Customising a RISC-V Core (v1.1)

Originally presented at OSHCamp 2019

Authors

This workshop is prepared by:

- Graham Markall, Software Toolchain Engineer at Embecosm.
  Email: hello@big-grey.co.uk
- Sam Leonard, Software / Hardware Engineer Intern at Embecosm.
  Computer Science undergraduate at Manchester University.

Goals

The overarching goal of this workshop is to add fully functional and correctly operating support for new instructions to the RI5CY core, using a cycle-accurate Verilog simulation as the development platform. This breaks down to a number of sub-goals, each of which you will achieve throughout this workshop:

- Build a cycle-accurate model of RI5CY using Verilator and drive it with a C++ testbench.
- Explore the structure and operation of RI5CY using GTKWave.
- Learn about the RISC-V encoding formats, and be capable of designing new instructions based on standard formats.
- Examine how the RI5CY decoder works.
- Implement decoding of new instructions in RI5CY.
- Implement a new functional unit in the RI5CY execution stage.
- Define and connect control signals to enable decoding to control the new execution unit.
- Implement an instruction whose execution completes in a single cycle (no pipeline stall)
- Implement an instruction whose execution spans multiple cycles.
- Connect appropriate control signals to correctly stall the pipeline for a multi-cycle instruction.

Prerequisite knowledge

- Basic Verilog and C++.

- No prior knowledge of processor architecture or the RISC-V ISA assumed, but it will be helpful to refer to the OSHCamp presentation and presentation notes accompanying this workshop (included in the repository of materials).
- If you are struggling with a particular exercise, it may be helpful to refer to the `worked-solutions` branch of the cloned RI5CY repository – this contains a sequence of commits that complete all exercises.
- If you require a more basic introduction to Verilog, then consider:
  - https://gmarkall.wordpress.com/teaching – an introduction to Verilator with Verilog.
  - http://chiphack.org – an introduction to Verilog using the MyStorm FPGA board.

Prerequisite tooling

You will need, along with their dependencies:

- **Verilator:** https://www.veripool.org/wiki/verilator
- **Binutils for RISC-V.** There are various ways of installing RISC-V toolchains on different platforms each of which will include the RISC-V Binutils – the assembler (`gas`) is required for this tutorial. If you need to install it from source, the section below describes how.
- **GTKWave:** http://gtkwave.sourceforge.net/

Installing binutils

To obtain binutils from the Sourceware repositories, use the following commands (where `$PREFIX` is the folder you'd like it to be installed, and `$CORES` is the number of cores you have):

```
wget https://ftp.gnu.org/gnu/binutils/binutils-2.33.1.tar.gz
tar xf binutils-2.33.1.tar.gz
mkdir binutils-build
cd binutils-build
../binutils-2.33.1/configure --target=riscv32-unknown-elf –prefix=$PREFIX
make -j$CORES
make install
```

You will then need to add it to your path:

```
export PATH=$PREFIX/bin:$PATH
```

Then you should be able to invoke the assembler (example command and output):

```
$ riscv32-unknown-elf-as –version
GNU assembler (GNU Binutils) 2.33.1
Copyright (C) 2019 Free Software Foundation, Inc.
This program is free software; you may redistribute it under the terms of
the GNU General Public License version 3 or later.
```

```
This program has absolutely no warranty.
This assembler was configured for a target of `riscv32-unknown-elf'.
```

Slack chat and tutorial materials

The Slack channel (for discussions outside of the OSHCamp workshop venue is:

- https://oshcampri5cyworkshop.slack.com/

The repository containing workshop materials is at:

- https://github.com/gmarkall/oshcamp-ri5cy-workshop/

Exercise 1 – Getting to know the testbench

Open a terminal and verify that Verilator can be run:

```
verilator --help
```

Verilator is a tool for generating cycle-accurate simulations of Verilog code that can be driven using a C++ test bench. We we will use it for our simulation and testing throughout the process of adding new instructions.

You should clone the RI5CY repository and work on the new-instruction-base branch (or find it already in `~/workshop/ri5cy`):

```
git clone -b new-instruction-base https://github.com/gmarkall/ri5cy.git
```

Build the model:

```
cd verilator-model
```

```
make
```

This will build:

- The Verilated model – the Verilog source for this is in the `rtl/` folder of the repository, and the code generated by Verilator (object files and header files) is in `verilator-model/obj-dir/`.
- The testbench, which is in the file `testbench.cpp`.
- Example code for the simulated core to run, from `examples.s`.

You can execute the testbench:

```
./testbench
```

which should yield:

```
$ ./testbench
Reading program from examples.bin
Writing program to memory
```

```
        80: 00000513
        84: 00100593
        88: 00200613
        8c: 00300693
        90: 00400713
        94: 00500793
        98: 00600813
        9c: 00700893
        a0: 00c582b3
        a4: 00e68333
        a8: 00f703b3
        ac: 007302b3
        b0: 00000513
        b4: 00000593
        b8: 00000613
        bc: 05d00893
        c0: 00000073

    About to halt and set traps on exceptions
    About to resume
    Cycling clock to run for a few instructions
    Halting
```

The testbench has performed the following steps:

1. Instantiate the Verilator model, and perform some steps to reset the core (at the beginning of main).
2. Load in the program and write it to the core's memory (the loadProgram function)
3. Generated a clock input to the model, which enables it to execute once the core is out of reset (see the `clockSpin` function and the `CYCLES_TO_RUN_FOR` constant).

In the default setup, the clock is only run for about 20 cycles, but this can be modified (e.g. if you have a larger program that you're running) by editing the value of `CYCLES_TO_RUN_FOR`.

The testbench interacts with the model through its debug unit, which is custom to the version of the RI5CY core that this exercise is based around. It is documented in the RI5CY manual, but it is not necessary to look into it further for this workshop.

Further Resources

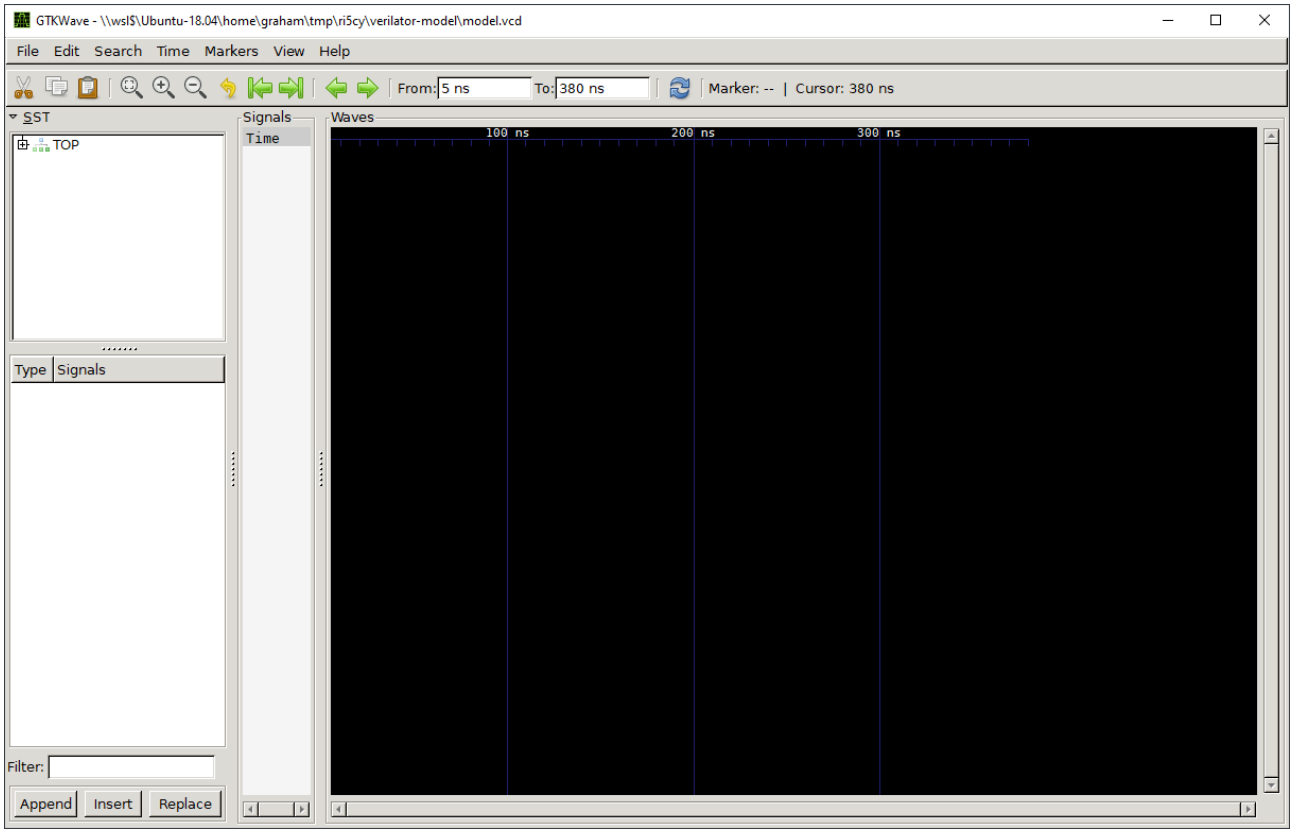
- The relevant version of the RI5CY manual can be located at https://github.com/gmarkall/ri5cy/blob/new-instruction-base/doc/user_manual.doc – it is not needed right now, but may be helpful to refer to later.

Exercise 2 – Visualising waves and exploring the core

The testbench also creates a Value Change Dump (VCD) file recording the state of every bit of logic in the core, which is used to inspect the state of the core at any moment in time. To get started, open GTKWave:

```
gtkwave model.vcd
```

You will initially be greeted with a window that has no signals displayed on it:



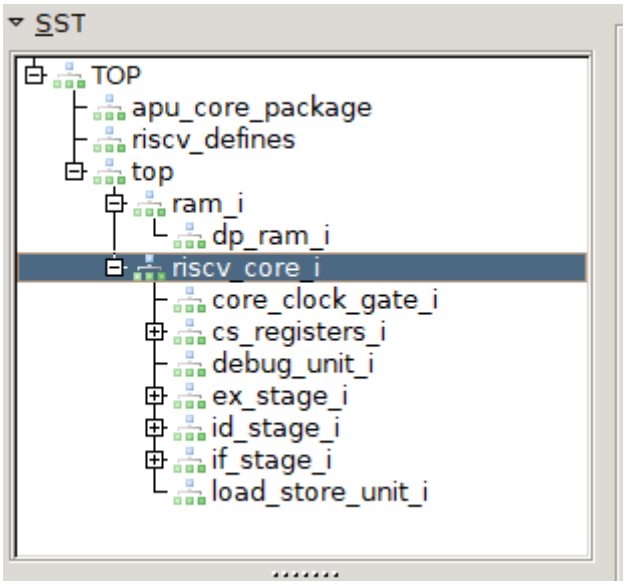
- The top left pane is for exploring the structure of the model – it is hierarchical, and matches the structure of the Verilog code from which the Verilator model was built
- The bottom left pane is for selecting signals within the current module selected in the top pane.
- The right pane displays the state of signals at each point in time.

Exploring the structure

At the very top of the model is the “TOP” module, and the “top” model beneath it. There are a couple of other modules, but the “top” module is of interest. It contains:

- **`ram_i`**: A small RAM, where both program and data memory is stored (though it is connected through two separate ports, an instruction port and a data port). The testbench `loadProgram` function interacts with this memory through Verilator tasks and functions.
- **`riscv_core_i`**: The RI5CY core itself.

Expanding the core module, we can see the components of the core:



The pipeline stages are, in order:

- **`if_stage_i`: Instruction fetch stage.** This contains a prefetch unit that loads instructions from the RAM through its instruction port, which is 128 bits wide.
- **`id_stage_i`: Decode stage.** This takes one instruction at a time from the fetch stage and sends control signals to the execution stage, and arranges for operands to be passed to the execution stage according to the specific instruction decoded.
- **`ex_stage_i`: Execute stage.** Contains all of the execution units - e.g. the Arithmetic Logic Unit (ALU), multiplier, and any other units that we might add.
- **`load_store_unit_i`: Writeback stage.** Connected to the RAM through its data port, and is used for reading operands from and writing results back to memory.

The two pipeline stages we are interested in are the Decode and Execute stages.

Viewing signals

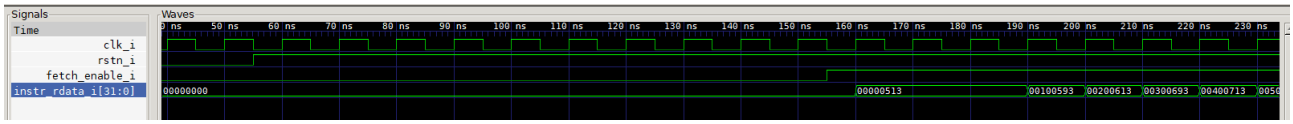
GTKWave lists all signals in the selected module. These can then be dragged to the middle pane to display them. Drag across the following signals from the following modules:

- **Clock:** `clk_i` from `top`

- **Reset:** `rstn_i` from `top`
- **Fetch enable:** `fetch_enable_i` from `top`
- **Instruction in ID stage:** instr_rdata_i from top → riscv_core_i → id_stage_i

It may be necessary to zoom in or out to get a good view, which can be done with GTKWave's toolbar buttons – they look like one of these two variants:

Once you have done this, you should have a view like this in the signals and waves windows:



We can understand what is happening as:

- Clock pulses are generated as time advances.
- The core is held in reset initially, and then brought out of reset. Note that the reset pin is active low, so the core is in reset whilst `rstn_i` is 0, and out of reset when it is 1.
- Some time later, instruction fetch is enabled.
- Once instruction fetch is enabled, instruction words start reaching the instruction decoder. You should be able to see that these instruction values correlate with those printed by the testbench during its execution (or see the example output above in Exercise 1)

Further exploration

Straightforward (worth a quick look):

- Examine the code of the test bench and identify how the clock, reset, and fetch enable signals are set. You will need to look at `testbench.cpp` and `top.sv`.

More difficult (optional, but recommended):

- This second exploration demonstrates that although block diagrams show each component appearing to be relatively separate, the interactions can flow through many connections

between modules, and it is helpful to be proficient at navigating the source through different levels of the implementation.

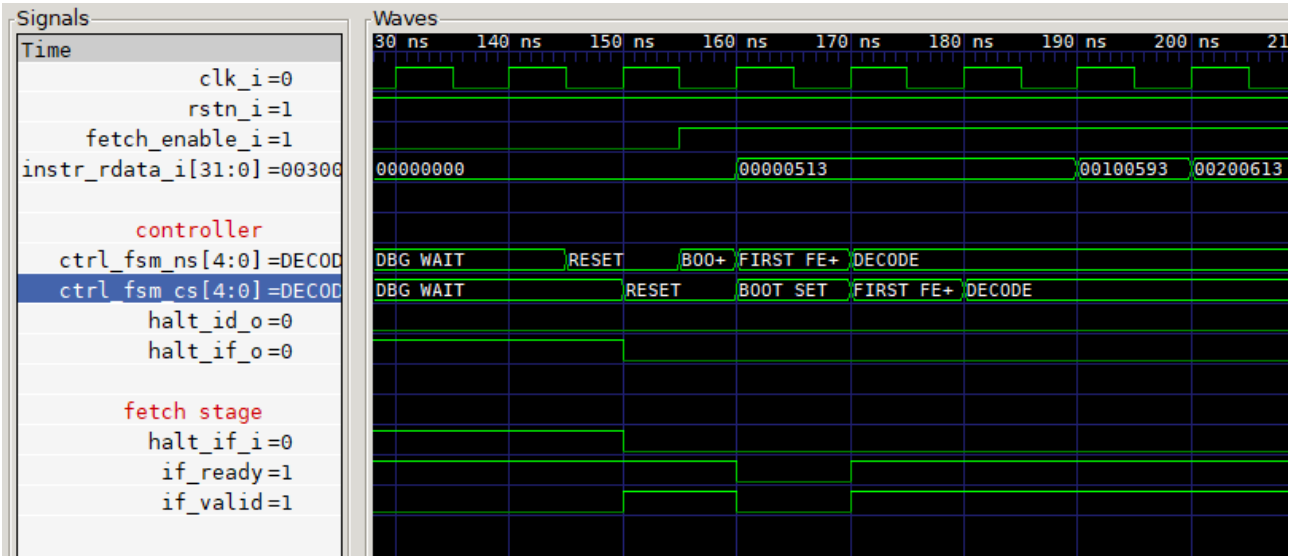
- Task: trace through the Verilog to see how the `fetch_enable_i` signal set by the testbench affects the decoder and instruction fetch modules (e.g. `riscv_if_stage.sv`, `riscv_id_stage.sv`, `riscv_decoder.sv`) – i.e.:
  - What is it connected to?
  - What do the things it is connected to influence? What state can be affected?

Some notes on the answer:

- In `module top`: `fetch_enable_i` → `riscv_core_i fetch_enable_i` (see `top.sv`)
- In `module riscv_core`: `fetch_enable_i` → `id_stage_i fetch_enable_i` (see `riscv_core.sv`)
- In `module riscv_id_stage`: `fetch_enable_i` → `controller_i fetch_enable_i` (see `riscv_id_stage.sv`)
- In `module riscv_controller`: `fetch_enable_i` being `1` causes a transition of the state machine controlled by `ctrl_fsm_cs` (derived from `ctrl_fsm_ns`):
  - Initially it moves from `RESET` to `BOOT_SET`.
  - The controller then proceeds to the `FIRST_FETCH` state.
  - In the `FIRST_FETCH` state, if the decoder is ready, then the controller proceeds to the `DECODE` state.
  - When in the `DECODE` state, the controller sets the `halt_if_o` signal to `0` (from an initial value of `1`).
  - See the assignments starting around line 206 of `riscv_controller.sv`, and the state machine states from line 255-455 (the unique_case statement on `ctrl_fsm_ns`).
- `halt_if_o` is then connected from `id_stage_i` to `halt_if` in the `riscv_core` module, and to the `halt_if_i` pin of `if_stage_i` in the `riscv_core` module
- In `riscv_if_stage`, the `halt_if_i` input is used to set instruction fetch into a not valid / not ready state (lines 394-395 of riscv_if_stage.sv):

```
assign if_ready = valid & id_ready_i;
assign if_valid = (~halt_if_i) & if_ready;
```

Some of the state propagated can be visualised in GTKWave by adding signals from the controller (top→ riscv_core_i → id_stage_i → controller_i) and fetch stage (top→ riscv_core_i→ if_stage_i) to GTKWave. An example of these signals around the time of fetch being enabled follows:



A couple of GTKWave tips

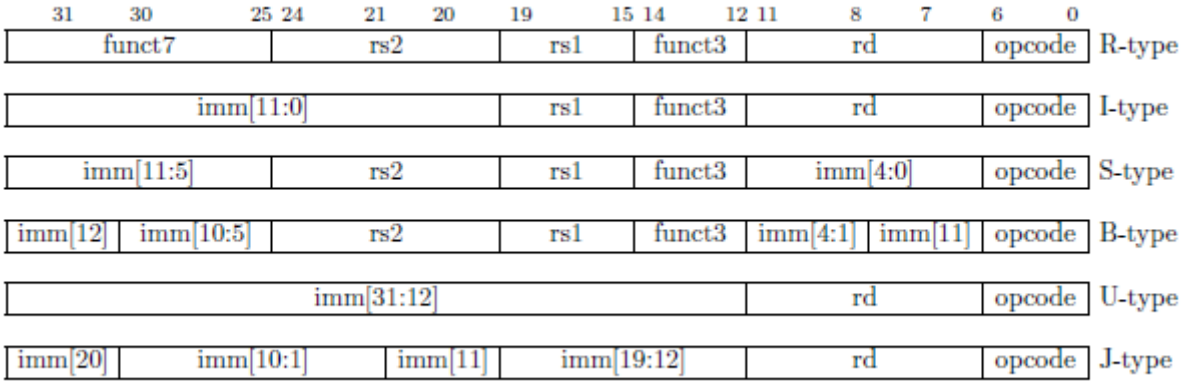
- You can insert comments (e.g. “controller”, “fetch stage” above), and blank lines to space things out by right-clicking the Signals pane.
- Interpreting numeric values is often tedious and mechanical. A filter file can map numbers to alphanumeric entities – in the above example these are the controller FSM states. The filter file mapping FSM numeric states to names contains:

```
00 RESET
01 BOOT_SET
02 SLEEP
03 WAIT_SLEEP
04 FIRST_FETCH
05 DECODE
06 IRQ_TAKEN_ID
07 IRQ_TAKEN_IF
08 IRQ_FLUSH
09 ELW_EXE
0A FLUSH_EX
0B FLUSH_WB
0C DBG_SIGNAL
0D DBG_SIGNAL_SLEEP
0E DBG_SIGNAL_ELW
0F DBG_WAIT
10 DBG_WAIT_BRANCH
11 DBG_WAIT_ELW
```

- Filters can be applied by right-clicking the signal and clicking Data Format→ Translate Filter File→ Enable and Select, and then selecting the text file containing the filter values. See page 57/58 of the GTKWave manual for more details of filters: http://gtkwave.sourceforge.net/gtkwave.pdf

Exercise 3 – Understanding RISC-V encodings

Before we can start work on the implementation of a new instruction, the encoding for the instruction must be defined. The RISC-V architecture provides us with some standard instruction formats, that instructions can fit into (from the RISC-V specification S 2.3, page 17):

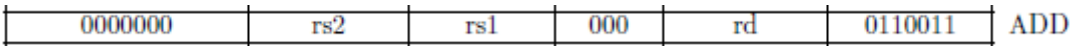


All of these instruction formats occupy 32 bits. The different fields within formats are:

- **Opcode (7 bits):** Defines the kind of instruction. A full ist can be found on Page 129 of the RISC-V specification. Some examples include:
  - 0x03: Load
  - 0x23: Store
  - 0x33: Op (arithmetic operations)
  - 0x63: Branch
  - 0x73: System
- **funct3 (3 bits):** Defines the operation performed by the instruction more specifically within the opcode. For example, the Load opcode (0x03) has the following variants:
  - 000: Load byte (8 bits)
  - 001: Load halfword (16 bits)
  - 010: Load word (32 bits)
  - 100: Load byte unsigned (8 bits)
  - 101: Load halfword unsigned (16 bits)

- **funct7 (7 bits):** Defines the operation performed by the instruction more specifically within the opcode. This is usually hardcoded to 0000000 for most standard instructions that use it.
- **rd (5 bits):** The destination register, from 0 to 31.
- **rs1 (5 bits):** Source register 1, from 0 to 31.
- **rs2 (5 bits):** Source register 2, from 0 to 31.
- **imm (12-20 bits):** An immediate value with length depending on the instruction format.

An example of a standard instruction encoding (RISC-V spec P. 130) is the add instruction:



This is an example of an R-type instruction with the arithmetic operation opcode 0x33 (0110011), with funct3 (000) and funct7 (0000000) to specify an add operation (as opposed to subtract, shift left, etc., which use other funct3 and funct7 values).

Encoding an add instruction

Given an instruction like:

```
add x5, x8, x9
```

how does this fit into the encoding?

- rd = 5 = 00101
- rs1 = 8 = 01000
- rs2 = 9 = 01001

Slotting these into the rest of the encoding yields:

Funct7	Rs2	Rs1	Funct3	Rd	Opcode
0000000	01001	01000	000	00101	0110011

As a binary string: 00000000100101000000001010110011

As hexadecimal: 0x009402B3

A note on RISC-V register names

RISC-V has two sets of register names:

- Architectural names, x0-x31. These correspond directly to the numbers of fields in rd, rs1, rs2, etc., so are easy to work with when thinking about the software.

- ABI names. These have names like a0, a1, …, a7, s0, s1, etc. These are easier for software developers to think about, and there is a mapping between ABI names and architectural names (defined in the RISC-V ELF psABI document). It is:

Name	ABI Mnemonic	Meaning
x0	zero	Zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5-x7	t0-t2	Temporary registers
x8-x9	s0-s1	Callee-saved registers
x10-x17	a0-a7	Argument registers
x18-x27	s2-s11	Callee-saved registers
x28-x31	t3-t6	Temporary registers

Exercise

(Optional – straightforward but mechanical) Using the encodings on P. 130 of the RISC-V specification, encode the following instructions:

```
lui a0, 0x74657
addi a0, a0, 1463
```

Resources

- The RISC-V specification v 2.3 can be found at https://content.riscv.org/wp-content/uploads/2019/06/riscv-spec.pdf
- The RISC-V ELF psABI document is at https://github.com/riscv/riscv-elf-psabi-doc/blob/master/riscv-elf.md

Exercise 4 – Understanding the RI5CY decoder

The decoder is in `riscv_decoder.sv`. There is a large case statement that decides what to do based on which bits are set in the instruction - at the highest level, the switching is based on the

opcode of the instruction. The top of the case statement starts at line 253, and the first case handles decoding of a JAL (jump and link) instruction:

```
unique case (instr_rdata_i[6:0])

  //////////////////////////////////////
  //      _ _   _ __  __ ____  ____   //
  //     | | | | |  \/  |  _ \/ ___|  //
  //  _  | | | | | |\/| | |_) \___ \  //
  // | |_| | |_| | |  | |  __/ ___) | //
  //  \___/ \___/|_|  |_|_|   |____/  //
  //                                  //
  //////////////////////////////////////

  OPCODE_JAL: begin   // Jump and Link
    jump_target_mux_sel_o = JT_JAL;
    jump_in_id            = BRANCH_JAL;
    // Calculate and store PC+4
    alu_op_a_mux_sel_o  = OP_A_CURRPC;
    alu_op_b_mux_sel_o  = OP_B_IMM;
    imm_b_mux_sel_o     = IMMB_PCINCR;
    alu_operator_o      = ALU_ADD;
    regfile_alu_we      = 1'b1;
    // Calculate jump target (= PC + UJ imm)
  end
```

The instruction being decoded is held in instr_rdata_i, so instr_rdata_i[6:0] selects out exactly the opcode bits. The first case handled, which we will examine, is `OPCODE_JAL`. Each opcode is defined in the RI5CY implementation in the `rtl/include/riscv_defines.sv` file, e.g.:

```
parameter OPCODE_SYSTEM    = 7'h73;
parameter OPCODE_FENCE     = 7'h0f;
parameter OPCODE_OP        = 7'h33;
parameter OPCODE_OPIMM     = 7'h13;
parameter OPCODE_STORE     = 7'h23;
...
parameter OPCODE_JAL       = 7'h6f;
...
```

The above are a few examples – there are several more defined in the file.

Understanding the JAL instruction

It is the job of the decoder to set appropriate signals for controlling the execution stage. We will first look at the JAL instruction, whose implementation only depends on the opcode. To execute the JAL instruction, the core needs to:

- Use the ALU to calculate the return address of the instruction (`PC + 4`)
- Store the result (in `x1` by default, though other registers can be used)
- Calculate the address of the target instruction
- Write the target address to the `PC` so that execution resumes from that new address/

The first two lines set signals required for control of the core:

```
jump_target_mux_sel_o = JT_JAL;
jump_in_id            = BRANCH_JAL;
```

The next few lines set up the ALU control signals: – Operand A to be obtained from the program counter, Operand B from the immediate encoded in the instruction[1] and also an increment of the PC by one instruction, the ALU operation is an addition, and the ALU will write back to the register file.

```
// Calculate and store PC+4
alu_op_a_mux_sel_o  = OP_A_CURRPC;
alu_op_b_mux_sel_o  = OP_B_IMM;
imm_b_mux_sel_o     = IMMB_PCINCR;
alu_operator_o      = ALU_ADD;
regfile_alu_we      = 1'b1;
```

Understanding branches – decoding based on funct3

A slightly more complex example is the branch opcode, which can perform several different operations – one for each of its variants:

- BEQ: Branch Equal
- BNE: Branch Not Equal
- BLT: Branch Less Than
- BGE: Branch Great than or Equal
- BLTU: Branch Less Than Unsigned
- BGEU: Branch Greater than or Equal Unsigned

The branch opcode case in the decoder is around line 295:

```
OPCODE_BRANCH: begin // Branch
  jump_target_mux_sel_o = JT_COND;
  jump_in_id            = BRANCH_COND;
  alu_op_c_mux_sel_o    = OP_C_JT;
  rega_used_o           = 1'b1;
  regb_used_o           = 1'b1;
```

1 Immediates are extracted from the instruction elsewhere, in `riscv_id_stage.sv` – see around line 446.

```
  unique case (instr_rdata_i[14:12])
    3'b000: alu_operator_o = ALU_EQ;
    3'b001: alu_operator_o = ALU_NE;
    3'b100: alu_operator_o = ALU_LTS;
    3'b101: alu_operator_o = ALU_GES;
    3'b110: alu_operator_o = ALU_LTU;
    3'b111: alu_operator_o = ALU_GEU;
    3'b010: begin
      alu_operator_o      = ALU_EQ;
      regb_used_o         = 1'b0;
      alu_op_b_mux_sel_o  = OP_B_IMM;
      imm_b_mux_sel_o     = IMMB_BI;
    end
    3'b011: begin
      alu_operator_o      = ALU_NE;
      regb_used_o         = 1'b0;
      alu_op_b_mux_sel_o  = OP_B_IMM;
      imm_b_mux_sel_o     = IMMB_BI;
    end
  endcase
end
```

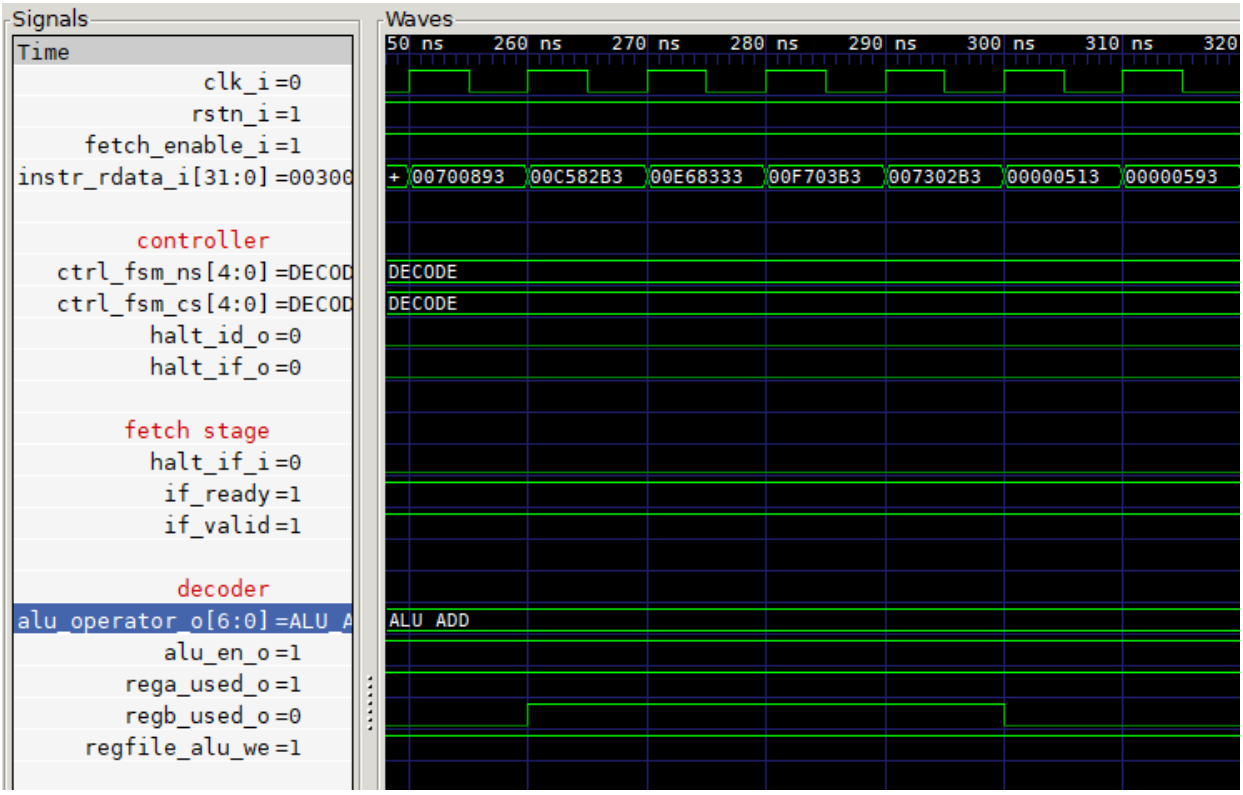
Note how funct3 is extracted as `instr_rdata_i[14:12]` and used to control the ALU operator depending on the type of branch.

Exercise – understanding the decoding of add

- An instruction is an add when it has an opcode of 0x33, funct3 of 0, and funct7 of 0. What is the definition for the opcode in `riscv_defines.sv`?
- Locate the case handling this opcode in the decoder's case statement.
- How does this case distinguish an add from other instructions?
- How does it distinguish an add from a subtract?

Notes:

- RI5CY also has some custom bit-manipulation instructions that are handled alongside this case – these will not be found in the RISC-V specification.
- Adding signals to GTKWave from the decoder may help in understanding what the decoder sets. For example:



- The instruction `00C582B3` corresponds to `add t0, a1, a2` from the example code loaded by the testbench.
- When this is decoded, the `regb_used_o` signal is asserted, signalling that the ALU's second operand is from the register file.
- It may be helpful to create a GTKWave filter file for alu_operator_o – you can find the mappings of numeric values to strings (e.g. `ALU_ADD`) in `riscv_defines.sv`.

Exercise 5 – designing a new instruction

Now we understand how decoding works we can modify the decoder. The specification for our new instruction should define:

- Which instruction format it uses (e.g. R, I, S, etc.)
- What its semantics are?
  - In other words, what does it do?
  - If it accepts an immediate, what is the immediate for?
  - It would be prudent to choose semantics that are something you can manage to implement in Verilog.
- What is its opcode?
- Are there any variants distinguished by using different funct3, funct7 values, etc?

The RISC-V opcode space is generally quite full already:

inst[4:2] / inst[6:5]	000	001	010	011	100	101	110	111 (> 32b)
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	48b
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	≥ 80b

The *custom-0/-1/-2* spaces are available for custom instructions. However, RI5CY already has quite a lot of custom instructions which take up this space already. In order to free up some opcode space, the version of RI5CY used for this workshop has had two classes of custom instructions removed:

- Post-incrementing loads, which take the *custom-0* space, opcode 0x0B.
- Post-incrementing stores, which take the *custom-1* space, opcode 0x2B.

So, these two opcodes are available for use.

Examples

This section provides some variants on an example instruction – you may wish to follow the rest of the tutorial using this example, then come back and design your own instruction to implement afterwards. These example instructions implement various string operations on the contents of registers.

- Opcode: 0x0B
- Instruction format: I
- Funct3 variants:
  - 000: UPPER
  - 001: LOWER
  - 010: LEET
  - 011: ROT13
- Immediate value: ignored (may always be hardcoded to 0).

Semantics

UPPER

Example usage: upper rd, rs1

Description: Converts the 4-character ASCII string in rs1 to uppercase.

Example:

```
li a0, 0x746575b7   # "test"
upper a1, a0        # a1 = "TEST"
```

`Example`

LOWER

Example usage: lower rd, rs1

Description: Converts the 4-character ASCII string in rs1 to uppercase.

Example:
```
li a0, 0x54455354   # "TEST"
lower a1, a0        # a1 = "test"
```

LEET

Example usage: leet rd, rs1

Description: Converts the 4-character ASCII string in rs1 to "leet-speak".

Example:
```
li a0, 0x746575b7   # "test"
leet a1, a0         # a1 = "t35t"
```

ROT13

Example usage: rot13 rd, rs1

Description: Applies the ROT13 transformation to the 4-character ASCII string in rs1.

Example:
```
li a0, 0x746575b7   # "test"
upper a1, a0        # a1 = "grfg"
```

Exercise

(Optional) Define your own custom instruction's encoding and semantics.

Exercise 6 – Implementing decoding of new instructions

In order to start implementing new instructions, a skeleton of the decoding logic must first be implemented and verified to be working correctly. To do this:

- Add a definition for `OPCODE_STR_OPS` (or alternative name if you are using an instruction of your own design) to `riscv_defines.sv`. This should match the opcode of your instruction (either 0x0B or 0x2B, depending on your choice).
- Add an additional case to the big case statement in `riscv_decoder.sv` for your new opcode. It should print out which kind of instruction is decoded. If the instruction does not match any valid encoding for your opcode, it should set `illegal_insn_o` to 1 to signal that the instruction is not valid. You may need to add another case switch inside your case

statement to differentiate between the different funct3 values (as is the case with the string operations).

  - Printing can be done in Verilog with `$display`.
  - Example: `$display("Decoded upper instruction");`

- To trigger the decoding logic you have just added, you need to add some instances of your instructions to the example code that the testbench runs. Since these instructions are custom, the assembler doesn't have any support for them, but it has a mechanism for encoding custom instructions. The next section outlines how to add these instructions.
  - At present the new instructions won't have any effect, because there is no implementation of anything in the execution stage.
  - This is OK for now – it is important to test and validate changes at each step.
  - Further exercises will walk through implementing execution for instructions.

Encoding custom instructions with the RISC-V assembler

Rather than manually encoding instructions for testing purposes, we will make use of a feature of the RISC-V assembler that allows us to encode instructions for any custom instruction – the .insn directive. This provides the ability to specify a custom instruction format and its operands. The general form of using the directive looks like:

.insn <type> opcode <parameters>

where the parameters depend on the type. A full list of the types and formats is in the binutils documentation, which can be accessed at https://sourceware.org/binutils/docs-2.32/as/RISC_002dV_002dFormats.html#RISC_002dV_002dFormats – here, we will look into the I format because it is used for the string operations in our examples. The format is:

```
I type: .insn i opcode, func3, rd, rs1, simm12
+-------------+-----+-------+----+-------------+
|      simm12 | rs1 | func3 | rd |      opcode |
+-------------+-----+-------+----+-------------+
31            20    15      12   7             0
```

So we can implement some example instructions with directives such as:

```
.insn i 0x0b, 0, x0, a1, 0 # upper x0, a1
.insn i 0x0b, 0, a5, a1, 0 # upper a5, a1
.insn i 0x0b, 1, a5, a1, 0 # lower a5, a1
.insn i 0x0b, 2, a5, a1, 0 # leet a5, a1
.insn i 0x0b, 3, a5, a1, 0 # rot13 a5, a1
```

Add these (or some suitable other directives for your own instruction) to `verilator-model/examples.s` and run make in the `verilator-model` directory to rebuild the code that the testbench loads in.

Execution

Run the testbench again, and you should see your changes to the decoder printing when these new instructions are decoded, e.g.:

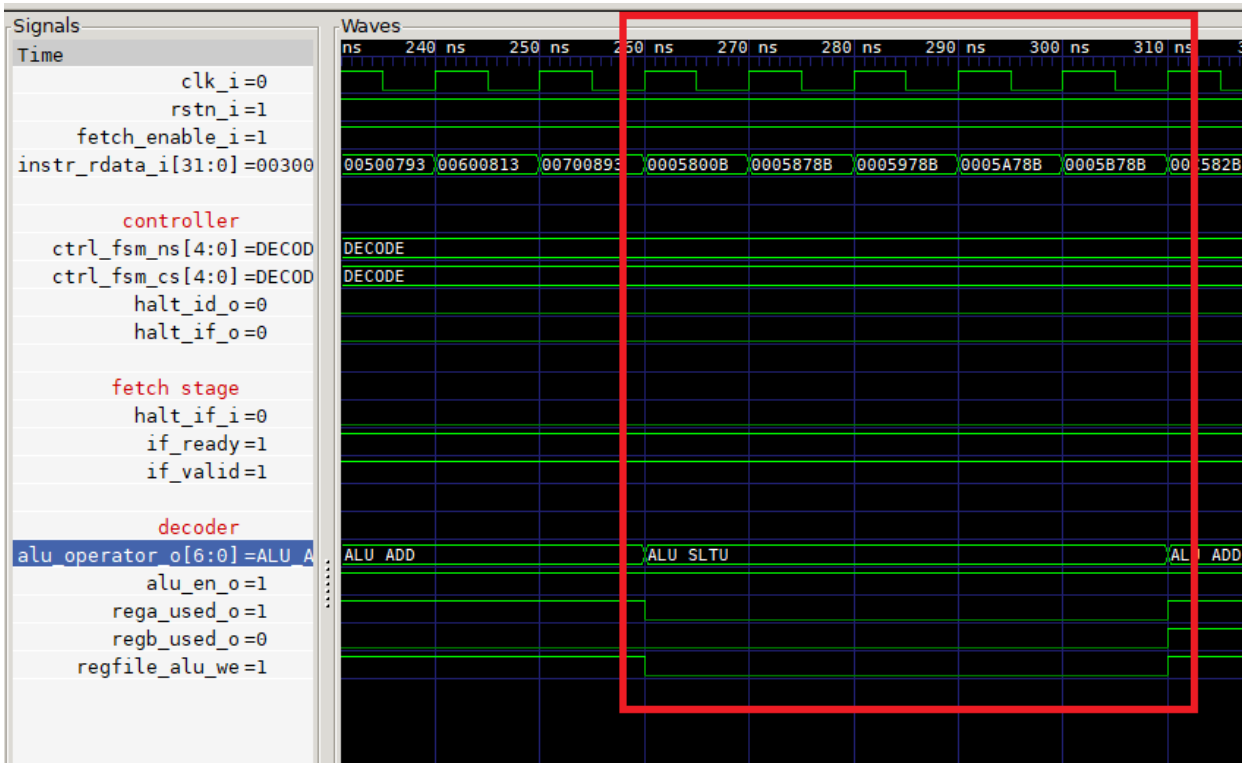
```
About to resume
Cycling clock to run for a few instructions
Decoded upper instruction
Decoded upper instruction
Decoded lower instruction
Decoded leet instruction
Decoded rot13 instruction
Halting
```

Sometimes you may see multiple messages per instruction (e.g. “Decoded lower instruction” multiple times even if there is only one instance of a lower instruction) – this is an artifact of the simuation and the way in which we are testing the implementation. The main consideration is that a message should appears at least once for each instruction.

Troubleshooting

If you are having difficulty with getting your prints to appear from the decoder, the following techniques may be helpful:

- Check the encoding of your instruction looks correct by running `riscv32-unknown-elf-objdump -dr examples.elf` – this will dump the encoding of each instruction in the compiled code. Converting the hexadecimal representations to binary and checking they match what you expect to be encoded may be helpful.
- Open the trace file in GTKWave and check that you see the instructions you have added appearing in instr_rdata_i of the decoder, e.g. (relevant instructions in the red box):



Exercise 7 – Connecting an execution unit

This exercise is more complex than previous ones, as there is a lot of “plumbing” to do to connect the decoder in the ID stage to an execution unit in the EX stage – it may be helpful to refer to the worked solution for this exercise.

To create an execution unit and control it with the decoder, we need to:

- Create new constants to represent the operations the execution unit can perform.
- Implement a new execution unit
- Set signals in the decoder to control the execution unit.
- Instantiate the execution unit in the ex stage
- Connect control signals starting from the decoder (`riscv_decoder.sv`), through:
  - The ID stage (`riscv_id_stage.sv`)
  - The Core module (`riscv_core.sv`)
  - The EX stage (`riscv_ex_stage.sv`)
  - The execution unit.
- Update the Makefile to include the new execution unit’s source in the Verilator model.

This gives us the structure to start implementing execution, but won't actually execute anything for now – instead, we will add debugging prints to the execution unit so that we can verify that signals from the decode stage are successfully routed.

Adding constants to represent operations

In `riscv_defines.sv`, there are many constants defined for control of execution units. For example for ALU control, some of the definitions are:

```
parameter ALU_OP_WIDTH = 7;

parameter ALU_ADD   = 7'b0011000;
parameter ALU_SUB   = 7'b0011001;
parameter ALU_ADDU  = 7'b0011010;
```

The operation width is determined by how many different operations there are – it should be wide enough to give each operation a distinct value. The width is used in the declaration of signals that hold these constants, so that they can consistently be declared with the exact width required – and if more ALU operation constants are added in the future, the width can be increased by changing only the definition here rather than editing many signals across many units.

We need to add the width of string operations and one constant for each operation. For the four operations in our example, this would look like:

```
parameter STR_OP_WIDTH = 2;

parameter STR_OP_UPPER = 2'b00;
parameter STR_OP_LOWER = 2'b01;
parameter STR_OP_LEET  = 2'b10;
parameter STR_OP_ROT13 = 2'b11;
```

Implementing a new execution unit

Create a new SystemVerilog file in the rtl/ folder with an appropriate name - for the string operations, riscv_str_ops.sv is an appropriate name. This file needs to contain a module for the execution unit.

Because the execution unit will make reference to the values we have defined in riscv_defines.sv for its control, we need to first import its values:

```
import riscv_defines::*;
```

Next we can define the module interface. We will start with a bare minimum interface, that has:

- Clock input: practically every module requires the clock input
- Enable input: for the decoder to enable the unit when a string operation has been decoded.
- Operator input: for the decoder to communicate which string operation is required.

A module interface that implements this could look like:

```
module riscv_str_ops
(
  input logic                     clk,
  input logic                     enable_i,
  input logic [STR_OP_WIDTH-1:0] operator_i
);
```

Next we need to add some implementation to the module. For now we will simply print out which operator is required, when the execution unit is enabled (when the unit is not enabled, there will still be some input on operator_i, but it will have no meaning so it is not worth printing anything out when the unit is not enabled). We can implement this with the following:

```
  always_ff @(posedge clk)
  begin
    if (enable_i) begin
      case (operator_i)
        STR_OP_UPPER:
          $display("%t: Exec Upper instruction", $time);
        STR_OP_LOWER:
          $display("%t: Exec Lower instruction", $time);
        STR_OP_LEET:
          $display("%t: Exec Leet speak instruction", $time);
        STR_OP_ROT13:
          $display("%t: Exec Rot13 instruction", $time);
      endcase
    end
  end
```

This example code also uses the `$time` function, which allows us to print out the simulation time with each statement.

This completes everything required for our very simple execution unit. You will need to terminate the module with:

```
endmodule
```

at this point.

Setting signals in the decoder

Recall from the previous exercise that we have a case statement in the decoder for the string operations:

```
OPCODE_STR_OPS: begin
  if (instr_rdata_i[14:12] == 3'b000) begin
    // Upper case
    $display("Decoded upper instruction");
  end
  else if (instr_rdata_i[14:12] == 3'b001) begin
    // Lower case
    $display("Decoded lower instruction");
```

```
  end
  else if (instr_rdata_i[14:12] == 3'b010) begin
    // Leet speak
    $display("Decoded leet instruction");
  end
  else if (instr_rdata_i[14:12] == 3'b011) begin
    // ROT13
    $display("Decoded rot13 instruction");
  end
  else
    illegal_insn_o = 1'b1;
end
```

We will need to modify this so that it no longer prints out when an instruction has been decoded (the execution unit will instead print out messages under the control of decode). Instead, we will set the signals to control the execution unit here.

There are various outputs from the decoder for controlling each execution unit, which you can see in the declaration of the module interface. Some of the signals controlling the ALU are (around line 77):

```
output logic                     alu_en_o,       // ALU enable
output logic [ALU_OP_WIDTH-1:0] alu_operator_o, // ALU operation selection
```

These correspond to inputs in the ALU. We will need to add similar signals for our string operations, that correspond to the inputs for our string operation execution unit. We could add these two signals to the decoder module interface:

```
output logic [STR_OP_WIDTH-1:0] str_operator_o,
output logic str_op_en_o,
```

Other signals set by many of the ALU operations are:

```
output logic        rega_used_o,   // rs1 is used by current instruction
output logic        regb_used_o,   // rs2 is used by current instruction
```

These are set when an instruction uses either rs1 and/or rs2 as operands – when they are set, logic in the pipeline ensures that the operands are available to the instruction execution. Since our string operations only use one source register, we only need to assert `rega_used_o` – other custom instructions that use two operands would also need `regb_used_o` asserting.

Now, we can modify the decoding case block to set these signals, e.g.:

```
OPCODE_STR_OPS: begin
  if (instr_rdata_i[14:12] == 3'b000) begin
    // Upper case
    str_op_en_o = 1'b1;
    str_operator_o = STR_OP_UPPER;
    rega_used_o = 1’b1;
  end
  else if (instr_rdata_i[14:12] == 3'b001) begin
    // Lower case
```

```
        str_op_en_o = 1'b1;
        str_operator_o = STR_OP_LOWER;
        rega_used_o = 1’b1;
      end
      else if (instr_rdata_i[14:12] == 3'b010) begin
        // Leet speak
        str_op_en_o = 1'b1;
        str_operator_o = STR_OP_LEET;
        rega_used_o = 1’b1;
      end
      else if (instr_rdata_i[14:12] == 3'b011) begin
        // ROT13
        str_op_en_o = 1'b1;
        str_operator_o = STR_OP_ROT13;
        rega_used_o = 1’b1;
      end
      else
        illegal_insn_o = 1'b1;
end
```

This covers all the cases when we are decoding a string operation instruction, but what happens to these signals when some other instruction is decoded? We need to make sure that when we are not decoding a string operation, appropriate outputs are produced - i.e. the enable is set to 0, but the operator doesn’t matter. Prior to the big decoding case statement, starting around line 180, all of the outputs of the decoder are set to default values. There are many, but for the ALU, some are:

```
alu_en_o                    = 1'b1;
alu_operator_o              = ALU_SLTU;
```

It is actually the default that the ALU is enabled – this makes sense because many instructions use it, and those that don’t will instead explicitly disable it. The default operator value is the ALU constant which is defined as 0. For our string operations, it makes more sense to disable the unit by default, as our string op decoding block will explicitly set the enable output to 1. So we can add along with the other assignments in this area something like:

```
str_op_en_o                 = 1'b0;
str_operator_o              = STR_OP_UPPER;
```

Instantiating the execution unit in the EX stage

Look through the riscv_ex_stage.sv file – you will see that various execution units are instantiated:

- riscv_alu
- riscv_mult
- riscv_apu_disp
- fpu_private

You may wish to spend a short time examining the connections of the pins of the execution units to get a feel for how things fit together in the EX stage.

Specifically looking at the instantiation of the ALU, we see that some of its inputs are connected to inputs to the execution unit:

```
riscv_alu
#(
  .SHARED_INT_DIV( SHARED_INT_DIV ),
  .FPU           ( FPU            )
  )
 alu_i
(
  .clk                ( clk             ),
  .rst_n              ( rst_n           ),
  .enable_i           ( alu_en_i        ),
  .operator_i         ( alu_operator_i  ),
```

In particular, the clock, enable, and operator signals are connected.

We need to add an instantiation of the riscv_str_ops module. We can follow a similar pattern to the other execution units, and instantiate it by adding:

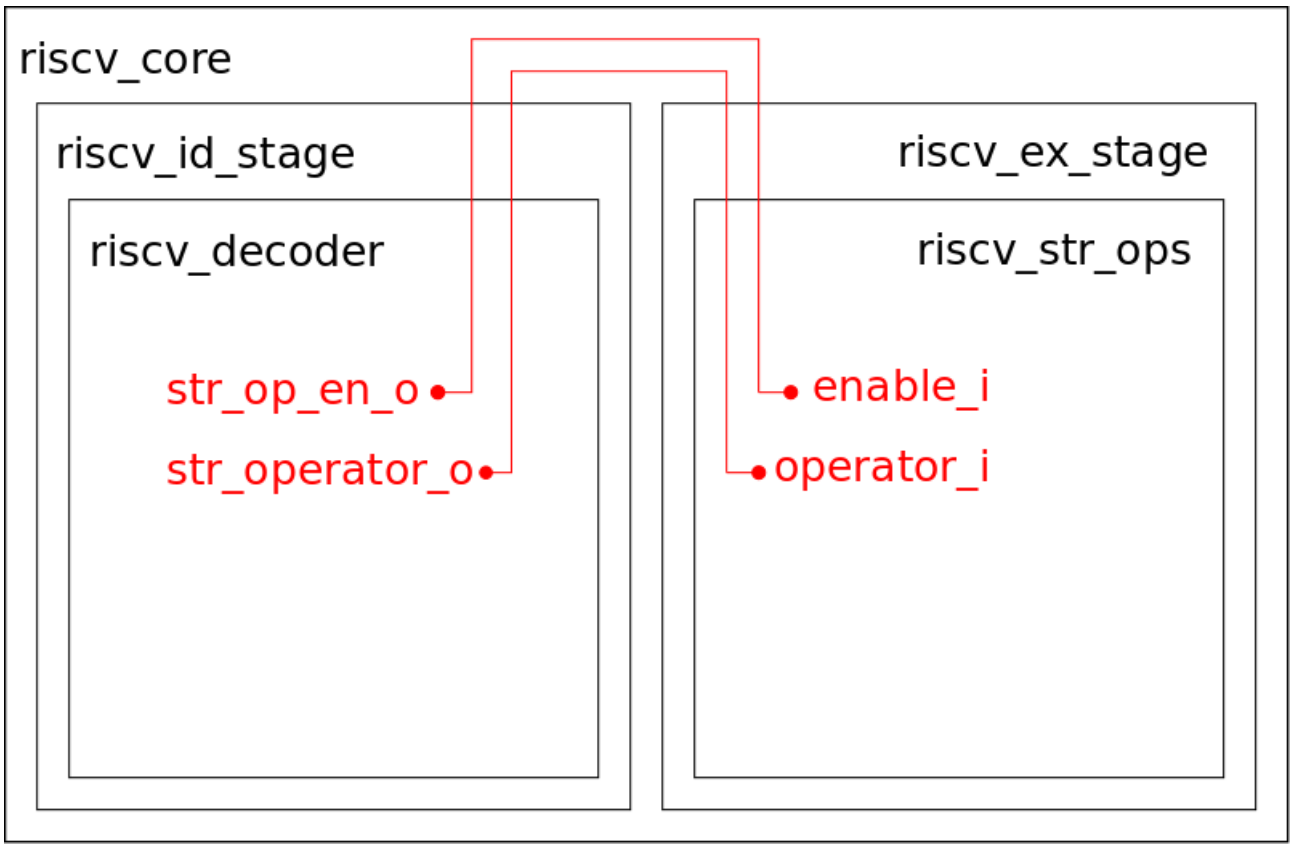
```
riscv_str_ops riscv_str_ops_i
(
 .clk                ( clk             ),
 .enable_i           ( str_op_en_i     ),
 .operator_i         ( str_operator_i  )
);
```

The enable and operator signals are inputs that we need to add to the riscv_ex_stage module (you may wish to add them just before the comment “ALU signals from ID stage”.

```
input logic         str_op_en_i,
input logic [STR_OP_WIDTH-1:0] str_operator_i,
```

Connecting control signals from the decoder to the execution unit

Conceptually, the structure of the relevant modules and signals is:



Module boundaries are shown in black, and signals are shown in red – note how the boundaries of several modules must be traversed by signals in order to connect them together in the decoder and execution unit.

To understand how this is implemented, trace through the Verilog source files how signals such as `alu_en_o` and `alu_operator_o` from the decoder are eventually connected to `enable_i` and `operator_i` in the `riscv_alu` module.

You will need to add similar connections for the string operation enable and operator signals, following the exact same structure and pattern. This is a mechanical process, but if you find you are stuck it may be helpful to refer to the worked solution at this point.

Editing the Makefile to build the execution unit

The Makefile in the verilator-model directory contains a list of Verilog sources to be built by Verilator – you will need to add the name of your new execution unit's source file (e.g. riscv_str_ops.sv) to this list. This may need adding to the list fairly early in the list (e.g. just before riscv_alu.sv) so that Verilator can find its declaration in the file in which it is instantiated.

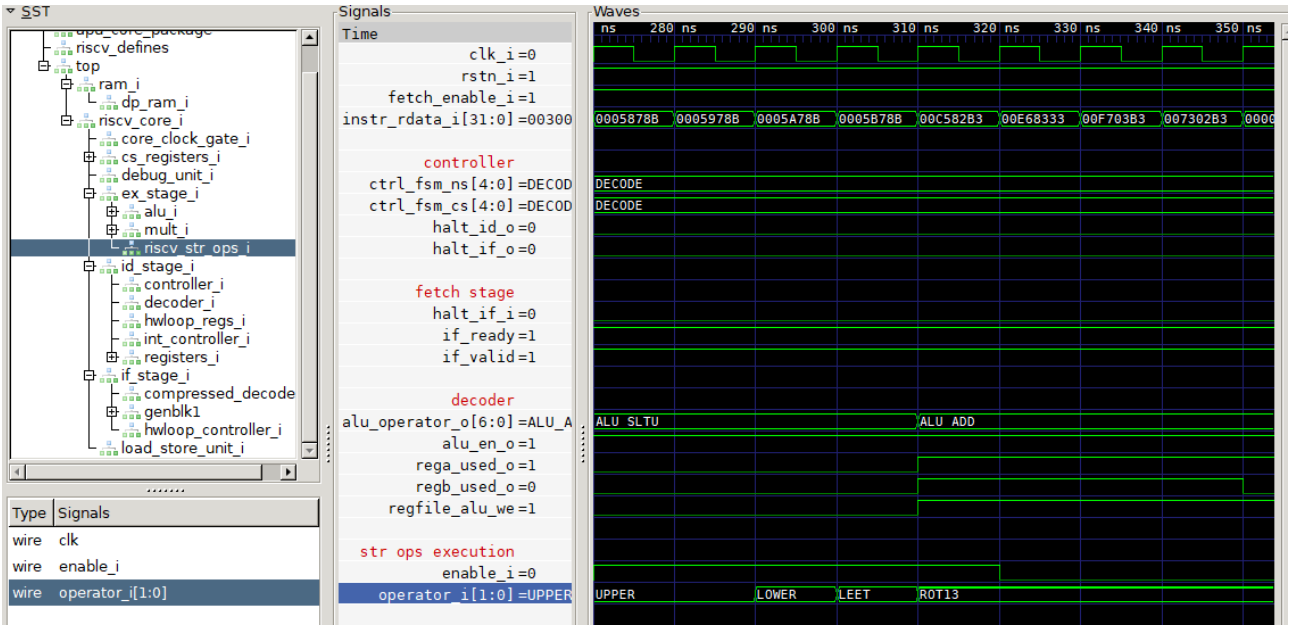
Building and running the testbench

Run make to build the modified model and then execute the testbench. You should see output like:

```
About to halt and set traps on exceptions
About to resume
Cycling clock to run for a few instructions
                275: Exec Upper instruction
                285: Exec Upper instruction
                295: Exec Lower instruction
                305: Exec Leet speak instruction
                315: Exec Rot13 instruction
Halting
```

This demonstrates that the execution unit is successfully being controlled by the decoding of string operations.

You can also view the signals controlling the execution unit with GTKWave – for example, addition of the enable and operator signals from the unit should look like (with an appropriate GTKWave filter file added for the operator signal):



Exercise 8 – Execution unit inputs and outputs

Now that we have the string operation execution unit under our control, the next step is to feed it an input operand that it can work with, and collect the result from it for writing back to the register file. This requires the following changes:

- Add an operand input and a result output to the string operation execution unit.
- Output the input operand as the result in the execution unit – this is a placeholder until we implement actual instruction operations in the next exercise.
- Connect the operand and result signals through relevant modules and set the string operations operand from the register file.

- Enable a register file write and set the register file write data from the string operations result when a string operation is decoded.
- Update our example code to exercise the use of operand and result.
- Use GTKWave to verify correct operation of the operand and result.

Adding an input and output to the execution unit

The riscv_str_ops module needs an additional 32-bit input for its input operand, and an additional 32-bit output. We can modify the module interface like so:

```
module riscv_str_ops
(
  input logic                     clk,
  input logic                     enable_i,
  input logic [STR_OP_WIDTH-1:0] operator_i,
  input logic [31:0]              operand_i,

  output logic [31:0]             result_o
);
```

Since we aren't implementing the semantics of the instruction right now, for testing purposes we will just assign the input right back to the output:

```
always_comb begin
  result_o = enable_i ? operand_i : 32'b0;
end
```

These are the the only changes needed for the execution unit for now – the remainder is for control signals in other parts of the core.

Connecting result and operand signals

In the EX stage (riscv_ex_stage.sv), change the instantiation of the riscv_str_op module to accommodate the new input and output:

```
riscv_str_ops riscv_str_ops_i
(
 .clk                ( clk             ),
 .enable_i           ( str_op_en_i     ),
 .operator_i         ( str_operator_i  ),
 .operand_i          ( str_operand_i   ),
 .result_o           ( str_op_result   )
);
```

You will also need to add a declaration of `str_operand_i` to the `riscv_ex_stage` module interface, and declare `str_op_result` within the `riscv_ex_stage module`.

The operand is assigned in the ID stage. In RI5CY, rs1 is extracted into alu_operand_a by the decoder, so we can assign this at the same time as setting the operator in riscv_id_stage.sv – modify the code added in an earlier exercise so that it sets the operand:

```
if (str_op_en) begin
  str_operator_ex_o         <= str_operator;
  str_operand_ex_o          <= alu_operand_a;
end
```

You will also need to add `str_operand_ex_o` to the ID stage's module interface, and assign it a default value in the same way that the string operation enable signal and operator are assigned default values.

The operand signal then needs connecting between the ID and EX stages in `riscv_core.sv`.

Enabling register file writes for string operations

It is the decoder that controls whether an instruction writes back to a register file. In the case block for string operation decoding, set:

```
regfile_alu_we = 1'b1;
```

for each instruction case in the string operations case block. This signal instructs the later pipeline stages that it should write data from `regfile_alu_wdata_fw_o` back into the register file. To ensure that this contains the result from our execution unit, we need to set it appropriately in the EX stage. Around line 210 or `riscv_ex_stage.sv`, we see there are various sources for this value:

```
if (alu_en_i)
  regfile_alu_wdata_fw_o = alu_result;
if (mult_en_i)
  regfile_alu_wdata_fw_o = mult_result;
if (csr_access_i)
  regfile_alu_wdata_fw_o = csr_rdata_i;
```

We can set the result by adding one additional conditional assignment:

```
if (str_op_en_i)
  regfile_alu_wdata_fw_o = str_op_result;
```

Updating the example code

All required changes to the core are now made. In order to verify correct operation, it will be helpful to add some instructions to the example code (examples.s) that uses some different input values with different input and output registers. Adding something like the following code for string operations is sufficient:

```
li a0, 0
li a1, 1
li a2, 2
li a3, 3
```

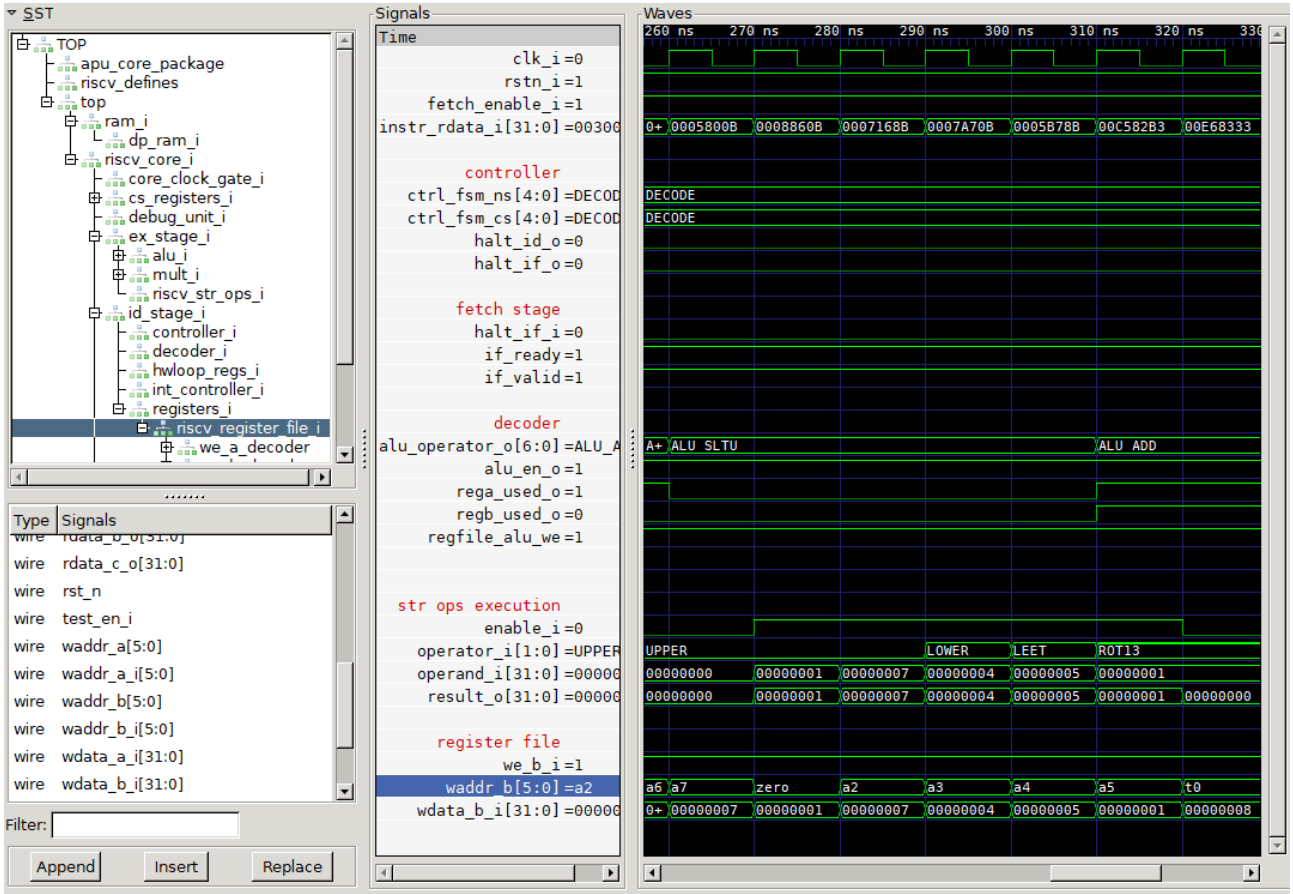
```
li a4, 4
li a5, 5
li a6, 6
li a7, 7
.insn i 0x0b, 0, a0, a1, 0 # upper a0, a1
.insn i 0x0b, 0, a2, a7, 0 # upper a2, a7
.insn i 0x0b, 1, a3, a4, 0 # lower a3, a4
.insn i 0x0b, 2, a4, a5, 0 # leet a4, a5
.insn i 0x0b, 3, a5, a1, 0 # rot13 a5, a1
```

After making the modifications, run `make` to rebuild this example code and run the testbench again.

Verifying correct operation with GTKWave

Reload the waveform in GTKWave. If you add the signals for the B port of the register file (see image below) then you should be able to observe that on the same cycle that each string operation instruction executes, the value of its operand is written back to the register file.

It may be helpful to create a GTKWave filter that displays the ABI register names instead of the register numbers – one can be found in the worked solutions for this exercise, if you would prefer not to write it.



Exercise 9 – Implementing single-cycle instruction semantics

In this exercise we will change the execution unit so that instead of performing a no-op, it implements the semantics of an instruction. Whilst all four instructions can be implemented, we will work through only implementing the upper instruction in this exercise.

The upper instruction should, for each 8-bit byte in its operand, either:

- If the byte is between 97 and 122 inclusive, subtract 32 from it and place in the result.
- Otherwise, place the byte in the result as-is.

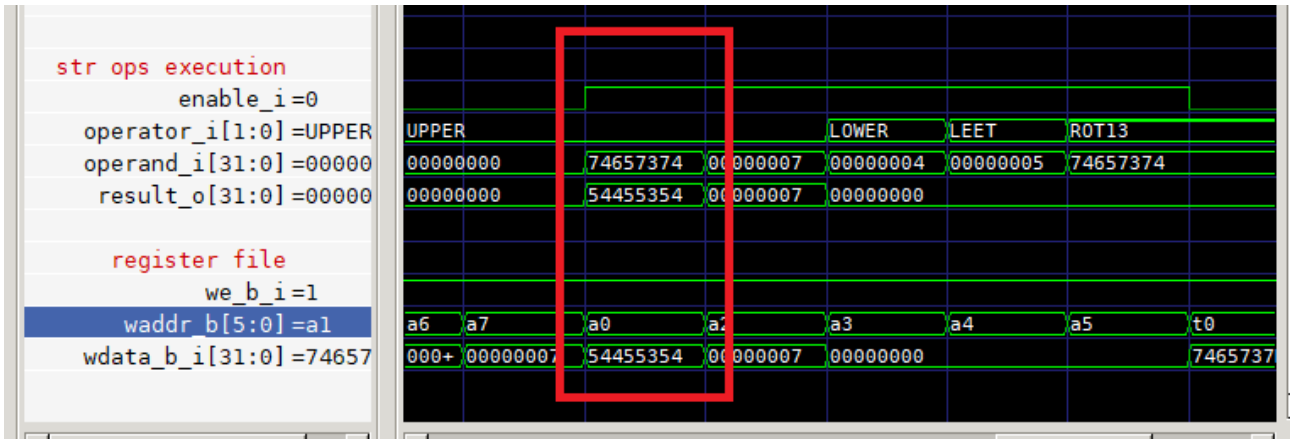
Replace the direct assignment of the operand to the result in `riscv_str_ops.sv` with some Verilog code that implements the above operations.

Testing

Modify the example assembly code so that a 4-character ASCII string is loaded into a register and used as the input for an upper instruction. For example:

```
li a1, 0x74657374           # a1 = "TEST"
.insn i 0x0b, 0, a0, a1, 0 # upper a0, a1
```

Rebuild the examples and re-run the test bench. When you reload the waveform, you should see the transformation of the input:



Here we see that the string "test" (0x74657374) is transformed into "TEST" (0x54455354) when the `upper a0, a1` instruction is executed.

Exercise 10 – Implementing a multi-cycle instruction

This exercise is based around implementing the LEET instruction in a way that uses multiple cycles. There are cases in which meeting a particular clock speed goal requires splitting the

execution of an instruction from a single cycle into multiple cycles that each complete a part of the computation. It is not usually required for fairly simple operations, but more complex ones with deep combinatorial paths, and/or a lot of fan-out (e.g. a fused multiply-add, or any arithmetic that uses a lot of bits of input operands) may require it.

Whilst multiple cycles probably wouldn't be required for the implementation of a simpler instruction like LEET (it could be implemented to complete in a single cycle), we split it into multiple cycles here for the purpose of illustrating how multiple-cycle instructions fit in to RI5CY.

This requires:

- Add ready inputs and outputs and a reset input to the execution unit.
- Add a state machine and some state for executing the instruction.
- Initialise the state machine on reset.
- Implement semantics of the leet instruction using a register to store intermediate values.
- Mux results from different instructions in the execution unit (required now we have more than one instruction).
- Connect reset and ready signals to the execution unit in the EX stage.
- Modify the test code to exercise the instruction.
- Examine the operation in GTKWave to verify correctness.

Adding signals to the execution unit

We need to add a reset input, and ready inputs and outputs, so the module interface becomes:

```
module riscv_str_ops
(
  input logic                     clk,
  input logic                     rst_n,
  input logic                     enable_i,
  input logic [STR_OP_WIDTH-1:0] operator_i,
  input logic [31:0]              operand_i,

  output logic [31:0]             result_o,

  output logic [31:0]             ready_o,
  input logic                     ex_ready_i
);
```

Implementing a state machine for the leet instruction

The state machine is used to track progressing of the execution of a leet instruction. This machine generally remains in the idle state until a leet instruction is executing. When an instruction is decoded, it should move through the following states on subsequent cycles:

- STEP0: translates “e” and “E” into “3”.
- STEP1: translates “s” and “S” into “5”.
- STEP2: translates “l” and “L” into “1”.
- FINISH: signals that the output is ready, as long as there isn’t something later stalling the pipeline.

To implement this state machine, we need to add enums for the current and next state:

```
enum logic [2:0] {IDLE, STEP0, STEP1, STEP2, FINISH} leet_CS, leet_NS;
```

Now we need to add logic for determining the next state based on the current state and other conditions. The requirements are:

- In the IDLE state, we move to STEP0 if the unit is enabled with the leet operator.
- In STEP0, the next state is STEP1
- In STEP1, the next state is STEP2
- In STEP2, the next state is FINISH
- In FINISH, the next state is IDLE, as long as we are not stalled by a later stage (otherwise the result would be lost by moving immediately back to the IDLE state).
- The unit signals that it is ready in the IDLE and FINISH states.
- The unit signals that it is active in all states except IDLE.

We can implement these requirements with the following:

```
logic leet_active;
logic leet_ready;

always_comb begin
  leet_ready = 1'b0;
  leet_NS = leet_CS;
  leet_active = 1'b1;

  case (leet_CS)
    IDLE: begin
      leet_active = 1'b0;
      leet_ready = 1'b1;
      if (operator_i == STR_OP_LEET && enable_i) begin
        leet_ready = 1'b0;
        leet_NS = STEP0;
      end
    end
    STEP0: begin
      leet_NS = STEP1;
    end
    STEP1: begin
```

```
          leet_NS = STEP2;
        end
        STEP2: begin
          leet_NS = FINISH;
        end
        FINISH: begin
          leet_ready = 1'b1;
          if (ex_ready_i)
            leet_NS = IDLE;
        end
      endcase
    end
```

We should also set our ready output based on the state – this can be assigned directly:

```
    assign ready_o = leet_ready;
```

Advancing state and initialising at reset

At each cycle we need to advance the state of the state machine, so that the next state becomes the current state of the next cycle. To ensure that the leet state machine is in a well-defined condition at reset, we also set its state when reset is applied:

```
always_ff @(posedge clk, negedge rst_n)
begin
  if (~rst_n)
    leet_CS <= IDLE;
  else
    leet_CS <= leet_NS;
end
```

Implementing semantics

We need to add some code that transforms the value being operated on during STEP0, STEP1, and STEP2. We can add a register to hold the intermediate value:

```
logic [31:0] leet_intermediate;
```

We will also modify the always block included in the last step to implement the transformation.

- In the IDLE state, if the execution unit and leet instruction are enabled, we need to move the operand into the intermediate so it is available on the next cycle.
- In STEP0 / STEP1 / STEP2 we need to store the transformed value on the next cycle in the intermediate.

We will add a new case based on the current state. The following is slightly abridged – the comments indicate the remainder of the implementation.

```
  case (leet_CS)
    IDLE: begin
```

```
    if (operator_i == STR_OP_LEET && enable_i)
      leet_intermediate <= operand_i;
  end
  STEP0: begin
    // Turn "e" and "E" into "3"
    // First Character
    char = leet_intermediate[7:0];
    if ((char == 69 || char == 101))
      char = 51;
    leet_intermediate[7:0] <= char;
    // Transform remaining 3 bytes (15:8, 23:16, 31:24)
  end
  STEP1: begin
    // Similar to STEP0 but which char values for “s” and “S” to “5”
  end
  STEP2: begin
    // Similar to STEP0 but which char values for “s” and “S” to “5”
  end
```

Muxing results from the string op unit

Now we have two different instructions that can potentially produce a result, we need to select from the appropriate result to generate the result output from the string op execution unit. In the always_comb block of the string op unit, add:

```
// Mux results
if (enable_i) begin
  // Dummy value for unimplemented instructions
  result_o = 32'hDEADBEEF;

  if (operator_i == STR_OP_UPPER)
    result_o = result;
  if (operator_i == STR_OP_LEET && leet_CS == FINISH)
    result_o = leet_intermediate;
end
```

to replace the old assignment of the result, which may have looked something like:

```
result_o = enable_i ? result : 32'b0;
```

Connecting up ready signals

In the EX stage we need to connect up the new signals that we have added to our string operation execution unit.

We need one additional declaration:

```
  logic           str_op_ready;
```

Then we can modify the instantiation to:

```
riscv_str_ops riscv_str_ops_i
(
 .clk                ( clk             ),
 .rst_n              ( rst_n           ),
 .enable_i           ( str_op_en_i     ),
 .operator_i         ( str_operator_i  ),
 .operand_i          ( str_operand_i   ),
 .result_o           ( str_op_result   ),
 .ready_o            ( str_op_ready    ),
 .ex_ready_i         ( ex_ready_o      )
);
```

Then we need to edit the assignment of the ready and valid signals used to stall the pipeline – this connects up our string operation unit to the pipeline control in the same way that other execution units are connected. If you look around line 509 of the EX stage, you will see:

```
assign ex_ready_o = (~apu_stall & alu_ready & mult_ready & lsu_ready_ex_i
                     & wb_ready_i & ~wb_contention) | (branch_in_ex_i);
assign ex_valid_o = (apu_valid | alu_en_i | mult_en_i | csr_access_i | lsu_en_i)
                     & (alu_ready & mult_ready & lsu_ready_ex_i & wb_ready_i);
```

You will need to modify these two expressions, so that the ready signal from the string operation unit is used in assigning `ex_ready_o`, and the string operation enable and ready signals are used in the assignment of `ex_valid_o`.

Modifying test code to validate correct execution

It will be helpful to add a couple of different ASCII strings to registers to test the leet instruction. Adding something like:

```
li a5, 0x74657374 # "test"
li a6, 0x6c656574 # "LEET"
.insn i 0x0b, 2, a4, a5, 0 # leet a4, a5
.insn i 0x0b, 2, a5, a6, 0 # leet a5, a6
```

to `examples.s` provides a couple of suitable test inputs. Since the multi-cycle execution of these instructions adds to the total number of cycles, it may be necessary to modify the number of cycles the test runs for in `testbench.cpp`:

```
// Number of cycles to run for
const uint32_t CYCLES_TO_RUN_FOR = 30;
```

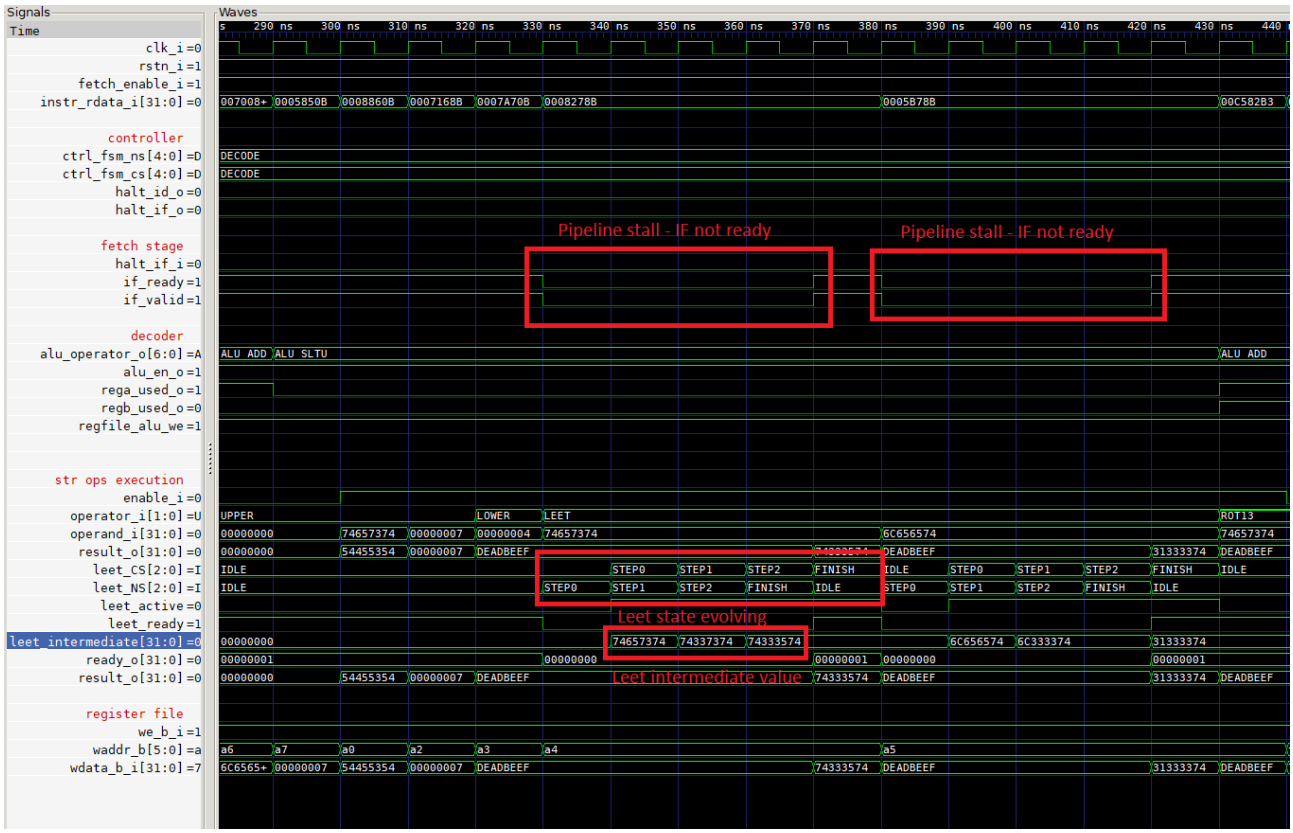
Rebuild everything with make and run the testbench again.

Examining operation

In GTKWave, add the new signals that you have created to the view. You should be able to observe the following occur when the LEET instructions are executed:

- The state machine moving through the states IDLE, STEP0, STEP1, STEP2, and FINISH. `leet_CS` should be one cycle behind `leet_NS`.
- The pipeline should be stalled when the leet instruction is executing – this can be observed in the values of `if_ready` and `if_valid` in the fetch stage.
- The value of the intermediate in `leet_intermediate` should evolve with each clock cycle through STEP0 to FINISH (depending on the value of the input).

The example signals in the screen capture below illustrate this:



Finish

Congratulations on reaching the end of the workshop!

It would be much appreciated if you could direct and feedback, comments, corrections, or questions to Graham Markall, either on the Slack Channel or by email – this will support improvement of this tutorial for future participants.