6.1 Traditional Design Process

The traditional front-end design process consists of the following steps for each design or IP: 1) **A requirement specification** (with potentially some algorithms in a high-level language) written in a natural language (e.g., English); 2) **An architectural plan** that represents the high level implementation approach; 3) **A verification and test plan** that addresses the testbench and verification approach and classes of tests; 4) **The RTL design and synthesis**; 5) The **design verification** with details about the testbench and automatic verification; and 6) **The final design documentation and delivery**.

The traditional design process relies too much on a natural language, such as English, as a mean of communication for the definition of requirements and verification and test approaches; that can often lead to several misunderstandings. In addition, that process does not use an executable, or provable method, to characterize design requirements and restrictions.

6.2 Design Process with SVA

Figure 6.2-1 represents a typical design process with SVA (*also see Figure 7.1-1 for the typical verification design flow)*. All designs must have requirements documentation. The requirements can be classified as system-level and module-level. The requirements document can be supplemented with SystemVerilog Assertions to avoid ambiguities caused by a natural language description.

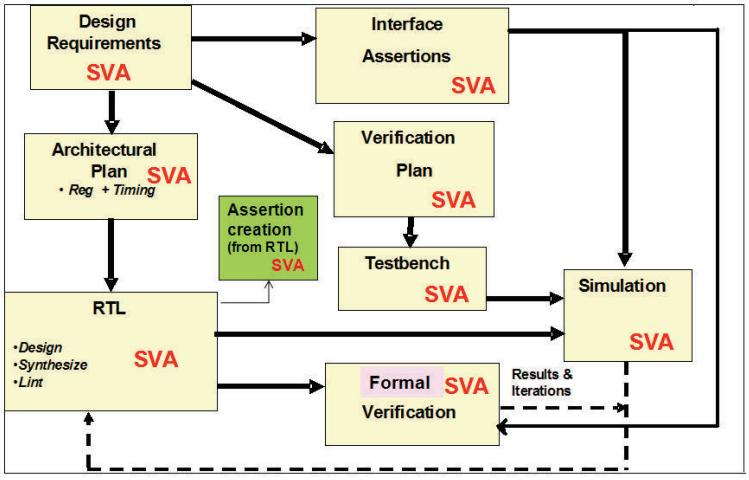

Figure 6.2-1 Typical Design Process with ABV Using SVA

6.2.1 System-level Assertions

SystemVerilog properties can be used to capture system/subsystem level requirements, and those properties can then be reused in a testbench environment, later in the design cycle. System-level requirements can have the following classifications:

- **Functional**: This represents the purpose of the design.
- **Performance:** That consists of items such as baud-rate, frequency, capacity, speed, throughput, and latencies.
- **Interface:** Since the design under consideration may interface to existing busses, and connections to other subsystems, the requirements document needs to include software, hardware, and communications interfaces. That should also include expected modes of operations and restrictions (e.g., parity type, frame size, master, slave).
- **Safety:** Many designs are subject to security and safety issues to insure safe operations under certain conditions or failures (e.g., during reset, power down, power-up, hot plugging, bus or unit failure).

- **Operational:** Some designs are subject to user interfaces for commands, configurations, or maintenance.
- **Resource:** Resources are requirements that are critical to a design, and include items such as power, memory, form-fit factor (i.e., size), etc.

The performance, interface, and safety classifications can be expressed in SystemVerilog Assertions using libraries defined in checkers or in modules. It is important to note that requirements are independent of the architecture and implementation. However, the architecture and implementations are a result of the requirements.
The following sub-sections demonstrate with a few examples, how SystemVerilog Assertions can be used in the definition of system-level requirements.

6.3 Requirements

Requirements are typically defined in a document (see Section 6.3 for an example of such a document). The "design requirements" should be free of implementation. The following subsections address types of requirements that can be supported by SVA to clarify the intent.

6.3.1 Cause and effect class of requirements

Many system-level requirements can usually be represented as cause-effect relationships. In SystemVerilog, the "cause" is called the antecedent, and the "effect" is called the consequent. For example:
Requirement: If the design is subjected to a command to fire the *pyro X* then within 5 cycles the *pyro X* relay shall be activated.

SVA property definition:

```
property pFirePyro;  // use of the clock is important only if it adds significance
  @ (posedge clk) xaction.Fire_PyroX_CMD==FIRE  |->
                    ## [1:5]  pyro_sub.PyroX_relay==ACTIVATED;
endproperty : pFirePyro
```

When properties of the requirements are expressed in a precise and an easy to read manner, such as with SVA, the review process of those properties tends to bring out important issues that may not have surfaced. For example, for this property safety is an important issue as it would be inappropriate to have an early misfire. With added safety, one could enrich this property with safety preconditions:

```
property pFirePyroSafe;
          @ (posedge clk) xaction.Fire_PyroX_CMD==FIRE &&
           pyro_sub.power_stable  &&
           pyro_sub.reset_cycle== DONE &&
          pyro_sub.armed == OK  |->
                ##[1:5] (pyro_sub.PyroX_relay==ACTIVATED;
endproperty : pFirePyroSafe
```

The *xaction* is a definition of the origin of the command. It may be a CPU command that can be emulated in a testbench by a *transactor* module. The *Fire_PyroX_CMD* is an identification of the instruction used to identify the command in a *transactor* testbench model. *FIRE, DONE*, and *OK* are states of various conditions. *ACTIVATED* is a state of the relay, and *pyro_sub* is the *Pyro* submodule. The *pFirePyroSafe* property states that if there is a command to fire the pyrotechnics, and the submodule power is stable, and the reset cycle is completed, and the pyrotechnics hardware is armed, then within 5 cycles the pyrotechnics relay must be activated.

6.3.2 Latencies

Latencies are critical in a requirements document since they identify a response time. For example:
Requirement: A 256 word *AHB_bus* transfer clocked with the AHB clock shall be relayed onto the *PCI_bus* within 100 to 500 cycles of the PCI clock

SVA property definition:

```
property pAhb2PCIxfr;
  @ (posedge ahb_clk)  (xaction.send256to_pci) |=>
           @ (posedge pci_clk)  ## [100:500] (pci_start_xfr);
endproperty : pAhb2PCIxfr
```

The above property states that after the command *send256to_pci* from an AHB bus interface synchronized to the *ahb_clk*, there must be a start of data transfer (signal *pci_start_xfr*) within 100 to 500 cycles, synchronized to the *pci_clk*.

6.3.3 Definition of Processing Algorithms

Processing algorithms are often specified in a requirements document with the algorithm captured as pseudo-code in a modeling language such as SystemC. For example:

Requirement: When the CPU commands a *Filter* operation, the image processing subsystem shall perform the *Filter* algorithm on the loaded image, as described by the function *Filter*, identified in the requirements document. The *Filter* operation on a frame size of 384x288 pixels shall be performed in less than 1000 cycles. Figure 6.2.1.3 is a view of a required image processing.

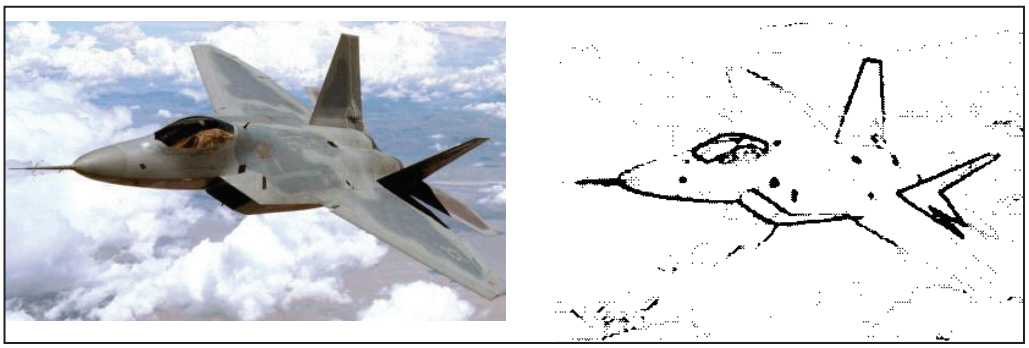

Figure 6.2.1.3 Sample Image Processing

SVA property definition:

```
property pFilterProcessing;
   xaction.do_Filter_cmd && image_processor.scene_loaded |->
        ## [*0:1000]  image_processor.done_Filter;
endproperty : pFilterProcessing
property pFilterProcessingCheck;
       image_processor.done_Filter  |->
            image_processor.Filter_scene ==  Filter(xaction.sent_scene);
endproperty : pFilterProcessingCheck
property pHotPointResults;
      xaction.get_Filter_hotpoints_cmd |-> ## [0:100]
           Filter_compare (results.hotpoints_array,
                           Filter_hotpoints(xaction.sent_scene));
endproperty : pHotPointResults
```

In these property descriptions, *image_processor* represents an image processor subsystem; *Filter* represents a function; *sent_scene* is an image to be submitted for processing that was sent to the memory; *Filter_scene* is the result of the Filter operation on the image; *Filter_hotpoints* is another function that identifies the compression algorithm of the detected hot points from the scene; *Filter_compare* is a function that compares array objects. Note that these functions can be written in C, as the SystemVerilog Direct Programming Interface (DPI) allows direct inter-language function calls between SystemVerilog and any foreign programming language with a C function call protocol and linking model. Functions implemented in C and made available with the `import` declarations in SystemVerilog can be called from SystemVerilog; such functions are referred to as imported functions.

The property *pFilterProcessing* states that if there is a filter processing command, and the image scene is loaded into the image processor memory, then within 1000 cycles, the image processor completes the filter algorithm on that image. The property *pFilterProcessingCheck* compares the actual results of the filter operation performed on the image processor against the expected results, per the algorithm defined in the requirements document. The property *pHotPointResults* states that upon the initiation of a command to read the hot points, the results of another algorithm are compared against expected results.

6.3.4 Interface Assertions

Interface assertions represent in a formal manner the properties of the interfaces and black-box protocols. The assertions of a common interface should be grouped in a `checker` to facilitate reuse (see Chapter 5). These assertions not only help in the documentation, understanding, and clarification of the interfaces, but also provide a verification mechanism of the design at the I/O interface level. Ideally, those properties need to be defined by a verification engineer from the requirement documents, rather than by the RTL designer. This avoids possible misunderstanding of the requirements made during the implementation. However any unique assumptions (e.g. use of a subset of the interface) should be captured as interface assumptions. While integrating the different blocks of RTL, these block-level assumptions can act as "guardian" for the individual block's behavior. Another very important aspect to be considered while dealing with interfaces is the interface functional coverage as that ensures that various sequences and properties are tested during simulation. An added benefit of adding interface assertions is that in a system-level simulation, they speed up debug of any failing simulations as they tend to be the closest to the block boundary where the problem originates (assuming that the problem is due to an interface protocol related issue).[49] Section 6.3 provides a complete SystemVerilog FIFO interface example with assertions.[50]

6.4 Architectural Plan

Architectures represent not only a top-level basis of the design, but also information about the registers, interface cycle timing, design assumptions, and restrictions, along with rationale for those decisions.[51] The architectural plan is the forefront to implementation and verification of the design.

The sequences, properties, and assertions statements clarify architectural issues and design assumptions. The SystemVerilog properties and assertions are very useful for reviews, and for the engineers who will implement and verify the design. Any property and assertion code written during the architecture planning stage can be reused during the verification phase of the design. Advanced ABV methodology recommends that any register definition shall also contain necessary functional coverage requirements on the individual fields and the cross of various fields (the individual fields may be spread across various registers). The cross coverage can be done with assertions or with SystemVerilog covergroups (see 5.6.3 for an example).

---

[49] Prior to the popularity of assertions, the authors have experienced cases where several interface errors were carried all the way through chip manufacturing. These errors ranged from something as trivial as the polarity of an interface, to misunderstandings in the protocols. Assertions based on requirements, and followed through by designers to the RTL and testbench phases could have avoided those errors.

[50] Also available in the downloadable files /ch6/*.sv.

[51] The use of term "Architecture" in this section is synonymous to "Micro-Architecture" used by design teams.

6.5 Verification and test Plan

The terms "verification" and "test" are often interchanged because they both deal with the concept of checking that the item in question is operational as intended. However, those two terms have different connotations as verification deals with the "what" to verify, and test deals with the "how" to implement the verification. Thus,

- **A verification plan** addresses the items to be verified, but without addressing the methodologies. For example, a verification plan for a CPU will address that the items to be verified include the ISA, the IOs, environment (e.g., ISA mix, memory types (fast/slow), application software written in X language, etc).
- **A Test plan** addresses how the items that need to be verified will be checked. For the CPU example, the test methodology may include simulation, emulation, use of assertions, use of UVM, constrained random tests, types of mixes, test application code, tools, instruments, etc.

The verification plan is a specification for the verification effort. It provides a strawman document that can be used by the design community to identify, early in the project, what needs to be verified. Early mistakes in the verification approach can be identified and corrected. A byproduct of the verification plan exercise is the revisit of the requirements. This enforces the process of verifying those requirements, thus helping in the identification of poorly specified or ambiguous requirements.

A test plan is a document that defines the following:

1. **The verification technologies**. The plan identifies the verification technologies that will be used for the project. These technologies include assertions, verification libraries, functional coverage, cross coverage, linting, code coverage, frameworks (e.g., UVM), simulators, emulators, formal verification, and tools. It should also identify how these technologies are used. For example, formal verification may be used at the subblock level, while simulators may be used at the chip level, and emulators may be used at the system / software verification level.

2. **Verification environment for the design-under-test**. This includes the structure of the testbench, and special instructions. The structure encompasses the component models (at the interface level), packages (at the declaration or higher level), and file structures. The verification environment will also include verification units to insure that the actual results produced by simulation of the design meet the expected results.

3. **Tests or transactions applied to the design**. These tests are used to verify the design's functional correctness as specified in the requirements specification. This includes tests at the top-level of the design as well as the subblocks. SystemVerilog Assertions can be used to specify assumptions about inputs, and transactions at the interfaces, along with expected results within a range of cycles, to allow for variations in the DUT cycle timing. Many of these transactions can be extracted from the requirements documents (system and architecture). A current trend in verification is to automatically generate stimulus/tests using the SystemVerilog Assertions and assumptions as a base for the definition of the constraints.

4. **Exit criteria**. These criteria identify when verification is complete, or at least achieved the goals. They may include code and functional coverage, as well as passing all tests and lint checks, etc. Code coverage may include line, branch, condition, toggle, path and FSM.[52] Functional coverage represents a user-defined model of functionalities of the design that the verification process should address. SystemVerilog has a rich set of constructs to capture the coverage model.

[52] Verification Methodology Manual http://vmmcentral.com/