The Fred VHDL Model

William F. Richardson

UUCS-95-021

Department of Computer Science
University of Utah
Salt Lake City, UT 84112 USA

November 21, 1995

Abstract

This is the companion document to my dissertation. It contains 47 pages of schematics, and 163 pages of VHDL code. It is pretty meaningless without the dissertation, and it only exists because I felt that I should archive this information somewhere.

Contents

List of Figures

Chapter 1

Schematic Heirarchy

This is the schematic hierarchy. “fred-with-mem” is the top level schematic. The elements listed in parentheses are VHDL modules. Any schematics not listed in this hierarchy are standard library modules.

```
fred-with-mem
    fred.1
        if_unit.1
            (dispatch)
            gate
            qflop_even
                (qflop)
            qflop_odd
                (qflop)
        if_unit.2
            (scoreboard)
            arbcall
                (arbiter_which)
            mux2x5
        pipe_nx128
            (cpx128)
            delaychain
                (delaytap zxor)
                cp_delay
        pipe_nx18
            (cpx18)
            delaychain
                (delaytap zxor)
                cp_delay
    fred.2
        pipe_nx5
            (cpx5)
            delaychain
                (delaytap zxor)
                cp_delay
        pipe_nx96
            (cpx96)
            delaychain
                (delaytap zxor)
                cp_delay
        rf_unit
            (registers regselect)
    fred.3
        ex_unit
```

```
            distributor
            fu_arith
                (arith_unit)
                pipe_nx3op
                    (cpx3op)
                    delaychain
                        (delaytap zxor)
                        cp_delay
            fu_control
                (control_unit)
                pipe_nx4op
                    (cpx4op)
                    delaychain
                        (delaytap zxor)
                        cp_delay
            fu_logic
                (logic_unit)
                pipe_nx3op
                    (cpx3op)
                    delaychain
                        (delaytap zxor)
                        cp_delay
            mux2x32
            zerox32
                buf8
        ex_unit.2
            arbcallx88
                (arbiter_which vdelay)
                mux2x32
                mux2x8
            callx32
                (vdelay)
                mux2x32
            fu_branch
                (branch_unit)
                pipe_nx4op
                    (cpx4op)
                    delaychain
                        (delaytap zxor)
                        cp_delay
            fu_memory
                (memory_unit)
                pipe_nx4op
                    (cpx4op)
                    delaychain
                        (delaytap zxor)
                        cp_delay
        pipe_nx32
            (cpx32)
            delaychain
                (delaytap zxor)
                cp_delay
        pipe_nx88
            (cpx88)
            delaychain
                (delaytap zxor)
                cp_delay
    fred-memory
        (arbiter_which verify_address zdelay)
```

```
memory_latches
    (vdelay)
    buf32
    buf4
    fifo-32
        tlnt-32
    fifo-36
        tlnt-32
mux2x30
mux2x4
ram16kx32
    (ram16kx8 vdelay)
    ram_byte_order
```

Chapter 2

VHDL Source Code

2.1 arbiter_which.vhd

```
---- This models an arbiter with an extra level output to indicate which choice
---- it made.  It uses a state machine with 5 states.  There is no provision
---- for stopping or resetting it, and it assumes that the inputs are
---- well-behaved.

---- The "w" output changes instantly, as soon as it can.  The grant signal is
---- subject to a delay (default .1ns), changeable with a DELAY attribute.

ENTITY arbiter_which IS
  GENERIC( delay: TIME := 100 PS);
  PORT (SIGNAL clr      :IN    VLBIT;
        SIGNAL r0       :IN    VLBIT;
        SIGNAL r1       :IN    VLBIT;
        SIGNAL d0       :IN    VLBIT;
        SIGNAL d1       :IN    VLBIT;
        SIGNAL g0       :OUT   VLBIT := '0';
        SIGNAL g1       :OUT   VLBIT := '0';
        SIGNAL w        :OUT   VLBIT := '0');
END arbiter_which;

ARCHITECTURE state_behavior OF arbiter_which IS
  TYPE states IS (wait_any,wait_d0,wait_d1,wait_d0_d1,wait_d1_d0);
BEGIN
  main : PROCESS ( r0, r1, d0, d1 )
    VARIABLE  state            : states := wait_any;
  BEGIN

    IF (clr /= '1') THEN
      g0 <= '0';
      g1 <= '0';
      w  <= '0';
      state := wait_any;

    ELSE

      CASE state IS
```

```
WHEN wait_any =>                          -- waiting for any request
  IF (r0'EVENT AND r1'EVENT) THEN         -- got both
    w <= '0';
    g0 <= r0 AFTER delay;                 -- do r0 first
    state := wait_d0_d1;
  ELSIF (r0'EVENT) THEN                   -- else got r0 only
    w <= '0';
    g0 <= r0 AFTER delay;
    state := wait_d0;
  ELSIF (r1'EVENT) THEN                   -- or got r1 only
    w <= '1';
    g1 <= r1 AFTER delay;
    state := wait_d1;
  ELSE                                    -- no other choice
    putline("*** Illegal Input to Arbiter in state WAIT_ANY ***");
  END IF;

WHEN wait_d0 =>                           -- waiting for d0 or r1
  IF (d0'EVENT AND r1'EVENT) THEN         -- got both
    w <= '1';
    g1 <= r1 AFTER delay;                 -- assume d0 first
    state := wait_d1;
  ELSIF (d0'EVENT) THEN                   -- got d0, go back to start
    state := wait_any;
  ELSIF (r1'EVENT) THEN                   -- else got r1
    state := wait_d0_d1;
  ELSE
    putline("*** Illegal Input to Arbiter in state WAIT_D0 ***");
  END IF;

WHEN wait_d1 =>                           -- waiting for d1 or r0
  IF (d1'EVENT AND r0'EVENT) THEN         -- got both
    w <= '0';
    g0 <= r0 AFTER delay;                 -- assume d1 first
    state := wait_d0;
  ELSIF (d1'EVENT) THEN                   -- got d1, go back to start
    state := wait_any;
  ELSIF (r0'EVENT) THEN                   -- else got r0
    state := wait_d1_d0;
  ELSE
    putline("*** Illegal Input to Arbiter in state WAIT_D1 ***");
  END IF;

WHEN wait_d0_d1 =>                        -- waiting for d0, r1 in queue
  IF (d0'EVENT) THEN
    w <= '1';                             -- d0 is only acceptible event
    g1 <= r1 AFTER delay;
    state := wait_d1;
  ELSE
    putline("*** Illegal Input to Arbiter in state WAIT_D0_D1 ***");
  END IF;

WHEN wait_d1_d0 =>                        -- waiting for d1, r0 in queue
```

```
        IF (d1'EVENT) THEN
          w <= '0';                                -- d0 is only acceptible event
          g0 <= r0 AFTER delay;
          state := wait_d0;
        ELSE
          putline("*** Illegal Input to Arbiter in state WAIT_D1_D0 ***");
        END IF;

      END CASE;                                    --  state

    END IF;

  END PROCESS main;
END state_behavior;
```

2.2 arith_unit.vhd

```
USE work.p_constants.ALL;
USE work.p_arith.ALL;
USE work.p_output.ALL;

---- This unit handles the arithmetic instructions. Currently, there is only
---- one of these units. More could be used, but a separate carry bit is
---- maintained internally to each unit.

---- The EXTRA input was added to provide last-result-reuse and to eliminate
---- zero values of op_a for immediate instructions. The bit usage is:
----
---- EXTRA(0) = reuseA
---- EXTRA(1) = reuseB
---- EXTRA(2) = reuseD
---- EXTRA(3) = zeroA (unused)

ENTITY arith_unit IS
  GENERIC( delay: TIME := 100 ps );
  PORT( SIGNAL clr            : IN  VLBIT;
        SIGNAL rin            : IN  VLBIT;
        SIGNAL ain            : OUT VLBIT;
        SIGNAL tellme         : IN  VLBIT;
        SIGNAL tag            : IN  VLBIT_1D (7 DOWNTO 0);
        SIGNAL dest           : IN  VLBIT_1D (4 DOWNTO 0);
        SIGNAL opcode         : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL op_a           : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL op_b           : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL d_req          : OUT VLBIT;
        SIGNAL d_ack          : IN  VLBIT;
        SIGNAL done           : OUT VLBIT_1D (87 DOWNTO 0);
        SIGNAL rout           : OUT VLBIT;
        SIGNAL aout           : IN  VLBIT;
        SIGNAL res            : OUT VLBIT_1D (36 DOWNTO 0);
        SIGNAL rq_req         : OUT VLBIT;
        SIGNAL rq_ack         : IN  VLBIT;
        SIGNAL rq             : OUT VLBIT_1D (31 DOWNTO 0);
        SIGNAL extra          : IN  VLBIT_1D (3 DOWNTO 0) );
END arith_unit;


ARCHITECTURE behavior OF arith_unit IS
BEGIN
  main : PROCESS

    VARIABLE debug            : BOOLEAN := FALSE; -- display everything

    --- factors affecting response time

    VARIABLE delay_add        : TIME := 100 ps;
    VARIABLE delay_sub        : TIME := 100 ps;
    VARIABLE delay_cmp        : TIME := 100 ps;
    VARIABLE delay_div        : TIME := 100 ps;
    VARIABLE delay_mul          : TIME := 100 ps;

    VARIABLE delay_to_use       : TIME;         -- scratch for culumative time


    --- constants

    CONSTANT MINOR_add          : INTEGER := 1;
    CONSTANT MINOR_sub          : INTEGER := 2;
    CONSTANT MINOR_div          : INTEGER := 3;


    --- output variables

    VARIABLE v_rout             : VLBIT;
    VARIABLE v_rq_req           : VLBIT;
    VARIABLE v_d_req            : VLBIT;


    --- internal variables

    VARIABLE index              : INTEGER;
    VARIABLE status             : INTEGER;
    VARIABLE minor              : INTEGER;

    VARIABLE Av                 : VLBIT_1D (31 DOWNTO 0);
    VARIABLE Bv                 : VLBIT_1D (31 DOWNTO 0);
    VARIABLE result             : VLBIT_1D (31 DOWNTO 0);
    VARIABLE lastresult         : VLBIT_1D (31 DOWNTO 0);
    VARIABLE bigresult          : VLBIT_1D (33 DOWNTO 0);

    VARIABLE Ai                 : INTEGER;
    VARIABLE Bi                 : INTEGER;
    VARIABLE Ri                 : INTEGER;

    VARIABLE carry              : VLBIT;


    --- scratch variables

    VARIABLE use_carry_in       : BOOLEAN;
    VARIABLE use_carry_out      : BOOLEAN;

    VARIABLE tmpi               : INTEGER;
    VARIABLE tmpv               : VLBIT_1D (31 DOWNTO 0);
    VARIABLE arg1               : VLBIT_1D (31 DOWNTO 0);
    VARIABLE arg2               : VLBIT_1D (31 DOWNTO 0);


    --- statistics

    VARIABLE s_arith_rin        : INTEGER;
    VARIABLE s_mul              : INTEGER;
    VARIABLE s_div              : INTEGER;
```

```
BEGIN


  --- If clear is asserted, reset everything and stick here.

  IF clr /= '1' THEN                          -- reset to beginning

    v_rout := '0';
    v_rq_req := '0';
    v_d_req := '0';
    carry := '0';
    arg1 := ZERO(31 DOWNTO 0);
    arg2 := ZERO(31 DOWNTO 0);

    ain <= '0';
    d_req <= v_d_req;
    done <= ZERO(87 DOWNTO 0);
    rout <= v_rout;
    res <= ZERO(36 DOWNTO 0);
    rq_req <= v_rq_req;
    rq <= ZERO(31 DOWNTO 0);

    s_arith_rin := 0;
    s_mul := 0;
    s_div := 0;

    WAIT UNTIL clr = '1';


  ELSE                                        -- only wakeup event is RIN

    status := STATUS_done;                    -- assume it will work
    minor := 0;

    --- This stuff had to be encoded by hand. I could have written a Perl
    --- script to do it, but it would have taken longer than doing it
    --- manually. This stuff shouldn't change that much (I hope). I also made
    --- some assumptions about unused bits, which is probably okay.


    delay_to_use := delay;                    -- default is one gate

    --- I don't bother to compute anything if I'm just going to throw the
    --- result away.

    IF dest /= REG_R0 THEN -- only if worth doing

      s_arith_rin := s_arith_rin + 1;


      index := op2index( opcode );            -- identify instruction
```

```
--- determine operand sources

IF extra(0)='1' THEN
  Av := lastresult;
ELSE
  Av := op_a;
END IF;

IF index <= 31 THEN                    -- immediate
  --- All Immediate operands are zero-extended (this could change).
  Bv := ZERO(31 DOWNTO 16) & opcode(15 DOWNTO 0);
ELSE                                   -- triadic
  IF extra(1)='1' THEN
    Bv := lastresult;
  ELSE
    Bv := op_b;
  END IF;
END IF;

Ai := v1d2int( Av );
Bi := v1d2int( Bv );


--- determine carry usage

use_carry_in := FALSE;
use_carry_out := FALSE;

IF ( index = 40 OR                     -- add rd,ra,rb
     index = 41 OR                     -- addu rd,ra,rb
     index = 46 OR                     -- sub rd,ra,rb
     index = 47 )                      -- subu rd,ra,rb
THEN
  use_carry_in := vlb2boo( opcode(9) );
  use_carry_out := vlb2boo( opcode(8) );
END IF;


IF debug THEN
  puthexline("op_a is ",Ai);
  puthexline("op_b is ",Bi);
  putline("use_carry_in is ",use_carry_in);
  putline("use_carry_out is ",use_carry_out);
END IF;


--- now, do the operation

CASE index IS

WHEN 8 | 40 =>                         -- add

  IF debug THEN
```

```
    putline("ADD");
  END IF;

  delay_to_use := delay_add;

  IF use_carry_in THEN
    bigresult := add32( Av, Bv, carry );
  ELSE
    bigresult := add32( Av, Bv, '0' );
  END IF;

  --- Overflow is possible, so check for it.

  IF bigresult(33) = '1' THEN           -- overflow exception

    status := STATUS_overflow;          -- report it
    minor := MINOR_add;
    arg1 := Av;
    arg2 := Bv;

  ELSE                                  -- no exception

    IF use_carry_out THEN
      carry := bigresult(32);           -- save carry if enabled
    END IF;

    result := bigresult(31 DOWNTO 0);

  END IF;



WHEN 9 | 41  =>                         -- addu

  IF debug THEN
    putline("ADDU");
  END IF;

  delay_to_use := delay_add;

  --- Just do it.  No faults.

  IF use_carry_in THEN
    bigresult := add32( Av, Bv, carry );
  ELSE
    bigresult := add32( Av, Bv, '0' );
  END IF;

  IF use_carry_out THEN
    carry := bigresult(32);             -- save carry if enabled
  END IF;

  result := bigresult(31 DOWNTO 0);
```

```
WHEN 10 | 42 =>                         -- div

  s_div := s_div + 1;

  IF debug THEN
    putline("DIV");
  END IF;

  delay_to_use := delay_div;

  --- Signed divide. Overflow is possible only for NEGMAX / -1.
  --- DivByZero is possible.

  IF Bi = 0 THEN                        -- divide by zero

    status := STATUS_int_div;
    minor := MINOR_div;
    arg1 := Av;
    arg2 := Bv;

  ELSIF Bi = -1 AND Av = NEGMAX THEN    -- overflow

    status := STATUS_overflow;
    minor := MINOR_div;
    arg1 := Av;
    arg2 := Bv;

  ELSE                                  -- just do it

    Ri := Ai / Bi;                      -- do it
    result := int2v1d( Ri );            -- save it

  END IF;



WHEN 11 | 43 =>                         -- divu

  s_div := s_div + 1;

  IF debug THEN
    putline("DIVU");
  END IF;

  delay_to_use := delay_div;

  --- Unsigned divide. DivByZero is possible, but overflow is not.

  IF Bi = 0 THEN                        -- divide by zero

    status := STATUS_int_div;
    minor := MINOR_div;
```

```
      arg1 := Av;
      arg2 := Bv;

    ELSE                                  -- just do it

      result := divum( Av, Bv );          -- treat as unsigned values

    END IF;



  WHEN 12 | 44 =>                         -- mul

    s_mul := s_mul + 1;

    IF debug THEN
      putline("MUL");
    END IF;

    delay_to_use := delay_mul;

    --- Just do it.  No faults.

    --- Shouldn't this be pipelined?  How deep?

    Ri := Ai * Bi;
    result := int2v1d( Ri );



  WHEN 13 | 45 =>                         -- cmp

    IF debug THEN
      putline("CMP");
    END IF;

    delay_to_use := delay_cmp;

    --- Do all sorts of comparisons. No faults.

    result := ZERO(31 DOWNTO 0);          -- initialize

    result(11) := boo2vlb( Av >= Bv );    -- hs
    result(10) := boo2vlb( Av < Bv );     -- lo
    result(9) := boo2vlb( Av <= Bv );     -- ls
    result(8) := boo2vlb( Av > Bv );      -- hi

    result(7) := boo2vlb( Ai >= Bi );     -- ge
    result(6) := boo2vlb( Ai < Bi );      -- lt
    result(5) := boo2vlb( Ai <= Bi );     -- le
    result(4) := boo2vlb( Ai > Bi );      -- gt
    result(3) := boo2vlb( Ai /= Bi );     -- ne
    result(2) := boo2vlb( Ai = Bi );      -- eq
```

```
WHEN 14 | 46 =>                         -- sub

  IF debug THEN
    putline("SUB");
  END IF;

  delay_to_use := delay_sub;

  IF use_carry_in THEN
    bigresult := sub32( Av, Bv, carry );
  ELSE
    bigresult := sub32( Av, Bv, '1' );
  END IF;

  --- Overflow is possible, so check for it.

  IF bigresult(33) = '1' THEN           -- overflow exception

    status := STATUS_overflow;          -- report it
    minor := MINOR_sub;
    arg1 := Av;
    arg2 := Bv;

  ELSE                                  -- no exception

    IF use_carry_out THEN
      carry := bigresult(32);           -- save carry if enabled
    END IF;

    result := bigresult(31 DOWNTO 0);

  END IF;



WHEN 15 | 47 =>                         -- subu

  IF debug THEN
    putline("SUBU");
  END IF;

  delay_to_use := delay_sub;

  --- Just do it.  No faults.

  IF use_carry_in THEN
    bigresult := sub32( Av, Bv, carry );
  ELSE
    bigresult := sub32( Av, Bv, '1' );
  END IF;

  IF use_carry_out THEN
```

```
          carry := bigresult(32);             -- save carry if enabled
        END IF;

        result := bigresult(31 DOWNTO 0);



      WHEN OTHERS =>                          -- anything else
        putline("*** Illegal Opcode for Arith Unit ***");
      END CASE;


      --- write output if any was produced.

      IF status = STATUS_done THEN

        IF debug THEN
          tmpi := v1d2int(result);
          puthexline("Result is ",tmpi);
        END IF;

        IF dest = REG_R1 THEN

          rq <= result;
          v_rq_req := NOT v_rq_req;
          rq_req <= v_rq_req AFTER delay_to_use;

          WAIT ON rq_ack;

        ELSE

          res <= dest & result;

          lastresult := result;

          IF debug THEN
            puthexline("latched ",tmpi);
          END IF;

          v_rout := NOT v_rout;               -- write it out
          rout <= v_rout AFTER delay_to_use;

          WAIT ON aout;

        END IF;

      END IF;

    END IF;                                   -- produce a result


    --- signal completion of this instruction

    IF tellme = '1' THEN                      -- only if needed
```

```
        tmpi := status + 256 * minor;
        tmpv := int2v1d( tmpi );
        done <= arg1 & arg2 & tmpv(15 DOWNTO 0) & tag;

        IF debug THEN
          puthexline("Sending status ",tmpi);
        END IF;

        v_d_req := NOT v_d_req;
        d_req <= v_d_req AFTER delay;

        WAIT ON d_ack;

      END IF;

      ain <= rin AFTER delay;                 -- done with this event

      IF debug THEN
        putline("Done with this instruction");
      END IF;

    END IF;                                   -- clr not asserted

    WAIT ON clr, rin;                         -- wait for next event

  END PROCESS main;

END behavior;
```

2.3 branch_unit.vhd

```
USE work.p_constants.ALL;
USE work.p_output.ALL;

---- The branch_unit computes branch targets for the IF Unit. It may place the
---- result in either the Branch Queue or the Exception Branch Queue.  If the
---- branch will be taken, it places the address on the Prefetch Queue.


---- The EXTRA input was added to provide last-result-reuse and to elminate
---- zero values of op_a for immediate instructions. The bit usage is:
----
---- EXTRA(0) = reuseA
---- EXTRA(1) = reuseB
---- EXTRA(2) = reuseD
---- EXTRA(3) = zeroA (unused)
----
---- The branch unit doesn't need it for anything, since it doesn't generate
---- results for the register file.



ENTITY branch_unit IS
  GENERIC( delay: TIME := 100 ps );
  PORT( SIGNAL clr              : IN  VLBIT;
        SIGNAL rin              : IN  VLBIT;
        SIGNAL ain              : OUT VLBIT;
        SIGNAL tellme           : IN  VLBIT;
        SIGNAL tag              : IN  VLBIT_1D (7 DOWNTO 0);
        SIGNAL dest             : IN  VLBIT_1D (4 DOWNTO 0);
        SIGNAL pc               : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL opcode           : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL op_a             : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL op_b             : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL cr0              : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL d_req            : OUT VLBIT;
        SIGNAL d_ack            : IN  VLBIT;
        SIGNAL done             : OUT VLBIT_1D (87 DOWNTO 0);
        SIGNAL br_req           : OUT VLBIT;
        SIGNAL br_ack           : IN  VLBIT;
        SIGNAL br               : OUT VLBIT_1D (31 DOWNTO 0);
        SIGNAL xbr_req          : OUT VLBIT;
        SIGNAL xbr_ack          : IN  VLBIT;
        SIGNAL xbr              : OUT VLBIT_1D (31 DOWNTO 0);
        SIGNAL pf_req           : OUT VLBIT;
        SIGNAL pf_ack           : IN  VLBIT;
        SIGNAL pf               : OUT VLBIT_1D (31 DOWNTO 0);
        SIGNAL extra            : IN  VLBIT_1D (3 DOWNTO 0) );
END branch_unit;


ARCHITECTURE behavior OF branch_unit IS
BEGIN
```

```
main : PROCESS

  VARIABLE debug              : BOOLEAN := FALSE; -- display everything

  --- factors affecting response time

  VARIABLE delay_abs          : TIME := 100 ps;
  VARIABLE delay_rel          : TIME := 100 ps;
  VARIABLE delay_cond_abs     : TIME := 100 ps;
  VARIABLE delay_cond_rel     : TIME := 100 ps;

  VARIABLE delay_to_use       : TIME;         -- scratch for culumative time


  --- output variables

  VARIABLE v_pf_req           : VLBIT;
  VARIABLE v_d_req            : VLBIT;
  VARIABLE v_br_req           : VLBIT;
  VARIABLE v_xbr_req          : VLBIT;


  --- internal variables

  VARIABLE index              : INTEGER;
  VARIABLE A                  : INTEGER;
  VARIABLE B                  : INTEGER;
  VARIABLE addr               : VLBIT_1D( 30 DOWNTO 0);
  VARIABLE taken              : BOOLEAN;


  --- scratch variables

  VARIABLE tmpv               : VLBIT_1D( 31 DOWNTO 0);
  VARIABLE tmpi               : INTEGER;
  VARIABLE tmpb               : VLBIT;

  --- statistics

  VARIABLE s_branch_rin       : INTEGER;
  VARIABLE s_taken            : INTEGER;
  VARIABLE s_relative         : INTEGER;
  VARIABLE s_indirect         : INTEGER;


BEGIN


  --- If clear is asserted, reset everything and stick here.

  IF clr /= '1' THEN                          -- reset to beginning

    v_pf_req := '0';
    v_d_req := '0';
```

```
      v_br_req := '0';
      v_xbr_req := '0';

      ain <= '0';
      pf_req <= v_pf_req;
      pf <= ZERO(31 DOWNTO 0);
      d_req <= v_d_req;
      tmpv := ZERO(31 DOWNTO 0);
      done <= ZERO(87 DOWNTO 0);
      br_req <= v_br_req;
      br <= ZERO(31 DOWNTO 0);
      xbr_req <= v_xbr_req;
      xbr <= ZERO(31 DOWNTO 0);

      s_branch_rin := 0;
      s_taken := 0;
      s_indirect := 0;
      s_relative := 0;

      WAIT UNTIL clr = '1';


    ELSE                                        -- only wakeup event is RIN

      s_branch_rin := s_branch_rin + 1;

      --- This stuff had to be encoded by hand. I could have written a Perl
      --- script to do it, but it would have taken longer than doing it
      --- manually. This stuff shouldn't change that much (I hope). I also made
      --- some assumptions about unused bits, which is probably okay.

      delay_to_use := delay;

      IF debug THEN
        tmpi := v1d2int(op_a);
        puthexline("op_a is ",tmpi);
      END IF;

      index := op2index( opcode );

      IF index = 87 THEN                        -- LDBR is special

        IF debug THEN
          tmpi := 4 * v1d2int(op_a(31 DOWNTO 2));
          puthex("LDBR <= ",tmpi);
          tmpb := op_a(1);
          putline("  ",tmpb);
        END IF;

        br <= op_a;                             -- always use Branch Queue
        v_br_req := NOT v_br_req;
        br_req <= v_br_req AFTER delay;         -- write it

        IF op_a(1) = '1' THEN                   -- do prefetch as well
```

```
          pf <= op_a;
          v_pf_req := NOT v_pf_req;            -- write it out
          pf_req <= v_pf_req AFTER delay;      -- and wait for both ACKs

          WAIT UNTIL pf_ack = v_pf_req AND br_ack = v_br_req;

        ELSE                                   -- no prefetch

          WAIT ON br_ack;                      -- just wait on BR_ACK

        END IF;


      ELSE                                     -- other instructions


        CASE index IS


        WHEN 16 =>                             -- bb0 n,ra,imm16

          s_relative := s_relative + 1;

          delay_to_use := delay_cond_rel;

          --- see if it's taken

          tmpi := v1d2int( opcode(25 DOWNTO 21) ); -- which bit
          taken := NOT vlb2boo( op_a(tmpi) );   -- is it set?

          --- compute word address

          addr := add2c( pc(31 DOWNTO 2), opcode( 15 DOWNTO 0) );


        WHEN 17 =>                             -- bb1 n,ra,imm16

          s_relative := s_relative + 1;

          delay_to_use := delay_cond_rel;

          --- see if it's taken

          tmpi := v1d2int( opcode(25 DOWNTO 21) ); -- which bit
          taken := vlb2boo( op_a(tmpi) );       -- is it set?

          --- compute word address

          addr := add2c( pc(31 DOWNTO 2), opcode( 15 DOWNTO 0) );


        WHEN 18 =>                             -- bxx ra,imm16
```

```
          s_relative := s_relative + 1;

          delay_to_use := delay_cond_rel;

          --- see if it's taken

          tmpi := v1d2int( op_a );             -- get ra

          IF ( ( (opcode(21) = '1') AND (tmpi > 0 ) ) OR
               ( (opcode(22) = '1') AND (tmpi = 0 ) ) OR
               ( (opcode(23) = '1') AND (tmpi < 0 ) ) )
          THEN
            taken := TRUE;
          ELSE
            taken := FALSE;
          END IF;

          --- compute word address

          addr := add2c( pc(31 DOWNTO 2), opcode( 15 DOWNTO 0) );


        WHEN 19 =>                             -- br imm26

          s_relative := s_relative + 1;

          delay_to_use := delay_rel;

          --- it's always taken

          taken := TRUE;

          --- compute word address

          addr := add2c( pc(31 DOWNTO 2), opcode( 25 DOWNTO 0) );


        WHEN 48 =>                             -- bb0 n,ra,rb

          s_indirect := s_indirect + 1;

          delay_to_use := delay_cond_abs;

          --- see if it's taken

          tmpi := v1d2int( opcode(25 DOWNTO 21) ); -- which bit
          taken := NOT vlb2boo( op_a(tmpi) );   -- is it set?

          --- compute word address

          addr := '0' & op_b( 31 DOWNTO 2 );


        WHEN 49 =>                             -- bb1 n,ra,rb
```

```
          s_indirect := s_indirect + 1;

          delay_to_use := delay_cond_abs;

          --- see if it's taken

          tmpi := v1d2int( opcode(25 DOWNTO 21) ); -- which bit
          taken := vlb2boo( op_a(tmpi) );       -- is it set?

          --- compute word address

          addr := '0' & op_b( 31 DOWNTO 2 );


        WHEN 50 =>                              -- bxx ra,rb

          s_indirect := s_indirect + 1;

          delay_to_use := delay_cond_abs;

          --- see if it's taken

          tmpi := v1d2int( op_a );              -- get ra

          IF ( ( (opcode(21) = '1') AND (tmpi > 0 ) ) OR
               ( (opcode(22) = '1') AND (tmpi = 0 ) ) OR
               ( (opcode(23) = '1') AND (tmpi < 0 ) ) )
          THEN
            taken := TRUE;
          ELSE
            taken := FALSE;
          END IF;

          --- compute word address

          addr := '0' & op_b( 31 DOWNTO 2 );


        WHEN 51 =>                              -- br rb

          s_indirect := s_indirect + 1;

          delay_to_use := delay_abs;

          --- it's always taken

          taken := TRUE;

          --- compute word address

          addr := '0' & op_b( 31 DOWNTO 2 );
```

```
WHEN OTHERS =>                          -- anything else
  putline("*** Illegal Opcode for Branch Unit ***");
END CASE;


--- write output on correct branch queue

tmpb := boo2vlb( taken );

tmpv := (addr(29 DOWNTO 0) & tmpb & '0'); -- compute branch target


IF taken THEN                           -- do prefetch if needed

  s_taken := s_taken + 1;

  pf <= tmpv;
  v_pf_req := NOT v_pf_req;
  pf_req <= v_pf_req AFTER delay_to_use; -- write it

END IF;


--- I had to modify the tag meaning slightly. Originally, all branch
--- instructions simply used the current value of cr0(Excp_Enable) to
--- determine which Branch queue to place the target in.  However,
--- branch instructions do not fault, so if an interrupt is noticed
--- after the branch has been issued but before it has completed, the
--- target goes down the wrong pipe.  Rather than add another bit to
--- the existing FIFOs, I just modified the tag value so that bit 7 is
--- used to indicate the queue in use when the instruction was issued.
--- This doesn't affect the memory unit since it always has to report
--- its status and the dispatch unit will wait for it to do so before
--- handling exceptions.  Only the branch unit uses the new tag bit for
--- anything.

IF tag(7) = '1' THEN                    -- use normal branch queue

  IF debug THEN
    tmpi := 4 * v1d2int(tmpv(31 DOWNTO 2));
    puthex("BR <= ",tmpi);
    putline("  ",tmpb);
  END IF;

  br <= tmpv;
  v_br_req := NOT v_br_req;
  br_req <= v_br_req AFTER delay_to_use;

  IF taken THEN                         -- must ACK both
    WAIT UNTIL pf_ack = v_pf_req AND br_ack = v_br_req;
  ELSE
    WAIT ON br_ack;                     -- just BR_ACK
  END IF;
```

```
        ELSE                                    -- use exception branch queue

          IF debug THEN
            tmpi := 4 * v1d2int(tmpv(31 DOWNTO 2));
            puthex("XBR <= ",tmpi);
            putline("  ",tmpb);
          END IF;

          xbr <= tmpv;
          v_xbr_req := NOT v_xbr_req;
          xbr_req <= v_xbr_req AFTER delay_to_use;

          IF taken THEN                         -- must ACK both
            WAIT UNTIL pf_ack = v_pf_req AND xbr_ack = v_xbr_req;
          ELSE
            WAIT ON xbr_ack;                    -- just XBR_ACK
          END IF;

        END IF;


      END IF;                                   -- not LDBR instruction


      --- signal completion of this instruction

      IF tellme = '1' THEN                      -- only if needed

        tmpv := int2v1d( STATUS_done );

        done(15 DOWNTO 0) <= tmpv(7 DOWNTO 0) & '0' & tag(6 DOWNTO 0);

        v_d_req := NOT v_d_req;
        d_req <= v_d_req AFTER delay;

        WAIT ON d_ack;

      END IF;

      ain <= rin AFTER delay;                   -- done with this event


    END IF;                                     -- clr not asserted


    WAIT ON clr, rin;                           -- wait for next event

  END PROCESS main;

END behavior;
```

2.4 control_unit.vhd

```
USE work.p_constants.ALL;
USE work.p_output.ALL;

---- The control_unit handles those few instructions which don't really fit
---- anywhere else.  Control register access is done here, as is read access
---- for the Branch Queue and PC value.


---- The EXTRA input was added to provide last-result-reuse and to elminate
---- zero values of op_a for immediate instructions. The bit usage is:
----
---- EXTRA(0) = reuseA
---- EXTRA(1) = reuseB
---- EXTRA(2) = reuseD
---- EXTRA(3) = zeroA (unused)

ENTITY control_unit IS
  GENERIC( delay: TIME := 100 ps );
  PORT( SIGNAL clr              : IN  VLBIT;
        SIGNAL rin              : IN  VLBIT;
        SIGNAL ain              : OUT VLBIT;
        SIGNAL tellme           : IN  VLBIT;
        SIGNAL tag              : IN  VLBIT_1D (7 DOWNTO 0);
        SIGNAL dest             : IN  VLBIT_1D (4 DOWNTO 0);
        SIGNAL pc               : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL opcode           : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL op_a             : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL op_b             : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL d_req            : OUT VLBIT;
        SIGNAL d_ack            : IN  VLBIT;
        SIGNAL done             : OUT VLBIT_1D (87 DOWNTO 0);
        SIGNAL rout             : OUT VLBIT;
        SIGNAL aout             : IN  VLBIT;
        SIGNAL res              : OUT VLBIT_1D (36 DOWNTO 0);
        SIGNAL rq_req           : OUT VLBIT;
        SIGNAL rq_ack           : IN  VLBIT;
        SIGNAL rq               : OUT VLBIT_1D (31 DOWNTO 0);
        SIGNAL putcr_req        : OUT VLBIT;
        SIGNAL putcr_ack        : IN  VLBIT;
        SIGNAL putcr            : OUT VLBIT_1D (31 DOWNTO 0);
        SIGNAL cr_num           : OUT VLBIT_1D (9 DOWNTO 0);
        SIGNAL getcr_req        : OUT VLBIT;
        SIGNAL getcr_ack        : IN  VLBIT;
        SIGNAL getcr            : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL extra            : IN  VLBIT_1D (3 DOWNTO 0) );
END control_unit;


ARCHITECTURE behavior OF control_unit IS
BEGIN
  main : PROCESS
  VARIABLE debug              : BOOLEAN := FALSE;


  --- factors affecting response time

  VARIABLE delay_mvpc         : TIME := 100 ps; -- this is the only hard one


  --- output variables

  VARIABLE v_rout             : VLBIT;
  VARIABLE v_rq_req           : VLBIT;
  VARIABLE v_d_req            : VLBIT;
  VARIABLE v_putcr_req        : VLBIT;
  VARIABLE v_getcr_req        : VLBIT;


  --- internal variables

  VARIABLE index              : INTEGER;
  VARIABLE op_aa              : VLBIT_1D(31 DOWNTO 0);
  VARIABLE op_bb              : VLBIT_1D(31 DOWNTO 0);
  VARIABLE lastresult         : VLBIT_1D(31 DOWNTO 0);

  --- scratch variables

  VARIABLE tmpi               : INTEGER;
  VARIABLE tmpv               : VLBIT_1D (31 DOWNTO 0);


  --- statistics

  VARIABLE s_mvpc             : INTEGER;
  VARIABLE s_getcr            : INTEGER;
  VARIABLE s_putcr            : INTEGER;
  VARIABLE s_mvbr             : INTEGER;


BEGIN


  --- If clear is asserted, reset everything and stick here.

  IF clr /= '1' THEN                        -- reset to beginning

    v_rout := '0';
    v_rq_req := '0';
    v_d_req := '0';
    v_putcr_req := '0';
    v_getcr_req := '0';

    ain <= '0';
    d_req <= v_d_req;
    done <= ZERO(87 DOWNTO 0);
    rout <= v_rout;
    res <= ZERO(36 DOWNTO 0);
    rq_req <= v_rq_req;
    rq <= ZERO(31 DOWNTO 0);

    putcr_req <= v_putcr_req;
    putcr <= ZERO(31 DOWNTO 0);
    cr_num <= ZERO(9 DOWNTO 0);
    getcr_req <= v_getcr_req;

    s_mvpc := 0;
    s_putcr := 0;
    s_getcr := 0;
    s_mvbr := 0;

    WAIT UNTIL clr = '1';


  ELSE                                        -- only wakeup event is RIN

    --- This stuff had to be encoded by hand. I could have written a Perl
    --- script to do it, but it would have taken longer than doing it
    --- manually. This stuff shouldn't change that much (I hope). I also made
    --- some assumptions about unused bits, which is probably okay.


    IF extra(0)='1' THEN                      -- reuse op_a result
      op_aa := lastresult;
    ELSE
      op_aa := op_a;
    END IF;

    IF extra(1)='1' THEN                      -- reuse op_b
      op_bb := lastresult;
    ELSE
      op_bb := op_b;
    END IF;

    IF debug THEN
      tmpi := v1d2int(op_aa);
      puthexline("op_a is ",tmpi);
      tmpi := v1d2int(op_bb);
      puthexline("op_b is ",tmpi);
    END IF;


    index := op2index( opcode );

    CASE index IS


    WHEN 20 =>                                -- mvpc rd,imm16
```

```
        IF dest /= REG_R0 THEN -- if worth doing

          s_mvpc := s_mvpc + 1;

          --- compute word address

          tmpv(30 DOWNTO 0) := add2c( pc(31 DOWNTO 2), opcode( 15 DOWNTO 0) );

          IF dest = REG_R1 THEN

            rq <= tmpv(29 DOWNTO 0) & ('0','0');

            v_rq_req := NOT v_rq_req;           -- write it out
            rq_req <= v_rq_req AFTER delay_mvpc;

            WAIT ON rq_ack;

          ELSE

            res <= dest & tmpv(29 DOWNTO 0) & ('0','0');

            lastresult := tmpv(29 DOWNTO 0) & ('0','0');

            IF debug THEN
              tmpi := v1d2int(lastresult);
              puthexline("latched ",tmpi);
            END IF;

            v_rout := NOT v_rout;               -- write it out
            rout <= v_rout AFTER delay_mvpc;

            WAIT ON aout;

          END IF;


        END IF;                                 -- done with it


      WHEN 92 | 93 =>                           -- getcr rd,cr

        IF dest /= REG_R0 THEN                  -- if worth doing

          s_getcr := s_getcr + 1;

          IF index = 92 THEN                    -- direct cr number
            cr_num <= opcode(9 DOWNTO 0);
          ELSE
            cr_num <= op_bb(9 DOWNTO 0);        -- indirect control register
          END IF;
          v_getcr_req := NOT v_getcr_req;
          getcr_req <= v_getcr_req AFTER delay;

          WAIT ON getcr_ack;
```

```
    --- NOTE: This would normally need to be four-phase, but since this
    --- is the only unit which can read the control registers, it should
    --- work fine.

    --- write it out

    IF dest = REG_R1 THEN

      rq <= getcr;

      v_rq_req := NOT v_rq_req;           -- write it out
      rq_req <= v_rq_req AFTER delay;

      WAIT ON rq_ack;

    ELSE

      res <= dest & getcr;

      lastresult := getcr;

      IF debug THEN
        tmpi := v1d2int(lastresult);
        puthexline("latched ",tmpi);
      END IF;

      v_rout := NOT v_rout;                       -- write it out
      rout <= v_rout AFTER delay;

      WAIT ON aout;

    END IF;

  END IF;                                 -- done with it


WHEN 94 | 95 =>                           -- putcr cr,ra

  s_putcr := s_putcr + 1;

  putcr <= op_aa;                         -- write out data
  IF index = 94 THEN                      -- direct cr number
    cr_num <= opcode(9 DOWNTO 0);
  ELSE
    cr_num <= op_bb(9 DOWNTO 0);          -- indirect control register
  END IF;
  v_putcr_req := NOT v_putcr_req;
  putcr_req <= v_putcr_req AFTER delay;

  WAIT ON putcr_ack;


WHEN 86 =>                                -- mvbr rd
```

```
        IF dest /= REG_R0 THEN -- if worth doing

          s_mvbr := s_mvbr + 1;

          IF dest = REG_R1 THEN

            rq <= pc;

            v_rq_req := NOT v_rq_req;           -- write it out
            rq_req <= v_rq_req AFTER delay;

            WAIT ON rq_ack;

          ELSE

            res <= dest & pc;

            lastresult := pc;

            IF debug THEN
              tmpi := v1d2int(lastresult);
              puthexline("latched ",tmpi);
            END IF;

            v_rout := NOT v_rout;               -- write it out
            rout <= v_rout AFTER delay;

            WAIT ON aout;

          END IF;

        END IF;                                 -- done with it


      WHEN OTHERS =>                            -- anything else
        putline("*** Illegal Opcode for Control Unit ***");
      END CASE;



      --- signal completion of this instruction

      IF tellme = '1' THEN                      -- only if needed

        tmpv := int2v1d( STATUS_done );
        done <= ZERO(87 DOWNTO 16) & tmpv(7 DOWNTO 0) & tag;

        v_d_req := NOT v_d_req;
        d_req <= v_d_req AFTER delay;

        WAIT ON d_ack;

      END IF;
      ain <= rin AFTER delay;                   -- done with this event


    END IF;                                     -- clr not asserted


    WAIT ON clr, rin;                           -- wait for next event

  END PROCESS main;

END behavior;
```

2.5 cpx128.vhd

```
USE work.p_constants.ALL;

---- This is used to capture the data for a large FIFO.  This is just a ViewSim
---- performance hack.  Passing the data through several latches uses an
---- incredible number of signals and nets, so I use this VHDL program to store
---- the data, and I use an external delay chain to model the time behavior.
---- While this models the C/CD P/PD interface, the CD and PD pins are absent.
---- This unit takes zero time.  External delays must be used to model the done
---- responses.

ENTITY cpx128 IS
  PORT (SIGNAL clr      :IN    VLBIT;
        SIGNAL c        :IN    VLBIT;
        SIGNAL p        :IN    VLBIT;
        SIGNAL i        :IN    VLBIT_1D(127 DOWNTO 0);
        SIGNAL o        :OUT   VLBIT_1D(127 DOWNTO 0));
END cpx128;

ARCHITECTURE behavior OF cpx128 IS
BEGIN
  main : PROCESS

    CONSTANT MaxDepth   : INTEGER := 16;

    TYPE pipe_type IS ARRAY( 0 TO MaxDepth ) OF VLBIT_1D(127 DOWNTO 0);

    VARIABLE pipe    : pipe_type;

    VARIABLE head    : INTEGER;
    VARIABLE tail    : INTEGER;
    VARIABLE count   : INTEGER;

    VARIABLE v_cd    : VLBIT;                   -- only for error checking
    VARIABLE v_pd    : VLBIT;                   -- not for output

    VARIABLE s_maxusage : INTEGER;              -- for statistics
    VARIABLE s_total_count : INTEGER;
    VARIABLE s_total_times : INTEGER;


  BEGIN

    IF (clr /= '1') THEN

      s_maxusage := 0;
      s_total_count := 0;
      s_total_times := 0;

      head := 0;
      tail := 0;
      count := 0;
      v_cd := '0';
      v_pd := '0';

      o <= ZERO(127 DOWNTO 0);

      WAIT UNTIL clr = '1';


    ELSE

      IF c'EVENT THEN                           -- data coming in

        IF c = v_cd THEN
          putline("*** Handshake violation on Capture ***");
        END IF;

        IF ( count < MaxDepth ) THEN            -- latch incoming data

          pipe(tail) := i;

          tail := tail + 1;

          IF tail >= MaxDepth THEN
            tail := 0;
          END IF;

          count := count + 1;
          s_total_count := s_total_count + count;
          s_total_times := s_total_times + 1;

          IF count > s_maxusage THEN            -- keep track of maxium
            s_maxusage := count;
          END IF;

          v_cd := NOT v_cd;                     -- remember this event

        END IF;

      END IF;


      IF p'EVENT THEN                           -- want data on output

        IF count = 0 OR p = v_pd THEN
          putline("*** Handshake violation on Pass ***");
        END IF;

        head := head + 1;                       -- so I can get rid of it

        IF head >= MaxDepth THEN
          head := 0;
        END IF;

        count := count - 1;
```

```
        v_pd := NOT v_pd;                       -- remember this event

      END IF;


      IF count = 0 THEN                         -- copy through if nothing
        pipe(head) := i;
      END IF;

      o <= pipe(head);                          -- show latest data


    END IF;                                     -- clr not asserted

    WAIT ON clr, c, p;

  END PROCESS main;
END behavior;
```

2.6 cpx18.vhd

```
USE work.p_constants.ALL;

---- This is used to capture the data for a large FIFO.  This is just a ViewSim
---- performance hack.  Passing the data through several latches uses an
---- incredible number of signals and nets, so I use this VHDL program to store
---- the data, and I use an external delay chain to model the time behavior.
---- While this models the C/CD P/PD interface, the CD and PD pins are absent.
---- This unit takes zero time.  External delays must be used to model the done
---- responses.

ENTITY cpx18 IS
  PORT (SIGNAL clr      :IN    VLBIT;
        SIGNAL c        :IN    VLBIT;
        SIGNAL p        :IN    VLBIT;
        SIGNAL i        :IN    VLBIT_1D(17 DOWNTO 0);
        SIGNAL o        :OUT   VLBIT_1D(17 DOWNTO 0));
END cpx18;

ARCHITECTURE behavior OF cpx18 IS
BEGIN
  main : PROCESS

    CONSTANT MaxDepth   : INTEGER := 16;

    TYPE pipe_type IS ARRAY( 0 TO MaxDepth ) OF VLBIT_1D(17 DOWNTO 0);

    VARIABLE pipe    : pipe_type;

    VARIABLE head    : INTEGER;
    VARIABLE tail    : INTEGER;
    VARIABLE count   : INTEGER;

    VARIABLE v_cd    : VLBIT;                   -- only for error checking
    VARIABLE v_pd    : VLBIT;                   -- not for output

    VARIABLE s_maxusage : INTEGER;              -- for statistics
    VARIABLE s_total_count : INTEGER;
    VARIABLE s_total_times : INTEGER;


  BEGIN

    IF (clr /= '1') THEN

      s_maxusage := 0;
      s_total_count := 0;
      s_total_times := 0;

      head := 0;
      tail := 0;
      count := 0;
      v_cd := '0';
      v_pd := '0';

      o <= ZERO(17 DOWNTO 0);

      WAIT UNTIL clr = '1';


    ELSE

      IF c'EVENT THEN                           -- data coming in

        IF c = v_cd THEN
          putline("*** Handshake violation on Capture ***");
        END IF;

        IF ( count < MaxDepth ) THEN            -- latch incoming data

          pipe(tail) := i;

          tail := tail + 1;

          IF tail >= MaxDepth THEN
            tail := 0;
          END IF;

          count := count + 1;
          s_total_count := s_total_count + count;
          s_total_times := s_total_times + 1;

          IF count > s_maxusage THEN            -- keep track of maxium
            s_maxusage := count;
          END IF;

          v_cd := NOT v_cd;                     -- remember this event

        END IF;

      END IF;


      IF p'EVENT THEN                           -- want data on output

        IF count = 0 OR p = v_pd THEN
          putline("*** Handshake violation on Pass ***");
        END IF;

        head := head + 1;                       -- so I can get rid of it

        IF head >= MaxDepth THEN
          head := 0;
        END IF;

        count := count - 1;
        v_pd := NOT v_pd;                     -- remember this event

      END IF;


      IF count = 0 THEN                       -- copy through if nothing
        pipe(head) := i;
      END IF;

      o <= pipe(head);                        -- show latest data


    END IF;                                   -- clr not asserted

    WAIT ON clr, c, p;

  END PROCESS main;
END behavior;
```

2.7 cpx32.vhd

```
USE work.p_constants.ALL;

---- This is used to capture the data for a large FIFO.  This is just a ViewSim
---- performance hack.  Passing the data through several latches uses an
---- incredible number of signals and nets, so I use this VHDL program to store
---- the data, and I use an external delay chain to model the time behavior.
---- While this models the C/CD P/PD interface, the CD and PD pins are absent.
---- This unit takes zero time.  External delays must be used to model the done
---- responses.

ENTITY cpx32 IS
  PORT (SIGNAL clr      :IN    VLBIT;
        SIGNAL c        :IN    VLBIT;
        SIGNAL p        :IN    VLBIT;
        SIGNAL i        :IN    VLBIT_1D(31 DOWNTO 0);
        SIGNAL o        :OUT   VLBIT_1D(31 DOWNTO 0));
END cpx32;

ARCHITECTURE behavior OF cpx32 IS
BEGIN
  main : PROCESS

    CONSTANT MaxDepth   : INTEGER := 16;

    TYPE pipe_type IS ARRAY( 0 TO MaxDepth ) OF VLBIT_1D(31 DOWNTO 0);

    VARIABLE pipe    : pipe_type;

    VARIABLE head    : INTEGER;
    VARIABLE tail    : INTEGER;
    VARIABLE count   : INTEGER;

    VARIABLE v_cd    : VLBIT;                   -- only for error checking
    VARIABLE v_pd    : VLBIT;                   -- not for output

    VARIABLE s_maxusage : INTEGER;              -- for statistics
    VARIABLE s_total_count : INTEGER;
    VARIABLE s_total_times : INTEGER;


  BEGIN

    IF (clr /= '1') THEN

      s_maxusage := 0;
      s_total_count := 0;
      s_total_times := 0;

      head := 0;
      tail := 0;
      count := 0;
      v_cd := '0';
      v_pd := '0';

      o <= ZERO(31 DOWNTO 0);

      WAIT UNTIL clr = '1';


    ELSE

      IF c'EVENT THEN                          -- data coming in

        IF c = v_cd THEN
          putline("*** Handshake violation on Capture ***");
        END IF;

        IF ( count < MaxDepth ) THEN           -- latch incoming data

          pipe(tail) := i;

          tail := tail + 1;

          IF tail >= MaxDepth THEN
            tail := 0;
          END IF;

          count := count + 1;
          s_total_count := s_total_count + count;
          s_total_times := s_total_times + 1;

          IF count > s_maxusage THEN           -- keep track of maxium
            s_maxusage := count;
          END IF;

          v_cd := NOT v_cd;                    -- remember this event

        END IF;

      END IF;


      IF p'EVENT THEN                          -- want data on output

        IF count = 0 OR p = v_pd THEN
          putline("*** Handshake violation on Pass ***");
        END IF;

        head := head + 1;                      -- so I can get rid of it

        IF head >= MaxDepth THEN
          head := 0;
        END IF;

        count := count - 1;
        v_pd := NOT v_pd;                           -- remember this event

      END IF;


      IF count = 0 THEN                             -- copy through if nothing
        pipe(head) := i;
      END IF;

      o <= pipe(head);                              -- show latest data


    END IF;                                         -- clr not asserted

    WAIT ON clr, c, p;

  END PROCESS main;
END behavior;
```

2.8 cpx3op.vhd

```
USE work.p_constants.ALL;

---- This is used to capture the data for a large FIFO.  This is just a ViewSim
---- performance hack.  Passing the data through several latches uses an
---- incredible number of signals and nets, so I use this VHDL program to store
---- the data, and I use an external delay chain to model the time behavior.
---- While this models the C/CD P/PD interface, the CD and PD pins are absent.
---- This unit takes zero time.  External delays must be used to model the done
---- responses.

ENTITY cpx3op IS
  PORT (SIGNAL clr      :IN    VLBIT;
        SIGNAL c        :IN    VLBIT;
        SIGNAL p        :IN    VLBIT;
        SIGNAL i        :IN    VLBIT_1D(113 DOWNTO 0);
        SIGNAL o        :OUT   VLBIT_1D(113 DOWNTO 0));
END cpx3op;

ARCHITECTURE behavior OF cpx3op IS
BEGIN
  main : PROCESS

    CONSTANT MaxDepth   : INTEGER := 16;

    TYPE pipe_type IS ARRAY( 0 TO MaxDepth ) OF VLBIT_1D(113 DOWNTO 0);

    VARIABLE pipe    : pipe_type;

    VARIABLE head    : INTEGER;
    VARIABLE tail    : INTEGER;
    VARIABLE count   : INTEGER;

    VARIABLE v_cd    : VLBIT;                   -- only for error checking
    VARIABLE v_pd    : VLBIT;                   -- not for output

    VARIABLE s_maxusage : INTEGER;              -- for statistics
    VARIABLE s_total_count : INTEGER;
    VARIABLE s_total_times : INTEGER;


  BEGIN

    IF (clr /= '1') THEN

      s_maxusage := 0;
      s_total_count := 0;
      s_total_times := 0;

      head := 0;
      tail := 0;
      count := 0;
      v_cd := '0';
      v_pd := '0';

      o <= ZERO(113 DOWNTO 0);

      WAIT UNTIL clr = '1';


    ELSE

      IF c'EVENT THEN                           -- data coming in

        IF c = v_cd THEN
          putline("*** Handshake violation on Capture ***");
        END IF;

        IF ( count < MaxDepth ) THEN            -- latch incoming data

          pipe(tail) := i;

          tail := tail + 1;

          IF tail >= MaxDepth THEN
            tail := 0;
          END IF;

          count := count + 1;
          s_total_count := s_total_count + count;
          s_total_times := s_total_times + 1;

          IF count > s_maxusage THEN            -- keep track of maxium
            s_maxusage := count;
          END IF;

          v_cd := NOT v_cd;                     -- remember this event

        END IF;

      END IF;


      IF p'EVENT THEN                           -- want data on output

        IF count = 0 OR p = v_pd THEN
          putline("*** Handshake violation on Pass ***");
        END IF;

        head := head + 1;                       -- so I can get rid of it

        IF head >= MaxDepth THEN
          head := 0;
        END IF;

        count := count - 1;
        v_pd := NOT v_pd;                          -- remember this event

      END IF;


      IF count = 0 THEN                            -- copy through if nothing
        pipe(head) := i;
      END IF;

      o <= pipe(head);                             -- show latest data


    END IF;                                        -- clr not asserted

    WAIT ON clr, c, p;

  END PROCESS main;
END behavior;
```

2.9 cpx4op.vhd

```
USE work.p_constants.ALL;

---- This is used to capture the data for a large FIFO.  This is just a ViewSim
---- performance hack.  Passing the data through several latches uses an
---- incredible number of signals and nets, so I use this VHDL program to store
---- the data, and I use an external delay chain to model the time behavior.
---- While this models the C/CD P/PD interface, the CD and PD pins are absent.
---- This unit takes zero time.  External delays must be used to model the done
---- responses.

ENTITY cpx4op IS
  PORT (SIGNAL clr      :IN    VLBIT;
        SIGNAL c        :IN    VLBIT;
        SIGNAL p        :IN    VLBIT;
        SIGNAL i        :IN    VLBIT_1D(145 DOWNTO 0);
        SIGNAL o        :OUT   VLBIT_1D(145 DOWNTO 0));
END cpx4op;

ARCHITECTURE behavior OF cpx4op IS
BEGIN
  main : PROCESS

    CONSTANT MaxDepth   : INTEGER := 16;

    TYPE pipe_type IS ARRAY( 0 TO MaxDepth ) OF VLBIT_1D(145 DOWNTO 0);

    VARIABLE pipe    : pipe_type;

    VARIABLE head    : INTEGER;
    VARIABLE tail    : INTEGER;
    VARIABLE count   : INTEGER;

    VARIABLE v_cd    : VLBIT;                   -- only for error checking
    VARIABLE v_pd    : VLBIT;                   -- not for output

    VARIABLE s_maxusage : INTEGER;              -- for statistics
    VARIABLE s_total_count : INTEGER;
    VARIABLE s_total_times : INTEGER;


  BEGIN

    IF (clr /= '1') THEN

      s_maxusage := 0;
      s_total_count := 0;
      s_total_times := 0;

      head := 0;
      tail := 0;
      count := 0;
      v_cd := '0';
      v_pd := '0';

      o <= ZERO(145 DOWNTO 0);

      WAIT UNTIL clr = '1';


    ELSE

      IF c'EVENT THEN                            -- data coming in

        IF c = v_cd THEN
          putline("*** Handshake violation on Capture ***");
        END IF;

        IF ( count < MaxDepth ) THEN            -- latch incoming data

          pipe(tail) := i;

          tail := tail + 1;

          IF tail >= MaxDepth THEN
            tail := 0;
          END IF;

          count := count + 1;
          s_total_count := s_total_count + count;
          s_total_times := s_total_times + 1;

          IF count > s_maxusage THEN            -- keep track of maxium
            s_maxusage := count;
          END IF;

          v_cd := NOT v_cd;                     -- remember this event

        END IF;

      END IF;


      IF p'EVENT THEN                            -- want data on output

        IF count = 0 OR p = v_pd THEN
          putline("*** Handshake violation on Pass ***");
        END IF;

        head := head + 1;                       -- so I can get rid of it

        IF head >= MaxDepth THEN
          head := 0;
        END IF;

        count := count - 1;
```

```
        v_pd := NOT v_pd;                       -- remember this event

      END IF;


      IF count = 0 THEN                         -- copy through if nothing
        pipe(head) := i;
      END IF;

      o <= pipe(head);                          -- show latest data


    END IF;                                     -- clr not asserted

    WAIT ON clr, c, p;

  END PROCESS main;
END behavior;
```

2.10 cpx5.vhd

```
USE work.p_constants.ALL;

---- This is used to capture the data for a large FIFO.  This is just a ViewSim
---- performance hack.  Passing the data through several latches uses an
---- incredible number of signals and nets, so I use this VHDL program to store
---- the data, and I use an external delay chain to model the time behavior.
---- While this models the C/CD P/PD interface, the CD and PD pins are absent.
---- This unit takes zero time.  External delays must be used to model the done
---- responses.

ENTITY cpx5 IS
  PORT (SIGNAL clr      :IN    VLBIT;
        SIGNAL c        :IN    VLBIT;
        SIGNAL p        :IN    VLBIT;
        SIGNAL i        :IN    VLBIT_1D(4 DOWNTO 0);
        SIGNAL o        :OUT   VLBIT_1D(4 DOWNTO 0));
END cpx5;

ARCHITECTURE behavior OF cpx5 IS
BEGIN
  main : PROCESS

    CONSTANT MaxDepth   : INTEGER := 16;

    TYPE pipe_type IS ARRAY( 0 TO MaxDepth ) OF VLBIT_1D(4 DOWNTO 0);

    VARIABLE pipe    : pipe_type;

    VARIABLE head    : INTEGER;
    VARIABLE tail    : INTEGER;
    VARIABLE count   : INTEGER;

    VARIABLE v_cd    : VLBIT;                   -- only for error checking
    VARIABLE v_pd    : VLBIT;                   -- not for output

    VARIABLE s_maxusage : INTEGER;              -- for statistics
    VARIABLE s_total_count : INTEGER;
    VARIABLE s_total_times : INTEGER;


  BEGIN

    IF (clr /= '1') THEN

      s_maxusage := 0;
      s_total_count := 0;
      s_total_times := 0;

      head := 0;
      tail := 0;
      count := 0;
      v_cd := '0';
      v_pd := '0';

      o <= ZERO(4 DOWNTO 0);

      WAIT UNTIL clr = '1';


    ELSE

      IF c'EVENT THEN                          -- data coming in

        IF c = v_cd THEN
          putline("*** Handshake violation on Capture ***");
        END IF;

        IF ( count < MaxDepth ) THEN           -- latch incoming data

          pipe(tail) := i;

          tail := tail + 1;

          IF tail >= MaxDepth THEN
            tail := 0;
          END IF;

          count := count + 1;
          s_total_count := s_total_count + count;
          s_total_times := s_total_times + 1;

          IF count > s_maxusage THEN           -- keep track of maxium
            s_maxusage := count;
          END IF;

          v_cd := NOT v_cd;                    -- remember this event

        END IF;

      END IF;


      IF p'EVENT THEN                          -- want data on output

        IF count = 0 OR p = v_pd THEN
          putline("*** Handshake violation on Pass ***");
        END IF;

        head := head + 1;                      -- so I can get rid of it

        IF head >= MaxDepth THEN
          head := 0;
        END IF;

        count := count - 1;
        v_pd := NOT v_pd;                       -- remember this event

      END IF;


      IF count = 0 THEN                         -- copy through if nothing
        pipe(head) := i;
      END IF;

      o <= pipe(head);                          -- show latest data


    END IF;                                     -- clr not asserted

    WAIT ON clr, c, p;

  END PROCESS main;
END behavior;
```

2.11 cpx88.vhd

```
USE work.p_constants.ALL;

---- This is used to capture the data for a large FIFO.  This is just a ViewSim
---- performance hack.  Passing the data through several latches uses an
---- incredible number of signals and nets, so I use this VHDL program to store
---- the data, and I use an external delay chain to model the time behavior.
---- While this models the C/CD P/PD interface, the CD and PD pins are absent.
---- This unit takes zero time.  External delays must be used to model the done
---- responses.

ENTITY cpx88 IS
  PORT (SIGNAL clr      :IN    VLBIT;
        SIGNAL c        :IN    VLBIT;
        SIGNAL p        :IN    VLBIT;
        SIGNAL i        :IN    VLBIT_1D(87 DOWNTO 0);
        SIGNAL o        :OUT   VLBIT_1D(87 DOWNTO 0));
END cpx88;

ARCHITECTURE behavior OF cpx88 IS
BEGIN
  main : PROCESS

    CONSTANT MaxDepth   : INTEGER := 16;

    TYPE pipe_type IS ARRAY( 0 TO MaxDepth ) OF VLBIT_1D(87 DOWNTO 0);

    VARIABLE pipe    : pipe_type;

    VARIABLE head    : INTEGER;
    VARIABLE tail    : INTEGER;
    VARIABLE count   : INTEGER;

    VARIABLE v_cd    : VLBIT;                   -- only for error checking
    VARIABLE v_pd    : VLBIT;                   -- not for output

    VARIABLE s_maxusage : INTEGER;              -- for statistics
    VARIABLE s_total_count : INTEGER;
    VARIABLE s_total_times : INTEGER;


  BEGIN

    IF (clr /= '1') THEN

      s_maxusage := 0;
      s_total_count := 0;
      s_total_times := 0;

      head := 0;
      tail := 0;
      count := 0;
      v_cd := '0';
      v_pd := '0';

      o <= ZERO(87 DOWNTO 0);

      WAIT UNTIL clr = '1';


    ELSE

      IF c'EVENT THEN                            -- data coming in

        IF c = v_cd THEN
          putline("*** Handshake violation on Capture ***");
        END IF;

        IF ( count < MaxDepth ) THEN             -- latch incoming data

          pipe(tail) := i;

          tail := tail + 1;

          IF tail >= MaxDepth THEN
            tail := 0;
          END IF;

          count := count + 1;
          s_total_count := s_total_count + count;
          s_total_times := s_total_times + 1;

          IF count > s_maxusage THEN             -- keep track of maxium
            s_maxusage := count;
          END IF;

          v_cd := NOT v_cd;                      -- remember this event

        END IF;

      END IF;


      IF p'EVENT THEN                            -- want data on output

        IF count = 0 OR p = v_pd THEN
          putline("*** Handshake violation on Pass ***");
        END IF;

        head := head + 1;                        -- so I can get rid of it

        IF head >= MaxDepth THEN
          head := 0;
        END IF;

        count := count - 1;
        v_pd := NOT v_pd;                      -- remember this event

      END IF;


      IF count = 0 THEN                        -- copy through if nothing
        pipe(head) := i;
      END IF;

      o <= pipe(head);                         -- show latest data


    END IF;                                    -- clr not asserted

    WAIT ON clr, c, p;

  END PROCESS main;
END behavior;
```

2.12 cpx96.vhd

```
USE work.p_constants.ALL;

---- This is used to capture the data for a large FIFO.  This is just a ViewSim
---- performance hack.  Passing the data through several latches uses an
---- incredible number of signals and nets, so I use this VHDL program to store
---- the data, and I use an external delay chain to model the time behavior.
---- While this models the C/CD P/PD interface, the CD and PD pins are absent.
---- This unit takes zero time.  External delays must be used to model the done
---- responses.

ENTITY cpx96 IS
  PORT (SIGNAL clr      :IN    VLBIT;
        SIGNAL c        :IN    VLBIT;
        SIGNAL p        :IN    VLBIT;
        SIGNAL i        :IN    VLBIT_1D(95 DOWNTO 0);
        SIGNAL o        :OUT   VLBIT_1D(95 DOWNTO 0));
END cpx96;

ARCHITECTURE behavior OF cpx96 IS
BEGIN
  main : PROCESS

    CONSTANT MaxDepth   : INTEGER := 16;

    TYPE pipe_type IS ARRAY( 0 TO MaxDepth ) OF VLBIT_1D(95 DOWNTO 0);

    VARIABLE pipe    : pipe_type;

    VARIABLE head    : INTEGER;
    VARIABLE tail    : INTEGER;
    VARIABLE count   : INTEGER;

    VARIABLE v_cd    : VLBIT;                    -- only for error checking
    VARIABLE v_pd    : VLBIT;                    -- not for output

    VARIABLE s_maxusage : INTEGER;               -- for statistics
    VARIABLE s_total_count : INTEGER;
    VARIABLE s_total_times : INTEGER;


  BEGIN

    IF (clr /= '1') THEN

      s_maxusage := 0;
      s_total_count := 0;
      s_total_times := 0;

      head := 0;
      tail := 0;
      count := 0;
      v_cd := '0';
      v_pd := '0';

      o <= ZERO(95 DOWNTO 0);

      WAIT UNTIL clr = '1';


    ELSE

      IF c'EVENT THEN                          -- data coming in

        IF c = v_cd THEN
          putline("*** Handshake violation on Capture ***");
        END IF;

        IF ( count < MaxDepth ) THEN           -- latch incoming data

          pipe(tail) := i;

          tail := tail + 1;

          IF tail >= MaxDepth THEN
            tail := 0;
          END IF;

          count := count + 1;
          s_total_count := s_total_count + count;
          s_total_times := s_total_times + 1;

          IF count > s_maxusage THEN           -- keep track of maxium
            s_maxusage := count;
          END IF;

          v_cd := NOT v_cd;                    -- remember this event

        END IF;

      END IF;


      IF p'EVENT THEN                          -- want data on output

        IF count = 0 OR p = v_pd THEN
          putline("*** Handshake violation on Pass ***");
        END IF;

        head := head + 1;                      -- so I can get rid of it

        IF head >= MaxDepth THEN
          head := 0;
        END IF;

        count := count - 1;
        v_pd := NOT v_pd;                          -- remember this event

      END IF;


      IF count = 0 THEN                            -- copy through if nothing
        pipe(head) := i;
      END IF;

      o <= pipe(head);                             -- show latest data


    END IF;                                        -- clr not asserted

    WAIT ON clr, c, p;

  END PROCESS main;
END behavior;
```

2.13 delaytap.vhd

```
USE work.p_constants.ALL;

---- This taps into a 16-element FIFO at a given point, so that the effective
---- length of the FIFO can be adjusted dynamically.  It just passes the REQ
---- and ACK signals, without touching the data. It is strictly a performance
---- hack to avoid altering the FIFO circuit just to change its length.  It
---- does not check or enforce correct handshaking.

ENTITY delaytap IS
  PORT ( SIGNAL clr       :IN    VLBIT;
         SIGNAL rin       :IN    VLBIT;
         SIGNAL ain       :OUT   VLBIT;
         SIGNAL rx        :OUT   VLBIT_1D(15 DOWNTO 0);
         SIGNAL ax        :IN    VLBIT_1D(15 DOWNTO 0) );
END delaytap;

ARCHITECTURE behavior OF delaytap IS
BEGIN
  main : PROCESS

    VARIABLE tap     : INTEGER := 8;

    VARIABLE v_rx    : VLBIT_1D(15 DOWNTO 0);
    VARIABLE v_ax    : VLBIT_1D(15 DOWNTO 0);

  BEGIN

    IF (clr /= '1') THEN

      ain <= '0';
      v_rx := ZERO(15 DOWNTO 0);
      v_ax := ZERO(15 DOWNTO 0);
      rx <= v_rx;

      WAIT UNTIL clr = '1';

    ELSE

      IF rin'EVENT THEN                          -- request new data

        v_rx(tap) := NOT v_rx(tap);              -- toggle output request
        rx <= v_rx;

      ELSE                                       -- else must be AX event

        IF ax(tap) /= v_ax(tap) THEN             -- wait for just mine

          v_ax := ax;                            -- remember vector

          ain <= rin;                            -- ack it

        END IF;                                  -- that's all
      END IF;

    END IF;                                    -- clr not asserted

    WAIT ON clr, rin, ax;

  END PROCESS main;
END behavior;
```

2.14 dispatch.vhd

```
USE work.p_constants.ALL;
USE work.p_dispatch.ALL;
USE work.p_output.ALL;

---- The dispatch module is the heart of the IF Unit.  The IF Unit is what
---- the rest of the processor connects to, but the dispatch module has
---- slightly different interfaces and behavior.


ENTITY dispatch IS
  GENERIC( delay      : TIME := 100 ps;         -- handshaking
           start_addr : INTEGER := 0;           -- initial PC value
           init_cr0   : INTEGER := 112 );       -- Exceptions, Ints, Supervisor
  PORT( SIGNAL clr              : IN  VLBIT;
        SIGNAL sb               : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL sb_num           : OUT VLBIT_1D (4 DOWNTO 0);
        SIGNAL sb_r_d           : OUT VLBIT;
        SIGNAL sb_r_g           : IN  VLBIT;
        SIGNAL sb_r_r           : OUT VLBIT;
        SIGNAL sb_set           : OUT VLBIT;
        SIGNAL sb_w_r           : OUT VLBIT;
        SIGNAL sb_w_a           : IN  VLBIT;
        SIGNAL br               : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL br_ack           : OUT VLBIT;
        SIGNAL br_p             : IN  VLBIT;
        SIGNAL br_p_ack         : IN  VLBIT;
        SIGNAL br_p_req         : OUT VLBIT;
        SIGNAL cr_num           : IN  VLBIT_1D (9 DOWNTO 0);
        SIGNAL done             : IN  VLBIT_1D (87 DOWNTO 0);
        SIGNAL done_ack         : OUT VLBIT;
        SIGNAL done_req         : IN  VLBIT;
        SIGNAL ex               : OUT VLBIT_1D (127 DOWNTO 0);
        SIGNAL ex_p             : IN  VLBIT;
        SIGNAL ex_p_ack         : IN  VLBIT;
        SIGNAL ex_p_req         : OUT VLBIT;
        SIGNAL ex_req           : OUT VLBIT;
        SIGNAL external_int     : IN  VLBIT;
        SIGNAL fault_pending    : OUT VLBIT;
        SIGNAL getcr            : OUT VLBIT_1D (31 DOWNTO 0);
        SIGNAL getcr_ack        : OUT VLBIT;
        SIGNAL getcr_req        : IN  VLBIT;
        SIGNAL i_addr           : OUT VLBIT_1D (31 DOWNTO 0);
        SIGNAL i_data           : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL i_status         : IN  VLBIT_1D (3 DOWNTO 0);
        SIGNAL inst_ack         : IN  VLBIT;
        SIGNAL inst_ack_ack     : OUT VLBIT;
        SIGNAL inst_req         : OUT VLBIT;
        SIGNAL putcr            : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL putcr_ack        : OUT VLBIT;
        SIGNAL putcr_req        : IN  VLBIT;
        SIGNAL reply_hazy       : OUT VLBIT;
        SIGNAL rf               : OUT VLBIT_1D (17 DOWNTO 0);
        SIGNAL rf_p             : IN  VLBIT;
        SIGNAL rf_p_ack         : IN  VLBIT;
        SIGNAL rf_p_req         : OUT VLBIT;
        SIGNAL rf_req           : OUT VLBIT;
        SIGNAL sync_ack         : IN  VLBIT;
        SIGNAL sync_nack        : IN  VLBIT;
        SIGNAL sync_req         : OUT VLBIT;
        SIGNAL x_cr0            : OUT VLBIT_1D (31 DOWNTO 0);
        SIGNAL xbr              : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL xbr_ack          : OUT VLBIT;
        SIGNAL xbr_p            : IN  VLBIT;
        SIGNAL xbr_p_ack        : IN  VLBIT;
        SIGNAL xbr_p_req        : OUT VLBIT;
        SIGNAL pf_debug         : OUT VLBIT;
        SIGNAL x_debug          : OUT VLBIT);   -- for debugging only
END dispatch;


ARCHITECTURE behavior OF dispatch IS
BEGIN
  main : PROCESS

    VARIABLE debug              : BOOLEAN := FALSE; -- display everything
    VARIABLE debug_reuse        : BOOLEAN := FALSE; -- display reuse stuff
    VARIABLE trace_dispatch     : BOOLEAN := FALSE; -- display dispatches
    VARIABLE trace_iw           : BOOLEAN := FALSE; -- display IW loads
    VARIABLE trace_excp         : BOOLEAN := FALSE; -- display IW loads
    VARIABLE all_tellme         : BOOLEAN := FALSE; -- for debugging only
    VARIABLE no_tellme          : BOOLEAN := FALSE; -- for debugging only
    VARIABLE reuse_result       : BOOLEAN := TRUE; -- reuse last results
    VARIABLE skip_r0_ops        : BOOLEAN := TRUE; -- speed up OP rD,r0,imm16
    VARIABLE out_of_order       : BOOLEAN := TRUE; -- issue insts out of order


    --- Necessary constants and type definitions.

    VARIABLE IW_tap      : INTEGER := IW_Max;   -- useful IW size
    VARIABLE BR_Max      : INTEGER := 8;        -- Branch Queue size
    VARIABLE XBR_Max     : INTEGER := 8;        -- Exception Branch Queue size
    VARIABLE R1_Max      : INTEGER := 8;        -- R1 Queue size


    --- factors affecting response time

    VARIABLE delay_decode       : TIME := 100 ps; -- dispatch output signals
    VARIABLE delay_doit         : TIME := 100 ps; -- handle DOIT instruction
    VARIABLE delay_done_req     : TIME := 100 ps; -- update IW on done
    VARIABLE delay_getcr        : TIME := 100 ps; -- read CR
    VARIABLE delay_inst_ack     : TIME := 100 ps; -- put new inst in IW
    VARIABLE delay_iw_search    : TIME := 100 ps; -- look for inst to dispatch
    VARIABLE delay_putcr        : TIME := 100 ps; -- write CR
    VARIABLE delay_rte          : TIME := 100 ps; -- restore state on RTE
    VARIABLE delay_save_state   : TIME := 100 ps; -- save state on fault
    VARIABLE delay_sync_ack     : TIME := 100 ps; -- respond to SYNC_ACK
    VARIABLE delay_sync_nack    : TIME := 100 ps; -- respond to SYNC_NACK

    VARIABLE delay_to_use      : TIME;          -- scratch for culumative time


    --- constant definitions

    CONSTANT Tag_Max_Value      : INTEGER := 127;

    CONSTANT ICR_Saved_Control  : INTEGER := 1;
    CONSTANT ICR_Fault_Status   : INTEGER := 2;
    CONSTANT ICR_Fault_Address  : INTEGER := 3;
    CONSTANT ICR_Next_PC        : INTEGER := 4;
    CONSTANT ICR_BR_Count       : INTEGER := 5;
    CONSTANT ICR_R1_Count       : INTEGER := 6;
    CONSTANT ICR_IW_Count       : INTEGER := 7;
    CONSTANT ICR_IW_Faults      : INTEGER := 8;

    --- These control registers are not affected by anything. They're just here
    --- for scratch storage in supervisor mode.

    CONSTANT ICR_S1             : INTEGER := 9;
    CONSTANT ICR_S2             : INTEGER := 10;
    CONSTANT ICR_S3             : INTEGER := 11;
    CONSTANT ICR_S4             : INTEGER := 12;


    --- type definitions

    TYPE tag_sb_type IS ARRAY ( 0 TO Tag_Max_Value ) OF BOOLEAN;

    TYPE cr_type IS ARRAY ( IF_ICR_Min TO IF_ICR_Max ) OF INTEGER;

    TYPE siw_type IS ARRAY ( IF_SIW_Min TO IF_SIW_Max ) OF INTEGER;

    TYPE pflag_type IS ARRAY ( 0 TO 31 ) OF BOOLEAN;

    TYPE lfu_type IS ARRAY ( 0 TO 4 ) OF INTEGER;


    --- control register 0 is located here.

    VARIABLE cr0               : VLBIT_1D (31 DOWNTO 0);


    --- internal variables

    VARIABLE iw                : iw_type;       -- instruction window
    VARIABLE iw_bottom         : INTEGER;       -- next available slot
    VARIABLE wat_flag          : BOOLEAN;       -- wait at top scoreboard bit
    VARIABLE single_flag       : BOOLEAN;       -- single instruction flag
    VARIABLE pc                : INTEGER;       -- program counter
    VARIABLE faulting_instruction : iw_slot;    -- remembers who caused fault
    VARIABLE doit_instruction  : iw_slot;       -- where DOIT came from
```

```
VARIABLE out_count          : INTEGER;       -- number of incomplete insts.
VARIABLE br_count           : INTEGER;       -- entries in Branch Queue
VARIABLE new_br_count       : INTEGER;
VARIABLE xbr_count          : INTEGER;       -- entries in X Branch Queue
VARIABLE new_xbr_count      : INTEGER;
VARIABLE r1_count           : INTEGER;       -- entries in R1 Queue
VARIABLE new_r1_count       : INTEGER;       -- used to decrement r1_count
VARIABLE r1_pending         : INTEGER;
VARIABLE new_r1_pending     : INTEGER;
VARIABLE tag_value          : INTEGER;       -- assign tags to insts.
VARIABLE tag_sb             : tag_sb_type;   -- prevent multiple tag use
VARIABLE waiting_for_inst   : BOOLEAN;       -- controls INST_REQ
VARIABLE cr                 : cr_type;       -- internal control regs
VARIABLE siw                : siw_type;      -- shadow IW slots
VARIABLE scoreboard         : VLBIT_1D(31 DOWNTO 0); -- internal copy only


--- Variables for use in dispatching out-of-order, reusing last results,
--- and skipping r0 operands.

VARIABLE pload              : BOOLEAN;       -- have I seen a prior load?
VARIABLE pstore             : BOOLEAN;       -- have I seen a prior store?
VARIABLE pbranch            : BOOLEAN;       -- have I seen a prior branch?
VARIABLE pcarry             : BOOLEAN;       -- have I seen a prior carry?
VARIABLE pdest              : pflag_type;    -- have I seen prior dest regs?
VARIABLE psrc               : pflag_type;    -- have I seen prior src regs?
VARIABLE ldest              : pflag_type;    -- have I seen issued dests?
VARIABLE fu_num             : INTEGER;       -- number of dest FU
VARIABLE reuseA             : BOOLEAN;       -- flag
VARIABLE reuseB             : BOOLEAN;       -- flag
VARIABLE reuseD             : BOOLEAN;       -- flag
VARIABLE zeroA              : BOOLEAN;       -- flag
VARIABLE latched_in_fu      : lfu_type;      -- tracks latched values
VARIABLE got_probes         : BOOLEAN;       -- have I probed queues?

VARIABLE try_dispatch       : BOOLEAN;       -- hack to dispatch more often
VARIABLE done_dispatch      : BOOLEAN;       -- hack to do it only once
VARIABLE done_ex_dispatch   : BOOLEAN;       -- hack to do it only once

--- Input signal event variables. These are the events which wake up this
--- process, for various reasons.  The first four are externally
--- arbitrated. The control register operations don't have to be, since
--- control registers are only written here under quiescent conditions.

VARIABLE l_inst_ack         : VLBIT;         -- arbitrated
VARIABLE l_done_req         : VLBIT;         -- arbitrated
VARIABLE l_external_int     : VLBIT;         -- arbitrated
VARIABLE l_getcr_req        : VLBIT;         -- can be done at any time
VARIABLE l_putcr_req        : VLBIT;         -- only done by PUTCR inst.


--- output signal value variables

VARIABLE v_br_ack           : VLBIT;
VARIABLE v_br_p_req        : VLBIT;
VARIABLE v_done_ack        : VLBIT;
VARIABLE v_ex              : VLBIT_1D (127 DOWNTO 0);
VARIABLE v_ex_p_req        : VLBIT;
VARIABLE v_ex_req          : VLBIT;
VARIABLE v_fault_pending   : VLBIT;
VARIABLE v_i_addr          : VLBIT_1D (31 DOWNTO 0);
VARIABLE v_inst_req        : VLBIT;
VARIABLE v_reply_hazy      : VLBIT;
VARIABLE v_rf              : VLBIT_1D (17 DOWNTO 0);
VARIABLE v_rf_p_req        : VLBIT;
VARIABLE v_rf_req          : VLBIT;
VARIABLE v_sb_r_r          : VLBIT;
VARIABLE v_sb_w_r          : VLBIT;
VARIABLE v_sync_req        : VLBIT;
VARIABLE v_xbr_ack         : VLBIT;
VARIABLE v_xbr_p_req       : VLBIT;
VARIABLE v_x_debug         : VLBIT;
VARIABLE v_pf_debug        : VLBIT;

--- scratch variables

VARIABLE tmpi              : INTEGER;
VARIABLE tmpb1             : BOOLEAN;
VARIABLE tmpb2             : BOOLEAN;
VARIABLE tmpv              : VLBIT_1D (31 DOWNTO 0);
VARIABLE tmpmvbr           : VLBIT_1D (31 DOWNTO 0);

VARIABLE iw_index          : INTEGER;       -- temp for searching IW
VARIABLE vA, vB, vD        : BOOLEAN;       -- valid flags for operands
VARIABLE srcA, srcB, srcD  : INTEGER;       -- source operand registers
VARIABLE dest              : INTEGER;       -- destination register
VARIABLE op_type           : INTEGER;       -- decode instruction type
VARIABLE status            : INTEGER;       -- status of decoded inst.
VARIABLE single, wat       : BOOLEAN;       -- flags for decoded inst.
VARIABLE tellme            : BOOLEAN;       -- flags for decoded inst.
VARIABLE found_one         : BOOLEAN;       -- found inst to issue
VARIABLE icount            : INTEGER;       -- count issued insts in IW

--- statistics

VARIABLE s_inst_ack        : INTEGER;       -- got new instruction for IW
VARIABLE s_iw_total        : INTEGER;       -- cumulative sum of IW count
VARIABLE s_iw_max          : INTEGER;       -- maximum IW fullness
VARIABLE s_done_req        : INTEGER;       -- reported done status
VARIABLE s_dispatch_req    : INTEGER;       -- tried to dispatch
VARIABLE s_did_dispatch    : INTEGER;       -- actually did it (not doit)
VARIABLE s_excp_dispatch   : INTEGER;       -- same, but only for faults
VARIABLE s_need_operands   : INTEGER;       -- inst. needed operands
VARIABLE s_excp_status     : INTEGER;       -- done status is faulty
VARIABLE s_did_excp        : INTEGER;       -- invoked exception handler
VARIABLE s_zeroA           : INTEGER;       -- count zeroA times

VARIABLE s_reuse_result    : INTEGER;       -- count last-result-reuse
```

```
  VARIABLE s_reuse_helps     : INTEGER;       -- count when it helps
  VARIABLE s_iw_ooo_count    : INTEGER;       -- out-of-order insts issued
  VARIABLE s_iw_ooo_total    : INTEGER;       -- how far out-of-order total
  VARIABLE s_iw_ooo_max      : INTEGER;       -- maximum out-of-order dist



  --- other stuff

  CONSTANT EX_PADDING : VLBIT_1D(127 DOWNTO 0) :=
    ( '0','0','0','0','0','0','0','0',
      '0','0','0','0','0','0','0','0',
      '0','0','0','0','0','0','0','0',
      '0','0','0','0','0','0','0','0',
      '0','0','0','0','0','0','0','0',
      '0','0','0','0','0','0','0','0',
      '0','0','0','0','0','0','0','0',
      '0','0','0','0','0','0','0','0',
      '0','0','0','0','0','0','0','0',
      '0','0','0','0','0','0','0','0',
      '0','0','0','0','0','0','0','0',
      '0','0','0','0','0','0','0','0',
      '0','0','0','0','0','0','0','0',
      '0','0','0','0','0','0','0','0',
      '0','0','0','0','0','0','0','0',
      '0','0','0','0','0','0','0','0' );

BEGIN


  ---
  --- If clear is asserted, reset everything and stick here.
  ---

  IF clr /= '1' THEN                          -- reset to beginning

    cr0 := int2v1d( init_cr0 );               -- init CR0
    x_cr0 <= cr0;

    FOR i IN 0 TO (IW_Max - 1) LOOP   -- clear IW
      iw(i).valid := FALSE;
    END LOOP;
    iw_bottom := 0;

    FOR i IN 0 TO Tag_Max_Value LOOP          -- clear the tag scoreboard
      tag_sb(i) := FALSE;
    END LOOP;

    wat_flag := FALSE;
    single_flag := FALSE;
    pc := start_addr;
    tag_value := 0;
    out_count := 0;
    br_count := 0;
```

```
      xbr_count := 0;
      r1_count := 0;
      new_r1_count := 0;
      r1_pending := 0;
      new_r1_pending := 0;
      waiting_for_inst := FALSE;
      done_dispatch := FALSE;
      done_ex_dispatch := FALSE;

      l_inst_ack := '0';
      l_done_req := '0';
      l_external_int := '0';
      l_getcr_req := '0';
      l_putcr_req := '0';

      v_br_ack := '0';
      v_br_p_req := '0';
      v_done_ack := '0';
      v_ex_p_req := '0';
      v_ex_req := '0';
      v_fault_pending := '0';
      v_inst_req := '0';
      v_reply_hazy := '0';
      v_rf_p_req := '0';
      v_rf_req := '0';
      v_sb_r_r := '0';
      v_sb_w_r := '0';
      v_sync_req := '0';
      v_xbr_ack := '0';
      v_xbr_p_req := '0';

      br_ack <= v_br_ack;                     -- now set outputs
      br_p_req <= v_br_p_req;
      done_ack <= v_done_ack;
      ex <= EX_PADDING;
      ex_p_req <= v_ex_p_req;
      ex_req <= v_ex_req;
      fault_pending <= v_fault_pending;
      getcr <= ZERO(31 DOWNTO 0);
      getcr_ack <= '0';
      inst_ack_ack <= '0';
      inst_req <= v_inst_req;
      putcr_ack <= '0';
      reply_hazy <= v_reply_hazy;
      rf <= ZERO( 17 DOWNTO 0 );
      rf_p_req <= v_rf_p_req;
      rf_req <= v_rf_req;
      sb_num <= ('0','0','0','0','0');
      sb_r_r <= v_sb_r_r;
      sb_r_d <= v_sb_r_r;
      sb_set <= '0';
      sb_w_r <= v_sb_w_r;
      sync_req <= v_sync_req;
      xbr_ack <= v_xbr_ack;
      xbr_p_req <= v_xbr_p_req;

      v_i_addr := int2v1d(pc);
      i_addr <= (v_i_addr(31 DOWNTO 2) & '0' & '0'); -- just for now

      v_x_debug := '0';
      x_debug <= v_x_debug;

      v_pf_debug := '0';
      pf_debug <= v_pf_debug;

      --- clear statistics

      s_inst_ack := 0;
      s_iw_total := 0;
      s_iw_max := 0;
      s_done_req := 0;
      s_dispatch_req := 0;
      s_did_dispatch := 0;
      s_excp_dispatch := 0;
      s_need_operands := 0;
      s_excp_status := 0;
      s_did_excp := 0;
      s_zeroA := 0;
      s_reuse_result := 0;
      s_reuse_helps := 0;
      s_iw_ooo_count := 0;
      s_iw_ooo_total := 0;
      s_iw_ooo_max := 0;


      WAIT UNTIL clr = '1';                    -- wait till ready to run

      --- Send out first instruction request to start things off.

      v_i_addr := int2v1d(pc);                 -- get 32-bit PC value
      i_addr <= (v_i_addr(31 DOWNTO 2) & cr0(Supervisor) & '0');
      v_inst_req := NOT v_inst_req;
      inst_req <= v_inst_req AFTER delay;

      waiting_for_inst := TRUE;

      IF debug THEN
        puthexline("sent out INST_REQ for address ",pc);
      END IF;


    ELSE                                       -- unit is running

---- Normal running condition

      --- The process loses events if I'm busy elsewhere, so I need to make my
      --- own virtual wait statement, instead of relying on the actual wait
      --- from VHDL. I'll just sample by hand, and only wait if there are no
      --- pending requests. Since the REQ/ACK is paired, this will work fine.
      --- The critical signals are externally arbitrated, so I can respond to
      --- any that I see here.

      got_probes := FALSE;
      try_dispatch := TRUE;

      done_dispatch := FALSE;

      IF ex_p = '0' THEN
        done_ex_dispatch := FALSE;
      END IF;

      WHILE ( inst_ack /= l_inst_ack OR
              try_dispatch OR
              done_req /= l_done_req OR
              external_int /= l_external_int OR
              getcr_req /= l_getcr_req OR
              putcr_req /= l_putcr_req )
      LOOP

        IF debug THEN
          putline("Top of event loop");
        END IF;


        --- Requests to read and write internal control registers must always
        --- be handled, but don't need to be arbitrated, since the internal
        --- control registers are only changed when starting and ending
        --- exception processing (when everything has stopped), and by PUTCR
        --- instructions (when everything has also stopped).

        IF getcr_req /= l_getcr_req THEN        -- request to read control reg

          l_getcr_req := getcr_req;

          tmpi := v1d2int( cr_num );            -- which register

          IF debug THEN
            putline("handling GETCR request for CR",tmpi);
          END IF;

          --- NOTE: This would normally need to be four-phase, but since there
          --- is only one unit which can read the control registers, it should
          --- work fine.

          IF tmpi >= IF_ICR_Min AND tmpi <= IF_ICR_Max THEN
            getcr <= int2v1d( cr(tmpi) );       -- actual register value
          ELSIF tmpi >= IF_SIW_Min AND tmpi <= IF_SIW_Max THEN
            getcr <= int2v1d( siw(tmpi) );      -- shadow IW
          ELSIF tmpi = 0 THEN
            getcr <= cr0;                       -- this isn't necessary
          ELSE
            getcr <= ZERO(31 DOWNTO 0);         -- no register here, use zero
          END IF;

          getcr_ack <= getcr_req AFTER delay_getcr;

        END IF;                                -- getcr_req


        --- Reading and writing can't happen at the same time, since PUTCR is
        --- a SINGLE instruction, which must be the only one active.

        IF putcr_req /= l_putcr_req THEN       -- request to write control reg

          l_putcr_req := putcr_req;

          tmpi := v1d2int( cr_num );           -- which register to write

          IF debug THEN
            putline("handling PUTCR request for CR",tmpi);
          END IF;

          IF (tmpi >= IF_ICR_Min AND tmpi <= IF_ICR_Max) THEN -- control reg
            cr(tmpi) := v1d2int( putcr );
          ELSIF (tmpi >= IF_SIW_Min AND tmpi <= IF_SIW_Max) THEN -- shadow IW
            siw(tmpi) := v1d2int( putcr );
          ELSIF tmpi = 0 THEN                  -- cr0
            cr0 := putcr;
            x_cr0 <= cr0;
          END IF;

          putcr_ack <= putcr_req AFTER delay_putcr;

        END IF;                                -- putcr_req


        IF inst_ack /= l_inst_ack THEN         -- new operand has arrived

          l_inst_ack := inst_ack;

          s_inst_ack := s_inst_ack + 1;

          IF debug THEN
            putline("got INST_ACK event");
          END IF;

          waiting_for_inst := FALSE;

          --- Is there room?  There had better be!

          IF ((IW_tap - iw_bottom) >= 1) THEN -- yes, there's room

            --- see if it really arrived, or if there was a memory fault

            tmpi := STATUS_none;               -- assume there's not
```

```
--- check for memory faults

IF ( ( i_status(0) = '1' AND cr0(Imem0) = '1' ) OR
     ( i_status(1) = '1' AND cr0(Imem1) = '1' ) OR
     ( i_status(2) = '1' AND cr0(Imem2) = '1' ) OR
     ( i_status(3) = '1' AND cr0(Imem3) = '1' ) )
THEN
  tmpi := STATUS_imem;
END IF;

iw(iw_bottom).status := tmpi;       -- save status
iw(iw_bottom).valid := TRUE;        -- mark it valid
iw(iw_bottom).address := v1d2int( v_i_addr );
iw(iw_bottom).issued := FALSE;


IF tmpi /= STATUS_none THEN         -- yes, there's a fault

  IF debug THEN
    putline("found a memory fault");
  END IF;

  IF v_fault_pending = '0' THEN     -- might need to signal it

    faulting_instruction.status := tmpi;
    faulting_instruction.address := v1d2int( v_i_addr );

    v_fault_pending := '1';         -- do so
    fault_pending <= v_fault_pending;
    v_x_debug := NOT v_x_debug;
    x_debug <= v_x_debug;
    IF trace_excp THEN
      putline("IMem exception: out_count = ",out_count);
    END IF;

  END IF;

ELSE                                -- no fault

  --- Just stick the new instructions in the IW. They will be
  --- decoded at dispatch time. This simplifies the return from
  --- exceptions, since I don't have to do anything special to
  --- the IW, other than to just reload it with nothing
  --- dispatched. Well, that's not quite true: I will need to set
  --- the single_flag again.

  --- I do need to set the single_flag now, so that I don't fetch
  --- anything that I don't need. I could just throw away all
  --- instructions after the SINGLE instruction, but that would
  --- be messier.

  tmpi := v1d2int( i_data );        -- get opcode

  IF tmpi /= OPCODE_doit THEN
```

```
    iw(iw_bottom).opcode := tmpi;

    IF debug OR trace_iw THEN
      put("IW(",iw_bottom);
      puthex(") got opcode ",iw(iw_bottom).opcode);
      puthexline(" from address ",iw(iw_bottom).address);
    END IF;

    IF is_single( i_data ) THEN     -- see if it's single
      iw(iw_bottom).single := TRUE;
      single_flag := TRUE;
      IF debug THEN
        putline("single flag is now ",single_flag);
      END IF;
    END IF;

    iw_bottom := iw_bottom + 1;     -- next slot

    s_iw_total := s_iw_total + iw_bottom;

    IF iw_bottom > s_iw_max THEN
      s_iw_max := iw_bottom;
    END IF;

  ELSE                              -- explicit doit

    cr0(Doit) := '1';               -- set Doit flag
    single_flag := TRUE;

    doit_instruction.address := v1d2int( v_i_addr );
    doit_instruction.opcode := tmpi;

    IF debug OR trace_iw THEN
      put("IW(-) ");
      puthex("got opcode ",iw(iw_bottom).opcode);
      puthexline(" from address ",iw(iw_bottom).address);
    END IF;

    IF debug THEN
      putline("single flag is now ",single_flag);
    END IF;


    --- signal desire for normal branch target
    IF cr0(Excp_Enable) = '1' THEN
      v_pf_debug := NOT v_pf_debug;
      pf_debug <= v_pf_debug;
    END IF;

  END IF;


  IF i_data(31) = '1' THEN          -- add implicit doit
```

```
        cr0(Doit) := '1';               -- set Doit flag
        single_flag := TRUE;

        doit_instruction.address := v1d2int( v_i_addr );
        doit_instruction.opcode := tmpi;

        IF debug OR trace_iw THEN
          putline("got implicit DOIT");
        END IF;
        IF debug THEN
          putline("single flag is now ",single_flag);
        END IF;

        --- signal desire for normal branch target
        IF cr0(Excp_Enable) = '1' THEN
          v_pf_debug := NOT v_pf_debug;
          pf_debug <= v_pf_debug;
        END IF;

      END IF;



      --- I can go ahead and increment the program counter now,
      --- because if I just added a SINGLE instruction, the PC won't
      --- be used until the instruction has finished, at which time
      --- the PC will have been changed if it needs to be. NOTE: This
      --- means that the DOIT instruction must either replace the PC
      --- or leave it alone. It shouldn't increment it. Same for RTE,
      --- and PUTCR only affects the user/supervisor mode bit.

      pc := pc + 4;                     -- increment program counter

      IF debug THEN
        puthexline("PC advanced to ",pc);
      END IF;

    END IF;                             -- (no memory fault)

  END IF;                               -- (there's room in the IW)

  inst_ack_ack <= inst_ack AFTER delay_inst_ack;

  IF debug THEN
    putline("done with this event");
  END IF;

END IF;                                 -- inst_ack


IF done_req /= l_done_req THEN       -- instruction has completed

  l_done_req := done_req;
```

```
s_done_req := s_done_req + 1;

IF debug THEN
  putline("got DONE_REQ event");
END IF;

--- Find instruction associated with the tag.

tmpi := v1d2int( done(7 DOWNTO 0) );  -- get tag

IF debug THEN
  putline("looking for tag ",tmpi);
END IF;

iw_index := -1;
FOR i IN 0 TO (iw_bottom - 1) LOOP    -- it will do 0 to 0
  IF iw(i).valid AND iw(i).issued AND iw(i).tag = tmpi THEN -- got it
    iw_index := i;
    EXIT;
  END IF;
END LOOP;

--- If tag wasn't found, crash and burn.

ASSERT iw_index /= -1 REPORT "Retiring Invalid Tag" SEVERITY FAILURE;

IF debug THEN
  putline("found it in slot ",iw_index);
END IF;


--- We've heard from the instruction. Count it off.

out_count := out_count - 1;

IF debug THEN
  putline("now out_count = ",out_count);
END IF;


--- Mark the tag as not in use. This is safe, because if the
--- instruction is faulty, nothing more will be dispatched until
--- the fault is handled. Right?

tag_sb(tmpi) := FALSE;


--- now handle reported status

tmpi := v1d2int( done(15 DOWNTO 8) ); -- get major status

IF debug THEN
  putline("major status is ",tmpi);
```

```
END IF;


IF tmpi = STATUS_done THEN            -- No fault. Remove it.

  IF iw(iw_index).dest == 1 THEN      -- produced r1 value
    r1_count := r1_count + 1;         -- so count it,
    r1_pending := r1_pending - 1;     -- since it succeeded

    --- clear scoreboard for R1

    sb_set <= '0';
    sb_num <= REG_R1;
    v_sb_w_r := NOT v_sb_w_r;
    sb_w_r <= v_sb_w_r AFTER delay;
    WAIT ON sb_w_a;

    IF debug THEN
      putline("r1_count is now ",r1_count);
      putline("r1_pending is now ",r1_pending);
      putline("clearing scoreboard bit ",iw(iw_index).dest);
    END IF;
  END IF;

  iw_remove( iw_index, iw, iw_bottom, wat_flag, single_flag );

  IF debug THEN
    putline("wat_flag is now ",wat_flag);
    putline("single_flag is now ",single_flag);
  END IF;

ELSE                                  -- faulty instruction

  s_excp_status := s_excp_status + 1;

  --- Save minor status as well as major. In this register, the minor
  --- status is the upper 16 bits, while the major status is the
  --- lower 16 bits.  Only 8 bits of each are actually returned from
  --- execution units, but the internally generated major status is
  --- actually 9 bits wide, so we split the register this way.

  tmpi := tmpi + 65536 * v1d2int( done(23 DOWNTO 16) );
  iw(iw_index).status := tmpi;        -- save status with instruction

  --- save other reported values
  iw(iw_index).arg1 := v1d2int( done(87 DOWNTO 56) );
  iw(iw_index).arg2 := v1d2int( done(55 DOWNTO 23) );

  IF iw(iw_index).dest >= 1 THEN      -- clear scoreboard bit if set

    IF debug THEN
      putline("clearing scoreboard bit ",iw(iw_index).dest);
    END IF;
```

```
        sb_set <= '0';
        tmpv := int2v1d( iw(iw_index).dest  ); -- IDIOTIC ViewLogic crap
        sb_num <= tmpv(4 DOWNTO 0);
        v_sb_w_r := NOT v_sb_w_r;
        sb_w_r <= v_sb_w_r AFTER delay;
        WAIT ON sb_w_a;

      END IF;

      IF v_fault_pending = '0' THEN       -- need to signal fault

        faulting_instruction := iw(iw_index); -- remember who did it

        v_fault_pending := '1';           -- do so
        fault_pending <= v_fault_pending;
        v_x_debug := NOT v_x_debug;
        x_debug <= v_x_debug;
        IF trace_excp THEN
          putline("faulty status: out_count = ",out_count);
        END IF;

      END IF;

    END IF;

    v_done_ack := NOT v_done_ack;
    done_ack <= v_done_ack AFTER delay_done_req;

    IF debug THEN
      putline("done with this event");
    END IF;

  END IF;                                 -- done_req


  IF external_int /= l_external_int THEN  -- guess what

    IF debug THEN
      putline("got EXTERNAL_INT event");
    END IF;

    l_external_int := external_int;

    --- decide whether to postpone the INT or not.  If it's
    --- disabled, or if I'm in the process of switching to an
    --- exception handler already, then put it off until later.

    IF cr0(Int_Enable) = '0' OR v_fault_pending = '1' THEN

      v_reply_hazy := NOT v_reply_hazy;   -- ask again later
      reply_hazy <= v_reply_hazy;

      IF debug THEN
        putline("reply hazy - ask again later");
      END IF;

    ELSE                                -- I can do it now

      faulting_instruction.status := STATUS_int;

      v_fault_pending := '1';           -- do so
      fault_pending <= v_fault_pending;
      v_x_debug := NOT v_x_debug;
      x_debug <= v_x_debug;
      IF trace_excp THEN
        putline("interrupt: out_count = ",out_count);
      END IF;

      try_dispatch := FALSE;

    END IF;


    IF debug THEN
      putline("done with this event");
    END IF;

  END IF;                               -- external_int



  IF NOT done_dispatch AND try_dispatch THEN -- try to dispatch something

    try_dispatch := FALSE;

    s_dispatch_req := s_dispatch_req + 1;

    IF debug THEN
      putline("got try_dispatch signal");
    END IF;


    --- Now try to dispatch an instruction from the IW.  All decoding,
    --- fault detection, tag assignments, etc. is done here to simplify
    --- returning from exceptions. I don't have to preload anything, just
    --- start dispatching again. If it's dispatched externally, increment
    --- out_count; DOITs are a special case, since they are handled
    --- entirely within this module, and don't produce a done_req
    --- signal. MVPC and MVBR must provide operands to the instruction.

    delay_to_use := delay;

    found_one := FALSE;                 -- looking for one

    IF ( v_fault_pending = '0' and      -- no dispatch while faults
         NOT wat_flag )                 -- no dispatch until cleared
    THEN
```

```
IF debug THEN
  putline("Looking for something to dispatch");
END IF;

delay_to_use := delay_iw_search;    -- time to find valid inst.


--- set up data structures for loop

pload := FALSE;
pstore := FALSE;
pbranch := FALSE;
pcarry := FALSE;
pdest := ( FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,
           FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,
           FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,
           FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE );
psrc := ( FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,
          FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,
          FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,
          FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE );
ldest := ( FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,
           FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,
           FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,
           FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE );

icount := 0;

FOR i IN 0 TO (iw_bottom - 1) LOOP  -- check each slot for valid

  IF ( NOT iw(i).valid ) THEN       -- stop when I run out

    IF debug THEN
      putline("end of IW entries");
    END IF;

    EXIT;

  END IF;

  IF debug THEN
    putline("looking at iw slot ",i);
    put("  valid = ",iw(i).valid);
    put("  issued = ",iw(i).issued);
    putline("  tag = ",iw(i).tag);
    puthexline("  status = ",iw(i).status);
    puthexline("  address = ",iw(i).address);
    puthexline("  opcode = ",iw(i).opcode);
  END IF;


  IF iw(i).issued THEN

    icount := icount + 1;           -- count issued insts in IW
    --- Keep track of all the currently pending destinations in
    --- case I want to reuse them.

    IF reuse_result THEN

      IF debug_reuse OR debug THEN
        putline("setting ldest ",iw(i).dest);
      END IF;

      ldest(iw(i).dest) := TRUE;

    END IF;

    NEXT;                          -- skip issued instructions

  END IF;


  --- Now, I'm looking at a valid, non-issued instruction. Decode
  --- it and see if I can issue it.

  tmpb1 := vlb2boo( cr0(Supervisor) ); -- stupid Viewlogic crap
  tmpb2 := vlb2boo( cr0(Excp_Enable) ); -- stupid Viewlogic crap
  decode( BR_Max, XBR_Max, R1_Max,  -- queue length inputs
          iw(i).opcode, tmpb1, tmpb2, -- inputs
          r1_count, r1_pending, br_count, xbr_count, -- inputs
          new_r1_count, new_r1_pending, -- outputs
          new_br_count, new_xbr_count, -- outputs
          status, op_type, single, wat, tellme, -- outputs
          vA, srcA, vB, srcB, vD, srcD, dest); -- outputs

  IF all_tellme THEN               -- for debugging only
    tellme := TRUE;
  ELSIF no_tellme THEN
    tellme := FALSE;
  END IF;

  iw(i).single := single;
  iw(i).wat := wat;
  iw(i).status := status;
  iw(i).dest := dest;              -- used for starting exceptions

  IF debug THEN
    putline("found a non-issued instruction in slot ",i);
  END IF;

  IF status /= STATUS_none THEN    -- need to signal fault

    faulting_instruction := iw(i);  -- remember who did it

    v_fault_pending := '1';         -- do so
    fault_pending <= v_fault_pending;
    v_x_debug := NOT v_x_debug;
    x_debug <= v_x_debug;

    IF debug THEN
      puthexline("but it's faulty. Status = 0x",status);
    END IF;

    IF trace_excp THEN
      put("invalid instruction:");
      puthex(" op ",iw(i).opcode);
      puthexline(" addr ",iw(i).address);
      putline("invalid instruction: out_count = ",out_count);
    END IF;

    EXIT;                           -- skip out of loop

  END IF;


  IF wat AND i /= 0 THEN            -- only dispatch WAT at top

    IF debug THEN
      putline("can't dispatch it: WAT");
    END IF;

    EXIT;                           -- jump out to prevent more
  END IF;


  --- Check for out-of-order dependencies. I need to do all the
  --- tests first, then go to the next slot if it's not valid, so
  --- that I can update the flags for this slot after I've
  --- considered the instruction. This stuff is only considered
  --- based on non-issued instructions. The scoreboard handles
  --- interaction between issued instructions.

  tmpb1 := TRUE;                    -- is it valid? assume so.

  IF ( pdest(dest) OR               -- destination is prior dest
       psrc(dest) OR                -- destination is prior source
         ( vA AND pdest(srcA) ) OR    -- sourceA is prior dest
         ( vB AND pdest(srcB) ) OR    -- sourceB is prior dest
         ( vD AND pdest(srcD) ) )     -- sourceD is prior dest
  THEN

    IF debug THEN
      putline("can't dispatch it: out-of-order dependency");
    END IF;

    tmpb1 := FALSE;

  END IF;


  --- instructions writing to the branch queue must be sequential
```

```
IF ( op_type = OP_branch OR op_type = OP_ldbr ) AND pbranch THEN

  IF debug THEN
    putline("can't dispatch it: prior branches");
  END IF;

  tmpb1 := FALSE;

END IF;


--- Instructions using the carry bit must be sequential. This is
--- a bit of cheating using the opcodes to watch for instructions
--- which use the carry bit. I should do this in p_dispatch, but
--- I'm in a hurry.

tmpv := int2v1d( iw(i).opcode );
tmpi := op2index( tmpv );

IF ( ( tmpi = 40 OR tmpi = 41 OR tmpi = 46 OR tmpi = 47 ) AND
     ( tmpv(8) = '1' OR tmpv(9) = '1' ) AND pcarry )
THEN

  IF debug THEN
    putline("can't dispatch it: prior carry");
  END IF;

  tmpb1 := FALSE;

END IF;


--- also check for MEM unit dependencies

--- I can reorder loads unless I've seen a store.
IF op_type = OP_load AND pstore THEN

  IF debug THEN
    putline("can't dispatch it: prior stores");
  END IF;

  tmpb1 := FALSE;

END IF;


--- I can't reorder stores at all.
IF ( ( op_type = OP_store OR op_type = OP_xmem ) AND
     ( pload OR pstore ) )
THEN

  IF debug THEN
    putline("can't dispatch it: prior memory instructions");
  END IF;

  tmpb1 := FALSE;

END IF;


--- Now update the out-of-order data structures for this slot,
--- and then skip out if it's not valid.

IF dest /= 0 THEN
  pdest(dest) := TRUE;
  IF debug THEN
    putline("setting pdest ",dest);
  END IF;
END IF;

IF vA AND srcA /= 0 THEN
  psrc(srcA) := TRUE;
  IF debug THEN
    putline("setting psrc ",srcA);
  END IF;
END IF;
IF vB AND srcB /= 0 THEN
  psrc(srcB) := TRUE;
  IF debug THEN
    putline("setting psrc ",srcB);
  END IF;
END IF;
IF vD AND srcD /= 0 THEN
  psrc(srcD) := TRUE;
  IF debug THEN
    putline("setting psrc ",srcD);
  END IF;
END IF;

IF op_type = OP_branch OR op_type = OP_ldbr THEN
  pbranch := TRUE;
  IF debug THEN
    putline("setting pbranch");
  END IF;
END IF;

IF ( ( tmpi = 40 OR tmpi = 41 OR tmpi = 46 OR tmpi = 47 ) AND
     ( tmpv(8) = '1' OR tmpv(9) = '1' ) )
THEN
  pcarry := TRUE;
  IF debug THEN
    putline("setting pcarry");
  END IF;
END IF;

IF op_type = OP_load THEN
  pload := TRUE;
```

```
              IF debug THEN
                putline("setting pload");
              END IF;
            END IF;

            IF op_type = OP_store OR op_type = OP_xmem THEN
              pstore := TRUE;
              IF debug THEN
                putline("setting pstore");
              END IF;
            END IF;


            NEXT WHEN NOT tmpb1;              -- go to next IW slot




            --- Check to see if queues are ready. I could just probe the
            --- queues that I'm interested in, but since I might want them
            --- all, and it's pretty likely to take longer to probe each one
            --- as needed instead of probing them all, I'll just probe them
            --- now.

----        IF NOT got_probes THEN
----
----          v_ex_p_req := NOT v_ex_p_req;   -- probe EX queue
----          ex_p_req <= v_ex_p_req AFTER delay;
----
----          v_rf_p_req := NOT v_rf_p_req;   -- probe RF queue
----          rf_p_req <= v_rf_p_req AFTER delay;
----
----          v_br_p_req := NOT v_br_p_req;   -- probe BR queue (for MVBR)
----          br_p_req <= v_br_p_req AFTER delay;
----
----          v_sb_r_r := NOT v_sb_r_r;       -- read the scoreboard
----          sb_r_r <= v_sb_r_r AFTER delay;
----
----
----          --- I need to wait for all four signals to acknowledge.
----
----          WAIT UNTIL ( ex_p_ack = v_ex_p_req AND
----                       rf_p_ack = v_rf_p_req AND
----                       br_p_ack = v_br_p_req AND
----                       sb_r_g = v_sb_r_r );
----
----
              scoreboard := sb;               -- grab a copy
----          sb_r_d <= sb_r_g;               -- unlock scoreboard
----
----
----          got_probes := TRUE;             -- only do this once
----
----        END IF;
```

```
--- Now use the queue information to determine whether I can
--- dispatch or not.


--- This is not quite right, since RTE and SYNC don't use the EX
--- queue, but those don't happen very often, especially when the
--- queue isn't ready.

IF ex_p = '0' OR done_ex_dispatch THEN -- room to issue new inst?

  IF debug THEN
    putline("EX queue isn't ready");
  END IF;

  NEXT;

END IF;


IF ( vA OR vB OR vD ) AND rf_p = '0' THEN -- need RF operands

  IF debug THEN
    putline("RF queue isn't ready");
  END IF;

  NEXT;

END IF;


IF op_type = OP_mvbr AND br_p = '0' THEN -- need branch target

  --- Can't dispatch if there's no branch target there. It must
  --- come from the normal Branch Queue, regardless of which
  --- queue is active.

  IF debug THEN
    putline("BR queue not ready (MVBR instruction)");
  END IF;

  NEXT;

END IF;


--- Before I check the scoreboard, see if I can reuse the last
--- result from a particular functional unit. I can only do this
--- if I haven't seen the particular destination register used as
--- a destination in the currently executing instructions in the
--- IW, so that I know that the result will be valid when I use
--- it.

IF reuse_result THEN              -- see where I can reuse values
```

```
  --- to which unit is this instruction going?
  CASE op_type IS
  WHEN OP_arith =>                -- unit 0
    fu_num := OP_arith;
  WHEN OP_logic =>                -- unit 1
    fu_num := OP_logic;
  WHEN OP_ctrl | OP_mvbr =>       -- unit 2
    fu_num := OP_ctrl;
  WHEN OP_load | OP_store | OP_xmem => -- unit 3
    fu_num := OP_load;
  WHEN OP_branch | OP_ldbr =>     -- unit 4
    fu_num := OP_branch;
  WHEN OTHERS =>                  -- not implemented
    fu_num := OP_branch;          -- does not latch
  END CASE;

  --- can I use the last value that's there?

  reuseA := ( vA AND srcA /= 0 AND srcA /= 1 AND
              NOT ldest(srcA) AND
              latched_in_fu(fu_num) = srcA );
  reuseB := ( vB AND srcB /= 0 AND srcB /= 1 AND
              NOT ldest(srcB) AND
              latched_in_fu(fu_num) = srcB );
  reuseD := ( vD AND srcD /= 0 AND srcD /= 1 AND
              NOT ldest(srcD) AND
              latched_in_fu(fu_num) = srcD );

ELSE                              -- last-result-reuse disabled

  reuseA := FALSE;
  reuseB := FALSE;
  reuseD := FALSE;

END IF;


--- Modify vA, vB, vD according to reuse policy, since these are
--- used to request operands.  I can ignore scoreboard conflicts
--- if I'm reusing the last result for that operand.

zeroA := skip_r0_ops AND vA AND (srcA = 0);

vA := vA AND NOT reuseA AND NOT zeroA;
vB := vB AND NOT reuseB;
vD := vD AND NOT reuseD;



--- Now check scoreboard conflicts.

IF ( (vA AND scoreboard(srcA)='1') OR
     (vB AND scoreboard(srcB)='1') OR
     (vD AND scoreboard(srcD)='1') OR
     scoreboard(dest)='1' )       -- also prevent WAW
THEN

  IF debug THEN
    putline("can't dispatch it: scoreboard");
  END IF;

  NEXT;

END IF;


--- If I get this far, then the current instruction is a valid
--- one to dispatch. The only thing that could prevent dispatch
--- is if I don't allow out-of-order issue.

--- See if I'm allowed to issue out-of-order.

tmpi := i - icount;               -- out-of-order distance
IF tmpi > 0 THEN

  IF NOT out_of_order THEN

    IF debug THEN
      putline("can't dispatch it: no out-of-order");
    END IF;

    EXIT;

  END IF;

  --- Okay, everything is ready.  Gather some stats on it.

  s_iw_ooo_count := s_iw_ooo_count + 1;
  s_iw_ooo_total := s_iw_ooo_total + tmpi;
  IF tmpi > s_iw_ooo_max THEN
    s_iw_ooo_max := tmpi;
  END IF;
END IF;


IF debug THEN
  putline("Found a dispatchable instruction in IW slot ",i);
END IF;


--- remember destination reg & unit for reuse later

IF ( reuse_result AND             -- enabled
     dest /= 0 AND dest /= 1 )    -- valid register
THEN                              -- valid destination to latch

  FOR k IN 0 TO 4 LOOP
```

```
          IF latched_in_fu(k) = dest THEN -- found it somewhere else
            latched_in_fu(k) := 0;
          END IF;
        END LOOP;

        latched_in_fu(fu_num) := dest;  -- remember who has it

      END IF;


      --- gather some more statistics

      IF zeroA THEN

        IF (NOT vB) AND (NOT vD) THEN   -- just count useful ones
          s_zeroA := s_zeroA + 1;
        END IF;

        IF debug THEN
          putline("zeroA");
        END IF;

      END IF;

      IF reuseA OR reuseB OR reuseD THEN

        s_reuse_result := s_reuse_result + 1; -- could reuse result

        IF ( (reuseA AND scoreboard(srcA)='1') OR
             (reuseB AND scoreboard(srcB)='1') OR
             (reuseD AND scoreboard(srcD)='1') )
        THEN
          s_reuse_helps := s_reuse_helps + 1; -- it actually helps
        END IF;

        IF debug OR debug_reuse THEN
          putline("reuseA is ",reuseA);
          putline("reuseB is ",reuseB);
          putline("reuseD is ",reuseD);
        END IF;

      END IF;


      found_one := TRUE;
      iw_index := i;
      EXIT;                             -- hop out and dispatch it

    END LOOP;                           -- end of look-for-instruction


    --- If I have something to dispatch, the loop below does the
    --- dispatch, but doesn't actually loop. It just provides a
    --- convenient way to abort for any reason.
```

```
WHILE found_one LOOP                -- iw_index says which one

  --- Now, do different things, depending on the instruction. Some
  --- are executed entirely in this module, and are not actually
  --- dispatched through external queues. DOITs are handled later,
  --- and should no longer appear in the IW at all.

  IF op_type = OP_rte THEN          -- rte instruction

    IF trace_dispatch THEN
      put("Dispatch IW(",iw_index);
      put("/",icount);
      puthex("): op ",iw(iw_index).opcode);
      puthex(" addr ",iw(iw_index).address);
      putline(" RTE");
    END IF;

    IF debug THEN
      putline("type OP_rte. restore status");
    END IF;

    --- The RTE instruction is replaced by the new IW contents, so
    --- it doesn't need to be removed explicitly.

    --- restore internal counters and flags

    --- The queue counts are NOT updated, since they should always
    --- be accurate. They are only provided so that the handler can
    --- empty the R1 Queue and Branch Queues. Loading these queues
    --- is visible by the dispatch unit, since it is done with
    --- normal instructions.

    pc        := cr(ICR_Next_PC);
    iw_bottom := cr(ICR_IW_Count);
    IF iw_bottom > IW_tap THEN
      iw_bottom := IW_tap;
    END IF;

    IF debug THEN
      putline("  IW_Count = ",iw_bottom);
    END IF;

    single_flag  := FALSE;          -- RTE set it, so clear it
    wat_flag     := FALSE;          -- RTE set it, so clear it
    tag_value    := 0;

    --- out_count should be zero, since RTE is WAT and SINGLE

    --- The tag_scoreboard should already be clear, since we've now
    --- heard from every outstanding instruction.  The register
    --- scoreboard may not be clear, but it can take care of
    --- itself.  Since we clear the destination bit for faulty
    --- instructions, it will eventually clear.

    --- Reload IW with all non-issued instructions. Note that
    --- although this sets the single_flag appropriately, it does
    --- NOT insert implicit DOIT instructions. If the IW was saved
    --- from a previous thread, it should already have the DOIT in
    --- there. Otherwise, if you want a DOIT, you have to put it in
    --- yourself via software. Be careful, though. Because DOITs
    --- are SINGLE instructions, having more than one in the IW at
    --- a time can cause a LOT of confusion.

    FOR k IN 0 TO iw_bottom - 1 LOOP
      iw(k).valid := TRUE;
      iw(k).address := siw(IW_CRNUM*k+IF_SIW_Min+IW_CR_ADDR);
      iw(k).opcode := siw(IW_CRNUM*k+IF_SIW_Min+IW_CR_OPCD);
      iw(k).wat := FALSE;
      iw(k).single := FALSE;
      tmpv := int2v1d( iw(k).opcode ); -- IDIOTIC ViewLogic crap
      IF is_single( tmpv ) THEN
        iw(k).single := TRUE;
        single_flag := TRUE;
      END IF;
      iw(k).issued := FALSE;
      iw(k).status := STATUS_none;

      IF debug THEN
        putline("IW slot ",k);
        put("  valid = ",iw(k).valid);
        put("  issued = ",iw(k).issued);
        putline("  tag = ",iw(k).tag);
        puthexline("  status = ",iw(k).status);
        puthexline("  address = ",iw(k).address);
        puthexline("  opcode = ",iw(k).opcode);
      END IF;
    END LOOP;

    FOR k IN iw_bottom TO IW_Max - 1 LOOP
      iw(k).valid := FALSE;
    END LOOP;

    --- Now replace CR0 with the Saved Control value

    cr0 := int2v1d( cr(ICR_Saved_Control) );
    x_cr0 <= cr0;

    IF cr0(Doit) = '1' THEN
      single_flag := TRUE;
    END IF;

    IF debug THEN
      puthexline("  CR0 = ",cr(ICR_Saved_Control));
      puthexline("pc is now ",pc);
      putline("single_flag is now ",single_flag);
      tmpb1 := vlb2boo( cr0(Doit) ); -- stupid Viewlogic crap
```

```
      putline("doit_flag is now ",tmpb1);
    END IF;


    --- That's it. The RTE has been replaced by the new IW
    --- contents, and everything is ready to continue dispatching.
    --- The SINGLE flag has been cleared if appropriate.

    --- Now, I need to go back and try to dispatch all over again.
    --- Since the RTE was the only remaining instruction in the IW
    --- when I started, there's no external event to wake up the
    --- dispatch cycle again.  I'll have to generate one.

    try_dispatch := TRUE;           -- do it again
    done_dispatch := FALSE;         -- don't count RTE
    EXIT;                           -- hop out of dispatch loop

    delay_to_use := delay_to_use + delay_rte; -- time to do RTE

    s_did_dispatch := s_did_dispatch + 1; -- count it


  ELSIF op_type = OP_sync THEN      -- sync instruction

    IF trace_dispatch THEN
      put("Dispatch IW(",iw_index);
      put("/",icount);
      puthex("): op ",iw(iw_index).opcode);
      puthex(" addr ",iw(iw_index).address);
      putline(" SYNC");
    END IF;

    --- This is complete, just by getting to the top. Remove it
    --- from the IW. Doesn't affect queue counts.  It takes
    --- essentially no extra time.

    tmpi := iw_index;
    iw_remove( tmpi, iw, iw_bottom, wat_flag, single_flag );

    IF debug THEN
      putline("type OP_sync. Done by default.");
      putline("wat_flag is now ",wat_flag);
      putline("single_flag is now ",single_flag);
    END IF;

    s_did_dispatch := s_did_dispatch + 1; -- count it


  ELSIF op_type = OP_sync_x THEN    -- sync.x instruction

    IF trace_dispatch THEN
      put("Dispatch IW(",iw_index);
      put("/",icount);
      puthex("): op ",iw(iw_index).opcode);
      puthex(" addr ",iw(iw_index).address);
      putline(" SYNC.X");
    END IF;

    --- Wait for external sync. Not interruptible.  Is that bad?
    --- Doesn't affect queue counts.

    IF debug THEN
      putline("type OP_sync_x");
    END IF;

    v_sync_req := NOT v_sync_req;
    sync_req <= v_sync_req AFTER delay_to_use;
    WAIT ON sync_ack, sync_nack;    -- might work, might not

    s_did_dispatch := s_did_dispatch + 1; -- either way it counts

    IF sync_ack'EVENT THEN          -- did it

      --- Done. Remove this instruction from the IW.

      tmpi := iw_index;
      iw_remove( tmpi, iw, iw_bottom, wat_flag, single_flag );

      IF debug THEN
        putline("done with SYNC.X");
        putline("wat_flag is now ",wat_flag);
        putline("single_flag is now ",single_flag);
      END IF;

      delay_to_use := delay_sync_ack;


    ELSE                            -- didn't sync

      IF debug THEN
        putline("SYNC.X failed.");
      END IF;

      iw(iw_index).status := STATUS_sync;
      faulting_instruction := iw(iw_index); -- remember who did it

      IF v_fault_pending = '0' THEN -- need to signal fault

        v_fault_pending := '1';     -- do so
        fault_pending <= v_fault_pending;
        v_x_debug := NOT v_x_debug;
        x_debug <= v_x_debug;

      END IF;

      delay_to_use := delay_sync_nack;

    END IF;
```

```
ELSE                             -- any other instruction

  IF debug THEN
    putline("type OP_other");
  END IF;


  --- Pick tag to assign to instruction. Note: It's possible to
  --- issue enough instructions that a tag would be reused before
  --- its first use has completed. For example, an instruction
  --- which uses R1 as a destination might take a VERY long time
  --- to get the result while a large number of much faster
  --- instructions complete. This could be a problem, unless some
  --- means of determining tag usage is provided. For the
  --- simulator, I'm just going to use a scoreboard to keep track
  --- of 8-bit tags, and if I can't find a free tag it means that
  --- I can't dispatch. In a real implementation I might use a
  --- different method (I really only need the same number of
  --- unique tags as there are slots in the IW, so a unary
  --- encoding would do).

  --- But wait! If the instruction doesn't report its DONE
  --- status, then I don't need to assign it a tag, because I'll
  --- remove it from the IW as soon as it goes out.

  --- I had to modify the tag meaning slightly. Originally, all
  --- branch instructions simply used the current value of
  --- cr0(Excp_Enable) to determine which Branch queue to place
  --- the target in.  However, branch instructions do not fault,
  --- so if an interrupt is noticed after the branch has been
  --- issued but before it has completed, the target goes down
  --- the wrong pipe.  Rather than add another bit to the
  --- existing FIFOs, I just modified the tag value so that bit 7
  --- is used to indicate the queue in use when the instruction
  --- was issued.  This doesn't affect the memory unit since it
  --- always has to report its status and the dispatch unit will
  --- wait for it to do so before handling exceptions.  Only the
  --- branch unit uses the new tag bit for anything.

  IF tellme  THEN                -- assign tags

    tag_value := tag_value + 1;
    IF tag_value > Tag_Max_Value THEN
      tag_value := 0;
    END IF;

    IF tag_sb(tag_value) THEN     -- this tag is still in use

      --- try to find one that's not being used

      FOR j IN 1 TO Tag_Max_Value LOOP
        tmpi := j + tag_value;    -- determine tag to try
        IF tmpi > Tag_Max_Value THEN
          tmpi := tmpi - Tag_Max_Value;
        END IF;

        IF NOT tag_sb(tmpi) THEN  -- if free, use it
          tag_value := tmpi;
          EXIT;
        END IF;

      END LOOP;

      --- if tag_sb(tag_value) is false, there are no free tags!

      ASSERT NOT tag_sb(tag_value) REPORT "Tag" SEVERITY FAILURE;

    END IF;                       -- finding tag

    tag_sb(tag_value) := TRUE;    -- mark tag in use

  END IF;                         -- assigning tag


  --- We got a tag, so let's dispatch the instruction.


  iw(iw_index).tag := tag_value;  -- only needed if tellme=TRUE
  iw(iw_index).issued := TRUE;    -- ditto

  IF op_type = OP_mvbr THEN       -- need branch target

    tmpmvbr := br;                -- grab branch target data
    v_br_ack := NOT v_br_ack;     -- eat branch target from queue
    br_ack <= v_br_ack;           -- ack branch data

    IF trace_dispatch THEN
      put("Dispatch IW(",iw_index);
      put("/",icount);
      puthex("): op ",iw(iw_index).opcode);
      puthex(" addr ",iw(iw_index).address);
      putline(" MVBR");
    END IF;

  ELSE

    IF trace_dispatch THEN
      put("Dispatch IW(",iw_index);
      put("/",icount);
      puthex("): op ",iw(iw_index).opcode);
      puthexline(" addr ",iw(iw_index).address);
    END IF;

  END IF;
```

```
IF ( ( debug OR debug_reuse ) AND
     reuse_result AND dest /= 0 AND dest /= 1 )
THEN
  put("unit ",fu_num);
  putline(" will latch register ",dest);
END IF;

--- The operand requests and scoreboard setting can go out
--- quickly, since this information was determined earlier when
--- we were checking for scoreboard for dispatch conditions.
--- The instruction information is what takes additional time.

IF dest >= 1 THEN               -- set scoreboard bit

  IF debug THEN
    putline("setting scoreboard bit ",dest);
  END IF;

  sb_set <= '1';
  tmpv := int2v1d( dest );      -- IDIOTIC ViewLogic crap
  sb_num <= tmpv(4 DOWNTO 0);
  v_sb_w_r := NOT v_sb_w_r;
  sb_w_r <= v_sb_w_r AFTER delay_to_use;
  WAIT ON sb_w_a;

END IF;

s_did_dispatch := s_did_dispatch + 1;

IF cr0(Excp_Enable) = '0' THEN  -- only count in exceptions
  s_excp_dispatch := s_excp_dispatch + 1;
END IF;


IF vA OR vB OR vD THEN          -- send operand request to RF

  s_need_operands := s_need_operands + 1;

  v_rf := rf_encode( vA, srcA, vB, srcB, vD, srcD );
  rf <= v_rf;
  v_rf_req := NOT v_rf_req;
  rf_req <= v_rf_req AFTER delay_to_use;

  --- I know it's ready, so I don't have to wait for the ACK

END IF;

IF wat THEN                     -- also wat_flag if applicable
  wat_flag := TRUE;
  IF debug THEN
    putline("wat_flag is now ",wat_flag);
  END IF;
END IF;
```

```
br_count := new_br_count;       -- update queue counts
xbr_count := new_xbr_count;
r1_count := new_r1_count;       -- this only decrements
r1_pending := new_r1_pending;   -- this only increments


IF debug THEN
  putline("br_count is now ",br_count);
  putline("xbr_count is now ",xbr_count);
  putline("r1_count is now ",r1_count);
  putline("r1_pending is now ",r1_pending);
END IF;


--- Send dispatch information

ex <= EX_PADDING;               -- clear all bits first

ex(90) <= boo2vlb( zeroA );

--- bits 89-87 are operand reuse bits

ex(89) <= boo2vlb( reuseD );
ex(88) <= boo2vlb( reuseB );
ex(87) <= boo2vlb( reuseA );

--- bit 86 is set if instruction should report DONE status
IF tellme THEN
  ex(86) <= '1';
  out_count := out_count + 1;   -- I might have to wait for it
  IF debug THEN
    putline("now out_count = ",out_count);
    putline("tag = ",iw(iw_index).tag);
  END IF;
ELSE
  ex(86) <= '0';
END IF;

--- bit 85 is set if operands are needed from the RF Unit
IF vA OR vB OR vD THEN
  ex(85) <= '1';
ELSE
  ex(85) <= '0';
END IF;

--- bit 76 is part of the tag field containing the current
--- value of cr0(Excp_Enable) but only for branch unit, so set
--- the default now.
ex(76) <= '0';

--- bits 84-77 are select lines for functional units
CASE op_type IS
WHEN OP_arith =>                -- unit 0
  ex(84 DOWNTO 77) <= ('0','0','0','0','0','0','0','1');
```

```
WHEN OP_logic =>                -- unit 1
  ex(84 DOWNTO 77) <= ('0','0','0','0','0','0','1','0');
WHEN OP_ctrl | OP_mvbr =>       -- unit 2
  ex(84 DOWNTO 77) <= ('0','0','0','0','0','1','0','0');
WHEN OP_load | OP_store | OP_xmem => -- unit 3
  ex(84 DOWNTO 77) <= ('0','0','0','0','1','0','0','0');
WHEN OP_branch | OP_ldbr =>     -- unit 4
  ex(84 DOWNTO 77) <= ('0','0','0','1','0','0','0','0');
  --- bit 76 is current value of cr0(Excp_Enable)
  ex(76) <= cr0(Excp_Enable);
WHEN OTHERS =>                  -- not implemented
  ex(84 DOWNTO 77) <= ('0','0','0','0','0','0','0','0');
END CASE;

--- bits 75-69 are for instruction tag
tmpv := int2v1d(tag_value);
ex(75 DOWNTO 69) <= tmpv(6 DOWNTO 0);

--- bits 68-64 is destination register (not always valid)
tmpv := int2v1d(dest);
ex(68 DOWNTO 64) <= tmpv(4 DOWNTO 0);

--- bits 63-32 are usually PC, but may hold Branch Queue data
IF op_type = OP_mvbr THEN
  ex(63 DOWNTO 32) <= tmpmvbr;
ELSE
  ex(63 DOWNTO 32) <= int2v1d( iw(iw_index).address );
END IF;

--- bits 31-0 are original opcode
ex(31 DOWNTO 0) <= int2v1d( iw(iw_index).opcode );

v_ex_req := NOT v_ex_req;
ex_req <= v_ex_req AFTER delay_to_use;

done_ex_dispatch := TRUE;

IF debug THEN
  putline("instruction dispatched");
END IF;

--- If the instruction isn't going to report its DONE status,
--- then I need to remove it from the IW now, because otherwise
--- it will never happen.

IF NOT tellme THEN              -- remove it

  tmpi := iw_index;
  iw_remove( tmpi, iw, iw_bottom, wat_flag, single_flag );

  IF debug THEN
    putline("instruction won't report -- removed from IW");
  END IF;
```

```
        END IF;

      END IF;                          -- any other instruction

      --- Now, I've dispatched one instruction.  I can't do more than
      --- one at a time, so I should quit the loop.  I don't really
      --- need the loop at all once I've found a potential instruction,
      --- but it's convenient as a way to abort the whole dispatch
      --- process.

      done_dispatch := TRUE;           -- done one

      EXIT;                            -- hop out of dispatch loop

    END LOOP;                          -- dispatching loop

  END IF;                              -- can I even try to dispatch

  IF debug THEN
    putline("done with this event");
  END IF;

END IF;                                -- dispatch_req



--- Now I should try to consume a DOIT, if there's one pending.
--- Obviously, I can't do this if faults are pending or I'm otherwise
--- prevented.  I don't have to do this in any particular order. This
--- isn't really a loop, it's just a convenient way to jump out if I
--- want to.

WHILE ( cr0(Doit) = '1' AND
        v_fault_pending = '0' )
LOOP


  --- Check for deadlock condition. If the Doit flag is set, no
  --- the IW is empty and nothing is in the Branch Queue, then I'm
  --- waiting for a branch that will never arrive.  Note that I won't
  --- notice deadlock at the same time that I notice external
  --- interrupts, since there is only one status register to indicate
  --- the cause of the exception.  However, when handling any other
  --- exception, this deadlock condition can be detected by software
  --- by checking for the same conditions.

  IF ( iw_bottom = 0 AND
       ( (cr0(Excp_Enable) = '1' AND br_count = 0) OR
         (cr0(Excp_Enable) = '0' AND xbr_count = 0) ) )
  THEN

    faulting_instruction := doit_instruction;
    faulting_instruction.status := STATUS_lock_b0;
```

```
            v_fault_pending := '1';         -- do so
            fault_pending <= v_fault_pending;
            v_x_debug := NOT v_x_debug;
            x_debug <= v_x_debug;
            IF trace_excp THEN
              putline("deadlock exception");
            END IF;

            EXIT;                               -- jump out

          END IF;


          IF cr0(Excp_Enable) = '1' THEN        -- use normal Branch Queue

            IF debug THEN
              putline("Probing BR queue");
            END IF;

----        v_br_p_req := NOT v_br_p_req;
----        br_p_req <= v_br_p_req AFTER delay;
----        WAIT ON br_p_ack;

            IF br_p = '0' THEN

              IF debug THEN
                putline("BR queue isn't ready");
              END IF;

              EXIT;

            END IF;

            IF br(1) = '1' THEN                 -- if condition code
              pc := 4 * v1d2int(br(31 DOWNTO 2)); -- take branch
            END IF;

            v_br_ack := NOT v_br_ack;           -- eat branch target from queue
            br_ack <= v_br_ack;                 -- ack branch data

            br_count := br_count - 1;

          ELSE                                  -- use X Branch Queue

            IF debug THEN
              putline("Probing XBR queue");
            END IF;

----        v_xbr_p_req := NOT v_xbr_p_req;
----        xbr_p_req <= v_xbr_p_req AFTER delay;
----        WAIT ON xbr_p_ack;

            IF xbr_p = '0' THEN
```

```
        IF debug THEN
          putline("XBR queue isn't ready");
        END IF;

        EXIT;

      END IF;

      IF xbr(1) = '1' THEN                -- if condition code
        pc := 4 * v1d2int(xbr(31 DOWNTO 2)); -- take branch
      END IF;

      v_xbr_ack := NOT v_xbr_ack; -- eat branch target from queue
      xbr_ack <= v_xbr_ack;               -- ack branch data

      xbr_count := xbr_count - 1;

    END IF;                               -- done consuming targets


    IF debug THEN
      puthexline("PC is now ",pc);
    END IF;

    --- Done. Clear the Doit and single flags, and let a new
    --- instruction be requested.

    cr0(Doit) := '0';
    single_flag := FALSE;

    IF debug THEN
      putline("Did a DOIT");
      putline("single_flag is now ",single_flag);
      putline("br_count is now ",br_count);
      putline("xbr_count is now ",xbr_count);
    END IF;

    IF trace_dispatch THEN
      put("Dispatch IW(-/-): ");
      puthex("op ",doit_instruction.opcode);
      puthex(" addr ",doit_instruction.address);
      putline(" DOIT");
    END IF;

    EXIT;                                 -- exit the loop now

  END LOOP;                               -- looking for DOITs


  --- Deal with faults. If a fault has been signaled, all outstanding
  --- instructions must either complete or fault. Deadlock cannot
  --- occur, since we prevent that by counting producer/consumer pairs
  --- as they are dispatched.  We don't have to lock the IW data
  --- structures, since we won't do anything until the entire chip is
```

```
--- quiescent, and therefore we are the only ones using the IW.

IF v_fault_pending = '1' THEN           -- do nothing until all done

  IF debug THEN
    putline("fault is pending. sit and wait");
    putline("out_count = ",out_count);
  END IF;

  --- Wait until we've heard from all outstanding instructions, then do
  --- interrupt handling stuff. I don't have to worry that all results
  --- have arrived at their destinations, provided I know that all
  --- instructions have completed. Anything that needs the results will
  --- just have to wait for them.  I do have to wait for any
  --- outstanding instruction fetches, since those have to be put in
  --- the IW.

  IF ( out_count = 0 AND NOT waiting_for_inst ) THEN

    s_did_excp := s_did_excp + 1;

    IF debug THEN
      putline("everything has stopped. save status");
    END IF;

    IF trace_excp THEN
      putline("iw_bottom = ",iw_bottom);
      putline("br_count = ",br_count);
      putline("r1_count = ",r1_count);
    END IF;

    --- clear IW, WAT, SINGLE, scoreboard, etc.

    tmpi := 0;
    FOR i IN 0 TO (IW_Max - 1) LOOP     -- save (internal) and clear IW

      IF iw(i).valid AND iw(i).status /= STATUS_none THEN
        tmpi := tmpi + 1;
      END IF;

      siw(IW_CRNUM*i+IF_SIW_Min+IW_CR_STAT) := iw(i).status;
      siw(IW_CRNUM*i+IF_SIW_Min+IW_CR_ADDR) := iw(i).address;
      siw(IW_CRNUM*i+IF_SIW_Min+IW_CR_OPCD) := iw(i).opcode;
      siw(IW_CRNUM*i+IF_SIW_Min+IW_CR_ARG1) := iw(i).arg1;
      siw(IW_CRNUM*i+IF_SIW_Min+IW_CR_ARG2) := iw(i).arg2;
      iw(i).valid := FALSE;

      IF debug THEN
        putline("IW slot ",i);
        put("  valid = ",iw(i).valid);
        put("  issued = ",iw(i).issued);
        putline("  tag = ",iw(i).tag);
        puthexline("  status = ",iw(i).status);
        puthexline("  address = ",iw(i).address);
        puthexline("  opcode = ",iw(i).opcode);
        puthexline("  arg1 = ",iw(i).arg1);
        puthexline("  arg2 = ",iw(i).arg2);
      END IF;

    END LOOP;

    --- Save exception status in (internal) control registers.
    --- The r1_count and br_count values are needed to know how to
    --- empty and save the R1 and Branch Queues.  This is done with
    --- normal instructions, so the counts do not need to be reloaded
    --- when the exception is over.

    cr(ICR_Saved_Control) := v1d2int(cr0);
    cr(ICR_Fault_Status)  := faulting_instruction.status;
    cr(ICR_Fault_Address) := faulting_instruction.address;
    cr(ICR_Next_PC)         := pc;
    cr(ICR_BR_Count)        := br_count;
    cr(ICR_R1_Count)        := r1_count;
    cr(ICR_IW_Count)        := iw_bottom;
    cr(ICR_IW_Faults)       := tmpi;

    IF debug THEN
      puthexline("  ICR_Saved_Control = ",cr(ICR_Saved_Control));
      puthexline("  ICR_Fault_Status = ",cr(ICR_Fault_Status));
      puthexline("  ICR_Fault_Address = ",cr(ICR_Fault_Address));
      puthexline("  ICR_Next_PC = ",cr(ICR_Next_PC));
      putline("  ICR_BR_Count = ",cr(ICR_BR_Count));
      putline("  ICR_R1_Count = ",cr(ICR_R1_Count));
      putline("  ICR_IW_Count = ",cr(ICR_IW_Count));
      putline("  ICR_IW_Faults = ",cr(ICR_IW_Faults));
    END IF;


    iw_bottom := 0;                     -- IW is now empty


    FOR i IN 0 TO Tag_Max_Value LOOP    -- clear the tag scoreboard
      tag_sb(i) := FALSE;
    END LOOP;

    wat_flag := FALSE;
    single_flag := FALSE;
    tag_value := 0;
    out_count := 0;

    --- turn off interrupts, exceptions, etc.
    --- Excp_Enable bit controls queue
    --- set supervisor mode.

    tmpv := ZERO(31 DOWNTO 0);          -- change CR0
    tmpv(Supervisor) := '1';
    cr0 := tmpv;
    x_cr0 <= cr0;
```

```
    --- clear fault_pending after interrupts are disabled

    v_fault_pending := '0';
    fault_pending <= v_fault_pending AFTER delay;


    --- If exceptions were already disabled, take error vector.

    tmpv := int2v1d( cr(ICR_Saved_Control) ); -- convert to vlbit
    IF tmpv(Excp_Enable) = '0' THEN
      faulting_instruction.status := STATUS_error;
    END IF;

    --- Set PC to interrupt handler address. Each exception vector
    --- has four status values available to it, so the status code
    --- can be truncated to determine the vector address.

    tmpv := int2v1d(faulting_instruction.status);
    tmpi := v1d2int(tmpv(15 DOWNTO 0)); -- get major status
    IF tmpi >= STATUS_trap THEN
      pc := 4 * v1d2int( tmpv( 7 DOWNTO 0) );
    ELSE
      pc := 4 * v1d2int( tmpv( 7 DOWNTO 2) );
    END IF;

    --- Now everything is ready to go. Let INST_REQ go out below.

    IF debug OR trace_dispatch THEN
      puthexline("faulting status is ",faulting_instruction.status);
      puthexline("ready for vector. PC is now ",pc);
    END IF;

    --- We can just sit and wait here, since nothing else should be
    --- going on anyway.

    WAIT FOR delay_save_state;          -- time to save state

  END IF;                               -- outstanding insts

END IF;                                 -- fault pending



--- Send out a request for more instructions to dispatch. If there's
--- a fault pending, we don't request any new instructions until the
--- PC has been set to the exception handler. Otherwise, if there's
--- room for another instruction in the IW, and the SINGLE flag isn't
--- set, and we haven't already, then request the next instruction.

IF v_fault_pending = '1' THEN           -- skip back to top of loop

  IF debug THEN
    putline("no inst reqs while faults are pending");
```

```
          END IF;

          NEXT;
        END IF;

        IF ((IW_tap - iw_bottom) >= 1 ) AND NOT single_flag THEN

          IF waiting_for_inst THEN              -- skip back to top of loop

            IF debug THEN
              putline("already requested an instruction");
            END IF;

            NEXT;
          END IF;

          v_i_addr := int2v1d(pc);              -- get 32-bit PC value
          i_addr <= (v_i_addr(31 DOWNTO 2) & cr0(Supervisor) & '0');
          v_inst_req := NOT v_inst_req;
          inst_req <= v_inst_req AFTER delay;

          IF debug THEN
            puthexline("sent out INST_REQ for address ",pc);
          END IF;

          waiting_for_inst := TRUE;

        END IF;

      END LOOP;                                 -- main process loop

    END IF;                                     -- CLR not asserted


    --- Wait for something new to happen. The four main events are arbitrated,
    --- so that data structures can be shared. The control register requests do
    --- not need arbitration, since control registers are only written when
    --- everything else is quiet. The other events just awaken the dispatch
    --- process.

    IF debug THEN
      putline("wait for something to happen");
    END IF;

    WAIT ON clr, inst_ack, done_req, external_int, getcr_req,
      putcr_req, br_p, xbr_p, rf_p, ex_p, sb;

  END PROCESS main;

END behavior;
```

2.15 logic_unit.vhd

```
USE work.p_constants.ALL;
USE work.p_output.ALL;

---- This unit handles the logic instructions. Currently, there is only
---- one of these units, and it is very simple. It could be expanded to do
---- split transactions, reordering memory operations, etc.


---- The EXTRA input was added to provide last-result-reuse and to elminate
---- zero values of op_a for immediate instructions. The bit usage is:
----
---- EXTRA(0) = reuseA
---- EXTRA(1) = reuseB
---- EXTRA(2) = reuseD
---- EXTRA(3) = zeroA (unused)

ENTITY logic_unit IS
  GENERIC( delay: TIME := 100 ps );
  PORT( SIGNAL clr             : IN  VLBIT;
        SIGNAL rin             : IN  VLBIT;
        SIGNAL ain             : OUT VLBIT;
        SIGNAL tellme          : IN  VLBIT;
        SIGNAL tag             : IN  VLBIT_1D (7 DOWNTO 0);
        SIGNAL dest            : IN  VLBIT_1D (4 DOWNTO 0);
        SIGNAL opcode          : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL op_a            : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL op_b            : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL d_req           : OUT VLBIT;
        SIGNAL d_ack           : IN  VLBIT;
        SIGNAL done            : OUT VLBIT_1D (87 DOWNTO 0);
        SIGNAL rout            : OUT VLBIT;
        SIGNAL aout            : IN  VLBIT;
        SIGNAL res             : OUT VLBIT_1D (36 DOWNTO 0);
        SIGNAL rq_req          : OUT VLBIT;
        SIGNAL rq_ack          : IN  VLBIT;
        SIGNAL rq              : OUT VLBIT_1D (31 DOWNTO 0);
        SIGNAL extra           : IN  VLBIT_1D (3 DOWNTO 0) );
END logic_unit;


ARCHITECTURE behavior OF logic_unit IS
BEGIN
  main : PROCESS

    VARIABLE debug             : BOOLEAN := FALSE;

    --- factors affecting response time

    VARIABLE delay_boolean     : TIME := 100 ps; -- boolean operations
    VARIABLE delay_barrel      : TIME := 100 ps; -- need barrel shifter
    VARIABLE delay_ff          : TIME := 100 ps; -- find first bit stuff
    VARIABLE delay_to_use       : TIME;         -- scratch for culumative time


    --- output variables

    VARIABLE v_rout             : VLBIT;
    VARIABLE v_rq_req           : VLBIT;
    VARIABLE v_d_req            : VLBIT;


    --- internal variables

    VARIABLE index              : INTEGER;

    VARIABLE Av                 : VLBIT_1D (31 DOWNTO 0);
    VARIABLE Bv                 : VLBIT_1D (31 DOWNTO 0);
    VARIABLE Wi                 : INTEGER;
    VARIABLE Oi                 : INTEGER;
    VARIABLE result             : VLBIT_1D (31 DOWNTO 0);
    VARIABLE lastresult         : VLBIT_1D (31 DOWNTO 0);


    --- scratch variables

    VARIABLE tmpv               : VLBIT_1D (31 DOWNTO 0);
    VARIABLE tmpb               : VLBIT;
    VARIABLE tmpi               : INTEGER;

    --- statistics

    VARIABLE s_logic_rin        : INTEGER;
    VARIABLE s_barrel           : INTEGER;
    VARIABLE s_ff               : INTEGER;


  BEGIN


    --- If clear is asserted, reset everything and stick here.

    IF clr /= '1' THEN                          -- reset to beginning

      v_rout := '0';
      v_rq_req := '0';
      v_d_req := '0';

      ain <= '0';
      d_req <= v_d_req;
      done <= ZERO(87 DOWNTO 0);
      rout <= v_rout;
      res <= ZERO(36 DOWNTO 0);
      rq_req <= v_rq_req;
      rq <= ZERO(31 DOWNTO 0);
```

```
      s_logic_rin := 0;
      s_barrel := 0;
      s_ff := 0;

      WAIT UNTIL clr = '1';


    ELSE                                        -- only wakeup event is RIN


      --- This stuff had to be encoded by hand. I could have written a Perl
      --- script to do it, but it would have taken longer than doing it
      --- manually. This stuff shouldn't change that much (I hope). I also made
      --- some assumptions about unused bits, which is probably okay.


      --- I don't bother to compute anything if I'm just going to throw the
      --- result away.

      IF dest /= REG_R0 THEN                    -- only if worth doing

        s_logic_rin := s_logic_rin + 1;

        delay_to_use := delay_boolean;          -- assume simple case

        index := op2index( opcode );            -- identify instruction

        --- determine operands

        IF extra(0)='1' THEN
          Av := lastresult;
        ELSE
          Av := op_a;
        END IF;

        IF index <= 31 OR (index >= 72 AND index <= 77) THEN -- immediate
          --- All Immediate operands are zero-extended (this could change).
          Bv := ZERO(31 DOWNTO 16) & opcode(15 DOWNTO 0);
        ELSE                                    -- triadic
          IF extra(1)='1' THEN
            Bv := lastresult;
          ELSE
            Bv := op_b;
          END IF;
        END IF;


        IF debug THEN
          tmpi := v1d2int(Av);
          puthexline("op_a is ",tmpi);
          tmpi := v1d2int(Bv);
          puthexline("op_b is ",tmpi);
        END IF;
```

```
--- get W and O fields just in case.

Wi := v1d2int( Bv(9 DOWNTO 5) );
IF Wi = 0 THEN
  Wi := 32;
END IF;
Oi := v1d2int( Bv(4 DOWNTO 0) );


--- now, do the operation

CASE index IS

WHEN 0 =>                               -- and rd,ra,imm16

  tmpv := Av AND Bv;
  result := Av(31 DOWNTO 16) & tmpv(15 DOWNTO 0);

WHEN 1 =>                               -- and.u rd,ra,imm16

  Bv := Bv(15 DOWNTO 0) & Bv(15 DOWNTO 0);
  tmpv := Av AND Bv;
  result := tmpv(31 DOWNTO 16) & Av(15 DOWNTO 0);

WHEN 2 =>                               -- mask rd,ra,imm16

  tmpv := Av AND Bv;
  result := ZERO(31 DOWNTO 16) & tmpv(15 DOWNTO 0);

WHEN 3 =>                               -- mask.u rd,ra,imm16

  Bv := Bv(15 DOWNTO 0) & Bv(15 DOWNTO 0);
  tmpv := Av AND Bv;
  result := tmpv(31 DOWNTO 16) & ZERO(15 DOWNTO 0);

WHEN 4 =>                               -- or rd,ra,imm16

  tmpv := Av OR Bv;
  result := Av(31 DOWNTO 16) & tmpv(15 DOWNTO 0);

WHEN 5 =>                               -- or.u rd,ra,imm16

  Bv := Bv(15 DOWNTO 0) & Bv(15 DOWNTO 0);
  tmpv := Av OR Bv;
  result := tmpv(31 DOWNTO 16) & Av(15 DOWNTO 0);

WHEN 6 =>                               -- xor rd,ra,imm16

  tmpv := Av XOR Bv;
  result := Av(31 DOWNTO 16) & tmpv(15 DOWNTO 0);

WHEN 7 =>                               -- xor.u rd,ra,imm16
```

```
          Bv := Bv(15 DOWNTO 0) & Bv(15 DOWNTO 0);
          tmpv := Av XOR Bv;
          result := tmpv(31 DOWNTO 16) & Av(15 DOWNTO 0);

        WHEN 32 =>                            -- and rd,ra,rb

          result := Av AND Bv;

        WHEN 33 =>                            -- and.c rd,ra,rb

          result := Av AND NOT Bv;

        WHEN 36 =>                            -- or rd,ra,rb

          result := Av OR Bv;

        WHEN 37 =>                            -- or.c rd,ra,rb

          result := Av OR NOT Bv;

        WHEN 38 =>                            -- xor rd,ra,rb

          result := Av XOR Bv;

        WHEN 39 =>                            -- xor.c rd,ra,rb

          result := Av XOR NOT Bv;

        WHEN 64 | 72 =>                       -- clr

          s_barrel := s_barrel + 1;

          delay_to_use := delay_barrel;

          FOR k IN 0 TO 31 LOOP
            IF k >= Oi AND k < (Oi + Wi) THEN   -- in range
              result(k) := '0';
            ELSE
              result(k) := Av(k);
            END IF;
          END LOOP;

        WHEN 65 | 73 =>                       -- set

          s_barrel := s_barrel + 1;

          delay_to_use := delay_barrel;

          FOR k IN 0 TO 31 LOOP
            IF k >= Oi AND k < (Oi + Wi) THEN   -- in range
              result(k) := '1';
            ELSE
              result(k) := Av(k);
            END IF;
```

```
    END LOOP;

  WHEN 66 | 74 =>                         -- ext

    s_barrel := s_barrel + 1;

    delay_to_use := delay_barrel;

    tmpi := Wi + Oi - 1;
    IF tmpi > 31 THEN                     -- determine sign bit
      tmpb := Av(31);
    ELSE
      tmpb := Av(tmpi);
    END IF;

    FOR k IN 0 TO 31 LOOP
      IF k < Wi AND k+Oi <= 31 THEN
        result(k) := Av(k+Oi);
      ELSE
        result(k) := tmpb;                -- copy sign bit
      END IF;
    END LOOP;

  WHEN 67 | 75 =>                         -- extu

    s_barrel := s_barrel + 1;

    delay_to_use := delay_barrel;

    FOR k IN 0 TO 31 LOOP
      IF k < Wi AND k+Oi <= 31 THEN
        result(k) := Av(k+Oi);
      ELSE
        result(k) := '0';
      END IF;
    END LOOP;

  WHEN 68 | 76 =>                         -- mak

    s_barrel := s_barrel + 1;

    delay_to_use := delay_barrel;

    FOR k IN 0 TO 31 LOOP
      IF k >= Oi AND k < (Oi + Wi) THEN   -- in range
        result(k) := Av(k-Oi);
      ELSE
        result(k) := '0';
      END IF;
    END LOOP;

  WHEN 69 | 77 =>                         -- rot

    s_barrel := s_barrel + 1;
    delay_to_use := delay_barrel;

    FOR k IN 0 TO 31 LOOP
      IF k+Oi <= 31 THEN
        result(k) := Av(k+Oi);
      ELSE
        result(k) := Av(K+Oi-32);
      END IF;
    END LOOP;

  WHEN 70 =>                              -- ff0 rd,rb

    s_ff := s_ff + 1;

    delay_to_use := delay_ff;

    tmpi := 32;
    FOR k IN 31 DOWNTO 0 LOOP
      IF Bv(k) = '0' THEN
        tmpi := k;
        EXIT;
      END IF;
    END LOOP;
    result := int2v1d(tmpi);

  WHEN 71 =>                              -- ff1 rd,rb

    s_ff := s_ff + 1;

    delay_to_use := delay_ff;

    tmpi := 32;
    FOR k IN 31 DOWNTO 0 LOOP
      IF Bv(k) = '1' THEN
        tmpi := k;
        EXIT;
      END IF;
    END LOOP;
    result := int2v1d(tmpi);

  WHEN OTHERS =>                          -- anything else
    putline("*** Illegal Opcode for Logic Unit ***");
  END CASE;



  --- write output

  IF dest = REG_R1 THEN

    rq <= result;
    v_rq_req := NOT v_rq_req;
    rq_req <= v_rq_req AFTER delay_to_use;
```

```
          WAIT ON rq_ack;

        ELSE

          res <= dest & result;

          lastresult := result;

          IF debug THEN
            tmpi := v1d2int(lastresult);
            puthexline("latched ",tmpi);
          END IF;

          v_rout := NOT v_rout;                 -- write it out
          rout <= v_rout AFTER delay_to_use;

          WAIT ON aout;

        END IF;


      END IF;                                   -- produce a result


      --- signal completion of this instruction

      IF tellme = '1' THEN                      -- only if needed

        tmpv := int2v1d( STATUS_done );
        done <= ZERO(87 DOWNTO 24) & tmpv(15 DOWNTO 0) & tag;

        v_d_req := NOT v_d_req;
        d_req <= v_d_req AFTER delay;

        WAIT ON d_ack;

      END IF;

      ain <= rin AFTER delay;                   -- done with this event


    END IF;                                     -- clr not asserted


    WAIT ON clr, rin;                           -- wait for next event

  END PROCESS main;

END behavior;
```

2.16 memory_unit.vhd

```
USE work.p_constants.ALL;
USE work.p_memory_unit.ALL;
USE work.p_output.ALL;

---- The memory_unit module provides the interface to the external data memory.


---- The EXTRA input was added to provide last-result-reuse and to elminate
---- zero values of op_a for immediate instructions. The bit usage is:
----
---- EXTRA(0) = reuseA
---- EXTRA(1) = reuseB
---- EXTRA(2) = reuseD
---- EXTRA(3) = zeroA (unused)

ENTITY memory_unit IS
  GENERIC( delay: TIME := 100 ps );
  PORT( SIGNAL clr              : IN  VLBIT;
        SIGNAL rin              : IN  VLBIT;
        SIGNAL ain              : OUT VLBIT;
        SIGNAL tellme           : IN  VLBIT;
        SIGNAL tag              : IN  VLBIT_1D (7 DOWNTO 0);
        SIGNAL dest             : IN  VLBIT_1D (4 DOWNTO 0);
        SIGNAL opcode           : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL op_s             : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL op_a             : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL op_b             : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL data_req         : OUT VLBIT;
        SIGNAL data_ack         : IN  VLBIT;
        SIGNAL data_ack_ack     : OUT VLBIT;
        SIGNAL read             : OUT VLBIT;
        SIGNAL byte_strobe      : OUT VLBIT_1D (3 DOWNTO 0);
        SIGNAL d_addr           : OUT VLBIT_1D (31 DOWNTO 0);
        SIGNAL d_read           : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL d_status         : IN  VLBIT_1D (3 DOWNTO 0);
        SIGNAL d_write          : OUT VLBIT_1D (31 DOWNTO 0);
        SIGNAL cr0              : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL d_req            : OUT VLBIT;
        SIGNAL d_ack            : IN  VLBIT;
        SIGNAL done             : OUT VLBIT_1D (87 DOWNTO 0);
        SIGNAL reg_q_req        : OUT VLBIT;
        SIGNAL reg_q_ack        : IN  VLBIT;
        SIGNAL reg_q            : OUT VLBIT_1D (36 DOWNTO 0);
        SIGNAL r1_q_req         : OUT VLBIT;
        SIGNAL r1_q_ack         : IN  VLBIT;
        SIGNAL r1_q             : OUT VLBIT_1D (31 DOWNTO 0);
        SIGNAL extra            : IN  VLBIT_1D (3 DOWNTO 0) );
END memory_unit;


ARCHITECTURE behavior OF memory_unit IS
BEGIN
```

```
main : PROCESS

  VARIABLE debug              : BOOLEAN := FALSE;

  --- Minor status code values.  The minor field is actually
  --- "operation + opsize".  Opsize is currently only 1, 2, or 4.
  --- I'm leaving room for size 8.

  CONSTANT MINOR_load         : INTEGER := 16;
  CONSTANT MINOR_store        : INTEGER := 32;
  CONSTANT MINOR_xmem         : INTEGER := MINOR_load + MINOR_store;


  --- factors affecting response time

  VARIABLE delay_decode       : TIME := 100 ps; -- decode & compute address

  --- internal variables

  VARIABLE lastresult         : VLBIT_1D(31 DOWNTO 0);
  VARIABLE op_aa              : VLBIT_1D(31 DOWNTO 0);
  VARIABLE op_bb              : VLBIT_1D(31 DOWNTO 0);
  VARIABLE op_ss              : VLBIT_1D(31 DOWNTO 0);

  VARIABLE opA                : INTEGER;
  VARIABLE opB                : INTEGER;
  VARIABLE addr               : INTEGER;

  VARIABLE imm                : BOOLEAN;
  VARIABLE opsize             : INTEGER;
  VARIABLE scaled             : BOOLEAN;
  VARIABLE do_sign_ext        : BOOLEAN;
  VARIABLE force_usr_mode     : BOOLEAN;
  VARIABLE do_load            : BOOLEAN;
  VARIABLE do_store           : BOOLEAN;

  VARIABLE read_result        : VLBIT_1D (31 DOWNTO 0);
  VARIABLE status             : INTEGER;
  VARIABLE minor              : INTEGER;
  VARIABLE arg1               : INTEGER;
  VARIABLE arg2               : INTEGER;

  --- output signal value variables

  VARIABLE v_byte_strobe      : VLBIT_1D (3 DOWNTO 0);
  VARIABLE v_d_addr           : VLBIT_1D (31 DOWNTO 0);
  VARIABLE v_data_req         : VLBIT;
  VARIABLE v_d_req            : VLBIT;
  VARIABLE v_r1_q_req         : VLBIT;
  VARIABLE v_reg_q_req        : VLBIT;

  --- scratch variables

  VARIABLE tmpi               : INTEGER;
```

```
    VARIABLE tmpv                 : VLBIT_1D (31 DOWNTO 0);
    VARIABLE DoTheLoop            : BOOLEAN;

    --- statistics

    VARIABLE s_memory_rin         : INTEGER;
    VARIABLE s_lda                : INTEGER;
    VARIABLE s_load_word          : INTEGER;
    VARIABLE s_load_half          : INTEGER;
    VARIABLE s_load_byte          : INTEGER;
    VARIABLE s_store_word         : INTEGER;
    VARIABLE s_store_half         : INTEGER;
    VARIABLE s_store_byte         : INTEGER;


  BEGIN


    --- If clear is asserted, reset everything and stick here.

    IF clr /= '1' THEN                          -- reset to beginning

      v_byte_strobe := ('0','0','0','0');
      v_d_addr := ZERO(31 DOWNTO 0);
      v_data_req := '0';
      v_d_req := '0';
      v_r1_q_req := '0';
      v_reg_q_req := '0';

      ain <= '0';
      byte_strobe <= v_byte_strobe;
      d_addr <= v_d_addr;
      d_write <= ZERO(31 DOWNTO 0);
      data_req <= v_data_req;
      data_ack_ack <= '0';
      d_req <= v_d_req;
      done <= ZERO(87 DOWNTO 0);
      r1_q <= ZERO(31 DOWNTO 0);
      r1_q_req <= v_r1_q_req;
      read <= '0';
      reg_q <= ZERO(36 DOWNTO 0);
      reg_q_req <= v_reg_q_req;

      s_memory_rin := 0;
      s_lda := 0;
      s_load_word := 0;
      s_load_half := 0;
      s_load_byte := 0;
      s_store_word := 0;
      s_store_half := 0;
      s_store_byte := 0;


      WAIT UNTIL clr = '1';
```

```
ELSE                                        -- only wakeup event is RIN

  s_memory_rin := s_memory_rin + 1;


  --- Determine what operands to use.

  IF extra(0)='1' THEN
    op_aa := lastresult;
  ELSE
    op_aa := op_a;
  END IF;

  IF extra(1)='1' THEN
    op_bb := lastresult;
  ELSE
    op_bb := op_b;
  END IF;

  IF extra(2)='1' THEN
    op_ss := lastresult;
  ELSE
    op_ss := op_s;
  END IF;

    IF debug THEN
      puthexline("op_a is ",tmpi);
      tmpi := v1d2int(op_bb);
      puthexline("op_b is ",tmpi);
      tmpi := v1d2int(op_ss);
      puthexline("op_s is ",tmpi);
    END IF;


  --- There are number of sequential steps, which I need to abort if I
  --- get a fault at any point. This loop lets me jump immediately to
  --- the end of the sequence when I detect a fault. It doesn't really
  --- loop.

  DoTheLoop := TRUE;                        -- do it
  WHILE DoTheLoop LOOP                      -- here it is
    DoTheLoop := FALSE;                     -- no, don't do it


    status := STATUS_done;                  -- assume it will work
    arg1 := 0;
    arg2 := 0;

    --- Determine what to do.

    mem_decode( opcode,
                imm,                        -- boolean
```

```
            opsize,                     -- integer
            scaled,                     -- boolean
            do_sign_ext,                -- boolean
            force_usr_mode,             -- boolean
            do_load,                    -- boolean
            do_store );                 -- boolean


--- Compute address

opA := v1d2int( op_aa );

IF imm THEN                             -- use imm16 value
  opB := v1d2int( opcode( 15 DOWNTO 0 ) );
ELSE                                    -- use register value
  opB := v1d2int( op_bb );
  IF scaled THEN                        -- scale factor
    opB := opB * opsize;
  END IF;
END IF;

addr := opA + opB;                      -- add 'em together



--- Model this computation time. I can just wait, because there is only
--- one possible wake-up event, and it isn't in parallel with anything.

WAIT FOR delay_decode;



--- deal with lda instructions as a special case

IF ( NOT do_load ) AND ( NOT do_store ) THEN -- lda instruction

  s_lda := s_lda + 1;

  read_result := int2v1d( addr );

  IF dest = REG_R1 THEN                 -- dest is R1 Queue

    r1_q <= read_result;

    v_r1_q_req := NOT v_r1_q_req;
    r1_q_req <= v_r1_q_req AFTER delay;

    WAIT ON r1_q_ack;

  ELSIF dest /= REG_R0 THEN             -- dest is not R0

    reg_q <= dest & read_result;

    lastresult := read_result;
```

```
        IF debug THEN
          tmpi := v1d2int(lastresult);
          puthexline("latched ",tmpi);
        END IF;

        v_reg_q_req := NOT v_reg_q_req;
        reg_q_req <= v_reg_q_req AFTER delay;

        WAIT ON reg_q_ack;

      END IF;

      EXIT;                                 -- jump out here

    END IF;                                 -- lda instruction


    --- make sure address alignment is okay for operation size

    IF (addr MOD opsize) /= 0 THEN          -- misalignment

      status := STATUS_align;               -- that's a fault
      IF do_load AND do_store THEN
        minor := MINOR_xmem + opsize;
      ELSIF do_load THEN
        minor := MINOR_load + opsize;
      ELSE
        minor := MINOR_store + opsize;
      END IF;
      arg1 := addr;
      arg2 := v1d2int( op_ss );             -- just in case

      EXIT;                                 -- jump to end

    END IF;


    --- set supervisor bit as desired

    tmpv := int2v1d( addr );
    IF force_usr_mode THEN
      v_d_addr := (tmpv(31 DOWNTO 2) & '0' & '0');
    ELSE
      v_d_addr := (tmpv(31 DOWNTO 2) & cr0(Supervisor) & '0');
    END IF;


    --- determine byte strobe values

    IF opsize = 4 THEN                      -- word
      v_byte_strobe := ('1','1','1','1');
    ELSIF opsize = 2 THEN                   -- half-word
      IF (addr MOD 4) = 0 THEN              -- lower half
        v_byte_strobe := ('0','0','1','1');
      ELSE                                  -- upper half
        v_byte_strobe := ('1','1','0','0');
      END IF;
    ELSE                                    -- byte
      tmpi := addr MOD 4;
      IF tmpi = 0 THEN                      -- byte 0
        v_byte_strobe := ('0','0','0','1');
      ELSIF tmpi = 1 THEN                   -- byte 1
        v_byte_strobe := ('0','0','1','0');
      ELSIF tmpi = 2 THEN                   -- byte 2
        v_byte_strobe := ('0','1','0','0');
      ELSE                                  -- byte 3
        v_byte_strobe := ('1','0','0','0');
      END IF;
    END IF;


    --- Do read operation if requested

    IF do_load THEN

      IF opsize = 4 THEN
        s_load_word := s_load_word + 1;
      ELSIF opsize = 2 THEN
        s_load_half := s_load_half + 1;
      ELSE
        s_load_byte := s_load_byte + 1;
      END IF;

      read <= '1';
      byte_strobe <= v_byte_strobe;
      d_addr <= v_d_addr;

      v_data_req := NOT v_data_req;
      data_req <= v_data_req AFTER delay;   -- send read request

      WAIT ON data_ack;                     -- wait for response


      --- check for memory fault

      IF ( ( d_status(0) = '1' AND cr0(Dmem0) = '1' ) OR
           ( d_status(1) = '1' AND cr0(Dmem1) = '1' ) OR
           ( d_status(2) = '1' AND cr0(Dmem2) = '1' ) OR
           ( d_status(3) = '1' AND cr0(Dmem3) = '1' ) )
      THEN
        status := STATUS_dmem;
      ELSE

        --- sign-extend or zero-extend the result according to size

        read_result := d_read;              -- get result
        IF opsize = 2 THEN

          IF do_sign_ext AND d_read(15) = '1' THEN
            read_result(31 DOWNTO 16) := ('1','1','1','1','1','1','1','1',
                                          '1','1','1','1','1','1','1','1');
          ELSE
            read_result(31 DOWNTO 16) := ZERO(31 DOWNTO 16);
          END IF;

        ELSIF opsize = 1 THEN

          IF do_sign_ext AND d_read(7) = '1' THEN
            read_result(31 DOWNTO 8) := ('1','1','1','1','1','1','1','1',
                                         '1','1','1','1','1','1','1','1',
                                         '1','1','1','1','1','1','1','1');
          ELSE
            read_result(31 DOWNTO 8) := ZERO(31 DOWNTO 8);
          END IF;

        END IF;

      END IF;


      --- now acknowledge receipt of status and data

      data_ack_ack <= data_ack AFTER delay;


      IF status /= STATUS_done THEN         -- found a fault

        IF do_store THEN                    -- this is xmem instruction
          minor := MINOR_xmem + opsize;
        ELSE
          minor := MINOR_load + opsize;
        END IF;
        arg1 := addr;
        arg2 := v1d2int( op_ss );

        EXIT;                               -- jump to end

      END IF;

    END IF;                                 -- of do_load



    --- Do write operation if requested

    IF do_store THEN

      IF opsize = 4 THEN
        s_store_word := s_store_word + 1;
      ELSIF opsize = 2 THEN
        s_store_half := s_store_half + 1;
      ELSE
        s_store_byte := s_store_byte + 1;
      END IF;

      read <= '0';
      byte_strobe <= v_byte_strobe;
      d_addr <= v_d_addr;

      --- align and shift the output data correctly

      IF opsize = 4 THEN                    -- full word at once
        d_write <= op_ss;
      ELSIF opsize = 2 THEN                 -- half word
        d_write( 31 DOWNTO 16 ) <= op_ss( 15 DOWNTO 0);
        d_write( 15 DOWNTO 0 ) <= op_ss( 15 DOWNTO 0);
      ELSE                                  -- byte
        d_write( 31 DOWNTO 24 ) <= op_ss( 7 DOWNTO 0);
        d_write( 23 DOWNTO 16 ) <= op_ss( 7 DOWNTO 0);
        d_write( 15 DOWNTO 8 ) <= op_ss( 7 DOWNTO 0);
        d_write( 7 DOWNTO 0 ) <= op_ss( 7 DOWNTO 0);
      END IF;


      v_data_req := NOT v_data_req;
      data_req <= v_data_req AFTER delay;   -- send read request

      WAIT ON data_ack;                     -- wait for response


      --- check for memory fault

      IF ( ( d_status(0) = '1' AND cr0(Dmem0) = '1' ) OR
           ( d_status(1) = '1' AND cr0(Dmem1) = '1' ) OR
           ( d_status(2) = '1' AND cr0(Dmem2) = '1' ) OR
           ( d_status(3) = '1' AND cr0(Dmem3) = '1' ) )
      THEN
        status := STATUS_dmem;
      END IF;


      --- now acknowledge receipt of status

      data_ack_ack <= data_ack AFTER delay;


      IF status /= STATUS_done THEN         -- found a fault

        IF do_load THEN                     -- this is xmem instruction
          minor := MINOR_xmem + opsize;
        ELSE
          minor := MINOR_store + opsize;
        END IF;
        arg1 := addr;
```

```
        arg2 := v1d2int( op_ss );

        EXIT;                            -- jump to end

      END IF;

    END IF;                              -- of do_store


    --- Everything was successful. Put any read result out the queue.


    IF do_load THEN                      -- if load operation

      IF dest = REG_R1 THEN              -- dest is R1 Queue

        r1_q <= read_result;

        v_r1_q_req := NOT v_r1_q_req;
        r1_q_req <= v_r1_q_req AFTER delay;

        WAIT ON r1_q_ack;

      ELSIF dest /= REG_R0 THEN          -- dest is not R0

        reg_q <= dest & read_result;

        lastresult := read_result;

        IF debug THEN
          tmpi := v1d2int(lastresult);
          puthexline("latched ",tmpi);
        END IF;

        v_reg_q_req := NOT v_reg_q_req;
        reg_q_req <= v_reg_q_req AFTER delay;

        WAIT ON reg_q_ack;

      END IF;

    END IF;


  END LOOP;                              -- here's the jump-out point


  --- Send completion status.

  IF tellme = '1' THEN                   -- only if needed

    tmpi := status + 256 * minor;
    tmpv := int2v1d( tmpi );
    done <= int2v1d(arg1) & int2v1d(arg2) & tmpv(15 DOWNTO 0) & tag;
```

```
        v_d_req := NOT v_d_req;
        d_req <= v_d_req AFTER delay;

        WAIT ON d_ack;

      END IF;

      ain <= rin AFTER delay;                     -- done with this event

    END IF;                                       -- clr not asserted

    WAIT ON clr, rin;                             -- wait for next event

  END PROCESS main;

END behavior;
```

2.17 p_arith.vhd

```
PACKAGE p_arith IS

  --- Stupid ViewLogic VHDL doesn't implement addum correctly for 32 bits. I
  --- have to write my own. Bunch of idiots.  Not only that, but it's
  --- impossible to make these functions work generically for any vector
  --- length, because you can't put expressions in the range of variable
  --- declarations.

  FUNCTION add32( Av, Bv : VLBIT_1D; Cin : VLBIT ) RETURN VLBIT_1D;
  FUNCTION sub32( Av, Bv : VLBIT_1D; Cin : VLBIT ) RETURN VLBIT_1D;

END p_arith;



PACKAGE BODY p_arith IS

  --- Compute sum of two 32-bit vectors (A + B). The result is two bits longer
  --- than the input vectors. Bit 33 of the result is the overflow bit, bit 32
  --- is the carry out.

  FUNCTION add32( Av, Bv : VLBIT_1D; Cin : VLBIT )
  RETURN VLBIT_1D IS
    VARIABLE res : VLBIT_1D( 33 DOWNTO 0 );
    VARIABLE carry : VLBIT;
  BEGIN

    carry := Cin;

    FOR k IN 0 TO 31 LOOP

      res(k) := Av(k) XOR Bv(k) XOR carry;
      carry := (Av(k) AND Bv(k)) OR (carry AND (Av(k) OR Bv(k)));

    END LOOP;

    res(32) := carry;                           -- carry out
    --- overflow bit
    res(33) := ( (Bv(Bv'left) AND Av(Av'left) AND NOT res(Av'left)) OR
                 (NOT Bv(Bv'left) AND NOT Av(Av'left) AND res(Av'left)) );

    RETURN res;

  END add32;



  --- Compute difference of two 32-bit vectors (A - B). The result is two bits
  --- longer than the input vectors. Bit 33 of the result is the overflow bit,
  --- bit 32 is the carry out.

  FUNCTION sub32( Av, Bv : VLBIT_1D; Cin : VLBIT )
  RETURN VLBIT_1D IS
    VARIABLE res : VLBIT_1D(33 DOWNTO 0);
    VARIABLE carry : VLBIT;
  BEGIN

    carry := Cin;

    FOR k IN 0 TO 31 LOOP

      res(k) := Av(k) XOR (NOT Bv(k)) XOR carry;
      carry := (Av(k) AND (NOT Bv(k))) OR (carry AND (Av(k) OR (NOT Bv(k))));

    END LOOP;

    res(32) := carry;                          -- carry out
    --- overflow bit
    res(33) := ( (NOT Bv(Bv'left) AND Av(Av'left) AND NOT res(Av'left)) OR
                 (Bv(Bv'left) AND NOT Av(Av'left) AND res(Av'left)) );

    RETURN res;

  END sub32;


END p_arith;
```

2.18 p_constants.vhd

```
--- Constants used in place of magic numbers, where possible.


PACKAGE p_constants IS

---- Misc. useful constants.

  CONSTANT REG_R1: VLBIT_1D(4 DOWNTO 0) := (’0’,’0’,’0’,’0’,’1’);
  CONSTANT REG_R0: VLBIT_1D(4 DOWNTO 0) := (’0’,’0’,’0’,’0’,’0’);
  CONSTANT ZERO  : VLBIT_1D(151 DOWNTO 0) := (’0’,’0’,’0’,’0’,’0’,’0’,’0’,’0’,
                                              ’0’,’0’,’0’,’0’,’0’,’0’,’0’,’0’,
                                              ’0’,’0’,’0’,’0’,’0’,’0’,’0’,’0’,
                                              ’0’,’0’,’0’,’0’,’0’,’0’,’0’,’0’,
                                              ’0’,’0’,’0’,’0’,’0’,’0’,’0’,’0’,
                                              ’0’,’0’,’0’,’0’,’0’,’0’,’0’,’0’,
                                              ’0’,’0’,’0’,’0’,’0’,’0’,’0’,’0’,
                                              ’0’,’0’,’0’,’0’,’0’,’0’,’0’,’0’,
                                              ’0’,’0’,’0’,’0’,’0’,’0’,’0’,’0’,
                                              ’0’,’0’,’0’,’0’,’0’,’0’,’0’,’0’,
                                              ’0’,’0’,’0’,’0’,’0’,’0’,’0’,’0’,
                                              ’0’,’0’,’0’,’0’,’0’,’0’,’0’,’0’,
                                              ’0’,’0’,’0’,’0’,’0’,’0’,’0’,’0’,
                                              ’0’,’0’,’0’,’0’,’0’,’0’,’0’,’0’,
                                              ’0’,’0’,’0’,’0’,’0’,’0’,’0’,’0’,
                                              ’0’,’0’,’0’,’0’,’0’,’0’,’0’,’0’,
                                              ’0’,’0’,’0’,’0’,’0’,’0’,’0’,’0’,
                                              ’0’,’0’,’0’,’0’,’0’,’0’,’0’,’0’,
                                              ’0’,’0’,’0’,’0’,’0’,’0’,’0’,’0’);

  CONSTANT NEGMAX : VLBIT_1D(31 DOWNTO 0) := (’1’,’0’,’0’,’0’,’0’,’0’,’0’,’0’,
                                              ’0’,’0’,’0’,’0’,’0’,’0’,’0’,’0’,
                                              ’0’,’0’,’0’,’0’,’0’,’0’,’0’,’0’,
                                              ’0’,’0’,’0’,’0’,’0’,’0’,’0’,’0’);


---- Control register 0 bits.

  CONSTANT Imem0        : INTEGER := 0;
  CONSTANT Imem1        : INTEGER := 1;
  CONSTANT Imem2        : INTEGER := 2;
  CONSTANT Imem3        : INTEGER := 3;
  CONSTANT Supervisor   : INTEGER := 4;
  CONSTANT Int_Enable   : INTEGER := 5;
  CONSTANT Excp_Enable  : INTEGER := 6;
  CONSTANT Doit         : INTEGER := 7;
  CONSTANT Dmem0        : INTEGER := 8;
  CONSTANT Dmem1        : INTEGER := 9;
  CONSTANT Dmem2        : INTEGER := 10;
  CONSTANT Dmem3        : INTEGER := 11;

---- Number of Instruction Window slots
```

```
  CONSTANT IW_Max       : INTEGER := 16;

---- Number of control registers for each IW slot

  CONSTANT IW_CRNUM     : INTEGER := 5;

  CONSTANT IW_CR_STAT   : INTEGER := 0;         -- what they mean
  CONSTANT IW_CR_ADDR   : INTEGER := 1;
  CONSTANT IW_CR_OPCD   : INTEGER := 2;
  CONSTANT IW_CR_ARG1   : INTEGER := 3;
  CONSTANT IW_CR_ARG2   : INTEGER := 4;


---- Location of control registers. These are the registers which are not
---- normally visible.  CR0 is a special case, since it is visible everywhere.

  --- IF Unit registers
  CONSTANT IF_ICR_Min   : INTEGER := 1;         -- normal registers
  CONSTANT IF_ICR_Max   : INTEGER := 12;
  CONSTANT IF_SIW_Min   : INTEGER := 100;       -- shadow IW registers
  CONSTANT IF_SIW_Max   : INTEGER := IF_SIW_Min + IW_CRNUM*IW_Max - 1;


---- Exception vector numbers

  CONSTANT VECTOR_reset    : INTEGER := 0;
  CONSTANT VECTOR_imem     : INTEGER := 1;
  CONSTANT VECTOR_dmem     : INTEGER := 2;
  CONSTANT VECTOR_illegal  : INTEGER := 3;
  CONSTANT VECTOR_error    : INTEGER := 4;
  CONSTANT VECTOR_int      : INTEGER := 5;


---- Status code values.

  CONSTANT STATUS_none     : INTEGER := 0;      -- no status yet
  CONSTANT STATUS_done     : INTEGER := 1;      -- completed successfully
                                                -- undecided
                                                -- undecided

  CONSTANT STATUS_imem     : INTEGER := 4;      -- imem fault
                                                -- undecided
                                                -- undecided
                                                -- undecided

  CONSTANT STATUS_dmem     : INTEGER := 8;      -- dmem fault
  CONSTANT STATUS_align    : INTEGER := 9;      -- misaligned data access
                                                -- undecided
                                                -- undecided

  CONSTANT STATUS_illegal  : INTEGER := 12;     -- unimplemented opcode
  CONSTANT STATUS_user     : INTEGER := 13;     -- user mode violation
                                                -- undecided
                                                -- undecided
  CONSTANT STATUS_error    : INTEGER := 16;     -- unrecoverable error
                                                -- undecided
                                                -- undecided
                                                -- undecided

  CONSTANT STATUS_int      : INTEGER := 20;     -- external interrupt
  CONSTANT STATUS_sync     : INTEGER := 21;     -- sync.x aborted externally
                                                -- undecided
                                                -- undecided

  CONSTANT STATUS_lock_b0  : INTEGER := 24;     -- deadlock on BR Queue empty
  CONSTANT STATUS_lock_b1  : INTEGER := 25;     -- deadlock on BR Queue full
  CONSTANT STATUS_lock_r0  : INTEGER := 26;     -- deadlock on R1 Queue empty
  CONSTANT STATUS_lock_r1  : INTEGER := 27;     -- deadlock on R1 Queue full

  CONSTANT STATUS_overflow : INTEGER := 28;     -- integer overflow
                                                -- undecided
                                                -- undecided
                                                -- undecided

  CONSTANT STATUS_int_div  : INTEGER := 32;     -- integer divide error
                                                -- undecided
                                                -- undecided
                                                -- undecided


---- Trap code values.  When bit 8 of the status is set, it indicates that
---- the vector was taken as the result of a trap instruction. In the dispatch
---- unit, the major and minor status codes are each 16 bits wide, but only 8
---- bits are returned from the various units.

  CONSTANT STATUS_trap     : INTEGER := 256;    -- bits 7-0 are vector number



---- Returns the index number of the given opcode. Useful for decoding by
---- functional units.

  FUNCTION op2index( opcode: VLBIT_1D(31 DOWNTO 0) ) RETURN INTEGER;


END p_constants;




PACKAGE BODY p_constants IS


---- Returns the index number of the given opcode. Useful for decoding by
---- functional units.

  FUNCTION op2index( opcode: VLBIT_1D(31 DOWNTO 0) ) RETURN INTEGER IS
```

```
    VARIABLE major : INTEGER;
    VARIABLE minor : INTEGER;
    VARIABLE index : INTEGER;
  BEGIN

    major := v1d2int( opcode( 30 DOWNTO 26 ) ); -- ignore bit 31
    minor := v1d2int( opcode( 15 DOWNTO 10 ) );

    IF major = 23 THEN
      index := 32 + minor;
    ELSE
      index := major;
    END IF;

    RETURN index;

  END op2index;


END p_constants;
```

2.19 p_dispatch.vhd

```
USE work.p_constants.ALL;

PACKAGE p_dispatch IS                          -- package declaration


  CONSTANT OPCODE_doit : INTEGER := 16#5c005c00#; -- plain doit opcode


  --- These are EX Unit functional units.
  CONSTANT OP_arith  : INTEGER := 0;
  CONSTANT OP_logic  : INTEGER := 1;
  CONSTANT OP_ctrl   : INTEGER := 2;
  CONSTANT OP_branch : INTEGER := 3;
  CONSTANT OP_load   : INTEGER := 4;            -- mem unit

  --- These are handled inside the IF Unit.
  CONSTANT OP_doit   : INTEGER := 100;
  CONSTANT OP_rte    : INTEGER := 101;
  CONSTANT OP_sync   : INTEGER := 102;
  CONSTANT OP_sync_x : INTEGER := 103;

  --- These are used mostly for decoding.
  CONSTANT OP_trap   : INTEGER := 200;
  CONSTANT OP_ill    : INTEGER := 201;
  CONSTANT OP_mvbr   : INTEGER := 202;          -- ctrl unit
  CONSTANT OP_ldbr   : INTEGER := 203;          -- branch unit
  CONSTANT OP_store  : INTEGER := 204;          -- mem unit
  CONSTANT OP_xmem   : INTEGER := 205;          -- mem unit


  TYPE iw_slot IS RECORD
    valid         : BOOLEAN;                   -- slot is filled
    tag           : INTEGER;                   -- dispatch tag
    address       : INTEGER;                   -- instruction address
    opcode        : INTEGER;                   -- instruction opcode
    wat           : BOOLEAN;                   -- is a WAT instruction
    single        : BOOLEAN;                   -- is a SINGLE instruction
    issued        : BOOLEAN;                   -- has been issued
    status        : INTEGER;                   -- completion status
    dest          : INTEGER;                   -- dest reg for SB checking
    arg1          : INTEGER;                   -- for fault recovery
    arg2          : INTEGER;                   -- for fault recovery
  END RECORD;

  TYPE iw_type IS ARRAY ( 0 TO IW_Max - 1 ) OF iw_slot;


  --- This function returns TRUE if the opcode is a SINGLE type.
  ---
  FUNCTION is_single( opcode : VLBIT_1D(31 DOWNTO 0) ) RETURN BOOLEAN;
```

```
--- This removes the specified instruction from the IW.
---
PROCEDURE iw_remove ( VARIABLE iw_index : INOUT INTEGER; -- which to remove
                      VARIABLE iw : INOUT iw_type; -- IW
                      VARIABLE iw_bottom : INOUT INTEGER; -- iw_bottom
                      VARIABLE wat_flag : INOUT BOOLEAN; -- wat flag
                      VARIABLE single_flag : INOUT BOOLEAN ); -- single flag


--- determines decode information needed to dispatch instructions
---
PROCEDURE decode ( VARIABLE BR_Max          : IN  INTEGER;
                   VARIABLE XBR_Max         : IN  INTEGER;
                   VARIABLE R1_Max          : IN  INTEGER;
                   CONSTANT iopcode         : IN  INTEGER;
                   VARIABLE supervisor_mode : IN  BOOLEAN;
                   VARIABLE exceptions_ok   : IN  BOOLEAN;
                   VARIABLE r1_count        : IN  INTEGER;
                   VARIABLE r1_pending      : IN  INTEGER;
                   VARIABLE br_count        : IN  INTEGER;
                   VARIABLE xbr_count       : IN  INTEGER;
                   VARIABLE new_r1_count    : INOUT INTEGER;
                   VARIABLE new_r1_pending  : INOUT INTEGER;
                   VARIABLE new_br_count    : INOUT INTEGER;
                   VARIABLE new_xbr_count   : INOUT INTEGER;
                   VARIABLE status          : INOUT INTEGER;
                   VARIABLE op_type         : INOUT INTEGER;
                   VARIABLE single          : INOUT BOOLEAN;
                   VARIABLE wat             : INOUT BOOLEAN;
                   VARIABLE tellme          : INOUT BOOLEAN;
                   VARIABLE vA              : INOUT BOOLEAN;
                   VARIABLE srcA            : INOUT INTEGER;
                   VARIABLE vB              : INOUT BOOLEAN;
                   VARIABLE srcB            : INOUT INTEGER;
                   VARIABLE vD              : INOUT BOOLEAN;
                   VARIABLE srcD            : INOUT INTEGER;
                   VARIABLE dest            : INOUT INTEGER );

--- packs variable values into RF operand request vector
---
---    17   operand S requested
--- 16-12   operand S register number
---    11   operand A requested
---  10-6   operand A register number
---     5   operand B requested
---   4-0   operand B register number
---
FUNCTION rf_encode ( CONSTANT vA    : BOOLEAN;
                     CONSTANT srcA  : INTEGER;
                     CONSTANT vB    : BOOLEAN;
                     CONSTANT srcB  : INTEGER;
                     CONSTANT vD    : BOOLEAN;
                     CONSTANT srcD  : INTEGER )
RETURN VLBIT_1D;
```

```
END p_dispatch;                                 -- end of package declaration



PACKAGE BODY p_dispatch IS                      -- package body is here


  --- Define a table to help with decoding. This table is produced from the
  --- opcodes.tex file in the write-up, by running it through a Perl script. It
  --- doesn't need to mention the .d variants, since they are never decoded.
  --- Instead, when instructions are fetched from memory, the .d is used to
  --- insert a DOIT in the IW. If the IW is changed by an exception handler,
  --- any .d instructions must be explicitly expanded into two IW slots by
  --- software.

---- Start of insertion.

  CONSTANT INDEX_doit   : INTEGER := 55;
  CONSTANT INDEX_rte    : INTEGER := 80;
  CONSTANT INDEX_putcr  : INTEGER := 94;
  CONSTANT INDEX_putcr1 : INTEGER := 95;

  CONSTANT T            : BOOLEAN := TRUE;
  CONSTANT F            : BOOLEAN := FALSE;

  TYPE decode_table_entry IS RECORD
    bp      : BOOLEAN;                          -- branch producer
    bc      : BOOLEAN;                          -- branch consumer
    rcd     : BOOLEAN;                          -- r1 consumer (dest reg)
    rca     : BOOLEAN;                          -- r1 consumer (src reg A)
    rcb     : BOOLEAN;                          -- r1 consumer (src reg B)
    dr      : BOOLEAN;                          -- destination register
    wat     : BOOLEAN;                          -- wait-at-top instruction
    single  : BOOLEAN;                          -- single instruction
    usr     : BOOLEAN;                          -- user instruction only
    tellme  : BOOLEAN;                          -- should report DONE status
    op_type : INTEGER;                          -- type of instruction
  END RECORD;

  TYPE decode_table_type IS ARRAY ( 0 TO 95 ) OF decode_table_entry;
  CONSTANT decode_table : decode_table_type := (
    --- Fields are: BP BC RP RCd RCa RCb DR WAT SGL USR TELLME TYPE
    (F,F,F,T,F,T,F,F,F,F,OP_logic ),   --   0  and rd,ra,imm16
    (F,F,F,T,F,T,F,F,F,F,OP_logic ),   --   1  and.u rd,ra,imm16
    (F,F,F,T,F,T,F,F,F,F,OP_logic ),   --   2  mask rd,ra,imm16
    (F,F,F,T,F,T,F,F,F,F,OP_logic ),   --   3  mask.u rd,ra,imm16
    (F,F,F,T,F,T,F,F,F,F,OP_logic ),   --   4  or rd,ra,imm16
    (F,F,F,T,F,T,F,F,F,F,OP_logic ),   --   5  or.u rd,ra,imm16
    (F,F,F,T,F,T,F,F,F,F,OP_logic ),   --   6  xor rd,ra,imm16
    (F,F,F,T,F,T,F,F,F,F,OP_logic ),   --   7  xor.u rd,ra,imm16
    (F,F,F,T,F,T,F,F,F,T,OP_arith ),   --   8  add rd,ra,imm16
    (F,F,F,T,F,T,F,F,F,F,OP_arith ),   --   9  addu rd,ra,imm16
```

```
(F,F,F,T,F,T,F,F,F,T,OP_arith ),   --  10  div rd,ra,imm16
(F,F,F,T,F,T,F,F,F,T,OP_arith ),   --  11  divu rd,ra,imm16
(F,F,F,T,F,T,F,F,F,F,OP_arith ),   --  12  mul rd,ra,imm16
(F,F,F,T,F,T,F,F,F,F,OP_arith ),   --  13  cmp rd,ra,imm16
(F,F,F,T,F,T,F,F,F,T,OP_arith ),   --  14  sub rd,ra,imm16
(F,F,F,T,F,T,F,F,F,F,OP_arith ),   --  15  subu rd,ra,imm16
(T,F,F,T,F,F,F,F,F,F,OP_branch),   --  16  bb0 n,ra,imm16
(T,F,F,T,F,F,F,F,F,F,OP_branch),   --  17  bb1 n,ra,imm16
(T,F,F,T,F,F,F,F,F,F,OP_branch),   --  18  ble ra,imm16
(T,F,F,F,F,F,F,F,F,F,OP_branch),   --  19  br imm26
(F,F,F,F,F,T,F,F,F,F,OP_ctrl  ),   --  20  mvpc rd,imm16
(F,F,F,F,F,F,F,F,F,F,OP_trap  ),   --  21  trap imm8
(F,F,T,T,F,T,F,F,F,T,OP_xmem  ),   --  22  xmem rs,ra,imm16
(F,F,F,F,F,F,F,F,F,F,OP_ill   ),   --  23
(F,F,F,T,F,T,F,F,F,T,OP_load  ),   --  24  ld.bu rd,ra,imm16
(F,F,F,T,F,T,F,F,F,T,OP_load  ),   --  25  ld.b rd,ra,imm16
(F,F,F,T,F,T,F,F,F,T,OP_load  ),   --  26  ld.hu rd,ra,imm16
(F,F,F,T,F,T,F,F,F,T,OP_load  ),   --  27  ld.h rd,ra,imm16
(F,F,F,T,F,T,F,F,F,T,OP_load  ),   --  28  ld rd,ra,imm16
(F,F,T,T,F,F,F,F,F,T,OP_store ),   --  29  st.b rs,ra,imm16
(F,F,T,T,F,F,F,F,F,T,OP_store ),   --  30  st.h rs,ra,imm16
(F,F,T,T,F,F,F,F,F,T,OP_store ),   --  31  st rs,ra,imm16
(F,F,F,T,T,T,F,F,F,F,OP_logic ),   --  32  and rd,ra,rb
(F,F,F,T,T,T,F,F,F,F,OP_logic ),   --  33  and.c rd,ra,rb
(F,F,F,F,F,F,F,F,F,F,OP_ill   ),   --  34
(F,F,F,F,F,F,F,F,F,F,OP_ill   ),   --  35
(F,F,F,T,T,T,F,F,F,F,OP_logic ),   --  36  or rd,ra,rb
(F,F,F,T,T,T,F,F,F,F,OP_logic ),   --  37  or.c rd,ra,rb
(F,F,F,T,T,T,F,F,F,F,OP_logic ),   --  38  xor rd,ra,rb
(F,F,F,T,T,T,F,F,F,F,OP_logic ),   --  39  xor.c rd,ra,rb
(F,F,F,T,T,T,F,F,F,T,OP_arith ),   --  40  add.io rd,ra,rb
(F,F,F,T,T,T,F,F,F,F,OP_arith ),   --  41  addu.io rd,ra,rb
(F,F,F,T,T,T,F,F,F,T,OP_arith ),   --  42  div rd,ra,rb
(F,F,F,T,T,T,F,F,F,T,OP_arith ),   --  43  divu rd,ra,rb
(F,F,F,T,T,T,F,F,F,F,OP_arith ),   --  44  mul rd,ra,rb
(F,F,F,T,T,T,F,F,F,F,OP_arith ),   --  45  cmp rd,ra,rb
(F,F,F,T,T,T,F,F,F,T,OP_arith ),   --  46  sub.io rd,ra,rb
(F,F,F,T,T,T,F,F,F,F,OP_arith ),   --  47  subu.io rd,ra,rb
(T,F,F,T,T,F,F,F,F,F,OP_branch),   --  48  bb0 n,ra,rb
(T,F,F,T,T,F,F,F,F,F,OP_branch),   --  49  bb1 n,ra,rb
(T,F,F,T,T,F,F,F,F,F,OP_branch),   --  50  ble ra,rb
(T,F,F,F,T,F,F,F,F,F,OP_branch),   --  51  br rb
(F,F,F,F,F,F,F,F,F,F,OP_ill   ),   --  52
(F,F,F,F,F,F,F,F,F,F,OP_ill   ),   --  53
(F,F,T,T,T,T,F,F,T,T,OP_xmem  ),   --  54  xmem.usr rs,ra[rb]
(F,T,F,F,F,F,F,T,F,F,OP_doit  ),   --  55  doit
(F,F,F,T,T,T,F,F,T,T,OP_load  ),   --  56  ld.h.usr rd,ra[rb]
(F,F,F,T,T,T,F,F,F,F,OP_load  ),   --  57  lda rd,ra[rb]
(F,F,F,T,T,T,F,F,F,F,OP_load  ),   --  58  lda.h rd,ra[rb]
(F,F,F,F,F,F,F,F,F,F,OP_ill   ),   --  59
(F,F,T,T,T,F,F,F,T,T,OP_store ),   --  60  st.h.usr rs,ra[rb]
(F,F,F,F,F,F,F,F,F,F,OP_ill   ),   --  61
(F,F,F,F,F,F,F,F,F,F,OP_ill   ),   --  62
(F,F,F,F,F,F,F,F,F,F,OP_ill   ),   --  63
```

```
    (F,F,F,T,T,T,F,F,F,F,OP_logic ),   --  64  clr rd,ra,rb
    (F,F,F,T,T,T,F,F,F,F,OP_logic ),   --  65  set rd,ra,rb
    (F,F,F,T,T,T,F,F,F,F,OP_logic ),   --  66  ext rd,ra,rb
    (F,F,F,T,T,T,F,F,F,F,OP_logic ),   --  67  extu rd,ra,rb
    (F,F,F,T,T,T,F,F,F,F,OP_logic ),   --  68  mak rd,ra,rb
    (F,F,F,T,T,T,F,F,F,F,OP_logic ),   --  69  rot rd,ra,rb
    (F,F,F,F,T,T,F,F,F,F,OP_logic ),   --  70  ff0 rd,rb
    (F,F,F,F,T,T,F,F,F,F,OP_logic ),   --  71  ff1 rd,rb
    (F,F,F,T,F,T,F,F,F,F,OP_logic ),   --  72  clr rd,ra,w5<o5>
    (F,F,F,T,F,T,F,F,F,F,OP_logic ),   --  73  set rd,ra,w5<o5>
    (F,F,F,T,F,T,F,F,F,F,OP_logic ),   --  74  ext rd,ra,w5<o5>
    (F,F,F,T,F,T,F,F,F,F,OP_logic ),   --  75  extu rd,ra,w5<o5>
    (F,F,F,T,F,T,F,F,F,F,OP_logic ),   --  76  mak rd,ra,w5<o5>
    (F,F,F,T,F,T,F,F,F,F,OP_logic ),   --  77  rot rd,ra,<o5>
    (F,F,F,F,F,F,F,F,F,F,OP_ill   ),   --  78
    (F,F,F,F,F,F,F,F,F,F,OP_ill   ),   --  79
    (F,F,F,F,F,F,T,T,T,F,OP_rte   ),   --  80  rte
    (F,F,F,F,F,F,F,F,F,F,OP_ill   ),   --  81
    (F,F,F,F,F,F,F,F,F,F,OP_ill   ),   --  82
    (F,F,F,F,F,F,F,F,F,F,OP_ill   ),   --  83
    (F,F,F,F,F,F,T,F,F,F,OP_sync  ),   --  84  sync
    (F,F,F,F,F,F,T,F,T,F,OP_sync_x),   --  85  sync.x
    (F,T,F,F,F,T,F,F,F,F,OP_mvbr  ),   --  86  mvbr rd
    (T,F,F,T,F,F,F,F,F,F,OP_ldbr  ),   --  87  ldbr ra
    (F,F,F,F,F,F,F,F,F,F,OP_ill   ),   --  88
    (F,F,F,F,F,F,F,F,F,F,OP_ill   ),   --  89
    (F,F,F,F,F,F,F,F,F,F,OP_ill   ),   --  90
    (F,F,F,F,F,F,F,F,F,F,OP_ill   ),   --  91
    (F,F,F,F,F,T,F,F,T,F,OP_ctrl  ),   --  92  getcr rd,cr
    (F,F,F,F,T,T,F,F,T,F,OP_ctrl  ),   --  93  getcr rd,rb
    (F,F,F,T,F,F,T,T,T,T,OP_ctrl  ),   --  94  putcr cr,ra
    (F,F,F,T,T,F,T,T,T,T,OP_ctrl  ));  --  95  putcr rb,ra

---- End of insertion.



  --- This function returns TRUE if the opcode is a SINGLE type.
  ---
  FUNCTION is_single( opcode : VLBIT_1D(31 DOWNTO 0) ) RETURN BOOLEAN IS
    VARIABLE index : INTEGER;
  BEGIN

    index := op2index( opcode );

    RETURN decode_table(index).single;

  END is_single;



  --- This removes the specified instruction from the IW.
  ---
  PROCEDURE iw_remove ( VARIABLE iw_index : INOUT INTEGER; -- which to remove
                        VARIABLE iw : INOUT iw_type; -- IW
                        VARIABLE iw_bottom : INOUT INTEGER; -- iw_bottom
                        VARIABLE wat_flag : INOUT BOOLEAN; -- wat flag
                        VARIABLE single_flag : INOUT BOOLEAN ) --  single flag
  IS
  BEGIN

    IF iw(iw_index).wat THEN                    -- clear WAT if appropriate
      wat_flag := FALSE;
    END IF;

    IF iw(iw_index).single THEN                 -- clear SINGLE if appropriate
      single_flag := FALSE;
    END IF;

    --- remove this instruction from the IW, and shift the others up

    FOR i IN (iw_index + 1) TO (iw_bottom - 1) LOOP
      iw(i-1) := iw(i);                         -- copy 'em upward
    END LOOP;
    iw_bottom := iw_bottom - 1;                 -- change last pointer
    iw(iw_bottom).valid := FALSE;               -- zap old last entry

  END iw_remove;



  --- determines decode information needed to dispatch instructions
  ---
  PROCEDURE decode ( VARIABLE BR_Max          : IN  INTEGER;
                     VARIABLE XBR_Max         : IN  INTEGER;
                     VARIABLE R1_Max          : IN  INTEGER;
                     CONSTANT iopcode         : IN  INTEGER;
                     VARIABLE supervisor_mode : IN  BOOLEAN;
                     VARIABLE exceptions_ok   : IN  BOOLEAN;
                     VARIABLE r1_count        : IN  INTEGER;
                     VARIABLE r1_pending      : IN  INTEGER;
                     VARIABLE br_count        : IN  INTEGER;
                     VARIABLE xbr_count       : IN  INTEGER;
                     VARIABLE new_r1_count    : INOUT INTEGER;
                     VARIABLE new_r1_pending  : INOUT INTEGER;
                     VARIABLE new_br_count    : INOUT INTEGER;
                     VARIABLE new_xbr_count   : INOUT INTEGER;
                     VARIABLE status          : INOUT INTEGER;
                     VARIABLE op_type         : INOUT INTEGER;
                     VARIABLE single          : INOUT BOOLEAN;
                     VARIABLE wat             : INOUT BOOLEAN;
                     VARIABLE tellme          : INOUT BOOLEAN;
                     VARIABLE vA              : INOUT BOOLEAN;
                     VARIABLE srcA            : INOUT INTEGER;
                     VARIABLE vB              : INOUT BOOLEAN;
                     VARIABLE srcB            : INOUT INTEGER;
                     VARIABLE vD              : INOUT BOOLEAN;
                     VARIABLE srcD            : INOUT INTEGER;
                     VARIABLE dest            : INOUT INTEGER )
  IS
    VARIABLE index        : INTEGER;
    VARIABLE opcode       : VLBIT_1D( 31 DOWNTO 0 );
    VARIABLE tmpi         : INTEGER;
  BEGIN

    opcode := int2v1d( iopcode );
    index := op2index( opcode );

    --- set up default values for outputs

    new_r1_count := r1_count;
    new_r1_pending := r1_pending;
    new_br_count := br_count;
    new_xbr_count := xbr_count;
    status := STATUS_none;
    op_type := decode_table(index).op_type;
    single := decode_table(index).single;
    wat := decode_table(index).wat;
    tellme := decode_table(index).tellme;

    vA := decode_table(index).rca;
    vB := decode_table(index).rcb;
    vD := decode_table(index).rcd;
    srcA := v1d2int( opcode(20 DOWNTO 16) );    -- operand A
    srcB := v1d2int( opcode(4 DOWNTO 0) );      -- operand B
    srcD := v1d2int( opcode(25 DOWNTO 21) );    -- operand D

    IF decode_table(index).dr THEN
      dest := srcD;
    ELSE
      dest := 0;
    END IF;




    --- do the easy ones first

    IF decode_table(index).op_type = OP_ill THEN -- unimplemented opcode
      status := STATUS_illegal;
      RETURN;                                   -- so quit
    END IF;


    IF ( opcode(31) = '1' AND                   -- inserts extra DOIT
         ( index = INDEX_doit OR                -- but that's not allowed for
           index = INDEX_rte OR                 -- these instructions!
           index = INDEX_putcr OR
           index = INDEX_putcr1 ) )
    THEN
      status := STATUS_illegal;
      RETURN;                                   -- so quit
    END IF;
```

```
--- test for user mode violations

IF (NOT supervisor_mode) AND decode_table(index).usr THEN

  --- Memory instructions may or may not be user mode instructions. The
  --- .usr flag is encoded with bit 9 of the opcode. It's only a user mode
  --- violation if that bit is set for a memory operation. If it's not a
  --- memory operation, then it is a user mode violation.

  IF ( ( decode_table(index).op_type /= OP_load AND -- not memory operation
         decode_table(index).op_type /= OP_store AND
         decode_table(index).op_type /= OP_xmem ) OR
       opcode(9) = '1' )                    -- or is mem, and not user
  THEN
    status := STATUS_user;                    -- users can't do this
    RETURN;
  END IF;

END IF;


IF decode_table(index).op_type = OP_trap THEN

  tmpi := v1d2int( opcode(7 DOWNTO 0) );    -- find trap number

  IF (NOT supervisor_mode) AND tmpi < 128 THEN
    status := STATUS_user;                  -- users only trap above 127
    RETURN;
  END IF;

  status := STATUS_trap + tmpi;             -- indicate trap
  RETURN;

END IF;

--- check deadlock problems

IF decode_table(index).bc THEN              -- branch consumer (0 or 1)
  IF ( exceptions_ok OR
       decode_table(index).op_type = OP_mvbr )
  THEN                                      -- using normal Branch Queue
    new_br_count := new_br_count - 1;
    IF new_br_count < 0 THEN                -- deadlock
      status := STATUS_lock_b0;
      new_br_count := br_count;
      new_xbr_count := xbr_count;
      RETURN;
    END IF;
  ELSE                                      -- using X Branch Queue
    new_xbr_count := new_xbr_count - 1;
    IF new_xbr_count < 0 THEN               -- deadlock
      status := STATUS_lock_b0;
```

```
        new_br_count := br_count;
        new_xbr_count := xbr_count;
        RETURN;
      END IF;
    END IF;
  END IF;

  IF decode_table(index).bp THEN              -- branch producer (0 or 1)
    IF ( exceptions_ok OR
         decode_table(index).op_type = OP_ldbr )
    THEN                                      -- using normal Branch Queue
      new_br_count := new_br_count + 1;
      IF new_br_count > BR_Max THEN           -- deadlock
        status := STATUS_lock_b1;
        new_br_count := br_count;
        new_xbr_count := xbr_count;
        RETURN;
      END IF;
    ELSE                                      -- using X Branch Queue
      new_xbr_count := new_xbr_count + 1;
      IF new_xbr_count > XBR_Max THEN         -- deadlock
        status := STATUS_lock_b1;
        new_br_count := br_count;
        new_xbr_count := xbr_count;
        RETURN;
      END IF;
    END IF;
  END IF;


  --- Must look at three fields to determine R1 consumer counts.

  IF vA AND srcA = 1 THEN
    new_r1_count := new_r1_count - 1;
  END IF;
  IF vB AND srcB = 1 THEN
    new_r1_count := new_r1_count - 1;
  END IF;
  IF vD AND srcD = 1 THEN
    new_r1_count := new_r1_count - 1;
  END IF;

  IF new_r1_count + r1_pending < 0 THEN       -- deadlock
    status := STATUS_lock_r0;
    new_br_count := br_count;
    new_xbr_count := xbr_count;
    new_r1_count := r1_count;
    RETURN;
  END IF;


  IF decode_table(index).dr AND dest = 1 THEN -- R1 producer
    tellme := TRUE;                           -- need to know when done
    new_r1_pending := r1_pending + 1;
    --- Should I check r1_count or new_r1_count to determine deadlock? I
    --- think I can check new_r1_count, since nothing can actually issue
    --- until the pipeline has cleared, and the R1 sources get used before
    --- the new destinations are produced.
    IF new_r1_count + new_r1_pending > R1_Max THEN -- deadlock
      status := STATUS_lock_r1;
      new_br_count := br_count;
      new_xbr_count := xbr_count;
      new_r1_count := r1_count;
      new_r1_pending := r1_pending;
      RETURN;
    END IF;
  END IF;


  --- Now, if the R1 Queue can satisfy the operand requirements NOW, without
  --- having to wait for pending operations to complete (because they might
  --- not be successful), then I want to go ahead and dispatch instead of
  --- blocking on the R1 scoreboard bit.  I do this by clearing vA, vB, or vD
  --- as needed, so that the scoreboard is not checked by the dispatch
  --- routine.  This doesn't affect the blocking due to the destination
  --- register, which must take place if needed.

  IF new_r1_count >= 0 THEN
    IF vA AND srcA = 1 THEN
      vA := FALSE;
    END IF;
    IF vB AND srcB = 1 THEN
      vB := FALSE;
    END IF;
    IF vD AND srcD = 1 THEN
      vD := FALSE;
    END IF;
  END IF;


  --- By getting this far, I've decoded everything with no faults.

END decode;



--- packs variable values into RF operand request vector
---
---    17   operand S requested
--- 16-12   operand S register number
---    11   operand A requested
---  10-6   operand A register number
---     5   operand B requested
---   4-0   operand B register number
---
FUNCTION rf_encode ( CONSTANT vA    : BOOLEAN;
                     CONSTANT srcA  : INTEGER;
                     CONSTANT vB    : BOOLEAN;
```

```
                       CONSTANT srcB  : INTEGER;
                       CONSTANT vD    : BOOLEAN;
                       CONSTANT srcD  : INTEGER )
  RETURN VLBIT_1D IS
    VARIABLE result : VLBIT_1D( 17 DOWNTO 0 );
    VARIABLE tmpv : VLBIT_1D( 31 DOWNTO 0 );
  BEGIN

    result( 17 ) := boo2vlb( vD );
    tmpv := int2v1d( srcD );
    result( 16 DOWNTO 12 ) := tmpv( 4 DOWNTO 0 );

    result( 11 ) := boo2vlb( vA );
    tmpv := int2v1d( srcA );
    result( 10 DOWNTO 6 ) := tmpv( 4 DOWNTO 0 );

    result( 5 ) := boo2vlb( vB );
    tmpv := int2v1d( srcB );
    result( 4 DOWNTO 0 ) := tmpv( 4 DOWNTO 0 );

    RETURN result;

  END rf_encode;

END p_dispatch;
```

2.20 p_memory_unit.vhd

```
USE work.p_constants.ALL;


PACKAGE p_memory_unit IS

  PROCEDURE mem_decode( opcode          : IN VLBIT_1D (31 DOWNTO 0);
                        imm             : INOUT BOOLEAN;
                        opsize          : INOUT INTEGER;
                        scaled          : INOUT BOOLEAN;
                        do_sign_ext     : INOUT BOOLEAN;
                        force_usr_mode  : INOUT BOOLEAN;
                        do_load         : INOUT BOOLEAN;
                        do_store        : INOUT BOOLEAN );

END p_memory_unit;


PACKAGE BODY p_memory_unit IS


  PROCEDURE mem_decode( opcode          : IN VLBIT_1D (31 DOWNTO 0);
                        imm             : INOUT BOOLEAN;
                        opsize          : INOUT INTEGER;
                        scaled          : INOUT BOOLEAN;
                        do_sign_ext     : INOUT BOOLEAN;
                        force_usr_mode  : INOUT BOOLEAN;
                        do_load         : INOUT BOOLEAN;
                        do_store        : INOUT BOOLEAN )
  IS
    TYPE retval_type IS RECORD
      imm                 : BOOLEAN;
      opsize              : INTEGER;
      scaled              : BOOLEAN;
      do_sign_ext         : BOOLEAN;
      force_usr_mode      : BOOLEAN;
      do_load             : BOOLEAN;
      do_store            : BOOLEAN;
    END RECORD;

    VARIABLE retval       : retval_type;
    VARIABLE index        : INTEGER;
  BEGIN

    index := op2index( opcode );

    --- This stuff had to be encoded by hand.  I could have written a Perl
    --- script to do it, but it would have taken longer than doing it manually.
    --- This stuff shouldn't change that much (I hope).  I also made some
    --- assumptions about unused bits, which is probably okay.


    CASE index IS
    WHEN 22 =>                                  -- xmem rs,ra,imm16
      retval := ( TRUE, 4, FALSE, FALSE, FALSE, TRUE, TRUE );
    WHEN 24 =>                                  -- ld.bu rd,ra,imm16
      retval := ( TRUE, 1, FALSE, FALSE, FALSE, TRUE, FALSE );
    WHEN 25 =>                                  -- ld.b rd,ra,imm16
      retval := ( TRUE, 1, FALSE, TRUE, FALSE, TRUE, FALSE );
    WHEN 26 =>                                  -- ld.hu rd,ra,imm1
      retval := ( TRUE, 2, FALSE, FALSE, FALSE, TRUE, FALSE );
    WHEN 27 =>                                  -- ld.h rd,ra,imm16
      retval := ( TRUE, 2, FALSE, TRUE, FALSE, TRUE, FALSE );
    WHEN 28 =>                                  -- ld rd,ra,imm16
      retval := ( TRUE, 4, FALSE, FALSE, FALSE, TRUE, FALSE );
    WHEN 29 =>                                  -- st.b rs,ra,imm16
      retval := ( TRUE, 1, FALSE, FALSE, FALSE, FALSE, TRUE );
    WHEN 30 =>                                  -- st.h rs,ra,imm16
      retval := ( TRUE, 2, FALSE, FALSE, FALSE, FALSE, TRUE );
    WHEN 31 =>                                  -- st rs,ra,imm16
      retval := ( TRUE, 4, FALSE, FALSE, FALSE, FALSE, TRUE );

    WHEN 54 =>                                  -- xmem[.usr] rs,ra,rb
      retval := ( FALSE, 4, FALSE, FALSE, FALSE, TRUE, TRUE );
      retval.scaled := vlb2boo( opcode(5) );
      IF opcode(9) = '1' THEN
        retval.force_usr_mode := TRUE;
      END IF;

    WHEN 56 =>                                  -- flavors of triadic load
      retval.imm := FALSE;
      retval.force_usr_mode := vlb2boo( opcode(9) );
      retval.scaled := vlb2boo( opcode(5) );
      retval.opsize := 4;                       -- default size
      IF opcode(8) = '1' THEN                   -- half
        retval.opsize := 2;
      END IF;
      IF opcode(7) = '1' THEN                   -- byte overrides half
        retval.opsize := 1;
      END IF;
      retval.do_sign_ext := vlb2boo( opcode(6) ); -- doesn't matter for word
      retval.do_load := TRUE;
      retval.do_store := FALSE;

    WHEN 57 =>                                  -- lda rd,ra[rb]
      retval := ( FALSE, 4, TRUE, FALSE, FALSE, FALSE, FALSE );

    WHEN 58 =>                                  -- lda.h rd,ra[rb]
      retval := ( FALSE, 2, TRUE, FALSE, FALSE, FALSE, FALSE );

    WHEN 60 =>                                  -- flavors of triadic store
      retval.imm := FALSE;
      retval.force_usr_mode := vlb2boo( opcode(9) );
      retval.scaled := vlb2boo( opcode(5) );
      retval.opsize := 4;                       -- default size
      IF opcode(8) = '1' THEN                   -- half
        retval.opsize := 2;
      END IF;
      IF opcode(7) = '1' THEN                     -- byte overrides half
        retval.opsize := 1;
      END IF;
      retval.do_sign_ext := FALSE;
      retval.do_load := FALSE;
      retval.do_store := TRUE;


    WHEN OTHERS =>
      putline("*** Illegal Opcode for Memory Unit ***");
      retval := ( FALSE, 0, FALSE, FALSE, FALSE, FALSE, FALSE );
    END CASE;

    imm := retval.imm;
    opsize := retval.opsize;
    scaled := retval.scaled;
    do_sign_ext := retval.do_sign_ext;
    force_usr_mode := retval.force_usr_mode;
    do_load := retval.do_load;
    do_store := retval.do_store;

  END mem_decode;

END p_memory_unit;
```

2.21 p_output.vhd

```
--------------------------------------------------------------------------------
---- output utilities
--------------------------------------------------------------------------------


PACKAGE p_output IS                            -- package declaration

  --- function to print 32-bit integer values in hex

  PROCEDURE puthex ( stuff : IN character_1d; val : IN INTEGER );

  --- function to print 32-bit integer values in hex followed by newline

  PROCEDURE puthexline ( stuff : IN character_1d; val : IN INTEGER );

END p_output;                                  -- end of package declaration


--------------------------------------------------------------------------------
--------------------------------------------------------------------------------


PACKAGE BODY p_output IS                       -- package body is here

  --- local function to print one hex digit
  ---
  PROCEDURE puthexdigit (VARIABLE val : IN vlbit_vector( 3 DOWNTO 0) ) IS
    VARIABLE dummy : INTEGER;
  BEGIN
    dummy := v1d2int(val);
    CASE dummy IS
      WHEN 0 => put("0");
      WHEN 1 => put("1");
      WHEN 2 => put("2");
      WHEN 3 => put("3");
      WHEN 4 => put("4");
      WHEN 5 => put("5");
      WHEN 6 => put("6");
      WHEN 7 => put("7");
      WHEN 8 => put("8");
      WHEN 9 => put("9");
      WHEN 10 => put("A");
      WHEN 11 => put("B");
      WHEN 12 => put("C");
      WHEN 13 => put("D");
      WHEN 14 => put("E");
      WHEN 15 => put("F");
      WHEN OTHERS => put("?");
    END CASE;
  END puthexdigit;

  --- package function to print integers as hex values
  ---
  PROCEDURE puthex ( stuff : IN character_1d ; val : IN INTEGER ) IS
    VARIABLE dummy : vlbit_vector (31 DOWNTO 0);
    VARIABLE stupid : vlbit_vector (3 DOWNTO 0);
  BEGIN
    put(stuff);
    dummy := int2v1d( val );
    stupid := dummy(31 DOWNTO 28);             -- this is stupid, but it
    puthexdigit( stupid );                     -- complains if I don't do it.
    stupid := dummy(27 DOWNTO 24);
    puthexdigit( stupid );
    stupid := dummy(23 DOWNTO 20);
    puthexdigit( stupid );
    stupid := dummy(19 DOWNTO 16);
    puthexdigit( stupid );
    stupid := dummy(15 DOWNTO 12);
    puthexdigit( stupid );
    stupid := dummy(11 DOWNTO 8);
    puthexdigit( stupid );
    stupid := dummy(7 DOWNTO 4);
    puthexdigit( stupid );
    stupid := dummy(3 DOWNTO 0);
    puthexdigit( stupid );
  END puthex;


  --- package function to print integers as hex values, followed by newline
  ---
  PROCEDURE puthexline ( stuff : IN character_1d; val : IN INTEGER ) IS
    VARIABLE dummy : vlbit_vector (31 DOWNTO 0);
    VARIABLE stupid : vlbit_vector (3 DOWNTO 0);
  BEGIN
    put(stuff);
    dummy := int2v1d( val );
    stupid := dummy(31 DOWNTO 28);             -- this is stupid, but it
    puthexdigit( stupid );                     -- complains if I don't do it.
    stupid := dummy(27 DOWNTO 24);
    puthexdigit( stupid );
    stupid := dummy(23 DOWNTO 20);
    puthexdigit( stupid );
    stupid := dummy(19 DOWNTO 16);
    puthexdigit( stupid );
    stupid := dummy(15 DOWNTO 12);
    puthexdigit( stupid );
    stupid := dummy(11 DOWNTO 8);
    puthexdigit( stupid );
    stupid := dummy(7 DOWNTO 4);
    puthexdigit( stupid );
    stupid := dummy(3 DOWNTO 0);
    puthexdigit( stupid );
    putline("");
  END puthexline;

END p_output;
```

2.22 qflop.vhd

```
--- A Q-flop allows a signal to be sampled on request.  It generates an
--- acknowledge when the signal is no longer metastable.  This implementation
--- is a little sloppy, but it will work for VHDL, since VHDL doesn't have
--- metastability problems.  This cheats in some ways, since the input is
--- passed straight through to the output. That means you don't have to probe
--- if you don't want to.

ENTITY qflop IS
  GENERIC( delay: TIME := 100 ps );
  PORT (SIGNAL r              : IN  VLBIT;
        SIGNAL a              : OUT VLBIT;
        SIGNAL i              : IN  VLBIT;
        SIGNAL o              : OUT VLBIT);
END qflop;

ARCHITECTURE behavior OF qflop IS
BEGIN
  main : PROCESS
    BEGIN

      IF bitunknown(r) THEN

        a <= '0';

      ELSIF r'EVENT THEN                          -- probe request

        IF bitunknown(i) THEN
          WAIT UNTIL NOT bitunknown(i);
        END IF;

        o <= i;
        a <= r AFTER delay;

      ELSE                                        -- input change

        o <= i;

      END IF;

      WAIT ON r,i;

    END PROCESS main;
END behavior;
```

2.23 registers.vhd

```
USE work.p_output.ALL;

---- This implements the register bank.  Incoming values are updated on the
---- outputs, and the scoreboard is cleared.  I don't have to worry about
---- arbitration, since the scoreboard will prevent any outstanding registers
---- from being read.


ENTITY registers IS
  GENERIC( delay: TIME := 100 ps );
  PORT( SIGNAL clr              : IN  VLBIT;
        SIGNAL clr_sb           : OUT VLBIT_1D (4 DOWNTO 0);
        SIGNAL clr_sb_ack       : IN  VLBIT;
        SIGNAL clr_sb_req       : OUT VLBIT;
        SIGNAL r02              : OUT VLBIT_1D (31 DOWNTO 0);
        SIGNAL r03              : OUT VLBIT_1D (31 DOWNTO 0);
        SIGNAL r04              : OUT VLBIT_1D (31 DOWNTO 0);
        SIGNAL r05              : OUT VLBIT_1D (31 DOWNTO 0);
        SIGNAL r06              : OUT VLBIT_1D (31 DOWNTO 0);
        SIGNAL r07              : OUT VLBIT_1D (31 DOWNTO 0);
        SIGNAL r08              : OUT VLBIT_1D (31 DOWNTO 0);
        SIGNAL r09              : OUT VLBIT_1D (31 DOWNTO 0);
        SIGNAL r10              : OUT VLBIT_1D (31 DOWNTO 0);
        SIGNAL r11              : OUT VLBIT_1D (31 DOWNTO 0);
        SIGNAL r12              : OUT VLBIT_1D (31 DOWNTO 0);
        SIGNAL r13              : OUT VLBIT_1D (31 DOWNTO 0);
        SIGNAL r14              : OUT VLBIT_1D (31 DOWNTO 0);
        SIGNAL r15              : OUT VLBIT_1D (31 DOWNTO 0);
        SIGNAL r16              : OUT VLBIT_1D (31 DOWNTO 0);
        SIGNAL r17              : OUT VLBIT_1D (31 DOWNTO 0);
        SIGNAL r18              : OUT VLBIT_1D (31 DOWNTO 0);
        SIGNAL r19              : OUT VLBIT_1D (31 DOWNTO 0);
        SIGNAL r20              : OUT VLBIT_1D (31 DOWNTO 0);
        SIGNAL r21              : OUT VLBIT_1D (31 DOWNTO 0);
        SIGNAL r22              : OUT VLBIT_1D (31 DOWNTO 0);
        SIGNAL r23              : OUT VLBIT_1D (31 DOWNTO 0);
        SIGNAL r24              : OUT VLBIT_1D (31 DOWNTO 0);
        SIGNAL r25              : OUT VLBIT_1D (31 DOWNTO 0);
        SIGNAL r26              : OUT VLBIT_1D (31 DOWNTO 0);
        SIGNAL r27              : OUT VLBIT_1D (31 DOWNTO 0);
        SIGNAL r28              : OUT VLBIT_1D (31 DOWNTO 0);
        SIGNAL r29              : OUT VLBIT_1D (31 DOWNTO 0);
        SIGNAL r30              : OUT VLBIT_1D (31 DOWNTO 0);
        SIGNAL r31              : OUT VLBIT_1D (31 DOWNTO 0);
        SIGNAL rt0_a            : OUT VLBIT;
        SIGNAL rt0_r            : IN  VLBIT;
        SIGNAL rt0_v            : IN  VLBIT_1D (36 DOWNTO 0);
        SIGNAL rt1_a            : OUT VLBIT;
        SIGNAL rt1_r            : IN  VLBIT;
        SIGNAL rt1_v            : IN  VLBIT_1D (36 DOWNTO 0);
        SIGNAL rt2_a            : OUT VLBIT;
        SIGNAL rt2_r            : IN  VLBIT;
        SIGNAL rt2_v            : IN  VLBIT_1D (36 DOWNTO 0);
        SIGNAL rt3_a            : OUT VLBIT;
        SIGNAL rt3_r            : IN  VLBIT;
        SIGNAL rt3_v            : IN  VLBIT_1D (36 DOWNTO 0));
END registers;


ARCHITECTURE behavior OF registers IS
BEGIN
  main : PROCESS

    --- type definitions

    TYPE register_type IS ARRAY ( 0 TO 31 ) OF VLBIT_1D (31 DOWNTO 0);

    --- internal variables

    VARIABLE reg               : register_type; -- registers

    VARIABLE debug             : BOOLEAN := FALSE;
    VARIABLE trace             : BOOLEAN := FALSE;


    --- Input signal event variables. These are the events which wake up this
    --- process, for various reasons.

    VARIABLE l_rt0_r           : VLBIT;
    VARIABLE l_rt1_r           : VLBIT;
    VARIABLE l_rt2_r           : VLBIT;
    VARIABLE l_rt3_r           : VLBIT;


    --- output signal value variables

    VARIABLE v_clr_sb_req      : VLBIT;

    --- scratch variables

    VARIABLE which_result      : INTEGER;
    VARIABLE dest              : INTEGER;
    VARIABLE stuff             : VLBIT_1D (36 DOWNTO 0);
    VARIABLE tmpv              : VLBIT_1D (31 DOWNTO 0);
    VARIABLE tmpi              : INTEGER;

  BEGIN

    ---
    --- If clear is asserted, reset everything and stick here.
    ---

    IF clr /= '1' THEN                          -- reset to beginning

      FOR i IN 0 TO 31 LOOP                     -- clear registers
        reg(i) := int2v1d( 0 );
      END LOOP;

      which_result := 0;
      l_rt0_r := '0';
      l_rt1_r := '0';
      l_rt2_r := '0';
      l_rt3_r := '0';

      v_clr_sb_req := '0';

      clr_sb_req <= v_clr_sb_req;
      clr_sb <= ('0','0','0','0','0');
      rt0_a <= '0';
      rt1_a <= '0';
      rt2_a <= '0';
      rt3_a <= '0';

      r02 <= reg(2);
      r03 <= reg(3);
      r04 <= reg(4);
      r05 <= reg(5);
      r06 <= reg(6);
      r07 <= reg(7);
      r08 <= reg(8);
      r09 <= reg(9);
      r10 <= reg(10);
      r11 <= reg(11);
      r12 <= reg(12);
      r13 <= reg(13);
      r14 <= reg(14);
      r15 <= reg(15);
      r16 <= reg(16);
      r17 <= reg(17);
      r18 <= reg(18);
      r19 <= reg(19);
      r20 <= reg(20);
      r21 <= reg(21);
      r22 <= reg(22);
      r23 <= reg(23);
      r24 <= reg(24);
      r25 <= reg(25);
      r26 <= reg(26);
      r27 <= reg(27);
      r28 <= reg(28);
      r29 <= reg(29);
      r30 <= reg(30);
      r31 <= reg(31);

      WAIT UNTIL clr = '1';                    -- wait till ready to run


    ELSE                                       -- unit is running

---- Normal running condition
```

```
--- The process loses events if I'm busy elsewhere, so I need to make
--- my own virtual wait statement, instead of relying on the actual
--- wait from VHDL. I'll just sample by hand, and only wait if there
--- are no pending requests. Since the REQ/ACK is paired, this will
--- work fine.

WHILE ( rt0_r /= l_rt0_r OR
        rt1_r /= l_rt1_r OR
        rt2_r /= l_rt2_r OR
        rt3_r /= l_rt3_r )
LOOP

  --- All result queues have this format:
  ---
  --- 36-32   destination register
  ---  31-0   value
  ---

  IF rt0_r /= l_rt0_r THEN
    l_rt0_r := rt0_r;
    stuff := rt0_v;
    which_result := 0;
  ELSIF rt1_r /= l_rt1_r THEN
    l_rt1_r := rt1_r;
    stuff := rt1_v;
    which_result := 1;
  ELSIF rt2_r /= l_rt2_r THEN
    l_rt2_r := rt2_r;
    stuff := rt2_v;
    which_result := 2;
  ELSIF rt3_r /= l_rt3_r THEN
    l_rt3_r := rt3_r;
    stuff := rt3_v;
    which_result := 3;
  ELSE
    putline("*** What Happened? ***");
  END IF;


  dest := v1d2int( stuff(36 DOWNTO 32) ); -- see where it goes

  IF dest /= 0 AND dest /= 1 THEN         -- ignore writes to r0 and r1

    tmpv := stuff(31 DOWNTO 0);
    reg(dest) := tmpv;                    -- write the new value

    IF debug OR trace THEN
      tmpi := v1d2int(tmpv);
      put("R",dest);
      puthexline(" <= ",tmpi);
    END IF;
```

```
CASE dest IS
WHEN 2 =>
  r02 <= tmpv;
WHEN 3 =>
  r03 <= tmpv;
WHEN 4 =>
  r04 <= tmpv;
WHEN 5 =>
  r05 <= tmpv;
WHEN 6 =>
  r06 <= tmpv;
WHEN 7 =>
  r07 <= tmpv;
WHEN 8 =>
  r08 <= tmpv;
WHEN 9 =>
  r09 <= tmpv;
WHEN 10 =>
  r10 <= tmpv;
WHEN 11 =>
  r11 <= tmpv;
WHEN 12 =>
  r12 <= tmpv;
WHEN 13 =>
  r13 <= tmpv;
WHEN 14 =>
  r14 <= tmpv;
WHEN 15 =>
  r15 <= tmpv;
WHEN 16 =>
  r16 <= tmpv;
WHEN 17 =>
  r17 <= tmpv;
WHEN 18 =>
  r18 <= tmpv;
WHEN 19 =>
  r19 <= tmpv;
WHEN 20 =>
  r20 <= tmpv;
WHEN 21 =>
  r21 <= tmpv;
WHEN 22 =>
  r22 <= tmpv;
WHEN 23 =>
  r23 <= tmpv;
WHEN 24 =>
  r24 <= tmpv;
WHEN 25 =>
  r25 <= tmpv;
WHEN 26 =>
  r26 <= tmpv;
WHEN 27 =>
  r27 <= tmpv;
WHEN 28 =>
```

```
          r28 <= tmpv;
        WHEN 29 =>
          r29 <= tmpv;
        WHEN 30 =>
          r30 <= tmpv;
        WHEN 31 =>
          r31 <= tmpv;
        WHEN OTHERS =>
          NULL;
        END CASE;

        --- clear the scoreboard

        IF debug THEN
          tmpi := v1d2int(stuff(36 DOWNTO 32));
          putline("clearing scoreboard bit ",tmpi);
        END IF;

        clr_sb <= stuff(36 DOWNTO 32);
        v_clr_sb_req := NOT v_clr_sb_req;
        clr_sb_req <= v_clr_sb_req AFTER delay;

        WAIT ON clr_sb_ack;

      END IF;

      IF which_result = 0 THEN
        rt0_a <= rt0_r AFTER delay;
      ELSIF which_result = 1 THEN
        rt1_a <= rt1_r AFTER delay;
      ELSIF which_result = 2 THEN
        rt2_a <= rt2_r AFTER delay;
      ELSIF which_result = 3 THEN
        rt3_a <= rt3_r AFTER delay;
      ELSE
        putline("*** What Happened? ***");
      END IF;

    END LOOP;                                -- main process loop

  END IF;                                    -- CLR not asserted


  WAIT ON clr, rt0_r, rt1_r, rt2_r, rt3_r;

END PROCESS main;

END behavior;
```

2.24 regselect.vhd

```
USE work.p_constants.ALL;
USE work.p_output.ALL;

---- This module puts together the operands for the EX Unit in response to the
---- requests from the IF Unit.  If R1 operands are requested, I need to gather
---- them in the correct order.

ENTITY regselect IS
  GENERIC( delay: TIME := 100 ps );
  PORT( SIGNAL clr              : IN  VLBIT;
        SIGNAL op               : OUT VLBIT_1D (95 DOWNTO 0);
        SIGNAL op_ack           : IN  VLBIT;
        SIGNAL op_req           : OUT VLBIT;
        SIGNAL r1_q_ack         : OUT VLBIT;
        SIGNAL r1_q             : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL r1_q_req         : IN  VLBIT;
        SIGNAL rf               : IN  VLBIT_1D (17 DOWNTO 0);
        SIGNAL rf_ack           : OUT VLBIT;
        SIGNAL rf_req           : IN  VLBIT;
        SIGNAL r02              : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL r03              : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL r04              : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL r05              : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL r06              : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL r07              : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL r08              : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL r09              : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL r10              : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL r11              : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL r12              : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL r13              : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL r14              : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL r15              : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL r16              : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL r17              : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL r18              : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL r19              : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL r20              : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL r21              : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL r22              : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL r23              : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL r24              : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL r25              : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL r26              : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL r27              : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL r28              : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL r29              : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL r30              : IN  VLBIT_1D (31 DOWNTO 0);
        SIGNAL r31              : IN  VLBIT_1D (31 DOWNTO 0) );
END regselect;
```

```
ARCHITECTURE behavior OF regselect IS
BEGIN
  main : PROCESS

    --- output signal value variables

    VARIABLE v_op_req          : VLBIT;
    VARIABLE v_r1_q_ack        : VLBIT;


    --- scratch variables

    VARIABLE sel               : INTEGER;
    VARIABLE tmpi              : INTEGER;
    VARIABLE tmpv              : VLBIT_1D (31 DOWNTO 0);
    VARIABLE op_a              : VLBIT_1D (31 DOWNTO 0);
    VARIABLE op_b              : VLBIT_1D (31 DOWNTO 0);
    VARIABLE op_s              : VLBIT_1D (31 DOWNTO 0);

    --- internal variables

    VARIABLE debug             : BOOLEAN := FALSE;


  BEGIN

    ---
    --- If clear is asserted, reset everything and stick here.
    ---

    IF clr /= '1' THEN                         -- reset to beginning

      v_op_req := '0';
      v_r1_q_ack := '0';

      rf_ack <= '0';
      op_req <= v_op_req;
      r1_q_ack <= v_r1_q_ack;
      op_s := ZERO(31 DOWNTO 0);
      op_a := ZERO(31 DOWNTO 0);
      op_b := ZERO(31 DOWNTO 0);
      op <= ( op_s & op_a & op_b );

      WAIT UNTIL clr = '1';                    -- wait till ready to run


    ELSE                                       -- unit is running

---- Normal running condition

      --- RF operand request queue has this format:
      ---
      ---    17   operand S requested
      --- 16-12   operand S register number
---    11   operand A requested
---  10-6   operand A register number
---     5   operand B requested
---   4-0   operand B register number
---

--- Operand A.

IF rf(11) = '1' THEN

  sel := v1d2int( rf(10 DOWNTO 6) );

  CASE sel IS
  WHEN 1 =>
    IF r1_q_req = v_r1_q_ack THEN         -- wait for it to be available
      WAIT UNTIL r1_q_req /= v_r1_q_ack;
    END IF;
    op_a := r1_q;                         -- get it
    v_r1_q_ack := r1_q_req;               -- ack it
    r1_q_ack <= v_r1_q_ack AFTER delay;
  WHEN 2 =>
    op_a := r02;
  WHEN 3 =>
    op_a := r03;
  WHEN 4 =>
    op_a := r04;
  WHEN 5 =>
    op_a := r05;
  WHEN 6 =>
    op_a := r06;
  WHEN 7 =>
    op_a := r07;
  WHEN 8 =>
    op_a := r08;
  WHEN 9 =>
    op_a := r09;
  WHEN 10 =>
    op_a := r10;
  WHEN 11 =>
    op_a := r11;
  WHEN 12 =>
    op_a := r12;
  WHEN 13 =>
    op_a := r13;
  WHEN 14 =>
    op_a := r14;
  WHEN 15 =>
    op_a := r15;
  WHEN 16 =>
    op_a := r16;
  WHEN 17 =>
    op_a := r17;
  WHEN 18 =>
    op_a := r18;
    WHEN 19 =>
      op_a := r19;
    WHEN 20 =>
      op_a := r20;
    WHEN 21 =>
      op_a := r21;
    WHEN 22 =>
      op_a := r22;
    WHEN 23 =>
      op_a := r23;
    WHEN 24 =>
      op_a := r24;
    WHEN 25 =>
      op_a := r25;
    WHEN 26 =>
      op_a := r26;
    WHEN 27 =>
      op_a := r27;
    WHEN 28 =>
      op_a := r28;
    WHEN 29 =>
      op_a := r29;
    WHEN 30 =>
      op_a := r30;
    WHEN 31 =>
      op_a := r31;
    WHEN OTHERS =>
      op_a := ZERO(31 DOWNTO 0);
    END CASE;

    IF debug THEN
      put("op_a <= r",sel);
      tmpi := v1d2int(op_a);
      puthexline(" = ",tmpi);
    END IF;

  END IF;


  --- Operand B.

  IF rf(5) = '1' THEN

    sel := v1d2int( rf(4 DOWNTO 0) );

    CASE sel IS
    WHEN 1 =>
      IF r1_q_req = v_r1_q_ack THEN         -- wait for it to be available
        WAIT UNTIL r1_q_req /= v_r1_q_ack;
      END IF;
      op_b := r1_q;                         -- get it
      v_r1_q_ack := r1_q_req;               -- ack it
      r1_q_ack <= v_r1_q_ack AFTER delay;
    WHEN 2 =>
    op_b := r02;
  WHEN 3 =>
    op_b := r03;
  WHEN 4 =>
    op_b := r04;
  WHEN 5 =>
    op_b := r05;
  WHEN 6 =>
    op_b := r06;
  WHEN 7 =>
    op_b := r07;
  WHEN 8 =>
    op_b := r08;
  WHEN 9 =>
    op_b := r09;
  WHEN 10 =>
    op_b := r10;
  WHEN 11 =>
    op_b := r11;
  WHEN 12 =>
    op_b := r12;
  WHEN 13 =>
    op_b := r13;
  WHEN 14 =>
    op_b := r14;
  WHEN 15 =>
    op_b := r15;
  WHEN 16 =>
    op_b := r16;
  WHEN 17 =>
    op_b := r17;
  WHEN 18 =>
    op_b := r18;
  WHEN 19 =>
    op_b := r19;
  WHEN 20 =>
    op_b := r20;
  WHEN 21 =>
    op_b := r21;
  WHEN 22 =>
    op_b := r22;
  WHEN 23 =>
    op_b := r23;
  WHEN 24 =>
    op_b := r24;
  WHEN 25 =>
    op_b := r25;
  WHEN 26 =>
    op_b := r26;
  WHEN 27 =>
    op_b := r27;
  WHEN 28 =>
    op_b := r28;
  WHEN 29 =>
```

```
          op_b := r29;
        WHEN 30 =>
          op_b := r30;
        WHEN 31 =>
          op_b := r31;
        WHEN OTHERS =>
          op_b := ZERO(31 DOWNTO 0);
        END CASE;

        IF debug THEN
          put("op_b <= r",sel);
          tmpi := v1d2int(op_b);
          puthexline(" = ",tmpi);
        END IF;

      END IF;

      --- Operand S.

      IF rf(17) = '1' THEN

        sel := v1d2int( rf(16 DOWNTO 12) );

        CASE sel IS
        WHEN 1 =>
          IF r1_q_req = v_r1_q_ack THEN         -- wait for it to be available
            WAIT UNTIL r1_q_req /= v_r1_q_ack;
          END IF;
          op_s := r1_q;                         -- get it
          v_r1_q_ack := r1_q_req;               -- ack it
          r1_q_ack <= v_r1_q_ack AFTER delay;
        WHEN 2 =>
          op_s := r02;
        WHEN 3 =>
          op_s := r03;
        WHEN 4 =>
          op_s := r04;
        WHEN 5 =>
          op_s := r05;
        WHEN 6 =>
          op_s := r06;
        WHEN 7 =>
          op_s := r07;
        WHEN 8 =>
          op_s := r08;
        WHEN 9 =>
          op_s := r09;
        WHEN 10 =>
          op_s := r10;
        WHEN 11 =>
          op_s := r11;
        WHEN 12 =>
          op_s := r12;
        WHEN 13 =>
      op_s := r13;
    WHEN 14 =>
      op_s := r14;
    WHEN 15 =>
      op_s := r15;
    WHEN 16 =>
      op_s := r16;
    WHEN 17 =>
      op_s := r17;
    WHEN 18 =>
      op_s := r18;
    WHEN 19 =>
      op_s := r19;
    WHEN 20 =>
      op_s := r20;
    WHEN 21 =>
      op_s := r21;
    WHEN 22 =>
      op_s := r22;
    WHEN 23 =>
      op_s := r23;
    WHEN 24 =>
      op_s := r24;
    WHEN 25 =>
      op_s := r25;
    WHEN 26 =>
      op_s := r26;
    WHEN 27 =>
      op_s := r27;
    WHEN 28 =>
      op_s := r28;
    WHEN 29 =>
      op_s := r29;
    WHEN 30 =>
      op_s := r30;
    WHEN 31 =>
      op_s := r31;
    WHEN OTHERS =>
      op_s := ZERO(31 DOWNTO 0);
    END CASE;

    IF debug THEN
      put("op_s <= r",sel);
      tmpi := v1d2int(op_s);
      puthexline(" = ",tmpi);
    END IF;

  END IF;


  op <= ( op_s & op_a & op_b );             -- send operands out
  v_op_req := NOT v_op_req;
  op_req <= v_op_req AFTER delay;
```

```
      WAIT ON op_ack;                         -- wait for ACK

      rf_ack <= rf_req AFTER delay;           -- done with this event

    END IF;                                   -- CLR not asserted


    WAIT ON clr, rf_req;

  END PROCESS main;

END behavior;
```

2.25 scoreboard.vhd

```
---- The scoreboard is a shared resource, so it is necessary for read and write
---- access to be mutually exclusive.  This module contains the scoreboard
---- values, and enforces the exclusion.
----
---- Read operations are three-phase: a transition on RD_R requests a lock on
---- the register, and is granted by a transition on RD_G, indicating that the
---- OUT data is valid.  The lock is released by a transition on RD_D.  While
---- the read lock is granted, the BITS output is stable and valid.
----
---- Write operations are two-phase.  The bit number of placed on the NUM input
---- bus, the value is placed on the SET port, and a transition is sent to
---- WR_R.  A transition on WR_A indicates that the write is completed.


ENTITY scoreboard IS
  GENERIC(delay: TIME := 100 ps );
  PORT( SIGNAL clr         : IN  VLBIT;
        SIGNAL rd_r        : IN  VLBIT;
        SIGNAL rd_g        : OUT VLBIT;
        SIGNAL rd_d        : IN  VLBIT;
        SIGNAL wr_r        : IN  VLBIT;
        SIGNAL wr_a        : OUT VLBIT;
        SIGNAL set         : IN  VLBIT;
        SIGNAL num         : IN  VLBIT_1D (4 DOWNTO 0);
        SIGNAL bits        : OUT VLBIT_1D (31 DOWNTO 0));
END scoreboard;


ARCHITECTURE state_behavior OF scoreboard IS
  TYPE states IS (wait_any,wait_done,wait_done_write);
BEGIN
  main : PROCESS ( rd_r, wr_r, rd_d, clr )
    VARIABLE  state        : states := wait_any;
    VARIABLE  v_bits       : VLBIT_1D (31 DOWNTO 0);
    VARIABLE  tmpi         : INTEGER;
    VARIABLE  s_reads      : INTEGER;
    VARIABLE  s_sets       : INTEGER;
    VARIABLE  s_clears     : INTEGER;
    VARIABLE  s_conflicts  : INTEGER;
  BEGIN

    IF ( clr /= '1' ) THEN

      state := wait_any;
      v_bits := ( '0','0','0','0','0','0','0','0',
                  '0','0','0','0','0','0','0','0',
                  '0','0','0','0','0','0','0','0',
                  '0','0','0','0','0','0','0','0' );
      bits <= v_bits;
      rd_g <= '0';
      wr_a <= '0';
      s_reads := 0;
      s_sets := 0;
      s_clears := 0;
      s_conflicts := 0;

    ELSE

      CASE state IS

      WHEN wait_any =>                         -- waiting for any request
        IF (rd_r'EVENT AND wr_r'EVENT) THEN    -- got both, do read first
          rd_g <= rd_r AFTER delay;            -- grant read permission
          state := wait_done_write;            -- and wait for it to finish
          s_reads := s_reads + 1;
          s_conflicts := s_conflicts + 1;
        ELSIF (rd_r'EVENT) THEN                -- else got rd_r only
          rd_g <= rd_r AFTER delay;            -- send grant signal
          state := wait_done;                  -- wait for release
          s_reads := s_reads + 1;
        ELSIF (wr_r'EVENT) THEN                -- or got wr_r only
          tmpi := v1d2int(num);                -- see which bit to write
          v_bits(tmpi) := set;                 -- change it
          bits <= v_bits;                      -- and update output
          wr_a <= wr_r AFTER delay;            -- and return ack
          IF ( set = '1' ) THEN
            s_sets := s_sets + 1;
          ELSE
            s_clears := s_clears + 1;
          END IF;
        END IF;

      WHEN wait_done =>                        -- waiting for rd_d
        IF (rd_d'EVENT AND wr_r'EVENT) THEN    -- got both, done with read
          tmpi := v1d2int(num);                -- see which bit to write
          v_bits(tmpi) := set;                 -- change it
          bits <= v_bits;                      -- and update output
          wr_a <= wr_r AFTER delay;            -- and return ack
          state := wait_any;                   -- wait for more stuff
          IF ( set = '1' ) THEN
            s_sets := s_sets + 1;
          ELSE
            s_clears := s_clears + 1;
          END IF;
        ELSIF (rd_d'EVENT) THEN                -- else got rd_d only
          state := wait_any;                   -- wait for more stuff
        ELSIF (wr_r'EVENT) THEN                -- else got wr_r only
          state := wait_done_write;            -- wait for release
          s_conflicts := s_conflicts + 1;
        END IF;

      WHEN wait_done_write =>                  -- wait for rd_d, then write
        IF (rd_d'EVENT) THEN                   -- got rd_d, done with read
          tmpi := v1d2int(num);                -- see which bit to write
          v_bits(tmpi) := set;                 -- change it
          bits <= v_bits;                      -- and update output
          wr_a <= wr_r AFTER delay;            -- and return ack
          state := wait_any;                   -- wait for more stuff
          IF ( set = '1' ) THEN
            s_sets := s_sets + 1;
          ELSE
            s_clears := s_clears + 1;
          END IF;
        END IF;

      END CASE;                                --  state

    END IF;

  END PROCESS main;
END state_behavior;
```

2.26 vdelay.vhd

```
ENTITY vdelay IS
  GENERIC( delay: TIME := 100 PS );
  PORT (SIGNAL i :IN    VLBIT;
        SIGNAL o :OUT   VLBIT := '0');
END vdelay;

ARCHITECTURE behavior OF vdelay IS
BEGIN
  main : PROCESS ( i )
    BEGIN
      o <= i AFTER delay;
    END PROCESS main;
END behavior;
```

2.27 verify_address.vhd

```
--- This verifies memory addresses.  All it does it check to see if the
--- byte-address is greater than the limit, and if so it sets bit 0 in
--- the status output.
---
--- In an actual memory system, this could signal cache misses, write
--- protection, segment violation, and lots of other things.  For my
--- purposes I just need some kind of memory fault to test.
---
--- Actually, there is some special behavior. This unit intercepts memory
--- requests, so that it can implement some special I/O functions. Any memory
--- access with address bit 31 set is used for some special I/O behavior.  This
--- behavior is not part of the Fred architecture. I just added it so that I'd
--- have a simple way to generate circuit events from within an assembly
--- language program.  I can use this to read and write from files, or to
--- wiggle a debugging pin.
---
ENTITY verify_address IS
  GENERIC( delay: TIME := 100 PS;
           maxaddr: INTEGER := 65535 );
  PORT (SIGNAL clr      :IN    VLBIT;
        SIGNAL reqi     :IN    VLBIT;
        SIGNAL addr     :IN    VLBIT_VECTOR (31 downto 2);
        SIGNAL acki     :IN    VLBIT;
        SIGNAL reqo     :OUT   VLBIT;
        SIGNAL acko     :OUT   VLBIT := '0';
        SIGNAL s        :OUT   VLBIT_VECTOR (3 downto 0) := ('0','0','0','0');
        SIGNAL wdata    :IN    VLBIT_1D(31 DOWNTO 0); -- write input data
        SIGNAL read     :IN    VLBIT;           -- read/write direction
        SIGNAL mdata    :IN    VLBIT_1D(31 DOWNTO 0); -- memory read data in
        SIGNAL xdata    :OUT   VLBIT_1D(31 DOWNTO 0); -- memory read data out
        SIGNAL x_debug  :OUT   VLBIT;           -- for debugging only
        SIGNAL y_debug  :OUT   VLBIT);          -- for debugging only
END verify_address;

ARCHITECTURE behavior OF verify_address IS
BEGIN
  main : PROCESS
      VARIABLE  num            : INTEGER := 0;
      VARIABLE  tmpi           : INTEGER;
      VARIABLE  v_reqo         : VLBIT;
      VARIABLE  v_x_debug      : VLBIT;
      VARIABLE  v_y_debug      : VLBIT;
      VARIABLE  bo             : BOOLEAN;
      VARIABLE  rt             : TIME;
      FILE      stdin          : TEXT IS IN "Stdin.dat";
      FILE      stdout         : TEXT IS OUT "Stdout.dat";
      FILE      rtdelay        : TEXT IS IN "Random.dat";
      VARIABLE  trace_out      : BOOLEAN := FALSE; -- display output
      VARIABLE  trace_in       : BOOLEAN := FALSE; -- display output
    BEGIN

      IF clr /= '1' THEN                        -- clear is asserted
    num := 0;                               -- reset outputs
    acko <= '0';
    v_reqo := '0';
    reqo <= v_reqo;
    s <= ('0','0','0','0');
    v_x_debug := '0';
    x_debug <= v_x_debug;
    v_y_debug := '0';
    y_debug <= v_y_debug;

    WAIT ON reqi, acki, clr;                -- wait for inputs

  ELSE                                      -- running normally

    WAIT ON reqi;                           -- wait for REQI event

    num := v1d2int( addr ) * 4;             -- get byte-address value

    IF addr(31) = '1' THEN                  -- special syscall behavior

      num := v1d2int( addr(31 DOWNTO 28) ); -- get syscall number

      CASE num IS

      WHEN 8 =>                             -- 0x80000000 is times()

        v_x_debug := NOT v_x_debug;         -- just wiggle a pin
        x_debug <= v_x_debug;


      WHEN 9 =>                             -- 0x90000000 is interrupts

        --- Anything to this address will generate an external interrupt
        --- after an amount of time determined by reading the delay from
        --- an external file.  There should be enough random numbers in the
        --- file to account for the needs of any simulation run.

        freadline( rtdelay, tmpi, bo );     -- try to read it

        ASSERT bo REPORT "Couldn't read random.dat" SEVERITY FAILURE;

        rt := 1 NS * tmpi;                  -- determine the time

        putline("generating interrupt in ",rt);

        v_y_debug := NOT v_y_debug;
        y_debug <= v_y_debug AFTER rt;


      WHEN 12 =>                            -- 0xC0000000 is file I/O

        --- The data that is read or written appears in the file as decimal
        --- integers, one per line. The value read is a signed 32-bit
        --- quantity.  I ignore the byte strobes and read or write the
        --- entire 32-bit value.  I essentially treat the file access as
        --- the implementation of the getc() and putc() functions.

        IF read = '1' THEN                  -- read a byte from the file

          freadline( stdin, tmpi, bo );     -- try to read it

          IF bo THEN                        -- it worked
            xdata <= int2v1d( tmpi );       -- so return the byte value
          ELSE
            xdata <= int2v1d( -1 );         -- return EOF on failure
          END IF;

          IF (trace_in) THEN
            putline("STDIN = ",tmpi);
          END IF;

        ELSE                                -- write a byte to the file

          tmpi := v1d2int( wdata(7 DOWNTO 0) ); -- this value

          fwriteline( stdout, tmpi, bo );   -- write it and make sure

          ASSERT bo REPORT "Couldn't write stdout" SEVERITY FAILURE;

          IF (trace_out) THEN
            putline("STDOUT = ",tmpi);
          END IF;

        END IF;

      WHEN OTHERS =>

        putline("Unimplemented system call ",num);

      END CASE;


      s <= ('0','0','0','0');               -- memory status is okay

      acko <= reqi AFTER delay;             -- send ACKO


    ELSIF ( num > maxaddr ) THEN            -- invalid address

      s <= ('0','0','0','1');

      acko <= reqi AFTER delay;             -- report memory status

    ELSE                                    -- valid address

      --- Pass request on to memory immediately. The zero delay is not
      --- realistic, but there's already enough delay in the memory system
```

```
          --- so that this doesn’t make any difference.

          v_reqo := NOT v_reqo;
          reqo <= v_reqo;

          s <= (’0’,’0’,’0’,’0’);               -- memory status is okay

          WAIT ON acki;                         -- wait for ACKI

          xdata <= mdata;                       -- pass data through, too

          --- This would be the perfect place to put in some unexpected random
          --- delay to model a cache miss.

          acko <= reqi;                         -- send ACKO


        END IF;

      END IF;

    END PROCESS main;
END behavior;
```

2.28 zdelay.vhd

```
----
---- This is a simple delay module.  The delay time can be changed at
---- simulation run-time. The default delay is zero.
----

ENTITY zdelay IS
  PORT (SIGNAL i :IN    VLBIT;
        SIGNAL o :OUT   VLBIT := '0');
END zdelay;

ARCHITECTURE behavior OF zdelay IS
BEGIN
  main : PROCESS ( i )

    VARIABLE delay : TIME := 0 NS;

    BEGIN
      o <= i AFTER delay;
    END PROCESS main;
END behavior;
```

2.29 zxor.vhd

```
----
---- This is an XOR with no delay.  It is the self-timed equivalent of a
---- wire-or.
----

ENTITY zxor IS
  PORT (SIGNAL a :IN    VLBIT;
        SIGNAL b :IN    VLBIT;
        SIGNAL y :OUT   VLBIT := '0');
END zxor;

ARCHITECTURE behavior OF zxor IS
BEGIN
  main : PROCESS ( a, b )

    VARIABLE delay : TIME := 0 NS;

    BEGIN
      y <= a XOR b AFTER delay;
    END PROCESS main;
END behavior;
```

Chapter 3

Schematic Diagrams

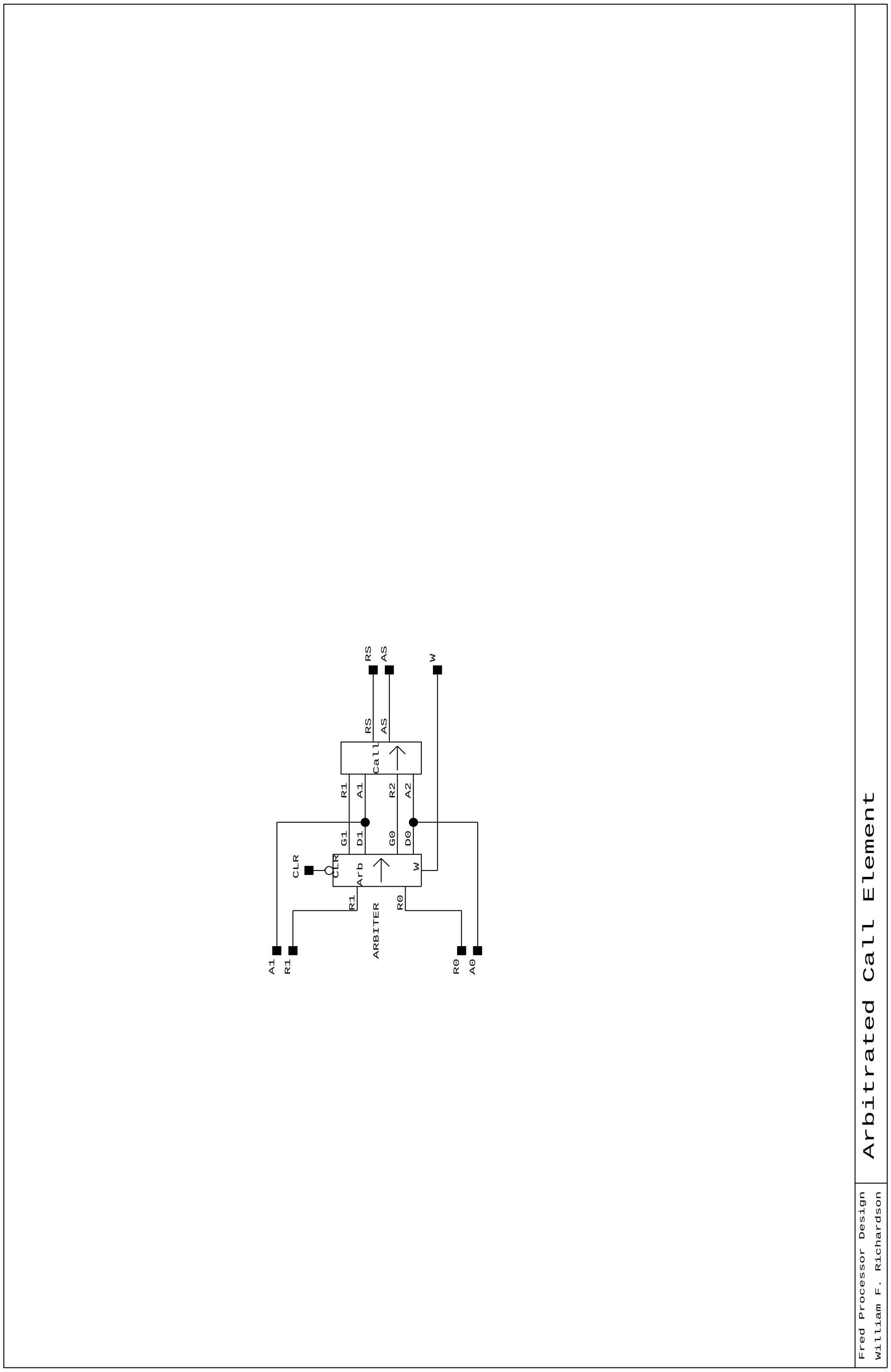


Figure 3.1: arbcall

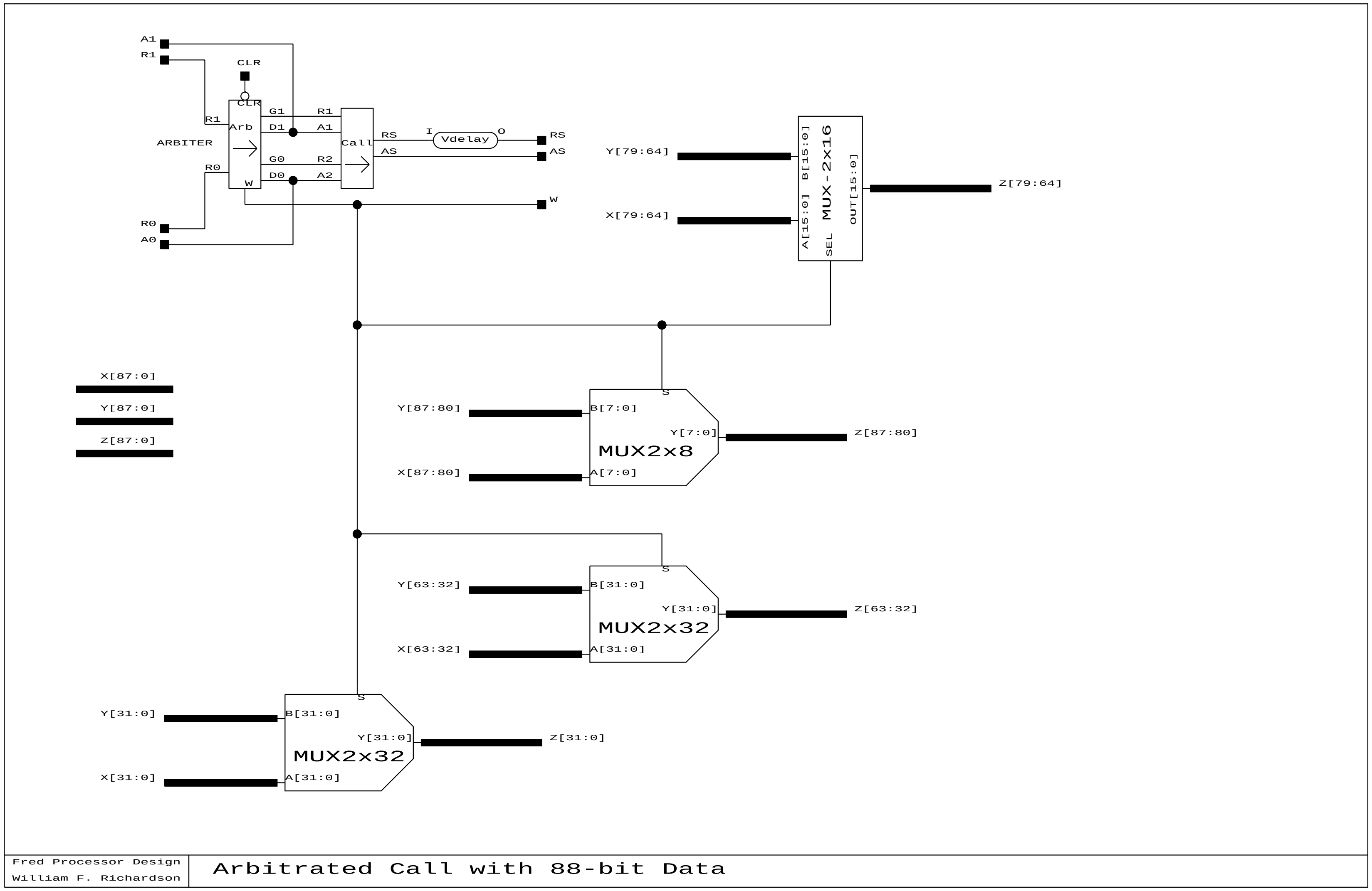


Figure 3.2: arbcallx88

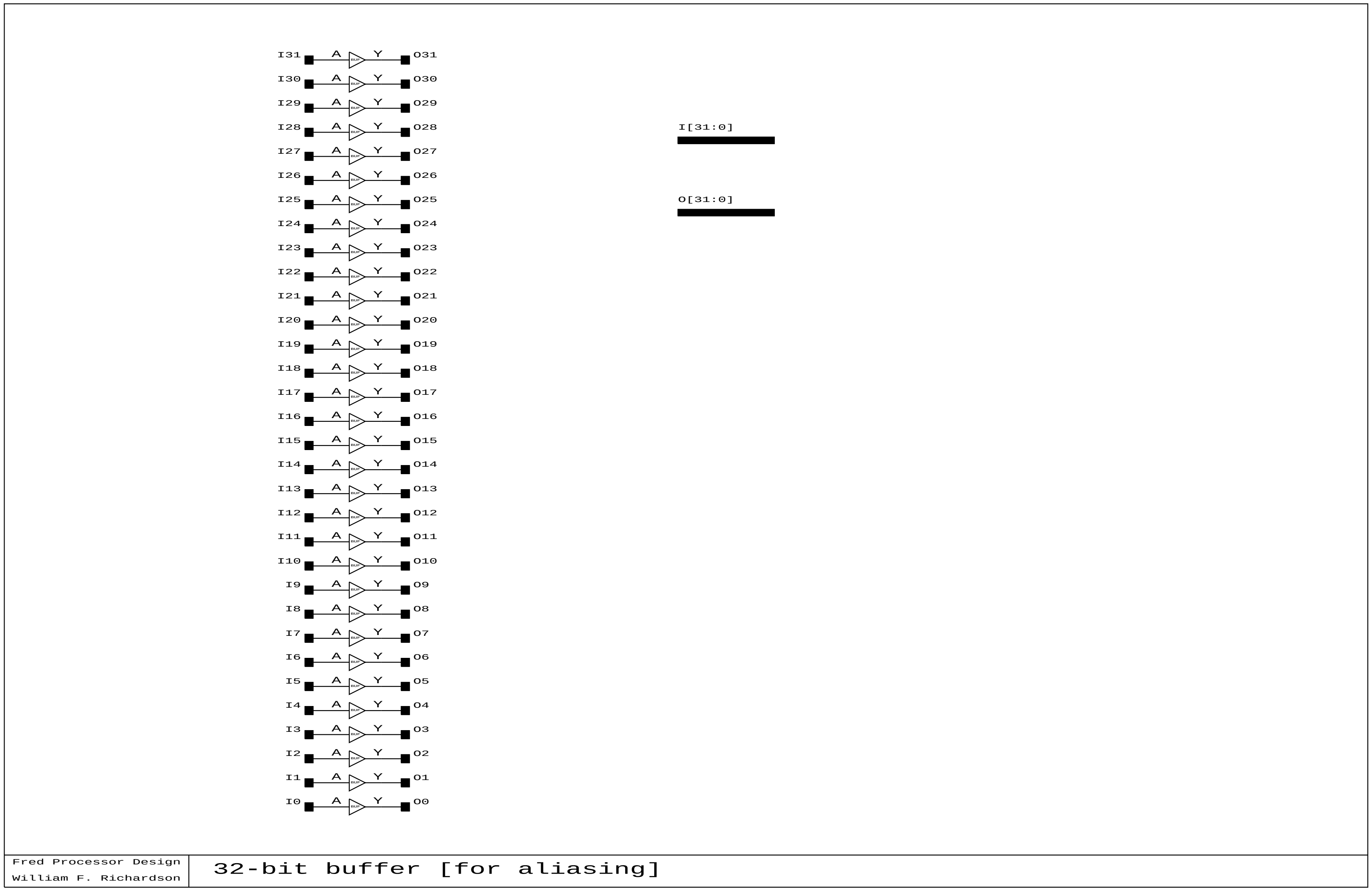
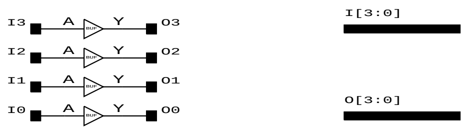


Figure 3.3: buf32



Fred Processor Design
William F. Richardson

4-bit buffer [for aliasing]

Figure 3.4: buf4

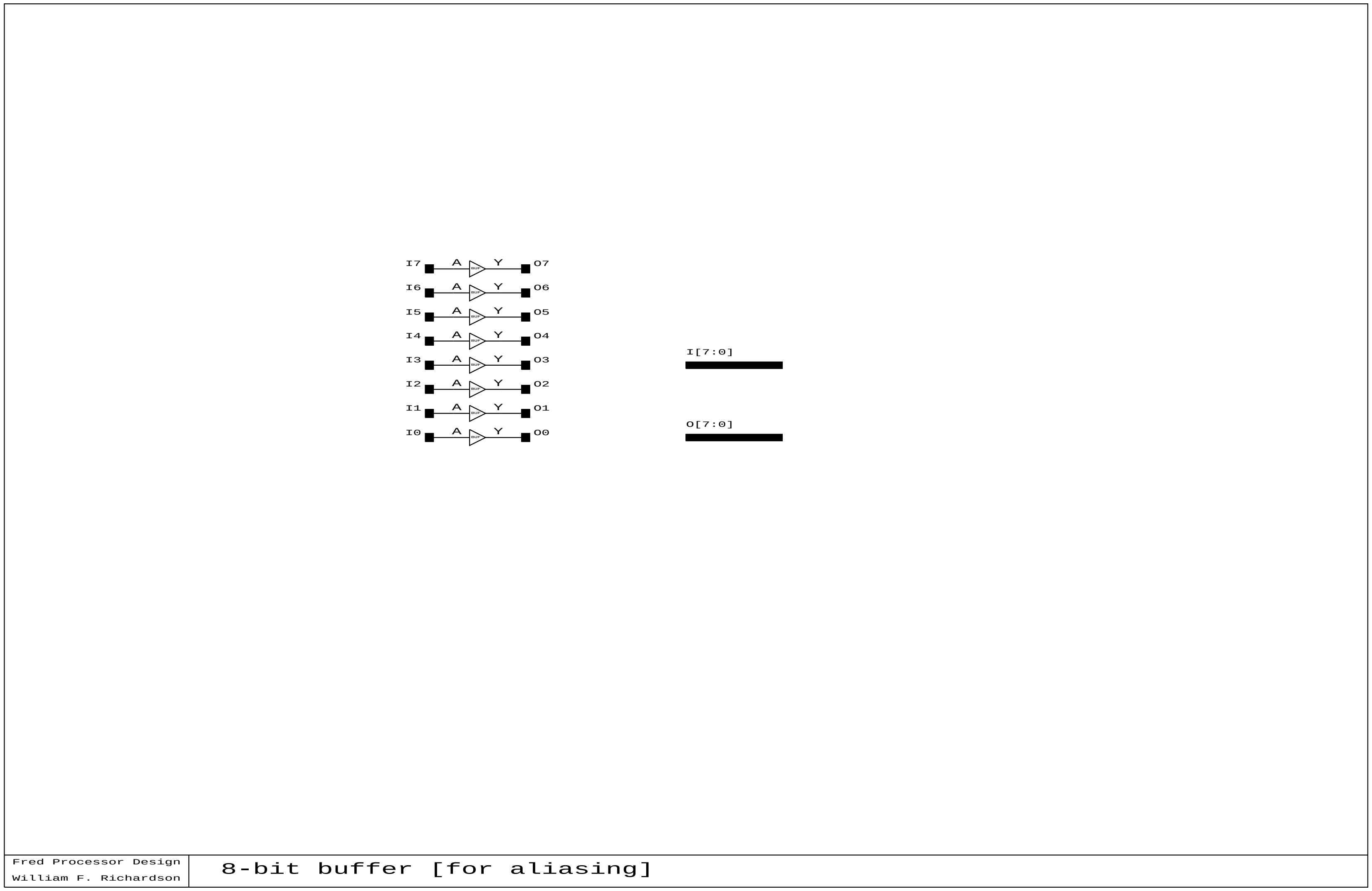


Figure 3.5: buf8

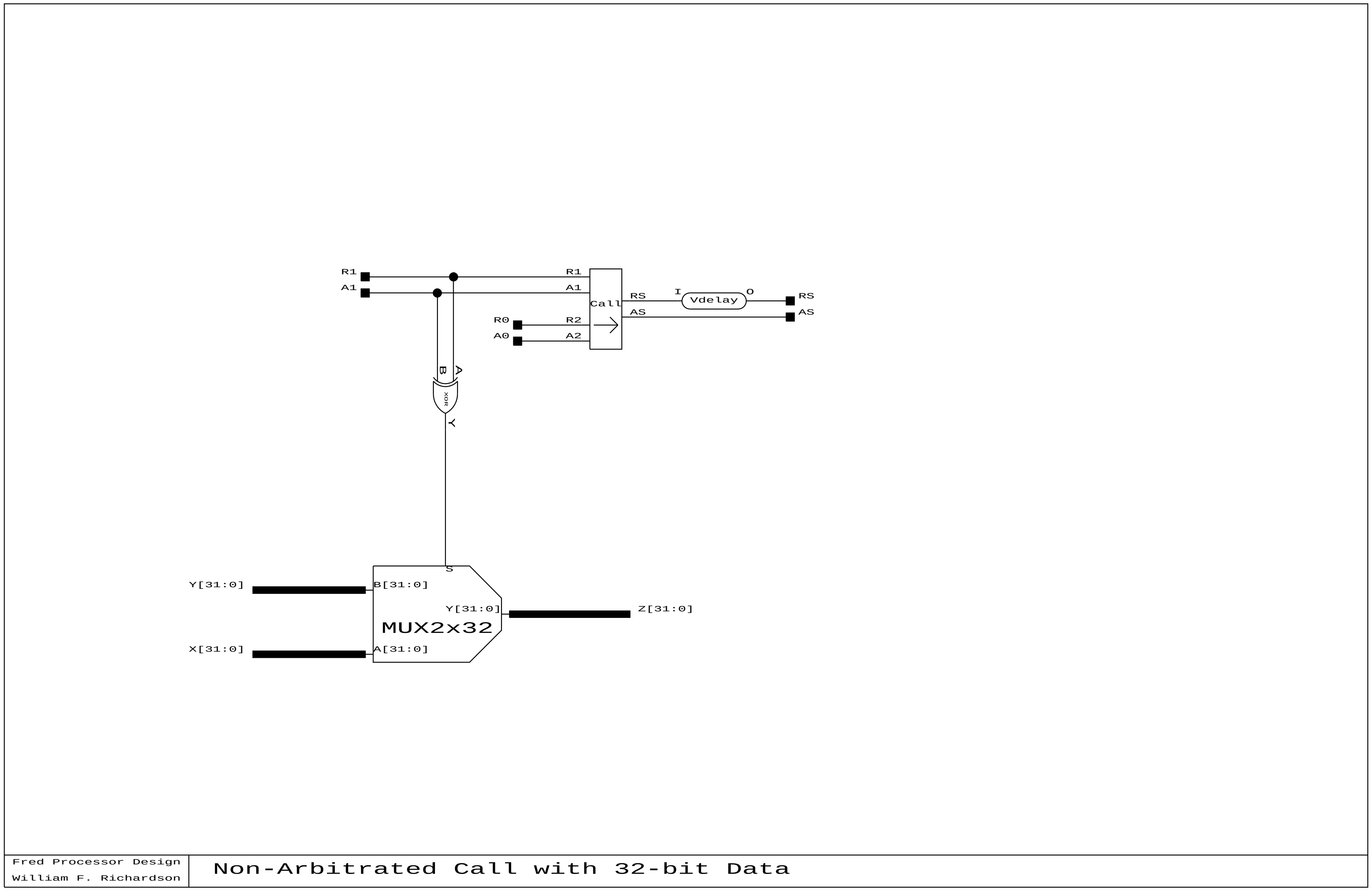
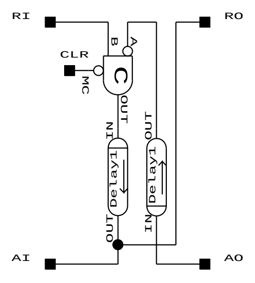


Figure 3.6: callx32



This is used to model the delay of a Capture/Pass latch.
It does not actually latch any data on its own.

Fred Processor Design
William F. Richardson

Capture-Pass latch delay circuit.

Figure 3.7: cp-delay

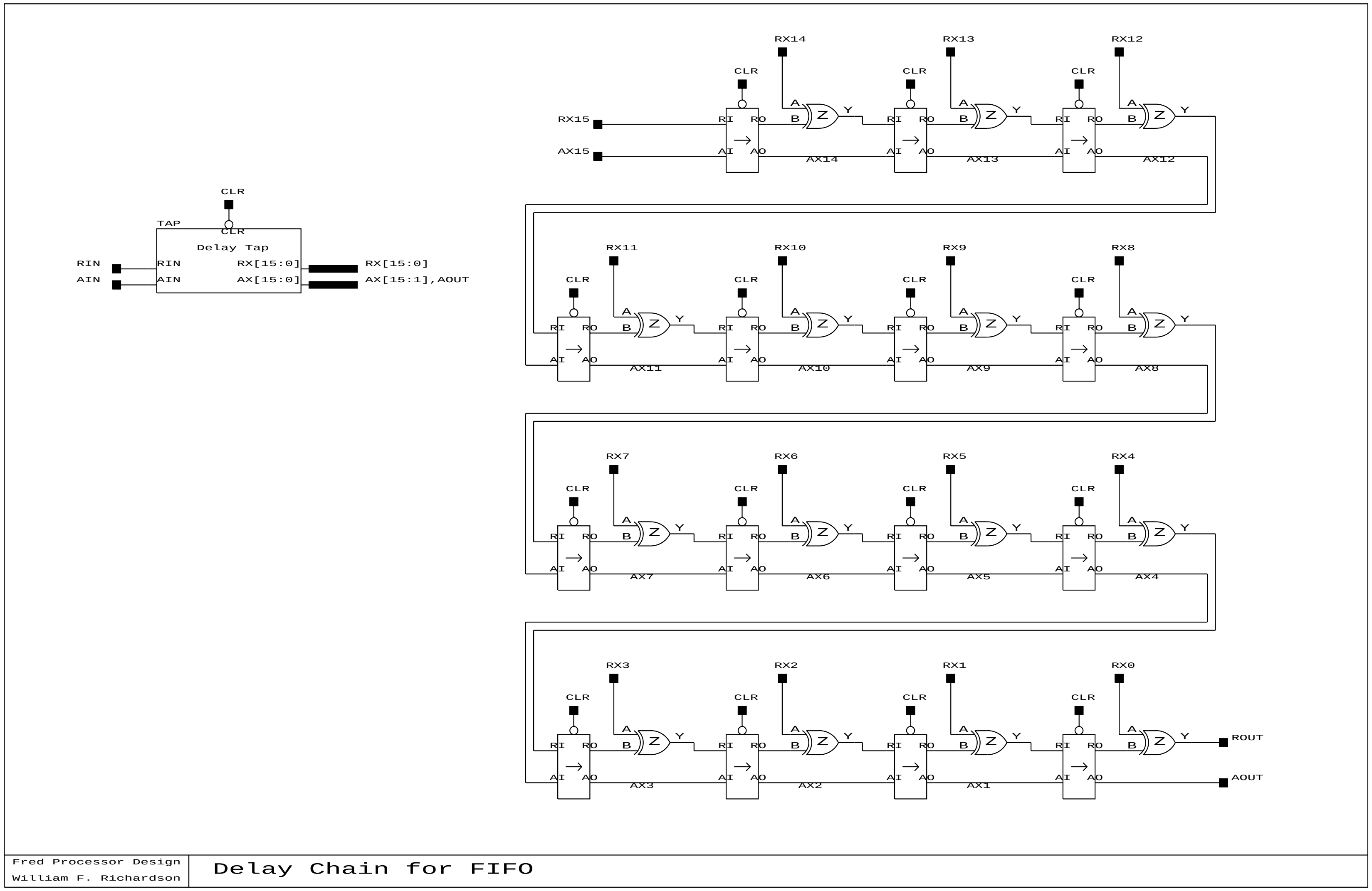


Figure 3.8: delaychain

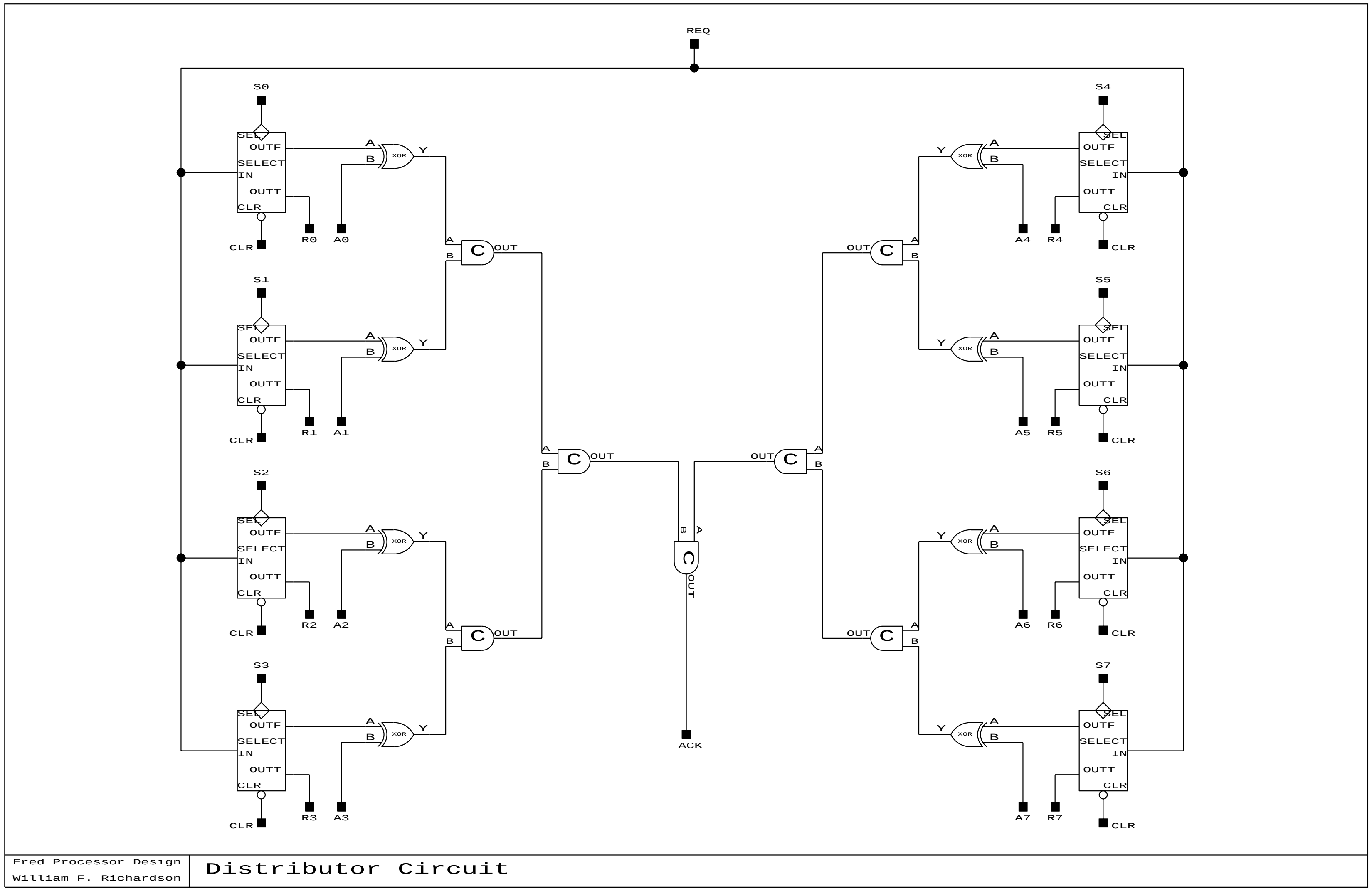


Figure 3.9: distributor

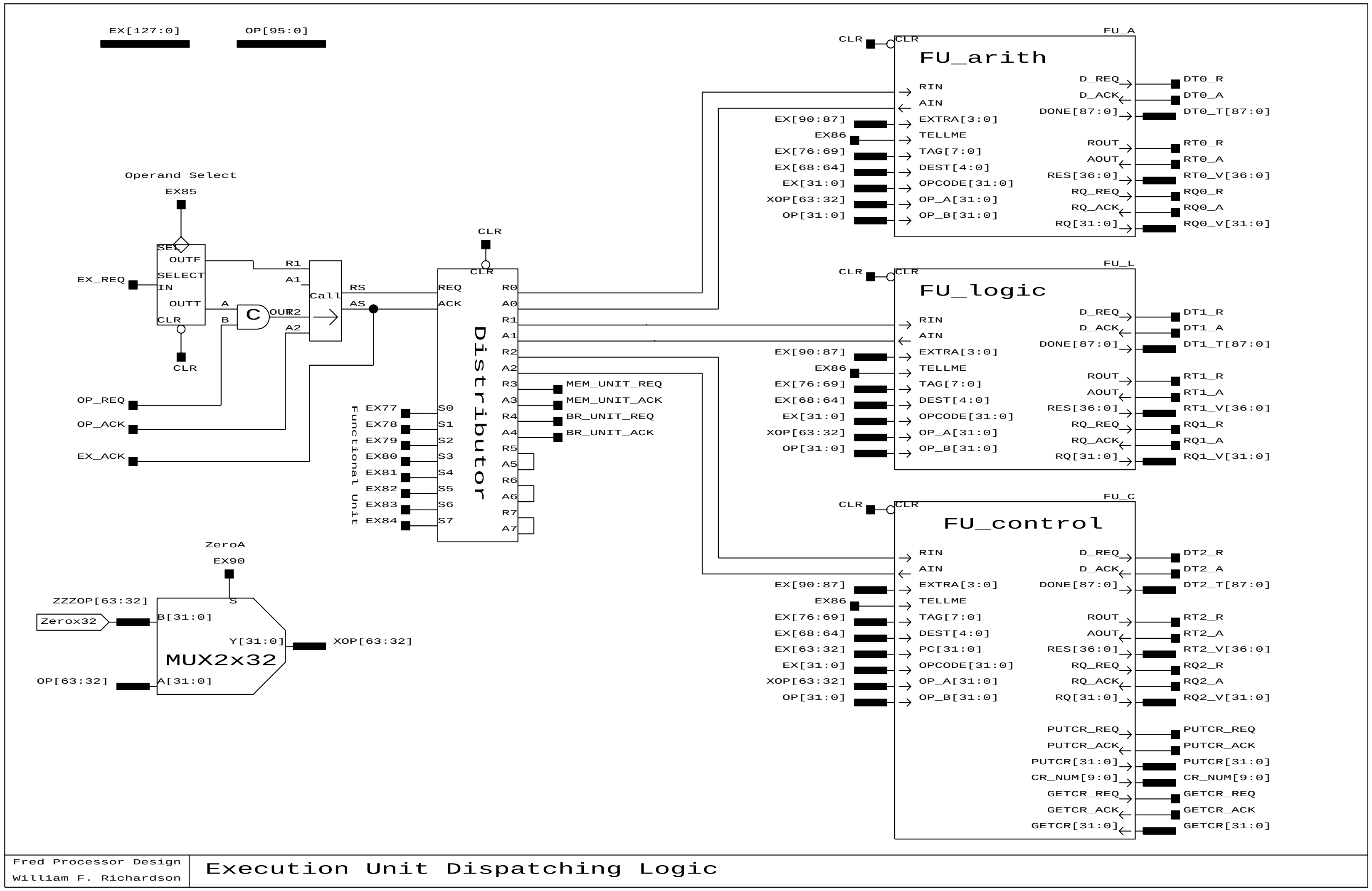


Figure 3.10: ex-unit1

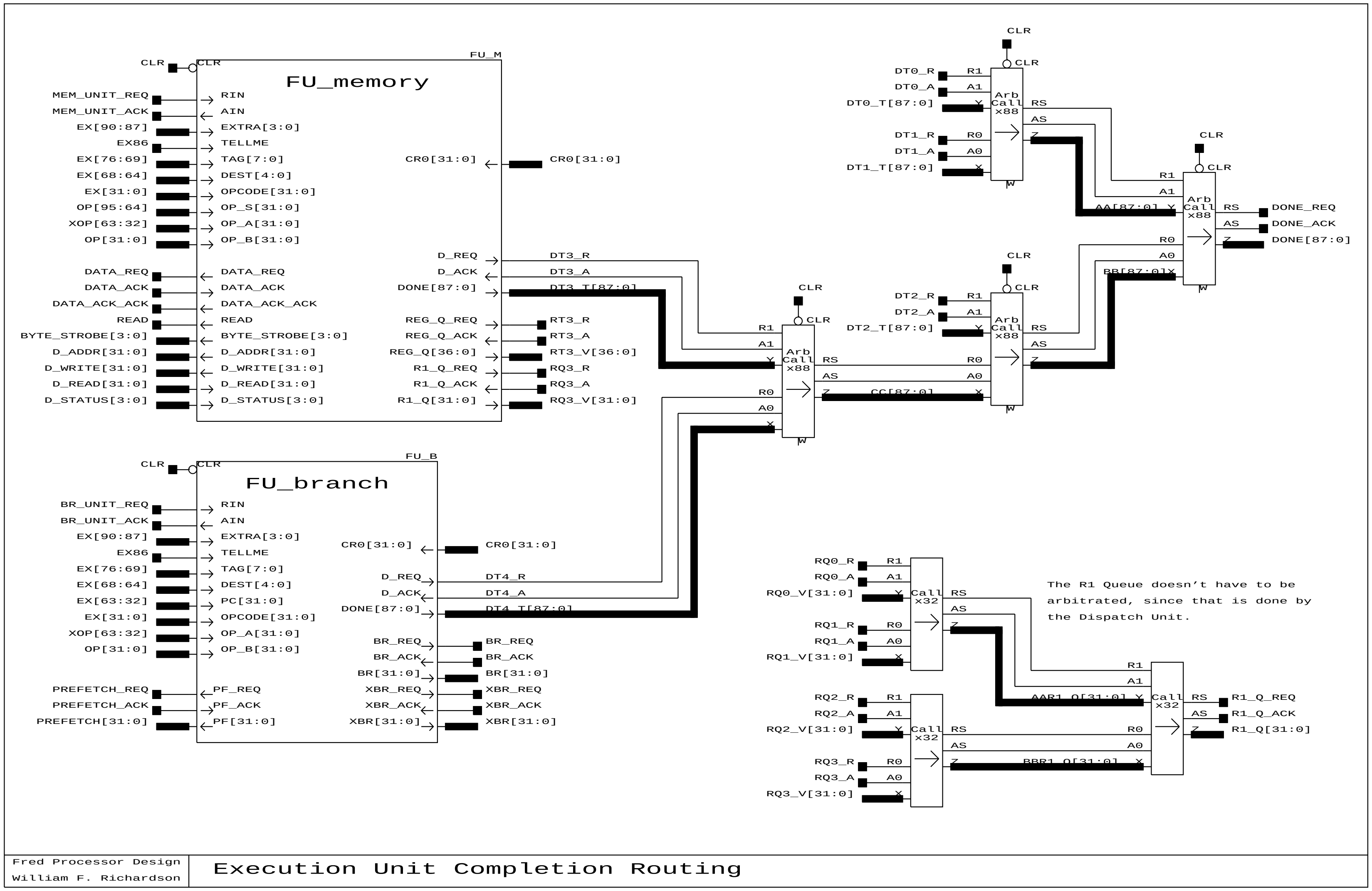
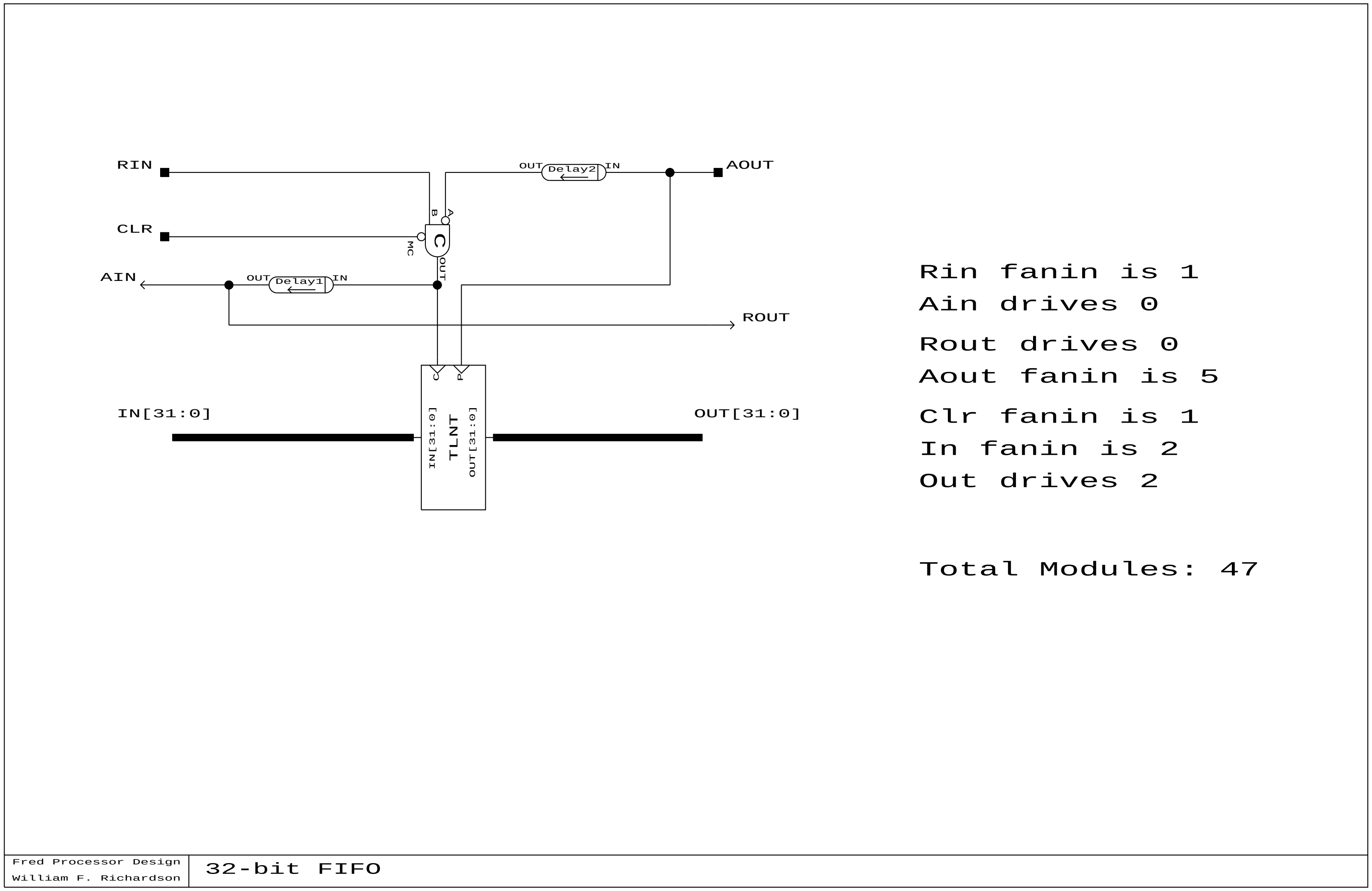


Figure 3.11: ex-unit2



Rin fanin is 1
Ain drives 0
Rout drives 0
Aout fanin is 5
Clr fanin is 1
In fanin is 2
Out drives 2

Total Modules: 47

Fred Processor Design
William F. Richardson

32-bit FIFO

Figure 3.12: fifo-32

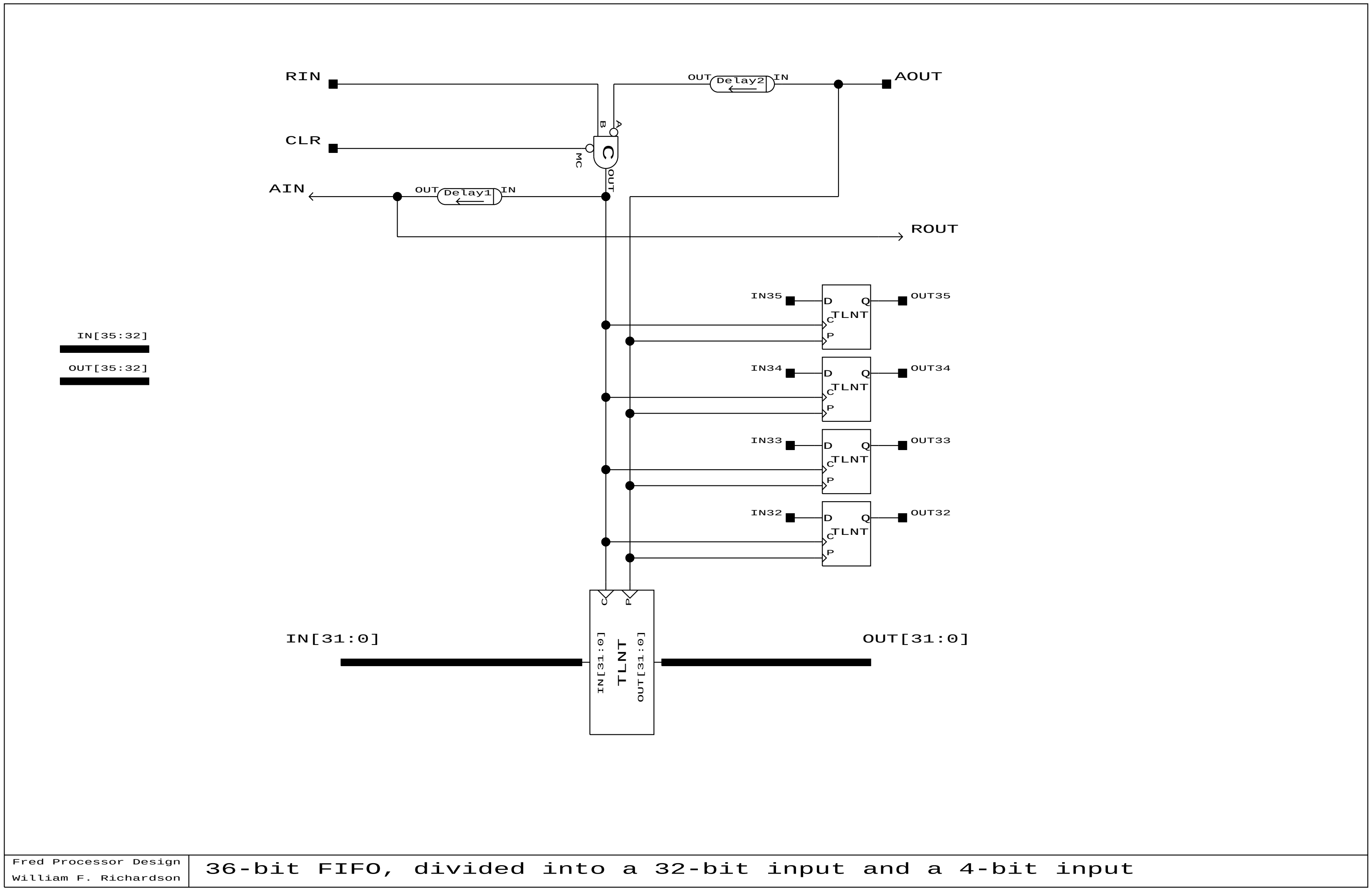
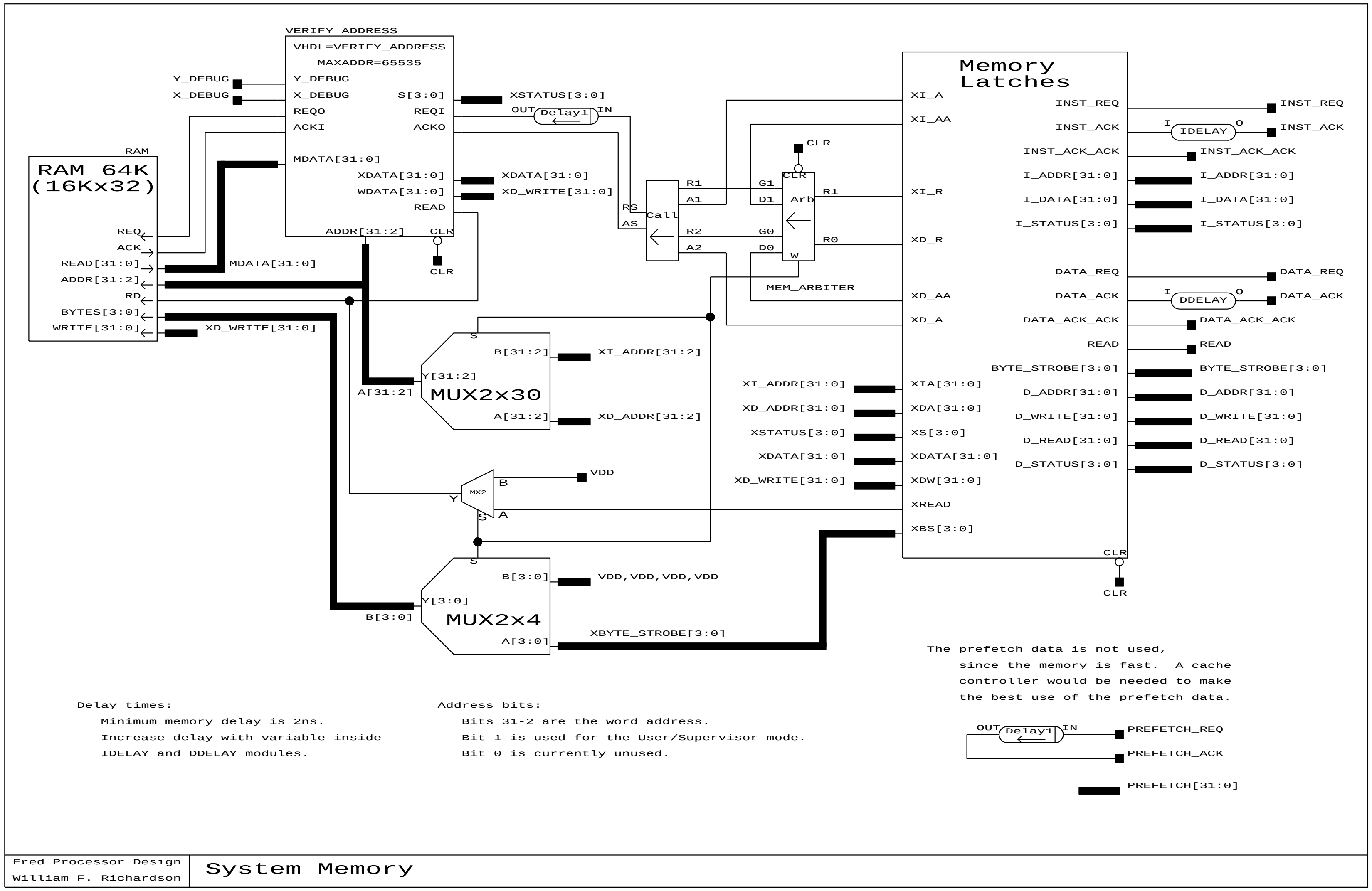


Figure 3.13: fifo-36



Delay times:
Minimum memory delay is 2ns.
Increase delay with variable inside IDELAY and DDELAY modules.

Address bits:
Bits 31-2 are the word address.
Bit 1 is used for the User/Supervisor mode.
Bit 0 is currently unused.

The prefetch data is not used, since the memory is fast. A cache controller would be needed to make the best use of the prefetch data.

Fred Processor Design
William F. Richardson

System Memory

Figure 3.14: fred-memory

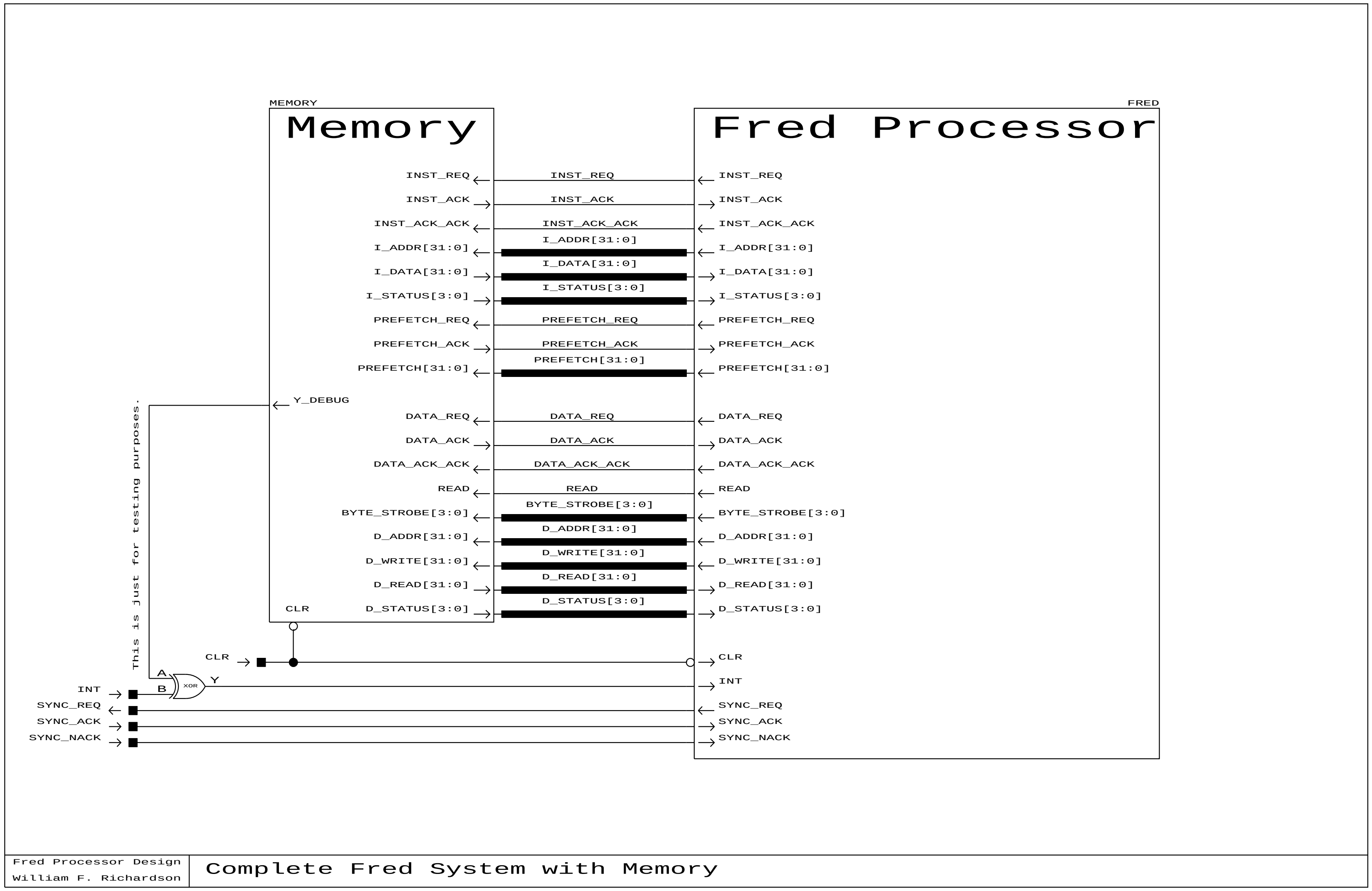


Figure 3.15: fred-with-mem

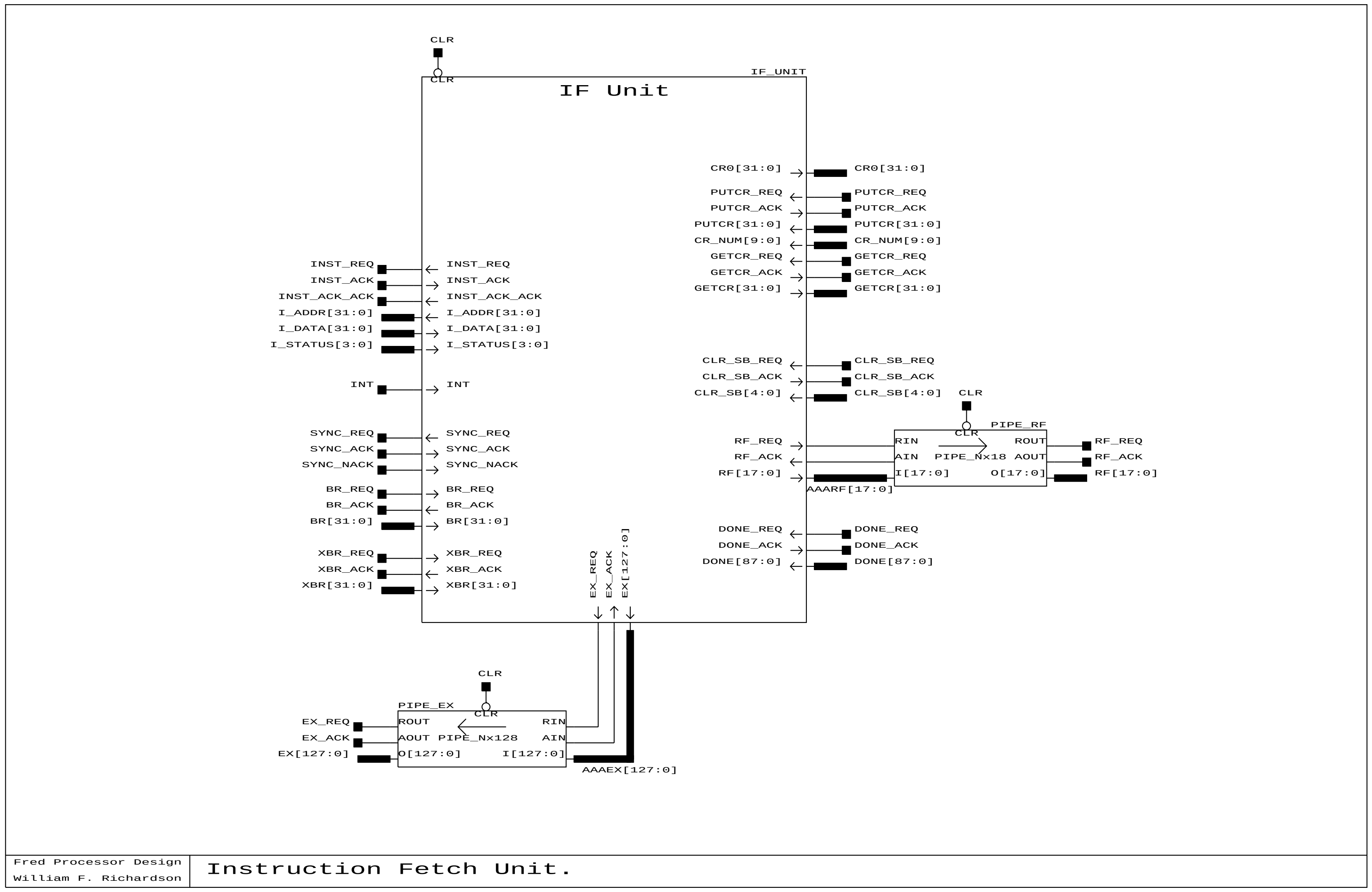


Figure 3.16: fred1

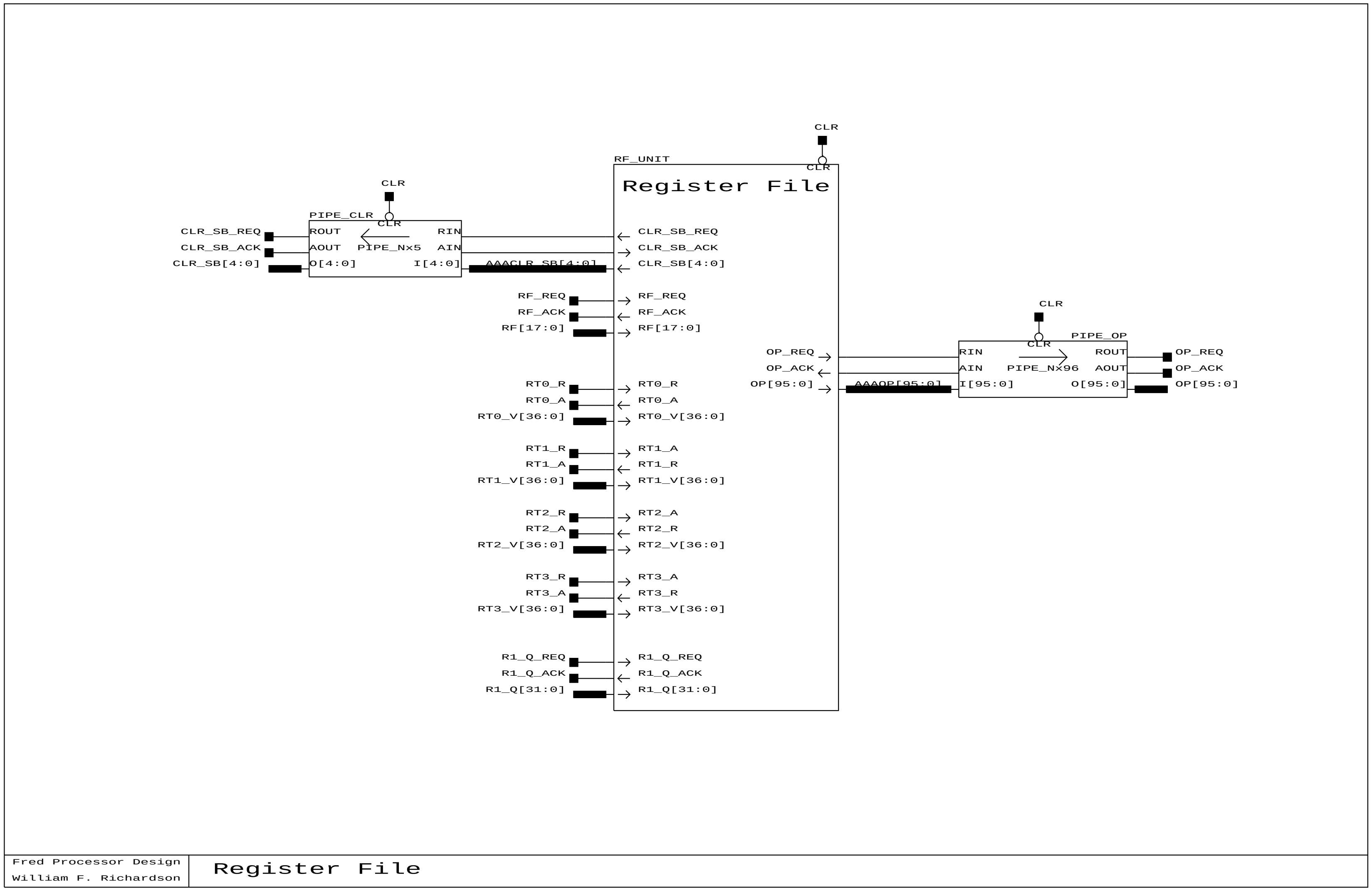


Figure 3.17: fred2

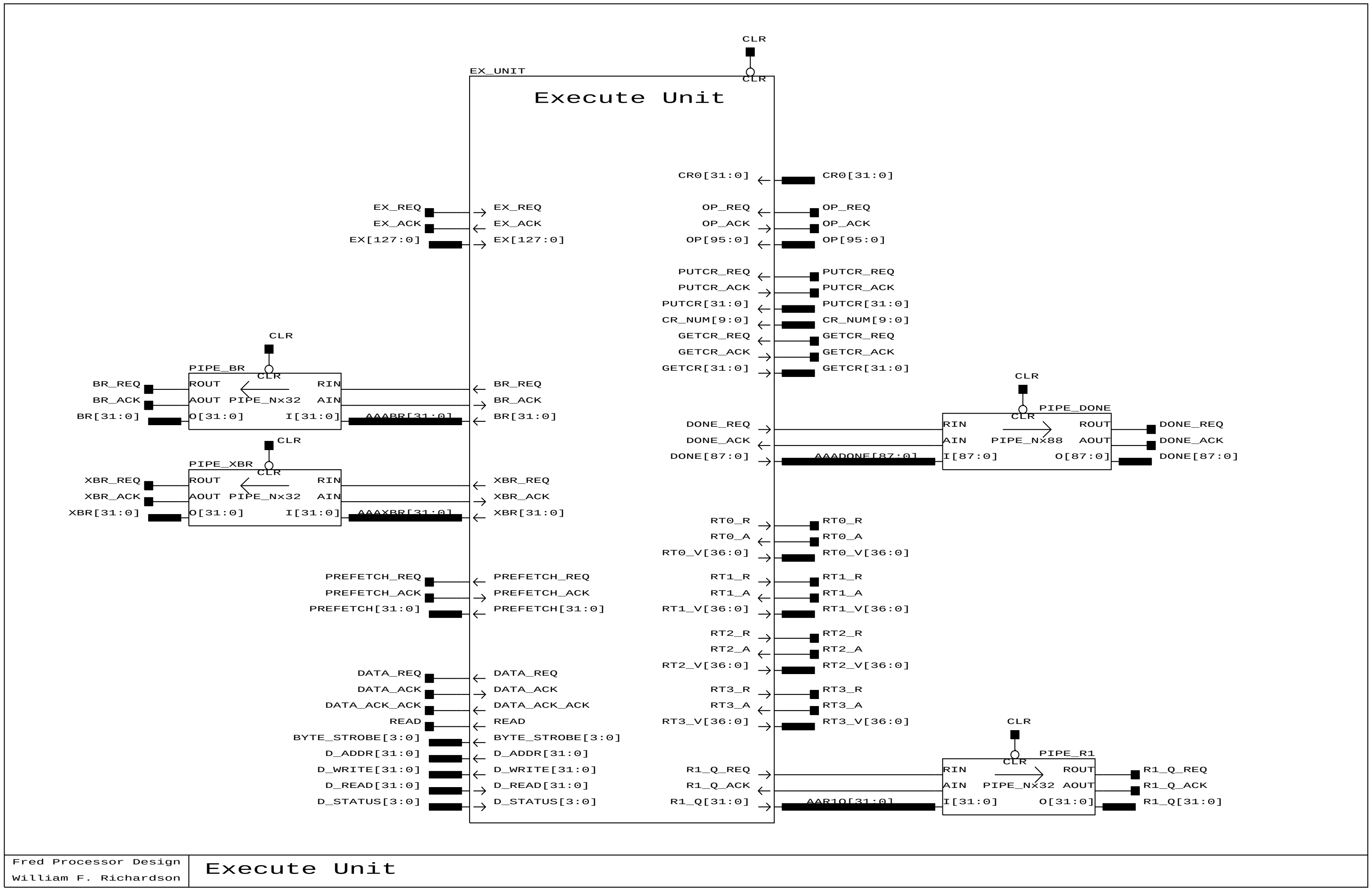


Figure 3.18: fred3

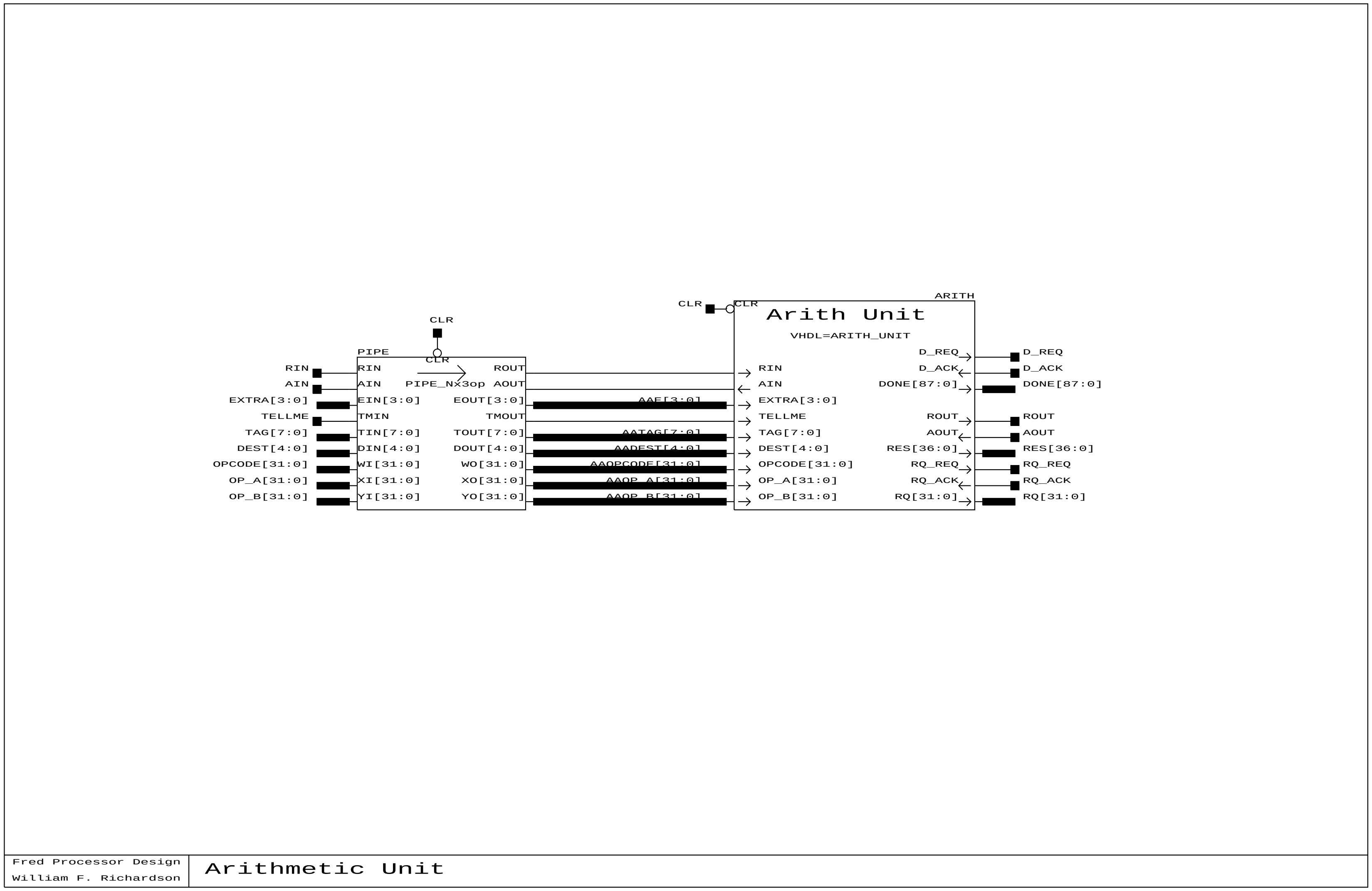


Figure 3.19: fu-arith

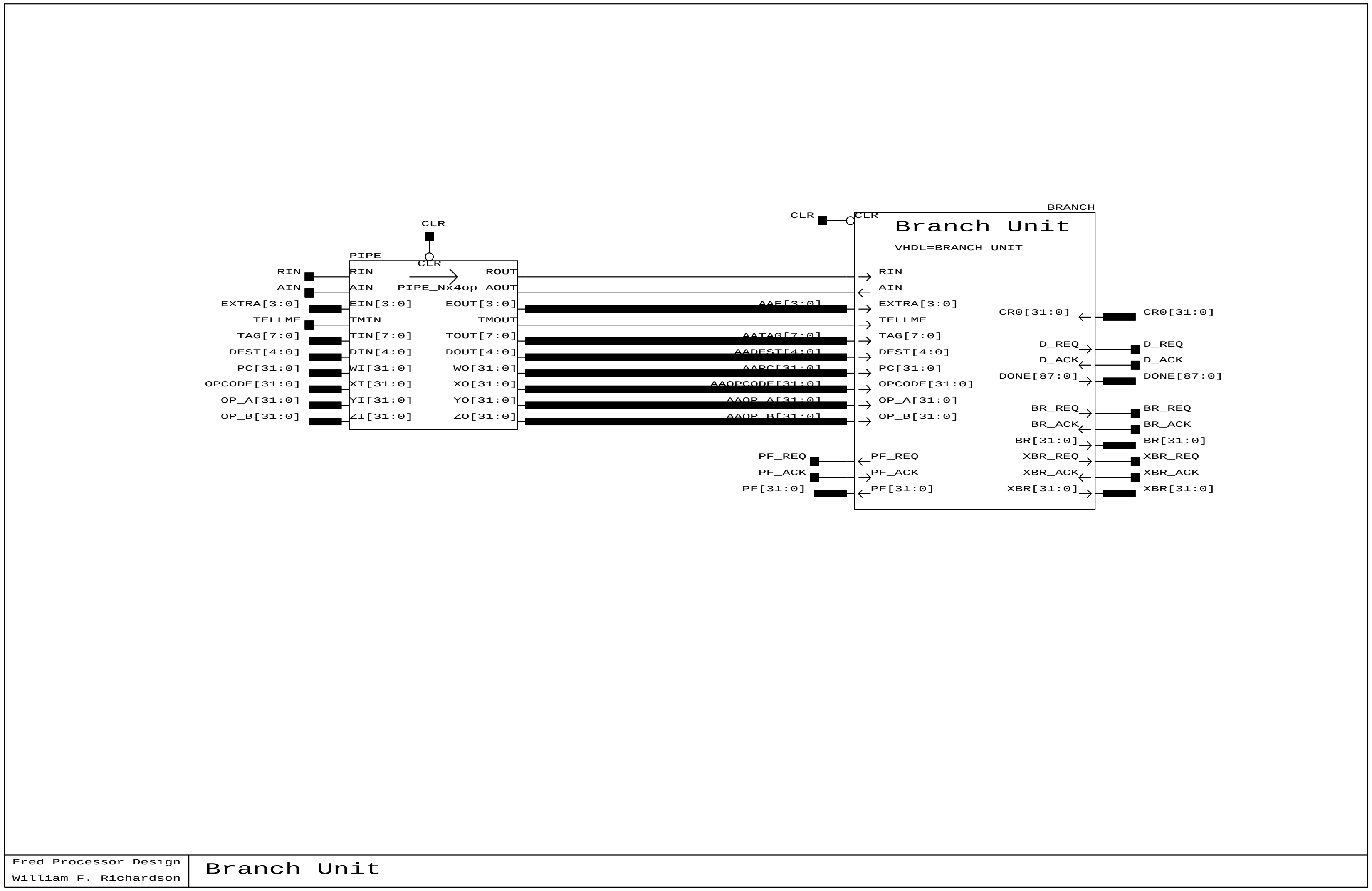


Figure 3.20: fu-branch

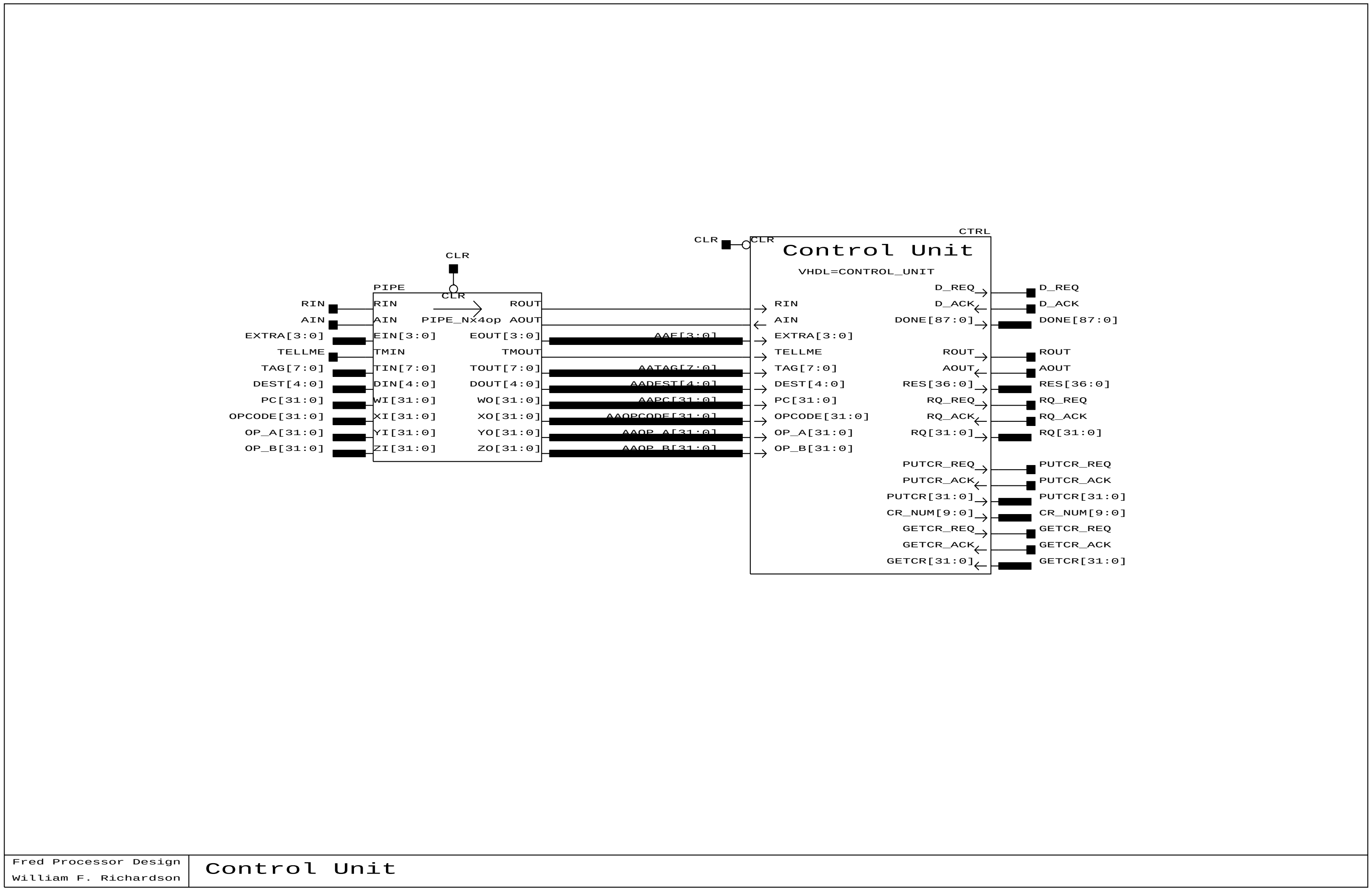


Figure 3.21: fu-control

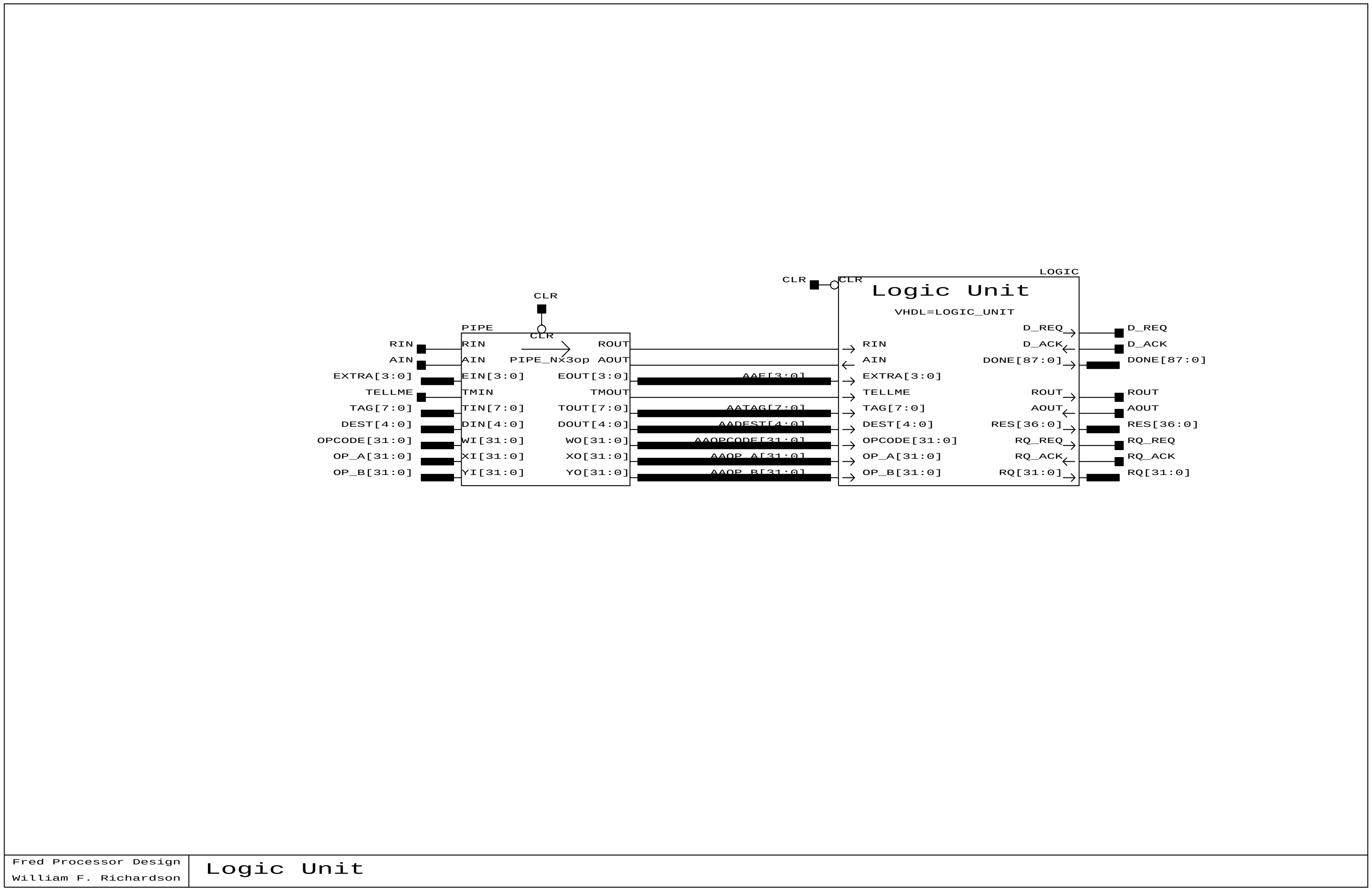


Figure 3.22: fu-logic

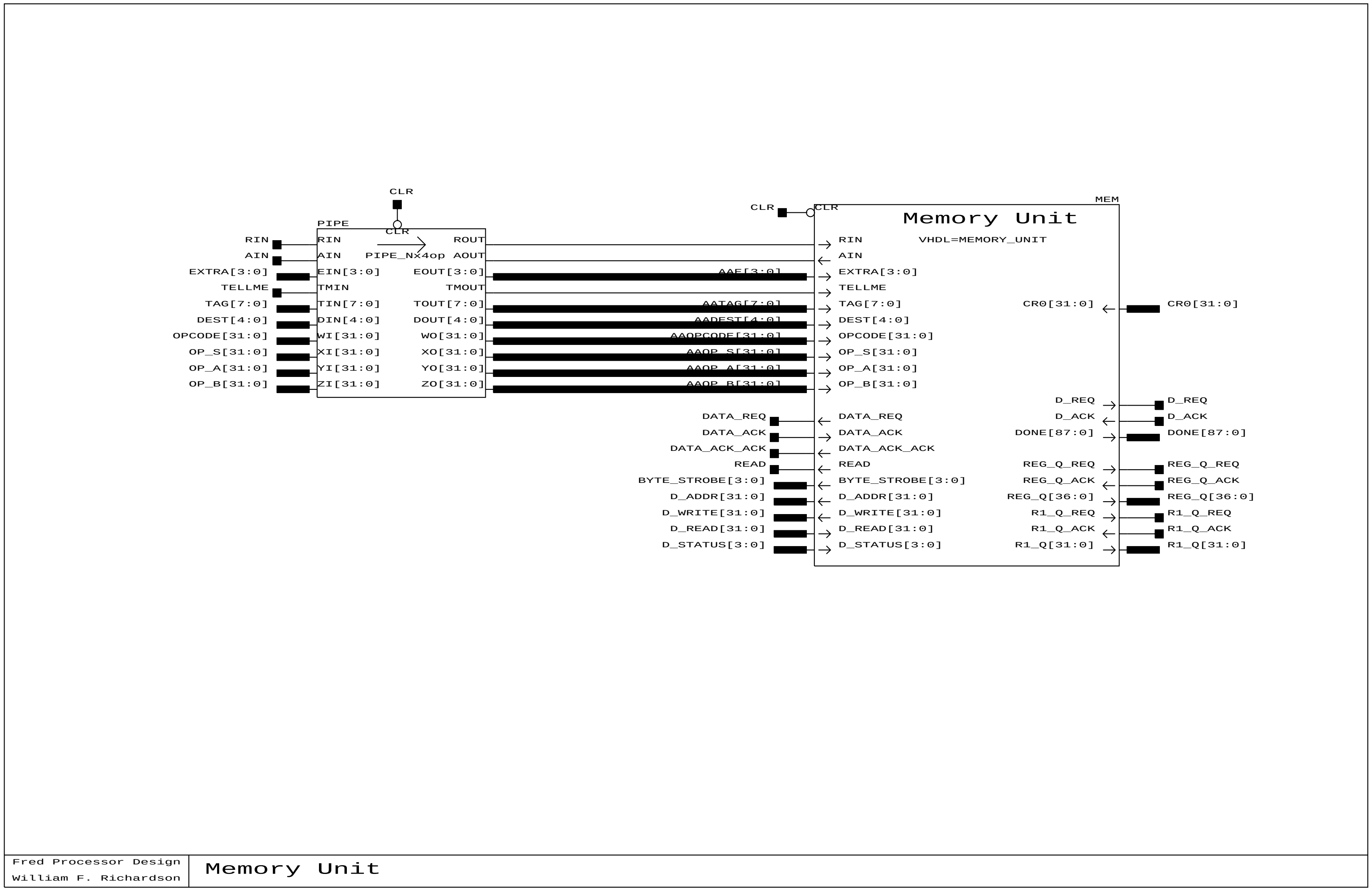
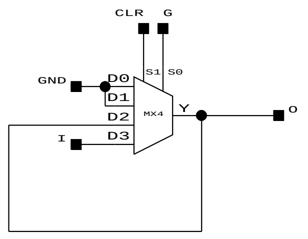


Figure 3.23: fu-memory

Figure 3.24: gate



When G is high, the input passes directly to the output.

When G is low, the input is latched.

Fred Processor Design
William F. Richardson

A gated latch.

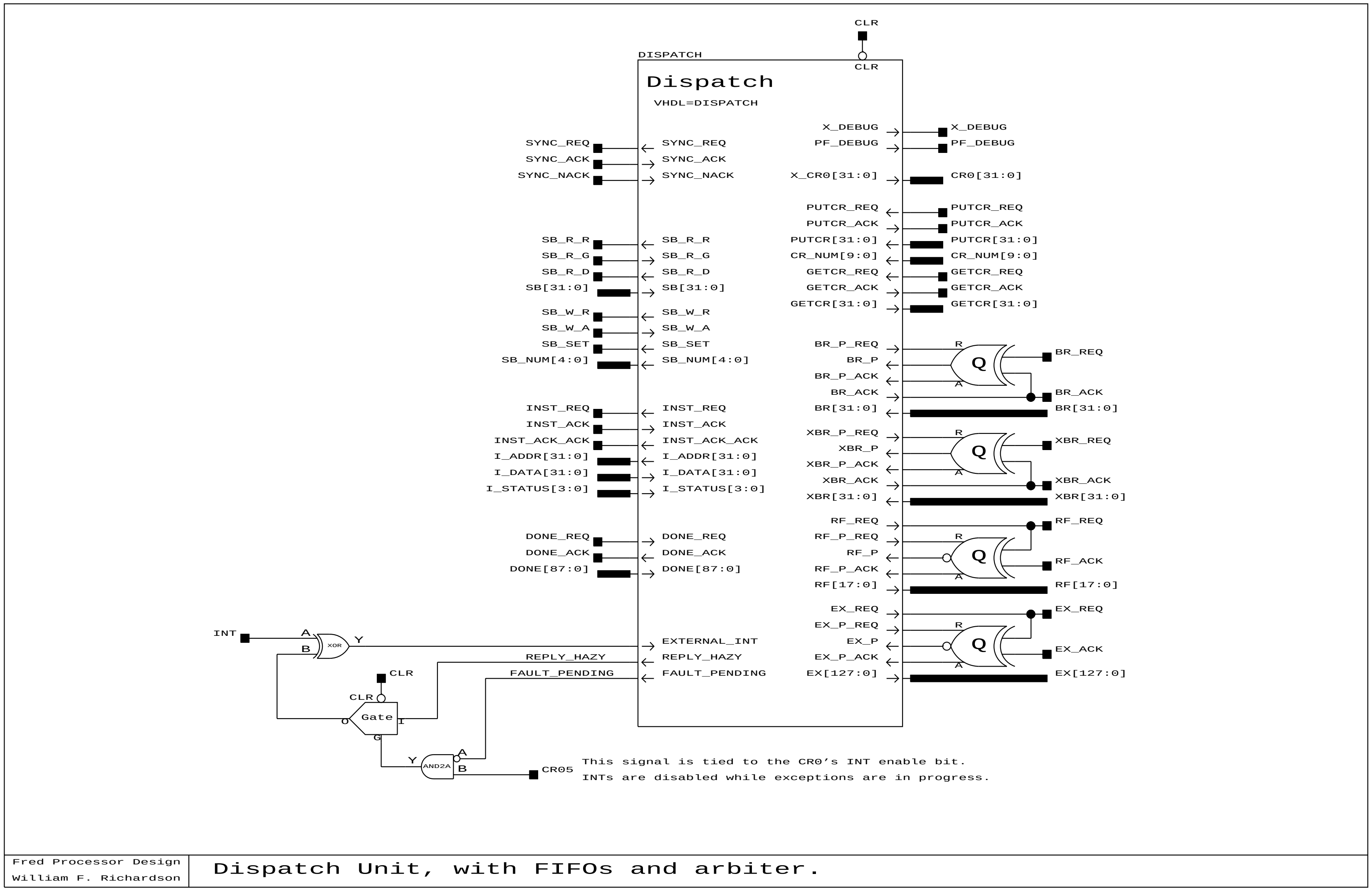
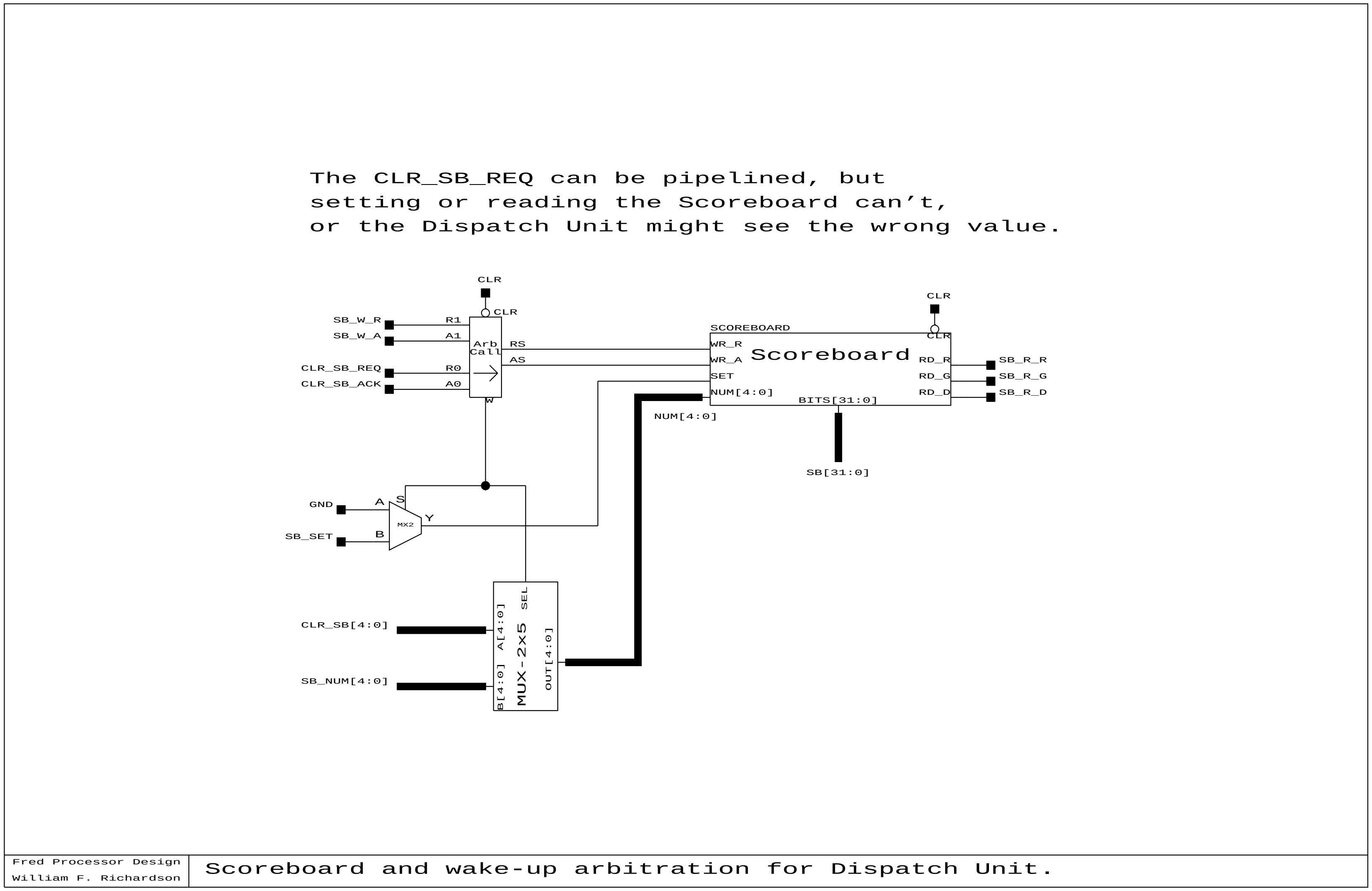


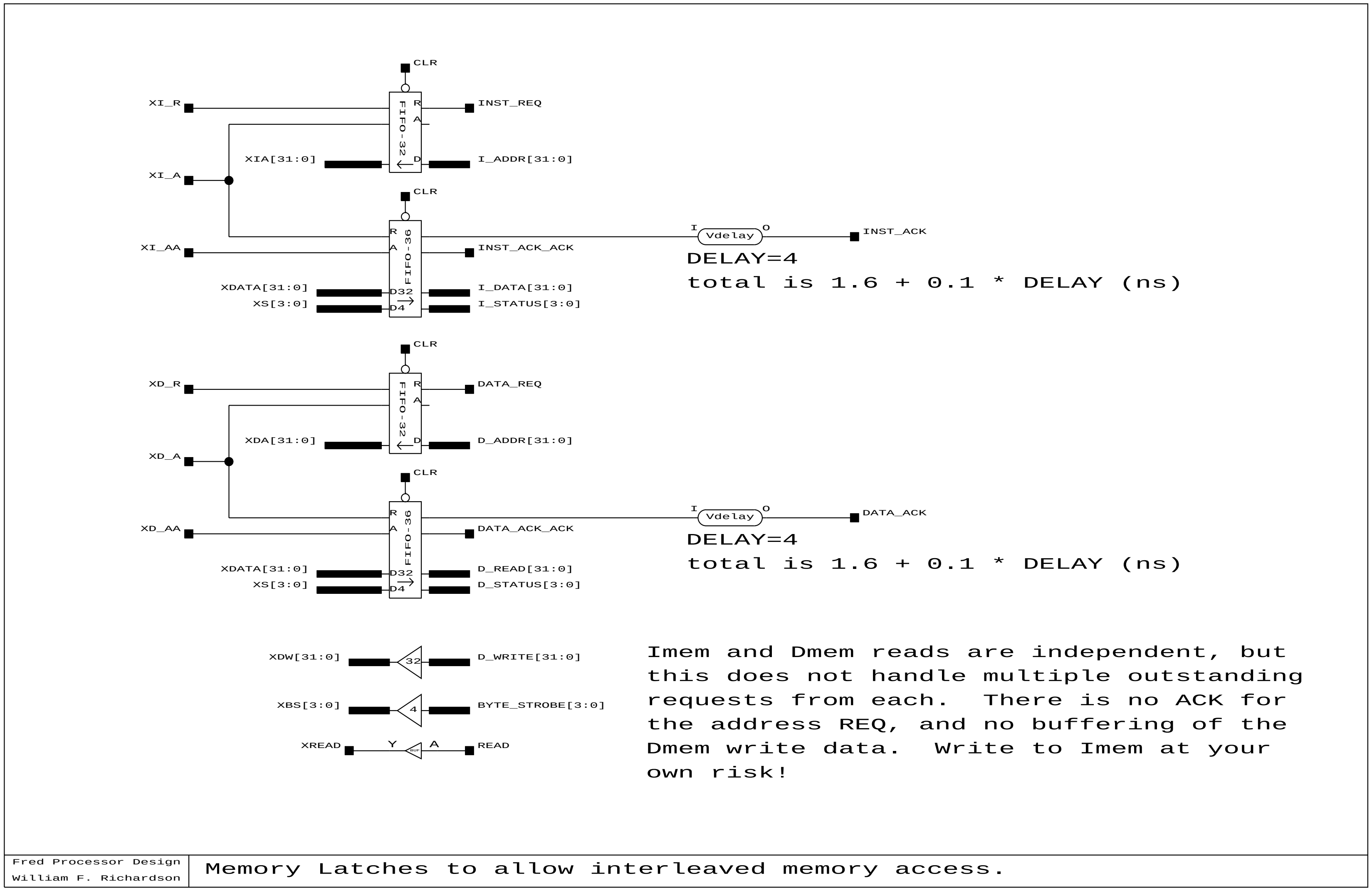
Figure 3.25: if-unit1



The CLR_SB_REQ can be pipelined, but
setting or reading the Scoreboard can't,
or the Dispatch Unit might see the wrong value.

Scoreboard and wake-up arbitration for Dispatch Unit.

Figure 3.26: if-unit2



Imem and Dmem reads are independent, but this does not handle multiple outstanding requests from each. There is no ACK for the address REQ, and no buffering of the Dmem write data. Write to Imem at your own risk!

Memory Latches to allow interleaved memory access.

Figure 3.27: memory-latches

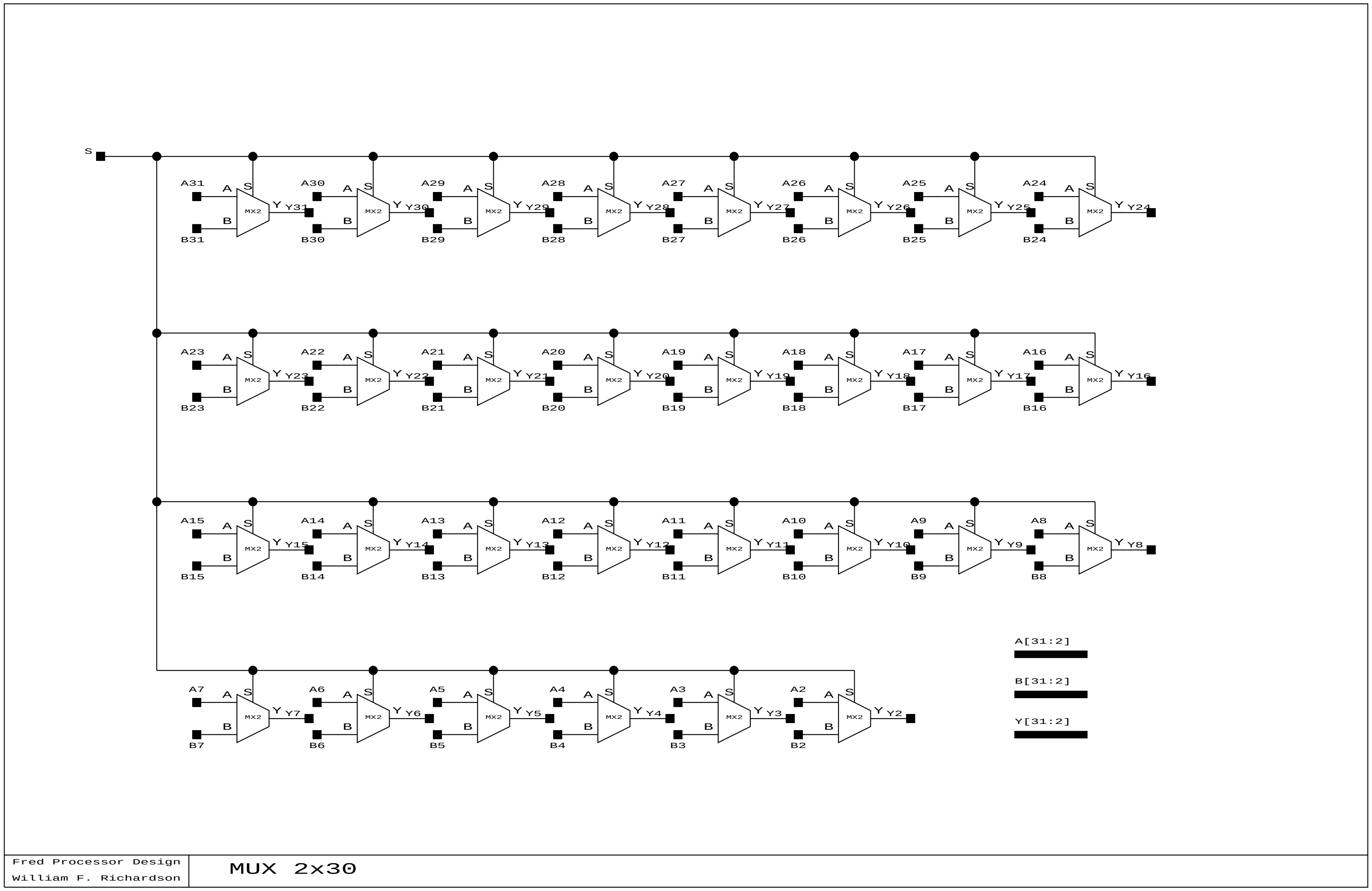


Figure 3.28: mux2x30

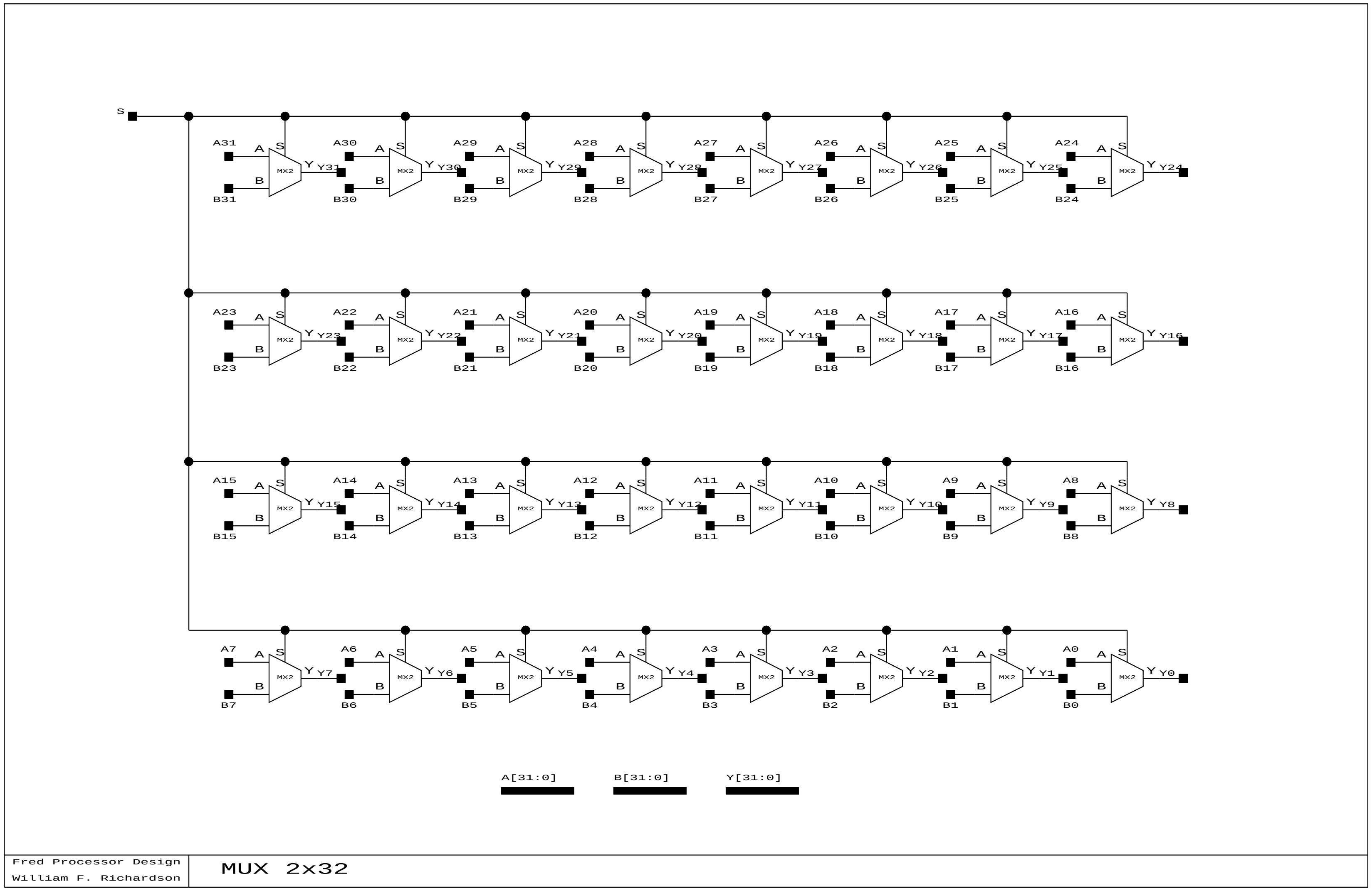

S
A31 A30 A29 A28 A27 A26 A25 A24
B31 B30 B29 B28 B27 B26 B25 B24
Y31 Y30 Y29 Y28 Y27 Y26 Y25 Y24
A23 A22 A21 A20 A19 A18 A17 A16
B23 B22 B21 B20 B19 B18 B17 B16
Y23 Y22 Y21 Y20 Y19 Y18 Y17 Y16
A15 A14 A13 A12 A11 A10 A9 A8
B15 B14 B13 B12 B11 B10 B9 B8
Y15 Y14 Y13 Y12 Y11 Y10 Y9 Y8
A7 A6 A5 A4 A3 A2 A1 A0
B7 B6 B5 B4 B3 B2 B1 B0
Y7 Y6 Y5 Y4 Y3 Y2 Y1 Y0
A
B
S
Y
MX2
A[31:0]
B[31:0]
Y[31:0]
Fred Processor Design
William F. Richardson
MUX 2x32

Figure 3.29: mux2x32

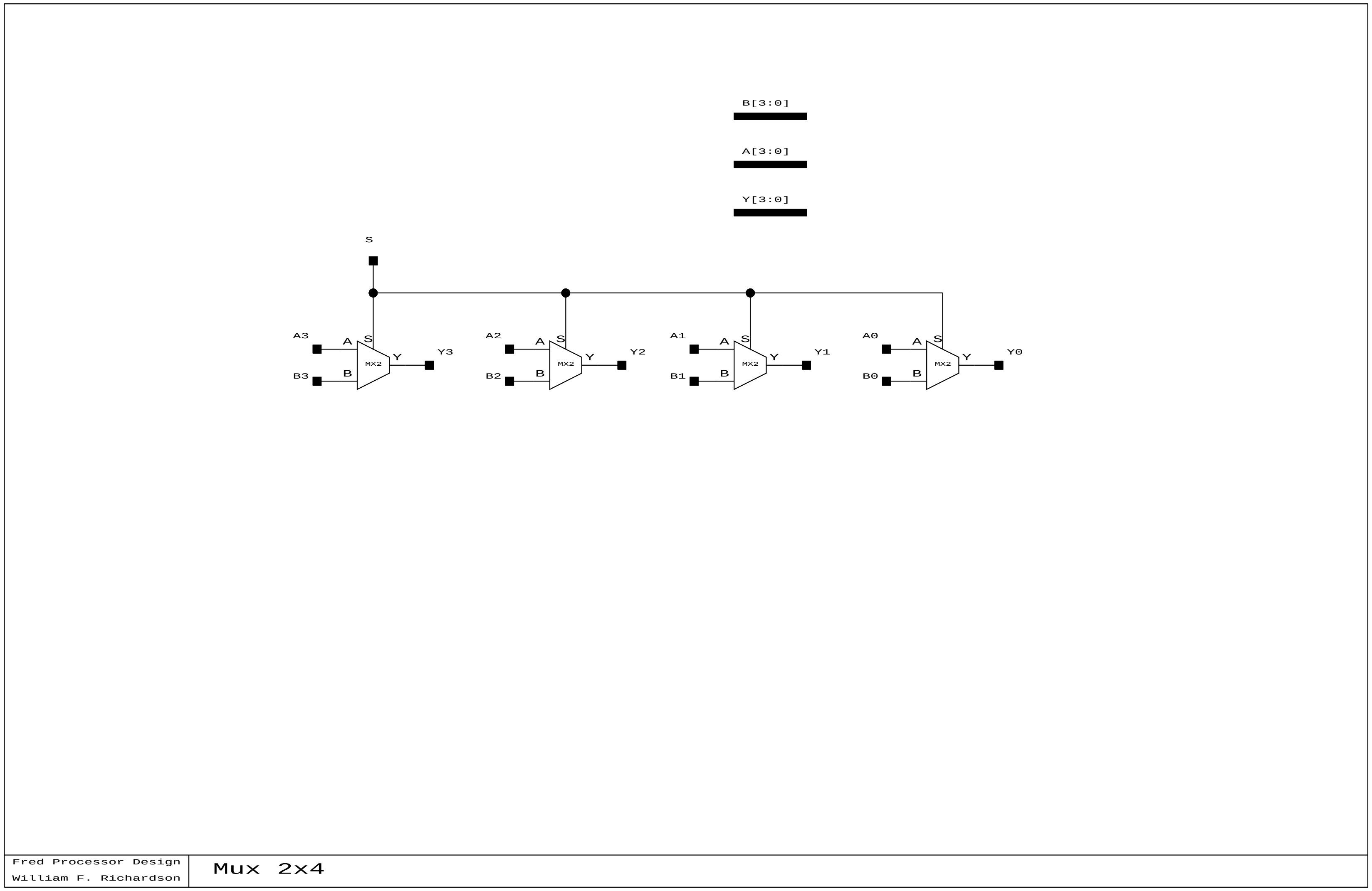


Figure 3.30: mux2x4

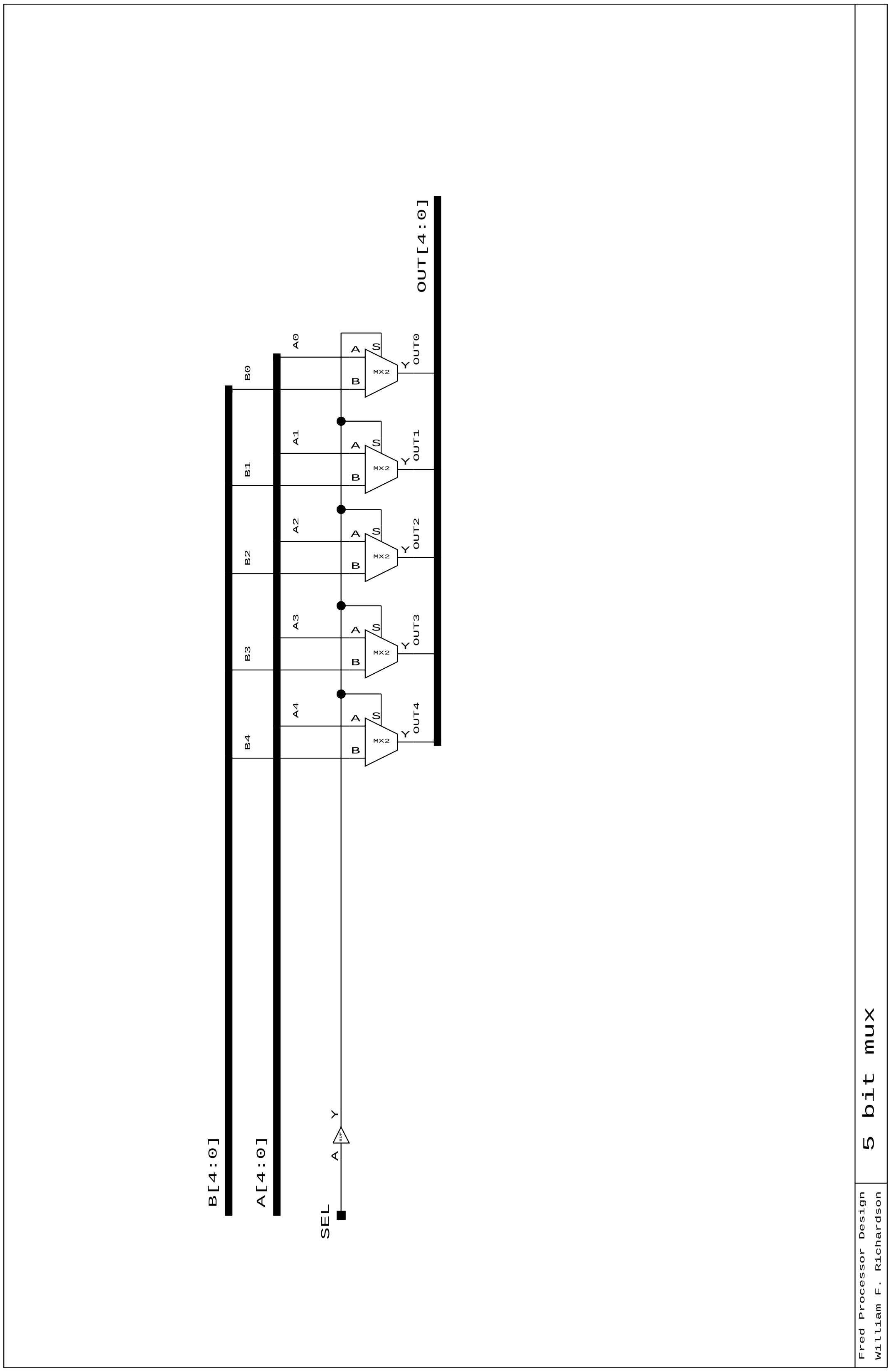


Figure 3.31: mux2x5

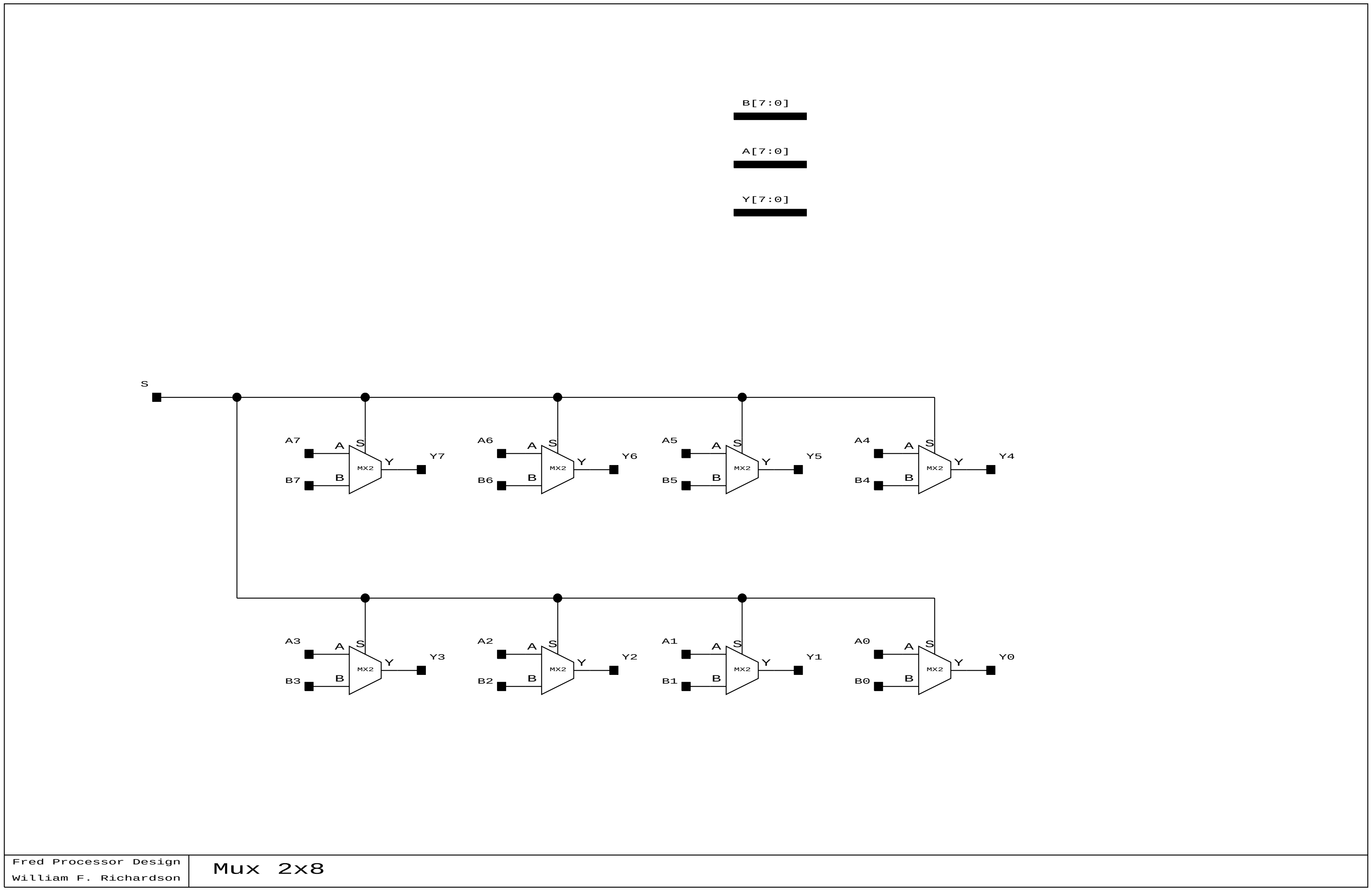


Figure 3.32: mux2x8

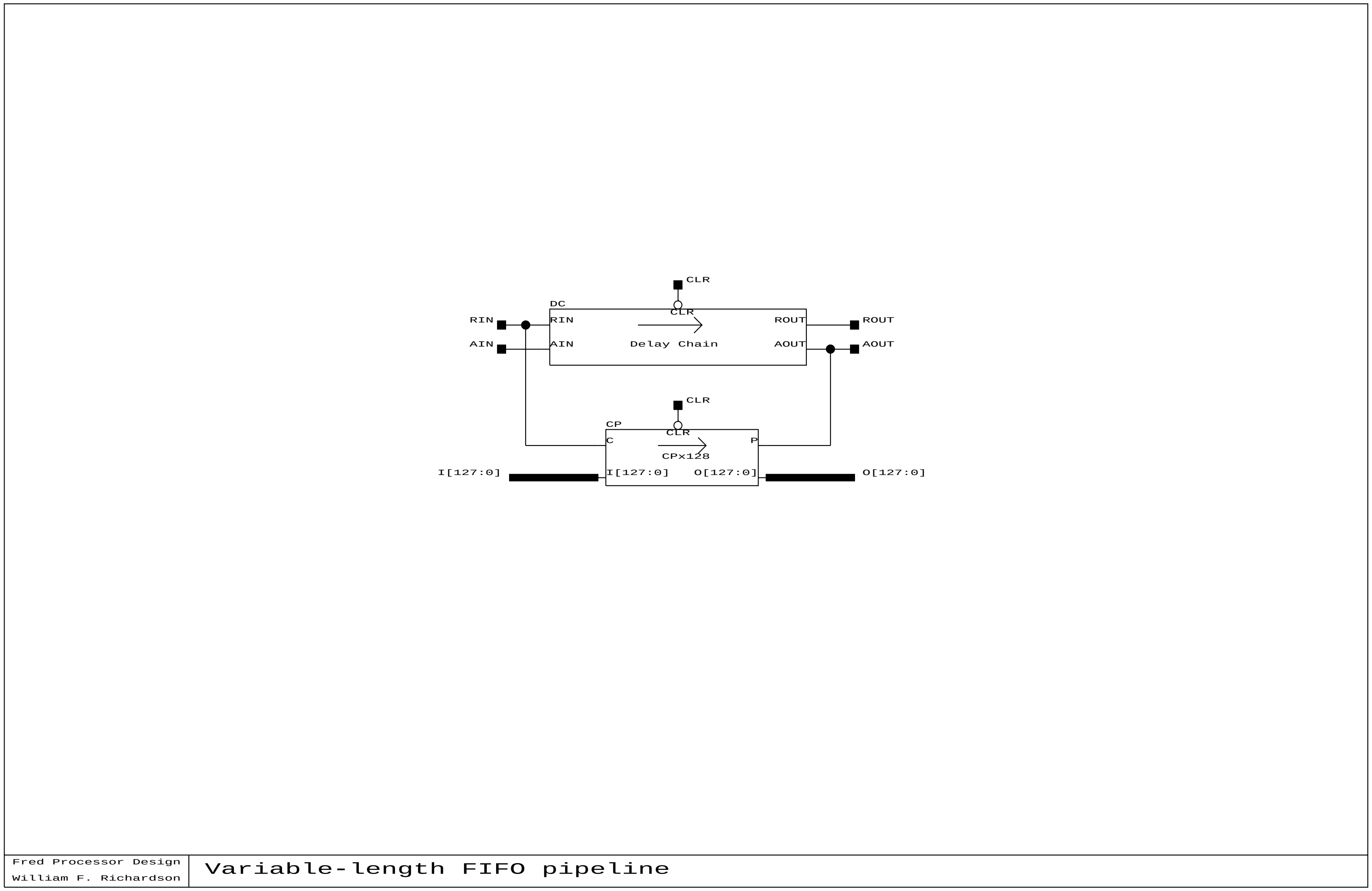
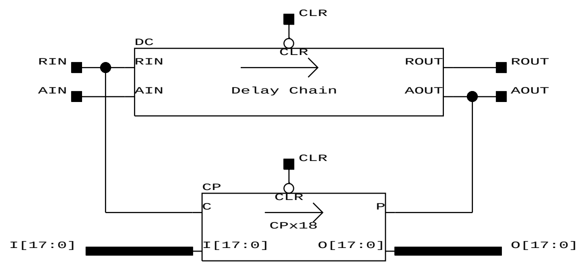


Figure 3.33: pipe-nx128



Fred Processor Design
William F. Richardson

Variable-length FIFO pipeline

Figure 3.34: pipe-nx18

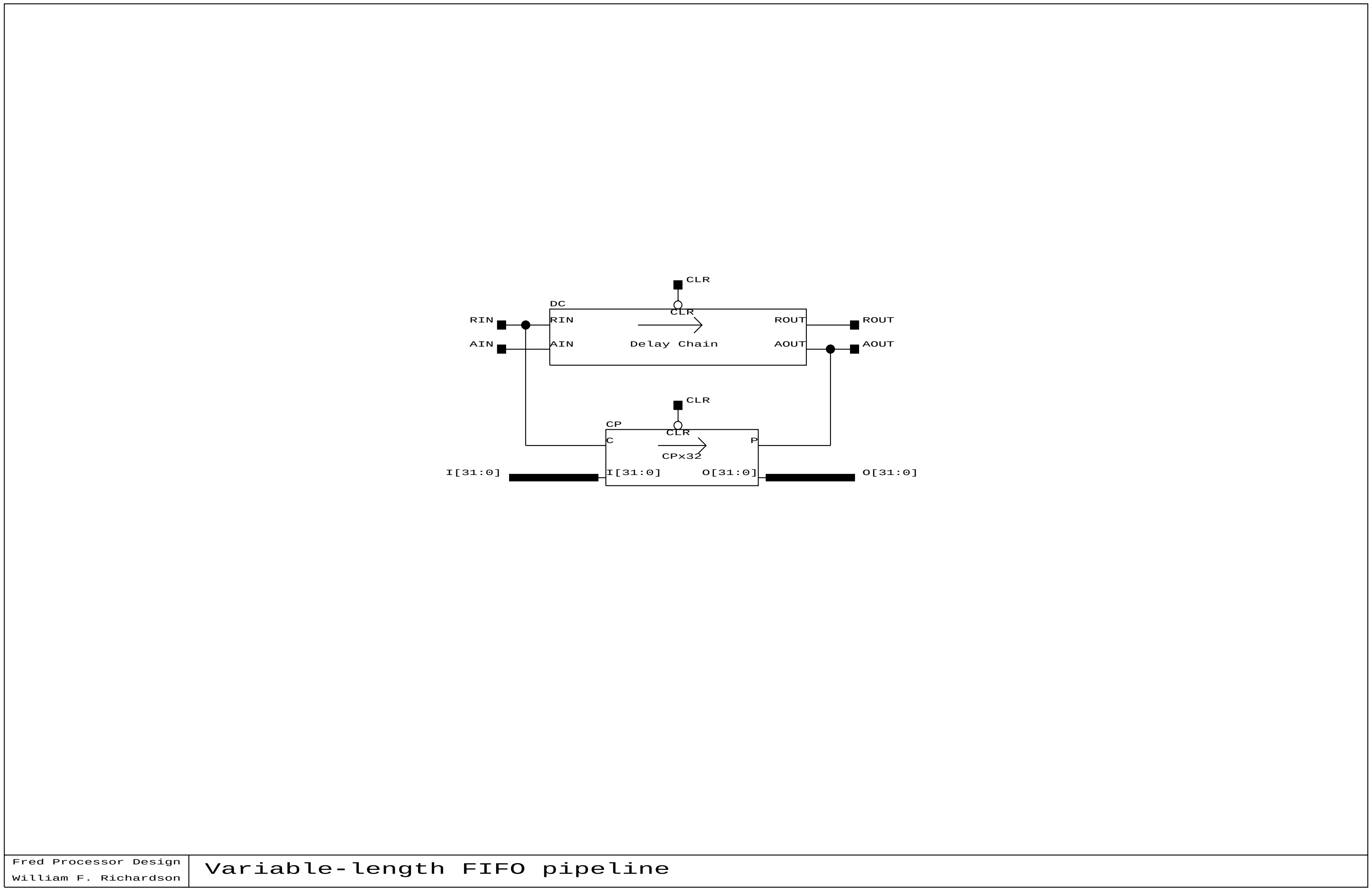


Figure 3.35: pipe-nx32

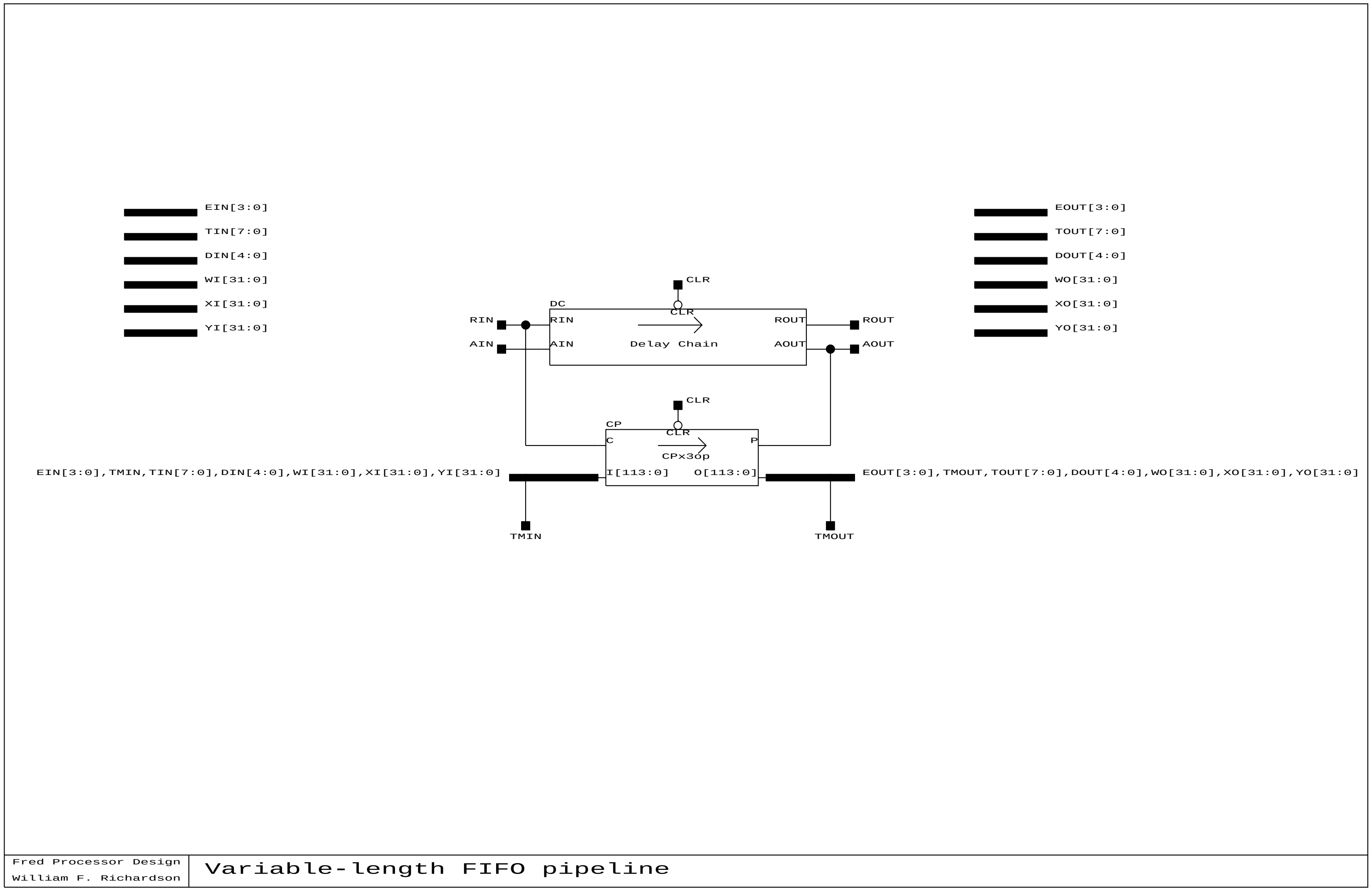


Figure 3.36: pipe-nx3op

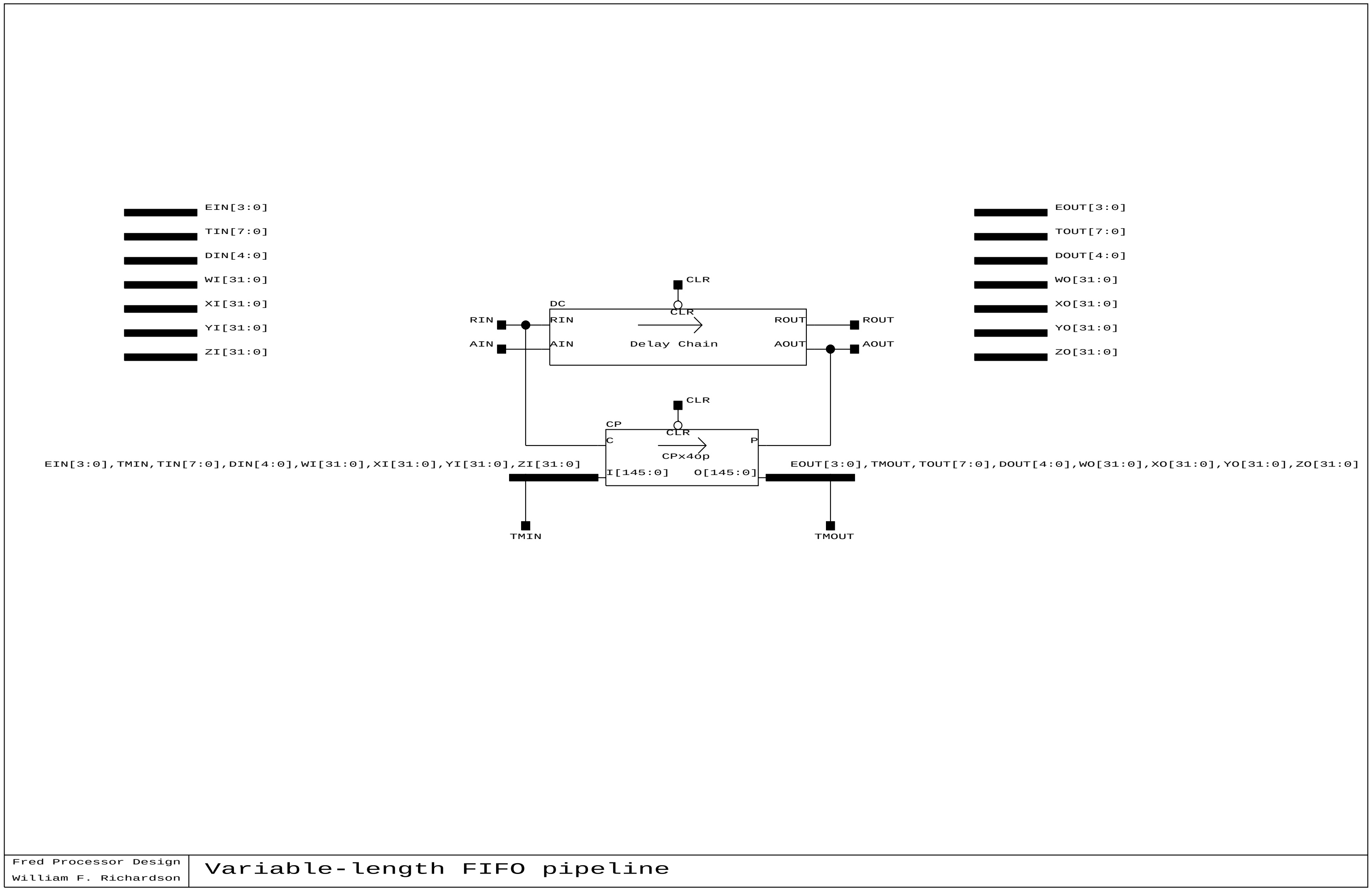


Figure 3.37: pipe-nx4op

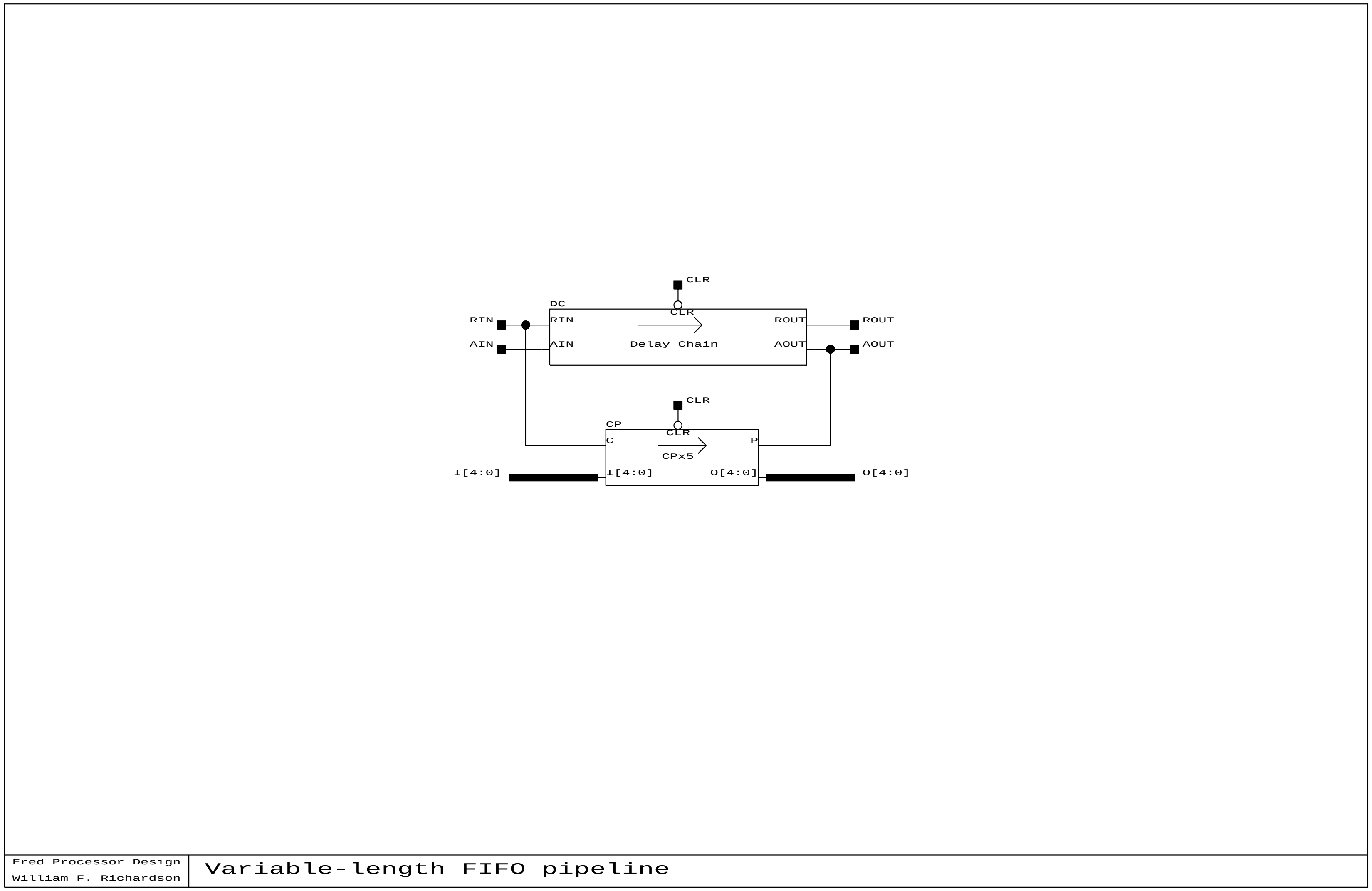


Figure 3.38: pipe-nx5

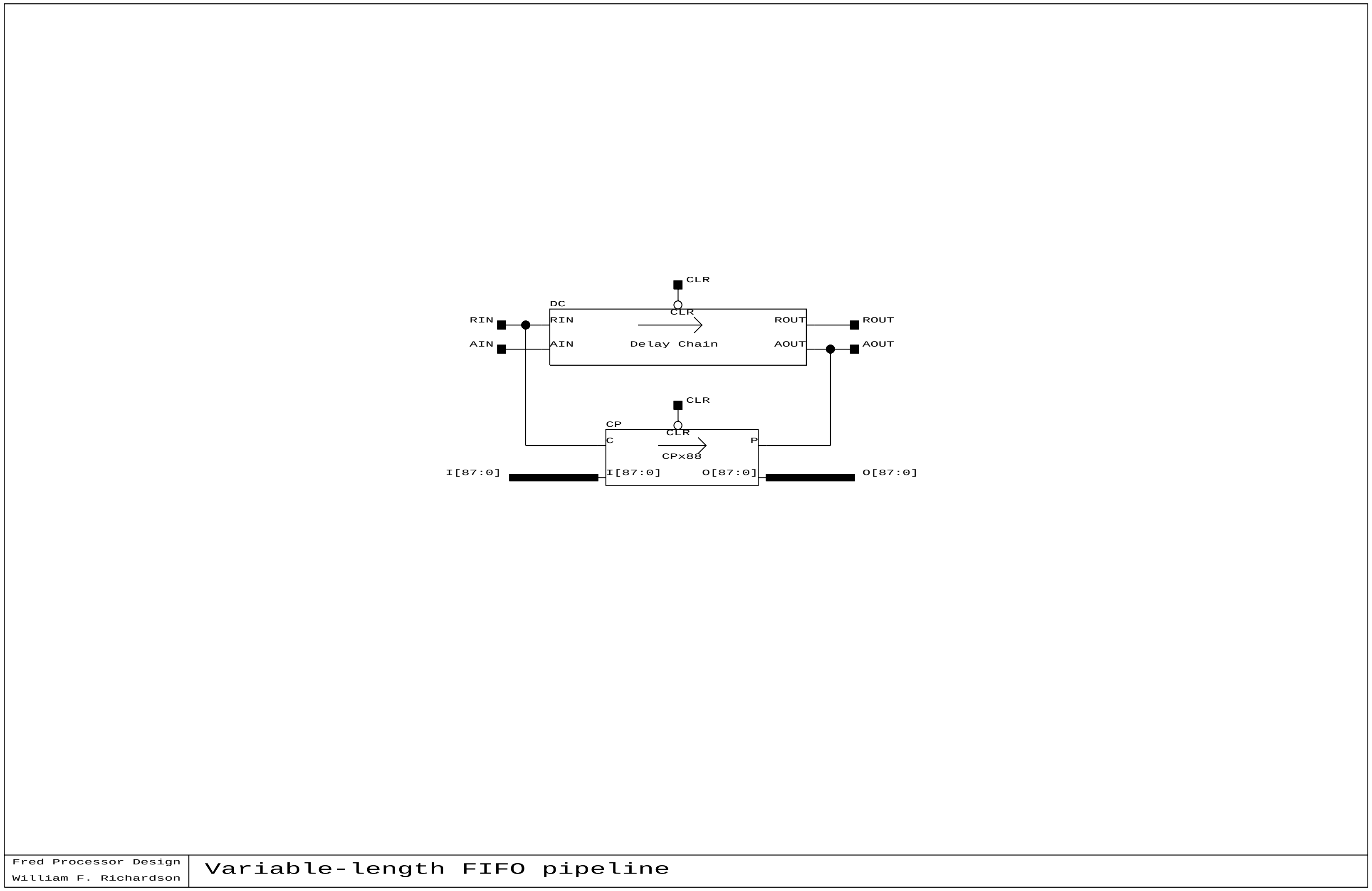


Figure 3.39: pipe-nx88

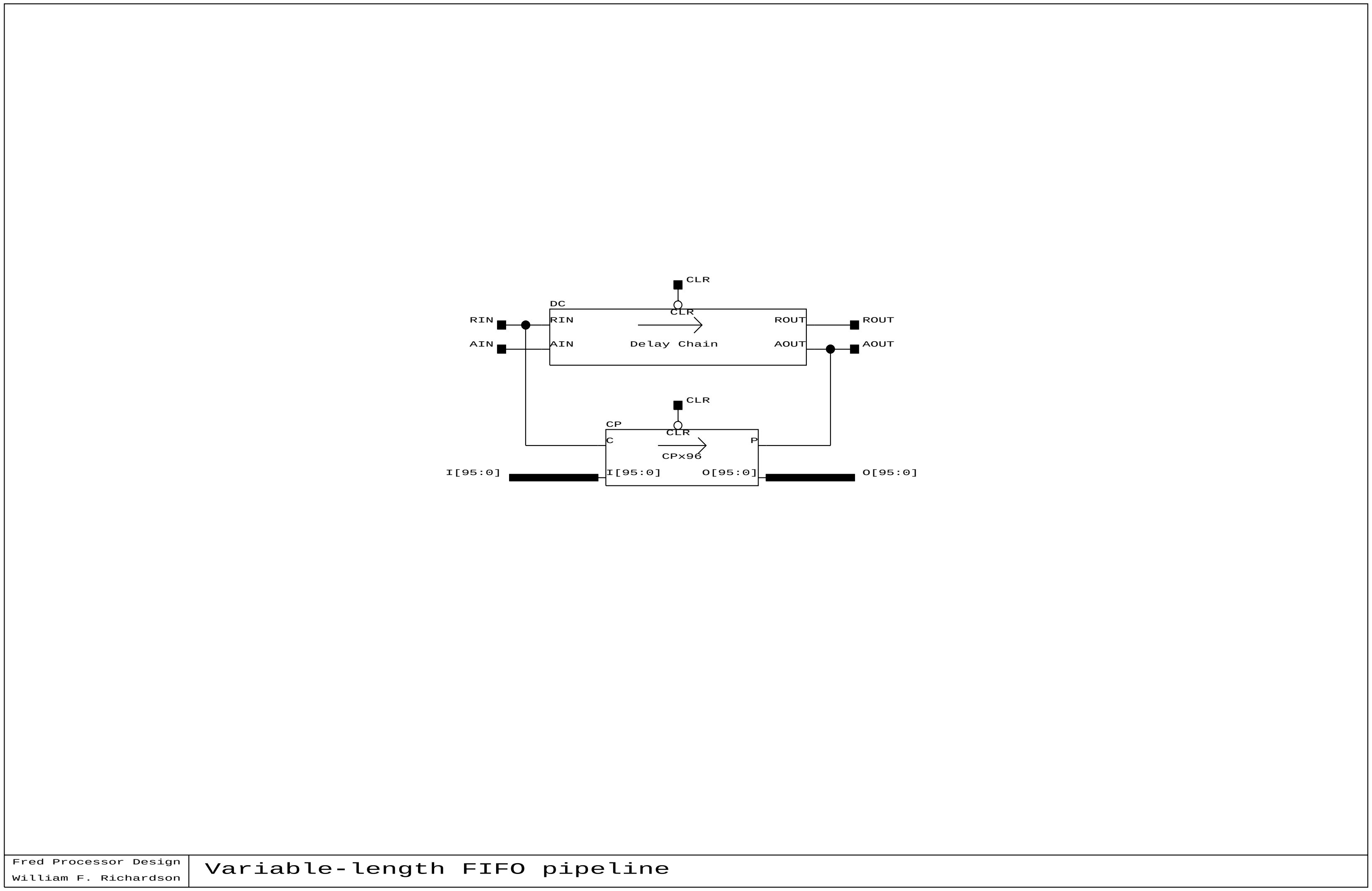


Figure 3.40: pipe-nx96

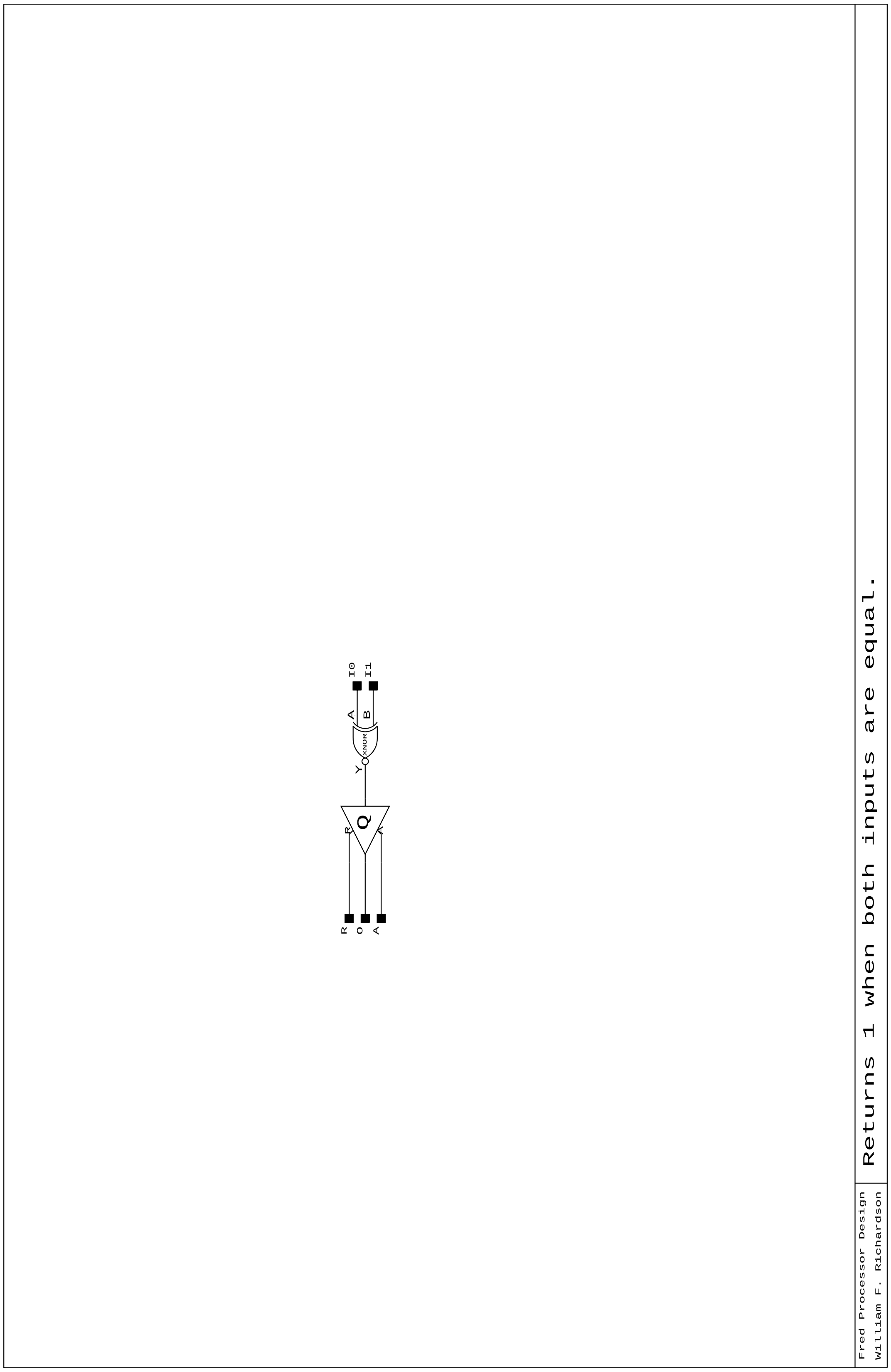
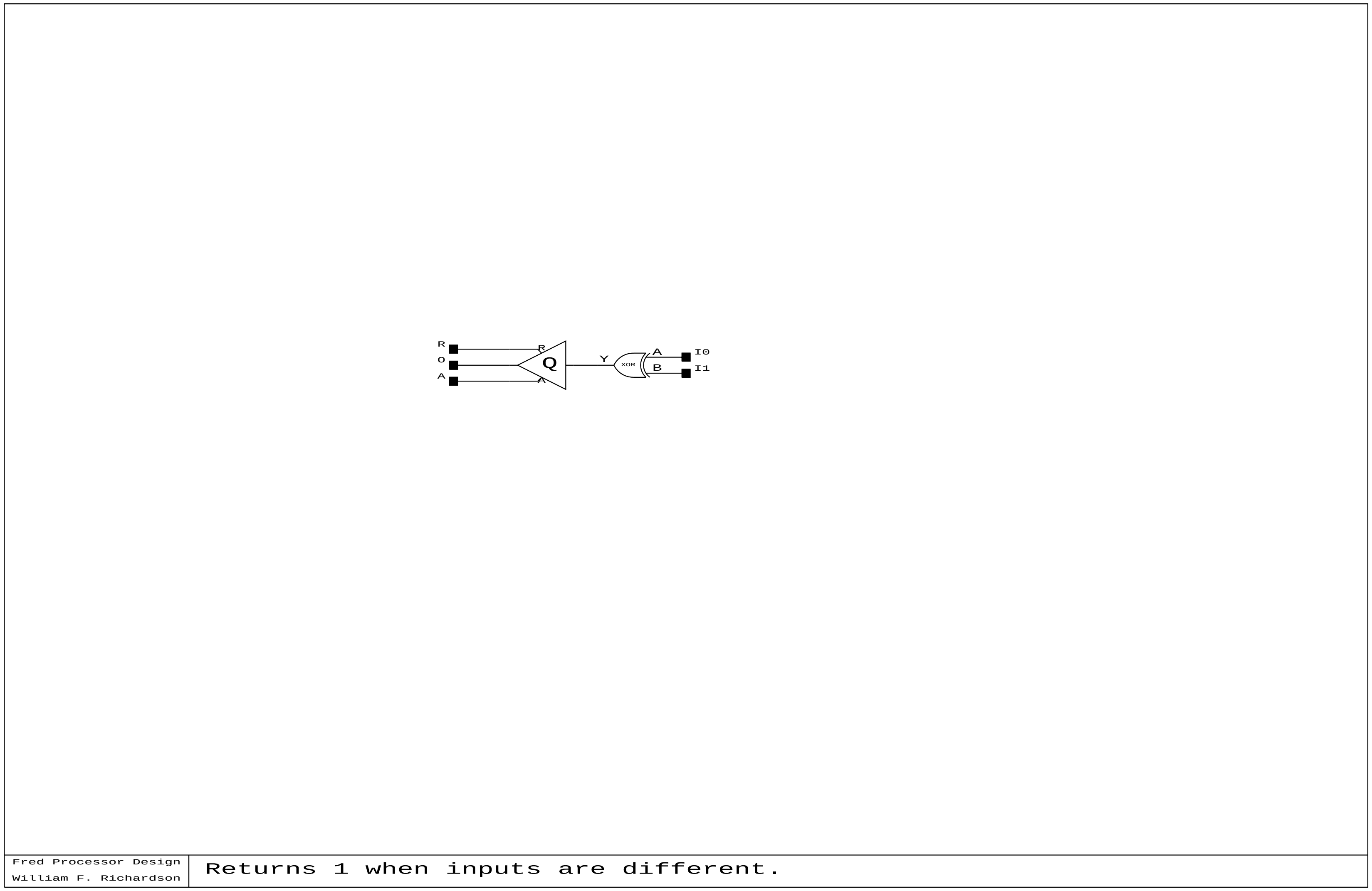


Figure 3.41: qflop-even



Fred Processor Design
William F. Richardson

Returns 1 when inputs are different.

Figure 3.42: qflop-odd

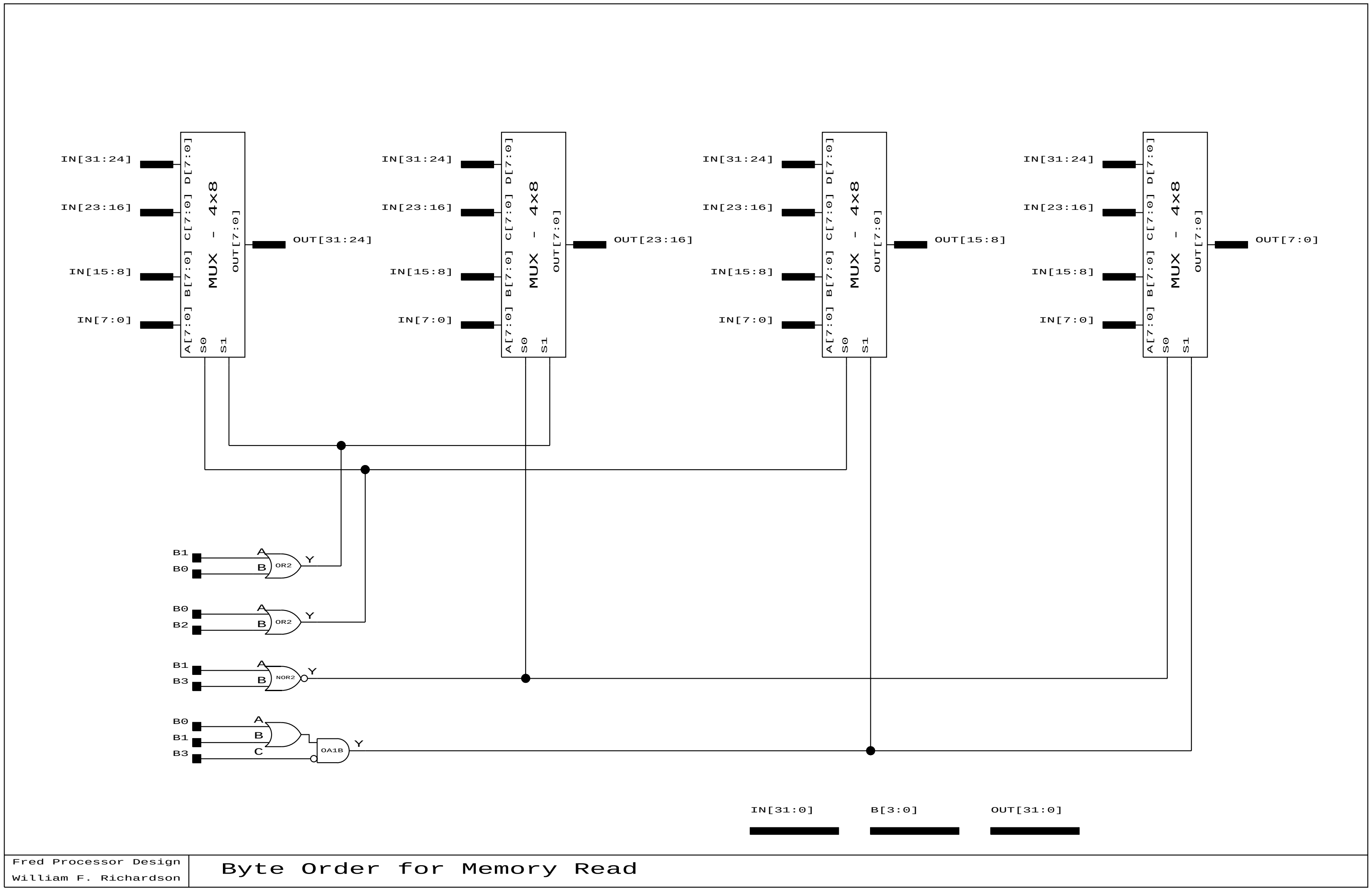


Figure 3.43: ram-byte-order

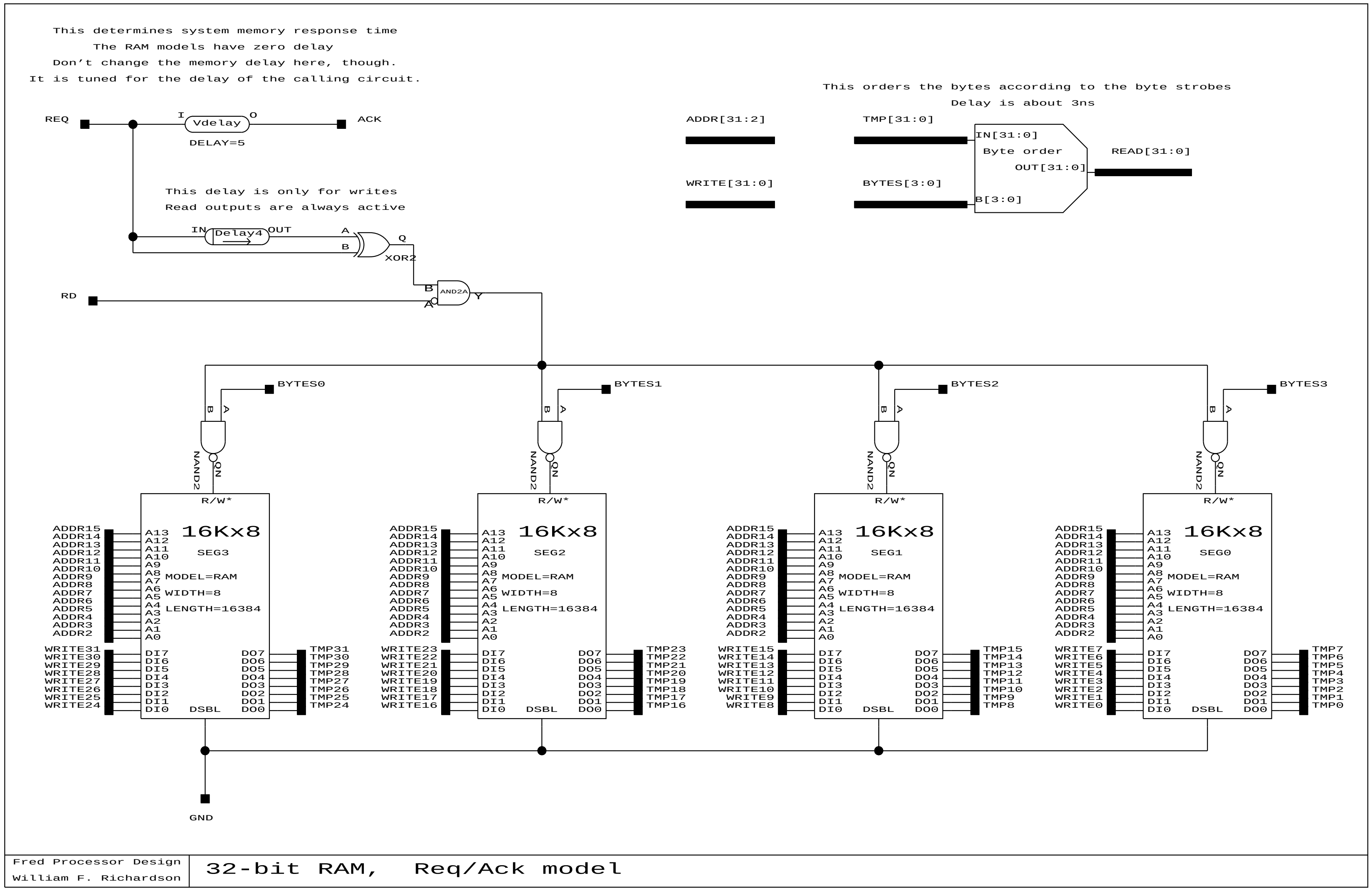

This determines system memory response time
The RAM models have zero delay
Don't change the memory delay here, though.
It is tuned for the delay of the calling circuit.
REQ
Vdelay
DELAY=5
ACK
This delay is only for writes
Read outputs are always active
Delay4
XOR2
AND2A
RD
This orders the bytes according to the byte strobes
Delay is about 3ns
ADDR[31:2]
WRITE[31:0]
TMP[31:0]
BYTES[3:0]
IN[31:0]
Byte order
OUT[31:0]
B[3:0]
READ[31:0]
BYTES0
BYTES1
BYTES2
BYTES3
NAND2
R/W*
16Kx8
SEG3
SEG2
SEG1
SEG0
MODEL=RAM
WIDTH=8
LENGTH=16384
DSBL
GND
Fred Processor Design
William F. Richardson
32-bit RAM, Req/Ack model

Figure 3.44: ram16kx32

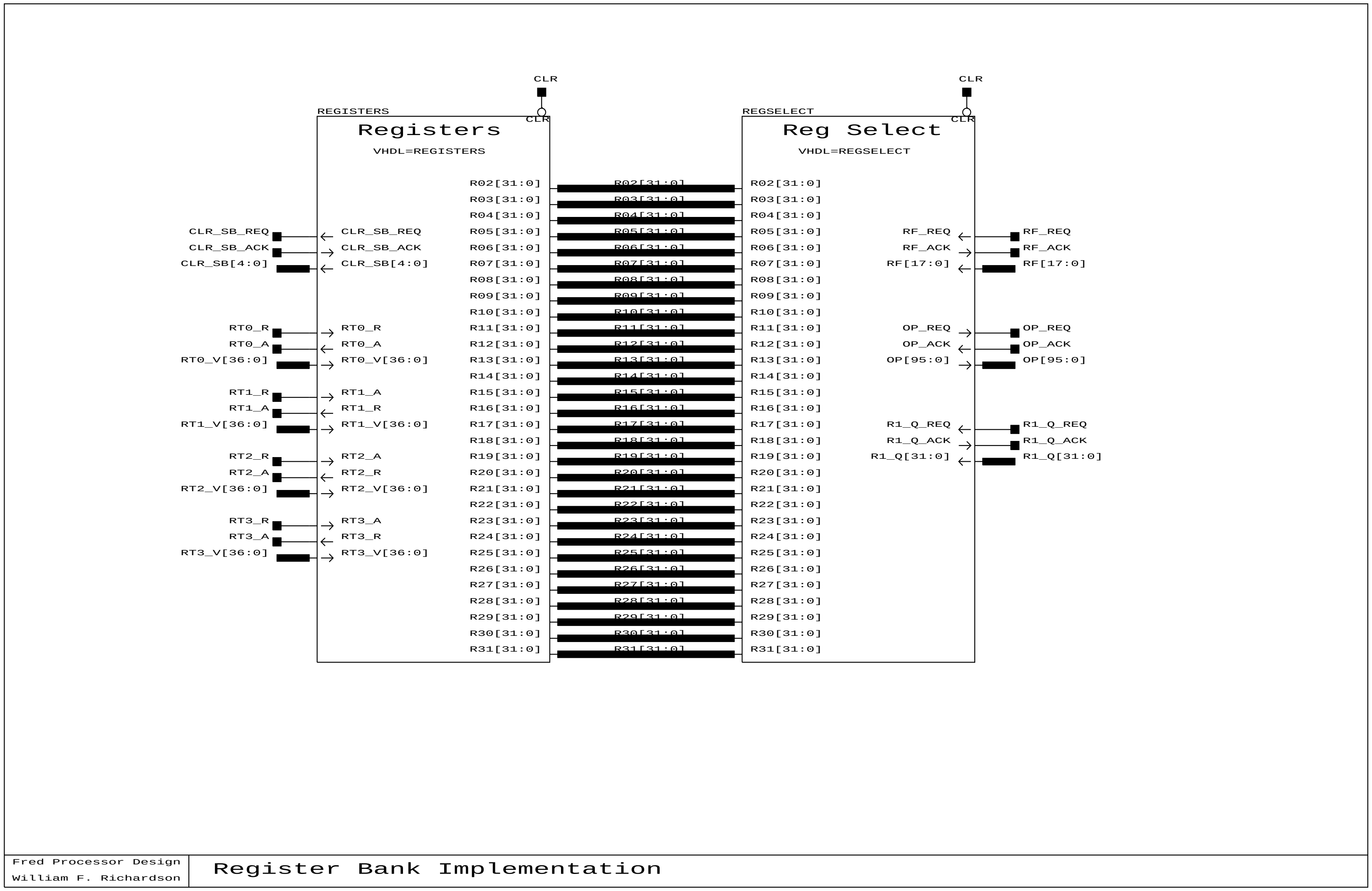
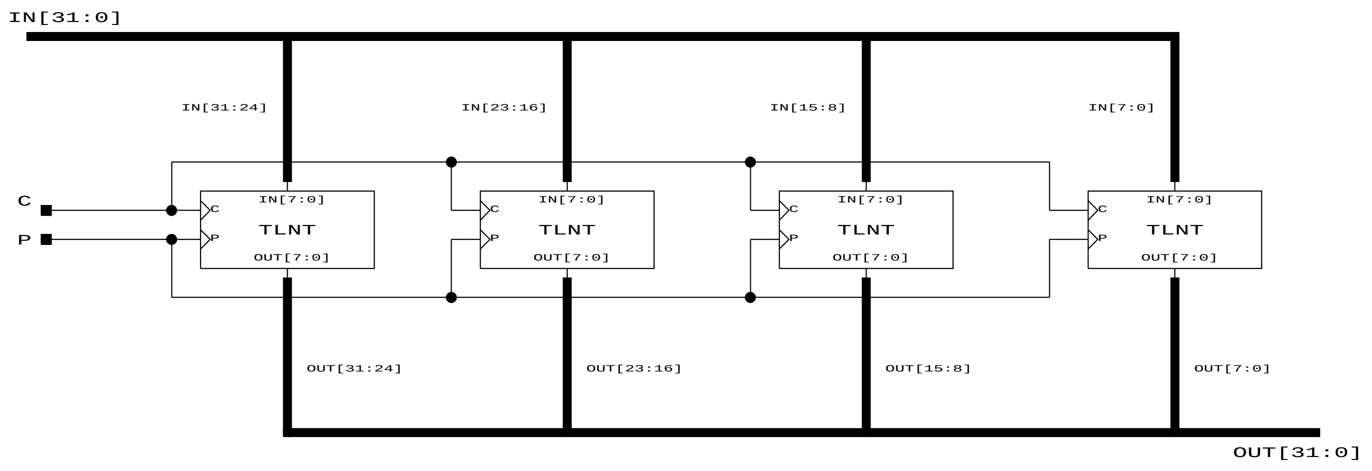


Figure 3.45: rf-unit



C fanin is 8
P fanin is 4
In fanin is 2
Out drives 2

Total Modules: 44

Fred Processor Design William F. Richardson	32-bit TLNT

Figure 3.46: tlnt-32

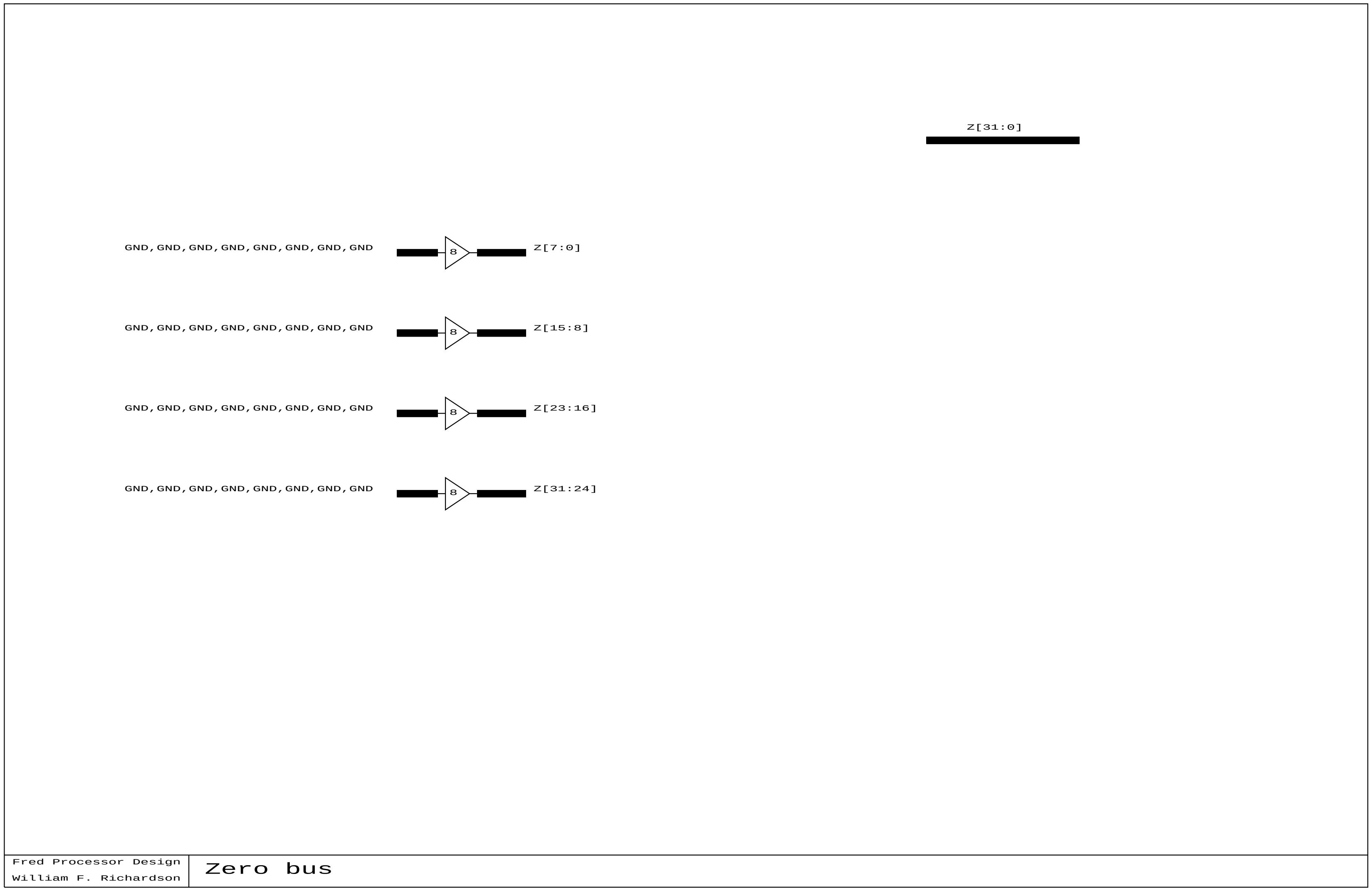

Z[31:0]
GND,GND,GND,GND,GND,GND,GND,GND
8
Z[7:0]
GND,GND,GND,GND,GND,GND,GND,GND
8
Z[15:8]
GND,GND,GND,GND,GND,GND,GND,GND
8
Z[23:16]
GND,GND,GND,GND,GND,GND,GND,GND
8
Z[31:24]
Fred Processor Design
William F. Richardson
Zero bus

Figure 3.47: zerox32