A Methodology for HW-SW Codesign in ATM

Giovanni Mancini*, Dave Yurach

Bell-Northern Research
P.O. Box 3511, Station C
Ottawa, Ontario, Canada, K1Y-4H7

Spiros Boucouris

Functionality Inc.
1062 Barwell Ave.
Ottawa, Ontario, Canada, K2B-8H5

Abstract

This paper presents a methodology and strategy used for hardware-software codesign and coverification of a large ATM switch whose functionality is largely embodied in new ASICs. A strategy based on software emulation is presented which supports the concurrent development and verification of the system software with the hardware being designed prior to lab samples being available. The goal is reduced system integration times and design iterations due to system errors.

1. Introduction

The verification of the embedded control software (ECS) of large switching systems is a very complex task. Today's dynamic market environments and the resulting short development cycles require the concurrent engineering of the ASIC's, boards and software which make up the system. The goal is to have a *right first time* prototype. This requires the development of a methodology which supports the design and verification of the ECS prior to the availability of the hardware it must interface to.

This paper discusses a methodology and strategy used for the functional design and verification of the ECS of an Asynchronous Transfer Mode (ATM) [1] switch whose functionality is largely embodied within newly developed ASICs. The goal was to enable the development of system software in parallel with the actual hardware development.

The paper has two goals. The first is to give the reader an appreciation of the requirements and complexity of the task at hand. The second is to present a specific methodology used to address the problem in the design of a large ATM backbone switch.

The complexity of the task is put into perspective by providing a brief overview of an ATM switching system. The goals of the work along with details of the methodology used are presented. The paper concludes with comments on possible future developments required.

2. The System

The system itself consists of three major components (Figure 1), the hardware, the embedded control software (ECS) and the system control software (SCS).

The SCS software provides control functions for one or more switches and the required network management functions. The software may run on standard

---

* Giovanni Mancini is presently with Cadence Design Systems, San Jose, CA.

32nd ACM/IEEE Design Automation Conference ®

computer platforms such as UNIX based workstations. These processors are located in the network control centers of the operating companies or may be rack mounted in a central office facility, in close proximity to the actual hardware. The SCS can be of the order of hundred of thousands to millions of lines of code.

The Switching Element (SE) consists of the ECS and a real time switched data path implemented in hardware. The control architecture of the entire SE is software dominant. The ECS not only provides control and interrupt processing for the hardware, it also provides services to the SCS. All accesses to the hardware by the SCS are processed by the ECS.

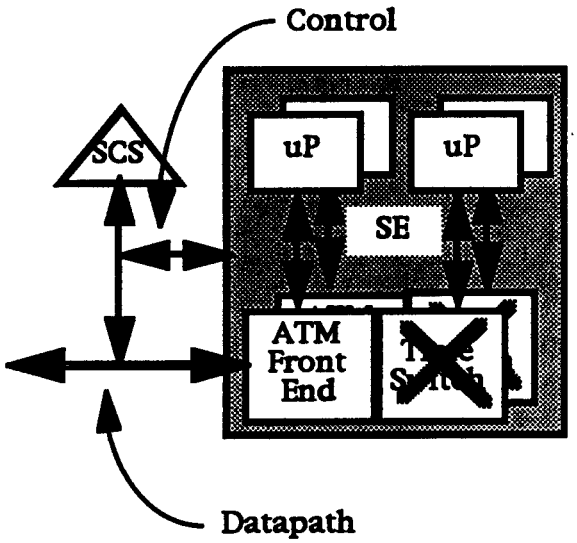

Figure 1. A functional representation of an ATM switch architecture.

The SE consists of proprietary hardware, commercial components and industry standard embedded microprocessors. The ECS is of the order of 60K lines of C/C++ code running under a commercially available real time operating system (RTOS).

The SE hardware is considerable in size. The exact component count depends on the specifics of the transport layers terminated. One SE can have well over 2 million design gates instantiated within its custom ASICs, in addition to commercial components.

The components of the SE hardware consist of a front end processing function which terminates the ATM cell streams from the network and carries out all ATM processing on the ingress and egress of cells. The cells are then relayed through a time switch function which switches the cells from the input port where they arrived to the output port from which they are to leave the switch. The ECS, which is distributed over numerous microprocessors, provides not only control for the hardware but also carries out a number of operations for the SCS.

3. Design Process

A typical system design process is represented by the diagram in Figure 2. The process starts with the initial architecture team developing a system architecture and specification. Using the system design specification, the design is partitioned into hardware and software design flows. The hardware flow is further divided into an ASIC design flow and a PCB design flow. Each of these three flows are tailored to the specific requirements of software design, ASIC design and PCB design. Once the hardware components are manufactured, the ASICs are incorporated onto the PCB and verified during hardware integration. Once base hardware functionality is verified, the software is added to the system during system integration.

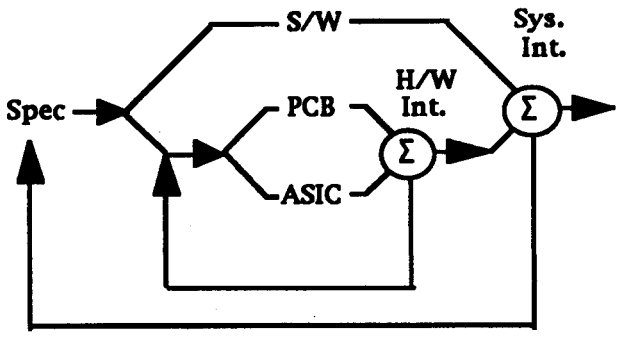

Figure 2. A typical system design flow.

Typically the EDA industry has focused on tool sets which support the reduction of the time required for the execution of these three flows. One of the biggest delays that can be encountered during product design is that which results from a design iteration. Major design iterations result from problems detected as late as hardware or system integration time. The individual flows produce fully functional components that when integrated into a system, either do not function or implement the wrong function. An iteration at this stage may result in missing a market window.

It is significant to note that in the development of such a new system as an ATM switch, it is not sufficient to only support the concurrent design and development of software and hardware. A detailed specification will allow this. It also requires a capability to verify the integrity and understanding of the software specification, prior to system integration.

We distinguish between codesign and coverification. Codesign refers to the process to create and/or verify the initial system specification which encompasses both hardware and software. Coverification refers to the process by which the software is verified against a simulated representation of the hardware prior to lab or system integration.

4. Goals

We had three main goals

- the elimination of design iterations due to ambiguous specifications
- the reduction of system integration time
- the elimination of major problems discovered during lab integration.

Simulation is the technique usually used to address these goals. In the design of a large switching system, the verification process is quite extensive. As designs get larger, especially at the system level, brute force techniques will result in a syndrome we refer to as *death by simulation*. The design team runs larger and larger simulations in order to verify the system. If they are uncertain as to the capability of the verification system it is extremely difficult for them to know when they have completed the task.

In order to overcome the death by simulation syndrome, one must tailor the strategy to take advantage of the architecture of the system. In the case of software dominant control systems such as ours, only a very small part of the hardware is actually relevant for the verification of the software.

5. Strategy

Of the various options available we chose a strategy based on transaction based modeling for codesign and software emulation of the relevant hardware for the coverification stage[3]. In order to maximize the benefit derived from this approach it was necessary to ensure that:

- Duplication of software was kept to a minimum.
- There was to be minimal impact on the software development process.
- The strategy was to provide a simple means of reconfiguration between model and actual hardware.
- No performance penalty on final ECS code would be introduced.

These restrictions were necessary both to facilitate the acceptance of the approach by software developers and to ensure that the maximum possible software testing could be done using the hardware models.

5.1. Modeling for HW-SW Codesign

Due to the complexity of the software, a transaction based model of both the SCS and ECS was developed using ObjecTime (OT) [3]. The goal is similar to the use of high level models in the design of hardware. The ObjecTime models were developed to verify a specification of the system software, and its interaction with the hardware, before it was designed. The software development environment

provided by OT can also be used to evolve the model into the real code.

In general, the performance of a full function OT model of such a large hardware system would render it useless. Given the layered nature of the SE, only a small part of the hardware functionality was relevant to verifying the software functionality. This significantly reduced set of the hardware function can easily be modeled in OT.

An OT model of the system can be used not only to verify the hardware-software operational model, but also for training. During the course of a project additional software developers will join the team. The model provides a fast and accurate mechanism for new members to learn about the design. It can also be used to resolve any uncertainties in the specification.

5.2. HW-SW Coverification

In addition to OT models to verify the front end of design, a software emulator for the hardware was developed to enable the concurrent development and verification of the ECS.

The emulator is similar to the OT model, except that it is implemented in C/C++ and has significantly better performance than a comparable ObjecTime model. The emulator can run on a host UNIX workstation or a microprocessor development board and mimics only the hardware functionality that is seen by software.

To obtain a better understanding of the applicability of a software emulator, we need to take a more detailed look at the system architecture (Figure 3). The ECS exercises control of the hardware through registers within the ASICs. The ASICs are attached to the microprocessor bus through a standard Software Interface (SWIF) block contained within all the ASICs. SWIF provides a uniform means of communications between the ECS and all of the ASICs. Each ASIC occupies a unique range within the microprocessor's memory map. Within that range, each of the ASIC's register occupies a unique memory address. The ASICs also generate vectored interrupts with each device assigned a unique vector value. Hence, all the ASICs under the control of a microprocessor appear to the ECS as an address range within memory and a source of vectored interrupts.

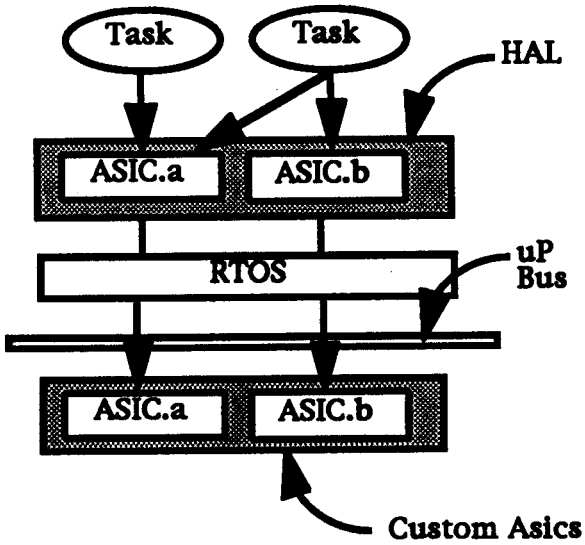

Figure 3. SE layered architecture.

Tasks within the ECS communicate with the hardware through a software layer referred to as the Hardware Adaptation Layer (HAL). The HAL provides a logical representation of the physical devices; a higher level view of the detailed device drivers; and fields interrupts generated by the hardware.

In an emulated system (Figure 4), the lower sections of the HAL are stubbed out and replaced by a process which is emulating the software relevant aspects of the hardware. The challenge here is to define a *stubbing* process which will not replace the ECS layer of code which we are attempting to verify; namely the HAL layer.

These restrictions led us to define the interface at the lowest possible level, the hardware register level. This minimizes the portion of application software replaced by the emulator, resulting in maximum real ECS code coverage.

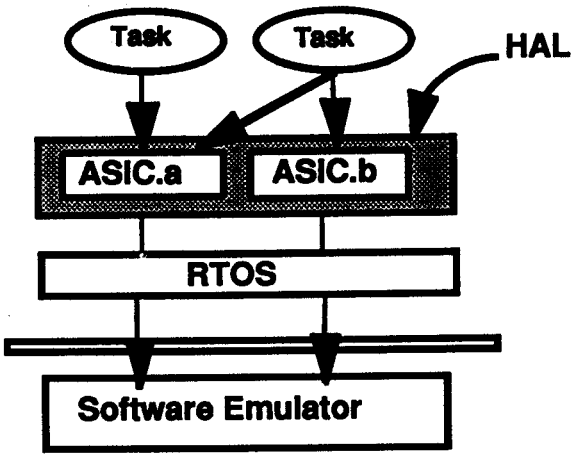

Figure 4. SE verification architecture with software emulator.

This interface, along with the SWIF, is defined quite early in the design process because of their interdependence. Consequently, emulator and model development can begin before the software or hardware design is complete and can proceed independent of the implementation process.

ASIC Representation: Memory corruption is a common problem in languages that do not support structures. A similar problem arises in real-time systems that must interact with a large number of hardware ASICs. This problem is further aggravated if there are several identical ASICs that must be controlled by the same software. In order to contain this problem, a structured representation was used even at the lowest levels. Thus each ASIC was represented by the equivalent of a C structure:

Register Name	Size (Bytes)	Offset
RESETS	2	0
RESETC	2	2
NTID	4	4
DATA	4	8

The register types (Reg16, Reg32 etc) were implemented as C++ classes. All Reg classes provided methods **read** for reading the value of a register and (if the register could be written) **write** for writing a value into the register.

Each ASIC was represented by a C++ class that served as its device driver. Instances of register classes provided access to the ASIC registers. Additional member functions, such as initialization and mode configuration, were defined to perform more complex operations on the ASIC. These typically required one or more register accesses.

The requirement for independent evolution of the software interface and the hardware model(s) led to the development of two parallel representations for each ASIC. One was the SW view of the ASIC

```
class asicX {
    public:
        Reg16   RESETS;
        Reg16   RESETC;
        Reg32   NTID;
        Reg32   DATA;
        // Driver methods ...
}
```

which encapsulated all the software drivers. The other was the HW view

```
class HWasicX {
    public:
        HWReg16   RESETS;
        HWReg16   RESETC;
        HWReg32   NTID;
        HWReg32   DATA;
};
```

which emulated the behavior of the ASIC with various degrees of accuracy.

SW View: The exact implementation of the "driver" portion of each ASIC is not of interest since it is entirely contained within the ECS. What is important is that each ASIC definition includes all the visible registers of the actual hardware. In effect each Register object is a window to the actual hardware register it represents. During initialization each Register object acquires the means of accessing the corresponding actual hardware register. The access methods provided by each register are limited to reading and writing a value from/to the corresponding

hardware register, possibly with some verification that the access was successful.

The hardware models are interfaced to the application software by redefining the Register classes so that they access the hardware models rather than actual hardware. This makes it possible to provide maximum coverage of the application software while the transition from model to actual hardware can be accomplished with a simple recompilation. It is worth noting that no recompilation is necessary, only relinking, if different hardware models are used.

ASIC Models: ASIC models were required to provide (as a bare minimum) all the registers of the actual ASIC they represented. Depending on the level of detail required for each ASIC we defined a graduated hierarchy of models. These models are all interchangeable and it is possible to have different models for different instances of the same ASIC. The different levels of models defined are:

Register Model: This model contained only the registers of the actual ASIC. In its most basic form it simply retained the last value written. None of the actual operation of the ASIC was modeled. Instead, hooks were provided so that ECS developers could provide their own actions to be executed whenever a register was accessed. This level of modeling makes it possible to test hardware accesses and selected (isolated) software responses.

Control Model: This model emulates most of the internal control path operation of the ASIC but none of the data path. The data path can still be emulated by supplying actions to register accesses. The model automatically handles inter-register dependencies, interrupt generation and hardware reset. This level tests more complex or interdependent software responses to changes of the hardware state. With register models, most of the lower levels of the application software can be tested along with isolated data-dependent responses.

Data Path Model: This model emulates the entire behavior of the ASIC. As such it includes the entire register model and also the data path operation of the ASIC. Ideally such a model would be provided by the hardware developers. Using data path models, more of the higher, data dependent, levels of the application software can be tested. Once again hooks are provided so the user can emulate selected interactions between ASICs.

HDL Model: This would be a Verilog or VHDL model, at the RTL level or above, of the actual ASIC. It would run on a hardware simulator and interfaced to the application software through model stubs. Since this level provides the full functionality of the hardware it is also possible to test sequence dependent software responses, without exact timing.

Emulator Control: Given the high level of abstraction within the emulator, a means is required to drive the emulator. For this we chose a text interface based on Tcl [4]. The purpose of this interface is to

- provide a means for configuring the hardware (i.e. emulator) into a particular state to enable the testing of specific threads within the ECS
- emulate the asynchronous generation of interrupts by the hardware
- Bind command procedures to registers, which would trigger on accesses to the register. This is a significant aid in the software verification.

This provides the ECS developer with an easy and flexible means to configure the emulator for testing even the most pathological scenarios. It also defines standard methods and mechanism for configuring the hardware.

6. Results

The implementation of the ASIC models and the application software were done in C++. A set of Register classes were defined that served as a replacement of the default Register classes that will access the actual hardware. These Registers were then combined in the ECS into ASICs. Each register was initialized with the base address of the ASIC in which it resides and its offset from the base address.

The model registers passed these parameters to a resolution routine provided by the modeling environment which determines which hardware register is to be bound and provides a pointer to the proper model register. ASIC models also contain models of all the registers of the actual ASIC. ASIC models are created dynamically on demand. This approach makes it possible to employ the actual application initialization code with the models and also to ensure that the model and application software have the same concept of which ASICs are present at what address.

The Control Model is a proper super set of the Register Model. Similarly, the Data Path Model is a proper super set of the Control Model. This enabled us to introduce models in a progressive manner with simple models becoming available very early and more elaborate models developed as needed. Using C++ inheritance and polymorphism we were able to maintain a consistent set of models for each ASIC and to easily upgrade the entire set in the case of hardware changes. The Tcl interface made it possible to do a number of application software tests using the more simplistic models. Eventually, the Tcl interface will also be used for testing the application software.

The Tcl interface provided benefits beyond its intended goals. It provided a simply means by which the test cases used to verify the ECS could be made regressionable. Since it provided standard means for configuring the hardware into a particular state, Tcl and scripts generated were also used by the hardware systems designs for verifying the hardware during hardware integration and as a platform for running manufacturing tests. It was also used to provide a means to provide an interface into the SCS and still provide maximum code coverage on the part of both the SCS and ECS.

An HDL model interface has not been developed yet. The additional gain would be minimal given the architecture of this system. Such a capability would have significant benefit in systems in which the division between control and data path is not as clearly aligned along the division of software and hardware.

7. Conclusions

The paper has described the goals and a methodology for hardware-software codesign and coverification of a large ATM switch whose functionality is largely embodied within new ASICs. Brute force techniques cannot be used. Rather the strategies used must take advantage of the system architecture to abstract out hardware details.

The methodology not only provides a solution to the problem but does it in a manner which naturally interfaces into the existing development environments of both the software and hardware developers.

The methodology supports early development and testing of real-time application software in parallel with hardware development. A hierarchy of models of increasing complexity was used to support the evolving software needs. The encapsulation of the ASIC models makes it possible to provide ever more complete models by redefining which ASIC models get bound to the ASIC's access method. The approach developed for the representation of the hardware in the application software provided maximum compatibility with the models while also eliminating a source of errors in hardware accesses.

The methodology is generic and can easily be applied to systems of similar architectures. The tool box nature of the

strategy used also enables significant reuse of the emulation software on new systems.

The approach is limited to system architectures in which have software dominant control with hardware data paths. In systems where the software actually implements part of the data path, an interface to actual HDL models, at the RTL or higher level, is required. This can readily be accomplished with the architecture of the emulation strategy provided.

8. References

[1] High Speed ATM Switching, IEEE Communications Magazine, Feb. 1993.

[2] B. Selic, G. Gullekson, J. McGee, I. Engelberg, " ROOM: An Object-Oriented Methodology for Developing Real Time Systems", Proc. of 5th Intl. Workshop on CASE, 1992.

[3] J. Ousterhout,"Tcl: An Embeddable Command Language", Proc. of 7 USENIX Winter Conference, January 1990, pp.133-146.

[4] G. Mancini,"HW-SW Co-Verification in ATM", Proc. of 7th HLSS, 1994, pp.1-7.