2015 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

# Methodology for Separation of Design Concerns Using Conservative RTL Flipflop Inference

**Maya H. Safieddine, Fadi A. Zaraket, Rouwaida Kanj**

*American University of Beirut, Lebanon*

**Ali Elzein, Wolfgang Roezner**

*IBM Systems and Technology Group, Austin TX*

1866 AUB American University of Beirut الجامعة الأميركية في بيروت

IBM

# Outline

- Aspect Oriented Design
  - Hardware Perspective
- Methodology for Separation of Hardware Design Concerns
  - RTL-to-RTL Transformation Block
  - Correctness
  - Concern Insertion Block
- Implementation and Results
- Conclusions

# Aspect Oriented: What's Behind the Big Name

- Concept borrowed from software: Aspects support separation of concerns
  - Example: log every function call w/o modifying functions
  - Languages (AspectC, JAspect) separate coding of aspects
- Easy to make changes in every aspect
- Did not take off due to the lack of useful applications, legacy software, infrastructure and integration

# Concern Coupling Example

**Class with 1) error handling, 2) counting and 3) locking for concurrency concerns woven in**

```
class Queue {
  Item *first, *last;
#ifdef COUNTING_ASPECT
  int counter;
#endif
#ifdef LOCKING_ASPECT
  os::Mutex lock;
#endif
public:
  Queue () : first(0), last(0) {
#ifdef COUNTING_ASPECT
    counter = 0;
#endif
  }
  void enqueue(Item* item) {
#ifdef LOCKING_ASPECT
    lock.enter();
    try {
#endif
#ifdef ERRORHANDLING_ASPECT
      if (item == 0)
        throw QueueInvalidItemError()
#endif
      if (last) {
        last->next = item;
        last = item;
      } else { last = first = item; }
#ifdef COUNTING_ASPECT
      ++counter;
#endif
#ifdef LOCKING_ASPECT
    } catch (...) {
      lock.leave(); throw;
    }
    lock.leave();
#endif
  }
```

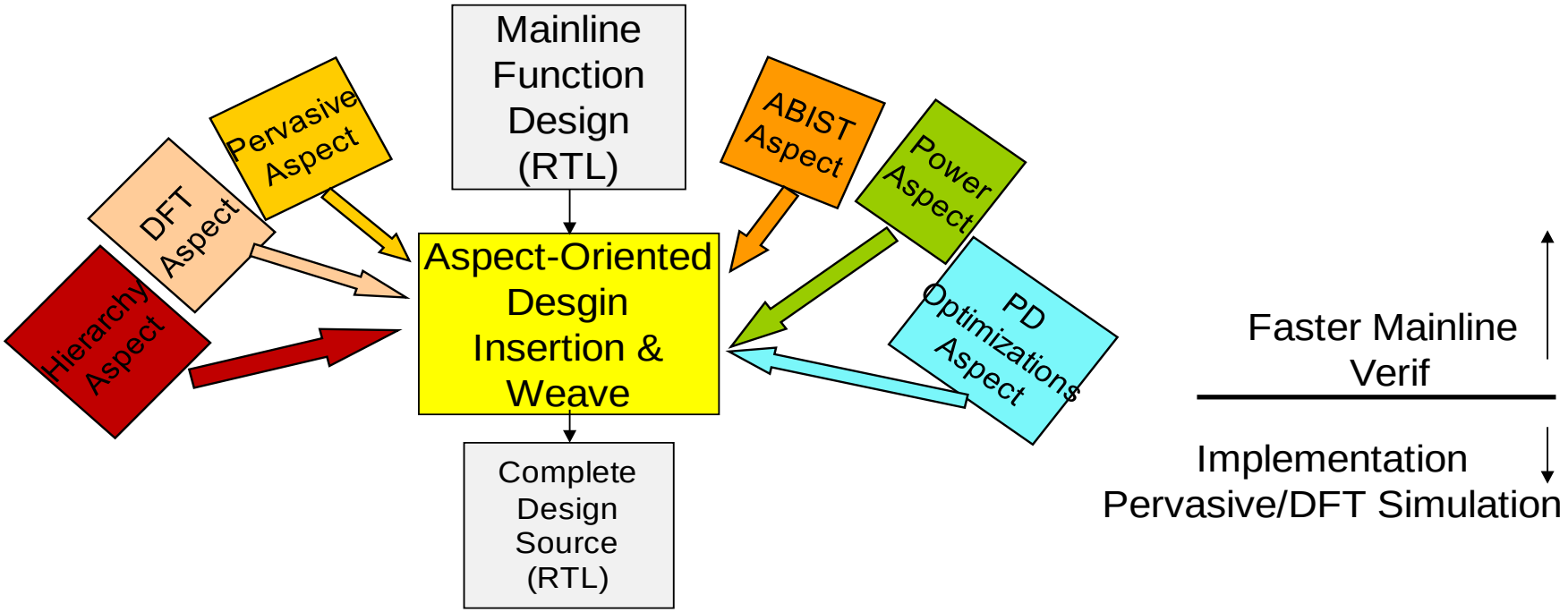
```
  Item* dequeue() {
    Item* res;
#ifdef LOCKING_ASPECT
    lock.enter();
    try {
#endif
      res = first;
      if (first == last)
        first = last = 0;
      else first = first->next;
#ifdef COUNTING_ASPECT
      if (counter > 0) --counter;
#endif
#ifdef ERRORHANDLING_ASPECT
      if (res == 0)
        throw QueueEmptyError();
#endif
#ifdef LOCKING_ASPECT
    } catch (...) {
      lock.leave();
      throw;
    }
    lock.leave();
#endif
    return res;
  }
#ifdef COUNTING_ASPECT
  int count() { return counter; }
#endif
}; // class Queue
```

**Functional Queue class**

```
class Queue { public:

Item * first, *last;

Queue() : first (0),
last(0) {}

void enqueue (Item *item){
  if (last){
    last-> next = item;
    last = item;
  } else {
    last = first = item;
  } }

void dequeue (Item *item){
  Item * res = first;
  if (first == last) {
    first = last = 0;
  } else {
    first = first->next; }
  return res;}

}; // class Queue
```

# Aspect Oriented Design

- Cross-cutting (coupling) of concerns exists in hardware
  - Designers have to deal with highly coupled code
  - Verification is challenging and late in design cycle
- We identify several hardware concerns that can be isolated into aspects

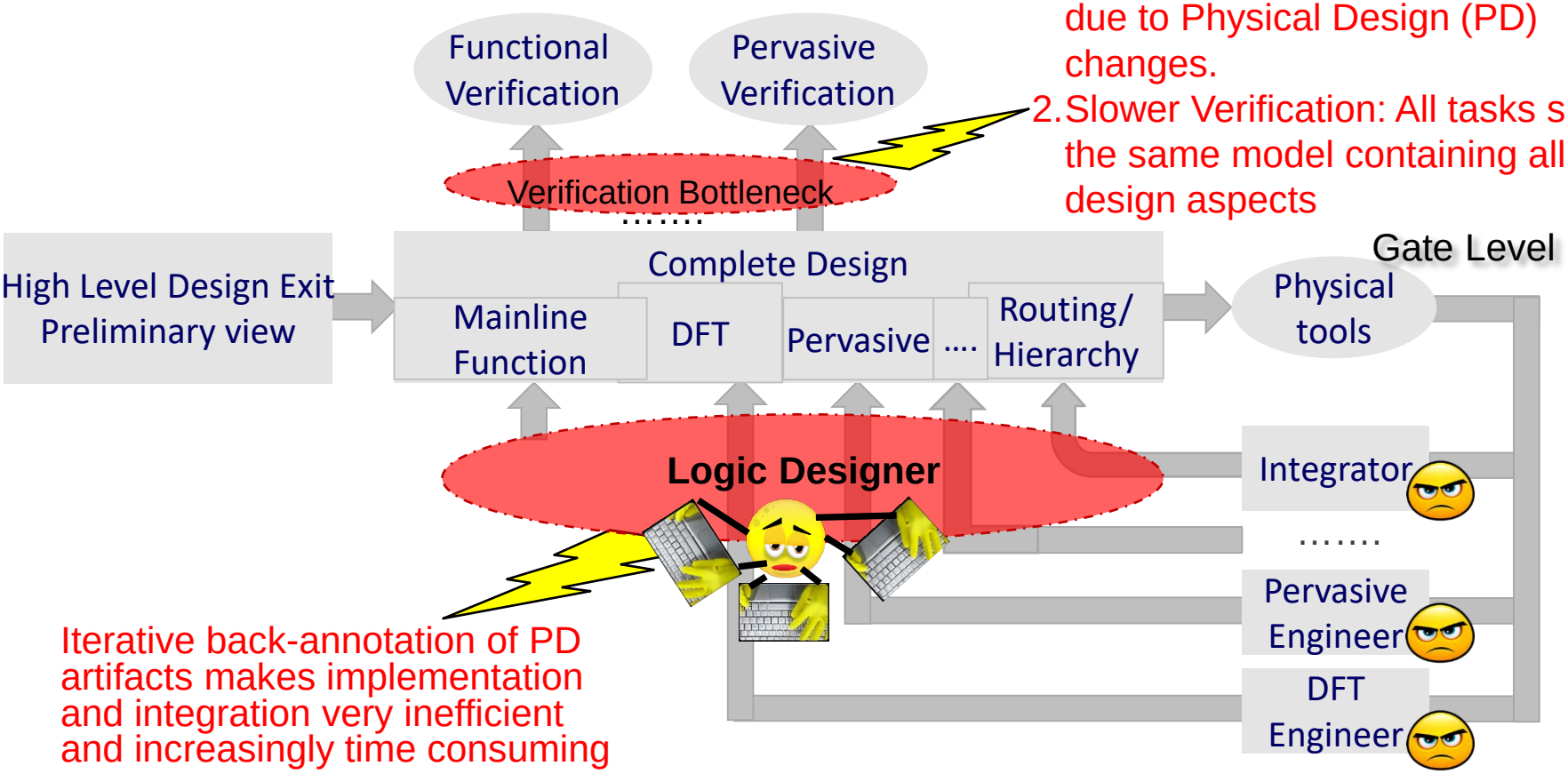


# Advantages of Aspect Oriented Design

- **Increase Designers productivity**
  - Automatic insertions of several aspects
  - Designers work in parallel coding 1) aspects, 2) aspect supporting HDL, and 3) functional HDL
  - Internally, aspects are referred to as recipe files
- **Faster Verification**
  - Smaller model: Aspect-specific verification happens on the model that only contains the corresponding aspect
  - Verification environment is more stable
    - Changes to aspects that are not part of the verification task have no effect on the environment
    - Easier to debug

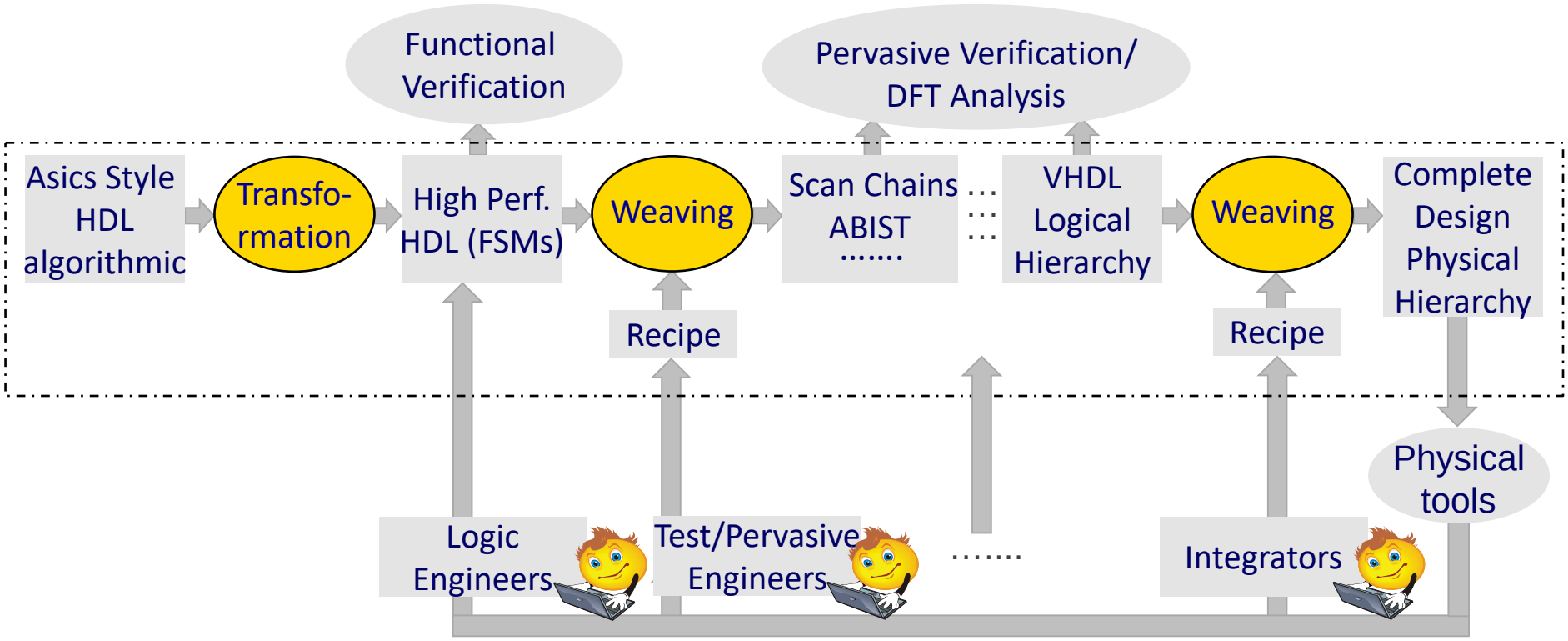
# Old Design Paradigm

1. Environment constantly changes due to Physical Design (PD) changes.
2. Slower Verification: All tasks share the same model containing all design aspects



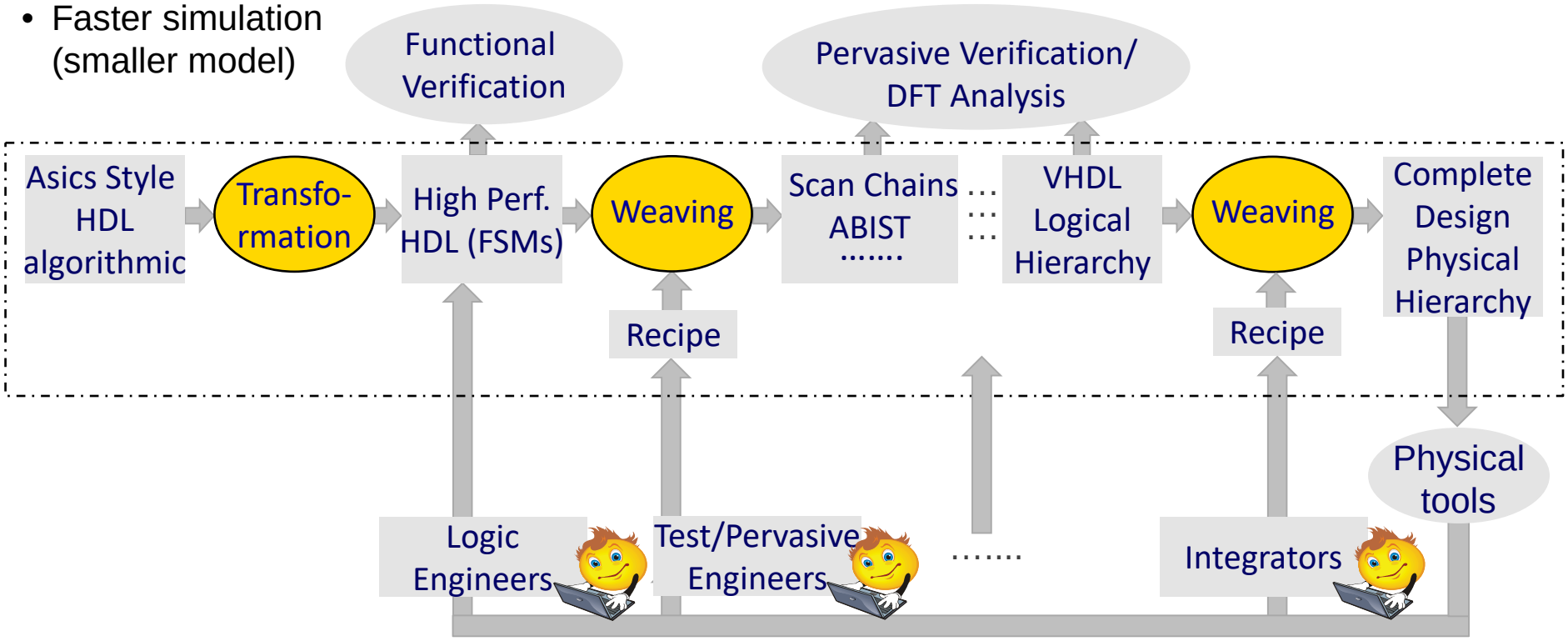

Iterative back-annotation of PD artifacts makes implementation and integration very inefficient and increasingly time consuming

# Aspect Oriented Based Design Paradigm



# Aspect Oriented Based Design Paradigm

- More Stable Verification Env
- Faster simulation (smaller model)



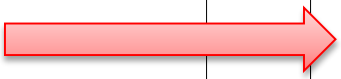

- Feedback now goes separately to different Engineers.
- Every Engineer is now in control of the changes needed to address the issues related to his/her area of expertise

# Four Transforms

1. **Structural Based**: Identifies RTL sequential blocks that can directly be mapped to DFFs
   ⇔ **Simple Blocks**
2. **Algorithmic Based**: Maps generic RTL sequential blocks into RTL combination blocks and flipflop instantiations
   ⇔ **Advanced Blocks**
3. **Synthesis Based**: Maps RTL sequential blocks into low level netlists
   ⇔ **Algorithmic Engine Fails**
4. **Synthesis-Algorithmic Based**: Synthesis results enable algorithmic transformation ⇔**Advanced Engine**
   - Not implemented yet: Currently reverting to Synthesis Based Transform

# Structural Based Transformation

```
//set-resets in structural always
 block
 //structural

 module set1(in,clk,set,reset,out);
  input in, clk, set, reset;
  output reg out;

  always @ (posedge clk or
                posedge reset)
    if (reset)
      out <= 0;
    else if (set)
      out <= 1;
    else
      out <= in;

 endmodule
```



```
module set1(in,clk,set,reset,out);
      input in, clk, set, reset;
      output out;
      wire out;

    dff   out_0(
       .d(in), .clk(clk),
.async_reset(reset),
       .async_data(0),
.sync_reset(set),
       .sync_data(1), .gate(1'b1),
.q(out));

    endmodule
```

# Algorithmic Based Transformation

**Advanced Blocks:**

1. Any signal set inside an always block with posedge or negedge (or any combination of them) is inferred flipflop

2. Extract clocking and asynchronous logic: create the DFF model

3. Create the combination always block

4. Generate equivalent HDL

# Combinational Block

1. Drop from the original process the asynchronous set/reset logic as well as the clocking construct.

2. For every flipflop signal assigned

| | Assignment | |
|---|---|---|
| **Flip Flop Signal** | **Blocking** | **Non-blocking** |
| **Target of assignment** | Replace it with FF input | Replace it with FF input |
| **Referenced** | Replace it with FF input | Keep it |

- For all flipflops add an assignment at the beginning of the process, assigning the output of the flipflop to its input
  - Ensures no latches inferred inside the combination always block
  - Alleviates understanding latch inference algorithms

# Algorithmic Transform

Input RTL

```
module example (
inputs cond1, cond2,
       a, b, c, d,
outputs reg r, r2,
inputs clk, …);
reg r3, x;
always@( posedge clk, …)
begin
  if(cond1==1) begin
    r3 <= a;
    x = a || b;
  end
  r <= x;
  if(cond2==1) begin
    x = c && d;
  end
  r2 <= r3 && x;
 end
endmodule
```

**Step 1: Flipflop Inference**

# RTL-to-RTL Transformation

Input RTL

```
module example (
inputs cond1, cond2,
       a, b, c, d,
outputs reg r, r2,
inputs clk, …);
reg r3, x;
always@( posedge clk, …)
begin
  if(cond1==1) begin
    r3 <= a;
    x = a || b;
  end
  r <= x;
  if(cond2==1) begin
    x = c && d;
  end
  r2 <= r3 && x;
 end
endmodule
```

**Step 1: Flipflop Inference**

**Inferred flipflops:**
**r, r2, r3, x**

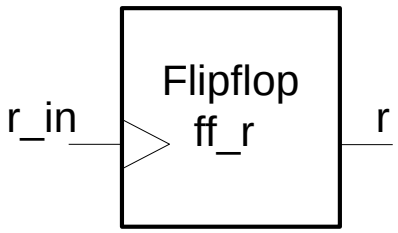
# RTL-to-RTL Transformation

**Input RTL**

```
module example (
inputs cond1, cond2,
       a, b, c, d,
outputs reg r, r2,
inputs clk, …);
reg r3, x;
always@( posedge clk, …)
begin
  if(cond1==1) begin
    r3 <= a;
    x = a || b;
  end
  r <= x;
  if(cond2==1) begin
    x = c && d;
  end
  r2 <= r3 && x;
 end
endmodule
```

**Step 2: Add wire and reg variables to map to the output and next state of the flipflops**

```
module example (
inputs cond1, cond2,
  a, b,  c, d,
outputs wire r, r2,
input clk, …);
wire r3,  x;
reg r_in, r2_in, r3_in, x_in;
…
```



# RTL-to-RTL Transformation

Input RTL

```
module example (
inputs cond1, cond2,
       a, b, c, d,
outputs reg r, r2,
inputs clk, …);
reg r3, x;
always@( posedge clk, …)
begin
  if(cond1==1) begin
    r3 <= a;
    x = a || b;
  end
  r <= x;
  if(cond2==1) begin
    x = c && d;
  end
  r2 <= r3 && x;
 end
endmodule
```

**Step 3: Flipflop Instantiation**

```
module example (
inputs cond1, cond2,
  a, b,  c, d,
outputs wire r, r2,
input clk, …);
wire r3,  x;
reg r_in, r2_in, r3_in, x_in;

ff ff_r(r_in, r,0, clk ….);
ff ff_r2(r2_in,r2,0,clk …);
ff ff_r3(r3_in,r3,0,clk …);
ff ff_x(x_in, x,0, clk …);
…
```

Sequential Block

# RTL-to-RTL Transformation

**Input RTL**

```
module example (
inputs cond1, cond2,
       a, b, c, d,
outputs reg r, r2,
inputs clk, …);
reg r3, x;
always@( posedge clk, …)
begin
  if(cond1==1) begin
    r3 <= a;
    x = a || b;
  end
  r <= x;
  if(cond2==1) begin
    x = c && d;
  end
  r2 <= r3 && x;
 end
endmodule
```

**Step 3: Flipflop Instantiation**

```
module example (
inputs cond1, cond2,
  a, b,  c, d,
outputs wire r, r2,
input clk, …);
wire r3,  x;
reg r_in, r2_in, r3_in, x_in;

ff ff_r(r_in, r,0, clk ….);
ff ff_r2(r2_in,r2,0,clk …);
ff ff_r3(r3_in,r3,0,clk …);
ff ff_x(x_in, x,0, clk …);
…
```

**Sequential Block**

**Concern weaving requires only instantiating the corresponding flipflop library by substituting ff**

# RTL-to-RTL Transformation

**Input RTL**

```
module example (
inputs cond1, cond2,
       a, b, c, d,
outputs reg r, r2,
inputs clk, …);
reg r3, x;
always@( posedge clk, …)
begin
  if(cond1==1) begin
    r3 <= a;      ← Non-blocking
    x = a || b;   ← Blocking
  end
  r <= x;
  if(cond2==1) begin
    x = c && d;
  end
  r2 <= r3 && x;
 end
endmodule
```

**Step 4: Construction of the combination logic block**

- **Explicit connection of flipflop input and output signals**

```
…
always@(*)  begin
  if(cond1==1) begin
    r3_in <= a;
    x_in = a || b;
  end
  r_in <= x_in;
  if(cond2==1) begin
    x_in = c && d;
  end
  r2_in <= r3 && x_in;
end
```

# RTL-to-RTL Transformation

```
module example (
inputs cond1, cond2,
       a, b, c, d,
outputs reg r, r2,
inputs clk, …);
reg r3, x;
always@(
begin
  if(cond1==1) begin
    r3 <= a;
    x = a || b;
  end
  r <= x;
  if(cond2==1) begin
    x = c && d;
  end
  r2 <= r3 && x;
 end
endmodule
```

**Input RTL**

**Inferred flipflops:**
**r, r2, r3, x**

**Step 5: Combination logic block**

- **Removal of all possible flipflop inferences**

```
always@(*)  begin
  r_in = r;
  r2_in = r2;
  r3_in = r3;
  x_in = x ;
  if(cond1==1) begin
    r3_in <= a;
    x_in = a || b;
  end
  r_in <= x_in;
  if(cond2==1) begin
    x_in = c && d;
  end
  r2_in <= r3 &&x_in;
end
```

**Combinational Block**

# RTL-to-RTL Transformation

**Transformed RTL output**

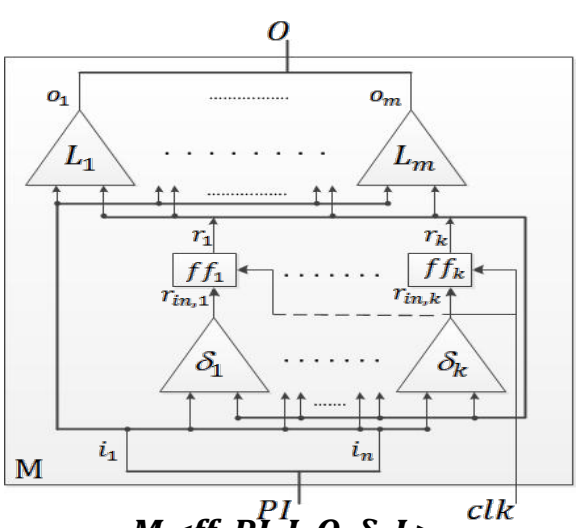
```
module example (
inputs cond1, cond2,
  a, b,  c, d,
outputs wire r, r2,
input clk, …);
wire r3,  x;
reg r_in, r2_in, r3_in, x_in;

ff ff_r(r_in, r,0, clk ….);
ff ff_r2(r2_in,r2,0,clk …);
ff ff_r3(r3_in,r3,0,clk …);
ff ff_x(x_in, x,0, clk …);
…
```

**Sequential Part**
**Instantiated Generic Flipflops**

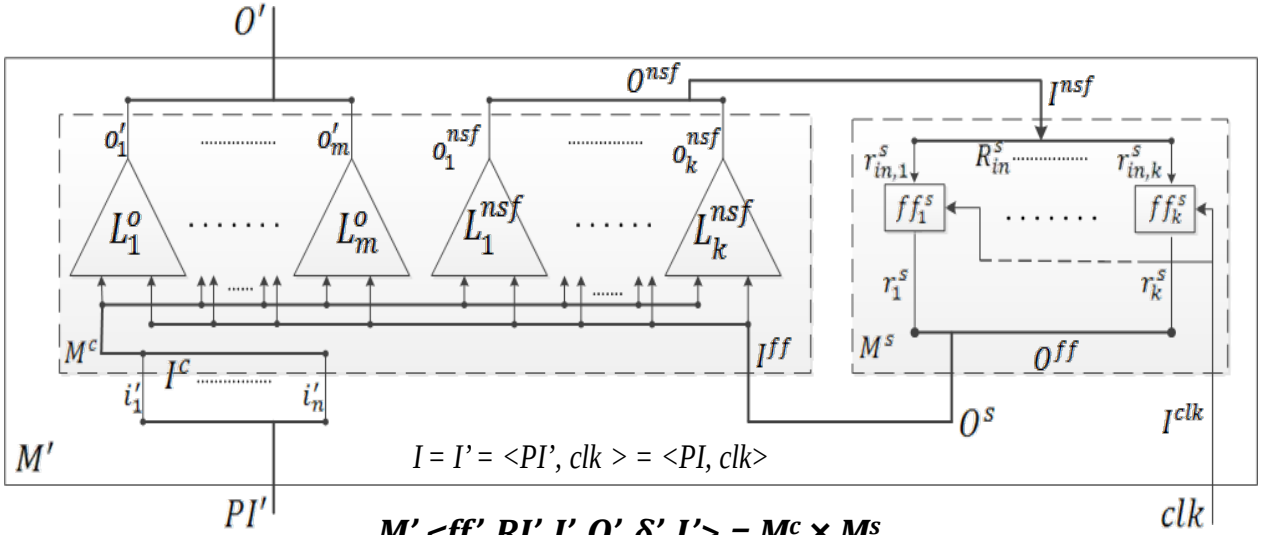
```
always@(*)  begin
  r_in = r;
  r2_in = r2;
  r3_in = r3;
  x_in = x ;
  if(cond1==1) begin
    r3 _in <= a;
    x _in = a || b;
  end
  r _in <= x_in;
  if(cond2==1) begin
    x_in = c && d;
  end
  r2_in <= r3 &&x_in;
end
```

**Combinational Part**
**Logic gates and Connections**

# Sketch of the Transformation Correctness Proof




$M <ff, RI, I, O, \delta, L>$




$M' <ff', RI', I', O', \delta', L'> = M^c \times M^s$

**Given**

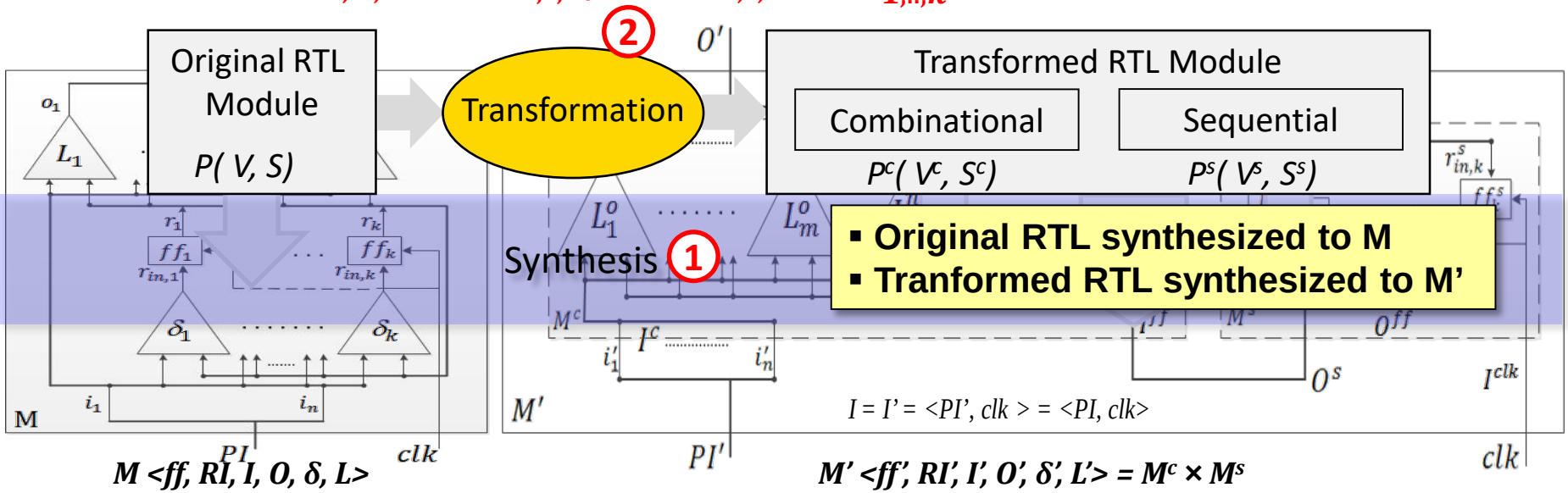
- Mealy Machine M ,
- Mealy Machine M' with separated Combinational and Sequential Structures

If $\{L_{1,..,m} \equiv L^0_{1,..,m}\}$ & $\{\delta_{1,..,k} \equiv L^{nsf}_{1,..,k}\}$ then M'≡M

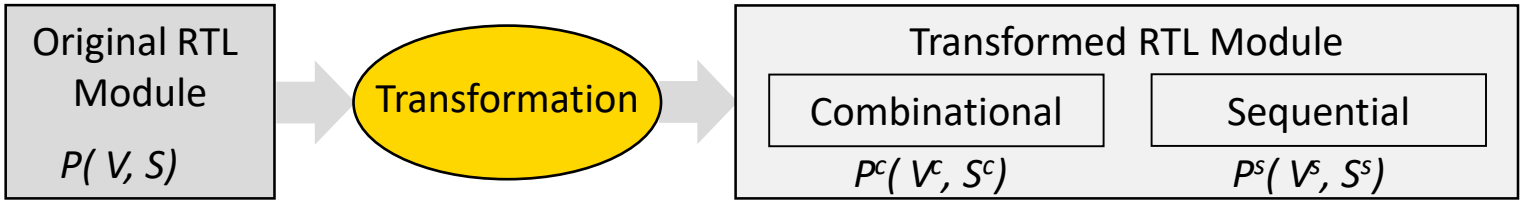
***ff***: *Flipflops /Registers*
***RI***: *Register initial values*
***I***: *Input wires*
***O***: *Output wires*
***δ***: *Next State functions*
***L***: *Output functions*

# Sketch of the Transformation Correctness Proof

$$\{L_{1,..,m} \equiv L^{o}_{1,..,m}\} \,\&\, \{\delta_{1,..,k} \equiv L^{nsf}_{1,..,k}\}$$



# Program P




***P( V, S)***

*Variables **V**:*
*Inputs **I**,*
*Outputs **O**,*
*Registers **R**,*
*Wires **W***

*Statements **S**:*
*blocking assignments (**BS**) "u = exp"*
*Non-blocking assignments (**NBS** ) "u <= exp"*
*Other Statements*

# Program P synthesis

***V**: Variables*
***I**: Inputs*
***O**: Outputs*
***R**: Registers*
***W**: Wires*
***S**: Statements*
***BS**: blocking assignments*
***NBS**: Non-blocking assignments*

***P( V, S)***
***V** = { **I**, **O**, **W**, **R**}*
***S** = {**BS**, **NBS**, Other}*
*assignment statements are in the form: u = exp*

Original RTL Module *P( V, S)*

Synthesis

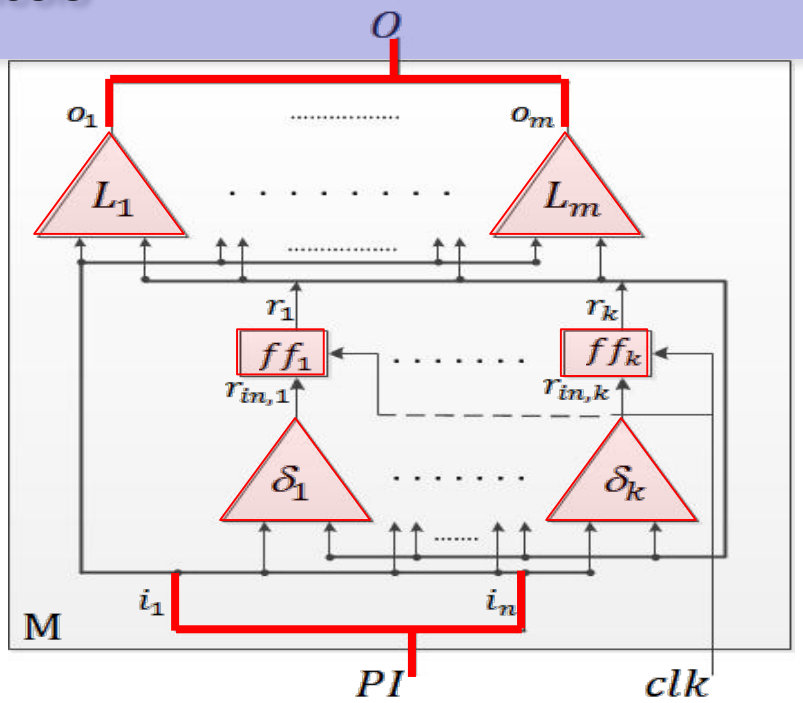
***ff** = Synthesis(**R**)*

***I** = Synthesis(**I**)*

***O** = Synthesis(**O**)*

***δ** = Synthesis(**S**| u∈**R**)*

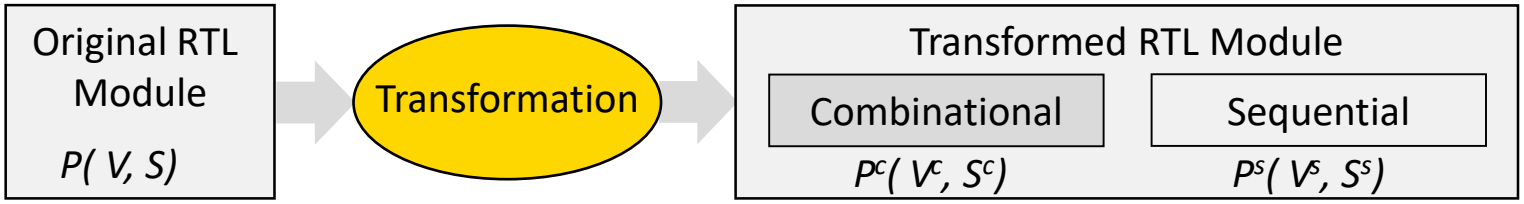
***L** = Synthesis(**S**| u∈**O**)*




***ff**: Flipflops /Registers*
***RI**: Register initial values*
***I**: Input wires*
***O**: Output wires*
***δ**: Next State functions*
***L**: Output functions*

# Program $P^c$

***V**: Variables*
***I**: Inputs*
***O**: Outputs*
***R**: Registers*
***W**: Wires*
***S**: Statements*
***BS**: blocking assignments*
***NBS**: Non-blocking assignments*




$$P^c(V^c, S^c)$$

$$V^c = \{I^c, O^c, R^c, W^c\}$$
$$R^c = \{\}$$
$$W^c = V \cup \{u_{in} : u \in R\}$$
$$O^c = O \cup \{u_{in} : u \in R\}$$
$$I^c = I \cup R$$

$$U_{tb} = \{u: u \in R, u \text{ is target of blocking statement}\}$$
$$U_{tn} = \{u: u \in R, u \text{ is target of nonblocking statement}\}$$

$S^c$ is defined iteratively as follows

Let $S^c_{(0)} = S$, $h_n = |U_{tn}|$ and $h_b = |U_{tb}|$

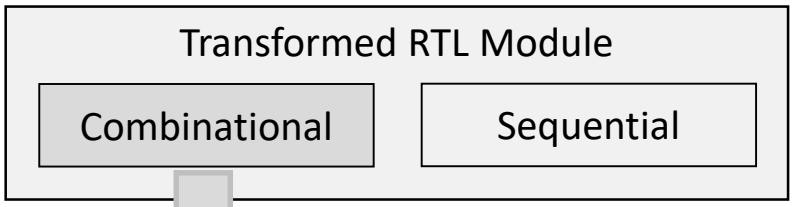
$$S^c_{(i)} = \begin{cases} S^c_{(i-1)}|_{u_i : u_i\_in} & i = 1 \ldots . h_b \\ S^c_{(i-1)}|_{u_i : u_i\_in} & i = h_b + 1 \ldots . h_b + h_n \\ & \text{and } u_i \text{ is a target} \end{cases}$$

***ff**: Flipflops /Registers*
***RI**: Register initial values*
***I**: Input wires*
***O**: Output wires*
***δ**: Next State functions*
***L**: Output functions*

# $P^c$ Synthesis

***V**: Variables*
***I**: Inputs*
***O**: Outputs*
***R**: Registers*
***W**: Wires*
***S**: Statements*
***BS**: blocking assignments*
***NBS**: Non-blocking assignments*

$$R^c = \{\}$$
$$W^c = V \cup \{u_{in} : u \in R\}$$
$$O^c = O \cup \{u_{in} : u \in R\}$$
$$I^c = I \cup R$$

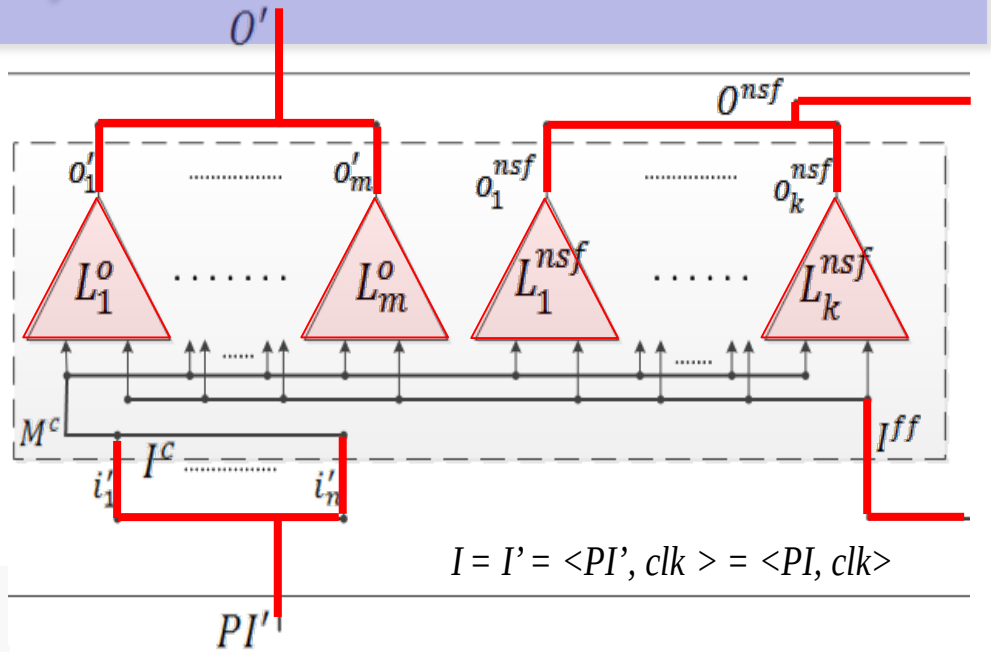



Synthesis

$\{PI', I^{ff}\} = Synthesis(I^c)$

$\{O', O^{nsf}\} = Synthesis(O^c)$

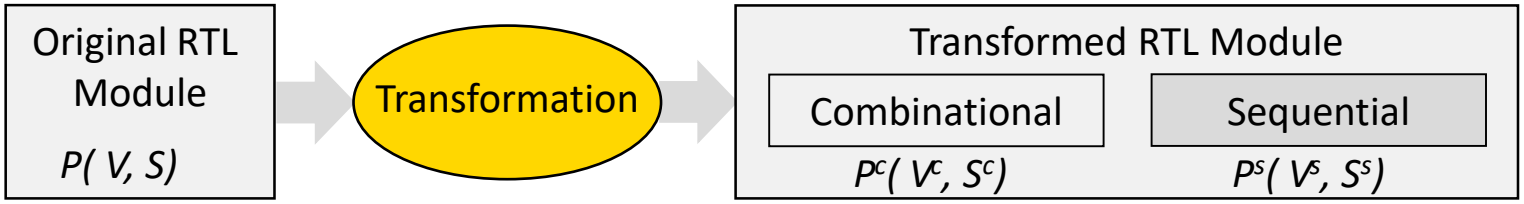
$\{L^{c,O}, L^{c,nsf}\} = Synthesis(S^c | u \in O^c)$




***ff**: Flipflops /Registers*
***RI**: Register initial values*
***I**: Input wires*
***O**: Output wires*
***δ**: Next State functions*
***L**: Output functions*

# Program $P^s$

***V**: Variables*
***I**: Inputs*
***O**: Outputs*
***R**: Registers*
***W**: Wires*
***S**: Statements*
***BS**: blocking assignments*
***NBS**: Non-blocking assignments*




$$P^s(V^s, S^s)$$

$$V^s = \{I^s, O^s, R^s, W^s\}$$
$$R^s = R$$
$$I^s = I + \{u_{in} : u \in R\}$$
$$O^s = R$$
$$W^s = \{\ \}$$
$$S^s = \{s_i^s : ff\ \text{ff\_u(u\_in, u, 0,clk, ...)}\}, where\ u \in R$$

***ff**: Flipflops /Registers*
***RI**: Register initial values*
***I**: Input wires*
***O**: Output wires*
***δ**: Next State functions*
***L**: Output functions*

# $P^s$ Synthesis

*V: Variables* *S: Statements*
*I: Inputs* *BS: blocking assignments*
*O: Outputs*
*R: Registers* *NBS: Non-blocking assignments*
*W: Wires*

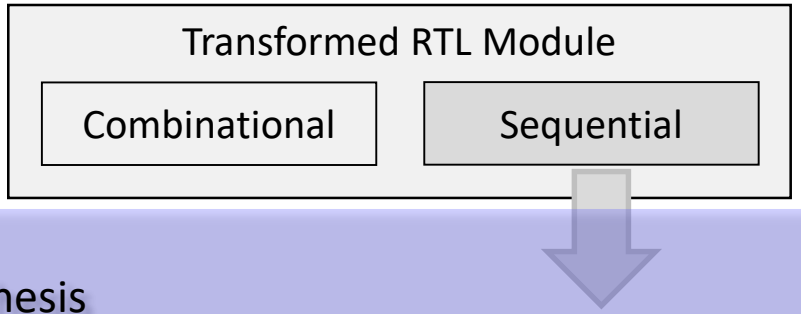
$R^s = R$

$I^s = I + \{u_{in} : u \in R\}$

$O^s = R,$ ⟵ Simple Assignment

$W^s = \{\ \}$

$S^s = \{s_i^s : ff$ ff_u(u_in, u, 0,clk, ...)$\}$
$where\ u \in R$ ⟵ Simple Declaration




Synthesis

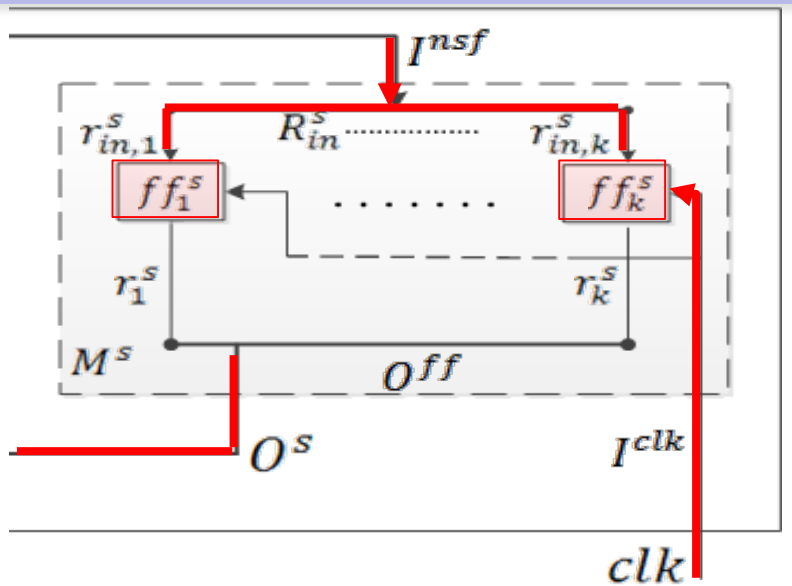
$ff^s$ = Synthesis($R^s$)

$\{I^{clk,...}, I^{nsf}\}$ = Synthesis($I^s$)

$O^{ff}$ = Synthesis($O^s$)

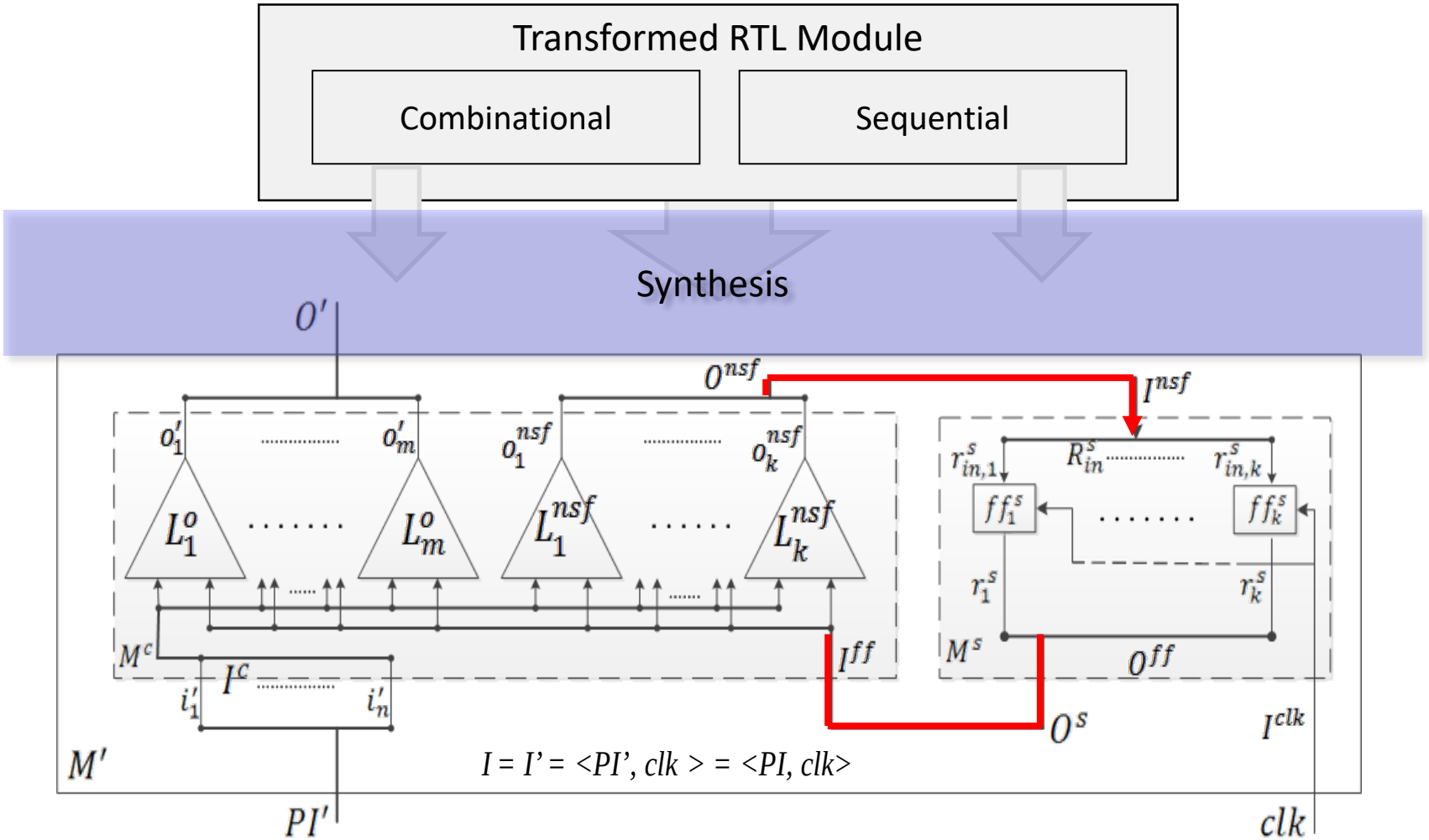
$\delta^s = \{\ \}$ ⟵ Empty Functions (wiring)

$L^s = \{\ \}$ ⟵ Empty Functions (wiring)




*ff: Flipflops /Registers*
*RI: Register initial values*
*I: Input wires*
*O: Output wires*
*δ: Next State functions*
*L: Output functions*

# P' Synthesis into M'



# Functional Equivalence by structural induction

- Equivalence of the synthesis of expressions of original and transformed programs

$$\forall \exp_{c,i} \; in \; P_c \; , \exp_i \; in \; P, \;\; \exp_{c,i} = \exp_i |_{u_j \; with \; u_{in,j}} \; for \; every \; u_j \; \in \boldsymbol{U_{tb}}$$

$$\therefore \; [syn(u_{in,j}) in \; M^c \leftrightarrow syn(u_j) in \; M] \wedge [syn(u_k) in \; M^c \leftrightarrow syn(u_k) in \; M]$$

$$\rightarrow [syn(\exp_{c,i}) \leftrightarrow syn(\exp_c)] \;\; where \; u_k \in \boldsymbol{U_{tn}}$$

- Base and recursive cases for the synthesis equivalence proof of $u_{in}$ and u in M and M'

$$syn(u_{in,j}) = \begin{cases} flipflop \; corresponding \; to \; u_i & base \; case \\ output \; wire \; of \; syn(\exp_{c,j}) & recursive \; case \end{cases}$$

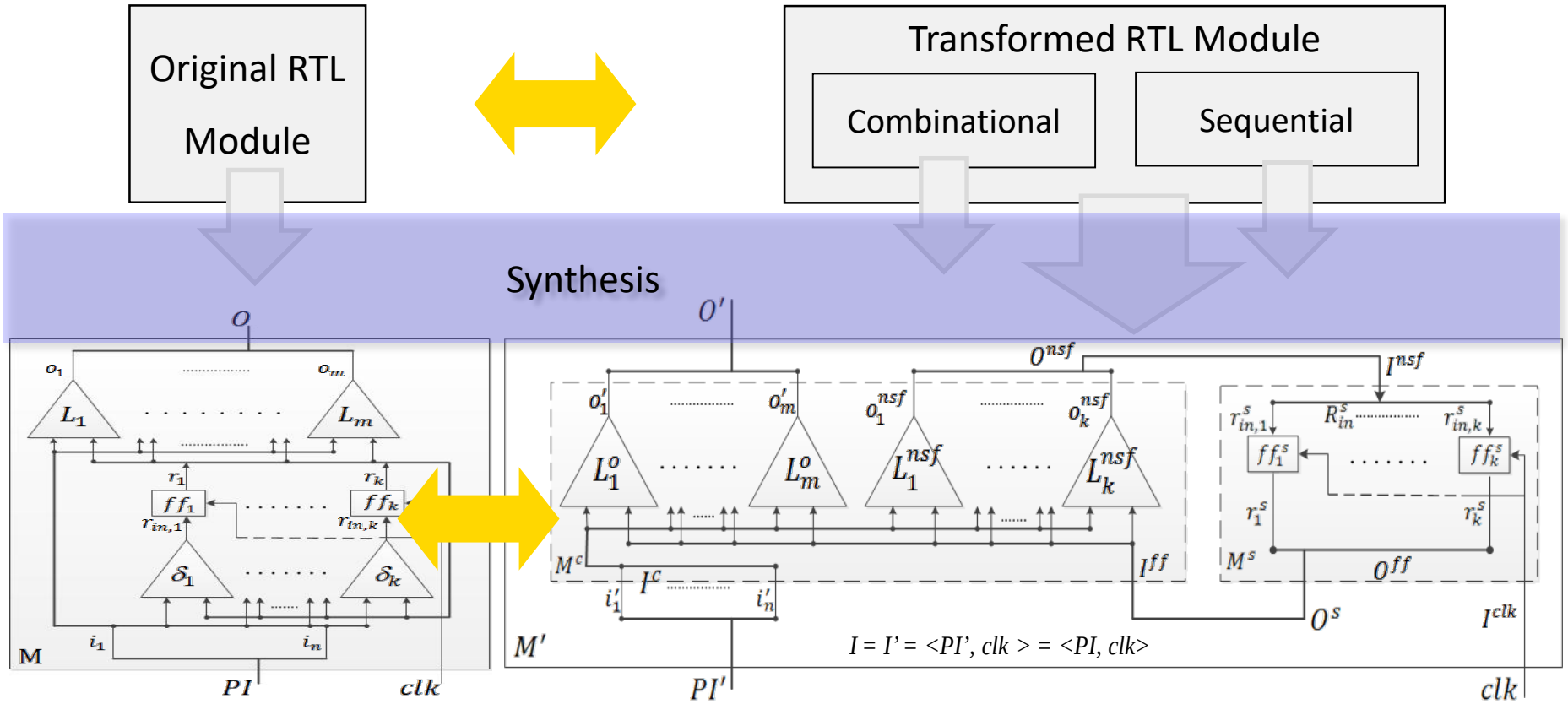
$$syn(u_k) = flipflop \; of \; value \begin{cases} initial \; register \; value \; (input) & base \; case \\ syn(\exp_{c,k}) & recursive \; case \end{cases}$$

- Equivalence of output functions in M' to output and transition functions in M

$$l^c = syn(\exp_c) \;\; \forall \; l^c \in L^c$$

$$\boldsymbol{Therefore: \; L^{c,nsf} \leftrightarrow \delta \;\; and \;\; L^{c,O} \leftrightarrow L}$$

# Transformation Equivalence




$$Since\ \{L_{1,..,m} \equiv L^{o}_{1,..,m}\} \& \{\delta_{1,..,k} \equiv L^{nsf}_{1,..,k}\}\ then\ M' \equiv M$$

$$\Leftrightarrow synthesis(P') = M' \equiv M = synthesis(P)$$

$$\Leftrightarrow P' \equiv P$$

# Implementation and Results

- Developed and deployed an implementation for Verilog RTL
- The implementation supports:
  - Automatic insertion of design concerns at RTL
  - Mapping to proprietary technology at RTL
  - Lifting verification to the RTL level
- Reduced verification time by 40% in the last design cycle
- No drawbacks on synthesis optimizations

# Conclusion

- We introduce the concept of aspect oriented design into hardware by a methodology for automatic concern insertion into RTL programs.
- The methodology reduces verification time by allowing it to take place at the RTL level.
- The methodology can be generalized to different hardware languages, and is proved to maintain original program functionality.
- It is currently used in an industrial settings and has produced significant design cycle time savings for state-of-the-art SoC designs.

# Thank You

# Synthesis Based Transformation Example

```
module mor1 (clk, en, d1, d2, q);
    input wire clk, en;
    input wire d1, d2;
    output reg q;

    reg tmp;

    always @(posedge clk) begin
      if (en)
        tmp = d1 & d2;
      else
        tmp <= !d1;
      q <= tmp;
    end
    endmodule
```

```
module mor1 (clk, en, d1, d2, q);
    input wire clk, en;
    input wire d1, d2;
    output wire q;

    reg tmp;
    wire \$4 ;
    wire [0:0] \$0 ;
    wire [0:0] \$1 ;
    wire \$5 ;
    wire [0:0] \$2 ;
   dff dff_q( .d(\$4 ), .clk( clk ), .q(q)….);
    dff dff_tmp( .d(\$5 ), .clk( clk ), .q(tmp)….);

     assign \$0  = (d1 & d2);
     assign \$1  = (~en);
     assign \$2  = (d1 == 1'b0);
     assign \$5  = (en ? \$0  : \$2 );
     assign \$4  = (en ? \$0  : tmp);


    endmodule
```