Utilizing signal-level data flow graph in analysing large-scale RTL circuits

Supported by the GAELS project:
Globally Asynchronous Elastic Logic Synthesis

Wei Song

13/05/2014
SDFG

- SDFG ---- Signal-level Data Flow Graph

```verbatim
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;
```

Controller recognition
Data-path recognition
Index

• Extracting SDFG from Verilog RTL designs
  – Definitions of SDFG
  – Type estimation

• Automatic detection of controllers
  – Type calculation
  – Register Relation Graph (RRG)
  – Controller recognition

• Automatic data-path extraction
  – Trimming of control related nodes and arcs
A Traffic Light Controller

```verilog
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;
```
Relation between signals
Tool Flow

1. Convert a multi-file Verilog RTL to abstract syntax trees (ASTs) using a Verilog Parser (Bison + Flex).
2. For each Module, generate an SDFG graph with all signals drawn as nodes with types (FF, combi, port, module).
3. Connect nodes by estimating the relations between signals (arc type estimation).
Node Insertion

\[ \text{cnt} \xrightarrow{\text{cnt} == 0} \text{state} \xrightarrow{\text{state_nxt}} \text{rstn} \xrightarrow{\text{clk}} \text{i_port} \xrightarrow{\text{o_port}} \text{combi_block} \xrightarrow{\text{seq_block}} \]

\[ \text{RY: 3 sec} \quad \text{GY: 5 sec} \quad \text{R: 50 sec} \quad \text{G: 50 sec} \]

\text{cnt} \text{state} \text{cnt} \text{nxt} \text{yellow green red} \text{rstn clk}

\text{red green yellow} \]
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;

if cnt == 0
  case state
    R: state_nxt = YR;
    YR: state_nxt = G;
    G: state_nxt = YG;
    Default: state_nxt = state;
  else
    state_nxt = YR;
    state_nxt = G;
    state_nxt = YG;
    state_nxt = R;
Arc Type Estimation

if cnt == 0
else
    case state
        R  state_nxt = state
        YR state_nxt = YR
        G  state_nxt = G
        Default state_nxt = YG
        state_nxt = R
        state_nxt = state
        R  state_nxt = state
        YR state_nxt = YR
        G  state_nxt = G
        Default state_nxt = YG
        state_nxt = R
Arc Type Estimation

if cnt == 0
    case state
        R
        YR
        G
        Default
        state_nxt = YR
        state_nxt = G
        state_nxt = YG
        state_nxt = R
    else
        state_nxt = state

control

data

FF

state

rstn

clk

red

green

yellow
Arc Type Estimation

The diagram shows a state machine with transitions labeled with signals such as `rstn`, `clk`, `cnt`, and `state`. The states are connected with arcs, and there are additional nodes labeled with `i_port`, `o_port`, `combi_block`, `seq_block`, `reset`, `clock`, `control`, and `data`. The colors red, green, and yellow are used to distinguish different transitions or states.
Available Types

- **DATA**
  - DDP: self loop
    - state = \texttt{state};
  - CAL: calculation
    - sum = \texttt{a} + \texttt{b};
  - ASS: assign
    - dout = \texttt{din};
  - DAT: other
    - dout = \texttt{din} & \texttt{enable};

- **CONTROL**
  - CMP: comparison
    - If(\texttt{a > b})
  - EQU: equal
    - If(\texttt{state == S\_BEGIN})
  - LOG: logic calculation
    - If(\texttt{valid \\&\\& en})
  - ADR: address
    - dout = \texttt{mem[adr]}
  - CTL: other
    - If(\texttt{a \\& b})
Case 1: Controller Extraction

- Traditional: Synopsys Design Compiler
  - Coding style, only FSM

- SDFG:
  - Pattern matching in SDFG and RRG (register relation graph)
  - FSM, counter, flag

- Wei Song and Jim Garside. **Automatic controller detection for large scale RTL designs.** In *Proc. of EUROMICRO Conference on Digital System Design (DSD)*, Santander, Spain, pp. 884-851, September 2013
Define a Controller

- Controller: Finite State Machine
  - A register which stores states
  - Control the behaviour of other signals
  - A finite state space
Examples

FSM is a controller

Counter can be a controller

PC is NOT a controller
Pattern Criteria

• **Definition 1**
  – A controller is a register which has at least one self-loop not going through higher hierarchy.
  (A register which stores states)

• **Definition 2**
  – A controller is a register which has at least one controlling output path.
  (A controlling register)

• **Definition 3**
  – The inputs for a controller must be constant or from itself.
  (Finite state space)
Register Relation Graph (RRG)

RRG is a simplified and flattened form of SDFG. It has only registers and ports. The pattern criteria can be directly applied on RRG.
Tool Flow

Multi-file Verilog RTL designs

Hierarchical internal abstract semantic tree

Hierarchical signal-level data flow graphs (Connections between *regs* and *wires*)

Register relation graph (Connections between *regs*)

Controller detection and report
Reduction of Combi. Nodes
Path Type Calculation

Type reduction

\[
m = \text{sel} \, ? \, 5 : 3;
\]
\[
o = m + 4;
\]
\[
o = \text{sel} \, ? \, 5+4 : 3+4;
\]

Parallel type preservation

\[
s = \text{tok} == 0 ? \text{tok} : 0;
\]
\[
d = d + s;
\]
\[
d = \text{tok} == 0 ? d + o : d + \text{tok};
\]
Path Exploration

- Full unfolding
  - Time consuming
  - >30 mins
Software Cache (Dyn. Prog.)

![Diagram of a cache system with nodes labeled regA, regB, regC, comE, comF, comG, comH, comJ, and outD. The diagram shows data and control flow between these nodes.]
## Results

<table>
<thead>
<tr>
<th>Design Name</th>
<th>Description</th>
<th>Reason to choose</th>
<th>No. of Regs</th>
</tr>
</thead>
<tbody>
<tr>
<td>OR1200</td>
<td>A 32-bit 5-stage OpenRISC microprocessor</td>
<td>Data path controlled PC, Scan chain, Combinational forward loop</td>
<td>124</td>
</tr>
<tr>
<td>Reed-Solomon</td>
<td>A claimed industrial standard Reed-Solomon decoder IP</td>
<td>A not so well-written design, Single block FSMs with irrelevant signal assignments</td>
<td>325</td>
</tr>
<tr>
<td>H.264/AVC</td>
<td>A 196K gate H.264/AVC baseline decoder</td>
<td>A well-written and large-scale design</td>
<td>855</td>
</tr>
</tbody>
</table>
Results

<table>
<thead>
<tr>
<th>Name</th>
<th>Time</th>
<th>Reported</th>
<th>Verified</th>
<th>FSM</th>
<th>Counter</th>
<th>Flag</th>
<th>Error</th>
<th>Error Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>OR1200</td>
<td>1s</td>
<td>19</td>
<td>17</td>
<td>7</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>89%</td>
</tr>
<tr>
<td>Reed-Solomon</td>
<td>2.0s</td>
<td>56</td>
<td>54</td>
<td>6</td>
<td>36</td>
<td>12</td>
<td>2</td>
<td>96%</td>
</tr>
<tr>
<td>H.264/AVC</td>
<td>7.1s</td>
<td>55</td>
<td>49</td>
<td>13</td>
<td>30</td>
<td>6</td>
<td>6</td>
<td>89%</td>
</tr>
</tbody>
</table>

Environment: Intel Core™ 2 Duo 3.00 GHz PC with 2GB memory

All FSMs are detected with a small number of false errors.

Limitations:
Combinational loops, separated assignment (a[0]=b; a[1]=c;), Rom-style tables, etc.
Computer Generated SDFG
> report_fsm
SUMMARY:
In this extraction, 2074 nodes has been scanned, in which 120 nodes are registers.
In total 30 FSM controllers has been found in 101 potential FSM registers.
The extracted FSMs are listed below:

........
[10] or1200_cpu/or1200_except/except_type FSM|ADR
[11] or1200_cpu/or1200_except/extend_flush FSM|FLAG
[12] or1200_cpu/or1200_except/state FSM|FLAG
[13] or1200_cpu/or1200_if/saved FLAG
[14] or1200_cpu/or1200_mult_mac/div_free FLAG

........
Case 2: Data-path Extraction

- Traditional
  - State space analysis
  - Only feasible for small designs

- SDFG
  - Trimming control related nodes
  - Fast even for large scale designs

Tool Flow

Multi-file Verilog RTL designs

Hierarchical internal abstract semantic tree

Hierarchical signal-level data flow graphs (Connections between \textit{regs} and \textit{wires})

Remove control arcs

Remove dangling nodes

Report data path in a reduced SDFG
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_less_than_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_less_than_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_less_than_B = 0;
        A_New = A_Hold - B_Hold;
    end else begin
        A_less_than_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
Remove Control Arcs
Trim the SDFG
Permutation Module (SHA-3)
Large Scale Designs
# Performance

<table>
<thead>
<tr>
<th>Design</th>
<th>Signal-level DFG</th>
<th>Extracted data path</th>
<th>Running time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>I/O</td>
<td>Module</td>
<td>Signal</td>
</tr>
<tr>
<td>OR1200</td>
<td>52</td>
<td>37</td>
<td>2074</td>
</tr>
<tr>
<td>RSD</td>
<td>7</td>
<td>24</td>
<td>1063</td>
</tr>
<tr>
<td>NOVA</td>
<td>19</td>
<td>140</td>
<td>7043</td>
</tr>
</tbody>
</table>
Async Verilog Synthesisor (AVS)

• AVS
  – A C/C++ analysis shell system
  – https://github.com/wsong83/Asynchronous-Verilog-Synthesiser

• Libraries
  – Parser: Bison, Flex, Boost::Spirit
  – Function: C++0x, GNU Boost, GNU MP Bignum lib
  – Shell: cppTcl, Tcl/Tk, rlwrap
  – Graphic: Qt, OGDF
  – File format: pugixml
Async Verilog Synthesisor (AVS)

- **Shell commands**
  - analyze: read in the Verilog HDL design files.
  - annotate_saif: annotate a saif file for a design.
  - current_design: set or show the current target design.
  - elaborate: build up a design from a Verilog module.
  - extract_datapath: extract the datapaths from an SDFG.
  - extract_rrg: extract a register relation graph (RRG) from.
  - extract_sdfg: extract the SDFG of a module.
  - partition: partition the current design.
  - read_saif: read a saif file for a design.
  - report_fsm: report the FSMs in a design.
  - report_partition: report possible partitions of the current design.
  - write_sdfg: write out the SDFG graph (SDFG/RRG/DataPath).
Summary

• SDFG
  – A new diagram which represents the relations between the signals in a large scale Verilog RTL design.

• Automatic Controller Extraction
  – Extracting all controllers using pattern matching

• Data-path extraction
  – Brutal but effective

• Future work
  – Port type estimation
  – Data rate estimation / data-flow extraction
  – System partition
THANK YOU!