EEE205 – Digital Electronics (II)

Lecture 7

Dr. Ming Xu

Dept of Electrical & Electronic Engineering

XJTLU

In This Session

Sequential Circuits in AHDL

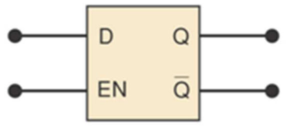
- The Latches and Flip-Flops
- Counters
- Shift Registers

Register Primitives

- Several types of register primitives are available in AHDL.
- Registers that use these primitives are declared in the VARIABLE section.

Primitive	Description
LATCH	Data (D) Latch
DFF/DFFE	Data (D) Flip-Flop
JKFF/JKFFE	JK Flip-Flop
SRFF/SRFFE	Set/Reset (SR) Flip-Flop
TFF/TFFE	Toggle (T) Flip-Flop

The D Latch



Inputs		Output
EN	D	Q
0	X	Q_0 (no change)
1	0	0
1	1	1

- Primitive: **LATCH**
- Inputs
  - Data Input (**d**)
  - Enable (**ena**)
- Outputs
  - Data Output (**q**)

The D Latch

- `dlatch` is an instance of LATCH.
- `enable`, `din` and `q` are the ports of the circuit block `dlatch_ahdl`.

```
SUBDESIGN dlatch_ahdl
(
     enable, din           :INPUT;
     q                     :OUTPUT;
)
VARIABLE
     dlatch                :LATCH;
BEGIN
     dlatch.ena = enable;
     dlatch.d = din;
     q = dlatch.q;
END;
```

The D Latch

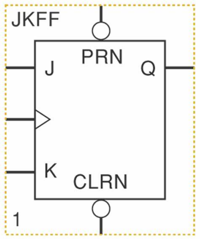
- If the instance name q is the same as the name of the circuit block output, it means the output of the latch is connected to the output port.

```
SUBDESIGN dlatch_ahdl
(
     enable, din           :INPUT;
     q                     :OUTPUT;
)
VARIABLE
     q                     :LATCH;
BEGIN
     q.ena = enable;
     q.d = din;
END;
```

The D Flip Flop

- Primitive: **DFF** or **DFFE**
- Inputs
  - Data Input (**d**)
  - Clock (**clk**)
  - Asynchronous Clear (**clrn**) and Set (**prn**), both active-LOW.
  - Enable (**ena**), **DFFE** only
- Outputs
  - Data Output (**q**)

The J-K Flip Flop



- Primitive: **JKFF** or **JKFFE**
- Inputs
  - Set (**j**) and Reset (**k**)
  - Clock (**clk**)
  - Asynchronous Clear (**clrn**) and Set (**prn**), both active-LOW.
  - Enable (**ena**), **JKFFE** only
- Outputs
  - Data Output (**q**)

The S-R Flip Flop

- Primitive: **SRFF** or **SRFFE**
- Inputs
  - Set (**s**) and Reset (**r**)
  - Clock (**clk**)
  - Asynchronous Clear (**clrn**) and Set (**prn**), both active-LOW.
  - Enable (**ena**), **SRFFE** only
- Outputs
  - Data Output (**q**)

The T Flip Flop

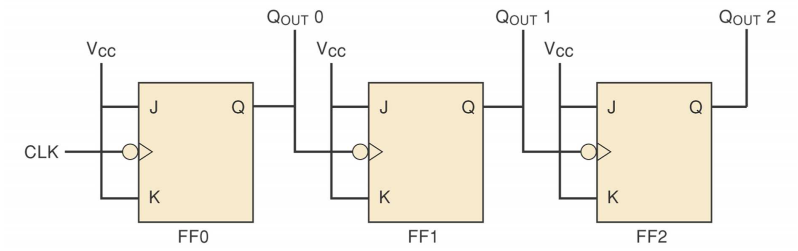
- Primitive: **TFF** or **TFFE**
- Inputs
  - Toggle (**t**)
  - Clock (**clk**)
  - Asynchronous Clear (**clrn**) and Set (**prn**), both active-LOW.
  - Enable (**ena**), **TFFE** only
- Outputs
  - Data Output (**q**)

An Example

```
%     JK flip-flop circuit     %
SUBDESIGN fig5_65
(
      jin, kin, clkin, preset, clear    :INPUT;
      qout                              :OUTPUT;
)
VARIABLE
ff1         :JKFF;      -- define this flip-flop as a JKFF type
BEGIN
      ff1.prn = preset; -- these are optional and default to vcc
      ff1.clrn = clear;
      ff1.j = jin;      -- connect primitive to the input signal
      ff1.k = kin;
      ff1.clk = clkin;
      qout = ff1.q;     -- connect the output pin to the primitive
END;
```

Asynchronous Counters

MOD-8 Counter



- The name of this register is q, like the name of the output port.
- So the output of each flip-flop is connected to the output port.

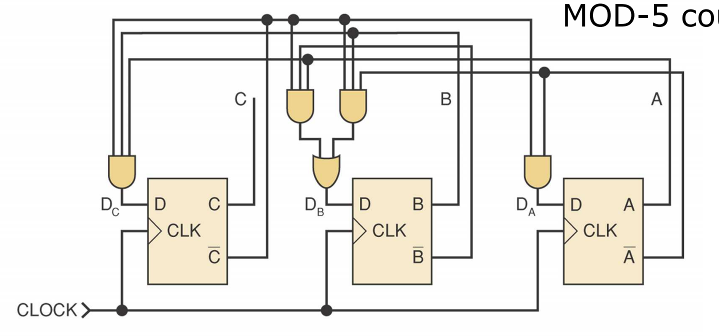
Asynchronous Counters

```
%  MOD 8 ripple up counter. %
SUBDESIGN fig5_71
(
      clock                   :INPUT;
      q[2..0]                 :OUTPUT;
)
VARIABLE
      q[2..0] :JKFF;           -- defines three JK FFs
BEGIN
                              -- note: prn, clrn default to vcc!
      q[2..0].j = VCC;        -- toggle mode J=K=1 for all FFs
      q[2..0].k = VCC;
      q[0].clk = !clock;
      q[1].clk = !q[0].q;
      q[2].clk = !q[1].q;     -- connect clocks in ripple form
END;
```

Synchronous Counters

- When using AHDL to implement a synchronous counter, we are no longer focusing on *wiring issues* but rather on describing the *circuit operation* concisely.

MOD-5 counter



Synchronous Counters

State Description – An MOD-5 Counter

```
SUBDESIGN fig7_35
(
      clock        :INPUT;
      q[2..0]      :OUTPUT;
)
VARIABLE
      count[2..0]  :DFF;       --create a 3-bit register
BEGIN
      count[].clk = clock;     --connect all clocks in parallel

            CASE count[] IS
--                 Present          Next
--------------------------------------------------------
                   WHEN  0     =>    count[].d = 1;
                   WHEN  1     =>    count[].d = 2;
                   WHEN  2     =>    count[].d = 3;
                   WHEN  3     =>    count[].d = 4;
                   WHEN  4     =>    count[].d = 0;
                   WHEN OTHERS =>    count[].d = 0;
             END CASE;
       q[] = count[];          -- assign register to output pins
END;
```

Synchronous Counters

Behavioral Description – An MOD-5 Counter

```
SUBDESIGN fig7_39
(
   clock    :INPUT;
   q[2..0]  :OUTPUT;  -- declare 3-bit array of output bits
)
VARIABLE
   count[2..0]  :DFF; -- declare a register of D flip flops.

BEGIN
   count[].clk = clock;  -- connect all clocks to synchronous source
   IF count[].q < 4 THEN    -- note; count[] is the same as count[].q
      count[].d = count[].q + 1; -- increment current value by one
   ELSE count[].d = 0;         -- recycle to zero: force unused states to 0
   END IF;
   q[] = count[];             -- transfer register contents to outputs
END;
```

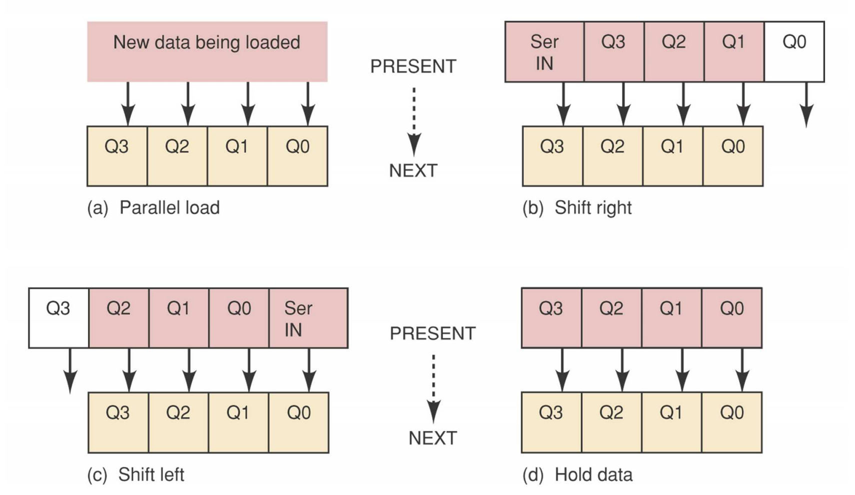
Synchronous Counters

Behavioral Description – A Full Feature Counter

```
SUBDESIGN fig7_42
(
   clock, clear, load, cntenabl, down, din[3..0]      :INPUT;
   q[3..0], term_ct :OUTPUT;  -- declare 4-bit array of output bits
)
VARIABLE
   count[3..0]    :DFF;       -- declare a register of D flip flops

BEGIN
   count[].clk = clock;       -- connect all clocks to synch source
   count[].clrn= !clear;      -- connect for asynch active HIGH clear
   IF load THEN count[].d = din[]; -- synchronous load
      ELSIF !cntenabl THEN count[].d = count[].q; -- hold count
      ELSIF !down THEN count[].d = count[].q + 1; -- increment
      ELSE count[].d = count[].q - 1;            -- decrement
   END IF;
   IF ((count[].q == 0) & down # (count[].q == 15) & !down)& cntenabl
   THEN     term_ct = VCC;    -- synchronous cascade output signal
   ELSE term_ct = GND;
   END IF;
   q[] = count[];             -- transfer register contents to outputs
END;
```

Shift Registers



Shift Registers

Serial In/ Serial Out

```
SUBDESIGN  fig7_77
(
      clk, shift, serial_in          :INPUT;
      serial_out                     :OUTPUT;
)
VARIABLE
      q[3..0]                        :DFF;
BEGIN
      q[].clk = clk;
      serial_out = q[0].q;                          -- output last register bit
      IF (shift == VCC)  THEN
            q[3..0].d = (serial_in, q[3..1].q);  -- concatenates for shift
      ELSE
            q[3..0].d = (q[3..0].q);                -- hold data
      END IF;
END;
```

Shift Registers

Universal Shift Register

```
SUBDESIGN fig7_83
(
   clock       :INPUT;
   din[3..0]   :INPUT;  -- parallel data in
   ser_in      :INPUT;  -- serial data in from Left or Right
   mode [1..0] :INPUT;  -- MODE Select: 0=hold, 1=right, 2=left, 3=load
   q[3..0]     :OUTPUT;
)
VARIABLE
   ff[3..0] :DFF;       -- define register set
BEGIN
   ff[].clk = clock;    -- synchronous clock
   CASE mode[] IS
      WHEN 0  => ff[].d     = ff[].q;         -- hold shift
      WHEN 1  => ff[2..0].d = ff[3..1].q);    -- shift right
                 ff[3].d    = ser_in;         -- new data from left
      WHEN 2  => ff[3..1].d = ff[2..0].q;     -- shift left
                 ff[0].d    = ser_in;         -- new data bit from right
      WHEN 3  => ff[].d     = din[];          -- parallel load
   END CASE;
   q[] = ff[];                 -- update outputs
END;
```