# CYBERWORKBENCH

NEC’s High Level Synthesis Solution

SEPTEMBER 01, 2016

NEC INDIA PVT LTD
[Company address]

# Contents

# Introduction

The design productivity gap problem is becoming more and more serious as VLSI systems become larger. In the mid-1980s, gate-level design shifted to register transfer level (RTL) design for designs that typically exceeded 100K gates.

Currently, several million gates circuits are commonly used just for random logic parts of a design, which equate to more than several hundred thousand lines of RTL code. It is therefore needed to move the design abstraction one more level in order to cope with this increasing complexity. Behavioral synthesis is a logical way to go as it allows “less detailed design description” and “higher reusability”.

A higher level of abstraction description requires smaller code and provides faster simulation time. For example a one million gates circuit requires about 300K lines of RTL (Verilog or VHDL) code, but only around 40K lines of C code. The RTL simulation of 300K lines, is on an average takes 10 to 100 times than the 40K lines of equivalent behavioral code.

The benefits of behavioral synthesis are palpable through multiple commercial chip successes, thus Behavior Synthesis, or High Level Synthesis, is gaining acceptance within the design community. Various commercial chips for printers, mobile phones, set-top boxes and digital cameras are being designed using behavioral synthesis these days. ANSI-C is the preferred programming language for behavioral synthesis because embedded software is often described in C and design tools like compilers, debuggers, libraries and editors are easily available and there is a big amount of legacy code.

Presenting here an overview of C-based design flow where the efficiency and simulation performance is compared against pure RTL with co-simulating it with embedded software. C-based behavioral IPs are advantageous over RTL IPs and the application specific processors can be benefited from it. The hardware architecture explorer at the behavioral level allowing a fast and easy way to study the area, performance and power trade-offs of different designs automatically.
This document covers how behavioral synthesis can be used for any hardware module (data and control intensive).

## C-Based Design Flow

NEC have been developing C-based behavioral synthesis called “Cyber” since the late 80’s and developing C-based verification tools such as formal verification and simulation around Cyber during the last 20 years  All these tools are integrated into an IDE, where designers execute these tools upon the C-source code. The name of this IDE tool suite “CyberWorkBench”(CWB).

## Basic Concept of CyberWorkBench

The main idea behind CyberWorkBench is an “all-in-C” approach. This is built around two principal ideas:

1. **“All-in-C Synthesis”**: means that all modules in a VLSI design, including control intensive circuits and data dominant circuits, should be described in behavioral C language. The system supports legacy RTL or gate net list blocks as black boxes, which are called as C functions. At the same time it allows designers to create all new parts in C.

2. **“All-in-C Verification”:** means that Verification (including debugging) tasks should be done at the C source code. In behavioral synthesis, a designer should not have to debug the generated RTL code. The CWB environment allows a designer to debug the original C source code and the CWB model checker allows designer to write properties or assertions directly on the C source code.

## Design Tool Overview

CWB targets general LSI systems which normally contain several CPUs or DSPs, dedicated hardware modules and some pre-designed or fixed RTL-or gate level IP modules, which are directly connected or through buses.

Initially, each dedicated hardware module is described in behavioral C. Once its functionality is verified using the C simulator and debugger, the hardware module is synthesized with the behavioral synthesizer. Configurable processors are also synthesized from the C description in CWB environment. Legend RTL modules are described as function, and handled as a black box.

The CPU bus and bus interface circuits are automatically generated using a CPU bus library. After synthesizing and verifying each hardware module, the design environment allows designers to create a cycles-accurate simulation model for the entire system including CPUs, DSPs and custom hardware modules. With this simulation model, designers can verify both functionality and performance of their hardware design as well as the embedded software run on the CPU, DSP and/or generated configurable processors. The behavioral C source code can also be debugged with the formal verification, property/assertion model checker tool. Global properties and in-context (immediate) assertions are described for/in the C source code. The equivalence between behavioral C and generated RTL can be verified both in dynamic and static way.

## *Synthesis Flow*

The CWB design flow is shown in Figure 1-1. A hardware design in extended ANSI-C (called "BDL", or "Cyber-C"), or SystemC is synthesized into synthesizable RTL with the "Cyber" behavioral synthesizer with a set of design constraints such as clock frequencies, number and kind of functional units and memories. Usually RTL is handles as a black box, but if necessary, the RTL can also be fed to behavioral synthesis. The behavioral synthesizer can insert extra registers to speed up the original RTL and generate new RTL of smaller delay. It also generates a cycle accurate simulation models in C++ or SystemC. The behavioral synthesis can therefore be considered as a Verilog, VHDL, C, C++, and SystemC unification step.

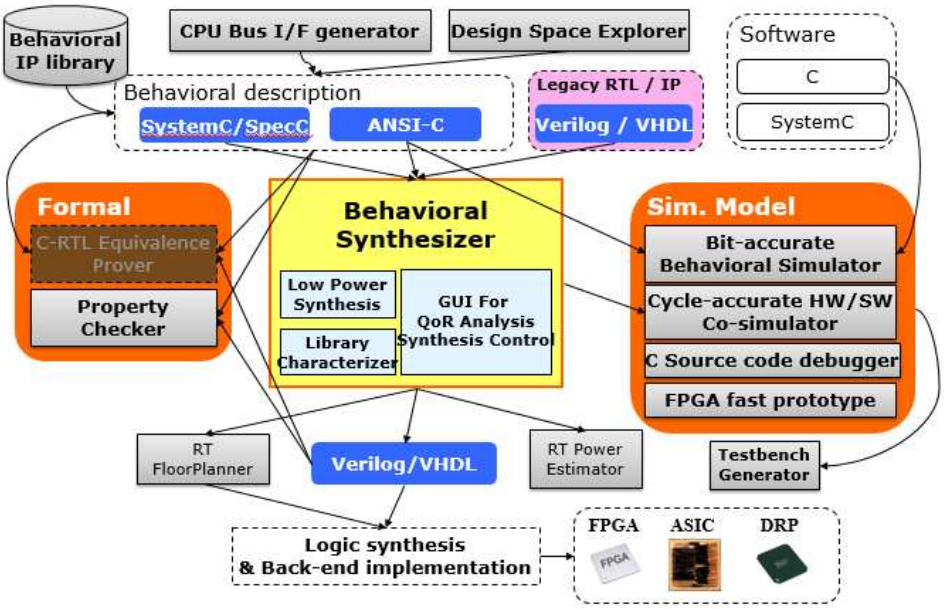


Fig 1-1. CyberWorkBench™ design flow

The “RTL Floor Planner” takes the RTL modules generated by the behavioral synthesizer with estimated area. Accurate timing information is extracted from the floor planner and fed back to the behavioral synthesizer. The behavioral synthesizer reads the timing information and re-schedules the C code considering this.

## *Verification Flow*

The functionality of the hardware described in C can be verified at the behavioral level, while performance and timing are verified at the cycle-accurate level (or RTL) through simulation. CWB has a behavioral C source code debugger linked to the cycle-accurate simulation and FPGA emulation tool. After verifying each hardware module, the entire SoC is simulated in order to analyze the performance and/or to find inter-modules problems such as low performance through bus collision, or inconsistent bit orders between modules. Since such entire chip performance simulation is extremely slow in RTL-based HW-SW co-simulation. CWB generates cycle accurate C++ simulation models which can run up to hundred times faster than RTL model. The HW-SW co-simulator, uses the generated cycle-accurate model for this purpose. The simulator allows designers to simulate and debug both hardware and software at the C source code level at the same time. If any performance problems are found, designers can change the hardware-software partitioning or algorithm directly at the C level, and can then repeat the entire chip simulation. This flow implies a much smaller and therefore faster re-design cycle than in a conventional RTL methodology. This entire chip simulation can be further accelerated using an FPGA emulation board. A “Testbench Generator” helps designers to run an RTL simulation with test patterns for behavioral C simulation faster and easier. Its inputs are test patterns for the C simulation and outputs a Verilog and/or VHDL testbench, which generates stimulus for the RTL simulation. It also creates a script to run commercial simulators to feed the behavioral test patterns and check the equivalence of outputs patterns between the behavioral and RTL simulation.

Another important feature of CWB is the formal verification tools, which is tightly linked to the behavioral synthesizer. With the behavioral synthesis information the formal verification tools can handle larger circuits than usual RTL tools and have C-source level debugging capability even though the model checker works on the generated RTL model. “C-RTL equivalence prover” checks the functional equivalence between a behavioral (un-timed or timed) C description and the generated RTL, using information on of the optimizations performed such as

loop unrolling, loop merge and array expansion performed by the behavioral synthesis. Without such information, the equivalence check is almost impossible for a large circuit.

Designers can specify assertions or properties at the behavioral C level, similar to the cycle accurate simulator. Such behavioral level properties/assertions are converted into RTL ones automatically, and are passed to our RTL model checker.

CWB generates a power enhanced RTL model which estimates the power consumed by the design. A set of power libraries for different technology is provided and used with the generated RTL estimates that power for the selected technology.
A “QoR” synthesis report file of the generated circuit shows a quick overview of the design quality. The report file includes area, number of states, critical path delay, number of wires and routability. This information is used for quick micro-architectural exploration as well as system architectural exploration. The system architecture explorer automatically generates different hardware architectures based on the preferences and constraints entered by the user (area, latency, power) at the C level. The designer can analyze the different generated architectures and finally choose the one that meets the design constraints at the smallest cost.

## Behavioral Synthesis

To support the “all-in-C” paradigm presented before, the behavioral synthesizer must cope with three types of circuits: (i) data-dominated, (ii) control-dominated, and (iii) control-flow intensive (CFI) ones. The three types of synthesis engines in order to support these varieties of circuit types: (i) automatic scheduling for CFI and data-flow circuits, (ii) fixed scheduling for control-dominated circuits, and (iii) pipeline scheduling for automatic pipelining or loop folding. Figure 1-2 shows a block diagram of CWB’s synthesizer engines.

Control dominated circuits such as PCI I/F, DMA controller, DRAM controller, Bus Bridge, etc., require cycle-by-cycle behavior description, which is fit for timing chart. The extended C language BDL can describe clock boundary in a behavioral description, and is able to express very complex timing behavior concisely. Such description is synthesized with “fixed scheduling” engine. For that circuits, which require fixed sequential communication protocols but all other computations can be freely scheduled, automatic scheduling engine is used for synthesis.

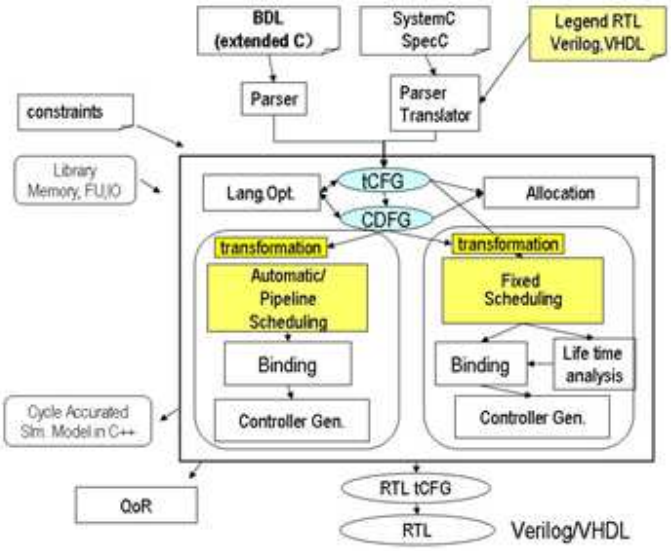


Fig 1-2. Configuration of Cyber Behavior Synthesis

For CFI circuit synthesis, scheduling and allocation techniques play a major role. The quality of synthesis is affected by the control flow structure. A smart scheduling algorithm must be designed to overcome the effects of the programming style. The scheduler will have to modify the control logic in order to obtain circuits with less latency while maintaining the data-flow intact.

Merging two branches into a single using CDFG transformation is not as effective because the procedure is complex and the merging does not always lead to better results. Thus, this approach uses a systematic scheduling algorithm without CDFG transformations. In other words, the scheduler schedules all operations in several basic blocks and several branches at the same time in a unique way, as if they were all operations in a single basic block. The approach handles many other types of speculations, global parallelization with a method called “Generalized Condition Vector”, which is extended version of “Condition Vector”

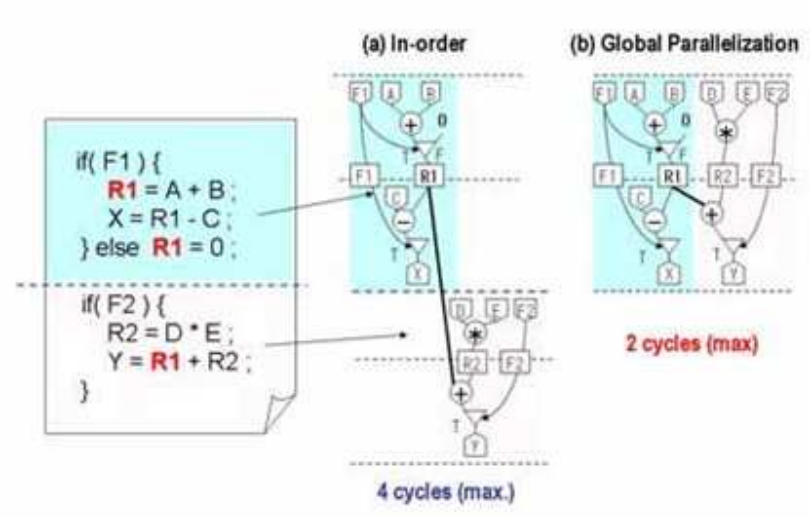


Fig 1-3. Parallelization of multiple branches for control-flow intensive applications (CFI)

## Behavioral Synthesis Advantages over Conventional Flows

The next subsections describes in detail some of the advantages of behavioral synthesis over conventional RTL methodologies like hardware-software co-design, source code re-usability, application specific processor optimizations and automatic architecture exploration.

### *Shorter Design Period and Less Design Cost*

Since C-based behavioral synthesis automates the functional design of hardware, it shortens the design cycle and at the same time shortens the design time of embedded software. Figure 1-4 shows the design cycle of two designs. The first uses the traditional RTL-based design flow and the second the proposed C-based design flow. The total design period and design men-month for the RTL-based design is larger than the C-based one, even though the gate size for RTL design (200K) is one third of that for the C-based (600K) one. The hardware design period of the C-based design is 1.5 months, much shorter than the RTL-based design which takes 7 months. It needs to be stressed that the software design in the C-based design takes only 2 months while it takes 6 months for the RTL-based. This is due to the fact that the embedded software can be debugged before the IC fabrication using the hardware-software co-simulator. In RTL design, the software is usually verified on the evaluation board since RTL co-simulation is too slow even for this size of circuits. Lastly, C-based design allows very quick generation of simulation models for embedded software at a very early stage, allowing hardware and software to be concurrently designed both in C.

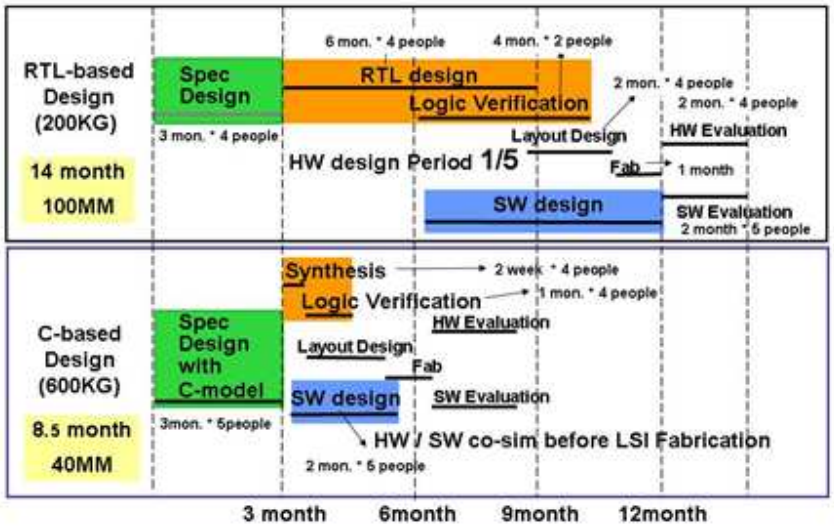


Fig 1-4. Comparison of Design Periods with C-based and RTL- based Design

## *Source Code Reusability and Behavioral IPs*

Another important aspect of CWB is the high-reusability of behavioral models, we call this as “behavioral IP” or “Cyberware”. An RT level reusable module, called “RTL-IP”, can be successfully used for circuits of fixed performance such as bus interface circuits. However, RTL-IPs for general functional circuits such as encryption can only use for a specific technology, since the RTL-IP’s “performance” is hard to adapt for newer technologies. On the contrary, a behavioral IP is more flexible and more reusable than RTL-IPs, since it can change its structure and behavior allowing the synthesis tool can generate circuits of different performances by simply changing high level synthesis constraints such as number of functional units and clock frequencies. Table-1-1 shows how various circuits of different “clock-frequency” can be generated from a single behavioral IP. This IP is a BS broadcast descramblers (Multi2). All generated circuits satisfy the required performance (more than 80Mbps) at various frequencies. Note that the highest clock circuit (108 MHz) uses less number of gates than the slow circuit (33MHz). This never happens in RTL-IPs, which follow the area-delay tradeoff relation of logic synthesis. A behavioral synthesizer generates a smaller circuit of higher clock frequency for the same performance, since less parallel operations are necessary to achieve the same performance at higher clock frequency.

Table 1-1. BS broadcast descrambler behavioral IP comparison

| Clock Frequency | Generated Gate size | Generated RTL size | Performance |
|---|---|---|---|
| 33MHz | 57KG | 7.0KL | 80Mbps |
| 54MHz | 42KG | 5.9KL | 80Mbps |
| 108MHz | 26KG | 2.5KL | 80Mbps |

Another important aspect is behavioral IPs are much easier to modify their “functionality” and “interface” than in RTL-IPs.

The behavioral IPs sometimes generates smaller circuits than RTL IPs as behavioral synthesis share registers and functional units for sequential algorithms, but recent RTL designers do not usually share registers since such time multiplexed sharing makes RTL simulation and debug very difficult.

## *Configurable Processor Synthesis*

Since chip fabrication cost have raised considerably, SoC are becoming as flexible as possible. For this purpose, recent SoC usually have several configurable processors besides a main CPU. These configurable processors should be small, have a high performance and low power consumption for a specific application. Such a configurable processor is also called Application Specific Instruction set Processor (ASIP). ASIPs employ custom instruction-sets to accelerate some applications. The CWB provides ASIP’s base processor and supplementary instructions that are described fully in behavioral C, which are behavioral synthesized. This allows the base-processors and the addition of instructions to share functional units. This sharing leads to much smaller circuits than the conventional RTL-based ASIPs. C-based ASIPs are more flexible than RTL-based ones in terms of public register number, pipeline stages or interrupt policy.

## *Automatic Architecture Exploration*

CWB allows the creation of multitude hardware architecture for a unique C design. The user can specify a set of constraints which all architectures have to meet (e.g. area, latency, power) and a set of different architectures that meets those constraints will automatically be generated. The area-performance-power trade- offs can be easily analyzed and the architecture that meets the constraints with the lowest cost can be chosen by the designer.

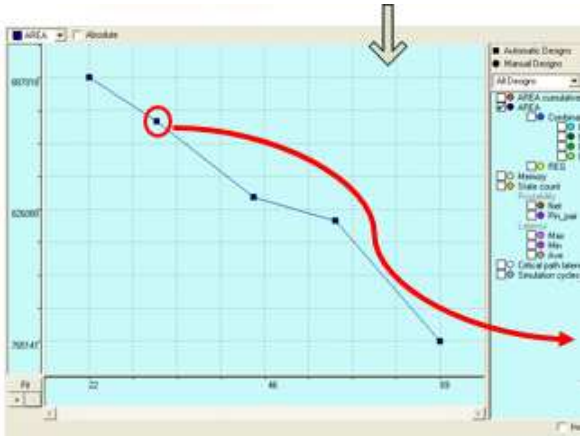

Fig 1-5. Automatic Architectures Exploration

## System VLSI Design Example Using C-Based Behavioral Synthesis

Fig.1-6 shows a design example of a real complex SoC used by a cell phones generated with the behavioral synthesizer. This SoC is called MP211, which has three ARM cores, one DSP, several dedicated hardware engines and various applications of mobile phone such as audio and video processing, voice recognition, encryption, Java and so on.

Wide ranges of circuits including control dominated circuits and data-intensive circuits were successfully implemented. The grey boxes (including bus) indicate modules that have been synthesized from C descriptions with the proposed behavioral synthesizer, while the white boxes are IP cores given in RTL format (some are legacy RTL components. All newly developed modules are designed with our C-based design flow. This example clearly illustrates that our C-based environment is able to design entire SoC designs, and not only algorithmic modules.

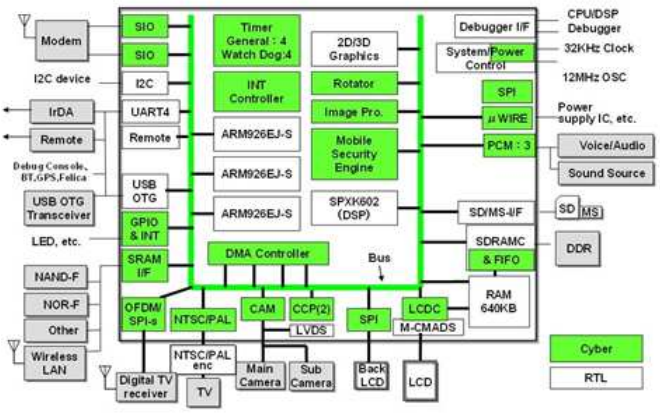


Figure 1-6. Behavioral design flow design example used in a cell phone SoC (Green boxes design using Cyber)

## Summary and Conclusion

The advantages of CyberWorkBench includes faster development time, hardware-software co-simulation and development, easier and faster verification as well as automatic system exploration are some of these.

CWB tool is as mature as logic synthesis in the late 80's, when designers started to use them widely RTL level design flows. These days' designers adopt this new design paradigm shifting from RTL "structural" domain thinking to "behavioral" domain thinking.

# Detailed Specification of CWB

## *Behavioral Synthesis*

Automatic conversion of C based code to HDL (Hardware Description Language)

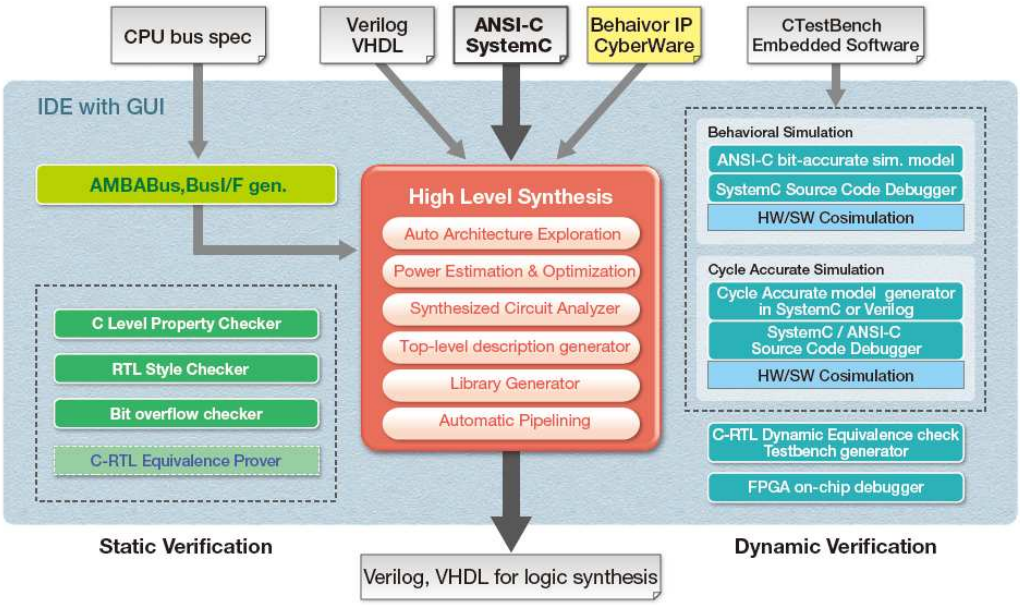


Figure 2-1. Block diagram of CyberWorkBench

## *Input supported*

a. ANSI-C

b. System C/ SpecC

c. VHDL (Legacy RTL with C-based design)

d. Verilog (Legacy RTL with C-based design)

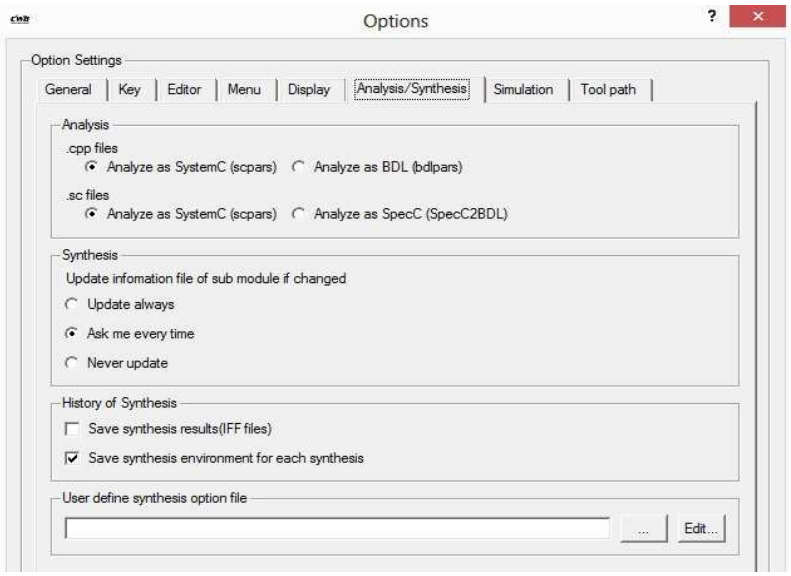


Figure 2-2. Different ways of analysis

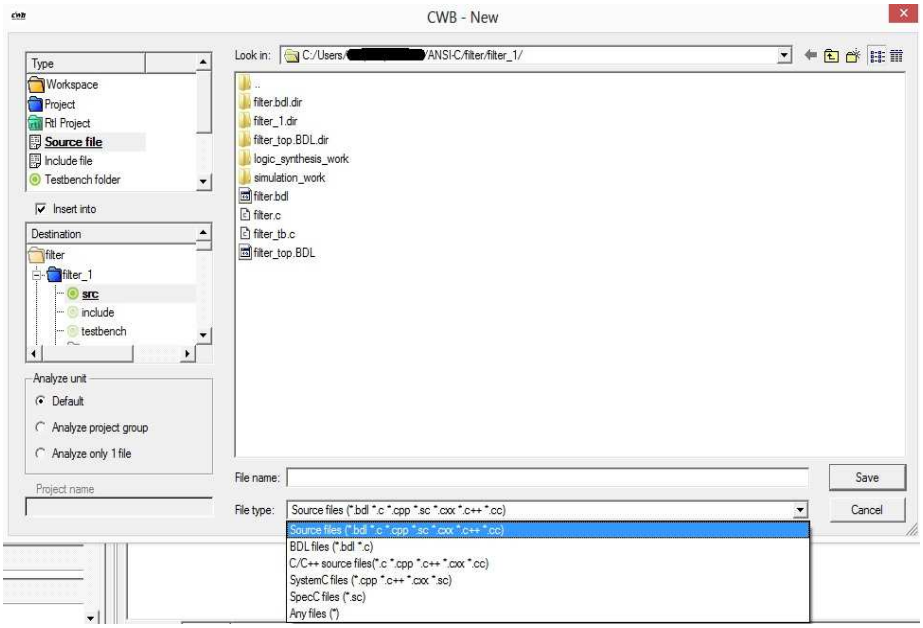


Figure 2-3. Input Supported as VHDL/Verilog (different file types supported)

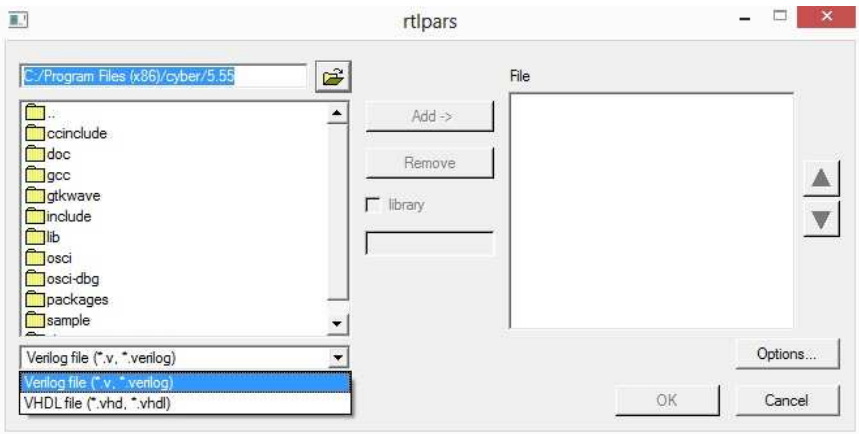


Figure 2-4. Input Supported as VHDL/Verilog

## *Output Languages*

The output generated from CWB is VHDL/ Verilog.


Figure 2-5. Generated output as VHDL/Verilog

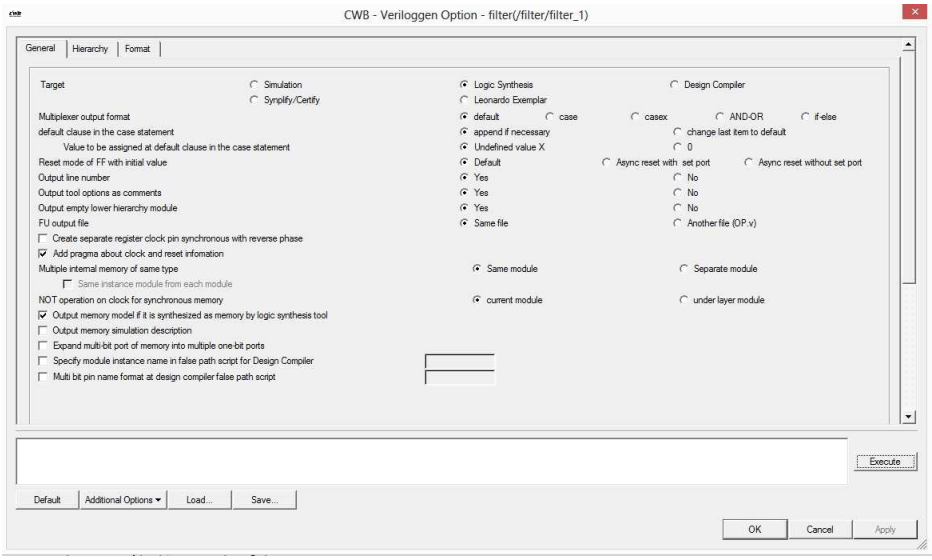


Figure 2-6(a). Different options available for selected output as Verilog

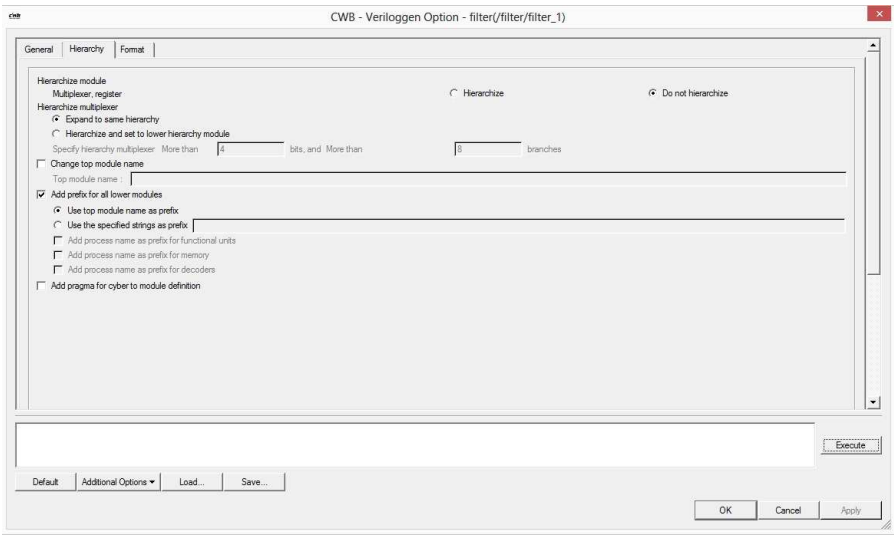


Figure 2-6(b). Different options available for selected output as Verilog

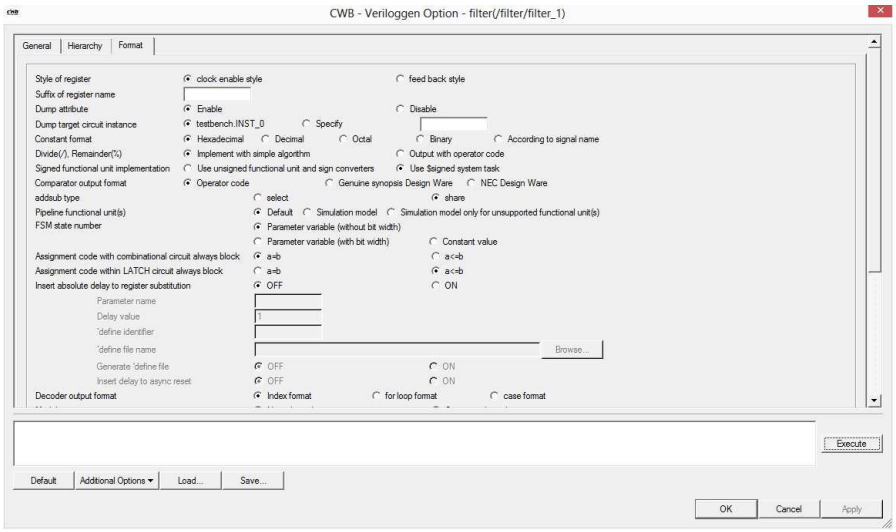


Figure 2-6(c). Different options available for selected output as Verilog

## *Target Device Supported*

Full ASIC/ FPGA support

Support all FPGA families from Xilinx and Altera

The Generated RTL is optimized for the specified technology.

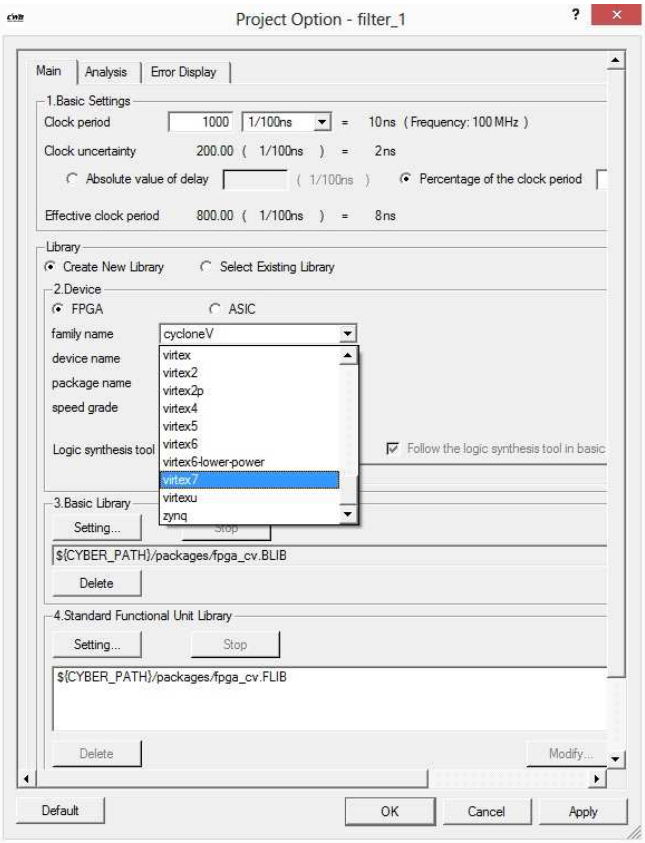


Figure 2-7. Different options for selecting families of Xilinx and Altera

## *Input Constraints*

Ability to specify the clock frequency

Ability to specify reset signal and its type.

Ability to control resource allocation: CWB allow user to specify any resource constraints that is desired during implementation of the specific module.

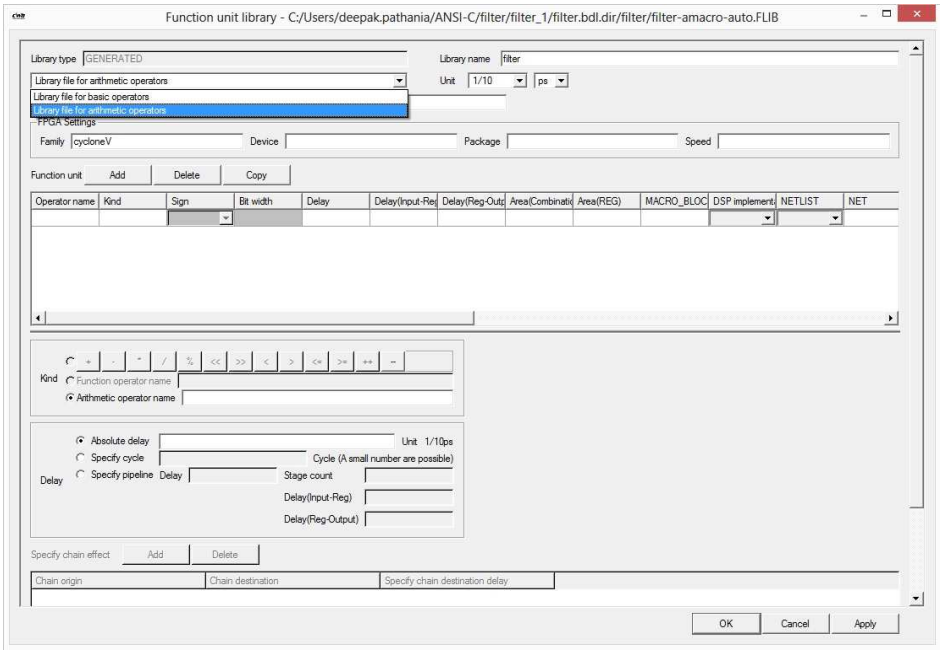


Figure 2-8. Function unit libraries for input constraints

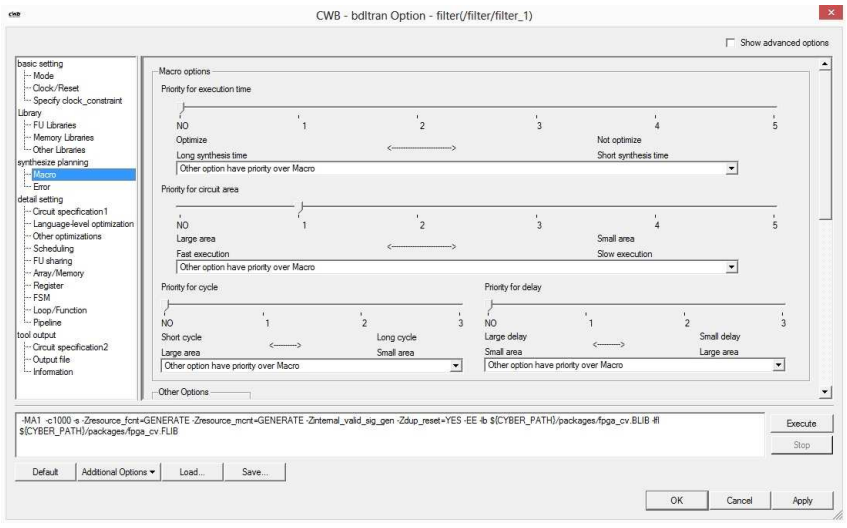


Figure 2-9. Different options for synthesize planning to control resource allocation.

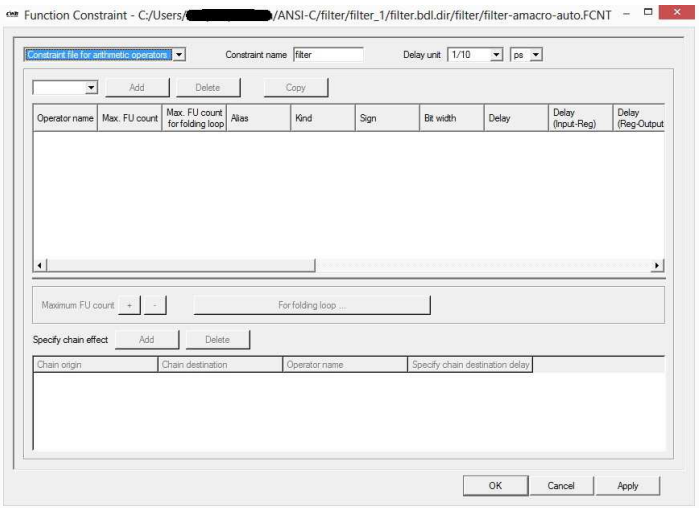


Figure 2-10. Different options for function constraint to control resource allocation.

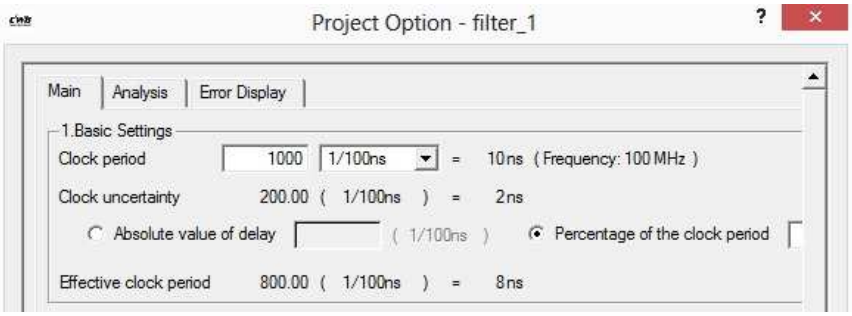


Figure 2-11. Options for specifying clock frequency

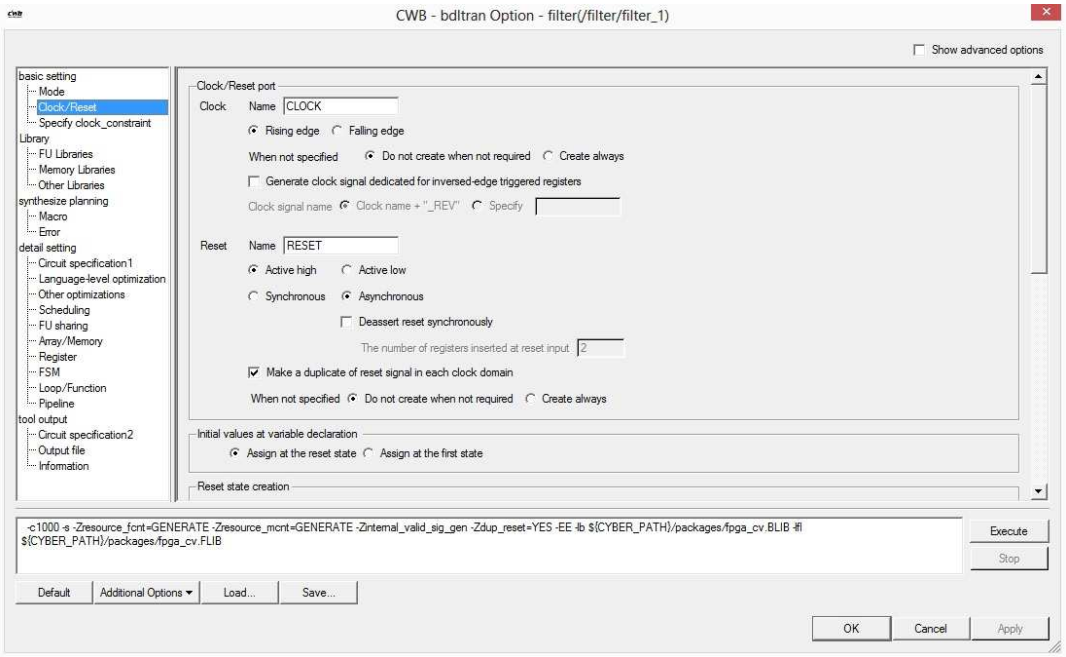


Figure 2-12. Settings for frequency reset

## *Behavioral Synthesis modes or Scheduling options (C to RTL conversion modes)*

• Automatic scheduling
• Manual Scheduling
• Mixed Scheduling (user specific)

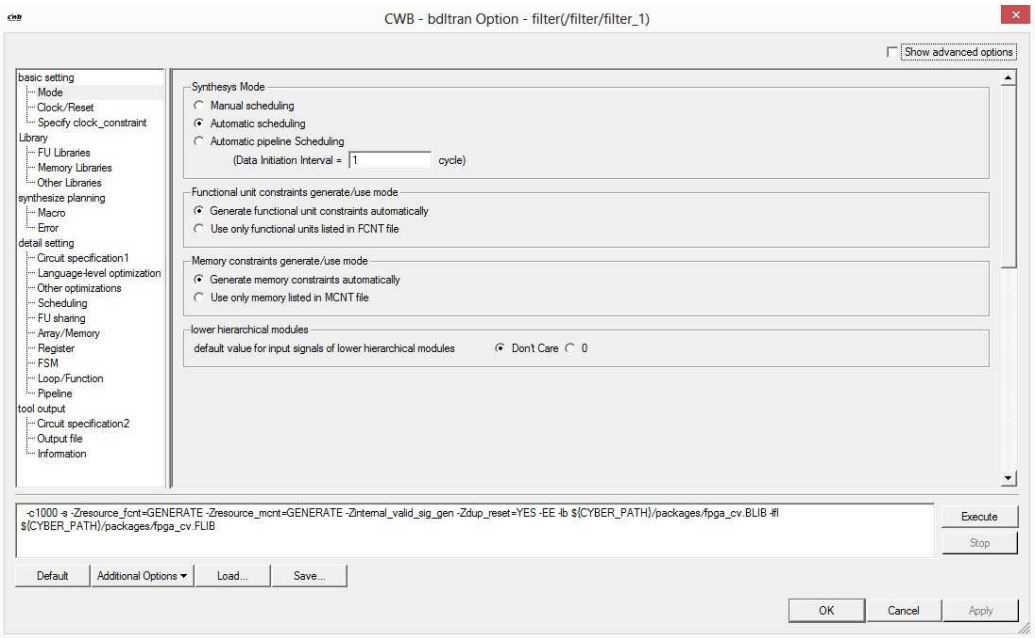


Figure 2-13. Behavioral synthesis mode (Scheduling options)

## *Other Features*

Hardware architecture or design level optimizations at each source code level as well as at module level and both by user specifications and automatically by tool such as:
Loop merging, optimization, unrolling, pipelining, false loop detection and loop parallelization.
Automatic bit-width optimization, Automatic priority logic generation for shared memories and registers, Speculations and suggestions for optimizations, Operator level optimizations, Array Overflow checks etc., Register sharing and exclusivity, Reset State Behavior defining, Interfaces, variables, arrays initializations, Selecting memory and register types (ports of memories and registers clock style or feedback style). Selecting memories as synchronous/ asynchronous and port access mechanisms.
Ability to define various types of interfaces for behavioral designs such as pipelined, buffered, serial or parallel. Ability to define scheduling of behavioral operations to be done in each cycle at source code level. Micro-architectural Design Exploration: Ability to generate multiple RTL designs from one behavior code under given or changing physical constraints. Generate Trade-off chart between area/resource, latency and timing and provide choices to user for best design

as per the requirement. Automatic top module generator to integrate different modules of (c-designed module or existing RTL designs modules) and able to define top module in C as per the user requirement. On-chip Bus Interface Generator to automatically generate bus interfaces for AMBA AHB, AXI, APB and bridges. On-chip Bus Generator to automatically generate on-chip buses (AMBA based). Ability to handle clocks domain crossing (Supports asynchronous and synchronous interface generation between two modules at different clocks) while generating top module and integration of modules. Ability to support Clock Gating. Detailed QoR report generation for the design, having information about area, latency and resources utilized.

Data path or RTL schematic Viewer capability. Dataflow diagram viewer and Signal table to show operations occurring in each state of FSM generated.

Synthesis options for generating kind of FSM encoding required. Synthesis options for kind of logic to be generated for constant array, variables and globals Ability to maintain history of changes made in the C code and behavioral synthesis in case user wishes to compare or move to previous design or next design. RTL generation options: RTL code generation as per the target (Logic or FPGA Synthesis, DC Compiler, Simulation etc.)

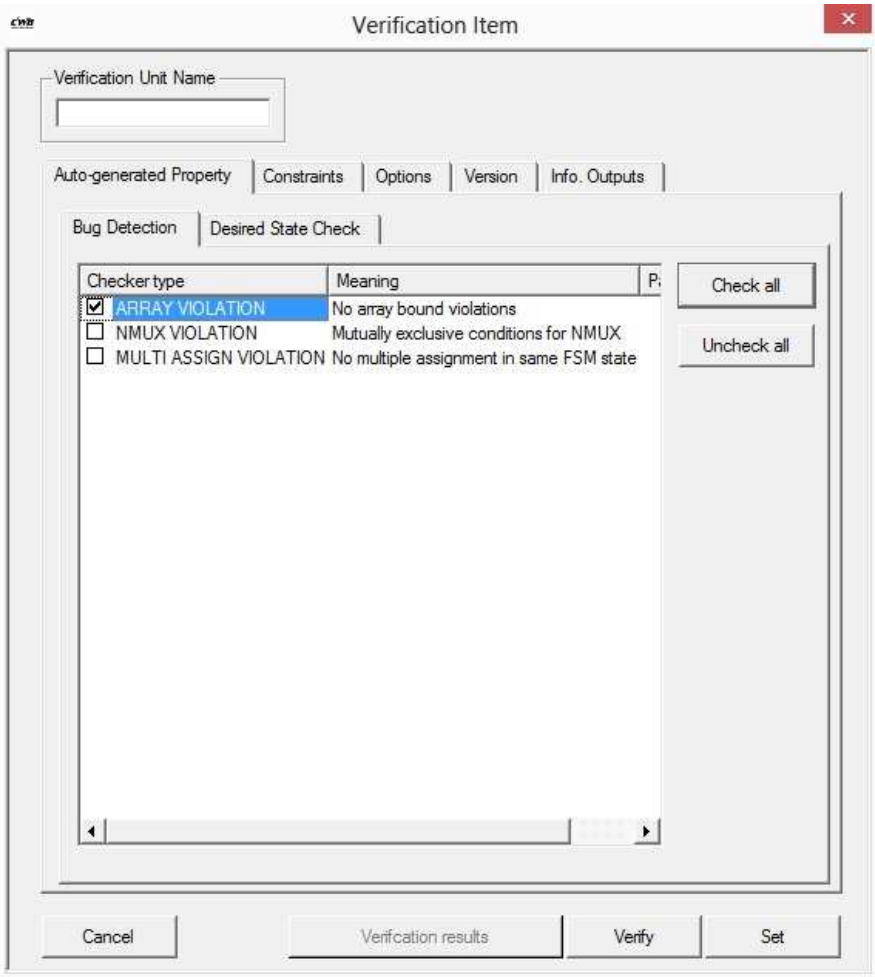


Figure 2-14. Array overflow check

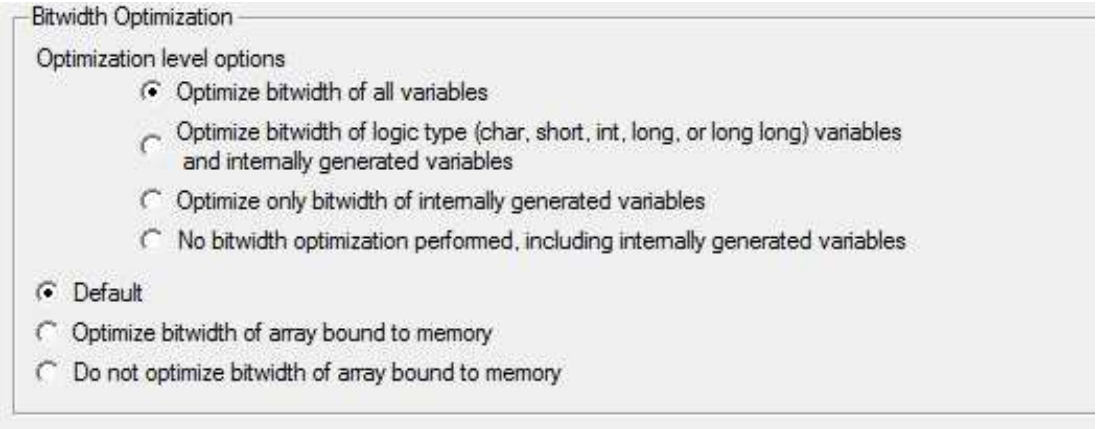


Figure 2-15. Automatic bit width optimization

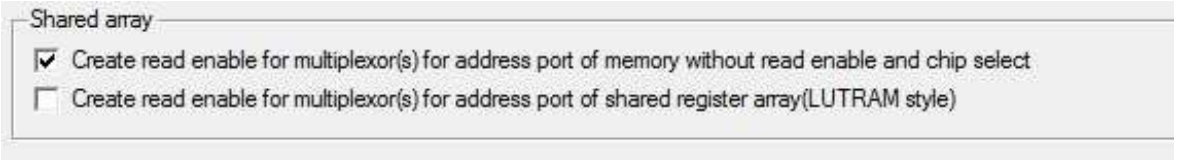


Figure 2-16. Automatic priority logic generation

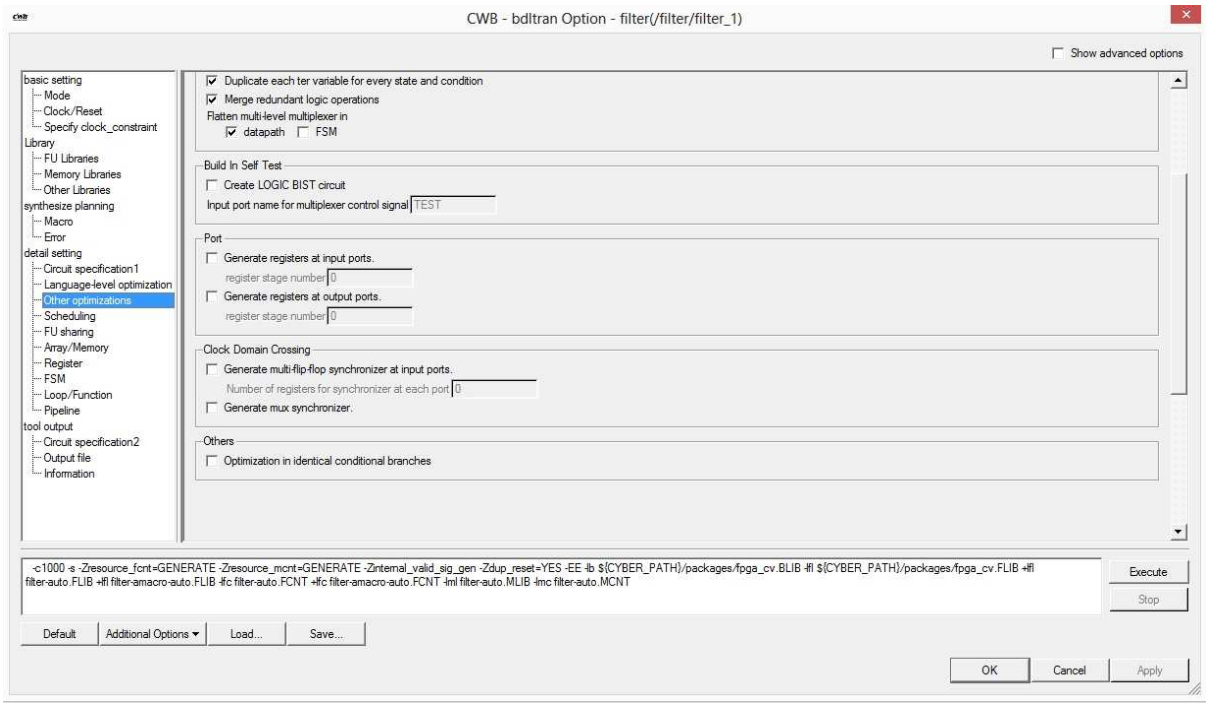


Figure 2-17. Clock domain crossing

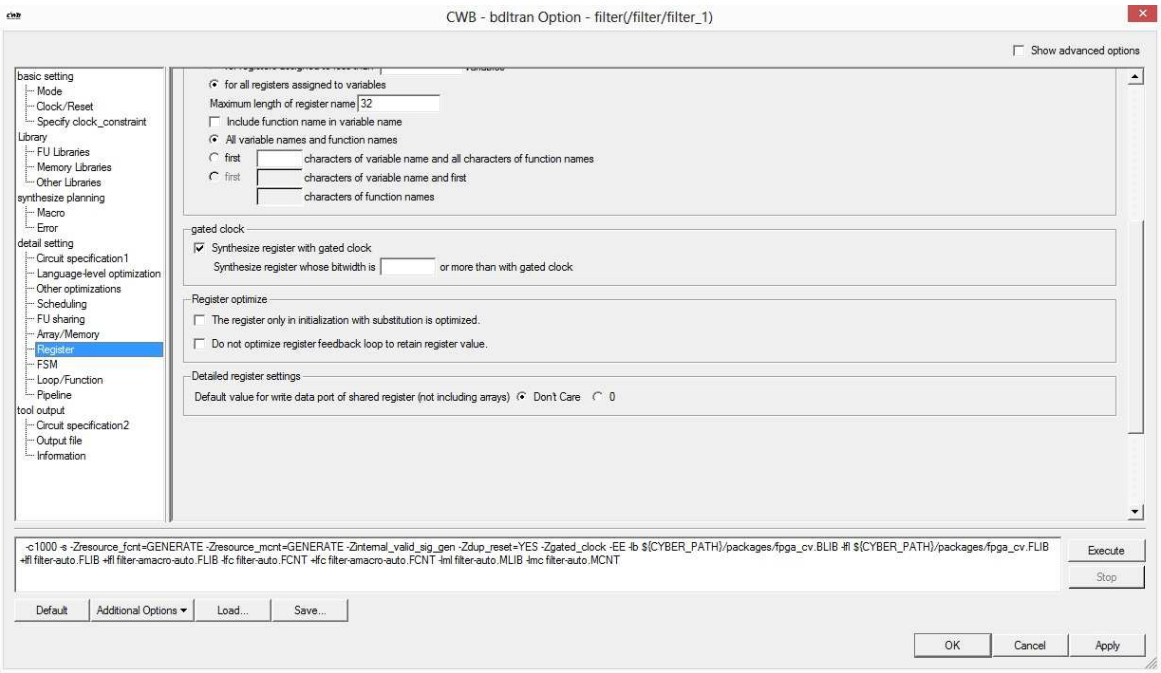


Figure 2-18. Clock gating

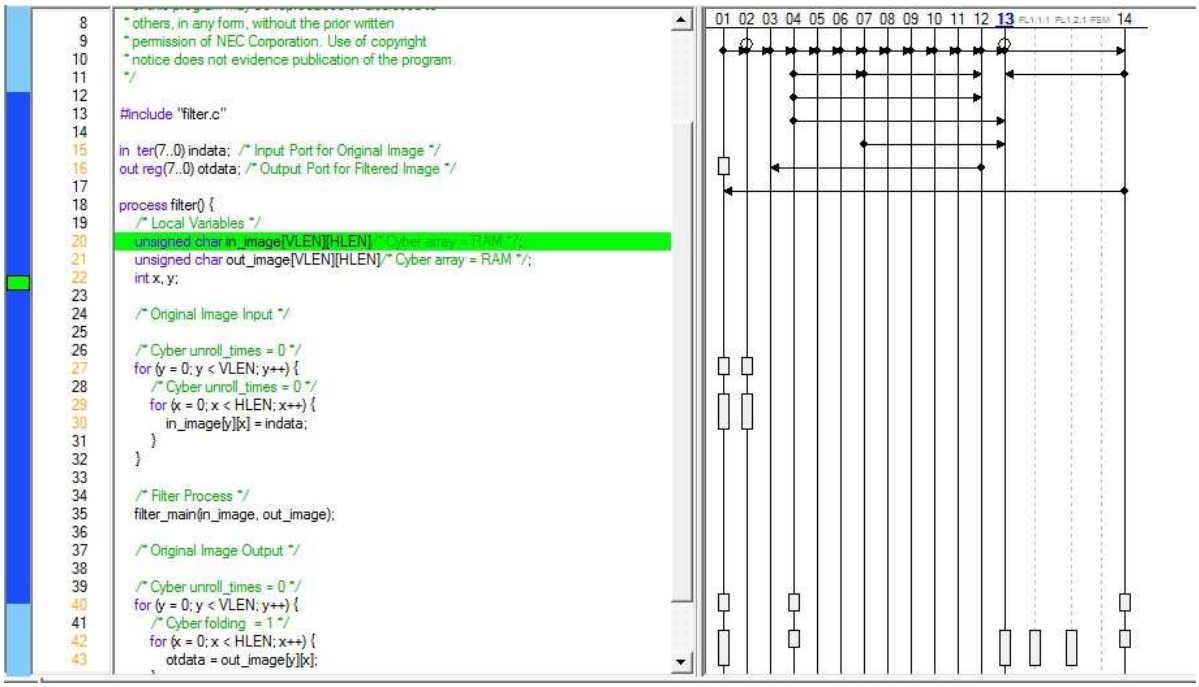


Figure 2-19. Dataflow viewer

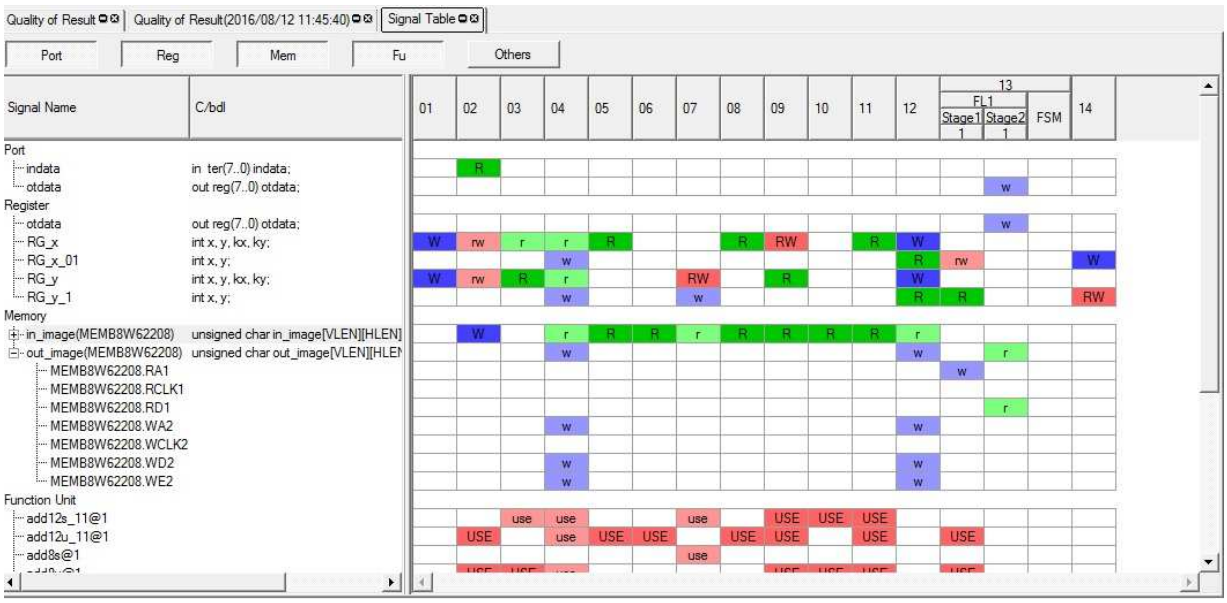


Figure 2-20. Dataflow viewer signal table

## Timed C Coding (Specify clock for C language coding)

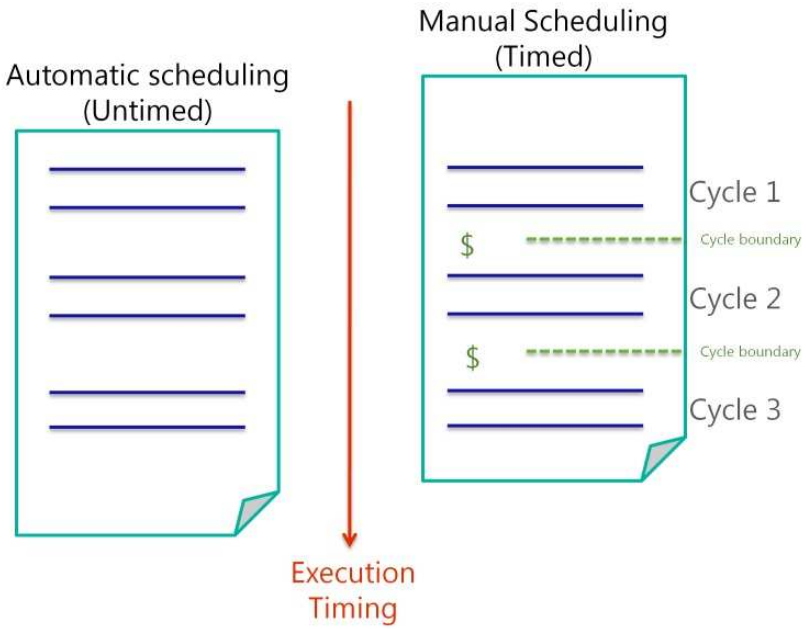


Figure 2-21. Different scheduling behavioral operations to be done each cycle

## Cyber Synthesis Report

### Summary

| Module Name | Basic Library Name |
|---|---|
| filter | CWBSTDBLIB |

| FPGA Family | FPGA Device | FPGA Package | FPGA Speed |
|---|---|---|---|
| cycloneV | - | - | - |

| Resource Utilization | | | |
|---|---|---|---|
| ALUTs *1 | Registers | Block Memory Bits | DSPs |
| 498 | 189 | 995,328 | 0 |

| Latency Index | Clock Period | Net | Port | |
|---|---|---|---|---|
| 746,929 | 10ns | 617 | 18 | |
| Total States | Critical Path Delay | Pin Pair | In | Out |
| 14 | 11.4213ns | 2,144 | 10 | 8 |

### Resource Utilization

| Module Name | Count | ALUTs *1 | Registers | Block Memory Bits | DSPs |
|---|---|---|---|---|---|
| Total | - | 498 | 189 | 995,328 | 0 |

### Functional Unit

| FU Name | Kind | Sign | Bit Width | Area | Reg | Delay (ns) | Count |
|---|---|---|---|---|---|---|---|
| add12s_11 | + | signed | (12,9) 11 | 13 | 0 | 0.99 | 1 |
| add12u_11 | + | unsigned | (11,9) 11 | 13 | 0 | 0.99 | 1 |
| add8s | + | signed | (8,6,1) 8 | 9 | 0 | 0.86 | 1 |

Figure 2-22. Detailed QOR report

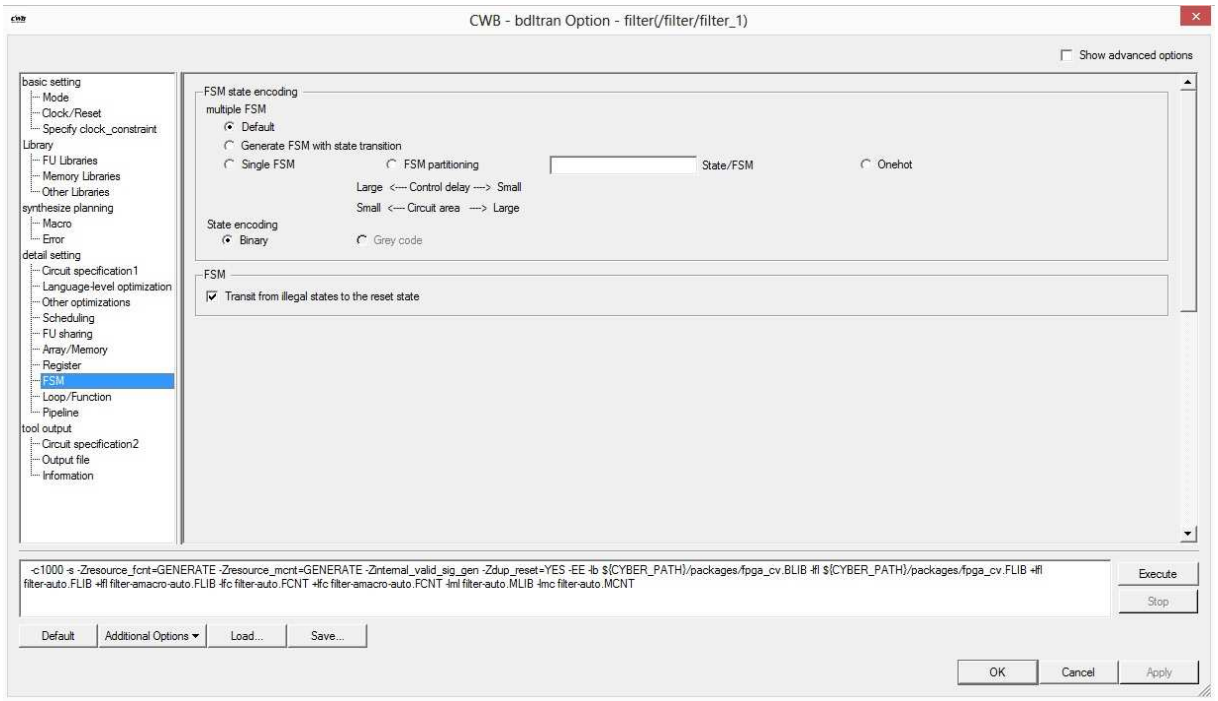


Figure 2-23. FMS encoding

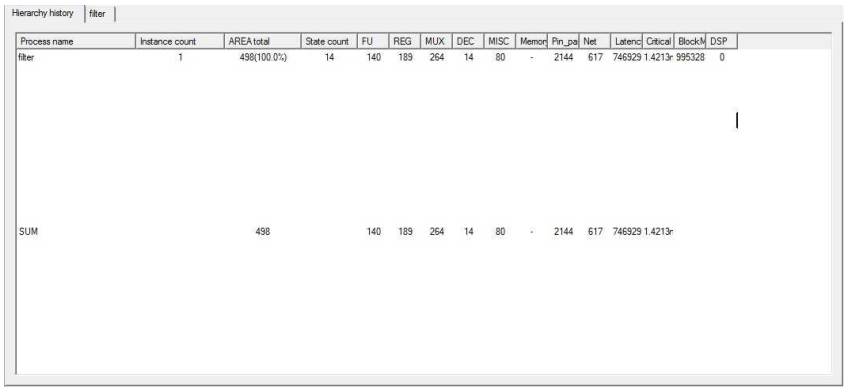


Figure 2-24. History of synthesis


Figure 2-25(a). Kind of logic for constant array variables global

```
/* 3 x 3 Filtered Coefficients */
const char coeff[KSIZE][KSIZE]/* Cyber array = LOGIC */ = {
   {-8, -8, -8},
   {-8, 64, -8},
   {-8, -8, -8}
};
```

Figure 2-25(b). Kind of logic for constant array variables global

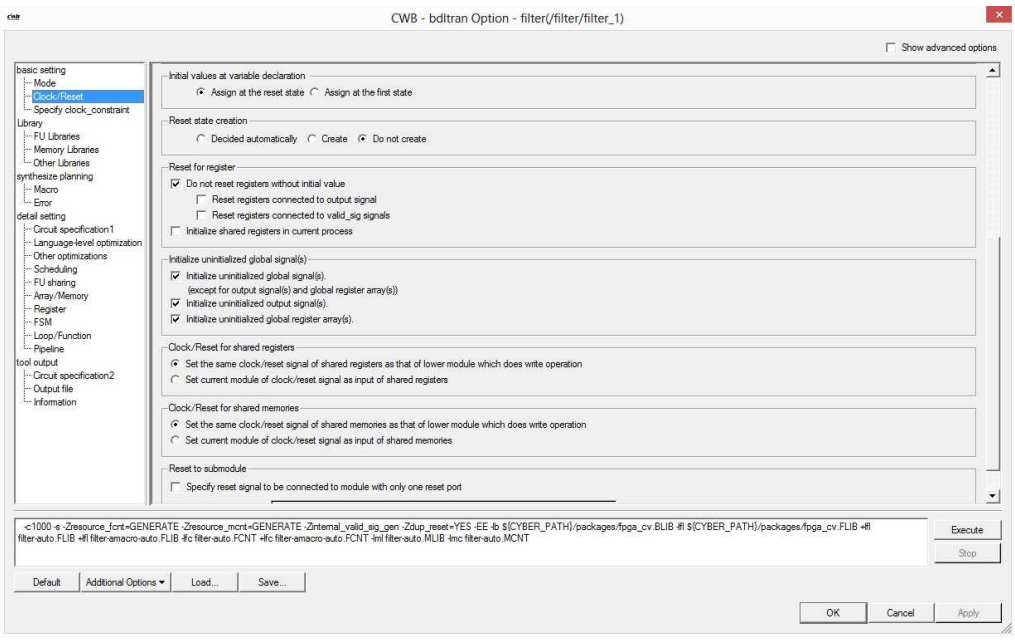


Figure 2-26. Different options for Interface variables arrays initializations

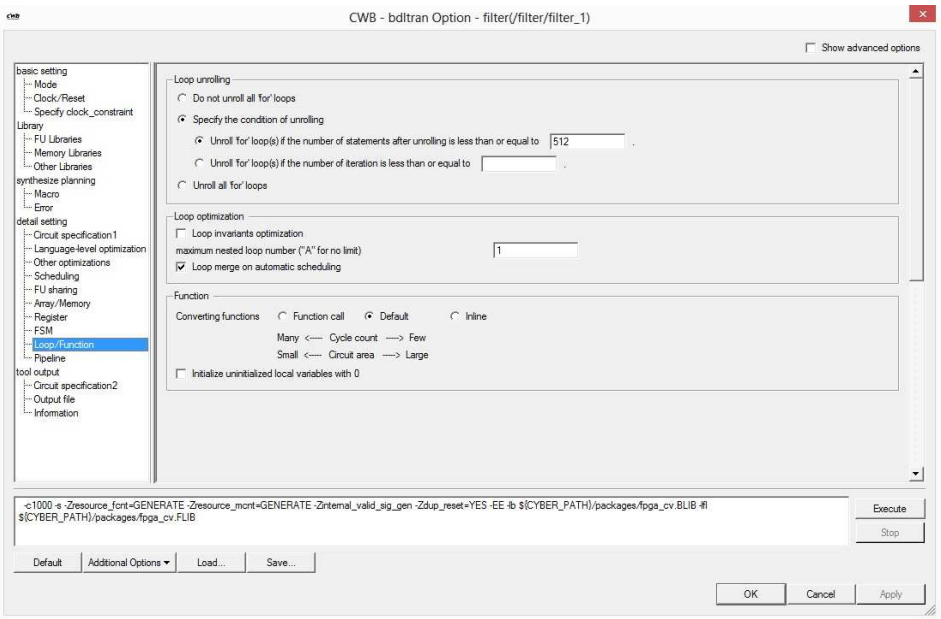


Figure 2-27(a). Options for Loop merging, optimization, unrolling, parallelization


Figure 2-27(b). Options for Loop merging, optimization, unrolling, parallelization

Micro-architectural Design Exploration: Ability to generate multiple RTL designs from one behavior code under given or changing physical constraints. Generate Trade-off chart between area/resource, latency and timing and provide choices to user for best design as per the requirement.

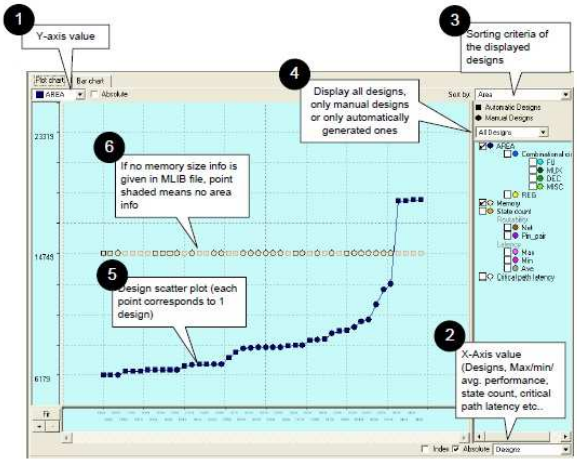


Figure 2-28(a). Micro-architectural design exploration

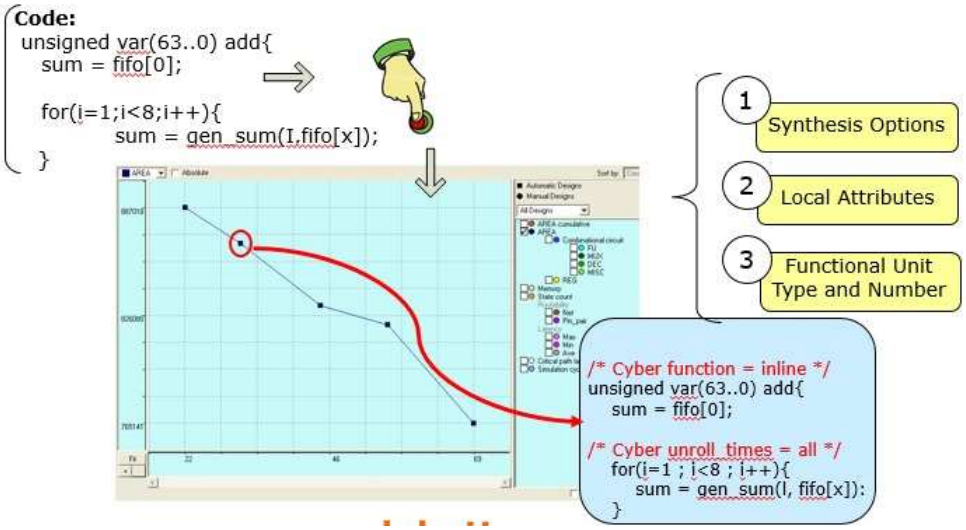


Figure 2-28(b). Micro-architectural design exploration

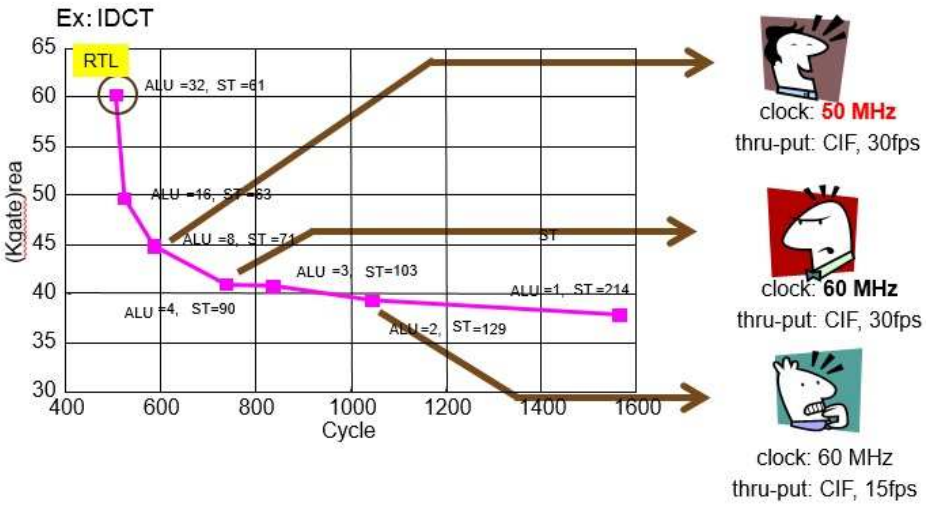


Figure 2-28(c). Micro-architectural design exploration

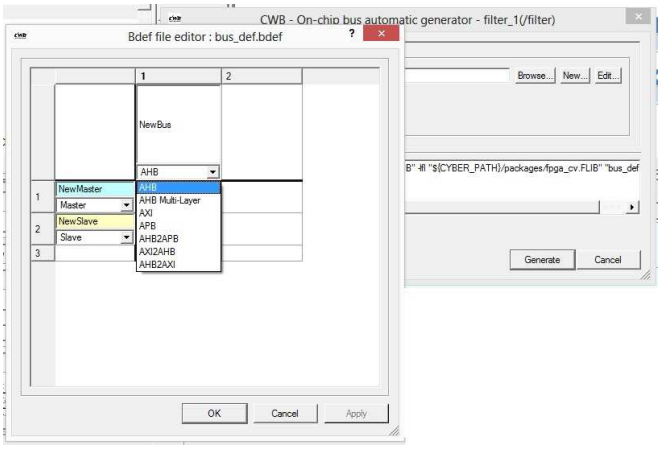


Figure 2-29. On chip bus generator

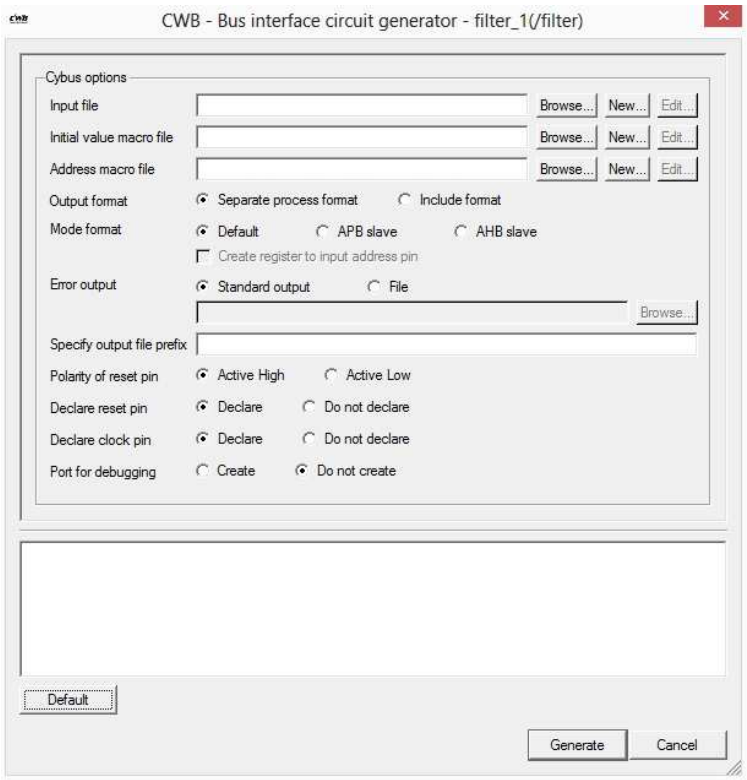


Figure 2-30. On chip bus interface generator

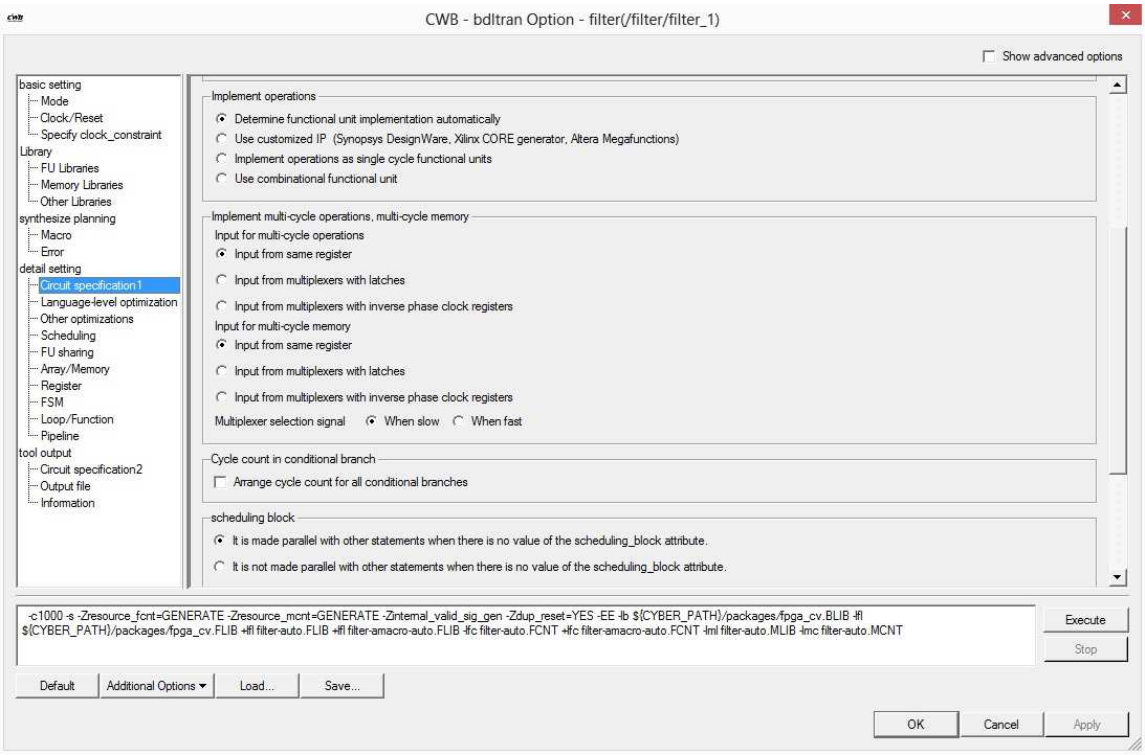


Figure 2-31. Operator Level Optimization

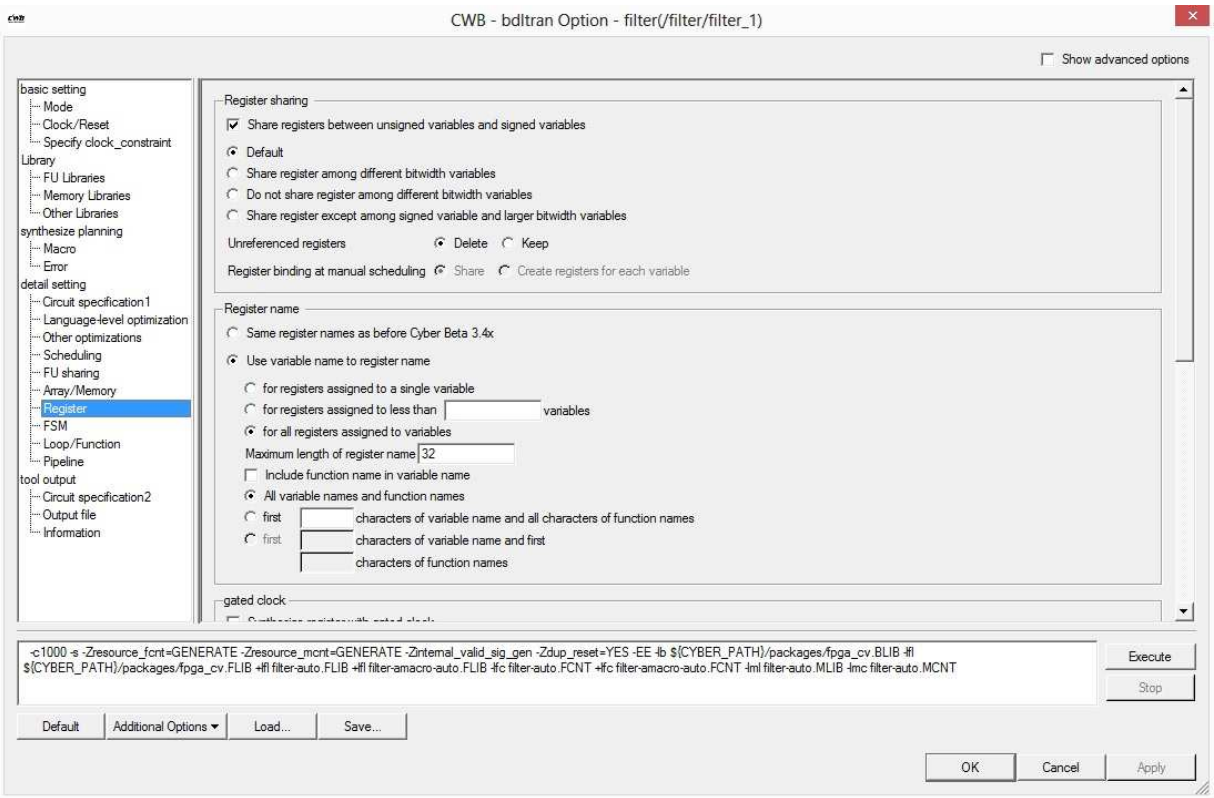


Figure 2-32. Different options for Register Sharing Exclusivity

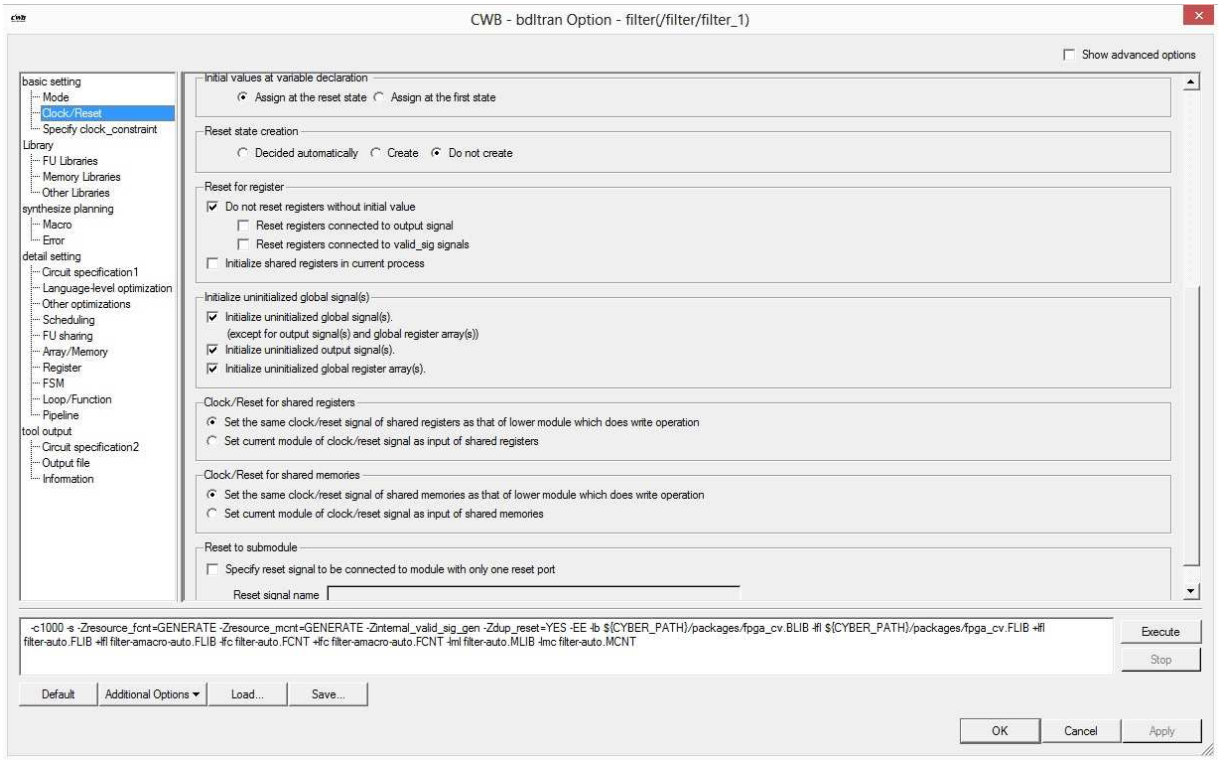


Figure 2-33. Options for reset state behavior defining

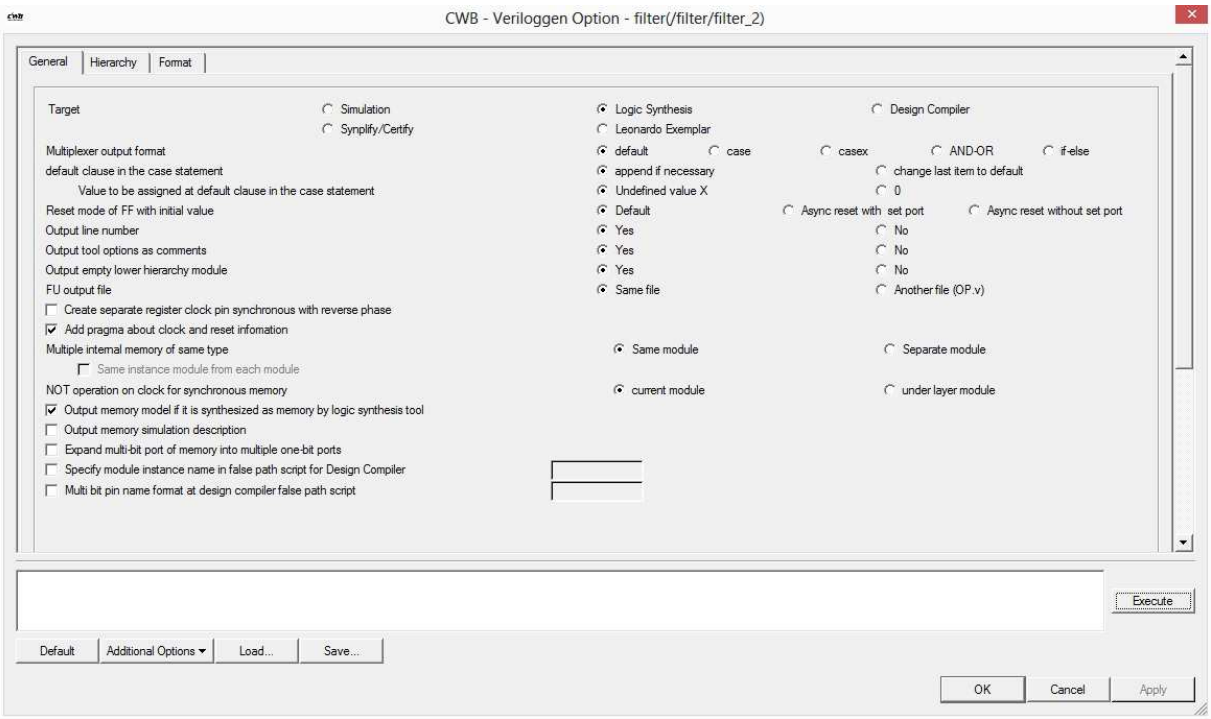


Figure 2-34. Different options for RTL generation

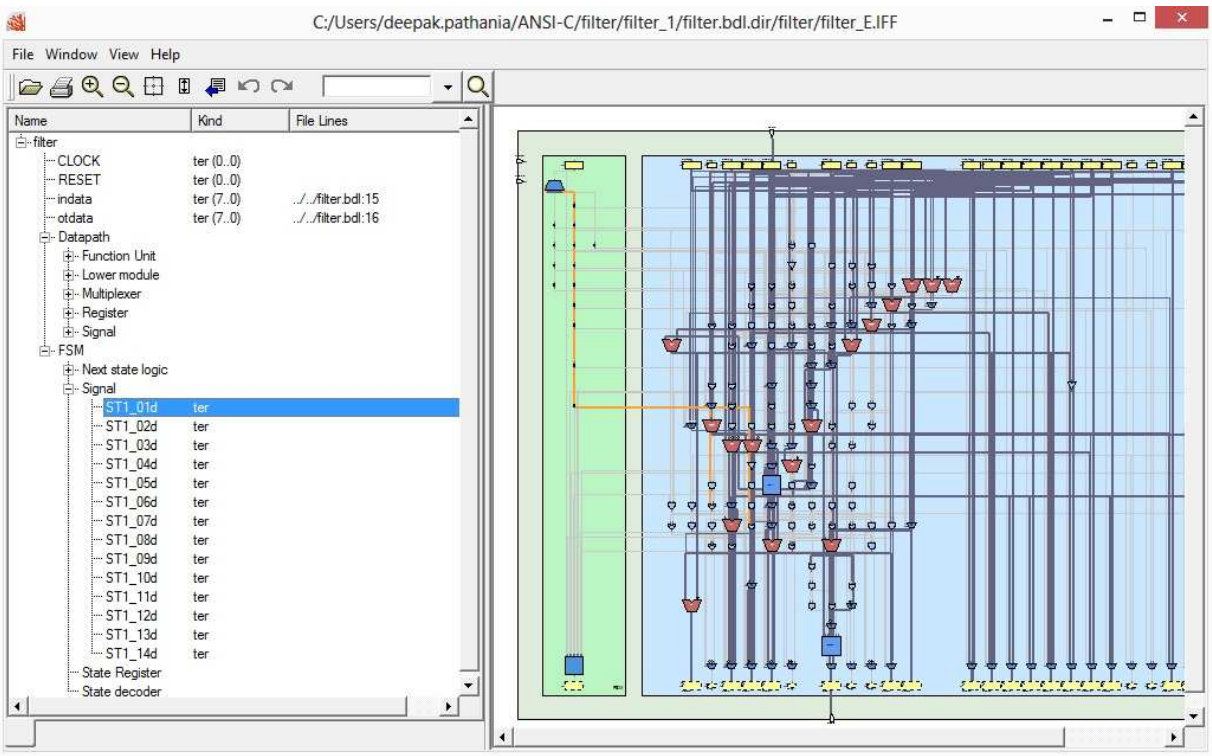


Figure 2-35. RTL Schematic Viewer

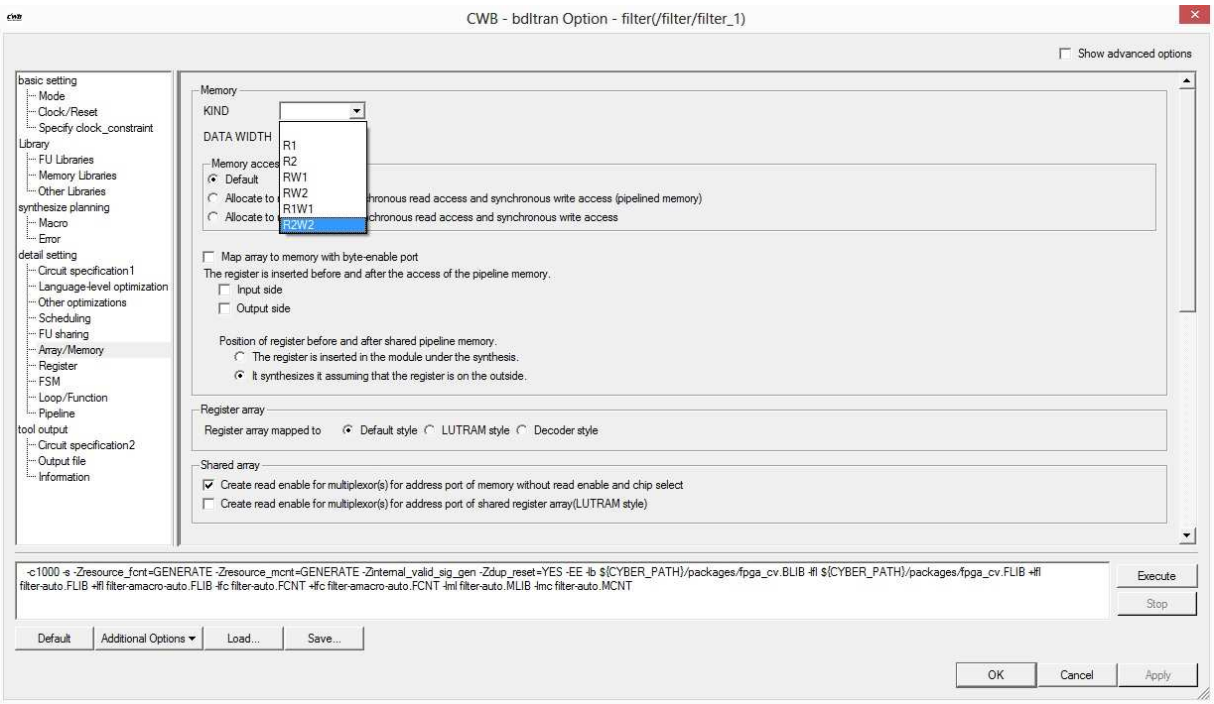


Figure 2-36. Different options for selecting memories sync async

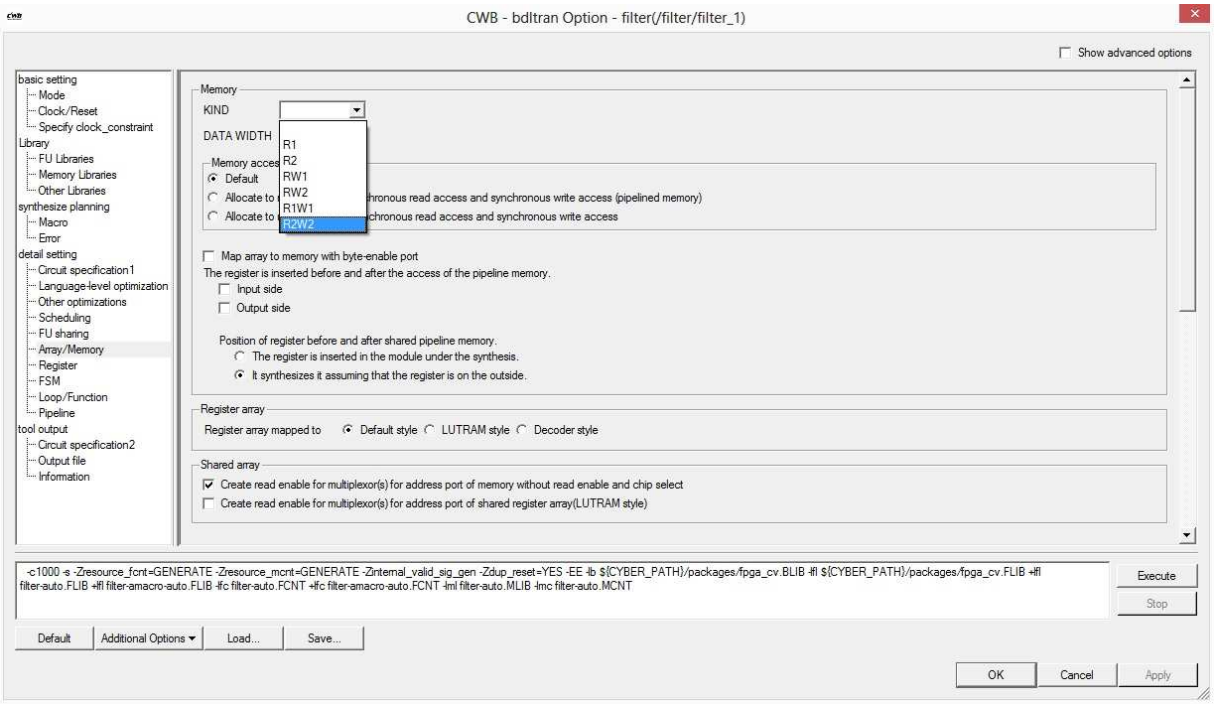


Figure 2-37. Different options for selecting memory register types

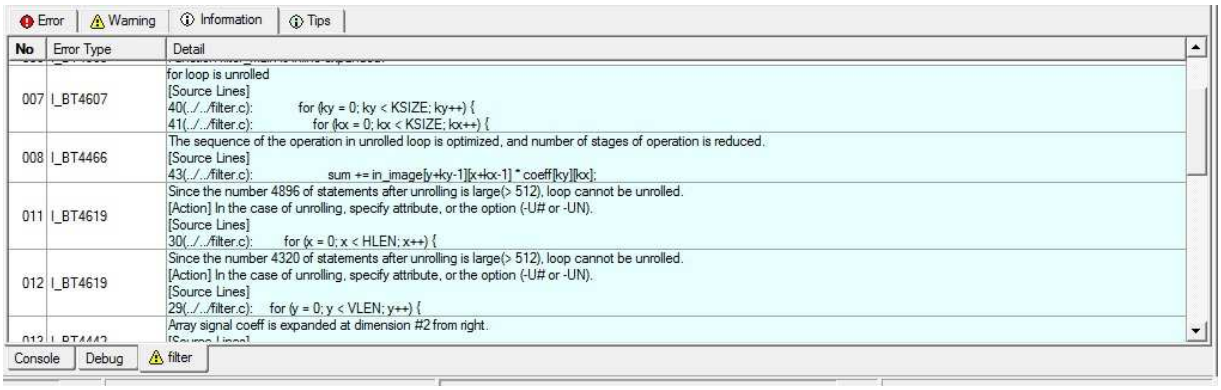


Figure 2-38. Speculations suggestions for optimizations

Automatic top module generator to integrate different modules of (c-designed module or existing RTL designs modules) and able to define top module in C as per the user requirement.

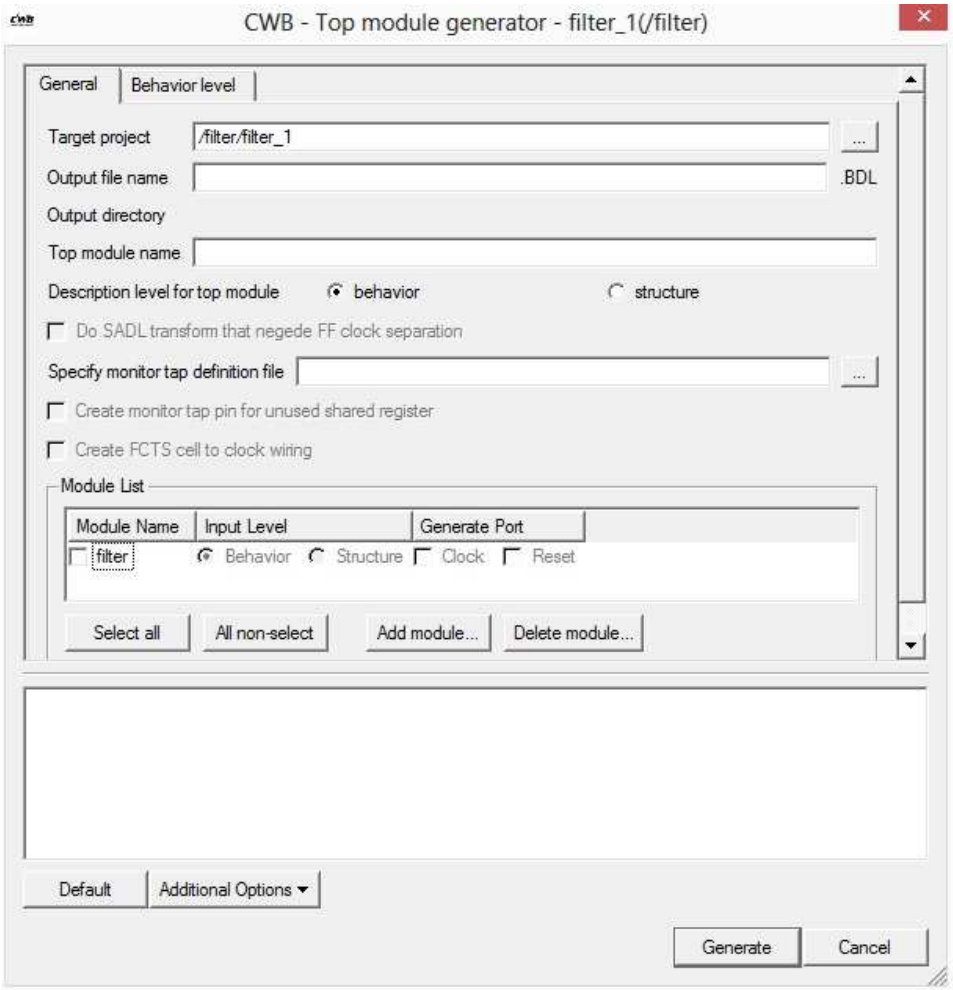


Figure 2-39. Top Module Generator

## *Verification*

Automated Test bench generator for simulation and synthesis and ability to generate test bench at behavioral level and automatic conversion of those test vector for RTL. Automated SystemC cycle accurate model generation for testing module level as well as system-level (top level).

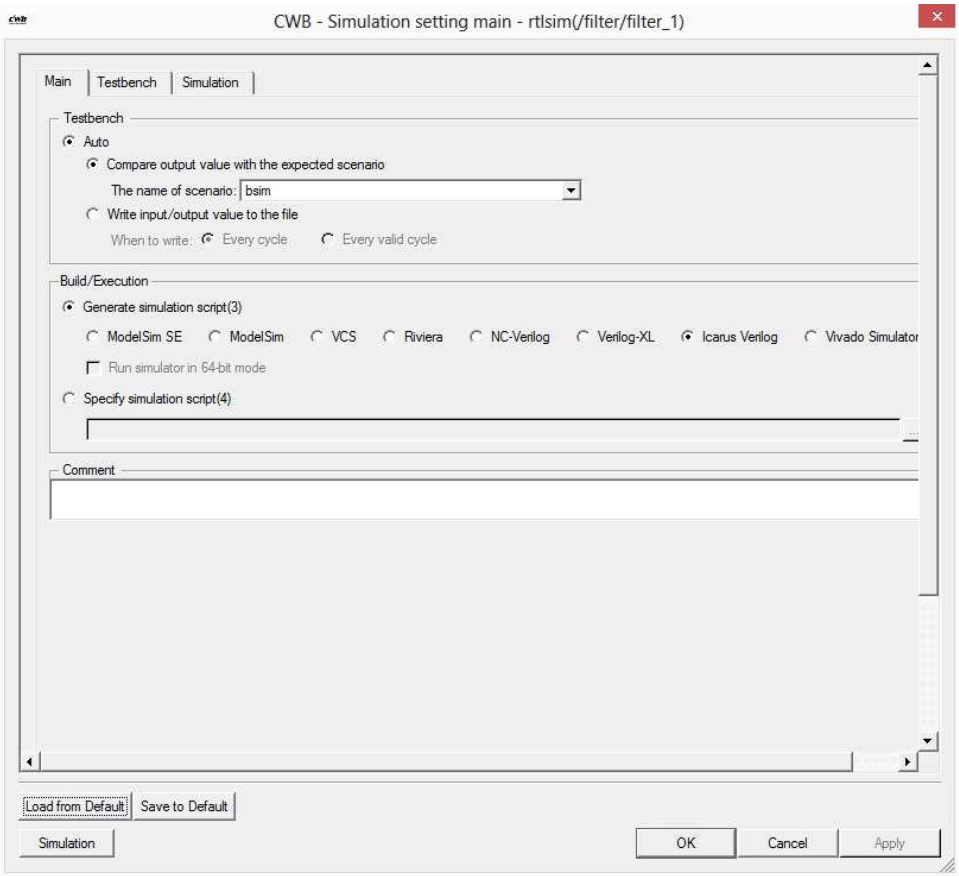


Figure 2-40. Verification Automated script generation

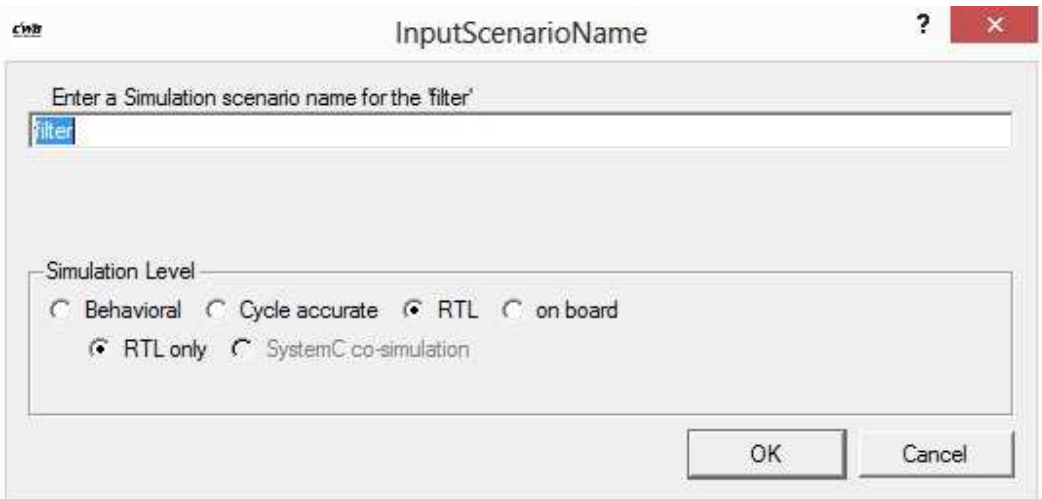


Figure 2-41(a). Verification Automated TestBench Generation

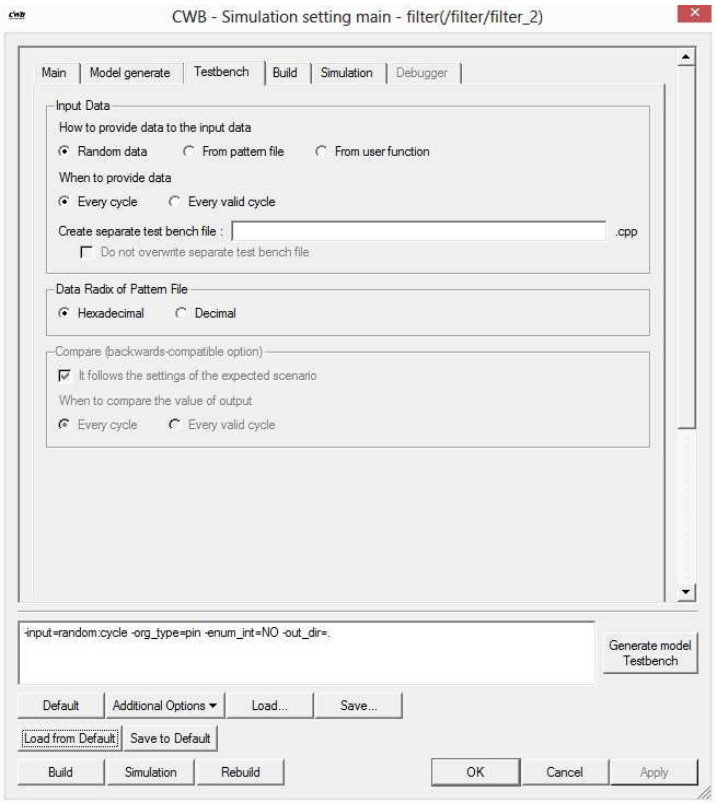


Figure 2-41(b). Verification Automated TestBench Generation

Automated verifications or comparisons of results of C behavioral source code results with Cycle accurate or RTL simulations results and display of results. Automated script generation for third party RTL simulators for performing RTL simulations of the designs. Cycle-Accurate SW/HW testing and co-simulation C source code Debugging of the RTL code, Cross Probing between C code and RTL code, C source code testing and verification.

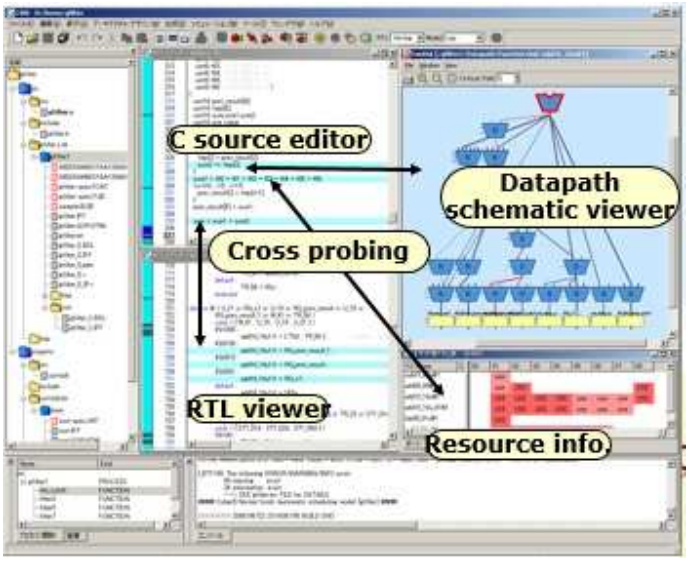

Figure 2-42. Cross Probing C-RTL code

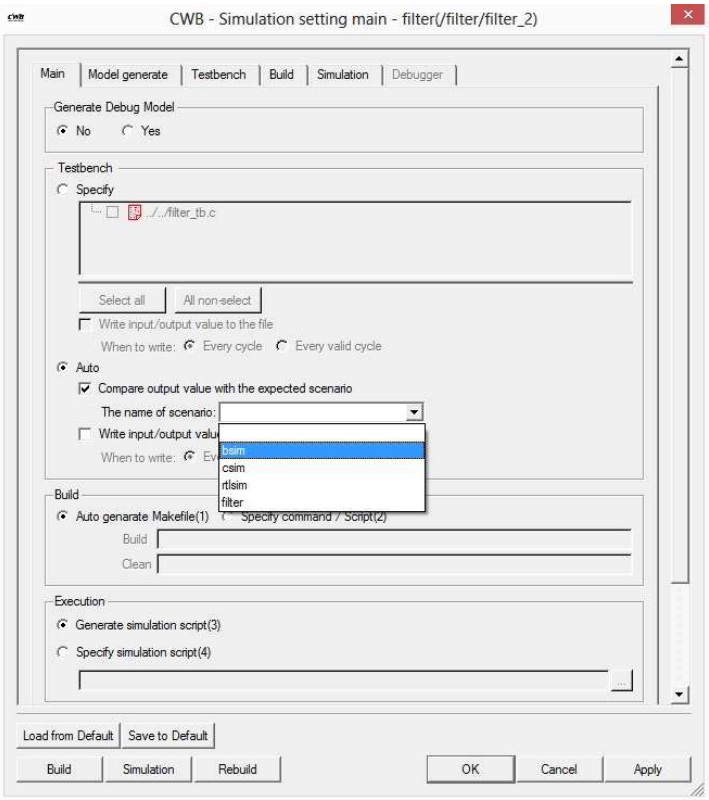


Figure 2-43. Different options for automated verification comparison of results of C

Ability to use Transaction Level (TLM) Test Vectors for cycle accurate simulations as well as RTL level simulations (Same test vectors for software can be re-used for cycle accurate simulations).

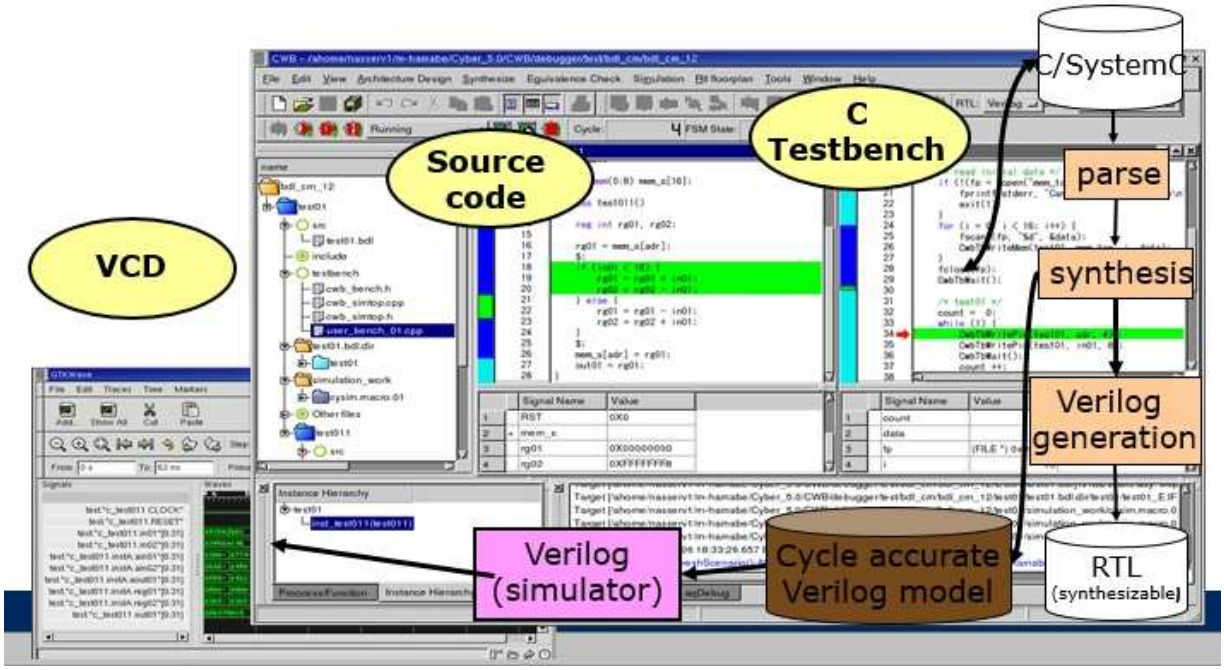

Figure 2-44. Verification C Source Code Debugger

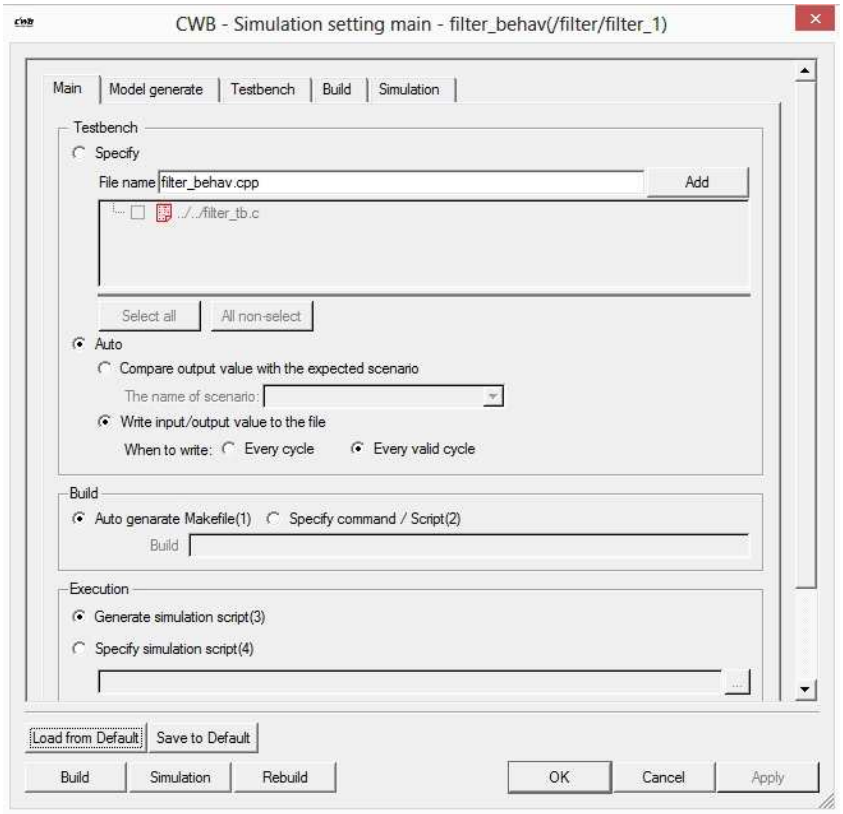


Figure 2-45. Different options for C Source Code Testing Verification

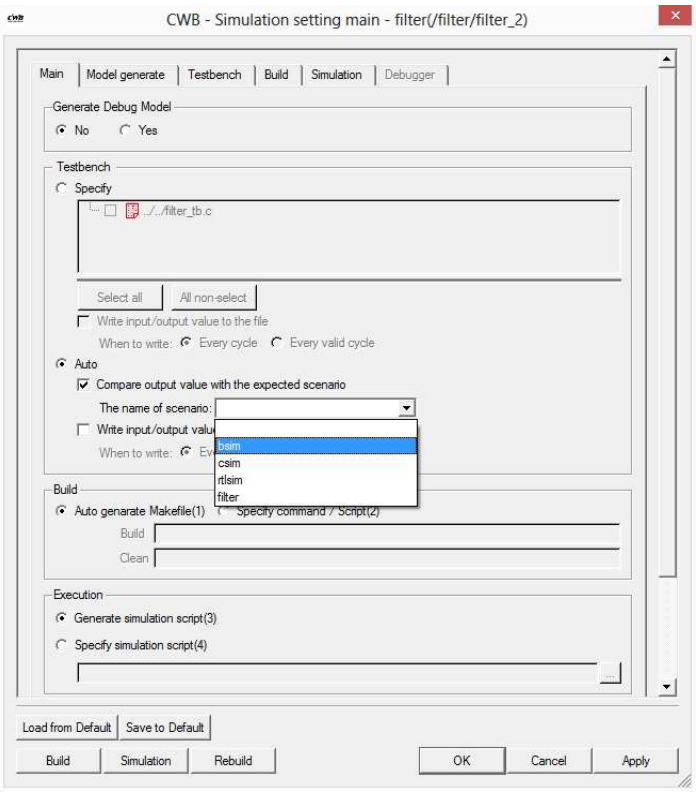


Figure 2-46. Cycle Accurate Software-Hardware Testing Co-simulation

## *Integration with Third Party Tools*

Output generated should be compatible with other synthesis tools like Design Compiler, ISE, Vivado, Synplify, Quartus etc. Ability to generate scripts for automatically invoking tools both for Command Line User Interface (CUI) and Graphical User Interface (GUI). Provide waveform viewer such as GTKWave integrated in the tool (With no additional license requirement). OSCI (Open SystemC Initiative) simulator for cycle-accurate simulations integrated in the tool.

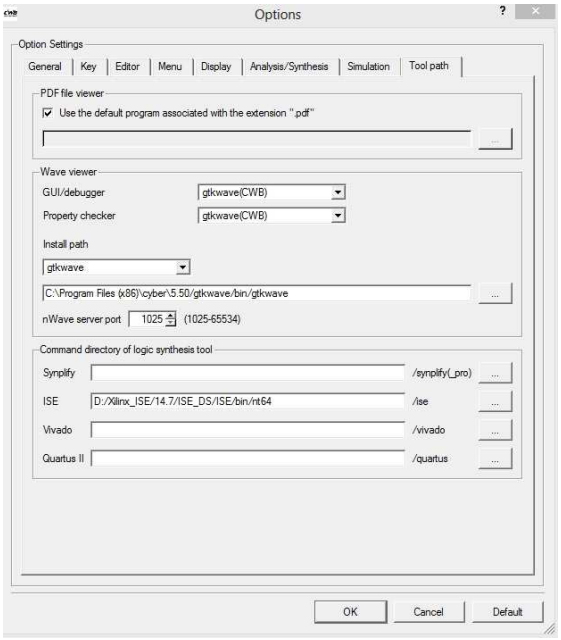


Figure 2-47. Different options for GTKWave Integration

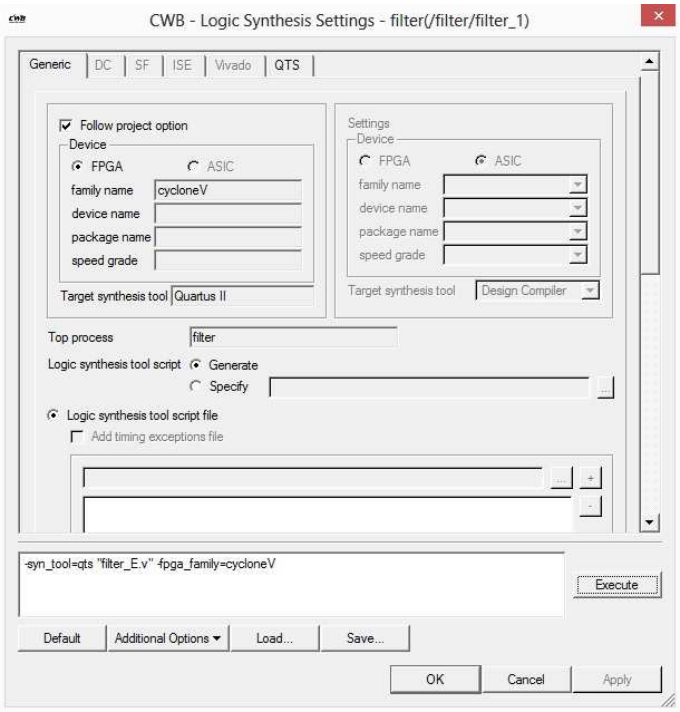


Figure 2-49. Different options for Output generated compatibility