Digital hardware design with *Clash*

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

Spring 2022 RISC-V Week, Paris
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

- FPGA Design House; FPGA services — using Clash
- Clash: Haskell \(\Rightarrow\) 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

\[ \text{mulAcc} \]

\[ \text{(x,y)} \rightarrow \text{*} \rightarrow \text{+} \rightarrow \text{0} \rightarrow z \]

\[ \text{mulAcc} : \text{s} \rightarrow \text{i} \rightarrow (\text{s}, \text{o}) \]

\[ \text{mulAcc}(x,y) = (s', z) \]

where

\[ s' = s + x \times y \]

\[ z = s + x \times y \]

Mealy Machine

\[ \text{mulAcc} : \text{Signal dom i} \rightarrow \text{Signal dom o} \]

\[ \text{mulAcc}(x,y) = z \]

where

\[ (x,y) = \text{unbundle xy} \]

\[ z = \text{register 0} (z + x \times y) \]

Signal function

\[ \text{mealy} \Rightarrow \text{moore} \]

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
Clash: Functional Perspective

\[ \text{mulAcc} \colon s \rightarrow i \rightarrow (s, o) \]

Mealy Machine

\[ \text{mulAcc} :: \text{Signal dom } i \rightarrow \text{Signal dom } o \]

Signal function
Clash: Functional Perspective

**Mealy Machine**

\[
\text{mulAcc} :: s \rightarrow i \rightarrow (s, o)
\]

\[
\text{mulAcc } s (x, y) = (s', z)
\]

where

\[
s' = s + x \times y
\]

\[
z = s
\]

**Signal function**

\[
\text{mulAcc} :: \text{Signal dom } i \rightarrow \text{Signal dom } o
\]

\[
\text{mulAcc } xy = z
\]

where

\[
(x, y) = \text{unbundle } xy
\]

\[
z = \text{register } 0 (z + x \times y)
\]
Clash: Functional Perspective

**Mealy Machine**

\[
\text{mulAcc} :: s \to i \to (s, o)
\]

\[
\text{mulAcc } s (x, y) = (s', z)
\]

\[
\text{where}
\]

\[
s' = s + x \ast y
\]

\[
z = s
\]

**Signal function**

\[
\text{mulAcc} :: \text{Signal} \text{ dom } i \to \text{Signal} \text{ dom } o
\]

\[
\text{mulAcc } xy = z
\]

\[
\text{where}
\]

\[
(x, y) = \text{unbundle } xy
\]

\[
z = \text{register 0} \ (z + x \ast y)
\]
Clash: Functional Perspective

**mulAcc**

\[
mulAcc \colon s \to i \to (s, o)
\]

\[
mulAcc \ s \ (x, y) = (s', z)
\]

where

\[
s' = s + x \times y
\]

\[
z = s
\]

**Mealy Machine**

**Signal function**

\[
mealy \ \implies \ moore
\]

\[
mulAcc \colon \text{Signal dom } i \to \text{Signal dom } o
\]

\[
mulAcc \ xy = z
\]

where

\[
(x, y) = \text{unbundle } xy
\]

\[
z = \text{register } 0 \ (z + x \times y)
\]

- Powerful abstraction mechanisms
- Strong typing system
- Straightforward simulation/test
- Control over hardware details
**Clash: Functional Perspective**

### mulAcc

- **Functional Perspective**
  - `mulAcc :: s → i → (s, o)`
  - `mulAcc s (x, y) = (s', z)`
    - `where`
      - `s' = s + x * y`
      - `z = s`

### Mealy Machine

- **mealy**
  - `mulAcc :: Signal dom i → Signal dom o`
  - `mulAcc xy = z`
    - `where`
      - `(x, y) = unpack xy`
      - `z = register 0 (z + x * y)`

### Signal function

- **moore**
  - `mulAcc :: Signal dom i → Signal dom o`
  - `mulAcc xy = z`
    - `where`
      - `(x, y) = unpack xy`
      - `z = register 0 (z + x * y)`

### Key Features

- **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

Mathematical spec

Haskell

CPU

C/C++/Python/

HLS

FPGA

VHDL/Verilog

Platforms

Time/Space
Design process

- (Mathematical) spec
- CPU: C/C++/Python/⋯
- FPGA: VHDL/Verilog

Platforms:
- HLS
- Clash

Data dependencies:
- Time/Space
Design process

(Mathematical) spec

Platform:
- CPU: C/C++/Python/…
- FPGA: VHDL/Verilog

Model:
- Data dependencies
- Time/Space

HLS
Design process

- **(Mathematical) spec** $\sim$ Haskell

Platforms:
- CPU: C/C++/Python/\ldots
- FPGA: VHDL/Verilog

Model

Data dependencies

Time/Space

HLS
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

<table>
<thead>
<tr>
<th>Taps IIR</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>49</td>
</tr>
<tr>
<td>10</td>
<td>61</td>
</tr>
<tr>
<td>20</td>
<td>78</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Multiplier</th>
<th>Number of operators</th>
<th>Pipeline stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

Freq: 550MHz
IIR: Formal model

\[ y(n) = \sum_{i=0}^{N} b_i x(n-i) - \sum_{j=1}^{M} a_j y(n-j) \]

\[ y(n) = c_0 + b_0 \hat{x}_n + \cdots + b_N \hat{x}_{n-N} - a_0 \hat{y}_{n-1} - \cdots - a_M \hat{y}_{n-M} \]
IIR: Formal model

\[ y[n] = a_0 \sum_{i=0}^{N} b_i x[n-i] - a_1 \sum_{j=1}^{M} a_j y[n-j] \]

...
IIR: Formal model

\[ y[n] = a_0 \left( \sum_{i=0}^{N} b_i x[n-i] - \sum_{j=1}^{M} a_j y[n-j] \right) + c \]

\[ + \sum_{i=0}^{N} b_i \hat{x}^i x[n-i] + a_0 \sum_{j=1}^{M} a_j \hat{y}^j y[n-j] \]

Word-for-word translation
Haskell = Math
Executable
Test: Slow!
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

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

\[ \vdots \]

\[ = c \left( 0 \bigoplus b_{0\cdots N} \hat{\times} x_{n\cdots n-N} - 0 \bigoplus a_{0\cdots M-1} \hat{\times} y_{n-1\cdots n-M} \right) \]
IIR: Formal model

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

\[ \vdots \]

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

```
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])
)```

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

Test: Slow!
IIR: Formal model

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

\[ = c \left( \sum_{i=0}^{N} b_i \hat{x}_{n-N} \ast x_{n-N-n} - \sum_{j=1}^{M} a_j \hat{y}_{n-M} \ast y_{n-M-n} \right) \]

Test: \( \text{testA} = yA[\theta..40] \)

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

\[ yA \ n \ \mid \ n < \theta \ = \ \emptyset \]
\[ \text{otherwise} = c \ * \ (\text{foldl} (+) \emptyset \ \text{zipWith} (*) (b[\theta..nn]) \ (yA[n-1..n-\theta])) \]
\[ - \text{foldl} (+) \emptyset \ \text{zipWith} (*) (a[\theta..mm-1]) \ (yA[n-1..n-\theta]) \]

Slow!
IIR: Parameter accumulation

\[
y_n = c \left( \bigoplus b_0 \cdots b_{N-1} \hat{\ast} x_{n-N} - \bigoplus a_0 \cdots a_{M-1} \hat{\ast} y_{n-M} \right)
\]
IIR: Parameter accumulation

\[ y_n = c \left( 0 \oplus b_{0\ldots N} \ast x_{n\ldots n-N} - 0 \oplus a_{0\ldots M-1} \ast y_{n-1\ldots n-M} \right) \]

\[ y_n(x_s, y_s) = y_n : y_{n+1}(x'_s, y'_s) \]

Definitions:
- \( us \cdot vs = 0 \oplus us \ast vs \)
- \( y_n = c \left( bs \cdot x_s - as \cdot y_s \right) \)
- \( x'_s = x_{n+1} \Rightarrow x_s \)
- \( y'_s = y_{n} \Rightarrow y_s \)

Proof of equivalence: induction on \( n \)
IIR: Parameter accumulation

\[ y_n = c \left( \bigoplus b_{0\cdots N} \ast x_{n-\cdots N} - \bigoplus a_{0\cdots M-1} \ast y_{n-\cdots M} \right) \]

Definitions:
- \( u \cdot v = 0 \bigoplus u \ast v \)
- \( y_n = c \left( b_s \cdot x_s - a_s \cdot y_s \right) \)
- \( x_s' = x_{n+1} \Leftrightarrow x_s \)
- \( y_s' = y_n \Leftrightarrow y_s \)

Proof of equivalence: induction on \( n \)

Test:
\[ \text{testB} = \text{take 40 } \theta (yB \theta (xs0,ys0)) \]

Model = Golden reference
IIR: Recursor

\[ y_n(x, y) = y_n : y_{n+1}(x', y') \]

Definitions:

\[ u \cdot v = 0 \quad u \ast v \]

\[ y_n = c( bs \cdot x - as \cdot y) \]

\[ x' = x_{n+1} \gg x \]

\[ y' = y_n \gg y \]
IIR: Recursor

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

Definitions:
- \( us \cdot vs = 0 \uplus us \ast vs \)
- \( y_n = c( bs \cdot xs - as \cdot ys ) \)
- \( xs' = x_{n+1} \Rightarrow xs \)
- \( ys' = y_n \Rightarrow ys \)

one-step function

\[ y^1(xs,ys) x_{n+1} = \langle (xs',ys'), y_n \rangle \]

Definitions:
- \( us \cdot vs = 0 \uplus us \ast vs \)
- \( y_n = c( bs \cdot xs - as \cdot ys ) \)
- \( xs' = x_{n+1} \Rightarrow xs \)
- \( ys' = y_n \Rightarrow ys \)

Recursor

\( R(y^1) \)

Proof of equivalence: induction on \( n \)
\[ y_n(x_s, y_s) = y_n : y_{n+1}(x_s', y_s') \]

**Definitions:**
\[ u_s \cdot v_s = 0 \oplus u_s \star v_s \]
\[ y_n = c(b_s \cdot x_s - a_s \cdot y_s) \]
\[ x_s' = x_{n+1} \Rightarrow x_s \]
\[ y_s' = y_n \Rightarrow y_s \]

**One-step function**

\[ y^1(x_s, y_s) x_{n+1} = \langle x_s', y_s', y_n \rangle \]

**Definitions:**
\[ u_s \cdot v_s = 0 \oplus u_s \star v_s \]
\[ y_n = c(b_s \cdot x_s - a_s \cdot y_s) \]
\[ x_s' = x_{n+1} \Rightarrow x_s \]
\[ y_s' = y_n \Rightarrow y_s \]

**Recursor**

\[ \mathcal{R}(y^1) \]

**Proof of equivalence:** induction on \( n \)

**Test:**
\[ \text{testC = sim yC (xs0, ys0) (x & [1..40])} \]
IIR: Recursor

- Mealy Machine $\Rightarrow$ 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
```

`:vhdl`
`:verilog`

sim yC
IIR: Architecture

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

```
us \cdot vs = foldl (+) 0 (zipWith (*) us vs)
```
IIR: Architecture

\[
\text{yC } (xs, ys) \times = ( (xs', ys') , y )
\]

\[
\text{where}
\]

\[
y = c \times (bs \times xs - as \times ys)
\]

\[
xs' = x + \ggg xs
\]

\[
ys' = y + \ggg ys
\]

\[
us \times vs = \text{foldl } (+) 0 (\text{zipWith } \times) us vs
\]

Dot product:
IIR: Architecture

\[ yC(x_s, y_s) \cdot x = ( (x_s', y_s'), y) \]
\[
\text{where}
\]
\[
y = c \cdot (b \cdot x_s - a \cdot y_s)
\]
\[
x_s' = x + \rightarrow x_s
\]
\[
y_s' = y + \rightarrow y_s
\]

\[ us \cdot vs = \text{foldl}(\oplus) \theta (\text{zipWith} \ (\ast)) \ us \ vs \]

Dot product:

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

⇒ Optimisations needed
IIR: Linearisation

\[ y = c (bs \cdot xs - as \cdot ys) \]
IIR: Linearisation

\[
\begin{align*}
y &= c (bs \cdot xs - as \cdot ys) \\
    &= c ( (bs \cdot xs) \cdot (ys + as)) \\
    &= c ( (bs \cdot xs) \cdot (ys + as)) \\
    &= ps \cdot xys \\
    &= \text{foldl} (+) 0 (\text{zipWith} (\cdot) ps xys) \\
    &= \text{foldl} ((+) \cdot (\cdot)) 0 pxys \\
    &= \text{foldl} f 0 pxys \\
    &= \text{foldr} f 0 pxys
\end{align*}
\]
IIR: Sequentialising over time

Standard transformation
- Standard code patterns
- State machine
IIR: Sequentialising over time

- Standard transformation
- Standard code patterns
IIR: Sequentialising over time

- Standard transformation
- Standard code patterns
- State machine
**IIR: Sequentialising over time**

\[
x_0 \rightarrow x_1 \rightarrow x_2 \rightarrow y_0 \rightarrow y_1 \rightarrow y_2 \rightarrow 0
\]

\[
f(0) = f(1) = f(2) = f(3) = f(4) = f(5)
\]

\[
x(0) = x(1) = x(2) = y(0) = y(1) = y(2) = y(3)
\]

\[
p(0) = p(1) = p(2) = p(3) = p(4) = p(5)
\]

\[
\text{acc} = (0, \ldots, 0)
\]

\[
\text{outp} = (0, \ldots, 0)
\]

<table>
<thead>
<tr>
<th>State</th>
<th>s</th>
<th>ps</th>
<th>xs</th>
<th>ys</th>
<th>acc</th>
<th>outp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idle</td>
<td>(1)</td>
<td>–</td>
<td>(2)</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Calc</td>
<td>(3)</td>
<td>(p_i \rightarrow ps)</td>
<td>(4)</td>
<td>(x_i \rightarrow ys)</td>
<td>(5)</td>
<td>–</td>
</tr>
<tr>
<td>Ready</td>
<td>Idle</td>
<td>–</td>
<td>–</td>
<td>acc</td>
<td>(6)</td>
<td>acc</td>
</tr>
</tbody>
</table>

\[
\text{Proof: invariant + induction}
\]

\[
yCseq (s, ps, xs, ys, acc) \text{ inp } = ((s', ps', xs', ys', acc'), \text{ outp })
\]

\[
\text{where}
\]

\[
\text{case } s \text{ of}
\]

\[
\begin{align*}
\text{Idle} & \rightarrow (s_-, ps, xs_-, ys, acc, \text{ Nothing }) \\
\text{Calc} & \rightarrow (s_-, ps \rightarrow ps, \text{ Prev } y \rightarrow ys, \text{ last } xs \rightarrow ys, \text{ acc } \rightarrow p^* y, \text{ Nothing }) \quad \text{where } p = \text{ last } ps
\end{align*}
\]

\[
\text{Ready} \rightarrow (\text{ Idle, ps } , xs , \text{ Prev acc } \rightarrow ys, \text{ Nothing }) \quad \text{Just acc}
\]

\[
\text{Test:}
\]

RISC-V Week 2022

Digital hardware design in Clash
IIR: Sequentialising over time

\[ \text{Proof: invariant + induction} \]

\[
\begin{array}{|c|c|c|c|c|c|c|}
\hline
& s & ps & xs & ys & acc & outp \\
\hline
\text{Idle} & (1) & p_0 & \rightarrow & ps & (2) & \rightarrow & \rightarrow & (5) & \rightarrow & \rightarrow \\
\text{Calc} & (3) & x_t & \rightarrow & ys & (4) & \rightarrow & \rightarrow & 0 & \rightarrow & \rightarrow \\
\text{Ready} & Idle & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
inp & (1) & (2) & y_t & (3) & (4) & (5) \\
\hline
x & \text{Calc} & x^0 & \rightarrow & xs & y^0 & \text{Calc} & y^0 & \rightarrow & xs & acc + p_t*y \\
\hline
\text{Idle} & \rightarrow & \rightarrow & \rightarrow & \rightarrow & 0 & \text{Calc} & \rightarrow & \rightarrow & xs & acc + p_t*y \\
\hline
\end{array}
\]

\[
yCseq \ (s, ps, xs, ys, acc) \ \text{inp} = ((s', ps', xs', ys', acc'), outp )
\]

\[
\begin{align*}
\text{where} \quad \text{case} & \\
\text{Idle} & \rightarrow \ (s_0, ps, xs) \\
\text{Calc} & \rightarrow \ (s_1, ps, \text{Prev } y \rightarrow xs, \text{last } xs \rightarrow ys, acc + p_t*y) \\
\text{Ready} & \rightarrow \ (\text{Idle}, ps, xs, \text{Prev } acc \rightarrow ys, \theta, \text{Just } acc)
\end{align*}
\]
IIR: Sequentialising over time

\[
\begin{align*}
\text{Test: } & \quad \text{testCseq} = \ldots \\
\text{Proof: invaraint + induction}
\end{align*}
\]
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)
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 \(\text{DSignal}\), parameterisable
HDL generation

- Typing: Polymorphic $\Rightarrow$ monomorphic
- Define top Entity
- 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: \( a \to b \)
Vector types: Vec \( n a \), BitVector \( n \)
Signal types: Signal \( \text{dom} a \), DSignal \( \text{dom} d a \)

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: \( a \to b \)

Vector types: Vec \( n \ a \), BitVector \( n \)

Signal types: Signal \( \text{dom} \ a \), DSignal \( \text{dom} \ d \ a \)

Tuples, Records, Algebraic types, ...

\[
\begin{align*}
\text{head} & : \text{Vec} \ (n+1) \ a \to a \\
\text{concat} & : \text{Vec} \ n \ (\text{Vec} \ m \ a) \to \text{Vec} \ (n* m) \ a
\end{align*}
\]
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, …

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

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

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

\[
\begin{align*}
\text{iterate} & \:: (\mathbb{N} \to a \to a) \to a \to \mathbb{N} \to [a] \\
\text{foldl} & \:: (a \to b \to a) \to a \to [b] \to a \\
\text{mapAccumL} & \:: (a \to b \to a) \to a \to [b] \to [a]
\end{align*}
\]

\[
\begin{align*}
\text{foldl} & \quad : (a \to b \to a) \to a \to Vec n b \to a \\
\text{scanl} & \quad : (a \to b \to a) \to a \to Vec n b \to Vec (n+1) a
\end{align*}
\]
Higher Order Functions

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

\[
\text{iterate } n \ f \ a
\]

\[
\text{foldl } f \ a \ xs
\]

\[
\text{map } f \ xs
\]

\[
\text{zipWith } f \ xs \ ys
\]

\[
\text{scanl } f \ a \ xs
\]

\[
\text{mapAccumL } f \ a \ xs
\]

\[
\text{cartProdWith } f \ xs \ ys
\]

\[
\text{foldl} : (a \to b \to a) \to a \to \text{Vec } n \ b \to a
\]

\[
\text{scanl} : (a \to b \to a) \to a \to \text{Vec } n \ b \to \text{Vec } (n+1) \ a
\]

\[
\text{cartProdWith} : (a \to b \to c) \to \text{Vec } n \ a \to \text{Vec } m \ b \to \text{Vec } n \ (\text{Vec } m \ c)
\]
Higher Order Functions

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

```
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
```

- `foldl :: (a → b → a) → a → Vec n b → a`
- `scanl :: (a → b → a) → a → Vec n b → Vec (n+1) a`
- `cartProdWith :: (a → b → c) → Vec n a → Vec m b → Vec n (Vec m c)`

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

- Provable loop transformations

\[ \text{result} = \text{foldl} \ f \ a \ vs \]
Higher Order Functions

- Provable loop transformations

\[
\text{result} = \text{foldl } f \ a \ vs
\]

\[
\text{result} = \text{foldl} (\text{foldl } f) \ a \ vss
\]
Higher Order Functions

- Provable loop transformations

\[ \text{result} = \text{foldl} \ f \ a \ vs \]

\[ \text{result} = \text{foldl} \ f \ a \ (\text{map} \ (\text{foldl} \ f \ a) \ vss) \]

Matrix multiplication:
\[ m_0 \times m_1 = \text{cartProdWith} (\cdot) m_0 (\text{transpose} m_1) \]

associative, neutral element
Higher Order Functions

- Provable loop transformations

\[
result = \text{foldl} \ f \ a \ vs
\]

associative, commutative, neutral element

\[
result = \text{foldl} \ f \ a \ pts
\]

where

\[
zs = \text{replicate} \ m \ a
\]

\[
pts = \text{foldl} \ (\text{zipWith} \ f) \ zs \ vss
\]
Algebraic Data Types = Embedded Languages

Algebraic types: *Constructors* + *Arguments*

```plaintext
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:

instrumentSem instr :: State → State

instrumentSem instr :: Instruction → State → State
```
Algebraic Data Types = Embedded Languages

Algebraic types: \textit{Constructors} + \textit{Arguments}

\begin{align*}
\text{type } PC & \quad = \quad \text{Unsigned 8} \\
\text{type } Nmbr & \quad = \quad \text{Signed 32} \\
\text{type } Addr & \quad = \quad \text{Unsigned 10} \\
\text{data } Instruction & \quad = \quad \text{Write Addr } Nmbr \\
& \quad \quad \quad \quad \text{Move Addr } Addr \\
& \quad \quad \quad \quad \text{Add Addr } Addr \quad Addr \\
& \quad \quad \quad \quad \text{Pred Addr } Addr \\
& \quad \quad \quad \quad \text{Eq0 Addr} \\
& \quad \quad \quad \quad \text{Jump } PC \\
& \quad \quad \quad \quad \text{CJump } PC \\
& \quad \quad \quad \quad \text{Stop}
\end{align*}
### Algebraic Data Types = Embedded Languages

Algebraic types: **Constructors + Arguments**

<table>
<thead>
<tr>
<th>type</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PC</strong></td>
<td>Unsigned 8</td>
</tr>
<tr>
<td><strong>Nmbr</strong></td>
<td>Signed 32</td>
</tr>
<tr>
<td><strong>Addr</strong></td>
<td>Unsigned 10</td>
</tr>
</tbody>
</table>

#### 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
Algebraic Data Types = Embedded Languages

Algebraic types: Constructors + Arguments

<table>
<thead>
<tr>
<th>type</th>
<th>PC</th>
<th>=</th>
<th>Unsigned 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>Nmbr</td>
<td>=</td>
<td>Signed 32</td>
</tr>
<tr>
<td>type</td>
<td>Addr</td>
<td>=</td>
<td>Unsigned 10</td>
</tr>
<tr>
<td>data</td>
<td>Instruction</td>
<td>=</td>
<td>Write Addr Nmbr</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Move Addr Addr</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Add Addr Addr Addr</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Pred Addr Addr</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Eq0 Addr</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Jump PC</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CJump PC</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Stop</td>
</tr>
</tbody>
</table>

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

Add 4 5 12
CJump 8
Algebraic Data Types = Embedded Languages

Algebraic types: Constructors + Arguments

<table>
<thead>
<tr>
<th>Type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>Unsigned 8</td>
</tr>
<tr>
<td>Nmbr</td>
<td>Signed 32</td>
</tr>
<tr>
<td>Addr</td>
<td>Unsigned 10</td>
</tr>
</tbody>
</table>

Data Instruction:
- Write Addr Nmbr
- Move Addr Addr
- Add Addr Addr Addr
- Pred Addr Addr
- Eq0 Addr
- Jump PC
- CJump PC
- Stop

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

Semantics, specification:

\[
\text{instrSem} \text{ instr} :: \text{State} \rightarrow \text{State}
\]

Add 4 5 12
CJump 8
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

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

Semantics, specification:

\[
\text{instrSem } \text{instr} :: \text{State} \rightarrow \text{State}
\]

\[
\text{instrSem} :: \text{Instruction} \rightarrow \text{State} \rightarrow \text{State}
\]
Instructions: specification

define type Mem = Vec 1024 Nmbr
define type State = (Mem, PC)
Instructions: specification

\[
\text{type } \text{Mem} = \text{Vec } 1024 \text{ Nmbr}
\]

\[
\text{type } \text{State} = (\text{Mem}, \text{PC})
\]
Instructions: specification

<table>
<thead>
<tr>
<th>Instruction</th>
<th>State 1</th>
<th>State 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write z n</td>
<td>mem = (z, n)</td>
<td>pc+1</td>
</tr>
<tr>
<td>Move a z</td>
<td>mem = (z, mem!!a)</td>
<td>pc+1</td>
</tr>
<tr>
<td>Add a b z</td>
<td>mem = (z, mem!!a + mem!!b)</td>
<td>pc+1</td>
</tr>
<tr>
<td>Pred a z</td>
<td>mem = (z, mem!!a - 1)</td>
<td>pc+1</td>
</tr>
<tr>
<td>Eq0 a</td>
<td>mem = (0, mem!!a == 0)</td>
<td>pc+1</td>
</tr>
<tr>
<td>Jump i</td>
<td>mem = i</td>
<td>pc+1</td>
</tr>
<tr>
<td>CJump i</td>
<td>mem = if mem!!0 == 1 then i else pc+1</td>
<td></td>
</tr>
<tr>
<td>End</td>
<td>mem</td>
<td>pc</td>
</tr>
</tbody>
</table>

RISC-V Week 2022
Digital hardware design in Clash

fibProg = [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]
**Instructions: specification**

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

instrSem instr (mem, pc) = case instr of
  Write zn -> (mem <=< (zn, mem))
  Move az -> (mem <=< (zn, mem))
  Add abz -> (mem <=< (zn, mem))
  Pred az -> (mem <=< (zn, mem))
  EqQ a -> (mem <=< (0, mem))
  Jump i -> (mem)
  CJump i -> (mem)
  End -> (mem)
```

**type Mem = Vec 1024 Nmbr**

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

**type Program = [ Instruction ]**

```haskell
fibProg :: Nmbr -> Program
fibProg n = [ Write 0 0 , Write 1 n , Write 2 1 , Write 3 0 , EqQ 1 , CJump 11 , Add 2 3 4 , Move 2 3 , Move 4 2 , Pred 1 1 , Jump 4 , End ]
```

**fibTest 6**
Dr. Gergö Érdi: *Retrocomputing with Clash – Haskell for FPGA Hardware Design*, https://gergo.erd.hu/retroclash/, December 2021
Thank you

jan.kuper@qbaylogic.com
qbaylogic.com