Enumerated Types in SV

Examples in Verilog

```verilog
`define FETCH 3'h0
`define WRITE 3'h1
`define ADD   3'h2
`define SUB   3'h3
`define MULT  3'h4
`define DIV   3'h5
`define SHIFT 3'h6
`define NOP   3'h7

module controller (output reg         read, write,
                   input  wire [2:0] instruction,
                   input  wire       clock, resetN);

  parameter WAITE = 0,
            LOAD  = 1,
            STORE = 2;

  reg [1:0] State, NextState;
```

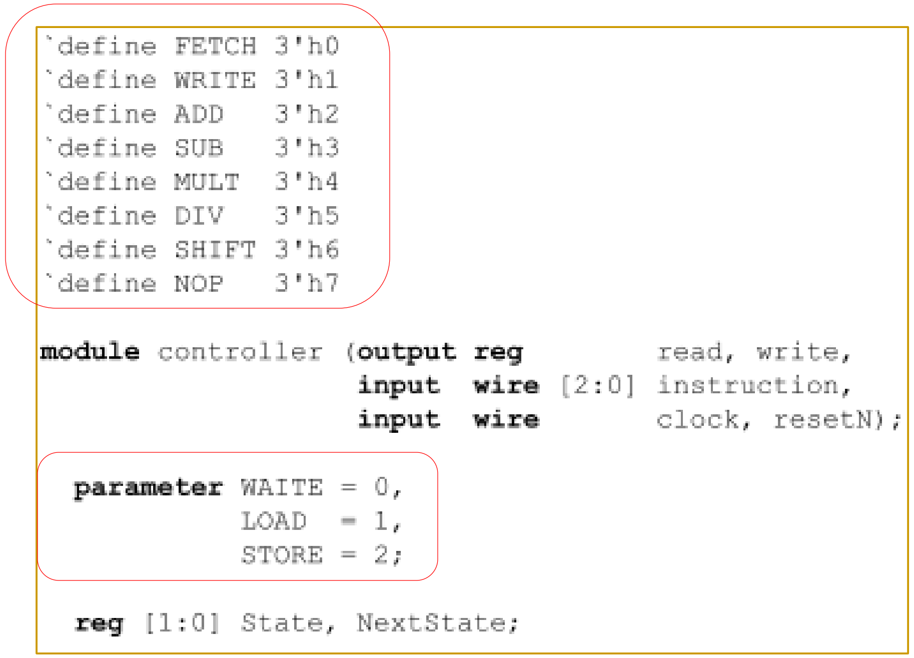
```verilog
  always @(posedge clock, negedge resetN)
    if (!resetN) State <= WAITE;
    else         State <= NextState;

  always @(State) begin
    case (State)
      WAITE:  NextState = LOAD;
      LOAD:   NextState = STORE;
      STORE:  NextState = WAITE;
    endcase
  end

  always @(State, instruction) begin
    read = 0; write = 0;
    if (State == LOAD && instruction == `FETCH)
      read = 1;
    else if (State == STORE && instruction == `WRITE)
      write = 1;
  end
endmodule
```

Enumerated Types

- Enumerated types provide a means for defining a variable that has a restricted set of legal values.
- The values are represented with labels instead of digital logic values.
- Enumerated types allow modeling at a higher level of abstraction, and yet still represent accurate, synthesizable, hardware behavior.

- In coding state machines, we need some enumerated types for states.

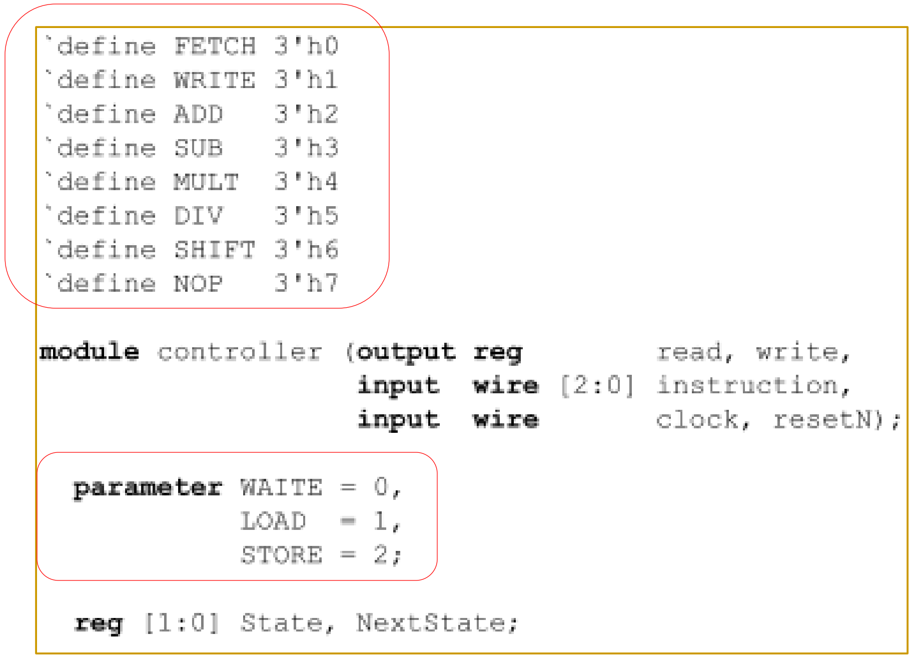
Verilog for Labeling Values



```
`define FETCH 3'h0
`define WRITE 3'h1
`define ADD   3'h2
`define SUB   3'h3
`define MULT  3'h4
`define DIV   3'h5
`define SHIFT 3'h6
`define NOP   3'h7

module controller (output reg        read, write,
                   input  wire [2:0] instruction,
                   input  wire       clock, resetN);

  parameter WAITE = 0,
            LOAD  = 1,
            STORE = 2;

  reg [1:0] State, NextState;
```

- Verilog does not have enumerated types.
- To create pseudo labels for data values, it is necessary to
  - define a parameter constant to represent each value, and
  - assign a value to that constant.
- Alternatively, Verilog's 'define text substitution macro can be used to define a set of macro names with specific values for each name.

Verilog for Labeling Values



```
`define FETCH 3'h0
`define WRITE 3'h1
`define ADD   3'h2
`define SUB   3'h3
`define MULT  3'h4
`define DIV   3'h5
`define SHIFT 3'h6
`define NOP   3'h7

module controller (output reg         read, write,
                   input  wire [2:0] instruction,
                   input  wire        clock, resetN);

  parameter WAITE = 0,
            LOAD  = 1,
            STORE = 2;

  reg [1:0] State, NextState;
```

- The example shows a state machine modeled using Verilog parameter constants and 'define macro names:
  - The parameters are used to define a set of states for the state machine, and
  - the macro names are used to define a set of instruction words that are decoded by the state machine.

User-defined Types: Typedef

- The original Verilog language only had built-in data types.
- SystemVerilog allows designers to create new, user-defined data types.
- Enumerated types can be declared as a user-defined type.

User-defined Types: Typedef

- An enumerated type declared using **typedef** is commonly referred to as a typed enumerated type.
  - **typedef enum** {WAITE, LOAD, READY} states_t;
  - states_t state, next_state;

SV for Labeling Values

```
enum {WAITE, LOAD, STORE} State, NextState;
```

- SV adds enumerated type declarations using the **enum** keyword.
- The declaration of an enumerated type is similar to C.
- SV enumerated types are useful for coding state machines.

The Default Base Type is **Int**

- Enumerated types are variables or nets with a set of labeled values.
- Designers can specify an explicit base type,
    - allowing enumerated types to more specifically model hardware.
- The default base type for enumerated types is **int**,
    - which is a 32-bit 2-state type.

The Explicitly Defined Base Type

- SV allows an explicit base type for the enumerated types to be declared.
  - enum logic [1:0] {WAITE, LOAD, READY} state;

  - // enumerated type with a **2-bit** wide,
  - // 4-state base type

Enumeration Value Size

- If an enumerated label of an explicitly-typed enumerated type is assigned a value, the size must match the size of the base type.

  -
    ```
    enum logic [2:0] {WAITE = 3'b001,
                      LOAD =   3'b010,
                      READY =  3'b100} state;
    ```

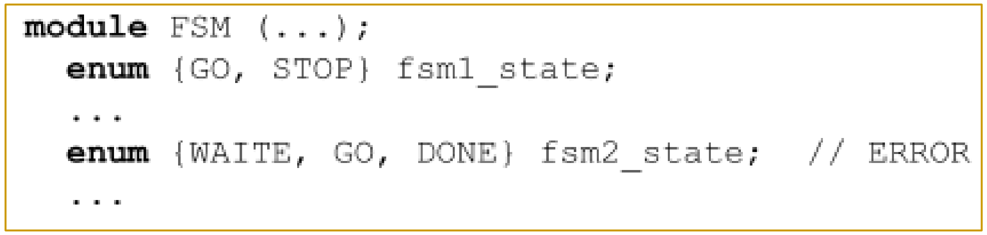
Labels with a Default Value

```
enum {WAITE, LOAD, STORE} State, NextState;
```

- By default, the actual value represented by the label in an enumerated type list is an integer of the int type.
- The first label in the enumerated list is represented with a value of 0,
  - the second label with a value of 1,
  - the third with a value of 2, and so on.

Label Uniqueness

- The labels within an enumerated type list must be unique within that scope.
- The following code has an error, as the enumerated label GO is used twice in the same scope:



```
module FSM (...);
  enum {GO, STOP} fsm1_state;
  ...
  enum {WAITE, GO, DONE} fsm2_state;   // ERROR
  ...
```

```
module FSM (...);

enum {GO=1, STOP} fsm1_state;

...

enum {WAITE, GO, DONE}
fsm2_state;

...

what happens???
```

Enumerated Type Label Scope

```
module FSM (...);

enum {GO=1, STOP} fsm1_state;

...

enum {WAITE, GO, DONE} fsm2_state;

...
```

- *Error-[IPD] Identifier previously declared*
  Identifier 'GO' previously declared as member of enum variable 'fsm1_state'

Enumerated Type Label Scope

- They are correct:

```
module FSM (...);
  ...
  always @(posedge clock)
    begin: fsm1
      enum {STOP, GO} fsm1_state;
      ...
    end

  always @(posedge clock)
    begin: fsm2
      enum {WAITE, GO, DONE} fsm2_state;
      ...
    end
  ...
```

Specify the Label's Value

- Designers can specify explicit values for any or all labels in the enumerated list. (State Encoding Problem)
- SV allows the value for each label in the enumerated list to be explicitly declared.

- For example, a state machine can be explicitly modeled to have
  - one-hot values, one-cold values, Johnson-count, Gray-code, or other type of values.

Enumerated Types

```
// Two 3-bit, 4-state enumerated variables with one-hot values
enum logic [2:0] {WAITE = 3'b001,
                  LOAD =   3'b010,
                  DONE =   3'b100} State, NextState;
```

- Enumerated types provide a means to declare an abstract variable that can have a specific list of valid values.
- Each value is identified with a user-defined name, or label.

Specify the Label's Value

- Each label in the enumerated list is represented as an integer value that corresponds to the label.

```
enum {ONE = 1, FIVE = 5, TEN = 10 } state;
```

Specify the Label's Value

```
enum {A=1, B, C, X=24, Y, Z} list1;
```

- The label A is explicitly given a value of 1,
  - B is automatically given the incremented value of 2,
  - C the incremented value of 3.
- X is explicitly defined to have a value of 24,
  - Y and Z are given the incremented values of 25 and 26, respectively.

- It is not necessary to specify the value of each label in the enumerated list.
- If unspecified, the value representing each label will be incremented by 1 from the previous label.

Uniqueness of Label Values

```
enum {A=1, B, C, D=3} list2; // ERROR
```

- Each label in the enumerated list must have a unique value.
- An error will result if two labels have the same value.
- The following example will generate an error,
  - because C = D = 3:

Enumerated Types

- If you don't initialize the label, each one would have a unique value.
- The value of first label by default is 0.

- // c is automatically assigned the increment-value of 8
  enum {a=3, b=7, c} alphabet;

  // Syntax error: c and d are both assigned 8
  enum {a=0, b=7, c, d=8} alphabet;

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

Correct ?

- enum {WAITE =3'b001, LOAD=3'b010, READY=3'b100} state;

Base Type Size = Value Size

- enum {WAITE =3'b001, LOAD=3'b010, READY=3'b100} state;

- The enum variable defaults to an **int** base type.
- An error will result from assigning a 3-bit value to the labels.
- It is an error to assign a label a value that is a different size than the size declared for the base type of the enumerated type.

Inconsistency on Base Type Size

- It is also an error to have more labels in the enumerated list than the base type size can represent.
  - enum logic {A=1'b0, B, C} list5;
  - // ERROR: too many labels for **1-bit** size

4-state Enumerated Types

- If the base type of the enumerated values is a 4-state type,
  - it is legal to assign values of X or Z to the enumerated labels.
  - enum logic {ON=1'b1, OFF=1'bz} out;

- If a value of X or Z is assigned to a label in an enumerated list, the next label must also have an explicit value assigned.
  - It is an error to attempt to have an automatically incremented value following a label that is assigned an X or Z value.
  - enum logic [1:0] {WAITE, ERR=2'bxx, LOAD, READY} state;
  - // ERROR: cannot determine a value for LOAD

Verilog and SV: Polymorphism

- Most Verilog and SV variable types are loosely typed:
  - Any value of any type can be assigned to a variable.
  - The value will be automatically converted to the type of the variable, following conversion rules specified in the Verilog or System-Verilog standard.
- Enumerated types are the exception to this general nature of Verilog.
- Enumerated types are semi-strongly typed.

Strong Typing on Enumerated Types

- Rules of Assignment:
  - An enumerated type can only be assigned (No operation result!) by:
    - A label from its enumerated type list
    - Another enumerated type variable of the same type (declared with the same typedef definition)
    - A value cast to the typedef type of the enumerated type

- Reason:
  - The computed results may be out of the scope of the range of the enumerated base type.

Strong Typing on Enumerated Types

- enum logic [2:0] {RED = 3'b001<<R_BIT,
- GREEN = 3'b001<<G_BIT,
- YELLOW = 3'b001<<Y_BIT} State, Next;
- ...
- always_comb begin: set_next_state
- **Next = 3'b000;**

Strong Typing on Enumerated Types

- It is illegal to directly assign **a literal value** to an enumerated type.
- An enumerated type must be assigned **labels** from its enumerated list, not literal values.

```
enum logic [2:0] {RED = 3'b001<<R_BIT,
                  GREEN = 3'b001<<G_BIT,
                  YELLOW = 3'b001<<Y_BIT} State, Next;
...
always_comb begin: set_next_state
Next = 3'b000; // clear Next - ERROR: ILLEGAL
```

Consistent Variable Assignment

```
typedef enum {WAITE, LOAD, READY} states_t;

states_t  state, next_state;

int foo;
```

- state = next_state;

Consistent Variable Assignment

```
typedef enum {WAITE, LOAD, READY} states_t;

states_t  state, next_state;

int foo;
```

- state = next_state; // legal operation
  - The state and next_state are both enumerated type variables of the same type (states_t).

Assignment for Non-Enumerated Types

```
typedef enum {WAITE, LOAD, READY} states_t;

states_t  state, next_state;

int foo;
```

- foo = state + 1;

Assignment for Non-Enumerated Types

```
typedef enum {WAITE, LOAD, READY} states_t;

states_t  state, next_state;

int foo;
```

- foo = state + 1; // legal operation

- The assignment statement is legal.
- The enumerated type of state is represented as a base type of **int**, which is added to the literal integer 1.
- The result of the operation is an **int** value, which is assigned to a variable of type **int**.

Operations on Enumerated Type Variables

```
enum {RED, GREEN, YELLOW} State, Next;

Next = State + 1;
```

Operations on Enumerated Type Variables

```
enum {RED, GREEN, YELLOW} State, Next;

Next = State + 1; // ILLEGAL ASSIGNMENT
```

- An operation on an enumerated type variable,
- Directly assigning this **int** result to the Next of an enumerated type variable is illegal.

Assignment of Computed Results

```
typedef enum {WAITE, LOAD, READY}
states_t;

states_t state, next_state;

int foo;
```

- `state = foo + 1; // ERROR: illegal assignment`

Assignment of Computed Results

```
typedef enum {WAITE, LOAD, READY} states_t;
states_t state, next_state;
int foo;
```

WAITE as an **int** with a value of 0,

LOAD as an **int** with a value of 1,

READY as an **int** value of 2.

- state = state + 1; // illegal operation
- state++; // illegal operation
- next_state += state; // illegal operation

- The computed results may be out of the scope of the range of the enumerated base type, which is a subtype of int.

- **This is based on IEEE Standard, but EDA tools may not implement them.**

Solution: Type Casting

```
enum {RED, GREEN, YELLOW} State, Next;

Next = State + 1; // ILLEGAL ASSIGNMENT
```

```
typedef enum {RED, GREEN, YELLOW} states_t;

states_t State, Next;

Next = states_t'(State + 1); // static cast

$cast(Next, State + 1); // dynamic cast
```

- A value of a different type can be assigned to an enumerated type using type casting.
- SV provides both a static cast operator and a dynamic cast system function.

Static Type Casting

```
typedef enum {RED, GREEN, YELLOW} states_t;

states_t State, Next;

Next = states_t'(State + 1); // static cast
```

- A static cast operation coerces an expression to a new type without performing any checking on whether the value coerced is a valid value for the new type.
- If the current value of State were YELLOW, then State + 1 would result in an out-of-bounds value.
- Using static casting, this out-of-bounds value would not be trapped.
- The SV allows software tools to handle out-of-bounds assignments in a nondeterministic manner.
  - This means the new value of the Next variable in the preceding static cast assignment could have different values in different software tools.

Dynamic Type Casting

```
typedef enum {RED, GREEN, YELLOW} states_t;

states_t State, Next;

Next = states_t'(State + 1); // static cast

$cast(Next, State + 1); // dynamic cast
```

- A dynamic cast performs run-time checking on the value being cast.
- If the value is out-of-range, then an error message is generated, and the target variable is not changed.
- By using dynamic casting, inadvertent design errors can be trapped, and the design corrected to prevent the out-of-bounds values.

Strong Type Checking

- Enumerated types have stronger rule checking than built-in variables and nets.
- These rules include:
  - The value of each label in the enumerated list must be unique
  - The variable size and the size of the label values must be the same
  - An enumerated variable can only be assigned:
    - A label from its enumerated list
    - The value of another enumerated type from the same enumerated definition
- The stronger rules of enumerated types provide significant advantages over Verilog.

An Example in Verilog

```
// Names for state machine states (one-hot encoding)
parameter [2:0] WAITE=3'b001, LOAD=3'b010, DONE=3'b001; // FUNCTIONAL BUG
// Names for mode_control output values
parameter [1:0] READY=3'b101, SET=3'b010, GO=3'b110; // FUNCTIONAL BUG
// State and next state variables
reg [2:0] state, next_state, mode_control;
// State Sequencer
always @(posedge clock or negedge resetN)
if (!resetN) state <= 0; // FUNCTIONAL BUG
else state <= next_state;
// Next State Decoder (sequentially cycle through the three states)
always @(state)
case (state)
WAITE: next_state = state + 1; // DANGEROUS CODE
LOAD : next_state = state + 1; // FUNCTIONAL BUG
DONE : next_state = state + 1; // FUNCTIONAL BUG
endcase
```

An Example in Verilog

```
// Output Decoder
always @(state)
case (state)
WAITE: mode_control = READY;
LOAD : mode_control = SET;
DONE : mode_control = DONE; // FUNCTIONAL BUG
endcase
endmodule
```

- The 6 bugs in the preceding example are all syntactically legal.
- Simulation will compile and run.
- Hopefully, the verification code will catch the functional problems.
- Synthesis might warn about some of the coding errors, but some of the bugs would still end up in the gate-level implementation of the design.

The Same Example in SV

```
module bad_fsm_systemverilog_style (...); // only relevant code shown
enum logic [2:0] {WAITE=3'b001, LOAD=3'b010, DONE=3'b001} // SYNTAX ERROR
state, next_state;
enum logic [1:0] {READY=3'b101, SET=3'b010, GO=3'b110} // SYNTAX ERROR
mode_control;
// State Sequencer
always_ff @(posedge clock or negedge resetN)
if (!resetN) state <= 0; // SYNTAX ERROR
else state <= next_state;
// Next State Decoder (sequentially cycle through the three states)
always_comb
case (state)
WAITE: next_state = state + 1; // SYNTAX ERROR
LOAD : next_state = state + 1; // SYNTAX ERROR
DONE : next_state = state + 1; // SYNTAX ERROR
endcase
```

The Same Example in SV

```
// Output Decoder
always_comb
case (state)
WAITE: mode_control = READY;
LOAD : mode_control = SET;
DONE : mode_control = DONE; // SYNTAX ERROR
endcase
endmodule
```

- The example shows these same coding bugs, but with the use of enumerated types instead of Verilog parameters and reg variables.
- The comments show that every functional bug in traditional Verilog has become a syntax error when using SystemVerilog
    - the compiler catches the bugs, instead of having to detect a functional bug, debug the problem, fix the bug, and re-verifying functionality.

Verilog → SystemVerilog

$$\begin{tikzcd}\boxed{\text{Functional Bugs}} \arrow[r] & \boxed{\text{Syntax Bugs}}\end{tikzcd}$$

Summary

- Enumerated types allow
    - the declaration of variables with a limited set of valid values, and
    - the representation of those values with abstract labels instead of hardware-centric logic values.
- Enumerated types allow modeling a more abstract level than Verilog,
    - making it possible to model larger designs with fewer lines of code.
- Hardware implementation details can be added to enumerated type declarations,
    - if desired, such as assigning 1-hot encoding values to an enumerated type list that represents state machine states.

References

- SystemVerilog for Design, by Stuart Sutherland et al.
- Sutherland, "SystemVerilog Saves the Day—the Evil Twins are Defeated! unique and priority" are the new Heroes", Synopsys Users Group (SNUG) conference, San Jose, California, March 2005.
- Mills and Sutherland, "SystemVerilog Assertions Are For Design Engineers Too!", Synopsys Users Group (SNUG) conference, San Jose, California, March 2006.
- Mills, "Being assertive with your X (SystemVerilog assertions for dummies)", Synopsys Users Group Conference (SNUG) San Jose, California, 2004.
- Mills, "Yet Another Latch and Gotchas Paper", Synopsys Users Group (SNUG) conference, San Jose, California, March 2012.
- Sutherland, "I'm Still in Love with My X!", Design and Verification Conference (DVCon), San Jose, California, February 2013.
- Greene, Salz and Booth, "X-Optimism Elimination during RTL Verification", Synopsys Users Group Conference (SNUG) San Jose, 2012.
- "Synthesizable Finite State Machine Design Techniques" by Clifford E. Cummings, Sunburst Design, Inc. 2003.
- "Synthesizing SystemVerilog" by S. Sutherland, D. Mills. 2013.