Personal Project Statement

Toby Browne

Memory, Integration, Debugging and Pipelining

Single Cycle Contributions

Control Unit

Once the original version of the *CU* had been written, I highlighted the need for a restructure, away from using nested multiplexers to a more readable and expandable form.
The initial revision of the *Control Unit* had defined control signals on an instruction-by-instruction basis, not taking advantage of the patterns between instructions of the same group.
This may have been maintainable if we were only implementing the minimum number of instructions required to run the reference program, however I knew it would not work for my aim of implementing all the instructions studied in lectures.

To investigate how the *Control Unit* could be optimized, I created the following spreadsheet to keep track of the instructions we had implemented and their respective control signals; this was shared to all team members and was very helpful in later stages of development and debugging.

Arithmetic Instructions

Mnemonic	Instruction	Type	Branch	ResultSrc	MemWrite	ALUCtrl	ALUSrc	ImmSrc	RegWrite
ADD rd, rs1, rs2	Add	R	0	00	0	0000	0	X	1
SUB rd, rs1, rs2	Subtract	R	0	00	0	1000	0	X	1
ADDI rd, rs1, imm12	Add Immediate	I	0	00	0	0000	1	000	1
SLT rd, rs1, rs2	Set Less Than	R	0	00	0	0010	0	X	1
SLTI rd, rs1, imm12	Set Less Than Immediate	I	0	00	0	0010	1	000	1
SLTU rd, rs1, rs2	Set Less Than Unsigned	R	0	00	0	0011	0	X	1
SLTIU rd, rs1, imm12	Set Less Than Immediate Unsigned	I	0	00	0	0011	1	000	1
LUI rd, upimm	Load Upper Immediate	U	0	00	0	1001	1	100	1
AUIP rd, imm20	Add Upper Immediate to PC	U							

Logic Instructions

Mnemonic	Instruction	Type	Branch	ResultSrc	MemWrite	ALUCtrl	ALUSrc	ImmSrc	RegWrite
AND rd, rs1, rs2	AND	R	0	00	0	0111	0	X	1
OR rd, rs1, rs2	OR	R	0	00	0	0110	0	X	1
XOR rd, rs1, rs2	XOR	R	0	00	0	0100	0	X	1
ANDI rd, rs1, imm12	AND Immediate	I	0	00	0	0111	1	000	1
ORI rd, rs1, imm12	OR Immediate	I	0	00	0	0110	1	000	1
XORI rd, rs1, imm12	XOR Immediate	I	0	00	0	0100	1	000	1
SLL rd, rs1, rs2	Shift Left Logical	R	0	00	0	0001	0	X	1
SRL rd, rs1, rs2	Shift Right Logical	R	0	00	0	0101	0	X	1
SRA rd, rs1, rs2	Shift Right Arithmetic	R	0	00	0	1101	0	X	1
SLLI rd, rs1, shamt	Shift Left Logical Immediate	I	0	00	0	0001	1	000	1
SRLI rd, rs1, shamt	Shift Right Logical Immediate	I	0	00	0	0101	1	000	1
SRAI rd, rs1, shamt	Shift Right Arithmetic Immediate	I	0	00	0	1101	1	000	1

Load/Store Instructions

Mnemonic	Instruction	Type	Branch	ResultSrc	MemWrite	ALUCtrl	ALUSrc	ImmSrc	RegWrite	DataWidth
LW rd, imm12(rs1)	Load Word	I	0	01	0	0000	1	000	1	000
LH rd, imm12(rs1)	Load Halfword	I	0	01	0	0000	1	000	1	001
LB rd, imm12(rs1)	Load Byte	I	0	01	0	0000	1	000	1	010
LHU rd, imm12(rs1)	Load Halfword Unsigned	I	0	01	0	0000	1	000	1	101
LBU rd, imm12(rs1)	Load Byte Unsigned	I	0	01	0	0000	1	000	1	110
SW rs2, imm12(rs1)	Store Word	S	0	X	1	0000	1	001	0	000
SH rs2, imm12(rs1)	Store Halfword	S	0	X	1	0000	1	001	0	001
SB rs2, imm12(rs1)	Store Byte	S	0	X	1	0000	1	001	0	010

Branch/Jump Instructions

Mnemonic	Instruction	Type	Branch	ResultSrc	MemWrite	ALUCtrl	ALUSrc	ImmSrc	RegWrite	PCSrc		
BEQ rs1, rs2, imm12	Branch Equal	B	1	X	0	1000	0	010	0	Zero		
BNE rs1, rs2, imm12	Branch Not Equal	B	1	X	0	1000	0	010	0	~Zero		
BGE rs1, rs2, imm12	Branch Greater Than Or Equal	B	1	X	0	0010	0	010	0	Zero		
BGEU rs1, rs2, imm12	Branch Greater Than Or Equal Unsigned	B	1	X	0	0011	0	010	0	Zero		
BLT rs1, rs2, imm12	Branch Less Than	B	1	X	0	0010	0	010	0	~Zero		
BLTU rs1, rs2, imm12	Branch Less Than Unsigned	B	1	X	0	0011	0	010	0	~Zero		JAL
JAL rd, imm20	Jump And Link	UJ	1	10	0	1000	1	011	1	1	JALR	1
JALR rd, imm12(rs1)	Jump And Link Register	I	1	10	0	1000	1	000	1	X	1	

This spreadsheet made any patterns between instructions very clear. Using it, I began the process of re-writing the *control unit* with Dimitris, a process he finished, removing the *maindecode* and *aludecode* modules in the process.

Relevant Commits:

Upload RISC-V Instruction Set Spreadsheet	Change to IType CU decode to allow addi.	Add I/R/S/B type flags.

Signed Less Than Instructions

Signed less than instructions require some more complex circuitry, due to how negative numbers are represented in binary. When represented in two's complement binary, negative numbers have a higher magnitude than positive numbers, therefore the MSB of the operation's operand must be taken into account.
To achieve the correct functionality, I added the following instruction into the *ALU*:

```
4'b0010: // less than
begin
    if(ALUop1[31] == ALUop2[31]) // if both negative or both negative
        ALUout = {31'b0, ALUop1 < ALUop2};
    else
        ALUout = {31'b0, ALUop1[31]};
end
```

If the two operands have the same sign (same MSB) then they can be compared as usual, otherwise the negative number will obviously be the lesser.
This instruction can then be called with the following control signals:

Mnemonic	Instruction	Type	Description	Branch	ResultSrc	MemWrite	ALUCtrl	ALUSrc	ImmSrc	RegWrite
SLT rd, rs1, rs2	Set Less Than	R		0	00	0	0010	0	X	1

Relevant Commits:

Add support for signed less than instructions.	Set less than (signed) correction.

LUI Instruction

The *LUI* instruction allows a 20 bit immediate operand to be stored in the most significant 20 bits of a register, as opposed to an *LI* instruction which only provides access to the least significant bits of a register.
To implement this instruction, I added a mode to the *resultSrc* multiplexer to allow the *result* signal to be set to the immediate value. Before this change, the *result* signal (carrying write-data to the register file) had to be an output from memory or the result of an ALU operation.
It's worth noting that I later changed this method of implementation when developing the pipelined processor.

I then modified the CU to output the correct control signals for this instruction. These signals can be seen below:

Mnemonic	Instruction	Type	Description	Branch	ResultSrc	MemWrite	ALUCtrl	ALUSrc	ImmSrc	RegWrite
LUI rd, upimm	Load Upper Immediate	U		0	00	0	1001	1	100	1

The immediate that gets stored in the register also has to be left-shifted by 12 bits so that the 20 bit immediate occupies the most significant bits of the register. I chose to implement this bit shift in the sign extend module, by adding a new mode of operation (*ImmSrc = 100*). The code I added to the *sign extend* module can be seen below:

```
3'b100:
ImmExt = {Instr[31:12], 12'b0};
```

Relevant Commits:

Add LUI support.

JALR Instruction

The JALR (Jump And Link Register) instruction jumps to an address calculated as an offset from a value stored in a register, and stores the *return address* in a register.

The first part of this operation was managed by other team members, however I was required to modify the hardware so that *PC+4* (the return address) can be stored in a register.
I did this by expanding the *ResultSrc* signal to a width of 2 bits, allowing me to add *PC+4* as a possible output to the *result* multiplexer. This would allow the *PC+4* signal to be carried to the register file.

The *JALR* instruction has the following control signals:

Mnemonic	Instruction	Type	Description	Branch	ResultSrc	MemWrite	ALUCtrl	ALUSrc	ImmSrc	RegWrite	PCSrc	JALR
JALR rd, imm12(rs1)	Jump And Link Register	I		1	10	0	1000	1	000	1	X	1

Debugging the JALR instruction showed that the *ImmSrc* control signal was being set incorrectly and an unnecessary offset was being added to the *ALUout* value inside the *PC* module, causing the *JALR* instruction to jump correctly. I resolved both of these issues.
Additionally, I decided to restructure the *PC* module so that it used *if* statements instead of *ternary operators*, this would provide greater readability since the addition of the JALR flag logic.

Relevant Commits:

JALR bugs fixed.

Add 3-MUX to switch between MemOut/ALUout/PC+4

Remove redundant PC offset

Integration CPU and Debugging

My most valuable contribution to the construction of the single-cycle CPU was helping integrate the modules made by each of the team members and debugging the CPU's execution of the provided reference program.

The integration process was fairly seamless, aided greatly by our correct use of branching in the project's *git repository*, it mainly consisted of modifications to signal widths and resolving conflicts between each branch's top-level *CPU* file.

The process of debugging the reference program was much more time-consuming. I mainly worked with Adam to get this working, and we were able to expose many small issues in the CPU's design.

The final (and most time-consuming) issue to solve was when we found that data stored in memory wasn't being retrieved correctly. I eventually found that this was because the *SB* instruction was padding the 24 unused bits with 0s, therefore affecting 4 bytes of memory, instead of directly modifying a single byte.
This caused the *store byte* instructions to corrupt the data stored in the neighboring memory addresses. The corrected write logic (affecting the minimum number of bytes) can be seen below:

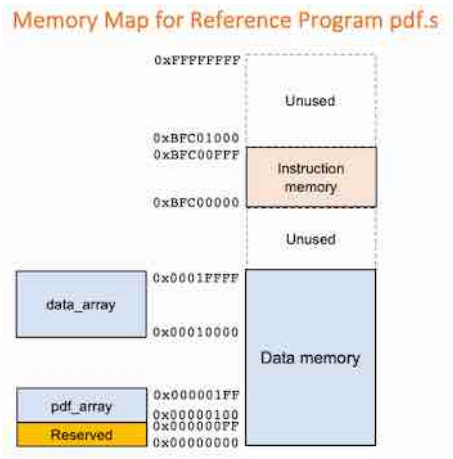
```
case(DataWidth)
    3'b010: // SB
                mem_array[addr] <= wdata[7:0];
    3'b001:
    begin
        mem_array[addr] <= wdata[7:0];
        mem_array[addr+1] <= wdata[15:8];
    end
    3'b000:
    begin
        mem_array[addr] <= wdata[7:0];
        mem_array[addr+1] <= wdata[15:8];
        mem_array[addr+2] <= wdata[23:16];
        mem_array[addr+3] <= wdata[31:24];
    end
```

Relevant Commits:

Merge branch 'cu'	Add RegWrite, MemWrite, make immSrc 2 bits in CPU.sv	Add Datamem and instrmem to top-level CPU file.
Made datamem only write to individual bytes (stop corruption)	Small changes.	Changed datamem mapping and mem files to run REF program

Memory

Lab 4 required a basic ROM module, however this CPU needed two memory modules, one for data and another for instructions. The former requires the ability to write as well as read.
Although these two memory modules are physically separate, they occupy the same *memory space* defined in the project brief and repeated below:



To implement this *memory map* a data and instruction memory module with 13,1072 bytes and 4,096 bytes of storage respectively are required. Since instruction memory starts at address *0xBFC00000*, this constant must be subtracted from any memory address provided to *instruction memory*.
Additionally, since *instruction memory* should only read aligned blocks of 4 bytes at a time (each instruction is a word), the last two bits of the address provided to the instruction memory are always set to 0 (making the address a multiple of 4) as a fail-safe.
The code for reading from *instruction memory* can be seen below:

```
// a fail-safe, ensures you only read from the beginning of a word.
logic[31:0] addr;
assign addr = {pc[31:2], 2'b0} - START_POS;

// reads 4 bytes at a time, starting from the input address
assign dout = {mem_array[addr + 3], mem_array[addr + 2], mem_array[addr + 1], mem_array[addr]};
```

Additionally, I was required to modify the *PC* module so that it would start counting from *0xBFC00000* and also be reset to this number should the reset signal be raised.

One of the main challenges I encountered when making this CPU's memory was RISC-V's inclusion of half-word and byte access instructions. This required the addition of a 3 bit control signal called *DataWidth* which specifies which of the three types of stores (SW, SH, SB) or the 5 types of loads (LW, LH, LB, LHU, LBU) is accessing memory.

The hardware for the store instructions can be seen below:

```
case(DataWidth)
    3'b010: // SB
                mem_array[addr] <= wdata[7:0];
    3'b001:
    begin
        mem_array[addr] <= wdata[7:0];
        mem_array[addr+1] <= wdata[15:8];
    end
    3'b000:
    begin
        mem_array[addr] <= wdata[7:0];
        mem_array[addr+1] <= wdata[15:8];
        mem_array[addr+2] <= wdata[23:16];
        mem_array[addr+3] <= wdata[31:24];
    end
```

The hardware for loading data less than a word is slightly more complicated, as the unused bits being returned must be padded, either with 0s (for unsigned loads) or the MSB (signed loads). The third bit of the *DataWidth* signal is used to specify this padding type. The hardware can be seen below:

```
case(DataWidth)
    3'b000:  // LW
        begin
            dout = {mem_array[addr+3], mem_array[addr+2], mem_array[addr+1], mem_array[addr]};
        end
    3'b001: // LH
        begin
            dout = {{16{mem_array[addr+1][7]}}, mem_array[addr+1], mem_array[addr]};
        end
    3'b010: // LB
        begin
            dout = {{24{mem_array[addr][7]}}, mem_array[addr]};
        end

    3'b101: // LHU
        begin
            dout = {16'b0, mem_array[addr+1], mem_array[addr]};
        end

    3'b110: // LBU
        begin
            dout = {24'b0, mem_array[addr]};
        end

    default: // default just load word
        begin
            dout = {mem_array[addr+3], mem_array[addr+2], mem_array[addr+1], mem_array[addr]};
        end
endcase
```

The control unit also needed reprogramming to utilize this new *DataWidth* signal and include all the new memory operations, the control signals I added to the *CU* can be seen below:

Mnemonic	Instruction	Type	Branch	ResultSrc	MemWrite	ALUCtrl	ALUSrc	ImmSrc	RegWrite	DataWidth
LW rd, imm12(rs1)	Load Word	I	0	01	0	0000	1	000	1	000
LH rd, imm12(rs1)	Load Halfword	I	0	01	0	0000	1	000	1	001
LB rd, imm12(rs1)	Load Byte	I	0	01	0	0000	1	000	1	010
LHU rd, imm12(rs1)	Load Halfword Unsigned	I	0	01	0	0000	1	000	1	101
LBU rd, imm12(rs1)	Load Byte Unsigned	I	0	01	0	0000	1	000	1	110
SW rs2, imm12(rs1)	Store Word	S	0	X	1	0000	1	001	0	000
SH rs2, imm12(rs1)	Store Halfword	S	0	X	1	0000	1	001	0	001
SB rs2, imm12(rs1)	Store Byte	S	0	X	1	0000	1	001	0	010

Relevant Commits:

Define size of instruction memory and add memory offset arithmetic.	Define data memory size	Made instruction memory
Add instruction memory and data memory modules	Add PC offset to align with the instruction memory	Divide 32 bit write data across 4 byte addresses
Make datamem read asynchronous	Parameterize datamem and instrmem.	Reversed read direction of datamem.
Fix issue with datamem	Added signed load instructions (LB/LH)	Add Store/Load Word/Halfword/Byte Instructions
Made datamem only write to individual bytes (stop corruption)	Changed datamem mapping and mem files to run REF program	

Testing

The instruction set spreadsheet mentioned earlier gave us an easy way to keep track of which instructions were functional, however often testing these instructions required the creation of a small assembly program from scratch, making the process quite time consuming.
To mitigate this issue, I wrote an assembly code program called *instrtest.s* which executes many test cases one after another, storing the result of each in the *a0 register* one after another.
I could then create a test bench program *instrtest_tb.cpp* which holds the expected results of each test, compares them to the program's output and returns to the tester how successful their iteration of the *CPU* is.

An example of two test cases for the *SLTIU* and *AND* instructions in *instrtest.s* can be seen below:

```
LI a0, 0
LI a3, 9
SLTIU a0, a3, -1


LI a0, 0
LI a2, 5
LI a3, 3
AND a0, a2, a3
```

As you can see, *a0* is reset to 0 between each test case. This allows the testbench program to differentiate between each test case - but does create the restriction that none of the test programs can have the result 0. The advantage of this approach is that it reduces the number of instructions required for each test case, which should make it easier to find the actual cause of any faults that occur, especially since the only other instruction in each test case is the very simple *LI* instruction.

The bulk of the *instrtest_tb.cpp* testbench file can be seen below:

```
int separateFlag = 0;

long correctOutputs[23] = {7, 2, 1, 4294967289, 1, 1, 1, 1, 1, 1, 1, 7, 6, 1, 7, 6, 4, 268435455, 4294967295, 4, 268435455, 4294967295, 4096};
int testCount = 0;
int correctTests = 0;
// run simulation for MAX_SIM_CYC cycles
for(simcyc = 0; simcyc < MAX_SIM_CYC; simcyc++){
    for (tick = 0; tick < 2; tick++){
        tfp->dump(2 * simcyc + tick);
        top->clk = !top->clk;
        top->eval();
    }

    if(top->a0 == 0){
        separateFlag = 1;
    }
    else{
        if(separateFlag == 1){
            separateFlag = 0;

            if(correctOutputs[testCount] == top->a0){
                correctTests+=1;
            }
            testCount+=1;
        }
    }
```

I deliberately designed this testing strategy for flexibility; it would be very easy to add test cases to the assembly file along with its expected output in the testbench file.

Relevant Commits:

Populate instrtest.s file	Add unit test program and testbench.

Pipelining Contributions

Branching Logic

In the single cycle CPU, we calculate the value of *PCsrc* inside the *Control Unit*, pipelining makes this impossible, as we don't know the value of the *Zero* flag until the *execute stage*.

To solve this problem, I added two extra outputs to the *Control Unit*, the first is a *Branch* flag, which activates during a *Branch* instruction the second is the *Funct3* component of the instruction,
These are both passed through the *decode pipeline register* to forward them to the *execute stage*, here they are used along with the *Zero* flag to find the value of *PCsrc*.
The code to calculate the value of *PCsrc* in the *execute stage* can be seen below:

```
begin
    if(BranchE == 0)
        PCSrc = 0;
    else
        if(JALE)
            PCSrc = 1;
        else
            case(Funct3E)
                3'b000: // beq
                begin
                    PCSrc = ZeroE;
                end
                3'b001: // bne
                begin
                    PCSrc = !ZeroE;
                end
                3'b100: // blt
                begin
                    PCSrc = !ZeroE;
                end
                3'b101: // bge
                begin
                    PCSrc = ZeroE;
                end
                3'b110: // bltu
                begin
                    PCSrc = !ZeroE;
                end
                3'b111: // bgeu
                begin
                    PCSrc = ZeroE;
                end

            endcase
end
```

Relevant Commits

Add new branching logic.

Change branch logic to use Funct3 instead of ALUCtrl

LUI and Jump Instructions

In the single-cycle CPU, *LUI* works by carrying the value of the immediate operand to the *result multiplexer* so it can be sent directly to the *register file*.
Since pipelining greatly increased the complexity of the top-level CPU file, I decided to reimplement LUI so that the immediate operand passes through the *ALU*.
This required me to add a new ALU mode which simply returns the immediate operand unchanged, the code for this new mode can be seen below:

```
4'b1001: // LUI instruction
    ALUout = ALUop2;
```

I also moved the location of the adder which calculates *PC + ImmExt* from the *fetch stage* to the *execute stage*, this way the value of the address to which the *PC* must jump is in the same stage as the branching logic, this also required some modification of the pipeline registers.

The *JALR* instruction works in a similar way to branch instructions. The *JALR* flag is passed into the *decode register* and delayed to the *execute stage*, before being passed to the *PC* module as normal.

The new layout of the branching circuitry meant that the *JAL* implementation must be modified, as it relied on having direct access to *PCsrc* from inside the *CU*.
I resolved this issue by adding a *JAL* flag which is output from the *CU* and delayed through the *decode register* to the *execute stage*. Here the value of *PCsrc* is simply set to that of *JAL*.

Relevant Commits:

Got ref program working on pipelining.

Edit Top-Level Signals for JALR.

Edit ALU for new LUI instruction.

Change PC+4 location and LUI instruction method.

Hazard Module

The hazard module's responsibility is to output control signals related to pipelining to manage instructions which cannot be pipelined normally and thus require some form of stalling or forwarding.
This module was written by Archisha, but the process of integrating pipelining meant that a few modifications were required.

For the ease of reading and modification, I edited the module to replace *ternary operators* with *if statements*, this exposed that there were some faults in the *forwardAE* and *forwardBE* signal logic which I also corrected.

The logic for these signals can be seen below:

```
// assign ForwardAE
if(((Rs1E==RdM) & RegWriteM) & (Rs1E!=0))
    ForwardAE = 2'b10;
else if(((Rs1E==RdW) & RegWriteW) & (Rs1E!=0))
    ForwardAE = 2'b01;
else
    ForwardAE = 2'b00;

// assign ForwardBE
if(((Rs2E==RdM) & RegWriteM) & (Rs2E!=0))
    ForwardBE = 2'b10;
else if(((Rs2E==RdW) & RegWriteW) & (Rs2E!=0))
    ForwardBE = 2'b01;
else
    ForwardBE = 2'b00;
```

Then I could add the *hazard module* to the top-level file and connect the associated signals.

Relevant Commits:

Hazard.sv fix.

More pipeline fixes.

Add hazard module to top-level design.

Reformatted hazard.sv

Pipeline Registers

The pipeline registers were also written by Archisha, but during integration some additions were required.

Firstly, I added stall and flush signals to the fetch, decode and program counter registers, as they had been omitted, I also had to change the existing enable signals of the pipeline registers to be active-low. This way a stall signal would dis-enable the register.

Additionally, there were many new control signals like *JAL*, JALR and *DataWidth* that I was required to integrate with the existing pipeline registers, ensuring they were only used "in-sync" with the instruction to which they belonged.

Finally, I was required to change the precedence of the enable and clear signals inside each pipeline register so that a clear signal would override the existence of an enable signal.

Relevant Commits:

Change pipeline reg en/clear precedence (and other changes).	Made pipeline register enables active-low.	Added stalls and flushes to pipeline registers.

Integration

The most challenging part of implementing pipelining was editing the top-level *CPU* and integrating the existing modules with the pipeline registers; the number of top-level signals was far greater for the pipelined *CPU* than the single cycle.

Besides general signal creation and modification (of which there was a lot), I was required to add two multiplexers to control the two possible register values passed to the *ALU*.
Each of these multiplexers will either return the regular register value being outputted to the register or the last *ALU* output or the result of the last instruction. These two multiplexers are controlled by the hazard signals *ForwardAE* and *ForwardBE*, and allow for forwarding of data between stages.
This is required when an instruction accesses data modified by the previous instruction.
The hardware for these two multiplexers can be seen below:

```
case(ForwardAE)
    2'b00:
        SrcAE = RD1E;
    2'b01:
        SrcAE = resultW;
    2'b10:
        SrcAE = ALUResultM;
    default:
        SrcAE = RD1E;
endcase

case(ForwardBE)
    2'b00:
        regOp2 = RD2E;
    2'b01:
        regOp2 = resultW;
    2'b10:
        regOp2 = ALUResultM;
    default:
        regOp2 = RD2E;
endcase
```

Relevant Commits:

Fixed MORE signal widths.	More signal fixes.	Signal name correction
Organised more signals.	More pipelining top-level setup.	Started integrating pipelining modules

Reflections

I found my time working on this project to be very rewarding and satisfying. I think this is mainly due to the types of responsibilities I adopted. Working on debugging and integration were some of the most time consuming and frustrating tasks however they were able to give me a deep and broad understanding of every single component of the CPU. Although these contributions were less documentable than my contributions to the CPU's memory I value them far more.

Debugging using tools like GTKwave can provide an understanding of a CPU's working impossible to gain from a textbook, as well as allowing me to use debugging skills and strategies I have acquired when working in software in a completely different paradigm.
This was most clear when debugging pipelining, which required a great deal of technical understanding, mental clarity and procedural thinking due to the number of processes taking place.

I believe the subject I learnt most about during this project was pipelining. Although my first-year modules gave me an idea of how this worked, the process of independently converting our single-cycle CPU to a pipelined processor awarded me a depth of comprehension I am very glad of.
Our team found ourselves quite pressed for time in the final two weeks, an experience which strengthened a great deal of the team's soft skills as well. I was required to manage risk, evaluate my strengths and where they would be most effectively applied as well as dedicate huge amounts of time to bring the project to completion.

Had we not been more pressed for time, I would have enjoyed making a more robust and complex testing system. I would likely have added more test cases to the test program I wrote as well as created a testbench that provided more helpful feedback when tests failed, perhaps reducing the use of external tools like GTKwave.

Finally, I am glad that I have been able to gain competence using SystemVerilog and Verilator, I am sure these are skills that will serve me well in my future career.