

Agenda

Introduction:

SystemVerilog Motivation

Vassilios Gerousis, Infineon Technologies
Accellera Technical Committee Chair

Session 1:

SystemVerilog for Design

Language Tutorial

Johny Srouji, Intel

User Experience

Matt Maidment, Intel

Session 2:

SystemVerilog for Verification

Language Tutorial

Tom Fitzpatrick, Synopsys

User Experience

Faisal Haque, Verification Central

Lunch: 12:15 – 1:00pm

Session 3: SystemVerilog Assertions

Language Tutorial

Bassam Tabbara, Novas Software

Technology and User Experience

Alon Flaisher, Intel

Using SystemVerilog Assertions and Testbench Together

Jon Michelson, Verification Central

Session 4: SystemVerilog APIs

Doug Warmke, Model Technology

Session 5: SystemVerilog Momentum

Verilog2001 to SystemVerilog

Stuart Sutherland, Sutherland HDL

SystemVerilog Industry Support

Vassilios Gerousis, Infineon

End: 5:00pm



Agenda

- Guiding Principle
- Data Organization
- Capturing More Design Intent
- Powerful Syntax
- Communication Encapsulation
- Summary



Guiding Principle

- Design needs to use the subset of SystemVerilog that is synthesizable
 - Critical for Design and Formal Verification
 - SV has many new features that may someday have synthesis semantics, but that's not pragmatic for the near-term
 - Need to start with features that are easily reachable for synthesis.
- There's a significant subset of SV that's synthesizable from the SV 3.0 donation.
- All constructs covered in this presentation are potentially synthesizable without creating wild, new semantics



Data Organization

- Signals are Meaningful In Groups
 - Instructions: Operation, Operands
 - Packets: Address, Data, CRC
- Verilog Provides Only Informal Grouping

```
reg [47:0] PktSrcAdr;  
reg [47:0] PktDstAdr;  
reg [7:0] InstOpCde;  
reg [7:0] InstOpRF [127:0];
```

By Name

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

By Vector Location



Data Organization

- Desire to organize data as in a high-level programming language so others can see explicit, meaningful relationships between data elements
- SystemVerilog Structs, Unions & Arrays alone or combined better capture design intent



Data Organization - Structs

- Structs Preserve Logical Grouping
- Reference to Struct yields longer expressions but facilitates more meaningful code

```
struct {  
    addr_t SrcAdr;  
    addr_t DstAdr;  
    data_t Data;  
} Pkt;
```

```
Pkt.SrcAdr = SrcAdr;  
if (Pkt.DstAdr == Node.Adr)
```



Data Organization – Packed Structs

- Preserve Logical Grouping as with Unpacked Struct
- Packed Struct Enables Convenient Access To
 - Struct Names *OR*
 - Arbitrary Bits

```
struct packed {  
    addr_t SrcAdr;  
    addr_t DstAdr;  
    data_t Data;  
} pPkt;  
  
IntRF[addr].Data = pPkt.Data;  
  
if (Node.Seg !=  
    pPkt[DstOffset+SegOffset  
        +:SegWidth])  
  
pPkt.SrcAdr ==  
pPkt[SrcOffset+:AdrSize]
```

Data Organization - Unions

- Unpacked Unions Enable Single Variable to Contain Data from Multiple Types
- Data Read from Unpacked Union Must Be from Last Field Written
- Requires Type Awareness
- Unclear For Synthesis
- Even with all integer types how many bits?

```
typedef union {
    byte_t [5:0] bytes;
    real rlevel;
    integer ilevel;
} Data_u_t;

struct {
    Data_u_t Data;
    logic isBytes;
    logic isReal;
    logic isInteger;
} DPkt;

DPkt.Data.rlevel = 3.124;
DPkt.IsReal = 1;
if (DPkt.IsReal)
    realvar = DPkt.Data.rlevel;
```

Data Organization – Packed Unions

- Packed Unions Enable Multiple Namespaces for Same-Sized, Integer Data
- Packed Unions Enable Many Convenient Name References

```
typedef logic [7:0] byte_t;
typedef struct packed {
    logic [15:0] opcode;
    logic [1:0] Mod;
...
    logic [2:0] Base;
} Instruction_t;
typedef union packed {
    byte_t [3:0] bytes;
    Instruction_t fields;
} Instruction_u;
Instruction_u inst;
```

```
inst.fields.opcode = 16'hDEAD;
inst.bytes[1] = 8'hBE;
inst[7:0] = 8'hEF;

inst == 32'hDEADBEEF;
```

- No Need To Test Type
- Data Maps To All Members



Data Organization - Arrays

- There is More Than One:
 - Instruction stored in a Memory

```
Instruction_u memory [127:0][3:0];
```

- Packet in a Queue

```
Pkt_t pkts [7:0];
```

- Fields of Fields of Fields

```
a[4].b[3].c[2].d;
```

- Byte Arranged In a group as two 64 Byte Segments

```
logic [1:0][63:0][7:0] cachelines;
```



Data Organization - Enum

```
parameter IDLE = 3'b000;
parameter INIT = 3'b001;
...
typedef enum logic [2:0]
    {idle, init, decode ...}
    fsmstate;

fsmstate pstate, nstate;

case (pstate)
    idle: if (sync)
            nstate = init;
    init: if (rdy)
            nstate = decode;
...
endcase

typedef enum {lo,hi} byteloc;
memory[addr][hi] = data[hi];
```

- Finite State Machines
 - Currently a List of Parameters
 - Why Not A Real List of Values?
 - Enumerate formally defines symbolic set of values
- Enumerates are strongly typed to ensure assignment to value in set
- Symbolic Indexes to Make Array References More Readable



Capturing More Design Intent

- `always_*`
- `unique / priority`
- Variables vs. Nets



Design Intent – always_*

- always blocks do not guarantee capture of intent
- If not edge-sensitive then only a warning if latch inferred
- always_comb, always_latch and always_ff are explicit
- Compiler Now Knows User Intent and can flag errors accordingly

```
//OOPS forgot Else but it's
//only a synthesis warning
always @(a or b)
    if (b) c = a;

//Compiler now asks
//"Where's the else?"
always_comb
    if (b) c = a;
//Intent: Conditional
//          Assignment
always_latch
    if (clk)
        if (en) Q <= d;
//Conversely unconditionally
//assigned -is it a latch?
always_latch
    q <=d
```



Design Intent – always_comb Sensitivity

- `always @(*)` or `always_comb` blocks
 - Grow to contain large amounts of code
 - Shrink to become concurrent spaghetti

```
always_comb begin
    case (addr)
    endcase
    if (<expr>)
    else
end
```

```
always_comb
    case (addr)
    endcase

always_comb
    if (expr)
    else
```

Either way, original intent gets lost in details

Design Intent – always_comb Sensitivity

- Functions Can Help
 - Tradeoff is management of parameter lists
 - Every change/addition to function must be declared in function port list
 - But No Delays Guarantee Combinational Logic
- But Code Is Often
 - Unique
 - Scoped to One Module
 - Not suited for traditional functions

```
always_comb begin
    dec=DecFunc(addr);
    res=Sel3(expr, expr, expr);
end
```

```
always_comb begin
    StepA
    StepB
end
always_comb begin
    StepC
    StepD
end
```



Design Intent – always_* Sensitivity

- always_comb provides a solution
- Consider that always_comb derives sensitivity from
 - RHS/expr in process
 - RHS/expr of statements in Function Calls

```
logic avar, a, b, c, d, e, sel;
always_comb begin
    a = b;
    StepA();
end
function StepA
    case (sel)
        2'b01: avar = a | c;
        2'b10: avar = d & e;
        default: avar = c;
    endcase
endfunction
```



```
always @(sel, b, c, d, e)
begin
    a = b;
    case (sel)
        ...
    endcase
end
```

Design Intent – always_* Sensitivity

- Unique functions referring to signals in module scope allow progressive, top-down expression
- Each function can fluidly accomplish part of work
 - Addition/Removal of signal does not require port change(s)
- Like Macro, but much more convenient
 - No “\” Continuation
- always_comb sensitivity enables balance between
 - Capturing high-level intent
 - Preserving combinational functions
 - Coding productivity

```
always_comb begin
    StepA();
    StepB();
end
always_comb begin
    StepC();
    StepD();
end
function StepA;
endfunction
function StepB;
endfunction
function StepC;
endfunction
function StepD;
endfunction
```



Design Intent – Unique/Priority

- Priority or No Priority? Back and Forth
- Synthesis Pragmas are a way of life to convey implementation
 - full_case
 - parallel_case
- unique/priority case/if modifiers formalize expression of these pragmas
- Standardization enables Simulation, Synthesis and Formal Tools To Behave Consistently



Design Intent – Unique

- unique case:
full_case /
parallel_case
- unique if-else:
full_case/
parallel_case

```
unique case (sel)
  3'b001 : muxo = a;
  3'b010 : muxo = b;
  3'b100 : muxo = c;
endcase
```

```
unique if (sel==3'b001)
  muxo = a;
else if (sel == 3'b010)
  muxo = b;
else if (sel == 3'b100)
  muxo = c;
```



Design Intent – Priority

- 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
```

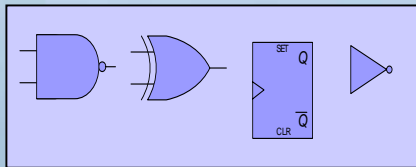
- priority if

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;
```

Design Intent - Variables vs. Net Types (Single/Multi-Driver)

- In Synthesized Design most nets have a single driver



- Combination of SystemVerilog variables and processes can enforce single driver/avoid multiple drivers
- Variables may be driven by only one of the following
 - continuous assignment
 - always_comb
 - always_latch
 - always_ff
 - module port
- Compilers Can Catch Your Mistakes

- Multiply Driven, Variable-Strength Nets generally take great care to design. Would like these nets to stand out in the design.
- Use Net Types to Distinguish

```
trireg [1:0] dbus;  
assign dbus = wen[0] ? cff : `Z;  
assign dbus = wen[1] ?  
{cff[0],cff[1]} : `Z
```

```
logic [1:0] n0,n1,n2,n3,n4;  
assign n0 = {ina,inb};  
always_comb  
    if (sela) n1 = ina;  
    else n1 = inb;  
always_latch  
    if (sela) n2 <= ina;  
always_ff @(posedge ck)  
    n3 <= ina;  
sbuf buf1 (.di(n1),.do(n4));  
sbuf buf2 (.di(n2),.do(n4));
```



Powerful Syntax

- Copy / Buffering
- Aggregate Expressions
- Arrays/For Loops/Iterator Variables
- Variables Can be Assigned Via always or continuous assign
- Netlist Simplification with .*



Syntax: Example Data Types

```
typedef logic net;  
net [7:0] byte_t;  
  
typedef struct {  
    byte_t code  
    net [2:0] group;  
} prefix_t;  
  
typedef struct packed {  
    net [1:0] mod;  
    net [2:0] regop;  
    net [2:0] rm;  
} modrm_t;
```

```
typedef struct packed {  
    net [1:0] scale;  
    net [2:0] index;  
    net [2:0] base;  
} sib_t;  
typedef struct {  
    byte_t [3:0] bytes;  
    net [2:0] cnt;  
} displacement_t;  
typedef struct {  
    byte_t [3:0] bytes;  
    net [2:0] cnt;  
} immediate_t;
```

```
typedef struct {  
    prefix_t prefix[2:0];  
    modrm_t modrm;  
    sib_t sib;  
    displacement_t disp;  
    immediate_t imm;  
    net valid;  
} instruction_t;
```

```
instruction_t icache [15:0][1:0];  
instruction_t icurrent, ifetch;
```



Syntax – Copy / Buffering

```
instruction_t icurrent, ifetch;  
always @(posedge clk)  
    icurrent <= ifetch;  
  
instruction_t icache [15:0][1:0];  
  
icache [set][way] <= icurrent;
```



Syntax – Aggregate Expressions

```
instruction_t current_inst,next_inst;  
net set,reset;  
  
always @(posedge clk)  
  if (set)  
    current_inst <= {valid:1,default:0};  
  else if (reset)  
    current_inst <= {default:0};  
  else  
    current_inst <= next_inst;
```



Syntax – Arrays/Loops/Iterators

- Local Iterators Are Convenient

```
for (int i=0; i < icurrent.imm.cnt;i++)  
    case (icurrent.imm.bytes[i])
```

- Previous Iterator Declaration Cumbersome

```
integer k;  
always_comb begin : ITSCOPE  
integer k;  
for (k=0;k<L0_PARAM;k=k+1)  
    //stop what you're doing  
    //& declare iterator k!
```



Syntax – Arrays/Loops/Iterators

- Avoids need for many unique iterators

```
int i0,i1,i2;  
...  
for (i0 = 0;i0<L1_PARAM;i0++)  
...  
for (i1 = L2_PARAM;i1>0;i1--)
```

- Avoids potential races from inadvertant iterator re-use

```
int j;  
always_comb  
    for (j=0;j<L3_PARAM;j++)  
always_comb  
    for (j=0;j<L4_PARAM;j++)
```



Syntax – Variable Assignment

- Assign or Always

```
assign ifetch.modrm = bus[7:0];  
  
always_comb begin  
    ifetch.modrm = bus[7:0];  
end
```



Syntax - .* Port Connections

- Creating netlists by hand is tedious
- Generated netlists are unreadable
 - Many signals in instantiations
 - Instantiations cumbersome to manage
- Implicit port connections dramatically improve readability
- Use same signal names up and down hierarchy where possible
- Port Renaming Accentuated

```
module top();  
    logic rd,wr;  
    tri [31:0] dbus,abus;  
    tb(.*);  
    dut(.*);  
endmodule
```

```
module top();  
    logic rd,wr;  
    tri [31:0] dbus,abus;  
    tb tb(.*, .ireset(start),  
        .oreset(tbreset));  
    dut d1(.*,.reset(tbreset[0]));  
    dut d2(.*,.reset(tbreset[1]));  
endmodule
```



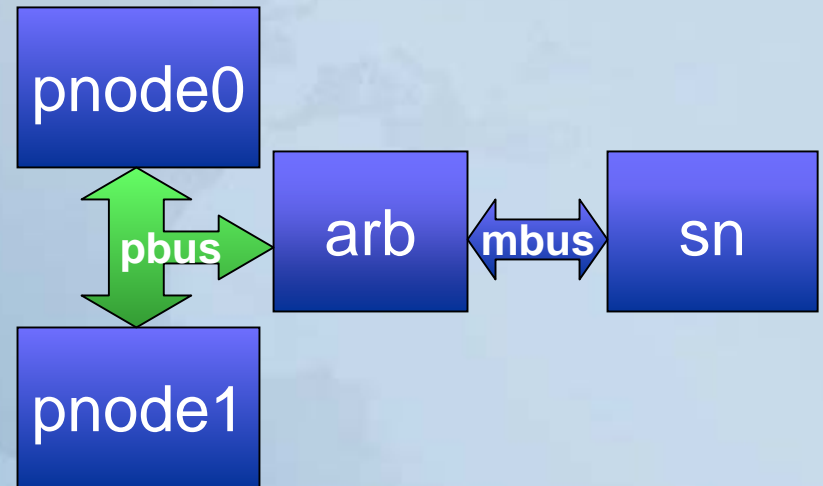
Communication Encapsulation

- Some signals of a communication protocol are bi-directional
- Structs cannot be bi-directional
 - Structs comprised of variables, ***No Net Types***
 - All struct members must have same direction
- Interfaces
 - Facilitate bundling of signals, ***including Net Types***
 - Simplify module port lists
 - Simplify netlist specification
 - Facilitate iterative development



Communication Encapsulation

```
interface peripheralbus
  (input logic ck);
endinterface
interface memorybus
  (input  logic ck);
endinterface
module system();
  logic rst, ck;
  oscillator osc(rst, ck);
  memorybus      mbus(ck);
  peripheralbus  pbus(ck);
  pnode1  pn1(pbus);
  pnode0  pn0(pbus);
  storage sn(mbus);
  arbiter arb(pbus, mbus);
endmodule
```



- High-Level Connectivity Fairly Stable



Communication Encapsulation

```
interface peripheralbus
```

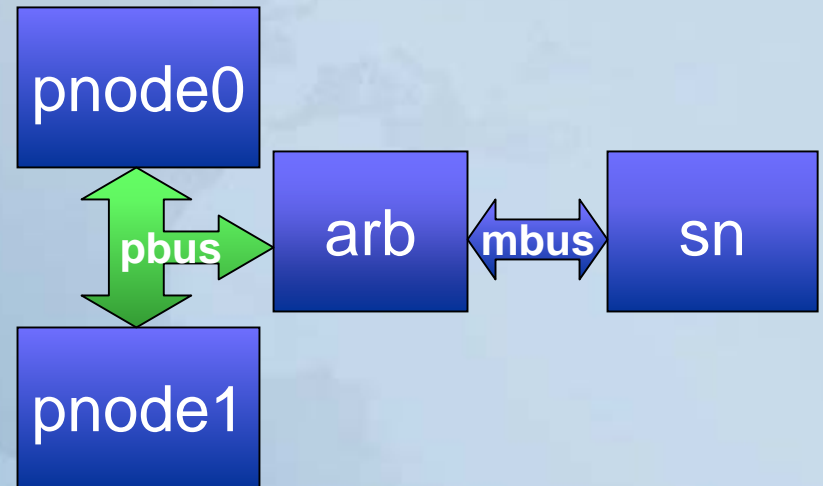
```
  (input logic ck);  
  wire [7:0] tbus;  
  logic      rst;  
  logic      tx;  
  logic      rdy;  
  logic      da;
```

```
endinterface
```

```
interface memorybus
```

```
  (input logic ck);  
  
  wire [31:0] data;  
  wire [31:0] adr;  
  logic      rden;  
  logic      wren;
```

```
endinterface
```

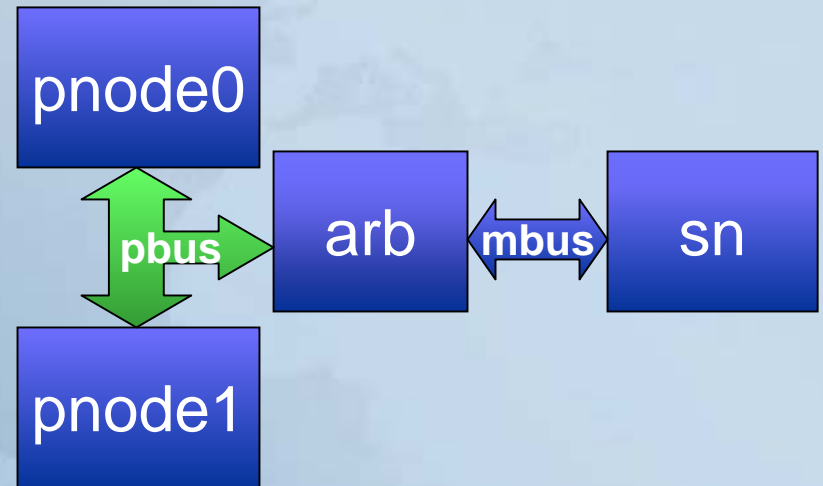


- Add/Remove Signals
- Changes Propagate to Module Ports



Communication Encapsulation

```
interface peripheralbus
  (input logic ck);
  wire [7:0] tbus;
  logic      rst;
  logic      tx;
  logic      rdy;
  logic      da;
  modport arb(input ck, output
rst,rdy,da,inout tbus);
  modport node(input rdy,rst,
da,output tx,ck,inout tbus);
endinterface
...
pnode1  pn1(pbus.node);
pnode0  pn0(pbus.node);
arbiter arb(pbus.arb,mbus);
```



- Define Access Rules
- Apply Rules To Ports



Summary

- SystemVerilog Effective Platform for Design
- Enhanced Data Organization
 - Powerful Data Types Better Map to Algorithms
 - Relationships Preserved, Naming More Descriptive
- Design Intent Preserved
 - Consistency across tools
 - Simple checks save time
- Syntax enables powerful, efficient expression
- Communication Encapsulation extends data organization to module ports

