Computer Organization and Architecture

Module 5

Prof. Indranil Sengupta

Dr. Sarani Bhattacharya

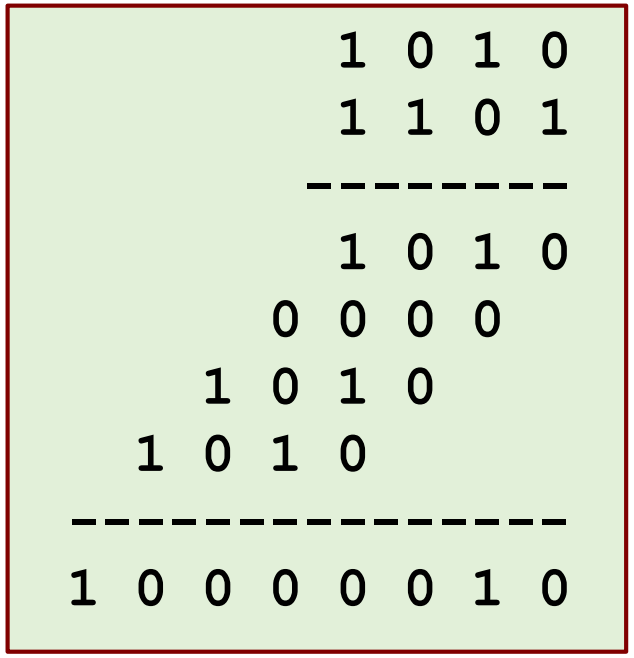
Department of Computer Science and Engineering

IIT Kharagpur

Multiplication

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.



Multiplicand M (10)
Multiplier Q (13)

Product P (130)

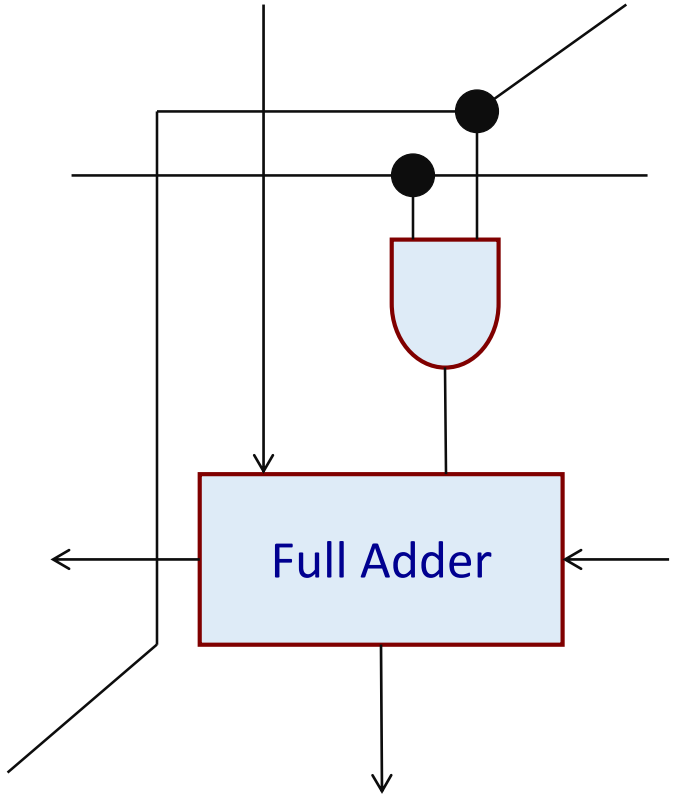
A General Case

$$
\begin{array}{cccccccc}
 & & & A_3 & A_2 & A_1 & A_0 & \\
 & & & B_3 & B_2 & B_1 & B_0 & \\
\hline
 & & & A_3B_0 & A_2B_0 & A_1B_0 & A_0B_0 & \\
 & & A_3B_1 & A_2B_1 & A_1B_1 & A_0B_1 & & \\
 & A_3B_2 & A_2B_2 & A_1B_2 & A_0B_2 & & & \\
A_3B_3 & A_2B_3 & A_1B_3 & A_0B_3 & & & & \\
\hline
\end{array}
$$

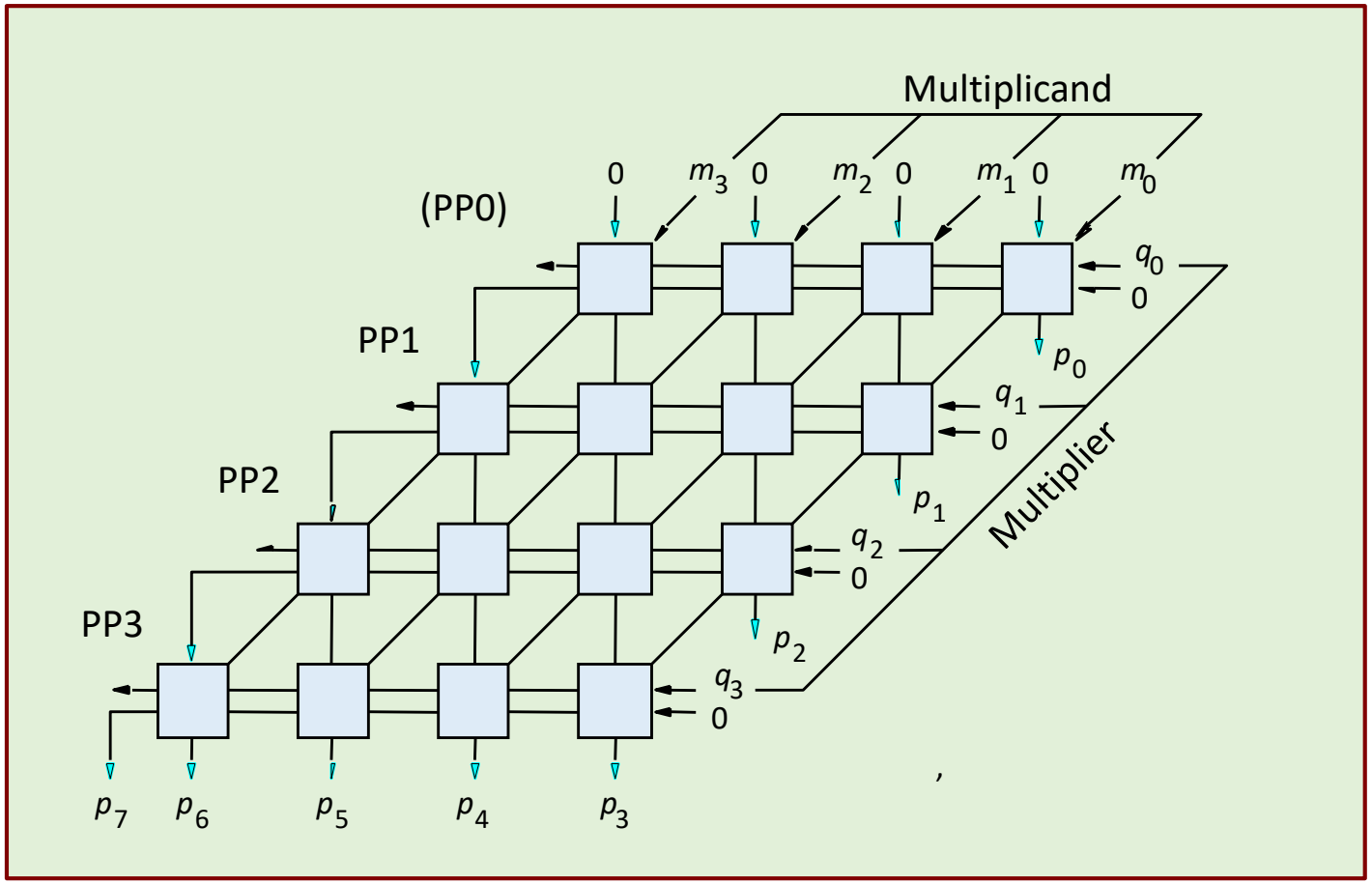
- Each A_iB_j is called a partial product.
- Generating the partial products is easy.
  - Requires just an AND gate for each partial product.
- Adding all the n-bit partial products in hardware is more difficult.

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.



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

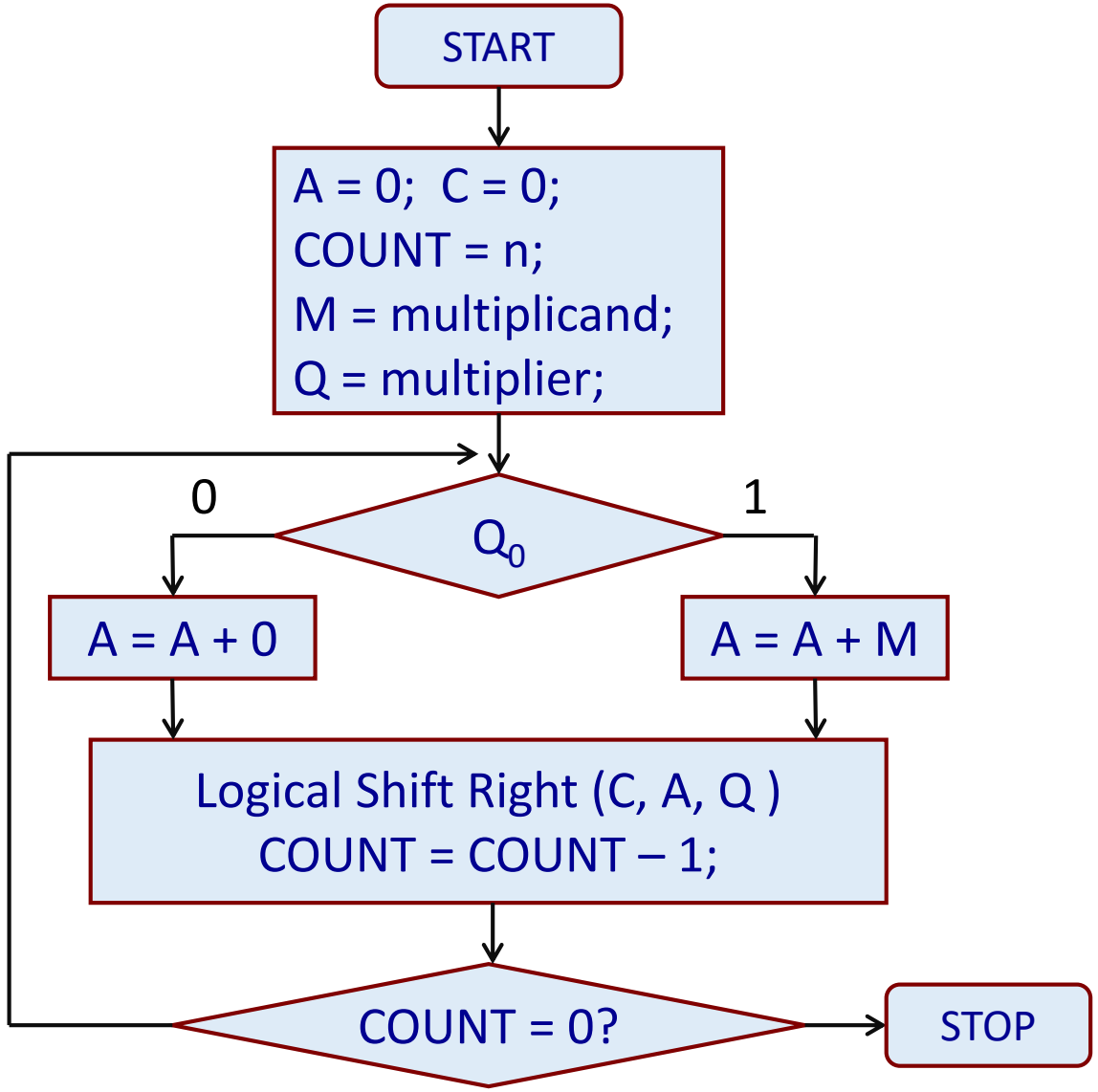


Product: $p_7\ p_6\ p_5\ p_4\ p_3\ p_2\ p_1\ p_0$

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.

- 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.



M: n-bit multiplicand

Q: n-bit multiplier

A: n-bit temporary register

C: 1-bit carry out from adder

Example 1: (10) x (13)

Assume 5-bit numbers.

M: $(0\ 1\ 0\ 1\ 0)_2$
Q: $(0\ 1\ 1\ 0\ 1)_2$

Product = 130
= $(0\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 1\ 0)_2$

	C	A	Q		
	0	0 0 0 0 0	0 1 1 0 **1**	Initialization	
	0	0 1 0 1 0	0 1 1 0 1	A = A + M	Step 1
	0	0 0 1 0 1	0 0 1 1 **0**	Shift	
	0	0 0 1 0 1	0 0 1 1 0	A = A + 0	Step 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	Step 3
	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	Step 4
	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	Step 5
	0	0 0 1 0 0	0 0 0 1 0	Shift	

Example 2: (29) x (21)

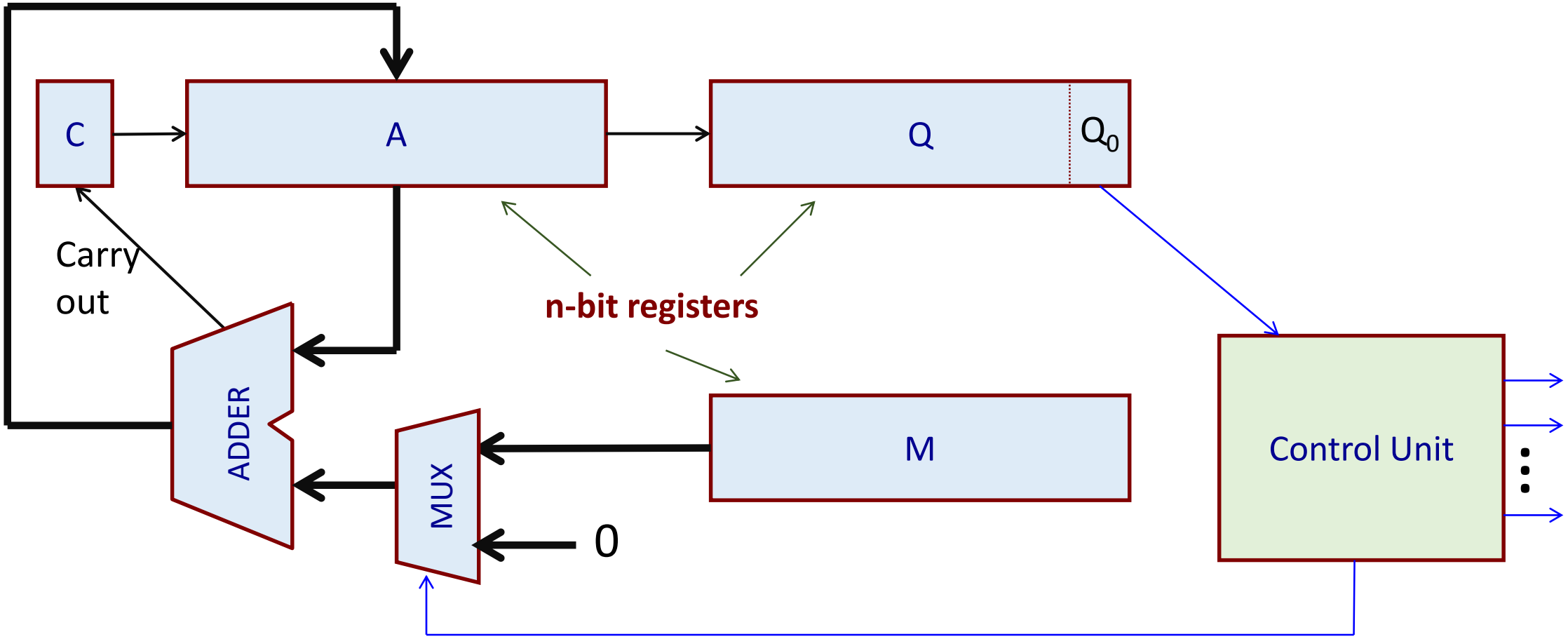
Assume 5-bit numbers.

M: $(1\ 1\ 1\ 0\ 1)_2$
Q: $(1\ 0\ 1\ 0\ 1)_2$

Product = 609
= $(1\ 0\ 0\ 1\ 1\ 0\ 0\ 0\ 0\ 1)_2$

	C	A	Q		
	0	0 0 0 0 0	1 0 1 0 (1)	Initialization	
Step 1	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	
Step 2	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	
Step 3	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	
Step 4	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	
Step 5	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	

Data Path for Shift-and-Add Multiplier



Signed Multiplication

- We can extend the basic shift-and-add multiplication method to handle signed numbers.
- One important difference:
  - Require to sign-extend all the partial products before they are added.
  - Recall that for 2's complement representation, sign extension can be done by replicating the sign bit any number of times.

0101 = 0000 0101 = 0000 0000 0000 0101 = 0000 0000 0000 0000 0000 0000 0000 0101

1011 = 1111 1011 = 1111 1111 1111 1011 = 1111 1111 1111 1111 1111 1111 1111 1011

An Example: 6-bit 2's complement multiplication

Note: For n-bit multiplication, since we are generating a 2n-bit product, overflow can never occur.

```
                1 1 0 1 0 1     (-11)
              x 0 1 1 0 1 0     (+26)
-------------------------------
0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 0 1 0 1
0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 0 1 0 1
1 1 1 1 0 1 0 1
0 0 0 0 0 0 0
-------------------------------
1 1 1 0 1 1 1 0 0 0 1 0    (-286)
```

Booth's Algorithm for Signed Multiplication

- In the conventional shift-and-add multiplication as discussed, for n-bit multiplication, we iterate n times.
  - Add either 0 or the multiplicand to the 2n-bit partial product (depending on the next bit of the multiplier).
  - Shift the 2n-bit partial product to the right.
- Essentially we need *n additions and n shift operations*.
- Booth's algorithm is an improvement whereby we can avoid the additions whenever consecutive 0's or 1's are detected in the multiplier.
  - Makes the process faster.

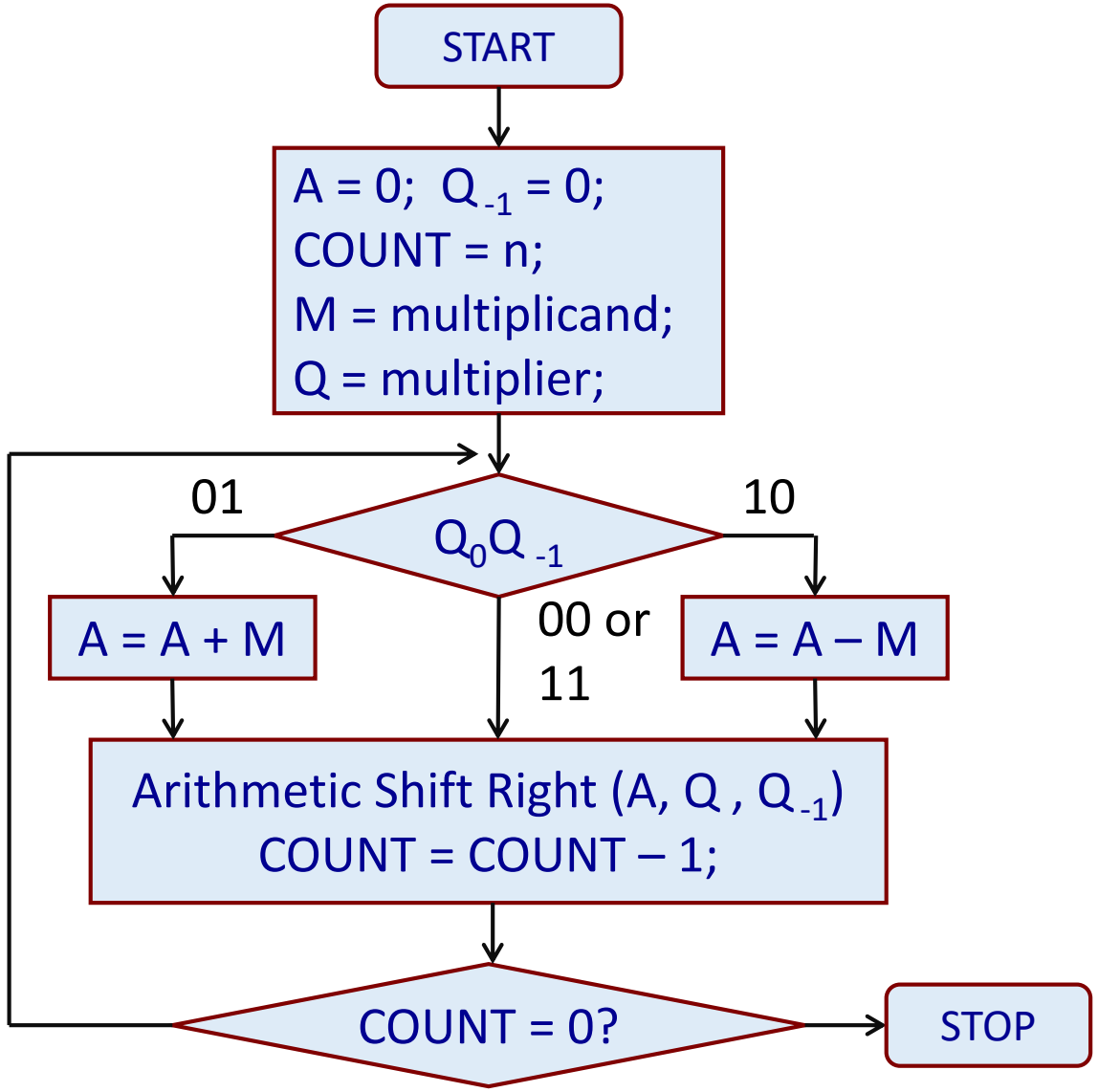
Basic Idea Behind Booth's Algorithm

- We inspect two bits of the multiplier (Q_i, Q_{i-1}) at a time.
  - If the bits are same (00 or 11), we only shift the partial product.
  - If the bits are 01, we do an addition and then shift.
  - If the bits are 10, we do a subtraction and then shift.
- Significantly reduces the number of additions / subtractions.
- Inspecting bit pairs as mentioned can also be expressed in terms of *Booth's Encoding*.
  - Use the symbols +1, -1 and 0 to indicate changes w.r.t. Q_i and Q_{i-1}.
  - 01 → +1, 10 → -1, 00 or 11 → 0.
  - For encoding the least significant bit Q_0, we assume $Q_{-1} = 0$.

- Examples of Booth encoding:

a)	0 1 1 1 0 0 0 0	::	+1 0 0 -1 0 0 0 0
b)	0 1 1 1 0 1 1 0	::	+1 0 0 -1 +1 0 -1 0
c)	0 0 0 0 0 1 1 1	::	0 0 0 0 +1 0 0 -1
d)	0 1 0 1 0 1 0 1	::	+1 -1 +1 -1 +1 -1 +1 -1

- The last example illustrates the *worst case* for Booth's multiplication (alternating 0's and 1's in multiplier).
  - In the illustrations, we shall show the two multiplier bits explicitly instead of showing the encoded digits.



M: n-bit multiplicand

Q: n-bit multiplier

A: n-bit temporary register

Q_{-1}: 1-bit flip-flop

Skips over consecutive 0's and 1's of the multiplier Q.

Example 1: (-10) x (13)

Assume 5-bit numbers.

M: $(1\ 0\ 1\ 1\ 0)_2$
-M: $(0\ 1\ 0\ 1\ 0)_2$
Q: $(0\ 1\ 1\ 0\ 1)_2$

Product = -130
$= (1\ 1\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 0)_2$

A	Q	Q_{-1}		
0 0 0 0 0	0 1 1 0	1	0	Initialization	
0 1 0 1 0	0 1 1 0	1	0	A = A - M	Step 1
0 0 1 0 1	0 0 1 1	0	1	Shift	
1 1 0 1 1	0 0 1 1	0	1	A = A + M	Step 2
1 1 1 0 1	1 0 0 1	1	0	Shift	
0 0 1 1 1	1 0 0 1	1	0	A = A - M	Step 3
0 0 0 1 1	1 1 0 0	1	1	Shift	
0 0 0 0 1	1 1 1 1	0	1	Shift	Step 4
1 0 1 1 1	1 1 1 0	0	1	A = A + M	
1 1 0 1 1	1 1 1 1	0	0	Shift	Step 5

Example 2:

(-31) x (28)

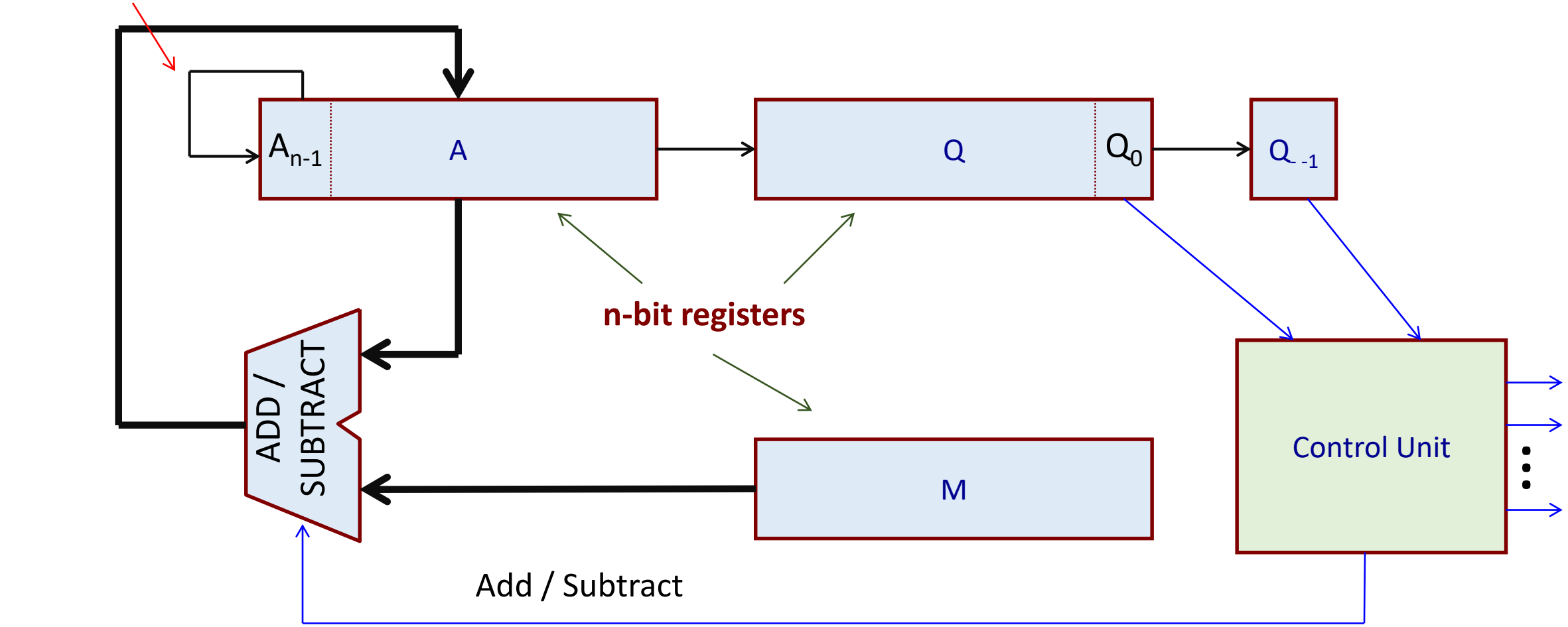
Assume 6-bit numbers.

M: $(1\,0\,0\,0\,0\,1)_2$
-M: $(0\,1\,1\,1\,1\,1)_2$
Q: $(0\,1\,1\,1\,0\,0)_2$

Product = -868
= $(1\,1\,0\,0\,1\,0\;\;0\,1\,1\,1\,0\,0)_2$

A	Q	Q_{-1}		
0 0 0 0 0 0	0 1 1 1 0	0	Initialization	
0 0 0 0 0 0	0 0 1 1 1	0	Shift	Step 1
0 0 0 0 0 0	0 0 0 1 1	1	Shift	Step 2
0 1 1 1 1 1	0 0 0 1 1 1	0	A = A - M	Step 3
0 0 1 1 1 1	1 0 0 0 1	1	Shift	
0 0 0 1 1 1	1 1 0 0 0	1	Shift	Step 4
0 0 0 0 1 1	1 1 1 0 0	0	Shift	Step 5
1 0 0 1 0 0	1 1 1 0 0 0	1	A = A + M	Step 6
1 1 0 0 1 0	0 1 1 1 0 0	0	Shift	

Data Path for Booth's Algorithm



Design of Fast Multiplier

a) Bit-Pair Recoding of Booth's Multiplication

- A technique that halves the maximum number of summands; derived directly from the Booth's algorithm.
- If we group the Booth-coded multiplier digits in pairs, we observe:
  - (+1, -1): (+1, -1) * M = 2 * M – M = M
  - (0, +1): (0, +1) * M = M
- We need a single addition instead of a pair of addition & subtraction.
  - Other similar rules can be framed.
  - Shown on next slide.

Original Booth-coded Pair	Equivalent Recoded Pair
(+1, 0)	(0, +2)
(-1, +1)	(0, -1)
(0, 0)	(0, 0)
(0, 1)	(0, 1)
(+1, 1)	--
(+1, -1)	(0, +1)
(-1, 0)	(0, -2)

- Every equivalent recoded pair has at least one 0.
- Worst-case number of additions or subtractions is 50% of the number of multiplier bits.
- Reduces the worst-case time required for multiplication.

Example: (+13) X (-22) in 6-bits.

```
Original:  Multiplier --   1   0   1   0   1   0
Booth:     Multiplier --  -1  +1  -1  +1  -1   0
Recoded:   Multiplier --   0  -1   0  -1   0  -2
```

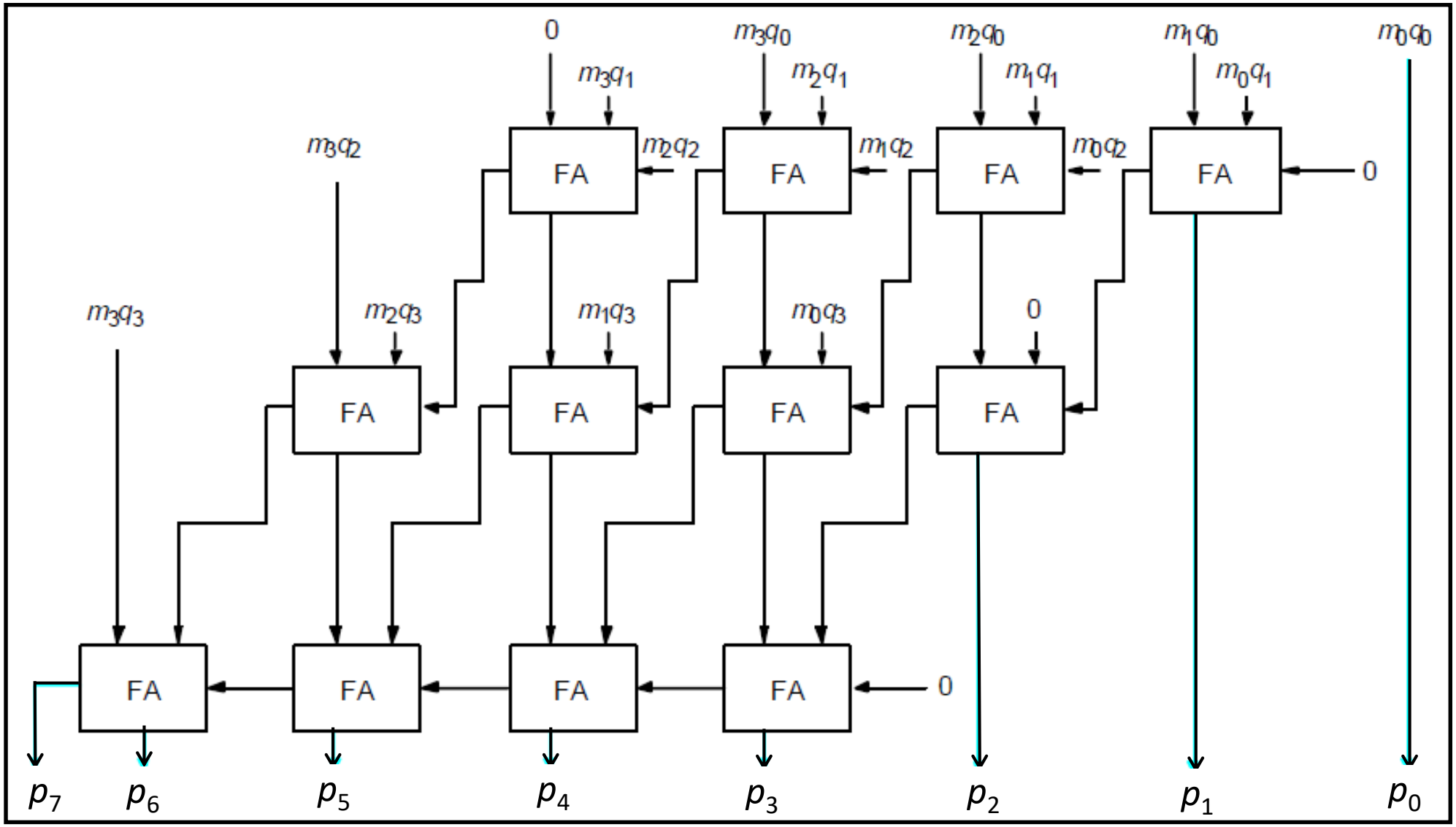
```
          0   0   1   1   0   1
        . -1  . -1  . -2
-------------------------------
  1 1 1 1 1 1 1 0 0 1 1 0
  1 1 1 1 1 1 0 0 1 1
  1 1 1 1 0 0 1 1
-------------------------------
  1 1 0 1 1 1 1 0 0 0 1 0
```

- M = 0 0 1 1 0 1 (+13)
- -1 * M = 1 1 0 0 1 1
- -2 * M = 1 0 0 1 1 0

b) Carry Save Multiplier

- We have seen earlier how carry save adders (CSA) can be used to add several numbers with carry propagation only in the last stage.
- The partial products can be generated in parallel using n^2 AND gates.
- The n partial products can then be added using a CSA tree.
- Instead of letting the carries ripple through during addition, we *save* them and feed it to the next row, at the correct weight positions.

4 x 4 Carry Save Multiplier



- **Wallace Tree Multiplier**
  - A Wallace tree is a circuit that reduces the problem of summing n n-bit numbers to the problem of summing two $\Theta(n)$-bit numbers.
  - It uses $n/3$ (floor of) carry-save adders in parallel to convert the sum of n numbers to the sum of $2n/3$ (ceiling of) numbers.
  - It then recursively constructs a Wallace tree on the $2n/3$ (ceiling of) resulting numbers.
  - The set of numbers is progressively reduced until there are only two numbers left.
  - By performing many carry-save additions in parallel, Wallace trees allow two n-bit numbers to be multiplied in $\Theta(\log_2 n)$ time using a circuit of size $\Theta(n^2)$.

- The figure shows a Wallace tree that adds 8 partial products $m^{(0)}$, $m^{(1)}$, …, $m^{(7)}$.
- The partial product $m^{(i)}$ consists of (n+i) bits.
- Each line represents an entire number – the label of an edge indicates the number of bits.
- The carry-lookahead adder at the bottom adds a (2n-1)-bit number to a 2n-bit number to give the 2n-bit product.