LEG Processor for Education

Maxwell Waugaman, Zakkai Davidson, Samuel Dietrich, Daniel Johnson,
Cassandra Meyer, Eric Storm, Avi Thaker, Ivan Wong
Harvey Mudd College
Claremont, California 91711–3116
Email: mwaugaman@hmc.edu

***Abstract*—Open core processors allow students to explore digital design through experimentation and practice. The LEG processor described in this paper has a simple design that makes it accessible to students. The LEG processor is compatible with the ARMv5 instruction set and includes a memory management unit that allows it to boot Linux 3.19 in simulation, a combination not found in other open core processors known to the group. In addition, LEG includes a testing suite that allows students to quickly verify modifications and extensions to the processor. This paper describes LEG's microarchitecture, testing framework, and potential use in an educational or classroom setting.**

I. Introduction

Over the course of two years, a group of undergraduate students developed LEG, an open core processor written in SystemVerilog and available at https://github.com/MWaug/LEG.

The LEG processor boots Linux, and supports mode switching, interrupts, I/O, and virtual address translation, a feature the group has not found in other open source cores compatible with an ARM instruction set. LEG is compatible with ARMv5 [1] excluding the Thumb extension. In addition, the testing framework used to verify the performance and functionality of LEG is especially valuable for education. New features that may be added as parts of assignments (such as branch prediction) can be easily validated.

The LEG processor was divided into three major pieces: the datapath, memory system, and testing framework. For 10 hours a week during school semesters, three students primarily worked on the datapath, one student worked on the memory system, and two students worked on the testing framework. The undergraduates involved in this project learned about memory architecture, datapath subsystems, exception handling, interrupts, and verification.

This project demonstrates the potential for processor design as a tool for education, as discussed previously in [2], [3], [4]. LEG was inspired by a previous MIPS design project [5] and reflects the shift toward ARM architecture in industry and in introductory texts on digital engineering [6]. The LEG processor gives students the opportunity to learn about digital engineering through processor design.

II. Microarchitecture

The LEG processor's datapath shown in Figure 1 uses the five stage pipeline from Patterson and Hennesey [6], similar to the ARM9E [7]. This five stage design and simple microarchitecture help make the processor accessible to students. A separate controller module directs the flow of data through the datapath and a hazard unit, not shown in Figure 1, controls the stall and flush signals of the pipeline registers.

LEG uses a 3-port register file, L1 instruction and data caches, a memory management unit (MMU), and a translation lookaside buffer (TLB). The processor was designed with the minimum feature set to boot Linux.

A. Micro Operations

To minimize hardware resource requirements for projects that may involve chip layout or FPGA synthesis, the datapath has no hardware multiplier and an ordinary 3-port 32 entry x 32 bit register file which allows the processor to read two registers and write one register per clock cycle. However, some instructions require reading more than two registers or writing more than one register. These instructions are broken down into an equivalent sequence of simpler ARM instructions, known as micro-operations (micro-ops). Table I lists the instructions that are handled by these micro-ops.

The micro-op controller is a Mealy finite state machine that reads instructions in the decode stage and inserts new instructions into the pipeline. The fetch stage is stalled for one cycle less than the number of operations, since the first operation takes the place of the initial instruction. The register file has 31 general purpose registers, and one scratch register. To avoid clobbering general purpose registers, the scratch register is reserved for intermediate results in the micro-op state machine. The sequence of instructions for the most complex load instruction, shown in Figure 2, uses this "scratch register" Rz.

B. Memory

The memory system includes L1 instruction and data caches, a TLB, and corresponding address translation and fault detection hardware shown in Figure 3. The bus connecting the peripherals, translation hardware, and caches is compatible with the AHB-Lite protocol [8], but does not implement burst mode, protection control, or HRESP. LEG is the only open core the group has found that is both compatible with an ARM instruction set and supports virtual address translation.

Other open core processors compatible with ARM architectures such as Storm [9], ARM4U [10], and Amber [11] do not support virtual memory, required for standard versions

978-1-4673-8584-8/16/$31.00

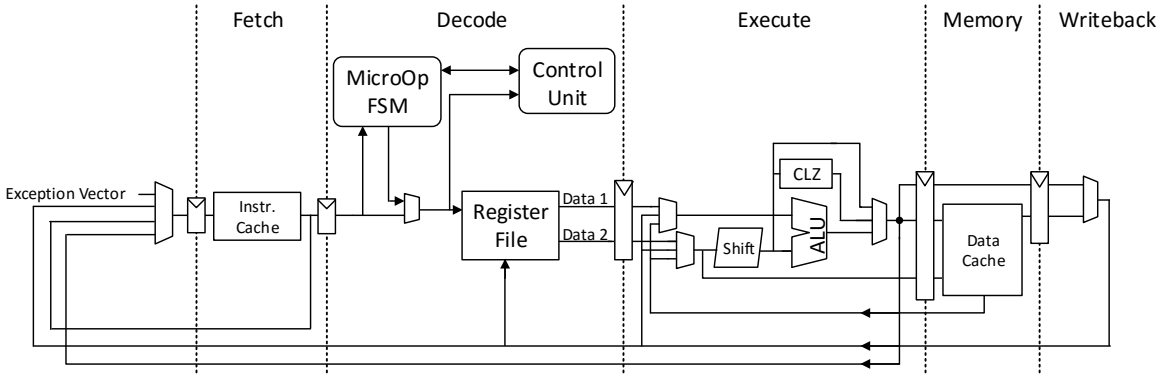

Fig. 1. Datapath excluding hazard logic. The datapath consists of five stages with a 3-port register file in the decode stage where micro-ops are inserted. Performance is improved by forwarding values to the execute stage from the memory and writeback stages.

Original: `LDR R3, [R5], -R3`

Micro-ops:
```
SUB Rz, R5, R3
LDR R3, [R5]
MOV R5, Rz
```

Fig. 2. An example of micro-ops executed to implement a complex load operation. The memory addressed by R5 is loaded into R3, then the address is updated by subtracting the original value of R3. This is achieved by saving the updated address in the scratch register, Rz, and moving the saved value into R5 after the load is completed.

TABLE I
NUMBER OF MICRO-OPERATIONS CORRESPONDING TO EACH COMPLEX INSTRUCTION

Instruction	Operations
Exception	1
LDRH / STRH	1-2
LDR / LDRB / STR / STRB	1-3
Register shifted register	2
BL / BLX	2
SWP / SWPB	3
LDM / STM	2-18
MUL / MLA	98-100
UMULL / UMLAL	133-166

of Linux. Processors such as the OpenSPARC [12] and the LEON3 [13] contain the same feature set, but are not compatible with an ARM instruction set and are too complex for most undergraduate educational uses. LEG's caches are physically tagged and virtually indexed, limiting the maximum cache size to 1kB per way, the smallest virtual memory page size. This memory system can demonstrate address translation and caching behavior to students.

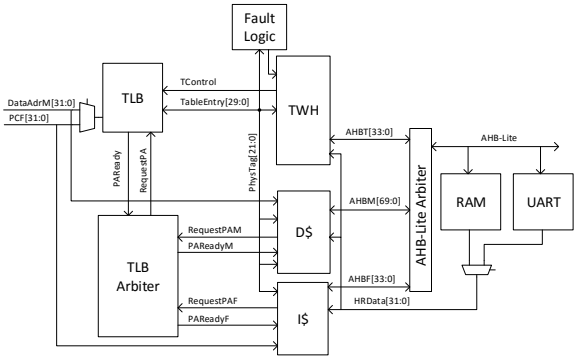

Fig. 3. Memory system including physically tagged caches, address translation hardware, and bus. The translation walk hardware (TWH), data and instruction cache all desire access to the bus (right). The AHB-Arbiter arbitrates bus control among those three modules. Instead of using two TLBS, one for each cache, one (top left) is shared among the instruction cache and data cache to conserve space. Similar to the AHB-Arbiter, the TLB arbiter mediates control over the TLB request and ready signals.

III. VERIFICATION

To verify the operation of the processor, a series of directed and randomized tests were designed to maximize the number of tested combinations of instructions. Tests can be written in C or assembly and compiled with gcc [14]. The processor was also tested with a practical and complex task: booting Linux. A minimal installation of the Linux 3.19 kernel was compiled for ARMv5 using Busybox in a RAM-based temporary filesystem to provide userspace utilities. The processor was verified by completing the Linux boot process in simulation.

A verification framework was designed for the LEG processor to facilitate rapid functionality debugging and performance profiling of the processor design. The framework, which will be released along with the processor and complete documentation, consists of three main parts:

- **QEMU** [15] is an open source machine emulator capable of simulating a variety of systems. This serves as a

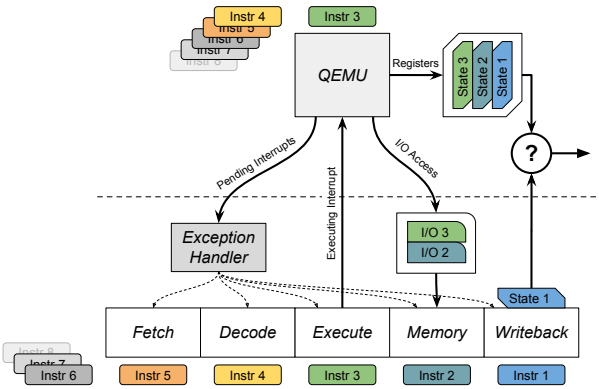

Fig. 4. Overview of the relationship between QEMU and the processor simulation. QEMU executes instructions, sends interrupt information to the processor, stores register states in the state queue (top right) and I/O in the I/O queue (middle, above "Memory"). The processor pipeline (bottom) executes a set of instructions. Components below the dashed line represent parts of the LEG processor, while those above the line are separate parts of the framework. Queues are surrounded by white boxes, and information is colored according to the instruction it corresponds to.

reference implementation and runs tests in lockstep with the processor. Note that the version of QEMU used in the testing framework has been modified to allow access to the internal emulated state.

- **GDB** [16] is a popular open source debugger that can interface with QEMU and is extended with a series of processor debugging commands.
- **ModelSim** is an hardware description language (HDL) simulation environment that simulates the processor under test and communicates with GDB using a Tcl script.

The testing framework is designed to run programs in lockstep on both QEMU and the simulated processor. Instructions are executed one at a time and the results are compared after each instruction to ensure that both implementations behave identically. This allows students to pinpoint the locations of any errors in the design. As shown in Figure 4, one difficulty in performing lockstep testing is that QEMU executes instructions one by one, but the LEG processor is pipelined. This is addressed by having QEMU execute instructions in sync with the processor's execute stage. Each time a new instruction passes through the execute stage, QEMU steps by one instruction and stores its registers in a state queue. Once the instruction reaches the writeback stage, the corresponding reference state is popped off the queue and compared to the LEG processor's current registers.

A. I/O Peripherals And Interrupts

I/O peripherals and interrupts are the main non-deterministic parts of processor execution. As such, they are the most difficult to test. Booting Linux involves many of these events, and the testing framework is designed to seamlessly maintain consistent state in the presence of such non-deterministic input.

To keep the state consistent between QEMU and the processor, the test framework uses QEMU-driven I/O. Thus QEMU is responsible for all I/O and interrupt signals.

The testing framework ensures that any non-deterministic action occurs in QEMU first and that the results are mirrored in the processor. When an instruction reads from a memory-mapped peripheral, the address and value read by QEMU are stored in the I/O queue, shown in Figure 4. This occurs while the instruction is in the execute stage of the processor. Once the instruction moves to the memory stage, Modelsim pops the corresponding I/O access off the I/O queue and returns the same value to the processor. This enables consistency across I/O accesses.

Interrupts are slightly more complex. The processor must delay multiple cycles before executing the interrupt to make sure that pending instructions have cleared the pipeline. By default, QEMU will always execute an interrupt immediately after it is raised. In this testing framework, interrupts are delayed in QEMU until the processor determines the interrupt should be triggered.

When QEMU receives an interrupt, the interrupt signals are forwarded to the LEG processor and QEMU is prevented from executing the interrupt immediately. Once the processor begins executing the interrupt, QEMU executes the interrupt as well. This ensures that QEMU and the processor are in the same state after the interrupt occurs.

B. Using the Verification Framework

Running tests is a simple process. The testing environment is initialized by running a shell script, which then starts a version of GDB extended with testing commands, connects to a new instance of QEMU, and loads a specified test program.

Within the testing environment, a variety of commands are available. The `lockstep` command starts a processor simulation in ModelSim, copies the current state of QEMU into the processor, and then executes instructions one by one until there is a mismatch. At this point, it displays a dump of the current state of QEMU and the processor and returns control to the user, who can further investigate the bug in GDB.

Many additional commands are available to assist in debugging. Some of the most useful are:

- `lockstep-auto`: Start lockstep, but whenever a mismatch is found, log it to a file. Then, copy over QEMU's state into the processor to correct the error, and continue executing. This allows multiple bugs to be found in one continuous run.
- `checkpoint`: Start a processor simulation and copy over QEMU's state, but do not begin lockstep. Instead, save a ModelSim checkpoint for manual debugging.
- `frombug`: Fast-forward QEMU to the state immediately before a particular bug was found. This enables rapid debugging of individual arithmetic instructions.

Lockstep testing of the simulated processor can verify approximately 14 instructions per second running on a single Intel Xeon E7540 CPU at 2.00GHz. This allows a 5000-instruction program to be completely tested in 6 minutes. Dozens of randomly generated assembly programs of this scale were

used during LEG development to rapidly profile and debug functionality for every type of instruction.

Additionally, a Divide-and-Conquer mode allows parallel debugging of a large executable, such as the Linux boot process, by splitting the execution into multiple stages and running lockstep over each simultaneously. This makes it feasible to test complex programs in instruction-by-instruction lockstep. The Linux boot process was tested in lockstep using the Divide-and-Conquer mode on a machine with 48 cores in less than twelve hours. Large executable debugging thus takes too long for small to moderately sized class design projects, but is feasible for larger projects that may attempt to optimize the performance of LEG for more complex programs.

IV. Size and Performance

The size and performance of LEG were estimated using synthesis and the Dhrystone 2.1 benchmark [17]. Synopsys's Design Compiler [18] was used to synthesize the entire chip excluding the memories, and Cadence Encounter [19] was used to automatically place and route the chip on a 0.6 μm process. This synthesized chip covers an area of 57 million λ^2 ($5.1\,mm^2$) and has a total of about 30,000 gates plus 4.1kB of memory, compared to approximately 75,000 gates in the ARM9E core [20]. The caches include 4kB of memory total (2kB for each) and the 16 entry TLB contains a 16 x 22 bit block of content addressable memory and a 16 x 31 bit RAM for an additional 848 bits of memory (106B). Note the size of the caches and TLB are parameterized, but the maximum size of the L1 caches, 1kB per way, is set by the minimum virtual page size. The processor achieved 0.36 DMIPS/MHz running Dhrystone in simulation without branch prediction and with caches enabled. Arbitration among the caches for TLB access causes a single cache stall at every simultaneous data and instruction cache access. This stall significantly reduces the resulting DMIPS/MHz number and leaves an opportunity to improve the processor by creating a second TLB.

V. Suggested Projects

The LEG processor provides many interesting opportunities for exploration. These include:

- **Branch Prediction**: Currently, branch prediction is not implemented on the processor. The fetch stage is stalled during branches until it is known whether the branch will be taken. Implementing this feature would greatly improve the performance of the processor.
- **Add Thumb**: Thumb instructions could be added to the LEG processor to make the processor compatible with the full ARMv5T specification.
- **FPGA Implementation**: LEG could be implemented on an FPGA to teach system design and test performance.
- **TLB Replacement Policy**: Least recently used (LRU) or other replacement policies could be implemented on the TLB. Performance metrics including TLB misses and cycles per instruction would likely improve with a better replacement policy. Currently, a subset of the virtual address determines replacement.
- **Memory Exploration**: Students can explore how cache sizing and replacement policy affects performance in Linux.
- **Component Replacement**: Students can learn by replacing a removed component of the processor or fixing a broken module to understand how the processor works. The testing framework facilitates this type of exploration, because it helps quickly verify the processor design and helps students debug.

The instruction cache is a candidate for a component replacement project. One student designed the instruction cache which included determining its interface, creating the module, and testing basic operation. This design process reinforced the concepts of virtual memory and locality, and exposed the student to more complex digital systems. Students could replace LEG's instruction cache with one they design then vary the number of ways and entries to determine the impact of caching on a program such as the Dhrystone benchmark.

In addition to the processor itself, the testing framework may be leveraged as a tool for education. For example, one could use lockstep in conjunction with an RTL simulation tool to show students how an instruction in GDB propagates through the LEG processor's pipeline. This extension is possible because the testing framework tracks instructions in each stage of the processor. The testbed could be similarly extended to teach I/O and interrupt handling.

VI. Conclusion

The LEG processor strikes a balance of simplicity and capability that enables its use in education. The microarchitecture is relatively simple and the processor without memories synthesizes to an area of 57 million λ^2. Despite this simplicity, the processor is capable of booting Linux and includes virtual memory, a feature the group has not found in other open cores compatible with an ARM instruction set. In addition, the testing framework developed for the processor quickly verifies modifications and creates unique learning opportunities that result from the lockstep design. We are making the processor open source because there exists potential for LEG to aid in education and we would like to share our learning experience designing LEG with other undergraduates.

References

[1] M. G. Morrow. (2016, Feb.) ECE 353 Introduction to Microprocessor Systems. ddi0100e_arm_arm.pdf. [Online]. Available: http://morrow.ece.wisc.edu/ECE353/arm_reference

[2] D. Jansen and B. Dusch, "Every Student Makes His Own Microprocessor," in *10th European Workshop on Microelectronics Education (EWME)*, May 2014, pp. 97–101.

[3] A. Chidanandan, J. Mellor, and L. Merkle, "Design and Implementation of a Minuscule General Purpose Processor in an Undergraduate Computer Architecture Course," in *Microelectronic Systems Education, 2007. MSE '07. IEEE International Conference on*, June 2007, pp. 43–44.

[4] A. Kim, G. Weistroffer, D. Grammer, and R. Klenke, "The VCU SRC II: a full-custom VLSI 32-bit RISC processor," in *University/Government/Industry Microelectronics Symposium, 2001. Proceedings of the Fourteenth Biennial*, 2001, pp. 201–204.

[5] N. Pinckney, T. Barr, M. Dayringer, M. McKnett, N. Jiang, C. Nygaard, D. Money Harris, J. Stanley, and B. Phillips, "A MIPS R2000 Implementation," in *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, June 2008, pp. 102–107.

[6] D. Patterson and J. Hennessy, *Computer Organization and Design: The Hardware/software Interface*. Morgan Kaufmann, 2013.

[7] (2016, Mar.) 1.1.1.The instruction pipelines. [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0240b/ch01s01s01.html

[8] M. Brehob. (2016, Feb.) EECS 373: Design of Microprocessor Based Systems. ARM_IHI0033A_AMBA_AHB-Lite_SPEC.pdf. [Online]. Available: http://www.eecs.umich.edu/courses/eecs373/readings/

[9] S. Nolting. (2016, Feb.) Storm Core. [Online]. Available: http://opencores.org/project,storm_core

[10] J. Masur. (2016, Feb.) ARM4U. [Online]. Available: https://github.com/freecores/arm4u

[11] C. Santifort. (2016, Feb.) Amber. [Online]. Available: http://opencores.org/project,amber

[12] Oracle. (2016, Feb.) OpenSPARC Overview. [Online]. Available: http://www.oracle.com/technetwork/systems/opensparc/index.html

[13] Cobham. (2016, Feb.) LEON3 Processor. [Online]. Available: http://www.gaisler.com/index.php/products/processors/leon3

[14] G. Pfeifer. (2016, Feb.) GCC, The GNU Compiler Collection. [Online]. Available: https://gcc.gnu.org/

[15] F. Bellard. (2016, Feb.) QEMU Open Source Processor Emulator. [Online]. Available: http://wiki.qemu.org/

[16] P. Alves. (2016, Feb.) GDB: The GNU Project Debugger. [Online]. Available: https://www.gnu.org/software/gdb/

[17] Dhrystone Benchmark. [Online]. Available: http://www.netlib.org/benchmark/dhry-c

[18] Synopsis Design Compiler Version E-2010.12-SP5. [Online]. Available: http://www.synopsys.com/Tools/Implementation/RTLSynthesis/DesignCompiler/

[19] Cadence Encounter Version 10.12. [Online]. Available: http://www.cadence.com/

[20] (2016, Mar.) What is the gate count figures for ARM7EJ-S? [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.faqs/ka3824.html