Bachelor's Thesis

Implementation of a multi One Instruction Set Computer (mOISC) in the hardware description language Clash

by

Jakob Groß

Matrikel-Nr.: 22635422

Supervision:
Prof. Dr. Oliver Keszöcze

August 26, 2022

Aufgabenstellung zur Bachelorarbeit

Thematischer Hintergrund

One Instruction Set Computer (OISC) führen lediglich einen einzige Art von Instruktion aus. Je nach gewählter Instruktion kann sie aber dennoch Turing-vollständig sein. Anwendungen finden sie z.B. in der Bildverarbeitung [1]. Auch die Nutzung für verschlüsselte Berechnungen wurde untersucht [2]. Aufgrund ihrer Einfachheit eignen sich OISCs insbesondere dafür, die Arbeitsweise eines Computers im Detail auf der Hardware-Ebene zu verstehen. Im Hardwareentwurf wird die Funktionalität häufig mittels einer Hardwarebeschreibungssprache (engl. Hardware Description Language, HDL) beschrieben. Diese Beschreibung wird dann für die gewünschte Zielarchitektur (FPGA, ASIC) synthetisiert. Hier gibt es zwei Sprachen, VHDL und Verilog, die den de-facto Industriestandard darstellen. In den letzten Jahren sind viele HDLs entworfen worden, die versuchen, einen moderneren Ansatz zu verfolgen. So ist die Referenzimplementierung des RISC-V Prozessors in Chisel (https://github.com/chipsalliance/rocket-chip) geschrieben worden. Auch wenn Chisel, und viele andere, neue HDLs, mehr Komfort und zum Teil neue Ansätze im Vergleich zu VHDL/Verilog bieten, sind sie dennoch (syntakisch) sehr nah an ihren geistigen Vorgängern. Die moderne HDL Clash hingegen setzt auf die Programmiersprache Haskell auf. Haskell/Clash bieten ein sehr hohes Abstraktionsniveau und gute Compiler-Unterstützung.

Beschreibung Problemstellung

Im Rahmen dieser Bachelorarbeit soll nun ein konkreter one instruction set computer, der mIOSC aus [3], in Clash implementiert werden. Das Ergebnis soll mit der Implementierung in der klassischen Sprache VHDL verglichen werden. Zusätzlich soll untersucht werden, ob sich die Architektur einfach um übliche Bausteine, wie eine Pipeline oder einen Cache, erweitern lässt.

Im Einzelnen sind hierbei folgende Aufgaben und Schritte auszuführen:

- Einarbeitung in das Gebiet der One Instruction Set Computer (nützliche Literatur: [1, 2, 3])
- Einarbeitung in die HDL Clash (nützliche Literatur: [4, 5] sowie die offizielle Website http://www.Clash-lang.org)
- Einarbeitung in andere moderne Hardwarebeschreibungssprachen (z.B. Chisel [6], Amaranth [7], myHDL [8], eventuell SystemC [9])

- Vergleich der Möglichkeiten und Anwendungsgebiete dieser Sprachen
- Implementierung des mOISC-Prozessors aus [3] in Clash
- Erläuterung etwaiger Abweichungen in der Architektur im Vergleich zum mOISC (beide Architekturen müssen zyklenakkurat identische Ausgaben erzeugen.)
- (Erweiterung des Prozessors um z.B. eine Pipeline oder einen Cache)
- Demonstration der Lauffähigkeit des Prozessors auf einem FPGA
- Vergleich der Ausdruckskraft des Clash-Codes mit dem VHDL-Code von [3]
- Vergleich der Synthese-Ergebnisse bezüglich Ressourcenverbrauch mit [3]
- (Vergleich von Programmlaufzeiten in Takten mit [3])
- Erstellung einer ausführlichen Dokumentation (inklusive Testaufrufe, Anleitung zum Flashen etc.) sowie ausführlicher Tests des Prozessors
- Bereitstellung des erstellten Quellcodes, so dass eine einfache Weiterverwendung möglich ist
- Zusammenschrift der Arbeit

Die Arbeit wird in englischer Sprache verfasst.

Literatur der Aufgabenstellung

1. Phillip Laplante und William Gilreath. “One Instruction Set Computers for Image Processing”. In: Journal of VLSI signal processing systems for signal, image and video technology 38.1 (1. Aug. 2004), S. 45-61.
2. Nektarios Georgios Tsoutsos und Michail Maniatakos. “Investigating the Application of One Instruction Set Computing for Encrypted Data Computation”. In: Security, Privacy, and Applied Cryptography Engineering. Hrsg. von Benedikt Gierlichs, Sylvain Guilley und Debdeep Mukhopadhyay. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013, S. 21-37.
3. Marco Crepaldi, Andrea Merello und Mirco Di Salvo. “A Multi-One Instruction Set Computer for Microcontroller Applications”. In: IEEE Access 9 (2021), S. 113454-113474.
4. Christiaan Baaij u. a. “Clash: Structural Descriptions of Synchronous Hardware Using Haskell”. In: Euromicro Conference on Digital System Design. Sep. 2010, S. 714-721.

5. Gergö Érdi. Retrocomputing with Clash: Haskell for FPGA Hardware Design. 12. Sep. 2021. 540 S.

6. Martin Schoeberl. Digital Design with Chisel. Seattle: Kindle Direct Publishing, 30. Aug. 2019. 220 S.

7. Amaranth HDL (Previously nMigen). Version 0.3. Amaranth HDL, 16. Dez. 2021.

8. MyHDL. Version 0.11. 1. Juni 2019.

9. David C. Black u. a. SystemC: From the Ground Up, Second Edition: From the Ground up. With download-code. 2nd ed. 2010 Edition. New York: Springer, 30. Dez. 2009. 304 S.

Contents

1 Introduction

The times in which embedded systems ran with a few hundred bytes on a hand-drawn microarchitecture, have been over since the mid-1980s [34]. At that time the most famous hardware description languages VHDL (Very High Speed Integrated Circuit Hardware Description Language) [16] and Verilog [15] were developed.

However, since that time the requirements for hardware development have evolved. Current trends in silicone processing point to an end of Moore's law. The disadvantages of significant generalization in processor design also point to changes in the design flow [33]. This shift from generality to specification is visible in the number of application-specific integrated circuits (ASICs) that have been developed in recent years. Google's tensor processing unit, chips for neural networks [26] and ASCIs for mining cryptocurrency like bitcoin show this development [3]. Together with ever-increasing demands for security and reusability, this has led to a paradigm shift in the hardware design landscape [21]. The goal of modern hardware description languages is to provide abstractions that promote reuse, security [21], correctness and performance. Over the years a variety of languages have been developed to meet these requirements.

In this thesis the Hardware desciption language Clash, a synthesizable subset of Haskell is used to reimplement a multi One Instruction Set Computer (mOISC) [11] in a functional style. At first, the related work in the field of Hardware description languages and One Instruction Set Computers is presented in chapter 2. In chapter 3 the mOISC is reimplemented in Clash while the differences between the classic HDLs and Clash are discussed. In chapter 4 different optimizations are presented and discussed. In chapter 5 the verification and testing of the mOISC in VHDL and Haskell are discussed. In chapter 6 the speedup of the implementation-process of the mOISC is discussed, and the differences between the VHDL and Clash implementation are presented. Finally in chapter 7 the results of the thesis are summarized.

2 Related Work

2.1 Modern Hardware Description Languages

As pointed out by turing-award winner Dave Patterson, Moore's law is comming to an end. The New York time declared it dead with the release of Google's Tensor Processing Unit(TPU) [1]. Combined with inefficiencies of general purpose processors this hints in the direction of ever more domain specific processors. As conluded by Lenny Troung [33], this signals a shift in the development of modern hardware as domain specific chips lead to ever smaller development teams and shorter development cycles. From this, a variety of new hardware languages will emerge, with developers searching for new methods to reduce time and development costs. The goal of those languages is to provide abstractions that allow for reuse, security [21], correctness and performance. Over the years a variety of languages have been developed. These can be separated into three categories:

- Term Rewriting Systems (TRS)
- High-Level Synthesis
- Functional HDLs

In Term Rewriting Systems, circuits are described as a set of rewrite rules, applied to inputs and state values. They model non-deterministic behavior and concurrency. Important examples are Bluespec and Bluespec Haskell [25]. Example code can be seen in figure 2.1. In high-level-synthesis (HLS) such as SystemC or VivadoHLS, the circuit is desrcibed as an ordinary C program which is then translated into a hardware description language. HLS compilers need to be able to translate a programm, unbound in time and space to a finite set of resources [33]. Development times can be reduced when taking the tradeoff of unefficiently used resources, so these language are more fitted for prototyping. Functional HDLs use the idea that a pure function can be used to model combinatoric logic [33]. As early as 1976, Friedman and Wise promoted purely applicative hardware desciption styles, which have been worked into the language Daisy [18]. More modern examples include Chisel, Lava and Clash. Embedded languages are an interesting approach as the toolchain, like parsers or type checkers, do not need to be implemented. However primitives used by these languages do not represent circuits working on signals, but a graph translated by an embedded circuit compiler to run simulation or synthesis [2]. While Chisel [5] and Lava [19] are embedded domain specific languages (DSL), Clash takes this even further, as it is a fully translatable subset of Haskell [32]. This allows most language features to be used while still being able to compile to hardware. Even

```
mkTop :: Module Empty
mkTop = module
    rules
        "rl_print_answer": when True ==> do
            $display "\n\n***** Deep Thought says: Hello, World!
             ↪  *****"
            $display "      And the answer is: %0d (or, in hex:
             ↪  0x%0h)\n"  42  42
            $finish
```

Figure 2.1: A Term rewriting example form bluespec Haskell [25]

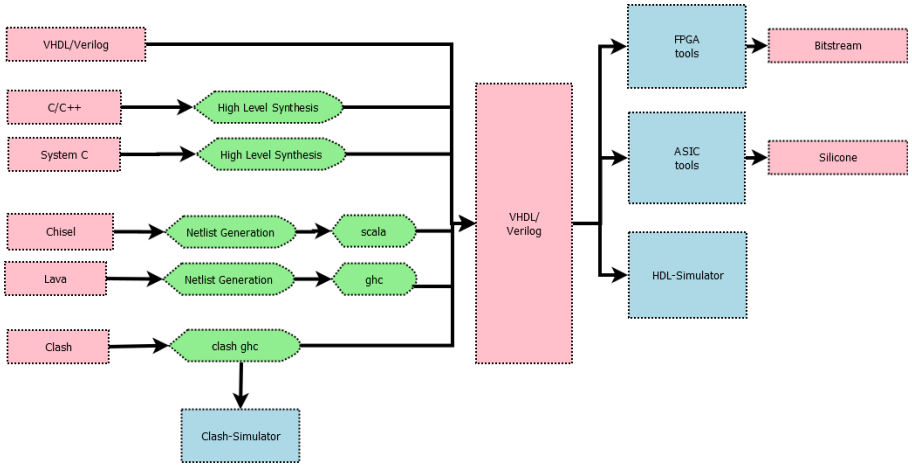


Figure 2.2: Compilation techniques used by HDLs

advanced features like polymorphic types and type-classes can be used. An overview of the compilation techniques used by these HDLs can be seen in figure 2.2.

2.2 mOISC - Multi One Instruction Set Computer

One Instruction Set Computers (OISCs) are a popular example to explain computer architecture. OISCs can be Turing-complete with a single instruction and can solve any computing problem [11]. This minimalist approach, the combined flow of data and control in a single instruction, can also be extended to cryptography [23]. Another interesting use case is the usage of OISCs in image processing [20]. One of the best known one instruction sets is `subleq` - SUBstract and jump if the result is Less or EQual to zero [22]. `subleq` is not the only possible instruction set. Transport Triggered Architecture (TTA) based on the `move` instruction is another possibility. Here operations are executed depending on the address. TTAs were already used commercially, like with the MaxQ processor,

which executes depending on index, addition, subraction, and other functions [4, Chapter 17]. However, a processor based on `subleq` solely has not yet been marketed. This is due to the reduced performance, the large memory requirements and the resulting high energy consumption [11].

However, since One Instruction Set architectures are very easy to implement and require only a small logic budget, M. Crepaldi et. Al. have developed a so-called mOISC, a multi One Instruction Set Computer, which can process 14 different One Instruction Sets. It can be set to either single instruction mode or use a mix of the different instructions. The computer has a very low consumption, both in logic in FPGA fabric, as well as in energy. This makes it the perfect example for a reimplementation, since a functioning processor has already been implemented in VHDL. Since clash can compile directly to VHDL this makes it perfect for a comparison. In their conclusion M. Crepaldi et al. [11] have already provided some insights to improve the processor which will be discussed in chapter 4.

2.2.1 Architecture

The computer is based on an absolutely addressable von Neumann architecture, where one memory contains both program flow and data. The registers are mapped to the addresses 0x00 to 0x07. Instructions have the format `instr a, b -> c`. Where `a` is the source address, `b` is the second source and destination address for the result of the operation and `c` is the jump address. Depending on the mode (machine code register - MCR) these instructions are stored in blocks of 4 (Complex Instruction Set Computer mode - CISC) or blocks of 3 (OSIC-mode) bytes in memory. An overview of the different instruction sets can be found in the apendix at page 39 in figure 7.2. The first 8 addresses are reserved for registers allowing Transport Triggered Architectures (TTA). The registers can be found in the appendix on page 39 in figure 7.3.

2.2.2 Block Design

The reference implementation was designed on the basis of a Cyclone-10-LP FPGA in VHDL. The included blocks for PLL, IO buffer and memory are mega-functions provided by Intel-Quartus.

An external 50 MHz clock signal is connected to an internal clock multiplexer. This outputs the frequencies 100 MHz, 50 MHz, 1 MHz or 10 KHz depending on the value of the internal clock speed register (CSR). The power consumption can be controlled by the different internal clock speeds. The CLK signal is forwarded to all internal systems together with the reset signal:

- The Arithmetic Logic Unit (ALU), which executes the various instructions and calculates the flags for overflow, and cmp, the latter being partially responsible for the control-flow within the state-machine.
- The Finite State-Machine (FSM) is responsible for the control-flow of the processor and orchestrates the other components.

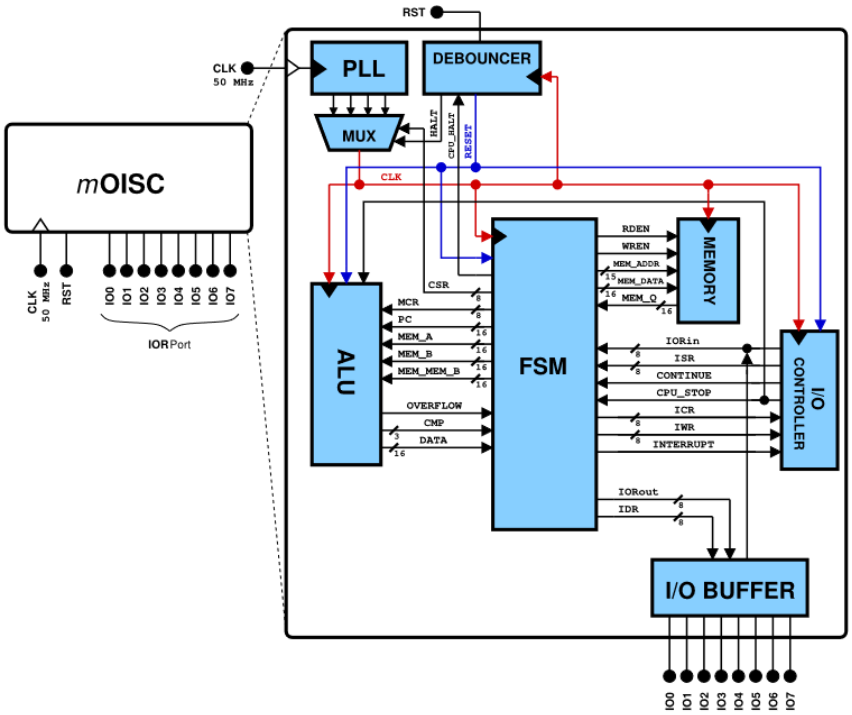


Figure 2.3: Block design of the MOISC - from [11]

- The IO Controller is reponsible for interrupt handling.
- The IO-Buffer is responsible for the IO-ports of the processor.
- The Memory saves instructions and data as this is a von Neumann architecture.
- The PLL/Clock provides four different clock signals depending on the processors `csr` - clockspeedregister.

The internal workings of these parts are described in the following sections, while the processor is being translated into Clash.

3 From classic HDLs to Clash

3.1 Modeling in Clash

Clash provides a variety of basic functions and types needed to represent basic hardware structures. A simple 8-bit adder can be represented like this:

```
adder:: Signal System (Signed 8) -> Signal System (Signed 8) ->
↪  Signal System (Signed 8)
adder a b = (+) <$> a <*> b
```

- `Signal`: An applicative functor representing an infinitely long stream of values. Applicative Functors allow the sequencing of functorial operations. This allows `Signal`s to encapsulate data, while still being able to define operations on it [24].
- `System`: The domain of a signal. The domain contains clock-period, reset-kind and other configurations.
- `Signed n`: A signed integer of `n` bits.
- `<$>`: Functor-Map inline representation
  `fmap :: (a -> b) -> f a -> f b.`
- `<*>`: Sequential-Application
  `(<*>) :: f (a -> b) -> f a -> f b.`

It follows that `adder` maps `(+)`, a function taking two `Signed n`, to the two `Signal`s wrapping the result in the same type. We can now write a simple testbench for this adder:

```
add0to9 :: [Signed 8]
add0to9 = sampleN 10 $ adder (fromList [0..9]) (fromList [0..9])
```

This will add the numbers 0 to 9 to themselves and return the results in a List. When executed in `Clashi`, the Clash interactive environment, it will yield the following result:

```
Clash.Prelude> add0to9
[0,2,4,6,8,10,12,14,16,18]
```

Since most programms do contain some type of state Clash offers the `register` function to model this. A simple 3-bit counter can be modeled like this:

```
counter :: HiddenClockResetEnable dom => Signal dom (Unsigned 3)
counter = register 0 ((+1) <$> counter)
```

The type Signature of register is `register :: a -> Signal dom a -> Signal dom a`. The first argument is the initial value, the second argument is the signal that is used to update the `register`. `register` needs a domain providing a clock a reset and an enable signal. This is the reason for the hidden type class constraint. Since `register`s have an intial value, they require a reset input. To be simulated the constraint has to be in the declaration. If it was not, the type would be less general than expected because of the monomorphism restriction [8].

```
sampleCounter :: [Unsigned 3]
sampleCounter = sampleN 12 $ withClockResetEnable (clockGen @System)
↪  (resetGen @System) (enableGen @System)  counter
```

This will sample the counter 12 times with a clock, reset and enable signal. The first output is zero, since the first first simulated timestep is before the first clock cycle [13, Chapter 4]. Since we are using unsigned numbers we can easily see the overflow of the counter.

```
Clash.Prelude> sampleCounter
[0,0,1,2,3,4,5,6,7,0]
```

This counter can now be translated into VHDL and run on an FPGA. For this one needs to define a TopEntity which defines additional needed Signals:

```
topEntity :: Clock System -> Reset System -> Enable System -> Signal
↪  System (Unsigned 3)
topEntity clk rst en = withClockResetEnable clk rst en counter
```

This will generate the following VHDL entity with comments provided by Clash showing some of the important structures:

```
entity topEntity is
  port(-- clock
       clk    : in Example_StartInClash_topEntity_types.clk_System;
       -- reset
       rst    : in Example_StartInClash_topEntity_types.rst_System;
       -- enable
       en     : in Example_StartInClash_topEntity_types.en_System;
       result : out unsigned(2 downto 0));
end;
architecture structural of topEntity is
  signal result_1 : unsigned(2 downto 0) := to_unsigned(0,3);
begin
  -- register begin
  result_1_register : process(clk,rst)
  begin
    if rst =  '1'  then
```

```
      result_1 <= to_unsigned(0,3);
    elsif rising_edge(clk) then
      if en then
        result_1 <= (result_1 + to_unsigned(1,3));
      end if;
    end if;
  end process;
  -- register end
  result <= result_1;
end;
```

The result can be imported into Vivado, connected to the LEDs on an FPGA and run. With these tools and other later explained concepts, it is possible to model a processor in Clash.

3.2 Comparison in Vivado

During the reimplementation it was very important to always have a verifiable model. Therefore, the first step of the translation was a cycle-accurate translation of the VHDL code by M. Crepaldi et. al into Clash [10]. The verification was performed during this process using VHDL testbenches. The goal was to have a processor developed in Clash that accurately matches the reference. In the next step this can be used with the Clash internal testbench for further development. For this purpose, the code, originally developed for Intel Quartus was translated into Xilinx Vivado compatible VHDL. A listing of changes can be found in the appendix at page 37.

3.3 Arithmetic Logic Unit

The first step in the translation was to model the Arithmetic Logic Unit (ALU). It is used to perform arithmetic operations on the data. Haskell is a strongly typed language. Therefore it is recommended to start with the type declarations. Since the ALU does not have a state, it can be modeled as a function without Signals. A first type declaration of the ALU looks like this:

```
alu :: forall memSize.
  (KnownNat memSize) =>
  Instruction ->         -- Machine code register
  Signed memSize ->      -- Memory at address a
  Signed memSize ->      -- Memory at address b
  Signed memSize ->      -- Memory at address pointed to by memory
↪   at address b
  BitVector memSize ->   -- Program counter for Pogram Counter set
  (
```

```
    Signed memSize, -- Data Result
    CompareResult,  -- Compare Result
    Bool            -- overflow
  )
```

This can be compared to the port declaration in VHDL. The reusability of the Clash code is directly visible in the type constraint `forall memSize.(KnownNat memSize)`. This allows arbitrary word lengths, for example, on a 42 bit architecture. The Haskell compiler will check if all incoming and outgoing variables have the same bit length. While the data types `Signed` and `Unsigned` correspond to their VHDL counterpart, `Instruction` and `CompareResult` show that Clash allows to define arbitrary types. The definition of Instruction looks like this:

```
data Instruction
    = SubLeq -- SUBtract and jump if Less or EQual
    | MovLeq -- MOVe and jump if Less or EQual
    | AddLeq -- ADD and jump if Less or EQual
    ...
    | PcS -- Program Counter Set
  deriving (Show, Generic, NFDataX, Lift, Eq)
```

All the instructions with their respective control flow can be found in the apendix on page 39 in figure 7.2. Here NFDataX describes a type which can contain an undefined value [9]. For the sake of clarity, the incoming and outgoing signals have also been bundled into data types according to their origin and destination. The result of the ALU, which will be sent back to the Finite State Machine is defined as follows:

```
  data Alu2Fsm memSize = Alu2Fsm
  { _data_res :: Signed memSize,
    _cmp :: CompareResult,
    _overflow :: Bool
  }
  deriving (Show, Generic, NFDataX)
```

With `memSize` again being a varibale length. Using the new types the declaration of the ALU can be changed to:

```
alu :: forall memSize. (KnownNat memSize) =>
    Fsm2Alu memSize -> Alu2Fsm memSize
```

`Fsm2Alu memSize` contains the described inputs and `Alu2Fsm memSize` the outputs.
For the flow of the ALU we use Haskell's `case` statement over `Instruction`.

```
alu Fsm2Alu {..} = case mcr of
    SubLeq -> Alu2Fsm data' cmp' overflow'
        where
        data' = mem_b - mem_a
```

```
        cmp' = cmpleq
        -- msb = most significant bit
        overflow' = (complement (msb data') .&. msb mem_a .&. msb
↪   mem_b) .|. (msb data' .&. complement (msb mem_a) .&. complement
↪   (msb mem_b)) == high
    MovLeq -> Alu2Fsm data' CMP_NONE False
        where
        data' = mem_a
    ShlLeq -> Alu2Fsm data' CMP_NONE False
        where
        data' = shiftL mem_b shftAmt
        shftAmt = fromInteger . toInteger $ pack mem_a
    AndLeq -> Alu2Fsm data' CMP_NONE False
        where
            data' = (.&.) mem_a mem_b
    ...
```

Only `AddLeq` and `SubLeq` can conditionally change `Overflow` and `Compare`. The remaining instructions always return the default values `CMP_NONE` and `Overflow=False`. Defining an overflow on logical operations like AND would be confusing for the user.

We use wildcard patterns for cleaner looking code:
`Fsm2Alu {..}` ≡ `Fsm2Alu mcr pc mem_a mem_b mem_mem_b`.

This signature describes a function that takes a `Fsm2Alu memSize` and outputs a `Alu2Fsm memSize`. We can thus take above signature and then encapsulate it in `Signal`s using Functor-Map. This way we can have much cleaner looking code avoiding to use the functor syntax every time.

```
continuousAlu :: forall memSize dom .
    (KnownNat memSize, HiddenClockResetEnable dom) =>
    Signal dom (Fsm2Alu memSize) -> Signal dom (Alu2Fsm memSize)
continuousAlu x = alu <$> x
```

One can synthesize this function but it is recommended to use a topEntity wrapper-function in which eventual type conversions are performed. For example to compare the generated VHDL with the original VHDL, the overflow signal needs to be converted from `Boolean` to `Bit`. This will result in a `std_logic` instead of a `boolean` in the VHDL synthesis. One can also use the `Synthesize` annotation to get more meaningful port-names (instead of eg. `cargs_1`). The `Synthesize` annotation even supports bundled datatypes by providing the `PortProduct` construct. The annotation of the `aluTopEntity` function is:

```
  {-# ANN topEntityAlu ( Synthesize
      { t_name = "ALU",
        t_inputs =
          [ PortProduct
```

```
>>> stack run Clash -- src/StandardEntities/Alu.hs -main-is
 ↪  topEntityAlu  --vhdl -fClash-no-escaped-identifiers -isrc>>
GHC: Setting up GHC took: 0.699s
GHC: Compiling and loading modules took: 9.563s
Clash: Parsing and compiling primitives took 0.148s
GHC+Clash: Loading modules cumulatively took 13.772s
Clash: Compiling StandardEntities.Alu.topEntityAlu
Clash: Normalization took 0.027s
Clash: Netlist generation took 0.004s
Clash: Compiling StandardEntities.Alu.topEntityAlu took 0.036s
Clash: Total compilation took 13.810s
```

Figure 3.1: Translation of the ALU to VHDL

```
          "in"
          [ PortName "mcr",
            PortName "pc",
            PortName "mem_a",
            PortName "mem_b",
            PortName "mem_mem_b"]],
    t_output =
      PortProduct
        "out"
        [ PortName "overflow",
          PortName "cmp",
          PortName "data"]
  }) #-}
```

Since this function is now discrete over `Signal`s and named it can be converted into Verilog, SystemVerilog or VHDL as seen in figure 3.1. The resulting code is not very readable but can be verified. It can be imported into Vivado and exported as an IP-Block for further use as can be seen in Figure 3.2. For IP-Block creation all top-level port declarations in VHDL need to be replaced by their type in the generated "types.vhdl" file (e.g. `std_logic` instead of `alu_types.clk_xilinxsystem`). For more information on IP-Block creation see [17].

3.4 mOISC Finite State Machine

The mOISCs Finite State Machine(FSM) is a simple mealy machine, used to orchestrate the other components. Since the FSM contains an internal State, one could use registers. However, Clash provides a more convenient way to do this - the `mealy` function [6]:

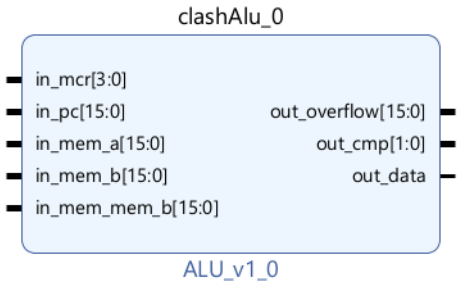


Figure 3.2: Alu-block in Vivado

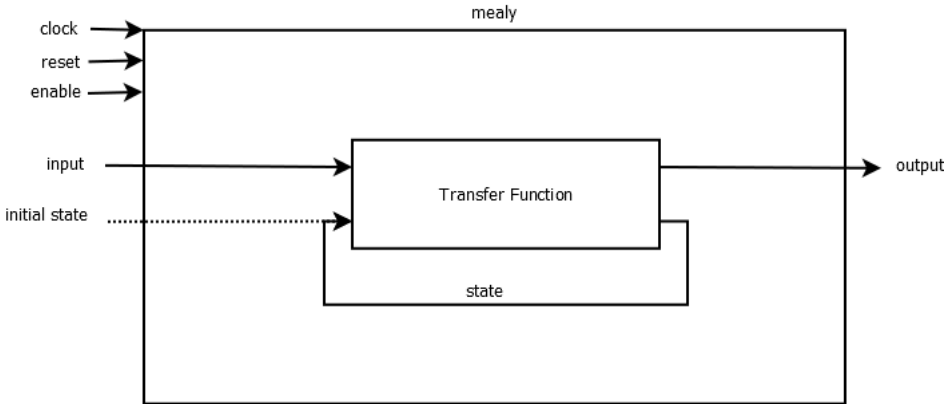


Figure 3.3: Mealy function

```
mealy :: (KnownDomain dom, NFDataX s)
    => Clock dom -- Clock Input
    -> Reset dom -- Reset Input (changes to initial state)
    -> Enable dom -- Enable Input
    -> (s -> i -> (s, o)) -- transfer function: state -> input ->
↪   (next state, output)
    -> s -- Initial state
    -> Signal dom i -- Input Signal
    -> Signal dom o  -- Output Signal
```

To define the FSM, we need to define the transfer function and thus input, output and state. The mealy function is pictured in figure 3.3.

```
fsmcycle ::
    forall memSize regSize.
    (KnownNat memSize, KnownNat regSize) =>
    FsmInternalState memSize regSize ->
    FsmInput memSize regSize ->
    (FsmInternalState memSize regSize, FsmOutput memSize regSize)
```

The FsmInternalState type represents the internal state of the FSM. It contains the mealy-state, registers and current outptut. The current output is saved in the internal state, so it only needs to be updated and not recalculated as a whole when the state changes. For a full overview of all states, see figure 7.4 in the appendix on page 39 . Haskells case statement is used again - this time over the FsmState type. By using record syntax we can updated only changed fields of the internal state:

```
fsmcycle fsmState fsmInput =
    case _cycleState fsmState of
    FETCH_MCR -> (fsmState', outputRegisters')
      where
        fsmState' =
          fsmState
            { _cycleState = cycleState',
              _outputRegisters = outputRegisters'
            }
        outputRegisters' =
          (_outputRegisters fsmState)
            { _memOutput = memOutput'
            }
        cycleState' = FETCH_0_SAVE_MCR
        memOutput' = readMemFromAddress $ _pci (_internalRegisters
↪   fsmState)
```

The last step in the cyclic accurate fsm implementation is to adjust the timing of the state changes to that of the original implementation. The original implementation has a state change every two cycles updating on the first of them. This is done by extending the output by one cycle and then registering the output for one cycle:

```
extendOne state input flag = case flag of
      True -> (state, _outputRegisters state, False)
      False -> (state', output', True)
        where
          (state', output') = fsmcycle state input
result = register (resetState, resetOutput, True) calc
calc = extendOne <$> (first <$>  result) <*> fsmInput <*>  (third
↪   <$> result)
```

After annotating and adding a topModule the code is compiled and imported to Vivado. We now have a working internal structure. The next step is to connect the internal structure to the external world.

3.5 IO-Buffer

The IO-Buffer consists of an inout-signal directed by the `oe` input, connected to the `idr`(Inout direction) register of the FSM. Clash provides a bidirectional signal. A function

taking this Signal needs to provide the following type declaration:

```
biSignalFunction::
    BiSignalIn ds dom n ->
    BiSignalOut ds dom n
```

Here ds describes the default behaviour of the signal. The type of ds is BiSignalDefault. It can be either PullUp, PullDown or Floating. dom is the domain of the signal and n is its size. BiSignal provides the functions readFromBiSignal and writeToBiSignal. These functions can only read or write the full signal. Since the IO-Buffer needs to read and write on individual Bits at the same time this is not possible with the standart BiSignal functions. The next approach was to use a Vec of BiSignals.

```
ioBuffer ::
  forall dom regSize.
  (HiddenClockResetEnable dom, KnownNat regSize) =>
  Signal dom (BitVector regSize) -> -- Input from FSM
  Signal dom (BitVector regSize) -> -- Input Direction
  Vec regSize (BiSignalIn 'Floating dom 1) -> -- Input from Outside
  (Signal dom (BitVector regSize), Vec regSize (BiSignalOut
↪  'Floating dom 1)) -- (Output to FSM, Output to Outside)
```

This approach does not work either since Clash can not look trough the Vec. Handling this case would be quite tricky for Clash as it would have to infere an inout [7:0] automatically as Gergő Érdi, the author of Retroprogramming in Clash [13] pointed out to me in personal correspondence.

However, Clash provides the concept of Blackboxes. A Blackbox is a function with HDL template code for synthesis, which Clash will be unable to simulate. A Blackbox needs to define a simulation function and a Template in either VHDL, Verilog or SystemVerilog. The simulation function can also be left undefined if the Blackbox is not used in the simulation, as is the case for this IO-Buffer. In the simulation we want to have seperate Signals for the input and output of the IO-Buffer for easier testing. The IO-Buffer has the following type declaration:

```
  myBlackBox ::
  Signal dom (BitVector 8) -> -- Input from FSm
  Signal dom (BitVector 8) -> -- Input Direction
  BiSignalIn 'Floating dom 8 -> -- Inout
  (Signal dom (BitVector 8), BiSignalOut 'Floating dom 8) -- (Output
↪  to FSM, Inout to Outside)
myBlackBox _ _ = undefined
```

The BlackBox is annotated as follows:

```
  {-# NOINLINE myBlackBox #-}
  {-# ANN myBlackBox (InlinePrimitive [VHDL] £ unindent [i|
```

```
  [ { "BlackBox" :
      { "name" : "CyclicAccurate.IoBuffer.myBlackBox",
        "kind" : "Declaration",
        "libraries" : ["UNISIM"],
        "imports" : ["UNISIM.vcomponents.all"],
        "template" :
" ~GENSYM[myBlackBox][0] : FOR i IN 0 TO 7 GENERATE
  IOBUF_inst : IOBUF
    generic map (DRIVE => 12)
    port map (
      O => ~RESULT(i),      -- Buffer output
      IO => ~ARG[2](i),    -- Buffer inout port
      I => ~ARG[0](i),      -- Buffer input
      T => not ~ARG[1](i)  -- high=input, low=output
    );
  END GENERATE;
  "}}] |]) #-}
```

NOINLINE is used to prevent the Blackbox from being optimized out by the compiler. libraries define the libraries described by `import ...;` in VHDL. imports define the packages described by `use ...;` in VHDL. ~ARG[i] describes the ith input of the Haskell function, while ~RESULT describes the output. The inout Port is described as input-argument since Clash only uses the BiSignalIn part when compiling to HDL. A Problem with this approach is, that IOBUF from the UNISIM library is Xilinx specific. So for making the code usable for other HDL-Tools one would have to provide differend implementations of the IOBUF primitive. After providing toplevel synthesis annotations for naming the ports, the output is as expected providing the following VHDL entity:

```
entity IoBuffer is
  port(fsm_in     : in std_logic_vector(7 downto 0);
       dir        : in std_logic_vector(7 downto 0);
       outside_io : inout std_logic_vector(7 downto 0);
       to_fsm     : out std_logic_vector(7 downto 0));
end;
architecture structural of IoBuffer is
begin
  myblackbox : FOR i IN 0 TO 7 GENERATE
    IOBUF_inst : IOBUF
      generic map (DRIVE => 12)
      port map (
        O => to_fsm(i),      -- Buffer output
        IO => outside_io(i),   -- Buffer inout port (connect
↪  directly to top-level port)
        I => fsm_in(i),      -- Buffer input
```

```
        T => not dir(i)       -- 3-state enable input, high=input,
↪   low=output
      );
  END GENERATE;
end;
```

The whole output can be imported into Vivado and made into a block like the FSM and ALU. The processor is now connected to the outside world, but it has no infromation what to execute. To load and save data, the processor needs to be able to access memory.

3.6 Memory

Clash provides different approaches to implement memory. The simplest approach is to use a `Vec` of `BitVector`s and store them in the state:

```
newtype RAM wordSize ramSize = RAM (Vec ramSize (BitVector
↪   wordSize))
    deriving (Show, Generic, NFDataX)
readRam::forall n.(KnownNat n) => RAM -> BitVector n -> BitVector n
readRam (RAM content) addr = content !! addr
writeRam ::forall n.(KnownNat n) => RAM -> BitVector n -> Bitvector
↪   n -> RAM
writeRam (RAM content) addr word = RAM (replace addr word content)
```

This will use a tremendous amount of lookup tables (LUT) and will not result in the most efficient hardware structure, especially when designing ASICs [7]. Clash also provides the `asyncRam` function, which has the potential to be more efficiently, but will still get translated to LUTs on FPGAs. A more efficient approach is to use a `blockRam`. It comes with the disadvantage of being synchronous, meaning memory requested at time t will only be available at time $t+1$. However since block rams are an FPGA primitive, this allows the creation of a memory of up to $270KB$ on the zybo z10 or $630KB$ on the zybo z20 FPGA [12]. Since initialization of the memory should be done from a binary file compiled by the reference compiler, the `blockRamFilePow2` primitive is chosen. It provides the following type declaration:

```
blockRamFilePow2 ::
  FilePath -- File with the initial content
  -> Signal dom (Unsigned n) -- Read address
  -- Size of the BRam is n^2
  -> Signal dom (Maybe (Unsigned n, BitVector m))
  -- (write address, value)
  -> Signal dom (BitVector m) -- value requested at t-1
```

The conditional write is provided using the `Maybe` monad. If the value is `Nothing`, the memory is not written. When a `Just (Unsigned n, BitVector m)` is provided, the

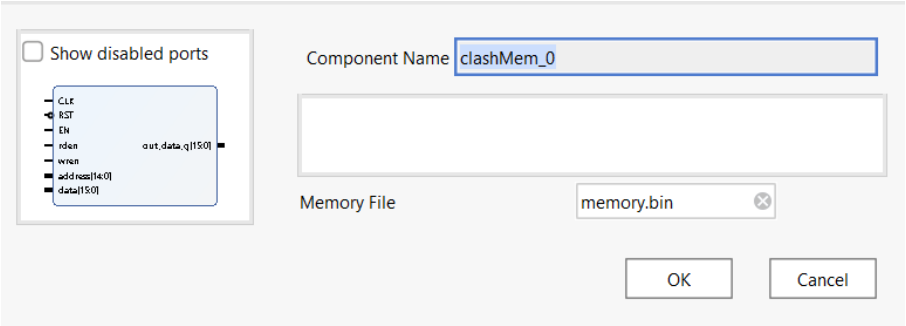


Figure 3.4: Initialization of the memory in Vivado with an arbitrary file

memory is written at the given address. Since this implementation does use a readEnable and a writeEnable signal, a wrapper around the memory is needed to convert the results:

```
  rdEnwrEnRam file rEn wEn addr dataWr = result
  where
    result = f $ blockRamFilePow2 file (unpack <$> addr)
↪   (convertWriteEnablemaybe <$> wEn <*> addr <*> dataWr)
    f a = mux (register False rEn) a 0
```

The `mux` function, representing a multiplexer, is used since the first value of the blockRam is undefined and all other values depend on readEnable. After compiling to VHDL the file "memory.bin" was replaced by `generic (memory_file : string)`, so during block creation in vivado one can then import arbitrary memory files as can be seen in figure 3.4.

3.7 IO-Controller

The IO-Controller is a simple FSM that is used to control the IO-buffer and interrupts. Since it does not have as many states as the fsm, a graph can be found in figure 3.5. Again the implementation is done using the `mealy` function. The type of the transfer function is as follows:

```
ioCtrCycle ::
  forall n dom.
  (HiddenClockResetEnable dom, KnownNat n) =>
  (IoCtrState, IoCtrlRegisters n, Ioctr2Fsm n) -> -- Internal State
  (Fsm2Ioctr n, IoBuf2Ioctr n) -> -- Input
  ((IoCtrState, IoCtrlRegisters n, Ioctr2Fsm n), Ioctr2Fsm n) --
↪   (nextState,output)
```

A sepcial feature of this transfer function is, that it takes two inputs, one from the FSM and one from the IO-Buffer.

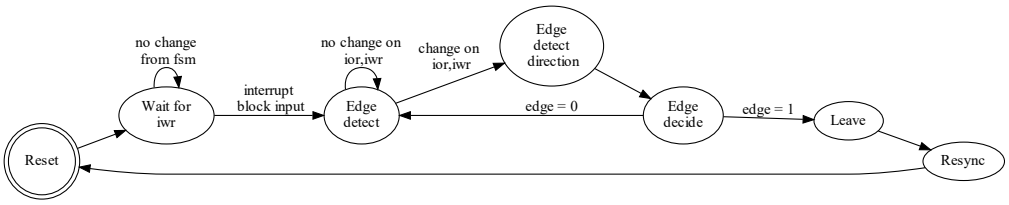


Figure 3.5: IO-Controller State Diagram

3.8 Clock Generator

Since the mOISC's speed can be regulated by the `csr` register, a way to change the speed is needed. Clash does not offer the possibility to change input clock, as there is no way to simulate this, since signals are assigned to a specific domain containing only one clock. A way to cheat this is to use a blackbox taking in the different clock domains and outputting one clock on a single domain. The Clash function is again `undefined`, since for simulation a single clock is enough and the `csr` register can be controlled directly. To create a new clock domain, the `createDomain` function is used:

```
createDomain vSystem{vName="Dom10Khz", vPeriod=10000}
```

The type annotation contains the same domain for `csr`, `cpu_stop` and `clk_out`, since they are connected by external functions and cannot be compiled together if not. The VHDL Blackbox then only takes the input clocks and outputs the new clock depending on csr:

```
~GENSYM[clkSel][0]: process(~ARG[2], ~ARG[3], ~ARG[4], ~ARG[5],
↪  ~ARG[0], ~ARG[1])
  begin
    case ~ARG[0] is
      when \\\"00000000\\\" =>
          ~RESULT <= ~ARG[2] and not ~ARG[1];
    ...
```

The `\\\"` is needed to double escape - first the Json-parser, then the Haskell compiler, resulting in a single `"` to describe a `std_logic_vector`.

3.9 Complete mOISC

Since all blocks are now implemented, the complete mOISC can be created in two ways. Either by creating a `topEntity` in Clash and compiling it to VHDL, or by connecting the blocks in Vivado. The connected Vivado design can be seen in figure 3.6. This allows the replacement of blocks with other implementations, for example the IO-Buffer with a different one.

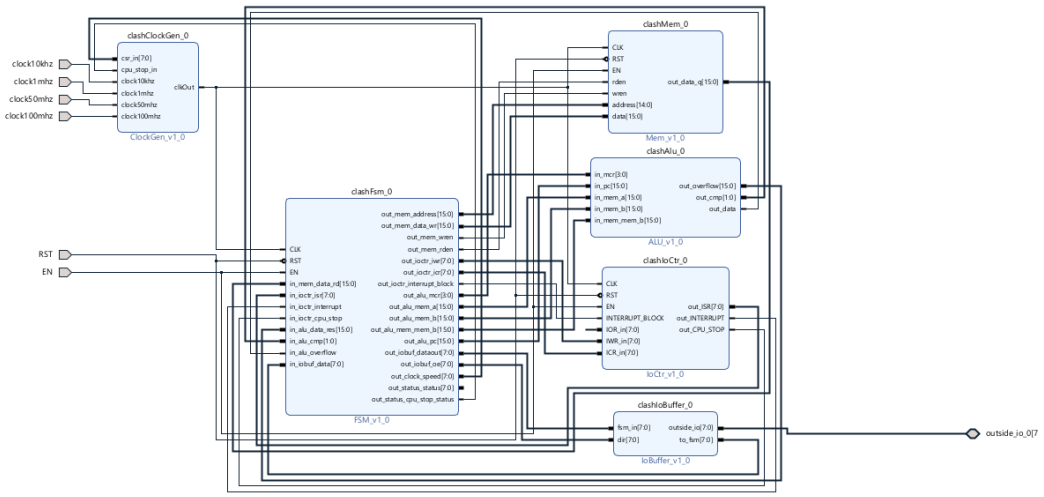


Figure 3.6: Complete mOISC in Vivado

The Clash implementation follows the same configurable principle. The `mOISC` is defined as follows:

```
mosicSimpleFsm :: FilePath
  -> Clock Dom10Khz
  -> Clock Dom1Mhz
  -> Clock Dom50Mhz
  -> Clock Dom100Mhz
  -> Reset XilinxSystem
  -> Enable XilinxSystem
  -> BiSignalIn 'Floating XilinxSystem 8 -- in
  -> BiSignalOut 'Floating XilinxSystem 8 -- (enable_out, out)
```

The filepath is left as an argument, so that the mOISC can be initialized with different memory files. Sadly the pure Clash implementation does not compile to VHDL, since the `undefined` functions in the `clockGen` result in no VHDL being produced. No workaround was found. The implementation is then just hooking up the different blocks and connecting the signals: In both cases the clocks have to be defined in Vivado. The implementation is then ready to be run on the FPGA. The first test was made with a simple C program blinking a LED:

```
// disable optimizations to prevent skipping the sleep cycles
void __attribute__((optimize("O0"))) sleep(int cycle){
    for (int count = 0; count < cycle; count++){
        #pragma no_op
  }
}
int main(){
    int number = 0;
```

```
    *mOISC_idr = 0xff;
    // IO Read Register to 0xFF to set all Bits as Output
    *mOISC_csr = 0;
    // Clock Speed Register to 192 to set the Clock Speed to 10 KHZ
    while(1){
        number++;
        if(number >= 256){
            number = 0;
        }
        *mOISC_ior = number;
        sleep(500);
    }
}
```

In the `sleep` function the `#pragma no_op` is used to prevent the compiler from skipping the cycles. Each loop takes at least 3 instructions. With average 8 cycles per instructions and the two clock fsm this will average to 50 cycles per loop. Based on the clockspeed of 10 kHz the call will take to about 2.5 second to complete. After synthesising the design it should have been able to run on the FPGA. Sadly the design did not synthesize either since clash produced different internal signal structures which are not compileable by Vivado.

4 Optimizations

4.1 Functional Approach

Clash, which is based on Haskell supports monads and other functional programming techniques. This can be used to improve the code. For the average programmer, monads are often a ivory tower concept. With sentences like: ”monads are just monoids in the category of endofunctors, what’s the problem?” [27] they seem to be very abstract. In practice, monads are just a way to structure code by chaining functions together. Monads extend the concept of a functors `fmap` by accepting an already wrapped value and returning a wrapped value, this function is called a monadic bind (`>>=`). The monadic bind can then be used to chain functions together. A good example for this is Haskells `Maybe` type. `Maybe` has the two constructors `Nothing` and `Just value`. The `Maybe` type is a functor, which means it provides a functor-map function [14]. This allows to apply functions to the wrapped value:

```
> (+3) <$> Just 3
Just 6
> (+3) <$> Nothing
Nothing
```

In case of `Nothing` the function is not applied and `Nothing` is returned. Since Maybe is also a monad, we can apply functions returning a wrapped value to it. For example the `half` function on integers produces `Nothing` if the value is not even:

```
half x = if even x
         then Just (x `div` 2)
         else Nothing
```

With the bind function we can chain apply it to a value:

```
>>> Just 15 >>= half
Nothing
>>> Just 16 >>= half >>= half
Just 4
```

When chaining a lot of binds together it can be useful to use the `do` notation.

```
do
  x <- Just 15
  y <- Just 16
  z <- Just 17
  return (x + y + z)
```

is equivalent to:

```
Just 15 >>= (\x ->
  Just 16 >>= (\y ->
    Just 17 >>= (\z -> (x + y + z))))
```

4.1.1 State Monad

The state monad is a monad that allows to store values in an implicit state. This is usefull when implementing a finite state machine to store the current state. In equivalence to Clash's `mealy` function the state monad provides the `runState` function:

```
runState ::
    State s a -> -- state computation
    s ->         -- initial state
    (a, s)       -- (return value, next state)
```

For this the monadic type `FSM` can be defined.

```
type FSM = State FsmInternalState
```

On this type a `step` function can be defined. This functions changes the internal state and produces output.

```
step:: FSMIn -> FSM FSMOut
step CpuIn{..} = gets nextState >>= \case
    CycleStart -> do
        modify $ \s -> s{nextState = UpdateR0}
        return $ CpuOut{data_out = 0}
    UpdateR0 -> do
        x <- gets r1
        modify $ \s -> s{r0 = x + 1}
    ...
```

The `modify` and `gets` functions are used to modify and retrieve the current state. When defining `lenses` on the `FsmInternalState` the do notation can be used to make the code even more readable. Lenses are defined on the `FsmInternalState`, and its subtypes, using the `makeLenses` function from the `lens` package using template Haskell:

```
makeLenses ''FsmInternalState
makeLenses ''FsmRegisters
makeLenses ''FsmOutput
```

This will create lenses for each field starting with an underscore. Inter alia this will create a `Getter` and a `Setter` for each field. The `Getter` provides the `use` function, retriving a value in a state-monad context. The `Setter` provides the `.=` allowing for state modification in a state-monad context. With this the `step` function can be rewritten:

```
step:: FSMIn -> FSM FSMOut
step CpuIn{..} = use nextState >>= \case
    CycleStart -> do
        nextState .= UpdateR0
        return $ CpuOut{data_out = 0}
    UpdateR0 -> do
        x <- use r1
        r0 .= x + 1
    ...
```

4.1.2 Reader Monad

The `Reader` monad allows to read a value from a monadic context. It provides the `asks` function to read a value from the context - in this case the input of the FSM. After defining Lenses on the `FsmInput`, the `Getter`s of the fields can be used with the `view` function to read values from the input. This replaces the record syntax in the input from `x <- _data_alu (_alu2Fsm _fsmInput)` to `x <- view (data_alu.alu2Fsm)`

4.1.3 Writer Monad

Since input and state are both kept in an implicit monad context the `Writer` monad is used for output to complete a pure monadic implementation. The `Writer` monad provides the `tell` function to write a value to an implicit output monad. For the output the `First` monad is used which keeps the last value written. The `First` type is instanciated as `First Nothing` so no initial output needs to be provided. To enable bundled datatypes the types need to be updated by wrapping the fields in a `First` and having a `GenericMonoid` derivation:

```
data Fsm2Ioctr regSize = Fsm2Ioctr
    { _iwr :: First (BitVector regSize),
    _icr :: First (BitVector regSize),
    _interrupt_block :: First Bool
    }
    deriving stock (Generic, Show, Eq)
    deriving anyclass NFDataX
    deriving Semigroup via GenericSemigroup (Fsm2Ioctr regSize)
    deriving Monoid via GenericMonoid (Fsm2Ioctr regSize)
makeLenses ''Fsm2Ioctr
```

The definition of `lens`es, providing the `scribe` function in the context of a `Writer` allows the definition of a new output function:

```
(.:=) :: (MonadWriter t m, Monoid s) => ASetter s t a1 (First a2) ->
↪  a2 -> m ()
y .:= x = scribe y $ First $ Just x
```

With this change neither the output has to be saved in the state nor does it have to be initialized. Changing e.g. the `Instruction` being sent to the ALU to:

```
aluOutput . mcr .:= MovLeq
```

The `First` can then be connected to the other blocks of the fsm or unwrapped with a default output to be compiled into a single block like this:

```
adress_output = (fromMaybe  0 <$> getFirst) . view (memOutput
↪  .address) <$> from_fsm
```

4.1.4 The FSM with the Rread Write State Monad (RWS)

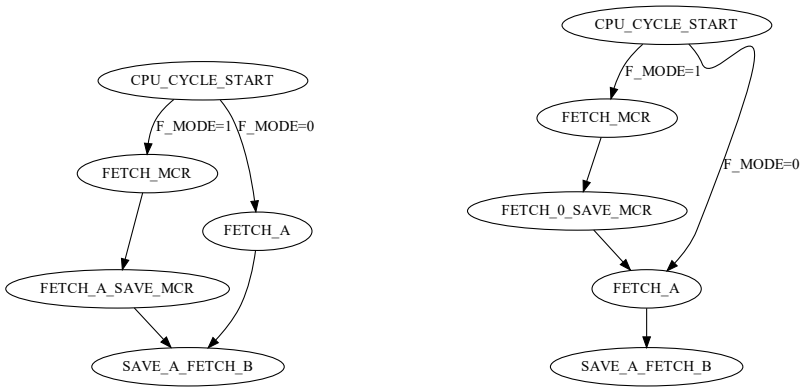
The `RWS` monad is a combination of the `Reader`, `Writer` and `State` monad. It allows to read from a context, write to an output and modify the state. The `RWS` monad is used to implement the FSM:

```
fsmRunner ::
  forall memSize regSize dom.
  (KnownNat memSize, KnownNat regSize, HiddenClockResetEnable dom)
↪  =>
  Signal dom (FsmInput memSize regSize) ->
  Signal dom (FsmOutput memSize regSize)
fsmRunner = mealy fsmMealy internalState0
  where
    fsmMealy s i =
      let ((), s', o) = runRWS mFSM i s
       in (s', o)
```

The `runRWS` function is used to run the `RWS` monad using clashs `mealy` function. One can also return the internal state for debugging purposes simply by adding `s'` to the output tuple.

4.2 FSM restructuring

Another possible improvement, is to change the whole structure of the FSM. Implementing this would go beyond the scope of this thesis, but it is worth mentioning some ideas. The current implementation uses a lot of unnecessary state-transitions in order to remain simple. For example, in multi-instruction mode the FSM changes from `FETCH_MCR` to `FETCH_0_SAVE_MCR` to `FETCH_A`. One transition can be saved by adding the state `FETCH_A_SAVE_MCR` state reducing the execution by one cycle as seen in figure 4.1. In addition `mem[mem[b]]` is fetched and saved even if it is not used. A conditional state-change to only fetch it, when the `MemR Instruction` is used, should be implemented. Another approach is to combine the states into fewer more complicated states. This would allow a pipelined implementation of the FSM as already proposed by M. Crepaldi [11]. This pipeline could be split into different functions being executed on the `RWS` monad one after another for each stage.



current optimized

Figure 4.1: Reducing execution time by introducing a new state

4.3 Primitives

To further improve the design the primitves used can be improved. The current implementation uses a two-cylce FSM. This can be improved by using a single-cycle FSM - being possible with the provided implementation. So far no tests on the maximum clock-speed have been performed. The design by M. Crepaldi [11] leaves room for higher speeds by defining only 4 clock speeds and specifying the rest as undefined. This could enable an FSM running at 533 MHz - the maximum clock speed of the Zybo Z7 [12]. One can also use a dual-port BRAM to increase memory access speed. For this, Clash provides the `trueDualPortBlockRam` function:

```
trueDualPortBlockRam :: Clock domA -- clock port A
    -> clock domB -- clock port B
    -> Signal domA (RamOp nAddrs a)        -- operation port A
    -> Signal domB (RamOp nAddrs a) --operation port B
    -> (Signal domA a, Signal domB a) -- output port A and B
```

With `RamOp` the operation can be specified by choosing one of its constructors: `RamRead (Index n)`, `RamWrite (Index n) a`, `RamNoOp`. In most cases this allows to reduce the memory loading states to be reduced to three (four in case of MemR).

1. Fetch A & B
2. Save A & B, Load MCR & C
3. Save MCR & C, Load mem[A] & mem[B]

After this the next pipeline state could use the ports until the results and flags need to be written back to the memory. When implementing all these changes, averaging to four cycles per instruction after pipelining [11] and using the max clock speed, in theory $533.000.000 \div 4 = 133.250.000$ instructions could be executed per second.

5 Verification and Testing

5.1 Classical VHDL Tests

For VHDL tests Vivados simulator is used. In the testbench enitity the Clash-generated VHDL code and its original counterpart are instantiated. Then a simple `process` is used to compare the outputs of the two instances at every clock cycle. For this, the `assert` statement is used:

```
Compare : process (CLKu)
begin
-- Compare ALU signals
assert CMPu_clash = CMPu_reference report "CMP clash: 0x" &
 ↪  to_hstring(CMPu_clash) & " != 0x" & to_hstring(CMPu_reference);
assert DATAu_clash = DATAu_reference or CMPu_reference =
 ↪  "XXXXXXXXXXXXXXXX" report "DATA clash: 0x" &
 ↪  to_hstring(DATAu_clash) & " != 0x" &
 ↪  to_hstring(DATAu_reference);
assert OVERFLOWu_clash = OVERFLOWu_reference report "OVR clash: " &
 ↪  std_logic'image(OVERFLOWu_clash) & " != " &
 ↪  std_logic'image(OVERFLOWu_reference);
...
```

This will throw an error if the outputs of the two instances are not equal and display the mismatching values. After loading a memory file the simulation can be run for a certain amount of clock cycles. The testbench will then compare the outputs of the two instances and show the results in a waveform viewer as seen in figure 5.1. Vivado also provides their Integrated Logic Analyzer (ILA) IP core. This core can be integrated into the design after synthesis and can be used to monitor the signals of the design while running on the fpga. Not to many signals can be monitored at the same time, because the core itself takes up a lot of resources [35]. This core can be used to test the memory at different speeds as it could be a bottleneck at faster clockspeeds, which would not be obvious from the simulation.

5.2 Testing in Clash

For testing the modules in Clash, a test environment with `tasty` is created. This environment uses Haskells QuickCheck to generate tests. In Quickcheck the tests are generated depending on the input. For the function `testDeEncode` the input is an `Instruction`:

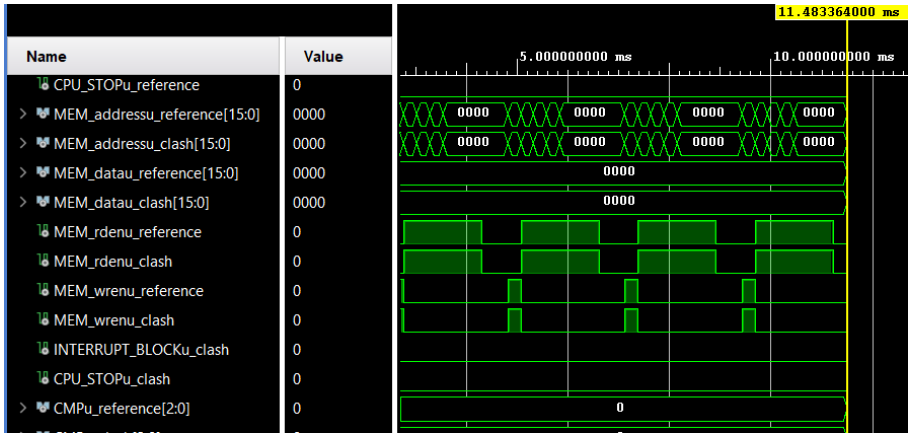


Figure 5.1: Waveform of the simulation testbench

```
testDeEnCode :: Instruction -> Bool
testDeEnCode i = i == (decodeInstruction $ (encodeInstruction i ::
↪  CLP.BitVector 16))
```

For this, the input has to be an instance of the `Arbitrary` typeclass [30]. This typeclass is used to generate random values of a certain type. The `Arbitrary` instance for `Instruction` is defined as follows:

```
instance QC.Arbitrary Instruction where
arbitrary = QC.elements [SubLeq,MovLeq ,AddLeq ,ShlLeq ,ShrLeq
↪  ,OrLeq,AndLeq ,XorLeq ,XnorLeq,Pc,Mem,MemR,PcS]
```

This will generate a random value of type `Instruction` by choosing one of the constructors of the `Instruction` type. To enable the test in the `tasty` environment it needs to be added to a `TestTree` [31]:

```
quickCheckTests :: TestTree
quickCheckTests = testGroup "Tests.Alu.TestInstructionsQC" [
    QC.testProperty "testDeEnCode" testDeEnCode
    ]
```

After starting the test environment with `stack test` the following output shows that de- and encoding instructions works as expected:

```
PS D:\git\clash-moisc\Clash\ClashProject> stack test
mosicImpl> test (suite: test-library)
  Tests
    Tests.Alu.TestInstructionsQC
      testDeEnCode: OK
        +++ OK, passed 100 tests.
```

However, the true strenght of QuickCheck can be seen when providing a faulty implementation. When testing the `AddLeq` instructions overflow for the first time the test fails:

```
testAddLeqOvr: FAIL
*** Failed! Falsified (after 1 test and 3 shrinks):
14
114
Use --quickcheck-replay=83648 to reproduce.
Use -p '/testAddLeqOvr/' to rerun this test only.
```

Here Quickcheck does automatic shrinking to provide a minimal example that fails the test. This makes it more easy to find and resolve bugs.

5.3 Verification using compiled programs from the mOISC-pipeline

For testing whole programs in Clash a seperate `topEntitySimulation` is used. This `topEntitySimulation` can be provided with a file containing the memory contents and outputs all relevant signals that would be used by the `ioBuffer` and the `clockGen`. This function will be called by the `testN` function for a first test.

```
testN :: [Char] -> Int -> String
testN filePath n = showX $ remdups  $ sampleN  n $
 ↪  topEntitySimulation filePath (clockGen @XilinxSystem)  (resetGen
 ↪  @XilinxSystem) (enableGen @XilinxSystem) (pure 0)
```

The input is left at zero for this test. The `remdups` function is used to remove consecutive duplicate outputs from the list so only changes in the output are shown. After simulating with a simple output test which writes to the `csr` register, then `idr` register and last to the `ior` register the output is correct:

```
CyclicAccurate.CyclicAccurateMOISC> testN "mem/iotest2.clash.bin"
 ↪  1000
"[(0b0000_0000,0b0000_0000,0b0000_0000,0), -- initial State
(0b1111_1111,0b0000_0000,0b0000_0000,0),   -- idr written
(0b1111_1111,0b0000_0000,0b1100_0000,0),   -- csr written
(0b1111_1111,0b1110_1000,0b1100_0000,0)]"  -- ior written
```

For further development this should be extended to use the reference implementations simulator for comparison. An `Arbitrary` instance for C-programms could be written to generate random small C programs. Those can then be compiled with all different compiler options and the resulting programms tested with the output compared to the reference implementation for a few hundred cycles. The automatic shrinking could be very useful for this.

6 Efficiency and Design Differences

6.1 Pareto Efficiency

This comparison will look at the different FSM implementations as most work and optimization was done on these. We will compare logic-budget, energy usage and lines of code (clash and VHDL). All compared objects have been compiled wtith a `memSize` of 16 and a `regSize` of 8 in Vivado 2021.2.1. The wattage is estimated by Vivados `Report Power` function, considering only the dynamic power consumption, as the static power consumption will stay the same when used together with other blocks. The results can be seen in figure 6.1. Since the Zybo Z7-20 has 53200 lookup tables and 106400 registers, a visualisation for the logic budget can be calculated by the formula $logicbudget\% = lutbudget\% + regbudget\%$ and plotted as seen in figure 6.2.

An interesting observation is that, despite being compiled to around 15000 lines of VHDL, the monadic implementation is the most efficient in terms of energy usage and of the clash implementations generally the most efficient. The pareto optimal points, in this comparison are thus the monadic implementation and the Crepaldi et. al implementation. One has to take these evaluations with a grain of salt, as the energy usage is estimated by the Vivado power estimation function with only a low confidence level. Also the monad implementation takes by far the most compile time (in average 10 times longer than the other implementations).

6.2 Speedup and Differences in the Design Process

In general ,the design process in clash is more similar to the design process in a HLS like SystemC or Vivado HLS C, as the user does not really know which way their functions will be compiled, especially when using advanced concepts like monads. Clash and Haskell generally have a high entry barrier, as the user has to learn a new language and a new way of thinking. Haskell alone is only used by 2.12% of the developer community as can be seen in the stackoverflow developer survey 2021 [29].

Implementation	LUT (%)	REG (%)	est. Wattage	l.o.c. Haskell	l.o.c. VHDL
M. Crepaldi et. al	471 (0.89%)	366 (0.34%)	32.7	-	$\sim$ 700
cyclic accurate	733 (1.38%)	326 (0.31%)	46.3	$\sim$ 1000	$\sim$ 2200
single cycle	626 (1.18%)	323 (0.31%)	66.0	$\sim$ 1000	$\sim$ 2100
monadic	625 (1.17%)	179 (0.17%)	16.6	$\sim$ 400	$\sim$ 15000

Figure 6.1: Comparison of the different mOISC FSM implementations

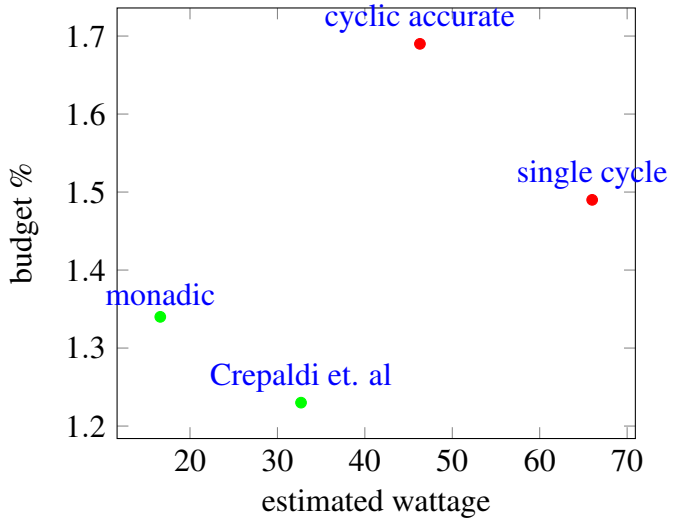


Figure 6.2: Logic budget vs wattage of the different mOISC FSM implementations.

However, once the user has learned the language, the design process can be very fast and efficient. The testing process using Haskells quickcheck is very interesting especially with automated shrinking. In the authors perspective most hardware developers already classify VHDL and Verilog as “high level” languages, as discussed in various reddit posts [28], and does not see many usecases for Clash outside of academics and hobbyists. The compilation pipeline is not very transparent and resulting VHDL code does not really natively run on FPGA boards. Here chisel [5], and other embedded DSLs may be more suited as a direct mapping to hardware can be achieved. However, as a learning experience, clash is very interesting and the author would recommend it to anyone interested in learning more about hardware design and functional programming. A sector in which clash could be used more effective is in highly specialized hardware, since many controll structures and mathematical formulas can be expressed very well in Haskell and thus in clash (e.g. a reactive controll system).

7 Conclusion

Clash can be used as a viable alternative to VHDL for the implementation of processors while prototyping. For a production ready design, VHDL is still the better choice, because it is more mature and has a larger community. Clash allows quick prototyping once one understands Haskells quirks and its compiler. The resulting VHDL code is not very readable but can be efficiently used by FPGA tools like Xilinx Vivado. Different ways to test Clash code have been shown and the resulting VHDL code can be tested with the same testbenches as VHDL code. No running version of the processor could be tested on an FPGA.

Some ideas for further optimizations have been presented in chapter 4. These optimizations can be implemented in the future to further improve the performance of the processor. More testcases should be implemented to test the processor more thoroughly. The processor can be used as a base for further research in the field of One Instruction Set Computers and Clash. Further ideas could be an automatic IP-Block pipeline and more different clock-speeds.

In conlusion Clash can be used as a viable alternative to VHDL for the implementation of processors while prototyping. For a production-ready design, VHDL is still the better choice.

Appendix A - Intel Quartus to Xilinx Vivado and vice versa

Function	Intel Quartus	Xilinx Vivado	Remarks
Signals	no initialization needed	initialization needed on startup	
Clocking	`altpll` from `altera_mf`	`PLLE2_BASE` from `UNISIM.vcomponents`	Vivado does not support 10 Khz - a Clock Divider has to be implemented
Memory	`altsyncram` with `rden` and `wrden`	`xpm_memory_spram`	Vivado has no write enable
IO-Buffer	`cyclone10lp_io_ibuf`	IOBUF	`oe` (direction pin) is inverted
Memory initialization	`.coe` files	`.mif` files	a python converter script from `.coe` to `.mif` and `.bin` for Clash can be found in the repository

Figure 7.1: IP Blocks and functions for Viavado and Quartus

Appendix B - Processor

MCR	Mnemonic	Name	DataFlow	ControlFlow
		$b < 0x08$		
-	MOV	MOVe	`mem[b] = mem[a]`	`pc = c`
		$b \geq 0x08$		
0xFF	SUBLEQ	SUBtract and jumpf if Less or EQual	`mem[b] = mem[b] - mem[a]`	
0cEE	MOVLEQ	MOVe and jumpf if Less or EQual	`mem[b] = mem[a]`	
0xCC	ADDLEQ	ADD and jumpf if Less or EQual	`mem[b] = mem[b] + mem[a]`	
0x99	SHLLEQ	SHift Left and jumpf if Less or EQual	`mem[b] = mem[b] << mem[a]`	
0x88	SHRLEQ	SHift Right and jumpf if Less or EQual	`mem[b] = mem[b] >> mem[a]`	`if mem[b]<=0`
0x77	ORLEQ	bitwise OR and jumpf if Less or EQual	`mem[b] = mem[b] | mem[a]`	`then p=c`
0x66	ANDLEQ	bitwise AND and jumpf if Less or EQual	`mem[b] = mem[b] & mem[a]`	`else pc+=3+u`
0x55	XORLEQ	bitwise XOR and jumpf if Less or EQual	`mem[b] = mem[b] ^mem[a]`	
0x44	XNORLEQ	bitwise XNOR and jumpf if Less or EQual	`mem[b] = (mem[b] ^mem[a])`	
0x33	PC	Program Counter save	`mem[b] = pc`	
0x22	MEM	MEMory double-depth addressing	`mem[mem[b]] = mem[a]`	`pc += 3 + u`
0x11	MEMR	MEMory Reverse double-depth addressing	`mem[a] = mem[mem[b]]`	`pc += 3 + u`
0x00	PCS	Program Counter Set	-	`if mem[b] == 0 then pc=c else pc=mem[b]`

Figure 7.2: Instruction set reproduced from [11].

Address	Name	Register	Execution Effects
0x00	Machine Code Register	MCR	R,W: non-blocking
0x01	CPU status and Halt Register	CHR	R: non-blocking, W (0xFF): blocking
0x02	Interrupt Wait Register	IWR	R: non-blocking, W: blocking
0x03	Interrupt Configuration Register	ICR	R,W: non-blocking
0x04	Clock Speed Register	CSR	R,W: non-blocking
0x05	Interrupt Status Register	ISR	R: non-blocking
0x06	I/O Direction Register	IDR	R,W: non-blocking
0x07	Input-Output Register	IOR	R,W: non-blocking

Figure 7.3: Registers reproduced from [11].

State	Execution
TTA_MEMR_WRITE_BACK	Transport Triggered Architecture double depth recursive write back
TTA_MEM_WRITE_BACK	Transport Triggered Architecture double depth write back
FETCH_0_SAVE_MCR	Save instruction to register
FETCH_MCR	Fetch instruction when run-mode = 1
ISR_UPDATE	Update Internal Interrupt Status Register
IWR_RESET	Update IWR Output and Set New PC
IWR_INT_SET	Set interrupt block
WAIT_FOR_INTERRUPT	Wait for intterupt from IO-Controller
ISR_WRITE_BACK	Write interrupt status register to memory
FETCH_A	Fetch Memory A
TTA_WRITE_BACK	Transport Triggered Architecture write back
LEQ_PC_WRITE_BACK	Write results back to memory and decide if flags need to be written
MEM_WRITE_BACK	double depth write back
MEMR_WRITE_BACK	double depth recursive write back
PCS_WRITE_BACK	Write back new PC_ptr
RESET	Reset the processor
CPU_CYCLE_START	Start of the cpu cycle, read IO, decide run mode
FETCH_B_SAVE_A	Fetch B and save A to register
FETCH_C_SAVE_B	Fetch C and save B to register
FETCH_D_SAVE_C	Fetch mem[B] and save C to register
FETCH_E_SAVE_D	Fetch mem[A] and save mem[B] to register
FETCH_F_SAVE_E	Fetch mem[mem[B]] and save mem[A] to register
FETCH_0_SAVE_F	Save mem[mem[B]] to register
BOOTSTRAP_CTRL	Load internal registers from memory
TTA_CTRL	Decide what to do when a register is written (interrupt, update IO,...)
EXEC_CTRL	Wait for ALU execution and decide next state depending on current instruction
LEQ_FLAGS_WRITE_BACK	Write back flags of LEQ instruction (overflow, compare result)
CPU_HALT	Halts the cpu untill reset
IWR_INT_TRIGGER	Write interrupt register to memory

Figure 7.4: FSM states

Bibliography

[1] Talwalkar Ameet. *A Conversation With Dave Patterson*. https://determined.ai/blog/dave-patterson-podcast-new. Sept. 2020.

[2] Christiaan Baaij et al. "C?aSH: Structural Descriptions of Synchronous Hardware Using Haskell". In: *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*. Sept. 2010, pp. 714–721. DOI: 10.1109/DSD.2010.21.

[3] Michael Bedford Taylor. "The Evolution of Bitcoin Hardware". In: *Computer* 50.9 (2017), pp. 58–66. ISSN: 1558-0814. DOI: 10.1109/MC.2017.3571056.

[4] John Catsoulis. *Designing Embedded Hardware: Create New Computers and Devices*. "O'Reilly Media, Inc.", May 2005. ISBN: 978-1-4493-7903-2.

[5] *Chipsalliance/Chisel3*. CHIPS Alliance. Aug. 2022.

[6] *Clash.Explicit.Mealy*. https://hackage.haskell.org/package/clash-prelude-1.6.3/docs/Clash-Explicit-Mealy.html#v:mealy.

[7] *Clash.Prelude.BlockRam.File*. https://hackage.haskell.org/package/clash-prelude-1.6.3/docs/Clash-Prelude-BlockRam-File.html#v:blockRamFile.

[8] *Clash.Signal*. https://hackage.haskell.org/package/clash-prelude-1.6.3/docs/Clash-Signal.html#hiddenclockandreset.

[9] *Clash.XException*. https://hackage.haskell.org/package/clash-prelude-1.6.3/docs/Clash-XException.html#t:NFDataX.

[10] Marco Crepaldi. *mOISC-dRISC*. Nov. 2021.

[11] Marco Crepaldi, Andrea Merello, and Mirco Di Salvo. "A Multi-One Instruction Set Computer for Microcontroller Applications". In: *IEEE Access* 9 (2021), pp. 113454–113474. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2021.3104150.

[12] Digilent, Inc. *Zybo Z7 Board Reference Manual*. https://digilent.com/reference/_media/reference/programmable-logic/zybo-z7/zybo-z7_rm.pdf.

[13] Gergő Érdi. *Retrocomputing with Clash*. Leanpub, Sept. 2021.

[14] *Functors, Applicatives, And Monads In Pictures - Adit.Io*. https://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html.

[15] "IEEE Standard for Verilog Hardware Description Language". In: *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)* (Apr. 2006), pp. 1–590. DOI: 10.1109/IEEESTD.2006.99495.

[16] "IEEE Standard VHDL Language Reference Manual". In: *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)* (Jan. 2009), pp. 1–640. DOI: 10.1109/IEEESTD.2009.4772740.

[17] Jeff Johnson. *Creating a Custom IP Block in Vivado*. https://www.fpgadeveloper.com/2014/08/creating-a-custom-ip-block-in-vivado.html.

[18] Steven Dexter Johnson. "Synthesis of Digital Designs from Recursion Equations". PhD thesis. USA: Indiana University, 1983.

[19] *Kansas Lava*. https://ku-fpg.github.io/software/kansas-lava/.

[20] Phillip Laplante and William Gilreath. "One Instruction Set Computers for Image Processing". In: *Journal of Signal Processing Systems* 38.1 (Aug. 2004), pp. 45–61. ISSN: 1939-8018. DOI: 10.1023/B:VLSI.0000028533.41559.17.

[21] Jason Lowdermilk and Simha Sethumadhavan. "Towards Zero Trust: An Experience Report". In: *2021 IEEE Secure Development Conference (SecDev)*. Oct. 2021, pp. 79–85. DOI: 10.1109/SecDev51306.2021.00027.

[22] Oleg Mazonka and Alex Kolodin. *A Simple Multi-Processor Computer Based on Subleq*. https://arxiv.org/abs/1106.2593.

[23] Oleg Mazonka, Nektarios Georgios Tsoutsos, and Michail Maniatakos. "Cryptoleq: A Heterogeneous Abstract Machine for Encrypted and Unencrypted Computation". In: *IEEE Trans.Inform.Forensic Secur.* 11.9 (Sept. 2016), pp. 2123–2138. ISSN: 1556-6013, 1556-6021. DOI: 10.1109/TIFS.2016.2569062.

[24] Conor Mcbride and Ross Paterson. "Applicative Programming with Effects". In: *Journal of Functional Programming* 18.1 (Jan. 2008), pp. 1–13. ISSN: 1469-7653, 0956-7968. DOI: 10.1017/S0956796807006326.

[25] Rishiyur S. Nikhil. *Designing Hardware Systems and Accelerators with Open-Source BH (Bluespec Haskell)*. Nov. 2021.

[26] Eriko Nurvitadhi et al. "Accelerating Binarized Neural Networks: Comparison of FPGA, CPU, GPU, and ASIC". In: *2016 International Conference on Field-Programmable Technology (FPT)*. Dec. 2016, pp. 77–84. DOI: 10.1109/FPT.2016.7929192.

[27] Saunders Mac Lane. *Categories for the Working Mathematician*. ISBN: 978-1-4757-4721-8.

[28] spca2001. *CLASH: Functional HDL Language Built on Haskell*. www.reddit.com/r/FPGA/comments/qu689b/. Reddit Post. Nov. 2021.

[29] *Stack Overflow Developer Survey 2021*. https://insights.stackoverflow.com/survey/2021/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2021.

[30] *Test.QuickCheck*. https://hackage.haskell.org/package/QuickCheck-2.14.2/docs/Test-QuickCheck.html#t:Arbitrary.

[31] *Test.Tasty*. https://hackage.haskell.org/package/tasty-1.4.2.3/docs/Test-Tasty.html#t:TestTree.

[32] The Clash Developers. *Clash Documentation*. https://clash-lang.readthedocs.io/en/latest/index.html.

[33] Lenny Truong and Pat Hanrahan. "A Golden Age of Hardware Description Languages: Applying Programming Language Techniques to Improve Design Productivity". In: (2019), 21 pages. DOI: 10.4230/LIPICS.SNAPL.2019.7.

[34] W.H. Wolf. "Hardware-Software Co-Design of Embedded Systems". In: *Proceedings of the IEEE* 82.7 (July 1994), pp. 967–989. ISSN: 1558-2256. DOI: 10.1109/5.293155.

[35] www.xilinx.com. *Integrated Logic Analyzer v6.2*. https://docs.xilinx.com/api/khub/documents/w1BckBqYd0w2WsdXTPu1og/content?Ft-Calling-App=ft%2Fturnkey-portal&Ft-Calling-App-Version=4.0.5&filename=pg172-ila.pdf.

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, August 26, 2022

_____

Jakob Groß