

2 Exercises

These exercises will take you through the implementation of the following modules inside a single cycle RISC-V core: **ALU**, **Register Memory**, and **Decoder**. A specification for each will be provided along with an automated testing environment. A SystemVerilog file are to be submitted for each of these modules.

2.1 Exercise 1: Implementing the ALU

ALU Specification

The ALU is a purely combinational module that takes in 32-bit inputs (a, b) and calculates the alu_result based on the control signal. *Table 2* details the operation that should occur depending on the inputs.

Detailed information about the implementation of each can be found inside *Lab 1* or [HERE](#).

SystemVerilog ALU Header

```
module ALU (  
    output logic [31:0] alu_result,  
    output logic zero_flag,  
    output logic neg_flag,  
    output logic carry_flag,  
    input [31:0] a, b,  
    input [3:0] control_signals  
);
```

| control_signals | Operation |
|-----------------|-----------|
| 0000 | ADD |
| 1000 | SUB |
| 0001 | SLL |
| 0010 | SLT |
| 0011 | SLTU |
| 0100 | XOR |
| 0101 | SRL |
| 1101 | SRA |
| 0110 | OR |
| 0111 | AND |

Table 2: ALU control signals

Important notes:

- All branch comparisons are done in the SUB operation so flags need only be used here.
- The flags are used in the branch comparisons. neg_flag is used for signed numbers and carry_flag to check if unsigned numbers are negative.
- All signed numbers are stored in 2s complement.
- SystemVerilog types are unsigned by default – (`$signed(a) > $signed(b)`) and (`a > b`) can be evaluated differently¹.

Submission instructions

This exercise requires the completion of an alu.sv file, which can be found inside lab2/work/exercise_1/.

Run the command `./simulate.sh` inside the exercise_1 directory to test your design.

¹The SystemVerilog commands `$signed()` and `$unsigned()` can be used to let the simulator know how you would like it to treat whatever is in the brackets, which can be useful when dealing with comparisons.

2.2 Exercise 2: Register Memory

Register Memory specification

This module is responsible for housing the 32 core registers. It takes in 3 addresses **addr1**, **addr2**, and **rd**. **addr1** and **addr2** are used to read values in their respective locations and output **rs1** and **rs2**.

Wdata (a 1 bit value) indicates whether the instruction should write into the registers. If 1, then it will write. If 0 then it is reading only.

rd is the destination register, and indicates which address to store **data_in**.

SystemVerilog regMem Header

```
module regMem (  
    output logic[31:0] rs1,  
    output logic[31:0] rs2,  
    input Clock,  
    input nReset,  
    input [31:0] data_in,  
    input [4:0] addr1,  
    input [4:0] addr2,  
    input [4:0] rd,  
    input Wdata  
);
```

Important Notes:

- Writing into the registers is done synchronously (updates only on the posedge of clk)
- Reading from a register is done asynchronously² (in the same clock cycle).
- Attempting to read from location 0 should always return 0.
- Attempting to write into the location 0 will do nothing.
- Data will always be read every clock cycle, but data will not always be written.
- Upon negedge **nReset**, all 32 registers should be reset to 0.

Submission instructions

This exercise requires the completion of a `regMem.sv` file, which can be found inside `lab2/work/exercise_2/`.

Run the command `./simulate.sh` inside the `exercise_2` directory to test your design.

²Reading is usually done synchronously because memory can be made more efficient this way, however it is asynchronous here so that timing issues are avoided in this implementation. Further labs will go into more detail about this.

2.3 Exercise 3: Data memory

Data Memory Specification

Data memory is made up of many 8 bit registers. Although it could go up to 4.3e9, 1024 will be used for simulation ease.

This module is responsible for reading and writing data into registers. Control signals **func3**, **address**, **data**, and **write_data** are used.

SystemVerilog dataMem Header

```
module dataMem (  
  output logic [31:0] data_out,  
  input logic Clock,  
  input logic write_data,  
  input logic [31:0] address,  
  input logic [31:0] data_in,  
  input logic [2:0] func3  
);
```

| func3 | description |
|-------|----------------------|
| 000 | Load/Store byte |
| 001 | Load/Store half-word |
| 010 | Load/Store word |
| 100 | Load byte (U) |
| 101 | Load half (U) |

Table 3: data control signals

Data alignment

Figure 2 details how data is aligned inside the registers. @(address) signifies where the data should be stored/loaded. The leftmost 8 bits inside MEM[address]; the rightmost 8 bits inside MEM[address+3].

Similarly, for half-words – leftmost in MEM[address]; rightmost in MEM[address+1].

```
SW @(0)  0x1234_5678      LH @(2) = 0x0000_5678  
SH @(8)  0x0000_FEDC      LB @(8) = 0xFFFF_FFFE  
SB @(12) 0x0000_00FF      LBU @(8) = 0x0000_00FE
```

| REG | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|-------|----|----|----|----|---|---|---|---|----|----|----|----|----|----|----|----|
| VALUE | 12 | 34 | 56 | 78 | | | | | FE | DC | | | FF | | | |

Figure 2: Data alignment example

Important Notes:

- Writing to data memory is synchronous and reading to data memory is asynchronous.
- The data registers do not need to be reset or initialized³.
- Sign extending is only done when loading a signed half-word or byte.

Submission instructions

This exercise requires the completion of a dataMem.sv file, which can be found inside lab2/work/exercise_3/.

Run the command `./simulate.sh` inside the exercise_3 directory to test your design.

³Since there are far too many registers, they are allowed to be unknown. Programs cannot assume any values in RAM, and must store before loading to avoid undefined behavior.

2.4 Exercise 4: Branching Logic

Branch logic specification

Branch logic is responsible for passing signals to the program counter - It uses control signals to see if it is meant to jump and

SystemVerilog branch logic Header

```
module(  
    output logic [31:0] pc_next,  
    input [31:0] pc_in,  
    input [31:0] alu_result,  
    input [31:0] imm_in,  
    input [2:0] func3,  
    input [1:0] branch_type,  
    input neg_flag,  
    input carry_flag,  
    input zero_flag  
)
```

| func3 | description |
|-------|-------------|
| 000 | BEQ |
| 001 | BNE |
| 010 | BLT |
| 100 | BGE |
| 101 | BLTU |
| 111 | BGEU |

Table 3: branch_type key

Algorithm

```
switch (branch_type){  
    case NONE: pc_next = pc_in + 4;  
    case JAL: pc_next = pc_in + imm;  
    case JALR: pc_next = pc_in + alu_result;  
    case BRANCH:  
        switch (func3){  
            case BEQ: pc_next =  
                pc_in + ((zero_flag) ? alu_result : 4);  
            case BNE: // Your algorithm  
            case BLT: // Your algorithm  
            case BGE: // Your algorithm  
            case BLTU: // Your algorithm  
            case BGEU: // Your algorithm  
        }  
    }  
}
```

| branch_type | Description |
|-------------|-------------|
| 00 | NONE |
| 01 | JAL |
| 10 | JALR |
| 11 | BRANCH |

Table 4: branch_type key

Important Notes:

- The carry flag can be used to detect when the subtraction of unsigned numbers would be negative.

Submission instructions

This exercise requires the completion of a branchLog.sv file, which can be found inside lab2/work/exercise_4/.

Run the command ./simulate.sh inside the exercise_4 directory to test your design.

2.5 Exercise 5: Implementing the decoder

The decoder is the brain of the CPU, and is responsible for setting all of the control signals depending on the instruction. It is also responsible for extracting the immediate from the instruction formats, since it may be in different places.

STEP 1 – Extracting the immediate

Many times, we want to specify values inside the instruction itself to be passed into the program. This is the function of immediate values. They are stored in different ways depending on the instruction type inside the RV32I specification *Figure 2*.

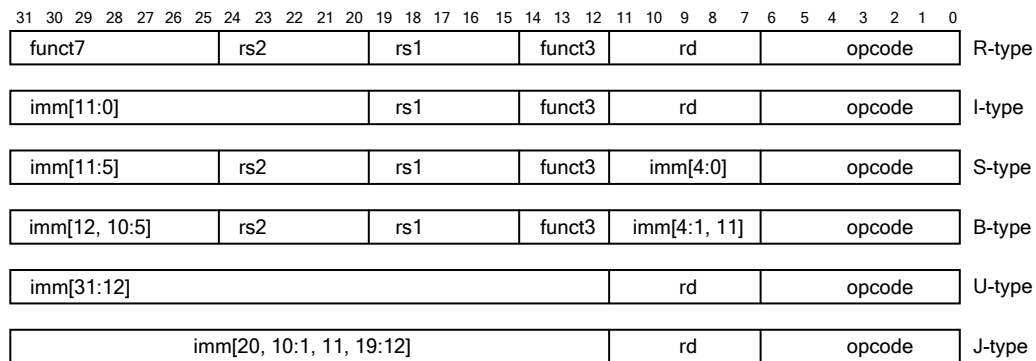


Figure 2: RV32I format

There are 9 different opcodes in the basic RV32I format, which can be seen in *table 2*. Some are one off instructions like LUI, AUIPC, JAL, and JALR, however the reset encapsulate many different instructions, using function 7 or function 3 to differentiate.

| Name | LUI | AUIPC | OPIMM | OP | LOAD | STORE | JAL | JALR | BRANCH |
|--------|-------|-------|-------|-------|-------|-------|-------|-------|--------|
| Opcode | 01101 | 00101 | 00100 | 01100 | 00000 | 01000 | 11011 | 11001 | 11000 |
| Type | U | U | I | R | I | S | J | I | B |

Table 2: OPcodes and values

Given the instruction as an input, differentiate inputs by their opcodes and return their corresponding 32-bit immediate.

Complete the function “get_immediate” inside of work/exercise_4/core_pkg.sv.

Important notes

- Ensure that there is a default case, returning 0.
- Ensure that all immediate values are sign extended.

STEP 2 – Setting the control signals

The control signals include the multiplexer signals **A** through **G**, **writeMem**, **writeReg**, **branch**, and **aluCode**.

With the exception of `aluCode`⁴, the control signals are the same for each opcode. by reference to the datapath *Figure 1*, whether the core branches, stores data, collects from ram, writes to the register memory, or calculates is decided by the control signals and multiplexers.

| OpCode | writeMem | writeReg | branch_type | A | B | C | D | E | G | alu_code |
|--------|----------|----------|-------------|---|---|---|---|---|---|-------------|
| LUI | 0 | 1 | NONE | 1 | 0 | 0 | 0 | 0 | 0 | ADD |
| AUIPC | 0 | 1 | NONE | 1 | 0 | 0 | 0 | 0 | 0 | ADD |
| OPIMM | 0 | 1 | NONE | 1 | 1 | 0 | 1 | 1 | 0 | * |
| OP | 0 | 1 | NONE | 1 | 1 | 1 | 1 | 1 | 0 | {f7, func3} |
| LOAD | 0 | 1 | NONE | 1 | 1 | 0 | 1 | 1 | 1 | ADD |
| STORE | 1 | 0 | NONE | 0 | 0 | 0 | 1 | 1 | 0 | ADD |
| JAL | 0 | 1 | JAL | 0 | 1 | 0 | 0 | 1 | 0 | ADD |
| JALR | 0 | 1 | JALR | 0 | 1 | 0 | 0 | 1 | 0 | ADD |
| BRANCH | 0 | 0 | BRANCH | 0 | 0 | 0 | 1 | 1 | 0 | SUB |

Table 4: Control signals for each opcode

***alucode** for `opImm` – Instructions `SRAI` and `SRLI` are differentiated using `func7`, while other instructions use this space for more immediate. This is a one off quirk with the RV32I specification, and needs to be handled here.

Handling the OPIMM case

```
OPIMM:
if (func3 == 3'b101)
    alucode = (f7) ? SRL : SRA;
else
    alucode = {1'b0, func3};
```

Using the datapath diagram *Figure 3*, the value tables, and with reference to the instructions - a comprehensive list of instructions can be found in the appendix - complete *Table 4* and implement this module in SystemVerilog in `lab2/work/exercise_5/decoder.sv`

To test your design, run command `./simulate` inside `lab2/work/exercise_5`.

⁴Sometimes relying on `func7` to differentiate between addition and subtraction and between arithmetic shifts and logical shifts.

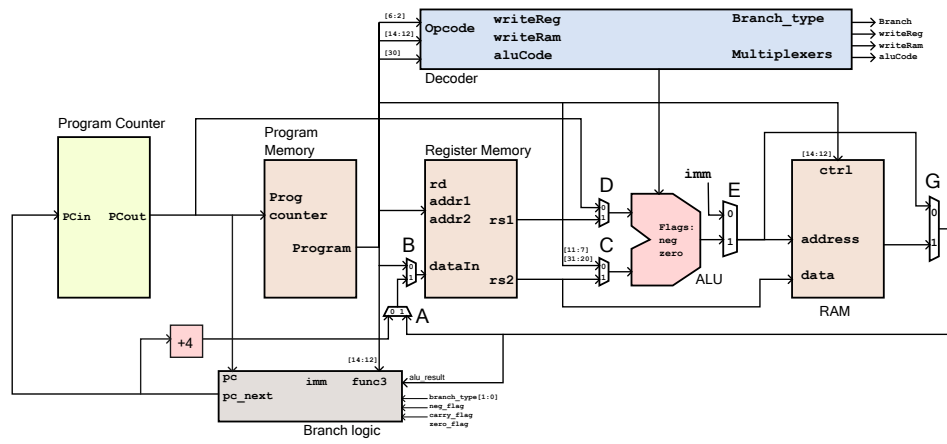


Figure 3: Annotated datapath