The BERIpad tablet: open-source construction, CPU, OS and applications

A Theodore Markettos*, Jonathan Woodruff*, Robert N M Watson*,
Bjoern A Zeeb*, Brooks Davis†, Simon W Moore*
*Computer Laboratory, University of Cambridge, UK
{firstname.lastname}@cl.cam.ac.uk
†SRI International, Menlo Park, CA, USA
brooks.davis@sri.com

***Abstract*—We present a full desktop computer system on a portable FPGA tablet. We have designed BERI, a 64-bit MIPS R4000-style soft processor in Bluespec SystemVerilog. The processor is implemented in a system-on-chip on an Altera Stratix IV FPGA on a Terasic DE4 FPGA board that provides a full motherboard of peripherals. We run FreeBSD providing a multi-user UNIX-based OS with access to a full range of general-purpose applications. We have a thorough test suite that verifies the processor in continuous integration. We have open-sourced the complete stack at beri-cpu.org including processor, system-on-chip, physical design and OS components. We relate some of our experiences of applying techniques from successful open-source software projects on the design of open-source hardware.**

***Index Terms*—Open-source, hardware, tablet, FPGA, Bluespec, BERI, MIPS, FreeBSD, Terasic, Altera**

I. Introduction

Open-source software is behind much of modern computing. The open-source model, including software engineering practices such as modularity, code reuse, version control, test suites with test-driven development and continuous integration, and the building of developer communities, are now well understood.

Open-source *hardware*, however, has failed to gain much traction. Part of this is inherent: hardware is a physical object and physical objects cannot be shared in the way software can. But the design of hardware involves intellectual property (IP) in just the same way that software does.

Meanwhile, system-on-chip design has polarised. On the one hand, semiconductor vendors produce systems with ever-increasing complexity with ever-more-dense fabrication processes. The cost and complexity of such ASICs continues to rise, such that custom design becomes impractical other than when producing huge volumes.

On the other hand, Field-Programmable Gate Arrays (FPGAs) offer a technology accessible at lower volumes. This provides an opportunity for open-source hardware. The FPGA itself is produced in the huge volumes that is necessary to make semiconductor production economic, and FPGA boards can be bought off-the-shelf at reasonable cost. The FPGA is programmable with IP which can be written and distributed in a similar development process to modern software. One example of a popular open-source FPGA project is the NetFPGA networking platform [1].

Fig. 1. BERIpad with application launcher

FPGAs are typically programmed in hardware definition languages (HDLs) such as VHDL or Verilog. System-on-chip (SoC) design typically involves integration of components from many different sources — modularity enables reuse. Traditional HDLs take a low-level approach, which means that an intricate knowledge of the interface and timing properties of a component are necessary to use it. As ASIC designers have found to their cost, incorrect assumptions about interfaces are a common cause of errors [2].

Such a low-level development process has held back open-source hardware on FPGAs: while components are available in repositories such as OpenCores [3], it is often quicker to re-implement them from scratch than to fully understand the nuances of the component's interface.

In this paper we describe a desktop-class computer system we have designed and open-sourced, including processor and peripherals. We have used an array of techniques from software engineering to aid our development process and build more solid foundations than many previous hardware projects.

We have used Bluespec SystemVerilog, a higher level language to take care of many of the interfacing difficulties.

Bluespec enables us to build a complex SoC in a much shorter time than a Verilog-based design. In addition we can run a high-speed cycle-accurate simulation of the design which can boot the open-source FreeBSD OS in simulation. We then apply regression testing and continuous integration to our processor so that we can be confident that it works the first time when synthesised on the FPGA.

Using Bluespec to aid productivity, we have designed BERI, a 64-bit MIPS R4000-style soft processor that runs at 100 MHz on the FPGA, as well as approximately 34 kHz in cycle-accurate simulation. The processor has a full test-suite that compares the processor against existing MIPS emulators and is run against every source code commit.

We also have full series of peripherals, providing modern interfaces such as Gigabit Ethernet, SD card, HDMI, USB 2.0, and up to 8GB of DDR2 memory, using either cores we developed in Bluespec or were provided by Altera. Bluespec makes it easier to wrap the details of bus interfacing, such as AXI [4] or Altera's Avalon, so that a peripheral device need only present a simple, easy to write interface.

We implement this on a Terasic DE4 FPGA board which contains an Altera Stratix IV 230GX FPGA. The FPGA board is built into a battery-powered tablet with touchscreen LCD which is easily portable and able to demonstrate our work at conferences and meetings.

Having designed this complex system-on-chip, we have ported the FreeBSD operating system and written drivers for all the peripherals (with the exception of USB, which is under way). A desktop/server-class OS with full driver support greatly increases the flexibility of the platform as complex software protocol stacks such as USB and IPv6 come for free. As a result we can give developers access to the board via SSH so that they can experiment with our open-source hardware without owning a physical FPGA board. Support for our platform has been upstreamed and will release in FreeBSD 10.0.

We have open-sourced all the levels of the design from processor through system-on-chip to OS. Since the design is at a higher level than traditional VHDL or Verilog, both hardware and software are easier to modify which means it is easy to customise to your application. In addition, components such as the BERI CPU are designed to be easily re-usable without modification in projects where smaller soft-cores such as an Altera NIOS II are insufficient due to addressing or data width constraints.

We begin with Section II which introduces our hardware platform. In Section III we describe the differences between Bluespec SystemVerilog and traditional HDLs such as Verilog. The architecture of the BERI processor is described in Section IV. The processor simulation environment is outlined in Section V and testing in Section VI.

Section VII then describes the rest of the system-on-chip. Section VIII explains bringup of FreeBSD and the system software. Finally we draw some conclusions.

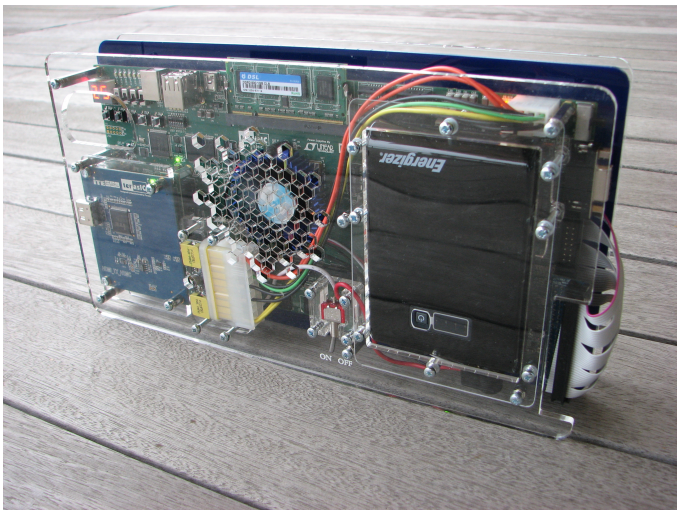


Fig. 2. Tablet construction

II. The BERIpad tablet platform

Our development platform is based on the Terasic DE4 board. The DE4 contains a large Altera Stratix IV 230GX FPGA with roughly 230K LEs and 17 Mbit of BRAM. We chose the board due to the large FPGA and the number of peripherals. The board supports up to 8 GB of DDR2 memory in two SODIMMs, and a number of high-speed transceivers which are wired to PCI Express, SATA and high speed module connectors. It also provides 64 MB of flash memory, a 2 MB static RAM, a USB controller, four gigabit Ethernet ports and about 80 pins of GPIO, plus the ability to take two daughter cards to further expand the platform. In short, most of the I/O of a modern PC.

We use this board in a number of configurations. We built the Bluehive[5], an array of 16 DE4s using the PCIe and SATA ports for high-speed board interconnect. We built a rackmount server, an array of DE4s to provide BERI with FreeBSD as a cloud service to external developers through SSH login. We also built the BERIpad, a portable system which we use for giving demonstrations at conferences and to visitors.

The BERIpad consists of a DE4 board in a custom laser-cut acrylic case. To the rear is mounted a Terasic MTL multitouch screen which provides the primary user interface to the tablet. On the front is mounted a Terasic HDMI output board. The SoC design drives the same image to both displays, which enables giving of presentations directly from the tablet. The unit is powered by an Energizer 40Wh laptop secondary battery via a PicoPSU voltage converter. The power supply arrangement uses off-the-shelf components to avoid issues with airport security due to safety issues with custom-made lithium battery packs. If we were constructing a larger volume of tablets, we would investigate a custom battery geometry and custom power supply to make the tablet thinner, have a longer battery life and involve the processor in power management.

The tablet is about 29 cm $\times$ 16 cm $\times$ 6 cm and weighs 1.6 kg. Battery life is about $1-1\frac{1}{2}$ hours, though we currently

do no power management in our design. One difficulty we discovered is that the battery uses the presence of an output cable being plugged in to enable the internal DC-DC converter. Despite a physical switch disconnecting the FPGA, if the battery remains plugged into the board it runs down over the course of a day or two. We hope to fix this with an alternative battery, but in the meantime simply unplugging the internal battery cable suffices.

The Terasic multitouch LCD has a resolution of 800×480, which is matched with an HDMI output of 640×480 for maximum compatibility with conference projectors. We are working on a more flexible video system which is able to run LCD and HDMI at different resolutions, and X11 support.

III. Bluespec: Language for Extensible Design

Bluespec SystemVerilog is a hardware description language developed out of work at Massachusetts Institute of Technology [6]. Bluespec features a rich and strict type system which includes interfaces for correct flow control. A Bluespec design is structured as a "term rewriting system", a collection of state that is updated by a set of rules. While this arrangement is a common design pattern in other hardware description languages, limiting the designer to this structure greatly improves automated analysis of the system allowing more detailed and relevant compiler warnings and much faster simulation. Bluespec compiles to Verilog, for synthesis into hardware, and C, for fast cycle-accurate simulation in software.

In the sections below we will step through a FIFO example which demonstrates the value of Bluespec syntax and interfaces to illustrate some basic benefits of the language for open-source, easily-extensible hardware.

The left column of Figure 3 is an implementation of a simple single element FIFO in Bluespec, and the right side is a very similar FIFO in Verilog. The number of lines in the Bluespec version is reduced from 36 to 13, not counting white space, but more importantly the function of the module is much more clear.

In the Bluespec case it is apparent that the *enq* Action can only occur if the *full* register is not True, and that the action itself assigns the *Packet_t* data value that is passed in to the internal *fifomem* register and *full* is set to True. In contrast, in the Verilog case, it is not clear that there is any particular action to enqueue a packet, but with careful inspection you will notice that when the *writep* signal is asserted, *fullp* is not set, and *rstp* is not high, *fifomem* will be set to the value of *din* and *fullp* will be set to 1.

Having inspected this code, you might mentally associate *din* with *writep* as an input interface but use of *din* and *writep* are not limited to this interface and may be used arbitrarily in the module. In contrast, Bluespec formalizes the idea of a transaction, making the *writep* implicit and the *packet* input associated with this interface local to this interface, preventing accidental misuse.

Furthermore the interface to the Bluespec module is much more clear than that of the Verilog module. The Bluespec module exposes three operations, *enq*, *deq*, and *first*. The Verilog module exposes seven signals which may be set or read independently, though only certain combinations and sequences of actions will result in a coherent result. Therefore use of this module requires a complex understanding of the meaning of each signal, greatly increasing the barrier to entry for working with this module.

This complexity problem grows exponentially worse with large Verilog systems with interoperating modules where each must be understood in order to modify the code in significant ways.

Figure 4 gives a similar test bench in each language which exercises the respective modules. Again it should be noted that not only is the code much shorter in the Bluespec case, but it is easier to understand. Crucially, the flow control signals that might incite incorrect operation are not exposed in the Bluespec case and so correct operation is default and implicit. In contrast, the Verilog case exposes all flow control signals rendering the system fragile as a result.

Both the Bluespec and the Verilog implementations of the FIFO use a *packet_t* type, however in the Verilog version this type will be a simple pseudonym for a bit vector of a certain size but in the Bluespec case this type will be distinguished from other types of the same width. As a result, an attempt to enqueue an element that is of another type will fail at compile time, even if they are of the same size. This allows the designer to embed his intended use for the FIFO in the design and prevent unintended use without explicit casts. This strict typing system also contributes to Bluespec designs being understandable and extendable without breakage.

In this example we have mentioned the interface system and strict typing, which both help eliminate errors. The Bluespec rule structure and the implicit clock and reset signals in Bluespec eliminate other classes of error. A full description of the language is available elsewhere, but we consider the guarded interface structure as the most significant language feature for assisting the manageability of large scale projects.

We believe that the factors mentioned here are a key reason why open-source hardware projects have not seen much success. Successful contribution to an existing system requires an inordinate level of understanding of the system such that it is often easier to reimplement a complex system than to understand an equivalent system built by someone else. We have found that a code base in Bluespec SystemVerilog is relatively approachable and extensible and have had many successful undergraduate, masters, and PhD level projects significantly extend our Bluespec processor. We hope this success will continue more broadly with BERI as a universally available open-source project.

IV. BERI, a processor in Bluespec

This section describes the general purpose 64-bit MIPS core [7] featured in our DE4 tablet. We have named this base processor BERI (Bluespec Extensible RISC Implementation) and have packaged it as an open-source project which is available with the DE4 tablet stack. We will give a tour of the Bluespec Extensible RISC Implementation (BERI) with the

Bluespec

```
module mkFIFO(FIFO#(Packet_t));
  Reg#(Packet_t) fifomem <- mkReg(0);
  Reg#(Bool)     full    <- mkReg(False);

  method Action enq (Packet_t packet)
        if (!full);
    fifomem <= packet;
    full <= True;
  endmethod

  method Action deq if (full);
    full <= False;
  endmethod

  method Packet_t first if (full);
    return fifomem;
  endmethod
endmodule
```

Verilog

```
module fifo (
input clk,
input rstp,
input packet_t din,
input readp,
input writep,
output packet_t dout,
output reg fullp
);

packet_t fifomem;

always @(posedge clk) begin
  if (rstp == 1) begin
    dout <= {0:31};
  end else begin
    dout <= fifomem;
  end
end

always @(posedge clk)
  if (rstp == 1'b0) begin
    if (writep == 1'b1 && fullp == 1'b0) begin
      fifomem <= din;
      fullp <= 1'b1;
    end
  end else fifomem={default:0};

always @(posedge clk)
  if (rstp == 1'b0) begin
    if (readp == 1'b1 && fullp == 1'b1)
      fullp <= 1'b0;
  end

always @(count) begin
  if (count < MAX_COUNT)
    fullp <= 1'b0;
  else
    fullp <= 1'b1;
end

endmodule
```

Fig. 3. FIFO in Bluespec versus Verilog

premise that the reader is evaluating the design for extension and use in a project. The Bluespec Extensible RISC Implementation (BERI) is currently an in-order core with a 6-stage pipeline which implements the 64-bit MIPS instruction set used in the classic MIPS R4000 [8]. Some 32-bit compatibility features are missing and floating point support is experimental. Achievable clock speed is above 100 MHz on the Altera Stratix IV and average cycles per instruction is close to 1.2 when booting the FreeBSD operating system. In summary, the high-level design and performance of BERI is comparable to the MIPS R4000 design of 1991, though the design tends toward extensibility and clarity over efficiency in the micro-architecture.

A. Starting Point: Greg Chadwick's Multi-threaded Usermode MIPS64

The BERI project started with an early version of the MAMBA multi-threaded usermode 64-bit core in Bluespec developed by Greg Chadwick for use in many-core research [9], [10]. Chadwick's core was a standard 5-stage pipeline with an extra stage for the context FIFO. This core executed 8 independent threads with associated register files using round-robin scheduling. The BERI processor expands on this implementation with a register rename scheme for full pipelining of a single thread, added multiply and divide, added both level 1 and level 2 caches, and added a TLB and all system register support necessary to boot a general-purpose operating system. In the sections below, we annotate each component with the Bluespec module name which implements it to aid in browsing the Bluespec source code.

B. The BERI Pipeline

The BERI pipeline in its current state is 6 stages long. Figure 5 gives a detailed high-level overview of its structure.

1) *Instruction Fetch* submit program counter to memory system
2) *Scheduler/Register Rename* find instruction dependencies and prepare for forwarding
3) *Decode* fully define instruction behaviour
4) *Execute* perform arithmatic or assignment
5) *Memory Access* submit any memory operation
6) *Writeback* potentially commit updates to architectural state

Bluespec

```
typedef Bit#(32) Packet_t;
module mkFifoBench();
  Reg#(Packet_t) counter <- mkReg(0);
  FIFO#(Packet_t) fifo <- mkFIFO;

  rule enq;
    fifo.enq(counter);
    counter <= counter + 1;
  endrule

  rule deq;
    $display(fifo.first);
    fifo.deq();
  endrule
endmodule
```

Verilog

```
module fifoBench (clk, reset);
packet_t counter [0:31];
reg readp;
reg writep;
wire dout;
wire fullp;

fifo myfifo(clk, reset, counter, readp, writep,
            dout, fullp);
always @(posedge clk) begin
  if (reset) begin
    counter <= 0;
  else begin
    if (!fullp) begin
       writep <= 1'b1;
    end else
       writep <= 1'b0;

    if (fullp) begin
      readp <= 1'b1;
      $display(dout);
    end else
      readp <= 1'b0;

    counter <= counter + 1;
  end
end
```

Fig. 4. FIFO testbench in Bluespec versus Verilog

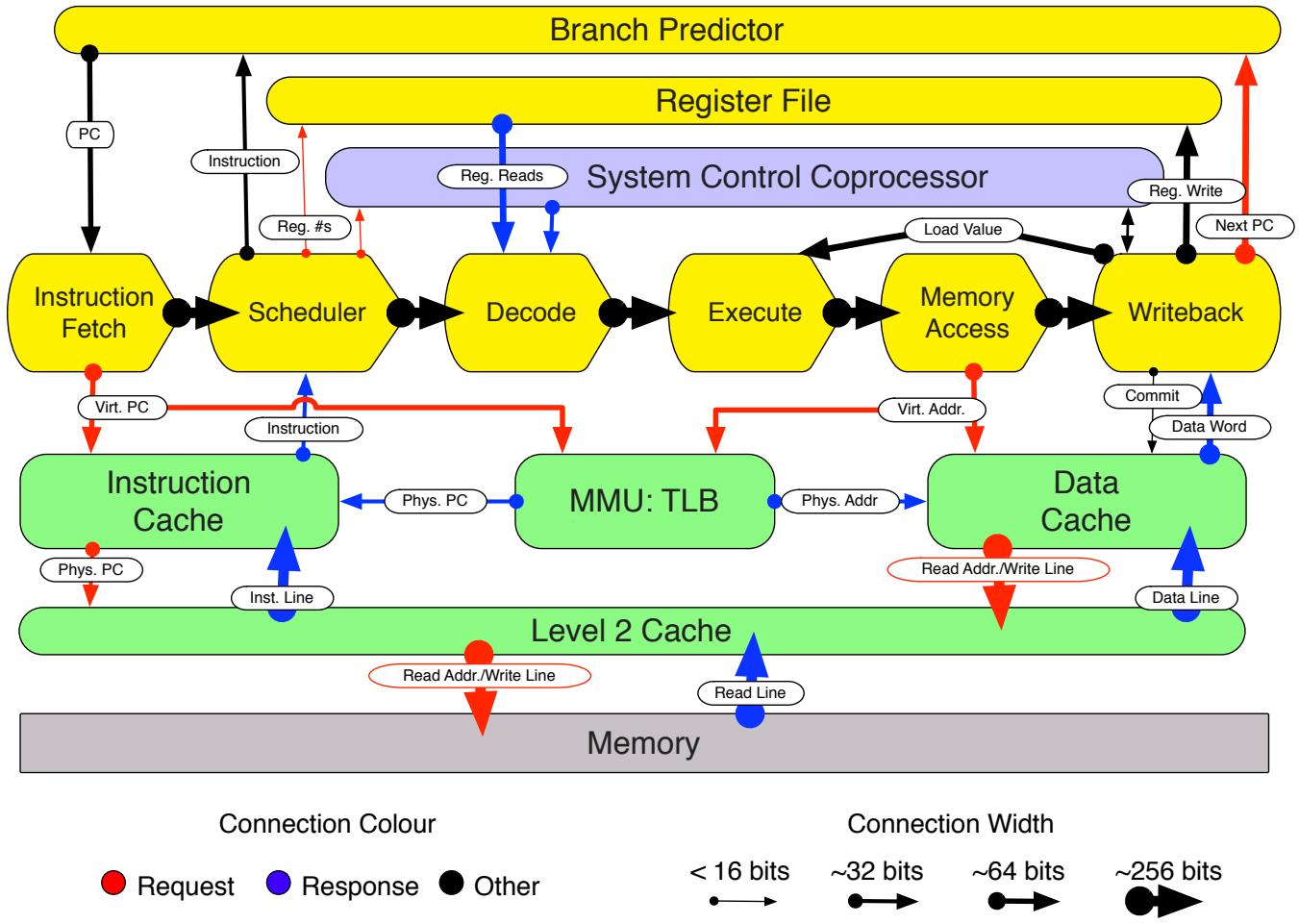


Fig. 5. MIPS Pipeline with Caches

```
rule instructionFetch(!theDebug.pause());
      // Get the next program counter (and current epoch) from the branch predictor.
    PcAndEpoch pce <- branch.getPc(nextId, False);
    Address nextPC = pce.pc;
    Bool breakpoint <- theDebug.checkPC(nextPC);
      // Initialize a default control token to insert in the pipeline.
    ControlTokenT ct = defaultControlToken;
    ct.epoch = pce.epoch;
    ct.id = nextId;
      // Set the renamed target register that this instruction will write to.
    ct.destRenamed = unpack(pack(nextId)[1:0]);
      // Also asign the next PC of the control token.
    ct.pc = nextPC;
    if (breakpoint) begin
      ct.dead = True;
      ct.flushPipe = True;
    end
      // Enq this control token to the toScheduler FIFO to be consumed
      // when the instruction is also ready to be consumed.
    toScheduler.enq(ct);
    nextId <= nextId + 1;
    theMem.instructionMemory.reqInstruction(nextPC[63:2], ct.id);
endrule
```

Fig. 6. Source code for BERI instruction fetch stage

1) Instruction Fetch (included in mkMIPSTop): The instruction fetch stage is implemented entirely in the mkMIPSTop module as a single rule. The Bluespec code is reproduced in Figure 6 as an example. Inspection of Figure 6 will identify the exercise of four interfaces for this stage of the pipeline. *branch.getPc* takes the next program counter and current epoch from the branch predictor. *theDebug.checkPC* verifies that this program counter is not a breakpoint. *toScheduler.enq* is just the interface of a FIFO for storing the ControlTokenT, which is the pipeline register type. Finally, *theMem.instructionMemory.reqInstruction* initiates a read from the instruction memory with the program counter. While this stage is simpler than the other stages, the approachability of the Bluespec style can be appreciated here. All of these interfaces other than the debug interface are also noted in Figure 5.

2) Scheduler/Register Rename (mkScheduler): The scheduler (or register rename) stage receives the instruction word from memory and performs a pre-decode of the instruction in order to fetch register operands and analyse dependencies. This stage categorizes instructions enough to determine which fields represent operand register numbers or destination register numbers. Operand register numbers are then submitted as reads to the two read ports of the register file and the destination register number is noted in a renamed register table. Every instruction is assigned one of four entries in the result table in the execute stage. Since the result of an instruction will be stored in the table for the next four instruction executions, the scheduler will direct any of the next four instructions that take the same register number as an operand to use the value from the table rather than the value fetched from the register file. This technique is generally called register renaming since we are continually changing the "names" of the four table entries in the execute stage to useful register numbers in the current working set. In addition to preparing register forwarding using the rename logic, the scheduler stage also submits register numbers for reads of both the register file and the system control processor (CP0), as noted in Figure 5. Finally, the scheduler stage reports to the branch predictor the branch type of the instruction based on its simple pre-decode to aide in predicting the next PC. Because of the MIPS branch delay slot, there is time to consider this analysis of each fetched instruction before beginning the fetch of any potential target.

3) Decode (mkDecode): The Decode stage of the pipeline consumes the register values read from the register file and also the instruction from the Scheduler stage and prepares the operands and ALU operation for the execute stage. The Decode stage sets every flag necessary for the correct operation of the pipeline such that there is no need in the rest of the pipeline to inspect the instruction itself. These flags include memory operation (Load or Store), memory operation width, signed or unsigned operation, etc. The decode stage reads from the register file and prepares the operands for the Execute stage such that the bits delivered to Execute can be directly fed into the arithmetic logic with the exception of values that must be taken from the result table in the Execute stage.

4) Execute (mkExecute): The Execute stage of the pipeline selects and performs an arithmetic operation as defined by the flags in the control token prepared by Decode. In the case of a multiply or divide instruction, an operation is initiated in the mkMulDiv module as a separate pipeline, a direct implication of the MIPS specification of multiply and divide as asynchronous operations. The main Execute logic is divided in a case statement between memory operations, branch operations, and normal arithmetic operations. While these could share the same logic, it was necessary to give the memory operation a dedicated and simplified add (or subtract) to reduce the critical path.

5) Memory Access (mkMemAccess): The Memory Access stage of the pipeline was introduced recently having been previously included in the Execute stage. The Memory Access

stage initiates any memory operation in the memory system. If there is no memory operation to be done, the control token is enqueued to the next stage with no change. A small amount of buffering in this stage allows arithmetic operations to be calculated during a data memory stall.

6) Writeback (mkWriteback): The Writeback stage of the pipeline happens in any one of three rules due to the need to exercise the memory read interface, the memory write interface, or neither. The Writeback stage consumes any result from memory as well as an exception report from the system control processor (CP0) and decides whether an instruction should commit. The Writeback stage then submits an exception report to CP0, possibly triggering an exception and flushing the pipeline. Writeback also reports a successful writeback to the register file to update architectural state and to the branch predictor unit which must decide whether the final "next PC" was that predicted by the unit. Writeback also must report a successful commit to the memory subsystem for both a read or a write in order to stop cancelled memory operations from proceeding. If the instruction does not commit because of a branch misprediction, the instruction epoch is incremented and the pipeline is flushed.

C. Caches

The BERI pipeline has 3 caches, an instruction level 1 cache (mkICache), a data level 1 cache (mkDCache), and a shared level 2 cache (mkL2Cache). The arrangement of these caches is noted in the bottom half of Figure 5. All levels of cache began with one design and so are very similar in structure. For example, all are direct mapped[1] with a 32-byte line size. The level 1 caches are both 16 KB in size, are write-through, virtually indexed and physically tagged, and complete in one cycle. The instruction and data cache lookups begins with the virtual address while the TLB begins a simultaneous lookup of the physical address also using the virtual address. In the next cycle, a rule in the level one caches consumes the physical address from the TLB (if it was a hit) and compares it with the result of its tag lookup. If the two match, it continues with the data it had fetched, either returning it to the main pipeline in the case of a read or merging it with new write data and storing back into its memory as well as writing through to the level 2. The instruction cache uses an 8-byte memory width internally as a balance between fill time (32-bytes/8-bytes = 4 cycles) and a maximum access width of 4 bytes. The data cache uses a full 32-byte memory width internally to facilitate wide writes from custom coprocessors which might want to take advantage of the naturally wide structure of the FPGA logic as well as the wide interface to main memory, as the main memory on the Terasic DE4 is DDR2 which runs at 400 MHz, transferring over 256 bits for every cycle of the BERI processor.

The level 2 cache is 64 KB in size and services the level 1 caches through a request merge module using a full 32-byte wide interface. The level 2 cache also responds in a single cycle on a hit, though the merge logic consumes one cycle in each direction yielding a total delay of 3 cycles for a level 1 miss that hits in the level 2.

[1]All levels of cache are direct mapped because memory is plentiful in the FPGA but combinational logic is not. Doubling or quadrupling cache size is nearly free, but adding complex selection logic is costly both in area and timing. Therefore we were able to increase hit rate more efficiently by increasing the size of a direct-mapped cache than by increasing associativity. We tried two way and four way set associativity in the level 2 and two way in the level one, both with way prediction. Timing was a problem in all cases and nearly the same hit-rate was achieved by simply quadrupling the size of the direct mapped cache.

D. Multiply and Divide Operations (mkMulDiv)

Multiply and divide operations are asynchronous and are initiated from the Execute stage of the pipeline. The multiply pipeline is only 2 stages long after being initiated in Execute. The divide pipeline processes 2 bits of operand every cycle for a worst case delay of 32 cycles for a 64 bit operand, though the pipeline will skip 8 contiguous zeros in the dividend so that dividing small numbers will not incur the worst case delay.

E. System Control Coprocessor (mkCP0)

The system control processor, or CP0, is integrated tightly with the pipeline as seen in Figure 5. The simplest function of CP0 is as a special purpose register file, allowing reads and writes from the main pipeline. CP0 registers are not forwarded in the pipeline, though the pipeline will conservatively block until completion of a CP0 update if it encounters an instruction which may read a CP0 register. CP0 registers include a CONFIG register which reports the features of this implementation of the processor, as well as a STATUS register with some writeable configuration fields. CP0 registers also include a timer register (COUNT) and a time-out register (COMPARE), the combination of which can be used to implement process time quanta by the operating system. A set of CP0 registers also implement TLB probes and updates, and the TLB logically belongs to CP0.

Exception handling in CP0 is especially complex since TLB exceptions triggered in CP0 must retain TLB miss information that is not held in the main pipeline. Thus potential victim addresses are stored in CP0 for both instruction and data memory operations. In the Writeback stage, the main pipeline requests the exception token from CP0, merges any exceptions with its own and branches to the correct trap handler. After the exception pipeline flush, the exception program counter, i.e. the victim address, and the cause information can be read from CP0 registers (EPC, CAUSE).

F. Translation Look-aside Buffer (mkTLB)

The translation look-aside buffer, or TLB, is instantiated inside the CP0 module and requests to the TLB pass through CP0 and are flagged for authorisation level. CP0 also notes all translated virtual addresses so that they will be available in the case of a miss exception.

Our translation lookaside buffer structure is unique. Large TLBs are usually set associative in modern designs, but the common MIPS spec only defines a fully associative table. The modern MIPS spec defines both a variable page sized table

(which is fully associative) and a fixed page size table (which is set associative), however FreeBSD, our target operating system, did not yet support this arrangement. However we noticed when debugging FreeBSD TLB operations that the operating system never used an indexed write above the WIRED register[2] (which was usually 1) without first doing a probe immediately before. The vast majority of TLB write operations used the *write random* instruction which allowed insertion using an arbitrary algorithm. We found that a TLB that only used fully associative entries at the bottom but that had a large direct mapped cache at the top worked transparently for FreeBSD.

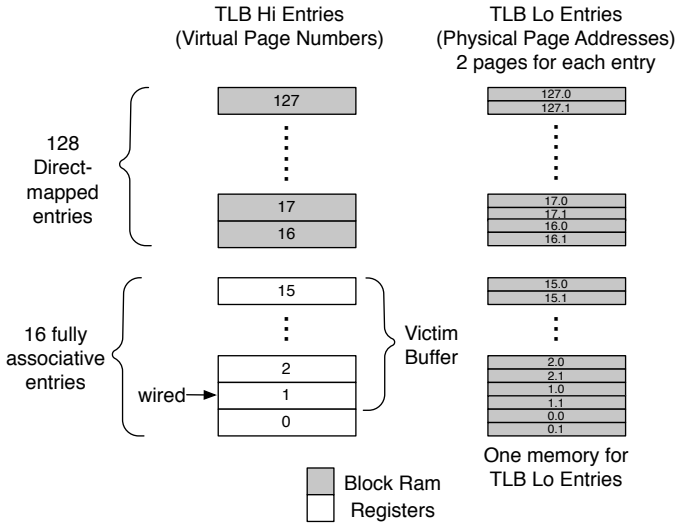


Fig. 7. BERI TLB structure

Our current implementation of the TLB has 16 fully associative entries and 128 direct mapped entries, as diagrammed in Figure 7. Any *write random* operation maps an entry into the top 128 entries using a simple hash of the lower bits of the virtual page number. A *write indexed* operation will only write to the specified index if it is in the bottom 16 entries, however a write request to an entry above 16 will simply write to the hash of the virtual page number, ensuring it will be found there on a lookup. If this *write indexed* operation immediately follows a *tlb probe* operation and is simply modifying an entry that was inserted using *write random*, then there is no anomaly. In FreeBSD, it seems that this is always the case.

A direct mapped TLB does not work in a simple arrangement because it is possible that a single instruction uses two virtual pages (for the instruction and data) that map to the same entry in the TLB. In this case the program ceases to make progress as it alternately misses the TLB for its instruction address and its data address. Therefore the entries in the fully associative set above the *wired* register (below which random writes should not occur) are treated as a victim buffer for the 128 direct mapped entries. Thus the hardware might move entries in the table without notifying the operating system. Since FreeBSD does not maintain a shadow status of the hardware page table and simply probes the hardware table before any modification, this does not cause a problem.

The mkTLB module contains only one TLB lookup but maintains a small cache of TLB entries for each interface, and the number of interfaces is variable. Currently each interface caches 4 page entries which are direct-mapped to prevent affecting the critical path. These entries are cleared on every TLB write to prevent coherence issues and a TLB cache miss causes a 3 cycle delay for a full TLB lookup.

G. Branch Prediction (mkBranch)

The branch predictor in BERI is less unusual than the TLB, but does take advantage of the peculiarity of the MIPS instruction set. As seen in Figure 5, the branch predictor unit has three key interfaces. The first is *getPc* which delivers a program counter and the associated instruction sequence epoch to the instruction fetch unit. The second is *putTarget* which receives the instruction and some flags from decode. The *putTarget* interface makes the actual prediction of the next program counter and produces two tokens, one to be consumed by *getPc* and the other to be consumed by *pcWriteback*. This third interface, *pcWriteback*, receives the canonical program counter writeback of every instruction and compares it against the prediciton made by *putTarget*. If the two do not match, the *pcWriteback* interface increments the instruction sequence epoch which causes the *getPc* interface to begin issuing program counters from the correct location and with the next epoch. All instructions of the previous epoch will be discarded by the writeback stage.

This branch predictor has been used as an exercise for a masters' level course and students were able to modify the branch predictor to improve hit rates for a software routine. The clear structure of the Bluespec language allows extensive modification by designers with only a cursory understanding of the original design. We hope that many better versions of the mkBranch unit will be developed for BERI as an open-source project.

H. Debug Unit (mkDebug)

BERI also includes a debug unit that is able to pause the pipeline, insert instructions, and trace execution. Commands are sent to the debug unit over a byte stream interface which is normally exported as an Avalon streaming interface. Details of this interface can be found in the open-source project documentation.

V. Simulation Infrastructure

The top level module for BERI when built for synthesis in Altera's Qsys tool is *mkTopAvalon* which translates a general purpose internal memory request and response format into an Avalon master interface and also includes a bus of peripherals written in Bluespec. However we have another Bluespec top

[2]The WIRED register specifies the TLB entry below which entries should not be arbitrarily replaced. The entries below the index in the WIRED register are "wired" and use of those mapped virtual addresses are guaranteed not to throw TLB exceptions.

Test Class	Number of Test Routines	Number of Tests
ALU	68	334
Branch	71	276
Cache	4	17
Coprocessor 0	66	312
Floating Point	36	93
Framework	18	77
Fuzz Regressions	10	10
Memory	45	307
TLB	16	50
Total	334	1476

TABLE I
TEST COUNTS

level interface, *mkTopSimulation*, which passes bus requests to a library in C through a Bluespec "BDPI" interface. The C library, called PISM, emulates the Avalon bus by inspecting a potential address and responding with a boolean indication of whether it is able to service a request to that address and then passing the request to the appropriate peripheral. We have PISM peripherals to emulate the Altera JTAG UART, a touch screen display, and the main DRAM memory. UNIX devices can also be presented as peripherals – files as SD card images, terminals as a console, and so on.

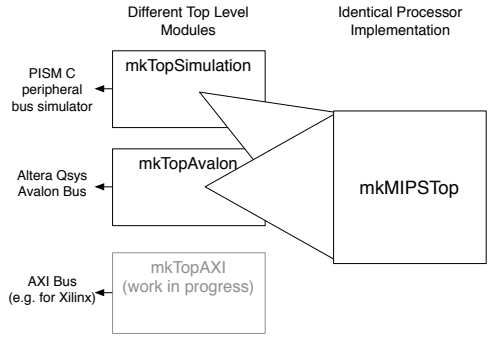


Fig. 8. BERI simulation versus synthesis instantiations

Relegating the differences between the simulated system and the synthesized system to the top level module ensures that our simulated design is as identical as possible to the synthesised design so that discrepancies between the two do not introduce bugs in hardware which are difficult to reproduce in simulation. This arrangement is diagrammed in figure 8. We expect to also produce a mkTopAXI top level module in the near future for use on Xilinx boards.

VI. TEST SUITE

The BERI open-source project also includes a unit test suite which currently includes over 1,400 tests spanning from basic functionality to TLB fills and cache tests. The test and category counts are reproduced in Table I.

The test suite can be run from the command line before every commit to ensure that changes to the processor have not broken functionality in unexpected ways. The test suite has also been integrated with the Jenkins automated test framework (Figure 9) so that it is run automatically on every commit to ensure the operational correctness of the hardware design at all times. This type of regression suite is common in open-source software projects but is less known in open-source hardware due to a lack of common infrastructure. We expect that the comprehensive test suite that accompanies BERI will assist its success as an open-source project by allowing contributers to the project to efficently check their own changes before submission for inclusion in future releases.

VII. SYSTEM-ON-CHIP DESIGN

We have used Bluespec to construct a variety of Avalon peripherals to build a system-on-chip around BERI. The top-level design, hiding clocks in the interests of space, can be seen in Figure 10.

The peripherals are held in a child Qsys file (Figure 11) to simplify the top level. They consist of a variety of components, some existing Altera Qsys components, some constructed entirely in Bluespec and some in Bluespec that instantiates existing Verilog.

The most complex custom component in our system is *MTL_Framebuffer_Flash*, which manages the flash and SRAM that share a bus on the DE4, and drives the displays using the SRAM as video memory. This component provides an Avalon Stream (not shown) to a Bluespec component that sends colour data to both the Terasic HDMI extension board and to the Terasic Multi-touch LCD Module. This framebuffer device is able to mirror the video streams from the LCD to HDMI to enable connecting the tablet to a projector.

Other components include the Ethernet MAC, as supplied by Altera, the Philips USB interface on the DE4 (for which we are yet to write FreeBSD drivers), the Altera University Program SD controller (written in VHDL and accessed via an Avalon interface). We use a variety of UARTs, both Altera's JTAG UART for terminal access and for access to the BERI debug unit, and an NS16550-compatible UART for physical RS232 access to the DE4 board (since the 16550 register layout is already supported in FreeBSD).

VIII. PORTING FREEBSD

FreeBSD is a widely used, contemporary, general-purpose, open-source operating system found in server and workstation environments, but also as the foundation for a broad range of mobile, embedded, and appliance products [11]. For example, you can find FreeBSD in the underlying OS layers of Apple OS X, Apple iOS, NetApp OnTap/GX, Juniper Junos, and Sony Playstation 3. As we began our project, FreeBSD already supported several MIPS processors and emulators, so porting FreeBSD to BERI was straightforward. It was necessary to extend FreeBSD in two ways: (1) adding BERI-specific boot code and device enumeration support; and (2) developing device drivers for Altera IP cores as well as hard peripherals found on the Terasic DE4.

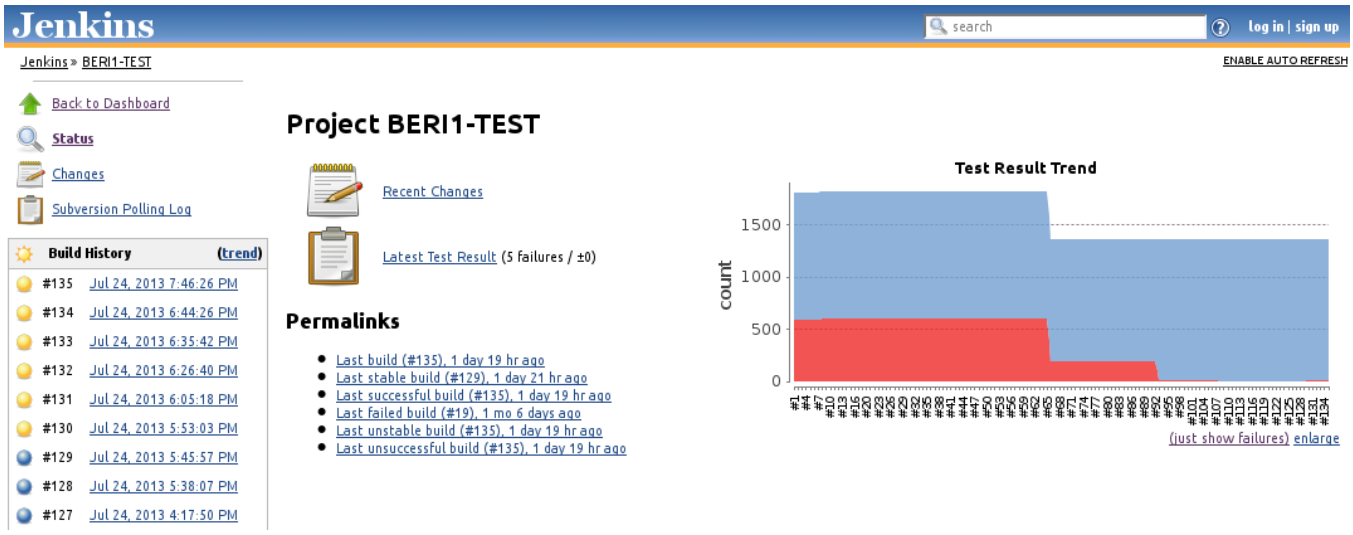


Fig. 9. BERI tests are run by Jenkins on every source commit

We have upstreamed the majority of our BERI platform code to the open-source project, and plan to merge the rest so that FreeBSD 10.0 fully supports BERI, making it accessible to a wide audience of potential academic and open-source consumers. Support for FreeBSD brings access to tens of thousands of off-the-shelf open-source libraries and applications, including the gcc compiler suite and Apache web server.

A. Boot and configuration

A small built-in ROM in BERI is able to relocate a FreeBSD kernel out of flash, or, if a DIP switch is set, make use of a kernel loaded directly into DRAM using JTAG. After relocation, the boot ROM interprets the kernel's ELF header, passing in information on DRAM configuration when it jumps to the kernel's start routine. We describe BERI board configurations using Flat Device Trees (FDT), which are also used on PowerPC and ARM-based systems [12]. FDT files describe the bus topology, generally simple for system-on-chip configurations, including memory addresses and interrupt information for peripherals available to the operating system. Figure 13 excerpts the FDT configuration file for the DE4 tablet, including an on-board 1GB DRAM (in the standard configuration), BERI CPU, BERI programmable interrupt controller (PIC), NS16550-compatible RS232 port, Altera JTAG UART, Altera University Program SD Card IP Core, on-board Intel Strata-Flash, Altera triple-speed Ethernet MAC, and Terasic multitouch display (MTL) (Figure 14).

A typical configuration loads a FreeBSD kernel from the DE4 on-board flash, and then uses an embedded memory root file system or SD Card root file system. In tablet configurations, *nios2-terminal* can be used to connect to the Altera JTAG UART to reach the OS console. In rack-mount configurations, we use the DE4 RS232 serial port connected to a console server.

B. Device drivers

Our BERI FPGA design exposes Altera soft-cores and on-board peripheral ICs via a memory-mapped Avalon bus. FDT notifies the FreeBSD device-driver framework of the locations of the device mappings and interrupt configurations. FreeBSD's BERI PIC driver programs the PIC to map enabled device interrupt sources to MIPS IRQ lines, suppressing interrupts from other devices.

We have developed device drivers for three Altera IP cores: the JTAG UART (*altera_jtag_uart*), triple-speed MAC (*atse*), and SD Card (*altera_sdcard*), which implement low-level console/tty, Ethernet interface, and block storage classes. In addition, we have implemented a generic driver for Avalon-attached devices (*avgen*), which allows memory mapping of arbitrary bus-attached devices without interrupt sources, such as the DE4 LED block, BERI configuration ROM, and DE4 fan and temperature control block.

Finally, we have developed a device driver for the Terasic multitouch display (*terasic_mtl*), which implements a memory-mapped pixel buffer, system console interface for the text frame buffer, and memory-mapped touchscreen input FIFO. Using this driver, UNIX can present a terminal interface, but applications can also overlay graphics and accept touch input.

C. Software platform

FreeBSD provides us with access to the complete open-source ecosystem of development tools and UNIX applications. In addition to the FreeBSD base system, which includes core components such as UNIX command-line tools, DHCP client, and SSH server, we are able to cross-build tens of thousands of third-party applications. Cross-building from x86 systems is important due to the limited CPU, memory, and storage resources of the BERI tablet, but we distribute pre-built applications in the form of FreeBSD *pkgng* packages based on

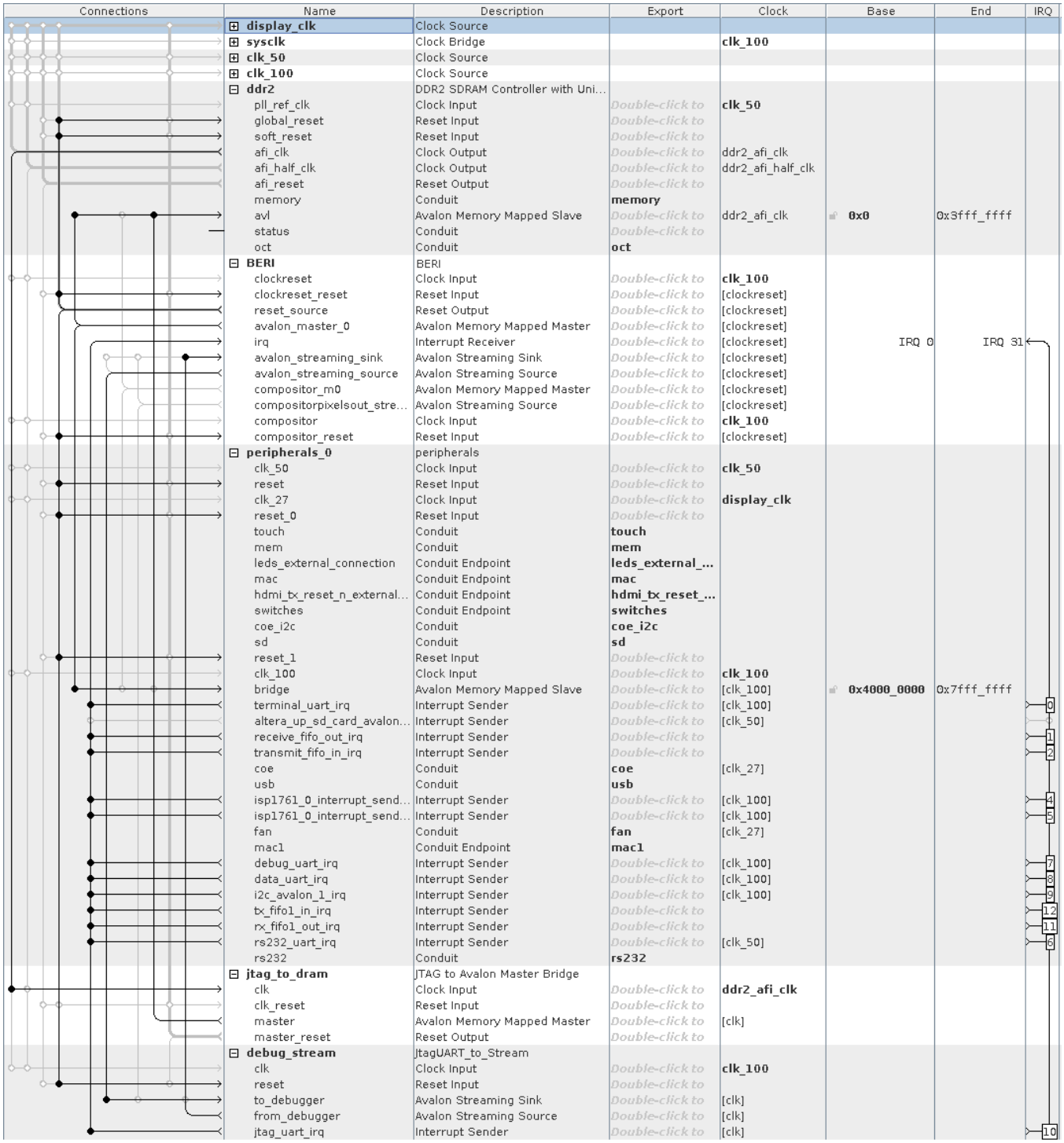
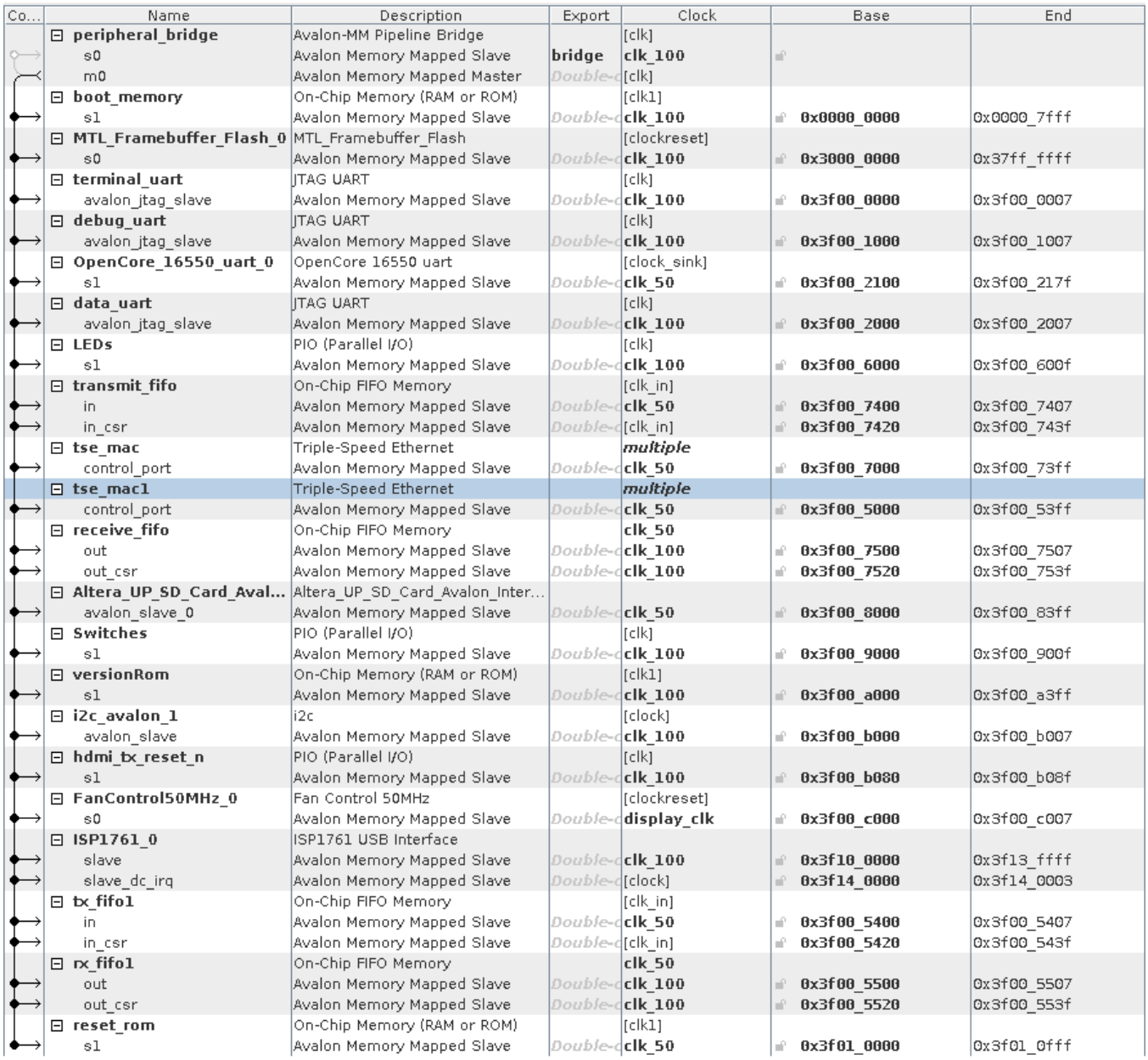


Fig. 10. Top level Qsys project (clocks not shown)



Co...	Name	Description	Export	Clock	Base	End
	peripheral_bridge	Avalon-MM Pipeline Bridge		[clk]		
	s0	Avalon Memory Mapped Slave	**bridge**	**clk_100**		
	m0	Avalon Memory Mapped Master	Double-c	[clk]		
	boot_memory	On-Chip Memory (RAM or ROM)		[clk1]		
	s1	Avalon Memory Mapped Slave	Double-c	**clk_100**	**0x0000_0000**	0x0000_7fff
	MTL_Framebuffer_Flash_0	MTL_Framebuffer_Flash		[clockreset]		
	s0	Avalon Memory Mapped Slave	Double-c	**clk_100**	**0x3000_0000**	0x37ff_ffff
	terminal_uart	JTAG UART		[clk]		
	avalon_jtag_slave	Avalon Memory Mapped Slave	Double-c	**clk_100**	**0x3f00_0000**	0x3f00_0007
	debug_uart	JTAG UART		[clk]		
	avalon_jtag_slave	Avalon Memory Mapped Slave	Double-c	**clk_100**	**0x3f00_1000**	0x3f00_1007
	OpenCore_16550_uart_0	OpenCore 16550 uart		[clock_sink]		
	s1	Avalon Memory Mapped Slave	Double-c	**clk_50**	**0x3f00_2100**	0x3f00_217f
	data_uart	JTAG UART		[clk]		
	avalon_jtag_slave	Avalon Memory Mapped Slave	Double-c	**clk_100**	**0x3f00_2000**	0x3f00_2007
	LEDs	PIO (Parallel I/O)		[clk]		
	s1	Avalon Memory Mapped Slave	Double-c	**clk_100**	**0x3f00_6000**	0x3f00_600f
	transmit_fifo	On-Chip FIFO Memory		[clk_in]		
	in	Avalon Memory Mapped Slave	Double-c	**clk_50**	**0x3f00_7400**	0x3f00_7407
	in_csr	Avalon Memory Mapped Slave	Double-c	[clk_in]	**0x3f00_7420**	0x3f00_743f
	tse_mac	Triple-Speed Ethernet		***multiple***		
	control_port	Avalon Memory Mapped Slave	Double-c	**clk_50**	**0x3f00_7000**	0x3f00_73ff
	tse_mac1	Triple-Speed Ethernet		***multiple***		
	control_port	Avalon Memory Mapped Slave	Double-c	**clk_50**	**0x3f00_5000**	0x3f00_53ff
	receive_fifo	On-Chip FIFO Memory		**clk_50**		
	out	Avalon Memory Mapped Slave	Double-c	**clk_100**	**0x3f00_7500**	0x3f00_7507
	out_csr	Avalon Memory Mapped Slave	Double-c	**clk_100**	**0x3f00_7520**	0x3f00_753f
	Altera_UP_SD_Card_Aval...	Altera_UP_SD_Card_Avalon_Inter...				
	avalon_slave_0	Avalon Memory Mapped Slave	Double-c	**clk_50**	**0x3f00_8000**	0x3f00_83ff
	Switches	PIO (Parallel I/O)		[clk]		
	s1	Avalon Memory Mapped Slave	Double-c	**clk_100**	**0x3f00_9000**	0x3f00_900f
	versionRom	On-Chip Memory (RAM or ROM)		[clk1]		
	s1	Avalon Memory Mapped Slave	Double-c	**clk_100**	**0x3f00_a000**	0x3f00_a3ff
	i2c_avalon_1	i2c		[clock]		
	avalon_slave	Avalon Memory Mapped Slave	Double-c	**clk_100**	**0x3f00_b000**	0x3f00_b007
	hdmi_tx_reset_n	PIO (Parallel I/O)		[clk]		
	s1	Avalon Memory Mapped Slave	Double-c	**clk_100**	**0x3f00_b080**	0x3f00_b08f
	FanControl50MHz_0	Fan Control 50MHz		[clockreset]		
	s0	Avalon Memory Mapped Slave	Double-c	**display_clk**	**0x3f00_c000**	0x3f00_c007
	ISP1761_0	ISP1761 USB Interface				
	slave	Avalon Memory Mapped Slave	Double-c	**clk_100**	**0x3f10_0000**	0x3f13_ffff
	slave_dc_irq	Avalon Memory Mapped Slave	Double-c	[clock]	**0x3f14_0000**	0x3f14_0003
	tx_fifo1	On-Chip FIFO Memory		[clk_in]		
	in	Avalon Memory Mapped Slave	Double-c	**clk_50**	**0x3f00_5400**	0x3f00_5407
	in_csr	Avalon Memory Mapped Slave	Double-c	[clk_in]	**0x3f00_5420**	0x3f00_543f
	rx_fifo1	On-Chip FIFO Memory		**clk_50**		
	out	Avalon Memory Mapped Slave	Double-c	**clk_100**	**0x3f00_5500**	0x3f00_5507
	out_csr	Avalon Memory Mapped Slave	Double-c	**clk_100**	**0x3f00_5520**	0x3f00_553f
	reset_rom	On-Chip Memory (RAM or ROM)		[clk1]		
	s1	Avalon Memory Mapped Slave	Double-c	**clk_50**	**0x3f01_0000**	0x3f01_0fff

Fig. 11. Qsys peripherals (only Avalon memory-mapped interfaces shown)

the 64-bit MIPS ISA. In the rack-mount environment, BERI nodes also have access to the Network File System (NFS) to supplement the on-board SD Card.

We have developed a small set of sample multi-touch applications for the tablet platform, including a touch drawing tool and a slide presentation package. Additionally, a UNIX shell can be used directly from the touchscreen via an on-screen keyboard app, allowing access to UNIX tools such as top (Figure 15). Both the operating system and FPGA bitfiles can be flashed onto the on-board Intel StrataFlash allowing in-field CPU upgrades.

IX. Conclusion

Currently, open-source FPGA design centres around projects such as OpenCores. OpenCores acts as a repository for HDL components, and encourages common interfaces through specifications such as the Wishbone interconnect[13].

However open-source FPGA design has not become commonplace for a number of reasons. Traditional VHDL and Verilog lack support for robust interfaces for data flow between components. Common bus standards such as AXI, Avalon and Wishbone help for top-level system-on-chip interconnect, but this does not suit interconnection of smaller component parts. Interface design and testing is a time-consuming and error-

```
Mini Bootloader Run
Test I2C on HDMI chip
Reset HDMI chipSetting clock_scale to 0x00000000000004E2
clock scale = 0x00000000000004E2 - passed
entry: platform_start()
cmd line:
envp:
memsize = 3eeffc00
Cache info:
  picache_stride    = 4096
  picache_loopcount = 4
  pdcache_stride    = 4096
  pdcache_loopcount = 4
cpu0: Unknown cid 0 processor v0.4
  MMU: Standard TLB, 40 entries
  L1 i-cache: direct-mapped with 512 sets, 32 bytes per
      line
  L1 d-cache: direct-mapped with 512 sets, 32 bytes per
      line
  Config1=0xcee07040<COP2>
Physical memory chunk(s):
0x1f95000 - 0x3eefffff, 1022799872 bytes (249707 pages)
Maxmem is 0x3ef00000
KDB: debugger backends: ddb
KDB: current backend: ddb
Copyright (c) 1992-2013 The FreeBSD Project.
Copyright (c) 1979, 1980, 1983, 1986, 1988, 1989, 1991,
    1992, 1993, 1994
       The Regents of the University of California. All
           rights reserved.
FreeBSD is a registered trademark of The FreeBSD Foundation
    .
FreeBSD 10.0-CURRENT #0 226199: Fri Mar 22 20:15:29 UTC
    2013
...

atse0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST>
    metric 0 mtu 1500
       options=80048<VLAN_MTU,POLLING,LINKSTATE>
       ether 02:07:ed:94:90:f0
       nd6 options=29<PERFORMNUD,IFDISABLED,AUTO_LINKLOCAL
           >
       media: Ethernet autoselect (1000baseT <full-duplex
           >)
       status: active

Wed Jul 17 20:52:17 UTC 2013

FreeBSD/mips (beri1) (ttyj0)

login: root
Jul 17 20:52:26 beri1 login: ROOT LOGIN (root) ON ttyj0
#
```

Fig. 12. Excerpt from BERI boot log

```
model = "SRI/Cambridge BeriPad (DE4)";
compatible = "sri-cambridge,beripad-de4";
cpus {
  cpu@0 {
    device-type = "cpu";
    compatible = "sri-cambridge,beri";
  };
};
soc {
  memory {
    device_type = "memory";
    reg = <0x0 0x40000000>;
  };
  beripic: beripic@7f804000 {
    compatible = "sri-cambridge,beri-pic";
    interrupt-controller;
    reg =  <0x7f804000 0x400 0x7f806000 0x10
      0x7f806080 0x10 0x7f806100 0x10>;
  }
  serial@7f002100 {
    compatible = "ns16550";
    reg = <0x7f002100 0x20>;
  };
  serial@7f000000 {
    compatible = "altera,jtag_uart-11_0";
    reg = <0x7f000000 0x40>;
  };
  sdcard@7f008000 {
    compatible = "altera,sdcard_11_2011";
    reg = <0x7f008000 0x400>;
  };
  flash@74000000 {
    partition@20000 {
      reg = <0x20000 0xc00000>;
      label = "fpga0";
    };
    partition@1820000 {
      reg = <0x1820000 0x027c0000>;
      label = "os";
    };
  };
  ethernet@7f007000 {
    compatible = "altera,atse";
    reg = <0x7f007000 0x400 0x7f007500 0x8
      0x7f007520 0x20 0x7f007400 0x8
      0x7f007420 0x20>;
  };
  touchscreen@70400000 {
    compatible = "sri-cambridge,mtl";
    reg = <0x70400000 0x1000
      0x70000000 0x177000 0x70177000 0x2000>;
  };
};
```

Fig. 13. Excerpt from Flat Device Tree (FDT) description of the DE4-based BERI tablet.

prone part of hardware design. Lack of type-checking and compile-time verification adds to this problem.

The lack of language support for robust interfaces means that typically designers use their own interfaces within and between sub-modules. This makes them harder to understand to the outsider, as the interesting functionality is obscured by the control flow.

A project that is harder to understand is one that requires more thorough documentation. Not only is such documentation time-consuming to write, the increased prerequisite understanding also raises the barrier of entry to new participants in an open-source project.

Furthermore, FPGA design already has higher barriers to entry than open-source software. FPGA boards have a non-zero cost. Synthesis favours powerful computers and is time consuming, often requiring hours per compile. Hardware debugging tools lag behind their software equivalents in ease-of-use and functionality. FPGA tools are also less amenable to traditional software engineering practices.

Taken together, these issues mean an open-source FPGA project faces a challenging up-hill battle toward acceptance and building a productive community.

In this work we have begun to address these issues in a number of ways. First, we started with a higher level hardware description language that makes writing correct code easier, and simplifies the data flow within and between modules. Clear interfaces make the design of a large component such as the BERI processor a tractable task and enable easy modification by others.

Second, the language has a cycle-accurate simulator that is

Fig. 14. BERIpad booting FreeBSD

Fig. 15. Process list and soft keyboard

more efficient than a VHDL or Verilog simulation, including the ability to boot an OS in simulation on a common PC. We provide a test suite that is run against the processor after every version control commit, which means regressions in such a complex piece of code are found quickly. This makes it easier for newcomers to extend the processor without fear that they may break other parts of the design.

We have implemented the processor on a commercial FPGA board and built a portable platform which means we can easily demonstrate it to people who do not have access to the hardware. The system-on-chip also contains peripherals that are designed using a similar methodology to the processor. The physical construction and system-on-chip design have also been open-sourced.

We have ported the open-source FreeBSD operating system to the platform. A feature-rich OS makes the platform a much more flexible one, brings a large catalogue of existing software, and provides a familiar environment for open-source software development. We are able to offer SSH access to a board running the BERI processor to those who do not have access to FPGA hardware, including the ability to upgrade the CPU remotely.

Open-source hardware still faces many challenges, but we hope by taking a software engineering approach we can begin to address them.

Download

Download our work from http://www.beri-cpu.org/

Acknowledgements

We would like to thank our colleagues — especially Jonathan Anderson, Ross Anderson, Gregory Chadwick, David Chisnall, Nirav Dave, Wojciech Koszek, Ben Laurie, Will Morland, Steven Murdoch, Robert Norton, Peter Neumann, Philip Paeps, Michael Roe, Colin Rothwell, Ben Thorner and Philip Withnall.

This work is part of the CTSRD Project that is sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8750-10-C-0237. The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the Department of Defense. Portions of this work were supported by Google, Inc.

References

[1] J. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo, "Netfpga–an open platform for gigabit-rate network switching and routing," in *Microelectronic Systems Education, 2007. MSE '07. IEEE International Conference on*, 2007, pp. 160–161.

[2] R. Bergamaschi, S. Bhattacharya, R. Wagner, C. Fellenz, M. Muhlada, F. White, J.-M. Daveau, and W. Lee, "Automating the design of SOCs using cores," *IEEE Design & Test of Computers*, vol. 18, no. 5, pp. 32–45, 2001.

[3] OpenCores website. http://opencores.org/

[4] Xilinx Inc, "AXI reference guide," 2011.

[5] S. W. Moore, P. J. Fox, S. J. T. Marsh, A. T. Markettos, and A. Mujumdar, "Bluehive - a field-programable custom computing machine for extreme-scale real-time neural network simulation," in *Proceedings of FCCM 2012*, pp. 133–140.

[6] *Bluespec SystemVerilog Version 3.8 Reference Guide*, Bluespec, Inc., Waltham, MA, November 2004.

[7] R. Watson, P. Neumann, J. Woodruff, J. Anderson, R. Anderson, N. Dave, B. Laurie, S. Moore, S. Murdoch, P. Paeps, *et al.*, "CHERI: A Research Platform Deconflating Hardware Virtualization and Protection," in *Workshop paper, Runtime Environments, Systems, Layering and Virtualized Environments (RESoLVE 2012)*, 2012.

[8] S. Mirapuri, M. Woodacre, and N. Vasseghi, "The MIPS R4000 processor," *IEEE Micro*, vol. 12, no. 2, pp. 10–22, 1992.

[9] G. A. Chadwick and S. W. Moore, "Mamba: A scalable communication centric multi-threaded processor architecture," in *Computer Design (ICCD), 2012 IEEE 30th International Conference on*. IEEE, 2012, pp. 277–283.

[10] G. A. Chadwick, "Communication centric, multi-core, fine-grained processor architecture," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-832, Apr. 2013. http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-832.pdf

[11] M. K. McKusick and G. V. Neville-Neil, *The Design and Implementation of the FreeBSD Operating System*. Pearson Education, 2004.

[12] Power.org, "Power.org standard for Embedded Power Architecture Platform Requirements (ePAPR)," 2011.

[13] OpenCores, "WISHBONE system-on-chip (SoC) interconnection architecture for portable IP cores," 2010.