Functional Fibonacci to a Fast FPGA

Stephen A. Edwards*
Columbia University, Department of Computer Science

June 2012

Abstract

Through a series of mechanical transformation, I show how a three-line recursive Haskell function (Fibonacci) can be translated into a hardware description language—VHDL—for efficient execution on an FPGA. The goal of this report is to lay the groundwork for a compiler that will perform these transformations automatically, hence the development is deliberately pedantic.

1 Transforming fib into simple tail recursion

We begin by importing some types and a test generation library:[1]

```
import Data.Int (Int8, Int32)
import Test.QuickCheck
```

Below is our starting point: a naïve, recursive algorithm to compute Fibonacci numbers coded in Haskell. This is a terribly inefficient way to compute these numbers (an $O(2^n)$ algorithm instead of $O(n)$), but we will use it to illustrate our implementation strategy for recursive functions. We restrict the domains of the argument and result to simplify the hardware and illustrate how to mix different types of integers.

```
fib :: Int8 → Int32
fib 1 = 1
fib 2 = 1
fib n = fib (n−1) + fib (n−2)
```

We will build exactly one instance of a circuit for computing the *fib* function, so the two recursive calls may not be performed in parallel. Instead, we will schedule one before the other, followed by the addition of their results.

To express this sequencing, we transform the code to continuation-passing style [1]. To each function, this adds a continuation argument k: a function to which the result will be passed as an argument. The result of each function call is passed to a continuation that represents the rest of the computation in which the function call appeared.

```
fibc 1 k = k 1
fibc 2 k = k 1
fibc n k = fibc (n−1)
  ( λ n1 → fibc (n−2)
    ( λ n2 → k (n1 + n2)))

fib' n = fibc n ( λ x → x)
```

*Much of this work arose from discussions with Jared Pochtar, Satnam Singh, and Simon Peyton Jones.

[1] This report is written a in a “Literate Programming” style. All the Haskell and VHDL code fragments have been extracted directly from this document into source files and run through their respective compilers for verification.

Next, we name the three lambda terms and perform lambda-lifting to capture all free variables as arguments to these newly created function. For example, in the lambda term

```
(λ n1 → fibc (n−2) (λ n2 → k (n1 + n2))),
```

n and *k* appear free, so when we transform it into *fibd2*, *n* and *k* become arguments in addition to the continuation-passed argument *n1*.

```
fibd0 n        = fibd1 n fibd4
fibd1 1  k     = k 1
fibd1 2  k     = k 1
fibd1 n  k     = fibd1 (n−1) (fibd2 n k)
fibd2 n  k n1  = fibd1 (n−2) (fibd3 n1 k)
fibd3 n1 k n2  = k (n1 + n2)
fibd4 x        = x

fib'' n = fibd0 n
```

We also added a wrapper function, *fibd0*, to restrict all continuation-related operations to the *fibd* functions. While not strictly necessary, this will later simplify the circuitry responsible for managing continuations.

In this example, continuations are constructed in one of three ways:

1. as just *fibd4* in *fibd0*;
2. as (fibd2 n k) in *fibd1*, where *n* is an integer related to depth of recursion and *k* is a continuation; and
3. as (fibd3 n1 k) in *fibd2*, where *n1* is an integer related to a partial result and *k* is a continuation.

This immediately suggests they can be encoded as a recursive type: this is the role of the *Cont* type in the code below.

We need one final type to de-functionalize [2] this code: something that distinguishes among the two remaining functions that do not appear as continuations; *fib0*, which does not take a continuation argument, and *fib1*, which does; and the calls to a continuation. This is the role of the *Call* type in the code below.

We are finally in a simple form: a single function that either transforms its arguments with simple arithmetic and calls itself tail-recursively or simply returns part of its argument. The *Cont* type encodes continuations in the form of a stack; the *Call* type effectively merges multiple functions into a single one.

```
data Cont = Fib2 Int8 Cont
          | Fib3 Int32 Cont
          | Fib4

data Call  = Fib0 Int8
           | Fib1 Int8 Cont
           | Cont Cont Int32

fibp       (Fib0 n)           = fibp (Fib1 n Fib4)
fibp       (Fib1 1  k)        = fibp (Cont k 1)
fibp       (Fib1 2  k)        = fibp (Cont k 1)
fibp       (Fib1 n  k)        = fibp (Fib1 (n−1) (Fib2 n  k))
fibp (Cont (Fib2 n  k) n1)    = fibp (Fib1 (n−2) (Fib3 n1 k))
fibp (Cont (Fib3 n1 k) n2)    = fibp (Cont k (n1 + n2))
```

```
fibp (Cont (Fib4) x)          = x

fib ''' n = fibp (Fib0 n)
```

Finally, we add a few simple QuickCheck tests that verify that the four versions produce identical results on small integers.

```
prop_fib01equal :: Int → Property
prop_fib01equal n = n > 0 && n < 15 ⟹ fib (fromIntegral n) == fib ' n

prop_fib12equal :: Int → Property
prop_fib12equal n = n > 0 && n < 15 ⟹ fib ' n == fib '' n

prop_fib23equal :: Int → Property
prop_fib23equal n = n > 0 && n < 15 ⟹ fib '' n == fib ''' (fromIntegral n)

main = do
  quickCheck prop_fib01equal
  quickCheck prop_fib12equal
  quickCheck prop_fib23equal
```

2 Coding fib in VHDL

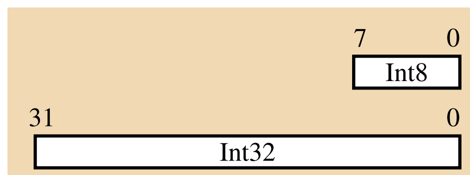
2.1 Types package

First, we'll define a package of VHDL types and functions that represent and manipulate the types in the Haskell program. The main challenge here is that VHDL does not support "union" types, so we write explicit constructor and accessor functions for them.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

package fib_package is
```

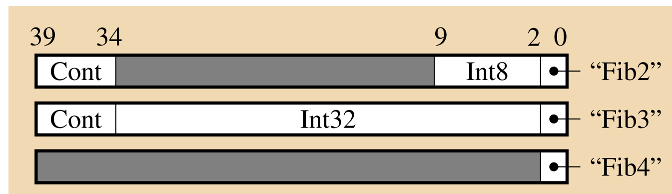
The layout and definition of eight- and thirty-two-bit integers are straightforward. We adopt a little-endian style.



We define both a VHDL constant and type for each:

```
  constant INT8_W : integer := 8;
  subtype int8_t   is unsigned(INT8_W−1 downto 0);
  constant INT32_W : integer := 32;
  subtype int32_t  is unsigned(INT32_W−1 downto 0);
```

The *Cont* type is more complicated since it is a union and will be stored in memory. A fundamental trick here is that the Cont pointer fields appear in the same position in both the Fib2 and Fib3 types, making them easy to reconstitute from the address in which the data is stored in memory. This works because the continuations follow a stack discipline and therefore can have a simple memory management scheme—a classical stack pointer.



We begin by defining constants and a type for the tag field.

```
constant CONT_TAG_W : integer := 2;
subtype cont_tag_t is unsigned(CONT_TAG_W − 1 downto 0);
constant FIB2_TAG : cont_tag_t := "00";
constant FIB3_TAG : cont_tag_t := "01";
constant FIB4_TAG : cont_tag_t := "10";
```

Now, constants and a type for the pointer type:

```
constant CONT_PTR_W : integer := 6;
subtype cont_ptr_t is unsigned(CONT_PTR_W − 1 downto 0);
```

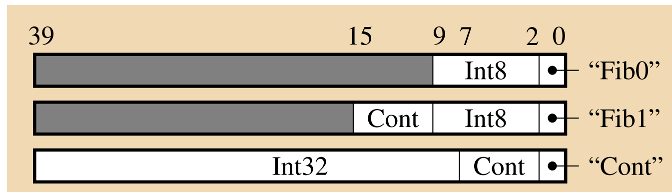
Next, a constant and type for the type itself.

```
constant CONT_W : integer := CONT_TAG_W + INT32_W + CONT_PTR_W;
subtype cont_t is unsigned(CONT_W − 1 downto 0);
```

We are not going to store the continuation pointers in memory (they are redundant since the type follows a stack discipline), so we will define yet another constant and type for the data we'll put in memory and a type for the memory itself.

```
constant CONT_IN_MEM_W : integer := CONT_TAG_W + INT32_w;
subtype cont_in_mem_t is unsigned(CONT_IN_MEM_W − 1 downto 0);
constant CONT_MEM_SIZE : integer := 2 ** CONT_PTR_W;
type cont_mem_t is array(0 to CONT_MEM_SIZE − 1) of cont_in_mem_t;
```

The *Call* type is never stored in a memory, so its layout is a little more mechanical.



Here are the constants and type for *Call*:

```
constant CALL_TAG_W : integer := 2;
subtype call_tag_t is unsigned(CALL_TAG_W − 1 downto 0);
constant FIB0_TAG : call_tag_t := "00";
constant FIB1_TAG : call_tag_t := "01";
constant CONT_TAG : call_tag_t := "10";

constant CALL_W : integer := CALL_TAG_W + INT32_W + CONT_PTR_W;
subtype call_t is unsigned(CALL_W − 1 downto 0);
```

This completes the type definitions in the package.

Now, we define functions for constructing and accessing fields in these types. First, we define functions for the *Call* type. The first three are type constructors, the "is" functions test the tag field, and the remainder access (numbered) fields in the types.

```
  function Fib0(n : int8_t )                    return call_t ;
  function Fib1(n : int8_t ; k : cont_ptr_t )   return call_t ;
  function Cont(k : cont_ptr_t ; n : int32_t )  return call_t ;

  function is_Fib0 (a : call_t ) return boolean;
  function Fib0_1  (a : call_t ) return int8_t ;

  function is_Fib1 (a : call_t ) return boolean;
  function Fib1_1  (a : call_t ) return int8_t ;
  function Fib1_2  (a : call_t ) return cont_ptr_t ;

  function is_Cont (a : call_t ) return boolean;
  function Cont_1  (a : call_t ) return cont_ptr_t ;
  function Cont_2  (a : call_t ) return int32_t ;
```

Now, functions for the Cont type. One anomaly is the definition for the Fib4 constructor: Since VHDL does not support zero-argument functions, it is defined as a constant.

```
  function Fib2(n : int8_t  ; k : cont_ptr_t ) return cont_t ;
  function Fib3(n : int32_t ; k : cont_ptr_t ) return cont_t ;
  constant Fib4 : cont_t :=
     "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX" &
     FIB4_TAG;

  function is_fib2 (a : cont_t ) return boolean;
  function Fib2_1  (a : cont_t ) return int8_t ;
  function Fib2_2  (a : cont_t ) return cont_ptr_t ;

  function is_Fib3 (a : cont_t ) return boolean;
  function Fib3_1  (a : cont_t ) return int32_t ;
  function Fib3_2  (a : cont_t ) return cont_ptr_t ;

  function is_Fib4 (a : cont_t ) return boolean;

end fib_package ;
```

Now we define all these functions. The code is tedious but straightforward—just the sort of thing you'd want a compiler to generate. Each follows directly from the bit-wise layout of the types shown earlier. Again, we begin with the functions related to the *Call* type.

```
package body fib_package is
  function Fib0(n : int8_t ) return call_t is begin
    return "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX" & -- 30 X's
           n & FIB0_TAG; end Fib0;
  function Fib1(n : int8_t  ; k : cont_ptr_t ) return call_t is begin
    return "XXXXXXXXXXXXXXXXXXXXXXXX" & -- 24 X's
           k & n & FIB1_TAG; end Fib1;
  function Cont(k : cont_ptr_t  ; n : int32_t  ) return call_t is begin
    return n & k & CONT_TAG; end Cont;
  function is_Fib0 (a : call_t ) return boolean is begin
    return a(CALL_TAG_W - 1 downto 0) = FIB0_TAG; end is_Fib0;
  function Fib0_1(a : call_t ) return int8_t is begin
    return a(INT8_W - 1 + CALL_TAG_W downto CALL_TAG_W); end Fib0_1;
```

```
function is_Fib1 (a : call_t ) return boolean is begin
  return a(CALL_TAG_W − 1 downto 0) = FIB1_TAG; end is_Fib1;
function Fib1_1(a : call_t ) return int8_t is begin
  return a(INT8_W − 1 + CALL_TAG_W downto CALL_TAG_W); end Fib1_1;
function Fib1_2(a : call_t ) return cont_ptr_t is begin
  return a(CONT_PTR_W − 1 + CALL_TAG_W + INT8_W downto
                                CALL_TAG_W + INT8_W); end Fib1_2;
function is_Cont (a : call_t ) return boolean is begin
  return a(CALL_TAG_W − 1 downto 0) = CONT_TAG; end is_cont;
function Cont_1(a : call_t ) return cont_ptr_t is begin
  return a(CONT_PTR_W − 1 + CALL_TAG_W downto
                                  CALL_TAG_W); end Cont_1;
function Cont_2(a : call_t ) return int32_t is begin
  return a(INT32_W − 1 + CALL_TAG_W + CONT_PTR_W downto
                             CALL_TAG_W + CONT_PTR_W); end Cont_2;
```

Now, the functions for the *Cont* type. One trick here is that the functions that return the *Cont* field in a *Fib3* actually calls the function for accessing the field in a *Fib2*. This is part of the trickery enabling *Cont* objects to be stored as a simple stack in memory.

```
function Fib2(n : int8_t ; k : cont_ptr_t ) return cont_t is begin
  return k & "XXXXXXXXXXXXXXXXXXXXXXXX" & -- 24 X's
         n & FIB2_TAG; end Fib2;
function Fib3(n : int32_t ; k : cont_ptr_t ) return cont_t is begin
  return k & n & FIB3_TAG; end Fib3;
function is_fib2 (a : cont_t ) return boolean is begin
   return a(CONT_TAG_W − 1 downto 0) = FIB2_TAG; end is_Fib2;
function Fib2_1 (a : cont_t ) return int8_t is begin
  return a(INT8_W − 1 + CONT_TAG_W downto CONT_TAG_W); end Fib2_1;
function Fib2_2 (a : cont_t ) return cont_ptr_t is begin
  return a(CONT_PTR_W − 1 + CONT_TAG_W + INT32_W downto
                                CONT_TAG_W + INT32_W); end Fib2_2;
function is_Fib3 (a : cont_t ) return boolean is begin
  return a(CONT_TAG_W − 1 downto 0) = FIB3_TAG; end is_Fib3;
function Fib3_1 (a : cont_t ) return int32_t is begin
  return a(INT32_W − 1 + CONT_TAG_W downto CONT_TAG_W); end Fib3_1;
function Fib3_2 (a : cont_t ) return cont_ptr_t is begin
  return Fib2_2(a); end Fib3_2;
function is_Fib4 (a : cont_t ) return boolean is begin
  return a(CONT_TAG_W − 1 downto 0) = FIB4_TAG; end is_Fib4;

end fib_package ;
```

2.2 The Fibp Block

Next, we will define an entity/architecture pair for the core *fibp* block, which contains combinational logic that performs the pattern matching and generates the argument for the tail call and the Cont type's constructor. The entity definition is straightforward because we defined types in *fib_package*.

While the argument to the *fibp* function is a single *Call* object, the function often needs to examine the *Cont* object embedded in it. Later, we will arrange this to be delivered through the *arg_cont* argument.

Control of the *Cont* constructor circuit is the main interesting thing going on here. In rules where a new *Cont* object is created, i.e., where *Fib2*, *Fib3*, or *Fib4* appears in a function definition, *cont_go* is asserted to create a new *Cont* object. The *Cont* constructor returns a pointer to this new object through *cont_ptr*, so this is a tricky combinational path: within a single cycle, *cont_go* is sent to the constructor, which computes the new address and returns it to be used by the *Call* constructors.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.fib_package.all;

entity fibp is
  port (
    go       : in  std_logic;
    arg      : in  call_t;
    arg_cont : in  cont_t;      -- Cont inside the Call, if applicable
    ready    : out std_logic;
    result   : out int32_t;
    tail_go  : out std_logic;   -- Indicates a tail call
    tail_arg : out call_t;      -- Argument to tail call
    cont_go  : out std_logic;   -- Cont constructor
    cont_arg : out cont_t;      -- Argument to Cont constructor
    cont_ptr : in  cont_ptr_t   -- Pointer to newly constructed Cont
  );
end entity;
```

Now for the architecture, which is a single combinational process. By design, this is a largely mechanical rewriting of the pattern matching and constructor rules of the final Haskell code. One thing that was lost is the names bound to fields in the pattern matching. Instead, each is expressed explicitly using one of the many accessor functions.

```
architecture rtl of fibp is

begin
  fibp : process (go, arg, arg_cont, cont_ptr )
  begin
    ready    <= '0'; result    <= (others => 'X');
    tail_go  <= '0'; tail_arg  <= (others => 'X');
    cont_go  <= '0'; cont_arg  <= (others => 'X');

    if go = '1' then

      -- fibp (Fib0 n) = fibp (Fib1 n Fib4)
      if is_Fib0 (arg) then
        cont_go  <= '1'; cont_arg <= Fib4;
        tail_go  <= '1'; tail_arg <= Fib1(Fib0_1(arg), cont_ptr );

      -- fibp (Fib1 1 k) = fibp (Cont k 1)
      elsif is_Fib1 (arg) and Fib1_1(arg) = to_unsigned (1, INT8_W) then
        tail_go  <= '1';
        tail_arg <= Cont(Fib1_2(arg), to_unsigned (1, INT32_W));

      -- fibp (Fib1 2 k) = fibp (Cont k 1)
      elsif is_Fib1 (arg) and Fib1_1(arg) = to_unsigned (2, INT8_W) then
        tail_go  <= '1';
        tail_arg <= Cont(Fib1_2(arg), to_unsigned (1, INT32_W));

      -- fibp (Fib1 n k) = fibp (Fib1 (n-1) (Fib2 n k))
      elsif is_Fib1 (arg) then
        cont_go <= '1'; cont_arg <= Fib2( Fib1_1(arg), Fib1_2(arg ));
        tail_go <= '1'; tail_arg <= Fib1(Fib1_1(arg) - 1, cont_ptr );

      -- fibp (Cont (Fib2 n k) n1) = fibp (Fib1 (n-2) (Fib3 n1 k))
      elsif is_Cont(arg) and is_Fib2 ( arg_cont ) then
        cont_go <= '1'; cont_arg <= Fib3( Cont_2(arg ), Fib2_2( arg_cont ));
        tail_go <= '1'; tail_arg <= Fib1( Fib2_1( arg_cont ) - 2, cont_ptr );

      -- fibp (Cont (Fib3 n1 k) n2) = fibp (Cont k (n1 + n2))
      elsif is_Cont(arg) and is_Fib3 ( arg_cont ) then
        tail_go  <= '1';
        tail_arg <= Cont(Fib3_2(arg_cont ), Fib3_1( arg_cont ) + Cont_2(arg ));

      -- fibp (Cont (Fib4) x) = x
      elsif is_Cont(arg) and is_Fib4 ( arg_cont ) then
        ready <= '1'; result <= Cont_2(arg);
      end if;
    end if;
  end process;
end architecture;
```

2.3 The Cont_Ctrl Block

This is essentially a memory controller for the *Cont* type: a stack. It's complicated by the need to produce the current "top of stack" by default, something needed by the *fibp* block when it invokes a continuation.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.fib_package.all;

entity cont_ctrl is
  port (
    clk     : in  std_logic;
    go      : in  std_logic;
    arg     : in  cont_t;
    result  : out cont_ptr_t;
    call    : in  call_t;
    current : out cont_t
  );
end entity;
```

We need a few internal signals: the stack pointer (i.e., the memory address to be read/written), a write signal, data to be written, and finally the array representing the local stack memory itself.

```
architecture rtl of cont_ctrl is

  signal ptr : cont_ptr_t;
  signal wr : std_logic;
  signal write_data : cont_in_mem_t;
  signal mem : cont_mem_t;

begin
```

The first part of the architecture is the combinational process that controls the operation of the memory. There are two cases: when *go* is asserted, a new *Cont* object is created and written into memory; otherwise, the "top-of-stack" is read, using the *Cont* field in the *call* argument as the address. Note that this code assumes the *Cont* fields are in the same place in both *Fib2* and *Fib3* objects.

Although it may appear *ptr* and *result* could be merged, splitting them avoids a (false) combinational cycle involving the *fibp* block.

```
 control : process (go, arg, call)
begin
  wr <= '0';
  ptr <= (others => 'X');
  write_data <= (others => 'X');
  result <= (others => 'X');

  if go = '1' then
    wr <= '1';
    write_data <= arg(CONT_IN_MEM_W - 1 downto 0);
    if is_Fib4(arg) then
      ptr      <= to_unsigned(0, CONT_PTR_W);
      result   <= to_unsigned(0, CONT_PTR_W);
    elsif is_Fib3(arg) or is_Fib2(arg) then
      ptr      <= Fib3_2(arg) + 1;
      result   <= Fib3_2(arg) + 1;
    end if;

  elsif is_Fib1(call) then
    ptr <= Fib1_2(call);
  elsif is_Cont(call) then
    ptr <= Cont_1(call);
  end if;
end process;
```

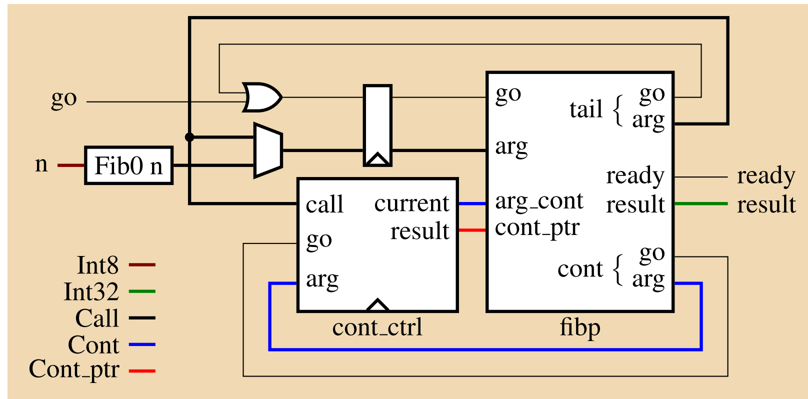
The last part of the architecture is the process describing the memory in which the *Cont* objects are placed. It is deliberately simple so that the synthesis tools will correctly infer RAM from it. In particular, it is a one-cycle RAM with write-through. The one bit of magic is that the *Cont* field of both *Fib2* and *Fib3* are in the same place and reconstituted from the address being read or written.

```
 ram : process (clk)
 begin
   if rising_edge(clk) then

     if wr = '1' then
       mem(to_integer(ptr)) <= write_data;
     end if;

     current(CONT_IN_MEM_W - 1 downto 0) <= mem(to_integer(ptr));
     current(CONT_PTR_W - 1 + CONT_IN_MEM_W downto
                              CONT_IN_MEM_W) <= ptr - 1;
   end if;
 end process;

end architecture;
```



This is nearly a direct translation of the block diagram. The interface is, by design, boilerplate.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.fib_package.all;

entity fib is
  port (
    clk    : in  std_logic;
    go     : in  std_logic;
    arg    : in  int8_t;
    ready  : out std_logic;
    result : out int32_t
  );
end entity;
```

The internal signals are controls for *fibp*, its tail recursion, and the *Cont* memory controller.

```
architecture rtl of fib is
  signal fibp_go       : std_logic;
  signal fibp_arg      : call_t;
  signal fibp_arg_cont : cont_t;
  signal tail_go       : std_logic;
  signal tail_arg      : call_t;
  signal cont_go       : std_logic;
  signal cont_arg      : cont_t;
  signal cont_ptr      : cont_ptr_t;
```

The body consists of instances of the *fibp* and *cont_ctrl* blocks defined earlier. Note that only the latter has a clock input.

```
begin
  fibp : entity work.fibp port map (
    go        => fibp_go,
    arg       => fibp_arg,
    arg_cont  => fibp_arg_cont,
    ready     => ready,
    result    => result,
    tail_go   => tail_go,
    tail_arg  => tail_arg,
    cont_go   => cont_go,
    cont_arg  => cont_arg,
    cont_ptr  => cont_ptr
  );

  cont_ctrl : entity work.cont_ctrl port map (
    clk      => clk,
    go       => cont_go,
    arg      => cont_arg,
    result   => cont_ptr,
    call     => tail_arg,
    current  => fibp_arg_cont
  );
```

Finally, we need a sequential process that either starts or tail-recurses *fibp*. Implicit here is a primitive arbiter: tail recursion takes precedence over *go*, which the environment shouldn't generate while the system is computing anyway.

```
  control : process (clk)
  begin
    if rising_edge (clk) then
      if tail_go = '1' then
        fibp_go <= '1'; fibp_arg <= tail_arg;
      elsif go = '1' then
        fibp_go <= '1'; fibp_arg <= Fib0(arg);
      else
        fibp_go <= '0';
        fibp_arg <= (others => 'X');
      end if;
    end if;
  end process;

end architecture;
```

References

[1] Andrew Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[2] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference*, pages 717–740, 1972. Reprinted in Higher-Order and Symbolic Computation 11(4):363–397 Dec. 1998.