Practical assignment ECA2:
processor design using Cλash

January 9, 2019

Introduction

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

deadline for handing in your solutions: 2019-02-09 on canvas.

Remarks on delivery

- Add your names and student numbers to the front page of your report.
- For all assignments you are asked to deliver Haskell/CλaSH code, please include your Haskell/CλaSH code in a text box in your report.
- There is no need to deliver VHDL code.
- You can score 8 points in this assignment, this is 0.8 points of the final grade.
- Deliver your report (pdf) and Haskell/CλaSH files (.hs) in a zip file.
- Zip-file name: `ECA2_processor_lastname1_lastname2.zip`.
- Report name: `ECA2_processor_lastname1_lastname2.pdf`.

Preliminary Remarks on Haskell and CλaSH

- A filename of a Haskell/CλaSH design should be of the form `<Filename>.hs`, starting with a capital letter.

- You can transform Haskell to CλaSH by the following steps:

  – Add the line

    ```
    import Clash.Prelude
    ```

    as the second line of your file. Among others, this redefines many standard list functions in Haskell towards corresponding vector functions in CλaSH. For example, in Haskell functions such as *take* and *map* work for lists, whereas in CλaSH they work for vectors. If you need such a function for lists, use *Data.List.take*, *Data.List.map*, etc. However, this only works in the `clash` interpreter and is not in synthesis.

  – Define the hardware types needed, using `Signed`, `Unsigned`, `Vec`, etc.

  – Instructions how to install CλaSH on your own system can be found on `clash-lang.org`.

- Finally, to generate VHDL code using CλaSH, you should define the variable *topEntity* and make sure that its type is not polymorphic and fully determined. Note that Haskell (and thus CλaSH also) can derive the type of an expression, to be checked in Haskell/CλaSH with:

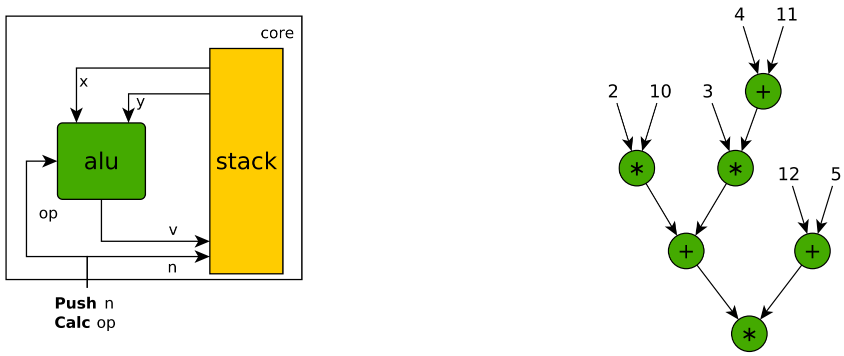
  ```
  :t <expression>
  ```

  VHDL code is generated by CλaSH using the command[1]

  ```
  :vhdl
  ```

  This will put the resulting VHDL code in a subdirectory `vhdl/<Filename>`. We assume that you have access to *Quartus* for synthesizing the VHDL generated by CλaSH.

[1] Don't bother about possible "`Can't make testbench`" errors, they are not relevant in our context.



(a) Schematic

(b) Expression tree

Figure 1: Stack machine

1 Stack processor (Haskell, ghci)

Create a file named `CPU_St.hs` and define your own data type *Opc* and *Instr*, start the file with the following:

```
module CPU_St where

data Opc = Add | Mul
  deriving Show

data Instr = Push Int | Calc Opc
  deriving Show

program = [Push 2, Push 10, Calc Mul, Push 3, Push 4, Push 11,
 Calc Add, Calc Mul, Calc Add, Push 12, Push 5, Calc Add, Calc Mul]
```

Deriving means that the compiler will figure out how to print the datatype. Define an *alu* functions that takes 3 arguments; opcode, x, and y, and performs the selected computation (see Figure 1a). Define a *core* function which takes the 2 arguments; the stack (list) and an instruction, and produces a new stack. Simulate the behaviour of your specification by means of the *scanl* function:

```
test = scanl core [] program
```

Assignment 1 (2 pts):

Provide your `CPU_St.hs` file and include in the report the code and output of the test function.

2 Heap and stack processor (Haskell, ghci)

Create a file named `CPU_HpSt.hs` with the following:

```haskell
module CPU_HpSt where

data Opc = Add | Mul
  deriving Show

data Value = Const Int | Addr Int
  deriving Show

data Instr = Push Value | Calc Opc
  deriving Show

program = [Push (Const 2), Push (Addr 0), Calc Mul, Push (Const 3),
 Push (Const 4), Push (Addr 1), Calc Add, Calc Mul, Calc Add,
 Push (Const 12), Push (Const 5), Calc Add, Calc Mul]
```

Define a function *value* that yields the numerical value of element type **Value**. Note: for a given heap and instruction this function gives the value that is stored at a specific address). Define the function *core* and *alu* (Figure 2a):

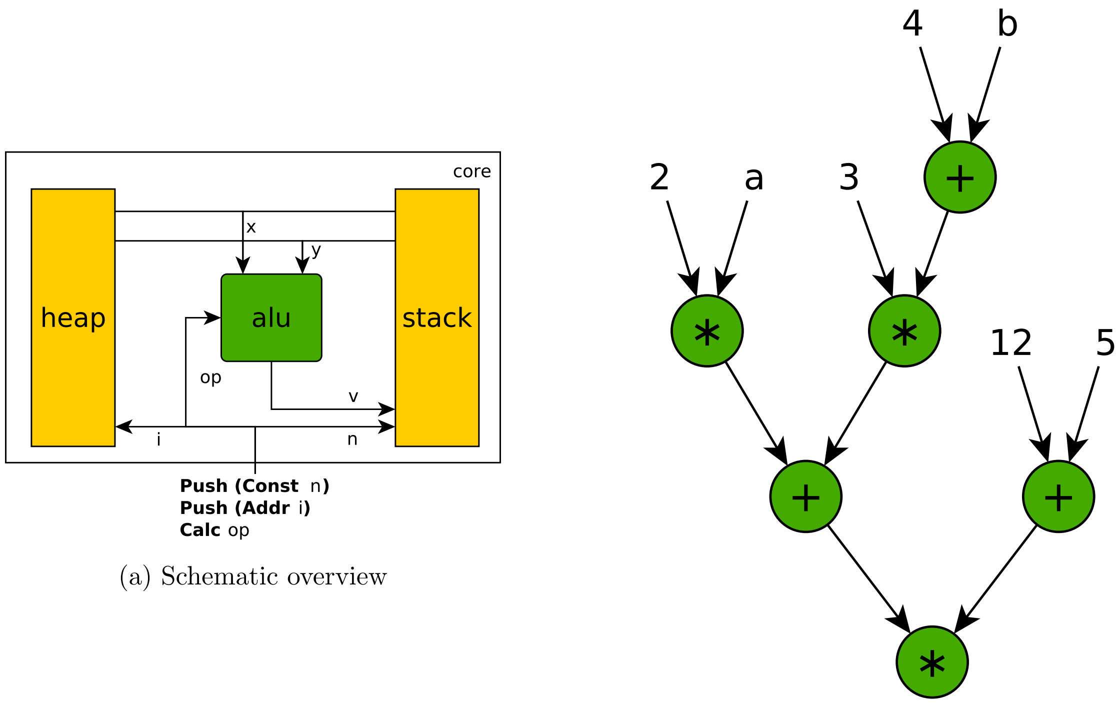
```haskell
core (stack,heap) instr = (stack', heap')
 ...
```

Test the design with the *scanl* function:

```haskell
test = scanl core ([],[10,11]) program
```

Assignment 2 (1 pts):

Provide your `CPU_StHp.hs` and include in the report the code and output of the test function.



(a) Schematic overview

(b) Expression tree

Figure 2: Heap and stack machine

3 Heap, stack, and program counter (Haskell, ghci)

Create a file `CPU_HpStPc.hs` and copy the content from the previous file (`CPU_HpSt.hs`) into the file. Change the module name to *CPU_HpStPc*. Add an instruction **Send**, that sends a value to the output (assume: non-negative). For all other instructions, the output should be -1. The *core* definition should be in the form of a Mealy Machine (state → input → (state, output)).

```
core (stack,heap) instr = ((stack', heap'), out)
 ...
```

You can test the *core* function by the simulation function given below: The output should be a list of -1's.

```
sim f s [] = []
sim f s (x:xs) = z:sim f s' xs
  where
    (s', z) = f s x

test = sim core ([],[11,10]) program
```

Until now the instructions are given as input arguments. Change the *core* function such that it receives the list of instructions once, and uses a program counter to fetch the different instructions (as input you may assume that there will only a clock tick as input, e.g., represented by the number 1):

```
core prog (pc, stack,heap) tick = ((pc', stack', heap'), out)
  ...
```

You can test your code by the *sim* function again, (you will get an exception that the index is to high):

```
test = sim (core program) (0,[],[10,11]) $ repeat 1
```

Add a value **Top** to the type *Value* that yields the value on the top of the stack, and an instruction **Pop** that removes the top value of the stack. Extend the definition of *core* accordingly. Add **Send Top** to the end of the program, simulate using the *sim* function as before.

```
...
program = [Push (Const 2), Push (Addr 0), Calc Mul, Push (Const 3),
 Push (Const 4), Push (Addr 1), Calc Add, Calc Mul, Calc Add,
 Push (Const 12), Push (Const 5), Calc Add, Calc Mul, Send Top]
...
```

Assignment 3 (1 pts):

Provide your `CPU_StHpPc.hs` and include in the report the code and output of the test function.

4 Heap, stack, program counter, and single stack access (Haskell, ghci)

Create a file `CPU_HpStPcReg.hs` and copy the content from the previous file (`CPU_HpStPc.hs` into the file. Change the module name to *CPU_HpStPcReg*. Until now the stack can provide two values simultaneously. Rewrite the *core*, such that only a single value can be read from the stack every cycle. Simulate (*sim*) using the program below.

```
...
program = [Push (Const 2), Push (Addr 0), Pop, Calc Mul, Push (Const 3),
 Push (Const 4), Push (Addr 1), Pop, Calc Add, Pop, Calc Mul, Pop,
 Calc Add, Push (Const 12), Push (Const 5), Pop, Calc Add, Pop,
 Calc Mul, Send Top, Push (Const 2), Send Top, Pop, Send Top]
...
```

Assignment 4 (1 pts):

Provide your `CPU_StHpPcReg.hs` and include in the report the code and output of the test function.

5 CPU Fixed (CλaSH, clashi)

Create a file `CPU_Fixed.hs` and copy the content from the previous file, `CPU_HpStPcReg.hs`, into the file. Reformulate your code in terms of CλaSH. Both heap and stack become vectors with a fixed length, hence you will need a stack pointer that indicates the top of the stack, and that is changed when a value is pushed to or popped from the stack. Use the *simulate* function in CλaSH to simulate your design.

Assignment 5 (3 pts):

Provide your `CPU_Fixed.hs`, and include in the report the code, output of *simulate* function, and RTL schematic in your report. Comment on the correspondence between the RTL schematic, code, and resource consumption.