Practical assignment ECA2:
Processor design in Clash

Introduction

Below is the set of homework exercises for ECA-2 regarding a stack processor, to be handed in by groups of two students. We assume you successfully completed the tutorial (tutorial.pdf).

deadline for handing in your solutions: 18 January 2026, 23:59, on canvas. **Deliver as group**, submission after the deadline means -5pts subtracted for each day.

Remarks on delivery

- We have provided 8 files, in the clashi directory

  `Exc1.hs`

  `Exc2_1_Stack.hs`

  `Exc2_2_Stack.hs`

  `Exc3_1_Processor.hs`

  `Exc3_2_Processor.hs`

  `Exc4_Fetch.hs`

  `Exc5_System.hs`

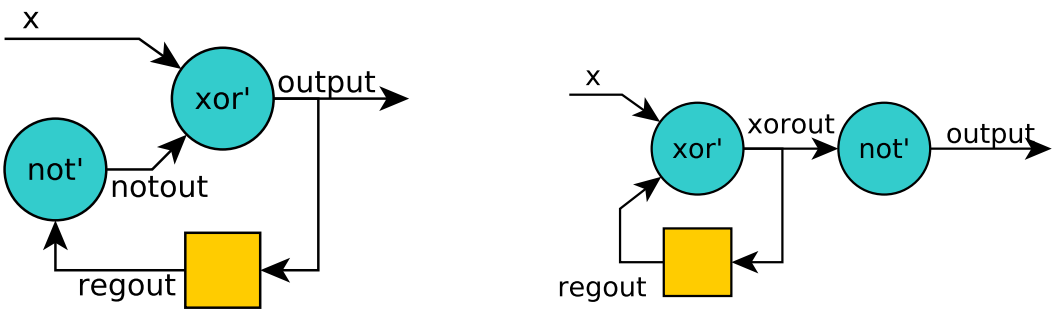
  `Exc6_Processor.hs`

  `Exc7_Fetch.hs`

  `Exc8_System.hs`

- Add your names and student numbers at the top of every file.
- Add your names and student numbers to the front page of your report.
- For all assignments you are asked to deliver Clash code. Please include your Clash code for that specific assignment in a text box in your report.
- There is no need to deliver Verilog/VHDL code.
- Use vector functions, reference documentation of Vectors:
  `hackage.haskell.org/package/clash-prelude/docs/Clash-Sized-Vector.html`

- In some assignments you are asked to describe/show the combinational path, here you need to draw this path in a figure.
- You can score 10 points in this assignment.
- Deliver your report (pdf) separately and all the Clash files on canvas.
- The grading also depends on the style and readability of the code.
- Keep your answers short and concise.
- **If you are asked to show only the function and/or test result, then no further comments are required.**



(a) *exampleSystem* (b) Exc1 system

Figure 1: *not'* and *xor'*

Exc1

Given are the *xor'* and *not'* functions, which operate on a single Unsigned 1 value. As an example, the system from Figure 1a is given in the *exampleSystem* function.

Assignment 1 (0.5 pts):

Create in `Exc1.hs` a system using `xor'` and `not'`, as shown in Figure 1b. Do not use the 'mealy' function to achieve your system, but compose the signals with <$> and <*> (fmap and applicative). Show your `system` function definition and the result of `testSystem1` and `testSystem2` in your report.

Exc2 Stack

Figure 2 shows a simple stack controller that can push values to a blockRAM. The types are given on the edges. The `blockRAMblock` function and its type are given. The yellow block is the state of stack controller, for now it only contains an Address. In your system composition use the `unbundle` function to extract the two signals the blockRAM needs from the signals your stack controller outputs. The output of the system is the output of the blockRAM.

Assignment 2.1 (1 pts):

Build in `Exc2_1_Stack.hs` a simple stack controller that can push values to a blockRAM. Use `mealy` to turn your controller into a block that takes a signal of Instr, and outputs a signal that

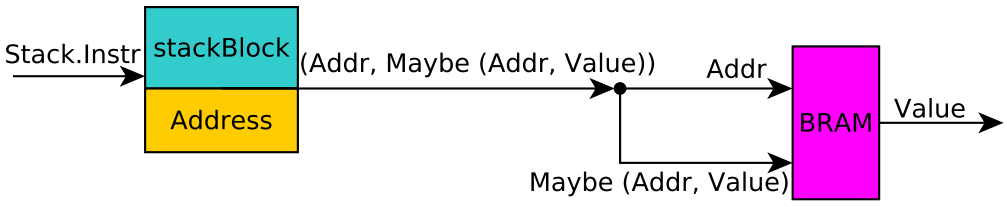


Figure 2: Exc2 stack system

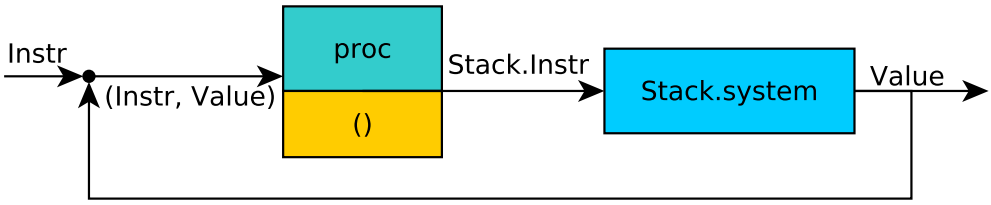


Figure 3: Module Processor for 3.1

is compatible with a blockRAM. Using your own stackControl block and the given blockRAM, compose the system as shown in Figure 2, you only need to use the `unbundle`. Show your function definitions and the result of `testSystem` and `testStackBlock` in your report. Be carefull where to increasing the stackpointer.

Assignment 2.2 (0.5 pts):

Extend you definition of stackController in `Exc2_2_Stack.hs` to add behaviour to pop a value from the stack when it gets the Pop instruction. The system layout does not change here. You can use the code from 2.1 as a starting point. Show your function definitions and result of the `testSystem` and `testStackBlock` function in your report.

Exc3 Processor

In this assignment you will expand the stack module with an expression processor. This processor will receive a signal with instructions of the form `Instr`, and a Value, that should be connected to the output of the stack controller. The stack instruction type is imported from the previous exercise and can be accessed as e.g. `Stack.Push` and `Stack.Pop`, similarly the composed system of the stackController is used with `Stack.system`.

Assignment 3.1 (1 pts):

Start by implementing a `Push` instruction. For the processor, define a mealy machine processor with empty state (the empty type is written as '()'), as an input a tuple of `Instr` and `Value`, and as output a stack instruction. The composed system should look like Figure 3. As output for the system, for now you can just take the `Value` produced by the `Stack.system`. Show your function definitions and result of the `testSystem` function in your report.

Assignment 3.2 (1 pts):

Next, implement a calc instruction for the expression processor. To perform the `Calc` instruction the processor should send two consecutive `Pop` commands to the stack controller. Once both values are received from the stack controller, the processor should apply the correct action, depending on the requested opcode in the original calc instruction. In 2.1 the output of the blockRAM was the output of the system, for this assignment the output of the system should come from the processor. Note that the output of the processor should be a `Maybe Value`, that is either Nothing when the processor does not have a calculation result, or (Just value) when the processor has calculated a

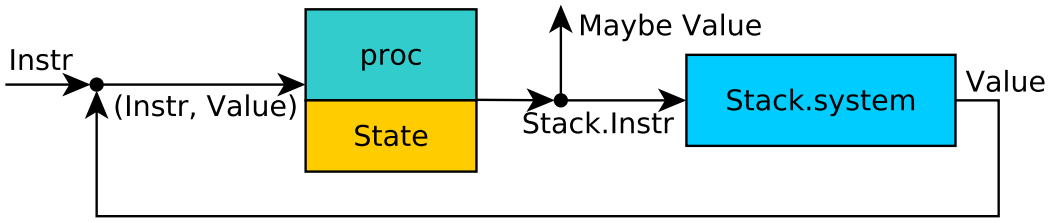


Figure 4: Module Processor for 3.2

value. Do not forget to push the resulting value to the stack. An overview of the system is shown in Figure 4. Show your function definitions and result of the *testSystem* function in your report.

Exc4 Fetch

The expression processor defined in the previous exercise has a major flaw; every time a `Calc` operation is instructed to the processor, it ignores the next two instructions, requiring the programmer to add `Nops` to their code. To solve this, a instruction fetch unit is required. This block will fetch instructions based on a program counter from its own blockRAM. Figure 5 shows the connection to its blockRAM in detail, this should be this file's system function.

Assignment 4 (1 pts):

Extend the `Exc4_Fetch.hs` file with a fetcher, that fetches an instruction every cycle and passes it as an output. If the instruction is a calc instruction, the fetcher should go into a stall state and emit a `Proc.Nop` instruction for two cycles, before continuing with the next instruction. Keep in mind that the first value from the blockRam is undefined. You can ignore the first value from blockRam by having a stall timer larger than 0 as initial state.

Since nothing will be written to the blockRAM you can apply a constant value of Nothing to it, a constant signal can be constructed as follows: `pure Nothing`

For this assignment the system function only consists of the fetching system, it must not include the processor.

Show your function definitions and result of the `testSystem` function in your report.

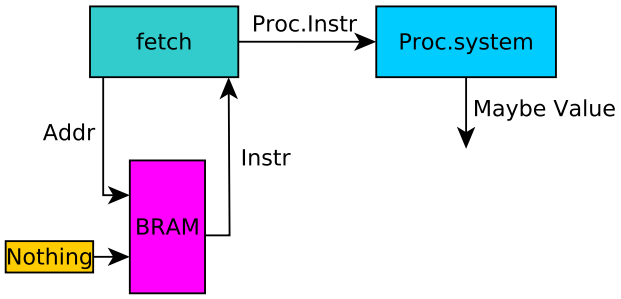


Figure 5: Fetch system for 4

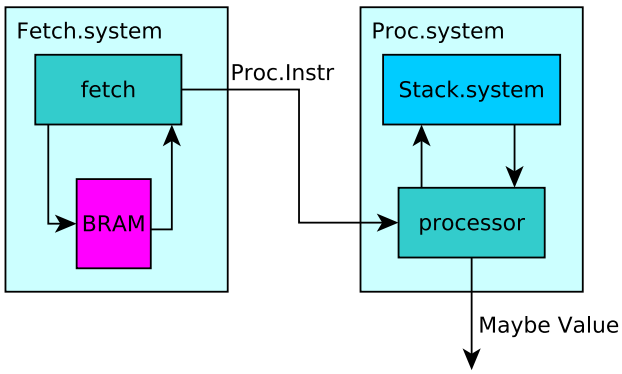


Figure 6: The composed system for 5

Exc5 System

In this assignment you will connect the previous components.

Assignment 5 (0.5 pts):

Connect the previous created systems in one system in `Exc5_System.hs` as illustrated in Figure 6. With everything in place, the program loaded into the instruction blockRAM will be executed and the result will be emitted by the system.

Show your function definitions and result of the `testSystem` function in your report.

Exc6 Processor

In this assignment you will extend the processor.

Assignment 6.1 (1 pts):

Extend the processor with a simple register file. The register file should contain 4 `Values`, and should be stored in the state of the processor.

Two instructions are added to the instruction set:

PushR addr the value stored in the registerfile at address addr should be pushed to the stack.

Save addr One value is popped from the stack and saved at address addr in the registerfile.

Notice that the Save instruction has different timing constraints than the Calc instruction, since only one value has to be popped of the stack. This time, instead of giving you the correct data type, add a new state to ProcState yourself which you can use for the save instruction. As the initial state of the registerfile, use any vector of length 4 of Values. Furthermore, the output of the processor should be changed to always be the first register, instead of the computation result of the ALU. Show your function and datatype definitions and result of the `testSystem` function in your report.

Assignment 6.2 (1 pts):

Later in these exercises you will want to construct processors with a specific initial register file. Therefore, add a system' which has the same type as system but additionally takes a RegisterFile as argument and constructs a processor block with that register file. Show your function definitions and result of the `testSystem'` function in your report.

Exc7

In this assignment you will extend the fetcher. Just as with the Calc instruction, it is required to add a Nop after a Save instruction to make it work.

Assignment 7 (0.5 pts):

Extend in `Exc7_Fetch.hs` your Fetch system to incorporate the `Save` and `PushR` instructions (you may reuse code from exercise 4). Show your function definitions and result of the `testSystem` function in your report.

Exc8

The processor defined in these assignments, including its 16-instruction program, is actually a very rudimentary 1-dimensional motion processor. The first register is the position of the object under simulation, the second its velocity, and the third its acceleration. These variables evolve over time as follows:

$$\text{acc} = acc * 0.9765625$$

$$\text{vel} = vel + acc * \Delta t$$

$$\text{pos} = pos + vel * \Delta t$$

Where Δt is the time-step, 0.0390625.

The initial acceleration, velocity, and position are then of course the initial state of the register file. Furthermore, since the program is only 16 instructions, and so is the length of the program memory. If the program counter overflows at the end of the program, you can start it again. Therefore, simulation of the system already provides a position change over time.

Assignment 8.1 (1 pts):

Take the `system` from assignment 5, paste it in `Exc8_System.hs` under the new type definition (since the output of `Proc.system` was changed to a `Value` instead of a `Maybe Value`), and execute it.

Synthesise the topEntity function using :vhdl or :verilog. Show the RTL schematic with all the blocks connected in your report. Comment on the number of DSP's and registers. Show your function definitions and result of the `testSystem` function in your report.

Assignment 8.2 (1 pts):

The core from 8.1 simulates exactly one object in motion. However, this core can be used to simulate a race along a straight track, with every racer having different abilities (initial velocities and accelerations), but the same starting position.

Using higher order functions create a definition for `system'` which instantiates 4 simulation cores, with different accelerations/velocities chosen based on some arbitrary formula.

With the helper function `wf` a simulation of the positions until one of the racers goes out of bounds is written to a csv file, you can then easily plot it using a spreadsheet program.

Show your function definitions and a plot of the 4 racers in your report.