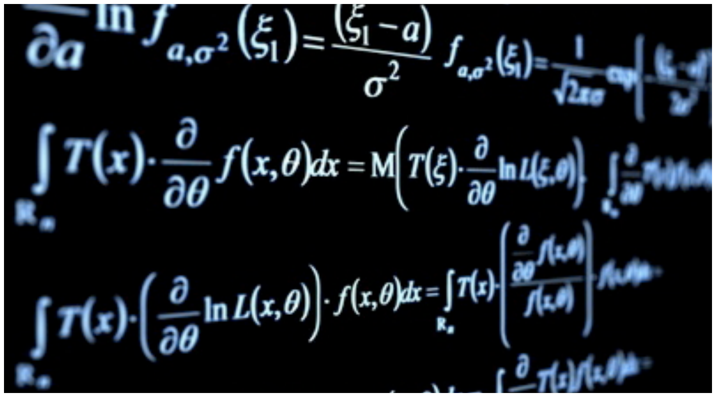
CλaSH

Compiling Circuit Descriptions

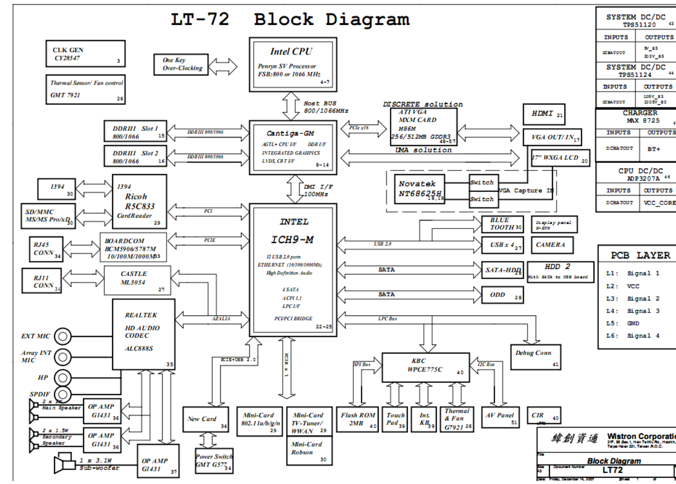
Overview

- What is CλaSH?
- The CλaSH compiler pipeline
- Transformations
- Structural vs Behavioural
- CλaSH vs Embedded
- Demo
- Conclusions

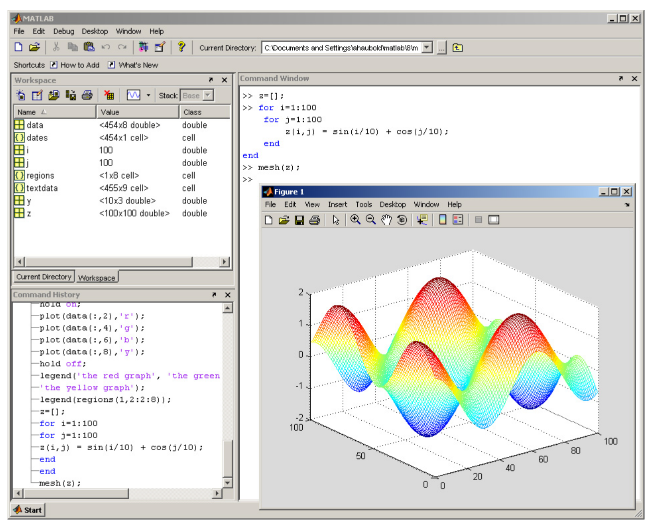
Hardware Design



math



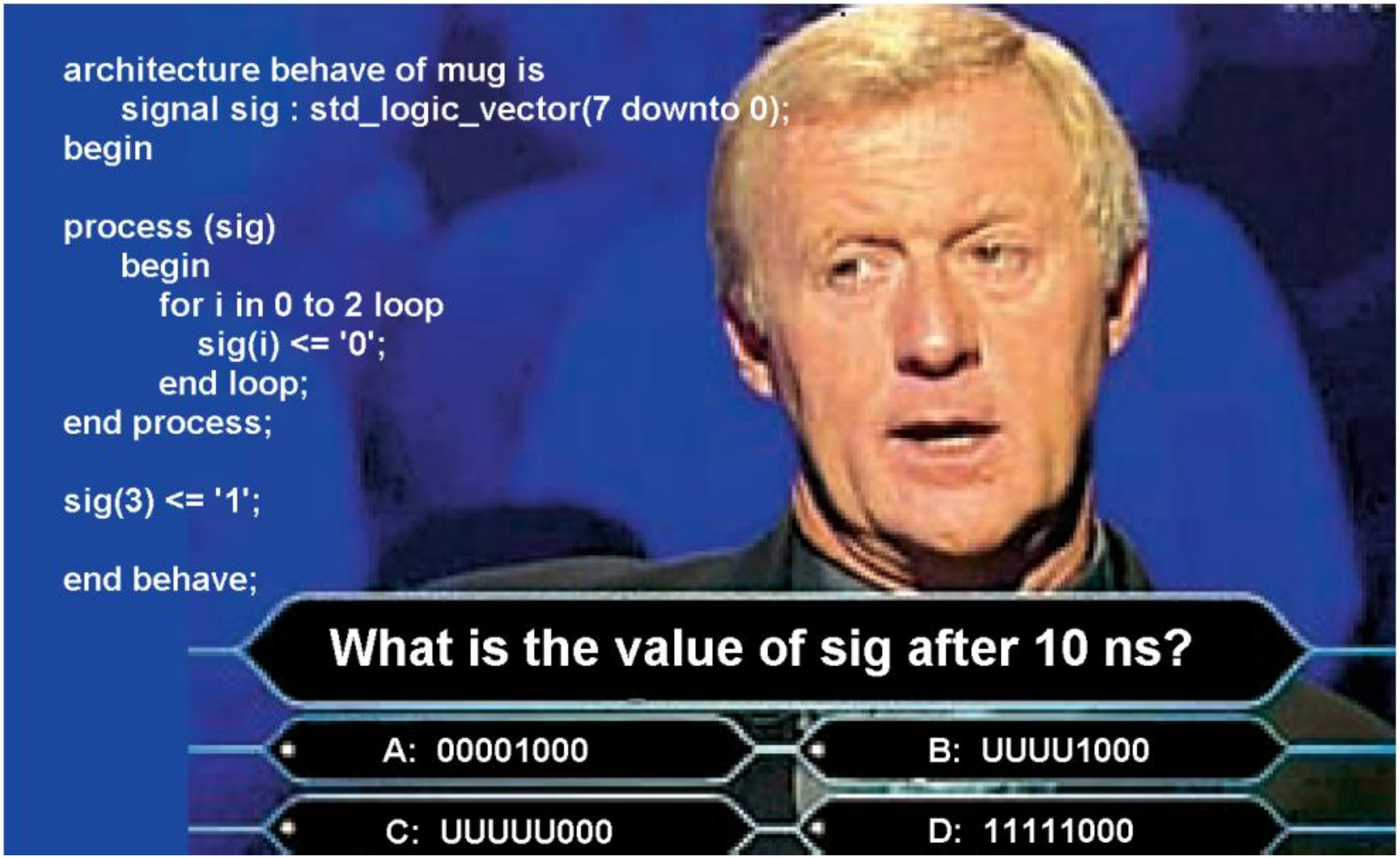
Schematic



Matlab/C

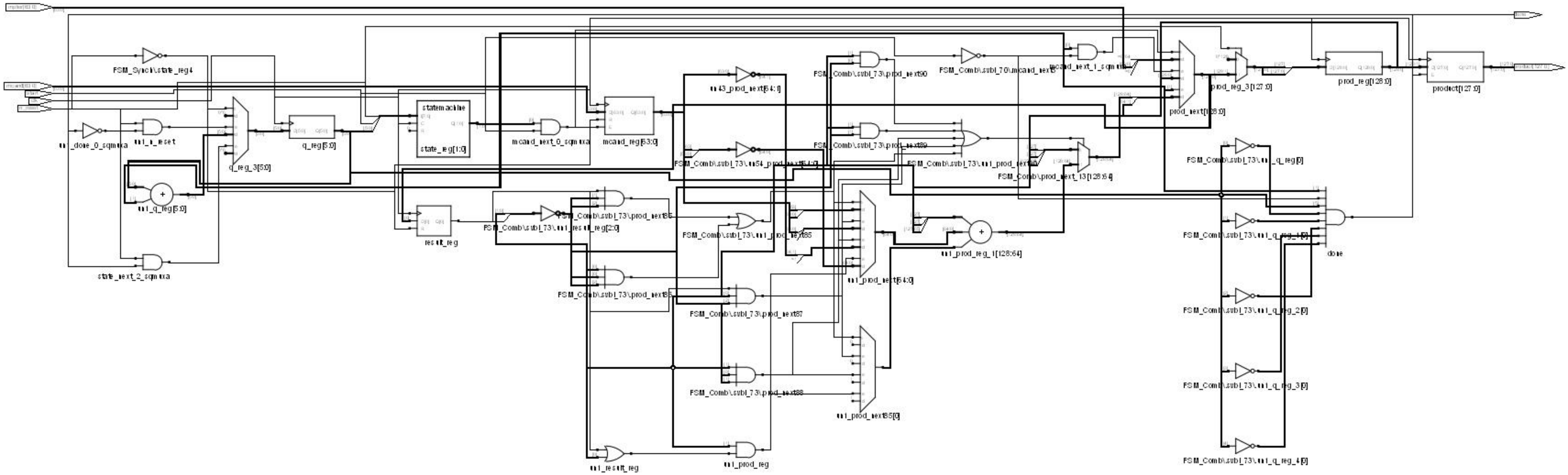
Hard work, LINT tools, test benches, staring at two 30” screens full of waveforms, some assertions (SVA) and linear logic (PSL), and a little magic.

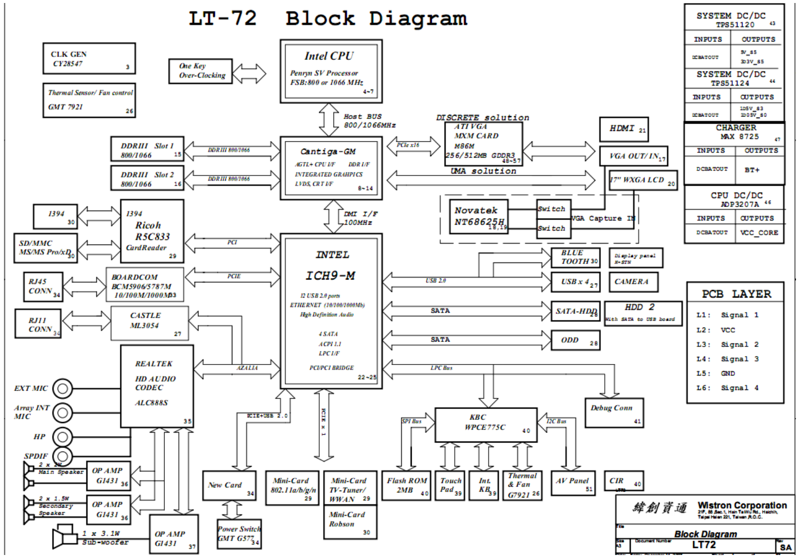
"Correct" VHDL



Synthesis

They're about the same





What is CλaSH?

▲ psychometry 140 days ago | link

I think we have a winner for the most poorly named programming language of the last decade. I thought that "Go" and "Hack" were bad, but at least I could type those on my keyboard.

▲ elliotec 140 days ago | link

Seriously. I spent more time trying to figure out how to read/say it than reading about it. C(lambda)aSH? clambdaash? Oh, CLaSH.

▲ dllthomas 140 days ago | link

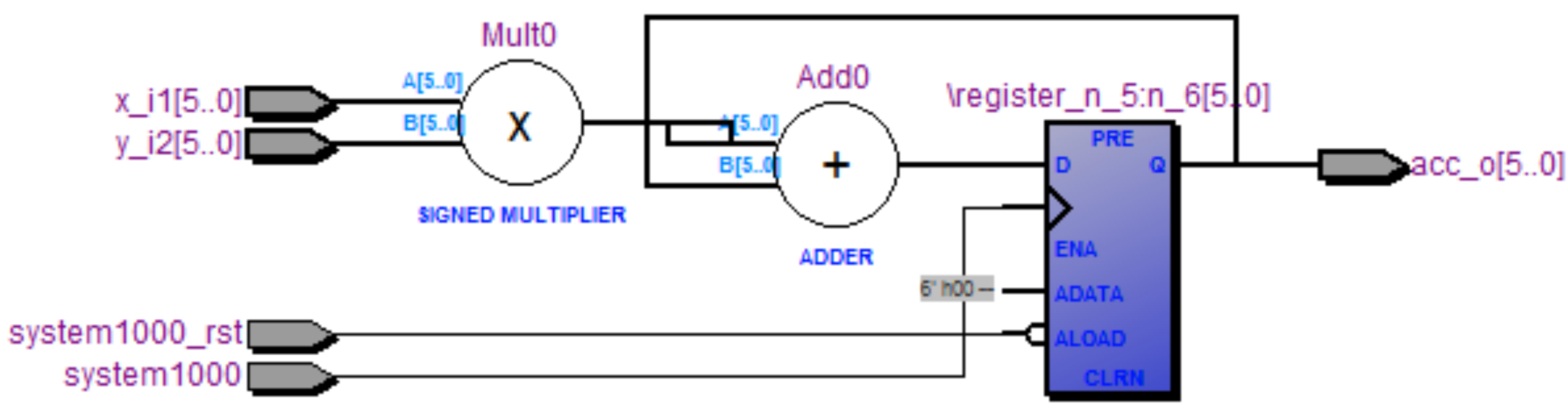
Clam-dash. It makes your shellfish go fast, or your car smell.

What is CλaSH?

- CAES Language for Synchronous Hardware
- A compiler that views Haskell programs as *structural* circuit descriptions.
- Input: (semantic) subset of Haskell
- Output: Low-level synthesisable VHDL

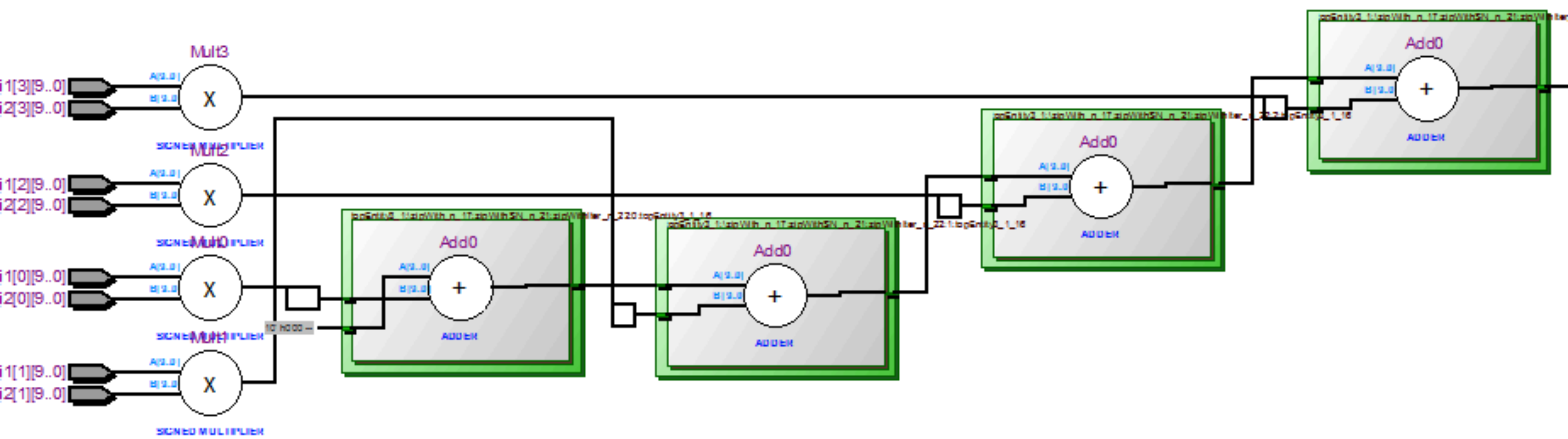
Small CλaSH example

```
mac :: Signal (Signed 6)
    -> Signal (Signed 6)
    -> Signal (Signed 6)
mac x y = acc
  where
    acc = register 0 ((x * y) + acc)
```



Another example

```
dotp :: Vec 4 (Signed 10)
     -> Vec 4 (Signed 10)
     -> Signed 10
dotp xs ys = foldl (+) 0 $ zipWith (*) xs ys
```



Compiler Pipeline

- Use GHC API to convert Haskell to GHC's Core
  - Custom set of optimizations enabled
  - No CorePrep (Breaks up Integer literals)
- Transform to CλaSH's Core (no coercions)
- Normalize CλaSH Core
- Convert to Netlist datatype
  - Includes primitive handling
- "Pretty" print VHDL

Core: System F'ish + letrec + case

```
data Term
  = Var     Type TmName                        -- ^ Variable reference
  | Data    DataCon                            -- ^ Datatype constructor
  | Literal Literal                            -- ^ Literal
  | Prim    Text Type                          -- ^ Primitive
  | Lam     (Bind Id Term)                     -- ^ Term-abstraction
  | TyLam   (Bind TyVar Term)                  -- ^ Type-abstraction
  | App     Term Term                          -- ^ Application
  | TyApp   Term Type                          -- ^ Type-application
  | Letrec  (Bind (Rec [LetBinding]) Term) -- ^ Recursive let-binding
  | Case    Term Type [Bind Pat Term]          -- ^ Case-expression

data Type
  = VarTy    Kind TyName         -- ^ Type variable
  | ConstTy  ConstTy             -- ^ Type constant
  | ForAllTy (Bind TyVar Type) -- ^ Polymorphic Type
  | AppTy    Type Type           -- ^ Type Application
  | LitTy    LitTy               -- ^ Type literal
```

Structure: an “operational” semantics

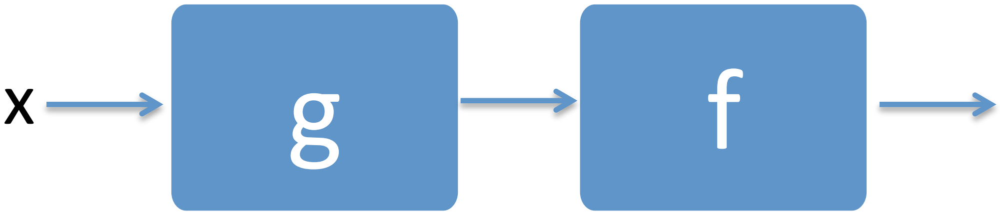
Syntactical element

- Function application
- Example:

```
f(g x)
```

Structure

- Component instantiation
- Example:



Structure: an “operational” semantics

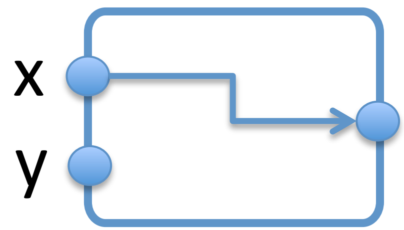
Syntactical element

- Lamda-binder
- Example:

```
\x y. x
```

Structure

- Input port
- Example:



Structure: an "operational" semantics

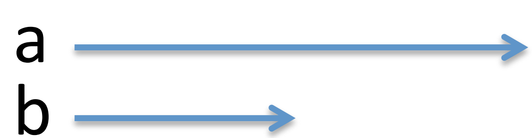
Syntactical element

- Case-statement on a product type, or, projection
- Example:

```
case z of (a,b) -> a
```

Structure

- Continue with one of the wires, basically a NOP
- Example:



Structure: an "operational" semantics

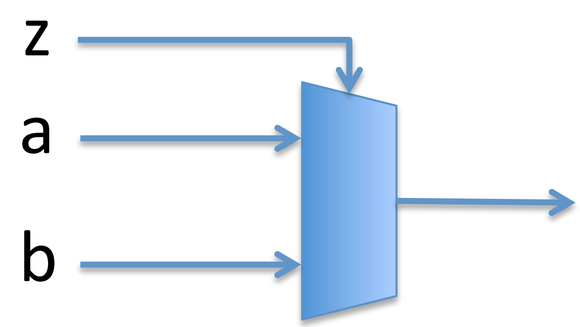
Syntactic element

- Case-statement on a sum-type.
- Example:

```
case z of
  True  -> a
  False -> b
```

Structure

- Multiplexer of the alternatives, with the subject driving the selection
- Example:



Normal form

- Only lamba's at outermost position
- Lambda's followed by a single letrec
- Let-bound values all in ANF
- Letrec body is a variable reference
- Completely monomorphic and first-order

Compilation example

Haskell

```
data Opc = Add | Mul

alu Add = (+)
alu Mul = (*)
```

Core

```
alu = \ds ->
  case ds of
    Add -> (+)
    Mul -> (*)
```

Compilation example

Core

```
alu = \ds ->
  case ds of
    Add -> (+)
    Mul -> (*)
```

Normalised Core

```
alu = \ds x y ->
 let a = (+) x y
     b = (*) x y
     r = case ds of
           Add -> a
           Mul -> b
 in res
```

Transformations

- Preserve first-order function hierarchy
- Don't lose sharing => larger circuit
- Open question: is case-of-case bad?

```
case (case e of {p1 -> a1 .. pN -> aN})
     {q1 -> b1 .. qN -> bN}
 =>
case e of {p1 -> case a1 of {q1 -> b1 .. qN -> bN}
           …
           pN -> case aN of {q1 -> b1 .. qN -> bN
          }
```

- Currently: only when the subject is higher-order

Primitive templates

```
[ { "BlackBox" :
    { "name"      : "CLaSH.Signal.Internal.register#”
    , "templateD" :
"register_~SYM[0] : block
  signal ~SYM[1] : ~TYP[2];
  signal ~SYM[2] : ~TYP[1];
begin
  ~SYM[2] <= ~ARG[1];
  process(~CLK[0],~RST[0],~SYM[2])
  begin
    if ~RST[0] = '0' then
      ~SYM[1] <= ~SYM[2];
    elsif rising_edge(~CLK[0]) then
      ~SYM[1] <= ~ARG[2];
    end if;
  end process;
  ~RESULT <= ~SYM[1];
end block;"
    }
  }
]
```

Embedded DSL / Lava

- Why CλaSH when we already have Lava?
- Building parallel circuits in Haskell for more than a decade
- “Straightforward” to implement:
  - Don’t have to deal with higher-order functions or recursions

PATTERN MATCHING!

EDSL vs Pattern Matching

- You can use pattern matching as part of a functions that generates a circuit.
- But you cannot use pattern matching as way to specify the behaviour of the circuit.

EDSL vs Pattern Matching

Not observable

```
f :: Bool -> Signal Int8
  -> Signal Int8
f True a  = a + 1
f False a = a - 1
```

Invalid pattern

```
f :: Signal Bool
  -> Signal Int8
  -> Signal Int8
f True  a = a + 1
f False a = a - 1
```

Pattern matching in CλaSH Example

```
-- generate clk enable signal
if sclSync then do
   cnt   .= clkCnt
   clkEn .= True
else if _slaveWait then do
   clkEn .= False
else do
   cnt   -= 1
   clkEn .= False

-- generate bus status controller
zoom busState (busStatusCtrl clkCnt cmd sdaChk
                             isdaOen _clkEn cState
                             i2ci)
```

Pattern matching in CλaSH Example

```
-- generate clk enable signal
if sclSync then do
   cnt   .= clkCnt
   clkEn .= True
else if _slaveWait then do
   clkEn .= False
else do
   cnt   -= 1
   clkEn .= False

-- generate bus status controller
zoom busState (busStatusCtrl clkCnt cmd sdaChk
                             isdaOen _clkEn cState
                             i2ci)
```

Demo

What isn't working

- GADT pattern matching
- Irreducible recursive functions cannot be translated
- Compiler is "really" slow and tends to eat up memory (8gb or over runs happen)
  - Efficient code as an afterthought doesn't work
  - Quadratic in the size of the let-binding
- Finding the right idiom for specifying hardware in Haskell

Future Work

- Inline HO-functions instead of specialisation
- Use a dependently-typed core to support e.g. hardware specifications using Idris.
  - How to merge with GHC's open type families?
- Get my ring-solver in good shape for inclusion in GHC 7.10
- Make the compiler faster:
  - Flatten MTL stack
  - Be more efficient with binders (perhaps use another library)
  - Make term type a triple of: (term,type,free variables)

Conclusions

- GHC API enables lots of fun projects and isn't that scary.
- Use CλaSH instead of Lava when you care about pattern matching.
- Stiil need to find an idiomatic way to write hardware in Haskell
- Stop writing VHDL/Verilog, use Haskell; CλaSH will do the rest.

Questions?

```
cabal install clash-ghc
```

Extensions used by the Prelude

```
other-extensions:   DataKinds
                    DefaultSignatures
                    DeriveDataTypeable
                    FlexibleContexts
                    GADTs
                    GeneralizedNewtypeDeriving
                    KindSignatures
                    MagicHash
                    MultiParamTypeClasses
                    ScopedTypeVariables
                    StandaloneDeriving
                    TemplateHaskell
                    TupleSections
                    TypeFamilies
                    TypeOperators
                    UndecidableInstances
```

Disabled GHC Optimizations

Name	Reason
Opt_SpecConstr	Creates local functions: normal form does not have them
Opt_DoEtaReduction	We want eta-expansion
Opt_PedanticBottoms	Stops eta-expansion through case-expressions