Personal Statement – Noor Elsheikh

Work overview from 20/11/22 to present:

- Designed top-level verilog schematic for the RISCV CPU `riscv.sv` under the lab 4 specification.

- Designed `mainDecoder.sv`, `aluDecoder.sv` and `controlUnit.sv` and respective testbenches, with control signals for fundamental I-type, R-type, B-type, S-type instructions.

- Designed and implemented JAL and JALR (J-type instructions) into the CPU with test programs.

- Implemented 20-bit immediate operand form into `signextend.sv` module.

- Top-level architectural changes e.g MUXes, extra internal components, supplementary in/outputs.

- Added remaining I-type and B-type instructions to control and coordinated conditional checks for additional branch instruction via ALU.

- Verified and documented valid operation of reference program with provided data arrays

- Additional test programs for debugging B-type and J-type operations

Design decisions and reflections

`mainDecoder.sv` and `aluDecoder.sv`

Designing the decoder was invaluable for developing an understanding in which the chronological path taken for each instruction is fully mapped out in my brain. I needed to deeply understand the exact path taken by each instruction so I could assign the correct signals to each component. Taking BEQ as a case study:

- ALUSrc: Program Counter calculates the branching target address (PC+ImmOp) not ALU, but we still need ALU.B = RD2 so the ALU can check the branch conditions e.g. ([rs1 == rs2]?) and fed back to the control unit with the “zero” flag.

- ALUOp: sets ALUctrl to 0b0000, which overlaps with the ALUctrl for add/addi instructions. I chose to permit overlap as the ALUResult doesn’t affect branch operation, only ALU's "zero" flag is needed. The control unit handles the consequent setting of the "zero" flag for add/addi instructions, ensuring that a branch will only be executed if both the "zero" flag is set AND the instruction is evaluated as B-type. Allowing this overlap simplifies the design of the ALU and keeps the architecture dense, reducing the number of control signals required.

- ResultSrc: We don’t write to the register for branch instructions, so these are “don’t care” bits. Extended from 1 to 2 bits which I detail later.

- Branch: Distinguishes J-type, B-type or neither. I used logic between the 2-bit branch signal and the zero flag to determine the PCSrc. This logic allowed me to also utilise "Branch" for J-type instructions, keeping the control signals dense.

- PCSrc: Logic between zero/Branch provides PCSrc = 0b01 leading to PC := BTA (if rs1==rs2)

- ImmSrc: Selects 12-bit immediate format for PC := PC+Immop (BTA)

- **RegWrite:** Disabled for B-type instructions. Enabled for J-type instructions to store the return address upon branching

- **MemWrite:** Disabled for B-type

I concatenated ALUOp with funct3, funct7[5] and opcode[5], to form a dense select signal in a single “casez” statement in the ALU Decoder which allows for “don’t care” bits to simplify the case matching.

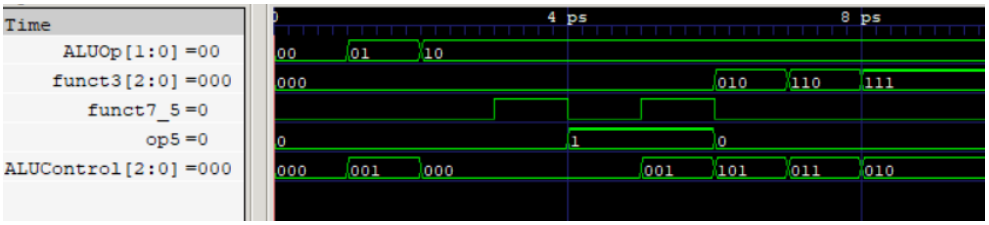
```
always_comb begin
    casez({ALUOp, funct3, {op5, funct7_5}})
        {2'b01, 3'b000, 2'b?}: ALUControl = 3'b000; //add, addi, beq,
        {2'b01, 3'b001, 2'b?}, {2'b10, 3'b000, 2'b11}:                          ALUControl = 3'b001; //bne, subtract
        {2'b10, 3'b001, 2'b?},  {2'b00, 3'b001, 2'b00}, {2'b01, 3'b101, 2'b?} : ALUControl = 3'b110; //bge, SLL
        {2'b10, 3'b010, 2'b?},  {2'b00, 3'b010, 2'b?}, {2'b01, 3'b110, 2'b?} :  ALUControl = 3'b101; //bltu, slt, slti, need to add sltiu
        {2'b10, 3'b111, 2'b?}, {2'b00, 3'b111, 2'b?}, {2'b01, 3'b111, 2'b?} :   ALUControl = 3'b010; //and, bgeu
        {2'b10, 3'b110, 2'b?}, {2'b00, 3'b110, 2'b?}, {2'b01, 3'b100, 2'b?}  :  ALUControl = 3'b011; //or, ori, blt
        {2'b10, 3'b100, 2'b?}, {2'b00, 3'b100, 2'b?} :                          ALUControl = 3'b100; //xor, xori
        {2'b10, 3'b101, 2'b?}, {2'b00, 3'b101, 2'b00} :                         ALUControl = 3'b111; //shift right logical
        {2'b11, 3'b?, 2'b?} :                                                   ALUControl = 3'b000; //JALR!
        default : ALUControl = 3'b000;
    endcase
end
```

This allows for more flexible matching and the execution of corresponding code. Overlap between ALUControl signals keeps it dense and scalable.

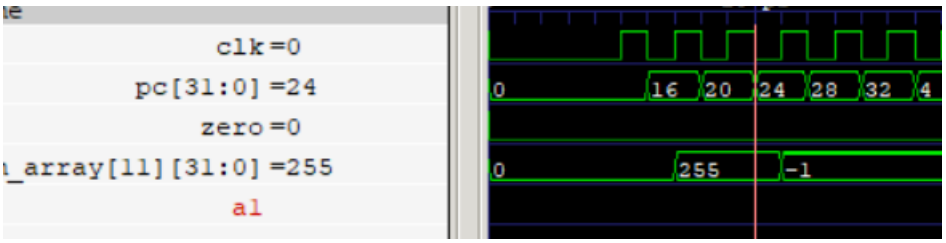
Reference program and Debugging

Debugging the reference program revealed issues seen as false positives in previous tests. I documented it’s verified final operation in the Reference Tests document. Zero was not being

ALUOp	funct3	{op_5, funct7_5}	ALUControl
00	x	x	000 (add)
01	x	x	001 (subtract)
10	000	00, 01, 10	000 (add)
	000	11	001 (subtract)
	010	x	101 (set less than)
	110	x	011 (or)
	111	x	010 (and)



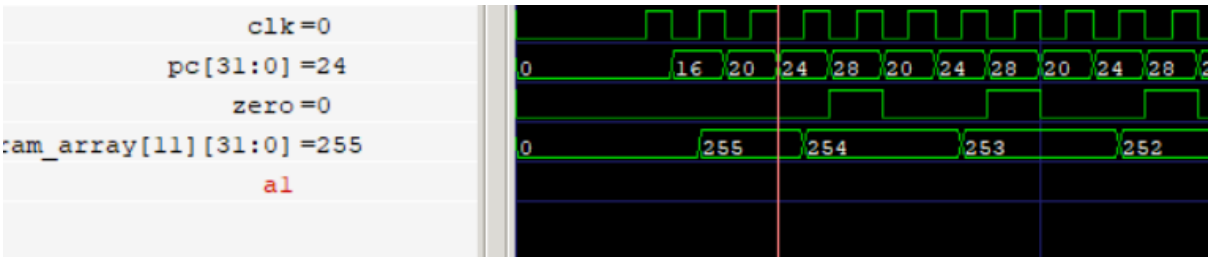
set for certain unsigned conditional checks due to overlap between control signals and lack of explicit distinction between signed and unsigned ALU operation. From this I learnt not being extensive enough in case handling for the sake of simplicity can be detrimental when it comes to corner cases in operation, so it’s best to be as thorough as possible from the start in considering all possible cases. I resolved this by creating additional conditional checks between operands explicitly defined in Verilog as either signed or unsigned. I further added BLTU, BLT and BGEU branch conditions and added control signals for immediate instructions; XORI, ORI, SLTI, ANDI, adding them to the cases for their equivalent S-type instruction ALUCtrl outputs I had previously added. The changes can be found in this commit.




```
init:
    LI      a1, 0xff
_loop1:
    SB      zero, base_pdf(a1)
    ADDI    a1, zero, -1
    BNE     a1, zero, _loop1
    RET
```

Reference program waveform before fix:

Zero flag not being set and “ADDI a1, zero, -1” performs a1 := (0) + (-1) = -1, which just sets it to -1 rather than decrementing it by 1, so this is an issue with the reference program itself. I changed this to ADDI a1, a1, -1.



The zero flag is now being correctly set so the branch is allowing the program counter to loop. a1 is now also being decremented by 1 on each iteration. Commit.

`Implementing JAL and JALR`

In future, when tasked with designing multiple features, I will plan ahead to ensure that they can all be integrated compatibly. This will prevent the need for backtracing. As I needed to make significant architectural changes to my JAL design when trying to implement the ideal JALR design afterwards. By considering compatibility from the outset, I can design systems that are more efficient and maintainable in the long term.

My first design query was how to route the return address (PC+4) to the register for storage. A naively could have made significant architectural changes, wiring the PC to a new PCplus4 module and using a MUX with a new control signal to input this into the RegWrite input. I simplified this approach by instead adding a “PCplus4” output to the existing PC. This simplified the design and avoided the need for additional modules. This output could be fed to the register for storage via the ALU by adding PCplus4 as an input to the MUX controlling input to the ALU.B input, extending ALUSrc select, so that ALU.B = ImmOp/RD2/PCplus4.

```
+     if (alusrc == 2'b01)
          alu_src_b = immext;
+     else if (alusrc == 2'b00)
          alu_src_b = regfile_d2;
+     else if (alusrc == 2'b10)
+         alu_src_b = PCplus4;
+     else
+         alu_src_b = 'b1111111;
```

Extending ALUControl width allowed space to add a JAL operation to the ALU which essentially ignores ALU.A input, simply feeding ALU.B = PCplus4 = ALUResult, storing this in register.

Lack of foresight meant I would soon learn this implementation is suboptimal, hogging the ALU. While JALR needs the ALU to calculate RS1 + ImmOp,

To alter JAL so the return address could be registered without the ALU, I disconnected PCplus4 from the ALU.B input, instead routing it directly into the Register.Write input via the result mux. This leaves the ALU free to calculate the target address for JALR and the J-type architecture could be much simplified. Although I would need to extend ResultSrc to 2 bits, I could reduce ALUSrc back to

1 bit (PCplus4 no longer being fed to ALU.B), I could remove the JAL operation from the ALU and reduce ALUCtrl back to 3-bits. Ultimately, this method is also much more scalable than the first design, which I've learnt from this project is an important factor to consider from the outset rather than concocting quick and immediate solutions.

J-type instructions require instead a 20-bit immediate. I extended ImmSrc to 2-bits and added the required 20-bit immediate form to signextend.sv for the select ImmSrc = 0b11:

I achieved jumping by hijacking the Branch control signal in the decoder, extending it to 2 bits so that Branch = 0b10 for a JAL instruction. I consequently had to extend PCSrc from 1 bit to 2 bits, so PC would select between (PC+4, PC+ImmOp(B-type), ImmOp (J-type). My mistake here was initially thinking JAL to not be PC-relative addressing and rather to jump directly to the given Immediate

```
+         2'b11: immop_o = {{12{toextend_i[31]}}, toextend_i[19:12],
    toextend_i[20], toextend_i[30:21], 1'b0}; //JAL Imm Form (20-bit imm)
```

operand. This was solved in my later implementation.

Truth table for JAL/R implementation:

Branch	Zero	PCSrc
00	0	00 (PC:=PC+4)
00	1	00 (PC:=PC+4)
01 (B-type)	0	00 (PC:=PC+4)
01 (B-type)	1	01 (PC:=PC+Imm12)
10 (JAL)	x	01 (PC:=PC+Imm20)
11 (JALR)	x	10 (PC:=ALUResult)

Initial finalised JAL commit

ALU is already setup to calculate RS1+Imm12 for I-type operations, which I could easily leverage for JALR rather than adding additional architecture. I assigned the control signals as follows:

ALUOp sets ALUControl to 0b000, which ALU evaluates as the case for ADD/ADDI, so it performs

```
7'b1100111 : begin //JALR
                ALUOp =  2'b11; //set to get ALU to ADD (PC+RS1+IMMOP)
                ALUSrc = 1'b1;
                ResultSrc = 2'b10;
                RegWrite = 1'b1;
                MemWrite = 1'b0;
                ImmSrc = 3'b000;
                Branch = 2'b11;
                end
```

RS1 + ImmOp, given ALUSrc = 1 so ALU.B input is Imm12. I fed this result as a new ALUResult input directly to the PC to the PC where it calculates PC+(ALUResult) to get the JTA. ResultSrc = 1'b10 selects PCplus4 being written to the register, so that we can store the return address.

Evaluation

My method of testing components at the start was very archaic. I was testing by essentially creating linear state machines to cycle through truth table inputs sequentially, which was disproportionally

laborious. Throughout this project I have refined my testing methodology and its efficiency. Simulating the top-level itself with assembled programs, cross-referencing the individual component outputs with expected outputs made debugging significantly easier. Avoiding the handling of corner cases from the start ended up being equally as unnecessarily time-consuming, as well as not having a clearly defined roadmap of the exact instructions we were going to implement in order to avoid having to alter control signals and their width several times over. If I had more time, I would have loved to implement the multi-cycle multiply function which I had designed. In future projects I would focus more on the implementation of specific instructions, as this is what most improved my understanding of RISCV and brought my intuition of its architecture to a level where I felt comfortable coming up with more creative solutions to different implementations.