Digital hardware design with *Clash*

Jan Kuper
QBayLogic B.V., Enschede
jan.kuper@qbaylogic.com

Spring 2022 RISC-V Week, Paris

QBayLogic.

QBayLogic

- FPGA Design House; FPGA services — using *Clash*
- *Clash*: *Haskell* ⇒ *VHDL/Verilog* compiler; Open source
- Spinoff University of Twente (NL); based on 10 years of research
- Founded in 2016, 2 people; Currently: 14 people

QBayLogic

QBayLogic.

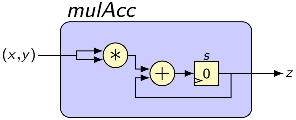
- FPGA Design House; FPGA services — using *Clash*
- *Clash*: *Haskell* ⇒ *VHDL/Verilog* compiler; Open source
- Spinoff University of Twente (NL); based on 10 years of research
- Founded in 2016, 2 people; Currently: 14 people

Projects

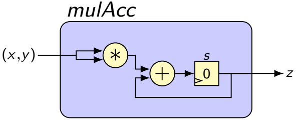
- Processor design (RISC-V)
- Simulations, Control systems (Adaptive cruise control)
- Accelerators (AI, Financial, Satellite communication[1])
- Memory controllers, Communication protocols

[1] Bits&Chips, september 2020: Jan Kuper (*QBayLogic*), Joost Kauffman (*Demcon-Focal*) – *High-level FPGA programming for nanosecond timing in terabit communication*

Clash: Functional Perspective



Clash: Functional Perspective



$mulAcc :: s \rightarrow i \rightarrow (s, o)$

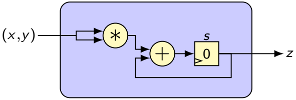
Mealy Machine

$mulAcc :: Signal\ dom\ i \rightarrow Signal\ dom\ o$

Signal function

Clash: Functional Perspective

QBayLogic.



$$mulAcc :: s \rightarrow i \rightarrow (s, o)$$

$$\begin{aligned} mulAcc\ s\ (x, y) &= (s', z) \\ &\quad \textbf{where} \\ &\qquad s' = s + x*y \\ &\qquad z = s \end{aligned}$$

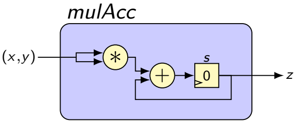
Mealy Machine

$$mulAcc :: Signal\ dom\ i \rightarrow Signal\ dom\ o$$

$$\begin{aligned} mulAcc\ xy &= z \\ &\quad \textbf{where} \\ &\qquad (x, y) = unbundle\ xy \\ &\qquad z = register\ 0\ (z + x*y) \end{aligned}$$

Signal function

Clash: Functional Perspective



$$mulAcc :: s \rightarrow i \rightarrow (s, o)$$

$$mulAcc\ s\ (x, y) = (s', z)$$
where
$$s' = s + x * y$$
$$z = s$$

Mealy Machine

mealy $\Longrightarrow$ *moore*

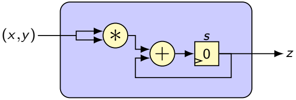
$$mulAcc :: Signal\ dom\ i \rightarrow Signal\ dom\ o$$

$$mulAcc\ xy = z$$
where
$$(x, y) = unbundle\ xy$$
$$z = register\ 0\ (z + x * y)$$

Signal function

Clash: Functional Perspective

QBayLogic.



$$mulAcc :: s \rightarrow i \rightarrow (s, o)$$

$$mulAcc\ s\ (x, y) = (s', z)$$
where
$$s' = s + x*y$$
$$z = s$$

Mealy Machine

mealy $\Longrightarrow$ *moore*

$$mulAcc :: Signal\ dom\ i \rightarrow Signal\ dom\ o$$

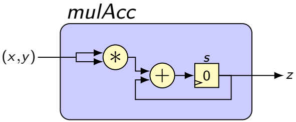
$$mulAcc\ xy = z$$
where
$$(x, y) = unbundle\ xy$$
$$z = register\ 0\ (z + x*y)$$

Signal function

- Powerful abstraction mechanisms
- Strong typing system
- Straightforward simulation/test
- Control over hardware details

Clash: Functional Perspective

QBayLogic.



$$mulAcc :: s \rightarrow i \rightarrow (s, o)$$

$$\begin{aligned} mulAcc\ s\ (x, y) &= (s', z) \\ &\quad \textbf{where} \\ &\quad\quad s' = s + x*y \\ &\quad\quad z = s \end{aligned}$$

Mealy Machine

mealy $\Longrightarrow$ *moore*

$$mulAcc :: Signal\ dom\ i \rightarrow Signal\ dom\ o$$

$$\begin{aligned} mulAcc\ xy &= z \\ &\quad \textbf{where} \\ &\quad\quad (x, y) = unbundle\ xy \\ &\quad\quad z = register\ 0\ (z + x*y) \end{aligned}$$

Signal function

- Powerful abstraction mechanisms
- Strong typing system
- Straightforward simulation/test
- Control over hardware details
- Model driven (one language: Haskell)
- Provable correctness
- Software *and* hardware
- Effective design process

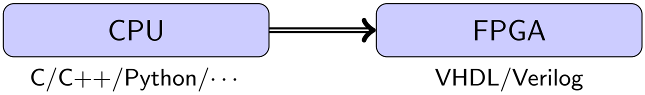
Design process

QBayLogic.

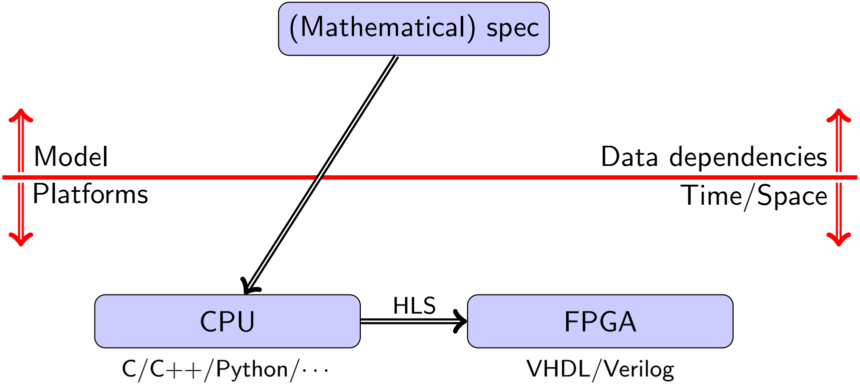
FPGA

VHDL/Verilog

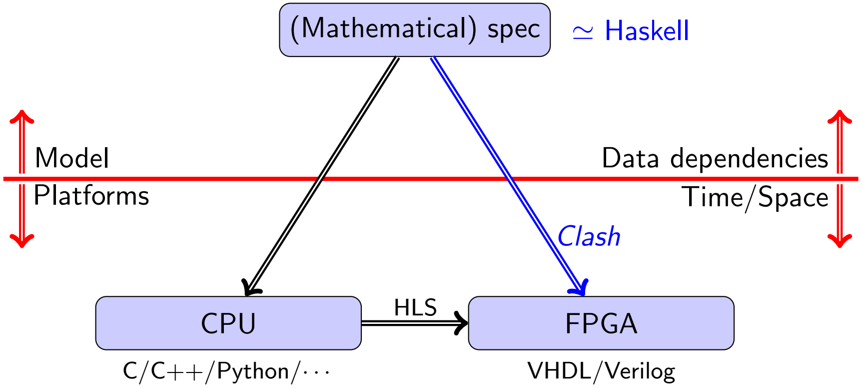
Design process



Design process



Design process



Example: IIR-filter

Medical application; Requirements (a.o.):

- FPGA: 300MHz
- 585 cycles/sample available
- Floating Point
- Number of arithmetical operators minimal

HLS failed ...

Example: IIR-filter

Medical application; Requirements (a.o.):

- FPGA: 300MHz
- 585 cycles/sample available
- Floating Point
- Number of arithmetical operators minimal

HLS failed ...

Results

	Number of operators	Pipeline stages
Multiplier	1	8
Adder	1	11

Taps IIR	Cycles
6	49
10	61
20	78

Freq: 550MHz

IIR: Formal model

IIR: Formal model

IIR: Formal model

IIR: Formal model

$$y_n = \frac{1}{a_0}\left(\sum_{i=0}^{N} b_i x_{n-i} \; - \; \sum_{j=1}^{M} a_j y_{n-j} \right)$$

IIR: Formal model

$$y_n = \frac{1}{a_0}\left(\sum_{i=0}^{N} b_i x_{n-i} \; - \; \sum_{j=1}^{M} a_j y_{n-j} \right)$$

$$\vdots$$

$$= c \left(0 \bigoplus b_{0\cdots N} \;\widehat{*}\; x_{n\cdots n-N} \; - \; 0 \bigoplus a_{0\cdots M-1} \;\widehat{*}\; y_{n-1\cdots n-M} \right)$$

IIR: Formal model

QBayLogic.

$$y_n = \frac{1}{a_0} \left(\sum_{i=0}^{N} b_i x_{n-i} - \sum_{j=1}^{M} a_j y_{n-j} \right)$$

⋮

$$= c \left(0 \bigoplus b_{0\cdots N} \,\widehat{*}\, x_{n\cdots n-N} - 0 \bigoplus a_{0\cdots M-1} \,\widehat{*}\, y_{n-1\cdots n-M} \right)$$

```
yA n  | n < 0       = 0
      | otherwise = c * (  foldl (+) 0 (zipWith (*) (b&[0..nn])   (x&[n,n-1..n-nn]))
                         - foldl (+) 0 (zipWith (*) (a&[0..mm-1]) (yA&[n-1,n-2..n-mm]))
                        )
```

IIR: Formal model

$$y_n = \frac{1}{a_0}\left(\sum_{i=0}^{N} b_i x_{n-i} - \sum_{j=1}^{M} a_j y_{n-j}\right)$$

$$\vdots$$

$$= c\left(0 \oplus b_{0\cdots N} \hat{*} x_{n\cdots n-N} - 0 \oplus a_{0\cdots M-1} \hat{*} y_{n-1\cdots n-M}\right)$$

- Word-for-word translation
- Haskell = Math
- Executable

Test:

```
testA = yA&[0..40]
```

Slow!

```
yA n  | n < 0       = 0
      | otherwise = c * ( foldl (+) 0 (zipWith (*) (b&[0..nn])   (x&[n,n-1..n-nn]))
                        - foldl (+) 0 (zipWith (*) (a&[0..mm-1]) (yA&[n-1,n-2..n-mm]))
                        )
```

IIR: Parameter accumulation

$$y_n \quad = c\left(0 \bigoplus b_{0\cdots N} \mathbin{\widehat{*}} x_{n\cdots n-N} \;-\; 0 \bigoplus a_{0\cdots M-1} \mathbin{\widehat{*}} y_{n-1\cdots n-M}\right)$$

IIR: Parameter accumulation

QBayLogic.

$$y_n = c\left(0 \bigoplus b_{0\cdots N} \mathbin{\widehat{*}} x_{n\cdots n-N} - 0 \bigoplus a_{0\cdots M-1} \mathbin{\widehat{*}} y_{n-1\cdots n-M}\right)$$

⇓

$$y_n\ (xs, ys) = yn : y_{n+1}\ (xs', ys')$$

Definitions:

$$us \cdot vs = 0 \bigoplus us \mathbin{\widehat{*}} vs$$

$$yn = c\left(bs \cdot xs - as \cdot ys\right)$$

$$xs' = x_{n+1} \mathbin{+\!\!\gg} xs$$

$$ys' = yn \mathbin{+\!\!\gg} ys$$

Proof of equivalence: induction on n

IIR: Parameter accumulation

$$y_n \;=\; c\left(0 \bigoplus b_{0\cdots N} \,\widehat{*}\, x_{n\cdots n-N} \;-\; 0 \bigoplus a_{0\cdots M-1} \,\widehat{*}\, y_{n-1\cdots n-M}\right)$$

$$\Downarrow$$

$$y_n\,(xs, ys) \;=\; yn \;:\; y_{n+1}\,(xs', ys')$$

Definitions:

$$us \cdot vs \;=\; 0 \bigoplus us \,\widehat{*}\, vs$$

$$yn \;=\; c\,(\,bs \cdot xs \;-\; as \cdot ys\,)$$

$$xs' = x_{n+1} \mathbin{+\!\!\gg} xs$$

$$ys' = yn \mathbin{+\!\!\gg} ys$$

Proof of equivalence: induction on n

```
us · vs = foldl (+) 0 (zipWith (*) us vs)

yB n (xs,ys) = y : yB (n+1) (xs',ys')
      where
        y   = c * (bs·xs - as·ys)
        xs' = x (n+1) +>> xs
        ys' = y +>> ys
```

Test:

```
testB = take 40 $ yB 0 (xs0,ys0)
```

Model = Golden reference

IIR: Recursor

QBayLogic.

$$y_n\ (xs, ys) \;=\; yn \;:\; y_{n+1}\ (xs', ys')$$

Definitions:

$$us \cdot vs = 0 \bigoplus us \mathbin{\widehat{*}} vs$$

$$yn = c\ (\ bs \cdot xs \;-\; as \cdot ys\ )$$

$$xs' = x_{n+1} \mathbin{+\!\!\gg} xs$$

$$ys' = yn \mathbin{+\!\!\gg} ys$$

IIR: Recursor

QBayLogic.

$$y_n\ (xs, ys) \;=\; yn \;:\; y_{n+1}\ (xs', ys')$$

Definitions:

$$us \cdot vs \;=\; 0 \bigoplus us \mathbin{\widehat{*}} vs$$

$$yn \;=\; c\ (\ bs \cdot xs \;-\; as \cdot ys\ )$$

$$xs' = x_{n+1} \mathbin{+\!\!\gg} xs$$

$$ys' = yn \mathbin{+\!\!\gg} ys$$

one-step function ⟹

$$y^1\ (xs, ys)\ x_{n+1} \;=\; \langle\, (xs', ys'),\ yn\, \rangle$$

Definitions:

$$us \cdot vs \;=\; 0 \bigoplus us \mathbin{\widehat{*}} vs$$

$$yn \;=\; c\ (\ bs \cdot xs \;-\; as \cdot ys\ )$$

$$xs' = x_{n+1} \mathbin{+\!\!\gg} xs$$

$$ys' = yn \mathbin{+\!\!\gg} ys$$

Recursor ⟹

$$\mathcal{R}\,(y^1)$$

Proof of equivalence: induction on n

IIR: Recursor

QBayLogic.

$$y_n\,(xs, ys) \;=\; yn \;:\; y_{n+1}\,(xs', ys')$$

Definitions:

$$us \cdot vs \;=\; 0 \bigoplus us \,\widehat{*}\, vs$$
$$yn \;=\; c\,(\,bs \cdot xs \;-\; as \cdot ys\,)$$
$$xs' = x_{n+1} \mathbin{+\!\!\gg} xs$$
$$ys' = yn \mathbin{+\!\!\gg} ys$$

one-step function ⟹

$$y^1\,(xs, ys)\; x_{n+1} \;=\; \langle\,(xs', ys'),\; yn\,\rangle$$

Definitions:

$$us \cdot vs \;=\; 0 \bigoplus us \,\widehat{*}\, vs$$
$$yn \;=\; c\,(\,bs \cdot xs \;-\; as \cdot ys\,)$$
$$xs' = x_{n+1} \mathbin{+\!\!\gg} xs$$
$$ys' = yn \mathbin{+\!\!\gg} ys$$

Recursor ⟹

$$\mathcal{R}\,(y^1)$$

```
us · vs = foldl (+) 0 (zipWith (*) us vs)

yC (xs,ys) x = ( (xs',ys') , y )
      where
        y   = c * (bs·xs - as·ys)
        xs' = x +>> xs
        ys' = y +>> ys
```

```
sim yC
```

Proof of equivalence: induction on n

Test:

```
testC = sim yC (xs0,ys0) (x&[1..40])
```

IIR: Recursor

- **Mealy Machine ⇒ Hardware**
- **Translatable to VHDL by *Clash***
- **Structure preserving**

```
us · vs = foldl (+) 0 (zipWith (*) us vs)

yC (xs,ys) x = ( (xs',ys') , y )
      where
        y   = c * (bs·xs - as·ys)
        xs' = x +>> xs
        ys' = y +>> ys
```

```
sim yC
```

```
:vhdl

:verilog
```

IIR: Architecture

```
yC (xs,ys) x = ( (xs',ys') , y )
      where
        y   = c * (bs·xs - as·ys)
        xs' = x +>> xs
        ys' = y +>> ys
```

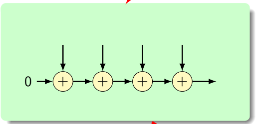
```
us · vs = foldl (+) 0 (zipWith (*) us vs)
```

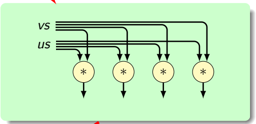
IIR: Architecture

QBayLogic.

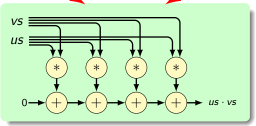
```
yC (xs,ys) x = ( (xs',ys') , y )
      where
        y   = c * (bs·xs - as·ys)
        xs' = x +>> xs
        ys' = y +>> ys
```

```
us · vs = foldl (+) 0 (zipWith (*) us vs)
```





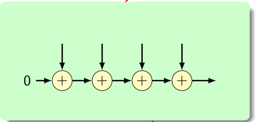
Dot product:

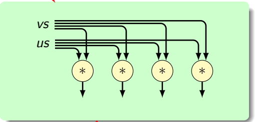


IIR: Architecture

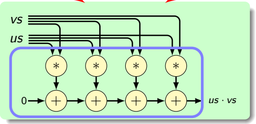
```haskell
yC (xs,ys) x = ( (xs',ys') , y )
      where
          y   = c * (bs·xs - as·ys)
          xs' = x +>> xs
          ys' = y +>> ys
```

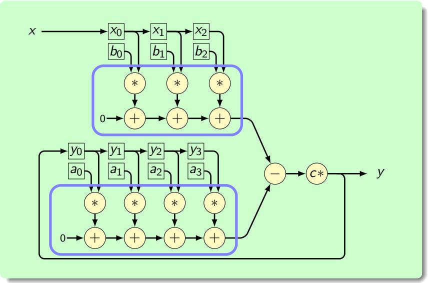
```haskell
us · vs = foldl (+) 0 (zipWith (*) us vs)
```





Dot product:



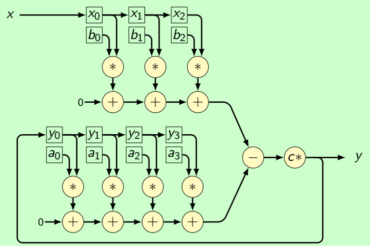


Performance characteristics:

- Area: many adders, multipliers
- Clock: longest path

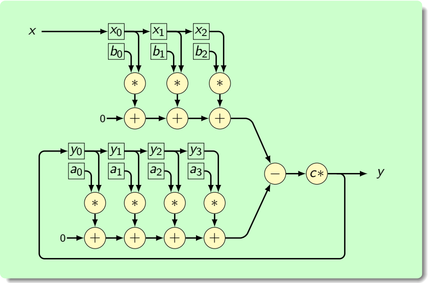
⇒ Optimisations needed

IIR: Linearisation

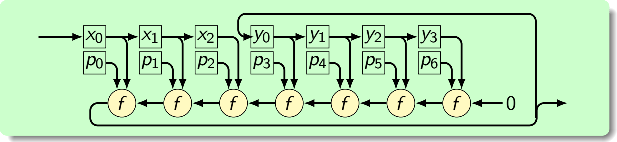


$$y \;=\; c\,(bs \bullet xs \;-\; as \bullet ys)$$

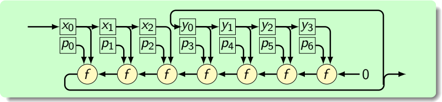
IIR: Linearisation



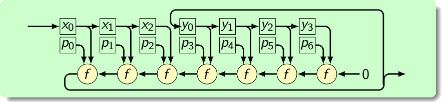
$$
\begin{aligned}
y &= c\,(bs \bullet xs \;-\; as \bullet ys) \\
  &= c\,((bs \mathbin{+\!\!+} -as) \bullet (xs \mathbin{+\!\!+} ys)) \\
  &= (c\,(bs \mathbin{+\!\!+} -as)) \bullet (xs \mathbin{+\!\!+} ys) \\
  &= ps \bullet xys \\
  &= \mathbf{foldl}\ (+)\ 0\ (\mathbf{zipWith}\ (*)\ ps\ xys) \\
  &= \mathbf{foldl}\ ((+) \triangleleft (*))\ 0\ pxys \\
  &= \mathbf{foldl}\ f\ 0\ pxys \\
  &= \mathbf{foldr}\ f\ 0\ pxys
\end{aligned}
$$

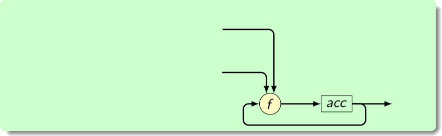


IIR: Sequentialising over time



IIR: Sequentialising over time

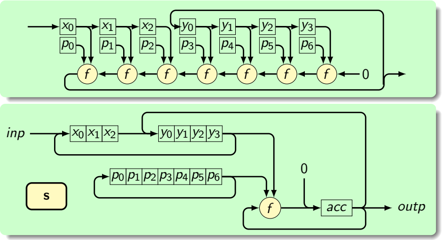




- Standard transformation
- Standard code patterns

IIR: Sequentialising over time

QBayLogic



	s	*ps*	*xs*	*ys*	*acc*	*outp*
Idle	(1)	–	(2)	–	–	–
Calc	(3)	$p_\ell \mathbin{+\!\!\!\gg} ps$	(4)	$x_\ell \mathbin{+\!\!\!\gg} ys$	(5)	–
Ready	**Idle**	–	–	$acc^\bullet \mathbin{+\!\!\!\gg} ys$	0	*acc*

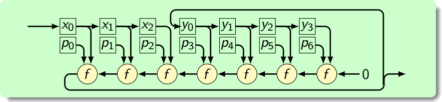
inp	(1)	(2)
x	**Calc**	$x^\circ \mathbin{+\!\!\!\gg} xs$
–	**Idle**	–

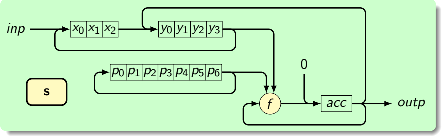
y_ℓ	(3)	(4)	(5)
$y^\bullet$	**Calc**	$y^\bullet \mathbin{+\!\!\!\gg} xs$	$acc + p_\ell * y$
y°	**Ready**	$y^\bullet \mathbin{+\!\!\!\gg} xs$	$acc + p_\ell * y$

Proof: invariant + induction

- Standard transformation
- Standard code patterns
- State machine

IIR: Sequentialising over time





	s	*ps*	*xs*	*ys*	*acc*	*outp*
Idle	(1)	–	(2)	–	–	–
Calc	(3)	$p_\ell \mathbin{+\!\!>\!\!>} ps$	(4)	$x_\ell \mathbin{+\!\!>\!\!>} ys$	(5)	–
Ready	**Idle**	–	–	$acc^\bullet \mathbin{+\!\!>\!\!>} ys$	0	*acc*

inp	(1)	(2)
x	**Calc**	$x^\circ \mathbin{+\!\!>\!\!>} xs$
–	**Idle**	–

y_ℓ	(3)	(4)	(5)
$y^\bullet$	**Calc**	$y^\bullet \mathbin{+\!\!>\!\!>} xs$	$acc + p_\ell * y$
y°	**Ready**	$y^\bullet \mathbin{+\!\!>\!\!>} xs$	$acc + p_\ell * y$

Proof: invariant + induction

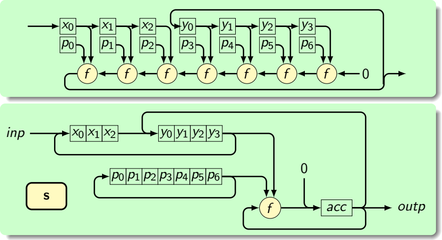
```
yCseq (s,ps,xs,ys,acc) inp = ((s',ps',xs',ys',acc'), outp )
  where
    (                 s'  , ps'   , xs'            , ys'              , acc'   , outp      )
     = case s      of
       -- ==============================================================================
            Idle  -> ( s_  , ps    , xs_            , ys               , acc    , Nothing  )  where (                    s_   , xs_          )
                                                                                                   = case inp     of
                                                                                                     -- ===================================
                                                                                                        Just x   -> ( Calc , New x +>> xs )
                                                                                                        Nothing  -> ( Idle , xs           )

            Calc  -> ( s_  , p+>>ps, Prev y +>> xs, last xs +>> ys , acc+p*y, Nothing  )  where p = last ps

                                                                                                  (                    s_   , y            )
                                                                                                   = case last ys of
                                                                                                     -- ===================================
                                                                                                        Prev v   -> ( Calc , v            )
                                                                                                        New v    -> ( Ready, v            )

            Ready -> ( Idle, ps    , xs            , Prev acc +>> ys, 0      , Just acc )
```

IIR: Sequentialising over time



	s	ps	xs	ys	acc	outp
Idle	(1)	–	(2)	–	–	–
Calc	(3)	$p_\ell +\!\!\gg ps$	(4)	$x_\ell +\!\!\gg ys$	(5)	–
Ready	**Idle**	–	–	$acc^\bullet +\!\!\gg ys$	0	acc

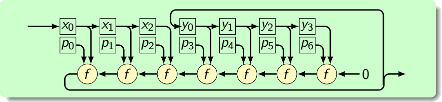
inp	(1)	(2)
x	**Calc**	$x^\circ +\!\!\gg xs$
–	**Idle**	–

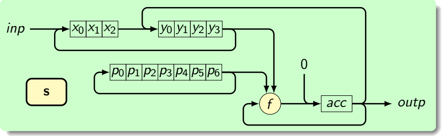
y_ℓ	(3)	(4)	(5)
$y^\bullet$	**Calc**	$y^\bullet +\!\!\gg xs$	$acc + p_\ell * y$
y°	**Ready**	$y^\bullet +\!\!\gg xs$	$acc + p_\ell * y$

Proof: invariant + induction

```
yCseq (s,ps,xs,ys,acc) inp = ((s',ps',xs',ys',acc'), outp )
  where
    (                          s'   , ps'    , xs'          , ys'            , acc'     , outp      )
     = case s     of
     --  ==========================================================================================
           Idle   -> ( s_   , ps     , xs_          , ys             , acc      , Nothing   )    where (                      s_     , xs_
                                                                                                     = case inp     of
                                                                                                     --  ==============================
                                                                                                           Just x   -> ( Calc , New x +>> xs )
                                                                                                           Nothing  -> ( Idle , xs           )
           Calc   -> ( s_   , p+>>ps , Prev y +>> xs, last xs +>> ys , acc+p*y  , Nothing   )    where p = last ps
                                                                                                    (                      s_     , y
                                                                                                     = case last ys of
                                                                                                     --  ==============================
                                                                                                           Prev v   -> ( Calc , v
                                                                                                           New v    -> ( Ready, v
           Ready  -> ( Idle , ps     , xs           , Prev acc +>> ys, 0        , Just acc  )
```

IIR: Sequentialising over time





	s	ps	xs	ys	acc	outp
Idle	(1)	–	(2)	–	–	–
Calc	(3)	$p_\ell \mathbin{+\!\!\gg} ps$	(4)	$x_\ell \mathbin{+\!\!\gg} ys$	(5)	–
Ready	**Idle**	–	–	$acc^\bullet \mathbin{+\!\!\gg} ys$	0	acc

inp	(1)	(2)
x	**Calc**	$x^\circ \mathbin{+\!\!\gg} xs$
–	**Idle**	–

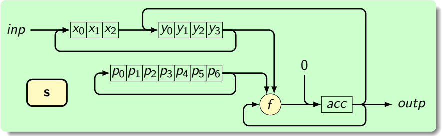
y_ℓ	(3)	(4)	(5)
$y^\bullet$	**Calc**	$y^\bullet \mathbin{+\!\!\gg} xs$	$acc + p_\ell * y$
y°	**Ready**	$y^\bullet \mathbin{+\!\!\gg} xs$	$acc + p_\ell * y$

Proof: invariant + induction

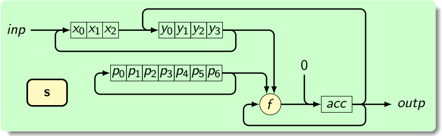
Test: `testCseq = ...`

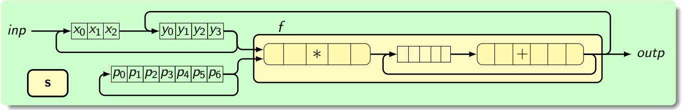
```
yCseq (s,ps,xs,ys,acc) inp = ((s',ps',xs',ys',acc'), outp )
  where
    (              s'    , ps'     , xs'           , ys'             , acc'    , outp     )
      = case s of
      --
        Idle  -> ( s_    , ps      , xs_           , ys              , acc     , Nothing  )
              where ( s_   , xs_
                      = case inp of
                      --
                        Just x  -> ( Calc , New x +>> xs
                        Nothing -> ( Idle , xs
        Calc  -> ( s_    , p+>>ps  , Prev y +>> xs , last xs +>> ys  , acc+p*y , Nothing  )
              where p = last ps
                    ( s_   , y
                      = case last ys of
                      --
                        Prev v  -> ( Calc , v
                        New v   -> ( Ready, v
        Ready -> ( Idle  , ps      , xs            , Prev acc +>> ys , 0       , Just acc )
```

IIR: Pipelining



IIR: Pipelining





- Pipelined multiplier, adder
- Predefined block (with feedback, priority rules)
  Processes input continuously; various input sequences
- Proven correctness, incl buffer behaviour
- Slightly modified state machine
- Pipeline depth expressable in *type* (*DSignal*, parameterisable)

HDL generation

- Typing: Polymorphic $\Rightarrow$ monomorphic
- Define *topEntity*
- Commands: `:vhdl`, `:verilog`
- Compilation is architecture preserving
- Simulation of VHDL/Verilog: not necessary

Types

Basic types: **Bit, Int, Char, Bool**

Number types: **Unsigned** *n*, **Signed** *n*, **UFixed** *m n*, **SFixed** *m n*, **Float**

Function types: $a \rightarrow b$

Vector types: **Vec** *n a*, **BitVector** *n*

Signal types: **Signal** *dom a*, **DSignal** *dom d a*

Tuples, Records, Algebraic types, ...

Types

Basic types: **Bit, Int, Char, Bool**

Number types: **Unsigned** *n*, **Signed** *n*, **UFixed** *m n*, **SFixed** *m n*, **Float**

Function types: $a \rightarrow b$

Vector types: **Vec** *n a*, **BitVector** *n*

Signal types: **Signal** *dom a*, **DSignal** *dom d a*

Tuples, Records, Algebraic types, ...

```
head   :: Vec (n+1) a  ->  a
concat :: Vec n (Vec m a) -> Vec (n*m) a
```

Types

QBayLogic.

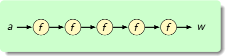
Basic types:	**Bit, Int, Char, Bool**
Number types:	**Unsigned** *n*, **Signed** *n*, **UFixed** *m n*, **SFixed** *m n*, **Float**
Function types:	$a \to b$
Vector types:	**Vec** *n a*, **BitVector** *n*
Signal types:	**Signal** *dom a*, **DSignal** *dom d a*

Tuples, Records, Algebraic types, . . .

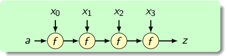
```
head   :: Vec (n+1) a  →  a
concat :: Vec n (Vec m a) → Vec (n*m) a
```

- Polymorphic type checking (theorem proving) at compile time
- Choose for monomorphic type for translation to VHDL/Verilog

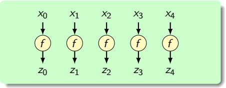
Higher Order Functions



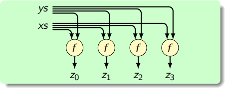
iterate n f a



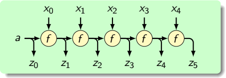
foldl f a xs



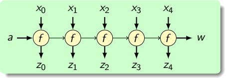
map f xs



zipWith f xs ys



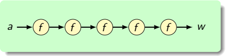
scanl f a xs



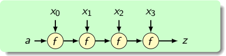
mapAccumL f a xs

- HOFs $\Rightarrow$ (for-)loops
- HOFs = structure
- Data dependencies known, no reverse engineering

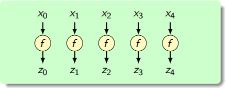
Higher Order Functions



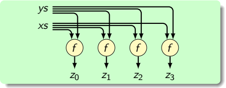
iterate n f a



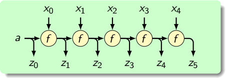
foldl f a xs



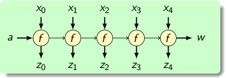
map f xs



zipWith f xs ys



scanl f a xs



mapAccumL f a xs

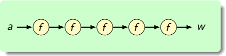
- HOFs $\Rightarrow$ (for-)loops
- HOFs $=$ structure
- Data dependencies known, no reverse engineering

$$foldl\ :: (a \rightarrow b \rightarrow a) \rightarrow a \rightarrow \textbf{Vec}\ n\ b \rightarrow a$$

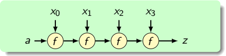
$$scanl :: (a \rightarrow b \rightarrow a) \rightarrow a \rightarrow \textbf{Vec}\ n\ b \rightarrow \textbf{Vec}\ (n{+}1)\ a$$

Higher Order Functions

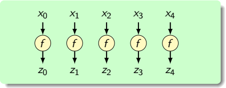
QBayLogic.



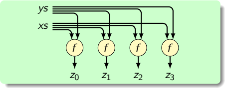
iterate n f a



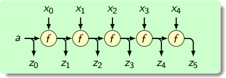
foldl f a xs



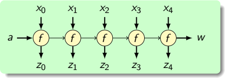
map f xs



zipWith f xs ys



scanl f a xs

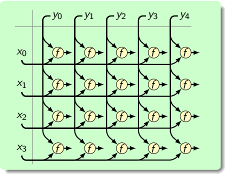


mapAccumL f a xs

- HOFs $\Rightarrow$ (for-)loops
- HOFs $=$ structure
- Data dependencies known, no reverse engineering

$$foldl \; :: \; (a \to b \to a) \to a \to \mathbf{Vec} \; n \; b \to a$$

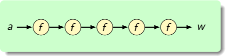
$$scanl \; :: \; (a \to b \to a) \to a \to \mathbf{Vec} \; n \; b \to \mathbf{Vec} \; (n{+}1) \; a$$



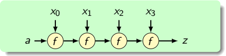
cartProdWith f xs ys

$$cartProdWith \; :: \; (a \to b \to c) \to \mathbf{Vec} \; n \; a \to \mathbf{Vec} \; m \; b \to \mathbf{Vec} \; n \; (\mathbf{Vec} \; m \; c)$$

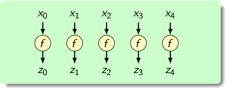
Higher Order Functions



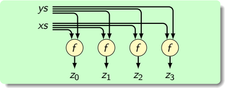
iterate n f a



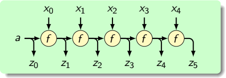
foldl f a xs



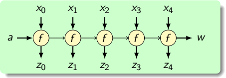
map f xs



zipWith f xs ys



scanl f a xs

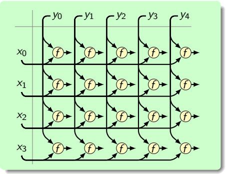


mapAccumL f a xs

- HOFs ⇒ (for-)loops
- HOFs = structure
- Data dependencies known, no reverse engineering

$$foldl\ :: (a \to b \to a) \to a \to \mathbf{Vec}\ n\ b \to a$$

$$scanl\ :: (a \to b \to a) \to a \to \mathbf{Vec}\ n\ b \to \mathbf{Vec}\ (n{+}1)\ a$$

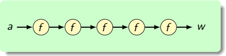


cartProdWith f xs ys

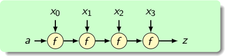
$$cartProdWith :: (a \to b \to c) \to \mathbf{Vec}\ n\ a \to \mathbf{Vec}\ m\ b \to \mathbf{Vec}\ n\ (\mathbf{Vec}\ m\ c)$$

Matrix multiplication: $m_0 \times m_1 = cartProdWith\ (\bullet)\ m_0\ (transpose\ m_1)$

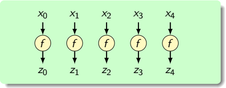
Higher Order Functions



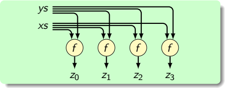
iterate n f a



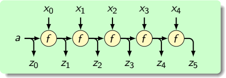
foldl f a xs



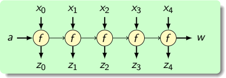
map f xs



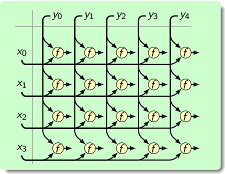
zipWith f xs ys



scanl f a xs

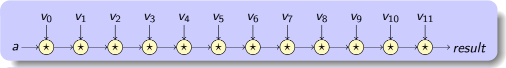


mapAccumL f a xs



cartProdWith f xs ys

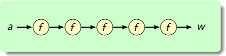
- Provable loop transformations



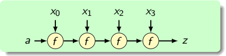
$$result = foldl\ f\ a\ vs$$

Higher Order Functions

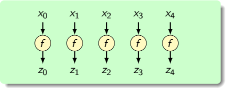
QBayLogic.



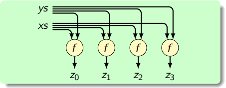
iterate n f a



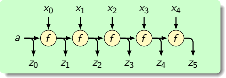
foldl f a xs



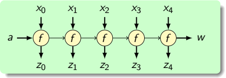
map f xs



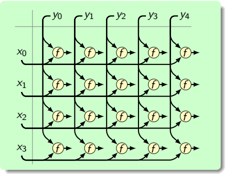
zipWith f xs ys



scanl f a xs

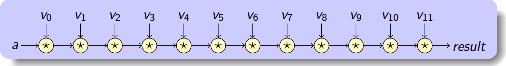


mapAccumL f a xs

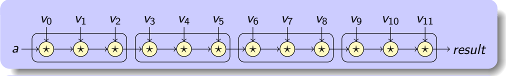


cartProdWith f xs ys

- Provable loop transformations

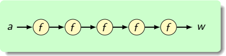


$$result = foldl\ f\ a\ vs$$

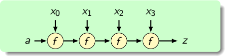


$$result = foldl\ (foldl\ f)\ a\ vss$$

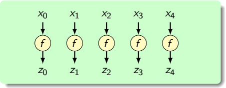
Higher Order Functions



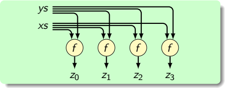
iterate n f a



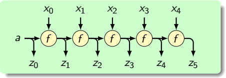
foldl f a xs



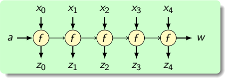
map f xs



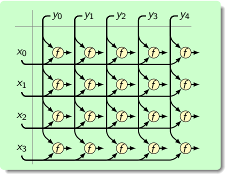
zipWith f xs ys



scanl f a xs

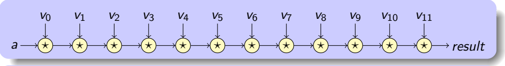


mapAccumL f a xs

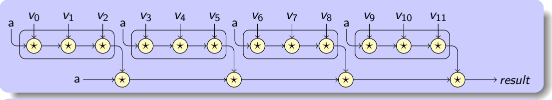


cartProdWith f xs ys

- Provable loop transformations



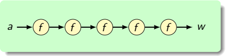
$$result = foldl\ f\ a\ vs$$



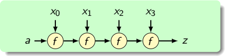
associative, neutral element

$$result = foldl\ f\ a\ (map\ (foldl\ f\ a)\ vss)$$

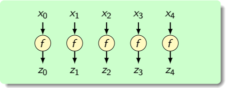
Higher Order Functions



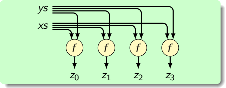
iterate n f a



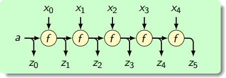
foldl f a xs



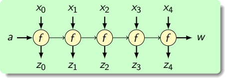
map f xs



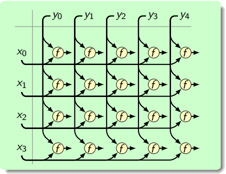
zipWith f xs ys



scanl f a xs

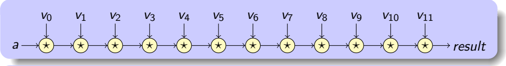


mapAccumL f a xs

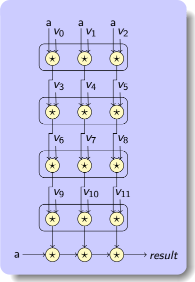


cartProdWith f xs ys

- Provable loop transformations



$$result \;=\; foldl\ f\ \ a\ vs$$



associative, commutative, neutral element

$$result \;=\; foldl\ f\ \ a\ pts$$

where

$$zs \;=\; replicate\ m\ a$$
$$pts \;=\; foldl\ (zipWith\ f\,)\ zs\ vss$$

Algebraic Data Types = Embedded Languages

Algebraic types: *Constructors* + *Arguments*

Algebraic Data Types = Embedded Languages

Algebraic types: *Constructors* + *Arguments*

```
type PC          = Unsigned 8
type Nmbr        = Signed   32
type Addr        = Unsigned 10

data Instruction = Write Addr Nmbr
                 | Move Addr Addr
                 | Add Addr Addr Addr
                 | Pred Addr Addr
                 | Eq0 Addr
                 | Jump PC
                 | CJump PC
                 | Stop
```

Algebraic Data Types = Embedded Languages

QBayLogic.

Algebraic types: *Constructors* + *Arguments*

$$
\begin{array}{lcl}
\textbf{type}\ PC & = & Unsigned\ 8 \\
\textbf{type}\ Nmbr & = & Signed\ \ 32 \\
\textbf{type}\ Addr & = & Unsigned\ 10 \\
\\
\textbf{data}\ Instruction & = & Write\ Addr\ Nmbr \\
 & | & Move\ Addr\ Addr \\
 & | & Add\ Addr\ Addr\ Addr \\
 & | & Pred\ Addr\ Addr \\
 & | & Eq0\ Addr \\
 & | & Jump\ PC \\
 & | & CJump\ PC \\
 & | & Stop
\end{array}
$$

Add 4 5 12

CJump 8

Algebraic Data Types = Embedded Languages

QBayLogic.

Algebraic types: *Constructors* + *Arguments*

```
type PC           =  Unsigned 8
type Nmbr         =  Signed 32
type Addr         =  Unsigned 10

data Instruction  =  Write Addr Nmbr
                  |  Move Addr Addr
                  |  Add Addr Addr Addr
                  |  Pred Addr Addr
                  |  Eq0 Addr
                  |  Jump PC
                  |  CJump PC
                  |  Stop
```

```
Add 4 5 12

CJump 8
```

- Embedded language = (algebraic) data type
- Readability; Pattern matching
- Processors, State machines, Routers, Protocols
- Default bit en-/decoding by *Clash*; customisation possible

Algebraic Data Types = Embedded Languages

Algebraic types: *Constructors* + *Arguments*

type *PC* = *Unsigned* 8
type *Nmbr* = *Signed* 32
type *Addr* = *Unsigned* 10

data *Instruction* = *Write Addr Nmbr*
| *Move Addr Addr*
| *Add Addr Addr Addr*
| *Pred Addr Addr*
| *Eq0 Addr*
| *Jump PC*
| *CJump PC*
| *Stop*

Add 4 5 12

CJump 8

- Embedded language = (algebraic) data type
- Readability; Pattern matching
- Processors, State machines, Routers, Protocols
- Default bit en-/decoding by *Clash*; customisation possible

Semantics, specification:

instrSem instr :: *State* → *State*

Algebraic Data Types = Embedded Languages

QBayLogic.

Algebraic types: *Constructors* + *Arguments*

type *PC* = *Unsigned* 8
type *Nmbr* = *Signed* 32
type *Addr* = *Unsigned* 10

data *Instruction* = *Write Addr Nmbr*
| *Move Addr Addr*
| *Add Addr Addr Addr*
| *Pred Addr Addr*
| *Eq0 Addr*
| *Jump PC*
| *CJump PC*
| *Stop*

Add 4 5 12

CJump 8

- Embedded language = (algebraic) data type
- Readability; Pattern matching
- Processors, State machines, Routers, Protocols
- Default bit en-/decoding by *Clash*; customisation possible

Semantics, specification:

instrSem instr :: *State* → *State*

instrSem :: *Instruction* → *State* → *State*

Instructions: specification

$$\textbf{type } \mathit{Mem} = \textbf{Vec } 1024 \ \mathit{Nmbr}$$
$$\textbf{type } \mathit{State} = (\mathit{Mem}, \mathit{PC})$$

Instructions: specification

type *Mem* = **Vec** 1024 *Nmbr*

type *State* = (*Mem*, *PC*)

```
instrSem :: Instruction -> State -> State

instrSem instr (mem,pc) = case instr of  -- =============================================
                                         --   mem                           pc
                                         -----------------------------------------------
          Write z n    -> ( mem <~ (z, n               ), pc+1             )
          Move  a z    -> ( mem <~ (z, mem!!a          ), pc+1             )

          Add   a b z  -> ( mem <~ (z, mem!!a + mem!!b), pc+1             )
          Pred  a z    -> ( mem <~ (z, mem!!a - 1      ), pc+1             )

          Eq0   a      -> ( mem <~ (0, mem!!a === 0    ), pc+1             )

          Jump  i      -> ( mem                         , i                )
          CJump i      -> ( mem                         , if   mem!!0 == 1
                                                          then i
                                                          else pc+1        )

          End          -> ( mem                         , pc               )
```

Instructions: specification

QBayLogic.

type *Mem* = **Vec** 1024 *Nmbr*

type *State* = (*Mem*, *PC*)

type *Program* = [*Instruction*]

```
instrSem :: Instruction -> State -> State

instrSem instr (mem,pc) = case instr of  -- ==========================================
                                         --   mem                        pc
                                         ------------------------------------------
  Write z n   -> ( mem <~ (z, n               ), pc+1            )
  Move  a z   -> ( mem <~ (z, mem!!a          ), pc+1            )

  Add   a b z -> ( mem <~ (z, mem!!a + mem!!b), pc+1            )
  Pred  a z   -> ( mem <~ (z, mem!!a - 1      ), pc+1            )

  Eq0   a     -> ( mem <~ (0, mem!!a === 0    ), pc+1            )

  Jump  i     -> ( mem                         , i               )
  CJump i     -> ( mem                         , if   mem!!0 == 1
                                                 then i
                                                 else pc+1       )

  End         -> ( mem                         , pc              )
```

```
fibProg :: Nmbr -> Program

fibProg n = [ Write 0 0
            , Write 1 n
            , Write 2 1
            , Write 3 0
            , Eq0   1
            , CJump 11
            , Add   2 3 4
            , Move  2 3
            , Move  4 2
            , Pred  1 1
            , Jump  4
            , End
            ]
```

```
fibTest 6
```

Instructions: specification

```
instrSem :: Instruction -> State -> State

instrSem instr (mem,pc) = case instr of   -- ================              ==============
                                          --   mem
                                            ----------------              --------------
  Write z n     -> ( mem <~ (z, n                                                       )
  Move  a z     -> ( mem <~ (z, mem!                                                    )

  Add   a b z   -> ( mem <~ (z, mem!                                                    )
  Pred  a z     -> ( mem <~ (z, mem!                                                    )

  Eq0   a       -> ( mem <~ (0, mem!                                                    )

  Jump  i       -> ( mem                                                                )
  CJump i       -> ( mem                                               mem!!0 == 1
                                                                       i
                                                                       pc+1             )

  End           -> ( mem                                                                )
```

type *Mem* = **Vec** 1024 *Nmbr*

type *State* = (*Mem*, *PC*)

type *Program* = [*Instruction*]

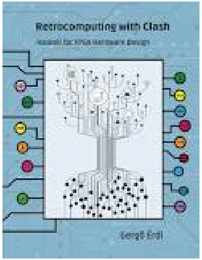
```
fibProg :: Nmbr -> Program

fibProg n = [ Write 0 0
            , Write 1 n
            , Write 2 1
            , Write 3 0
            , Eq0   1
            , CJump 11
            , Add   2 3 4
            , Move  2 3
            , Move  4 2
            , Pred  1 1
            , Jump  4
            , End
            ]
```

```
fibTest 6
```

```
(<0,0,0,0,0>,0)
(<0,6,0,0,0>,1)
(<0,6,1,0,0>,2)
(<0,6,1,0,0>,3)
(<0,6,1,0,0>,4)
(<0,6,1,0,0>,5)
(<0,6,1,0,1>,6)
(<0,6,1,1,1>,7)
(<0,6,1,1,1>,8)
(<0,5,1,1,1>,9)
(<0,5,1,1,1>,3)
(<0,5,1,1,1>,4)
(<0,5,1,1,1>,5)
(<0,5,1,1,2>,6)
(<0,5,1,1,2>,7)
(<0,5,2,1,2>,8)
(<0,4,2,1,2>,9)
(<0,4,2,1,2>,3)
(<0,4,2,1,2>,4)
(<0,4,2,1,2>,5)
(<0,4,2,1,3>,6)
(<0,4,2,2,3>,7)
(<0,4,3,2,3>,8)
(<0,3,3,2,3>,9)
(<0,3,3,2,3>,3)
(<0,3,3,2,3>,4)
(<0,3,3,2,3>,5)
(<0,3,3,2,5>,6)
(<0,3,3,3,5>,7)
(<0,3,5,3,5>,8)
(<0,2,5,3,5>,9)
(<0,2,5,3,5>,3)
(<0,2,5,3,5>,4)
(<0,2,5,3,5>,5)
(<0,2,5,3,8>,6)
(<0,2,5,5,8>,7)
(<0,2,8,5,8>,8)
(<0,1,8,5,8>,9)
(<0,1,8,5,8>,3)
(<0,1,8,5,8>,4)
(<0,1,8,5,8>,5)
(<0,1,8,5,13>,6)
(<0,1,8,8,13>,7)
(<0,1,13,8,13>,8)
(<0,0,13,8,13>,9)
(<0,0,13,8,13>,3)
(<1,0,13,8,13>,4)
(<1,0,13,8,13>,10)
```

Book



Dr. Gergö Érdi: *Retrocomputing with Clash – Haskell for FPGA Hardware Design*, https://gergo.erdi.hu/retroclash/, December 2021

Thank you

jan.kuper@qbaylogic.com
qbaylogic.com