Tampereen yliopisto
Tampere University

SystemVerilog Introduction

This slide set is based on IEEE 1800-2012 © standard of SV

Objective

- Learn basic concepts of SystemVerilog in verification point of view
  - Most of the SystemVerilog concepts are for verification

- Lots of slides and information, not low-level details
  - To become professional with SV, one must read the standard, explore and test, and read the standard more

- Not all tools (compilers, simulators) are supporting all the aspects
  - Tool manuals need to be read too

Things left out

- Detailed information about how the language is processed in the simulator, eg. scheduling
  - this might be vital information when you really dig into SV, as there are all kinds of possibilities provided

- Most of the SystemVerilog standard:
  - Use this slide set only as quick start guide, refer to the standard for usage

SystemVerilog

- SystemVerilog is extended and improved Verilog to meet better requirements in HW verification
  - Originally an extension to Verilog, but merged to one language
- Originally developed by Accellera, then later standardized by IEEE
  - Currently IEEE standard 1800-2017
- Extends Verilog-2005
  - SV is Verilog –compatible (as C++ is C compatible), but SV allows RTL descriptions that would not work on Verilog compilers

SystemVerilog

- Originally an object-oriented programming extension to Verilog
  - The ”system” part
  - Purposefully designed to improve verification tasks
  - ”de-facto” assertion language

- OOP allows building testbenches and other useful tools to help designing, implementing, testing, and verifying
  - OOP brings abstraction and reuse

About this lecture

- Before diving into the object-oriented properties, this lecture explains the basic concepts in SystemVerilog

- Most concepts are common to design and verification

- Large number of slides, half of them covered on the lecture
  - All slides shared for self-study

Tampereen yliopisto
Tampere University

SystemVerilog Introduction

General Conventions

This slide set is based on IEEE 1800-2012 © standard of SV

Comments etc.

- A comment line starts with //
- A comment block starts with /* and ends with */

- something starting with **$** is a system task/function
  - in addition to built in, you can create your own!

- Compiler directives start with ` like **`define** (in Finnish qwerty, **shift+´)**

Number literals

```
659         // is a decimal number
'h 837FF    // is a hexadecimal number
'o7460      // is an octal number
4af         // is illegal (hexadecimal format requires 'h)
4'b1001     // is a 4-bit binary number
5'D 3       // is a 5-bit decimal number (bases are not case sensitive)

3'b01x      // is a 3-bit number with the least significant bit unknown
12'hx       // is a 12-bit unknown number
16'hz       // is a 16-bit high-impedance number

8 'd -6     // this is illegal syntax
-8 'd 6     // this defines the two's complement of 6,
            //     held in 8 bits—equivalent to -(8'd 6)
4 'shf      // this denotes the 4-bit number '1111', to
            //     be interpreted as a 2's complement number,
            //     or '-1'. This is equivalent to -4'h 1
-4 'sd15    // this is equivalent to -(-4'd 1), or '0001'
16'sd?      // the same as 16'sbz
```

Strings

- between double quotes ””
- Strings can be assigned to integral types, like arrays
- can be casted

```
byte mystring[0:12] = "Hello world\n";
```

Tampereen yliopisto
Tampere University

SystemVerilog Introduction

General Building Blocks and Structure

This slide set is based on IEEE 1800-2012 © standard of SV

Design elements

- Primary building blocks and containers for declarations and code

- `**module**, program, **interface**, checker, **package**, primitive, config`

- We will only cover module for now
  - interface and package in detail later on this lecture

Module

- Module is the basic block, contains (verification / RTL) code and interconnections between verification and design blocks
  - It is the place in which you would describe your RTL

```
module <name> (<ports with directions>);
    // content blocks and declarations
endmodule
```

Hierarchy

- Hierarchical design can be implemented by instatiating modules inside each other

```
<name> <inst_name> (
   <port>(<connect_to>),
   <port>(<connect_to>)
);
```

Hierarchy example

```
module top; // no ports in this module
    // signal declarations
    logic a,b,sel;   // logic is the variable datatype
    wire y;          // wire declares a net

    // component instantiation
    mux mymux(.a(a), .b(b), .sel(sel), .y(y);

endmodule

// component declaration: input and output ports
module mux(input wire a,b,sel, output logic y);
    y = sel ? a : b;
endmodule
```

Starting a DFF module

```
module dff (
  input clk,    // Clock input
  input rst_n,  // Reset input
  input d,      // D input
  output q      // Q output
);

endmodule
```

What would we need to define in this?

Tampereen yliopisto
Tampere University

SystemVerilog Introduction

Data Types

Data types and data objects

- Data type defines a type of data value that can be manipulated with operations given for that data type
  - **`logic`**, **`wire`**, `bit`, `byte`, `integer`, `time...`

- Data object is a named entity/item of a data value with given data type

Variables

- Basic data variable
- Name: **`logic`**
  - (old) verilog syntax: **`reg`**
  - ➢ Recognize **`reg`**, use **`logic`** in your own code
- 4-state type
- Cannot have multiple drivers, no drive strength information

•0 – logic zero or false condition
•1 – logic 1 or true condition
•x – unknown logic value
•z – high impedance state

- Also (4-state) integer, (2-state) bit, byte, int …, (float) real and more

Nets

- Don’t store data, but only represent connections
- Name: **`wire`**
  - 4-state type
- also uwire, tri, trireg, there’s more, but wire is the most common

DFF with data types

```
module dff (
  input  wire  clk,   // Clock input
  input  wire  rst_n, // Reset input
  input  wire  d,     // D input
  output logic q      // Q output
);

endmodule
```

Could we make it wider than 1-bit?

Packed arrays

- Usual data types are by default treated as 1-bit wide scalars
- packed arrays can be used to define multibit vectors

```
wire [15:0] busa; // a 16-bit bus

// a 4-bit packed array made up of (from most to
// least significant) v[3], v[2], v[1], and v[0]
logic [3:0] v;

// a 4-bit packed array in range -8 to 7
logic signed [3:0] signed_reg;

logic [-1:4] b; // a 6-bit packed array

// declares three 5-bit variables
logic [4:0] x, y, z;
```

Arrays

- packed array, when dimensions are before the identifying name
  - also referred to as vector
  - guaranteed continuous stream of bits in memory
  - can be made only of single bit data types!
  - can have unsigned/signed, eg. 48bit arithmetics are possible

- unpacked array, when dimensions are after the identifying name
  - can be made of any data type

- multidimensional arrays are supported

Arrays

```
bit [31:0] array; // Packed
int addr [32];    // Unpacked
// or
int addr [31:0];

byte b = array[15:8];
array[31:0] = addr[5];


// 10 elements of 4 8-bit bytes
// (each element packed into 32 bits)
bit [3:0] [7:0] joe [1:10];
```

Constants and parameters

- **constants** are named data objects that do not change over the elaboration time or run-time
  - elaboration time constants: **parameter, localparam**, specparam
  - run-time constants: **const**
- **parameter** can be overridden in instantion of **module**, **interface** or **program**
  - For example, you can create a bus interface that by default is 8 bits wide
  - Then, you can instantiate 16 bits wide bus by overriding the width parameter
- Parameters can depend on other parameters!
- Parameters can have type and range!

Parameter example

```
interface my_bus #(parameter width = 8);
    logic [width-1:0] bus;
endinterface: my_bus

module top;
    localparam BUS_WIDTH = 16;
    // assignment by name
    my_bus #((.width(BUS_WIDTH))) bus16

    // or assignment by order:
    my_bus #((BUS_WIDTH)) bus16_by_order
endmodule
```

DFF with parameterized arrays

```
module dff #(parameter data_width_g = 8)
(
  input  wire clk,                      // Clock input
  input  wire rst_n,                    // Reset input
  input  wire  [data_width_g-1:0] d, // D input
  output logic [data_width_g-1:0] q  // Q output
);


endmodule
```

How to define functionality?

Data types

FURTHER READING

Tampereen yliopisto
Tampere University

What’s the deal with those wire’s and reg’s in Verilog?

https://blogs.sw.siemens.com/verificationhorizons/2013/05/03/wire-vs-reg/

Integers

Name	Type
shortint	2-state, 16 bit signed
int	-”- 32bit signed
longint	-”- 64bit signed
byte	2 state, 8-bit signed
bit	2 state, user defined
logic	4 state, user defined
reg	4 state, user defined
integer	4 state, 32bit signed
time	4 state, 64 bit unsigned

- User defined default to **unsigned**, can be set **signed**
- Signed ones can be set **unsigned**

Other types

- data type **void** can be used to present ”no value” e.g. in function returns

- **string** is a data type (do not confuse to string literals) that has traditional C++ like string features
  - one character is type of **byte**
  - no truncation like in assigning string literal to a vector of bytes
  - built in manipulation functions

Other types

- **event** type is for communication and synchronization between concurrent processes

- user can define own types with **typedef**

- **enum** allows to enumerate data type content
  - anonymous enums are allowed too
  - be careful with variable types, remember that 2 state cannot have x/z states

```
// state is the name of variable
enum bit [1:0] {IDLE=2’b00, S0=2’b01,
S1=2’b10, S2=2’b11} state;
```

Scope and lifetime

- Variables outside desing elements have scope of the file (compilation unit) and lifetime of whole simulation (static)
- Variables inside module, interface, program or checker are local to that and have static lifetime
- Variables inside task, function or block are local and static by default
  - with **automatic** the variable has lifetime of a call

Casting

- Data types can be casted with ’
- There are limitations what can be casted to what
  - not as elaborate as C for example
- Sign can be casted with **signed’()** and **unsigned’()**
- **$cast** built in function allows dynamic casting

```
type’(expression)

int a = int’(2.0*5.0);
```

Aggregation

- **struct** can be used to collect data types under one name
  - by default structures are unpacked
    - read: can contain any data types

- **struct packed** allow accesing a bit vector with split names
  - all the data types are in following memory addresses

- **union** packs different datatypes over each other
  - one can access same place or subset of it through different names
  - can have mismatched size

- **union packed**
  - cannot have mismatched member sizes

- **union tagged** is type checked union

Aggregation

```
typdef struct {
    bit [7:0] code;
    bit [31:0] addr;
} instr_t;
instr_t instr;
instr.code = 0;

typdef union {
    bit [31:0] code;
    bit [31:0] addr;
} ex_u;
```

```
struct packed signed{
    int a;
    byte b;
} pack;

// byte b (1st goes MSB)
pack[7:0]
```

Tampereen yliopisto
Tampere University

Dynamic arrays

- Can contain any data type
  - created with []
  - construct with **new[]**, get size with **size()**, **delete()** to clear

```
// Dynamic array of 4-bit vectors
bit [3:0] nibble[];

// Fixed-size unpacked array composed
// of 2 dynamic subarrays of integers
integer mem[2][];

// arr2 sized to length 4; dynamic subarrays
// remain unsized and uninitialized
int arr2 [][] = new [4]; arr2.size;
```

Array manipulation

- lots of methods and ways in addition to basic slicing and assignment
- e.g. find(), shuffle(), sort()
- sum(), and(), or(), xor()

Queue

- Dynamic collection of homogenous elements, kept in order
  - constant time acces and insert/delete first/last
- access first element with 0 and last element with **$**
- same manipulation as for arrays
- create as unpacked array but use **$** as the size

```
bit myqueue[$];
integer Q[$:8] = {1,2,3}; // 8 max size
```

Queue methods

- size(), insert(idx, element), delete(idx),
- element pop_front(), element pop_back(),
- push_front(element), push_back(element)

Tampereen yliopisto
Tampere University

SystemVerilog Introduction

Assignments, Operators, Expressions

ASSIGNMENTS

Assignments

- Two ways to assign values

- continuous assignments
  - assign to **nets** or **variables**
  - similar to gate driving nets
  - right-side is combinational logic that drives the net continuously
- procedural assignments
  - assign only to **variables**

Continuous assignment

- Driven continuously into variables and nets
  - that is, when something changes on the right side, it is assigned immediately (if no delays included) to that variable/net

- assignment can be done in the declaration or later
  - Later: keyword **assign**

- **nets** can be continuosly assigned by multiple assignments
  - Multiple drivers
- **variables** can be continously assigned only once
  - Single driver

Continuous assignment example

```
// net declaration with continuous assignment
wire mynet = enable;

// or
wire mynet;
assign mynet = enable;
```

Procedural assignments

- Procedural assignments happen inside procedures
  - **always, initial, task, function**, covered later

- ”triggered” assignments, happen when the assignment is reached
  - Compare: continuous assignment holds through the simulation

Blocking procedural assignments

- *target* **=** *evaluated statement;*

- The assignment must go through before the execution moves to the next statement
  - Assignments processed sequentially

Non-blocking procedural assignments

- *target* **<=** *evaluated statement;*

- The assignments are scheduled at the end of the timestep
  - Assignments processed in parallel

Blocking and non-blocking

```
// blocking assignment
initial begin
    a = 1;     // a will be assigned 1 immediately
    b = #2 0;  // b = 0 at time 2, execution BLOCKED until!
    c = #10 1; // c = 1 at time 12!
end

// non-blocking assignment
initial begin
    d <= #10 1; // d will be assigned 1 at time 10
    e <= #2 0;  // e will be assigned 0 at time 2
end

// swap, at the end of time unit, a = 1 and b = 0
initial begin
    a = 0;
    b = 1;
    a <= b;
    b <= a;
end
```

Tampereen yliopisto
Tampere University

Assignment summary

- Continuous:
  - **assign** primitive
  - In declaration
- Procedural
  - Blocking: **=**
  - Non-blocking: **<=**

DFF with continuous assignment?

```
module dff #(parameter data_width_g = 8)
(
  input  wire clk,                     // Clock input
  input  wire rst_n,                   // Reset input
  input  wire  [data_width_g-1:0] d, // D input
  output logic [data_width_g-1:0] q  // Q output
);

  ~~assign q = d;~~

endmodule
```

Just a direct connection!

Not what we want.

Assignment extension and truncation

- **the left side rules**

- when right side has fewer bits, it is padded
  - unsigned -> padded according to the statement
  - signed -> padded with sign extension

- when left side has fewer bits, truncation is made
  - truncation of signed may lose sign

- **Be careful, and try to avoid!**

Extension and truncation example

```
logic [7:0]  data;
logic [31:0] addr  = 'hDEADBEEF;
logic        addr2 = 'b0;
…

data <= 0;

data <= addr;

addr <= addr2;
```

What will happen on these lines?

Everything's fine?

Compiler error?

Not what we expected?

Tampereen yliopisto
Tampere University

Assignments

FURTHER READING

Tampereen yliopisto
Tampere University

Procedural continous assignment

- with keywords **assign** and **force** one can continuosly drive expression result to a variable (**assign** for variables, **force** for nets)

- **deassign** will end the assignment to a variable, variable holds its current value

- **release** will end **force** assignment

Assignment patterns

- SV provides way to assign multiple values or patterns into variables, structures and arrays
- pattern syntax is **’{ patterns };**

```
typedef byte U[3];
var U A = '{1, 2, 3};

// same as '{y, y}
unpackedbits = '{2 {y}} ;

// same as '{'{y,y,y},'{y,y,y}}
int n[1:2][1:3] =
'{2{'{3{y}}}};
```

```
typedef struct {real r, th;} C;
var C x = '{th:PI/2.0, r:1.0};
```

Array concatenation and alias

- **concatenation** can be used to assign to multiple signals:

```
wire a,b,c;
wire [2:0] y;

assign {a,b,c} = y; // a = y[2], b = y[1], c = y[0]
```

- **alias** can be used to have multiple names for same net

```
module byte_swap (inout wire [31:0] A, inout wire [31:0] B);
    alias {A[7:0],A[15:8],A[23:16],A[31:24]} = B;
endmodule
```

OPERATORS

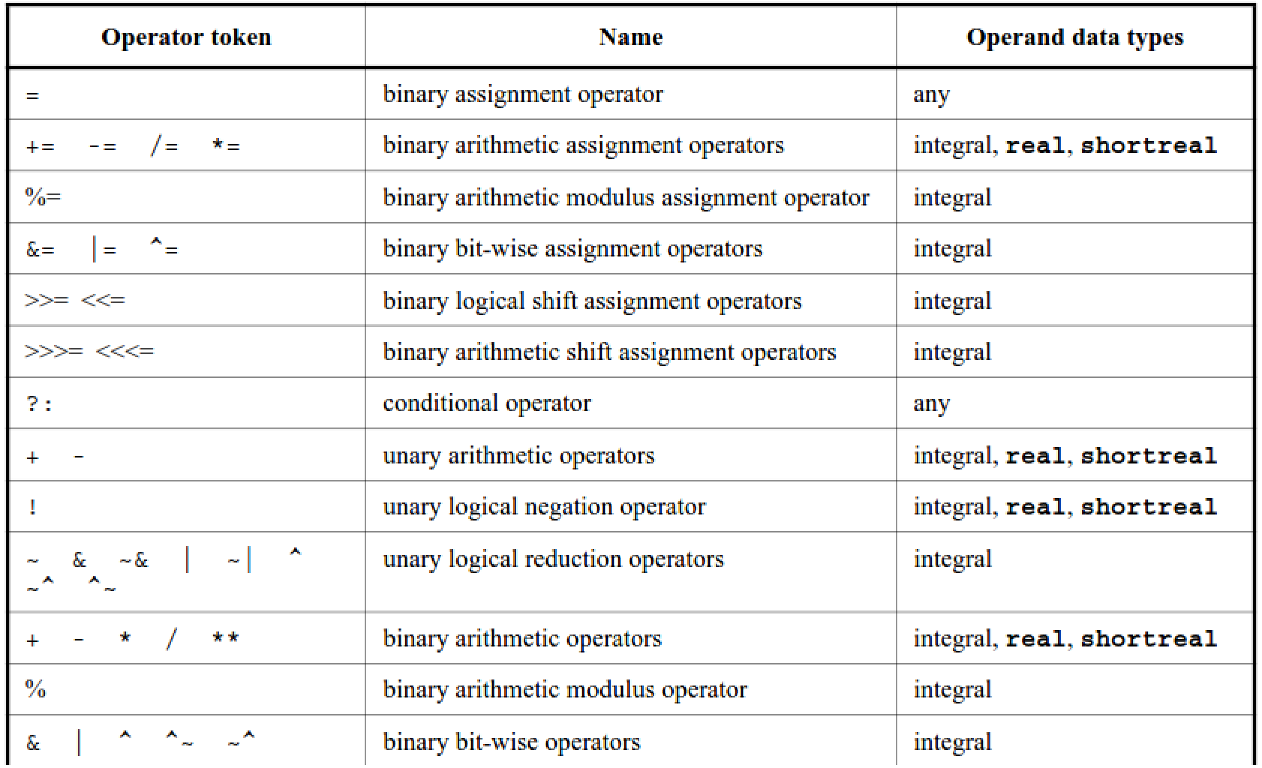
Operators in general

- Follow very closely to C/C++
- Quite self explanatory, but few exceptions

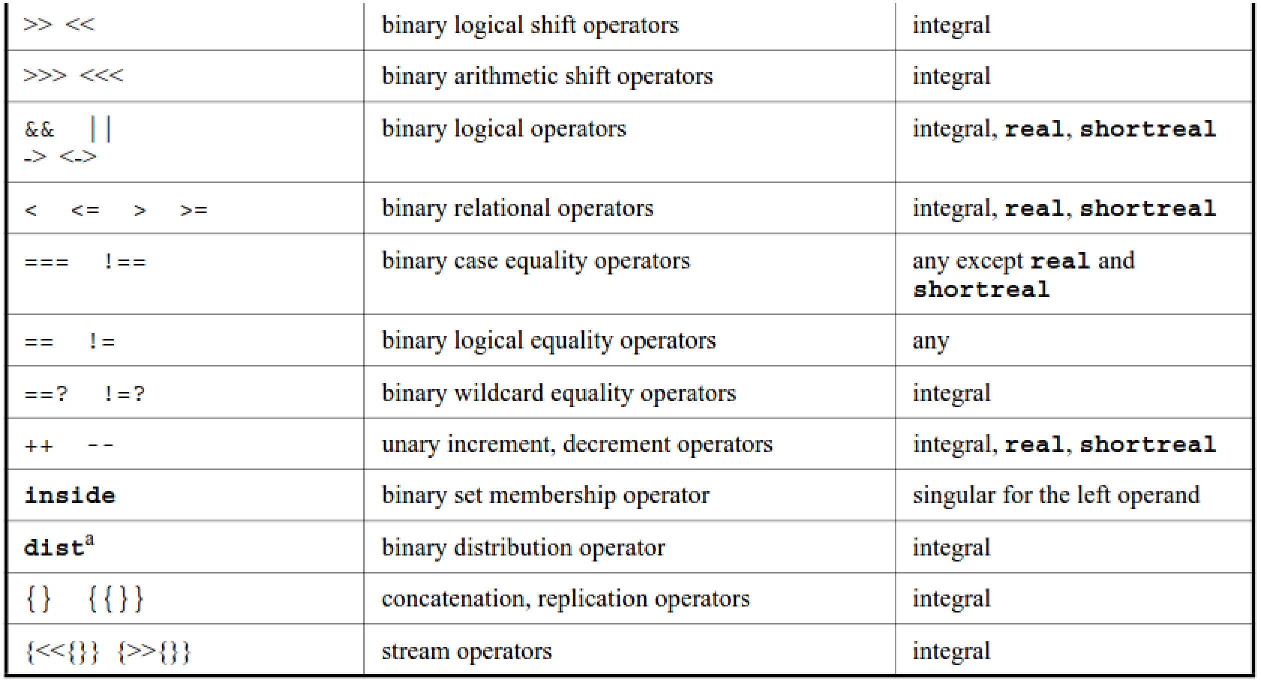
- === and !== consider x and z states too
- == and != can produce unknown (x), if x or z is involved

- ==? and !=? the ? treats right side x and z as wildcards

Table 11-1—Operators and data types



Operator token	Name	Operand data types
=	binary assignment operator	any
+= -= /= *=	binary arithmetic assignment operators	integral, **real**, **shortreal**
%=	binary arithmetic modulus assignment operator	integral
&= |= ^=	binary bit-wise assignment operators	integral
>>= <<=	binary logical shift assignment operators	integral
>>>= <<<=	binary arithmetic shift assignment operators	integral
?:	conditional operator	any
+ -	unary arithmetic operators	integral, **real**, **shortreal**
!	unary logical negation operator	integral, **real**, **shortreal**
~ & ~& | ~| ^ ~^ ^~	unary logical reduction operators	integral
+ - * / **	binary arithmetic operators	integral, **real**, **shortreal**
%	binary arithmetic modulus operator	integral
& | ^ ^~ ~^	binary bit-wise operators	integral



`>> <<`	binary logical shift operators	integral
`>>> <<<`	binary arithmetic shift operators	integral

`&& ||`
`-> <->`
	binary logical operators	integral, **`real`**, **`shortreal`**

`< <= > >=`	binary relational operators	integral, **`real`**, **`shortreal`**
`=== !==`	binary case equality operators	any except **`real`** and **`shortreal`**
`== !=`	binary logical equality operators	any
`==? !=?`	binary wildcard equality operators	integral
`++ --`	unary increment, decrement operators	integral, **`real`**, **`shortreal`**
`inside`	binary set membership operator	singular for the left operand
`dist`[a]	binary distribution operator	integral
`{} {{}}`	concatenation, replication operators	integral
`{<<{}} {>>{}}`	stream operators	integral

Operators

FURTHER READING

Tampereen yliopisto
Tampere University

Operators in general

- shifting with >> and << is logical or zero padded
- shifting with >>> and <<< is arithmetic, and padding is made according to unsigned/signed

- **inside** tells if left side operand can be found from right side list

Predecense

- From left to right
- for operators ?: and -> and <-> the association is right to left

- When operator has higher predences, it is evaluated first in left to right order

- Some operators are short circuit evaluated
  - if the result can be determined without evaluating all operators, they may be skipped

Tampereen yliopisto
Tampere University

SystemVerilog Introduction

Processes:

procedures, blocks, timing control

PROCEDURES

Procedures

- As hardware is parallel, SV must have methods for creating parallel procedures
  - **initial**
  - **always**, always_comb, always_latch, always_ff
  - final

Initial, always, final

- Initial is executed once
  - at the start of simulation

- always is executed always, until the simulation ends

- final is executed once
  - at the very end of the simulation

Initial

- **initial** procedure is executed only once
- It is typically used for initialization tasks
- Or to provide the initial stimulus to the simulated part

- **Example:** does not do much yet

```
initial inputs = 'b000000; // initialize inputs at time 0
```

Tampereen yliopisto
Tampere University

always*

- always procedures are repeated until the simulator ends
  - always, always_comb, always_lacth, always_ff
- Requires some kind of timing control
  - most of the time, you'll use always with an event/sensitivity list

always

- general purpose (parallel) procedure
- use for repetitive behaviour, e.g. generate clock

- with timing/event control can be used for creating combinational, latched, and sequential HW
  - without timing/event control, will deadlock the simulation

```
always #10 clk = ~clk; // toggle clk every 10 time units
always a = ~a; // deadlock! consumes all the simulation time
```

always_comb

- Especially made for modeling combinational logic
  - Executed according to **inferred sensitivity** list that is figured out of the code/expressions it contains

  - Linting tool (etc.) should be used to ensure correct use

```
always_comb a = b & c;
```

always_latch

- Similar to always_comb, but for latched logic

always_ff

- similar to always_latch and always_comb, but for synthesizable sequential logic

- can contain only one event control, and no blocking timing controls

DFF with always procedure

```
module dff #(parameter data_width_g = 8)
(
  input  wire clk,                      // Clock input
  input  wire rst_n,                    // Reset input
  input  wire  [data_width_g-1:0] d,  // D input
  output logic [data_width_g-1:0] q   // Q output
);
  ~~assign q = d;~~
  ~~always_ff q = d;~~

endmodule
```

Is this one line all we need inside always procedure?

Deadlock! *How to prevent it?*

Tampereen yliopisto
Tampere University

Procedures

FURTHER READING

Tampereen yliopisto
Tampere University

final

- final is executed once at the simulation end time

- it should execute without any delay

- if multiple final procedures exist, they are executed in arbitrary order
  - simulation tools should however keep this execution in order over different runs!

- calling $finish will trigger the end time for simulation and the call of the final procedures

BLOCKS

Tampereen yliopisto
Tampere University

begin–end

- sequential block
  - procedural statements inside such are executed sequentially

- can contain event control (covered later)
- any delay values should be relative to each other

- Like the { } in C

Initial procedure with begin-end

```
initial begin
    a = 0;
    for (int index = 0; index < size; index++)
     // initialize memory word
     memory[index] = 0;
end

initial begin
    // initialize at time zero
    inputs = 'b000000;
    // first pattern
    #10 inputs = 'b011001;
    // second pattern
    #10 inputs = 'b011011;
    #10 inputs = 'b011000;
    #10 inputs = 'b001000;
end
```

fork-join

- parallel block
  - ➢ procedural statements inside such are executed concurrently

- execution jumps away from the block when all statements are executed
  - a **join, join_any, join_none** ending the block control this
  - join = continue when all spawned processes are completed
  - join_any = continue when one spawned process completes
  - join_none = continue without waiting any spawned processes to complete

fork clock example

```
initial begin
    clock1 <= 0;
    clock2 <= 0;
    fork
        forever #10 clock1 = ~clock1;
        #5 forever #10 clock2 = ~clock2;
    join_none
end
```

DFF with begin-end block

```
module dff #(parameter data_width_g = 8)
(
  input  wire clk,                    // Clock input
  input  wire rst_n,                  // Reset input
  input  wire  [data_width_g-1:0] d, // D input
  output logic [data_width_g-1:0] q  // Q output
);
~~  always_ff begin~~
~~      q = d;~~
~~  end~~

endmodule
```

Still does not work!

But possible to do multiple things sequentally in always

TIMING CONTROL

Procedural timing control

- Two types, delay control and event expression

- A delay control is done with #
- An event control is done with @
- A **wait** statement combines event in while loop

delay control

- Useful for example generate desired waveforms or to separate stimulus from each other

```
always #10 clk = ~clk; // toggle clock every 10 time units

logic [7:0] r; // r declared as an 8-bit variable

begin
    // a waveform controlled by sequential delays
    #2 r = 'h35; // Note: Blocking assignment!
    #2 r = 'hE2;
    #2 r = 'h00;
    #2 r = 'hF7;
end
```

event control @

- synchronization to a value change in net or a variable, or a occurence of declared event

- can detect direction of the change
  - **posedge** the value goes towards 1
    - 0-> x,z,1 or x,z->1
  - **negedge** the value goes towards 0
    - 1->x,z,0 or x,z->0
  - **edge** the values goes towards 0 or 1

event control @ examples

```
// controlled by any value change in the reg r
@r rega = regb;

// controlled by posedge on clock
@(posedge clock) rega = regb;

// always controlled by negedge on clock
always @(negedge clock) rega = regb;

// always controlled by edge on clock
always @(edge clock) rega = regb;
```

DFF with event control

```
module dff #(parameter data_width_g = 8)
(
  input  wire clk,                     // Clock input
  input  wire rst_n,                   // Reset input
  input  wire  [data_width_g-1:0] d,  // D input
  output logic [data_width_g-1:0] q   // Q output
);

  always_ff @(posedge clk or negedge rst_n)
  begin
    q <= d;
  end

endmodule
```

But how to handle reset?

Tampereen yliopisto
Tampere University

Timing control

FURTHER READING

events in fork join

```
real AOR[]; // dynamic array of reals
byte stream[$]; // queue of bytes
// waits for array to be allocated
initial wait(AOR.size() > 0) ....;
// waits for total number of bits
// in stream greater than 60
initial wait($bits(stream) > 60)...;

Packet p = new; // Packet 1 -- Packet is defined in 8.2
Packet q = new; // Packet 2
initial fork
     @(p.status); // Wait for status in Packet 1 to change
     @p; // Wait for a change to handle p
     # 10 p = q; // triggers @p.
     // @(p.status) now waits for status in Packet 2 to change,
     // if not already different from Packet 1
join
```

Tampereen yliopisto
Tampere University

Event or -operator

- multiple events can be waited with **or**
- also comma (,) can be used for the very same behaviour

```
always @(a,b,c);

// is same as

always @(a or b or c);
```

@*

- with @* all the variables will be in the event list
  - e.g. one of the variable changes -> event fires
- if the need is as sensitivity list in the beginning of always block for combinational logic, use always_comb instead

```
// equivalent to @(a or b or c or d or tmp1 or tmp2)
always @* begin
     tmp1 = a & b;
     tmp2 = c & d;
     y = tmp1 | tmp2;
end
```

conditional events - iff

- if the expression after **iff** is true, the event can happen
- the expression is evaluated when the event happens!
- **iff** has precedence over **or**… or use parentheses

```
module latch (output logic [31:0] y, input [31:0] a, input
enable);
    always @(a iff enable == 1)
    y <= a; //latch is in transparent mode
endmodule
```

sequence events ##1

- a sequence is waited until the event is considered occured
  - see **sequence** syntax in 16.7

```
sequence abc;
    @(posedge clk) a ##1 b ##1 c;
endsequence

program test;
    initial begin
        @ abc $display( "Saw a-b-c" );
        L1 : ...
    end
endprogram
```

Tampereen yliopisto
Tampere University

wait or level sensitive

- **wait** waits until the statement experssion evaluates and executed following statement with no delay
- it implements level sensitive event control
  - that is, if the statement evaluates true when reached, it will pass, not wait for next edge

```
begin
    wait (!enable) #10 a = b;
    #10 c = d;
end
```

level sensitive sequence control

- sequences have a built in **triggered** method that returns true if the sequence has happened
- this can be used with **wait** statement to level sensitive control

```
sequence abc;
    @(posedge clk) a ##1 b ##1 c;
endsequence
...
wait( abc.triggered );
```

repeat

- with **repeat** one can wait that the event occurs given times until proceeding

```
// will execute event_expression three times
repeat (3) @ (event_expression)
```

Tampereen yliopisto
Tampere University

intra-assignment

- one can delay assignment with so called intra-assignment delay and event control
- insert delay/event control between the assignment
- useful to fix race conditions in fork-join, assign both variables, add delay, delay ensures that both will be assigned correctly

```
// same as temp = b; #5 a = temp;
a = #5 b;

// wait 5 clk cycles until the data is
assgined to a, notice that data can change
during this, but the old value is assigned
a <= repeat(5) @(posedge clk) = data;
```

```
// race free
fork
    a = #5 b;
    b = #5 a;
join
```

Process and process control

- Just for curiosity, see 9.6 for precise info
- **wait fork** blocks until the child subprocesses have completed
- **disable** stops a task before it reaches end, it can also terminate a named block
  - can be used to get goto –like structure
- built in class **process** is given, has own class methods, cannot be extended, cannot be constructed (instead use process::self() method)
  - a bit unclear how this is actually beneficial..

Semaphores

- Built in class for key bucket synchronization
- Process reserves key(s), process cannot continue until it gets key(s)
- The initial key amount is given in constructor **new**(int keyCount = 0), default is 0
- processes can put keys in the bucket with **put**(keyCount=1)
- processes can take keys with **get**(keyCount=1), if not enough keys available, the call will block until enough keys are returned
- process can non-blockingly try to get keys with **int try_get**(keyCount=1), return value is 0 if no keys were available

mailboxes

- built in class for exchanging messages between processes
- receiver can check mailbox for new messages
  - wait if there is no mail
  - or proceed
- The mailbox can be bound or unbounded
  - bound has a limit, sender may be blocked until receiver empties box
  - unbound has unlimited room
- By default, accepts any type as message and one mailbox can contain several types
  - a mailbox can be parametrized to accept only one type

mailboxes

- **mailbox** or parametrized **mailbox #(type)**
- constructor **new**(bound=0)
- send message **put(**msg**)** (blocking) or **int try_put(**msg**)** (non-blocking)
- receiver message **get(ref** msg**)** or **peek(ref** msg**)**
  - the difference is that peek() does not remove the message from the mailbox
- try to receive without blocking **try_get(ref** msg**), try_peek(ref** msg**)**
- get number of messages with **int num()**

Named event

```
event e;
-> e; // trigger e
wait(e.triggered);
@ e;
```

- One can create named events that can be wait for with **wait()** or @
- **event** name;
- Event has **name.triggered** that tells if the event has occured
  - can be used with **wait(**name.triggered)
- Event can be dispatched with operator **->**
- Events can be wait in order with **wait_order(**event1, event2,..**)**
  - events must occur in the given order, othewise a runtime error is produced
- Events can be compared

CONTROL STRUCTURES

Conditional if – else

- as in many languages, else part is optional
- no need for separating the body, but **begin-end** block may be used (and recommended)

```
if ( x > 0 )
      y = 0;
else
      y = 1;
```

```
if ( x > 0 )
      begin
            y = 0;
            z = y;
      end
else
      y = 1;
```

```
if ( x > 0 )
      y = 0;
else if ( x < 0 )
      y = 1;
```

case statement

- similar to C/C++
- can handle x and z

```
logic [15:0] data;

case (data)
    16'd0: x = 0;
    16'd1: x = 1;
    default x = 0;
endcase
```

```
logic [1:0] data;

case (data)
    2'b0x,2'b00: x =
0;
    2'b10: x = 1;
    default x = 0;
endcase
```

loops

- six different ways to loop
- **forever, repeat(),** while(), for(;;), do while(), foreach()
- quite similar to other languages

forever

- **forever** loops forever
- good for generating clocks
- **remember to avoid zero delay hang**

```
initial begin
    clock1 <= 0;
    clock2 <= 0;
    fork
        forever #10 clock1 = ~clock1;
        #5 forever #10 clock2 = ~clock2;
    join
end
```

repeat

- repeats the body as many times as given parameter

```
// prints three steps
repeat (3) begin
    $display(”step ”);
end
```

Tampereen yliopisto
Tampere University

DFF with control structures

```
module dff #(parameter data_width_g = 8)
(
  input  wire clk,                     // Clock input
  input  wire rst_n,                   // Reset input
  input  wire  [data_width_g-1:0] d, // D input
  output logic [data_width_g-1:0] q  // Q output
);
  always_ff @(posedge clk or negedge rst_n)
  begin
    if (~ rst_n) begin
      q <= 0;
    end
    else begin
      q <= d;
    end
  end

endmodule
```

Ready!

How many of these begins and end do we absolutely need?

Control structures

FURTHER READING

Tampereen yliopisto
Tampere University

unique-if

- **unique if**
  - make a violation report, if there is no matching condition within the if-block
  - Ensures that there is no overlap in if-else-if’s (so it can be done parallel)
- **unique0 if**
  - reverse of **unique**, no violation if no matches, but ensures that there is no overlap
- **priority**
  - same as **unique** but allows multiple matches for one variable while requires that the given if-else-if conditions are evaluated in order

```
unique if ((a==0) || (a==1)) $display("0 or 1");
else if (a == 2) $display("2");
else if (a == 4) $display("4"); // values 3,5,6,7 cause a
violation report
```

case with do not cares

- **casez** handles z’s as do not cares
  - use ? in place of z’s in case statements
- **casex** handles x’s as do not cares
  - use x in place of x’s in case statements

```
logic [7:0] ir;

casez (ir)
    8'b1???????: instruction1(ir);
    8'b01??????: instruction2(ir);
    8'b0001????: instruction3(ir);
    8'b000001??: instruction4(ir);
endcase
```

case statement with unique

- the same **unique, unique0** and **priority** can be used for case as for if
  - that is to check that at least one of the cases will be covered
  - **unique** and **unique0** ensure that no overlap and safe to execute in parallel
  - **priority** case should match only the first match

for loop

```
for (int i = 0; i <= 255; i++)
    ...

begin
    automatic int i;
    for (i = 0, int j = 0; i <= 255; i++)
    ...
end
```

while and do while

- both loop as long as the expression is true
- while tests the expression in beginning
- do while tests the expression at the end

```
while (tempreg) begin
    if (tempreg[0])
    count++;
    tempreg >>= 1;
end
```

```
do
    $display(“looping\n”);
    #10
while (true);
```

Tampereen yliopisto
Tampere University

foreach

- goes through iterable arrays
- can cover over multidimensional arrays too

```
string words [2] = '{ "hello",
"world" };
int prod [1:8] [1:3];

// print each index and value
foreach( words [ j ] )
    $display( j , words[j] );

foreach( prod[ k, m ] )
    prod[k][m] = 0;
```

jumps

- three kind of **break, continue, return**
- quite self explanatory
- **break** jumps out of loop, no questions asked
- **continue** jumps at the end of the loop for another round (expression is evaluated)

- **return** jumps out of function, can return a value, must be correct type

Tampereen yliopisto
Tampere University

SystemVerilog Introduction

Tasks and Functions

This slide set is based on IEEE 1800-2012 © standard of SV

Subroutines: Tasks and functions

- **Tasks** and **functions** are for repeating work
  - corresponds to functions in many languages

- Tasks **consume time**, but **cannot return value**
  - can have delays
  - you implement logic inside
  - can pass return value through arguments

- Functions **do not consume time**, but **can have return value**
  - cannot have delays
  - non-void one can be operand of expressions

Tasks

- Can have time delays etc. inside
- Can take arguments
- Control is returned when task is completed
  - If a task enables other tasks, all need to complete before returning

```
task mytask1 (output int x, input logic y);
    ...
endtask

task mytask2;
    output x;
    input y;
    int x;
    logic y;
    ...
endtask
```

Functions

- Functions should not contain anything that consumes time
  - #, ##, always
- As a task can consume time, functions should not call any tasks
- Functions can call other functions
- Functions can suspend processes

- Main use to produce values for evaluating expressions

Function return value

- Function can return a value
  - The type can be explicitly defined
  - it can be only a range and sign, but **logic** is defaulted then
- Function can be **void** without any return value

- The arguments are passed same way than to tasks

Functions

- return value can be given as **return** or using built in implicit variable with the same name as the function

```
function logic [15:0] myfunc1(int x, int y);
    ...
    myfunc1 = x*y;  // return value assigned
endfunction

function logic [15:0] myfunc2;
    input int x;
    input int y;
    ...
    return x*y; // return value through return
endfunction
```

Tasks and Functions

FURTHER READING

Tampereen yliopisto
Tampere University

Task arguments

- **input** – copy value in the beginning
  - default
- **output** – copy value out at the end
- **inout** – copy value in the beginning and out at the end
- **ref** – pass value to the task as reference

- data type maybe specified, or inherited, or default to **logic**

Task execution

- Statements inside task are executed sequentially until **endtask**
- task can **return** at any point
- task is called as it would be “function” by passing desired values as arguments
  - values should meet the declared types
  - if argument is **output** but the calling value is **ouput** as well, a compilation error should rise

Argument passing

- Pass by value
  - default, the value is copied

- Pass by reference with **ref**
  - does not copy values, only the handle
  - especially for large values, structures, arrays
  - change to the data will change it outside too
    - as no copy is made of the values
  - only variable, class property, unpacked structure member, unpacked array element are legal to pass as by reference
  - NO NETS!

Argument passing

- Arguments can have default value with **=** operator
- Arguments can be bind by the name
- if function/task takes no arguments, () are optional

```
function int fun( int j = 1, string s = "no" );
    ...
endfunction

fun( 2, ”yes”);
fun( .j(2), .s(”yes”));
fun(.s(”yes”)); // 1 gets default value
fun( , ”yes”);  // 1 gets default value
```

Tampereen yliopisto
Tampere University

SystemVerilog Introduction

Code structure and Simulation

This slide set is based on IEEE 1800-2012 © standard of SV

Packages

- **package…endpackage** provides method to capsulate namespaces
- **package** is **imported** to design instead of **`include**
  - **including** a file is simple text replace, which may cause compilation errors due to overlapping names or produce cryptic dependencies
  - **import** gives visibility to the package
- accessing package is done through **::** operator

Packages

```
package p;
      typedef enum { FALSE, TRUE } bool_t;
endpackage
package q;
      typedef enum { ORIGINAL, FALSE } teeth_t;
endpackage

module top1 ;
      import p::*;
      import q::teeth_t;

      teeth_t myteeth;

      initial begin
            myteeth = q:: FALSE; // OK
            myteeth = FALSE;     // ERROR: Direct reference to FALSE refers to the
      end                         // FALSE enumeration literal imported from p
endmodule
```

Interface

- Encapsulates communication between design blocks
  - It capsulates several wires / ports inside one item that can be passed between blocks
- Can have other elements, parameters, constants, variables, functions, and tasks
  – If two modules are connected through the interface item, the communication may be just a subroutine call of the interface

```
interface <name> (<port etc. declarations>);
endinterface
```

Interface example

```
interface i2c(input logic clock);
    logic sda, scl;
endinterface: i2c

module masterDev(i2c bus);
    always @(posedge clock)   bus.scl <= ~bus.scl;
endmodule

module slaveDev(i2c bus);
endmodule

module top;
    logic clk = 0;
    i2c i2c_intf(.clock(clk));
    // connect master&slave to same interface
    masterDev(.bus(i2c_intf));
    slaveDev(.bus(i2c_intf));
endmodule
```

modport

- **modport** declares the direction of the logic connections within an interface

```
interface spi;
    wire clk, cs, mosi, miso;
    modport master (input miso, output clk, cs, mosi);
    modport slave (output miso, input clk, cs, mosi);
endinterface

module m(spi i);
endmodule

spi spi_intf();
m(.i(spi_intf.master));
```

Simulation time and precision

- With SystemVerilog you can do functional simulation with timing incorporated
  - **timeunit** is the measurement unit, s, ms, us, ns, ps, fs
  - **timeprecision** is the degree of accuracy for delays
  - or **`timescale** <time unit> **/** <time precision>
    - Effective as ”if no timeunit/precision given” up until another `timescale is encountered
- Global time unit is the simulation time unit that is the smallest of the given time units (same for precision)
  - also referred as **step**

Compilation and Elaboration

- Compilation checks the code syntax and semantic errors
- Elaboration binds the design components together before the simulation
- Typically ”compilation” == compilation+elaboration
- Namespaces can cause trouble in compilation

Code structure and simulation

FURTHER READING

Tampereen yliopisto
Tampere University

Packages

- http://blogs.mentor.com/verificationhorizons/blog/2010/07/13/package-import-versus-include/

21. I/O System tasks

- Very similar to C/C++
- fd = $fopen(filename, type);
- $fdisplay(fd, ”text”);
- c = $fgetc(fd);
- ret = $fgets (str, fd); // read line
- logic [7:0] mem[1:256]; $readmemh("mem.data", mem); // read file of hexadecimals to array

20. utility system tasks

Simulation control tasks (20.2)

$finish	$stop
$exit	

Simulation time functions (20.3)

$realtime	$stime
$time	

Timescale tasks (20.4)

$printtimescale	$timeformat

Conversion functions (20.5)

$bitstoreal	$realtobits
$bitstoshortreal	$shortrealtobits
$itor	$rtoi
$signed	$unsigned
$cast	

Data query functions (20.6)

$bits	$isunbounded
$typename	

Array query functions (20.7)

$unpacked_dimensions	$dimensions
$left	$right
$low	$high
$increment	$size

Math functions (20.8)

$clog2	$asin
$ln	$acos
$log10	$atan
$exp	$atan2
$sqrt	$hypot
$pow	$sinh
$floor	$cosh
$ceil	$tanh
$sin	$asinh
$cos	$acosh
$tan	$atanh

Severity tasks (20.9)

$fatal	$error
$warning	$info

Elaboration tasks (20.10)

$fatal	$error
$warning	$info

Assertion control tasks (20.11)

$asserton	$assertoff
$assertkill	

Assertion action control tasks (20.12)

$assertpasson	$assertpassoff
$assertfailon	$assertfailoff
$assertnonvacuouson	
$assertvacuousoff	

Assertion functions (20.13)

$onehot	$onehot0
$isunknown	$sampled
$rose	$fell
$stable	$changed
$past	$countones
$past_gclk	$rose_gclk
$fell_gclk	$stable_gclk
$changed_gclk	$future_gclk
$rising_gclk	$falling_gclk
$steady_gclk	$changing_gclk

Coverage control functions (20.14)

$coverage_control	$coverage_get_max
$coverage_get	$coverage_merge
$coverage_save	$get_coverage
$set_coverage_db_name	$load_coverage_db

Probabilistic distribution functions (20.15)

$random	$dist_chi_square
$dist_erlang	$dist_exponential
$dist_normal	$dist_poisson
$dist_t	$dist_uniform

Stochastic analysis tasks and functions (20.16)

$q_initialize	$q_add
$q_remove	$q_full
$q_exam	

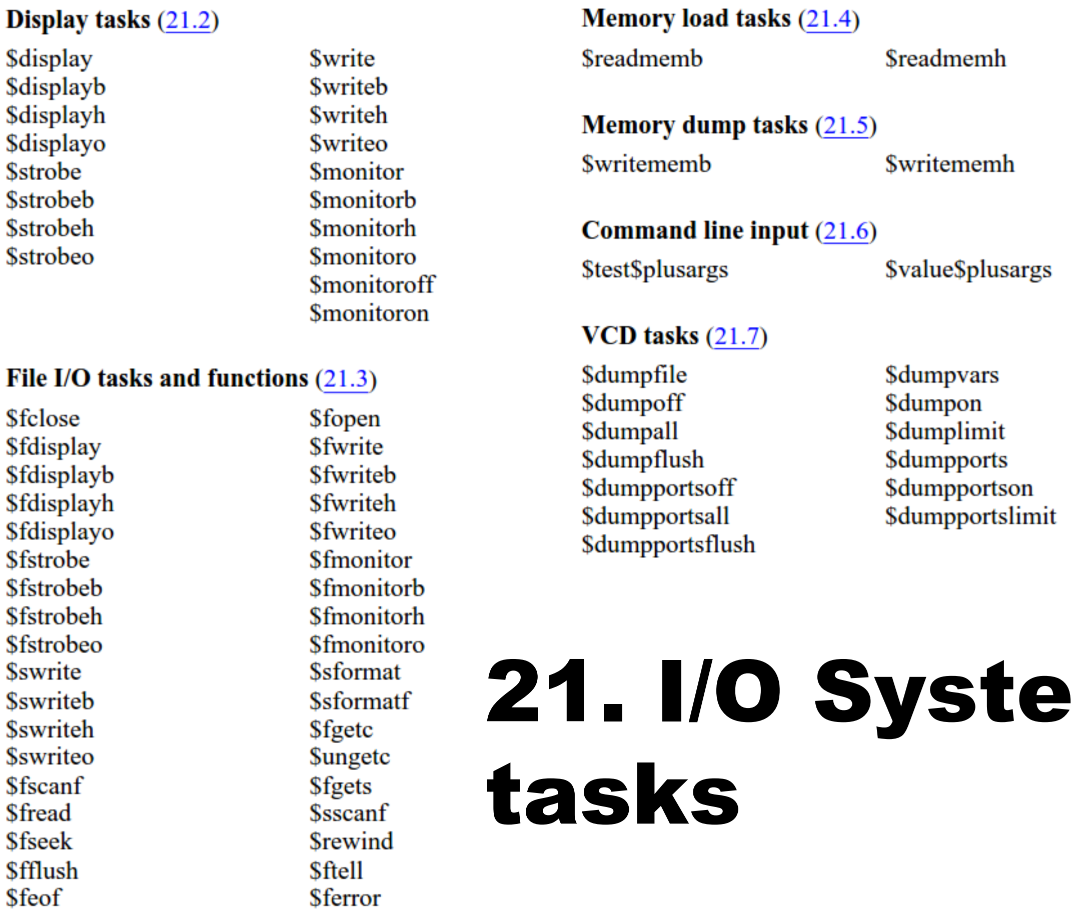
PLA modeling tasks (20.17)

$async$and$array	$async$and$plane
$async$nand$array	$async$nand$plane
$async$or$array	$async$or$plane
$async$nor$array	$async$nor$plane
$sync$and$array	$sync$and$plane
$sync$nand$array	$sync$nand$plane
$sync$or$array	$sync$or$plane
$sync$nor$array	$sync$nor$plane

Miscellaneous tasks and functions (20.18)

$system

Tampereen yliopisto
Tampere University



Display tasks (21.2)

$display	$write
$displayb	$writeb
$displayh	$writeh
$displayo	$writeo
$strobe	$monitor
$strobeb	$monitorb
$strobeh	$monitorh
$strobeo	$monitoro
	$monitoroff
	$monitoron

File I/O tasks and functions (21.3)

$fclose	$fopen
$fdisplay	$fwrite
$fdisplayb	$fwriteb
$fdisplayh	$fwriteh
$fdisplayo	$fwriteo
$fstrobe	$fmonitor
$fstrobeb	$fmonitorb
$fstrobeh	$fmonitorh
$fstrobeo	$fmonitoro
$swrite	$sformat
$swriteb	$sformatf
$swriteh	$fgetc
$swriteo	$ungetc
$fscanf	$fgets
$fread	$sscanf
$fseek	$rewind
$fflush	$ftell
$feof	$ferror

Memory load tasks (21.4)

$readmemb	$readmemh

Memory dump tasks (21.5)

$writememb	$writememh

Command line input (21.6)

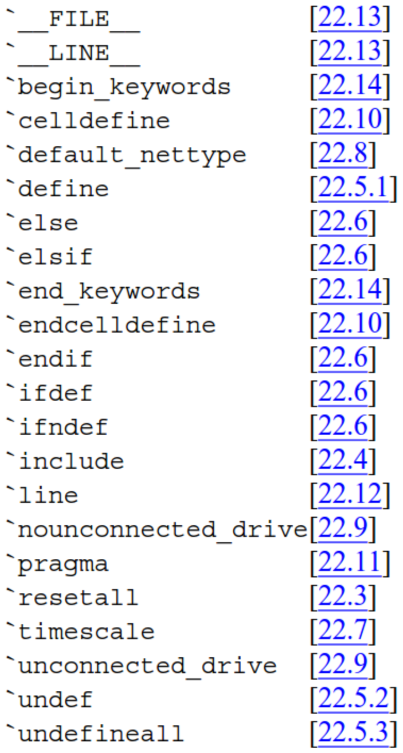
$test$plusargs	$value$plusargs

VCD tasks (21.7)

$dumpfile	$dumpvars
$dumpoff	$dumpon
$dumpall	$dumplimit
$dumpflush	$dumpports
$dumpportsoff	$dumpportson
$dumpportsall	$dumpportslimit
$dumpportsflush	

21. I/O System tasks

22. compiler directives



`__FILE__	[22.13]
`__LINE__	[22.13]
`begin_keywords	[22.14]
`celldefine	[22.10]
`default_nettype	[22.8]
`define	[22.5.1]
`else	[22.6]
`elsif	[22.6]
`end_keywords	[22.14]
`endcelldefine	[22.10]
`endif	[22.6]
`ifdef	[22.6]
`ifndef	[22.6]
`include	[22.4]
`line	[22.12]
`nounconnected_drive	[22.9]
`pragma	[22.11]
`resetall	[22.3]
`timescale	[22.7]
`unconnected_drive	[22.9]
`undef	[22.5.2]
`undefineall	[22.5.3]

- similar to C/C++
- notice timescale, default_nettype,