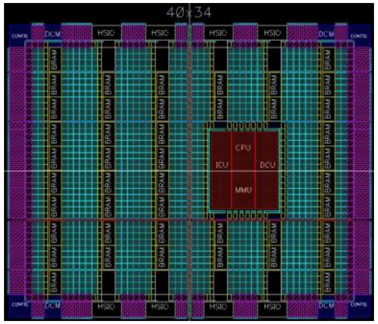
Running Dynamic Algorithms on Static Hardware



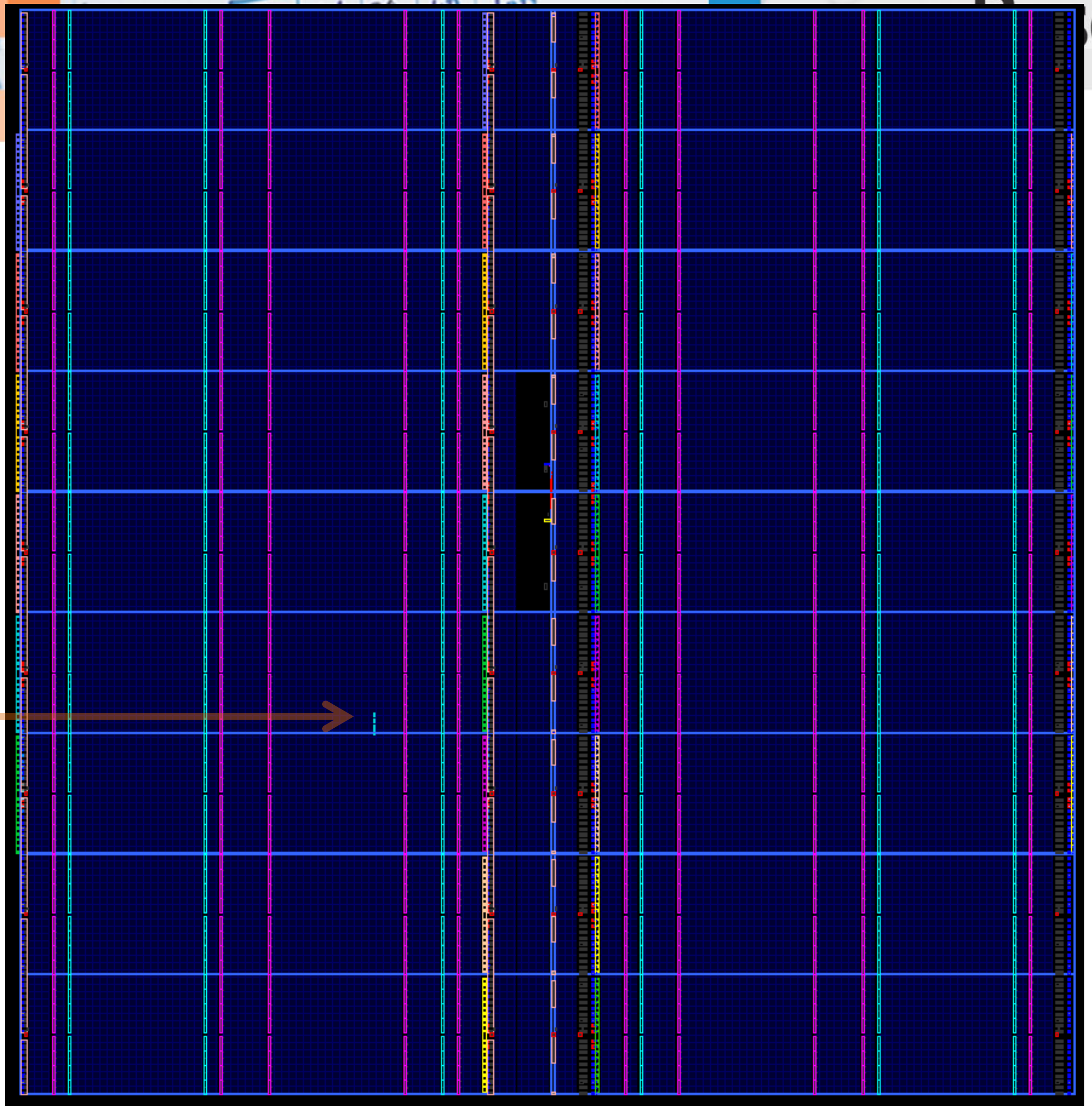

```
fib 0 = 0
fib 1 = 1
fib n = fib (n−1) + fib (n−2)
```

```
island fib n = case n of
                0 −> let zero = 0 in return zero
                1 −> let one = 1 in return one
                _ −> let one = 1 in
                      let n1 = − n 1 in
                      recurse s1 n (n1)
    s1 n  n1 = let two = 2 in
               let n2 = − n two in
               recurse s2 n1 (n2)
    s2 n1 n2 = let r = + n1 n2 in
               return r
```

Stephen Edwards (Columbia)
Simon Peyton Jones, MSR Cambridge
Satnam Singh, MSR Cambridge

14820 sim-adds
1,037,400,000,000
additions/second

32-bit
integer
Adder
(32/474,240)
>700MHz



332x1440

XC6VLX760 758,784 logic cells, 864 DSP blocks, 1,440 dual ported 18Kb RAMs

The holy grail

- Software is quick to write
- Gates are fast and power-efficient to run
- FPGAs make it seem tantalisingly close

Does not work

ality

The progra
(software)

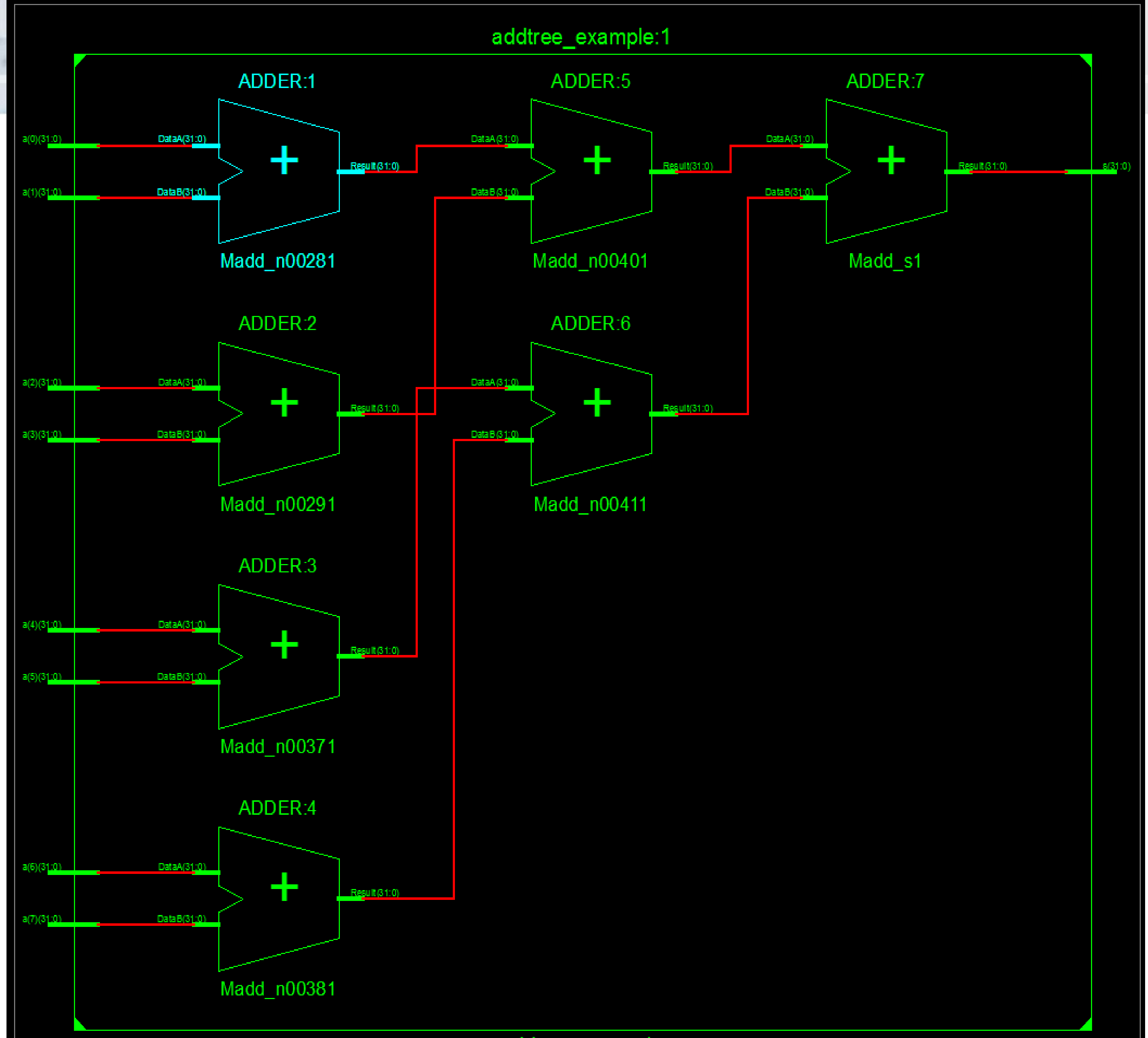
ates

- Programs are
  - Recursive
  - Dynamic
  - Use the heap

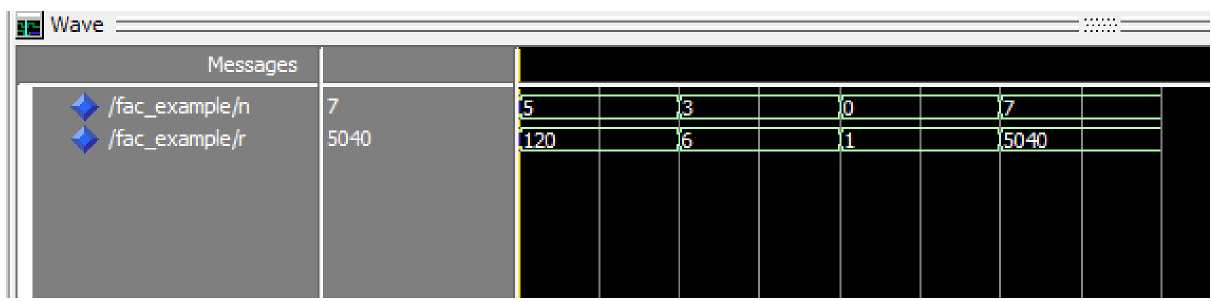
- Hardware is
  - Iterative
  - Static
  - No heap

This talk: towards bridging the gap

```
function addtree (a : int_array) return integer is
   variable len, offset : natural ;
   variable lhs, rhs : integer ;
 begin
   len := a'length ;
   offset := a'left(1) ;
   if len = 1 then
     return a(offset) ;
   else
     lhs := addtree (a(offset to offset+len/2-1)) ;
     rhs := addtree (a(offset+len/2 to offset+len-1)) ;
     return lhs + rhs ;
   end if ;
 end function addtree ;
```


addtree_example:1
ADDER:1
ADDER:5
ADDER:7
ADDER:2
ADDER:6
ADDER:3
ADDER:4
Madd_n00281
Madd_n00401
Madd_s1
Madd_n00291
Madd_n00411
Madd_n00371
Madd_n00381

```
entity fac_example is
  port (signal n : in natural ;
        signal r : out natural) ;
end entity fac_example ;

architecture behavioural of fac_example is

  function fac (n : in natural) return natural is
  begin
    if n <= 1 then
      return 1 ;
    else
      return n * fac (n-1) ;
    end if ;
  end function fac ;

begin

  r <= fac (n) ;

end architecture behavioural ;
```



```
c := a + b ; -- cycle 0
d := 2 * c ; -- cycle 1
e := d – 5;  -- cycle 2
```

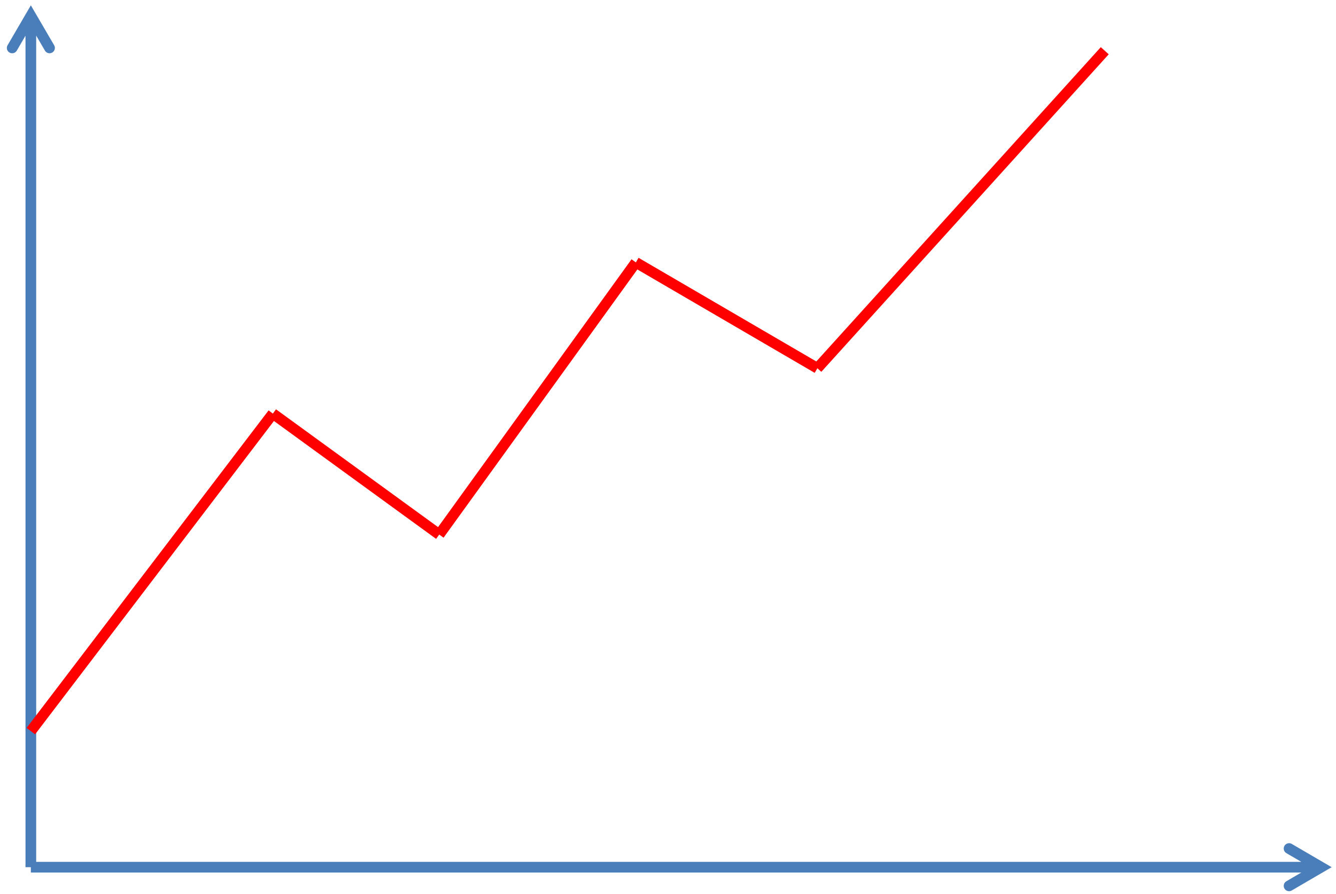
```
process
  variable state : integer := 0 ;
  variable c, d, e : integer ;
begin
  wait until clk’event and clk=‘1’;
  case state is
    when 0 => c := a + b ; state := 1 ;
    when 1 => d := 2 * c ; state := 2 ;
    when 2 => e := d – 5 ; state := 3 ;
    when others => null ;
  end case ;
end process ;
```

```
entity fibManual is
  port (signal clk, rst : in bit ;
        signal n : in natural ;
        signal n_en : in bit ;
        signal f : out natural ;
        signal f_rdy : out bit) ;
end entity fibManual ;


use work.fibManualPackage.all ;
use work.StackPackage.all;
architecture manual of fibManual is

begin

  compute_fib : process
    variable stack : stack_type := (others => 0) ;
    variable stack_index : stack_index_type := 0 ; -- Points to next free elem
    variable state : states := ready ;
    variable jump_stack : jump_stack_type ;
    variable jump_index : stack_index_type := 0 ;
    variable top, n1, n2, fibn1, fibn2, fib : natural ;
  begin
    wait until clk'event and clk='1' ;
    if rst = '1' then
      stack_index := 0 ;
      jump_index := 0 ;
      state := ready ;
    else
```

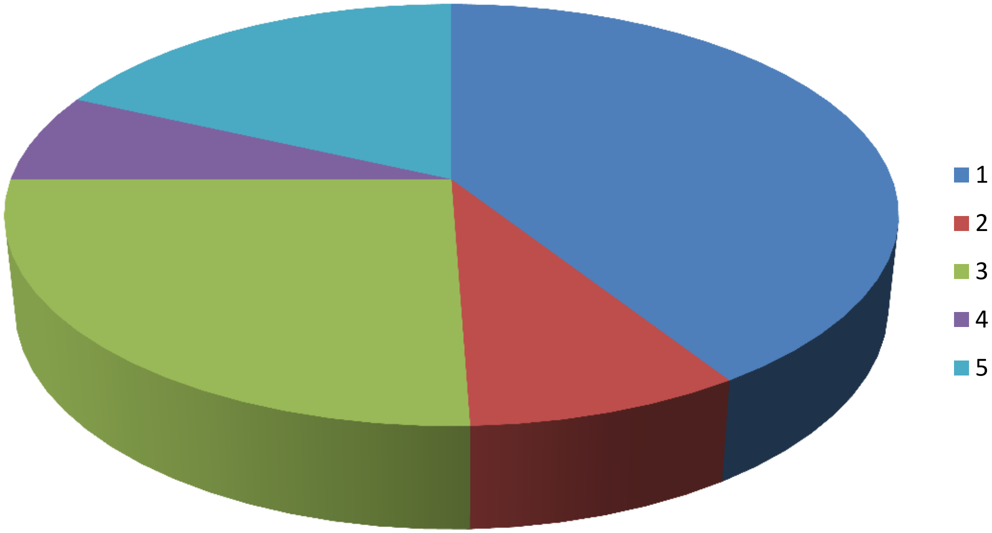
```
case state is
  when ready => if n_en = '1' then -- Ready and got new input
                -- Read input signal into top of stack
                top := n ;
                push (top, stack, stack_index) ;
                -- Return to finish
                push_jump (finish_point, jump_stack, jump_index) ;
                state := recurse ; -- Next state top of recursion
              end if ;
  when recurse
    => pop (top, stack, stack_index) ;
       case top is
         when 0 => push (top, stack, stack_index) ;
                   -- return
                   pop_jump (state, jump_stack, jump_index) ;
         when 1 => push (top, stack, stack_index) ;
                   -- return
                   pop_jump (state, jump_stack, jump_index) ;
         when others => -- push n onto the stack for use by s1
                   push (top, stack, stack_index) ;
                   -- push n-1 onto stack
                   n1 := top - 1 ;
                   push (n1, stack, stack_index) ;
                   -- set s1 as the return point
                   push_jump (s1, jump_stack, jump_index) ;
                   -- recurse
                   state := recurse ;
       end case ;
```

```
        when s1 => -- n and fib n-1 has been computed and is on the stack
                -- now compute fib n-2
                pop (fibn1, stack, stack_index) ; -- pop fib n-1
                pop (top, stack, stack_index) ; -- pop n
                push (fibn1, stack, stack_index) ; -- push fib n-1 for s2
                n2 := top - 2 ;
                push (n2, stack, stack_index) ; -- push n-2
                -- set s2 as the jump point
                push_jump (s2, jump_stack, jump_index) ;
                -- recurse
                state := recurse ;
        when s2 => pop (fibn2, stack, stack_index) ;
                pop (fibn1, stack, stack_index) ;
                fib := fibn1 + fibn2 ;
                push (fib, stack, stack_index) ;
                -- return
                pop_jump (state, jump_stack, jump_index) ;
        when finish_point => pop (fib, stack, stack_index) ;
                        f <= fib ;
                        f_rdy <= '1' ;
                        state := release_ready ;
                        stack_index := 0 ;
                        jump_index := 0 ;
        when release_ready => f_rdy <= '0' ;
                        state := ready ;
      end case ;
    end if ;
  end process compute_fib ;

end architecture manual ;
```

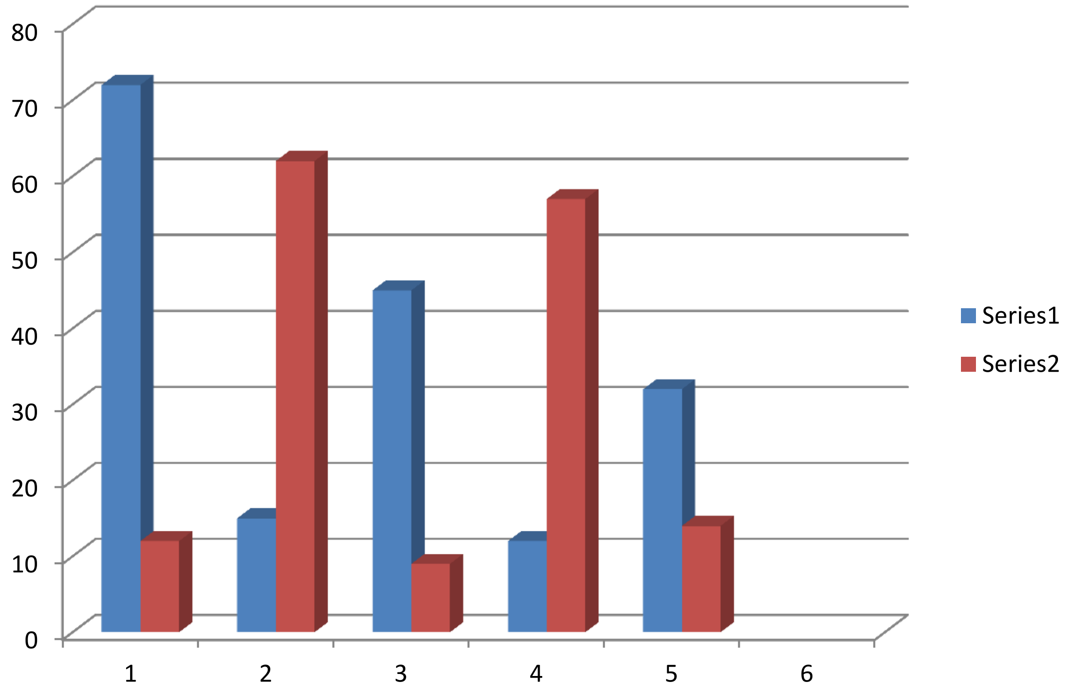
PLDI 1998



PLDI 1999



PLDI 2000



POPL 1998

$$\frac{p \xrightarrow[I \cup \{S\}]{O,k} p' \qquad S \in O}{\texttt{signal}\ S\ \texttt{in}\ p\ \texttt{end} \xrightarrow[I]{O\setminus\{S\},k} \delta_1^k(\texttt{signal}\ S\ \texttt{in}\ p'\ \texttt{end})}$$

$$\frac{p \xrightarrow[I \setminus \{S\}]{O,k} p' \qquad S \notin O}{\texttt{signal}\ S\ \texttt{in}\ p\ \texttt{end} \xrightarrow[I]{O,k} \delta_1^k(\texttt{signal}\ S\ \texttt{in}\ p'\ \texttt{end})}$$

Microsoft Research

POPL 1999

$$\frac{p \xrightarrow[I\cup\{S\}]{O,k} p' \qquad S \in O}{\texttt{signal } S \texttt{ in } p \texttt{ end} \xrightarrow[I]{O\setminus\{S\},k} \delta_1^k(\texttt{signal } S \texttt{ in } p' \texttt{ end})}$$

$$\frac{p \xrightarrow[I\setminus\{S\}]{O,k} p' \qquad S \notin O}{\texttt{signal } S \texttt{ in } p \texttt{ end} \xrightarrow[I]{O,k} \delta_1^k(\texttt{signal } S \texttt{ in } p' \texttt{ end})}$$

$$\frac{p \circ\!\!\xrightarrow[I\setminus\{S\}]{O^-,k^-} p^- \qquad S \in O^- \qquad p \circ\!\!\xrightarrow[I\cup\{S\}]{O^+,k^+} p^+ \qquad S \in O^+}{\texttt{signal } S \texttt{ in } p \texttt{ end} \circ\!\!\xrightarrow[I]{O^+\setminus\{S\},k^+} \delta_1^{k^+}(\texttt{signal } S \texttt{ in } p^+ \texttt{ end})}$$

$$\frac{p \circ\!\!\xrightarrow[I\setminus\{S\}]{O^-,k^-} p^- \qquad S \notin O^- \qquad p \circ\!\!\xrightarrow[I\cup\{S\}]{O^+,k^+} p^+ \qquad S \notin O^+}{\texttt{signal } S \texttt{ in } p \texttt{ end} \circ\!\!\xrightarrow[I]{O^-,k^-} \delta_1^{k^-}(\texttt{signal } S \texttt{ in } p^- \texttt{ end})}$$

$$\frac{\texttt{emit S} \circ\!\!\xrightarrow[\{\texttt{A}\}]{\{\texttt{S}\},0} \texttt{nothing} \qquad \texttt{S} \in \{\texttt{S}\} \qquad \texttt{emit S} \circ\!\!\xrightarrow[\{\texttt{A,S}\}]{\{\texttt{S}\},0} \texttt{nothing} \qquad \texttt{S} \in \{\texttt{S}\}}{\texttt{signal S in emit S end} \circ\!\!\xrightarrow[\{\texttt{A}\}]{\emptyset,0} \texttt{nothing}}$$

$$\frac{\texttt{pause} \circ\!\!\xrightarrow[\{\texttt{A}\}]{\emptyset,1} \texttt{nothing} \qquad \texttt{S} \notin \emptyset \qquad \texttt{pause} \circ\!\!\xrightarrow[\{\texttt{A,S}\}]{\emptyset,1} \texttt{nothing} \qquad \texttt{S} \notin \emptyset}{\texttt{signal S in pause end} \circ\!\!\xrightarrow[\{\texttt{A}\}]{\emptyset,1} \texttt{signal S in nothing end}}$$

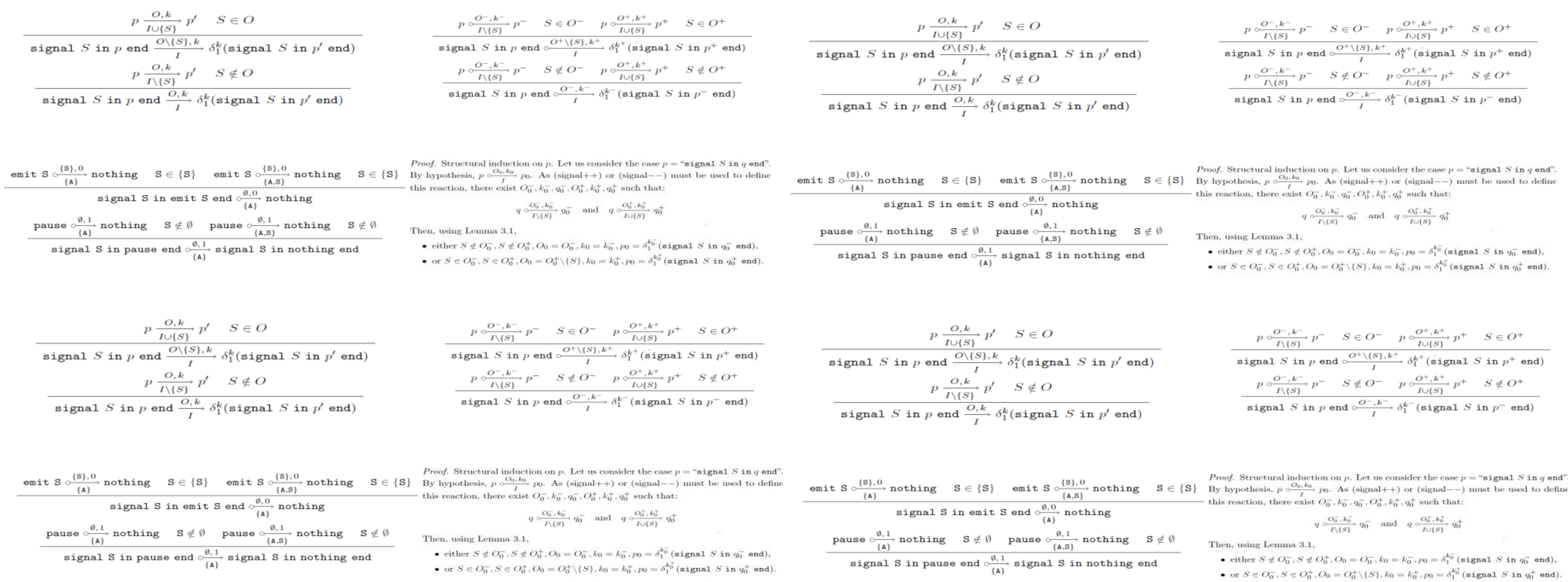
Proof. Structural induction on p. Let us consider the case $p =$ "$\texttt{signal } S \texttt{ in } q \texttt{ end}$". By hypothesis, $p \circ\!\!\xrightarrow[I]{O_0,k_0} p_0$. As (signal++) or (signal−−) must be used to define this reaction, there exist $O_0^-, k_0^-, q_0^-, O_0^+, k_0^+, q_0^+$ such that:

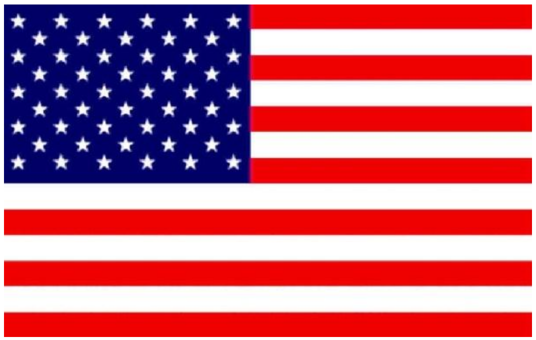
$$q \circ\!\!\xrightarrow[I\setminus\{S\}]{O_0^-,k_0^-} q_0^- \quad \text{and} \quad q \circ\!\!\xrightarrow[I\cup\{S\}]{O_0^+,k_0^+} q_0^+$$

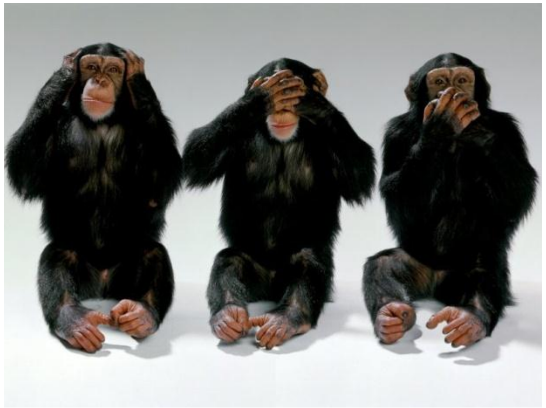
Then, using Lemma 3.1,

- either $S \notin O_0^-, S \notin O_0^+, O_0 = O_0^-, k_0 = k_0^-, p_0 = \delta_1^{k_0^-}(\texttt{signal } S \texttt{ in } q_0^- \texttt{ end})$,
- or $S \in O_0^-, S \in O_0^+, O_0 = O_0^+\setminus\{S\}, k_0 = k_0^+, p_0 = \delta_1^{k_0^+}(\texttt{signal } S \texttt{ in } q_0^+ \texttt{ end})$.

POPL 2000







$$\frac{}{\Pi; \Sigma; \Theta \vdash n : \texttt{int}}\text{(T-INT)}$$

$$\frac{}{\Pi; \Sigma; \Theta \vdash !\ell : \Sigma(\ell), \{rd_\ell\}}\text{(T-READ)}$$

$$\frac{\Pi; \Sigma; \Theta \vdash e : A, \varepsilon_1 \qquad A <: B \qquad \varepsilon_1 \subseteq \varepsilon_2}{\Pi; \Sigma; \Theta \vdash e : B, \varepsilon_2}\text{(T-SUB)}$$

Our goal

- Write a functional program, with
  - Unrestricted recursion
  - Algebraic data types
  - Heap allocation
- Compile it quickly to FPGA
- Main payoff: rapid development, exploration
- Non-goal: squeezing the last drops of performance from the hardware

Generally: **significantly broaden** the range of applications that can be directly compiled into hardware with no fuss

Applications

- Searching tree-structured dictionaries
- Directly representing recursive algorithms in hardware
- Huffman encoding
- Recursive definitions of mathematical operations

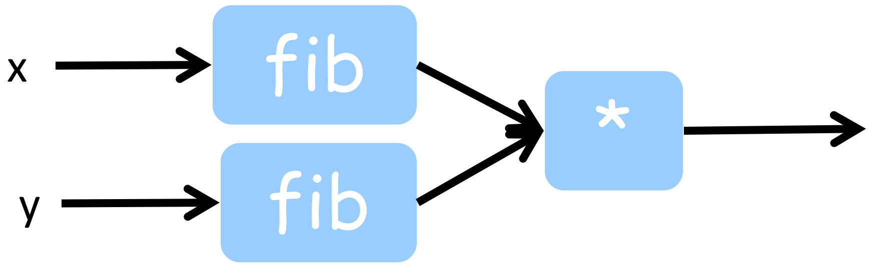
Compiling programs to hardware

- First order functional language
- Inline (absolutely) all function calls
- Result can be directly interpreted as hardware
- **Every call instantiates a copy of that function's RHS**
- No recursive functions
- [Readily extends to unrolling recursive functions with statically known arguments]

Our simple idea

- **Extend "Every call instantiates a copy of that function's RHS" to recursive functions**

```
fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib n = 1 + fib (n-1) + fib (n-2)

main x y = fib x * fib y
```

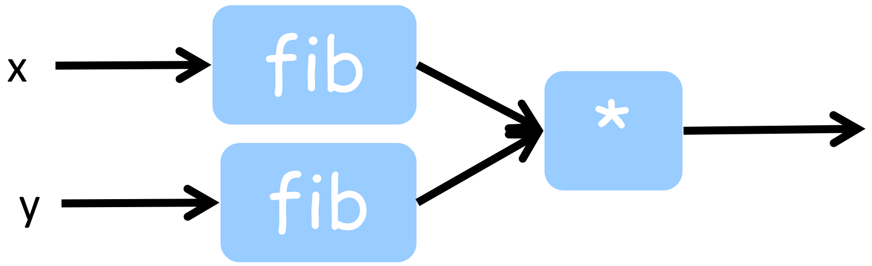


Our simple idea

- **Extend "Every call instantiates a copy of that function's RHS" to recursive functions**

```
fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib n = 1 + fib (n-1) + fib (n-2)

main x y = fib x * fib y
```

Question: what is in these "fib" boxes?

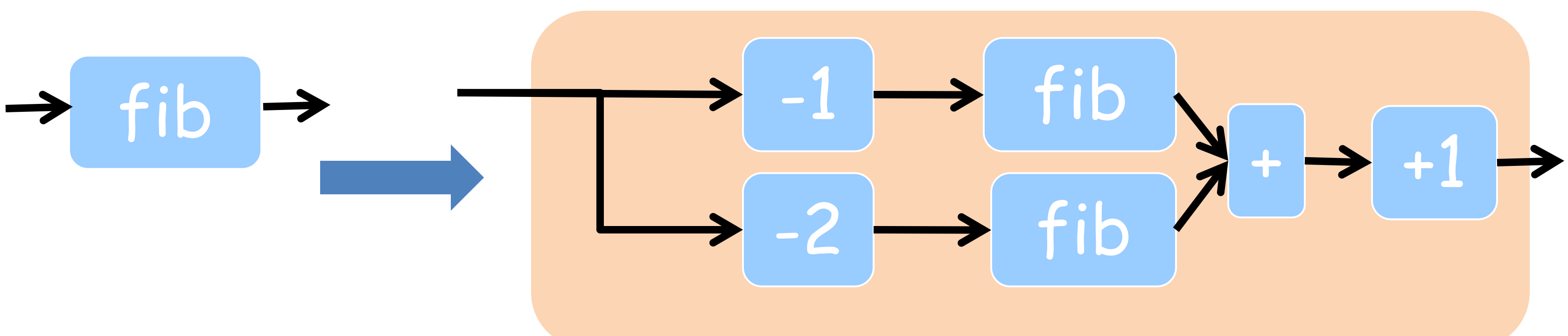


Our simple idea

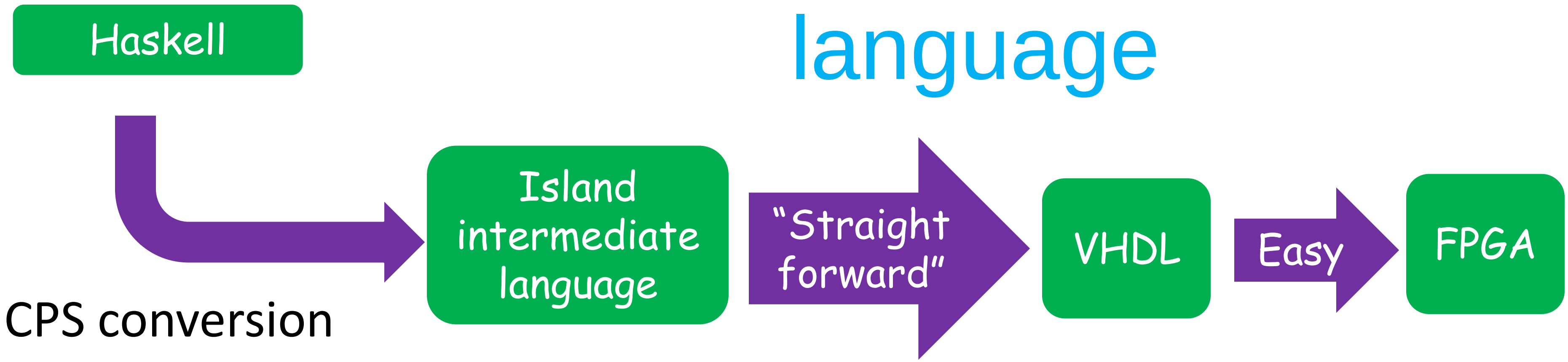
- **Extend "Every call instantiates a copy of that function's RHS" to recursive functions**

```
fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib n = 1 + fib (n-1) + fib (n-2)

main x y = fib x * fib y
```

Non-answer: instantiate the body of fib



Our “island” intermediate language



- The Island Intermediate Language is
  - Low level enough that it’s easy to convert to VHDL
  - High level enough that it’s (fairly) easy to convert Haskell into it

Microsoft Research

fib in IIL

```
fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib n = 1 + fib (n-1) + fib (n-2)
```

```
island {
  fib n = case n of
                0 -> return 1
                1 -> return 1
                _ -> let n1 = n-1
                     in recurse n1 [s1 n]

  s1 n r1 = let n2 = n-2
            in recurse n2 [s2 r1]

  s2 r1 r2 = let r = 1+r1+r2
             in return r }
```

```
fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib n = 1 + fib (n-1) + fib (n-2)
```

- s1, s2 correspond to return addresses, or continuations.
- Converting to IIL is just CPS conversion
- But there are choices to make

Entry point

Pop stack; apply saved state to result (i.e. 1)

Start again at entry point, pushing [s1 n] on stack

Resume here when [s1 n] is popped

```
island {
  fib n = case n of
                0 -> return 1
                1 -> return 1
                _ -> let n1 = n-1
                         in recurse fib n1 [s1 n]

  [s1 n] r1 = let n2 = n-2
                  in recurse fib n2 [s2 r1]

  [s2 r1] r2 = let r = 1+r1+r2
                    in return r  }
```

fib in IIL

```
fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib n = 1 + fib (n-1) + fib (n-2)
```

```
island {
  fib n = case n of
                 0 -> return 1
                 1 -> return 1
                 _ -> let n1 = n-1
                      in recurse fib n1 [s1 n]

  [s1 n] r1 = let n2 = n-2
              in recurse fib n2 [s2 r1]

  [s2 r1] r2 = let r = 1+r1+r2
               in return r }
```

State	Stack
fib 2	ε
fib 1	[s1 2]:ε
[s1 2] 1	ε
fib 0	[s2 1]:ε
[s2 1] 1	ε
return 3	

Each step is a combinatorial computation, leading to a new state

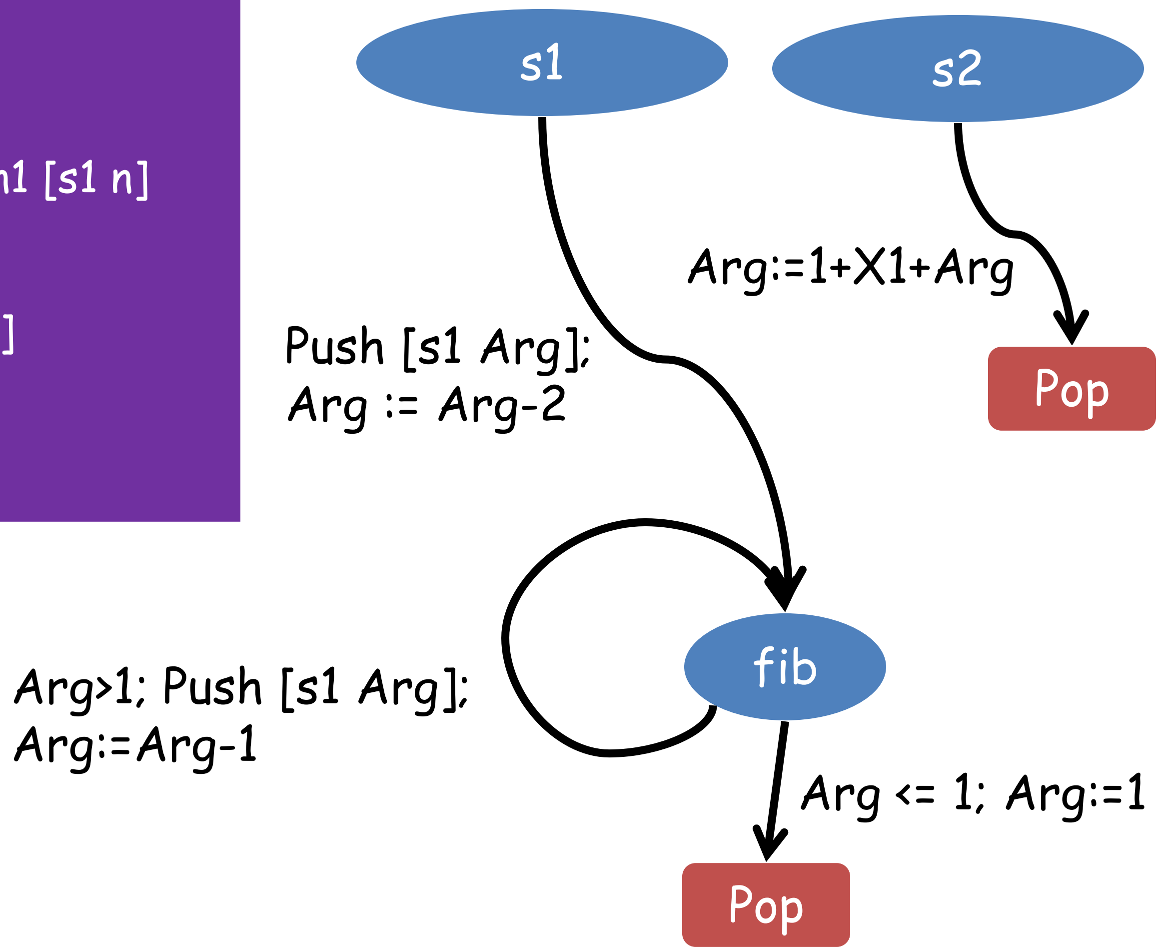
```
fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib n = 1 + fib (n-1) + fib (n-2)
```

```
island {
  fib n = case n of
                  0 -> return 1
                  1 -> return 1
                  _ -> let n1 = n-1
                       in recurse n1 [s1 n]

  [s1 n] r1 = let n2 = n-2
              in recurse n2 [s2 r1]

  [s2 r1] r2 = let r = 1+r1+r2
               in return r    }
```

Registers (state, memory)

	State	X1
	s1	2
Top	s2	7

Arg

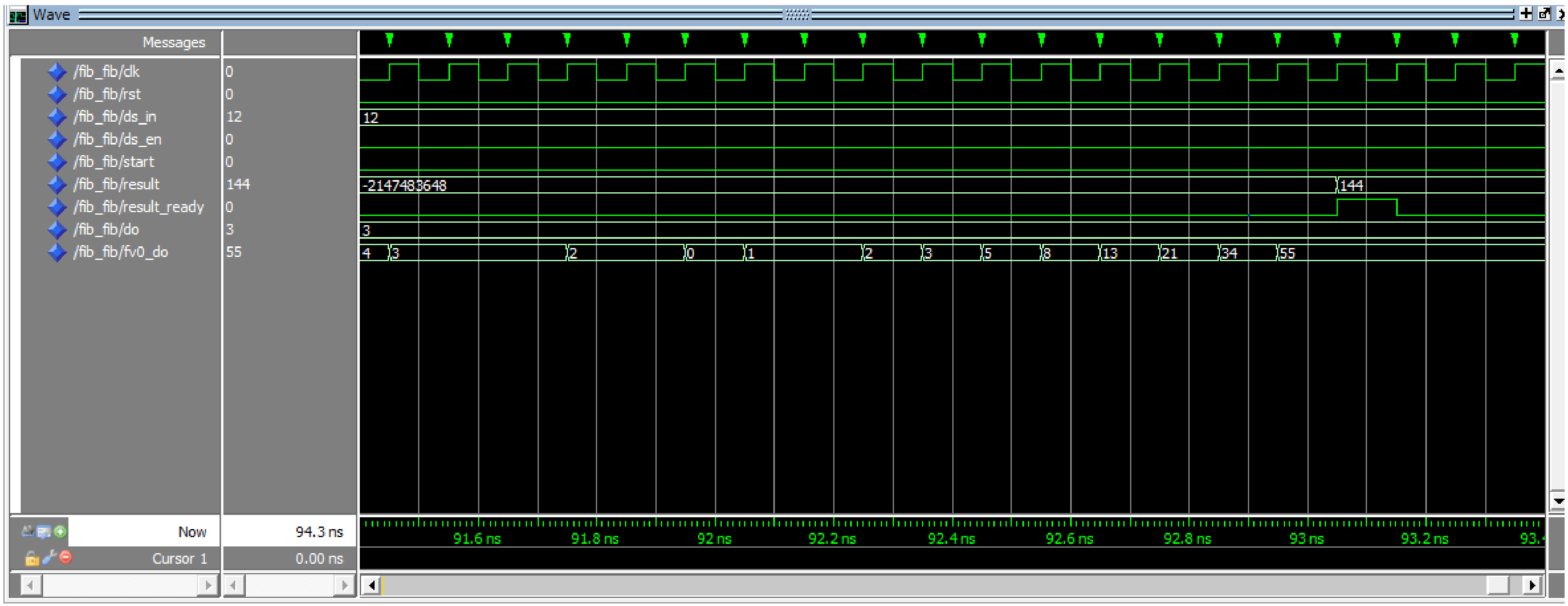


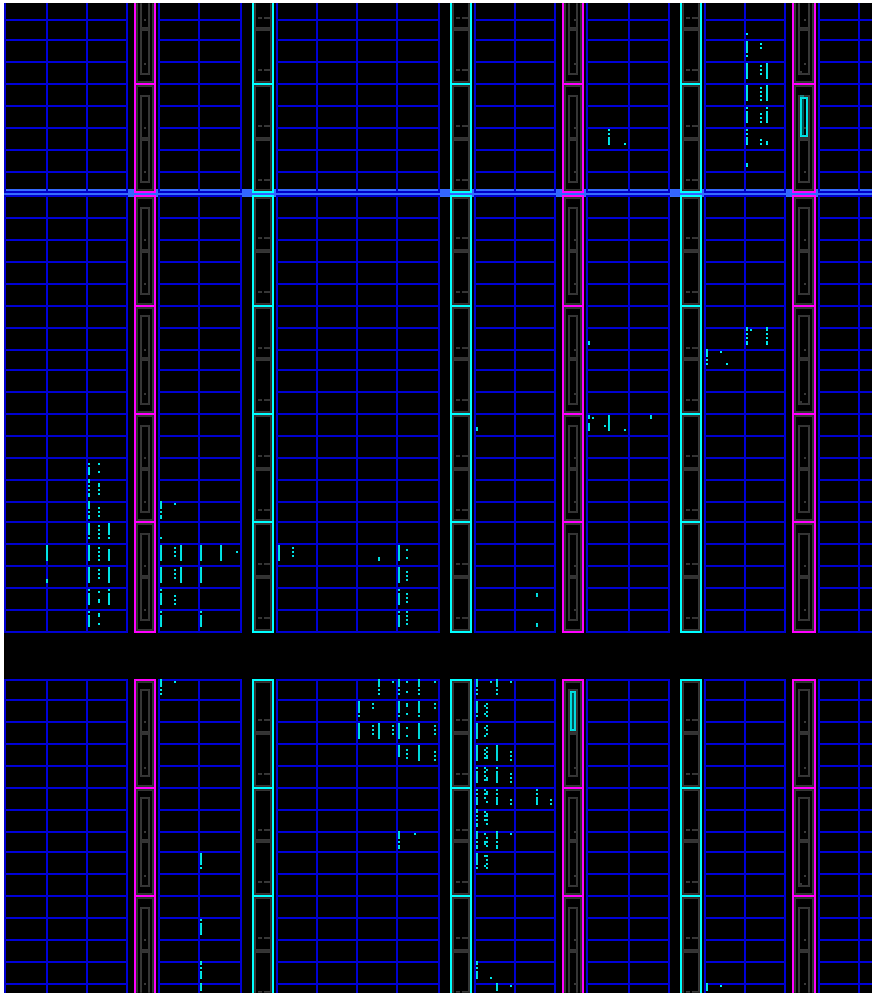
```
data IIR
   = ADD IIRExpr IIRExpr IIRExpr
   | SUB IIRExpr IIRExpr IIRExpr
   | MUL IIRExpr IIRExpr IIRExpr
   | GREATER IIRExpr IIRExpr IIRExpr
   | EQUAL IIRExpr IIRExpr IIRExpr
…
   | ASSIGN IIRExpr IIRExpr
   | CASE [IIR] IIRExpr [(IIRExpr, [IIR])]
   | CALL String [IIRExpr]
   | TAILCALL [IIRExpr]
   | RETURN IIRExpr
   | RECURSE [IIRExpr] State [IIRExpr]
   deriving (Eq, Show)
```

```haskell
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n
   = n1 + n2
     where
     n1 = fib (n - 1)
     n2 = fib (n - 2)
```

```
$ ./haskell2vhdl Fib.hs
Compiling Fib.hs
Writing Fib_fib.vhd ...
[done]
```

```
STATE 1 FREE
  PRECASE
    ds1 := ds
  CASE ds1
    WHEN 1 =>
      RETURN 1
    WHEN 0 =>
      RETURN 0
    WHEN others =>
      v0 := ds1 - 2
      RECURSE [v0] 2 [ds1]
  END CASE
STATE 3 FREE n2
  n1 := resultInt
  v2 := n1 + n2
  RETURN v2
STATE 2 FREE ds1
  n2 := resultInt
  v1 := ds1 - 1
  RECURSE [v1] 3 [n2]
```

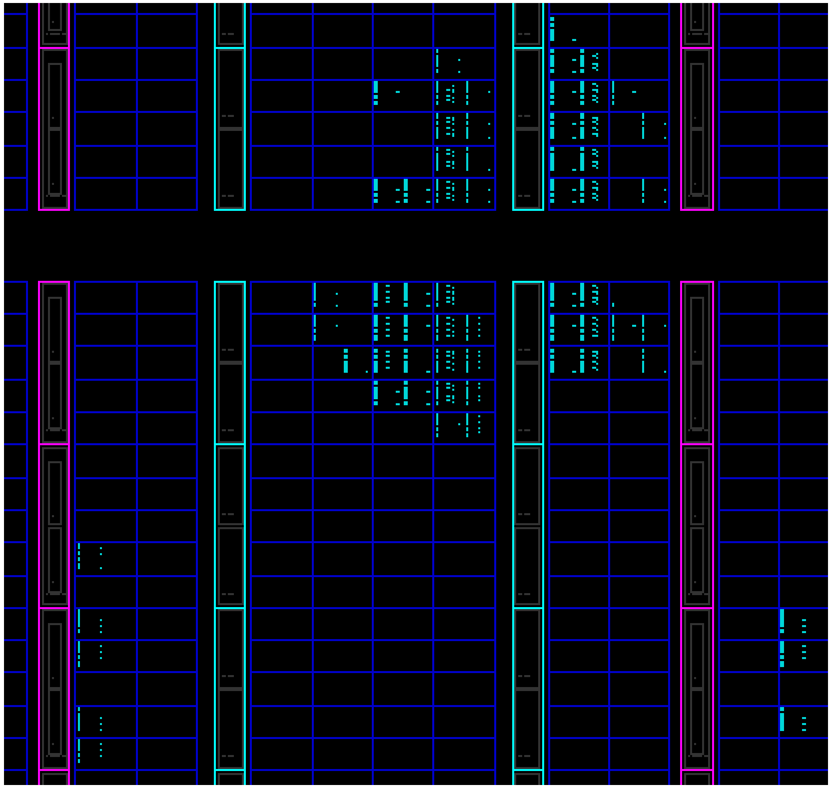

Wave
Messages
/fib_fib/clk 0
/fib_fib/rst 0
/fib_fib/ds_in 12
/fib_fib/ds_en 0
/fib_fib/start 0
/fib_fib/result 144
/fib_fib/result_ready 0
/fib_fib/do 3
/fib_fib/fv0_do 55
12
-2147483648
144
3
4 3 2 0 1 2 3 5 8 13 21 34 55
Now 94.3 ns
Cursor 1 0.00 ns
91.6 ns
91.8 ns
92 ns
92.2 ns
92.4 ns
92.6 ns
92.8 ns
93 ns
93.2 ns



```
gcd_dijkstra :: Int -> Int -> Int
gcd_dijkstra m n
  = if m == n then
      m
    else
      if m > n then
        gcd_dijkstra (m - n) n
      else
        gcd_dijkstra m (n - m)
```

```
STATE 1 FREE
 PRECASE
  v0 := m == n
  wild := v0
 CASE wild
  WHEN true =>
   RETURN m
  WHEN false =>
   PRECASE
    v1 := m > n
    wild1 := v1
   CASE wild1
    WHEN true =>
     v2 := m - n
     TAILCALL  [v2, n]
    WHEN false =>
     v3 := n - m
     TAILCALL  [m, v3]
   END CASE
 END CASE
```

```
$ make gcdtest
#    Time: 1450 ns  Iteration: 1  Instance: /gcdtest/fib_circuit
# ** Note: Parameter n = 6
#    Time: 1450 ns  Iteration: 1  Instance: /gcdtest/fib_circuit
# ** Note: GCD 12 126 = 6
#    Time: 1500 ns  Iteration: 0  Instance: /gcdtest
#   quit
```



```haskell
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n
  = n1 + n2
    where
    n1 = fib (n - 1)
    n2 = fib (n - 2)
```