ECE 341

Lecture # 6

Instructor: Zeshan Chishti
zeshan@pdx.edu

October 15, 2014

Portland State University

Lecture Topics

- Design of Fast Adders
  - Carry Looakahead Adders (CLA)
  - Blocked Carry-Lookahead Adders
- Multiplication of Unsigned Numbers
  - Array Multiplier
  - Sequential Circuit Multiplier

- Reference:
  - Chapter 9: Sections 9.2 and 9.3

CLA Delay Calculation

Consider the expression:

$$c_{i+1} = G_i + P_iG_{i-1} + P_iP_{i-1}G_{i-2} + \ldots\ldots\ldots + P_iP_{i-1}\ldots P_1G_0 + P_iP_{i-1}\ldots P_0c_0$$

- All the G_i and P_i functions can be obtained in parallel in **one** gate delay
- AND terms in each c_{i+1} calculation require **one** additional gate delay
- ORing the AND terms in each c_{i+1} calculation requires **one** additional gate delay

Therefore,

- Total delay in calculating **carry** outputs = 1 + 1 + 1 = **3** gate delays

Sum outputs require one additional XOR delay after carries are computed

- Total delay in calculating **sum** outputs = 3 + 1 = **4** gate delays

n-bit CLA requires 4 gate delays independent of *n*

CLA Fan-in Limitation

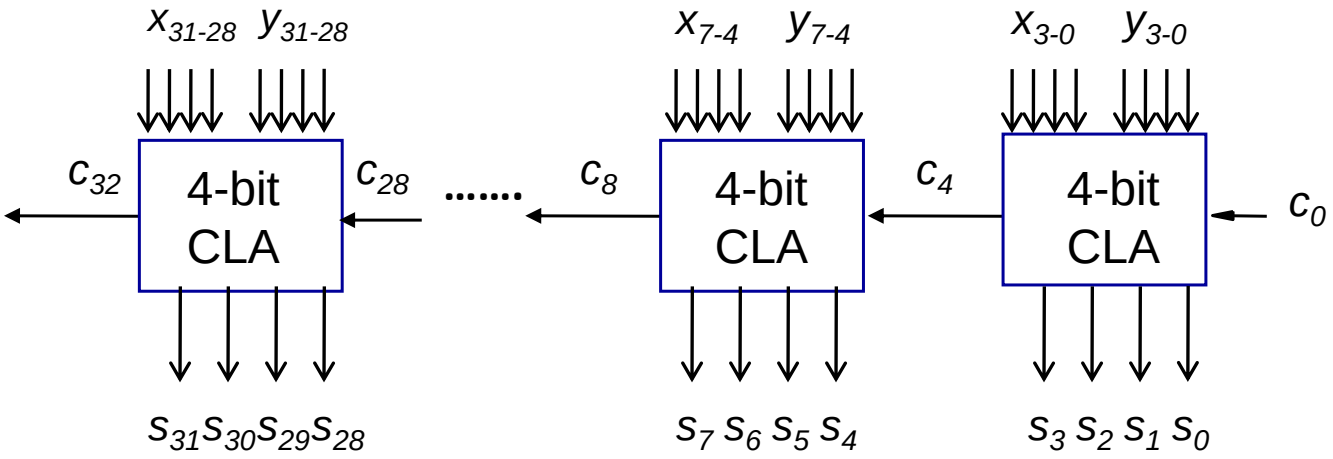
- Performing *n*-bit CLA in 4 gate delays, independent of *n*, good only in theory
- In practice, CLA is limited by fan-in constraints

$$c_{i+1} = G_i + P_iG_{i-1} + P_iP_{i-1}G_{i-2} + \ldots\ldots\ldots + P_iP_{i-1}\ldots P_1G_0 + P_iP_{i-1}\ldots P_0c_0$$

- OR gate & last AND gate in the expression for c_{i+1} require $i+2$ inputs, each
- For a 4-bit CLA, the MSB carry-out (c_4) requires a fan-in of 5
- 5 is the practical fan-in limit for most gates

- In order to add operands larger than 4-bits, we can cascade multiple CLAs
- Cascade of CLAs is called **Blocked Carry-Lookahead adder**

Blocked Carry-Looakhead Adder



32-bit Blocked CLA composed of eight 4-bit CLA blocks

After input operands (X, Y and c_0) are applied to the 32-bit adder:

- All the P_i and G_i terms in each CLA calculated in parallel in **1** gate delay
- c_4 available after **3** gate delays
- c_8 available 2 gate delays after c_4 = 3 + (1*2) = **5** gate delays
- c_{12} available (2*2) gate delays after c_4 = 3 + (2*2) = **7** gate delays
- c_{16} available after (3*2) gate delays after c_4 = 3 + (3*2) = **9** gate delays
- c_{32} available after (7*2) gate delays after c_4 = 3 + (7*2) = **17** gate delays
- s_{28}, s_{29}, s_{30}, s_{31} available after 17+1 = **18** gate delays

Carry-outs ripple from one CLA block to the next. Can we avoid this rippling?

Faster Blocked Carry-Lookahead adder

Key Idea: Generate the carry outputs c_4, c_8, c_{12}, … of CLA blocks **in parallel**, similar to how c_1, c_2, c_3, c_4 are generated in parallel *within* a CLA block

- Carry-out from a 4-bit block can be given as:

$$c_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0c_0$$

- This can be re-written as:

$$c_4 = G_0^1 + P_0^1c_0$$

where

$$G_0^1 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 \text{ and } P_0^1 = P_3P_2P_1P_0$$

We can similarly compute G_1^1, G_2^1, G_3^1 …..

G_i^1 & P_i^1 are **first-level** generate & propagate functions, where *i* denotes the CLA block

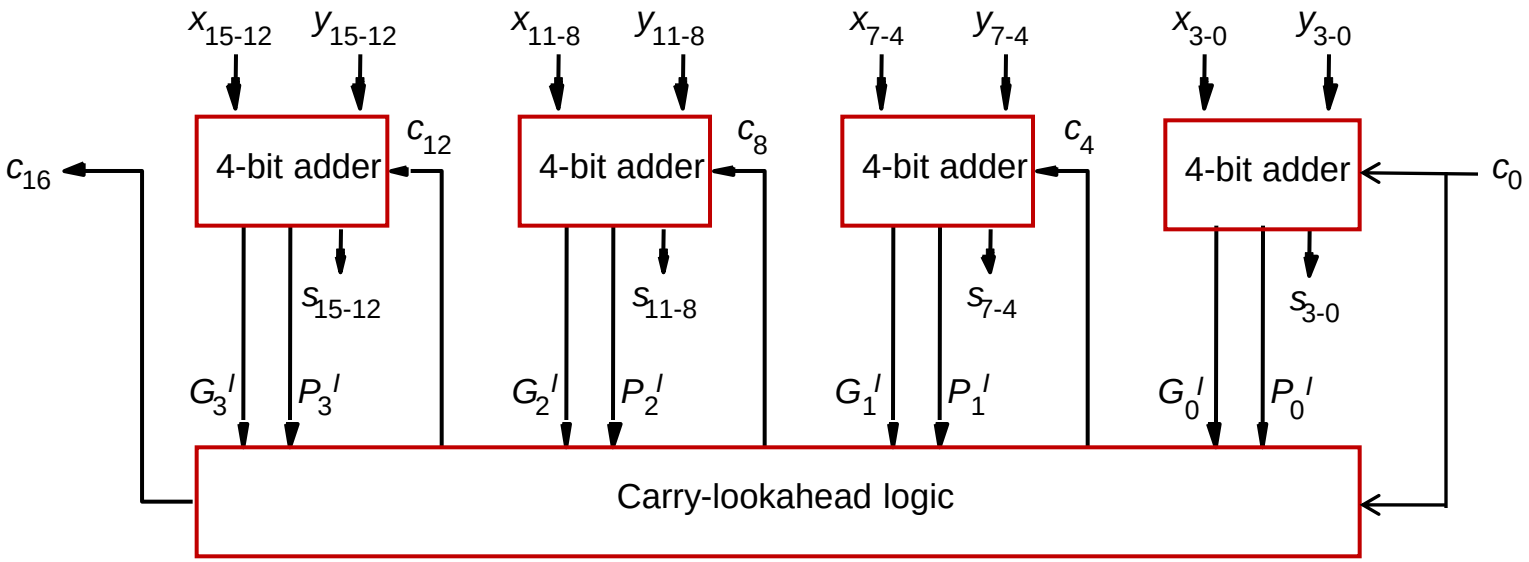
- $G_k^1 = 1$ implies that the *k*th CLA block generates a carry
- $P_k^1 = 1$ implies that the *k*th CLA block propagates a carry

Carry-out of *k*th block = $G_k^1 + P_k^1G_{k-1}^1 + P_k^1P_{k-1}^1G_{k-2}^1 + \ldots + P_k^1P_{k-1}^1\ldots P_1^1G_0^1 + P_k^1P_{k-1}^1\ldots P_0^1c_0$

- For example:

$$c_{16} = G_3^1 + P_3^1G_2^1 + P_3^1P_2^1G_1^1 + P_3^1P_2^1P_1^1G_0^1 + P_3^1P_2^1P_1^1P_0^1c_0$$

Blocked CLA with First-level Propagates and Generates

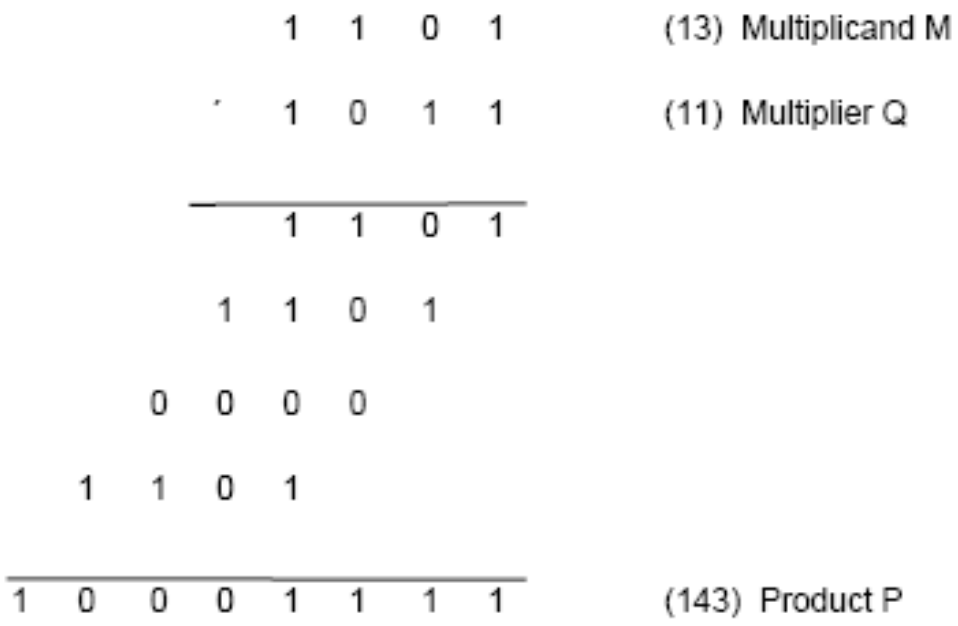


After input operands (X, Y and c_0) are applied to the above 16-bit adder:

- P_i and G_i terms *within* each CLA calculated in parallel in **1** gate delay
- First-level generates ($G_k{}^1$) available after 1 + 2 = **3** gate delays
- Carry-outs of CLA blocks (c_4, c_8, c_{12}, c_{16}) available after 3 + 2 = **5** gate delays
- Carries *within* CLA blocks (such as c_{15}) available after 5 + 2 = **7** gate delays
- Sum outputs (such as s_{15}) available after 7 + 1 = **8** gate delays
- Compare this with the blocked CLA formed by cascading, where c_{15} and s_{15} required 9 and 10 gate delays respectively

Multiplication

Multiplication of Unsigned Numbers



Product of two *n*-bit numbers is at most a *2n*-bit number

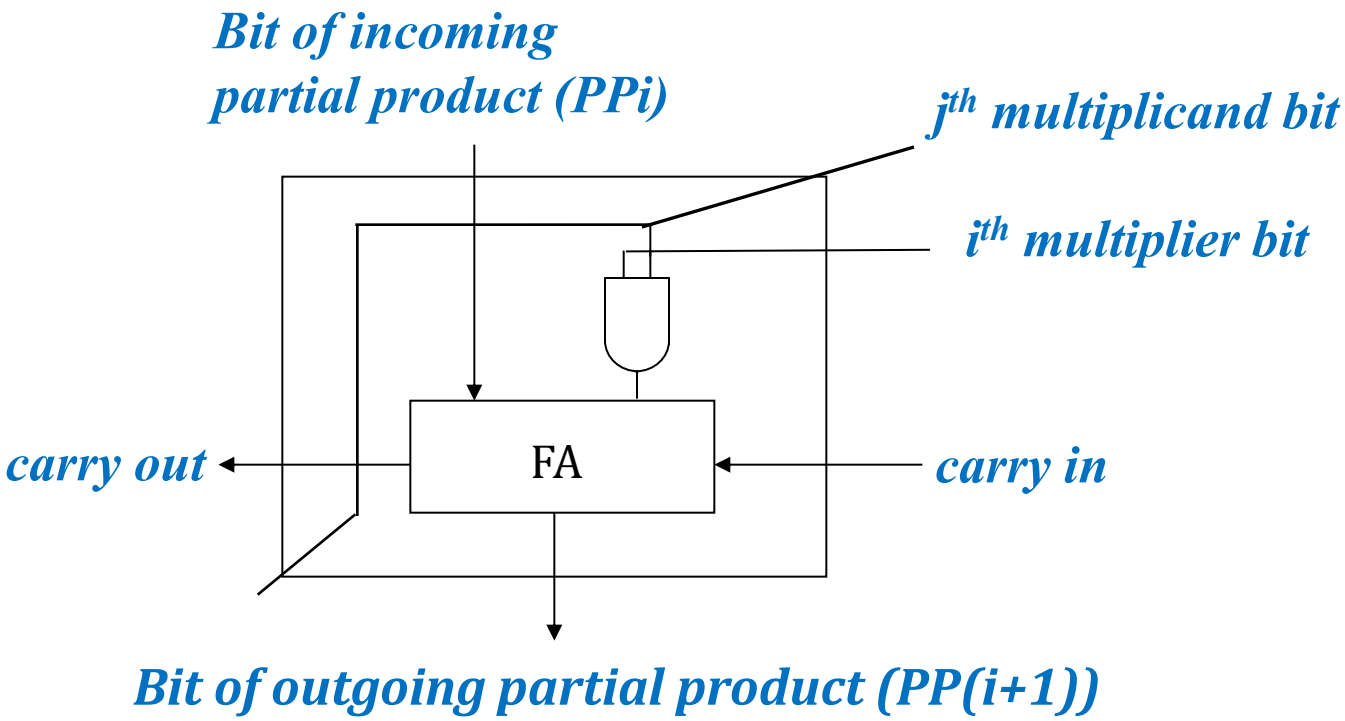
Unsigned multiplication can be viewed as addition of shifted versions of the multiplicand.

Multiplication of Unsigned Numbers (contd.)

- In hand multiplication, we add the shifted versions of multiplicand at the end (column-by-column)
  - Alternative would be to accumulate partial products at each stage (row-by-row)

- Multiplication logic for two *n*-bit numbers can be implement as follows :
  - Initialize the partial product ***PP0*** to a value of 0
  - Start from the LSB of multiplier and proceed towards MSB, one bit at a time. For each bit position of the multiplier, perform the following step:
    - If the i^{th} bit of the multiplier is 1, shift the multiplicand by *i* bit positions and add it to ***PPi*** in order to obtain ***PP(i+1)***
  - After *n* steps, the partial product ***PPn*** represents the final product

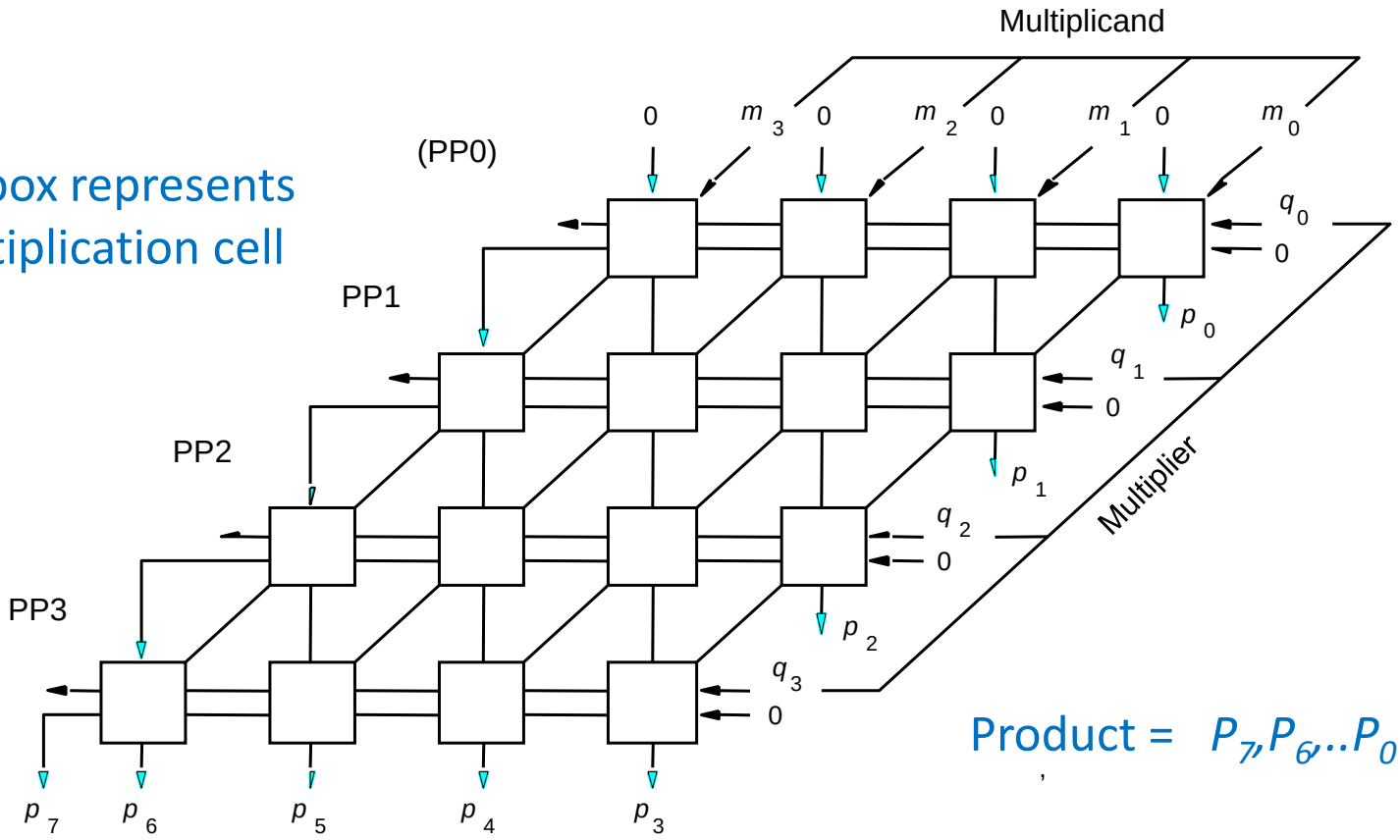
Multiplication of Unsigned Numbers



Typical multiplication cell

Combanitorial Array Multiplier

Each box represents a multiplication cell



Product = $P_7, P_6 .. P_0$

Multiplicand is shifted by displacing it through an array of adders

Combinatorial Array Multiplier (cont.)

- Array multipliers are highly inefficient:
  - Need n n-bit adders => number of gate counts is proportional to n^2
  - Impractical for large numbers such as 32-bit or 64-bit numbers typically used in computers
  - Perform only one function, namely, unsigned integer product

- Solution: Improve gate efficiency by using a mixture of combinatorial array techniques and sequential techniques
  - Instead of n n-bit adders, use one n-bit adder
  - Use a register to hold the accumulated partial product
  - This is called a **sequential multiplier**

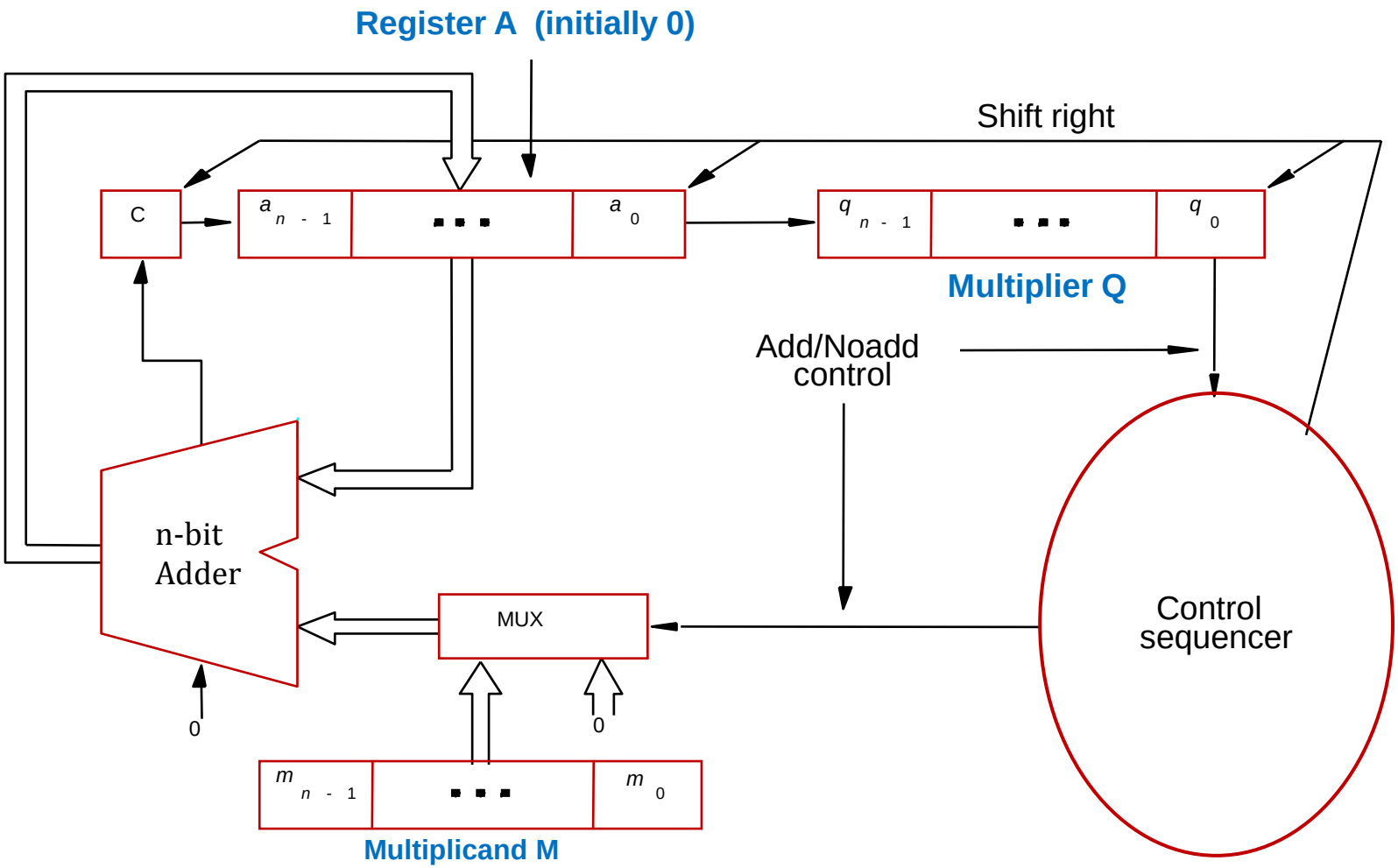
Sequential Multiplication

- Recall the rule for generating partial products:
  - If the i^{th} bit of the multiplier is 1, add the appropriately shifted multiplicand to the current partial product.
  - Multiplicand is shifted **left** when being added to the partial product

Key Observation:

- Adding a **left-shifted** multiplicand to an **unshifted** partial product is equivalent to adding an **unshifted** multiplicand to a **right-shifted** partial product

Sequential Circuit Multiplier

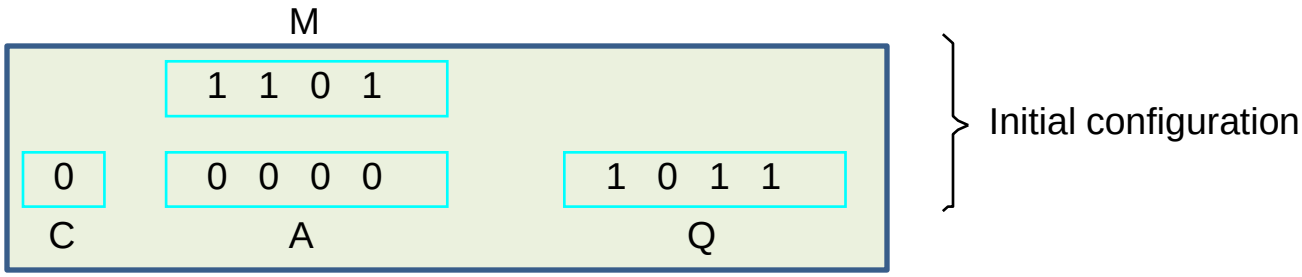


Sequential Multiplication Algorithm

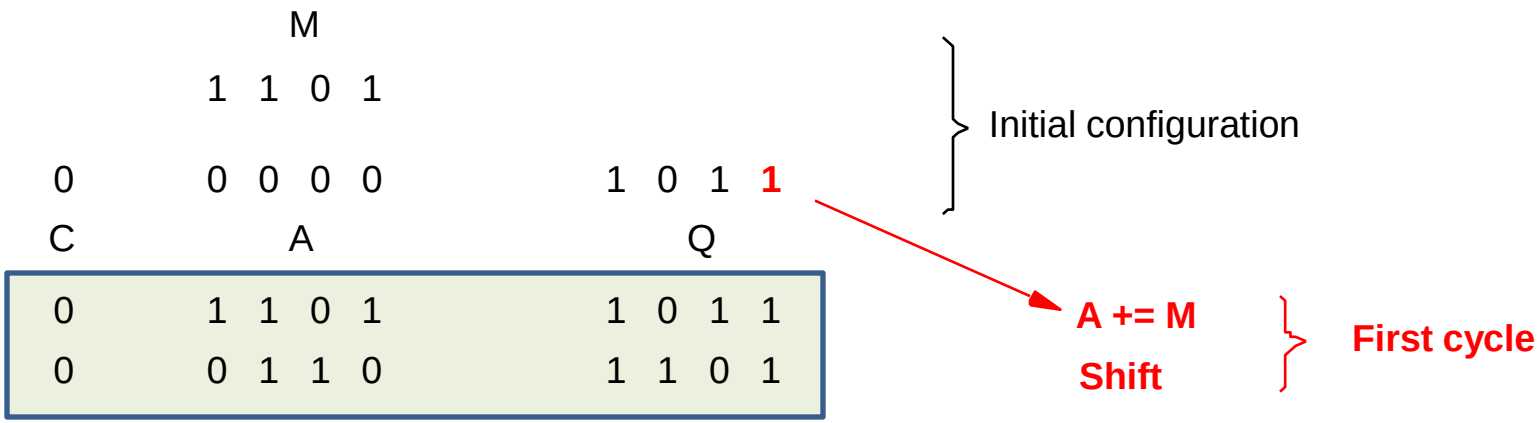
- Initialization:
  - Load multiplicand in “M” register, multiplier in “Q” register
  - Initialize “C” and “A” registers to all zeroes
- Repeat the following steps “n” times, where “n” is the number of bits in the multiplier
  - If (LSB of Q register == 1)

    A = A + M (carry-out goes to “C” register)
  - Treat the C, A and Q registers as one contiguous register and shift that register’s contents right by one bit position
- After the completion of “n” steps
  - Register “A” contains high-order half of product
  - Register “Q” contains low-order half of product

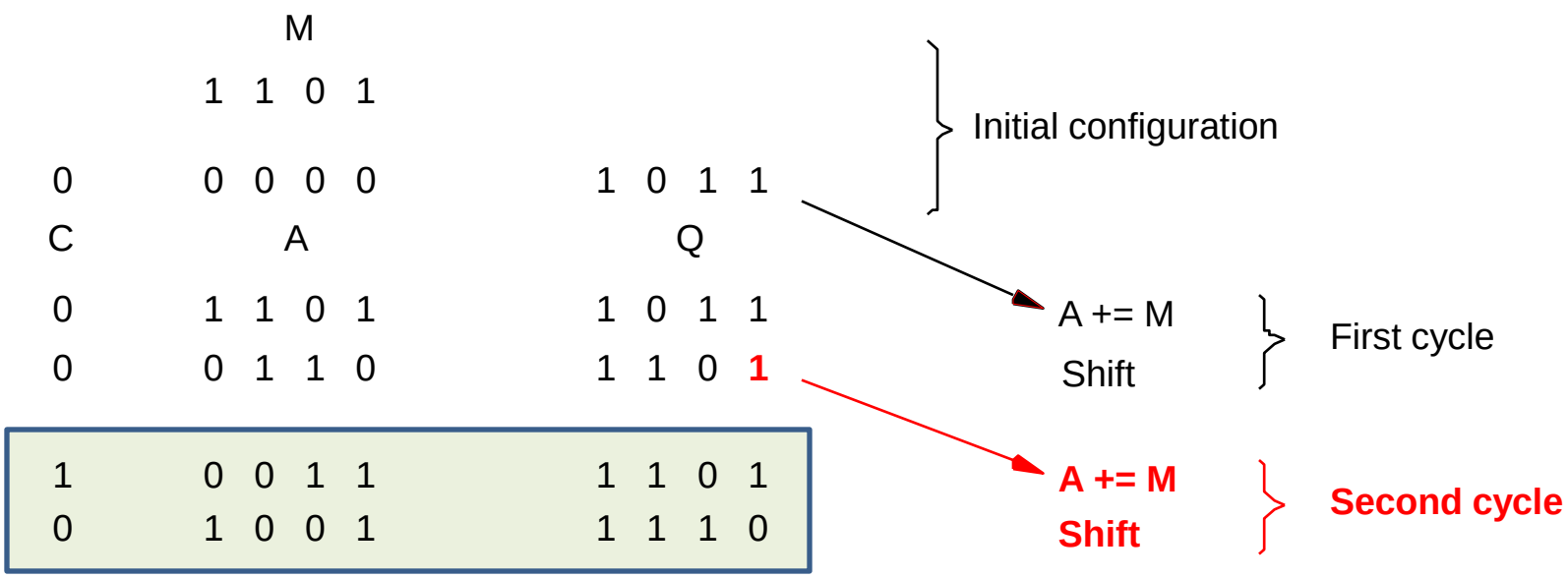
Sequential Multiplication Example



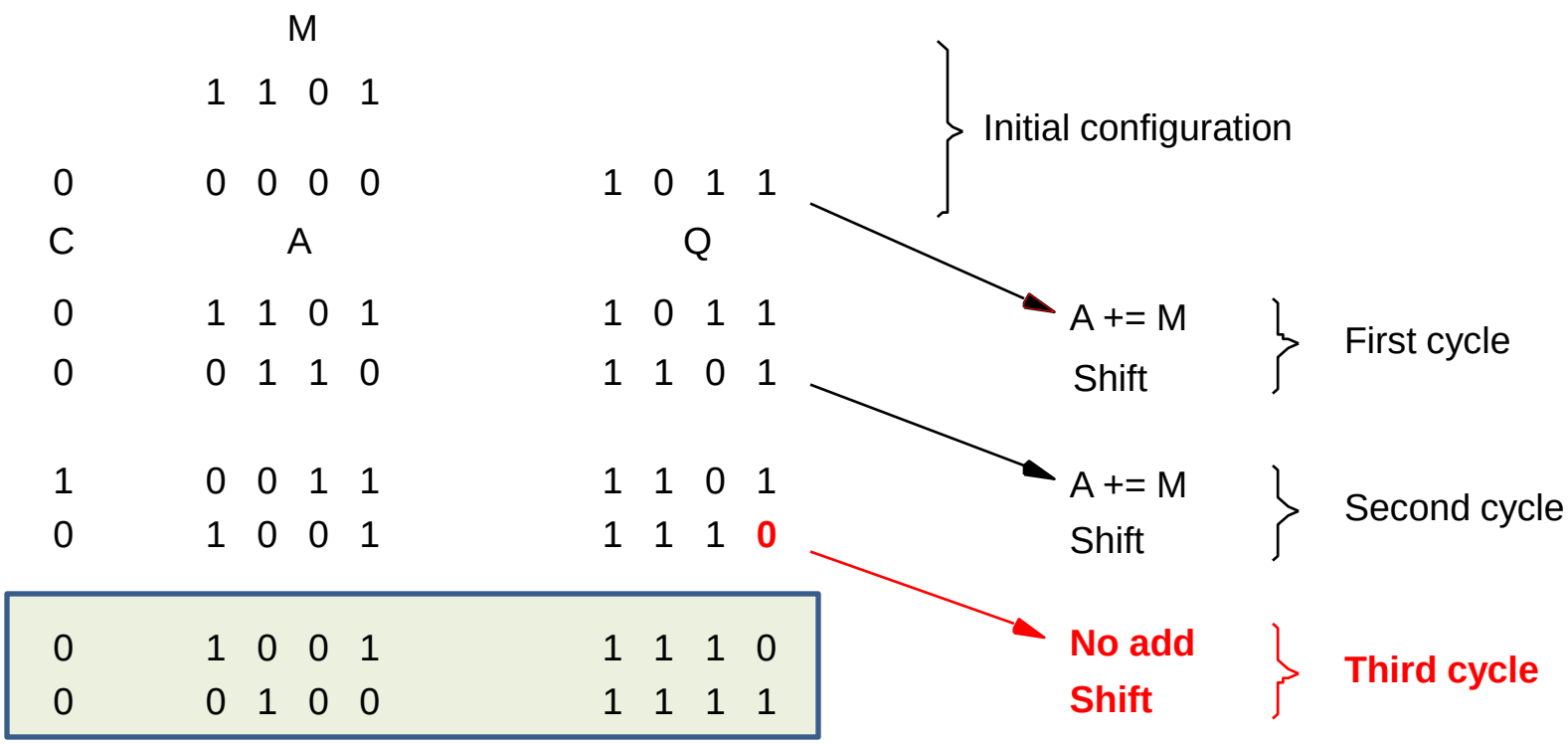
Sequential Multiplication Example



Sequential Multiplication Example



Sequential Multiplication Example



Sequential Multiplication Example