Automatic Controller Detection for Large Scale RTL Designs

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

***Abstract*—Automatic detection of the finite state machines (FSMs) in a register transfer level (RTL) design is a widely utilised technique in logical synthesis for optimised FSM implementation and in hardware verification for the fast coverage of the control circuit. It is believed that FSM detection can also be used to explore the potential system partitions. Chosen an optimal partition, a large scale synchronous RTL system can be automatically converted into energy efficient globally asynchronous and locally synchronous (GALS) systems. A new FSM detection algorithm is presented providing a full coverage of all FSM-like controllers. It uses several criteria to detect FSMs on a register level abstracted graph generated from the RTL design. It is the first FSM detection algorithm that provides full FSM detection in the granularity level of signals without any restrictions on coding styles.**

I. Introduction

Register transfer level (RTL) hardware description languages (HDLs) [1] and logical synthesis techniques [2] have been the driving strength of the continuous development in synchronous very large scale integrated circuits (VLSIs) for several decades. In VLSI designs described in RTL, finite state machines (FSMs) are an important type of building blocks. They control the sequences of certain otherwise parallel operations and enforce the communication protocols between parallel modules.

The automatic detection and recognition of FSMs is an old topic that has been researched since the broad adoption of automatic logical synthesis. In the process of logical synthesis, the detected FSMs are analysed for their state space and output loads [2]. According to the state transition pattern and the load for each state, different encoding schemes can be used to optimise the implementation of an FSM for area or speed benefits. In recent years, FSM analysis also finds its usage in hardware verification. Since the size of a hardware system increases significantly with the shrinking device geometry, which leads to the exponential state space growth [3], verifying the system as a whole becomes extremely difficult if it is still possible. A way to cope with this problem is to verify the control portion of the system separately, where the automatic FSM detection and the state space analysis are among the crucial issues.

The usage of FSM detection in this paper is however different. The state spaces and the connections of the FSMs, which are automatically detected, are used to reveal suitable system partitions. Re-implementing the global communication of a chosen partition in asynchronous circuit, a software would be able to automatically convert a synchronous RTL system into a globally asynchronous and locally synchronous (GALS) system [4]. It is well known that asynchronous circuits can be used to reduce energy consumption and resolve the global clock issues [4]–[6]. However, the current design flow is to manually partition the system, and then design the global asynchronous communication circuits by hands or using special asynchronous design tools, both of which are intimidating tasks to traditional synchronous hardware engineers.

To handle this problem, the approach pursued in this research is to automatically identify suitable system partitions and afterwards replace them with global asynchronous communication circuits. The first step of this two step process is the more difficult one. A partition divides a system into a number of loosely related blocks which talk with each other using channels having variable data rates. Since these channels must be controlled by FSMs (or counters), detecting these FSMs is crucial to the identification of suitable boundaries. Since a software has to exhaust all potential partitions before choosing an optimal one, all FSMs, including the counters used as controllers, must be detected. This full coverage is the major difference between our FSM detection algorithm and the algorithms used for logical synthesis or hardware verification, where missing some FSMs does not compromise the functionality of the hardware. This is also the reason why the paper is titled as controller detection instead of FSM detection.

II. Previous Work

Throughout literature, there are three types of FSM detection techniques: matching of certain code styles, program slicing and pattern recognition.

Current commercial synthesis tools [7] are the typical examples of using code styles to detect FSMs. According to the Verilog user guide provided by the Synopsys HDL Compiler™, an FSM must be written as a register assigned only with predefined values, used only in **case**, **if**, **==** and **!=** statements (expressions), and never used as ports. These strict restrictions match only the FSMs written in the standard form of one or two **always** blocks. Hierarchical FSMs and the counters used as controllers are rejected by the matching

This work is supported by EPSRC Grant “Globally Asynchronous Elastic Logic Synthesis (GAELS)” (EP/I038306/1).

algorithm. However, this is not a serious problem for synthesis tools because overlooking an FSM results to only sub-optimised control circuits.

Program slicing is another effective FSM extracting technique provided that the names of the controllers of interest are foreknown and the code is not parsed. It works on the source code. Given a set of signals, the program slicing algorithm extracts all the codes related to the given signals through a line by line scan of the source code. The result is a reduced source code which contains only the lines related to the given signals. When the given signals are FSMs, the result would be the sub-circuit describing and utilising these FSMs. This technique was first used in software projects to savage useful code segments [8]. Later it was adopted in hardware languages [9] to extract the control paths from a large design. However, it is not a fully automatic process as users are required to identify the names of the FSMs before slicing the source code. In other words, it is more of an effective program extractor rather than an FSM detector.

Pattern recognition is the only technique possible to automatically recognise and detect all FSMs without strict restrictions on coding styles. It applies several matching criteria to certain abstracted graphic representations of parsed designs. If a node matches the pattern defined by the criteria, it is recognised as an FSM. As the graph is an abstracted representation, this technique does not depend on any coding styles and it can be applied to designs described by any hardware languages. Using the process-module graph, an algorithm [3,10] has been able to report all the processes (**always** blocks in Verilog) that contain FSMs.

The detection algorithm proposed in this paper belongs to the category of pattern recognition techniques. Compared with the work in [3], this paper is able to improve the granularity from **always** block to signals. In this way, the new algorithm can recognise the FSMs buried inside the **always** blocks having non-FSM signals. Furthermore, design hierarchy is flattened and broken into signal level graphs. The new algorithm has no requirements on the content of sub-modules, which is however a must in [3]. In summary, this is the first FSM detection algorithm providing the full coverage of FSMs with the granularity level of signals.

III. FSM Recognition Criteria

A synchronous circuit can be understood as a network of registers (mostly flip-flops) connected by combinational circuits. Described in register transfer level (RTL) hardware description languages (HDLs), such as the Verilog HDL language [1], the registers are grouped into multi-bit signals, the combinational circuits are described by abstracted statements and the whole circuit is partitioned into hierarchical modules. In other words, synchronous circuits described in RTL designs reveal a huge amount of abstract information, including system partitions, wire groups, control/data division, control/data flow, logical operations, etc. Extracting this information, a software is possible to automatically understand the behaviour of an RTL design. This paper tries to use this information to detect all FSMs.

Using the abstract information, an RTL design can be converted into a register relation graph (RRG), where registers are represented by nodes and combinational circuits are depicted as directed arcs. These arcs are classified into four different types, including control, data, clock and reset, depending on the relations between sources and targets. An FSM is a control register used to control the modification of other registers, while its own value follows a dynamic path in a predefined finite state space. Each state transition occurring on this path is affected by the status of the environment at the transition. According to the behaviour of FSMs, a pattern in the RRG can be defined and used to detect them. Followings are the proposed criteria for FSM detection:

Definition 1. *Let a register be an FSM, at least one of its output paths is a self-loop path which does not go through higher hierarchies.*

Since the transition of an FSM is restricted by its predefined state space, each transition always and must transit from a known previous state to a new state depending on the status of its environment. As a result, an FSM must have a self-loop to identify the previous state. The extra hierarchical constraint is set for practical concerns. In normal RTL designs, the calculation of the next state is located in the same module containing the FSM or sub-modules inside the FSM module, which complies with the constraint. However, some RTL designs have manual test chains or combinational loops [11] which may confuse the FSM detection algorithm. The hierarchical constraint is added to prevent any global combinantional loop from making a false self-loop to a non-FSM register.

Definition 2. *Let a register be an FSM, at least one of its output paths is a control path towards another register.*

The value of an FSM register must has been used to control the modification of other registers; otherwise, this register is a non-control one on data paths.

Definition 3. *Let a register be an FSM, all its input data comes from self-loop paths or constant numbers.*

This criterion is inferred from the finite state space limitation of FSMs. Although the state transitions of FSMs depend on their environment, the value set of each FSM is predefined. Therefore, the data inputs of an FSM come from only itself or a predefined constant value.

Some typical patterns in RRGs are shown in Fig. 1 where the operation and the type of each arc is labelled as "(operation)/*type*". Fig. 1a shows the register connection pattern of a general program counter (*pc*) in a microprocessor. Although it has a self-loop for address increment and it controls the next value of the *instruction* register, according to Definition 3, it is not recognised as an FSM due to its data input from *instruction*. A jump instruction can modify the value of *pc*, which makes the state space of *pc* infinite. The accumulator (*acc*) depicted in Fig. 1b is not an FSM either no matter

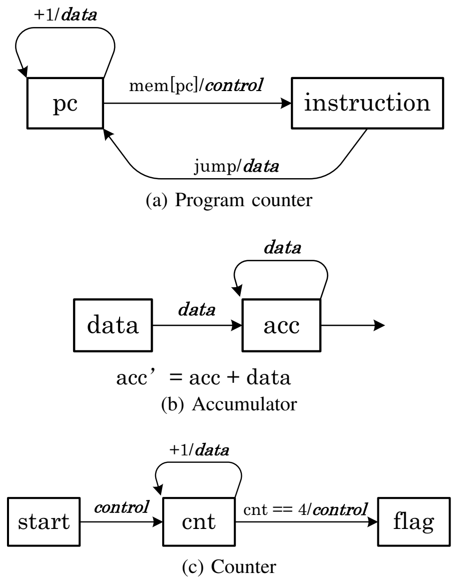


Fig. 1: Examples of some typical registers

whether it has control output paths. For the same reason as *pc*, it has an infinite state space due to its data input from *data*.

Fig. 1c shows a programmable counter (*cnt*), which is not recognised as an FSM by commercial synthesis tools [7], but accepted as an FSM in this paper as long as it has control outputs. Since the counter has no data input other than the self-loop, its value range is finite so as its state space. If this counter is used to control other registers, it should be considered as an FSM although it does not match the normal FSM coding style.

IV. Register Relation Graph

Fig. 1 has shown several examples of the register relation graph (RRG). The formal definition of an RRG can be described as follow:

Definition 4. *A register relation graph is a directed multi-graph denoted by a quadruple* $RRG = (V, A, T_A, F_A)$*, where*
V *is a finite set of nodes representing registers and ports;*
$A \subseteq V \times V$ *is a finite set of arcs denoting the combinational paths between nodes;*
$T_A \in \{control, data, clock, reset\}$ *is a finite set of available arc types;*
$F_A : A \to T_A$ *is a function mapping types to arcs.*

The software flow of detecting FSMs is illustrated in Fig. 2. Assuming a multi-file hierarchical RTL design is parsed into an abstract syntax tree (AST) [2], a signal level data flow graph (DFG) can be extracted from the AST to reveal the hierarchical signal relations among all registers and combinational units. Generated from the signal level DFG, the RRG is a flattened register relation graph which has no hierarchy or combinational blocks. The paths between all registers in the signal level DFG are iterated and reduced to arcs in the RRG. Applying the FSM detection criteria on the RRG and the signal level DFG, a group of FSM registers are recognised.

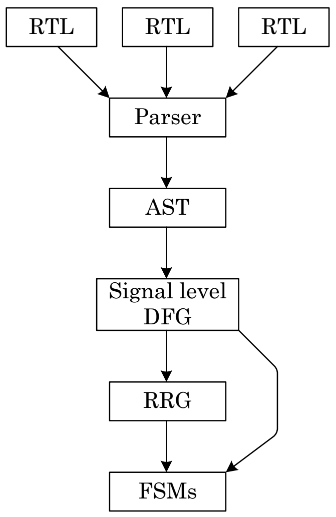


Fig. 2: Software flow of detecting FSMs

A. Signal level data flow graph

Definition 5. *A signal level data flow graph 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 the parallel 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;*
$F_A : A \to T_A$ *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;*
$F_V : V \to T_V$ *is another function mapping the types of all components.*

The signal level DFG is a hierarchical graph, where a node can be a sub-module ($module$) storing the port mapping to a sub-DFG, a sliced **always** block (seq_block or $combi\_block$) containing the statements related to a single signal, a sliced **assign** statement ($combi\_block$) assigning a single signal, an input port (i_port), or an output port (o_port). The arcs between nodes have the same type definition as in RRGs but they describe detailed connections between AST components rather than registers.

As an example, Fig. 3 lists a traffic light controller with its signal level DFG drawn in Fig. 4. The traffic light switches between red and green for every 50 seconds. A 3-second yellow is added when switching from red to green while another 5-second one is inserted after green. The controller design has a main FSM (*state*) to control the colour. It is described in a standard two-block structure where the next state is described in a combinational **always** block. A programmable counter (*cnt*) is used to count the remaining time of each colour.

```
module traffic(clk, rstn, red, green, yellow);
   parameter R = 0;  // red state
   parameter YR = 1; // yellow state after red
   parameter G = 2;  // green state
   parameter YG = 3; // yellow state after green
   input  clk, rstn;
   output red, green, yellow; // light control
   reg [1:0] state;      // state machine
   reg [1:0] state_nxt; // next state
   reg [5:0] cnt;        // second counter

   always @(posedge clk or negedge rstn)
     if(~rstn)
       state <= R;
     else
       state <= state_nxt;

   always @(state or cnt) // next state
     if(cnt == 0)
       case(state)
         R:  state_nxt = YR;
         YR: state_nxt = G;
         G:  state_nxt = YG;
         default:
           state_nxt = R;
       endcase // case (state)
     else
       state_nxt = state;

   always @(posedge clk or negedge rstn)
     if(~rstn)
       cnt <= 0;
     else if(cnt == 0)
       case(state)
         R:  cnt <= 2;
         YR: cnt <= 49;
         G:  cnt <= 4;
         default:
           cnt <= 49;
       endcase // case (state)
     else
       cnt <= cnt - 1;

   assign red = state == R ? 1 : 0;
   assign green = state == G ? 1 : 0;
   assign yellow =
     (state == YR || state == YG) ? 1 : 0;
endmodule
```

Fig. 3: Traffic light controller

The signal level DFG shown in Fig. 4 is generated from the AST by an automatic process. It reflects the relations between the components in the Verilog source code.

All sequential **always** blocks (*seq_block*) are depicted as rectangles labelled "FF". Continuous assignments and combinational **always** blocks are denoted by blank circles. Input and output ports are represented by circles labelled with "I" and "O" respectively. If there is a module entity, it would be drawn as a rectangle labelled "Module". The corresponding signal names are noted besides the nodes. Arcs are used to demonstrate the relations between nodes. If the value of a signal is determined by another signal through an assignment, such as *state_nxt* assigns the value of *state* in the first **always** block, a *data* arc is used to link them together, just as the bold black arrow from *state_nxt* to *state*. All the signals appeared in

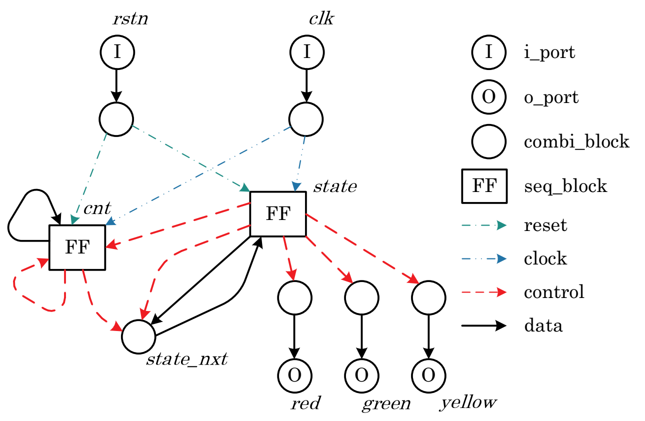


Fig. 4: Signal level DFG of the traffic light controller

the condition statement of **if**, **case**, **for**, **while** and the condition operator "**?:**" are considered as control signals affecting the assignments of the node where they are used. In the traffic light controller, since *state* is used in both **case** statements as the case condition, two corresponding *control* arcs (a red dash arrow) is drawn from *state* to *cnt* and *state_nxt* respectively. The arcs for *reset* and *clock* are depicted in different arrows as well.

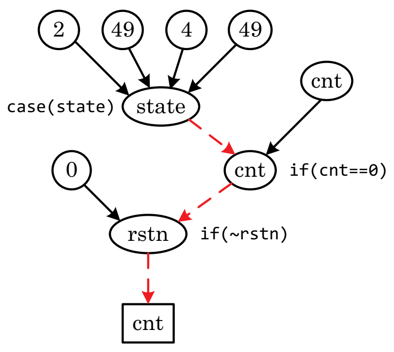
This representation of different nodes and arcs in Fig. 4 will be inherited in all succeeding graphs for easy understanding.

Due to the automatic process, the signal level DFG has some special issues deserving mentioning. The signals appeared in range selector "**[]**" are treated as control signals as variable range expressions are normally used the control of a multiplexer. A dummy combinational node (*combi_block*) is inserted on the inner side of each input/output port (*i_port* or *o_port*). This ensures that the ingress degree of an *i_port* and the egress degree of an *o_port* is always one, which is purely required by the cross-hierarchy arc searching algorithm for simplicity reasons.

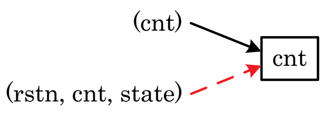
B. Block analysis

Generating the signal level DFG from the AST is an automatic process. A DFG is drawn for each **module** during the process. The design hierarchy is stored in the *module* node generated for each module entity and the port mapping inside the *module* node. At the beginning, a DFG is a graph containing unconnected nodes representing all the components in a single **module**. A scanning algorithm named block analysis is used to scan every **always** block and continuous assignment to get a pair of signal sets, *control* and *data*, for each node in the DFG. Using these two signal sets, corresponding arcs can be drawn in the DFG, which finishes the process.

If an **always** block or a continuous assignment assigns the values of multiple signals, it must be sliced into multiple nodes in the DFG before block analysis. It is possible to use program slicing techniques [8,9] to slice the RTL source codes before parsing them but this is done in the AST in this research, since our parser supports the compilation time unfolding (**generate**



(a)



(b)

Fig. 5: Relation tree for *cnt* block

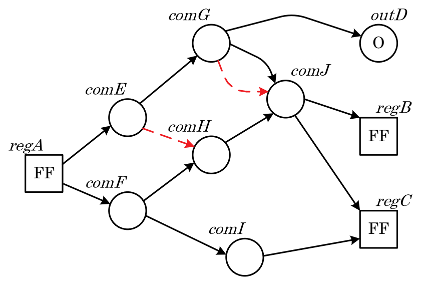
blocks [1]) features which cannot be done before semantic analyses.

Taking the **always** block assigning *cnt* in the traffic light controller for an example, the block analysis would first generate a relation tree as shown in Fig. 5a from the AST. Then this relation tree is used to obtain the pair of signal sets.

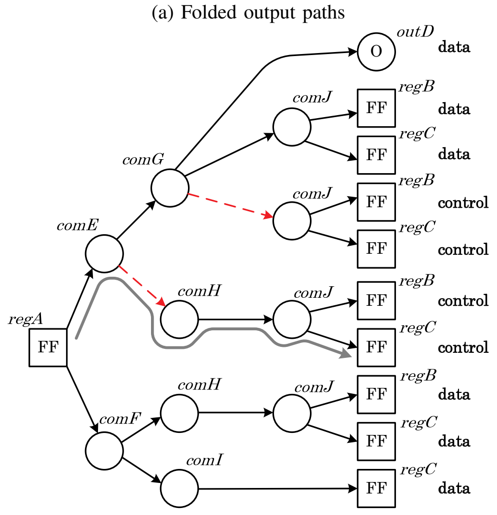
The block analysis is a traversal over the AST. The target signal being assigned by this AST (*cnt* in this case) is set as the root at the beginning. During the traversal, if a statement is a conditional one (**if**, **case**, **for** or **while**), it is converted into a control node (linked with its parent using a $control$ arc) containing all the signals in the condition expression. Therefore the signals *rstn* and *cnt* are put into control nodes due to the **if** statements and the signal *state* is also placed in a control node due to the **case** statement. All control nodes are linked to their parents with $control$ arcs.

If the statement is an assignment, it is converted into a data node (linked with its parent using a $data$ arc) containing all data signals and constant values in the right-hand side expression. In Fig. 5a, four data nodes containing constant values (2, 49, 4 and 49) are linked with the node *state* representing the four branches of the **case** statement. When the right-hand side expression of an assignment has control signals (the signals in **[]** and **?:** operations), they are put in a parallel control node linked to the same parent of the data node. In both cases, the nodes of an assignment statement are leaf nodes. When a condition statement does not have a default branch (**if** without **else** or **case** without **default**), an extra data node containing the target signal is added.

Once the relation tree is done, all the signals in the tree are put into two signal sets as shown in Fig. 5b. The signals in control nodes are put into the control set and the signals in data nodes are put into the data set. Constant values are ignored as they are not drawn in the signal level DFG. Some signals (*cnt*) may appear in both sets, indicating that they are used as both data and control.



(a) Folded output paths



(b) Unfolded output paths

Fig. 6: Path unfolding

Using these sets, the nodes in the signal level DFG are connected. For each signal in the data set, a data arc is added in the signal level DFG. Similarly, a control arc is added for each signal in the control set. As a result shown in the signal level DFG (Fig. 4), node *cnt* has a $data$ input arc from itself, and three $control$ input arcs from *state*, *rstn* (which is later replaced with a $reset$ arc) and, again, itself. When the **always** block is an edge triggered one, the signal in the sensitive list but not in the control set is the clock signal, while the one in both is the reset signal and its arc type is changed to $reset$.

C. Path iteration

A path in an RTL design is a combinational link from a register or an input port to another register or an output port. In an RRG, it is represented as an arc. The process of generating an RRG from a signal level DFG is to eliminate all combinational nodes and the hierarchy. The key step in this process is to iterate all the output paths of a register or an input port along with their types in the signal level DFG, and then connect all the ending registers or output ports of these paths with arcs having the same types in the RRG.

Assuming the output paths of a register named *regA* have the signal level DFG shown in Fig. 6a. This is a tree having *regA* as the root and it is folded because some intermediate nodes have more than one path from the root. To iterate all paths from the root to all leaves, the tree has to be unfolded into an expanded tree shown in Fig. 6b. The number of paths can increase exponentially with the growing depth of the folded tree in the worst cases.

Besides exploring all leaf nodes, the other task of the path iteration is to calculate the type of each path. In the unfolded tree shown in Fig. 6b, the type of a path is a compilation of the arc types through the way. If all the arcs have the same type, such as the path from *regA* to *outD*, obviously the path type is the same type of all the arcs. However, if the arc types are mixed, the path type depends on the priorities of different arc types. In this paper, $control$ has a higher priority than $data$. Considering the path from *regA* to *regC* through arc *comE*→*comH* (highlighted with a grey line in Fig. 6b), since *comE* controls the value of *comH*, *comE* actually controls the value of all downstream nodes towards *regC* and the type of this path should be $control$. The compiled type for each path of the unfolded tree is shown besides the leaf in Fig. 6b.

Iterating the unfolded tree is time consuming as it takes exponential time to finish. As an example, the RRG generation for the OR1200 microprocessor (124 register signals) [11] takes less than 2 seconds while the same process for the H.264/AVC baseline decoder (855 register signals) [12] cannot finish in 10 minutes.

To resolve this problem, it is found that software caches (dynamic programming [2]) can be used to obtain the types of all paths without unfolding the tree. This is possible based on two observations: one is that an arc in an RRG is specified by the source node, the target node and the arc type, while the intermediate nodes are omitted. There is no need to travel both paths if they end at the same leaf node and have the same type. The other one is that the type of a path can be compiled from its disjunctive sub-paths. If the type of a sub-path is known, there is no need to travel it. As a result, if a sub-tree is shared by many paths in the unfolded tree, it is possible to store these types and leaf nodes in a software cache, and use this information to compile the types when the sub-tree needs to be visited again.

The recursive path iteration algorithm using the aforementioned software cache is presented in Fig. 7. The software cache is a two-layered map named `rmap`. The first layer stores a map for each node in the folded tree. The second layer stores all the reachable leaves of an intermediate node and the types of the sub-paths between the intermediate node and the leaves. Written in the C++ template fashion, `ramp` is defined as `map<Node, map<Node, TYPE>>` where `Node` denotes the data structure storing a node and `TYPE` represents a path type. Using this cache, an algorithm can obtain the types of all leaves by traversing the folded rather than the unfolded tree.

Provided with a root node `R` (a register or an input port), function `getOutPaths()` is the function returning the path types of all connected leaves (registers or output ports) in the form of a path map (`map<Node, TYPE>`). This function implements a depth-first search (DFS) to traverse all the nodes in the folded tree using a recursive sub-function

```
// return the output paths of a register node R
map<Node, TYPE> getOutPaths(Node R) {
  // node relation map
  map<Node, map<Node, TYPE>> rmap;
  // the paths to be returned
  map<Node, TYPE> paths;
  for each output node N of R {
    rmapUpdate(R, N, rmap);    // traverse
  }
  return rmap[R];
}

// rmap update function
void rmapUpdate(Node P,   // parent node
                Node N,   // this node
                map<Node, map<Node, TYPE>> rmap
               ) {
  if N is a register or a top level output {
    // end point, update the type in rmap
    rmap[P][N] |= type(P,N);
  } else {
    if rmap[N] existed { // visited
      for each pair<Node, TYPE> (M,t) in rmap[N] {
        // update the type
        rmap[P][M] |= type(P,N) + t;
      }
    } else { // new node
      for each output node M of N
       rmapUpdate(N, M, rmap);    // traverse
      }
    }
  }
}
```

Fig. 7: Output path iteration algorithm

`rmapUpdate()`. When the DFS is done, the map stored for the root node `R` in the cache (`rmap[R]`) is the path map expected.

The sub-function `rmapUpdate()` handles the search for each node. It takes the parent node `P`, the current node `N` and the cache `rmap` as input arguments. If the current node is a leaf, the type of the arc `P`→`N` (`type(P,N)`) is obtained and stored at cache position `rmap[P][N]`. If a map is already stored for the current node in the cache, its sub-tree must have been fully searched according to the DFS algorithm. In this case, the map `rmap[N]` stores the types of all paths starting from `N` and can be used to generate a new map (`rmap[P][M]`) recording the types of all sub-paths starting from `P` and through `N` using type addition (+). A full search is needed only when the current node is not visited yet.

The type addition operation follows the truth table defined in Table I. The path type `TYPE` is a 4-bit vector $(clock, reset, control, data)$ denoting the four possible types. Multiple bits may be set in `TYPE` if multiple paths with different types co-exist between a pair of nodes. Table I shows only the operations for sub-type $(control, data)$ as the special type $control$ has a higher priority than $data$. When the upstream sub-path `A` or the downstream sub-path `B` is purely $control$, the result type is $control$. For all other cases and for the sub-type $(clock, reset)$, addition is equivalent to binary OR.

Utilising the output path iteration algorithm to all registers

TABLE I: (A + B) for sub-type $(control, data)$

		B			
		00	01	10	11
A	00	00	01	10	11
	01	01	01	10	11
	10	10	10	10	10
	11	11	11	10	11

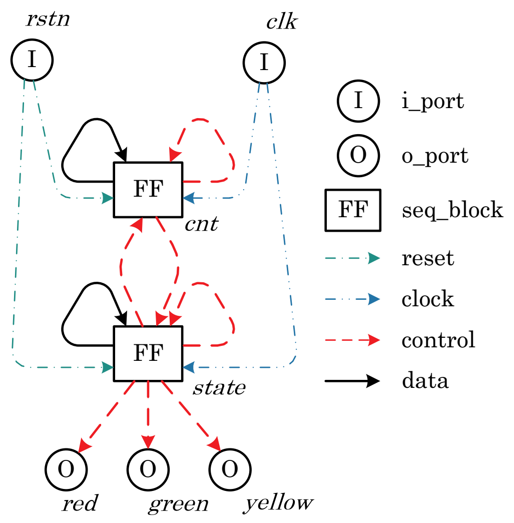


Fig. 8: RRG of the traffic light controller

and top level input ports of a signal level DFG, an RRG can be produced. The RRG of the traffic light controller is depicted in Fig. 8. The *combi_block* node *state_nxt* and the dummy nodes for all ports are reduced to arcs with different types. The only non-port nodes left in the RRG are the two registers *state* and *cnt*.

V. FSM DETECTION

The FSM detection algorithm utilises the three criteria defined in Section III to detect FSMs. The pseudo-algorithm is described in Fig. 9. It reads in the RRG `G` and the signal level DFG `DG` and returns a set of FSMs. There are two internal loops: The first one uses Definition 1 to find all potential FSM registers with self-loops in or below their hierarchies in the signal level DFG. Similar to `getOutPaths()`, function `getLoopPaths()` returns all self-loops that do not go through higher hierarchies using a software cache.

The second loop removes all fake FSMs from the potential set according to Definition 2 and 3. Function `getControlOutPaths()` returns all the control arcs starting from `R` in the RRG `G`. If this set is empty, register `R` has no control output path and is discarded. Similarly, function `getDataInPaths()` returns all the none-loop data arcs towards node `R` in `G`. If this set is not empty, register `R` has input data paths from other registers and is removed as well.

Function `getLoopPaths()` and `getOutPaths()` are the most time consuming procedures. Thanks to the software cache, the average time complexity of both algorithms is $O(MN)$ where M is the number of registers in the RTL

```
set<Node> getFSMs(RRG G, DFG DG) {
  set<Node> FSMs;  // set of FSM registers

  // find out all registers with self-loops
  for each register R in G {
    if (empty != getLoopPaths(R, DG))
      FSMs.insert(R);
  }

  // remove fake FSMs
  for each register R in FSMs {
    // remove registers with no control outputs
    if (empty == getControlOutPaths(R, G))
      FSMs.erase(R);
    // remove registers with non-self data inputs
    if (empty != getDataInPaths(R, G)))
      FSMs.erase(R);
  }

  return FSMs;
}
```

Fig. 9: FSM detection algorithm

design and N is the average number of the arcs in the folded tree shown in Fig. 6a. The maximum area overhead of the cache is $O(N^2)$ where N is the number of nodes in the largest folded tree.

All algorithms described in this paper have been implemented using C++. The final software [13] has a full featured Verilog HDL parser recognising all synthesisable language features.

VI. CASE STUDY

The FSM detection algorithm has been utilised to detect FSMs in three large scale RTL designs chosen from the OpenCores® project repository.

The first design is the well-known OR1200 OpenRISC microprocessor [11] which is a 32-bit 5-stage RISC processor. The program counter is one of the difficulties in this design. It controls the behaviour of the processor but is not an FSM. The other problem is the register forwarding loop and the debugging unit, which together cause a combinational control loop through the current instruction register. Due to this loop, a small number of data registers are mistakenly recognised as FSMs. It is difficult to accurately identify the true FSMs in this design.

The second design is a Reed-Solomon decoder [14]. Although the design is provided as an industrial standard hardware IP, the coding style is actually ad hoc. Control registers are frequently written in an **always** block assigning other non-control registers. Complicated expressions are used in variable range expressions, which further blurs the boundary between control and data. Traditional FSM detection algorithms would fail in such designs because their granularity levels are not small enough [3] or because the coding style of this design is not standard [7]. An accurate FSM detection for this design relies deeply on the correct slicing of its bulky **always** blocks.

TABLE II: Results of the FSM detection for different test cases

Design	DFG Nodes	Registers	Time	FSMs		Rate	Types			
				Reported	Verified		FSM	Counter	Bit	Fake
OR1200	2074	124	$< 1s$	19	17	89%	7	5	5	2
Reed-Solomon	1063	325	$2.0s$	56	54	96%	6	36	12	2
H.264/AVC	7043	855	$7.1s$	55	49	89%	13	30	6	6

The last design is an ASIC verified H.264/AVC baseline decoder, which has the largest gate count (around 196K) in all chosen cases. Used as global pace synchronisers, several counters have large numbers of output paths. One of these counters is connected to more than 400 registers leading to an unfolded relation tree with around 280K leaves. This is the case chosen to examine the time efficiency of the cache-based path iteration algorithm.

Running the FSM detection algorithm on an Intel Core™2 Due 3.00 GHz PC with 2 GB memory, Table II reveals the results of the proposed FSM detection algorithm. The number of nodes in the signal level DFG and the number of registers are listed in the first two columns to demonstrate the non-trivial scales of all designs. The third column shows the running time of the detection algorithm (including the time for path iteration). The detection finishes in around 7 seconds for the 196K gate design of the H.264/AVC baseline decoder, which verifies its speed efficiency for industrial scale designs.

Starting from the fourth column, Table II reveals the details of the FSM reports. All reported FSMs and the fake FSMs discarded by the detection algorithm are manually verified off-line. The number of FSMs reported by the detection algorithm is shown in column "reported FSMs" while the true FSMs verified by hands is counted in column "verified FSMs" with the success rate shown in column "rate". There are two types of errors for any detection algorithms: false positive errors (missing true FSMs) and false negative errors (recognising fake FSMs). The manual verification shows that the FSM detection algorithm has no false positive error (therefore it is not listed in Table II) but has false negative errors, which is denoted by the success rate.

The manual verification reveals that the reported FSMs can be sorted into four categories: "FSM", the traditional FSMs recognisable to synthesis tools; "counter", the fixed range counters used as controllers; "bit", the 1-bit control flags; and "fake", the data registers mistakenly recognised as FSMs. Only pattern recognition techniques, including this research and [3], recognise the controllers of types "counter" and "bit".

False negative errors occur for different reasons. In the OR1200 microprocessor, all the two errors are caused by the global combinational loop, which confuses the path type calculation. In the Reed-Solomon decoder and the H.264/AVC baseline decoder, the mistakenly reported FSMs are used in range expressions which actually belong to data paths. Since variable range expressions can be used as table indices (normally on data paths) or data selectors (normally control), it is difficult to tell their types without some help from designers. Nevertheless, our FSM detection algorithm is still considerably more effective than other existing techniques because it can automatically detect all FSMs in the granularity level of signals with only a small number of false negative errors.

VII. Conclusion

This paper presents a new FSM detection algorithm for large scale RTL designs. A design is converted to a signal level DFG revealing the connections between hardware components and an RRG showing the relation between registers. Applying three detection criteria on the signal level DFG and the RRG, the FSM detection algorithm is able to recognise all FSMs with a small number of false negative errors. The detection accuracy and speed efficiency are proved in three large scale projects. This is the first pattern recognition algorithm that is able to detect all FSMs in the granularity level of signals.

References

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

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

[3] C.-N. J. Liu and J.-Y. Jou, "An automatic controller extractor for HDL descriptions at the RTL," *IEEE Design & Test of Computers*, vol. 17, no. 3, pp. 72–77, 2000.

[4] 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.

[5] L. A. Plana, S. B. Furber, S. Temple, M. Khan, Y. Shi, J. Wu, and S. Yang, "A globally asynchronous, locally synchronous infrastructure for a massively-parallel multiprocessor," *IEEE Design & Test of Computers*, vol. 24, no. 5, pp. 454 – 463, 2007.

[6] F. Clermidy, C. Bernard, R. Lemaire, J. Martin, I. Miro-Panades, Y. Thonnart, P. Vivet, and N. Wehn, "A 477mW NoC-based digital baseband for MIMO 4G SDR," in *International Solid-State Circuits Conference, Digest of Technical Papers*, 2010, pp. 278–279.

[7] Synopsys, Inc., *HDL Compiler™ for Verilog User Guide – Version G-2012.06*, June 2012.

[8] F. Lanubile and G. Visaggio, "Extracting reusable functions by flow graph based program slicing," *IEEE Transactions on Software Engineering*, vol. 23, no. 4, pp. 246–259, April 1997.

[9] T. Li, Y. Guo, and S.-K. Li, "Automatic circuit extractor for HDL description using program slicing," *Journal of Computer Science and Technology*, vol. 19, pp. 718–728, 2004.

[10] C.-N. Liu and J.-Y. Jou, "A FSM extractor for HDL description at RTL level," in *Proc. of Asia-Pacific Conference on Hardware Description Languages*, 1998, pp. 33–38.

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

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

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

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