Imperial College London

MSc in Analogue & Digital IC Design

Clocked circuits, Shift Registers & BCD converter

Peter Cheung
Department of Electrical & Electronic Engineering
Imperial College London

Course webpage: www.ee.ic.ac.uk/pcheung/teaching/MSc_Experiment/
E-mail: p.cheung@imperial.ac.uk

I hope you have completed Part 1 of the Experiment. This lecture leads you to Part 2 of the experiment and hopefully helps you with your progress to Part 2. It covers a number of topics:

1. How do we specify clocked (i.e. sequential) circuits in Verilog?
2. How do we specify a flexible counter?
3. How to specify and use shift registers?
4. How to specify linear-feedback shift registers as a pseudo random binary sequence generator?
5. How to convert binary (or hexadecimal) numbers to binary coded decimal (BCD) numbers?

Power of Verilog: Integer Arithmetic

- Arithmetic operations make computation easy:

```
module add32(a, b, sum);
  input[31:0] a,b;
  output[31:0] sum;
  assign sum = a + b;
endmodule
```

- Here is a 32-bit adder with carry-in and carry-out:

```
module add32_carry(a, b, cin, sum, cout);
  input[31:0] a,b;
  input cin;
  output[31:0] sum;
  output cout;
  assign {cout, sum} = a + b + cin;
endmodule
```

PYKC 19 Oct 2017 MSc Lab – Mastering Digital Design Lecture 2 Slide 2

Verilog is very much like C. However, the declaration of **a, b** and **sum** in the **module add32** specifies the data width (i.e. number of bits in each signal **a**, **b** or **sum**). This is often known as a "**vector**" or a "**bus**". Here the data width is 32-bit, and it is ranging from bit 31 down to bit 0 (e.g. **sum[31:0]**).
You can refer to individual bits using the index value. For example, the least-significant bit (LSB) of sum is **sum[0]** and the most-significant bit (MSB) is **sum[31].** **sum[7:0]** refers the the least-significant byte of **sum**.
The '+' operator can be used for signals of any width. Here a 32-bit add operation is specified. **sum** is also 32-bit in width. However, if **a** and **b** are 32-bit wide, the sum result could be 33-bit (including the carry out). Therefore this operation could result in a wrong answer due to **overflow** into the carry bit. The 33th bit is truncated.
The second example **module add32_carry** shows the same adder but with carry input and carry output. Note the LHS of the **assign** statement. The **{cout, sum}** is a **concatenation** operator – the contents inside the brackets **{ }** are concatenated together, with **cout** is assigned the MSB of the 33th bit of the result , and the remaining bits are formed by **sum[31:0]**.

Different types of Boolean Operators

- **Bitwise operators**: perform bit-sliced operations on vectors bit by bit
  - **~(4'b0101) = {~0,~1,~0,~1} = 4'b1010**
  - **4'b0101 & 4'b0011 = 4'b0001**
- **Logical operators**: return true or false (1-bit) results
  - **!(4'b0101) = ~1 = 1'b0**
- **Reduction operators**: act on each bit of a SINGLE input vector
  - **&(4'b0101) = 0 & 1 & 0 & 1 = 1'b0**

Bitwise

~a	NOT
a & b	AND
a | b	OR
a ^ b	XOR
a ~^ b	XNOR

Logical

!a	NOT
a && b	AND
a || b	OR

Note distinction between ~a and !a

Reduction

&a	AND
~&	NAND
|	OR
~|	NOR
^	XOR

PYKC 19 Oct 2017 MSc Lab – Mastering Digital Design Lecture 2 Slide 3

There are three different types of Boolean operators:

Bitwise operators perform what you would expect as if there are parallel gates used for each bit of the operands. Therefore **a&b** means that each bit from **a** and **b** is passed through an AND-gate.

Logical operators only result in 0 or 1 (i.e. 1-bit result) In this example !a **(not a)** where **a = 0101**, will result in first, **a** being evaluated as a logical value (i.e. logical '1' or true). Therefore the result **~a** is logical 0 (or false).

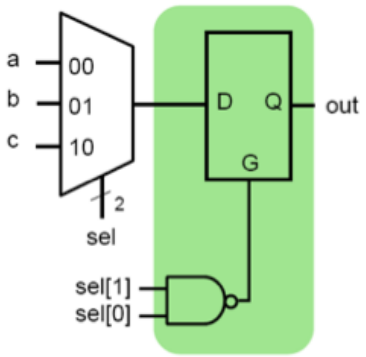
Reduction operators is applied to a single operand (and sometimes known as unary operators). It performs the operation one-bit at a time to the operand.

Incomplete specification: adds unwanted latch circuit

if `out` is not assigned during any pass through the always block, then the previous value must be retained!

```
module maybe_mux_3to1(a, b, c,
                      sel, out);
  input [1:0] sel;
  input a,b,c;
  output out;
  reg out;

  always @(a or b or c or sel)
  begin
    case (sel)
      2'b00: out = a;
      2'b01: out = b;
      2'b10: out = c;
    endcase
  end
endmodule
```

Synthesized Circuit



- When sel = 2'b11, G = 0, therefore the latch stores the previous output value as required by Verilog in this situation.

PYKC 19 Oct 2017 MSc Lab – Mastering Digital Design Lecture 2 Slide 4

The consequence of this is an unexpected extra latch being added to the hardware. In order to cope with the unspecified condition of **sel = 2'b11**, the output of the MUX is fed to be **latch**.

Noted that a latch is **level-triggered**; a flipflop is **edge-triggered**. A latch has the property that when the gate input **G** is high**, Q = D** (i.e. it is **transparent**: input goes straight to output). If **G** is low, the latch become **opaque**, meaning that it retains the previous value.

The green shaded latch in the diagram and the controlling NAND gate are the unintended consequences of this incompletely specified 3-to-1 multiplexer.

Always avoid incomplete specification

- Solution 1: Precede all conditionals with a default assignment for all signals:

```
always @(a or b or c or sel)
  begin
    out = 1'bx;
    case (sel)
      2'b00: out = a;
      2'b01: out = b;
      2'b10: out = c;
    endcase
  end
endmodule
```

- Solution 2: Fully specify all branches of if-else construct, or include a default statement in case construct:

```
always @(a or b or c or sel)
 begin
 case (sel)
     2'b00: out = a;
     2'b01: out = b;
     2'b10: out = c;
     default: out = 1'bx;
   endcase
 end
endmodule
```

PYKC 19 Oct 2017 MSc Lab – Mastering Digital Design Lecture 2 Slide 5

There are two solutions to avoid the unintended latch being added.
Solution 1 is to put outside the **case** statement a "**default**" value for out. Here **1'bx** (i.e. 'x') means **undefined**.
Solution 2 is better: inside the **case** statement block, always add the **default** line. This will catch ALL the unspecified cases and avoid the introduction of the spurious unintended latches.

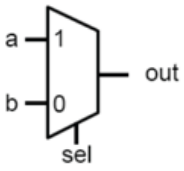
Lesson: always include a default assignment in any **case** statement to capture unintended incomplete specification.

How to specify a sequential circuit?

- Edge-triggered flipflop is specified with: **always @ (posedge clk):**

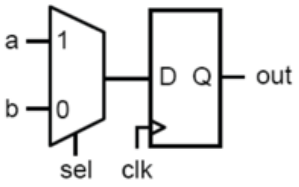
Combinational cct

```
module combinational(a, b, sel,
                     out);
  input a, b;
  input sel;
  output out;
  reg out;
  always @ (a or b or sel)
  begin
    if (sel) out = a;
    else out = b;
  end
endmodule
```



Sequential cct

```
module sequential(a, b, sel,
                  clk, out);
  input a, b;
  input sel, clk;
  output out;
  reg out;
  always @ (posedge clk)
  begin
    if (sel) out <= a;
    else out <= b;
  end
endmodule
```



PYKC 19 Oct 2017 MSc Lab – Mastering Digital Design Lecture 2 Slide 6

We have previously seen the 2-to-1 MUX being specified as combinational circuit in Verilog using the **always** construct with the **sensitivity list**.
The right hand diagram shows how a clocked **sequential circuit** is being specified using **always** block, but with a **sensitivity list** that includes the keyword **posedge** (or **negedge**). Note that the clocking signal **clk** is an arbitrary name – you could call it "**fred**" or anything else!
The **sensitivity list** NO LONGER contains the input signals **a**, **b** or **sel**. Instead the hardware is specified to be sensitive the positive edge of **clk**. When this happens, the output changes according to the specification inside the **always** block.
Two assignments ("**=**" and "**<=**") are shown here. I will explain the difference between these later.

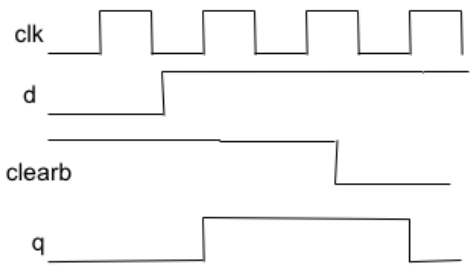
Synchronous clear in D-flipflop

- **posedge** and **negedge** makes an always block sequential and edge-triggered
- Sensitivity list in a sequential **always** block determines what circuit is synthesized

D flipflop with **synchronous** clear

```
module dff_sync_clear(d, clearb,
clock, q);
input d, clearb, clock;
output q;
reg q;
always @ (posedge clock)
begin
  if (!clearb) q <= 1'b0;
  else q <= d;
end
endmodule
```

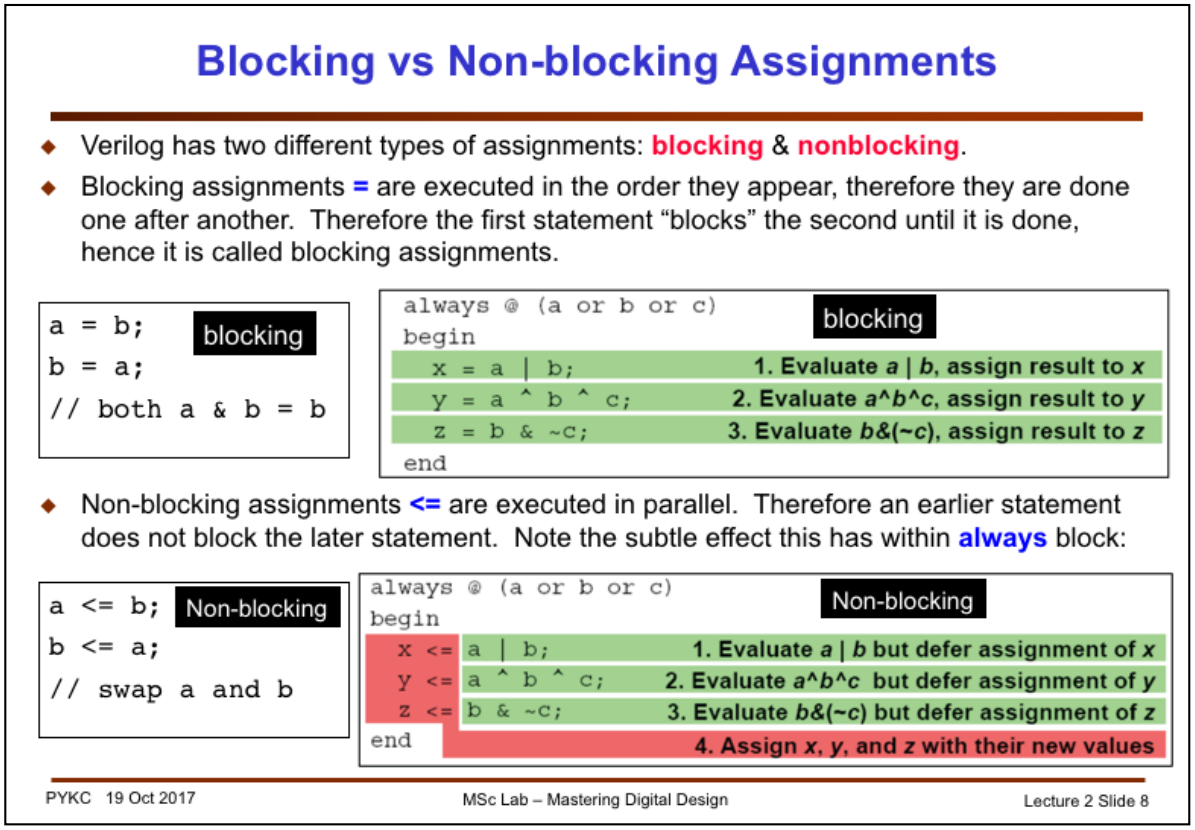
+ve edge on clock triggers action in always block



- Beware of **race condition** if you have two or more always blocks – they execute in parallel!

PYKC 19 Oct 2017 MSc Lab – Mastering Digital Design Lecture 2 Slide 7

Therefore in Verilog, you specify flipflops using **always block** in conjunction with the keyword **posedge** or **negedge**.
Here is a specification for a D-flipflop with synchronous clear which is low active (i.e. clear the FF when **clearb** is low).
You may have more than one **always** block in a module. But if this is the case, beware that the two **always** blocks will **execute in parallel**. Therefore they must NOT specify the same output, otherwise a **race condition** exists and the result is unpredictable.

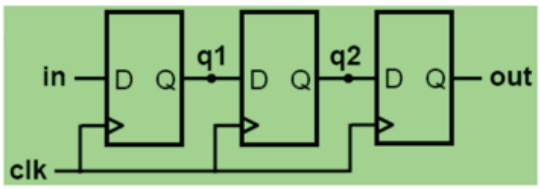


In Verilog '**=**' is known as **blocking assignment**. They are **executed in the order** they appear within the Verilog simulation environment. So the first '=' assignment blocks the second one. This is very much like what happens in C codes.
In the top left example, both **a** and **b** eventually have the value **b**.
In the top right example, each statement is evaluated in turn and assignment is performed immediately at the end of the statement.

Non-block assignment is '**<=**', and statements with this assignments are **executed in parallel** (i.e. order do not matter).
In the bottom left example, **a** and **b** are **swapped** over because you can view that the two assignments happen at the same time.
In the bottom right example, three evaluations are made, and the assignment to x, y and z happens at the same time on exiting from the **always** block.

Be careful to use the correct assignment

- Here are two versions of a 3-stage shift register consisting of 3 flipflops using blocking and nonblocking assignments.
- Will they give the same results?




```
module blocking(in, clk, out);
  input in, clk;
  output out;
  reg q1, q2, out;

  always @ (posedge clk)
  begin
    q1 = in;
    q2 = q1;
    out = q2;
  end

endmodule
```

```
module nonblocking(in, clk, out);
  input in, clk;
  output out;
  reg q1, q2, out;

  always @ (posedge clk)
  begin
    q1 <= in;
    q2 <= q1;
    out <= q2;
  end

endmodule
```

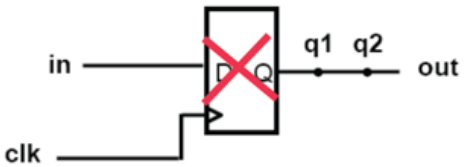
PYKC 19 Oct 2017 MSc Lab – Mastering Digital Design Lecture 2 Slide 9

Understanding the difference between '=' and '<=' is important. Suppose we want to specify a three-stage shift register (i.e. three D-FF in series as shown in the schematic).
Here are two possible specification. Which one do you think will create the correct circuit and which one is wrong?

Use NONBLOCKING assignment for sequential logic

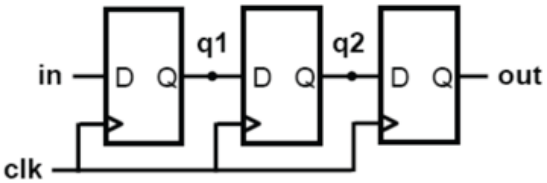
```
always @ (posedge clk)
begin
   q1 = in;
   q2 = q1;
   out = q2;
end
```

- At each rising clock edge:
  q1 = in,
  then q2 = q1 = in,
  then, out = q2 = q1 = in.
- Therefore out = in, which is NOT the intention.




```
always @ (posedge clk)
begin
   q1 <= in;
   q2 <= q1;
   out <= q2;
end
```

- At each rising clock edge, *q1*, *q2* and *out* simultaneously receive the old values of *in*, *q1*, *q2* respectively.



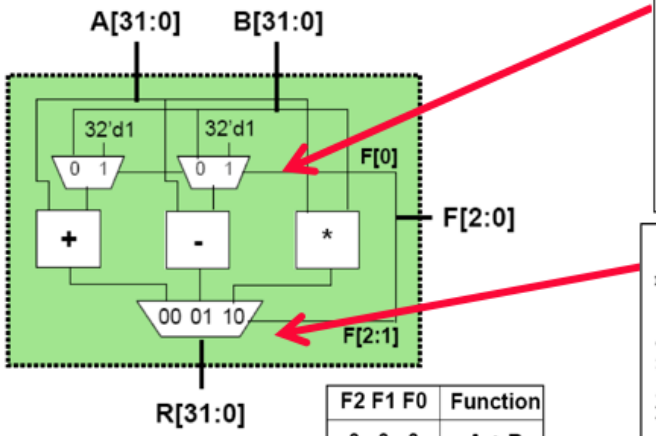
PYKC 19 Oct 2017 MSc Lab – Mastering Digital Design Lecture 2 Slide 10

The left hand specification is **wrong**. Since the three assignments are performed in sequence, **out = q2 = q1 = in**. Therefore the resultant circuit is ONE D-flipflop.
The right hand side is **correct**. **q1**, **q2** and **out** are updated simultaneously on exit from the **always** block. Therefore their "original" values MUST be retained. Hence this will result in three D-flipflops being synthesised (i.e. created).

In general, you should always use '**<=**' inside an **always** block to specify your circuit.

A larger example – 32-bit ALU in Verilog

- Here is an 32-bit ALU with 5 simple instructions:



F2 F1 F0	Function
0 0 0	A + B
0 0 1	A + 1
0 1 0	A - B
0 1 1	A - 1
1 0 X	A * B

2-to-1 MUX

```
module mux32two(i0,i1,sel,out);
input [31:0] i0,i1;
input sel;
output [31:0] out;

assign out = sel ? i1 : i0;

endmodule
```

3-to-1 MUX

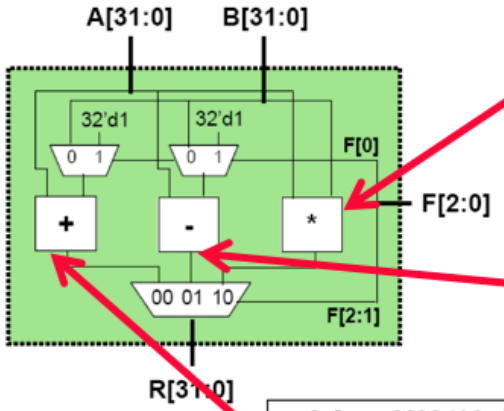
```
module mux32three(i0,i1,i2,sel,out);
input [31:0] i0,i1,i2;
input [1:0] sel;
output [31:0] out;
reg [31:0] out;

always @ (i0 or i1 or i2 or sel)
begin
  case (sel)
    2'b00: out = i0;
    2'b01: out = i1;
    2'b10: out = i2;
    default: out = 32'bx;
  endcase
end
endmodule
```

PYKC 19 Oct 2017 MSc Lab – Mastering Digital Design Lecture 2 Slide 11

Now let us put all you have learned together in specifying (or designing) a 32-bit ALU in Verilog.
There are five operators in this ALU. We assume that there are three arithmetic blocks, and three multiplexers (two 2-to-1 MUX and one 3-to-1 MUX).

The arithmetic modules

- Here is an 32-bit ALU with 5 simple instructions:




```
module mul16(i0,i1,prod);
input [15:0] i0,i1;
output [31:0] prod;

// this is a magnitude multiplier
// signed arithmetic later
assign prod = i0 * i1;

endmodule
```

```
module sub32(i0,i1,diff);
input [31:0] i0,i1;
output [31:0] diff;

assign diff = i0 - i1;

endmodule
```

```
module add32(i0,i1,sum);
input [31:0] i0,i1;
output [31:0] sum;

assign sum = i0 + i1;

endmodule
```

PYKC 19 Oct 2017 MSc Lab – Mastering Digital Design Lecture 2 Slide 12

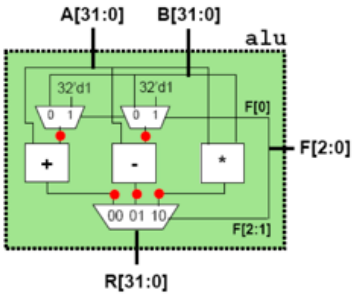
Each hardware block is defined as a Verilog module. So we have the following modules:

mux32two – a 32-bit multiplexer that has TWO inputs
mux32three – a 32-bit multiplexer that has THREE inputs
mul16 – a 16-by-16 binary multiplier that produces a 32-bit product
add32 – a 32-bit binary adder
sub32 – a 32-bit binary subtractor

Top-level module – putting them together

- Given submodules:

```
module mux32two(i0,i1,sel,out);
module mux32three(i0,i1,i2,sel,out);
module add32(i0,i1,sum);
module sub32(i0,i1,diff);
module mul16(i0,i1,prod);
```




```
module alu(a, b, f, r);
  input [31:0] a, b;
  input [2:0] f;
  output [31:0] r;

  wire [31:0] addmux_out, submux_out;
  wire [31:0] add_out, sub_out, mul_out;

  mux32two   adder_mux(b, 32'd1, f[0], addmux_out);
  mux32two   sub_mux(b, 32'd1, f[0], submux_out);
  add32      our_adder(a, addmux_out, add_out);
  sub32      our_subtracter(a, submux_out, sub_out);
  mul16      our_multiplier(a[15:0], b[15:0], mul_out);
  mux32three output_mux(add_out, sub_out, mul_out, f[2:1], r);

endmodule
```

intermediate output nodes

module names

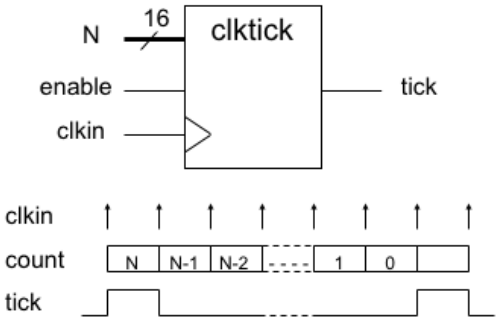
(unique) instance names

corresponding wires/regs in module alu

Now let us put all these together.
Note that **mxu32two** is being used twice and therefore this **is instantiated** two times with two different **instance names**: **adder_mux** and **sub_mux**.
Connections between modules are implicit through the use of **signal names**. For example, the 16-bit inputs to the multiplier are taken from the lower 16-bits of **a** and **b** inputs (i.e. **a[15:0]** and **b[15:0]**).

A Flexible Timer – clktick.v

- Instead of having a counter that count events, we often want a counter to provide a measure of **time**. We call this a timer.
- Here is a useful **timer** component that use a clock reference, and produces a pulse lasting for one cycle pulse every N+1 clock cycles.
- If "enable" is low (not enabled), the clkin pulses will be ignored.



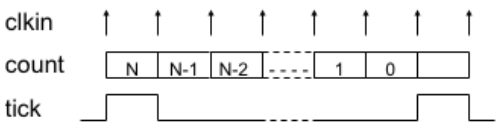

```
module clktick (
  clkin,     // Clock input to the design
  enable,    // enable clk divider
  N,         // Clock division factor is N+1
  tick       // pulse_out goes high for one cycle (n+1) clock cycles
);           //    End of port list

parameter   N_BIT = 16;
//-------------Input Ports-----------------------------
input clkin;
input enable;
input [N_BIT-1:0] N;

//-------------Output Ports----------------------------
output   tick;
```

PYKC 19 Oct 2017 MSc Lab – Mastering Digital Design Lecture 2 Slide 14

Counters are good in counting events (e.g. clock cycles). We can also use counters to provide some form of time measurement.
Here is a useful component which I can a clock tick circuit. We are not interested in the actual count value. What is needed, however, is that the circuit generates a single clock pulse (i.e. lasting for one clock period) for every N+1 rising edge of the clock input signal **clkin**.
We also add an enable signal, which must be set to '1' in order to enable the internal counting circuit.
Shown below is the module interface for this circuit in Verilog.
Note that the **parameter** keyword is used to define the number of bits of the internal counter (or the count value N). This makes the module easily adaptable to different size of counter.

clktick.v explained

- "count" is an internal N_BIT counter.
- We use this as a down (instead of up) counter.
- The counter value goes from N to 0, hence there are N+1 clock cycles for each tick pulse.



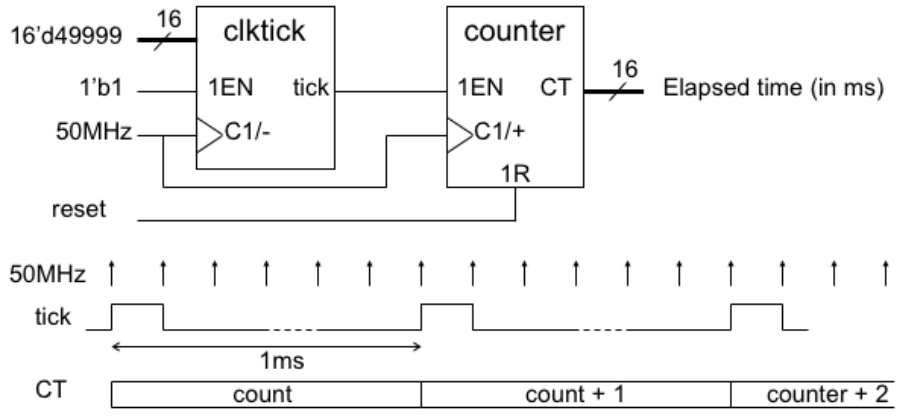

```
//-------------Output Ports Data Type------------------
// Output port can be a storage element (reg) or a wire
reg [N_BIT-1:0] count;
reg         tick;

initial     tick = 1'b0;

//------------ Main Body of the module ------------------

   always @ (posedge clkin)
      if (enable == 1'b1)
         if (count == 0)   begin
            tick <= 1'b1;
            count <= N;
            end
         else  begin
            tick <= 1'b0;
            count <= count - 1'b1;
            end

endmodule // End of Module clktick
```

PYKC 19 Oct 2017 MSc Lab – Mastering Digital Design Lecture 2 Slide 15

The actual Verilog specification for this module is shown here.
There has to be an internal counter **count** whose output is NOT visible external to this module. This is created with the **reg [N_BIT-1:0] count;** statement.
The output **tick** has to be declared as **reg** here because its value is updated inside the always block.
Also note that instead of adding '1' on each positive edge of the clock, this design uses a **down** counter. The counter counts from N to 0 (hence N+1 clock cycles). When that happens, it is reset to N and the tick output is high for the next clock cycle.

Cascading counters

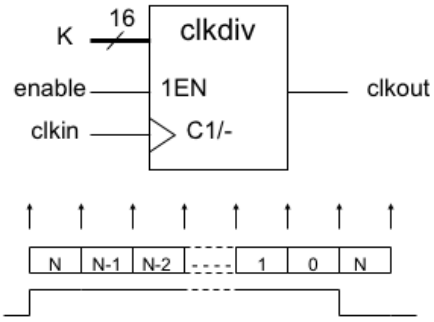
- By connecting **clktick** module in series with a counter module, we can produce a counter that counts the number of millisecond elapsed as shown below.



Using this style of designing a clock tick circuit allows us to easily connect multiple counters in series as shown here.
The **clktick** module is producing a pulse on the **tick** output every 50,000 cycles of the 50MHz clock. Therefore **tick** goes high for 20 microsecond once every 1 msec (or 1KHz).
The **clktick** module is sometimes called a **prescaler** circuit. It prescale the input clock signal (50MHz) in order for the second counter to count at a lower frequency (i.e. 1KHz).
The second counter is now counting the number of millisecond that has been elapsed since the last time reset 1R goes high.
The design of this circuit is left as a tutorial problem for you to do.

A clock divider

- Another useful module is a clock divider circuit.
- This produces a symmetrical clock output, dividing the input clock frequency by a factor of 2*(K+1).

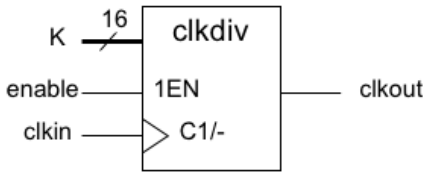


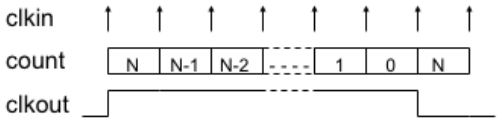

```
module clkdiv (
  clkin,        // Clock input signal to be divided
  enable,       // enable clk divider when high
  K,            // clock frequency divider is 2*(K+1)
  clkout     // symmetric clock output Fout = Fin / 2*(K+1)
);           //    End of port list

parameter   K_BIT = 16;     // change this for different number of bits divider
//-------------Input Ports-----------------------------
input clkin;
input enable;
input [K_BIT-1:0] K;

//-------------Output Ports----------------------------
output         clkout;
```

PYKC 19 Oct 2017 MSc Lab – Mastering Digital Design Lecture 2 Slide 17

Here is yet another useful form of a counter. I call this a clock divider. Unlike the **clktick** module, which produces a one cycle tick signal every N+1 cycle of the clock, this produces a symmetric clock output **clkout** at a frequency that is 2*(K+1) lower than the input clock frequency. Shown here is the module interface in Verilog. Again we have used the **parameter** statement to make this design ease of modification for different internal counter size.

clkdiv.v explained

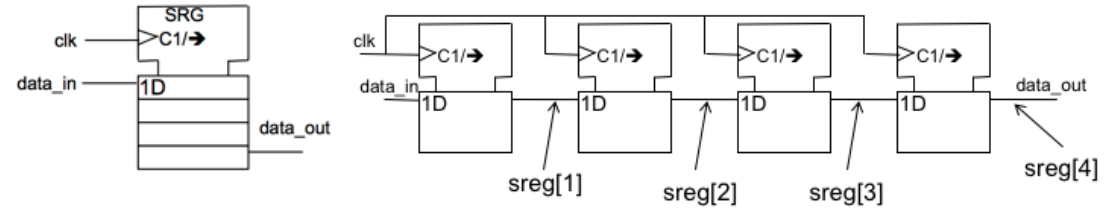




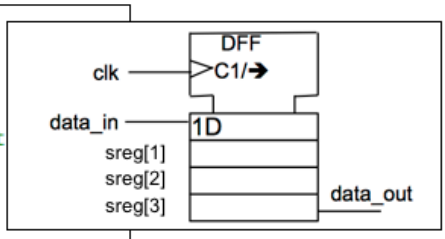

```
//-------------Output Ports Data Type------------------
// Output port can be a storage element (reg) or a wire
reg [K_BIT-1:0] count;
reg          clkout;

initial  clkout = 1'b0;

//------------ Main Body of the module ------------------------

   always @ (posedge clkin)
      if (enable == 1'b1)
         if (count == 10'b0)  begin
            clkout <= ~clkout;              // toggle the clock output signal
            count <= K;  // shift right one bit
            end
         else
            count <= count - 1'b1;

endmodule // End of Module clkdiv
```


The Verilog specification is similar to that for **clktick**. This also has an internal counter that counts from K to 0, then the output **clkout** is toggled whenever the count value reaches 0.

Shift Register specification in Verilog




```
module sreg4 (data_out, data_in, clk);

   output   data_out;    // serial data output
   input    data_in;     // serial data input
   input    clk;         // clock input

   reg  [4:1]  sreg;     // 4 stage D-FF for this shift
   initial sreg = 4'b0;
   always @ (posedge clk)
     begin
       sreg[4] <= sreg[3];
       sreg[3] <= sreg[2];
       sreg[2] <= sreg[1];
       sreg[1] <= data_in;
     end

    wire  data_out;
    assign data_out = sreg[4];
endmodule
```




```
sreg <= {sreg[3:1],data_in};
```

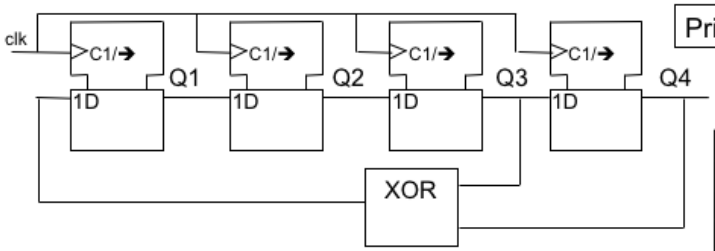
PYKC 19 Oct 2017 MSc Lab – Mastering Digital Design Lecture 2 Slide 19

To specify a shift register in Verilog, use the code shown here (in blue box). We use <= assignment to make sure that **sreg[4:1]** are updated only at the end of the always block.
On the right is a short-hand version of the four assignment statements:

sreg <= {sreg[3:1], data_in}

This way of specifying the input to the assignment is powerful. We use the concatenation operation { …. } to make up four bits from **sreg[3:0]** and **data_in** (with **data_in** being the LSB) and assign it to **sreg[4:1]**.

Linear Feedback Shift Register (LFSR)



Primitive polynomial: $1 + X^3 + X^4$

- Assuming that the initial value is 4'b0001.
- This shift register counts through the sequence as shown in the table here.
- This is now acting as a 4-bit counter, whose count value appears somewhat random.
- This type of shift register circuit is called "Linear Feedback Shift Register" or LFSR.
- Its value is sort of random, but repeat very N-1 cycles (where N = no of bits).
- The "taps" from the shift register feeding the XOR gate(s) is defined by a polynomial as shown above.

Q4	Q3	Q2	Q1	count
0	0	0	1	1
0	0	1	0	2
0	1	0	0	4
1	0	0	1	9
0	0	1	1	3
0	1	1	0	6
1	1	0	1	13
1	0	1	0	10
0	1	0	1	5
1	0	1	1	11
0	1	1	1	7
1	1	1	1	15
1	1	1	0	14
1	1	0	0	12
1	0	0	0	8
0	0	0	1	1

PYKC 19 Oct 2017 MSc Lab – Mastering Digital Design Lecture 2 Slide 20

We can also make a shift register count in binary, but in an interesting sequence. Consider the above circuit with an initial state of the shift register set to 4'b0001. The sequence that this circuit goes through is shown in the table here. It is NOT counting binary. Instead it is counting in a sequence that is sort of **random**. This is often called a pseudo random binary sequence (or counter).

The shift register connect this way is also known as a "Linear Feedback Shift Register" or LFSR. There is a whole area of mathematics devoted to this type of computation, known as "finite fields" which we will not consider on this course.

The circuit shown below is effective implementing a sequence defined by a polynomial shown: $1 + X^3 + X^4$. The term "1" specifies the input to the left-most D-FF. This signal is derived as an XOR function (which is the finite field '+') of two signals "tapped" from stage 3 (i.e. X^3) and stage 4 (i.e. X^4) of the shift register.

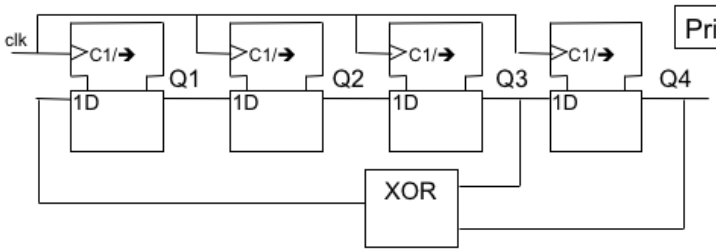
For a m stage LFSR, where m is an integer, one could always find a polynomial (i.e. tap configuration) that will provide maximal length. This means that the sequence will only repeat after 2^m-1 cycles. Such a polynomial is known as a "primitive polynomial".

m		m	
3	$1+X+X^3$	14	$1+X+X^6+X^{10}+X^{14}$
4	$1+X+X^4$	15	$1+X+X^{15}$
5	$1+X^2+X^5$	16	$1+X+X^3+X^{12}+X^{16}$
6	$1+X+X^6$	17	$1+X^3+X^{17}$
7	$1+X^3+X^7$	18	$1+X^7+X^{18}$
8	$1+X^2+X^3+X^4+X^8$	19	$1+X+X^2+X^5+X^{19}$
9	$1+X^4+X^9$	20	$1+X^3+X^{20}$
10	$1+X^3+X^{10}$	21	$1+X^2+X^{21}$
11	$1+X^2+X^{11}$	22	$1+X+X^{22}$
12	$1+X+X^4+X^6+X^{12}$	23	$1+X^5+X^{23}$
13	$1+X+X^3+X^4+X^{13}$	24	$1+X+X^2+X^7+X^{24}$

The table here shows some of the popular primitive polynomials for different value of m.

Since the output of such a counter is peudorandom, it is a commonly used circuit to produce random binary sequence for different applications.

lsfr4.v



Primitive polynomial: $1 + X^3 + X^4$

```
module lfsr4 (data_out, clk);

   output [4:1]   data_out;    // four bit random output
   input          clk;         // clock input

   reg  [4:1]  sreg;     // 4 stage D-FF for this shift register

   initial sreg = 4'b1;

   always @ (posedge clk)
       sreg <= {sreg[3:1], sreg[4] ^ sreg[3]};

   assign data_out = sreg;
endmodule
```


Here is the Verilog specification for a 4-bit LFSR.

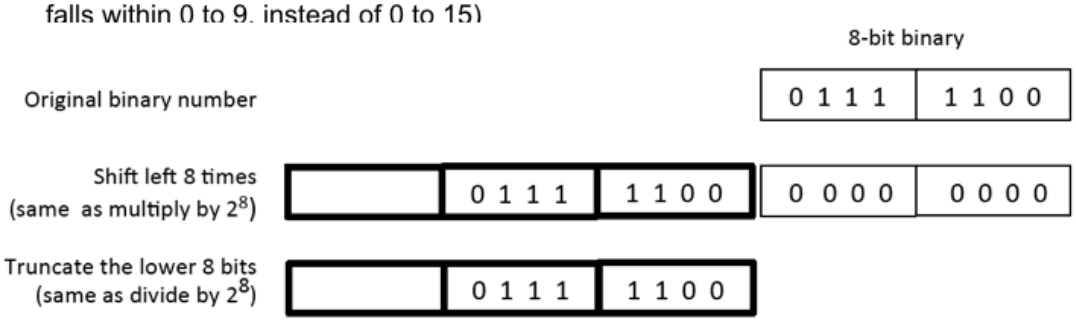
Displaying a binary number as decimal

- As the part 1 of the Lab Experiment VERI, you will be implementing the 7 segment decoder we designed in the last lecture. This will show every four binary bits as a hexadecimal digit on the display.
- Hex numbers are difficult to interpret. Often we would like to see the binary value displayed as decimal. For that we need to design a combinational circuit to converter from binary to binary-coded decimal. For example, the value 8'hff or 8'b11111111 is converted to 8'd255 in decimal.

We now take another example of a relative complex combinational circuit, and see how we can specify our design in Verilog.
The goal is to design a circuit that converts an 8-bit binary number into three x 4-bit binary coded decimal values (i.e. 12 bit).
There is a well-known algorithm called "**shift-and-add-3**" algorithm to do this conversion. For example, if we take 8-bit hexadecimal number 8'hff (i.e. all 1's), it has two hex digits. Once converted to binary coded decimal (BCD) it becomes 255 (3 BCD digits).

Shift and Add 3 algorithm [1] – shifting operation

- Let us consider converting hexadecimal number 8'h7c (which is decimal 8'd124)
- Shift the 8-bit binary number left by 1 bit = multiply number by 2
- Shifting the number left 8 times = multiply number by 2^8
- Now truncate the number by dropping the bottom 8 bits = divide number by 2^8
- So far we have done nothing to the number – it has the same value
- The idea is that, as we shift the number left into the BCD digit "bins", we make the necessary conversion to the hex number so that it confirms to the BCD rule (i.e. falls within 0 to 9, instead of 0 to 15)



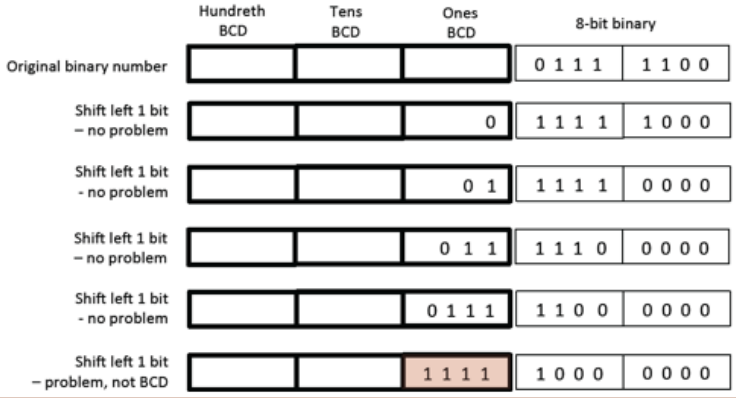
Before we examine this algorithm in detail, let us consider the arithmetic operation of shifting left by one bit. This is the same as a x 2 operation.

If we do it 8 times, then we have multiplied the original number by 256 or 2^8.

Now if you ignore the bottom 8-bit through a truncation process, you effectively divide the number by 256. In other words, we get back to the original number in binary (or in hexadecimal).

Shift and Add 3 algorithm [2] – shift left with problem

- If we take the original 8-bit binary number and shift this three times into the BCD digit positions. After 3 shifts we are still OK, because the **ones digit** has a value of 3 (which is OK as a BCD digit).
- If we shift again (4th time), the digit now has a value of 7. This is still OK. However, no matter what the next bit it, another shift will make this digit illegal (either as hexadecimal "e" or "f", both not BCD).
- In our case, this will be a "f"!



PYKC 19 Oct 2017 MSc Lab – Mastering Digital Design Lecture 2 Slide 24

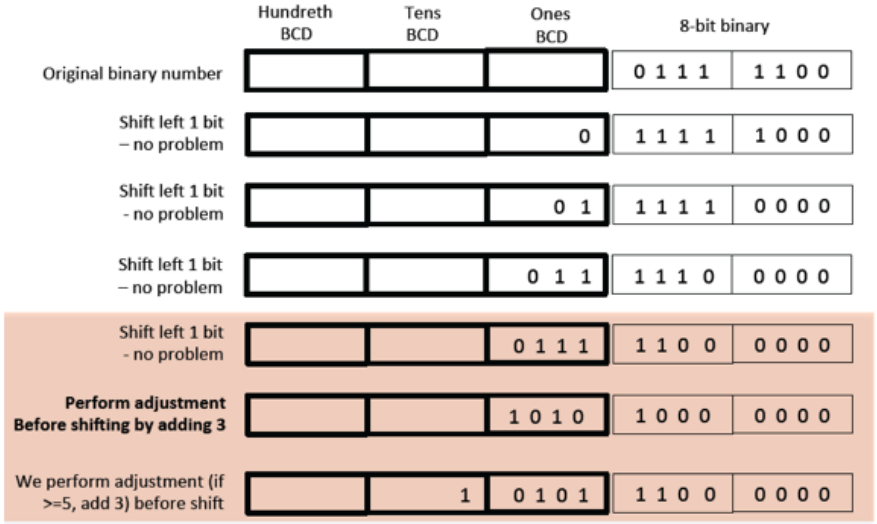
Our conversion algorithms works by shift the number left 8 times, but each time make an adjustment (or correction) if it is NOT a valid BCD digit.

Let us consider this example. We can shift the number four time left, and it will give a valid BCD digit of 7.

However, if we shift left again, then 7 becomes hex F, which is NOT valid. Therefore the algorithm demands that 3 is added to 7 (7 is larger or equal to 5) before we do the shift.

Shift and Add 3 algorithm [3] – shift and adjust

- So on the fourth shift, we detect that the value is > or = 5, then we adjust this number by adding 3 before the next shift.
- In that way, after the shift, we move a 1 into the tens BCD digit as shown here.



PYKC 19 Oct 2017 MSc Lab – Mastering Digital Design Lecture 2 Slide 25

The rationale of this algorithm is the following. If the number is 5 or larger, after shift left, we will get 10 or larger, which cannot fit into a BCD digit. Therefore if the number 5 (or larger) we add 3 to it (after shifting is adding 6), which measure we carry forward a 1 to the next BCD digit.

Shift and Add 3 algorithm [4] – full conversion

- In summary, the basic idea is to shift the binary number left, one bit at a time, into locations reserved for the BCD results.
- Let us take the example of the binary number 8'h7C. This is being shifted into a 12-bit/3 digital BCD result of 12'd124 as shown below.

	Hundreth BCD	Tens BCD	Ones BCD	8-bit binary	
Original binary number				0 1 1 1	1 1 0 0
Shift left three times no adjust			0 1 1	1 1 1 0	0
Shift left Ones = 7, ≥5			0 1 1 1	1 1 0 0	
Add 3			1 0 1 0	1 1 0 0	
Shift left Ones = 5		1	0 1 0 1	1 0 0	
Add 3		1	1 0 0 0	1 0 0	
Shift left 2 times Tens = 6, ≥5		1 1 0	0 0 1 0	0	
Add 3		1 0 0 1	0 0 1 0	0	
Shift left BCD value is correct	1	0 0 1 0	0 1 0 0		

PYKC 19 Oct 2017 MSc Lab – Mastering Digital Design Lecture 2 Slide 26

To recap: the basic idea is to shift the binary number left, one bit at a time, into locations reserved for the BCD results. Let us take the example of the binary number 8'h7C. This is being shifted into a 12-bit/3 digital BCD result as shown above.

After 8 shift operations, the three BCD digits contain respectively: hundredth digit = 4'b0001, tens digit = 4'b0010 and ones digit = 4'b0100, thus representing the BCD value of 124.

The key idea behind the algorithm can be understood as follow (see the diagram in the slide):

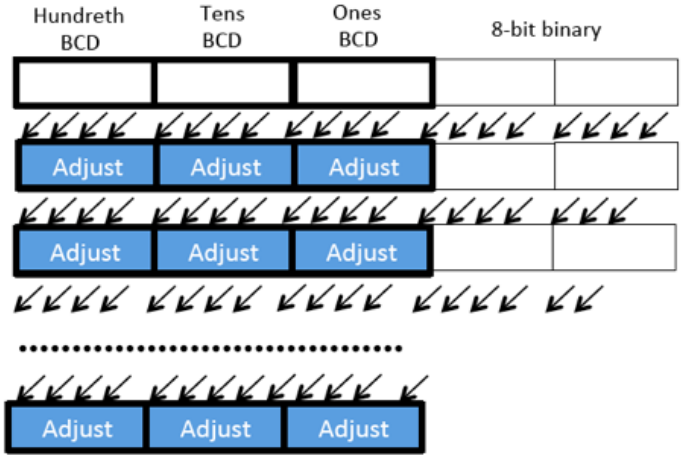
1.Each time the number is shifted left, it is multiplied by 2 as it is shifted to the BCD locations;

2.The values in the BCD digits are the same as as binary if its value is 9 or lower. However if it is 10 or above it is not correct because for BCD, this should carry over to the next digit. A correction must be made by adding 6 to this digit value.

3.The easiest way to do this is to detect if the value in the BCD digit locations are 5 or above BEFORE the shift (i.e. X2). If it is ≥5, then add 3 to the value (i.e. adjust by +6 after the shift).

Hardware implementation (1) – binary to BCD

- The hardware to perform binary to BCD conversion is shown below.
- Shifting is easy – just wiring all signals one position to the left.
- For each of the BCD locations, we need an "adjust" module which perform the follow operation: if the value is ≥5, then add 3.



In order to understand how to we may implement this converter in hardware, you have to understand that shifting in hardware is easy. You just need to connect signals with one bit shift to the left. It DOES NOT need any gates, just wires!

Now we also need to do the adjust module, which simply performs the operation:

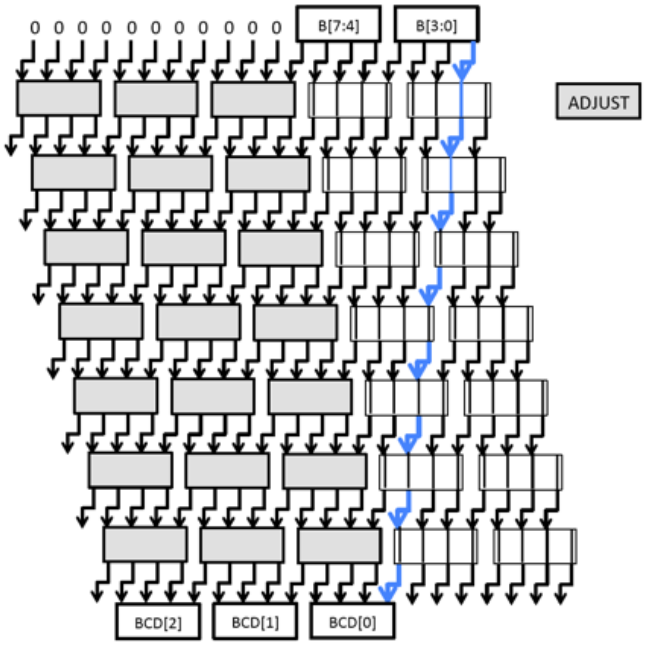
if (in >= 5) out = in + 3 else out = in

The easiest way to implement such a module is to use a **case statement**. This is set as a tutorial problem in Problem Sheet 1.

Hardware implementation (2) – array of gates

- Here is the full array of logic gates to do the conversion.
- After 8 shift and adjustment on the way, the result should be three BCD digits.
- Each ADJUST block perform the following operation:

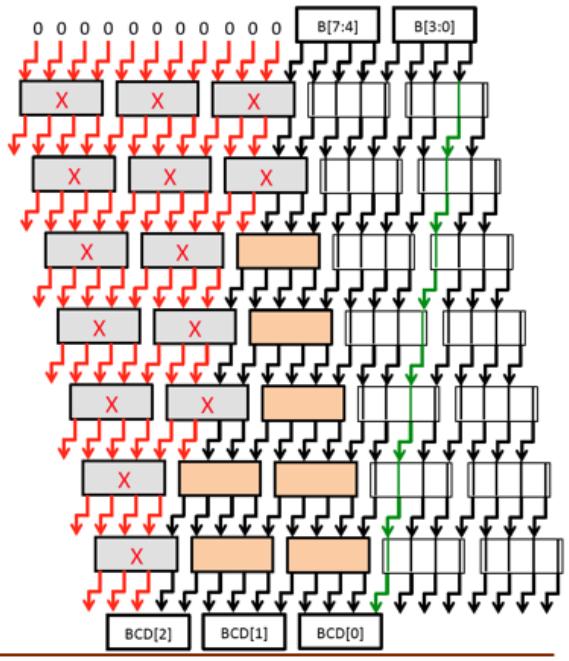
```
if (input >= 5)
        output = input + 3
else
        output = input
```



The entire full array is shown here. The shade module is the adjust module (which we call: **add3_ge5**).
As I said in the last slide, the easiest way to implement (specify) **add3_ge5** is using a case statement.
The BLUE signal path traces what happens to the least significant bit of the original number.

Hardware implementation (3) – propagate 0 to simplify

- If we now propagate forward all the 0s, we can eliminate all ADJUST modules except those in RED.
- All the others are just wires from input to output because the input values are GUARANTEED to be smaller than 5.



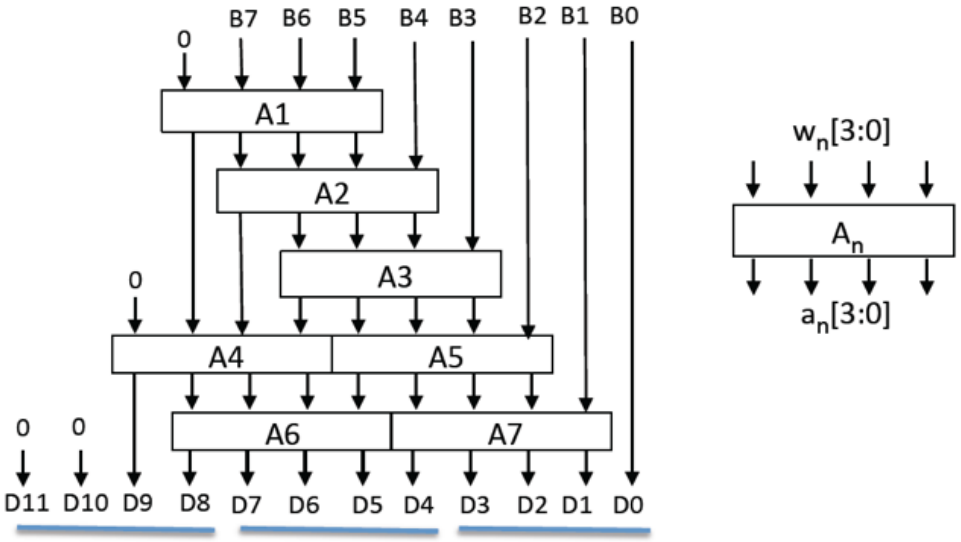
PYKC 19 Oct 2017 MSc Lab – Mastering Digital Design Lecture 2 Slide 29

The full array is more complicated than need be. If we propagate the '0's forward in the array of gates, you will find those marked with 'X' will always have its input less than 5. In which, **output = input** in these modules. THIS IS JUST A SET OF FOUR WIRES.

The only remaining **add3_ge5** modules are those shaped in orange.

Putting things together

- Once you have specified the **adjust module (A_n)** in Verilog, you can wire up the entire converter as shown here:



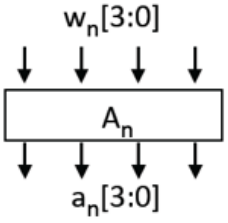
PYKC 19 Oct 2017 MSc Lab – Mastering Digital Design Lecture 2 Slide 30

After simplification, here are ALL the remaining **add3_ge5** modules for the 8-bit binary to BCD conversion (bin2bcd8). I have labeled the input ports to **add3_ge3 wn[3:0]** and the output parts **an[3:0]** where n is 1 to 7.

Binary to BCD conversion in Verilog

- Here is the Verilog code to perform the 8-bit binary to BCD conversion:

```
module bin2bcd8 (B, BCD_0, BCD_1, BCD_2);

   input [7:0] B;     // binary input number
   output [3:0]   BCD_0, BCD_1, BCD_2;   // BCD digit LSD to MSD

   wire [3:0]  w1,w2,w3,w4,w5,w6,w7;
   wire [3:0]  a1,a2,a3,a4,a5,a6,a7;

   // Instantiate a tree of add3-if-greater than or equal to 5 cells
   //    ... input is w_n, and output is a_n
   add3_ge5 A1 (w1,a1);
   add3_ge5 A2 (w2,a2);
   add3_ge5 A3 (w3,a3);
   add3_ge5 A4 (w4,a4);
   add3_ge5 A5 (w5,a5);
   add3_ge5 A6 (w6,a6);
   add3_ge5 A7 (w7,a7);

   // wire the tree of add3 modules together
   assign  w1 = {1'b0, B[7:5]};      // wn is the input port to module An
   assign  w2 = {a1[2:0], B[4]};
   assign  w3 = {a2[2:0], B[3]};
   assign  w4 = {1'b0, a1[3], a2[3], a3[3]};
   assign  w5 = {a3[2:0], B[2]};
   assign  w6 = {a4[2:0], a5[3]};
   assign  w7 = {a5[2:0], B[1]};

   // connect up to four BCD digit outputs
   assign BCD_0 = {a7[2:0],B[0]};
   assign BCD_1 = {a6[2:0],a7[3]};
   assign BCD_2 = {2'b0,a4[3],a6[3]};

endmodule
```



Assuming that we have designed a module "**add3_ge5**" to perform the adjustment as required, the converter can be implemented in Verilog by simply "WIRING UP" the various modules together.

The interconnections are specified in the wire statements.
The next block is instantiating 7 **add3_ge5** modules.
The next block of code is to wire the modules together.
Finally the last statements are to connect up the signals from the modules to the output ports.