CIRCUIT CELLAR®
THE MAGAZINE FOR COMPUTER APPLICATIONS

FEATURE ARTICLE

Building a RISC System in an FPGA

Part 1: Tools, Instruction Set, and Datapath

Jan Gray

To kick off this three-part article, Jan's going to port a C compiler, design an instruction set, write an assembler and simulator, and design the CPU datapath. Get reading, you've only got a month before your connecting article arrives!

I used to envy CPU designers—the lucky engineers with access to expensive tools and fabs. But, field-programmable gate arrays (FPGAs) have made custom-processor and integrated-system design much more accessible.

20–50-MHz FPGA CPUs are perfect for many embedded applications. They can support custom instructions and function units, and can be reconfigured to enhance system-on-chip (SoC) development, testing, debugging, and tuning. Of course, FPGA systems offer high integration, short time-to-market, low NRE costs, and easy field updates of entire systems.

FPGA CPUs may also provide new answers to old problems. Consider one system designed by Philip Freidin. During self-test, its FPGA is configured as a CPU and it runs the tests. Later the FPGA is reconfigured for normal operation as a hardwired signal processing datapath. The ephemeral CPU is free and saves money by eliminating test interfaces.

THE PROJECT

Several companies sell FPGA CPU cores, but most are synthesized implementations of existing instruction sets, filling huge, expensive FPGAs, and are too slow and too costly for production use. These cores are marketed as ASIC prototyping platforms.

In contrast, this article shows how a streamlined and thrifty CPU design, optimized for FPGAs, can achieve a cost-effective integrated computer system, even for low-volume products that can't justify an ASIC run.

I'll build an SoC, including a 16-bit RISC CPU, memory controller, video display controller, and peripherals, in a small Xilinx 4005XL. I'll apply free software tools including a C compiler and assembler, and design the chip using Xilinx Student Edition.

If you're new to Xilinx FPGAs, you can get started with the Student Edition 1.5. This package includes the development tools and a textbook with many lab exercises.[3]

The Xilinx university-program folks confirm that Student Edition is not just for students, but also for professionals continuing their education. Because it is discounted with respect to their commercial products, you do not receive telephone support, although there is web and fax-back support. You also do not receive maintenance updates—if you need the

Register	Use
r0	always zero
r1	reserved for assembler
r2	function return value
r3–r5	function arguments
r6–r9	temporaries
r10–r12	register variables
r13	stack pointer (sp)
r14	interrupt return address
r15	return address

Table 1—*The xr16 C language calling conventions assign a fixed role to each register. To minimize the cost of function calls, up to three arguments, the return address, and the return value are passed in registers.*

Listing 1—*This sample C code declares a binary search tree data structure and defines a binary search function.* `Search` *returns a pointer to the tree node whose* `key` *compares equal to the argument key, or* `NULL` *if not found.*

```
typedef struct TreeNode {
  int key;
  struct TreeNode *left, *right;
} *Tree;

Tree search(int key, Tree p) {
  while (p && p->key != key)
    if (p->key < key)
      p = p->right;
    else
      p = p->left;
  return p;
}
```

next version of the software, you have to buy it all over again. Nevertheless, Student Edition is a good deal and a great way to learn about FPGA design.

My goal is to put together a simple, fast 16-bit processor that runs C code. Rather than implement a complex legacy instruction set, I'll design a new one streamlined for FPGA implementation: a classic pipelined RISC with 16-bit instructions and sixteen 16-bit registers. To get things started, let's get a C compiler.

C COMPILER

Fraser and Hanson's book is the literate source code of their lcc retargetable C compiler.[1] I downloaded the V.4.1 distribution and modified it to target the nascent RISC, xr16.

Most of `lcc` is machine independent; targets are defined using machine description (md) files. `Lcc` ships with 'x86, MIPS, and SPARC md files, and my job was to write `xr16.md`.

I copied `xr16.md` from `mips.md`, added it to the makefile, and added an xr16 target option. I designed xr16 register conventions (see Table 1) and changed my md to target them.

At this point, I had a C compiler for a 32-bit 16-register RISC, but needed to target a 16-bit machine with sizeof(int)=sizeof(void*)=2. lcc obtains target operand sizes from md tables, so I just changed some entries from 4 to 2:

```
Interface xr16IR = {
  1, 1, 0,  /* char */
  2, 2, 0,  /* short */
  2, 2, 0,  /* int */
  2, 2, 0,  /* T* */
```

Next, `lcc` needs operators that load a 2-byte `int` into a register, add 2-byte `int` registers, dereference a 2-byte pointer, and so on. The `lcc ops` utility prints the required operator set. I modified my tables and instruction templates accordingly. For example:

```
reg:  CVUI2(INDIRU1(addr)) \
  "lb r%c,%0\n" 1
```

uses `lb rd,addr` to load an unsigned char at `addr` and zero-extend it into a 16-bit `int` register.

```
stmt: EQI2(reg,con) \
  "cmpi r%0,%1\nbeq %a\n" 2
```

uses a `cmpi, beq` sequence to compare a register to a constant and branch to this label if equal.

I removed any remaining 32-bit assumptions inherited from `mips.md`, and arranged to store long `ints` in register pairs, and call helper routines for `mul`, `div`, `rem`, and some shifts.

My port was up and running in just one day, but I had already read the lcc book. Let's see what she can do. Listing 1 is the source for a binary tree search routine, and Listing 2 is the assembly code `lcc-xr16` emits.

INSTRUCTION SET

Now, let's refine the instruction set and choose an instruction encoding. My goals and constraints include: cover C (integer) operator set, fixed-size 16-bit instructions, easily decoded, easily pipelined, with three-operand instructions (`dest = src`$_1$ *`op`* `src`$_2$/`imm`), as encoding space allows. I also want it to be byte addressable (load and store bytes and words), and provide one addressing mode—`disp(reg)`. To support long `ints` we need add/subtract carry and shift left/right extended.

Which instructions merit the most bits? Reviewing early compiler output from test applications shows that the most common instructions (static frequency) are `lw` (load word), 24%; `sw` (store word), 13%; `mov` (reg-reg move), 12%; `lea` (load effective address), 8%; `call`, 8%; `br`, 6%; and `cmp`, 6%. `Mov`, `lea`, and `cmp` can be synthesized from `add` or `sub` with `r0`. 69% of loads/stores use `disp(reg)` addressing, 21% are absolute, and 10% are register indirect.

Therefore we make these choices:

- `add`, `sub`, `addi` are 3-operand
- less common operations (logical ops, add/sub with carry, and shifts) are 2-operand to conserve opcode space
- `r0` always reads as 0
- 4-bit immediate fields
- for 16-bit constants, an optional immediate prefix `imm` establishes the most significant 12-bits of the instruction that immediately follows
- no condition codes, rather use an interlocked compare and conditional branch sequence
- `jal` (jump-and-link) jumps to an effective address, saving the return address in a register
- `call` *`func`* encodes `jal r15,`*`func`* in one 16-bit instruction (provided the function is 16-byte aligned)
- perform mul, div, rem, and variable and multiple bit shifts in software

The six instruction formats are shown in Table 2 and the 43 distinct instructions are shown in Table 3. `adds`, `subs`, `shifts`, and `imm` are uninterruptible prefixes. Loads/stores take two cycles, `jump` and `branch-taken` take three cycles (no branch

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 2—*The xr16 has six instruction formats, each with 4-bit opcode and register fields.*

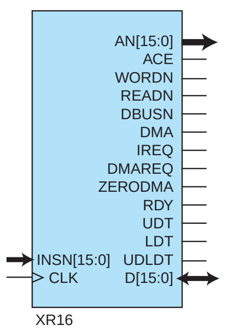

Figure 1—*The xr16 processing symbol ports, which include instruction and data buses, next address and memory controls, and bus controls, constitute its interface to the system memory controller.*

delay slots). The four-bit `imm` field encodes either an `int` (-8–7): `add/sub`, logic, shifts; unsigned (0–15): `lb`, `sb`; or unsigned word displacement (0, 2–30): `lw`, `sw`, `jal`, `call`.

Some assembly instructions are formed from other machine instructions, as you can see in Table 4. Note that only `signed char` data use `lbs`.

ASSEMBLER

I wrote a little multipass assembler to translate the `lcc` assembly output into an executable image.

The xr16 assembler reads one or more assembly files and emits both image and listing files. The lexical analyzer reads the source characters and recognizes tokens like the identifier `_main`. The parser scans tokens on each line and recognizes instructions and operands, such as register names and effective address expressions. The symbol table remembers labels and their addresses, and a fixup table remembers symbolic references.

In pass one, the assembler parses each line. Labels are added to the symbol table. Each instruction expands into one or more machine instructions. If an operand refers to a label, we record a fixup to it.

In pass two, we check all branch fixups. If a branch displacement exceeds 128 words, we rewrite it using a jump. Because inserting a jump may make other branches `far`, we repeat until no `far` branches remain.

Next, we evaluate fixups. For each one, we look up the target address and apply that to the fixup subject word. Lastly, we emit the output files.

I also wrote a simple instruction set simulator. It is useful for exercising both the compiler and the embedded application in a friendly environment.

Well, by now you are probably wondering if there is any hardware to this project. Indeed there is! First, let's consider our target FPGA device.

THE FPGA

The Xilinx XC4005XL-PC84C-3 is a 3.3-V FPGA in an 84-pin J-lead PLCC package. This SRAM-based device must be configured by external ROM or host at power-up. It has a 14 × 14 array of configurable logic blocks (CLBs) and 61 bonded-out I/O blocks (IOBs) in a sea of programmable interconnect.

Every CLB has two 4-input look-up tables (4-LUTs) and two flip-flops. Each 4-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 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.[2]

The XC4000XL architecture would appear to have been designed with CPUs in mind. Just eight CLBs can build a single-port 16 × 16-bit register file (using LUTs as SRAM), a 16-bit adder/subtractor (using carry logic), or a four-function 16-bit logic unit. Because 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's easy to stall the pipeline when necessary. Long line buses and 3-state drivers form an efficient word-wide multiplexer of the many function unit results, and even an on-chip 3-state peripheral bus.

THE PROCESSOR INTERFACE

Figure 1 gives you a good look at the xr16 processor macro symbol. The interface was designed to be easy to use with an on-chip bus. The key signals are the system clock (CLK), next memory address ($AN_{15:0}$), next access is a read (READN), next access is 16-bit data (WORDN), address clock enable: above signals are valid, start next access (ACE), memory ready input: the current access completes this cycle (RDY), instruction word input ($INSN_{15:0}$), on-chip bidi-

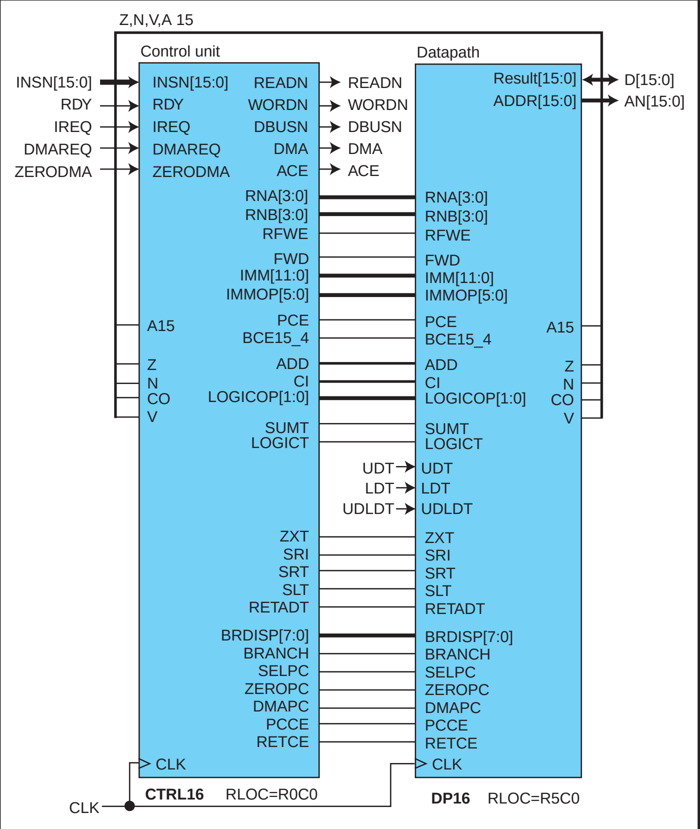

Figure 2—*The control unit receives instructions, decodes them, and drives both the memory control outputs and the datapath control signals.*

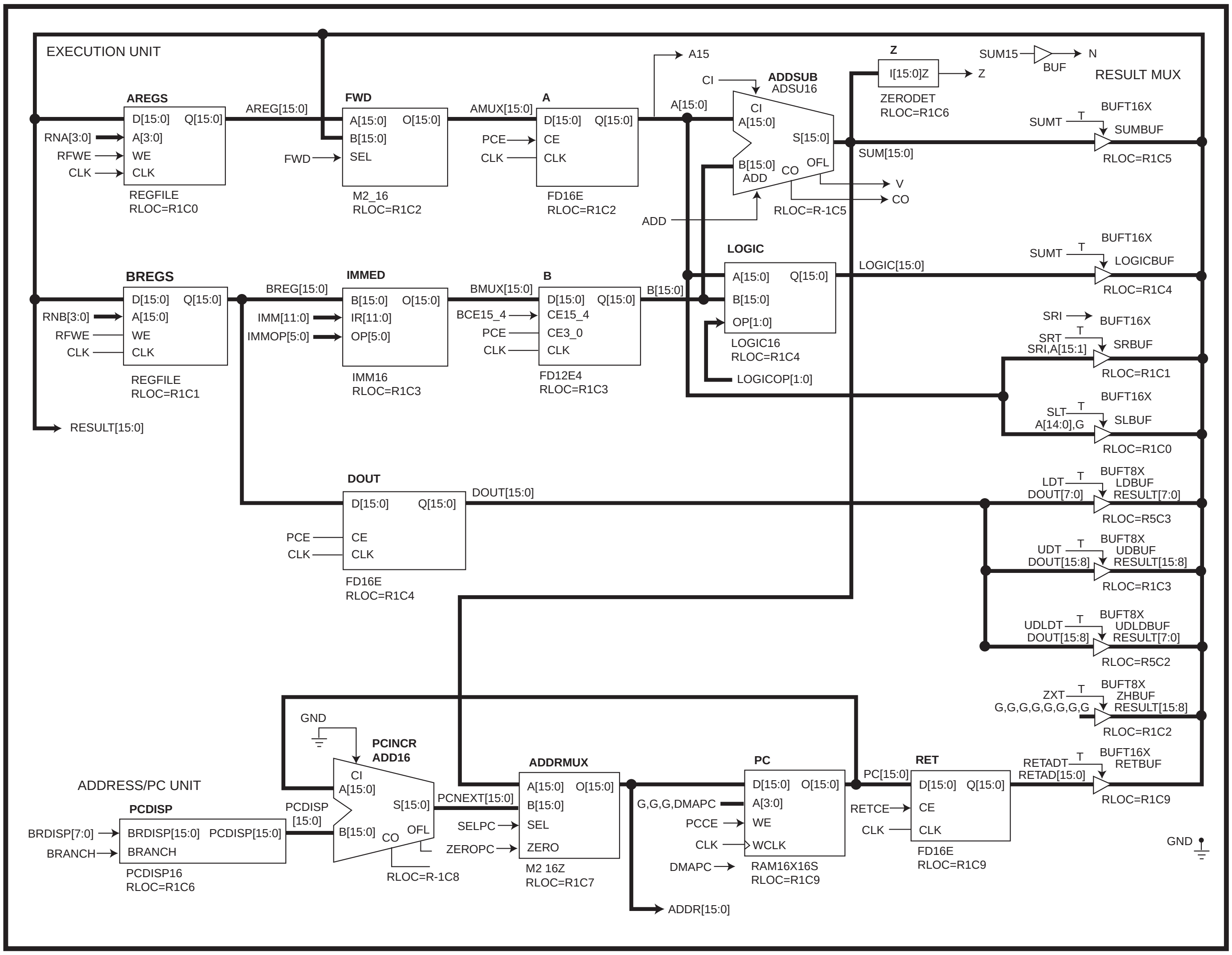

Figure 3—*The pipelined datapath has an execution unit, a result multiplexer, and an address/PC unit. Operands from the register file or immediate field are selected and latched into the A and B operand registers. Then the function units, including ADDSUB, operate upon A and B, and one of the results is driven onto RESULT$_{15:0}$ and written back into the register file. Meanwhile, the address/PC unit increments the PC to help fetch the next instruction.*

rectional data bus to load/store data (D$_{15:0}$).

The memory/bus controller (which I'll explain further in Part 3) decodes the address and activates the selected memory or peripheral. Later it asserts RDY to signal that the memory access is done.

As Figure 2 shows, the CPU is simply a datapath that is steered by a control unit. Next month, I'll examine the control unit in greater detail. The rest of this article explores the design and implementation of the datapath.

DATAPATH RESOURCES

The instruction set evolved with the datapath implementation. Each new idea was first evaluated in terms of the additional logic required and its impact on the processor cycle time.

To execute one instruction per cycle you need a 16-entry 16-bit register file with two read ports (add r3, r1, r2) and one write port (`add r3, r1, r2`); an immediate operand multiplexer (mux) to select the immediate field as an operand (`addi r3, r1, 2`); an arithmetic/logic unit (ALU) (`sub r3, r1, r2`; `xor r3, r1`); a shifter (`srai r3, 1`), and an effective address adder to compute reg+offset (`lw r3, 2(r1)`).

You'll also need a mux to select a result from the adder, logic unit, left or right shifter, return address, or load data; logic to check a result for zero, negative, carry-out, or overflow; a program counter (PC), PC incrementer, branch displacement adder (`br L`), and a mux to load the PC with a jump target address (`call _foo`); and a mux to share the memory port for instruction fetch (addr ← PC) and load/store (addr ← effective address).

Careful design and reuse will let you minimize the datapath area because the adder, with the immediate mux, can do the effective address add, and the PC incrementer can also add branch displacements. The memory address mux can help load the PC with the jump target.

DATAPATH SCHEMATIC

Figure 3 is the culmination of these ideas. There are three groups of resources. The execution unit is the heart of the processor. It fetches operands from the register file and the immediate fields of the instruction register, presents them to the add/sub, logic, and (trivial) shift units, and writes back the result to the register file. The result multiplexer selects one result from the various function units. The address/PC unit drives the next memory address, and includes the PC, PC adder, and address mux. Now, let's see how each resource is implemented in our FPGA.

Hex	Fmt	Assembler	Semantics
0dab	rrr	add rd,ra,rb	rd = ra + rb;
1dab	rrr	sub rd,ra,rb	rd = ra – rb;
2dai	rri	addi rd,ra,imm	rd = ra + imm;
3d*b	rr	{and or xor andn adc sbc} rd,rb	rd = rd *op* rb;
4d*i	ri	{andi ori xori andni adci sbci slli slxi srai srli srxi} rd,imm	rd = rd *op* imm;
5dai	rri	lw rd,imm(ra)	rd = *(int*)(ra+imm);
6dai	rri	lb rd,imm(ra)	rd = *(byte*)(ra+imm);
8dai	rri	sw rd,imm(ra)	*(int*)(ra+imm) = rd;
9dai	rri	sb rd,imm(ra)	*(byte*)(ra+imm) = rd;
Adai	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;
Ciii	i12	call func	r15 = pc, pc = imm12<<4;
Diii	i12	imm imm12	imm'next$_{15:4}$ = imm12;
7xxx	–	reserved	
Exxx	–	reserved	
Fxxx	–	reserved	

Table 3—*The xr16 needs only 43 different instructions to efficiently implement an integer-only subset of the C programming language.*

REGISTER FILE

During each cycle, we must read two register operands and write back one result. You get two read ports (AREG and BREG) by keeping two copies of the 16 × 16-bit register file REGFILE, and reading one operand from each. On each cycle you must write the same result value into both copies.

So, for each REGFILE and each clock cycle you must do one read access and one write access. Each REGFILE is a 16 × 16 RAM. Recall that each CLB has two 4-LUTs, each of which can be a 16 × 1-bit RAM. Thus, a REGFILE is a column of eight CLBs. Each REGFILE also has an internal 16-bit output register that captures the RAM output on the CLK falling edge.

To read and write the REGFILE each clock, you double-cycle it. In the first half of each clock cycle, the control unit presents a read-port source operand register number to the RAM address inputs. The selected register is read out and captured in the REGFILE output register as CLK falls.

In the second half cycle, the control unit drives the write-port register number. As CLK rises, the RESULT$_{15:0}$ is written to the destination register.

OPERAND SELECTION

With the two source registers AREG and BREG in hand, you now select the A and B operands, and latch them in the A and B registers. Some examples are shown in Table 5.

The A operand is AREG unless (as with add$_2$) the instruction depends on the result of the previous instruction. Next month, you'll see why this pipeline data hazard is avoided by forwarding the `add`$_1$ result directly into the A register, just in time for `add`$_2$.

FWD, a 16-bit mux of AREG or RESULT, does this result forwarding. It consists of 16 1-bit muxes, each a 3-input function implemented in a single 4-LUT, and arranged in a column of eight CLBs. The FWD output is captured in the A operand register, made from the 16 flip-flops in the same CLBs. As for the B operand, select either the BREG register file output port or an immediate constant.

For `rri` and `ri` format instructions, B is the zero- or sign-extended 4-bit `imm` field of the instruction register. But, if there's an `imm` prefix, load B$_{15:4}$ with its 12-bit `imm12` field, then load B$_{3:0}$ while decoding the `rri` or `ri` format instruction which follows.

So, the B operand mux IMMED is a 16-bit-wide selection of either BREG, $0_{15:4}||IR_{3:0}$, $sign_{15:4}||IR_{3:0}$, or $IR_{11:0}||0_{3:0}$ ("||" means bit concatenation).

I used an unusual 2-1 mux with a fourth "force constant" input for this zero/sign extension function, primarily because it fits in a single 4-LUT. So, as with FWD, IMMED is an 8-CLB column of muxes.

The B operand register uses IMMED's CLBs 16 flip-flops. The register has separate clock enables for B$_{15:4}$ and B$_{3:0}$, to permit separate loading of the upper and lower bits for an `imm` prefix.

For `sw` or `sb`, read the register to be stored, via BREG, into DOUT$_{15:0}$, another column of eight CLBs flip-flops.

ALU

The arithmetic/logic-unit consists of a 16-bit adder/subtractor and a 16-bit logic unit, which concurrently operate on the A and B registers.

LOGIC computes the 16-bit result of A and B, A or B, A xor B, or A andnot B, as selected by LOGICOP$_{1:0}$. Each logic unit output bit is a function of the four inputs A_i, B_i, and LOGICOP$_{1:0}$, and fits in a single

Listing 2—*Here's the xr16 assembly code (with comments added) that* `lcc` *generates from Listing 1.* `lcc` *has done a good job, although a few register-to-register moves are unnecessary.*

```
_search:  br L3          ; r3=k r4=p
L2:       lw r9,(r4)
          cmp r9,r3      ; p->k < k?
          bge L5
          lw r4,4(r4)    ; p = p->right
          br L6
L5:       lw r4,2(r4)    ; p = p->left
L6:L3:    mov r9,r4
          cmp r9,r0      ; p==0?
          beq L7
          lw r9,(r4)
          cmp r9,r3      ; p->k != k?
          bne L2
L7:       mov r2,r4      ; retval = p
L1:       ret
```

4-LUT. Thus, the 16-bit logic unit is a column of eight CLBs.

ADDSUB adds B to A, or subtracts B from A, according to its ADD input. It reads carry-in (CI) and drives carry-out (CO), and overflow (V). ADDSUB is an instance of the ADSU16 library symbol, and is 10 CLBs high—one to anchor the ripple-carry adder, eight to add/sub 16 bits, and one to compute carry-out and overflow.

Z, the zero detector, is a 2.5-CLB NOR-tree of the $SUM_{15:0}$ output.

The shifter produces either A>>1 or A<<1. This requires no logic, so mux simply selects either SRI || $A_{15:1}$ or $A_{14:0}$ || 0. SRI determines whether the shift is logical or arithmetic.

RESULT MULTIPLEXER

The result mux selects the instruction result from the adder, logic unit, A>>1, A<<1, load data, or return address. You build this 16-bit 7-1 mux from lots of 3-state buffers (TBUFs). In every cycle, the control unit asserts some resource's output enable, driving its output onto the $RESULT_{15:0}$ long line bus that spans the FPGA.

In the third article of this series, I'll share the CPU result bus as the 16-bit on-chip data bus for load/store data. During `sw` or `sb`, the CPU drives $DOUT_{7:0}$ and/or $DOUT_{15:8}$ onto $RESULT_{15:0}$. During `lw` or `lb`, the selected memory or peripheral drives the load data on $RESULT_{15:0}$ or $RESULT_{7:0}$.

ADDRESS/PC UNIT

This unit generates memory addresses for instruction fetch, load/store, and DMA memory accesses. For each cycle, we add PC += 2 to fetch the next instruction. For a taken branch, we add PC += 2×disp8. For `jal` and `call`, we load PC with the effective address SUM from ADDSUB.

Refer to Figure 3 to see how this arrangement works. PCINCR adds PC and the PCDISP mux output (either +2 or the branch displacement) giving PCNEXT. ADDRMUX then selects PCNEXT or SUM as the next memory address.

If the next memory access is an instruction fetch, ADDR ← PCNEXT, and PCCE (PC clock enable) is asserted to update PC with PCNEXT. When the next access is a load/store, SELPC and PCCE are false, and ADDR ← SUM, without updating PC.

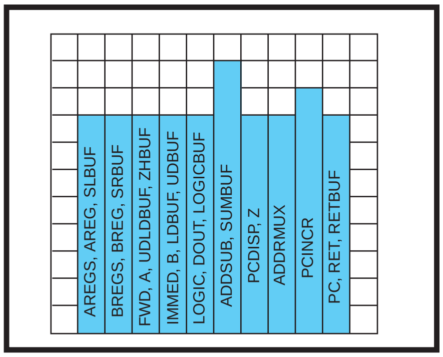

Figure 4—*In the datapath floorplan, RLOC attributes applied to the datapath schematic pin down the datapath elements to specific CLB locations. The $RESULT_{15:0}$ bus runs horizontally across the bottom eight rows of CLBs.*

PCDISP is a 16-bit mux of $+2_{15:0}$ and 2×disp8, 5 CLBs tall. PCINCR is an instance of the ADD16 library symbol, 9 CLBs tall. ADDRMUX is a 16-bit 2-1 mux with a fourth input, ZERO, to set PC to 0 on reset. It's 16 LUTs, 8 CLBs tall.

PC is not a simple register, but rather it is a 16-entry register file. PC_0 is the CPU PC, and PC_1 is the DMA address. PC is a 16 × 16 RAM, eight CLBs tall.

I used RLOC attributes to place the datapath elements. Figure 4 is the resulting floorplan on the 14 × 14 CLB FPGA. Each column of CLBs provides logic, flip-flops, and TBUF resources.

THE DATAPATH IN ACTION

Next, let's see what happens when we run `0008: addi r3,r1,2`. Assuming that PC=6 and r1=10, PCINCR adds PCDISP=2 to PC=6, giving PCNEXT=8. Because SELPC is true, ADDR ← PCNEXT=8, and the next memory cycle reads the word at 0008. Because PCCE is true, PC is updated to 8.

Some time later, RDY is asserted and the control unit latches 0x2312 (`addi r3,r1,2`) into its instruction register. The control unit sets RNA=1, so AREG=`r1`. BREG is not used. FWD is false so A=AREG=`r1`=10. IMMOP is set to sign-extend the 4-bit imm field, and so B=2.

We add A+B=10+2 and as SUMT is asserted (low), we drive SUM=12 onto the RESULT bus. The control unit asserts RFWE (register file write enable), and sets RNA=RNB=3 to write the result into both REGFILEs' `r3`.

DEVELOPMENT TOOLS

This hardware was designed, simulated, and compiled on a PC using the Foundation tools in Xilinx Student Edition 1.5. I used schematics for this project because their 2-D layout makes it easier to understand the data flow because they offer explicit control and because they support the RLOC (relative location) placement attributes that are essential to floorplanning (to achieve the smallest, fastest, cheapest design).

To compile my schematics into a configuration bitstream, Foundation runs these tools:

- `map`: technology mapping—map schematic's arbitrary logic structures into the device's LUTs and flip-flops
- `par`: place and route—place the logic and flip-flops in specific CLBs and then route signals through the programmable interconnect
- `trce`: static timing analysis—enumerate all possible signal paths in the design and report the slowest ones
- `bitgen`: generate a bit stream configuration file for the design

HIGH-PERFORMANCE DESIGN

The datapath implementation showcases some good practices, such as exploiting FPGA features (using embedded SRAM, four input logic

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)	lb rd,imm(ra)
(load-byte,	xori rd,0x80
sign-extending)	subi rd,0x80
j addr	jal r0,addr
ret	jal r0,0(r15)

Table 4—*Many assembly pseudo-instructions are composed from the native instructions. Only rare* `signed char` *data use the rather expensive* `lbs`.

Instruction(s)	A	B
add rd,ra,rb	AREG	BREG
addi rd,ra,i4	AREG	*sign-ext* imm
sb rd,i4(ra)	AREG	*zero-ext* imm
imm 0x123 addi rd,ra,4	ignored AREG	imm12 || $0_{3:0}$ $B_{15:4}$ || imm
add_1 r3,r1,r2 add_2 r5,r3,r4	AREG RESULT	BREG BREG

Table 5—*Depending on the instruction or instruction sequence, A is either AREG or the forwarded result, and B is either BREG or an immediate field of the instruction register.*

structures, TBUFs, and flip-flop clock enables), floorplanning (placing functions in columns, ordering columns to reduce interconnect requirements, and running the 3-state bus horizontally over the columns), iterative design (measuring the area and delay effects of each potential feature), and using timing-driven place-and-route and iterative timing improvement.

I apply timing constraints, such as `net CLK period=28;`, which causes `par` to find critical paths in the design and prioritize their placement and routing to best meet the constraints. Next, I run `trce` to find critical paths. Then I fix them, rebuild, and repeat until performance is satisfactory.

I've built some tools, settled on an instruction set, built a datapath to execute it, and learned how to implement it efficiently in an FPGA. Next month, I'll design the control unit.

Jan Gray is a software developer whose products include a leading C++ compiler. He has been building FPGA processors and systems since 1994, and now he designs for Gray Research LLC. You may reach him at jan@fpgacpu.org.

SOFTWARE

Visit the *Circuit Cellar* web site for more information, including specifications, source code, schematics, and links to related sites.

REFERENCES

[1] C. Fraser and D. Hanson, *A Retargetable C Compiler: Design and Implementation*, Benjamin/Cummings, Redwood City, CA, 1995.

[2] T. Cantrell, "VolksArray," *Circuit Cellar*, April 1998, pp. 82-86.

[3] D. Van den Bout, *The Practical Xilinx Designer Lab Book*, Prentice Hall, 1998. (Available separately and included with Xilinx Student Edition.)

SOURCE

Xilinx Student Edition 1.5
Xilinx, Inc.
(408) 559-7778
Fax: (408) 559-7114
www.xilinx.com