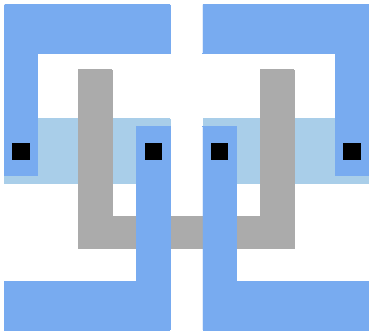
UNIVERSITÄT
HEIDELBERG

SystemVerilog
Interesting Synthesizable Features



Michael Ritzert

michael.ritzert@ziti.uni-heidelberg.de

08.02.2023

Why Should I Be Interested?

- Because there are many very useful extensions to „plain“ Verilog.
- Writing code is easier.
- Debugging in the simulator can be a lot easier.
- Even the performance might improve.

- Many new features add a bit of overhead to the code. More typing, more types, more LOC…
- But it's worth it. Trust me.

New Data Types

- 4-state: `logic`. Can replace both reg and wire.
- 2-state: `bit`. No `'x` or `'z`. Saves time and memory in simulation.

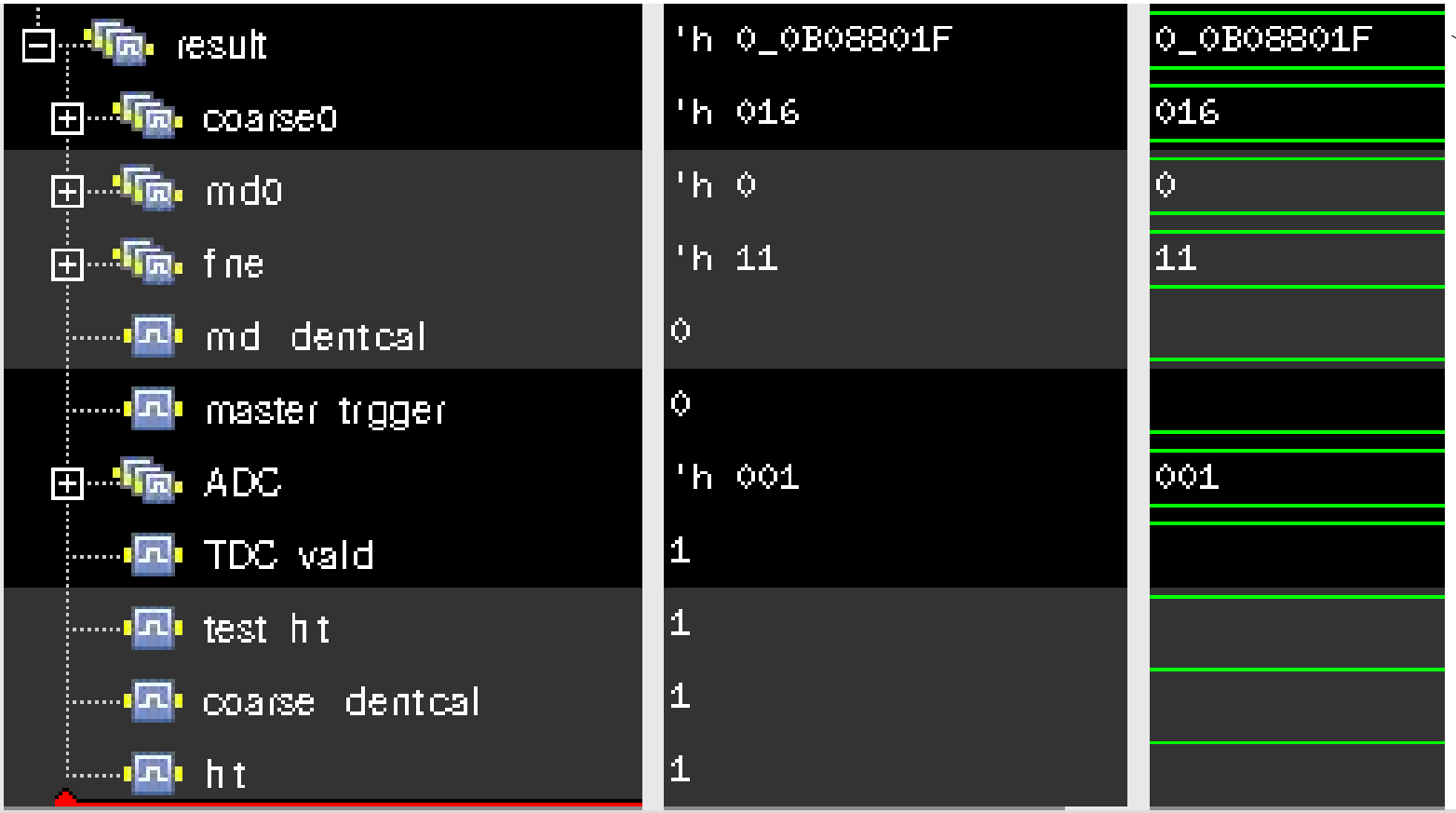
- `logic` behaves like `reg`, if assigned within `always_ff`, and like `wire`, if assigned in `always_comb`.
- The `always_ff` syntax is identical to `always`.
  - But the compiler warns if what is described is not an FF.
- `always_comb` replaces and improves(!) `always @(*)`.
  - Cannot miss inputs in the sensitivity list.

  There is also `always_latch`.

- Problem in FPGAs: Initialization in the declaration does not work together with `always_ff`/`always_latch`.
  ⇒ Do not use in code targeted for FPGAs.

Structures

- ```
  typedef struct packed {
    logic [11:0] coarse0;
    logic [2:0] mid0;
    …} result_t;
  ```
  essentially assigns names to parts of a vector.
- Can be assigned to/from vectors of same width.


  ```

enum

- New type: `enum`: Can represent one value out of a list of options.
- Representation can be specified:

```
enum bit [1:0] {
    NO = 2'b00,
    MAYBE = 2'b01,
    YES = 2'b10
} variable;
```

- Need typedef to make it a type:
  `typedef enum bit [1:0] {…} typename.`
- Then use variables of `enum` type,
- or just use the `enum` as a place to collect constants.
  They are compatible to the declared type of the `enum`:

```
logic [1:0] answer;
assign answer = YES;
```

enums as array indizes

```
typedef enum int {
  pad_trigger = 0, …, pad_inhibit = 3, …,
  pad_serout = 5
} lvds_pad_t;
```

```
typedef struct packed {
      logic [5:0] en_lvds_power;
      // VHDL-style logic [lvds_pad_t] is not possible
      …
  } global_config0_t;
```

```
localparam global_config0_t c_global_init_value0 = '{
  en_lvds_power : '{
     pad_trigger : 1'b1,
     pad_child1_serin : 1'b0,
     …
   } };
```

```
global_config0_t global_conf0;
```

```
always_comb gated_inhibit = INHIBIT &&
  global_conf0.en_lvds_power[pad_inhibit];
```

type to define the constants

configuration value

reset value for the register

variable holding the live register

access to a bit

- Same syntax as Verilog, but MUCH nicer display in the simulator.
- ```
  localparam [5:0] S_IDLE   = 6'b001001;
  localparam [5:0] S_DECIDE = 6'b010001;
  ```

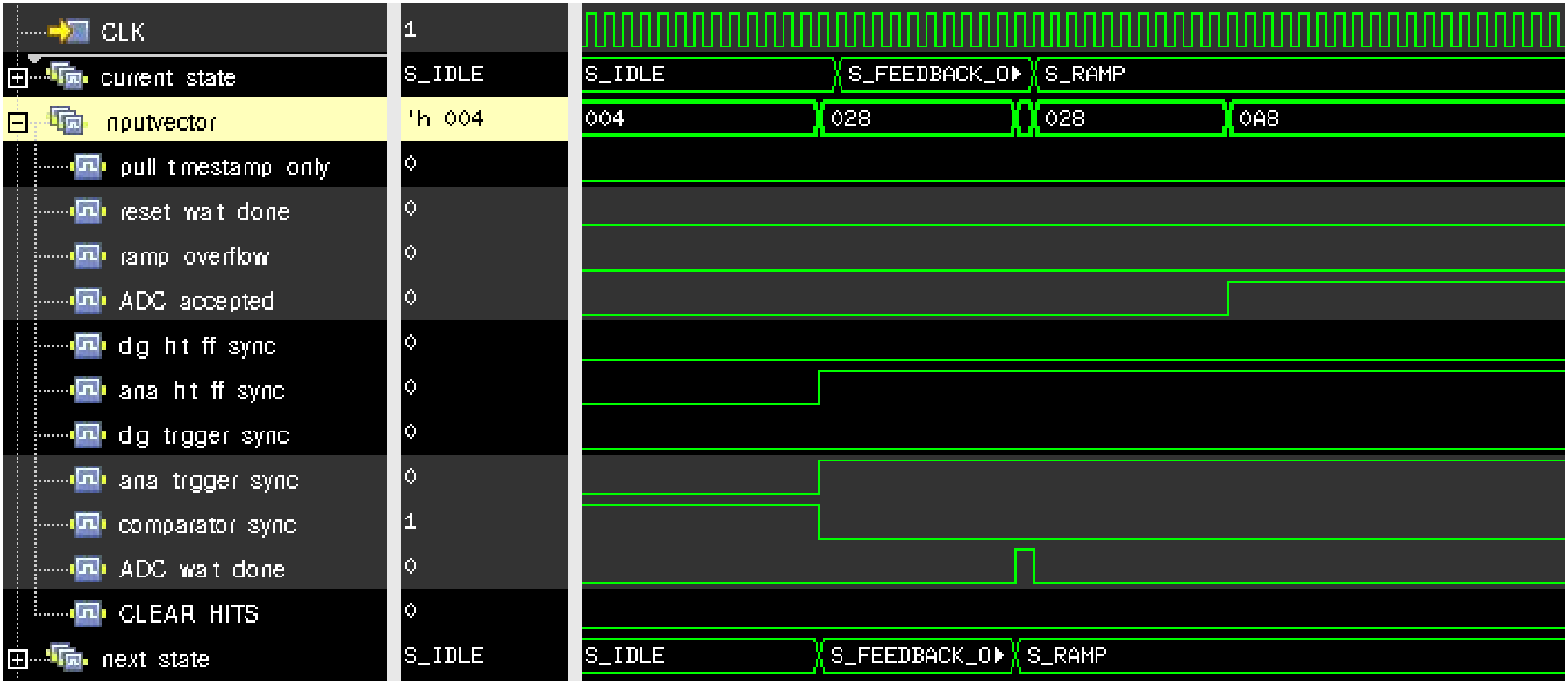
- ⇒
```
typedef enum bit [5:0] {
    S_IDLE   = 6'b001001,
    S_DECIDE = 6'b010001
} State_t;
State_t current_state, next_state;
```
⇒ nice display in Simvision.

- Otherwise completely compatible with `localparam` approach.
```

Statemachines: Input Vector

- If you want to take it even further:

```
struct packed { logic in1; logic in 2 } inputvector;
assign inputvector = { in1, in2 };
```



Also unions

- Where I generate time stamps, I collect bits from different sources: coarse, mid and fine counters.
- ```
  typedef struct packed {
        logic [11:0] coarse;    // bits 19:8
        logic [2:0] mid;        // bits 7:5
        logic [4:0] fine;       // bits 4:0
  } split_timestamp_t;
  ```
- But where I use the timestamp, I don't care about all these details.
- ⇒
  ```
  typedef union packed {
        split_timestamp_t split:
        logic [19:0] full;
  } timestamp_t;
  ```
- ⇒ The 20 bits of the `union` can be accessed in two different ways.
- To initialize a timestamp, I access `split.coarse`, `split.mid`, `split.fine`.
- To work with it, I use `full`.
- ⇒ Convenient + one source of possible errors (accessing the wrong bits) eliminated.
  ```

Advanced case and if

- Two modifiers to state your intent:
  - **`unique`** `case`/`if`…`else`: Conditions do not overlap. Excactly one will match. Checked by the simulator.

    ⇒ Conditions can be evaluated in parallel. Possible speedup.
  - **`priority`** `case`/`if`…`else`: Use the first matching branch.

    ⇒ Priority logic required.

- New constraint **`inside`**: Match against several values at once:
  ```
  if (value inside {1, 4, 9}) square <= 1'b1;
  ```
- Also useful with `case`: `case (…) inside` is a better alternative to `casex`/`casez`.

- Both can be combined: `unique case (…) inside`.

- **WARNING:** Blindly adding `unique`/`priority` on every `case`/`if` can break a design!

Arrays of Arrays

- Multi-dimensional arrays can easily be declared.
- `logic [20*12-1:0] DAC1;` (used as 20 12-Bit Entries)
  ⇒ `reg [19:0] [11:0] DAC2;`
- `DAC1[n*12+:12] <= value;`
  ⇒ `DAC2[n] <= value;`
- I will not go into the details of packed vs. unpacked, here.
  (`logic [11:0] DACs [20]`).
- When dereferencing: First unpacked, then packed, left to right.
  `logic [3:0] [3:0] wildArray [10][10];`
  ⇒ `wildArray[a][b][c][d]` accesses:
  `logic [c][d] wildArray [a][b]`.

- Both types can be assigned to each other:
  `assign DAC1 = DAC2;`

Downsides

- Integration into analog designs is poor (from analog side).
- E.g. on the digital side, we'd like to have `DAC[19:0][11:0]`, but this gives pin names that cannot be „caught“ with Cadence net expressions.
  ⇒ we still go with `DAC0[11:0]`, `DAC1[11:0]`, ….
- Note: This only applies to ports into/from the analog block.

  There’s no reason not to use the multi-dimensional arrays in the core code.

- Size of a variable (e.g. struct) in bits:
  `$bits(var)`.
- Adress bits for N memory addresses: 0…**N-1**:
  `$clog2(N)`.
- To **represent** N, use `$clog2(N+1)`.
  (Makes a differerence for $N=2^x \mid x \in \mathbb{Z}$).

- Module connections can use `.name` to connect a port to a net with the same name, or `.*` to connect all ports to matching nets:

```
module inst (input wire CLK, input wire RESET);

  inst I(.CLK(CLK), .RESET(RESET));
= inst I(.CLK, .RESET);
= inst I(.*);
~ inst I(.*, .RESET(!reset_n));
```

this assignment is used!

Interfaces

- `structs` can be passed between modules, but all signals of the `struct` are of the same directions.
- `interfaces` are there to overcome this.
- They contain a number of signals, and (typically) several `modport` declarations.
  - Different modules require different directions of the signals. Here is where this is defind.
- `interfaces` must be instantiated once.
  - Same syntax as for `modules`.

Interface Example: FIFO Data I

Let's consider a module with interfaces to two FIFOs, one for reading, one for writing:

```
module FT245SyncFIFO (
       input RX_FIFO_WRITE_CLK,
       output RX_FIFO_WRITE_DATA,
       input RX_FIFO_FULL,
       output RX_FIFO_WRITE_ENABLE,

       input TX_FIFO.READ_CLK,
       input TX_FIFO.READ_DATA,
       input TX_FIFO.EMPTY,
       output TX_FIFO.READ_ENABLE
   );
```

Ports belonging together logically, but with different directions
⇒ cannot use `structs`.

We cannot use structs, but still we'd like to somehow combine the ports.

⇒ pass around only one single object.

⇒ no need to modify all pass-through connections in the hierarchy, when a new port is added.

Interface Example: FIFO Data II

```
interface USB_FIFO #(
        parameter byte P_WIDTH = 32
    ) (
        input wire WRITE_CLK,
        input wire READ_CLK
    );
    logic [ P_WIDTH-1:0 ] WRITE_DATA;
    logic [ P_WIDTH-1:0 ] READ_DATA;
    logic FULL;
    logic EMPTY;
    logic WRITE_ENABLE;
    logic READ_ENABLE;

    modport SOURCE(
            input WRITE_CLK,
            output WRITE_DATA,
            input FULL,
            output WRITE_ENABLE
        );

    modport SINK(
            input READ_CLK,
            input READ_DATA,
            input EMPTY,
            output READ_ENABLE
        );
```

Note: can be parametrized
the clocks are passed in, where the interface is created.

instantiation (on top level):

```
USB_FIFO tx_fifo(
        .READ_CLK( intFTDI.CLK ),
        .WRITE_CLK( IO.CLK )
    );
```

as input to module:

```
module FT245SyncFIFO (
        USB_FIFO.SOURCE RX_FIFO,
        USB_FIFO.SINK TX_FIFO
    );


modport FIFO(
    input WRITE_CLK,
    input READ_CLK
    input WRITE_DATA;
    output READ_DATA;
    output FULL;
    output EMPTY;
    input WRITE_ENABLE;
    input READ_ENABLE;
);
endinterface : USB_FIFO
```

Packages I

- Shared definitions can be put in a „package“.
  - „constants“
  - type definitions: `structs`, `enums`, `interfaces`, ...
- Declared similar to a module with

```
package MyPackage;
⋮
endpackage : MyPackage
```

- Then in the module where the definitions are used:

```
import MyPackage:*;
```

(or `import MyPackage:name`).

- Note: Nets or registers cannot be declared in a package.

- Note: Compilers and simulators need to see the packages first in the list of inputs files.

Packages II

- One possible scenario:
  - Our design is very large.
  - Too large to simulate it, actually.
  - But it contains many identical blocks, i.e. „pixels“.

  ⇒ We could simulate with just a few pixels.
- But how to configure synthesis and simulation?
- Option a) Use `parameters`.

```
module #(parameter int num_pixels = 65536) top (
   input wire [num_pixels-1:0] pixel_in
);
```

  Then simulate with num_pixels set to 256.
- OK, this works.

  But suddenly we have (possibly many) parameters everywhere in the hierarchy.

Packages III

- Option b) Use packages.

  For synthesis:

```
package configuration;
   localparam int num_pixels = 65536;
endpackage : configuration
```

  For simulation:

```
package configuration;
   localparam int num_pixels = 256;
endpackage : configuration
```

- In the modules:

```
import configuration:*;
module top (
   input wire [num_pixels-1:0] pixel_in
);
```

- Then just use the different packages in the file lists for synthesis and simulation.

Packages IV

- Or boil it down to one define during the simulation:

```
package configuration_syn;
     localparam int conf_num_pixels = 65536;
  endpackage : configuration_syn

package configuration_sim;
     localparam int conf_num_pixels = 256;
  endpackage : configuration_sim

package configuration;
  `ifdef SIMULATION
  import configuration_sim:*;
  `else
  import configuration_syn:*;
  `endif
  localparam int num_pixels = conf_num_pixels;
  localparam int pixel_cnt_width
     = $clog2(num_pixels);
endpackage : configuration
```

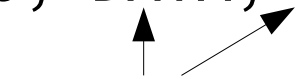
no code duplication for derived constants

Type as Parameter

- Advanced SystemVerilog (mostly for simulation):
  Pass types as `parameters`.
- Problem: Write a function to write a JTAG register.
  - Input data may be different (`struct`) types with different lengths
- A `task` cannot take a parameter, so wrap it in a `class`:

```
class JTAGtools #(parameter type T = integer);
   static task automatic write_dr(…, input T in,
                                  output T out);
      localparam int config_reg_depth = $bits(T);
      localparam [config_reg_depth-1:0] shifter;
      localparam int cnt_width = $clog2($bits(T));
      …
   endtask
endclass
```

- ⇒ `JTAGtools #(global_config_t)::write_dr(TCK, TMS, TDI, TDO, DATA, READBACK);`



of type `global_config_t`

Streaming Operators

- Used for bit-packing/-unpacking.
- Syntax rather complex, but lots of functionality.
- One example I found:

```
int j = { "A", "B", "C", "D" };
{>> {j}} // generates stream "A" "B" "C" "D"
{<< byte {j}} // generates stream "D" "C" "B" "A" (little endian)
{<< 16 {j}} // generates stream "C" "D" "A" "B" (16-bit chunks)

int a, b, c; (32 bits each)
bit [95:0] y = {>>{ a, b, c }};
```

Thank you!