University of Delaware
Department of Electrical and Computer Engineering
Computer Architecture and Parallel Systems Laboratory

Design and Integration of New Architecture Features into a Many-Core Chip Architecture - A Report on a Novel Architecture/Software Co-Verification Platform

Juergen Ributzka, Yuhei Hayashi and Guang R. Gao

CAPSL Technical Memo 096
April 12th, 2010

University of Delaware • 140 Evans Hall • Newark, Delaware 19716 • USA
http://www.capsl.udel.edu • ftp://ftp.capsl.udel.edu • capsladm@capsl.udel.edu

Abstract

Historically, design and integration of a new architectural feature requires time consuming full system verification/validation. This challenge is particularly serious for the research and development of multi-/many-core chips due to the scale of the chip and time-to-market constraints. We conducted a logic design and integration of a new architecture feature, called Synchronization State Buffer (SSB) into the IBM Cyclops-64 (C64) many-core architecture logic. We used DEEP, a FPGA-based emulation platform that was developed for the purpose of hardware/software co-verification of the C64 chip. Our experiences have demonstrated the following key features are critical to achieve our objectives:

Fast compilation of the logic design is very important during the early stages of the logic design to allow for quick turn-around times. To this end, the full C64 chip logic can be compiled under two minutes for software simulation.

Full-System hardware emulation with debugging support is crucial for whole chip logic verification - some deep, concurrency-related bugs may be exposed only after many cycles of full chip parallel execution. DEEP's emulation mode allows such bugs to be found very fast - in minutes, which would have taken otherwise years with software simulation.

High-Speed hardware emulation (over 20KHz) of the whole system allows us to run entire parallel applications. This allows the experimentation of novel program and code generation paradigms within reasonable time.

Our experience show that future many-core research and development could greatly benefit from advanced emulation paradigms as we have explored in this study.

1 Introduction

Historically, design and integration of a new architectural feature requires time consuming full system verification/validation. This challenge is particularly serious for multi-/many-core chips research and development, due to the scale of the chip and time-to-market constraints.

Currently, full-system verification still requires an armada of computers or expensive specialized hardware to achieve reasonable emulation speed. A cluster of computers can be made easily available to a larger group of developers, but the overall emulation speed is still limited. On the other side, specialized hardware is much faster, but it is a scarce resource. Faster and cheaper hardware emulation and verification systems are needed to mitigate this problem.

The need for better and faster verification frameworks is growing even stronger with the introduction of new execution models. These models provide feedback to hardware architects about possible advantageous features. This results in a symbiotic relationship, which requires hardware / software co-development, verification platforms and methodologies. As new and more powerful many-core architectures rise, they bring current software stacks to the verge of expandability. New execution models, operation systems and system software will be required. Current modifications on software stacks made it work for the few cores which we have, but at the expense of performance and portability. However, such changes will not be enough for the coming future many-core architectures which may present anywhere between hundreds to thousands of cores.

Among the new many-core designs, we have the IBM Cyclops-64 (C64) many-core architecture. The architecture consists of 160 homogeneous processing elements called Thread Units (TUs). Two TUs share one Floating-Point Unit (FPU). Two TUs and one FPU build a logical processor. The 80 processors are connected via a high speed crossbar interconnect. Each TU controls a portion of the on-chip memory, which can be configured during boot-up into two sections. One section is used as Scratch Pad memory, which can be accessed very fast by the TU that owns it. The other TUs can still access all other TUs' Scratch Pad, but they have to go through the slower crossbar. The other section contributes to the Global Interleaved SRAM, which is accessed by all TUs via the crossbar. The access to the Global Interleaved SRAM is guaranteed to be sequentially consistent. The same cannot be guaranteed for the Scratch Pad memory, due to the fast access of the TU that owns it. There are no data caches on this architecture - only Instruction-Caches (ICs). The programmer has full control over the memory hierarchy, which includes the Scratch Pad memory, the Global Interleaved SRAM and the off-chip DDR2 memory. A more detailed description can be found in Section 2.1.

To test the C64 chip hardware features and its software stack, the DEEP emulation system was created. The DEEP system is the original emulation system for the C64 architecture, which was specifically designed and built for many-core architecture emulation and verification. Its unique iterative approach allows the emulation of huge many-core systems with a limited set of FPGAs [1]. The emulation system supports two different emulation modes. Mode 1 is slower, but new designs can be tested within minutes. Mode 2 gives maximum performance in terms of speed, but the whole design needs to be synthesized first and this can take several hours. The first mode is used during early development, where design changes occur more often. After the design has matured over time and the amount of changes have been greatly reduced, the second mode is used to run more extensive tests. A more detailed description of both modes is presented in Section 2.2.

One of the most important new features that these new many-core chips require is the ability to provide efficient synchronization constructs. One way to provide this is to enhance the architecture with hardware support for these constructs. One of the most recent hardware enhancements is the Synchronization State Buffer (SSB). The SSB is a small buffer which allows the storage of a limited amount of synchronization state bits, which can be associated with any memory location. This idea is based on the observation that only a limited amount of memory locations are undergoing synchronization at the same time. This idea was originally proposed and implemented in a functional-accurate simulator by Zhu et al. [2]. A more detailed description of SSB is given in Section 2.3 and 3.

This paper reports a design study based on the C64 many-core architecture using the DEEP emulation and simulation framework. We conducted a logic design to enhance C64 with the SSB feature by integrating it into the C64 architecture logic. Finally we carried out a performance study. This study involves the hardware changes in the existing ISA and memory controller, as well as software changes related with potential new programming paradigm and compilation strategies.

Our experiences have demonstrated the following key features that are critical to achieve

our objectives. In particular, an experimental platform/infrastructure is needed such that:

- In simulation mode - it can compile the logic very fast. The full C64 chip logic can be compiled in less than two minutes on a normal workstation. This is important during early development, where a lot of changes occur.
- In emulation mode - it is able to perform full chip logic emulation and provides debugging support for verification at a reasonable speed for our purpose. This is important, since exposing some deep, concurrency-related bugs would need weeks, months or even years of execution time under simulation mode, while it only take a few minutes under full emulation mode.
- In emulation mode - it can provide high-speed emulation to run parallel applications on the whole chip, thus allowing us to experiment with novel program paradigms inspired by SSB and corresponding code generation schemes within reasonable time. It executes our benchmark programs at well over 20KHz speed (about 1/40,000 of the actual (powerful) chip speed).

The remainder of the paper is structured as following: Section 2 gives additional background information about the C64 many-core architecture, the emulation system DEEP, and the fine-grain synchronization extension SSB. Section 3 explains our design and implementation of SSB in the C64 architecture. Section 4 shows our observations and findings during this process. Section 5 gives a overview of related work and Section 6 concludes the paper.

2 Background

2.1 The IBM Cyclops-64 Architecture

The IBM Cyclops-64 (C64) architecture is logically partitioned into 80 homogeneous processors which are connected to a 96-port crossbar. A processor contains two Thread Units (TUs), which share one Floating-Point Unit (FPU). Therefore, it is possible to have 160 independent and concurrent threads running at the same time. Every TU is attached to one SRAM bank. Each TU can access all SRAM banks via the crossbar. The SRAM banks can be configured during chip boot-up into two distinct sections. One section of the SRAM bank contributes to the Global Interleaved Shared Memory; the other section can be used as Scratch Pad memory. A TU has a direct, low-latency access to its own Scratch Pad. The Scratch Pad of other TUs can still be accessed through the crossbar. Sequential Consistency is guaranteed for the Global Interleaved Shared Memory, but not for the Scratch Pad. TUs are in-order single-issue and out-of-order completion cores and have a quad-ported register file (two read and two write ports) with 64 $\times$ 64bit General Purpose Registers (GPRs). All TUs share a common signal bus, which provides fast barrier support in hardware. Ten TUs (five processors) share one Instruction-Cache (IC) and four ICs share one crossbar port. There is no Data Cache (DC).

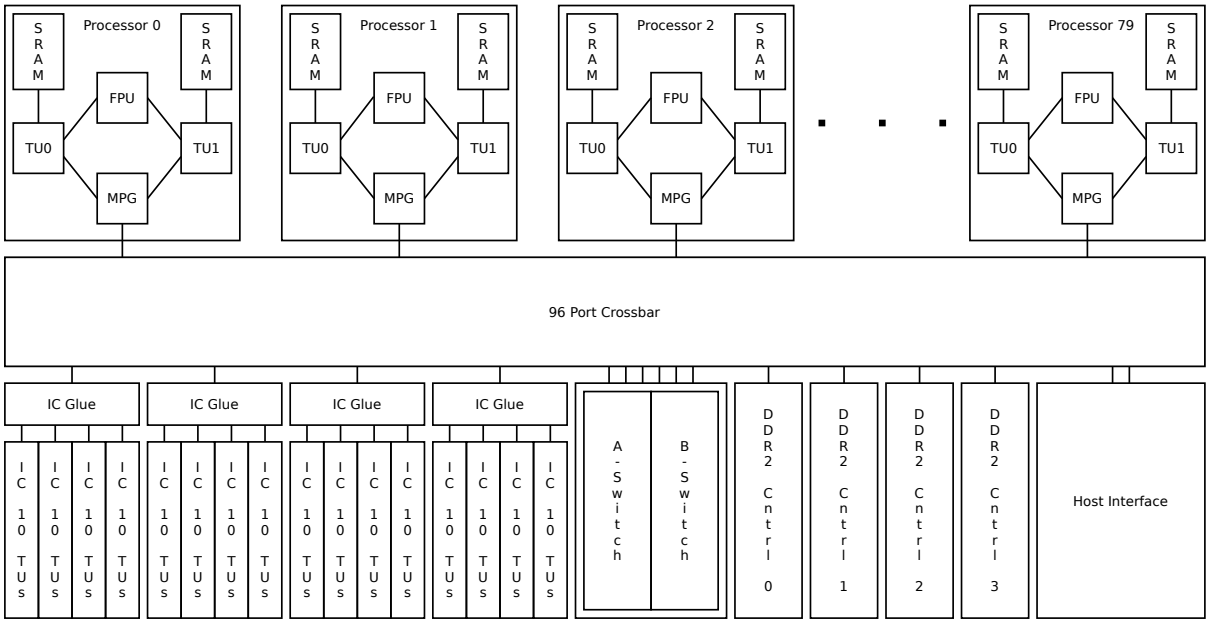


Figure 1: IBM Cyclops-64 (C64) Many-Core Architecture: The architecture consists of 80 processors (Processor 0 -79). Each processor has two Thread Units (TUs) called TU0 and TU1. Both share one Floating-Point Unit (FPU) and one crossbar port (MPG). Each TU is connected to a SRAM bank, which can be accessed by all other TUs via the crossbar. Ten TUs share one Instruction-Cache (IC). The system has four on-chip DDR2 memory controllers to access off-chip memory. The A-Switch is used to connect to the six surrounding neighbors in a 3D-mesh network.

Off-chip DDR2 memory is connected through four on-chip DDR2 memory controllers. Each memory controller is connected to its own crossbar port. Each chip can be connected to six neighboring chips in a 3-D mesh network. The network switch is also integrated into the chip and has six connections to the crossbar. The host interface is connected to two crossbar ports. In summary, the chip's crossbar interconnect possesses a total of 96 ports: eighty for the processors, four ports for the I-cache, four ports for on-chip DDR2 memory controllers, six ports for inter-chip communication, and two ports for the host interface. A logical overview of the chip is shown in Figure 1.

The architecture uses a explicit memory hierarchy similar to the one found in the NVIDIA CUDA or the Cell/B.E. architecture. Moreover, there is no paging or virtual memory support between all the memory hierarchy segments. More information about the C64 architecture and its system software can be found here [3, 4, 5].

2.2 Emulation System

The DEEP emulation system was developed in order to both validate the C64 chip's hardware features and test its software stack. The challenge and cost in testing new hardware designs lies

in the difficulty of verifying whether a circuit will work under real-world conditions. Software-based simulators can get close to the behavior of the real circuit, but take much longer to execute than the actual hardware. Therefore, it is unrealistic to run enough benchmarks for a whole chip on existing logic simulators to verify hardware design and / or test its software stack. Hidden bugs (similar to heissenbugs, mandelbugs, and bohrbugs) must be discovered by running real-world applications using full system emulation. While high emulation speed is required, it is very important to quickly respond to logic design changes, especially, at the early stage of logic design. Although the amount of bugs at the early states of development are usually high, they are easily found using simple synthetic test cases or small benchmark kernels. That means turn-around time regarding logic change is more important than emulation speed - at least during early stages of development. The major objectives of the DEEP emulation system are to support all design and test stages, to realize good turn-around time for the early stages and high emulation speed for the later stages, to do the whole chip emulation as well as to provide an efficient debugging environment.

In order to achieve its main objective, the DEEP system supports two different modes: simulation mode and emulation mode. The simulation mode is a logic processor based logic simulation methodology. In this mode, a logic design is translated into logic programs, and then it is simulated on a large number of logic processors. Generally, a logic design consists of a netlist of gates and memory cells, and it can always be mapped to a series of primitive logic operations such as AND, OR, MUX, etc.. Due to simple translation, the DEEP system can quickly generate logic programs from an original logic design. For instance, the C64 combinatorial logic design, 43 million gates (estimation), can be translated into logic programs within two minutes. The simulation mode is available on both, a FPGA-based hardware (see Figure 2) and a general workstation. The DEEP hardware has 32 Altera Stratix II FPGAs(EP2S90F1020C4); 20 for processing units, 10 for switches, and 2 for host communication. In each processing unit, 20 logic processors are implemented and one logic instruction queue are shared by them.

Since all logic is executed as logic programs, it is easy to check any signals in a target logic in the simulation mode. Moreover, not only simple signal tracing is possible, but also program tracing is supported when a processor is simulated. Using program tracing, correctness of a target benchmarks can be easily confirmed. If an error is discovered, we can switch to signal tracing or use both tracing strategies although simulation speed slows down considerably. The key feature of this simulation mode is fast translation into logic programs and good debugging support.

On the other hand, the emulation mode design is based on an iterative emulation methodology [1]. Since the whole many-core architecture design cannot be fitted into a single FPGA of the DEEP hardware, or any current available FPGA on the market, the architectural design is separated into submodules, which can fit into a FPGA. Even though each submodule fits into one FPGA, a lot of FPGAs would be required to implement the entire chip in the emulation system. Furthermore, many hardware resources would be necessary for communication between submodules in different FPGAs. Instead of mapping each submodule to a different FPGA, the emulation system adopts an iterative emulation approach [1]. Combinatorial logic

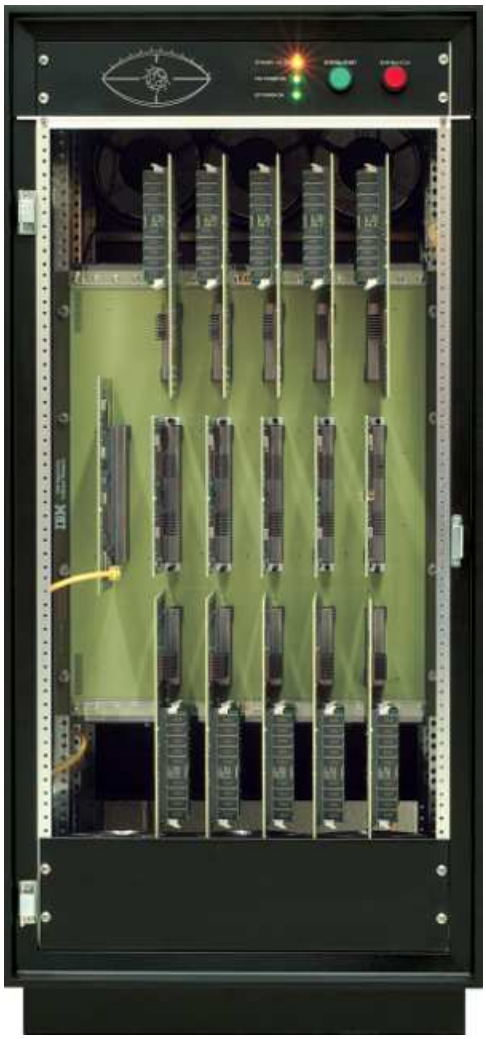


Figure 2: DEEP: The emulation platform consists of 32 Altera Stratix II FPGAs; 20 for processing units, 10 for switches, and 2 for host communication.

equivalent submodules are implemented on only one (or a few) FPGAs, and then iteratively utilized to emulate all instances of the submodule. This emulation method drastically reduces the required number of FPGAs. Each submodule's Flip-Flops (FFs) and internal RAM blocks are isolated from the original design. The content of the FFs and RAMs are independent of each submodule's instance, so they must be stored separately. The emulation system utilizes internal memories for FFs and external memories for RAM blocks, and only the combinatorial logic is implemented in the FPGA. The flow described above is done by the DEEP software automatically. By adopting the iterative emulation methodology, huge logic designs, which cannot fit into existing single FPGA, can be emulated in one or several FPGAs. Because a target logic design needs to be synthesized and mapped into a FPGA, it takes much more preparation time than the simulation mode until the logic design is ready to be emulated. However, after the logic design is mapped into the FPGA, it works as the real logic on an FPGA even though

it requires to emulate the logic iteratively. In case of the C64 design (with SSB) the average emulation speed of the whole chip is around 20k cycles/sec. In addition, debugging support is also necessary in the emulation mode because such support is very helpful to locate a bug after running benchmarks for a long time. The debugging support feature of the simulation mode is utilized in the emulation mode. In this mode, all combinatorial logic is mapped into a FPGA, so it cannot be observed directly. Fortunately, the content of FFs and memory blocks is reachable because they are stored in memories of the DEEP hardware. Being able to read this data from the emulation hardware, all signals can be observed by simulating combinatorial logic on a host workstation of the DEEP system.

2.3 Synchronization State Buffer

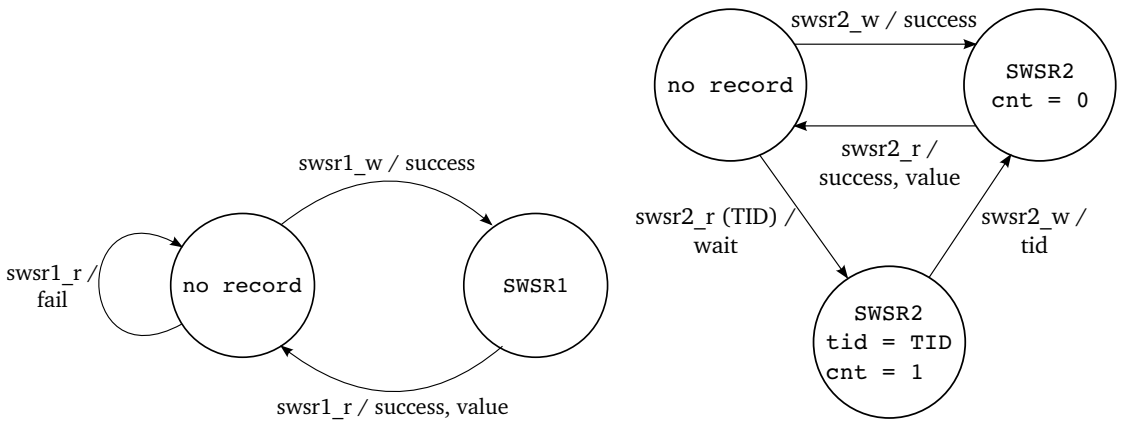
The Synchronization State Buffer (SSB) is a hardware extension to the C64 architecture proposed by Zhu et al. [2]. The idea is to add a small buffer to each memory controller. Instead of tagging the whole memory with additional bits (i.e. full/empty bit), the tags are stored in the buffer. In this way, the advantages of tagged memory for fine grain synchronization can be exploited with a smaller increase in memory size. However it also limits the number of memory locations that can be synchronized at the same time. On the other hand, according to [2] the number of synchronizations at any given period of time is much smaller than the total memory and a buffer should be sufficient. If the buffer is full though it can trap and switch to a software solution. SSB allows to tag every memory location with more than just one bit. Therefore, several synchronization constructs can be implemented. The original SSB proposed the following constructs: Read and Write Locks, and Single-Writer-Single-Reader and Single-Writer-Multiple-Reader synchronization. Nevertheless, any other kind of memory operation that needs additional state bits, like forwarding memory cells, tracing bits, hardware support for watchpoints, etc, could be implemented using SSB. The extension was implemented in a functional-accurate simulator called FAST. For more information please refer to the original publication [2] and Section 3.

3 Design and Integration of the Synchronization State Buffer into the Cyclops-64 Architecture

In this section, we will explain the design principles for the Synchronization State Buffer (SSB) and its integration. Our major interests were the Single-Writer-Single-Reader (SWSR) synchronization operations. The original SSB design has two different SWSR modes. Mode 1 employed a busy-wait approach for the reader until the data is ready. The second mode utilized the sleep-wakeup features of the architecture to reduce crossbar traffic. The operational semantics for the different Single-Writer-Single-Reader modes are as follow:

Mode 1: Busy-Wait

If the writer is first, then an entry is created in the SSB and the status “SUCCESS” is



(a) Mode 1: busy-wait

(b) Mode 2: sleep-wakeup

Figure 3: Single-Writer-Single-Reader (SWSR) State Diagrams

returned to the writer. When the load arrives, it is allowed to proceed and the entry is removed from the SSB. The value and the status "SUCCESS" is returned to the reader. If the reader is first, then no entry is created and the status "FAIL" is returned to the reader. The reader has to retry until the status "SUCCESS" is returned. The corresponding state diagram is shown in Figure 3(a).

Mode 2: Sleep-Wakeup

If the writer is first, then an entry is created in the SSB and the status "SUCCESS" is returned to the writer. When the load arrives, it is allowed to proceed and the entry is removed from the SSB. The value and the status "SUCCESS" is returned to the reader. If the reader is first, then an entry is created and the status "WAIT" is returned to the reader. The reader goes to sleep and waits until it is woken up by the writer. When the writer arrives second the Thread ID (TID) of the waiting reader is returned. The writer sends the wakeup signal to the waiting reader. The reader has now to retry the load again. This time it will succeed and the entry is removed from the SSB. The corresponding state diagram is shown in Figure 3(b).

We extended all modes to support any size (byte, half word, word and double word) and signedness (signed and unsigned) of memory operation. We created a new instruction format to accommodate the requirements of the SSB instructions. The new instruction format accommodates the major opcode (OP), the return register (RT), the address register (RA), the value register (RB), the SSB opcode (UU), the size (Sz) and the signedness (S). The field X is not used. The size of each field is shown in Table 1. The UU field is used the specify which SSB operation is used. A list of the implemented SSB operations is shown in Table 2.

The following list shows the new SSB assembly instructions. Every SSB assembly instruction has different versions for the different sizes and signedness. For brevity they are reduced to a

Table 1: SSB OP Format

Field	**OP**	**RT**	**RA**	**RB**	**UU**	**Sz**	**S**	**X**
Size(bit)	4	6	6	6	6	2	1	1

Table 2: SSB OP Opcode

UU	**SSB OP**	**Description**
0	RLock	Read Lock
1	WLock	Write Lock
2	UnLock	Unlock
3	SWSR1_R	Single-Writer-Single-Reader Mode 1 Read
4	SWSR1_W	Single-Writer-Single-Reader Mode 1 Write
5	SWSR2_R	Single-Writer-Single-Reader Mode 2 Read
6	SWSR2_W	Single-Writer-Single-Reader Mode 2 Write
7 - 16	reserved	n/a
17 - 63	not used	n/a

general instruction group.

RLOCK RT,RA

The instruction tries to obtain a read lock on the memory location defined in register RA. This instruction implicitly writes to two registers. The return code is written to register RT and the return value is written to register RT+1. A value of 0 in RT means "SUCCESS" and a value of -1 means "FAIL". If the read lock was successfully acquired, then the value of the memory location is available in RT+1.

WLOCK RT,RA

The instruction tries to obtain a write lock on the memory location defined in register RA. This instruction implicitly writes to two registers. The return code is written to register RT and the return value is written to register RT+1. A value of 0 in RT means "SUCCESS" and a value of -1 means "FAIL". If the read lock was successfully acquired, then the value of the memory location is available in RT+1.

UNLOCK RT,RA

The instruction tries to unlock a previously acquired read or write lock for the memory location defined in register RA. The return code is written to register RT. A value of 0 in RT means "SUCCESS" and a value of -1 means "FAIL".

SWSR1_R RT,RA

This instructions tries to read a value from the memory location defined in register RA. This instruction implicitly writes to two registers. The return code is written to register RT and the return value is written to register RT+1. A value of 0 in RT means "SUCCESS" and a value of -1 means "FAIL". If the read was successful, then the value of the memory location is available in RT+1.

SWSR1_W RT,RA,RB
This instructions tries to write the value specified in RB to the memory location defined in register RA. The return code is written to register RT. A value of 0 in RT means "SUCCESS" and a value of -1 means "FAIL".

SWSR2_R RT,RA
This instructions tries to read a value from the memory location defined in register RA. This instruction implicitly writes to two registers. The return code is written to register RT and the return value is written to register RT+1. A value of 0 in RT means "SUCCESS" and a value of -2 means "WAIT". If the read was successful, then the value of the memory location is available in RT+1.

SWSR2_W RT,RA,RB
This instructions tries to write the value specified in RB to the memory location defined in register RA. This instruction implicitly writes to two registers. The return code is written to register RT and the thread id (TID) to RT+1. A value of 0 in RT means "SUCCESS", a value of -1 means "FAIL", and a value of -2 means "NO WAITER".

The Thread Units (TUs) instruction decoder was extended to support the additional SSB instructions. Some of the instructions require more then one result register. Due to restrictions in the instruction format, crossbar package format, and in the register file, we use the result register and the next following register as bundled result registers. For example the Single-Writer-Single-Reader instruction swsr1_rd r6,r8 reads a signed double word value from the address specified in register r8. The return code is written to register r6 and the value is written to register r7. The write-back register is selected to be the next register after the return-code register in the register file. The Storage Interface (SI) in the TU was adapted to handle this special case and to generate crossbar packages for the new instructions if necessary.

SSB was placed in the SI of each TU. The SI is responsible for the arbitration of the memory requests coming from the network (crossbar) or the TU for the SRAM. Furthermore, it has to handle returning data from the network to the TU. The diagram in Figure 4 shows the relevant parts of the SI in the processor for SSB. Memory requests for the SRAM can originate from the TU or the network. A simple Least-Recently-Used (LRU) schema is used to arbitrate between these requests. With the addition of SSB, the arbitration schema had to be modified. Since both TU and network can produce SSB requests, a LRU schema is used at the entrance of SSB. We also keep the LRU schema for the normal memory requests to the SRAM, but if SSB has a memory request for the SRAM it takes priority over all other memory requests for the SRAM. This approach is still fair, because LRU was already performed at the entrance of SSB and due to the serialization effects of the crossbar interconnect.

The SSB combinatorial logic itself is rather simple and does not require much hardware resources (less than 4.4%). The only resource that is limited is the size of the SRAM that is used to store the additional tag information. The SSB in this implementation is 8-way associative and 39 bit wide for each entry. The required fields for a SSB entry in this architecture are: *State*, *Cnt*, *Address*, *Processor ID (PID)*, *Thread ID (TID)*, and *Size*. The size of each field is

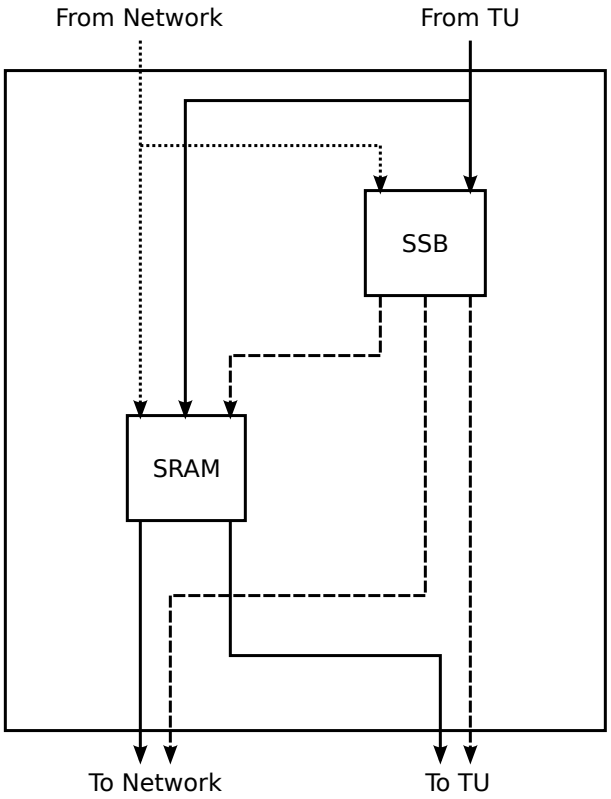


Figure 4: Simplified Storage Interface

Table 3: SSB Entry

Field	**State**	**Cnt**	**Address**	**PID**	**TID**	**Size**
Size(bit)	4	8	15	7	3	2

Table 4: SSB States

State	**Function**	**Description**
0	INVALID	Invalid
1	RLOCK	Read Lock
2	WLOCK	Write Lock
3	WRLOCK	Write-Recursive Lock
4	SWSR1	Single-Writer-Single-Reader Mode 1
5	SWSR2	Single-Writer-Single-Reader Mode 2
6 - 15	n/a	reserved

shown in Table 3. The State field is further explained in Table 4.

The SSB creates special network return packages to accommodate support for SSB return codes, interrupts and performance counters. The format of the SSB return packages is shown in Table 5. *TrCode* specifies the type of package. *Int* is used to raise a SSB interrupt. The

Table 5: SSB Return Package

Field	**TrCode**	**Int**	**Sync**	**SSB Code**	**PID**	**TID**	**Error**	**GPR**	**Data**
Size(bit)	6	1	1	3	7	3	1	6	64

interrupt is always risen in the TU that produced it and not in the TU were the SSB is located. This is necessary, because even if a TU is turned off, its SRAM can still be accessed by other TUs. *Sync* is used by the performance counter. It indicates if a SSB load was successful or not. *SSB Code* is sign-extended to 64bit and written to the register specified in the GPR field. *PID* and *TID* are used by the crossbar for routing. *Error* has the same behavior as for normal memory operations and raises an External interrupt. This normally happens when a user level store tries to access protected data or a load/store access is outside the valid memory space. The content of the *Data field* is written to register GPR+1.

4 Our Observations

During our implementation of the Synchronization State Buffer (SSB) into the IBM Cyclops-64 (C64) architecture, we encountered several logic bugs. In this section we will describe our experiences with the simulation and emulation framework and how they helped us to reach our key observations. A list of the encountered bugs is displayed in Table 6. At the end we were able to run all our benchmarks successfully until completion and they generated the correct result. Even for small problem sizes we could observe substantial speedup if SSB synchronization constructs were used compared to the existing synchronization construct (hardware barrier) of the original architecture.

Table 6: Hardware and Software Bug List

Bug #	Symptom	Description	Development Stage	Difficulty of Discovery	Platform
1	Test program hangs after SSB instruction	Used the wrong load signal for a latch during register read stage of the pipeline. As result the scoreboard bits are continuously set and the following instruction will wait forever for the bits to be cleared	early	very easy	Simulation

continued on next page

continued from previous page

Bug #	Symptom	Description	Development Stage	Difficulty of Discovery	Platform
2	Test program for SWSR Mode 2 fails	Tag was prematurely removed from the SSB.	early	very easy	Simulation
3	SSB interrupt	Tag was removed when the state changed from RLock to WRLock.	early	very easy	Simulation
4	Memory operations return wrong results	Redesign of the crossbar package format changed the encoding of operations. The location of the size field is not fixed and depends on encoded operation. As result the decoder was returning the wrong size.	early	very easy	Simulation
5	Test program for SSB fails	SSB return code was not sign-extended to 64bit	early	very easy	Simulation
6	Test program hangs. Load operations do not return.	The memory controller has a return FIFO. If the FIFO is full it does not accept any more memory request. Changes introduced due to SSB broke the FIFO counter which blocked the memory controller after four load operations.	early	easy	Simulation
7	Small benchmark fails. Atomic add does not work.	Changes to the crossbar package format removed one bit from the opcode that atomic memory operations use.	early	easy	Simulation
8	Small benchmark fails. Unexpected write back to register file.	Logic bug in memory controller leads to load/store duplication if SSB was used one cycle before.	early	easy	Simulation
9	Test program for SSB fails. Non-double word SSB operations return wrong result.	Memory controller does not block SSB operations correctly, because they are delayed by one cycle due to SSB lookup.	late	easy	Emulation
10	Kernel hangs.	The major redesign of the Storage Interface (SI) due to Bug 9 created several bugs, because the interface behavior between Thread Unit (TU), Load Store Multiple (LSM) Unit, Network A (NA), Network B (NB), SSB and memory controller were not completely understood.	late	difficult	Emulation

continued on next page

continued from previous page					
Bug #	Symptom	Description	Development Stage	Difficulty of Discovery	Platform
11	Benchmark hangs, because load return package was lost	Bug in crossbar interface. Only exposed during heavy network traffic. A normal return package can get lost, if the previous return package was a SSB return package, which requires two cycles to process.	very late	very difficult	Emulation
12	Benchmark generates wrong result	Bug in instruction scheduler of the compiler	very late	very difficult	Emulation

4.1 Key Observations

During the process of implementing SSB in the C64 architecture we deduced the following key observations thanks to the development and debugging information:

- Fast compilation support during early development phase proved to be very helpful in designing new features, testing the design and finding bugs. For more details see Section 4.2.

- Full-system emulation was indispensable to find deep buried system bugs. For more details see Section 4.3.

- High-speed emulation is crucial to run parallel applications and benchmarks for performance evaluation. For more details see Section 4.4.

4.2 Fast Compilation and Simulation

During early development, we used the simulation platform to create new architecture designs within minutes and simulated a smaller C64 chip with just ten Thread Units (TUs). This gave us faster simulation speed and we could use assembly code from the beginning to test our new design. This saved us the time of writing tedious unit tests for sub modules. This was especially helpful, because the interface behavior between certain sub modules were rather complicated or sometimes unexpected. Most of the bugs (Bug 1-8) were found in simulation mode. Only very few bugs were found in the SSB core component (Bug 2, 3 & 5). Most of the other bugs were related to unclear interface behavior (Bug 8) or very simple logic bugs (Bug 1, 4, 6 & 7). Moreover, after a bug has been discovered in emulation mode and bigger changes were necessary (Bug 10), switching back again to simulation mode for implementation and initial verification proved to be very helpful.

4.3 Full-System Emulation

After we had removed all the bugs we could find in simulation mode with our simple assembly test cases, we moved the design to emulation mode. Even though it takes more time to synthesize the design (around 10 hours), we could run more extensive test cases written in C, which freed us from the burden of low-level assembly programming. This allowed us to write much more extensive tests in a shorter amount of time and to run test on the full system. Due to this, we found two more bugs (Bug 9 & 10).

4.4 High-Speed Emulation

Finally, after all the bugs had been fixed we could run our parallel applications and benchmarks to validate the performance benefits of fine-grain synchronization. Even though all our test cases passed, we still encountered a very deep buried system bug (Bug 11) with our benchmarks. This bug will not show up in the real architecture and is only triggered by our SSB extension. These kinds of bugs show how important it is not only to do a full-system verification, but to actually run real parallel applications on the platform. We could not have done this without our high-speed emulation system. After we fixed the last hardware bug, we were able to run our benchmarks until completion, but the checksum failed. This time it was the compiler's fault (Bug 12). It did not schedule our new SSB instructions correctly. After we fixed this issue, all the benchmarks ran until completion and produced the correct results.

5 Related Work

There have been many logic verification technologies and products developed in both academia and industry [6, 7]. These technologies have been used to address the many challenges in the logic verification processes for a wide range of digital system designs.

The iterative emulation methodology, which is adopted in the emulation mode of DEEP, was introduced by Dr. Sakane et al. [1] in 2003 and the idea of iterative emulation for identical logic module instances with an FPGA was invented. They used shift registers to hold the state bits of a group of identical logic module for their target system of the 32bit Cyclops-E [8] architecture which has 8 processors. Their emulation system was implemented into one Xilinx Virtex-II FPGA because of the limited size of the verification target. Moreover, there is no debugging support, so most of the software debugging was done with a small number of control registers integrated in the emulation system. From this knowledge and experience base the development of the DEEP emulation system was nurtured. Key concepts such as importance of fast turn-around time and good debugging support, and the effectiveness of the iterative emulation for high emulation speed were born from these experiences.

ASIC based logic verification environments and emulation systems have been developed from many Electrical Design Automation (EDA) tool vendors. These tools have been designed for

high emulation speed, fast turn-around time, and powerful logic debugging support. Mentor Graphics Veloce platform and Cadence Palladimu system are good examples. The Veloce platform [9] is an ASIC based logic verification system developed by Mentor Graphics. This hardware accelerated logic simulation platform utilizes a custom designed emulation chip that contains a programmable logic block for the target logic and a fixed functional block. This fixed functional block handles signal tracing and interconnecting operations. The target logic is synthesized, partitioned and mapped into the emulation chip by Veloces software tools. The many emulation chips in the Veloce box are running in parallel to achieve a very high-speed emulation environment. The control software provides the interfaces for logic debugging and external IO simulation. The Palladium platforms [10] from Cadence provide simulation acceleration and in-circuit emulation in a single system. With its incorporated peripherals, embedded processors and multiple ASICs, the Palladium platform helps the chip validation engineers to achieve first silicon and system software required for successful delivery. Both platforms are gaining popularity among major chip development companies. When it comes to the many-core system validation project, both platforms have their limitations in capacity and speed because their target is general logic designs and not many-core processor architectures. Even though both platforms are commercially available, their prohibitive prices prevent most academic researchers to have access to them.

The RAMP [11] system developed at Berkeley is a FPGA-based many-core emulation platform. This system deploys Xilinx Vertex-II Pro FPGAs on 16-21 BEE2 boards [12] to implement a many-core system composed of 1000 plus cores. The purpose of this project is to explore the architectural design space for future many-core computer architectures and enable early software development and debugging. The proposed RAMP Design Framework (RDF) addresses the challenge of supporting both cycle-accurate emulation of detailed parameterized machine models and rapid functional-only emulations. Compared with the DEEP system, the RAMP system is intended to define and create the next generation of tools for computer-architecture and computer-science research, while the DEEP system was developed to validate the chip logic design of a real many-core architecture under more demanding requirements for capacity and speed.

Fine-grain synchronization has been supported in several architectures before, like (HEP [13], Monsoon, Tera [14], MDP [15], Cedar, Multicube, KSR1, Alewife/Sparcle [16], M-Machine, J-Machine, [17], Eldorado [18], and others).

6 Summary

We implemented the Synchronization State Buffer (SBB) in the IBM Cyclops-64 (C64) many-core architecture. We used the DEEP simulation and emulation framework for design and implementation. We performed full system logic verification and ran parallel applications on the whole chip in emulation mode. We used the debugging capabilities of simulation and emulation system extensively to fix all the logic bugs we found in our design. Thanks to the high speed of our emulation system we were able to easily find hidden bugs, which would have

been otherwise only encountered after chip production. Based on the bugs we encountered we concluded that a chip development framework should at least provide the following features:

Fast compilation of the logic design during the early stages of logic design to allow for quick turn-around times. Our system can compile the full C64 chip logic in less than two minutes for software simulation.

Full-System hardware emulation with extensive and fast debugging support capabilities is a must for whole chip logic verification. Fast emulation speed is required to find bugs before the chip is produced and not years after it has been phased out. DEEP's emulation mode allows such bugs to be found very quickly.

High-Speed hardware emulation (over 20KHz) of the whole system is an invaluable tool for system software development, performance prediction and evaluation. Important user applications can be run before chip production to validate the key features of the new design and its actual performance impact. This also allows the experimentation of novel program and code generation paradigms within reasonable time.

We believe that development system that incorporate these key features will have an tremendous impact on future architecture design methodologies and time-to-market.

This emulation environment will allow us to further research new hardware features and enhancements. We intend on implementing other synchronization constructs with more favorable properties and perform an in-depth performance evaluation and hardware resource cost analysis.

Acknowledgements

We like to thank all the voluntary reviewers for their comments and help. We greatly appreciate the support from ET International for providing the system software and in particular Jean Christophe Bayler for modifying the toolchain to support our hardware extension. We deeply appreciate the original idea for the emulation system by Hirofumi Sakane and Fei Chen for the actual implementation and providing us such a great framework. Our utmost respect goes to Monty Denneau for creating such a great architecture and the possibility to work with it. Thanks to all CAPSL members for their support. Special thanks go to Joseph Manzano for his countless hours of support. This work would have not been possible without the support by NSF (CNS-0509332, CSR-0720531, CCF-0833166, CCF-0702244), and other government sponsors.

References

[1] H. Sakane, L. Yakay, V. Karna, C. Leung, and G. Gao, "DIMES: An iterative emulation platform for multiprocessor-system-on-chip designs," in *2003 IEEE International Conference on Field-Programmable Technology (FPT), 2003. Proceedings*, 2003, pp. 244–251.

[2] W. Zhu, V. Sreedhar, Z. Hu, and G. Gao, "Synchronization State Buffer: Supporting Efficient Fine-Grain Synchronization on Many-Core Architectures," in *Proceedings of the 34th annual international symposium on Computer architecture*. ACM, 2007, p. 45.

[3] Y. M. P. Zhang, T. Jeong, F. Chen, H. Wu, R. Nitzsche, and G. R. Gao, "A Study of the On-Chip Interconnection Network for the IBM Cyclops64 Multi-Core Architecture," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, 2006, p. 10.

[4] J. del Cuvillo, W. Zhu, Z. Hu, and G. Gao, "TiNy Threads: A Thread Virtual Machine for the Cyclops64 Cellular Architecture," in *19th IEEE International Parallel and Distributed Processing Symposium, 2005. Proceedings*, 2005, p. 8.

[5] J. Del Cuvillo, W. Zhu, Z. Hu, and G. Gao, "Toward a Software Infrastructure for the Cyclops-64 Cellular Architecture," in *High-Performance Computing in an Advanced Collaborative Environment, 2006. HPCS 2006. 20th International Symposium on*, 2006, pp. 9–9.

[6] M. Dubois, J. Jeong, Y. Song, and A. Moga, "Rapid hardware prototyping on rpm-2," *IEEE Des. Test. Comput., July-September*, pp. 112–118, 1998.

[7] J. Darringer, E. Davidson, D. J. Hathaway, B. Koenemann, M. Lavin, J. K. Morrell, K. Rahmat, W. Roesner, E. Schanzenbach, G. Tellez, and L. Trevillyan, "EDA in IBM: Past, Present, and Future," *IEEE TRANSACTIONS ON COMPUTER AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS*, vol. 22, pp. 1476–1497, 2000.

[8] G. Almási, C. Caşcaval, J. G. C. nos, M. Denneau, D. Lieber, J. E. Moreira, and H. S. Warren Jr, "Dissecting Cyclops: A Detailed Analysis of a Multithreaded Architecture," *ACM SIGARCH Computer Architecture News*, vol. 31, no. 1, p. 38, 2003.

[9] M. Graphics, "Veloce soc verification system," http://www.mentor.com/.

[10] Cadence, "Incisive palladimu series, scalable, high-throughput, emulation platform," http://www.cadence.com/.

[11] J. Wawrzynek, D. Patterson, M. Oskin, S. Lu, C. Kozyrakis, J. Hoe, D. Chiou, and K. Asanovic, "Ramp: Research accelerator for multiple processors," *IEEE Micro*, vol. 27, pp. 46–57, 2007.

[12] C. Chang, J. Wawrzynek, and R. Brodersen, "Bee2: A high-end reconfigurable computing system," *IEEE Design and Test of Computers*, vol. 22, pp. 114–125, 2005.

[13] B. J. Smith, "Architecture and applications of the HEP multiprocessor computer system," *Real-Time Signal Processing IV*, vol. 298, pp. 241–248, 1981.

[14] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, "The Tera Computer System," *ACM SIGARCH Computer Architecture News*, vol. 18, no. 3b, pp. 1–6, 1990.

[15] W. J. Dally, L. Chao, A. Chien, S. Hassoun, W. Horwat, J. Kaplan, P. Song, B. Totty, and S. Wills, "Architecture of a Message-Driven Processor," in *Proceedings of the 14th Annual International Symposium on Computer Architecture*. ACM New York, NY, USA, 1987, pp. 189–196.

[16] A. Agarwal, J. Kubiatowicz, D. Kranz, B.-H. Lim, D. Yeung, G. D'Souza, and M. Parkin, "Sparcle: An Evolutionary Processor Design for Large-Scale Multiprocessors," *IEEE micro*, vol. 13, no. 3, pp. 48–61, 1993.

[17] S. W. Keckler, W. J. Dally, D. Maskit, N. P. Carter, A. Chang, and W. S. Lee, "Exploiting Fine-Grain Thread Level Parallelism on the MIT Multi-ALU Processor," *ACM SIGARCH Computer Architecture News*, vol. 26, no. 3, pp. 306–317, 1998.

[18] J. Feo, D. Harper, S. Kahan, and P. Konecny, "ELDORADO," in *Proceedings of the 2nd Conference on Computing Frontiers*. ACM, 2005, p. 34.