Translating SMALL Programs to FPGA Configurations

Ying Shen and Theodore S. Norvell
Faculty of Engineering and Applied Science
Memorial University of Newfoundland
St. John's, Newfoundland, Canada, A1B 3X5
email: {shen, theo}@engr.mun.ca

1. Introduction

SMALL [1] is a hardware-oriented parallel programming language with synchronous communication, which is being developed at the Memorial University of Newfoundland. It consists of a simple set of notations and is suitable for designing the behaviour of synchronous circuits. Figure 1 shows a simple SMALL program. The control structures include standard sequencing constructs, and a parallel composition construct. "Assert" statements, such as *outBit ! 1*, transfer values to signals during the current clock cycle. The `tick` statement indicates the transition to the next clock cycle; each loop iteration implicitly ends with a transition to the next clock cycle. All statements executed in the same clock cycle are executed simultaneously.

Our aim is to create hardware with the click of a single button, so that hardware can be as created as easily as software. The present paper discusses a compiler for translating SMALL programs to FPGA configurations. This work is part of a larger programme to create effective programming languages and tools for reasoning about hardware and designing hardware with very short design cycles. In our compiler, SMALL programs are first converted to Parallel Algorithmic State Machine (PASM) charts (see Figure 2). These PASM charts are then converted to netlists consisting of and-, or-, and not-gates, and flip-flops, which are then written out as structural VHDL descriptions. Finally, FPGA configurations are obtained by using commercial synthesis and place-and-route tools.

The compiler's front-end and the Netlist Generator are written in the applicative programming language Haskell.

The most closely related work is conducted by Page and his group [2]. They have compiled programs written in a subset of occam into FPGAs.

2. The PASM Chart for an Example SMALL Program

Figure 1 shows an example SMALL program describing a parity generator and some test inputs. At each clock period, the *outBit* is the parity of those *inBits* that have been seen in the previous clock periods. This program is converted to the PASM chart shown in Figure 3, by the compiler's front-end. The PASM chart consists of signals, registers, nodes, and edges. Signals are used to communicate between statements executing in parallel. Registers, by contrast, are used to pass values forward in time. Most of the registers and signals in the PASM chart come from the source code; additional signals are introduced (*#r*1* and *#l*0,* in the example) to coordinate the termination of parallel statements. Registers and signals are either "global" or "local"; global signals and registers are used to communicate with other circuits and devices. Each signal or register's type is a multidimensional array of Booleans. The PASM chart is essentially a flow chart in which time passes only at the "state" nodes (see Figure 3).

3. Netlist Generator

The Netlist Generator transforms PASM charts to structural VHDL descriptions involving only gates and flip-flops. The Netlist Generator must do the following:

- Generate circuits for the signals and registers.
- Generate circuits for the nodes.
- Generate circuits for the expressions.
- Link all the above circuits according to the edges.
- Output a VHDL file representing the final netlist circuits.

Generating Signal, Register, Assert and Assignment Circuits

Signals are represented simply by arrays of or-gates and their output wires. For any node that may assert a signal, we create inputs to the or-gates. Figure 4 shows a global Boolean signal.

Registers are slightly more complex (see Figure 5). We need to send both an indication that a change is needed (*assign*) and a new value (*val*). Each assignment node that may assign to a register contributes an input to each of the initial or-gates (see Figure 6).

Assignment to (or assertion of) subscripted arrays requires computing which "assign" (or "assert") wires should be active, based on the subscript.

Generating Node Circuits

Figures 6, 7, and 8, show circuits generated for various types of nodes. Each circuit contains a "go" gate that indicates that the node is active in the current state. Except for state nodes, nodes cause no delay, and thus signal that they are done as soon as possible. Circuits for condition nodes route their "go" signal to one of two successor nodes.

The state nodes —generated by `tick` statements— delay execution for one clock cycle. The very first node of each PASM chart is always a state node with no predecessors, we treat it specially (see Figure 9).

Connecting Node Circuits

The node circuits are connected as follows. The output wire named done_*N* of each node, except for a condition node, will be connected to the named *go* gates of its successor nodes.

Generating Expression Circuits

In the SMALL language, there are several kinds of expressions: identifiers, constants, subscripts, subarrays, arrays, and various unary and binary operators. Each expression is transformed into a circuit with the value outputs for the value of the expression and an overflow output that indicates whether overflow occurred in the calculation of the value output. Each circuit expression operates within one clock cycle. The detailed circuit design for each expression can be found in [4].

Output to VHDL Descriptions

We output a description of the netlist in the structural subset of VHDL. The description uses the IEEE std_logic_1164 packages. VHDL input and output ports are generated for each global signal and register, and also for the clock.

4. Translation to FPGA Configurations

The process of translating a netlist described by a VHDL file to FPGA configurations is divided into the following phases.

- Optionally functional simulation of the structural VHDL description is carried out using Synopsys VHDL System Simulation (VSS) or a similar product to verify that the netlist circuit created by the Netlist Generator performs the specified requirements.
- The Synopsys FPGA Compiler (FC) is used for the gate-level synthesis and logic optimization.
- The FPGA configuration data for a specific Xilinx FPGA chip, for example, XC4028EX-3-PG299, is produced by the Design Manager of the Xilinx Alliance Series version 1.4.

For SMALL programs we have tested, the simulation result are the same as results obtained by directly simulating the PASM chart.

5. Conclusion

We have shown a method for implementing an imperative hardware programming language on FPGAs. The translation algorithm from SMALL programs to FPGA configurations is simple. The netlist created by the Netlist Generator is also very simple. Its components consist of only D-type flip-flops and basic gates. All D-type flip-flops use the same clock line. The Xilinx XC4000 series FPGAs were chosen as the initial target technology. Alternate technologies could be used to implement the netlist. VHDL is used for the target language, but the netlist is not dependent on the target language.

Some enhancements are currently being made. They include the enhancement for the SMALL language, reducing translation time and space, and optimizing the netlist circuit.

Acknowledgements

We would like to thank the Natural Sciences and Engineering Research Council for funding, and Mr. Wei Song for his ongoing contributions to the SMALL compiler.

References

1 T. S. Norvell, "SMALL: A Programming Language for State Machine Design," *Canadian Conference for Electrical and Computer Engineering*, 1997.
2 Page, and W. Luk, "Compiling occam into Field-Programmable Gate Arrays," *Field-Programmable Gate Arrays*, eds. W. Moore, and W. Luk, Abingdon EE &CS Books, pp. 271-283, 1991.
3 T. S. Norvell, *Specification, and Change List for the Netlist Generator*, unpublished, 1998.
4 Y. Shen, *Compiling a Synchronous Programming Language into Field Programmable Gate Arrays*, M.Eng Thesis, Faculty of Engineering and Applied Science, Memorial University or Newfoundland, Sept. 1999.

```
global sig inBit: bool
global sig outBit: bool
par
    while true do
         repeat
            outBit ! 0
         until inBit
         tick
         repeat
            outBit ! 1
         until inBit
    od
||
      inBit ! 0      tick
      inBit ! 1      tick
      inBit ! 0      tick
      inBit ! 1      tick
      inBit ! 1      tick
rap
```

Figure 1: An Example SMALL Program

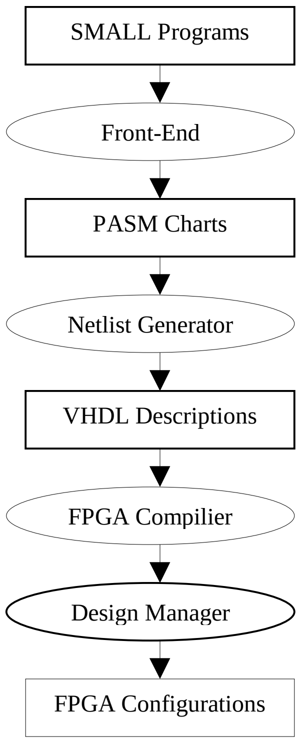


Figure 2: Data Flow Diagram for Translation

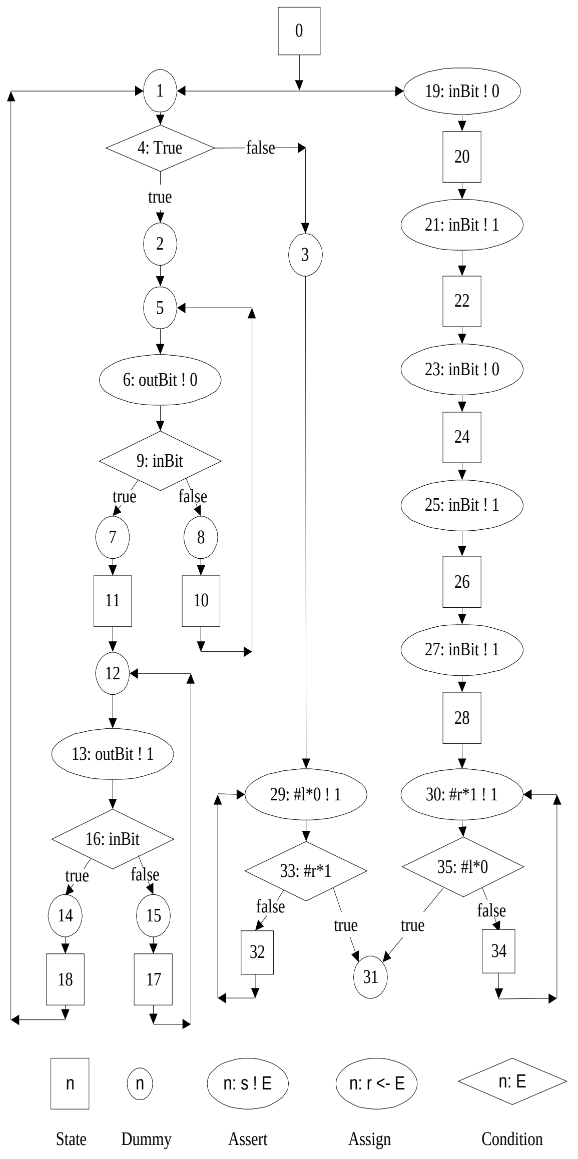


Figure 3 A PASM Chart

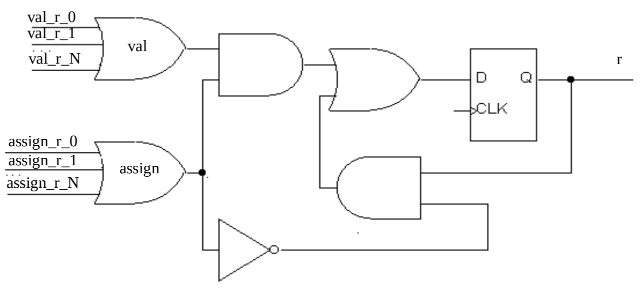


Figure 5. A Local Register Circuit

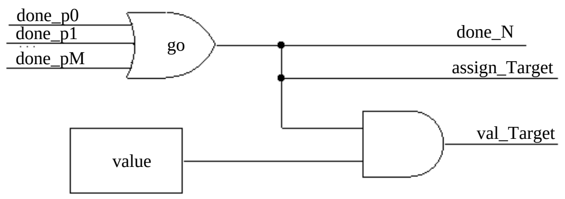


Figure 6. An assignment node circuit

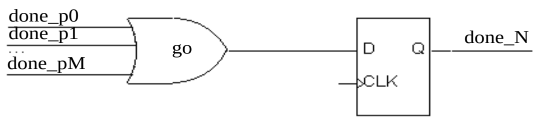


Figure 7. A State Node Circuit

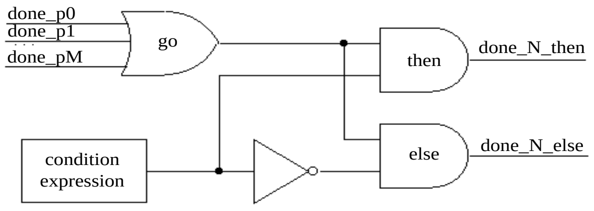
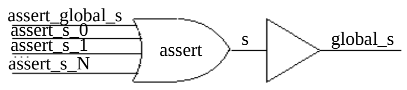


Figure 8. A Condition Node Circuit



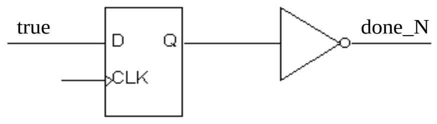


Figure 9 The initial State Node Circuit

Figure 4. A Global Signal Circuit