Semantics-directed Machine Architecture in ReWire

***Abstract*—The functional programming community has developed a number of powerful abstractions for dealing with the full spectrum of programming models. But from the hardware synthesis point of view, functional programming languages come with their own costly baggage: the unrestricted use of recursion and higher-order programming makes hardware synthesis impractical. This paper identifies a subset of the pure functional language Haskell that can be compiled directly to efficient implementations as hardware state machines, and a prototype compiler, called ReWire, which implements this idea. We show how ReWire lets designers leverage ideas from the functional programming world, including monads, to tackle the complexity of hardware design in a modular way, without compromising efficiency.**

I. Introduction

This paper introduces ReWire, a compiler for a subset of the pure, functional language Haskell which produces synthesizable VHDL code, suitable for use on FPGAs, for sequential circuits. As a derivative of Haskell, ReWire inherits Haskell's main advantages: strong static typing, type inference, type safety, type classes to support overloading, higher-order functional programming abstractions such as monads, and the absence of side effects. The combination of these features is a powerful basis for formal verification: properties of the system may be proved by applying simple equational reasoning techniques to a high-level source program, and the compiler produces efficient circuits directly from that program, meaning there is no semantic gap between model and implementation.

Edwards [1] describes the challenges of synthesizing hardware from C. There is a great profusion of hardware design languages (HDL) that take the C language as a starting point for hardware specification and synthesis. The most successful of these C-based languages are, Edwards argues, only superficially related to C—they may retain the surface syntax of C, but they also extend and restrict C in essential ways.

Many of Edwards' critiques may be summarized as follows: C and its descendants are fundamentally built on a fixed programming model which does not conform to the essence of sequential hardware. C's programming model includes flat memories, stacks, loops and recursion—the fundamental building blocks of a procedural programming model. Furthermore, C is single-threaded with no native support for concurrency or timing. A language for targeting hardware, by contrast, requires more flexibility and expressiveness: notions of memory and timing may differ starkly from one design to another. Many software mainstays (e.g., stack-based recursion) may not be readily implemented in hardware. Memory hierarchies are more complex in hardware than in software and, consequently, an HDL needs more fine-grained control in the specification of memory.

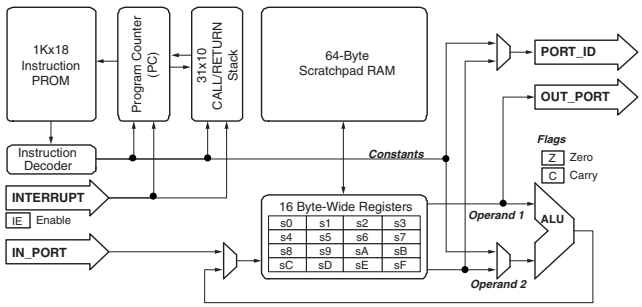
Functional languages allow program properties (e.g., correctness and security) to be verified through *equational reasoning*. This presents a tantalizing prospect for producing verified hardware. Other features of functional programming languages, however, do not map to hardware constructs efficiently or, in some cases, at all. Sequential circuits typically have some kind of interface with the outside world. Input-output interactions are notoriously difficult to handle in a pure functional language—indeed, the absence of side effects is part of what enables the equational reasoning paradigm to begin with. On the other hand, there are ways in which functional languages may be *too* expressive for the hardware domain. How, for example, does one compile arbitrary recursive functions to logic gates?

In order to handle interactions with the outside world, ReWire provides a construct called a *reactive resumption*, originally discovered in the context of concurrency theory [3]. Reactive resumptions are a purely functional representation of reactive systems such as sequential hardware circuits. In ReWire, the use of recursion at runtime is restricted to *guarded*, *tail recursive* functions that return reactive resumptions. Furthermore, reactive resumptions are the only recursive structures that are allowed at runtime. ReWire is a subset of Haskell—i.e., any ReWire program is also a Haskell program. But, the restrictions on recursion mean that, just as in VHDL or Verilog, some valid Haskell constructs are not synthesizable—if any forbidden forms of recursion are present in the program, and those forms cannot be transformed away by the ReWire partial evaluator (described in Section IV), then the result is not synthesizable.

This article proceeds as follows. The remainder of this section describes a case study in which a Xilinx PicoBlaze is specified in ReWire. Section II is an overview of the design of ReWire. Section III describes the structure of the ReWire compiler. Section IV outlines how domain-specific extensions to ReWire may be incorporated in the language using monads. Section V talks about a simple CPU. Section VI presents related work and Section VII concludes and describes future work.

PicoBlaze in ReWire: The best way to learn a new programming language is to focus on an example before approaching the formal syntactic and semantic details. In this section, we present a high level overview of a specification of the Xilinx PicoBlaze embedded microcontroller [2] written in ReWire.

Fig. 1a shows a block diagram of the PicoBlaze and Fig. 1b presents the corresponding ReWire type declarations. The PicoBlaze controller features 16 single byte registers, a 64 byte internal RAM, a 31 byte "stack" for call and return, support for up to 256 input/output devices, and an ALU with carry and zero flags. The register file type, `RegFile`, is declared in ReWire as a `Table W4 W8` (i.e., a mapping from `W4` to `W8`). Generally, for any n, the type `W`n is a built-in type in ReWire which stands for "n-bit word." There are five flags in PicoBlaze, which are, respectively, the zero (`Z`), carry (`C`),



(a) Block Diagram

```
type RegFile  = Table W4 W8
type FlagFile = (Bit,Bit,Bit,Bit,Bit)
type Mem      = Table W6 W8
data Stack    = Stack { contents :: Table W5 W10,
                        pos :: W5 }
data Inputs   = Inputs { instruction_in :: W18,
                         in_port_in     :: W8,
                         interrupt_in   :: Bit,
                         reset_in       :: Bit }
data Outputs  = Outputs { address_out       :: W10,
                          port_id_out       :: W8,
                          write_strobe_out  :: Bit,
                          out_port_out      :: W8,
                          read_strobe_out   :: Bit,
                          interrupt_ack_out :: Bit }
```

(b) Corresponding ReWire Types

Fig. 1: Xilinx PicoBlaze Microcontroller ([2], page 8) Readily Represented in ReWire.

zero-save (`Zsave`), carry-save (`Csave`), and interrupt enable (`IE`) flags. The type declaration for the flag registers is `FlagFile` in Fig. 1b. The scratchpad RAM is represented as a table, `Table W6 W8`.

In ReWire (as in Haskell [4]), a *type synonym* is a new name for an existing type. Each of the aforementioned types is declared as a type synonym—i.e., with the `type` form. The right-hand sides of the aforementioned declarations involve only built-in types (i.e., word, tuple and table types) and so they are declared with `type`. As in Haskell, to introduce new data types, ReWire has `data` declarations. The stack, input and output types are defined with `data` declarations using record syntax. Record types have the form, $\{\mathtt{x}_1 :: \mathtt{t}_1, \cdots, \mathtt{x}_n :: \mathtt{t}_n\}$, where each $\mathtt{x}_i$ is a field name of type $\mathtt{t}_i$. Records are more convenient than tuple types when there are a large number of fields. The `PicoState` type, declared below, encapsulates all components of the current state of the processor:

```
data PicoState
    = PicoState { reg_file :: RegFile, flags :: FlagFile,
                  memory   :: Mem, stack  :: Stack,
                  outputs :: Outputs, inputs :: Inputs }
```

We can now define the `PicoBlaze` monad. The best way to understand what this means is to see how it is used (we will have more to say about it in subsequent sections).

```
type PicoBlaze = ReT Outputs Inputs (StT PicoState I)
```

The `PicoBlaze` monad defines a new domain-specific language that allows us to write the program describing the PicoBlaze processor. Rather than delving into the details of monads, it is simpler to understand how a familiar idea is represented with it. Below, the fetch-decode-execute loop for PicoBlaze (called "`fde`") is written in ReWire:

```
fde :: PicoBlaze ()
fde = do s <- getPicoState
         let i     = inputs s
             instr = instruction_in i
         ie <- getFlagIE
         if reset_in i == 1
            then reset_event
            else if ie == 1 && interrupt_in i == 1
                    then interrupt_event
                    else decode instr
         fde
```

The `fde` computation first gets the current state of the processor with `getPicoState` and assigns it to `s`. The inputs on the input ports are bound to `i` and `instr` is bound to the instruction word. The current value of the interrupt enable flag is read and assigned to `ie`. If the reset signal has been set (i.e., `reset_in i == 1`), then the processor transitions to the `reset_event` state (not shown). Otherwise, if the interrupt flag is set and an interrupt has occurred, then the processor makes the transition to the `interrupt_event` (not shown). Otherwise, the processor decodes and executes the instruction. Finally, `fde` starts its loop again.

Fig. 2 shows the ReWire code corresponding to the PicoBlaze add immediate instruction. The definition of `addImm` ends with two calls to `tick`. The `tick` operation delimits single cycles. Each PicoBlaze instruction takes two cycles. In `addImm`, the first cycle (i.e., the operations up to the first `tick`) contains all of the instruction's action, while the second cycle—the second `tick`—does nothing but wait for a single cycle [2].

II. ReWire Core

Our prototype compiler is structured around a core language, which is a subset of Haskell, called ReWire Core. Fig. 3 illustrates the structure of a ReWire Core program via an example near and dear to functional programmers' hearts: the Fibonacci sequence $F = (0, 1, 1, 2, 3, 5, \dots)$, where $F_n = F_{n-1} + F_{n-2}$ for all $n > 1$. The construction of this simple example illustrates how a mathematical structure from concurrency theory called a *reactive resumption*, combined with some standard functional programming techniques, forms the basis of sequential circuit specifications in ReWire. Later, in Section III, we will use the same example to demonstrate how ReWire produces efficient implementations of reactive resumption-based specifications as hardware state machines.

As a subset of Haskell, ReWire Core places two primary restrictions on the form of programs. First, every program must be defined in terms of a set of equations, which may be mutually recursive, producing a reactive resumption. Second, the form of recursion and the use of higher-order constructs (functions operating on functions, or functions operating on resumptions) is restricted. In Section IV, we demonstrate how a source-to-source program transformation called *partial evaluation* [5] can be used to enhance the expressiveness

```
decode :: Instruction -> CPU ()
decode "011000<x:____><k:________>" = addImm x k
...
addImm :: Register -> Register -> CPU ()
addImm sX kk = do v <- getReg sX
                  c <- getFlag FlagC
                  let (c',v') = (v `addWithCarry` kk) c
                  putFlag FlagZ (toBit $ v' == 0)
                  putFlag FlagC c'
                  putReg sX v'
                  incrPC
                  tick
                  tick
```

Fig. 2: ReWire Code for PicoBlaze Add Immediate Instruction.

of ReWire by making conservative extensions to the core language, which can automatically be translated back into the core language.

Reactive Resumptions: The design of ReWire starts from the observation that computation in sequential hardware is *clocked*, meaning that it produces an *output* and takes an *input* at each discrete tick of a clock; and that it may have *memory*, in the sense that the behavior of the circuit at time *t* may depend on events that have taken place before *t*. To represent this sort of computation, ReWire provides a type called a *reactive resumption*. The type `React o i a`—read "the type of a reactive resumption computation with output type *o*, input type *i*, returning a value of type *a*"—may be represented in Haskell according to the following type declaration.

```
data React o i a =
    D a
  | P o (i -> React o i a)
```

In English, this means that a value of the type `React o i a` is either a "done thing" (tagged with `D`), in which case it carries a value of type `a`; or a "paused thing" (tagged with `P`), in which case it carries an output of type `o`, and a function that takes an input of type `i` and produces a new computation of type `React o i a`. The *recursive* nature of this definition—`React o i a` is defined in terms of itself—is essential. Constructing a `P`-value requires both an output value and a *function* that produces from an input value another resumption representing (in informal terms) the "rest" of the computation.

Restrictions on Recursion: In the software realm, recursion is a pervasive tool for functional programming. Recursion in its full generality, however, is difficult to implement in hardware, and doing so may even require features that are actually undesirable, such as a stack or a heap. More limited forms of recursion such as *tail recursion*, however, are much more amenable to implementation in hardware. A function is *tail recursive* if the last operation it performs is calling itself. Tail recursive procedures can be transformed into loops using *tail call elimination* [6]. ReWire Core restricts recursive specifications as follows: (1) recursion is only allowed in functions producing a result of type `React`; (2) only tail recursion is allowed; (3) all recursive functions must be *productive*, meaning they eventually produce a `P` or `D`; (4) recursive functions may only have arguments of simple types (i.e. no recursive data structures such as trees). In summary, every ReWire Core program is a set of productive, mutually tail recursive equations with result types in `React`.

```
 1 module Fibonacci where

 2 type Output = W8
 3 type Input  = (Bit,Bit)   -- (reset,hold)
 4 type Re a   = React Output Input a

 5 start :: Re ()
 6 start = P 0 (\ (r,h) ->
 7   case r of
 8     0 -> loop 0 1
 9     1 -> start)

10 loop :: W8 -> W8 -> Re ()
11 loop cur nxt = P cur (\ (r,h) ->
12   case r of
13     0 -> case h of
14            0 -> loop nxt (cur+nxt)
15            1 -> loop cur nxt
16     1 -> start)
```

Fig. 3: Fibonacci Sequence in ReWire Core

Example: The machine of Fig. 3 flashes the elements of the Fibonacci sequence in binary on the (8-bit) output line, beginning from $F_0 = 0$ and moving to the next element of the sequence at each clock tick. In order to make things slightly more interesting, we also supply two one-bit input lines—a reset signal and a "hold" signal—to the machine. The initial machine state, called `start`, sends `0` on the output line and transitions back to `start` when the reset signal is active, and otherwise to state `loop` with arguments of `0` for the `cur` parameter and `1` for the `nxt` parameter. The other machine state, called `loop`, takes the `cur` and `nxt` parameters supplied by the previous state, signals `cur` on the output lines, and transitions either to `start` (when the reset line is active), to `loop` with `cur` and `nxt` values unchanged (when the reset line is inactive and the hold line is active), or to `loop` with the current value of `nxt` moved into the `cur` argument and `cur+nxt` supplied for `nxt` (when both reset and hold are inactive).

At this point, the reader may notice that the PicoBlaze specification of Section I uses language features that lie outside of ReWire Core. For example, the calls to `interrupt_event` and `decode` are not proper tail calls; furthermore, the `do` notation which allows for the manipulation of mutable registers is not part of ReWire Core. The mechanism by which these extended language constructs are handled is detailed later in Section IV. In the meantime, the following section outlines how ReWire Core programs can be implemented as hardware state machines, via a prototype compiler producing synthesizable VHDL suitable for use on FPGAs.

III. Generating VHDL from ReWire Core

The pipeline used to generate synthesizable VHDL consists of three phases. The first and second phases involve the generation and simplification of code in a simple intermediate representation called the State Machine Intermediate Language (SMIL). Translating ReWire Core into SMIL allows us to perform a number of optimizations that would be difficult to implement at the source (ReWire Core) level, but without having to implement them against VHDL in all of its complexity. Once simplification is completed, the final compiler phase produces VHDL code from simplified SMIL.

```
out width 8; in width 2; init out 0; init state start

state start ()
  when signals[0] == '0', out 0, goto loop (0,1);
  when signals[1] == '1', out 0, goto start;
state loop (cur[8], nxt[8])
  when signals[0] == '0', out 0, goto start;
  when signals[1] == '0', out cur, goto loop(nxt,cur+nxt);
  when signals[1] == '1', out cur, goto loop(cur,nxt);
```

Fig. 4: `Fibonacci`: Intermediate SMIL Code

A. SMIL Generation and Simplification

SMIL is a simple compiler-intermediate language whose programs are explicitly structured as finite state machines. In addition to named control states, each state is accompanied with a set of data variables (e.g. `cur` and `nxt` associated with state `loop` in Fig. 4), whose values are supplied by the previous state when a transition is made. These variables may be used to implement registerized state. SMIL is a simply typed language; the only types in the language are sized bit vectors. When translating ReWire Core to SMIL, a simple correspondence holds: those functions that are supplied as arguments to `P` correspond to states in the SMIL machine.

After the initial SMIL code is generated, a few rounds of simplification are performed, applying such standard techniques as common subexpression elimination [6]. The post-simplification SMIL code for this example is given in Figure 4.

B. VHDL Generation

The final phase of the compiler pipeline generates VHDL from the simplified SMIL code. The VHDL generator produces a typical two-process implementation of the state machine (Fig. 5): one process handles the updating of the state register at every rising clock edge, and the other process contains all the combinational logic needed to produce the next-state value. The only persistent signal in the resulting design is `state_reg`, which contains (1) a tag indicating whether the machine is "done" or "paused"; (2) a register for the current output signals; (3) a tag of width $\lceil \log_2(\textit{number of control states}) \rceil$ indicating the current control state; and (4) space for the data variables in the control states (here, just `cur` and `nxt` from `cs1`—if more than one control state has data variables, we only need to make enough space for the largest case). Here our decision only to allow tail calls in the source language has paid off; rather than implementing the data passing from state to state as a stack, we can simply allocate a single global data variable, and overwrite it with a new value at state transition time.

Evaluation: Simulation and Synthesis: Having verified the circuit's functionality in simulation (Fig. 6), we used Xilinx's XST synthesis tool to implement this circuit on a Xilinx Spartan-3E XC3S500E FPGA, speed grade -4. The maximum clock frequency for the inferred circuit is 201.005 MHz, which is reasonably fast for this chip. Size and timing statistics for the resulting implementation are given in the table below. As one might expect, the circuit uses only a tiny fraction of the logic elements on the board.

```
-- Signals declared in architecture header.
signal state_reg, state_reg_initial,
  state_reg_next, state_reg_next_cs0, state_reg_next_cs1
       : STD_LOGIC_VECTOR(0 to 25) := (others => '0');

sm_process: process(sm_inputs,clock,state_reg_next) is
begin
  if rising_edge(clock) then
    state_reg <= state_reg_next; end if;
end process sm_process;

state_reg_initial <= machine_reset;

sm_outputs <= state_reg(1 to 8);

with state_reg(9) select state_reg_next <=
 state_reg_next_cs0 when '0',
 state_reg_next_cs1 when '1',
 state_reg_initial  when others;

state_reg_next_cs0 <=
 machine_tick("00000000", "00000001")
    when sm_inputs(0) = '0' else
 machine_reset
    when sm_inputs(1) = '1' else
 state_reg_initial;

state_reg_next_ll_cs1 <=
 machine_reset when sm_inputs(0) = '1' else
 machine_tick(state_reg(10 to 17), state_reg(18 to 25))
    when sm_inputs(1) = '1' else
 machine_tick(state_reg(18 to 25),
              PRIM_W8_Plus(state_reg(10 to 17),
                           state_reg(18 to 25)))
    when sm_inputs(1) = '0' else
 state_reg_initial;
```

Fig. 5: `Fibonacci`: Final VHDL Output

	Used	Available	Utilization
Slices	15	4656	0.32%
Slice Flip Flops	18	9312	0.19%
4-Input LUTs	55	9312	0.59%
Bonded IOBs	11	232	4.74%
GCLKs	1	24	4.17%

IV. Conservative Extensions to ReWire

The core language of Section II represents a subset of Haskell that can easily be compiled to efficient hardware state machines. It is easy to see, however, that for some applications the restrictions of ReWire Core are too limiting. Consider, for example, a design such as a CPU that has a large amount of persistent state such as registers and flags. In ReWire Core, one would have to "thread the state through" explicitly, passing pre- and post-values from one state to the next. From an abstraction point of view, we would be much better off if we could somehow extend this core language to support stateful programming, in essence extending the language with constructs particular to our circuit's intended structure. There is a common design pattern called a *monad* [7] that allows us to do exactly this.

A. Monads

Monads are the principal means by which impure programming models (i.e., stateful, programming with exceptions, concurrency, non-determinism, etc.) are expressed within pure languages like Haskell. A monad is simply a type constructor, `M`, combined with two operations:

```
return :: a -> M a
(>>=)  :: M a -> (a -> M b) -> M b
```

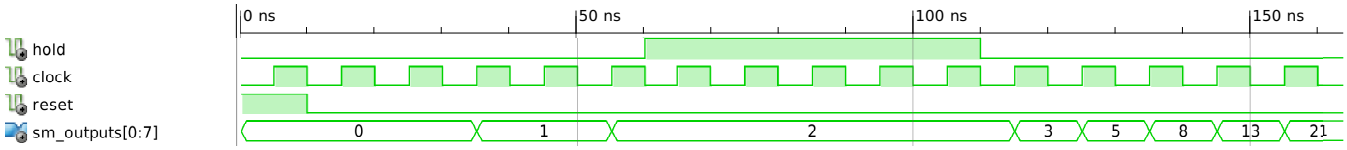


Fig. 6: `Fibonacci`: Waveform from the ISim Simulator

These operations must obey the "monad laws"—i.e., that `>>=` is associative and `return` is its left and right unit. We use monads as a means of conservatively extending ReWire Core. Memory architectures are expressed as state monads, particularly those created by multiple applications of the state monad transformer discussed below.

Reactive Resumptions are Monads: One example of a monad, as it happens, are reactive resumptions themselves. In particular, for any output and input types `o` and `i`, `React o i` is a monad, with its bind and return operations defined as:

```
(>>=) :: React o i a -> (a -> React o i b) -> React o i b
D v   >>= f = f v
P o k >>= f = P o (\ i -> k i >>= f)

return :: a -> React o i a
return v = D v
```

While all monads have bind and return operations, it is also very common to define additional operations that are appropriate to the particular monad. For `React`, we define an operation called `signal` which has the effect of writing a value to the circuit's output, waiting for the next clock tick, and returning the input at the following clock tick.

```
signal :: o -> React o i i
signal o = P o (\ i -> return i)
```

With these building blocks—still defined completely in terms of the primitive notion of a resumption—we can rewrite our Fibonacci machine in a more "imperative" style as follows, using a special syntax for monads called `do`-notation provided by ReWire (and Haskell). Note that here we have also economized on code size by defining a custom helper function `resetIf` that takes a bit and a resumption computation as arguments, and performs the computation if the bit is zero, but transitions to the reset state in other cases.

```
 1 module Fibonacci where

 2 type Output = W8
 3 type Input  = (Bit,Bit)     -- (rst,hold)
 4 type Re a   = React Output Input a

 5 resetIf :: Bit -> Re () -> Re ()
 6 resetIf rst m = case rst of
 7                   1 -> start
 8                   0 -> m

 9 start :: Re ()
10 start = do (rst,hold) <- signal 0
11            resetIf rst (tick 0 1)

12 tick :: W8 -> W8 -> Re ()
13 tick cur nxt = do (rst,hold) <- signal cur
14                   resetIf rst (
15                     case hold of
16                       1 -> tick cur nxt
17                       0 -> tick nxt (cur+nxt))
```

Monad Transformers: Now say we wanted to add some additional functionality—e.g., a register—to our design. It is not obvious how to accomplish this with `React` as we have described it thus far. But there is a standard tool for adding such functionality known as *monad transformers*, by which monads can be extended with new operations in a canonical fashion [7].

Treating `React` as a monad lets us use monad transformers. A particularly useful transformer is the state monad transformer `StT`, which takes an existing monad `m` and adds an updateable state `s` to it. If `m` is a monad, then `StT s m` is a state monad with state of type `s`. The definition of that state monad transformer threads state through the existing monad. Note that the monad operations, `>>=` and `return`, are overloaded. The state monad transformer also adds two new operations, update (`u`) and get (`g`), as well as the lifting operation, `liftST`, which is used to redefine any existing `m`-operations. See the appendix for full definitions.

```
u :: (s -> s) -> StT s m ()
g :: StT s m s
liftST :: m a -> StT s m a
```

In Section V, a fetch-decode-execute cycle is defined for a monad with a register file state, `RegF`. This state is added to `React` using the state monad transformer, `StT RegF`.

B. Partial Evaluation

Partial evaluation [5] is a source-to-source program transformation that has been studied within the programming languages community since the 1980's. It has usually been studied in the context of functional languages. Partial evaluation might also be called static evaluation because it transforms a source program into a simplified version by performing as much of the computation statically as is possible.

The canonical example of partial evaluation involves the power function, expressed in Haskell as:

```
power :: Int -> Int -> Int
power 0 x = 1
power n x = x * (power (n-1) x)
```

Let us say that one knew the value of the first parameter was `2`. The expression `power 2` could be simplified via the following derivation:

```
power 2 x = x * (power (2-1) x)      = x * (power 1 x)
          = x * x * (power (1-1) x)  = x * x * power 0 x
          = x * x * 1                = x * x
```

It should be noted that this specialized version of `power`, `power 2`, is no longer recursive. A partial evaluator performs just this kind of evaluation/reduction/simplification automatically. ReWire uses a partial evaluator to rewrite non-Core constructs to expressions in Core when possible. It is frequently possible

to do so, which substantially increases the utility of the ReWire framework. In particular, partial evaluation is what makes our conservative extensions conservative.

V. Case Study: Simple CPU

This case study concerns the description in ReWire of a simple CPU including its instruction set and fetch-decode-execute cycle. The ReWire code in its entirety is listed in the Appendix. The CPU has an 8 bit address space and its register set contains two 8 bit general purpose registers, `r1` and `r2`, and an 8 bit program counter, `pc`. It has two input ports, the first of which is a single bit wide reset line to (re)start the CPU. The second port is a connection to a 16 bit wide data bus. Likewise, the CPU has two output ports connecting it to a single bit line controlling an LED as well as to a 16 bit address line.

The remainder of this section describes the ReWire program for the CPU in its entirety. The program is small. Discounting comments and blank lines, it contains on the order of 70 lines of code, roughly half of which are type signatures for functions included only for readability's sake. The heart of the ReWire program for the CPU is an interpreter for its instruction set—this interpreter is precisely the execute phase of the fetch-decode-execute cycle.

The abstract syntax for the instruction set is:

```
data Instr = Branch0 W8    -- branch if r0=0
           | LoadR1 W8     -- load r1 from offset
           | LoadR2 W8     -- load r2 from offset
           | Add           -- r1 := r1+r2
           | SetLED        -- led := lsb[r1]
           | Invalid       -- invalid instruction
```

The instructions include instructions to branch, load the `r1` and `r2` registers, and a simple add. There is also an instruction to assign the least significant bit of the `r1` register to the LED output port—assigning 1 (resp., 0) to the port turns on (resp., off) the LED. Finally, there is an `Invalid` instruction because not all 16 bit words correspond to valid instructions. The effect of executing `Invalid` will be to return to the reset state.

The following are types underlying the architecture:

```
type InpSig = (Bit,       -- external reset line
               W16)       -- input word from bus
type OutSig = (Bit,       -- LED off/on
               Addr)      -- output address to bus
type Addr   = W8
type RegF   = (W8,W8,W8) -- (r1,r2,pc)
```

`InpSig` is the type of the input ports and `OutSig` is the type of the output ports. Addresses (`Addr`) are 8-bits wide and the register file (`RegF`) contains three registers as described above.

The CPU monad is a composition of three monad transformers. The first is a reactive component (`Re`), encapsulating the i/o of signals from the external environment. The second and third components add the raw material for manipulating the register file (`StT RegF`) and the current signals (`StT CPUState`).

```
type Re       = React OutSig InpSig
type U        = StT RegF Re
type CPU      = StT CPUState U
type CPUState = OutSig
```

Instructions on the CPU are defined in terms of the following operations. The operation, `put rf`, sets the current register file to register file `rf` and `get` reads the current register file. The operations `putOutSig` and `getOutSig` are similar, affecting the output signals. These are defined in terms of the state monad transformer operations `u` and `g`.

```
put       :: RegF -> CPU ()
get       :: CPU RegF
putOutSig :: OutSig -> CPU ()
getOutSig :: CPU OutSig
```

With `put` and `get`, we define commands for setting and reading the individual `RegF` components; those for `r1` and `r2` are similar to `putPC` and `getPC`, so we omit them.

```
putPC :: W16 -> CPU ()
putPC pc = get >>= \ (r1,r2,_) -> put (r1,r2,pc)
getPC :: CPU W16
getPC    = get >>= \ (_,_,pc) -> return pc
```

With `putOutSig` and `getOutSig`, we define commands for setting and reading the output signals. The LED is cleared and set via `putLED`. Similar commands for reading from and writing to the address bus are omitted:

```
putLED :: Bit -> CPU ()
putLED led  = getOutSig >>= \ (_,a) -> putOutSig (led,a)
```

The decode phase of the fetch-decode-execute cycle is simply a function that splits a `W16` into two bytes, an opcode byte and an operand byte (used only for branches and loads). Due to the small size of the instruction set, there are a number of wasted bits in this encoding.

```
decode :: W16 -> Instr
decode w = case split16 w of
               (0x80,byt) -> Branch0 byt
               (0x81,byt) -> LoadR1 byt
               (0x82,byt) -> LoadR2 byt
               (0x83,_)   -> Add
               (0x84,_)   -> SetLED
               (_,_)      -> Invalid
split16 :: W16 -> (W8,W8)
split16 (W16 b0 b1 b2 b3 b4 b5 b6 b7
             b8 b9 ba bb bc bd be bf) = (byt1,byt2)
   where byt1 = W8 b0 b1 b2 b3 b4 b5 b6 b7
         byt2 = W8 b8 b9 ba bb bc bd be bf
```

The `CPU` monad includes the current signals, and `onSig` is a convenience function that applies a function to the input signals and returns the result in `CPU`. The `sigCPU` function returns the current input signals.

Below are the start and fetch states. The `start` state is the initial state in the FSM computation underlying the CPU's operation. It sets the initial `pc` register and LED to `0`, writes `(0,0)` to the output port, and then transitions to `fetch`.

```
start   :: CPU ()
start       = do putPC 0
                 putLED 0
                 putOutSig (0,0)
                 fetch
fetch   :: CPU ()
fetch       = do pc    <- getPC
                 putAddr pc
                 putPC (pc+1)
                 (r,w) <- sigCPU
                 resetIf r (exec (decode w))
```

The function `resetIf` is a helper function that returns the simple CPU to the `reset` state if the test is false (`0`).

The execution phase for the simple CPU is found in Fig. 7.

The CPU described here has been synthesized for the same target FPGA as the Fibonacci machine of Section III. The

```
exec :: Instr -> CPU ()
exec (Branch0 pc') = getPC >>= \ pc -> putPC (pad pc') >> sigCPU >>= \ (rst,_) -> resetIf rst fetch
exec i@(LoadR1 a)  = putAddr (pad a) >> sigCPU >>= \ (rst,_) -> resetIf rst $ exec_2 i
exec i@(LoadR2 a)  = putAddr (pad a) >> sigCPU >>= \ (rst,_) -> resetIf rst $ exec_2 i
exec Add           = getR1 >>= \ r1 -> getR2 >>= \ r2 -> putR1 (r1+r2) >> sigCPU >>= \ (r,_) -> resetIf r fetch
exec SetLED        = getR1 >>= \ r1 -> putLED (lsb r1) >> sigCPU >>= \ (r,_) -> resetIf r fetch
exec Invalid       = start
exec_2 :: Instr -> CPU ()
exec_2 (LoadR1 _)  = sigCPU >>= \ (r,w) -> putR1 (trim w) >> resetIf r fetch
exec_2 (LoadR2 _)  = sigCPU >>= \ (r,w) -> putR2 (trim w) >> resetIf r fetch
lsb :: W8 -> Bit
lsb (W8 _ _ _ _ _ _ _ b7)             = b7
```

Fig. 7: The Execution Phase of the Simple CPU.

maximum clock rate is 133.515 MHz, and the device usage is characterized by the following table.

	Used	Available	Utilization
Slices	115	4656	2.47%
Slice Flip Flops	48	9312	0.52%
4-Input LUTs	213	9312	2.29%
Bonded IOBs	27	232	11.64%
GCLKs	1	24	4.17%

A. Simulation in Haskell

Because ReWire is a proper subset of the Haskell programming language, ReWire programs may also be executed using the Glasgow Haskell Compiler (GHC). To simulate the simple CPU, the function call `(simulate start n [] [])` steps through `n` steps of the cpu, recording each `(InpSig,OutSig)` pair in a list. An example execution follows:

```
GHC> simulate start 5 [] []
[ ((0x0,0x8180),(0x0,0x0)) ,
  ((0x0,0xFFFF),(0x0,0x80)),
  ((0x0,0xFFFF),(0x0,0x80)),
  ((0x0,0x8400),(0x0,0x1)) ,
  ((0x0,0x8400),(0x1,0x1))    ]
```

VI. Related Work

From the point of view of programming language taxonomy, ReWire falls into the category of functional languages, with a strong emphasis on monadic programming. Other approaches to synthesis grounded in declarative programming include Raabe et al.'s "sketches" [8], and a case study in synthesizing an MPEG-4 decoder from a data flow program [9].

Within the functional programming world, Lava [10], [11] is a domain-specific language for hardware specification embedded in Haskell. While the particulars of the Lava idea vary by implementation, Lava is in essence a method to specify circuit behavior at the level of signals, meaning it operates on roughly the same level of abstraction as VHDL or Verilog. ReWire, by contrast, attempts to compile a subset of Haskell itself to hardware circuits, and relies on a somewhat more abstract set of primitives (namely reactive resumptions). Cλash [12] is both a compiler for a subset of Haskell to VHDL. Like ReWire, Cλash uses Haskell itself as a source language rather than embedding a domain-specific language in Haskell. Cλash requires some limits be placed on kinds of algebraic data types used as well as the basic operating types. ForSyDe is a platform to compile models of hardware written in Haskell to circuitry [13]. ForSyDe operates similarly to Lava in that it uses a Haskell as a host for an embedded domain-specific language, with circuits represented by types provided by the ForSyDe platform.

A number of works [14], [15], [16], [17], [18] have tackled the problem of recursion in hardware synthesis. Skliarova and Sklyarov provide an overview of this line of work [19]. The approach taken by ReWire is to view non-tail recursion as a language extension, which can only be synthesized in cases where partial evaluation manages to eliminate it or turn it into tail recursion.

VII. Conclusions and Future Work

Edwards [1] has commented on the difficulty of compiling from a C like language to hardware. This has led him to pursue Haskell as a source [20]. The case study in this paper suggests that synthesizing sequential circuits from purely functional languages is indeed feasible. Furthermore, the use of higher-order functional abstractions, such as monads, greatly speeds the construction of complex circuits, and makes their specifications much more extensible.

One possible avenue of future work is to adapt the ReWire compiler to enable programs that mix CPU and FPGA-based computation. A mixed-mode compiler would take the non-synthesizable portions of the program and compile them for use on a CPU-based system containing an FPGA, with the two parts of the program communicating over the PCI-E bus. In combination with existing work [21] on coordinating resumptions with software schedulers, this may present a powerful framework for hybrid computing in a high level language.

Further optimization to the generated state machines should be possible. In particular, partial evaluation may result in an unnecessary propagation of control states—if the same function is inlined in two places, a duplicated control state may result. The layout of the data-variable register may also be a target for optimization. As it is currently implemented, data variables are simply given space within the state register in order from left to right as they occur in the program text. This sometimes results in unduly complex next-state logic as unchanged values must sometimes be moved over to make room for new state elements at state transition time. We expect that standard compiler techniques such as liveness analysis [6] will be fruitful here.

Recent research has demonstrated the value of monadic semantics to the formal specification and verification of x86-

and ARM-based systems [22], [23]. The besetting challenge for all such formal methods research is that the systems under consideration are formalized *post facto* to their construction and consequently the formal methods scientist must engage in a painstaking reconstruction of the system semantics from informal and sometimes incomplete natural language documents. The formal methods effort must face a difficult dilemma. Should the systems be formalized *post facto* in all their complex glory or should some simplified abstract model be pursued instead?

With ReWire, the text of the design is verified as-is and the compiler transforms that same design into hardware, thereby unifying the languages of specification, design and implementation. This may alleviate the necessity of reconstructing system semantics as they design and formal specification are one and the same.

Recent work by the several of the authors applies effects systems to fault isolation for kernels written in a resumption monad-based language akin to ReWire [21]. Kernels that type check in this system isolate faults, meaning that a fault occuring in one kernel thread domain do not interfere with other domains. Information flow security properties have also been formulated as effect systems [24]. As part of an ongoing project investigating security in the setting of many core computers, the authors are extending and adapting their fault isolation effect system to ReWire to enforce security properties. ReWire, being a strongly typed, functional language, may provide a vector for adapting a range of language-based security techniques [25] to hardware.

REFERENCES

[1] S. A. Edwards, "The challenges of synthesizing hardware from C-like languages," *IEEE Design & Test*, vol. 23, no. 5, pp. 375–386, 2006.

[2] *PicoBlaze 8-bit Embedded Microcontroller User Guide*. Xilinx, Inc., 2011. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/ug129.pdf

[3] J. de Bakker and E. de Vink, *Control Flow Semantics*. MIT, 1996.

[4] S. Peyton Jones, Ed., *Haskell 98 Language and Libraries, the Revised Report*. Cambridge University Press, 2003.

[5] N. D. Jones, C. K. Gomard, and P. Sestoft, *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[6] S. S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[7] S. Liang, P. Hudak, and M. Jones, "Monad transformers and modular interpreters," in *POPL'95*, pp. 333–343.

[8] A. Raabe and R. Bodik, "Synthesizing hardware from sketches," in *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*, 2009, pp. 623–624.

[9] J. Janneck, I. Miller, D. Parlour, G. Roquier, M. Wipliez, and M. Raulet, "Synthesizing hardware from dataflow programs: An mpeg-4 simple profile decoder case study," in *Signal Processing Systems, 2008. SiPS 2008. IEEE Workshop on*, 2008, pp. 287–292.

[10] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, "Lava: Hardware design in Haskell," in *3rd ICFP*, 1998, pp. 174–184.

[11] A. Gill, T. Bull, G. Kimmell, E. Perrins, E. Komp, and B. Werling, "Introducing Kansas Lava," in *Proc. of the Symp. on Imp. and App. of Func. Lang.*, ser. LNCS, vol. 6041, Sep 2009.

[12] M. E. T. Gerards, C. P. R. Baaij, J. Kuper, and M. Kooijman, "Higher-order abstraction in hardware descriptions with Cλash," in *Proc. of the 14th EUROMICRO Conf. on Dig. Sys. Des.*, 2011, pp. 495–502.

[13] I. Sander and A. Jantsch, "System modeling and transformational design refinement in ForSyDe," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 1, pp. 17–32, 2004.

[14] G. Ferizis and H. Gindy, "Mapping recursive functions to reconfigurable hardware," in *Field Prog. Logic and App.*, 2006, pp. 1–6.

[15] T. Maruyama and T. Hoshino, "A c to hdl compiler for pipeline processing on fpgas," in *Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on*, 2000, pp. 101–110.

[16] S. Ninos and A. Dollas, "Modeling recursion data structures for fpga-based implementation," in *Field Prog. Log. and App.*, 2008, pp. 11–16.

[17] D. Mihhailov, V. Sklyarov, I. Skliarova, and A. Sudnitson, "Hardware implementation of recursive algorithms," in *53rd IEEE Int. Midwest Symp. on Circ. and Sys.*, 2010, pp. 225–228.

[18] G. Stitt and J. Villarreal, "Recursion flattening," in *Proc. 18th ACM Great Lakes Symp. on VLSI*, 2008.

[19] I. Skliarova and V. Sklyarov, "Recursion in reconfigurable computing: A survey of implementation approaches," in *Field Prog. Logic and App., 2009.*, 2009, pp. 224–229.

[20] S. A. Edwards, "A finer functional fibonacci on a fast FPGA," Columbia University, Dept. of Comp. Sci., Tech. Rep. CUCS-005-13, February 2013.

[21] W. L. Harrison, A. Procter, and G. Allwein, "The confinement problem in the presence of faults," in *Proc. 14th Int. Conf. on Formal Eng. Methods (ICFEM)*, 2012, pp. 182–197.

[22] S. Sarkar, P. Sewell, F. Z. Nardelli, S. Owens, T. Ridge, T. Braibant, M. O. Myreen, and J. Alglave, "The semantics of x86-CC multiprocessor machine code," in *36th ACM POPL*, 2009, pp. 379–391.

[23] A. Fox and M. O. Myreen, "A trustworthy monadic formalization of the ARMv7 instruction set architecture," in *Proc. 1st Int. Conf. on Inter. Thm. Prov.*, 2010, pp. 243–258.

[24] L. Bauer, J. Ligatti, and D. Walker, "Types and effects for non-interfering program monitors," in *Software Security—Theories and Systems.*, ser. LNCS, vol. 2609, 2003, pp. 154–171.

[25] A. Sabelfeld and A. Myers, "Language-based information-flow security," *IEEE Journ. on Sel. Areas in Commun.*, vol. 21, no. 1, Jan. 2003.

APPENDIX

A. Functional Programming in Haskell

This section provides a quick primer on functional programming in Haskell. A program in a functional language is nothing more than a function in the ordinary mathematical sense—that is, a mapping from inputs to outputs.

A Haskell function is written in much the same way that one might in high school algebra; the function `inc` that takes an integer and adds one to it is:

```
inc :: Int -> Int -- type signature
inc n = n + 1
```

The type signature for `inc` declares that it has type `Int -> Int`. That is, `inc` is a function that takes an `Int` input to an `Int` output (the double colon "`::`" is pronounced "has type"). An *anonymous function* or *λ-abstraction* is a way of defining a function without giving it a name. The λ-abstraction, `\ n -> n + 1`, is literally the same function as `inc` except that it is nameless.

Haskell possesses flexible and powerful means for defining and programming with new data types. We could define a type of security modes for a CPU with support for a security kernel:

```
data Mode = Hi | Lo
```

The new type, `Mode`, possesses two constructors, `Hi` and `Lo`. The Haskell type system automatically enforces the proper use of newly defined data types—i.e., the only way of creating a value of type `Mode` is with its constructors. Any value of type `Mode` will have one of the prescribed forms, `Hi` or `Lo`.

Haskell has an expression resembling C's `switch` statements called a "case" expression. If `e :: Mode`, then the following is a case expression that checks the form of `e` and, evaluates `e1` (`e2`) if it is `Hi` (`Lo`):

```
case e of
     Hi -> e1
     Lo -> e2
```

Haskell is higher-order, which means that functions are first-class values (just like `Int` or `Char`). Functional programs can be composed using ordinary function composition. The composition of two functions `f` and `g`, is defined in Haskell: `(f . g) x = f (g x)`). Higher-order also means that partial application of a function is also possible; `inc'` below is a perfectly legal and equivalent means of defining increment by partial application of `add`:

```
add :: Int -> Int -> Int
add i j = j + i

inc' :: Int -> Int
inc' = add 1
```

Consider the following function `fib` which takes a non-negative integer and returns the corresponding element of the Fibonacci sequence (i.e., $0, 1, \ldots, f_{i-2}, f_{i-1}, f_{i-2}+f_{i-1}, \ldots$). This can be written as a recursive functions `fib` or `trfib`:

```
fib :: Int -> Int
fib 0   = 0
fib 1   = 1
fib n   = fib (n-2) + fib (n-1)
trfib n = fibacc (0,1) n          -- tail recursive
  where
    fibacc (a,b) 0 = a
    fibacc (a,b) n  = fibacc (b,a+b) (n-1)
```

Implementing `fib` as-is requires a potentially unbounded stack while a tail recursive version does not. Many compilers would attempt to transform `fib` into `trfib` to enable tail-call elimination [6] to be performed. Tail-call elimination replaces a recursive function with a loop. In the context of ReWire, tail-recursion allows us put a finite upper bound on the memory required by a function.

B. Source Code for Simple CPU

```
type InpSig = (Bit,        -- external reset signal
               W16)        -- input word from address bus
type OutSig = (Bit,        -- LED off/on
               Addr)       -- output address to bus
type Addr   = W8
type RegF   = (W8,W8,W16) -- (r1,r2,pc)

type Re  = ReT OutSig InpSig I
type U   = StT RegF Re
type CPU = StT CPUState U
type CPUState = OutSig

put :: RegF -> CPU ()
put rf = liftStT $ u (const rf)

get :: CPU RegF
get = liftStT $ g

putPC :: W16 -> CPU ()
putPC pc = get >>= \ (r1,r2,_) -> put (r1,r2,pc)

putR1 :: W8 -> CPU ()
putR1 r1 = get >>= \ (_,r2,pc) -> put (r1,r2,pc)

putR2 :: W8 -> CPU ()
putR2 r2 = get >>= \ (r1,_,pc) -> put (r1,r2,pc)

getPC :: CPU W16
getPC = get >>= \ (_,_,pc) -> return pc

getR1 :: CPU W8
getR1 = get >>= \ (r1,_,_) -> return r1

getR2 :: CPU W8
getR2 = get >>= \ (_,r2,_) -> return r2

getOutSig :: CPU OutSig
getOutSig = g

putOutSig :: OutSig -> CPU ()
putOutSig sigs = u (const sigs)

getAddr :: CPU Addr
getAddr = getOutSig >>= return . snd

putAddr :: W16 -> CPU ()
putAddr a = getOutSig >>= \ (l,_) -> putOutSig (l,a)

putLED :: Bit -> CPU ()
putLED l = getOutSig >>= \ (_,a) -> putOutSig (l,a)

-- Decoded instructions.
data Instr = Branch0 W8   -- branch to absolute target if r0
     =0
           | LoadR1 W8    -- load r1 from address
           | LoadR2 W8    -- load r2 from address
           | Add          -- r1 := r1+r2
           | SetLED       -- my led := lsb[r1]
           | Invalid      -- invalid instruction
           deriving Show

split16 :: W16 -> (W8,W8)
split16 (W16 b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 ba bb bc bd be bf
     ) = (byt1,byt2)
   where byt1 = W8 b0 b1 b2 b3 b4 b5 b6 b7
         byt2 = W8 b8 b9 ba bb bc bd be bf

decode :: W16 -> Instr
decode w = case split16 w of
               (0x80,byt) -> Branch0 byt
               (0x81,byt) -> LoadR1 byt
               (0x82,byt) -> LoadR2 byt
```

```
               (0x83,_)   -> Add
               (0x84,_)   -> SetLED
               (_,_)      -> Invalid

liftRe :: Re a -> CPU a
liftRe = liftStT . liftStT

resetIf :: Bit -> CPU () -> CPU ()
resetIf 0 r = r
resetIf 1 _ = machine_reset

sigCPU :: CPU InpSig
sigCPU = onSig id
   where onSig :: (InpSig -> a) -> CPU a
         onSig k = getOutSig >>= \ req ->
                   liftRe (stepRe req (I . k))

skip = sigCPU >> return ()

start :: CPU ()
start = do putPC 0
           putLED 0
           putOutSig (0,0)
           fetch

fetch :: CPU ()
fetch = do pc    <- getPC
           putAddr pc
           putPC (pc+1)
           (r,w) <- sigCPU
           resetIf r (exec (decode w))


exec :: Instr -> CPU ()
exec (Branch0 pc') = do pc    <- getPC
                        putPC pc'
                        (r,_) <- sigCPU
                        resetIf r fetch

exec i@(LoadR1 a)  = do putAddr a
                        (r,_) <- sigCPU
                        resetIf r (exec_2 i)

exec i@(LoadR2 a)  = do putAddr a
                        (r,_) <- sigCPU
                        resetIf r (exec_2 i)

exec Add           = do r1 <- getR1
                        r2 <- getR2
                        putR1 (r1+r2)
                        (r,_) <- sigCPU
                        resetIf r fetch

exec SetLED        = do r1    <- getR1
                        putLED (lsb r1)
                        (r,_) <- sigCPU
                        resetIf r fetch
            where lsb :: W8 -> Bit
                  lsb (W8 _ _ _ _ _ _ _ b) = b

exec Invalid       = do skip
                        reset

trim (W16 _ _ _ _ _ _ _ _ b8 b9 ba bb bc bd be bf) = byte
   where byte = W8 b8 b9 ba bb bc bd be bf
exec_2 :: Instr -> CPU ()
exec_2 (LoadR1 _) = do (r,w) <- sigCPU
                       putR1 (trim w) >>
                       resetIf r fetch

exec_2 (LoadR2 _) = do (r,w) <- sigCPU
                       putR2 (trim w)
                       resetIf r fetch

go = deST (deST machine_reset cpu0) regf0
     where cpu0 = (0,0)
           regf0 = (0,0,0)

--simulation code for simple cpu example
simulate (D v)   _ is os = zip os is
simulate _ 0 is os       = zip os is
simulate (P q k) i is os = do
                           let (_,addr) = q
                           simulate (deId $ k (0,mem addr))
                                    (i-1) (is ++ [q]) (os ++
                                   [(0,mem addr)])

where mem a = fromMaybe 0 $ lookup a tbl
      tbl = [(0x0000,0x8180),    -- r1 := mem[0x80]
             (0x0001,0x8400),    -- setled
             (0x0002,0x8181),    -- r1 := mem[0x81]
             (0x0003,0x8400),    -- setled
             (0x0004,0x8000),
             (0x0080,0xFFFF),
             (0x0081,0x0000)]
```