Running Dynamic Algorithms on Static Hardware

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

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 = + n1 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
XC6VLX760 758,784 logic cells, 864 DSP blocks, 1,440 dual ported 18Kb RAMs

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

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

1,037,400,000,000 additions/second

332x1440
The holy grail

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

c := a + b ; -- cycle 0

 d := 2 * c ; -- cycle 1
 e := d - 5; -- cycle 2
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;
    end if;
  end process compute_fib;
end architecture manual;
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 1999
\[ \begin{align*}
\text{signal } S \text{ in } p \text{ end } & \frac{O\setminus\{S\},k}{\delta_1^k(\text{signal } S \text{ in } p' \text{ end})} \\
\text{signal } S \text{ in } p \text{ end } & \frac{O\setminus\{S\},k}{\delta_1^k(\text{signal } S \text{ in } p' \text{ end})}
\end{align*} \]
POPL 1999

\[
p \xrightarrow{O,k} \frac{p^+}{I \cup \{S\}} p' \quad S \in O
\]

\[
\text{signal } S \text{ in } p \text{ end } \frac{O \setminus \{S\}, k}{I} \delta^k_1\text{(signal } S \text{ in } p' \text{ end)}
\]

\[
p \xrightarrow{O,k} \frac{p'}{I \setminus \{S\}} p' \quad S \notin O
\]

\[
\text{signal } S \text{ in } p \text{ end } \frac{O,k}{I} \delta^k_1\text{(signal } S \text{ in } p' \text{ end)}
\]

\[
\text{emit } S \rightarrow \frac{\{S\}, 0}{\{A\}} \text{ nothing } S \in \{S\}
\]

\[
\text{emit } S \rightarrow \frac{\{S\}, 0}{\{A\}} \text{ nothing } S \in \{S\}
\]

\[
\text{signal } S \text{ in } \text{emit } S \text{ end } \frac{\{S\}, 0}{\{A\}} \text{ nothing}
\]

\[
\text{pause } \rightarrow \frac{\{S\}, 0}{\{A\}} \text{ nothing } S \notin \emptyset
\]

\[
\text{signal } S \text{ in } \text{pause } \rightarrow \frac{\{S\}, 0}{\{A\}} \text{ nothing } S \notin \emptyset
\]

\[
\text{signal } S \text{ in } \text{pause } \rightarrow \frac{\{S\}, 0}{\{A\}} \text{ nothing end}
\]

\[
\text{signal } S \text{ in } \text{nothing end}
\]

\[
\text{signal } S \text{ in } p \text{ end } \frac{O \setminus \{S\}, k^+}{I} \delta^k_1\text{(signal } S \text{ in } p^+ \text{ end)}
\]

\[
\text{signal } S \text{ in } p \text{ end } \frac{O \setminus \{S\}, k^-}{I} \delta^k_1\text{(signal } S \text{ in } p^- \text{ end)}
\]

\[
\text{signal } S \text{ in } p \text{ end } \frac{O \setminus \{S\}, k^-}{I} \delta^k_1\text{(signal } S \text{ in } p^- \text{ end)}
\]

\[
\text{signal } S \text{ in } p \text{ end } \frac{O \setminus \{S\}, k^+}{I} \delta^k_1\text{(signal } S \text{ in } p^+ \text{ end)}
\]

**Proof.** Structural induction on \( p \). Let us consider the case \( p = \text{"signal } S \text{ in } q \text{ end"} \). By hypothesis, \( p \xrightarrow{O_0, k_0} q_0 \). As \((\text{signal}++)\) or \((\text{signal}--)\) must be used to define this reaction, there exist \( O_0^-, k_0^-, q_0^-; O_0^+, k_0^+, q_0^+ \) such that:

\[
q \xrightarrow{O_0^-, k_0^-} q_0^- \quad \text{and} \quad q \xrightarrow{O_0^+, k_0^+} q_0^+
\]

Then, using Lemma 3.1,

- either \( S \notin O_0^-, S \notin O_0^+, O_0 = O_0^-, k_0 = k_0^-; p_0 = \delta^k_1\text{(signal } S \text{ in } q_0^- \text{ end)}\),
- or \( S \in O_0^-, S \in O_0^+, O_0 = O_0^+ \setminus \{S\}, k_0 = k_0^+, p_0 = \delta^k_1\text{(signal } S \text{ in } q_0^+ \text{ end)}\).
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

```haskell
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

```haskell
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
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)

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 :: 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 
}

<table>
<thead>
<tr>
<th>State</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>fib 2</td>
<td>ε</td>
</tr>
<tr>
<td>fib 1</td>
<td>[s1 2]:ε</td>
</tr>
<tr>
<td>[s1 2] 1</td>
<td>ε</td>
</tr>
<tr>
<td>fib 0</td>
<td>[s2 1]:ε</td>
</tr>
<tr>
<td>[s2 1] 1</td>
<td>ε</td>
</tr>
<tr>
<td>return 3</td>
<td></td>
</tr>
</tbody>
</table>

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