

CIRCUIT CELLAR®

THE MAGAZINE FOR COMPUTER APPLICATIONS

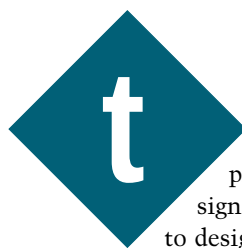
Building a RISC System in an FPGA

FEATURE ARTICLE

Part 3: System-on-a-Chip Design

Jan Gray

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 IIRAM
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) $\times$ (read or write) $\times$ (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 (xT) and control register clock enables (xCE). These enable signals are asserted according to the transaction type (see Table 3).

For example, during $sw\ r0$,

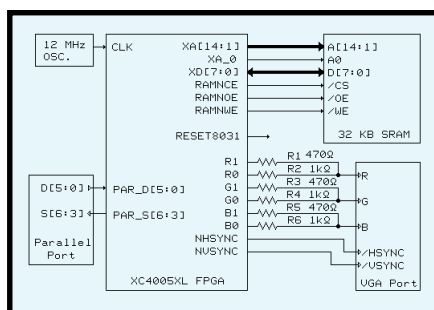


Figure 1—The system schematic depicts the subset of the XS40 needed for our project. The 8031 (not shown) is held in reset.

$0xFF00$, 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:

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

Later, it asserts UDLDLT and XDOUTT:

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

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	$D_{15:8} \leftarrow DOUT_{15:8}$
UDLDLT	$D_{7:0} \leftarrow DOUT_{15:8}$
XDOUTT	$XD_{7:0} \leftarrow D_{7:0}$
LXDT	$D_{7:0} \leftarrow XDIN_{7:0}$
UXDT	$D_{15:8} \leftarrow XDIN_{15:8}$
$pLDT$	$D_{7:0} \leftarrow pD_{7:0}$
$pUDT$	$D_{15:8} \leftarrow pD_{15:8}$
$pLCE$	$pD_{7:0} := D_{7:0}$
$pUCE$	$pD_{15:8} := D_{15:8}$

Table 2—There are a set of enables p_i^* 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 in-

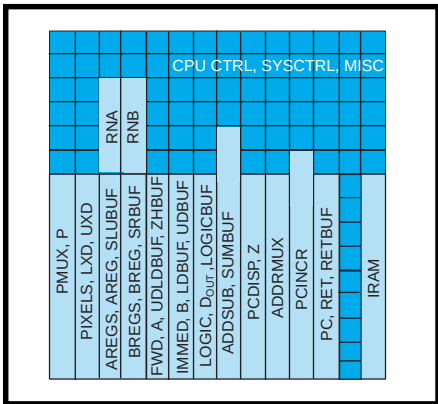


Figure 3—The rest of the device contains the automatically placed processor control unit and other logic.

clude A14:0 (address), D7:0 (data in/out), /CS (chip select), /WE (write enable), and /OE (output enable).

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. XA₀ (XA₀) 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

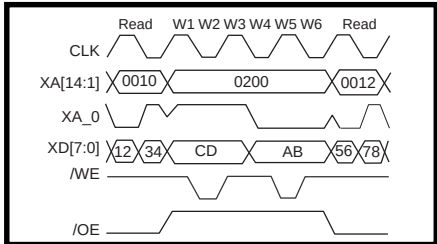


Figure 2—The RAM interface signals for three memory transactions are: read 1234 from address 0010, write ABCD to address 0200, and read 5678 from address 0012.

buffers), and IFDs (input flip-flops).

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 XA₀=0, reading the MSB, and latches it in XDIN. In the second half cycle, the system sets XA₀=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 address/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, XA₀=1
- W2: assert /WE
- W3: deassert /WE, hold XA and data
- W4: assert data MSB, XA₁=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 CTRL₀, 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.

sorted 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 1b 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

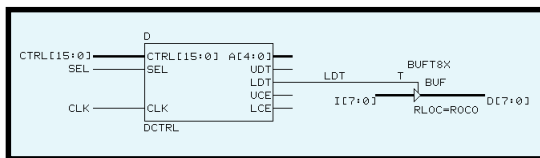


Figure 5—The XIN8 (PARIN) implementation shows the CTRL decoder output LDT that enables the input byte to be driven onto the data bus.

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!

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 2ⁿ-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.

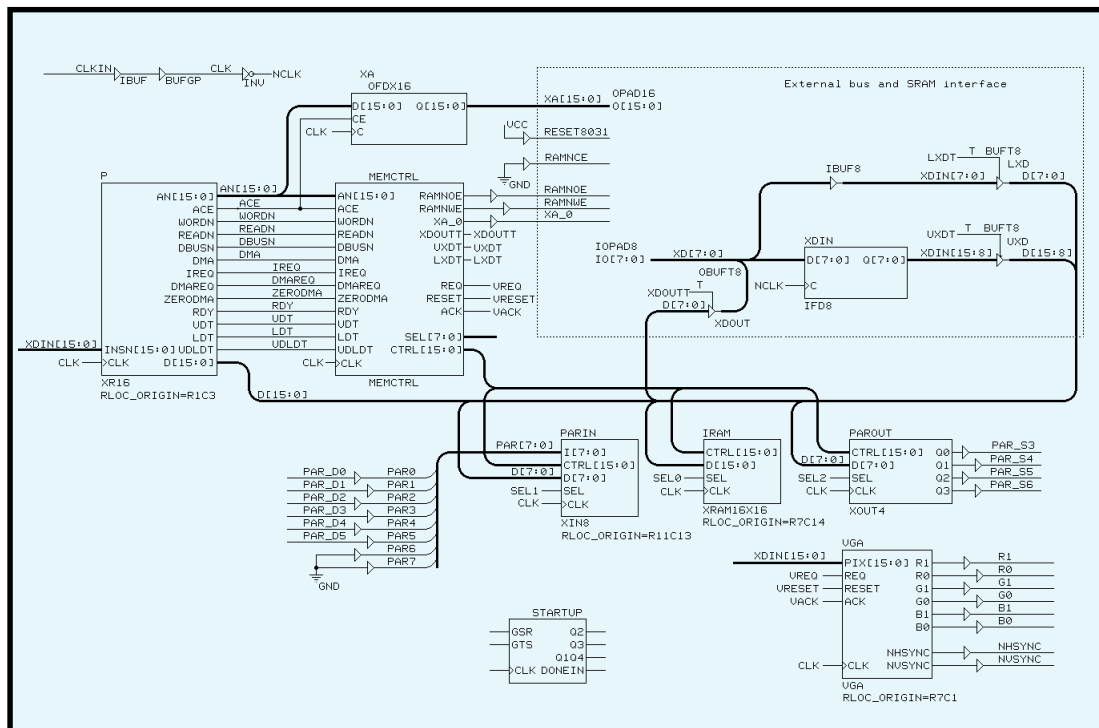


Figure 4—The processor (P) issues requests to MEMCTRL, accessing instruction and data via the on-chip bus $D_{15:0}$ or external SRAM. Integrated peripherals provide parallel port I/O and on-chip RAM. The VGA controller fetches pixel data via DMA.

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.

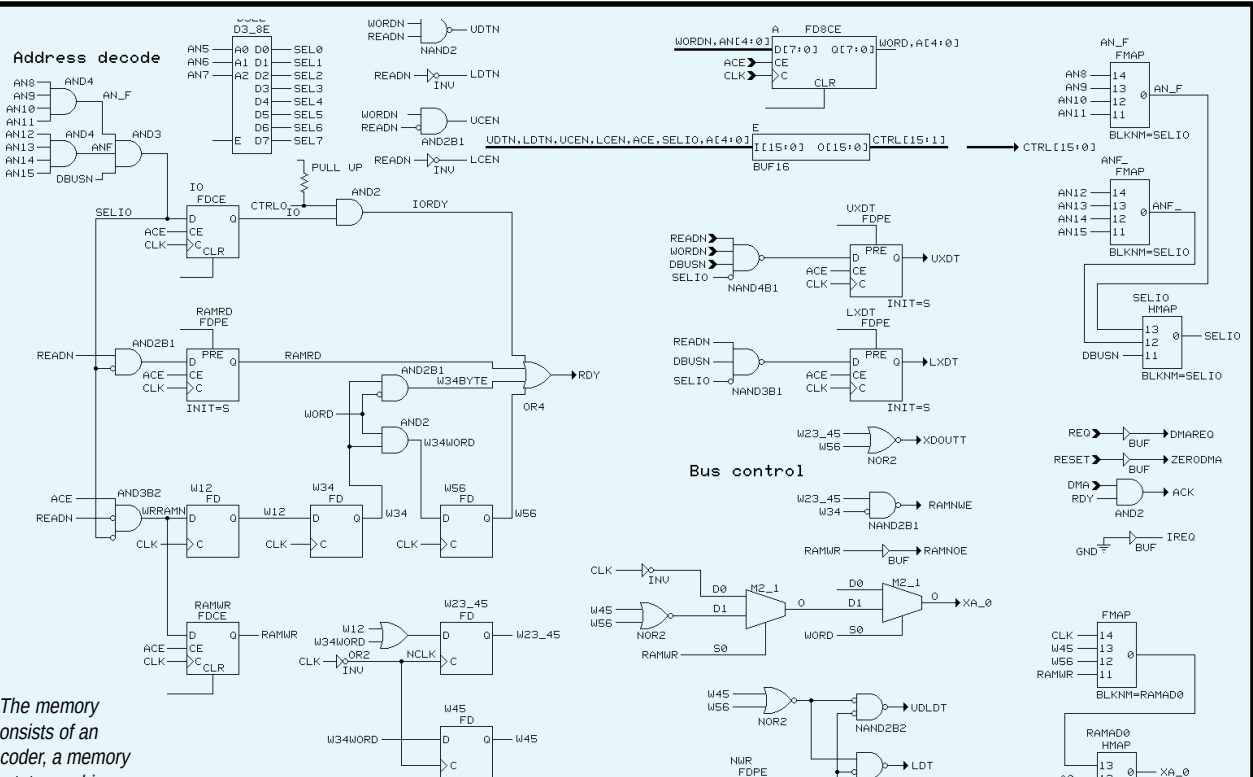


Figure 6—The memory controller consists of an address decoder, a memory transaction state machine, and miscellaneous on-chip bus and external RAM control logic.

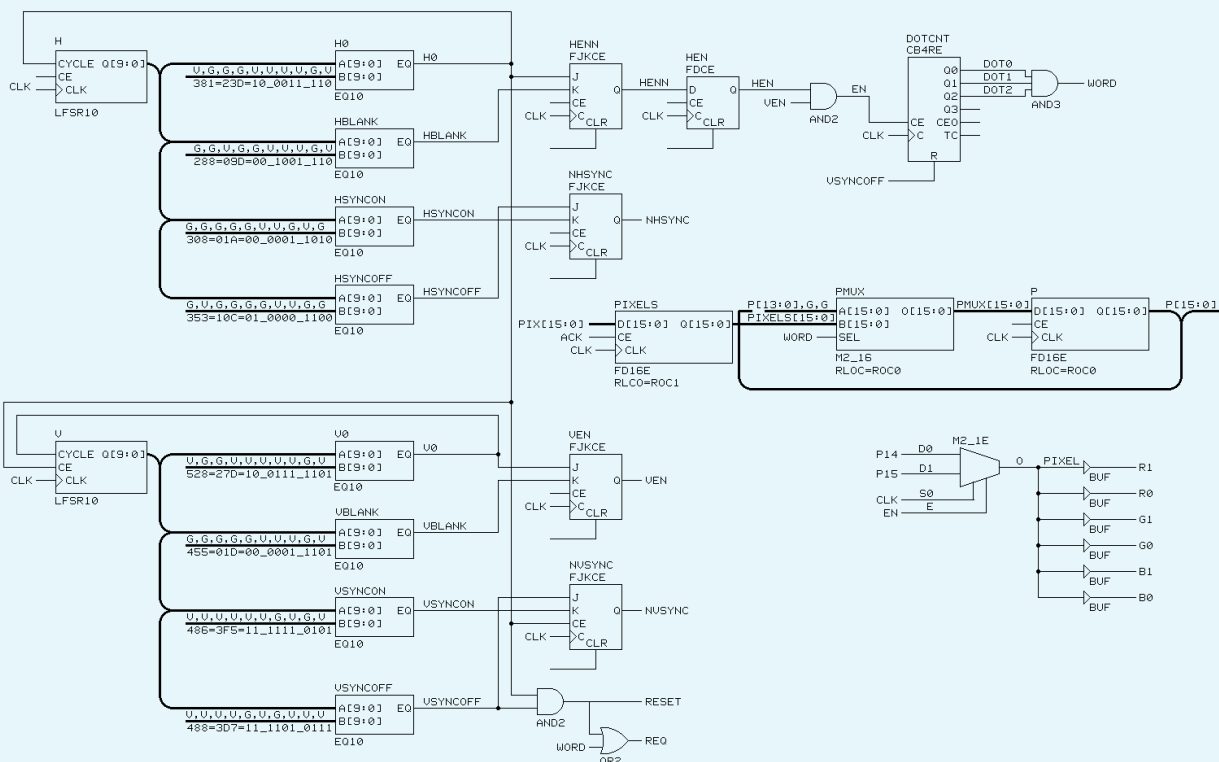


Figure 7—As you can see, the video controller contains two LFSR counters that each have four comparators for comparing the LFSR bit patterns to the count patterns that are output by the program that I wrote.

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 relevant 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