

# Harnessing FPGAs for Computer Architecture Education

Mark Holland, James Harris, Scott Hauck

Department of Electrical Engineering  
University of Washington, Seattle, WA 98195, USA  
mholland@ee.washington.edu, harrisjs@u.washington.edu, hauck@ee.washington.edu

**Abstract.** Computer architecture and design is often taught by having students use software to design and simulate individual pieces of a computer processor. We are working on a method that will take this classwork beyond software simulation into actual hardware implementation. Students will be able to design, implement, and run a single-cycle MIPS processor on an FPGA. This paper presents the first steps in this work: an FPGA-optimized MIPS processor, a debugging tool which provides complete control and observability of the processor, and the reduction of the MIPS instruction set into eight instructions that will be used by the processor.

## 1 Introduction

Computer Design and Organization is a common upper-level engineering course that is offered at universities throughout the world. In this class students learn computer design by implementing individual pieces of a computer processor. This approach has important limitations: while students can complete and simulate their designs using software, they do not get the chance to implement and run their designs. Studies of engineering curriculums have shown that students learn better when given active, hands-on projects [1], [2], suggesting that students would thrive on the opportunity to implement and run their processors in hardware.

This paper presents a Field Programmable Gate Array (FPGA) implementation of a computer processor, a processor debugging tool, and the reduction of the MIPS instruction set into eight instructions. The FPGA-optimized processor can be used to enhance computer architecture education by programming selected CPU parts onto the processor, giving the incomplete processors to students, and allowing the students to design and integrate the missing pieces. The processor debugging tool would provide the students with complete control over their processor, allowing them to debug and fix their incomplete processors. Reducing the MIPS instruction set to eight instructions allows us to use a simpler implementation of the processor. Combined in the classroom, these tools would allow students to implement and run their processor designs on an actual chip.

FPGAs are logic chips that can be programmed an indefinite number of times. The area of an FPGA is devoted largely to reprogrammable logic, but vendors have recently added sizable memories to their chips in order to give them a wider range of

capabilities. In Xilinx Virtex devices [3] this memory is called Block Ram (BRAM) [4]. This BRAM provides for shallow RAM structures to be implemented on the FPGA.

The reprogrammable nature of FPGAs makes them perfect for educational purposes because they can be reused year after year, resulting in low overhead costs. In addition, the reprogrammable nature of FPGAs allows computer architecture students to have as many iterations as necessary when implementing their processors.

Our work is modeled on the single-cycle implementation of the MIPS processor, which is shown in Figure 1. We also have a pipelined implementation of the MIPS processor, but most of the paper will focus on the single-cycle implementation as it will be our primary instructional tool. Both processors operate on the MIPS instruction set as described in *Computer Organization and Design* by David A. Patterson and John L. Hennessy [5]. This book is widely used in the teaching of computer architecture. Using it means that the FPGA-optimized processor presented here will be easily incorporated into a large number of classrooms.

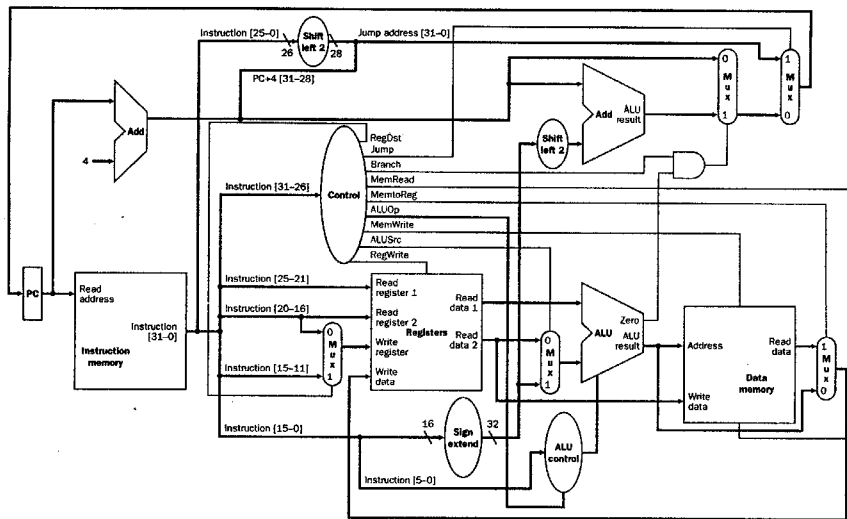


Fig. 1. Single Cycle MIPS Processor [5]

It is important to note that we are not the first group to place a processor on an FPGA: companies such as Xilinx [6], Altera [7], [8], and Gray Research [9] have all succeeded in doing so. Their work, however, has been aimed at obtaining high performance processors or efficient mapping strategies for processors on FPGAs. Our design goals are quite different as we are trying to use FPGA-optimized CPUs as tools for teaching computer architecture to students. The focus of the class must be on CPU design, not on FPGAs.

Our unique design goals did not allow us to use any of the preexisting FPGA-optimized CPUs. One major drawback of existing implementations is that the processors are invariably tied to the FPGAs they are implemented on: to teach CPU design using any of them would require almost as much background in FPGAs as is would in processor design. We want to keep the focus away from the FPGAs; they should simply be tools for learning CPU design. As such, we wanted to create a simple CPU that would require almost no FPGA background.

Other design requirements also led us to create our own CPUs. In the classroom, our processors will need to be easily dissected and understood. Also, since our students will be putting their processors together one part at a time, we needed a processor that can be externally tested and debugged, one which can be tested for correct operation even when incomplete. Considering these requirements, we determined that the best course of action was to design the processor from scratch, allowing us to build in parallel a debugging tool that would be able to control and debug the processor.

The FPGA-optimized processor and debugging tool are described in Sections 2 and 3 respectively. Section 4 discusses the reduction of MIPS into eight instructions. In Section 5 we discuss the integration of our work into the classroom. Section 6 offers results and discussion of the implementation, and Section 7 gives conclusions. Section 8 provides a description of future work that could be done to improve the educational effectiveness of the work presented in this paper, and lastly, Section 9 provides additional notes.

## 2 FPGA Implementation of the Processor

The most important task in our work was to implement a single-cycle processor onto an FPGA so that it would look and behave like the processor presented in Hennessey and Patterson. We chose to do our implementation on a XESS-XSV board [10], shown in Figure 2. The board components used in this implementation are as follows: one XILINX Virtex XCV300 FPGA [3], two independent 512K x 16 SRAM banks [11], one parallel port interface [12] that will allow communication between a PC and the FPGA, and one push-button switch. The Virtex XCV300 FPGA contains 322,970 system gates and 65,535 BRAM bits.

The FPGA is used for nearly all of the control and datapath of the processor. The only exceptions are the data and instruction memories, which due to a lack of space on the FPGA, are implemented in SRAM. The parallel port is used for loading configurations to the FPGA as well as communicating with the PC, which is necessary for the debugging tool. The push-button is used to reset the processor.

While many of the smaller aspects of the processor are trivial to design and implement on the FPGA<sup>1</sup>, considerable effort was needed to implement the processor's memories. This is because FPGAs are primarily designed to handle logic, not memories, and are

---

<sup>1</sup> The ALUs, shifters, control logic, sign extenders, and multiplexors can all be described in very few lines of verilog code.

therefore fabricated with a very small amount of memory space. Thus the register file, instruction memory, and data memory must be carefully implemented so that the chip's resources are not overused.

The register file is a 32-bit, 32-address memory that has two read ports and one write port. This means that during any clock cycle, the processor must be able to read values from two different memory addresses as well as write a value to a third address. In order to avoid taxing the standard logic resources of the FPGA, we implemented the register file using Block Ram (BRAM). A single BRAM can have data widths of no greater than 16 bits and has a maximum of two ports; so multiple BRAMs were needed in order to implement the 32-bit, three-ported register file.

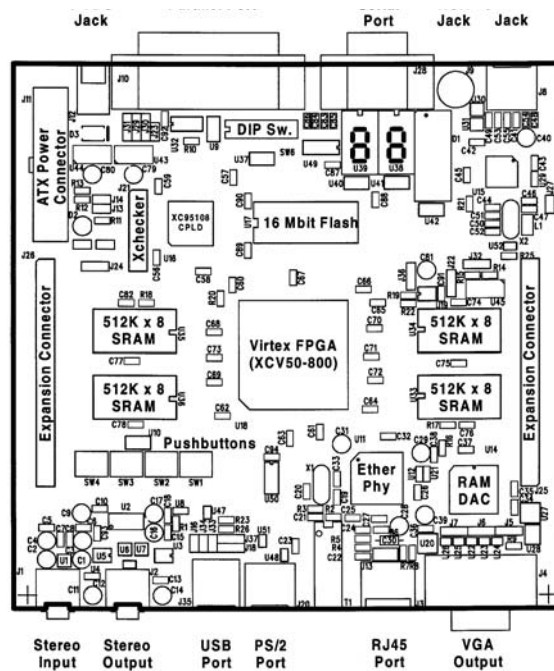
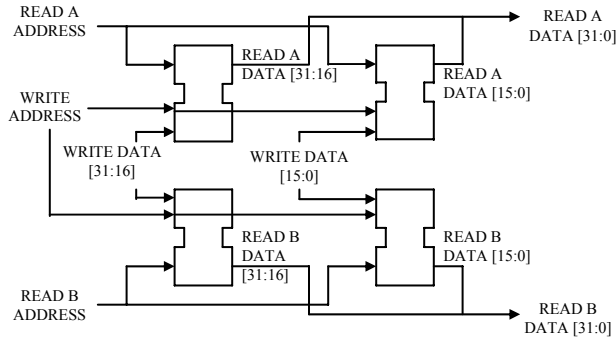


Fig. 2. The XSX Board Schematic [10]

Figure 3 shows the implementation of the register file using four dual-ported 16-bit BRAMs. These four 16-bit BRAMs functionally combine to create two 32-bit BRAMs, which is what the processor requires. It is necessary that writes be performed on both of these 32-bit BRAMs so that a read from either of the BRAMs will return the proper value. One read port can then be assigned to each of the 32-bit BRAMs, providing us with a 32-bit, 32-address register file complete with two read ports and one write port.

By itself, the FPGA does not have enough storage area to hold the instruction and data memories, so we chose to put these memories into off-chip SRAM. The instruction and data memories need to be large enough to hold a complete program in them, and the 2K addresses that are available in the FPGA's BRAM are insufficient. Combining the two SRAM banks results in 512K addresses, which is sufficiently large for the purposes of our instructional processor.



**Fig. 3.** Register File Implementation

Our goal was to present the processor exactly as it is shown in *Patterson and Hennessy*, which suggests that we should have implemented the memories as two distinct units. Using one bank of off-chip SRAM for the memories therefore presented a problem: how would we make one memory appear to be two memories? Also, since we were now using resources that are not on the FPGA, how would we hide the complicated communications between the FPGA and the SRAM so that the students would be unaware that it is occurring?

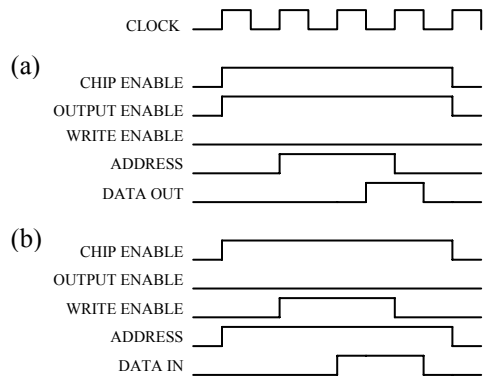
In the normal operation of the processor, it is common that the instruction memory and the data memory would both be accessed during the same clock cycle. Since both of these memories are implemented in the same SRAM and cannot be accessed simultaneously, this requires that the processor clock be divided into multiple clock cycles: one period for reading instructions, period for reading data, and one period for storing data.

In addition, any memory reference requires that the FPGA send multiple signals to the SRAM at specific times in order to instigate a read or a write to memory. The read and write waveforms for the SRAM, as implemented in our processor, are shown in Figure 4. These waveforms can each be produced using 4 clock cycles, with signals being changed on the rising edges of the clocks.

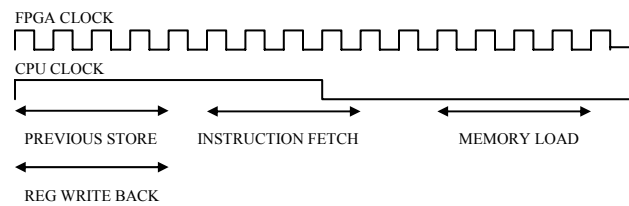
Each communication with the SRAM requires a minimum of 4 clock cycles itself, which means that the memory now requires 12 clock cycles per CPU clock cycle in order to behave like two memories. To accommodate this, we use a clock of 16 times the frequency of the processor to control the instruction and data memory

communications, supplying extra time where combinational delays exist. The 16-cycle sub-clock is hidden from the students so that the memory will look and behave like the idealized data and instruction memories from *Patterson and Hennessy*.

The overall timing diagram for our processor, including clocks and operations, is shown in Figure 5. In the single-cycle implementation the clock is always started prior to the current instruction fetch, is run for a certain number of cycles, and then stopped after a memory load. In this way the clock can be run for 0 cycles, which would load an instruction and perform a memory load but would not commit any values to memory, allowing only combinational effects to occur. Another thing to notice is that the memory store and register write backs are timed so that they are edge triggered with respect to the CPU clock. This is another example of how we designed the processor so that it would behave just as in described in *Patterson and Hennessy*.



**Fig. 4. (a) Read, (b) Write Waveforms for the SRAM**



**Fig. 5. Processor Timing Diagram**

The pipelined implementation of the processor had to use a slightly different clocking scheme than the single-cycle implementation. Using the single-cycle clocking scheme for the pipelined processor, a conflict occurs when a specific register is being written to and read from during the same clock cycle. The fact that the clock is always started

before the instruction fetch means that the read would occur before the write, resulting in the wrong value being read from the register. In the pipelined processor, therefore, the write back must occur before the read occurs. This is accomplished by always starting the pipelined clock at the write back, allowing the proper value to be read from the same register during the same clock cycle.

### 3 The Debugging Tool

The debugging tool is necessary for making our FPGA-optimized processor usable in the classroom. The tool allows the students to observe and control the internal states of their processors, a necessity for the designing and debugging processes.

We chose to create the debugging tool using Visual Basic. Communications between the PC and the FPGA are carried out over the parallel port. The parallel port provides a total of twelve signals that can be seen by both the PC and the FPGA. Eight of these signals (bits) are changeable only by the PC and four of them only by the FPGA. Both the PC and the FPGA can thus put bits onto the parallel port that can be seen by the other device.

Of these parallel port bits, we use one bit in each direction as an acknowledge signal. When the PC wants the FPGA to accept new data it changes the value of its acknowledge bit, which the FPGA recognizes as a prompt to read the other seven PC-controlled bits. When the FPGA has successfully read the bits, it changes the value of its own acknowledge bit, which tells the PC that the transaction was successful. This leaves seven actual data bits to be controlled by the PC, and three data bits to be controlled by the FPGA.

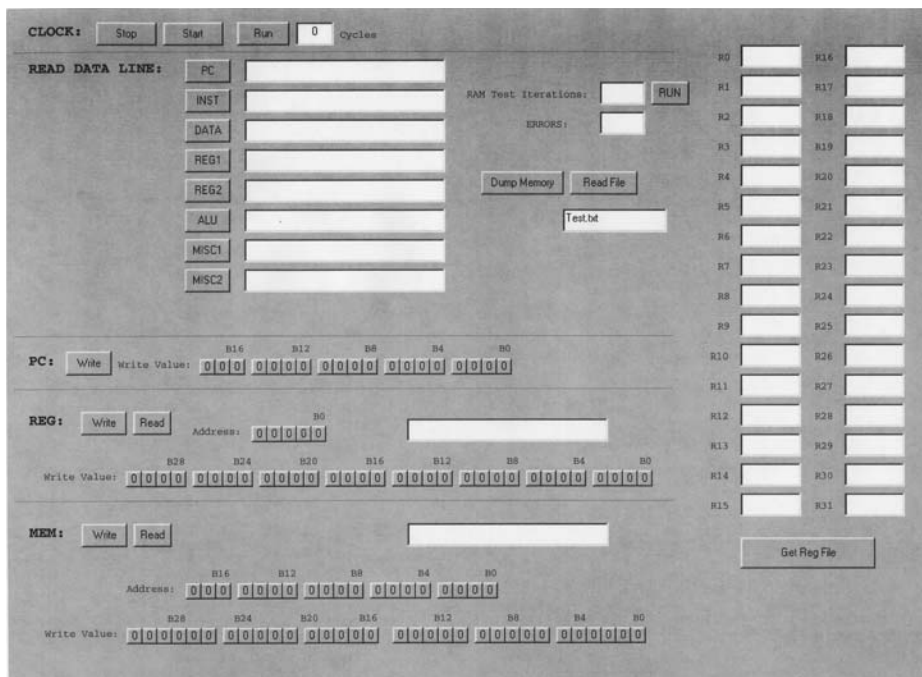
The PC initiates all communications used by the debugging tool. All transaction consists of the PC giving the FPGA a 64-bit command, the FPGA carrying out the command, and the FPGA returning 32 bits. The list of commands, along with send and receive fields, is shown in Table 1. The communication protocol currently operates at only 300 bytes/second, but work is being done to increase this rate.

The interface of the debugging tool is shown in Figure 6. The tasks that it can perform can be broken into 5 basic categories: clock manipulation, reading of data lines, writing the program counter, reading or writing the register file, and reading or writing the instruction or data memory. These tasks provide the user with control of all vital parts of the processor as well as complete observability of the processor's internal state.

We supply two methods for controlling the clock: one which runs the clock for a specified number of cycles and one which runs the clock until the user gives a stop signal. The student can read data lines simply by selecting one of the provided lines: PC, INST, DATA (data memory output), REG1, REG2, ALU, MISC1, or MISC2. MISC1 and MISC2 are lines that the student can hook up to any part of the processor, providing added debugging flexibility. Students can write to the program counter, register file and memories by supplying the address and data value to be written, with the exception of the program counter which does not require an address. They can then perform reads of the register file and memory simply by supplying the read address.

**Table 1.** Debugger Commands and OP Fields

OPERATION	SENT FIELDS	RETURNED FIELD
READ PROGRAM COUNTER	-	PROGRAM COUNTER
READ INSTRUCTION	-	INSTRUCTION
READ DATA MEMORY OUTPUT	-	MEMORY OUTPUT
READ REGISTER 1 OUTPUT	-	REGISTER 1 OUTPUT
READ REGISTER 2 OUTPUT	-	REGISTER 2 OUTPUT
READ ALU OUTPUT	-	ALU OUTPUT
READ LINE MISCELLANEOUS 1	-	MISC 1 OUTPUT
READ LINE MISCELLANEOUS 2	-	MISC 2 OUTPUT
RUN CLOCK	# OF CYCLES	FINAL PC VALUE
START CLOCK	-	INITIAL PC VALUE
STOP CLOCK	-	FINAL PC VALUE
WRITE PROGRAM COUNTER	WRITE VALUE	NEW PC VALUE
WRITE REGISTER	WRITE ADDRESS & VALUE	NEW REG VALUE
WRITE MEMORY	WRITE ADDRESS & VALUE	NEW MEM VALUE
READ FROM REGISTER	READ ADDRESS	REGISTER VALUE
READ FROM MEMORY	READ ADDRESS	MEMORY VALUE



**Fig. 6.** The Debugging Interface



## 4 Reduction of MIPS Instruction Set

We determined that it would not be ideal to implement a processor that ran the complete MIPS instruction set: it would be too time consuming and too taxing on our FPGA resources while providing negligible educational value to the students. At the same time we want our processor to be able to load and run programs, which would seem to require the ability to execute any instruction. To accommodate these goals we searched for a small set of instructions that could be used to describe the complete set, but that would allow us to retain the flavor of a real microprocessor.

At the far end of the spectrum, it is possible to create a “one instruction computer” using only one instruction to build the rest of the instruction set [13]. Another option we considered was using an 8-bit implementation rather than a 32-bit set. Both of these options, however, would oversimplify the processor to a point where it no longer behaved like a standard processor. Since the processor will be used solely for educational purposes, we wanted to make sure that it would retain enough characteristics of a standard processor that it would be useful in the classroom.

Other methods of reducing the instruction set proved themselves too inefficient. For example, we could have pre-loaded our processor with all  $2^{32}$  possible 32-bit values: a negation could then have been accomplished simply by loading the proper value from memory. The memory overhead from a scheme like this is ridiculously large, however, which makes it impractical to use.

In the end, we chose to reduce MIPS to the eight instructions shown in Table 2. All of these are standard instructions from the MIPS instruction set.

**Table 2.** Reduced MIPS Instruction Set

INSTRUCTION / FORMAT	OPERATION
NOR rd, rs, rt (Nor)	perform bit-wise logical NOR on the contents of register rs and rt, place the result in rd
SUBU rd, rs, rt (Subtract Unsigned)	subtract the contents of rt from rs and place in rd (unsigned)
LW rt, offset(base) (Load Word)	load into rt the value from memory location that is the sum of offset and the value in register base
SW rt, offset(base) (Store Word)	Store the value from register rt into the memory location that is the sum of offset and the value in register base
BGEZ rs, offset (Branch on Greater Than or Equal to 0)	if the contents of rs $\geq$ 0, branch to (PC + offset) (offset is shifted two bits and sign extended)
JALR rd, rs (Jump and Link Register)	jump to the address in register rs, placing the next instruction in rd
SYSCALL (System Call)	a system call exception transferring control to the exception handler
BREAK (Break)	a breakpoint trap occurs, transferring control to the exception handler

We can describe any MIPS instruction by some combination of these eight instructions. This set of instructions maintains the feel of a real processor by providing a wide range of functionality. It contains a logical operator (NOR), an arithmetic operator (SUBU), loading and storing instructions (LW, SW), branching and jumping capabilities (BGEZ, JALR), and exceptions (SYSCALL, BREAK). In practice, programs will be run through a pre-assembler that will reduce all instructions down to this set of eight. Instructions will then be turned into machine code, loaded onto the processor, and run.

Different approaches were needed for each type of instruction that we wanted to reduce out of the set. For example, an ADDU (add unsigned) can be accomplished by subtracting one of the values from 0 and then subtracting that value from the other. Similarly, an AND is accomplished by negating both values and then performing a NOR.

Many instructions make use of the fact that adding a number to itself is the same as shifting the number once to the left. Shift lefts and shift rights are easily accomplished using this concept, and multiplies are also aided by this fact. We perform our multiplies by shifting the multiplier left one bit at a time, each time checking the MSB. If the MSB is 1, the multiplicand is added to the product, which is also shifted each time we check the multiplier. Thus, a multiply is carried out in the same manner that a human would multiply two numbers by hand. (The multiplier is forced to be positive so that this will work. If a negative multiplier is present, we negate both the multiplier and multiplicand to achieve this. Checking the MSB of a number is done via BGEZ.)

The divide instruction is also implemented in the same way that a person would do it by hand. The dividend is shifted one bit at a time and is compared to the divisor upon each shift. At any step, if the dividend is greater than or equal to the divisor, the divisor is subtracted from the dividend and the quotient is incremented by one. Just like with the multiply, the quotient is shifted along with the dividend in order to keep the answer correct. At the end, the remaining dividend is placed into the remainder register.

Table 3 shows how many instructions must be used to replace each MIPS instruction. The total is the number of instructions used to replace the given instruction, while the average is the average number of instructions that will be executed in place of the given instruction. Note that the average can be greater than the total code expansion because of loops within the code. LUI and LI (load upper immediate and load immediate, which load 16-bit constants into a register) require only one instruction because the assembler assists them by placing the value to be loaded into memory prior to operation.

It is clear that by reducing the number of usable instructions we are consequently inflating the overall instruction count and thus reducing our processor's performance. Remember, however, that performance is not one of our major design goals. A simpler implementation is more valuable to us than fast processing, so we believe the instruction inflation shown in Table 3 is acceptable.

**Table 3.** Instruction Inflation due to Reducing Set

INST	TOT	AVG	INST	TOT	AVG	INST	TOT	AVG	INST	TOT	AVG
LB	13	12.5	J	2	2	ORI	3	3	DIVU	30	352.5
LBU	8	8	JAL	2	2	OR	2	2	SLTI	9	5.83
LH	15	9	JR	1	1	XORI	6	6	SLTIU	9	6
LHU	9	6.5	JALR	1	1	XOR	5	5	SLT	8	4.83
LW	1	1	BEQ	5	4	NOR	1	1	SLTU	9	5
LWL	22	47.25	BNE	6	3.63	ADDI	2	2	SLL	9	51.53
LWR	54	81.75	BLEZ	4	2.5	ADDIU	2	2	SRL	16	64.13
LUI	1	1	BGTZ	5	2.75	ADD	17	5.9	SRA	17	65.09
LI	1	1	BLTZ	2	1.5	ADDU	2	2	SLLV	8	50.53
SB	29	17.75	BGEZ	1	1	SUB	2	2	SRLV	15	63.13
SH	15	9.5	BLTZAL	3	2.5	SUBU	1	1	SRAV	16	64.09
SW	1	1	BGEZAL	2	2	MULT	39	212	SYSCALL	1	1
SWR	25	14	AND	3	3	MULTU	26	336.8	BREAK	1	1
SWL	26	16.25	ANDI	4	4	DIV	38	356.8			

## 5 Classroom Integration

We will provide the students with access to the following items: an XSV board, access to a computer with Xilinx Foundation Software, our processor debugging tool, a parallel port connector, a Foundation project containing an incomplete version of our processor, and a power supply for powering the board. With these tools the students will be able to design and produce a processor on their FPGA. A possible set of projects is suggested here in order to illustrate the use of our tools.

An early assignment for the students will likely be the designing of the register file. Each student will design their register file and add it to the Foundation project, then implement the design onto their FPGAs. Once done, they can use their debugging tool to verify their register file design in multiple ways. The simplest way will be to write values directly to the register file and then read them to assure correct operation, but they could also use the debugging tool to simulate a specific instruction and see that the instruction has the proper effects on the register file.

The next design project might be the ALU. After designing the ALU and implementing it onto the FPGA along with their register files, the students would again be offered with a number of ways to test their designs. First, they could simply use the debugging tool to stimulate the ALU's inputs and control values, observing that the proper value is outputted. They could also hook up the register file outputs to the ALU inputs, testing the two units together by giving addresses to the register file and observing the ALU output. A third option is to simulate a specific instruction and observe its propagation through the register file and ALU. Again, the debugging tool will provide many testing options to the students, allowing them to debug in almost any manner they wish.

When the students have designed a complete single-cycle processor, the debugging tool will then be used not only to stimulate inputs and read outputs, but also to control the CPU clock. Students will be able to read or write any of the processor's memories in order to obtain the state they want, and then they will be able to walk the CPU clock in any increment they wish. For example, a student might load a specific instruction into memory and then run the processor for 0 cycles, thereby loading the instruction and observing the propagation of values through the combinations paths of the CPU. The student could also run the processor for a full cycle, committing to memory whatever the instruction specifies, thereby making certain that the memories are working correctly. Again, a number of debugging options are made available, giving the students the freedom to debug however they wish.

Beyond a single-cycle implementation, the students might also be asked to design a multi-cycle or pipelined processor. In reality, these and almost any projects relating to processor design can be completed because the debugging tool supplies complete control and observability of the processor's internal state.

## 6 Results and Discussion

The results of our implementation of two FPGA-optimized processors appear in Table 4. For the single-cycle implementation we were able to fit our design quite easily into our XILINX Virtex XCV300 PFGA, using only 22% of the LUTs and 25% of the BRAMs. Our FPGA runs at a maximum frequency of 25MHz, which means that the actual processor runs at 1.5MHz (25/16). The pipelined processor produced similar utilization numbers, using 29% of the LUTs, 25% of the BRAMs, and running at 23 MHz (1.4MHz for the processor).

The debugging tool has also been tested, and successfully supplies a user with complete control of the processor even when it is incomplete.

**Table 4.** Implementation Results

PROCESSOR TYPE	FPGA CLOCK FREQ	PROC. CLOCK FREQ	% LUTS USED	% BRAM USED
SINGLE-CYCLE MIPS	25 MHz	1.5 MHz	22%	25%
PIPELINED MIPS	23 MHz	1.4 MHz	29%	25%

The work presented in this paper describes a completely functional and debuggable processor. While we have offered an outline of some possible class projects, instructors who wish to use our implementation will ultimately make the decision on what aspects of architecture to teach. By providing the students with a Foundation file

that already has most parts of the processor, instructors will be able to have their students design and replace the missing parts. Upon completion, students will have a working processor that they can actually operate.

## **7 Conclusion**

The FPGA-optimized processor, debugging tool, and reduced instruction set presented in this paper represent the primary step in our goal of introducing FPGAs into the teaching of computer architecture. The processor and debugging tool should be easy to integrate into the classroom, where they will provide students with a hands-on experience that was previously unavailable to them. We are convinced that when teachers incorporate these tools into their classrooms that students will display better retention of the intended lessons as well as increased enthusiasm about the work performed.

## **8 Future Work**

The next logical step for this research is to integrate it into the classroom. The work presented here will be polished into a workable lesson plan that will likely be tested by a set of students in the spring of 2002. With their comments and critiques, the lessons will then be revamped with the intent of incorporating this work into the teaching of Computer Design and Organization at the University of Washington in the fall of 2002.

There also exists the possibility of merging this work with student-built compilers and operating systems. By combining the teachings of processor design, compilers, and operating systems, students will eventually be able to turn chips into complete working computers. By doing this, the interactions and tradeoffs between these various fields can be explored in real terms, with concrete results.

## **9 Notes**

Along with the previously mentioned eight instructions, our processor also supports the instructions that are presented in *Patterson and Hennessy*: BEQ, ADD, SUB, AND, OR, SLT, and J. This set includes simple and easily understood instructions, and supporting them will provide the students a larger range of instructions with which to test and debug their processor. Thus in total our processor supports 15 instructions.

## Acknowledgments

Thank you to Jered Aasheim for his preliminary work on this project, also to Tim Midget for his work on the reduced MIPS instruction set. This project was funded in part by a donation from Xilinx. Scott Hauck was supported by an NSF Career Award and a Sloan Foundation Fellowship. Mark Holland was supported by an NSF Graduate Fellowship.

## References

- [1] Courter, S.S., S. B. Millar, and L. Lyons, "From the Students' Point of View: Experiences in a Freshman Engineering Design Course", *Journal of Engineering Education*, vol. 87, no. 3, 1998.
- [2] Mahendran, M, "Project-Based Civil Engineering Courses", *Journal of Engineering Education*, vol. 84, no. 1, 1995.
- [3] Xilinx Inc., *The Programmable Logic Data Book*, pp. 3.3-3.12, 1999.
- [4] Xilinx Inc., *Dual Port Block Memory: Product Specification*, May 28 1999.
- [5] Patterson, D. A., J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, San Francisco, 1998.
- [6] Alliance Core, *ARC 32-Bit Configurable RISC Processor: Datasheet*, July 3 2000.
- [7] Altera Corporation, *ARM-Based Embedded Processor Device Overview: Datasheet*, ver. 1.2, February 2001.
- [8] Altera Corporation, *MIPS-Based Embedded Processor Device Overview: Datasheet*, ver. 1.2, February 2001.
- [9] Gray, J., "Designing a Simple FPGA-Optimized RISC CPU and System-on-a-Chip", 2000.
- [10] XESS Corporation, *XSV Board V1.0 Manual*, March 1 2000.
- [11] Alliance Semiconductor, *5V/3.3V 512K x 8 CMOS SRAM: Datasheet*, ver. 1.0, January 12 2001.
- [12] XESS Corporation, *XSV Parallel Port Interface: Application Note* by D. Vanden Bout, ver. 1.0, April 10 2000.
- [13] Styer, Eugene, "One Instruction Computers", <http://eagle.eku.edu/faculty/styer/oisc.html>, 1996.