CλasH

From Haskell To Hardware

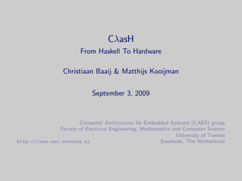
Christiaan Baaij & Matthijs Kooijman

September 3, 2009

Computer Architecture for Embedded Systems (CAES) group
Faculty of Electrical Engineering, Mathematics and Computer Science
University of Twente
Enschede, The Netherlands
`http://caes.ewi.utwente.nl`

2009-08-26

CλasH



CλasH
From Haskell To Hardware

Christiaan Baaij & Matthijs Kooijman

September 3, 2009

Computer Architecture for Embedded Systems (CAES) group
Faculty of Electrical Engineering, Mathematics and Computer Science
University of Twente
http://caes.ewi.utwente.nl
Enschede, The Netherlands

- Small tour: what can we describe in CλasH
- Quick real demo

What is CλasH?

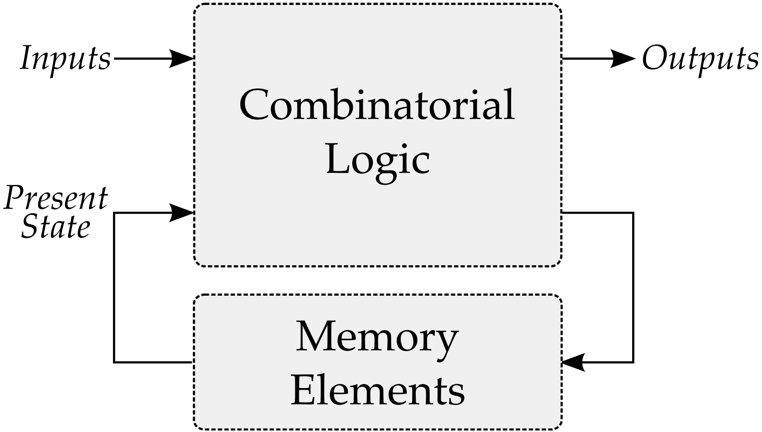
- CλasH: CAES Language for Hardware Descriptions
- Rapid prototyping language
- Subset of Haskell can be translated to Hardware (VHDL)
- Structural Description of a Mealy Machine

What is CλasH?

- CλasH: CAES Language for Hardware Descriptions
- Rapid prototyping language
- Subset of Haskell can be translated to Hardware (VHDL)
- Structural Description of a Mealy Machine

- We are a Computer Architectures group, this has been a 6 month project, no prior experience with Haskell.
- CλasH is written in Haskell, of course
- CλasH is currently meant for rapid prototyping, not verification of hardware desigs
- Functional languages are close to Hardware
- We can only translate a subset of Haskell
- All functions are descriptions of Mealy Machines

What again is a Mealy Machine?



Inputs

Outputs

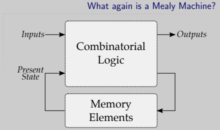
Combinatorial Logic

Present State

Memory Elements

2009-08-26

CλasH
Introduction
Mealy Machine
What again is a Mealy Machine?



What again is a Mealy Machine?

- Mealy machine bases its output on current input and previous state

Haskell Description

```
mealyMachine ::
  InputSignals →
  State →
  (State, OutputSignals)
mealyMachine inputs state = (new_state, output)
  where
    new_state = logic state input
    outputs   = logic state input
```

- Current state is part of the input
- New state is part of the output

Haskell Description

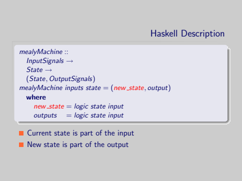
```
mealyMachine ::
  InputSignals →
  State →
  (State, OutputSignals)
mealyMachine inputs state = (new_state, output)
  where
    new_state = logic state input
    outputs   = logic state input
```

- Current state is part of the input
- New state is part of the output

Haskell Description

```
mealyMachine ::
  InputSignals →
  State →
  (State, OutputSignals)
mealyMachine inputs state = (new_state, output)
  where
    new_state = logic state input
    outputs   = logic state input
```

- Current state is part of the input
- New state is part of the output



Haskell Description

```
mealyMachine ::
  InputSignals →
  State →
  (State, OutputSignals)
mealyMachine inputs state = (new_state, output)
  where
    new_state = logic state input
    outputs   = logic state input
```

- Current state is part of the input
- New state is part of the output

- State is part of the function signature
- Both the current state, as the updated State

Simulating a Mealy Machine

```
run func state [] = []
run func state (i : input) = o : out
  where
    (state', o) = func i state
    out         = run func state' input
```

- State behaves like an accumulator

Simulating a Mealy Machine

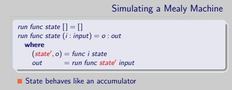
```
run func state [] = []
run func state (i : input) = o : out
  where
    (state', o) = func i state
    out         = run func state' input
```

- State behaves like an accumulator

Simulating a Mealy Machine

```
run func state [] = []
run func state (i : input) = o : out
  where
    (state', o) = func i state
    out         = run func state' input
```

- State behaves like an accumulator



Simulating a Mealy Machine

```
run func state [] = []
run func state (i : input) = o : out
  where
    (state', o) = func i state
    out         = run func state' input
```

- State behaves like an accumulator

- This is just a quick example of how we can simulate the mealy machine
- It sort of behaves like MapAccumN

Small Use Case

- Small Polymorphic, Higher-Order CPU
- Each function is turned into a hardware component
- Use of state will be simple

Small Use Case

- Small Polymorphic, Higher-Order CPU
- Each function is turned into a hardware component
- Use of state will be simple

- Small "toy"-example of what can be done in CλasH
- Show what can be translated to Hardware
- Put your hardware glasses on: each function will be a component
- Use of state will be kept simple

Imports

Import all the built-in types, such as vectors and integers:

```
import CLasH.HardwareTypes
```

Import annotations, helps CλasH to find top-level component:

```
import CLasH.Translator.Annotations
```

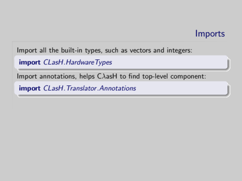
Imports

Import all the built-in types, such as vectors and integers:

```
import CLasH.HardwareTypes
```

Import annotations, helps CλasH to find top-level component:

```
import CLasH.Translator.Annotations
```



Imports

Import all the built-in types, such as vectors and integers:

```
import CLasH.HardwareTypes
```

Import annotations, helps CλasH to find top-level component:

```
import CLasH.Translator.Annotations
```

- The first input is always needed, as it contains the builtin types
- The second one is only needed if you want to make use of Annotations

Type definitions

First we define some ALU types:

$$\begin{aligned}&\textbf{type}\ \mathit{Op}\ s\ a\ \ = a \rightarrow \mathit{Vector}\ s\ a \rightarrow a\\&\textbf{type}\ \mathit{Opcode} = \mathit{Bit}\end{aligned}$$

And some Register types:

$$\begin{aligned}&\textbf{type}\ \mathit{RegBank}\ s\ a = \mathit{Vector}\ (s + \mathit{D1})\ a\\&\textbf{type}\ \mathit{RegState}\ s\ a = \mathit{State}\ (\mathit{RegBank}\ s\ a)\end{aligned}$$

And a simple Word type:

$$\textbf{type}\ \mathit{Word} = \mathit{SizedInt}\ \mathit{D12}$$

Type definitions

First we define some ALU types:

```
type Op s a   = a → Vector s a → a
type Opcode = Bit
```

And some Register types:

```
type RegBank s a = Vector (s + D1) a
type RegState s a = State (RegBank s a)
```

And a simple Word type:

```
type Word = SizedInt D12
```

Type definitions

First we define some ALU types:

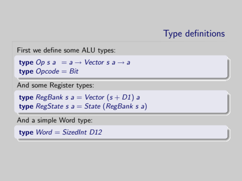
```
type Op s a   = a → Vector s a → a
type Opcode = Bit
```

And some Register types:

```
type RegBank s a = Vector (s + D1) a
type RegState s a = State (RegBank s a)
```

And a simple Word type:

```
type Word = SizedInt D12
```



Type definitions

First we define some ALU types:

```
type Op s a  = a → Vector s a → a
type Opcode = Bit
```

And some Register types:

```
type RegBank s a = Vector (s + D1) a
type RegState s a = State (RegBank s a)
```

And a simple Word type:

```
type Word = SizedInt D12
```

- The first type is already polymorphic, both in size, and element type
- It's a small example, so Opcode is just a Bit
- State has to be of the State type to be recognized as such
- SizedInt D12: One concrete type for now, to make the signatures smaller

Operations

We make a primitive operation:

$$primOp :: (a \rightarrow a \rightarrow a) \rightarrow Op\ s\ a$$
$$primOp\ f\ a\ b = a \text{ `}f\text{` } a$$

We make a vector operation:

$$vectOp :: (a \rightarrow a \rightarrow a) \rightarrow Op\ s\ a$$
$$vectOp\ f\ a\ b = foldl\ f\ a\ b$$

- We support Higher-Order Functionality

Operations

We make a primitive operation:

$$\begin{aligned}&\mathit{primOp} :: (a \rightarrow a \rightarrow a) \rightarrow \mathit{Op}\ s\ a\\&\mathit{primOp}\ f\ \ a\ b = a\,`f`\,a\end{aligned}$$

We make a vector operation:

$$\begin{aligned}&\mathit{vectOp} :: (a \rightarrow a \rightarrow a) \rightarrow \mathit{Op}\ s\ a\\&\mathit{vectOp}\ f\ \ a\ b = \mathit{foldl}\ \ f\ \ a\ b\end{aligned}$$

- We support Higher-Order Functionality

Operations

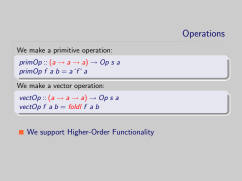
We make a primitive operation:

```
primOp :: (a → a → a) → Op s a
primOp f a b = a `f` a
```

We make a vector operation:

```
vectOp :: (a → a → a) → Op s a
vectOp f a b = foldl f a b
```

- We support Higher-Order Functionality



Operations

We make a primitive operation:

```
primOp :: (a → a → a) → Op s a
primOp f a b = a ‘f‘ a
```

We make a vector operation:

```
vectOp :: (a → a → a) → Op s a
vectOp f a b = foldl f a b
```

- We support Higher-Order Functionality

- These are just frameworks for ’real’ operations
- Notice how they are High-Order functions

Simple ALU

We define a polymorphic ALU:

```
alu ::
   Op s a →
   Op s a →
   Opcode → a → Vector s a → a
alu op1 op2 Low  a b = op1 a b
alu op1 op2 High a b = op2 a b
```

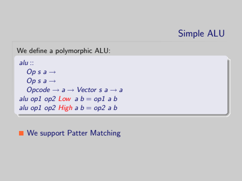
- We support Patter Matching

Simple ALU

We define a polymorphic ALU:

```
alu ::
   Op s a →
   Op s a →
   Opcode → a → Vector s a → a
alu op1 op2 Low  a b = op1 a b
alu op1 op2 High a b = op2 a b
```

- We support Patter Matching



Simple ALU

We define a polymorphic ALU:

```
alu ::
   Op s a →
   Op s a →
   Opcode → a → Vector s a → a
alu op1 op2 Low  a b = op1 a b
alu op1 op2 High a b = op2 a b
```

- We support Patter Matching

- Alu is both higher-order, and polymorphic
- We support pattern matching

Register Bank

Make a simple register bank:

```
registerBank ::
  (Some context...) ⇒ (RegState s a) → a → RangedWord s →
  RangedWord s → Bit → ((RegState s a), a)
registerBank (State mem) data_in rdaddr wraddr wrenable =
  ((State mem′), data_out)
  where
    data_out = mem ! rdaddr
    mem′ | wrenable ≡ Low = mem
         | otherwise      = replace mem wraddr data_in
```

- We support Guards

Register Bank

Make a simple register bank:

```
registerBank ::
  (Some context...) ⇒ (RegState s a) → a → RangedWord s →
  RangedWord s → Bit → ((RegState s a), a)
registerBank (State mem) data_in rdaddr wraddr wrenable =
  ((State mem′), data_out)
  where
    data_out = mem ! rdaddr
    mem′ | wrenable ≡ Low = mem
         | otherwise      = replace mem wraddr data_in
```

- We support Guards

Register Bank

Make a simple register bank:

```
registerBank ::
  (Some context...) ⇒ (RegState s a) → a → RangedWord s →
  RangedWord s → Bit → ((RegState s a), a)
registerBank (State mem) data_in rdaddr wraddr wrenable =
  ((State mem′), data_out)
  where
    data_out = mem ! rdaddr
    mem′ | wrenable ≡ Low = mem
         | otherwise      = replace mem wraddr data_in
```

- We support Guards

- RangedWord runs from 0 to the upper bound
- mem is statefull
- We support guards

Simple CPU

Combining ALU and register bank:

```
{-#ANN actual_cpu TopEntity#-}
actual_cpu ::
  (Opcode, Word, Vector D4 Word, RangedWord D9,
  RangedWord D9, Bit) → RegState D9 Word →
  (RegState D9 Word, Word)
actual_cpu (opc, a, b, rdaddr, wraddr, wren) ram = (ram′, alu_out)
  where
    alu_out = alu (primOp (+)) (vectOp (+)) opc ram_out b
    (ram′, ram_out) = registerBank ram a rdaddr wraddr wren
```

- Annotation is used to indicate top-level component

Simple CPU

Combining ALU and register bank:

```
{-#ANN actual_cpu TopEntity#-}
actual_cpu ::
  (Opcode, Word, Vector D4 Word, RangedWord D9,
  RangedWord D9, Bit) → RegState D9 Word →
  (RegState D9 Word, Word)
actual_cpu (opc, a, b, rdaddr, wraddr, wren) ram = (ram′, alu_out)
  where
    alu_out = alu (primOp (+)) (vectOp (+)) opc ram_out b
    (ram′, ram_out) = registerBank ram a rdaddr wraddr wren
```

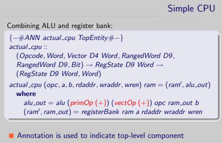
- Annotation is used to indicate top-level component

Simple CPU

Combining ALU and register bank:

```
{−#ANN actual_cpu TopEntity#−}
actual_cpu ::
  (Opcode, Word, Vector D4 Word, RangedWord D9,
  RangedWord D9, Bit) → RegState D9 Word →
  (RegState D9 Word, Word)
actual_cpu (opc, a, b, rdaddr, wraddr, wren) ram = (ram′, alu_out)
  where
    alu_out = alu (primOp (+)) (vectOp (+)) opc ram_out b
    (ram′, ram_out) = registerBank ram a rdaddr wraddr wren
```

- Annotation is used to indicate top-level component



Simple CPU

Combining ALU and register bank:

```
{−#ANN actual_cpu TopEntity#−}
actual_cpu ::
  (Opcode, Word, Vector D4 Word, RangedWord D9,
  RangedWord D9, Bit) → RegState D9 Word →
  (RegState D9 Word, Word)
actual_cpu (opc, a, b, rdaddr, wraddr, wren) ram = (ram′, alu_out)
  where
    alu_out = alu (primOp (+)) (vectOp (+)) opc ram_out b
    (ram′, ram_out) = registerBank ram a rdaddr wraddr wren
```

- Annotation is used to indicate top-level component

- We use the new Annotion functionality to indicate this is the top level
- the primOp and vectOp frameworks are now supplied with real functionality, the plus (+) operations
- No polymorphism or higher-order stuff is allowed at this level.
- Functions must be specialized, and have primitives for input and output

Demo

- We will simulate the small CPU from earlier
- Translate that CPU code to VHDL
- Simulate the generated VHDL
- See the hardware schematic of the synthesized VHDL

More than just toys

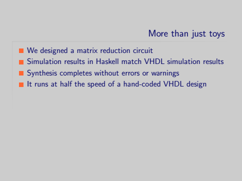
- We designed a matrix reduction circuit
- Simulation results in Haskell match VHDL simulation results
- Synthesis completes without errors or warnings
- It runs at half the speed of a hand-coded VHDL design

More than just toys

- We designed a matrix reduction circuit
- Simulation results in Haskell match VHDL simulation results
- Synthesis completes without errors or warnings
- It runs at half the speed of a hand-coded VHDL design

More than just toys

- We designed a matrix reduction circuit
- Simulation results in Haskell match VHDL simulation results
- Synthesis completes without errors or warnings
- It runs at half the speed of a hand-coded VHDL design



More than just toys

- We designed a matrix reduction circuit
- Simulation results in Haskell match VHDL simulation results
- Synthesis completes without errors or warnings
- It runs at half the speed of a hand-coded VHDL design

- Toys like the poly cpu one are good to give a quick demo
- But we used CλasH to design 'real' hardware
- Reduction circuit sums the numbers in a row of a (sparse) matrix
- Half speed is nice, considering we don't optimize for speed

So how do you make Hardware from Haskell?

In three simple steps

- No Effort:
  GHC API Parses, Typechecks and Desugars the Haskell code
- Hard:
  Transform resulting Core, GHC's Intermediate Language, to a normal form
- Easy:
  Translate Normalized Core to synthesizable VHDL

So how do you make Hardware from Haskell?

In three simple steps

- No Effort:
  GHC API Parses, Typechecks and Desugars the Haskell code
- Hard:
  Transform resulting Core, GHC's Intermediate Language, to a normal form
- Easy:
  Translate Normalized Core to synthesizable VHDL

So how do you make Hardware from Haskell?

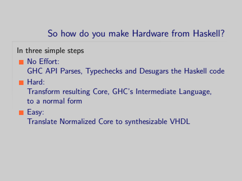
In three simple steps

- No Effort:
  GHC API Parses, Typechecks and Desugars the Haskell code
- Hard:
  Transform resulting Core, GHC's Intermediate Language, to a normal form
- Easy:
  Translate Normalized Core to synthesizable VHDL

So how do you make Hardware from Haskell?

In three simple steps

- No Effort:
  GHC API Parses, Typechecks and Desugars the Haskell code
- Hard:
  Transform resulting Core, GHC's Intermediate Language, to a normal form
- Easy:
  Translate Normalized Core to synthesizable VHDL



So how do you make Hardware from Haskell?

In three simple steps

- No Effort:
  GHC API Parses, Typechecks and Desugars the Haskell code
- Hard:
  Transform resulting Core, GHC's Intermediate Language, to a normal form
- Easy:
  Translate Normalized Core to synthesizable VHDL

- Here is a quick insight as to how WE translate Haskell to Hardware
- You can also use TH, like ForSyDe. Or traverse datastructures, like Lava.

Some final words

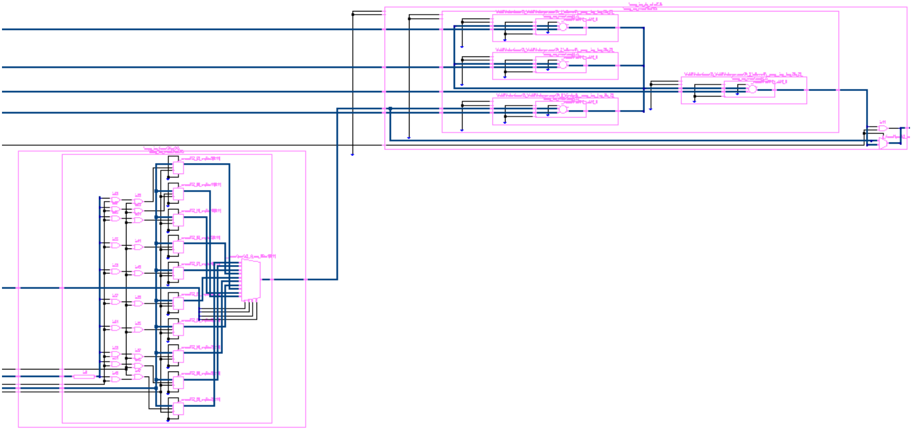
- Still a lot to do: translate larger subset of Haskell
- Real world prototypes can be made in CλasH
- CλasH is another great example of how to bring functional expressivity to hardware designs

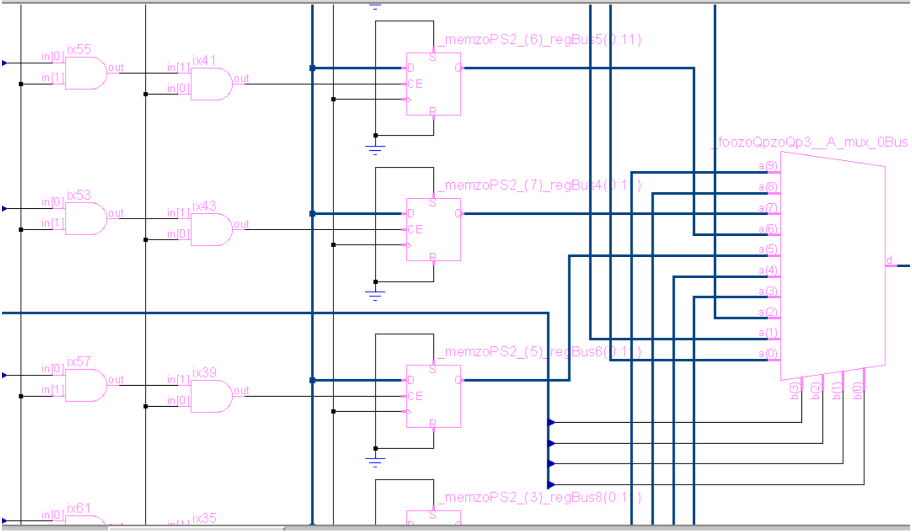
Thank you for listening

CλasH Clone URL:
`git://github.com/christiaanb/clash.git`

Complete signature for registerBank

```
registerBank ::
   (NaturalT s
   , PositiveT (s + D1)
   , ((s + D1) > s)∼True)) ⇒
   (RegState s a) → a → RangedWord s →
   RangedWord s → Bit → ((RegState s a), a)
```




ix55
in[0]
in[1]
out
ix41
in[1]
in[0]
out
memzoPS2(6)_regBus5(0:11)
S
D
CE
Q
R
ix53
ix43
memzoPS2(7)_regBus4(0:11)
_foozoQpzoQp3__A_mux_0Bus
a(9)
a(8)
a(7)
a(6)
a(5)
a(4)
a(3)
a(2)
a(1)
a(0)
d
b(3)
b(2)
b(1)
b(0)
ix57
ix39
memzoPS2(5)_regBus6(0:11)
ix61
ix35
memzoPS2(3)_regBus8(0:11)


\foldlVectorfoozoQZ_\foldlVectorparamzoQR_1_\othercell__comp__ins__tmp_28n_29_
\comp_ins_reszoGozoRC2\
_reszoRPzoRP2__add12_0
clock
resetn
clock
cin
cout
\paramzoRLzoRL2\(0:11)
\paramzoRNzoRN2\(0:11)
a(11:0)
b(11:0)
+
d(11:0)
\reszoRPzoRP2\(0:11)
resetn
Component_10
funzoR4Component_9
\foldlVectorfoozoQZ_\foldlVectorparamzoQR_2_\othercell__comp__ins__tmp_28n_29_
\comp_ins_reszoGozoRC2\
_reszoRPzoRP2__add12_0
clock
resetn
clock
cin
cout
\paramzoRLzoRL2\(0:11)
\paramzoRNzoRN2\(0:11)
a(11:0)
b(11:0)
+
d(11:0)
\reszoRPzoRP2\(0:11)
resetn
Component_10
funzoR4Component_9