Appendix A sample problems.

1. Do problem A.1 on page A-47 (note: use an average of the astar and gcc benchmarks for instruction frequencies assuming 60% of all branches are taken).

   Answer: gcc/astar has the following breakdown of instructions: loads: 22.5%, stores: 14.5%, branches: 19%, jumps: 3%, ALU operations: 41%.

   CPI = 22.5% * 5 + 14.5% * 3 + 19% * 60% * 5 + 19% * 40% * 3 + 3% * 3 + 41% * 1 = 2.86.

2. RISC-V uses a 5-stage fetch-execute cycle (as covered in appendix C.1, but covered in the slides 20-24 under the appendix A power point notes). Because the base-displacement computation has to be computed first, the fetch-execute cycle has the EX stage before the MEM stage. Thus, if base-displacement is used, the memory address is computed in the EX stage and that address is sent to the data cache in the MEM stage for a load or store. For non-loads/store instructions, the MEM stage can be skipped. Let's consider making the following change to this architecture. Rather than having the EX stage first, the MEM stage comes first. This allows us to have ALU operations that first retrieve a single datum from memory. For this to work, the address must be a direct memory reference and not a base-displacement address. Let's assume this change reduces the number of loads (because now some loaded values can be handled directly in ALU operations that fetch the datum in their MEM stages). Assume 30% of all ALU operations use a datum loaded immediately before the ALU operation where the loaded datum is only used once, in that ALU operation. This allows us to remove some of the loads. The downside to this change is that to use base-displacement requires two separate instructions: one to compute the effective address and one to perform the memory reference. Let's assume 40% of all loads/stores use base displacement. Using the astar instruction mix, determine whether this change will result in a speedup or slowdown.

   Answer: The change being made here swaps the EX and MEM stages. The advantage is that we can remove some loads when those loaded values are used once, in the next ALU operation AND the loaded value uses a direct memory reference rather than base-displacement. The disadvantage is that loading/storing anything using base-displacement now takes 2 instructions. So in some cases, we remove a load instruction and in some cases, we add an ALU instruction. How often do we remove a load and how often do we add an ALU operation? The astar benchmark has 46% ALU operations and 34% loads/stores. We are told that 30% of all ALU operations use the loaded value once, so that means 46% * 30% = 13.8% of the time we can remove a load. On the other hand, of the 34% loads/stores, we are told 40% use base-displacement requiring that we add an ALU instruction which adds 34% * 40% = 13.6% new instructions. This revised architecture offers a very minor speedup of 13.8 / 13.6 = 1.015 or 1.5%. Its probably not worthwhile because this change will have other impacts.

3. Do problem A.7a on page A-49.

   Answer:

```
        ld      x1, 5000(x0)        // x1 stores C
        addw    x2, x0, x0          // x2 stores i, initialized to 0
        addiw   x3, x0, 100         // x3 stores 100, to determine the end of the loop
        addw    x4, x0, x0          // x4 will store the byte offset to arrays A and B
loop:   beq     x2, x3, exit        // leave loop once i==100
        ld      x5, 3000(x4)        // x5 = B[i]
        add     x6, x5, x1          // x6 = B[i] + C
        sd      x6, 1000(x4)        // A[i] = B[i] + C
```

```
        addiw   x2, x2, 1           // i++
        addiw   x4, x4, 8           // set byte offset to next long int in A and B
        j       loop
exit:   …
```

NOTE: the book says i is stored at memory location 7000 but we don't actually need to store it in memory.

The code has 4 instructions prior to the loop and 7 instructions in the loop, which iterates 100 times, so the number of instructions that executes is 7 * 100 + 4 = 704. The code contains 1 memory reference prior to the loop and 2 in the loop, so 2 * 100 + 1 = 201 memory references. There are 11 total instructions, each is 32 bits (4 bytes) so the code takes 44 bytes.

4. Convert the following C code to RISC-V code:

```
sum=0;
for(i=0;i<n;i++)
   sum+=a[i];
average=(float)sum/n;
```

Assume all variables are 32-bit int values except average which is a 32-bit float, where n is stored at location 10000, array a starts at location 20000, and average is stored at memory location 10004.

Answer:

```
        addw    x1, x0, x0      // x1 = sum
        addw    x2, x0, x0      // x2 = i (initialized to 0)
        lw      x3, 10000(x0)   // x3 = n
        addiw   x4, x0, 20000   // x4 is the byte offset into a
loop:   beq     x2, x3, exit    // leave loop once i==n
        lw      x5, 0(x4)       // x5 = a[i]
        addw    x1, x1, x5      // sum+=a[i]
        addiw   x4, x4, 4       // increment the byte offset to the next array element
        addiw   x2, x2, 1       // i++
        j       loop
exit:   fcvt.w.s  f0, x1        // f0 = (float)sum
        fcvt.w.s  f2, x3        // f2 = (float)n
        fdiv.s  f1, f0, f2      // f1 = sum/n
        fsw     f1, 10004(x0)   // store average
```

5. Using RISC-V, write the code to obtain the minimum and maximum values in a 32-bit int array. Assume the array is stored at memory location 10000 and contains 100 elements. Store the two values in locations 10400 and 10404 respectively.

Answer:

```
        lw      x1, 10000(x0)       // x1 = a[0]
        addw    x2, x1, x0          // x2 = min
        addw    x3, x1, x0          // x3 = max
        addiw   x4, x0, 1           // x4 = i (set to 1 since we already did a[0])
        addiw   x5, x0, 4           // x5 = byte offset, we've already processed a[0]
        addiw   x6, x0, 100         // x6 = n (100)
loop:   beq     x4, x6, exit        // leave loop when i==100
        lw      x1, 10000(x5)       // x1 = a[i]
        slt     x7, x1, x2          // a[i] < min?
```

```
        jeq     x7, x0, next        // a[i] not < min, go to next comparison
        add     x2, x1, x0          // reset min to a[i]
        j       next2               // otherwise branch to bottom of loop since min != max
next:   slt     x7, x3, x1          // max < a[i]?
        jeq     x7, x0, next2       // max not < a[i], go to bottom of loop
        add     x3, x1, x0          // reset max to a[i]
next2:  addi    x4, x4, 1           // bottom of loop, i++
        addi    x5, x5, 4           // increment x5 to next array location in a
        j       loop
exit:   sw      x2, 10400(x0)       // store min
        sw      x3, 10404(x0)       // store max
```

6. Do problem A.8a on page A-49 – A-50.

Answer:

```
        addw    x1, x0, x0          // x1 = p (initialized to 0)
        addiw   x2, x0, 8           // x2 is n (8)
        addiw   x3, x0, 8           // x3 is the byte offset into the arrays
        addi    x10, x0, 9798       // load constants into registers
        addi    x11, x0, 19235      // unlike x1, x2, x3, these are all longs (64 bits)
        addi    x12, x0, 3736
        addi    x13, x0, 32768
        addi    x14, x0, -4784
        addi    x15, x0, -9437
        addi    x16, x0, 4221
        addi    x17, x0, 128
        addi    x18, x0, 20218
        addi    x19, x0, -16941
        addi    x20, x0, -3277
loop:   beq     x1, x2, exit        // leave loop when p==n
        ld      x4, 1000(x3)        // x4 = R[p]
        ld      x5, 2000(x3)        // x5 = G[p]
        ld      x6, 3000(x3)        // x6 = B[p]
        mul     x7, x4, x10         // compute Y[p]
        mul     x8, x5, x11
        mul     x9, x6, x12
        add     x7, x7, x8
        add     x7, x7, x9
        div     x7, x7, x13
        sd      x7, 4000(x3)        // store result in Y[p]
        mul     x7, x4, x14         // compute U[p]
        mul     x8, x5, x15
        mul     x9, x6, x16
        add     x7, x7, x8
        add     x7, x7, x9
        div     x7, x7, x13
        add     x7, x7, x17
        sd      x7, 5000(x3)        // store result in U[p]
        mul     x7, x4, x18         // compute V[p]
        mul     x8, x5, x19
        mul     x9, x6, x20
```

```
        add     x7, x7, x8
        add     x7, x7, x9
        div     x7, x7, x13
        add     x7, x7, x17
        sd      x7, 6000(x3)        // store result in V[p]
        addiw   x1, x1, 1           // increment loop index
        addiw   x3, x3, 8           // increment byte offset by 8 (dealing with long ints)
        j       loop                // redo loop
exit:   …
```

7. Do problem A.9a-c on page A-50.

   Answer: Instruction length = 14 bits, 6 bits for an address

   a. 3 two-address instructions: use op codes 00, 01, 10 leaving 12 bits for the addresses, so we can accommodate the two 6-bit addresses. 63 one-address instructions: all start with op code 11 followed by 6 bits for the specific instruction, so these op codes are 11000000…11111110, followed by 6 bits for one address. 45 zero-address instructions: these op codes all start with 11111111 and use the rest of the bits to denote the specific op code, that is, these instructions are 11111111000000…11111111101100
   b. The 3 two-address instructions are the same as in part a, but for the 65 one-address instructions, we would need 7 bits for these op codes, so they all start with 11 and then range from 110000000 to 111000001, leaving 5 bits for the address, so we don't have enough space.
   c. The 3 two-address instructions require two bits for the op code leaving bit 11 for one and zero operand instructions. We have some number of one-operand instructions, call that n. These n instructions require 2 op code bits for 11 and 6 bits for the operand, leaving 6 bits for the rest of the op code portion. However, one of these patterns must be reserved for the zero operand instructions. Therefore, we can have up to $2^6 - 1 = 63$ one-operand instructions. Note this is the same as in part a where we can see that it doesn't matter how many zero-address instructions (up to 2^6 since we have 6 remaining op code bits).