Lecture 7

Special Structures

Some “Special Structures”

- Internal memory, FIFOs
- State Machines
- Tri-State buffers, high-speed multiplexers
- Pullups, pulldowns, keepers, DDR on I/O
- Multipliers, DSP Blocks
- Delay-locked Loops and clock managers
- Test circuitry (i.e., built-in JTAG support)

Up until now, we have generally covered creation of logic that is pretty much universal to all FPGAs. “the basics” – D flipflops and combinational logic.

The above things are handled differently from FPGA to FPGA.

Xilinx Block RAMs

```
architecture behaviour of regfile is

type mem_type is array(0 to 31) of std_logic_vector(7 downto 0);
signal my_regfile : mem_type;
signal adr_reg : unsigned(4 downto 0);
begin
  regproc : process(clk)
  begin
     if rising_edge(clk) then
        if write_en = '1' then
           my_regfile(to_integer(unsigned(address))) <= data_in;
        end if;
     adr_reg <= unsigned(address);
     end if;
  end process regproc;

data_out <= my_regfile(to_integer(adr_reg));

end behaviour;
```

This infers 1 of our 12 block rams (18kbit) arranged as a 256x16 or 512x8 (And wastes 2048 – 256 = 1792 bytes)

RAM

- Many FPGAs have some built-in RAM.
  - Flops aren’t the most dense storage method
- Xilinx provides a couple of different means of doing this
  - Distributed RAM
    - Small RAMs distributed throughout the FPGA which can be combined together to form various size RAMs
  - Block RAM
    - Larger blocks of RAM which are located on the periphery of the device.

RAM Example

```
entity gpregfile is
port (clk, we       : in std_logic;
                    adr                   : in std_logic_vector(7 downto 0);
                    din       : in std_logic_vector(7 downto 0);
                    dout      : out std_logic_vector(7 downto 0));
end gpregfile;

architecture Behavioral of gpregfile is
constant Size: integer := 256;                    -- Number of registers

-- Define type for register
type memory_type is array (integer range <>) of std_logic_vector(7 downto 0);
signal memory: memory_type((Size - 1) downto 0); -- Allocate RAM
begin
   process(clk)
      begin
          if rising_edge(clk) then      -- Only update on rising edge
             if we = '1‘ then           -- If a write, store new data
                memory(conv_integer(unsigned(adr))) <= din; -- Stores value
             end if;
          end if;
   end process;

dout <= memory(conv_integer(unsigned(adr)));

end Behavioral;
```

Possible Implementation #1

- Design uses 256x8 = 2048 Flipflops
  - Out of 9000 in our device (21% used)

```
INFO:Xst:738 - HDL ADVISOR - 2048 flip-flops
were inferred for signal <memory>. You may be
trying to describe a RAM in a way that is
incompatible with block and distributed RAM
resources available on Xilinx devices, or with
a specific template that is not supported.
Please review the Xilinx resources
documentation and the XST user manual for
coding guidelines. Taking advantage of RAM
resources will lead to improved device usage
and reduced synthesis time.
```

Possible Implementation #2

- Design uses “distributed RAM”
  - LUT normally used for function generation is in fact already a 16x1 RAM. Some of these have extra signals hooked up to make use of them directly

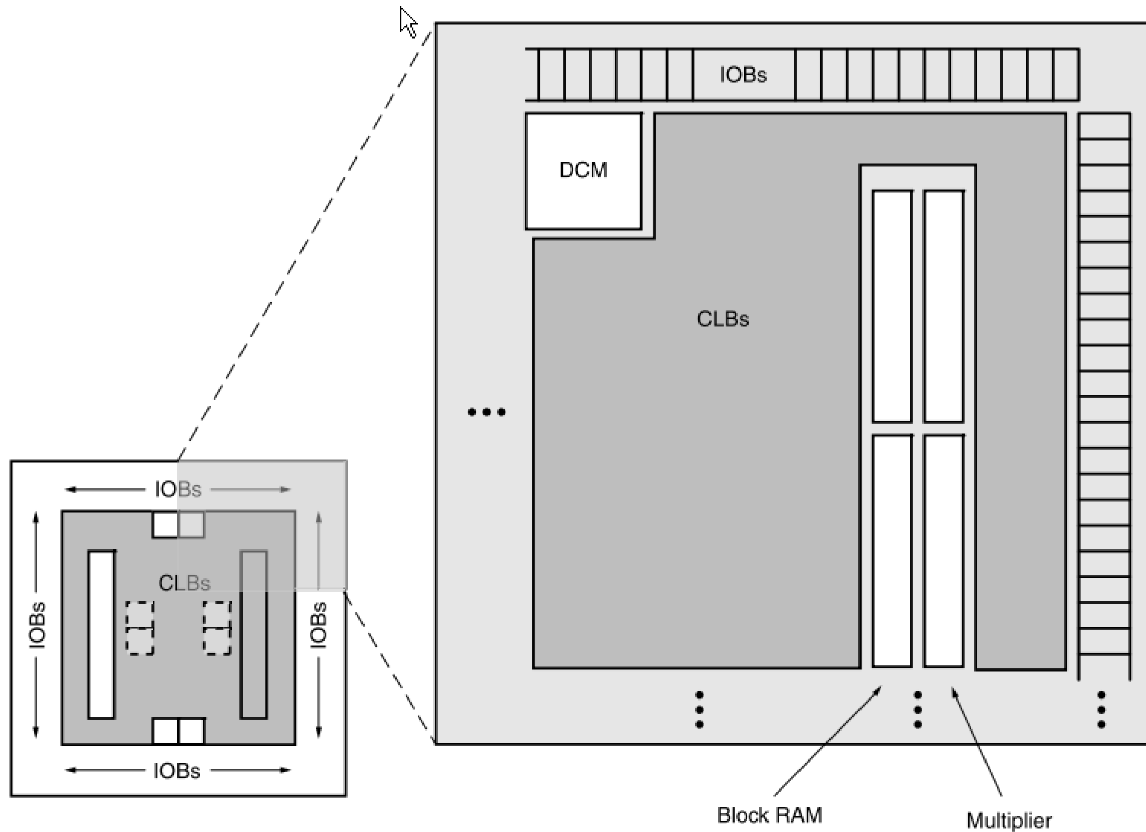
```
Advanced HDL Synthesis Report

Macro Statistics
# RAMs                                                    : 1
 256x8-bit single-port distributed RAM
```

Usage goes down to 1% (factor of 16)

Possible Implementation #2 (cont)

```
INFO:Xst - HDL ADVISOR - The RAM <Mram_memory> will be implemented on LUTs either because
you have described an asynchronous read or because of currently unsupported block RAM
features. If you have described an asynchronous read, making it synchronous would allow you
to take advantage of available block RAM resources, for optimized device usage and improved
timings. Please refer to your documentation for coding guidelines.
    -----------------------------------------------------------------------
    | ram_type           | Distributed                         |          |
    -----------------------------------------------------------------------
    | Port A                                                              |
    |     aspect ratio   | 256-word x 8-bit                    |          |
    |     clkA           | connected to signal <clk>           | rise     |
    |     weA            | connected to signal <we>            | high     |
    |     addrA          | connected to signal <adr>           |          |
    |     diA            | connected to signal <din>           |          |
    |     doA            | connected to signal <dout>          |          |
```

Possible Implementation #3



Our XC3S500E has 20 blocks of 18kbits each.

Block RAM Inference

- Need to make code match hardware or else synthesizer won't use
  - Additional requirement is that code matches some template that synthesizer "wants"

```
adr_reg <= adr when rising_edge(clk);
dout <= memory(conv_integer(unsigned(adr_reg)));
```

Registered Read Address is the key here…
Usage in slices goes to 0
BRAM : 1/20 used

Xilinx Block RAMs

Block RAM and Distributed RAM are made to be DUAL-PORT RAM. Meaning that two sub-devices can be accessing different addresses at the same time.

```
entity dpblockram is
 port (clk  : in std_logic;
  we   : in std_logic;
  a    : in std_logic_vector(4 downto 0);
  dpra : in std_logic_vector(4 downto 0);
  di   : in std_logic_vector(3 downto 0);
  spo  : out std_logic_vector(3 downto 0);
  dpo  : out std_logic_vector(3 downto 0));
 end dpblockram;
```

```
architecture syn of dpblockram is

 type ram_type is array (31 downto 0) of
   std_logic_vector (3 downto 0);
 signal RAM : ram_type;
 signal read_a : unsigned(4 downto 0);
 signal read_dpra : unsigned(4 downto 0);

 begin
 process (clk)
 begin
    if (clk'event and clk = '1') then
       if (we = '1') then
          RAM(to_integer(unsigned(a))) <= di;
       end if;
       read_a <= unsigned(a);
       read_dpra <= unsigned(dpra);
    end if;
 end process;

 spo <= RAM(to_integer(read_a));
 dpo <= RAM(to_integer(read_dpra));

 end syn;
```

Instantiation of 4kbit Block RAM Without Inference (Spartan II Example)

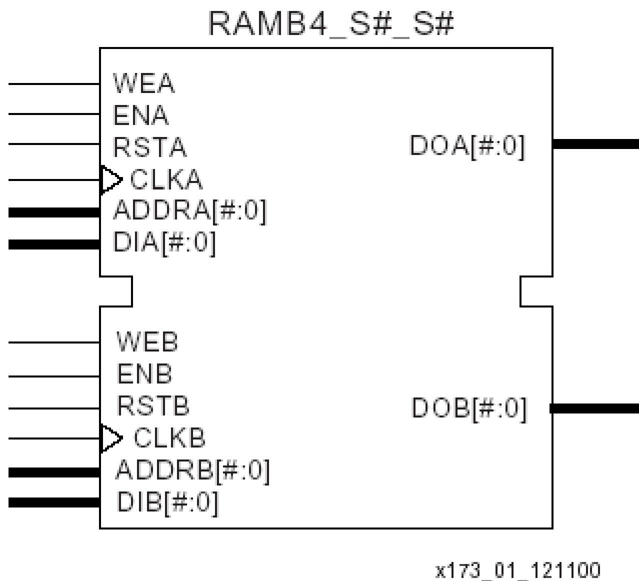


Figure 1: **Dual-port Block SelectRAM+ Memory**

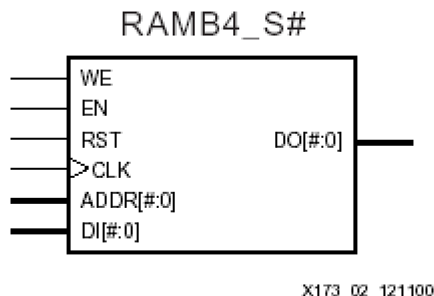


Figure 2: **Single-port Block SelectRAM+ Memory**

Table 3: **Available Library Primitives**

Primitive	Port A Width	Port B Width
RAMB4_S1	1	N/A
RAMB4_S1_S1		1
RAMB4_S1_S2		2
RAMB4_S1_S4		4
RAMB4_S1_S8		8
RAMB4_S1_S16		16
RAMB4_S2	2	N/A
RAMB4_S2_S2		2
RAMB4_S2_S4		4
RAMB4_S2_S8		8
RAMB4_S2_S16		16
RAMB4_S4	4	N/A
RAMB4_S4_S4		4
RAMB4_S4_S8		8
RAMB4_S4_S16		16
RAMB4_S8	8	N/A
RAMB4_S8_S8		8
RAMB4_S8_S16		16
RAMB4_S16	16	N/A
RAMB4_S16_S16		16

Refer to XAPP173

Inference vs. Instantiation Pros/Cons

- Code describes how module works
- Can be ported to another device
- Not dependent on vendor specific libraries for functional simulation
- Demand = increase in capabilities of synthesizers

- No counting on synthesizer to make hardware that you want
- Clear understanding of resource utilization
- When/if you port to different device, you will have to focus on the instantiated subsections, since they won't port.

Some things have no convenient way to model in VHDL, and/or are not supported by current synthesizers. These things must be instantiated using special libraries from the vendor.

Instantiation of a Block RAM w/o Inference

Block Memory Generator v2.1

LogiCORE

Block Memory Generator
v2.1

ADDRA[3:0]
DINA[15:0]
ENA
REGCEA
WEA[0:0]
SSRA
CLKA
DOUTA[15:0]
ADDRB[3:0]
DINB[15:0]
ENB
REGCEB
WEB[0:0]
SSRB
CLKB
DOUTB[15:0]

IP Symbol

Component Name mem

Memory Type
- Single Port RAM
- Simple Dual Port RAM (selected)
- True Dual Port RAM
- Single Port ROM
- Dual Port ROM

Algorithm

Defines the algorithm used to concatenate the block RAM primitives. See the datasheet for more information.

- Minimum Area
- Fixed Primitives (selected)

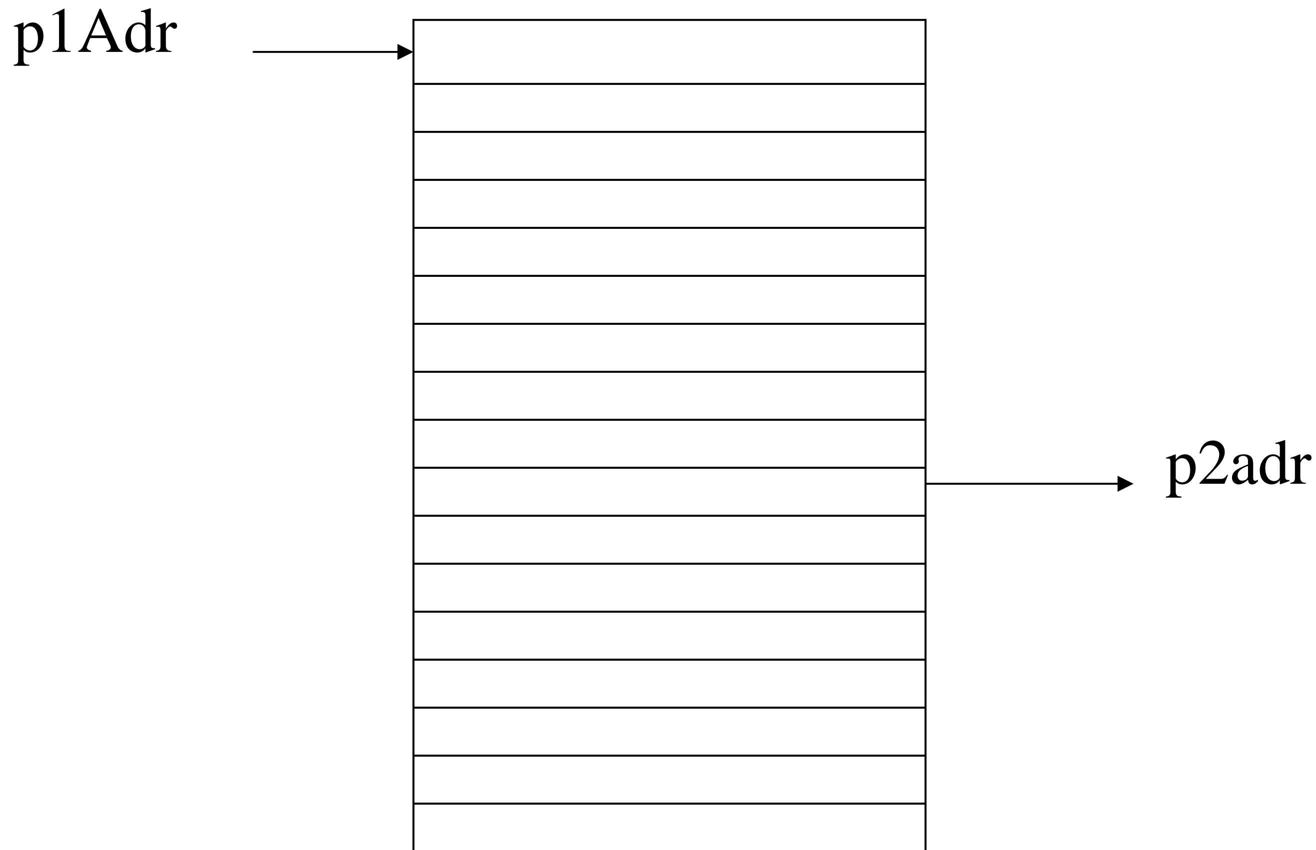
Primitive (Write Port A) : 8kx2

Actual Primitive(s) Used : 8kx2

View Data Sheet | Page 1 of 5 | < Back | Next > | Finish | Cancel

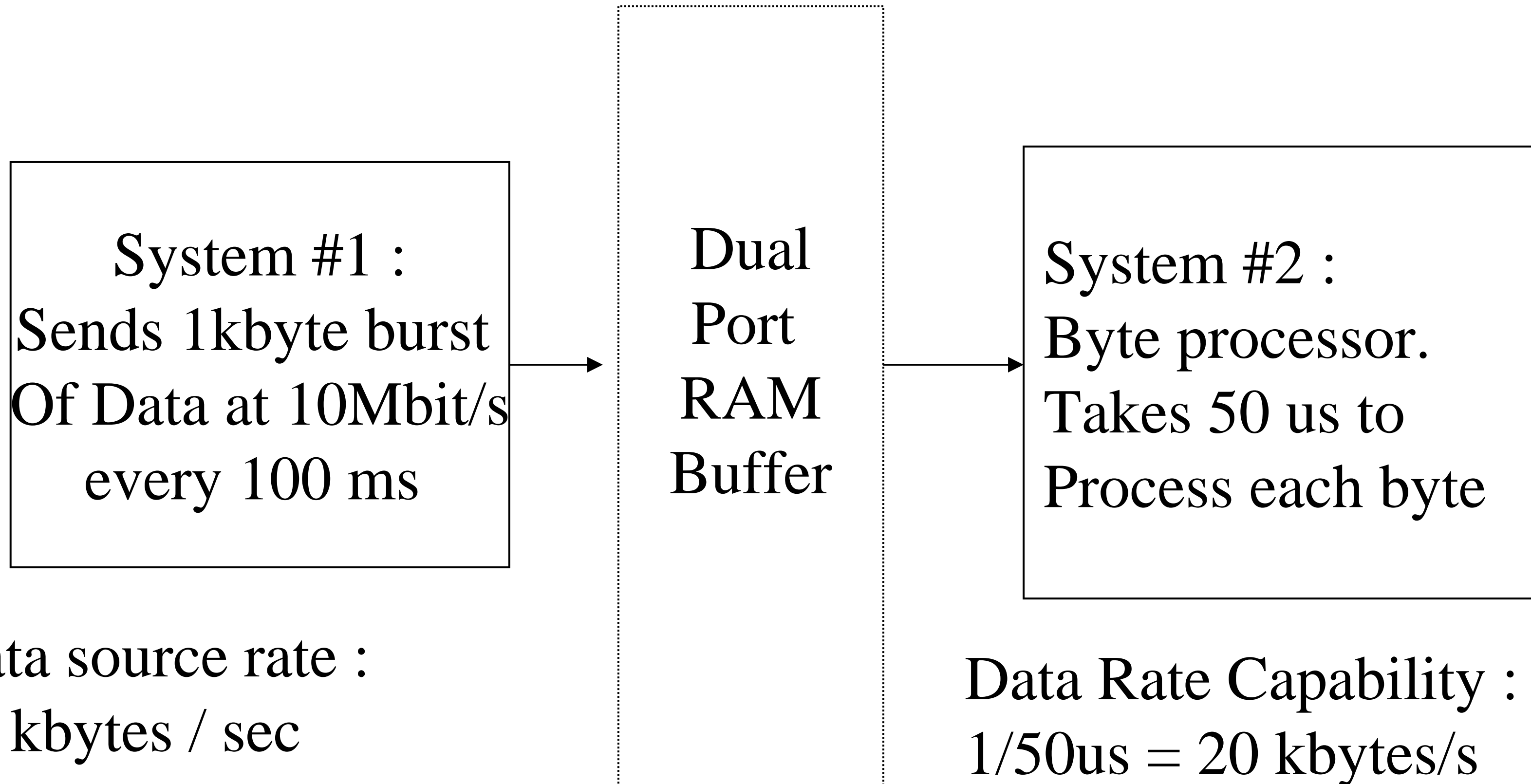
Dual Port RAMs

RAM buffers can then be used as a communication path for two loosely synchronized subsystems on the FPGA, or as an interface between FPGA and outside



This makes a perfect memory for a FIFO. Used for averaging out burst data transfers..etc.

FIFO



DP RAM with some supporting logic (flags and address pointers) is nicely suited to this.

```
architecture behavior of ramfifo32x8 is

 type ram_type is array (31 downto 0) of std_logic_vector (7 downto 0);

 signal RAM : ram_type;
 signal read_adr  : unsigned(4 downto 0);
 signal write_adr : unsigned(4 downto 0);

 begin
    process (clk)
       begin
          if (clk'event and clk = '1') then
              if (reset = '1') then
            read_adr <= to_unsigned(0,read_adr'length);
            write_adr <= to_unsigned(0,write_adr'length);
          end if;
          if (we = '1') then
             RAM(to_integer(write_adr)) <= datai;
             write_adr <= write_adr + 1;
          end if;
          if (re = '1') then
             read_adr <= read_adr + 1;
          end if;
          end if;
       end process;

 datao <= RAM(to_integer(read_adr));

 end behavior;
```

Improvements on this FIFO can implement flags (full, empty) and asynchronous clocks

FIFO Instantiation

Fifo Generator v3.2

LogiCORE

Fifo Generator v3.2

DIN[0:0] → DOUT[0:0]
WR_EN → FULL
WR_CLK → ALMOST_FULL
PROG_FULL
RD_EN → WR_ACK
RD_CLK → OVERFLOW
WR_DATA_COUNT[9:0]
CLK → EMPTY
OG_FULL_THRESH[9:0] → ALMOST_EMPTY
G_EMPTY_THRESH[9:0] → PROG_EMPTY
VALID
_THRESH_ASSERT[9:0] → UNDERFLOW
_THRESH_NEGATE[9:0] → RD_DATA_COUNT[9:0]
_THRESH_ASSERT[9:0] → DATA_COUNT[9:0]
_THRESH_NEGATE[9:0]
RST
SRST

IP Symbol

Component Name fifo

FIFO Implementation

Choose the FIFO implementation from one of the following:

Read/Write Clock Domains	Memory Type	Sup Fea (1)
Common Clock (CLK)	Block RAM	
Common Clock (CLK)	Distributed RAM	
Common Clock (CLK)	Shift Register	
Common Clock (CLK)	Built-in FIFO	
Independent Clocks (RD_CLK, WR_CLK)	Block RAM	X
Independent Clocks (RD_CLK, WR_CLK)	Distributed RAM	
Independent Clocks (RD_CLK, WR_CLK)	Built-in FIFO	

(1) Non-symmetric aspect ratios (different read and write data widths)

(2) First-Word Fall-Through

(3) Uses Virtex 4 and Virtex 5 built in FIFO primitives

View Data Sheet   Page 1 of 6   < Back   Next >   Finish   Cancel

Virtex 4 FIFO

Figure 4-14 shows a top-level view of the Virtex-4 FIFO architecture. The read pointer, write pointer, and status flag logic is dedicated for FIFO use only.

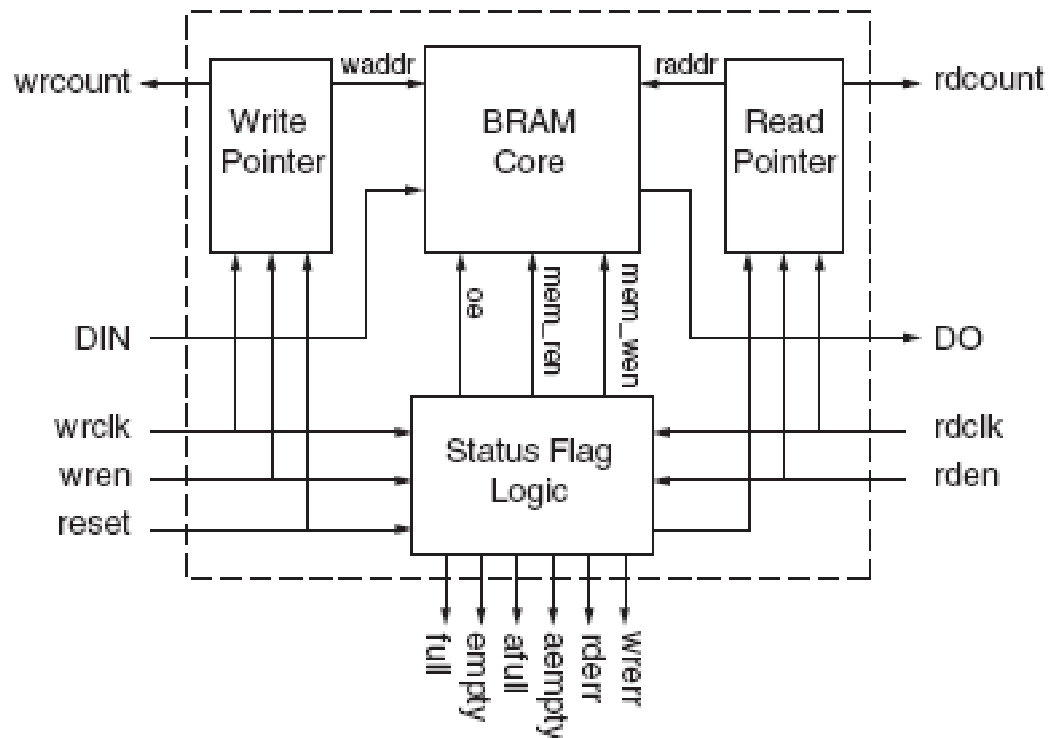


Figure 4-14: **Top-Level View of FIFO in Block RAM**

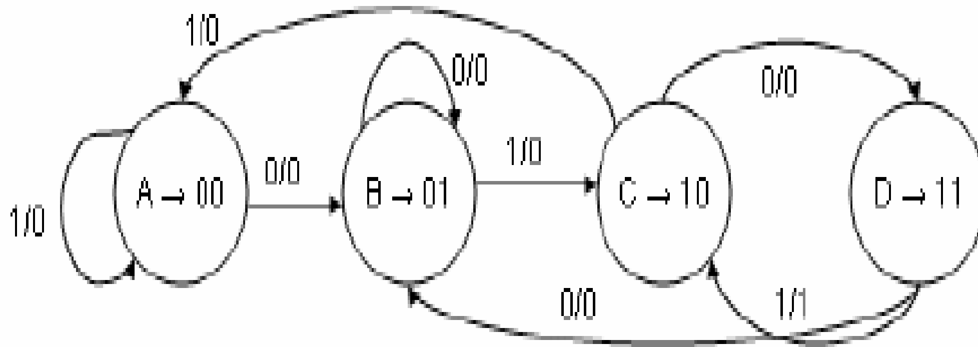
Other Uses: Block RAM Hex to BCD

Addr	10s 1s
00	=> 0000 0000
01	=> 0000 0001
02	=> 0000 0010
03	=> 0000 0011
04	=> 0000 0100
05	=> 0000 0101
06	=> 0000 0110
07	=> 0000 0111
08	=> 0000 1000
09	=> 0000 1001
0A	=> 0001 0000
0B	=> 0001 0001
0C	=> 0001 0010
0D	=> 0001 0011
0E	=> 0001 0100
0F	=> 0001 0101
10	=> 0001 0110
11	=> 0001 0111
12	=> 0001 1000
13	=> 0001 1001
14	=> 0010 0000
15	=> 0010 0001
16	=> 0010 0010
17	=> 0010 0011
18	=> 0010 0100
19	=> 0010 0101
1A	=> 0010 0110
1B	=> 0010 0111
1C	=> 0010 1000
1D	=> 0010 1001
1E	=> 0011 0000
1F	=> 0011 0001

- One of the problems faced on a recent HW was implementing a Hex to BCD converter.
- Note that this is “free” if the Block RAM won’t otherwise be used

Other Uses: FSM In a Block RAM

0101 detector FSM



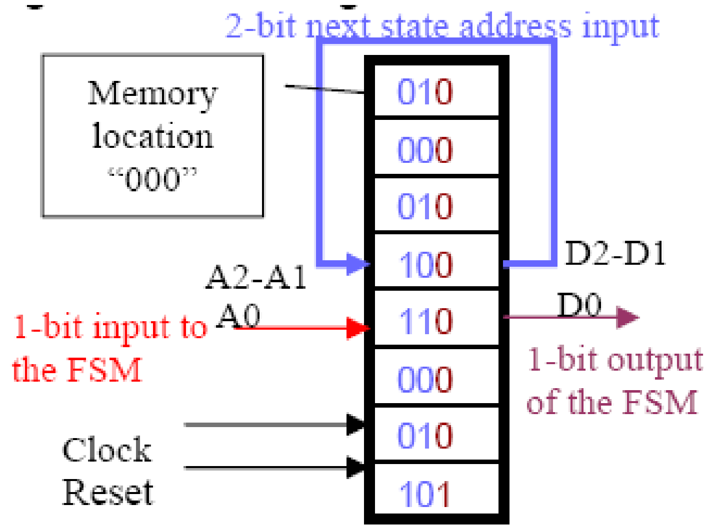
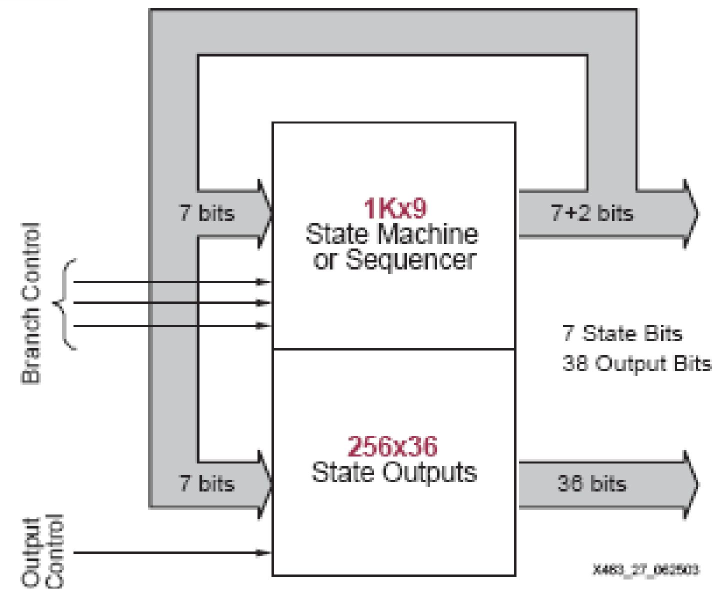


Figure 2b. SEMB implementation



128-State Finite State Machine with 38 Outputs in a Single Block RAM

Refer to Xapp463

Other Block RAM Uses

- Memory for embedded processor
- 20-bit binary counter, or 18-bit binary up-down counter in one block ROM, configured 1K x 18, running up to 200 MHz.
- Sine-cosine look-up tables using one port for sine, the other one for cosine, with 90 degree-shifted addresses, 18-bit amplitude, 10-bit angular resolution.
- The take-home point – if Block RAMs aren't needed for "standard" memory, they can be used to implement any logic that can be thought of as a look-up table

SRL16

- The Shift Register Logic, 16-bits (SRL16) is an alternative mode for the look-up tables where they are used as 16-bit shift registers
- Spartan-3 Generation FPGAs can configure the look-up table (LUT) in a SLICEM slice as a 16-bit shift register without using the flip-flops available in each slice
  - Shift-in operations are synchronous with the clock, and output length is dynamically selectable
  - A separate dedicated output allows the cascading of any number of 16-bit shift registers to create whatever size shift register is needed
- The SRL16 can be automatically inferred by the software tools, but considering their effective use can lead to more cost effective designs
- Can be inferred, instantiated directly, or built using the CoreGen

SRL16 Continued

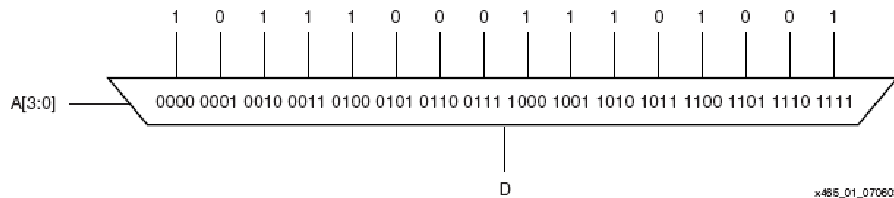


Figure 7-1: **LUT Modeled as a 16:1 Multiplexer**

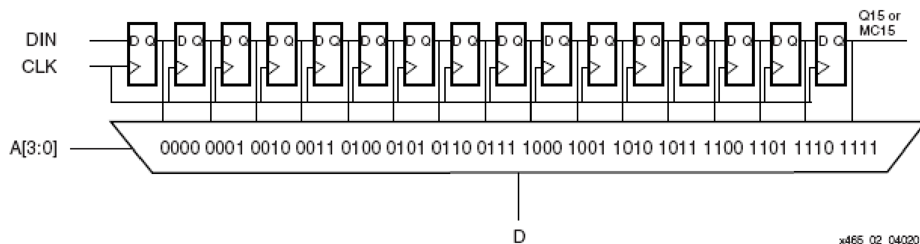


Figure 7-2: **LUT Configured as an Addressable Shift Register**

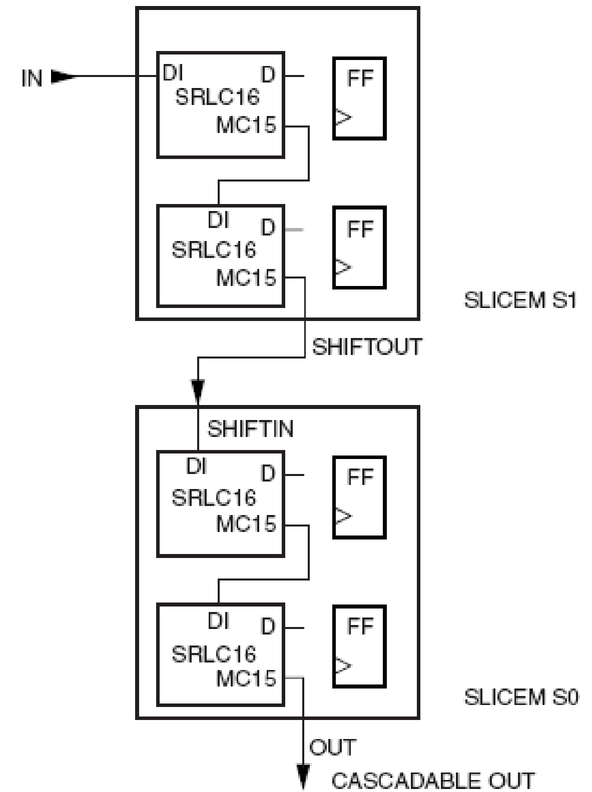


Figure 7-6: **Cascading Shift Register LUTs in a CLB**

SRL16 Continued

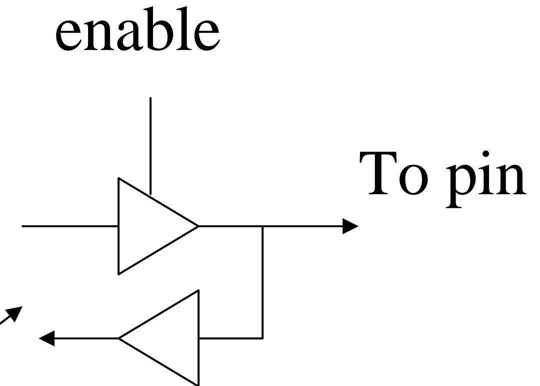
```
signal shiftreg : std_logic_vector(15 downto 0);
begin
process(clk)
begin
        if rising_edge(clk) then
                shiftreg <= shiftreg(14 downto 0) & din;
        end if;
end process;

dout <= shiftreg(15);

end behavioral;
```

Only 2?

Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	1	4656	0%
Number of Slice Flip Flops	2	9312	0%
Number of 4 input LUTs	1	9312	0%
Number of bonded IOBs	3	66	4%
Number of GCLKs	1	24	4%

Tri-state buffer

Almost universally, FPGAs have **pins** with this capability. This way FPGA is capable of communicating with bidirectional bus devices in the real world.



Whole bunch of these are hooked together on board, with a scheme to make sure that no two devices try to talk on the bus at the same time.

Though tri-states used to be a "special feature" where you would have to use a vendor supplied component. Synthesis tools now allow inference of these almost universally.

Xilinx S2 I/O

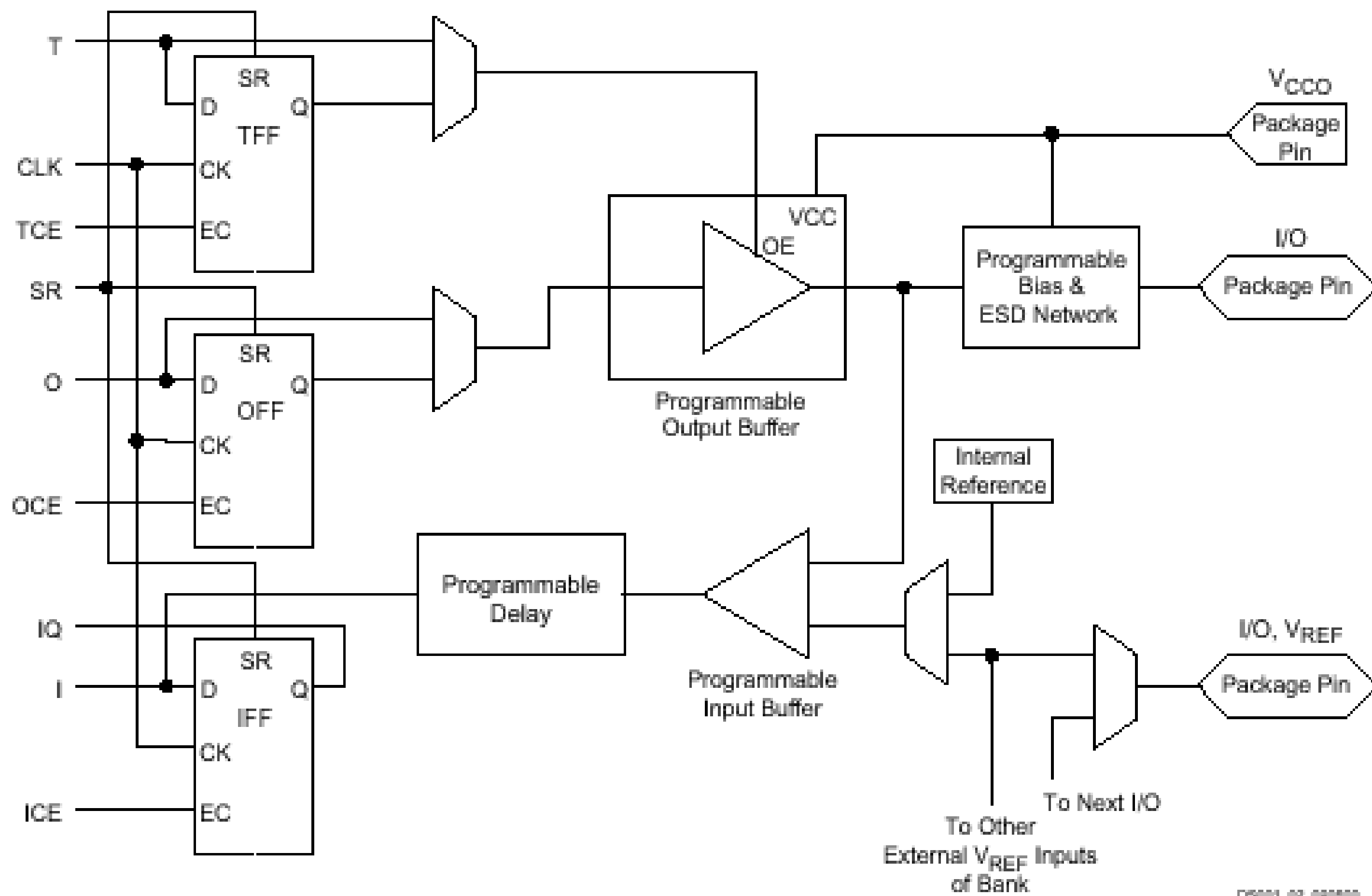
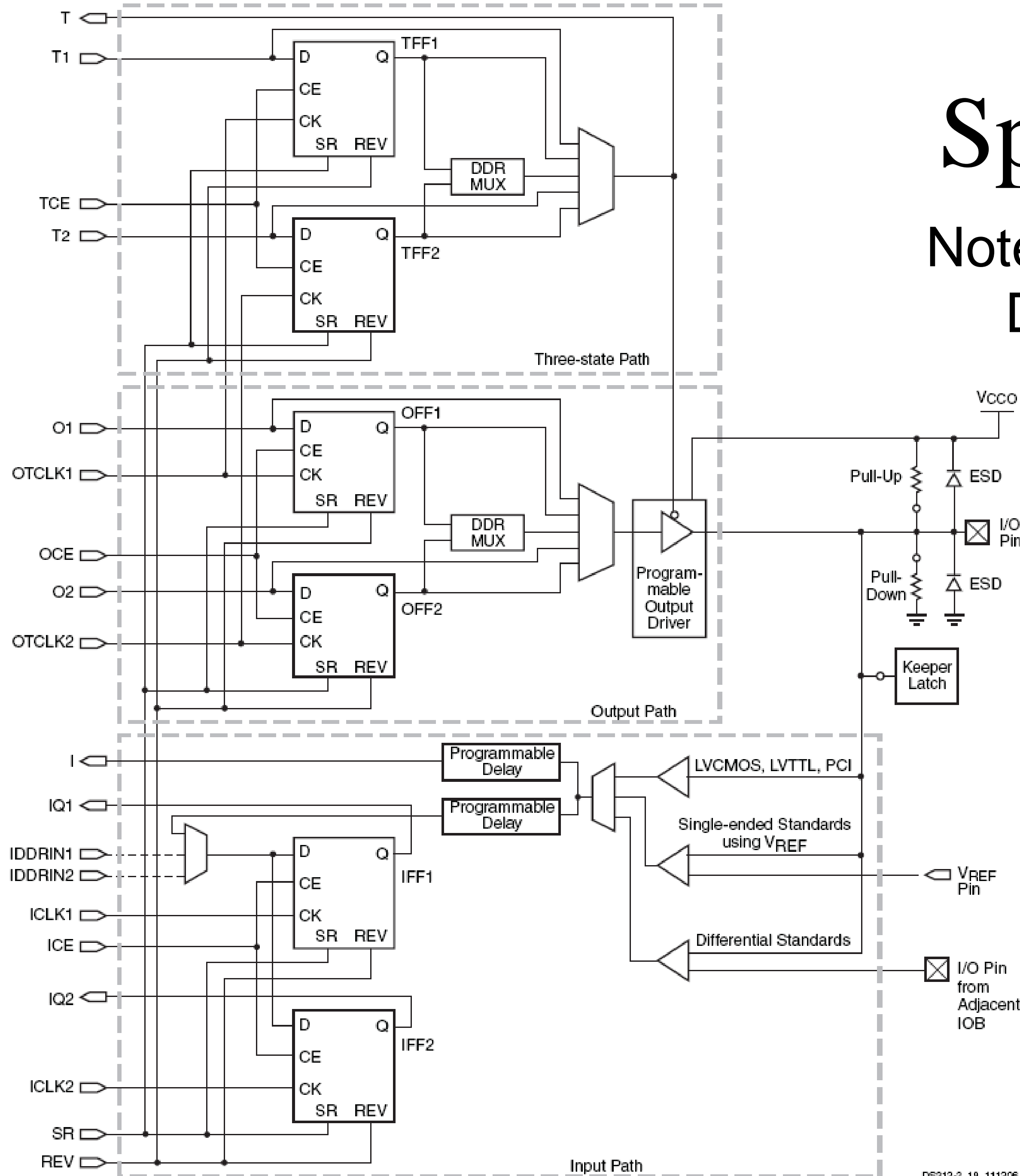


Figure 1: **Spartan-II Input/Output Block (IOB)**



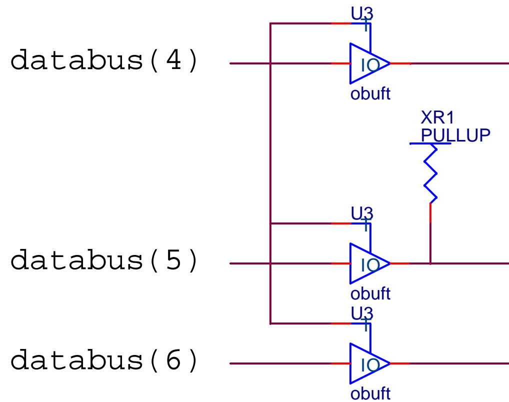
Spartan 3 I/O

Note the addition of the DDR and keeper

Inference of tri-state I/O

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity test is
    Port ( databus : inout std_logic_vector(7 downto 0);
           drive_data : in std_logic;
           value_in : in std_logic_vector(3 downto 0));
end test;

architecture Behavioral of test is
   signal databus_i : std_logic_vector(databus'range);

begin

   databus_i <= value_in & value_in;
   databus <= databus_i when drive_data = '1' else (others => 'Z');

end Behavioral
```

Process with if/then is also appropriate




```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
library unisim;
use unisim.all;
entity test is
   Port ( databus : inout std_logic_vector(7 downto 0);
          drive_data : in std_logic;
          value_in : in std_logic_vector(4 downto 0));
end test;

architecture Behavioral of test is

  signal databus_i : std_logic_vector(databus'range);

  component PULLUP
    port (O: out std_logic);
  end component;

begin

  databus_i <= value_in & value_in;
  databus <= databus_i when drive_data = '1' else (others => 'Z');

  U1: PULLUP port map (O=>databus(5));

end Behavioral
```

Pullups / Pulldowns

- Xilinx S2/S3 has option of pullup, pulldown, "keeper"
- Can only be attached to tri-state I/O pins
- Can not presently be inferred by synthesis
- keeper = weak-keeper circuit
  - eliminates bus chatter when no-one has control of bus (bus is tri-stated)
  - in-place of bus pullups, pulldowns, which always return the bus to a all '0's or '1's state when it floats

Pullup in UCF File

- Easier Method of Adding the weak pullup or down to the design.

NET "reset" LOC = "I14" | PULLUP ;

Internal tri-states

Not universal in FPGAs !

```
architecture Behavioral of mytstate is
signal int_tstate : std_logic_vector(7 downto 0);
signal my_reg : std_logic_vector(7 downto 0);
begin
process(clk)
begin
   if rising_edge(clk) then
            if enable = '1' then
                     my_reg <= int_tstate;
                  end if;
         end if;
end process;

int_tstate <= my_reg when a = '1' else "ZZZZZZZZ";
int_tstate <= my_reg_2 when b = '1' else "ZZZZZZZZ";
int_tstate <= my_reg_3 when c = '1' else "ZZZZZZZZ";
```

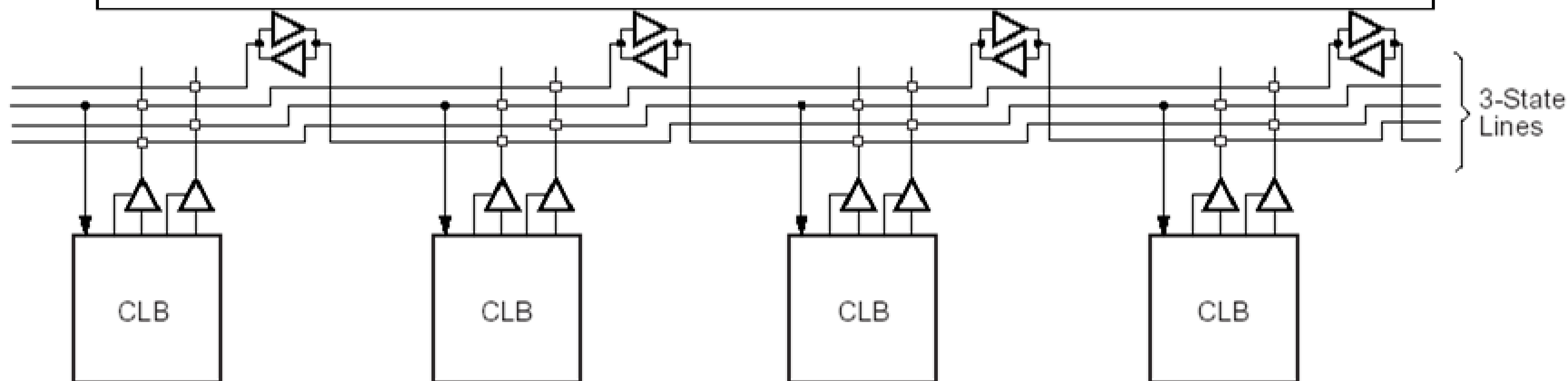
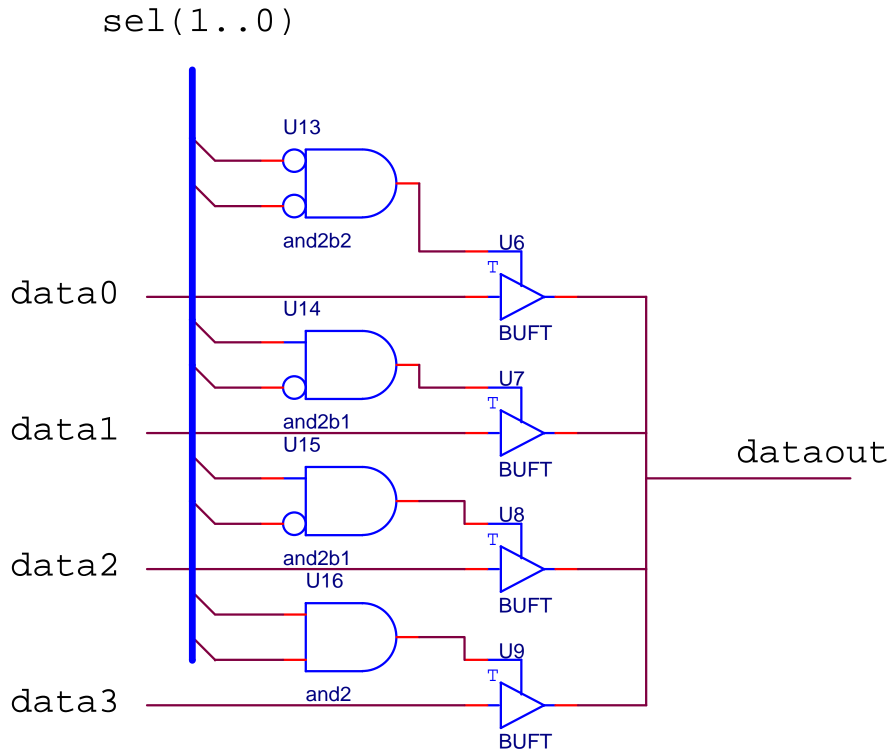


Figure 6: **BUFT Connections to Dedicated Horizontal Bus Lines**

BUFT Multiplexers



dataout <= data0 when sel = “00” else “ZZ”;
dataout <= data1 when sel = “01” else “ZZ”;
dataout <= data2 when sel = “10” else “ZZ”;
dataout <= data3 when sel = “11” else “ZZ”;

BUFT Mux App (An ALU)

```
compute_result : process(input_a,input_b,cin,control,bitexp)
begin
  case control is
        when ALU_ADD =>
              result_i <=  abus+bbus;
        when ALU_SUB =>
              result_i <= abus-bbus;
        when ALU_INC =>
              result_i <= abus + 1;
… and so on …
```

```
result_i <=  abus+bbus when control = ALU_ADD else (others => 'Z') ;

result_i <= abus-bbus when control =ALU_SUB else (others => 'Z');

result_i <= abus + 1 when control =ALU_INC else (others => 'Z');

… and so on …
```

Synthesis Comparison

Original ALU Cell Usage :

# BELS	: 168
# GND	: 1
# LUT2	: 1
# LUT3	: 77
# LUT4	: 19
# MUXCY	: 8
# MUXF5	: 35
# MUXF6	: 18
# XORCY	: 9

Delay: 13.423ns (Levels of Logic = 16)
Source: control_3
Destination: z_flag

BUFT Muxed Cell Usage :

# BELS	: 173
# GND	: 1
# LUT1	: 24
# LUT2	: 41
# LUT4	: 42
# MUXCY	: 32
# VCC	: 1
# XORCY	: 32
# Tri-States	: 144
# BUFT	: 144

Delay: 6.993ns (Levels of Logic = 12)
Source: input_a_0
Destination: z_flag

FPGA Embedded Internal Tristate vs. Multiplexers

- More space efficient (less area + routing)
- *Possibility* of contention causing damage to chip
- Speed

- Portability
- “Safe”

JTAG Structure

The JTAG port is always active and available before, during,
and after FPGA configuration. Add the BSCAN_SPARTAN3
primitive to the design to create user-defined JTAG instructions
and JTAG chains to communicate with internal logic.

This dedicated JTAG port can be very useful – for example, when using the Xilinx Picoblaze, the program memory can be updated without resynthesizing/reprogramming the entire Xilinx.

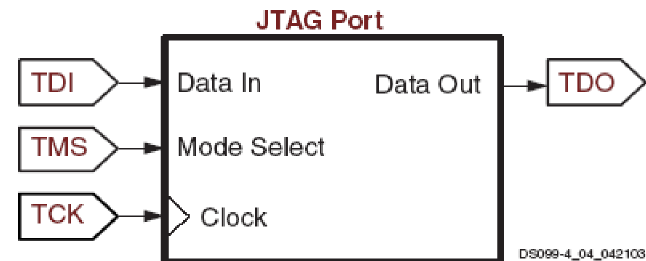


Figure 4: **JTAG Port**

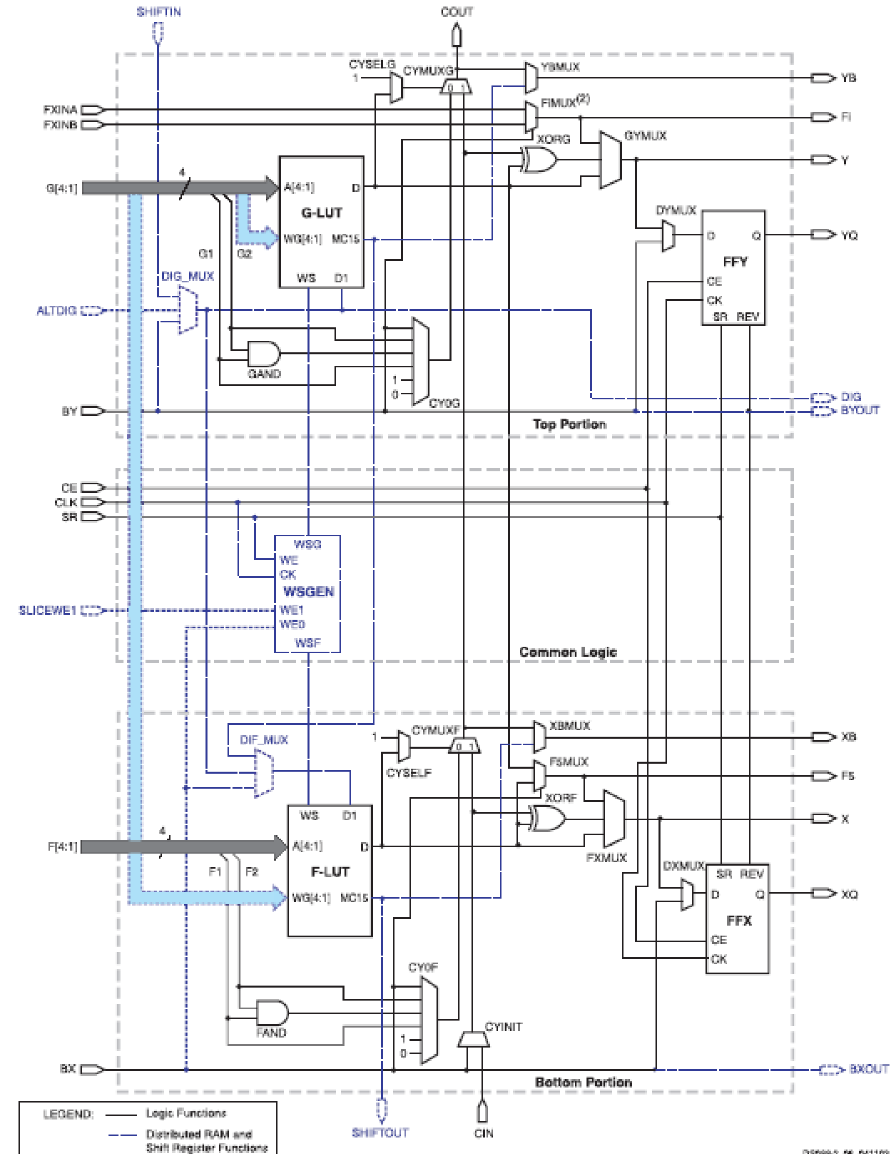
JTAG in KCPSM PROGROM

```
v2_bscan: BSCAN_VIRTEX2
 port map(   TDO1 => tdo1,
       TDO2 => tdo2,
          UPDATE => update,
           SHIFT => shift,
           RESET => reset,
             TDI => tdi,
            SEL1 => sel1,
           DRCK1 => drck1,
            SEL2 => sel2,
           DRCK2 => drck2,
     CAPTURE => capture);
```

And obviously then lots of logic to use these signals

S3 Dedicated Multiplexers

- The SPARTAN 3 does not have internal tri-states
- Rather, the S3 has dedicated multiplexers in each of the CLBs, so logic defined using tri-state buffers is implemented in high-speed dedicated multiplexers

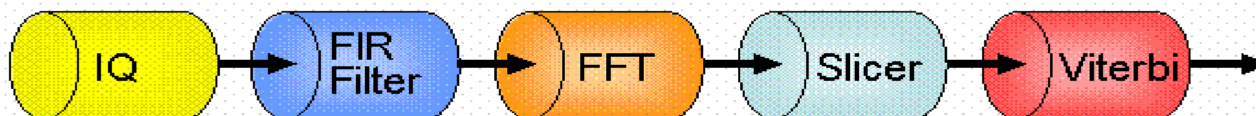


Embedded Multipliers

- A major "niche" FPGAs are increasingly filling is as dedicated Digital Signal Processing (DSP) processors/co-processors
  - For a long time this was not the case – FPGAs did not have the processing capability, and more importantly, didn't have the toolsets to support DSP design/verification
  - Now products exist so that this situation is greatly improved – consider Xilinx free IP and Modelsim/MATLAB co-simulation tools
- Parallelism and high-clock speeds = lots of processing
  - For example, consider a Rake Receiver
  - Operations that are applied to a relatively few number of samples (memory is "scarce") and can occur in parallel are perfect for FPGAs
- Adding multipliers to FPGAs simplifies life – the conversion of algorithms to fixed point is understood, but additional complication is added when algorithms have to be tuned to eliminate multiplication

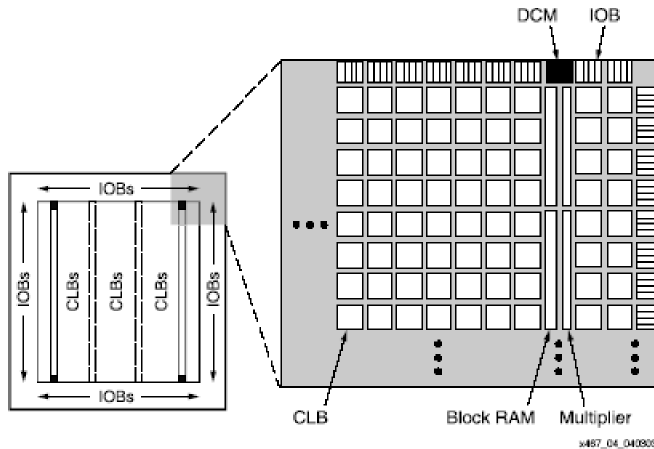
DSP versus FGPA

- Altera has implemented the new benchmark on one of its forthcoming Stratix FPGAs and has reported preliminary results to BDTI.
- Similarly, Motorola has provided preliminary results for one of its high-end DSPs, the 300 MHz MSC8101 (based on the StarCore SC140 core).
- **Based on these results, it appears that even a mid-range DSP-enhanced FPGA from Altera's Stratix line will be able to handle more than an order of magnitude more channels than the MSC8101-for a similar projected per-chip price**.



BDTI's new telecom benchmark for FPGAs (field-programmable gate arrays) is a simplified single-channel receiver, shown in the block diagram above. The IQ block performs demodulation into in-phase (I) and quadrature (Q) signals; the Slicer block maps fast Fourier transform (FFT) outputs to points in a QAM (quadrature amplitude modulation) constellation. The receiver benchmark is intended to be representative of the kinds of processing found in communications infrastructure equipment for applications such as DSL, cable modems, and fixed wireless systems.
Source: *FPGAs for DSP*, Copyright © 2002 Berkeley Design Technology, Inc.

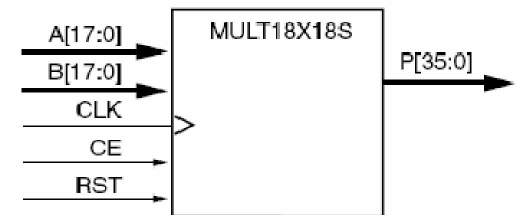
Embedded Multipliers in Spartan III



Notes:
1. The two additional block RAM/multiplier columns of the XC3S4000 and XC3S5000 devices are shown with dashed lines. The XC3S50 device has a single column of block RAM/multipliers along the left edge.

Figure 4: **Location of Multipliers in Spartan-3 Architecture**

(a) Asynchronous 18-bit Multiplier



(b) 18-bit Multiplier with Register at Outputs

One per Block RAM

Table 1: **Summary of Spartan-3 FPGA Attributes**

Device	System Gates	Equivalent Logic Cells	CLB Array (One CLB = Four Slices)			Distributed RAM (bits[1])	Block RAM (bits[1])	Dedicated Multipliers	DCMs	Maximum User I/O	Maximum Differential I/O Pairs
			Rows	Columns	Total CLBs						
XC3S50[2]	50K	1,728	16	12	192	12K	72K	4	2	124	56
XC3S200[2]	200K	4,320	24	20	480	30K	216K	12	4	173	76
XC3S400[2]	400K	8,064	32	28	896	56K	288K	16	4	264	116
XC3S1000[2, 3]	1M	17,280	48	40	1,920	120K	432K	24	4	391	175
XC3S1500[3]	1.5M	29,952	64	52	3,328	208K	576K	32	4	487	221
XC3S2000	2M	46,080	80	64	5,120	320K	720K	40	4	565	270
XC3S4000[3]	4M	62,208	96	72	6,912	432K	1,728K	96	4	712	312
XC3S5000	5M	74,880	104	80	8,320	520K	1,872K	104	4	784	344

Example of Multiplier Use, Instantiation

```
-- Component Declaration for MULT18X18 should be placed
-- after architecture statement but before begin keyword

component MULT18X18
   port (P : out STD_LOGIC_VECTOR (35 downto 0);
         A : in STD_LOGIC_VECTOR (17 downto 0);
         B : in STD_LOGIC_VECTOR (17 downto 0));
end component;


-- Component Instantiation for MULT18X18 should be placed
-- in architecture after the begin keyword
MULT18X18_INSTANCE_NAME : MULT18X18
port map (P => user_P,
          A => user_A,
          B => user_B);
```

Any FPGA with an embedded multiplier will allow the multiplier block to be instantiated like this. However, the nice thing about multipliers is that they are easily recognized by the synthesizer and instantiation is at most rarely used.

Example of Multiplier Use, Infer

(Old Exam Problem)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.ALL;

entity filter is
    generic ( widthIn : natural := 8;
              filterLen : natural := 8);
    port ( clock    : in std_logic;
           loadTaps : in std_logic;
           taps     : in std_logic_vector(widthIn-1 downto
      0);
           x        : in std_logic_vector(widthIn-1 downto
      0);
           y        : out std_logic_vector(widthIn*2-1
      downto 0) );
end filter;

architecture Behavioral of filter is
   type arrayType is array(filterLen-1 downto 0) of
      unsigned(widthIn-1 downto 0);
   signal tapsArray : arrayType;
   signal xArray    : arrayType;
   signal yInt      : unsigned(15 downto 0);
begin

   load : process(clock)
      begin
         if rising_edge(clock) and loadTaps = '1' then
            tapsArray(filterLen-1 downto 1) <=
      tapsArray(filterLen-2 downto 0);
            tapsArray(0) <= unsign
         end if;
      end process;
.
.
.
```

```
.
.
.
   filt : process(clock)
   begin
      if rising_edge(clock) then
         xArray(filterLen-1  downto 1) <= xArray(filterLen-2
     downto 0);
         xArray(0) <= unsigned(x);
      end if;
   end process;

   --function "*" (L, R: UNSIGNED) return UNSIGNED;
   -- Result subtype: UNSIGNED((L'LENGTH+R'LENGTH-1) downto
      0).
   -- Result: Performs the multiplication operation on two
      UNSIGNED vectors
   --         that may possibly be of different lengths.
   yInt <= xArray(0) * tapsArray(0) +
           xArray(1) * tapsArray(1) +
           xArray(2) * tapsArray(2) +
           xArray(3) * tapsArray(3) +
           xArray(4) * tapsArray(4) +
           xArray(5) * tapsArray(5) +
           xArray(6) * tapsArray(6) +
           xArray(7) * tapsArray(7);



   y <= std_logic_vector(yInt);
```

Device utilization summary:		
Number of External IOBs	34 out of 97	35%
Number of LOCed External IOBs	0 out of 34	0%
Number of MULT18X18s	**8 out of 12**	**66%**
Number of Slices	121 out of 1920	6%
Number of BUFGMUXs	1 out of 8	12%

Example of Multiplier Use, Infer (MULT18X18S)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity mult18x18s is
    port (a : in std_logic_vector(7 downto 0);
          b : in std_logic_vector(7 downto 0);
          clk : in std_logic;
          prod : out std_logic_vector(15 downto 0));
end mult18x18s;

architecture arch_mult18x18s of mult18x18s is
begin
    process(clk) is begin
          if clk'event and clk = '1' then
                    prod <= a*b;
          end if;
    end process;
end arch_mult18x18s;
```

This form allows XST or Synplify to infer the registered block multiplier.

However, this form won't work for Leonardo Spectrum, which requires the "prod" to be copied as:

prod_reg <= prod;

This is informative when discussing inferring "special structures", and in fact any structure, on an FPGA. When moving towards targeting features of a particular FPGA, knowledge of the target becomes important, and influences the coding style.

As another example, consider how specifying the reset condition for a flip-flop in an Xilinx structure means that after programming the states are known, even without an explicit reset (the end of the programming process resets). VHDL written for a Xilinx could presume this behavior, meaning the code isn't as portable as it appears to be.

Chaining Multipliers

```
Multiply Using 8-bit multiplier

               1010 1101
            *  0110 0010
               ---------
               0000 0000
              1010 1101
             0000 0000
            0000 0000
           0000 0000
          1010 1101
         1010 1101
    +   0000 0000
      ------------------

Multiply Using 4-bit multipliers

1101 * 0010 + 0010*1010 << 4 + 0110*1101 << 4 + 0110*1010 << 8
```

Note that the bottom four bits of 1101 * 0010 are not added to anything else.

Chaining Multipliers

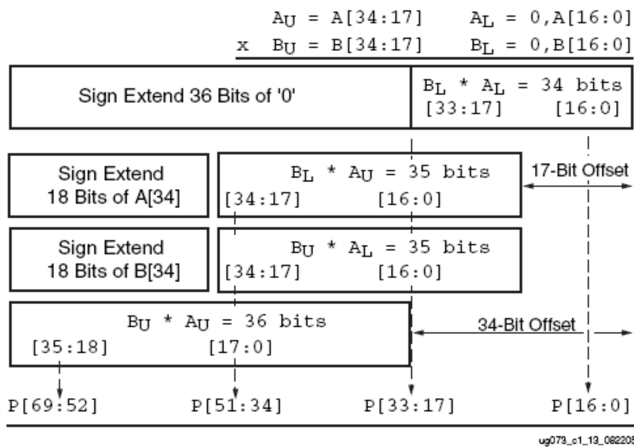


Figure 1-13: **35x35-Bit Multiplication from 18x18-bit Multipliers**

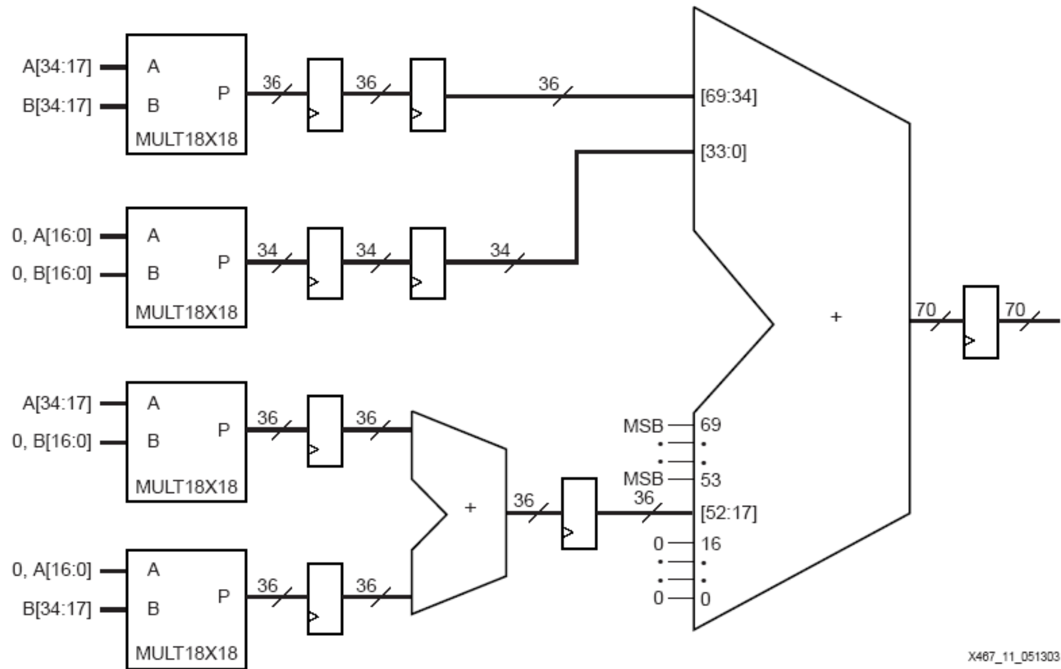


Figure 6: **35x35 Signed Multiplier**

Pg. 35 of UG073

Multiplier Core Generator

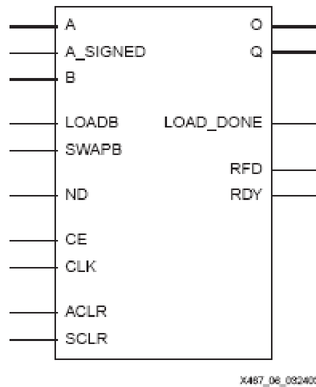


Figure 8: **Core Multiplier Generator Symbol**

One of the uses of generating a multiplier "core" is by the "System Generator" for DSP. While it is easy to include the multiply in our own designs with a "*", having defined interface wrappers allows for the automatic generation of VHDL algorithm implementations.

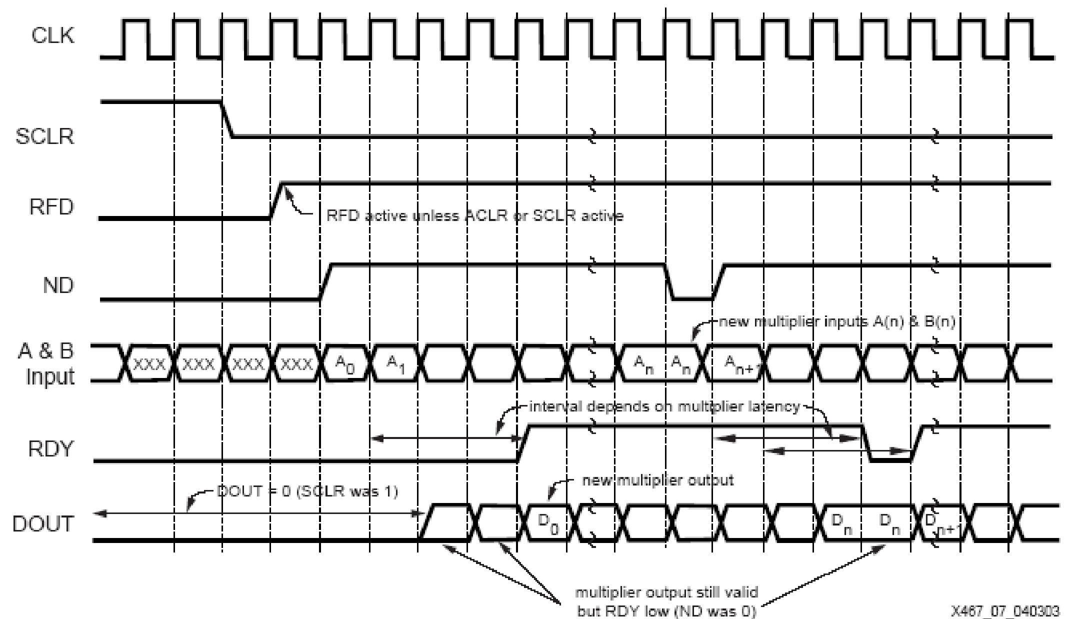


Figure 9: **Multiplier Generator Timing Diagram**

CoreGen

Multiplier

LogiCORE

Multiplier v10.0

A[17:0]
B[17:0]
CLK
CE
SCLR
P[35:0]

Component Name : mult

Multiplier Type
- (selected) Parallel Multiplier
- Constant-Coefficient Multiplier

Input Options

Port A
- Data Type : Signed
- Width : 18 Range: 2..64

Port B
- Data Type : Signed
- Width : 18 Range: 2..64

IP Symbol | Resource Estimates

View Data Sheet | Page 1 of 3 | < Back | Next > | Finish | Cancel

Virtex 4 Xtreme DSP (DSP48)

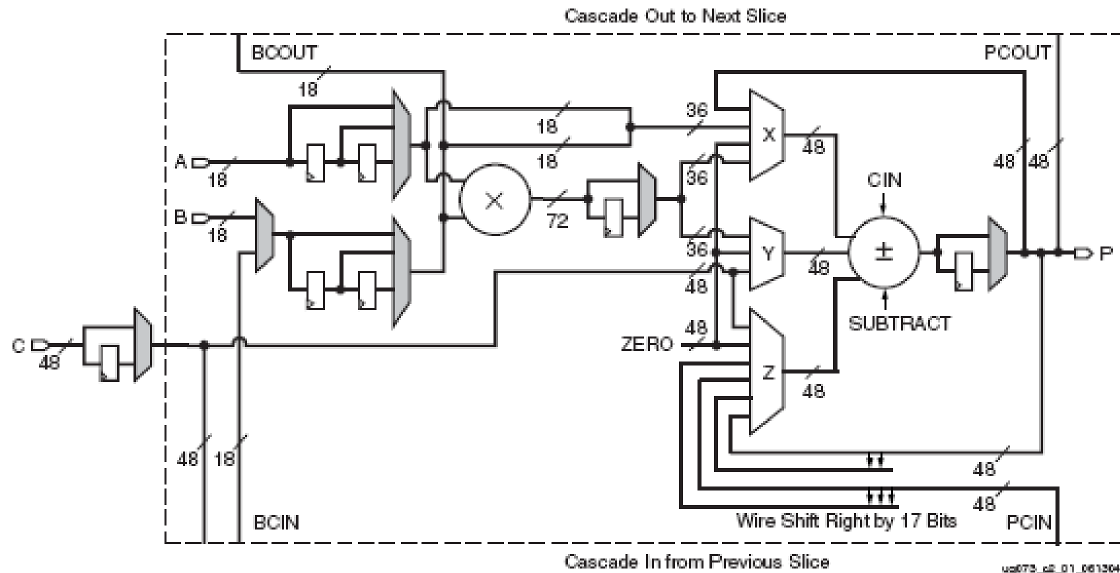


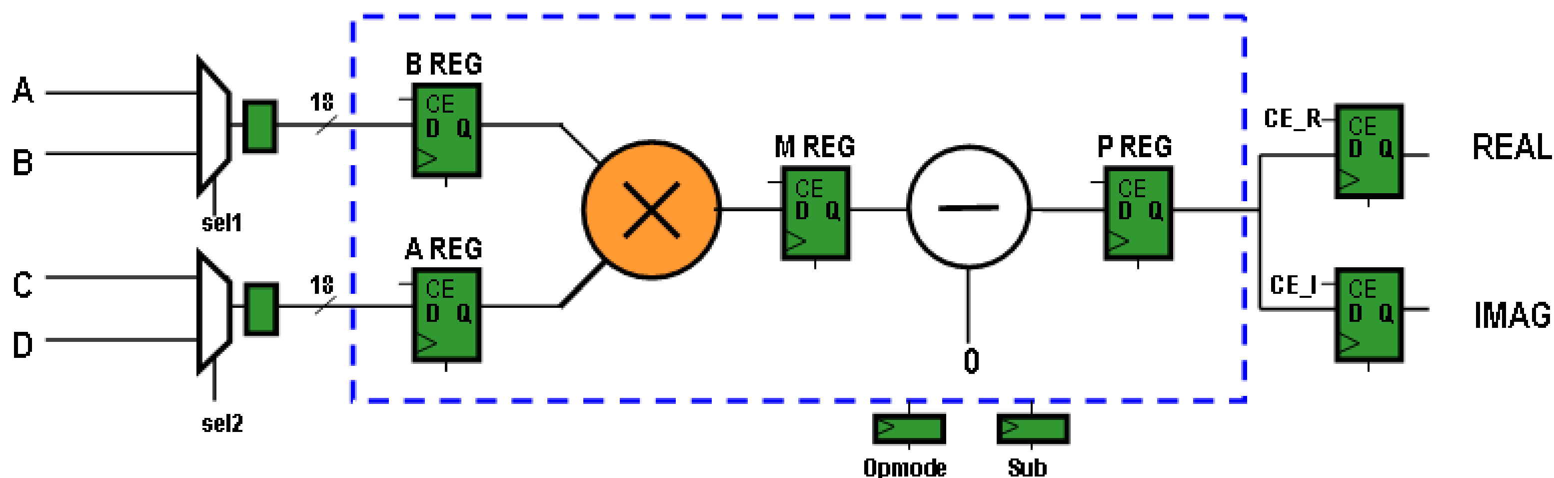
Figure 2-1: **DSP Slice Architecture**

- 18-bit x 18-bit, two's-complement multiplier with a full-precision 36-bit result, sign extended to 48 bits
- Three-input, flexible 48-bit adder/subtracter with optional registered accumulation feedback
- Dynamic user-controlled operating modes to adapt DSP48 slice functions from clock cycle to clock cycle
- Cascading 18-bit B bus, supporting input sample propagation
- Cascading 48-bit P bus, supporting output propagation of partial results
- Input port *C* for multiply-add operation, large three-operand addition, or flexible rounding mode
- Pipeline registers before and after arithmetic stages

Dynamic Opmode

Complex Multiplier

$(a+jb).(c+jd) = [a.c - b.d] + j[b.c + a.d]$



CLK Cycle	Function	OPMODE	Sub	Sel1	Sel2	CE_R	CE_I

Performance
400 Mhz
Size:
1 XDSP Slice
59 Slices
(5 for control)

NOTE: Control signals can be stored in a Disributed Memory

Dynamic Opmode

Complex Multiplier

$$(a+jb).(c+jd) = [a.c - b.d] + j[b.c + a.d]$$

A
B
sel1
C
D
sel2
18
18
B REG
A REG
A x D
M REG
A.D + B.C
P REG
CE_R
CE_I
REAL
IMAG
Opmode
Sub

CLK Cycle	Function	OPMODE	Sub	Sel1	Sel2	CE_R	CE_I
1	**Multiply Subtract**	0001010	1	0	0	0	1
2	**Multiply Accumulate**	0101010	0	1	0	0	0
3	**Multiply**	0001010	0	0	1	1	0
4	**Multiply Accumulate**	0101010	0	1	1	0	0

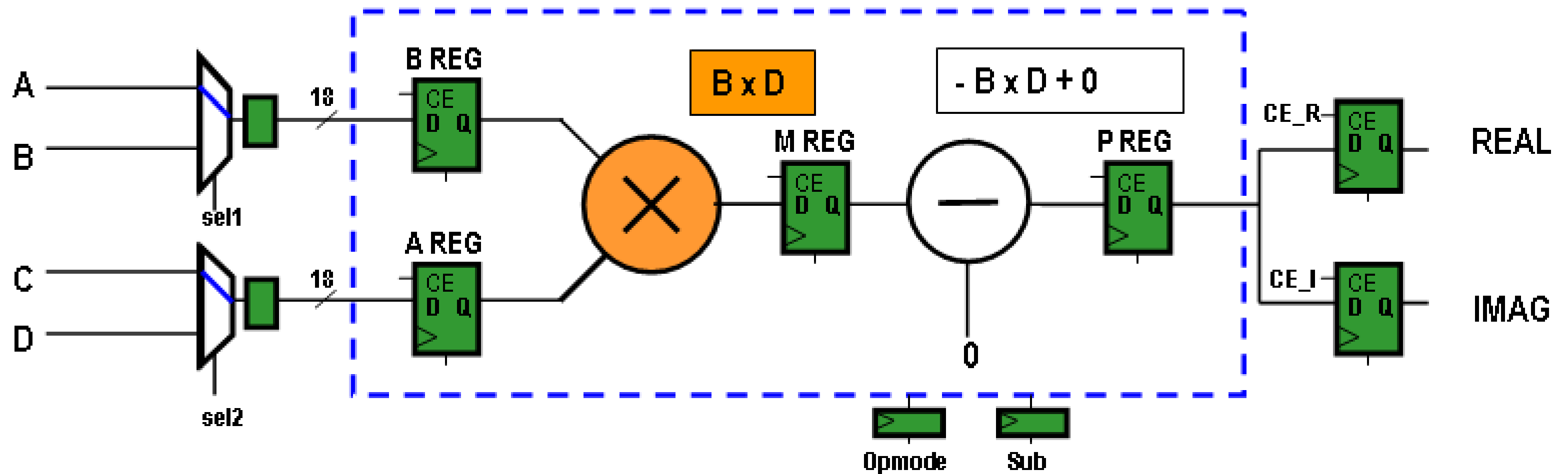
NOTE: Control signals can be stored in a Disributed Memory

Performance
400 Mhz
Size:
1 XDSP Slice
59 Slices
(5 for control)

Dynamic Opmode

Complex Multiplier

$$(a+jb).(c+jd) = [a.c - b.d] + j[b.c + a.d]$$

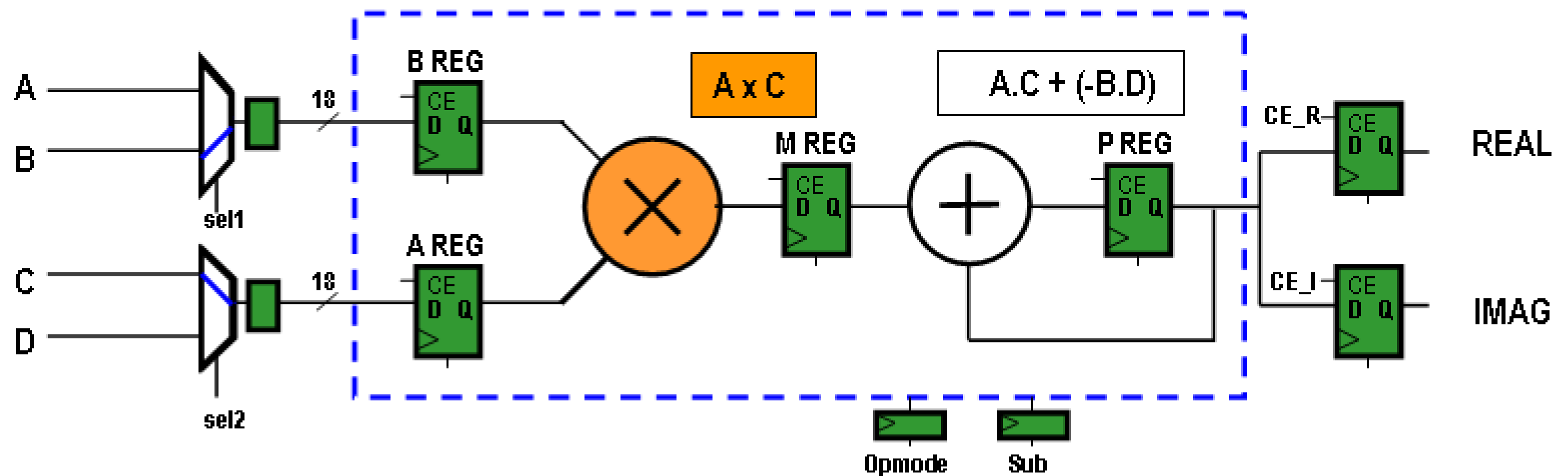


CLK Cycle	Function	OPMODE	Sub	Sel1	Sel2	CE_R	CE_I
1	Multiply Subtract	0001010	1	0	0	0	1

NOTE: Control signals can be stored in a Disributed Memory

Performance
400 Mhz
Size:
1 XDSP Slice
59 Slices
(5 for control)

Dynamic Opmode



CLK Cycle	Function	OPMODE	Sub	Sel1	Sel2	CE_R	CE_I
1	Multiply Subtract	0001010	1	0	0	0	1
2	Multiply Accumulate	0101010	0	1	0	0	0

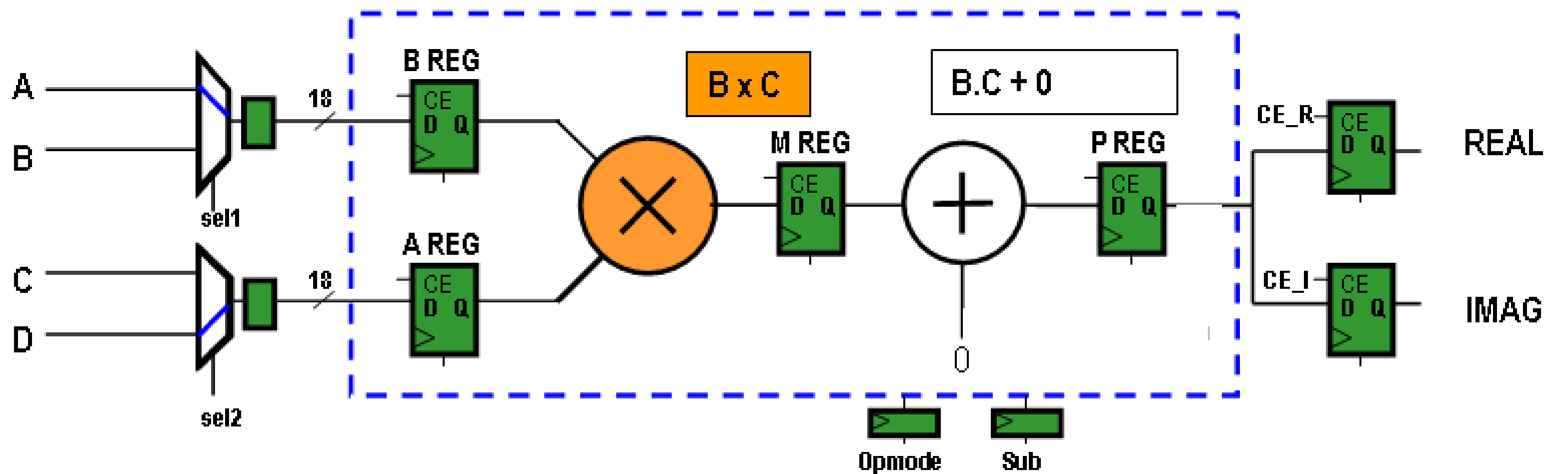
Performance
400 Mhz
Size:
1 XDSP Slice
59 Slices
(5 for control)

NOTE: Control signals can be stored in a Disributed Memory

Dynamic Opmode

Complex Multiplier

(a+jb).(c+jd) = [a.c - b.d] + j[b.c + a.d]



CLK Cycle	Function	OPMODE	Sub	Sel1	Sel2	CE_R	CE_I
1	**Multiply Subtract**	0001010	1	0	0	0	1
2	**Multiply Accumulate**	0101010	0	1	0	0	0
3	**Multiply**	0001010	0	0	1	1	0

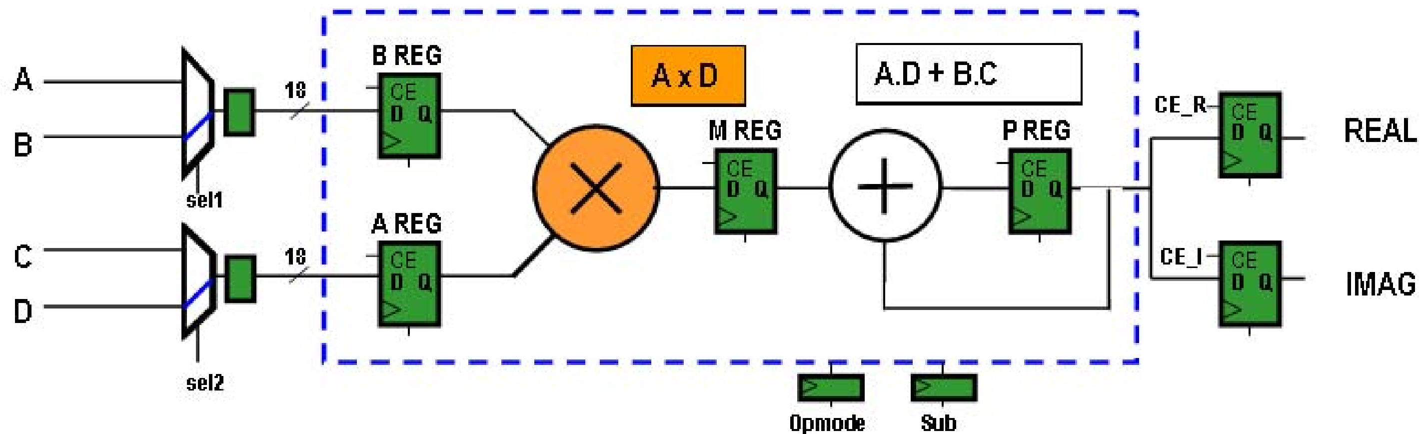
Performance
400 Mhz
Size:
1 XDSP Slice
59 Slices
(5 for control)

NOTE: Control signals can be stored in a Disributed Memory

Dynamic Opmode

Complex Multiplier

$$(a+jb).(c+jd) = [a.c - b.d] + j[b.c + a.d]$$



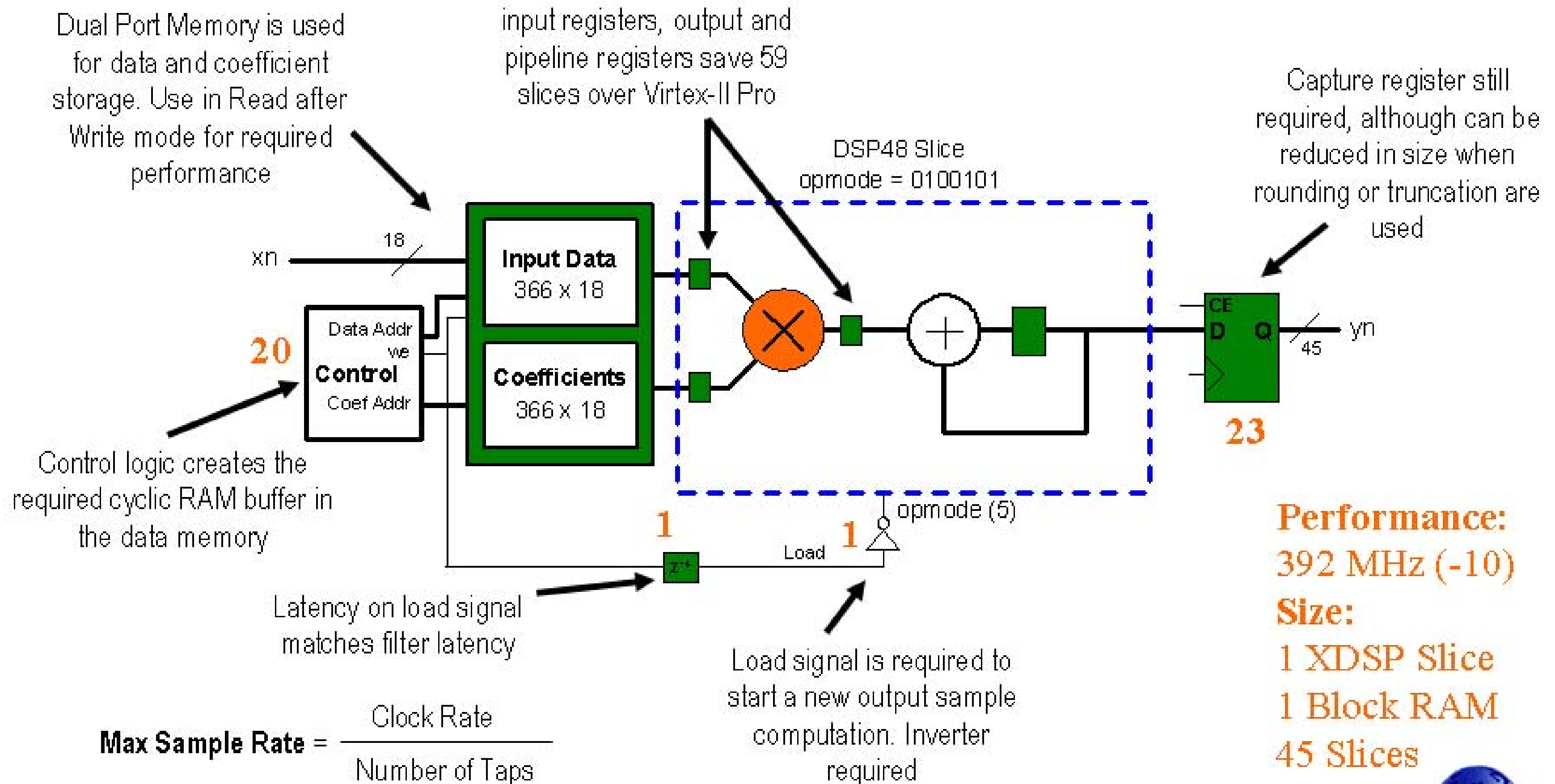
CLK Cycle	Function	OPMODE	Sub	Sel1	Sel2	CE_R	CE_I
1	Multiply Subtract	0001010	1	0	0	0	1
2	Multiply Accumulate	0101010	0	1	0	0	0
3	Multiply	0001010	0	0	1	1	0
4	Multiply Accumulate	0101010	0	1	1	0	0

NOTE: Control signals can be stored in a Disributed Memory

Performance
400 Mhz
Size:
1 XDSP Slice
59 Slices
(5 for control)

Virtex-4 MAC FIR Filter

Filter Specification: Sampling Frequency = 1.2288 Mhz, Coefficients = 366



Digital Clock Manager (DCM)

- Multiply or divide an incoming clock frequency or synthesize a completely new frequency by a mixture of clock multiplication and division.
- DCM includes a Delay Locked Loop (DLL) to eliminate skew between both internal and external clock distribution networks
- Phase Adjustment
- Also includes an "Arbitrary" clock generator, called the Digital Frequency Synthesizer (DFS)
  - Useful for many things – adjust speed for power issues, etc.
- Clock-Conditioning
- Dual-Data Rate I/O

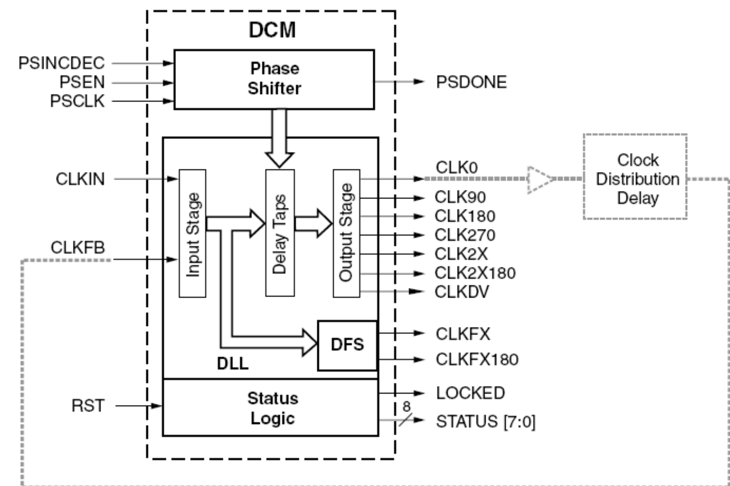


Figure 3-3: **DCM Functional Block Diagram**

VHDL Template

```
DCM_inst : DCM
  generic map (
     CLKDV_DIVIDE => 2.0, --  Divide by: 1.5,2.0,2.5,3.0,3.5,4.0,4.5,5.0,5.5,6.0,6.5
                          --     7.0,7.5,8.0,9.0,10.0,11.0,12.0,13.0,14.0,15.0 or 16.0
     CLKFX_DIVIDE => 1,   --  Can be any integer from 1 to 32
     CLKFX_MULTIPLY => 4, --  Can be any integer from 1 to 32
     CLKIN_DIVIDE_BY_2 => FALSE, --  TRUE/FALSE to enable CLKIN divide by two feature
     CLKIN_PERIOD => 0.0,          --  Specify period of input clock
     CLKOUT_PHASE_SHIFT => "NONE", --  Specify phase shift of NONE, FIXED or VARIABLE
     CLK_FEEDBACK => "1X",         --  Specify clock feedback of NONE, 1X or 2X
     DESKEW_ADJUST => "SYSTEM_SYNCHRONOUS", --  SOURCE_SYNCHRONOUS, SYSTEM_SYNCHRONOUS or
                                            --     an integer from 0 to 15
     DFS_FREQUENCY_MODE => "LOW",     --  HIGH or LOW frequency mode for frequency synthesis
     DLL_FREQUENCY_MODE => "LOW",     --  HIGH or LOW frequency mode for DLL
     DUTY_CYCLE_CORRECTION => TRUE, --  Duty cycle correction, TRUE or FALSE
     FACTORY_JF => X"C080",          --  FACTORY JF Values
     PHASE_SHIFT => 0,        --  Amount of fixed phase shift from -255 to 255
     STARTUP_WAIT => FALSE) --  Delay configuration DONE until DCM LOCK, TRUE/FALSE
  port map (
     CLK0 => CLK0,     -- 0 degree DCM CLK ouptput
     CLK180 => CLK180, -- 180 degree DCM CLK output
     CLK270 => CLK270, -- 270 degree DCM CLK output
     CLK2X => CLK2X,   -- 2X DCM CLK output
     CLK2X180 => CLK2X180, -- 2X, 180 degree DCM CLK out
     CLK90 => CLK90,   -- 90 degree DCM CLK output
     CLKDV => CLKDV,   -- Divided DCM CLK out (CLKDV_DIVIDE)
     CLKFX => CLKFX,   -- DCM CLK synthesis out (M/D)
     CLKFX180 => CLKFX180, -- 180 degree CLK synthesis out
     LOCKED => LOCKED, -- DCM LOCK status output
     PSDONE => PSDONE, -- Dynamic phase adjust done output
     STATUS => STATUS, -- 8-bit DCM status bits output
     CLKFB => CLKFB,   -- DCM clock feedback
     CLKIN => CLKIN,   -- Clock input (from IBUFG, BUFG or DCM)
     PSCLK => PSCLK,   -- Dynamic phase adjust clock input
     PSEN => PSEN,     -- Dynamic phase adjust enable input
     PSINCDEC => PSINCDEC, -- Dynamic phase adjust increment/decrement
     RST => RST        -- DCM asynchronous reset input
  );
```

DCM

Table 3-8: **DFS Unit Clock Input Frequency Requirements (-4 Speed Grade)**

Function	Minimum Frequency	Maximum Frequency	Units
Data Sheet Specification	CLKIN_FREQ_FX_MIN	CLKIN_FREQ_FX_MAX	
Spartan-3 FPGA	1	280	MHz
Spartan-3E FPGA	0.200	333	MHz
Spartan-3A/3AN FPGA	0.200	333	MHz

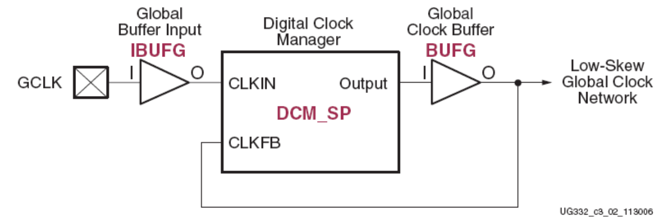
Table 3-9: **Spartan-3E/3A/3AN FPGAs: DLL Unit Clock Input Frequency Requirements**

Spartan-3 Generation FPGA Family	Speed Grade	Minimum	Maximum	Units
		CLKIN_FREQ_DLL_MIN	CLKIN_FREQ_DLL_MAX	
Spartan-3A/3AN FPGAs	-4	5	250	MHz
	-5		280	MHz
Spartan-3E (Stepping 1) FPGAs	-4		240	MHz
	-5		270	MHz

Table 3-10: **Spartan-3 FPGAs: DLL Unit Clock Input Frequency Requirements**

Spartan-3 Generation FPGA Family	DLL Frequency Mode Attribute (DLL_FREQUENCY_MODE)			
	= LOW		= HIGH	
	Minimum Frequency	Maximum Frequency	Minimum Frequency	Maximum Frequency
	CLKIN_FREQ_DLL_LF_MIN	CLKIN_FREQ_DLL_LF_MAX	CLKIN_FREQ_DLL_HF_MIN	CLKIN_FREQ_DLL_HF_MIN
Spartan-3 FPGAs	18 MHz	167 MHz	48 MHz	280 MHz

a. Global Buffer Inputs and Clock Buffers Drive a Low-Skew Global Network in the FPGA



b. A Digital Clock Manager (DCM) Inserts Directly into the Global Clock Path

Figure 3-2: **DCMs are an Integral Part of the FPGA's Global Clock Network**