Hands-on Computer Architecture – Teaching Processor and Integrated Systems Design with FPGAs

Jan Gray
Gray Research LLC, P.O. Box 6156, Bellevue, WA, 98008
jsgray@acm.org

Abstract – **Field programmable gate arrays are an ideal substrate for computer architecture project courses. FPGA-based processor development offers some learning opportunities that pure simulation approaches cannot rival. This paper first introduces the XSOC Project, a free kit that includes the xr16 RISC CPU core, system-on-a-chip infrastructure, peripheral cores, C compiler, and simulator. Such kits have numerous applications in a first architecture course. It then suggests that FPGA-based processors can also be applied to the study of advanced architecture topics including memory systems, multithreading, LIWs, chip multiprocessors, and architectural support for programming languages and networking.**

I. Introduction

FPGAs are ideal for teaching hands-on digital design and computer design. It is now possible to achieve an affordable FPGA-based computer design kit that is simple enough to be understood end-to-end, and rich enough to demonstrate whole computer systems.

This paper presents one such kit and then examines how FPGA processors and systems could be applied in teaching undergraduate computer architecture. The paper concludes with speculation on how FPGA CPUs might also be applied in graduate level advanced architecture studies and research.

II. The XSOC Project

To promote do-it-yourself processor development, the author posted numerous Usenet postings [1] and a web site [2] on FPGA processor design, but surprisingly few FPGA-based CPUs emerged over the years [3]. Perhaps there were too many barriers to success – lack of published reference designs, expensive FPGA tools, and lack of compiler support. To help remove these barriers, the author prepared how-to articles [4] and the XSOC Project Kit [5], to provide a concrete end-to-end example of a practical FPGA-based processor SoC.

The goals of this kit are to promote processor and integrated systems design in FPGAs; to show that FPGA-based SoCs can be cost-effective alternatives to ASICs; to help establish a community of designers and a library of reusable cores; and especially, to make it practical for students and hobbyists to design and build their own custom processors and systems.

The kit depends upon the recent emergence of low-cost FPGA development tools. XSOC uses Xilinx Student Edition 1.5 (XSE), approximately $100, which includes the Xilinx Foundation Express tools, including schematic capture, HDL synthesis, simulator, and FPGA place-and-route development tools. XSE also includes a textbook [6] that gently introduces digital design concepts through FPGA lab exercises, culminating in the design of simple 4- and 8-bit processors. The exercises can be performed on the XESS XS40 prototyping board, which includes an XC4005XL or XC4010XL FPGA, a 100 MHz programmable oscillator, 32-128 KB of RAM, 8031 MCU, parallel port, VGA port, keyboard/mouse port, as well as full documentation, tools to download FPGA designs and memory images to the board, and a thriving support mailing list.

Building on this foundation, the XSOC Kit includes design files (both schematics and synthesizable Verilog) of the XSOC system-on-a-chip, the xr16 pipelined RISC processor core, on-chip bus and off-chip memory controller, and peripherals cores including on-chip RAM, parallel port, and a bilevel VGA controller. It also includes a port of the lcc retargetable C compiler, an assembler and instruction set simulator, and documentation and specifications. Full sources are included, and XSOC may be used without charge for non-commercial educational and research purposes.

Most soft CPU cores are synthesized implementations of legacy instruction sets that fill large, expensive FPGAs, and have slow cycle times. In contrast, the xr16 processor demonstrates that a simple, thrifty CPU design can achieve a cost-effective integrated computer system in a tiny FPGA.

The XSOC system-on-a-chip (excluding RAM/ROM) fits in a 392 logic cell XC4005XL; the processor itself is about 270 logic cells. (A logic cell is an FPGA unit of area that includes one 4-input lookup table (LUT) and one D flip-flop.) Put another way, the entire xr16 core occupies less than ½ of 1% of the area of the largest Xilinx Virtex-E device (XCV3200E, 73,008 logic cells). It has a cycle time of approximately 30 ns in

Xilinx SpartanXL devices, which should approach 10 ns in newer Xilinx Virtex FPGA devices.

III. xr16 Processor

The goal of the xr16 processor design was to demonstrate a full-featured pipelined RISC processor that runs integer C code and fits in an XC4005XL. Area and performance were important, but so were simplicity and ease-of-understanding.

The xr16 processor is a classic RISC processor, with a 16-bit instruction word, sixteen 16-bit registers, byte and word load/store, and add with carry support to form long integers from register pairs. A stretch version, xr32, with 32-bit registers, is forthcoming.

A. Instruction set design

The xr16 instruction set was designed as follows. First the lcc retargetable C compiler was ported to a "generic" 16-bit RISC. Then a handful of sample C programs were compiled and histograms of instruction frequencies were studied. Next it was determined which instructions could be synthesized from others. Finally, a strategy for encoding word-wide immediate constants was selected. The result:

- only `add`, `sub`, `addi` are 3-operand;
- `r0` always reads as 0;
- 4-bit immediate fields; for 16-bit constants, a prefix `imm` establishes the most-significant 12-bits of the immediate operand of the instruction that follows;
- interlocked compare and conditional branch sequence instead of architected condition codes;
- `jal` (jump-and-link) jumps to an effective address, saving the return address in a register;
- `call` *`func`* compactly encodes `jal r15,`*`func`*;
- perform mul, div, and variable-bit shifts in software.

The xr16 processor has six instruction formats and 43 instructions; xr32 adds 2 additional instructions (load/store longword).

Format	15 12	11 8	7 4	3 0
rrr	op	rd	ra	rb
rri	op	rd	ra	imm
rr	op	rd	fn	rb
ri	op	rd	fn	imm
i12	op	imm12		
br	op	cond	disp8	

Table 1: XR Instruction Formats

Hex	Fmt	Assembler	Semantics
0*dab*	rrr	`add rd,ra,rb`	rd = ra + rb;
1*dab*	rrr	`sub rd,ra,rb`	rd = ra – rb;
2*dai*	rri	`addi rd,ra,imm`	rd = ra + imm;
3*d***b*	rr	`{and or xor andn adc sbc} rd,rb`	rd = rd *op* rb;
4*d***i*	ri	`{andi ori xori andni adci sbci slli slxi srai srli srxi} rd,imm`	rd = rd *op* imm;
5*dai*	rri	`lw rd,imm(ra)`	rd = *(ushort*)(ra+imm);
6*dai*	rri	`lb rd,imm(ra)`	rd = *(byte*)(ra+imm);
7*xxx*	–	*reserved*	
8*dai*	rri	`sw rd,imm(ra)`	*(ushort*)(ra+imm) = rd;
9*dai*	rri	`sb rd,imm(ra)`	*(byte*)(ra+imm) = rd;
A*dai*	rri	`jal rd,imm(ra)`	rd = pc, pc = ra + imm;
B*dd*	br	`{br brn beq bne bc bnc bv bnv blt bge ble bgt bltu bgeu bleu bgtu} label`	if (*cond*) pc += 2*disp8;
C*iii*	i12	`call func`	r15 = pc, pc = imm12<<4;
D*iii*	i12	`imm imm12`	imm'next$_{15:4}$ = imm12;
E*dai*	rri	`ll rd,imm(ra)`	rd = *(ulong*)(ra+imm);
F*dai*	rri	`sl rd,imm(ra)`	*(ulong*)(ra+imm) = rd;
E*xxx* **F***xxx*	–	*reserved (xr16)*	

Table 2: XR Instruction Set

Some instructions are synthesized from others.

Assembly	Maps to
`nop`	`and r0,r0`
`mov rd,ra`	`add rd,ra,r0`
`cmp ra,rb`	`sub r0,ra,rb`
`subi rd,ra,imm`	`addi rd,ra,-imm`
`cmpi ra,imm`	`addi r0,ra,-imm`
`com rd`	`xori rd,-1`
`lea rd,imm(ra)`	`addi rd,ra,imm`

`lbs rd,imm(ra)` *(load-byte, sign-extending)*	`lb rd,imm(ra)`
	`xori rd,0x80`
	`subi rd,0x80`

`j addr`	`jal r0,addr`
`ret`	`jal r0,0(r15)`

Table 3: Synthesized Instructions

To keep things simple, there are no branch delay slots. The architecture also reflects a streamlined implementation. One shared memory port means load/store instructions take two cycles. To save an adder and a mux, jumps and taken branches take three cycles. And to save another mux, result forwarding is performed on only the first register operand. (The assembler handles other cases.)

B. Implementation

The FPGA. XSOC/xr16 is implemented in a Xilinx XC4005XL-PC84C-3. This device has a 14×14 array of configurable logic blocks (CLBs) and 61 I/O blocks (IOBs) in a sea of programmable interconnect.

Every CLB has two 4-input lookup tables (LUTs) and two flip-flops. Each LUT can implement any logic function of 4 inputs, or a 16×1-bit synchronous static RAM, or ROM. Each CLB also has "carry logic" to help build fast, compact ripple-carry adders.

Each IOB offers input and output buffers and flip-flops. The output buffer can be 3-stated for bidirectional I/O.

The programmable interconnect routes CLB/IOB output signals to other CLB/IOB inputs. It also provides wide-fanout low-skew clock lines, and horizontal *long lines* which can be driven by 3-state buffers at each CLB.

The XC4000XL is ideal for implementing processors. Just 8 CLBs can build a single-port 16×16-bit register file (using LUTs as SRAM), a 16-bit adder/ subtractor (using carry logic), or logic unit. Since each LUT has a flip-flop, the device is *register rich*, enabling a pipelined implementation style; and as each flip-flop has a dedicated clock enable input, it is easy to stall the pipeline. Long lines and 3-state drivers form efficient word-wide result multiplexers and on-chip buses.

Pipeline design. The xr16 has a 3-stage pipeline:

- IF: instruction fetch
- DC: decode and operand fetch
- EX: execute and write-back results

Each pipeline stage incurs an instruction fetch memory access, an optional load/store access, and optional DMA accesses. Each memory access can take one or more memory cycles – if the memory RDY signal is not asserted, the pipeline does not advance.

For pipeline data hazards, the xr16 includes a result forwarding mux on the A operand only.

Conditional branches and jumps take place during the EX pipeline stage; the IF and DC stage instructions in progress are annulled.

Control unit design. The control unit inputs are the next instruction $INSN_{15:0}$ and RDY signals from memory and the Z,N,CO,V outputs from the datapath. The control unit outputs include the next memory access control signals and the datapath control signals.

As instructions flow through the instruction register (IR) pipeline, they are decoded. In the DC stage, and the control unit drives the register file and operand selection control signals.

If the DC stage instruction is a conditional branch, the EX stage instruction must be an `add`/`sub`, and its condition code outputs are evaluated against the branch condition. If the branch is taken, the branch displacement is added to PC in the EX stage. On jumps and taken branches the control unit FSM annuls the two instructions in the branch shadow.

In the EX stage, the control unit drives ALU, result mux, and address/PC unit control outputs.

On interrupts, the control unit replaces the fetched instruction with `jal r14,10(r0)`, which calls the interrupt handler. Interrupt return is `jal r0,0(r14)`.

Datapath design. The datapath executes instructions at up to 1 IPC. It consists of a register file, operand selection multiplexers, ALU, result multiplexer, and an address/PC unit.

The 2R-1W register file is implemented as two copies of a 1R-1W file, each a 16x16 SRAM (16 LUTs). The two register operands are read in the first half of each cycle, and the EX stage result is written back into both copies in the second half. The A operand is either the register file port A output, or the forwarded result value, selected by a multiplexer. The B operand is either the register file port B output or a sign-/zero-extended immediate value formed from the IR's `imm` and/or `imm12` fields. The A and B operand muxes and registers are each a column of 16 LUTs and flip-flops.

The ALU consists of a 16-bit adder/subtractor (20 LUTs) and a 16-bit logic unit (16 LUTs). The shifter requires no logic, it merely skews the A operand register left or right by one bit.

The result multiplexer selects an EX stage result value from the adder, logic unit, "shifters", return address, or a word or zero-extended byte load result. It implements a 7-input 16-bit-wide mux using long lines and 3-state buffers, conserving precious logic.

The address/PC unit adds either +2 or the branch displacement to PC (8+16 LUTs). The next address is either the next PC or the effective address computed in the ALU, as selected by ADDRMUX (16 LUTs).

The processor also acts as the DMA engine. Instead of a simple register, PC is a 16x16 *register file*, with PC_0

storing the program counter, and $PC_{15..1}$ storing DMA address counters.

Figure 1 is the XSOC system and xr16 processor top-level schematic, with processor P, memory and on-chip bus controller MEMCTRL, on-chip bus and peripherals PARIN, PAROUT, IRAM, and the VGA controller.

The on-chip data bus uses an *abstract peripheral control bus* to provide glue-logic-free interfacing to peripheral cores. This abstraction also makes it possible to evolve the on-chip bus protocol without impacting existing systems and peripheral cores.

The processor schematic (not shown) simply interconnects an instance of the control unit (Figure 2) and the datapath (Figure 3).

The design is synchronous and it is safe to stop the clock. During system bring up, the XS40 prototype board was attached to a PC parallel port and the clock was driven at 1 Hz using a shell script.

To minimize area and cycle time, the datapath is hand-floorplanned using RLOC attributes (Figure 4). Datapath CLBs are white, other placed CLBs are light gray. A few critical paths are manually technology mapped using FMAPs. The design is placed and routed with timing constraints to further optimize critical paths.

Figure 5 shows the XSOC FPGA in the context of the XS40 board. The 8031 is not used and is held in reset.

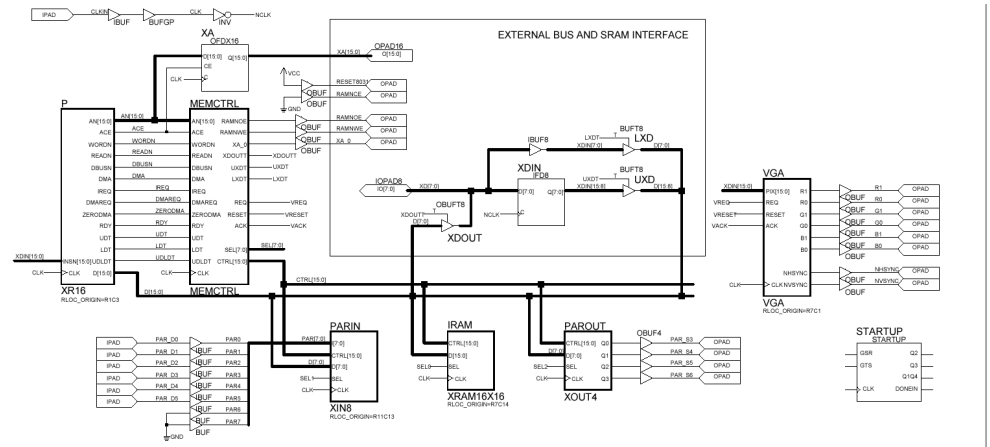

Figure 1: XSOC/xr16 Schematic

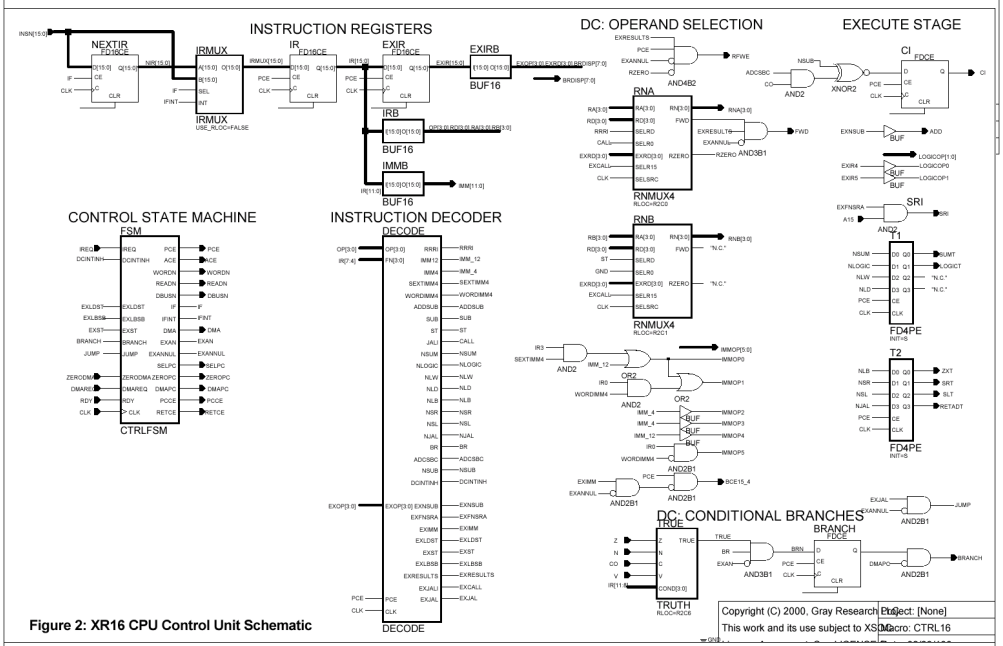

Figure 2: XR16 CPU Control Unit Schematic

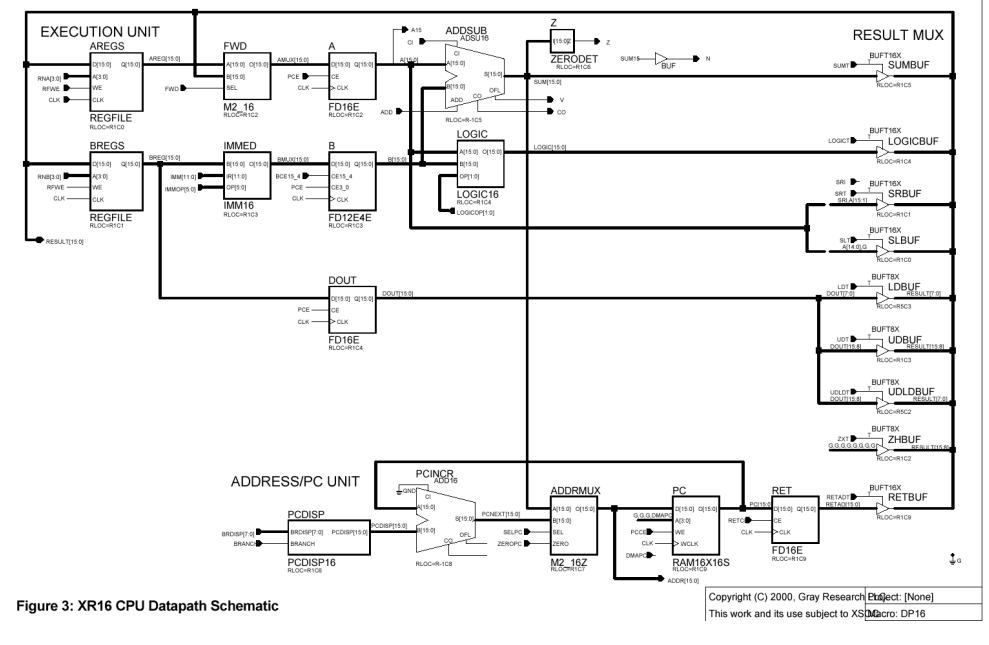

Figure 3: XR16 CPU Datapath Schematic

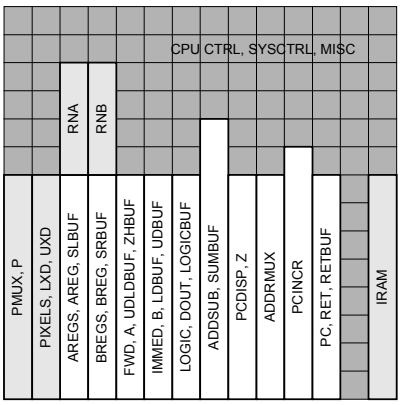

Figure 4: XSOC/xr16 Floorplan

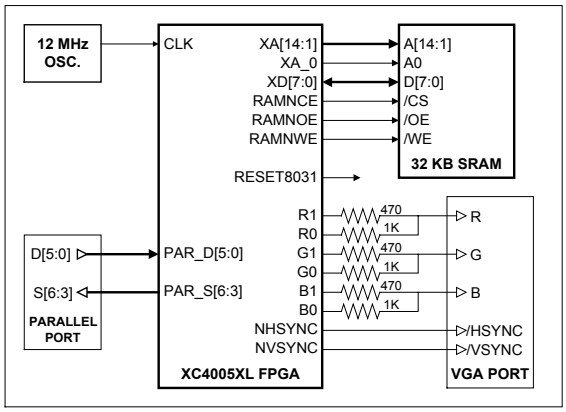

Figure 5: XSOC In Situ

Figure 6 shows the VGA display while running a demo.

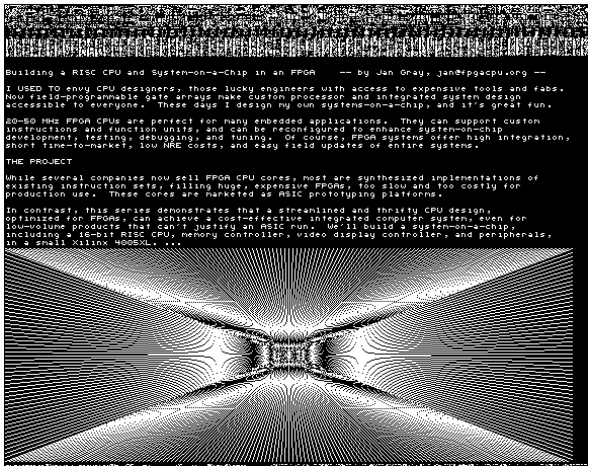

Figure 6: XSOC Graphics Demo Display

The 576x455 bilevel bitmapped VGA controller displays all 32 KB of RAM. The top lines of the screen are the demo program binary, followed by the display font tables. Below that are some two dozen lines of text output, and some XOR line graphics. And below that lies the stack.

It's fun to watch all of memory at 60 Hz. One can observe the top of the call stack moving up and down, the stack variables changing, and counters counting, and one is left with a visceral impression of the speed of the machine and where its program spends its time.

C. Development tools

The XSOC Project Kit includes source code (or references to same) and Win32 binaries for *lcc-xr16*, a port of the lcc retargetable C compiler [7] for xr16, and *xr16*, the xr16 assembler and instruction set simulator.

The first port of lcc to target xr16's 16-bit int and pointer model, initially based upon the MIPS machine description, took the author (a compiler developer) only one day. Further modifications to also target the 32-bit xr32 processor required just a few hours to revise the approximately 200 lines of xr32 specific instruction templates. These pleasing results are a testament to lcc's retargetability, and reflect that the XR processors were designed as targets of this compiler.

The xr16 assembler/instruction set simulator is also straightforward. The assembler, about 1300 lines of code, can be run for the side-effect of emitting a listing file and image file, or to initialize memory for the instruction set simulator. The latter is a simple switch-based interpreter, some 400 lines of code, which runs a perfectly adequate 3,000,000 instructions per second on a 266 MHz PC.

As mentioned, the kit includes full design sources in both schematics and Verilog source. The latter are compact enough to run the entire system test bench using the free (1,000 line limit) Veriwell Verilog simulator. It is quite instructive to compare and contrast the instruction set simulator output to the Verilog simulator output, the latter highlighting pipeline stalls, annulled instructions, and so forth.

IV. Teaching Applications of FPGA CPUs

XSOC serves as a proof by example that an entire system-on-a-chip (sans RAM) can be built in an modest FPGA, using inexpensive tools. This section explores the teaching value of such FPGA computer systems.

Why build an undergraduate architecture course around FPGA-based processors and systems? Because there is such value in the experience of building real hardware.

Besides the emotional appeal of booting a computer made of your own ideas and your own hands (and how many educators have had that pleasure?), FPGA CPUs can impart a *realism* to the learning experience that is probably not available in more textbook or simulator-based approaches.

So much of computer architecture is about making tradeoffs such as performance versus area versus cycle time versus power, etc. While there is much value in a course project to develop a processor model in an HDL, and then study its behavior in a simulator, it doesn't go far enough. It's like teaching how to balance a home budget but with a bottomless checking account. By not closing the loop with some kind of *realistic* cycle time, area, and resource-usage data, the design tradeoffs aspect suffers.

Of course, student projects could close the loop through the use of real EDA tools, but that would be unnecessarily expensive and complicated. In practice, student editions of FPGA EDA tools suffice, producing the desired timing analysis and resource usage reports.

In an implementation-oriented course, the tradeoffs are so much more quantifiable. Students can now experience, as did the author, that there is an area and delay cost to everything, even a multiplexer; how to find a critical path and how retiming moves them about; how to trade-off area for new functionality vs. area for reduced cycle times; the importance of floorplanning; and may even discover that adding something to a design can make it slower.

Of course, the lessons of FPGA implementation will not directly apply to custom silicon implementations. A student of FPGA CPUs might be surprised to learn that a 16x32-bit register file, 32-bit adder, or 32-bit 2-1 multiplexer, each 16 CLBs in an FPGA, vary widely in area in a full custom design. But the method of systematically evaluating design alternatives and tradeoffs is the same regardless of the implementation technology.

(Since, compared to an FPGA, a full custom design offers perhaps 20 times more gates, each 10 times faster, it follows that designing processors in programmable logic is not unlike taking a time machine back through the last seven years of Moore's Law. 100 MHz pipelined scalar and 2-issue RISC processors are feasible, but 800 MHz associatively indexed out-of-order issue buffers are not!)

Here are some other benefits of the realism imposed by a hardware-based course project.

- There is less hand-waving allowed – designs must be more thorough and complete or the implementation tools will fail to compile them.
- Students learn the testing imperative – to the extent students produce an untestable design, or skimp on writing test-benches, they will learn their lesson in the lab hunched over a hot oscilloscope.
- Students live the "system bring up" experience, working through the adversity of a design that does not start or that fails intermittently, and ending with the sublime glow felt when the darn thing finally works.

Realism aside, FPGA CPUs have other benefits. The use of a custom, teaching-oriented CPU, FPGA or otherwise, should permit course material on architecture and implementation to be streamlined as compared to a legacy instruction set architecture or even a subset. For example, the xr16 core is so simple that a student should be able to understand the purpose and function of every last gate.

Studying computer design with FPGAs also confers vocational training benefits. As FPGAs get faster, larger, and cheaper, and as the minimum volume for gate array starts continues to rise, FPGAs will increasingly displace gate arrays and even full custom implementations from many application areas. In time, the majority of digital systems designs could well be in programmable logic, and FPGA CPU cores could become as commonplace as are discrete embedded processors today. FPGA system design expertise should be a quite marketable skill.

V. FPGA CPU Project Ideas

Is it realistic to expect undergraduates (perhaps working in teams) to produce working FPGA CPUs and systems? Perhaps – it would not seem to be too difficult a stretch, at least for EE students who have already had a first course in digital design and exposure to HDLs.

Of course, the specific course project can be tailored to the class level, class prerequisites, and to available teaching and lab resources. Assume students receive a kit with infrastructure software (instruction set architecture and core interface specifications, compiler, assembler, instruction set simulator, test suites), system-on-chip cores, and in some cases, the processor core itself. Here are some projects they might tackle:

- Implement a processor core for the given ISA.

- Double its performance – evolve a non-pipelined core into a pipelined core.
- Add a cache, MMU, or exceptions.
- Given C code with a critical inner loop, build an (on-chip) coprocessor to speed it up. Or add new custom instructions to speed it up. Don't forget to enhance the compiler, assembler, simulator, and test suite!
- Port the design to a new FPGA device architecture, and retime the pipeline.
- Build a system-on-a-chip for a particular embedded application.
- Add a new (on-chip) peripheral core – design the core, add it to the SoC, write the interrupt handler or device driver, add testing support to the test bench.
- Develop a test suite or test bench for the system or processor.
- Reimplement a subset of a famous legacy ISA.

Our favorite project idea simulates the competitive processor design industry. Student teams are issued a CPU design kit, including compiler tools, a working, non-pipelined processor core, a benchmark suite, and an FPGA board, which runs "out of the box", and are instructed to evolve and optimize their processor, including its instruction set architecture and tools, in order to run the benchmark suite as fast as possible (or in as little total energy as possible). At end of term, teams submit their designs and vie for the coveted "fastest CPU design" trophy. This sort of project could uniquely motivate students to practice all manner of quantitative analysis and design activities.

VI. FPGA CPUs for Advanced Computer Architecture Studies and Research

FPGA devices improve at a rapid pace. Comparing the Xilinx XC4013 (1152 logic cells in 1993) with the XCV3200E (73,008 logic cells in 2000), reveals an improvement of 2^6 in just seven years. Recent FPGAs from Xilinx and Altera, include Virtex-E and APEX, now provide vast quantities of programmable logic and some dozens of large (e.g. 256x16b) embedded RAM blocks. This section considers how these new devices enable direct prototyping of some more advanced areas of computer architecture research.

The xr16 core consumes 270 logic cells. The 32-bit xr32 core will be approximately 430 logic cells. Redesigning xr32 for Virtex, and trading off some logic for speed (using *dual*-port RAM for the register files, and replacing the result multiplexer TBUFs with actual LUT-based muxes), this could rise to 600-700 logic cells. Even so, such a streamlined 32-bit RISC would still use only 1% of the largest Virtex-E device, and less than 5% of a mid-range (15,552 logic cell) XCV600E. Therefore FPGAs now have adequate capacity to prototype 8-32-way 32-bit chip multiprocessors.

Then there are the new embedded RAM blocks. The XCV600E provides 72 256x16 (or 512x8, etc.) dual-port synchronous SRAMs with sub-5 ns cycle times. Block RAM has many architectural applications [8], including:

- registers: vector register files, windowed register files, and multi-context register files;
- stacks: operands, activation records, control;
- on-chip RAM, ROM, and microcode control stores;
- caches: data, tags, write-accumulation structures, victim buffers;
- branch prediction: branch history tables and branch target address/instruction caches;
- MMUs: segmentation registers, and translation lookaside buffers (no associative lookup though);
- debug and tracing support: breakpoint code and data address, count or value registers, branch traces, PC traces, memory access traces.

and systems applications, including:

- interconnects: on-chip packet/cell buffers, queues, and virtual channel buffers;
- graphics: video line input or output buffers or delay lines, texture caches, sprites, pattern generators, display lists, span buffers (color, Z), color mapping LUTs.
- garbage collection: read, write barriers via page table attribute bits or region table address checks, card marking bit array;
- multimedia: DCT and IDCT support (8x8 pixel blocks, coefficient tables, compression tables);

Indeed, there are enough embedded RAM blocks in that mid-range XCV600E to prototype an 8 or 16 CPU chip multiprocessor, where each processor is multithreaded with an 8-context 32x32-register file (2 block RAMs), and with a 256x32b I-cache (2 block RAMs), that share one 1024x32b L2 cache (8 block RAMs).

FPGA I/O capabilities have also made great strides, with the newer devices supporting a smorgasbord of signaling standards and supporting interfaces such as 200 MHz ZBT SSRAM and 266 MHz DDR SDRAM, and providing inter-FPGA signaling rates of up to 300 Mbps/pin. It should be possible to prototype fast multi-banked memory systems and interconnect fabrics.

Returning to the subject of advanced architecture studies, again it would seem that there is some value in the additional grounding in reality inherent in a project-oriented course.

Assume once again that students are issued an FPGA CPU and system-on-a-chip kit, including software tools and a toolkit of processor, system, interface, and peripheral cores designs. Here are some of the topics students might investigate, analyze, simulate, and then prototype.

- Build a 2-3 operation long-instruction word machine, including tools support. (Challenges here are mostly in the register file design and the code generator.)
- Build a 2-issue superscalar processor.
- Build a chip multiprocessor including a suitable memory system.
- Build a multithreaded processor for a given workload.
- Build a fault tolerant processor from several lock-step self-checking processor pairs.
- Add architectural support for non-pausing multi-threaded garbage collection via hardware read and write barriers.
- Add architectural support for network routing, packet inspection, etc. possibly including integrated memory streaming or DMA instructions.
- Add architectural support for message passing between two processors on one (or separate) devices.
- Add architectural support for debugging or tracing.
- For a particular signal processing problem, add a fixed FPGA DSP datapath to the FPGA computer system.

Put another way, modern FPGAs make it possible to study-by-prototype just about anything that appears in a modern computer system. Indeed, except for large or content associative or many-ported memories, there are few structures in the computer architect's toolbox that are not easily implemented in an FPGA.

VII. Related Work

Several schools have used FPGA processor design projects to help teach computer design.

At Virginia Tech in 1995, students of EE6504, *Rapid Prototyping of Computing Machinery,* designed 16-bit HOKIE RISC processors for an XC4010 FPGA. [9]

The 1998 Cornell EE475 architecture class labs included VHDL design and FPGA verification of a simple CPU and a subsequent pipelined version. [10]

At Hiroshima City University, "more than 75% (about 40) of the students in a class succeed to create their own original FPGA computers within 15 weeks in the first term of their junior student days every year since 1996." Students work in pairs, design RISC or CISC CPUs, write HDL and target FPGAs. [11]

At Georgia Tech, students model pipelined RISC processors in HDLs and synthesize and run them in low-cost Altera and Xilinx prototyping cards. [12]

VIII. Conclusion

Simulation is good, but it does not model the elation felt when one's computer design boots in real hardware. All that material on architectural tradeoffs, retiming, and floorplanning, seems so much more relevant as applied to the device on your laboratory workbench.

"I hear and forget. I see and I remember.
I do and I understand."
– old Chinese saying, courtesy Prof. Philip Leong

IX. References

[1] J. Gray, "FPGA CPU Usenet Posting Archives", www.fpgacpu.org/usenet/, March 2000.

[2] J. Gray, "Homebrewing RISC Microprocessors In FPGAs", www3.sympatico.ca/jsgray/homebrew.htm, August 1996.

[3] J. Gray, "FPGA CPU Links", www.fpgacpu.org/links.html, March 2000.

[4] J. Gray, "Building a RISC System in an FPGA: Part 1: Tools, Instruction Set, and Datapath; Part 2: Pipeline and Control Unit Design; Part 3: System-on-a-Chip Design", Circuit Cellar Magazine, #116-118, March-May 2000.

[5] J. Gray, "The XSOC Project Kit", www.fpgacpu.org/xsoc/, March 2000.

[6] D Vanden Bout, *The Practical Xilinx® Designer Lab Book*, Prentice Hall, 1998.

[7] C. Fraser and D. Hanson, *A Retargetable C Compiler: Design and Implementation*, Benjamin Cummings, 1995. www.cs.princeton.edu/lcc.

[8] J. Gray, "The Myriad Uses of Block RAM", www.fpgacpu.org/usenet/bb.html, Oct. 1998.

[9] P. Athenas, "The Hokie *Instant* RISC Microprocessor", www.ee.vt.edu/courses/ee6504/, 1995.

[10] B. Land, "Elect Eng 475 Microprocessor Architectures", instruct1.cit.cornell.edu/Courses/ee475/.

[11] R. Takahashi and N. Yoshida: "Diagonal Examples for Design Space Exploration in an Educational Environment CITY-1", Proc. 1999 Int'l Conf. on Microelectronic Systems Education pp.71-73 (1999). Also "Microcomputer Design Educational Environment City-1", www.lcl.ce.hiroshima-cu.ac.jp/~activity/City-1/.

[12] J. Hamblen, "Using Large CPLDs and FPGAs for Prototyping and VGA Video Display Generation in Computer Architecture Design Laboratories", IEEE Technical Committee on Computer Architecture Newsletter, Feb. 1999, pp.12-14.