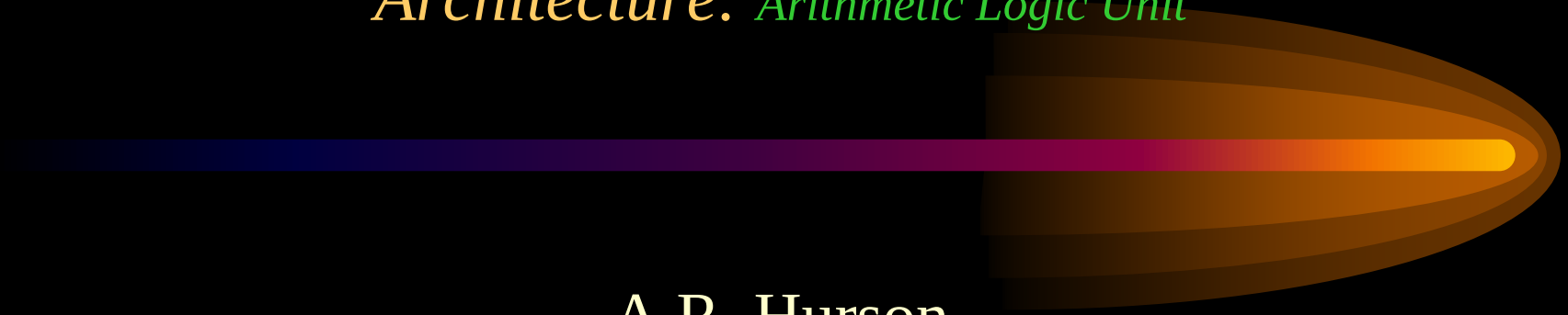
CS 5803
Introduction to High Performance Computer Architecture: Arithmetic Logic Unit



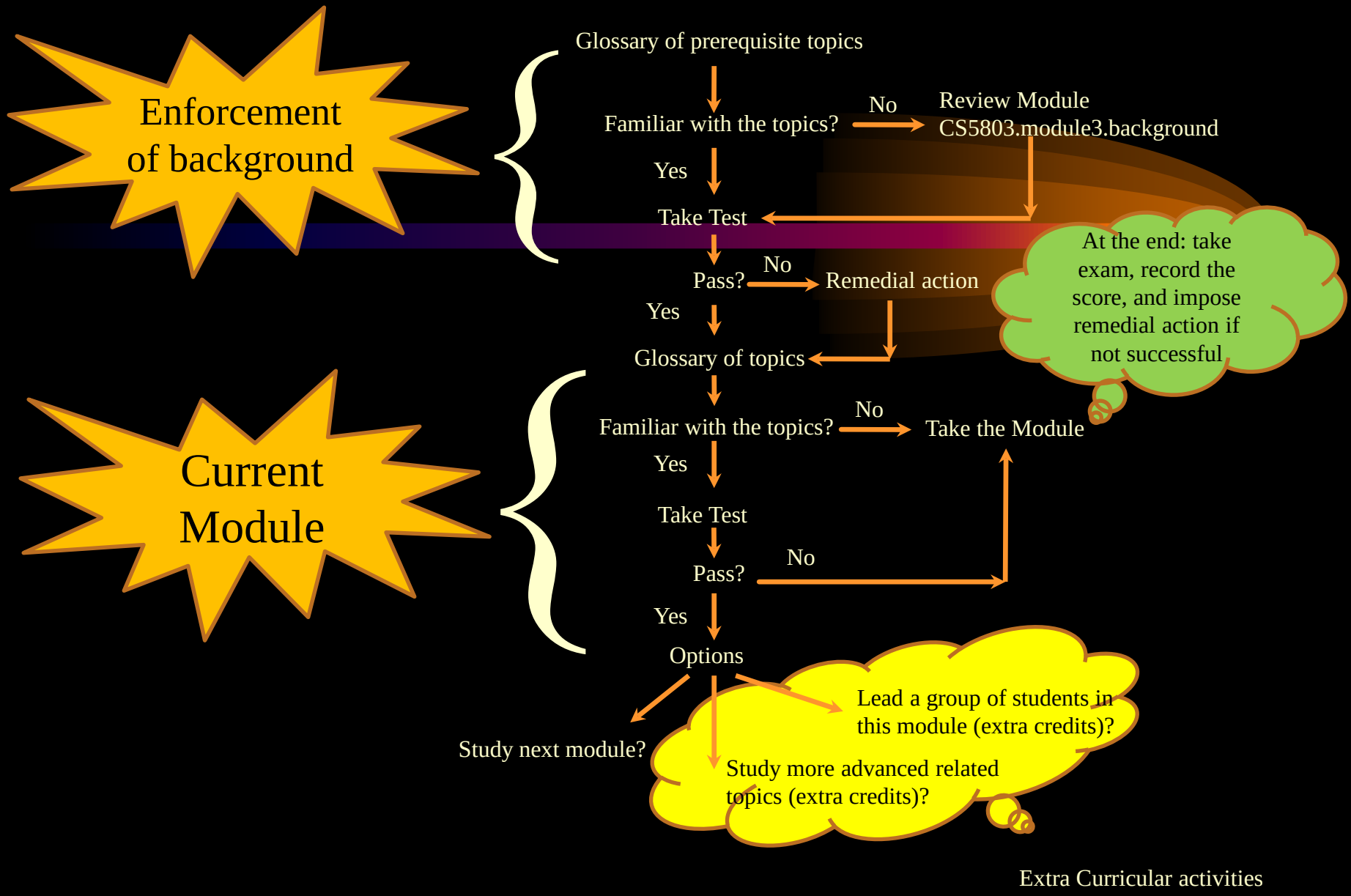
A.R. Hurson
323 CS Building,
Missouri S&T
hurson@mst.edu

- Outline
  - Motivation
  - Design of a simple ALU
  - How to design an ALU
  - Fast ALU design
    - Fast Adder
    - Fast Multiplier
    - Fast Divider

Note, this unit will be covered in almost three weeks. In case you finish it earlier, then you have the following options:

1) Take the early test and start CS5803.module4
2) Study the supplement module (supplement CS5803.module3)
3) Act as a helper to help other students in studying CS5803.module3

Note, options 2 and 3 have extra credits as noted in course outline.


Enforcement of background
Glossary of prerequisite topics
Familiar with the topics?
No
Review Module CS5803.module3.background
Yes
Take Test
Pass?
No
Remedial action
Yes
Glossary of topics
At the end: take exam, record the score, and impose remedial action if not successful
Current Module
Familiar with the topics?
No
Take the Module
Yes
Take Test
Pass?
No
Yes
Options
Study next module?
Lead a group of students in this module (extra credits)?
Study more advanced related topics (extra credits)?
Extra Curricular activities

- You are expected to be familiar with:
  - Representation of numbers,
  - Basic arithmetic operations in digital systems, including: addition, multiplication, and division,
  - Concept of serial, parallel, and modular ALU
- If not then you need to study CS5803.module3.background

- Arithmetic and Logic Unit (ALU)
  - In an attempt to improve the performance, this section will talk about the Arithmetic Logic Unit.
  - In regard to our earlier *CPU time*, we are looking at techniques to reduce *p*.

$$T = I_C * CPI * \tau = I_C * (p + m*k) * \tau$$

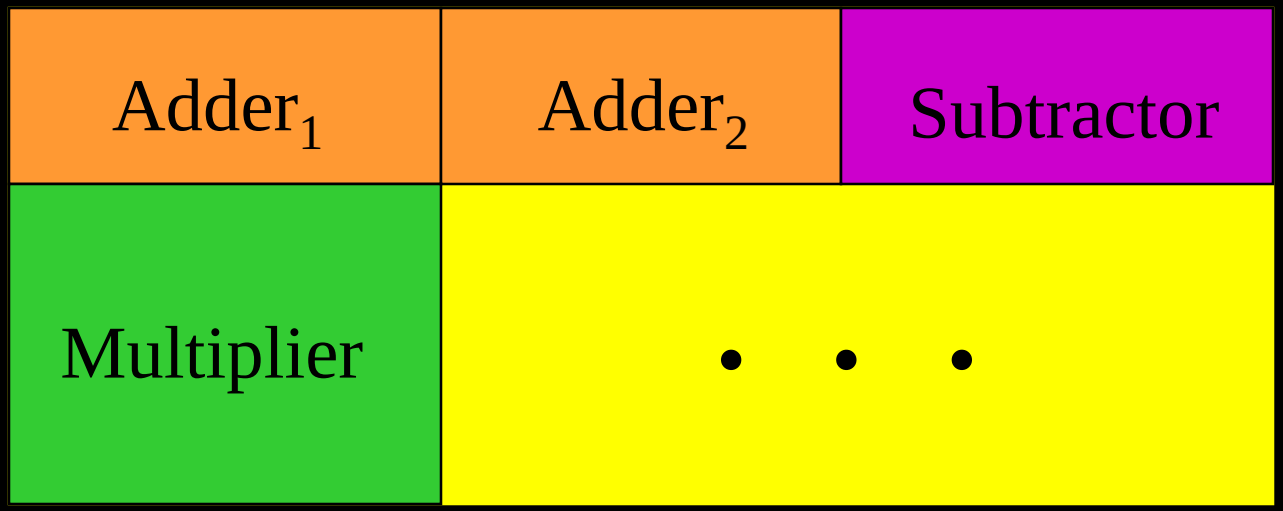
- Arithmetic and Logic Unit (ALU)
  - It is a functional box designed to perform basic arithmetic, logic, and shift operations on the data.
  - Implementation of the basic operations such as logic, program control, and data transfer operations are easier than arithmetic and I/O operations. Therefore, in this section we concentrate on arithmetic operations.

- Arithmetic and Logic Unit (ALU)
  - An ALU can be of three types:
    - Serial
    - Parallel (see module3.background for definitions and more discussion about serial and parallel ALU)
    - Functional (Modular)

- Arithmetic and Logic Unit (ALU)
  - As discussed before a parallel ALU offers a higher speed relative to a serial ALU.
  - How can one improve the performance (speed) of ALU further?
  - Is it possible to build (design) an ALU faster than a parallel ALU?

- Arithmetic and Logic Unit (ALU)
  - Functional (modular) ALU
    - ALU is a collection of independent units each tailored for a specific operation. As a result, independent operations can be overlapped.
    - This approach allows an additional degree of concurrency relative to a parallel ALU, since it allows several operations to be performed on data simultaneously.
    - This speed improvement comes at the expense of extra overhead needed to detect data independent operations.

- Arithmetic and Logic Unit (ALU)
  - Functional (modular) ALU



- Arithmetic and Logic Unit (ALU)
  - Is it possible to improve the performance of an ALU further?
  - Naturally, we can improve the performance (physical speed) by taking advantage of the advances in technology.
  - How can we improve the logical speed of the ALU further?

- Arithmetic and Logic Unit (ALU)
  - In a functional ALU, is it possible to devise algorithms which allow one to improve the performance of the basic operations?
  - If this is a valid direction, then the question of how to design a fast ALU will change to "how to design a fast adder, a fast multiplier, ...?"

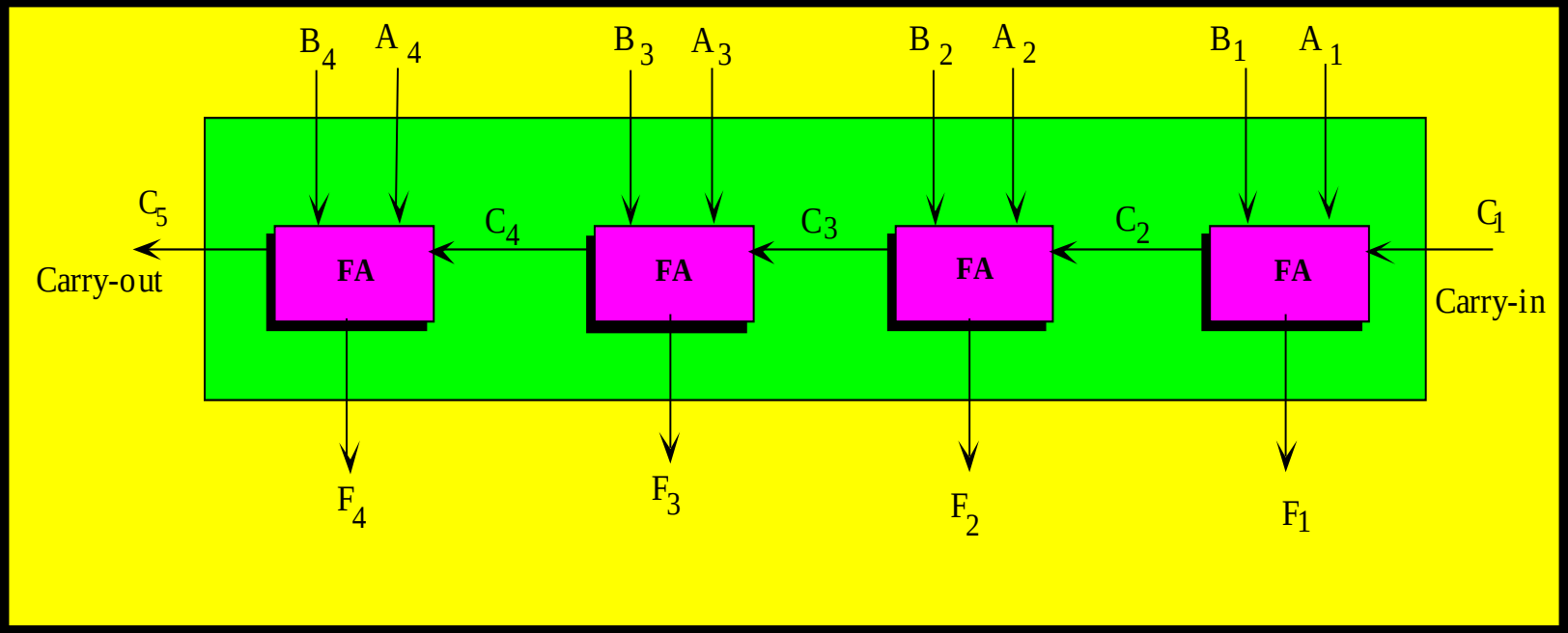
- Question
  - As a computer architect, how do you design an ALU? In another words, in an attempt to design an ALU, what issues do you need to take into consideration?

- Fast Adder
  - How to design an adder faster than a parallel adder?
  - What is the major bottle-neck in a parallel adder?
  - Is the carry generation and propagation the major bottleneck?
  - Is it possible to eliminate, moderate, or reduce the delay of carry generation and propagation?

- Arithmetic and Logic Unit (ALU)
  - Carry Lookahead
    - Scheme 1
    - Scheme 2
  - Carry Select
  - Carry Lookahead plus Carry Select

- Fast Adder
  - Carry Lookahead — Generate and propagate carries ahead of time, relative to a parallel adder.

- Fast Adder
  - Basic Building Block — A 4-Bit Adder



- Fast Adder
  - Basic Building Block — A 4-Bit Adder (Timing)

$F_1 = 4\Delta t$ $C_2 = 4\Delta t$

$F_2 = 6\Delta t$ $C_3 = 6\Delta t$

$F_3 = 8\Delta t$ $C_4 = 8\Delta t$

$F_4 = 10\Delta t$ $C_5 = 10\Delta t$

- Fast Adder
  - Carry Lookahead (Scheme 1)

$$C_{i+1}=A_iB_i+(A_i \oplus B_i)C_i=A_iB_i+(A_i+B_i)C_i$$

Carry Generate term (G_i)

Carry Propagate Term (P_i)

- Fast Adder
  - Carry Lookahead (Scheme 1)
    - In a 4-bit full adder

$C_1 = 0$

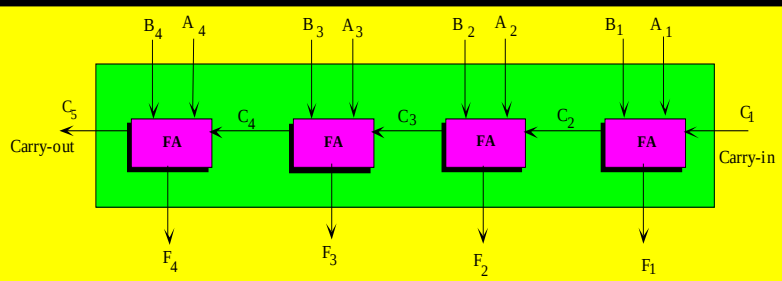
$C_2 = g_1 + P_1 C_1$

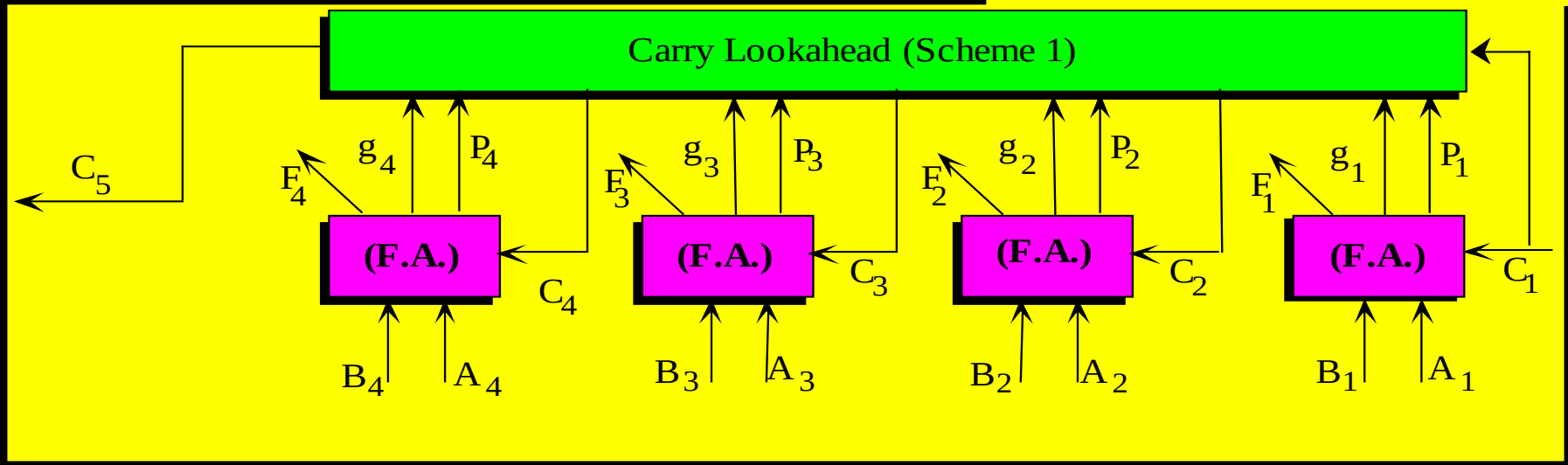
$C_3 = g_2 + P_2 C_2 = g_2 + P_2 g_1 + P_2 P_1 C_1$

$C_4 = g_3 + P_3 C_3 = g_3 + P_3 g_2 + P_3 P_2 g_1 + P_3 P_2 P_1 C_1$

$C_5 = \ ...$

- Fast Adder — Carry Lookahead (Scheme 1)
  - Extended 4-Bit Full Adder





- Fast Adder — Carry Lookahead (Scheme 1)
  - Extended 4-Bit Full Adder — Timing

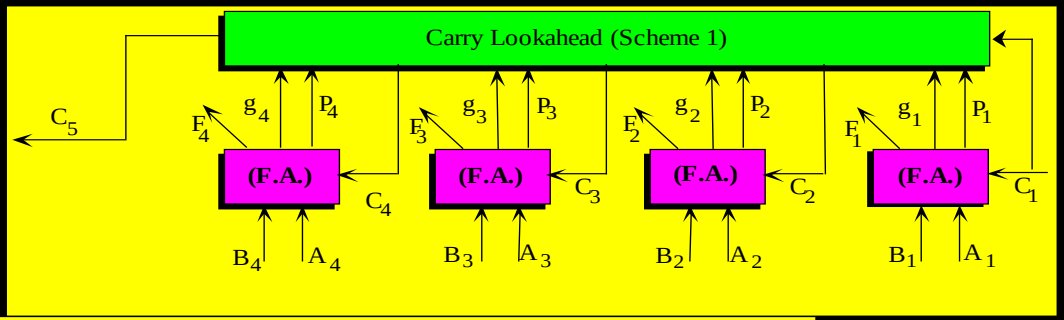
    $d \cong 2\Delta t$

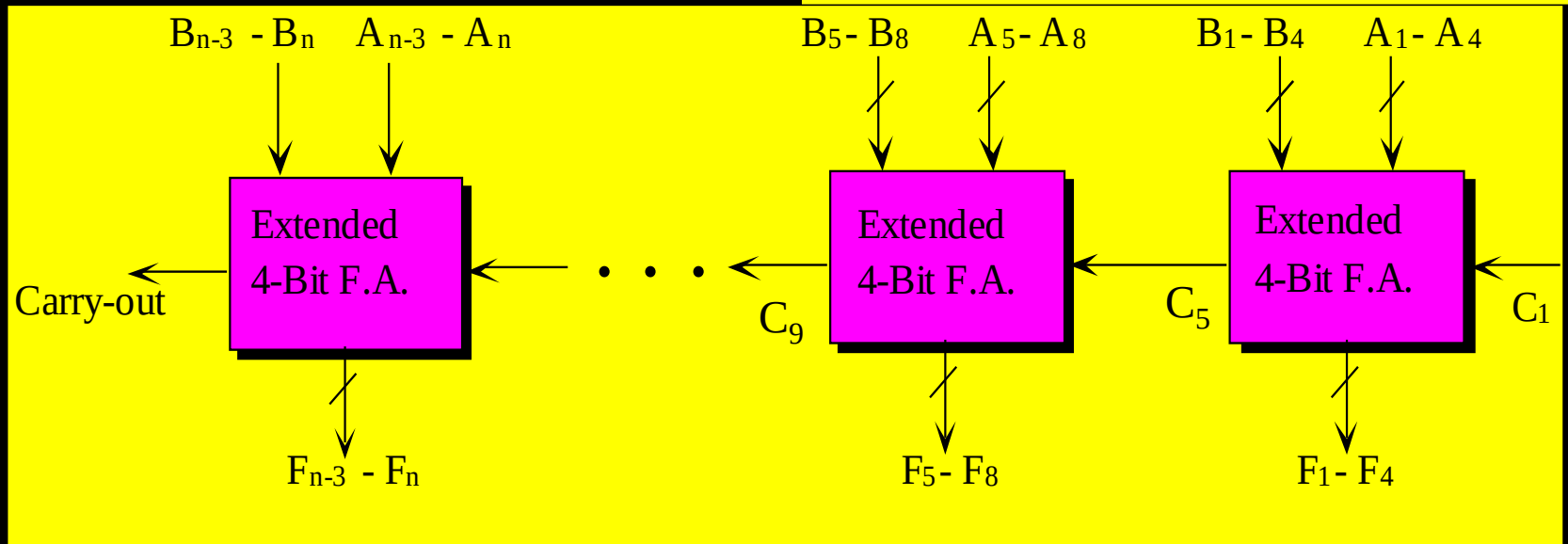
    p^s and g^s are generated in d

    C^s are generated after another d

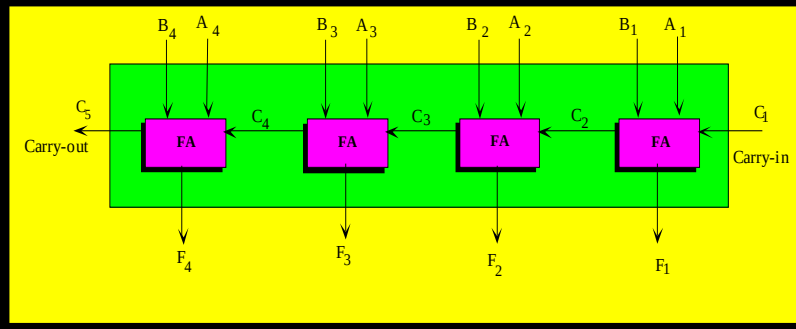
    F^s are generated after another d

Fast Adder — Carry Lookahead (Scheme 1)





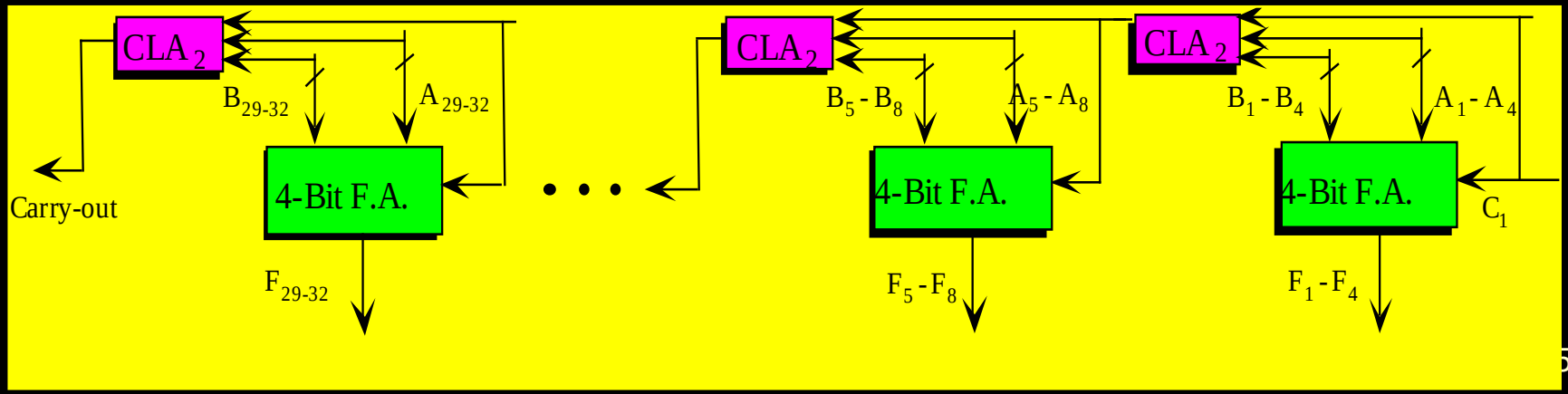
- Fast Adder — Carry Lookahead (Scheme 2)



Timing

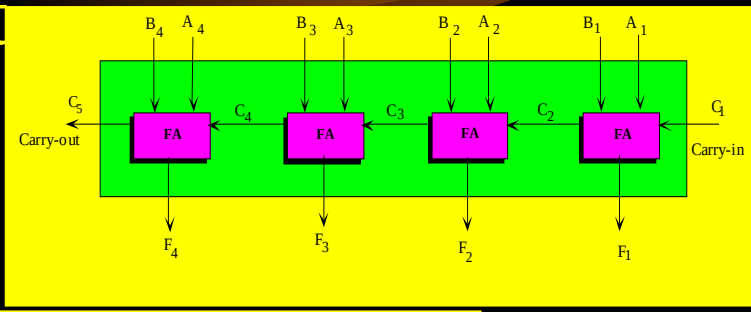
CLA = 5Δt

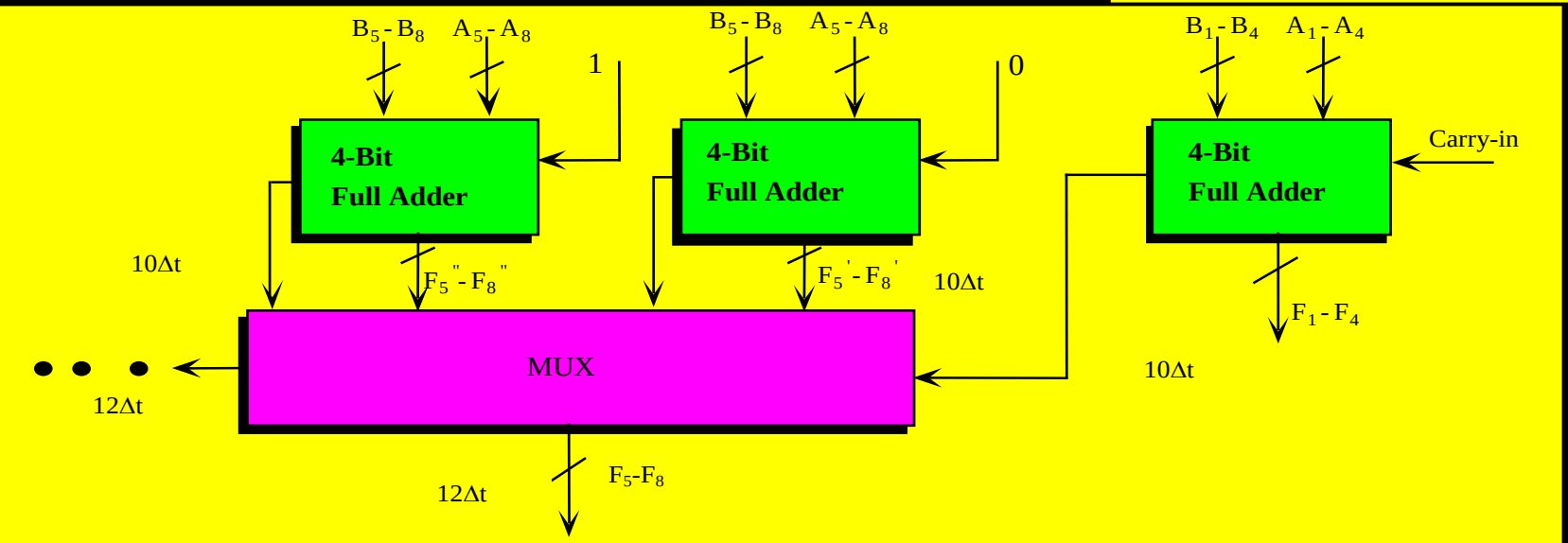
Cascades of CLAs overlap 1Δt operation

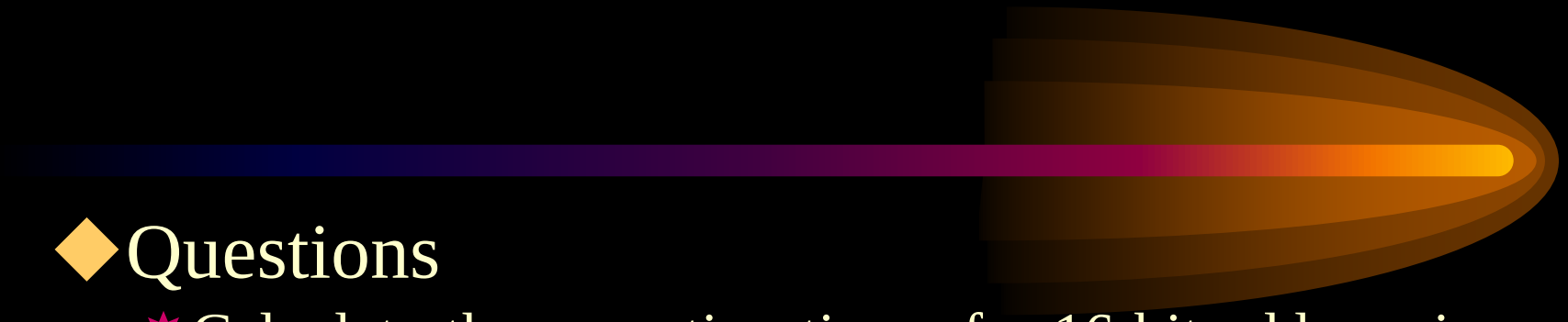


- Fast Adder
  - Carry Select
    - Carry-in to a 4-bit full adder is either 0 or 1.
    - Duplicate each stage - e.g., 4-bit full adder.
    - Initiate each unit in a stage with carry-in of 0 and 1.
    - Use a multiplexer to select the correct answer.

- Fast Adder — Carry Select





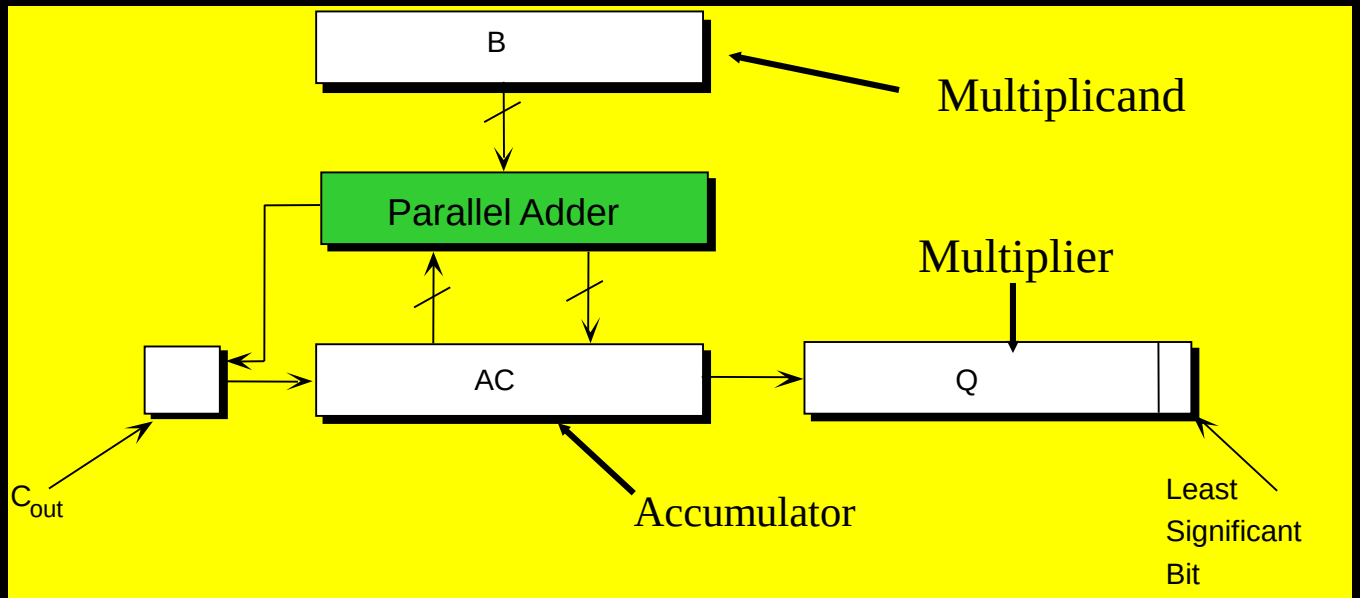


- Questions
  - Calculate the execution time of a 16-bit adder using carry lookahead scheme 1.
  - Formulate the execution time of an n-bit adder using carry lookahead scheme 1 (n is a multiple of 4).
  - Calculate the execution time of a 16-bit adder using carry lookahead Scheme 2.
  - Formulate the execution time of an n-bit adder using carry lookahead scheme 2 (n is a multiple of 4).

- Questions
  - Calculate the execution time of a 16-bit adder using carry select scheme.
  - Formulate the execution time of an n-bit adder using carry select scheme.
  - Is it possible to combine carry lookahead and carry select concepts to design a faster adder?

- Multiplication
  - Multiplication can be performed as a sequence of repeated additions.
  - *A* * *B* is interpreted as add *A*, *B* times. However, such a scheme is very inefficient with a time complexity of O(m) where *m* is the magnitude of *B*.
  - A better approach to multiplication, add-and-shift, produces a time complexity of O(n) where *n* is the length of the *B*.

- Add-and-shift — hardware configuration
  - Multiplier and multiplicand are two n-bit unsigned numbers,
  - Result is a 2n-bit number stored in an accumulator and multiplier registers.



- Add-and-shift — Algorithm
  - In each iteration the least-significant bit of multiplier is checked;
    - if one, then multiplicand is added to the accumulator and the contents of accumulator and multiplier is shifted right one position.
    - if zero, just shift accumulator and multiplier to the right.
    - See module3.background for additional discussion about Add-and-shift algorithm.

- Multiplication — Booth's Algorithm
  - Booth's algorithm is an extension to the add-and-shift approach.
  - In each iteration two bits of multiplier are being investigated and proper action(s) will be taken according to the following coding table:

00	no action	shift right once
01	add multiplicand	shift right once
10	sub multiplicand	shift right once
11	no action	shift right once

See module3.background for more discussion about Booth's algorithm.

- Multiplication — Modified Booth's Algorithm
  - Check 3 bits of multiplier at a time and take proper steps as follows:

000	no action	shift right twice
001	add multiplicand	shift right twice
010	add multiplicand	shift right twice
011	add 2*multiplicand	shift right twice
100	sub 2*multiplicand	shift right twice
101	sub multiplicand	shift right twice
110	sub multiplicand	shift right twice
111	no action	shift right twice

- Multiplication — Booth's Algorithm
  - Any version of Booth's algorithm allows a sequence of consecutive 1^s to be bypassed.
  - Modified Booth's Algorithm is faster than Booth's Algorithm.
  - Booth's Algorithm can be further extended by looking at 4 bits, (5 bits, ...) at a time and taking proper actions according to the proper encoding table.

Multiplication — Modified Booth's Algorithm

001011 * 011001 Extension

000000 0110010
001011 010 ⇒ add B, shift twice

001011 0110010
000010 1101100
101010 100 ⇒ sub 2B, shift twice

101100 1101100
111011 0011011
010110 011 ⇒ add 2B, shift twice

010001 0011011

000100 010011 0

Answer

- Summary
  - How to design an ALU?
  - Fast adder
    - Carry Lookahead
    - Carry Select
  - Multiplication
    - Add-and-shift
    - Booth's algorithm and its extension

- Multiplication
  - The add-and-shift algorithm can be used to multiply numbers (say *A* and *B*) in 2^s complement, if the result is adjusted properly. Three cases can be recognized.
    - Case 1: A positive; B negative
    - Case 2: A negative; B positive
    - Case 3: A negative; B negative

- Multiplication — Case 1: A positive B negative
  - Proof

$$A * \widetilde{B} = A * (2^n - B) = 2^n A - A * B$$

$$A * B = 2^n A - A * \widetilde{B}$$

$$2^{2n} - A * B = 2^{2n} - 2^n A + A * \widetilde{B}$$

$$\widetilde{AB} = 2^n(2^n - A) + A * \widetilde{B} = 2^n \widetilde{A} + A * \widetilde{B}$$

Multiply A and B using add-and-shift algorithm and adjust the result by $2^n . \widetilde{A}$

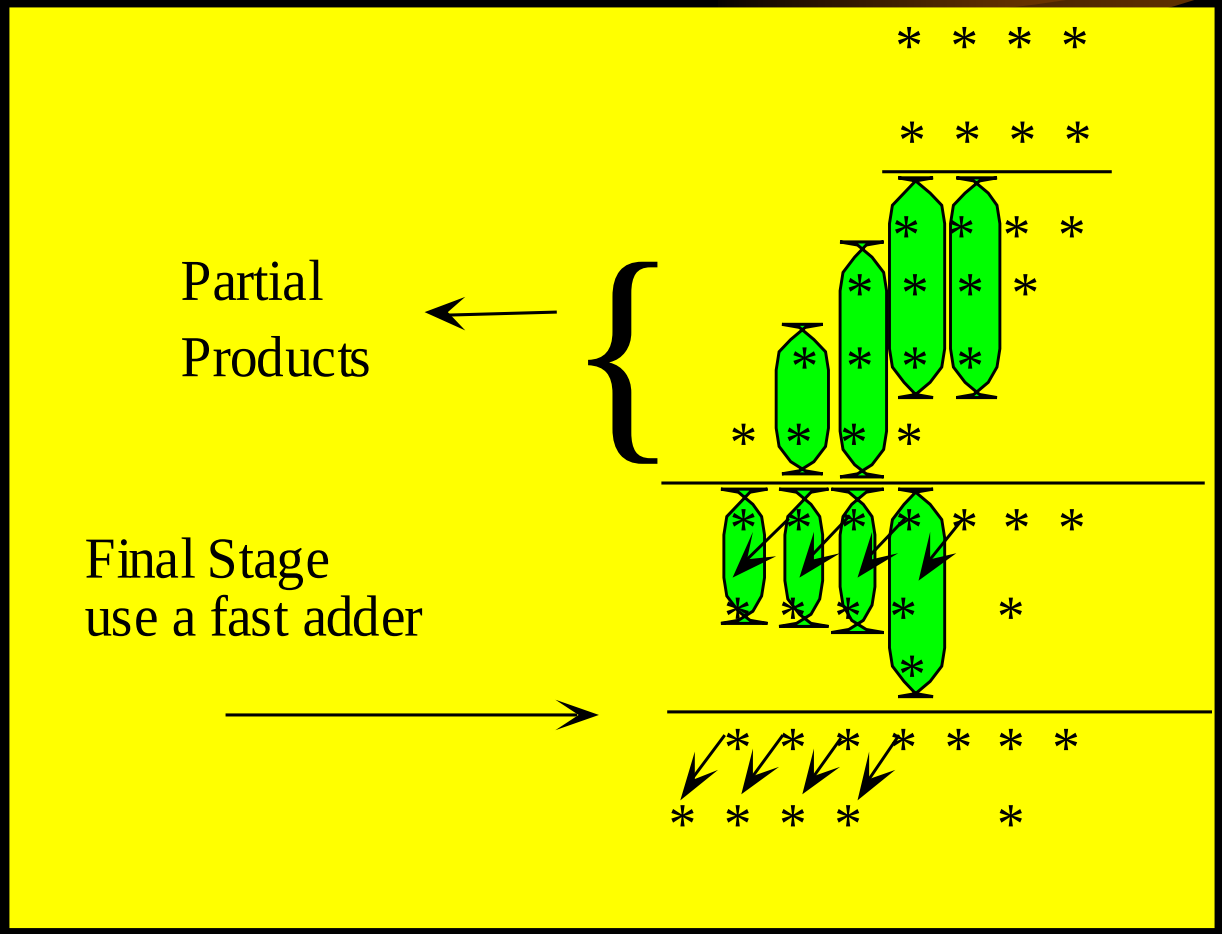
- Questions
  - Justify case 2 and case 3.
  - Is it possible to use the same technique for 1^{s} complement numbers?

- Multiplication — Example
  - Perform 00101 * 11010 using add-and-shift algorithm, numbers are in 2^s complement format:

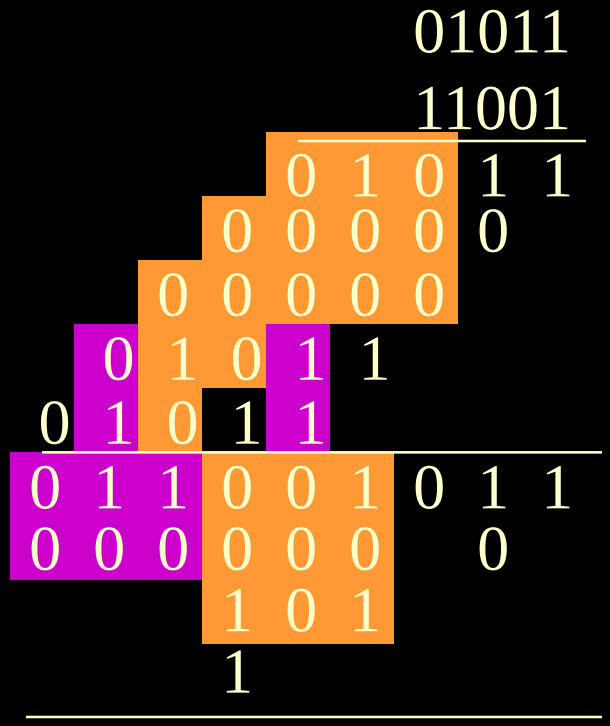
E	AC	A	
0	0 0 0 0 0	1 1 0 1 0	A_n = 0, shift right EACA
0	0 0 0 0 0	0 1 1 0 1	A_n = 1, add B
	0 0 1 0 1		
0	0 0 1 0 1	0 1 1 0 1	Shift right EACA
0	0 0 0 1 0	1 0 1 1 0	A_n = 0, shift right EACA
0	0 0 0 0 1	0 1 0 1 1	A_n = 1, add B
	0 0 1 0 1		
0	0 0 1 1 0	0 1 0 1 1	Shift right EACA
0	0 0 0 1 1	0 0 1 0 1	A_n = 1, add B
	0 0 1 0 1		
0	0 1 0 0 0	0 0 1 0 1	Shift right EACA
0	0 0 1 0 0	0 0 0 1 0	Adjust the result
	1 1 0 1 1		
	1 1 1 1 1	0 0 0 1 0	← Answer

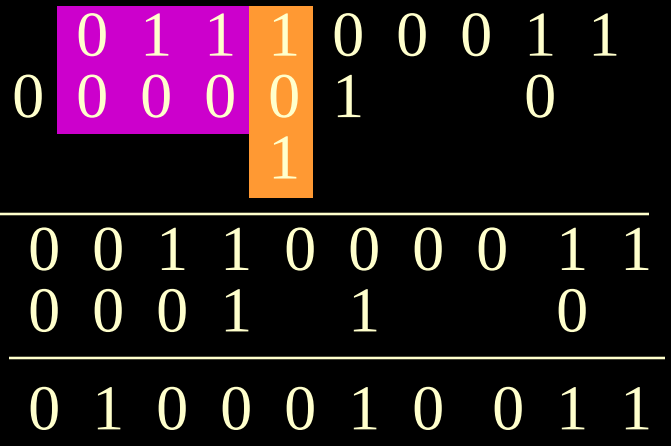
- Fast Multiplication
  - Reduction of Summands
    - Generate matrix of summands (partial products).
    - Go over several reduction stages using 2-2 and 3-2 adders.
    - In final stage (2 rows) use a fast adder to generate the result.

- Fast Multiplication — Reduction of Summands



- Fast Multiplication — Reduction of Summands





- Summary
  - Multiplication
    - Add-and-shift
    - Booth's algorithm and its extension
  - Reduction of Summands

- Fast Multiplication — Reduction of Summands
  - It is suitable for unsigned numbers.
  - Number of reduction stages depends on the length of the multiplier.
  - Execution time:

$$T_{RS} = \Delta t + m * (4 \quad \Delta t) + P$$

- Δt: Time to generate matrix of summands
- m: No. of reduction stages
- P: Execution time of the fast adder

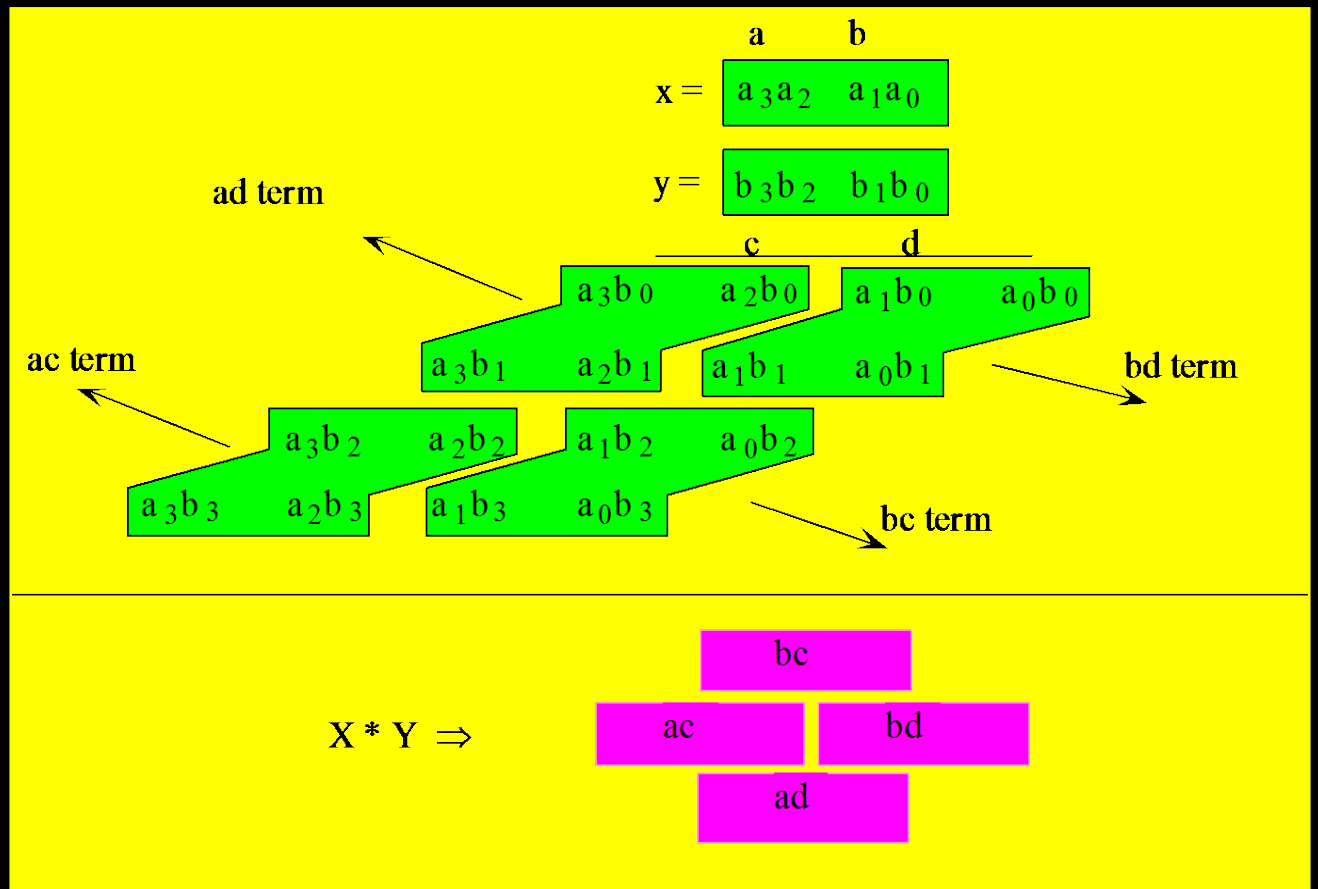
- Fast Multiplication
  - Iterative Method
    - Multiplication of 2 n-bit numbers can be converted into four multiplications of n/2-bit numbers and two additions.
    - This scheme can be iteratively applied to all multiplication terms

- Fast Multiplication — Iterative Method

$$X = 2^{n/2} a + b,\ Y = 2^{n/2} c + d$$

$$X*Y=(2^{n/2} a+b)*(2^{n/2} c+d)= 2^{n}(ac)+ 2^{n/2}(ad+bc)+bd$$

Fast Multiplication — Iterative Method



- Fast Multiplication
  - Iterative Method — An Example

$x = a_3a_2a_1a_0$

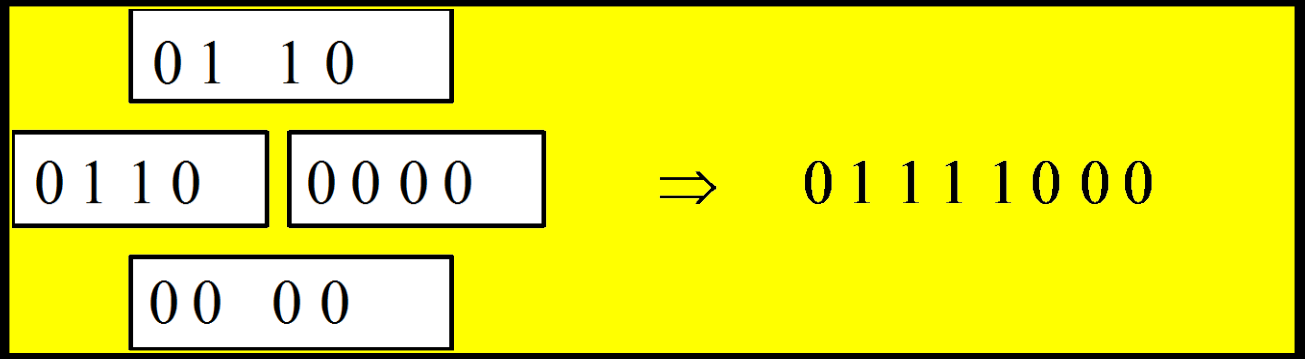
$y = b_3b_2b_1b_0$

$$x*y = 2^4(a_3a_2)(b_3b_2) + 2^2[(a_3a_2)(b_1b_0) + (a_1a_0)(b_3b_2)] + (a_1a_0)(b_1b_0)$$

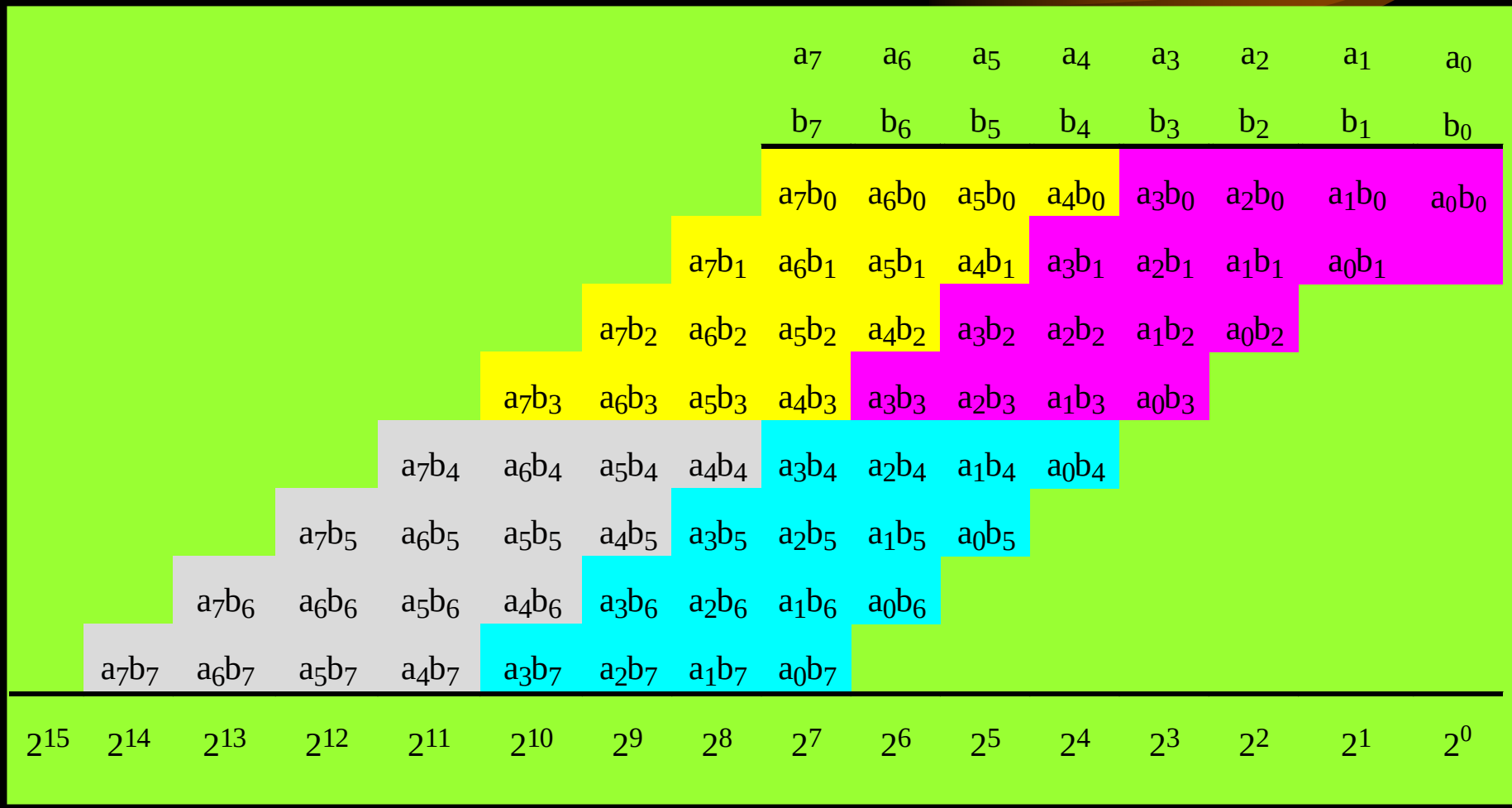
- Fast Multiplication
  - Iterative Method — An Example

    x = 1010

    y = 1100



Introduction to High Performance Computer Architecture



- Fast Multiplication
  - Hurson's Scheme — Observations
    - In a parallel multiplier unit first an n*n matrix of partial products (M) is generated and then elements in each column are added.

$$m_{ij} \varepsilon M \begin{cases} 1 & if\ B_i=Q_j = 1 \\ 0 & otherwise \end{cases} \qquad 1 \le i, j \le n$$

- Fast Multiplication
  - Hurson's Scheme — Observations
    - An element m_{ij} in M is the result of an AND operation between the i^{th} bit of multiplicand and j^{th} bit of multiplier.
    - In each column, zero elements do not affect the summation in that column and carry to the next column.
    - Each pair of 1^s in a column contributes a carry to the next column.
    - The result of summation for each column is either zero (even number of 1^s) or one (odd number of 1^s).

- Fast Multiplication — Hurson's Scheme
  - Generate only non-zero elements in each column.
  - For each pair of 1^s in a column generate a carry to the next column.
  - Count the number of 1^s in each column.

Fast Multiplication

Hurson's Scheme — Example

```
    B = 11010
  *Q = 11011
 ____________
  111111110
    1  11
         1
 ____________
1 111111110
  1111 11
    1 11
 ____________
1010111110
```

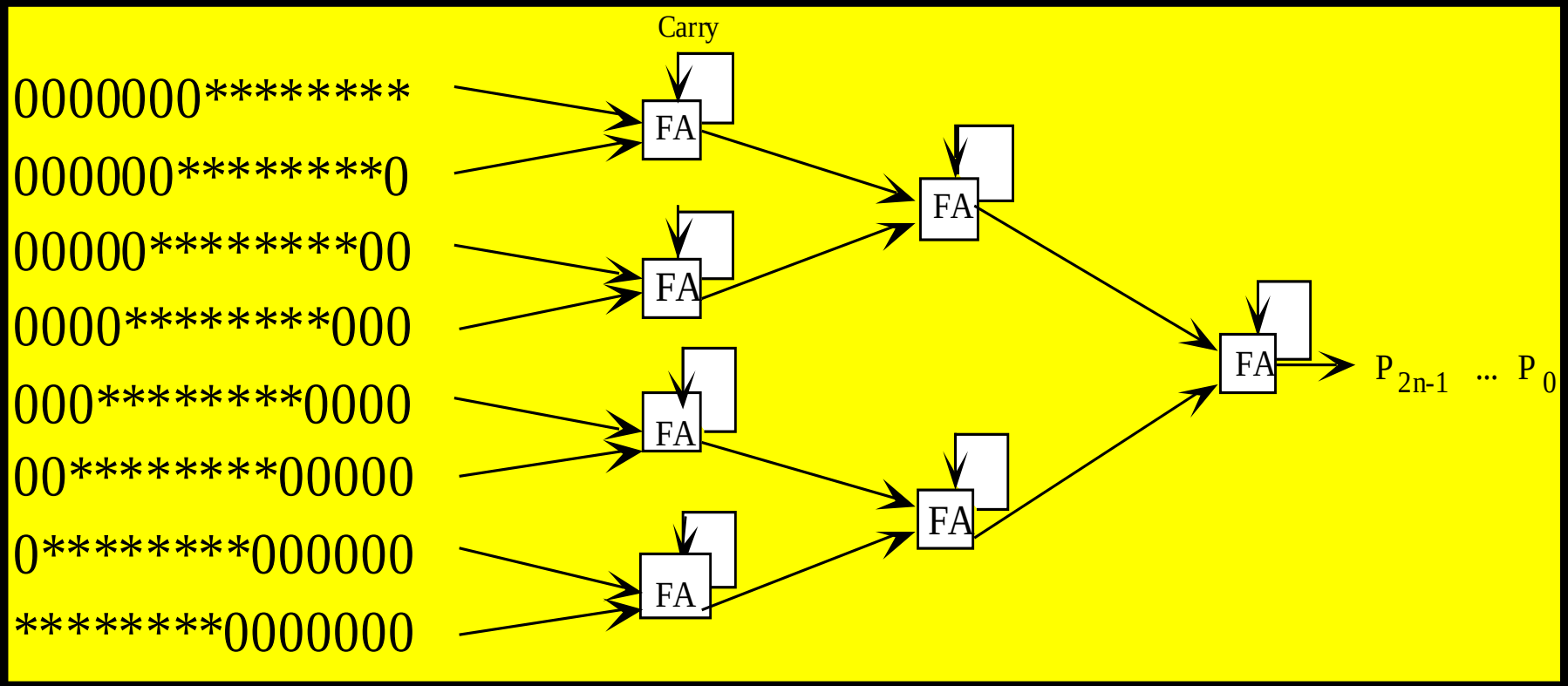
Matrix of partial products without zero elements

Generate carries

Count the number of 1s in each column

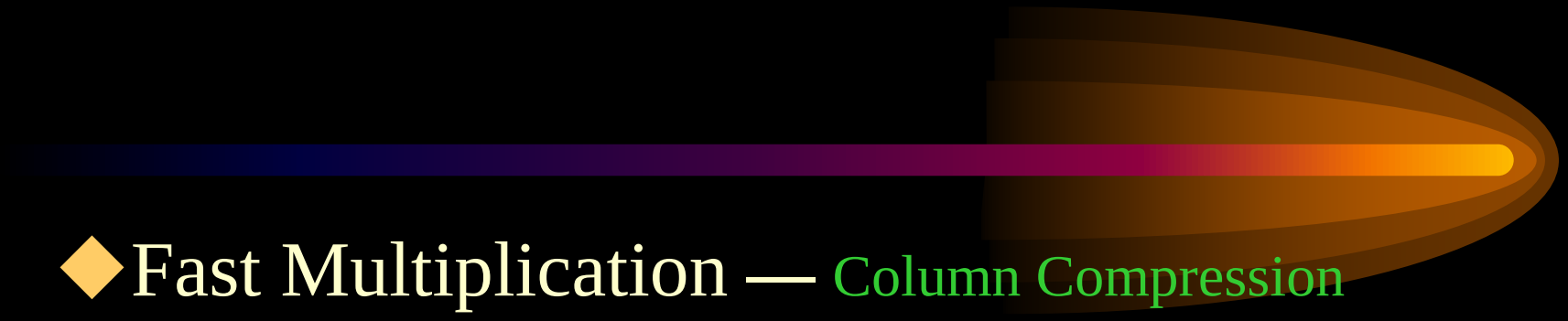
- Fast Multiplication
  - Full Adder Tree
    - Generate matrix of summands.
    - Use a binary tree of full-adders to calculate the result in a pipeline fashion

Fast Multiplication — Full Adder Tree



- Questions
  - For an n * n multiplication, calculate the execution time of the operation using full adder tree scheme.
  - Show the "snap shots" of the events to perform: 1101 * 1011 using full adder tree scheme.

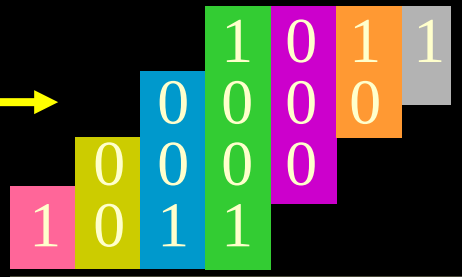
- Fast Multiplication — Column Compression
  - Assume that a population counter is available that can count the number of 1^s in an n-bit word, producing a $1+\lfloor \log_2 n \rfloor$ bit result.
  - Similar to the reduction of summands technique one can go through several reduction stages to compress the number of bits in each column.



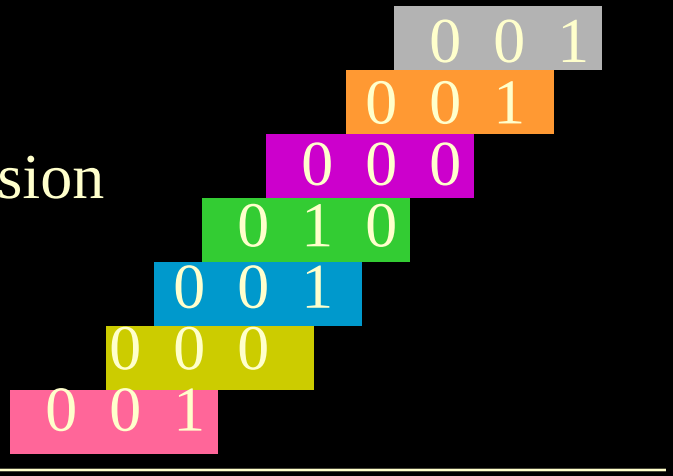
- Fast Multiplication — Column Compression
  - Generate matrix of summands.
  - Go over several stages using population counters.
  - In final stage (2 elements in each column) use a fast adder to generate the result.

- Fast Multiplication — Column Compression

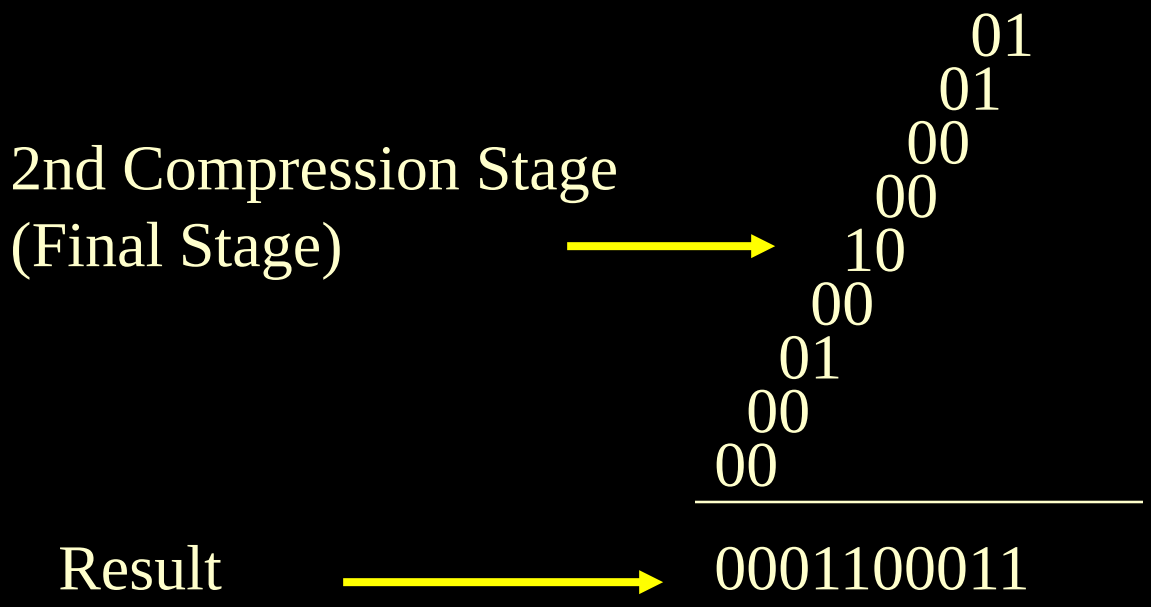
Matrix of Summands



1st Compression Stage



- Fast Multiplication — Column Compression



- Question
  - Formulate the execution time of an n * n multiplier unit using column compression scheme.

- Division
  - Similar to multiplication, one can develop a routine to perform division as a sequence of subtractions.
  - However, such an algorithm is very inefficient and slow.
  - Instead one can develop an algorithm which performs division as a sequence of Compare, Shift and Subtract operations.

Division

- One should note that division in a binary system is much simpler than the division in decimal system, since the quotient digits are either 0 or 1.
- To minimize the hardware requirements, we should remember that:
  - Comparison can be performed via arithmetic operation (s).
  - Subtraction can be performed via complement-addition.
- In other words; division requires almost the same hardware modules as multiplication does.

Division

- Division can be carried out as a sequence of *n* iterations.
- Dividend is a double register.
- One bit of the quotient is generated in each iteration.
- At the end of the operation, the quotient is in the 1st half part of the double register (low-order part), and remainder is in the 2nd half part.
- Sign of the quotient is the X-OR of the signs of dividend and divisor.
- Sign of the remainder is the same as the sign of the dividend.

- Division
  - Methods of Division
    - There are several different algorithms for division:
      - Restoring Method
      - Non-Restoring Method
      - Direct Comparison

- Fast Division — SRT Method
  - Faster direct division can be developed on normalized numbers by observing sequences of more than one bit of the dividend or partial remainder - i.e., sequences of 0^s and 1^s can be skipped.
  - This method was proposed to improve binary floating-point arithmetic.

Fast Division — SRT Method

- Assumptions
  - The dividend and divisor are binary fractions.
  - The divisor (B) is an n-bit normalized number — i.e., $B = .1b_{n-2} \ldots b_1 b_0$, $.5 \le B < 1$.
  - The dividend-quotient (AQ) combination is a 2n-bit register - i.e.,

$$AQ = \overbrace{.00 \ldots 0}^{k \text{ Zeros}} \; 1 \; a_{n-(k+2)} \; a_1 \; a_0 \; q_{n-1} \; \ldots \; q_1 \; q_0$$

- Fast Division — SRT Method
  - Assumptions
    - The dividend is normalized during the division operation.
    - Divide overflow condition will be detected and steps are taken in order for it to be overcome.

- Fast Division — SRT Method
  - The divisor is normalized and the dividend-quotient combination is adjusted by shifting it left the same number of positions that the divisor was shifted during normalization.
  - This step allows that the relative magnitudes of divisor and dividend remain the same.

- Fast Division — SRT Method
  - AQ is normalized — i.e., for each shift left a 0 is inserted for q_0 — Skipping over zeros.

$$AQ = .1\ a_{n-2} \ \ldots\ a_1\ a_0\ q_{n-1} \ \ldots\ q_k \underbrace{00 \ldots 0}_{\text{K Zeros}}$$

  - After this step, repeat the following sequence of steps:

- Fast Division — SRT Method
  - Subtract divisor from the dividend:
    - If positive result, a 1 is inserted for q_0 and left shift AQ register.

- Fast Division — SRT Method
  - If negative result - i.e.,

$$AQ = 1.1 \ldots \quad a_1 a_0 q_{n-1} \quad \overbrace{\ldots 00 \ldots 0}^{k}$$

    - Insert 0 for q_0 and shift left AQ register
    - Shift over 1^s, and insert 1^s until

$$AQ = 1.0 \ldots \quad a_1 a_0 q_{n-1} \quad \overbrace{\ldots 0 \ldots 0}^{k+1} \quad \overbrace{111 \ldots 1}^{m}$$

    - Add B to A and shift AQ to left

- Fast Division — SRT Method
  - Perform the following operation

	A	Q	
AQ =	.00000	10111	$(23 * 2^{-10})$
B =	.00101		$(5 * 2^{-5})$
Normalized B =	.10100		

Normalized B = 1.01100

- Fast Division — SRT Method

	.00000	10111
Adjust AQ	.00010	111**
Shift over 0^s	.10111	**000
Subtract B	1.01100	
Positive Result:	0.00011	**000
Shift AQ left, $q_0 \leftarrow 1$	.0011*	*0001
Shift over 0^s	.11**0	00100

Remainder

Quotient

- Fast Division — SRT Method

	A	Q	
AQ =	.00001	10111	$(55 * 2^{-10})$
B =	.01010		$(10 * 2^{-5})$
Normalized B =	.10100		
Normalized B =	1.01100		

Fast Division — SRT Method

	.00001	10111
Adjust AQ	.00011	0111*
Shift over 0^s	.11011	1*000
Subtract B	1.01100	
Positive Result:	0.00111	1*000
Shift AQ left, $q_0 \leftarrow 1$	.01111	*0001
Shift over 0^s	.1111*	00010
Subtract B	1.01100	
Positive Result:	0.0101*	00010
Shift AQ left, $q_0 \leftarrow 1$	.101* 0	00101

- Fast Division — SRT Method

	A	Q
AQ =	.00101	00100
B =	.01111	
Normalized B =	.11110	
Normalized B =	1.00010	

Fast Division — SRT Method

	.00101	00100
Adjust AQ	.01010	0100*
Shift over 0s	.10100	100*0
Subtract B	1.00010	
Negative Result:	1.10110	100*0
Shift AQ left, $q_0 \leftarrow 0$	1.01101	00*00
Add B	.11110	
Positive Result:	0.01011	00*00
Shift AQ left, $q_0 \leftarrow 1$	.10110	0*001
Subtract B	1.00010	
Negative Result:	1.11000	0*001

Fast Division — SRT Method

Negative Result:	1.11000	0*001
Shift AQ left, $q_0 \leftarrow 0$	1.10000	*0010
Shift over 1^s	1.0000*	00101
Add B	.11110	
Negative Result:	1.1111*	00101
Shift AQ left, $q_0 \leftarrow 0$	1.111*0	01010
Correct remainder by	1.1111*	
shifting A and adding B	.11110	
	0.1110*	

quotient

Remainder

- Fast Division — Divisor Reciprocation
  - This method generates the reciprocal of the divisor using an iterative process, and then obtains the quotient by multiplying the dividend by the divisor reciprocal

$$A/B = A * (1/B)$$

- Fast Division — Divisor Reciprocation
  - The divisor (B) is assumed to be a positive and normalized number,

    $1/2 \le B < 1 \Rightarrow 1 < 1/B \le 2$

  - An initial value $X_0 \approx 1/B$ is determined using a ROM table or a combinational logic circuit,

    $B = .1b_2b_3 \ldots b_n \Rightarrow X_0 = 1.d_1d_2 \ldots d_n$

Fast Division — Divisor Reciprocation

- Then the following iterative cycles will be performed to determine the inverse value with reasonable accuracy

$$a_0 = B x_0 \quad \begin{cases} x_1 = x_0(2 - a_0) \\ a_1 = a_0(2 - a_0) \end{cases}, \quad \begin{cases} x_2 = x_1(2 - a_1) \\ a_2 = a_1(2 - a_1) \end{cases}, \quad \begin{cases} x_n = x_{n-1}(2 - a_{n-1}) \\ a_n = a_{n-1}(2 - a_{n-1}) \end{cases}$$

- The number of iterations (n) will be chosen to satisfy the following relation:

$$|1 - B x_n| \leq \varepsilon$$

Relative error

- Fast Division — Divisor Reciprocation
  - Assume B = .75 ⇒ 1/B = 1.3333 …
  - Take X_0 = 1, naturally X_0 is not the exact inverse of B and the error is δ = .333333...

$X_1 = X_0 (2-BX_0) = 1 (2-.75) = 1.25$ δ = .08333...

$X_2 = X_1 (2-BX_1) = 1.25 (2-.75 * 1.25) = 1.328125$
δ = .005208333...

$X_3 = X_2 (2-BX_2) = 1.328125 (2-.75 * 1.32815)$
=1.333313 δ = .000020333...

Newton's Method

- Fast Division — Multiplicative Division
  - The operation of division is replaced by that of finding a factor *F* such that:

    B * F =1 and A * F = Q
  - An iterative method can be used to determine *F*.

- Fast Division — Multiplicative Division
  - In each iteration a constant factor **(**multiplying factor**)** F_i $(1 \leq i \leq n)$ is calculated to converge the denominator (Divisor) rapidly toward 1.

$$Q = \frac{A * F_0 * F_1 * \ldots * F_n}{B * F_0 * F_1 * \ldots * F_n}$$

- Fast Division — Multiplicative Division
  - The numerator (dividend) and the denominator (divisor) are both positive fractions.
  - The divisor is a normalized number and the dividend is shifted accordingly.

$$1/2 \leq B < 1, \quad B = 1 - \delta \quad \Rightarrow 0 < \delta \leq 1/2$$

- Fast Division — Multiplicative Division
  - $F_i^s (0 \le i \le n)$ are chosen such that $B_{i-1} < B_i$, Where

$$
\begin{aligned}
B_0 &= B^* F_0 \\
B_1 &= B^* F_0 * F_1 \\
&\vdots \\
B_{i-1} &= B * F_0 * F_1 * \ldots * F_{i-1} \\
B_i &= B * F_0 * F_1 * \ldots * F_{i-1} * F_i \\
&\vdots \\
B_n &= B_{n-1} * F_n
\end{aligned}
$$

- Fast Division — Multiplicative Division
  - $B = 1 - \delta \Rightarrow F_0 = 1 + \delta$, hence:

    $B_0 = (1- \delta)(1+ \delta) = 1- \delta^2 \Rightarrow$ B_0 is closer to 1 than B
  - $F_1 = 1+ \delta^2$ hence:

    $B_1 = B_0 * F_1 = (1- \delta^2)(1+ \delta^2) = (1- \delta^4)$

    .
    .
    .

    $F_i = 1+ \delta^{2^i}$

- Fast Division — Multiplicative Division
  - Note: The initial multiplying factor (F_0) can be obtained by a table look up.
  - Note: Since we are dealing with binary numbers

$$F_i = 1 + \delta^{2^i} = 2 - (1 - \delta^{2^i}) = 2 - B_{i-1} = \overline{B_{i-1}}$$

- Fast Division — Multiplicative Division
  - For each iteration two multiplications are required:
    - One to process the next denominator from which the next multiplying factor is obtained, and
    - One that produces the next numerator.

Fast Division — Multiplicative Division

F_0 Obtain	$B_0 = B * F_0$	$A_0 = A * F_0$
$F_1 = \tilde{B}_0$	$B_1 = B_0 * F_1$	$A_1 = A_0 * F_1$
	If $B_1 = 1$ then $A_1 = Q$, Terminate	
$F_2 = \tilde{B}_1$	$B_2 = B_1 * F_2$	$A_2 = A_1 * F_2$
	If $B_2 = 1$ then $A_2 = Q$, Terminate	

- Residue Arithmetic
  - Concept
  - Advantages
  - Disadvantages