

Advanced VLSI Design
Prof. A. N. Chandorkar
Department of Electrical Engineering
Indian Institute of Technology – Bombay

Lecture – 13
Arithmetic Implementation Strategies for VLSI – Part IV

Good morning, we will continue with our effort to understand the arithmetic systems in most of the digital systems. We are so far seen the generalities about the systems, which allows you to do all arithmetic operations like adder, subtracter or multiplication or division. Today, we shall start looking into having completed the kinds of adders, which we can implement.

Please remember any of the choice of an adder circuit or adder sub system is decided by 3 important features, which are decided by VLSR requirement. Necessarily, they are speed up the operation you want, then the power which you can tolerate to consume and finally of course the area, which this particular system will occupy on silicon. Based on these 3, we are discussed which kind of adders can be used when; depending on these 2 parameters.

And we are also seen that many of the circuits, which we used in adders have multiple advantages. In some cases, some are better one; the bits are higher, some are better if they are smaller, some are better if there is only speed is the criteria or some are very, very good, if they are very low power like say current mode circuits. Now we will go to the next and the final version of our arithmetic which we say multipliers. Multiplier is most important operation in many of the real life systems.

And the way multipliers operation occurs is basically requires some kind of generating partial products and then using them, then using adders to add them out. So, let us see what is the exactly what the multipliers are? What are their options available for us to actually implement? So, this talk of multiplier, I will talk about briefly about introduction, which probably I did.

(Refer Slide Time: 02:18)

OUTLINE

- Introduction
- Arithmetic operations
- Types of Multipliers
- Individual Multiplier circuit performance
- Booth's Algorithm and implementation in a Multiplier
- Barrel shifter
- Final comments
- References

Then, we will talk about arithmetic operations, types of multipliers, individual multiplier circuit performance and the finally we look into Booths algorithm, which is the most important multiplier operation these days in most of the digital hardware. Finally, we just show you since, any operation in the multipliers requires addition and shifting, we quickly see one or two circuits of a Barrel shifter.

(Refer Slide Time: 02:55)

Types of Multipliers

- Two types of Multiplication are needed. Hence we need two types of Multipliers:

(1) Fixed point Multiplier

Integer Multiplication

needs Addition and shift operations

(2) Floating Point Multiplier

Uses Fixed bits for Signed bit,

Exponent and Mantissa. Single Precision uses

32 bits for representation of FP number which

has 1 Signed bit, 8 Exponent bits, and 23

Mantissa bits $\rightarrow (-1)^S \times F \times 2^E$

Which, allows you to data to bits to flow; shift towards either left or right and these are called Barrel shifters and finally we will may give some commands and then I may provide you a list of references. Now, types of multipliers, which are required in any digital hardware are of 2 kinds; one of course, we all know are; is called a fixed point multipliers. Among the fixed point multiplier, the most popular one is the integer multiplication.

And of course, it can also have decimals, it can have fraction and multiplication but anyway any fractional number can be handled same way as the integer numbers and therefore they only need some addition and shift operations to do a fixed point multiplication. The other possibility of course, is a floating point multiplier, in which the functions are represented as 2 to the power or 10, in case of decimal; 10 to the power numbers.

They use fix bits for sign, Exponent and Mantissa. For example, a single precision floating point number is generally represented in 32 bits, which has one signed bit, 8 exponent bits and 23 mantissa bits. So, for example, it can be written as -1 to the power S , which is the signed bit into F , which is essentially your exponent; your mantissa bits and 2 to the power E , E stand for the exponent bits.

(Refer Slide Time: 04:27)

Basics of Multiplication	
Multiplier	1 0 0 1
Multiplicand	x 1 1 0 1
Partial	1 0 0 1
Products	0 0 0 0
	1 0 0 1
	1 0 0 1
SUM	1 1 1 0 1 0 1

So, let us look into the easiest of multiplication, which we see normally in any operation in decimal and now you are here it is something a binary multiplication shown to you. Let us say, I want to multiply a number; which is a multiplier is 1 0 0, rather I should have said the other way, the upper ones are always called the multiplicand and the lower ones is multiplier, but does not matter, it is only a matter of definition.

So, if this is your multiplicand, just reverse the name so, 1 0 0 1 is 9 and 1 1 1 0 is 8 + 4 + 13, okay. So, we like to see if I multiply 9/13, what is the number I am going to get? Say, okay, so, with this on the thinking let us start 1 1 7 should be our answer. So the way we do it, we take this, multipliers first bit, multiplied to each bit of a multiplicand and write then here and then shift for the next bit multiplication and write down the; these are called partial products.

We keep writing partial products for each bit of multiplier and then finally, we add vertically to get this number 1110101 and since this number is around 117 so, we know basically what we did? We first figure out that among the multiplier bits, how many bits are you have available? Whether it is a signed bit or unsigned bit, then you have a multiplicand, how many bits it has?

(Refer Slide Time: 06:08)

Multiplication

Traditional approach involves:

1. Evaluation of partial products.
2. Accumulation of shifted partial products.

An example,

1100 = 12 ₁₀	}	Note: Binary Multiplication is equivalent to a logical AND operation. So step (1) consists of the logical ANDing of the multiplicand and the relevant multiplier bit. Each column of the partial product must be added and carry(if any) should be passed to next column.
0101 = 5 ₁₀		
1100		
0000		
1100		
0000		
0111100 = 60 ₁₀		

Classes of multipliers :

- serial,
- serial/parallel,
- parallel.

Choice depends on System Requirement of :

- Speed,**
- Throughput,**
- Numerical accuracy,**
- Area.**

And then, we start multiplying from the first multiplier LSB to the multiplicands numbers and keep generating partial products and then finally vertically we should; please remember every part, second bit we actually shift to the left and then we add the vertical lines so that, we get the sum. This is the standard multiplication even in decimal. So, we use the same thing.

So, traditionally, for example; just for the sake of this, there are 2 things we are doing in the case of multiplication. We are first evaluating the partial products and then accumulation of shifted partial products what we call sum, is then created. Now example is 1100 is 12₁₀, this is the multiplicand and 0101 is 5, if in decimal and the multiplication in decimal is 60, we know very well.

So, if I do the same thing 1* 000, then 0*0 and then 1*0 1110 and finally I start doing accumulated, please remember, every partial product we shift and then finally after completing all partial products we add vertically to create 0111100, which is nothing but

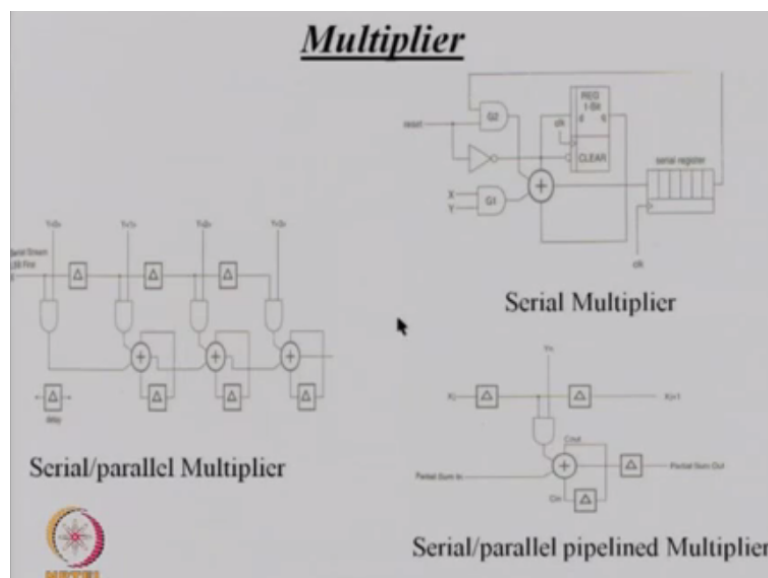
decimal 60. So, basically what we did? Binary multiplication equivalent to a logical AND operation.

One can see 1.0 what is up we did or 0.1 we did, so these are essentially a binary multiplication equivalent to a logical AND operation. So, the step one consist of logical handling of multiplicand and relative position of a multiplier bit. Each column of the partial product must be added and carry if any generated should be passed on to the next columns.

So, this is a typically what all of us been aware in case of decimal as well as in case of binary numbers. Now, before I start ahead may be I will give you a class of multipliers, which are popular in the digital hardware systems and the choice of course of any system requirement is; as I keep saying all the time, speed, throughput, area, these are of course VLSI requirements or system requirements.

And the finally, one of the major requirement in many of the digital hardware system is, how many bits you should continue to work on? Which is essentially, say a numerical accuracy, how much accurate functions are; how many the number you want to have finally. For example, 0.0000099901, is that; is what that is 0.0001 is good enough, is your choice and depending on the accuracy you provide, one may have to decide which kind of system you require.

(Refer Slide Time: 08:50)



Typically, there are 3 kinds of multipliers basic operations possible; one is called serial, the other of course is parallel and the third if not the most important being is the serial parallel.

Here are the 3 multiplier shown to you here; here is the circuit which does the serial multiplication. For example, here you have some kind of a circuit which; I already said that, the reset requirement.

This is my adder, this is my shift register or register which actually can give you the delay; 1-bit delay in this case, it may be a flip flop which runs through a clock, okay and you have a clear signal as well. So, when you get reset, this clears a flip flop. Now, the way it operates that you have 2 numbers x and y to be multiplied, so you add them out and this output, if there is a last carry coming from the; initially the carry will be 0.

So, the output of this flip flop is cleared, so it is 0 plus this and you generate the first partial product. Then, this partial product pass on to this register in the first bit. In the next bit, please remember I am actually feeding this output back to the input, okay. To this which is the adding the partial sum, the last partial product should be added to the next one by 1-bit shift. So okay, this is what is going to be done.

So, every time after one clock cycle, whatever is available in the LSB on the register here will be transferred back this to the adder. Now, the next bit will appear with the last partial bit, the new carry, which would have been generated here is now fed back last carry and the process one runs through. The advantage of serial multiplier is obvious that you have only one adder requirement even if there are, 16 bit or 32 bit or 64-bit operation to be performed.

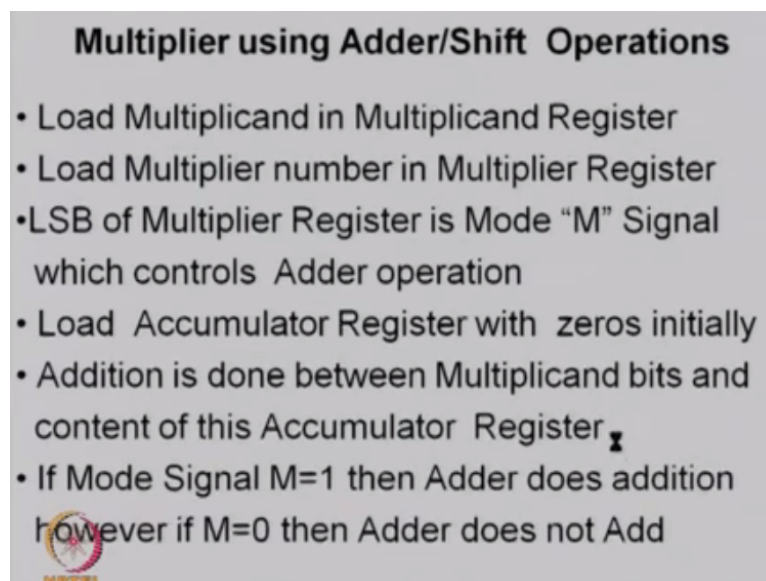
However, this advantage can be obvious that if you have a large number of bits to be multiplied obviously, one clock cycle only 1-bit operation is performed. So, if there are n bit numbers, n clock cycles will be required to generate the full multiplier output. So, it may be little slower but it is much less hardware intensive compared to others. For example, here is another one, which essentially shows a serial parallel multiplier.

Essentially, you put all the bits simultaneously together create partial products in this. All that we do is; between the 2 adders since you need a carry out, you provide 1-bit delay here which is essentially or register and that provides you the last carry and then this process keeps continuing ahead. So, it does not say that it is; it will be very, very fast, though this operation will be simultaneously done so the one AND gate delays only required.

parallel adder. So you have a 4-bit coming from here, 3-bits coming from here and yes of course are the control bit, which is coming from the multiplier register, the LSB part.

So, the idea is in this adder circuit is, add shift register multiplication is; that this mode control M signal here, if it is one here then, we say addition operation is performed by a parallel adder. If this receives 0 here, it does not do any add operation, only there will be clock cycle shift operation will be performed. So, please remember whether you do addition or you do not do addition, every clock when the new data appears, the shift has to be performed.

(Refer Slide Time: 14:24)



Multiplier using Adder/Shift Operations

- Load Multiplicand in Multiplicand Register
- Load Multiplier number in Multiplier Register
- LSB of Multiplier Register is Mode "M" Signal which controls Adder operation
- Load Accumulator Register with zeros initially
- Addition is done between Multiplicand bits and content of this Accumulator Register
- If Mode Signal $M=1$ then Adder does addition however if $M=0$ then Adder does not Add

So, the way it is the circuit shows, load multiplicand in multiplication register as we said here, load 1 0 1 multiplicand register and load multipliers number in the multiplier register. LSB of multiplier register, please remember, LSB of multiplier register is essentially is taken out as the mode, okay as the mode value M, so please remember initially this is 0, when the next clock, when 0 comes out, this will receive M0.

Next when it further shades, another 0 may come, so no add operation. When this 1 will come, there will be an add operation here. So, please remember every clock this will shift to the right and when it shifts to right, this bit will move out and the last bit, which comes out acts like the mode control value, okay. So, load accumulator register and then we initially as I said, load accumulator register is already initially cleared okay.

(Refer Slide Time: 15:36)

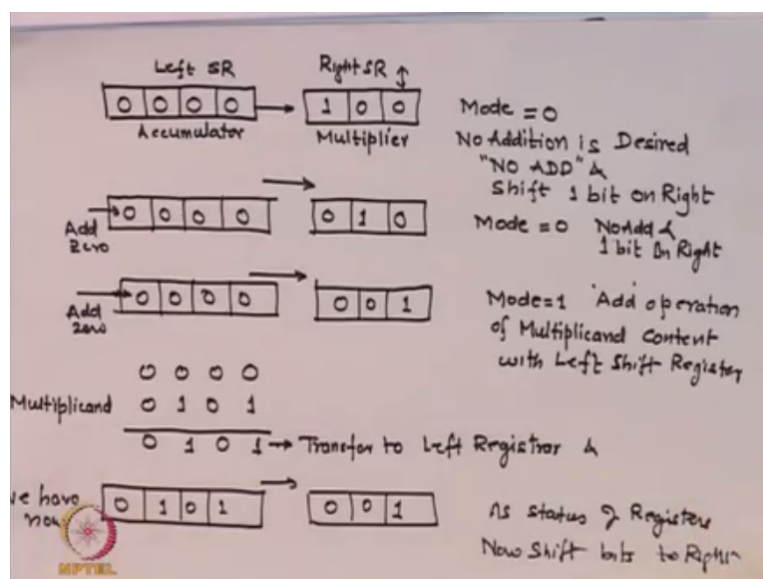
Multiplier using Adder/Shift Operations

- After every "Add" or "No Add" operation the Accumulator and Multiplier Shift register does a Right-shift operation under Clock control
- This creates change in LSB of Multiplier Register and which means one has new value of Mode control signal M
- This new Mode signal again decides whether Adder should **start** addition of bits of Accumulator register and Multiplicand register or **Not**
- This process is continued as many times as number of bits in the Multiplier register. In our example, one will have 3 shifts and Add (or No-add) operations.

And whenever the multiplication accumulation register receives any addition, it stores the new data and then it shifts the operation and gives whether M is 1 or 0. Please remember again, I will give you an example little later. After every add or no add operation, accumulator and multiplier shift register does a right shift operation under clock control because the next bits of multiplier and multiplicand will be now in question to operate.

This creates change in LSB of a multiplier register and which means that one has new value of mode control signal M. the new mode signal again decide whether adder should start addition of bits of accumulator register with multiplicand register or if M is 0, is not. This process is continuous as many times as number of bits of the multiplier register. In above example, one will have 3 shifts and add or no add operations. Because, we are using 3 bits.

(Refer Slide Time: 16:28)

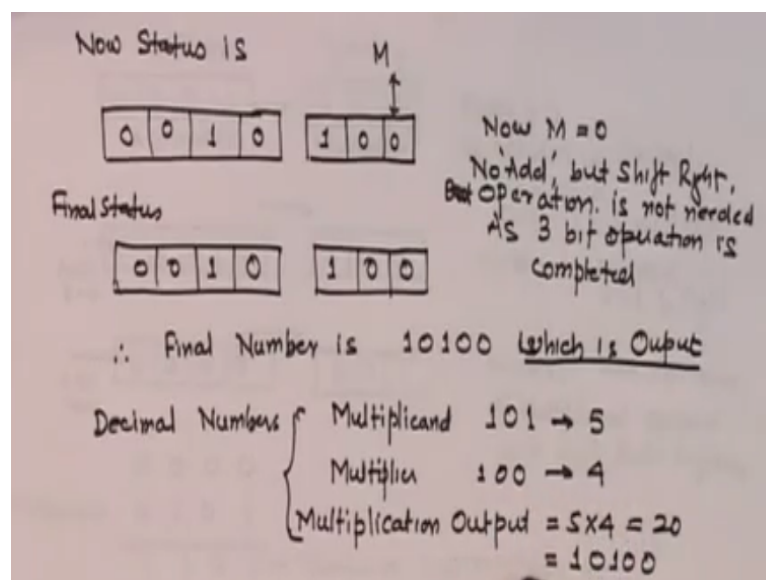


So, let us look at the example here, initially you have 1 0 0 and 1 0 1 as the operation, so we start with initial accumulator register with 0000 then, the other multiplier register has 1 0 0, since the last bit is here mode signal 0. So, we expect that the parallel operator; parallel adder does not add any do any add operation. However, as we said even if we do not do any add operation, or we do add operation, shift 1 bit on the right is necessary every clock.

So, we shift this data 0 0 0, of course we add since this will move ahead, the blank 1 is now added 0, automatically and now 1 0 0 will shift to 0 1 0 and now mode is again 0. Since, mode control is signal is still 0 and which is returned to parallel adder mode control, there is still no add operation required by now, but we still have to shift this data. So we do another shift operation once again, so you have 4 zeros again and 0 0 1 but now mode is 1.

So, since mode is 1, the mode control signal 1 will start the adder operation, so you have a multiplicand which is 1 0 1 and now that is added with this 4 bits, 0000 plus multiplicand which is 0101 and the addition of this is 0101 and that is then 0101001, but since we have already done an addition operation, this is the status of accumulator register, this is status of multiplier register.

(Refer Slide Time: 18:17)



However, every add or no add operation needs shift, so we shift the data on the right, so you get 0010100. But the next time, you see M is now 0, so no add operation is required. However, again the data will be; since they already all the 3 bits are over, no more shifts are required, so you get 0010100, so if you write this number, 10100, which is your output, if you see very clearly 1 0 1 is 5, 1 0 0 is 4 in decimal, multiplication is 20, which is 10100.

So, this is the easiest of multiplication operation which one can perform in the normal serial kind of registers which we have. The advantages I keep saying in this kind of; you can see you need a one adder and 2 registers and 1 accumulator register perform. The only thing is as many bits you have as many times you will have to shift and that means that many clock cycles you have to go through before the final operation, final result is available in some of this accumulator and multiplier register area.

(Refer Slide Time: 19:46)

Array Multiplication

$$X = \sum_{i=0}^{m-1} X_i 2^i$$

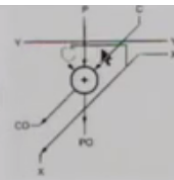
$$Y = \sum_{j=0}^{n-1} Y_j 2^j$$

$$P = X \times Y = \sum_{i=0}^{m-1} X_i 2^i \cdot \sum_{j=0}^{n-1} Y_j 2^j$$

$$= \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} (X_i Y_j) 2^{(i+j)}$$

$$= \sum_{k=0}^{m+n-1} P_k 2^k$$

Array Multiplier Cell



An $n \times n$ multiplier needs:
 $n(n-2)$ FAs,
 n HAs,
 n^2 AND gates.
 For worst case,
 $delay = (2n + 1) \tau_g$
 τ_g is the worst case adder delay

	X3	X2	X1	X0	Multiplier			
	Y3	Y2	Y1	Y0				
	X3Y0	X2Y0	X1Y0	X0Y0				
	X3Y1	X2Y1	X1Y1	X0Y1				
	X3Y2	X2Y2	X1Y2	X0Y2				
	X3Y3	X2Y3	X1Y3	X0Y3				
P7	P6	P5	P4	P3	P2	P1	P0	Product

So, it is much less hardware intensive but comparatively slow, comparatively slow. This, another operation, which we will like see, parallelly to be done, add multiplier operation; add shift operations, multiplying large number of bits. One technique, which we often use in this case is to find out, how is; this multiplication can be performed. So, let us say I have 2 numbers okay, which shows $X = X_i 2^i$ to the power i , $Y_j 2^j$ to the power j , as the Y number and the product is $X \times Y$.

I will do this again little more detail, but just to give the $X_i 2^i$ to the power i , $Y_j 2^j$ to the power j some of all the bits and if we want to find the product, then it is $X_i Y_j 2^{i+j}$ and if we put k as the product, this term $X_i Y_j$ is P_k , then $k=0$ to $m+n-1$ $P_k 2^k$ to the power k ; $i+j$ is k in our case, so in that case this is the product which you get. So, each $P_k 2^k$ to the power, 2^k to the power gives the position and P_k is the partial product.

So, if you have n by n multiplier needs n into $n-2$ full adders, so please look at the simple adder multiplier cell, you have one X ; sorry this is your X and this is your Y . so, the first

AND gate gives me partial product of XY , if you have the; if this is not the first adder, then you will require a carry. If it is this, it can be half adder because you do not need initial carry. So, the output of a AND gate, which is XY is transferred to this adder.

If it is initial, first cell, then the initial P also a partial product is 0. So, partial product is does not exist so but, if naturally, it may have for the next month, so P is also inputted here, carry is also inputted here. The partial, whatever is the partial product XY for this, we create here $X_i Y_j$, then we add, create output carry, output product and please remember this is my X and this is my Y , this is my input carry, this is my output carry.

And this is my initial product for the last case and then the new product is this one. This will then become the new input product and the next X and Y will appear and this process will continue. So, if you look at it, since the first one where you received first X and Y , do not have any carry to generate, so those places where that happens, you do not need any full adder, you may need half adder.

Please remember, half adder is a less hardware intensive, less number of gates and also is relatively faster. So, you need n into $n - 2$ FAs, full adder and half adders and obviously for each of this n into n squares, so n square and gates to create all partial products. For the worst case delay, one can say, if τ_g is the worst case adder delay for this block, when $2n + 1 \tau_g$ is called the worst case delay of this kind of multiplier, okay.

Typically, if you would 4-bit multiplier partial products if you see; let us say, I have 2 numbers multiplicand is X_0, X_1, X_2, X_3 , multiplier is Y_0, Y_1, Y_2, Y_3 represented by these 2 numbers, then we do partial product Y_0 into X_0, Y_0 into X_1, Y_0 into X_2, Y_0 into X_3 and then we repeat with Y_1 ; Y_1 into X_0 and so on and so forth and then all this partial products are added; this columns are added, X_0, Y_0 transfers here.

Then, $X_1 Y_0 + X_0 Y_1$ is transferred as some of these and we keep doing. So, here is only one term and 2 terms and 3 terms and 4 terms, 5 terms. The way we operate is whenever we get this first term, next time we actually will add this up and then we only add this one. When this happen we would have already added these 2 in the same operation, we would have added on this next operation then the finally generation 4 will do fourth operation.

(Refer Slide Time: 24:24)

Baugh-Wooley multiplier

- Algorithm for two's-complement multiplication.
- Adjusts partial products to maximize regularity of multiplication array.
- Moves partial products with negative signs to the last steps; also adds negation of partial products rather than subtracts.

So, the method is repeated product availability in earlier game, can be reused as I show product input and then new x and y can be added every now and then in this same column and new product sum can be obtained. The first and the foremost multiplier uses, which uses this algorithm we shown here, which is simple algorithm is due to credited to Baugh-Wooley multiplier, which is an algorithm for twos complement multiplication.

It adjusts partial product to maximum regularity of multiplication array, whose partial product with negative signs to the last steps and also add negation of partial product rather than subtract. Please remember no negative numbers; no subtracters are probably we want to use, so we can do it by actually creating the do some negative adds as we called, instead of using a subtracter circuit, okay.

(Refer Slide Time: 25:14)

ALGORITHM of Baugh-Wooley MULTIPLIER

X and Y are the Mutiplicand and Multiplier
2's Complementated Numbers. Then

$$X = -x_{n-1}2^{n-1} + \sum_{i=0}^{n-2} x_i 2^i$$

$$Y = -y_{n-1}2^{n-1} + \sum_{j=0}^{n-2} y_j 2^j$$

Product

$$P = X \cdot Y$$

Now, I will before we go to Baugh-Wooley circuit which is shown, which is standard array multiplier. Let me again do some little bit of (()) (25:04) again, which is not very difficult but just to see you, which is used in Baugh-Wooley multiplier. Let say, X is a multiplicand and Y is the multiplier and both numbers are represented as their complement; twos complement numbers.

Then X is equal to; can be written as; please remember how X can be now written, is $-X_{n-1} 2^{n-1} + \sum_{i=0}^{n-2} X_i 2^i$, this is the twos complement method of representing numbers. Similarly, Y can be represented as $-Y_{n-1} 2^{n-1} + \sum_{j=0}^{n-2} Y_j 2^j$ and we know the product term is $X * Y$.

(Refer Slide Time: 26:09)

$$\therefore P = X_{n-1} Y_{n-1} 2^{2n-2} + \sum_{i=0}^{n-2} \sum_{j=0}^{n-2} X_i Y_j 2^{i+j}$$

$$- X_{n-1} \sum_{j=0}^{n-2} Y_j 2^{n+j-1} - Y_{n-1} \sum_{i=0}^{n-2} X_i 2^{n+i-1}$$

To Avoid Subtractor Cell use, we represent -ve quantities as

$$- X_{n-1} \sum_{j=0}^{n-2} Y_j 2^{n+j-1} = X_{n-1} \left[-2^{2n-2} + 2^{n-1} + \sum_{j=0}^{n-2} Y_j 2^{n+j-1} \right]$$

Now, if we do this, okay if we do this, we rewrite these terms. Let us see, how we rewrite. We take the products, remember there are 2 terms in X and 2 terms in Y, so though XY will produce 4 terms. So, P is $X_{n-1} Y_{n-1} 2^{2n-2}$, then you have $i = 0$ $j = 0$, summation for i and summation for j up to $n - 2$. Then, the partial product $X_i Y_j 2^{i+j}$.

Now, you have 2 more terms, because there is a $-X_{n-1} 2^{n-1}$ and $-Y_{n-1} 2^{n-1}$. So, those terms will also get added now; multiplied so there will be 2 more addition terms. The first is $-X_{n-1} \sum_{j=0}^{n-2} Y_j 2^{n+j-1}$, that is $X * Y$, now $Y * X$ and the power will be $2n + j - 1$, $2n + i - 1$. Now this essentially means that these 2 terms are first 2 terms are going to be added.

However, the last 2 are subtracted. We will not like therefore any subtractor to be use; please remember require addition; subtractor hardware, so we do not need any subtracting operations. So, what we do is? We do negative addition as I keep saying and therefore we represent these negative numbers in this format. This is the format; please remember 2 to the power; some number when I say it actually gives you, for example let us say, let me tell you what I am trying to say?

(Refer Slide Time: 27:41)

Handwritten mathematical derivation on a whiteboard:

$$\therefore P = X_{n-1}Y_{n-1}2^{2n-2} + \sum_{i=0}^{n-1} \sum_{j=0}^{n-2} x_i y_j 2^{i+j}$$

$$- X_{n-1} \sum_{j=0}^{n-2} y_j 2^{n+j-1} - Y_{n-1} \sum_{i=0}^{n-2} x_i 2^{n+i-1}$$

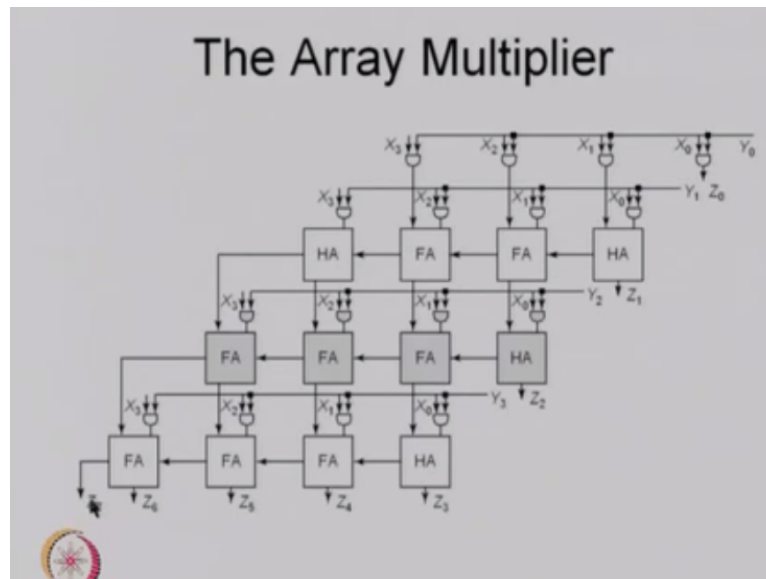
To Avoid Subtractor Cell use, we represent -ve quantities as

$$- X_{n-1} \sum_{j=0}^{n-2} y_j 2^{n+j-1} = X_{n-1} \left[-2^{2n-2} + 2^{n-1} + \sum_{j=0}^{n-2} y_j 2^{n+j-1} \right]$$

If I have a number 10110 what I am essentially saying is, 2 to the power 0 into 1 sorry 0 into 1 + 1 * 2 to the power 1 + 1 * 2 to the power 2 + 0 * 2 to the power 3 + 1 * 2 to the power 4. So, every bit position here, here, here essentially gives me the 2 to the power coefficient there, so if I say, I am here and I want to subtract something or this, I can move my by this position, and if I do should the position I am actually doing the essential equivalent of a subtractions.

So, this method of actually doing subtractions through a negative number can be represented as $-X_{n-1}Y_{n-1}2^{n+j-1}$ and $-Y_{n-1}X_{n-1}2^{n+i-1}$. Please remember this is 2 to the power $n-1$ - of 2 to the power $n-2$; $j=0$ to $n-2$. Similarly, I can write for Y; - Y term, which I said the $-Y$ and -1 and -2 can be rewritten in the same form as $Y_{n-1}2^{2n-2} + 2^{n-1} + \sum_{j=0}^{n-2} y_j 2^{n+j-1}$. So, I can write these 2 terms in this format.

(Refer Slide Time: 29:47)



I have these 2 terms are anyway positive terms and therefore now we are in an interesting situation that we can then only need positive operations or add operations in this case and 2 to the power numbers essentially the shift operation, shifting this is the shift operation. Typically, array multiplier shown here if you see here is typically array multiplier shown here, these are all Y_0 s, Y_1 s, Y_2 s, Y_3 s, okay.

Then, each vertical line is X , say this is X_0 , this is X_1 , which is not shown in each gate receives X_0 X_1 X_2 X_3 . Similarly, you need X_2 again you need X_2 , you will need X_3 , X_3 so, this X is essentially travelling diagonally each X is diagonally, whereas Y ; we have taken as horizontal lines. So the; what it does this, the first partial product is $X_0 Y_0$ which is your Z_0 . Then the next partial term can be created by X_0 times Y .

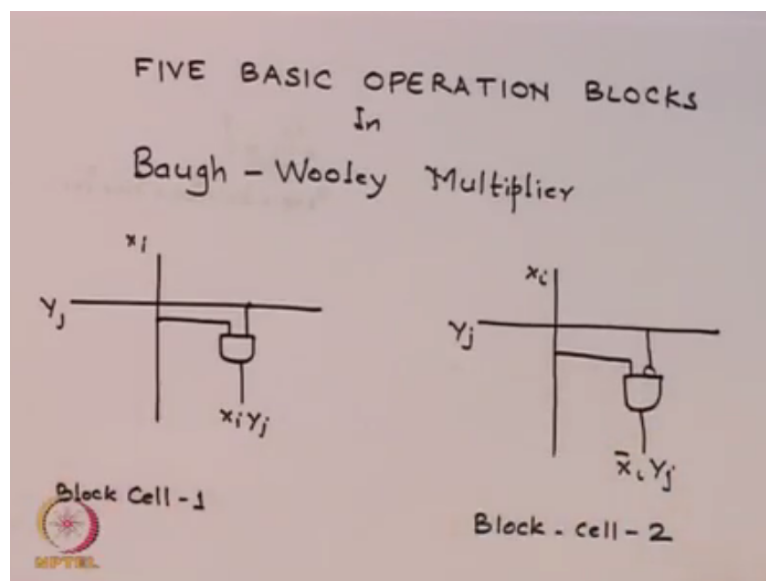
But you need now this addition with $X_1 Y_0$, so this is $X_1 Y_0$ is coming from here, $X_0 Y_1$ coming from here and since it is the first time you are doing an addition operation that is no carry available here for a half adder is good enough and if that happens, the together half adder creates Z_1 , but now it generates carry. For the next of this, you get X_2 same way and now this whichever you carry you are generated with this theme numbers plus this $X_0 Y_2$ numbers can be again it will not have any carry.

Because, this is the first time appearing. So, you need a half adder. So, one can see from here the last X , wherever X_0 is appearing, you actually need half adders but whenever $X_1 Y_1$ or $X_2 Y_2$ ahead you will require full adders for those all operation. Here again you see, there is

no full adder requirement because there is no additional X or Y coming from this side, so no additional carry because already one, only there is this term is not occurring here.

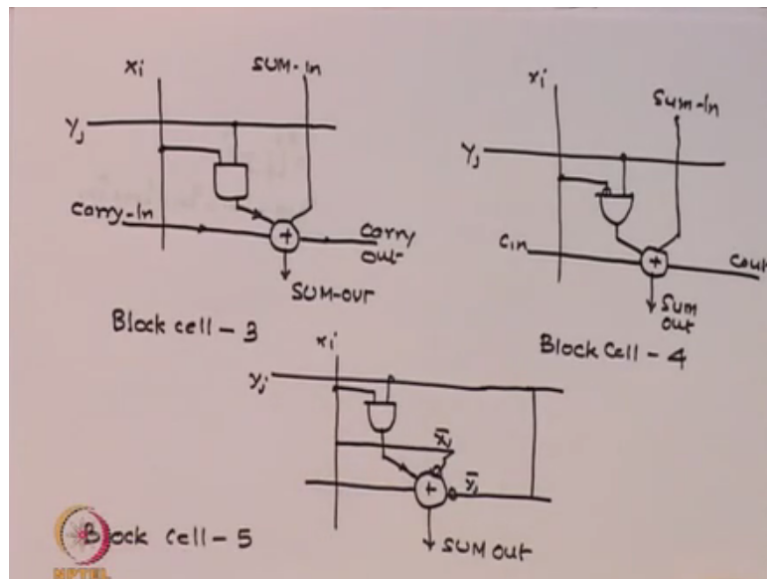
So, you need does not need; you do not need carry inputs here, so you need half adders here. So, typically what you are doing is successive creation of products and some through full adders is transferred to the next bit, so you can see this is the total addition going through this, whatever is added here is now added with this. Whatever is added here is now added with this with half adder you get this Z3, by same logic, you get addition of these the vertical lines and you get Z0 Z1 Z2 Z3 Z4 Z5 Z6, okay.

(Refer Slide Time: 32:42)



Now, of course, the last carry, which will generate will be your Z7. Now, if you see the kind of operation you may have to perform for subtractor or minus values, can be shown through there. There are 5 kinds of cells or block cells, which you use in a Baugh-Wooley multiplier, the first one of course is the generation of $X_i Y_j$ term, this is X_i , this is Y_j , simple AND gates. This is block cell 1, then you may require a $\bar{X}_i Y_j$.

(Refer Slide Time: 33:15)



This is subtraction kind of requirements if you see, then you may require inverter here, okay. This is block cell 2, then if you see this another cell you may require is $X_i Y_j$, you create an $X_i Y_j$ term, you have the last sum which is coming, which is what full adder will give. It may receive a carry, may generate a carry and the final sum out, which what that full adder circuit which you are seeing there, you can see.

The other one is you may require X bar or Y bar kind of things, this is X bar Y and same operation as this you require and finally you may require some kind of an XOR equivalent, for example in the final adders, this is nothing but $X_i Y_j$ the complemented is X_i bar, complement of this is Y_j bar, so this is X bar, $X_i Y_j + X$ bar Y_j bar + carry; this kind of operation can then lead to an XOR or XNOT kind of operations.

(Refer Slide Time: 34:33)

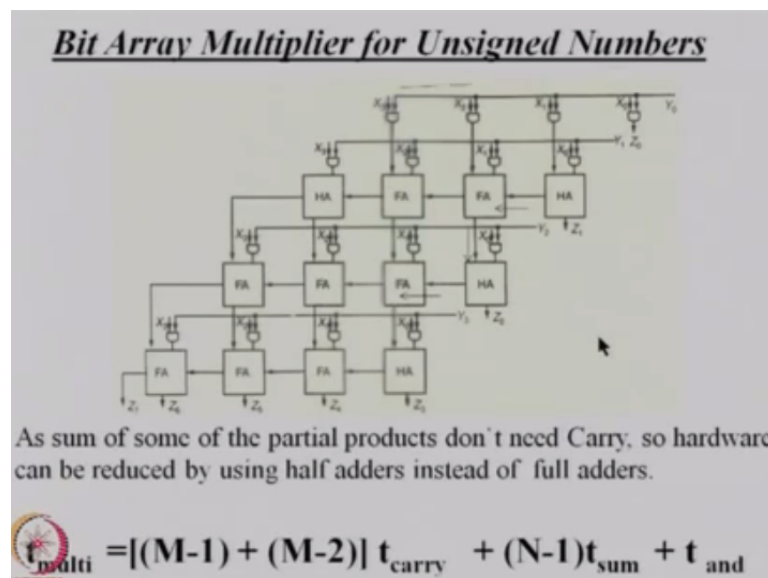
In this Multiplier we need

1. 2's Complement Generator
2. AND GATES to get Partial Products
3. Full Adders

To Save Area and also to Improve Speed $n \times m$ bit Multiplier is arranged in an ARRAY-FORM.

So, these are the 4 blocks which are normally you will find in a Baugh-Wooley multipliers. If we see carefully these 4 figures, 5 figures, you can see from here, in this multiplier, we need two's complement generator, AND gates to get partial products and full adders do additions. So, these are the only 3 gates which we probably will; 3 kinds of system, blocks we will require to do a multiplication.

(Refer Slide Time: 35:11)



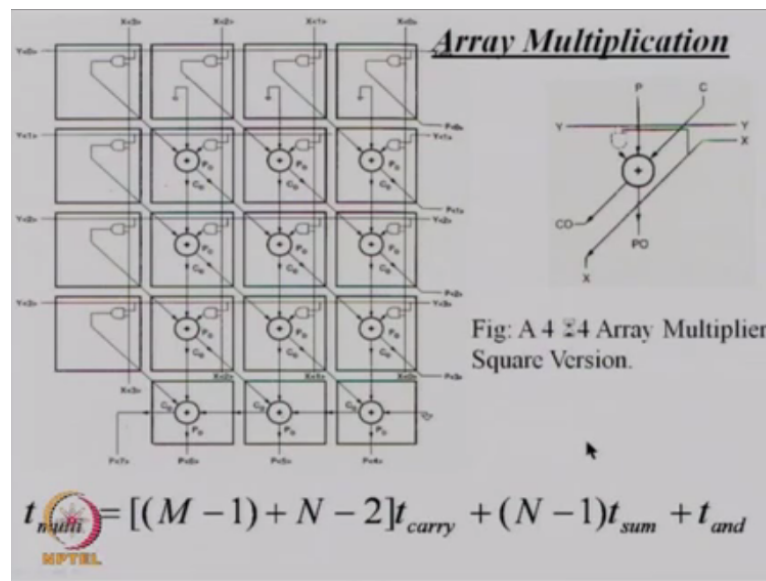
To save an area and also to improve speed an n/m bit multiplier is always arranged in an array which is what the slide is showing. You can see this is XY array has been done. However, a better arrangement is also possible okay and which is shown in my next slide. You can see from here, actually I will say exactly the same it is not different. So, only thing why I am showing you is the place what is the kind of delay you are going to get.

Let us say, you have N/M , $N/$ this, so this is same as $X_0 Y_0$ then $X_1 Y_1$ $X_0 Y_0$ kind of thing we are doing as we did. So, the path is this, this, this, this, this, and this. Please remember the path is this, this, this, this, this, and this. I remember, I please tell me this is the critical path of the circuit, okay. So, if you look at the delay associated with the critical path; critical path I repeat is coming like this.

So, you require all 4 adders okay, so you have a $N-1$ kind of full adders okay only some operations will be required so, you have $N-1$ t_{sum} , because please remember when we are performing this operation, the other operations are simultaneously done in the earlier cycles, so you do not need to know this. So, you have only $N-1$ operation, every; since X and Y are created simultaneously, so you have only one AND gate delay.

And then you have a carry path; this is what I was trying to show a carry path. So, you have $M - 1 + M - 2$ carry for X and Y, I mean for the; this is one is vertical down, one is horizontally down. So, it is the delay that is associated with $M - 1 + M$; this is of course M cross N carry. So, this is the net multiplier delay. I may tell you again the delay is essentially from this path, this is the time delay, I evaluate.

(Refer Slide Time: 37:22)



Since as some of the partial product do not need to carry so, hardware can be reduce as I keep saying only a half adders are sufficient. A better arrangement for the same, which I shown here; you know this is my X, which is travelling vertically down, this is my Y, this is my AND gates, which receives X and Y here at this AND gate, this is my adder, which can be mostly full adder in case the carry appears otherwise half adders if there is no carry appears.

Now, this is my output carry, this is my initial partials products which is then added to next partial products and keep generating the new. So, if I keep array of 4/4 in this fashion, you can see from here, this is my X, sorry this is how diagonally I am crossing X and Y, okay and please remember these are my X and these are my Y, slightly shown in a better fashion and the product is travelling.

In a; instead of product travelling vertically, actually my; this circuit is product is travelling in the AND gate, which is going to the adder is essentially looked into this direction. So, this is XY with the last this P comes, initially of course this is 0, so it keeps doing. So, if you can see using this kind of arrangement which is shown here, the multiplier; which is same, if it is M

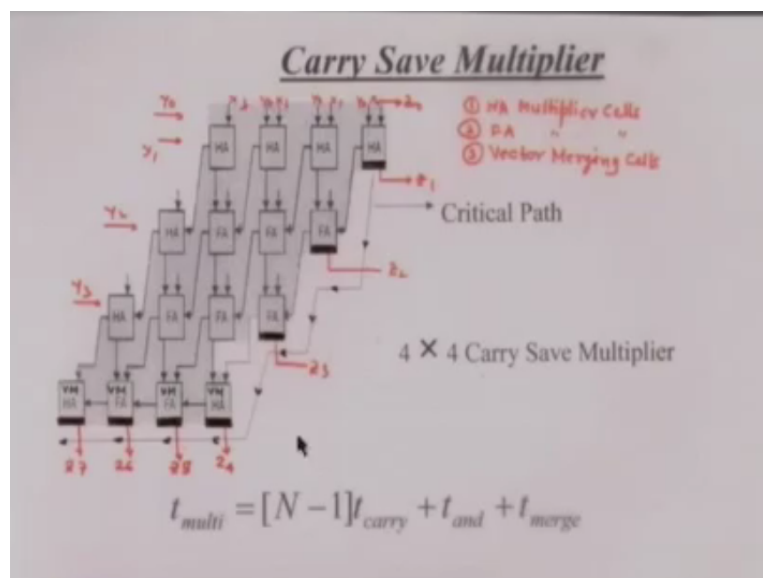
$(N-1)t_{\text{carry}} + t_{\text{sum}} + t_{\text{AND}}$ is the net delay which is; this is same as what I earlier shown.

And one can see from here that delay can be minimised; delay will obviously increase if the array size is larger that means if your 8 bit by 8 bit multiplications or 16 bit by 16 bit multiplications, the multiplied time will keep increasing. So, the adder part does not really increase; adder part does not really add; please remember this is $M + N$ kind of things but this is only n kind of things.

In general, sum time is not that high compared to this product of this, so this essentially dominant over this term and time of course is very, very invisible. So, we have seen earlier in our carry save operations earlier in an adder thing. We know the carry save adder has the biggest advantage we know about it, it is little faster for the simple reason that carry save adder allows you to do 3-bits addition first and generating 2 terms C and S .

The S essentially, is the sum of those bits X Y and j for example and without taking carry into consideration and the carry term C is generated without taking the sum into consideration and then we add C and S with the any other initial carry it you had with a simple CLA or any normal full adder to generate your carry save operations, full adder. Now, and we say since it does not propagate carry nor it has to look ahead the carry, it has the; it is the fastest adder.

(Refer Slide Time: 40:38)



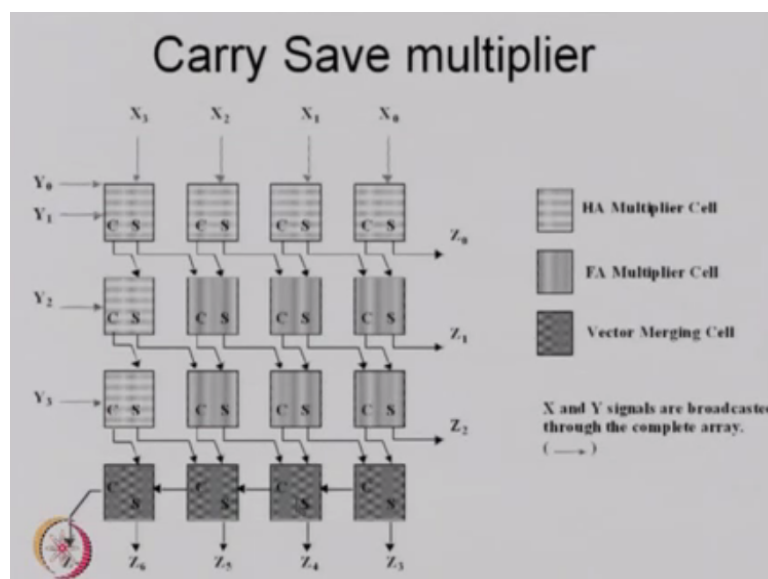
So, the same circuit, which we discuss earlier instead of using the normal adder, we can use at least those place remember I already shown you the critical path in my circuit there, those

full adders which are in the critical path to derive the time at least those should be utilised using carry save adder. So, you can see these are the carry save adders, these are called vector merge.

For the simple reason that, here the actually you are adding, generating; of course please remember first one will can be always half adder if there is no carry generation. So, the way we operate here in the carry save adder, twos addition and then the third is added here, twos addition and then the third is added here, we keep doing this operation 33232 operations and can generate $Z_1 Z_2 Z_3$ this is my X and these are my Y , which are fed here.

Now, this is 4/4 carry; please remember the critical path is all that matters to me for the worse case delays and therefore the multiplier delays $N-1$ t carry, okay because there is no carry propagation except for the actual carry which are required for the next stage; one carry only. You can see $4 + 3, 7$; this is $4 3$ t carry are only required plus this; only $3t$ carries plus merge time; merge time is essentially in which all Z_s are parallely available.

(Refer Slide Time: 42:41)

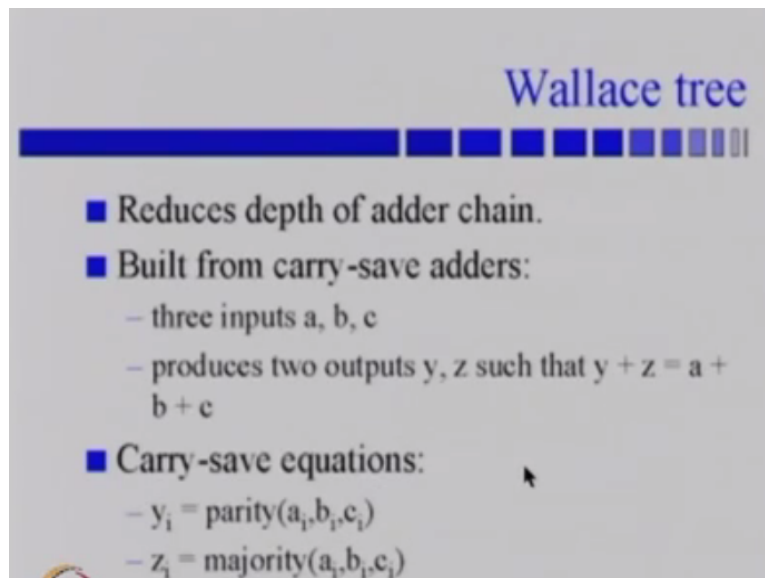


Please remember these are parallely available to you, okay, so you have t merge and of course AND gate to generate XY terms. So, a carry save multiplier has half a multiplier cell full adder multiplier cell and there are among the half adder full adder, some of them are carry saves and the others are; these are called vector merging. The same figure can be little better way shown here, these are X_s , these are Y_s , okay.

And you can see 3 terms; $X_3 Y_0 Y_1$ creates CNS. The S is now transferred to the next cell with the $X_2 Y_0 Y_1$ is transferred here and then it; this is; these 2 numbers and then generate another 2 numbers and diagonally passes back there and since it simultaneously passes from this side, from this side, from this side, this is called vector merging and therefore the delay essentially is what I have just now discuss.

So, the half and multiplier cell they are full multiplier cell, they are vector merging cell. Please remember you can use all carry save adders, which may do but some of them in the last circuit if you see, they need not even carry save because any way I am sorry; they are all carry save, they need; they are not the ones in the critical path. Because they are simultaneous; I am very sorry what I said?

(Refer Slide Time: 44:00)

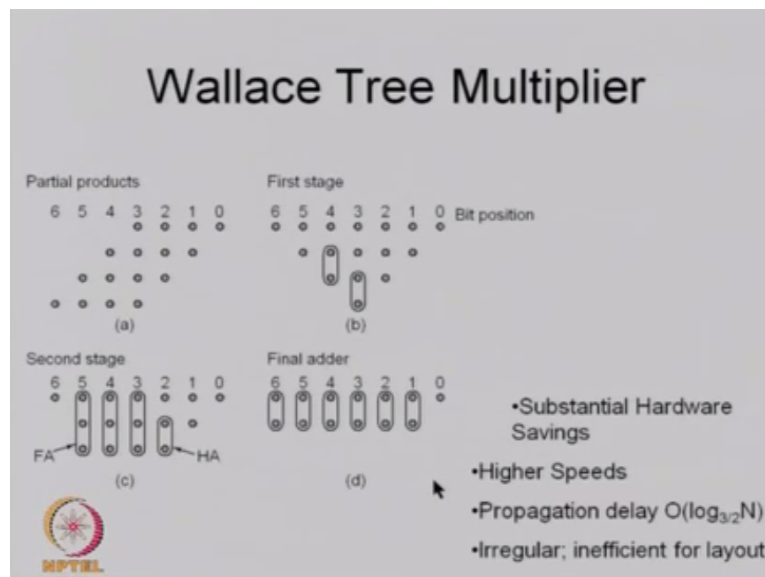


Wallace tree

- Reduces depth of adder chain.
- Built from carry-save adders:
 - three inputs a, b, c
 - produces two outputs y, z such that $y + z = a + b + c$
- Carry-save equations:
 - $y_i = \text{parity}(a_i, b_i, c_i)$
 - $z_i = \text{majority}(a_i, b_i, c_i)$

These are full adders, same carry save, but these are the only ones, which will transmit the data then therefore in the critical path. So, only the critical path delays are after relevance which this receives this. The other possibility of a generating a multiplication is using what is called a Wallace tree. We know when a tree operations, any tree operations reduces the depth of the adder chain.

(Refer Slide Time: 44:32)



We still use carry save adders, so you have 3 input a b c, which produces 2 outputs y and z. In the last case and we create these terms. We know this we already done this carry save operations. How do we do a Wallace tree multiplication or this? So the first thing we do is? In your multiplier, let us say these are the positions, 0 1 2 3 4 5 6 bits, this is the first partial product terms $X_0 Y_0$ $X_0 Y_1$ kind of things.

Then this is again $X_0 Y_1$ X_1 kind of thing. There are 4/4 product I am showing you. Now, what you do is? We know this is the operation; these are of course zeros here, okay. So, we instead of writing in this format, we write for each position, 1, 0 to 6, we write bit position, we write whether 1 or 0 exist, okay. So, for example these 4, 4; first 4 will exist because of this but the next 4 exist from 1 to 5 okay, 1 to 4 so it is 1 to 4, so this is 1 to 4 okay.

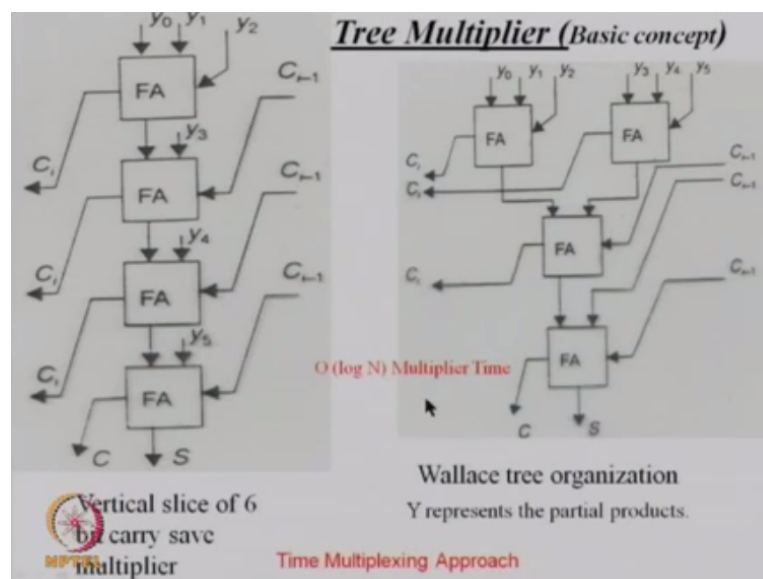
Then you can see from here, the next is from 2 to 5, so from 2 to 5 and then from it is 3 to 6, so 3 to 6. So, the first we write for each of them whether; see if in this column this has to be added, this has to be added, this has to be added. So, we now start looking for actual additions. We say okay in the first bit position only 0. Next you have only 2, then in the third you have 3 okay. In the fourth, you have 4, okay.

In the fourth you have again 3, in the fifth you have 2 and the sixth you have 1, okay, we are just inverted it nothing big, same thing rewritten in this form. Now, what we do in the; after we put this is called the first stage operation, to create the tree, the next stage is; in the first stage itself, circle the last ones. For example, for the third and fourth, you circle the last 2, so if that means we do this addition it will create only one numbers here, okay.

And then you will have 3, 3; 2 kind of thing can be operation can be created, okay. Then it will be 3 3 3, all three operations and then we can see in the next operation if we see therefore, if 1, 2 of course this is also 2 operations are could we have been directly here, but so we look into second position, there will 3, so you create for the 2, then you will create 1 out of that and then $1 + 1$ is 2.

This we already created $2 + 1$, $2 + 1$, $2 + 1$, this, so if you rewrite this, it becomes of course this is 0 we take, so $2 2 2 2 2$, since $2 2 2$ operation is very simple to add, so you have partially doing your summing here and bringing at then only some of the 2. By using this kind of thing we can reduce substantially hardware can be saved, substantial saving in the hardware, the operation will be very high speed, we will see this.

(Refer Slide Time: 48:07)



And the delay will be now $\log_3 N$. Of course, since it is the kind of operations you are performing the; it will be a irregular structure not in array, universal structure and therefore many times the lay out becomes vary and efficient. Here is the tree multiplier basic concept. So, basically you have a carry save multiplier shown here, you have a full adder, this is Y_0 Y_1 and the next is Y_2 creates some C and this and add and the next is carry.

The last similar adder must have come, add this now they have 3 bits create this and this 2 and keep create in a vertical diagonal directions. Here is something which better looking figure, which is same as this but lightly better. You have Y_0 Y_1 fed to Y_2 creates C_i , okay

and then this is your S term, which is nothing but sum of Y0 Y1 Y2, which is then fed with the carry generated out of this, okay.

Full adder sum, partial sum and you add with this to create the new carry and new sum. The next carry is now fed here and create new carry and new sum. So, it is log N multiplier times order of log N is the delay here; Y represent the partial products and X represent this. So, essentially till you are doing time multiplier, so you are trying to save some operations because they are carry save operations, no carries are required in the self-operations.

And these are transferred to the only next stages and the next sum simultaneously made available to you. So, this means using this tree multiplier concept, one can save the time or that is high speed. The number of adders now required will be only 4 to do these operations and 6 bit operations as we see and this is therefore less in hardware high speed. But you can see if I lay out this block, it will be very difficult.


(Refer Slide Time: 50:06)

Booth's Algorithm

Used for Multiplication in Integer Arithmetic
It uses Radix 4 multiplication which reduces No. of adders and hence speeds
A Booth recorded Multiplier examines three Bits of Multiplicand at a time to determine, whether to add 0, -1, +1, 2 or -2 of that rank of the multiplicand.
Consider 16 Bit 2's complement No. Multiplier Multiplicand 'x' can be written as :

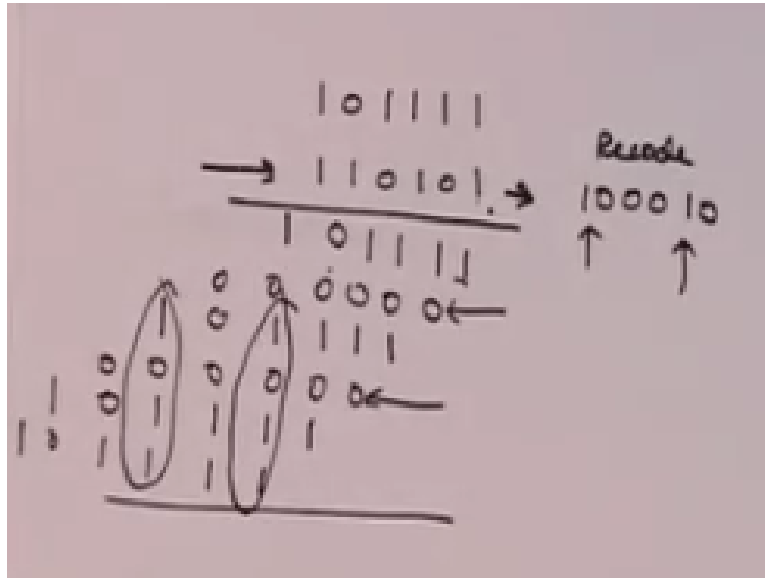
$$x = \sum_{i=1}^{15} x_i 2^{-i} - x_0 2^0$$

$$x = \sum_{i=1}^8 x_{2i-1} 2^{-2i+1} + \sum_{i=1}^7 x_{2i} 2^{-2i+1} - 2 \sum_{i=1}^7 x_{2i} 2^{-2i-1} - x_0 2^0$$

$$x = [x_{2i-1} + x_{2i} - 2x_{2(i-1)}] 2^{-2i+1}$$


Now, if you look at your multiplier any time very carefully, okay. What is the problem with normal multipliers? If you see a normal multiplier, you have partial sums, you are going to create depending on your multiplier and multiplicand kind of numbers, it can be fine that there may be large number of ones available to you and if you have ones those many mean terms usually partial products will be available.

(Refer Slide Time: 50:47)



For example, I may show you what I am saying? If I have a number; 1 0 1 1 1 1 and I multiply it by 1 1 0 1 0 1, you can see from here, I can create this so many terms so, 1 1 1 1 0 1, then 0 0 0 0 0 0, then the next one is 1 1 1 1 0 1 then again 0 0 0 0 0 0, again 1 1 1 1 0 1, and finally again 1 1 1 1 0 1, so if you see an operation except for these 2, which is shown here, every other partial product you are creating, larger the number of partial products you have; for example, in this each of this for example here you have 3 ones, you have 3 ones okay.

So, the larger the ones availability in your partial products, larger will be the operation of addition and therefore even if you use carry save, you could require larger times. However, if I actually convert into say let us say something like this for the sake of completeness, then I have only 2 terms associated with this multiplier one ones, the rest terms I have need not even write because these are the terms any way going to be zeros.

So, I will only do addition of 2 terms, which is very fast. This is essentially called a multiplier term is coded or recoded, this is your twos complement number let us say and you recoded into a format, which allows you to deduce the number of ones and if that happens the number of adders which reduce the number of additions and therefore it will increase the speed enormously.

(Refer Slide Time: 53:13)

In Booth's MULTIPLIER we recode
2's Complement Nos.
Since we use Binary Nos, we observe
that :
j-long Sequence of 1's is equivalent to
 $\Rightarrow (j-1)$ long Sequence of 0's.
Replacement of 1's by 0's reduce Partial
Product Terms.

So, one of the major criteria of any VLSI chip, as we discussed here also is to show that; now here is before I go the actual Booths algorithm, I may just show you what I am really talking. In Booths multiplier, we recode twos complement number; since we use binary number, we observed that j long sequence of ones is equivalent to $j - 1$ long sequence of zeros. So, replacement of ones by zeros reduces the partial product terms, this is what is called recoding.

Is that word clear? I; the sequence of ones can be converted of sequence of larger number of zeros and therefore reducing the partial products and since the partial product terms are smaller, then time taken to add them will be also smaller. So, this is basically the principle or basically the need of a recoding and that is what Booth feedback in 1900, odd years as first time suggested that this is mathematically possible.

Because of the; this law that j long sequence of ones can be equivalent to $j - 1$ long sequence of zeros and using this theorem, Booth has arrived at an algorithm for additions. What is this? It say that Booth recorded multiplier, recoded sorry; it is not recorded, it is a recoded multiplier examine 3 bits of; this is for the Radix 4, this is essentially modified kind of thing but let us see what I am talking about.

A Booth recoded multiplier examines 3 bits of multiplicand and time to determine whether to add 0, -1, +1, 2 or -2 of the rank of the multiplicand. Before we go to this, maybe I will actually discuss the same issue little later, but let us look at the kind of things we do here.

Before we; we will come back to this expression little later; this one but okay just look at the number.

X is can be written as $i = i_2$, let us I am using 16 bit numbers, $15, x_i 2^{i-1} - x_0$, so this is very important. What I wrote is first to, 0 of course I have taken out okay. So, it is $x_i 2$ to the power $-i - x_0 2$ to the power 0, okay. So, if you have this you can remove this term from this, so you have 16 bit numbers. This can be further written as $i = 1, 2$ this 8, now I divide into this 877 kind of thing.

So, $x_{2i-1} 2$ to the power $-i + 1, i_1$ to 7, like this plus minus 2 to I minus 1 and again x series. If we collect these terms, then x can be written as $x_{2i-1} 2, x_{2i}$, and $-2x_{i-1}$ into 2 with the power $-2y + 1$. Now, this is essentially what I am going to do in my evaluations that any number has 3-bit equivalence $x_{2i-1} 2, x_{2i} x_{2i-1}$, minus of that of course with the minus 2 signs which is equivalent of the $x_i 2$ to the power.

(Refer Slide Time: 56:44)

Basic of Recoding

- Consider a positive multiplier consisting of a block of 1s surrounded by 0s. For example, 00111110. The product is given by :

$$M \times "00111110" = M \times (2^5 + 2^4 + 2^3 + 2^2 + 2^1) = M \times 62$$
 where M is the multiplicand. The number of operations can be reduced to two by rewriting the same as

$$M \times "01000010" = M \times (2^6 - 2^1) = M \times 62.$$

Now, this is what essentially, we know how can we represent the x numbers and before we do ahead, let me tell you how do I do the recoding? Consider a positive multiplier consisting of a block of one surrounded by zeros, so it is 00111110, the product is given by M; M is the multiplier you want; multiplicand you have and this is your multiplier which can be written as $M * 2$ to the power of 5, 2 to the power 4, 2 to the power 3, 2 to the power 2, 2 to the power 1, so this is 62 M, where M of course further multiplicand.

The number of operations can be reduced to 2 by just simply rewriting it, 2 to the power 6 is 62 – 2 to the power 1 is 2, which is 64 – 2 which is 62. So now, what operations I am performing? Right now I was performing 1 2 3 4 5 operations okay. I can now reduce to only 2 operations; this is essentially the basic thinking in recoding. Please remember this number if I recode in this format, this 11110, in this format, you have larger number of zeros minus this.

Please remember 2 to the power 6 will have larger zeros, because 10000 kind of term, this 2 will be of course -1, 0 okay. Now, you can see from here, this number has most of the zeros okay. So, you can see it is only the positional advantage you got it, you only have to do now 2 operations because most of them are 1, only these 2 operations I may have to perform to actually perform this whole multiplication.

(Refer Slide Time: 58:44)

Booth's Algorithm Continued.....

Hence Multiplication $y \times x$ can be written as

$$x \cdot y = \sum_{i=1}^n [x_{2i-1} + x_{2i} - 2x_{2(i-1)}] y^{-2i+1}$$

[] takes values as 0, ± 1 or ± 2 . Hence No. of partial products generated are reduced and they are simple multiple of input operand (-2y, -y, 0, y, 2y)

A partial product of type $\pm 2y$ is achieved with a left or right shift

Booth's Recoding values

X_{i-1}	X_i	X_{i+1}	OPERATION	NEG	ZERO	TWO
0	0	0	add 0	1	1	0
0	0	1	add 2	0	0	1
0	1	0	sub 1	1	0	0
0	1	1	add 1	0	0	0
1	0	0	sub 1	1	0	0
1	0	1	add 1	0	0	0
1	1	0	sub 2	1	0	0

This is essentially recoding this into this format, okay. Continuing with our Booths multiplier operation, it takes values of 0, plus minus 1, -2 as we just now said and number of partial product generated are reduced and they are simple multiples of input operand -2y -y0 y 2y, if y is the multiplier and x is the multiplicand. Please remember this is the table, which will right now I do not want to discuss this table, I am coming back to this table again.

(Refer Slide Time: 59:08)

Booth Multiplier

Here:

- Encoding scheme reduces number of stages in multiplication.
- It performs two bits of multiplication at once—requires half the stages.
- Each stage is slightly more complex than simple multiplier, but adder/subtractor is almost as small/fast as adder.

In a Booth multiplier, encoding scheme reduces number of stages in multiplication. It performs 2 bits of multiplication at once requires half the stages, each stage is slightly more complex than the simple multiplier but adder and subtractor is almost as small and as fast, as normal adders, okay just to give you again the same twos complement number can minus can be represent like this.

Rewrite 2 to the power something is $3 - 2$ to the power a , therefore $-y$ can be written in this format, then we already discuss this earlier, it is same representation. Consider first 2 term by looking at the 3 bits of y , we can determine whether to add x or $2x$ to the partial product and I will give you an example what I meant, okay. Even before this, let me say; tell you what I really code okay.

(Refer Slide Time: 01:00:06)

Booth's Algorithm

- Booth's algorithm involves repeatedly adding one of two predetermined values of **A** and **S** to a product **P**,
- & then performing a rightward arithmetic shift on **P**.
- Let **m** and **r** be the multiplicand and multiplier, respectively; and let x and y represent the number of bits in **m** and **r**.

The simple Booth, before go to the modified one which I started, let me first discuss the Booths algorithm, which is the simple Booth algorithm; Booth algorithm involves repeated adding one or 2 pre written values of A and S to a product P and then performing rightward arithmetic shift on P.

(Refer Slide Time: 01:00:30)

Procedure in Algorithm

- Determine the values of A and S, and the initial value of P. All of these numbers should have a **length equal to $(x + y + 1)$** .
 - A: Fill the most significant (leftmost) bits with the value of **m**. Fill the remaining **$(y + 1)$** bits with zeros.
 - S: Fill the most significant bits with the value of **$(-m)$** in two's complement notation. Fill the remaining **$(y + 1)$** bits with zeros.
 - P: Fill the most significant **x** bits with zeros. To the right of this, append the value of **r**. Fill the least significant (rightmost) bit with a zero.

Let us say, you have m and r be the multiplicand and multiplier and x and y represent number of bits in m and r. So, the algorithm says, determine the values of A and S and the initial value of P, all of these numbers should have a length equal to $x + y + 1$. Now, A; fill the most; I will come to an example, you will see it, fill the most significant bits with the values of m, fill the remaining $y + 1$ with zeros.

For the S, fill the most significant bits with the value of minus m in twos complement notation and fill the remaining $y + 1$ with zeros. For the P, fill the most significant x bit with zeros to the right of this append the value of r, fill the least significant right most bit with the zero. I will give you an example and I think that I will be clear to you but before that what is the operation to be perform?

(Refer Slide Time: 01:01:15)

Procedure in Algorithm(cont.)

- Determine the two least significant (rightmost) bits of P .
 - If they are 01,
 - Find the value of $P + A$. Ignore any overflow.
 - If they are 10,
 - Find the value of $P + S$. Ignore any overflow.
 - If they are 00,
 - Do nothing. Use P directly in the next step.
 - If they are 11,
 - Do nothing. Use P directly in the next step.



Determine the 2 least significant bit of P , if they are 0, 1 then do this operation $P + A$, and ignore always overflow. If these 2 last bits are 1, 0, then do operation $P + S$, again ignore overflow. If they are 0, 0, do nothing, use P directly in the next step and if they are 1, 1 again do nothing P directly in the next step. So, only if it is 0, 1 or 1, 0 you do the $P + A$ or $P + S$ operation otherwise do not do just move.

(Refer Slide Time: 01:01:51)

Example for Booth Algorithm based Multiplier

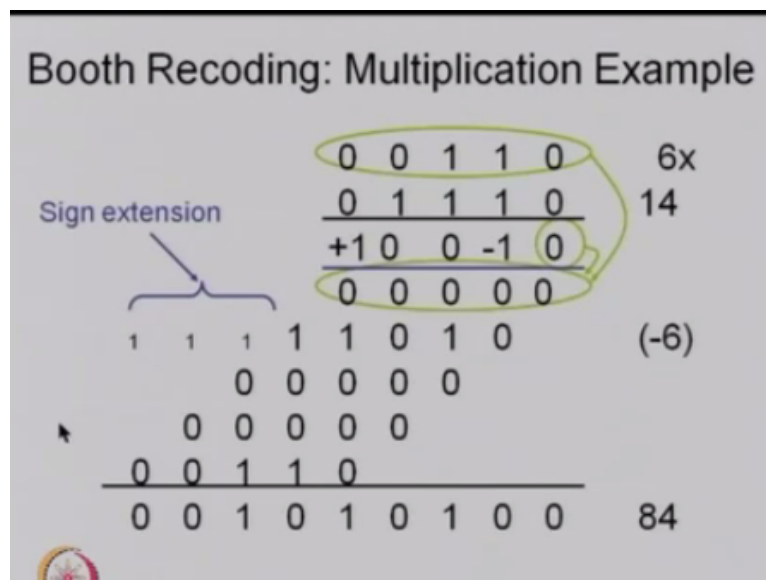
- $A = 0011\ 0000\ 0$
 - $S = 1101\ 0000\ 0$
 - $P = 0000\ 1100\ 0$
 - Perform the loop four times :
 - $P = 0000\ 1100\ 0$. The last two bits are 00.
 - $P = 0000\ 0110\ 0$. Arithmetic right shift.
 - $P = 0000\ 0110\ 0$. The last two bits are 00.
 - $P = 0000\ 0011\ 0$. Arithmetic right shift.
 - $P = 0000\ 0011\ 0$. The last two bits are 10.
 - $P = 1101\ 0011\ 0$. $P = P + S$.
 - $P = 1110\ 1001\ 1$. Arithmetic right shift.
 - $P = 1110\ 1001\ 1$. The last two bits are 11.
 - $P = 1111\ 0100\ 1$. Arithmetic right shift.
- The product is 1111 0100, which is -12.



Now here is an example, I think that will clarify what I said. A is 0011 and the rest is 008 bits and the finally happened is 0 here. S is; please remember this is 3 and this is again 4 +; sorry $8 + 2, 4, 12; 13; 13, 3$ is the 39, okay but if it is in the minus numbers, then this is -3 and this is 4, so I may actually looking for $4 * -3$ as my number, okay. So initially you have the P is 110 okay, which is first term, perform the loop 4 times.

So first $P = 0001100$ add the 0, the last 2 bits of P are 00, so arithmetic right shift since they are 00, do not do anything just shift one side, okay. Now, again we see the last 2 bits are 00 so just shift again, so you get 000110. Now, the next 2 bits are 1, 0; so do operation $P + S$. So, this is your P , add S to that and if you do this operation, you get this and again shift to the right; again this side.

(Refer Slide Time: 01:03:48)



Now, we see an observe 1 1, so we okay 11 means again the last 2 bits are 1, 1 so no operation to perform only shift so this, and if this see, this number 11101011, which is nothing but -12. So, if I perform this operation, I can always create this number, this is our basic idea of both recoding. Instead of having only 1 or 0 this, you can have the number in 1 and -1 codes and by doing this; we can generate the number in minus itself. I will give you an example.

(Refer Slide Time: 01:04:01)

Booth Recoding: Advantages and Disadvantages

- Depends on the architecture
 - Potential advantage: might reduce the # of 1's in multiplier
- In the multipliers that we have seen so far:
 - Doesn't save in speed (still have to wait for the critical path, e.g., the shift-add delay in sequential multiplier)
 - Increases area: recoding circuitry AND subtraction

Before we go this, the Booth recoding it advantages and disadvantage are depends on the architecture potential advantage might reduce the number of ones in multiplier. In the multiplier, that we have seen so far, does not save any speed still have to wait for a critical path, increase area, recoding, circuitry, AND subtraction. So, a new idea was figured out, okay, so what do we do really in the coding part.

(Refer Slide Time: 01:04:46)

Example

Multiplicand A	0 0 1 1 0 1	(13)
Multiplier x	1 1 1 0 1 0	(-6)

Booth Recoding of x

1 1 1 0 1 0

1 1 0 1 0
 +1 -1
 0 0
 -1
 1 -1
 1 1

0	0	-1	1	-1	0
0.A		-1A		-2A	

So, I may actually show you how do I coded as an example, before I actually look into what I am really this, okay here is my operation. Let us say in twos complement, I have this 001101 is 13 and 111010 is -6, so I recode the multiplier x, okay. My initial number is 110010, so the way I recode is the following. The way recoding is done, I think I will go back and show the other slide but this is to simplify before I go there.

For every, the first of course, you leave zeros but the next ones whenever you see one, just below that put -1 and +1 in the next bit position. For 0, you put only 0,0, for 1; -1 +1 for this one -1 +1; for this one -1 and then you just write down the numbers, this is of course initially was 0, 0 here. So, you say first number is; -1, 0 the second; please remember second is -1 and +1, +1 and -1, this is 0 and 0, this add is -1 is 0, -1 +1 is 0, so for this; so please remember I am actually doing this operation, this operation and this operation.

If I do this, I get -1 0, -1 +1, 0 0 and finally 1. Now, Booth encoding or recoding as we said, essentially says and we done we now back and show what I am talking about? Is this operation is say; do multiplication do; do operation which is called -2 times the A is the multiplicand and add to that, this is addition of -1A and this is 0, so just no addition. Now, I will come back to this little later once again, okay.

(Refer Slide Time: 01:07:00)

Modified Booth Multiplier: Idea

- Group pairs, leaving -2, -1, 0, 1, 2
 - Grouping reduces # of partial products by half
- Booth recoding results in:
 - Gets rid of 3's (sequences of 1's in general)

$$\begin{array}{cccccccccccc} 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \end{array}$$

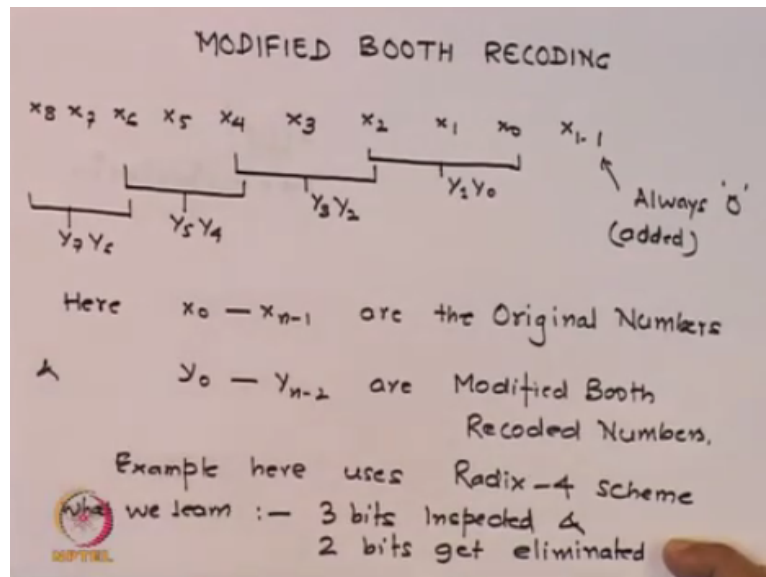
$$\begin{array}{ccccc} (+1 & -1) & & (+1 & -1) & & (+1 & -1) \\ & (+1 & -1) & & (+1 & -1) & & \\ & & (+1 & -1) & & & & \end{array}$$

+1 0 +2	-1 +1 -1	0 0 0	-1 0 -2	0 +1 +1	-1 0 -2
------------	-------------	----------	------------	------------	------------

[04]an

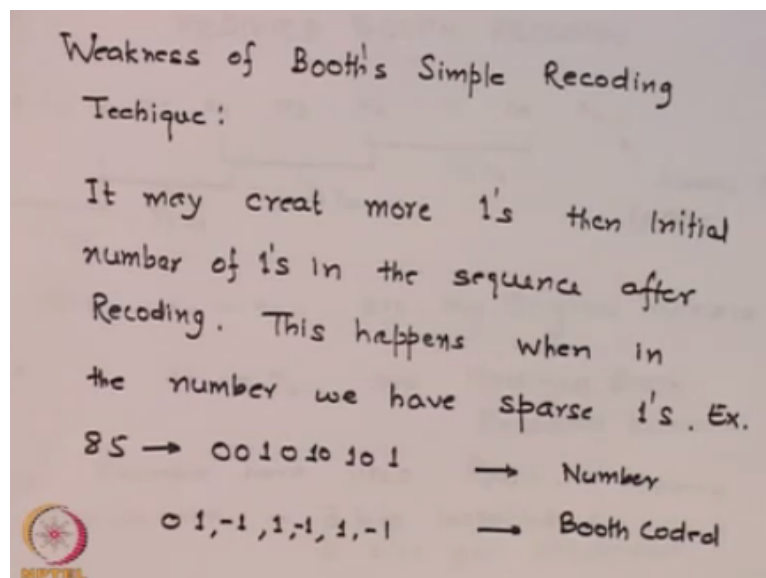
When I come back to this number evaluation, so, how do I get this? Path is the following, so this is how I do, okay. I have a group appears leaving -2 -1 this and as I said produces the number of partial product by half, so how it is done? It gets rid of 3s; sequence of ones in general, okay and I suppose, I have that expressions with me, here is the one what I am saying, we can see both simultaneously to some extent, okay.

(Refer Slide Time: 01:07:27)



You have x_0 to x_{n-1} as your number in two's complement, add x_{-1} which is always added to the LSB extreme LSB side and is always 0, okay. So, if you see at this table, this number, you have 011011100 and 1 and this last 0 is appended by me, okay. Now what do I do? I said for everyone I write -1 and +1 okay. I do not have to write zeros, because zeros do not add. For this one, I write -1+1, for this one I write this, for this one, I write -1 +1, for this one, I write -1 +1, for this one I write this.

(Refer Slide Time: 01:08:50)



And if I then add vertically down, so it is -1 0 then it is 0 +1 -1 0 0 0 -1, now we know Booth encoding or recoding says this is equivalent of -2, this is equivalent of 0, 1 so, from here now we come back to this. What was the problem in Booth's normal recoding? In normal coding has some difficulty one can see, which is not very obvious to many, okay. In a normal Booth simple recoding, it may create, if you just do the normal recoding as we did earlier.

Then, you may have initial number, which has certain number of ones but when you recoded, you may have larger number of ones; -1 or +1 whatever it is. Here is an example, this happens particularly when there is ones are very sparse. For example, given in a book this is the K. Roys book, it says that 85 can be a number, which is 001010101 in a twos complement.

And if I Booth coding it, this will give me 01, -1, 1, -1, 1, -1, which essentially means now there are more operations to perform, one means, there is operation to perform, 0 means no operation to perform. You have only 4 operations here in normal case, here you have 3 + 3, 6 operations of ones. So, in case of Booth, normal Booth multiplication, there is a possibility of error.

In the sense, you do not save partial product sums actually increase sometimes, in this particular area occurs when there is a sparse ones. The live number of ones any Booth recoding will reduce that ones to more zeros and therefore number of partial products will reduce and this is very, very relevant in what we call modified Booths recoding. So, in a modified Booth recoding what I am going to do is?

I have; I generate x_{n-1} , add this append this number 0, I leave this number and look into the first 3 bits from LSB, this is additional LSB plus we are not counting in inspection. So, we say, we will start inspecting first 3 bits and using the inspection of $x_0 x_1 x_2$, I can recoded into $y_1 y_0$. However, in the Booth normal recodings, I would have done $x_1 x_0 x_3 x_0$ and then there would have been possibility if they both would have been 0 or 1 alternatively.

So, sparsity would have come, now what I do is, I take the last one once again and now with this; so, even if it is 0 or 1 with this it will be taken care and then I will generate another recoded values which is $y_3 y_2$. I start again with x_4 go to x_6 , I create $y_4 y_5$, I will start with x_6 , go to y_7 and so on and so forth create $y_6 y_7 y_8 y_9$, things of that, all odd numbers, okay finally. So you have x_0 to x_{n-1} is the original number and y_0 to y_{n-2} is the modified recoded number.

(Refer Slide Time: 01:12:01)

Inspected Bits			Recoded Bits		Recoded Digit times Multiplicand	
x_i	x_{i-1}	x_{i-2}	y_i	y_{i-1}		
0	0	0	0	0	0	0
0	0	1	0	1	+1	1A
0	1	0	0	1	+1	1A
0	1	1	1	0	+2	2A
1	0	0	-1	0	-2	-2A
1	0	1	0	-1	-1	-1A
1	1	0	0	-1	-1	-1A
1	1	1	0	0	0	0

You can see, we are using a 4 bit Radix 4 scheme here, we inspect 3 and every time we inspect 3, 2 bits gets eliminated because common this is there. Now, this is essentially what Booth encoding is about or recoding is about? Example here is; I have a x_1 ; 000 then the recoded bits are 00, I will come back the table again and again you will see the same thing. This is 001, the $y_i y_{i-1}$ is 0, 1; 010 is 0, 1; 011 is 1, 0; 100 is -1, 0; this is code 100 is 0, -1; 1 1 0 is 0, -1; 1 1 1 is 0, 0.

Now, the operation we have to perform is called A, is your multiplicand, so how many times this recorded digit times that multiplication has to be done, multiplicand has to be added. So the actual from 0, 0 does an operation of 0; 0, 1 does 1; 0, 1 does 1; 1, 0 does 2; -1, 0; -2, 0, -1; -1, essentially, it says the operation should have this is zero addition, $0 * A$ addition, this is whatever is the last is you add one times your multiplicand, one times multiplicand, 2 times multiplicand.

Then add - 2 times means actually subtract kind of thing, minus 1A times, -1A times, 0 times this is called Booths encoding table, okay. Now, if you look at the Booth encoding table, in this expression how to get that? Please come back to the slide again, so for every one array represent -1 +1, I gave you the colour because I will not say it; this one gives blue one is +1 --1, this black one is +1 -1, green is; of course zeros are all zeros will not added at all, you can write 0 0 you finish.

(Refer Slide Time: 01:14:30)



Modified Booth Multiplier: Idea (cont.)

- Can encode the digits by looking at three bits at a time
- Booth recoding table:

i+1	i	i-1	add
0	0	0	0*M
0	0	1	1*M
0	1	0	1*M
0	1	1	2*M
1	0	0	-2*M
1	0	1	-1*M
1	1	0	-1*M
1	1	1	0*M

– Must be able to add *multiplicand* times -2, -1, 0, 1 and 2

– Since Booth recoding got rid of 3's, generating partial products is not that hard (shifting and negating)


Then 1 is +1 -1 and this one is +1 and then you add vertically so, you get -1 and of course 0 is here so -1 0 then the next is 0 +1, then you have -1 0, 0 0 -1 +1 and from the Booths table, we know -1 0 is -2, 0 +1 is +1, -1 0 is -2, 0 0 is 2, -1 +1 is -1, +1 is 2. So, I have I know what operation to perform when I convert the recoded system into this and here is what I do the same thing which I said earlier can be rewritten i, i -1, i +1 are the 001 and the kind of operations you perform.

(Refer Slide Time: 01:14:47)

Modified Booth Multiplier: Idea

- Interpretation of the Booth recoding table:

i+1	i	i-1	add	Explanation
0	0	0	0*M	No string of 1's in sight
0	0	1	1*M	End of a string of 1's
0	1	0	1*M	Isolated 1
0	1	1	2*M	End of a string of 1's
1	0	0	-2*M	Beginning of a string of 1's
1	0	1	-1*M	End one string, begin new one
1	1	0	-1*M	Beginning of a string of 1's
1	1	1	0*M	Continuation of string of 1's


 121

Since Booth recoding, got rid of 3s generating partial products is not that hard because it is only shifting and negating has to be done okay. This is the same thing again explanation is given more detail, number of strings of ones in the side, end of strings of ones this is called isolated one, this means end of strings are ones, this means beginning of string of ones, end of one string beginning new ones, beginning of string of ones and continuation of string of ones.

(Refer Slide Time: 01:15:17)

Modified Booth Recoding: Summary

- Grouping multiplier bits into pairs
 - Orthogonal idea to the Booth recoding
 - Reduces the num of partial products to half
 - If Booth recoding not used → have to be able to multiply by 3 (hard: shift+add)
- Applying the grouping idea to Booth → Modified Booth Recoding (Encoding)
 - We already got rid of sequences of 1's → no mult by 3
 - Just negate, shift once or twice




122

The kind of operations, add operations you perform has this explanation. In summary, what do you do? Grouping multiplier bits into pairs, orthogonal ideas to the Booth recoding, reduces the number of partial product to half, if Booth recoding not used, we have to have been able to multiply by 3, which is hard shift plus this, 3 multiply addition to be done. Applying the grouping idea to Booth, modifies recoding as it is called as encoding.

(Refer Slide Time: 01:15:52)

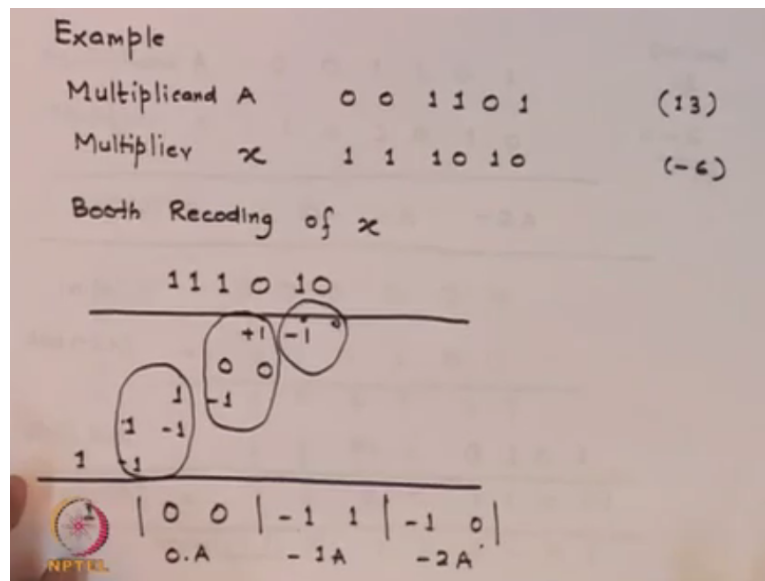
Modified Booth Multiplier: Summary (cont.)

- Uses high-radix to reduce number of intermediate addition operands
 - Can go higher: radix-8, radix-16
 - Radix-8 should implement $\times 3$, $\times -3$, $\times 4$, $\times -4$
 - Recoding and partial product generation becomes more complex
- Can automatically take care of signed multiplication



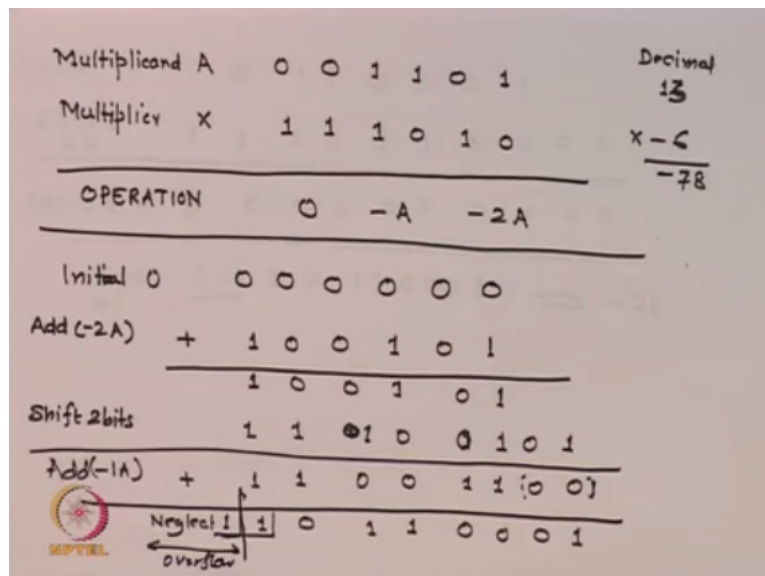
We have already got rid of sequence of ones, no multiplication by 3 numbers, just negate shift once or twice and that is the idea. Use high radix to reduce number of intermediate addition operands, can go higher, you can have radix of 8, radix of 16. Of course, you will have to implement $3 - 3$, $4 - 4$ large number of such these operations to be performed but it will be more accurate and sometimes much faster.

(Refer Slide Time: 01:16:36)



Recoding and partial product generation become more complex than, of course, you can automatically take care of signed multiplication. Typical Booth multiplier is shown here but before I go now, I will show you the example of that, here is my example, which I just now was talking to you, okay. I have an operation, which is shown here, multiplicand is 13, 00110 and you have multiplier which is x, which is in twos complement of -6 is 111010.

(Refer Slide Time: 01:17:17)



So, I recode multiplier x, this is 111010, I again put -1 +1, -1 +1, -1 +1 this and add. So I get -1 0, -1 1, 0 0 and I know -1 0 from booth recoding table is -2A, this -1 -1 means -1 is 0 0 means 0A. Having known the operations to be performed, I start looking for the actual things which I want to do. Here is your decimal number 13 * -6, you expect an answer -78, you want to do this operation of 0 -A -2A, okay.

Let us say initial, product or sum is 000000, partial product sum call it. The first operation you want to do $-2A$ and A is; please remember multiplicand, okay. Now, if I do this 2 of this and shift, I get 100101 take complement and shift you can get this 100101, and then add since it is 0, this number will remain 100101, okay then shift 2 bits because you have to 2, so shift 2 bits, 11100101.

To this now, add $-1A$, $-1A$ – same thing is complement of that is 1100 is complement of that, please take it complement, ones complement is 110011, okay and then append since there were 2 addition number here, because of shift, you operate 00 here okay and add. So, you get 1 0 then 1 and 1; 0, 1 and 1; 0, 1, 1, 1 and 1; 0, 1 and 1; carry 1 and but we say since it is overflow, this part is an, so neglect.

(Refer Slide Time: 01:19:14)

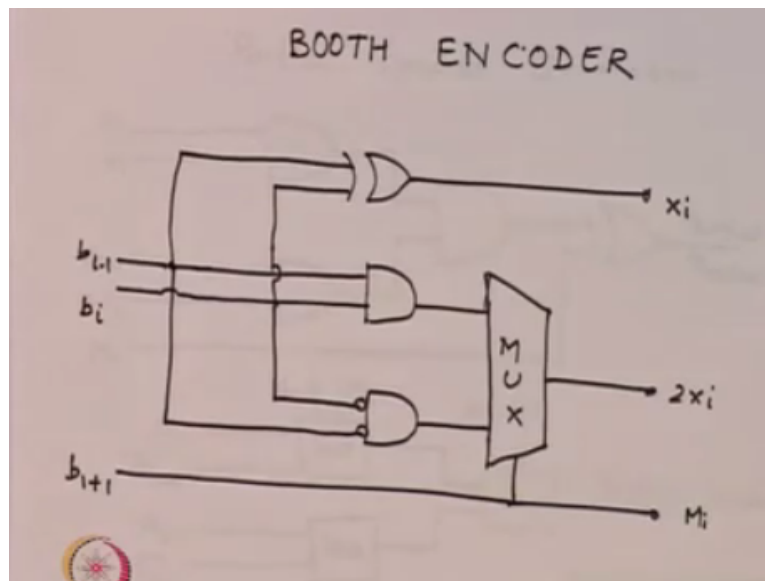
Last 1 0 1 1 0 0 0 1
 Shift 2 bits 1 1 1 0 1 1 0 0 0 1
 Add (0.A) + 0 0 0 0 0 0 0 0 0 0
 Sign Bit 1 1 | 1 0 1 1 0 0 0 1 $\Rightarrow -78$

So, the number, which I got is 10110011. Then, we have to shift this the least number by 2 bits 1111 this and if I do it and after shift I get add to this one, so I get ; sorry add 0 A to it, of course now add 0, 0 means, no addition. So, this is the number 101110, this last 2 of course are signed bits, 10110001 which essentially with the signed bit this is 78 with a minus sign. So, what does that Booth encoding has done?

You can see, since Booth encoding actually only uses those terms which have ones and by Booth encoding or recoding, we are reduce the number of ones, the net partial products sums required much smaller you can see in a 4 step operation in the first of course is recoding one

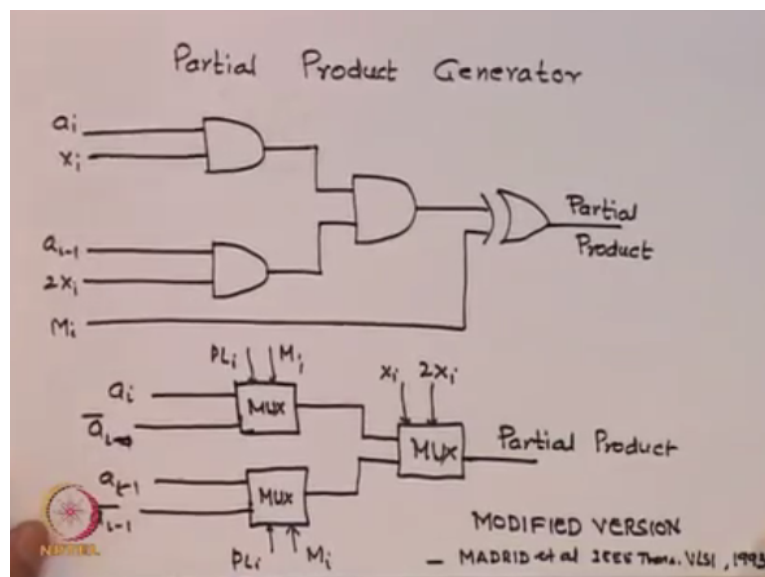
operation, then 3 operation is here and 2 operation is here and 4, 5 operations I am able to generate multiplying of 13 into even signed bit multiplication.

(Refer Slide Time: 01:20:49)



Before we leave this part, okay I may like to show you of course you require Booth encoding, what kind of circuits we use, you need an XO, you need 2 inverters, 2 AND gates or an OR gate you can say, okay and a multiplexer, this is b_i , b_{i-1} ; these are the bits, which are entering, the x_i is of course is XOR of b_i , please remember this is b_{i-1} , this is b_i , and this is b_i , b_{i-1} , XOR is x_i .

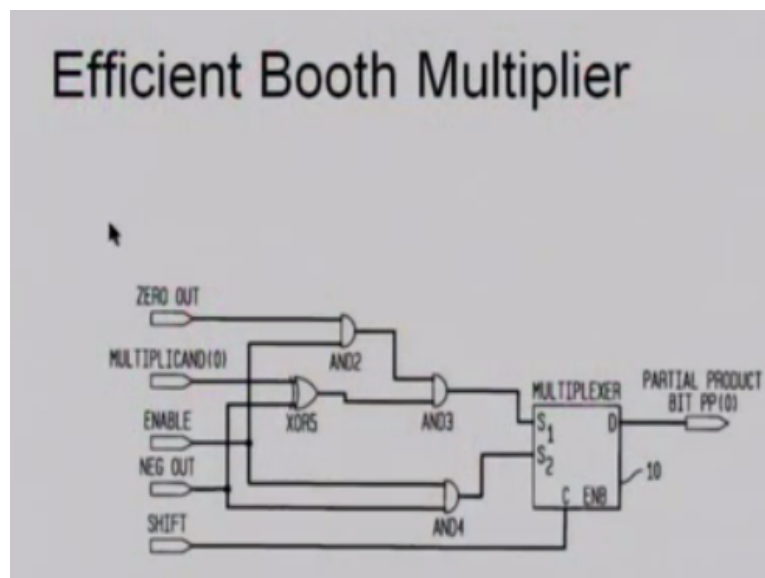
(Refer Slide Time: 01:21:29)



Then, this addition of this, are complement of this is passed as $2x_i$ and this is directly passed as m_i , this is what recoding while asking. The other part circuit you need is to create partial product generator, so you have you need 3 AND gate and XOR gate produce the partial

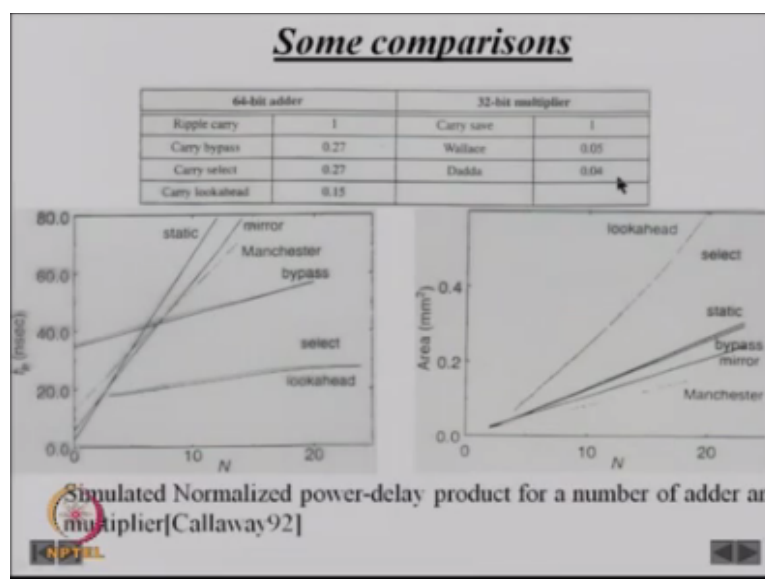
product and you need a modified version you which does not use a AND gates but only muxes, 3 muxes.

(Refer Slide Time: 01:22:16)



Then, these circuits are taken from MADRID papers in IEEE on VLSI 1993. So, you can see basically you require only few muxes for encoding and passing the partial products and shift operation because there is every time you are shifting, you need shift registers, so and it should be able to shift the data left and right. So, before we leave this part probably okay, the one circuit, which I already shown is efficient Booth multiplier which is same as what just now I said.

(Refer Slide Time: 01:22:20)

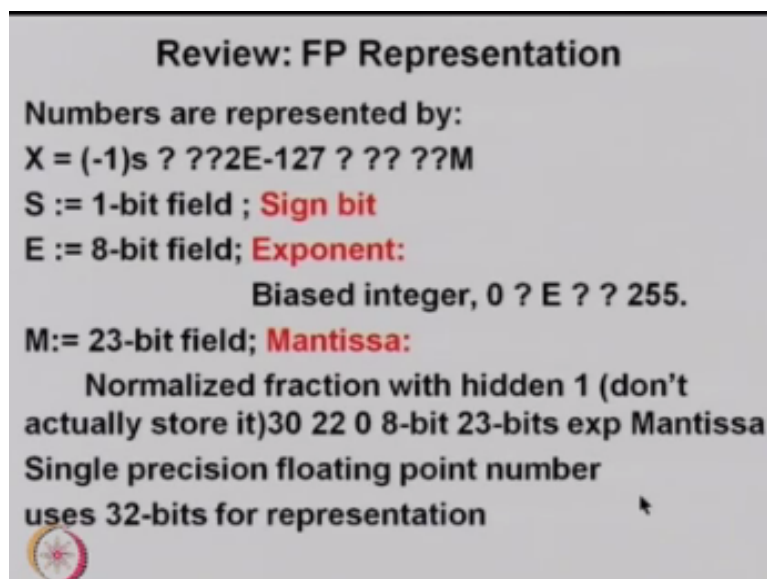


This of course, a slide need not worry about adder, which we already taken earlier, this is essentially we are looking for; let us say carry save as a unit one, okay and compared to this if

you look at the speeds for a Wallace tree, you will have 0.05 and if you do the other 0.05 if you do Booth, it will be 0.001. So, essentially power delay products this is essentially a power delay product can be reduced in a Booths encoding.

There are variety of version carry save multiplier 32 bit is the reference then if use tree, it will be 120th of that, it will be 125th, if you use Booths encoding further on that, it will be 100. So, that is the idea of improving the speed power product of any multiplier. Before we leave to shift operation, the last part of my circuit requirement; multiplier requirement is the floating point representation.

(Refer Slide Time: 01:23:32)



Review: FP Representation

Numbers are represented by:

$$X = (-1)^s \cdot 2^E \cdot M$$

S := 1-bit field ; **Sign bit**

E := 8-bit field; **Exponent:**

Biased integer, 0 ≤ E ≤ 255.

M := 23-bit field; **Mantissa:**

Normalized fraction with hidden 1 (don't actually store it)

30 22 0 8-bit 23-bits exp Mantissa

Single precision floating point number uses 32-bits for representation

We know integer operations, so we also should be able to do some kind of floating point multiplications. Before we go to that, let us look at the numbers. We can see that typically, any number X is represented as minus; whatever signed bit plus some integer numbers here before the exponent and then you have E to the power some exponent number in Mantissa. So, it is called a one-bit field for the signed bit, 8-bit field for the exponent.

(Refer Slide Time: 01:24:32)

Floating Point Representation

- The mantissa represents a fraction using binary notation:
- $M = .s_1, s_2, s_3 \dots = 1.0 + s_1 \cdot 2^{-1} + s_2 \cdot 2^{-2} + s_3 \cdot 2^{-3} + \dots$
- Example: $X = -0.75_{10}$ in single precision ($-(1/2 + 1/4)$)
- $-0.75_{10} = -0.11_2 = (-1) \times 1.1_2 \times 2^{-1}$
- $= (-1) \times 1.1_2 \times 2^{126-127}$
- $S = 1$; Exp $\Rightarrow 126_{10} = 0111\ 1110_2$;
- $M = 100\ 0000\ 0000\ 0000\ 0000\ 0000_2$
- $X = 1\ 0111\ 1110\ 100\ 0000\ 0000\ 0000\ 0000\ 0000$



You may have a biased integers 0 to 255 and you have 23-bit Mantissa. So, totally, typically floating point number is represented in the 32-bit representation, which is called single position, which will have; for example, 0 bit, 8 bits, 1 bit for signed one do not actually store it, you have 8 bit for the exponent fields and 23 bit for the Mantissa field, okay. Here is the number to show the same thing, this is your number; M Mantissa is $s_1 s_2 s_3 \dots$ in this form.

Example; -0.75 in 10 in single position is $1/2 + 1/4$, this is can be written in twos complement and twos binary, -1.11_2 this, if you write this, this format, 2 to the power $126 - 2$ to the power 127 , S is 1, sign bit, exponent is 126 and 127, so can be represent in binary this and Mantissa is 1 0 0; all zeros and therefore a number, which is shown here in 32 bit is the following. This is 20 second Mantissa, okay.

(Refer Slide Time: 01:25:56)

FP Addition

- $-1.610 \times 10^{-1} + 9.999 \times 10^1$
- • Step 1: – Align decimal point:
- $0.016 \times 10^1 + 9.999 \times 10^1$
- • Step 2:– Add:
- 10.015×10^1
- • Step 3:– Normalize:
- 1.0015×10^2
- • Step 4: – Round:
- 1.002×10^2
- • May need to repeat steps 3 and 4 if result not normal after rounding. (renormalization)

Then there are exponent bits, please remember one data is say, you have one signed bit, 23 exponent bits, sorry 8 bit exponent and 23 Mantissa fields. So, this is 23 Mantissa bit, then 8 exponent bit, okay and the last of course is your, this is your last zeros of 31, this is your sign bit, which is shown here. So, how do I do addition in this? First, let us say you have number -1.610 in decimal at this, so in decimal what do we do is, represent that number in tens number and see to it that there decimal points are align.

So, both are represent tens, 10 to the power one, so this can be 0.016 10 to the power 1 and this can be 9.99 10 to the power1 and then because this is common, we do not have to do anything just add these 2 terms you get 10.01 10 to the power1. Then we normalise, the next operation is we normalise. So, what do we mean by normalise? We do not want 2 decimal numbers, bits before decimal should be only one, so we say it is into 10, so 1.015.

(Refer Slide Time: 01:27:05)

Floating Point

- Still use a fixed number of bits
 - Sign bit S, exponent E, significand F
 - Value: $(-1)^S \times F \times 2^E$
- IEEE 754 standard

	Size	Exponent	Significand	Range
Single precision	32b	8b	23b	$2 \times 10^{\pm 38}$
Double precision	64b	11b	52b	$2 \times 10^{\pm 308}$

© 2002 Lapack, W&L

Then, we may round it off, how many accuracy you want? Say, for example, 1.002 is good enough if I neglect 5, 5 or more than 5 we make it 2, last bit so 1.002. We may need to repeat step in 3, if the normalisation is after rounding is not correct. This is a single precision number, any number is represents $-1 \times 2^E \times F$, this is how, the IEEE 754 standard into right a floating point number.

And single precision, which is 32 bit, 8 bits of exponents, 23 bits of significand and the range is 2 to the power 10 + - 38 and double precision is 64, 11 bit exponent, 52 bits of significand, and the range is 2 into 2 power this, okay. So, once we know we can since; each is an individual integer numbers; fixed numbers the operations can be independently performed using integer theories and one can do this.

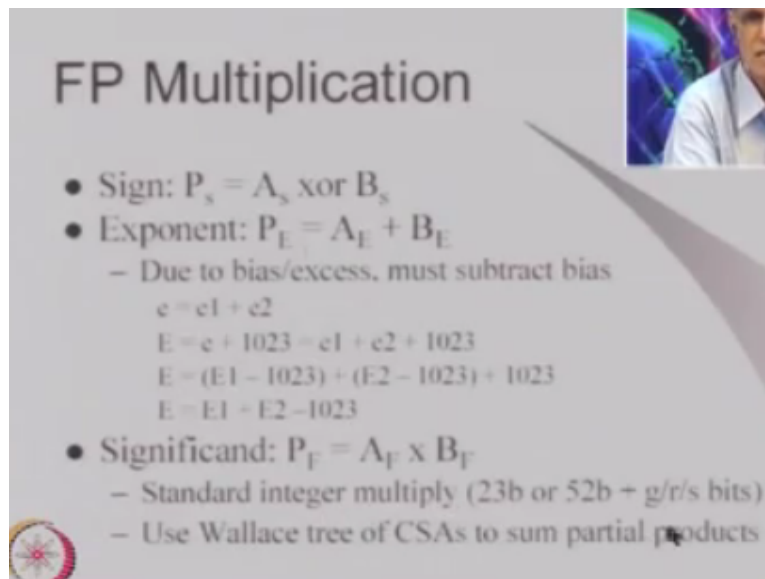
(Refer Slide Time: 01:27:48)

FP Multiplication

- Compute sign, exponent, significand
- Normalize
 - Shift left, right by 1
- Check for overflow, underflow
- Round
- Normalize again (if necessary)

For doing a multiplication compute sign, exponent, significand, normalise; shift, left, right by one, check for overflow, under flow, round it and normalise. It is identical to the same.

(Refer Slide Time: 01:28:00)

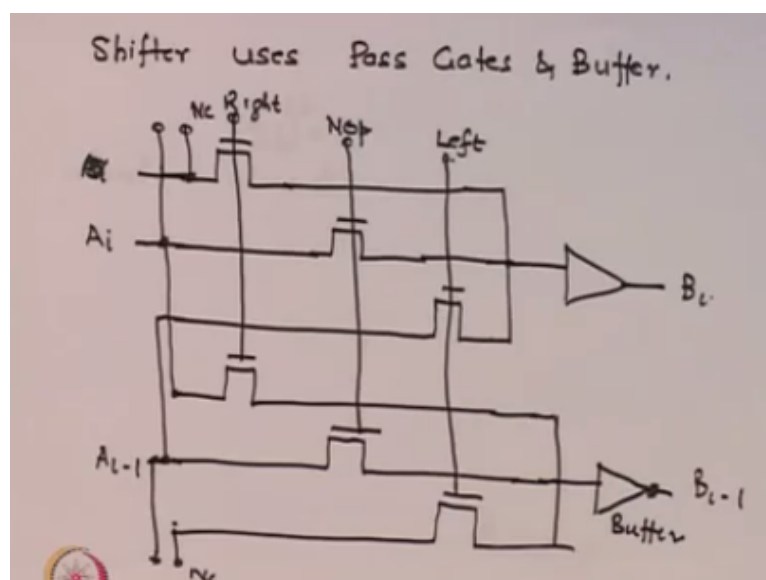


FP Multiplication

- Sign: $P_s = A_s \text{ xor } B_s$
- Exponent: $P_E = A_E + B_E$
 - Due to bias/excess, must subtract bias
 - $e = e1 + e2$
 - $E = e + 1023 = e1 + e2 + 1023$
 - $E = (E1 - 1023) + (E2 - 1023) + 1023$
 - $E = E1 + E2 - 1023$
- Significand: $P_F = A_F \times B_F$
 - Standard integer multiply (23b or 52b + g/r/s bits)
 - Use Wallace tree of CSAs to sum partial products

Sign is P_s , A , XOR, B_s , exponent is A_E , B_E , due to bias excess, must subtract bias kind of thing, significand is A_F , B_F , standard integer multiplier, use Wallace tree for the addition, creating partial products. So, please remember floating point numbers are independently handled in 3 zones, this, exponent is separately handled, Mantissa; sign are separately handled.

(Refer Slide Time: 01:28:56)

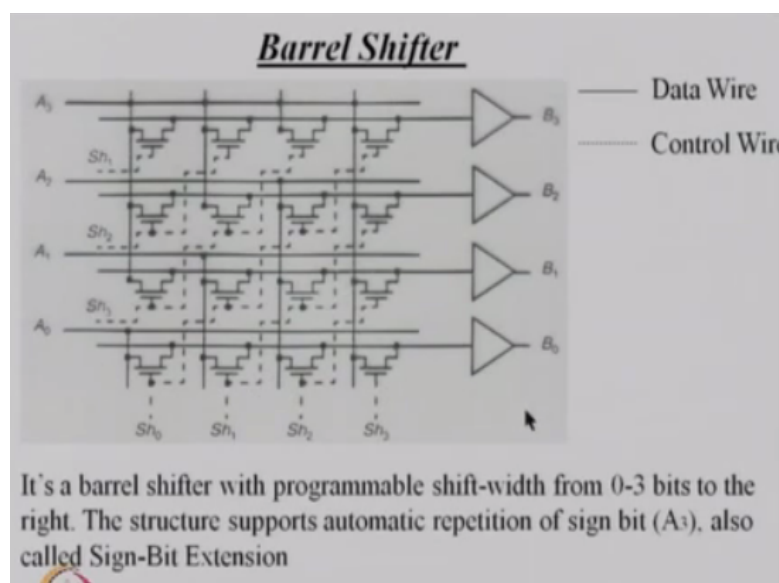


And because of that, we can and put into a different shift register positions to actually get the multiplier operations. Now, the last but not the least part of this whole circuit; is we are keep talking of shifting, so we kept talking of shifting the data to the left. A typical shift register

based; pass gate based shift register is shown here, which allows the data to move to right or left. One can see here only pass gates have been used and buffer of course since you may have to drive.

The first one is the; for the right, the second one is for no operation and the third one is left. So, if you want to move the data from right, so you make this as 1 and let us say this is my A_i , no operation means 0, left is not operation, so this data is appearing here, right shift. So, you can see from here it has gone to the right, $i - 1$. If you want left, you can see that if I want left, I must go above, I want to reach here.

(Refer Slide Time: 01:29:58)

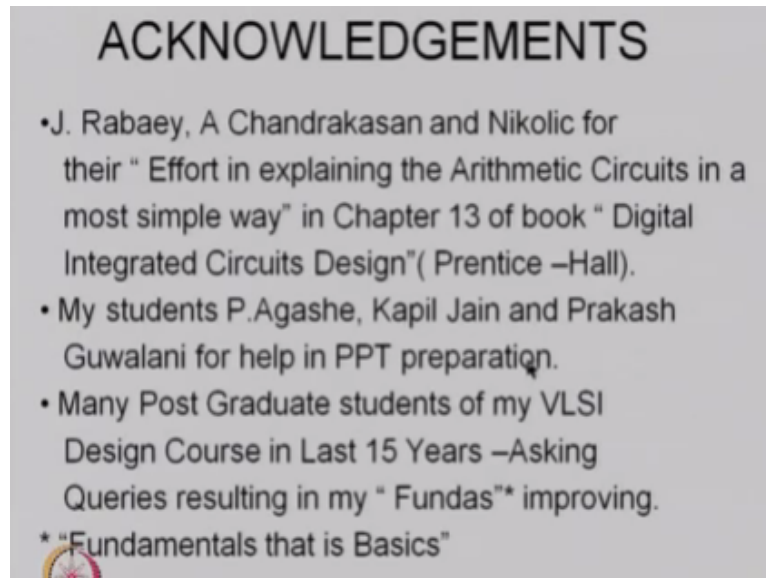


So, obviously I will go up this is high, and since this is high, this is transferred. So, left and right data can be transferred and no operation can be also; to show you this data wire and control, this is a 4 bit shift; Barrel Shifter is shown here, there I shown you 2 is 4 , it is identical, shift 1, shift 2; Sh1, Sh2, Sh0 are the 4 shift signals, it can be depending on right and left whether these are, these are turned none, data can be transferred here.

Data can be transferred here or transferred here depending on which pass gates are switched on, okay. This can be controlled by small logic which will allow this to create shifts, signals and those shift signals will allow you to create A_0 to go to B_3 or A_0 to go to B_1 or vice versa coming down, each bits can be reposition left or right by using this kind of Barrel shifter, 4 bit simultaneously can be given.

They can be put like this or they can be put directly like this. So, obviously shift register, shift operations can be easily performed left and right using a Barrel shift register. So, we have now seen in a multiplier, you need adder, you need a shifter. We already had seen all kinds of adders in our earlier implementation. We are today seen all kinds of multiplier possibility, we also looked into floating point possibilities and using Barrel shifters and those 2 blocks, different kinds of blocks depending on the area, power, speed, and of course, the accuracy.

(Refer Slide Time: 01:31:48)



One can choose different hardware circuits and different hardware circuits will lead to a different performance index and based on that you can choose it and implement any addition adder multiplier in your actual hardware. These are the books from where much of my work was taken. Basically, you can; for the first level of understanding you can use Rabaey's book. There are other 2 books we all know is Eshraghian and Weste, and this and I already given my other references to you.

(Refer Slide Time: 01:32:25)

Book-References

1. "Adapted from UCB Course site (Copyright) of Prof. Rabaey & from "Digital Integrated Circuits: A Design Perspective," Rabaey J.M., Chandrakasan A., Nikolic B., Prentice Hall (India), 2003.
2. Waste N.H. and Eshraghian K, "Principles of CMOS VLSI Design", Addison Wesley Publishing Company," Santa Clara, CA, 1994.
3. Madiseti Vijay k, " VLSI Digital Signal Processors," IEEE Press, Butterworth- Heinemann, Newton, MA, 1995.
4. Hwang K, "Compiler Arithmetic: Principles, Architectures, and Design," John Wiley and Sons, New York, 1979.
5. Lars Wanhammar, "DSP Integrated Circuits", Academic Press, 2000
6. Kiat-Seng Yeao and Kaushik Roy, "Low Voltage , Low Power VLSI Subsystems", Mc Graw Hill, 2006

Of course, there are many thanks to my students because they are once who create many of old style, this is of course, my VLSI, my post graduate students in VLSI in last 15, 20 years may be more than 15 years. They asked me many things which allow me to understand better and of course there are some good book references you can see some of the slides from the University of California, Berkeley Course Site due to Rabaey and others and their book.

Credit to printers Hall, for allowing it to do that. Then, there is a book by Addison Wesley, which is one of the very famous old book Waste N. H. Eshraghian, Principles of CMOS VLSI design, which are published in 1994, but still seems to be one of the best system design, device 2 system design book, many of the circuit shown here I have been taken there. Then there is a book on DSP, processors which is written by Madiseti Vijay, which is published by ButterWorth Heinemann.

So some slide some data, things were taken from this book and then there is a book on complier arithmetic by Hwang. This is John Wiley, it is one of the oldest book in hand. But, if you really read classic books, you really understand much more okay and therefore I recommend those who are looking for advanced VLSI, should look for the last 3 book very carefully.

(Refer Slide Time: 01:34:03)

Technical paper References

Number Systems

A. Avizienis, "[Digital Computer Arithmetic: A Unified Algorithmic Specification](#)", Proceedings of Symposium on Computers and Automata, p. 509-525, Brooklyn, New York, April 13-15, 1971.

Redundant Number Systems:

A. Avizienis, "[On a Flexible Implementation of Digital Computer Arithmetic](#)", Proceedings of IFIP Congress 62, Munich 1962.

A. Avizienis, "[Arithmetic Microsystems for the Synthesis of Function Generators](#)", Proceedings of the IEEE, Vol.54, No.12, December 1966.

A. Avizienis, "[Theory of Digital Computer Arithmetic](#)", Class notes for Engr 225A, UCLA, 1968/69.

A. K. Yeung, J. M. Rabaey, "[A 210Mb/s Radix-4 Bit-level Pipelined Viterbi Decoder](#)", Proceedings of International Solid-State Circuits Conference, San Francisco, February 1995.

The another book, which you can see is Lars Wanhammar, DSP ICs, which is published by Academic press and last but not the least, the very recent book appearing from McGraw Hill, which is written by Kiat Seng Yeao and Kaushik Roy much of the data, powers, speed, this I have been taken from Kaushik Roys book and due regard to them. There are many references on number systems.

(Refer Slide Time: 01:34:04)

ADDERS

Manchester Carry Chain:

T. Kilburn, D. B. G. Edwards, D. Aspinall, "[Parallel Addition in Digital Computers: A New Fast "Carry" Circuit](#)", Proceedings of IEE, Vol. 106, pt. B, p. 464, September 1959.

V. G. Oklobdzija and E. R. Barnes, "[Some Optimal Schemes For ALU Implementation In VLSI Technology](#)", *Proceedings of the 7th Symposium on Computer Arithmetic ARITH-7*, pp. 2-8. Reprinted in *Computer Arithmetic*, E. E. Swartzlander, (editor), Vol. II, pp. 137-142, 1985.

V. G. Oklobdzija and E. R. Barnes, "[On Implementing Addition In VLSI Technology](#)", *IEEE Journal of Parallel and Distributed Computing*, No. 5, pp. 716-728, 1988.

V. G. Oklobdzija, "[Simple And Efficient CMOS Circuit For Fast VLSI Adder Realization](#)", *Proceedings of the International Symposium on Circuits and Systems*, pp. 1-4, 1988.

(Refer Slide Time: 01:34:08)

ADDERS (CONT.)

Carry-Select Adder:

O. J. Bedrij, "[Carry-Select Adder](#)", IRE Transactions on Electronic Computers, p. 340, June 1962.

Conditional-Sum Adder:

Sklanski, "[Conditional-Sum Addition Logic](#)", IRE Transaction on Electronic Computers, EC-9, pp. 226-231, 1960.

CLA Adder:

Weinberger, J.L. Smith, "[A Logic for High-Speed Addition](#)", National Bureau of Standards, Circulation 591, p. 3-12, 1958.

Naini, D. Bearden, W. Anderson, "[A 4.5nS 96-b CMOS Adder Design](#)", IEEE 1992 Custom Integrated Circuits Conference, 1992.

B.D. Lee, V.G. Oklobdzija, "[Improved CLA Scheme with Optimized Delay](#)", Journal of VLSI Signal Processing, Vol. 3, p. 265-274, 1991.

(Refer Slide Time: 01:34:10)

Ling Adder:

H. Ling, "[High Speed Binary Parallel Adder](#)", IEEE Transactions on Electronic Computers, EC-15, p.799-809, October, 1966.

H. Ling, "[High-Speed Binary Adder](#)", IBM J. Res. Dev., vol.25, p. 156-66, 1981.

R. W. Doran, "[Variants on an Improved Carry Look-Ahead Adder](#)", IEEE Transactions on Computers, Vol.37, No.9, September 1988.

N. T. Quach, M. J. Flynn, "[High-Speed Addition in CMOS](#)", IEEE Transactions on Computers, Vol.41, No. 12, December, 1992.

S. Naffziger, "[A Sub-Nanosecond 0.5um 64b Adder Design](#)", Digest of Technical Papers, 1996 IEEE International Solid-State Circuits Conference, San Francisco, 8-10 Feb. 1996, p.362 -363.

S. Naffziger, "[High Speed Addition Using Ling's Equations and Dynamic CMOS Logic](#)", U.S. Patent No. 5,719,803, Issued:

(Refer Slide Time: 01:34:12)

ADDERS(CONT.)

Parallel Prefix Adders:

R. P. Brent and H. T. Kong, "[A Regular Layout for Parallel Adders](#)", IEEE Transactions on Computers, Vol. C-31, No.3, March 1982, p.260-264.

S. Knowles, "[A Family of Adders](#)", Proceedings of the 14th IEEE Symposium on Computer Arithmetic, Adelaide, Australia, April 14-16, 1999.

F. K. Gurkaynak, et al, "[Higher-Radix Kogge-Stone Parallel Prefix Adder Architectures](#)", Proceedings of IEEE International Symposium on Circuits and Systems, Geneva, Switzerland, May 28-31, 2000.

A. Farooqui, V. G. Oklobdzija, F. Chehraz, "[Multiplexer Based Adder for Media Signal Processing](#)", 1999 International Symposium on VLSI Technology, Systems, and Applications, Taipei, Taiwan, June 8-10, 1999.

(Refer Slide Time: 01:34:13)

V. G. Oklobdzija, Bart R. Zeydel, Hoang Dao, Sanu Mathew, Ram Krishnamurthy, "[Energy-Delay Estimation Technique for High-Performance Microprocessor](#)", Processing of the Symposium on Computer Arithmetic, 1063-6899, 2003.

Sanu Mathew, Mark Anders, Ram K. Krishnamurthy, Shekhar Borkar, "[A 4-GHz 130-nm Address Generation Unit With 32-bit Sparse-Tree Adder Core](#)", IEEE Journal of Solid-State circuits, Vol38, No.5, May 2003.

D. W. Dobberpuhl, et. al, "[A 200-MHz 64-b dual-issue CMOS Microprocessor](#)", IEEE Journal of Solid-State Circuits, Vol. 27, No. 11, November, 1992 P. 1555-1567.

J. Park, H. C. Ngo, J. A. Silberman, S. H. Dhong, "[470 ps 64-bit Parallel Binary Adder](#)", Digest of Technical Papers, 2000 Symposium on VLSI Circuits, 15-17 June Honolulu, 2000, p. 192 - 193.

Multi-Operand Addition:

P. Kornerup, "[Reviewing 4-2 Adders for Multi-Operand Addition](#)", Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors, ASAP'02, San Jose, California

(Refer Slide Time: 01:34:15)

MULTIPLIERS

C. S. Wallace, "[A Suggestion for a Fast Multiplier](#)", IEE Transactions on Electronic Computers, EC-13, p.14-17, 1964.

L. Dadda, "[Some Schemes for Parallel Multipliers](#)", Alta Frequenza, Vol.34, p.349-356, March 1965.

L. Dadda, "[On Parallel Digital Multipliers](#)", Reprinted from Alta Frequenza, Vol.45, p.574-580, 1976.

W. J. Stenzel, W. J. Kubitz, "[A Compact High-Speed Parallel Multiplication Scheme](#)", IEEE Transaction on Computers, C-26, p.948-957, 1977.

Irving T. Ho, Tien Chi Chen, "[Multiple Addition by Residue Threshold Functions and Their Representations by Array Logic](#)", IEEE Trans. on Computers, Vol. C-22, No. 8, pp. 762-767, August 1973.

(Refer Slide Time: 01:34:16)

Multipliers (Contd.)

D. Villeger and V. G. Oklobdzija, "[Analysis Of Booth Encoding Efficiency In Parallel Multipliers Using Compressors For Reduction Of Partial Products](#)", *Proceedings of the 27th Asilomar Conference on Signals, Systems and Computers*, pp. 781-784, 1993.

D. Villeger and V. G. Oklobdzija, "[Evaluation Of Booth Encoding Techniques For Parallel Multiplier Implementation](#)", *Electronics Letters*, Vol. 29, No. 23, pp. 2016-2017, 1993.

V. G. Oklobdzija and D. Villeger, "[Improving Multiplier Design By Using Improved Column Compression Tree And Optimized Final Adder In CMOS Technology](#)", *IEEE Transactions on VLSI Systems*, Vol.3, No.2, June, 1995.

Gary W. Bewick, "[Fast Multiplication: Algorithms and Implementation](#)", Ph.D. Thesis, Department of Electrical Engineering, Stanford University, February 1994.

V.G. Oklobdzija, D. Villeger, S. S. Liu, "[A Method for Speed Optimized Partial Product Reduction and Generation of Fast Parallel Multipliers Using and Algorithmic Approach](#)", *IEEE Transaction on Computers*, Vol.45, No.3, March 1996.

V. Oklobdzija, "[High-Speed VLSI Arithmetic Units: Adders and Multipliers](#)", in

(Refer Slide Time: 01:34:18)

Multipliers (Cont.)

P. Stelling, V. G. Oklobdzija, "Design Strategies for Optimal Hybrid Final Adders in a Parallel Multiplier", special issue on VLSI Arithmetic, Journal of VLSI Signal Processing, Kluwer Academic Publishers, Vol.14, No.3, December 1996.

P. Stelling, C. Martel, V. G. Oklobdzija, R. Ravi, "Optimal Circuits for Parallel Multipliers," IEEE Transaction on Computers, Vol. 47, No.3, pp. 273-285, March, 1998.

Sanu Mathew, Mark Anders, Ram K. Krishnamurthy, Shekhar Borkar, "A 4-GHz 130-nm Address Generation Unit With 32-bit Sparse-Tree Adder Core", IEEE Journal of Solid-State Circuits, Vol38, No.5, May 2003.

Division

J. E. Robertson, "A New Class of Digital Division Methods", IRE Trans. on Electronic Computers, Vol. EC-7, pp. 218-222, September 1958.

M. J. Flynn, "On Division by Functional Iteration", IEEE Transactions on Computers, C-19, p.702-706, 1970.

M. Ercegovic, "A Higher-Radix Division with Simple Selection of Quotient Digits", Proceedings of the 6th Symposium on Computer Arithmetic, Aarhus, Denmark, June 20 - 22, 1983.

J. Fadrianto, "Algorithm for High-Speed Shared Radix 4 Division and Radix 4 Square Root", Proceedings of the 8th Symposium on Computer Arithmetic, Como Italy, May 19-21, 1987.

V. G. Oklobdzija, "An Algorithmic and Novel Design of a Leading Zero Detector Circuit:

They are references on adders, they are references on the other huge numbers you can see, lot of actual this available for multipliers and this. With this part, we complete the total arithmetic operations for any processor or any hardware in this part of the advanced VLSI course. Some of the problems, which I gave during this all I will add at the end of them they will my model problem as we already solve them.

Some problems I will add to it later, which you can solve. Many times, most of these problems can be only solve on what I would say on the using this spice. So, you must have at least the initial version of spice, if you have cadence tools or Synopsys tools or tools or model if you have the mentor graphic tools, you have a good spice available on it, you can choose any of the hardware shown here for any given technology.

You can try implementing many of those blocks in your real system design and verify which ones, which I have actually have given you as a hint to take whether they work. Thank you very much for the day.