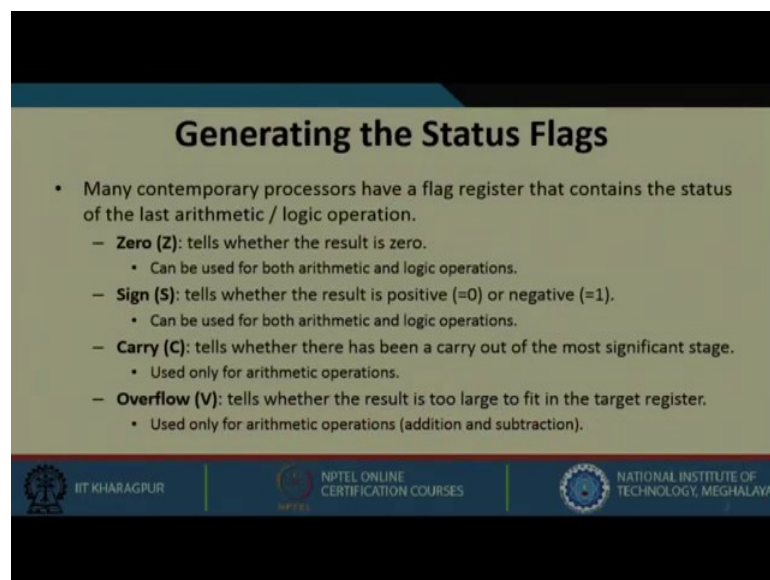


Computer Architecture and Organization
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 35
Design of Multipliers (Part I)

In the last couple of lectures we had looked at various kinds of adders. So, how they can be designed, and some comparative study among them. In this lecture first we shall be looking at some aspect about addition which we have not talked about yet; namely generating so called conditional flags, and then we shall be moving on to design of multipliers.

(Refer Slide Time: 00:57)



Generating the Status Flags

- Many contemporary processors have a flag register that contains the status of the last arithmetic / logic operation.
 - **Zero (Z)**: tells whether the result is zero.
 - Can be used for both arithmetic and logic operations.
 - **Sign (S)**: tells whether the result is positive (=0) or negative (=1).
 - Can be used for both arithmetic and logic operations.
 - **Carry (C)**: tells whether there has been a carry out of the most significant stage.
 - Used only for arithmetic operations.
 - **Overflow (V)**: tells whether the result is too large to fit in the target register.
 - Used only for arithmetic operations (addition and subtraction).

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

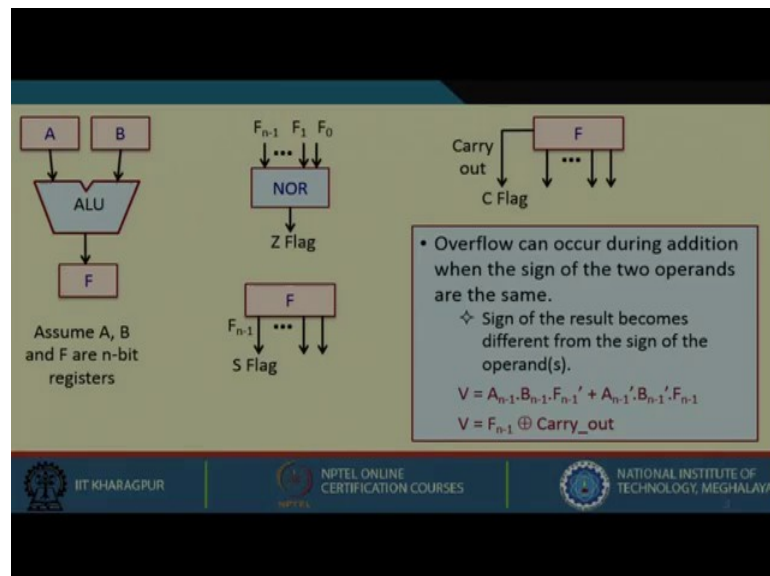
In this lecture we shall be starting with generating the status flags. Now what is a status flag? In many conventional processors if you look at the instruction set architecture, we will find that they have some specific condition flags like sign flag, zero flag, overflow flag, and so on. These flags are automatically set as a result of some arithmetic or logic operations; like for example, you can do an addition, and after addition you can compare whether the result is negative or positive. This you can do by checking whether the sign flag is 0 or 1; or you can check if the result is 0, this you can do by checking whether the zero flag is 0 or 1.

There are a set of flip-flops or one bit registers. The flags they are automatically set or reset depending on the result of some previous operation. So, after that you can use a conditional branch instruction; for example, to check the status of the flag and take some decision accordingly.

We shall be talking about four of the commonly use flags, then several other flags also used in some machine; this Z flag tells whether the result is 0 or not. Normally this flag is set by both arithmetic and logic operations; S flag tells whether the result is positive or negative. So, for 2's complemented representation, a 0 will mean positive; 1 will mean negative.

That means, the most significant bit of the result is the same as the S flag, and this flag is set for both arithmetic and logic operations. Typically carry (C) already we have seen in adder; how carry is generated from one stage to another. This flag will tell whether there has been a carry out of the final most significant stage. And this carry information is used only for arithmetic operations. And lastly there is a flag called overflow (V) that tells whether the result is too large to fit in the register. It is used for arithmetic operations, typically addition and subtraction sometimes also multiplication.

(Refer Slide Time: 04:05)

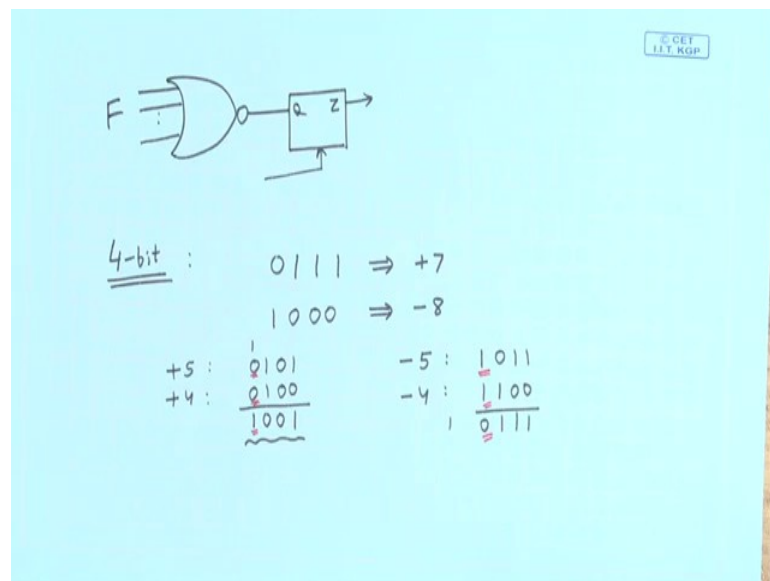


We will see these things one-by-one; so generation of this flags is not difficult rather easy. Let us consider very simple typical scenario like this, where we have an arithmetic logic unit which takes as inputs from two registers A and B, and the result is stored into

another register F. Let us assume all of them are n bit registers. Firstly, the Z flag can be very easily generated by using a single n-input NOR gate. So, what is the NOR gate? The output of NOR gate will be 1 if all the input are 0. So, if all the bits of the result are 0, the output NOR gate will be 1; that will indicate the Z flag.

So, actually the circuit will be like this, you have a large NOR gate.

(Refer Slide Time: 05:03)



So, the input from the result register F is fed here, and the output here goes to a flip flop. This is a Z flip flop, let us call this input Q. So, this will be loaded by some control signal and once loaded, this Z flag will be available. Similarly the S flag can be generated directly from the most significant bit of the result; just take this F_{n-1} and feed it to S flip flop. So, generating this sign flag is also very easy.

This ALU will also be having a carry out signal, that carry out signal will be directly going into the C flag flip flop. Now talking about overflow, see here we are talking about overflow during addition and subtraction. Overflow can occur during addition only when the sign of the two operands are the same. Now if one of the number is positive and the other number is negative you can never have an overflow. Let us take a very simple example of a 4 bit representation.

So, in 4 bit in the positive side I can represent maximum 0111, which means +7, in the negative side I can represent 1000 which means -8. So, let us say I am adding +5 and +4.

Now quite naturally some of the numbers will not fit in 4 bits because the maximum you can store is 7.

So, one observation is the sign of the numbers is positive, but the result has become negative. You look for two negative numbers, same thing will happen. Let us take -5 and -4. Now here also we see that the sign bit of the original numbers were 1, but the sign bit of the result has become different. This is the condition for overflow detection.

Now one thing we are not considering overflow for division operation at present. Because in multiplication normally when we add two n bit numbers the product is stored in a $2n$ bit register.

(Refer Slide Time: 09:46)

- The MIPS architecture does not have any status flags.
- Why?
 - MIPS ISA is designed for efficient pipeline implementation.
 - Several instructions can be in various stages of execution in the pipeline.
 - Flag registers result in *side effects* among instructions.
- MIPS stores information about the flags temporarily in a GPR.

```
slt $t0, $s1, $s2
beq $t0, $zero, Label
```

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

So, there can never be an overflow there. So, for that purpose for the time being we are assuming that overflow is important only for addition and subtraction, and not for multiplication.

Now, again talking of the flag registers, the MIPS architecture that we have been studying does not have any status flags at all. Why? Because the instruction set of MIPS was designed primarily for very efficient pipeline implementations. Now we will be studying this in much more detail later, that when there are several instructions which are running in a pipeline they are in various stages of execution. Now if there is a single set of flag register there can be confusion; the first register may be setting the Z flag, the

second register may also be trying to set the Z flag. So, there can be some confusion. So, for a pipeline implementation existence of the flags create something called side effects, which are undesirable.

So, MIPS uses an entirely different philosophy. They do not use any status flags; rather they temporarily stored flag information in a general purpose register. I am showing simple example; there is an instruction called set less than (SLT). What it does is it will check if S1 is less than S2 or not? If so, then the target register will be set to 1; otherwise this target register will be set to 0. So, you are storing either a 0 or a 1 in the target register. So, as if the whole target register your are using as a flag, and immediately after that you can have an instruction like a BEQ, where you are checking whether the result is 0 or not; 0 means this was not less than; then you jump to a label.

So, earlier what you do? Earlier you do branch if not zero jump to a label, but now since you do not have a flag register you have to use one of these set instructions first, then use a conditional branch instruction checking the value of the target register whether is 0 or nonzero.

(Refer Slide Time: 12:22)

Multiplication of Unsigned Numbers

- Multiplication requires substantially more hardware than addition.
- Multiplication of two n-bit number generates a 2n-bit product.
- We can use shift-and-add method.
 - Repeated additions of shifted versions of the multiplicand.

1 0 1 0	Multiplicand M (10)
1 1 0 1	Multiplier Q (13)

1 0 1 0	
0 0 0 0	
1 0 1 0	
1 0 1 0	

1 0 0 0 0 0 1 0	Product P (130)

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

Let us now come to multiplication. Talking about multiplication of unsigned numbers the first thing is that the amount of hardware required for multiplication is substantially greater as compared to addition. Multiplication of two n bit numbers can generate 2n bit product. We are following a method that is very similar to the pen and pencil method we

are familiar with. So, when you multiply two numbers using this so-called decimal number system, we follow a very similar approach. So, what we do? Let us say this is our multiplicand this is our multiplier all expressed in binary; they are unsigned numbers, not 2's complement.

So, I check the first digit of the multiplier. We multiply 1 by this, you get 1 0 1 0, then shift one place left; multiply by 0 with this we get 0 0 0 0; multiply by 1 with this shift, multiply 1 with this shift, then finally you add all the bits up. So, the product is 130; you see that the product requires 8 bits. So, this one example shows that when you multiply two 4 bit numbers, your result may become double of that --- 8 bits.

So, the product can be $2n$ bits, and this method have a addition is called shift and add. So, we do repeated additions of shifted versions of the multiplicand; because here you are only multiplying by 1 or 0. So, one may this same multiplicand will be appearing 1 0 1 0 1 0 1 0 1 0 after various number of shifting. So, either we can write down all these partial products and then add them of together, or you can continuously go on adding as the next one is generated.


(Refer Slide Time: 15:05)

A General Case


A_3	A_2	A_1	A_0	
B_3	B_2	B_1	B_0	

A_3B_0	A_2B_0	A_1B_0	A_0B_0	
A_3B_1	A_2B_1	A_1B_1	A_0B_1	
A_3B_2	A_2B_2	A_1B_2	A_0B_2	
A_3B_3	A_2B_3	A_1B_3	A_0B_3	


- Each A_iB_j is called a partial product.
- Generating the partial products is easy.
 - Requires just an AND gate for each partial product.
- Adding all the n -bit partial products in hardware is more difficult.



IIT KHARAGPUR



NPTEL ONLINE
CERTIFICATION COURSES



I am writing this in a very general term; this A_i and B_i can be 0 or 1. So, B_0 it can be 0 or 1 is multiplied to all this 4 bits. $B_0.A_0$, $B_0.A_1$, $B_0.A_2$, $B_0.A_3$. Similarly with B_1 , B_2 and B_3 . Each of these products we have shown here is called a partial product; now for adding to 4 bit numbers if you can just check there are 16 partial products.

So, for multiplying two n-bit numbers, there will n^2 partial products. To generate every partial product you need just an AND gate.

(Refer Slide Time: 16:37)

Design of a Combinational Array Multiplier

- We can directly map the multiplication process as discussed to hardware.
 - We use an array of cells to generate the partial products.
 - Instead of adding the partial products at the end, we add the partial products at every stage of the multiplication.
- The required multiplication cell is as shown.
 - Combines capabilities of partial product generation and also addition of partial products.

The diagram shows a logic circuit for a multiplication cell. It consists of an AND gate at the top, which takes two inputs from the left and produces a single output. Below the AND gate is a Full Adder block. The output of the AND gate is connected to the top input of the Full Adder. The Full Adder also has two other inputs from the left and produces two outputs: one to the right and one downwards. The slide also features logos for IIT KHARAGPUR and NPTEL ONLINE CERTIFICATION COURSES, and a small inset video of a speaker.

Products generated by AND gates, you add them in a suitably. You will be needing a complex adder circuit to add so many partial products together. So, you need a lot of hardware.

From this basic principle let us try to come up with the simple hardware implementation of an adder; this is called a combinational array multiplier. So, what we are doing? We are using something called a multiplication cell. So, whatever the process we just now shown here shift and add generating the partial products, you are directly trying to map it in hardware. We are using an array of cells like this.

We are not waiting till the end to add all the partial products, we are adding the partial product at every stage. This AND gate generates the partial products and this full adder will be adding this partial product with another partial product coming from the previous stage after shifting, and this full adder will be getting a carry input it will generating a carry output, and this will be a generating a SUM.

(Refer Slide Time: 18:06)

- Extremely inefficient, and requires very large amount of hardware.
- n^2 multiplication cells for an $n \times n$ multiplier.
- Advantage is that it is very fast.

Product: $p_7 p_6 p_5 p_4 p_3 p_2 p_1 p_0$

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

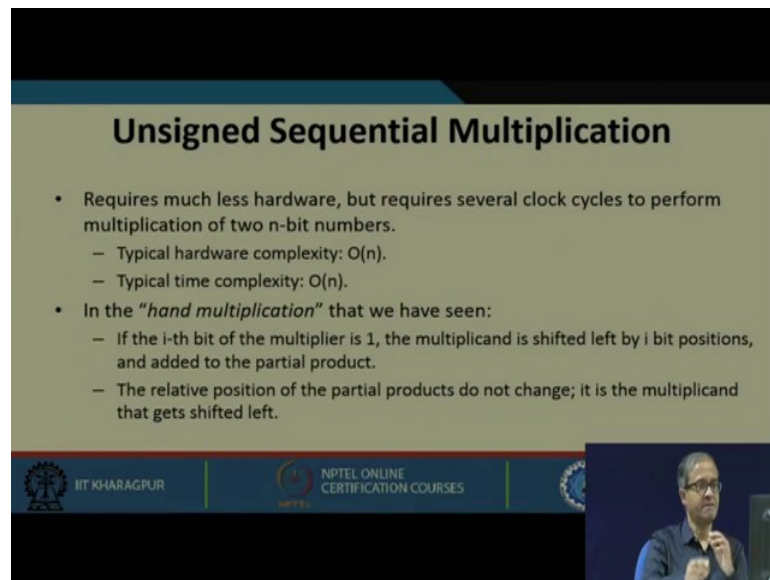
It will clear if you see this schematic diagram how it is done. Here for 4×4 multiplication I am showing, there will be 16 such cells. So, diagonally from the right the bits of the multiplicands are coming m_0, m_1, m_2, m_3 ; that means, same bit is also coming here going to the next stage. So, the m_0 is also going to the next stage, m_1 is also going to the next stage, m_2 and m_3 , and from the other side the quotient bits are applied q_0, q_1, q_2, q_3 it is from this side.

So, the quotient and the multiplier if you just AND them, you will getting the partial products; so that partial products are getting added you have a ripple carry adder. These full adders are all connected in cascade. So, initially the input is $0\ 0\ 0\ 0\ 0$; if first partial product is getting added to 0 you get the partial product here.

Next one: so the way this multiplier is designed there is automatically a one bit left shift you can see. Next stage is computing the partial products can adding to the previous one, next one is computing a next partial product after again one bit shift adding to the previous one, in this why it goes on. So, finally, at the end whatever you get will be the product.

This method is very simple; whatever you are doing by hand you have directly mapped it to hardware, but the problem is that this kind of a multiplier is extremely inefficient, it requires extremely large amount of hardware; that means, you need n^2 such cells, but the advantage is that it is much faster.

(Refer Slide Time: 20:36)



Unsigned Sequential Multiplication

- Requires much less hardware, but requires several clock cycles to perform multiplication of two n -bit numbers.
 - Typical hardware complexity: $O(n)$.
 - Typical time complexity: $O(n)$.
- In the “*hand multiplication*” that we have seen:
 - If the i -th bit of the multiplier is 1, the multiplicand is shifted left by i bit positions, and added to the partial product.
 - The relative position of the partial products do not change; it is the multiplicand that gets shifted left.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

So, we have seen the multiplication method that is called combinational array multiplier. So, it is a combinational circuit. After some delays the result will be generated. Now we look at some multiplication methods which will be requiring much less amount of hardware, but will be sequential in nature; means you will have to run the circuit for several clock cycles, to complete the multiplication.

So, let us look at unsigned sequential multiplication first. So, exactly what we have said is that requires much less hardware, but requires several clock cycles. Much less hardware means what? Earlier for combinational array multiplier we are using n^2 cells. So, the hardware complexity was $O(n^2)$, but here we are saying would be needing hardware which will be proportional to the number of bits being multiplied and also the time require you also be $O(n)$.

Now, in hand multiplication what we have seen? You have seen that if the bit of the multiplier is 1, we shift the multiplicand by one bit position, and add to the partial product like I am just showing an example here.

(Refer Slide Time: 22:10)

The image shows a handwritten binary multiplication on a light blue background. At the top right, there is a small logo for 'CET IIT KGP'. The multiplication is as follows:

$$\begin{array}{r} 0101 \\ \times 1011 \\ \hline 0101 \\ 0101 \leftarrow \\ 01111 \\ 0000 \leftarrow \\ 001111 \\ 0101 \leftarrow \\ \hline 0110111 \end{array}$$

Let us multiply 0 1 0 1 with 1 0 1 1. First with this one line multiply, this becomes 0 1 0 1, then with this one and multiply with one shift 0 1 0 1 I; immediately add them then comes this 0 0 means he again shift and add 0 just immediately, add them, last 1 0 1 0 one just again another shift 0 1 1 0, this is your final product.

So, you are shifting multiplicand by variable amount, but the relative position of the partial product you are not changing. The bit positions are changing, kept in the same position, but for actually implementation we will be doing something else, instead of doing this will be making a small change.

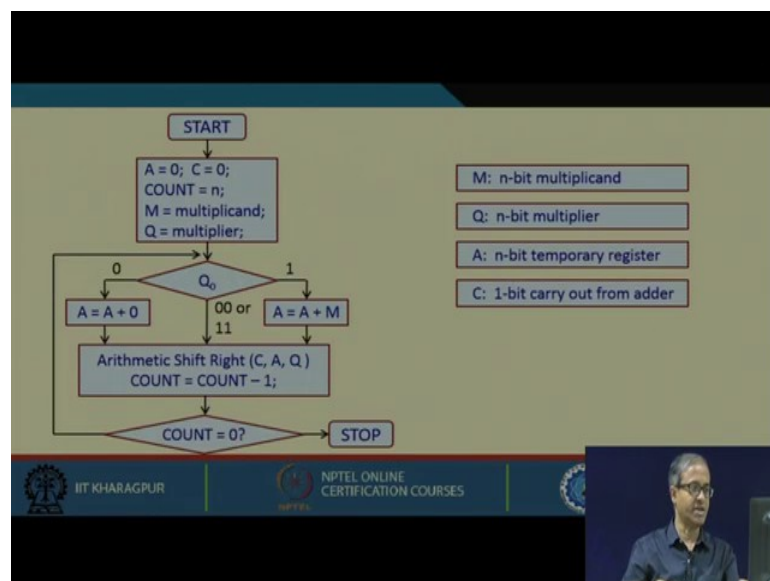
(Refer Slide Time: 23:33)

- In the “*shift-and-add*” multiplication that we discuss now, we make the following modifications.
 - We do not shift the multiplicand (i.e., keep its position fixed).
 - We right shift an $2n$ -bit partial product at every step.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

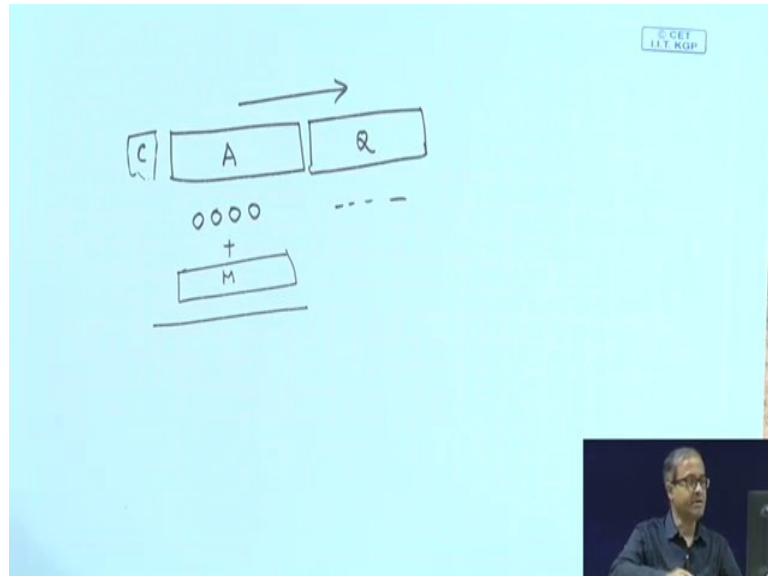
We do not shift the multiplicand, will keep the position fixed; rather the partial product will be shifting right at every step. Like here you are keeping the partial product same and shifting the multiplicand left every step, what was saying will keep the multiplicand in the same place partial product will be shifting right. So, these two are equivalent. So, either you keep this in the same place and shift this left, or keep this in the same place and shift this right. So, we are following the second order because it is easier to implement in hardware.

(Refer Slide Time: 24:15)



This shift and add multiplication the overall steps in the form of flow chat is shown here. See here the idea is as follows we are computing something like a partial product.

(Refer Slide Time: 24:40)



You see we are using some registers A, M and Q. Normally registers A and Q are consider together; initially A will be loaded with all zeros and Q will be loaded with the quotient. We will be shifting this to the right, and whenever we have to add the multiplicand will be adding here, the M will be added into this position and this whole thing will be shifting right. This will be the principle we will be following.

So, we start by initializing this A register to 0, C is a carry bit = 0, count is the number of times n, M is multiplicand, Q is multiplier. So, I am assuming M and Q are both n-bit., A is a n-bit temporary register initialized to 0, and C is the carry out from an adder.

So, what we do we check the last bit of the quotient Q_0 . If it is 1, we have to add; if it is 0, we are adding A_0 right in the shift and add method. So, you look at the last bit Q_0 . If the Q_0 bit is 1, you add the multiplicand to A. So, C A Q together will be shifting right. So, we are doing an arithmetic shift right C A Q, and then decrementing the count by 1. We are checking whether count is 0 or not if it is not 0 we go back, if it is 0 we stop.

(Refer Slide Time: 27:13)

	C	A	Q		
Example 1: $(10) \times (13)$	0	0 0 0 0 0	0 1 1 0	1	Initialization
Assume 5-bit numbers.					
M: $(01010)_2$	0	0 1 0 1 0	0 1 1 0 1		$A = A + M$
Q: $(01101)_2$	0	0 0 1 0 1	0 0 1 1	0	Shift
Product = 130	0	0 0 1 0 1	0 0 1 1 0		$A = A + 0$
= $(001000010)_2$	0	0 0 0 1 0	1 0 0 1	1	Shift
	0	0 1 1 0 0	1 0 0 1 1		$A = A + M$
	0	0 0 1 1 0	0 1 0 0	1	Shift
	0	1 0 0 0 0	0 1 0 0 1		$A = A + M$
	0	0 1 0 0 0	0 0 1 0	0	Shift
	0	0 1 0 0 0	0 0 1 0 0		$A = A + 0$
	0	0 0 1 0 0	0 0 0 1 0		Shift

I am illustrating with help of an example. Let us consider two 5 bit numbers, 10 and 13. 10 in 5 bits is this, 13 in 5 bits is this. So, the product is expected to be 130. So, initially my A is 0, Q contains the quotient 0 1 0 1 0 1, and the carry bit is 0. So, what we do is check whether Q is 0 or 1; I say it is 1. So, I have to add multiplicand M to this. So, I do $A = A + M$. This continues.

We have to check the bits one by one. So, again I check this, this is 0; so now, will have to add 0. So, there is no change in A, and again we do a shift right. So, the next bit of the quotient comes here, 1 you add M1 again to it. So, 0 1 0 1 0 plus 0 0 0 1 0 it becomes 0 1 1 0 0 this you can check and you do again you do a shift. So, this one comes here finally. So, you check this. So, you again add M to this 0 0 1 1 0; it becomes 1 0 0 0 0. So, again do a shift last time we check this bit the last bit 0. So, you add 0 no change in shift this is your final result.

So, you see the quotient which I had add loaded initially in this register has slowly shifted out, and is replaced by the product finally. So, at the end I had made 5 shifts. So, the quotient bits have gone out. So, the result does not contain the quotient bits any more. So, after addition shifting, addition shifting, addition shifting they will slowly fill up this whole 10 bit register, and this is the product.

(Refer Slide Time: 29:48)

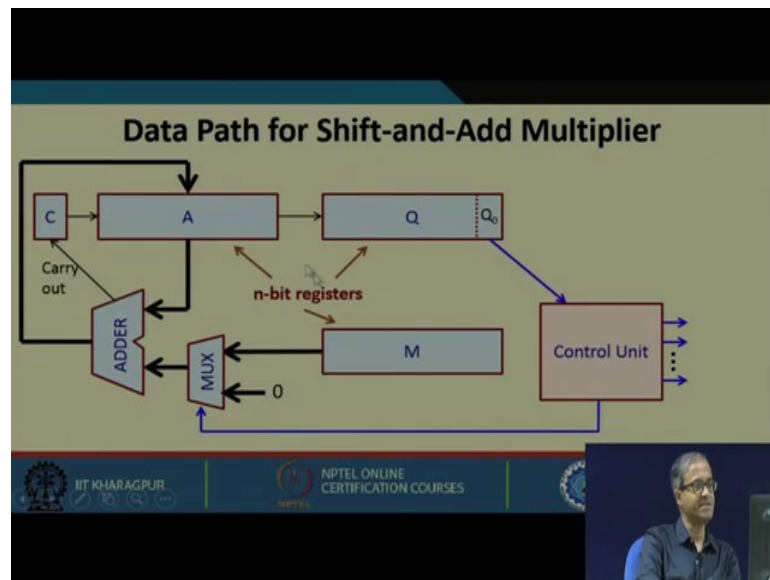
	C	A	Q		
Example 2: (29) x (21)					
Assume 5-bit numbers.					
M: (11101) ₂					
Q: (10101) ₂					
Product = 609					
= (1001100001) ₂					
	0	0 0 0 0 0	1 0 1 0	1	Initialization
	0	1 1 1 0 1	1 0 1 0 1		A = A + M
	0	0 1 1 1 0	1 1 0 1	0	Shift
	0	0 1 1 1 0	1 1 0 1 0		A = A + 0
	0	0 0 1 1 1	0 1 1 0	1	Shift
	1	0 0 1 0 0	0 1 1 0 1		A = A + M
	0	1 0 0 1 0	0 0 1 1	0	Shift
	0	1 0 0 1 0	0 0 1 1 0		A = A + 0
	0	0 1 0 0 1	0 0 0 1	1	Shift
	1	0 0 1 1 0	0 0 0 1 1		A = A + M
	0	1 0 0 1 1	0 0 0 0 1		Shift

Now, in this example the carry bit was never 1. Let us take another example where the carry bit can be 1. Let say we multiply 29 and 21; these are the two numbers the product is supposed to be 609. This same way you look at one you add the multiplicand 1 1 1 0 1 then do a right shift next bit is 0. So, you add 0 no change then do a right shift 1.

So, after shift this carry will get shifted in here right then you check again the next bit 0 no addition; that means, you are adding 0, no change again shift next bit is 1, the last bit you again add M to it, you again get a carry out of 1, again shift right this one will again go in and this will be your last and final result 609.

So, this is how the basic shift can add multiplication works.

(Refer Slide Time: 31:05)



Now talking about the hardware circuit for (Refer Time: 31:12) it is very simple. What we shown in the example we need exactly that. We need a register for A, we need a register for Q, we need a carry flip flop, and they will be connected as a shift register, we need an adder with one of the inputs coming from M, and the other input either coming from the multiplicand or 0.

And are all n bit registers A Q and M; the last bit of Q is Q_0 , and there will be a control unit that will be checking Q_0 at every step. And will be selecting a multiplexer appropriately; either it is selecting 0 or M. And also it will be generating control signal for the other part. So, means after addition the shifting will be going on, it will keep track of how many times it will be repeating. So, control unit will be doing that. The data path as you can see is very simple; other than the registers you need just an adder and a multiplexer, and just one flip flop, carry.

So, with this we come to the end of this lecture. In the next lecture, we shall be looking at some other methods of multiplication, particularly signed multiplication, how we can multiply two sign numbers and some improvements therein.

Thank you.