a[i] + b[i]
        vst             v0, x3      // when done, store a[i] back to memory
        vdisable
        vpdisable
```

```
RISC-V:
top:    beq     x1, x2, out
        lw      x5, 0(x3)
        lw      x6, 0(x4)
        slt     x7, x6, x5
        beq     x7, x0, else      // if b[i] < a[i], go to else clause
        addiw   x5, x5, x6
        sw      x5, 0(x3)
        j       bottom
else:   addiw   x6, x6, x5
        sw      x6, 0(x4)
bottom: addiw   x3, x3, 4
        addiw   x4, x4, 4
        addiw   x1, x1, 1
        j       top
out:

RV64V:
        vsetdcfg        2*I32
        vsetpcfgi       1
        vld             v0, x3
        vld             v1, x4
        vplt            p0, v1, v0      // same condition as above
        vadd            v0, v0, v1      // and same if clause
        vst             v0, x3          // store the result back to a
        vpnot           p0              // flip every bit of p
        vadd            v1, v1, v0      // for all p[i] now true, do the else clause
        vst             v1, x4          // store the result back to b
        vdisable
        vpdisable
```

7. Provide PTX code for the following C code assuming x, y and z are all double precision arrays.

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

Answer: The loop is removed and handled by the SIMD processors. The code becomes

```
        ld.global.f64 RD0,[x]
        ld.global.f64 RD1,[y]
        ld.global.f64 RD2,[z]
        setp.gt s32 P1, RD0, RD1        // set the predicate register P1[i] if x[i] > y[i]
        @!P1, bra ELSE                  // for all P1[i] that are false, branch to ELSE
        mov.f64 RD0, RD2                // for the remainder, do x[i] = z[i]
        st.global.f64 [x], RD0          // store results of x back to memory
        @P1, bra DONE                   // and branch to DONE
ELSE:   mov.f64 RD1, RD2                // for those where P1[i] are false, do y[i] = z[i]
        st.global.f64 [y], RD1          // and store results of y back to memory
DONE: …
```

8. Consider the following C code

```
for(i=0;i<n;i++)
        sum=sum+a[i]*b[i];
```

a[i]*b[i] is known as a dot product in that we are computing pairwise products of two vectors. But this code specifically is accumulating each product into a single sum. This is known as a reduction problem.

   a. Why is a reduction problem a challenge to execute efficiently on a SIMD or GPU?
   b. Can you come up with a solution that improves the execution efficiency of such code (even if you aren't taking full advantage of the parallelism available)? If there are n elements for arrays a and b, approximately how many operations are required to solve this problem using your solution?

Answer:

   a. The multiplication can be done in parallel across lanes of the SIMD and GPU. But because every product is added to one sum variable, all sum operations must be serialized and thus we obtain no performance increase over a single functional unit (other than being able to produce all of the multiplications quickly).
   b. We can store the products in another vector and then use parallel functional units to add together pairs of values. Those sums are then added together in additional pairs. For instance, we might do sum[0] = product[0] + product[1], sum[2] = product[2] + product[3], sum[4] = product[4] + product[5], etc. We follow this with [0] = sum[0] + sum[2], sum[4] = sum[4] + sum[6], etc followed by sum[0] = sum[0] + sum[4] and sum[8] = sum[8] + sum[16], etc. Eventually, our last addition will be sum[0] = sum[0] + sum[n/2]. This is known as a tournament algorithm. All n multiplications can be handled in parallel. We then perform n/2 additions in parallel, followed by n/4 additions in parallel, followed by n/8 additions in parallel, etc. We require log n + 1 parallel operations (the +1 is the original parallel multiplications). If our array had 128 elements and assuming we have 128 lanes, the original code requires 256 operations (128 multiplications, 128 additions). If we executed the multiplications in parallel and the additions in sequence, it would take 129 operations. With a tournament algorithm, it would take 8 operations (1 parallel multiplication and 7 parallel additions where the 1st parallel addition would use 64 lanes, the 2nd would use 32 lanes, the third would use 16 lanes, etc, and the last would use 1 lane).