CS24 Assignment 1

This homework requires a number of supporting files, provided in the archive **`cs24hw1.tgz`** on the course website. You can download it from there, and then use this command to expand it in your working directory:

```
tar -zxvf cs24hw1.tgz
```

The archive contains a top-level directory **`cs24hw1`**, in which all other files are contained.

When you have completed the assignment, you must submit all of your work as a single archive file, following the instructions outlined here. *Failure to do so will result in point deductions!* Always make your graders' lives easy. ☺

Go ahead and do your work within the **`cs24hw1`** directory. Then, when you are done, you can re-archive this directory into a tarball and submit the resulting file:

*(From directory containing **`cs24hw1`**; replace **`username`** with your IMSS or CMS username.)*
```
tar -czvf cs24hw1-username.tgz cs24hw1
```

Then, submit the resulting file through the course website.

Make sure to follow the specified filenames in your work. Make sure the tarball you create doesn't impose strict permissions. If your submission is difficult for the TAs to work with, points will be deducted. If you have any questions or concerns on how to package up your submission, feel free to ask Donnie or a TA! We will be happy to help.

Part 1: Integers, and C/`make` Warm-Up (20 points)

This section of the assignment uses the files in the directory **`cs24hw1/bits`** of the archive available on the website.

1. **Using C's shift and bit-manipulation operators, write a C function that takes an unsigned integer *n*, and returns a count of the number of bits in *n* that are 1. *(10 points)***

   There is a file **`onebits.c`** which contains some simple code to wrap your function; find the function **`count_onebits()`** and fill in the implementation.

   Once you have completed this function, test your program with various values and make sure that it works correctly. You can use the **`make`** program to build your code:

```
[user@host:~]> make
gcc -c onebits.c
gcc -o onebits onebits.c
[user@host:~]> ./onebits 3 4 7
Input:  3   One-bits:  2

Input:  4   One-bits:  1

Input:  7   One-bits:  3

[user@host:~]>
```

Notice that **`make`** takes care of all the steps to build your program, based on the contents of the **`Makefile`** in this directory.

2. **Here is an interesting expression: `n = n & (n - 1)` *(10 points)***

   What does this expression do to **`n`**? How does it work? How might it provide a faster way to count one-bits in a value? Write your answers to these questions in a file **`problem2.txt`** (in the **`bits`** directory) and include this file in your submission. Be specific in your answers!

   Once you have figured out these answers, make a copy of **`onebits.c`** called **`faster_onebits.c`**, and reimplement **`count_onebits()`** to use the above expression to count one-bits instead of using shifts and bit-masks.

   Update the **`Makefile`** to also build this new source file into a binary called **`faster_onebits`** (just duplicate and modify the rules for building **`onebits`**, if you haven't worked with makefiles before).

For this section, we will be grading these files:

- **`bits/onebits.c`**
- **`bits/faster_onebits.c`**
- **`bits/Makefile`**
- **`bits/problem2.txt`**

Part 2: Floating-Point Data Representation (30 points)

This section of the assignment uses the files in the directory **`cs24hw1/floats`** of the archive.

1. **Work through all four parts of CS:APP2e Practice Problem 2.46, pp.102-103.[1]** ***(5 points)***

   This is a practice problem so the answer is in the back of the chapter, but for your own sake, *don't look at the answer until you have worked through the entire question.* If you want to check your answers when you are finished, this is fine, but you need to include all your work to get full credit for the problem.

   Put your answers and work in a file **`floats/problem1.txt`**

2. **Floating-point summation *(25 points)***

   In the **`floats`** directory you will find a program called **`fsum`**, which has a simple purpose – add up a sequence of floating point values and print out the sum. The program's input must specify the number of floating-point values, and then the values themselves. For example:

```
3
3.3522
467.2158
0.003199
```

[1] If you are using the first edition (CS:APP), this is Practice Problem 2.32 on pp. 82-83.

Several data files are provided for you to use, and can be fed into this program using input redirection:

```
[user@host:~]> ./fsum < f1.txt
... program output ...
```

This is where it starts to get strange. One would expect that if you are summing a sequence of numbers, the result should be independent of the order the numbers are added. However, this is not the case with floating point numbers. The **`fsum`** program demonstrates this by adding up its inputs in three different orders: the first is the order that values appear in the input, the second is in increasing order of magnitude, and the third is in decreasing order of magnitude.

In a file **`floats/problem2.txt`**, answer these questions: ***(10 points)***

Why are these results different? Of the three approaches that **`fsum`** uses, which is the most accurate, and why? Be specific in your answers.

What kinds of inputs would also cause the "most accurate" approach to exhibit large errors? *(Hint: think about what causes the floating-point sum to be inaccurate, and then describe the characteristics of an input data-set that would cause even the "most accurate" version to also suffer from such inaccuracies. Also, the size of the data-set can have just as much impact on the accuracy of the results as the specific values in the data-set.)*

Next you must devise and implement a more accurate methodology for computing the sum of a series of floating-point numbers. In the **`fsum.c`** program, complete the **`my_fsum()`** function to compute the sum of the inputs using your own methodology. *You must also clearly explain your approach in the comments for the function, and why your approach will yield a more accurate sum, to receive full credit for this problem.* ***(15 points)***

Hints:

Your solution should only use **`float`**. Do not use **`double`**; that isn't the point of the question.

You can use whatever approach you want, but it should yield a substantial improvement in accuracy. There are several approaches that produce high-accuracy floating-point sums (including one that is as precise as can be achieved with floating-point!); feel free to do some research if you want to find an interesting one.

An easy approach is to introduce a compensation term into the summation. Note these two expressions:

$$nextsum := sum + values[i]$$
$$c := (nextsum - sum) - values[i]$$

The parentheses in the second expression are important. Algebraically, what should c always be equal to? In what situations would it deviate from this expected value? How can we take advantage of this to improve accuracy? (If you use this approach in your solution, make sure your explanation includes answers to these questions.)

(Also, if you use this approach, you should apply the compensation on each iteration; don't apply it once at the end, or else you really haven't improved things hardly at all.)

If you are curious about the "precise floating-point summation" approach, you should look at the Python **`fsum()`** function in the **`math`** module. If you implement a correct, working version of this approach in your submission, you will receive some bonus points.

For this section, we will be grading these files:

- **`floats/problem1.txt`**
- **`floats/problem2.txt`**
- **`floats/fsum.c`**

Part 3: Programmable Branching Processor (50 points)

In the directory **`cs24hw1/proc`**, you will find the implementation of a simple processor similar to the one we discussed in class. (*Note that the opcode values are different from those in the lecture slides!*) There are two versions of the processor, one that includes branching and one that does not. We will use the version with branching, and we will write a simple program for this processor.

There is an appendix to this assignment that describes the branching processor; please refer to it for more details about what operations are supported, how instructions are encoded, and so forth.

The first step is to build the processor on your computer. When you run **`make`**, you will see that it executes a number of commands and eventually generates a file **`branching_run`**. This is the program you will use to run code on the branching processor.

However, the implementation for this processor is not complete! You will need to fill in the missing pieces before you can use the processor.

1. **Complete the implementation of the processor's Arithmetic/Logic Unit.** ***(15 points)***

   Find the function **`alu_eval()`** in the file **`alu.c`**. This is the function that emulates the ALU in the processor. It receives two data inputs (**`alu->in1`** and **`alu->in2`**), as well as the operation (aka "opcode") of the operation to perform (**`alu->op`**). Based on these inputs, the ALU needs to compute various results and generate an output for the rest of the processor to use.

   You should use a **`switch`** statement to implement this code:

   ```
   switch (aluop) {

   case ALUOP_ADD:
       result = ... ;  /* Implement ALUOP_ADD. */
       break;

   case ALUOP_INV:
       ...
   }
   ```

   You can use standard C operators to implement each ALU operation. (For example, you don't need to implement twos-complement negation as invert/increment; just use unary negate.)

   Make sure to comment each case, at least briefly, as to its purpose! You don't need to write anything detailed unless it's really warranted, but you should comment each operation.

   Make sure that if the opcode is not recognized by the ALU, then the ALU simply outputs zero. The ALU should recognize all **`ALUOP_*`** values specified in **`instruction.h`**, *except for* **`ALUOP_BNZ`** and **`ALUOP_DONE`**, which the ALU is simply not responsible for.

   Once you have completed your implementation, you can use the **`testalu`** program to verify your work. (**`make -f Makefile.branch testalu`** will build this program.)

2. **Complete the implementation of the processor's branching support.** ***(10 points)***

   The other part of the processor to complete is the logic for performing branching in this processor. You will need to add code to these files:

- **`branch_unit.c`** – complete the **`branch_test()`** function
- **`branching_program_counter.c`** – complete the **`nextPC()`** function

The details of what you need to do are specified in each of these files, so just open them up and you should have all the instructions you need.

Once you have completed these functions, try out the branching processor with a simple program that computes the $(i + 1)^{th}$ value of the Fibonacci sequence. The instructions are contained within **`ifib.ibits`** (".ibits" stands for "instruction bits"), and **`ifib_5.rbits`**, **`ifib_10.rbits`**, and **`ifib_12.rbits`** are three different sets of register inputs you can try with this program (".rbits" stands for "register bits"). For this implementation of the Fibonacci sequence, $fib(0) = 0$, and $fib(1) = 1$.

3. Write a program for this branching processor that performs integer division on two arguments. *(25 points)*

The program should take two nonnegative, signed integer arguments, the dividend and the divisor, and it should produce two signed integer results, the quotient and the remainder. (Of course, since neither input is negative, the quotient and remainder should also never be negative.) Each of these values is 32 bits. **Make sure your inputs and outputs are stored in these registers:**

- The dividend should be in register 0
- The divisor should be in register 1
- The quotient should end up in register 6
- The remainder should end up in register 7

- *You may store other constants in registers 2-5, as needed. You do not need to make your program generate these constants; just create your own .rbits input files that include the needed constants. Obviously, anything that is a "constant" had better actually be constant across all sets of inputs you create…*

You can implement division any way you would like, but we would recommend using addition and subtraction for computing the result. Here is a simple and slow technique, written up as pseudocode:

```
function divide(dividend, divisor) begin
    quotient = 0
    remainder = dividend

    while true do
        remainder = remainder – divisor
        if remainder >= 0 then
            quotient = quotient + 1
        else
            break
        end if
    done

    remainder = remainder + divisor
```

```
        return quotient, remainder;
    end
```

Of course, the registers themselves hold unsigned integers, and our processor doesn't support signed comparisons, so you will have to implement the conditions in a way supported by the processor. *(Hint: Think about two's complement and overflows, and what happens to various bits.)*

Store your completed program in the file **`divider.ibits`**. Then, for each of these pairs of inputs, create an input file **`div_a_b.rbits`**, run the processor to generate **`div_a_b.out`**, and check that your answers are correct:

- 15 ÷ 5 (**`div_15_5.rbits`**)
- 19 ÷ 4 (**`div_19_4.rbits`**)
- 9 ÷ 16 (**`div_9_16.rbits`**)

And finally, what happens with this pair of inputs?

- 10 ÷ 0 (**`div_10_0.rbits`**)

Briefly explain the program's behavior with this set of inputs, in a file **`div_10_0.txt`**. (If you happened to include error-checking in your implementation, what would happen if you didn't have that error-checking?)

For this section, we will be grading these files:

- **`proc/alu.c`**
- **`proc/branch_unit.c`**
- **`proc/branching_program_counter.c`**
- **`proc/divider.ibits`** *(Make sure to comment this file!)*
- **`proc/div_*_*.rbits`** *(Comment this one too!)*
- **`proc/div_10_0.txt`**

A Helpful Utility

One of the things you may find yourself doing is translating sequences of zeroes and ones into hexadecimal values. If you want to automate this process, check out **`proc/convert.c`**, which is designed specifically for this task. You will need to implement a little bit of code in this file (see the **`convert`** function), but it will allow you to create an input file that includes both whitespace and comments, and still generate the proper hexadecimal output. For example, if you have an input file **`foo.rawbits`**:

```
# My cool program.  Registers are:
#     R1 = some value
#     R2 = another value

00 1100 1 1  # Very important instruction!
10 0100 0 1  # Also very important!
01 1011 1 0
11 1111 1 1  # Meh.
```

You could do this: **`./convert < foo.rawbits`**, and the program would output:

```
# My cool program.  Registers are:
#     R1 = some value
#     R2 = another value

33    # Very important instruction!
91    # Also very important!
6e
ff    # Meh.
```

(The **<** is used to redirect the contents of the file **`foo.rawbits`** into the standard-input of the program when it runs. If you want to store the program's output into another file, you can do something like this: **`./convert < foo.rawbits > foo.ibits`**)

There is no build rule for this program, so you will have to compile it by hand, or you will need to add it to the makefile's build rules.

If you decide to use this program, you <u>will not</u> be graded on your modifications to it.

Final Submission Note

As described at the start of this assignment, once you have completed all parts of this assignment, you should re-archive your **`cs24hw1`** directory according to the specified instructions and then submit the archive file via Moodle. Make sure your submission also includes the answer-files specified in each question of the assignment!

Appendix: Branching Processor

Overview

The file **`branching_run.c`** is the top level for the branching processor simulator. The processor is assembled and its cycle-by-cycle execution occurs in **`branching_processor.c`**. Each component of the processor is encapsulated in its own C file (**`branching_program_counter.c`**, **`instruction_store.c`**, **`register_file.c`**, **`alu.c`**, **`branching_decode.c`**, etc.). The file **`branching_control.c`** provides the control unit, which sequences each of the components to implement one clock-step of the processor. The provided **`Makefile.branch`** will build the **`branching_run`** simulator, as well as a helper program **`testalu`** for exercising the ALU logic.

The simulator wires up the individual units using a bus abstraction (**`bus.c`**, **`bus.h`**). Each "bus" is a memory location (an unsigned integer) which can be set or read. Each of the functional units instantiates its own output bus, and is later given handles to the buses of all of its inputs (called "pins").

The file **`branching_processor.c`** performs the wiring between units. Wired in this way, each unit can simply read its inputs, perform its operation, and set its output. Note that the **`connect()`** routines used in **`branching_processor.c`** are not transitive; if you need to connect more than three things, make sure you use a single **`connect3()`** call, rather than two **`connect()`** calls.

The instruction format and encodings are defined in **`instruction.h`**, and are also described in the next section.

The **`branching_run`** program takes 3 arguments:

- instruction file (read)
- initial register file (read)
- final register file (written)

The instruction file and register file each consist of a series of lines with one hex number per line. The i^{th} line, should contain the value for the i^{th} entry in the instruction store and register file, respectively.

These files can contain comments to the right of the hex-values on each line. Use this feature to clearly document both your instructions and initial data.

Instructions and Instruction Formats

The instructions supported by the branching processor are described in this section. They fall into two main categories: ALU operations and branching operations. The ALU operations are as follows:

Operand	Encoding	Operation
ADD	0x00	dst ← src1 + src2
INV	0x01	dst ← ~src1
SUB	0x02	dst ← src1 – src2
XOR	0x03	dst ← src1 ^ src2
OR	0x04	dst ← src1 | src2
INCR	0x05	dst ← src1 + 1
AND	0x08	dst ← src1 & src2
SRA	0x0B	dst ← src1 >> 1; dst[31] = src1[31]
SRL	0x0C	dst ← src1 >> 1; dst[31] = 0
SLA	0x0D	dst ← src1 << 1
SLL	0x0E	dst ← src1 << 1
DONE	0x0F	Stop execution.

All of the above instructions are encoded in this format:

Bits	13:10	9	8:6	5:3	2:0
Field	op	w	src1	src2	dst

The meanings of the fields are as follows:

- op – operation to be performed (typically by ALU)
- w – write the ALU output to register file? (1 = yes, 0 = no)
- src1 – address of first ALU operand in register file
- src2 – address of second ALU operand in register file
- dst – address in register file into which the result should be stored

If a particular instruction does not require some of these fields, they should be set to 0.

The one other instruction supported is the "branch if not zero" operation BNZ.

Operand	Encoding	Operation
BNZ	0x0A	if (src1 != 0) pc ← branch_addr

Unlike the other instructions, the BNZ instruction is encoded as follows:

Bits	13:10	9	8:6	5:0
Field	op	w	src1	branch_addr

The only difference between BNZ and the other instructions is that branch_addr occupies the bits where src2 and dst normally reside.

Branching Processor Execution

Every clock tick, the branching processor performs these tasks:

```
op, w, src1, src2, dst, branch_addr = instruction_store[pc]

in1=register_file[src1]
in2=register_file[src2]

if (w==1)
        register_file[dst] ←(in1 op in2)

if (op == BNZ && src1 != 0)
        pc ← branch_addr
else
        pc ←pc +1
```

This logic is contained within the **`branching_control.c`** file, and the functions that the **`clock()`** function calls.

Branching Processor Diagram

Here is a schematic diagram of the components in the branching processor. The branching-control logic is highlighted in red.