MetaRTL: Raising the Abstraction Level of RTL Design

Jianwen Zhu

Technical Report ECE-00-xx
May 2000
(Version as of March 19, 2001)

Electrical and Computer Engineering
University of Toronto
Toronto, ON M5S 3G4, Canada

jzhu@eecg.toronto.edu

Abstract

The register transfer abstraction (RTL) has been established as the industrial standard for ASIC design, soft IP exchange and the backend interface for chip design at higher level. Unfortunately, the "synthesizable" VHDL/Verilog incarnation of the RTL abstraction has problems which prevent it from more productive use. For example, the confusion as the result of using simulation semantics for synthesis purpose, the lack of facility for component reuse at the "protocol" level, and the lack of memory abstraction. After a detailed discussion of these problems, this paper proposes a new RTL abstraction, called MetaRTL, which can be implemented by a modest extension to the traditional imperative programming languages. The productivity gain is further demonstrated by the description of a synthesis tool, called MetaSyn, which provides the "added-value". Experiments on the benchmark set show that MetaRTL is far more concise than the "synthesizable" HDL specification, and incurs no overhead for synthesis result.

Contents

1 Introduction

Due to their complexity, VLSI designs are performed at different levels of abstraction. Among them register transfer level (RTL) is one of the most important. For ASIC design, the dominant practicing methodology starts at RTL. For the booming intellectual property (IP) market, RTL is becoming the *de facto* soft IP exchange standard. Furthermore, RTL serves as the "assembly language", or the backend interface, for the higher level (for example, the behavioral level) design methodology.

In theory, the RTL design can be formalized as Gajski's FSMD model [4], which is an extension of the FSM model with the so-called register transfer operations, each of which can be considered as an assignment of value, computed as an expression over a set of register values, to another register. The FSM model can be best visualized by the ASM chart, invented by IBM in the 1960s.

In practice, a specification language is needed to capture the RTL design. Typically, an RTL language is used to specify the FSMD model as well as some additional information. Some information is considered essential: for example, the mapping between expression operators and the actual hardware components. Some information is considered only syntactical sugar: for example, constructs to facilitate modular design. The importance of syntactical sugar, however, cannot be underestimated, since it is designed to combat design complexity, which becomes increasingly important when one moves to systems-on-chip design.

The dominant RTL specification languages in use today are VHDL and Verilog, the IEEE standard hardware description languages (HDLs). Unfortunately, the current RTL design methodology based-on HDLs is not without problems. Some of the fundamental problems are listed as follows:

- HDLs are designed as a simulation language. The gap between the simulation semantics and the synthesis semantics causes unnecessary confusion.
- HDLs are not designed with design reuse in mind.
- HDLs are not designed with a type system as powerful as that of software languages.
- HDLs do not provide any abstractions for memories.
- HDLs do not simulate fast enough at the RTL level.

In this paper, we first discuss related work in the literature. We then extend the discussion on the implications of the above-mentioned issues in Section 3. We then propose in Section 4 a new RTL abstraction, called MetaRTL, which extends the FSMD with a rich set of constructs addressing the identified issues. In Section 5, we present a synthesis tool, which translates MetaRTL into the industrial standard "synthesizable" HDLs. To demonstrate the added-value of MetaRTL over HDLs, we show their difference with a number of benchmarks.

2 Related Work

A number of efforts have emerged recently to use software programming languages to model register transfer level hardware. For example, CynApps announced its Cynlib [3], a C++ class library which provides features so that C++ can be used to model hardware. The Open SystemC Initiative, announced a similar library called SystemC [8] [10]. Another implementation with arguably superior simulation performance is the OCAPI library [9] developed by IMEC. While the expressive power of the software languages can be leveraged to some extent, the goal of these approaches is to repeat HDL semantics in C syntax, and the problems enumerated in this paper remain unsolved.

The V++ synchronous language [2] developed at Cadence Berkeley Lab, as well as its predecessors, such as Esterel [1] and Lustre [6], also try to employ a synthesis semantics (synchronous reactive model) rather simulation semantics. However, none of them is designed with a strong type system, and no memory abstraction is supported.

The SpecC system level design language [5], supports protocol level component reuse the same way as that is proposed in this paper, although it lacks the polymorphic type system desired. With its powerful type systems, the OpenJ language [11] has provided a language framework to experiment with system level design languages, although it did not explicitly define an RTL abstraction. Nevertheless, both languages seem to be suitable frameworks for MetaRTL to apply.

3 Problems with "Synthesizable" HDLs

3.1 Simulation Semantics

VHDL and Verilog (HDLs in the text follows) were designed as simulation languages for gate-level hardware systems. To emulate the behavior of hardware, a HDL programmer write a program which specifies a discrete event system (simulation semantics), rather than how hardware is constructed (synthesis semantics). While many constructs in HDLs can be conveniently mapped to hardware, the discrete event semantics introduces artifacts which are hard, or even impossible to map to hardware (synthesizable). For example, delay is a concept that can neither be interpreted as certain hardware nor certain design constraints. Signals imply potentially infinite size of memory to hold values.

Given that, the industry has devised the so-called "synthesizable" subsets of HDLs, where problematic constructs or problematic uses of certain constructs are excluded. Still, one has to devise a discrete event system to simulate the hardware one has in mind, only to let the EDA tools to discover, or "infer" that hardware later. This added level of indirection is not only unintuitive but also error-prone.

```
...
if( a = '0' ) then
    c <= b;
end if;
...
```

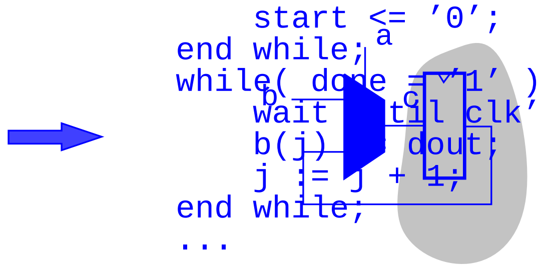

Figure 1. Problem with simulation semantics.

Example 1 *Unwanted latch inferring. A wire or register cannot simply be declared in HDLs, instead, they have to be inferred through the use of signals. In order for an output signal to be interpreted as a wire, assignment has to be performed in all branches of a process. A latch will be inferred otherwise, as is shown in Figure 1. It is not uncommon for beginners to forget the signal assignment for certain don't-care conditions, which results in unwanted latches.*

3.2 Design Reuse

HDLs provide support for design reuse only at a low level through component instantiation. In order to reuse a component, one has to instantiate the component by mapping the ports of the component to corresponding wires. While this procedure is good enough for the reuse of combinational components, the reuse of sequential components and more complex IP cores is more complex. In these components, certain protocols, are predefined to communicate with the components. Typically, such protocol contains states and should be specified as an FSMD by itself. Lacking mechanism to specify component protocol in HDLs, one has to consult the data sheet of component and spent considerate amount of time to design the component interface. And every time the component is replaced by another component with similar functionality during design exploration, the interface circuitry has to be redesigned.

```
u1 : Comp( start, done, din, dout );
...
...
start = '1';
for in in 0 to 8 do
    wait until clk'event and clk = '1';
    din <= a(i);
end for;
while( done = '0' ) do
    wait until clk'event and clk = '1';
    start <= '0';
end while;
while( done = '1' ) do
    wait until clk'event and clk = '1';
    b(j) := dout;
    j := j + 1;
end while;
...
```

Figure 2. Problem with design reuse.

Example 2 *IP reuse. As shown in Figure 2, an IP component needs to be reused in a design. Since using the component involves a complex protocol with handshaking before feeding the input data and obtaining the output data cycle by cycle, the HDL designer has to design interface circuit conforming to this protocol specified in the data sheet, in addition to the instantiation of the component. This tedious process is unnecessary.*

3.3 Type System

While HDLs may have a fairly strong type system (e.g., VHDL), their synthesis subset, can only be considered as an untyped system: all values are bits or bit vectors. This is in contrast with most software languages, which contain a rich set of basic data types as well as mechanism to define abstract data types. Without a strong type system in HDLs, one has to rely on human effort for type checking and type conversion, a task only practiced at the stone age of programming.

Polymorphism in HDLs are only partially supported by generic values. An RTL design can hence be parameterized with values: for example, bitwidth of data and addresses. It is impossible, however, to parameterize an RTL design over the components it may use. This restriction limits the granularity of IP offering, especially for those who offer system level IPs.

3.4 Memory Abstraction

It is fair to state that any interesting application will involve the use of memories. For example, in signal processing applications, memories are used extensively to store data samples. In networking applications, memories are used to buffer data packets as well as maintain protocol states and routing tables.

Despite its importance, there is no memory abstraction in "synthesizable" HDLs. This is in contrast to traditional programming languages, where abstract data types as well as pointers are extensively used to layout and access memory.

```
...
struct {
    int    field1;
    struct {
        char field3;
        } field2;
    } *p1;
short    a, *p2;
char     b[10];

...
a = 0;
b[3] = 'a';
p1->field1 = 1;
p1->field2.field3 = 2;
p2 = &a;
...
```

Figure 3. Problem with memory abstraction.

Example 3 *Memory abstraction in C. Consider the C code segments in Figure 3, where memories can be accessed via variables, arrays and pointers. None of these programming abstractions exist in synthesizable HDLs.*

4 MetaRTL: a New RTL Abstraction

While RTL design has been widely regarded as a "solved" problem, we reconsider the very first question one should always ask, based on the observations made in Section 3: Given the role of RTL design in the entire VLSI design methodology, what exactly should the RTL abstraction abstract away and what it should not.

A revisit to the FSMD model suggests that the RTL abstraction is in fact conceptually "closer" to the traditional programming language based on the imperative semantics than the HDLs based on the discrete-event semantics. After all, both FSMD and imperative semantics represent state machines, and the only fundamental difference between them is that states in FSMD implies timing: state change is synchronized with an outstanding clock; while state in imperative semantics only indicates order. The other helpful abstractions that people have developed for imperative languages can be and should be safely borrowed.

```
  MetaRTL  ::=  (Class)*
    Class  ::=  class ID [[ Formal (, Formal)* ] ]
                { (Field | Method)* }
     Type  ::=  ID [[ Actual | (, Actual)* ] ]
   Formal  ::=  class ID | Type ID
   Actual  ::=  Type | Expr
    Field  ::=  sclass Type ID [ = Expr ];
   sclass  ::=  in | out | inout | reg | wire | latch | Type
   Method  ::=  [ always | public] Type ID
                [ ( Param (, Param)* ] ) { (Stmt)* }
    Param  ::=  [ in | out | inout ] Type ID
     Stmt  ::=  [LeftValue = ] Expr ;
             |  if ( Expr ) Stmt [else Stmt]
             |  switch ( Expr ) (CaseStmt)+
             |  [ID] : Stmt
             |  do Stmt while( Expr ) ;
             |  while (Expr) Stmt
             |  break ;
             |  return Expr ;
 CaseStmt  ::=  case Expr : Stmt | default : Stmt
     Expr  ::=  Literal
             |  this
             |  LeftValue
             |  Expr . ID ( [ Expr (, Expr)* ] )
LeftValue  ::=  ID | Type . ID
             |  Expr . ID
```

Figure 4. MetaRTL syntax.

In Figure 4, we show the syntax of a "new" RTL abstraction that we proposed. The abstraction is presented in the form of language, but as we show later, the basic concepts can be used to extend existing languages. We give it a name, MetaRTL, for its multilingual purpose.

MetaRTL is new in the sense that it differs significantly from the HDL-based RTL abstraction in use today. It is not really that "new" in the sense that it is in essence a syntactic-sugar-free, polymorphic, object-oriented language.

More specifically, the basic unit of design encapsulation in MetaRTL is called a type, specified by the class construct. A type represents either a set of data values, called the value type, or a set of objects, called the object type. A type contains a set of fields and a set of methods. A method contains a sequence of statements, each of which consists of expressions. A type and a class can be used interchangeable except when a type is parameterized, in which case the class is the "template", and the type is an instance of the template. A class can be parameterized over other types and constants.

```
class Alu1 {
  in int      i1, i2;
  in bits[1]  opcode;
  out         o;

  ...
  public int abs( int a ) {
    i1 = a; opcode = 0; return o;
    }
  public int min( int a, int b ) {
    i1 = a; i2 = b; opcode = 1; return o;
    }
  }

```

Figure 5. A combinational component.

Nevertheless, MetaRTL differs from a traditional programming language in the following ways:

- A MetaRTL object type can specify a set of hardware objects. Each object represents a piece of digital synchronous hardware.
- A field of value type in MetaRTL object type can be prefixed with a "storage class" modifier. The `in, out, inout, wire, reg` modifiers indicate that the corresponding field designates an input port, an output port, an inout port, a wire and a register respectively. All other modifiers suggest the kind of memory the corresponding field should be mapped to.
- A field of a hardware object type in MetaRTL instantiate the piece of digital hardware represented by the corresponding object type.
- A method in MetaRTL object type can be prefixed with the `always` modifier, indicating that the method specifies a piece of hardware belonging to that object. Alternatively, it can be prefixed with the `public` modifier, indicating that the method specifies the piece of interface hardware to communicate with the object in order for certain functions to be performed.
- While the syntax is exactly the same as their software counterpart, statements in MetaRTL are not an abstraction of the instructions sequences executed on processors, instead, they specify a synchronous state machine. Section 4.1 gives a more detailed description of the hardware semantics.

In the sequel, we show how MetaRTL addresses the issues that HDL-based RTL abstraction failed to address using a set of examples, which leads to the design of the square root approximation unit (SRA). The SRA unit computes $\sqrt{in1^2 + in2^2}$, as detailed in [4].

Figure 5 and Figure 6 show two combinational components. Figure 7 shows a polymorphic constant shifter, where the constant can be specified as a parameter. Figure 8 shows the sequential SRA component.

```
class Alu2 {
  in int      i1, i2;
  in bits[2]  opcode;
  out         o;

  ...
  public int abs( int a ) {
    i1 = a; opcode = 0; return o;
    }
  public int min( int a, int b ) {
    i1 = a; i2 = b; opcode = 1; return o;
    }
  public int add( int a, int b ) {
    i1 = a; i2 = b; opcode = 2; return o;
    }
  public int sub( int a, int b ) {
    i1 = a; i2 = b; opcode = 3; return o;
    }
  }

```

Figure 6. Another combinational component.

4.1 Synthesis Semantics

In MetaRTL, the hardware semantics for each construct is exactly defined. Each object type specifies a hardware design unit. Fields in MetaRTL mean exactly what they are declared for: the fields with `in, out, inout` modifiers imply ports of the design unit; the fields with `wire` modifier imply wires; while fields with `reg` modifier imply registers. Other fields are variables whose addresses will be automatically allocated by the compiler.

The logic contained in the hardware unit is completely and only defined in the `always` methods. In general, statements in a method imply a synchronous state machine, where the labels and loop boundaries indicate state boundaries. For example, the ":"s in Line 66–72 indicates state boundaries, even though the label names are implicit. When no such boundaries exist, the method represents a combinational circuit. For example, in method `main` at Line 39, neither explicit labels nor loops are present, which indicates that `main` represents a combination circuit. Accesses and assignments to wires, ports and registers imply connections instead of the conventional value assignment.

```
class CnstShift[int op2] {                                35
  in int       i;                                         36
  out int      o;                                         37
                                                          38
  always void main() { o = i << op2; }                    39
  public int  shift( int a ) {                            40
    i = a; return o;                                      41
   }                                                      42
  }                                                       43
                                                          44
```

Figure 7. A polymorphic component.

```
class Sra {                                                        45
  in bit          start = 0;                                       46
  out bit          done = 0;                                       47
  in int          in1;                                             48
  in int          in2;                                             49
  out int          dout;                                           50
                                                                   51
  reg int          R1, R2, R2;                                     52
                                                                   53
  Alu1             u1;                                             54
  Alu2             u2;                                             55
  CnstShift[1]     u3;                                             56
  CnstShift[3]     u4;                                             57
                                                                   58
  always void output() {                                           59
    dout = R1;                                                     60
    }                                                              61
                                                                   62
  always void ctrl() {                                             63
    while( start == 0 );                                           64
    R1 = in1; R2 = in2;                                            65
:    R1 = u1.abs( R1 ); R2 = u2.abs( R2 );                         66
:    R1 = u1.max( R1, R2 ); R2 = u2.min( R1, R2 );                 67
:    R2 = u3.shift(R1); R3 = u4.shift( R2 );                       68
:    R2 = u2.sub( R1, R2 );                                        69
:    R2 = u2.add( R3, R2 );                                        70
:    R1 = u1.max( R2, R1 );                                        71
:    done = 1;                                                     72
    }                                                              73
                                                                   74
  public int sra( int a, int b ) {                                 75
    start = 1; in1 = a; in2 = b;                                   76
    while( done == 0 ) {                                           77
      in1 = a; in2 = b;                                            78
      }                                                            79
    return dout;                                                   80
    }                                                              81
  }                                                                82
                                                                   83
```

Figure 8. A sequential component.

4.2 Type System

The type system of MetaRTL resembles that of a modern software programming language. This brings several advantages over HDLs. First, Arithmetic data types as well as others can be used in place of the HDL bit vector data types. Even though their hardware semantics are the same and hence brings no improvement for synthesis quality, the type system can exclude a number of design errors at compile time. In addition, the tedious work of type conversion and promotion can be assumed by the compiler. Second, since MetaRTL has a polymorphic type system, a design unit can both have constants and other data types as generic parameters. The latter adds another dimension of parameterizability over HDLs. Third, although not defined in Figure 4, subtyping can be easily added to bring the same benefit as it does to software.

4.3 Design Reuse

While the powerful type system of MetaRTL certainly improves reusability, another feature of MetaRTL is the protocol method. Indicated by the `public` keyword, protocol methods encapsulate interfacing mechanism to the design unit. For example, at Line 7, the protocol method `abs` specifies how to interface with an `Alu1` unit to perform the `abs` function (compute absolute value): one should connect the operand `a` to the input port `i1`, and connect the input `opcode` to constant 0, and get output at the output port `o`. As a more complex example, the `sra` at Line 75 shows how to interface an SRA unit to perform the square root computation. Note that this method specifies a protocol which has to be implemented as an FSMD.

With protocol methods, the user of a component can simply make appropriate method calls to achieve the desired operations. This tremendously reduces the effort of using a component, in other words, increases the reusability of the component.

4.4 Memory Abstraction

MetaRTL allows the use of memory variables and pointers (object references). Designers can access memory variable by their names instead of their explicit addresses, as in the case of HDLs. The synthesis tool not only performs memory bank and memory address allocation for these variables, but also transforms each access to the memory (load and store) into appropriate calls to predefined memory component protocol methods. Surprisingly, while this abstraction of memory is nothing but a restoring of what software compilers have been doing since the beginning, it greatly improves the productivity of designing memory intensive applications.

5 Experimental Result

We have embedded the concepts defined in MetaRTL into both a C-based and a Java-based research SLDL. We have also developed a tool, called MetaSyn, which synthesizes the SLDLs into synthesizable VHDL.

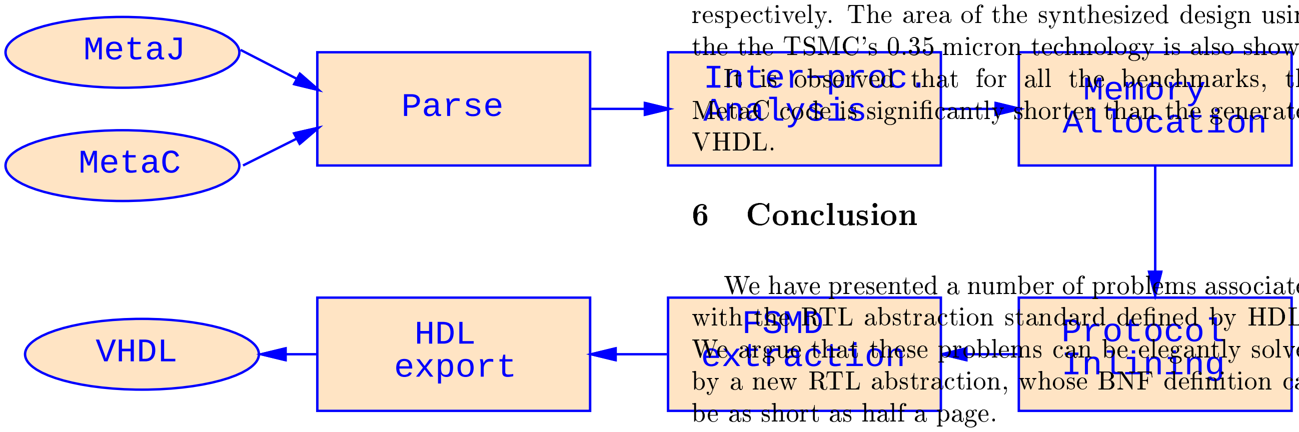

Figure 9. MetaSyn block diagram.

As illustrated in Figure 9, MetaSyn performs a number of tasks. It first parses the source code into an intermediate representation. It then performs a number of analysis tasks, which are referred to as "interprocedural" because they require computation across class boundaries. One example of such analysis is pointer analysis, which computes the storage class a pointer value may point to. Global memory allocation is in turn performed to assign addresses to variables. It will then perform protocol inlining where all the calls to protocols are recursively expanded into the caller. Next, MetaSyn extracts the FSMD model from the intermediate representation and export it into the VHDL format that is consistent with the industry's synthesizable standard.

Benchmark	MetaC (#lines)	VHDL (#lines)	Area (um^2)
fft	45	1028	500652
fir	29	355	272206
iir	46	1201	476529
latnrm	37	748	408111
lmsfir	39	796	405976
mmult	28	451	331058
smult	12	94	57157
sra	16	288	98599

Table 1. Experimental result.

We have tested MetaSyn with a number of benchmarks, most of which are taken from Lee and Chow's DSP benchmark set [7]. Each benchmark is synthesized into gate level implementation using a commercial logic synthesis tool from the VHDL code produced by MetaSyn. Table 1 shows the number of lines of the benchmarks in MetaC and the generated VHDL respectively. The area of the synthesized design using the the TSMC's 0.35 micron technology is also shown.

It is observed that for all the benchmarks, the MetaC code is significantly shorter than the generated VHDL.

6 Conclusion

We have presented a number of problems associated with the RTL abstraction standard defined by HDLs. We argue that these problems can be elegantly solved by a new RTL abstraction, whose BNF definition can be as short as half a page.

7 Acknowledgment

The author would like to thank Mr. Varodayan David Prakash for his help in the experiment.

8 References

[1] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19:87–152, 1992.

[2] S. Cheng, P. McGeer, M. Meyer, T. Truman, A. Sangiovanni-Vincentelli, and P. Scaglia. The V++ system design language. In *Proceedings of the Design Automation and Test Conference in Europe*, 1998.

[3] *CynLib Web Site.* `http://www.cynapps.com/CynApps/products/cynlib/opensource.html`.

[4] D. Gajski. *Principle of Digital Design.* Prentice Hall, Upper Saddle River, NJ, 1997.

[5] D. Gajski, J. Zhu, D. Doemer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology.* Kluwer Academic Publishers, Boston, March 2000.

[6] C. Halbwachs et al. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1992.

[7] C. Lee and P. Chow. *http://www.eecg.toronto.edu/~corinna.*

[8] S. Liao, S. Tjiang, and R. Gupta. An efficient implementation of reactivity for modeling hardware in the Scenic design environment. In *Proceeding of the 34th Design Automation Conference*, 1997.

[9] P. Schaumont, R. Cmar, S. Vernalde, M. Engels, and I. Bolsens. Hardware reuse at the behavioral level. In *Proceeding of the 36th Design Automation Conference*, New Orleans, June 1999.

[10] *SystemC Web Site.* `http://www.systemc.org`.

[11] J. Zhu and D. G. Gajski. OpenJ: A system level design language. In *Proceedings of the Design Automation and Test Conference in Europe*, Munich, Germany, March 1999.