Introduction

In the project, the source code for the skeleton of a single cycle RISC-V processor was provided. This included a RISC-V assembler included a subset of the entire ISA, which was implemented in the file Core.cpp. The basic construction of the software simulation basically followed the flow of Figure 4.21 of Computer Organization and Architecture RISC-V edition by Patterson and Hennessy, shown below.

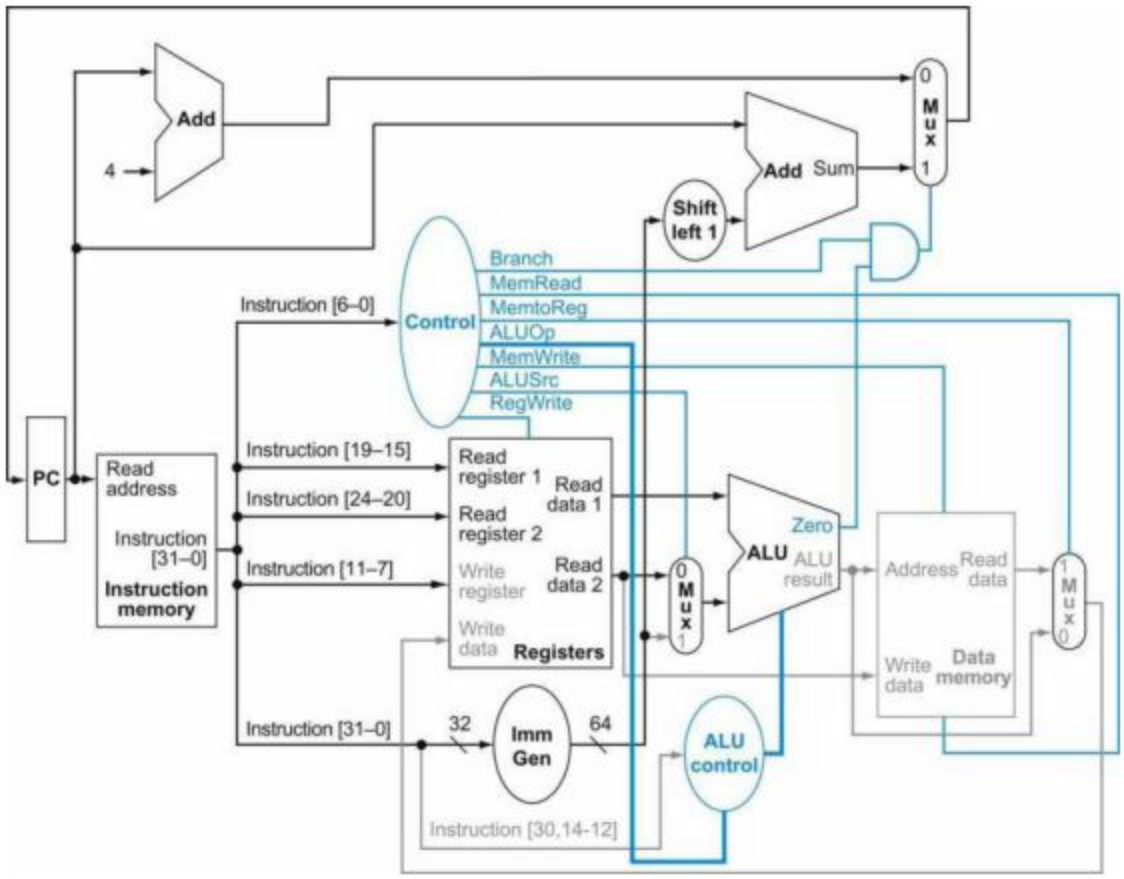

Figure 1: A simple datapath with a control unit for a processor

Implementation

Four classes were added to the project and instantiated as member of the core class. These included Control, Imm_gen, Data_Memory, Registers, and ALU. Muxing, wiring, and control logic between components were implemented in the Core class.

Control

The purpose of the control class was to take the opcode and provide all signals coming out of the control in Figure 4.21 from Computer Organization and Architecture RISC-V. To do this, Figure 4.22 was used to map opcode type to the seven control signals.

Input or output	Signal name	R-format	ld	sd	beq
Inputs	I[6]	0	0	0	1
	I[5]	1	0	1	1
	I[4]	1	0	0	0
	I[3]	0	0	0	0
	I[2]	0	0	0	0
	I[1]	1	1	1	1
	I[0]	1	1	1	1
Outputs	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

Figure 2. The complete control output truth table for R-format, ld, sd, and beq instructions

For the opcodes and instructions not specified in this table, the control signals were determined by referring back to Figure 1 and analytically finding the necessary control signal outputs in order to drive the proper signals to all the logical units.

As an example, the immediate type instructions are not specified by the truth table show in Figure 2, however, by following the datapath laid out in Figure 1, it can be determined that the control signals are the same as R-type instructions with the exception of ALUsrc which would be high (or 1) so that the ALU would be exposed to the Imm Gen output of the immediate values specified in the instruction.

Algorithmic Logic Unit (ALU)

As the name suggests, the ALU class is responsible for all the arithmetic and logical operations that the RISC-V assembly is capable of performing. In order to differentiate between operations, the ALU has to be controlled via the ALU control unit show in Figure 1.

Similar to the control truth table shown in Figure 2, the ALU has a similar truth table that describes all of its necessary operations and states. The table, shown below

in Figure 3, describes what the ALUOp codes are for ld, sd, beq, and R-type instructions in addition to the ALU control inputs which specify some of the operations required.

Instruction opcode	ALUOp	Operation	Funct7 field	Funct3 field	Desired ALU action	ALU control input
ld	00	load doubleword	XXXXXXX	XXX	add	0010
sd	00	store doubleword	XXXXXXX	XXX	add	0010
beq	01	branch if equal	XXXXXXX	XXX	subtract	0110
R-type	10	add	0000000	000	add	0010
R-type	10	sub	0100000	000	subtract	0110
R-type	10	and	0000000	111	AND	0000
R-type	10	or	0000000	110	OR	0001

Figure 3: Complete truth table of ALU control for R-type, ld, sd, and beq instructions

For other functions, like shifts, xors, and equivalent immediate operations, the funct3 and funct7 fields were used to procedurally determine the type of the operation necessary. By looking at the machine code binary values for various operations, all possible permutations were accounted for and coded.

Immediate Generation

The next class created was the Imm Gen logical unit which sign extends the 12-bit immediate field in the instruction bits. By taking in the 32-bit instruction, conditionals were written based on opcode in order to extract the correct immediate values from the proper bitfields. For instance, I-type instructions place their 12-bit immediate in bits 31:20, but SB-type instructions have their 12-bits placed in the funct7 and rd locations with the bits in non-sequential order.

Data Memory and Registers

The Data Memory and Registers class were written pretty similarly in the C++ program. Although in reality, these two logical units are quite different, a software implementation of both can be simplified to simple arrays/maps which can be indexed into for both reading and writing to those registers or memory spaces. Again, all the inputs to the Data Memory and Registers were taken care by the control outputs and wires so these classes were simple implementations.

Simulating a Complex Program

Upon completing the RISC-V simulator, an assembly program was written for an example C code that computed matrix operations. This was tested against the RISC-V

simulator and timed. Then, the program was rewritten for multiple cores at different frequencies to see how multi-core processing decreases execution time. The results are captured Table 1.

Table 1: Execution times of the program with varying number of cores and frequency

	Execution Time (ns)
Single Core, 1.0 GHz	338
Dual Core, 0.5 GHz	336
Quad Core, 0.25 GHz	341

The results demonstrate that the replication of resources can make the program run at a similar time even when run at a slower clock frequency.