

MIT Open Access Articles

A general technique for deterministic model-cycle-level debugging

The MIT Faculty has made this article openly available. **Please share** how this access benefits you. Your story matters.

Citation: Khan, Asif, Muralidaran Vijayaraghavan, and Mithal Arvind. "A General Technique for Deterministic Model-Cycle-Level Debugging." Tenth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMCODE2012) (July 2012).

As Published: <http://dx.doi.org/10.1109/MEMCOD.2012.6292307>

Publisher: Institute of Electrical and Electronics Engineers (IEEE)

Persistent URL: <http://hdl.handle.net/1721.1/90485>

Version: Author's final manuscript: final author's manuscript post peer review, without publisher's formatting or copy editing

Terms of use: Creative Commons Attribution-Noncommercial-Share Alike



A general technique for deterministic model-cycle-level debugging

Asif Khan, Muralidaran Vijayaraghavan and Arvind

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology, Cambridge, MA
{aik, vmurali, arvind}@csail.mit.edu

Abstract—Efficient use of FPGA resources requires FPGA-based performance models of complex hardware to implement one *model cycle*, i.e., one time-step of the original synchronous system, in several *implementation cycles*. Generally implementation cycles have no simple relationship with model cycles, and it is tricky to reconstruct the state of the synchronous system at the model-cycle boundaries if only implementation-cycle-level control and information is provided. A good debugging facility needs to provide: complete control over the functioning of the target design being simulated; fast and easy access to all the significant target design state for both monitoring and modification; and some means of accomplishing deterministic execution when the target design is a multicore processor running a parallel application. Moreover, these features need to be provided in a manner which does not incur substantial resource and performance penalties.

In this paper, we present a debugging technique based on the LI-BDN theory. We show how the technique facilitates deterministic model-cycle-level debugging. We used it to build the debugging infrastructure for Arete, which is an FPGA-based cycle-accurate multicore simulator. The resource and performance penalties of our debugging technique are minimal; in Arete the debugging infrastructure has area and performance overheads of 5% and 6%, respectively.

I. INTRODUCTION

As designs of digital systems continue to become more complex, designers are increasingly adopting FPGAs for both performance modeling and rapid prototyping. The FPGA fabric allows designers to exploit the inherent parallelism in these systems, and delivers a tremendous performance improvement over software. This adoption, however, comes at a price. Parts of the target system being modeled or prototyped often do not map well to the structures in the FPGA fabric, both in terms of resources and timing. An established solution to the problem is to implement the synchronous behavior of the

target system in an asynchronous manner on the FPGA using techniques such as Connectors [1], Channels [2], A-Ports [3] and LI-BDNs [4].

These techniques involve decomposing the synchronous system into many asynchronous modules that communicate with each other through FIFOs. Each module is able to tolerate variations in the latencies of the modules that it communicates with. Moreover, the notion of clock cycles in the synchronous system, called *model cycles*, is changed to the duration between the enqueueing of the output FIFOs of a module and the dequeueing of its input FIFOs. This allows each module to use as many FPGA cycles as necessary to perform the work of one model cycle in a resource and timing efficient manner. A module may use different numbers of FPGA cycles to simulate different model cycles. Moreover, various modules in a system may use a varying number of FPGA cycles to simulate the same model cycle.

Debugging is an integral part of the design effort. A comprehensive debugging infrastructure needs to provide model-cycle-level access to all the pertinent state in the system. Providing such low-level access is not straightforward in such asynchronous implementations of synchronous systems as described above. Taking a snapshot of the state in a certain FPGA cycle is possible, but the snapshot may contain values of state elements from different model cycles. Either the lagging modules have to be advanced or the hastening modules have to be rolled back in order for the state snapshot to reconcile to a particular model cycle.

At a high level, a designer should be able to issue a `stop(modelCycle n, state S)` command, which will freeze the entire system in model cycle `n` and provide the values of all the state elements included in vector `S`. A `start(state S)` command will also be needed to resume the operation of the asynchronous implementation with the state elements initialized to the

values specified in vector S .

Parallel system with inherent non-determinism, such as multicore processors running parallel applications, offer yet another challenge for debugging. A large body of work [5]–[8] exists that strives to achieve deterministic execution. To circumvent the non-determinism in the system, these solutions have to keep a log of all the non-deterministic events, the performance and resource overheads of which can be prohibitive.

The main contributions of this paper are 1) a technique for building a deterministic model-cycle-level debugging infrastructure, based on the LI-BDN modeling methodology, and 2) an application of the technique to build a comprehensive debugging infrastructure for Arete [9], which is an FPGA-based multicore processor simulator.

We show that the debugging infrastructure in Arete provides a rich set of features, while incurring small resource and performance overheads. It allows for stopping and starting any module in the processor model independently by making a novel use of the provisions of the LI-BDN methodology, and avoids complex forwarding and rollback mechanisms. It also allows us to remove the non-determinism from events such as DRAM access, network access and I/O, without keeping expensive logs. *Paper organization:* Section II presents the various debugging techniques used in FPGA-based models and prototypes. Section III discusses how the LI-BDN methodology is used to implement a synchronous system on an FPGA in an asynchronous manner. Section IV describes how deterministic model-cycle-level debugging can be achieved from LI-BDNs. Section V discusses the debugging infrastructure in Arete and presents statistics on its resource and performance overheads. Section VI summarizes our work and discusses some of the future avenues we are planning to explore.

II. SURVEY OF DEBUGGING TECHNIQUES FOR FPGA-BASED DESIGNS

In this section we discuss some of the common debugging techniques used in FPGA-based designs. Figure 1 provides a summary of the comparison between these techniques and the technique based on the LI-BDN methodology that we present in this paper.

A. System monitoring through scan chains

System monitoring solutions based on scan chains [10]–[13] are perhaps the most widely used tools for debugging FPGA-based designs. They integrate logic analyzers and other test and measurement cores

with the target design on FPGA. A remote graphical user interface communicates with these cores, and provides the designer with a logic analyzing solution.

In ChipScope [10], for example, the designer generates integrated logic analyzer (ILA) cores for all the modules in his design that he wishes to monitor. The ILA cores are customizable and include logic for detecting trigger events. They also include logic for capturing and storing data using on-chip Block RAMs. An integrated controller (ICON) core is then used to provide communication between all the ILA cores and the software running on a host PC. The communication takes place over the JTAG boundary scan port of the FPGA. The ILA cores and the ICON core can be integrated into the design at either the HDL-source-code-level or the synthesized-netlist-level.

Although these tools provide some very useful features for debugging synchronous designs, they lack control over both the FPGA and the model clocks. Moreover, the monitoring cores are synchronous and use the FPGA clock. To use these tools for debugging synchronous designs implemented in an asynchronous manner, the designer would have to develop forwarding and rollback mechanisms to be able to construct model time accurately. He would also have to develop some means of operating the monitoring cores using the model clock. These tools do not include support for deterministic execution, and the cost of comprehensively monitoring large designs may be prohibitive.

UltraSOC [14] and ARM CoreSight [15] are similar debugging tools targeted towards SoCs.

B. SCE-MI-based emulation environment

An emulation environment based on the SCE-MI standard, such as Bluespec emVM [16], comprises of an FPGA configured with a hardware design and a host PC running the emulation console. The FPGA and the host PC are connected by a physical link such as the PCIe, ethernet, RS-232, *etc.* The emulation console communicates with the components of the hardware design through implementation-independent transactors. These transactors allow the designer to start and stop the FPGA clock. They also allow control over the hardware design in the form of reset and various testing and debugging tasks. Probing functionality such as waveform viewing is also often provided.

Although the provided set of monitoring and debugging features is not as extensive as that in ChipScope, SCE-MI based emulation environments provide the ability to freeze the entire synchronous design in any FPGA

Debugging support	Clock control type	Deterministic execution support	Resource-performance overhead
Scan chains	None	No	Substantial
SCE-MI-based	FPGA	No	Substantial
ISA-based	None	No	Moderate
Asynchronous models	Model ^a	Possible	Substantial
LI-BDN-based	Model	Yes	Minimal

^a requires a forwarding or rollback mechanism

Fig. 1. Summary of the comparison between the LI-BDN-based debugging technique and other common debugging techniques used in FPGA-based designs

cycle. However, for debugging synchronous designs implemented in an asynchronous manner, the designer would face the same set of challenges as he would with ChipScope. A lack of support for deterministic execution and substantial resource and performance overheads also limit the appeal of such tools.

C. ISA-based debugging

When using FPGAs for modeling processors, designers also have the option of implementing the debugging facilities prescribed in the ISA. These facilities enable debugging functions, such as reset, instruction and data breakpoints, and single-stepping of programs. They generally consist of debug control and status registers, address and data value comparison registers, and a debug interrupt. Whenever a debug event takes place, it raises a debug exception (if enabled by setting the appropriate bits in the control register). A debug interrupt handler routine is then invoked which performs the appropriate debug operation.

ISAs also include instructions, such as Debugger Notify Halt (DNH) in the Power ISA [17], which cause the processor to stop fetching and executing instructions, and allow the processor to be managed by an external debugging facility. Such a facility is allowed to access processor resources and control its execution.

Although ISA prescribed debugging facilities provide fine-grained control over the processor’s resources, they may be quite difficult to implement. For instance, implementing a precise debug interrupt in an out-of-order processor can be quite cumbersome.

D. Debugging in various asynchronous FPGA-based models

Various FPGA-based simulators that rely on asynchronous modeling of synchronous designs, such as ProtoFlex [18], UT-FAST [1], RAMP Gold [19] and HAsim [20], implement debugging facilities in an ad-hoc manner. They need to implement forwarding and

rollback mechanisms in order to achieve model-cycle-level debugging. In ProtoFlex, printf-like statements are added to the generated RTL to provide monitoring during software simulation of the RTL. The Connectors in UT-FAST include support for triggering, logging of traces and user-specified aggregation. RAMP Gold embeds a microcode injector into the functional processor pipeline for debugging and simulation control. HAsim provides an elaborate mechanism for model-cycle-level control which involves waiting for all the ports to become empty.

III. SYNCHRONOUS CIRCUITS SIMULATED ASYNCHRONOUSLY

A synchronous system has strict timing contracts among its various modules. This makes it hard to implement a synchronous system *as is* on various hardware or software target platforms because each platform has its own timing constraints which may not map very well to the timing constraints of the synchronous specification. Designers refine the modules of the synchronous system to make them tolerant of the latencies of other modules. Latency Insensitive Bounded Dataflow Networks (LI-BDN) [4] is one such technique to refine a synchronous specification into a latency-tolerant asynchronous specification, albeit with a clock.

We present the synchronous specification of a decode-execute-commit module in Figure 2 which describes the three stages of an in-order pipeline. The stages are connected through `PipelineReg`, a one-element FIFO which can be enqueued and dequeued in the same cycle when it is full. The module takes in a fetched instruction whose presence is indicated by an associated valid bit, and acknowledges its receipt. The module also provides a new instruction address when a branch gets resolved. The execute stage provides the two read register indices to the register file module. The commit stage sends a register update to the register file which includes a valid bit, a register index and a new register value.

```

module decExecCmt {
  Input fInst, valReg1, valReg2;
  Output fInstAck, branchPC, rdReg1, rdReg2, upd;
  PipelineReg dr, er;

  action {
    if( fInst.valid && !dr.full ) {
      dr.enq( decode( fInst ) );
    }

    if( !dr.empty && !er.full ) {
      er.enq( execute( dr.first, valReg1, valReg2 ) );
      branchPC = branch( dr.first, valReg1, valReg2 );
      dr.deq;
    }

    if( !er.empty ) {
      er.deq;
    }

    fInstAck = fInst.valid && !dr.full;
    rdReg1 = dr.first.reg1;
    rdReg2 = dr.first.reg2;
    updReq = !er.empty ? Valid er.first : Invalid;
  }
}

```

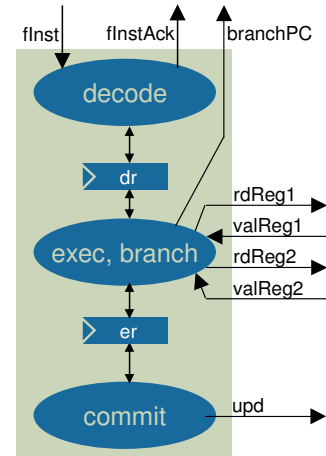


Fig. 2. Synchronous specification of a decode-execute-commit module

```

module regFile {
  Input rdReg1, rdReg2, upd;
  Output valReg1, valReg2;
  Reg entries[ sizeRF ] rf ( initial 0 );

  action {
    if( upd.valid ) {
      rf[ upd.index ] <= upd.val;
    }

    valReg1 = upd.valid && rdReg1 == upd.index ? upd.val : rf[ rdReg1 ];
    valReg2 = upd.valid && rdReg2 == upd.index ? upd.val : rf[ rdReg2 ];
  }
}

```

Fig. 3. Synchronous specification of a 2-read, 1-write register file module

Figure 3 presents the synchronous specification of a register file module. The module can take in three requests simultaneously: reading of two register values, and update of one register value. The presence of the update request is indicated by an associated valid bit. If all the requests are present simultaneously, and either of the registers being read is also being updated, the updated value is bypassed as the read response.

The two modules can be composed in a straightforward manner through the common port names.

A. Overview of the LI-BDN technique

We give a brief overview of the LI-BDN technique using the register file example. We start with the cycle-level specification described above and depicted in Figure 4(a). Such a specification does not map well to the

FPGA fabric in terms of both resources and timing. We transform the specification into an LI-BDN so that the register array which has three ports and combinational reads can be simulated with a Block RAM which has two ports and one-cycle-latency reads. We start by attaching FIFOs to all the ports and done flags to all the output ports, as shown in Figure 4(b). Note that these FIFOs are in addition to the FIFOs which may be part of the synchronous specification. Now as Figure 4(c) depicts the `valReg1` output depends on the `rdReg1` and the `upd` inputs, which are both available. So we enqueue `valReg1` and set its done flag. We handle the `valReg2` output in the same manner. Finally, after all the outputs are enqueued and all the inputs are available, we update the Block RAM, dequeue all the inputs and reset all the done flags, as shown in Figure 4(d). The conversion from

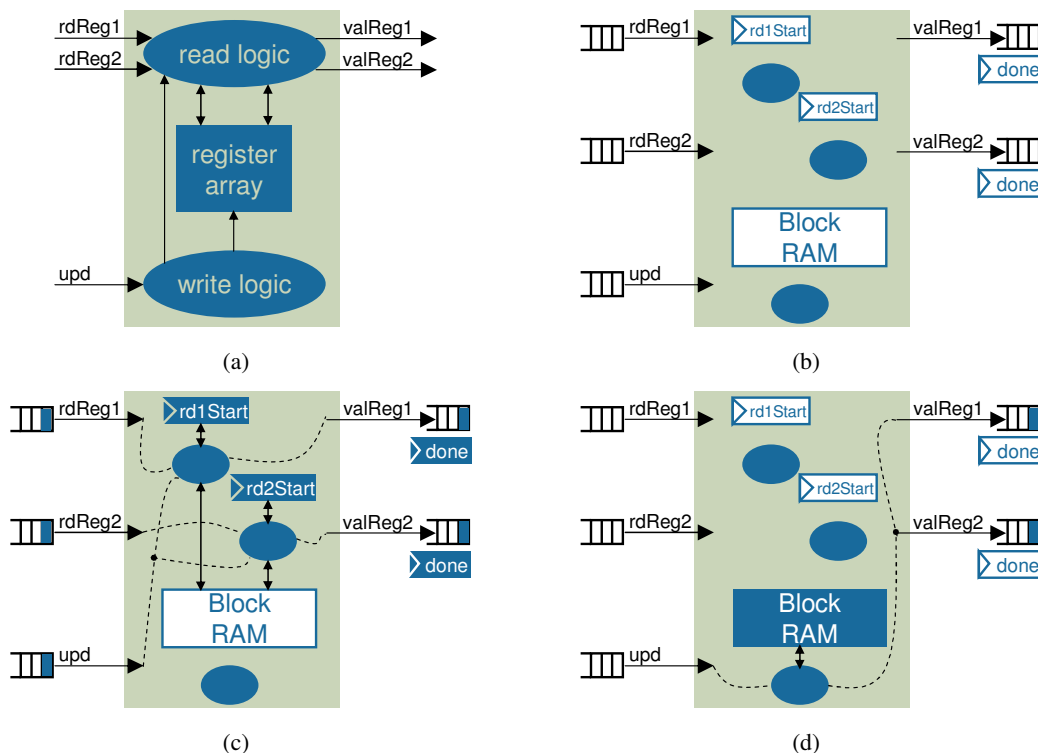


Fig. 4. Transforming a cycle-level specification into an LI-BDN

a specification into an LI-BDN is what we call the *LI-BDN transformation* of a module [4], [21]. The control logic for the LI-BDN transformation of the register file module is provided in Figure 5 (the non-highlighted portion of the code). The two requirements, that an output waits only for the inputs that it depends on, called the no-extraneous dependencies (NED) requirement, and that all the input FIFOs are dequeued when all the inputs are available and all the outputs have been produced, called the self-cleaning (SC) requirement, together guarantee the absence of deadlocks from the LI-BDN transformation.

The decode-execute-commit module can be converted into an LI-BDN using the same transformation algorithm. It will contain a separate rule for producing each output, followed by a finish rule to update the state, dequeue all the inputs and reset all the done flags.

The time duration between the enqueueing of the output FIFOs and the dequeueing of the input FIFOs comprises one model cycle for the transformed module. During one model cycle, the transformed module can use any number of implementation cycles to produce the outputs or to update the state. In this manner, the model cycle is decoupled from the implementation cycle which enables an efficient implementation of the model on the desired

platform while maintaining model-cycle-level accuracy.

IV. DEBUGGING USING THE LI-BDN TECHNIQUE

The major requirement for debugging a large and complex model is to have the ability to freeze it in a particular model cycle so that a precise snapshot of all the state can be obtained. This requirement gets quite tricky when the synchronous specification of a target design is decoupled from its platform-specific implementation. A designer typically requires the values of the state during a particular model cycle as opposed to the implementation cycle. Even if the entire implementation is frozen during a particular implementation cycle, various asynchronous modules in the implementation have to either rollback or advance so that the entire design converges to a particular model cycle. Such an ability is similar to taking a snapshot of the architectural state of an out-of-order processor for precise exceptions.

We present a novel technique, based on the LI-BDN theory, for freezing an asynchronous implementation of a synchronous design during a particular model cycle. The technique does not involve forwarding or rollback of modules. As shown in Figure 6, we introduce a new `proceed` input to the register file module from Figure 4(b). We also add debugging logic, and `debugReq` and `debugResp` FIFOs to the module. An external

```

libdn regFile {
  LiBdnIn rdReg1, rdReg2, upd, proceed;
  LiBdnOut valReg1, valReg2;
  FifoIn debugReq;
  FifoOut debugResp;
  BlockRAM entries[ sizeRF ] rf ( initial 0 );
  Reg rd1Start, rd2Start ( initial False );

  rule rd1 {
    if( !valReg1.done && !valReg1.full && !rdReg1.empty && !upd.empty && !rd1Start ) {
      rf.req1( Read, rdReg1.first, DontCare );
      rd1Start <= True;
    }
    if( rd1Start ) {
      valReg1.enq( upd.first.valid && rdReg1.first == upd.first.index ? upd.first.val :
                                                           rf.resp1 );

      valReg1.done <= True;
      rd1Start <= False;
    }
  }

  rule rd2 {
    if( !valReg2.done && !valReg2.full && !rdReg2.empty && !upd.empty && !rd2Start ) {
      rf.req2( Read, rdReg2.first, DontCare );
      rd2Start <= True;
    }
    if( rd2Start ) {
      valReg2.enq( upd.first.valid && rdReg2.first == upd.first.index ? upd.first.val :
                                                           rf.resp2 );

      valReg2.done <= True;
      rd2Start <= False;
    }
  }

  rule finish {
    if( valReg1.done && valReg2.done && !proceed.empty ) {
      if( proceed.first == Normal ) {
        if( upd.first.valid ) {
          rf.req1( Write, upd.first.index, upd.first.val );
        }
        rdReg1.deq; rdReg2.deq; upd.deq; proceed.deq;
        valReg1.done <= False; valReg2.done <= False;
      }
      else if( !debugReq.empty ) {
        if( debugReq.first.type == Read && !debugResp.full ) {
          if( !rd1Start ) {
            rf.req1( Read, debugReq.first.index, DontCare );
            rd1Start <= True;
          }
          else {
            debugResp.enq( rf.resp1 );
            rd1Start <= False;
          }
        }
        else if( debugReq.first.type == Write ) {
          rf.req1( Write, debugReq.first.index, debugReq.first.val );
        }
        else if( debugReq.first.type == Finish ) {
          if( upd.first.valid ) {
            rf.req1( Write, upd.first.index, upd.first.val );
          }
          rdReg1.deq; rdReg2.deq; upd.deq; proceed.deq;
          valReg1.done <= False; valReg2.done <= False;
        }
        debugReq.deq;
      }
    }
  }
}

```

Fig. 5. FPGA-optimized LI-BDN of the register file module with support for debugging

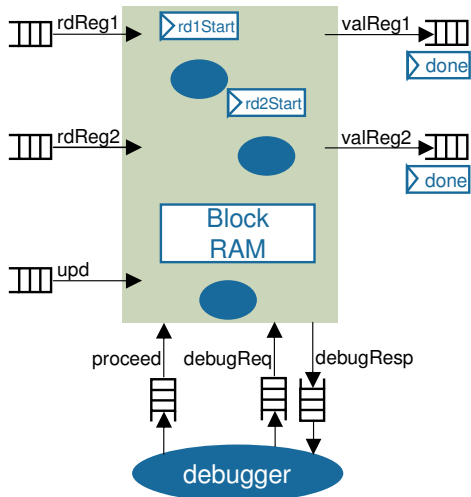


Fig. 6. LI-BDN of the register file module with support for model-cycle-level debugging

debugger can freeze the module in model cycle n by enqueueing a *Normal* token $n-1$ times into the `proceed` input FIFO and enqueueing a *Debug* token the n^{th} time. Once the module receives a *Debug* token, it enters the debug mode and waits for debug commands that are sent through the `debugReq` FIFO. The debug commands can either read or update the Block RAM. Responses for Block RAM read requests become available one cycle after the request is made, and are sent back to the external debugger through the `debugResp` FIFO. When the external debugger sends a *Finish* command, the module leaves the debug mode, updates the Block RAM, dequeues all the LI-BDN input FIFOs, resets all the done flags and proceeds onto the next model cycle. The highlighted code in Figure 5 shows the debugging logic added to the LI-BDN of the register file module. Only the parts that deal with `rf` are specific to the register file module, the rest can be added to any LI-BDN module for debugging.

A. Correctness of the LI-BDN-based debugging technique

An LI-BDN module obtained through the transformation discussed in Section III simulates a model cycle by first producing all the outputs once (in an order determined by the availability of the inputs), followed by the firing of the `finish` rule. It can take multiple implementation cycles to produce an output or to fire the `finish` rule. The output rules can fire concurrently, but cannot fire in parallel with the `finish` rule. The `finish` rule acts as a barrier and prevents output rules from refiring before the model cycle is completed. The

LI-BDN transformation of any synchronous specification can be easily automated and we assume that it is correct.

The additional input, `proceed`, introduced for debugging, only affects the firing of the `finish` rule. The `finish` rule waits for a `proceed` token to arrive before firing, which can result in a prolonged model cycle. The latency of an LI-BDN module, *i.e.*, the number of implementation cycles consumed to simulate a model cycle, can be varied without affecting the correctness of the module. Moreover, the external debugger has to enqueue a `proceed` token, either *Normal* or *Debug*, for every model cycle.

The additional FIFOs, `debugReq` and `debugResp`, are *out-of-band* communication links that remain outside the scope of the LI-BDN. The debug commands delivered by the `debugReq` FIFO are only serviced when the module is in the debug mode, which can only be activated after all the outputs are enqueued, but before the model state is updated. The LI-BDN resumes normal operation upon leaving the debug mode.

Debugging logic, as in the case of the register file example above, can be introduced into an LI-BDN module such that it remains completely disjoint from the LI-BDN control logic in the module. This is evident from the highlighted code in Figure 5. Moreover, the debugging logic does not introduce any new behaviors into the target design being modeled. It only allows for reading and writing of model state when the LI-BDN module is in the debug mode.

It is also evident that the addition of the debugging logic does not violate the NED and SC requirements of the LI-BDN transformation, preserving the deadlock-freedom guarantee.

B. Deterministic execution

There are many sources of non-determinism in complex, parallel systems such as the randomness of the DRAM access latency. This complicates the debugging further by prohibiting deterministic replays. The use of LI-BDNs in modeling provides an opportunity to suppress the non-determinism. In the case of DRAMs, the access latency can be fixed to any desired value. This is possible because the LI-BDN can utilize different numbers of FPGA cycles to simulate different model cycles, and accommodate the randomness appropriately. The enqueueing of the output FIFOs (or the dequeuing of the input FIFOs) can happen when the non-deterministic event has taken place.

As an example, we will show how a DRAM module with a non-deterministic read latency is converted into


```

module memory {
  Input req;
  Output resp;
  DRAM dram;

  action {
    dram.req = req;
    resp = dram.resp;
  }
}

```

Fig. 7. Synchronous specification of a DRAM module with non-deterministic read latency

```

libdn memory {
  LiBdnIn req;
  LiBdnOut resp;
  DRAM dram;
  Reg tempResp ( initial Invalid );

  rule tempRule {
    if( !tempResp.valid ) {
      tempResp <= dram.resp;
    }
  }

  rule respRule {
    if( !resp.done && !resp.full ) {
      resp.enq( tempResp );
      resp.done <= True;
      if( tempResp.valid ) {
        tempResp <= Invalid;
      }
    }
  }

  rule finish {
    if( resp.done && !req.empty ) {
      dram.req = req.first;
      req.deq;
      resp.done <= False;
    }
  }
}

```

Fig. 8. LI-BDN of a DRAM module with non-deterministic read latency

an LI-BDN module with a deterministic read latency, *viz.*, a combinational read. Figure 7 presents the synchronous specification of a memory module that uses a DRAM. Both `req` and `resp` have associated valid bits. The system which uses this module makes `req` valid for only one cycle, consumes `resp` in the cycle in which `resp` is valid, and does not make another valid `req` until it receives a valid `resp`. `dram` has a non-deterministic response time, and produces a response, `dram.resp`, which is valid for only one cycle.

Figure 8 presents the LI-BDN of the non-deterministic memory module shown in Figure 7. It increments model time irrespective of the validity of `dram.resp`. `tempResp` ensures that a valid `dram.resp` is not

```

libdn memory {
  LiBdnIn req;
  LiBdnOut resp;
  DRAM dram;
  Reg start ( initial False );

  rule respRule {
    if( !resp.done && !resp.full &&
        !req.empty && !start ) {
      if( req.first.valid ) {
        dram.req = req.first;
        start <= True;
      }
      else {
        resp.enq( Invalid );
        resp.done <= True;
      }
    }
    if( dram.resp.valid ) {
      resp.enq( dram.resp );
      resp.done <= True;
      start <= False;
    }
  }

  rule finish {
    if( resp.done ) {
      req.deq;
      resp.done <= False;
    }
  }
}

```

Fig. 9. LI-BDN of a DRAM module with combinational reads

dropped.

Figure 9 presents the LI-BDN of a combinational-read memory module. If `req` is valid in a model cycle, the LI-BDN sends it to `dram`, and waits, without incrementing the model cycle, until `dram.resp` becomes valid. When `dram.resp` becomes valid, the LI-BDN enqueues it into `resp` and completes the model cycle by dequeuing `req`. Even though `dram` can take a non-deterministic number of FPGA cycles to produce a valid response, model cycles are incremented deterministically leading to deterministic execution.

The two LI-BDNs presented in Figures 8 and 9 model different memory modules, but both fulfill the NED and SC requirements, and are deadlock-free.

V. LI-BDN-BASED DEBUGGING INFRASTRUCTURE FOR A MULTICORE PROCESSOR MODEL: A CASE STUDY

Using the LI-BDN-based debugging methodology described in Section IV, we built a comprehensive debugging facility for Arete [9], which is an FPGA-based cycle-accurate multicore simulator. Arete may be implemented as a distributed multicore simulator on a multi-FPGA platform, which requires the debugging infrastructure to be implemented in a distributed manner. We

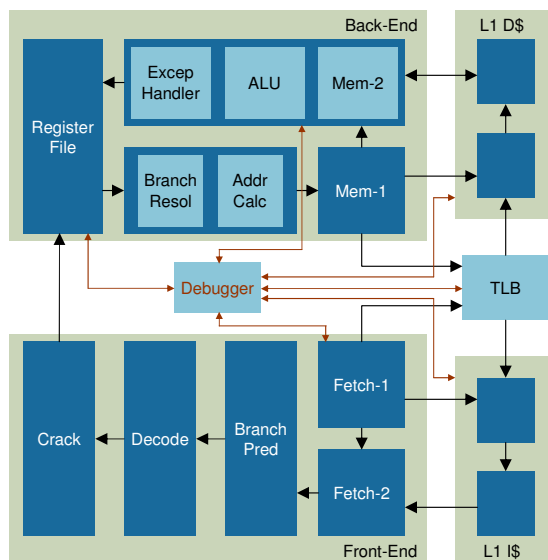


Fig. 10. Arete core with model-cycle-level debugging facilities

make use of the tiled microarchitecture of the processor to partition the model among various FPGAs in such a way that only one configuration file can be used for all the FPGAs. This enables a simple replication of the debugging facilities, but complicates the design of the controller.

The distributed debugging facilities in FPGA are controlled by a software running on a MicroBlaze soft core. The MicroBlaze core communicates with the model through the PLB. The software presents a GDB-like interface to the user. Its features include model initialization, break points, single-stepping, access to processor state such as program counter, general purpose registers (GPRs), special purpose registers (SPRs), TLB array, and data and tag arrays in caches, and access to performance counters which include model cycles, FPGA cycles, instructions, stalls due to data and control hazards, and cache hits and misses.

Figure 10 shows the debugging facilities incorporated into a core in Arete. These FPGA-based facilities include logic and state for

- model initialization, which may be done in a distributed manner on a multi-FPGA platform, and requires assigning a unique identifier to each of the identical model partitions,
- distribution and accumulation of debugging and performance information from various tiles, cores and modules,
- instruction address, data address and model cycle comparisons.

	Without Debugging	With Debugging	Change
LUTs	61155	64154	+5%
Flip flops	49359	51331	+4%
Block RAMs	111	111	0%
DSP slices	24	24	0%
FPGA clock frequency (MHz)	125	117	-6%

Fig. 11. Resource and performance penalties of the debugging infrastructure in Arete. Model parameters: 1 tile, 2 in-order 10-stage cores, 64 KB 4-way associative L1, 512 KB 4-way associative L2

The use of the LI-BDN modeling methodology in building Arete also enables us to provide deterministic execution of parallel applications on the multicore processor model. We make the observation that the three sources of non-determinism in Arete are memory, on-chip network, and external inputs. We transform the DRAM along with the memory controller, and the on-chip network (implemented in a distributed manner on multiple FPGAs) into LI-BDN modules. This allows us to fix their latencies in the manner described in Section IV. We deal with external inputs by freezing the model whenever the program expects such an input. This ensures that the external input is always received in the same model cycle. Both of these techniques have very low resource and performance overheads, and help to avoid keeping expensive logs of non-deterministic events.

Figure 11 shows the minimal overhead of including the deterministic model-cycle-level debugging facility in Arete. It causes an increase of 5% in resource utilization, and reduces FPGA clock frequency by 6%. As described in the register file example in Section IV, the debugging facility requires limited additional state and logic resources, and the overhead is expected to scale linearly with model size. Moreover, the debugging facility has no impact on the average number of FPGA cycles required to simulate a model cycle, which remains 9.

VI. CONCLUSION

We have presented a debugging technique based on the LI-BDN modeling methodology. The technique facilitates deterministic model-cycle-level debugging, while avoiding both forwarding or rollback mechanisms for model-cycle-level control, and logging of non-deterministic events for deterministic replay. We used the

technique to build the debugging infrastructure for Arete, which is an FPGA-based cycle-accurate multicore simulator. The debugging infrastructure provides a rich set of features, while incurring small resource and performance overheads.

Moving forward, we are planning to develop a tool that will automatically transform a synchronous specification into an LI-BDN which will include the debugging facility.

ACKNOWLEDGEMENTS

We would like to thank Kattamuri Ekanadham and Jessica Tseng for their help and guidance. The research work presented in this paper was funded in large part by IBM. We are thankful to Quanta Computer for supporting this work in the later stages. We would also like to thank Xilinx for donating FPGAs, FPGA boards and development tools. Many thanks to all members of the CSG-Arvind group whose valuable input helped us shape the narrative in this paper.

REFERENCES

- [1] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. Reinhart, D. E. Johnson, J. Keefe, and H. Angepat, "FPGA-Accelerated Simulation Technologies (FAST): Fast, Full-System, Cycle-Accurate Simulators," in *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 249–261.
- [2] G. Gibeling, A. Schultz, and K. Asanovic, "RAMP Architecture and Description Language," in *2nd Workshop on Architecture Research using FPGA Platforms*, February 2006.
- [3] M. Pellauer, M. Vijayaraghavan, M. Adler, Arvind, and J. Emer, "A-Port Networks: Preserving the Timed Behavior of Synchronous Systems for Modeling on FPGAs," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 2, no. 3, pp. 1–26, 2009.
- [4] M. Vijayaraghavan and Arvind, "Bounded Dataflow Networks and Latency-Insensitive circuits," in *MEMOCODE'09: Proceedings of the 7th IEEE/ACM International Conference on Formal Methods and Models for Codesign*. Piscataway, NJ, USA: IEEE Press, 2009, pp. 171–180.
- [5] D. F. Bacon and S. C. Goldstein, "Hardware-Assisted Replay of Multiprocessor Programs," *SIGPLAN Not.*, vol. 26, no. 12, pp. 194–206, Dec. 1991.
- [6] J.-D. Choi and H. Srinivasan, "Deterministic Replay of Java Multithreaded Applications," in *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, ser. SPDT '98. New York, NY, USA: ACM, 1998, pp. 48–59.
- [7] M. Xu, R. Bodik, and M. Hill, "A "Flight Data Recorder" for Enabling Full-System Multiprocessor Deterministic Replay," in *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, june 2003, pp. 122 – 133.
- [8] G. Altekar and I. Stoica, "ODR: Output-Deterministic Replay for Multicore Debugging," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 193–206.
- [9] A. Khan, M. Vijayaraghavan, S. Boyd-Wickizer, and Arvind, "Fast and Cycle-Accurate Modeling of a Multicore Processor," in *ISPASS '12: Proceedings of the International Symposium on Performance Analysis of Systems and software*, April 2012.
- [10] *ChipScope Pro – Software and Cores User Guide Version 13.4*, Xilinx, January 2012.
- [11] *Identify – Simulator-like Visibility into Hardware Debug*, Synopsys, 2011.
- [12] P. Graham, B. Nelson, and B. Hutchings, "Instrumenting Bitstreams for Debugging FPGA Circuits," in *Field-Programmable Custom Computing Machines, 2001. FCCM '01. The 9th Annual IEEE Symposium on*, April 2001, pp. 41 –50.
- [13] K. Camera, H. K.-H. So, and R. W. Brodersen, "An integrated debugging environment for reprogrammable hardware systems," in *Proceedings of the sixth international symposium on Automated analysis-driven debugging*, ser. AADEBUG'05. New York, NY, USA: ACM, 2005, pp. 111–116.
- [14] <http://www.ultrasoc.com/>.
- [15] <http://www.arm.com/products/system-ip/coresight/index.php>.
- [16] *emVM User Manual*, Bluespec, January 2012.
- [17] *Power ISA Version 2.05*, IBM, October 2007.
- [18] E. S. Chung, M. K. Papamichael, E. Nurvitadhi, J. C. Hoe, K. Mai, and B. Falsafi, "ProtoFlex: Towards Scalable, Full-System Multiprocessor Simulations Using FPGAs," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 2, no. 2, pp. 1–32, 2009.
- [19] Z. Tan, A. Waterman, R. Avizienis, Y. Lee, H. Cook, D. Patterson, and K. Asanovic, "RAMP Gold: An FPGA-based Architecture Simulator for Multiprocessors," in *DAC '10: Proceedings of the 47th Annual Design Automation Conference*, 2010, pp. 463–468.
- [20] M. Pellauer, M. Vijayaraghavan, M. Adler, Arvind, and J. Emer, "Quick Performance Models Quickly: Closely-Coupled Partitioned Simulation on FPGAs," in *ISPASS '08: Proceedings of the International Symposium on Performance Analysis of Systems and software*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 1–10.
- [21] T. Harris, Z. Ruan, and D. Penry, "Techniques for LI-BDN synthesis for hybrid microarchitectural simulation," in *Computer Design (ICCD), 2011 IEEE 29th International Conference on*, oct. 2011, pp. 253 –260.