



Ch. 3: Data Types in Verilog-AMS HDL

INTEGER AND REAL DATA TYPES

Integer Data Type

- Declared using the `integer` keyword; 32-bit signed two's complement representation.
- Value range: to -2^{31} to $2^{31} - 1$
- Can be used in both analog and digital contexts:
 - **Analog context:** initialized to 0.
 - **Digital context:** initialized to '`x`' (unknown).
- Supports scalar, 1D, and multidimensional arrays.
- All integer operations are performed as 2's complement arithmetic.

Syntax:

```
integer a;  
integer a[1:64]; // One-dimensional array  
integer flag_array[0:8][0:3]; // Two-dimensional array
```

Real Data Type

- IEEE 754 double-precision floating-point (64-bit).
- Declared using the `real` keyword.
- Always initialized to `0.0` at simulation start.
- Can store continuous-time quantities like voltage, gain factors, etc.

Syntax:

```
real gain;  
real gain_factor[1:30]; // Array of gain factors  
real vtable[0:16][0:7][0:64]; // Multidimensional array
```

String Data Type



- `string` is a dynamic data type that can store variable-length character sequences.
- Supports runtime resizing and manipulation.
- Assignment of a literal string to a `string` variable does not truncate or pad.
- Special value `""` denotes an empty string.
- Null character `"\0"` is not permitted inside strings.

Operations

- **Concatenation:** `{"abc", "def"}` results in `"abcdef"`
- **Replication:** `{3 {"Hi"}}` results in `"HiHiHi"`
- **Comparison:** `==, !=, <, >, <=, >=` (lexical order)

Syntax:

```
string variable_name = "example"; // valid
string declaration
string empty_str; // implicitly initialized to ""
```

Array Support:

```
string names[1:3] = {"first", "middle", "last"};
string paths[0:2][0:1] = {"{"dir1", "fileA"},
{"dir2", "fileA"}, {"dir1", "fileB"}];
```

Parameters



- Can be of types: `integer`, `real`, `string`, arrays, etc.
- Parameters represent constants, hence it is illegal to modify their value at runtime.
- However, parameters can be modified at compilation time with the `defparam` statement or in the `module_instantiation` statement.
- This allows customization of module instances.

Syntax:

```
parameter real gain = 5.0;  
localparam integer width = 8;
```

Parameter Arrays

```
parameter real coeffs[0:3] = '{0.1, 0.2, 0.3, 0.4};  
parameter real c[0:2][0:2] =  
'{ '{0.0,0.1,0.1}, {0.1,0.0,0.1}, {0.1,0.1,0.0} }';
```

Value Constraints

- Range restrictions can be added using `from`, `exclude` keywords.

```
parameter real resistance = 1.0 exclude 0.0;
```

```
parameter string type = "NMOS" from '{"NMOS",  
"PMOS"}';
```

Parameters [contd.]



Parameter Aliasing

- `aliasparam` allows creating alternate names.
- Only one of the alias or original may be overridden during instantiation.

Syntax:

```
parameter real delay = 1.0;  
aliasparam trise = delay;
```

Local Parameters

- Declared using `localparam`, which cannot be overridden.
- Useful for defining internal constants that must remain unchanged.

Syntax:

```
localparam real abs_tol = 1e-6;
```

net_discipline

Definition and Role

- `net_discipline` is a specialized data type introduced in Verilog-AMS.
- Used to declare analog nets and assign discipline domains to both analog and digital signals.
- Crucial for continuous-time and mixed-signal simulation.

Signal Structure

- A signal can be:
 - **Digital**: described with `wire`, `reg`.
 - **Analog**: requires `net_discipline` and solver resolution.
 - **Mixed**: combines both, solved using unified semantics.
- Signals form **hierarchical nets**; analog nodes connect continuous segments.

Physical Interpretation

- Nodes represent physical connection points.
- Analog nodes obey **Kirchhoff's conservation laws**:
 - Potential continuity (voltage)
 - Flow conservation (current)

Nature

Purpose and Role

- A **nature** defines the type of quantity for a **potential** or **flow**.
- A nature is a collection of attributes.(Can consist of either pre-defined attributes or user-defined)
- Each discipline binds its **potential** and **flow** to a declared nature.

Required Fields in a Nature Declaration

- **units** — Measurement unit (e.g., "V" for volts).
- **access** — Function for reading/writing node value (e.g., **V**, **I**).
- **abstol** — Absolute tolerance for convergence during simulation.

Optional Fields

- **idt_nature** — Specifies nature of the integral of this quantity.
- **ddt_nature** — Specifies nature of the derivative.
- **inherit** — Base nature used when deriving new ones.

Syntax:

```
nature Voltage;  
  units = "V";  
  access = V;  
  abstol = 1u;  
endnature
```

Derived nature:

```
nature highvoltage : Voltage; //A derived  
nature uses : to inherit attributes from a base  
nature.  
  abstol = 0.01;  
endnature
```

Note:**units** and **access** **must not** be redefined.

Discipline



Definition and Function

- A **discipline** binds a **potential** and **flow** to specific **natures**, specifying how analog/mixed-signal nets behave.
- Used to declare physical domains (electrical, mechanical, etc.) and guide the simulator in applying conservation laws.

Key Elements

- **potential**: Describes across-variable (e.g., voltage, position)
- **flow**: Describes through-variable (e.g., current, force)
- **domain**: Defines solver behavior:
 - **continuous** → analog resolution
 - **discrete** → event-driven/digital logic

Syntax:

```
discipline <name>;  
  domain <continuous|discrete>;  
  potential <nature_name>;  
  flow <nature_name>;  
enddiscipline
```


Branches



Definition and Purpose

- A **branch** represents the **connection path** between two nodes.
- Enables access to potential (voltage) and flow (current) between nets.
- Essential for modeling conservation laws in conservative systems.
- If both nets are conservative, then the branch is a conservative branch(it defines a branch potential and a branch flow)
- If one net is a signal-flow net, then the branch is a signal flow branch and it defines either a branch potential or a branch flow, but not both
- Scalar and vector branching?

Syntax:

```
branch (<node1>, <node2>)  
<branch_name>;
```

Example:

```
branch (a, b) br_ab;  
analog I(br_ab) <+ V(br_ab)/R;
```

- $V(\text{br_ab})$ represents the potential between the two nodes or across the branch



Ch. 6: Hierarchical structures

Introduction to genvar & generate



What is generate?

- A construct used for conditional and loop-based instantiation of hardware components in Verilog.
- Helps in designing parameterized hardware efficiently.


What is genvar?

- A special integer variable used inside generate loops.
- Evaluated at elaboration time, not simulation time.
- Cannot be used in `always` or `initial` blocks.

Key Use Cases:

- Creating multiple instances of modules, registers, or wires.
- Designing parameterized and scalable hardware structures.
- Reducing manual code duplication.

Basic genvar Example (Generating Multiple Instances)



This example generates 8 identical instances of an inverter module.

```
module inverter_array(input logic [7:0] in, output logic [7:0] out);

    genvar i;

    generate

        for (i = 0; i < 8; i = i + 1) begin : gen_inverters

            inverter inv (.in(in[i]), .out(out[i]));

        end

    endgenerate


endmodule

module inverter(input logic in, output logic out);

    assign out = ~in;

endmodule
```

defparam Statement Rules




The `defparam` statement allows overriding parameter values using hierarchical names. Can modify any module instance's parameter throughout the design hierarchy.

Restrictions:

- A `defparam` inside a `generate` block or array of instances cannot change parameters outside its own hierarchy.
- It cannot target parameters in another instance of the same `generate` block, even if created in the same loop.
- `defparam` is not allowed inside `paramset` instances.

Each `generate` block instance is treated as a separate scope.


Basic genvar Example (Generating Multiple Instances)



This example generates 8 identical instances of an inverter module.

```
genvar i;  
  
generate  
  
for (i = 0; i < 8; i = i + 1) begin : somename  
  
flop my_flop(in[i], in1[i], out1[i]);  
  
defparam somename[i+1].my_flop.xyz = i ;  
  
end  
  
endgenerate
```

Hierarchical System Parameters – Overview



Each module implicitly declares six system parameters:

- `$mfactor`, `$xposition`, `$yposition`, `$angle`, `$hflip`, `$vflip`

These can be accessed like regular parameters within modules or paramsets

Can be overridden via:

- `defparam` statements
- Module instance parameter assignment by name
- `paramset` declarations

Override syntax: prefix with `.`, e.g., `.mfactor`

Automatic Scaling with \$mfactor



If `$mfactor` \neq 1, simulator applies automatic scaling rules:

- Branch flow contributions: multiplied by `$mfactor`
- Branch flow probes (including indirect assignments): divided by `$mfactor`
- Noise contributions (flow): noise power \times `$mfactor`
- Noise contributions (potential): noise power \div `$mfactor`
- `$mfactor` is propagated to instantiated submodules

Ensures module behavior = `$mfactor` identical modules

But evaluated once, saving simulation cost

Cannot disable automatic scaling

Simulator warns if `$mfactor` is misused (e.g., manual scaling + automatic scaling = double count)

Examples of Correct and Incorrect Use



Incorrect usage:

```
I(a,b) <+ V(a,b) / r * $mfactor; // ERROR – double scaling
```


- Module: badres
- Explicit multiplication triggers double application

Correct usage:

```
if (r / $mfactor < 1.0e-3)  
  V(a,b) <+ 0.0;  
else  
  I(a,b) <+ V(a,b) / r;
```

- Module: parares
- \$mfactor used in logic condition only – no error
- **Other geometric parameters** (\$xposition, \$yposition, etc.):
- No automatic effect on simulation
- Used for layout-aware modeling only when explicitly handled in code

Paramsets – Definition and Structure



Paramset: A reusable, named configuration block for module parameters

Declared using:

verilog

CopyEdit

```
paramset <paramset_name> <module_or_paramset_name>;
```

```
...
```

```
endparamset
```

The second identifier:

- Typically a **module name**
- Can also be another **paramset**, allowing **chaining**

Must end with a module in the chain

Supports various declarations:

- parameter, localparam, integer, real, aliasparam

Paramset Usage and Rules



Assignments allowed:

- `.module_parameter = expression;`
- `.system_parameter = expression;`
- `.module_output_variable = expression;`
- Analog function statements
- **Constant expressions** support:
 - a. Literals, unary/binary ops, conditionals, hierarchical references
- **Constraint enforcement:**
 - a. If assigned value violates module parameter's declared range → Error
 - b. Simulator uses only paramset's own parameter ranges for selection
- **Example:**
 - a. Two `nch` paramsets for `nmos3` model:
 - i. **Short-channel:** `l = 0.25u, ad = 0.5*w, nfs = 0.8e12`
 - ii. **Long-channel:** `l = 1u, ad = 0.4*w, nfs = 0.7e12`
- Ensures flexibility and specialization without altering the base module

Module Port Declarations in Verilog-AMS



Port Type (Discipline):

- Declared using the port's electrical/discipline type:
`electrical [range] port;`
- If omitted, the port is restricted to structural use only (i.e., cannot be referenced in behavioral code).

Port Direction:

- Must be explicitly declared in the module body:
`input [discipline] [net_type] [range] port;`
- `output [discipline] [net_type] [range] port;`
- `inout [discipline] [net_type] [range] port;`

Module Port Declarations in Verilog-AMS (Contd.)

- 
- Both **type** and **direction** declarations must match in range.

- **Valid:**

```
input [0:3] in;
```

```
electrical [0:3] in;
```

- **Error:**

```
input [3:0] in;
```

```
electrical [0:3] in;
```

Named Port Connections



- Ports are connected by explicitly specifying the module port name and the local signal name:
`.port_name(signal_name)`
- The order of connections is irrelevant when using names.

Rules:

- Port names must match those in the module definition.
- Cannot use bit-select or part-select syntax for port names.

Named Port Connections (Contd.)



- The connected signal may be:
 - A net identifier
 - A vector member
 - A concatenated vector
- All connections in a given instance must be either all by name or all by position (not mixed).

Example:

```
adc2 hi (.in(in), .out(out[3:2]), .remainder(rem_chain));
```

```
adc2 lo (.in(rem_chain), .out(out[1:0]), .remainder(rem));
```

Hierarchical Naming and Scoping



Hierarchical Identifiers:

- Every identifier in Verilog-AMS has a unique hierarchical path formed by module instances and named blocks.
- Scopes are defined by module instances and named begin-end blocks.
- The `$root` keyword denotes the top of the design hierarchy.

Syntax:

```
$root.topmodule.submodule.signal
```

```
module1.instance1.signal
```

```
adder[5].sum
```


Hierarchical Naming and Scoping (Contd.)



Key Principles:

- Identifiers are scoped locally within their modules or blocks.
- A full path name ensures unambiguous access.
- Named objects can be accessed using full hierarchical paths from any point in the design.

Example:

```
$root.samplehold.op1.gain    // Accessing parameter in nested module
```



Ch. 8: Scheduling Semantics

Overview of Stratified Event Queue in Verilog



What Is the Stratified Event Queue?

- A layered mechanism in Verilog that segments simulation events into ordered processing regions.
- Ensures deterministic timing behavior and correct sequencing in both digital and mixed-signal domains.
- Essential for modeling concurrency, causality, and race-free behavior in hardware simulations.

Simulation Cycle

- A simulation cycle refers to processing all events scheduled at the current simulation time.
- Events are processed region-by-region to enforce order and ensure predictable results.

Event Execution Order



1. Active Events (Region 1)

- Digital assignments, procedural code, and continuous assignments.
- Can be processed in any order—source of non-determinism.

1b. Explicit Digital-to-Analog (D2A) Events

- Special region for D2A transitions.
- Executed *after* all active digital events to maintain causality with analog solver.

2. Inactive Events

- Triggered by #0 delay.
- Executed after Region 1 and 1b.
- Used to defer code execution within the same simulation time.

Event Execution Order (Contd.)



3. Non-Blocking Assignment (NBA) Updates

- Scheduled from \leq operations.
- Ensures non-blocking behavior.
- Processed after active, D2A, and inactive regions.

3b. Analog Macro-Processes

- Added for mixed-signal simulation.
- Analog processes (e.g., `@cross`, `@initial_step`) are handled here.
- Executed after all digital updates, maintaining analog-digital alignment.

Observations and Future Events



4. Monitor Events

- Triggered by \$monitor, \$strobe, \$debug.
- Processed *last* in current simulation time.
- Cannot schedule further events — purely observational.
- Always re-enabled in subsequent time steps.

5. Future Events

- Scheduled for a time later than the current simulation time.
- Subdivided into:
 - Future inactive events.
 - Future non-blocking assignment updates.
- Enable modeling of temporal behavior and delayed effects (e.g., #10, @(posedge clk)).

Special Cases & Mixed-Signal Integration



Explicit Zero Delay (#0)

- Suspends process and queues it as an **inactive event**.
- Ensures deterministic reordering at the same simulation timestamp.

PLI Callbacks

- Callbacks like `tf_synchronize()` and `vpi_register_cb(cb_readwrite)` behave like inactive events.
- Hook into simulation without disrupting event ordering.

Analog to Digital (A2D) Events

- Triggered via analog-controlled statements like:
 - `@cross(V(sig) - threshold, +1)`
 - `@timer(...)`
- Scheduled in Region 3b.
- Behave like digital events but aligned with analog solver convergence.

Switch Processing & Analog-Digital Event Handling



Switch (Transistor) Processing

- Switch elements (tran, rtran) allow bidirectional signal flow, unlike standard logic gates.
- Requires a relaxation technique to solve interconnected nodes iteratively.
- Handling 'X' states: The simulator evaluates all possible transistor states to determine steady-state behavior.

Explicit D2A Events (Region 1b)

- Digital-to-Analog events are processed after active digital events.
- Ensures analog solvers use finalized digital values before evaluation.

Analog Macro-Process Events (Region 3b)

- When multiple events trigger an analog macro-process, only one evaluation occurs.
- The analog solver ensures all dependencies are resolved before processing updates.



THE END