Designing a RISC-V Integer ALU

Computer Architecture and Design – Practical 1

October 2019

1 Objectives

The aim of this practical is to give you an opportunity to explore combinatorial logic design by implementing the integer ALU of a baseline 32-bit RISC-V processor. During this practical exercise you will learn how implement a combinational logic block according to a given specification, verify the design, and evaluate it's speed and logical complexity. This practical requires that you write a report of your activities and hand it in with your lab sheet. For more details on how this practical is assessed, please refer to the marking scheme document that is available on the LEARN page where you obtained this document.

2 Introduction

For this practical a packaged Vivado project, containing an empty ALU module, has been created for you to use. This contains a top-level testbench that instantiates the empty ALU. Your task is to implement all functions expected of a baseline configuration of RISC-V processor, *i.e.* it should support all of the 32-bit integer operations. This handout contains a detailed functional specification for the ALU, which you should follow. The top-level testbench contains a reference model for the ALU, allowing it to check that the outputs from your ALU are correct for any given set of input stimuli. When you simulate your ALU the testbench generates a pseudo-random sequence of ALU operations, presents them to both your ALU and the reference model, and reports any differences. By running a large number of inputs through your design you can gain confidence that the design works as intended. When a failure in your design is reported by the testbench you can use the waveform viewer in Vivado to examine the internal signals within your design, debug the failing operation, and revise your design accordingly.

3 ALU Functional Specification

The ALU is to be implemented as a purely combinational module, *i.e.* without any internal flip-flops. It therefore does not require a clock or reset input. The module header, including the input and output ports, is shown below:

```
module alu(
  input        [3:0]    exe_alu_opc_r,  // ALU sub-opcode (defined in params.v)
  input                 exe_sel_pc_r,   // select between PC (1) or Xreg rs1 (0)
  input        [31:0]   exe_pc_r,       // value of PC at the EXE stage
  input        [31:0]   exe_reg1_r,     // value of Xreg rs1 at the EXE stage
  input        [31:0]   exe_src2_r,     // second source register at EXE
  output  reg  [31:0]   alu_result      // ALU result output
);
```

This module has five inputs and one output. The input names are all prefixed with `exe_` as this is the name of the pipeline stage that provides those input signals. The output signal is prefixed with `alu_` as this is the name of the module that drives that signal. You are free to change the output declaration from `reg` to `wire`, depending on how you choose to implement that signal. If the output is driven from within a process then leave it defined as a `reg`, but if you drive this using a continuous `assign` statement, then remove the `reg` keyword to convert this output signal to a `wire` declaration.

The operation to be performed is selected by the `exe_alu_opc_r`. This is a 4-bit value capable of specifying up to 16 distinct operators. The assignment of operators to these 4-bit encodings is defined by a set of `parameter` declarations contained in an included file called `params.v`. The relevant lines from that file are shown below:

```
parameter [3:0]  ALU_OPC_ADD     = 4'b0000; // addition
parameter [3:0]  ALU_OPC_SUB     = 4'b1000; // subtraction
parameter [3:0]  ALU_OPC_SLL     = 4'b0001; // shift-left logical
parameter [3:0]  ALU_OPC_SLT     = 4'b0010; // set if less than
parameter [3:0]  ALU_OPC_SLTU    = 4'b0011; // set if less than, unsigned
parameter [3:0]  ALU_OPC_XOR     = 4'b0100; // bit-wise exclusive OR
parameter [3:0]  ALU_OPC_SRL     = 4'b0101; // shift-right logical
parameter [3:0]  ALU_OPC_SRA     = 4'b1101; // shift-right arithmetic
parameter [3:0]  ALU_OPC_OR      = 4'b0110; // bit-wise OR
parameter [3:0]  ALU_OPC_AND     = 4'b0111; // bit-wise AND
```

As you would expect, the ALU takes two 32-bit inputs representing the two source operands of each dyadic operator. These are `exe_reg1_r` and `exe_src2_r`. The `exe_reg1_r` operand is a source register, whereas `exe_src2_r` may be a source register or a literal value. This is not relevant, however, within the ALU where we simply operate on 32-bit values regardless of whether they are register values or literals.

You will notice that there are two other inputs, `exe_pc_r` and `exe_sel_pc_r`. These are here because the ALU evaluates not only the operator for a set of dyadic instructions but also computes the target address of relative branch instructions. This requires the ALU to add together the PC and a literal offset value. When computing the branch target address, the PC value is presented on the `exe_pc_r` input, and the literal value is presented on `exe_src2_r` input. When `exe_sel_pc_r` input signal is true, an addition performed by the ALU should obtain its first operand from `exe_pc_r`. When `exe_sel_pc_r` input signal is false an addition should take the first source operand from `exe_reg1_r`. Operators other than addition always use `exe_reg1_r` and `exe_src2_r` as their first and second operands respectively.

4 Getting Started

A zipped project file `prac1.zip` can be found on the course website, on the Practicals page, just below the information about Practical 1. The zipped project contains all of the Verilog source files, libraries, and build scripts needed to create a Vivado project that is capable of simulating and synthesising your design. For this practical there is no requirement to run your desing on the FPGA, as all testing can be done in simulation. Instead of going through the tedious process of creating the project manually, there is a TCL script called `create_project.tcl` that you can run once after unzipping the project. This is done as follows:

1. Take a copy of `prac1.zip` and unzip it in a suitable location.

2. You will find the unzipped project has a root directory called `prac1`, within which various subdirectories exist. `cd` to the `prac1/proj` directory.

3. As always, start by sourcing the Vivado settings script
   `source /disk/scratch/Xilinx/Vivado/2018.3/settings64.sh`
4. Launch Vivado from the command line by typing `vivado`.
5. At the bottom of the Vivado window select the **TCL Console** tab, and in that console type:
   `source create_project.tcl`

This will take a few moments to create the project, after which you will see the project manager view of the project. If you open up the Design Sources window you will see the `alu.v` file. Open that file and you will see an empty module containing the module header as shown above. You will implement your ALU within this file.

5 Part (a) – Designing and Simulating an ALU module

To quickly check that your project has been created successfully, try running a simulation of the empty ALU. Not surprisingly this will fail immediately! The testbench provides diagnostic information in the **TCL Console** pane. This is in the lower part of the Vivado window; you can double-click on the tab to enlarge it (and to return it to its original size). At this point you should create your ALU implementation and iterate through test/debug cycles until simulation runs to completion without reporting any errors.

5.1 Implementing your ALU

You have complete flexibility in the approach you take to implement your ALU. You can define and instantiate sub-modules for related groups of instructions if you wish, or you may implement it as a single module. There is no single best solution, but you should aim for a design that is concise and readable, as well as correct and efficient. Factors to consider are:

- Are the ADD and SUB operations implemented by a common adder, or will your design require two carry-propagate adders?
- Do your left-shift and right-shift implementations share the same shifter, or are they each implemented in completely independent blocks of combinational logic?
- What result does your ALU return if the operator is not one of the values defined in `params.v`?

5.2 Developing and Testing your ALU

The operations implemented by this ALU are partitioned into four groups, each containing a small number of related operations. These are:

1. Logical operations, comprising bitwise AND, OR, XOR.
2. Binary addition and subtraction operations.
3. Comparison operations, comprising set-less-than (SLT) and unsigned set-less-than (SLTU).
4. Shift operations, comprising shift-left-logical (SLL), shift-right-logical (SRL), and shift-right-arithmetic (SRA).

You are recommended to develop and test each group before moving on to the next group (you can tackle them in any order). The testbench provides a mechanism by which you can select the groups that you want to include in a simulation run. Therefore, if you initially implement just the logical operations, you can enable testing of just the logical operations and disable all other groups. To do this, open up the testbench source file `tb.v` and adjust the `localparam` definitions on lines 50–51. After any modifications to `tb.v` don't forget to save it before you re-run the simulation.

The testbench runs one test per clock cycle, selected pseudo-randomly from the set of enabled instruction groups. You can set the number of test cycles that are performed during the run by adjusting the `NUM_CYCLES` parameter in `tb.v`. **Your design must pass 10,000 simulation cycles with all groups enabled to be considered complete.**

In the next practical you will integrate your ALU within a full RISC-V processor and run it on the FPGA, so having a good solution for this practical will help you to achieve an efficient solution for the next practical. However, if you are unable to achieve an efficient working solution for this practical you will be supplied with a library component for the ALU so that you are not disadvantaged.

At this point call over a lab demonstrator and get Part (a) signed off on your laboratory sheet.

6 Part (b) – Synthesising the ALU Module

Once you have a functioning ALU module that passes the all tests for at least one instruction group, you should synthesise your design. Synthesis is the process whereby Vivado maps your Verilog down to individual gates connected by wires. The result is called a **netlist**. In general there may be many possible netlists for a given Verilog module, so the synthesis software must be constrained so that it can aim for a particular speed or area goal, and so that it can know when its internal optimization processes have reached that goal. Sometimes constraints are too optimistic and cannot be met. In this case the Vivado log will indicate a synthesis error, and will provide reports that you can examine to understand what constraints could not be met.

For your ALU design the only constraint of interest is the maximum allowable timing delay from any input to the `alu_result` output. In order to define a maximum delay through this combinational module we must:

1. Create a virtual clock, with a defined period T_{ck}.
2. Define the maximum and minimum delays from the start of the clock to the arrival of input signals
3. Define the maximum and minimum delays from the availability of output signals to the end of the clock period

In this case there are no delays for items (2) and (3) so we set those delays to 0. This means our ALU is constrained so that the maximum delay through any logic path within the ALU is no longer than T_{ck}. Initially you will not know what a reasonable value of T_{ck} might be, so you should experiment a little. So let's start by setting an initial value of 20ns for T_{ck}.

1. Open up the timing constraints file provided with the project (`timing.xdc`)
2. Set the period of the virtual clock `clk` to 20ns
3. Save the modified `timing.xdc` file
4. Run the Synthesis
5. Open up the Timing Summary Report and the Utilization Report

After synthesis you should carefully check the TCL Console output for any sign of synthesis errors, and fix any errors before proceeding. In particular, look out for combinational loops and inferred latches – these are common design bugs that show up during synthesis.

Record your results at this stage for your writeup: from the Timing Report record the length of the most critical timing path and the timing slack; and from the Utilization Report record the number of LUTs that are used to implement your design. In your utilization report window, click on *Primitives* in the left-hand pane. This will bring up a table showing the number of each type of primitive cell that was used within the FPGA for your ALU. The *Summary* section shows all types of cells, including block memories (BRAMs) and arithmetic units (DSPs).

If the timing report indicated that your design met the timing constraints then the constraints could be tighter. At this point you should reduce the virtual clock period and re-synthesize. This process can be repeated as many times as necessary, until you are confident that you have found the tightest timing constraint that can be met. **Record your timing and design area measurements for each iteration**.

At this point call over a lab demonstrator and get Part (b) signed off on your laboratory sheet. Show the demonstrator your `timing.xdc` file and the resulting timing summary report.