A Finer Functional Fibonacci on a Fast FPGA

Stephen A. Edwards
Columbia University, Department of Computer Science
Technical Report CUCS–XXX-13

February 2013

Abstract

Through a series of mechanical, semantics-preserving transformations, I show how a three-line recursive Haskell program (Fibonacci) can be transformed to a hardware description language—Verilog—that can be synthesized on an FPGA. This report lays groundwork for a compiler that will perform this transformation automatically.

1 Introduction

This is an extension of my earlier report [2]. Here, I model memory operations directly in Haskell and target Verilog instead of VHDL.

Following the style of Reynolds [4], I start with a high-level functional program and transform it, step by step, into a far simpler dialect that has a trivial translation into synchronous hardware suitable for running on an FPGA. The longer-term goal is to automate all of these tasks in a compiler to enable most Haskell programs to be transformed into hardware.

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

2 Transforming Fib to Hardware Form

Our starting point is the following naïve algorithm for computing Fibonacci numbers. With an eye toward a hardware implemention, we have constrained the domain and range of this function to be 8 and 32-bit unsigned integers, respectively.

```
fib  :: Word8 → Word32
fib 1 = 1                           -- Base case
fib 2 = 1                           -- Base case
fib n = fib (n−1) + fib (n−2)       -- Recurse twice and sum results
```

2.1 *Scheduling: Transforming to Continuation-Passing Style*

First, to make the control flow, and in particular the sequencing of recursive calls, explicit, we transform this to continuation-passing style [1]. We split the operations in the third case into separate functions and add a continuation argument *k*—a function responsible for continuing the computation to which the result of the recursive call will be passed.

```
fib′  :: Word8 → (Word32 → Word32) → Word32
fib′ 1 k = k 1                  -- Base case
fib′ 2 k = k 1                  -- Base case
fib′ n k = fib′ (n−1)           -- First recursive call
   (λn1 → fib′ (n−2)            -- Second recursive call
      (λn2 → k (n1 + n2)))      -- Sum results

fib  :: Word8 → Word32
fib n = fib′ n (λx → x)
```

2.2 *Capturing Free Variables: Lambda Lifting*

Next, we name each lambda term and perform lambda lifting [3] to capture all free variables as arguments. This replaces true recursion with tail recursion with continuations.

```
call  ::  Word8 → (Word32 → Word32) → Word32               -- Entry
c1    ::  Word8 → (Word32 → Word32) → Word32 → Word32      -- Continuation 1
c2    ::  Word32 → (Word32 → Word32) → Word32 → Word32     -- Continuation 2
c3    ::  Word32 → Word32                                  -- Return

call   1   k      = k  1                     -- Base case (return)
call   2   k      = k  1                     -- Base case (return)
call   n   k      =  call  (n−1)  (c1  n  k)   -- First recursive call (call)
c1     n   k  n1  =  call  (n−2)  (c2  n1  k)  -- Second recursive call (call)
c2     n1  k  n2  = k  (n1  +  n2)           -- Sum Results (return)
c3     x          = x                        -- Return final result

fib   ::  Word8 → Word32
fib  n =  call  n  c3
```

2.3 Representing Continuations with a Type

Next, we represent continuations with an algebraic data type. There are three ways continuations are "constructed" in this example: *c1* with two arguments, *c2* with two arguments, and *c3* by itself, so the type becomes

```
data Continuation = C1 Word8  Continuation -- Second recursive call
                  | C2 Word32 Continuation -- Sum Results
                  | C3                     -- Return final result
```

We fractured the *fib* function into three pieces (*call*, *c1*, *c2*), but for in sequential operation, such as our hardware will eventually exhibit, only one is ever active at once. We can encode the various calls with another algebraic data type. There are two choices: *call* is called directly; the other three are continuations.

```
data Call = Call Word8 Continuation     -- Tail-Recursive Entry
          | Cont Continuation  Word32   -- Continuation
```

With this, we can combine these pieces into a single tail-recursive function and a wrapper:

```
fib' :: Call → Word32
fib' (Call         1  k) = fib' (Cont k  1)              -- Base case
fib' (Call         2  k) = fib' (Cont k  1)              -- Base case
fib' (Call         n  k) = fib' (Call (n−1) (C1 n  k))  -- First call
fib' (Cont (C1 n  k) n1) = fib' (Call (n−2) (C2 n1 k))  -- Second call
fib' (Cont (C2 n1 k) n2) = fib' (Cont k (n1 + n2))      -- Sum results
fib' (Cont (C3)        x) = x                            -- Return result

fib :: Word8 → Word32
fib n = fib' (Call n C3)
```

2.4 *Making In-Memory Types Explicit*

In part, our memory machinery uses pointers in the usual way, although such pointers are local to each unique memory. Here, we choose to use eight bits per pointer because that is the smallest standard integer type in Haskell; selecting appropriate sizes for each pointer will be an interesting exercise. This also raises the question of whether we want to introduce the notion of global memory addresses.

```
type ContPtr = Word8 -- Pointer to a Continuation object
```

Because we know the continuations are always in a stack, there is no need to store their "next" pointers in memory.

```
data ContBits = CB1 Word8   -- Second Recursive Call
              | CB2 Word32  -- Sum Results
              | CB3         -- Bottom of stack/return final result
```

We model the small memory in which we will store this data on the FPGA with an immutable Haskell array that begins life uninitialized.

```
type ContMem = Array ContPtr ContBits

emptyMem :: ContMem
emptyMem = array (minBound::ContPtr, maxBound::ContPtr) []
```

While a reference to a continuation is atomic to a user-level program, in our Haskell implementation it consists of a pointer-array pair.

```
type ContRef = (ContPtr, ContMem)

data Continuation = C1 Word8  ContRef  -- Second Recursive Call
                  | C2 Word32 ContRef  -- Sum Results
                  | C3                 -- Return final result
```

2.5 *Making Memory Operations Explicit*

Now, we introduce two functions: *store*, which stores a new continuation in memory and returns a reference, and *load*, which, given a reference, returns a contintuation.

The *store* function looks at the successor of the continuation being stored and uses that to calculate the address of the new continuation (i.e., the address just above the successor). The base case is *C3*, which is always placed in address 0.

```
store  :: Continuation → ContRef
store c = let (p, m, c′) = case c of
                             C1 n  (p, m) → (p, m, CB1 n)
                             C2 n1 (p, m) → (p, m, CB2 n1)
                             C3           → (0, emptyMem, CB3) in
          let p′ = p + 1 in        -- Place in next memory location
          (p′, m // [(p′, c′)])    -- Write memory
```

This *store* implementation is a little unorthodox. A traditional write operation takes an address and a value; our *store* only takes a value, much like a constructor in an object-oriented language, but the value directs where it should be stored in the stack.

The *C3* continuation is unique in that there is ever at most one of them in existence at any time. It would be possible to make it implicit: never stored in memory but encoded as a special index (e.g., −1). However, I suspect using a sentinel value in memory is faster since it simplifies the *load* logic (which does not have to check for a special index), even though it wastes some memory and, in this case, actually requires an additional bit (to distinguish three cases instead of two). However, I should compare these two with an experiment.

The *load* function has the opposite effect of *store* and has more traditional types: given a reference to a continuation, it returns the data it contains. In addition to using the data in memory, *load* also reconstructs the reference to the next continuation by subtracting one from the reference. However, because *load* takes a cycle on our FPGAS (i.e., the address is presented a clock cycle before the data becomes available) we break *load* into two functions and use the function call to denote the clock cycle boundary. There is a scheduling question here: where should the $p - 1$ operation reside? I arbitrarily put it before the clock cycle.

```
load  :: ContRef → Continuation
load (p, m) = let p′ = p − 1 in          -- Successor just below us
                  loadp (p′, m, m ! p)    -- Read memory

loadp :: (ContPtr, ContMem, ContBits) → Continuation
loadp (p′, m, d) = case d of
                     CB1 n  → C1 n  (p′, m)  -- Reconstruct
                     CB2 n1 → C2 n1 (p′, m)
                     CB3    → C3
```

2.6 Inserting Explicit Memory Operations

We now have all the pieces to transform the *fib* function into a form suitable for hardware. We scheduled the operations, especially the recursive calls, by transforming the code to continuation-passing style, eliminated free variables using lambda lifting, introduced an explicit continuation type to make it purely tail-recursive, removed the recursion from the continuation type by introducing explicit references, and finally introduced memory management functions able to load and store continuation objects to and from memory.

We need to introduce one additional type: a concrete representation of the *Call* type. As an optimization, we will "inline" the value of the continuation object when it is actually used (i.e., being pattern-matched) in the *Cont* case; we will leave it as a reference in the *Call* case since we do not need to know its contents.

```
data Call = Call Word8 ContRef              -- Tail-Recursive Entry
          | Cont Continuation Word32        -- Continuation
```

With this final type in hand, we can now express the *fib* function in a form suitable for hardware translation.

```
fibp :: Call → Word32
fibp (Call        1   kr) = fibp (Cont (load kr) 1)
fibp (Call        2   kr) = fibp (Cont (load kr) 1)
fibp (Call        n   kr) = fibp (Call (n−1) (store (C1 n kr)))
fibp (Cont (C1 n  kr) n1) = fibp (Call (n−2) (store (C2 n1 kr)))
fibp (Cont (C2 n1 kr) n2) = fibp (Cont (load kr) (n1 + n2))
fibp (Cont (C3)       x ) = x

fib :: Word8 → Word32
fib n = fibp (Call n (store C3))
```

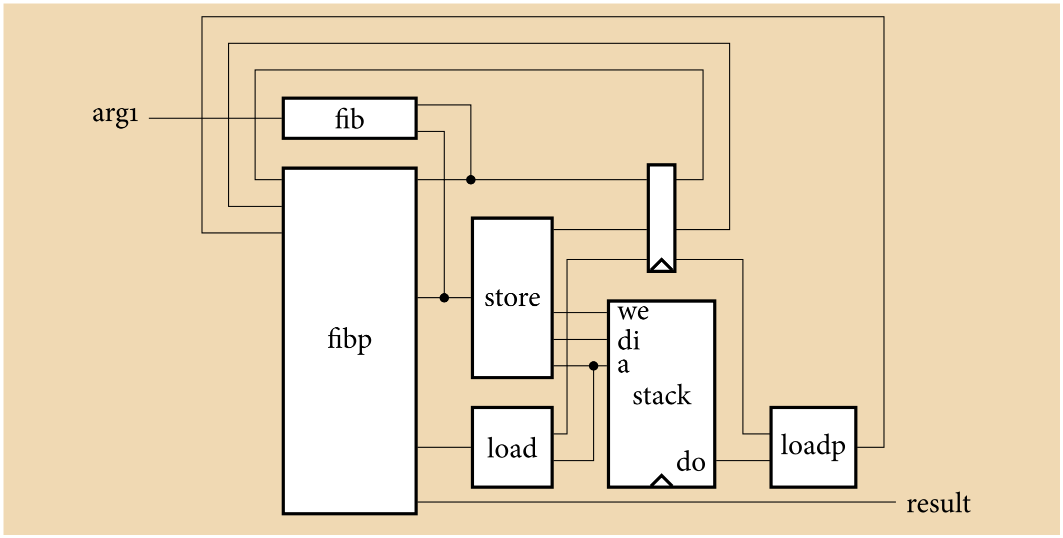
The *fibp* function is tail recursive and its bodies consists of primitive arithmetic functions (e.g., +, −), type constructors, and explicit memory read and write operations whose results are only passed as arguments to (tail-) recursive calls of *fibp*.

In hardware, each invocation of *fibp* will be mapped to a single cycle. Because the FPGAS we consider use synchronous (one-cycle) RAMS, the memory operations work naturally in this setting—the result of a load is only used in the next clock cycle.

Furthermore, although *load* and *store* are called from multiple sites, these sites are mutually exclusive so only a single instance of these functions is needed along with a trivial arbiter to select their arguments.

3 Implementing Fib in Verilog

Here is the block diagram illustrating the various functions and how they communicate. Information flow is generally left to right; the feedback paths to the input of *fipb* are the exceptions. Note that the small circles—where two output signals join—actually represent simple arbiters that select between two possible sources for a signal.



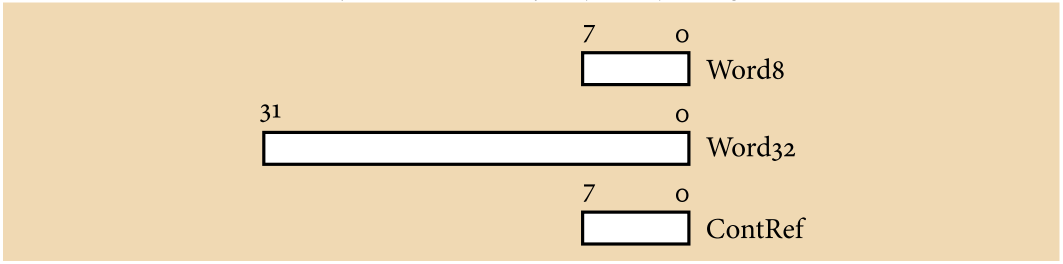
A compulsory timescale definition for the simulator:

```
`timescale 1 ns / 100 ps
```

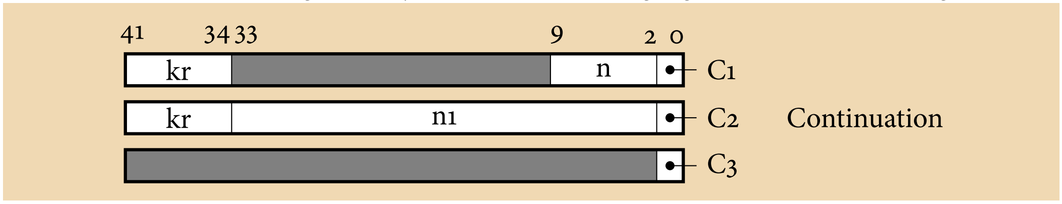
3.1 *Verilog Types for Haskell Types*

Representing eight- and thirty-two-bit unsigned words is straightforward; we use a little-endian style. We arbitrarily represent the *ContPtr* type with 8 bits. Choosing an appropriate numbers of bits for each pointer type (tantamount to bounding the size of memory or the maximum possible number of objects of a particular type) is an interesting problem.

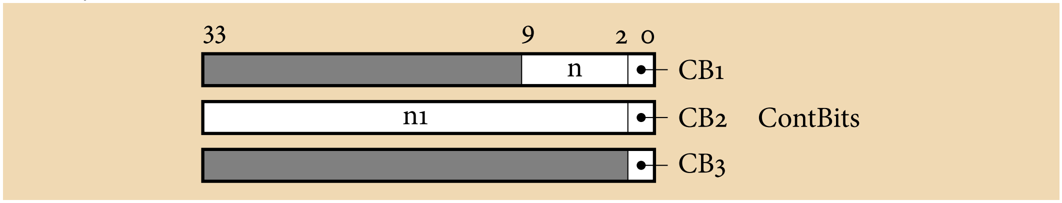
In our Haskell model, the *ContRef* consists of a pointer and the memory into to which it points, but in hardware, we leave the memory implicit because we know every such reference will refer to the same memory. Thus, a *ContRef* object is just eight bits.



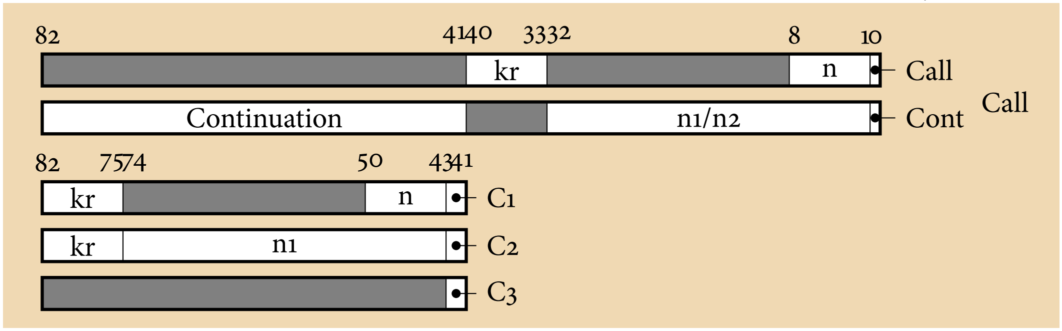
For the *Continuation* type, we use the first few bits to encode the tag and align the *ContRef* fields. I suspect optimizing such layouts is an interesting algorithmic problem in general.



The *ContBits* type—what is stored in the stack memory—is the same after dropping the *ContRef* field:



The *Call* type implements a trick. In this program, the *Continuation* field of a *Cont* is always generated by a call to the *load* function, and the *ContRef* field of a *Call* is always generated by a call to *store*. Because of the nature of memory in the FPGA, *load* is a sequential (single-cycle) function. By assigning these fields to different bits in the *Call* type, it obviates the need to add multiplexers that select between these two sources in the next cycle.



This layout is taking even more advantage of the flexibility of the types. I am not sure whether it is good idea to share the eight bits of the *n* field in the *Call* type with the *n1/n2* field in the *Cont* type. Doing so reduces the number of flip-flops but introduces the need for muxes that decide how to load them. This is a particular kind of state assignment problem that involves a lot of don't-care values and a particular way of looking at states; I've not seen a problem exactly like this in the literature.

3.2 *Constants: Type Tag Fields*

I encode the tag fields of the three main algebraic types in binary. One-hot may be better for some.

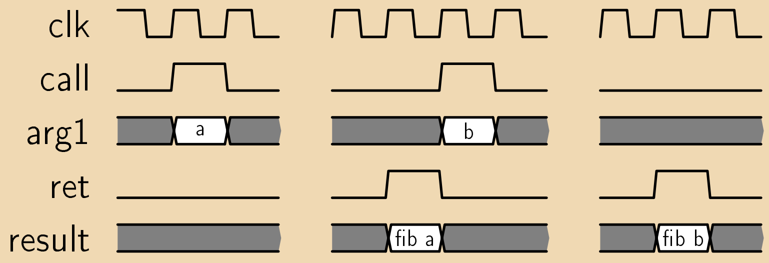
```
`define C1 2'd 0      // Continuation
`define C2 2'd 1
`define C3 2'd 2

`define CB1 2'd 0     // ContBits
`define CB2 2'd 1
`define CB3 2'd 2

`define Call  1'd 0   // Call
`define Cont 1'd 1
```

3.3 *Module Interface*

The main module, which implements the *fib* function, takes a single 8-bit input and produces a 32-bit output. The *call* input indicates the input (*arg1*) is valid (and that the function should start); the *ret* output indicates the result is valid and the function has terminated. This suffices for now, but later we will want a more complicated communication protocol.

```
module fib(input               clk ,
           input               call ,        // strobe
           input        [7:0]  arg1 ,        // Word8
           output reg          ret ,         // strobe
           output reg   [31:0] result        // Word32
           );
```

The bigger question going forward is what, exactly, the environment must guarantee about these signals. E.g., are the input arguments valid only when *call* is high or must they be held? When, if ever, should we use four-phase handshaking?



3.4 *Local Signals*

Each function needs a signal indicating whether it is active in the current cycle and a vector for its arguments. Exceptions include the top-level function *fib*, which is only called from outside. Its activation, argument, and result signals are ports.

```
reg          fibp_call ;
reg  [82:0]  fibp_arg1 ;          // Call

reg          store_call ;
reg  [41:0]  store_arg1 ;         // Continuation
reg  [7:0]   store_result_d ;     // ContRef
reg  [7:0]   store_result ;       // ContRef

reg          load_call ;
reg  [7:0]   load_arg1 ;          // ContRef

reg          loadp_call ;
reg  [7:0]   loadp_arg1 ;         // ContRef
reg  [41:0]  loadp_result ;       // Continuation
```

The *fibp* function calls itself recursively, so it needs a register that indicates it will run in the next cycle and its argument. We introduce *_d* signals to express these.

```
reg          fibp_call_d ;
reg  [32:0]  fibp_arg1_d ;   // Call without output from load, store
reg  [32:0]  fibp_arg1_q ;   // Call without output from load, store
```

Both the *store* and *load* functions need some internal variables.

```
reg  [7:0]    store_p ;        // ContRef
reg  [7:0]    store_pprime ;   // ContRef
reg  [33:0]   store_cprime ;   // ContBits
reg           store_write ;    // Write to continuation memory

reg  [7:0]    load_p;          // ContRef
reg  [7:0]    load_pprime;     // ContRef
reg           loadp_call_d ;
```

3.5 The Continuation Stack Memory

For simplicity, the continuation memory is a simple single-port RAM, although the FPGA supports fancier types. This follows Altera's suggested template for a single-port RAM with old-data-read-during-write behavior.

```
reg  [33:0]   cont_ram  [255:0];    // ContBits: the stack itself

reg   [7:0]   cont_addr;            // ContRef
reg           cont_write_enable ;
reg   [33:0]  cont_write_data ;     // ContBits
reg   [33:0]  cont_read_data ;      // ContBits

always @(posedge clk) begin
  cont_read_data  <= cont_ram[cont_addr];
  if ( cont_write_enable )
    cont_ram[cont_addr] <= cont_write_data ;
end
```

3.6 The State Process

The other state-holding elements are, by design, simple. The strobe and arguments for *fibp* are latched, as is the result from *store* and the argument for *loadp*, since each must cross the clock boundary.

The state-holding elements consist of the machinery for the tail calls and the *load* and *store* functions.

```
always @(posedge clk) begin
    fibp_call      <= fibp_call_d ;
    loadp_call     <= loadp_call_d ;
    fibp_arg1_q    <= fibp_arg1_d ;
    store_result   <= store_result_d ;
    loadp_arg1     <= load_pprime;
end
```

3.7 The fib and fibp functions

The *fib* and *fibp* functions share a resource: the *fibp* function itself, but arbitration between the two is simple since we can assume that two calls never occur simultaneously. As such, both bodies can be synthesized together and we can let the synthesis tool infer the muxes.

To begin with, we reset the outputs from this block. Activation signals are cleared; everything else gets X's.

```
always @(*) begin
  ret   = 0;              // Return a result
  result  = 32'b X;

  fibp_call_d  = 0;
  fibp_arg1_d  = 33'b X;

  load_call  = 0;         // Load continuation from memory
  load_arg1  = 8'b X;

  store_call   = 0;       // Store continuation in memory
  store_arg1   = 42'b X;
```

The argument to *fibp* is assembled from three sources: state-holding flip-flops from *fibp* itself, the output from *store* (the address stored), and the output from *lopd* (the continuation).

```
  fibp_arg1  = { loadp_result ,   store_result  ,  fibp_arg1_q };
```

When *fib* is invoked, *call* is asserted and the argument is presented at the *arg1* port. In this case, *C3* is stored in the memory (to initialize the stack) and the function is invoked in the next cycle via *fibp_call_d*.

```
  if  ( call )  begin
    // fibp (Call n (store C3))
    fibp_call_d  =  1;
    fibp_arg1_d  = {24'b X,  arg1 ,  `Call };
    store_call   =  1;
    store_arg1   = {40'b X,  `C3};
  end
```

The body of *fibp* implements the pattern-matching, arithmetic, and load/store operations defined by the *fibp* function described in §2.6.

Its function is to produce four strobe/value pairs: one for the result, one for a tail call, one for calling *load*, and one for calling *store*.

All the other cases correspond to the patterns in the *fibp* function from §2.6.

```
  if ( fibp_call ) begin
    if ( fibp_arg1 [0] == `Call && fibp_arg1 [8:1] == 8′d 1) begin
      // fibp (Call 1 kr) = fibp (Cont (load kr) 1)
      load_call = 1;
      load_arg1 = fibp_arg1 [40:33];
      fibp_call_d = 1;
      fibp_arg1_d = { 32′d 1, `Cont };
    end else if ( fibp_arg1 [0] == `Call && fibp_arg1 [8:1] == 8′d 2) begin
      // fibp (Call 2 kr) = fibp (Cont (load kr) 1)
      load_call = 1;
      load_arg1 = fibp_arg1 [40:33];
      fibp_call_d = 1;
      fibp_arg1_d = { 32′d 1, `Cont };
    end else if ( fibp_arg1 [0] == `Call) begin
      // fibp (Call n kr) = fibp (Call (n-1) (store (C1 n kr)))
      store_call = 1;
      store_arg1 = { fibp_arg1 [40:33], 24′b X, fibp_arg1 [8:1], `C1 };
      fibp_call_d = 1;
      fibp_arg1_d = { 24′b X, fibp_arg1 [8:1] - 8′d 1, `Call };
    end else if ( fibp_arg1 [0] == `Cont && fibp_arg1 [42:41] == `C1) begin
      // fibp (Cont (C1 n kr) n1) = fibp (Call (n-2) (store (C2 n1 kr)))
      store_call = 1;
      store_arg1 = { fibp_arg1 [82:75], fibp_arg1 [32:1], `C2 };
      fibp_call_d = 1;
      fibp_arg1_d = { 24′b X, fibp_arg1 [50:43] - 8′d 2, `Call };
    end else if ( fibp_arg1 [0] == `Cont && fibp_arg1 [42:41] == `C2) begin
      // fibp (Cont (C2 n1 kr) n2) = fibp (Cont (load kr) (n1 + n2))
      load_call = 1;
      load_arg1 = fibp_arg1 [82:75];
      fibp_call_d = 1;
      fibp_arg1_d = { fibp_arg1 [32:1] + fibp_arg1 [74:43], `Cont };
    end else if ( fibp_arg1 [0] == `Cont && fibp_arg1 [42:41] == `C3) begin
      // fibp (Cont (C3) x ) = x
      ret = 1;
      result = fibp_arg1 [32:1];
    end
  end

end
```

3.8 *The load and store functions*

Because both *load* and *store* access the continuation memory (never simultaneously), it is easiest to combine their two bodies in a single process and have the synthesis tool insert the mux that feeds either the read or the write address to the memory.

```
always @(*) begin
   load_p = 8′b X;
   load_pprime = 8′b X;
   loadp_call_d = 0;

   store_p = 8′b X;
   store_pprime = 8′b X;
   store_cprime = 34′b X;
   store_result_d = 8′b X;

   cont_addr = 8′b X;
   cont_write_enable = 0;
   cont_write_data = 34′b X;
```

The *load* function calculates p' and passes the old address to the continuation memory. There is no explicit "read enable" input; the memory is assumed to read unless written to.

```
   if ( load_call ) begin
      load_p = load_arg1 ;
      load_pprime = load_p − 8′d 1;
      loadp_call_d = 1;
      cont_addr = load_p;
   end
```

The *store* function seems richer; it converts a *Continuation* object into a *ContBits* object and instructs the memory to store it. In practice, the conversion should be just a bit slice.

```
    if ( store_call ) begin
        // case c of
        if ( store_arg1 [1:0] == `C1) begin // C1 n
            store_p = store_arg1 [41:34];
            store_cprime = {24'b X, store_arg1 [9:2], `CB1};
        end else if ( store_arg1 [1:0] == `C2) begin // C2 n1
            store_p = store_arg1 [41:34];
            store_cprime = { store_arg1 [33:2], `CB2};
        end else if ( store_arg1 [1:0] == `C3) begin // C3
            store_p = 8'd 0;
            store_cprime = {32'b X, `CB3};
        end
        store_pprime = store_p + 8'd 1;
        cont_write_enable = 1;
        cont_write_data = store_cprime;
        cont_addr = store_pprime;
        store_result_d = store_pprime;
    end
end
```

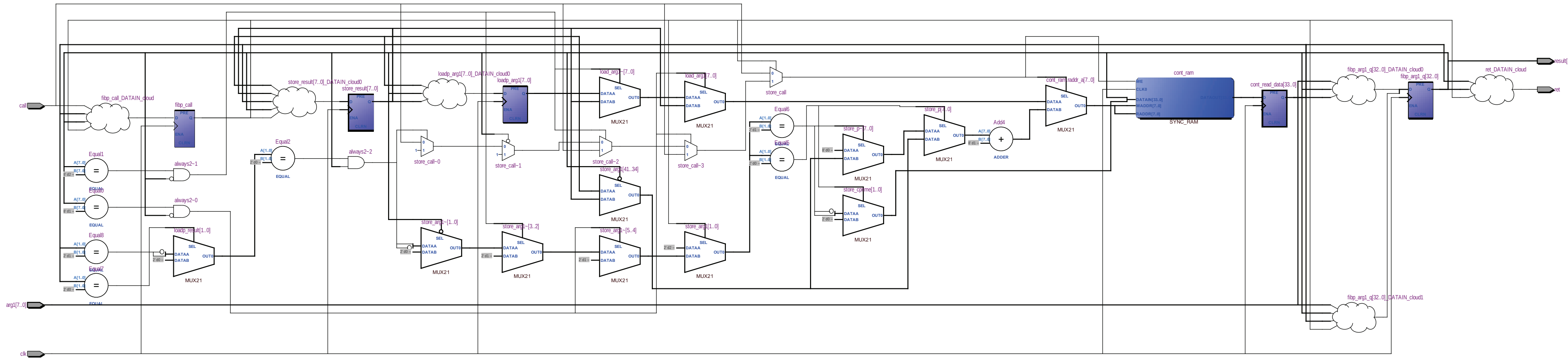
3.9 *The loadp function*

The *loadp* function reconstructs a *Continuation* object from the *ContBits* object read from memory. This should distill down to little more than concatenating bit vectors. The result, in *loadp_result* is passed to the *fibp* function.

```
always @(*) begin
   loadp_result  = 42'b X;
   if ( loadp_call ) begin
     // case d of
     if (cont_read_data [1:0]  == `CB1)
        // CB1 n -> C1 n (p', m)
        loadp_result  = {loadp_arg1 , 24'b X, cont_read_data [9:2],   `C1};
     else if (cont_read_data [1:0]  == `CB2)
        // CB2 n1 -> C2 n1 (p', m)
        loadp_result  = {loadp_arg1 , cont_read_data [33:2],   `C2};
     else if (cont_read_data [1:0]  == `CB3)
        // CB3 -> C3
        loadp_result  = { 40'b X,  `C3 };
   end
end
```

This is the end of the *fib* module.

```
endmodule
```



The schematic, as reported by the RTL Netlist Viewer from Quartus.

4 Verilog Testbench

```
`timescale 1 ns / 100 ps
module top();

    reg clk = 0;
    reg call;
    reg [7:0] arg1;
    wire [31:0] result;
    wire ret;

    always #10 clk = ~clk;

    fib dut(clk, call, arg1, ret, result);

    initial begin
        $dumpfile("fib.vcd");
        $dumpvars(0, dut);
        call = 0;

        @(posedge clk);
        call = 1;
        arg1 = 8'd 1;
        @(posedge clk);
        call = 0;

        @(ret == 1);
        if ( result != 8'd 1) begin
            $display("%d:␣ERROR:␣expected␣1;␣got␣%d", $time, result);
            $finish;
        end

        @(posedge clk);
        call = 1;
        arg1 = 8'd 2;
        @(posedge clk);
        call = 0;

        @(ret == 1);
        if ( result != 8'd 1) begin
            $display("%d:␣ERROR:␣expected␣1;␣got␣%d", $time,  result );
            $finish ;
        end

        @(posedge clk);
        call  = 1;
        arg1  = 8'd 6;
        @(posedge clk);
        call  = 0;

        @(ret == 1);
        if ( result != 8'd 8) begin
            $display("%d:␣ERROR:␣expected␣8;␣got␣%d", $time,  result );
            $finish ;
        end

        @(posedge clk);
        call  = 1;
        arg1  = 8'd 10;
        @(posedge clk);
        call  = 0;

        @(ret == 1);
        if ( result != 8'd 55) begin
            $display("%d:␣ERROR:␣expected␣55;␣got␣%d", $time,  result );
            $finish ;
        end

        $display("%d:␣OK:␣Verilog␣Simulation␣terminated␣ successfully ", $time);
        $finish ;
    end

    initial begin
     #10000;
     $display("%d:␣ERROR:␣terminated␣without␣finding␣result", $time);
     $finish ; // Failsafe
    end

endmodule
```

References

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

[2] Stephen A. Edwards. Functional Fibonacci to a fast FPGA. Technical Report CUCS–010–12, Columbia University, Department of Computer Science, New York, New York, USA, June 2012.

[3] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Proceedings of Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 190–203, Nancy, France, 1985. Springer.

[4] 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.