CSE-141L: Computer Architecture Lab
Summer Session 1
Lab 1: 9-Bit Instruction Set Architecture

Due 2:00 pm PT July 12th 2021

1. Overview

In this lab, you shall design the instruction set and overall architecture for your own special-purpose (very) reduced instruction set (*RISC*) processor. You will design the hardware for your processor core in subsequent labs.

Your processor shall have 9-bit instructions (machine code) and shall be optimized for three simple programs, described below. For this lab, you shall design the instruction set and instruction formats and code three programs to run on your instruction set. Given the tight limit on instruction bits, you need to consider the target programs and their needs carefully. The best design will come from an iterative process of designing an ISA, then coding the programs, redesigning the ISA, etc. Your instruction set architecture shall feature *fixed-length instructions* 9 bits wide. Your instruction-set specification should describe:

- What operations it supports and what their respective opcodes are. For ideas, see MIPS or ARM instruction lists
- How many instruction formats it supports and what they are (in detail – how many bits for each field, and where they're found in the instruction). Your instruction format description should be detailed enough that someone could write an *assembler* (a program that creates machine code from assembly code) for it. (Again, refer to ARM or MIPS.)
- Number of registers, and how many general-purpose or specialized. All internal data paths and storage will be 8 bits wide.
- Addressing modes supported (this applies to both memory instructions and branch instructions). That is, how are addresses constructed or calculated? Lookup tables? Sign extension? Direct addressing? Indirect, as used in linked lists or ARM or MIPS to address the data_memory from reg_file contents?

For this to fit in a 9-bit field, the memory demands of these programs will have to be small. For example, you will have to be clever to support a conventional main memory of 256 bytes (8-bit address pointer). You should consider how much data space you will need before you finalize your instruction format. Your instructions are stored in a separate memory, so that your data addresses need be only big enough to hold data. Your data memory is byte-wide, i.e., loads and stores read and write exactly 8 bits (one byte). Your instruction memory is 9 bits wide, to hold your 9-bit machine code, but it can be as deep as you need to hold all three programs.

You shall write three programs on your ISA. You may assume that the first starts at address 0, and the other two are found in memory after the end of the first program (at some nonoverlapping address of your choosing). The specification of your branch instructions may depend on where your programs reside in memory, so you should make sure they still work if the starting address changes a little (e.g., if you have to rewrite one of the programs and it causes the others to also shift). This approach will allow you to put all three programs in the same instruction memory later on in the quarter.

We will impose the following constraints on your design, which will make the design a bit simpler.

- Your ALU instructions will be comparable in complexity to those in ARM.
- You may also have a single ALU condition/flag register (e.g., carry out, or shift out, sign result, zero bit, etc., like ARM's Z, N, C, and V status bits) that can be written at the same time as an 8-bit register, if you want.
- Your data memory (Verilog design will be provided) is *byte-addressable*, and will have only one address pointer input port, for both input and output.
- Your register file (or whatever internal storage you support) will have no more than two output ports and one input port. You may use separate pointers for reads and writes, if you wish. Please restrict register file depth to no more than 16 registers.
- Manual loop unrolling of your code is not allowed.
- You may use lookup tables (LUTs) / decoders, but these are limited to 32 elements each (i.e., address pointer width up to 5 bits). We do not allow something like a 512-element LUT with 32-bit outputs which simply maps your restricted 9-bit machine code field to a 32-bit clone of ARM or MIPS instructions.

2. Report Quality

You shall turn in a lab report, plus program listings. The report will answer the questions posed below. In describing your architecture, keep in mind that the person grading it has much less experience with your ISA than you do. It is your responsibility to make everything clear – one objective of this course is to help you improve your technical writing and reporting skills, which will benefit you richly in your career.

For each lab, there will be a set of requirements and questions that direct the format of the writeup and make it easier to grade, but strive to create a report you can be proud of. Sometimes that may require a little repetition, e.g. describing something where you think it belongs in the report, and then again in the "question" part, so the graders won't miss it.

3. Performance/Simplicity

Generic, general-purpose ISAs (that is, those that will execute other programs just as efficiently as those shown here) will be seriously frowned upon. We really want you to optimize for these programs only, as is common practice in consumer embedded systems with tight cost constraints. Therefore, you are highly encouraged to design an ISA that run the given programs as efficiently as possible, given the constraints. To give you the opportunity to compare your designs against other teams, we will announce the instruction count (one important performance metric) that each team achieves in a scoreboard that we will later post. Also, in addition to the above given constraints, the following suggestions will either improve your performance or greatly simplify your design effort: In optimizing for performance, distinguish between what must be done in series vs. what can be done in parallel. An instruction that does an add and a subtract (but neither depends on the output of the other) takes no longer than a simple add instruction. Similarly, a branch instruction where the branch condition or target depends on a memory operation will make things more difficult later on.

4. Grading/Policy

While this lab *directly* accounts for only 5% of your grade, it is a crucial step towards the goal of the class–designing a complete processor, so you need the results of this step to be successful in the subsequent labs (where the majority of your grade comes from). Thus, we encourage you to use this lab as an opportunity to get feedback from the staff on the feasibility and performance of your designs. You should submit the report for this lab before the due date (Due 2:00 pm PT July 12th 2021). Any late submission will not be graded, due to the tight schedule of the Summer session. But you can always submit a revised version of your ISA description.

5. Components of Your Report

1. **Team**: List the names of all members of your team.
2. **Introduction**: This should include the name of your architecture, overall philosophy, specific goals strived for and achieved.
3. **Instruction Format**: List all formats and an example of each. (ARM has R, I, and B type instructions, for example)

4. **Operations**: List all supported instructions and their opcodes/formats.
5. **Internal Operands**: How many registers your architecture supports? Is there anything special about any of the registers, or all of them are general purpose?
6. **Control Flow (Branches)**: What type of branches are supported? How are the target addresses calculated? What is the maximum branch distance?
7. **Addressing Modes**: What memory addressing modes are supported, e.g. direct, indirect? How are addresses calculated? Give examples.
8. **Class of the Machine**: Can you classify your machine in any of the classical ways (e.g., stack machine, accumulator, register, load-store)? If so, which? If not, devise a name for your class of machine.
9. **An Example Instruction**: Give one example of an "assembly language" instruction in your machine, then translate it into machine code.
10. **Assembly Instructions for Program 1 (Divide)**
11. **Assembly Instructions for Program 2 (Mean)**
12. **Assembly Instructions for Program 3 (Covariance)**

For 10-12, give assembly instructions. Make sure your assembly format is either very obvious or well described, and that the code is well commented. If you also want to include machine code, the effort will not be wasted, since you will need it later. We shall not correct/grade the machine code. State any assumptions you make. If you need initial nonzero values in registers or memory, you need to put them there. (Exception – the test bench will load the inputs for you.)

Program 1: Divide

Test bench will place a 16-bit unsigned integer dividend into data memory addresses 0 (most significant byte) and 1 (least significant byte) and an unsigned 8-bit integer divisor in data memory address 2, all with the start bit high, then lower the start bit to launch your device's execution of the first program. Your program will return (store) a 16-bit unsigned integer quotient in data memory addresses 4 (MSB) and 5 (LSB), with an 8-bit quotient fraction in address 6. At the completion of the division, your device will bring the done flag high for two clock cycles, then wait for the next start command from the test bench, to launch the second program.

-numerical formats:

```
data_memory[0][7:0] = 2^15,2^14,...,2^8   weighted bits of dividend
data_memory[1][7:0] = 2^7,2^6,...,2^0     weighted bits of dividend
data_memory[2][7:0] = 2^7,2^6,...,2^0     weighted bits of divisor
data_memory[4][7:0] = 2^15,2^14,...,2^8   weighted bits of quotient
data_memory[5][7:0] = 2^7,2^6,...,2^0     weighted bits of quotient
data_memory[6][7:0] = 2^-1,2^-2,...,2^-8  weighted bits of quotient
```

-numerical examples:

```
             1/1 = 1:    data_memory [0] = 00000000
                         data_memory [1] = 00000001
                         data_memory [2] = 00000001
                         data_memory [4] = 00000000
                         data_memory [5] = 00000001
                         data_memory [6] = 00000000

             1/2 = 0.5:  data_memory [0] = 00000000
                         data_memory [1] = 00000001
                         data_memory [2] = 00000010
                         data_memory [4] = 00000000
                         data_memory [5] = 00000000
                         data_memory [6] = 10000000
```

Program 2: Mean

This program calculates the arithmetic mean of a series of 16-bit fixed-point numbers. The test bench will place $N \leq 256$, an 8-bit unsigned number, into data memory address 0, which determines the total number of input values. Followed by that, it will place N 16-bit numbers into address 1 to 2N of memory. The numbers exactly follow the numerical format described in Program 1, i.e., the first byte represent the integer part of the number while the second byte represent the fraction part. Then the test bench will lower the start bit to launch your device's execution of the second program. Your program will return (store) a 16-bit unsigned fixed point number in data memory addresses 2N+1 (int part) and 2N+2 (fraction part). At the completion of your code, your processor will bring the done flag high for two clock cycles, then wait for the next start command from the test bench, to launch the third program.

-numerical examples:

```
      Mean(5.5, 0.5) = 3:
      data_memory [0] = 00000010  % N, number of inputs
      data_memory [1] = 00000101  %int part of input 1
      data_memory [2] = 10000000  %fraction part of input 1
      data_memory [3] = 00000000  %int part of input 2
      data_memory [4] = 10000000  %fraction part of input 2
      data_memory [5] = 00000000  %int part of output
      data_memory [6] = 00000011  %fraction part of output

      Mean(0.140625,9) = 4.5703125
      data_memory [0] = 00000010  % N, number of inputs
      data_memory [1] = 00000000  %int part of input 1
      data_memory [2] = 00100100  %fraction part of input 1
      data_memory [3] = 00001001  %int part of input 2
      data_memory [4] = 00000000  %fraction part of input 2
      data_memory [5] = 00000100  %int part of output
      data_memory [6] = 10010010  %fraction part of output
```

Program 3: Covariance

Covariance is a measure of how much two random variables vary together. Covariance can be calculated using the following formula:

$$COV(X, Y) = \frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})(y_i - \bar{y})$$

X and Y are the input random variables, N is the total number of samples, x_i and y_i are the i^{th} sample of X and Y, and $\bar{x}$ and $\bar{y}$ are the arithmetic mean of X and Y.

In this program we want to calculate the covariance of two input random variables. The test bench will place an 8-bit number $N \leq 256$ in the data memory address 0. Followed by that, the data memory address 1:N will contain N 8-bit numbers that correspond to the samples obtain from the random variable X. Then, the test bench will place the N samples from random variable Y, in address N+1:2N. Then, the test bench will lower the start bit to launch your device's execution of the third program. Your program will calculate and store the covariance of X and Y in, a 16-bit unsigned fixed point number, into data memory addresses 2N+1 (int part) and 2N+2 (fraction part). At the completion of your code, your processor will bring the done flag high for two clock cycles, then wait for the next start command from the test bench, to launch the third program.

-numerical examples:

```
X = [1, 2 , 3]
Y = [10, 20, 27]
Cov(X, Y) ≈ 5.67:
data_memory [0] = 00000011  % N, number of inputs
data_memory [1] = 00000001  % x1
data_memory [2] = 00000010  % x2
data_memory [3] = 00000011  % x3
data_memory [4] = 00001010  % y1
data_memory [5] = 00010100  % y2
data_memory [6] = 00011011  % y3
data_memory [7] = 00000101  % output int
data_memory [8] = 10101011  % output farc = 2^-1 + 2^-3 + 2^-5 + 2^-7 + 2^-8 = 0.6679
```