

Computer Organization Project

The Marauder's Map

Professor: Hadi Shahriar Shahhoseini, Ph.D.

TAs: Faraz Ghoreishy and Arvin Delavari

School of Electrical Engineering, Iran University of Science and Technology

June, 2024

1 Introduction

The goal of this project is to study multi-cycle implementation of a RISC-V processor and adding a fixed-point arithmetic unit. This processor is going to execute a simple RISC-V assembly code which is the subject of the first part in the first assignment. This code calculates the distance of lines connecting different points on a map. You are going to use the LUMOS RISC-V processor core which is designed for educational purposes. LUMOS stands for Light Utilization with Multicycle Operational Stages. The multi-cycle implementation allows for more complex instructions and better utilization of processors resources. This processor can execute light applications as it only supports a subset of the 32-bit base integer ISA of RISC-V.

2 LUMOS RISC-V Core

As you know we can break down each instruction into a series of steps corresponding to the various functional units that are needed. We can use these steps to create a multi-cycle implementation. In this architecture, each step takes 1 clock cycle to complete. This allows the components in the design and different functional units to be used more than once per instruction, as long as it is used on different clock cycles. This sharing of resources can help reduce the amount of hardware required. This classic view of CPU design partitions the design of a processor into datapath design and control design. Data path design focuses on the design of ALU and other functional units as well as accessing the registers and memory. Control path design focuses on the design of the state machines to decode instructions and generate the sequence of control signals necessary to appropriately manipulate the data path.

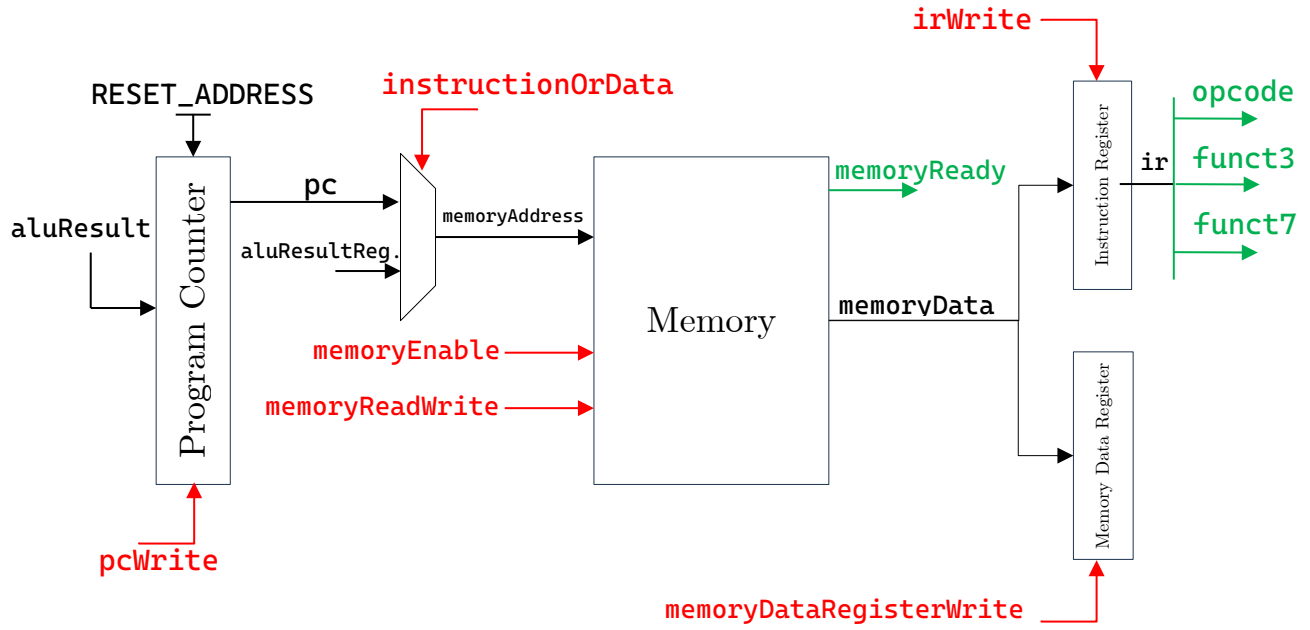


Figure 1 - First part of data path, fetch and decode

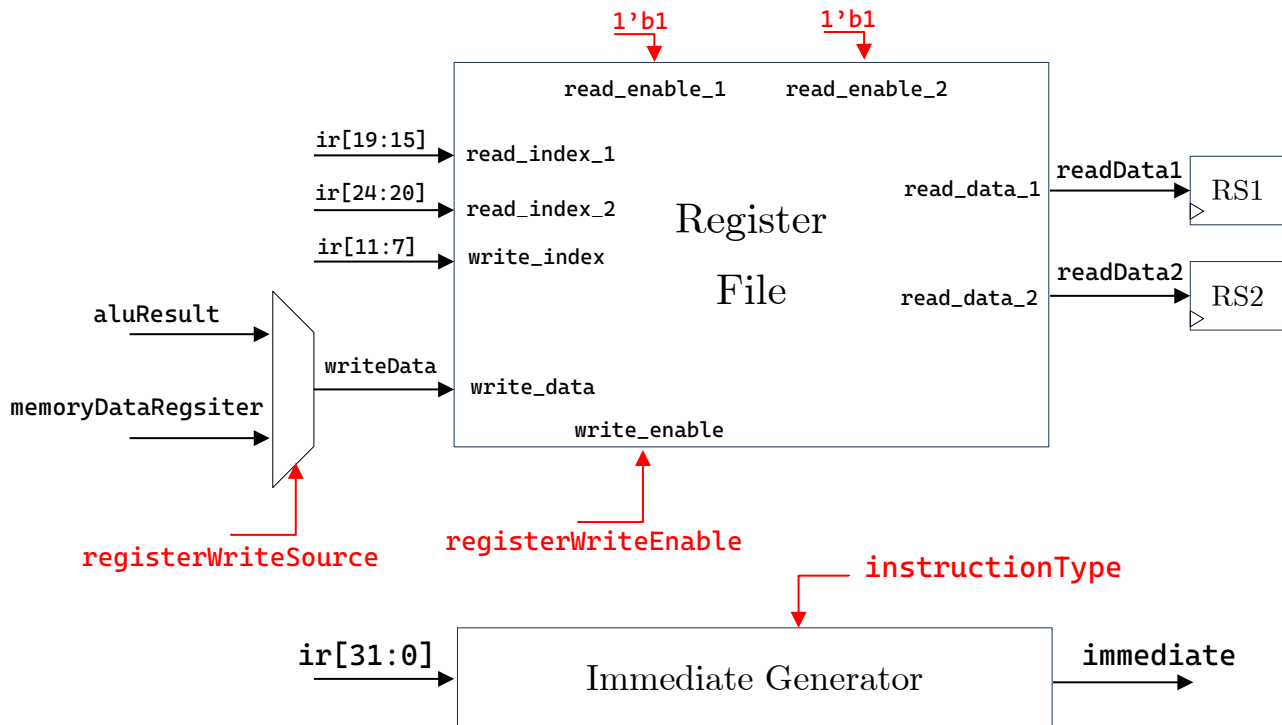


Figure 2 - Second part of data path, operand fetch and immediate generation

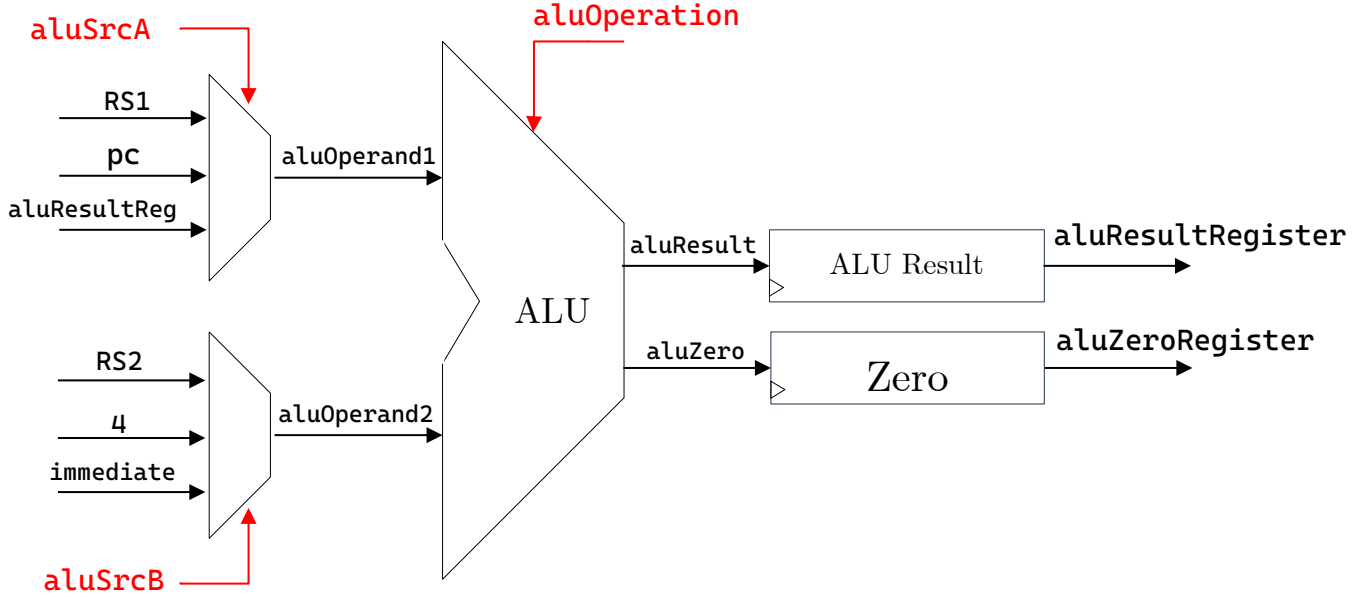


Figure 3 - Final part of data path, execution stage

The data path file written in Verilog is provided in **LUMOS.v**. The first few lines include the components of data path in the main code. The first section of **LUMOS** module is the controllerFSM which we will discuss further. The data path provided in figures 1-3 acts as follows for completely executing an instruction; assuming that program counter register contains a valid value, first by setting a correct signal value to **instructionOrData** we will assure that the data we are reading from the memory is indeed an instruction. After a brief delay contributed to the memory access time, the signal **memoryReady** will rise to indicate that data from memory is ready to be read by the processor. At this moment **irWrite** will assert the value on data bus to be set on the instruction register(**ir**).

As the processor we will be studying in this project implements the RISC-V ISA, the **opcode**, **funct3**, **funct7** and other fields of the instruction will be decoded accordingly. The RISC-V ISA manual can be found at <https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf>. LUMOS processor used in this project implements the 32-bit variant, known as RV32.

In the operand fetch stage, the controller sets the correct instruction type for generating the 32-bit immediate value. The operand fetch stage always reads values from the register file and assigns them to registers RS1 and RS2. In the execution stage the controller selects the first operand of the ALU between three signals: register RS1, program counter register and feedback line from ALU output (**aluResult**). The controller also selects the second operand of the ALU between three signals: register RS2, a constant value of 4, immediate generated in the operand fetch stage. Controllerunit signals the operation to be taken by the ALU.

In the write-back stage, three different set of instructions are capable of writing to the register file. When executing load and store instructions the controller sets the ALU inputs and operations to calculate target address and sets the **memoryAddress** accordingly. In case of load instructions, the result will be written to **memoryDataRegister**. The controller then assigns the right values to input multiplexor at the write port of register file. This multiplexor can select between **memoryDataRegister**, **aluResult** and at the write-back stage. In general design of the processor the multiplexor at the input of write port of register file should be able to select between different sources that provide instruction results.

3 Fixed-Point Unit

Any RISC-V processor implementation must support a base integer ISA (in our case, RV32I). In addition, an implementation may support one or more extensions. The standard instruction-set extension for single-precision floating-point, which is name “F”, adds single-precision floating-point computational instructions compliant with the IEEE 754-2008 arithmetic standard.

However, in this project you are going to create a Fixed-Point Unit and related circuitry to make the LUMOS RISC-V core capable if executing FADD, FSUB, FMUL and FSQRT instructions rather approximately (i.e. it will be less precise than single precision). As previously discussed, a RISC-V core includes a register file with 32 registers each having the length of XLEN (XLEN = 32 for RV32I). RISC-V standard specifies a new register file again with 32 registers each having the length of FLEN (FLEN = 32 for RV32IF). These new registers will be exclusively used for “F” extension instructions and will not be treated as having integer values.

FLEN-1	0
f0	
f1	
f2	
f3	
f4	
f5	
f6	
f7	
f8	
f9	
f10	
f11	
f12	
f13	
f14	
f15	
f16	
f17	
f18	
f19	
f20	
f21	
f22	
f23	
f24	
f25	
f26	
f27	
f28	
f29	
f30	
f31	

FLEN

The following instructions should be implemented in order to run a simple RISC-V code using F-extension for calculating the distance of the line connecting a series of points on map:

31-25	24-20	19-15	14-12	11-7	6-0	
imm[11:0]		rs1	010	rd	0000111	FLW
imm[11:5]	rs2	rs1	010	imm[4:0]	0100111	FSW
0000000	rs2	rs1	rm	rd	1010011	FADD
0000100	rs2	rs1	rm	rd	1010011	FSUB
0001000	rs2	rs1	rm	rd	1010011	FMUL
0101100	00000	rs1	rm	rd	1010011	FSQRT

You can find more information about how each instruction operates, in the RISC-V ISA manual.

4 Fixed-Point Arithmetic

With decimal numbers, we're used to the idea of using a decimal separator such as a point or a comma to separate integer and fractional parts. We can do the same thing in binary and use the bits to represent any powers of two we want.

For example, think of 6.75 as being $4 + 2 + \frac{1}{2} + \frac{1}{4}$.

In binary this can be shown as:

8	4	2	1	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$
0	1	1	0	1	1	0	0

In this style we can separate the fractional part with a point. We are choosing to interpret the values as being fixed point, but from the hardware logic point of view it's just an 8-bit integer. If we comply with a rule that the position of the fixed point is consistent, we'll get the expected result from mathematical operations.

4.1 Q Notation

To express the number of integer and fractional bits we use Q number format: ***Qi.f*** where ***i*** is the number of integer bits and ***f*** is the number of fractional bits. For example, 0110.1100 has four integer and four fractional bits, so is Q4.4.

All the usual binary math yields correct results when used with fixed-point numbers. Addition works in the same way as for integers:

$$\begin{array}{rcl}
 & 0011.1010 & 3.6250 \\
 + & 0100.0001 & + 4.0625 \\
 = & 0111.1011 & = 7.6875
 \end{array}$$

We can also do subtraction using two's complement to express negative numbers:

$$\begin{array}{rcl}
 & 0011.1010 & 3.6250 \\
 + & 1110.1000 & - 1.5000 \\
 = & 0010.0010 & = 2.1250
 \end{array}$$

4.2 Range and Precision

Using two's complement, a 4-bit value ranges from -8 to +7. [1000 → 0111]

A 4-bit fraction can represent numbers as small as $\frac{1}{16}$ and as large as $\frac{15}{16}$. [0001 → 1111]

Therefore, Q4.4 notation ranges from -8 to +7.9375 with a precision of 0.0625.

4.3 Converting to and from Integers

Conversion to and from regular binary integers simply requires using the appropriate left or right shift. For example, if we're using Q16.16 format we need to left-shift an integer 16 positions to create the Q16.16 fixed-point number:

```
1000101          → Decimal 69
<< 16           → Left shift 16 positions
```

```
100 0101 0000 0000 0000 0000
```

Showing all bits in Q16.16 notation:

```
0000 0000 0100 0101.0000 0000 0000 0000
```

Now we can perform all arithmetic operations on this number with other Q16.16 numbers. For example, let's add decimal 14.84375 to this number:

	0000 0000 0100 0101.0000 0000 0000 0000	69.0
+	0000 0000 0000 1110.1101 1000 0000 0000	+ 14.84375
=	0000 0000 0101 0011.1101 1000 0000 0000	= 83.84375

If you need to convert a Q16.16 number back to a regular integer, you have to right-shift 16 positions:

```
0000 0000 0101 0011.1101 1000 0000 0000    → Decimal 83.84375
                >> 16                        → Right shift 16 positions
```

```
0000 0000 0101 0011
```

Showing all bits in Q16.16 notation:

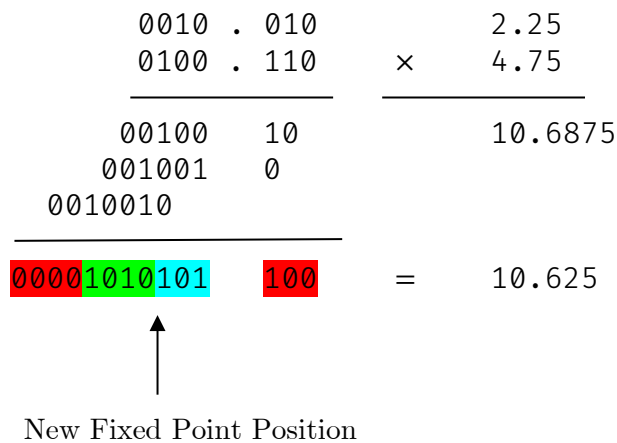
```
0000 0000 0100 0101.0000 0000 0000 0000
```

Using the fixed-point Q22.10 notation you don't need to include additional FPU hardware for your RISC-V core to be capable of executing FADD, FSW, FLW. But the control unit needs to be altered to accommodate these new instructions.

5 Fixed-Point Multiplication

Similar to addition and subtraction, simple binary multiplication can be used for fixed-point numbers. However, there is extra consideration for selecting the number of bits in the product that represent the final result. As the numbers we are multiplying are IBITS + FBITS long, after multiplication the fixed point moves FBITS to the left.

$\begin{array}{r} 0010 \text{ . } 010 \\ 0100 \text{ . } 110 \\ \hline 00100 \quad 10 \\ 001001 \quad 0 \\ 0010010 \\ \hline \end{array}$	\times	$\begin{array}{r} 2.25 \\ 4.75 \\ \hline 10.6875 \end{array}$
$\begin{array}{l} 0000 \text{ } 1010101 \text{ } 100 \end{array}$	$=$	10.625



New Fixed Point Position

As can be seen from the figure above, after multiplication we ignore the lower FBITS of the product the conform to the original Q format. This causes an inherent approximation when multiplying fixed-point numbers.

6 Fixed-Point Square Root Calculation

In this part you are going to design an Algorithmic State Machine (ASM) to calculate the square root of integer and fixed-point numbers. Although lower-latency methods exist, the method explained here includes a straightforward digit-by-digit approach, using only subtraction and bit shifts. Most square roots are irrational, so the algorithm would result only an approximate answer. The number we're finding the root of is known as the *radicand*. To explain this algorithm, consider the following example:

Consider the radicand to be 10101001 (169 in decimal).

This algorithm works with pairs of digits. Before calculations, radicand is split into pairs starting with the least significant. Our radicand here becomes 10_10_10_01.

<pre> 10 10 10 01 ↓ 10 01 01 01 01 10 01 01 00 01 00 01 10 11 01 10 01 01 10 01 01 10 01 00 00 00 </pre>	<p>Our radicand</p> <p>Bring down the most significant pair</p> <p>Subtract 01</p> <p>The answer is NOT negative so the first digit is 1</p> <p>Bring down the next pair of digits and append to previous result</p> <p>Append 01 to the existing answer, 1, and subtract</p> <p>The result is NOT negative so next digit is 1</p> <p>Bring down the next pair of digits and append to previous result</p> <p>Append 01 to the existing answer, 11, and subtract</p> <p>The result is negative so the next digit is 0</p> <p>We discard the result because it's negative.</p> <p>Keep the existing digits, 0110, and append the next pair</p> <p>Append 01 to the existing answer, 110, and subtract</p> <p>The result is NOT negative so next digit is 1</p>
---	---

There are no more pairs of digits, and the result of our last step is 0, so our answer is exact 1101 (13 in decimal):

1 1 0 1 → One digit for each pair of digits in the radicand

Now that you are familiar square root calculation, you are going to add support for fixed-point calculation. Supporting fixed-point square roots is straightforward; We just need to run more iterations to account for the fractional digits. There's no possibility of overflow, as the root of the radicand greater than 1 is always smaller than the radicand itself.

Your module needs a new parameter FBITS for the fractional bits in the radicand. Using the Q22.10 format parameters will be set as follows:

$$\text{WIDTH} = 32$$
$$\text{FBITS} = 10$$

For this new radicand you need to perform $(32+10)/2$ iterations.

7 Assignments

A member of your group needs to fork the LUMOS processor core repository from the IUST Computer Organization GitHub page.

| <https://github.com/IUST-Computer-Organization/LUMOS.git>

You should be able to see the LUMOS repository as one your own in your GitHub page. After that add the other members of your group as Collaborators to this repository. This way, all members of the group have access to the files and commit changes directly.

Among the files of this directory, you'll find the following files:

LUMOS.v which includes the top module where datapath and controller are located.

LUMOS_Testbench.v which is the simulation environment and should be used in order to run the simulation:

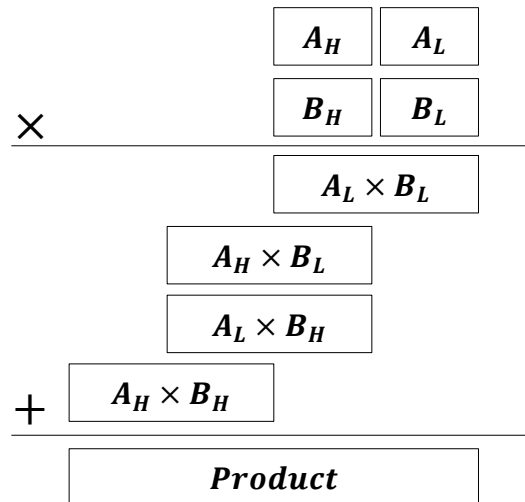
```
iverilog -o LUMOS.vvp LUMOS_Testbench.v
vvp LUMOS.vvp
gtkwave LUMOS.gtkw
```

Study the controller and datapath in **LUMOS.v** in order to better familiarize yourself with the internal operations of the core. You can see that the F Register File is already implemented and the controller supports FLW and FSW instructions. When an OP_FP opcode is decoded the controller sets the signals in a way that the output of the Fixed_Point_Unit module writes to the register file.

Analyze the `Fixed_Point_Unit` module instantiation and its input and outputs (The module is located near the end of `LUMOS.v`). You can see there is “ready” signal which is connected to the FPU. As multiplication and square root calculations require multiple clock cycles, the controller wait for the FPU function to finish and then continue its step. This is asserted using the “ready” signal.

Your job is to only modify the `Fixed_Point_Unit.v` file. In this file you’ll first find an `always` block where the outputs of the unit are assigned. As you can see the addition and subtraction are already taken care of. For multiplication the **result** signal takes its value from selected bits of the **product** signal. Explain this bit selection in your report.

The **ready** signal is also driven by **product_ready**. The circuit you are going to describe should assign proper values to **product** and **product_ready**. From line 50 you can see the reg and wire variables needed for this circuit. You are provided with a simple 16-bit multiplier and you need to use this module to calculate 32-bit multiplication. You need to use this module 4 times in order to completely perform 32-bit multiplication and this operation produces 4 partial products that need to be summed in the final step. Use the following figure as a hint for this operation:



Performing the partial product generation in 4 steps and 1 step for addition, this circuit produces results after 5 clock cycles.

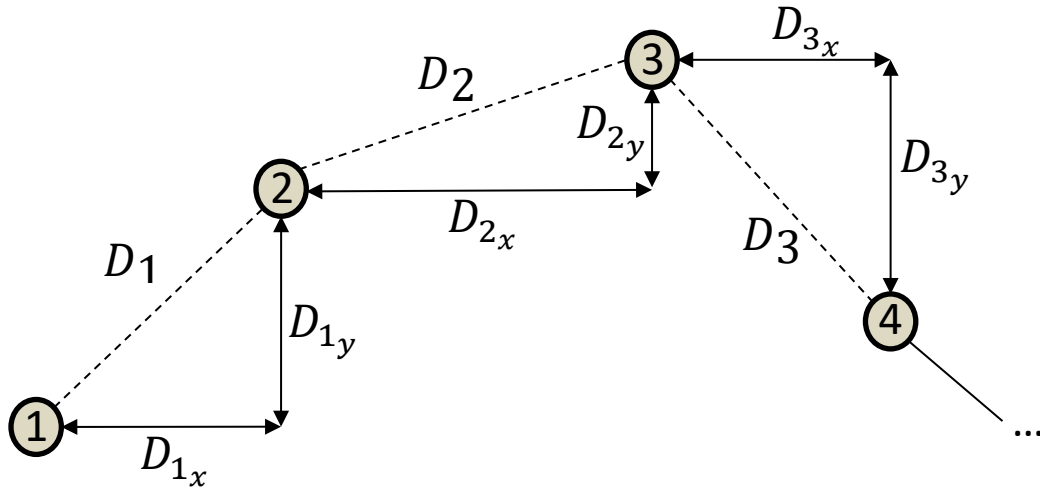
Bonus Points: You are free to modify the architecture by adding only one other instantiation of the 16-bit multiplier. Using the pipeline structure discussed in class increase the performance of your multiplier.

Implementing the square root calculator is similar to the multiplier. From line 40 you can see the **root** and **root_ready** reg variables. After that describe you circuit that assigns correct values to these registers and asserts the ready signal at the right time.

A testing environment for the FPU is provided in the Fixed_Point_Unit_Testbench.v. You are free to modify this file as you see fit or you can use it as provided. This file is used only to simulate the FPU and to make sure of the FPU's correct functionality.

In your repository you'll find a directory called **Firmware**. In this directory the assembly code for calculating the distance is provided in **Assembly.S**. You don't need to modify this file. **Don't change** the contents of the **Firmware.hex** as it loads the memory with the distances used in the calculation. We want to calculate the distance of the path connecting 50 points spread in the Cartesian plane. For the distance between each point the x and y components are given in the Memory unit. Each component occupies 1 Word of the Memory and therefore is represented as a 32-bit unsigned integer:

	Memory	ADDRESS
D_1	D_{1x}	00_H
	D_{1y}	01_H
D_2	D_{2x}	02_H
	D_{2y}	03_H
D_3	D_{3x}	04_H
	D_{3y}	05_H
.	.	.
	.	.
	.	.
D_{49}	D_{49x}	60_H
	D_{49y}	61_H



Study the code written in **Assembly.S**. Explain each section of the code and its purpose in your report. Explain where the values are loaded from, steps of the calculation and what register the final result is in.

Test your project with the LUMOS testing environment. You should be able to see the Fixed-Point Register loading with correct values and other steps of the calculation. If the code is executed correctly, you'll see the waveforms ending automatically at the end and the calculated distance in an F register. Make sure to include pictures of the waveform in your report.

8 Submission

The README.md file in your repository is considered as your final report. Make sure to include the answers to all the questions asked above and a detailed explanation of your codes describing the square root calculator and multiplier. Commit all your changes in an orderly fashion.

Note: Always check the size of your **.vcd** file before committing. As simulations may take a long time to complete or if your circuit isn't performing in a right way the simulation may never reach its end, the **.vcd** file grows in size (sometimes up to multiple GigaByte!). If this happens don't commit your **.vcd** file because it takes a very long time to upload and your GitHub repository might not work for a while.

After the project is finished the collaborator members fork the repository to their own GitHub pages. Each member must submit a text file to LMS containing the link to their repository in the following format:

Name_StudentNumber_Project.txt (BillGates_12345678_Project.txt)