Instruction Driven ALU

October 1, 2021

1 Instructions

This document outlines your Verilog take-home test instructions. There are three Sections in this document; **Instructions**, **Design of a small ALU**, and **Your Task**. Make sure to read them all!

1.1 Files

When you unpack your zip-file you should find the files listed in Table 1 in a folder called alu/

File name	Description
alu.sv	This is the file you need to modify
alu_tb.sv	Top level for the simulation
Makefile	Used to build and simulate with Icarus Verilog
test.asm	Assembly for the test program
test.hex	Assembled version of the test program
test_reference.output	Expected output from the test program
secret.hex	A longer, randomly generated program, can be used for testing purposes
spec.pdf	This file

Table 1: Files

1.2 Running

We recommend using Icarus Verilog and GTKwave to compile, simulate, and debug your design. Most distributions based on Debian will have these tools ready be installed from their package managers. For example on a fresh install of Ubuntu 21.04[1] you can get the necessary tools by running:

```
sudo apt install iverilog gtkwave make
```

When these tools are installed, you can navigate to the just extracted folder and run the simulation of the test program by executing **make**. To run the secret program just execute **make secret**. Note that the simulation will run forever without any output before you implement your solution.

When running with the test program i.e. **make test**, your output will be compared against test_reference.output. If there are mismatches the return code will be non zero and you will see something like what is shown in Listing 1. **Note that test.hex is by no means exhaustive in testing all aspects of your implementation.**

Running the simulation will capture the waves in either **test.fst** or **secret.fst**. With GTKwave installed you can view the waves with **gtkwave test.fst** or **gtkwave secret.fst**

Note, **iverilog** has support for most of Verilog and limited support for SystemVerilog. See Table 2 for known limitations and suggested work-arounds.

[1] If you are currently not on Linux or your distribution does not have these tools, consider installing Ubuntu 21.04 on a local VM using Virtualbox.

Listing 1: Example of simulation with wrong output

```
...
diff -u test.output test_reference.output
--- test.output 2021-09-13 16:56:41.391309086 -0400
+++ test_reference.output       2021-09-13 16:56:01.440457148 -0400
@@ -1,7 +1,7 @@
 0x048 (H)
 0x052 (R)
 0x054 (T)
-0x000 ()
+0xfff ()
 0xfff ()
 0x000 ()
 0xfc0 ()
make: *** [Makefile:11: test] Error 1
...
```

SV construct	Work-around
`always_comb`	`always@*` or `assign`
`always_ff@(posedge clk)`	`always@(posedge clk)`
unpacked array in `struct`	break up into `item_0`, `item_1`, etc.
`assign a = some_type_t'(b)`	casts are not needed, just do `assign a = b`
`s[N:0] = o[N-1:0] + p[N-1:0]`	`s[N:0] = {1'b0, o[N-1:0]} + {1'b0, p[N-1:0]}`

Table 2: Known **iverilog** SystemVerilog limitations

You can choose to complete the assignment using any other tools you are familiar with! However, we will only assist you with troubleshooting Icarus Verilog and/or GTKwave issues.

If you cannot get Icarus Verilog and/or GTKwave to work and/or you have no other tools at your disposal, please email `hrthardwareintern@hudson-trading.com`**.**

1.3 Submitting

Please submit your work by sending an email to `hrthardwareintern@hudson-trading.com`. **Put your name in the subject line** and attach **alu.sv**.

2 Design of a small ALU

Your task is to design a simple instruction driven ALU following the specifications in this document. **The implementation should be written for hardware synthesis.**, i.e. behavioral code is not acceptable. In order to improve the max operating frequency, the ALU should be a two stage pipeline design. Register values are read out and operated on in the first stage of the pipeline. The result from the selected operation is then registered before being written back to the register file in the next clock cycle. A sketch[2] of the ALU system is shown in Figure 1. **The artifacts of the two stage pipeline should be transparent to the programmer.** Meaning that a program ran on an ALU without the extra pipeline stage should behave and give the same output as the two stage design required here. This also means that executing the instructions one by one in e.g. an emulator should have the same result as running the program in the two stage pipeline design.

All clocked logic in the ALU should be triggered on the rising edge of `clk`. The ALU should have four general purpose registers that are 12 bits wide. All operations are performed on 12 bit unsigned integers unless otherwise stated. If the result of an operations exceeds 12 bits, only the 12 least significant bits are stored back to the destination register.

There are two types of instructions Arithmetic and Immediate the instruction formats are specified in Table 3, where Rd is where the result is written, Rx/Ry are where the operand values are taken from, and *immediate* is a constant value

[2]Note this is just a sketch, your hardware design is likely going to need more components then what is shown here.

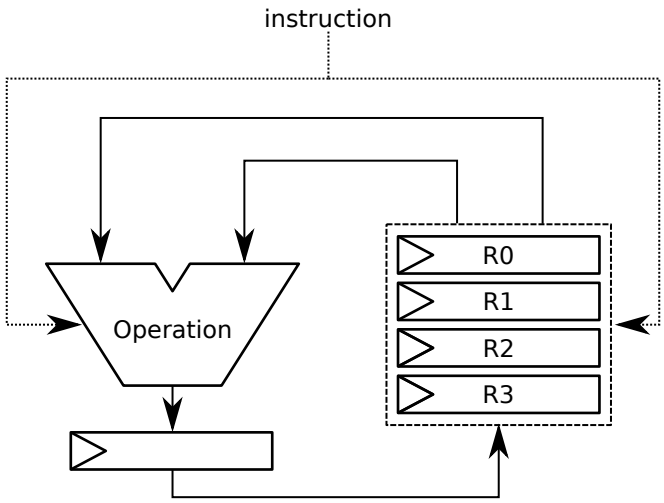

Figure 1: ALU sketch

Type	bits				
	11:8	7:6	5:4	3:2	1:0
Arithmetic	op-code	Rd	Rx	Ry	fill
Immediate	op-code	Rd	*immediate* (bits 5:0)		

Table 3: Instruction format definition

carried with the instruction. The fill bits in the Arithmetic instructions should be ignored by hardware and will be set to 0 in the instruction word. For example $R3 = R1 + R2$ would result in the instruction word `0x7D8` and $R3[5:0] = 0x17$ in the instruction word `0xBD7`. Table 4 list the operations that the ALU should be able to perform. Note that only ADD and ADDC should update the carry, it shall remain unchanged for any other instruction.

Operation	**op-code**	**operation**	**description**
OR	0x0	$Rd = Rx \vee Ry$	bitwise or
XOR	0x1	$Rd = Rx \oplus Ry$	bitwise xor
AND	0x2	$Rd = Rx \wedge Ry$	bitwise and
NOT	0x3	$Rd = \neg Rx$	bitwise not
LSHIFT	0x4	$Rd = Rx \ll 1$	logical left shift of Rx by one bit
RSHIFT	0x5	$Rd = Rx \gg 1$	logical right shift of Rx by one bit
ARSHIFT	0x6	$Rd = Rx \ggg 1$	arithmetic right shift of Rx by one bit
ADD	0x7	$\{c, Rd\} = Rx + Ry$	add and update carry
ADDC	0x8	$\{c, Rd\} = Rx + Ry + c$	add with and update carry
SUB	0x9	$Rd = Rx - Ry$	subtraction
LOADLO	0xA	$Rd[5:0] = immediate$	load lower 6 bits of Rd
LOADHI	0xB	$Rd[11:6] = immediate$	load upper 6 bits of Rd
OUT	0xC	$out\_data = Rx$	Output value of Rx and also assert out_data_valid
HALT	0xD	$out\_data = Rx$	Same actions as for OUT while also asserting $halt$

Table 4: ALU operations

3 Your Task

Implement your code in the skeleton file alu.sv where you see `// Your code here...`. You should not need to modify any other file. **Note that a submission with a single stage pipeline design is not acceptable and will disqualify you from moving forward in the interview process.** Furthermore **The HDL code should be synthesizable, a behavioral solution will also disqualify you from moving forward in the interview process.**

When you think your ALU behaves correctly, submit alu.sv, as instructed earlier. We will run some extensive tests on our end to verify that your implementation is correct.

Also please answer the following questions as comments in alu.sv.

1. There are no NOP (don't change any register values) or CLEAR (set a register to 0) instruction specified, how can we perform these operations with the instructions we have available to us?
2. How did you test your design?

If you have any additional work, related to the design or testing process, you would like to share with us, please create a zip archive, called **extra.zip**, with the extra material and attach the archive to the submission email.