The Sprockell

Jan Kuper
University of Twente
j.kuper@utwente.nl

In this paper we describe a Von Neumann type single core processor. It has many simplifications in comparison with a single core realistic processor, but it nevertheless is Turing complete. We also describe the role it plays in teaching both functional programming as hardware design.

1 Introduction

At the University of Twente we use Haskell in courses on hardware design to specify hardware architectures, and to translate to VHDL by CλaSH (see [1]), so that these archtitectures can be mapped onto an FPGA. In that context a simple single core processor is developed, in order to let students experiment with extensions to the processor, and being able to simulate the processor and their extensions in Haskell. Besides, after some minor transformations the code is translatable into synthesizable VHDL by CλaSH, so that it can be mapped onto an FPGA. Apart from hardware courses, the processor is also used in the course on functional programming, for students to develop a programming language as embedded language within Haskell and to write a compiler for that programming language.

The processor is called *Sprockell*: a *S*imple *proc*essor in Has*kell* (see Figure 1). It is an instruction set architecture which has many simplifications in comparison with a real processor, for example, we will assume that the execution of an instruction as well as fetching data from memory takes only one clock cycle, there is no pipelining, there are no cache memories, there is no I/O. We assume a program memory that is separated from data memory, and only one program can be executed at the same time.

2 Definition of the Sprockell

In Figure 1 it can be seen that the program memory (pmem) contains a list of instructions (see below for the complete instruction set). The decode function[1] $\mathcal{D}$ decodes these instructions one by one and sends signals onto all its outgoing wires. The formulation “sends signals onto all its outgoing wires” is represented in the definition of the function $\mathcal{D}$ by the fact that the result of $\mathcal{D}$ for every instruction is a record consisting of 13 fields, where every field corresponds to one of the outgoing wires of the decoder.

[1]In order to keep the definitions concise, we will use symbols for functions and variables that are not directly recognizable by Haskell. However, it will be immediately clear how to turn these name sto Haskell recognizable identifiers

Submitted to:
TFPIE 2013

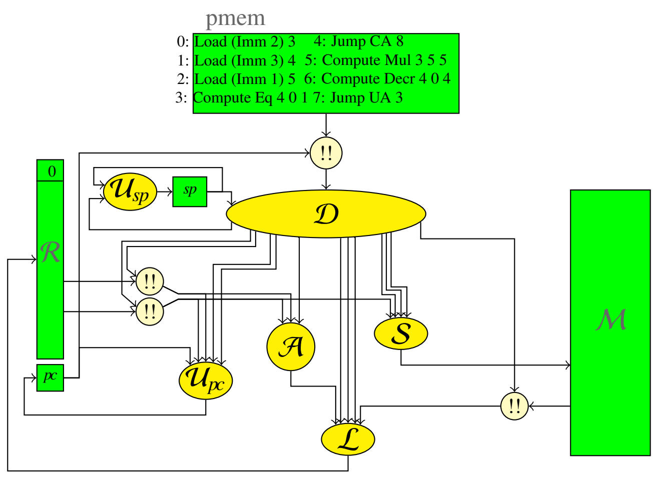

pmem
0: Load (Imm 2) 3   4: Jump CA 8
1: Load (Imm 3) 4   5: Compute Mul 3 5 5
2: Load (Imm 1) 5   6: Compute Decr 4 0 4
3: Compute Eq 4 0 1   7: Jump UA 3
!!
0
$\mathcal{U}_{sp}$
sp
$\mathcal{D}$
$\mathcal{R}$
!!
!!
$\mathcal{A}$
$\mathcal{S}$
$\mathcal{M}$
pc
$\mathcal{U}_{pc}$
!!
$\mathcal{L}$

Figure 1: Sprockell

The Sprockell is a *load-store* architecture, where the load function $\mathcal{L}$ is able to load data from various sources into some register in the register bank $\mathcal{R}$. The sources of these data can be a constant value delivered by the decoder, it can be the output of the alu, or it can be a value from some address in data memory. Which value the load function has to choose, is determined by a sepcial code sent to the load function by the decoder. Clearly, also the address of the register in which the load function has to put the value, is coming from the decoder.

The store function $\mathcal{S}$ saves a value in data memory. As with the load function, this value may come from different sources: it may be a constant sent by the decoder, or it may be a value from some address in the register bank. Here too, the decoder delivers the information which value to choose, which register to read, and which address in data memory to save to.

The alu $\mathcal{A}$ performs an operation, indicated by an opcode, on two values from the register bank, and sends its resut to the load function $\mathcal{L}$.

The last elements we mention in this introductory description are the program counter and the stack pointer. As always, the program counter tells which instruction from program memory should be fetched for the decoder (shown in Figure 1 by the indexing operation !! from Haskell). The program counter is stored in a register which is updated by the function $\mathcal{U}_{pc}$, the program counter update function, based on information from again the decoder. For the stack pointer the same holds: it is stored in a register that is updated by the stack pointer update function $\mathcal{U}_{sp}$.

So, all in all the state of the architecture consists of the register bank $\mathcal{R}$, data memory $\mathcal{M}$, and two registers for the program counter *pc*, and for the stack pointer *sp*. The Sprockell itself is defined as a function which transforms its state every clock cycle, based on the instruction that has to be executed. In the sections below we will formalize the above intuitive descriptions of the various subcomponents and combine them in the definition of the Sprockell as a whole.

We remark that in order to save space and to have some viusal recognition based on the names of the components, we choose for a more mathematical formulation. However, this formulation may be readily translated into Haskell in a word for word fashion, by choosing names fro the symbols, such as *alu* for $\mathcal{A}$, *load* for $\mathcal{L}$, *dataMemory* for $\mathcal{M}$, etcetera. Since Haskell recognizes Unicode, one might also choose to leave some of the symbols unchanged, and the result will nevertheless be an executable Haskell program, and simulation can be done with the same function *simulate* as before.

The specification given below is complete in the sense that it can also be mapped onto real hardware, e.g., onto an FPGA, thus producing a soft core on an FPGA. However, in order to give the code to CλaSH to be translated into synthesizable code, still some mainly cosmetic massaging has to be done on the Haskell code.

2.1 Memory structure

As mentioned above, the *state* of the Sprockell consists of the register bank $\mathcal{R}$, the data memory $\mathcal{M}$, and the two registers *pc* and *sp* for the program counter and the stack pointer, respectively. For reasons of simplicity we choose to let all values be integers, and $\mathcal{M}$ and $\mathcal{R}$ be lists of integers:

Register bank:	$\mathcal{R} :: [Int]$
Data memory:	$\mathcal{M} :: [Int]$
Program counter:	$pc :: Int$
Stack Pointer:	$sp :: Int$

Note that for real hardware it is not sufficient to choose for integers, nor for lists of integers: for integers one has to choose the number of bits with which the integers will berepresented, and also for lists one has to make a choice for the length of the list. We will come back to this in Section **??**.

To update the register bank or the data memory we define an update operation <~ to put a value *v* on position *i* in a list:

$$\begin{aligned} xs \ \lessdot\!\sim\ (i,v) &= ys +\!\!+ [v] +\!\!+ zs \\ &\quad \textbf{where} \\ &\quad (ys, \_{:}zs) = splitAt\ i\ xs \end{aligned}$$

Applying this operation to the register bank or to the data memory has the following limitations:

- register 0 of the register bank always contains the value 0, so putting a value in this register means that the value will be lost,
- before putting a value in the data memory, it has to be enabled for writing.

2.2 The alu $\mathcal{A}$

Concerning the functional components in the *Sprockell*, we start with the alu function $\mathcal{A}$. As can be seen in Figure 1, the alu has three input signals. Thus, the function $\mathcal{A}$ that specifies the alu has three arguments. The first of these arguments is the opcode *opc* which decides which operation the alu should perform, the other two arguments *x* and *y* are the values on which this operation should be performed. The opcodes are defined as an *embedded language*, i.e., as an algebraic data type in Haskell, which can be extended as desired:

$$\textbf{data}\ \ OpCode\ =\ NoOp \mid Id \mid Incr \mid Decr \mid Neg \mid Add \mid Sub \mid Mul \mid Eq \mid Gt \mid \cdots$$

The meaning of these opcodes become clear in the definition the alu function $\mathcal{A}$, which is a simple case-expression, defined by *pattern matching* on the opcode:

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathcal{A}\ \ opc\ x\ y = \ \textbf{case}\ opc\ \textbf{of}} \\
\quad NoOp & \text{−>} & 0 \\
\quad Id & \text{−>} & x \\
\quad Incr & \text{−>} & x+1 \\
\quad Decr & \text{−>} & x-1 \\
\quad Neg & \text{−>} & -x \\
\quad Add & \text{−>} & x+y \\
\quad Sub & \text{−>} & x-y \\
\quad Mul & \text{−>} & x*y \\
\quad Eq & \text{−>} & tobit\ (x == y) \\
\quad Gt & \text{−>} & tobit\ (x > y) \\
 & \vdots & \\
\multicolumn{3}{l}{\textbf{where}} \\
\multicolumn{3}{l}{tobit\ True = 1} \\
\multicolumn{3}{l}{tobit\ False = 0}
\end{array}$$

Note that in Haskell the relation ">" results in a boolean, so the function *tobit* is needed to transform this into an integer.

2.3 The load function $\mathcal{L}$

The *load* function $\mathcal{L}$ has several input values:

- we choose to let the result of the load function $\mathcal{L}$ be the updated register bank as a whole, so also the register bank $\mathcal{R}$ itself is an argument to the load function,
- three values from which the function $\mathcal{L}$ has to choose to put into the register bank: an immediate value c coming from the decoder, a value from data memory d, or the output z from the alu,
- a code *ldc* to tell the load function which value to put in the register bank, or not to load anything at all,
- of course, the register r in which to put the value.

The codes which value to load is defined in an embedded language *LoadCode*:

$$\textbf{data}\ \ LoadCode\ = \ NoLoad \mid LdImm \mid LdAddr \mid LdAlu$$

Now the definition of the load function $\mathcal{L}$ again is a straightforward case-expression, though the case where no value has to be loaded into the register bank is defined in a separate clause:

$$\begin{array}{l}
\mathcal{L}\ \ NoLoad\ \mathcal{R}\ r\ (c,d,z)\ = \ \mathcal{R} \\
\mathcal{L}\ \ ldc\ \mathcal{R}\ r\ (c,d,z)\ = \ \mathcal{R} <\sim (r,v) \\
\qquad\qquad\qquad \textbf{where} \\
\qquad\qquad\qquad\quad v = \ \textbf{case}\ ldc\ \textbf{of} \\
\qquad\qquad\qquad\qquad \begin{array}{lll} LdImm & \text{−>} & c \\ LdAddr & \text{−>} & d \\ LdAlu & \text{−>} & z \end{array}
\end{array}$$

2.4 The store function $\mathcal{S}$

The *store* function $\mathcal{S}$ has the following input arguments:

- as with the load function $\mathcal{L}$, we choose to let the result of the store function $\mathcal{S}$ be the updated data memory as a whole, so also the data memory $\mathcal{M}$ itself is an argument to the function $\mathcal{S}$,
- two values from which the function $\mathcal{S}$ has to choose to put into the register bank: an immediate value c coming from the decoder, or a value x from data memory,
- a code *stc* to tell the store function which value to put in the data memory, or not to store anything at all,
- of course, the address a at which to store the value.

The codes which value to store are again defined in an embedded language *StoreCode*:

data *StoreCode* = *NoStore* | *StImm* | *StReg*

Again, the definition of the store function $\mathcal{S}$ is a straightforward case-expression, taking the *NoStore* case as a separate clause leaving the data memory $\mathcal{M}$ unchanged:

```
S NoStore M a (c,x) = M
S stc M a (c,x) = M <~ (a,v)
                  where
                    v = case stc of
                          StImm  ->  c
                          StReg  ->  x
```

2.5 The program counter update function $\mathcal{U}_{pc}$

The program counter is updated by the function $\mathcal{U}_{pc}$, based on a jump code to be provided by the decoder. The jump codes are defined in an embedded language *JumpCode*:

data *JumpCode* = *NoJump* | *UA* | *UR* | *CA* | *CR* | *Back*

The meaning of the jump codes is as follows:

- *NoJump*: just go to the next instruction,
- in *UA*, *UR*, *CA*, *CR* the *U*/*C* stand for *Unconditional* and *Conditional*, respectively, i.e., jump in any case, or based on the value x (0 or 1) of a condition. *A*/*R* stand for *Absolute* and *Relative*, respectively, i.e., jump to instruction with number n, or jump a n instructions forward (backward in case n is negative) from the current instruction,
- *Back* says that the program counter can jump back to a previously remembered instruction, to be used in case of, e.g., return from a subroutine.

The program counter update function now again is straightforwardly defined by a case-expression (i*pc* is the program counter, *jmpc* the program counter code, *y* the previously stored program counter):

```
U_pc (jmpc,x) (n,y) pc = case jmpc of
                           NoJump  ->  pc+1
                           UA      ->  n
                           UR      ->  pc+n
                           CA       |  x==1       -> n
                                    |  otherwise  -> pc+1
                           CR       |  x==1       -> pc+n
                                    |  otherwise  -> pc+1
                           Back    ->  y
```

2.6 The stack pointer update function $\mathcal{U}_{sp}$

The stack is a dedicated sequence of memory locations in the data memory, starting at a freely to determine memory address. The idea of defining the stack pointer update function should be clear by now, and we give the definitions staright away. The stack pointer update code:

```
data SPCode = Up | Down | None
```

The stack pointer update function, where *sp* is the stack pointer, and *spc* the stack pointer code:

```
U_sp spc sp = case spc of
                Up    ->  sp+1
                Down  ->  sp−1
                None  ->  sp
```

2.7 The instruction set

Also the *instruction set* is defined as an embedded language, called *Assembly*:

```
data Assembly  =  Compute OpCode Int Int Int
               |  Jump JumpCode Int
               |  Load Value Int
               |  Store Value Int
               |  Push Int
               |  Pop Int
```

The type *Value* consists of two sorts of values: immediate values (constants) and values indicated by their address in data memory. It is defined as follows:

```
data Value  =  Addr Int
            |  Imm Int
```

The following table describes the meaning of the instructions:

Compute *opc* i_0 i_1 i_2: the alu will perform the operation *opc* on the values from registers i_0 and i_1, and the result will be put in register i_2,

Jump *jmpc n*: the program counter will be changed by the number *n*, based on the jump code *jmpc*,

Load (*Imm n*) *j*: the value *n* will be loaded into register *j*,

Load (*Addr i*) *j*: the value from address *i* in data memory will be loaded into register *j*,

Store (*Imm n*) *j*: the constant *n* will be stored in data memory at address *j*,

Store (*Addr i*) *j*: the value from register *i* will be stored in data memory at address *j*,

Push *i*: the value from register *i* will be pushed onto the stack,

Pop *i*: the top value of the stack ill be loaded into register *i*.

The program memory is a list of assembly instructions, i.e., the program memory has type [*Assembly*].

2.8 The decode function $\mathcal{D}$

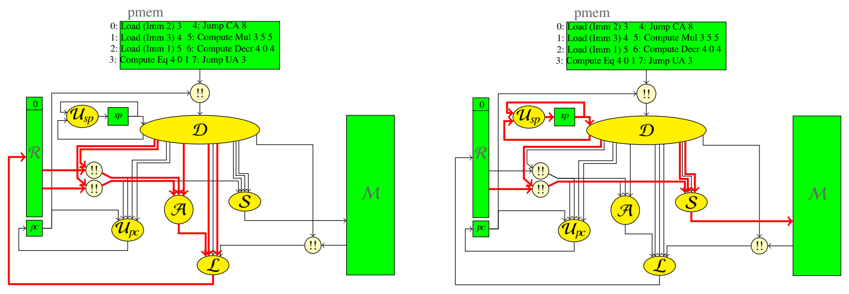
The *decode* function $\mathcal{D}$ translates an instruction into signals for all other functions in the Sprockell. That is to say, the function $\mathcal{D}$ gets two arguments: the stack pointer *sp* and an assembly instruction α, and produces a record consisting of 13 fields, as shown in picture 1 This record type represents the “machine code” and is defined as:

```
data MachCode = MachCode { ldCode   :: LoadCode,
                           stCode   :: StoreCode,
                           opCode   :: OpCode,
                           jmpCode  :: JumpCode,
                           spCode   :: SPCode,
                           jmpN     :: Int,
                           immvalR  :: Int,
                           immvalS  :: Int,
                           reg0     :: Int,
                           reg1     :: Int,
                           addr     :: Int,
                           toreg    :: Int,
                           toaddr   :: Int }
```

We define an empty record for the machine code $\mathbf{C_0}$:

```
C0 = MachCode { ldCode=NoLoad, stCode=NoStore, opCode=NoOp,
                jmpCode=NoJump, spCode=None, jmpN=0,
                immvalR=0, immvalS=0,
                reg0=0, reg1=0, addr=0, toreg=0, toaddr=0 }
```

The function $\mathcal{D}$ now is defined by updating the empty machine code $\mathbf{C_0}$ for every instruction separately, by using a case-expression. Note that the fact that the instruction set is defined as an embedded language, offers the possibilty of pattern matching on each instruction:


pmem
0: Load (Imm 2) 3 4: Jump CA 8
1: Load (Imm 3) 4 5: Compute Mul 3 5 5
2: Load (Imm 1) 5 6: Compute Decr 4 0 4
3: Compute Eq 4 0 1 7: Jump UA 3
!!
$\mathcal{U}_{sp}$
sp
$\mathcal{D}$
0
$\mathcal{R}$
pc
$\mathcal{U}_{pc}$
$\mathcal{A}$
$\mathcal{S}$
$\mathcal{L}$
$\mathcal{M}$

(a) *Compute instruction* (b) *Push instruction*

Figure 2: Examples of the effect of instructions

$\mathcal{D}\ sp\ \alpha\ =$ **case** α **of**

Compute *opc* $i_0\ i_1\ i_2$	–> $\mathbf{C}_0\{ldCode{=}LdAlu,\ opCode{=}opc,\ reg0{=}i_0,\ reg1{=}i_1,\ toreg{=}i_2\}$
Jump *jc n*	–> $\mathbf{C}_0\{jmpCode{=}jc,\ jmpN{=}n,\ reg0{=}1,\ reg1{=}6\}$
Load (*Imm n*) *j*	–> $\mathbf{C}_0\{ldCode{=}LdImm,\ immvalR{=}n,\ toreg{=}j\}$
Load (*Addr i*) *j*	–> $\mathbf{C}_0\{ldCode{=}LdAddr,\ addr{=}i,\ toreg{=}j\}$
Store (*Imm n*) *j*	–> $\mathbf{C}_0\{stCode{=}StImm,\ immvalS{=}n,\ toaddr{=}j\}$
Store (*Addr i*) *j*	–> $\mathbf{C}_0\{stCode{=}StReg,\ reg0{=}i,\ toaddr{=}j\}$
Push *i*	–> $\mathbf{C}_0\{stCode{=}StReg,\ spCode{=}Up,\ reg0{=}i,\ toaddr{=}sp{+}1\}$
Pop *i*	–> $\mathbf{C}_0\{ldCode{=}LdAddr,\ spCode{=}Down,\ addr{=}sp,\ toreg{=}i\}$

In order to illustrate the definition of the decoder, we give two examples. In Figure 2(a) it is shown which extra signals (marked with red) in comparison to the empty machine code are activated by the decode function$\mathcal{D}$ to execute the *compute* instruction. From the corresponding clause in the definition of $\mathcal{D}$ we derive that these extra signals are:

- two register addresses by which the values for the alu $\mathcal{A}$ are selected,
- the opcode signal directly to the alu $\mathcal{A}$,
- two signals to the load function $\mathcal{L}$, saying that the outcome z of $\mathcal{A}$ has to be put in the register bank, and to which register that value has to be put.

Likewise, Figure 2(b) can be compared to the clause in the decode function $\mathcal{D}$ to see that the following signals are added to the empty machine code for the *push* instruction::

- the value from register i has to be selected,
- the store function $\mathcal{S}$ should know that the value x from the register bank has to be put in data memory $\mathcal{M}$, and that it has to be stored on top of the stack, i.e., at addrees $sp{+}1$,
- since an element is put on top of the stack, the stack pointer has to be increased by one, such that the stack pointer again points to the top element of the stack.

We leave it to the reader to check the decoding of the othe rinstructions.

2.9 The Sprockell function

Finally we come to the function *sprockell*, in which all the above defined functions are composed together. We first remark that the function *sprockell* is of the pattern as described by a Mealy Machine (see Section **??**):

- it is parameterized with a sequence αs of instructions in the program memory,
- its state $(\mathcal{R},\mathcal{M},pc,sp)$ consists of the register bank, the data memory, and the program counter and stack pointer,
- the input is irrelevant, since for these lecture notes we chose to leave the processor without I/O. The input may be interpreted as a clock tick,
- the result consists of the updated state and some output, which can be freely defined, e.g., as a specific memory element to follow the changes.

$$
\begin{array}{l}
sprockell\ \ \alpha s\ \ (\mathcal{R},\mathcal{M},pc,sp)\ \ \_\ \ =\ \ ((\mathcal{R}',\mathcal{M}',pc',sp'),\ out) \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{where} \\
\qquad\qquad\qquad\qquad\qquad\qquad \begin{array}{lcl}
MachCode\{..\} & = & decode\ sp\ (\alpha s\,!!\,pc) \\[4pt]
\mathcal{R}^{+} & = & \mathcal{R} +\!\!+ [pc] \\
(x,y) & = & (\mathcal{R}^{+}!!reg0, \mathcal{R}^{+}!!reg1) \\
z & = & \mathcal{A}\ \ opCode\ x\ y \\
d & = & \mathcal{M}!!addr \\[4pt]
\mathcal{R}' & = & \mathcal{L}\ \ ldCode\ \mathcal{R}\ toreg\ (immvalR,d,z) \\
\mathcal{M}' & = & \mathcal{S}\ \ stCode\ \mathcal{M}\ toaddr\ (immvalS,x) \\
pc' & = & \mathcal{U}_{pc}\ (jmpCode,x)\ (jmpN,y)\ pc \\
sp' & = & \mathcal{U}_{sp}\ \ spCode\ sp \\[4pt]
out & = & \cdots
\end{array}
\end{array}
$$

Note that the first line of the where-clause says that we may use the field names of the machine code record as if they were normal variables. The next line defines an "extended register bank" such that we can also choose the value of the program counter by indexing this extended register. That is practical in case a value of the program counter is saved on the stack in case of subroutine calls.

The variables x and y are defined as the values from the register bank at addresses $reg0$ and $reg1$, which come from the machine code vector, i.e., they are chosen by the decoder. The variable z results from applying the alu $\mathcal{A}$ to these values x and y, and applying the operation indicated by $opCode$, again afield from the machine code record. Likewise, d is the value from the data memory $\mathcal{M}$.

In the last four lines the various parts of the state are updated by applying the corresponding update functions to their arguments.

3 Simulation

The Sprockell can now be simulated by choosing an appropriate sequence $\alpha s$of instructions, and appropriate values for the initial register bank and data memory. Clearly, the expected values to fill register bank and data memory are zeroes. The program counter should start at 0, and the stackpointer at that value that indicates the address in data memory where the stack starts. Now the processor may be simulated by the following expression:

$$\mathit{simulate}\ \ (\mathit{sprockell}\ \mathit{as})\ \ (\mathcal{R}_0, \mathcal{M}_0, pc_0, sp_0)\ \ [0..]$$

The list of instructions in the program memory in Figure 1 calculates the value of 2^3. It puts 2 in register 3, 3 in register 4, and puts the result in register 5. If we define *out* above as

$$(pc, \mathcal{R}!!1, \mathcal{R}!!3, \mathcal{R}!!4, \mathcal{R}!!5)$$

then the simulation gives the following sequence of 5-tuples:

$$\begin{array}{l}
[(0,0,0,0,0),\ (1,0,2,0,0),\ (2,0,2,3,0),\ (3,0,2,3,1),\\
\ (4,0,2,3,1),\ (5,0,2,3,1),\ (6,0,2,3,2),\ (7,0,2,2,2),\ (3,0,2,2,2),\\
\ (4,0,2,2,2),\ (5,0,2,2,2),\ (6,0,2,2,4),\ (7,0,2,1,4),\ (3,0,2,1,4),\\
\ (4,0,2,1,4),\ (5,0,2,1,4),\ (6,0,2,1,8),\ (7,0,2,0,8),\ (3,0,2,0,8),\\
\ (4,1,2,0,8),\ (8,1,2,0,8),\ (*** \text{ Exception : Prelude.(!!) : index too large}
\end{array}$$

The first line contains the initialisation of the values 2, 3, 1 in the registers 3, 4, 5 (respectively), and the other lines all start with the result of instruction 3 which computes whether register 4 equals zero. Note that the values in the registers are the values *before* the instruction indicated by the program counter (on the first position each 5-tuple) is executed.

Note also that instruction 3 puts the result in register 1, since that is the register where the conditional jump looks to decide whether it should jump or not (as determined by the choice *reg*0=1 in the definition of the decode function for the jump instruction).

Finally, note that the simulation ends by an “index too large” error, since instruction 4 will cause that the program counter gets the value 8, whereas the largest index of the sequence is 7. Clearly, that is not the most elegant solution, but in the framework of these lecture notes, we don’t elaborate this point any further.

4 Educational usage

Above we described a simple though Turing complete processor in order to show the naturality by which the components and the total processor can be specified and simulated using Haskell. AAs mentioned above, this definition is given to students at the University of Twente to experiment with various problems, and to execute a series of tasks on the Sprockell.

First of all, the Sprockell is used in the course on Functional Programming, in which students have to do a homework exercise with it. This exercise consists of the design of a programming language and to write a compiler for it such that the resulting sequence of instructions is executable on the Sprockell. Clearly, most students come up with an imperative programming language, defined as an embedded language, along the following lines:

```
type Variable     =  String
data Expression   =  ···
data Program      =  Program [Statement]
data Statement    =  Assign Variable Expression
                  |  If Expression [Statement] [Statement]
                  |  While Expression [Statement]
```

Now a compiler simply is a function from these data types to a list of assembly instructions, using some additional parameters for variable look-up tables, etc.

It should be noted that tathe moment that the students participate in the course on Functional Programming, most of them did *not* take a course on compiler construction yet. Nevertheless, they all manage to complete the task within two weeks (the course is every year taken by some 30-40 students). Several students also include subroutines in their language definitions, which in turn may contain nested subroutines. The exercise is quite stimulating, and students continue working on it to define pointers, etcetera. In the homework assignment students arefree to change the architecture of the Sprockell, if they wish to do so. Some indeed do, e.g., to add I/O.

Students are encouraged to add a textual formulation of their programming language and to write a parser that translates their programs into the embedded language that they designed before.

The course on Functional Programming is an introductory course, and there is no time to study the implementation of functional languages. There are plans to develop a continuation of the course, and in that there will be space to study the implementation of a functional language and to map that to the Sprockell.

The clarity of the definition of the Sprockell, gave rise to teh lecture of teh course on Compiler Construction at the University of Twente to invite us for a guest lecture on an approach through Haskell. Currently, the course on Compiler Construction uses traditional tools such as ANTLR.

Apart from the above mentioned courses, the Sprockell is also used in a hardware design course. In those courses students are supposed to experiment with the architecture itself, for example to add I/O, to pipeline the processor, to add memory hierarchies, to add assembly instructions that take more than one clock cycle. A popular extension is to turn the processor into a universal one, i.e., to store a program in datamemory and to define an operating system, for example for multi-tasking, etcetera.

In particular the availability of CλaSH opens a wealth of possibilities, for example to design routers by which several copiesof the Sprockell can be connected and put on an FPGA. That also opens the possibility to evaluate the performance and resource usage on the FPGA.

We do not have a systematic evaluation of students approval of teh Sprockell, but the overwhelming reaction of students is thatthey appreciate the approach. They quickly grasp the idea and can easily apply it to all sorts of tasks as described above. Besides, they feel stimulated to experiment form themselves with processor architectures as well as with programming language design and compiler construction. The comment most heard from students is that they can easily extend the architecture and thus gain a lot of insight.

References

[1] C. P. R. Baaij, M. Kooijman, J. Kuper, W. A. Boeijink & M. E. T. Gerards (2010): *CλaSH: Structural Descriptions of Synchronous Hardware using Haskell*. In: *Proceedings of the 13th EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools, Lille, France*, IEEE Computer Society, USA, pp. 714–721.