Optimizing Blocks in an SoC Using Symbolic Code-Statement Reachability Analysis

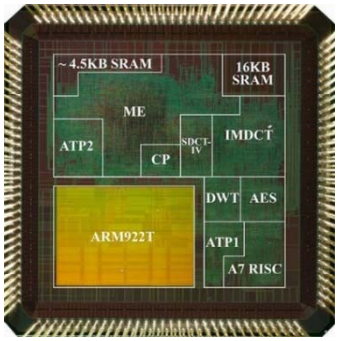
Hong-Zu Chou

January 21, 2010

Hong-Zu Chou, ***National Taiwan Univ., Taipei, Taiwan***
Kai-Hui Chang, ***Avery Design Systems, Inc., Andover, MA, USA***
Sy-Yen Kuo, ***National Taiwan Univ., Taipei, Taiwan***

Motivation

- The use of Intellectual Properties (IPs) in System-on-Chip (SoC) circuits has become a common design practice recently
  - To accelerate the circuit design process
  - To reduce design cost

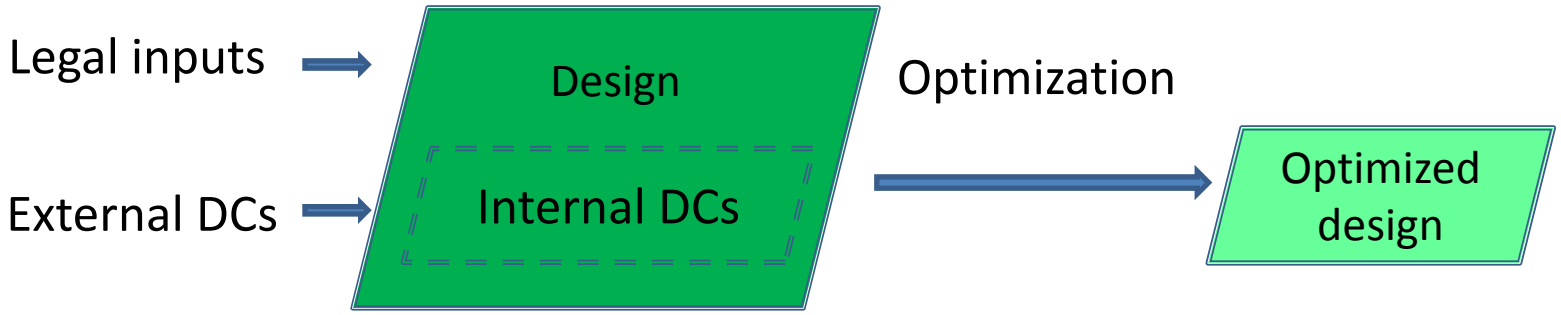


- Unnecessarily large design blocks may exist in the final chip because the blocks may contain unused functions
  - Code blocks for unused functions occupies more die area and consumes unnecessary power after synthesis

Optimizations should be applied to find and remove the unused logic from the reused design blocks

Our Contributions

- A methodology that reuses existing verification environments or surrounding blocks for circuit optimization
  - Utilize abundant external don't-cares that exist in the environment
  - Perform hardware/software co-optimization



Our Contributions (cont.)

- A new algorithm that utilizes high-level symbolic simulation to perform *formal code-statement reachability analysis*
  - Accurately identify redundant code that should be removed
  - Operate on higher-level code instead of gate-level netlists
    - The changes are easier to understand and verify
    - Removing one line of RTL code can eliminate thousands of gates

Our Contributions (cont.)

- An innovative synthesis construct called *sym_wait* that can accelerate symbolic simulation
  - Verify the latency of different symbolic traces and then merge them
  - Provide additional checks for latency-related problems in testbenches or design

Outline

- Preliminaries
- Circuit optimization using constrained-random testbench
- Implementation insights
- Experimental results
- Conclusions

Don’t-Cares (Xs)

- Don’t-Cares in logic synthesis
  - **Controllability Don’t-Cares (CDCs)**
  - Observability Don’t-Cares (ODCs)
- X-pessimistic and X-optimistic characteristics in logic simulation

```
b = 1; c = 1;
Output = ( a & b ) | ( ~a & c);
```

a \ bc	0			
0				
1				
x	0	x	x	x

Example of X-Pessimism

```
always @(*)
   if (sel)    y = a;
   else        y = b;
```

y
a
b
b

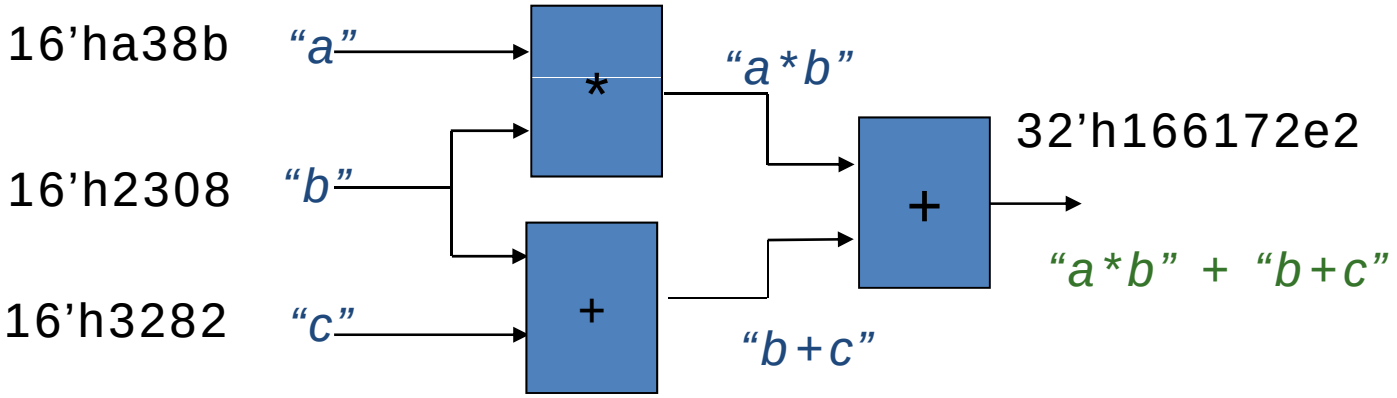
x	b

Example of X-Optimism

Logic simulation can not work for code reachability analysis

Symbolic Simulation Primer

- Symbolic simulation considers all possible executions paths of a system simultaneously
- Symbolic variables are used in the simulation state representation in order to index multiple executions of the system
- Constrained random verification environments can be used directly for symbolic simulation



Problem Formulation

- Given a design containing:
  - *N* conditional code blocks
  - A constrained-random testbench that can generate all sets of possible input patterns
- Goal
  - Remove unused blocks and produce a smaller RTL design based on the given input constraints

Idea: Perform code statement reachability analysis to identify unused blocks, and then remove them

Pseudo Code and Example of Code Statement Reachability Analysis

```
1    event = event_queue.pop();
2    curr_sym_cond = event.sym_cond;
3    while execute statement triggered by event
4       if statement is a conditional block with condition cond
5          curr_sym_cond &= cond;
6          do not execute statment if curr_sym_cond is proven to be 0;
7       else if leaving conditional block with condition cond
8          restore curr_sym_cond by removing cond as constraint;
9       else if a new event nevent needs to be generated
10         nevent →sym_cond = curr_sym_cond;
11         event_queue.add(nevent);
12      statement = statement.next;
```

```
        always @ ( cond, i1, i2 ) begin
S1:     if (cond) begin
S2:        a = i1;
S3:        $display("Important debugging message");
        end else
S4:        #1 a = i2;
S5:     $display("Execution finished");    end
S6:     always @(a)    b = ~a;
```

event_queue.pop();

event = @ (cond, i1, i2)
*statement =**S1***
curr_sym_cond=1 & *cond*
statement.next=cond?S2:S4

cond ==true event_queue.push();

event = @ (cond, i1, i2)
*statement =**S2***
curr_sym_cond=1 & (*cond ==true)*
nevent = @a
event_queue={@a}
statement.next=S3

cond ==false event_queue.push();

event = @ (cond, i1, i2)
*statement =**S4***
curr_sym_cond=1 & (*cond ==false)*
nevent = @a
curr_sym_cond=1 ~~*& (cond ==false)*~~
event_queue={@a}
statement.next=S5

cond ==true

event = @ (cond, i1, i2)
curr_sym_cond=1& (*cond ==true)*
*statement =**S3***
curr_sym_cond=1~~*& (cond ==true)*~~
statement.next=S5

event = @ (cond, i1, i2)
*statement =**S5***
curr_sym_cond=1

event_queue.pop();

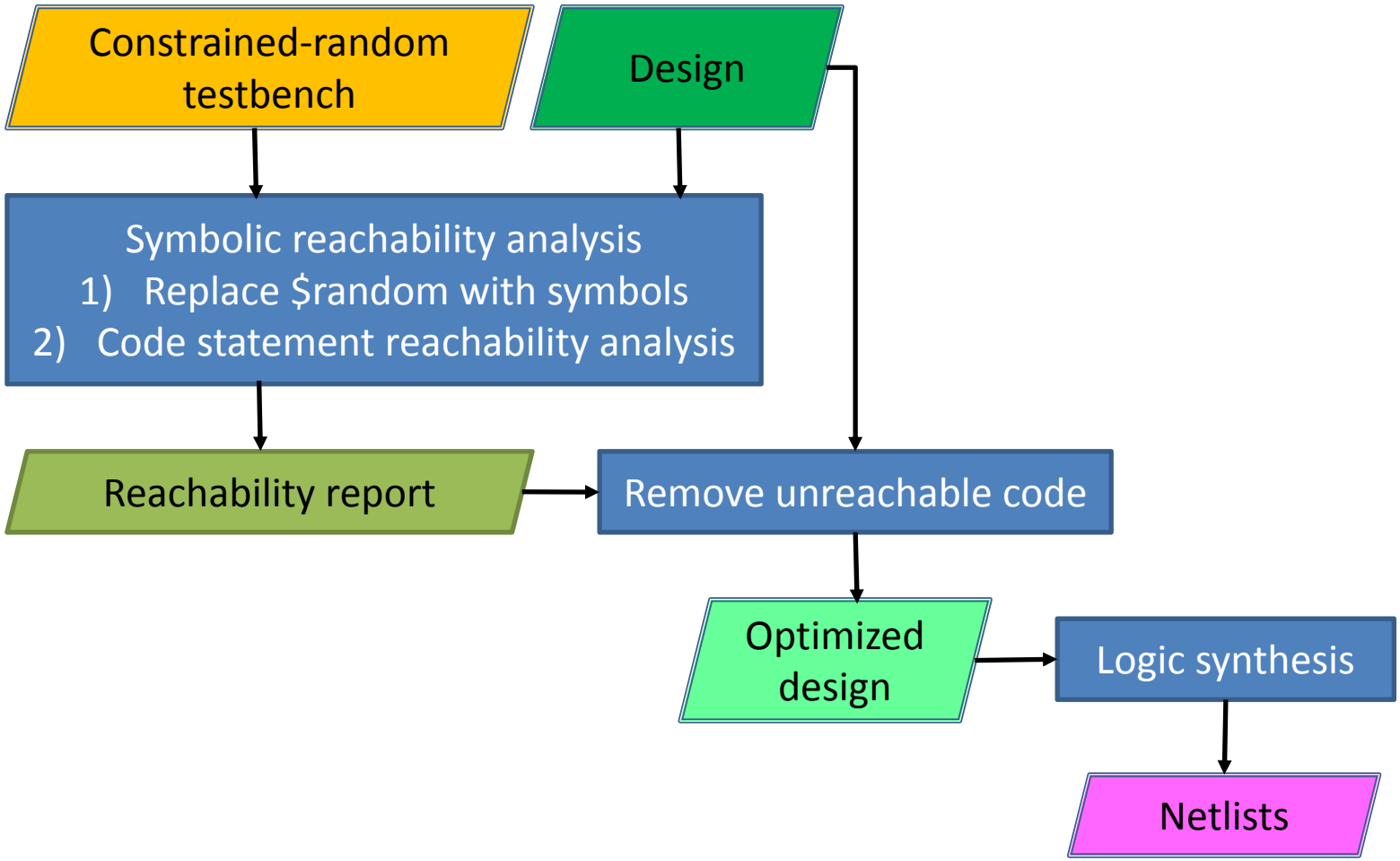
event = @a
*statement =**S6***
curr_sym_cond=1

event = @ (cond, i1, i2)
*statement =**S5***
curr_sym_cond=1

event_queue.pop();

event = @a
*statement =**S6***
curr_sym_cond=1

Synthesis Optimization Using Symbolic Code Statement Reachability Analysis



Combining Logic and Symbolic Simulation

- Logic simulation is extremely fast for random simulation
  - incomplete corner-case coverage
  - X-pessimistic and X-optimistic
- Formal analysis can ensure the correctness of reachability analysis

1. Perform logic simulation for a period of time to identify the conditional blocks that are reachable
2. Use symbolic simulation to check all conditional blocks that are still not reachable

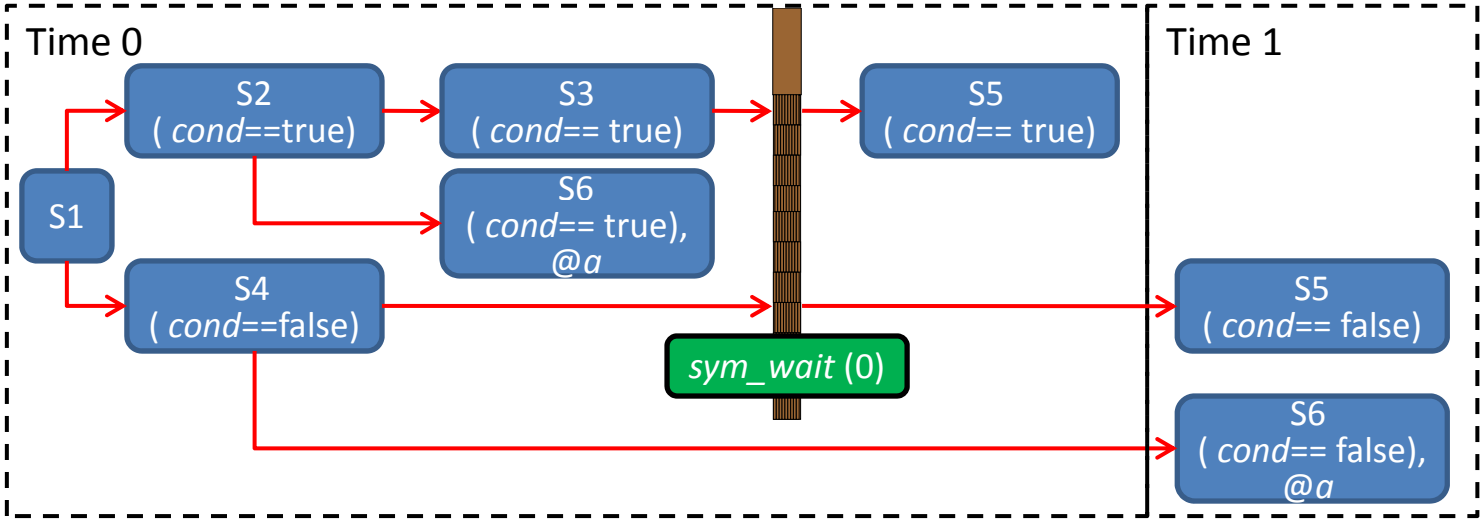
Greatly reduce the use of formal analysis and achieve better performance

Merging symbolic traces using *sym_wait*

Sym_wait can be easily embedded in design/testbench, providing a more efficient way to perform symbolic simulation

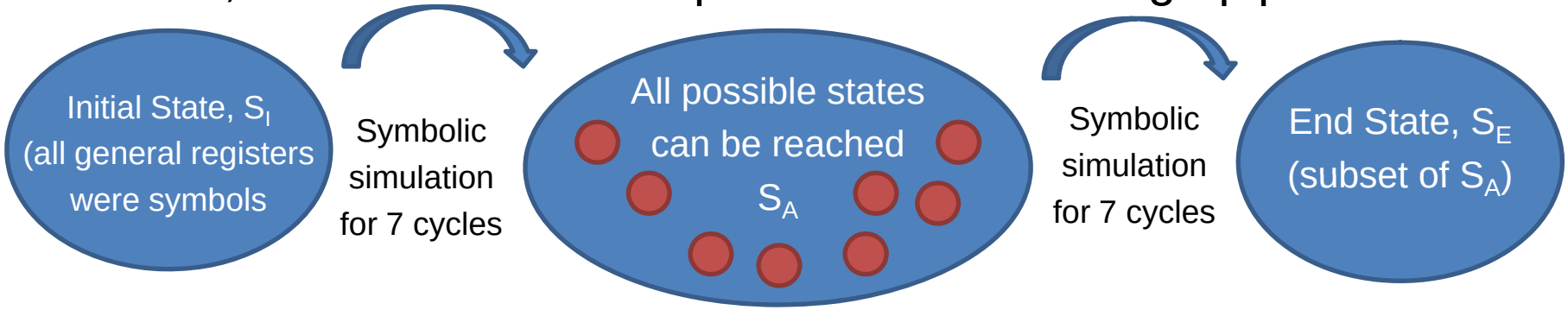
```
        always @ ( cond, i1, i2 ) begin
S1:     if (cond) begin
S2:       a = i1;
S3:       $display("Important debugging message");
        end else
S4:       #1 a = i2;
S5:     $display("Execution finished");    end
S6:     always @(a)    b = ~a;
```

sym_wait (0)



Ensuring the Correctness of Optimizations

- Symbolic simulation can only ensure the correctness and the completeness of verification within the simulated cycles
- Proof by induction
  - if the state in the last simulated cycle is a subset of any state before the last cycle, then the properties verified to hold for the simulated cycles will hold forever
- Example: symbolic code-statement reachability analysis for DLX, a 32-bit RISC microprocessor with 5-stage pipeline



Symbolic simulation for 14 cycles is enough to ensure the correctness of customized design

Case Study: Crossbar switch

- Characteristics
  - Contains two input ports and two output ports
  - Forwards a packet from any input to any output based on its priority bit and a round-robin arbitration scheme
- Performance issue of symbolic simulation
  - Different payload sizes require different numbers of cycles to transmit

Simulation traces for different delays can not be merged, degrading the efficiency of symbolic simulation

- Effectiveness of *sym_wait*

Connects a driver and a receiver through a FIFO

	Run time	Memory Consumption
Without *sym_wait*	1525	394
With *sym_wait*	442	219
Reduction Ratio	**71%**	**44%**

Case Study: DLX Processor

- Characteristics
  - A 32-bit RISC microprocessor with 5-stage pipeline
  - A simplified version of the MIPS architecture and provides a good reference point for verification

Instruction allowed (DLX)	Run time	#Cond. blocks	#Cells	Reduction ratio	Timing slack
All	**--**	**274**	**13902**	**--**	**2659 ps**
NOP	1 sec	122	2426	81.4%	1248 ps
ADD, ADDI, NOP	100 min	148	7793	68%	1563 ps
ADD, ADDI, SW, LW, NOP	103 min	168	9240	29.4%	1606ps
ADD, ADDI, SW, LW, SRL, SLL, SRA, BEQ, NOP	97 min	176	11170	14.7%	2306 ps
ADD, ADDI, AND, ANDI, XOR, SLT, SLTI, SW, LW, SRL, SLL, SRA, BEQ, BNE, J, JAL, NOP	138 min	208	12661	3.3%	2596 ps

Case Study: Alpha Processor

- Characteristics
  - A processor which includes 64-bit registers, instructions and datapaths

Instruction allowed (DLX)	Run time	#Cond. Blocks	#Cells	Reduction ratio	Timing slack
All	**--**	**175**	**31381**	**--**	**3269 ps**
NOP	1 sec	98	1339	95.7%	747 ps
ADDQ, MULQ, CMPEQ, NOP	25 min	120	27941	10.9%	3240 ps
ADDQ, MULQ, CMPEQ, LDQ, STQ, NOP	24.5 min	127	28300	9.8%	3280 ps
ADDQ, MULQ, CMPEQ, LDQ, STQ, JMP, BSR, SRL, SLL, SRA, NOP	18.5 min	142	30265	3.7%	3268 ps
ADDQ, SUBQ, MULQ, MPEQ, CMPULE, LDQ, STQ, JMP, RET, BSR, SRL, SLL, SRA, AND, BIS, XOR, NOP	15.5 min	149	32195	-2.6%	3298 ps
ADDQ, SUBQ, CMPEQ, CMPULE, LDQ, STQ, JMP, RET, BSR, SRL, SLL, SRA, AND, BIS, XOR, NOP	15.5 min	146	20476	34.7%	1848 ps

Conclusions

- Code-statement reachability analysis using high-level symbolic simulation
  - More accurate than logic simulation
  - More scalable and effective than gate-level approaches
- *Sym_wait* to accelerate symbolic simulation
  - Easily embedded in design/testbench
  - Provides additional checks for latency-related problems
  - Avoids trace explosion problem
- Methodology that can reuse existing verification environments for circuit optimization
  - More flexible
  - Allows hardware/software co-optimization
  - Optimizes designs in SoC environments, both in terms of gate count and timing slack