

Designing, Verifying and Building an Advanced L2 Cache Sub-System using SystemC

Thomas Tessier
Vice President of R&D
Panève, LLC.

Dr. Hai Lin
Chief Chip Designer
Panève, LLC.

Daniel Ringoen
Sr SOC Engineer
Panève, LLC.

Eileen Hickey
Sr Verification Engineer
Panève, LLC.

Steven Anderson
Sr FAE
Forte Design Systems

Abstract:

System on a chip designs contain increasingly complex modules which necessitate ever more complex interfaces. This dual increase in complexity has made it much more difficult for both designers and verification engineers to complete their tasks. Fortunately, advances in verification methods, design tools, languages and vendor IP can have a major impact in reducing the difficulty of the resulting research and development process.

This paper will detail our experience with a small team development effort for an advanced programmable, multithreaded, multicore processor targeted at streaming video applications. The system as tested, would create a 10M+ gate ASIC with 10 Mbytes of on-chip RAM. We will be focusing on that portion of the chip which implements a complex L2 Cache which maintains coherency between multiple cores. The L2 cache is connected with wide interfaces to both an On-Chip-Bus, and to instruction and data L1 Caches. A system of this complexity simply cannot be successfully implemented without extensive verification. Having appropriate simulation models for the interfaces is one of the keys to success. Of course, it is also necessary for those simulation models to be transformed into optimal hardware components in order to follow a “synthesize what you verify” methodology.

The source code and testbenches for this chip were coded entirely in SystemC. This model was used for extensive design verification as a behavioral simulation model. It was then synthesized into Verilog RTL using a commercial High Level Synthesis tool, and the resulting RTL models were re-simulated with the same test infrastructure to verify equivalency between SystemC and Verilog models. Finally, testbenches were deployed to the prototype FPGA along with design code for further in-system verification.

We will discuss in detail the use of commercial interface IP provided by the HLS vendor. The interfaces are templates which can be configured to transfer a complex

data structure comprised of many separate fields. Some examples of the interfaces used are handshaking point-to-point, clock domain crossing, memory based FIFO's and register based FIFO's. User read and write access can be blocking or non-blocking. Chains of interfaces can also be built into a stall-able pipeline. All of the interfaces include TLM and PIN level specializations so that a single set of user source code can be used for TLM or PIN level simulations. This same TLM source code can be then synthesized to RTL, which was then taken through a typical RTL design flow to build hardware.

Having these pre-verified configurable interface blocks not only simplified the design process it also helped us successfully verify the system. The initial prototype is undergoing test in a Virtex 6 device and work is commencing to target a commercial ASIC with a full system.

Introduction

This paper will describe some of our experience with a development effort for an advanced multi-core, multi-threaded processor with the focus on the development of the L2 Cache. A fully configured chip can have multiple processing units connected via a coherent on-chip L2 cache. This paper focuses on the implementation and verification of the portion of the chip which implements a complex L2 Cache including the required interfaces to a local high speed External Bus and L1 Cache. The local high speed External Bus would also connect to various external interfaces such as DDR3, Flash, Ethernet and DVI.

The source for this chip was coded entirely in SystemC. This model was used for extensive design verification as a behavioral simulation model as well as for high-level synthesis using a commercial tool from Forte Design Systems to create scheduled Verilog RTL. The generated RTL code was taken through a typical RTL design flow and implemented on a Xilinx FPGA [1] as a proof of concept. The next step will be to implement a complete multi-core processor in an ASIC foundry process.

The core processing units were coded as single cycle methods to give us complete control over the data pipeline. Even at this low level of abstraction the advantages of C++ templates gave us very compact code. The L2 cache and external interfaces of the chip were written using multi-cycle SystemC threads with interfaces and memories provided as part of the Cynthesizer tools from Forte [2]. A higher level of abstraction was therefore possible for these parts resulting in additional benefits.

We extensively used Cynthesizer interface IP based on a simple ready / valid handshake for point to point signaling (P2P). The P2P signaling protocol can reliably transfer complex data structures between processing elements. In addition, we used Cynthesizer memory model IP mapping to either Xilinx Block or distributed RAM. Both the interface IP and the memory IP take advantage of C++ mechanisms to encapsulate the protocols providing consistent APIs for us to use and reducing the coding required for interconnection.

In addition to the design advantages we found with this methodology, we will discuss some of the verification advantages and challenges we met. We will also take some time to suggest areas for future improvements. But first let's take a look at the processor architecture.

H/W Architecture

Diagram 1 shows an example of 2 processor cores, L1 cache, our L2 cache structure plus interfaces to our External Interface (EXT) local high speed bus. The basic processing element in most of the modules is a clocked thread. Inside each individual thread, the computation loop may be scheduled in one or more cycles. The processing loop in some of the threads has been pipelined where needed to increase throughput.

Diagram 2 shows a fairly detailed block diagram of our L2 cache. All of the interfaces in this block were built with P2P channels. The data types for those channels, shown in the bubbles, are complex structures with any number of data fields.

In our system the L2 interface is responsible for interrogating all packets on the EXT to decide whether it needs to acquire that packet and process it. The latency from EXT request to EXT response must be done in two cycles which necessarily includes a TAG lookup. This extremely low latency could only be met by running the memories on a faster clock than the EXT. This was the tightest latency within the L2 Cache Memory system. We used the various controls provided by Forte to ensure that

the design had minimum latency but met our timing budget.

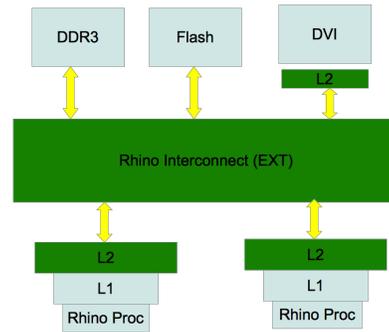


Diagram 1 – System Architecture

Our External Interface allows multiple processors to request access. Each processing unit can put out a request and continue processing other threads while waiting for

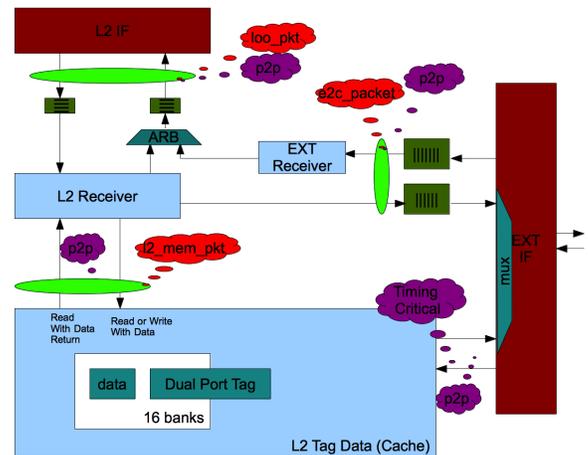


Diagram 2 – L2 Cache

the fulfilled packet. The EXT also can connect through external chip interfaces while still maintaining cache coherency. We used this feature to scale our prototype testing to multiple processors using multiple FPGAs. The EXT connection itself is a combination of packets via P2P and simple wired interconnect. All of the other interfaces in the block diagram were implemented using interfaces provided by Forte.

Most of the memories in the cache, except for the TAG, are single ported. In many cases a memory is used to store a complex data structure. Other memories in the design include those used in FIFO interfaces. We built FIFOs that mapped into Xilinx Distributed RAM as well as

Block RAM. FIFO's were required for EXT interfaces, buffering internal to the cache, and in the L2 cache.

For interfaces connecting off-chip it was necessary to use interfaces that include clock-domain crossing (CDC) circuitry. These were used extensively for interfaces, i.e. DDR3, and memory mapped IO, i.e. Ethernet and DVI. We found the P2P signaling protocol combined with CDC circuitry to be a very effective way to make reliable off-chip connections.

One of the key SystemC/C++ coding features we used is related to introducing clock-domain crossing circuitry at various points in the design. From a coding perspective the interface that incorporates CDC circuitry is identical to the P2P interface we use within a clock domain. This polymorphism allowed us to introduce CDC simply by instantiating a different class for the interface without needing more extensive coding changes. We took advantage of this flexibility to make decisions as to where clock domain crossings needed to be inserted late in the design process.

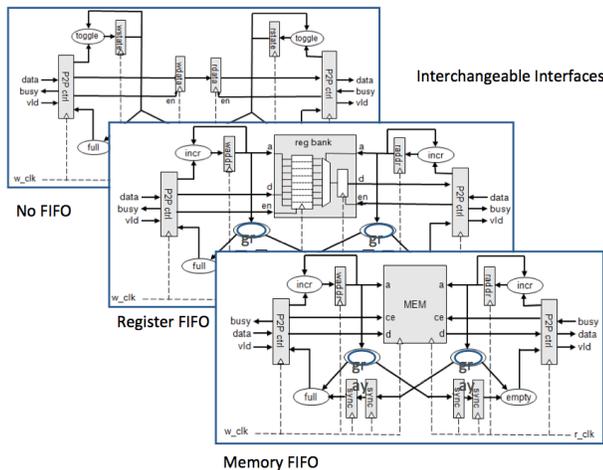


Diagram 3 – Interface Polymorphism

SystemC Modeling

The only way to design and verify a chip of this size and complexity is with a model that directly reflects the desired hardware architecture. There is simply no other way to create a simulation model that would allow us to adequately verify the connections between system modules and that our algorithms function correctly with this communication in place. Consider for a moment if you had a truly high level model of the processor where the system memory is flat and accessible in a single cycle to any address. Obviously this is not true or we wouldn't need a cache.

Although some of the coding required to create this model is fairly low level, it would be a mistake to think that this SystemC model is really just another RTL model. We found a tremendous amount of leverage from using C++ and SystemC. For instance we estimate that our SystemC model of approximately 50K lines of source code is a direct replacement for up to 2 million lines of RTL. Much of this leverage is directly provided by the basic C++ concepts such as templates, classes, polymorphism, and operator overloading. We used these object-oriented capabilities very effectively to manage the complexity of our design. The most obvious benefit is that we can build prototypes with 1 core then develop an ASIC with "n" cores from the same source code.

Another language feature that simplifies source code is encapsulated interfaces in a C++ class. For instance when reading a memory in SystemC your source code can be as simple as:

```
int address, a, memory[16];
a = memory[address];
```

All of the details like setting a chip enable, read / write signals, driving the address signals, waiting for the setup and hold times, etc. can be handled through an encapsulated interface called a metaport. Using operator overloading the memory read function can be invoked using array access notation as shown above. Of course, someone needs to write the protocol that goes into that interface class. But if it is constructed as a template then that same source code can be re-used anywhere we access a memory and our user source code can then be as simple as an array de-reference.

These concepts of metaports, templates and classes become even more powerful if we consider a CDC FIFO with an embedded dual port memory. After someone wrote that complex model that manages the FIFO and interfaces to the memory users can simply write source code that has a get() function to receive data, and a put() function to write data. We exploited the fact that the CDC Interface could be built with and without a FIFO and accessed identically from the standpoint of the external code. This allowed us to first get the interfaces working properly then to focus on whether a FIFO was needed to buffer the data across the clock domain crossing. When going from fast to slow we often used a FIFO to buffer the data; when going from slow to fast we often just use the interchangeable register-based CDC interface.

All of these IP blocks just described are provided as part of the Cynthesizer tool set. We also used these same techniques such as templates extensively in our own source code. The EXT for instance is a templated class.

The template is:

```
template < int t_workQueueDepth, int t_outboundQueueDepth,
bool t_isSlaveClient, enum ExtClientType
t_clientAccessType, int t_maxAddr >
class ext_adapter
: public sc_module
```

Through the use of template parameters, this single body of source code is used to support the interface for three clients, each with different workload and buffering characteristics.

As previously mentioned, our L2 TAG interface needed to respond to an EXT request in two cycles. The TAG interrogation that needed to be performed as part of this interaction used a read / modify / write which would have required a 4 port memory that was not practical with the chip resources we had available. Our solution was to use a memory class clocked at twice the base processor frequency. This class is provided as an option in the memory model generator that is part of Forte’s Cynthesizer toolset. By leveraging this class, we were able to make a much simpler dual port memory act to the surrounding logic as a 4 port memory.

There were lots of interfaces that needed to be developed between off-chip resources and the EXT. Of course, we also had to build our L1 and L2 cache with interfaces to the EXT. Xilinx provided some handy RTL IP blocks for the off-chip interfaces, but we still needed to develop the interfaces between those Xilinx IP blocks and our processing elements. Plus the entire cache model had to be built in SystemC.

Fortunately, Cynthesizer includes a set of pre-verified interface IP that we could easily leverage, all based on the P2P signaling protocol. This simple handshake interface can be used in stall-able pipelines and can be leveraged into very complex interfaces like the CDC FIFO described earlier.

The P2P protocol helped us meet all of our latency requirements by supporting the transfer of a data value on every clock cycle. It ensures reliable transfer by permitting the downstream module to assert its busy output and stall the upstream module. This prevents data from being lost in the case where the downstream data consumers cannot keep up with the data providers.

One of the details required when storing a complex data structure in a memory is some code that tells the High Level Synthesis (HLS) tool how to pack your structure into the raw bits of your memory. For Cynthesizer, this is done by writing `cynw_interpret()` functions. See the ‘Challenges Overcome’ section for further comments on this process.

Using these pre-verified interfaces and memory models definitely increased our productivity. In fact we estimate

that overall our design time was reduced from 2 man years down to 5 man months, approximately a 5 X productivity improvement.

Design Verification

We found a combination of significant advantages and disadvantages to doing our verification using C++ and SystemC. One of the biggest advantages of this methodology turned out to be how fast testbenches could be developed. We were able to create test cases with behavioral models providing stimulus through P2P interfaces and very quickly get functioning testbenches. In fact we estimate that testbench creation was as much as 10 X more productive than a comparable RTL based testbench would have been. Much of this productivity increase is simply the result of having access to the pre-verified P2P interfaces. This eliminated a lot of debugging of wired interfaces that we have experienced in RTL design projects.

Once we had a behavioral testbench, it was very easy to synthesize that model to build hardware that included a DUT plus a traffic generator. This enabled a very quick turn-around emulation capability that we used to great advantage. We also used Forte-provided synthesizable Random Number Generator for building traffic generators. Both the L2, Flash and DDR3 interfaces were extensively tested in hardware prototypes using this method.

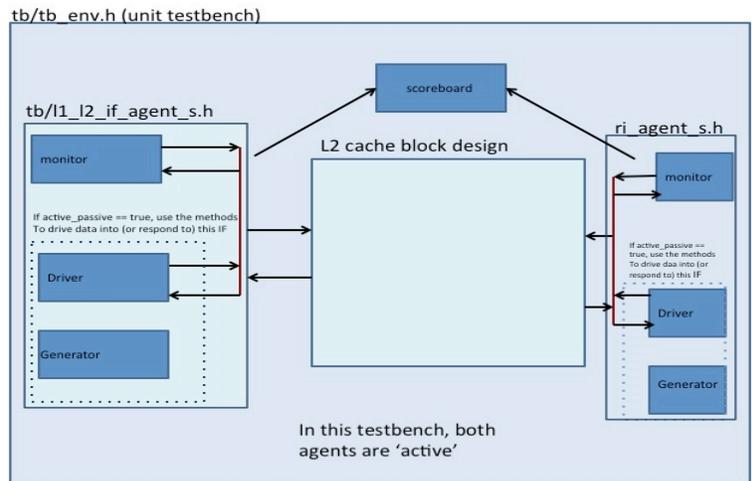


Diagram 4 - Unit Testbench

The main disadvantage we found was the difficulty of debugging such a large body of SystemC code. The SystemC library does not support the same level of run-time checking provided by HDL simulators which left us

in a number of situations where we found it difficult to debug. For example we had difficulty debugging C++ mistakes such as overwriting the stack of an SC_METHOD which would not be possible in a language such as Verilog. We believe that SystemC verification and debugging could be greatly improved by the availability of a good SystemC lint tool, a functional coverage measurement capability, and additional run-time checking in the library itself.

Many SystemC design projects make extensive use of transaction-level simulations. Unfortunately, because of our design style, we found that TLM simulations were impractical to use for any large simulation in our project. However we did use some TLM models for a few components to help us eliminate unwanted behavioral simulation delays. For instance the L2 TAG had multiple memories that all needed to be accessed in a single cycle from one thread. With the PIN level models for these memories each one would insert its own delay in the simulation. By substituting TLM models for those memories we were able to eliminate these 'artifact' delays.

Panève [3] intends to provide high quality IP for our customers and the testbench verification at the front end of the design->build process contributes the highest percentage of confidence in the functional behavior of the design. The matchup of the testbenches and the design provides ease of use, since both are written in SystemC, while the testbench environment takes advantage of the SystemC verification extensions added in 2002 under 2.0 -- scv_* library components.

Since the testbench structure starts with unit level testing and moves the same test modules, called 'agents' to the upper level testbenches, we get consistency in the testing, as well as re-usability of the testbench components. Each agent contains a monitor to verify the handshake correctness of the interface as well as feeding the data from the interface forward to an external scoreboard module to verify data correctness; a driver and generator, both constrained by the 'active' bool of the agent to drive, or 'respond' to transactions at the interface and a common data area in the agent for such items as lists of received transactions that would need to be forwarded or responded to, through the driver.

In order to verify the cache coherency, planning is necessary, not only for the actual points of functional behavior most interesting to the designers, but to formulate the needed coverage measurements that quantify the effectiveness of the verification testing.

Panève uses Jasper Design Automation's Gameplan since not only are the verification points easily organized and prioritized, the ability to include the coverage metrics makes Gameplan the total receptacle for our quality concerns.

Within Gameplan, we have items such as checking the coherency between the L1 and L2 cache entries as well checking the coherency of the individual bits kept in the tag of the L2 and L1. Also, items referring to the retirement mechanism of a cache entry from L1 to L2, and the requirements around the updating of L2 with the modifications on L1 entries, complicated by the characteristics of data delivery to and from the core.

At the unit test level, since the test fixture is the same as at the upper design level, the maximum legal and illegal behaviors are verified in the fastest possible manner, including, but not limited, to use of scv_* components such as the scv_sparse_array (from the SystemC verification extension library) which not only provides an easily defined (at the constructor) default value for read back of 'unwritten' locations, but the ability to range all over the address bits without significantly impacting the simulation footprint in memory, beyond the design and the testbench, during the time of the simulation (think hash implementation). As 'dyed in the wool' hardware engineers, having such tools at our disposal without having to design them, support them, and document them ourselves, saves not only simulation time, but gets testbenches built quickly. Since a large portion of the design is cache, this accelerates the design verification overall. The unit level testbenches also allow the designers the fastest turn-around for testing new design implementations as well as observability into behavior resulting from randomized data being flung about by the testbenches. Such data from the scv_smart_ptr component of the scv library provides ease of randomization, proper data persistence, and automatic garbage collection of these ephemeral constructs, also reducing the end-to-end

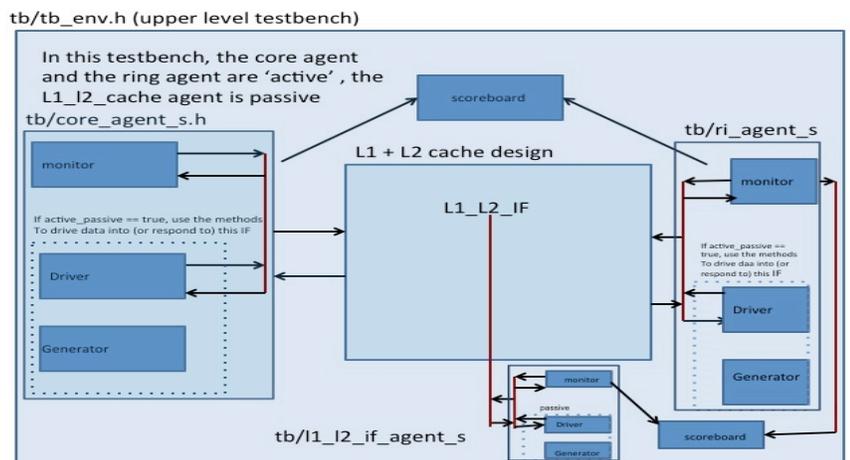


Diagram 5 - Upper Level Testbench

simulation memory consumption.

As the levels of cache as well as the core are combined, building outward with the 'agents' and switching the 'active' booleans to passive, provides observability into the internal IFs as the monitors are still available, while using the same driving agents (as at the unit levels) at the outer IFs. Although the rigidity of the combined design blocks reduces the randomness of the behaviors induced in the design, the ability to observe the interaction of the design blocks with each other, increments the level of the quality in the final IP.

Linking the high-level design with the RTL flow

We are a SystemC house so there was no RTL legacy to worry about. This gave us complete freedom to choose our implementation tools. Diagram 6 shows the design flow we used that incorporates the Synthesizer tools from Forte Design Systems.

For our flow we used the example ASIC library provided by Forte as our synthesis target technology. This required some experimentation to figure out what comparable clock frequency to use for the high level synthesis to ensure that our RTL synthesis could build us a gate level equivalent that would meet timing. We instructed Cynthesizer to build RTL with no direct gate instantiations for datapath components. We then exported that generated RTL and moved it into Xilinx XST to synthesize our FPGA "Proof of Concept" implementation.

Although Forte provided hooks into the Synplify-Pro FPGA synthesis tool we discovered that there were limitations to this approach which included library characterization at a given frequency as well as the general tool costs themselves. The Xilinx-only flow worked very well for our purposes.

Challenges that were overcome

There were some seemingly simple housekeeping tasks that turned out to be harder than expected. For instance the `cynw_interpret()` functions mentioned earlier that pack a user defined struct into a memory word. Even a minor change to a member of the struct would require us to go back and re-write a fair amount of code which of course is error prone. We found a simple coding solution by using enumerated `cynw_interpret` functions. For example here is an enum struct that is used to define field boundary limits:

```
enum ext_packet_field_boundaries
{
    EXT_ADDR_LSB_NUM = 0,
    EXT_ADDR_MSB_NUM = EXT_ADDR_LEN_B-1,
    REQUEST_ID_LSB_NUM,
    REQUEST_ID_MSB_NUM = EXT_ADDR_MSB_NUM + CLIENT_ID_LEN_B,
    EXT_CMD_LSB_NUM,
    EXT_CMD_MSB_NUM = REQUEST_ID_MSB_NUM + EXT_CMD_LEN_B,
    EXT_L2BLKSTATE_LSB_NUM,
    EXT_L2BLKSTATE_MSB_NUM = EXT_CMD_MSB_NUM + CACHE_TAG_STATE_B,
    RESP_BUSY_LSB_NUM,
    RESP_BUSY_MSB_NUM = RESP_BUSY_LSB_NUM, // single bit field
    RPH_PACKED_SIZE_B,
    DATA_LSB_NUM = RPH_PACKED_SIZE_B,
    DATA_MSB_NUM = RESP_BUSY_MSB_NUM + DATA_BLK_SIZE,
    PACKED_SIZE_B
};
```

Then the `cynw_interpret` function can use the defined enum types in range expressions to parse the data structure into memory words. In this case we are packing a structure of type `ext_packet` into a 72 bit word that can be stored in a memory:

```
inline void cynw_interpret (const ext_packet_header& from,
sc_biquint<72>& to)
{
    to.range(EXT_ADDR_MSB_NUM, EXT_ADDR_LSB_NUM) = from.ext_addr;
    to.range(REQUEST_ID_MSB_NUM, REQUEST_ID_LSB_NUM) =
        from.request_id;
    to.range(EXT_CMD_MSB_NUM, EXT_CMD_LSB_NUM) = from.ext_cmd;
    to.range(EXT_L2BLKSTATE_MSB_NUM, EXT_L2BLKSTATE_LSB_NUM) =
        from.ext_l2blkstate;
    to.range(PACKET_VALID_MSB_NUM, PACKET_VALID_LSB_NUM) =
        from.packet_valid;
    to.range(RESP_BUSY_MSB_NUM, RESP_BUSY_LSB_NUM) = from.resp_busy;
}
```

Equally, some of the packets used at the P2P interfaces were built from multiple structs to maximize commonality and encapsulation – an object oriented design goal. The above enum approach was then needed to be augmented with a 'MACRO' approach to make the enum process seamless and encapsulated at the packet definition.

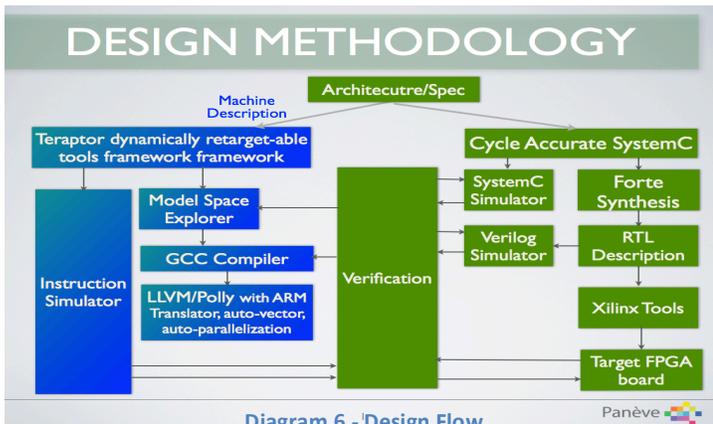


Diagram 6 - Design Flow

In the `ext_packet` definition file:

```
// These two macros are used in e2c_packet.h and c2e_packet.h
#define EXT_PACKET_PACK( )
to.range(EXT_ADDR_MSB_NUM,EXT_ADDR_LSB_NUM)
= from.packet.header.ext_addr;
to.range(REQUEST_ID_MSB_NUM,REQUEST_ID_LSB_NUM)
= from.packet.header.request_id;
to.range(EXT_CMD_MSB_NUM,EXT_CMD_LSB_NUM)
= from.packet.header.ext_cmd;
to.range(EXT_L2BLKSTATE_MSB_NUM, EXT_L2BLKSTATE_LSB_NUM)
= from.packet.header.ext_l2blkstate;
to.range(RESP_BUSY_MSB_NUM,RESP_BUSY_LSB_NUM)
= from.packet.header.resp_busy;
to.range(WAS_ACKED_MSB_NUM,WAS_ACKED_LSB_NUM)
= from.packet.header.was_acked;
to.range(UNUSED_PAD_MSB_NUM,UNUSED_PAD_LSB_NUM)
= from.packet.header.unused_pad;
to.range(DATA_MSB_NUM,DATA_LSB_NUM)
= from.packet.body.data;

#define EXT_PACKET_UNPACK( )
to.packet.header.ext_addr =
from.range(EXT_ADDR_MSB_NUM,EXT_ADDR_LSB_NUM);
to.packet.header.request_id =
from.range(REQUEST_ID_MSB_NUM,REQUEST_ID_LSB_NUM);
to.packet.header.ext_cmd =
from.range(EXT_CMD_MSB_NUM,EXT_CMD_LSB_NUM);
to.packet.header.ext_l2blkstate =
from.range(EXT_L2BLKSTATE_MSB_NUM,EXT_L2BLKSTATE_LSB_NUM);
to.packet.header.resp_busy = (bool)(sc_uint<1>)
from.range(RESP_BUSY_MSB_NUM,RESP_BUSY_LSB_NUM);
to.packet.header.was_acked = (bool)(sc_uint<1>)
from.range(WAS_ACKED_MSB_NUM,WAS_ACKED_LSB_NUM);
to.packet.header.unused_pad = (bool)(sc_uint<1>)
from.range(UNUSED_PAD_MSB_NUM,UNUSED_PAD_LSB_NUM);
to.packet.body.data =
from.range(DATA_MSB_NUM,DATA_LSB_NUM);
```

Then, in the file for the one of the packets that uses the `ext_packet`:

```
struct e2c_packet
{
    struct ext_packet      packet;

    // The following fields are only for client/EXT communication
    // and are not passed around in the ext packet

    // EXT to client flags
    struct ext_packet_status  status;
    .
    .
    inline void cynw_interpret (const e2c_packet& from,
sc_biguint<E2C_PACKED_SIZE_B>& to)
    {
        EXT_PACKET_PACK( );
        to.range(E2C_STATUS_IS_DATA_MSB_NUM,
E2C_STATUS_IS_DATA_LSB_NUM)
        = from.status.is_data;
        to.range(E2C_STATUS_IS_SOURCE_MSB_NUM,
E2C_STATUS_IS_SOURCE_LSB_NUM)
        = from.status.is_source;
        to.range(E2C_STATUS_IS_IO_DEST_MSB_NUM,
E2C_STATUS_IS_IO_DEST_LSB_NUM)
        = from.status.is_io_dest;
    }

    inline void cynw_interpret (const sc_biguint<E2C_PACKED_SIZE_B>&
from, e2c_packet& to)
    {
        EXT_PACKET_UNPACK( );
        to.status.is_data = (bool)(sc_uint<1>)
from.range(E2C_STATUS_IS_DATA_MSB_NUM,
E2C_STATUS_IS_DATA_LSB_NUM);
        to.status.is_source = (bool)(sc_uint<1>)
from.range(E2C_STATUS_IS_SOURCE_MSB_NUM,
E2C_STATUS_IS_SOURCE_LSB_NUM);
        to.status.is_io_dest = (bool)(sc_uint<1>)
from.range(E2C_STATUS_IS_IO_DEST_MSB_NUM,
E2C_STATUS_IS_IO_DEST_LSB_NUM);
    }
}
```

But our primary challenge at the beginning of this effort was having a very small team of engineers. This is certainly not uncommon as a startup but even larger companies have similar resource constraints. Without the

adoption of the SystemC design methodology and all the tools and IP provided with Cynthesizer we could not have completed this project in the time frame we did.

This was a very ambitious project. We leveraged multiple aspects of SystemC to allow our team to design and verify less code while producing a large complicated design. Would you prefer verifying 50K lines of SystemC or 2 Million lines of Verilog? That was always our guiding light in our use of SystemC. Furthermore the small team could focus on detail problems at a much higher level and then develop testbenches very quickly to prove the problem was addressed.

Challenges that remain unresolved

Constrained Random Simulation, which we generate with ease using SystemC, will be more useful when we have a way of scoring how effectively our stimulus exercised the design. Functional Coverage should not have been an afterthought in the language. There are vendors that provide Functional Coverage for SystemC and we are exploring them.

Code coverage is another area of concern, how do you know you are exercising all the code you have in the design, what about dead code detection. These are all places that Code Coverage could assist the design group.

Assertions are another powerful verification method that can work hand in hand with functional coverage. Assertions should be part of the language. I know that C++ has assertions but a more formalized recommendation within the language would go a long way for third parties to write tools to manage assertions.

`$display` and `printf` are NOT productive debug methods. GDB is not that useful on a large design but we used it extensively which cost us a lot of time. A good solid lint tool would have caught a lot of issues but debugging these complex models has to be easier for wide adoption. As I told someone this felt like using Verilog/VHDL in 1995!

Conclusion

This was a very ambitious design project accomplished with a small team within a fairly short period of time. This would not have been possible without changing the level of abstraction used to write the models that define the system. The level of abstraction of our code was made possible by basic C++ constructs along with vendor IP for interconnect and the SystemC design language. Once complete and verified this model can be efficiently transformed into RTL that can be targeted to an FPGA or ASIC with the Cynthesizer tool from Forte Design Systems.

References

- [1] Xilinx: <http://www.xilinx.com>
- [2] Forte Design Systems: <http://www.forteds.com>
- [3] Panève: <http://www.paneve.com>