Multi-Stage Programming for High-Level Description of Circuit Families

Fulong Chen
Department of Computer Science
Anhui Normal University
Wuhu, Anhui, China

Walid Taha
Department of Computer Science
Rice University
Houston, Texas, USA

Lei Deng
School of Computer
Northwestern Polytechnical University
Xi'an, Shaanxi, China

***Abstract*—Hardware description languages use a low level of abstraction making the task of hardware designers error-prone and time consuming. The resulting descriptions tend to be lengthy and hard to reason about. This paper presents a minimal functional language for describing hardware circuits. Using multi-stage programming concepts, Fast Fourier Transform circuit descriptions can be generated in this language from a higher level generic description. It bridges the gap between the generated code and the required low-level hardware description by presenting the translator to automatically convert the generated code into Verilog, a streamline hardware description language.**

I. Introduction

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.

The two most widely-used and well-supported HDL varieties used in industry are Verilog [2] and VHDL [3]. 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.

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. One outstanding difference between HDLs and functional languages, e.g. OCaml [1], 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.

A typical design is the Fast Fourier Transform (FFT) circuit design. Usually used in wireless communications, voice and image processing, spectrum analysis and so on, e.g., JPEG, MPEG and H.26x, FFT algorithm is an improvement of Discrete Fourier Transform (DFT) with high speed and low cost in resources. It needs only several dozen lines of C or Matlab codes. However it is very difficult to be described in HDLs. And it is also difficult to convert C or Matlab program of FFT into HDL code.

This paper is a step toward creating a new method to design digital circuits from higher level abstractions. Using Multi-Stage Programming(MSP) concepts [6], a family of FFT circuit descriptions can be automatically and efficiently generated from a higher level generic description [7][11]. Starting from this FFT example (Section 2), we identify a minimal functional language for describing hardware circuits (Section 3). We show how to translate descriptions written in this minimal language to Verilog (Section 4). We finally discuss three different design schemes to implement FFT (Sections 5.1,5.2 and 5.3) and compare the resulting circuits (Section 5.4).

II. Multi-Stage Programming for Generic Circuit Design

This methodology includes two stages:

- Describing the behavior for some circuit in MetaOCaml[6],which extends OCaml with three constructs for staging and makes generated functions more efficient,and generating a simple OCaml function;
- Translating the simple OCaml function and generating Verilog codes in a given scheme.

Decimation-in-time FFT $F(x, N)_k$ of the N-point complex-valued vector x is calculated by:

$$F(1,x)_0 = x_0,$$
$$F(1,x)_k = F(\tfrac{N}{2}, E(x))_k + F(\tfrac{N}{2}, O(x))_k \cdot w_N^k,$$
$$F(N,x)_{k+\frac{N}{2}} = F(\tfrac{N}{2}, E(x))_k - F(\tfrac{N}{2}, O(x))_k \cdot w_N^k,$$

where

$$E(x)_j = x_{2j}, O(x) = x_{2j+1}, j = 0, 1, ..., N-1,$$
$$w_N^k = e^{-2\pi k/N} = \cos(-\tfrac{2k\pi}{N}) + i\sin(-\tfrac{2k\pi}{N}),$$
$$k = 0, 1, ..., N-1.$$

The first stage is high-level modeling for FFT algorithm with MetaOCaml. It uses staged and abstract interpretation to construct a simple OCaml function, eliminate redundancy and recursion in previous work[14], and split complex operations into some simple sequential floating point additions, subtractions and multiplications. The following code shows the OCaml function generated for 4-point FFT:

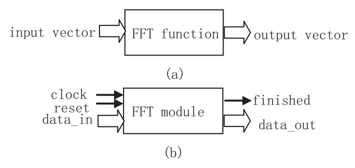


Fig. 1. FFT function and module

```
let fft4= fun x_1->
let x1_2 = x_1.(0) in
let y1_3 = x_1.(1) in
let x1_4 = x_1.(2) in
let y1_5 = x_1.(3) in
let x1_6 = x_1.(4) in
let y1_7 = x_1.(5) in
let x1_8 = x_1.(6) in
let y1_9 = x_1.(7) in
let z_10 = (x1_2 +. x1_6) in
let z_11 = (y1_3 +. y1_7) in
let z_12 = (x1_4 +. x1_8) in
let z_13 = (y1_5 +. y1_9) in
let z_14 = (x1_2 -. x1_6) in
let z_15 = (y1_3 -. y1_7) in
let z_16 = (x1_4 -. x1_8) in
let z_17 = (y1_5 -. y1_9) in
x_1.(0) <- (z_10 +. z_12);
x_1.(1) <- (z_11 +. z_13);
x_1.(2) <- (z_14 +. z_17);
x_1.(3) <- (z_15 -. z_16);
x_1.(4) <- (z_10 -. z_12);
x_1.(5) <- (z_11 -. z_13);
x_1.(6) <- (z_14 -. z_17);
x_1.(7) <- (z_15 +. z_16);
x_1
```

It is a top-level function (Fig. 1(a)) in which input vector x_1 is a real number array. After finished, it returns x_1 as its output vector.This function can be implemented directly in syntheziable Verilog codes(Fig.1(b)).

In this paper,we focus on the automatic translation from OCaml function into Verilog codes.

III. Syntax for ocaml function

By analyzing the generated code in the previous example and the code generated for various sizes of FFT circuits, the following OCaml subset is identified. In the base language representation of the target language subset, the types in the OCaml subset must match those of the Verilog subset. OCaml base types `int` and `float` map to Verilog `wire` or `reg`.One-dimensional arrays are represented by OCaml `array` type.

The following presents the BNF of the base OCaml syntax minimal subset.

The main restrictions imposed on this base minimal syntax subset are: a) no loop command; b) no division operation; c) no recursion operation; d) and no branch structure so as to be able to be easily translated into Verilog code.

Variable	x	$\in$	X
Expression	e	$::=$	$(e) \mid x \mid r \mid i \mid e\ o\ e$
Statement	s	$::=$	$e \mid s; s$
			$\mid$ **let** $x\ = e$ **in** $s \mid x.(e) < -e$
Operator	o	$::=$	$+. \mid -. \mid *.$
Real	r	$::=$	$d\{d\}.\{d\}$
Integer	i	$::=$	$d\{d\}$
Digit	d	$::=$	$0..9$
Program	p	$::=$	**let** $x =$ **fun** $\{x\}- > s;;$

IV. Translation to Verilog

The output in the first stage is a simple OCaml function. Then in the second stage, it is to convert that function into Verilog code. Therefore, it is necessary to establish the mapping relationship between source language- OCaml and target language- Verilog so as to accomplish the transformation, that is, to transform OCaml function into Verilog code. In the above generated code, apart from data transmission, there are only some 32-bit IEEE754 floating point computations such as additions, subtractions and multiplications (will be existent in 4 more points FFT). These computations form a sequence of 4-tuple(one operator and three operands):

`<op,opa,opb,opc>.`

Each computation can be implemented by an Arithmetic Unit (AU).

The transformation is implemented in three distinct parts:

- OCaml lexical analyzer: extracts integer, real, symbol (`+.`, `-.`, `*.`, `<-`, `=`, `(`, `)`, `;`) and keywords (`let / in`) from the simple OCaml program.
- Intermediate code generator: translates statements and expressions and generates a sequence of 4-tuple `<op, opa, opb, opc>`, which are four types of operations (`op` is respectively `=`, `+.`, `-.`,or `*.`).
- Verilog code generator: converts the 4-tuple sequence into Verilog code in accordance with the selected design scheme for implementing the FFT circuits.

A. OCaml lexical analyzer

In order to analyze the simple OCaml program, the analyzer is defined to split the text of input program into independent lexical units, called lexemes, with OCaml type:

```
type lexeme= Lint of string
(*integer*)
| Lfloat of string (*real*)
| Lident of string (*identifier*)
| Lsymbol of string (*operator or
symbol*)
| Lend;;
```

A particular lexeme denotes the end of an expression: `Lend`.It is not present in the text of input program, but is created by the analyzer.

For the OCaml function of 4-point FFT not including the first and the last line in Section 2, the analyzer, which will be invoked by the intermediate code generator, extracts the following lexemes:

```
 {Lident "let",Lident "x1_2",
Lsymbol "=",Lident "x_1[0]",
Lident "in",...,Lident "x_1[7]",
Lsymbol "<-",Lsymbol "(",
Lident "z_15",Lsymbol "+",
Lident "z_16",Lsymbol ")"}
```

B. Intermediate code generator

The intermediate code generator, working in a pushdown automaton,reads a string, extracts its lexemes by calling the analyzer, passes them to the operator stack and the operand stack,or removes them from the stacks while forming a computation with some operation functions on stack- `push`, `pop` and `top`,and finally, decomposes the OCaml function into a sequence of 4-tuple. Each 4-tuple is one computation or data transmission.

In the above 4-point FFT OCaml function, a sequence of 4-tuple will be generated as follows:

```
 {
<=,x1_2, ,x_1.(0)>,
..., <=,x1_9, ,x_1.(7)>,
<+.,x1_2,x1_6,z_10>,
..., <-.,y1_5,y1_9,z_17>,
<+.,z_10,z_12,x_1.(0)>,
..., <+.,z_15,z_16,x_1.(7)>
}.
```

C. Verilog code generator

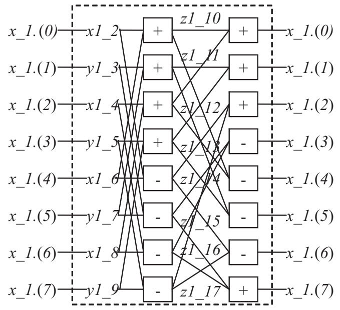
After obtaining the sequence of 4-tuple, the Verilog code generator translates it into Verilog codes corresponding to the chosen design scheme. Combining them with Verilog module structure, the generator constructs a FFT module and builds any size of FFT circuits.Details will be presented in the following section.

V. Design schemes

A. Parallel circuits

In this scheme, the generator generates an AU for each computation in the sequence of 4-tuple and connects them according to their data dependence relationship. For two operations: `<op1, opa1, opb1, opc1>` and `<op2, opa2, opb2, opc2>`, which are respectively implemented by AU1 and AU2, if `opc1` and `opa2` (or `opb2`) have the same variable name, AU1 needs to connect its completed output port with the enabled input port of AU2 so as to fire the latter. Certainly, if an operation has no pre-operation, its AU's enabled input port is connected with the power supply wire; if an operation has no follow-up operation, it's AU's completed output port needs to be connected with an AND gate for generating the signal for the finished output port of the FFT module.

As shown in Fig.3, the parallel implementation in parallel circuits brings high performance [8][9]. However they cost too many resources and wires. For example, in 4-point FFT(Fig.2(a)), 8 adders, 8 subtractors, 8×32 bits input and output wires are needed even if they don't work at the same time. And in FFT8, we need 16×32 bits input and output wires, 26 adders and subtractors, and 4 multipliers. In 16-point FFT(Fig.2(b)), we need 32×32 bits input and output wires, 75 adders and subtractors, and 28 multipliers. For n-point FFT, number of resources and wires will rapidly expand at a rate of $O(nlogn)$. Therefore, the parallel implementation brings low scalability so that it is not fit for customization of large size of FFT circuits.



(a)

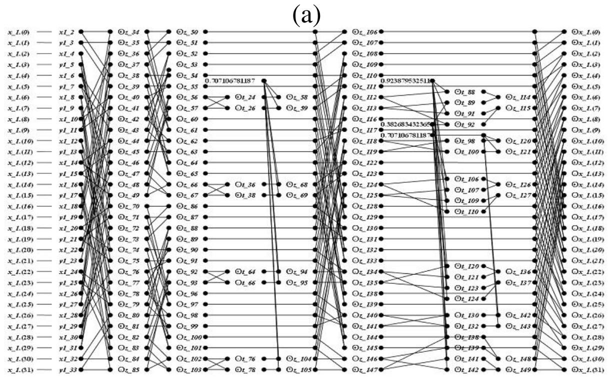

(b)

Fig. 2. FFT Structure

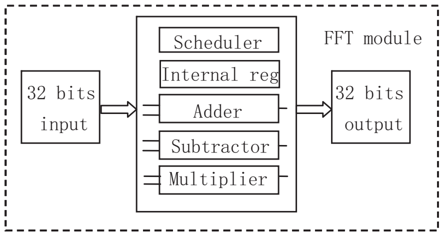


Fig. 3. FFT module for the sequential circuit

B. Sequential circuits

In sequential circuits, the FFT module is a scheduler, which only needs to repeatedly invoke an adder, a subtractor and a multiplier, controlled under a Finite State Machine (FSM), as shown in Fig. 3. The scheduler is a state machine working in three steps (Fig. 4): 1) it receives input data by its input ports from the invoker and puts them into internal registers; 2) each time it puts two data on the input ports of the corresponding AU and enable it to compute, and then save the result in internal registers until all computations are completed; 3) it sends all output data by its output ports.

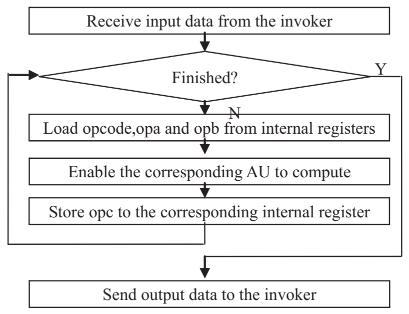

Receive input data from the invoker
Finished?
Y
N
Load opcode,opa and opb from internal registers
Enable the corresponding AU to compute
Store opc to the corresponding internal register
Send output data to the invoker

Fig. 4. Workflow for the sequential FFT

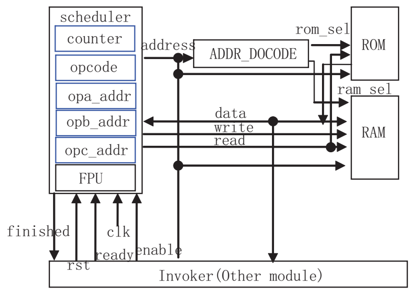


Fig. 5. Reconfigurable FFT circuit

C. Reconfigurable circuits

Similar to sequential circuits, the scheduler repeatedly invokes one adder, one subtractor and one multiplier (Fig.5). But it is no longer responsible for containing a large number of data and operations in its internal registers. Fig. 6 shows its workflow. Transform data is stored in RAM by the FFT invoker before transforming. When all data are ready, the FFT invoker sends a signal to the FFT scheduler. The latter fetches an operating code from ROM, analyzes it and gets addresses of three operands- `opa`, `opb` and `opc`. Then, it loads `opa` and `opb` from RAM, and puts them on the input ports of the corresponding AU and enables it. Once the AU finishes its computation, the FFT scheduler stores the result- `opc` into the RAM and fetches the next operating code from the ROM. After all computations are finished, the FFT scheduler sends a signal to the FFT invoker, and the latter can take the result of FFT transform from the RAM. Herein, the scheduler isn't changed by the Verilog codes generator because any size of FFT uses the same scheduler. Therefore, it is designed as an available module. What the generator needs to do is to encode those operating codes stored in the ROM. Each operating code includes four parts: one part is the code of the operator; the other three parts are addresses of three operands. Each operand may be a variable or a constant. If it is a variable, its address is the address of the operand in the RAM; otherwise, it's the address of a constant in the ROM. So the generator also needs to encode those constants and store them in the ROM.

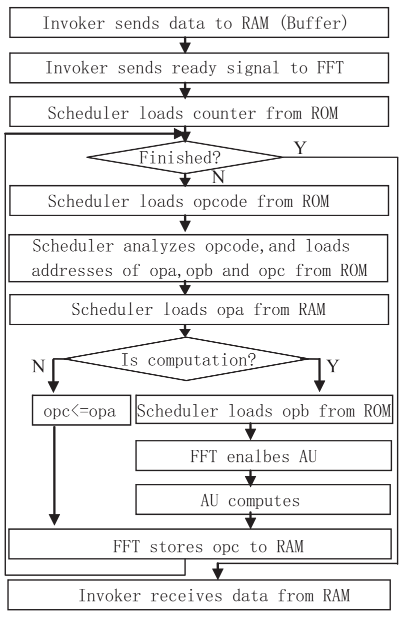

Invoker sends data to RAM (Buffer)
Invoker sends ready signal to FFT
Scheduler loads counter from ROM
Finished?
Y
N
Scheduler loads opcode from ROM
Scheduler analyzes opcode, and loads addresses of opa, opb and opc from ROM
Scheduler loads opa from RAM
Is computation?
N
Y
opc<=opa
Scheduler loads opb from ROM
FFT enalbes AU
AU computes
FFT stores opc to RAM
Invoker receives data from RAM

Fig. 6. Workflow for the reconfigurable FFT

D. Comparison

Table 1 lists numbers of cycles for parallel, sequential and reconfigurable circuits. In parallel circuits with a high performance and a waste of resources, due to direct input and output, the number of cycles for finishing FFT transformation only depends on the length of critical path of all computations and increases slowly. That number increases quickly in sequential and reconfigurable circuits.

However Table 2 indicates that sequential circuits (synthesized in Xilinx ISE9.2i XST with automotive Spartan3 Family, XC3S200 Device and FTG256 Package) only need a small number of I/O wires and AUs, and save lots of resources. Sequential circuits have new problems: low speed and many internal registers/wires. In order to improve the speed, pipelining [12][10] and multi-fired mechanisms can be taken into account. For decreasing the number of internal registers/wires, the scheduler reuses those registers which have completed the functions in former computations. For high scalability and reconfigurability, reconfigurable circuits will bring the same performance and save much more resources- without many internal registers and wires. It can be adjusted for the purpose of implementing any size of FFT without changing its scheduler. Designers only need to change the operating code stored in the ROM. It can bring better performance if butterfly circuits [8][9] are used for butterfly calculation.

VI. Related work

The major methods of implementation of FFT algorithm include DSP, ASIC, FPGA, etc. DSP has the flexibility of software-only implementation. However, the large-scale computing FFT will use up a lot of computing time of DSP, bringing the system to reduce its data throughput, thus affecting its performance. High-speed real-time digital signal processing

TABLE I
NUMBER OF CYCLES FOR FFT

Size		2	4	8	16	32
Parallel Circuits	I/O	0				
	+-*	1	2	5	8	11
Sequential Circuits	I/O	4/4	8/8	16/16	32/32	64/64
	+/-/*	2/2/0	8/8/0	26/26/4	75/75/28	199/199/108
Reconfigurable Circuits	Data Transmission	4	8	16	32	64
	+/-/*	2/2/0	8/8/0	26/26/4	75/75/28	199/199/108

TABLE II
RESOURCES OF FFT CIRCUITS

Size		2	4	8	16	32
Parallel Circuits	Floating Point Adder(s)/Subtractor(s)/Multiplier(s)	2/2/0	8/8/0	26/26/4	74/74/28	194/194/108
	I/O wires (*32 bits) for data	4/4	8/8	16/16	32/32	64/64
Sequential Circuits	Floating Point Adder(s)/Subtractor(s)/Multiplier(s)	1/1/0		1/1/1		
	I/O wires (*32 bits) for data	1/1				
	D-type flip-flop(s)	452	964	2500	6852	18116
	Fixed point Adder(s)/Comparator(s)/Multiplexer(s)	3/3/32				
Reconfigurable Circuits	Floating Point Adder(s)/Subtractor(s)/Multiplier(s)	1/1/0		1/1/1		
	I/O wires (*32 bits) for data	1/1				
	D-type flip-flop(s)	167				
	Fixed point Adder(s)/Comparator(s)/Multiplexer(s)	2/1/32				

system has a high performance requirement. Therefore, FFT computation is usually implemented in ASIC or FPGA. Traditional method for constructing FFT circuits is to directly write HDL codes and synthesize its circuits.

Recently, there are also some researches on converting from software programming languages to HDLs. C-to-Verilog[13] automates circuit design and allows users to compile existing C functions into RTL Verilog codes. 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.

Our method is to specify a function module in a functional language which is a subset of OCaml. Being different than C, which is an imperative language, OCaml is a type-safe strict functional programming language without side effects, more strong in mathematical function programming than C. Without the use of pointers, a function in OCaml can return multiple values. And under the help of the Verilog code generator, we can automatically generate its Verilog codes. This is helpful to programmers proficient in software programming languages and unfamiliar with HDLs. We don't need to write HDL codes for every FFT circuits.

VII. CONCLUSIONS

Multi-stage programming for customizing circuits is applicable to the design of digital system. First of all, with high-level description method, modeling for application systems, eliminating redundant computations, verifying the correctness, refining and optimizing computations, it transforms the solutions to the complex problems into a sequence of some simple operations. Then, using low-level language to describe the simple sequence into circuits, this process can be achieved by automatic code converter and generator. This approach puts more work on the high-level modeling stage. As the result of high-level description languages which are highly abstract, scalable, reusable, easy to specify solutions to problem and guarantees their accuracy, and in the latter stage for design of circuits, developers almost do not need do further verification, cost in a very low price and can automatically customize a whole family of circuits. This improves the productivity of hardware design.

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 Cherif Salama for helping us with writing this paper.

REFERENCES

[1] E. Chailloux,P. Manoury and B. Pagano,*Developing Applications With Objective Caml*,Paris,France:O'Reilly,2000

[2] *IEEE Standard Verilog Hardware Description Language*,IEEE Std 1364-2005,2006.

[3] *IEEE Standard VHDL Language Reference Manual*,IEEE Std 1076-2008,2009.

[4] *IEEE Standard for SystemVerilog-Unified Hardware Design, Specification, and Verification Lanaguage*,IEEE Std 1800-2005,2005.

[5] *IEEE Standard SystemC Language Reference Manul*,IEEE Std 1666-2005,2005.

[6] W. Taha,"Gentle Introduction to Multi-stage Programming" in *Proc.GTTSE'07*,2007,p.260-290.

[7] O. Kiselyov,K. N. Swadi and W.Taha,"A Methodology for Generating Verified Combinatorial Circuits" in *EMSOFT'04*,2004,p.249-258.

[8] W.Chao and H. Y. Peng,"Twin Butterfly High Throughput Parallel Architecture FFT Algorithm" in *ISISE'08*,2008,p.637-640.

[9] J. Y.Yu and Y. Li,"An Efficient Conflict-Free Parallel Memory Access Scheme for Dual-Butterfly Constant Geometry Radix-2 FFT Processor" in *ICSP'08*,2008,p.458-461.

[10] X. Fan,"A VLSI-Oriented FFT Algorithm and Its Pipelined Design" in *ICSP'08*,2008,p.414 - 417.

[11] O. Kiselyov and W. Taha,"Relating FFTW and Split-Radix" in *ICESS'04*,2004,p.488 - 493.

[12] H. Xiao ,A.Pan,Y. Chen and X. Zeng, "Low-cost reconfigurable VLSI architecture for fast fourier transform", *IEEETransactions on Consumer Electronics*, vol. 54, pp. 1617–1622, Nov. 2008.

[13] C to Verilog,http://www.c-to-verilog.com/index.html

[14] A Staged Implementation of FFT, http://www.metaocaml.org/fft.ml