Bluebox: Rapid Hardware Prototyping with Debugging Extensions

Oriol Arcas*[1,2], Nehir Sönmez†[2], Adrián Cristal‡[1,2], and Osman S. Unsal§[2]

[1]Department of Computer Architecture, Universitat Politècnica de Catalunya - BarcelonaTech
[2]Barcelona Supercomputing Center - Centro Nacional de Supercomputación

February 25, 2013

Abstract

This paper describes the development of Bluebox, a synthesizable processor architecture. We describe and apply a design methodology that enables rapid prototyping, based on development stage integration and unified tools. We use Bluespec SystemVerilog, a high level hardware description language, to implement this hardware model. By defining debugging, tracing and checkpointing extensions, and a GDB interface, we point out the usefulness of our infrastructure to port Linux. In our integrated development environment, we can run standard software objects, perform line-by-line C stepping, output and visualize execution traces, and save and restore simulation sessions.

1 Introduction

In digital systems design, two properties are desirable for the tools and programming languages used. First, productivity in terms of programming and simulation speed. Second, accuracy in terms of fidelity to the final hardware model. Traditionally, it seemed that the available tools could not have both at the same time. For rapid design-space exploration and model simulation, languages like C and its variants (C++, SystemC, etc.) are preferred for their high productivity. To develop the final hardware model, register-transfer level (RTL), mostly Verilog and VHDLW are chosen because of its accuracy and tight control over the resulting circuitry. Over time, some of these languages have become *de facto* standards and families of variations and extensions have appeared based on them.

Designers tend to choose the most appropriate tools and languages for each step of the process. This methodology has several inconveniences for the designer, most of them related to the poor integration between the different stages of the workflow. As a consequence, the work done in one of the stages can hardly be reused in the following ones. Conversely, trying to use the same tools or languages in all the stages may lead to penalties in productivity. For instance, using Verilog for the initial functional simulation can result in long simulation times and a lower programmability than using C++.

The situation is worse for large digital systems. There exist different languages and tools that tried to address this issue. A first set of tools can be grouped together under the generic tag "C-to-gates": Catapult C [1], Impulse C [2] and the recent Xilinx Vivado HLS [3] (formerly AutoESL) are three examples. These tools and development environments try to map C-like hardware descriptions to different RTL hardware description languages (HDL). Other new generation HDLs are based on functional programming paradigms, suggesting that imperative programming does not match well with hardware

*oriol.arcas@bsc.es
†nehir.sonmez@bsc.es
‡adrian.cristal@bsc.es
§osman.unsal@bsc.es

development. Lava [4] and Bluespec SystemVerilog [5] are two of those languages strongly influenced by the Haskell functional language.

These languages and tools try to integrate into the design flow's stages while maintaining the productivity and accuracy levels. This unified development environment has the following benefits:

- High productivity compared to RTL HDLs.
- Short simulation times.
- Cross probing, or the ability to map information from from one stage of the design flow to another.
- The work done in one stage of the design process can be reused in others.

Under these conditions new possibilities appear to the designer, that previously were impractical or directly impossible. As the development of an architecture advances, debugging and verification needs have become stronger. However, traditional debugging methods for hardware models are slow and tedious for the developers. More advanced techniques are required in order to verify digital systems, in our case a RISC processor running Linux. These new methods should help to identify problems related to the processor's interrupt mechanism, the virtual memory or the operating system's process scheduler.

In this work we present the Bluebox system-on-chip, a RISC processor implemented in Bluespec SystemVerilog (BSV). We took an educational MIPS-compatible processor core written in BSV and upgraded it to a full Linux-capable processor core. During this process, we developed debugging and checkpointing extensions. We show how this tools and infrastructure greatly facilitated our objective. The main contribution of this work is the description of the integrated development/debugging methodology. Thanks to it, we have been able to perform complex verifications during the development process, as opposed to doing them in a separate development stage or environment.

The architectural details of Bluebox are depicted in 2. In Section 3 we explain how we instrumented the BSV code with unobtrusive debugging extensions. We describe how we implemented a checkpointing mechanism that allows resuming the simulations at any cycle in Section 4. We defined a generic debugging interface, explained in Section 5, that can be controlled from the state-of-the-art debugger GDB. Finally, in Section 6 we present the conclusions of this article.

2 The Bluebox architecture

In order to perform computer architecture research a prototyping platform was needed. During the development of the [6] and [7] research platforms, we learned valuable lessons about FPGA-based multiprocessor architectures: The low productivity of RTL languages; the difficulty of simulating RTL models or debugging on-FPGA designs; and the importance of using standard tools and languages, both in the software and the hardware layers.

The objective of this project was producing an architecture with the following characteristics:

- Developed using a high level hardware description language BSV.
- The processing element should leave a small footprint on the final technology, e.g. an FPGA. In the future, our intention is to develop a multicore architecture and fit as many cores as possible.
- The performance should be high enough to compete with a detailed software simulator, so that the development effort was worthwhile. Other well-known RISC systems reach frequencies from 50 MHz [8] to 200 MHz [9] on a similar platform (a Xilinx Virtex 5 FPGA). We decided that our platform should run at a frequency of at least 100 MHz.
- The microarchitecture should resemble as much as possible modern RISC processors: at least a 5-stage, in-order pipeline, with support for virtual memory, interrupts and exceptions.
- It should be able to run a full Linux 2.6 kernel.
- The processor should execute a well-known Instruction Set Architecture (ISA), instead of a new or exotic one, so that we could use state-of-the-art tools like GCC and GLIBC.

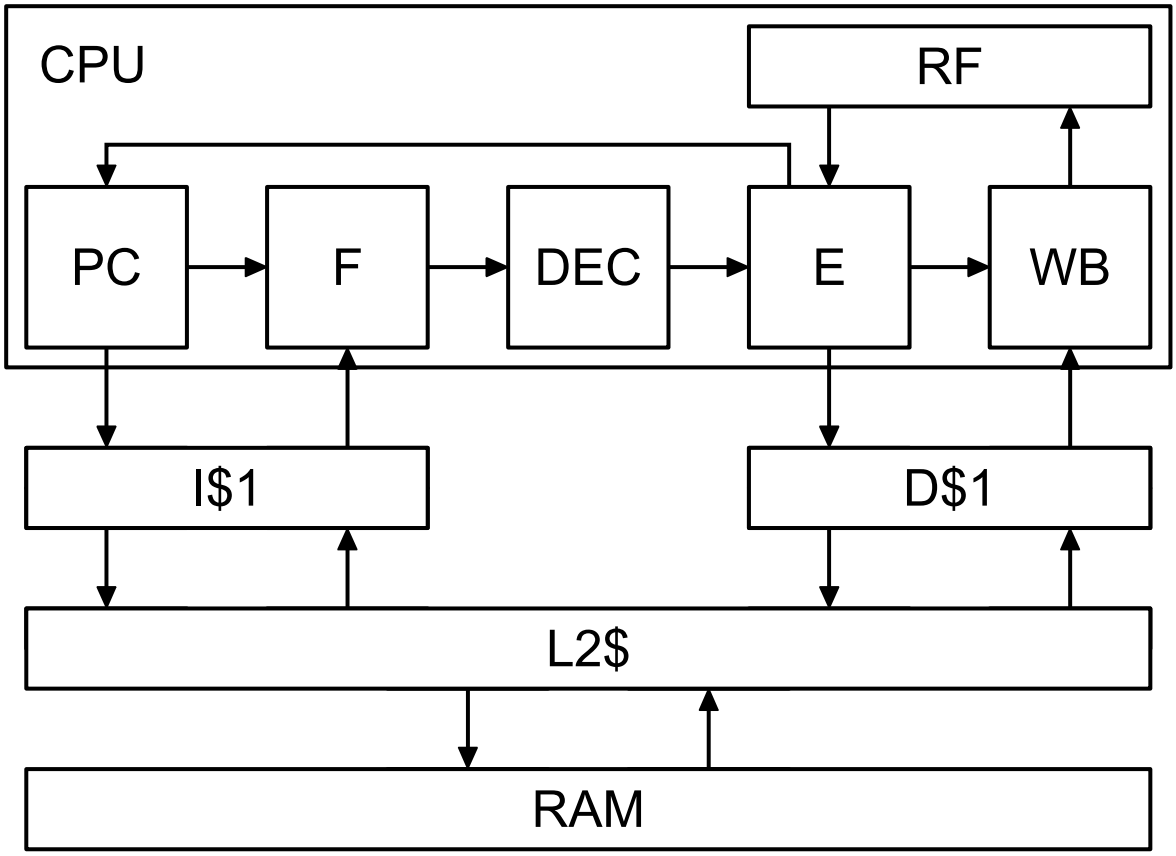

Figure 1: Original sMIPS architecture with a 5-stage pipeline.

The result is the Bluebox architecture. This initial version consists in a 32-bit RISC processor with an L1 and L2 cache hierarchy, capable of running the Linux 2.6 kernel at 100 MHz on a Virtex 5 FPGA. Our architecture can execute the MIPS-I ISA and has similar characteristics to a MIPS R3000 processor.

2.1 The Bluebox processor

The Bluebox processing element is based on the sMIPS processor [10]. The sMIPS processor was originally developed as an educational tool, and it is written completely in BSV. The original implementation is a non-pipelined RISC processor that executes a subset of the MIPS-I ISA. It does not support virtual memory and it does not have an exception handling mechanism. As a particularity, it does not support branch delays, a mandatory characteristic of MIPS architectures.

We started the development of the Bluebox processor from a version of the sMIPS processor with an extended 5-stage pipeline. During the development process, all the additions have been carefully parametrized and can be configured with macros. Any new features can be disabled without affecting the functionality of the rest of the system.

Thanks to BSV automatic handshaking mechanisms and inherent rule conditions, the triggering of the stages is managed transparently to the designer. This means that hardware units are only executed when some implicit and explicit conditions are satisfied. One particularly useful example are BSV's builtin FIFOs. These hardware queues include implicit signals that, when interfaced by a hardware unit, will pause the execution when no data is available, or when trying to push data to a full queue. In Bluebox, the triggering of the stages is managed transparently to the designer and with very simple semantics data is passed from one stage to the other. This produces the typical behavior of an in-order pipeline:

Table 1: Instructions added to the sMIPS processor.

Mnemonic	**Description**
MULT, DIV, MULTU, DIVU, MFLO, MFHI, MTLO, MTHI	Integer multiply and divide.
LB, LH, LBU, LHU, SB, SH	Partial word load and store.
LWL, LWR, SWL, SWR	Unaligned memory access.
MFC0, MTC0, MFC1, MTC1, MFC2, MTC2	Coprocessor management.
TLBP, TLBR, TLBWI, TLBWR	TLB management.
SYSCALL, BREAK, RFE	System calls and exceptions.

every stage will remain paused (i.e. will not modify the state of the system) while there is no data to process from the previous stage, or while the next stage is not ready to handle the current data. The automatic handling of all the "plumbing" tasks by BSV greatly simplified the size and complexity of the code, compared to traditional Verilog and VHDL descriptions.

Bluebox inherited its pipeline partition from sMIPS. The sMIPS 5-stage architecture is shown in figure 1. The first stage, Program Counter (PC), keeps the address of the next instruction to be executed. This address is sent to the level 1 instruction cache, which optimistically responds in 1 cycle in case the data was cached. The next stage, Fetch (F), receives the instruction's opcode and pushes it to the Decoding (DEC) stage. During this step, the opcode is decoded to a very long instruction word. At the same time, the instructions are checked for any semantic or security errors.

The Execute (E) stage is the first that modifies the state of the processor. Any instruction that reaches this step is considered as already executed (except memory access errors, which will be described in the following section 2.3). At this part of the process, the Register File operands are read, and arithmetic operations are performed with their contents. The result is sent to the following stage, Write Back (WB). If the instruction required a memory access, the address is sent to the level 1 data cache, which will perform it and send the result to the WB stage. During this stage, the arithmetic results or the memory contents are stored in the Register File. Bluebox automatically handles any data dependency hazards between the E and WB stages, although it is not mandatory in the MIPS-I architecture.

To fully support the MIPS-I ISA, new instructions and functionalities had to be implemented. As said before, sMIPS intentionally does not support branch delay execution. This mechanism was included as a configuration option of Bluebox, enabled by default, with very simple modifications. More complex hardware had to be designed to support some new instructions. Instructions added to the sMIPS to obtain the Bluebox architecture are listed in table 1.

The first addition to the sMIPS ISA was an integer multiplier that supported the MIPS instructions for 32-bit multiply and divide. This unit performs 32-cycle operations using a 64-bit multiplier-accumulator register. The appropriate explicit conditions were added to the E stage, which will pause the pipeline in case the software requires data from the multiplier that is not ready. This naive algorithm is not particularly efficient, but allowed us to reuse the hardware for both the multiply and divide instructions, which kept the size of the unit small.

Another important set of instructions that we included were dedicated to partial memory access. 32-bit MIPS processors access memory in chunks of 4 bytes. However, the standard defines ways to access only 1, 2 or 3 bytes. We modified the pipeline and the level 1 data cache to support partial word accesses using byte-level masks. The unaligned access support is a special case of partial word access, where depending on the address from 1 to 4 bytes may be loaded or stored. It is not an unaligned access in the sense of affecting different words or cache lines, because the bytes modified always belong to the same 4-byte word. But these instructions are used often in sequential memory operations where the start address, the end address or both are not aligned to 4-byte boundaries.

2.2 The Coprocessor 0

The MIPS-I ISA defines up to 4 coprocessors, which can be interfaced through standard instructions. Some of these coprocessors are reserved, like the Coprocessor 0 and the Coprocessor 1. The Coprocessor 0 implements many of the functionalities usually managed by the operating system. The Coprocessor 1 implements the floating point arithmetic, but the Bluebox architecture still does not support such kind of hardware and this coprocessor it is disabled (something that is allowed by the standard). Implementing it is left as future work.

In the MIPS R3000 specification, the Coprocessor 0 has 10 32-bit registers. The CPU accesses these registers through the Move To Coprocessor 0 (MTC0) and Move From Coprocessor 0 (MFC0). These registers define the state of the system related to interrupts, exceptions, virtual memory and security level. To implement this important part of the system, the pipeline data path had to be modified to include information about exceptional situations. As shown in figure 2, the F and E stages were extended to handle exceptions and errors. The new F_{ex} and E_{ex} units are executed instead of the original ones when special conditions occur. E_{ex} will be executed when the DEC stage detects an incorrect instruction, a system call instruction is executed, or when an interruption or an exception alters the normal program flow. In these cases, E_{ex} discards the current instruction and instructs the PC unit to jump to the exception service vector address. This special memory region handles interrupts, system calls and exceptions.

The instructions that were already fetched before an exception occurred, and present in the pipeline in the F and DEC stages, are discarded. For this purpose, the sMIPS hardware that processes branch and jump instructions is reused. This mechanism also discards instructions that were fetched before a conditional branch was effectively decided in the E stage. To do so, instructions are marked with *epoch* values. Consecutive instructions have the same *epoch* value. After an instruction flow alteration, the global epoch value is modified and the instructions already fetched and marked with the old value are discarded. In the Bluebox architecture, the *epoch* marks have four possible values, from 0 to 3. This increment was needed because when exceptions occur in a branch delay, two consecutive *epoch* switches are performed. With the original binary values, that would lead to a null effect of the *epoch* mechanism.

2.3 Virtual Memory support

The Coprocessor 0 also contains the Translation Lookaside Buffer (TLB). The TLB is the unit that translates virtual addresses to physical addresses. All the requests to the level 1 instruction and data caches are intercepted by the TLB, and the requested address is translated. The translation is done with an associative memory that matches the virtual address. Each virtual entry is associated to a physical entry. In Bluebox, the TLB contains an arbitrary number of virtual-physical address pairs. Because the 8 first are reserved to the operating system, at least 16 should be used. The MIPS R3000 standard specifies that the TLB should have 64 entries. When a memory access is not found by the TLB, an exception occurs and the instruction flow is altered to jump to the exception service vector. The software is responsible to keep the TLB updated to avoid these exceptions.

From the point of view of the software, there only exists a unified TLB for instructions and data. But in Bluebox, to allow simultaneous accesses from the PC (instruction request) and the E (data request) stages, the TLB has been doubled. These two TLBs contain identical copies of the virtual-physical pair entries.

The address translation is an expensive process in terms of time and resources. This happens because of the Content Addressable Memory (CAM) used to match the virtual addresses. This can stress the hardware path that starts at the E stage, crosses the TLB and ends at the level 1 data cache. To alleviate this problem, in Bluebox the cache entries are Virtually Indexed and Physically Tagged (VIPT). These schema allows to access the caches while the TLB is performing the translation. In the next cycle, the cache line's tag is compared to the physical address translated by the TLB.

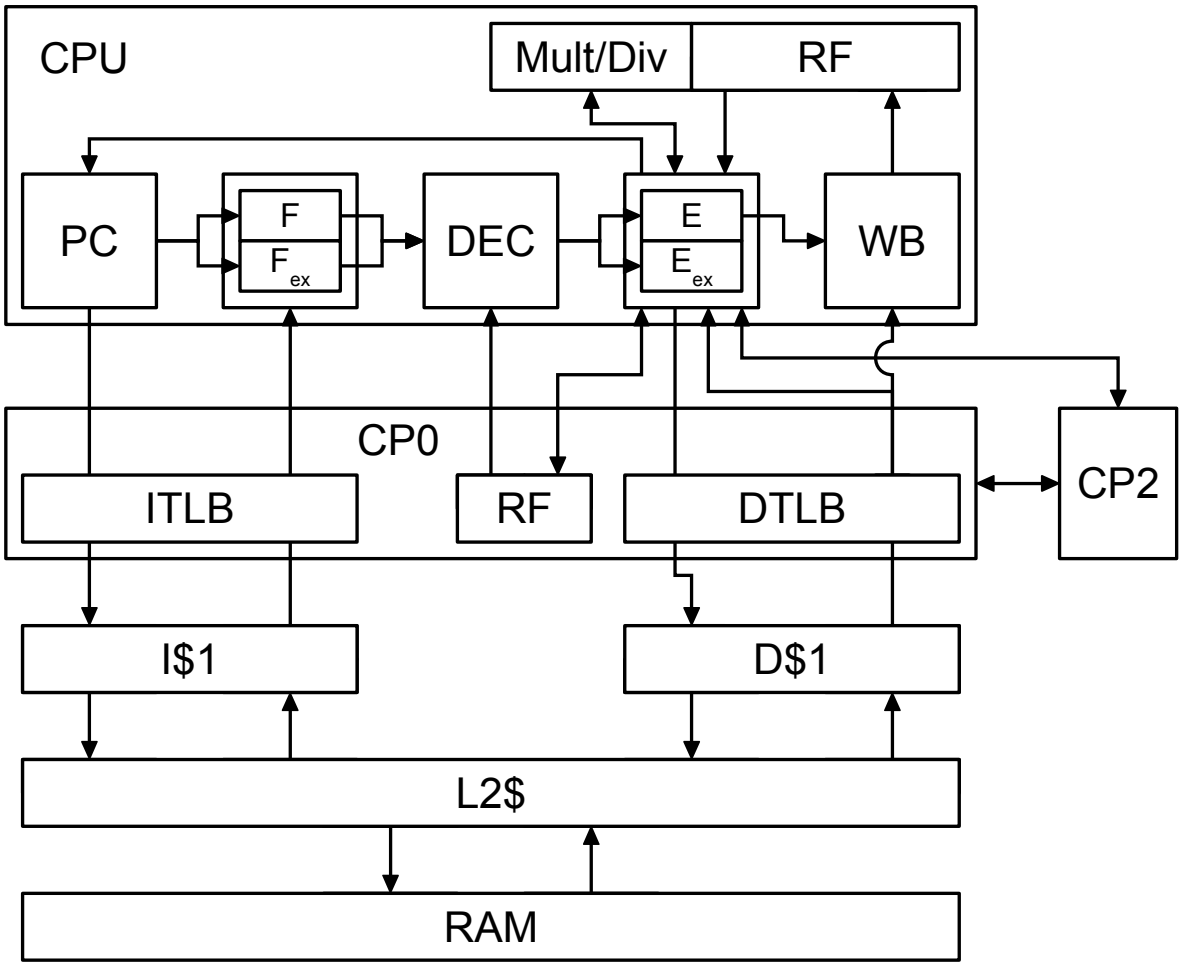

Figure 2: Bluebox basic architecture.

2.4 The memory hierarchy

Bluebox inherits the memory system from a multicore version of sMIPS. Each Bluebox processor has a level 1 instruction cache and a level 1 data cache that serve successful requests in one cycle. The level 2 cache is an inclusive, unified cache. Although this initial version of Bluebox only supports a single processor, the memory system is ready to be connected to an arbitrary number of processors. In the multicore version, the L2 cache acts as a directory that stores all the memory lines present in the system. This could be revised in the future so that it does not become a bottleneck for medium and large systems.

All the caches have a write-back policy. The memory system is completely coherent using a Modified-Shared-Invalid (MSI) scheme. The interconnection is not a shared bus, but a direct connection between the components: CPU-L1-L2. The L2 arbitrates the requests from the L1 caches, and processes them sequentially and in a round-robin fashion.

Parametrizing the hardware models is a simple task in BSV. This is possible thanks to some built-in features like a powerful typing system and the conditional compilation syntax. Thanks to this the Bluebox memory hierarchy can be configured to adapt to any needs. The width of the cache lines can be selected between 8 and 16 bytes, but must be the same in all the system. The size of the L1 caches can be configured independently, which benefits applications that perform data-intensive calculations.

The L1 caches are indexed directly and virtually. The L2 cache is associative and the number of ways can be configured as well. This feature was implemented for practical reasons rather than for optimization purposes: The atomic variable modification mechanism used in Bluebox, through the Load Linked/Store Conditional instructions, sometimes lead to endless-loop executions. This happened because the LL/SC mechanism is based on the physical presence of the cache lines in the local data cache. During a LL/SC transaction, the instruction cache may evict the data linked with LL if both lines are mapped to the same L2 line index. This produces an always-failing SC behavior, and an endless loop of failing atomic updates. With a 2-way associative cache, an instruction line and a data line can coexist with the same L2 line index.

The 32-bit addressing mode of Bluebox can handle 4 GB of data, which imposes the maximum RAM that can be accessed. 4 MB of that RAM are reserved for the boot memory, which contains the first software routine executed by the processor.

2.5 The Coprocessor 2

In MIPS-I there are no specific instructions for I/O and for interfacing peripheral devices. Special memory regions can be defined that act as interfaces for the external hardware. In Bluebox we decided to use the MIPS coprocessor interface so that we did not had to deal with the virtual memory issues of memory-mapped I/O registers. All the non-standard MIPS functionalities were implemented in the Coprocessor 2. This coprocessor can have up to 32 read/write registers, accessed through the MFC2 and MTC2 instructions.

Only some of these registers are used in the current version of Bluebox, and more are allocated as we add new functionalities. Register 1 is read-only and contains static information about the hardware characteristics: Instruction cache size, data cache size, L2 cache size, width of the cache size and number of TLB entries. This information is calculated from the designer's configuration and automatically hardcoded during synthesis. The internal clock is also implemented in this unit. This 64-bit cycle counter can be read from registers 15 (lowest 32 bits) and 16 (highest 32 bits). Register 17 acts as a timer. It causes an interruption when its value is equal to the lowest 32 bits of the internal clock. This interruption is only unset when the timer register is written again. A UART controller can be included in the design. Writing to the register 31 will send a byte through that interface. Other registers are used for debugging purposes only and will be explained in Section 3.

2.6 The Bluebox Linux port

Most of the applications that we plan to use with this prototyping platform have support for Linux or can only run on it. This includes software such as testbenches and academic applications for researchers. To limit the portability issues, one of the main objectives of this project was to run the Linux 2.6 kernel with GLIBC. This way, most of the POSIX applications can be compiled and executed with this state-of-the-art operating system.

Implementing the exact specification of an already-existing MIPS architecture would be impractical for two reasons: It would require a lot of implementation and testing work, and it would limit the possibility to modify the system to perform experiments. For this reason, we decided to implement a new Linux port for our architecture. At this point, the decision of adopting a well-known ISA like MIPS was worthwhile. Little work had to be done when creating a new port for an existing architecture and there is plenty of documentation and literature.

We used a GNU cross-compiler toolchain to develop and compile the MIPS kernel from Intel x86 platforms, following the Cross-Linux From Scratch (CLFS) guide for MIPS [11]. Because patches and tools are provided by the CLFS project, we chose the same version of the Linux kernel used in that guide: 2.6.17.13. The GCC version used is 4.4.1 and the GLIBC version used is 2.4.

The new MIPS port consisted of 1946 new lines of code added to the sources of the Linux kernel. Most of the code was reused from the ports of similar architectures. The hardware abstractions used by Linux were adapted to the interface provided by the Coprocessor 2. This includes the timer mechanism explained before, a serial I/O driver for the UART controller, and a new register used as a buffer to emulate the newer MIPS32 thread-level storage mechanism. The kernel image, without the subsystems and features that Bluebox does not implement, occupies 3 MB including an initial RAM filesystem with Busybox.

3 Debugging extensions

Trying to execute the Linux kernel is a hard test for a hardware model. It is an extensive stress test. However, an operating system is one of the most complex pieces of software that can be executed. One reason is that it uses and manages no trivial hardware mechanisms, like virtual memory and exceptions. Another cause is that it implements advanced techniques to manage the processes and the computer resources efficiently and fast.

Traditional hardware debugging mechanisms consist in test benches, waveform dumps and printf-like commands inserted in the HDL code. These methods are not powerful enough to find errors triggered at such level of software abstractions. During the development of this project we deployed several debugging utilities and techniques that differed from the traditional ones in one or more characteristics. They:

- Can be applied to the same development language or tool used to describe the final hardware model.
- Extend the hardware model unobtrusively, i.e. the behavior is exactly the same as of the original version.
- Do not slowdown the simulation time.
- Are flexible enough for the debugging necessities.

We choose one method that instrumented the hardware simulation while fulfilling the previous conditions. It consisted in interfacing the hardware units with external C++ constructs. This is possible with the Verilog Procedural Interface (VPI), which enables to execute external C code as Verilog's procedures. However, Verilog simulations are significantly slower than BSV [12]. This happens because of the behavioral nature of Verilog. When simulating BSV models, the rule-based, built-in scheduler significantly reduces the amount of calculations done.

Listing 1: BDPI interface examples.

```
1 import "BDPI" function ActionValue#(Bool) debugAction(Bit#(32) addr);
2 import "BDPI" function Bit#(32) getGPReg(Rindx indx);
3 import "BDPI" function Action setGPReg(Rindx indx, Bit#(32) val);
```

Listing 2: C prototypes for functions defined with BDPI.

```
1 char debugAction(unsigned int);
2 unsigned int getGPReg(char);
3 void setGPReg(char, unsigned int);
```

3.1 The BSV C interface

The BSV language also offers the possibility to interface C and C++ external code through the so-called BDPI interface. Using this high-level language, we could increase the productivity. At the same time, the C++ extensions allowed us to implement flexible and powerful debugging mechanisms.

The methods defined through the VPI and BDPI interfaces can only be used during simulations on a host computer. These mechanisms should not be confused with debugging techniques that can be used during the "real" usage of the hardware in an ASIC chip or an FPGA device. The C++ extensions interfaced by BSV are only included when compiling a simulation environment of the hardware model. Under this configuration, the BSV compiler generates a set of C++ source files that are compiled into an executable binary object. This object can be run with bluesim, the BSV's simulator. When the designer decides to generate a synthesizable RTL model, the compiler generates a set of Verilog source files. BDPI cannot be used under these circumstances.

External C++ methods are declared in BSV as functions (do not modify the state of the hardware) or actions (do modify the state of the hardware). In listing 1 three examples are shown. The `debugAction` method, which has a 32-bit parameter, modifies the state of the computer and returns a boolean result. The `getGPReg` method, which returns a 32-bit value without affecting the state of the computer. And the `setGPReg` method, that receives two parameters and modifies the state of the computer, without returning any value.

The BSV compiler generates C prototypes for the functions defined with BDPI. Listing 2 shows the C headers of the functions defined before. During the generation of the simulation model, the compiler will expect these functions to be defined by the designer.

3.2 The Bluebox extension points

If only information could be transferred from both programming languages, the C++ extensions could only serve as a monitoring tool. The interesting point of this approach is the fact that during the simulation, when a C++ extension is called, the computational control of the simulator is transferred to the external code. As it is a single-threaded simulator, the simulator literally waits for the external program to return the control, i.e. returning from the called function. This allows us to monitor and pause/resume arbitrary units of the BSV model.

Using this methodology, it is possible for an external program to obtain selective information from the hardware behavior and to pause the simulation at certain clock ticks. Basically, this is the definition of a software debugger. If the C++ extensions do not modify the state of the hardware model, only the simulation flow is affected, while the execution will be as accurate as the original model.

To implement a software debugger that can control the Bluebox CPU, we instrumented 7 hardware units. Figure 3 shows the interfaces added to the hardware and what units are affected. 6 of them are only for monitoring purposes. Only one affects the simulation flow and is used to pause the execution. This

Listing 3: The E stage instrumented with a BDPI call.

```
1 rule exec ( /* E stage conditions... */ );
2   let fm <- debugAction(dQ.first.pc);
3   // E stage body...
4 endrule
```

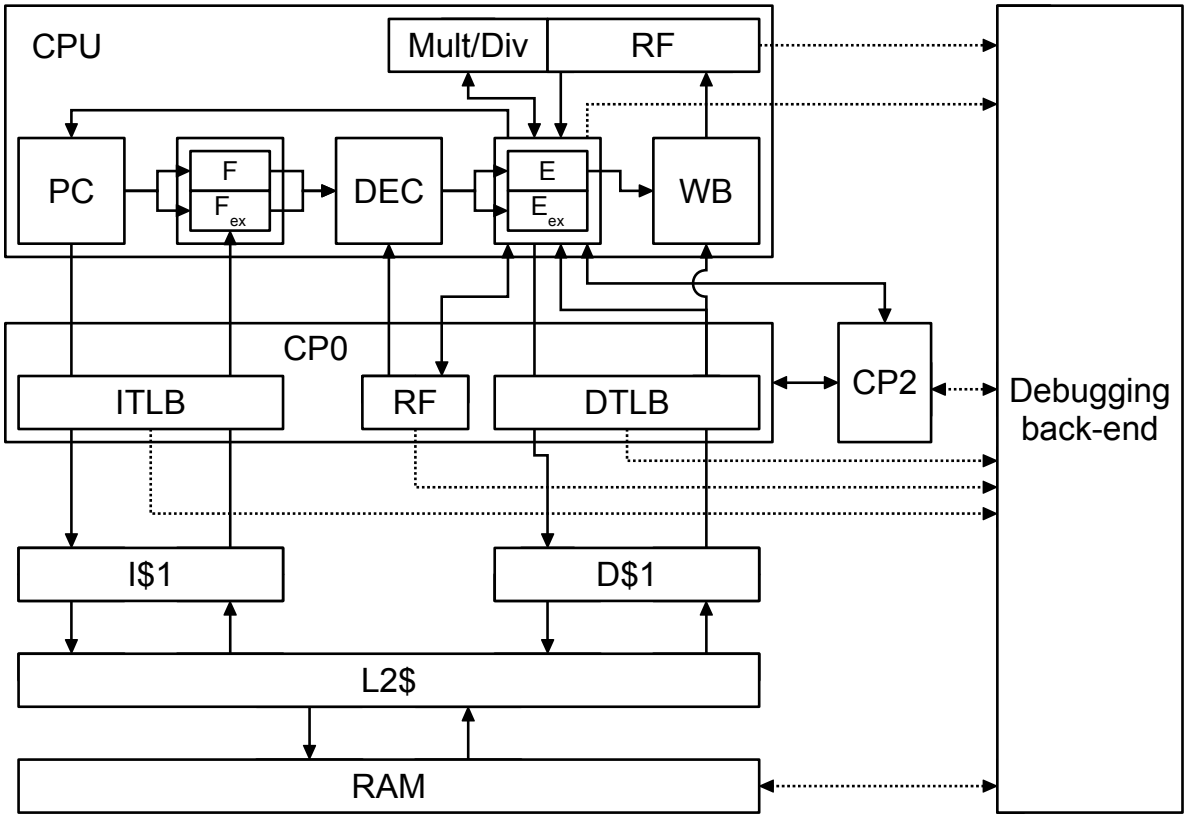

Figure 3: Bluebox architecture with debugging extensions.

extension was added to the E unit, as shown in listing 3. This allowed us to detect what instructions arrived to the execution stage, and to pause the CPU before executing them. This C++ external method is called when the E stage is active, but usually it returns the control as fast as possible for not degrading the performance of the simulation. Under certain events, the external software can pause the simulation and wait for an user event. Different conditions can be implemented, depending on the current instruction address (stepping, breakpoints), if a special situation was detected (exceptions, watchpoints), or due to external causes (the user paused the simulation).

The monitoring interfaces are activated when a relevant part of the computer's state has been modified: the CPU's register file, the CP0's register file, the TLB and the RAM. This helps the simulator to keep updated information. This information can be used to visualize the state of the computer (e.g., instruction disassembly, register file, memory dumps, etc.). But it is important to take decisions such as when to pause the simulation flow as well.

3.3 Migrating the state of the system

Most of the state of the computer remains in the BSV model. Thanks to the C++ extensions, the external software can monitor the changes on this state. But BDPI imposes one limitation: The external

software cannot access the internal state of the BSV model. This makes impossible for an external program to, for instance, modify the contents of the CPU's register file. Or, for instance, preloading hardware memories externally.

For simplicity, when simulating the Bluebox system we preload the bootloader into an internal memory. The Linux kernel's image is loaded "magically" to the RAM memory as well. When synthesized as a real hardware, the latter must be done by the boot code. However, the BSV preloading mechanism, based on the Verilog memory format, highly restricts the input methods. MIPS software must be translated to a raw ASCII format in order to be preloaded to a BSV memory.

For this reason, we decided to migrate the RAM to the external debugger. Instead of monitoring the changes to the memory, now the BSV model peeks into the RAM contents hosted by the C++ software, and informs it if a write operation must be done. If done carefully, migrating the state of the computer should not affect the accuracy of the simulation. This is possible because the external C++ methods are instantaneous. The only condition is that the contract of the implemented functions respects the original hardware that has been migrated.

With the RAM migrated, we improved the memory preloading with new methods. Not only raw data could be loaded, but now also binary formats are accepted. We implemented an ELF interpreter that could preload unmodified binary images, like the Linux kernel image as generated by the GCC toolchain. Our extension finds the instruction and data segments from the binary object and loads them to the corresponding addresses. This improvement further integrated the debugging work flow stages, and greatly simplified the simulation process. This extension also allowed us to import additional metadata from the executables, such as the symbol table and extra debugging information like the DWARF code. This information was a valuable help in conjunction with other tools, as explained in Section 5.

3.4 Advanced tracing

Thanks to the debugging extensions, most of the problems in our design related to functional errors could be solved. But as we advanced more in the process of loading the Linux kernel, more complex problems appeared. In particular, towards the end of the boot process the kernel starts the first software thread, the `init` task. This process performs an important initialization work that implies an intensive multithreading effort. In addition, the MIPS virtual memory is used for the first time to prepare the initial filesystem before switching to user space. Most of the bugs that reside in the exception handling and memory management hardware will emerge at this point.

In multithreaded environments, where the operating system fails *too* and the hardware is unreliable, simple instruction disassembly and debugger stepping will not be effective enough for the user. Moreover, the operating system abstractions that fail are too complex to deduce the original hardware cause. Software collaboration is required to better understand what is happening during the simulation.

We implemented additional hardware interfaces that the operating system could use. This way, we extended the monitoring process to the software layer. We added a new register to the Coprocessor 2, which the software could use to inform about thread scheduling changes: Every time the Linux scheduler switched the active task, the new Process ID (PID) was sent through the new register with a single instruction. This method affected slightly the execution, but the changes were minimal.

Additional information was gathered automatically and unobtrusively. The hardware also registered when the exceptions happened, at what address and for which cause. The branch instructions where instrumented to obtain information about what software functions where called. All this information together describes the behavior of the kernel's execution threads. Thanks to the ELF symbol table section, the software can identify the original C function or data structure of any memory address accessed by the CPU. Figure 4 shows the debugging extensions, its submodules and how it communicates with the BSV model.

We used the Paraver visualization environment [13] to graphically display all the information. This tool was created to visualize huge amounts of data from supercomputer traces. After each debugging session, our debugging extension stores all the CPU's context switches, the causes of the exceptions, the security

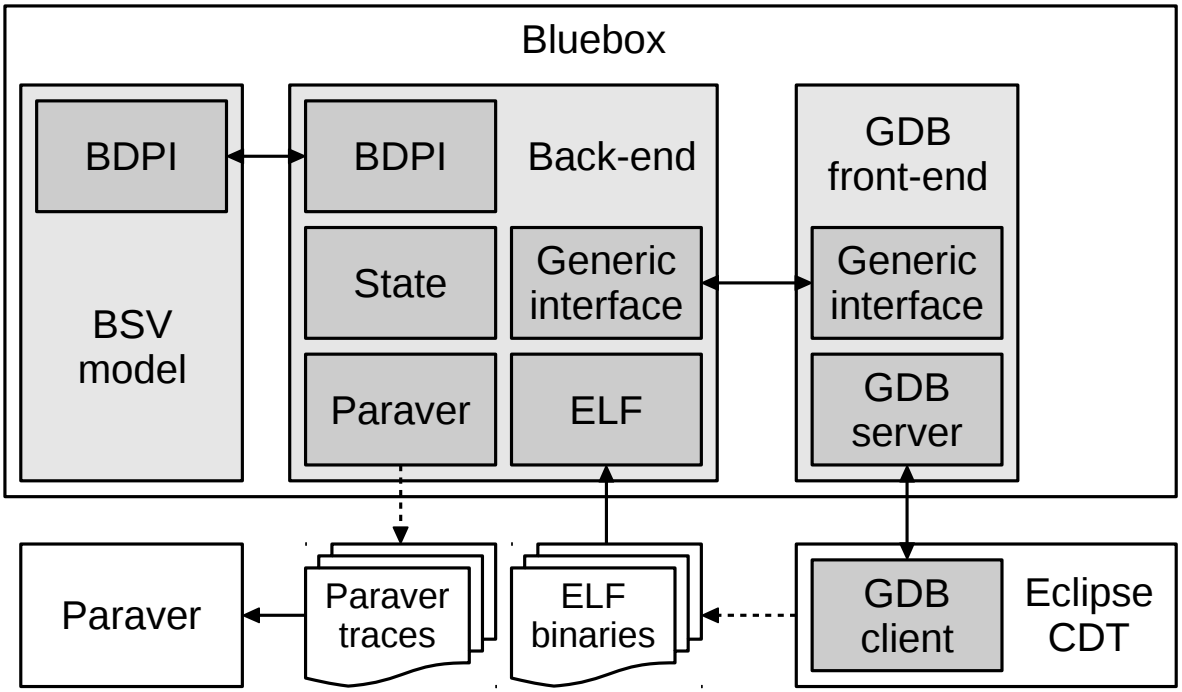

Figure 4: Bluebox back-end with a GDB front-end controlled by Eclipse.

level (user or system mode) and the PID of the running Linux process, and the traces of all the software functions accessed during the execution flow. Figure 5 shows how Paraver can be used to visualize the Linux processes created during the execution of the init program. We explored in [14] the usefulness of collecting execution traces for complex multithreaded applications. Such graphical hints are a valuable help for the designer, however in that work we realized that tracing is much more effective at early stages of the development process.

4 Checkpointing support

Monitoring in real time the hardware model with custom software helped to find unexpected bugs in our hardware model. However, the Bluespec simulator does not support reverse debugging (i.e. undoing the execution flow). When a symptom of failure appears, it is impossible to step back to its cause. This reduces the debugging experience in a trial and error process, which leads to long simulation times.

To alleviate this situation, we decided to implement a checkpointing mechanism. We call checkpoints to take snapshots of the system's state. If this state can be loaded to the system, the execution should continue from the moment where the checkpoint was stored. This can avoid having to simulate the simulation process that produced that state, and thus it saves a valuable time for the designer. GDB and other state-of-the-art debuggers implement this technique to pause and restart debugging sessions.

As explained in the previous Section 3.3, the external software cannot modify the state of the BSV hardware. To have direct access to the state of the computer, it must be migrated to the external software. Previously, only the RAM contents were migrated to be able to preload custom data. To load and store snapshots of the system's state, all the information must be migrated.

In the original version, the external debugger replicated the state and the hardware notified any changes to maintain the coherency. The checkpointing-enabled version of Bluebox holds as little state information as possible. The state is not replicated, but remote, and the hardware peeks at it and requests updates through the BDPI interface. The experiments did not show a significant slowdown of the simulator's performance using this technique.

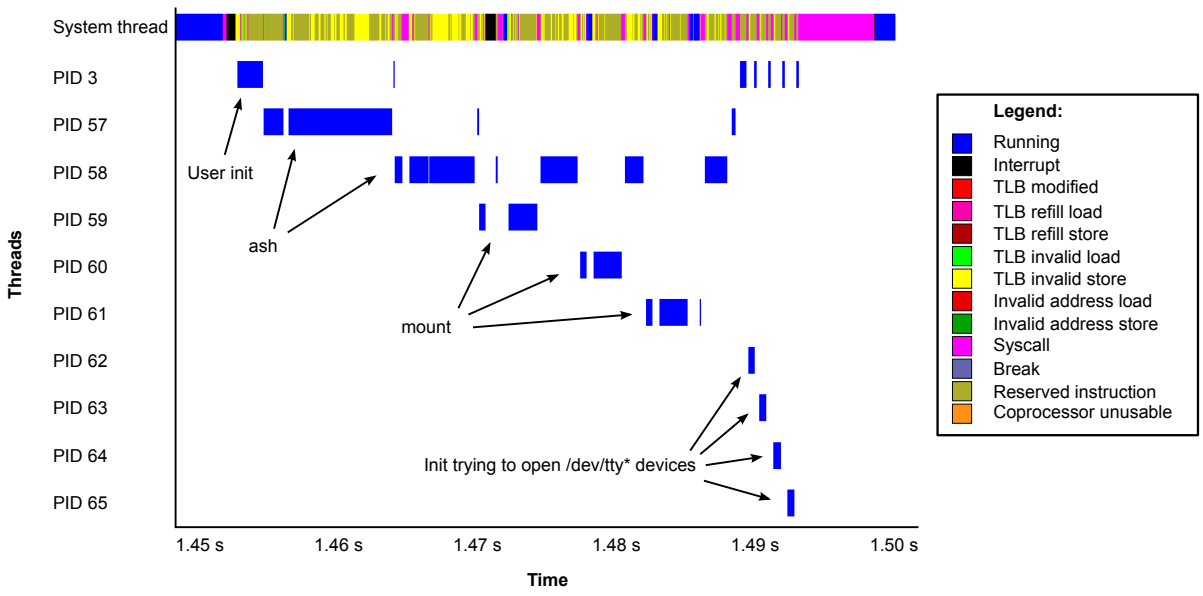

Figure 5: Paraver trace showing the Linux kernel launching the `init` process. The Y axis indicates the process ID (PID). The first row is the original kernel process, which launches the other threads and receives the system calls. The X axis displays the execution time in seconds. Colored rectangles indicate that the corresponding process is active during that period of time. The selected processes and time range correspond to the final stage of the Linux boot process, when the kernel creates threads in user space to launch the Busybox shell application.

The checkpointing technique produces optimal results if the whole state of the computer can be obtained. This includes not only the hardware that is accessible to the software, but also units opaque to the software, like the pipeline or the caches. Saving all this information would imply migrating the contents of FIFOs, registers and other BSV building blocks to the external debugger. This process is time-consuming, and would require modifying large portions of the Bluebox source code.

If only significant parts of the state are migrated to the external software, the resumed simulation flow will not be exact but it can be correct from the point of view of the software functionality. We call that subset of the computer's state a "functional partition". When resuming from a functional partition, the non-saved part of the state must be reconstructed or defined in a way that the simulation flow will produce similar results. Table 2 shows how we partitioned the state.

In the debugging version of Bluebox, most of the functional partition was monitored by the software, and only the RAM was migrated. In the checkpointing version, all the functional partition has been migrated to the software. The contents of the multiplication unit are monitored in one version and have been migrated in the other, because they are not relevant when debugging the processor, but must be stored when saving a checkpoint.

The contents of the pipeline (instructions in the FIFOs between stages) and the memory caches are not monitored nor migrated to software. To guarantee correct execution, when saving a checkpoint some actions must be taken:

- Flushing the pipeline: Pipeline stages PC, F, DEC and E remain paused, but WB finishes any pending work. This guarantees that instructions previous to the checkpoint have been executed correctly and their effects have not been lost.
- Flushing the memory caches: This guarantees that the RAM stored in the checkpoint is coherent with the data modified by the CPU.

Table 2: Computer state distribution.

State holder	Debugging only version	Checkpointing version
State in hardware (not monitored)	MULT, Pipeline, CP2 RF, I$1, D$1, L2$	Pipeline, I$1, D$1, L2$
State in hardware (monitored)	PC, GP RF, CP0 RF, TLB	
State in software	RAM	PC, GP RF, CP0 RF, MULT, CP2 RF, TLB, RAM

Note: PC is the address of the instruction in the E stage; GP RF is the general purpose register file; CP0 RF is the Coprocessor 0 register file; Pipeline is the state of the CPU pipeline; CP2 RF is the Coprocessor 2 register file; MULT are the contents of the HI and LO registers; TLB are the contents of the address translation entries; I$1, D$1 and L2$ are the contents of the cache lines; and RAM are the contents of the memory.

Listing 4: Source code of the BSV rule that manages the cache flushing.

```
1  rule flush_caches ( !wbQ.notEmpty && flushMode &&
2       dataReqRdyW && flushAddr < l2lines );
3    dataReq <= DataReq { op:FlL, addr: flushAddr, data:?
4       `ifdef USE_CPU_BYTE_ENABLE
5       , we:?
6       `endif
7    };
8    flushAddr <= flushAddr + fromInteger(valueof(TDiv#(CacheLineSz,8)));
9  endrule
```

To flush the cached lines some modifications had to be done to the CPU and the memory caches (small compared to migrating their contents to the external software). After finishing the last instruction, the WB stage enters a special state where it sends a `flush` message for all the L2 cache lines. The source code of this special BSV rule is shown in listing 4. The level 1 data cache bypasses these messages to the level 2 cache. After receiving a flush message, the L2 cache writes back to the main memory any modified lines. As the L2 cache is inclusive, any modified data stored in the L1 caches will be transferred to the main memory too.

The saved state of the CPU consists in a set of 32-bit and 64-bit registers. This information can be stored in 460 bytes. To optimize the size of the RAM contents in the checkpoint's data, only the modified memory pages are stored. We implemented a special k-tree data structure that handles all the touched pages. The nodes of that tree are tuples of the form (address, size). When the RAM is written, a new node (address, page size) is added. Consecutive nodes are merged. In our experiments, after booting the Linux kernel the modified memory can be represented with a single node of 20 to 30 MB. The overhead of maintaining the memory tree is negligible because RAM writes are issued infrequently (when the L2 cache evicts a line).

When resuming from a checkpoint, the system will suffer some variations in performance and simulation flow respect the original session while the pipeline and the caches are filled up again.

5 Debugger front-end interfaces

We decoupled the external software in two parts. The back-end has been presented in Section 3. It receives the BDPI calls from BSV and manages the state migrated outside from the hardware model. It also includes the ELF loader, the Paraver trace generator and the checkpoint manager. We defined a generic interface that allows controlling the back-end from a front-end, so that different debugging

Table 3: GDB commands implemented by the Bluebox debugger.

Command	Description
c	Continue running.
s	Single stepping.
m	Read the memory.
g, p	Read the general purpose registers.
Z, z	Insert or remove a breakpoint.
D, k	Stop the simulation.
?, H, q, v	Other commands.

applications can be built depending on the designer's needs. Through this interface, the back-end offers some additional services like virtual address translation and access to ELF's DWARF information.

The simplest front-end is a program that displays debugging console information on a shell terminal, much like a typical GDB shell. This application shows the contents of the general purpose registers and the Coprocessor 0 registers. In each stage, it reads the opcodes around the currently executed instruction. An disassembled trace of that memory is displayed. The user can execute execution flow commands (stepping or running), memory inspection commands (dumping a range of virtually or physically addressed memory) or storing to disk a checkpoint. When the simulator is running freely, the user can pause the execution at any moment pressing the Control+C keyboard sequence.

We implemented another debugging console, similar to the previous one, but based on the terminal library ncurses. This alternative version displays the information in colored text, and refreshes the terminal window instead of printing and scrolling down.

5.1 Debugging Bluebox with GDB

The third front-end that we implemented greatly differs from the previous ones. This program acts as a glue code for the state-of-the-art GDB debugger. When run on a system, GDB can take control of a target process and allow the user to debug it. But, in our case, a GDB instance should be running in parallel to Bluebox on the x86 host. To do that GDB implements a remote debugging protocol. In the remote system, a GDB server applies the requests from the GDB debugger. **It must be clear that we are not debugging programs running on Bluebox, but we are debugging the Bluebox system itself.**

In our debugger, the front-end implements some GDB stubs that are converted into requests for the Bluebox back-end. Table 3 lists the remote GDB commands that our server supports. We heavily multithread the code in order to communicate simultaneously with the back-end and GDB (which implies a lot of blocking system calls).

The Bluebox GDB server can be controlled from a GDB instance through a TCP/IP port. A GDB instance can be used to control that server, but other applications that understand the GDB remote protocol can be used too. We configured the Eclipse CDT (a C/C++ integrated development environment [15]) to connect its integrated debugger to our remote GDB server. In figure 4 the three main parts of the Bluebox system are shown (BSV hardware model, back-end and front-end) with their submodules.

Eclipse is a good platform to edit and cross-compile C/C++ sources like the Linux kernel and Busybox. We hosted their sources and we configured the tool to cross-compile the software to MIPS ELF binaries. Later on, we preload those binaries to Bluebox and control the simulation from Eclipse too. With this design flow, the development and debugging of the software is done using the Eclipse integrated environment. This allowed us to perform advanced debugging techniques like line-by-line debugging for C code and variable and stack inspection. Moreover, after the simulation we could use Paraver to do *post-mortem* behavior analysis. And all these methods could be applied to complex software like the Linux kernel.

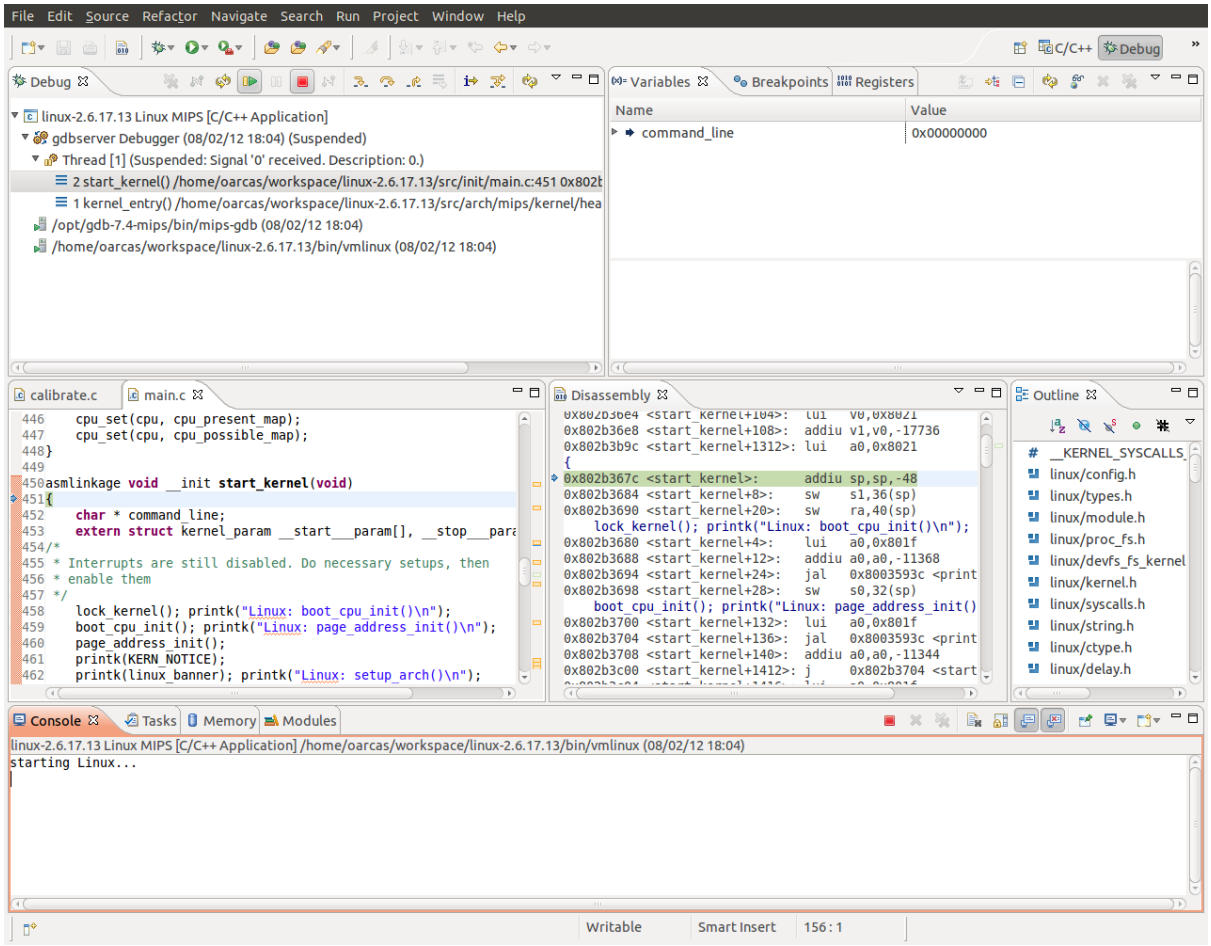

Figure 6: Screenshot of Eclipse connected to Bluebox while debugging the Linux kernel.

6 Conclusions and related work

6.1 Related work

In recent computer architecture research that makes use of FPGAs, the hardware model is usually backed up with a functional software description of the same architecture. This implies the necessity of propagating any changes done in one design to the other. For example the correct functionality of RAMP Gold, a very efficient multi-core prototype, is verified by running it first on the C-Gold simulator, prior to trying it on the FPGA [16]. Later, for developing RTL code, they iterate between simulating it on Modelsim and testing it on the actual hardware. In this paper, we propose a methodology that enables using the same architecture definition both for simulation and for the actual hardware. This also removes the need for maintaining multiple versions.

The authors of Protoflex created a debugging package in BSV, which allowed insertion of high-level print statements, which could be inserted into any point in action methods or rules, that provides "the power of C-like debugging in an RTL environment" [17]. Our mechanism that interfaces a GDB debugger is a more powerful tool with many more functionalities.

For online debugging, the RAMP Blue model makes use of a hard PowerPC core to control the execution and for debugging through a console channel [18]. Similarly, the Arete system uses a Microblaze core for for debugging, pausing the execution and reading registers [19]. Another approach as the authors of RDL (RAMP design language) suggest, is the use of plugins to insert debugging logic and connect it to a software front end [20].

Our infrastructure does not deal with the online debugging case yet, however, this method is complementary to ours: As long as we can stall the pipeline (the execute stage) and read the hardware state, it does not matter if we are simulating or online debugging. In either case, we hope for finding all bugs and errors in simulation phase (offline) through our hardware model that shows high fidelity and our debugging and visualization extensions.

Most of state-of-the-art simulators include checkpointing mechanisms. ModelSim [21] can save and restore the current state of the RTL model to save simulation time. But RTL behavioral simulations are slow compared to BSV simulations or software simulators. GDB supports "bookmarks" for certain architectures, which store the state of the application being debugged. However, this operation only affects the software, not the state of the whole system. The gem5 architectural simulator [22] supports checkpointing, but for some configurations the contents of the memory caches must be flushed to the main memory as in Bluebox. Our system is the first to implement a checkpointing technique with a BSV hardware model.

6.2 Future work

The memory hierarchy is ready for a multiprocessor system: The L2 cache can interface an arbitrary number of cores, and the coherency system already includes synchronization mechanisms like Load Linked/Store Conditional. We plan to enable this feature and obtain a synthesizable Chip-Multiprocessor (CMP) system. The Linux kernel port for Bluebox will have to be modified to enable the multiprocessor support.

The debugger extensions, trace visualization and checkpointing support of Bluebox helped us to have a full Linux kernel. But discovering the hardware errors from software symptoms is still an annoying reverse-engineering task. We want to include reverse debugging support to the Bluebox simulator. This technique permits stepping back during the simulation, which is the easiest way for the designer to catch bugs. One way to implement reverse debugging requires periodical checkpointing, which is already possible in our system.

We strongly believe that the if errors are caught during the development and simulation stage, the productivity increases. But even when using accurate simulators unexpected results can appear in the final technology. We want to port some of the debugging support to the final hardware. Because of the

rule-based nature of BSV, it should be easy to implement hardware execution flow control mechanisms, like we did with C++ extensions. A built-in debug unit could control the Bluebox pipeline, receiving commands from an on-chip soft processor, e.g., a Microblaze processor. This debugging CPU could translate requests from a GDB instance running on a host computer.

6.3 Conclusions

In this work, we have presented the Bluebox a hardware model of a MIPS RISC processor. This system runs the well-known Linux kernel 2.6, and standard tools like GCC and GLIBC can be used to compile applications to run on it. We proposed a design methodology where the stages are integrated with unified tools and languages. This environment allowed us to reuse the work done during the debugging, while the verified model can be synthesized at any stage of the process.

We used the high-level, functional hardware description language BSV to develop this model. Thanks to the level of abstraction provided by BSV, we could achieve high productivity, while the simulation still exhibits high fidelity. Using the BDPI interface, we have been able to implement debugging extensions to the hardware model. This avoided to implement a separated simulation and debugging environment using different tools or languages.

The C++ extensions included advanced techniques such as parsing and loading ELF binaries and their debugging information, advanced tracing using the Paraver tool, checkpointing to save and restore the simulation sessions, and controlling the hardware model from the state-of-the-art GDB debugger and the Eclipse integrated development environment.

References

[1] Mentor Graphics Catapult C, 2012. http://www.mentor.com/esl/catapult/overview.

[2] Impulse C, 2012. http://www.impulseaccelerated.com.

[3] Xilinx Vivado Design Suite, 2012. http://www.xilinx.com/products/design-tools/vivado/.

[4] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: hardware design in Haskell. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 174–184, New York, NY, 1998. ACM.

[5] Bluespec Inc., 2012. http://www.bluespec.com.

[6] Nehir Sonmez, Oriol Arcas, Gokhan Sayilar, Osman S. Unsal, Adrián Cristal, Ibrahim Hur, Satnam Singh, and Mateo Valero. From plasma to beefarm: Design experience of an fpga-based multicore prototype. In *7th International Symposium on Applied Reconfigurable Computing (ARC 2011)*, Belfast, UK, 2011. Elsevier.

[7] Nehir Sonmez, Oriol Arcas, Otto Pflucker, Osman S. Unsal, Adrián Cristal, Ibrahim Hur, Satnam Singh, and Mateo Valero. Tmbox: A flexible and reconfigurable 16-core hybrid transactional memory system. In *19th IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM 2011)*, Salt Lake City, US, 2011. IEEE.

[8] OpenRISC 1200 processor, 2012. http://opencores.org/openrisc,or1200.

[9] Microblaze Soft Processor Core, 2012. http://www.xilinx.com/tools/microblaze.htm.

[10] sMIPS Processor Specification, 2005. http://csg.csail.mit.edu/6.884/handouts/labs/smips-spec.pdf.

[11] Jim Gifford and Ryan Oliver. Cross Linux From Scratch version 1.0.0-MIPS, 2005. http://www.cross-lfs.org.

[12] Derek Chiou, Dam Sunwoo, Nikhil Patil, Joonsoo Kim, Bill Reinhart, Hari Angepat, and D. Eric Johnson. Lessons from Implementing a FAST Prototype. In *3rd Workshop on Architectural Research Prototyping (WARP'08)*, Beijing, China, 2008. ACM.

[13] V. Pillet, J. Labarta, T. Cortes, and S. Girona. PARAVER: A Tool to Visualize and Analyze Parallel code. In *WoTUG-18*, pages 17–31, Manchester, UK, 1995.

[14] Oriol Arcas, Philipp Kirchhofer, Nehir Sonmez, Martin Schindewolf, Osman S. Unsal, Wolfgang Karl, and AdriŽn Cristal. A low-overhead profiling and visualization framework for hybrid transactional memory. *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, 0:1–8, 2012.

[15] Eclipse C/C++ Development Tooling, 2012. http://www.eclipse.org/cdt/.

[16] Zhangxi Tan, Andrew Waterman, Rimas Avizienis, Yunsup Lee, Henry Cook, David Patterson, and Krste Asanović. RAMP gold: An FPGA-based architecture simulator for multiprocessors. In *DAC '10*, pages 463 – 468, 2010.

[17] Eric S. Chung, Eriko Nurvitadhi, James C. Hoe, Babak Falsafi, and Ken Mai. A complexity-effective architecture for accelerating full-system multiprocessor simulations using FPGAs. In *FPGA '08*, pages 77–86, 2008.

[18] Alex Krasnov, Andrew Schultz, John Wawrzynek, Greg Gibeling, and Pierre yves Droz. RAMP blue: a message-passing manycore system in FPGAs. In *FPL 2007*, pages 27–29, 2007.

[19] Asif Khan, Muralidaran Vijayaraghavan, Silas Boyd-Wickizer, and Arvind. Fast and Cycle-Accurate Modeling of a Multicore Processor. In *Proc. ISPASS 2012)*, pages 1–8, April 2012.

[20] Greg Gibeling. RDLC2: The RAMP Model, Compiler & Description Language. Master's thesis, UC Berkeley, 2008.

[21] ModelSim - Advanced Simulation and Debugging, 2012. http://model.com/.

[22] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.