Integration

At this stage the integration of all units and pipeline registers was done. Before the integration, it was tested separately on some crucial parts in the data path. The control logic unit was tested with an automated program that would compare the output of the control unit with a pre-coded CSV file. The CSV file with the control unit outputs was generated from the spreadsheet that was created in the planning stage. More testing details can be found in the testing section.

Since the ALU, instruction memory/cache, data memory/cache, register file was imported from the previous CPU project from the CO224, it was not tested separately. Even Though those modules were imported from the previous project, some changes needed to be done. The improvements that were done to the ALU and the register file can be found in the part 2 report. Apart from that following are the improvements that were done to the instruction memory/cache and the data memory/cache.

- **Instruction memory**

The instruction memory consists of **1024** bytes. It was designed to support byte addressing. The instruction memory can store data blocks of **16** bytes (4 words). Since it has byte addressing 28 bits were used to address a block. To load the instructions to the instruction memory a **.bin** file that was generated from the assembler program was used. This **.bin** file has 1024 bytes of instructions data. The instruction block fetching delay was set to 40-time units to simulate the latency of the instruction memory.

- **Instruction data cache**

The instruction memory cache consists of 8 blocks. Each block has 4 words. To address a block 3-bit index was used. A 2-bit offset is used to fetch the required 32-bit instruction. The remaining 28 bits were used as the tag. Apart from the dirty and a valid bit was assigned per block. The structure is as below.

Dirty	valid	Tag [24:0]	index[2:0]	offset[1:0]	data[127:0]

- **Data memory**

The data memory consists of **256** bytes. It was designed to support byte addressing since some instructions require byte addressing rather than 32-bit word addressing. The data memory is designed to serve 16-byte data blocks in total. But with the block size of the data memory cache. Since it serves a 16-byte length of blocks 28 bits were used to address each block. The data memory is capable of resetting all data bytes to zero with a positive edge of the reset.

- **Data memory cache**

The data memory cache consists of 8 blocks with a size of 16 bytes. Each block has four 32-bit words that can be selected by the 2-bit offset. A 3-bit index was used to address the 8 blocks. Also, a byte offset is used to select the specific byte from the 32-bit word. The structure of the block is as below.

Dirty	valid	Tag [24:0]	index[2:0]	offset[1:0]	byte_offset[1:0]	data[127:0]

Since some instructions such as SB, SH, LB, and LH require byte addressing the cache has additional functionality to mask the required bytes from the 32-bit word in fetching and writing.
The additionally added byte_offset is used to select the specific bytes in the 32-bit word.

Pipeline registers

Pipeline registers are the key feature of the pipeline architecture. In this design, there are 5 stages that require 4 sets of pipeline registers. The structure of the pipeline registers throughout the whole data path is as below.

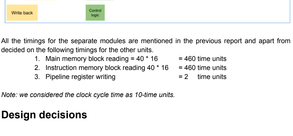
Pipeline registers for data propagation

Stage 1-2	Stage 2-3	Stage 3-4	Stage 4-5
PR_INSTRUCTION PR_PC_S1	PR_REGISTER_WRITE_ADDR_S2 PR_PC_S2 PR_DATA_1_S2 PR_DATA_2_S2 PR_IMMEDIATE_SELECT_OUT	PR_REGISTER_WRITE_ADDR_S3 PR_PC_S3 PR_ALU_OUT_S3 PR_DATA_2_S3	PR_REGISTER_WRITE_ADDR_S4 PR_PC_S4 PR_ALU_OUT_S4 PR_DATA_CACHE_OUT

Pipeline registers for control signal propagation

Stage 1-2	Stage 2-3	Stage 3-4	Stage 4-5
	PR_BRANCH_SELECT_S2 PR_ALU_SELECT PR_OPERAND1_SEL PR_OPERAND2_SEL PR_MEM_READ_S2 PR_MEM_WRITE_S2 PR_REG_WRITE_SELECT_S2 PR_REG_WRITE_EN_S2	PR_MEM_READ_S3 PR_MEM_WRITE_S3 PR_REG_WRITE_SELECT_S3 PR_REG_WRITE_EN_S3	PR_REG_WRITE_SELECT_S4 PR_REG_WRITE_EN_S4

Timings



All the timings for the separate modules are mentioned in the previous report and apart from them, we have decided on the following timings for the other units.

1. Main memory block reading = 40 * 16 = 460 time units
2. Instruction memory block reading 40 * 16 = 460 time units
3. Pipeline register writing = 2 time units

Note: we considered the clock cycle time as 10-time units.

Design decisions

- ***Moving the shifters inside the ALU***

We have moved the shifter inside the ALU rather than having a separate barrel shifting module. We have implemented this way to reduce the complexity of the Design

- ***Design a separate branch selection unit***

We have implemented the branching unit as a separate module without implementing it inside the CPU because more than five instructions are using this unit. Like **JAL, LALR, BEQ, BNE, BLT, BGE, BLTU, BGEU**

- ***Implementing the byte selection inside the data cache***

The SB, SH, LB, LH kinds of instructions are required to store or load a byte or 2 bytes as their requirement. Therefore it is required to implement a unit that has the ability to mask out the required byte or 2 bytes according to the requirement of the instruction.

In our design, we decided to implement that unit inside of the data memory module. Let's see y what it does. Consider the requirement of the instruction is to load a single byte from a 32-bit word. Then the data cache will fetch the corresponding 32-bit word that contains the relevant byte. Then the additional byte filtering mechanism will extract the required byte and extend its sign if needed(sign will not be extended in LBU, LHU instructions) and serve the byte as output. In the case of two byte loads for LH the process will be the same, the only change will be the size of the mask. For the store scenario, the required byte will be replaced with the 32-bit word without changing the other three bytes.

Testing

- ***Automated control unit testing***

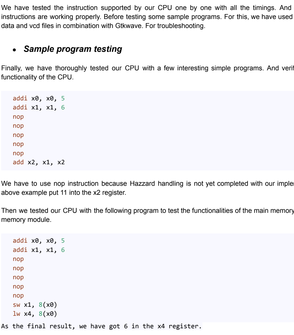
As have implemented the automated testing procedure for the control unit with help of a CSV file. If we want to change or optimize any control unit signals. The testing process is fully automated. So after we have changed the structure or any optimizations in the control unit, the testing code will automatically check for all the supported instructions for the CPU and give us the results and if there are any failures. It also provides which signals are failed. The sample test result is as follows.

```
Checking AND    -> Test Pass
Checking MUL    -> Test Pass
Checking MULH   -> Test Pass
Checking MULHSU -> Test Pass
Checking MULHU  -> Test Pass
Checking DIV    -> Test Pass
Checking REM    -> Test Pass
Checking REMU   -> Test Pass
Checking LUI    -> Test Pass
Checking JAL    -> Test Pass
Checking JALR   -> Test Pass
Checking BEQ    -> Test Pass
Checking BNE    -> Test Pass
Checking BLT    -> Test Pass
Checking BGE    -> Test Pass
Checking BLTU   -> Test Pass
Checking BGEU   -> Test Pass

Checking ECALL  -> x
00000100000000000xxx0001
00000000000000000000000
Test Failed


Checking EBREAK -> x
00000100000000000xxx0001
00000000000000000000000
Test Failed

44/47 Test cases passed
```

- ***Pipeline register propagation testing***

We have tested the pipeline propagations. With the help of Gtkwave timing diagrams. A sample test result is given below.

Data propagation through the pipeline



Control signal propagation through the pipeline

- ***Instruction functionality testing***

We have tested the instruction supported by our CPU one by one with all the timings. And verified all the instructions are working properly. Before testing some sample programs. For this, we have used the register file data and vcd files in combination with Gtkwave. For troubleshooting.

- ***Sample program testing***

Finally, we have thoroughly tested our CPU with a few interesting simple programs. And verified the correct functionality of the CPU.

```
addi x0, x0, 5
addi x1, x1, 6
nop
nop
nop
nop
nop
add x2, x1, x2
```

We have to use nop instruction because Hazzard handling is not yet completed with our implementation. The above example put 11 into the x2 register.

Then we tested our CPU with the following program to test the functionalities of the main memory and the cache memory module.

```
addi x0, x0, 5
addi x1, x1, 6
nop
nop
nop
nop
sw x1, 8(x0)
lw x4, 8(x0)
As the final result, we have got 6 in the x4 register.
```