SystemVerilog Symposium
Track I: SystemVerilog Basic Training

Copyright notice

Sunburst Design, Inc.

14314 SW Allen Blvd., PMB 501

Beaverton, OR 97005 USA

Voice: 503-641-8446

FAX: 503-641-8486

e-mail: cliffc@sunburst-design.com

web: www.sunburst-design.com

Agenda

• 9:00 - 10:15 am	SystemVerilog Overview & Methodology
	SystemVerilog Design
• 10:15 - 10:30 am	** *BREAK* **
• 10:30 - 11:30 am	SystemVerilog Design & Verification
	SystemVerilog DPI & Assertions
• 11:30 - 1:30 pm	*LUNCH* / Accellera SystemVerilog Update
• 1:30 - 3:30 pm	EDA Vendor Fair

SystemVerilog References

- www.systemverilog.org web site
  - SystemVerilog 3.1 LRM
  - DAC 2003 presentations

Download a copy - free!

Some of the slides in this presentation came from the DAC 2003 presentations

- FSM Design Techniques Using SystemVerilog 3.0

  `www.sunburst-design.com/papers`

- Assertion-Based Design by Foster, Krolnik & Lacey

- Pending Publications
  - SystemVerilog by Sutherland, Davidmann & Flake
  - The Art of Verification with SystemVerilog by Haque, Khan & Michelson

    (same authors as "The Art of Verification with Vera")

SystemVerilog Methodology & Overview

History of the Verilog HDL

Reference Material

- 1984: Gateway Design Automation introduced Verilog
- 1989: Gateway merged into Cadence Design Systems
- 1990: Cadence put Verilog HDL into the public domain
- 1993: OVI enhanced the Verilog language - *not well accepted*
- 1995: IEEE standardized the Verilog HDL (IEEE 1364-1995)
- 2001: IEEE standardized the Verilog IEEE Std1364-2001
- 2002: IEEE standardized the Verilog IEEE Std1364.1-2002

  RTL synthesis subset
- 2002: Accellera standardized SystemVerilog 3.0
  - Accellera is the merged replacement of OVI & VHDL International (VI)
- 2003: Accellera standardized SystemVerilog 3.1
- 200?: IEEE Verilog with SystemVerilog enhancements

Why Call It SystemVerilog 3.x?

- SystemVerilog is *revolutionary evolution* of Verilog
- Verilog 1.0 - IEEE 1364-1995 "Verilog-1995" standard
  - The first IEEE Verilog standard
- Verilog 2.0 - IEEE 1364-2001 "Verilog-2001" standard
  - The second generation IEEE Verilog standard
  - Significant enhancements over Verilog-1995
- SystemVerilog 3.x - Accellera extensions to Verilog-2001
  - A third generation Verilog standard
  - DAC-2002 - SystemVerilog 3.0
  - DAC-2003 - SystemVerilog 3.1

The intention is to donate SystemVerilog to the IEEE Verilog committee for IEEE standardization

SystemVerilog Superset of Verilog-2001

SystemVerilog

from C / C++

verification

assertions	mailboxes		
test program blocks	semaphores	classes	dynamic arrays
clocking domains	constrained random values	inheritance	associative arrays
process control	direct C function calls	strings	references

modeling

interfaces	dynamic processes	int	globals	break
nested hierarchy	2-state modeling	shortint	enum	continue
unrestricted ports	packed arrays	longint	typedef	return
implicit port connections	array assignments	byte	structures	do-while
enhanced literals	enhanced event control	shortreal	unions	++ -- += -= *= /=
time values & units	unique/priority case/if	void	casting	>>= <<= >>>= <<<=
logic-specific processes	root name space access	alias	const	&= |= ^= %=

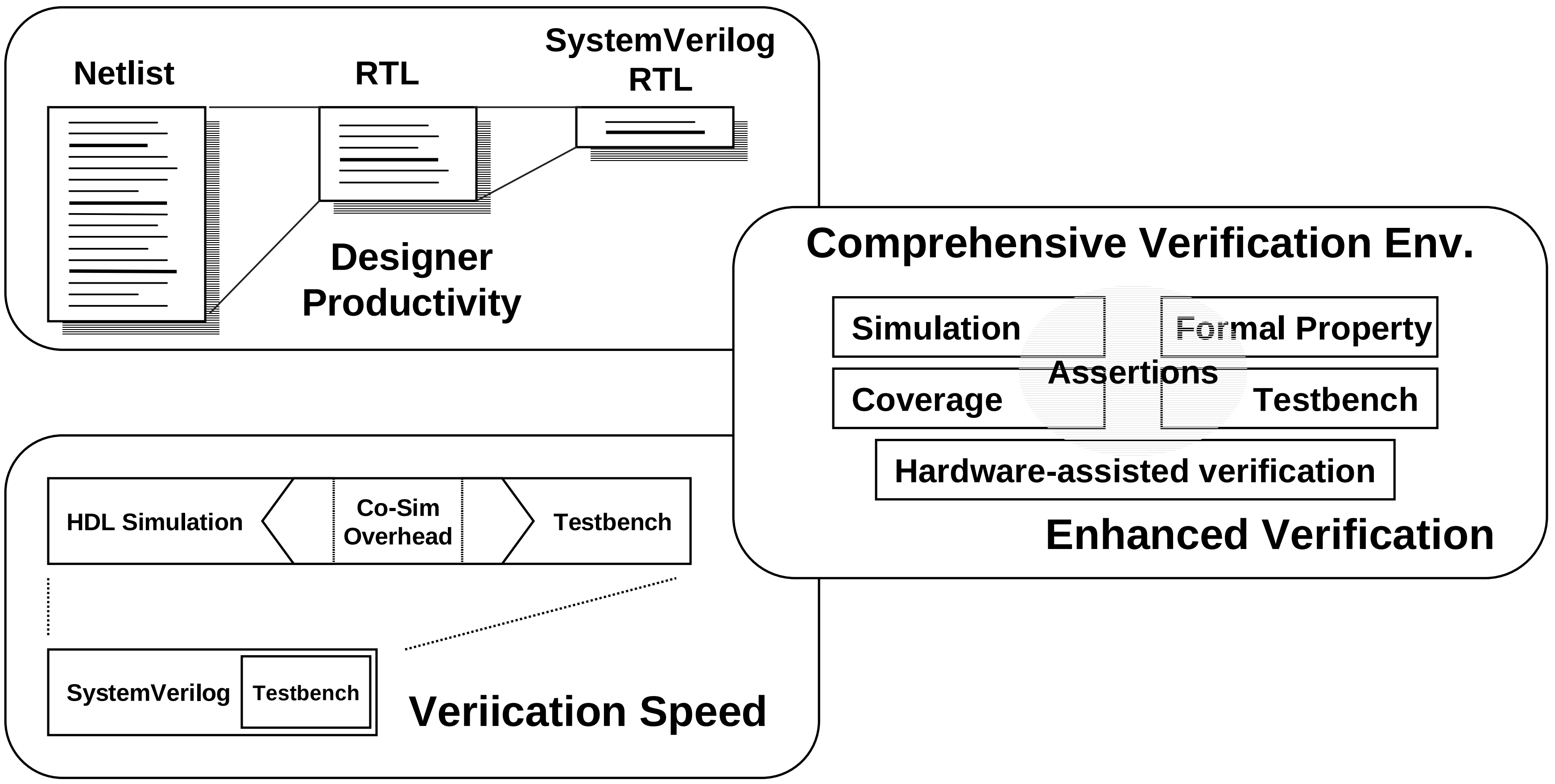
Verilog-2001

ANSI C style ports	standard file I/O	(* attributes *)	multi dimensional arrays
generate	$value$plusargs	configurations	signed types
localparam	`ifndef `elsif `line	memory part selects	automatic
constant functions	@*	variable part select	** (power operator)

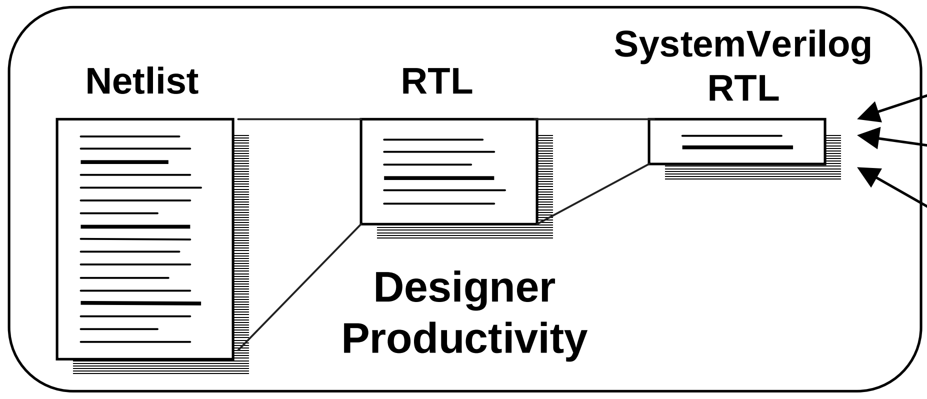
Verilog-1995

modules	$finish $fopen $fclose	initial	wire reg	begin–end	+ = * /
parameters	$display $write	disable	integer real	while	%
function/task	$monitor	events	time	for forever	>> <<
always @	`define `ifdef `else	wait # @	packed arrays	if-else	
assign	`include `timescale	fork–join	2D memory	repeat	

SystemVerilog Means Productivity



Increasing Designer Productivity



Structures and user defined data types

Interfaces to encapsulate communication

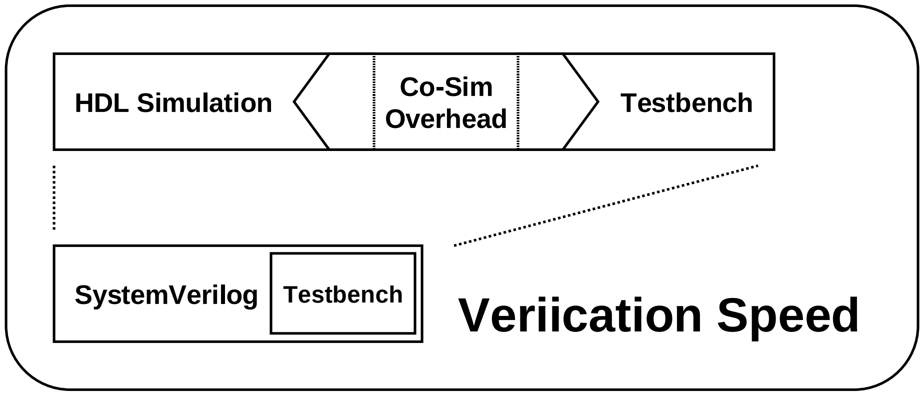
. * implicit port instantiation

- 2X – 5X less code to capture the same functionality
  - Less code ➔ Fewer bugs
  - Easier to interpret and communicate among teams
- Evolutionary: Reduces learning curve

New constructs to help *eliminate simulation and synthesis mismatches*

Less code & still synthesizable

Greater Verification Speed / Single Language for Design & Test



Enables faster tools and acceleration

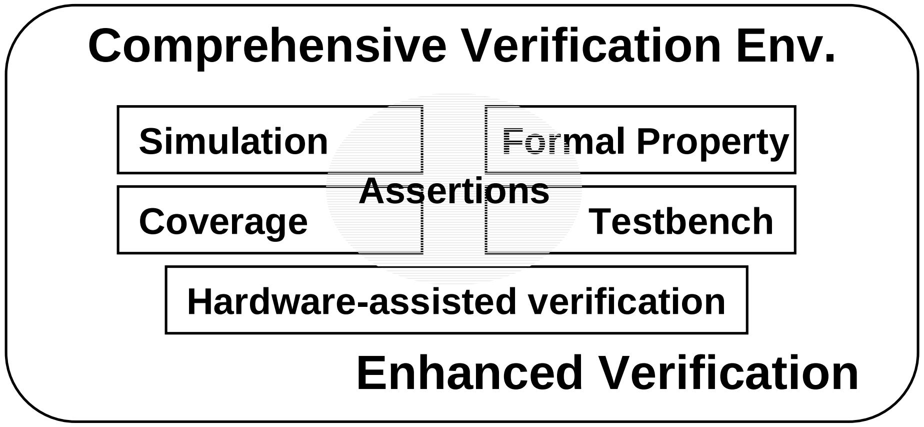
- Unified language for design and verification improves effectiveness
  - Advanced constrained-random test generation
  - Fully integrated, complete assertion technology
  - Easy integration of C Models

Easy learning curve - facilitates quicker adoption

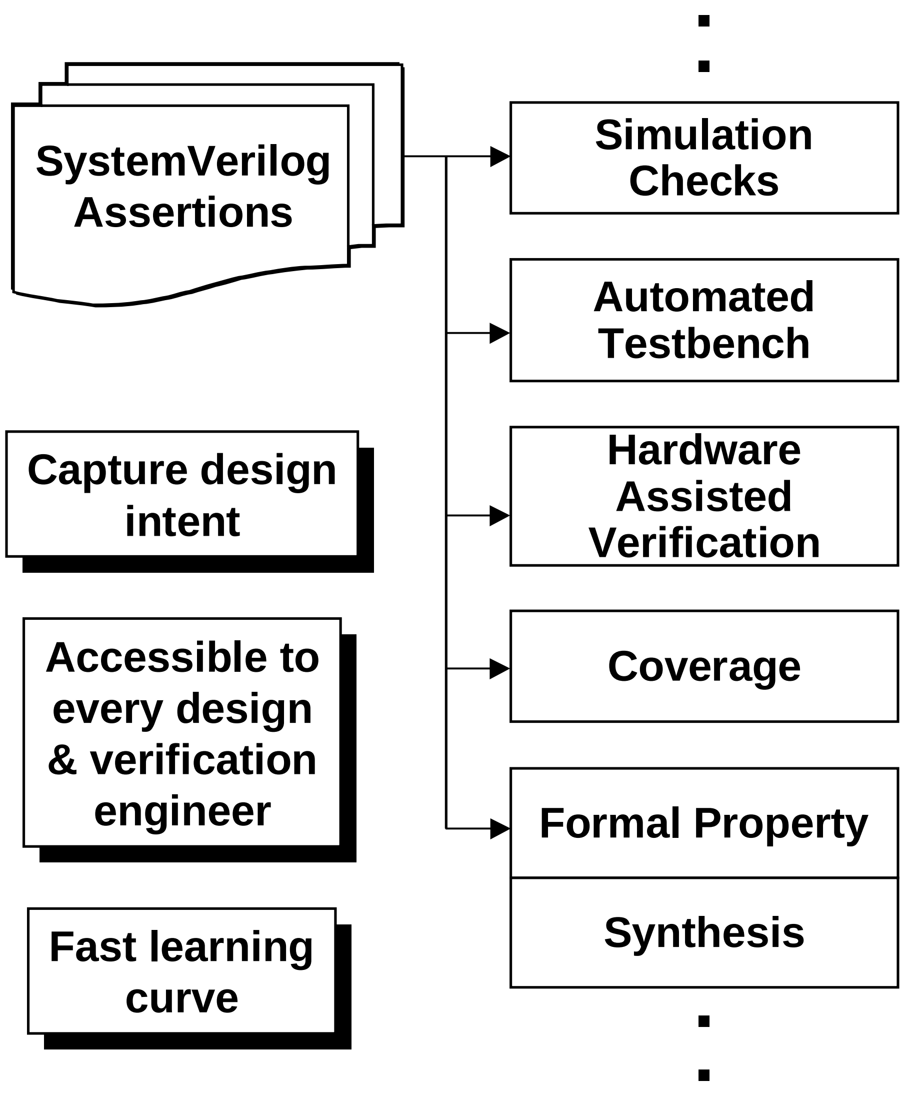
Reduces design and verification complexity with advanced constructs

Improved communication between design and verification teams

SystemVerilog Powers Comprehensive Verification and Design

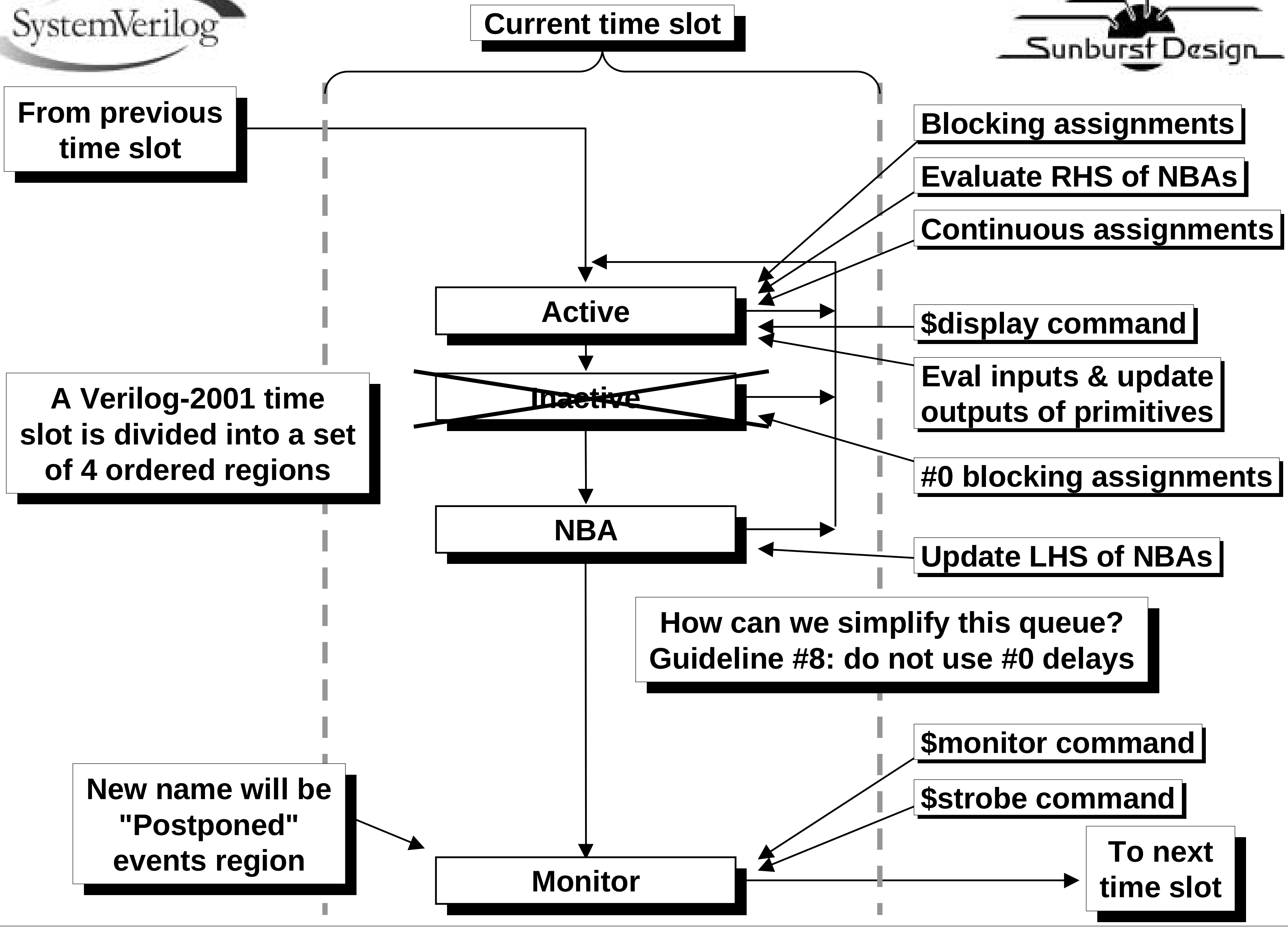


- Integrated assertions expand effectiveness of verification methodology
  - Assertion-based verification
  - Formal property verification
  - Across the board verification acceleration



SystemVerilog Design Language

Verilog-2001 Event Scheduling



A Verilog-2001 time slot is divided into a set of 4 ordered regions

How can we simplify this queue? Guideline #8: do not use #0 delays

New name will be "Postponed" events region

8 Guidelines to avoid Coding Styles that Kill!

Follow these guidelines and remove 90-100% of all Verilog race conditions

- In general, following specific coding guidelines can eliminate Verilog race conditions:

Guideline #1: Sequential logic - use nonblocking assignments

Guideline #2: Latches - use nonblocking assignments

Guideline #3: Combinational logic in an always block - use blocking assignments

Guideline #4: Mixed sequential and combinational logic in the same always block - use nonblocking assignments

Guideline #5: Do not mix blocking and nonblocking assignments in the same always block

Guideline #6: Do not make assignments to the same variable from more than one always block

Guideline #7: Use $strobe to display values that have been assigned using nonblocking assignments

Guideline #8: Do not make #0 procedural assignments

These guidelines still apply to SystemVerilog RTL designs

SystemVerilog Basic Data Types

- SystemVerilog has:
  - 4-state data types ← **0, 1, X, Z** ← **Uninitialized variables = X**
    Uninitialized nets = Z
    (same as Verilog-2001)
  - 2-state date types ← **0, 1** ← **Uninitialized variables = 0**
    Uninitialized nets* = 0
    (new to SystemVerilog)

New to SystemVerilog

```
reg      r;  // 4-state, Verilog-2001 (sizeable) data type
integer  i;  // 4-state, Verilog-2001 32-bit signed data type
logic    w;  // 4-state, (sizeable) 0, 1, X or Z

bit      b;  // 2-state, (sizeable) 1-bit 0 or 1
byte     b8; // 2-state, 8-bit signed integer
shortint s;  // 2-state, 16-bit signed integer
int      i;  // 2-state, 32-bit signed integer
longint  l;  // 2-state, 64-bit signed integer
```

```
reg   [15:0] r16;
logic [15:0] w16;
bit   [15:0] b16;
```

reg, logic & bit can be sized

Other data types have also been added

* - **bit** can be used as either a variable or a net - more later

Almost Universal Data Type - logic (or reg)

- **`logic`** is roughly equivalent to the VHDL **`std_ulogic`** type
  - Unresolved
  - Either permits only a single driving source *-OR-* procedural assignments from one or more procedural blocks
  - In SystemVerilog: **`logic`** and **`reg`** are now the same (like **`wire`** and **`tri`** in Verilog)

Illegal to make both continuous assignments and procedural assignments to the same variable

`logic` is a 4-state type

`bit` is an equivalent unresolved 2-state type

- **`wire`** net type still required for:
  - Multiply driven buses (such as bus crossbars and onehot muxes)
  - Bi-directional buses (more than one driving source)

User Defined Types - typedef

- Allows the creation of either *user-defined* or *easily-changable* type definitions
  - Good naming convention uses "**_t**" suffix

```
typedef existing_type mytype_t;
```

defines.vh

```
`ifdef STATE2
  typedef bit   bit_t; // 2-state
`else
  typedef logic bit_t; // 4-state
`endif
```

Design Strategy: Use All typedef's

- Interesting design strategy to switch easily between 4-state and 2-state simulations
  - Only use **`typedef`**-ed types

defines.vh

```
`ifdef STATE2
  typedef bit   bit_t; // 2-state
`else
  typedef logic bit_t; // 4-state
`endif
```

tb.v

```
module tb;
  bit_t q, d, clk, rst_n;

  dff u1 (.q(q), .d(d), .clk(clk),
          .rst_n(rst_n));

  initial begin
    // stimulus ...
  end
endmodule
```

dff.v

```
module dff (
  output bit_t q,
  input  bit_t d, clk, rst_n);

  always @(posedge clk)
    if (!rst_n) q <= 0;
    else        q <= d;
endmodule
```

Default 4-state simulation

```
verilog_cmd defines.vh tb.v dff.v
```

Faster 2-state simulation

```
verilog_cmd defines.vh tb.v dff.v +define+STATE2
```

Unresolved & Resolved Types

- 4-state and 2-state design strategy is easiest to use with unresolved types

Efficient type usage for VHDL `std_ulogic` equivalence

defines.vh

```
`ifdef STATE2
  typedef bit   bit_t; // 2-state
`else
  typedef logic bit_t; // 4-state
`endif
```

No easy switching between `std_ulogic` and `std_logic` equivalents

No comparable type capability for VHDL `std_logic` equivalence

No multi-driver 2-state type

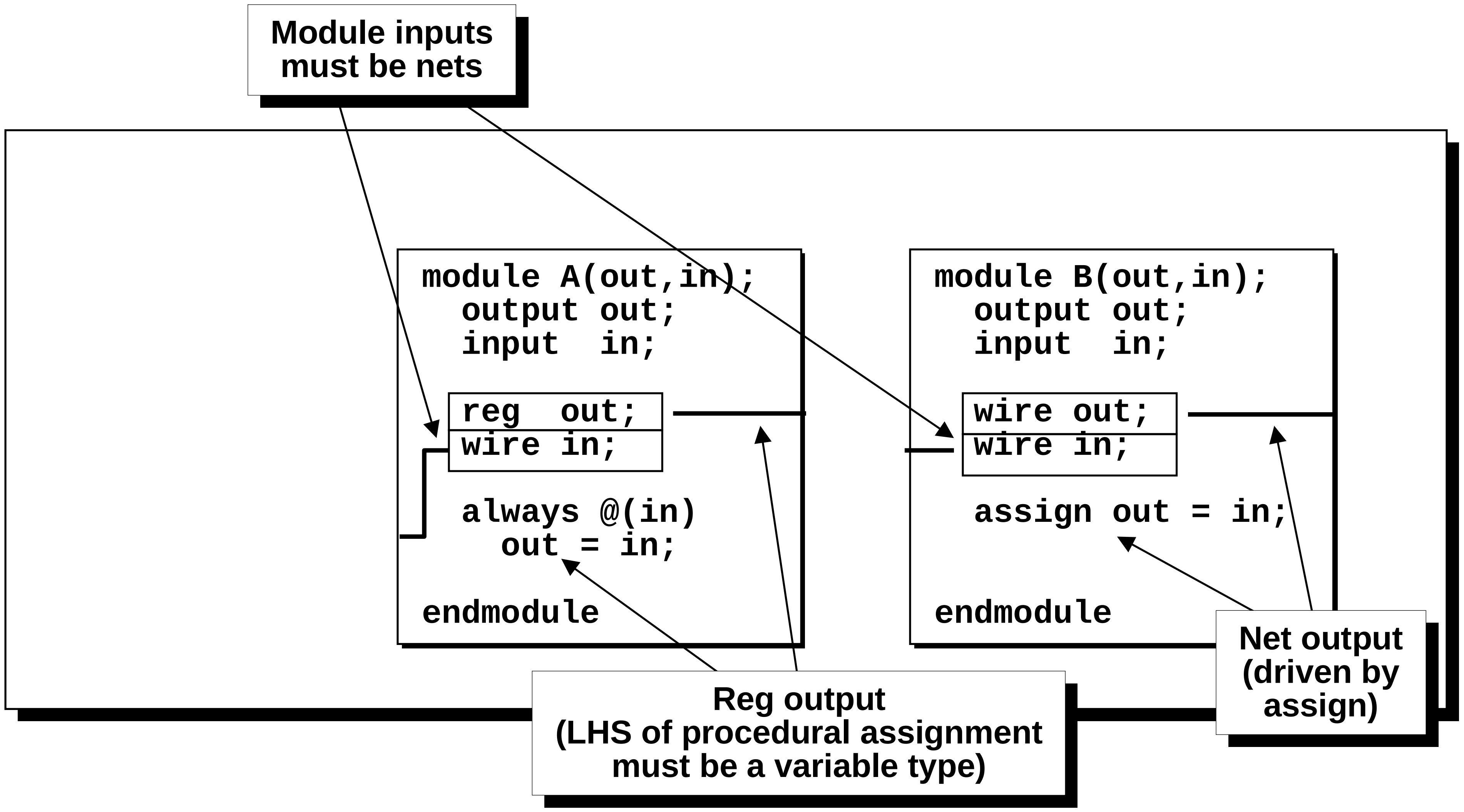
defines.vh

```
`ifdef STATE2
  typedef bit  bit_t; // 2-state
  typedef ???  tri_t; // 2-state
`else
  typedef reg  bit_t; // 4-state
  typedef wire tri_t; // 4-state
`endif
```

... maybe this will get fixed in the next version of SystemVerilog???

Verilog-2001 Data Types

(Internal to a Module)

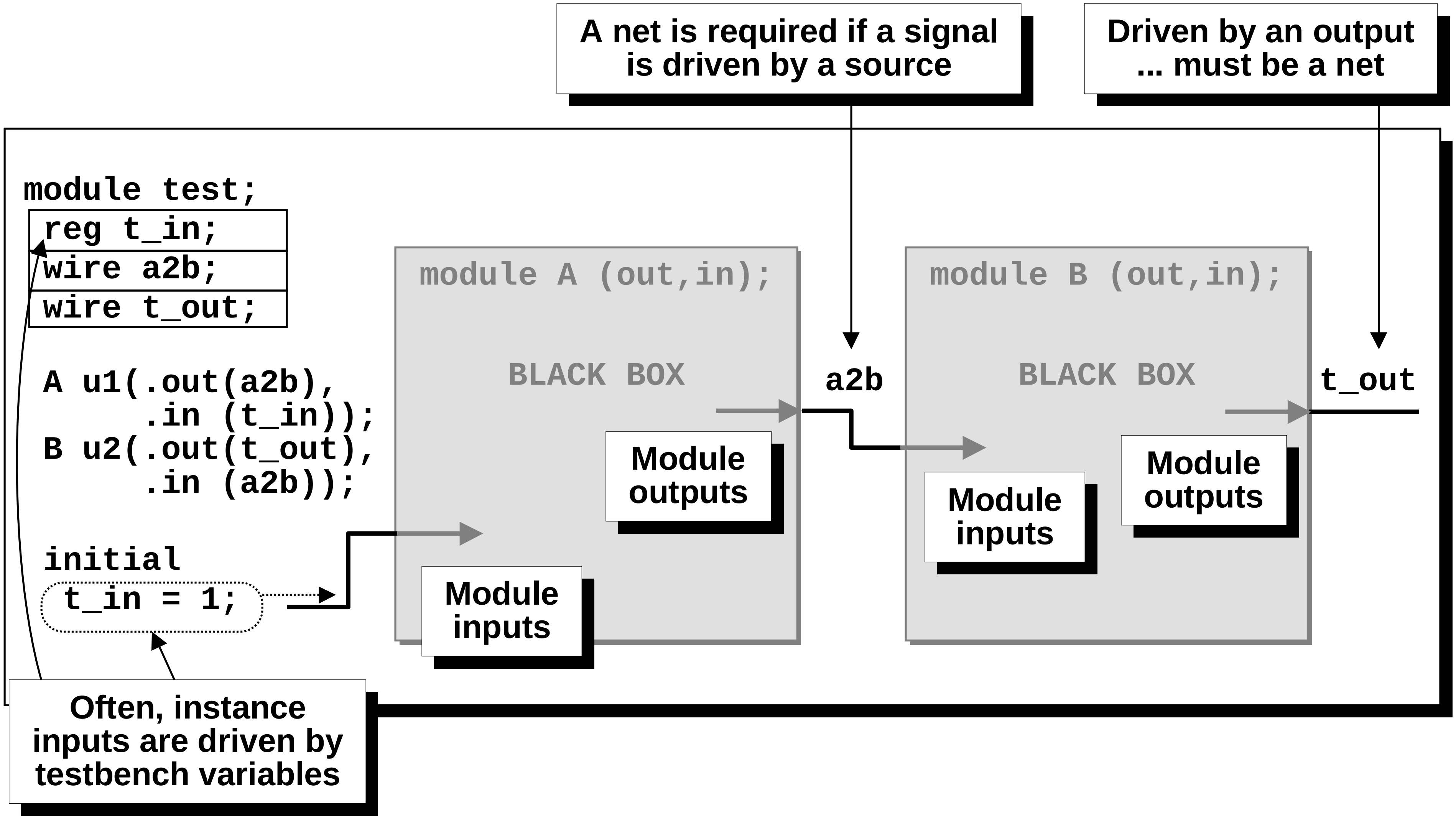

Module inputs must be nets
Reg output (LHS of procedural assignment must be a variable type)
Net output (driven by assign)

```
module A(out,in);
  output out;
  input  in;

  reg  out;
  wire in;

  always @(in)
    out = in;

endmodule
```

```
module B(out,in);
  output out;
  input  in;

  wire out;
  wire in;

  assign out = in;

endmodule
```

Verilog-2001 Data Types

(External to a Module)

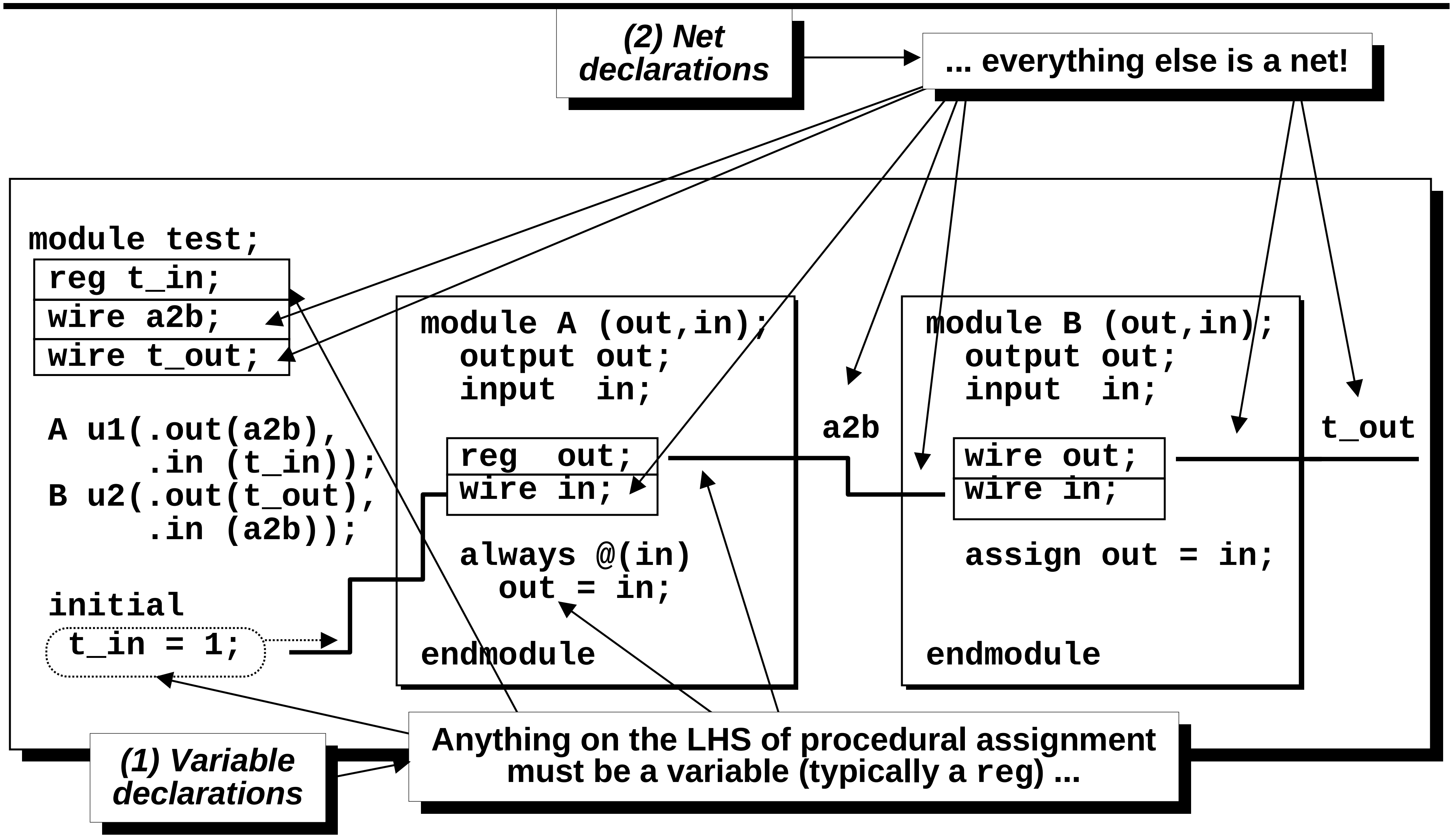


```
module test;
  reg t_in;
  wire a2b;
  wire t_out;

  A u1(.out(a2b),
       .in (t_in));
  B u2(.out(t_out),
       .in (a2b));

  initial
   t_in = 1;
```

A net is required if a signal is driven by a source
Driven by an output ... must be a net
module A (out,in);
BLACK BOX
Module outputs
Module inputs
a2b
module B (out,in);
BLACK BOX
Module inputs
Module outputs
t_out
Often, instance inputs are driven by testbench variables

Verilog-2001 Data Types

(Putting It All Together)



```
module test;
  reg t_in;
  wire a2b;
  wire t_out;

  A u1(.out(a2b),
       .in (t_in));
  B u2(.out(t_out),
       .in (a2b));

  initial
   t_in = 1;
```

```
module A (out,in);
  output out;
  input  in;

  reg  out;
  wire in;

  always @(in)
    out = in;

endmodule
```

```
module B (out,in);
  output out;
  input  in;

  wire out;
  wire in;

  assign out = in;

endmodule
```

(1) Variable declarations

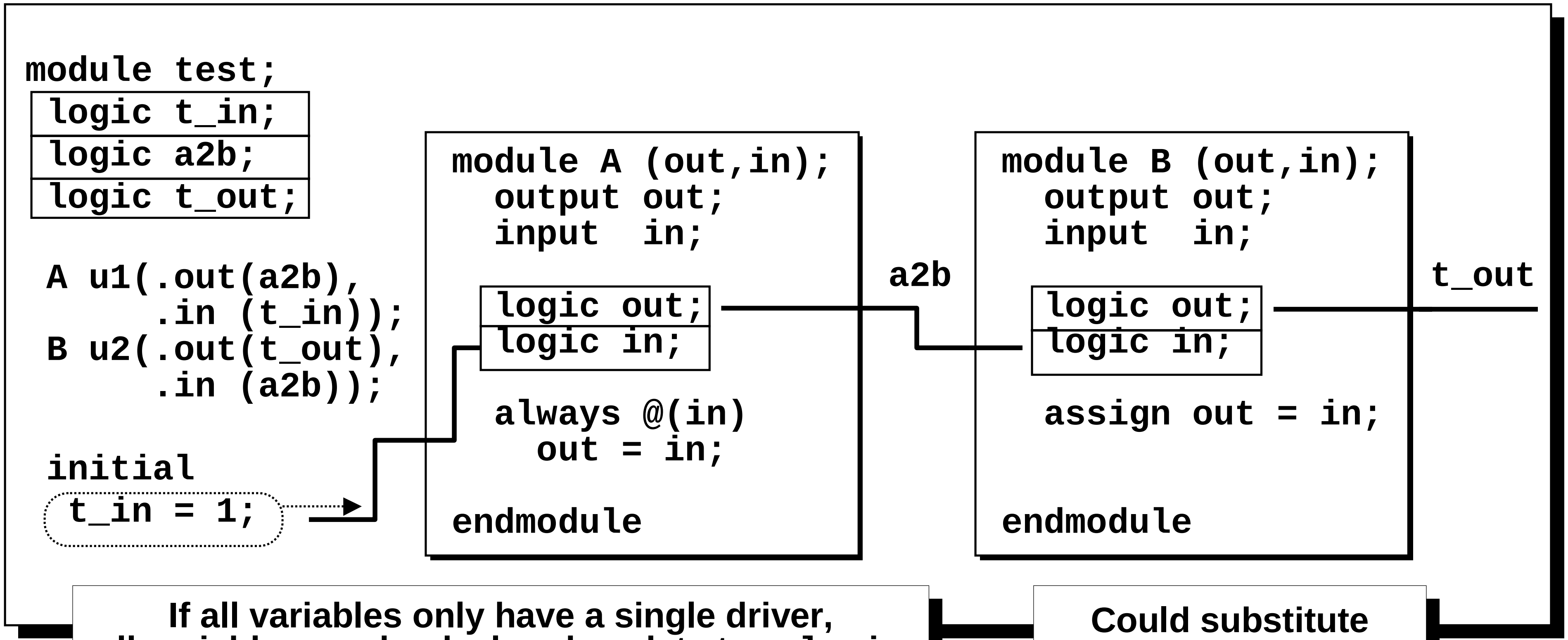
Anything on the LHS of procedural assignment must be a variable (typically a `reg`) ...

(2) Net declarations

... everything else is a net!

SystemVerilog - Unrestricted Ports

- SystemVerilog permits procedural or continuous assignments to variables



```
module test;
 logic t_in;
 logic a2b;
 logic t_out;

 A u1(.out(a2b),
      .in (t_in));
 B u2(.out(t_out),
      .in (a2b));

 initial
  t_in = 1;
```

```
module A (out,in);
  output out;
  input  in;

  logic out;
  logic in;

  always @(in)
    out = in;

endmodule
```

```
module B (out,in);
  output out;
  input  in;

  logic out;
  logic in;

  assign out = in;

endmodule
```

If all variables only have a single driver, all variables can be declared as data type `logic`

Could substitute **any type for `logic` including `reg`**

Enhanced Literal Number Syntax

```
module fsm_sv1b_3;
  ...

  always @* begin
    next = 'x;
    case (state)

  ...
endmodule
```

Useful "fill" operations similar to the VHDL command `(others => ... )`

`'x` is equivalent to Verilog-2001 `'bx`

`'z` is equivalent to Verilog-2001 `'bz`

`'1` is equivalent to making an assignment of `-1` (2's complement of `-1` is all 1's)

`'0` is equivalent to making an assignment of `0` (for consistency)

Logic-Specific Processes

- SystemVerilog has three new logic specific processes to show designers intent:

```
always_comb
always_latch
always_ff
```

```
always_comb begin
  tmp1 = a & b;
  tmp2 = c & d;
  y = tmp1 ! tmp2;
end
```

```
always_latch
  if (en) q <= d;
```

```
always_ff @(posedge clk, negedge rst_n)
  if (!rst_n) q <= 0;
  else        q <= d;
```

Allows simulation tool to perform some linting functionality

always_comb
Logic-Specific Process

- **`always_comb`**
  - permits simulation tool to check for correct combinational coding style

Correct

```
module ao1 (
  output bit_t y,
  input  bit_t a, b, c, d);
  bit_t        tmp1, tmp2;

  always_comb begin
    tmp1 = a & b;
    tmp2 = c & d;
    y = tmp1 | tmp2;
  end
endmodule
```

Possible error message:

ERROR: combinational logic requested but latch was inferred

```
module ao1b (
  output bit_t q,
  input  bit_t en, d);

  always_comb
    if (en) q <= d;
endmodule
```

always_latch

Logic-Specific Process

- **`always_latch`**
  - permits simulation tool to check for correct latch coding style

Correct

```
module lat1 (
  output bit_t q,
  input  bit_t en, d);

  always_latch
    if (en) q <= d;
endmodule
```

Possible error message:

ERROR: combinational feedback loop - latch not inferred

```
module lat1b (
  output bit_t q,
  input  bit_t en, d);

  always_latch
    if (en) q <= d;
    else    q <= q;
endmodule
```

always_ff
Logic-Specific Process

- **`always_ff`**
  - permits simulation tool to check for correct registered logic coding style

Correct

```
module dff1 (
  output bit_t q,
  input  bit_t d, clk, rst_n);

  always_ff @(posedge clk, negedge rst_n)
    if (!rst_n) q <= 0;
    else        q <= d;
endmodule
```

Possible error message:

ERROR: incorrect sensitivity list - flip-flop not inferred

```
module dff1b (
  output bit_t q,
  input  bit_t d, clk, rst_n);

  always_ff @(clk, rst_n)
    if (!rst_n) q <= 0;
    else        q <= d;
endmodule
```

always @* -vs- always_comb

Exact differences are still being debated by IEEE VSG and Accellera SystemVerilog Committees

```
module fsm_sv1b_3
  ...

  always @* begin
    next = 'x;
    case (state)
    ...
  end

  ...
endmodule
```

```
module fsm_sv1b_3
  ...

  always_comb begin
    next = 'x;
    case (state)
    ...
  end

  ...
endmodule
```

Some differences exist

- `always_comb` **is sensitive to changes within the contents of a** `function`
- `always_comb` **may allow checking for illegal latches**
- `always_comb` **triggers once automatically at the end of time 0**
- `@*` **permitted within an** `always` **block**

Void Functions

- Function that does not return a value
- Does not have to be called from a Verilog expression
  - A void function can be a stand-alone call, like a Verilog task
- Unlike Verilog tasks, void functions
  - *cannot* wait
  - *cannot* include delays
  - *cannot* include event triggers
  - is searched by **`always_comb`** for signals to add to the sensitivity list

always_comb & void Functions

```
module comb1 (
  output bit_t [2:1] y,
  input  bit_t       a, b, c);

  always_comb
    orf1(a);

  function void orf1;
    input a;
    y[1] = a | b | c;
  endfunction

  always_comb
    ort1(a);

  task ort1;
    input a;
    y[2] = a | b | c;
  endtask
endmodule
```

Equivalent to:
`always @(a,b,c)`

-OR-

```
always @*
  orf1(a);
```

Equivalent to:
`always @(a)`

A void function behaves like 0-delay `task`

b & c are hidden inputs

Equivalent to:
`always @(a)`

-OR-

```
always @*
  ort1(a);
```

Equivalent to:
`always @(a)`

Verilog `task`

b & c are hidden inputs

Only `always_comb` is sensitive to changes within the contents of a `function`

always_ff
For Dual Data Rate (DDR) Sequential Logic ??

- Possible future enhancement to synthesis tools ??

Currently illegal syntax for synthesis

No posedge (clk)
No negedge (clk)

```
module ddrff (
  output bit_t q,
  input  bit_t d, clk, rst_n);

  always_ff @(clk, negedge rst_n)
    if (!rst_n) q <= 0;
    else        q <= d;
endmodule
```

Remove posedge to permit triggering on both edges ??

always_ff shows designer's intent

Could this synthesize to a DDR flip-flop in an ASIC vendor library ??

Design Intent - Unique/Priority

- Priority or No Priority? Back and Forth
- Synthesis pragmas are frequently abused!

```
full_case parallel_case
```

The "Evil Twins"

These directives tell the synthesis tool something different about the design than is told to the simulator

Potential pre- & post-synthesis simulation differences

- **`unique`** & **`priority`** *modifiers* give the same information to both the simulator and synthesis tool

```
unique case
priority case
unique if
priority if
```

Enables consistent behavior between:
- **simulator**
- **synthesis tool**
- **formal tools**

Design Intent - Priority

All possibilities have been defined - any other possibility would be an error

Simulation *AND* synthesis full_case!

- priority case:
  - full_case

```
priority case (1'b1)
 irq0: irq = 4'b1 << 0;
 irq1: irq = 4'b1 << 1;
 irq2: irq = 4'b1 << 2;
 irq3: irq = 4'b1 << 3;
endcase
```

- priority if
  - All branches specified without requiring ending else

```
priority if (irq0) irq = 4'b1;
else     if (irq1) irq = 4'b2;
else     if (irq2) irq = 4'b4;
else     if (irq3) irq = 4'b8;
```

At least one `irq0-irq3` must be high ... else simulation run-time error

A `default` or `else` statement nullifies the `priority` keyword

```
priority case (1'b1)
  irq0: irq = 4'b1 << 0;
  irq1: irq = 4'b1 << 1;
  irq2: irq = 4'b1 << 2;
  irq3: irq = 4'b1 << 3;
  default: irq = 0;
endcase
```

```
priority if (irq0) irq = 4'b1;
else     if (irq1) irq = 4'b2;
else     if (irq2) irq = 4'b4;
else     if (irq3) irq = 4'b8;
else               irq = 4'b0;
```

Design Intent - Unique

Simulation *AND* synthesis full_case & parallel_case!

- unique case:
  - full_case / parallel_case

```
unique case (1'b1)
  sel[0] : muxo = a;
  sel[1] : muxo = b;
  sel[2] : muxo = c;
endcase
```

- unique if-else:
  - full_case / parallel_case

```
unique if (sel[0]) muxo = a;
else   if (sel[1]) muxo = b;
else   if (sel[2]) muxo = c;
```

If `sel == 3'b011` ... simulation run-time error

***ANY unexpected* or *overlapping* `sel` value will cause a simulation run-time error**

```
unique case (sel)
  sel[0] : muxo = a;
  sel[1] : muxo = b;
  sel[2] : muxo = c;
  default: muxo = 'x;
endcase
```

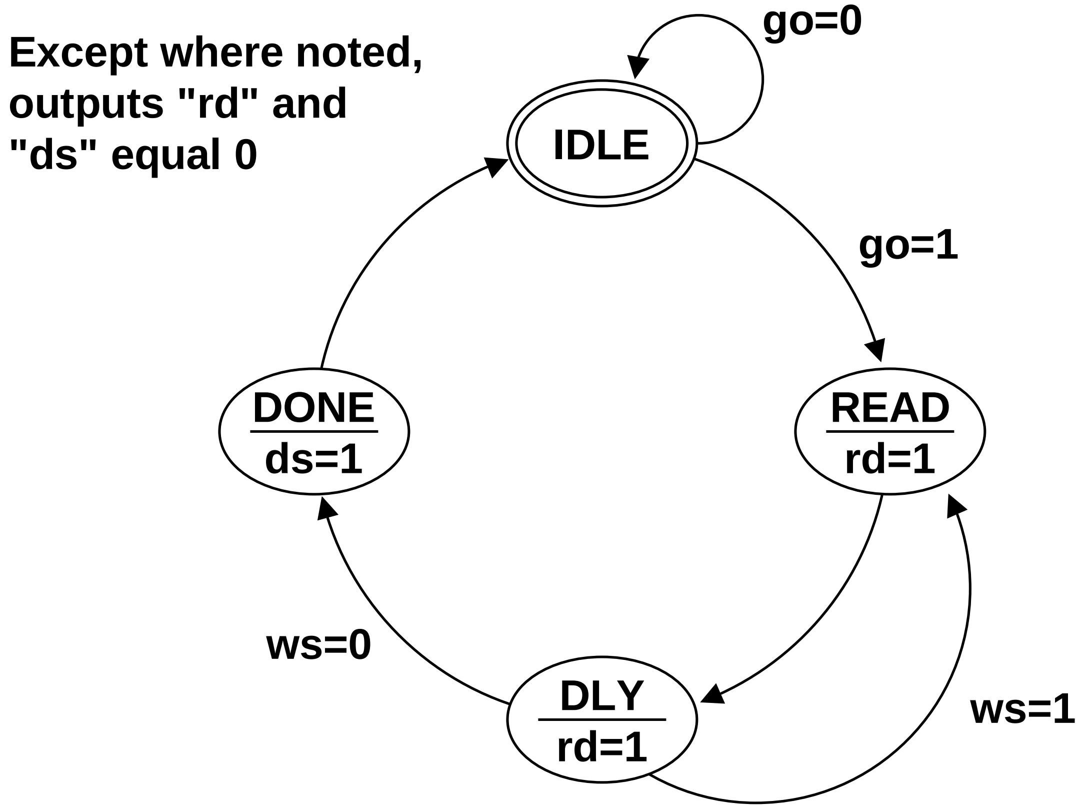
```
unique if (sel[0]) muxo = a;
else   if (sel[1]) muxo = b;
else   if (sel[2]) muxo = c;
else               muxo = 'x;
```

No run-time errors for unspecified combinations

unique still tests for *overlapping* conditions

Moore FSM Example - State Diagram

- FSM state diagram



Two-Always Block Coding Style

(Symbolic Parameter Assignments - Sequential Always Block)

```
module sm2a (rd, ds, go, ws, clk, rstN);
  output rd, ds;
  input  go, ws, clk, rstN;

  parameter IDLE = 2'b00,
            READ = 2'b01,
            DLY  = 2'b10,
            DONE = 2'b11;

  reg [1:0] state, next;

  always @(posedge clk or negedge rstN)
    if (!rstN) state <= IDLE;
    else       state <= next;
...
```

Verilog had no enumerated types

parameters are used in FSM design

Two-Always Block Coding Style
(Combinational Always Block - Continuous Assignment Outputs)

```
...
  always @(state or go or ws) begin
    next = 2'bx;
    case (state)
      IDLE : if (go)  next = READ;
             else     next = IDLE;
      READ :          next = DLY;
      DLY  :
             if (!ws) next = DONE;
             else     next = READ;
      DONE :          next = IDLE;
    endcase
  end

  assign rd = ((state==READ)||(state==DLY));
  assign ds =  (state==DONE);
endmodule
```

Simulation debug trick

Synthesis optimization trick

Output method #1 (continuous assignments)

Two-Always Block Coding Style

(Combinational Always Block - Always Block Outputs)

```
...
  always @(state or go or ws) begin
    next = 2'bx;
    rd = 1'b0;
    ds = 1'b0;
    case (state)
      IDLE : if (go)    next = READ;
             else       next = IDLE;
      READ : begin      rd   = 1'b1;
                        next = DLY;
             end
      DLY  : begin      rd   = 1'b1;
               if (!ws) next = DONE;
               else     next = READ;
             end
      DONE : begin      ds   = 1'b1;
                        next = IDLE;
             end
    endcase
  end
endmodule
```

Initial default value assignments initialize the outputs to a default state

Output method #2 (always-block assignments)

Enumerated Data Types

Strong typing at higher levels of abstraction

Anonymous 2-state int types

```
enum {red, yellow, green} light1, light2;
```

```
enum {bronze=3, silver, gold} medal;
```

silver=4, gold=5

```
enum {a=0, b=7, c, d=8} alphabet;
```

Syntax error (*implicit*) c=8 (explicit) d=8

```
enum {bronze=4'h3, silver, gold} medal;
```

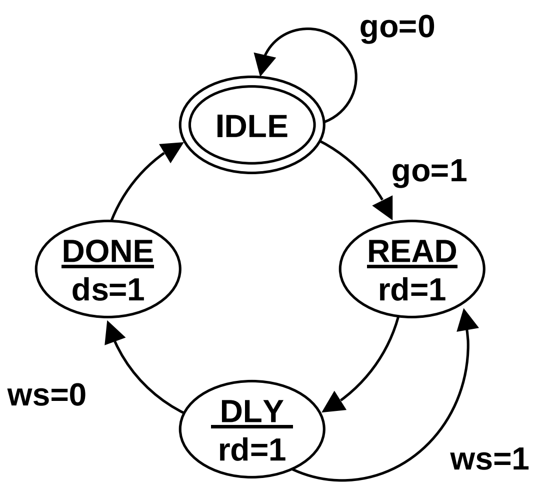
silver = 4'h4, gold=4'h5

```
typedef enum {red, yellow, green, blue, white, black} colors_t;

colors_t  light1, light2;

initial begin
  light1 = red;
  if (light1 == red) light1 = green;
end
```

traffic_light = 0 ("red")

traffic_light = 2 ("green")

Enumerated Types Abstract



Except where noted, outputs "rd" and "ds" equal 0

```
module fsm_sv1a_3
  ...

  enum {IDLE,
        READ,
        DLY,
        DONE,
        XX  } state, next;

  ...
endmodule
```

enum keyword

Comma-separated enumerated names enclosed within { }

Abstract enumerated names with no assigned values (values can be assigned later)

Enumerated variables `state` & `next`

Enum - Assigned Integer Values

```
module fsm_sv1b_3
  ...

  enum {IDLE = 3'b000,
        READ = 3'b001,
        DLY  = 3'b010,
        DONE = 3'b011,
        XX   = 3'b111} state, next;

  ...
endmodule
```

User-assigned values to enumerated names

Default type

If no data type is specified, `int` type is assumed

SystemVerilog

fsm1 Example - 3 Alw

Abstract Enums

Sunburst Design

```
module fsm_sv1a_3
  (output reg rd, ds,
   input      go, ws, clk, rst_n);

  enum {IDLE,
        READ,
        DLY,
        DONE,
        XX  } state, next;

  always @(posedge clk, negedge rst_n)
    if (!rst_n) state <= IDLE;
    else        state <= next;

  always @* begin
    next = XX;
    case (state)
      IDLE : if (go)   next = READ;
             else      next = IDLE;
      READ :           next = DLY;
      DLY  : if (!ws)  next = DONE;
             else      next = READ;
      DONE :           next = IDLE;
    endcase
  end
...
```

Abstract enumerated names

```
...
  always @(posedge clk, negedge rst_n)
    if (!rst_n) begin
      rd <= 1'b0;
      ds <= 1'b0;
    end
    else begin
      rd <= 1'b0;
      ds <= 1'b0;
      case (next)
        READ : rd <= 1'b1;
        DLY  : rd <= 1'b1;
        DONE : ds <= 1'b1;
      endcase
    end
endmodule
```

comma-separated sensitivity list

@* combinational sensitivity list (abbreviated syntax & reduces RTL errors)

Enumerated testing and assignments

Enum - Assigned 4-State Values

```
module fsm_sv1b_3
  ...

  enum reg [1:0] {IDLE = 2'b00,
                  READ = 2'b01,
                  DLY  = 2'b10,
                  DONE = 2'b11,
                  XX   = 'x   } state, next;

  ...
endmodule
```

4-state data types can be specified to permit X and Z assignment

X-assignment is very useful for simulation debugging and synthesis "don't-care" optimization

fsm1 Example - 3 Always Blocks

Assigned Enums

```
module fsm_sv1b_3
  (output reg rd, ds,
   input      go, ws, clk, rst_n);

  enum reg [1:0] {IDLE = 2'b00,
                  READ = 2'b01,
                  DLY  = 2'b10,
                  DONE = 2'b11,
                  XX   = 'x   } state, next;

  always @(posedge clk, negedge rst_n)
    if (!rst_n) state <= IDLE;
    else        state <= next;

  always @* begin
    next = XX;
    case (state)
      IDLE : if (go)  next = READ;
             else     next = IDLE;
      READ :          next = DLY;
      DLY  : if (!ws) next = DONE;
             else     next = READ;
      DONE :          next = IDLE;
    endcase
  end
...
```

```
...
  always @(posedge clk, negedge rst_n)
    if (!rst_n) begin
      rd <= 1'b0;
      ds <= 1'b0;
    end
    else begin
      rd <= 1'b0;
      ds <= 1'b0;
      case (next)
        READ : rd <= 1'b1;
        DLY  : rd <= 1'b1;
        DONE : ds <= 1'b1;
      endcase
    end
endmodule
```

Assigned enumerated values

This is the only change required to go from *abstract* to *encoded*

fsm1 - 3 Always Blocks

SystemVerilog 3.0 - Assigned Enums

```
module fsm_sv1b_3
  (output reg rd, ds,
   input      go, ws, clk, rst_n);

  enum reg [1:0] {IDLE = 2'b00,
                  READ = 2'b01,
                  DLY  = 2'b10,
                  DONE = 2'b11,
                  XX   = 'x   } state, next;

  always_ff @(posedge clk, negedge rst_n)
    if (!rst_n) state <= IDLE;
    else        state <= next;

  always_comb begin
    next = XX;
    unique case (state)
      IDLE : if (go)  next = READ;
             else     next = IDLE;
      READ :          next = DLY;
      DLY  : if (!ws) next = DONE;
             else     next = READ;
      DONE :          next = IDLE;
    endcase
  end
...
```

```
...
  always @(posedge clk, negedge rst_n)
    if (!rst_n) begin
      rd <= 1'b0;
      ds <= 1'b0;
    end
    else begin
      rd <= 1'b0;
      ds <= 1'b0;
      case (next)
        READ : rd <= 1'b1;
        DLY  : rd <= 1'b1;
        DONE : ds <= 1'b1;
      endcase
    end
endmodule
```

Allows more lint-like checking of code

Equivalent to full_case parallel_case for both synthesis tool *AND SIMULATOR* and formal tools

Enumerated Types - Design Flow

Start with undefined abstract states

```
enum {IDLE,
      READ,
      DLY ,
      DONE,
      XX  } state, next;
```

Good for abstraction

Good for waveform viewing

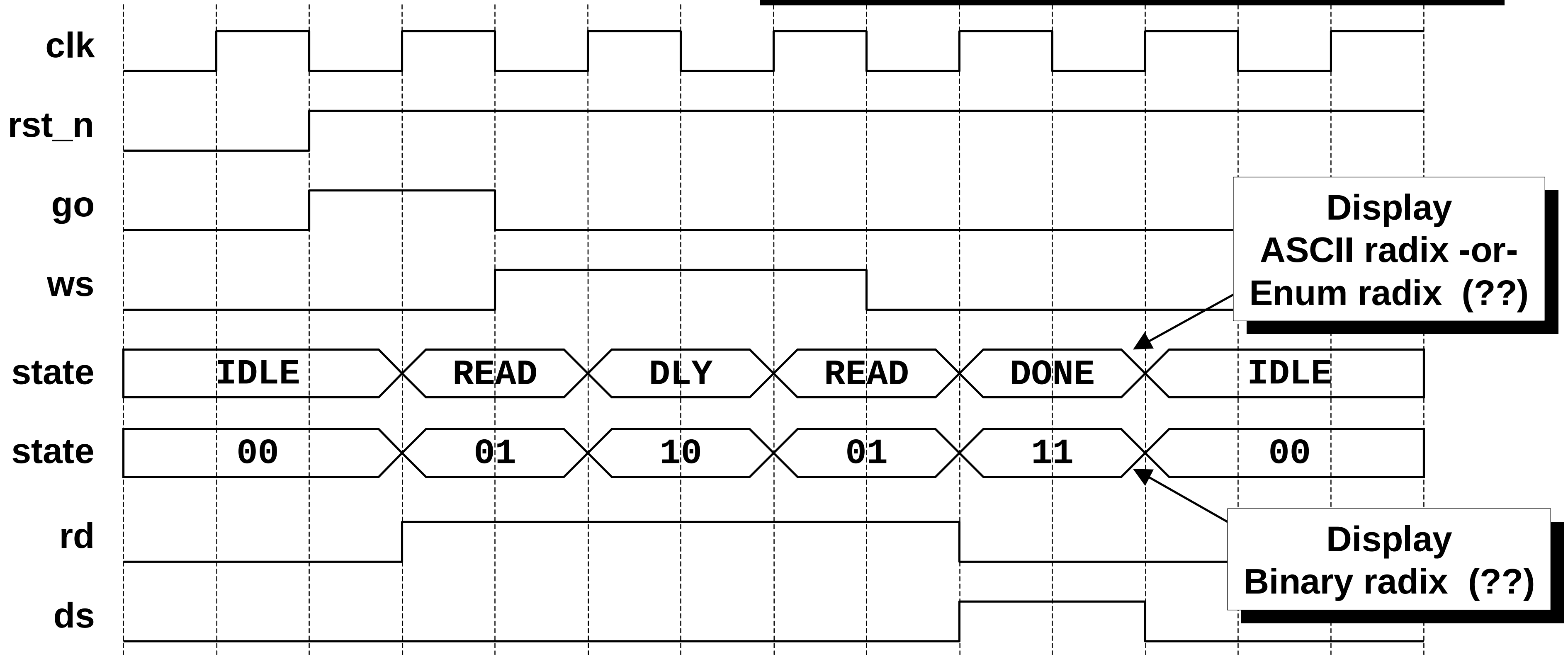
Add a 4-state variable enum type to permit valid state assignments and X-state assignment

```
enum reg [1:0] {IDLE = 2'b00,
                READ = 2'b01,
                DLY  = 2'b10,
                DONE = 2'b11,
                XX   = 'x   } state, next;
```

Good for debugging and synthesis

Enum Types - Waveform Display

Easier than coding Verilog ASCII monitors


clk
rst_n
go
ws
state
IDLE
READ
DLY
READ
DONE
IDLE
state
00
01
10
01
11
00
rd
ds
Display ASCII radix -or- Enum radix (??)
Display Binary radix (??)

```
enum reg [1:0] {IDLE = 2'b00,
                READ = 2'b01,
                DLY  = 2'b10,
                DONE = 2'b11,
                XX   = 'x   } state, next;
```

Exact standard waveform display capabilities are still being defined

SystemVerilog

Enhanced for-Loop

Sunburst Design

Verilog-2001

```
module for4a (
  output reg [31:0] y,
  input      [31:0] a,
  input             s);
  integer i;

  always @(a or s)
    for (i=0; i<32; i=i+1)
      if (!s) y[i] = a[i];
      else    y[i] = a[31-i];
endmodule
```

Separate iteration-variable declaration

Explicit increment

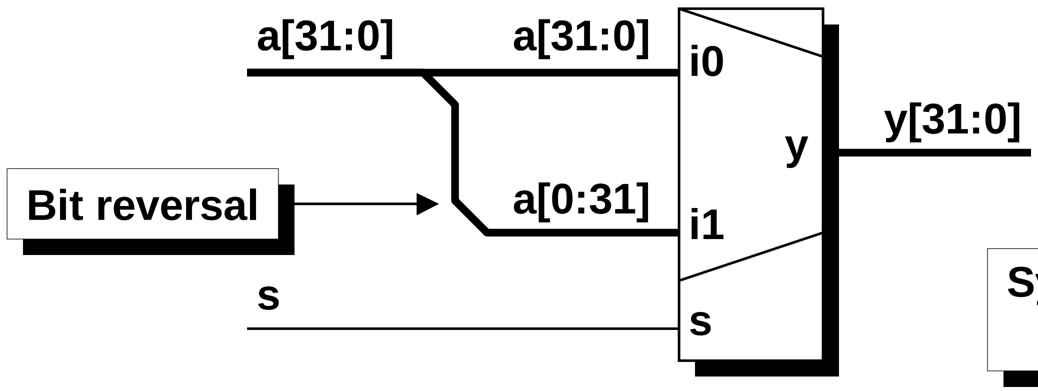
SystemVerilog

```
module for4b (
  output logic [31:0] y,
  input        [31:0] a,
  input               s);

  always @(a or s)
    for (int i=0; i<32; i++)
      if (!s) y[i] = a[i];
      else    y[i] = a[31-i];
endmodule
```

Local iteration-variable declaration

Auto-increment

Local iteration-variables are automatic variables that do not exist after exiting the loop



SystemVerilog also added a do-while loop (bottom testing loop)

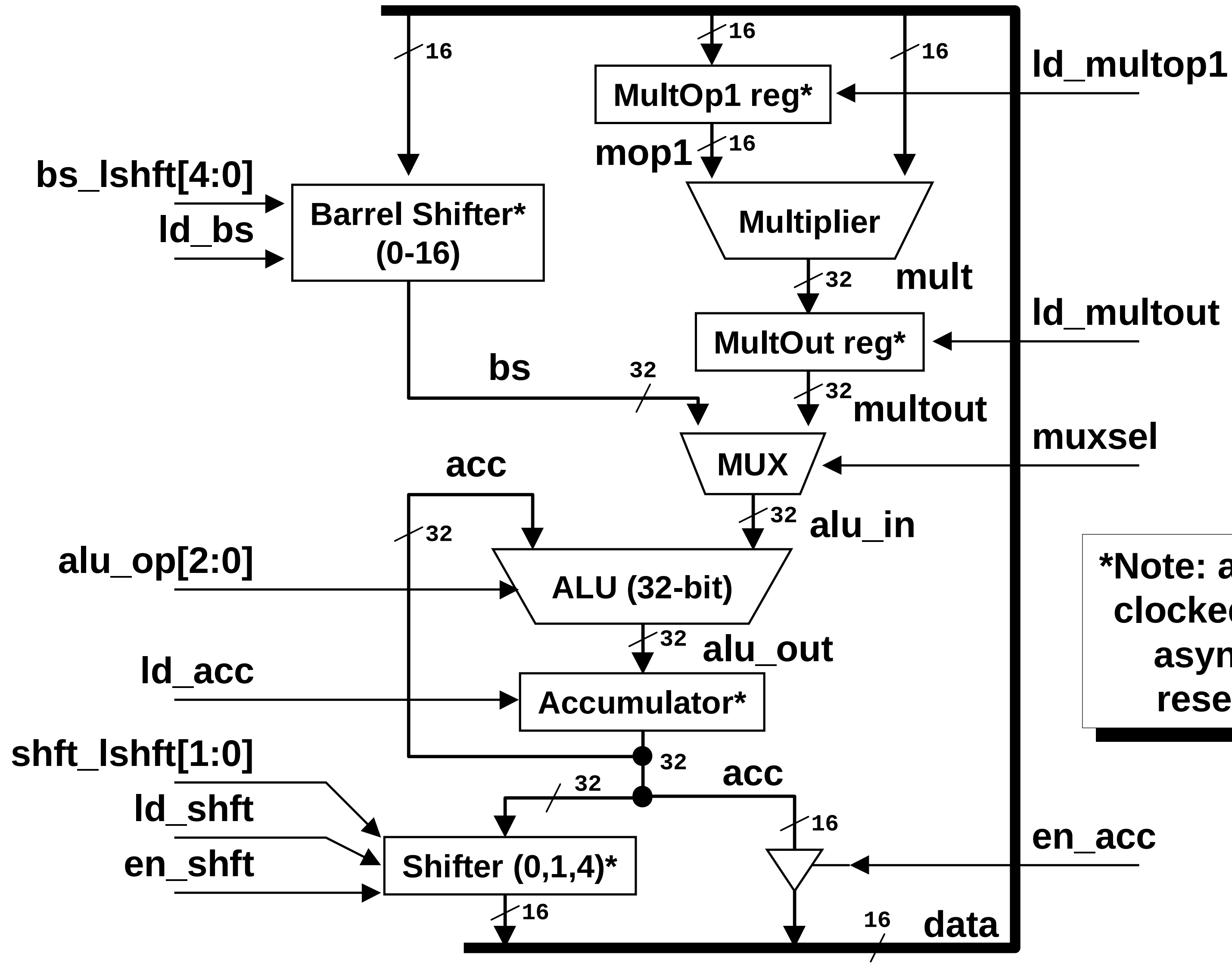
Implicit Port Connections

- Verilog and VHDL have both had the ability to instantiate modules using either positional or named port connections
- SystemVerilog solves top-level verbosity with two new implicit port connection enhancements:
  - **.name** port connections
  - **.*** implicit port connections

Port connections at the top-level of a large ASIC can be both tedious and verbose

Medium-size example follows

Central Arithmetic Logic Unit (CALU) Block Diagram



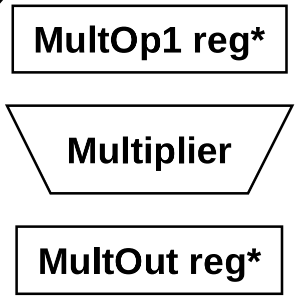
*Note: all registers are clocked by "clk" and asynchronously reset by "rst_n"

CALU Top-Level Module 1/2

SystemVerilog .name Implicit Ports Version

```
module calu3 (
  inout  [15:0] data,
  input  [ 4:0] bs_lshft,
  input  [ 2:0] alu_op,
  input  [ 1:0] shft_lshft,
  input         calu_muxsel, en_shft, ld_acc, ld_bs,
  input         ld_multop1, ld_multout, ld_shft, en_acc,
  input         clk, rst_n);

  wire   [31:0] acc, alu_in, alu_out, bs, mult, multout;
  wire   [15:0] mop1;

  multop1       multop1       (.mop1, .data, .ld_multop1,
                               .clk, .rst_n);
  multiplier    multiplier    (.mult, .mop1, .data);
  multoutreg    multoutreg    (.multout, .mult,
                               .ld_multout, .clk, .rst_n);
...
```

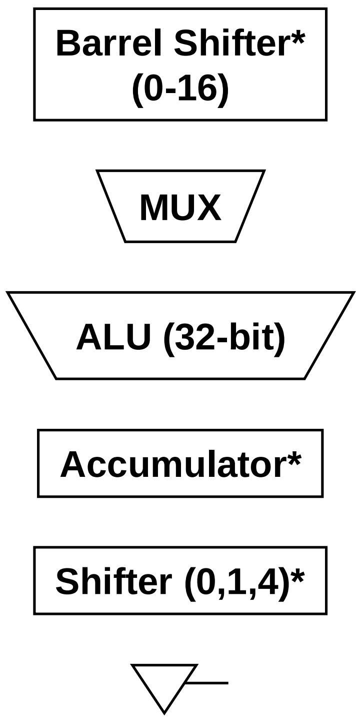
Matching port names are listed just once



```
multop1 multop1 (.mop1, .data, .ld_multop1,
                                  .clk, .rst_n);
```

CALU Top-Level Module 2/2

SystemVerilog .name Implicit Ports Version



```
...
  barrel_shifter barrel_shifter (.bs, .data, .bs_lshft,
                                 .ld_bs, .clk, .rst_n);
  mux2           mux            (.y(alu_in),
                                 .i0(multout), .i1(acc),
                                 .sel1(calu_muxsel));
  alu            alu            (.alu_out, .zero(), .neg(),
                                 .alu_in, .acc, .alu_op);
  accumulator    accumulator    (.acc, .alu_out, .ld_acc,
                                 .clk, .rst_n);
  shifter        shifter        (.data, .acc, .shft_lshft,
                                 .ld_shft, .en_shft,
                                 .clk, .rst_n);
  tribuf         tribuf         (.data, .acc(acc[15:0]),
                                 .en_acc);
endmodule
```

All of the advantages of named port connections

Less verbose!

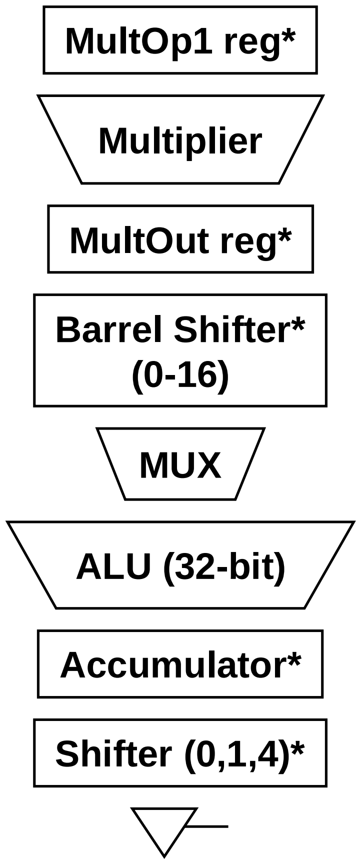
CALU Top-Level Module

SystemVerilog .* Implicit Ports Version

```
module calu4 (
  inout  [15:0] data,
  input  [ 4:0] bs_lshft,
  input  [ 2:0] alu_op,
  input  [ 1:0] shft_lshft,
  input         calu_muxsel, en_shft, ld_acc, ld_bs,
  input         ld_multop1, ld_multout, ld_shft, en_acc,
  input         clk, rst_n);

  wire   [31:0] acc, alu_in, alu_out, bs, mult, multout;
  wire   [15:0] mop1;

  multop1         multop1         (.*);
  multiplier      multiplier      (.*);
  multoutreg      multoutreg      (.*);
  barrel_shifter  barrel_shifter  (.*);
  mux2            mux             (.y(alu_in), .i0(multout),
                                   .i1(acc), .sel1(calu_muxsel));
  alu             alu             (.*, .zero(), .neg());
  accumulator     accumulator     (.*);
  shifter         shifter         (.*);
  tribuf          tribuf          (.*, .acc(acc[15:0]));
endmodule
```

This style emphasizes where port differences occur

Much less verbose!



Rules for Implicit `.name` and `.*` Port Connections

- Rule: mixing **`.*`** and **`.name`** ports in the same instantiation is prohibited

- Permitted:
  - **`.name`** and **`.name(signal)`** connections in the same instantiation

    -OR-
  - **`.*`** and **`.name(signal)`** connections in the same instantiation

- Rules: **`.name(signal)`** connections are required for:
  - size-mismatch
  - name-mismatch
  - unconnected ports

```
inst u1 (..., .data(data[7:0]), ...);
inst u2 (..., .data(pdata),     ...);
inst u3 (..., .berr( ),         ...);
```

NOTE: stronger typing of ports than Verilog-2001 *and* more concise!

SystemVerilog

CALU Top-Level Module

Coded Four Different Ways

Sunburst Design

Positional ports

```
module calu1 (
  inout  [15:0] data,
  input  [ 4:0] bs_lshft,
  input  [ 2:0] alu_op,
  input  [ 1:0] shft_lshft,
  input         calu_muxsel, en_shft, ld_acc, ld_bs,
  input         ld_multop1, ld_multout, ld_shft, en_acc,
  input         clk, rst_n);

  wire   [31:0] acc, alu_in, alu_out, bs, mult, multout;
  wire   [15:0] mop1;

  multop1        multop1        (mop1, data, ld_multop1,
                                 clk, rst_n);
  multiplier     multiplier     (mult, mop1, data);
  multoutreg     multoutreg     (multout, mult,
                                 ld_multout, clk, rst_n);
  barrel_shifter barrel_shifter (bs, data, bs_lshft,
                                 ld_bs, clk, rst_n);
  mux2           mux            (alu_in, multout, acc,
                                 calu_muxsel);
  alu            alu            (alu_out, , ,
                                 alu_in, acc, alu_op);
  accumulator    accumulator    (acc, alu_out, ld_acc,
                                 clk, rst_n);
  shifter        shifter        (data, acc, shft_lshft,
                                 ld_shft, en_shft,
                                 clk, rst_n);
  tribuf         tribuf         (data, acc[15:0],
                                 en_acc);
endmodule
```

31 lines of code
680 characters

Named port connections

```
module calu2 (
  inout  [15:0] data,
  input  [ 4:0] bs_lshft,
  input  [ 2:0] alu_op,
  input  [ 1:0] shft_lshft,
  input         calu_muxsel, en_shft, ld_acc, ld_bs,
  input         ld_multop1, ld_multout, ld_shft, en_acc,
  input         clk, rst_n);

  wire   [31:0] acc, alu_in, alu_out, bs, mult, multout;
  wire   [15:0] mop1;

  multop1        multop1        (.mop1(mop1), .data(data),
                                 .ld_multop1(ld_multop1),
                                 .clk(clk), .rst_n(rst_n));
  multiplier     multiplier     (.mult(mult), .mop1(mop1),
                                 .data(data));
  multoutreg     multoutreg     (.multout(multout),
                                 .mult(mult),
                                 .ld_multout(ld_multout),
                                 .clk(clk), .rst_n(rst_n));
  barrel_shifter barrel_shifter (.bs(bs), .data(data),
                                 .bs_lshft(bs_lshft),
                                 .ld_bs(ld_bs),
                                 .clk(clk), .rst_n(rst_n));
  mux2           mux            (.y(alu_in),
                                 .i0(multout),
                                 .i1(acc),
                                 .sel1(calu_muxsel));
  alu            alu            (.alu_out(alu_out),
                                 .zero(), .neg(), .alu_in(alu_in),
                                 .acc(acc), .alu_op(alu_op));
  accumulator    accumulator    (.acc(acc), .alu_out(alu_out),
                                 .ld_acc(ld_acc), .clk(clk),
                                 .rst_n(rst_n));
  shifter        shifter        (.data(data), .acc(acc),
                                 .shft_lshft(shft_lshft),
                                 .ld_shft(ld_shft),
                                 .en_shft(en_shft),
                                 .clk(clk), .rst_n(rst_n));
  tribuf         tribuf         (.data(data), .acc(acc[15:0]),
                                 .en_acc(en_acc));
endmodule
```

43 lines of code
1,020 characters

.name implicit ports

```
module calu3 (
  inout  [15:0] data,
  input  [ 4:0] bs_lshft,
  input  [ 2:0] alu_op,
  input  [ 1:0] shft_lshft,
  input         calu_muxsel, en_shft, ld_acc, ld_bs,
  input         ld_multop1, ld_multout, ld_shft, en_acc,
  input         clk, rst_n);

  wire   [31:0] acc, alu_in, alu_out, bs, mult, multout;
  wire   [15:0] mop1;

  multop1        multop1        (.mop1, .data, .ld_multop1,
                                 .clk, .rst_n);
  multiplier     multiplier     (.mult, .mop1, .data);
  multoutreg     multoutreg     (.multout, .mult,
                                 .ld_multout, .clk, .rst_n);
  barrel_shifter barrel_shifter (.bs, .data, .bs_lshft,
                                 .ld_bs, .clk, .rst_n);
  mux2           mux            (.y(alu_in),
                                 .i0(multout), .i1(acc),
                                 .sel1(calu_muxsel));
  alu            alu            (.alu_out, .zero(), .neg(),
                                 .alu_in, .acc, .alu_op);
  accumulator    accumulator    (.acc, .alu_out, .ld_acc,
                                 .clk, .rst_n);
  shifter        shifter        (.data, .acc, .shft_lshft,
                                 .ld_shft, .en_shft,
                                 .clk, .rst_n);
  tribuf         tribuf         (.data, .acc(acc[15:0]),
                                 .en_acc);
endmodule
```

32 lines of code
757 characters

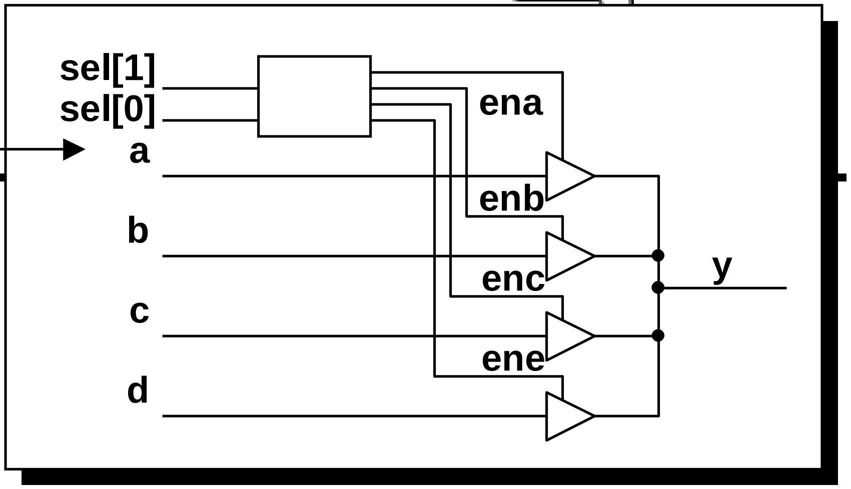
.* implicit ports

```
module calu2 (
  inout  [15:0] data,
  input  [ 4:0] bs_lshft,
  input  [ 2:0] alu_op,
  input  [ 1:0] shft_lshft,
  input         calu_muxsel, en_shft, ld_acc, ld_bs,
  input         ld_multop1, ld_multout, ld_shft, en_acc,
  input         clk, rst_n);

  wire   [31:0] acc, alu_in, alu_out, bs, mult, multout;
  wire   [15:0] mop1;

  multop1        multop1        (.*);
  multiplier     multiplier     (.*);
  multoutreg     multoutreg     (.*);
  barrel_shifter barrel_shifter (.*);
  mux2           mux            (.y(alu_in), .i0(multout),
                                 .i1(acc), .sel1(calu_muxsel));
  alu            alu            (.*, .zero(), .neg());
  accumulator    accumulator    (.*);
  shifter        shifter        (.*);
  tribuf         tribuf         (.*, .acc(acc[15:0]));
endmodule
```

23 lines of code
518 characters

Disadvantage of Implicit .* Port Connections

Intended logic



- Easy to connect the wrong ports to the wrong nets for common identifier names
  - Could be difficult to debug

Common enable name

```
module drivera (
  output [7:0] y,
  input  [7:0] a,
  input        ena);

  assign y = ena ? a : 8'bz;
endmodule
```

```
module driverb (
  output [7:0] y,
  input  [7:0] b,
  input        ena);

  assign y = ena ? b : 8'bz;
endmodule
```

```
module driverc (
  output [7:0] y,
  input  [7:0] c,
  input        ena);

  assign y = ena ? c : 8'bz;
endmodule
```

```
module driverd (
  output [7:0] y,
  input  [7:0] d,
  input        ena);

  assign y = ena ? d : 8'bz;
endmodule
```

Alias-ed Buses He[lp] Port Connections

New SystemVerilog keyword `alias`

```
module onehot_busmux (
  output [7:0] y,
  input  [7:0] a, b, c, d,
  input  [1:0] sel);
  wire ena, enb, enc, ene;
  wire [7:0] y1, y2, y3, y4;
  alias y = y1 = y2 = y3 = y4;
  ena_decode u0 (.*);

  drivera    u1 (.*);
  driverb    u2 (.*, .ena(enb));
  driverc    u3 (.*, .ena(enc));
  driverd    u4 (.*, .ena(end));
endmodule
```

y, y1, y2, y3 & y4 are all aliased (connected) together

Shorter .* instantiations

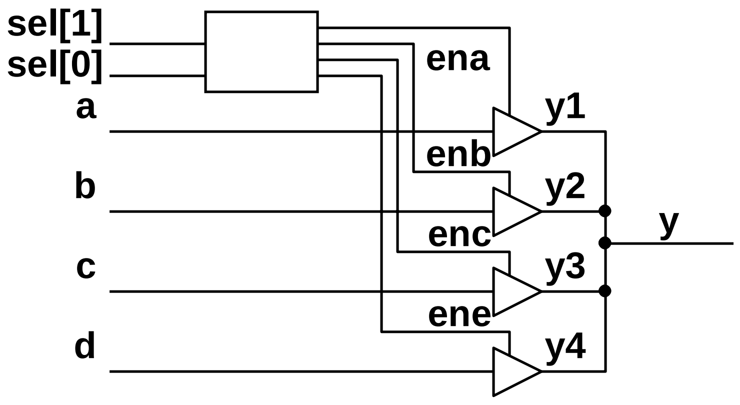
```
module ena_decode (
  output reg ena, enb, enc, ene,
  input      [1:0] sel); ...
```

```
module drivera (
  output [7:0] y1,
  input  [7:0] a,
  input        ena); ...
```

```
module driverb (
  output [7:0] y2,
  input  [7:0] b,
  input        ena); ...
```

```
module driverc (
  output [7:0] y3,
  input  [7:0] c,
  input        ena); ...
```

```
module driverd (
  output [7:0] y4,
  input  [7:0] d,
  input        ena); ...
```



Implicit .* Port Connection Advantages & Disadvantages

Advantages

- Much easier to assemble top-level ASIC/FPGA designs
- Easier to scan large top-level designs with 1,000's of ports

Only exceptions are listed and stand out

- Not as verbose as either **`.name`** or explicit port connections
- Still has all of the advantages of explicit port connections
- Easy to assemble a block-level testbench
  - Make the testbench signals match the port names

Disadvantages

- Can cause unwanted connections
  - Could be difficult to debug
  - Complicates separate compilation
  - Users may call AEs when they use it wrong ("*your tool is broken*!")

Legitimate tool developer fear!!

- Not useful for low-level, structural netlists
  - Primitive port names rarely match connecting nets

All ports are implicitly connected

The `.*` notation removes all of the unnecessary verbosity

SystemVerilog Enhancements for Design & Verification

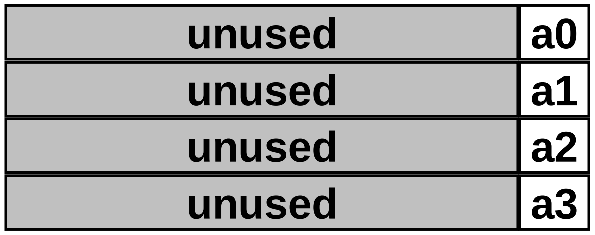
Benefits of Single Language for Design AND Testbench

- Easy learning curve
  - Facilitates quicker adoption
- Improved communication between design and verification teams
- Reduces design and verification complexity with advanced constructs

Packed & Unpacked Arrays

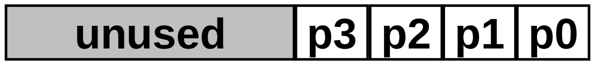
Unpacked array of bits

```
bit a [3:0];
```



Packed array of bits

```
bit [3:0] p;
```



1K 16 bit unpacked memory

```
bit [15:0] memory [1023:0];
initial begin
  memory[i]       = ~memory[i];
  memory[i][15:8] = 0;
end
```

Packed indexes can be sliced

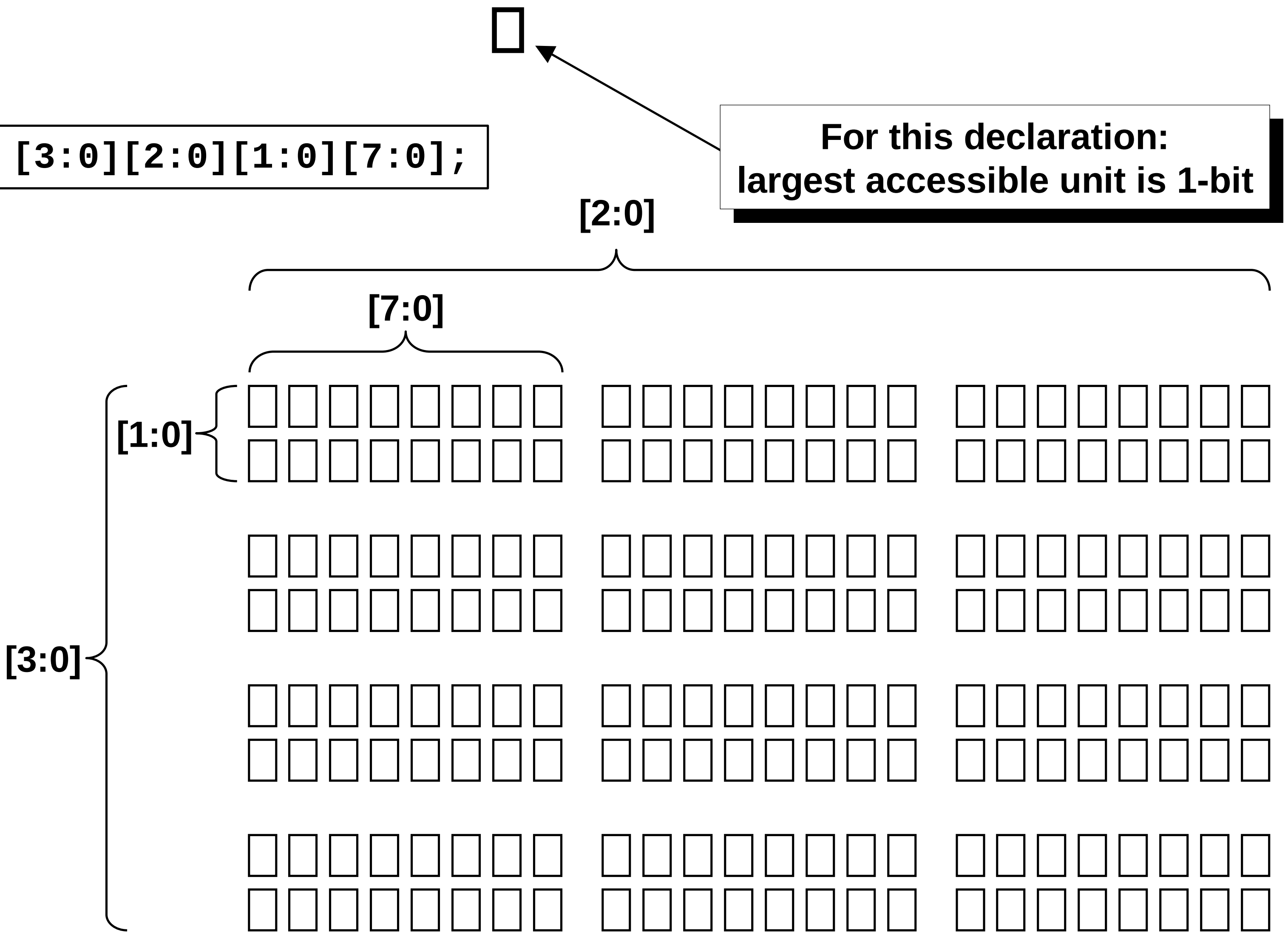
1K 16 bit packed memory

```
bit [1023:0][15:0] vframe;
always @(vcmd)
  if (vcmd == INV) vframe = ~vframe;
```

Can operate on an entire memory!

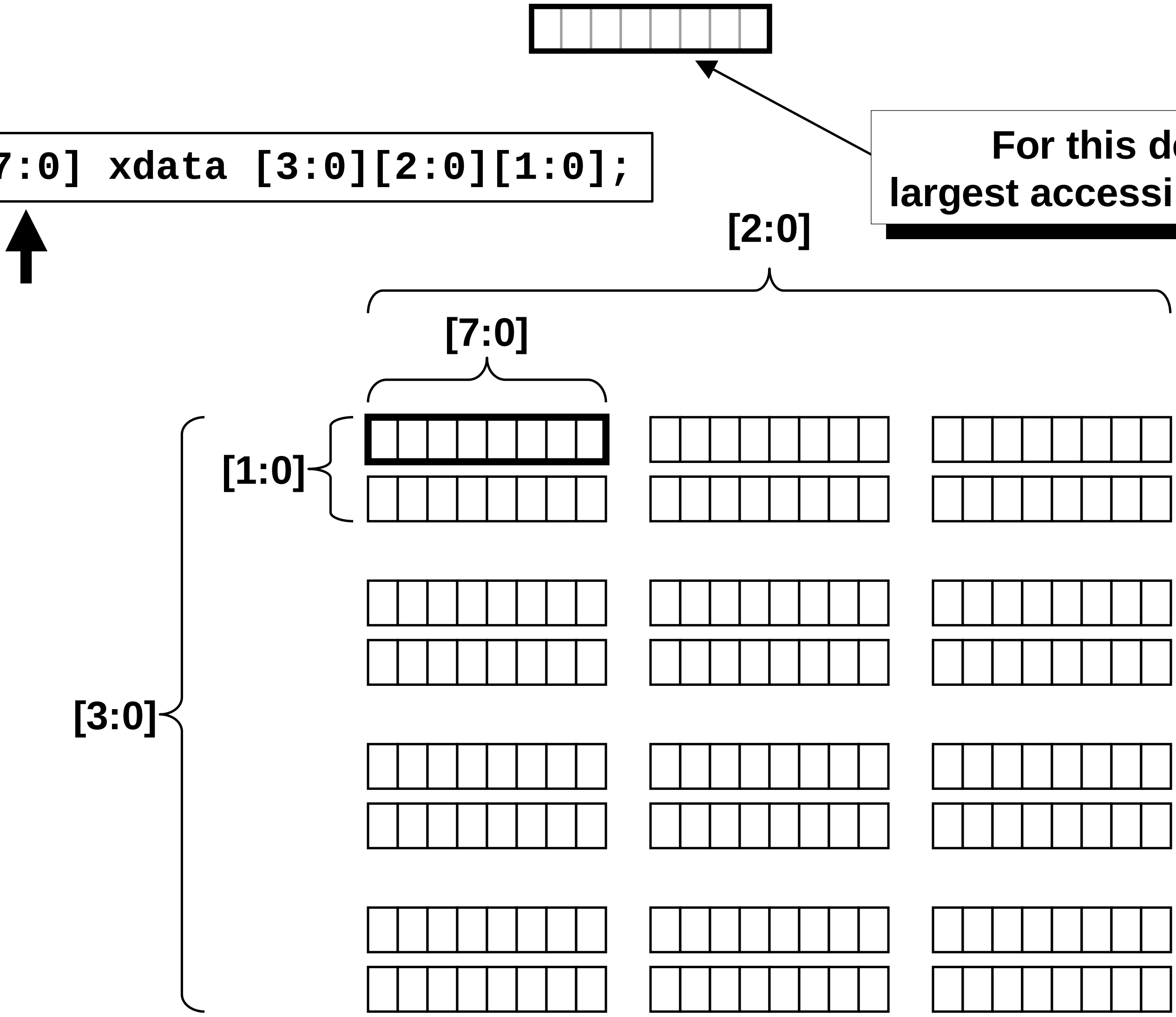
Array - 4-D Unpacked

```
logic xdata [3:0][2:0][1:0][7:0];
```



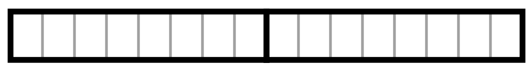
Array - 1-D Packed & 3-D Unpacked

```
logic [7:0] xdata [3:0][2:0][1:0];
```



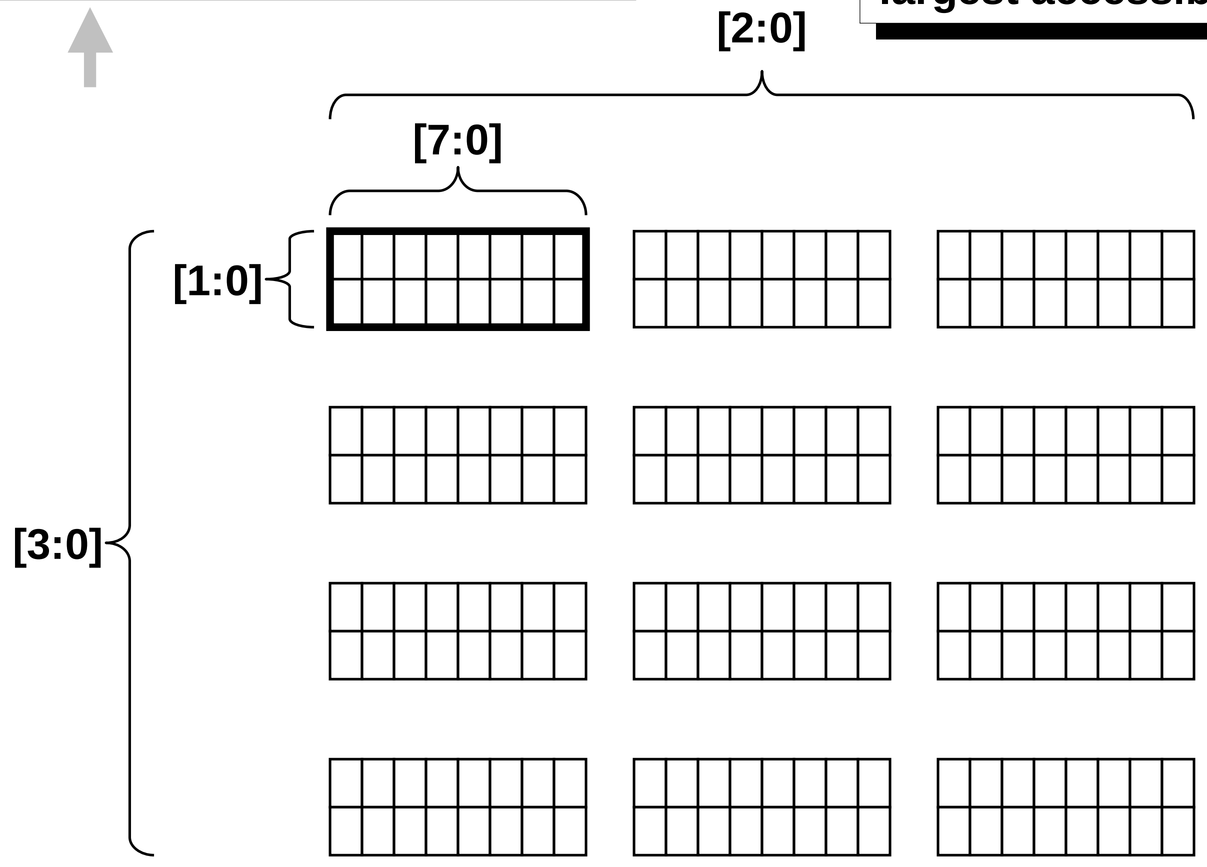
For this declaration: largest accessible unit is 8-bits

Array - 2-D Packed & 2-D Unpacked

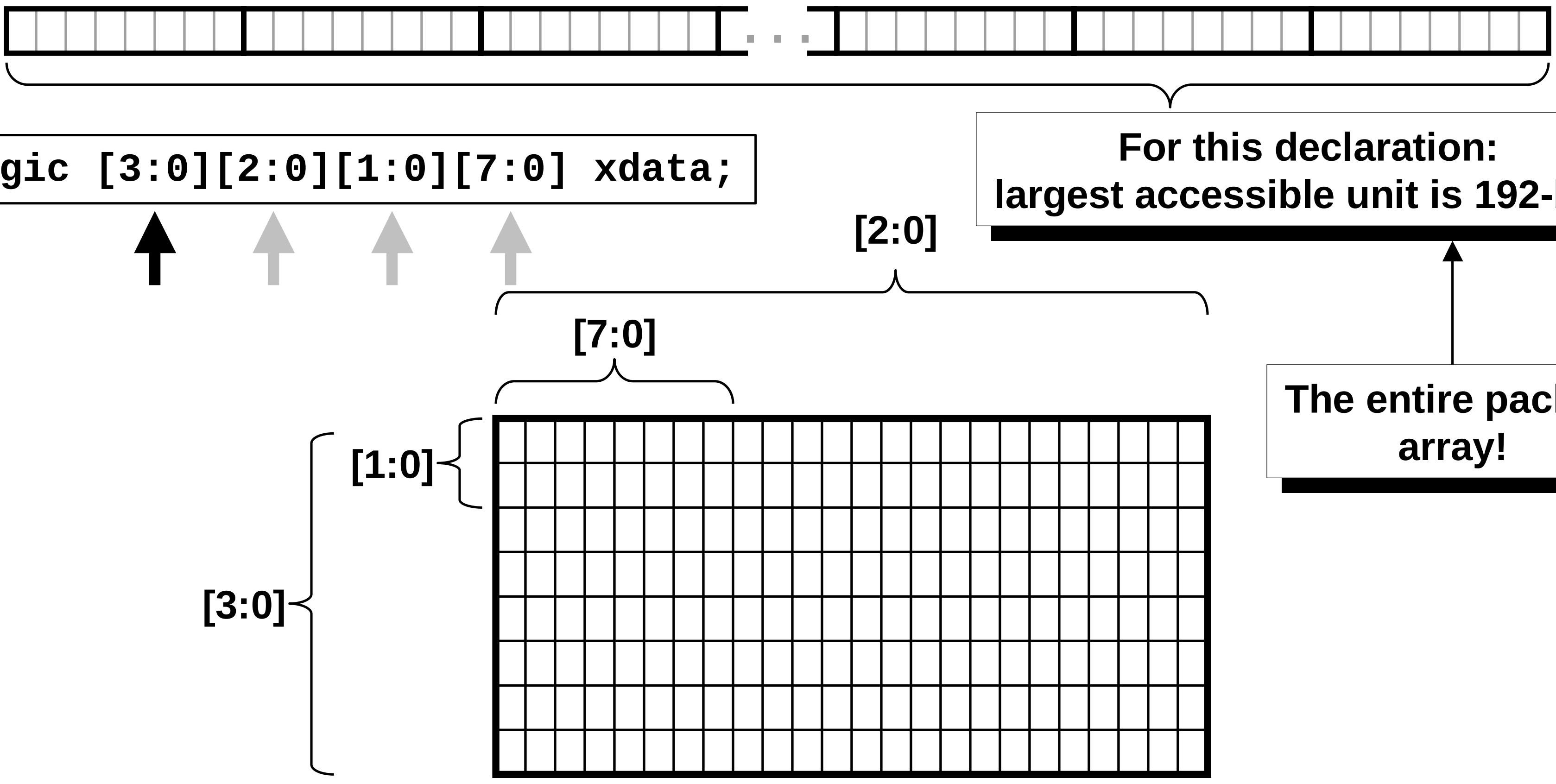


```
logic [1:0][7:0] xdata [3:0][2:0];
```

For this declaration:
largest accessible unit is 16-bits



Array - 4-D Packed



```
logic [3:0][2:0][1:0][7:0] xdata;
```

For this declaration:
largest accessible unit is 192-bits

The entire packed array!

Data Organization

- Signals are meaningful in groups
  - Instructions: operation & operands
  - Packet fields: address, data, CRC
- Verilog provides only informal grouping

```
reg [47:0] pktsrc_adr;
reg [47:0] pktdst_adr;
reg [7:0]  InstOpCde;
reg [7:0]  InstOpRF [127:0];
```

By name

```
`define opcode 31:16
reg [31:0] Instruction;
Instruction[`opcode]
```

By vector location

Data Organization

- Goal: organize data the same as in high-level programming
  - Allows others to see explicit, meaningful relationships in the design

- SystemVerilog adds **`structs`**, **`unions`** & arrays
  - can be used separately or combined to accurately capture design intent

Data Organization - Structs

- **`structs`** preserve logical grouping

```
struct {
  addr_t src_adr;
  addr_t dst_adr;
  data_t data;
} pkt;

initial begin
  pkt.src_adr = src_adr;
  if (pkt.dst_adr == node.adr);
  ...
end
```

- References to **`struct`** members are longer expressions but facilitate more meaningful code

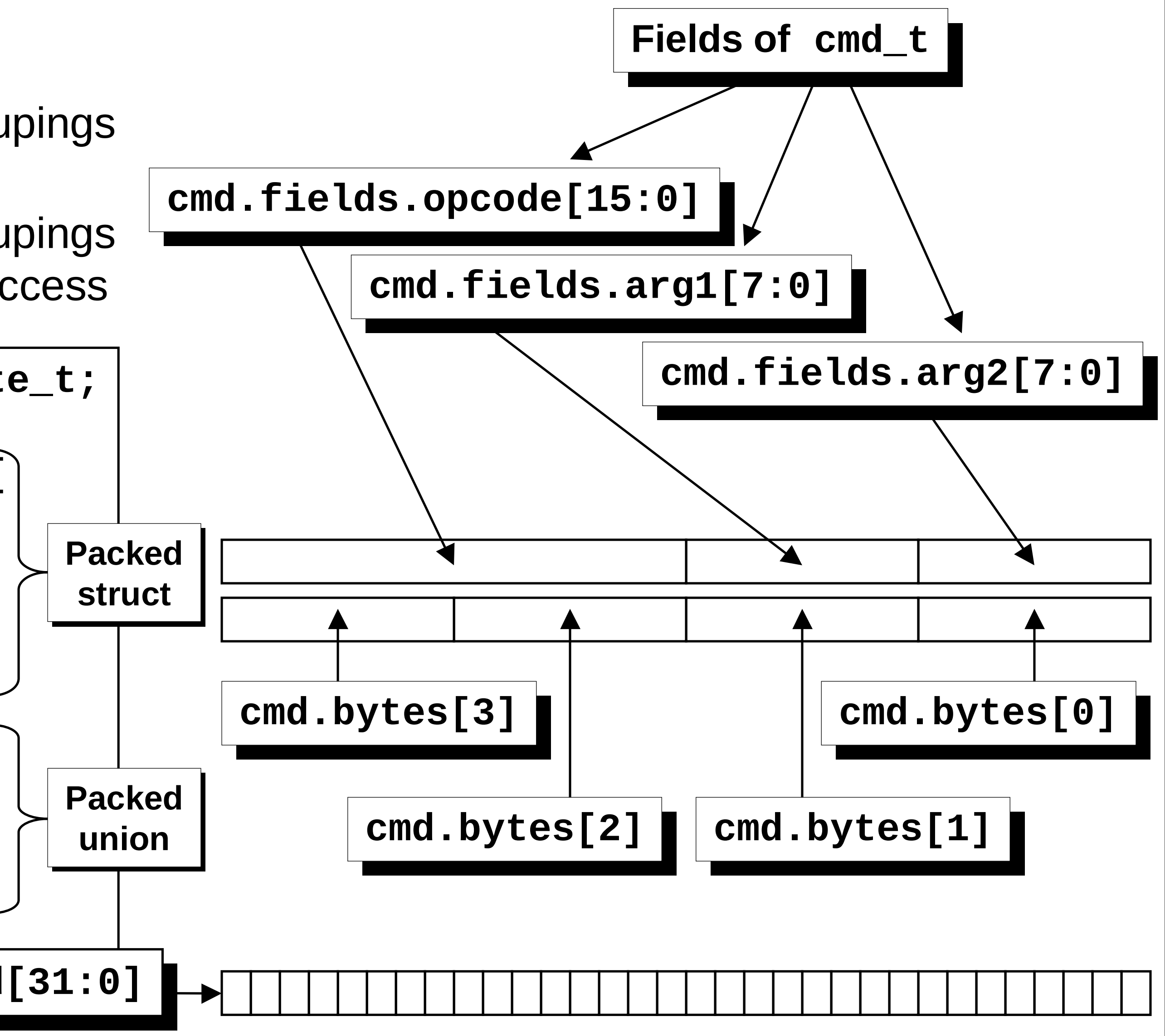
Assign the `src_adr` to the `src_adr` field of the `pkt` structure

Compare (test) the `dst_adr` field of the `pkt` structure to the `adr` field of the `node` structure

Data Organization - Packed Structs & Packed Unions

- *Unpacked* **struct**
  - preserves logical groupings
- *Packed* **struct**
  - preserves logical groupings
  - enables convenient access

```
typedef logic [7:0] byte_t;

typedef struct packed {
  logic [15:0] opcode;
  logic  [7:0] arg1;
  logic  [7:0] arg2;
} cmd_t;

typedef union packed {
  byte_t [3:0] bytes;
  cmd_t        fields;
} instruction_u;

instruction_u cmd;
```

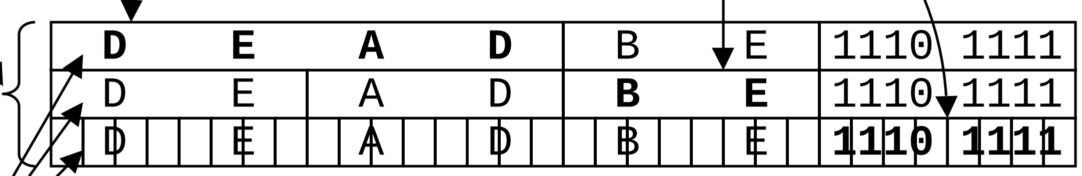


Data Organization - Packed Structs & Packed Unions

```
initial begin
  cmd.fields.opcode = 16'hDEAD;
  cmd[7:0] = 8'hEF;
  cmd.bytes[1] = 8'hBE;
end
```

This is all the same packed data structure

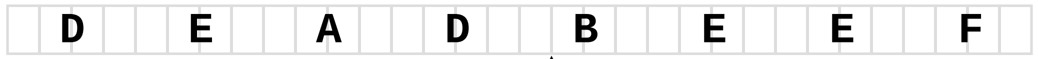
```
typedef logic [7:0] byte_t;

typedef struct packed {
  logic [15:0] opcode;
  logic  [7:0] arg1;
  logic  [7:0] arg2;
} cmd_t;

typedef union packed {
  byte_t [3:0] bytes;
  cmd_t        fields;
} instruction_u;

instruction_u cmd;
```



Data maps to all members

Packed unions enable many convenient names to reference the same data

No need to test data types



cmd now equals 32'hDEADBEEF

Break!

(10:15 - 10:30 AM)

SystemVerilog Symposium will resume at 10:30 AM

Next Topic: **SystemVerilog Interfaces**

(+ more Design & Testbench enhancements)

Port Instantiations & Port Connections

- SystemVerilog Enhancements
  - Inferred port connections using **`.name`** & **`.*`**
  - SystemVerilog connections by interface - level I
  - SystemVerilog connections by interface - level II

    Includes: tasks, functions, assertions, etc.

    Great for testbench development - the interface includes the legal interface commands

    Great for IP development - the interface reports when it is used wrong

Concise instantiation

Encapsulation of interface information

Added testbench and assertion value

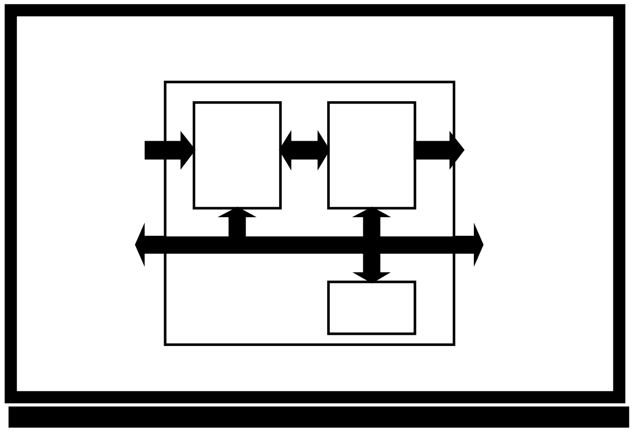
Interface Capabilities

- Interfaces can pass record data types across ports
- Interface element types can be:
  - declared
  - passed in as parameters
- An interface can have:
  - parameters, constants & variables
  - functions & tasks
  - assertions

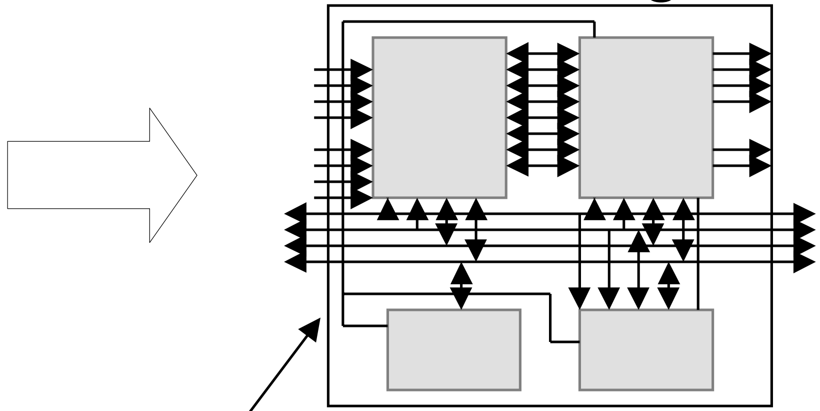
SystemVerilog Interfaces

Verilog-2001 style design

Design On A White Board

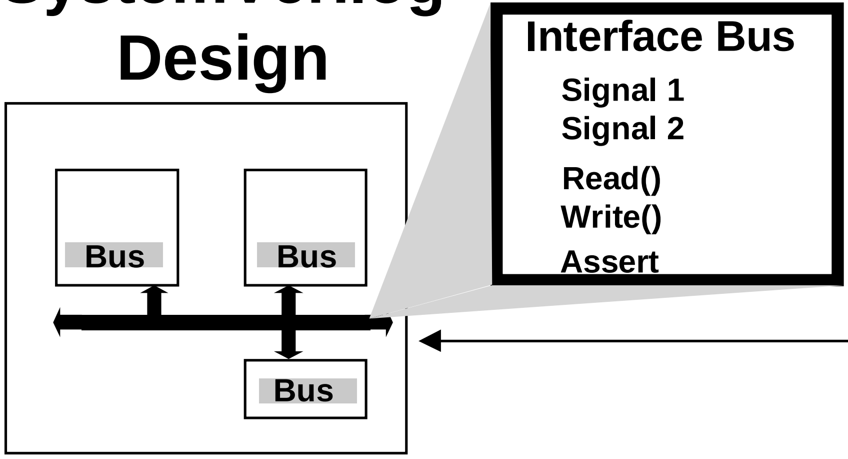


HDL Design



Complex signals
- Bus protocol repeated in blocks
- Hard to add signal through hierarchy

SystemVerilog Design



Communication encapsulated in interface
- Reduces errors - easier to modify
- Significant code reduction - saves time
- Enables efficient transaction modeling
- Allows automated block verification

What is an Interface?

- Provides a new hierarchical structure
  - Encapsulates interconnect and communication
  - Separates communication from functionality
  - Eliminates "wiring" errors
  - Enables abstraction in the RTL

A simple interface is a bundle of wires

```
int i;
logic [7:0] a;

interface intf;
  int i;
  logic [7:0] a;
endinterface : intf
```

Just like a simple struct is a bundle of variables

```
int i;
logic [7:0] a;

typedef struct {
  int i;
  logic [7:0] a;
} s_type;
```

Referencing Interface Variables & Functionality

- Interface *variables* are referenced relative to the interface instance name as *if_instance_name*.*variable*

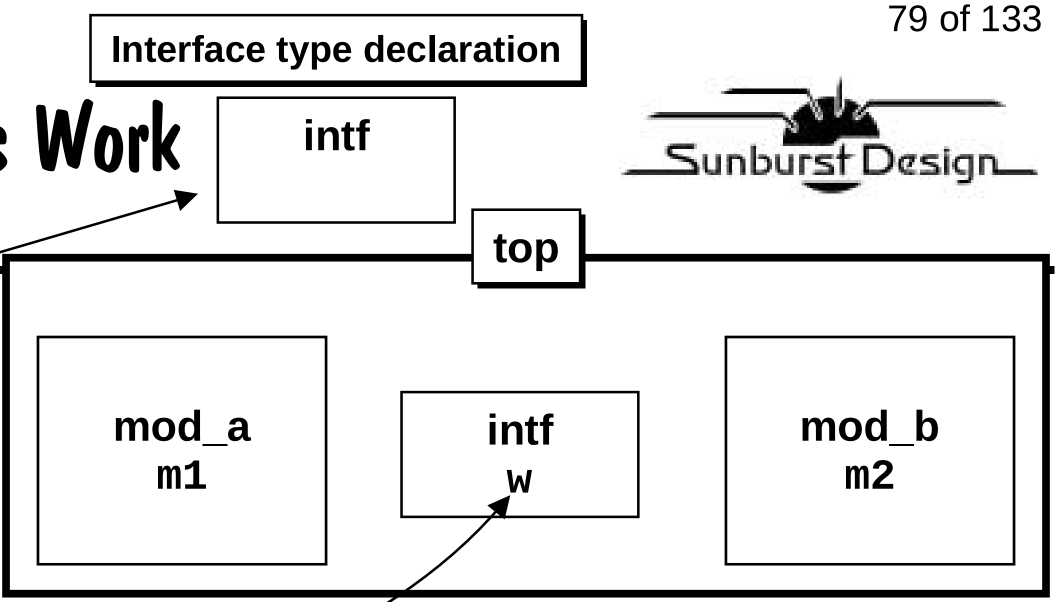
- Interface *functions* are referenced relative to the interface instance name as *if_instance_name*.*function*

- *Modules connected via an interface:*
  - Can call the interface task & function members to drive communication
  - Abstraction level and communication protocol can be easily changed

    Replace an interface with a new interface containing the same members

Interfaces change *but connected modules do not change*

How Interfaces Work



```
interface intf;
  logic a, b;
  logic c, d;
  logic e, f;
endinterface
```

```
module top;
  intf w ();

  mod_a m1 ();
  mod_b m2 ();
endmodule
```

Instantiate interface

Instantiate mod_a and mod_b

```
module mod_a;
endmodule
```

```
module mod_b;
endmodule
```

No ports yet

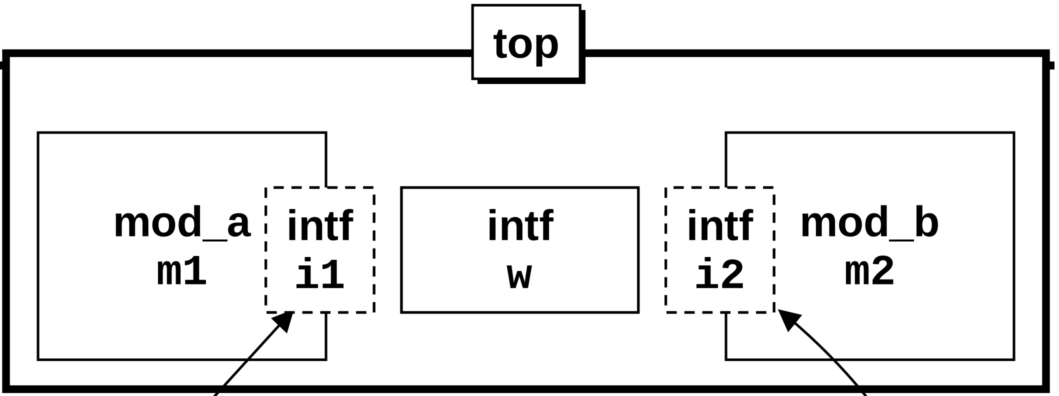
How Interfaces Work

```
interface intf;
  logic a, b;
  logic c, d;
  logic e, f;
endinterface
```

```
module top;
  intf w ();

  mod_a m1 ();
  mod_b m2 ();
endmodule
```

```
module mod_a (intf i1);
endmodule
```

```
module mod_b (intf i2);
endmodule
```



Add interface references to mod_a and mod_b

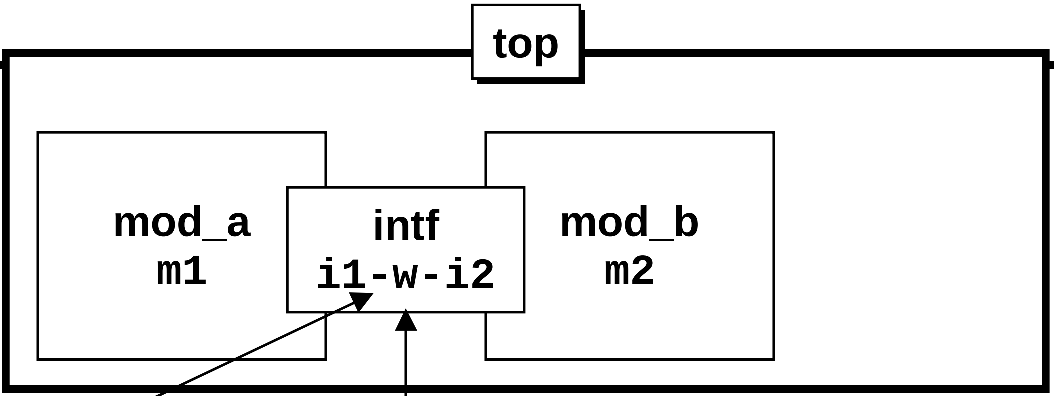
How Interfaces Work

```
interface intf;
  logic a, b;
  logic c, d;
  logic e, f;
endinterface
```

```
module top;
  intf w ();

  mod_a m1 (.i1(w));
  mod_b m2 (.i2(w));
endmodule
```

```
module mod_a (intf i1);
endmodule
```

```
module mod_b (intf i2);
endmodule
```



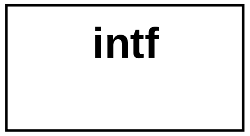
An interface is similar to a module straddling two other modules

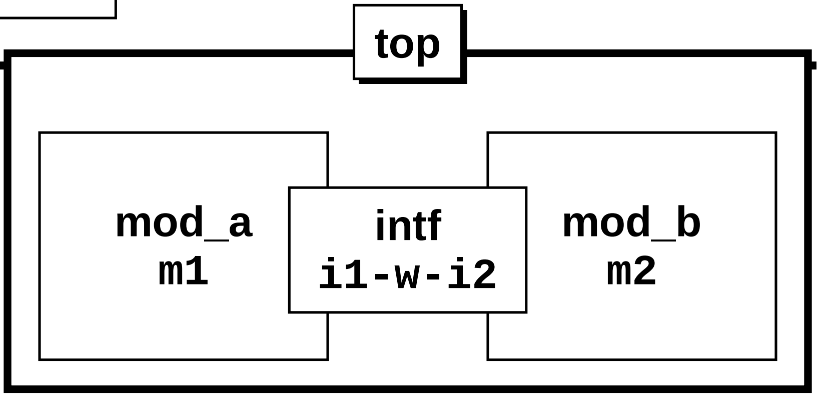
Show that the `w` instance of the interface is *aliased* to the `i1` reference in `mod_a` and the `i2` reference in `mod_b`

Interfaces can also contain `modports` (`modports` are discussed later)

Interface Usage

Interface type declaration





```
interface intf;
  logic a, b;
  logic c, d;
  logic e, f;
endinterface
```

Interface type declaration

```
module top;
  intf w ();

  mod_a m1 (.i1(w));
  mod_b m2 (.i2(w));
endmodule
```

Interface instantiation

Instantiated interface *aliased* to referenced interface access names

```
module mod_a (intf i1);
endmodule
```

```
module mod_b (intf i2);
endmodule
```

References to defined interface types

Local access names for the referenced interfaces

Interface

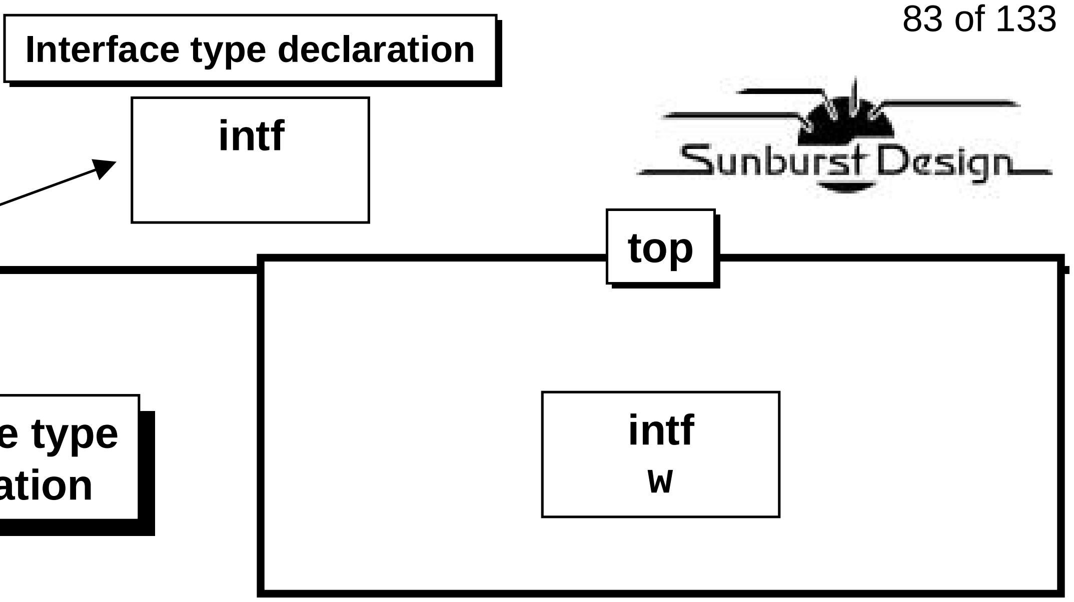
Legal Usage

```
interface intf;
  logic a, b;
  logic c, d;
  logic e, f;
endinterface
```

Interface type declaration

```
module top;
  intf w ();


endmodule
```

Interface instantiation



It is legal to declare and instantiate an interface without accessing the interface nets and variables ...

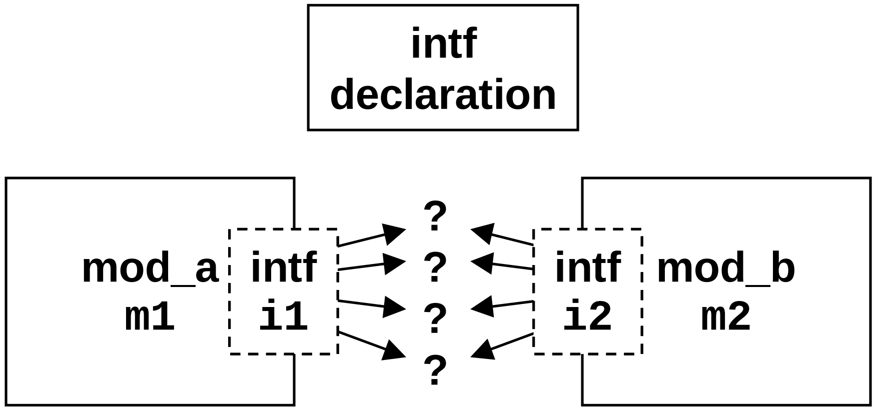
... but not very useful !

Interface

Illegal Usage

```
interface intf;
  logic a, b;
  logic c, d;
  logic e, f;
endinterface
```

Interface type declaration



ERROR:
(1) The interface was declared
(2) The interface was referenced
(3) The interface was *never* instantiated

```
module mod_a (intf i1);
endmodule
```

```
module mod_b (intf i2);
endmodule
```

References to defined interface types

Local access names for the referenced interfaces

Simple Interfaces

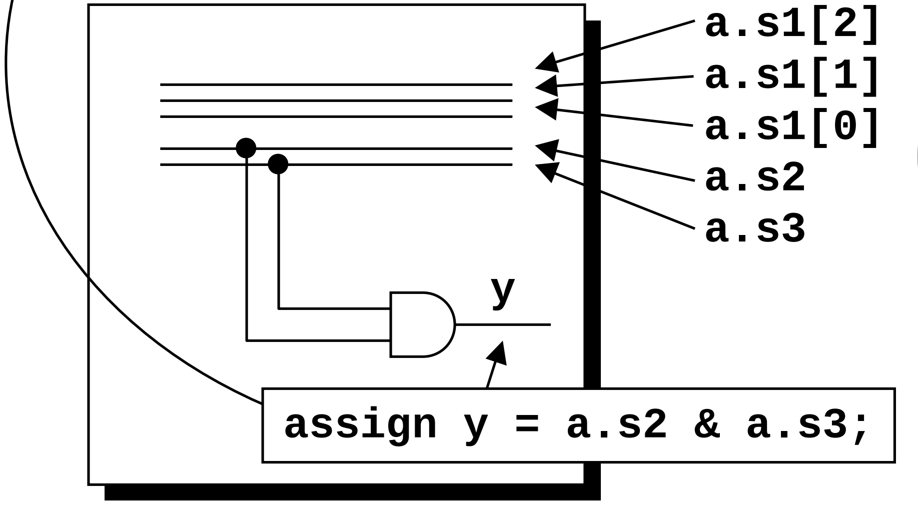
```
interface blk_if;
  logic [2:0] s1;
  logic       s2;
  logic       s3;
endinterface
```

An interface listed in a module port header *references* the nets and variables that are declared in the interface

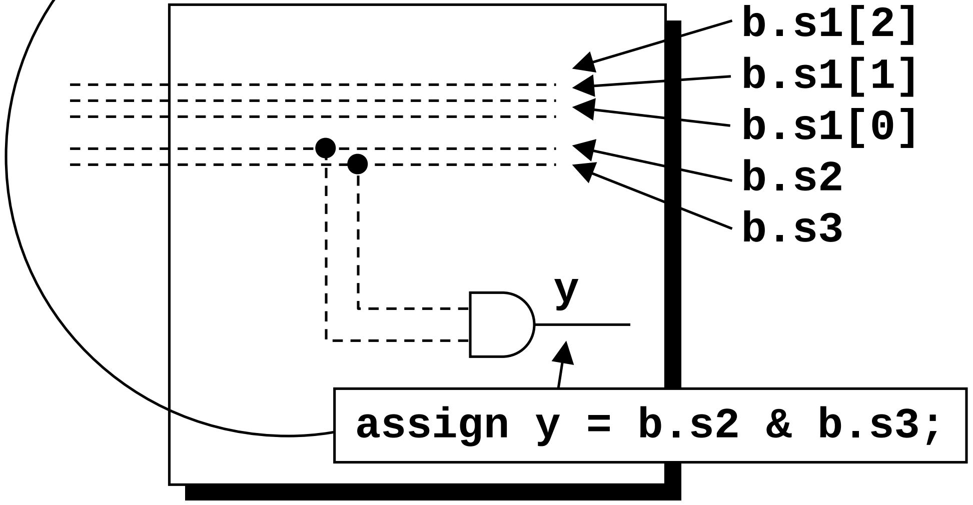
```
module m1;

  blk_if a;
  assign y = ...
endmodule
```

Interfaces instantiated within a module declares a grouping of nets and variables

```
module m2 (blk_if b);

  assign y = ...
endmodule
```



```
assign y = a.s2 & a.s3;
```



```
assign y = b.s2 & b.s3;
```

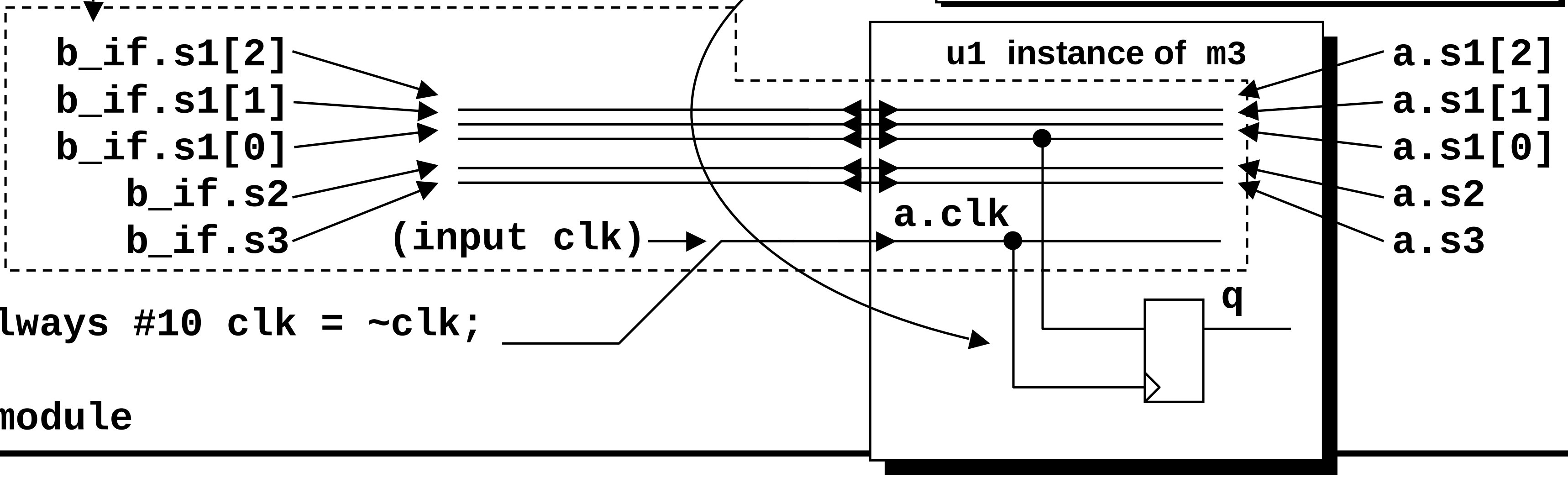
Simple Interfaces with Ports

```
interface blk_if (input bit clk);
  logic [2:0] s1;
  logic       s2;
  logic       s3;
endinterface
```

This blk_if interface has an implicit clk input

```
module tb;
  bit clk;

  blk_if b_if (.clk);

  m3 u1 (.a(b_if));
```

```
module m3 (blk_if a);

  always @(posedge a.clk)
    q <= a.s1[0];

endmodule
```



```
  always #10 clk = ~clk;

endmodule
```

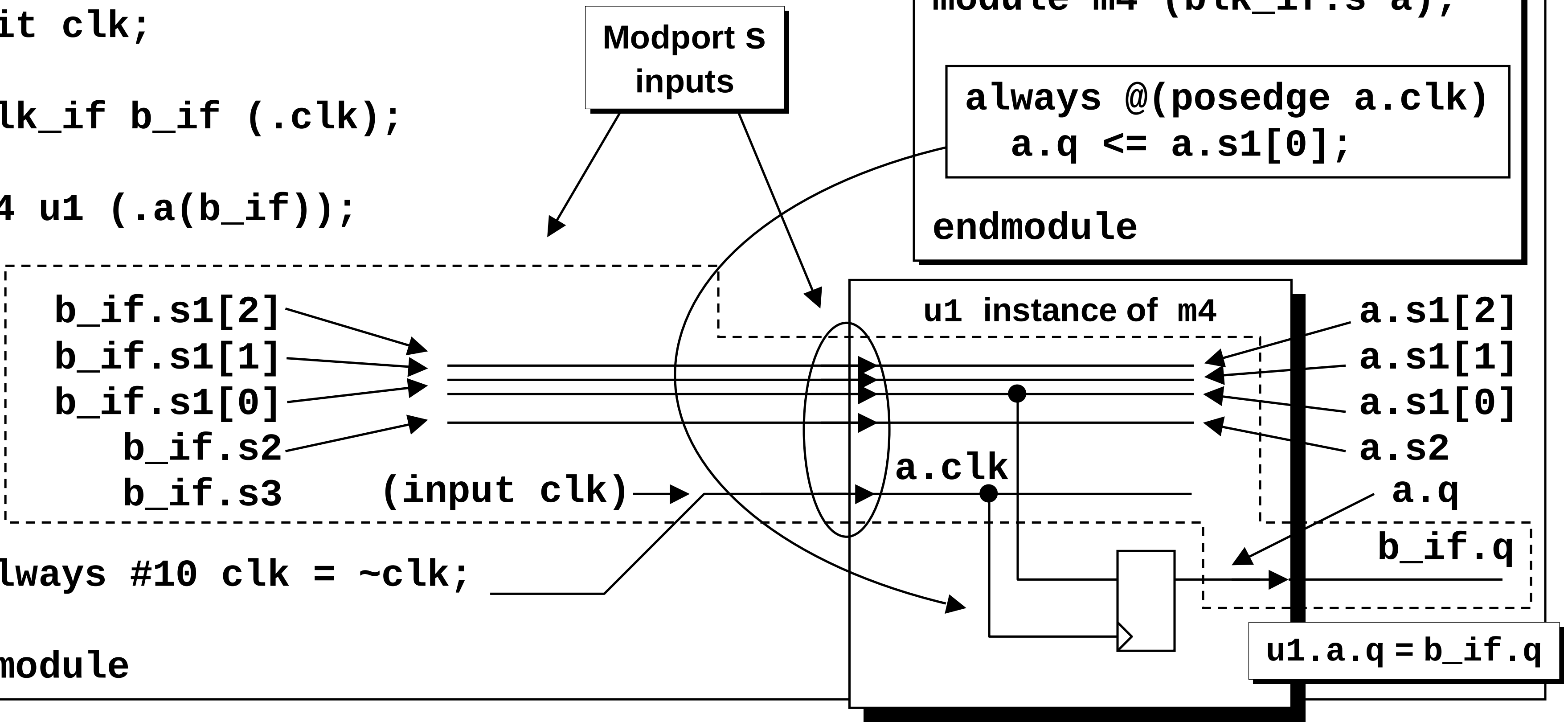
Simple Interface with Modports

```
interface blk_if (input bit clk);
  logic [2:0] s1;
  logic       s2;
  logic       s3;
  logic       q;
  modport s (input clk, s1, s2, output q);
endinterface
```

Modport s defines s1 and s2 to be inputs and q to be an output

```
module tb;
  bit clk;

  blk_if b_if (.clk);

  m4 u1 (.a(b_if));

  always #10 clk = ~clk;

endmodule
```

```
module m4 (blk_if.s a);

  always @(posedge a.clk)
    a.q <= a.s1[0];

endmodule
```



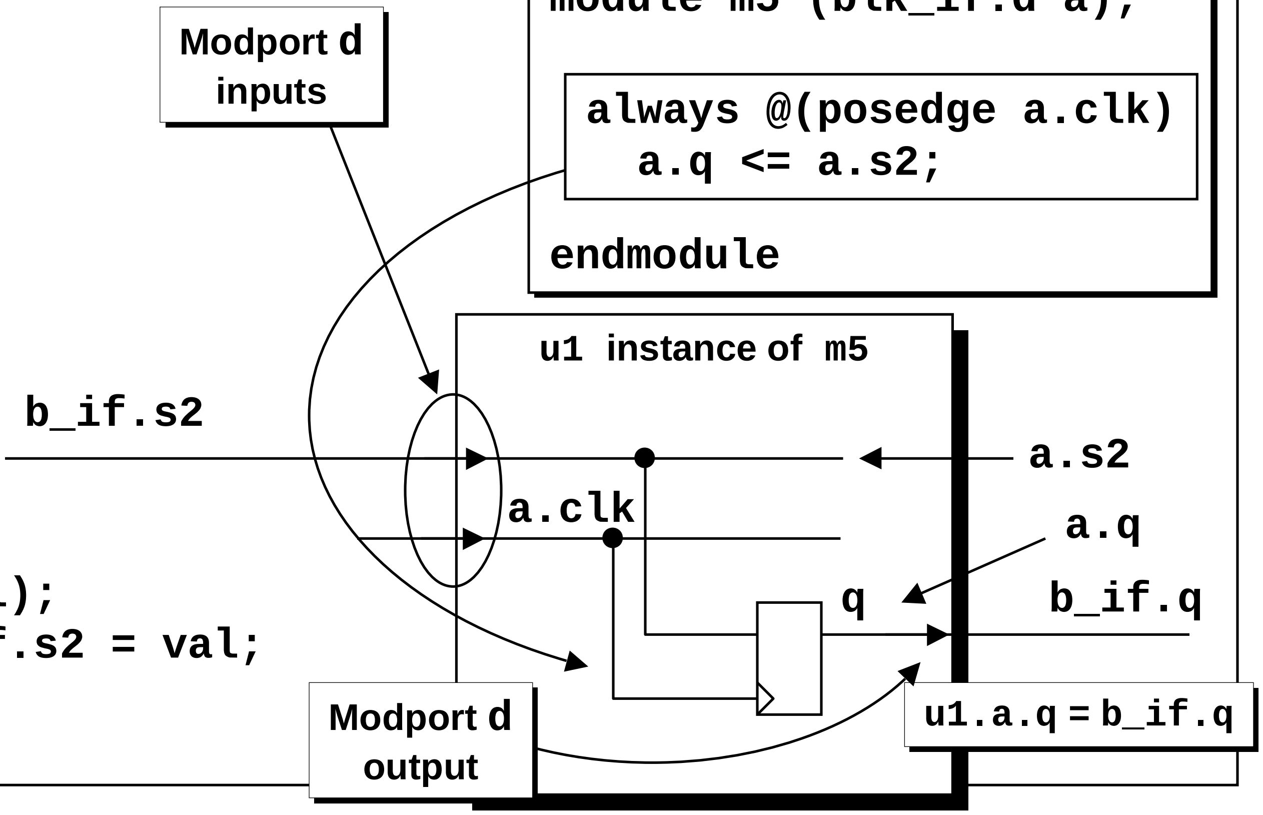
Working with Interfaces - tb tasks

```
interface blk_if (input bit clk);
  logic [2:0] s1;
  logic       s2;
  logic       s3;
  logic       q;
  modport d (input clk, s2, output q);
endinterface
```

Modport d defines input s2 and output q

```
module tb;
  bit clk;

  blk_if b_if (.clk);

  m5 u1 (.a(b_if));


  initial begin
    write(1'b1);
    ...

  task write (input val);
    @(negedge clk) b_if.s2 = val;
  endtask
endmodule
```

```
module m5 (blk_if.d a);

  always @(posedge a.clk)
    a.q <= a.s2;

endmodule
```



Interfaces tasks

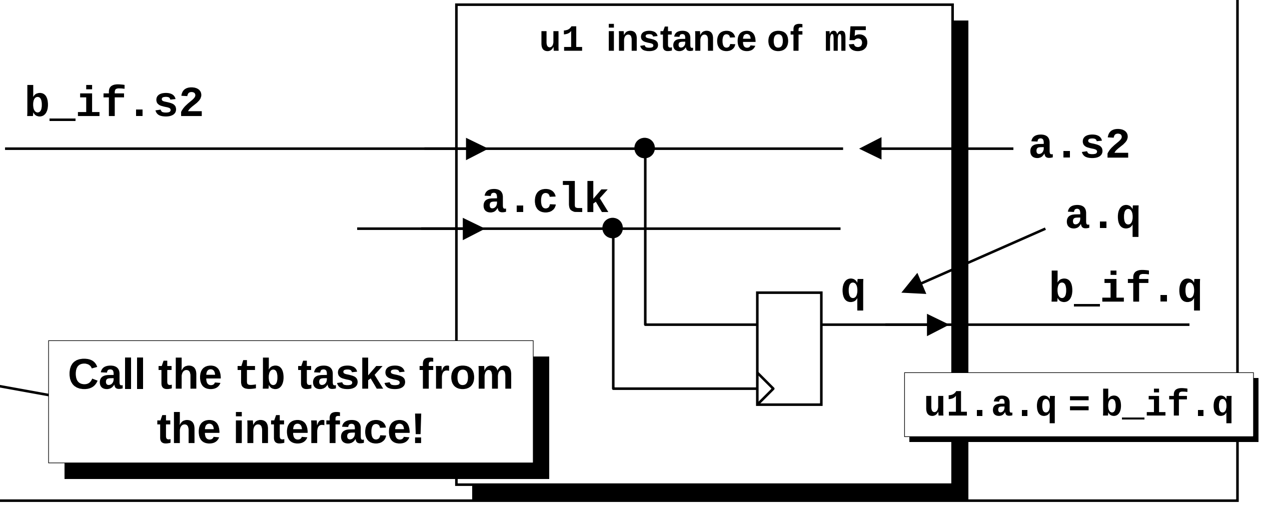
```
interface blk_if2 (input bit clk);
  logic [2:0] s1;
  logic       s2;
  logic       s3;
  logic       q;
  modport d (input clk, s2, output q);

  task write (input val);
    @(negedge clk) s2 = val;
  endtask
endinterface
```

Put the testbench tasks into the interface

```
module tb;
  bit clk;

  blk_if2 b_if (.clk);

  m5 u1 (.a(b_if));

  initial begin
    b_if.write(1'b1);
    ...

endmodule
```

Call the tb tasks from the interface!

```
module m5 (blk_if2.d a);

  always @(posedge a.clk)
    a.q <= a.s2;

endmodule
```



Interfaces - Do Not Use Net Types

- In general, an interface will connect outputs from one module to inputs of another module
  - outputs could be ***procedural*** or continuously driven
  - inputs are continuously driven
- Since the interface connects outputs to inputs:
  - The output will frequently be *procedurally* assigned
  - *Even if the output is a continuous assignment, it may be changed to a procedural assignment when connected to a difference module*
- Bi-directional and multiply driven nets
  - These require interface wire declarations

Cannot make procedural assignments to any of these interface signals (all nets - no variables!)

```
interface bad_if1;
  wire [2:0] n1;
  wire       n2;
  wire       n3;
endinterface
```

Bi-directional data bus

```
interface good_if1;
  logic [7:0] addr;
  logic       n2;
  wire  [7:0] data;
endinterface
```

SystemVerilog Enhancements for Verification

SystemVerilog Clocking & Program Blocks

- Clocking blocks (domains) and cycle-based attributes
  - To ease testbench development
  - Promote testbench reuse
  - Cycle-based signal sampling
  - Cycle-based stimulus generation (drives)
  - Synchronous samples
  - Race-free program context
- Program blocks to partition and encapsulate test code

SystemVerilog Powerful Assertion Enhancements

- Assertion mechanism for verifying
  - design intent
  - functional coverage intent

- New property and sequence declarations

- Assertions and coverage statements with action blocks

Assertions will be discussed later

SystemVerilog Verification Enhancements

- Direct Programming Interface (DPI) - DirectC ← **Briefly discussed later**
- New types:
  - 'C'-types, string, dynamic array, associative array
- Pass by reference subroutine arguments
  - better than Verilog-2001 reentrant tasks
- Synchronization:
  - Dynamic process creation
  - Process control
  - Inter-process communication.
- Enhancements to existing Verilog events
- Built-in synchronization primitives:
  - Semaphore & mailbox
- Classes & methods
  - Object-Oriented mechanism for abstraction & encapsulation

$\}$ **Download a copy of the SystemVerilog LRM**

Final Blocks

- A final block is like an initial block
  Triggers at the end of a simulation

`$finish` command

Event queue is empty

Termination of all `program` blocks (causes an implicit `$finish`)

PLI executes `tf_dofinish()` or `vpi_control(vpiFinish, ...)`

After all spawned processes are terminated

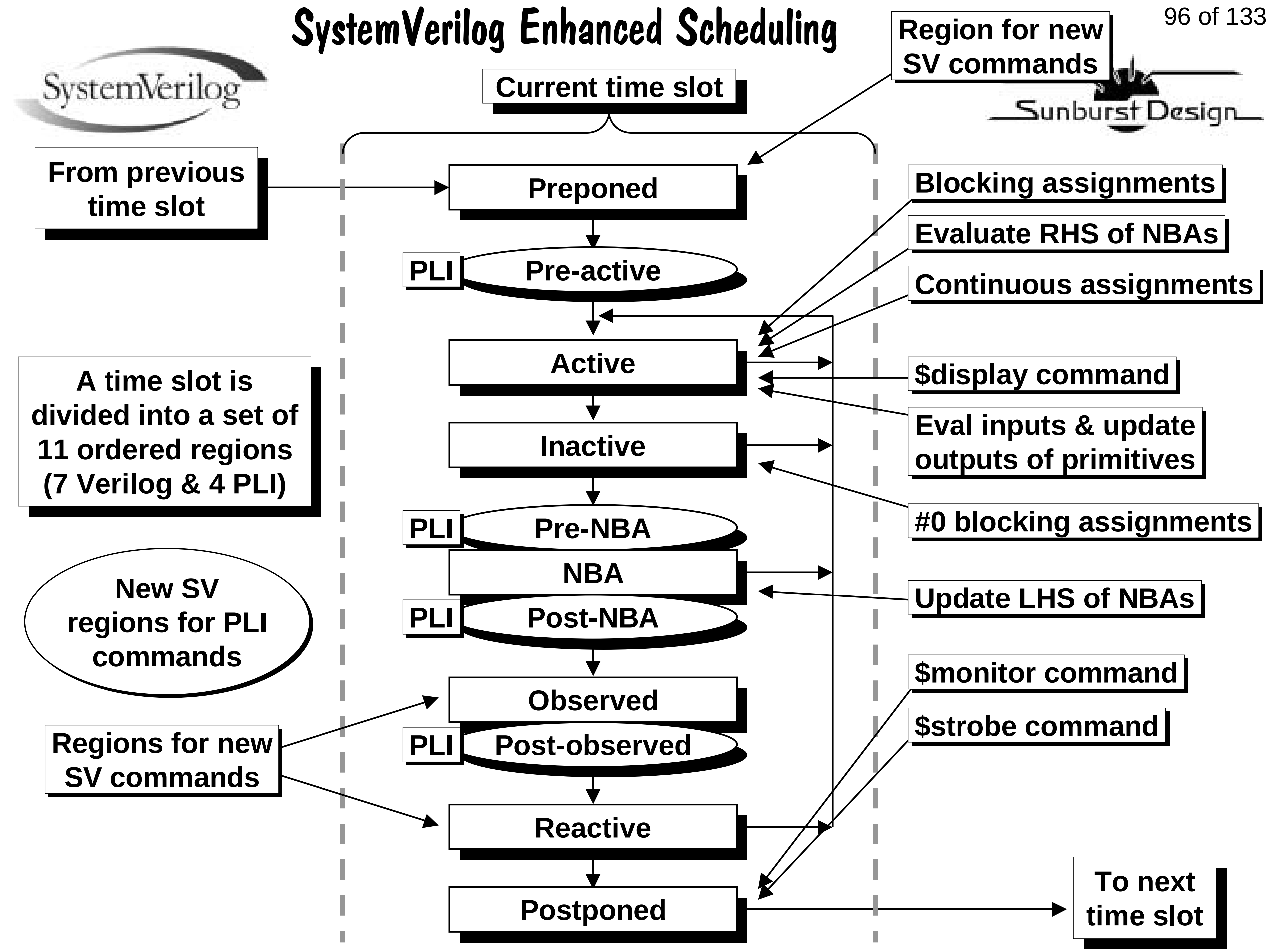
After all pending PLI callbacks are canceled

`final` block(s) execute

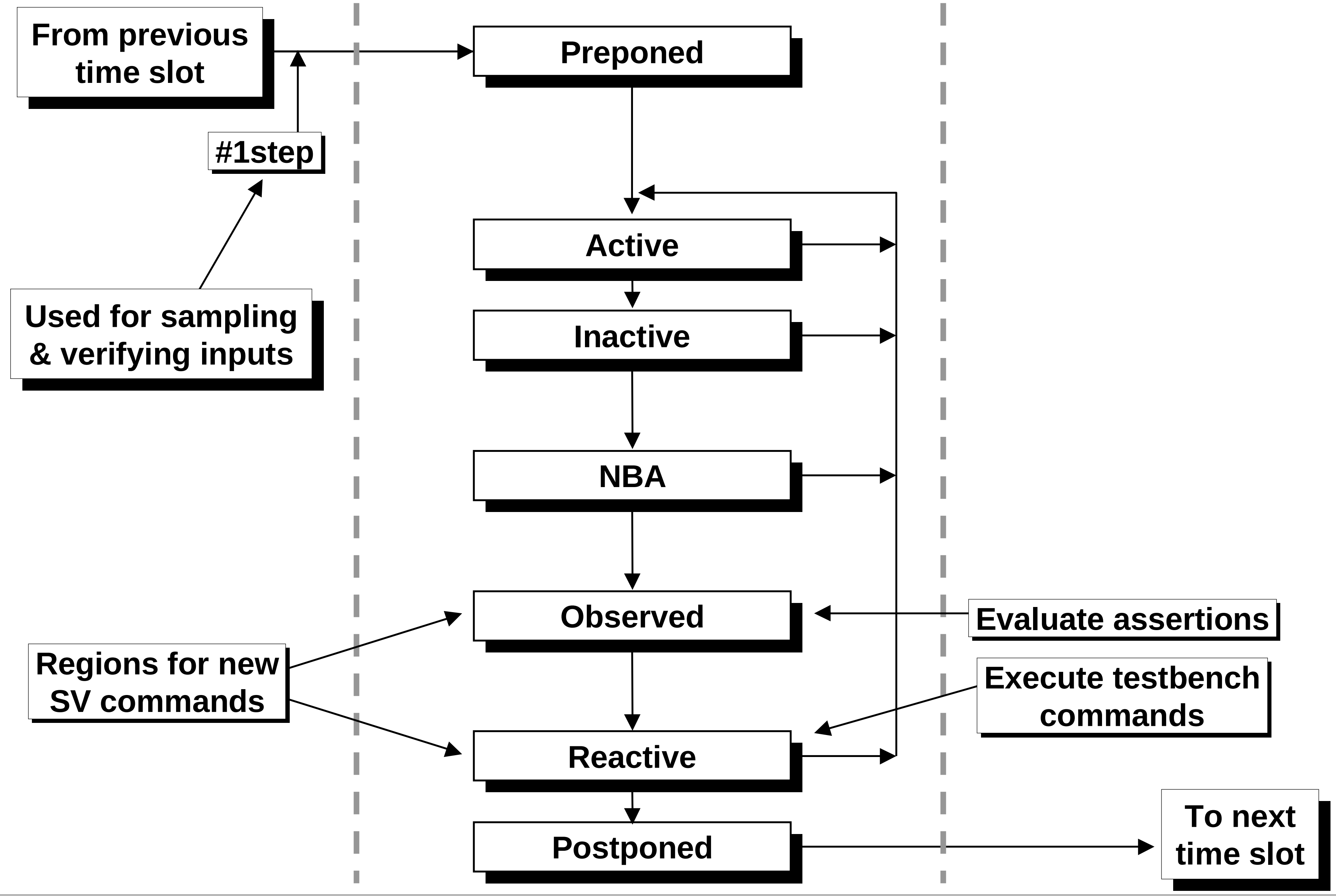
```
final begin
  if ((ERROR_CNT == 0) && (VECT_CNT != 0)) begin
    $write("\nTEST PASSED - %0d vectors", VECT_CNT);
    $write(" - %0d passed\n\n",           PASS_CNT);
  end
  else begin
    $write("\nTEST FAILED - %0d vectors", VECT_CNT);
    $write(" - %0d passed - %0d failed\n\n", PASS_CNT, ERROR_CNT);
  end
end
```

`final` blocks *cannot* have delays

SystemVerilog Enhanced Scheduling



Scheduling of New SV Commands



Clocking Block

- SystemVerilog adds a clocking block to facilitate
  - Sampling for verification
  - Driving stimulus

By `default` specified `input(s)` are sampled `10ns` before the posedge `clk`

New `clocking` & `endclocking` keywords

By `default` specified `output(s)` are driven 2ns after the posedge `clk`

```
clocking bus @(posedge clk);
  default input #10ns output #2ns;
  input  data, ready, enable = top.mem1.enable;
  output done;
  output negedge ack;
  input #1step addr;
endclocking
```

`input(s)` to sample

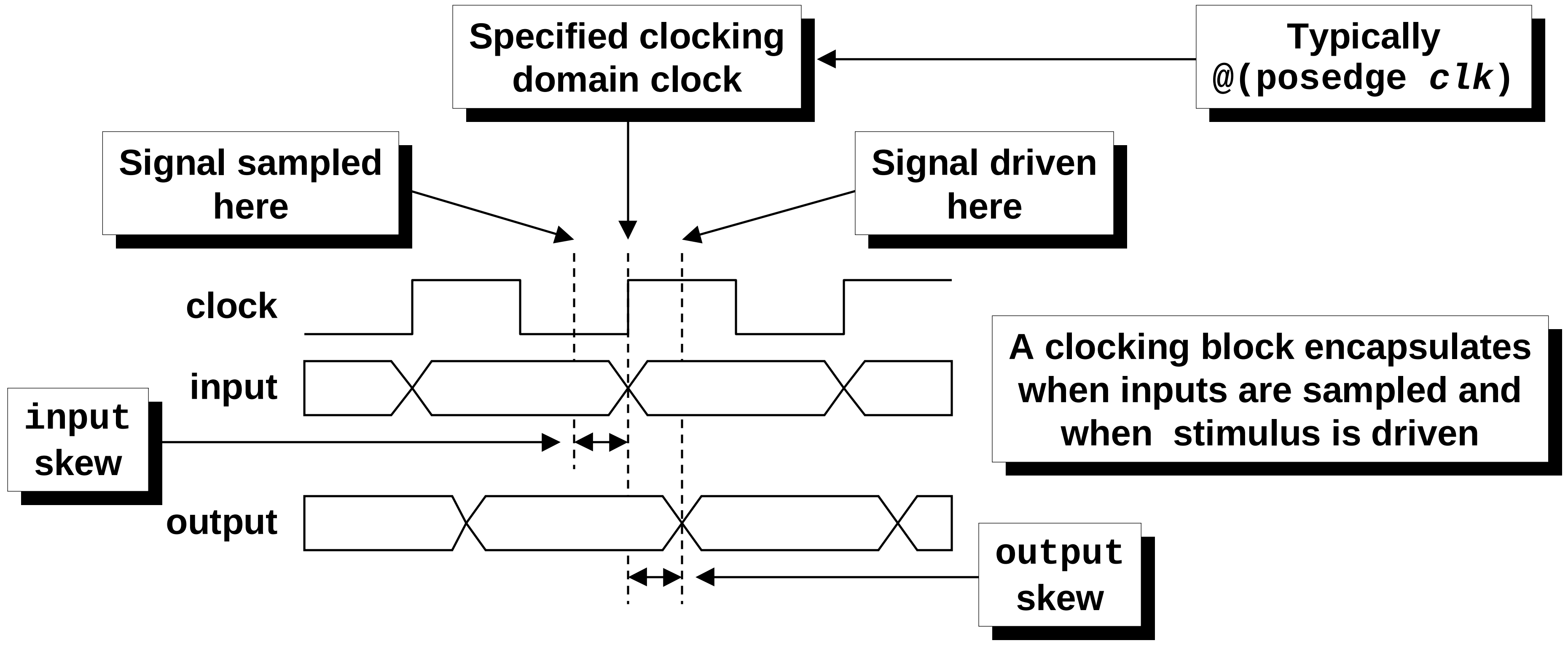
`output(s)` to drive

exception `output` (drive this one on the negedge clk)

exception `input` (sample this one on 1step before the posedge clk)

Clocking Block Skews

- Specify synchronous sample & drive times
  - Input skew is for sampling
  - Output skew is for driving
  - Default input skew is 1step - Default output skew is 0



A clocking block encapsulates when inputs are sampled and when stimulus is driven

Clocking
Synchronous Interfaces

Clocking event is posedge `wclk`

```
clocking fifo @(posedge wclk);
  default input #1step output negedge);
  input  wfull;
  output wdata, winc, wrst;
endclocking
```

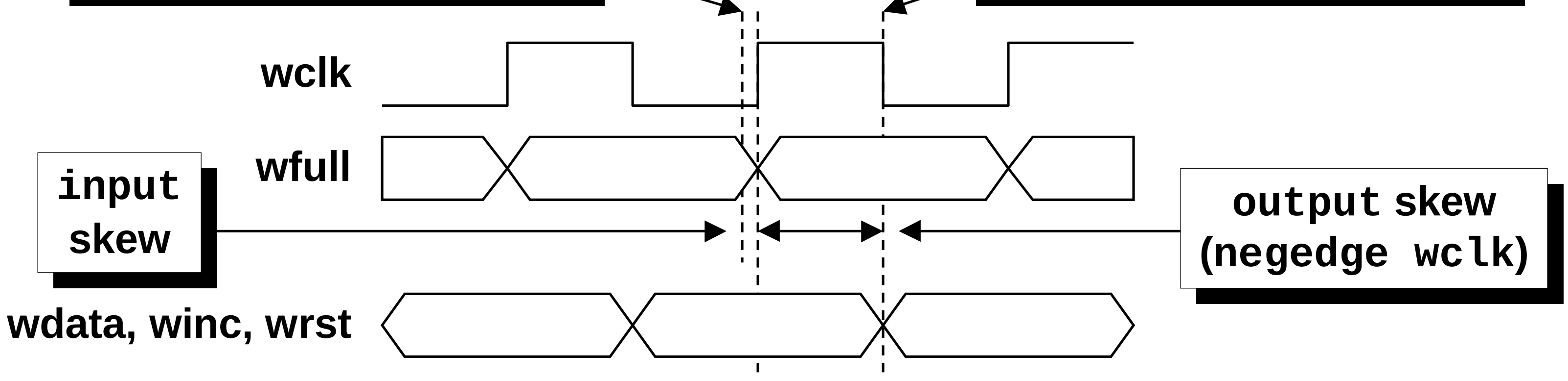
By default, sample inputs just before the posedge `wclk`

By default, drive FIFO stimulus on the negedge `wclk`

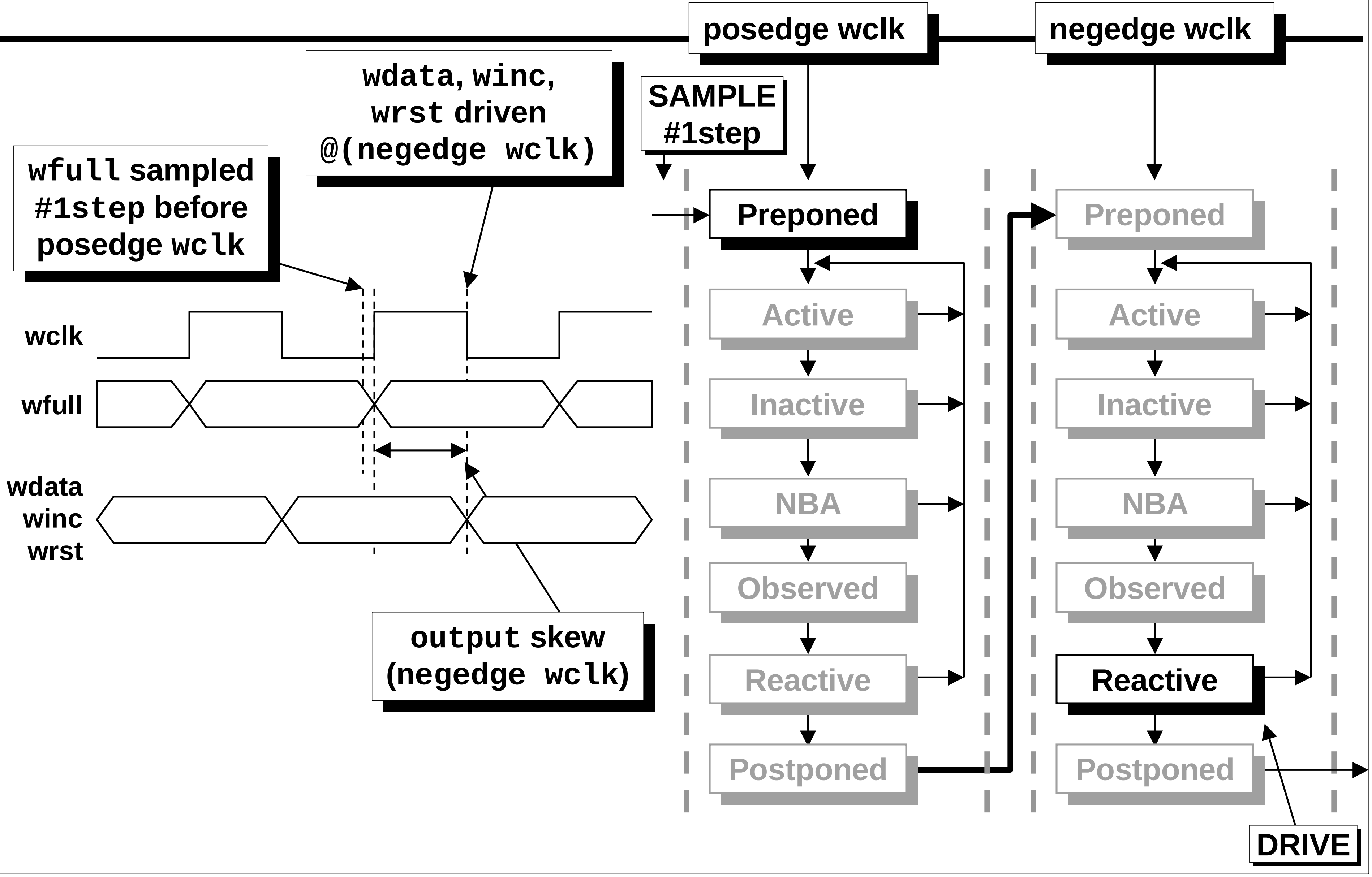
Testbench Uses:
`fifo.wfull` `fifo.wdata`
`fifo.winc` `fifo.wrst`

`wfull` sampled `#1step` before the posedge `wclk`

`wdata`, `winc`, `wrst` driven `@(negedge wclk)`



Scheduling of New SV Commands



Default Clocking & Synchronous Drives

- Designate one clocking as default

```
default clocking tb.fifo.wclk;
```

- One default permitted per module, interface, program

- Cycle Delay Syntax:

```
## <integer_expression>
##5; // wait 5 cycles
##1 fifo.wdata <= 8’hFF;

##2; fifo.wdata <= 8'hAA;

fifo.wdata <= ##2 d;
```

Wait 1 (`wclk`) cycle and then drive `wdata`

Wait 2 default clocking cycles, then drive `wdata`

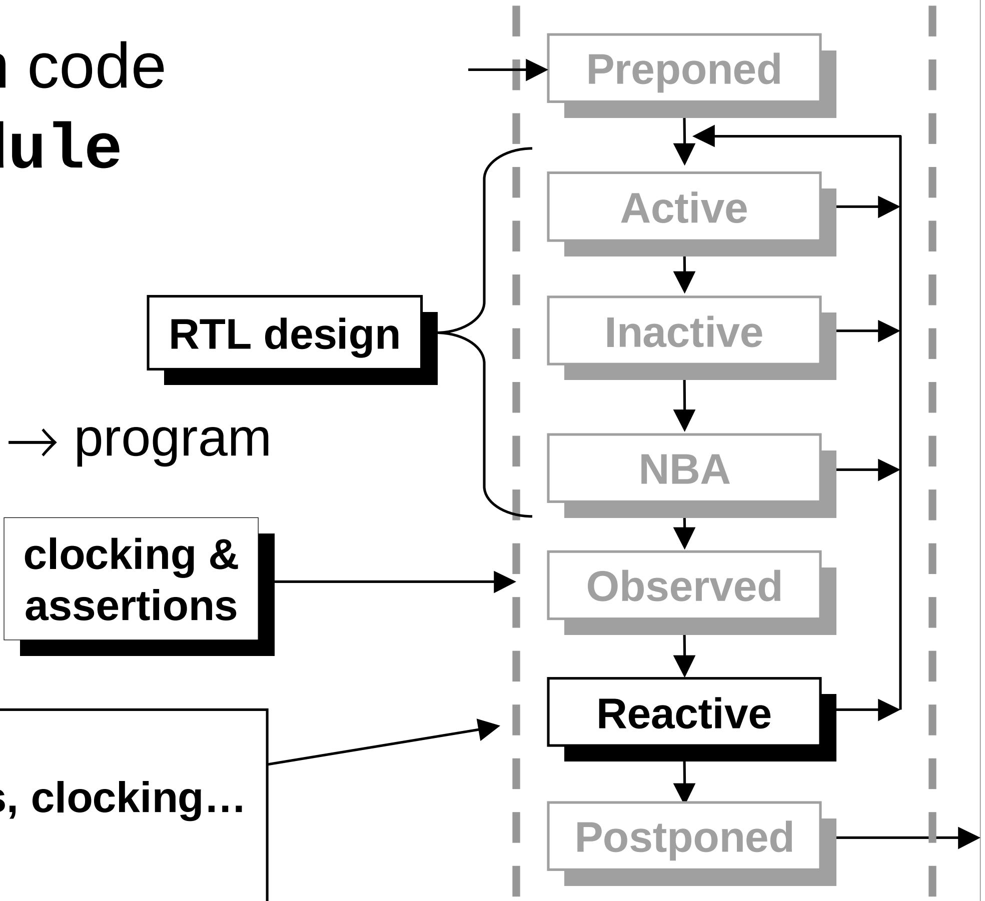
Remember the value of d and then drive `wdata` 2 (`wclk`) cycles later

Program Block

- Purpose: Identifies verification code
- A **`program`** differs from a **`module`**
  - *Only **`initial`** blocks allowed*
  - Special semantics

    Executes in *Reactive* region

    design → clocking/assertions → program



```
program name (<port_list>);
  <declarations>; // type, func, class, clocking…
  <continuous_assign>
  initial <statement_block>
endprogram
```

SystemVerilog DPI

(Direct Programming Interface - 5 Slides)

SystemVerilog Enables Easy Access to Your C Models

- Improves ease of use ← It does not require users to be PLI experts
- Allows direct calls to C functions
- Provides high level data types similar to C/C++

What is missing?

- C programs cannot call SystemVerilog tasks (Superlog allowed this) ☹
- Cannot consume time from a C program ☹
- Proposal has been made to allow exporting of SystemVerilog tasks ☺

Direct C Function Calls

DPI (Direct Programming Interface)

- SystemVerilog 3.1 - Section 26 for detailed descriptions
- Both sides of DPI are fully isolated
  - SystemVerilog does not have to analyze the C-code
  - The 'C' compiler does not have to analyze the SystemVerilog code
- Imported functions
  - C-language functions called by SystemVerilog
  - Imported functions must be declared:

```
import "DPI"  ...  function  ... ;
```

Multiple forms & options described in section 26

- Exported functions
  - SystemVerilog functions that are called by C-programs
  - Exported functions must be declared:

optional argument

```
export "DPI" [ c_identifier= ] function function_identifier ;
```

- All 'C' and SystemVerilog functions must complete in zero time

SystemVerilog & SystemC

- SystemVerilog has a built-in C-interface
  - Efficient RTL modeling & simulation
  - Efficient gate-level simulation
  - SystemC is not efficient for RTL and gate-level simulation
  - If you want to work strictly with C programming, you probably do not need SystemC

This is what most engineers want in a Verilog simulator

- If you have to write software in C++ you will probably want to make SystemC interact with SystemVerilog
  - Software trained engineer will be comfortable with SystemC
  - SystemC has all the power of C++ operator overloading, pointers,
  - SystemC has a helper library to help with signal level and timing details - majority of the work will be done in C++

SystemVerilog to SystemC (cont.)

- Pointers are not handled in SystemVerilog
  - Changing values by pointers - hard to detect events
- SystemC provides a common C-language syntax that may be better supported by behavioral synthesis tools

Easier for behavioral synthesis vendors to support just one 'C' coding style

SystemVerilog DPI Summary

- The DPI does not require PLI
  - The DPI makes it easy to co-simiulate SystemVerilog & C-code
  - You will link compiled-C object code with the SystemVerilog simulator
- C models running with SystemVerilog will be fast
  - No PLI interface to slow down SystemVerilog-C communication
- SystemVerilog will still be enhanced with PLI support

The PLI is still used by EDA tools to probe the structure and to interact with the design

SystemVerilog Assertions

Assertion-Based Design by Foster, Krolnik & Lacey

New SystemVerilog assertions book

<u>Assertion book shows examples using:</u>
SystemVerilog Assertions (SVA)
Property Specification Language (PSL)
Open Verification Library (OVL) assertions

What Is An Assertion?

- An assertion is a "*statement of fact*" or a "*claim of truth*" about the design

"...an *assertion* is a statement about a design's *intended behavior*"

- If the assertion is not true, the assertion fails
- *Design assertions are best added by design engineers*
- An assertion's sole purpose is to ensure consistency between the designer’s intention, and what is created

Who Uses Assertions

- Assertions are in use by many prominent companies, including:
  - Cisco Systems, Inc.
  - Digital Equipment Corporation
  - Hewlett-Packard Company
  - IBM Corporation
  - Intel Corporation
  - LSI Logic Corporation
  - Motorola, Inc.
  - Silicon Graphics, Inc.

Source - Harry Foster, Adam Krolnik & David Lacey, *Assertion Based Design*, Kluwer Academic Publishers, www.wkap.nl, 2003

Bug-Detection Efficiency Using OVL Assertions

- HP ASIC Project:
  - 2,793 total assertions
  - Less than 3% overhead*
- Cisco ASIC Project:
  - 10,656 total assertions
  - Only 15% overhead*
  - Only ~50 unique assertions

*Source - Sean Smith, *Synergy between Open Verification Library and Specman Elite*, Club Verification, Verisity Users' Group Proceedings, Santa Clara, CA, March 18-20, 2002

Bug-Detection Efficiency Using Assertions

- Designers from these companies reported success :
  - **34%** of all bugs found by assertions on DEC Alpha 21164 project
  - **25%** of all bugs found by assertions on DEC Alpha 21264 project

  - **17%** of all bugs were found by assertions on Cyrix M3(p1) project

  - **25%** of all bugs were found by assertions on Cyrix M3(p2) project

750 bugs were identified prior to adding assertions

The week *after adding multiple assertions* to the design, the bug reporting rate *tripled!*

Assertions found 50% of all remaining bugs

Source - Foster, Krolnik & Lacey

Impact of RTL Assertions - Intel® Centrino™ Mobile Technology

- RTL assertions have been used at Intel for over a decade
  - Utilized by a variety of tools
- Basic combinational assertions
  - Most are either forbidden or mutual exclusion (mutex)
  - The RTL includes thousands of assertions
- RTL assertions caught >25% of all bugs!
  - Assertions were very effective in bug hunting (>25%) in the cluster test environment (CTE)
  - Second after designated cluster checkers (>50%)
  - Assertions were most effective in bug hunting (>27%) in full-chip environment
- Assertions were the first to fail
  - They were more local than checkers
  - They were mostly combinatorial
- RTL assertions shortened the debug process

Assertions helped most with full-chip debug!

Bugs found by assertions happened early in the design process

Assertions point directly to the bug

Bug-Detection Efficiency HP Update

- Update on HP ASIC project status (~August 2002):
  - ~4,300 assertions
  - ~10% simulation overhead
  - ~85% of total bugs reported in a one year period

- How did HP determine the 85% percentage?
  - Engineers cut-and-pasted OVL error messages into the bug reports
  - grep'ed the bug reports to determine % of OVL errors
  - Actual percentage may be higher

... engineers do not report all RTL bugs found

Source - Harry Foster - personal communication

Who Should Add Assertions?

- Primary source: **THE DESIGNER!!**
  - Assertions communicate the *designers intent*
  - Added as *facts about the design* as they are recognized
- Secondary source: the Verification Engineer
  - Assertions must still be added by the designer
  - Verification engineers may need to *tutor the designer* early in the project
- The designer will become more **ASSERT**ive on the next project!

For the Design Engineer ...

***Assertions are active design comments* used to document intended behavior of a design and help to identify both design and verification flaws related to the design under development and test**

Assertion Benefits

- Two testing requirements
  - Proper input stimulus required to activate a bug
  - Proper input stimulus required to propagate bug effects to DUT outputs

Assertions only require proper input stimulus!

Assertions improve observability!

DEC Alpha team and Cyrix M3 team added assertions after the "simulation complete" point ...

... *and found additional bugs using the same set of tests* that previously passed

- Reduces debug time
  - Assertions improve observability
  - HDL verification does not detect bugs directly in the offending logic

Assertions enable bug detection exactly when and where they occur

HDL testing typically detects a bug multiple clocks after it happened

HDL testing often shows bugs at some other distant location in the design

Source - Foster, Krolnik & Lacey

Assertion Methodology for New Designs (Key Learnings*)

Assertions in Design Reviews

- Make assertions an integral part of the design review process
  - Design engineers have found design bugs just by analyzing the type of assertion needed for a location within the design

Bugs have been detected just by adding assertions

- *There is a cost to adding assertions -* teams must accept:
  - Slightly longer RTL implementation process
  - Significantly shorter debug process
  - Problems creep into the design during creation

1% - 3% increase in RTL coding time to add assertions

Up to 50% reduction in verification time by adding assertions

New code ... new bugs!

* Source - Foster, Krolnik & Lacey

Best Assertion Practices

- Co-locate RTL assertions with the design code they validate
  - Better documents the code
  - Helps identify portions of the code that have or are missing assertions

Keep assertions close to relevant RTL code

- Design & develop IP with assertions

Create IP with assertions already embedded

- Track identified problems
  - Did *directed tests* find the bug?
  - Did *random tests* find the bug?
  - Did *assertions* find the bug?

Tracking this information will help justify assertion methodologies

- Analyze failures not identified by assertions

Can new assertions be written to detect the bug?

Assertion Density

Assertion density = number of assertions / line of code

- **General guideline:** anywhere you typically add a comment to document a potential *concern*, *assumption*, or *restriction* in the RTL, this is an ideal location to add an assertion

Add assertion to module interfaces

- Block interface assertions
  - Different designers may interpret the interface specification differently
  - Add all module interface assertions at the top of the RTL module

This keeps them close to the interface signals' definitions

"If a person can't write an assertion for a block or chip interface, he or she probably is not clear about the interface"

Source - Foster, Krolnik & Lacey

Assertion Methodology for Existing Designs

- Adding assertions to a mature design loses some of the benefits of capturing early designer assumptions
  - *But adding assertions to existing designs still yields benefits*
- Cyrix design:
  - Bug report rate tripled after assertions were added

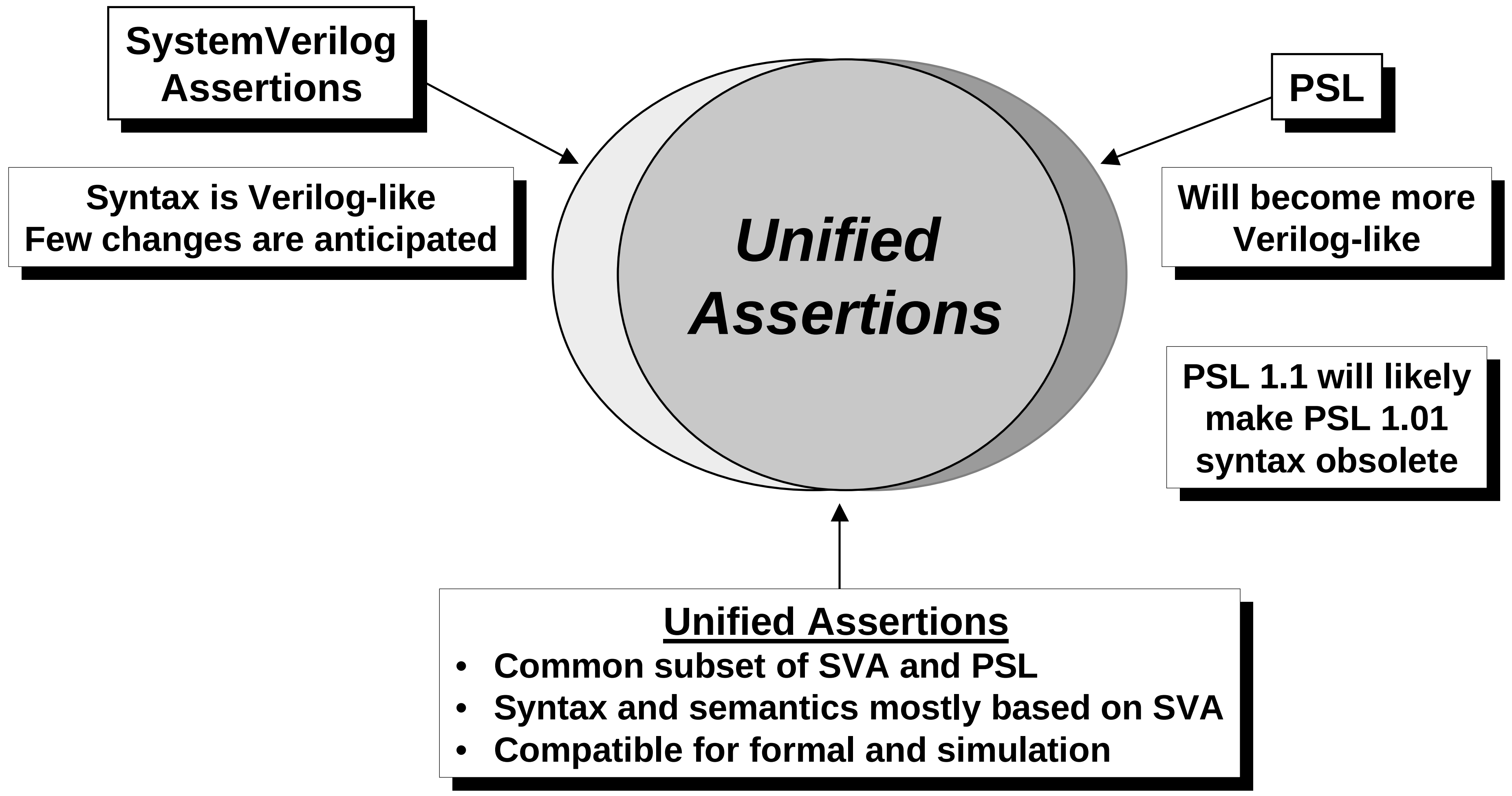
20 issues per week increased to 60 issues per week

The time required to close out problems fell from 7 days to 2 days

- Good locations for assertions:
  - Comments like “this will never occur” or “either of these will cause . . .”
- Write assertions for block interfaces ←

Interfaces are always a good place for assertions

SystemVerilog Integrates Assertions Into The Verilog Language



SystemVerilog Assertions

Syntax is Verilog-like
Few changes are anticipated

PSL

Will become more Verilog-like

PSL 1.1 will likely make PSL 1.01 syntax obsolete

<u>Unified Assertions</u>

- **Common subset of SVA and PSL**
- **Syntax and semantics mostly based on SVA**
- **Compatible for formal and simulation**

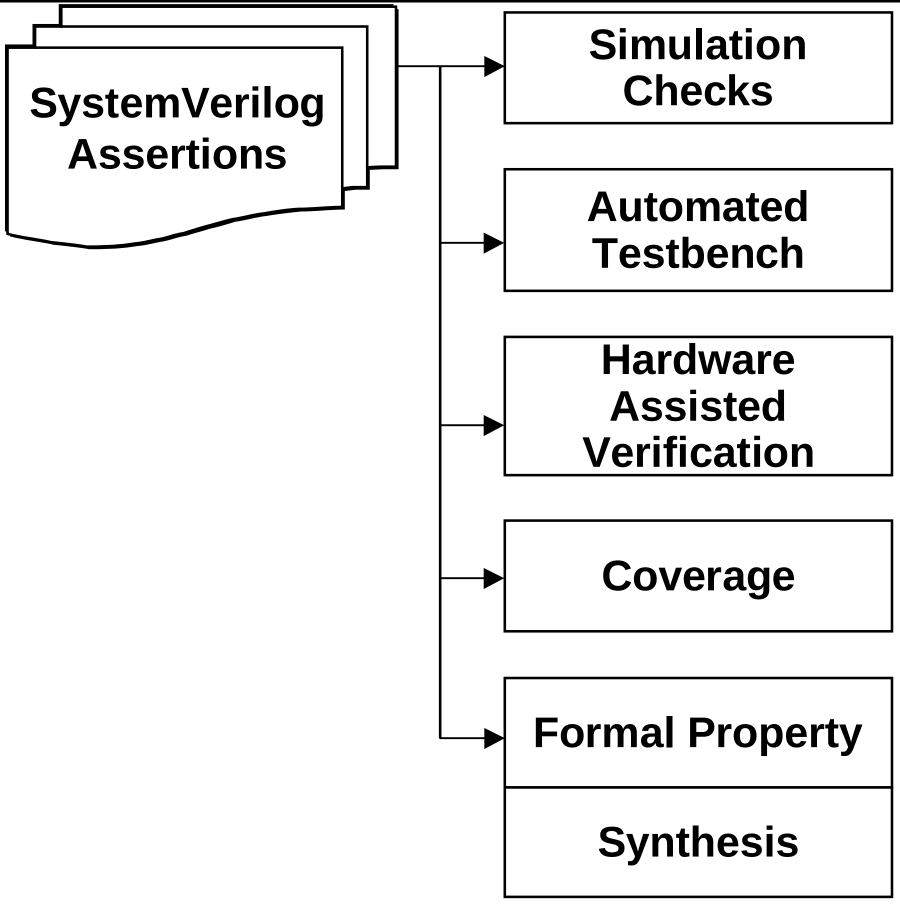
Source:
http://www.accellera.org/membermeetdac.html

SystemVerilog Provides Powerful Assertion Capabilities

- Accessible to every designer
- Same familiar Verilog-like language

Fast learning curve

- Part of SystemVerilog
  - No pragmas or specialized language for maximum user productivity
  - Easily usable by both design and verification engineers
- Flexible usage – inlined or in a separate file
- Seamless interaction with testbench and debugging utilities for a simple verification flow



SystemVerilog Assertions

- SystemVerilog assertion capabilities are much greater than OVL assertions
  - SystemVerilog adds two types of assertions
- Immediate assertions
  - Executed like a statement in a procedural block
  - Primarily intended to be used with simulation
  - Uses the keyword: `assert`
- Concurrent assertions
  - Based on clock semantics
  - Used to sample values of variables
  - Uses the key words: `assert property`

Forbid Consecutive Address Strobes

```
property no_two_ads;
  @(posedge busclk)
        disable iff (rst) not (ADSOut [*2]);
endproperty

assert property (no_two_ads);
```

Disable assertion testing when `rst` is high

`ADSOut` should never be asserted on 2 consecutive posedge `busclk`'s

Alternatively:

PSL-like *non-overlapping* implication `|=>`

If this is true in the current cycle ...

... this should NOT be true starting in the next cycle

```
property no_two_ads;
  @(posedge busclk)
    disable iff (rst) (ADSOut |=> !ADSOut);
endproperty
```

`ADSOut` should be followed by `!ADSOut` on the next posedge `busclk`

SystemVerilog

Bus-Request Handshake Assertion

Sunburst Design

own goes high in 1-5 cycles then bus request (breq) should go low 1 cycle after own goes high

```
sequence own_then_release_breq;
  ##[1:5] own ##1 !breq
endsequence

property legal_breq_handshake;
  @(posedge busclk) disable iff (rst)
  $rose(breq) |-> own_then_release_breq;
endproperty

assert property (legal_breq_handshake);
```

PSL-like *overlapping* implication |->

Wait for breq to go high (`$rose`) and then look for the `own_then_release_breq` sequence

Assert the `legal_breq_handshake` property

FSM Related Assertions

- The designer claims (asserts):
  - An FSM must only transition as follows:

    Valid state transitions: ***DETECT - LOADING - DETECT*** **-OR-**

    Valid state transitions: ***DETECT - LOADING - DELIVERY - DETECT***

Find DETECT

Disable testing for this clock cycle if rst is high

... followed by ...

DETECT - LOADING - DETECT

DETECT - LOADING - DELIVERY - DETECT

```
property LOADING_valid_transitions (@(posedge clk) disable iff (rst)
  (state==DETECT |=>
    (state==LOADING ##1 state==DETECT) or
    (state==LOADING ##1 state==DELIVERY ##1 state==DETECT)));
endproperty

assert property (LOADING_valid_transitions)
  else $error("Illegal state transition into or out of LOADING\n");
```

Assertion Severity Tasks

- SystemVerilog defines assertion failure-reporting (or info-reporting) system tasks

```
$fatal ( ... );
```

Reports run-time fatal message

Terminates simulation with an error code

Default

```
$error ( ... );
```

Reports run-time error message

Does not terminate simulation

If the assertion fails and there is no `else` clause, `$error` is called by default

```
$warning ( ... );
```

Reports run-time warning (can be suppressed - tool specific)

```
$info ( ... );
```

Reports general assertion info (no specific severity)

Assertion System Functions

- SystemVerilog defines a small set of standard system function assertions
  - $onehot (<expr>)
  - $onehot0 (<expr>)
  - $inset (<expr>, <expr> {, <expr})
  - $insetz (<expr>, <expr> {, <expr})
  - $isunknown (<expr>)

True if only one bit in the expression is 1

True if the expression is all 0's or if only one bit in the expression is 1

True if the first expression matches one of the other expressions

Same as `$inset` except ? & `Z` are treated as "don't-cares"

True if any bit in the expression is `X` (^expression === `1'bx`)

The return type of these assertions is bit
1 = true / 0 = false

SystemVerilog Assertions

- SystemVerilog has a rich set of assertion capabilities
  - See SystemVerilog 3.1 - Section 17

  - Immediate assertions
  - Concurrent assertions
  - Boolean expressions
  - Sequences
  - Property definitions
  - Multiple clock support
  - Binding properties to instances

Accellera SystemVerilog Update & EDA Vendor Fair

Plan for the future!

Take time to talk to the vendors about SystemVerilog products and solutions

Again, a special thanks to HP for providing the workstations for the vendor demos

SystemVerilog Symposium
Track I: SystemVerilog Basic Training

Thank you for taking time to come and listen!