

**System Design Through VERILOG**  
**Prof. Shaik Rafi Ahmed**  
**Department of Electrical and Electronics Engineering**  
**Indian Institute of Technology, Guwahati**

**Test benches**  
**Lecture - 24**  
**Static RAM and Braun Multiplier**

(Refer Slide Time: 00:31)

**Synchronous Static RAM**

**Verilog code:**

```

module syncRAM (dataIn, dataOut, Addr, CS, WE, RD, Clk);
// parameters for the width
parameter ADR = 8;
parameter DAT = 8;
parameter DPTH = 8;
//ports
input [7:0] dataIn;
output reg [DAT-1:0] dataOut;
input [ADR-1:0] Addr;
input CS, WE, RD, Clk;
//Internal variables
reg [DAT-1:0] SRAM [DPTH-1:0];
always @ (posedge Clk)
begin
if (CS == 1'b1) begin
if (WE == 1'b1 && RD == 1'b0) begin
SRAM [Addr] = dataIn;
end
else if (RD == 1'b1 && WE == 1'b0) begin
dataOut = SRAM [Addr];
end
end;
end;
end;
endmodule

```

In the case studies initially we discussed about arithmetic logic unit and then register file. So, which will acts as a small memory which is present inside the CPU. And the other case study that we are going to discuss today is static random access memory.

So, as we have discussed in the earlier lectures also. So, if we take the CPU or the microprocessor. So, difference between the microprocessor and CPU is, a CPU built into a single integrated circuit is called microprocessor.

So, here we have arithmetic logic unit which is one of the important block and then we have the register array or register file, this is register file which we have discussed in the previous slides. Now the memory is external to this CPU.

So, this register file will act as a small memory, but the main memory will be interfaced outside the microprocessor or CPU. Now we will discuss about this memory. Again memory

there are two types of the memories one is called as static RAM and dynamic RAM in random access memory.

In read only memory also there are different types, but here we will discuss about the random access memory. So in this static random access memory is having some advantages like it does not require any refreshing circuitry whereas, dynamic RAM requires refreshing circuitry.

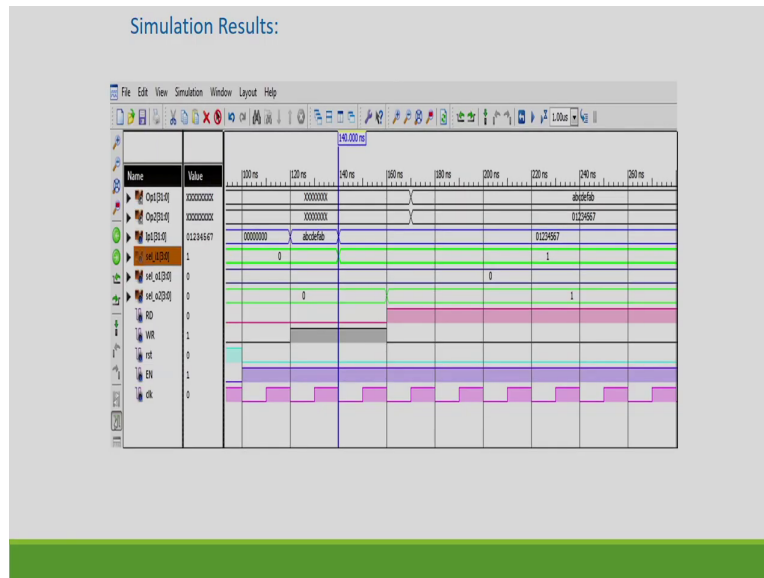
So, here I will discuss about the static random access memory. So, you take any memory, so what are the different signals for the memory? We have discussed this in earlier lectures also.

So, in the memory there will be some address lines which is unidirectional address lines and there will be some data lines which are bidirectional and then any chip will be having chip select. Normally in ICs chip select signal will be active low, but here for the sake of simplicity I am assuming active high signal.

So, CS is equal to 1 means this IC is selected then we can read as well as write. So, there is a read signal then we have one write enable signal and of course, we require the clock.

This is the write enable signal, read signal, chip select signal, clock then this data as I have told bidirectional this data in is this direction data is inputted dataOut is this direction this will be inside this, this will be outside this. Similarly, address there will be some address line this is unidirectional.

(Refer Slide Time: 04:39)



Now this address lines is going to decide the size of the memory the address lines is going to decide the size of the memory. So, here I am taking 8 address lines. So, in general if n address lines are there, the memory size is given by 2 raised to the power of n. So, this is equal to 2 raised to the power of 8 because n is equal to 8 in this case this is I think 256 bytes.

So, that means, this memory will be having 256 different locations 0 1 2 so, on up to the last one is 255, 256 different locations. And this data line is going to decide the number of bits in each location because this is also 8 bit. So, each location is capable of storing 8 bits of information this will have 8 bits, this will have 8 bits. So, like that each location will have 8 bits of the information.

Now we can perform either read operation or write operation using two different read and write signals. So, in order to perform either read or write operation the initial condition is chip select should be 1. If chip select is not 1 we cannot perform neither read operation nor write operation. So, we have written the Verilog code for this one and test bench and the corresponding waveforms.

So, we can see here the module name is synchronous static RAM. So, we will call as synchronous RAM. So, the inputs are dataIn is the input, dataOut is the output, address is input, chip select is input, write enable, read, clock then we are defining this later. So, on the

parameters we are going to define as address as 8, data as 8 and data path is also 8. So, the number of bits to be proceed.

And then we have defined the input port as dataIn, this dataIn I am defining here. So, this is data minus 1 0 what is DAT? Is 8, so this will be 7 is to 0. So, this is total 8 lines this becomes 7 is to 0 because DAT is 8 similarly output register. So, this also again 7 is to 0. So, 8-bit dataIn 8-bit dataOut then address is also 8 bit because ADR is 8 this is also 7 is to 0.

And these are single bit signals CF, CS, chip select, write enable, read, clock then the internal variable. This is external register we have defined this as register external register we have defined as dataOut. So, this dataOut will be taking the data from the memory. So, we can call this inside this memory. So, we can have 256 locations. So, we are calling this SRAM data minus 1 7 is to 0 as SRAM 7 is to 0.

So whatever this data that is available here as 8 bits, this we are calling as SRAM 7 is to 0. So, this data we are going to output onto this data. So, this is internal variable. Up to here its initialization coming for the main program always at positive edge of the clock we are assuming that the clock is positive edge. Begin if chip select equal to 1 if chip select is 0 no operation. So, this chip will not be selected we cannot perform neither read operation nor write operation.

So, in order to perform either read or write operation the primary condition is chip select should be 1, then we are checking for the write enable and read. So, this is write enable is 1, read is 0 because these are active high signals this will indicate write operation. So, in write operation what we will do? We will write some data into the memory we will take the data onto the dataIn and we will write into the memory.

So, what is SRAM? This SRAM is equal to dataIn; whatever the data that is taken in the dataIn will be write into the SRAM address that we have defined here end. On the other hand if read is equal to 1 and write enable is 0, this is read operation.

In read operation reverse whatever the data that is available here that will be outputted onto the dataOut. So, for that the assignment is dataOut is equal to SRAM address these two are opposite then end. So, this is the end module.

So, this is actually behavioral modeling. So, we are not discussed about the internal circuitry of this synchronous static RAM. So, we just discussed about the external devices only external the signals of the SRAM.

(Refer Slide Time: 11:05)

```

Test bench
`timescale 1ns / 1ps ✓
module syncRAM_tb;
// Inputs
✓ reg [7:0] dataIn;
reg [7:0] Addr;
reg CS;
reg WE;
reg RD;
reg Clk;
// Outputs
✓ wire [7:0] dataOut;
// Instantiate the DUJ
syncRAM uut (.dataIn(dataIn),
.dataOut(dataOut),.Addr(Addr),.CS(CS),
.WE(WE),.RD(RD),.Clk(Clk));
initial begin
// Initialize Inputs
dataIn = 8'h0;
Addr = 8'h0;
CS = 1'b0;
WE = 1'b0;
RD = 1'b0;
Clk = 1'b0;
// Wait 100 ns for global reset to finish
#100;
// Add stimulus here
dataIn = 8'h0;
Addr = 8'h0;
CS = 1'b1;
WE = 1'b1;
RD = 1'b0;
#20;
dataIn = 8'h0;
Addr = 8'h0;
#20;
dataIn = 8'h1;
Addr = 8'h1;
end

```

Now what is the corresponding test bench? This is the corresponding test bench. So, we have timescale. So, 1 nanosecond by precision of 1 picoseconds then we know that all the inputs will be declared as registers in test bench and all the outputs will be declared as wire.

Then after initialization then the next step is instantiate the design under test, here I am using the same names we can use the different names also as you have discussed in the previous lecture with an example.

So, here I have used the same names dot followed by CS and then CS. So, this will give the connections between the test bench and the design under test. Now initially dataIn we are taking 8 hexadecimal zeros means 0 0 0 0 0 0 0 H. And then address is also 8 hexadecimal, chip select is single bit 1 bit binary 0 initially I have taken as chip select as 0. So, write is also 0, read is also 0, clock is 0.

So, we will wait 100 nanoseconds for the reset. Because chip select is 0 means it is some sort of a reset type of the operation. Then after 100 nanoseconds I will take the dataIn as 0 0 0 0 only address is also 0 0 0 0 the same thing I have taken, but this time chip select is 1 then

write is 1 read is 0 means this is write operation. After 100 nanoseconds it will perform the write operation with data as all zeros address is also 0 0 means what does it mean?

So, we are going to write in the first location this is all zeros 0 0 0 0 see here we are going to enter all zeros for this assignment. Then after 20 nanoseconds, so I will make this dataIn is also 0 address is also 0. So, I will keep as it is and then after 20 nanoseconds I will make all ones means totally 140 nanoseconds. So, this data become all ones address becomes all ones the last one these are all ones this is address is also all ones.

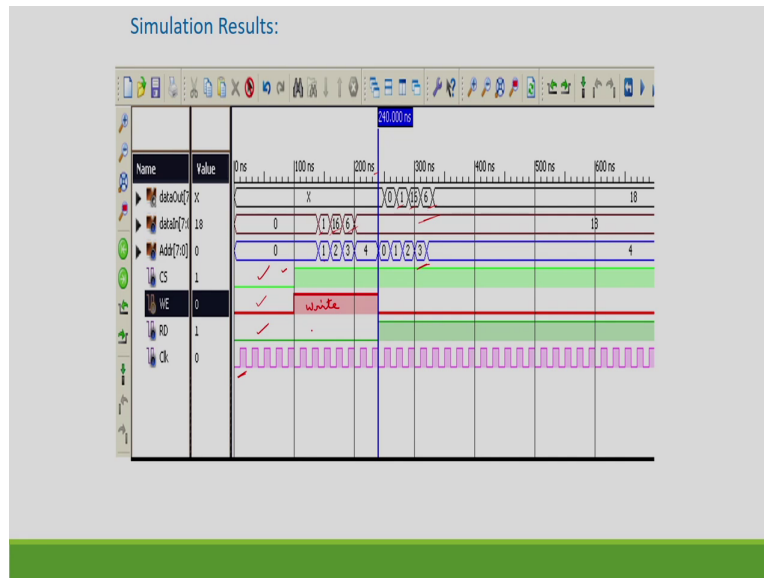
(Refer Slide Time: 13:59)

```
Test Bench contd..
#20;                               #20; ✓
dataIn = 8'h10;                     Addr = 8'h1; ✓
Addr = 8'h2;                         #20; ✓ ✓
#20;                               Addr = 8'h2; ✓
dataIn = 8'h6;                       #20; ✓
Addr = 8'h3;                         Addr = 8'h3; ✓
#20;                               #20; ✓
dataIn = 8'h12;                      Addr = 8'h4; ✓
Addr = 8'h4;                         end
#40;                               always #10 Clk = ~Clk; ✓
Addr = 8'h0;                         endmodule ✓
WE = 1'b0; } ✓
RD = 1'b1; } ✓
```

Then after another 20 nanoseconds I will make 8 bit hexadecimal 1 all zeros, 8-bit hexadecimal all twos, after 20 nanoseconds I am giving different values of dataIn address, different value of dataIn address. After total of this much time the total time is 40 plus 20 plus 20 plus 20 plus the previous 120 it seems. So, now, I will make this as a read operation because write is 0 read is 1.

Again for every 20 nanoseconds I will change the bit pattern of the address. Then for the clock I will use this similar type of the logic that we have discussed in the earlier examples. So, for every 10 nanoseconds clock becomes clock bar means, the clock period is equal to 20 nanoseconds clock frequency is 1 by 20 nanoseconds.

(Refer Slide Time: 15:02)



So, you can see the corresponding waveforms. You can see that initially for 100 nanoseconds, the write signal is 0 read chip select is 0 write is 0 read is 0 that is what we have written in the test bench. Then after 100 and nanoseconds, we make this chip select equal to 1 and then write operation as I have discussed this is write operation because write equal to 1 read is 0.

So, while writing also we will write. So, whatever the data that is in and then address. So, in that address this data will be written this is dataOut this data whatever the dataIn is 1 is outputted here, 16 is outputted here, 6 is outputted here. Because the corresponding timings are 220 nanoseconds and the clock is complemented for every 10 nanoseconds.

So we can check this values from the values that have given in the test bench these values you can verify. So, this is about the next third case study, so which is on the memory. So, with this 3 case studies you should be able to design now any microprocessor. So, because the main blocks we have already discussed. So, arithmetic logic unit, register file, external memory the remaining blocks are minor only. So, we have some control circuit is like decoders, multiplexers.

So, with this knowledge you should be able to design a CPU or microprocessor which can perform the different operation you can vary the specifications. So, the number of operation that is to be performed, the number of registers, the address, width of the microprocessor, the

data lines by changing the different parameters we can design the different microprocessors. So the next case studies on digital filter design.

(Refer Slide Time: 17:22)

**Digital Filter Implementation using Verilog**

- Filter is a frequency selective system
- Filter can be FIR or IIR

✓ **FIR Filter**

Consider a 3-tap FIR filter:  $y(n) = ax(n) + bx(n-1) + cx(n-2)$

$T_M$ : multiplication-time  
 $T_A$ : Addition-time

finite impulse response (FIR)  
infinite .. ..

3 multipliers → unsigned signed Braun multiplier  
2 adders

This is also another important relation. So, the people who works on the Verilog implementation of dsp algorithms, so one of the important dsp block is digital filter. So, how to implement digital filter using Verilog?

So, we know that a filter is a frequency selective network it selects some frequencies and rejects the remaining frequencies. Based on whether this is going to select the low frequencies high frequencies a band of frequencies or it rejects the band of frequencies we have basically four types of the filters low pass, high pass, band pass, band reject.

Again the filters will be broadly classified into two types one is called as FIR another is called IIR. FIR is finite impulse response or infinite impulse response. So, here we will not discuss about how to design this.

So, designing of this filters will be studied at dsp course, here in this particular course we will discuss how to implement this filters using Verilog. So, we will discuss about the Verilog code then test benches and then the output waveforms. So, I will start with the FIR filter later I will discuss about the IIR filter.



So, if you consider a 3 tap FIR filter the difference equation for the 3 tap FIR filter is. So,  $y[n]$  is equal to  $a \cdot x[n] + b \cdot x[n-1] + c \cdot x[n-2]$ . So, this is output, this is input these are the taps or weights we can also call this as taps or weights. So, by properly choosing this abc values we can set the specifications of the filters that we are not going to discuss.

Suppose, if I take a low pass filter. So, ideally it should pass all the frequencies from 0 to  $\omega_c$  with unity gain it has to reject all the frequencies beyond  $\omega_c$ , but practically we will get something like this one. So, there are four parameters basically one is called as pass band gain, pass band frequency, stop band gain, stop band frequency.

So, to satisfy these parameters we will design filter. So, accordingly we will fix abc. So, that filter design you might have studied at dsp course now here we will be discussing only implementation. So, here basically we require 3 multipliers which says some T M is the computation T M of each multiplier and two adders. So, initially I will start with a simple multiplier which is unsigned multiplier, multiplier can be either signed or unsigned.

But in most of the practical applications we need the signed multipliers only because the data can be either positive or negative. But in order to better understand this filter I will first start with the unsigned multiplier then later the IIR filter I will implement with the signed multiplier.

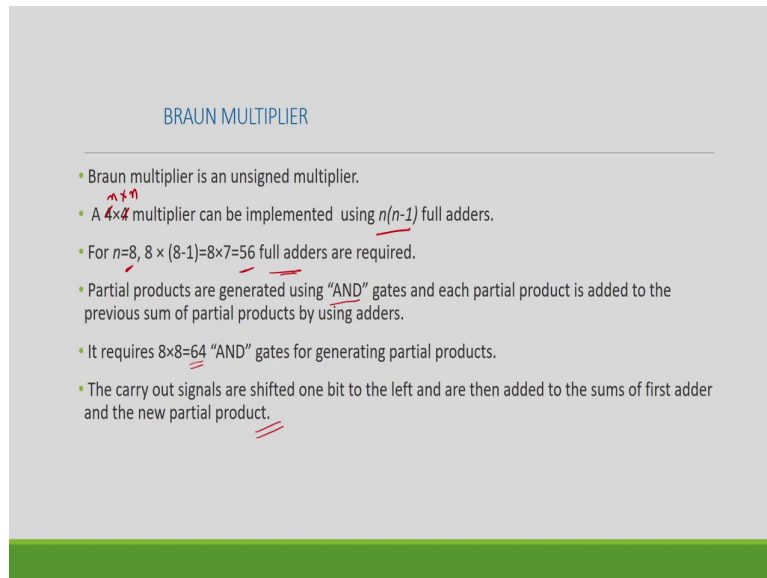
FIR filter I will use unsigned multiplier one of the important unsigned multiplier is Braun multiplier. And then this adder you can use any of the adders, but I will use here ripple carry adder which we have discussed in the earlier lectures.

This adder we are going to multiply with ripple carry adder this multiplies with Braun multiplier then I will implement the FIR filter. So, for that initially we have to I mean implement Braun multiplier, you write down the Verilog code then Verilog code for the RCA, then Verilog code for this delays which are D flip flops this already I have discussed in the earlier lectures this also I have already discussed.

So, now, we will discuss about the Braun multiplier here and then we will instantiate this D flip flop, Braun multiplier and RCA to implement the final FIR filter. So, to start with this we

will first discuss about the Braun multiplier, Braun multiplier is unsigned multiplier as I have told.

(Refer Slide Time: 22:48)



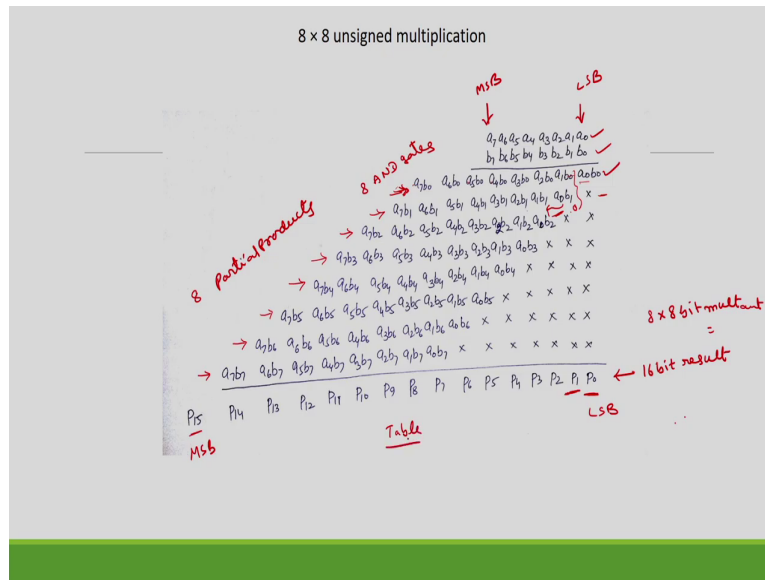
The slide is titled "BRAUN MULTIPLIER" and contains the following text:

- Braun multiplier is an unsigned multiplier.
- A  $4 \times 4$  multiplier can be implemented using  $n(n-1)$  full adders.
- For  $n=8$ ,  $8 \times (8-1) = 8 \times 7 = 56$  full adders are required.
- Partial products are generated using "AND" gates and each partial product is added to the previous sum of partial products by using adders.
- It requires  $8 \times 8 = 64$  "AND" gates for generating partial products.
- The carry out signals are shifted one bit to the left and are then added to the sums of first adder and the new partial product.

If I consider a 4 bit by 4-bit multiplier this requires in general for n sorry this is n actually n bit by n bit multiplier we require n into n minus 1 full adders. If I take n is equal to 8, this will be 8 into 8 minus 1 which is 56 full adders are required and to generate the partial products we require the AND gates. The number of AND gates required is simply n into n, n is 8. So, total 64 AND gates are required.

Then as usual in any multiplication we will first find out the partial products and then we will shift and then add. So, this will be better understand from the algorithm in the next slide.

(Refer Slide Time: 23:39)



So, I have taken basically two 8 bit numbers  $a_7 a_6 \dots a_0$  and  $b_7 b_6 \dots b_0$ , where  $a_7$  is MSB most significant bit and  $a_0 b_0$  are LSB's least significant bits. Then we will perform this multiplication in the conventional manner we will multiply first with  $b_0$   $a_0 b_0$  a 1 ok we will put the values here then we will shift and then we will put the values. So, like that this is the complete table.

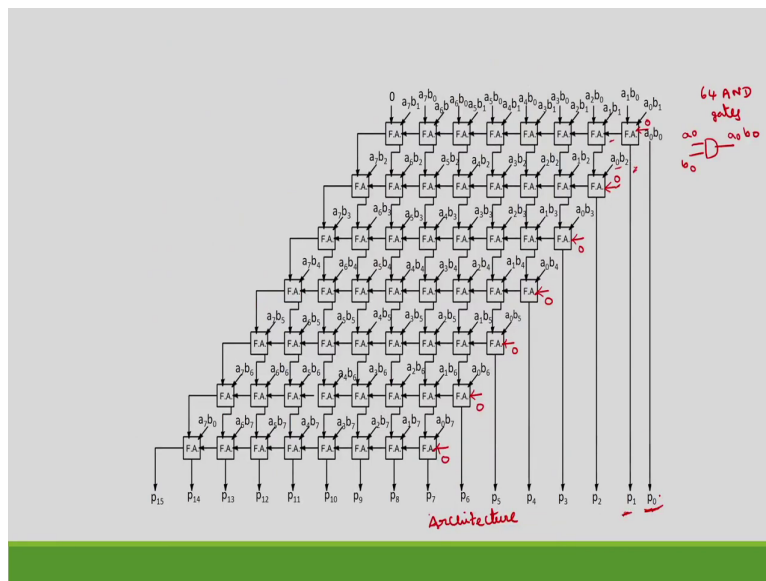
so here to get the final 16-bit output, they are performing two 8 bit by 8 bit multiplications, the result can be maximum result can be 16 bit. So, that 16-bit result will be showed by P 15 to P 0 this is LSB, this is MSB then how to implement this P 0 to P 15? So, P 0 implementation is basically AND gate nothing is divided AND gate. So, totally we have to generate all this partial product, this is a partial product, this is another partial product, this is another partial product.

So, totally we have eight partial products. Once if we generate the eight partial products then we can perform the addition of this partial products using fully adders. So, in order to generate this partial products, how many AND gates are required? In each partial product we require 8 AND gates and total 8 partial products. So, total 8 into 8 64 AND gates. So, this is 1 2 3 4 5 6 7 8. So, like that total 64 AND gates are required.

So we can write directly and the keyword primitive and you can obtain this and the all the outputs of this, AND gates we will assign some wires because the outputs of this AND gates you have to use to generate the final product terms. And then in the second row to get P 1 what you have to do? a 1 b 0 a 0 b 1 you have to adder of course, this you can perform with half adder also, but to have the uniformity I will make the third bit as 0 and I will use full adder to perform this.

Then to perform this, so this 3 bits are directly there, but here you may get one carry from here also. So, total four bits we have to add. So, we require one full adder one half adder, but to have the uniformity again, I will use all full adders only if it requires only two bits third bit I will make as 0. So, this table is shown in the next slide.

(Refer Slide Time: 26:41)



We can see that a 0 b 0 I am not showing the AND gate here. So, total I am using 64 AND gates to generate all the product terms. So, this is generated through AND gate with a 0 b 0 as the inputs output is a 0 b 0.

So, this itself is your P 0 ok then P 1 is as I have told a 1 b 0 plus a 0 b 1 the third input here I have not shown this is 0 for all this is 0, this is 0. In fact, for this half errors are enough, but to have the uniformity because we can instantiate full adders. So, I am implementing the entire algorithm using only full adders.

So, that is why the third bit I make as zeros. So, we can see here this a 1 b 0 plus a 0 b 1 a 1 b 0 plus a 1 a 0 b 1. So, and 0 full adder the sum bit is this along with this carry bit we have to perform addition of 3 more bits. So, this a 3 b 0 a 1 b 1 a 0 b 2 along with this carry. So, that will be performed by using this. So, we are adding two bits along with the carry, then we will get sum and carry this sum and the third bit then other bit we will make as 0, so that we will get P 2.

So, like that we can easily verify. So, this table can be mapped on to this architecture there is a one to one mapping is there. Now we will first write the Verilog code for the Braun multiplier and then we will just repeat the codes of ripple carry adder which we have discussed in the earlier lectures and then D flip flop. Then finally, we will instantiate all these 3 blocks to implement the final 3 tap FIR filter. So, this is the basic module for the Braun multiplier.

(Refer Slide Time: 28:56)

```

`timescale 1ns / 1ps
module braun_multiplier
(input [7:0] a,input [7:0] b,
output [15:0] prod);
wire prod_buff [62:0];
wire carry_buff [54:0];
wire sum_buff[41:0];
and a1(prod[0],a[0],b[0]);
//partial products
and a2(prod_buff[0],a[1],b[0]);
and a3(prod_buff[1],a[0],b[1]);
and a4(prod_buff[2],a[2],b[0]);
and a5(prod_buff[3],a[1],b[1]);
and a6(prod_buff[4],a[0],b[2]);
and a7(prod_buff[5],a[3],b[0]);
and a8(prod_buff[6],a[2],b[1]);
and a9(prod_buff[7],a[1],b[2]);
and a10(prod_buff[8],a[0],b[3]);
and a11(prod_buff[9],a[3],b[1]);
and a12(prod_buff[10],a[2],b[2]);
and a13(prod_buff[11],a[1],b[3]);
and a14(prod_buff[12],a[3],b[2]);
and a15(prod_buff[13],a[2],b[3]);
and a16(prod_buff[14],a[3],b[3]);
and a17(prod_buff[15],a[4],b[0]);
and a18(prod_buff[16],a[4],b[1]);
and a19(prod_buff[17],a[5],b[0]);
and a20(prod_buff[18],a[5],b[1]);
and a21(prod_buff[19],a[6],b[0]);
and a22(prod_buff[20],a[6],b[1]);
and a23(prod_buff[21],a[7],b[0]);
and a24(prod_buff[22],a[7],b[1]);
and a25(prod_buff[23],a[4],b[2]);
and a26(prod_buff[24],a[5],b[2]);
and a27(prod_buff[25],a[6],b[2]);
and a28(prod_buff[26],a[7],b[2]);
and a29(prod_buff[27],a[4],b[3]);
and a30(prod_buff[28],a[5],b[3]);
and a31(prod_buff[29],a[6],b[3]);
and a32(prod_buff[30],a[7],b[3]);
and a33(prod_buff[31],a[0],b[4]);
and a34(prod_buff[32],a[1],b[4]);
and a35(prod_buff[33],a[2],b[4]);
and a36(prod_buff[34],a[3],b[4]);
and a37(prod_buff[35],a[4],b[4]);
and a38(prod_buff[36],a[5],b[4]);

```

So, this is 8 bit input, 8 bit output and product output is 16 bit P 0 to P 15 then we require the buffers as I have told at the output of each AND gate we need a buffer. Because that output of the AND gate we have to give as input for the full adders. So, we require total of 63 buffers and similarly.

So, carry buffers we require 55 sum buffers 41. So, the first product 0 is simply a 0 b 0 because we do not need any full adders, here we do not need any full adder to generate P 0.

For P 1 we are using a 1 b 0 and then using AND gate. So, we have to generate all the product terms that is why these many AND gates are 64 AND gates total this is 1 AND gate remaining all are 64 AND gates.

(Refer Slide Time: 30:06)

```
and a39(prod_buff[37],a[6],b[4]);
and a40(prod_buff[38],a[7],b[4]);
and a41(prod_buff[39],a[0],b[5]);
and a42(prod_buff[40],a[1],b[5]);
and a43(prod_buff[41],a[2],b[5]);
and a44(prod_buff[42],a[3],b[5]);
and a45(prod_buff[43],a[4],b[5]);
and a46(prod_buff[44],a[5],b[5]);
and a47(prod_buff[45],a[6],b[5]);
and a48(prod_buff[46],a[7],b[5]);
and a49(prod_buff[47],a[0],b[6]);
and a50(prod_buff[48],a[1],b[6]);
and a51(prod_buff[49],a[2],b[6]);
and a52(prod_buff[50],a[3],b[6]);
and a53(prod_buff[51],a[4],b[6]);
and a54(prod_buff[52],a[5],b[6]);
and a55(prod_buff[53],a[6],b[6]);
and a56(prod_buff[54],a[7],b[6]);
and a57(prod_buff[55],a[0],b[7]);
and a58(prod_buff[56],a[1],b[7]);
and a59(prod_buff[57],a[2],b[7]);
and a60(prod_buff[58],a[3],b[7]);
and a61(prod_buff[59],a[4],b[7]);

and a62(prod_buff[60],a[5],b[7]);
and a63(prod_buff[61],a[6],b[7]);
and a64(prod_buff[62],a[7],b[7]);

//Full adder
`timescale 1ns / 1ps
module fa( output sum, output carry_out,
input x, input y, input carry_in );wire
prop,gen,pr_cin;xor x1(prop,x,y);xor
x2(sum,prop,carry_in);and
a1(pr_cin,prop,carry_in);and a2(gen,x,y);or
o1(carry_out,gen,pr_cin); endmodule
```

This is straight forward this one after the AND gates for the next one is full adder. So, we have to generate this full adder code, this we have already discussed in the earlier lectures. So, we can easily go through this one. So, using this full adder we will instantiate these full adders in the Braun multiplier.

(Refer Slide Time: 30:25)

```
// full adder array
fa f1(prod[1],carry_buff[0],prod_buff[0],prod_buff[1],1'b0);//
fa f2(sum_buff[0],carry_buff[1],prod_buff[3],prod_buff[2],carry_buff[0]);//
fa f3(sum_buff[1],carry_buff[2],prod_buff[6],prod_buff[5],carry_buff[1]);//
fa f4(sum_buff[2],carry_buff[3],prod_buff[15],prod_buff[9],carry_buff[2]);//
fa f5(sum_buff[3],carry_buff[4],prod_buff[17],prod_buff[16],carry_buff[3]);//
fa f6(sum_buff[4],carry_buff[5],prod_buff[19],prod_buff[18],carry_buff[4]);//
fa f7(sum_buff[5],carry_buff[6],prod_buff[21],prod_buff[20],carry_buff[5]);//
fa f8(sum_buff[6],carry_buff[7],1'b0,prod_buff[22],carry_buff[6]);//
fa f9(prod[2],carry_buff[8],sum_buff[0],prod_buff[4],1'b0);//
fa f10(sum_buff[7],carry_buff[9],sum_buff[1],prod_buff[7],carry_buff[8]);//
fa f11(sum_buff[8],carry_buff[10],sum_buff[2],prod_buff[10],carry_buff[9]);//
fa f12(sum_buff[9],carry_buff[11],sum_buff[3],prod_buff[12],carry_buff[10]);//
fa f13(sum_buff[10],carry_buff[12],sum_buff[4],prod_buff[23],carry_buff[11]);//
fa f14(sum_buff[11],carry_buff[13],sum_buff[5],prod_buff[24],carry_buff[12]);//
fa f15(sum_buff[12],carry_buff[14],sum_buff[6],prod_buff[25],carry_buff[13]);//
fa f16(sum_buff[13],carry_buff[15],carry_buff[7],prod_buff[26],carry_buff[14]);//
fa f17(prod[3],carry_buff[16],sum_buff[7],prod_buff[8],1'b0);//
fa f18(sum_buff[14],carry_buff[17],sum_buff[8],prod_buff[11],carry_buff[16]);//
fa f19(sum_buff[15],carry_buff[18],sum_buff[9],prod_buff[13],carry_buff[17]);//
fa f20(sum_buff[16],carry_buff[19],sum_buff[10],prod_buff[14],carry_buff[18]);//
```

8x7=56

So how many such full adders are required? 8 into 7 56 such full adders are required.

(Refer Slide Time: 30:39)

```
fa f41(prod[6],carry_buff[40],sum_buff[28],prod_buff[47],1'b0);//
fa f42(sum_buff[35],carry_buff[41],sum_buff[29],prod_buff[48],carry_buff[40]);//
fa f43(sum_buff[36],carry_buff[42],sum_buff[30],prod_buff[49],carry_buff[41]);//
fa f44(sum_buff[37],carry_buff[43],sum_buff[31],prod_buff[50],carry_buff[42]);//
fa f45(sum_buff[38],carry_buff[44],sum_buff[32],prod_buff[51],carry_buff[43]);//
fa f46(sum_buff[39],carry_buff[45],sum_buff[33],prod_buff[52],carry_buff[44]);//
fa f47(sum_buff[40],carry_buff[46],sum_buff[34],prod_buff[53],carry_buff[45]);//
fa f48(sum_buff[41],carry_buff[47],carry_buff[39],prod_buff[54],carry_buff[46]);//
fa f49(prod[7],carry_buff[48],sum_buff[35],prod_buff[55],1'b0);//
fa f50(prod[8],carry_buff[49],sum_buff[36],prod_buff[56],carry_buff[48]);
fa f51(prod[9],carry_buff[50],sum_buff[37],prod_buff[57],carry_buff[49]);
fa f52(prod[10],carry_buff[51],sum_buff[38],prod_buff[58],carry_buff[50]);
fa f53(prod[11],carry_buff[52],sum_buff[39],prod_buff[59],carry_buff[51]);
fa f54(prod[12],carry_buff[53],sum_buff[40],prod_buff[60],carry_buff[52]);
fa f55(prod[13],carry_buff[54],sum_buff[41],prod_buff[61],carry_buff[53]);
fa f56(prod[14],prod[15],carry_buff[47],prod_buff[62],carry_buff[54]);
endmodule
```

So, you can see here full adder 1 2 full adder 56. So, total 56 full adders are required and based on these values we have to assign the inputs and outputs of the full adder module that is the endmodule that is. So, this Braun multiplier will give all the product terms from P 0 to P 15. Now we have the multiplication Verilog code the full adder and then using full adder we

can construct the ripple carry adder. And then for D flip flop we have some code finally, we will implement the FIR filter.

(Refer Slide Time: 31:13)

```
\\D flip-flop
`timescale 1ns / 1ps
module DFF (input Clk, input [15:0] D,
output reg [15:0] Q ); always@
(posedge Clk) Q <= D; endmodule

\\16-bit Ripple Carry Adder (RCA)
`timescale 1ns / 1ps
module RCA_16bit(output [15:0] sum, output
cout,
input [15:0] a, b);
wire [15:1] c;
wire cout;
fa faa(sum[0], c[1], a[0], b[0], 1'b0);
fa fa[14:1](sum[14:1], c[15:2], a[14:1], b[14:1],
c[14:1]);
fa fa15(sum[15], cout, a[15], b[15],
c[15]);endmodule
```

*+ Braun multiplier*

So, this is D flip flop, so in the 3 tap FIR filter we have delays that delays can be implemented by using the D flip flop. So, this code also we have already explained in the earlier lectures, then ripple carry adder is basically full adders. So, this 16-bit ripple carry adder will be having 16 full adders.

We have discussed this ripple carry adder also in the earlier lectures. So, using these blocks now ripple carry adder, D flip flop and Braun multiplier we finally, implement the 3 tap FIR filter.

(Refer Slide Time: 31:54)

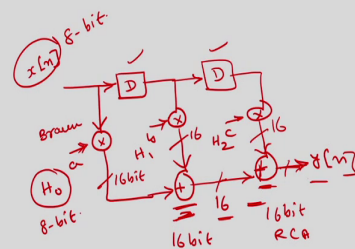


```

✓ Verilog code for 3-tap FIR filter ✓
timescale 1ns / 1ps
module fir_3tap(input Clk,input [7:0] Xin,input [7:0] H0, input [7:0] H1, input [7:0]H2,
output reg [15:0] Yout);
wire [15:0] M0,M1,M2,add_out1,add_out2;
wire [15:0] Q1,Q2;
//Braun Multiplier instantiation
✓ braun_multiplier m1(.a(Xin),.b(H2),.prod(M2));
✓ braun_multiplier m2(.a(Xin),.b(H1),.prod(M1));
✓ braun_multiplier m3(.a(Xin),.b(H0),.prod(M0));
//adders
✓ RCA_16bit r1(.sum(add_out1),.a(Q1),.b(M1));
✓ RCA_16bit r2(.sum(add_out2),.a(Q2),.b(M0));

//flip-flop instantiations (for introducing a delay).
✓ DFF dff1 (.Clk(Clk),.D(M2),.Q(Q1));
✓ DFF dff2 (.Clk(Clk),.D(add_out1),.Q(Q2));
//Assign the last adder output to final output.
always@ (posedge Clk)
Yout <= add_out2;
endmodule

```



This is the Verilog code for 3 tap FIR filter. So, we know that in 3 tap FIR filter we require 3 multiplications 2 adders and then this was the diagram that we have discussed earlier. So, this two D flip flops now using two D flip flops for this and this is input x of n and there are 3 multipliers these are 3 multipliers which multiplies with a b c. So, these 3 multipliers represented by Braun multiplier M 1 M 2 M 3.

So, we are calling this as instead of a we are calling as H 0 this up to you here I have used H 0 this is H 0, H 1, H 2 and then two adders ripple carry adders because this ripple carry adders size should be 16 bit because we are multiplying two numbers here. So, one is H 0 or a.

So, this we have taken as 8-bit, x of n we have taken as 8 bit. So, this 8 bit by 8-bit multiplication we have implemented using the Braun multiplier. So, the output of this one will be 16 bit, this is also 16 bit. So, we need a 16-bit adder.

Similarly, the output of the 16-bit adder will be again we will get 17 bit also. So, the carry bit if you neglect again 16 bit only this is also 16 bit RCA here yn also we will get 17 bit, but you are considering the 16 bit by truncating one bit. If you want you can store here 17 bits also second RCA we can implement a 17 bit RCA so, that we can increase the accuracy.

So, this is the Verilog code for 3 tap FIR filter by instantiating 3 multipliers 2 ripple carry adders 2 D flip flops. And then always at positive edge we will assign final output that is

output of this second adder output of the second adder as the final output. This is final output this is output of the second adder then the corresponding test bench for this one is this is test bench.

(Refer Slide Time: 34:50)

```
Test bench for 3-tap FIR filter

`timescale 1ns / 1ps
module tb_3tapfir;
reg Clk;
reg [7:0] Xin;
reg [7:0] H0;
reg [7:0] H1;
reg [7:0] H2;
wire [15:0] Yout;
// Instantiate the DUT
fir_3tap uut (.Clk(Clk), .Xin(Xin), .H0(H0), .H1(H1), .H2(H2), .Yout(Yout));
initial Clk = 0;
always #5 Clk = ~Clk;
//Initialize and apply the inputs.
Initial begin
H0=2'd10; H1=2'd20; H2=2'd30;
Xin = 0;
#40 Xin = 3;
#10 Xin = 1;
#10 Xin = 1;
#10 Xin = 2;
#10 Xin = 1;
#10 Xin = 4;
#10 Xin = 5;
#10 Xin = 6;
#10 Xin = 0;
end
endmodule
```

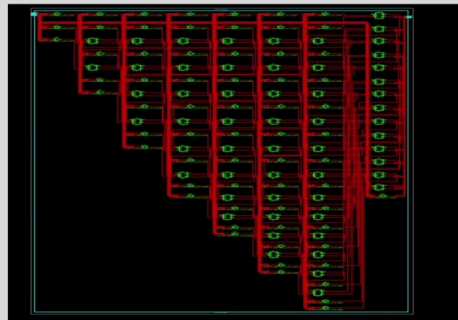
*H0 = 10  
H1 = 20  
H2 = 30*

So, we have define all the inputs as the registers, output as a wire this is module declaration then we have instantiate DUT I have used the same names for input of the test bench as well as DUT. And here I am going to complement the clock for every 5 nanoseconds. So, initially I have given this is two-digit decimal 10 this is H 0 is 10, H 1 is 20 these are decimal values H 2 is 30.

Initially I have given the input as 0 after 40 nanoseconds I will make input as 3, after another 10 nanoseconds 1, then one of these are the values of the x of n then finally, we have ended the module. So, this is basically the RTL schematic of a Braun multiplier.

(Refer Slide Time: 35:41)

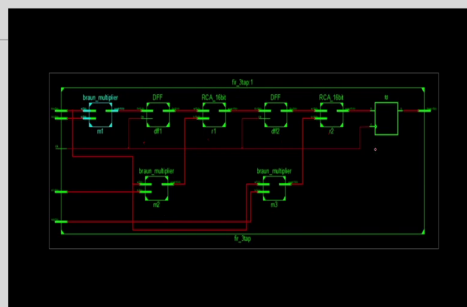
RTL schematic of braun multiplier



We can easily see the structure how this multiplication, the products, partial products will be generated then we have finally, 3 tap FIR filter.

(Refer Slide Time: 35:56)

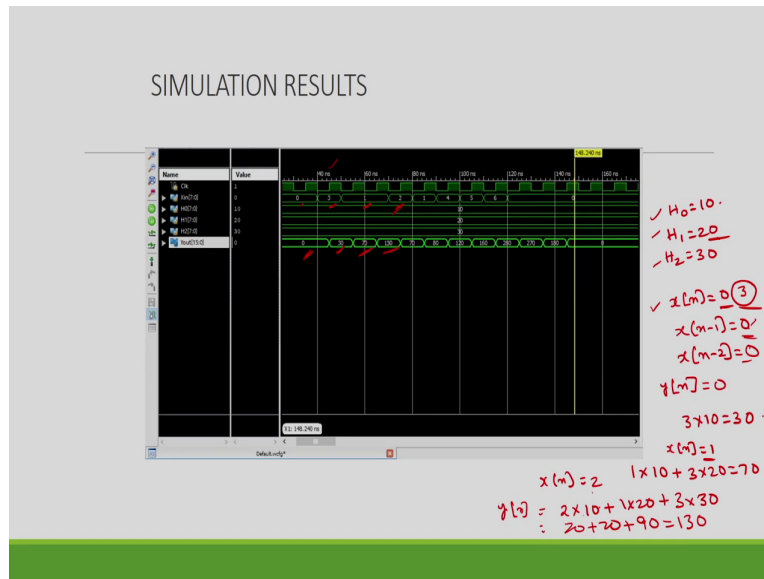
RTL schematic of 3-tap FIR filter top module



So, this is a Braun multiplier which is the previous one, this is another multiplier this is another multiplier, this is D flip flop, this is RCA and final output block this is the total 3 tap FIR filter top module.

(Refer Slide Time: 36:18)

## SIMULATION RESULTS



Then coming for the simulation results of this one, so initially we have taken this  $x$  as 0 if we take 0 whatever this  $H_0 H_1 H_2$ ,  $H_0$  we have taken as 10,  $H_1$  we have taken as 20,  $H_2$  we have taken as 30. Initially  $x$  is equal to  $x$  of  $n$  is equal to 0 if I assume that the signal is causal  $x$  of  $n$  minus 1 will be 0,  $x$  of  $n$  minus 2 is 0 because the first sample itself is this previous sample. So, I will assume that 0.

Then what is  $y$  of  $n$ ? This into this  $H_0$  into  $x$  of 0 0 plus 0 plus 0 output 0. So, after some time 10 nanosecond it seems 40 nanoseconds. So, after 40 nanoseconds, so what you have given is  $x$  of  $n$  is you have given as 3 this we have given as 3. So, this value becomes this value becomes this. So, previous values becomes 0 0. So, what is the output? So, these two are zeros. So,  $H_1$  multiplication factor is 0,  $H_2$  is also 0, but  $H_0$  is 10. So, 3 into 10 you will get 30 value. So, you can see here 30 value.

So, after some time you have given  $x$  of  $n$  as 1. So, the previous value 3 becomes  $x$  of  $n$  minus 1 ok  $x$  of  $n$  minus 1 means you have to multiply with 20. So, 1 into  $H_0$  is 10 plus the previous value is 3 into 20 which is equal to 70, you can see here. So, in after another 40 nanoseconds  $x$  of  $n$  we have given as  $x$  of  $n$  you have given as 2, this is 3, this is 1, this is 2.

So, this 2 will be now with  $H_0$ . So, 2 into 10 the previous value is 1, 1 with  $H_1$  plus 1 into 20 plus and the previous value is 3, 3 into  $H_2$  which is 30. So, this is equal to 20 plus 20 plus 90 130. We can see here this is a 2 into 10 the latest value is  $x$  of  $n$  is 2 2 into 10 20, 1 into 20

$H_1 H_0 H_1 H_2$  are 10, 20, 30 right 10, 20, 30. So, this is 1 into 20 and 3 into 30 this you should get 130. So, this is how you can verify the operation of this 3 tap FIR filter.

So, next we will discuss about the IIR filter implementation, but this time while implementing the IIR filter, we will use a signed multiplier. So, there are different types of signed multipliers. So, first I will discuss about the signed multipliers and then using signed multipliers, I will show the representation of or the implementation of IIR filter.

Thank you.