Addendum to SemVer: System-on-Chip Designs

Overview

SoC (System-on-Chip) designs are similar to software in many ways, e.g. they are written in human-readable computer languages and often developed using version control, but differ in what comprises their public Application Programming Interface (API). This is an addendum to the SemVer 2.0.0 specification which clarifies how to apply SemVer to SoC designs. It is assumed that you have read and understood:

- RFC 2119
- SemVer 2.0.0

In C software libraries, an API includes paths of header files, values of constants, names of exposed functions, and anything else which a library user might reasonably rely on. In command-line applications, an API includes the names of command-line options and their default values, precedence of configuration files, and anything else which an application user might reasonably rely on. Those examples of public APIs in software demonstrate that a public API can be described more generally as *anything which a user might reasonably rely on.*

SoC designs are typically written in specialized languages for digital logic such as Verilog (IEEE Std 1364), SystemVerilog (IEEE Std 1800), or VHDL (IEEE Std 1076) which facilitate synthesis to physical digital logic circuits. Different from software, SoC designs have users with fundamentally different requirements related to higher-level designs, high-level software, and physical implementation. These downstream users likely have differing perspectives about what constitutes the most important part of the public API – A software user might depend on the address and reset value of a register, but not depend on the hierarchical path to the corresponding FF (flip-flop) because that does not affect their software. In contrast, a user working on physical implementation might see the register address and reset value as trivial details, but depend on the hierarchical path of the FF to ensure that it is implemented with the correct type of cell.

The public API is restricted to the reasonable ways that users are expected to use a release. In-house projects may use this restriction to avoid incrementing MAJOR too often, i.e. the distinction between a breaking change and a bugfix can be redefined if you (1) *identify **all** downstream projects/users* and (2) *obtain explicit agreement from **all** users.* This exemption allows large subsystem and chip-level projects to make arbitrary changes under MINOR increments while reserving MAJOR increments for project-specific milestones. Only in-house projects may use this exemption because publicly available projects cannot identify all downstream projects/users.

Downstream Users and Auxiliary Components

Downstream projects, i.e. those which depend on your SoC design(s), usually fall into these categories:

1. Documentation: Describe the intention, features, operation, and limitations of your design as required by other users. The extent of overlap with other categories is highly specific to each particular project.
2. Integration: Include your design as a hierarchical component in a larger system. In SystemVerilog, this means declaring an instance of your `module` with suitable connections to parameter and signal ports.
3. Verification: Check that your design meets specifications. Includes both dynamic and static methods, and checks may be written in a different language from your SoC design, e.g. TCL, SystemC, or Python.
4. Physical implementation: Convert your abstract design into a concrete realization on a physical ASIC or FPGA platform. Includes synthesis, layout, and modifications for testing.
5. Software: Most modern SoC designs feature some programmable components which view your system from a memory-mapped perspective. Includes system-level tests (overlapping with verification), validation and characterization (overlapping with implementation), firmware, and possibly end-user applications.

In addition to the main description of digital logic, e.g. a collection of SystemVerilog files, a SoC design will usually include auxiliary components which may (or may not) correspond to additional files.

- Filesystem structure of the release delivery and associated filelists.
- Standards which the source code adheres to. Includes standards of all types from file encoding (e.g. ASCII vs UTF-8) and whitespace/formatting rules to the specification language (e.g. Verilog vs SystemVerilog) and naming conventions. Other users may have flows which extract information which depend on these seemingly minor details.
- Hierarchical paths and specific identifiers (names). In verification, specific FFs may be forced to obtain coverage on a difficult-to-reach state. In physical implementation, specific FFs may be selected as requiring special treatment to meet timing.
- Constraints on clocks, timing, pin-mapping, cell placement, routing, etc.
- Scripts for design modification flows such as DFT, instrumentation, optimization, obfuscation, etc.
- Waivers on errors or warnings from particular tools.
- Power intent, normally specified with UPF (IEEE Std 1801).
- Abstract models and unit tests. Other users (likely excluding implementation) may depend on these to validate their own work.
- Address map and structure of software-visible registers. All types of software may depend on specific addresses, the layout of register fields, and more subtle attributes like reset values and volatility.

When releasing a new version of a SoC design, it is important to consider how changes to all components of the release will affect all users developing downstream projects.

Changes in SystemVerilog

An illustrative example, shown in SystemVerilog, is useful to demonstrate API components of a typical SoC peripheral where sequential logic is implemented with D-type FFs. Let's say that our module `Alu` performs arithmetic operations on its inputs, drives known values on its outputs, and provides register access via the AMBA APB protocol. In the most recently released version, there is one configuration register called `CFG` at the address `12'h444`, with a reset value of `32'd5`, arranged as two fields `CFG[2:1]=OPERATION` and `CFG[0]=ENABLE`.

```
module Alu
  #(parameter int RESULT_W = 16
  )
  ( input  var logic [1:0][7:0]     i_operands
  , output var logic [RESULT_W-1:0] o_resultant
  , APB.slave                       ifc_APB
  );

  localparam bit MYCONSTANT = 1'b1;

  // To be combinatorially assigned via `always_comb`, `assign`,
  // or connection to sub-module.
  logic foo_d;

  // To be sequentially assigned via `always_ff`.
  logic foo_q;

  // ... snip ...

  ArithmeticPipe u_pipeA1
    ( .i_opA  (i_operands[3:0])
    , .i_opB  (i_operands[7:4])
    , .o_taps (foo_d)
    );

  always_ff @(posedge ifc_APB.clk, posedge ifc_APB.arst)
    if (ifc_APB.arst)
      foo_q <= '0;
    else if (updateFoo)
      foo_q <= foo_d;

  // ... snip ...

endmodule
```

The public API of this example module consists of the module declaration, APB registers, hierarchical paths to sequential elements, and other packaged components like helper scripts and design constraints.

MAJOR Versions

Given a version number MAJOR.MINOR.PATCH, increment the:

1. MAJOR version when you make incompatible API changes

Referencing the example, the MAJOR version must be incremented with any of the following changes:

1. Modified module name which integrators use to declare an instance of the peripheral, e.g. `Alu` → `MyArithmetic`. Existing code using the name `Alu` will not elaborate unchanged.
2. Removed parameter port, e.g. ~~`RESULT_W`~~. Existing code overriding the parameter value will not elaborate unchanged.
3. Modified parameter port kind, e.g. `parameter` → `localparam`, i.e. overridable to non-overridable. Existing code overriding the parameter value will not elaborate unchanged.
4. Modified parameter port name, e.g. `RESULT_W` → `OUT_WIDTH`. Existing code using the name `RESULT_W` will not elaborate unchanged.
5. Modified parameter port default value, e.g. `16` → `5`, including addition or removal of the explicit default value. Existing code may depend on the default value for critical functionality.
6. Removed signal port, e.g. ~~`o_resultant`~~. Existing code using that port will not elaborate unchanged.
7. Modified signal port datatype, e.g. `logic [1:0][7:0]` → `logic [15:0]`. Existing code may depend on the size and structure of the port datatype, and input expressions may be cast to an unexpected width or datatype.
8. Modified signal port name, e.g. `i_operands` → `i_numbers`. Existing code using the name `i_operands` will not elaborate unchanged.
9. Removed interface port, e.g. ~~`ifc_APB`~~. Existing code using the APB interface will not elaborate unchanged.
10. Modified interface port type, e.g. `APB.slave` → `AXI.slave`. Existing code using the APB interface will not elaborate unchanged.
11. Modified interface port name, e.g. `ifc_APB` → `myApb`. Existing code using the name `ifc_APB` will not elaborate unchanged.
12. Removed or modified sequential signal name, e.g. `foo_q` → `bar_q`. Existing code referencing `foo_q` will not find the inferred FF(s). You may not notice the breakage until your colleagues in physical implementation notify you that their scripts don't work. In the worst cases, FFs requiring special treatment can be silently missed.
13. Any added, removed, or renamed hierarchical middle layer, e.g. `Alu.u_pipe` → `Alu.u_wrapperA.u_pipe`. Existing code, particularly for physical implementation, may depend on the hierarchical names including generate blocks.
14. Removed, or renamed hierarchical bottom layer, e.g. `Alu.u_pipe1` → `Alu.u_pipe[1]`. Existing code, particularly for physical implementation, may depend on the hierarchical names including generate loops.
15. Removed or functionally modified package function. Higher-level designs may depend on any aspect of a function in a supplied package at elaboration time, simulation time, and in synthesis.
16. Added, removed, or modified any machine-readable comment, e.g. tool-

specific directives like `// synopsys parallel_case`. Existing flows are likely to depend on these for critical functionality.

17. Removed software-accessible register, e.g. ~~`CFG`~~. Existing system software accessing the `CFG` address will not operate equivalently.
18. Modified software-accessible register address, e.g. `12'h444` → `12'h888`. Existing system software accessing the address `0x444` will not operate equivalently.
19. Modified software-accessible register field layout, e.g. `CFG[0]=ENABLE` → `CFG[31]=ENABLE`. Existing system software accessing the register will not operate equivalently.
20. Modified software-accessible register reset value, e.g. `32'd5` → `32'd0`. Existing system software accessing the register will not operate equivalently, particularly software performing non-atomic read-modify-write operations on startup like `cfg->operation++`.

To summarize, the MAJOR version must be incremented with any changes which *require* updates to any projects that fetch the newly released version. Note, changes to match the documentation of a previous release should be considered bug fixes, so may only warrant a MINOR increment.

MINOR Versions

> Given a version number MAJOR.MINOR.PATCH, increment the:
>
> 2. MINOR version when you add functionality in a backwards compatible manner

Where SemVer specifies adding functionality, SoC designs must update *at least* the MINOR version with any of the following modifications:

1. Added parameter port, e.g. `ANOTHER`. Existing code will elaborate unchanged.
2. Modified parameter port datatype, e.g. `int` to `bit [3:0]`, including removal of the explicit datatype. Existing code may elaborate unchanged, but override values may be cast to an unexpected width or datatype. If existing code needs changes to elaborate with the updated version, then increment MAJOR instead.
3. Added signal port, e.g. `output o_another`. Existing code may elaborate unchanged and a new signal port implies new functionality.
4. Modified signal port direction, e.g. `inout myport` → `output myport`. Existing code may elaborate unchanged, but simulation semantics may be different. If existing code needs changes to elaborate with the updated version, then increment MAJOR instead.
5. Modified signal port nettype, e.g. `input logic` → `input var logic`. Default nettype of `input` and `inout` signal ports with datatype `logic` is `tri`, but for `output` ports it's `var`. Existing code may elaborate unchanged, but simulation semantics may be different. If existing code needs changes to elaborate with the updated version, then increment MAJOR instead.
6. Added interface port, e.g. `OCP.slave`. Existing code may elaborate unchanged and a new interface port implies new functionality.

7. Modified sequential signal datatype or expression, e.g. `logic [1:0] foo_q` → `FooEnum_t foo_q`. Backwards-compatible changes only require a MINOR increment, but incompatible changes like reducing the *intended* width of a FF vector require a MAJOR increment.
8. Added hierarchical bottom layer, e.g. `Alu.u_pipeA2`. New hierarchy implies new functionality, not just a bug fix.
9. Added package function, e.g. `function bit isPrime (int x);`. A new function implies new functionality, not just a bug fix.
10. Added software-accessible register, e.g. `STATUS`. Existing system software will not operate equivalently, and updated software may use the new functionality.

To summarize, the MINOR version must be incremented with any changes which add or modify functionality in a manner which *does not require* downstream users to make changes. If downstream users are required to make changes to their project in order to accept the new version, increment MAJOR instead.

PATCH Versions

> Given a version number MAJOR.MINOR.PATCH, increment the:
>
> 3. PATCH version when you make backwards compatible bug fixes

As with SemVer, only backwards-compatible changes (for all downstream users) are allowed within a PATCH increment version.

1. Added, removed, or modified internal constant, e.g. `MYCONSTANT` → `BETTERNAME`. Internal constants should not be relied upon downstream.
2. Added, removed, or modified internal combinational signal, e.g. `foo_d` → `bar_d`. Internal combinational signals should not be relied upon downstream. Exemption: If you change signals which are *intended* to be probed or forced by downstream users, increment MAJOR instead, e.g. `disableChecks` → `turnOffChecks`.
3. Added internal sequential signal, e.g. `new_q`. Additional FFs will affect area, power, achievable fmax and cost, but are unlikely to break physical implementation flows outright. Note, removed or renamed internal signals require a MAJOR increment.
4. Non-functionally modified package function, e.g. to improve simulation performance without affecting its domain, codomain, or image.
5. Any tag comment, e.g. `/* TODO: Something */`. Note, where changes include updates to tag comments, there's a good chance the changes also involve enough to warrant a MINOR or MAJOR increment.
6. Any human-only comment, e.g. `/* Isn't this nice */`.

SemVer for SystemVerilog Cheatsheet

How	What	Increment (at least)
mod	Top-level module name	MAJOR
add	Parameter port	MINOR
rem	Parameter port	MAJOR
mod	Parameter port kind	MAJOR
mod	Parameter port datatype	MINOR
mod	Parameter port name	MAJOR
mod	Parameter port default value	MAJOR
add	Signal port	MINOR
rem	Signal port	MAJOR
mod	Signal port direction	MINOR
mod	Signal port nettype	MINOR
mod	Signal port datatype	MAJOR
mod	Signal port name	MAJOR
add	Interface port	MINOR
rem	Interface port	MAJOR
mod	Interface port type	MAJOR
mod	Interface port name	MAJOR
any	Internal constant	PATCH
any	Combinatorial signal	PATCH
add	Sequential signal	PATCH
rem	Sequential signal	MAJOR
mod	Sequential signal name	MAJOR
mod	Sequential signal datatype	MINOR
mod	Sequential signal expression	MINOR
any	Hierarchy middle layer	MAJOR
add	Hierarchy bottom layer	MINOR
mod	Hierarchy bottom layer	MAJOR
rem	Hierarchy bottom layer	MAJOR
add	Package function	MINOR
rem	Package function	MAJOR
mod	Package function behavior	PATCH
any	Tool directive comment	MAJOR
any	Tag comment	PATCH
any	Human-only comment	PATCH
add	Software register	MINOR
rem	Software register	MAJOR
mod	Software register address	MAJOR
mod	Software register field layout	MAJOR
mod	Software register reset value	MAJOR

FAQ

1 - I don't want to increment MAJOR every time I do a little refactoring. Should internal signal names really be considered part of the public API?

A common point of confusion is that SoC design is not the same as software. The SoC design process (at the RTL level) sits somewhere between the design processes for software and high-volume hardware like injection moulds or PCBs. High-volume manufacturing processes require extreme rigor in even the tiniest change because a subtle unnoticed mistake can quickly cost a huge amount of wasted resources. In general, software development processes are considerably more relaxed because fixes can be deployed relatively easily to downstream users.

The fundamental difference is that software can be safely divided into discrete layers of abstraction, i.e. a web developer writing a JavaScript function has little need to care about the exact sequence of CPU instructions used to execute it. This is because the software ecosystem has such well-defined abstractions that JavaScript's environment perfectly follows a set of rules based on digital Boolean logic, so the effects of any change can be fully predicted and contained. Physical hardware is much more complex. For example, let's say a PCB designer wishes to move a capacitor; Electrically, there are no changes so verification could seem trivial, however, downstream users might notice that the change affects EMI compliance, heat distribution, weight distribution, manufacturability, and mechanical dimensions that they depend on. The physical world is considerably more difficult, perhaps impossible, to partition into neat layers of abstraction, so designers of physical products need to have awareness of a much wider range of possible effects on downstream users.

SoC designs in RTL are typically close to physical hardware so downstream users working on synthesis and layout will frequently (and necessarily) depend on things like the names of internal signals, which might be unexpected to engineers from software backgrounds. These differences are also reflected in RTL languages: For example, SystemVerilog does not have keywords like `public` or `private`. The closest thing is `protected` which only applies to `class`es, i.e. only in the context of non-synthesizable code.

If this all seems too restrictive, there are a few points worth clarifying:

1. SemVer item 4 notes that MAJOR version zero is for initial development, so anything may change between MINOR and PATCH increments, and the public API *should not* be condsidered stable. If you're still in the stage of development where refactoring and renaming changes are frequent, then the design isn't yet stable so just keep the MAJOR at zero until you're satisfied that the design doesn't need regular "tidying".
2. SemVer has FAQ points (here, here, and here) noting that part of your responsibility (to your users) is to communicate effectively about changes which might cause them unexpected and unplanned work.
3. If you're adamant that signal names are not part of the public API, it is sufficient to simply state this in documentation and/or release notes. For example, "Synthesis scripts should not depend on any signal matching the regular expression `.*_d`".

2 - My design depends on other designs which use SemVer. Do I need to increment MAJOR for every dependency update?

- If your changes pull in an incompatible dependency, i.e. it's MAJOR has increased, then yes because that incompatibility can break things for your users.
- If you're using a version control like git, you don't need to make a release for each depedency individually, just like you wouldn't make a release from every commit. As a responsible developer, your releases should be planned and the effects of any changes should be understood.
- Similar to the previous question, you can class any part of your design as private simply by stating so in documentation and/or release notes, but the default stance for SoC designs is that everything is considered public.

3 - In my project, we plan our releases such that MAJOR indicates architectural updates, MINOR indicates new features, and PATCH indicates bugfixes. How do these rules apply?

It's great to have a well-written plan so that users know what key features to expect as your project progresses, and it's nice that you provide a consistent way of communicating project progress. However, that is not Semantic Versioning.

Semantic Versioning is a scheme for communicating functional compatibility between releases, not about communicating project progress. Alternative schemes may appear cosmetically similar to SemVer (by using the format `X.Y.Z`) but not adhere to the same rules. By stating that your project uses SemVer, you're making a set of promises to your downstream users, most notably that they should be able to pull in MINOR and PATCH increments without *any* integration work.

These are some of the things that SemVer numbers do *not* convey:

- How the release corresponds to a project plan.
- The quality or completeness of any aspect.
- What features are included or excluded.
- How much effort was involved, e.g. the number of man-months.
- The size of the changes, e.g. the number of changed lines.
- How much smaller/faster/neater/better the new release is.
- Whether the release is stable for one set of users, e.g. verification, versus another set of users, e.g. physical implementation.

Adhering to SemVer only communicates, very coarsely, how much work is mandatory for downstream users moving from one version to another.

4 - My project wants to make regular releases from parallel branches of development. How can we use SemVer?

The most important thing is to communicate clearly and precisely what that means to your users, otherwise they might see their flows unexpectedly breaking and get annoyed with you. If the development branches diverge significantly, you can consider splitting the project into a "core" and higher-level "variants" which include the core.