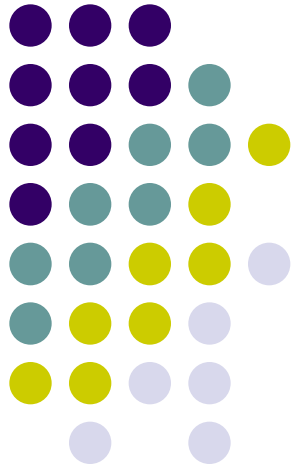
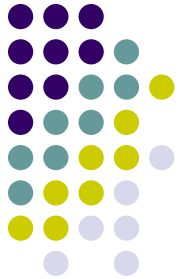
AHDL (Chapters 6 & 7)

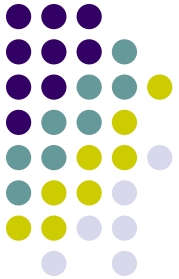


Using TTL Library Functions with HDL



- MSI chips such as adders and ALU ICs are the building blocks of digital systems.
- How to use these components in HDL? ➔ macrofunction
- A macrofunction is a self-contained description of a logic circuit with all its inputs, outputs and operational characteristics defined.
- Good documentation on the macrofunction is critical.

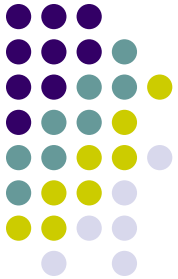
Function Prototype



- Example: 74382 (p. 343) 4-bit ALU with 8 different operations.

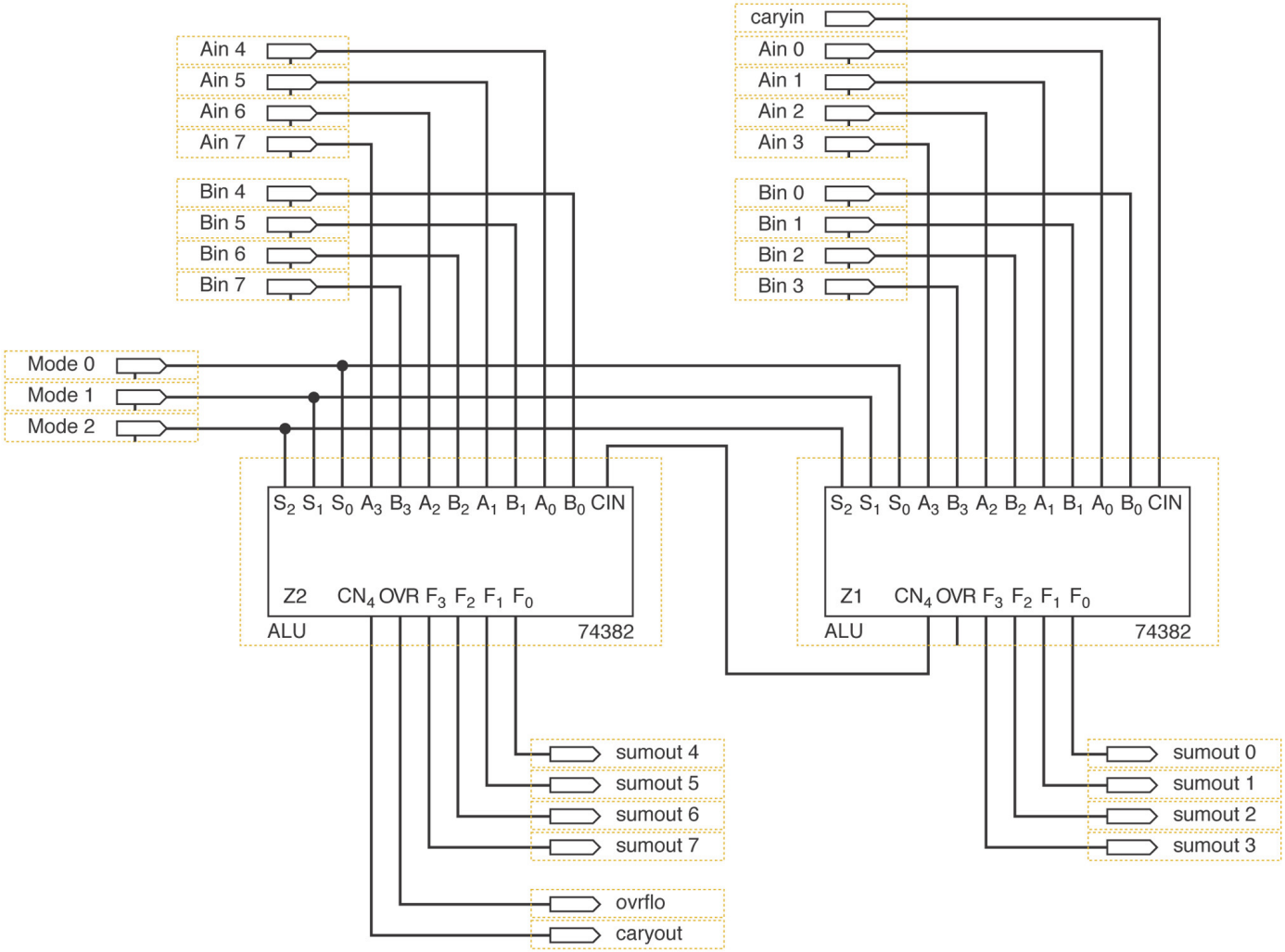
```
FUNCTION 74382(s[2..0], a[3..0],b[3..0],cin))
RETURNS (ovr, cn4, f[3..0])
```

AHDL Macrofunctions

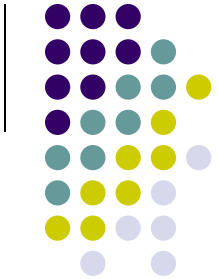


- Comments/documentation first
- Then Function prototype
- Then constant and global definitions
- INCLUDE derivative is allowed.
- Includes files with extension: .inc

Graphic Description of an 8-bit ALU

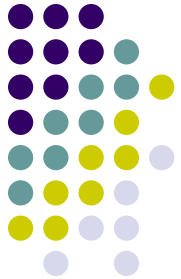


AHDL definition of an 8-bit ALU



```
1   %     8-bit ALU cascaded 74382 ALU in AHDL
2         Digital Systems 9th ed
3         MAY 22, 2002                %
4   FUNCTION 74382 (s[2..0], a[3..0], b[3..0], cin)
5   RETURNS (ovr, cn4, f[3..0]);
6
7   SUBDESIGN fig6_21
8   (
9         caryin            :INPUT;
10        mode[2..0]        :INPUT;              -- add/subtract controls
11        bin[7..0]         :INPUT;
12        ain[7..0]         :INPUT;
13        sumout[7..0]      :OUTPUT;
14        caryout           :OUTPUT;
15        ovrflo            :OUTPUT;
16  )
17  VARIABLE
18        z1    :74382;
19        z2    :74382;
20  BEGIN
21        z1.s[] = mode[];          -- both ALUs in same mode
22        z1.a[] = ain[3..0];       -- low nibble A
23        z1.b[] = bin[3..0];       -- low nibble B
24        z1.cin = caryin;
25        z2.s[] = mode[];          -- both ALUs in same mode
26        z2.a[] = ain[7..4];       -- high nibble A
27        z2.b[] = bin[7..4];       -- high nibble B
28        z2.cin = z1.cn4;          -- low Cout to high Cin
29
30        caryout = z2.cn4;         -- 9th bit out
31        sumout[3..0] = z1.f[];    -- low nibble result
32        sumout[7..4] = z2.f[];    -- high nibble result
33        ovrflo = z2.ovr;
34  END;
```

Logic Operation on Bit Arrays



```
SUBDESIGN bitwise_and
( d[3..0], g[3..0]   : INPUT;
  xx[3..0]           : OUTPUT;)
BEGIN
  xx[]= d[] & g[];
END; -- Note: x is a reserved identifier in
  AHDL, use xx
```

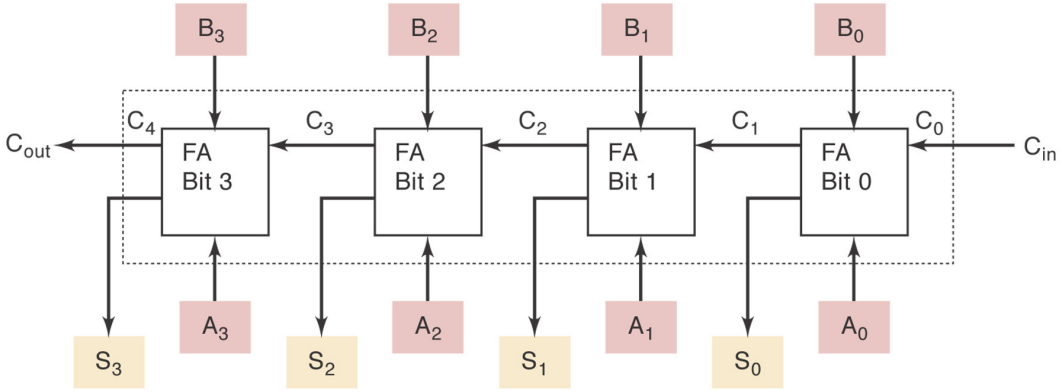
HDL Adders

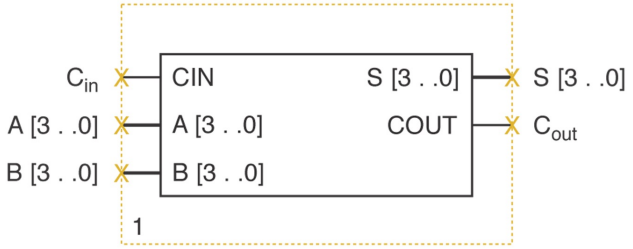
AUGEND	A_3	A_2	A_1	A_0	A
ADDEND	B_3	B_2	B_1	B_0	B
CARRYin	C_3	C_2	C_1	C_0	Cin
SUM	S_3	S_2	S_1	S_0	S

Generate sum
$S = A \oplus [B \oplus Cin]$

AUGEND	A_3	A_2	A_1	A_0	A
ADDEND	B_3	B_2	B_1	B_0	B
CARRYin	C_3	C_2	C_1	C_0	Cin
CARRYout	C_4	C_3	C_2	C_1	Cout

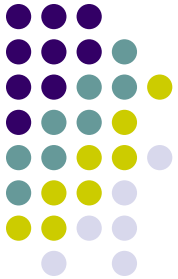
Generate carry bits
$Cout = A \cdot B + A \cdot Cin + B \cdot Cin$





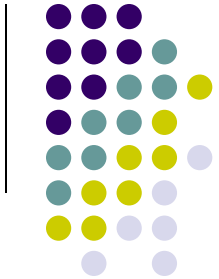
Block Symbol

AHDL 4-bit Adder



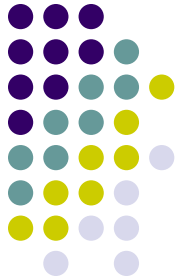
```
SUBDESIGN fig6_25
(
      cin          :INPUT;      -- carry in
      a[3..0]      :INPUT;      -- augend
      b[3..0]      :INPUT;      -- addend
      s[3..0]      :OUTPUT;     -- sum
      cout         :OUTPUT;     -- carry OUT
)
VARIABLE
      c[4..0]      :NODE;       -- carry array is 5 bits long!

BEGIN
      c[0] = cin;
      s[] = a[] $ b[] $ c[3..0];    -- generate sum
      c[4..1] = (a[] & b[]) # (a[] & c[3..0]) # (b[] & c[3..0]);
      cout = c[4];                  -- carry out
END;
```

N-bit Adder/Subtractor



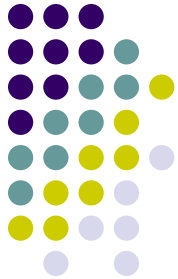
```
1   CONSTANT number_of_bits = 8;                  -- set total number of bits
2   CONSTANT n = number_of_bits - 1;              -- n is highest bit index
3
4   SUBDESIGN fig6_27
5   (
6      add          :INPUT;      -- add control
7      sub          :INPUT;      -- subtract control and LSB Carry in
8      a[n..0]      :INPUT;      -- Augend bits
9      bin[n..0]    :INPUT;      -- Addend bits
10     s[n..0]      :OUTPUT;     -- Sum bits
11     caryout      :OUTPUT;     -- MSB carry OUT
12  )
13  VARIABLE
14     c[n+1..0]   :NODE;       -- intermediate carry vector
15     b[n..0]      :NODE;       -- intermediate operand vector
16  BEGIN
17     b[] = bin[] & add # NOT bin[] & sub;
18     c[0] = sub;                      --Read the carry in to group variable
19     s[] = a[] $ b[] $ c[n..0];       --Generate the sums
20     c[n+1..1] = (a[] & b[]) # (a[] & c[n..0]) # (b[] & c[n..0]);
21     caryout = c[n+1];                -- output the carry of the MSB.
22  END;
```

Library of Parameterized Modules



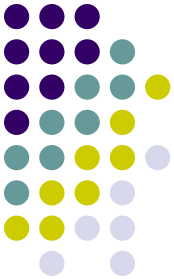
- Megafunctions: include a library of parameterized modules (LPMs)
- Offers a generic solution for the various types of logic circuits that are useful in digital systems.
- *Parameterized* means that when you instantiate a function from the library, you are specify some parameters that define certain attributes for the circuit.
- Example: LPM_ADD_SUB megafunction has a parameter LPM_WIDTH.

AHDL LPM Adder/Subtractor



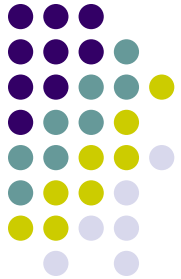
```
1   INCLUDE "LPM_ADD_SUB.INC";
2
3   SUBDESIGN fig6_31
4   (
5         a[7..0], b[7..0], caryin, functn   :INPUT;
6         s[7..0], caryout, ovr              :OUTPUT;
7   )
8   VARIABLE
9         eightbit   :LPM_ADD_SUB WITH (LPM_WIDTH = 8);
10
11  BEGIN
12        eightbit.dataa[]  = a[];
13        eightbit.datab[]  = b[];
14        eightbit.cin      = caryin;
15        eightbit.add_sub = functn;
16        s[] = eightbit.result[];
17        caryout = eightbit.cout;
18        ovr = eightbit.overflow;
19  END;
```

Basic Counters in HDL



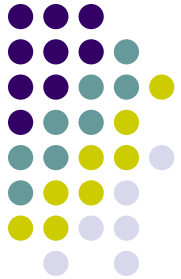
- Counters are constructed with FFs.
- In Chapter 5, we describe the FFs using AHDL.
- Becomes too tedious if we have to use multiple FFs.
- Describe circuits with a higher-level of abstraction.
- Will consider synchronous counters only.

Synchronous Counter Design with D FF



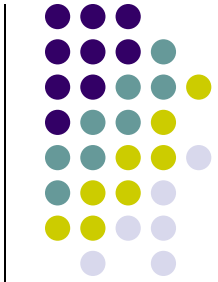
- Easier than using J-K FFs.
- The NEXT state of the D FF is the same as its PRESENT D input values.
- Example: Table 7-7

State Transition Description Methods



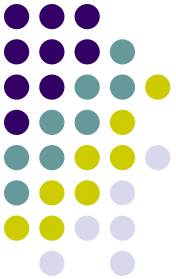
- List the PRESENT state/NEXT state table.

AHDL MOD-5 Counter



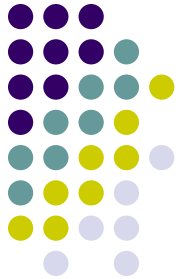
```
1   SUBDESIGN fig7_40
2   (
3         clock          :INPUT;
4         q[2..0]        :OUTPUT;
5   )
6   VARIABLE
7         count[2..0]    :DFF;          --create a 3-bit register
8   BEGIN
9         count[].clk = clock;          --connect all clocks in parallel
10
11              CASE count[] IS
12  --                  Present              Next
13  ------------------------------------------------------------
14                      WHEN  0      =>     count[].d = 1;
15                      WHEN  1      =>     count[].d = 2;
16                      WHEN  2      =>     count[].d = 3;
17                      WHEN  3      =>     count[].d = 4;
18                      WHEN  4      =>     count[].d = 0;
19                      WHEN OTHERS  =>     count[].d = 0;
20              END CASE;
21         q[] = count[];              -- assign register to output pins
22  END;
```

Another Version of MOD-5 Counter



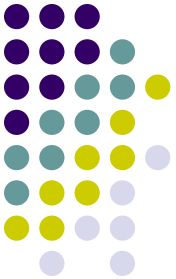
```
1   SUBDESIGN fig7_41
2   (
3       clock      :INPUT;
4       q[2..0]    :OUTPUT;
5   )
6   VARIABLE
7      q[2..0]     :DFF;    -- create a 3-bit register
8   BEGIN
9      q[].clk = clock;    -- connect all clocks in parallel
10     TABLE
11        q[].q =>    q[].d;
12        0     =>    1;
13        1     =>    2;
14        2     =>    3;
15        3     =>    4;
16        4     =>    0;
17        5     =>    0;
18        6     =>    0;
19        7     =>    0;
20     END TABLE;
21  END;
```

Behavioral Description



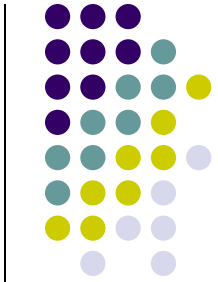
- The behavioral level of abstraction is a way to describe circuit by describing its behavior in terms very similar to the way you might describe its operation in English.
- Deals more with the cause-and-effect relationship than with the path of data flow or wiring details.

Behavioral Description of a Counter



```
1   SUBDESIGN fig7_44
2   (
3      clock     :INPUT;
4      q[2..0]   :OUTPUT;  -- declare 3-bit array of output bits
5   )
6   VARIABLE
7      count[2..0]  :DFF; -- declare a register of D flip flops.
8
9   BEGIN
10     count[].clk = clock;  -- connect all clocks to synchronous source
11     IF count[].q < 4 THEN    -- note; count[] is the same as count[].q
12        count[].d = count[].q + 1; -- increment current value by one
13     ELSE count[].d = 0;        -- recycle to zero: force unused states to 0
14     END IF;
15     q[] = count[];           -- transfer register contents to outputs
16  END;
```

AHDL Full-Featured Counter



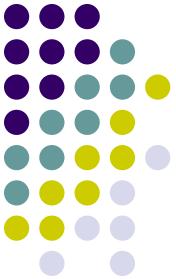
```
1  SUBDESIGN fig7_46
2  (
3     clock, clear, load, cntenabl, down, din[3..0]      :INPUT;
4     q[3..0], term_ct :OUTPUT;  -- declare 4-bit array of output bits
5  )
6  VARIABLE
7     count[3..0]    :DFF;       -- declare a register of D flip flops
8
9  BEGIN
10    count[].clk = clock;       -- connect all clocks to synch source
11    count[].clrn= !clear;      -- connect for asynch active HIGH clear
12    IF load THEN count[].d = din[]; -- synchronous load
13       ELSIF !cntenabl THEN count[].d = count[].q; -- hold count
14       ELSIF !down THEN count[].d = count[].q + 1; -- increment
15       ELSIF count[].d = count[].q - 1;            -- decrement
16    END IF;
17    IF ((count[] == 0) & down # (count[] == 15) & !down)& cntenabl
18    THEN     term_ct = VCC;    -- synchronous cascade output signal
19    ELSE term_ct = GND;
20    END IF;
21    q[] = count[];             -- transfer register contents to outputs
22 END;
```

LPM Counters

```
1   INCLUDE "1pm_counter.inc";
2
3   SUBDESIGN fig7_48
4   (
5      clk, load, enable, clear, updn     :INPUT;
6      d[6..0]]                           :INPUT;
7      q[6..0], tc                        :OUTPUT;
8   )
9   VARIABLE
10     counter          :LPM_COUNTER WITH
11                      (LPM_WIDTH=7, LPM_MODULUS="100");
12  BEGIN
13     counter.clock = clk;
14     counter.cnt_en = enable;
15     counter.sload = load;
16     counter.aclr = !clear;
17     counter.updown = updn;
18     counter.data[] = d[];
19     q[] = counter.q[];
20     tc = counter.cout & enable;
21  END;
```

State Machines

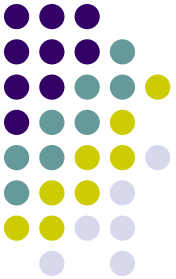
```
1   SUBDESIGN fig7_50
2   (  clock, start, full, timesup, dry     :INPUT;
3      water_valve, ag_mode, sp_mode        :OUTPUT;
4   )
5   VARIABLE
6   cycle:  MACHINE
7           WITH STATES (idle, fill, agitate, spin);
8   BEGIN
9   cycle.clk = clock;
10
11     CASE cycle IS
12        WHEN idle =>IF start THEN cycle = fill;
13                    ELSE     cycle = idle;
14                    END IF;
15        WHEN fill =>IF full THEN cycle = agitate;
16                    ELSE     cycle = fill;
17                    END IF;
18        WHEN agitate=> IF timesup THEN cycle = spin;
19                    ELSE     cycle = agitate;
20                    END IF;
21        WHEN spin => IF dry THEN cycle = idle;
22                    ELSE     cycle = spin;
23                    END IF;
24        WHEN OTHERS => cycle = idle;
25     END CASE;
26
27     TABLE
28        cycle     => water_valve,     ag_mode, sp_mode;
29        idle      => gnd,             gnd,     gnd;
30        fill      => vcc,             gnd,     gnd;
31        agitate   => gnd,             vcc,     gnd;
32        spin      => gnd,             gnd,     vcc;
33     END TABLE;
34  END;
```

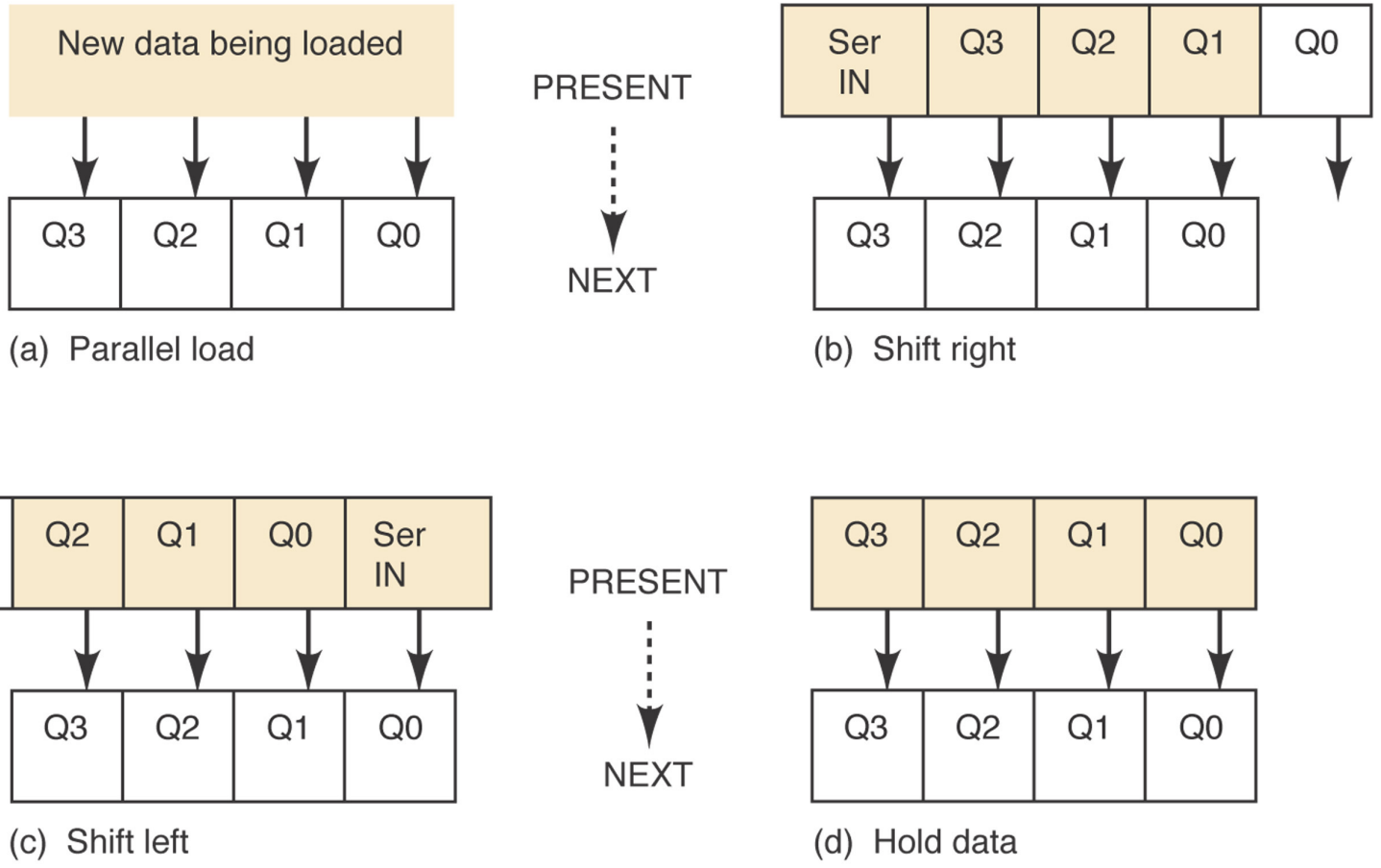


AHDL Registers

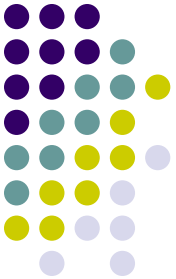
```
1   INCLUDE "lpm_shiftreg.inc";
2
3   SUBDESIGN fig7_66
4   (
5      clock          :INPUT;
6      din[3..0]      :INPUT;
7      ld/sh          :INPUT;
8      ser_out        :OUTPUT;
9   )
10  VARIABLE
11     shiftreg   :LPM_SHIFTREG WITH
12                    (LPM_WIDTH = 4);
13  BEGIN
14     shiftreg.clock      = clock;
15     shiftreg.data[]     = din[];
16     shiftreg.load       = ld/sh;
17     shiftreg.shiftin    = GND;
18     ser_out             = shiftreg.shiftout;
19  END;
```

Data Transfers on Shift Registers



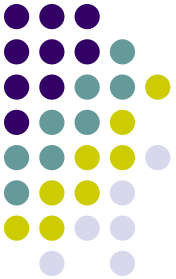


AHDL Bidirectional Shift Register



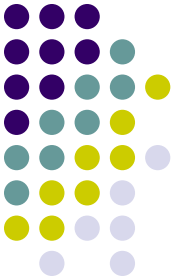
```
1   SUBDESIGN fig7_69
2   (
3      clock       :INPUT;
4      din[3..0]   :INPUT;  -- parallel data in
5      ser_in      :INPUT;  -- serial data in from Left or Right
6      model[1..0] :INPUT;  -- MODE Select: 0=hold, 1=right, 2=left, 3=load
7      q[3..0]     :OUTPUT;
8   )
9   VARIABLE
10     ff[3..0] :DFF;       -- define register set
11  BEGIN
12     ff[].clk = clock;    -- synchronous clock
13     CASE mode[] IS
14        WHEN 0  => ff[].d     = ff[].q;        -- hold shift
15        WHEN 1  => ff[2..0].d = ff[3..1].q);   -- shift right
16                   ff[3].d    = ser_in;        -- new data from left
17        WHEN 2  => ff[3..1].d = ff[2..0].q;    -- shift left
18                   ff[0].d    = ser_in;        -- new data bit from right
19        WHEN 3  => ff[].d     = din[];         -- parallel load
20     END CASE;
21     q[] = ff[];              -- update outputs
22  END;
```

AHDL 4-Bit Ring Counter



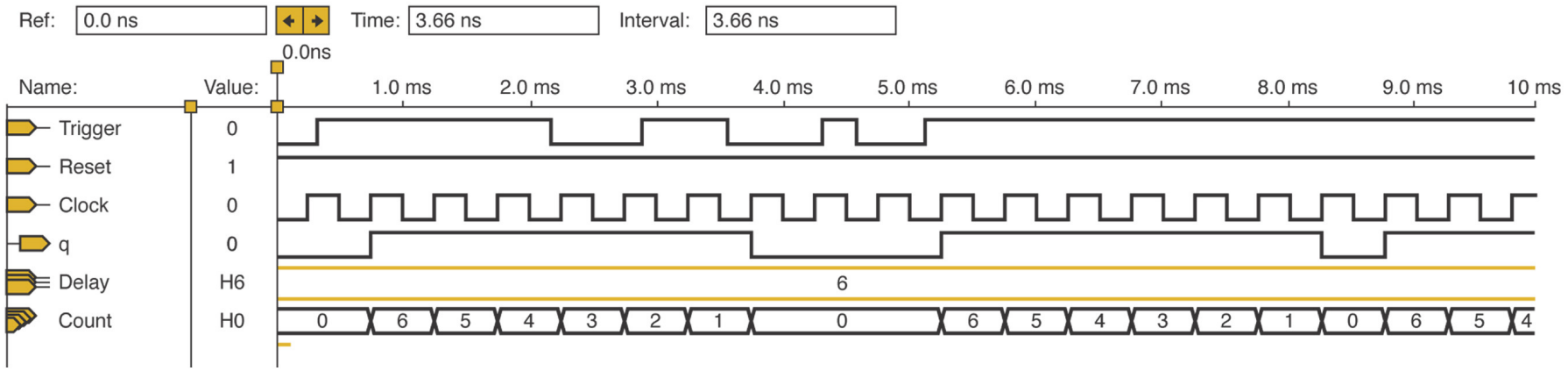
```
1   SUBDESIGN fig7_71
2   (
3      clk          :INPUT;
4      q[3..0]      :OUTPUT;
5   )
6   VARIABLE
7      ff[3..0]     :DFF;
8      ser_in       :NODE;
9   BEGIN
10    ff[].clk = clk;
11    IF ff[3..1] == B"000" THEN ser_in = VCC;   -- self start
12    ELSE ser_in = GND;
13    END IF;
14    ff[3..0].d = (ser_in, ff[3..1].q);         -- shift right
15    q[] = ff[];
16  END;
```

AHDL Nonretriggerable One-Shot

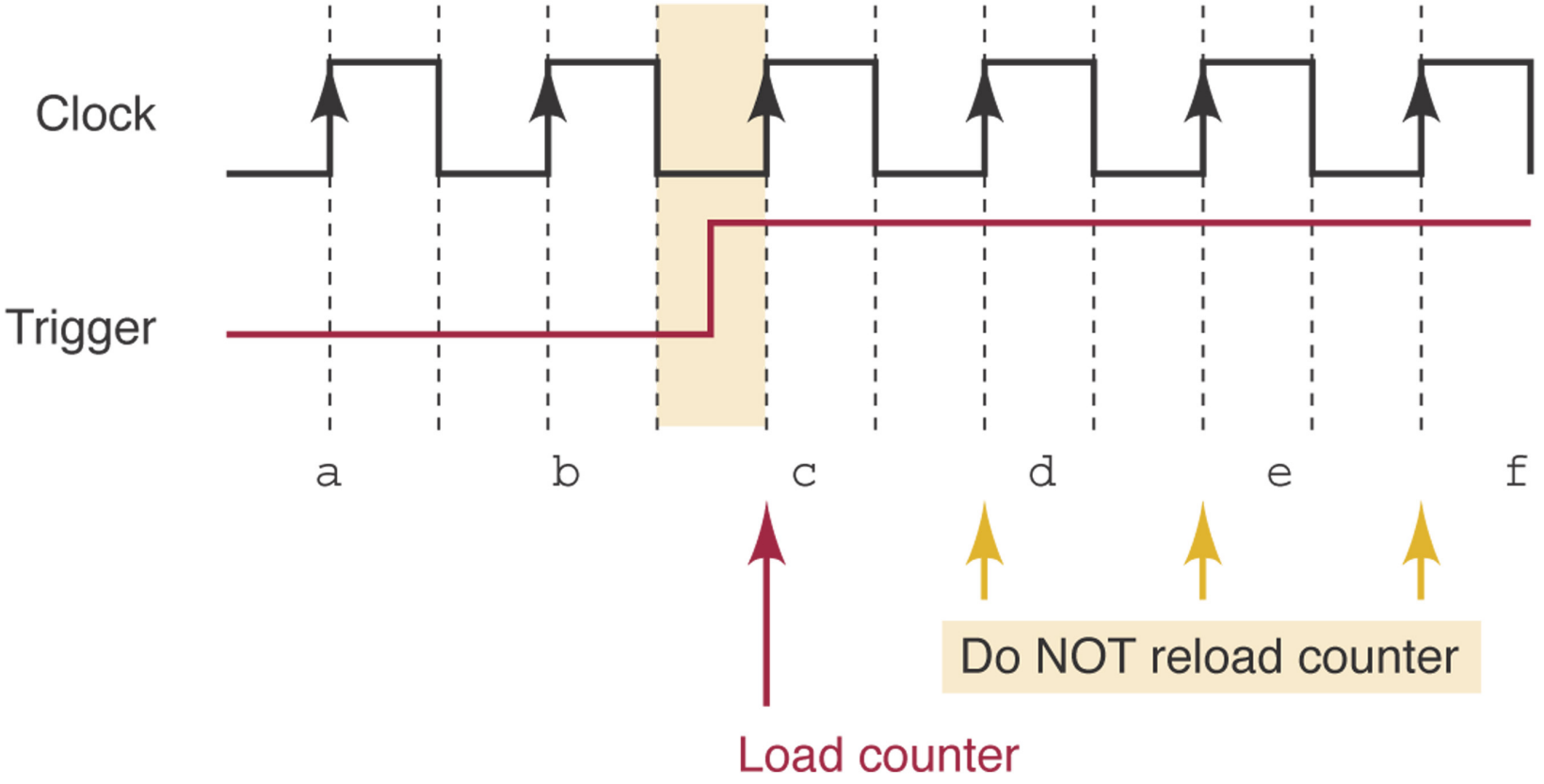


```
1   SUBDESIGN fig7_73
2   (
3      clock, trigger, reset    : INPUT;
4      delay[3..0]              : INPUT;
5      q                        : OUTPUT;
6   )
7   VARIABLE
8      count[3..0]     : DFF;
9   BEGIN
10     count[].clk = clock;
11     count[].clrn = reset;
12     IF trigger & count[].q == b"0000" THEN
13           count[].d = delay[];
14     ELSIF count[].q == B"0000" THEN count[].d = B"0000";
15     ELSE count[].d = count[].q - 1;
16     END IF;
17     q = count[].q != B"0000";  -- make output pulse
18  END;
```

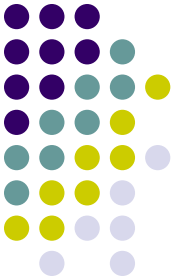
Simulation


Ref: 0.0 ns
Time: 3.66 ns
Interval: 3.66 ns
0.0ns
Name:
Value:
1.0 ms
2.0 ms
3.0 ms
4.0 ms
5.0 ms
6.0 ms
7.0 ms
8.0 ms
9.0 ms
10 ms
Trigger
0
Reset
1
Clock
0
q
0
Delay
H6
6
Count
H0
0 6 5 4 3 2 1 0 6 5 4 3 2 1 0 6 5 4

Detecting Edges



AHDL Retriggerable, Edge-Triggered One-Shot



```
1   SUBDESIGN fig7_77
2   (
3      clock, trigger, reset   : INPUT;
4      delay[3..0]             : INPUT;
5      q                       : OUTPUT;
6   )
7   VARIABLE
8                count[3..0]    : DFF;
9                trig_was       : DFF;
10  BEGIN
11     count[].clk = clock;
12     count[].clrn = reset;
13     trig_was.clk = clock;
14     trig_was.d = trigger;
15
16     IF trigger & !trig_was.q THEN
17           count[].d = delay[];
18     ELSIF count[].q == B"0000" THEN count[].d = B"0000";
19     ELSE count[].d = count[].q - 1;
20     END IF;
21     q = count[].q != B"0000";
22  END;
```