PDSparc: A Drop-In Replacement for LEON3 Written Using Synopsys Processor Designer[1]

David Whelihan, Ph.D. and Kate Thurmer MIT Lincoln Laboratory, Lexington, MA, USA

Distribution A: Public Release

ABSTRACT

Microprocessors are the engines that drive the modern world. For decades this space has been dominated by large manufacturers, such as Intel and AMD, which design and fabricate a range of stand-alone processors. However the proliferation of small computing devices such as cell phones, laptop computers, and internet-enabled appliances has opened a significant new niche: the Application Specific Standard Product (ASSP) microprocessor. These processors usually start out as soft-cores that are parameterized at design time to realize exclusively the specific needs of the application. The microprocessor is a small part of a working system and requires peripherals such as DRAM controllers and communication sub-systems to properly carry out its function. Therefore, creating a full system requires significant top-level integration.

This work introduces PDSparc, an ASSP based on the OpenSparc architecture. PDSparc was generated using the Synopsys Processor Designer (PD) tool, which enables detailed specification of a pipelined processor using a C-like language called LISA [13]. PDSparc replaces a LEON3 processor, a derivative of the Sparc v8 microarchitecture [5], in a full synthesizable SoC system [3] provided by Cobham Gaisler under the Gnu Public License. This integration required significant reverse engineering of the provided cache interfaces, and was facilitated by a novel socket-based debug interface to the PDSparc simulation model. This debug interface greatly accelerated system development by permitting micro-stepping of the PDSparc processor code in synchronization with the peripheral and bus package running on a commercial RTL simulator.

We anticipate that this paper will be of interest to the practitioners in the field who can use the approach and the lessons learned described in the paper in building their own processor execution cores.

[1] This work was sponsored by the Assistant Secretary of Defense for Research & Engineering under Air Force Contract #FA8721i 05i Ci 0002. Opinions, interpretations, recommendations and conclusions are those of the authors and are not necessarily endorsed by the United States Government.

Table of Contents

Table of Figures

1. Introduction

Microprocessors are ubiquitous in today's increasingly computerized world, appearing in watches, toys, industrial control applications, and enterprise systems. The broad range of applications and form factors, as well as varying size, weight and power constraints, means that a one-size-fits-all processor is insufficient to meet the needs of either the commercial or military spaces. The result is a wide variety of choices for processor architecture, from the heavyweight Intel Xeon [1] to ARM M-series embedded processors [2], and many in between. The success of the most popular processor lines is due to a number of factors, including compatibility with existing code bases, availability of compilation tool chains, and performance.

The prevalence of inexpensive FPGA technology, which facilitates rapid production of arbitrary designs, provides an opportunity for users to build custom processors that deliver exactly what the application requires. Such processors, known as Application Specific Standard Products (ASSP), are distributed as Register Transfer Level (RTL) models that can be synthesized using industrial tools into nearly any hardware technology. These processors are used extensively throughout the design process, from prototyping to full production of critical military and commercial systems. The goal of this project was to rapidly create a reference baseline ASSP processor that could be used to demonstrate and even productize novel system features for DoD applications. MIT LL chose Synopsys Processor Designer (PD) as a tool to enable rapid creation of an execution pipeline called PDSparc that would match the OpenSparc v8 [5,6] specification. Further, this execution pipeline would replace an existing processor in an open SoC environment called GRLIB [3]. This enables unprecedented configuration freedom; processor instructions and interfaces can be added to a full-fledged SoC with ease using PD. This paper describes the process of designing a drop-in replacement for the LEON3 [4] variant of Sparc v8 processors.

This paper is intended for an audience experienced in RTL system design and conversant in microprocessor architecture. The goals of this work are twofold:

- To describe the creation of an open, flexible reference design that can be extended to multiple applications by enabling Instruction Set Architecture-level customization in the context of a full-featured, synthesizable system.
- To describe how the LISA language as interpreted by Processor Designer can be used to build non-trivial execution cores capable of utilizing advanced system features such as streaming caches.

The paper is structured as follows: Chapter 2 details the overall structure of the PDSparc SoC environment. Chapter 3 provides a brief overview of the Processor Designer tool suite, and then describes how it was used to build PDSparc. Chapter 4 describes the GRLIB SoC environment, as well as the challenges faced when interfacing it with our custom processor. Chapter 5 details a novel verification environment that extends some PD features to deliver source-level debugging of PD processor code simultaneously with source-level debugging of processor instructions and RTL wave tracing. Finally, Chapters 6 and 7 discuss conclusions and future work.

2. PDSparc

PDSparc is a general purpose processor system created to facilitate experimentation and customization of embedded processing for a wide variety of DoD applications. It is based on an open-source LEON 3 processor, which conforms to the Sparc v8 specification, and an SoC support system made available by Cobham Gaisler [7] under the Gnu Public License. The SoC package, GRLIB, includes a processor system generator that outputs a LEON3 multi-core processing system with between one and four cores, an L1 cache, interrupt controllers, DRAM controllers, and a number of other peripherals. GRLIB is also supplied with the scripts and configuration necessary to target many popular commercial FPGA boards.

The configurability of the GRLIB system makes it very attractive for a variety of prototyping needs. Its use of the time-tested Sparc v8 architecture, which has a great deal of software support and industrial momentum [8], makes the GRLIB system extremely versatile. This configurability enables the creation of custom systems that supply just the processing power needed by an application, and no more, which is useful for applications in low-power environments.

Yet another level of configurability is possible: custom instructions. Most processor architectures today have fixed set of operations that they can perform. These operations are exported to the user as a set of instructions that embody the processor's Instruction Set Architecture (ISA). Processors have historically been divided into Complex Instruction Set Computers (CISC), in which complex operations are rolled up into single instructions, and Reduced Instruction Set Computers (RISC), in which it is up to the application writer (or compiler) to build complex capabilities from a relatively small set of simple operations. The advantages of RISC over CISC architectures are many, including the ability to easily pipeline operations (because all or most operations take the same amount of time to execute), and speed, as critical paths can be much shorter.

The RISC model is good for generalized computing, in which a wide variety of applications require good performance. However in the case of embedded processors, which serve very specific processing demands and require higher performance, some specialization is desirable. Such specialized processors are sometimes known as Application Specific Standard Products (ASSP). For example, a signal-processing ASSP might include a special instruction that performs FFT operations much more quickly by utilizing dedicated logic to perform the operation. The goal of the PDSparc is to create a configurable processor system with the added capability to tailor the instruction set to the application at hand. The tailored instruction set is enabled by Synopsys' Processor Designer Tool [9].

3. Architecting PDSparc Using Processor Designer

Synopsys Processor Designer is a tool that automatically generates synthesizable computer execution pipelines with arbitrary instruction sets from a processor definition written in the Language for Instruction Set Architectures (LISA) [13]. LISA is a C-like language first

developed in 1997 at RWTH Aachen University. It simplifies the specification of processing pipelines and the instructions they support by reducing processing logic to trees of *operations*. Pipelines are specified by listing the names of the pipeline stages, and then the registers separating those stages. For example, PDSparc's five-stage instruction pipeline is defined in Figure 1 (not all pipe registers are shown for brevity).

```
PIPELINE         pipe = { FE; DC ; EX ; MEM; WB ; INST_DN};

  /* A pipeline register to pass data through the pipeline */
  PIPELINE_REGISTER IN pipe
  {
    PROGRAM_COUNTER uint32 pc; /* the address of the
                                  instruction */

    uint32 insn0;  /* 32-bit instruction register */

    unsigned bit[10] bp_idx_src0;  /* operand 1 register index
                                   (as source register) */
    bool   operand_src0_is_provisional; /* this is used if we
                                           have to recompute
                                           branch addresses in
                                           writeback */
    uint32 operand_src0;          /* operand 1 value */

 …
```

Figure 1: The PDSparc LISA instruction pipeline definition

In LISA, the PIPELINE command defines the names of the pipeline stages. There are registers, defined by the PIPELINE_REGISTER command between each set of pipeline registers, FE/DC, DC/EX, EX/MEM, MEM/WB, where FE is the fetch unit, DC, is the instruction decode unit, EX is the execution unit, MEM is the data cache memory input, and WB is the writeback unit that commits state to the processor register file. The last stage INST_DN is a dummy stage that ensures that the tool will place state elements after the Writeback (WB) stage. In addition to the Instruction pipeline, PDSparc has a tightly coupled three stage load/store pipeline. The full architecture is illustrated in Figure 2. In that figure, each pipeline stage has a register set at its inputs, outputs, or both. All logic in between those stages is combinational (unclocked), and constitutes the functionality of that particular stage. A particular stage is *stalled* when the output of the combinational stage is not registered in its downstream register set by disabling update functionality of the register. Stalls are explicitly called in the LISA language by addressing the stage to stall and which register set (IN or OUT) to stall. For example, to stall the Execution unit (EX) in the next cycle, the LISA writer would include EX.OUT.stall() in their code. This would prevent the EX/MEM state register set from clocking in the result of the EX logic. This

command will generally also stall the DC/EX and FE/DC registers as they are upstream of the stalled register and cannot forward their data. This stall will result in a pipeline bubble (an undefined operation that has no effect) passing through to the Writeback (WB) stage. This automatic stall mechanism greatly simplifies the task of managing the pipeline to maintain performance by keeping the pipeline full- despite confounding factors such as load/store access delays.

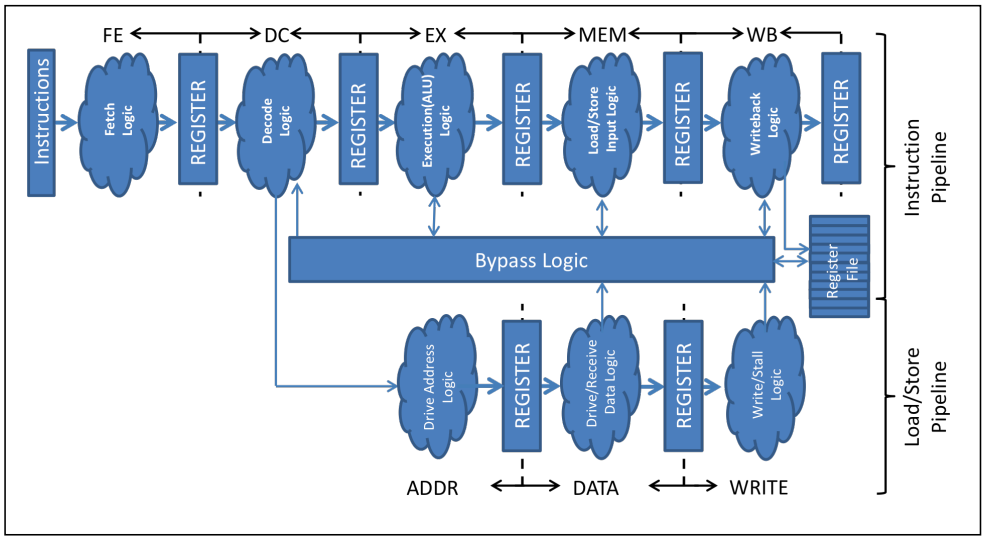

Figure 2: PDSparc pipelined architecture

There are two kinds of stall conditions in a simple RISC processor: instruction fetch stalls, in which the next instruction is not yet available to insert into the pipeline when there is space to do so, and load/store stalls, in which the instructions executing in the pipeline require loads or stores to memory[3]. Because either can stall the pipeline, it is difficult to extricate one or the other during development. Rather, the fetch unit, which ordinarily only fetches instructions must also be cognizant of the state of the load/store logic, because a load/store stall downstream has the same effect as a fetch stall upstream: the pipeline cannot advance, and no new instructions may be inserted into the pipeline.

Nonetheless, the processes of instruction fetch and load/store access have their own semantics and protocols. This is the reason that PDSparc has two pipelines: one for instruction execution, and one for load/store accesses. Processor Designer permits the definition of multiple pipelines, and they can easily be made to interact via signals. As will be described in the next section, the protocol and functionality of a streaming instruction cache which strives to make sure that an instruction is ready for consumption by the instruction pipeline at every cycle, is very different

[3] We are not including stalls due to asynchronous interrupts, or multi-cycle arithmetic operations in this discussion.

from the protocol and functionality of a random access data cache. Thus, separating the two but ensuring that they maintain a lockstep relationship results in much cleaner and extensible code.

Interfacing to external systems in PD is relatively straightforward as LISA defines a number of pin types that can be used to hook directly to standard busses. Pins can be defined in the main LISA code for the processor, or in special modules called *protocols*. In the design of PDSparc, two protocol features were used: a C++ test driver, and a LISA module that represents the true bit-accurate functionality of the protocol. Consistent with the rest of PD, protocols are made of operations. Those operations have inputs and outputs, and generally encode elements of more complex protocols, such as busses. The use of a protocol called cache_ifc is shown in the code block shown in Figure 3.

Authorized User 6/22/2015 2:33 PM
Deleted: <sp><sp>
Unknown
Formatted: Font color: Text 1, Pattern: Clear (White)

In that code, an operation called *write_phase*, which is bound to the third stage in the load/store pipeline *ldst_pipe.WRITE*, uses C++-like method calls to call operations on the protocol cache_ifc, which in turn sets signals connected to the processor's level 1 instruction and data caches. The protocol abstraction is very useful in test and development, because the second feature of the protocol, the C++ driver implementation, can be used to interface to any test environment, or emulate any connected device. The test interface has an identical signature to the LISA protocol, except that internally, it is actually a C++ method on a C++ class.

The Processor Designer LISA model is compiled by the tool to create an Instruction Set Simulator (ISS). The ISS is a static library (.a) file that can be linked into any program in order to drive tests on the architecture. When the ISS is used, rather than utilize the LISA protocol operations, the tool links with the C++ protocol code to provide stimulus and record outputs. This allows the use of all C++ functions and libraries, from file IO to sockets (which will be important later in this paper), and enables high-level debugging and test automation.

The ISS, when linked with a debug interface supplied with PD can single-step through assembly code running on the processor, but more importantly, it can micro-step through the LISA code between clock cycles. This enables source-level debugging of the processor in a sequential, rather than concurrent (as in HDL debugging) style. This is enabled by the PD scheduler, which schedules operations such that concurrent operations appear sequential, but all dependencies are met. Thus it is possible to step through all active combinational paths at the LISA code level. This powerful technique makes engineering complex logical structures such as bypass paths, which direct data backwards in the pipeline for immediate use, much simpler.

When the processor is logically correct, and verified through the ISS, the user can run a tool called Processor Generator (PG), which converts the LISA code into Verilog, VHDL, or both Hardware Description Language (HDL). When PG runs, it uses the LISA protocol operations, rather than the C++ protocol operations that the ISS linked with. The result is a pure RTL version of the processor with actual pins that will be driven by external logic. PG will generate human readable HDL, and it will also insert JTAG debug capability for increased visibility.

However, in the move to HDL, we necessarily lose debug flexibility. In the next section, the processor-cache interface will be described. Developing the proper protocols required reverse engineering the GRLIB streaming cache, which was not a small effort, and demanded maximum visibility. Therefore, MIT LL developed a hybrid debug approach that used the protocol interface to connect the ISS, which affords extreme visibility, directly to the GRLIB RTL cache and other subsystems written in HDL, all running on an industry standard HDL simulator. This was a tremendous advantage, and was made possible by the high degree of logical equivalence between the ISS LISA code and the PG-generated HDL.

4. Cobham Gaisler's GRLIB

The Cobham Gaisler GRLIB package is a configurable multi-core System-On-Chip(SoC) that can be targeted at many different industry-standard FPGA boards. This makes it ideal for prototyping new SoCs. GRLIB includes a graphical configuration toolchain for generation of the SoC RTL, and utilizes the *sparc-elf* [12] toolchain for program compilation. The GRLIB configuration chain generates synthesizable VHDL logic for all sub-systems, from instruction and data caches to RAM and ROM to Interrupt (IRQ) logic. It also will generate a configurable LEON3 processor based on the Sparc v8 specification.

The goal in this project was to replace the LEON3 with a PD-generated Sparc v8 processor that could be "dropped into" the existing GRLIB system. Processor Designer is targeted exclusively to the generation of execution pipelines, so the dividing line between the GRLIB RTL and the PDSparc processor is be at the cache-processor boundary. This meant that the processor needs to adhere to the cache functionality in GRLIB.

The most basic model of a processor cache interface involves an explicit address request, followed by data delivery. For a pipelined RISC processor however, such a relationship between the processor pipeline and the cache would result in many bubbles in the pipeline, as instruction fetch requests would take multiple cycles. In order to keep the pipeline full, a RISC system employs a streaming cache that delivers data to the pipeline on every cycle, unless *stalled*. The interesting thing about such a cache interface is that the seat of control is less in the processor pipeline, and more in the cache controller. This is because the pipeline is completely dependent

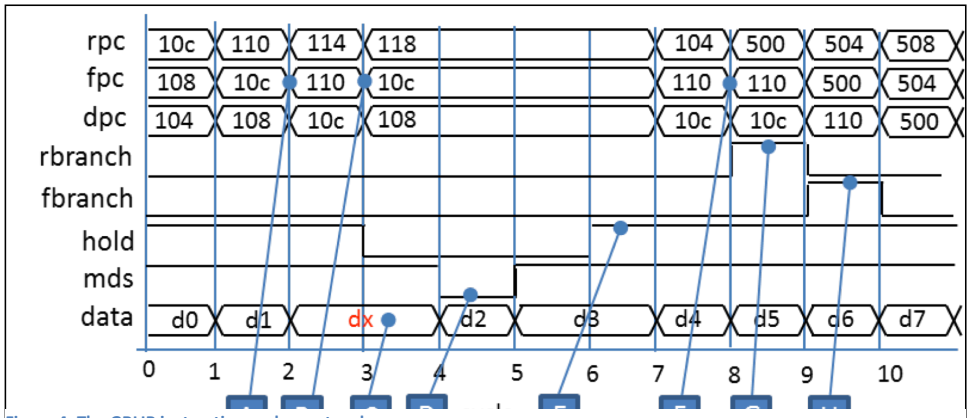

Figure 4: The GRLIB instruction cache protocol

on the cache feeding it with instructions and data. In a RISC, with no multi-cycle instructions, the only reason to stall the pipeline is the when data is unavailable.

There is therefore an extremely tight coupling between the cache and the pipeline. This also implies that the interactions are more complex, as are the protocols that tie the cache and the pipeline together. The task of reverse engineering the cache interface, and translating the protocol to a LISA-generated execution pipeline was not straightforward. In addition, the protocol and timing of the streaming instruction cache is markedly different from the random-access data cache. The task was therefore to design a pipeline that would handle the cache protocol and provide similar performance to the original LEON3. A trimmed down version of the protocol is shown in Figure 4.

The instruction cache interface is comprised of eight primary signals and busses. The three program counter busses, *rpc*, *fpc* and *dpc* are the mechanism by which the processor stays in sync with the streaming cache, and is able to receive instructions consecutively. *Rpc* is the request program counter, which is the next instruction to be fetched. *Fpc* is the fetch pc, which is the pc of the current instruction being actively pulled in by the FE stage of the pipeline. *Dpc* is the decode pc, and corresponds to the current instruction being decoded. The reason that all three busses are needed is that there are cases where the cache cannot deliver the proper instruction data, such as when a cache miss occurs. The cache miss sequence is illustrated in Figure 4 starting at time 2, marked as event A by a blue box. At that point *dpc* gets the value of *fpc*, *fpc* gets the value of *rpc*, and *rpc* gets the next address to fetch as instructions move through the pipeline. At time 2, the fetch (FE) unit clocks the new instruction into the pipeline by reading the data bus. At time 3 (event B), the sequence continues with the FE unit registering data for instruction 0x110, and the decode unit decoding instruction 0x10c. However, the active low hold signal driven by the cache is asserted, meaning that the last data received by the FE unit may be invalid. In cycle 3 (event C), the possibly bogus instruction is decoded by the DC unit, but no action is taken pending resolution of the hold condition. At time 4, the active low *mds* signal is asserted, signaling that the instruction pointed to by *dpc* is indeed bogus, and the correct value is being driven on the data bus. This triggers the DC unit to execute a flush, which has similar LISA syntax to the stall command: DC.OUT.flush(), thus invalidating the instruction currently being decoded. At that point, the instruction input to the DC unit is modified to reflect the true instruction value being driven on the data bus. This requires a LISA trick however, as the pipeline is currently stalled, and pipeline registers are not updated on clock edges due to the hold condition. LISA has several different kinds of register types, most notably REGISTER types and STATE_REGISTER types. All pipeline registers are of type REGISTER, which are not updated in a pipe stage under stall. However, STATE_REGISTERs are updated unconditionally. Rather than use the automatically generated REGISTER types for the DC unit instruction inputs, STATE_REGISTERs were used so that the DC unit input could be modified without releasing the pipeline from the stall.

The stall means that the pipeline is now behind, as the instruction on the data bus is the instruction being decoded, not the one currently being fetched. As soon as hold is deasserted (event E), the pipeline assumes that the value on the data bus is the currently fetched value. Since the pipeline will not increment *rpc* until the cycle after hold is deasserted, the pipeline is automatically resynchronized.

Branches are an important part of instruction fetch units, and if the Fetch and Decode units are not tightly synchronized, they can significantly degrade performance. The LEON3 processor in the original GRLIB configuration is able to fetch, decode, and branch in two cycles. This entails decoding the instruction, calculating the branch target, and driving the proper signals to redirect the cache. During that time, one more instruction will have been fetched. This instruction is in what is commonly called the branch delay slot, and is used to optimize performance and avoid pipeline bubbles by allowing the compiler to insert an instruction after a branch. PDSparc currently takes three cycles to branch, but a two-cycle branch should be possible in the future. Branches are initiated by driving the *rbranch* signal, and driving the new branch address onto *rpc* without incrementing *fpc* and *dpc* as shown at event G. The instruction in the *dpc* slot is the branch instruction, and the instruction in the *fpc* slot is in the branch delay slot. In the next cycle, the branch target transitions to *fpc*, and the *fbranch* signal is driven high (event H). After that point, instruction fetch continues normally.

The instruction fetch unit is relatively straightforward in that it assumes there is a stall condition at the DC unit or later in the pipeline if the instruction cache *hold* signal is asserted, or the data cache *hold* signal is asserted. This is based on the aforementioned tight coupling between the processor and the two caches. The cache makes the assumption that if load/store data - data bound/from the data cache - is delayed then the pipeline cannot move forward, and instruction fetch is stalled also. For reasons that will be explained shortly, the converse is also true, and an instruction cache hold implies a data cache hold/stall.

The data cache is significantly different from the instruction cache in that it is both streaming (back-to-back requests are possible), and random access. In addition, the data cache supports loads and stores, as opposed to only instruction loads. The data cache protocol has a similar structure to the instruction cache protocol in that it has a multi-cycle fetch process with hold conditions. To simplify the logic, and better match required functionality to the proper logic, PDSparc implements a separate load/store pipeline that operates in lockstep with the instruction pipeline. The load/store pipeline has three stages: an ADDRESS stage, in which the transaction address is driven on a bus called *eaddress*, a DATA stage, in which data to store is driven on a bus called *edata*, or data is loaded, and a final WRITE stage in which stores are committed. The process of performing streaming loads and stores is shown in figure X.Figure 5 In that figure, both the data cache bus and the instruction pipeline state are shown. Back-to-back store and back-to-back load instructions are shown as colored boxes on the instruction pipeline plot. The GRLIB cache supports streaming of two-cycle loads, which after a one cycle startup can be delivered on every subsequent clock, barring a hold condition. Stores however, require three

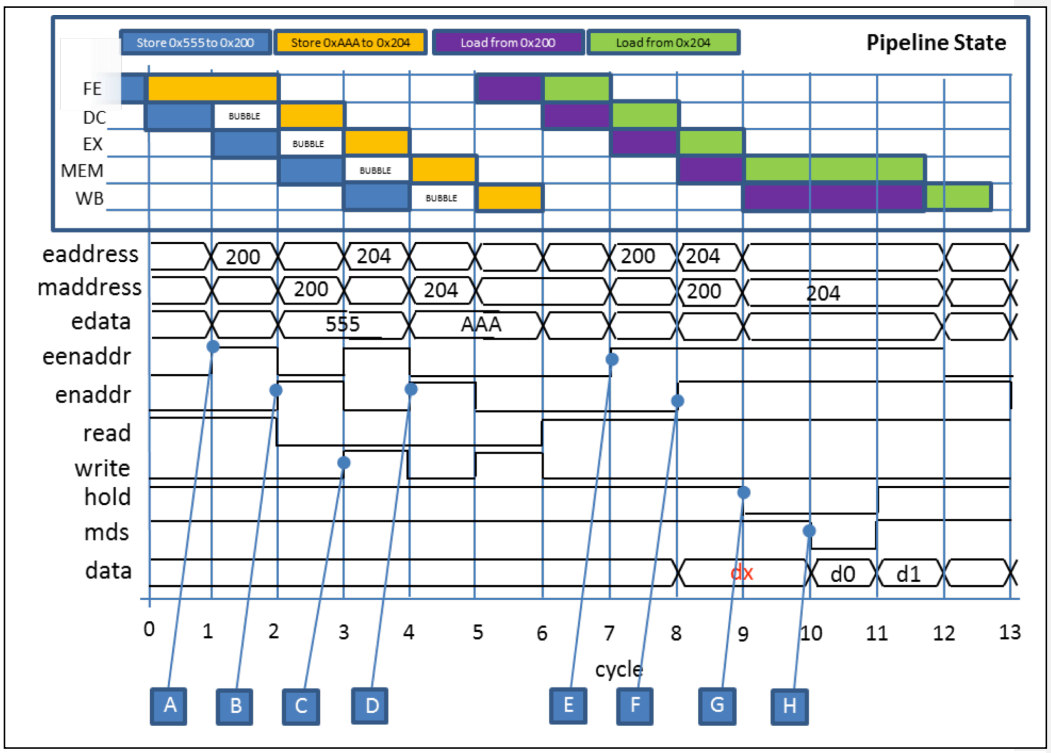

Figure 5: The GRLIB data cache protocol with PDSparc pipeline state

cycles with one cycle of latency before a new transaction can be initiated. Event A in Figure 5 shows a back-to-back store. At cycle1, the active high *eenaddr* signal is asserted, and the store address is driven onto the *eaddress* bus. While Cobham Gaisler's rationale in choosing of signal names is a bit murky, the assertion of these signals corresponds to when the store instruction has reached the EX stage in the instruction pipeline, leading to prepending an 'e' to the signal name. At cycle 2 (event B), *eenaddr* is deasserted and *enaddr* asserted. Simultaneously, the *maddress* bus drives the store address, the data is driven on the *edata* bus, and *read* is deasserted. At this point, the instruction has reached the MEM pipeline stage. It is important to note that while the instructions are moving through the instruction pipeline, they are having no effect on execution after the DC unit injected them into the load/store pipeline, which drives all protocol commands via LISA protocols. At event C, the final stage of the write is executed when the store request reaches the third stage of the load/store pipeline. In that stage, the *write* signal is driven.

While the first store to address 0x200 was executing, a second store was fetched (to address 0x204). Since the store mechanism requires a cycle of latency, the second store needs to wait one cycle before it can be injected into the load/store pipeline at time 1. This requires the injection of a "bubble" into the instruction pipeline, which is supported by the LISA code by calling

DC.OUT.stall() for one cycle. The streaming cache can also be stalled in that case by driving a signal called *inull*, and rewinding the *rpc* address. This signals the cache that it must wait, and ensures that no instructions are missed. Once the first load/store reaches the DATA stage in the load/store pipeline, it has accepted the write and leading to *eenaddr* and *eaddress* being asserted. At event D, the first store has retired from both pipelines, and the second store is in the MEM and DATA phases of the instruction and load/store pipelines, respectively.

The second transaction set in Figure 5 is a back-to-back streaming load. Loads utilize the same basic protocol as writes, but are vulnerable to hold conditions due to cache misses and other confounding situations. As in the store example, a load request to address 0x200 is issued at event E by driving 0x200 onto the *eaddress* bus and asserting *eenaddr*. A second load enters the instruction pipeline at event F and also drives these signals (with *eaddress* being driven to 0x204). In that cycle, *maddress* is driven to 0x200. The streaming protocol specifies that the data for the load of address 0x200 may be ready in this cycle. The load/store pipeline therefore reads from the *edata* bus, and signals to the instruction pipeline that data is available. However, the instruction pipeline treats this data as provisional, as the data cache cannot signal a miss until the cycle following the DATA cycle. This case is shown starting at event G, where the data cache hold signal is asserted. At that point, the instruction has reached the WB (writeback) stage, and is in the WRITE stage of the load/store pipeline.

There is a significant hazard at this point, as the instruction prior to the load may have just left the EX unit. If that instruction depended upon the loaded value, then whatever result computed in EX is incorrect. To avoid this, the PDSparc instruction decoder (DC) compares the destination register of the last instruction with the source registers of the current instruction. If there is a dependency, the DC will issue a pipeline bubble just like in the back-to-back store case, ensuring that the dependent instruction is no further than the EX unit on a data cache stall. This situation is even more complex for dependent branches, which are not covered in this paper. The real enabler for handling all of these dependencies however is a robust and clear set of bypass logic which gets just-received values to the source paths of currently executing instructions. This bypassing, which can be difficult to construct, is relatively simple in LISA as pipeline stages that must check for dependencies can activate one or more bypass operations that roll the process up into an if-then-else statement.

If there were a dependency between the first load and a hypothetical instruction, it would be fulfilled starting at event H, where *mds* is asserted and the correct data is delivered. When that happens, the load in the WRITE stage of the load/store pipeline signals to the WB stage of the instruction pipeline that the delayed load is ready. This percolates through the bypass logic to upstream instructions in the DC and EX units via the bypass operation.

5. PDSparc Verification

While Cobham Gaisler does provide excellent documentation on the high-level functionality and register mappings of the GRLIB IP cores, the details of the bus interactions were not explicitly spelled out, leading to a significant reverse-engineering effort. With an all-RTL development path this would have been incredibly difficult if not impossible. Fortunately, the Processor Designer tool's Instruction Set Simulator (ISS) proved instrumental in matching PDSparc to the GRLIB subsystem. One difficulty, however, was that the protocols employed by the ISS are written in C++, and are meant to drive test vectors. This is adequate for systems where the protocol is well understood, but for a minimally-understood interactive cache protocol, it proved difficult to match. The other option is to use Processor Generator (PG) to generate RTL that matches the ISS and connect that directly in place of the LEON3 processor in the RTL simulator. While PG generates an accurate representation of the LISA model, a significant amount of visibility is lost relative to the ISS when moving to RTL. The MIT LL team took a third approach: use the GRLIB RTL simulation to drive the ISS. This removed the need to "guess" the protocol and implement an emulator in C++ in order to do source and pipeline-level debugging of the processor LISA code. Ultimately, this proved to be a tremendous time-saver.

Because the LISA ISS debugger links with C++, it was possible to utilize the entire C++ library - including sockets. The team created a C++ class called the *cache_connector,* which encapsulates all of the signals travelling between the processor and GRLIB into C++ structures. On every cycle, the PD ISS calls a function that pushes the output structures onto a C socket and reads the input structures off of the same socket. Since the ISS runs in zero system time, every cycle the individual pipeline stages have the opportunity to manipulate the values in the output structures and read the values in the input structures.

The protocol named cache_ifc in the LISA context instantiates an object of type *cache_connector* whose constructor opens up a socket and waits for a connection. On the other side of the socket is an industry standard mixed-language simulator running the GRLIB RTL system with a processor stub module written in SystemC [10]. That stub module handles all input and output from the GRLIB RTL model, and itself instantiates the same *cache_connector* object that is instantiated in the ISS C protocol. Thus the cache connectors are always exchanging the same data structures across the socket.

Sockets work by running send and receive commands from their instantiating programs. When the ISS runs its protocol, for which there is a main operation that runs unconditionally, it calls a method on the *cache_connector* object called *cc_exchange* that exchanges data structures across the socket. The protocol main function will block on the *cache_connector*'s receive operation until the proper structures are read. In this way, the virtual system clock is similar to the eventual RTL reality in which the low-level SoC drives the clock into the processor. The SoC model activates the SystemC LEON3 stand-in on every clock edge. On the first positive clock edge, the systemC module transmits the current state of the cache signal outputs across the socket, and

waits for the ISS to send the cache signal inputs back. The ISS receives the cache output structures and sends the current value of the cache input structures, which are received by the systemC module and driven out into the cache RTL model.

To tighten the round-trip loop, the systemC module is active on both edges of the clock. On a positive edge, it sends the recently set values of the cache outputs to the ISS, and leaves the cache inputs unchanged. When the ISS receives these values, it ticks the clock on the ISS in zero simulation time, so the values to be latched in the next cycle are ready immediately. It is therefore possible to drive values on the next simulation clock by activating the systemC module on the negative edge of the clock. On the negative clock edge, the systemC module drives the just-received cache input signals into the model, and leaves the cache outputs unchanged to the ISS.

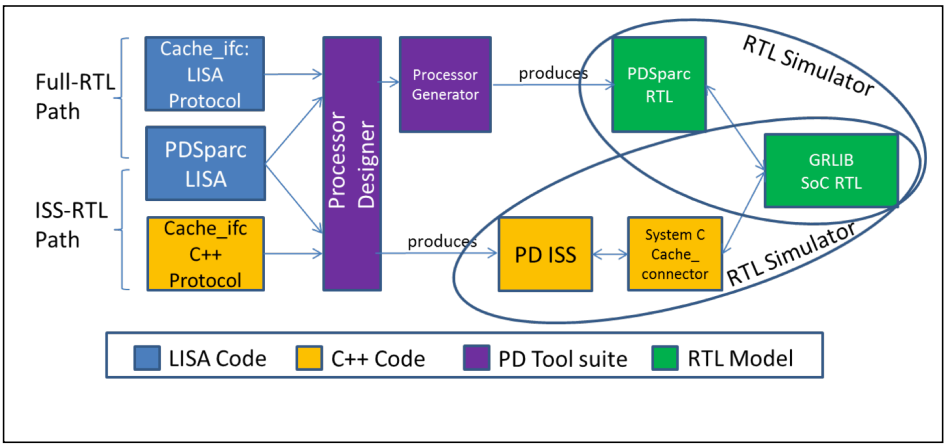

Figure 6: The dual simulation options for PDSparc

The result is that the ISS runs in tight lockstep with the GRLIB SoC model, and all vectors into the processor ISS are functionally correct, and will be nearly identical to the interactions between the Processor Generator generated HDL and the SoC[4]. What this enables is the ability to single-step both the GRLIB RTL and the ISS through clock cycles, as well as perform source-level debugging on the executing processor model. For instance, it is possible to run the RTL simulator for 38025 nanoseconds while wave tracing, then break the ISS and jump into microstep mode which permits LISA source-level debugging in between SoC clock ticks. One can then via the ISS debug interface place a breakpoint either in the processor code stream, or in a particular LISA operation. In the RTL simulator, one can then run for n clock cycles until the breakpoint at either the LISA or processor instruction level is hit. At that point, the state of both

[4] There are some minor differences introduced by the socket, but PG accepts #ifdef commands that allow one to conditionally compile regions of code for simulation vs generation.

pipelines is shown graphically, and the LISA code can be single-stepped as if it were running under a C simulator.

This functionality greatly sped up the development of PDSparc. At some point however, the processor was working well enough that it was possible to add other features from the Sparc v8 specification. To ensure that adding features would not break the base functionality of the processor, the MIT LL team built a regression environment based on the ISS-socket-GRLIB interface. Because the ISS is actually compiled as a C static library (.a), it can be linked into and controlled from any C program. We therefore built a C test harness that controls the ISS, and interrogates it for pass/fail conditions. This interrogation was enabled by a set of simulation-only ISA instructions that were added to the PDSparc ISA. These instructions are understood and usable via the PD-generated LLVM [11] compiler and assembler, and permit the code stream to include assembly opcodes for instructions such as: “test_pass 0”, or “test_fail 5”, where the argument is intended to be an error code. The pass/fail status and error code are exported to the C test harness for logging and test termination detection. The system also sends a termination code through the socket to the SystemC module, which then terminates the GRLIB simulation at test completion. This regression environment can be run with a single command and execute as many tests as are needed. In terms of performance, a 100000 cycle test takes less than 20s to execute and report a result.

6. Conclusion

This paper describes the creation of a drop-in replacement called PDSparc for a LEON3 processor into a synthesizable SoC support structure. PDSparc was written in the LISA language and compiled with Synopsys Processor Designer. The implementation of a reasonably high-performance multicore capable pipelined RISC is a major feat, whose difficulty was compounded by a lack of detailed protocol information. It is safe to say that without the LISA source-level debugging capabilities enabled by Processor Designer and the ability to link to running RTL, this project would have taken far more resources that the 7 person-months in which it was achieved. The result is a processor which is still under development in terms of extended Sparc v8 features, but is now competitive in terms of performance with the original LEON3 it replaced .The major difference in performance stems from the extra cycle required to branch, which will be remedied in the future.

7. Future Work

The MIT LL development team will continue to evolve PDSparc to better comply with the Sparc v8 specification. In addition, the team intends to open-source PDSparc for general use.

8. Acknowledgements

The MIT LL team would like to thank Drew Taussig from Synopsys for his patience in educating us on Processor Designer, and for his excellent and timely help and advice in the development of PDSparc.

9. References

[1] http://www.intel.com/content/www/us/en/processors/xeon/xeon-processor-e7-family.html

[2] http://www.arm.com/files/pdf/IntroToCortex-M3.pdf

[3] http://www.gaisler.com/index.php/downloads/leongrlib

[4] http://www.gaisler.com/index.php/products/processors/leon3

[5] http://www.gaisler.com/doc/sparcv8.pdf

[6] http://www.atmel.com/Images/doc4226.pdf

[7] http://www.gaisler.com/

[8] https://www.oracle.com/servers/sparc/index.html

[9] http://www.synopsys.com/IP/ProcessorIP/asip/processor-designer/Pages/default.aspx

[10] http://accellera.org/

[11] http://llvm.org/

[12] http://wiki.osdev.org/GCC_Cross-Compiler

[13] V. Zivojnovic, S. Pees, H. Meyr, *LISA – machine description language and generic machine model for HW/SW co-design*, Proceedings of the IEEE Workshop on VLSI Signal Processing (San Francisco), Oct. 1996