BSG System Verilog Coding Standards

Prof. Michael Taylor & The Bespoke Silicon Group (Now at U. Washington)

Reduce naming paralysis and increase consistency!
See generally BaseJump STL (http://github.com/bespoke-silicon-group/basejump_stl) for examples of these coding guidelines in use.

See these slides for approved SystemVerilog code constructs, usages and style:

Part 1
Part 2

Don't use anything that's not on those slides -- it usually won't turn into correct hardware, or is unsupported across a wide variety of synthesizers.

Hardware Design Process

If you are designing a module that does not operate entirely in a single cycle (e.g. a single pipeline stage with feedback or multiple pipeline stages that have control signals), **don't just starting writing SystemVerilog**. First, iterate on the datapath on the whiteboard or a piece of paper. Be sure to look for opportunities to leverage the code from BaseJump STL -- it is already very efficient and well tested!

Keep on looking at all of the cases the datapath has to handle and make sure you have all of the pieces in the datapath before thinking about control logic. Then, label the control signals on your data path. You don't need to draw control logic itself unless it is a critical path that you intend to hand engineer.

After you have converged, then draw the circuit up using this style. Make sure most signals flow from left to right (this schematic actually disobeys this rule more than you should). Make sure registers that correspond to the same clock cycle are aligned. Make sure signal widths are labeled. Make sure registers are clearly indicated, and that all datapath control signals are labeled, but not drawn. **You should use google slides as your drawing tool**[1]**.** If you do not,

[1] *You might think that a schematic editing tool would better at drawing schematics than google slides, but they are not. :-P First, our schematics are not executable specifications; they are used to communicate design intent. Where you place things in the drawing matters, and we also leave out logic and wires (like control logic) that make the end intent less clear. Schematic generation tools don't do this well. Second, we don't want our schematics locked up in a proprietary format that makes it difficult for others to view, for instance by requiring them to install or register an app. Third, we want a representation that is long-lived, and will not be lose when a startup goes under. Fourth, although Google slides will require that you draw*

we will not use your work.

If your design is a datapath controlled by a state machine, then also draw the state machine in the google slides. If it is a pipelined datapath, there are multiple simultaneous states, and probably you will not be able communicate it clearly through graphics.

Make sure that your datapath flows from left to right, and registers that live at the pipeline stages are aligned vertically. Show the location of registers so that we understand the timing of signals in the pipeline. If you have hierarchy, show the “x-ray” view that shows registers and the connectivity of the registers inside the abstracted module, so that we can look at a module and tell how the timing works between modules. (If the module uses a valid->ready, valid & ready or ready->valid protocol as its only interface, you don’t need to show the abstract registers, although it may still be helpful.)

Use and Extend BaseJump STL

Read the BaseJump paper.

If you are designing hardware, and it has a component that is in BaseJump version, we expect you to use the BaseJump STL version. If it is not in BaseJump STL, but there is a common hardware primitive that you can pull out and put in BaseJump STL, then we expect you also to do that.

When you add something to BaseJump STL, it may be that you only need a special case of the generic concept. For example, maybe you only need a 1-D wormhole routed network, but you acknowledge that 2-D (or N-D) is also a common case. In this case, make the interface to the module be parameterized to support the more general case, but don’t worry about supporting the general case. Instead, just error out on the unimplemented or untested permutations.

Employ Latency Insensitive Interfaces

A basic principle of software engineering is that you want to design modules with simple so that you do not need to know how they work inside in order for them to work. This allows us to compose software and know that it will still work, just by looking at the interfaces. Traditional HW interfaces are often so complicated that you need timing diagrams in order to understand

---

some symbols (or use the ones in the provide style example), getting better at Google slides will translate to other activities, i.e. presenting at conferences. Fifth, using Google slides makes it trivially easy to add the schematic to a slide. =)

how to use them. Having interfaces that require both space and time to be orchestrated are complex and error prone.

The solution to this problem is to make use of Latency Insensitive Interfaces[2], which says that interfaces between modules should employ a set of standardized handshakes that perform handoff of data between components. Then we can compose together modules knowing that the interface is standardized, and that behavior does not depend on strange timing relationships. The use of the interfaces also enables an important property in the system, which is that a module can have variable delay, and the whole system will still work.

See the BaseJump STL Paper for an in-depth treatment of the motivation for the interfaces that we use and how to select among them:

- **v->r** (valid then ready) aka **valid-yumi**
- **r&v** (ready and valid) aka **rv&**
- **r->v** (ready then valid)
- **valid credit**

Code naming. Here is a concise summary of what we use for port suffixes for these handshake signals.

- `valid, ready_and` (r&v)
- `valid, ready_then` (r->v)
- `valid, yumi`[3] (v->r)
- `valid, credit` (v/c)

r->v is helpful if used on an input interface and demanding if used on an output interface. **valid->yumi** is helpful if used on an output interface and demanding on an input interface. The **r&v** and **valid/credit** interface requires that both sides of the interface be helpful. Two demanding interfaces cannot be connected, a FIFO would have to be inserted; a two-element FIFO is required to maintain one-item-per-cycle throughput; but if that is not required a one-element FIFO may work.

[2] Other folks have called systems with this property elastic systems.
[3] Yumi, as in “yummy, I consumed the data you gave me”. We use this instead of then_ready because we found from experience it is a little confusing to have a signal that is just a permuted name of another common signal.

		Consumer Demanding	Consumer Helpful
Producer	Demanding	Use a FIFO	`r->v`
Producer	Helpful	`v->r`	`rv->&`

Figure 4: Taxonomy of producer-consumer interfaces based on if producer and consumer are *helpful* or *demanding*.

For top-level links, **r&v** or **valid/credit** should be used on input and output interfaces, because it tolerates the most latency since it does not require a round-trip between producer and consumer. For these interfaces, the signals are expected to be sent early and received late in the cycle. Valid/credit is used for situations where the hardware needs more information about how much data the consumer can commit to absorbing.

For locally connected modules, you can also use **r->v** or **valid->yumi**. The signal before the -> is expected to arrive early in the cycle, and the signal after the -> is expected to arrive late in the cycle, so very little computation should be dependent on it. Here, the assumption is these modules will be placed relatively close by and wire delay is negligible.

The following timing table summarizes the implied timing of the handshake signals. (Note the `data` associated with the v wire matches the timing of the valid signals.) Note how the top-level interfaces are sent early but arrive late -- this is because of wire delay.

	v sent	**v arrives**	**r/y/credit sent**	**r/y/credit received**
r->v	late	late	early	early
v->r	early	early	late	late
rv&	early	late	early	late
valid/credit	early	late	early	late

Prefer helpful interfaces IF it does not add logic. To the extent that it does not require added logic, you should use helpful interfaces. But if it involves adding additional FIFOs, you should consider a demanding interface, since the user of the module can always add a FIFO to convert a demanding interface into a helpful one, and it may be that the user doesn't need the interface

to be helpful and would not need to add a FIFO. This is a key part of eliminating bloat from valid/ready interfaces.

Multi-interface modules. So far our rules have looked at modules with a single interfaces, and how to legally connect two such modules. However some modules may have several parallel interfaces, and use signals from one interface to combinationally derive signals on another interface. In this case, the way we connect these interfaces can indirectly violate the contracts that are implied by those interfaces. It is strongly discouraged for a module to connect via a combinational cloud a *late arriving* input signal to an output signal that must be *sent early*; for example a `yumi_i` signal can not be used to combinationally compute a `ready_and_o` or `ready_then_o` signal. But what about a *late arriving* signal to *late sending* signal? In this case, we permit it, but only if there is only a few gates added on that path. (Note since `ready_and` is *late arriving* and *early sending*, you cannot use input signals from one `ready_and` interfaces to derive output signals of another `ready_and` interface, since the input signal is late arriving and the output signal is early sending.)

These are prohibited.

1. Connecting two demanding interfaces without an intervening FIFO.
2. Misnaming an interface. For example, a ready_and/valid interface where valid is computed using the ready_and signal.
3. Anything but `credit` or `ready_and` is used on a top-level interface where two large modules are communicating and are located into separate places on the chip physical design. Common examples are NOC interconnected blocks.
4. Interface mismatch (`ready_and`, `ready_then`, `credit` or `yumi` signal name does not match between producer and consumer ports). If something is being converted, it should be named appropriately.
5. Not anding the `valid` signal with `ready_and`.
6. Within a module, connecting a *late arriving* input signal of one of its interfaces to an *early sending* output signal of another of its interfaces.

These are highly discouraged. A justification must be given in the comments, and it must be approved as an exception before committing to the repo.

7. Significant internal logic is used to compute a module's `ready_and` signal (i.e. `ready_and` not early).
8. Significant logic is used to compute a `credit` signal (i.e. `credit` not early). Credits should come out of flops and at most a few gates.
9. Naming a signal `ready` and not `ready_and` or `ready_then` (there is a fair bit of legacy code that does this and we would like to fix it.) Also `then_ready` is not a valid signal name, `yumi` should be used instead.

Never Chase Away Bugs; Catch Them and Kill Them

A common telltale sign of an amateur hardware or software designer is that when encountering a bug, rather than deeply understanding it and engineering a solid fix, they declare success when they find some other unrelated aspect of the code to change that “makes the bug go away”.

That bug is not gone; it is hiding! Just like a real insect, chasing the bug under the sofa is not solving the problem, in fact you are making it harder to track down the bug later, and they may multiply in the meantime. If you see a bug, you are lucky — bugs typically take a crap load of testing to even run into, and here you ran into one for free. So take advantage of the opportunity, approach it carefully and thoughtfully and be sure to thoroughly eradicate it when you see it.

Similarly, it is also wise not to ignore a bug if it is in a relatively simple program; you are likely to encounter the same bug in a piece of code that is much more difficult to debug.

Do Not Silently Fail or Succeed on Invalid Inputs

If a module gets an invalid input, it should at the very least print out a warning that clearly identifies the time and the module name where the issue happened, even if the module silently handles the input (i.e. by discarding it, or sending a NULL response.) Tracking down silent failures is extremely expensive in time, so we need to catch these kinds of issues as early as possible. The comments of the module should also discuss the cost of in-band flagging (i.e. returning an error) and ways of flagging the error, although this is often prohibitively expensive and not worth the effort something that doesn’t happy. But if it is something does happen sometimes even though it is technically wrong (e.g. speculation), then it should be discussed how we could differentiate between truly wrong behavior, and whether this wrong behavior can truly be ignored (i.e. some memory addresses have side effects if randomly touched!)

Envision the Meta Bug

A corollary to the above is that if you find a bug, you must spend the time thinking about the meta-bug. Not just what the bug you are staring at is and how to fix it, but how did this bug come to pass? What class of bugs does it represent, and how could the entire class be eliminated? Don’t just kill the bug, get the hole where the bug came from.

A common technique in HW design is random testing. Since the state space for hardware is exponential, it is impossible to find all bugs by random search. In fact, it is impossible to cover all but a trivial fraction of the search space. So random testing does not actually get you

coverage. Instead of fixing the specific random bug with a quick fix, you should be using it to identify metabugs — learn as much as you can about the space of bugs that this randomly found bug represents. Probabilistically, the bug you found is actually quite common in the state space of your hardware, and there are many variants of it out there.

Coding Style

Comment your Code

Here are some simple rules for when you should comment your code:

If you spent more than 10 seconds thinking about how to write that line of code, then write down what you were thinking about in a nearby comment!

If there was a technical discussion of the best way to do something, write that down. If you drew a picture on the whiteboard, then put an ascii version in the comments. (Or if it is a more complicated picture, draw it in a Google slide)[4]. If you have a waveform (somewhat uncommon since we use latency insensitive interfaces), you can use this cool meta-language (see also https://github.com/wavedrom/wavedrom) to generate the diagram (but check in the source as well!)

If somebody asks a question about your code during github review, put your response to that question in as a comment.

Use High True, Not Low True

To avoid bugs, all signals are **high true**, including resets and enables. If this is not the case (i.e. you are using a generated SRAM from a generator, make a big deal about it in comments in the code, i.e LOW TRUE!)

Use the Standard Suffixes

Dominate suffixes are listed first; i.e., an input that is known to be registered immediately before entry into the module is foo_r_i and not foo_i_r;

input ports	`_i`
output ports	`_o`

[4] Please create in your personal account and assign ownership to prof.taylor@gmail.com so that the document survives graduation (or retirement) from the university! Name it something unambiguous and place it in the Google share folder of the related project.

registers	**_r**
next values for registers	**_n**
struct types	**_s**
enum types	**_e**
macro parameter	**_mp**
global parameters These are constants that we want to define a globals across the entire source base.	**_gp**
parameters	**_p**
Local parameters and derived parameters that are needed for defining inputs and outputs but that should not be changed;	**_lp**
input or output of a child module (i.e. for the child it's an output, but for us, it's not an output signal); prefer _lo if both apply	**_lo, _li**
read address of a memory (wire going in)	**_r_addr_i**
write address of a memory (wire going in)	**_w_addr_i**
enable	**_en (but prefer v_i)**
write enable	**_we (but prefer w_v_i)**
read enable	**_ren (but prefer r_v_i)**
mux select	**_sel**
valid	**_v**
async	**_async**
For cases where a boolean indicates if a read or write request is being done, make it unambiguous that 1 means write.	**_write_not_read**
Input [..] foo_i, which was a parameterized structure that was passed in with the type erased, and has been cast to that parameterized structure.	**foo_cast_i**

In some cases, the suffix is enough to identify the signal. In this case, it is okay to omit the starting underscore. E.g. `v_i`, `v_o`. Similarly, if there is only one input and one output, you can just use `i` and `o` for the port names.

The name prefixes are there to make it much easier to reason about signals for people who are unfamiliar with the code, and greatly aids in both post-synthesis and post-place-and-route tracing of signals.

Prefixes

enum values	`e_`
log of a value	`lg_`
casez value (e.g. 32'b01??_?11)	`cz_`

Filename suffixes

Note that BaseJump STL currently uses .v instead of .sv. New projects should use these conventions.

SystemVerilog standard file	.sv
SystemVerilog include file	.svh
SystemVerilog package	_pkg.sv

Note: Macros should not be contained in _pkg.sv files, since correct operation depends on include order.

Enums

Never depend on the value of an enum directly, or one bit patterns in the enum.

(If you absolutely must for QOR, then you should define a macro next to the enum in the same file and have comments that they need to be simultaneously updated, but this is highly discouraged.)

- E.g.

```
typedef enum [1:0]
    { eAdder=0, eSub=1, eNop1=2, eNop2=3 } instr_e;

instr_e instr;

// NO
assign is_nop_i = instr[1];
assign is_adder_i = (! instr)

// YES
assign is_nop_i   = (instr == eNop1) || (instr == eNop2);
assign is_adder_i = (instr == eAdder);

// NO  (what about later added items like eSub?)
is_adder_i = (is_nop_i == 0);

unique case(instr)
  eAdder:            is_adder_i = 1;
  eNop1, eNop2:      is_nop_i   = 0;

 default:  // handle all other cases

endcase
```

Standard Parameters

If a module has a single parameter that is the number of bits wide input and/or output signals are, use **`width_p`** for that parameter. If there is more than one such width, then call it **`<xxx>_width_p`** where <xxx> is a descriptive name.

If a module has a parameter which is the number of elements in an array parameter, then call it **`els_p`** (i.e. short for elements_p). If there are several such parameters to a module, then call them **`<xxx>_els_p`**, where <xxx> is a descriptive name.

`harden_p` is a parameter that signals to the tools whether it should use a hardened version of the item, rather than a synthesized version. For example, this could apply to SRAMs, to

synchronizers, to particular gates that are placed in a certain place on an ASIC, to particular gates that should not be optimized away in FPGA, etc.

Parameter Decimation For complex modules (e.g, cores), the parameter space can become quite overbearing and greatly increase the code size as we route the parameters from the top-level module all the way down to the leaves. Instead of doing this, we suggest creating a configuration package. But rather than setting a single global package that defines these parameters, and does not allow instantiation of two different variants of the top-level module simultaneously, we suggest that the top-level package define arrays of parameters that are indexed by an enum. The enum value is passed down the hierarchy instead of many parameters. Then, as different consistent versions of a core are developed, a new enum can be added to the configuration package, and that enum can be used to index into the parameter array for each individual parameter.

Configuration parameter inheritance. The parameter decimation enum trick works quite well in terms of reducing the propagation of configuration parameters through the hierarchy, but maintaining these configurations can result in a lot of replicated code, where configurations only differ in a few key parameters. For instance, when choosing between configurations with different numbers of cores, the branch prediction information and cache configurations might be identical. In order to reduce redundancy and make the configuration file scalable, we use a set of clever macros.

```
// The parameter structure passed to modules, as described above
typedef struct packed {logic foo; logic bar;} my_param_s;

// Works by detecting the sentinel default value and selecting
whether //   to override each parameter.
`define MY_PARAM_INVALID "inv"
`define override_my_param(parameter_mp, override_params_mp, default_params_mp)
  parameter_mp: (override_params_mp.``parameter_mp`` == `MY_PARAM_INVALID)
                ? default_params_mp.``parameter_mp``
                : override_cfg_mp.``parameter_mp``
// Note this approach still requires a macro which overrides each
//   parameter in the parameter struct individually. One good thing
//   about this approach is that adding a parameter to the parameter
//   struct without adding to the config derivation will error out,
//   since all fields will not be set.
`define derive_my_params(new_params_mp, override_params_mp, default_params_mp)
  localparam my_params_s new_params_mp =
    '{`override_my_param(foo, override_params_mp, default_params_mp)
      ,`override_my_param(bar, override_params_mp, default_params_mp)
      };
```

```
localparam my_param_s base_params_lp = '{foo: 0, bar: 0};
// equivalent to:
//   my_params_s my_params_with_foo_lp = '{foo: 1, bar: 0};
localparam my_param_s foo_override_lp = '{foo: 1, default: `MY_PARAM_INVALID};
`derive_my_params(my_params_with_foo_lp,foo_override_lp,base_params_lp);
// equivalent to:
//   my_params_s my_params_with_bar_lp = '{foo: 0, bar: 1};
localparam my_param_s bar_override_lp = '{bar: 1, default: `MY_PARAM_INVALID};
`derive_my_params(my_params_with_bar_lp,bar_override_lp,base_params_lp);
```

You can see this in practice here:
https://github.com/black-parrot/black-parrot/blob/master/bp_common/src/include/bp_common_aviary_defines.svh
https://github.com/black-parrot/black-parrot/blob/master/bp_common/src/include/bp_common_aviary_pkg.sv

Parameter Consistency

You should not rely upon the user of a module to specify a set of parameters that are consistent; instead you should have independent parameters that are specified by the user (_p) and dependent derived parameters (_lp) that are derived by the code.

```
module #(parameter datapath_width_p = "inv"
        ,parameter index_bits_p = "inv"
        ,parameter assoc_bits_p = "inv"
        ,parameter tag_bits_p);
```

versus

```
module #(parameter  data_width_p = "inv"
        ,parameter index_bits_p = "inv"
        ,parameter assoc_bits_p = "inv"
        ,parameter tag_bits_lp =
data_width_p-index_bits_p-assoc_bits_p
        );
```

If you need a parameter in order to declare the width of an input or output port, it should go in the module signature. But if you do not need it for that purpose, it should go in the body of the module.

Module Instantiation

Use comma first notation and **fully qualified argument passing for both parameters and module inputs and outputs**; e.g.:

```
mSram #(.addr_bits_p(im_addr_bits_p)
        ,.width_bits_p(im_width_bits_p)
        ,.mask_sections_p(lg_assoc_p)
        ,.mask_section_width_p(im_width_bits_p)
        )
    im (.clk_i(clk)
        ,.reset_i(reset)
        ,.data_i(..)
        ,.addr_i(im_rwa)
        ,.mask_i(..)
        ,.wen_i(..)
        ,.ren_i(..)
        ,.data_o(im_do)
        );
```

Avoid `.*` and pass-by-parameter order. `.*` is okay if the module is 100% pass through.

Packed Versus Unpacked Arrays

Use packed arrays:

```
logic [els_p-1:0][width_p-1:0] foo;
```

or:

```
input [els_p-1:0][width_p-1:0] foo_i;
```

and never unpacked array[5]:

```
logic [width_p-1:0] foo [els_p-1:0];
```

[5] A rare exception is if you are creating a new synchronous read memory module for BaseJump STL (i.e. bsg_mem_1rw_sync_synth), then the unpacked array is more appropriate because it will synthesize to blockrams in Vivado.

Avoid splitting structs across multiple combinational statements

SystemVerilog tools will often do the wrong thing for code that looks like this.

```
struct foo;

assign foo.a = 23;

always_comb
  if (c)
    foo.b = 1;
  else
    foo.b = 0;
```

or

```
struct foo;

always_comb
   foo.a = 23;

always_comb
  if (c)
    foo.b = 1;
  else
    foo.b = 0;
```

Struct Initialization Must Avoid Inferred Widths

Vivado treats field initializers differently than Design Compiler. In DC 2019, the width of the target struct field can determine the width of the operator; in Vivado 2020, the operator's width is self-determined, i.e. only by the inputs to the operator.

```
logic [31:0] bar;
localparam pad_lp = 32;

typedef struct {
   logic [63:0] fract;
} foo;
```

THIS:

```
// fract is assigned the correct 64-bit value in both DC and Vivado
assign foo = '{fract: {bar, (pad_lp)'(0)}};
```

OR THIS:

```
// force output of shift to be 64 bit
wire [63:0] tmp = bar << pad_lp;
assign foo = '{fract: tmp };
```

NOT THIS:

```
// DC infers a 64-bit shift output; Vivado a 32-bit shift output
assign foo = '{fract: bar << pad_lp };
```

Pull resets + enables into flop creation idioms and out of combi logic.

I.e., do this:

```
// register creation
always_ff @(posedge clk_i)
  if (reset_i)  // reset has priority over en, as expected
     foo_r <= init_p;
  else if (en_i)
     foo_r <= foo_n;

// combinational next value
assign  foo_n = other_i + 32'b33;
```

and not:

```
always_ff @(posedge clk_i)
  foo_r <= foo_n;

always_comb
begin
  foo_n = foo_r;
  if (en_i)
    foo_n = other_i + 32'b33;
  if (reset_i)
```

```
    foo_n = init_p;
End
```

Be Aware of Synchronous Reset X-Pessimism Issues

Post-synthesis simulation can have issues figuring out that a system will transition from X to a known state. Besides using idioms above, there are some backup measures:

a. Make sure **hdlin_ff_always_sync_set_reset** is set to “true” in synthesis.
b. Check the synthesis report to see if the reset was identified (SR or SS should be set!):

```
=============================================================================
|    Register Name    |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
=============================================================================
|      state_reg      | Flip-flop |   3   |  Y  | N  | N  | N  | N  | N  | N  |
=============================================================================
```

c. Declare your synchronous reset signals to synopsys. Usually, it is just one signal, but in some weird cases, you might need several

```
// synopsys sync_set_reset "out_reset_i, out_reset_r"
```

See the appendix for “emergency measures” if you need to pass a module without resynthesizing even though it has an issue.

Never mix <= and =, and restrict use of blocking code to just flop creation idioms.

```
always @(posedge clk_i)
begin
   A_r <= b_r;
   c = b_r + 3’b1;
   If (x)
    A_r <= c;
end
```

Beware of accidental inferred latches

This is one of those anachronistic features of Verilog that we generally expect everybody to know, and that use of always_comb (instead of always @(*)) should prevent from happening, but we will say it here. In combinational logic, be sure that if a value is set in a combinational

block, that all paths in a combinational block set that value:

```
always_comb
    if (a)
     begin
       aar = 23;
       bar = 0;
       car = 0;
     end
   else if (b)
       aar = 0;  // error BAR is not set
       car = 23;
   else
     begin
        aar = 0;
        bar = 0;
        car = 23;
     end
```

In cases, for example in state machines, where there is a lot of commonality between signals (for example, most signals are “off” and some are turned on), then we suggest you start with a block of defaults, and then specify the exceptions. This results in fewer lines of code, less copy-pasting, and frames how we think about the code.

```
always_comb
begin
    aar = 0;
    bar = 0;
    car = 0;

    if (a)
      aar = 23;
    else if (b)
       bar = 23;
    else if (c)
       car = 23;
End
```

Always give a default for unique case, even if empty

If you use the “unique” keyword, in addition to saying that each of the case items is orthogonal to the others, implicitly says that **you do not care what the outcome is if the value is not one of the ones listed**. Eek. Take this example:

```
a = 23;

unique case (val)
  0:  a = 0;
  1:  a = 1;
  2:  a = 2;
endcase

  // error, if val is not 0,1,or 2, value of a is don't care in synth!
```

This is super error prone, in particular, because in Verilog simulation, this will cause the value of of **a** to be 23, but in synthesis, this will synthesize to don’t care, and generate a random value for a. Super not fun to track down, and super error prone. The fix is to always give a default case with unique case statements, even if the default is empty (!):

```
a = 23;

unique case (val)
  0:  a = 0;
  1:  a = 1;
  2:  a = 2;
  default:
endcase
```

Some tools may require a begin end statement after the `default:`

Have unused states in state machine go back to “waiting state”

A soft error or a tapeout glitch could result in a state register going to an unknown state, where the machine stays stuck indefinitely. It is wise to have the state machine jump back to the “waiting state” using a default case in the next state logic.

Use casez, not casex for pattern matching.

See the SystemVerilog slides for more details.

Be sure to use a default if it is a unique case.

Use don't cares (X) only in simulation; not in synthesis

An X in simulation is a way of indicating that something is invalid. An X in synthesis means that you don't care what the result is. These are very different things! In one case, you are trying to make it very clear that something is going wrong, and in the other case, you are saying, please pick a random output value. We want the first behavior but not the second.

Synthesis. Never indicate in your synthesizable verilog that you don't care what the result is. Adding don't cares means that the output of your module depends on the synthesis tool and the settings that you chose. It can result in hidden bugs that pop up later in your design, and they will be extraordinarily difficult to track down, since they will only appear in the gate-level netlist. Or for example, if you use an FPGA to simulate large test cases and synthesize using Vivado, Design Compiler very well could use different optimizations for ASIC, so your long running applications could work in FPGA but being too long, would not be run in ASIC gate-level simulation and would not work in the real chip since DC did something different. See also the rule "Always give a default for unique case."

Simulation. X's can be very useful in simulation. If you want to signal the output of a module is invalid in a particular case, then an X may be appropriate, as in this example[6]:

```
always_comb
  begin
      case (sel)
        2'b00:   y = a;
        2'b01:   y = b;
        2'b10:   y = c;
        2'b11:   y = d;
        default: y = 'x;
      endcase
  end
```

From a synthesis perspective, all valid values of sel are specified, so you are not specifying don't cares. From a simulation perspective, you are saying that you want to propagate don't cares through your case statement, and not just skip the case statement.

Getting the right behavior for Synthesis and Simulation. There are other cases where you may want to signal an 'X in simulation, but not want to tell design compiler that you don't care what the output is. In this case, we suggest using `` `ifndef SYNTHESIS ``, but be very sure that you are not accidentally creating a latch by having a non-default condition!

---

[6] http://www.sutherland-hdl.com/papers/2013-DVCon_In-love-with-my-X_paper.pdf

```
always_comb
begin
  is_crazy_n = is_crazy_r;

`ifndef SYNTHESIS
  if (crazy)
    is_crazy_n = 'X;
`endif

end
```

An alternate idea (proposed by Dan Petrisko), which we have adopted (in `BaseJump STL's bsg_defines.h`), is the following:

```
`ifdef SYNTHESIS
`define BSG_UNDEFINED_IN_SIM(val) (val)
`else
`define BSG_UNDEFINED_IN_SIM(val) ('X)
`endif
```

So then the above code becomes:

```
always_comb
begin
  is_crazy_n = is_crazy_r;
  if (crazy)
    is_crazy_n = BSG_UNDEFINED_IN_SIM(is_crazy_n);
end
```

Signal Width Definition

Don't use the [len-1:0] notation except in declarations (we would use them in declarations, but SystemVerilog does not allow it).

```
logic [im_width_bits_p-1:0] foo;

if (foo[0+:im_width_bits_p])
```

Combinational Assertions

Frequently, in a combinational block, we would like to add an assertion. The challenge is, that assertion could accidentally fire due to combinational propagation causing a transient condition. The SystemVerilog `assert final` construct was designed to address this issue:

```
// x and y are derived from complex combinational logic

assert final (reset_i !== '0 || x != y)
   else $error("x = %d, y = %d should not be equal!",x,y);
```

Note that if being out of reset is a precondition to the invariant being true (often the case) then the reset clause should be placed inside the assert.

Input / Output Ordering

We use a convention where conceptually earlier inputs dominate later ones if applicable. So for instance:

```
dff (input clk_i, input en_i, input reset_i, .. )
```

vs.

```
dff (input clk_i, input reset_i, input en_i, .. )
```

allows us to indicate that in the first case the enable prevents the reset from being asserted, while in the second case reset takes effect even if en is low.

With this said, you should never rely on the signature to be correctly ordered, and in the above case where en_i has priority over reset_i, the input should be called clear_i, since we always want reset_i to work regardless of what enables are there.

Error out on Untested Permutations

Problem: The use of metaprogramming constructs like **`parameter`** and **`generate`** statements results in a high testing burden for code because there are many cases.

This leads us down two roads:

In one road, we eliminate the use of these constructs, which is undesirable because it erases structure.

On the other hand, by specifying the parameter as a knob, we certify that all such parameters

are valid, which would require an impracticable amount of testing and a lot of wasted time.

Solution: As a result, for every parameter that is added, an **assert** should be added that flags for untested combinations of **`parameters`**. In general, if we expect it to be relatively unlikely that the user would want to change the value, we should **`localparam`** (or at least with suffix _lp if it needs to be part of the interface to the module) lieu of **`parameter`** so as not to falsely certify that a particular knob is functional.

```
initial
   begin
       assert (lg_assoc_p == 1);
           else $fatal("fetch only tested with lg_assoc_p==1;"
                       "no expectation it works with others");
   end
```

Generators

From time to time, writing a piece of code in SystemVerilog is too cumbersome. In those rare cases (usually when you want to generate a custom case statement), you may want to use a generator. For example, BaseJump STL has bsg_ascii_to_rom.py for generating ROMs.

The standard language for such generators is Python. A strong argument must be made for use of any other language.

`define

Use of `define is discouraged but not forbidden. When it is used, a comment should be placed explaining why a `define was necessary, and why the other methods (for example, using a localparam; VCS 2017.6 does not support the alternative) were not possible.

The pre-approved usages of `define are:

- To enable parameterization of structs because of missing support in SystemVerilog. We can use a `define to define the struct using its input parameters, within the context of a module, and then that particular definition will not leak out into other module's namespaces.
- To factor the code for reused functions that cannot be easily expressed as a module (i.e. `BSG_SAFE_CLOG2)

The `define should be context insensitive; i.e. all variables referenced should be explicitly given in the parameter list. Just like in C, parameters should be surrounded by parens to ensure that

when expressions are given to the macro, that they do not mix in unexpected ways with operator overloading, e.g.

`define foo(x) x*4
foo(3+1) → // 3+1*4 == 7, not 16!

Go easy on the Swiss Army Knife Design Pattern

Avoid “swiss army knife” modules that have many inputs and outputs, and then you select with parameters what those inputs and outputs do. This is very common in Xilinx IP generators.

The reasoning for avoiding these is that those modules are difficult to verify and use. The user must remember what combinations of parameters are valid, and which multiplexed inputs/outputs to use and not use in which cases. Verification needs to ensure that all outputs/inputs are valid under all parameter combinations. Additionally, these modules may not correctly specialize, and end up with extra logic (or at least require extra verification that they do not do this.)

An exception is if you are actually modeling a swiss army knife [or one of those Xilinx modules. =)] -- then provide the full interface, e.g. a DRAM interface --

Nonetheless, this is not a total prohibition against use of the pattern. Mild swiss army knife is sometimes preferable, in cases where a large amount of code reuse is possible with a small change that does not materially impact the design -- for example in bsg_noc, where we select between credit and ready signals (although even this has shown dangers!).

In some cases, there are two highly-related modules, one is a simpler version of the other. If the simpler case is frequently used, it may make sense to make a wrapper function for the simpler module that wraps around the other more complex module. On the other hand, if the simpler functionality is seldom used, then it’s probably better not to have it, and just require the user to specify the extra parameter.

SystemVerilog vs Verilog: always_ff

Use **always_ff** instead of **always**, use **always_comb** instead of **always (*).**

SystemVerilog vs Verilog: use wire for brevity

For better code brevity, instead of:

```
logic foo;

assign foo = bar + car;  // a bit long
```

use

```
wire foo = bar + car; // short and sweet
```

But not for logic/reg or struct symbols (these will not synthesize!!):

```
logic foo = bar+car;    // BAD, does not synthesize
reg foo = bar+car;      // BAD, does not synthesize
```

or

```
my_struct_s foo = { .. };  // BAD, does not synthesize
```

SystemVerilog vs Verilog: use logic and not reg

Use **logic** instead of **reg** and **wire**, except in the previously mentioned case of binding a wire to a value, which doesn't work with logic (or structs):

i.e. **wire foo = boo + 27;**

Use SystemVerilog structs to organize related signals and route them through the hierarchy.

`localparam` in module definition

VCS supports localparams in module definitions, i.e.:

```
module foo #(localparam happy_lp=1)
  (output o);

  assign o = happy_lp;

endmodule
```

At least since 2014.10. However relatively recent versions of Cadence Incisive (15.20, ~2017) and iverilog (11/23/14) do not support it. Since we want to be cross-platform, we should avoid; just use parameter and suffix with `_lp`.

Non-synthesizable modules

If a module is not synthesizable, clearly indicate that by prefacing it with the prefix **nonsynth**, i.e.

module bsg_nonsynth_clk_gen

In some cases, a module may be non-synthesizable, but intended to be replaced with an equivalent “hard” module. We would like a uniform interface, so we would call it one thing even though there are two versions. In this case, we suggest:

module bsg_hardsynth_clk_gen

Multiple Clocks

For modules that involve the interaction of multiple clocks, then every wire should indicate in the beginning of the name which clock it is synchronous to.

I.e.:

```
module bsg_launch_sync_sync #(parameter bits_p=”inv”)
  (input iclk_i
   , input iclk_reset_i
   , input oclk_i
   , input  [bits-1:0] iclk_data_i
   , output [bits-1:0] iclk_data_o
   , output [bits-1:0] oclk_data_o );
```

If a signal is not synchronous (including mesosynchronous) to a clock, then it should be labeled **_async**.

As is the case with all clock domain crossing, movement of signals across clock boundaries must be done very carefully. If you have not implemented logic like this before, talk to Prof. Taylor.

Do Not Use Interfaces; Use Structs for Bundling Wires

Interfaces, especially arrays of interfaces, which are very important, still are not supported in all synthesis tools[7]. If you write using interfaces, you will have to rewrite your code.

Justify your wrappers

Wrappers sometimes increase the size of the code-base disproportionately to the amount of functionality that they provide.

Sometimes they provide a valuable abstraction; allowing the user to leverage a bunch of code from elsewhere but slightly modifying the behavior in some way to reduce complexity to the user. An example is when you provide a few standard interfaces by leveraging a much more complicated submodule. Here, you are reducing code size and code replication by factoring. This is a great reason to use a wrapper.

In other cases, you are merely packaging non-interacting modules together. In this case, we suggest thinking about whether you need to do that; especially if the non-interacting modules could be used in a standalone way. Instead, just compress out this unnecessary level of hierarchy and flatten it into the parent module. An example would be packaging an outward link and an inward link together, where those links do not interact with each other.

Avoid Default Parameters

Don't set a default parameter unless they are really universal defaults. Recent SystemVerilog tools support this, but some older tools do not. Here is some example code that shows how to do this:

```
module add_invert #(`BSG_INV_PARAM(width_p))
(input   [width_p-1:0] i
, output [width_p-1:0] o
);
  assign o = i;
endmodule

`BSG_ABSTRACT_MODULE(add_invert)
```

To work around older tool issues, we define two macros in `bsg_defines.v`:

```
`define BSG_ABSTRACT_MODULE(fn) \
```

[7] It looks like DC 2020 may have added support for arrays of interfaces, so maybe in 5 years or so we will do a sweep of the other tools and see if they also support it.

```
    module fn``__abstract();     \
     if (0) fn not_used();       \
    endmodule

`define BSG_INV_PARAM(param) param
//`define BSG_INV_PARAM(param) param="inv"
```

The first macro is used to signal to the tools that the module is not a top-level module and should not be elaborated, according to this rule in the SystemVerilog IEEE Standard[8]:

The second is a handy macro that you can globally make all of BaseJump STL define the parameter back to “inv” if a tool incompatibility is encountered. (Note: one logical possibility is to specify ‘X as the default. we have never tried this and do not know if it works)

Use Proper Resets

FPGA style resets should **NOT** be used; i.e.:

```
reg foo = 0;   // FORBIDDEN
logic foo = 0; // FORBIDDEN
```

These rely on bitstream download to initial state, and are not portable to ASIC. Using this style of reset does have some benefits in FPGAs in terms of PPA, but restricts the usage model of the code, so we do not consider it to be worth the loss in portability.

Explicit resets are not required for every flop; however X’s should be cleared out of control state after some number of cycles K of reset being high. Additionally, the outputs of all modules (both control and datapath) should stabilize to a fixed value (i.e. should stop toggling) by K cycles. If K is more than 5 cycles, it should be documented in the comments.

All of your logic should be ready to go on the clock cycle after reset goes low, so we know that if the pipeline reset tree is balanced, everything will enter a valid state at the same time.

Whenever possible, synchronous reset should be used. Synchronous resets are easier to understand, more immune to glitches, and rely less on a perfect backend flow, and do not rely

---

[8] *Top-level modules are modules that are included in the SystemVerilog source text, but do not appear in any module instantiation statement, as described in 23.3.2. This applies even if the module instantiation appears in a generate block that is not itself instantiated (see 27.3). A design shall contain at least one top-level module. A top-level module is implicitly instantiated once, and its instance name is the same as the module name. Such an instance is called a top-level instance.*

on people correctly synchronizing deassertion of the reset. There are a few very rare cases where asynchronous reset is necessary; usually when you cannot rely on a clock working but still need to reset the module. This is usually only the case for initialization circuits for the chip, or for clock gated or power gated modules.

Don’t Roll Your Own Memories; Use BaseJump STL

Never code memories or register files in SystemVerilog or Verilog. Use the memories in BaseJump STL to ensure portability between multiple ASIC foundries and FPGA, using the BaseJump STL “hard” methodology.

Avoid memories that are greater than 512 bits of storage that employ asynchronous reads, which prevent the use of hardened memories and will cause synthesis of memories out of flip-flops.

Avoid multi-ported memories, unless it is a processor register file, or is a FIFO that requires reads and writes every cycle. And even in those cases, have a good reason for using a 1R1W and a really good reason for using more ports than that. You should be looking for architectural opportunities that eliminate the need for medium or large sized multiported SRAMs.

If you are synthesizing the logic, be sure to swap in the hardened SRAMs, as it will greatly reduce the number of instances in your design, leading to much faster runtimes of the tools. If you have a 128 KB ram, that is a 1-2 million instance design, which will take 12 hours to run through the tools.

For memories, do not use the xxx_synth modules directly, use the xxx modules that instantiate them. The synth versions are the fall back implementations after a hardened version is checked for.

Complexities of 1R1W RAMS

1R1W RAM Read/Write Conflicts. Most PDK’s provide 1R1W RF generators. However the semantics of what happens when simultaneous read and writes to the same address occur are not always the same. Here are some of the semantics that we have seen in the wild:

1. All of the data in the SRAM is corrupted when this happens.
2. The value in that memory location is corrupted, output is unknown.
3. The write happens and the read data out is invalid.
4. The read happens first and then the write.

5. The write happens first and then the read happens.

With some wrapper logic, and if masking is not required, we can convert underlying SRAMs that work as 1,2,3,4 into case 5, potentially with additional setup time delay, by comparing the address and disabling the read enable, and muxing in the result from the write. For synchronous rams built out of flip-flops like bsg_mem_1r1w_sync_synth.v in BaseJump STL, this is a natural semantic as well, since we can flop the address and do the write in parallel and then read out the data on the next cycle. In a masked SRAM, it would require some very specific semantics of masked write reads.

The most defensive (i.e. least error prone) and portable practice is to design your logic to not ever do simultaneous read and write of the same address (i.e. assume case #1 above) of your 1R1W's. This puts you in control, as the designer, to exploit your knowledge of the design to avoid this case; for example:

A. You are designing a FIFO and can prevent this case from happening because it corresponds to reading an empty FIFO. Your code will have to contain a proof that it prevents this scenario, and of course you should simulate with X's.
B. You may just add the bypass logic above for converting 1..4 to 5 (most likely not possible for a masked RAM).
C. You may choose to add a flow control signal to stall one of the requests. This approach opens the door for being able to replace the 1R1W with 1RW's (see below, generally only useful for > 128 words), but may add excessive complexity to your architecture.
D. You may be able to drop a request (generally a path that leads to complexity, then hard-to-find bugs, then sorrow).

Later on, if this ends up on a critical path, or has suboptimal architectural stalls for a target node, you can add logic that removes this avoidance code for specific tech nodes that have behaviors 2..5 and uses a standardized parameter to specify the specific behavior you want, and *if* the hardened implementation is correctly written, it will flag an error if that mode is unsupported.

Area per Bit for Memory

Numbers are approximate, for 12/14/16nm process technology, in um^2 per bit. Flip/flop and latch data assumes perfect placement, in reality, maybe 1.2X larger. Regions highlighted in grey are almost never the right solution. Regions in red/pink require a strong justification why some alternative architecture would not be better.

64 bits wide

Words	16	32	64	128	256	512	1024
1R1W, Flip flop	1.3	1.3	1.3	1.3	1.3	1.3	1.3
1R1W, Latches	1	1	1	1	1	1	1
1RW, RF Macro	1.3	0.7	0.41	0.27	0.18	0.14	0.13
1R1W, RF Macro	1.37	0.79	0.50	0.37	0.32	0.28	0.25

As we can see, the benefit of the hardened macros really kicks in at 32 words x 64 bits (=2Kbit). Interestingly, at least for this data, 1R1W hard RF's are only 12%, 22%, 37% bigger than 1RW hard RF's for 32,64,128 elements, respectively.

32 bits wide						
Words	32	64	128	256	512	1024
1R1W, Flip flop	1.3	1.3	1.3	1.3	1.3	1.3
1R1W, Latches	1	1	1	1	1	1
1RW, RF Macro	0.88	0.51	0.31	0.25	0.16	0.13
1R1W, RF Macro	1.05	0.66	0.50	0.45	0.33	0.28

Note, missing 16 word data, the columns in the tables are aligned based on total numbers of bits. Benefit of hardened macros also kicks in around 32 words (=1 Kbit), presumably because there are fewer sense amps (**so maybe the rule is, you need at least 32 words of storage for breakeven**; i.e. 32 bit cells per sense amp.) 1R1W hard RF's are only 1.2X, 1.32X, 1.6X, 1.8X bigger for 32, 64, 128, 256 word 1RW hard RF's respectively. Flip-flop based arrays with more than 2 ports may hold the advantage for higher word counts since the only choice is to replicate the 1R1W's to get more ports.

Near-peak efficiency of 1RW RF's is reached around 4KB of data, although 2 KB is not outrageous.

The “at least 32 words” rule may also apply for less wider SRAMs; however, typically they do not go that far down in size.

16 bits wide			
Words	64	128	256
1R1W Flip Flop	1.3	1.3	1.3
1RW RF	0.64	0.39	0.26

8 bits wide			
Words	128	256	512
1R1W Flip Flop	1.3	1.3	1.3
1RW RF	0.54	0.37	0.26

Memory Power

Inside SRAMs, you typically read out a lot of bitlines, and then they are run through a MUX, and then to the sense amps (because sense amps are large). This MUX inherently means that you are reading out data and throwing it away, burning power. So barring all else, you want lower mux factors in your SRAMs to reduce power. However, there is not a whole lot of flexibility there. And it may result in weird aspect ratios.

For small RAMs < 4 KB, SRAM energy increases only a small amount, like 10% when you double size. When you increase the width say from 32 to 64 (and keep total bits the same, but decrease words), you spend more area and energy on the sense amps, maybe around 1.5X energy. However you are getting 2X as many bits out of the array. If you are going to use 100% of the bits you read out, it is a no-brainer to read every other cycle and save energy 0.75X). Or alternatively, if you need more bandwidth, it is a no-brainer to double the width and read every cycle. However if neither of these is true, $1.5/(p*2+(1-p)*1) = 1.5/(p+1) = 1 \rightarrow p=50\%$ of your accesses have to use both words for it to break even for power.

Example: For a manycore IMEM, it might make sense, for example, to combine a tag plus two data words, rather than one tag plus one data words in fetching. You would do 3 fetches per 5-instruction loop instead of 5, so you are at (3*1.5)/5 = 0.9. In a 6-instruction loop, you are at

(4*1.5)/7 = 0.86. A 20 - instruction, loop you are at 0.79 (for the imem energy alone). The greater motivation could be that you can associate a tag with every pair of imem words, reducing IMEM area by 18%. On the flip-side, for manycore DMEM, you may have 1 load instruction every 3 instructions. Suppose that you went double width but never used it; energy would be (3/(3+0.25)) about 8% worse. Supposing you used it always; then energy would be (6/(2+3+0.5*0.5)), about 14% better.

Nuances of Replacing 1R1W's with 1RW's

Note: the rest of this section is for advanced discussion of use of 1R1W and can be safely ignored for novice designers.

Using 1RW for low duty cycle reader/writer SRAMs. In some cases, you may have independent readers and writers, but they both access the SRAM less than half the time. So effectively, there is a mismatch between the logical use of the SRAM and the physical bandwidths required. This pseudo 1R1W fifo in BaseJump STL is an example of taking advantage of this property. It prioritizes writes, so that it can always receive incoming data, and works the reads in the cracks. There is a parameter that says how many consecutive writes it should be able to handle before suffering performance problems for reads. This module is very useful for off-chip communication, where you need to buffer a large number of words to cover a high bandwidth-delay product, but the rate of incoming data is much slower than the core frequency.

Wide 1RWs as an area efficient alternative to 1R1W's. In large FIFO cases and other cases where reads or writes are always sequential, you can use a 1RW of twice the width to implement a 1R1W such as this fifo example in BaseJump STL. You can bunch up the words coming in and write them in pairs, and then read them out in pairs. However this has some unexpected overheads: 7 additional words of data stored in flip flops; twice as many sense amps in the SRAM leading to lower per bit efficiency, and a wide SRAM aspect ratio that may not be realizable with the SRAM generator; changing semantics about exactly how many elements might be in the FIFO, and longer latency. Breakeven versus 1R1W happens at 128 words (== still probably not worth it) for a 32-bit wide FIFO[9], and gains are quite large at 256 words.

Banked 1RW as an alternative to 1R1W's. An alternative to use a 1R1W SRAM is to use N banks of 1RW SRAMs that are of 1/Nth the size. If access patterns are sequential in nature, or if

[9] Double width 1RW: 7 words @ 1.3 u per bit + 128 words @ 0.41u = **62u per width bit**
Single width 1R1W: 128 words @ 0.50u = **64u per width bit**
Same numbers for 256 words: **78u per width bit** versus **115u per width bit**

they are extremely evenly distributed across the SRAMs and you have the ability to reorder a lot then you may be able to get near-perfect performance (with some latency cost) by using two banks.

If access patterns are random and happen every cycle, then it is a much harder challenge. The rule of thumb from 90's processor design is that you use 8 banks to approximate two ports (i.e. 8/64 probability of bank conflict = 12.5% conflict rate). The larger the window you can reorder requests (and the longer the latency), the lower the conflict rate you can achieve (e.g., 8.4% conflict rate over two consecutive cycles for 8 banks). Alternatively, if you use the second port only every other cycle, and can time shift to either slot, then the same 8-banked system would have a 1/64 probability of bank conflict.

Here are the area breakevens (excluding the actual logic overhead of any FIFOs, and ignoring conflict impacts on performance) for 1RW vs. 1R1W:

Banks	32-bit	64-bit
2	128 words	192 words
4	512 words	512 words
8	1024 words	1024 words

But by breakeven, we mean, if you are just breaking even, ignoring the complexity, so you should **not** consider banking at this point, but above it, there are significant potential area gains. So the high order takeaway is that you better have more than 128 words before thinking too much about banking.

Don't Use Latches or Negative Edge Triggered Flip-Flops

Avoid using latches, use only positive edge triggered flip-flops. Negative edge triggered flops are not as bad as latches but are still discouraged, because of complexity and because they impose requirements on the duty cycle of the clock (== timing assertions to tell the STA tools how much duty cycle variation you can afford).

Negative edge flops are used in the rare case where you are implementing DDR circuits that receive data on both clock edges.

Latches may be used for dense storage of data, with approximately 25% area reduction, but this should ideally be abstracted away behind BaseJump primitives as an implementation detail so

that can we port easily to FPGA and other platforms.

For generate statements, use `begin` and `end` and label begin

Synthesis tools want you to label your begin statements, and these labels are reflected in the post-synthesis names of gates. In many cases we don't care what the labels are, but would like to reduce the entropy in our code.

For this reason, we suggest using begin: fi for the first if statement; begin: fi2 for the second, and begin: rof for the first for statement and rof2 etc for the second. If you are writing structural RTL, then you may want to be more deliberate in naming. If you see in synthesis names like genblk and genblk2, it is because you forgot a statement. In cases where you expect the backend tools not to want to differentiate based on which part of the generate statement was taken, you can use the same name for both if and else clauses.

```
if (x_p)
  begin: fi
      adder     (this_input, that_input, another_input);
  end
else
  begin: fi
      wow_adder (this_input,          3, another_input);
  end
```

Don't label blocks that are not generate statements; it just adds extra entropy to the code.

Pay attention to Constant Padding

The rules for SystemVerilog constants and width extension are somewhat complex and even if you know them well, other people may not.

For a fixed constant, of course, just use the length:

```
27'b1
```

For zero, use:

```
'0
```

For string of all 1's, use:

`'1`

For a constant that depends on an input parameter, it is less straightforward, we advise using:

```
width_p `(1)
```

which is less subject to incorrect interpretation. It is technically the value 32'b1 zero-extended (or truncated) to width_p. It is more readable than { { (width_p-1) { 1'b0 } },1'b1 }.

You can also do math on the parameter:

```
(width_p-1) `(1)
```

Be careful of:

0 → actually 32'b0; so {0, 0} is a 64-bit number
1 → actually 32'b1; so { 1, 1} is a 64-bit number
~1 → not zero; it is {31'b1,1'b0}
`1 → ==1 only if it is a unsigned one bit value; it is actually -1 in two's complement.

Using $signed

The `$signed` operator (and also `signed `) has a lot of tricky rules, as well as the `signed` datatype. There are inconsistencies between tools (e.g. DC and Vivado 2020) that result in synthesis mismatches. In general, we suggest you avoid using them at all.

If you are trying to sign-extend a value to a wider width, we recommend you use the `BSG_SIGN_EXTEND and `BSG_ZERO_EXTEND macros rather than trying to implement it with signed data types.

There are a few places where the signed data type is necessary in order to use a verilog operator: signed comparisons and right arithmetic shift. In these specific cases, you should make sure that all inputs to the operator are signed, and that the output variable that the operator is assigned to is also signed; e.g.:

```
wire result = ( $signed(rs1_i) < $signed( rs2_i ) );
```

Aesthetic Style: Consistency

Don't use tabs in your files (except Makefiles), since they render inconsistently based on what your editor settings are.

If a given project has a declared style, use that style. (e.g., BlackParrot)

Otherwise,

1) if you modify a file, be consistent with the style in that file.
2) for new files, use a style consistent with the majority of the other files in the project.
3) for new projects, you can use the approximate style of this document.
4) do not change the white space in the file except to remove tabs; to avoid unnecessary git diffs

Aesthetic Style: Exploit Symmetry

When you are writing code, seek to exploit symmetry. If you have two or more complex statements that are highly similar, align the subterms vertically so that it is easy to see what is the same and what is different.

For example, instead of:

```
wire [1:0] v_li    = {(r_v_i & v_r[r_addr_i]) | w_li[1], (r_v_i & ~v_r[r_addr_i]) | w_li[0]};
```

Write:

```
wire [1:0] v_li    = {(r_v_i &  v_r[r_addr_i]) | w_li[1]
                     ,(r_v_i & ~v_r[r_addr_i]) | w_li[0]};
```

This way the reader can easily compare the pieces of the expression that are different and the same.

Aesthetic Style: Code Ordering

Code can often be arbitrarily ordered in a verilog file, but this can make it difficult to read. When possible, place signals that use signals after the inputs have been computed. Additionally, declare the signal relatively close to its first use in the file, so that the context is located as close as possible.

Beware of `if` and logical operator optimism on X conditions

If and logical statements in simulation are evaluated in a surprising-to-the-programmer way if the input contains X's[10]:

```
wire cond;

always_comb
..
a = 0;
if (cond)
  a = 1;
```

In most cases, if cond is `X`, we would like `a` to be `X`, but in simulation, it will just skip the `a=1` statement and `a` will keep its last value, `0`. In gate-level simulation, the synthesizer can generate basically any behavior. `a = cond;` would be a safer way to write this code.

Multibit behavior of `if` and logical expressions (i.e. && or ||) is also somewhat perplexing. `if` and `||` evaluates to true if any bit is set to 1. `&&` evaluates to false if any bit is set to 0. `X`'s are only considered if the signal does not evaluate to true.

Here is an example that illustrates the differences between the two:

```
a = 2'b1X && 2'bX1; // evals to 1; equivalent to (|2'b1X) & (|2'bX1)
a = 2'b1X & 2'bX1;  // evaluates to X
```

Be "pure" in your Logical Expressions

To improve clarity, we group logical expressions (especially for `if` statements, but also for general logic expressions) into three cases; 1) the common case in control logic where you are operating on single bit data, 2) a special case where you are manipulating *only* word-level arithmetic data and 3) the most general case, where you are manipulating at least some bit-vector data, with or without arithmetic data.

A. **Purely single bit data.** Here, you should use the bitwise operators (e.g., `&` and `|`), and none of the logical operators (e.g., `&&`, `||`, `!`), to convey that you are operating on single-bit data in the expression, which is a purely 1-bit boolean expression.

---

[10] http://www.sutherland-hdl.com/papers/2013-DVCon_In-love-with-my-X_paper.pdf
The logical AND and OR operators evaluate each operand to determine if the operand is true or false. These operators have two levels of X-optimism that can hide X or Z values: • An operand is considered to be true if any bit is a 1, and false if all bits are 0. For example, the value 4'b010x will evaluate as true, hiding the X in the least-significant bit.. • Logical operators "short circuit", meaning that if the result of the operation can be determined after evaluating the first operand, the second operand is not evaluated.

B. **Purely logical operators for purely non-bitvector inputs.** Here you have a bunch of word values that you that you are trying to combine together, C-style into logical boolean expressions. The expression will have a bunch of numerical values that you are testing for equality, less than, greater than, or zero/non-zeroness. You can not have any of the inputs to the expression be bitvector values. In this case, it is permissible to use logical boolean operators (i.e. &&, ||, if, !), but you should not use any bitwise operators (i.e., &, |, ^), especially the bitwise ~ operator. This is consistent with C usage.

C. **Purely bit-level operators for when at least one bitvector is involved.** The third case is that you actually have bitvector terms somewhere in the expression. In this case, you should not use the logical boolean operators. Instead, you should explicitly convert each of the input terms to the appropriate single boolean value, and then use the bitwise boolean operators to generate the final value. You must explicitly use ==, `==‘0, !=,!= ‘0, unary |, unary &, unary ^,` to reduce multi-bit values to boolean values before applying logical bitwise boolean operators. One particular gotcha to watch out for is the innocuous looking `if (~x),` with `x` a multi-bit signal, which does not resolve the input value down to a single bit as required above, but instead is true if any of the bits in x are zero (i.e. `~&x`), in contrast to `if (x == ‘0)` which is `&(~x)`, which is true only if all of the bits are zero.

Aside: Issues with mixing ~ with if, &&, ||; or ! with binary &,|

Generally with single bit signals, this will not cause any grief, but with multi-bit signals it is likely to.

```
if  (a && ~b)
```

is actually saying “ if there is any 1 in a, and any 0 in b”, when actually you probably meant:

```
if  (a && !b)
```

which is actually saying, “if there is any 1 in a, and b is all zeros”.

So for example,

```
val = 2’b01;
result = (1'b1 && ~val) // → evaluates to true (!!)
```

And for completeness, let’s evaluate:

```
val = 2'b01;
a   = 2'b10;
result = (val & !a) // → a & 0 → as "expected"

val = 2'b10;
a   = 2'b00;
result = (val & !a) // → a & 1 → 0, unexpected
                    // failure to reduce term before anding
                    // should be val && !a, or |val & !a
```

For reusable modules, omit the outer loop

For complex modules, if the module has an outer generate loop and all of the logic is contained within that outerloop, then consider omitting the outer loop for the generic interface. Otherwise, we must consider for all modules, whether an outer loop should be added. So for example, imagine a module that instantiates els_p fifos, but all of the fifos are independent. Consider whether this is really necessary, or whether the outer loop should just be in the parent module.

Prevent unexpected concurrency in module interfaces.

Modules dropping inputs or otherwise failing under rare concurrent events is a common bug pattern that takes a long time to track down due to the infrequency of such events. (An example would be, the front end of a processor responding to an icache miss, dcache miss, a branch mispredict, and an interrupt all at the same time.)

Generally the assumption for any module should be that if it has multiple interfaces, they are concurrent; and there must be support for handling the concurrency.

Whenever possible non-concurrent interfaces should be collapsed into a single interface (particularly if the non-concurrent interface comes from the same place) since that avoids accidental introduction of concurrency later. The classic example is a memory that can do only one read or write per cycle. Rather than having a read_v and write_v signal, you should have a signal valid (v_i) signal, and then another signal, write_not_read_i which indicates which one it is.

In the comments there should be explicit reasoning about what concurrent events can and cannot happen.

Timing constraints for stable signals; don't use set_false_path or set_disable_timing

Sometimes signals that are known to be stable end up on critical paths because they are needed in several places and there is no good place to place them. The preferred approach to solving this is to use pipeline registers to distribute these signals and maintain the cycle time and timing integrity. This fits in the standard flow.

Sometimes people are tempted to mark paths using set_false_path or set_disable_timing. This raises concerns about whether those paths are resistant to signal integrity, whether the tool will take unlimited slack and do unreasonable things, etc. In very limited cases, it may be permissible to use set_multicycle_path, for 2 cycles, on these paths. However the specific case is that these signals will be set BEFORE reset and never again changed. This way, any metastable signals have a chance to shake out before getting started. Setting a multicycle path on a reset signal is almost always a mistake, because the receiving flops can have violated setup or hold times, and different parts of the machine could come out of reset at the same time.

Do not use set_false_path or set_disable_timing unless the data truly never goes between those two points, they are silent chip killers, and this is actually an exceptionally rare case.

Clock gating: introduction

Clock gating is super important for energy and area efficiency in modern CMOS. The principle mechanism for clock gating occurs when the tool detects a DFF with enable that is more than 3 data bits wide. Typically without clock gating this is implemented using an array of DFF's with a MUX in front of it (in a FinFet technology); or DFF_ENs cell which is 25-30% larger than a plain DFF. With clock gating, the MUX's are removed and the enable signal is fed into a special clock gate cell that is inserted, which then provides the clock for the DFF's. When the enable signal is low, the clock gate will suppress the clock from reaching the flops. The net effect is pretty decent area savings because the muxes are gone, and also less capacitance and thus less power for the same reason. There is an additional effect, which is, when the enable is low, the clock signal does not toggle on all of the DFF's. Although clock and data often have similar capacitance, the clock energy is 4/5 or more of the power of the flip-flop without clock gating, because the clock switches 2X per cycle, while the data is 0.5X per cycle (in the case of random data) and often less for real data patterns.

To make sure that the synthesis tool is finding clock gating opportunities, you want to code the datapath registers using bsg_dff_en, rather than embedding the writes to the dff inside complicated always_begin blocks. So instead of:

```
always @(posedge clk_i)
  foo_r <= foo_n;

always_comb
begin
  foo_n = foo_r;

 if (complex expression)
  foo_n = <something else>
end
```

you want something more along the lines of (add a helpful comment if you considered clock gating when writing the code):

```
// expectation: clock gating
bsg_dff_en(.clk_i, .en_i(en_li)
           ,.data_i(something else)
           ,.data_o(foo_r)
           );

always_comb
      en_li = complex expression;
```

Clock and data gating: modules that wait around

Imagine you are writing code for a module that spends significant % of time waiting for an input, while doing nothing (for example an bsg_idiv_iterative divider, which because of the low intensity of divides in a program, spends most of its time waiting). If this module already has enables on its input flops, make sure that these flops are coded as bsg_dff_en's, *and make sure that the enable is only set in the WAIT state when valid is asserted*. This will clock gate the input flops of the module, and prevent unused data from toggling through the module's combinational logic, saving energy. Potentially, this may become a critical path, and we may add a parameter to avoid this if it ends up being a common critical path.

A more nuanced tradeoff is if the module does not currently have enables on input flops. If we add an enable, this is actually adding area and capacitance. So here, if we add an enable, we may be more focused on the savings due to data gating (preventing signals from toggling in combinational logic after the flop) rather than clock gating. Generally these kinds of optimizations will be driven more by energy profiling than by pre-emptive coding; and the wider the registers, and the more follow-on combinational logic there is, and less "waiting" you need

for this to make sense to do. For sure, if you insert this kind of code, you should explicitly tag it as data gating logic.

Another interesting case is the case where the inputs go through muxes before going to registers. In this case, the WAIT state should configure the mux so that when data is not valid, it chooses the input that selects from internal inactive non-toggling logic, rather than the live inputs. This will reduce the toggling of the data bits on the input flops. Similarly, if there is a configuration register that is set at the beginning, but is not modified during the operation (for example, the inner loop of the iterative divider) of the module, then you would want to direct the mux select line to an input that does not toggle. If there is no such input, but the flop has an enable anyways, you can consider using a bsg_mux_one_hot, which can zero the output of the mux if the select line is 00. However, zeroing an output does cause toggles, so you would want to make sure that you will be setting this for multiple cycles to amortize the cost of the toggles caused by zeroing the output.

Appendix A

Emergency X propagation Measures

Sometimes you may find that a X-pessimism issue has snuck through to a gate level netlist and it is too expensive to regenerate the netlist with the appropriate sync reset directives.. Here is how you might possibly verify that this issue is okay by trying all combinations of initial values for the registers in question.

1. First, extract the module into a verilog file. Then, configure a makefile for a vcs simulation, pulling in the library files for the standard cells. VCS will use an initial_file that contains the initial values for simulation.

Makefile:

```
include bsg_cadenv/cadenv.mk

LIBRARIES=$(sort $(subst _pwr,,$(shell  find /gro/cad/pdk/gf_14/invecas/STDLIB/7P5T/BASE -iname "IN14LPP_SC7P5T_84CPP_BASE*.v"))) $(sort $(subst _pwr,,$(shell  find /gro/cad/pdk/gf_14/invecas/STDLIB/7P5T/HPK -iname "IN14LPP_SC7P5T_84CPP_HPK*.v")))

FILES = scan.v CGRA_CoreCtrl__df87050082522213.v -y $(LIBRARIES)

$(warning $(VCS_HOME))

all:
	$(VCS)  $(FILES) -sverilog +vcs+initreg+config+initial_file
	./simv
```

2. Then, instant multiply copies of the module, one for each combination you want to try. You should ideally only need to wire up clock and reset.

scan.v:

```
`timescale 1ps/1fs

module verify_reset();
  localparam width_p = 7;

  logic              clk;

  initial
    begin
            $display("t=%t %b %b",$time, foo.state[1], foo1.state[1]);
            #100
            $display("t=%t %b %b",$time, foo.state[1], foo1.state[1]);

            $display("starting clock");
            clk = 0;

            for (integer i = 0; i < 7; i++)
              begin
          #100 clk = ~clk;
          #100 clk = ~clk;
              end

       $finish;
    end

 brg_cgra_pod_CGRACoreCtrl__df87050082522213_0 foo
  (
   .clk(clk)
   ,.reset('1)
   );

 brg_cgra_pod_CGRACoreCtrl__df87050082522213_0 foo1
  (
   .clk(clk)
   ,.reset('1)
   );

  // print out all values of wire
  always @(negedge clk)
    $display("%t negedge CLK -> %b %b", $time, foo.state[1], foo1.state[1]);

endmodule // verify_reset
```

3. Create an initialization file for the registers in question.

initial_file:

```
instance verify_reset.foo.state_reg_1_ 0
instance verify_reset.foo1.state_reg_1_ 1
```

We have found that the tools sometimes hang when doing this, and you may need to add this to the testbench:

```
  initial begin
   force DUT.pod.hb_cgra_xcel.cgra_xcel.dpath.cgra.ctrl.state[1] =
1'b`X_PESSIMISM_FORCE_VALUE;
   @(posedge DUT.pod.hb_cgra_xcel.cgra_xcel.dpath.cgra.ctrl.reset);
   @(negedge DUT.pod.hb_cgra_xcel.cgra_xcel.dpath.cgra.ctrl.reset);
    release DUT.pod.hb_cgra_xcel.cgra_xcel.dpath.cgra.ctrl.state[1];
   end
```

Future Ideas

Reconcile StyleGuide with:

https://github.com/lowRISC/style-guides/blob/master/VerilogCodingStyle.md#function-and-task-calls

Other Guides

BSG Guide to JasperGold