

CIRCUIT CELLAR®

THE MAGAZINE FOR COMPUTER APPLICATIONS

Building a RISC System in an FPGA

FEATURE ARTICLE

Jan Gray

Part 2: Pipeline and Control Unit Design

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



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

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

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

PIPELINED EXECUTION

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

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

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

t_1	t_2	t_3	t_4	t_5
IF ₁	DC ₁ IF ₂	EX ₁ DC ₂ IF ₃	EX ₂ DC ₃ IF ₄	EX ₃ DC ₄

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

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

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

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

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

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

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

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

Unlike most pipelined CPUs, the xrl6 only forwards results to the A operand—a speed/area tradeoff. The assembler handles any rare port B data

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

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

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

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

I :      lw r6,2(r10)    ;load r6 with p->flags
IL:     andi r6,7        ;is (p->flags & 7)
I2:     addi r0,r6,-1    ;==1?
I3:     bne T           ;
IB5:    lw r6,6(r10)     ;yes: load r6 with p->y
...

```

hazards by swapping A and B operands, if possible, or inserting nops if not.

MEMORY ACCESSES

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

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

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

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

Then, although you fetched I_3 in t_3 , you must not latch it into the instruction register (IR) as t_3 ends,

because neither EX_L nor DC_2 are finished at this point. In particular, DC_2 must await the load result in order to forward it to A, because I_2 uses r6—the result of I_L !

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

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

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

BRANCHING OUT

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

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

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

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

INTERRUPTS

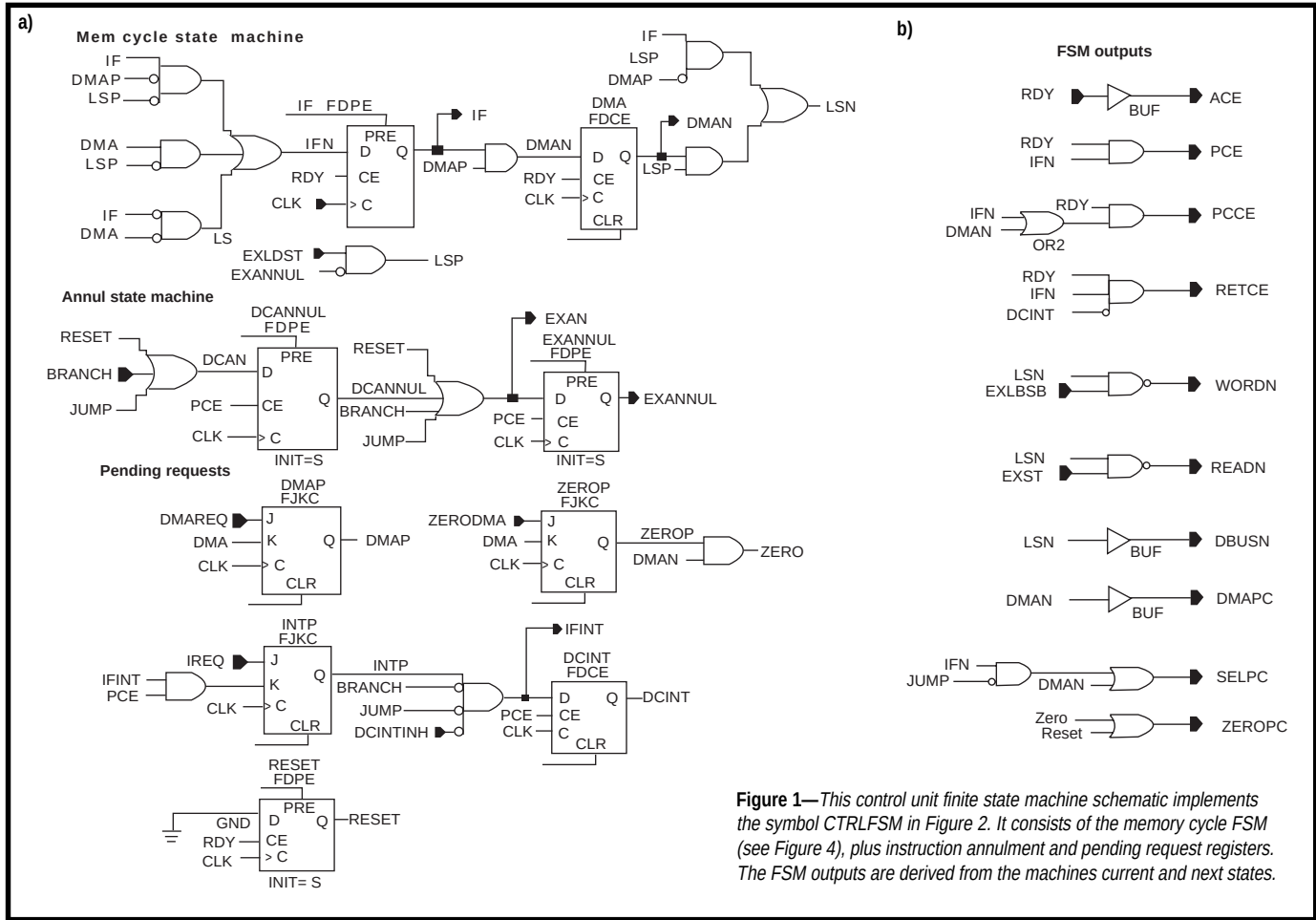
When an interrupt request occurs, you must jump to the interrupt handler, preserve the interrupt return address, retire the current pipeline, execute the handler, and later return to the interrupted instruction.

and leaves the return address in `r14`, which is reserved for this purpose. When the handler has completed, it executes `iret`, (i.e. `jal r0,0(r14)`) and execution resumes with the interrupted instruction.

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

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

After an instruction is fetched, if



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

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

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

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

This PC register file costs eight CLBs for the RAM, but saves 16 CLBs (otherwise necessary for a separate 16-bit DMA address counter and a 16-bit 2-1 address mux), and shaves a couple

of nanoseconds from the system’s critical path. It’s a nice example of a problem-specific optimization you can build with a customizable processor.

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

CONTROL UNIT DESIGN

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

- fix are DC stage signals):
- 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

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

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

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

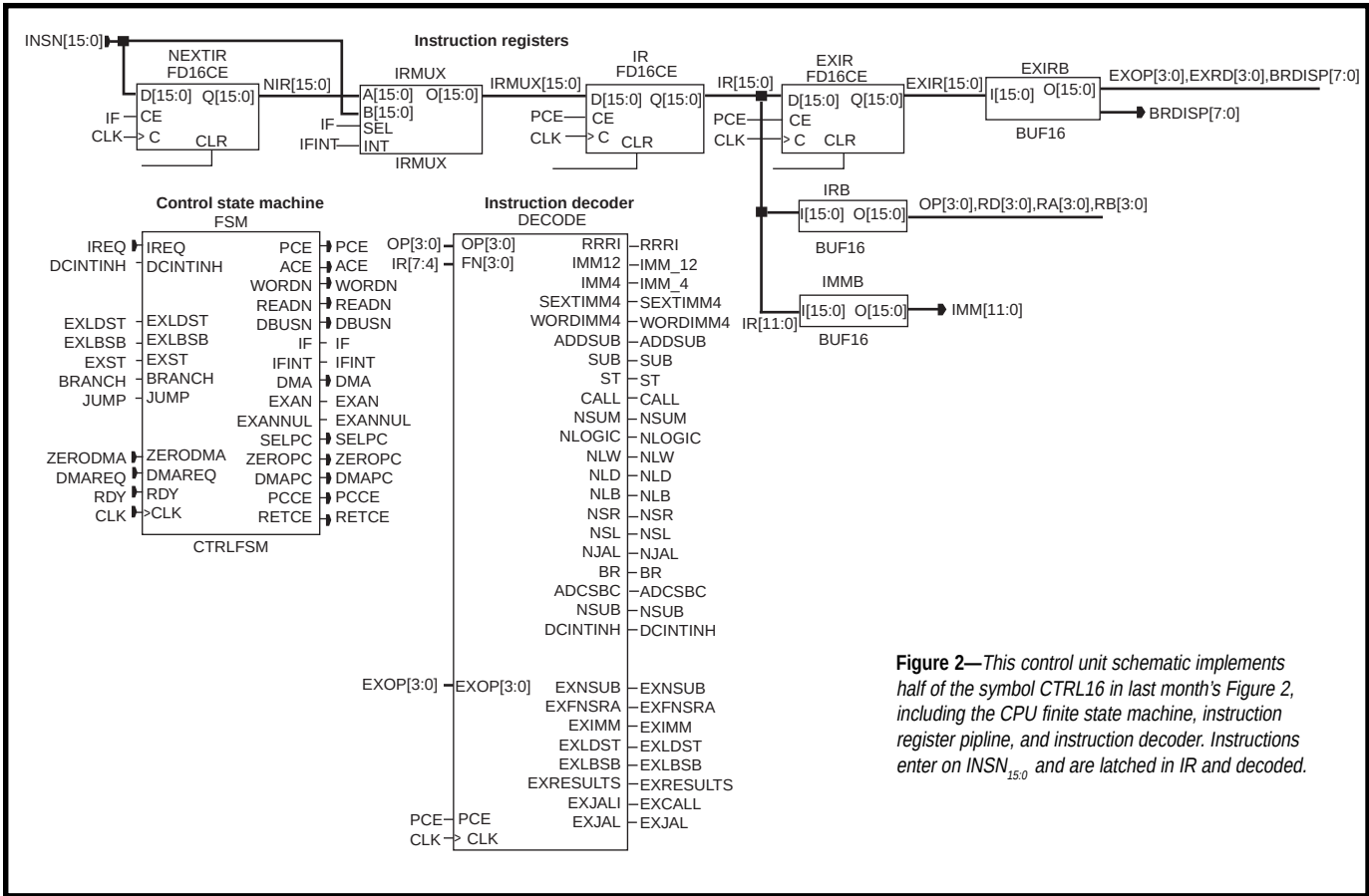


Figure 2—This control unit schematic implements half of the symbol CTRL16 in last month’s Figure 2, including the CPU finite state machine, instruction register pipeline, and instruction decoder. Instructions enter on $INSN_{15:0}$ and are latched in IR and decoded.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

0xAE01

If a DMA or load/store access is pending, IF enables NEXTIR to capture the previously fetched instruction (take a look back at time t₃ in Table 3). Otherwise, the instruction fetch is the only memory access in the pipe stage. So, IF is then asserted with PCE, and IRMUX selects the INSN_{15:0} input as the next instruction to complete.

DECODE STAGE

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

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

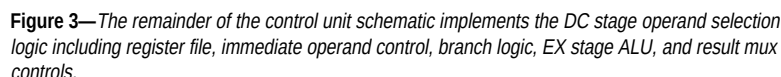
The instruction decoder DECODE is simple. It is a set of 30 ROM 16x1s, gate expressions, and a handful of flip-flops. Each ROM inputs OP_{3:0} or EXOP_{3:0} and outputs some decoded signal. The decoder is relatively compact because xr16 has a simple instruction set, and its 4-bit opcodes are a good match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

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

Next cycle	Next address	Outputs
IF	AN ← PC ₀ += 2	SELPC PCCE
IF branch	AN ← PC ₀ += 2×disp8	BRANCH SELPC PCCE
IF jal call	AN ← PC ₀ = SUM	PCCE
IF reset	AN ← PC ₀ = 0	SELPC ZERO PCCE
LS load/store	AN ← SUM	—
DMA	AN ← PC ₁ += 2	SELPC DMAPC PCCE
DMA reset	AN ← PC ₁ = 0	SELPC ZERO PCCE

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

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



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

- **BRANCH**: if set, add $2 \times \text{disp8}$ to PC, otherwise add +2
- **SELPC**: if set, next address is $\text{PCNEXT}_{15:0}$ otherwise $\text{SUM}_{15:0}$
- **ZEROPC**: if set, next address is 0
- **PCCE** (PC clock enable): update PC.

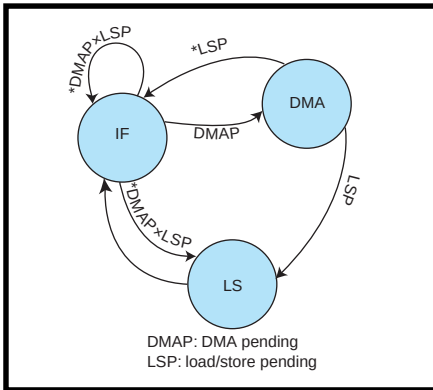


Figure 4—Each memory cycle is an instruction fetch unless there is a DMA transfer pending or the EX stage instruction is a load or store. The FSM clocks when one memory transaction completes and another begins (on RDY).

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

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

WRAP-UP

This month, we considered pipelined processor design issues and explored the detailed implementation of our xr16 control unit—and lived! The CPU design is complete. The final article in this series will describe the final system on a chip. Building FPGA processors and systems since 1994, and he now designs for Gray Research LLC. You may reach him at jan@fpgacpu.org.

SOFTWARE

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

REFERENCE

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