AMBA DesignWare® and coreAssembler Simplify the Design Flow and Improve Design Timing for ST Microelectronics Digital Radio Controller & Audio Decoder

Mauro Bosco, ST Microelectronics

Sam Bordbar, Synopsys Professional Services

Andreas Vielhaber, Synopsys, Inc

Milan, Italy

ABSTRACT

The complexity of today's System On Chip (SoC) designs requires a faster and simpler flow and methodology for new SoC design projects.

To get more SoCs to market faster and less expensively, STMicroelectronics combined forces with Synopsys® Professional Services to conceive a new flow and methodology for digital audio system platform.

This paper describes how SYNOPSYS® coreAssembler simplifies the process of assembling an AMBA system using AMBA DesignWare® by automating the configuration and interconnection steps, providing an automated path to implement the AMBA platform, and improving the verification by using the VIP. This flow has been used for designing and validating a Digital Radio System Controller and Audio Decoder architecture developed by ST Microelectronics Digital Broadcasting Radio Division (Automotive Product Group).

Table of Contents

1.0 List of acronyms

Aac+	MPEG-4 high efficiency advanced audio coding
AHB	advanced high-performance bus
AMBA	advanced microcontroller bus architecture
APB	advanced peripheral bus
cA	coreAssembler
cB	coreBuilder
DAB	digital audio broadcasting
DMA	direct memory access
DW	DesignWare
GPIO	general purpose input output
HVL	hardware verification language
I2C	inter-integrated circuit
IP	intellectual property
IRDA	infrared data association
Mp3	acronym for ISO MPEG Audio Layer 3
PTG	protocol transaction generator
RTC	real time clock
SDR-DRAM	single data rate synchronous dynamic random access memory
SDRAM	synchronous dynamic random access memory
SPS	Synopsys Professional Services
SRAM	static random access memory
SoC	system on chip
TCG	transaction choice generator
TCL	tool command language
TSG	transaction sequence generator
UDMA	ultra direct memory access
USB	universal serial bus
VHDL	VHSIC hardware description language
VIP	verification intellectual property
Wma	windows media audio

2.0 Content of the paper

This paper is structured in eight sections.

3.0 Introduction

Digital Audio decoders are becoming very popular devices in portable players. At the same time, these audio decoders are also involved in Digital Radio systems as one of the block of the receiver system.

Companies like ST are considering Digital Radios a good business opportunity. Even if this business is in the starting phase, there are countries where these systems are already growing very fast, for example XM Radio and Sirius in the USA, DAB in UK and other European countries.

If ST wants to compete and win this market, it has to be able to develop the right devices to support all the new functionalities required by the market.

Consider digital radio as the sum of three main tasks: Receiver, decoder and system controller this paper will address the last two.

The features required for the audio decoding and the system controlling are:

- High instruction throughput to implement the decoding phase of all the available digital audio formats such as mp3, wma and aac+, and to be powerful enough to decode the future evolution and new algorithms.

- High capability of data management and storage. It has to control most of the available on the market external memories: SDRAM, NAND flash, Secure digital, Multimedia Cards and IDE devices.

- Highly flexible user interface and general-purpose system controller. It has to integrate all the needed standard interfaces: synchronous and asynchronous serial interface, master and slave I2C controller, GPIO lines, keypad controller, USB and so on, to control and communicate with the receiver devices.

- Low power consumption. This requirement is becoming more and more important especially on portable devices.

The idea of the cooperation between ST APG-DBR and Synopsys SPS is not to develop a general-purpose architecture but to implement a methodology, which is capable to optimize a device starting from a customer specification in a minimum time. Architectural comparison requires a lot of time and effort to build all the needed systems and to perform the wanted evaluation. To reduce the complexity of this step a new flow has been developed using CoreAssembler, AMBA DesignWare and VIP.

In the following lines, we will try to approach the main steps executed in the developed flow to compare different AMBA based architecture with the goal of identifying the best solution that fits the customer specifications. In the next paragraphs, you will find more details on each of the following points.

First step is to identify all the IPs needed to implement the specification requirements. Usually at the end of this phase, we find in a solution using custom and third party IPs.

Second step is to define architectures implementing all the selected IPs. By using CoreAssembler together with the AMBA DesignWare®, we are able to quickly build all the architectures needed by the designers without losing any time for debugging errors during the bus generation and the system connectivity. To be able to use non DesignWare® IPs in coreAssembler, a package activity within coreBuilder has been performed.
The result of this step is the possibility to build, with a minimum effort, all the wanted or needed architectures.

Third step of our flow is dedicated to building a test bench able to stress the different systems produced in the step two. This test bench is used to identify the pro and cons of all the developed architectures. Our traditional way to perform this task was to run a piece of software. Everyone knows that the ideal test is the real application, but at this point, that code is really far from being available. We approach this problem by identifying the possible critical points of the application and trying to write pieces of code that are able to stress the identified critical points. Synopsys improved our traditional verification flow with the DesignWare Verification IP (DW_VIP) technology.

3.1 Architecture implementation

To understand the flow described on this paper, a real system developed by ST APG-DBR will be discussed.

The requirements of the customer for this device are the following:

- Audio decoder capability: The device has to provide enough MIPS (70 is the fixed value) to perform generic audio decoding.
- System controller capability: The device has to control the Receiver System through an I2C bus. Between the device and the receiver there should be a powerful communication channel in order to allow bi-directional data exchange.
- Storage capability: The device has to provide different solutions for data storage. Both volatile and not volatile devices have to be controlled.
- Powerful user interface: the device has to provide the set of common features available on portable players.
- Low power consumption .

To satisfy the request of a powerful user interface a lots of IPs have been implemented. A keypad controller allows the user to use a keypad with up to 32 keys. A Real time clock (RTC) clocked by external crystal or internal frequency is implemented to be used as a clock and also to perform some power-up and power-down features. A USB 1.1 interface adds the capability of downloading data from a PC or any USB Host. An IRDA interface manages an infrared data exchange between the system and the external world. A simple A2D (7 effective bits) can be used to monitor battery level or supply values. To complete the user interface 24 GPIO lines have been added and two different serial protocols (synchronous and asynchronous) are supported.

The MicroDrive is becoming quite common in portable players; these hard disks offer a high storage capacity with a very small dimension. The system being developed controls IDE devices including MicroDrive- supporting IO, DMA and UDMA data transfer.
If MicroDrive is the most interesting solution to satisfy high storage capability, NAND flash is becoming the better tradeoff between storage capability, dimension, consumption, performance and cost. To support this kind of memory a NAND flash controller has been implemented in this platform.
The IDE device and NAND flash cover the needs of high storage capability but what is still missing is the possibility to have removable storage devices. Memory cards complete the needs of our applications. The cards controlled by the system are Secure Digital, Multimedia and compact flash.
To be as flexible as the market requires in terms of controlling the memory, a flexible memory controller has been integrated to control synchronous SDR- DRAM as well as static memory SRAM NOR flash and ROM and It can also be configured to control memory mapped devices.
Considering the system as a general-purpose system controller an I2C master interface is available to guarantee the control of all the I2C slave devices. A data communication channel between the devices and the slaves has been designed to support most of the common serial protocols. This channel allows the exchange of data with the controlled devices.

The requirement to realize an Audio decoder performing up to 70 MIPS together with the spec of a low power device pushed us to choose an ARM7TDMIS core. Pros of this core are the small area and the low power consumption, the con is the difficulty to reach a high frequency. The challenge of having audio decoding together with controller capability and data managing drove us to consider multi layer architecture. To reach the 70 MIPS the system runs at 80 MHz and a separate bus AHB1 is dedicated to the core in order to have a Lite AHB wrapper able to improve the MIPS for the MHz provided by the core. Data storing can be performed at the same time of audio decoding without affecting the performance using the second layer AHB2. Master of this bus is a powerful and flexible DMA controller.
Buffer of internal memories are available on each bus. Program Code, being that the ARM7TDMI is not a cached core, has to be stored in the internal ROM, but the possibility to execute code from internal or external RAM are also available.
To reduce power consumption clock control logic has been implemented. This logic provides the features of changing the system frequency dynamically following the requirement of the piece of program code executed at that time.

Figure 1. Illustrates the implemented system:

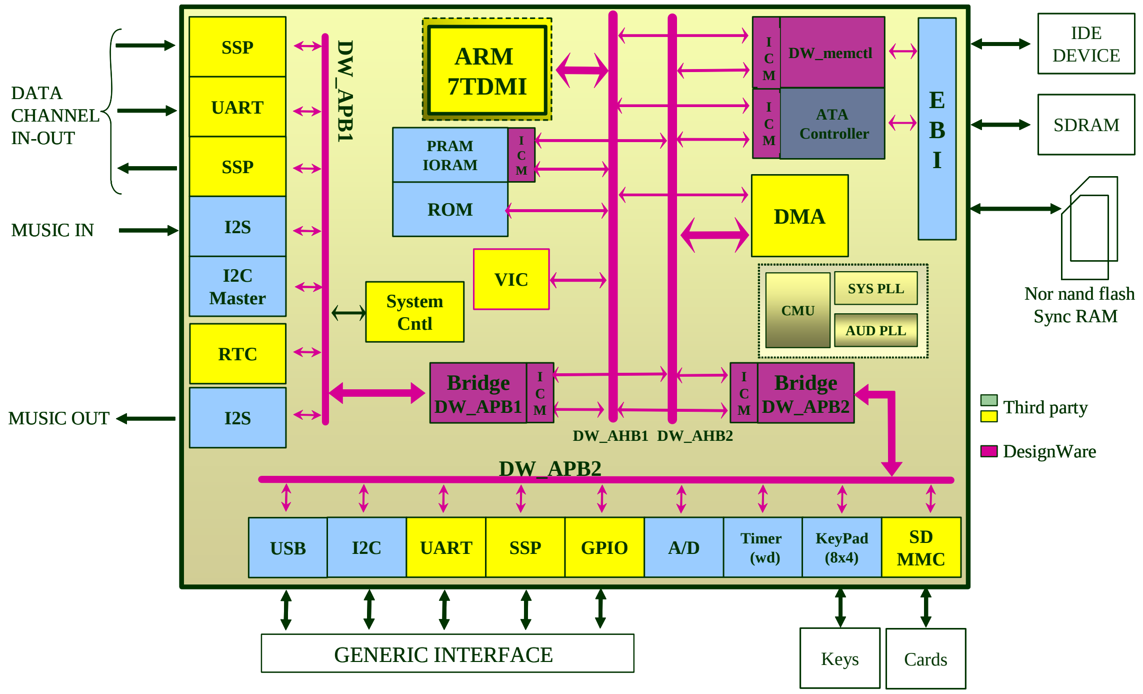


Figure 1: Digital Radio System Controller & Audio Decoder overview

4.0 IP Reuse and System Integration

4.1 IP challenge

The main objectives in creating and delivering a reusable IP can be summarized as follows:

- Easily configurable to fit different applications
- Designed for use in multiple technologies
- Thorough commenting
- Well designed verification environments and suites
- Robust scripts
- Easy to use and friendly implementation interface
- Good documentation

The methodology described here addresses most of the above.

As an IP developer, you need to deliver an IP that can reliably be used by the integrator. The IP has to be easy to support and maintain, enabling long-term maintenance by capturing design knowledge. The business solution, which required direct support and a one-on-one knowledge transfer with every end user for an IP provider, would not be a viable one. The integration flow of an IP has to reduce all costs (not only the development cost but also synthesis, testing, integration in SoC, support and maintenance). In most cases, the end users want to customize the IP for specific applications. In order to achieve this effectively, the creation of every stage of design, from specification to silicon has to be done with the understanding that it will be modified and reused in other projects by other design teams. It needs to use tools and processes that capture the design information in a consistent, easy to communicate form, and that make it easy to integrate modules into a design when the original designer is not available. Eventually, it should also force the integrator to go through specific tasks to

synthesize and verify the core. The integration flow has to ensure quality of results, ease of use and tool support over multiple versions or licenses. The package has to provide all the necessary views and tests across all possible parameter values. This core has to be not only easily configurable but also technology independent. Of course, the protection of your IP is an important requirement for the developer.
Finally, this IP has to be easily integrated in an IP library and easy to be checked for reliability.
The innovation of core tools is to address all the previous needs: it has to provide industry-leading tools that enable our users to create, package, deploy, integrate and assemble configurable soft IP.
The solution can be found in the new methodology offered by Synopsys's suite of reuse tools.

4.2 coreAssembler

The coreAssembler (cA) tool simplifies the process of assembling a subsystem of components by automating the configuration and interconnection steps, providing an automated path to implementation for the entire subsystem, and providing a starting point for verification.

In the coreAssembler environment, the user is guided through the integration process using an "activity" based methodology. These activities are undertaken sequentially, and where necessary the tool ensures that all steps essential for later integration work are completed before the next activity can be initiated.
One of the main features in coreAssembler is to automatically generate the top-level VHDL or Verilog code for the subsystem.

Creating the system's RTL code is divided into the following, relatively intuitive, steps shown in Figure 2.

- Adding the system components.
- Configuring the system components.
- Completing the connections.
- Generating the system RTL.

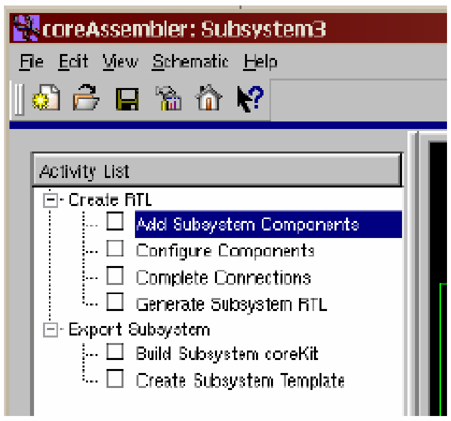


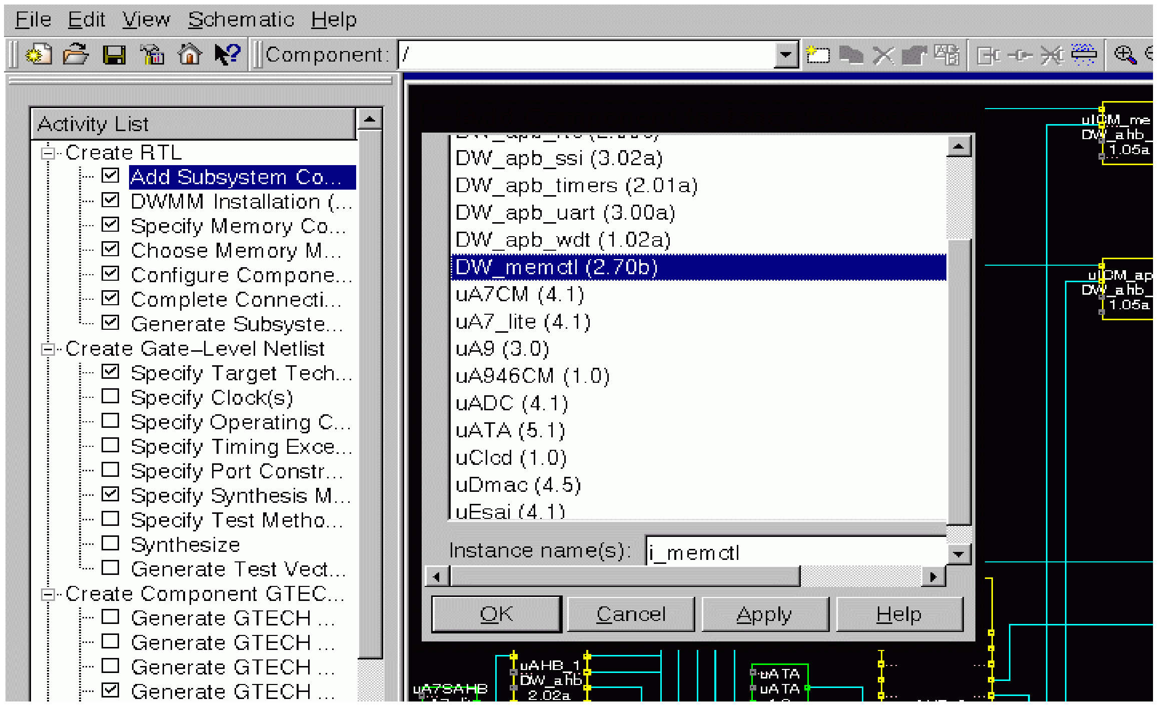
Figure 2: coreAssembler's Activity List

4.2.1 Adding the system components

Three different kinds of components have been used in this project; AMBA DesignWare®, ST custom and third party components.

AMBA DesignWare® components

The Synopsys DesignWare® AMBA Synthesizable Components environment is a parameterizable bus system containing AMBA version 2.0-compliant AHB and APB (Advanced Peripheral Bus) components.
Pre-design, pre-verified and fully technology independent DW IPs is delivered as coreKits.
By setting the tool's search path variables pointed to the DW home directory, all available modules should be accessible via a convenient, browseable location within the cA environment shown in Figure 3.



Activity List
- Create RTL
  - ☑ Add Subsystem Co...
  - ☑ DWMM Installation (...
  - ☑ Specify Memory Co...
  - ☑ Choose Memory M...
  - ☑ Configure Compone...
  - ☑ Complete Connecti...
  - ☑ Generate Subsyste...
- Create Gate–Level Netlist
  - ☑ Specify Target Tech...
  - ☐ Specify Clock(s)
  - ☐ Specify Operating C...
  - ☐ Specify Timing Exce...
  - ☐ Specify Port Constr...
  - ☑ Specify Synthesis M...
  - ☐ Specify Test Metho...
  - ☐ Synthesize
  - ☐ Generate Test Vect...
- Create Component GTEC...
  - ☐ Generate GTECH ...
  - ☐ Generate GTECH ...
  - ☐ Generate GTECH ...
  - ☑ Generate GTECH ...

DW_apb_ssi (3.02a)
DW_apb_timers (2.01a)
DW_apb_uart (3.00a)
DW_apb_wdt (1.02a)
DW_memctl (2.70b)
uA7CM (4.1)
uA7_lite (4.1)
uA9 (3.0)
uA946CM (1.0)
uADC (4.1)
uATA (5.1)
uClcd (1.0)
uDmac (4.5)
uEsai (4.1)

Instance name(s): i_memctl

Figure 3: The Add Subsystem Components Activity View

DesignWare (DW) IPs have pre-attached interfaces, which provide instructions to the tool on how to connect a system component based on AMBA 2.0 standard. By adding these IPs to the system, cA will automatically connect them together.

The DW components that have been used in this project are:

- 2 x DW_AHB
- 2 x DW_APB
- 5 x DW_ICM (Inter Connect Matrix)
- 1 x DW_memctl

Custom and third party IP

There are two different ways to add these kinds of IPs to the system.
The first way is to import the component and then create and attach the interfaces. This method can be used if a component has not been packaged using the coreBuilder. The

second way is to package the IP within coreBuilder and install it as coreKit into the coreAssembler where the interfaces are already being created (see Figure 4). The advantage of the second choice is that the same IP can be used to create different workspace versions without recreating the interfaces every time a new workspace is needed.
This reason was enough to choose packaging alternative for this project.

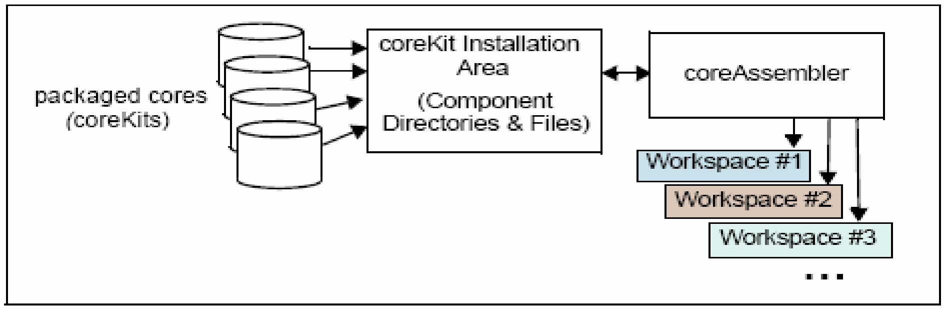


Figure 4: coreKit and Workspaces

Packaging with coreBuilder
One part of the complete set of IP reuse tools available from Synopsys is coreBuilder, which can package the core into a coreKit with all of the associated design files such as verification files, test bench files and the documentation for the core. This provides a convenient format for the distribution, installation, and integration into the core integrator's design and verification environment. For this project, it was decided to use only few limited features of this tool in order to have a coreKit version with all interfaces of needed custom and 3rd party IPs.
To create the interfaces, there is a specified activity in coreBuilder where the interface definition needs to be defined. For this particular project that is an AMBA based design the DW_interface_definition.tcl file, which is AMBA compliant, either AHB or APB has been defined.
Like modules, interfaces also have ports, which reflect module ports in them. Their width can be parameterizeable, or their inclusion made optional based upon individual requirements. However when the port is created, instead of a typical 'input' or 'output' declaration being made, the direction is specified in terms of "Provider" or "Consumer".

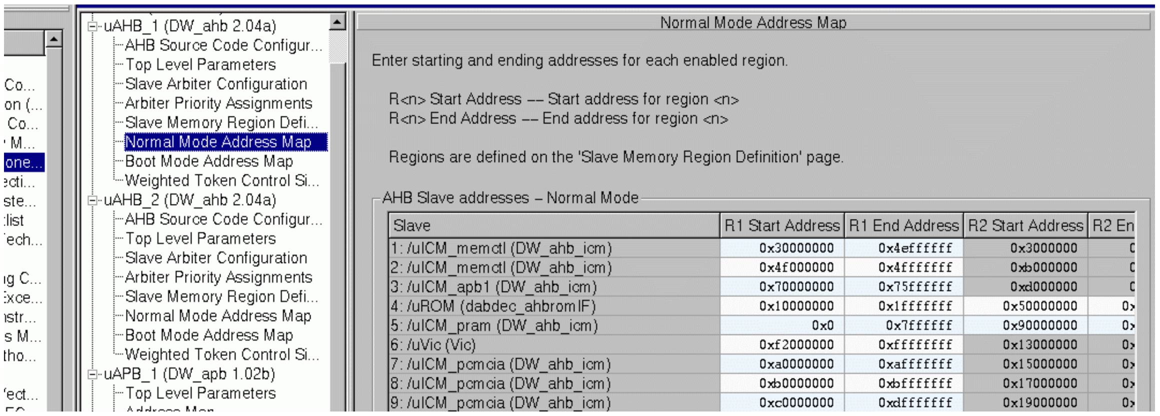
At this point in the integration process, it is also necessary to perform system-level configuration. This involves specifying parameters (through the relevant interfaces) that are important to multiple modules, such as AHB bus widths. These values, which require setting only once, are then propagated downwards to the component level where they are no longer modifiable on an individual basis, thus preventing inconsistency.

Once you have successfully added all components and configured your interfaces, the "Configure Components" activity can begin.

4.2.2 Configuring the system components.

The Configure Components activity is component-specific and provides one top-level entry per component in a tree dialog. You need to review and update component

parameters as needed for the proper configuration of your subsystem i.e. AHB memory map can easily be configured as shown in Figure 5.
Some component configuration parameters are set automatically based on values of interface parameters. These parameters will appear disabled in the component configuration dialogs.



uAHB_1 (DW_ahb 2.04a)
AHB Source Code Configur...
Top Level Parameters
Slave Arbiter Configuration
Arbiter Priority Assignments
Slave Memory Region Defi...
Normal Mode Address Map
Boot Mode Address Map
Weighted Token Control Si...
uAHB_2 (DW_ahb 2.04a)
AHB Source Code Configur...
Top Level Parameters
Slave Arbiter Configuration
Arbiter Priority Assignments
Slave Memory Region Defi...
Normal Mode Address Map
Boot Mode Address Map
Weighted Token Control Si...
uAPB_1 (DW_apb 1.02b)
Top Level Parameters
Normal Mode Address Map

Enter starting and ending addresses for each enabled region.

R<n> Start Address -- Start address for region <n>
R<n> End Address -- End address for region <n>

Regions are defined on the 'Slave Memory Region Definition' page.

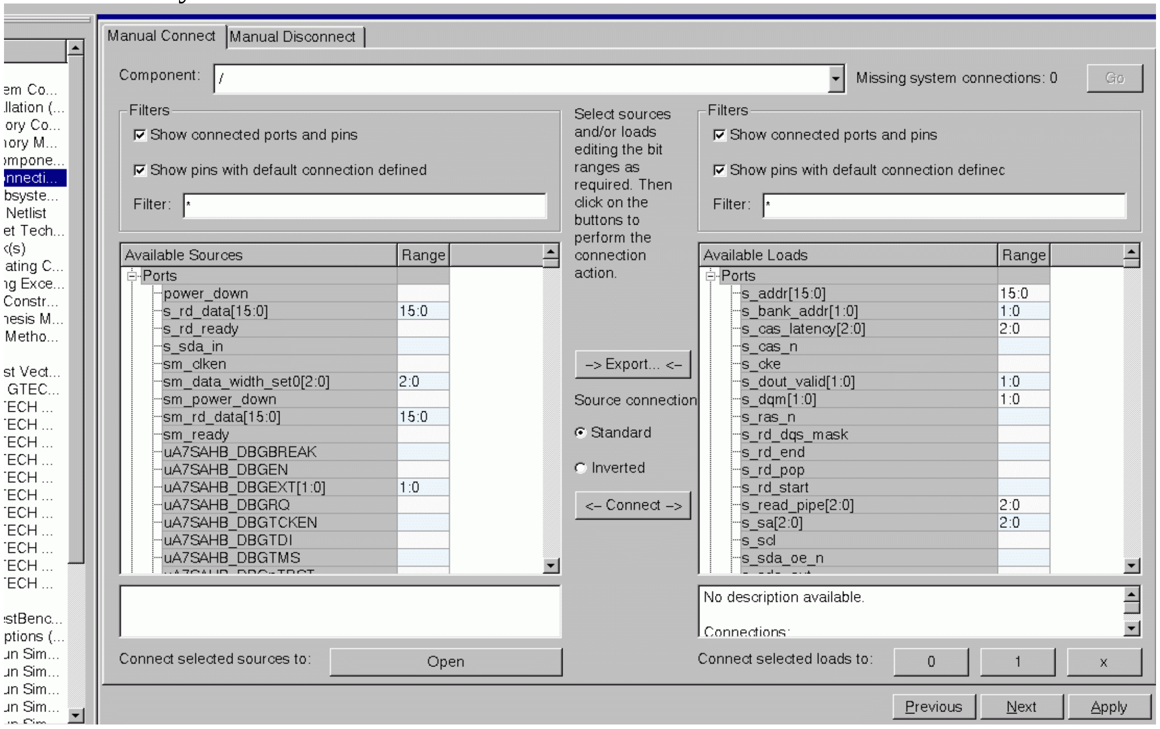
AHB Slave addresses – Normal Mode

Slave	R1 Start Address	R1 End Address	R2 Start Address	R2 En
1: /uICM_memctl (DW_ahb_icm)	0x30000000	0x4effffff	0x3000000	C
2: /uICM_memctl (DW_ahb_icm)	0x4f000000	0x4fffffff	0xb000000	C
3: /uICM_apb1 (DW_ahb_icm)	0x70000000	0x75ffffff	0xd000000	C
4: /uROM (dabdec_ahbromIF)	0x10000000	0x1fffffff	0x50000000	0x
5: /uICM_pram (DW_ahb_icm)	0x0	0x7ffffff	0x90000000	0x
6: /uVic (Vic)	0xf2000000	0xffffffff	0x13000000	0x
7: /uICM_pcmcia (DW_ahb_icm)	0xa0000000	0xafffffff	0x15000000	0x
8: /uICM_pcmcia (DW_ahb_icm)	0xb0000000	0xbfffffff	0x17000000	0x
9: /uICM_pcmcia (DW_ahb_icm)	0xc0000000	0xdfffffff	0x19000000	0x

Figure 5: Configure the memory map of AHB bus

4.2.3 Completing the connections.

This activity is to connect any remaining pins that were not automatically connected (see Figure 6). Unconnected input pins can be connected to unconnected output pins, tied off to a constant value, or exported from the subsystem (that is, connected to an automatically created input port of the subsystem). Unconnected output pins can be connected to an existing input pin, explicitly marked as unconnected (open), or exported from the subsystem.



Component: /
Missing system connections: 0

Filters
[✓] Show connected ports and pins
[✓] Show pins with default connection defined
Filter: *

Select sources and/or loads editing the bit ranges as required. Then click on the buttons to perform the connection action.

Source connection
[✓ Standard]
Inverted

Available Sources	Range
Ports	
power_down	
s_rd_data[15:0]	15:0
s_rd_ready	
s_sda_in	
sm_clken	
sm_data_width_set0[2:0]	2:0
sm_power_down	
sm_rd_data[15:0]	15:0
sm_ready	
uA7SAHB_DBGBREAK	
uA7SAHB_DBGEN	
uA7SAHB_DBGEXT[1:0]	1:0
uA7SAHB_DBGRQ	
uA7SAHB_DBGTCKEN	
uA7SAHB_DBGTDI	
uA7SAHB_DBGTMS	

Connect selected sources to: Open

Filters
[✓] Show connected ports and pins
[✓] Show pins with default connection definec
Filter: *

Available Loads	Range
Ports	
s_addr[15:0]	15:0
s_bank_addr[1:0]	1:0
s_cas_latency[2:0]	2:0
s_cas_n	
s_cke	
s_dout_valid[1:0]	1:0
s_dqm[1:0]	1:0
s_ras_n	
s_rd_dqs_mask	
s_rd_end	
s_rd_pop	
s_rd_start	
s_read_pipe[2:0]	2:0
s_sa[2:0]	2:0
s_scl	
s_sda_oe_n	

No description available.

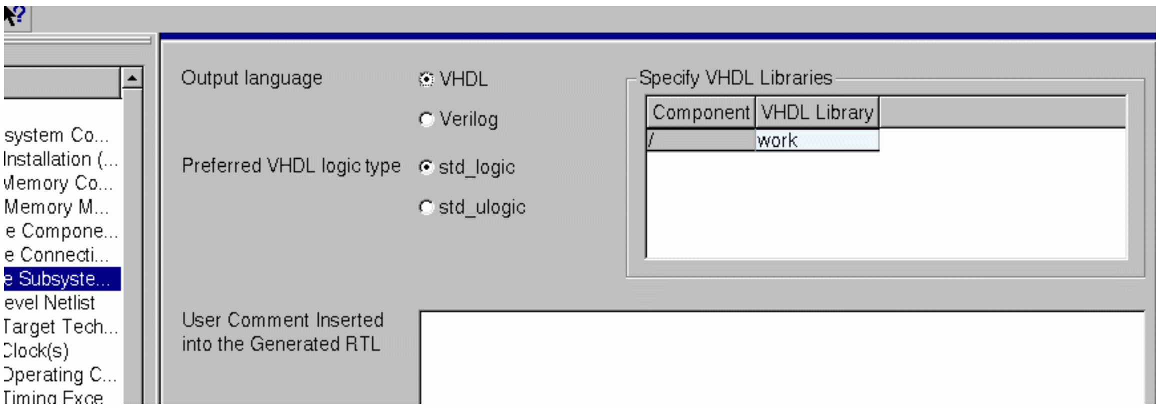
Connections:

Connect selected loads to: 0 1 x

Figure 6: Complete Connection Activity View

4.2.4 Generating the system RTL.

At this point in the integration process, the subsystem is ready and by clicking the [**Apply**] button, the top-level RTL code for the subsystem will be created as shown in Figure 7 .
By clicking the Apply button in this dialog, coreAssembler (cA) will write the top-level HDL code to the file *<workspace>/src/<design>.v[hd]* where *<workspace>* is the current workspace name and *<design>* is the subsystem top-level design name.



Output language: [✓ VHDL] Verilog

Preferred VHDL logic type: [✓ std_logic] std_ulogic

Specify VHDL Libraries

Component	VHDL Library
/	work

User Comment Inserted into the Generated RTL

Figure 7: Generate Subsystem RTL Activity View

5.0 System Verification

Processor complexity, custom logic content and system performance are all increasing at the same time that schedules are being squeezed and resources are stretched. Today's IC and System-on-Chip (SoC) design trends have placed an immense burden on the shoulders of verification engineers. The now-famous Collett study shows that 70% of project effort for complex ICs is spent on verification. Over 60% of design re-spins are caused by functional errors.
Not surprisingly, project teams are looking for more effective methods to verify their designs. The need for these methods is driven by complex protocols and IPs, ambiguous or misleading specifications, unanticipated or untested usages and corner case behaviors. Verification engineers are consequently looking towards new methodologies that reduce test bench development time, and speed-up the time it takes to achieve complete verification of their ASIC or SoC.

System verification has changed dramatically in past years, from using bus functional models, driven at pin or bus cycle level, to yet higher levels of abstraction that include constrained random test methodologies, typically found in high-level verification languages. The new enhanced verification models use their protocol knowledge to drive transactions onto the bus within the user defined application specific needs of the design. Additionally a divide and conquer strategy allows for an incremental verification strategy by using standards as boundaries to partition the verification problem.
Reusability and smart verification techniques are key elements of state of the art verification strategies.

5.1 Traditional flow

Manually creating vectors to accurately reflect system behavior was major task of traditional flow. System complexity allowed concentrating on building test cases and even building verification models.

5.2 Synopsys DesignWare® Verification IP approach

New enhanced verification models use their protocol knowledge to drive transactions onto the bus within the user defined application specific needs of the design. They allow engineers to build adaptive, reactive test benches that preclude the drudgery of directed tests and hard-to-maintain 'golden' reference files. Creating a test suite now becomes significantly simpler for verification engineers who do not need to spend time learning the details of the protocol and weeks or months writing a directed test suite. The advantage of using these enhanced models is immense and results from both simplified usage and improved coverage.
A key element of such verification models are constrained random test capabilities, which allow engineers to rapidly test their designs across a range of parameters and assist in creating test benches that are adaptive and reactive. Instead of specifying each individual event to exercise the design, the engineer specifies ranges within which the test bench then exercises the target device. In addition, the possibility to access information of every transaction by usage of notifications can be counted as an element. Feedback from monitors and models identify test suite hits and allows the test bench to adapt and check new areas. This new functionality in the models replaces much of the effort associated with manually creating vectors to accurately reflect system behavior. Functional coverage reports are written automatically. Thus, statistics of the simulation are available without spending effort in programming such functionality.

Since the verification IP is language independent, it can be used in VHDL, Verilog and VERA environments. It can be applied on different strategies:

- Block level test bench written by the designer in Verilog or VHDL
- Sub-system test bench written by verification team in HVL, C, Verilog or VHDL

This language neutrality eases reuse and debug.

5.3 Mapping the DW Verification IP (VIP) approach on the system.

A number of verification scenarios needed to be implemented in order to do system verification for the current system shown in Figure 1. Besides creating sufficient test scenarios, it is essential the VIP approach, not break the ST-used verification method. Thus VHDL was used as the test bench language. Reuse of already existing test benches and applying only minor changes was essential.

To generate "realistic" traffic on the AHB/APB busses, the ARM 7TDMI was exchanged with a VIP AHB master, as depicted in Figure 10. By using constrained random test capabilities, different peripherals on AHB and APB busses can be accessed in a wide range. To use constrained random test features the user needs to define a protocol transaction generator, PTG as shown in Figure 8. The shown VHDL command creates equally distributed transactions of type write and read for the address region `32'h98000000:32'h9801FFFF=1.`

```
-- reads and writes form/to IORAM
ahb_master_vmt_pkg.new_ahb_master_ptg ("ARM_master",ptg_AHB_1_IORAM_RW,
                              &"XFER_TYPE WRITE=1,READ =1"
                              &"XFER_SIZE 32=1, 16=1, 8=1; "
                              &"BURST_TYPE INCR=1;"
                              &"NUM_BEATS  1:4=1;"
                              &"ADDRESS 32'h98000000:32'h9801FFFF=1;");
```

Figure 8: Definition of a protocol transaction generator

All transaction created are of undefined INCR type with a number of beats between 1 and 4. Using the above method different transaction types were created for the IORAM, the PRAM, the ROM and the USB interface on APB bus 1. The given method frees the user from knowing the AHB2.0 specification in detail. The user can instead concentrate on creating test cases matching its needs. To issue transactions for the listed peripherals the user additionally needs to define a distribution for those as well.

```
-- define a tcg to chose between all possible reads and writes
ahb_master_vmt_pkg.new_tcg("ARM_master",tcgARM);
ahb_master_vmt_pkg.add_tcg_item ("ARM_master",tcgARM,ptg_AHB_1_ROM_R, 20);
ahb_master_vmt_pkg.add_tcg_item ("ARM_master",tcgARM,ptg_AHB_1_PRAM_RW,
20);
ahb_master_vmt_pkg.add_tcg_item ("ARM_master",tcgARM,ptg_AHB_1_IORAM_RW,
20);
ahb_master_vmt_pkg.add_tcg_item ("ARM_master",tcgARM,ptg_AHB_1_USB_RW, 20);
ahb_master_vmt_pkg.add_tcg_item ("ARM_master",tcgARM,tsgDMA, 20)
```

Figure 9: Defining a distribution for different protocol transaction generators.

In the shown case, Figure 9, all peripherals will be accessed equally often. To do so, a transaction choice generator (tcg) is used. To track all transactions on the busses, checking

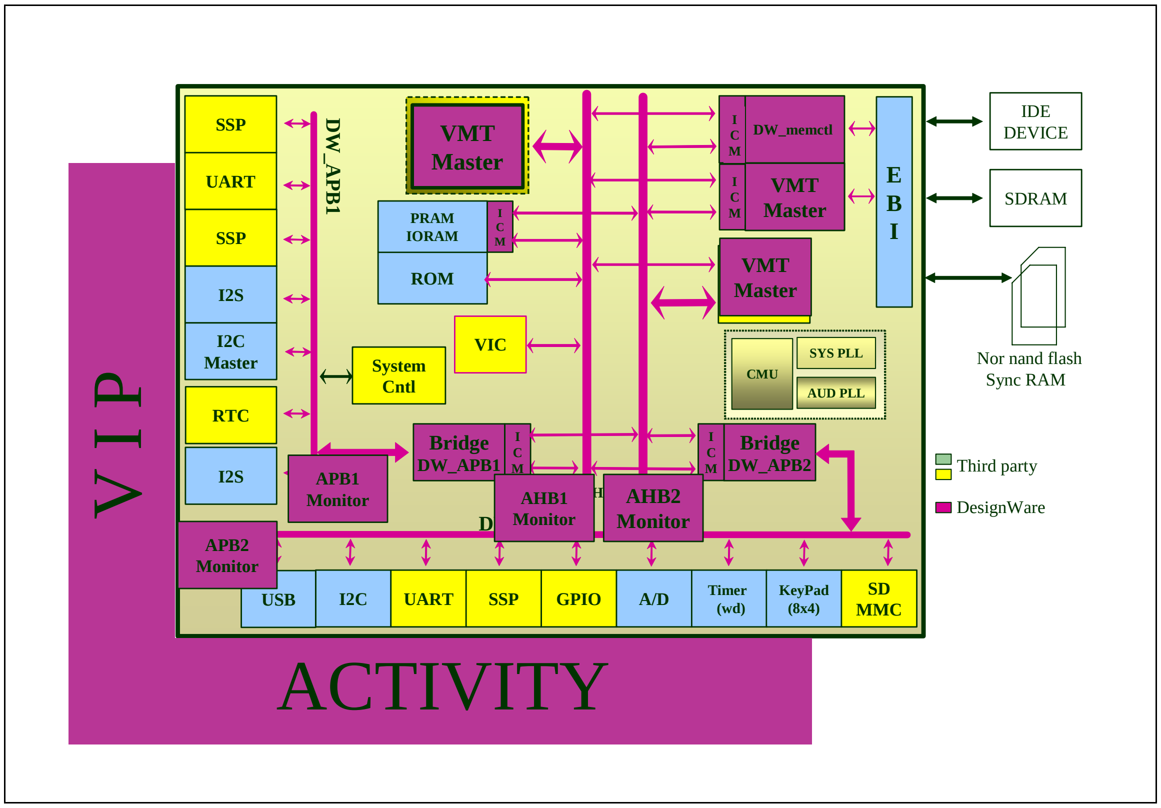


Figure 10: Verification IP applied on the system

for protocol conformance and creating coverage reports, monitor models have been attached to the AHB and APB busses. The Monitor can be configured for any valid AHB bus configuration from 8 to 1024-bits. The monitor connects to the actual AHB signals, as defined in the AMBA Specification. There is one monitor-per-AHB bus, which is capable of supporting up to 16 masters and 16 slaves. The monitor can control protocol checkers, which can be stopped and started using configuration parameters. Functional coverage is used to monitor particular states on the bus. Errors, logged by the monitor, are also tracked by the coverage object. Statistical reports are available for both valid and error states. This allows the monitor to be used to help ensure verification tests are covering the complete functionality. Functional coverage data is saved and restored for incremental coverage reporting.

Major effort has been spent on imitating the DMA by usage of DesignWare verification IP (VIP) models. A master and a slave VIP AHB model were used to rebuild the DMA. A certain slave region of the DMA slave was used to acknowledge a start of a DMA transaction. For that the replacement of the ARM 7TDMI (VIP AHB master) needed to issue a transaction sequence to notify a DMA transaction. This was achieved by using transaction sequence generators (tsg) as shown in Figure 11. This transaction sequence generator became an integral part of all AHB transactions, depicted in Figure 9. As a result, 20% of all AHB transactions issued by the ARM 7TDMI replacement were DMA transfers.

```
--define a tsg for writing instructions to the DMA
  ahb_master_vmt_pkg.new_tsg ("ARM_master",tsgDMA);
  ahb_master_vmt_pkg.add_tsg_item ("ARM_master",tsgDMA,ptg_DMA_7);
  ahb_master_vmt_pkg.add_tsg_item ("ARM_master",tsgDMA,ptg_DMA_0);
  master[1].set_tg_payload(tsgDMA, payload);
```

Figure 11: Definition of a transaction sequence generator.

Once the DMA slave recognized a DMA transaction notification, the DMA master initiated a READ from IORAM and a WRITE with same data to the PRAM. Figure 12 shows the VHDL source code for the DMA rebuilt. An endless DMA loop ensures that during whole simulation the DMA slave is looking for the DMA transaction notification sequence. After recognition, a read_burst command is used to read data from IORAM and afterwards a write_burst command is used to write the just read from the PRAM. Using the above method, system verification and bus utilization could be achieved without having the actual DMA implementation in place.

```
-- create a burst buffer for DMA data transfer
ahb_master_vmt_pkg.new_burst_buffer("DMA_master_ahb2", 32,
 VMT_MEM_PATTERN_INCR, to_stdlogic(0,1024), 4, MA_burst_buffer_hanle);
ahb_master_vmt_pkg.set_burst_buffer_xfer_attr("DMA_master_ahb2",
 DMA_burst_buffer_handle,1, DMA_DataAttr);


DMA_LOOP:loop

  WAIT UNTIL hclk'event AND hclk = '1';
  WAIT UNTIL hclk'event AND hclk = '1';

  -- check if ARM master has sent appropriate instructions
  ahb_slave_vmt_pkg.get_mem("DMA_slave_ahb1",DMA_sid, DMA_Address, 32,
                          DMA_get_mem_buffer);
  write(DMA_l, STRING'("  ******** DMA_get_mem_buffer = "));
  write(DMA_l, STD_LOGIC_VECTOR'(DMA_get_mem_buffer));
  writeline(output, DMA_l);

  if DMA_get_mem_buffer = to_stdlogic(7,1024) then
  -- "111" to establish DMA transfer
     write(DMA_l, STRING'("  ******** DMA is wrting data from PRAM to
           IORAM "));
     writeline(output, DMA_l);

     -- read from IO RAM and write same values to PRAM

     ahb_master_vmt_pkg.read_burst("DMA_master_ahb2", DMA_sid,
               DMA_IORAM_address, 4, DMA_burst_buffer_handle,
               DMA_read_burst_buffer_handle);
     ahb_master_vm2t_pkg.get_burst_result("DMA_master_ahb2", DMA_sid,
                DMA_read_burst_buffer_handle, DMA_read_buffer);
     ahb_maste_vmt_pkg.write_burst("DMA_master_ahb2", DMA_sid,
               DMA_ARM_Address, 4, DMA_read_buffer, DMA_write_buffer);


  end if;

 end loop DMA_LOOP;
```

Figure 12: DMA functionality rebuilt by usage of VIP AHB master and slave

Besides the described verification scenarios, extensive transaction logging of the monitors was used to do post simulation analysis of bus utilization. A TCL based approach was applied.

6.0 Results

As a result, STMicroelectronics is finding that even the most complex subsystem requires less in-depth knowledge of the IP, and also reduces the configuration, integration and verification time. Usage of the Synopsys' AMBA DesignWare® even decreases the silicon area by not including the unneeded and unused features.

7.0 Acknowledgments

At the end, we want to point out that the achievement of the results described in this paper has been possible thanks to the close cooperation between the STMicroelectronics and the Synopsys Professional Services division, whose contribution to the development of the new flow has been fundamental to shorten the time and to increase the quality of the work.

8.0 References

[1] Reuse Methodology Manual
[2] coreAssembler User Guide