SystemVerilog for Synthesis
Introduction for Student's Guidance

Prof. Dr. Matthias W. Fertig

Literatur


[Sutherland, 2017] Stuart Sutherland.
RTL Modeling with SystemVerilog for Simulation and Synthesis,
ISBN 9-781-5467-7634-5, 2017.

SystemVerilog enhancements

- Verilog too limited for complexity of typical IC designs
- Enhancements for modeling digital logic
- Enhancements for verifying very large and complex designs
- done by non-profit organization Accellera Systems Initiative and outside IEEE
- first release by Accellera in 2002 $\boxed{\textit{IEEE Verilog } 2001}$ (called Verilog++)
- Accellera SystemVerilog 3.1 is a set of extensions to $\boxed{\textit{IEEE } 1364-2001}$

Gate-level models

- SystemVerilog utilizues gate-level primitives
- Digital logic gates approximate silicon implementation
- SystemVerilog provides built-in gate-level primitives

  so called *User Defined Primitives (UDPs)*
- UDPs are defined in a truth table format

Syntax of gate-level primitives

```
<gate_type> <delay> <instance_name>(<outputs>,<inputs>)
```

Example: One-bit adder gate-level model

```
‘begin_keywords "1800-2012"
module gate_adder
(input wire a, b, ci,
output wire sum, co);
timeunit 1ns; timeprecision 100ps;
wire n1, n2, n3;
xor g1 (n1, a, b);
xor #1.3 g2 (sum, n1, ci);
and g3 (n2, a, b);
and g4 (n3, n1, ci);
or #(1.5,1.8) g5 (co, n2, n3);
endmodule:  gate_adder
‘end_keywords
```

Continuous assgnment

```
assign {co,sum} = a + b + ci
```

- `assign` keyword
- Represents simple combinatorial logic
- Works with vectors and bundles of data
- A *scalar* signal is one bit wide
- A *vector* is a signal with more than one bit
- `{..}` concatenates signals and assigns according to the signal order
- `+` is the add-operator

Procedural blocks

`always`, `always_comb`, `always_ff` and `always_latch`

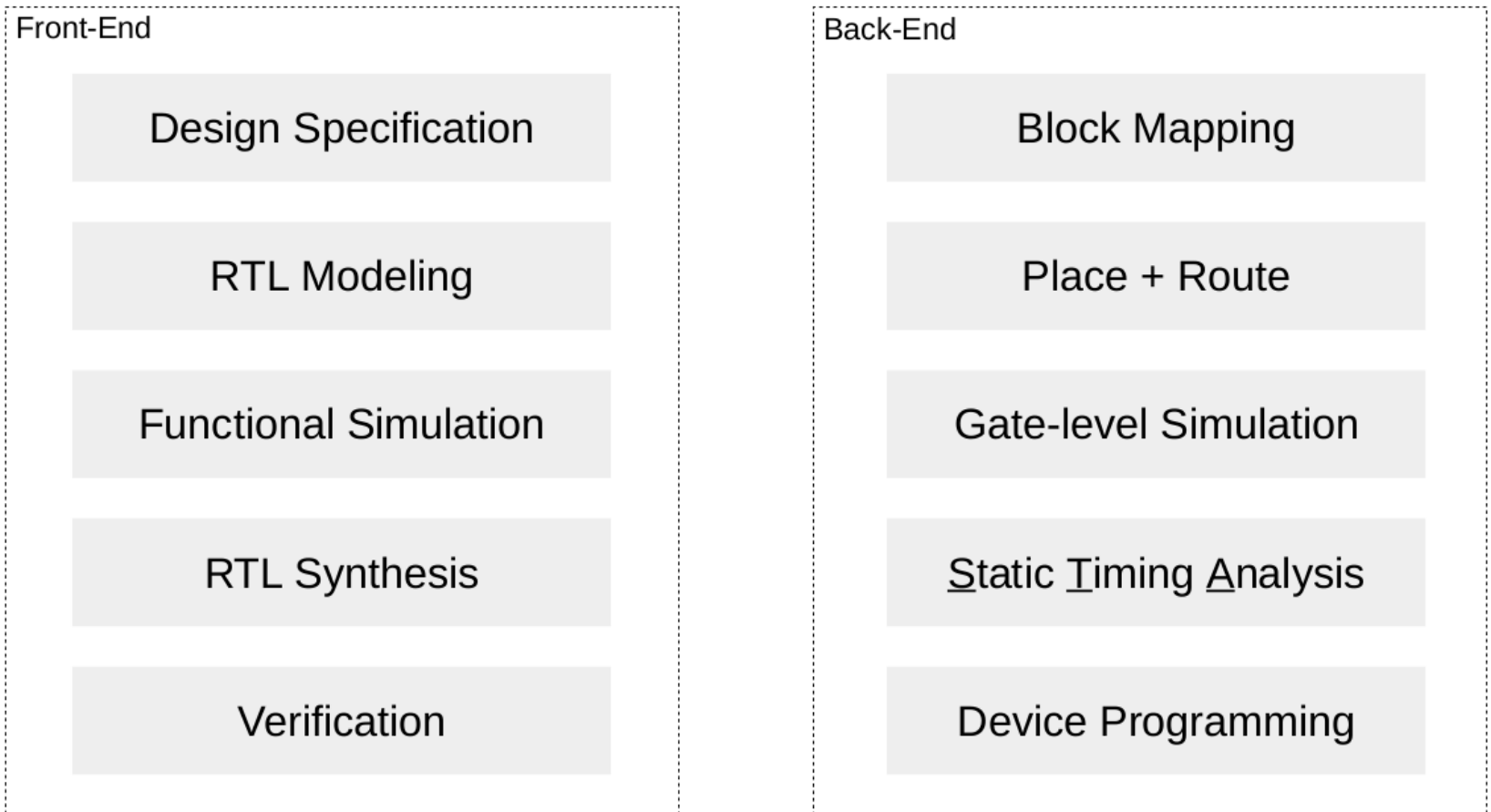
- Encapsulates one or more lines of programming statements
- Information when the statements should be executed
- example for positive clock edge triggered logic

```
always_ff @ ( posedge clk ) begin

...

end
```

Example: One-bit adder RTL model

```
‘begin_keywords "1800-2012"
module rtl_adder
(input logic a, b, ci,
output logic sum, co);
timeunit 1ns/1ns;
assign {co,sum} = a + b + ci;
endmodule:  rtl_adder
‘end_keywords
```

The FPGA design flow

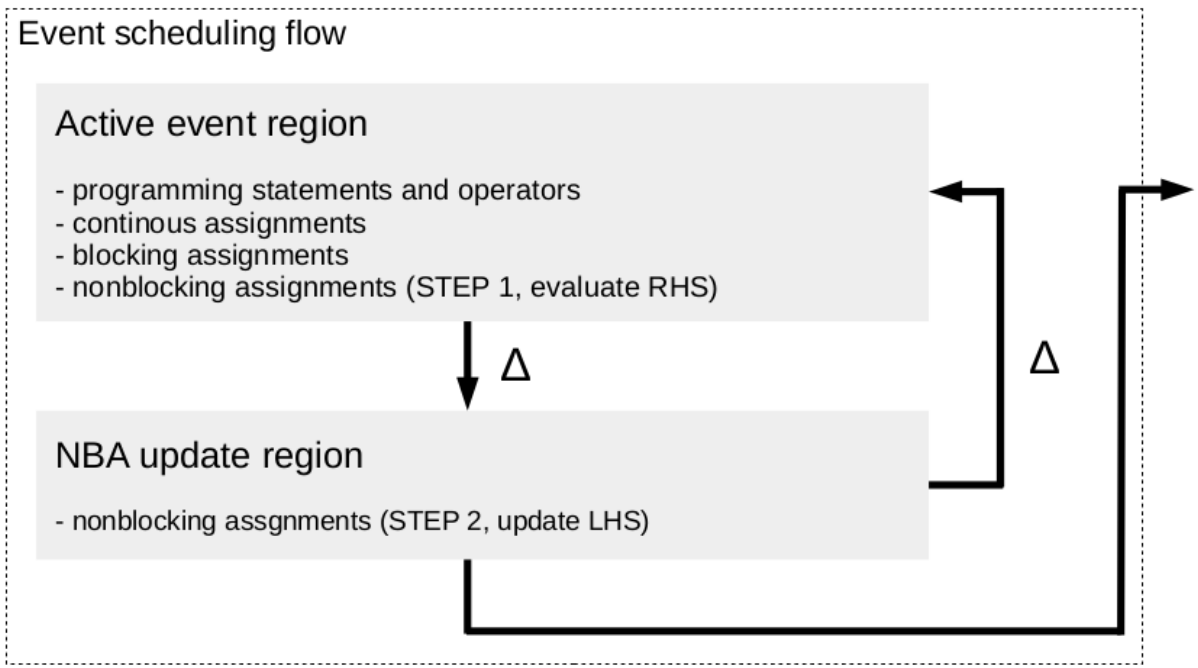


Best Practice Guideline

Use packages for shared declarations
and not the $unit declaration space.

Event regions

- simulators cannot schedule events in past time slots
- RTL models use the *active event region* and *NBA event region*



SystemVerilog Simulation and Synthesis

Non-Blocking assignment <=

- used to model sequential logic
- Finite State Machines, latches, flip-flops

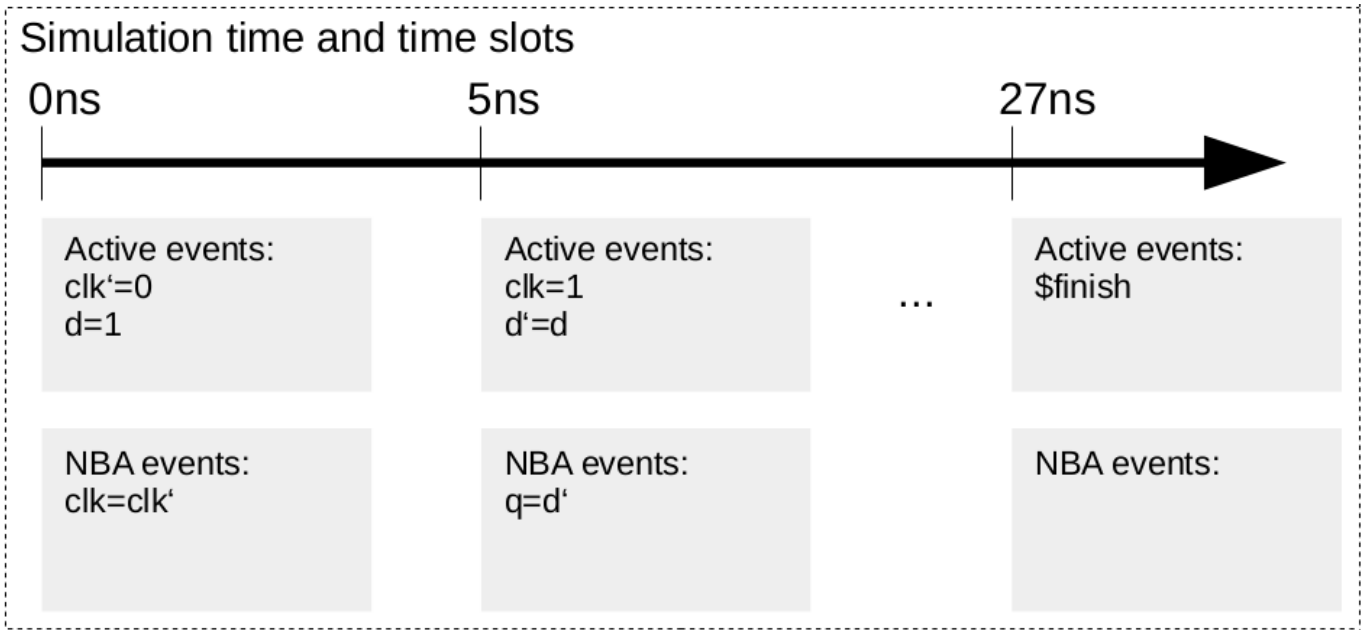
Example: 8-Bit D-Flip-Flop

```
‘begin_keywords "1800-2012"
module d_reg (input logic clock,
input logic [7:0] d,
output logic [7:0] q );
timeunit 1ns/1ns;
always @(posedge clock)
q <= d;
endmodule:  d_reg
‘end_keywords
```

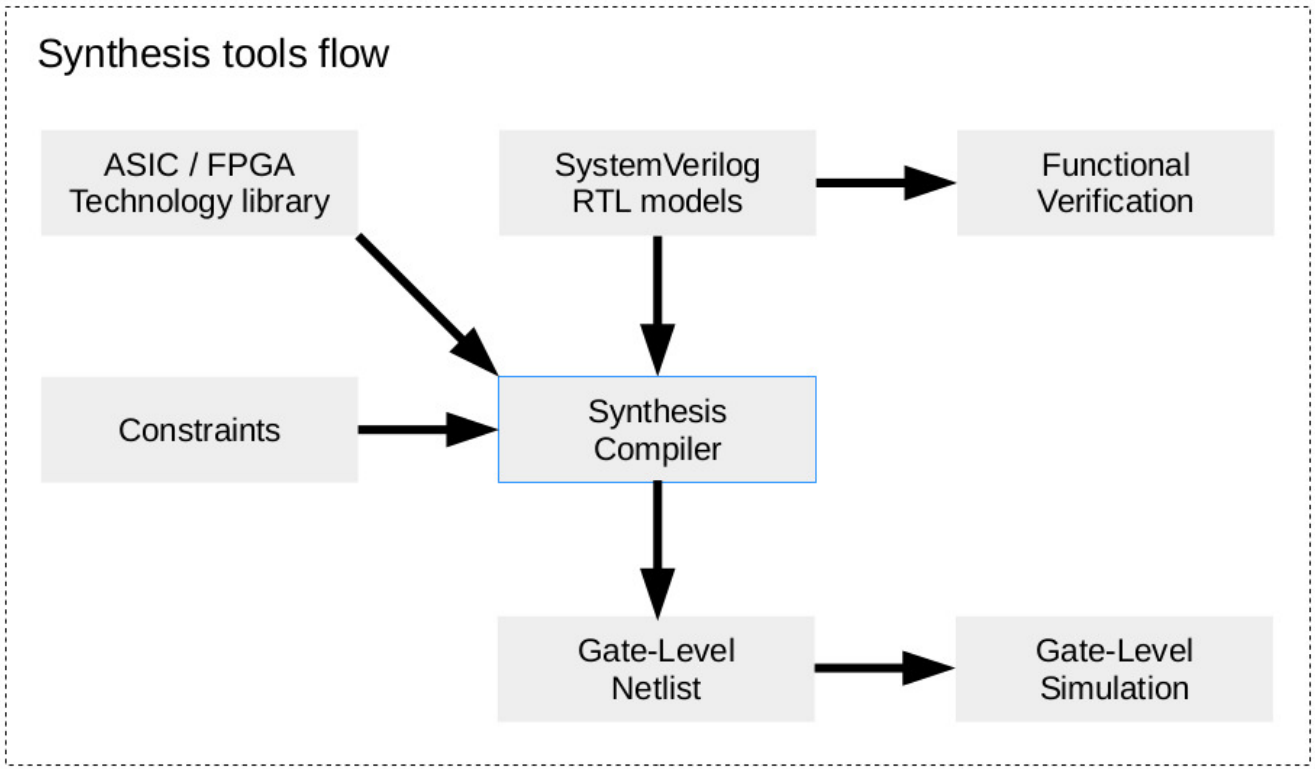
Example: 8-Bit D-Flip-Flop testbench

```
‘begin_keywords "1800-2012"
module test (input logic
clock,
output logic [7:0] d,
input logic [7:0] q);
timeunit 1ns/1ns;
initial begin
d = 1;
#7 d = 2;
#10 d = 3;
#10 $finish;
end
endmodule:  test
‘end_keywords
```

Example: 8-Bit D-Flip-Flop event scheduling



Example: Synthesis tool flow



Synthesis compiler input information

- RTL *models*
  - written by design engineer
  - description of functional behaviour
  - to be reaslized as ASIC or on FPGA
- technology *library*
  - for the target ASIC or FPGA
- synthesis *constraints*
  - defined by design engineer
  - compiler directives for placement, cell usage etc.
  - timing, area and power requirements

Synthesis compiler output information

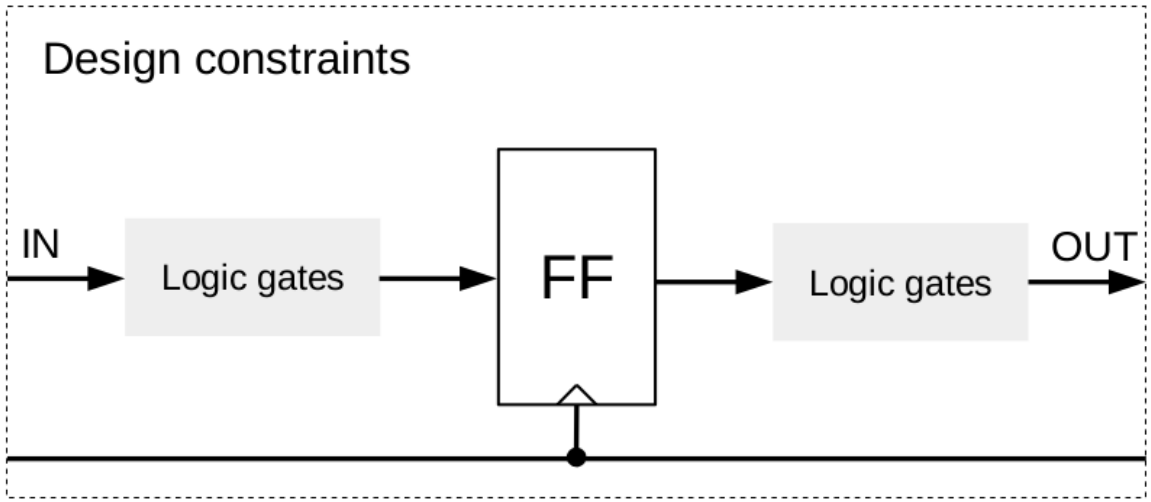
- Gate-level *netlist*
  - list of components and wires (nets)
  - components are standard cells
  - EDIF, VHDL, Verilog-2001 or SystemVerilog format
- *area* and *congestion* information
  - derived from standard cell placements
- *power* information
  - obtained from standard cell numbers and electrics
- *timing* and *clock − speed* information
  - obtained from placement and routing, flip-flop timing and static timing analysis

SystemVerilog synthesis compilers

- Commercial synthesis compilers
  - Cadence
  - Mentor Graphics
  - Synopsis
- Proprietary synthesis compilers
  - Xilinx
  - Intel (formerly Alterra)

Only a subset of the SystemVerilog language
is supported by synthesis compilers!

Example: Constraints



Timing constraints

- `IN` input arrival time relative to clock edge
- `OUT` required arrival time relative to clock edge
- `OUT` driver requirements (fan-out)

SystemVerilog Simulation and Synthesis

Lint checker

- check if RTL code meets synthesizable coding rules
- parse HDL source code and check against specific coding rules
- are configurable
- can add checks for specific coding guidelines

Logic equivalence checker

- analyze functionality of two models for logic equivalence
- compare versions of RTL models or gatelevel netlists
- compare RTL model and synthesis gatelevel netlist after *Engineering Change Orders* (ECOs)
- one type of formal verification

Synthesizable *net* type

- data type must be `logic`, explicitely or inferred
- multiple drivers resolve according to semantic rules
- all net types model silicon behaviour w/ driver strength 0–7
- not all net types exist in ASIC or FPGA synthesis compilers

Non-Synthesizable *net* type

- `uwire`, `pull0`, `pull1`, `wand`, `triand`, `wor`, `trior`, `trireg` are not supported by all synthesis compilers
- `uwire` does not permit multiple drivers
- `wand` resolves *'wired and'*, `wor` resolves *'wired or'*,

Best Practice Guideline

Use `logic` data type to connect components with single-driver signals and `wire` or `tri` net types only if multiple drivers are allowed.

Example on inferred `wire` net declarations

```
module mixed_rtl_gate_adder(
input a, // implicit net, 4-state
input logic b, // implicit net, 4-state
input reg ci, // implicit net, 4-state
output s, // implicit net, 4-state
output logic co); // implicit variable, 4-state
xor g1 (n1, a, b); // undelcared n1, implicit wire net
xor g2 (s, n1, c1);
and g3 (n2, a, b); // undeclared n2, wire net
assign n3 = n1 & c1; // undeclared n3, wire net
always_comb begin co = n2 | n3; end
endmodule:  mixed_rtl_gate_adder
```

Best Practice Guideline

Use `` `default_net_type `` as a pair of directives to switch at beginning and back at end of compilation unit.

Example on inferred uwire net declarations

```
'default_net_type uwire;
module mixed_rtl_gate_adder(
input a, // implicit net, 4-state
input logic b, // implicit net, 4-state
input reg ci, // implicit net, 4-state
output s, // implicit net, 4-state
output logic co); // implicit variable, 4-state
xor g1 (n1, a, b); // undelcared n1, uwire net
xor g2 (s, n1, c1);
and g3 (n2, a, b); // undeclared n2, uwire net
assign n3 = n1 & c1; // undeclared n3, uwire net
always_comb begin co = n2 | n3; end
endmodule:  mixed_rtl_gate_adder
```

Combined style port lists

- puts full declaration of each port in the list parantheses
- each port declaration separated by comma
- similar ports can be declared by comma-separated list of port names
- IEEE-SystemVerilog standard refers to combined-style as *ANSI C style*

```
module alu(
input wire logic signed [31:0] a,b,
input wire logic [ 3:0] opcode,
output var logic signed [31:0] result,
output wire logic signed overflow,
output wire logic signed err);
```

Synthesis considerations

- tasks and functions to be defined `automatic`
- no static variables
- synthesis will create copies of tasks and functions for modules or interfaces
- `automatic` will allocate new storage on each function or task call
- synthesis does not support variable declarations in packages for the same storage considerations

RTL Expression Operators

Operator expression rules

Don't care optimisim

- X-optimism operators generate a known result in case of one or more operand bits X or Z
- X-pessimism operators cause all bits X in case of one or more operand bits X or Z
- optimism applies when simulation can correctly predict result

```
// operator optimism
assign a = 4'b01zx; // some bits z and x
assign b = 4'b1111; // all bits one
assign c = a & b; // bitwise OR of a and b, c = 1111
assign d = a | b; // bitwise AND of a and b, d = 11xx
```

Concatenate and replicate

- Concatenate and replicate operators join expressions to form vector expressions
- Total sum of bits is sum of individual bits in the expression
- *'Replicated contatenations'* join expressions `{m,n}` and replicate `r{}` a specific number of times, i.e. `{r{m,n}}`
- *'Simple contatenations'* join expressions only, i.e. `{m,n}`

```
logic [3:0] a = 4'b0101;
logic [7:0] b = 8'hAB;
// {a,b} yields 12-bit value 12'h5AB
// {4'hF,a} yields 8-bit value 8'F5
// {8{2'b10}} yields 16-bit value 16'b1010101010101010
// {{4{a[3]}},a} yields 8-bit value 8'b00000101
```

RTL Expression Operators

Concatenate and replicate operators

Consider that ...

Concatenate and replicate operators
are synthesizable.

RTL Expression Operators

Conditional operator

Example: Multiplexed 4-Bit D-Register

```
// 4-bit register with multiplexed D input,
// using conditional operator.
`begin_keywords "1800-2012"
module muxed_register
#( parameter WIDTH = 4 ) // register size
( input logic clk, // 1-bit input
input logic data_select, // 1-bit input
input logic [WIDTH-1:0] d1, d2, // scalable in
output logic [WIDTH-1:0] q_out ); // scalable out
timeunit 1ns; timeprecision 1ns;
always_ff @(posedge clk) q_out <= data_select ?  d1:d2;
endmodule:  muxed_register `end_keywords
```

RTL Expression Operators

Conditional operator

Example: 4-Bit adder with tri-state output

```
module tri_state_adder
#( parameter WIDTH = 4 ) // register size
( input logic clk, // 1-bit input
input logic ena, // 1-bit input
input logic [WIDTH-1:0] a, b, // scalable in
output tri logic [WIDTH-1:0] out );  // tri-state net
assign out = ena ?  ( a + b ) :  'z;
endmodule:  tri_state_adder
```

- Conditional operator `? :` selects if out is assigned sum or high-impedance z
- `'z` is literal value that sets all bits of an expression to high-impedance

RTL Expression Operators

Reduction operators

Example: XOR parity checker

```
(input data_t data_in,
input clk,
input rstN,
output logic error);

always_ff @(posedge clk or negedge rstN) // async res
if (!rstN) error <= 0; // active-low reset
else error <= ^ {data_in.parity_bit, data_in.data};
// reduction-XOR returns 1 if an odd number of bits
// are set in the combined data and parity_bit
endmodule:  parity_checker

`end_keywords
```

Set membership `inside` operator

- List members can change during simulation
- List members can be expressions like other variables or nets

```
always_comb begin
exp_spec <= exp inside {exp_inf, exp_zero};
end
```

- Set of values can be stored in array

```
always_comb begin
is_prime <= data inside {PRIMES};
end // PRIMES is array of prime numbers
```

- Operator can be used in continuous assignments, i.e. `assign`

```
assign is_prime = data inside {PRIMES};
```

Best Practice Guideline

Let synthesis do its job!

Modern tools recognize barrel shifters etc.

There is no benefit to optimize at RTL-level.

RTL Expression Operators

Assignment Operators

Blocking assignment `=`

- Combinatorial logic
- Multiplexer, decoder, comparators, ...

Continuous assignment `<=`

- Sequential logic
- Flip-Flops, Finite State Machines, Registers, Counters, Pipelines, ...

Example: Assignment Operators

```
typedef enum logic [1:0]
{ AND_OP, ...  } op_t;

...
always_comb begin
out = in;
case (op)
AND_OP : res &= b;
...
endcase
end
```

RTL Expression Operators

Operator precedence

Three exceptions of Operator Associativity

- `?:` `->` `<->`

Example 1

```
assign S = A + B - C;
```

`A` is added to `B` before `C` is subtracted.

Example 2

```
assign C = A + B ** 4;
```

Evaluate `B**4` first before `A` is added.

Example 3

```
assign C = (A + B) ** 4;
```

Evaluate `A+B` first before `**4` is performed.

RTL Programming Statements

System Verilog procedural blocks

The programming statements discussed in this module are appropriate for RTL modeling!

Procedural blocks

- *Procedural block* is a container for programming statements
- Procedural block controls when statements are executed, e.g. rising edge
- *Initial procedure* and *always procedure*
- Initial procedure is verification-specific and not for synthesis but can be used for `$readmemb` or `$readmemh`
- Always procedure is infinite loop to model continuous behaviour of hardware
- Four types of *always procedure*: `always`, `always_ff`, `always_comb`, `always_latch`

RTL Programming Statements

System Verilog procedural blocks

General purpose always procedure

- `always`
- Tools do not know when the intended usage is for synthesizable RTL models
- Synthesis has coding restrictions to general always procedures

Specialized RTL always procedure

- `always_ff`, `always_comb`, `always_latch`
- Coding restrictions for synthesis ensure RTL and gatel-level behaviour of ASIC or FPGA match
- `always_ff` for modeling sequential logic, i.e. flip-flops
- `always_comb` for modeling combinatorial logic, i.e. decoders
- `always_latch` for modeling latched behaviour, i.e. auto sensitivity list

RTL Programming Statements

System Verilog procedural blocks

Combinatorial logic sensitivity

- Programming statements to be evaluated whenever input changes
- Explicit or inferred sensitivity list possible
- `always_comb` infers sensitivity list

```
always@(a,b) S = A + B; // expl sensitivity list
```

```
always_comb S = A + B; // impl sensitivity list
```

Latched logic sensitivity

- `always_latch` infers sensitivity list for logic blocks that can store their logic state

```
always_latch
if( en ) O <= S; // infers a latch as well
// implicit sensitivity list is (en, S)
```

RTL Programming Statements

Looping Statements

For loops

`for ( initial_ass ; end_expr ; step_ass ) statement/statement group`

- *Initial assignment* executes once at beginning of loop
- *End expression* evaluated every iteration, loop terminates if compare returns *true*
- *Step assignment* executed after every iteration, end expression is evaluated then
- *Statement/statement group* executed every iteration, after the end expression is evaluated

```
parameter N = 6;
logic [N-1:0] a, b, y;
always_comb begin
for( int i=0; i<N; i++ ) y[i] = a[i] ^ b[(N-1)-i];
end
```

For loop synthesis

- Synthesis compilers perform a loop unrolling

```
always_comb begin
y[0] = a[0] ^ b[3];
y[1] = a[1] ^ b[2];
y[2] = a[2] ^ b[1];
y[3] = a[3] ^ b[0];
end
```

- Number of iterations must be a fixed number so that synthesis can unroll the loop
- For loops are of advantage for large bus assignments to reduce code length

Intentional and unintentional latches

Synthesis compilers will infer a latch to match simulation behaviour of value retention. Latches ensure that gatel-level logic meets simulation behaviour.

Unintentional latch inference (2)

- Inadvertent latches in state machine models for unnused states (A) or one-hot encoding (B)

A.
```
always_comb begin
case (state)
2'b00:  q = S0;
2'b01:  q = S1;
2'b10:  q = S2;
endcase
end
```

B.
```
always_comb begin
case (1'b1)
state[0]:  q = S0;
state[1]:  q = S1;
state[2]:  q = S2;
endcase
end
```

Intentional and unintentional latches

Option 1 - Fully implemented vs. reduced decision logic

- Fully implemented means all states decoded, i.e. `2'b11` in (A)
- Reduced decision logic removes logic for `2'b11` from (A)
- Unwanted behaviour if logic hazards/glitches causing `2'b11`
- Design more robust with fully specified decision logic

A.

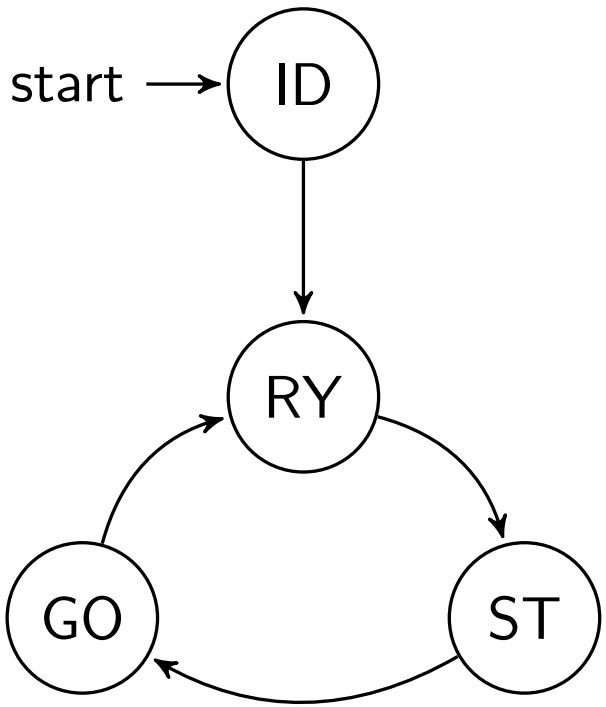
```
always_comb begin
case (state)
2'b00:  q = S0;
2'b01:  q = S1;
2'b10:  q = S2;
default:  q = ERR;
endcase
end
```

B.

```
always_comb begin
case (1'b1)
state[0]:  q = S0;
state[1]:  q = S1;
state[2]:  q = S2;
default:  q = ERR;
endcase
end
```

Intentional and unintentional latches

Default branch latch avoidance (Option 1)




```
typedef enum logic [2:0]
{ID=3'b000, RY=3'b001,
ST=3'b011, GO=3'b111}
states_t;
```

```
states_t state, next;

always_comb begin
case (state)
ID : blue = RY;
RY : blue = ST;
ST : blue = GO;
GO : blue = RY;
default:  next = ID;
endcase
end

// NO inferred latches
// for next!
```