Journal of Computational Information Systems 6:14 (2010) 4915-4922
Available at http://www.Jofcis.com

Implicitly Heterogeneous Multi-Stage Programming for FPGAs

Fulong CHEN[1,†], Rajat GOYAL[2], Edwin WESTBROOK[3], Walid TAHA[3]

[1]*Department of Computer Science and Technology, Anhui Normal University, Wuhu Anhui 241000, China*

[2] *Integrated M. Tech, Mathematics and Computing, India Institute of Technology, New Delhi 110016, India*

[3]*Department of Computer Science, Rice University, Houston, Texas 77005, USA*

Abstract

Previous work on semantics-based multi-state programming language design focused on homogeneous and heterogeneous software designs. In homogenous software design, the source and the target software programming languages are the same. In heterogeneous software design, they are different software languages. This paper proposes a practical means to circuit design by providing specialized offshoring translations from subsets of the source software programming language to subsets of the target hardware description language (HDL). This approach avoids manually writing codes for specifying the circuit of the given algorithm. To illustrate the proposed approach, we design and implement a translation to a subset of Verilog suitable numerical and logical computation. Through the translator, programmers can specify abstract algorithms in high level languages and automatically convert them into circuit descriptions in low level languages.

Keywords: Multiple-Stage Programming; Offshoring Translation; Circuit Design; Verilog

1. Introduction

Multi-stage programming (MSP) languages allow the programmer to use abstraction mechanisms such as functions, objects, and modules. In homogenous MSP language design [1], the source and the target software programming languages are the same. In heterogeneous design [2], they are different software languages. Previous work on implicitly heterogeneous multi-stage programming has implemented two target software languages-C and FORTRAN in MetaOCaml. However the conversion from software programming languages to hardware description languages (HDLs) is not supported.

1.1. Contributions

This paper proposes a practical approach to design a formal framework for describing a functional language. In this approach, the programmer doesn't need to know about the details of the target language representation and write his/her program generators to explicitly produce the code in the target HDL.

The design begins by defining the subset of the source OCaml language available to the programmer (Section 2). With the source OCaml language, the OCaml programmer can directly write a OCaml program to solve an arithmetic problem, or use three high-level staging constructs (bracket .<*e*>, escape .~*e* and run .!*e*) in MSP languages [1][2] to produce OCaml program fragments. The proposed target HDL is

† Corresponding author.
Email addresses: chenfulong@gmail.com (Fulong CHEN).

December, 2010

Verilog (Section 3). Its details are only available to offshoring Verilog developers and not to OCaml programmers. It consists of some synthesizable constructs which is a light subset of Verilog. Similar to a compiler, offshoring Verilog translations (Section 4) are provided by the language implementer and not by the programmer. All translations are implemented by an offshoring Verilog translator which can convert the source OCaml program into the target Verilog code. Different from offshoring C and FORTRAN, offshoring Verilog runs on not only PC but also FPGA side. Section 5 presents an efficient simple method to run and verify the generated circuit quickly in FPGA board. We make use of the UART (Universal Asynchronous Receiver/Transmitter) [3] for the communication. Through UART, designers can send the stimulus signals and receive the feedback signals on PC side. Finally, Section 6 measures the Offshoring Verilog performance and the consumption of resources for target circuits.

1.2. Related Work

Traditionally, embedded and digital systems construction involved the following steps: detailed specification, code design, simulation, verification, synthesis and test. However, there is a huge gap between the application system and design scheme. Developers need to know the product function, requirements and how to verify that product meets its requirements. Therefore, specification has become the key to success.

There are different abstraction levels of specifications: system level, algorithm (behavior)level, Register Transfer Level (RTL), logic (gate) level and circuit (switch) level. HDLs are primarily concerned with resource reuse and fully support the last three levels. The two most widely-used and well-supported HDL varieties used in industry are Verilog [4] and VHDL [5].

Verilog is most commonly used in the design, verification, and implementation of digital logic chips at the RTL level of abstraction. A restricted small subset of statements in the Verilog language is synthesizable. Verilog modules that conform to a synthesizable coding-style, known as RTL, can be physically realized by synthesis software. Synthesis-software algorithmically transforms the (abstract) Verilog source into a netlist, a logically equivalent description consisting only of elementary logic primitives (AND, OR, NOT, flipflops, etc.) that are available in a specific VLSI technology. Further manipulations to the netlist ultimately lead to a circuit fabrication blueprint (such as a photo mask-set for an ASIC), or a bitstream file for an FPGA. VHDL is similar to Verilog.

However, HDLs have no enough system-level support for complex circuit design. They don't fully support the algorithm-level specification. Due to the low level of abstraction, designing Verilog/VHDL modules manually takes very skilled engineers and a significant time investment. Writing in HDLs can be more tedious and time consuming than writing a software program to do the same thing. After the design is complete, verification takes even more time.

Currently, there are some expansions in traditional HDLs for enhancing their describing capability. SystemVerilog [6] is a superset of Verilog-2005 [4], which extends some synthesizable design specifications which make this HDL more close to software programming languages, facilitate the design of circuits and enhance the capability to describe a complex algorithm. However, for the same function, the design descriptions in SystemVerilog are still much different from software programs, and designers still cannot ignore some timing constructs.

SystemC [7] has semantically similarities to Verilog and VHDL, and is emerging as a standard for high-level hardware/ software co-design and system-level modeling. However, it is neither used nor supported as widely as Verilog. Some specifications in SystemC are not synthesizable despite being valid for prototyping and simulation. To become a widely used HDL, SystemC still has a long way to go.

Recently, there are also some researches on converting from software programming languages including imperative and functional programming languages to HDLs. C-to-Verilog [8] automates circuit design and allows users to compile existing C functions into RTL Verilog codes. These codes can be synthesized into an FPGA. Each C function is mapped into a Verilog module in which: (1)its arguments are taken as input ports, return value as an output port, *clk* as a clock signal wire, reset as a reset signal wire and *rdy* as a ready signal wire; (2)operations are implemented by some data paths and organized by a FSM. One blemish is that it combines data paths with FSM within an always-statement. Another one is that the only one return value is insufficient for multiple return values.

Fortunately, the high level specification languages, especially functional languages, have some advantages such as clarity, maintainability, functionality and other high-level abstraction mechanisms for narrowing the cracks. Therefore some researchers try to use functional languages to design and reason about hardware. These fall into two camps. The first includes Sheeran's Ruby [9], O'Donnell's Hydra [10], and the Lava lineage of languages [11], all of which utilize μ FP, an extension of Backus's FP (Functional Programming) language, to describe and verify structural descriptions of circuits in formally defined semantics. The second group includes approaches to compiling behavioral models of circuits to hardware, e.g. Mycroft and Sharp's SAFL [12] language which associated FLASH compiler can map a standard ML-like language directly to Verilog.

One outstanding difference between HDLs and functional languages, e.g. OCaml [13], is that the former allows the description of a concurrent system -many parts, each with its own sub-behavior, working together at the same time, and in the latter all run sequentially-one instruction at a time.

2. Source Language

This section presents a subset of the source language in offshoring Verilog. It is aimed at supporting implicitly heterogeneous MSP for basic numerical and logical computation and simulating imperative programming languages in functional languages.

2.1. Syntax in Source Language

As shown in Fig.1, the BNFs of the source OCaml subset are defined in offshoring Verilog.

There are some most interesting syntactic features. (1)The set is essentially a subset of OCaml that has some basic numerical and logical operations. Only boolean, integer and real are supported in constants, variables and expressions. The type of each constant, variable or expression can be identified by the *Trx* parser. (2) Local declarations are used for assignments. The left hand side (LHS) could be a variable or a variable tuple and the right hand side (RHS) could be an expression, an expression tuple or a function application. (3)A complete source program in the above grammar includes a main function name and some other function definitions. There are four kinds of functions such as sequence, condition and tail recursive function which can simulate the controlling structures [14] in imperative programming languages.

(4)Different from the complete OCaml syntax, it just generates an intermediate abstract syntax tree (AST) as the input of offshoring Verilog translator. In fact, in order to simplify the offshoring Verilog translator and make use of the *Trx* parser, it is not a strict subset of the complete OCaml's BNFs but rather it is an intermediate form that accepts a very similar language.

Variable Name	$\hat{x} \in \mathbb{X}$
Function Name	$\hat{f} \in \mathbb{X}$
Constant	$\hat{c} \in \mathbf{Bool} \cup \mathbf{Int} \cup \mathbf{Real}$
Unary OP.	$\hat{o_1} \in \{\ \mathbf{not}\ , -, -.\}$
Binary OP.	$\hat{o_2} \in \{\ \texttt{\&\&}\ ,\ \texttt{||}\ , +, -, *, /, +., -., *., /., >, >=, <, <=, !=, =\}$
Expression	$\hat{e} ::= \hat{c} \mid \hat{x} \mid (\hat{e}) \mid \hat{e}\ \hat{o_2}\ \hat{e} \mid (\hat{o_1}\ \hat{e})$
Variable Tuple	$\hat{X} ::= \hat{x}, \hat{X} \mid \hat{x}$
Expression Tuple	$\hat{E} ::= \hat{e}, \hat{E} \mid \hat{e}$
Declaration	$\hat{d} ::=$ **let** $(\hat{X}) = (\hat{E})$ **in** $\hat{d} \mid$ **let** $(\hat{X}) = \hat{f}(\hat{E})$ **in** $\hat{d} \mid \varepsilon$
Sequence Function	$\hat{f}_s ::=$ **let** $\hat{f}(\hat{X}) = \hat{d}(\hat{X})$
Condition Function	$\hat{f}_c ::=$ **let** $\hat{f}(\hat{X}) = \hat{d}$ **if** $\hat{x}$ **then** $(\hat{X})$ **else** $(\hat{X})$
Tail recursive Function	$\hat{f}_t ::=$ **let rec** $\hat{f}(\hat{X}) = \hat{d}$ **if** $\hat{x}$ **then** $\hat{f}(\hat{X})$ **else** $(\hat{X})$
Functions	$\hat{F} ::= \hat{f}_s$ **in** $\hat{F} \mid \hat{f}_c$ **in** $\hat{F} \mid \hat{f}_t$ **in** $\hat{F} \mid \varepsilon$
Main Function	$\hat{p} ::= \hat{F}\ \hat{f}$

Fig. 1 Grammar for the OCaml Subset

Bit Constant	$c ::= 0 \mid 1$
Signal Name	$s \in \mathbb{X}$
Signals	$S ::= s \mid s, S$
Module Name	$m \in \mathbb{X}$
Instance Name	$n \in \mathbb{X}$
Module Definition	$D ::=$ `module` $m(p); b$ `endmodule` $D \mid \varepsilon$
Port Type	$t ::=$ `input` $\mid$ `output`
Port Declaration	$p ::= t\ s, p \mid t\ s$
Module Body	$b ::= V\ A\ M$
Variable Declaration	$V ::=$ `wire` $s; V \mid \varepsilon$
Assignment Statement	$A ::=$ `assign` $s = s; A \mid$ `assign` $s = c; A \mid \varepsilon$
Module Instance	$M ::= m\ n(S); M \mid \varepsilon$

Fig. 2 Grammar for the Verilog Subset

The following program fragment is a complete source program used to calculate the Fibonacci value as an example in the above grammar.

```
(*Example 1. Fibonacci*)
let rec fibo (f0,f1,n)=
      let cond=n!=0 in let new_n=n-1 in let f2=f0+f1 in
      if cond then fibo (f1,f2,new_n) else f0
in
let fib (n)= let f=fibo (0,1,n) in f in fib
```

2.2. Sequence Function

A sequence function is only composed of some declarations and its return value(s). These declarations are some ordered assignment statements in which numerical or logical expressions are assigned to variables. In Example 1, *fib* is a sequence function.

2.3. Condition Function

A condition function performs different computations or actions depending on whether a programmer-specified boolean condition evaluates to **true** or **false**. In a condition, besides some declarations, **if-then-else** construct is to decide which branch's value(s) will act as the return value(s).

2.4. Tail Recursive Function

In functional programming languages, tail recursion (or tail-end recursion) is a special case of recursion in which the last operation of the function, the tail call, is a recursive call. Such recursions can be easily transformed to iterations in imperative programming languages. In Example 1,*fibo* is a tail recursive function.

3. Target Language

Fig.2 presents the BNF of the target Verilog grammar subset. This set is essentially a very small subset of Verilog that has structural models. Besides some basic gates including AND, OR and NOT, some basic circuits such as numerical and logical operation units, registers, multiplexer and controllers are pre-designed or customized (Table1).

Table 1 Basic Circuits

Module Name	Output Ports	Input Ports	Functionality
land,lor	done,c	a,b,start,clk	Logical AND,OR
lnot	done,c	a,start,clk	Logical NOT
gt,ge,lt,le,eq,neq, fgt,fge,flt,fle,feq,fneq	done,c	a^{32},b^{32},start,clk	Integer and real >,>=,<,< =,==,!=
add,sub,mul,div, fadd,fsub,fmul,fdiv	done,c^{32}	a^{32},b^{32},start,clk	Integer and real +,-,*,/
neg, fneg	done, c^{32}	a^{32},start,clk	Negative integer or real
data_reg	dout	din,load,clr_n,clk	1-bit data register
data_reg32	$dout^{32}$	din^{32},load,clr_n,clk	32-bit data register
mux	c	a,b,select	1-bit multiplexer
mux32	c^{32}	a^{32},b^{32},select	32-bit multiplexer
loopctrl	store,done,en,select	start,ready,finished,clk	Loop controller
mainctrl	receive,start,send	ready,done,finished,clk	Main controller
receiver	ready,$data^{+}$	rxd,receive,clk	UART frame receiver
transmitter	finished,txd	$data^{+}$,send,clk	UART frame transmitter

4. Offshoring Verilog Translation

The offshoring Verilog translator is like a compiler. Given an AST of the OCaml program defined in the source language grammar, it can produce the concerning Verilog AST according to some translation rules. Unsupported source programs will generate errors. In the translator, translations for expression, declaration, sequence, condition and loop are the most important parts.

4.1. Translation for Expressions

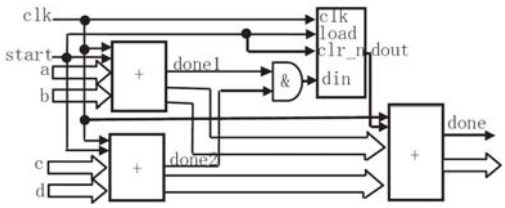


Fig.3 Structure of (a+b)+(c+d)

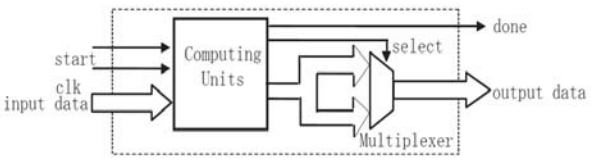


Fig.4 Structure of Condition Function

In Fig.1, there are three types of expressions including: (1) boolean expressions producing **true** or **false**; (2) integer expressions producing integer values; (3) real expressions producing real values. The operations in these expressions will be mapped to the corresponding circuits in Table 1. Each operation will be triggered

by the signal *start* to work and produce a signal *done* to show that it is completed. Therefore, the ordered operations work under the dependant relations between the two signals *start* and *done*. For example, the expression (*a*+*b*)+(*c*+*d*) will work as shown in Fig.3.

4.2. Translation for Declarations

A declaration is like an assignment in which some values generated by expressions or function calls are assigned to some variables. In the translator, it will be mapped to some continuous assignment statements in Verilog subset. For the *fibo* function in Example 1, three parallel continuous assignments are generated for variables *cond*, *new_n* and f2 even if they are calculated sequentially in OCaml.

4.3. Translation for Sequence Functions

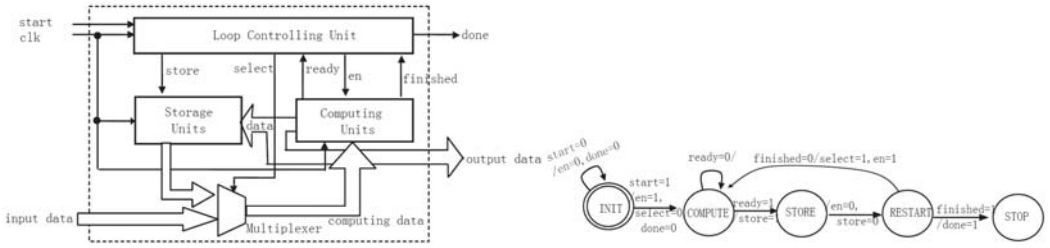
A sequence function will be mapped to a module which has some units for implementing the operations in its declarations. Its input arguments will be mapped to **input** ports and its return variable(s) carrying return value(s) will be mapped to **output** ports. It has a trigger signal *start* and a completeness signal *done*.

4.4. Translation for Condition Functions

Different from sequence functions, a condition function has two possible exits. A condition variable's value decides which exit will produce the return value(s). Therefore, besides some computing units, it will also produce some multiplexer to select the value(s) satisfying the condition (Fig.4).

4.5. Translation for Tail Recursive Functions

Like the condition function, a tail recursive function also has two possible exits. The difference is that it will re-enter itself in one exit and leave in the other exit. In order to implement this function, in addition to some computing units and multiplexer, a loop controller and some storage units are needed in its Verilog module. Fig.5(a) shows its structure. The module will work under the control of a loop controller, which is a pre-designed circuit in Table 1 and works according to the state chart as shown in Fig.5(b). In its module, the signal *finished* is connected with its condition variable.



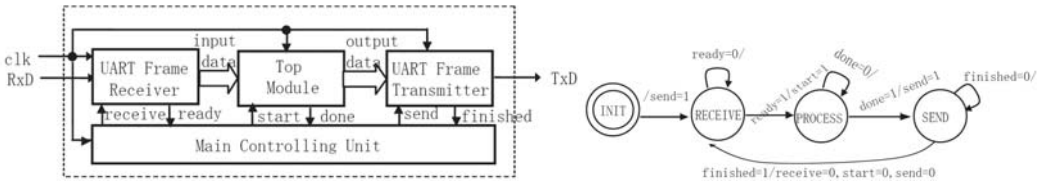
(a)Structure of Loop Function (b) State Chart of Loop Controller

Fig.5 Loop Function

5. Running the Circuit on the FPGA board

As a final task, we can verify how the design performs in the target application. Some IC corporations such as NEC, Qualectron, NTE, etc., provide plentiful in-circuit test (ICT) [17] equipments helpful to circuit verification. One limitation is that they can only generate and monitor limited number of signals. The other limitation is that they need specific equipments and cannot be controlled on PC side, that is, hard to program on PC side.

However the phenomenal growth in design size and complexity continues to make the process of FPGA design verification a critical bottleneck for today's FPGA systems. Limited access to internal signals, advanced FPGA packages, and printed circuit board (PCB) electrical noise are all contributing factors in making design debug and verification the most difficult process of the design cycle.



(a)Structure of Running Platform (b) State Chart of MCU

Fig.6 In-circuit Running Platform

In order to break those limitations, we make use of the UART for the communication. An in-circuit running platform (Fig.6(a)) includes an UART frame receiver used to receive serial input values from PC with UART and convert them into an input parallel vector ($data^+$), a top module for the main function, an UART frame transmitter used to convert the output vector ($data^+$) of the main function into serial signals and send them to PC with UART, and a Main Controlling Unit (MCU) used to control the entire periodic work process in Fig 6(b). In addition to the clock signal provided by the FPGA board, MCU has three input signals (*ready, done, finished*) and three output signals (*receive, start, send*).

6. User Interfaces

The interfaces for OCaml programmer are easy to use. Two interfaces are supported in offshoring Verilog. Programmers can now write .!*Trx.run_verilog .<e>*., where *e* may be the program fragment like Example 1.The function *runV* is provided for users to send input data to FPGA and receive the output data from FPGA.

7. Conclusions and Future Work

To our best knowledge, this paper presents a practical way to design circuit by providing specialized offshoring translations from a subset of the source software programming languages to a subset of the target HDLs. This approach avoids manually writing codes for specifying the circuit of the given algorithm. To illustrate the proposed approach, we design and implement a translation to a subset of Verilog suitable numerical and logical computation. It supports Hardware/Software (HW/SW) co-verification - verified circuit component on FPGA side and controlling program on PC side. Our design will enable the quick

development of new IC product and provides an FPGA-based test bed for possible IC implementations. Currently only some numerical and logical operations are supported in the source language. More complex operations could be added to extend the functionality of offshoring Verilog.

Acknowledgments

This paper is supported by Project of Scientific Research for Young University Teachers of Anhui Province of China under Grant No.2008jq1057. We also would like to thank Ronald Garcia, Cherif Salama, Jun Inoue and Rex Page for commenting this paper.

References

[1] Taha,W. A Gentle Introduction to Multi-stage Programming, Part II. In Proceedings of 2nd Summer School on Generative and Transformational Techniques in Software Engineering, pages 260-290 ,2007.
[2] Eckhardt, J., Kaiabachev, R., Pa˘sali´c, E., Swadi, K., Taha, W. Implicitly Heterogeneous Multi-Stage Programming. In Proceedings of the ACM International Conference on Generative Programming and Component Engineering,pages 275-292,2005.
[3] Strangio, C. E. The RS232 Standard. CAMI Research Inc., Acton, Massachusetts (1993) 15.
[4] IEEE Standard Board. IEEE Standard Verilog Hardware Description Language(IEEE Std 1364-2005). IEEE, New York,2006.
[5] IEEE Standard Board. IEEE Standard VHDL Language Reference Manual(IEEE Std 1076-2008). IEEE, New York,2009.
[6] IEEE Standard Board. IEEE Standard for SystemVerilog- Unified Hardware Design, Specification, and Verification Lanaguage(IEEE Std 1800-2005). IEEE, New York,2005.
[7] IEEE Standard Board. IEEE Standard SystemC Language Reference Manul(IEEE Std 1666-2005). IEEE, New York,2005.
[8] C to verilog, http://www.c-to-verilog.com/index.html.
[9] Sheeran,M. Hardware Design and Functional Programming: a Perfect Match. Journal of Universal Computer Science, Vol.11, No.7,pages 1135-1158 ,2005.
[10] O'Donnell,J. From Transistors to Computer Architecture: Teaching Functional Circuit Specification in Hydra, In Proceedings of Symposium on Functional Programming Languages in Education, Springer-Verlag LNCS 1022, pages 195–214 ,1995.
[11] Bjesse,P.,Claessen,K.,Sheeran,M. and Singh,S. Lava- Hardware Design in Haskell.In Proceedings of the 3rd ACM SigPlan International Conference on FunctionalProgramming, pages 174-184 ,1998.
[12] Mycroft, A. and Sharp, R.W. Hardware Synthesis using SAFL and Application to Processor Design. In Proceedings of the 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods, pages 13- 39 ,2001.
[13] Chailloux E., Manoury P.,Pagano B. Developing Applications With Objective Caml. OReilly, Paris ,2000.
[14] Abelson, H., Sussman, J. Structure and Interpretation of Computer Programs, 2nd edition. MIT Press,1993.
[15] Kiselyov, O., Swadi K. N., Taha, W. A methodology for generating verified combinatorial circuits. In Proceedings of the 4th ACM international conference on Embedded software, pages 249-258,2004.
[16] Salama, C., Malecha, G., Taha,W. Static Consistency Checking for Verilog Wire Interconnects. In Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation ,pages 121-130 ,2009.
[17] Bateson, J. In-circuit Testing. Van Nostrand Reinhold, New York, 1985.
[18] Chen, X., Huang, W., Park, N. Design verification of FPGA implementations. IEEE Design & Test of Computers, 16(2), pages 66-73,1999.