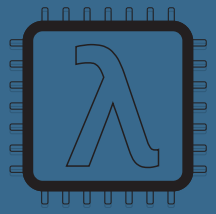


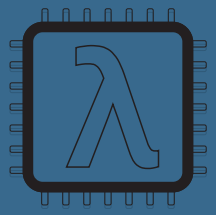
Numerical mathematics on FPGAs using CλaSH

From Haskell to a hardware accelerator

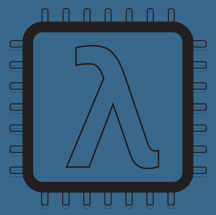
Martijn Bakker

Computer Architecture for Embedded Systems (CAES)
University of Twente

July 1, 2015



Overview

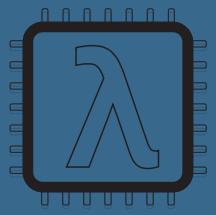


Properties

- Main building block: functions

- No assignments, only *unchangeable* definitions
- No statements, only expressions
- 'Variables' that cannot vary

- The execution of a program is a function evaluation

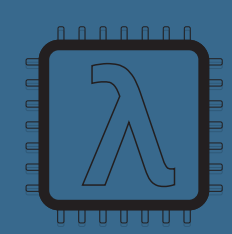


Resulting features

- No global, mutable state
- No side effects
- Pure functions

- Lazy evaluation
- Higher-order functions
- Strong type system
- Partial function application
- Function composition

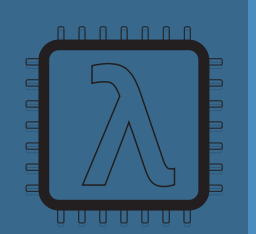
- Clear structure of the program, similar to mathematics



Example of functional programming

Listing 1: A very short introduction to functional programming

```
1  fib :: Integral a => a -> a
2  fib n | n == 0  = 1
3        | n == 1  = 1
4        | otherwise = fib (n-2) + fib (n-1)
5
6  fib_list :: Integral a => a -> [a]
7  fib_list n = map fib [0..n]
8
9  choose_list :: Integral a => (a -> a) -> [a]
10 choose_list function = map function [0..]
11
12 sum_list :: Integral a => [a] -> a
13 sum_list list = foldl (+) 0 list
```



The Field-Programmable Gate Array

- 'Programmable hardware'
- Create your own circuit instead of a list of instructions

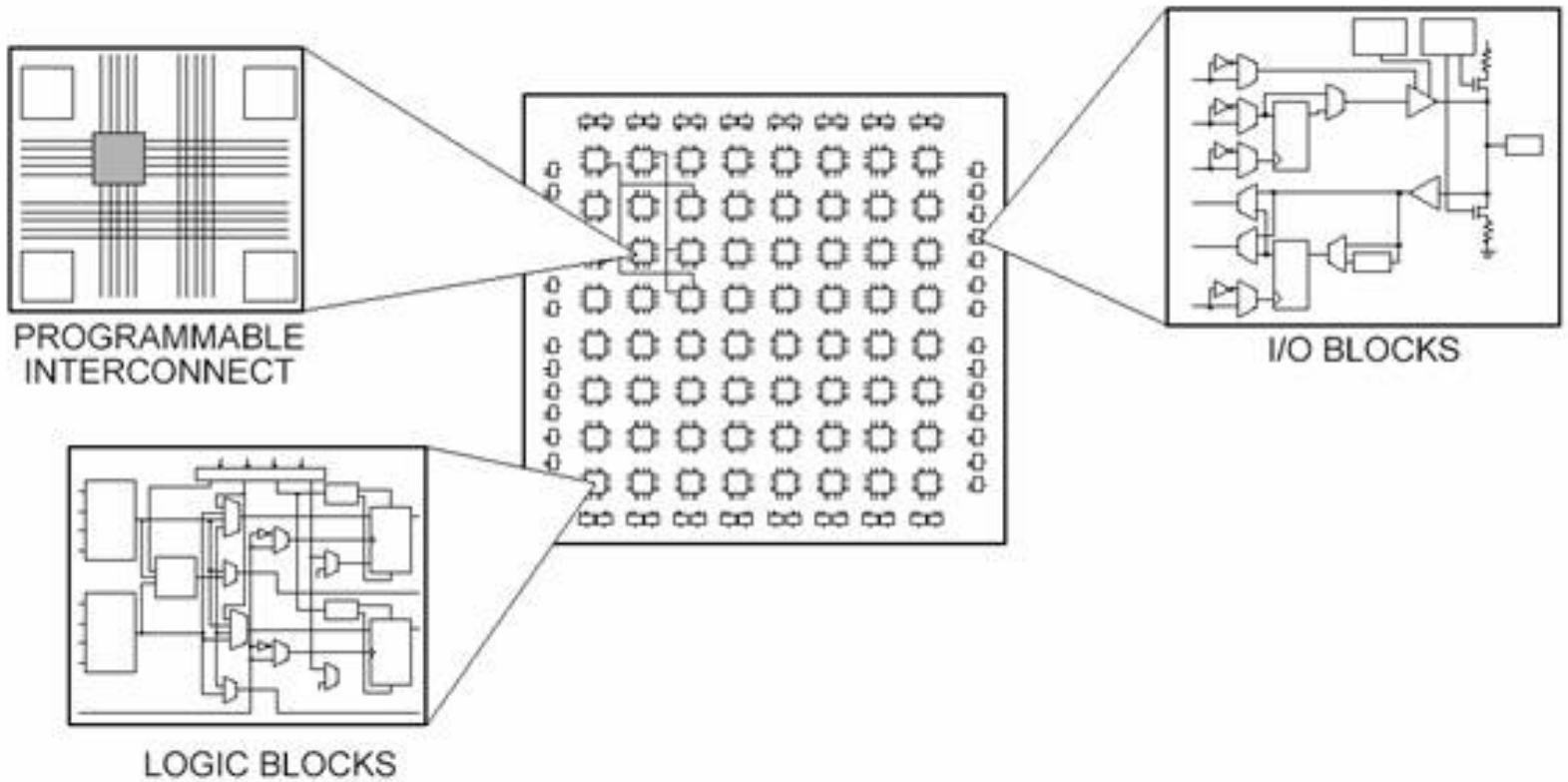
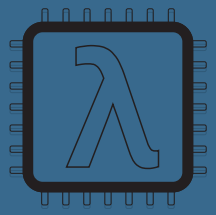


Figure: FPGA structure



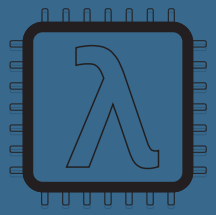
Why would you want to use an FPGA?

Advantages over CPUs

- High throughput
- Low *guaranteed* latency
- Low power use
- Support for new "instructions"

Advantages over ASICs

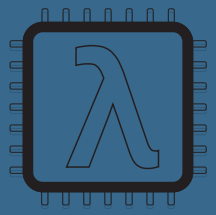
- Reconfigurability



Drawbacks

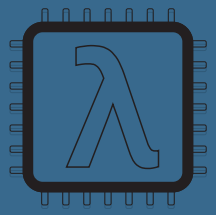
Listing 2: A single AND-gate specified in VHDL

```
1  LIBRARY IEEE;
2  USE IEEE.std_logic_1164.all;
3
4  ENTITY andgate IS
5  PORT (
6     input_0    : in  std_logic;
7     input_1    : in  std_logic;
8     output_0   : out std_logic
9     );
10 END ENTITY andgate;
11
12 ARCHITECTURE a OF andgate IS
13 BEGIN
14    output_0 <= input_0 and input_1;
15 END ARCHITECTURE a;
```



Drawbacks

1. FPGA development is **hard**
   - Vendor-specific, closed-source tools
   - Small communities
   - Ancient HDLs
   - Synthesis, debugging and verification is a slow process
2. FPGAs cannot be reconfigured as quickly as CPUs
3. FPGAs run at lower clock speeds than ASICs

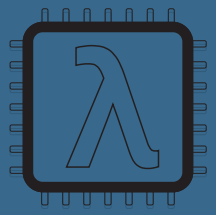


CλaSH - What?

- CAES Language for Synchronous Hardware

- A library for the specification of hardware in Haskell.
- Includes a compiler: generation of VHDL and Verilog
- Written by Christiaan Baaij at CAES

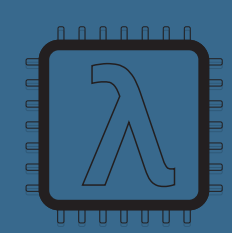
- www.clash-lang.org



CλaSH - Why?

Similarities between hardware and functional programming

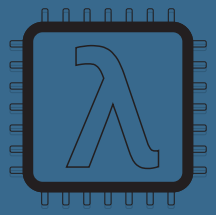
- Hardware consists of largely combinatorial parts: pure functions
- Combinatorial circuits contain a dependency tree: function dependencies
- Higher-order functions



CλaSH - Apparent issues?

- Hardware has a mutable state
- Functional programming has no mutable states

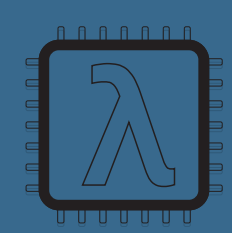
Solution: the Mealy machine.



CλaSH - Apparent issues?

- Hardware has a mutable state
- Functional programming has no mutable states

Solution: the Mealy machine.



The Mealy machine

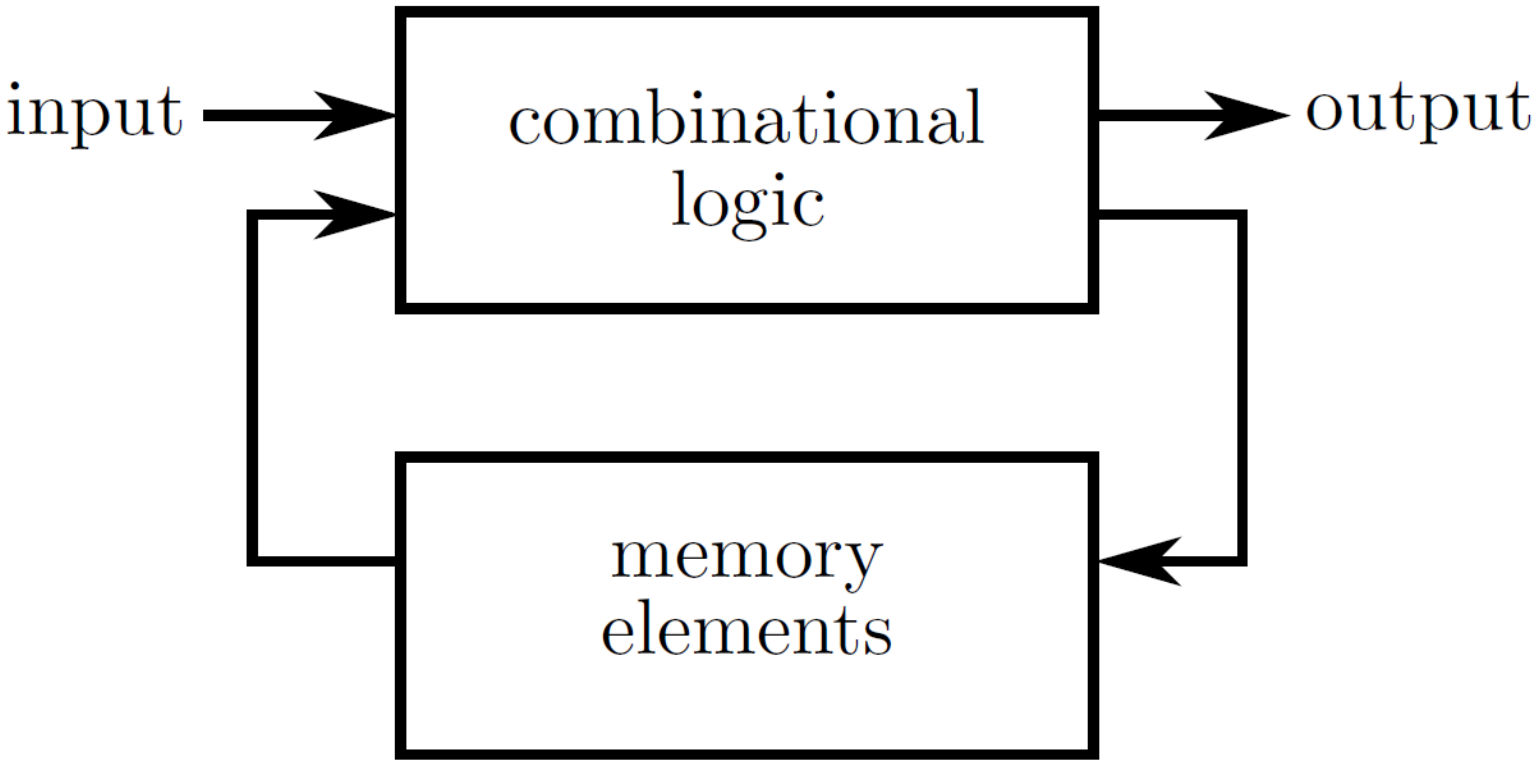
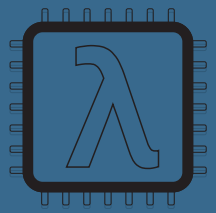


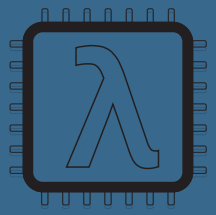
Figure: Block diagram describing a Mealy machine. The loop gets executed once every clock cycle.



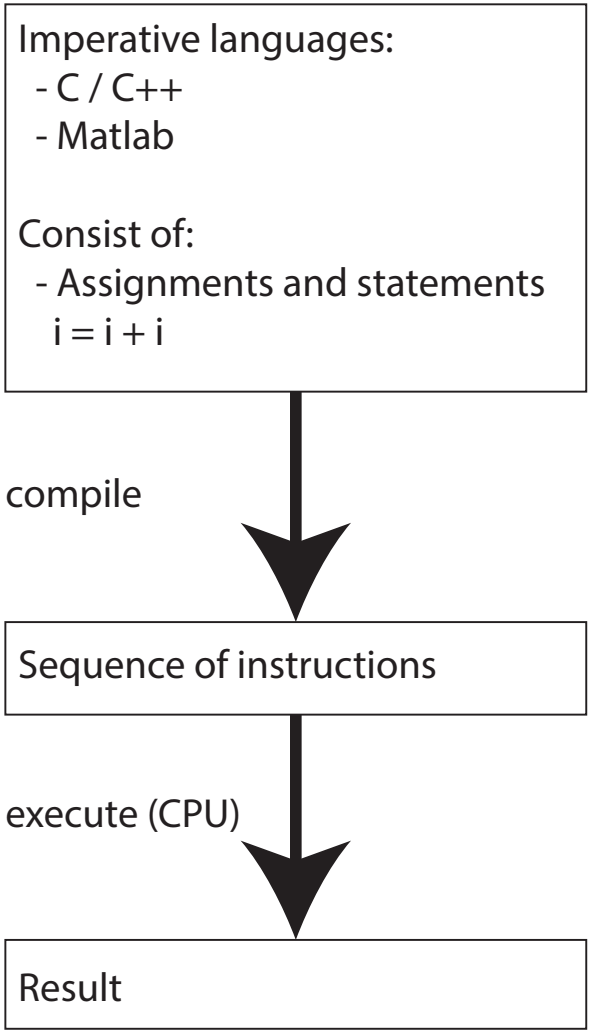
Example

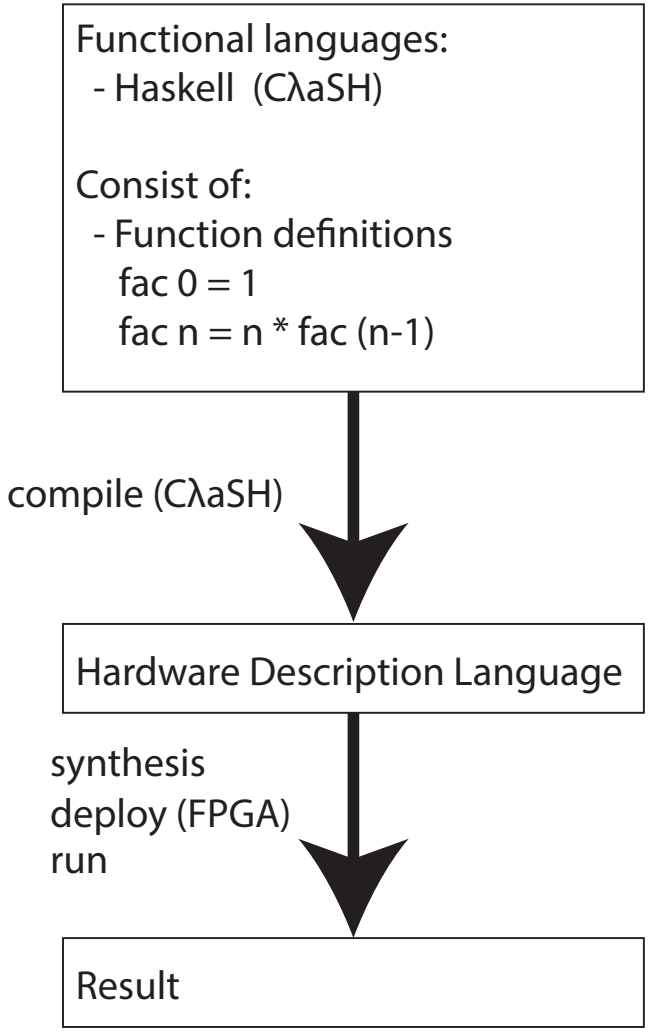
Listing 3: A basic multiply-accumulate circuit specification in CλaSH

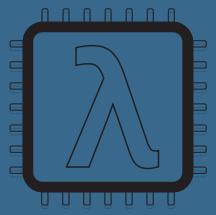
```
1  module MAC where
2    import CLaSH.Prelude
3
4    topEntity :: Signal (Signed 9, Signed 9) -> Signal (Signed 9)
5    topEntity = mealy mac 0
6
7    mac :: Num t => t -> (t, t) -> (t, t)
8    mac state input = (state', output)
9      where
10       (x,y)   = input          -- unpack the two inputs
11       state'  = state + x*y    -- the new state
12       output  = state'         -- output the new state
```



Wrap-up of introduction



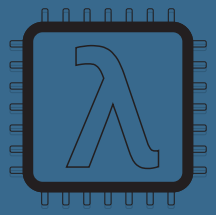




Definition

The **feasibility** and **performance** of performing **numerical approximations** to **ODEs** on an FPGA with hardware specified in CλaSH.

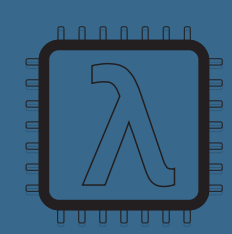
1. **Feasibility**: does the most simple version work?
2. **Performance**: comparison with a desktop CPU
3. **Numerical approximations**: simple integration schemes (Euler, RK2, RK4)
4. **ODEs**: 4 coupled first order differential equations with constant coefficients (4x4 matrix)
5. **FPGA**: a Terasic SoCKit development board, containing an Altera Cyclone V SoC FPGA



Breakdown - requirements for the system

- Number representation
- Number storage
- Control protocol
- Supplying a suitable clock frequency

1. Programming
2. Getting matrix constants and initial values in
3. Controlling and monitoring the state
4. Performing the actual computation
5. Getting results out of the FPGA (go to 3)



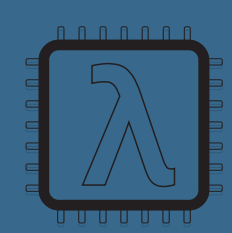
Number representation

- Floating point or fixed point?

- CλaSH only supports fixed point

- Signed fixed point with 8 integer bits and 24 fractional bits
- Total width: 32 bits
- Integer range: [-128 .. 127]
- ULP[1]: $2^{-24} \approx 6 \times 10^{-8}$, roughly 7 decimal digits.

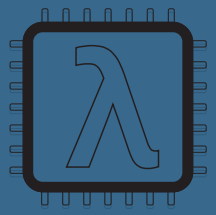
[1] Unit of least precision



Number representation

- Floating point or fixed point?
- CλaSH only supports fixed point
- Signed fixed point with 8 integer bits and 24 fractional bits
- Total width: 32 bits
- Integer range: [-128 .. 127]
- ULP[1]: $2^{-24} \approx 6 \times 10^{-8}$, roughly 7 decimal digits.

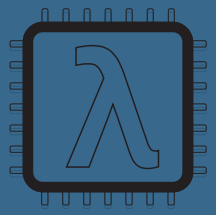
[1] Unit of least precision



Number representation

- Floating point or fixed point?
- CλaSH only supports fixed point
- Signed fixed point with 8 integer bits and 24 fractional bits
- Total width: 32 bits
- Integer range: [-128 .. 127]
- ULP[1]: $2^{-24} \approx 6 \times 10^{-8}$, roughly 7 decimal digits.

[1] Unit of least precision



Number storage

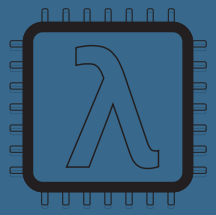
- Registers[2], SRAM[3] or SDRAM[4]?

- All variables get updated every single cycle.
- All data fits in the registers.
- No need for SRAM or SDRAM.

[2] Direcly accessible on-chip: latches

[3] Static RAM, on-chip, single cycle access delays

[4] Synchronous Dynamic RAM, off-chip, 'high' latency



Number storage

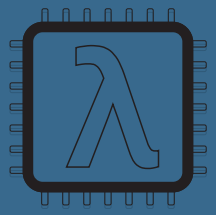
- Registers[2], SRAM[3] or SDRAM[4]?

- All variables get updated every single cycle.
- All data fits in the registers.
- No need for SRAM or SDRAM.

[2] Direcly accessible on-chip: latches

[3] Static RAM, on-chip, single cycle access delays

[4] Synchronous Dynamic RAM, off-chip, 'high' latency



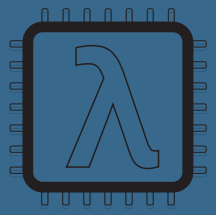
Supplying a suitable clock frequency

- A higher* clock frequency means higher throughput

- *up to a certain limit, otherwise your system fails.
- Output stabilization of the combinatorial circuits takes time.

- FPGA contains a crystal (50 MHz) and PLL[5] circuits.
- CλaSH + IO + Altera PLL = non-deterministic behaviour.
- Solution: a simple integer frequency divider.

[5] Phase-Locked Loop



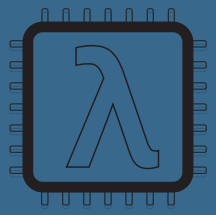
Supplying a suitable clock frequency

- A higher* clock frequency means higher throughput

- *up to a certain limit, otherwise your system fails.
- Output stabilization of the combinatorial circuits takes time.

- FPGA contains a crystal (50 MHz) and PLL[5] circuits.
- CλaSH + IO + Altera PLL = non-deterministic behaviour.
- Solution: a simple integer frequency divider.

[5] Phase-Locked Loop



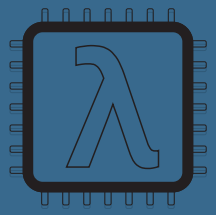
Supplying a suitable clock frequency

- A higher* clock frequency means higher throughput

- *up to a certain limit, otherwise your system fails.
- Output stabilization of the combinatorial circuits takes time.

- FPGA contains a crystal (50 MHz) and PLL[5] circuits.
- CλaSH + IO + Altera PLL = non-deterministic behaviour.
- Solution: a simple integer frequency divider.

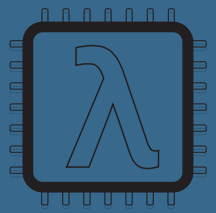
[5] Phase-Locked Loop



Actual computation - External Types

Listing 4: CλaSH topEntity for the ODE solver

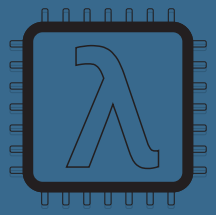
```
1 topEntity :: Signal InputSignals -> Signal OutputSignals
2 topEntity = mealy solveODE initialState
3
4 solveODE state input = (state', output)
5   where
6     state  = (systemState,systemConstants,oul,block)
7     state' = (systemState',systemConstants',oul',block')
```



Actual computation - Internal Types

Listing 5: Internal types for the ODE solver

```
1  type Data = SFixed 8 24
2  type UInt = Unsigned 32
3  type ValueVector = Vec 5 Data
4  type ConstantVector = Vec 20 Data
5
6  data ODEState = ODEState { valueVector :: ValueVector
7                           , time :: Data
8                           } deriving (Show)
9
10 type Equation = (ODEState, ConstantVector) -> ValueVector
11 type Scheme = SystemConstants -> Equation -> ODEState -> ODEState
```

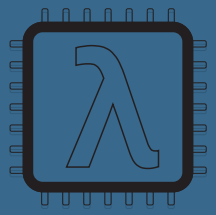


Actual computation - Equation

$$\mathbf{x}' = \begin{bmatrix} c_1 & c_2 \\ c_3 & c_4 \end{bmatrix} \mathbf{x} \tag{1}$$

Listing 6: Computing the derivative

```
1  matrix2d :: Equation
2  matrix2d (odestate, constants) = dxs
3    where
4      xs = valueVector odestate
5
6      c1 = constants !! 4
7      c2 = constants !! 5
8      c3 = constants !! 6
9      c4 = constants !! 7
10
11     x0 = c1 * (xs !! 0) + c2 * (xs !! 1)
12     x1 = c3 * (xs !! 0) + c4 * (xs !! 1)
13
14     dxs = fst $ shiftInAt0 xs (x0 :> x1 :> Nil)
```

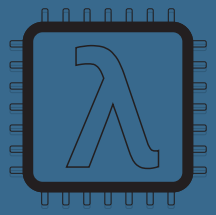


Actual computation - Scheme

$$\mathbf{x}_{k+1} = \begin{cases} \mathbf{x}_k + h\mathbf{x}'_k & \text{if } t_k < t_{max} \\ \mathbf{x}_k & \text{otherwise} \end{cases} \tag{2}$$

Listing 7: Eulers method

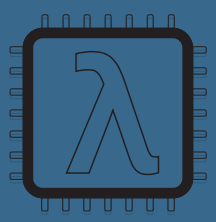
```
1  euler :: Scheme
2  euler constants equation state = state'
3    where
4      c_user = userconstants constants
5      c_maxtime = maxtime constants
6      h = timestep constants
7      t = time state
8      xs = valueVector state
9
10     --Apply Euler's integration scheme
11     eulerxs = zipWith (+) xs $ map (*h) (equation (state, c_user))
12     (xs', t') = if t < c_maxtime then (eulerxs, t + h)
13                                  else (xs,t) -- already at maximum time
14
15     state' = ODEState {valueVector = xs', time = t'}
```



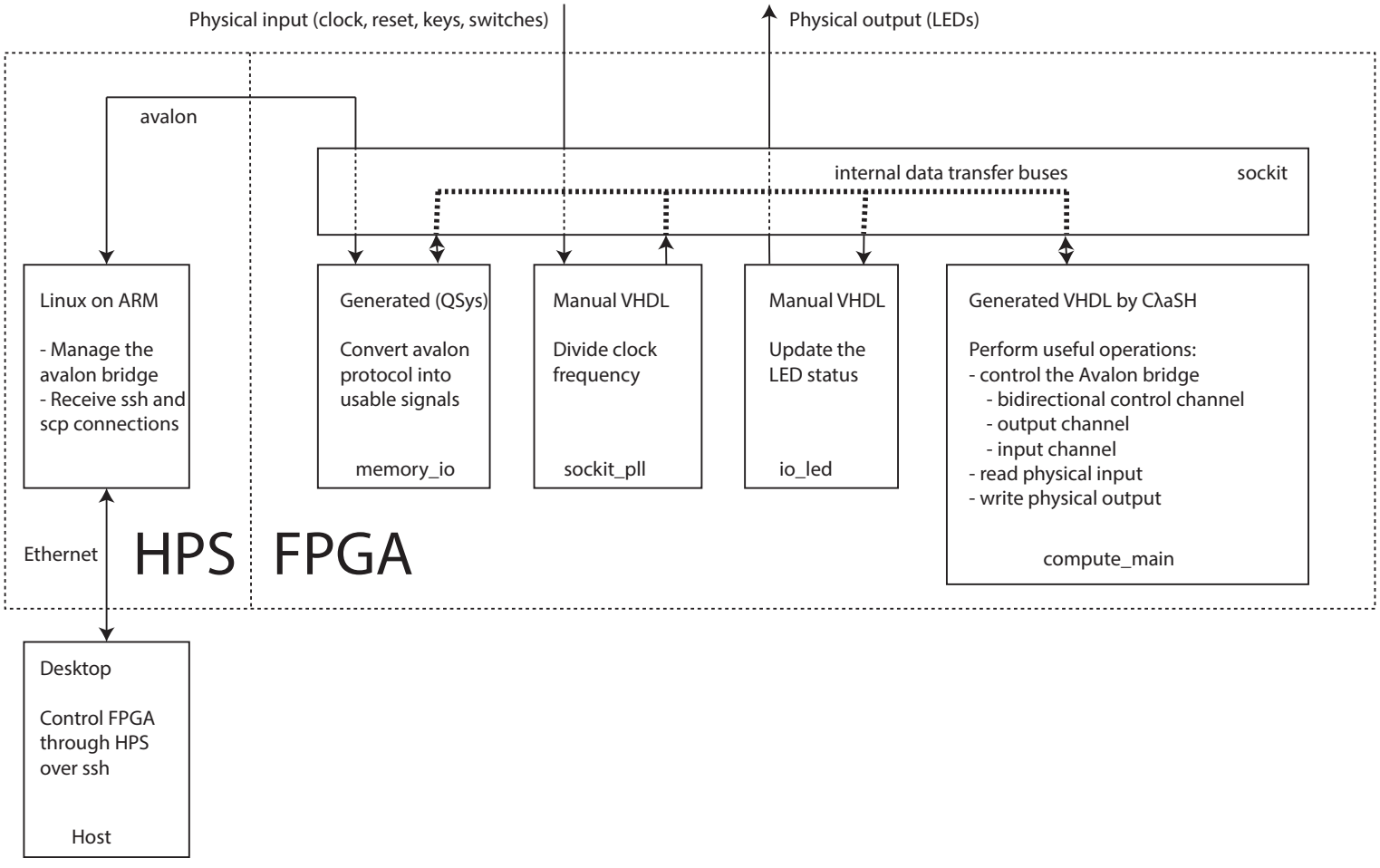
Actual computation - Control

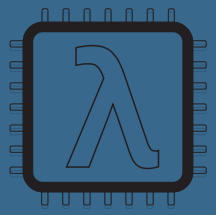
Listing 8: Controlling the ODE solver (important lines start with '!')

```
1  ! scheme = euler
2  ! equation = matrix4d
3
4  (systemState', oul', block')
5    --Handle the setup (reset the state, insert input values, start the computation)
6    | i_c == 2 && i_cs == 1 && i_ca == 0 = ( initialSystemState, 0, 0)
7    | i_is == 1                          = ( systemState{ odestate = s_odestate_in' }, 0, 1)
8    | i_c == 1 && i_cs == 1 && i_ca == 0 = ( systemState{ step = 0 } , 0, 0)
9
10   --Handle the computation and output:
11 ! | block == 1 && i_os == 1            = ( systemState, pack (xs !! i_oa), block)
12 ! | block == 0 && s_step < c_maxstep   = ( systemState_up' , 0, block)
13 ! | block == 0 && s_step >= c_maxstep = ( systemState{ step = uIntMax}, pack uIntMax, 1)
14
15   --Default, do nothing
16   | otherwise                          = ( systemState, oul, block)
17   where
18     s_odestate_in' = s_odestate {valueVector = replace i_ia (unpack i_i :: Data) xs}
19
20 !   s_odestate_up  = scheme systemConstants equation s_odestate
21     valueVector_wt = replace 4 (time s_odestate_up) (valueVector s_odestate_up)
22     s_odestate_up' = s_odestate_up {valueVector = valueVector_wt }
23     s_step'        = s_step + 1
24
25     systemState_up' = systemState{ odestate = s_odestate_up', step = s_step'}
```



An overview of the entire system

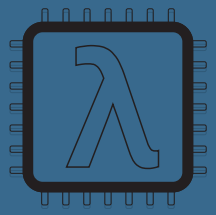




Results

- Each solution plot contains 3 curves
  1. FPGA-approximation
  2. MATLAB-approximation (same functionality as FPGA)
  3. Analytical or obtained from ode45(), the real solution
- Each error plot contains at least 2 curves
  1. FPGA - real
  2. FPGA - MATLAB approximation

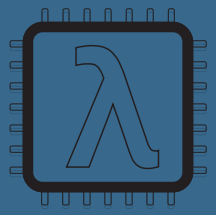
- MATLAB uses IEEE 754 double precision floating point
- FPGA uses 8/24 signed fixed point.



Results

- Each solution plot contains 3 curves
  1. FPGA-approximation
  2. MATLAB-approximation (same functionality as FPGA)
  3. Analytical or obtained from ode45(), the real solution
- Each error plot contains at least 2 curves
  1. FPGA - real
  2. FPGA - MATLAB approximation

- MATLAB uses IEEE 754 double precision floating point
- FPGA uses 8/24 signed fixed point.



Simple oscillation - I $x(t) = 50\cos(t)$

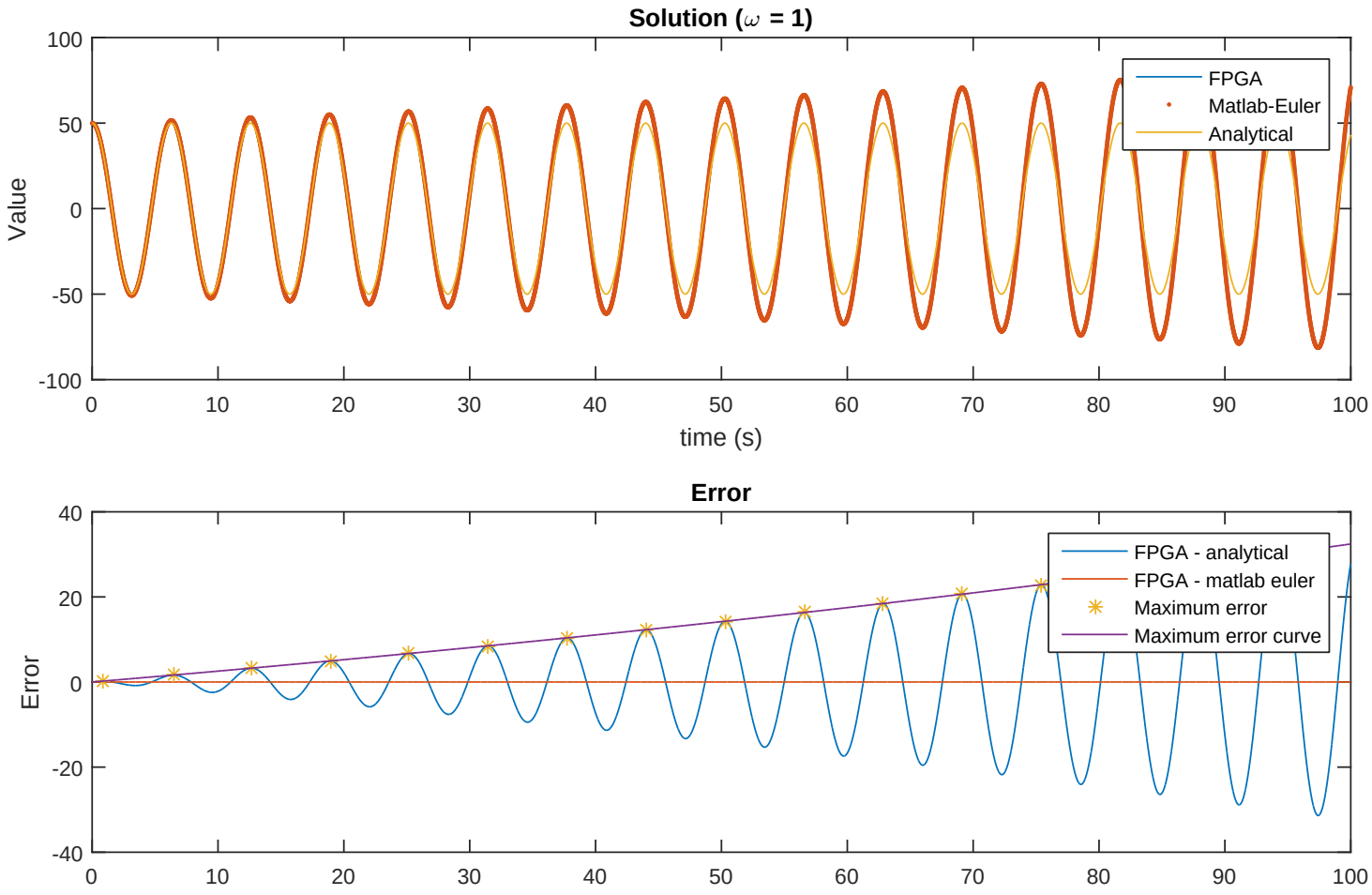
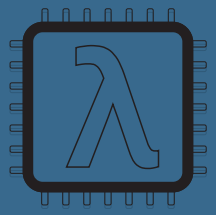


Figure: The final error is ≈ 30 (h $= 0.01$)



Simple oscillation - II $x(t) = 50\cos(t)$

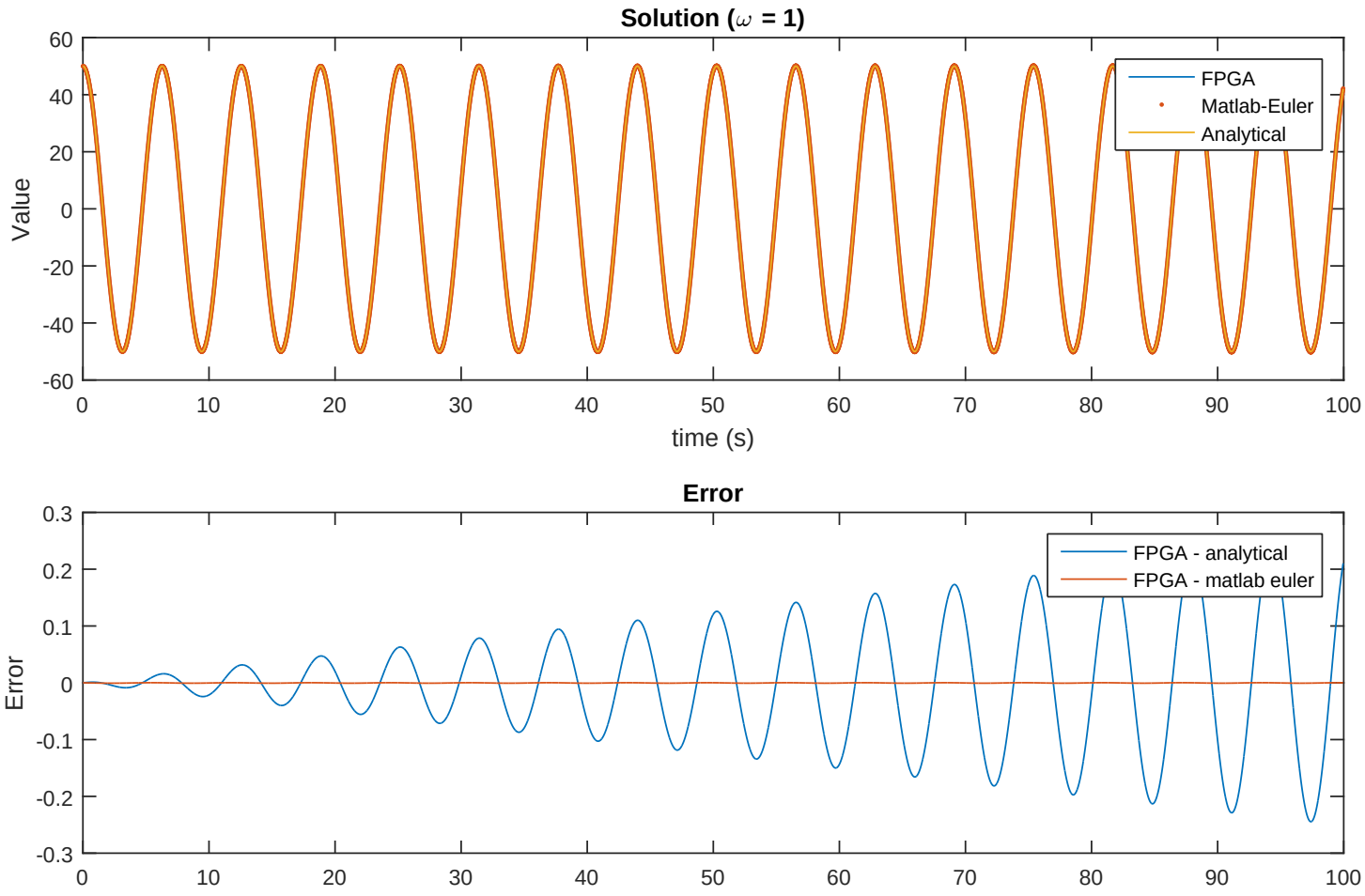
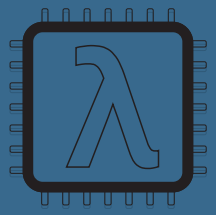


Figure: The final error decreased by a factor 150 ($h = 1 \times 10^{-4}$)



Simple oscillation - III $x(t) = 50\cos(t)$

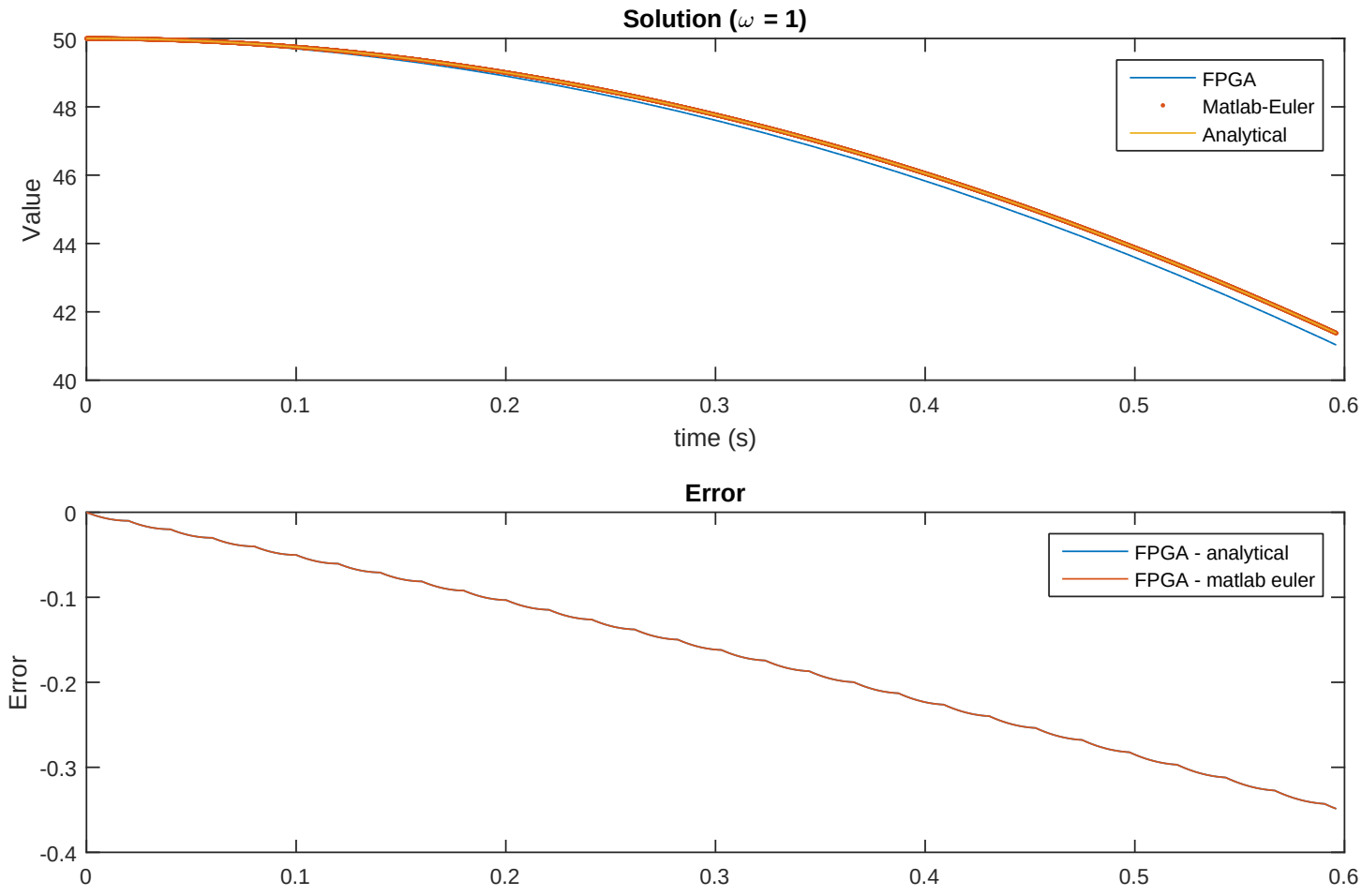
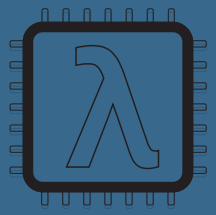


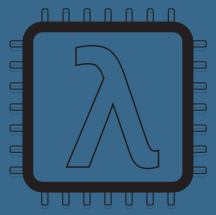
Figure: Breakdown of the number representation ($h = 1 \times 10^{-7}$)



Switching equations

- Eulers method (first order) to RK2 (second order)
- Simple oscillations to 4 coupled equations
- Matrix chosen such that all eigenvalues negative

$$\mathbf{x}' = \begin{bmatrix} 2 & 3 & 2 & 0 \\ -5 & -5 & -3 & 1 \\ 3 & -1 & -2 & -3 \\ 4 & 2 & 2 & -3 \end{bmatrix} \mathbf{x} \quad \text{with} \quad \mathbf{x}(t_0) = \begin{bmatrix} 7 \\ 5 \\ 7 \\ 5 \end{bmatrix} \tag{3}$$



RK2 - I

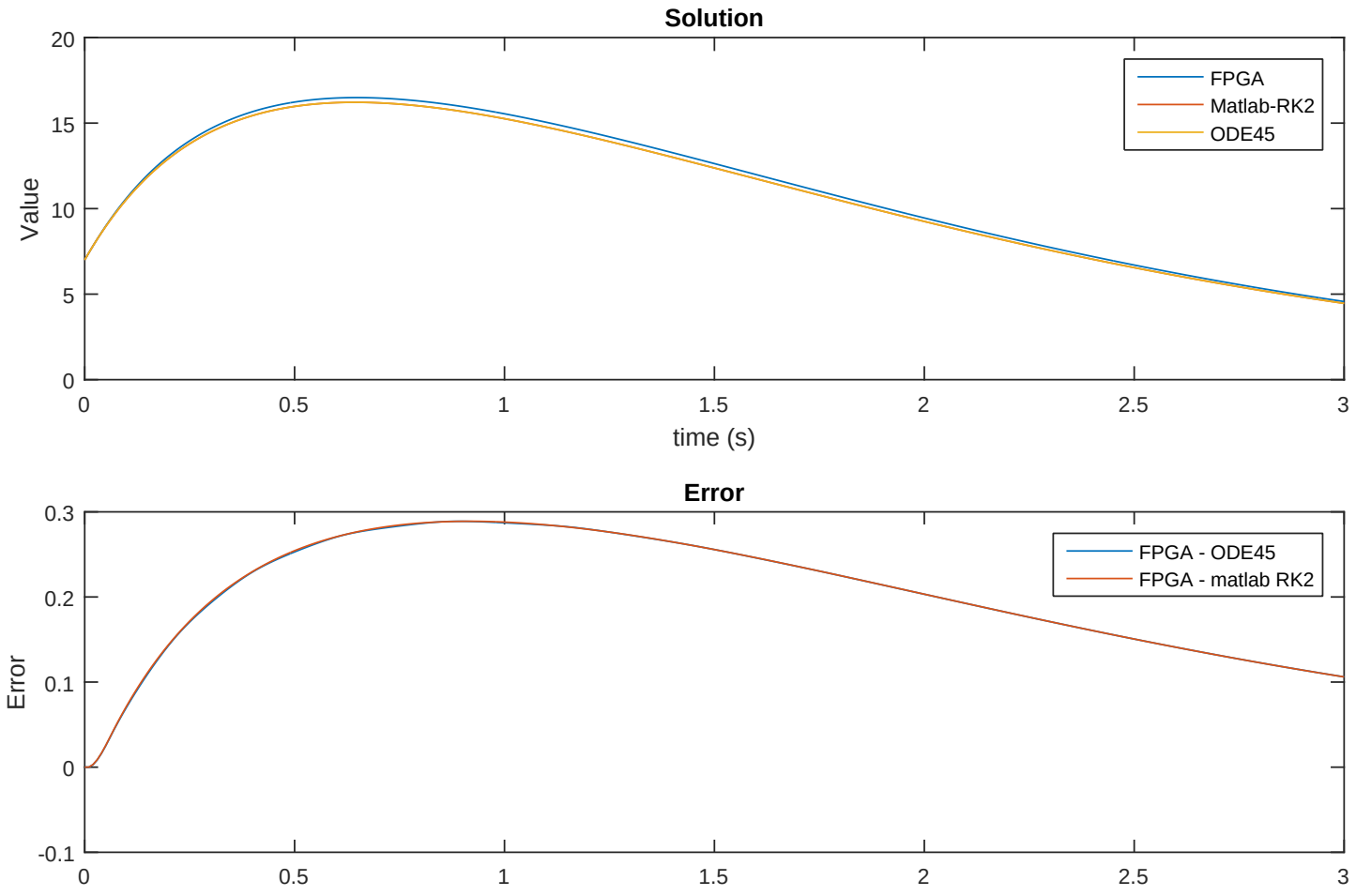
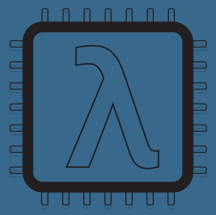


Figure: A high value of h, maximum error ≈ 0.3 (h $= 0.01$)



RK2 - II

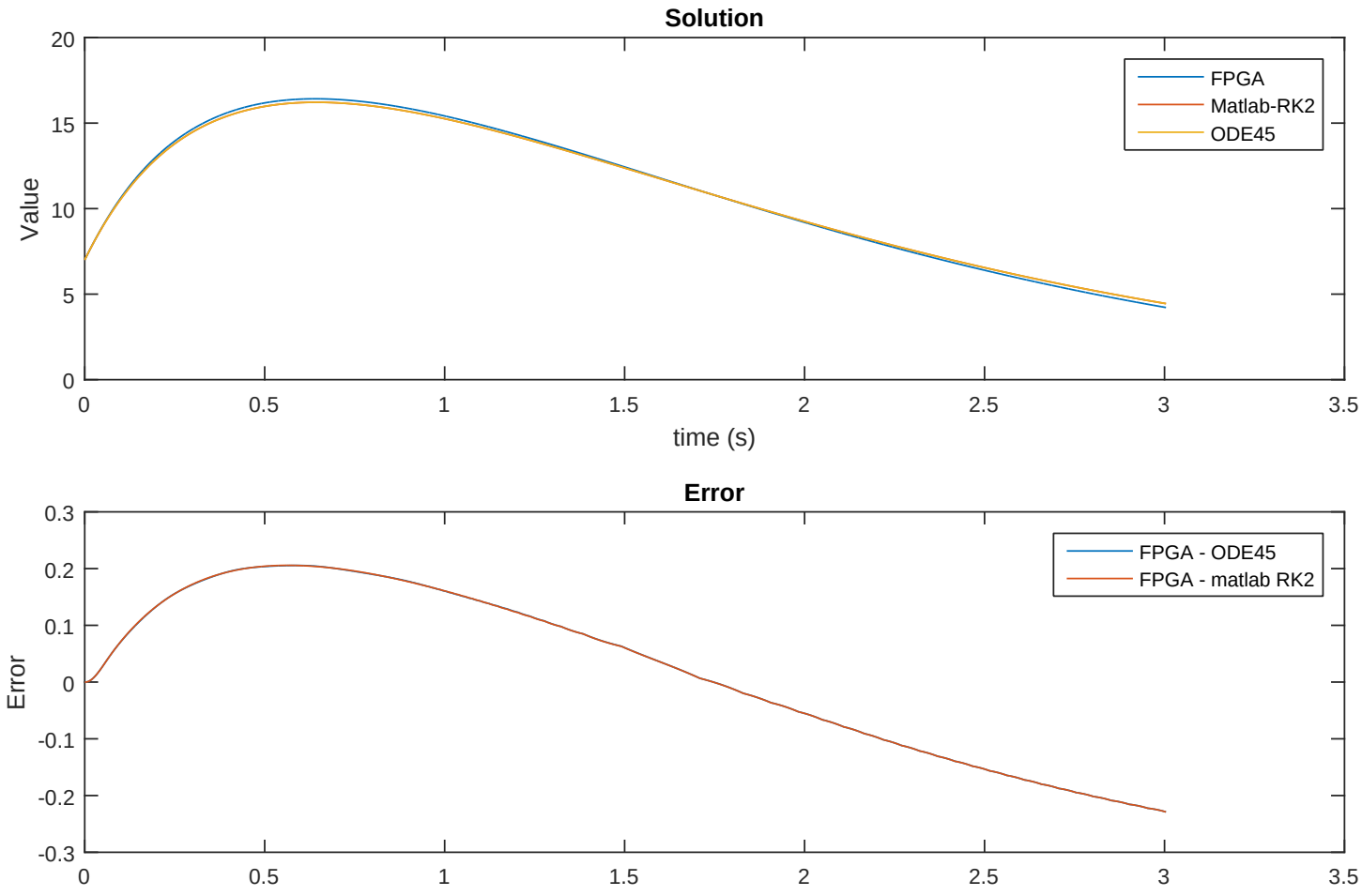
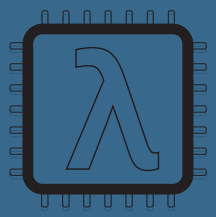


Figure: A very low value of h, maximum error ≈ 0.2 ($h = 5 \times 10^{-7}$)



RK4

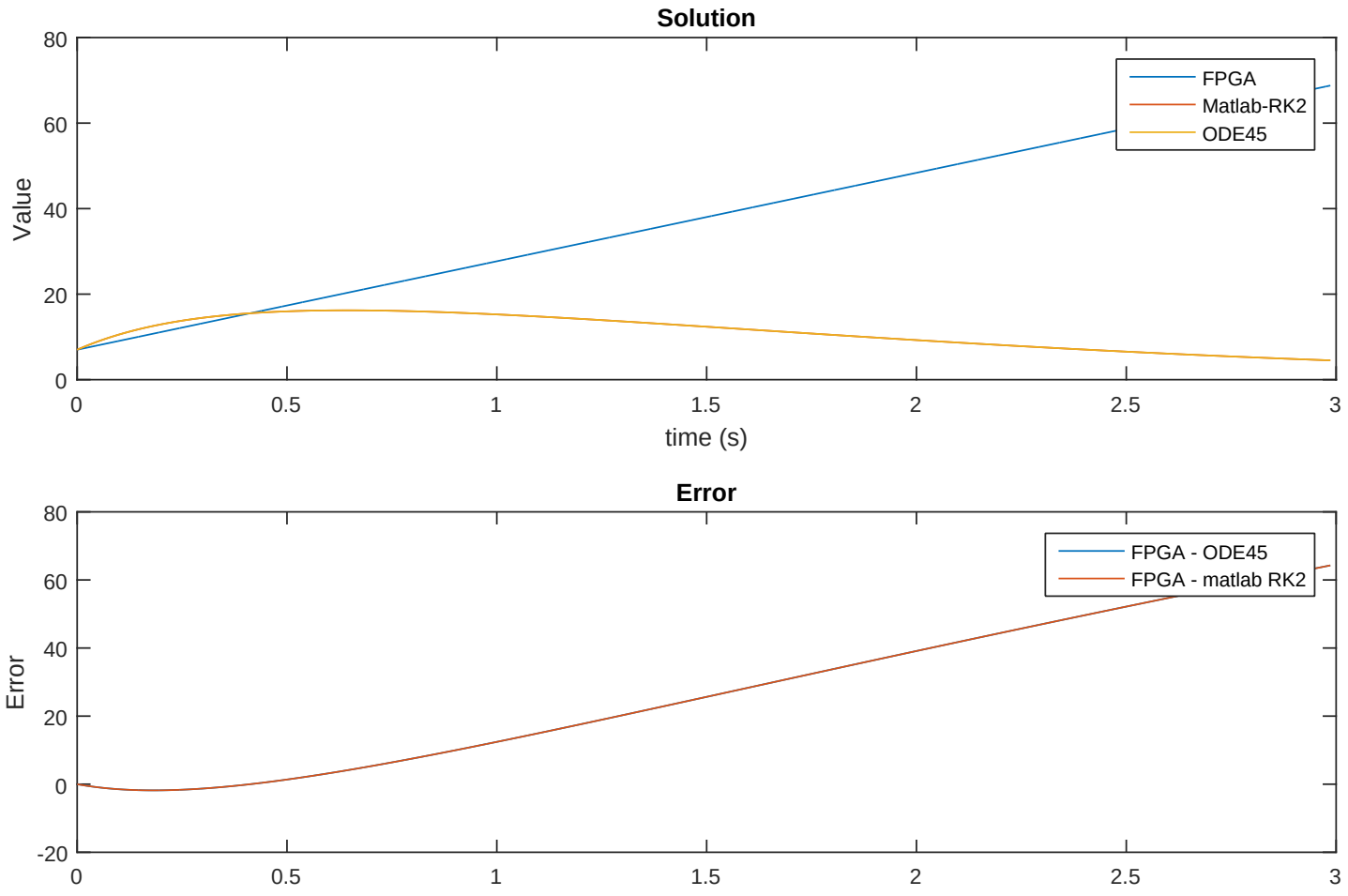
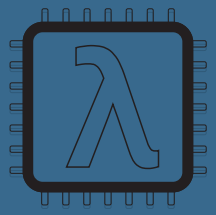


Figure: For all tested values of h, ($h = 1 \times 10^{-5}$)



Euler revisited

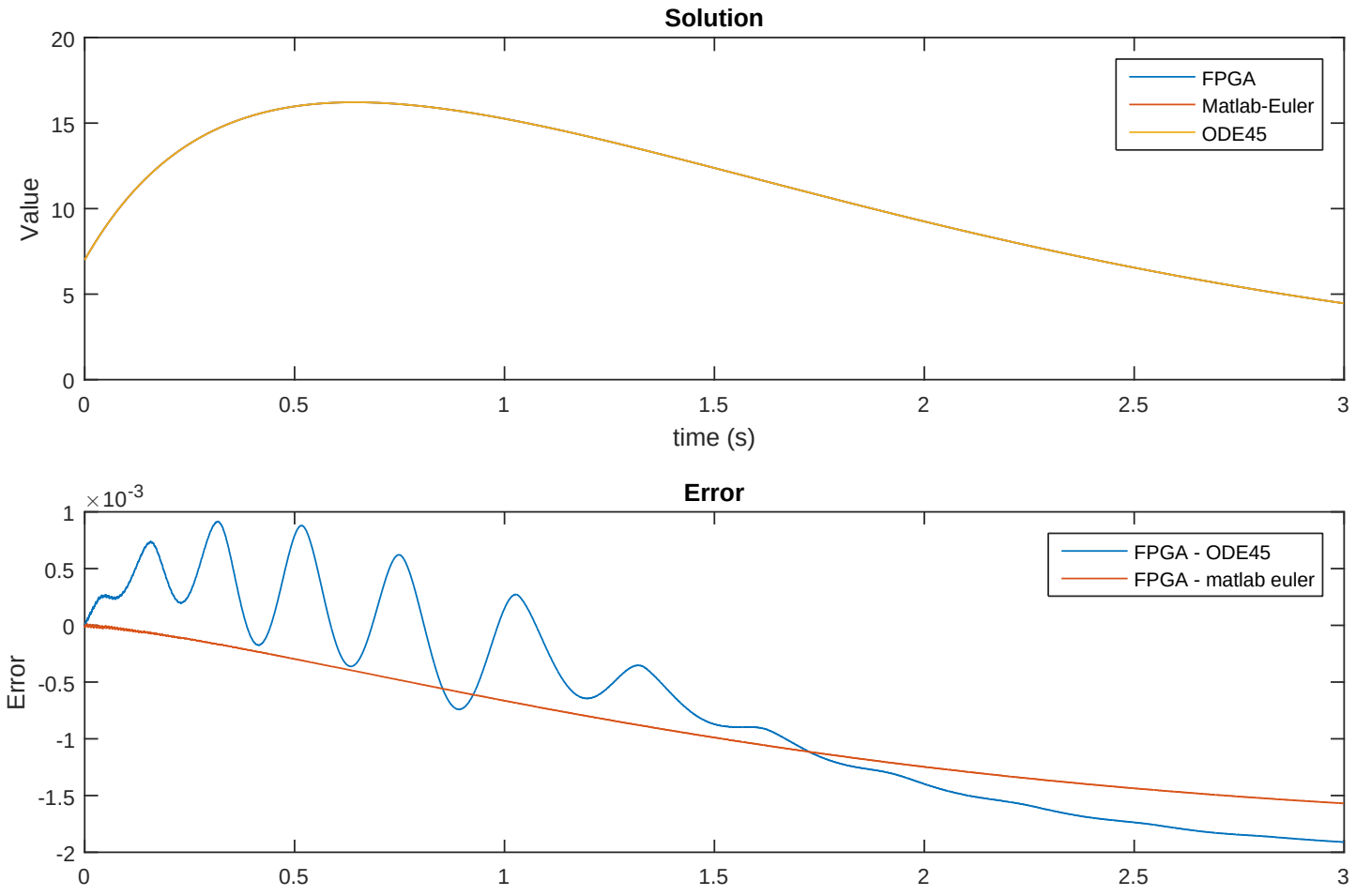
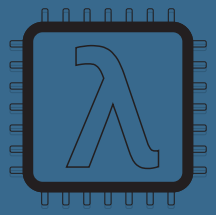


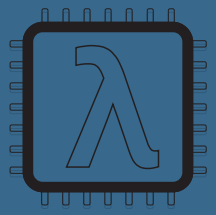
Figure: Euler outperforming both RK2 and RK4, ($h = 1 \times 10^{-4}$)



Results

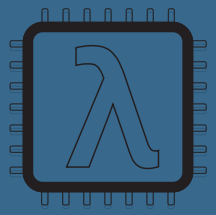
Table: Eulers method, sorted by iterations per second.

Device	Iterations	time (s)	Output interrupts	Iterations per second ($\times 10^6$)
CPU - C++ - int	1×10^8	1,35	1	74,1
FPGA	1×10^8	2,18	1	45,8
FPGA	1×10^8	2,25	1×10^3	44,4
FPGA	1×10^8	2,30	1×10^4	43,4
FPGA	1×10^8	3,27	1×10^5	30,6
FPGA	1×10^7	0,39	1	25,6
FPGA	1×10^6	0,21	1	4,76
CPU - C++ - float	5×10^7	58,1	1	0,86
FPGA	1×10^5	0,19	1	0,53
CPU - Haskell - float	1×10^6	3,23	1	0,31
CPU - MATLAB - float	1×10^7	304	1	0,03



Notes

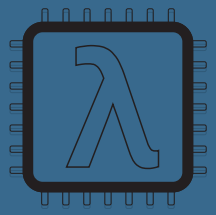
- FPGA for "low-power, cost-sensitive design needs"
- No automatic derivation of the optimal clock frequency
- 4x4 matrix equation only used ~10% of the FPGA space
- Power usage is ~ 2 orders of magnitude lower than the CPU
- CPUs are heavily optimized for fast integer arithmetic



The entire process takes ~10 minutes

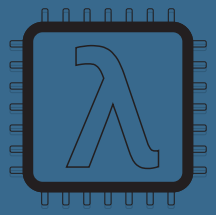
1. CλaSH - Windows/Linux
2. Synthesis (Quartus) - Windows
3. Driver compilation - (arm-linux-gnueabihf-g++) - Linux
4. Run and extract results - Windows/Linux
5. Verification of results (MATLAB) - Windows

- Normal work flow - Use two machines
- Linux for compiling and deploying the driver
- Windows for the other steps



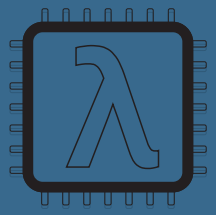
Discussion

- Accuracy
  - Equality between the FPGA and MATLAB implementations
  - Limitations of the number representation and CλaSH
- Functionality
  - 4x4 matrix with constant coefficients
  - Variety of basic solver schemes
  - Limitations of the number representation and CλaSH
- Performance
  - Sub-par development FPGA is close to a modern CPU core
  - Trade-off between power usage and performance



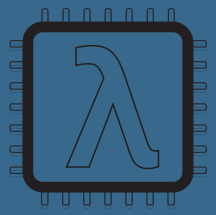
What has been created?

- Single-command toolchain integration for deploying CλaSH on an FPGA
- IO system of sufficiently high performance
  - Configuration of the Avalon bridges
  - C++ library for easy FPGA accessibility



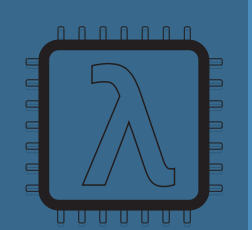
What has been shown?

- Hardware acceleration of numerical solvers is feasible
- Number representations are important
- CλaSH is usable for projects with complex IO by use as a module
- FPGAs are fast: subpar FPGA close to a modern CPU core



Acknowledgements

- Jan Kuper - Introducing me to functional programming, CλaSH and giving feedback
- Christiaan Baaij - Creating CλaSH and answering related questions
- Ruud van Damme & Jan Broenink - Feedback
- Rinse Wester - Input on configuring and using the Avalon bridges



Frequency too high? Random number generator.

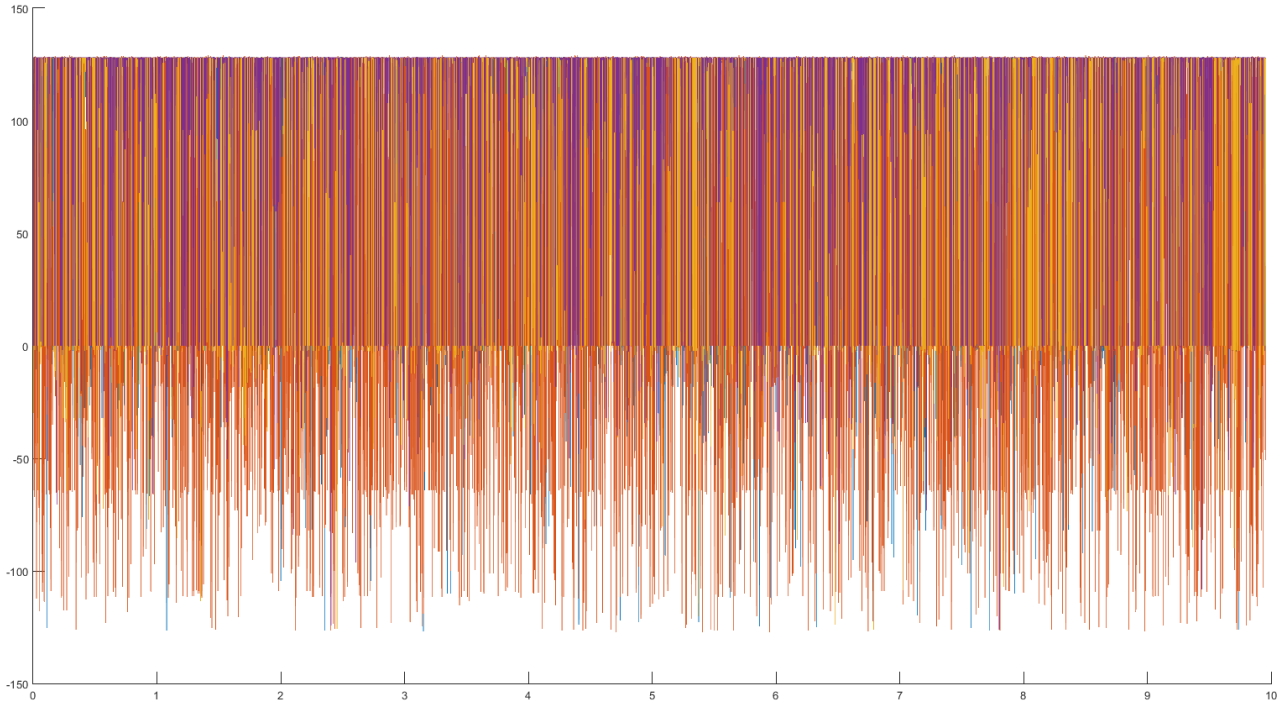
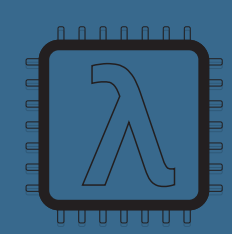


Figure: The main source of failure after overclocking.



Nondeterministic behaviour

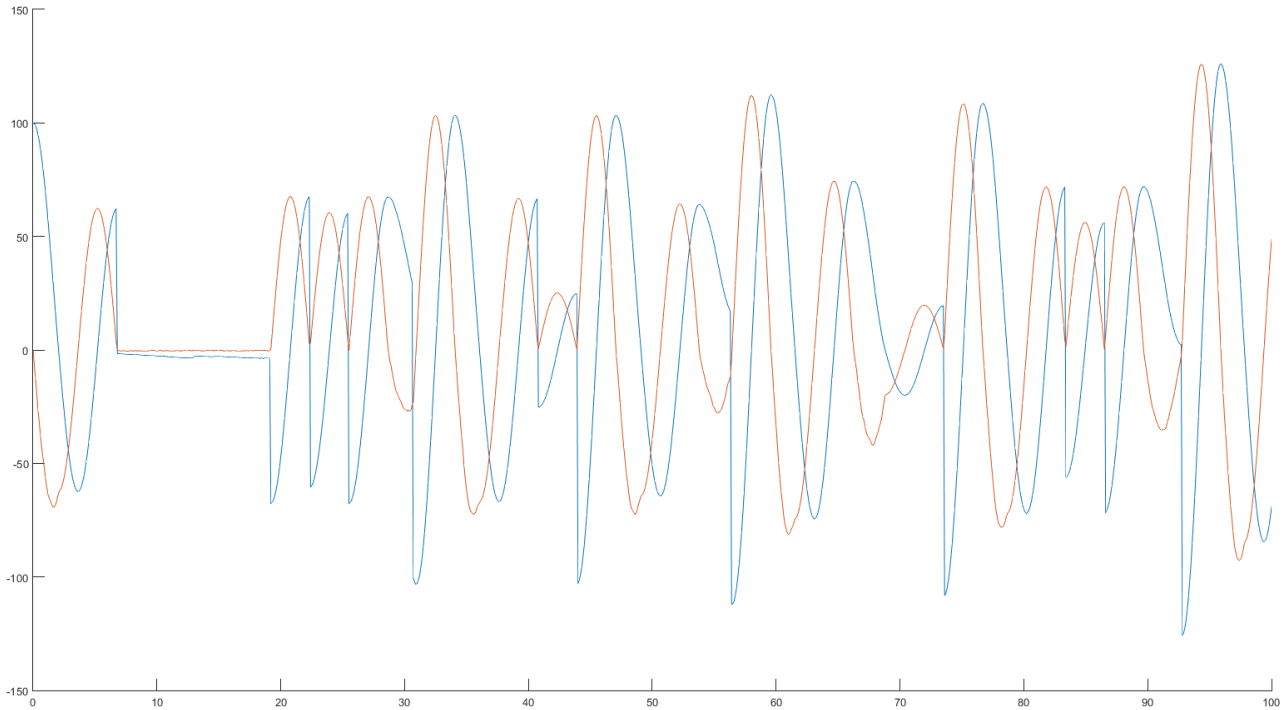
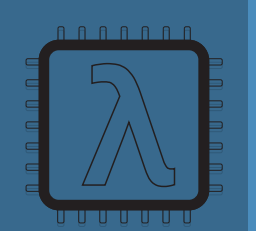


Figure: Only when using the Altera PLL



The core of the circuit: the matrix multiplier

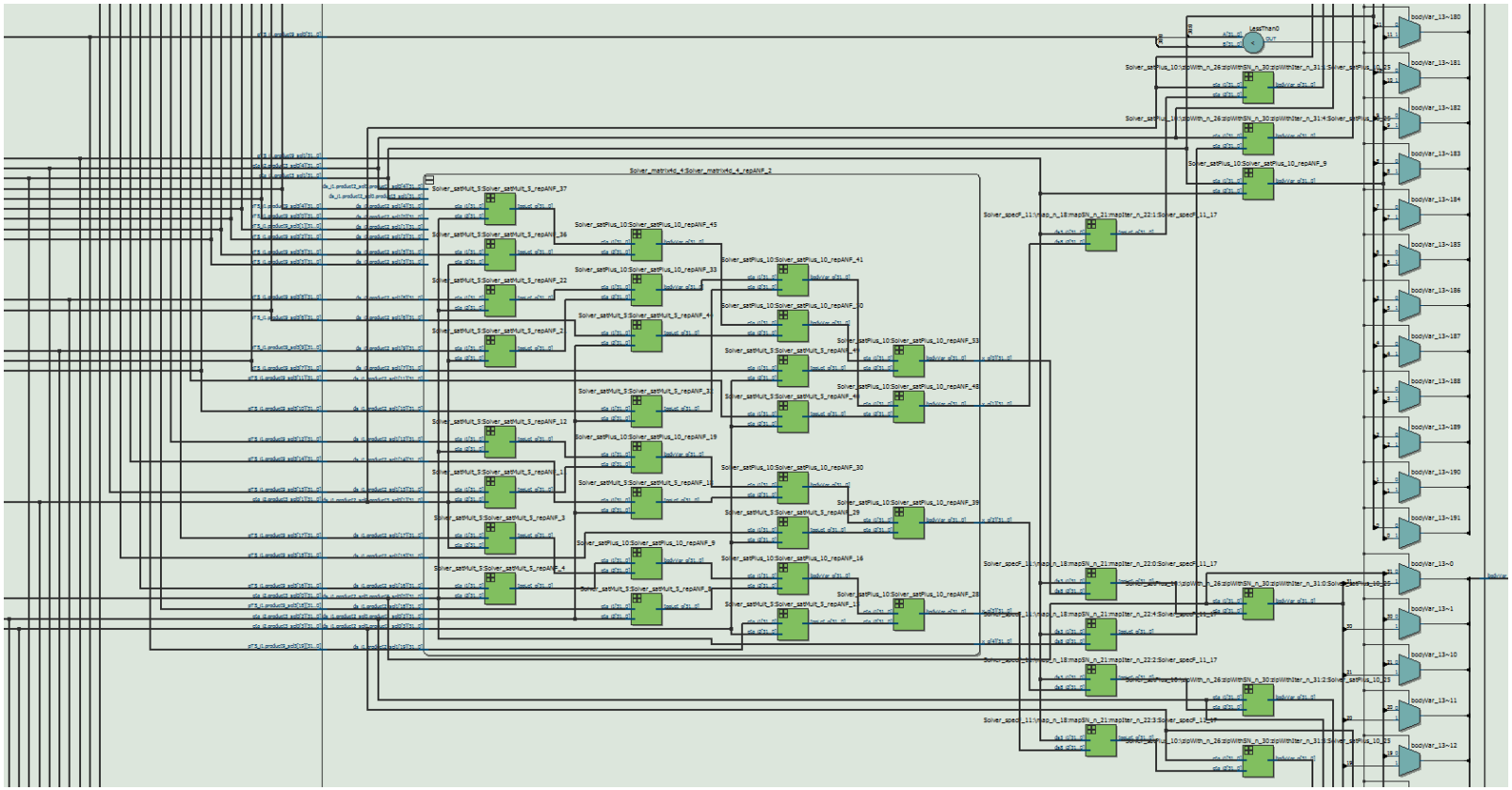
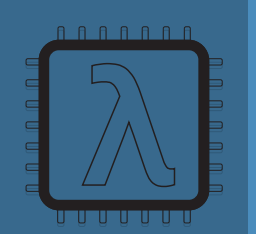


Figure: A 4x4 matrix vector multiplication consists of $4 * (4 + 3) = 28$ operations



Synthesized hardware

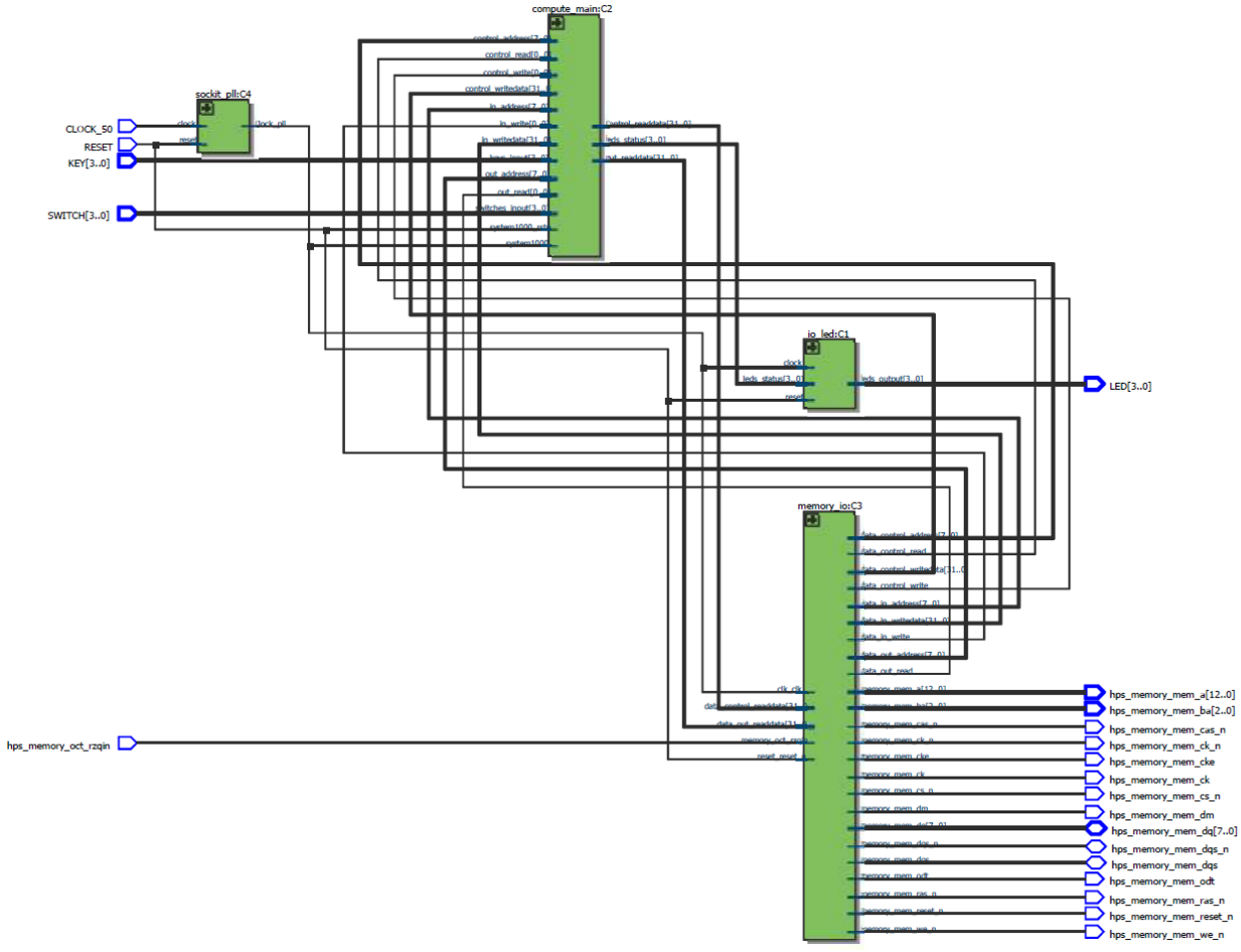


Figure: An overview of the circuit, generated by Quartus