CIRCUIT CELLAR®
THE MAGAZINE FOR COMPUTER APPLICATIONS

Building a RISC System in an FPGA

FEATURE ARTICLE

Jan Gray

Part 1: Tools, Instruction Set, and Datapath

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!

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.*

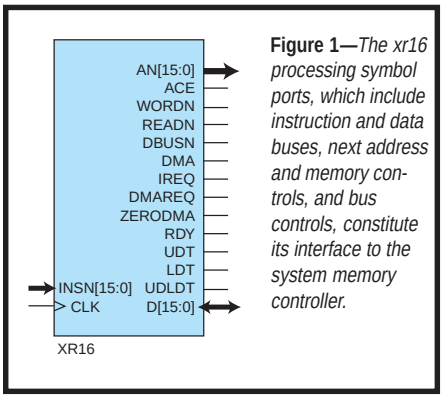

Figure 1—*The xr16 processing symbol ports, which include instruction and data buses, next address and memory controls, and bus controls, constitute its interface to the system memory controller.*

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.

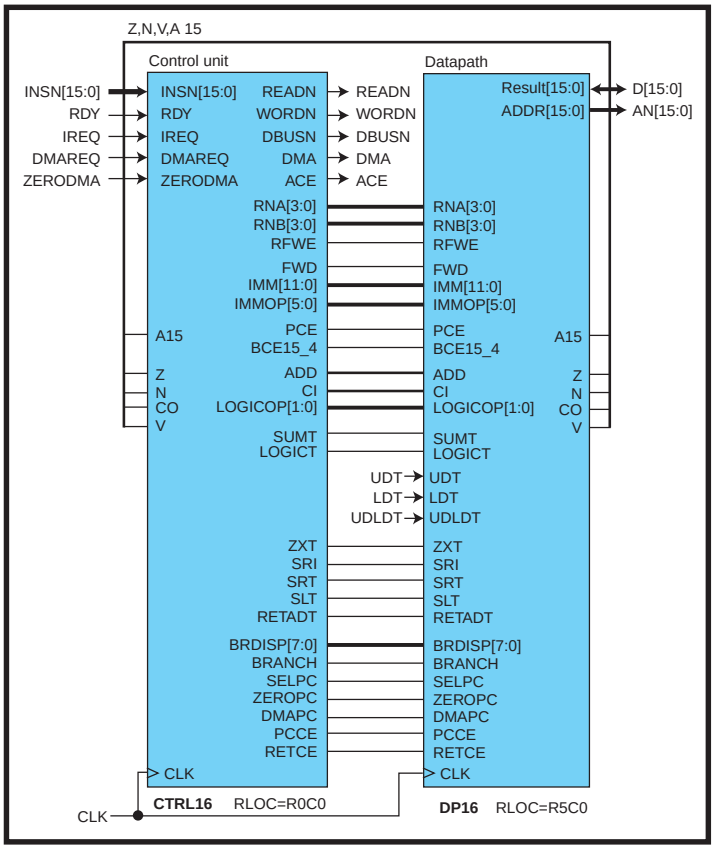

Figure 2—*The control unit receives instructions, decodes them, and drives both the memory control outputs and the datapath control signals.*

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-

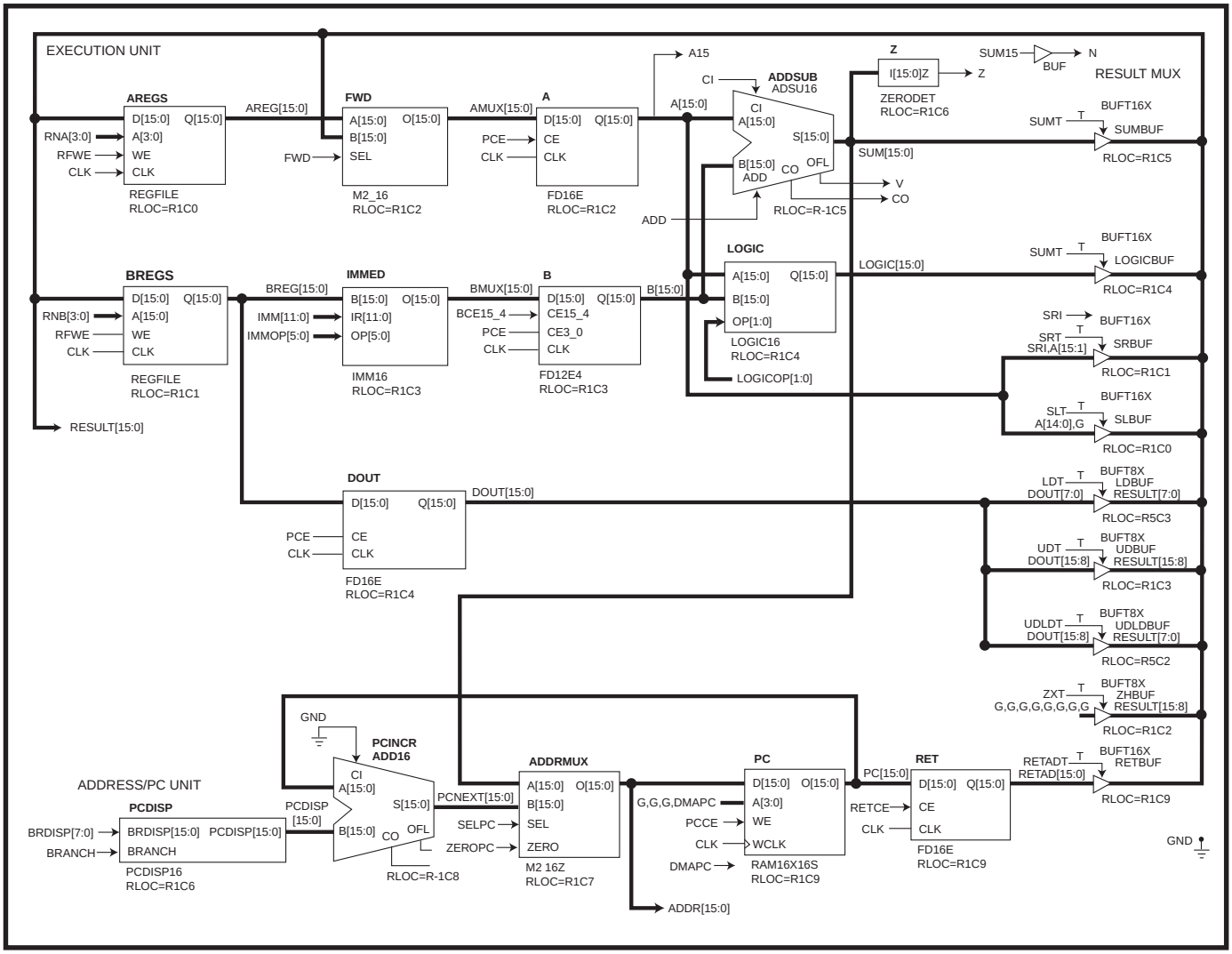

Figure 3—*The pipelined datapath has an execution unit, a result multiplexer, and an address/PC unit. Operands from the register file or immediate field are selected and latched into the A and B operand registers. Then the function units, including ADDSUB, operate upon A and B, and one of the results 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
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 = *(int*)(ra+imm);
6*dai*	rri	lb rd,imm(ra)	rd = *(byte*)(ra+imm);
8*dai*	rri	sw rd,imm(ra)	*(int*)(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;
7*xxx*	–	reserved	
E*xxx*	–	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.

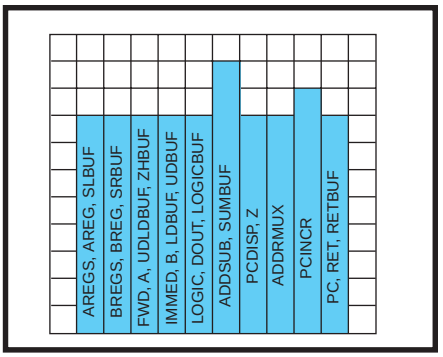

Figure 4—*In the datapath floorplan, RLOC attributes applied to the datapath schematic pin down the datapath elements to specific CLB locations. The $RESULT_{15:0}$ bus runs horizontally across the bottom eight rows of CLBs.*

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) (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 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

CIRCUIT CELLAR®
THE MAGAZINE FOR COMPUTER APPLICATIONS

Building a RISC System in an FPGA

FEATURE ARTICLE

Jan Gray

Part 2: Pipeline and Control Unit Design

In Part 1, Jan introduced his plan to build a pipelined 16-bit RISC processor and System-on-a-Chip in an FPGA. This month, he explores the CPU pipeline and designs the control unit. Listen up, because next month, he'll tie it all together.

Last month, I discussed the instruction set and the datapath of an xr16 16-bit RISC processor. Now, I'll explain how the control unit pushes the datapath's buttons.

Figure 2 in Part 1 (*Circuit Cellar*, 116) showed the CTRL16 control unit schematic symbol in context. Inputs include the RDY signal from the memory controller, the next instruction word $INSN_{15:0}$ from memory, and the zero, negative, carry, and overflow outputs from the datapath.

The control unit outputs manage the datapath. These outputs include pipeline control clock enables, register and operand selectors, ALU controls, and result multiplexer output enables. Before designing the control circuitry, first consider how the pipeline behaves in both good and bad times.

PIPELINED EXECUTION

To increase instruction throughput, the xr16 has a three-stage pipeline—instruction fetch (IF), decode and operand fetch (DC), and execute (EX).

In the IF stage, it reads memory at the current PC address, captures the resulting instruction word in the instruction register IR, and increments PC for the next cycle. In the DC stage, the instruction is decoded, and its operands are read from the register file or extracted from an immediate field in the IR. In the EX stage, the function units act upon the operands. One result is driven through three-state buffers onto the result bus and is written back into the register file as the cycle ends.

Consider executing a series of instructions, assume no memory wait states. In every pipeline cycle, fetch a new instruction and write back its result two cycles later. You simultaneously prepare the next instruction address PC+2, fetch

t_1	t_2	t_3	t_4	t_5
IF_1	DC_1	EX_1		
	IF_2	DC_2	EX_2	
		IF_3	DC_3	EX_3
			IF_4	DC_4

Table 1—*Here the processor fetches instruction I_1 at time t_1 and computes its result in t_3, while I_2 starts in t_2 and ends in t_4. Memory accesses are in boldface.*

instruction I_{PC}, decode instruction I_{PC-2}, and execute instruction I_{PC-4}.

Table 1 shows a normal pipelined execution of four instructions. That's the simple case, but there are several pipeline complications to consider—data hazards, memory wait states, load/store instructions, jumps and branches, interrupts, and direct memory access (DMA).

What happens when an instruction uses the result of the preceding instruction?

Listing 1—*This C code produces assembly code that includes a load I_L and a branch I_B. Each causes pipeline headaches.*

```
if ((p->flags & 7) == 1)
    p->x = p->y;

I :     lw r6,2(r10)   ;load r6 with p->flags
I^L:    andi r6,7      ;is (p->flags & 7)
I^2:    addi r0,r6,-1  ;==1?
I^3:    bne T
I^B:    lw r6,6(r10)   ;yes: load r6 with p->y
 5      ...
```

I_1: andi r1,7
I_2: addi r2,r1,1

Referring to time t_3 of Table 1, EX_1 computes r1=r1&7, while DC_2 fetches the old value of r1. In t_4, EX_2 incorrectly adds 1 to this stale r1.

This is a data hazard, and there are several ways to address it. The assembler can reorder instructions or insert nops to avoid the problem. Or, the control unit can detect the hazard and stall the pipeline one cycle, in order to write-back the result to the register file before fetching it as a source register. However, these techniques hurt performance.

Instead, you do result forwarding, also known as register file bypass. The datapath DC stage includes FWD, a 16-bit 2-1 multiplexer (mux) of AREG (register file port A), and the result bus. Most of the time, FWD passes AREG to the A operand register, but when the control unit detects the hazard (DC source register equals EX destination register), it asserts its FWD output signal, and the A register receives the I_1 result just in time for EX_2 in t_4.

Unlike most pipelined CPUs, the xr16 only forwards results to the A operand—a speed/area tradeoff. The assembler handles any rare port B data hazards by swapping A and B operands, if possible, or inserting nops if not.

t_1	t_2	t_3	t_4	t_5
IF_1	DC_1	EX_1	*EX_1*	
	IF_2	DC_2	*DC_2*	EX_2
		IF_3	***IF_3***	DC_3
				IF_4

Table 2—*During t_3, the instruction fetch memory access of I_3 is not RDY, so the pipeline registers do not clock, and the pipeline stalls until RDY is asserted in t_4. Repeated pipeline stages are italicized.*

MEMORY ACCESSES

The processor has a single memory port for reading instructions and loading and storing data. Most memory accesses are for fetching instructions. The processor is also the DMA engine, and a video refresh DMA cycle occurs once every eight clocks or so. Therefore, in any given clock cycle, the processor executes either an instruction fetch memory cycle, a DMA memory cycle, or a load/store memory cycle.

Memory transactions are pipelined. In each memory cycle, the processor drives the next memory cycle's address and control signals and awaits RDY, indicating the access has been completed. So, what happens when memory is not ready?

The simplest thing to do is to stop the pipeline for that cycle. CTRL deasserts all pipeline register clock enables PCE, ACE, and so forth. The pipeline registers do not clock, and this extends all pipeline stages by one cycle. In Table 2, memory is not ready during the fetch of instruction I_3 in t_3, and so t_4 repeats t_3. (Repeated pipe stages are italicized.)

I_L in Listing 1 is a load word instruction. Loads and stores need a second memory access, causing pipeline havoc (see Table 3). In t_4 you must run a load data access instead of an instruction fetch. You must stall the pipeline to squeeze in this access.

Then, although you fetched I_3 in t_3, you must not latch it into the instruction register (IR) as t_3 ends, because neither EX_L nor DC_2 are finished at this point. In particular, DC_2 must await the load result in order to forward it to A, because I_2 uses r6—the result of I_L!

Finally, if (in t_3) you don't save the just-fetched I_3 somewhere, you'll lose it, because in t_4, the memory port is busy with the load cycle. If you lose it, you'll have to re-fetch it no sooner than t_5, with the result that even a no-wait load requires three cycles, which is unacceptable.

To fix this problem, the control unit has a 16-bit NEXTIR register and an IR source multiplexer (IRMUX). In t_3, it captures I_3 in NEXTIR, and then in t_4, IR is loaded from NEXTIR instead of from the memory port (which is busy with the load). NEXTIR ensures a two-cycle load or store, at a cost of eight CLBs.

As with instruction fetch accesses, load/store memory accesses may have to wait on slow memory. For example, had RDY not been asserted during t_4, the pipeline would have stalled another cycle to wait for EX_L access to complete.

BRANCHING OUT

Next, consider the effect of jumps (`call` and `jal`) and taken branches. By the time you execute the jump or taken branch I_J during EX_J (updating PC), you'll have decoded I_{J+1} and fetched I_{J+2}. These instructions in the branch shadow (and their side effects) must be annulled.

Continuing the Table 3 example from time t_5, and assuming the branch is taken at t_7, you must annul the EX_5 stage of I_5, and the DC_6 and EX_6 stages of I_6. (Annulled stages are struck

t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9
IF_L	DC_L	EX_L	***EX_L***					
	IF_2	DC_2	*DC_2*	EX_2				
		IF_3	*IF_3*	DC_3	EX_3			
			IF_B	DC_B	EX_B			
				IF_5	DC_5	~~EX_5~~		
					IF_6	~~DC_6~~	~~EX_6~~	
						IF_T	DC_T	

Table 3—*Pipelined execution of the load instruction I_L, I_2, I_3, the branch I_B, the annulled I_5 and I_6, and the branch target I_T. During t_4 you stall the pipeline for the I_L load/store memory cycle. The branch I_B executed in t_7 causes I_5 and I_6 to be annulled in t_8 and t_9. Annulled instructions are struck through.*

through). Execution continues at instruction I_T. T_9 is not an EX_5 load cycle, because the I_5 load is annulled.

Because you always annul the two branch shadow instructions, jumps and taken branches take three cycles. Jumps also save the return address in the destination register. This return address is obtained from the datapath's RET register, which holds the address of the instruction in the DC pipeline stage.

INTERRUPTS

When an interrupt request occurs, you must jump to the interrupt handler, preserve the interrupt return address, retire the current pipeline, execute the handler, and later return to the interrupted instruction.

When INTREQ is asserted, you simply override the fetched instruction with `int`, that is, `jal r14,10(r0)` via the IRMUX. This jumps to the interrupt handler at 0x0010 and leaves the return address in r14, which is reserved for this purpose. When the handler has completed, it executes `iret`, (i.e, `jal r0,0(r14)`) and exection resumes with the interrupted instruction.

There are two pipeline issues here. First, you must not interrupt an interlocked instruction sequence (any `add`, `sub`, `shift`, or `imm` followed by another instruction). If an interlocked instruction is in the DC stage, the interrupt is deferred one cycle.

Secondly, the `int` must not be inserted in a branch or jump shadow, lest it be annulled. If a branch or jump is in the DC stage, or if a taken branch or jump is in the EX stage, the interrupt is deferred.

The simplicity of the process pays off once again. The time to take an interrupt and then return from a null interrupt handler is only six cycles.

You might be wondering about the interrupt priorities, non-maskable interrupts, nested interrupts, and interrupt vectors. These artifacts of the fixed-pinout era need not be hardwired into our FPGA CPU. They are best done by collaboration with an on-chip interrupt controller and the interrupt handler software.

The last pipeline issue is DMA. The PC/address unit doubles as a DMA engine. Using a 16 × 16 RAM as a PC register file, you can fetch either an instruction (AN ← PC_0 += 2) or a DMA word (AN ← PC_1 += 2) per memory cycle.

After an instruction is fetched, if

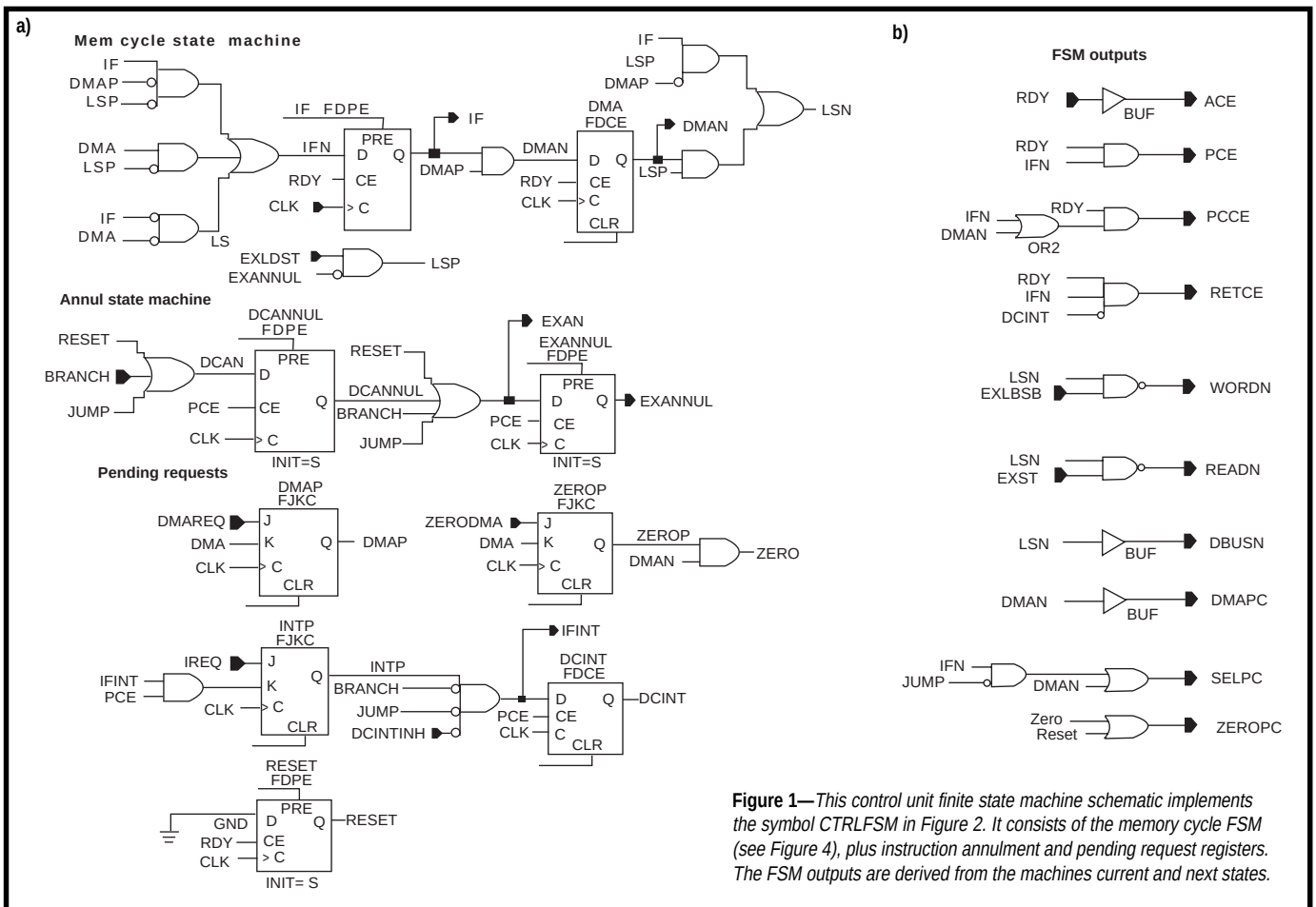

Figure 1—*This control unit finite state machine schematic implements the symbol CTRLFSM in Figure 2. It consists of the memory cycle FSM (see Figure 4), plus instruction annulment and pending request registers. The FSM outputs are derived from the machines current and next states.*

RNA	When
RA	DC: add sub addi lw lb sw sb jal
RD	DC: *all rr, ri format*
0	DC: call
EXRD	EX: *all but call*
15	EX: call

RNB	When
RB	DC: add sub, *all rr fmt*
RD	DC: sw sb
EXRD	EX: *all but call*
15	EX: call

Table 4—*RNA and RNB control the A and B ports of the register file. While CLK is high, they select which registers to read, based upon register fields of the instruction in the DC stage. While CLK is low, they select which register to write, based upon the instruction in the EX stage.*

DMAREQ has been asserted, you insert one DMA memory cycle.

This PC register file costs eight CLBs for the RAM, but saves 16 CLBs (otherwise necessary for a separate 16-bit DMA address counter and a 16-bit 2-1 address mux), and shaves a couple of nanoseconds from the system's critical path. It's a nice example of a problem-specific optimization you can build with a customizable processor.

To recap, each instruction takes three pipeline cycles to move through the instruction fetch, operand fetch and decode, and execute pipeline stages. Each pipeline cycle requires up to three memory access cycles (mandatory instruction fetch, optional DMA, and optional EX stage load or store). Each memory access cycle requires one or more clock cycles.

CONTROL UNIT DESIGN

Now that you understand the pipeline, you are ready to design the control unit. (For more information on RISC pipelines, see *Computer Organization and Design: The Hardware/Software Interface,* by Patterson and Hennessy.) [1] First, some important naming conventions. Some control unit signal names have prefixes and suffixes to recognize their function or context (most signal names sans prefix are DC stage signals):

- N*sig*: *not* signal—signal inverted
- DC*sig*: a DC stage signal
- EX*sig*: an EX stage signal
- *sig*N: signal in "next cycle"—input to a flip-flop whose output is *sig*
- *sig*CE: flip-flop clock enable
- *sig*T: active low 3-state buffer output enable

Each instruction flows through the three stages (IF, DC, and EX) of the control unit (see Figure 2) pipeline. In the *IF* stage, when the instruction fetch read completes, the new instruction at $INSN_{15:0}$ is latched into IR.

In the *DC stage*, DECODE decodes IR to derive internal control signals. In the first half clock cycle, CTRL drives $RNA_{3:0}$ and $RNB_{3:0}$ with the source registers to read, and drives FWD and $IMM_{5:0}$ to select the A and B operands. If the instruction is a branch, CTRL determines if it is taken. Then as the pipeline advances, the instruction passes into EXIR.

In the *EX stage*, CTRL drives ALU and result mux controls. If the in-

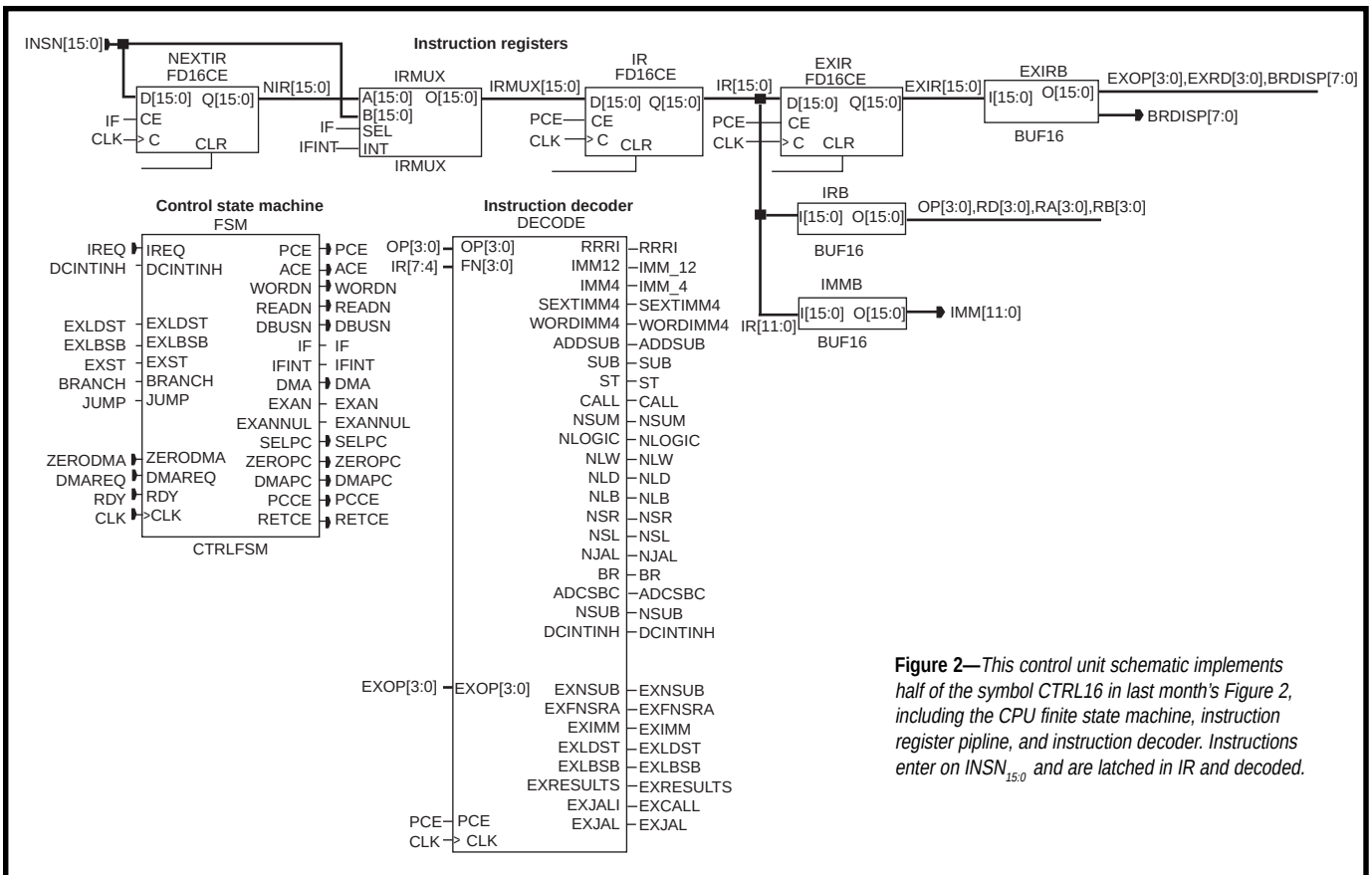

Figure 2—*This control unit schematic implements half of the symbol CTRL16 in last month's Figure 2, including the CPU finite state machine, instruction register pipline, and instruction decoder. Instructions enter on $INSN_{15:0}$ and are latched in IR and decoded.*

struction is a load/store, it inserts a memory access. In the last half cycle, RNA and RNB both drive the destination register number to store the result into the register file.

Let's consider each part of the control finite state machine (see Figure 1). The control FSM has three states:

- IF: current memory access is an instruction fetch cycle
- DMA: current access is a DMA cycle
- LS: current access is a load/store

Figure 4 shows the state transition diagram. The FSM clocks when one memory transaction completes and another begins (on RDY). CTRLFSM also has several other bits of state:

- DCANNUL: annul DC stage
- EXANNUL: annul EX stage
- DCINT: `int` in DC stage
- DMAP: DMA transfer pending
- INTP: interrupt pending

DCANNUL and EXANNUL are set after executing a jump or taken branch. They suppress any effects of the two instructions in the branch shadow, including register file write-back and load/store memory accesses. So, an annulled add still fetches and adds its operands, but its results are not retired to the register file.

DCINT is set in the pipeline cycle following the insertion of the `int` instruction. It inhibits clocking of RET for one cycle, so that the `int` picks up the return address of the interrupted instruction rather than the instruction after that.

The highest fan-out control signal is PCE, the pipeline clock enable. Most datapath registers are enabled by PCE. It indicates that all pipe stages are ready and the pipeline can advance. PCE is asserted when RDY signals completion of the last memory cycle in the current pipeline cycle. If memory isn't ready, PCE isn't asserted, and the pipeline stalls for one cycle.

The control FSM also takes care of managing the memory interface via the following signals:

Enable	Instruction	Source
SUMT	add sub addi adc sbc adci sbci	$SUM_{15:0}$
LOGICT	and or xor andn andi ori xori andni	$LOGIC_{15:0}$
SLT	slli	$A_{14:0} \| \| 0$
SRT	srli srai	$SRI \| \| A_{15:1}$
ZXT	lb	$0_{15:8}$
RETADT	jal call	$RETAD_{15:0}$
none	sw sb br* imm	—

Table 5—*Here's a look at the result multiplexer output enable controls. The instruction determines which enable is asserted and which function unit drives* $RESULT_{15:0}$.

- RDY: memory cycle complete (input from the memory controller)
- READN: next memory cycle is a read transaction—true except for stores
- WORDN: next cycle is 16-bit data—true except for byte loads/stores
- DBUSN: next cycle is a load/store, and it needs the on-chip data bus
- ACE (address clock enable): the next address $AN_{15:0}$ (a datapath output) and the above control outputs are all valid, so start a new memory transaction in the next clock cycle. ACE equals RDY, because if memory is ready, the CPU is always eager to start another memory transaction.

There are no IF stage control outputs. Internal to the control unit, three signals control IF stage resources. Those three signals are:

  - PCE: enable IR and EXIR clocking
  - IF: asserted in an instruction fetch memory cycle
- IFINT: force the next instruction to be `int = jal r14,10(r0) =` 0xAE01

If a DMA or load/store access is pending, IF enables NEXTIR to capture the previously fetched instruction (take a look back at time t_3 in Table 3). Otherwise, the instruction fetch is the only memory access in the pipe stage. So, IF is then asserted with PCE, and IRMUX selects the $INSN_{15:0}$ input as the next instruction to complete.

DECODE STAGE

The greater part of the control unit operates in the DC stage. It must decode the new instruction, control the register file, the A and B operand multiplexers, and prepare most EX stage control signals.

The instruction register IR latches the new instruction word as the DC stage begins. The buffers IRB and IMMB break out the instruction fields OP, RD, and so forth—$IR_{15:12}$ is renamed $OP_{3:0}$ and so on (the tools optimize away these buffers).

The instruction decoder DECODE is simple. It is a set of 30 ROM 16x1s, gate expressions, and a handful of flip-flops. Each ROM inputs $OP_{3:0}$ or $EXOP_{3:0}$ and outputs some decoded signal. The decoder is relatively compact because xr16 has a simple instruction set, and its 4-bit opcodes are a good match for the FPGA's 4 LUTs.

The register file control signals, shared by both the DC and EX stages, are $RNA_{3:0}$: port A register number; $RNB_{3:0}$: port B register number; and RFWE: register file write enable.

Next cycle	Next address	Outputs
IF	$AN \leftarrow PC_0 \mathrel{+}= 2$	SELPC PCCE
IF *branch*	$AN \leftarrow PC_0 \mathrel{+}= 2\times disp8$	BRANCH SELPC PCCE
IF *jal call*	$AN \leftarrow PC_0 = SUM$	PCCE
IF*reset*	$AN \leftarrow PC_0 = 0$	SELPC ZEROPC PCCE
LS *load/store*	$AN \leftarrow SUM$	—
DMA	$AN \leftarrow PC_1 \mathrel{+}= 2$	SELPC DMAPC PCCE
DMA *reset*	$AN \leftarrow PC_1 = 0$	SELPC ZEROPC DMAPC PCCE

Table 6—*Here's a look at the result multiplexer output enable controls. The instruction determines which enable to assert and thus determines which function unit drives the RESULT bus.*

With CLK high, CTRL drives RNA and RNB with the DC stage instruction's source register numbers. With CLK low, CTRL drives RNA and RNB with the EX stage destination register number.

RFWE is asserted with PCE when there is a result to write back. It is false for instructions, which produces no result (immediate prefix, branch, or store) for annulled instructions, and for destination r0.

The muxes RNA and RNB produce $RNA_{3:0}$ and $RNB_{3:0}$, as shown in Table 4, as selected by decode outputs RRRI, CALL, ST, EXCALL, and CLK. `Call` is irregular. It computes r15 = pc, pc = r0 + imm12<<4, and the registers r15 and r0 are implicit.

The FWD signal causes RESULT to be forwarded into A, overriding AREG. CTRL asserts FWD when the EX stage destination register equals the DC stage source register A (detected within RNA), unless the EX stage instruction is annulled or its destination is r0.

Last month, I discussed IMMED, the BREG/immediate operand mux. $IMMOP_{5:0}$ controls IMMED, based upon the decoder outputs WORDIMM, SEXTIMM4, IMM_12, and IMM_4.

$B_{3:0}$ is clock enabled on PCE, but $B_{15:4}$ uses B15_4CE. B15_4CE is PCE, unless the EX stage instruction is `imm`. Thus, the `imm` prefix establishes $B_{15:4}$, and the subsequent immediate operand instruction provides $B_{3:0}$ only.

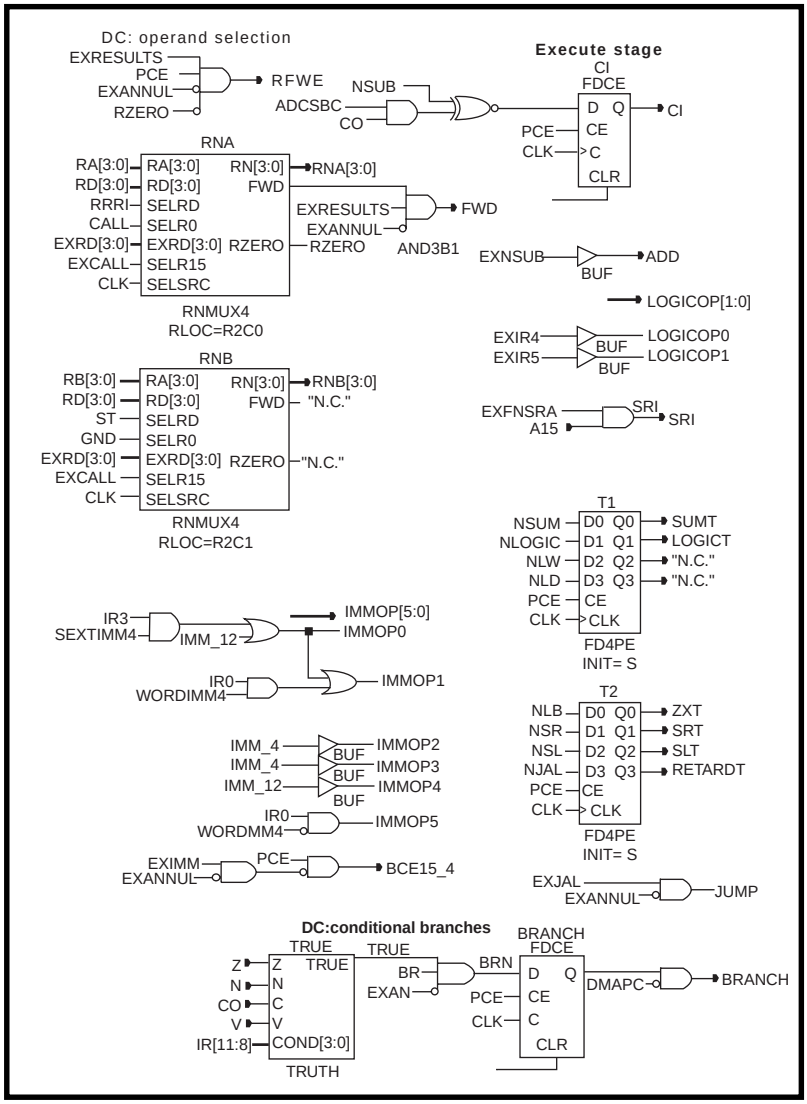

Figure 3—*The remainder of the control unit schematic implements the DC stage operand selection logic including register file, immediate operand control, branch logic, EX stage ALU, and result mux controls.*

Now, turning to conditional branches, if the DC stage instruction is a branch, then the EX stage instruction must be `add`, `sub`, or `addi`, which drives the control unit's condition inputs Z (zero), N (negative), CO (carry-out), and V (overflow).

Late in the DC stage, the TRUE macro evaluates whether or not the branch condition COND is true with respect to the condition inputs. If so, and if the branch instruction is not annulled, the BRANCH flip-flop is set. Therefore, as the pipeline advances and the branch instruction enters the EX stage, the BRANCH control output is asserted. This directs PCINCR to take the branch by adding 2×disp8 to the PC.

THE EXECUTE STAGE

Now, let's discuss the EX stage ALU, result mux, and address unit controls. The ALU and shift control outputs are:

- ADD: set unless the instruction is `sub` or `sbc`
- CI: carry-in. 0 for `add` and 1 for `sub`, unless it's `adc` or `sbc` where we XOR in the previous carry-out
- $LOGICOP_{1:0}$: select `and`, `or`, `xor`, or `andn`. $LOGICOP_{1:0}$ is simply $EXIR_{5:4}$ (i.e., EX stage copy of $FN_{1:0}$)
- SRI: shift right input—0 for `srli` and A_{15} for `srai` (shift right arithmetic)

`slxi` and `srxi` (shift extended left/right for multi-word shift support) are not yet implemented. Be my guest!

The result mux control outputs SUMT, LOGICT, SLT, SRT, SXT, and RETADT are active low RESULT bus 3-state output enables. Each cycle, all EX stage function units produce results. One asserted *T* enables its unit's 3-state buffers to drive the RESULT bus, as shown in Table 5.

ZXT zeroes $RESULT_{15:8}$ during `lb`. As you'll see next month, the system drives $RESULT_{7:0}$ with the byte load result.

The following outputs control the address unit:

- BRANCH: if set, add 2×disp8 to PC, otherwise add +2
- SELPC: if set, next address is $PCNEXT_{15:0}$, otherwise $SUM_{15:0}$
- ZEROPC: if set, next address is 0
- PCCE (PC clock enable): update PC_i

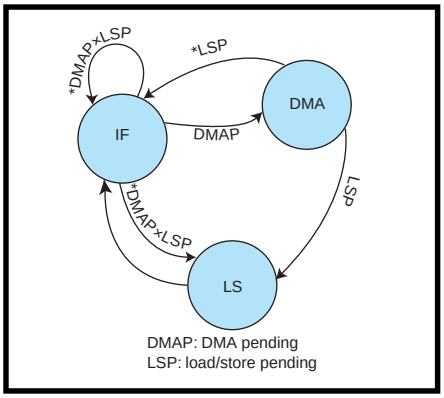

Figure 4—*Each memory cycle is an instruction fetch unless there is a DMA transfer pending or the EX stage instruction is a load or store. The FSM clocks when one memory transaction completes and another begins (on RDY).*

- DMAPC: if set, fetch and update PC_1 (DMA address), otherwise PC_0 (PC)

Depending on the next memory cycle and the current EX stage instruction, the control unit selects the next address by asserting certain combinations of control outputs (see Table 6).

WRAP-UP

This month, we considered pipelined processor design issues and explored the detailed implementation of our xr16 control unit—and lived! The CPU design is complete. The final article in this series tackles the design of this system-on-a-chip.

Jan Gray is a software developer who produces embedded computers. He has been building FPGA processors and systems since 1994, and he now 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.

REFERENCE

[1] D. Patterson and J. Hennessy, *Computer Organization and Design: The Hardware/Software Interface,* Morgan Kaufmann, San Mateo, CA, 1994.

CIRCUIT CELLAR®
THE MAGAZINE FOR COMPUTER APPLICATIONS

Building a RISC System in an FPGA

FEATURE ARTICLE

Jan Gray

Part 3: System-on-a-Chip Design

Now that the xr16 RISC processor is complete, it's time to tie everything together and wrap up this series. In this final part, Jan designs a demo system that includes an on-chip bus, memory controller, video controller, and peripherals.

The xr16 RISC processor is designed, now it's time to design the rest of the System-on-a-Chip (SoC). Besides the CPU, the FPGA hosts an on-chip bus, bus controller, parallel port, RAM, video controller, and an external SRAM controller.

This month, I'll show how simple interfaces can make SoC design as straightforward as classic CPU, glue logic, memory, peripherals, and PCB design used to be.

XS40 BOARD

The project targets the XESS XS40-005XL V.1.2 FPGA board in Photo 1, which includes a Xilinx XC4005XL, 12-MHz oscillator (see Figure 1), 32-KB SRAM, 8031 MCU, 7-segment LED, voltage regulators, and parallel port and VGA port connectors. It's simple, inexpensive, and is featured in *The Practical Xilinx Designer Lab Book* included with Xilinx Student Edition.

I chose this board because it is well supported with documentation and tools, and because it can be used for both the XSE exercises and this project.

A SYSTEM-ON-A-CHIP

I'll build an integrated system from the resources at hand—the FPGA, RAM, the video and parallel ports, and the 12-MHz oscillator.

I used the RAM for program, data, and video memory. The byte-wide, asynchronous SRAM isn't ideal, but it is fast enough for you to read and latch a byte on each clock edge, thereby fetching a 16-bit instruction during each cycle.

By displaying all 32 KB of RAM, you can fashion a bitmapped 576 × 455 monochrome video display at VGA-compatible sync frequencies. How quaint, to watch every bit on screen!

Refer also to Figure 4, the FPGA top-level schematic. It includes the

Address	Resource
0000-7FFF	external 32-KB RAM, video frame buffer
0000	reset handler
0010	interrupt handler
FF00-FFFF	I/O control registers, 8 peripherals × 32 bytes
FF00-FF1F	0: 16-word on-chip IRAM
FF21	1: parallel port input byte
FF41	2: parallel port output byte
FF60-FF7F	3: *unused*
…	…
FFE0-FFFF	7: *unused*

Table 1—*The system memory map includes eight decoded peripheral control register address blocks.*

processor (P), the system memory/bus controller (MEMCTRL), the on-chip 16-bit data bus ($D_{15:0}$), on-chip peripherals (PARIN, PAROUT, and IRAM), the external SRAM interface, and the VGA video controller.

DECISIONS, DECISIONS

Before examining the design, let's briefly explore the on-chip bus design space. (This is not the sort of thing you worry about when designing to someone else's microprocessor, but in an FPGA SoC, you have a little more freedom.)

Bus design issues include how many bus masters are permitted, how is the bus clocked and pipelined, how wide is it, does it provide byte addressing, and is it split or unified with the processor core RESULT bus.

For XSOC, the pipelined on-chip 16-bit data bus $D_{15:0}$ is single-mastered (but recall the CPU also performs DMA transfers), the bus clock is the CPU clock, and the on-chip data bus is unified with the processor's $RESULT_{15:0}$ data bus. All of these design decisions help to keep this project simple.

BUS CONTROLS

MEMCTRL, the system bus/memory controller, interfaces the processor to the on-chip and off-chip peripherals. It receives the pipelined "next transaction" memory request signals $AN_{15:0}$, WORDN, READN, DBUSN, and ACE from the CPU. Then, it decodes the address, enables some peripheral or memory, and later asserts RDY in the clock cycle in which the memory cycle completes. I/O registers are memory mapped (see Table 1).

There are eight transaction types: (external RAM or I/O) × (read or write) × (byte or word), all decoded from $AN_{15:0}$, WORDN, and READN.

MEMCTRL manages transfers on the on-chip data bus $D_{15:0}$ and the external data bus $XD_{7:0}$ by asserting various tri-state output enables (*x*T) and control register clock enables (*x*CE). These enable signals are asserted according to the transaction type (see Table 3).

For example, during `sw r0,` `0xFF00`, MEMCTRL decodes an I/O write word request. It asserts LDT and UDT, driving the store data onto $D_{15:0}$, and asserts IRAM/LCE and IRAM/UCE, writing $D_{15:0}$ into IRAM's SRAMs:

$$IRAM/D_{15:0} := D_{15:0} \leftarrow DOUT_{15:0}$$

Next, consider a store to external RAM: `sw r0,0x0100`. Because the external data bus is only eight bits wide, first store the least significant byte, then the most significant byte. First, MEMCTRL asserts LDT and XDOUTT:

$$XD_{7:0} \neg D_{7:0} \neg DOUT_{7:0}$$

Later, it asserts UDLDT and XDOUTT:

$$XD_{7:0} \leftarrow D_{7:0} \leftarrow DOUT_{15:8}$$

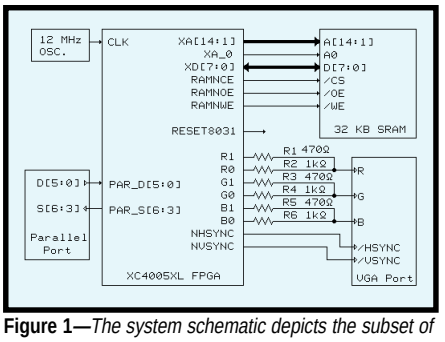

Figure 1—*The system schematic depicts the subset of the XS40 needed for our project. The 8031 (not shown) is held in reset.*

BUS INTERFACE

Now, let's design an on-chip bus peripheral interface to enable robust and easy reuse of peripheral cores and to prepare for an ecology of interoperable cores to come.

It helps to distinguish between core users and core designers. The former are more numerous, while the latter are more experienced. Therefore, I make ease-of-use tradeoffs in favor of core users.

Because FPGAs are malleable and FPGA SoC design is so new, I wanted an interface that can evolve to address new requirements without invalidating existing designs.

With these two considerations in mind, I borrowed a few ideas from the software world and defined an abstract control signal bus with all of the common control signals collected into an opaque bus $CTRL_{15:0}$.

MEMCTRL drives CTRL and also does I/O address decoding, driving the eight I/O selects $SEL_{7:0}$.

Now, you need only instantiate the core, attach CLK, CTRL, D, some SEL_i, any core-specific inputs and outputs, and you're done!

Contrast this with interfacing to a traditional peripheral IC. Each IC has its own idiosyncratic set of control signals, I/O register addresses, chip selects, byte read and write strobes, ready, interrupt request, and such. They don't call it glue logic for nothing.

Of course, we can't just sweep all the complexity under the rug. Each core must decode CTRL and recover the relevant control signals. This is done with the DCTRL (CTRL decoder) macro (see Figure 5). DCTRL inputs SEL_i, $CTRL_{15:0}$, and CLK and outputs local I/O register address, upper and lower byte output enables (read strobes), and clock enables (write strobes).

Within each DCTRL instance, you do final address decoding for the specific peripheral, combining its SEL_i signal with the I/O select within $CTRL_{15:0}$. Here XIN8 only uses LDT (the LSB output enable). The other DCTRL outputs are unloaded and automatically eliminated by the FPGA implementation tools.

Using DCTRL and the on-chip tri-state bus, the typical overhead per peripheral is only one or two CLBs, and perhaps a column of TBUFs.

Control signal abstraction can also make bus interface evolution easy. If you revise MEMCTRL and DCTRL together, arbitrary changes to $CTRL_{15:0}$ can be made without invalidating any

Enable	Effect
LDT	$D_{7:0} \leftarrow DOUT_{7:0}$
UDT	$D_{15:8} \leftarrow DOUT_{15:8}$
UDLDT	$D_{7:0} \leftarrow DOUT_{15:8}$
XDOUTT	$XD_{7:0} \leftarrow D_{7:0}$
LXDT	$D_{7:0} \leftarrow XDIN_{7:0}$
UXDT	$D_{15:8} \leftarrow XDIN_{15:8}$
*p*LDT	$D_{7:0} \leftarrow p/D_{7:0}$
*p*UDT	$D_{15:8} \leftarrow p/D_{15:8}$
*p*LCE	$p/D_{7:0} := D_{7:0}$
*p*UCE	$p/D_{15:8} := D_{15:8}$

Table 2—*There are a set of enables* p/* *within each peripheral.* $DOUT_{15:0}$ *is the CPU store data output register (see Part 1,* Circuit Cellar *116).*

existing designs. And, to add new bus features, simply design a new decoder DCTRL_v2, causing no changes to existing DCTRL clients.

EXTERNAL I/O INTERFACE?

There isn't one. If it were necessary to attach external peripherals, perhaps to the $XD_{7:0}$ bus, you might design some on-chip external peripheral adapter macros. Just like an on-chip peripheral, each adapter would take CTRL and some SEL_i, but its job would be to use additional I/O pins to control its peripheral IC's chip selects and so forth. Of course, as a $CTRL_{15:0}$ client, it would be able to raise interrupts, insert wait states, and so forth.

EXTERNAL RAM

The external RAM is a classic 32-KB fast asynchronous SRAM with a 15-ns access time (t_{AA}). Its pins include A14:0 (address), D7:0 (data in/out), /CS (chip select), /WE (write enable), and /OE (output enable).

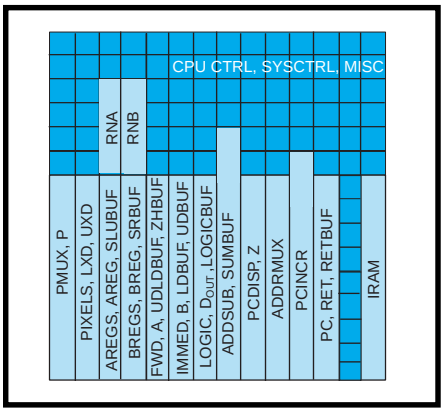

Figure 3—*The rest of the device contains the automatically placed processor control unit and other logic.*

Refer to Figure 2 and the external bus and SRAM interface block of Figure 5.

$XA_{14:1}$ is 14 IOBs configured as OFDXs (output flip-flops with clock enables). $XA_{14:1}$ captures the next address $AN_{14:1}$ at the start of each new memory transaction. XA_0 (XA_0) is the least significant bit of the external address. It is a logic output and can change on either CLK edge.

$XD_{7:0}$ is eight IOBs configured as eight sets of simultaneous OBUFTs (tri-state output buffers), IBUFs (input buffers), and IFDs (input flip-flops).

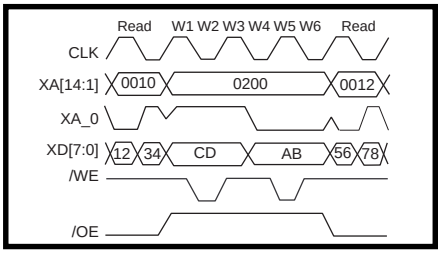

Figure 2—*The RAM interface signals for three memory transactions are: read 1234 from address 0010, write ABCD to address 0200, and read 5678 from address 0012.*

During a RAM write, XDOUTT is asserted, RAMNOE is deasserted, and the OBUFTs drive $D_{7:0}$ out onto $XD_{7:0}$.

During a RAM read, XDOUTT is deasserted, RAMNOE is asserted, and the RAM drives its output data onto $XD_{7:0}$. The data is input through the IBUFs and latched in the XDIN IFDs (on each falling CLK edge).

To keep the CPU busy with fresh new instructions, the system reads both bytes of a 16-bit word in one cycle. In the first half cycle, it sets XA_0=0, reading the MSB, and latches it in XDIN. In the second half cycle, the system sets XA_0=1, reading the LSB, and reads it through IBUFs. The catenation of these two bytes, $XDIN_{15:0}$, feeds the CPU's INSN port, the video controller's PIX port, and $D_{15:0}$ via the byte-wide tri-state buffers LXD and UXD.

Writes to asynchronous SRAM require careful design. Let's see if we can safely write one byte per clock cycle. The key constraints are:

- address must be valid before asserting /WE
- data must be valid before deasserting /WE
- /WE must be deasserted briefly
- no adddress/data hold time after /WE

I required a fully synchronous design to be able to slow or stop the clock and was unwilling to employ any asynchronous delay tricks.

Accomplishing this requires one half clock to settle the write address, one half clock to assert /WE, and one half clock to deassert it. Therefore, byte writes take two full cycles, and word writes take three (e.g., a word write takes six half cycles W1–W6):

- W1: assert $XA_{14:1}$, data LSB, XA_0=1
- W2: assert /WE
- W3: deassert /WE, hold XA and data
- W4: assert data MSB, XA_1=0
- W5: assert /WE
- W6: deassert /WE, hold XA and data

MEMCTRL DESIGN

I've discussed the responsibilities of MEMCTRL design: address decoding, on-chip bus control, and external RAM control. Now, let's review its implementation (see Figure 6).

In address decoding, if the next access is a load/store to address FFxx, the access is to memory-mapped I/O, and SELIO is asserted. Otherwise, it's a RAM access.

Within each peripheral's DCTRL instance, its SEL_i (decoded from $AN_{7:5}$) and $CTRL_{SELIO}$ combine to develop that peripheral's output and clock enables.

For bus control, the current state of the memory transaction finite state machine determines which controls are asserted. The CPU asserts ACE (address clock enable) to request the next transaction and awaits RDY. MEMCTRL decodes the request, and the FSM enters the IO, RAMRD, or RAMWR state. The latter has three sub-states—W12, W34, and W56—corresponding to pairs of the W1–W6 half-states described previously.

In the IO state, RDY is asserted unless the selected peripheral deasserts $CTRL_0$, the I/O ready line, thereby inserting a wait state.

In the RAMRD state, RDY is as-

Transaction	Cycles	Enables
RAM read byte	1	LXDT
RAM read word	1	LXDT, UXDT
RAM write byte	2	LDT, XDOUTT
RAM write word	3	LDT or UDLDT, XDOUTT
I/O read byte	1+	*p*LDT
I/O read word	1+	*p*LDT, *p*UDT
I/O write byte	1+	LDT, *p*LCE
I/O write word	1+	LDT, UDT *p*LCE, *p*UCE

Table 3—*Depending on the memory transaction, different bus output enables and register clock enables are asserted.*

serted immediately because all RAM reads require only one clock cycle. In the RAMWR state, RDY is asserted on W34 for byte stores and on W56 for word stores.

The write controller uses flip-flops W23_45 and W45, which are clocked on CLK falling edges. So, W34 is true during W3 and W4, while W45 is true during W4 and W5. From the W* signals you derive glitch-free control signals XA_0, /WE, /OE, and so on.

The rest of MEMCTRL is straightforward. Note how E encodes (renames) the various peripheral control signals to $CTRL_{15:0}$.

I technology-mapped some logic using FMAPs. Timing analysis had revealed poor automatic mapping of this logic. This change shaved a few nanoseconds off the critical path.

Now that we've covered the implementation of MEMCTRL, let's turn our attention to peripherals.

PARALLEL PORT I/O

I provided parallel port I/O to communicate with the host. The XS40 board provides eight parallel port data inputs and five status outputs. Reserving a few for debug I/Os, I used six inputs and four outputs.

During `lb rd,FF41`, the PARIN input peripheral is selected, driving the inputs 00 || $PAR\_D_{5:0}$ onto $D_{7:0}$ (see Figure 5).

During `sb r1,FF21`, the PAROUT output peripheral is selected, capturing the store data $D_{3:0}$ in flip-flops, which drive the $PC\_S_{6:3}$ status outputs.

XOUT4 is as simple as XIN8. It has a DCTRL decoder, of course, and clocks $D_{3:0}$ on LCE (LSB clock enable). This parallel port requires only three CLBs, eight TBUFs, and 10 IOBs!

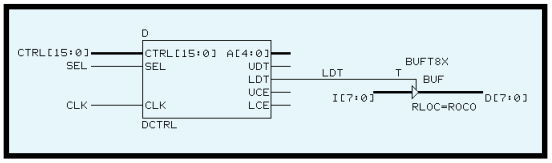

Figure 5—*The XIN8 (PARIN) implementation shows the CTRL decoder output LDT that enables the input byte to be driven onto the data bus.*

ON-CHIP RAM

XSOC also includes a 16 × 16-bit RAM peripheral. It uses all of the DCTRL outputs: $A_{4:1}$ to select the word to read or write, LCE and UCE as lower and upper byte write strobes, and LDT and UDT as lower and upper byte output enables.

VIDEO CONTROLLER

The bit-mapped video controller, based on ideas from [1], displays all 32 KB of external SRAM at 576 × 455 resolution, monochrome.

It runs autonomously from the CPU, and so is not a peripheral on the on-chip bus. It uses DMA to fetch video data, which consumes about 10% of memory bandwidth.

A video signal is a series of frames; each frame is a series of lines, and each line is a series of pixels. The video controller fetches 16-pixel words of video memory, shifts the pixels out serially, and uses horizontal and vertical sync pulses to format the pixels into frames and lines for the monitor.

Generating VGA-compatible horizontal and vertical sync timings, VGA shifts pixels out at 24 MHz, twice the system clock rate, shifting one out when CLK is high and a second when it is low. The horizontal and vertical sync pulses are advanced a few clocks (lines) to center the display in the frame (see Table 5).

The VGA ports are described in Table 6. The first five ports request new pixel data via the DMA controller. The rest are the VGA video outputs. The red, green, and blue intensities R1, R0, G1, G0, B1, and B0 drive resistor-based 2-bit D/A converters, providing up to 64 colors (4 × 4 × 4). However, at this resolution, with 32 KB of RAM, you can only support a monochrome (1-bit/pixel) display. So, each pixel bit drives all six outputs, drawing black or white pixels.

To generate horizontal and vertical syncs and a video blanking signal, you need a 9-bit horizontal cycle counter and a 10-bit vertical line counter.

After 288 clocks, it's time to blank the video. Assert horizontal sync after 308 clocks, deassert it after 353, and reset the counter and re-enable video after 381 clocks (one line).

In the vertical direction, the VGA controller must blank video after 455 lines, assert vertical sync after 486 lines, deassert it after 488 lines, and reset the counter, re-enable video, and reset the video DMA address counter after 528 lines.

The simplest way to build each counter is with a Xilinx library binary counter, such as a CC16RE. But because I had just about filled the FPGA, and because they're cool, I designed a more compact 10-bit linear feedback shift register (LFSR) counter. This uses a 10-bit serial shift register which has an input that is the XOR of certain shift register output taps.

An n-bit LFSR repeats every 2^n-1 cycles, but you can make an arbitrary m-cycle counter by complementing the LFSR input bit, thereby short-circuiting the full sequence when a particular bit pattern is recognized. My LFSR counter design program can be downloaded from the *Circuit Cellar* web site.

Referring to Figure 7, note the video controller contains two LFSR counters, H and V. Each has four comparators to compare the LFSR bit patterns to the count patterns output by my program.

Each of the J-K flip-flops HENN, NHSYNC, VEN, and NVSYNC are set on reaching one counter value and reset on reaching another.

Photo 1—*Here's the XS40 board, with the project design loaded into the FPGA and running a demo program that's drawing graphics on the monitor.*

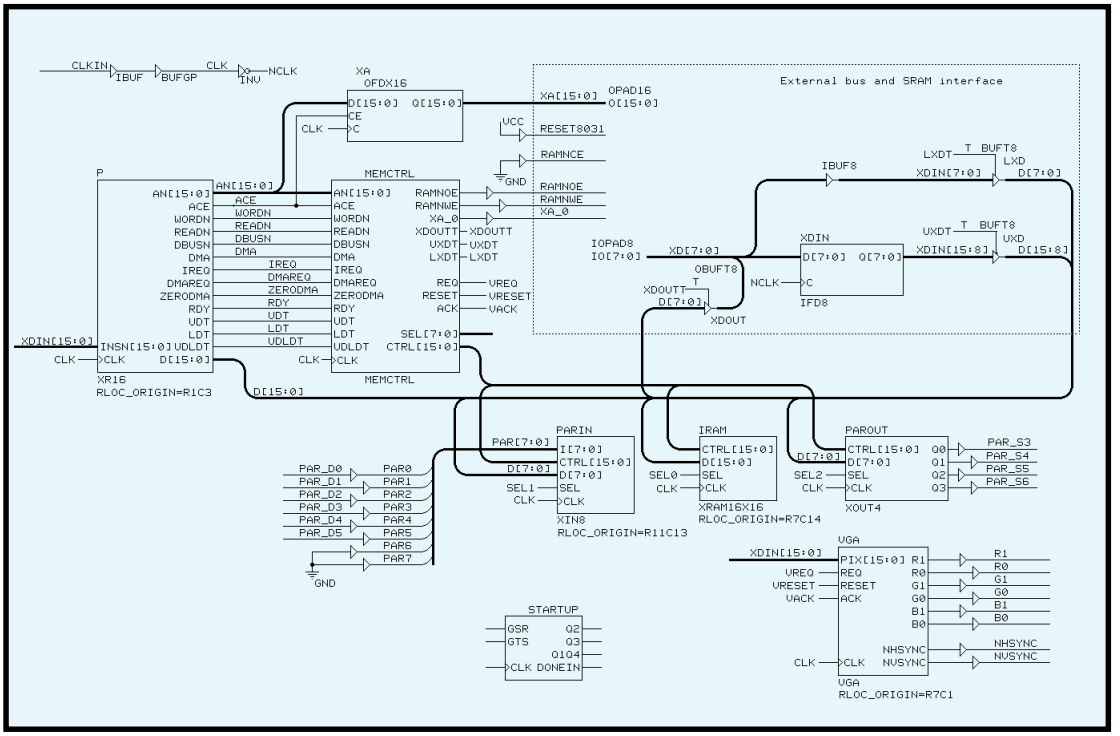

Figure 4—*The processor (P) issues requests to MEMCTRL, accessing instruction and data via the on-chip bus $D_{15:0}$ or external SRAM. Integrated peripherals provide parallel port I/O and on-chip RAM. The VGA controller fetches pixel data via DMA.*

NHSYNC is asserted low during clocks 308–353, and NVSYNC during lines 486–488. HEN is the pipelined horizontal video enable, and VEN is the vertical video enable. When both are true, you fetch and shift out video data.

In the video datapath, each clock shifts out two bits of video data. Every eight clocks, WORD goes true, and it requests a new 16-bit word of video data from memory. REQ is asserted, registering a pending DMA transfer with the CPU.

Five or fewer clocks later, the CPU performs the DMA load, asserting ACK. The video data word is latched in the PIXELS staging register. On the eighth clock, this word is loaded into the PMUX 8 × 2 parallel-load serial-out shift register.

Two bits shift out of PMUX during each clock, and feed a 2–1 mux that drives the 1-bit pixel each half clock.

SYSTEM BRING-UP

After designing the CPU, I designed a simple test-fixture using on-chip ROM and ran my test programs in the Foundation simulator.

After simulating test programs for hundreds of cycles, I compiled the design using the Xilinx tools and tested it on my XS40 board. Using a parallel port output for CLK, I wrote shell scripts to single-step the processor and observe $PC_{7:1}$ on the LEDs. Later, I ran the CPU at up to 20 MHz.

Starting from a core set of working instructions, it was easy to test the rest, one at a time. If something went awry, I could do a binary search for the problem, insert a `stop: goto stop;` breakpoint into my test, recompile, and download. A real remote debugger would be nice!

Armed with a working CPU, it is easy to add and test new features, one by one. I added double-cycled reads from external RAM, then MEMCTRL, then LED output registers. Writing text messages to the seven-segment LED was a big milestone. RAM writes were next. And, late in the project I added DMA, the video controller, and interrupts.

I want to emphasize the importance of thorough testing. You have your work cut out for you when properly testing a pipelined processor and an SoC.

This has been a proof-of-concept project, and I have focused on design issues. To ship something like this, you would need to budget as much or more time for validation as for the design and implementation.

The final system floorplan, as placed on our 14 × 14 CLB FPGA, is shown in Figure 3.

SERIES WRAP-UP

In this three-part series, I have presented the complete design and implementation of a real, full-featured, pipelined microprocessor and an integrated System-on-a-Chip. I designed a new instruction set, ported a C compiler, and discussed how to

Port	Description	Quantity	Value
$PIX_{15:0}$	next 16-bit pixel word	two-pixel clock	83.3 ns
REQ	request DMA of next word	one-pixel half-clock	41.7 ns
RESET	reset DMA address counter	visible pixels/line	576
ACK	DMA acknowledge input	visible clocks/line	288
CLK	system clock	horizontal sync "on" clock	308
R1,R0	2-bit red intensity	horizontal sync "off" clock	353
G1,G0	2-bit green intensity	line total clocks	381
B1,B0	2-bit blue intensity	line time	31.8 ms
NHSYNC	active-low horizontal sync	visible lines/frame	455
NVSYNC	active-low vertical sync	vertical sync "on" line	486
		vertical sync "off" line	488
		frame total lines	528
		frame time	16.8 ms

Tables 5 & 6—*The 12-MHz clock and 24-MHz pixel shift frequency determines the pixels per line and lines per frame, as well as the horizontal and vertical counter values for sync and blanking events.*

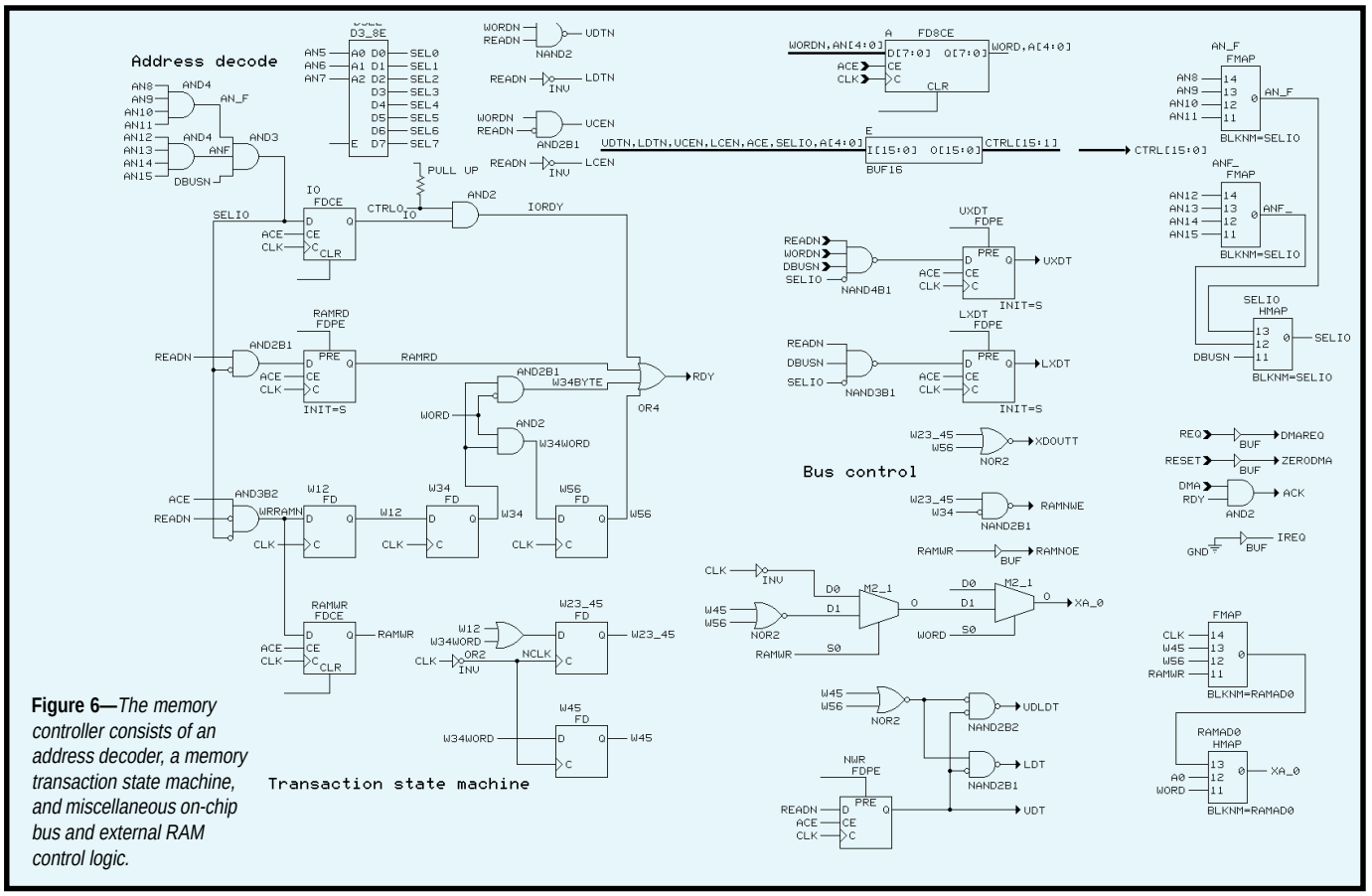

Figure 6—*The memory controller consists of an address decoder, a memory transaction state machine, and miscellaneous on-chip bus and external RAM control logic.*

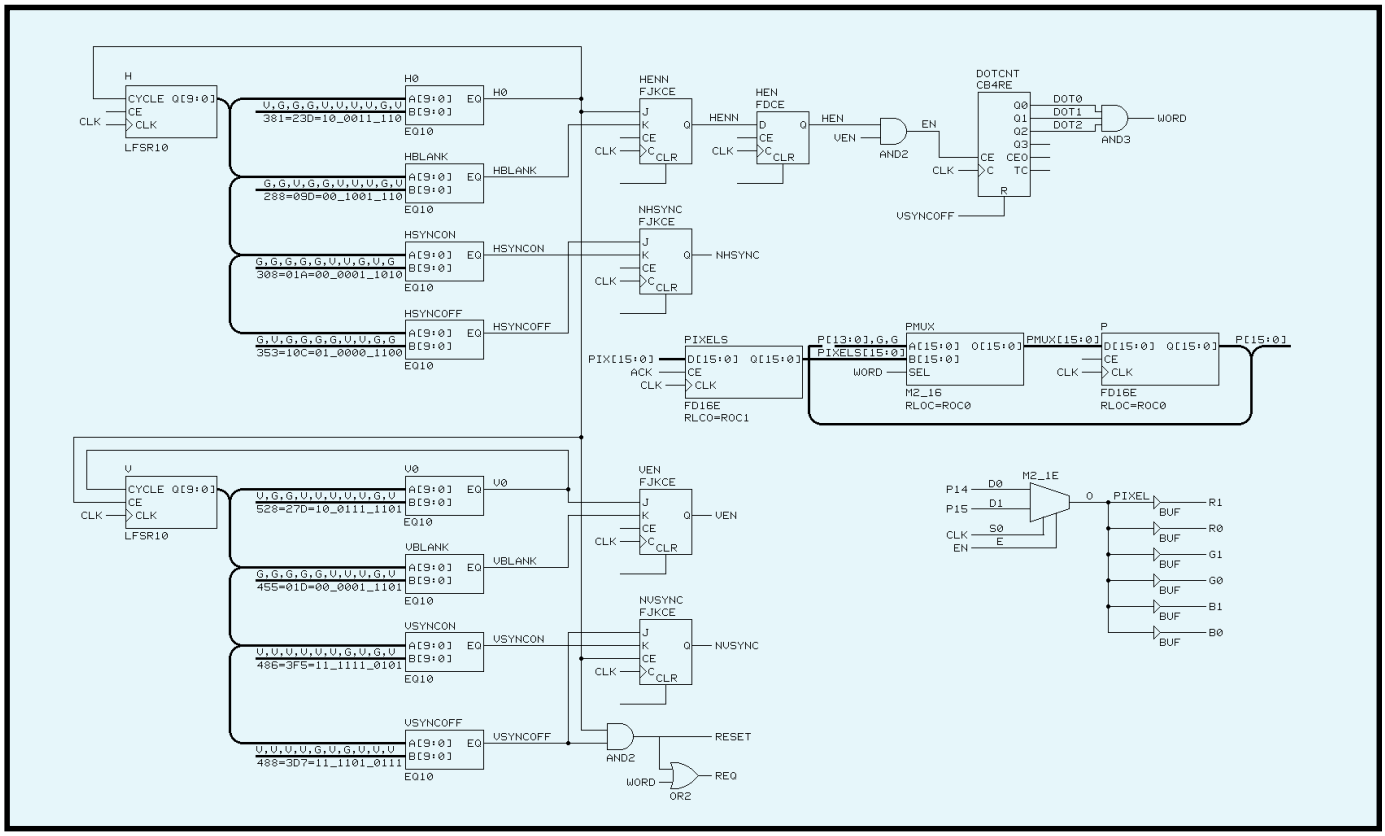

Figure 7—*As you can see, the video controller contains two LFSR counters that each have four comparators for comparing the LFSR bit patterns to the count patterns that are output by the program that I wrote.*

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

Please note that I do not warrant that you have the right to build something based upon the ideas discussed in this series of articles under the relevent intellectual property laws in your jurisdiction.

SOFTWARE

You may download more information, including specifications, source code, schematics, and links to related sites from the *Circuit Cellar* web site.

REFERENCE

[1] *VGA Signal Generation with the XS Board*, XESS App Note www.xess.com/fpga/vga.pdf

SOURCES

XESS XS40-005XL
www.xess.com/fpga

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