Automatic Data Path Extraction in Large-Scale Register-Transfer Level Designs

Wei Song, Jim Garside and Doug Edwards
School of Computer Science, the University of Manchester
Manchester M13 9PL United Kingdom
{songw, jdg, doug}@cs.man.ac.uk

***Abstract*—Extracting data paths in large-scale register-transfer level designs has important usage in automatic verification of synchronous circuits and synthesis of asynchronous circuits. Current tools rely on users to provide the data/control partition or use state-space analyses to extract data paths. Due to the explosion of state-space, the latter method can be used in only small designs. To resolve this problem, a graphic search and trim method, which can extract data paths in large scale designs, is presented. A design is first translated into a graphic representation, namely a signal-level data flow graph (DFG), to reveal the connections between signals. By estimating the types (control or data) of these connections, a linear search algorithm can then remove all control-related signals in the graph, which effectively produces a DFG with pure data paths. Results show that this method extracts data paths of large scale designs in seconds.**

I. Introduction

A large-scale digital system can be recognized as a large set of heterogeneous function units dynamically connected, loaded and scheduled by various local and global controllers. These function units and their mutual connections comprise data paths while the resource schedulers, including the local and global controllers, constitute control paths. Data paths are the hardware implementation of the behavioural-level data flow whose correct operation relies on loading them with the right data at the right time, which is scheduled by the control paths.

In behavioural-level synthesis, data flows are usually provided by users or extracted from an input design written in behavioural-level (normally sequential) languages. Data path synthesis [1] maps the flows to function units, explores possible parallel operations and sharing resources among non-overlapped operations. After data and control paths are implemented into register-transfer level (RTL) descriptions, they are blended with no explicit boundary. As a result, extracting data paths from RTL descriptions is actually a reverse engineering process which recovers the behavioural-level data flow buried in the logical-level implementation.

Extracting data paths is not required in normal hardware synthesis flows because RTL designs are mapped into gate-level circuits using logical synthesis, which does not require a clear view of the global data flow. However, it is useful and even important in some other areas. Reuse of intellectual properties (IPs) and legacy codes is one of those areas. Assuming an undocumented soft IP is utilized, an automatic extraction of data flows would be helpful for understanding its behaviour and it is also crucial for automatically verifying its functions [2].

Synthesis of asynchronous systems [3,4] is another area where the extraction of data paths is important. Asynchronous systems [5] is well-known for their low dynamic power and tolerance to delay variations but their development is impeded due to the lack of synthesis tools. Two of the most utilized tool flows: de-synchronization [4] and behavioural synthesis [3,6], require different treatments for data and control paths. Therefore, data paths must be separated from control paths before any circuit optimization. This reveals one of the limitations of de-synchronization. It treats all registers as part of data paths without differentiating the registers used as data buffers from those used as state machines. The generated asynchronous circuit is thus a direct mapping of the original synchronous circuit, leading to large area and potentially low speed.

The usage of automatic extraction pursued by this paper is related to the synthesis of globally asynchronous and locally synchronous (GALS) [7] systems. Given a synchronous RTL design, a tool is being designed to automatically re-partition the design for low dynamic power. If a suitable partition is found, an equivalent GALS system can be achieved by running sub-modules at different clock frequencies and replacing the synchronous buses between sub-modules with asynchronous handshake circuits. Since the boundary of a suitable partition should be located on infrequently utilized data links, an accurate data path annotated with accurate data rates must be extracted beforehand.

To our best knowledge, no method has been proposed for extracting data paths in large-scale RTL designs. Existing synthesis and verification tools either rely on the RTL designers to provide a data flow or extract low-level data flows by analysing the state space [2]. The explosive number of states limits the use of the latter method in large-scale RTL designs. Rather than analysing the state space of low-level data paths, the method proposed in this paper extracts them in a signal-level data flow graph (DFG) derived from the RTL design. Since the arcs in the signal-level DFG are typed into $control$ or $data$, a recursive search algorithm is used to trim all $control$ nodes, which generates a DFG with pure data paths. Although no estimation of data rates is provided yet, it can be done if combined with state analyses of the control paths later in the design cycle.

II. Tool Flow

The proposed tool flow [8] is illustrated in Fig. 1. A Verilog HDL parser has been implemented using Bison and Flex, which is able to read hierarchical RTL designs written

978-1-4799-3432-4/14/$31.00 ©2014 IEEE

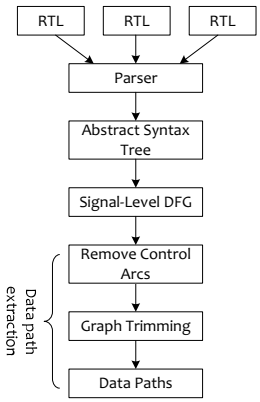

Fig. 1. The tool flow for data path extraction

in multi-file Verilog HDLs complying with the IEEE 1364-2001 standard [9]. After the hierarchical design is successfully elaborated, a graphic representation, namely a signal-level data flow graph (DFG) [10] revealing the connections and relations between all Verilog signals, is then extracted from the parsed abstract syntax tree (AST). Depending on the logical dependence between signals, arcs in the signal-level DFG are annotated with four types: $data$, $control$, $clock$ and $reset$. To extract data paths, all $control$, $clock$ and $reset$ arcs are removed from the signal-level DFG (remove $control$ arcs in Fig. 1). Then using a recursive search algorithm, all control-related nodes and arcs are further removed (graph trimming in Fig. 1), which leaves a DFG with pure data paths.

III. Signal-Level Data Flow Graph

To illustrate the details of signal-level DFGs, a RTL Verilog implementation of the greatest common divisor (GCD) (listed in Fig. 2) is used as an example. It has two flip-flops, *A_Hold* and *B_Hold*, to store the two GCD operands, *A* and *B*. In every cycle, the pair of operands are swapped if *A_Hold* is less than *B_Hold*; otherwise, *A_Hold* stores their difference. When *B* equates zero, the value in *A_Hold* is the greatest common divisor and is output through port *Y*.

A signal-level DFG is a multi-graph revealing the internal connections and relations between signals. The formal definition of a signal-level DFG is described as follow [10]:

Definition 1. *A signal-level DFG is a directed multi-graph denoted by a six-tuple* $DFG = (V, A, T_A, F_A, T_V, F_V)$*, where* V *is a finite set of nodes representing components in the AST;* $A \subseteq V \times V$ *is a finite set of arcs denoting the connection between components;* $T_A \in \{control, data, clock, reset\}$ *is a finite set of available arc types;* $F_A : A \rightarrow T_A$ *is a function mapping types to arcs;* $T_V \in \{seq\_block, combi\_block, i\_port, o\_port, module\}$ *is a finite set of available component types;* $F_V : V \rightarrow T_V$ *is another function mapping the types of all components.*

The corresponding signal-level DFG of the GCD is depicted in Fig. 3. Several steps are taken to translate a RTL Verilog design into a signal-level DFG:

1) Every module is depicted in a separate DFG.

```
module GCD (Clock,Reset,Load,A,B,Done,Y);
   input  Clock,Reset,Load;
   input  [7:0] A,B;
   output Done;
   output [7:0] Y;
   reg A_lessthan_B, Done;
   reg [7:0] A_New, A_Hold, B_Hold, Y;

   always @(posedge Clock)
     if(Reset) begin
        A_Hold = 0; B_Hold = 0;
     end else if(Load) begin
        A_Hold = A; B_Hold = B;
     end else if(A_lessthan_B) begin
        A_Hold = B_Hold;
        B_Hold = A_New;
     end else
        A_Hold = A_New;

   always @(A_Hold or B_Hold)
     if(A_Hold >= B_Hold) begin
        A_lessthan_B = 0;
        A_New = A_Hold - B_Hold;
     end else begin
        A_lessthan_B = 1;
        A_New = A_Hold;
     end

   always @(A_Hold or B_Hold)
     if(B_Hold == 0) begin
        Done = 1; Y = A_Hold;
     end else begin
        Done = 0; Y = 0;
     end
endmodule
```

Fig. 2. Greatest Common Divisor (GCD)

2) Each signal (**wire**, **reg**, **integer**, etc.) is represented as a node in the graph. Depending on the hardware implementation, flip-flops (seq_block) are drawn in rectangles labelled with "FF", combinational signals ($combi\_block$) are drawn in circles, input ports (i_port) are drawn in circles labelled with "I", and output ports (o_port) are circled as well but labelled "O".
3) Each instance ($module$) is drawn in a rectangle labelled "module" (for hierarchical support). It has an internal link pointing to the corresponding DFG of the sub-module.
4) To normalize the hierarchical connections, a dummy node (drawn in a circle) is add for each port (name the original port with a suffix "_P").
5) Traverse all **always** and **assign** statements and connect nodes with arcs annotated with estimated types.

In the final step of generating a signal-level DFG (step 5), a relation tree is generated for each signal to estimate the types of arcs. For example, after traversing the first **always** block, a relation tree is built for signal *A_Hold* as shown in Fig. 4. The tree grows with the level of conditional statements. Each **if** statement is converted into a condition node representing the signals in the conditional expression and two sub-trees denoting the two cases. The arc between the parent node and the condition node is typed $control$. Other conditional statements, such as **case** and **for**, are converted similarly. The leaf nodes of the relation tree are signals appeared in the right-

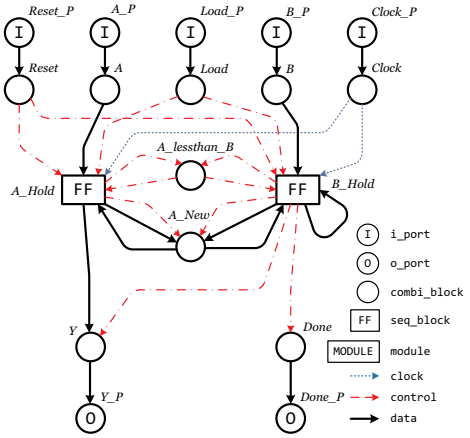

Fig. 3. The signal-level DFG of the GCD module

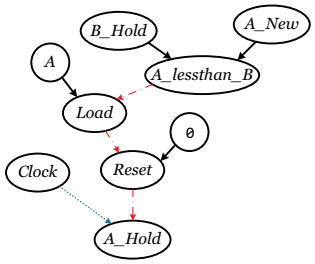

Fig. 4. Relation tree of *A_Hold*

hand side of assignments and are connected using *data* arcs.

After a relation tree is generated for a DFG node, all ingress arcs of in the DFG can be drawn accordingly. For all the non-root and non-constant nodes in the relation tree, an arc is drawn from each of them to the root in the DFG. The type of each arc is set to the type of the single egress arc of its source in the relation tree. In this way, all nodes in a signal-level DFG are connected after all **always** and **assign** are traversed.

The type estimation of arcs in the signal-level DFG is crucial for an accurate extraction of data paths. In general, all signals appeared in the conditional expression of **if**, **case**, **for**, **while** and **?** are connected with a *control* arc, while other non-control signals appeared in the right-hand side of an assignment are connected using *data* arcs. The lack of **else** or **default** in **if** or **case** is automatically checked and a self *data* arc is added when the check results false. Signals in the conditional expression of a ROM style **case** statement are treated as *data* as this sort of **case** statements are usually used in data paths.

IV. Data Path Extraction

The extraction of data paths from the signal-level DFG is done in two steps: removing all *control* arcs and trimming.

Since a pure data path should not contain any control signals, all control-related nodes and arcs should be removed from the signal-level DFG. As the result of the first step, the DFG of the GCD module with all *control* arcs removed is shown in Fig. 5a. Although all remaining arcs are typed *data*, it still contains nodes and arcs that belong to control paths. For example, the combinational node *A_lessthan_B* controls

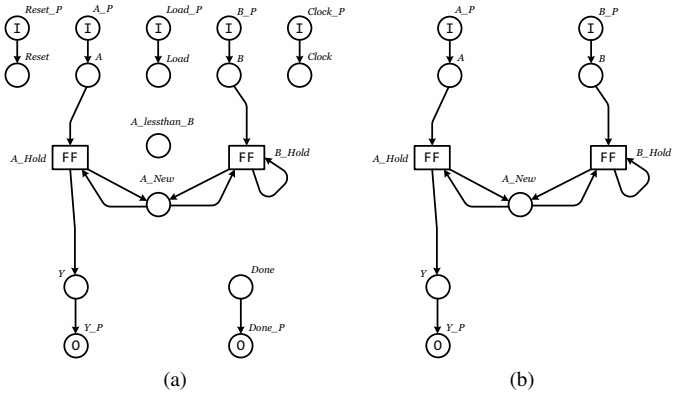

Fig. 5. (A) the signal-level DFG with *control* arcs removed and (b) the extracted data path

the swap of *A_Hold* and *B_Hold*; input port *Load* controls the load of initial values; output port *Done* signals the completion of the GCD calculation. They are all control signals that should be removed as well. Thanks to the removal of all *control* arcs, the nodes and arcs related to *control* signals are disconnected from the data paths, which provides a possibility of trimming them.

In the step of trimming a DFG, a recursive search algorithm is utilized to remove all control-related nodes (along with all ingress and egress arcs) which qualified with the following criteria:

1) A combinational node with either no egress arc or no ingress arc.
2) A register node without a non-self egress arc.
3) An input port without an egress arc.
4) An output port without an ingress arc.
5) A module without an egress arc.

Since the removal of *control* arcs separates control-related nodes from data paths and leaves them dangling, criterion 1 and 2 would trim them from the graph and make the nodes connected to them new dangling nodes. Using a recursive search, all dangling flip-flops and combinational nodes are trimmed. Criterion 3 and 4 removes the control-related I/O ports. Criterion 5 further removes the modules containing pure control circuits. The timing and memory complexity of the recursive search is linear with the total number of arcs in the DFG.

The DFG after trimming is shown in Fig. 5b, where a clean data path is depicted: from the two data inputs, *A* and *B*, through the two computation registers, *A_Hold* and *B_hold*, to the final result port, *Y*. With no user intervention, the extraction method has successfully found the data path and trimmed away all control-related nodes and ports.

V. Test Cases

A. Hierarchical data path

In the previous section, the data path of a single module design, the greatest common divisor, is automatically extracted. The proposed method can handle designs with multi-level hierarchies. To demonstrate this capability, the data path of the permutation module of a SHA-3 encoder [11] is extracted.

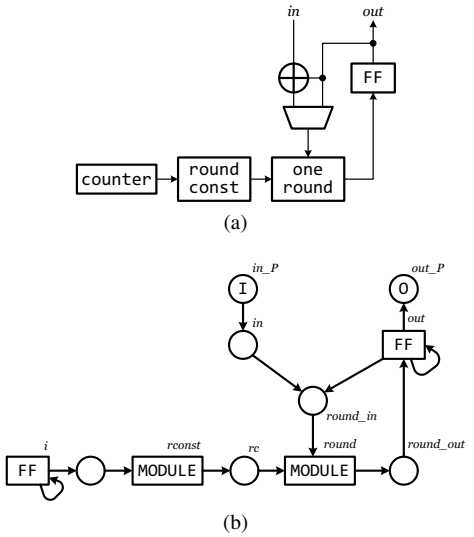

Fig. 6. (A) the block diagram of the permutation module in a SHA-3 encoder and (b) the extracted data path

TABLE I. RESULTS OF DATA PATH EXTRACTION

Design	Signal-level DFG			Extracted data path			Running
	I/O	Module	Signal	I/O	Module	Signal	time (s)
OR1200	52	37	2074	40	33	1142	1
RSD	7	24	1063	3	23	659	<1
NOVA	19	140	7043	9	103	4279	10

The block diagram provided in the user manual and the automatically extracted data path are depicted in Fig. 6a and 6b respectively. In the extracted data path, two sub-modules are drawn in rectangles labelled "MODULE". Registers with no ingress arcs, such as the counter register *i*, are retained as they may produce cyclic data inputs. It is easy to see that the extracted data path complies with the block diagram provided by the author.

B. Large-scale designs

To demonstrate the scalability of the proposed method for large scale circuits, three designs from the OpenCore project repository are chosen as test cases: OR1200, a 5-stage microprocessor [12]; RSD, an industrial standard Reed-Solomon decoder [13]; NOVA, a FPGA proven H.264/AVC baseline decoder [14]. Running the extraction program [8] on an Intel Core™2 Due 3.00 GHz PC with 2 GB memory, Table I reveals the results of the extracted data paths.

For each design, the total number of top-level I/O ports, sub-modules and signals (including wires and registers) are listed for the signal-level DFG and the extracted data path graph respectively. As shown in the figures, the number of signals has reduced significantly in the data path graph, along with reduction in I/O ports. This indicates that the proposed method successfully identifies the control-related signals and I/O ports and simplifies the signal-level DFG with data paths retained. All data path graphs are automatically extracted without user intervention, although a more accurate extraction can be achieved by explicitly specifying the top-level $control$ I/O ports. For all designs, the extraction process finishes in a small number of seconds. As the largest design, NOVA takes the longest time to extract but still the total calculation time is only 10 seconds, demonstrating the scalability of the proposed method.

VI. CONCLUSION

This paper presents a fast and scalable method which can automatically extract data paths for large-scale RTL designs. Using the parsed abstract syntax tree, the method translates the design into a hierarchical signal-level DFG with the connections between signals annotated with estimated $control$ or $data$ types. Using these estimated types, the method is able to remove all control-related signals, modules and ports in the signal-level DFG leaving the graph purely with data paths. Test cases show that the automatic extraction method can successfully identify data paths in RTL designs. Thanks to the use of data flow graphs, the proposed method can extract the data paths of large-scale RTL designs in linear time, which has not been achieved by traditional state-space analytic methods. Utilizing this method, fully automatic tools could be built to verify the functions of large synchronous designs or synthesize them into asynchronous ones.

ACKNOWLEDGEMENT

The authors would like to thank the grant from the Engineering and Physical Sciences Research Council (EP/I038306/1).

REFERENCES

[1] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, Inc., 1994.

[2] I. Ghosh, A. Raghunathan, and N. K. Jha, "A design-for-testability technique for register-transfer level circuits using control/data flow extraction," *IEEE Trans. on CAD*, vol. 17, no. 8, pp. 706–723, 1998.

[3] D. Shang, F. Burns, A. Koelmans, A. Yakovlev, and F. Xia, "Asynchronous system synthesis based on direct mapping using VHDL and Petri nets," *IEE Proceedings – Computers and Digital Techniques*, vol. 151, no. 3, pp. 209–220, 2004.

[4] J. Cortadella, A. Kondratyev, L. Lavagno, and C. P. Sotiriou, "Desynchronization: synthesis of asynchronous circuits from synchronous specifications," *IEEE Trans. on CAD*, vol. 25, no. 10, pp. 1904–1921, October 2006.

[5] J. Sparsø and S. Furber, *Principles of Asynchronous Circuit Design — A Systems Perspective*. Kluwer Academic Publishers, Boston, U.S.A, 2001.

[6] D. Edwards and A. Bardsley, "Balsa: an asynchronous hardware synthesis language," *The Computer Journal*, vol. 45, no. 1, pp. 12–18, 2002.

[7] M. Krstić, E. Grass, F. K. Gürkaynak, and P. Vivet, "Globally asynchronous, locally synchronous circuits: overview and outlook," *IEEE Design and Test of Computers*, vol. 24, no. 5, pp. 430–441, 2007.

[8] W. Song. (2013) An asynchronous verilog synthesis (AVS) system. [Online]. Available: https://github.com/wsong83/Asynchronous-Verilog-Synthesiser

[9] IEEE Computer Society, *IEEE Standard Verilog® Hardware Description Language*, September 2001.

[10] W. Song and J. Garside, "Automatic controller detection for large scale RTL designs," in *Proc. of DSD*, September 2013, pp. 844–851.

[11] H. Hsing. (2012) SHA3 (KECCAK). [Online]. Available: http://opencores.org/project,sha3

[12] OpenRISC Community. (2009) Or1200 openrisc processor. [Online]. Available: http://opencores.org/or1k/OR1200_OpenRISC_Processor

[13] Varkon Semiconductors. (2010) Reed solomon decoder. [Online]. Available: http://opencores.org/project,reed_solomon_decoder

[14] K. Xu. (2009) H.264/avc baseline decoder. [Online]. Available: http://opencores.org/project,nova