

Building a RISC CPU and System-on-a-Chip in an FPGA

Jan Gray,
Gray Research LLC

FINAL DRAFT: 10/15/1999

Tools, Instruction Set, and Datapath

Part 1 of 3. Jan builds a pipelined 16-bit RISC processor and system-on-a-chip in an FPGA. This month he ports a C compiler, designs an instruction set, writes an assembler and simulator, and designs the CPU datapath.

I USED TO envy CPU designers, those lucky engineers with access to expensive tools and fabs. Now field-programmable gate arrays make custom processor and integrated system design accessible to everyone. These days I design my own systems-on-a-chip, and it's great fun.

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 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

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

In contrast, this series demonstrates that 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.

We'll build a system-on-a-chip, including a 16-bit RISC CPU, memory controller, video display controller, and peripherals, in a small Xilinx 4005XL.

We'll apply free software tools including a C compiler and assembler, and we'll design our chip using Xilinx Student Edition, an inexpensive suite of FPGA design and verification tools.

Our goal is a simple, fast 16-bit processor that runs C code. Rather than implement a complex legacy instruction set, we'll design a new one streamlined for FPGA implementation: a classic pipelined RISC with 16-bit instructions and sixteen 16-bit registers.

First we'll need a C compiler.

C COMPILER

Fraser and Hanson [1] is the literate source code of their lcc retargetable C compiler. I downloaded the lcc 4.1 distribution and modified it to target our 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 (Table 1) and changed my md to target these registers.

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 (<i>sp</i>)

r14	interrupt return address
r15	return address

Table 1: XR16 register assignments

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`. That's easy: lcc obtains target operand sizes from md tables. I simply changed some entries from 4 to 2:

```
Interface xrl6IR = {
    1, 1, 0, /* char */
    2, 2, 0, /* short */
    2, 2, 0, /* int */
    2, 2, 0, /* T* */
    ...
}
```

Next, lcc needs operators which load a 2-byte int into a register, add 2-byte int registers, dereference a 2-byte pointer, etc. The lcc *ops* utility prints the required operator set. I modified my tables and instruction templates accordingly. Two examples:

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

⇒ “use `lb rd, addr` to load an unsigned char at `addr` and zero-extend it into this 16-bit int register”;

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

⇒ “use a `cmpi, beq` sequence to compare this register to this 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.

That's all there was to it – 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. Figure 1 is the source for a binary tree `search` routine, and Figure 2 is the assembly code lcc-xr16 emits (comments added).

```
typedef struct TN {
    int k;
    struct TN *left, *right;
} *T;

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

Figure 1: search.c

```
_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
```

Figure 2: search.asm

INSTRUCTION SET

Now we'll refine the instruction set and choose an instruction encoding. Our goals and constraints are:

- cover C (integer) operator set;
- fixed size, 16-bit instructions;
- easily decoded and pipelined ⇒ simple and regular;
- 3-operand instructions (dest = src₁ op src₂/imm) as encoding space allows;
- byte addressable – load and store bytes and words;
- big endian (arbitrary choice);
- one addressing mode, `disp(reg)`;
- long ints ⇒ add/subtract with carry, shift left/right extended.

Which instructions merit the most bits? Reviewing early compiler output from test applications shows 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 immediate instruction which 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.

There are six instruction formats:

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: XR16 instruction formats

and 43 distinct instructions:

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

Table 3: XR16 instruction set

Details:

- `adds`, `subs`, `shifts`, and `imm` are uninterruptible prefixes;
- `loads/stores` take 2+ cycles;

- jump and branch-taken take 3 cycles (no branch delay slots).
- 4-bit imm field encodes:
 - int (-8..7): add/sub, logic, shifts;
 - unsigned (0..15): lb, sb;
 - unsigned word displacement (0,2,...,30): lw, sw, jal, call;

Some assembly instructions are formed from other machine instructions:

Assembly	Maps to
nop	and r0,r0
mov rd,ra	add rd,ra,r0
cmp ra,rb	sub r0,ra,rb
subi rd,ra,im	addi rd,ra,-im
cmpi ra,im	addi r0,ra,-im
com rd	xori rd,-1
lea rd,imm(ra)	addi rd,ra,imm
lbs rd,imm(ra)	lb rd,imm(ra)
(load-byte, sign-extending)	xori rd,0x80 subi rd,0x80
j addr	jal r0,addr
ret	jal r0,0(r15)

Table 4: Some pseudo-instructions

Note only signed char data use lbs.

ASSEMBLER

I wrote a little multi-pass assembler to translate the lcc assembly output into an executable image.

The xr16 assembler reads one or more assembly files and emits an image file and a listing file. It has a simple structure. The *lexical analyzer* reads the source characters and recognizes tokens, such as the identifier `_main`. The *parser* scans the 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. Since 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.

SIMULATOR

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 3.3V 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 14x14 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 lookup tables (4-LUTs) and two flip-flops. Each 4-LUT can implement any logic function of 4 inputs, or a 16x1-bit synchronous static RAM, or ROM. Each CLB also has "carry logic" to help build fast, compact ripple-carry adders.

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

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

See [2] for more information.

The XC4000XL architecture seems to have been designed with CPUs in mind! Just 8 CLBs can build a single-port 16x16-bit register file (using LUTs as SRAM), a 16-bit adder/subtractor (using carry logic), or a 4 function 16-bit logic unit. Since each LUT has a flip-flop, the device is *register rich*, enabling a pipelined implementation style; and as each flip-flop has a dedicated clock enable input, it is easy to stall the pipeline 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.

PROCESSOR INTERFACE

Figure S1 is the XR16 processor macro symbol. The interface was designed to be easy to use with an on-chip bus. The key signals are:

- CLK: the system clock;

- AN_{15:0}: next memory address;
- READN: next access is a read;
- WORDN: next access is 16-bit data;
- ACE: address clock enable: above signals are valid, start next access;
- RDY: memory ready *input*: the current access completes this cycle.
- INSN_{15:0}: instruction word input;
- D_{15:0}: on-chip bidirectional data bus (load/store data).

The memory/bus controller (part three of this series) decodes the address and activates the selected memory or peripheral. Later it asserts RDY to signal that the memory access is done.

CPU ORGANIZATION

As Figure S2 shows, the CPU is simply a datapath that is steered by a control unit. Next month we examine the control unit in great detail. The rest of this article explores the design and implementation of the datapath.

DATAPATH RESOURCES

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

Here are the resources needed to execute one instruction per cycle:

- 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`);
 - an effective address adder to compute `reg+offset` (`lw r3, 2(r1)`);
 - a multiplexer 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 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`).
- Through careful design and reuse, we can minimize the datapath area:

- the adder, with the immediate mux, can do the effective address add;
- the PC incremter can also add branch displacements; and
- the memory address mux can help load the PC with the jump target.

DATAPATH SCHEMATIC

Figure S3, the datapath schematic, is the culmination of these ideas. There are three groups of resources:

Execution unit: the heart of the processor. 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.

Result multiplexer: selects one result from the various function units.

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.

REGISTER FILE

Each cycle we must read two register operands and write back one result.

We 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. Each cycle we must write the same result value into both copies.

So for each REGFILE, each clock cycle we must do one read access and one write access.

Each REGFILE is a 16×16 RAM. Recall each CLB has two 4-LUTs, each of which can be a 16×1-bit RAM. Thus a REGFILE is a column of 8 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, we 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, we now select the A and B operands, and latch them in the A and B registers. Some examples:

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 dout:=BREG
imm 0x123	ignored	imm12 0 _{3:0}
addi rd, ra, 4	AREG	B _{15:4} imm
add ₁ r3, r1, r2	AREG	BREG
add ₂ r5, r3, r4	RESULT	BREG

Table 5: Operand selection

(“||” means bit concatenation.)

The A operand is AREG unless (as with add₂) the instruction depends upon the result of the previous instruction. Next month we'll see why this pipeline *data hazard* is avoided by *forwarding* the add₁ result directly into the A register, just in time for add₂.

FWD, a 16-bit mux of AREG or RESULT, does this result forwarding. Its consists of 16 1-bit muxes, each a 3-input function implemented in a single 4-LUT, and arranged in a column of 8 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, we must 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 is an imm prefix, we load B_{15:4} with its 12-bit imm12 field, and *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}.

I used an unusual 2-1 mux with a fourth “force constant” input for this zero/sign extension function, 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 CLB's 16 flip-flops. It 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, we read the register to be stored, via BREG, into DOUT_{15:0}, another column of 8 CLB flip-flops.

ALU

The arithmetic/logic-unit consists of a 16-bit adder/subtractor and a 16-bit logic unit, which concurrently operate upon 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 4 inputs A_i, B_i, and LOGICOP_{1:0}, and fits in a single 4-LUT. Thus the 16-bit logic unit is a column of 8 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, 8 to add/sub 16 bits, and one to compute carry-out and overflow.

Z, the zero detector, is a 2½ CLB NOR-tree of the SUM_{15:0} output.

SHIFTER

The shifter produces either A>>1 or A<<1. This requires no logic; the result 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.

We build this 16-bit 7-1 mux from lots of 3-state buffers (TBUFs). 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 the series, we 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.

Each cycle we add PC += 2 to fetch the next instruction. For a taken branch, we add PC += 2×disp8. For lw, lb, sw,

sb, jal, and call, we load PC with the effective address SUM from ADDSUB.

Refer to the Figure S3 to see how this works. PCINCR adds PC and the PCDISP mux output (either +2 or the branch displacement) giving PCNEXT. ADDRMUX selects PCNEXT or SUM as the next memory address.

When 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.

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 a 16 entry register file. PC₀ is the CPU PC, and PC₁ is the DMA address. PC is a 16×16 RAM, 8 CLBs tall.

FLOORPLAN

We use RLOC attributes to place the datapath elements. Here is the resulting floorplan on our 14×14 CLB FPGA.

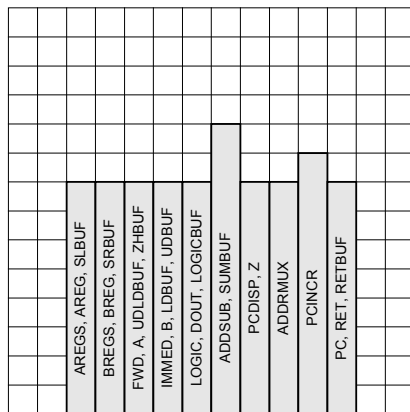


Figure 3: Datapath floorplan

RESULT_{15:0} runs horizontally across the bottom 8 rows of CLBs.

THE DATAPATH IN ACTION

Let's see what happens when we run 0008: addi r3, r1, 2

Assume PC=6 and r1=10. PCINCR adds PCDISP=2 to PC=6, giving PCNEXT=8. Since SELPC is true, ADDR ← PCNEXT=8, and the next memory cycle reads the word at 0008. Since 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 our schematics into a configuration bitstream, Foundation runs these tools:

- map: *technology mapping*: map schematics' 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 FPGA DESIGN

The datapath implementation showcases some good practices:

- exploit FPGA features: use embedded SRAM, four input logic structures, TBUFs, and flip-flop clock enables;
- floorplan: place functions in columns; order columns to reduce interconnect requirements; run the 3-state bus horizontally over the columns;
- iterative design: measure the area and delay effects of each potential feature;
- use timing-driven place-and-route;

- use iterative timing improvement.

I apply timing constraints, such as net CLK period=28;, which cause 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.

WRAP UP

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

Jan Gray is a software developer and amateur computer designer. His products include a leading C++ compiler and an application server. Jan built his first FPGA RISC system in 1995. Write him at jan@fpgacpu.org.

GETTING STARTED

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

(The Xilinx University Program folks confirm that Student Edition is not just for students, but also for professionals' continuing education. Since 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 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.)

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 Ink*, 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.)

SOURCES

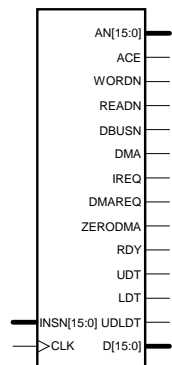
Xilinx Student Edition 1.5

www.amazon.com/exec/obidos/ASIN/0130205869

Xilinx FPGAs

www.xilinx.com/products/sp_xl.htm,
[.../xc4000XLA.html](http://www.xilinx.com/products/sp_xl.htm#xc4000XLA)

Copyright © 1998-2000, Gray Research LLC. All Rights Reserved.



XR16

Figure S1: CPU Macro Symbol

Copyright (C) 1999, Jan Gray	Project: [None]
All rights reserved	Sheet: XSOC1
	Date: 08/08/99

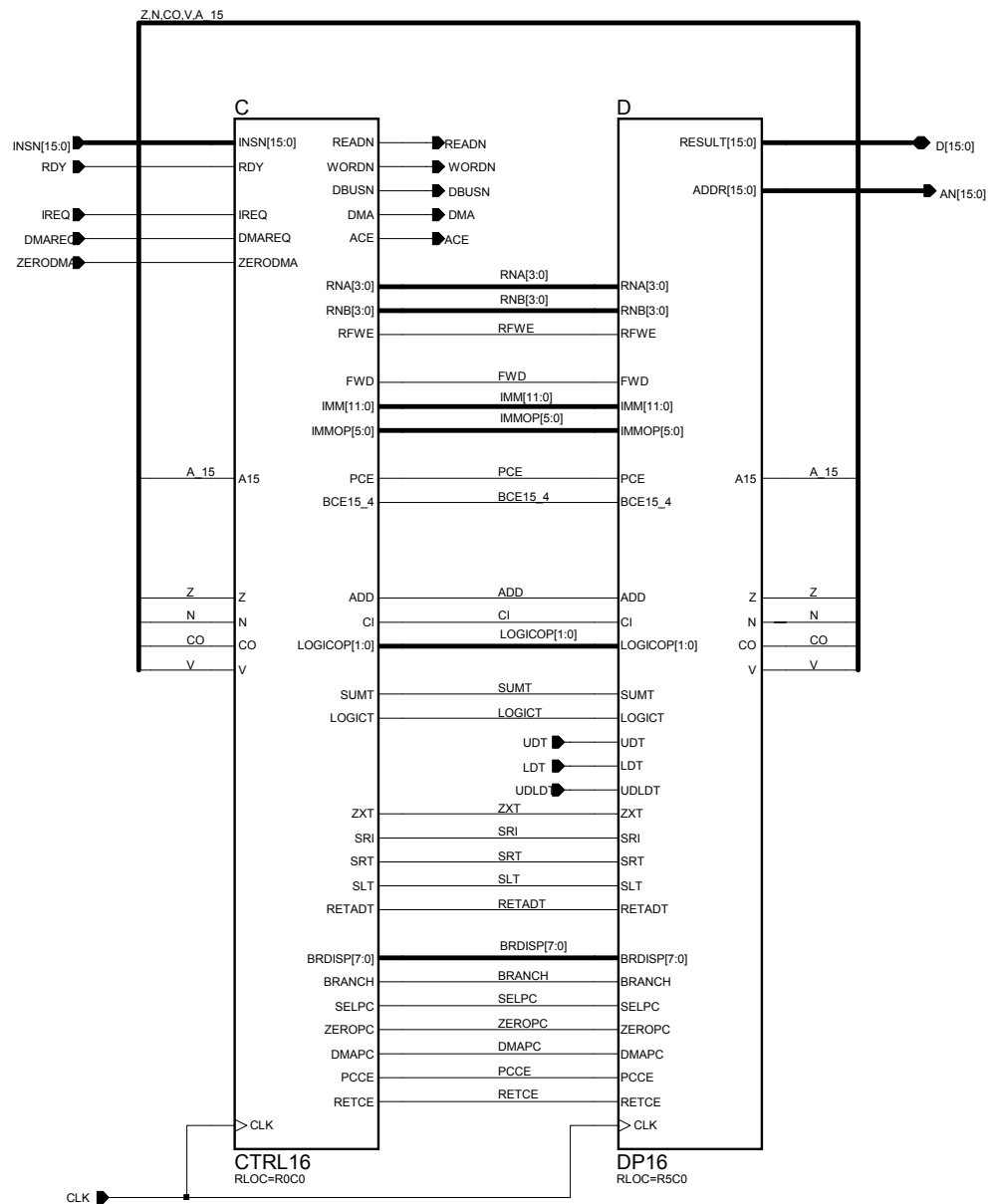


Figure S2: XR16 CPU Schematic

Copyright (C) 2000, Gray Research LLC.	Project: [None]
This work and its use subject to XSOC License Agreement. See LICENSE file.	Macro: XR16
	Date: 02/23/100

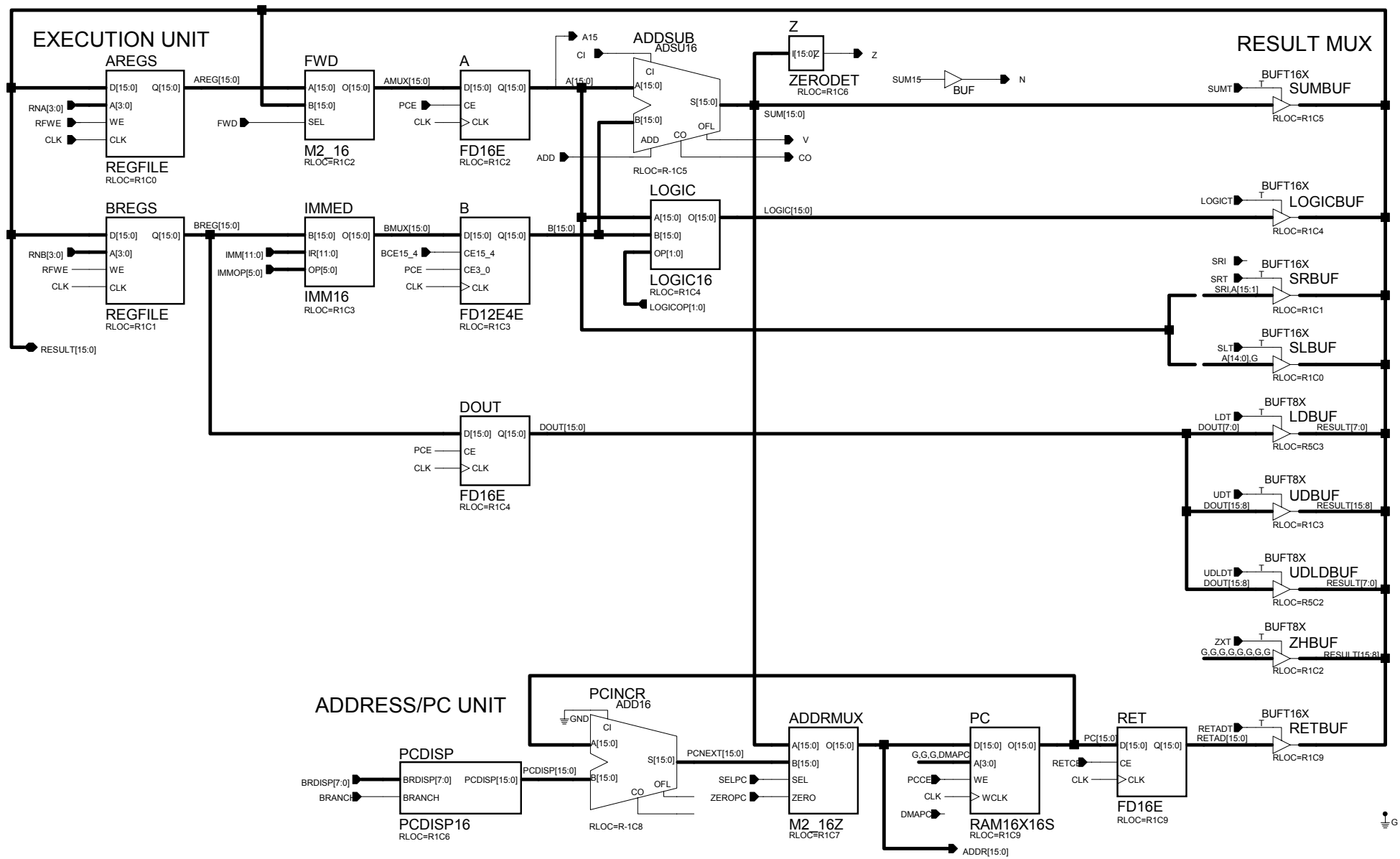


Figure S3: XR16 CPU Datapath Schematic

Copyright (C) 2000, Gray Research	Project: [None]
This work and its use subject to XSD	Macro: DP16
License Agreement. See LICENSE	Date: 02/23/100

Building a RISC CPU and System-on-a-Chip in an FPGA

Jan Gray,
Gray Research LLC

FINAL DRAFT: 10/15/1999

Pipeline and Control Unit Design

Part 2 of 3. Jan builds 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.

LAST MONTH we discussed the instruction set and the datapath of our XR16 16-bit RISC processor. Now we'll design the control unit that pushes the datapath's buttons.

Figure S2 (last month) shows 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 include pipeline control clock enables, register and operand selectors, ALU controls, and result multiplexer output enables.

Before we design the control circuitry, we first consider how the pipeline behaves in good times and bad.

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, we read memory at the current PC address, capture the resulting instruction word in the instruction register IR, and increment 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 3-state buffers onto the result bus, and is written back into the register file as the cycle ends.

Consider executing a series of instructions and assume no memory wait states. Every pipeline cycle we

fetch a new instruction and write back its result two cycles later. In each cycle we simultaneously prepare next instruction address PC+2, fetch instruction I_{PC} , decode instruction I_{PC-2} , and execute instruction I_{PC-4} .

t_1	t_2	t_3	t_4	t_5
IF₁	DC ₁	EX ₁		
	IF₂	DC ₂	EX ₂	
		IF₃	DC ₃	EX ₃
			IF₄	DC ₄

Table 1: Pipelined execution

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**.

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 DMA.

DATA HAZARDS AND RESULT FORWARDING

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

I_1 :	and _i r1, 7
I_2 :	add r2, r1, 1

Figure 1: I_1 result $r1$ used in I_2

Referring back 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 adds 1 to this old $r1$. Oops!

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. But these techniques hurt performance.

Instead, we 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, inserting nops if not.

MEMORY INTERFACE

The processor has a single memory port to read instructions and to load and store data. Most memory accesses are to fetch instructions. The processor is also the DMA engine, and a video refresh DMA cycle occurs once every eight clocks or so.

So 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. Each memory cycle, the processor drives the *next* memory cycle's address and control signals and awaits RDY, indicating the access has completed.

WAIT!

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, etc. The pipeline registers do not clock, and this extends all pipeline stages by one cycle.

In the following example, 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*.

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: IF_3 memory not ready at t_3

LOADS AND STORES

Loads and stores need a second memory access – pipeline havoc! Scenario:

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

```
 $I_L$ :    lw r6,2(r10)
 $I_2$ :    andi r6,7
 $I_3$ :    addi r0,r6,-1
 $I_B$ :    bne T
 $I_5$ :    lw r6,6(r10)
... 
```

Figure 2: Load instruction in situ

Its pipelined execution:

t_1	t_2	t_3	t_4	t_5
IF_L	DC_L	EX_L	EX_L	
	IF_2	DC_2	DC_2	EX_2
		IF_3	IF_3	DC_3
				IF_B

Table 3: Pipelined load execution

There are several gotchas here.

1. In t_4 we run a load/store memory cycle instead of an instruction fetch. We must stall the pipeline to squeeze in this access.

2. Although we fetched I_3 in t_3 , we must not latch it into the instruction register IR as t_3 ends, because neither EX_L nor DC_2 are finished yet. 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 !

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

To fix this problem, we provide a 16-bit NEXTIR register and an IR source multiplexer IRMUX. In t_3 we capture 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). For this 8 CLB overhead, we ensure 2-cycle loads and stores.

WAIT! WAIT!

Let's combine the last two topics, and consider how the pipeline for the code in Fig. 2 might proceed if the memory cycles for the instruction fetch IF_3 and the load EX_L each require two clocks, e.g. one wait state RAM. We'll start at clock t_3 .

t_3	t_4	t_5	t_6	t_7
EX_L	EX_L	EX_L	EX_L	
DC_2	DC_2	DC_2	DC_2	EX_2
IF_3	IF_3	IF_3	IF_3	DC_3
				IF_B

Table 4: Fetch, load with wait states

This is fairly straightforward: RDY is not asserted during t_3 and t_5 so we repeat these cycles in t_4 and t_6 .

JUMPS AND BRANCHES

Next consider the effect of jumps (call and jal) and taken branches. By the time we execute the jump or taken branch I_j during EX_j , updating PC, we will 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 , we must annul the EX_5 stage of I_5 and the DC_6 and EX_6 stages of I_6 . (Annulled stages are ~~struck through~~.) Execution continues at instruction I_T :

t_5	t_6	t_7	t_8	t_9
EX_2				
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 5: Branch shadow annulled

T_9 is not an EX_5 load cycle because the I_5 load is annulled.

Since we always annul the two branch shadow instructions, jumps and taken branches take 3 cycles.

Jumps also save the return address in the destination register. This return address is obtained from the RET register, which holds the address of the instruction in the DC pipeline stage.

INTERRUPTS

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

We do this in a very simple way. (Thank goodness we have no branch delay slots or condition codes!) When INTREQ is asserted, we override the fetched instruction with *int*, e.g. `jal r14,10(r0)`, via the IRMUX.

This jumps to the interrupt handler at 0x0010 and leaves the return address in r14, reserved for this purpose.

When the handler completes, it executes `iret`, e.g. `jal r0, 0(r14)`, and execution resumes with the interrupted instruction.

There are two pipeline issues.

1. We must not interrupt an interlocked instruction sequence, e.g. 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.

2. 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 pays off. The time to take an interrupt and then return from a null interrupt handler is only 6 cycles.

You might be wondering, “what about interrupt priorities, nested interrupts, 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.

DMA

The last pipeline issue is peripheral direct memory access (DMA). The PC/address unit doubles as a DMA engine. Using a 16x16 RAM as a PC *register file*, we can fetch either an instruction ($AN \leftarrow PC_0 += 2$) or a DMA word ($AN \leftarrow PC_1 += 2$) per memory cycle.

After an instruction is fetched, if `DMAREQ` has been asserted, we insert one DMA memory cycle. (One suffices for this project.)

This PC register file costs 8 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.

CYCLES OF CYCLES

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, optional EX stage load or store).

Each memory access cycle requires one or more clock cycles.

ON TO THE DESIGN

Now that we understand the pipeline, we are ready to design the control unit. (Lost? Learn all about pipelines in [1].)

First, here are some important naming conventions. Some control unit signal names have prefixes and suffixes to recognize their function or context.

- *Nsig*: not signal – signal inverted;
- *DCsig*: a DC stage signal;
- *EXsig*: an EX stage signal;
- *sigN*: signal in “next cycle” – input to a flip-flop whose output is *sig*;
- *sigCE*: flip-flop clock enable;
- *sigT*: active low 3-state buffer output enable.

Most signal names sans prefix are DC stage signals.

THE CONTROL UNIT

Figure S4 is the control unit schematic, the implementation of CTRL16 of Figure S2. The control unit consists of:

- the control finite state machine;
- the instruction registers (next, decode stage, execute stage);
- the instruction decoder;
- decode stage operand selection logic (register file, forwarding, and immediate operand control);
- execute stage logic (ALU and result mux control); and
- conditional branch control.

Here’s how an instruction flows through the control unit pipeline:

IF stage: When the instruction fetch read completes, the new instruction at $INSN_{15:0}$ is latched into IR.

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.

EX stage: Now CTRL drives ALU and result mux controls. If the

instruction 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.

Now we’ll consider each part of the control unit in more detail.

FINITE STATE MACHINE

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.

Here’s the state transition diagram:

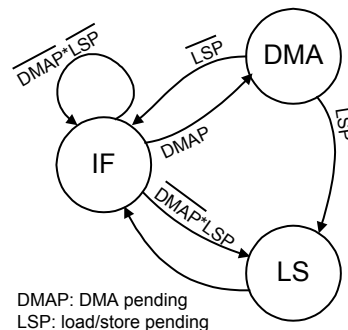


Figure 3: Control unit FSM

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).

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.

Figure S5 is the control FSM schematic. The LS state has been encoded as “~IF & ~DMA”.

Because the memory interface is pipelined, many control outputs depend not on the current state, but on the *next* state signals (IFN, DMAN, and LSN).

PIPELINE CONTROL

The highest fanout 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. Conversely, if memory is not ready, PCE is not asserted and the pipeline stalls for one cycle.

MEMORY INTERFACE

The control FSM also manages the memory interface via these signals:

- 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.

IF: THE INSTRUCTION FETCH STAGE

There are no IF stage control outputs. But internal to the control unit, three signals control IF stage resources (see the Instruction Registers block of S4):

- 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 fetched instruction. (See time t_3 in Table 3.) Otherwise, the instruction fetch is the only memory access in pipe stage, so IF is asserted *with PCE*, and IRMUX selects the INSN_{15:0} input as the next instruction.

DC: THE DECODE STAGE

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

As the DC stage begins, we have just latched a new instruction word in the instruction register IR. We use the buffers IRB and IMMB to break out the instruction fields OP, RD, etc., so IR_{15:12} is renamed OP_{3:0} and so forth. (The tools optimize away these buffers.)

INSTRUCTION DECODER

DECODE (schematic not shown) is simple. It is a set of 30 ROM16X1s and 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 we chose a simple instruction set, and because the 4-bit opcodes are a good match for 4-LUTs.

REGISTER FILE

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
- RFEW: register file write enable.

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.

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

The muxes RNA and RNB produce RNA_{3:0} and RNB_{3:0} as shown below, as selected by decode outputs RRRI, CALL, ST, EXCALL, and CLK.

Field	When
RA	DC: <i>add sub addi lw lb sw sb jal</i>
RD	DC: <i>all rr, ri format</i>
0	DC: <i>call</i>
EXRD	EX: <i>all but call</i>
15	EX: <i>call</i>

Table 6: RNA sources

Field	When
RB	DC: <i>add sub, all rr fmt</i>
RD	DC: <i>sw sb</i>
EXRD	EX: <i>all but call</i>
15	EX: <i>call</i>

Table 7: RNB sources

Call is irregular. It computes

$r15 = pc, pc = r0 + imm12 \ll 4;$
and registers r15 and r0 are implicit.

A OPERAND

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.

B OPERAND

Last month we 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 instead. 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.

CONDITIONAL BRANCHES

If the DC stage instruction is a conditional branch, then the EX stage instruction must be some *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 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. Then 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 $2 \times disp8$ to the PC.

EX: THE EXECUTE STAGE

It remains to discuss the EX stage ALU, result mux, and address unit controls.

ALU

The ALU and shift control outputs are:

- ADD: set unless the instruction is `sub` or `sbc`.
- CI: carry-in. 0 for `adds` and 1 for `subs`, 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} (e.g. EX stage copy of FN_{1:0}).
- SRI: shift right input: 0 for `srli`, A₁₅ for `srai` (shift right *arithmetic*). S_{lxi} and S_{rxi} (shift extended left/right, for multi-word shift support) are not yet implemented. Be my guest!

RESULT MUX

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:

Enable	Instruction	Source
SUMT	<code>add sub addi adc sbc adci sbci</code>	SUM _{15:0}
LOGICT	<code>and or xor andn andi ori xori andni</code>	LOGIC _{15:0}
SLT	<code>slli</code>	A _{14:0} 0
SRT	<code>srli srai</code>	SRI A _{15:1}
ZXT	<code>lb</code>	0 _{15:8}
RETADT	<code>jal call</code>	RETAD _{15:0}
<i>none</i> (no result)	<code>sw sb br* imm</code>	-

Table 8: Result output enables

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

PC/ADDRESS UNIT

These 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;
- DMAPC: if set, fetch and update PC₁ (DMA address), otherwise PC₀ (PC).

Depending upon the next memory cycle and the current EX stage instruction, the control unit selects the next address by asserting certain combinations of control outputs:

Next Cycle	Next address	Outputs
IF	AN ← PC ₀ += 2	SELPC PCCE
IF <i>branch</i>	AN ← PC ₀ += 2×disp8	BRANCH SELPC PCCE
IF <i>jal call</i>	AN ← PC ₀ = SUM	PCCE
IF <i>reset</i>	AN ← PC ₀ = 0	SELPC ZEROPC PCCE
LS <i>load/store</i>	AN ← SUM	-
DMA	AN ← PC ₁ += 2	SELPC DMAPC PCCE
DMA <i>reset</i>	AN ← PC ₁ = 0	SELPC ZEROPC DMAPC PCCE

Table 9: Address unit control outputs

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. Next month, the final article in this series tackles the design of our system-on-a-chip.

Jan Gray is a software developer and amateur computer designer. His products include a leading C++ compiler and an application server. Jan built his first FPGA RISC system in 1995. Write 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] Patterson, D. and J. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufmann, San Mateo, CA, 1994.

Copyright © 1998-2000, Gray Research LLC. All Rights Reserved.

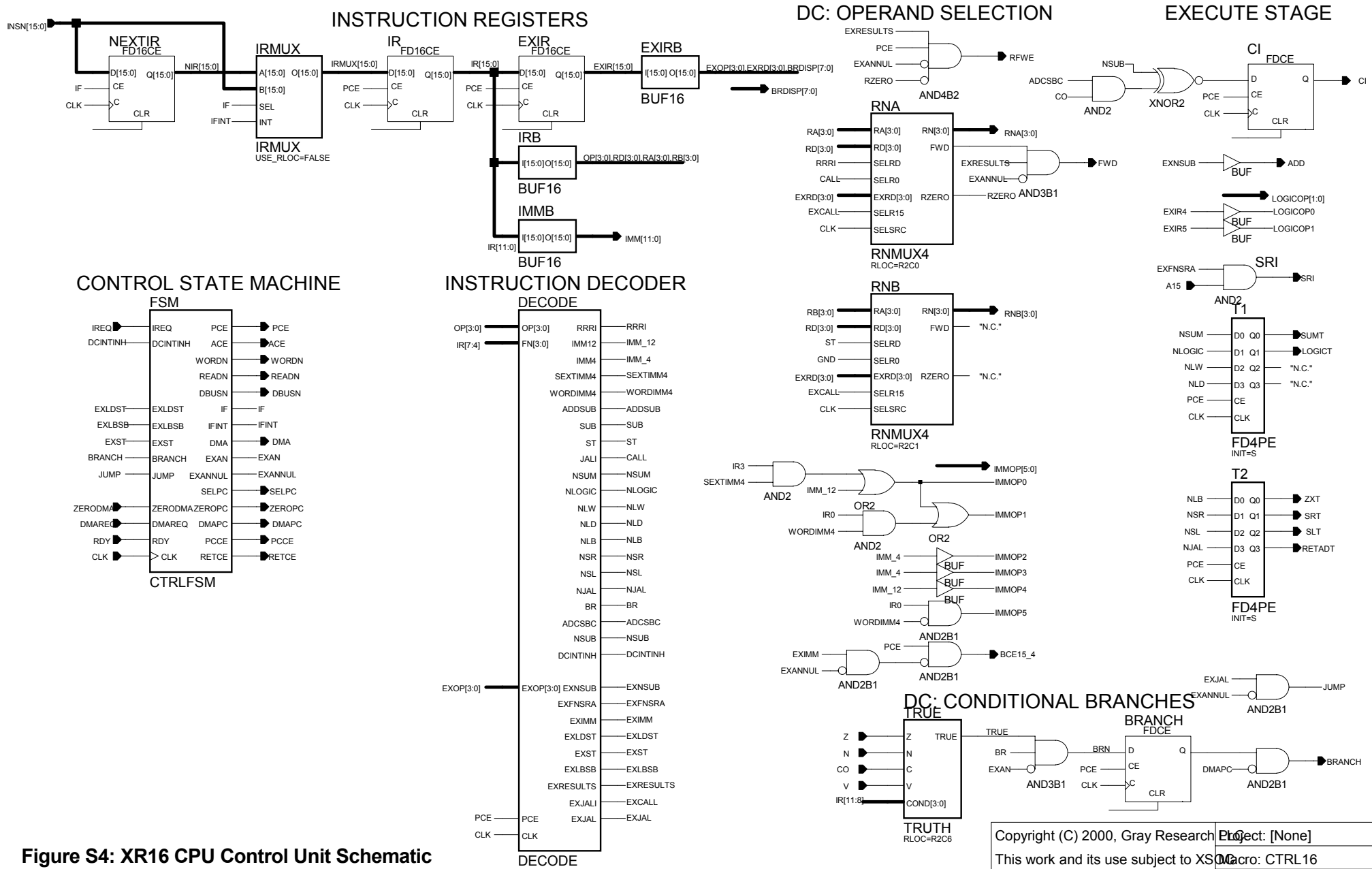
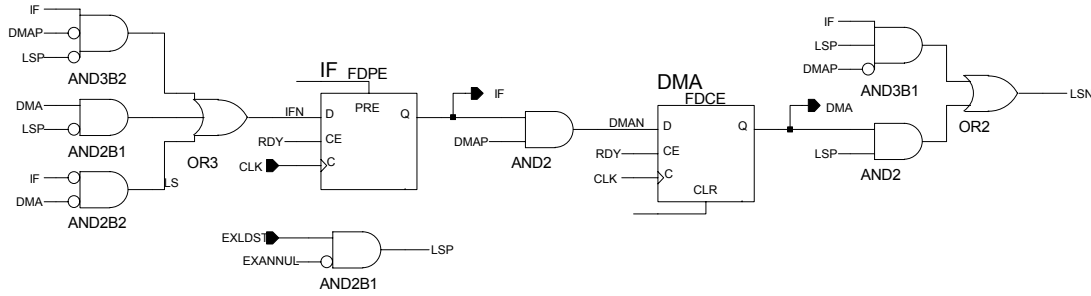


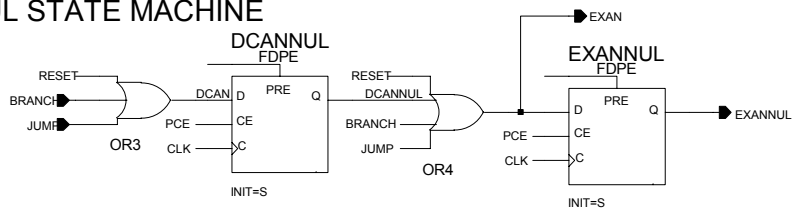
Figure S4: XR16 CPU Control Unit Schematic

Copyright (C) 2000, Gray Research, Inc. Project: [None]
 This work and its use subject to Xilinx Macro: CTRL16
 License Agreement. See LICENSE File: 02/23/100

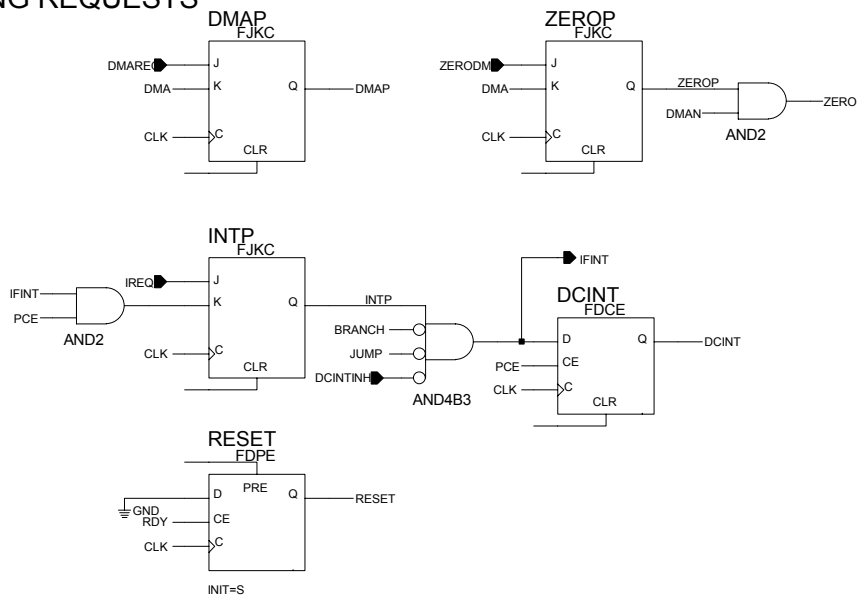
MEM CYCLE STATE MACHINE



ANNUL STATE MACHINE



PENDING REQUESTS



FSM OUTPUTS

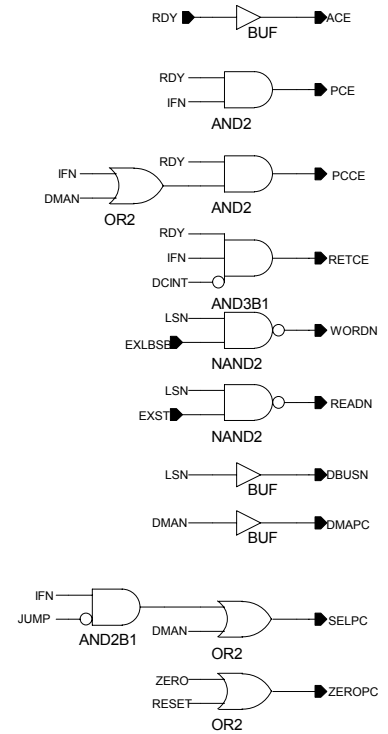


Figure S5: XR16 CPU Control Unit State Machines

Building a RISC CPU and System-on-a-Chip in an FPGA

System-on-a-Chip Design

Part 3 of 3. Jan builds a pipelined 16-bit RISC processor and system-on-a-chip in an FPGA. This month he designs a demonstration system, including an on-chip bus, memory controller, video controller, and peripherals.

LAST MONTH we completed our XR16 16-bit RISC processor. This month we design the rest of our system-on-a-chip (SoC). Besides the CPU, our FPGA hosts an on-chip bus, bus controller, parallel port, RAM, video controller, and an external SRAM controller.

In this article we'll see how some simple interfaces can make SoC design as straightforward as classic "CPU + glue logic + memory + peripherals" PCB design used to be.

XS40 BOARD

Our project targets the XESS XS40-005XL v1.2 FPGA board (Figure 1), which includes a Xilinx XC4005XL, 12 MHz oscillator, 32 KB SRAM, 8031 MCU, 7-segment LED, voltage regulators, and parallel port and VGA port connectors. It is simple and inexpensive, and is featured in "The Practical Xilinx Designer Lab Book" included with Xilinx Student Edition.

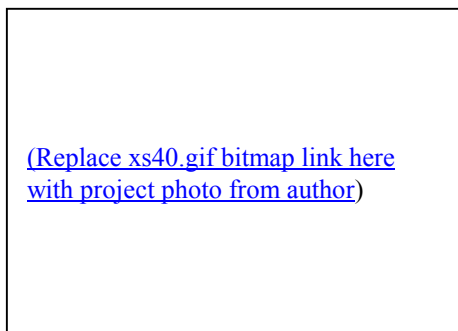


Figure 1: XS40 FPGA board

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

Jan Gray,
Gray Research LLC

FINAL DRAFT: 10/15/1999

A SYSTEM-ON-A-CHIP

We'll build an integrated system from the resources at hand – the FPGA, RAM, the video and parallel ports, and the (leisurely) 12 MHz oscillator.

We use the RAM for program, data, and video memory. The *byte-wide, asynchronous* SRAM isn't ideal, but it is fast enough that we can read and latch a byte on each clock edge, to fetch a 16-bit instruction each cycle.

By displaying all 32 KB of RAM, we can fashion a bitmapped 576x455 monochrome video display at VGA compatible sync frequencies. How quaint, to watch every bit on screen!

Figure 2 depicts the subset of the XS40 that our project uses. The 8031 (not shown) is held in perpetual reset.

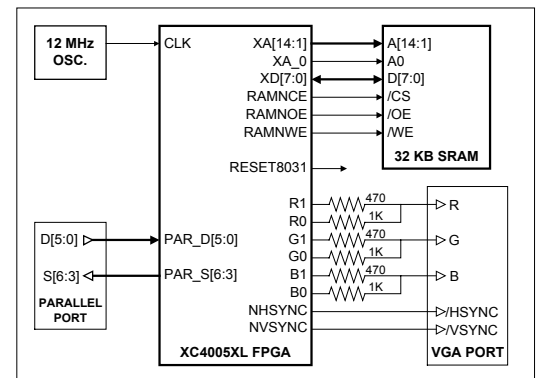


Figure 2: System schematic

See also Figure S6, the FPGA top-level schematic. It includes the 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.

ON-CHIP BUS VARIATIONS

Before we study the design, we explore the on-chip bus design space – what are our options?

(This is not the sort of thing you worry about when designing to someone else’s microprocessor. But in an FPGA SoC, you have more freedom, and more angst.)

Here are some of the issues:

- How many CPUs or masters? *One; the CPU does DMA arbitration.*
- Are bus clocks CPU clocks? *Yes.*
- One memory port or two? *One.*
- How wide are the various buses? *Internal: 16-bits; external: 8-bits.*
- Byte addressable? *Yes, but no byte access to word-wide peripherals.*
- Are transactions pipelined? *Yes.*
- Is the CPU internal result bus isolated from the on-chip data bus? *No.* For this project, I chose to combine, not isolate, the CPU bus RESULT_{15:0} (see first article) and the on-chip data bus D_{15:0}. This simplifies some things.

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, 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.

Here’s our memory map:

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
FF60-FF7F	3: <i>unused</i>
...	...
FFE0-FFFF	7: <i>unused</i>

Table 1: Memory map

There are eight different 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 (xT) and control register clock enables (xCE):

Enable	Effect
LDT	D _{7:0} ← DOUT _{7:0}
UDT	D _{15:8} ← DOUT _{15:8}
UDLDT	D _{7:0} ← DOUT _{15:8}
XDOUTT	XD _{7:0} ← D _{7:0}
LXDT	D _{7:0} ← XDIN _{7:0}
UXDT	D _{15:8} ← XDIN _{15:8}
p/LDT	D _{7:0} ← p/D _{7:0}
p/UDT	D _{15:8} ← p/D _{15:8}
p/LCE	p/D _{7:0} := D _{7:0}
p/UCE	p/D _{15:8} := D _{15:8}

Table 2: Bus output enables and peripheral clock enables

There are a set of enables p/* within each peripheral. DOUT_{15:0} is the CPU store data output register (see first article). These enable signals are asserted according to transaction type:

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: Enables by transaction type

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} ← P/D/DOUT_{15:0}
 Next, consider a store to external RAM: sw r0, 0x0100. Since the external data bus is only eight bits wide, we first store the least-significant byte, then the most-significant byte. First MEMCTRL asserts LDT and XDOUTT:

XD_{7:0} ← D_{7:0} ← P/D/DOUT_{7:0}.
 Later it asserts UDLDT and XDOUTT:
 XD_{7:0} ← D_{7:0} ← P/D/DOUT_{15:8}.

ON-CHIP BUS – PERIPHERAL INTERFACE

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

It helps to distinguish between core *users* and core *designers*. The former will be more numerous, the latter more experienced. If there are to be ease-of-use tradeoffs, let’s make them in favor of core users.

Since FPGAs are so malleable, and since FPGA SoC design is so new, we need an interface that can evolve to address new requirements, without invalidating existing designs.

With these considerations in mind, I borrowed some ideas from the software world, and defined an *abstract control signal bus*: all of the common control signals are 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 a system designer need only instantiate the core, attach CLK, CTRL, D, some SEL_i, any core-specific inputs and outputs, and they’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, etc. 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 easily done with the DCTRL (CTRL decoder) macro. For example, see the XIN8 schematic (Fig. S7). DCTRL inputs SEL_i, CTRL_{15:0}, and CLK, and outputs local I/O register address and upper and lower byte output enables (read strobes) and clock enables (write strobes).

Within each DCTRL instance we 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 rest of DCTRL is loadless and is automatically eliminated by the FPGA implementation tools.

Using DCTRL and our on-chip tri-state bus, the typical overhead per

peripheral is just one or two CLBs and perhaps a column of TBUFs.

How does control signal abstraction help? So long as we revise MEMCTRL and DCTRL together, we can make arbitrary changes to CTRL_{15:0} without invalidating any existing designs! And to add new bus features, simply design a new decoder DCTRL_v2, causing no changes to existing DCTRL clients.

EXTERNAL PERIPHERAL INTERFACE?

There isn't one. If it were necessary to attach external peripherals, perhaps to the XD_{7:0} bus, we 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 insert wait states, etc., on behalf of its peripheral.

EXTERNAL RAM

The external RAM is a classic 32 KB fast asynchronous SRAM with a 15 ns access time (t_{AA}).

Refer to Figure 2 and the "External Bus and SRAM Interface" block of Figure S6. Table 5 below summarizes the external RAM interface.

FPGA	RAM	Description
XA _{14:1} , XA ₀	A _{14:0}	byte address
XD _{7:0}	D _{7:0}	data bus
RAMNCE	/CS	chip select
RAMNWE	/WE	write enable
RAMNOE	/OE	output enable

Table 4: FPGA/RAM interface

XA_{14:1} are 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} are eight IOBs configured as eight sets of simultaneous OBUFTs (tri-state output buffers), IBUFs (input 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}. These data are input through the IBUFs and also are latched in the XDIN IFDs (on each falling CLK edge).

Read transactions: To keep the CPU busy with fresh new instructions, we contrive to read both bytes of a 16-bit word in one cycle. In the first half cycle, we set XA₀=0, read the MSB, and latch it in XDIN. In the second half cycle, we set XA₀=1, read the LSB, and input 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.

Write transactions: Writes to asynchronous SRAM requires careful design. Can we safely write one byte per clock cycle? Let's see. The key constraints are:

- address valid before asserting /WE;
 - data valid before deasserting /WE;
 - /WE must be deasserted briefly;
 - no address/data hold time after /WE.
- We desire a fully synchronous design (so we can slow or stop the clock) and are therefore unwilling to employ any asynchronous delay tricks.

It follows that we need 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. For example, 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.

Figure 3 shows three transactions:

- read word 1234 from address 0010;
- write word ABCD to address 0200;
- read word 5678 from address 0012.

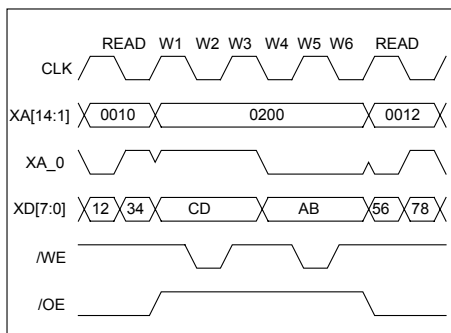


Figure 3: RAM read/write timing

MEMCTRL DESIGN

We have discussed its responsibilities: address decoding, on-chip bus control, and external RAM control. Now let's review its implementation (Fig. S8).

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 derive that peripheral's output and clock enables.

Control: The current state of the memory transaction finite state machine determines which controls are asserted.

The CPU asserts ACE 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 above.

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 asserted immediately, since 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 also uses flip-flops W23_45 and W45 which are clocked on CLK falling edges. (That is, W34 is true during W3 and W4, while W45 is true during W4 and W5). From the W* signals we derive glitch-free control signals XA₀, /WE, /OE, etc.

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

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

Let's see about some peripherals.

PARALLEL PORT I/O

We provide 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, we'll use six inputs and four outputs.

During `sb_r1, FF41`, the PARIN input peripheral is selected; it drives the inputs `00 || PAR_D5:0` onto `D7:0`. We studied its XIN8 implementation above.

During `sb_r1, FF21`, the PAROUT output peripheral is selected. It captures the store data `D3:0` in flip-flops which drive the `PC_S6:3` status outputs.

XOUT4 is as simple as XIN8. It has a DCTRL decoder, of course, and it clocks `D3:0` on LCE (LSB clock enable).

Altogether this parallel port requires only 3 CLBs, 8 TBUFs, and 13 IOBs!

ON-CHIP RAM

Just to show how it's done, we also have a 16x16-bit RAM "peripheral". It uses all of the DCTRL outputs: `A4:1` 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 576x455 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 out these pixels serially, and uses horizontal and vertical sync pulses to "frame" the pixels into frames and lines for the monitor.

We generate VGA compatible horizontal and vertical sync timings. We shift pixels out at 24 MHz, twice our system clock rate, shifting one pixel when CLK is high and a second when CLK is low. Here are the details:

Quantity	Value
two pixel clock	83.3 ns
one pixel half-clock	41.7 ns
visible pixels/line	576
visible clocks/line	288
horizontal sync "on" clock	308
horizontal sync "off" clock	353
line total clocks	381
line time	31.8 μ s

visible lines/frame	455
vertical sync "on" line	486
vertical sync "off" line	488
frame total lines	528
frame time	16.8 ms

Table 5: Video timing

The horizontal and vertical sync pulses are advanced a few clocks (lines), to center the display in the frame.

The video controller has these ports:

Port	Description
PIX _{15:0}	next 16-bit pixel word
REQ	request DMA of next word
RESET	reset DMA address counter
ACK	DMA acknowledge input
CLK	system clock
R1,R0	2-bit red intensity
G1,G0	2-bit green intensity
B1,B0	2-bit blue intensity
NHSYNC	active-low horizontal sync
NVSYNC	active-low vertical sync

Table 6: Video controller ports

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`, and `B1,B0` drive resistor-based 2-bit D-A converters, providing up to $4 \times 4 \times 4 = 64$ different colors. However, at our resolution, with so little RAM, we can only support a monochrome (1-bit/pixel) display, so each pixel bit drives all six outputs, drawing black or white pixels.

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

After 288 clocks, we blank the video. We assert horizontal sync after 308 clocks, deassert it after 353, and reset the counter and re-enable video, after 381 clocks (one line).

Similarly, we 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 since we had just about filled our small FPGA, and because they're cool, I instead designed a cheaper 10-bit

linear feedback shift register (LFSR) counter. This uses a 10-bit serial shift register whose input 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, short circuiting the full sequence, when a particular bit pattern is recognized. So I wrote a program to design arbitrary width, arbitrary cycle LFSR counters. (See the web site.)

Referring to Fig. S9, 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.

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

Video datapath: Each clock we shift out two bits of video data. Every eight clocks WORD goes true and we fetch 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 8x2 parallel-load serial-out shift register.

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

SYSTEM BRING UP

This is the fun part. 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 in my XS40 board. Using a parallel port output for CLK, I wrote shell scripts to single step the processor and observe `PC7:1` 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 “binary search” for the problem, inserting `stop: goto stop;` breaks into my test, recompiling, and downloading. A real remote debugger would be nice!

Armed with a working CPU, it is easy to add and test new features, one at a time. I added double-cycled reads from external RAM, then MEMCTRL, then LED output registers. Writing text messages to the 7-segment LED was a big milestone. RAM writes were next. And late in the project, I added DMA, the video controller, and interrupts.

Let me emphasize the importance and difficulty of thorough testing. It is really hard work to properly test a pipelined processor and a 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.

FLOORPLAN

Here’s the final system floorplan as placed on our 14x14 CLB FPGA:

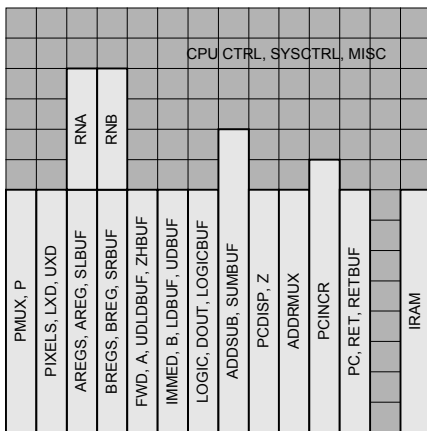


Figure 3: System floorplan

SERIES WRAP-UP

We have reviewed the complete design and implementation of a real, full-featured, pipelined microprocessor and an integrated system-on-a-chip. We designed a new instruction set, and ported a C compiler. We have seen how to build practical CPUs in cheap FPGAs, and we have learned about pipelined processor design.

In *this* article we have seen how to build an integrated system with reusable cores.

The web site provides the full set of schematics, tools, links, and so forth necessary for you to download and build this project, with the help of the Student Edition tools and the XESS XS40-005XL proto board. *Have fun!*

Please note that I do not warrant that you have a right to build something based upon these ideas under the relevant intellectual property laws in your jurisdiction.

THANKS

I am deeply grateful to my wife for her constant support during this project.

Thanks to Tom Cantrell, Philip Freidin, and Brian von Herzen for their great comments and encouragement. (Philip built the first FPGA RISC CPU in an XC4005 way back in 1990!)

This project builds upon the works of Chris Fraser and Dave Hanson (lcc), Dave Vanden Bout (XESS products), and Xilinx (great FPGAs and affordable Student Edition tools).

Ashok Patel inspired these articles. Long ago he and his friends kindly taught this teenager digital design. Later he challenged me to share in kind what I have learned.

Jan Gray is a software developer and amateur computer designer. His products include a leading C++ compiler and an application server. Jan built his first FPGA RISC system in 1995. Write 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] “VGA Signal Generation with the XS Board”, XESS App Note, www.xess.com/fpga/vga.pdf

SOURCES

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

Copyright © 1998-2000, Gray Research LLC. All Rights Reserved.

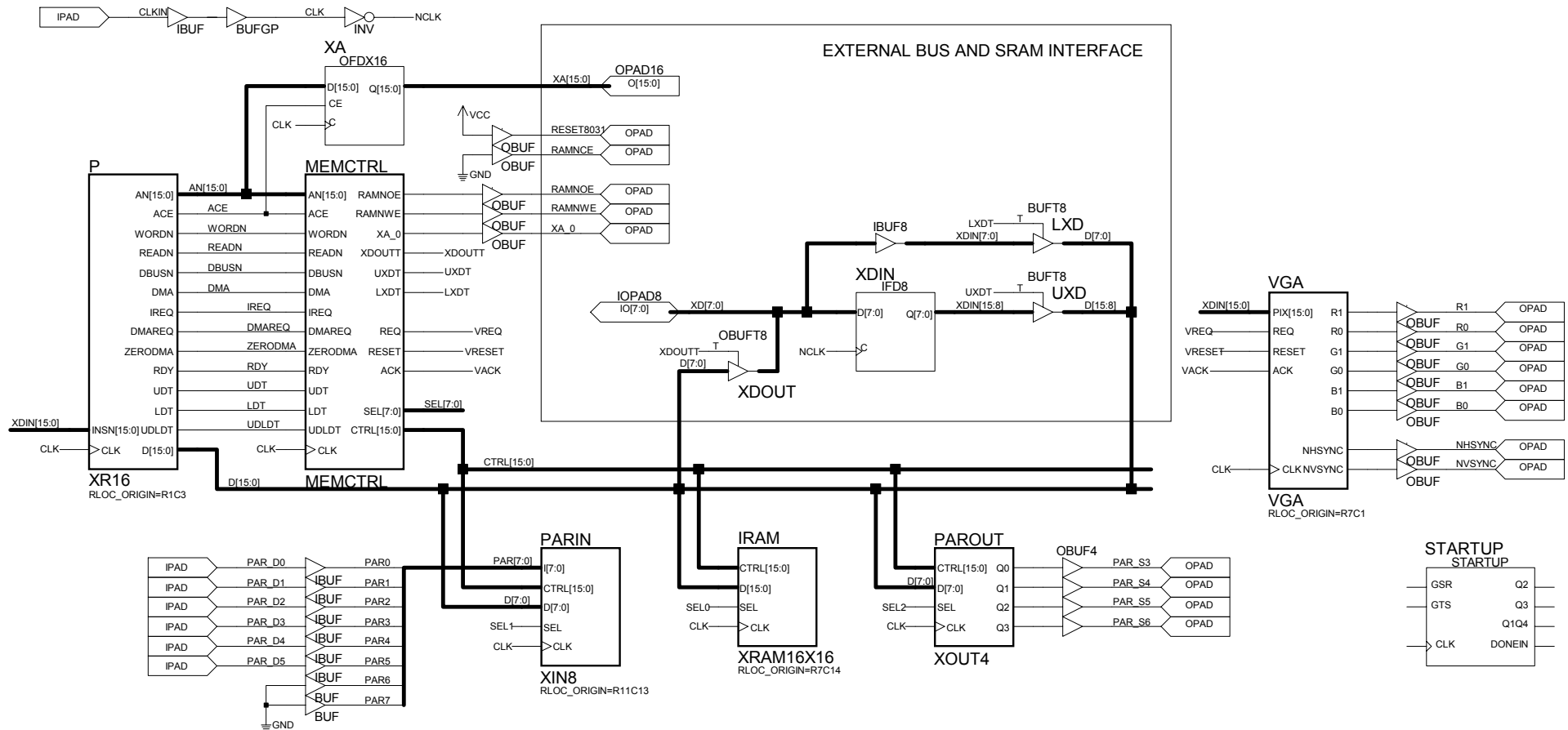


Figure S6: System Schematic

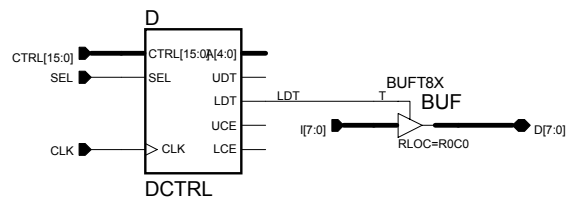
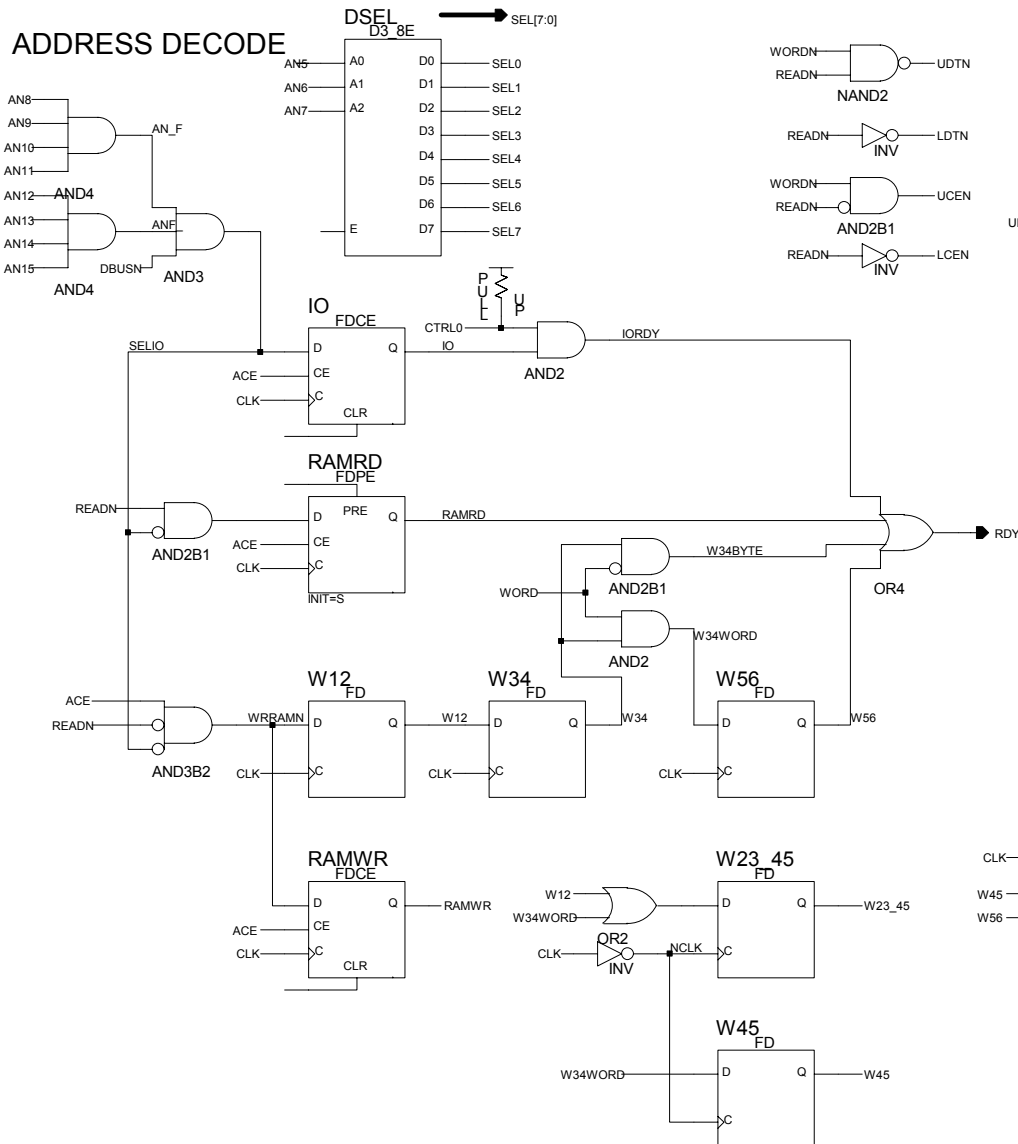
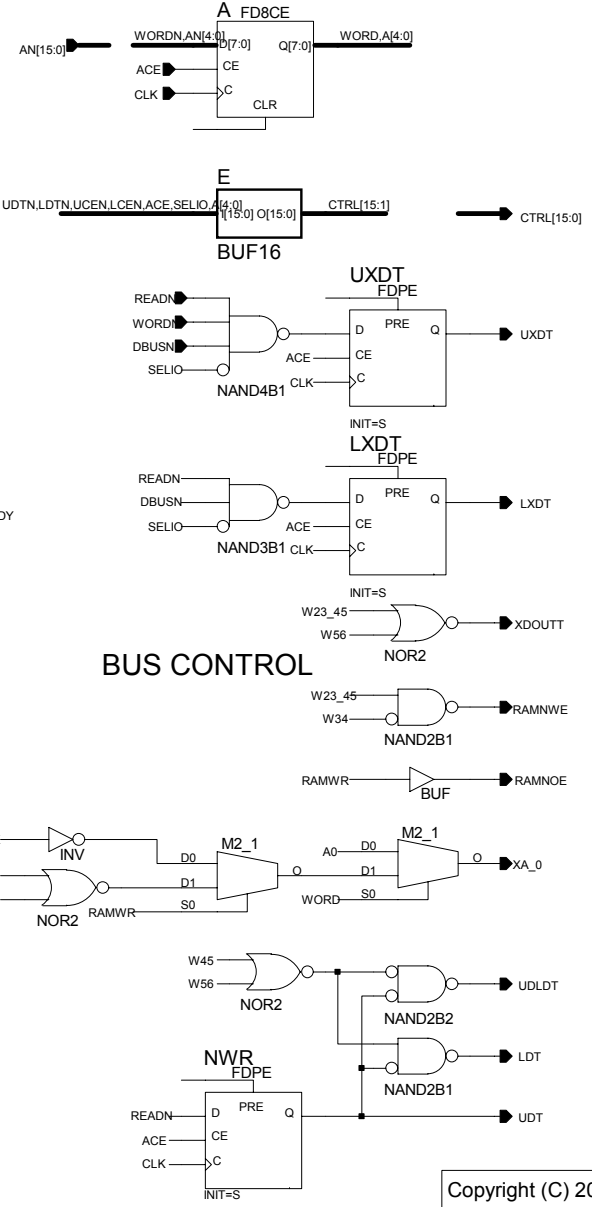


Figure S7: 8-bit Input Peripheral Schematic

ADDRESS DECODE



TRANSACTION STATE MACHINE



BUS CONTROL

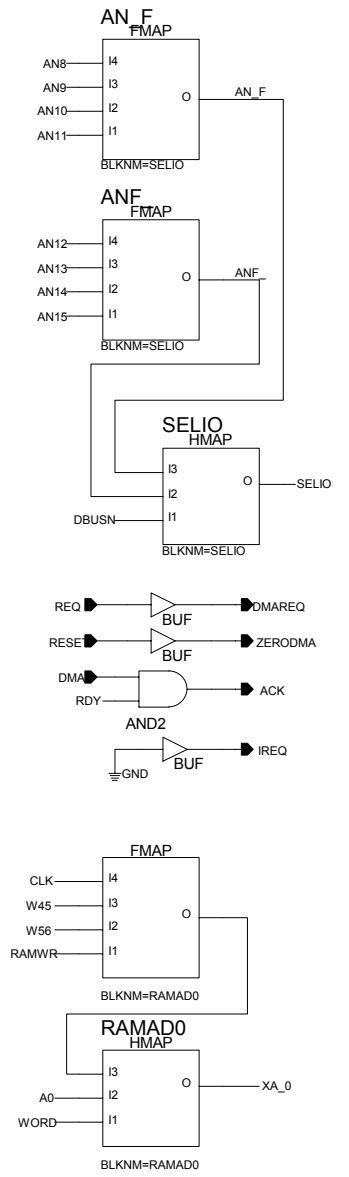


Figure S8: Memory Controller Schematic

Copyright (C) 2000, Gray Research, Inc. Project: [None]
 This work and its use subject to Xilinx Macro: MEMCTRL
 License Agreement. See LICENSE File: 02/23/100

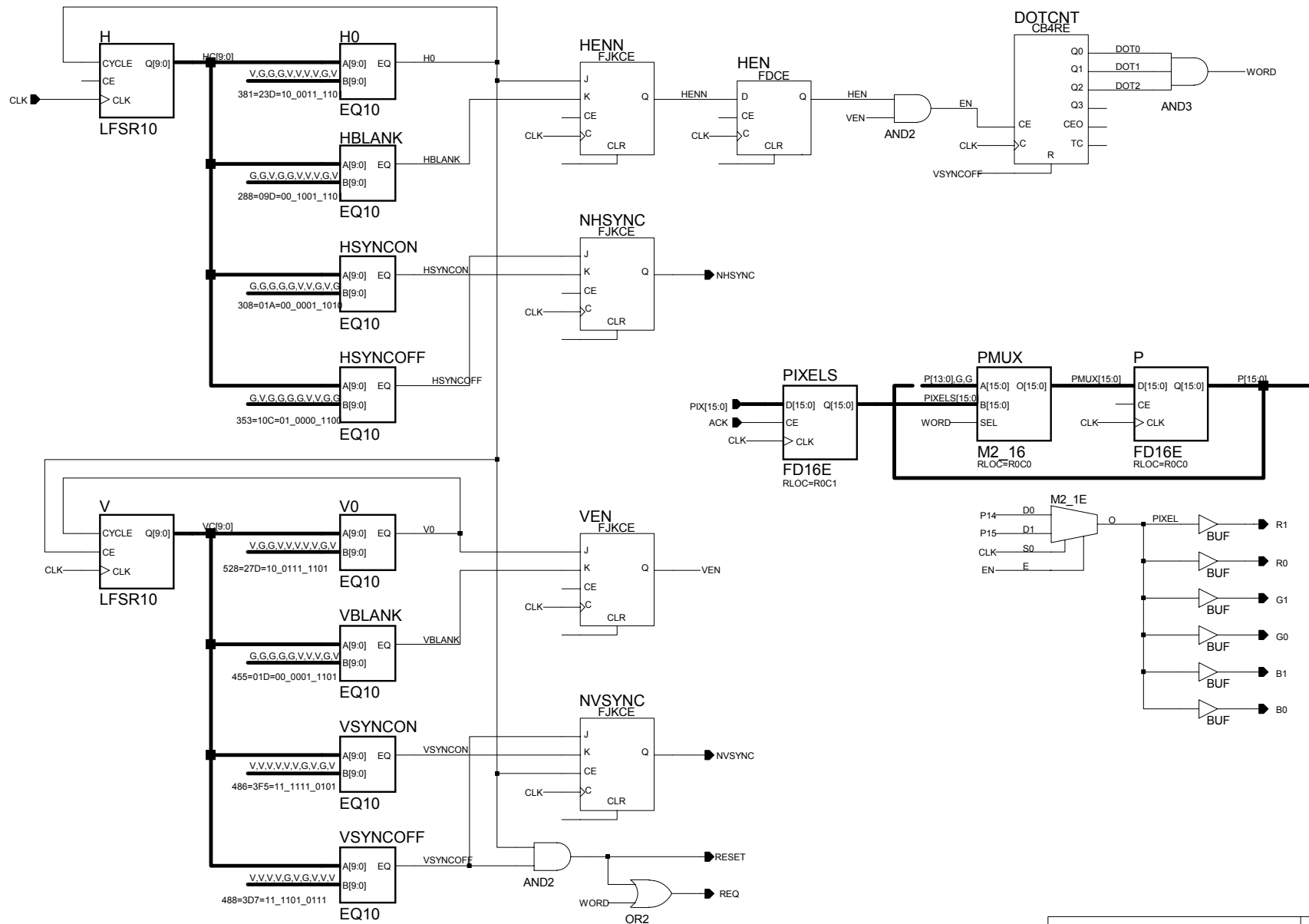


Figure S9: Video Controller Schematic

Copyright (C) 2000, Gray Research	Project: [None]
This work and its use subject to XSD	Macro: VGA
License Agreement. See LICENSE	Date: 02/23/100