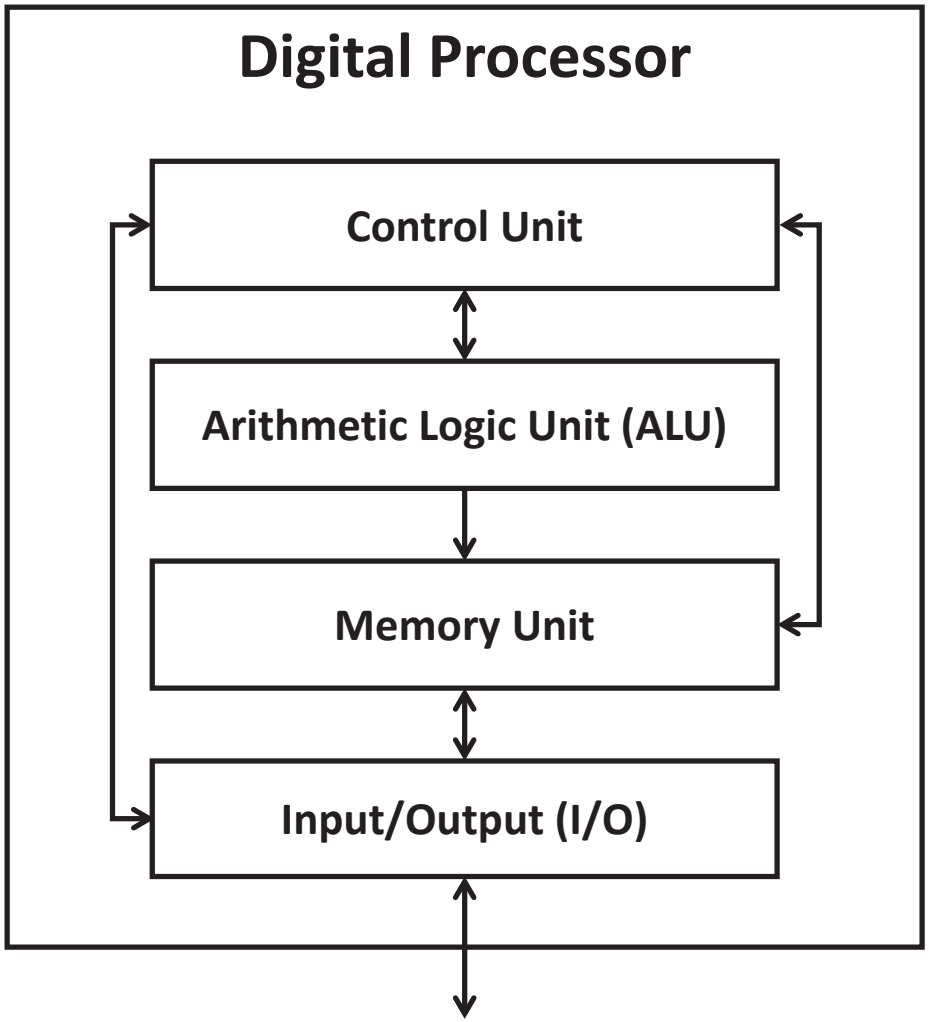
Week 4-1

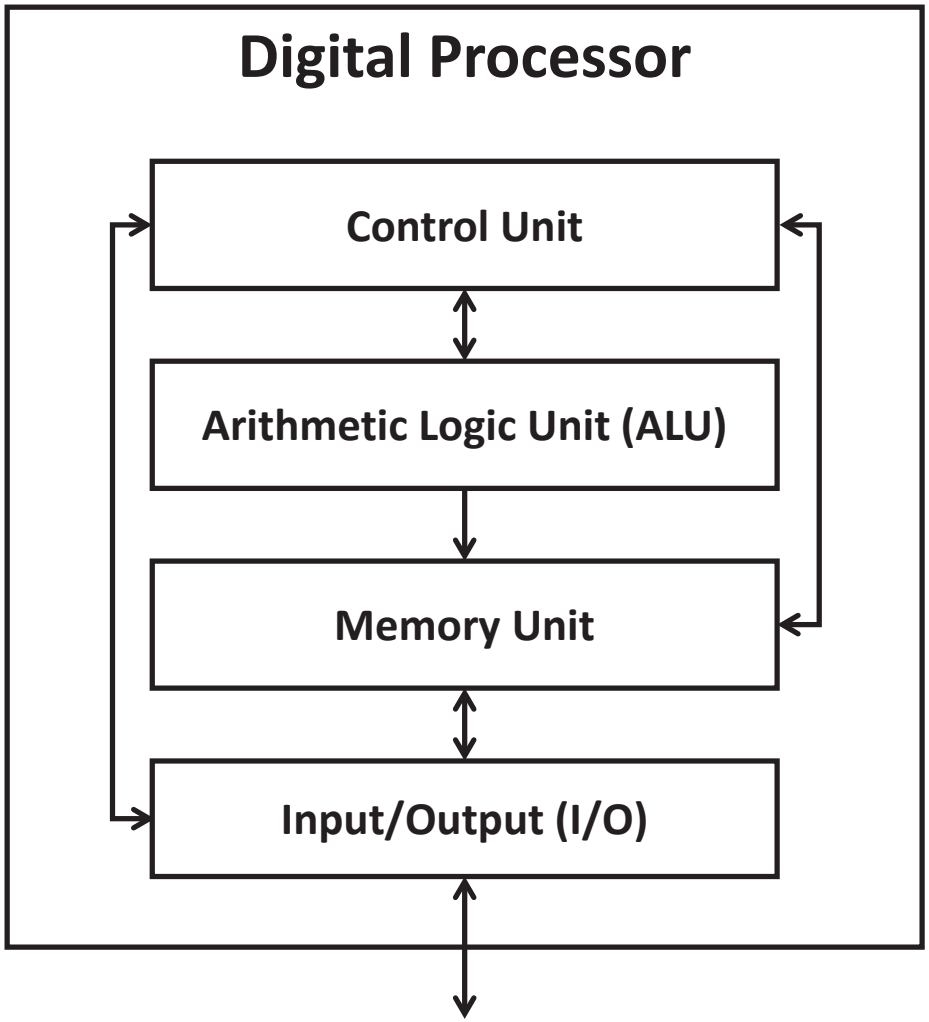
Introduction to the Arithmetic Logic Unit (ALU)

The Digital Processor



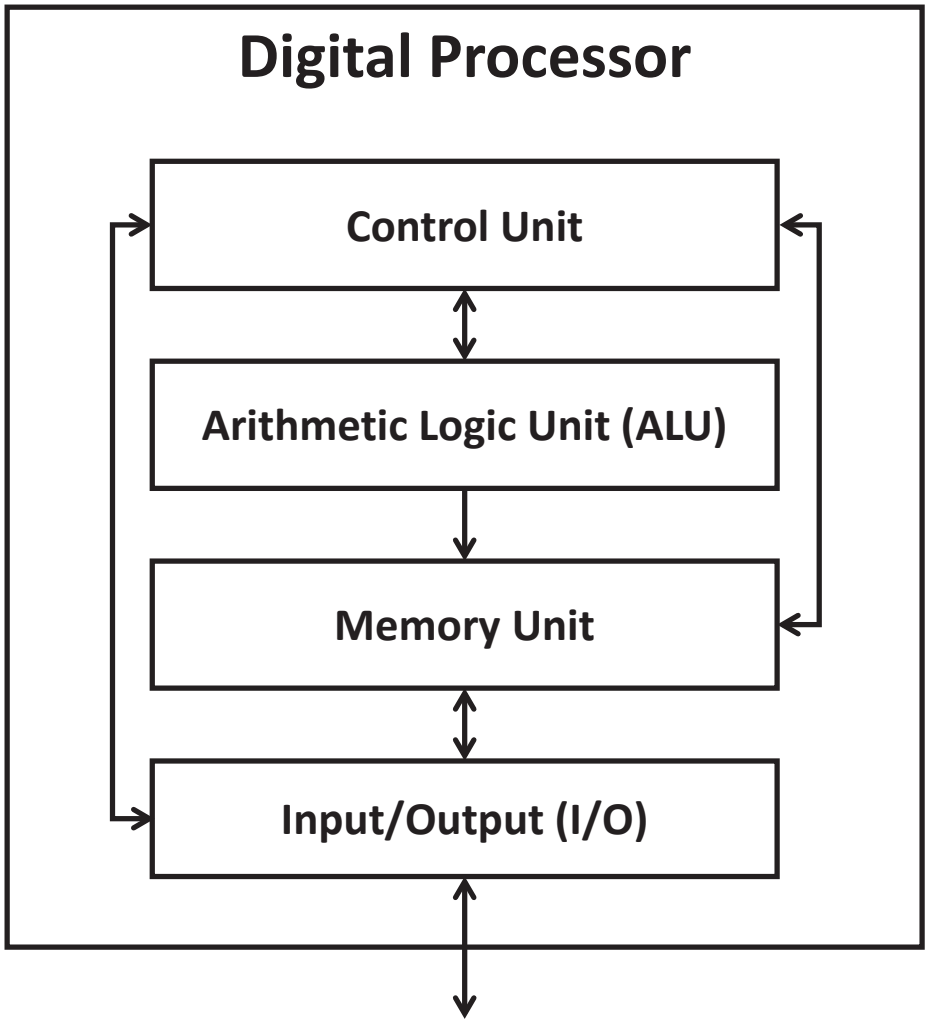
- The digital processor is fundamental to the modern digital computer
  - General purpose
    - Programmable
    - Tasks and problems are encoded into algorithms, which get executed as programs
- Partitioned into:
  - Control Unit
    - Finite state machine (PLA, random logic, etc.)
    - Counters
  - Arithmetic Logic Unit (ALU)
  - Memory Unit
  - Input/Output (I/O)

The Digital Processor



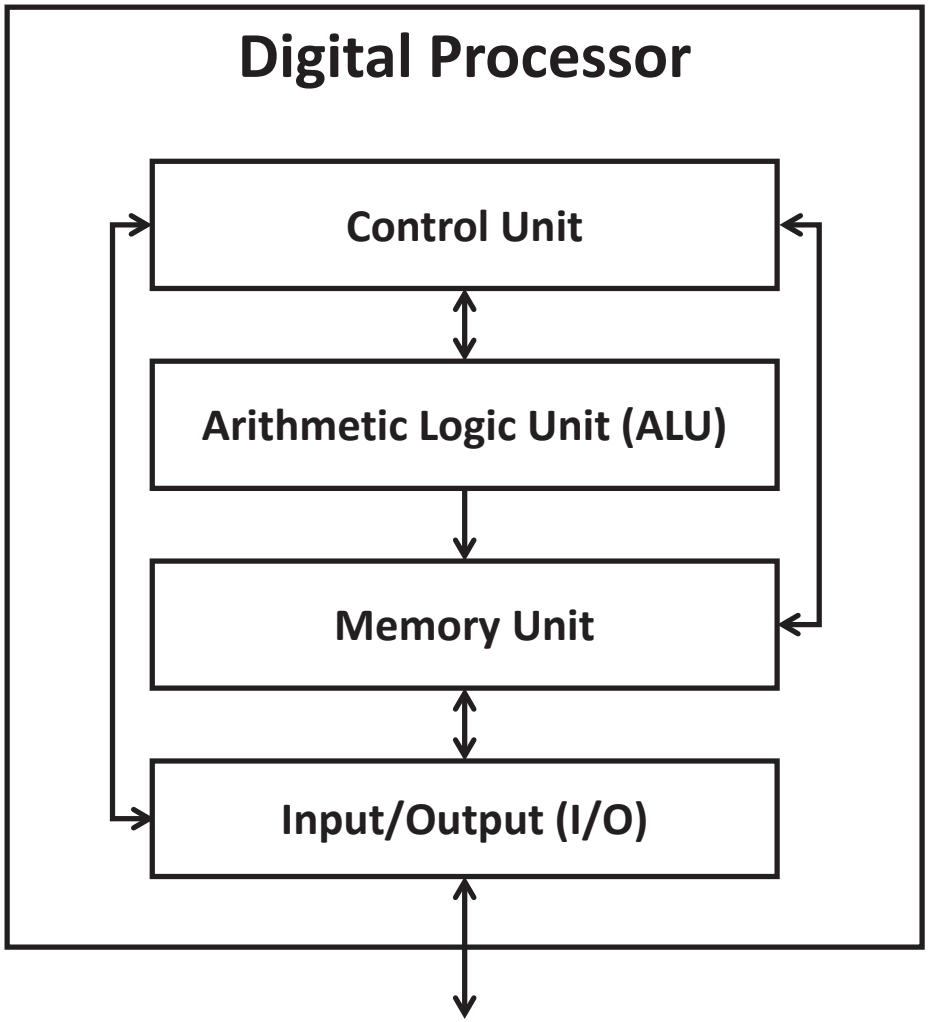
- The digital processor is fundamental to the modern digital computer
  - General purpose
    - Programmable
    - Tasks and problems are encoded into algorithms, which get executed as programs
- Partitioned into:
  - Control Unit
    - Sends commands to other units to execute programs
  - Arithmetic Logic Unit (ALU)
    - Bit-sliced datapath (adder, multiplier, shifter, comparator, etc.)
  - Memory Unit
  - Input/Output (I/O)

The Digital Processor



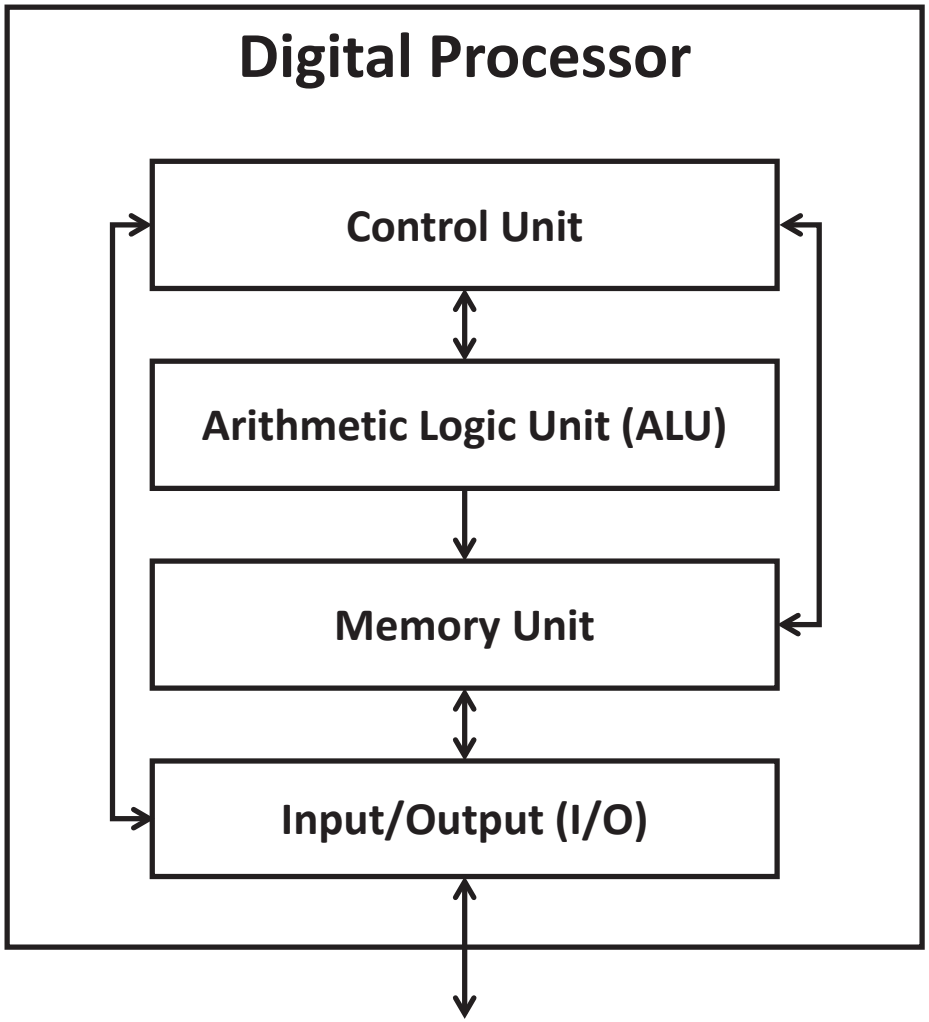
- The digital processor is fundamental to the modern digital computer
  - General purpose
    - Programmable
    - Tasks and problems are encoded into algorithms, which get executed as programs
- Partitioned into:
  - Control Unit
    - Sends commands to other units to execute programs
  - Arithmetic Logic Unit (ALU)
    - Performs computations
    - Addition, subtraction, comparison, etc.
  - Memory Unit
    - Random Access Mem. (RAM), Read-Only Mem. (ROM), Buffers, Registers
  - Input/Output (I/O)

The Digital Processor



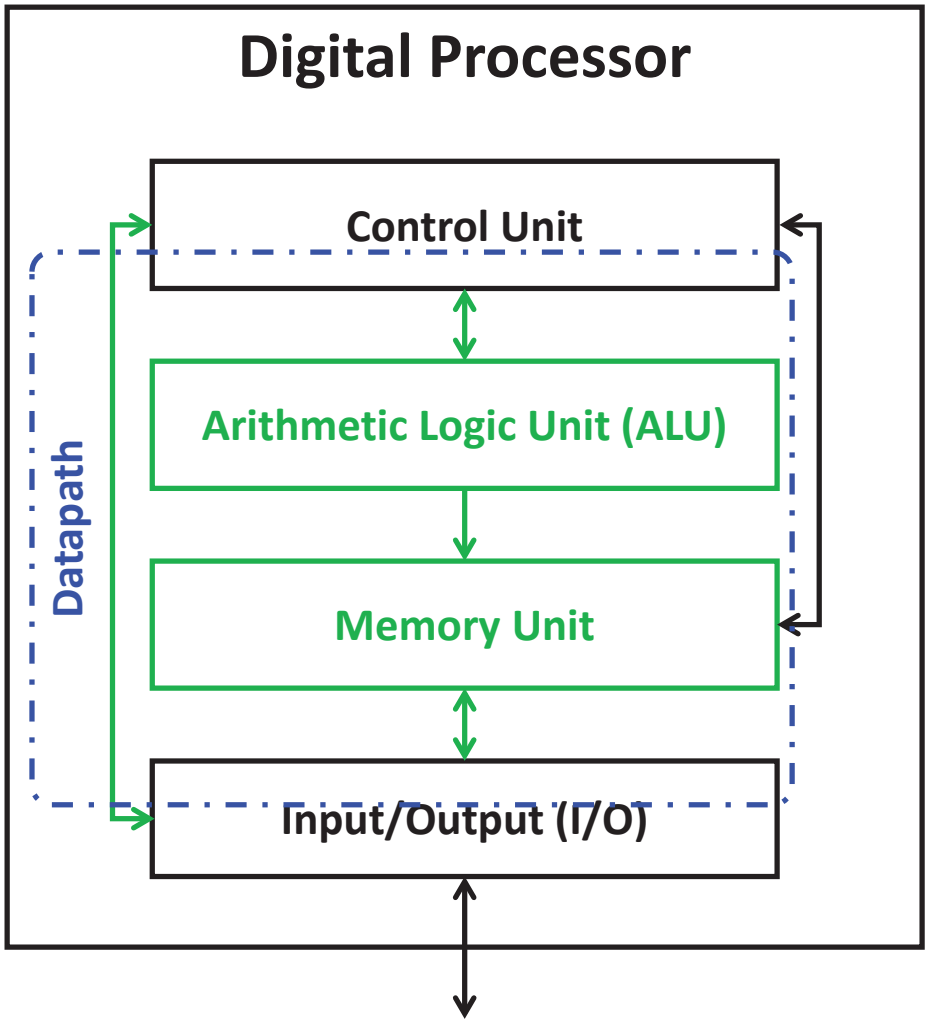
- The digital processor is fundamental to the modern digital computer
  - General purpose
    - Programmable
    - Tasks and problems are encoded into algorithms, which get executed as programs
- Partitioned into:
  - Control Unit
    - Sends commands to other units to execute programs
  - Arithmetic Logic Unit (ALU)
    - Performs computations
    - Addition, subtraction, comparison, etc.
  - Memory Unit
    - Stores data
    - Stores program
  - Input/Output (I/O)
    - Interconnect (switches, arbiters, bus)

The Digital Processor



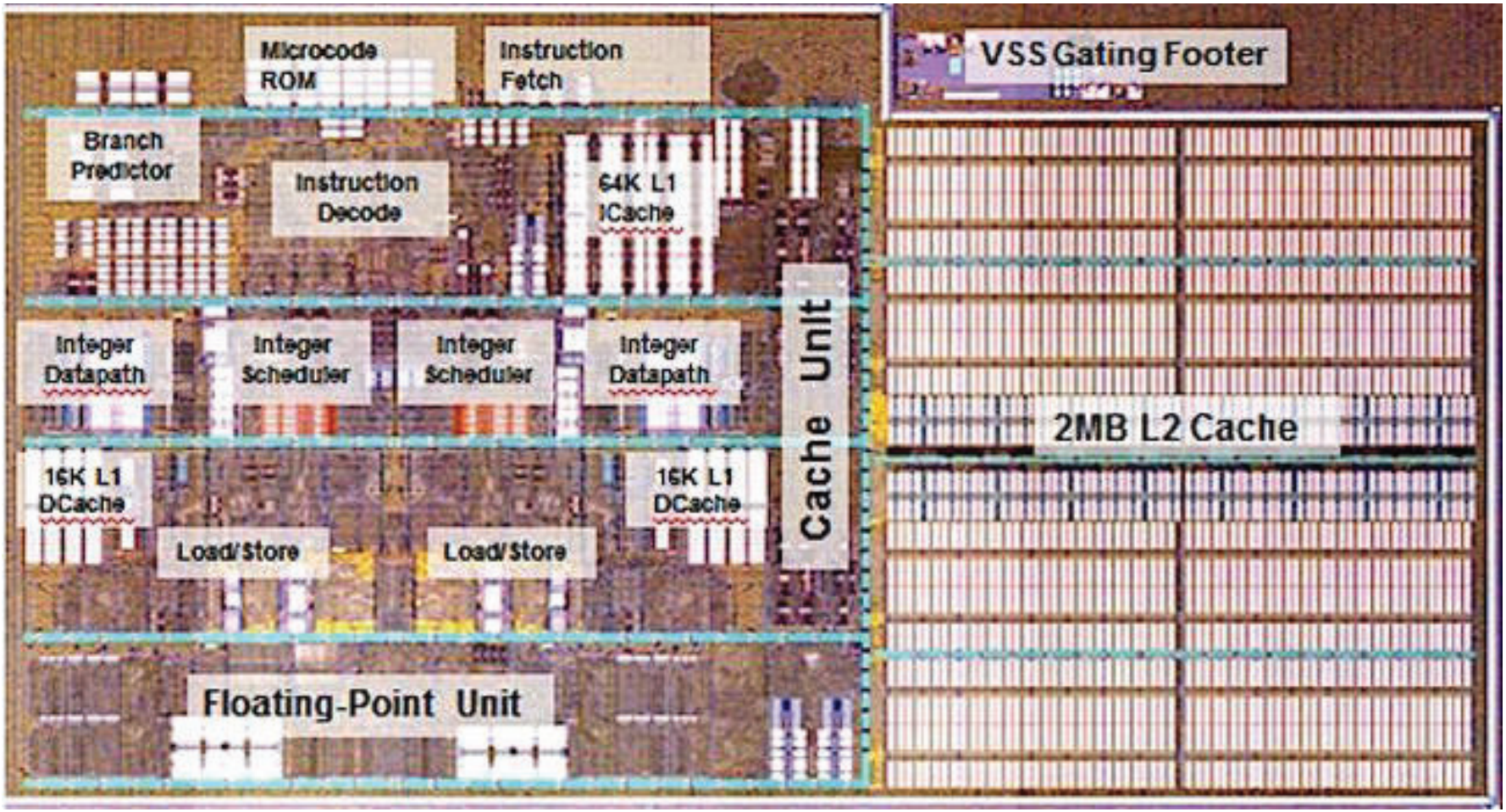
- The digital processor is fundamental to the modern digital computer
  - General purpose
    - Programmable
    - Tasks and problems are encoded into algorithms, which get executed as programs
- Partitioned into:
  - Control Unit
    - Sends commands to other units to execute programs
  - Arithmetic Logic Unit (ALU)
    - Performs computations
    - Addition, subtraction, comparison, etc.
  - Memory Unit
    - Stores data
    - Stores program
  - Input/Output (I/O)
    - Interface with sensors, humans, and other electronic systems

The Datapath



4b	= µController
8b	= Video (each RGB)
16b	= Audio
32/64b	= CPU
1024b	= filter, encryption

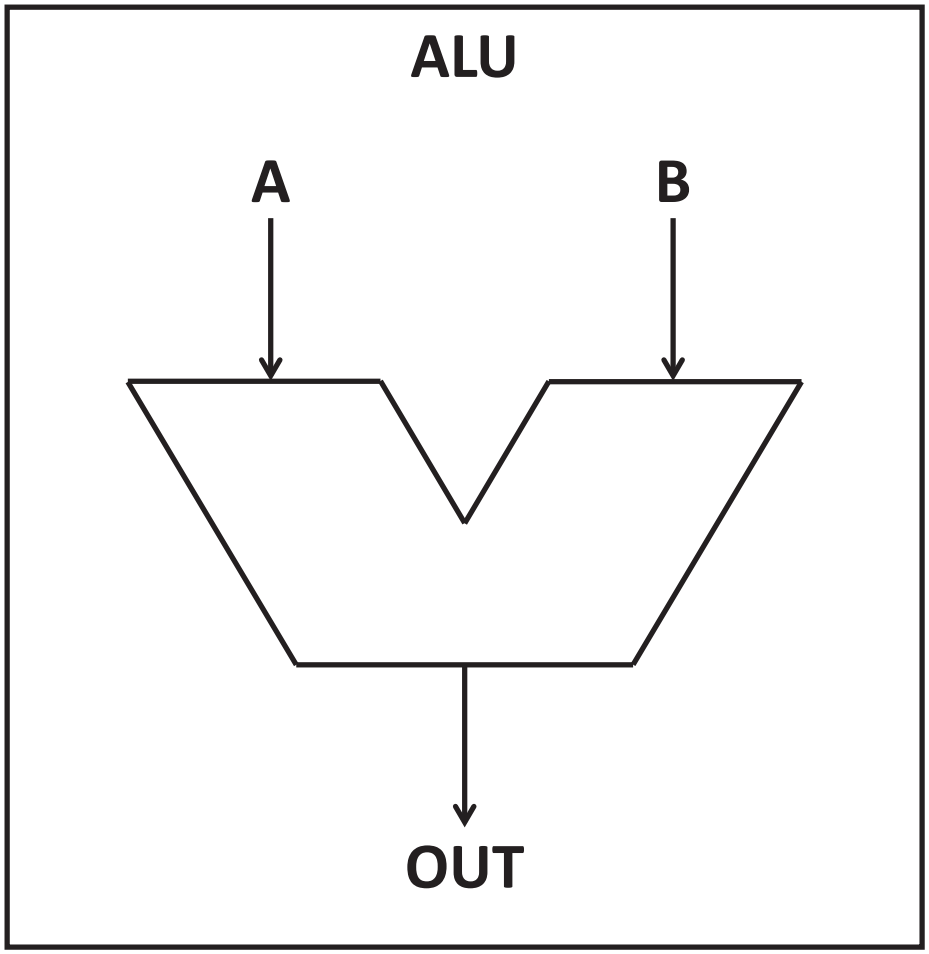
The x86 Datapath



AMD Bulldozer (launched in 2011 on 32nm CMOS technology)

Custom design is necessary for ALU and UHP logic

The Arithmetic Logic Unit (ALU)



The Arithmetic Logic Unit (ALU) is the part of the CPU performs various arithmetic and logic operations

- Additions and subtractions on data
- Multiplication and division on data
- Boolean logic such as AND, OR, XOR, etc.
- Bit-shifts (left/right, arithmetic/logical)
- Bit-rotations (left/right)
- Evaluate conditions
  - Subtract (A – B) and check if result is
    - zero (if A == B?)
    - negative (A < B)
    - positive (A > B)
- Calculate:
  - Addresses for program counter
    - Branch targets
  - Math functions (e.g. sine, exp, etc.)

Week 4-2

The Half and Full Adders

The Half-Adder (HA)

Half-adder

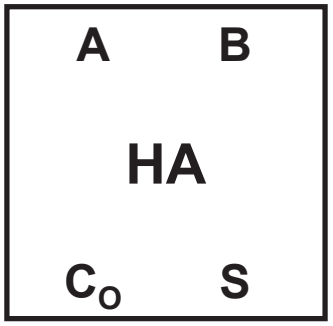
A	B	Carry (C_O)	Sum (S)
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

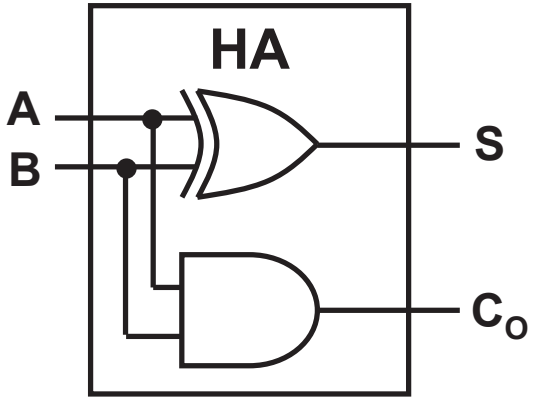
A	B	Carry (C_O)
0	0	0
0	1	0
1	0	0
1	1	1

A	B	Sum (S)
0	0	0
0	1	1
1	0	1
1	1	0

C_O: carry-out
Result of A+B: $C_O S$

$$C_O = A \cdot B$$
$$S = A \oplus B$$





The Full Adder (FA)

Full-adder

A	B	Carry-in (C_I)	Carry (C_O)	Sum (S)
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

C_O: carry-out
Result of A+B: $C_O S$

$$C_O = \overline{A} \cdot (B \cdot C_I) + A \cdot (B + C_I)$$
$$S = A \oplus B \oplus C_I$$

$$C_O = (B \cdot \overline{C_I}) + A \cdot (B + C_I)$$
If $C_O = 0, S = \overline{C_O} \cdot (A + B + C_I)$
If $C_O = 1, S = C_O \cdot A \cdot B \cdot C_I$

There is a flexibility in how to implement full adder:

- **C_O and S generated using independent logic gates**
  - **More area and power consumption (delay?)**
- **Some logic for generating C_O and S are shared**
  - **Lesser area and power consumption (delay?)**

Implementing the Full Adder (FA)

$$C_O = (B \cdot C_I) + A \cdot (B + C_I)$$

C_O is a majority operation:

Look at A, B and C_I and set C_O to:

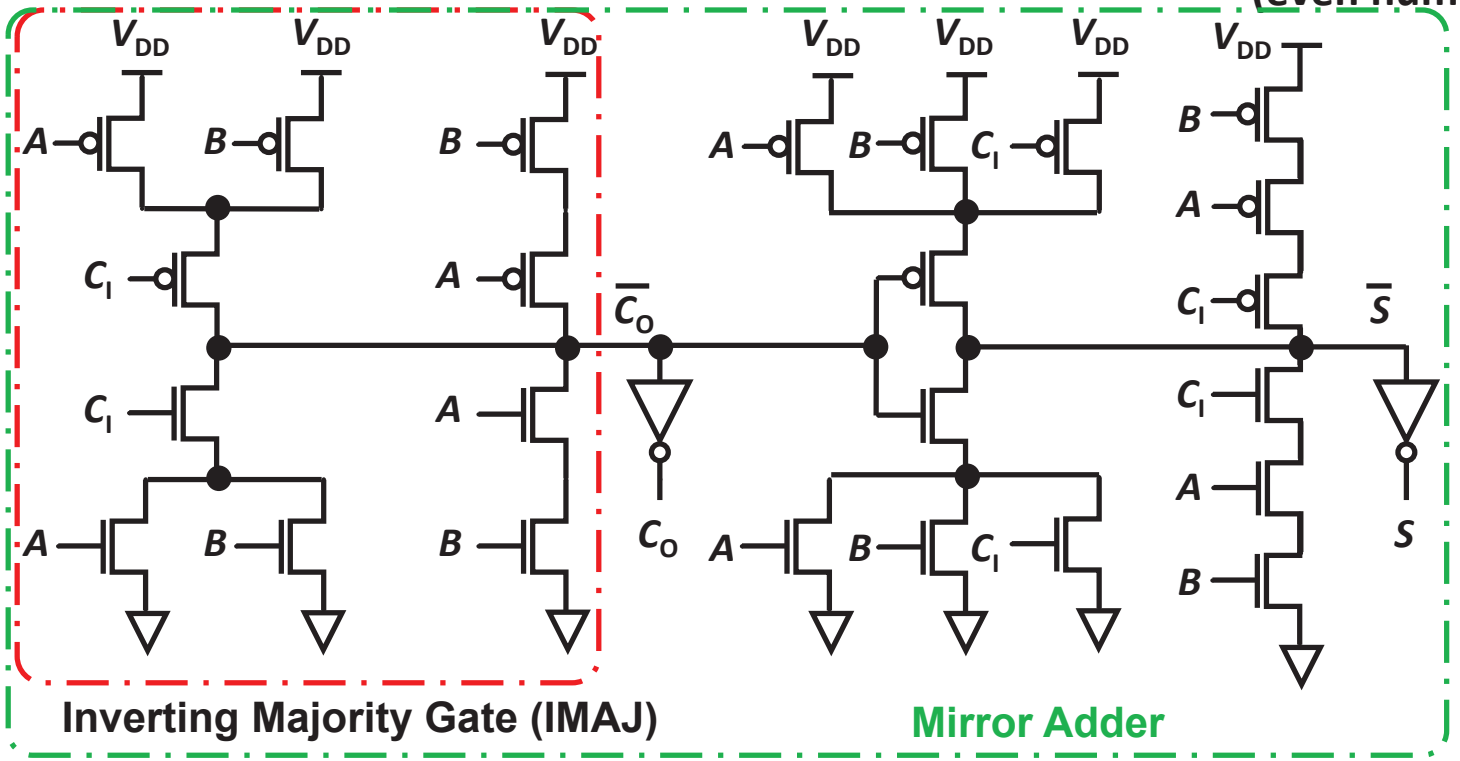
- **'1' if two or more inputs are '1'**
- **'0' if two or more inputs are '0'**

dual problems

$$S = A \oplus B \oplus C_I$$

Set S to:

- **'1' if inputs have odd number of '1' (even number of '0')**
- **'0' if inputs have odd number of '0' (even number of '1')**

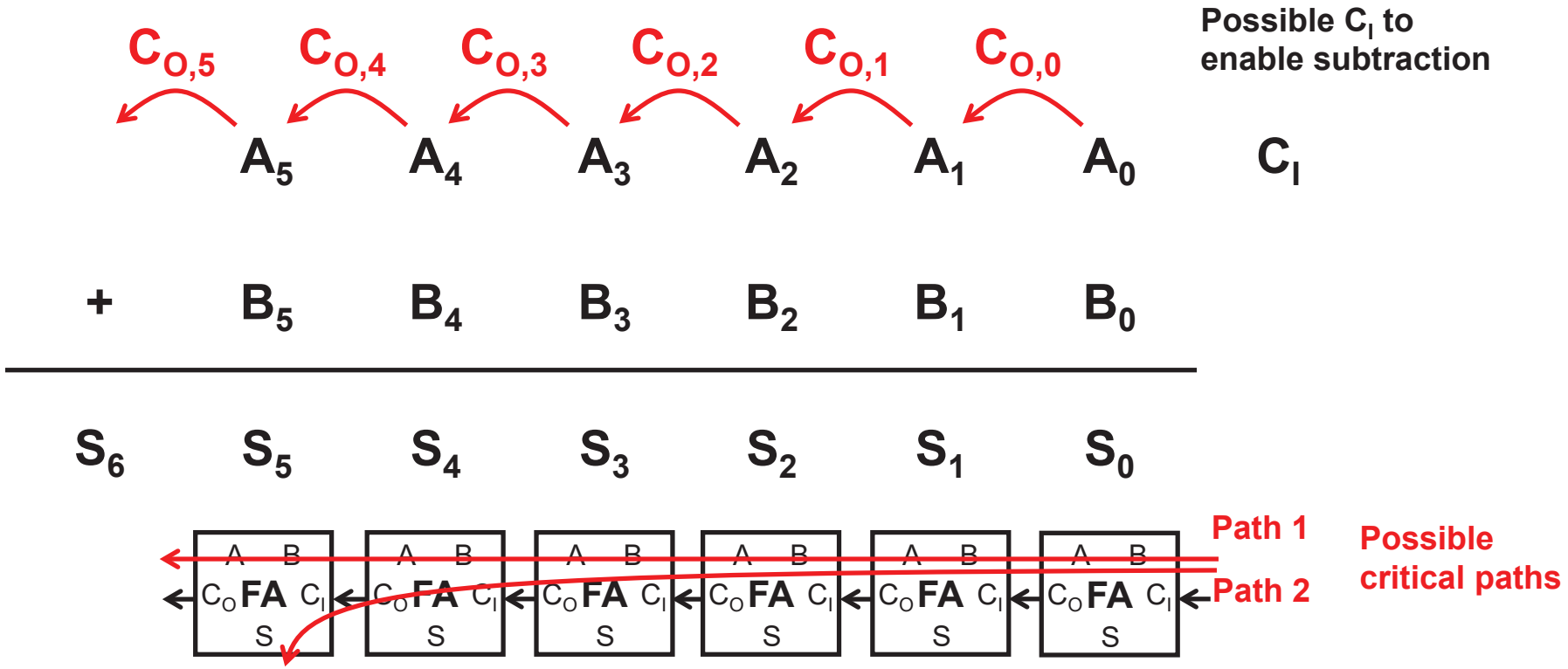


- **M1 ON: inputs have even number of '1'**
- **M2 ON: if inputs have even number of '0'**
- **Highly symmetric schematic enables symmetric layout (PDN and PUN have same graphs)**
- **Inverters improve drive**

Week 4-3

The Ripple-Carry Adder

Adding Two *N*-bit Numbers

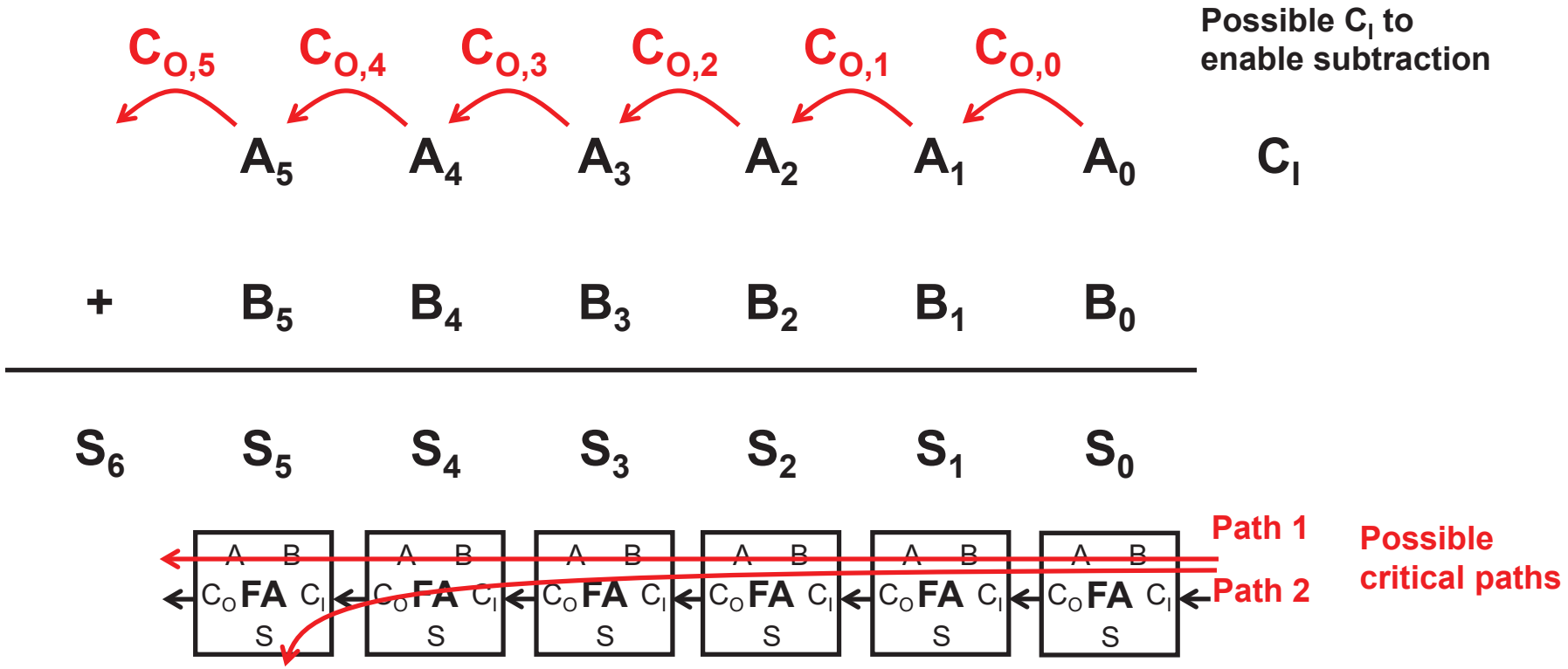


- **Ripple carry adder (RCA)**
  - **The full adder at each bit position needs to wait for carry from right**
  - **Carry "ripples" from C_I to C_O of the MSB**
  - **Critical path delay is from C_I to C_O or S of the MSB (how to improve?)**

Week 4-4

Optimizing the Carry Chain

Adding Two *N*-bit Numbers



- **Ripple carry adder (RCA)**
  - **The full adder at each bit position needs to wait for carry from right**
  - **Carry "ripples" from C_I to C_O of the MSB**
  - **Critical path delay is from C_I to C_O or S of the MSB (how to improve?)**

Implementing the Full Adder (FA)

$$C_O = (B \cdot C_I) + A \cdot (B + C_I)$$

C_O is a majority operation:

Look at A, B and C_I and set C_O to:

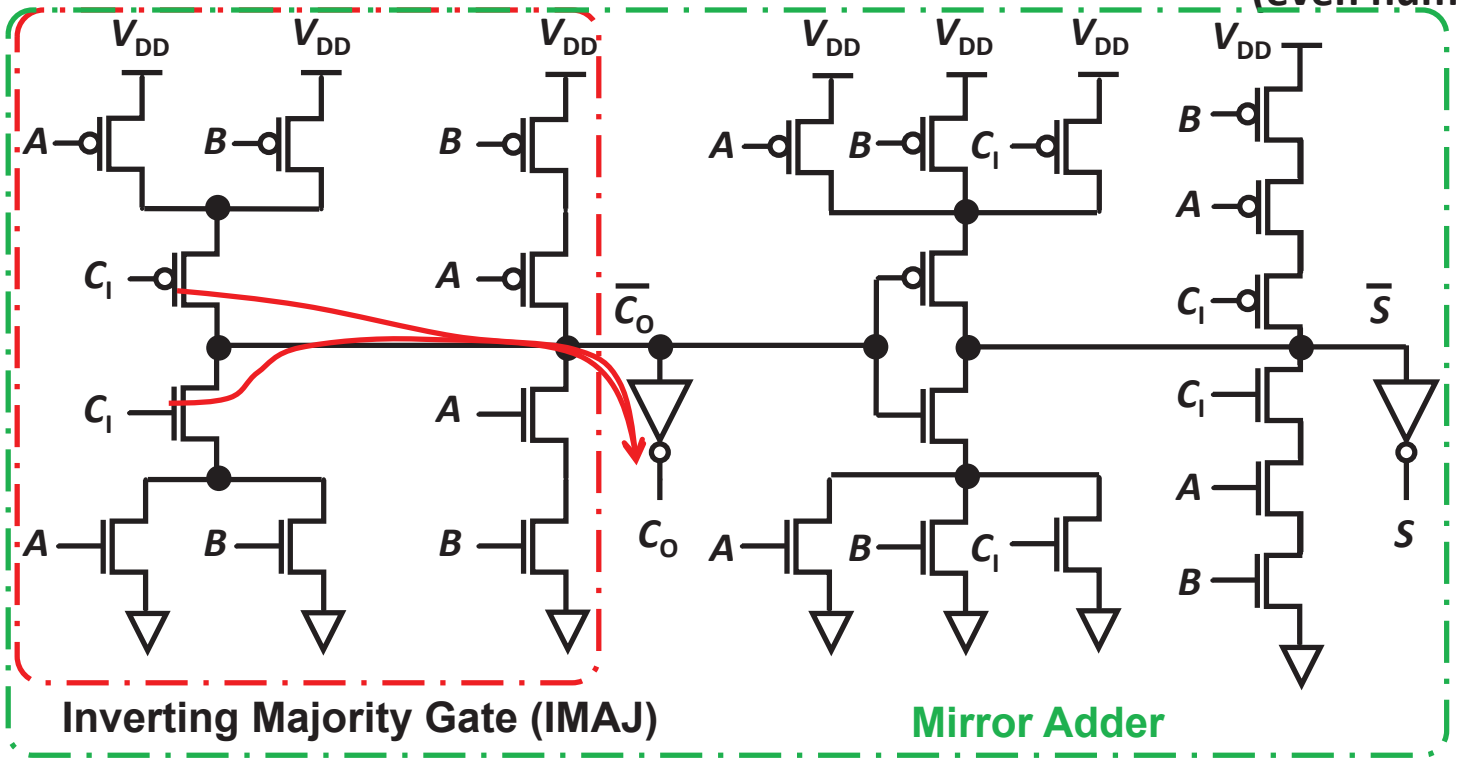
- **'1' if two or more inputs are '1'**
- **'0' if two or more inputs are '0'**

dual problems

$$S = A \oplus B \oplus C_I$$

Set S to:

- **'1' if inputs have odd number of '1' (even number of '0')**
- **'0' if inputs have odd number of '0' (even number of '1')**

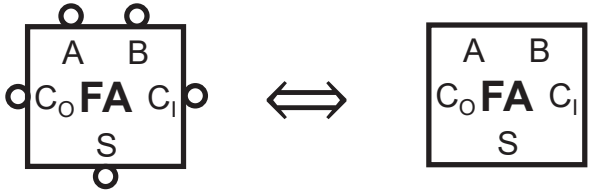


- **M1 ON: inputs have even number of '1'**
- **M2 ON: if inputs have even number of '0'**
- **Highly symmetric schematic enables symmetric layout (PDN and PUN have same graphs)**
- **Inverters improve drive**

The Full Adder (FA)

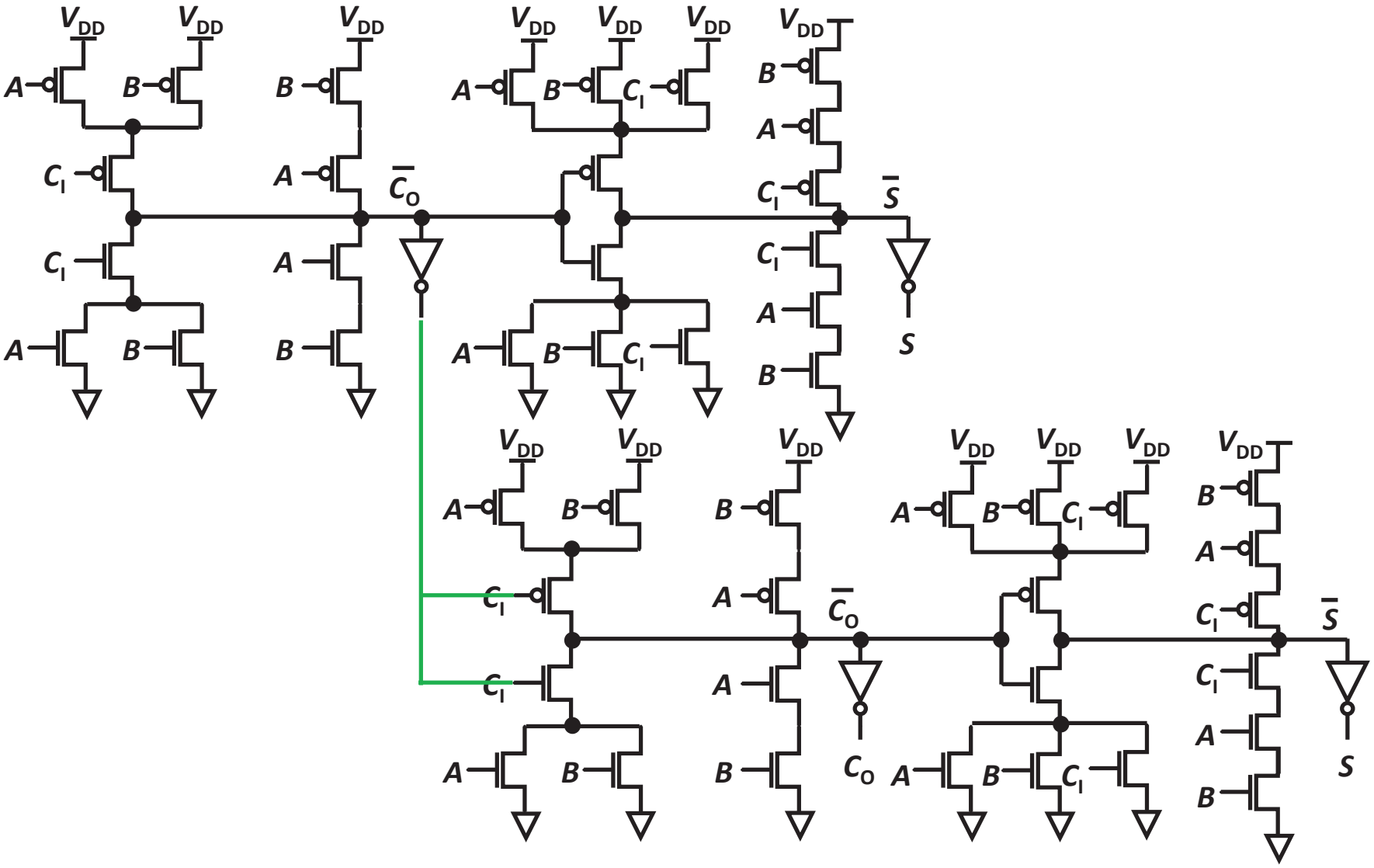
$\overline{A}$	$\overline{B}$	$\overline{C_I}$	$\overline{C_O}$	$\overline{S}$
1	1	1	1	1
1	1	0	1	0
1	0	1	1	0
1	0	0	0	1
0	1	1	1	0
0	1	0	0	1
0	0	1	0	1
0	0	0	0	0

A	B	Carry-in (C_I)	Carry (C_O)	Sum (S)
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

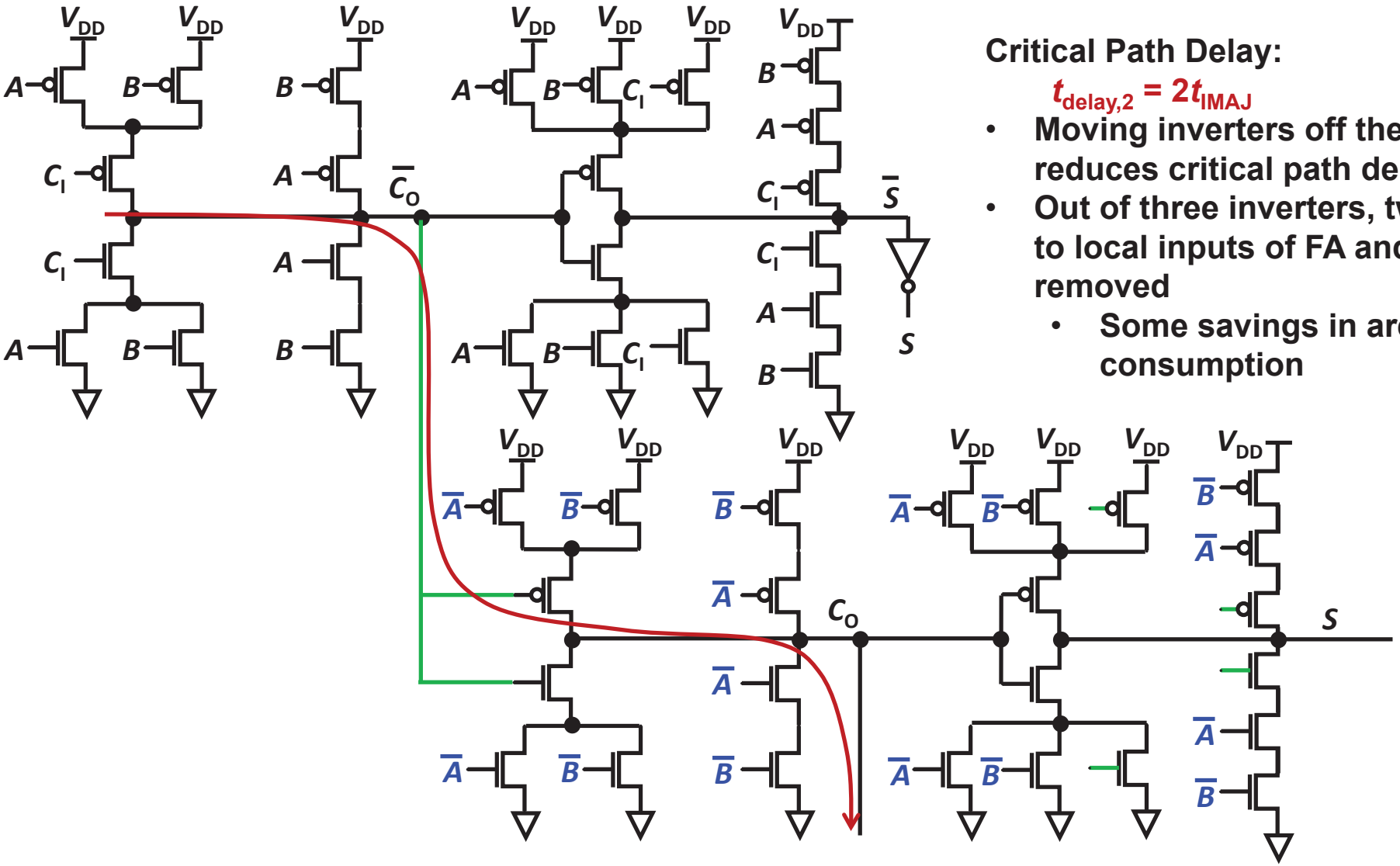


If all inputs to the FA are active high (low), then all outputs are active high (low)

A 2-bit Ripple Carry Adder



A 2-bit Ripple Carry Adder



Critical Path Delay:

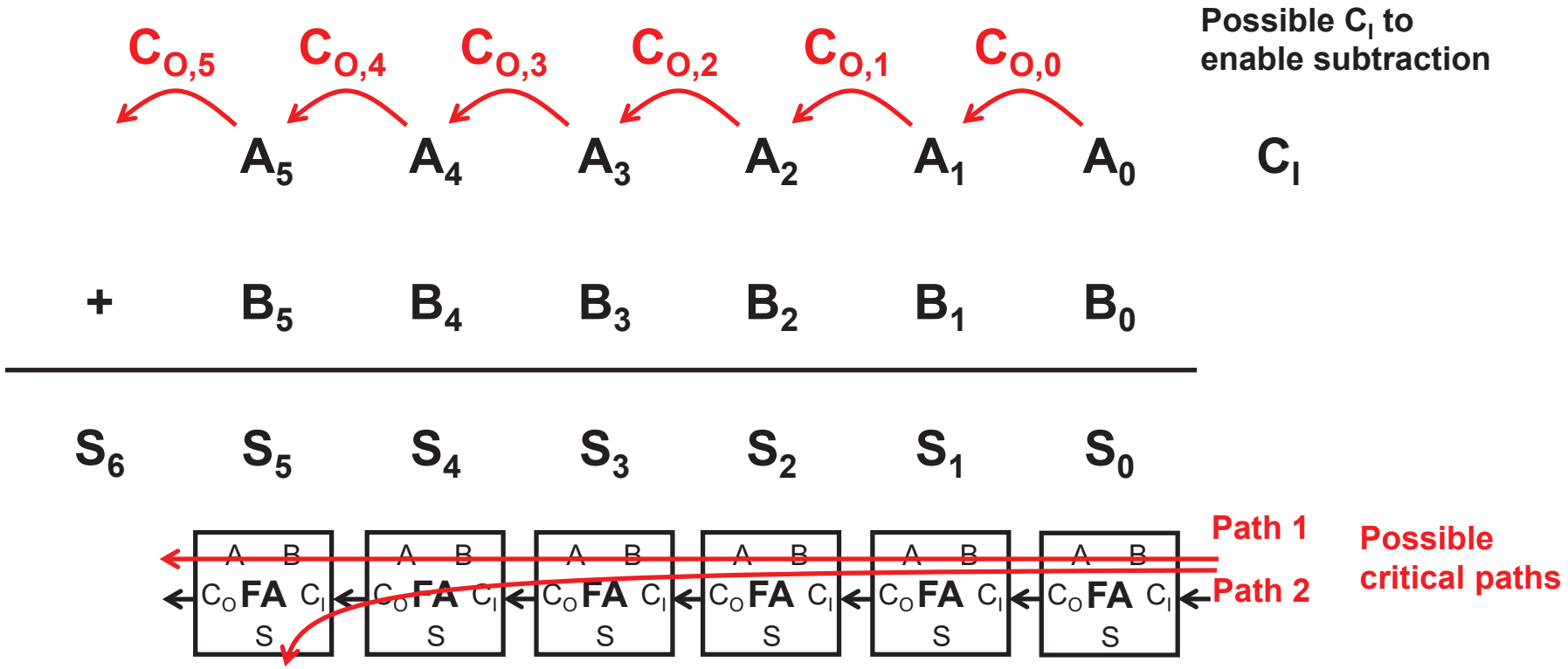
$t_{delay,2} = 2t_{IMAJ}$

- **Moving inverters off the critical path reduces critical path delay**
- **Out of three inverters, two are moved to local inputs of FA and one is removed**
  - **Some savings in area and power consumption**

Week 4-5

The Manchester Carry Chain

Adding Two *N*-bit Numbers



- **Ripple carry adder (RCA)**
  - **The full adder at each bit position needs to wait for carry from right**
  - **Carry "ripples" from C_I to C_O of the MSB**
  - **Critical path delay is from C_I to C_O or S of the MSB (how to improve?)**

Overcoming the Long Carry Delay

$$C_O = (B \cdot C_I) + A \cdot (B + C_I)$$
$$C_O = (A + B) \cdot C_I + A \cdot B$$
$$C_O = \underbrace{(A \oplus B)}_{\text{Local inputs guaranteeing } C_O = C_I} \cdot C_I + \underbrace{A \cdot B}_{\text{Local inputs guaranteeing } C_O = 1}$$

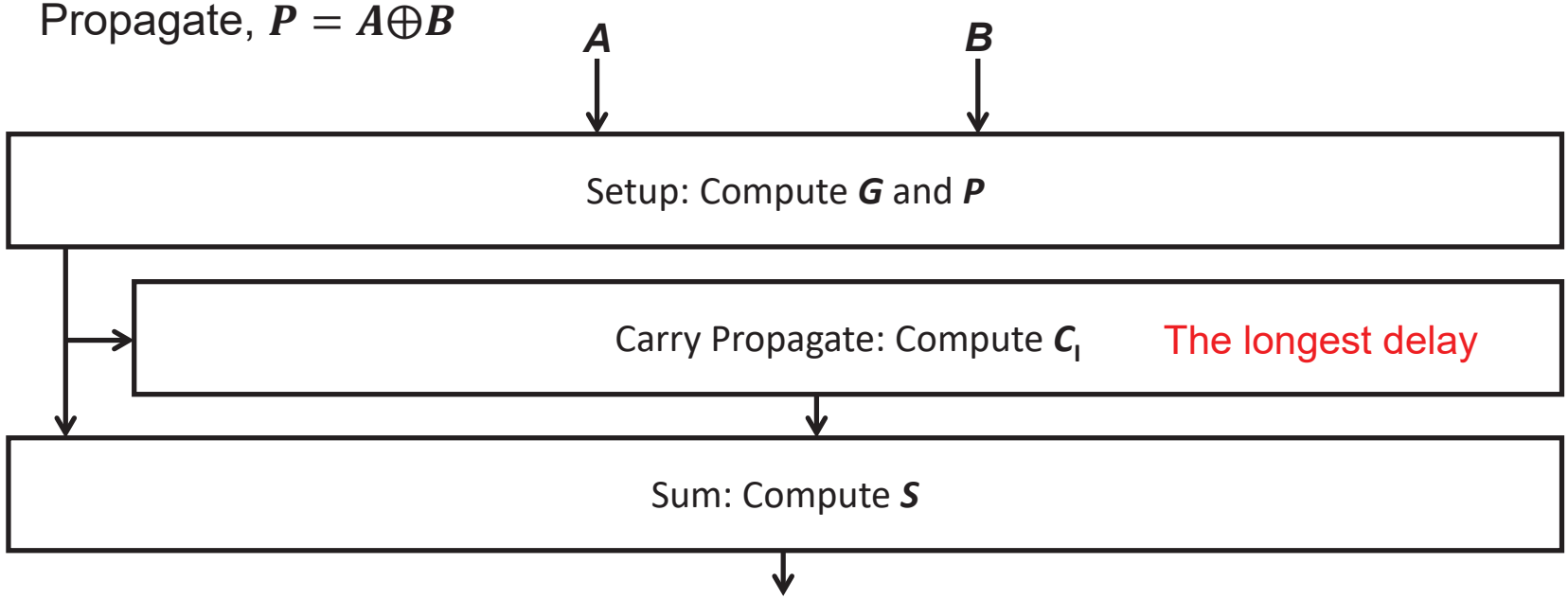
Generate, $G = A \cdot B$

Propagate, $P = A \oplus B$

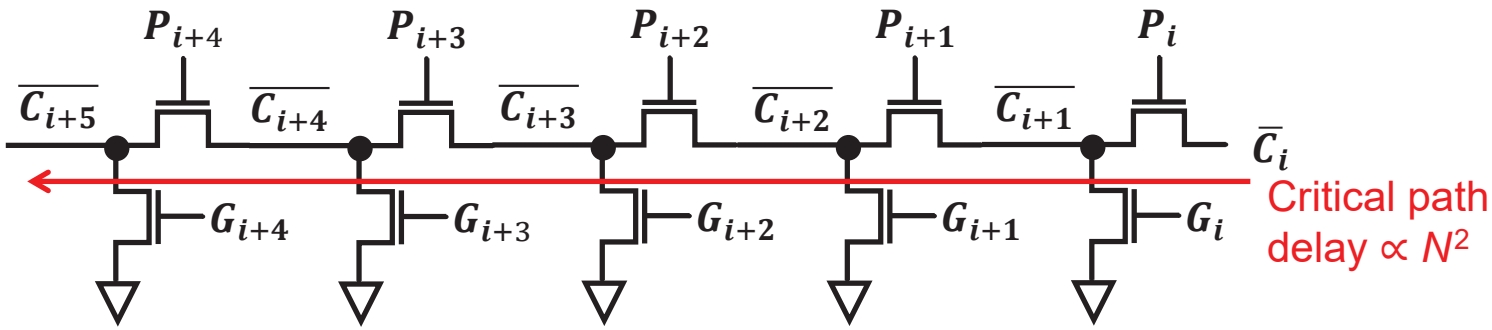
$$C_O = P \cdot C_I + G$$
$$S = P \oplus C_I = G \cdot C_I + P \cdot \overline{C_I}$$

Adder can be composed of 3 stages:

1. Compute ***G*** and ***P*** at each bit position from the corresponding ***A*** and ***B*** bits (Setup)
2. Compute C_I at every bit position (Carry Propagate)
3. Compute ***S*** at every bit position (Sum)



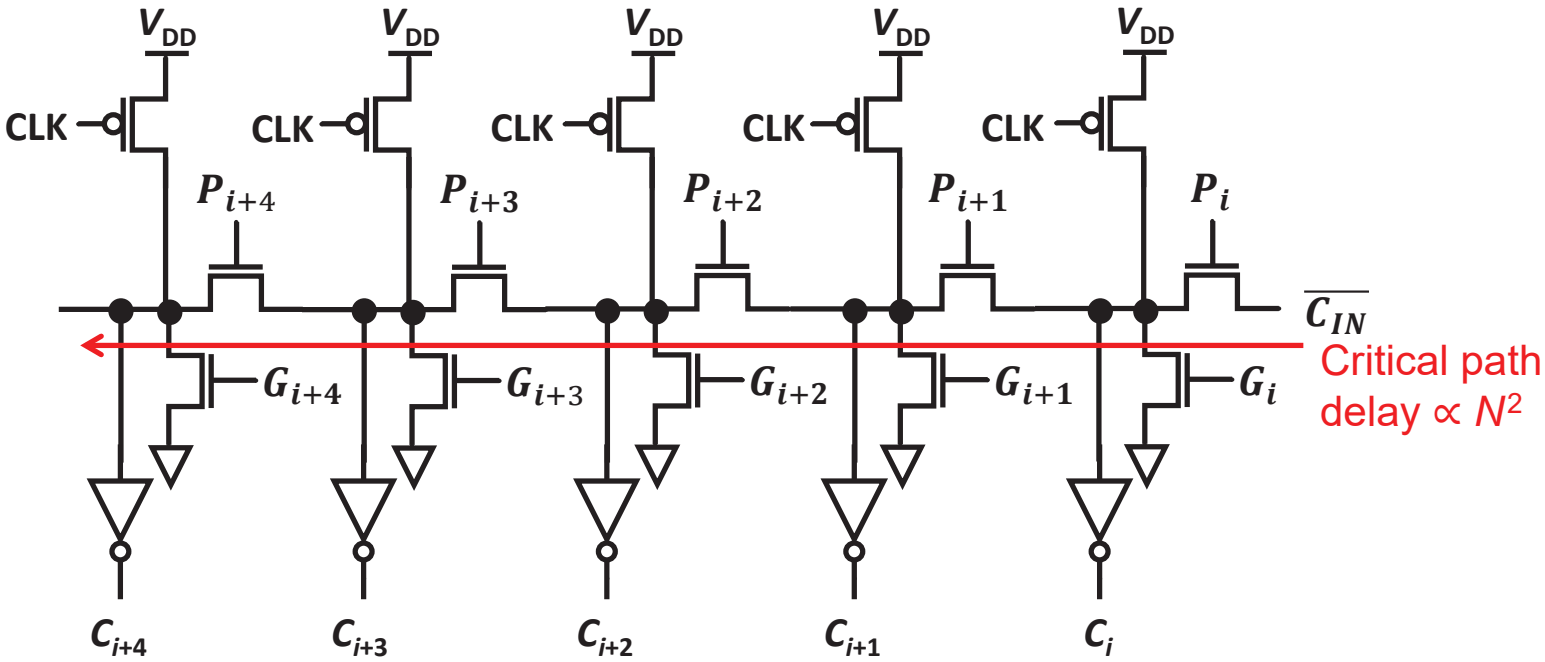
The Manchester Carry Chain



C_i: Carry-in of bit position i

- Pass transistor implementation
- 2N transistors (N is number of bits of adder)
- Suffers from V_{TN} drop

The Manchester Carry Chain



C_i: Carry-out of bit position i

- Dynamic logic implementation
- 5**N** transistors (**N** is number of bits of adder)
  - Might require footer NMOS
- Higher power consumption
- Inverters provide buffer to drive fanout

Overcoming the Long Carry Delay

$$C_O = (B \cdot C_I) + A \cdot (B + C_I)$$
$$C_O = (A + B) \cdot C_I + A \cdot B$$
$$C_O = \underbrace{(A \oplus B)}_{\text{Local inputs guaranteeing } C_O = C_I} \cdot C_I + \underbrace{A \cdot B}_{\text{Local inputs guaranteeing } C_O = 1}$$

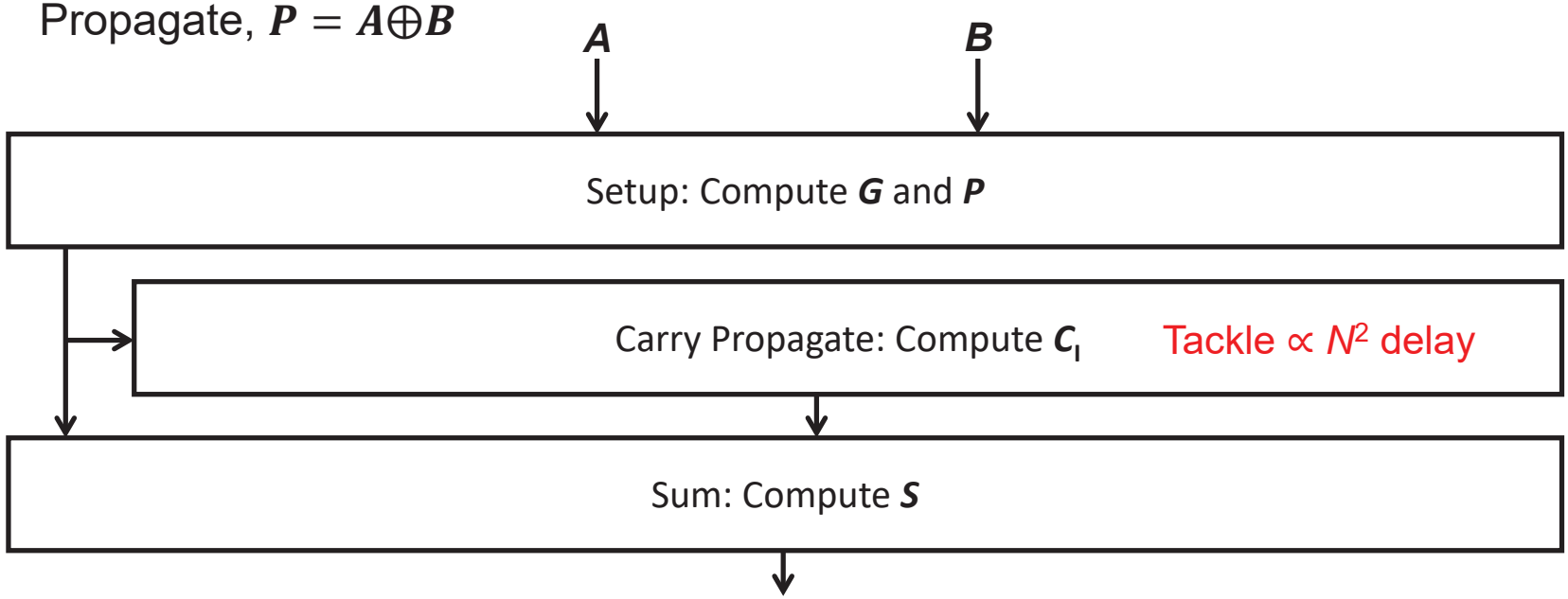
Generate, $G = A \cdot B$

Propagate, $P = A \oplus B$

$$C_O = P \cdot C_I + G$$
$$S = P \oplus C_I = G \cdot C_I + P \cdot \overline{C_I}$$

Adder can be composed of 3 stages:

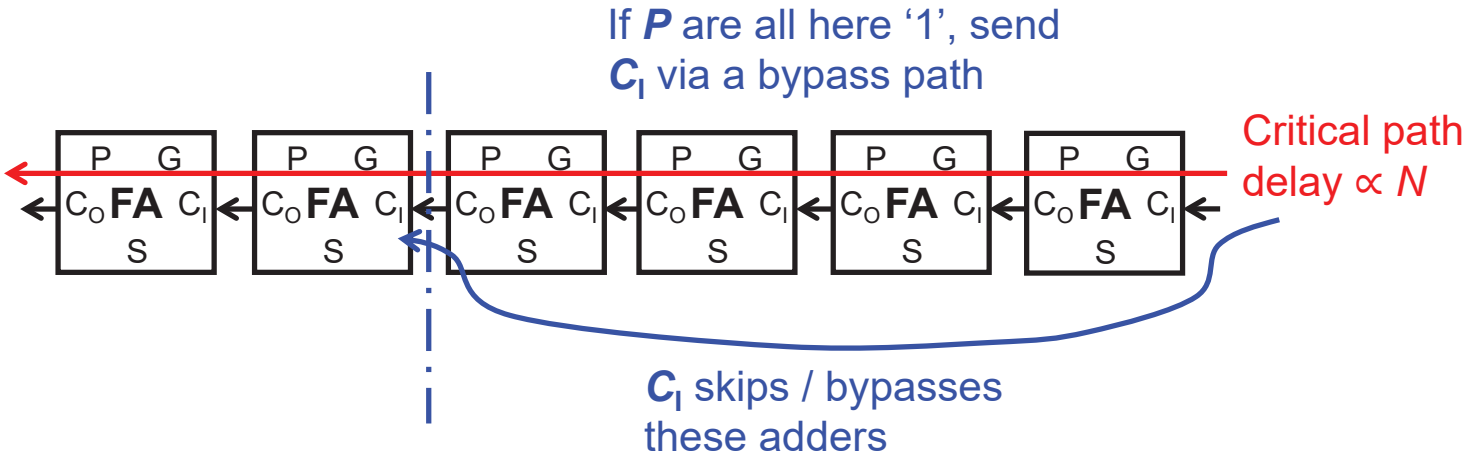
1. Compute ***G*** and ***P*** at each bit position from the corresponding ***A*** and ***B*** bits (Setup)
2. Compute C_I at every bit position (Carry Propagate)
3. Compute ***S*** at every bit position (Sum)



Week 4-6

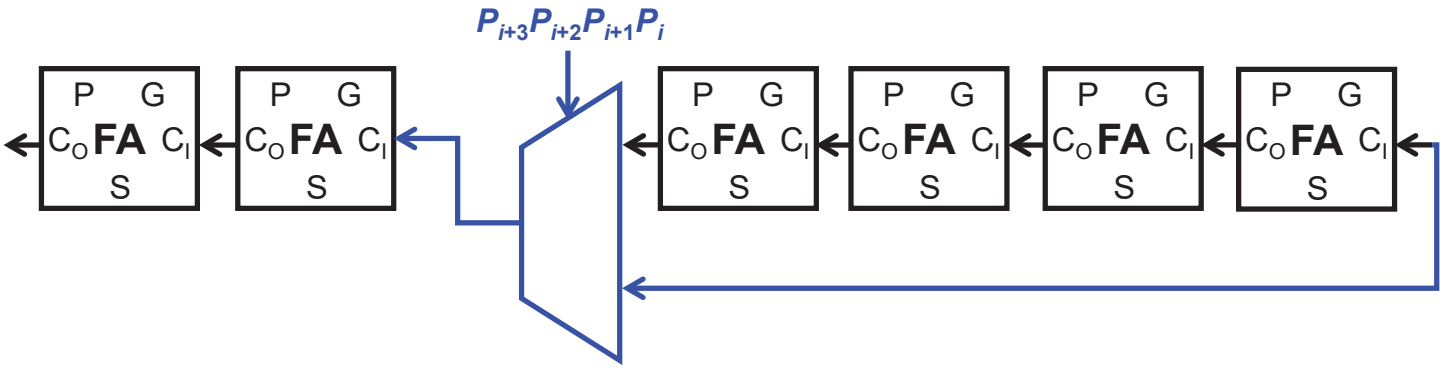
The Carry-Skip (Carry-Bypass) Adder

The Carry-skip / Carry-bypass Adder



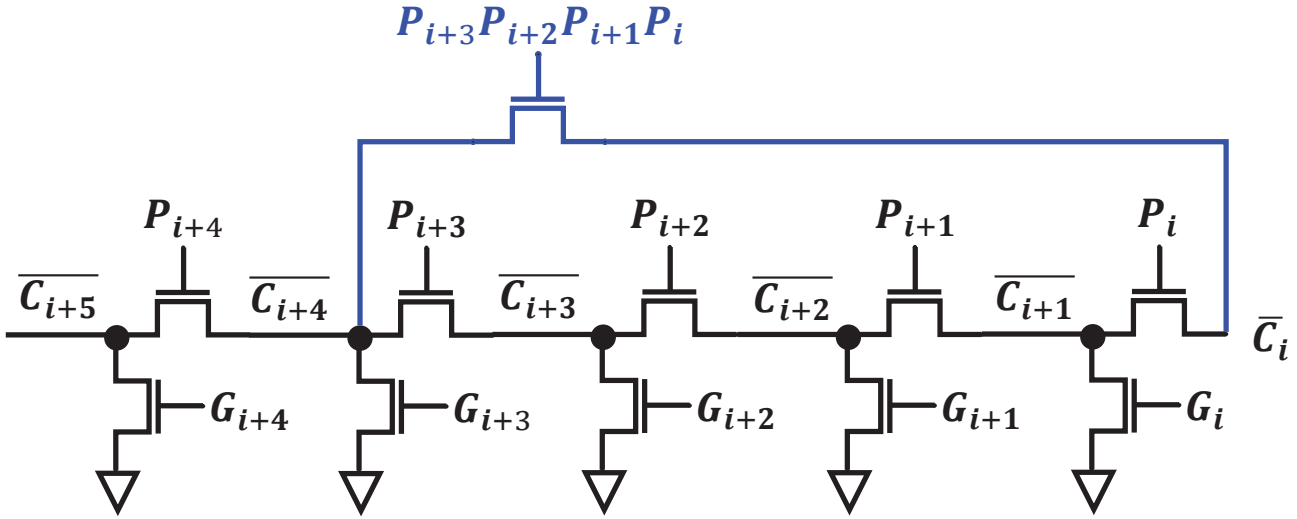
- **Critical path is activated if *P* are all '1'**
- **Since *P* are pre-computed, additional logic can be introduced to forward C_I of the chain to C_I of an intermediate FA**

The Carry-skip / Carry-bypass Adder



- **Critical path is activated if *P* are all '1'**
- **Since *P* are pre-computed, additional logic can be introduced to forward C_I of the chain to C_I of an intermediate FA**
- **Need bypass multiplexer**

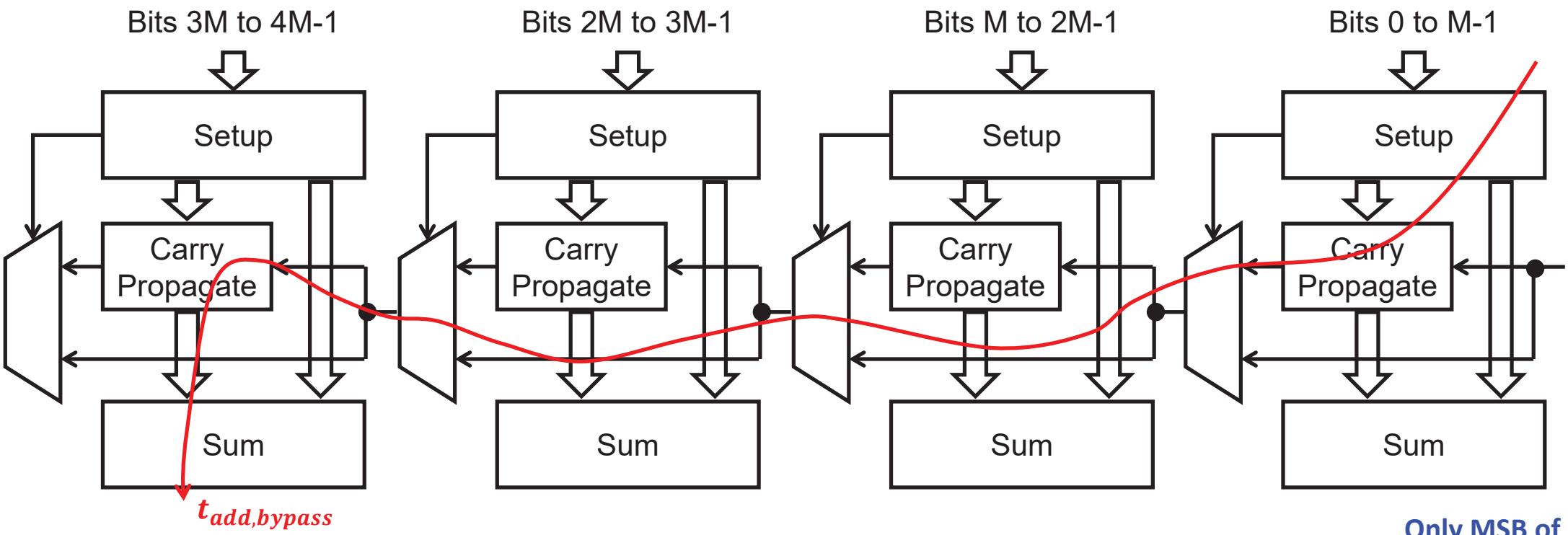
Bypass in The Manchester Carry Chain



C_i: Carry-in of bit position i

- Add only a single bypass transistor

Critical Path Delay (*M*-bits per Stage, Total *N*-bits)

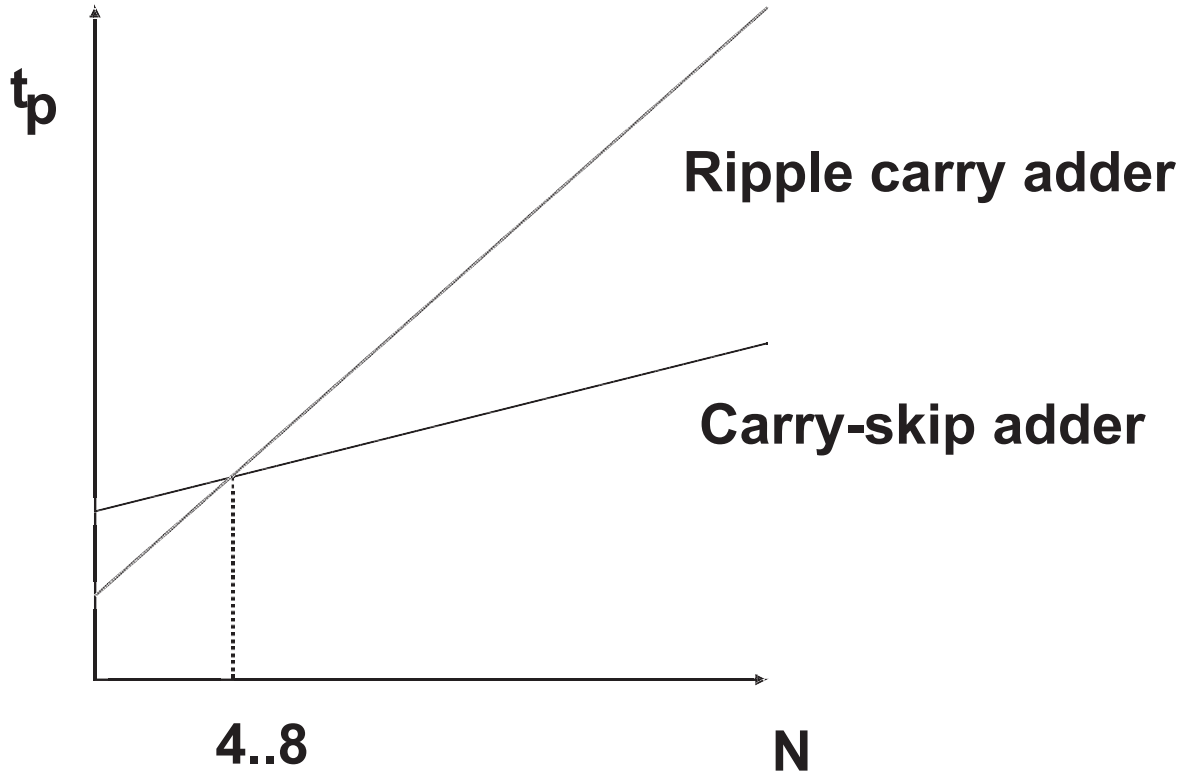


t_{setup}: delay of setup
t_{carry}: delay of 1-bit of carry
t_{mux}: delay of multiplexer (mux)
t_{sum}: delay of sum

Only MSB of final stage

$$t_{add,bypass} = t_{setup} + Mt_{carry} + (N/M)t_{mux} + (M-1)t_{carry} + t_{sum}$$

Delay Comparison (Carry-skip vs Ripple Carry)

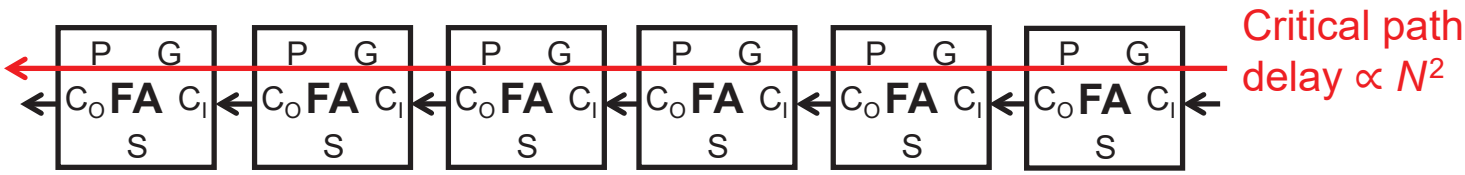


The carry-skip adder will be faster than the ripple carry of ***N*** larger than 4~8 bits.

Week 4-7

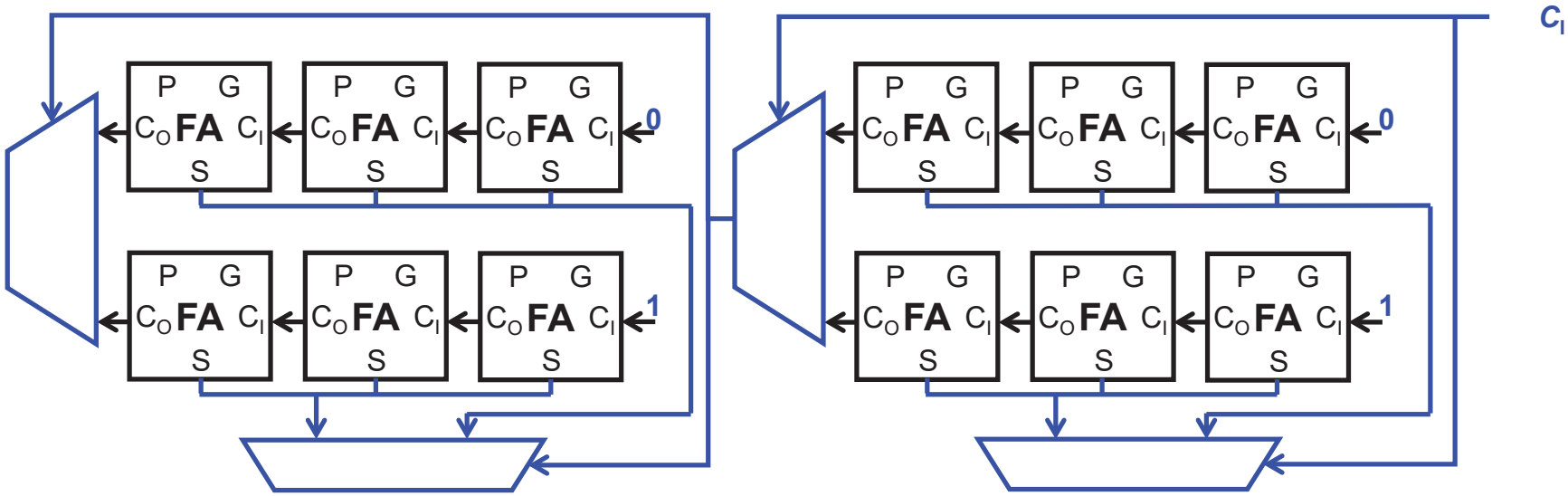
The Carry-Select Adder

The Carry-select Adder



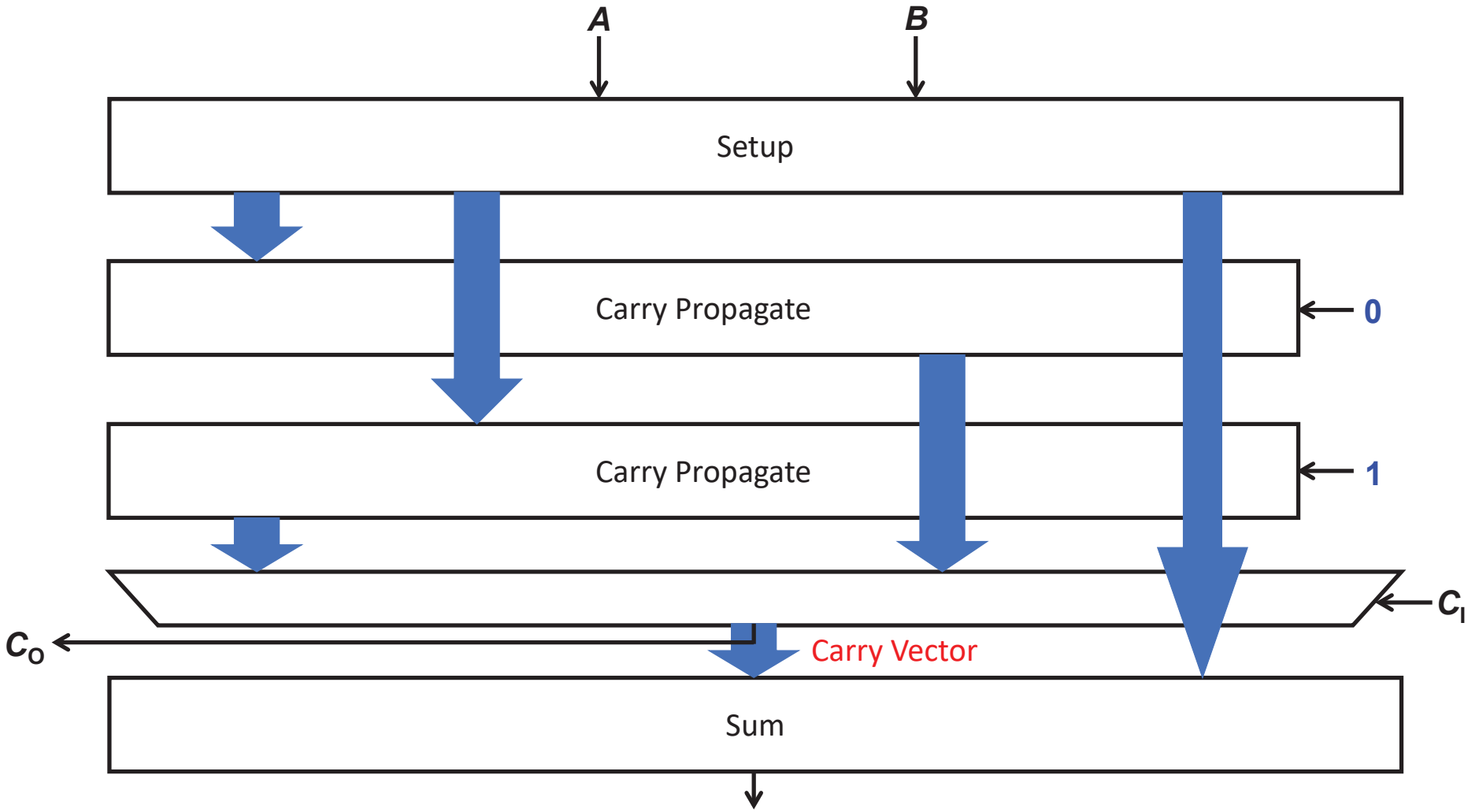
- **Critical path is activated if *P* are all '1'**
- **The intermediate FAs need to wait for the correct C_I to arrive to return the correct result**
- **Compute results for both possibility of C_I**
- **Select the correct result to output once C_I is known**

The Carry-select Adder

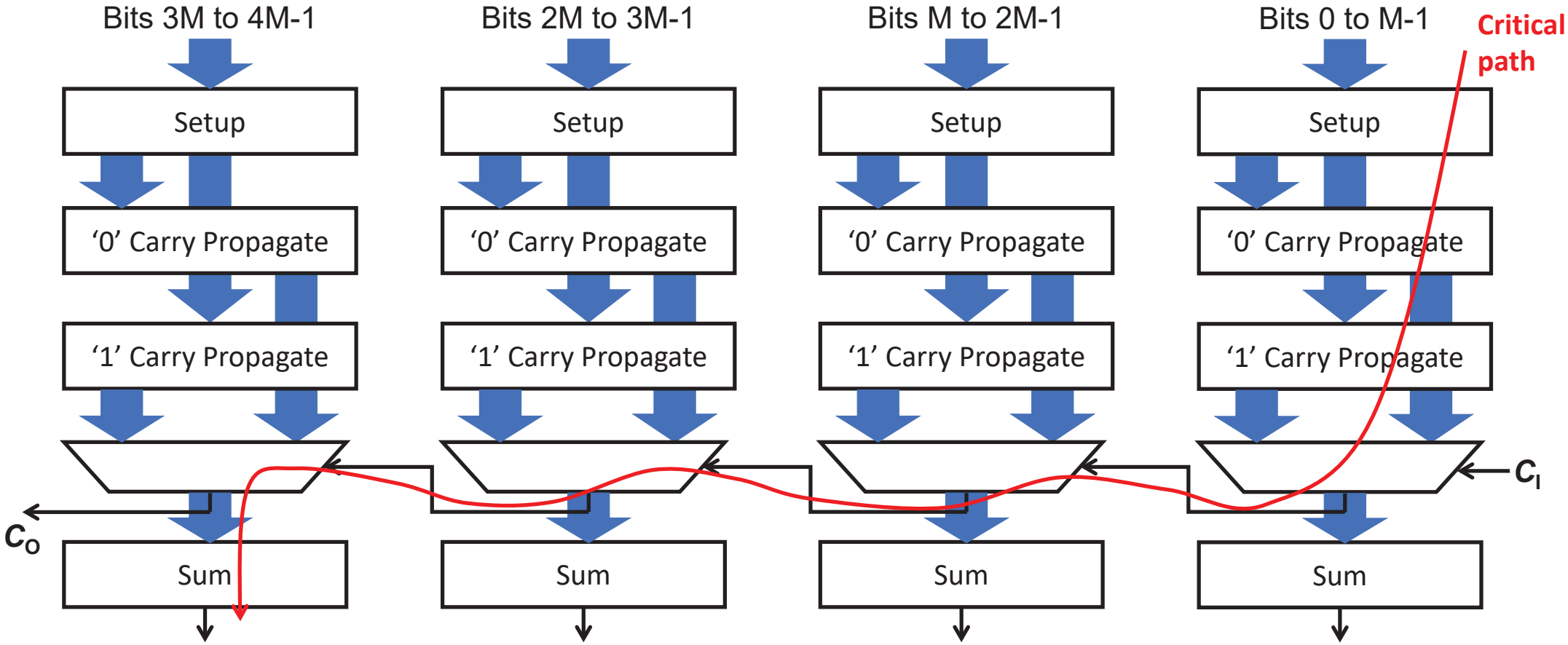


- **Critical path is activated if *P* are all '1'**
- **The intermediate FAs need to wait for the correct C_I to arrive to return the correct result**
- **Compute results for both possibility of C_I**
- **Select the correct result to output once C_I is known**

The Carry-select Adder without Duplicating *Sum*



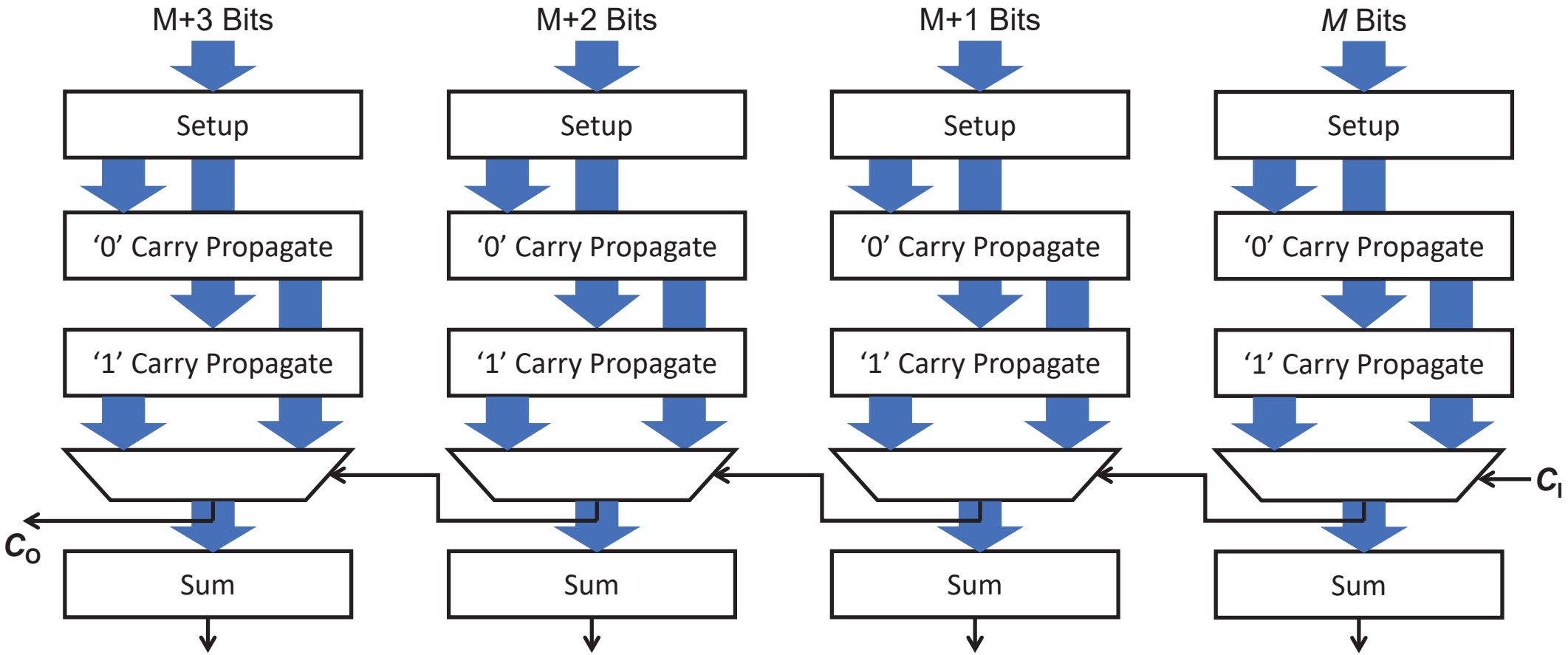
Linear Carry-select Adder (*M*-bits/stage, Total *N*-bits)



t_{setup}: delay of setup
t_{carry}: delay of 1-bit of carry
t_{mux}: delay of multiplexer (mux)
t_{sum}: delay of sum

$$t_{add,carry-select} = t_{setup} + Mt_{carry} + (N/M)t_{mux} + t_{sum}$$

Square Root Carry-select Adder (*P* stages, Total *N*-bits)



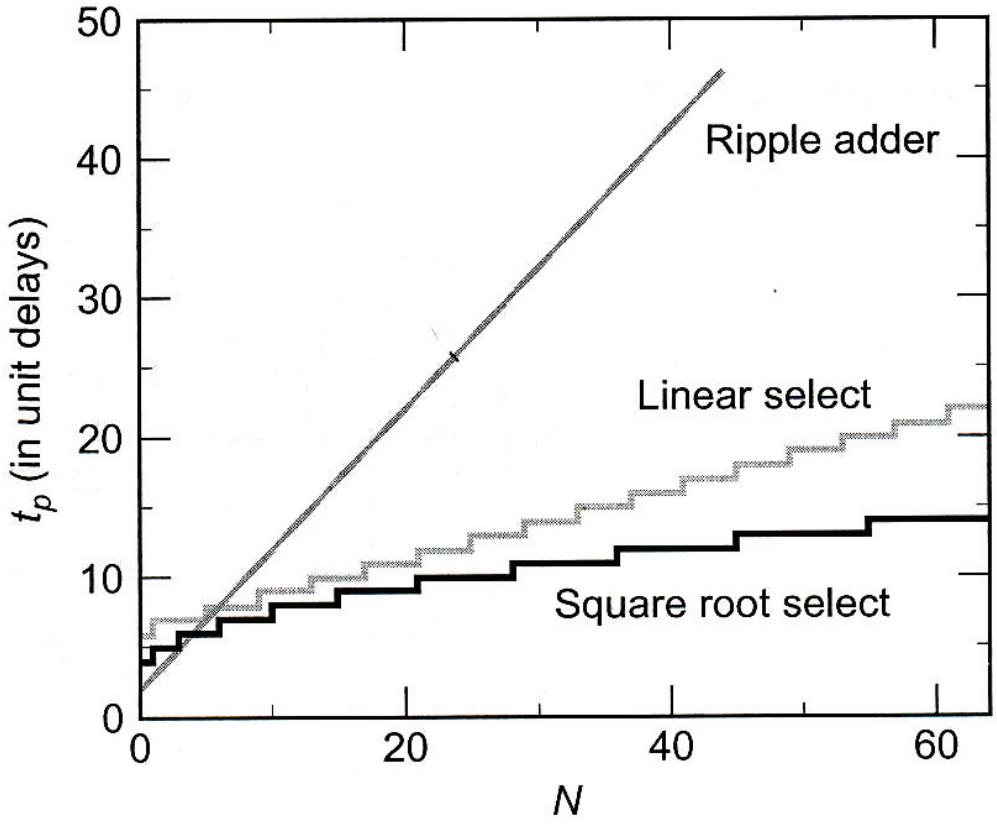
$$N = M + (M+1) + (M+2) + \cdots + (M+P-1) = MP + 0.5P(P-1) \approx 0.5P^2$$

In linear carry-select adder, (N/M) is the number of stages

$$t_{add} = t_{setup} + Mt_{carry} + \left(\sqrt{2N}\right)t_{mux} + t_{sum}$$

Why is the delay of carry propagation in the first stage (and not other stages) here?

Delay Comparison (Carry-select vs Ripple Carry)



The carry-select adder is more advantageous as ***N*** becomes very large

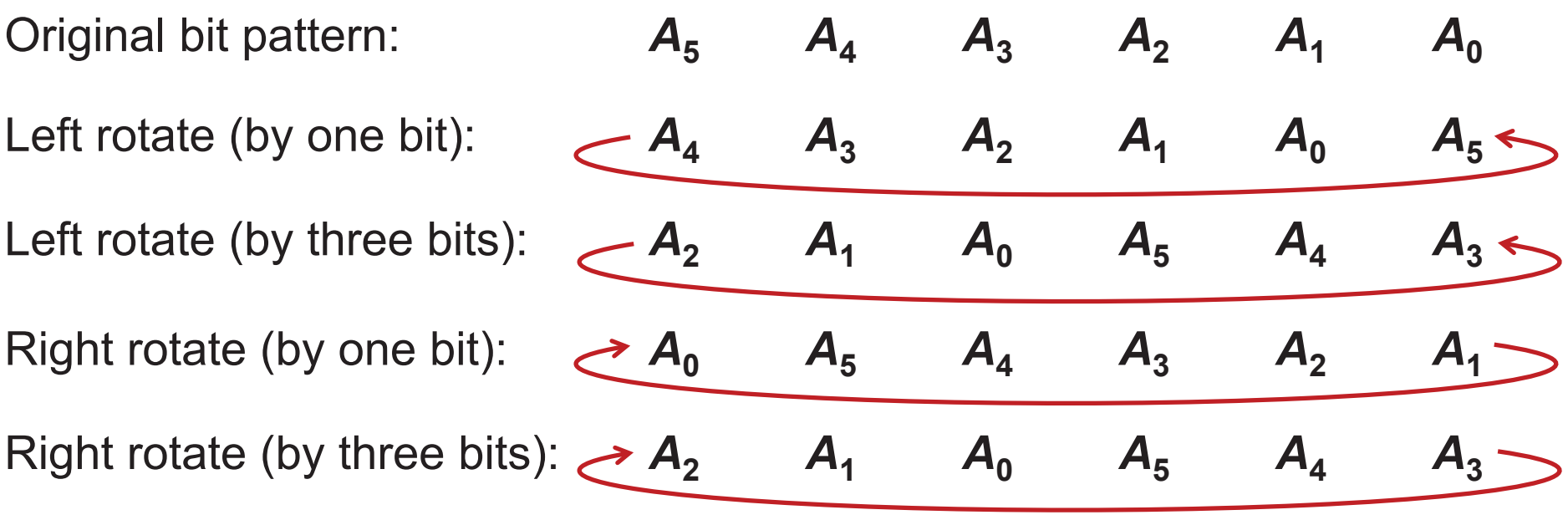
Week 4-8

Shifters

Bit Shifting Operations

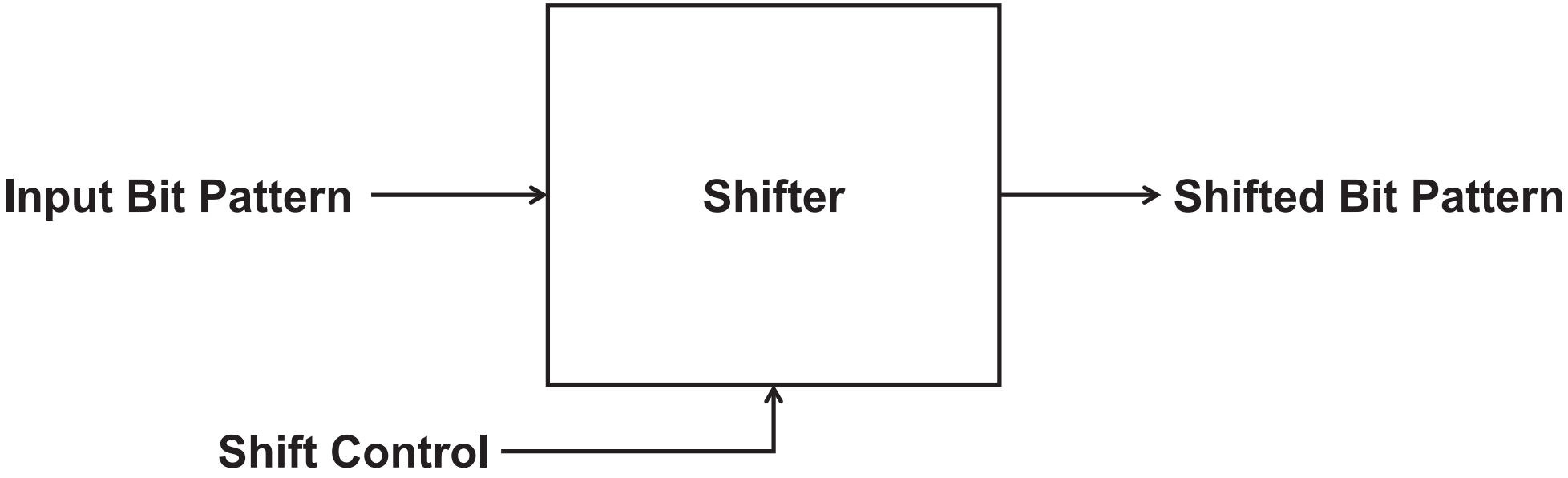
Original bit pattern:	A_5	A_4	A_3	A_2	A_1	A_0
Left shift (by one bit):	A_4	A_3	A_2	A_1	A_0	0
Left shift (by three bits):	A_2	A_1	A_0	0	0	0
			Logical shifts			
Right shift (by one bit):	0	A_5	A_4	A_3	A_2	A_1
Right shift (by three bits):	0	0	0	A_5	A_4	A_3
			Arithmetic shifts			
Right shift (by one bit):	A_5	A_5	A_4	A_3	A_2	A_1
Right shift (by three bits):	A_5	A_5	A_5	A_5	A_4	A_3

Bit Shifting Operations - Rotations



Bit rotations (or *circular shifts*) are like bit shift operations. Bits are not dropped when they get shifted past the extreme ends but are instead shifted to the opposite extreme end in a wrapped around manner.

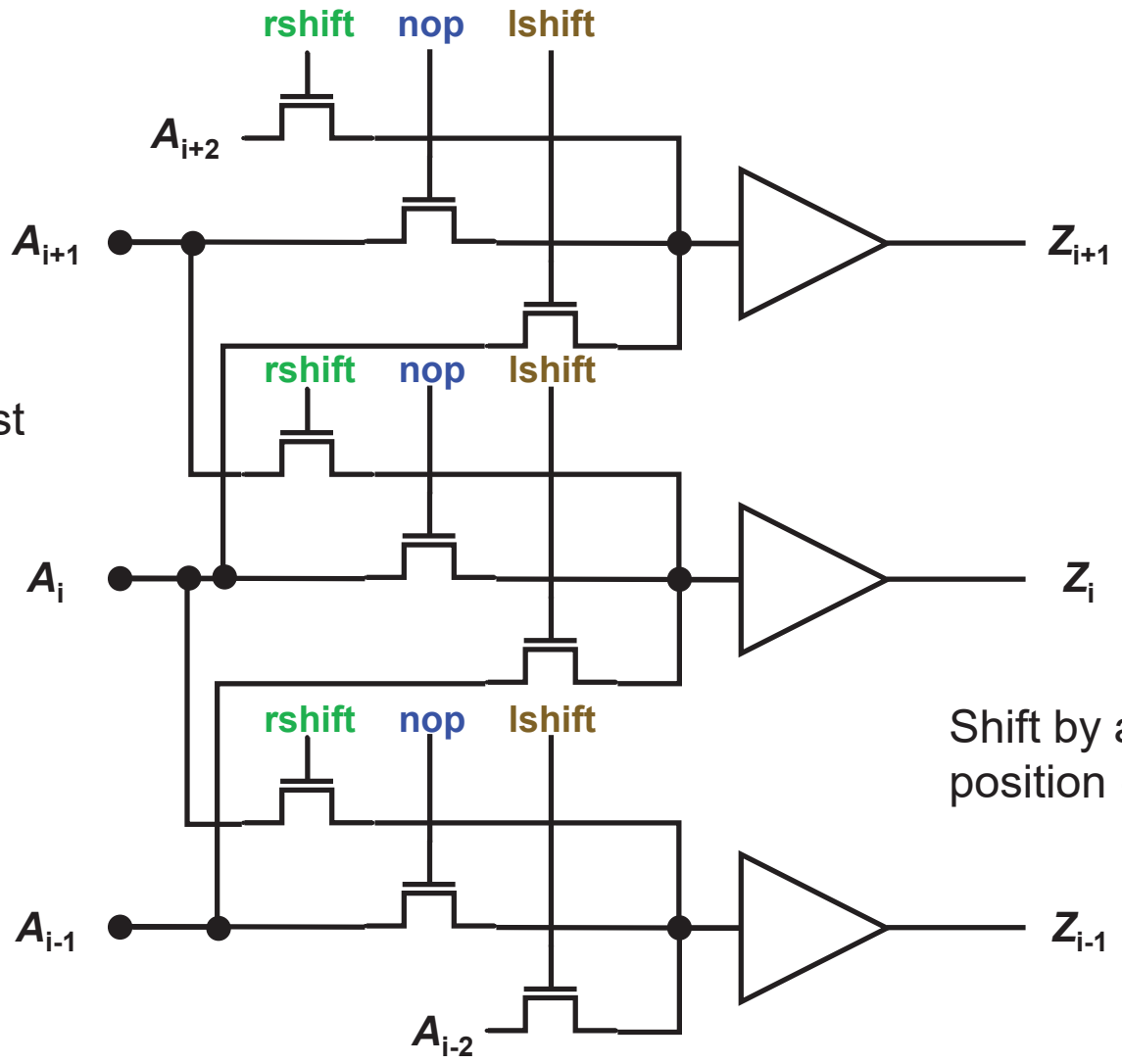
Shifter Circuits



The shifter circuitry is usually a specially designed multiplexer

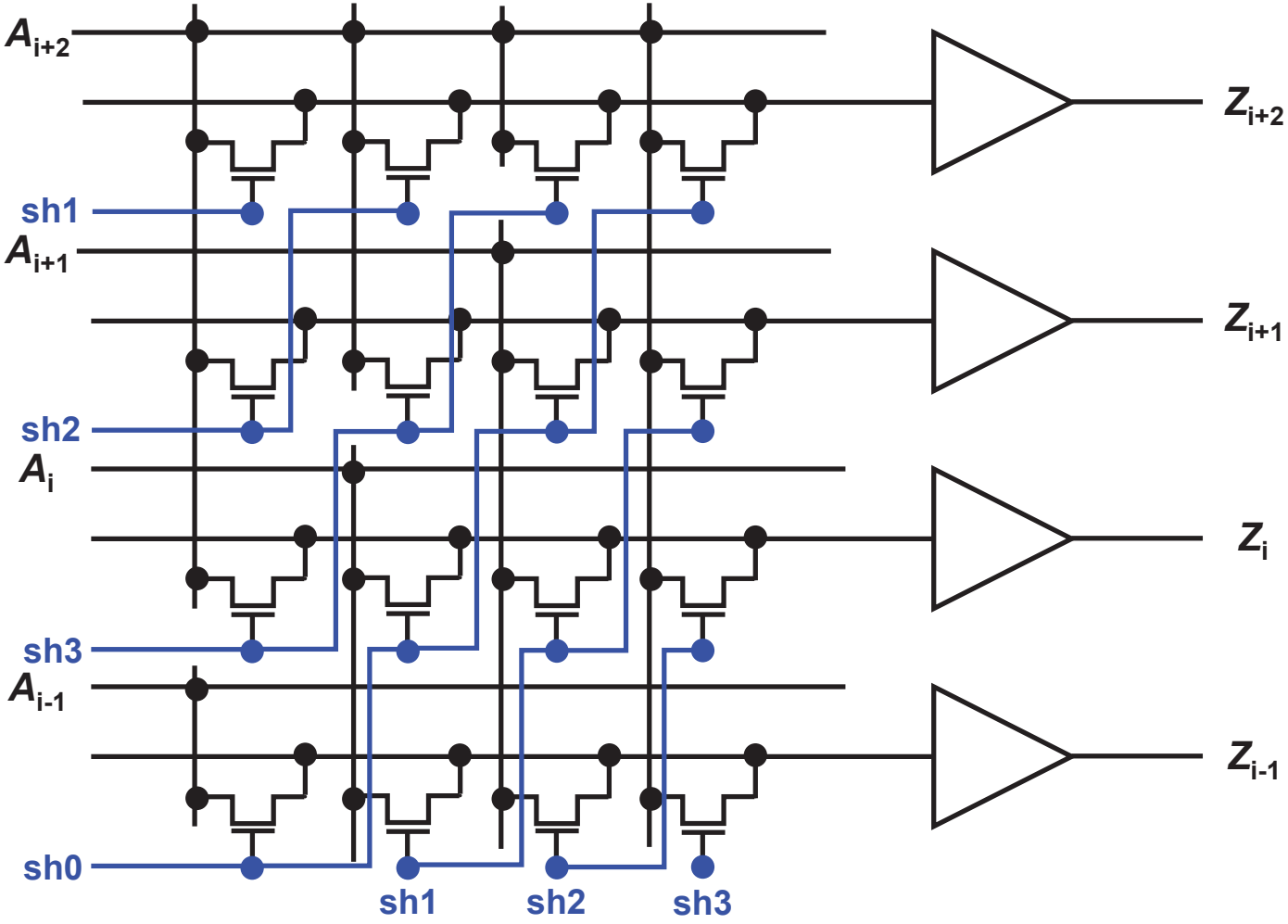
The Binary Shifter

Only one of rshift, nop or lshift can be '1' and the rest must be '0' (one-hot code)



Shift by at most one bit position (limited capability)

The Barrel Shifter

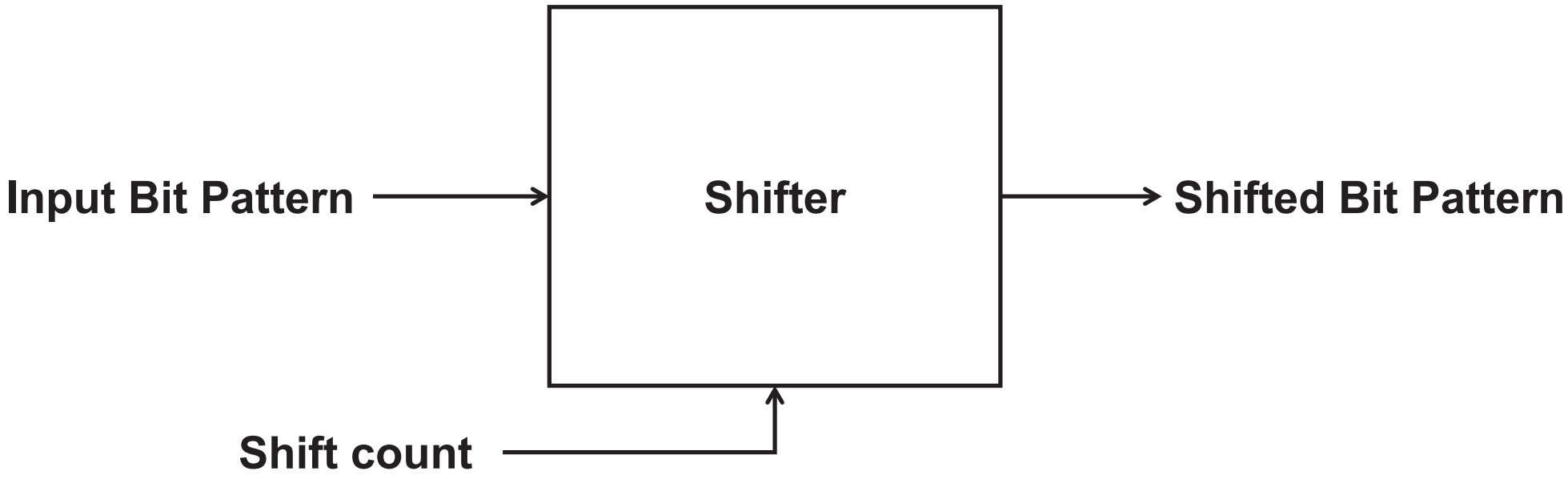


Only one of sh***X*** can be ‘1’ and the rest must be ‘0’ (one-hot code)

Switch matrix / crossbar size = $N \times M$ to support different shifts

Area dominated by complex wiring

The Logarithmic Shifter

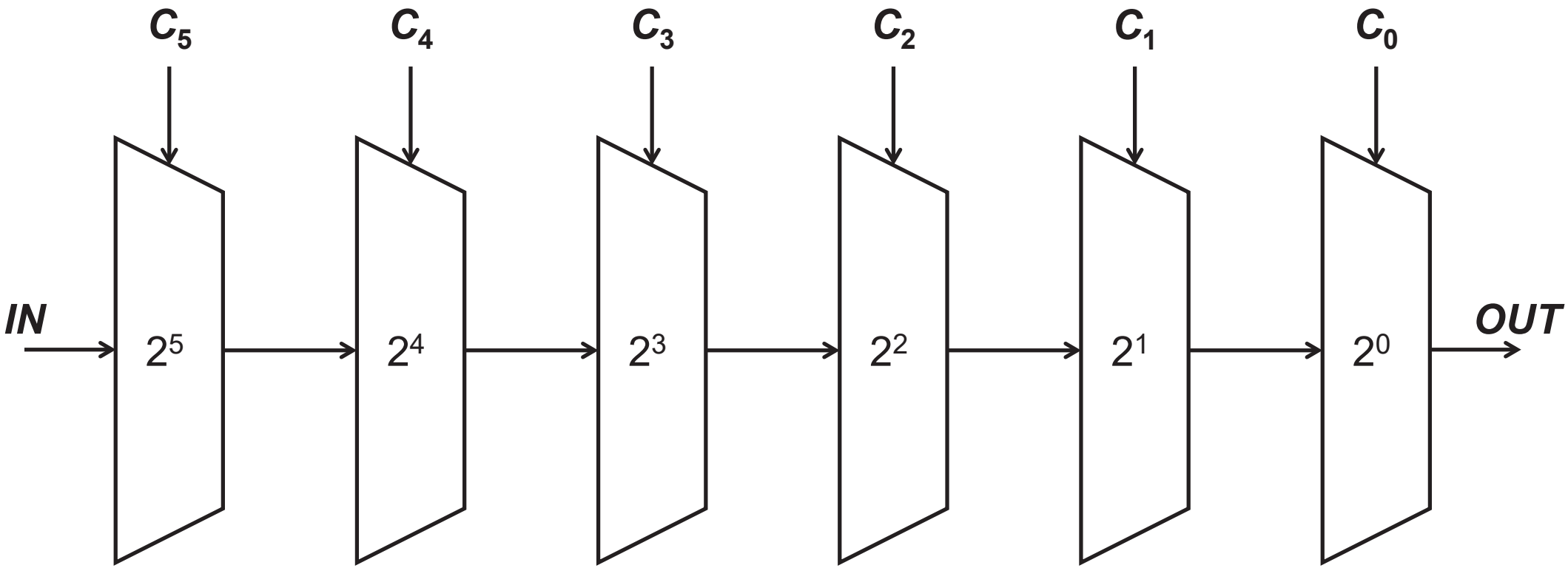


One-hot code wastes resources (e.g., I/O pins, wiring)

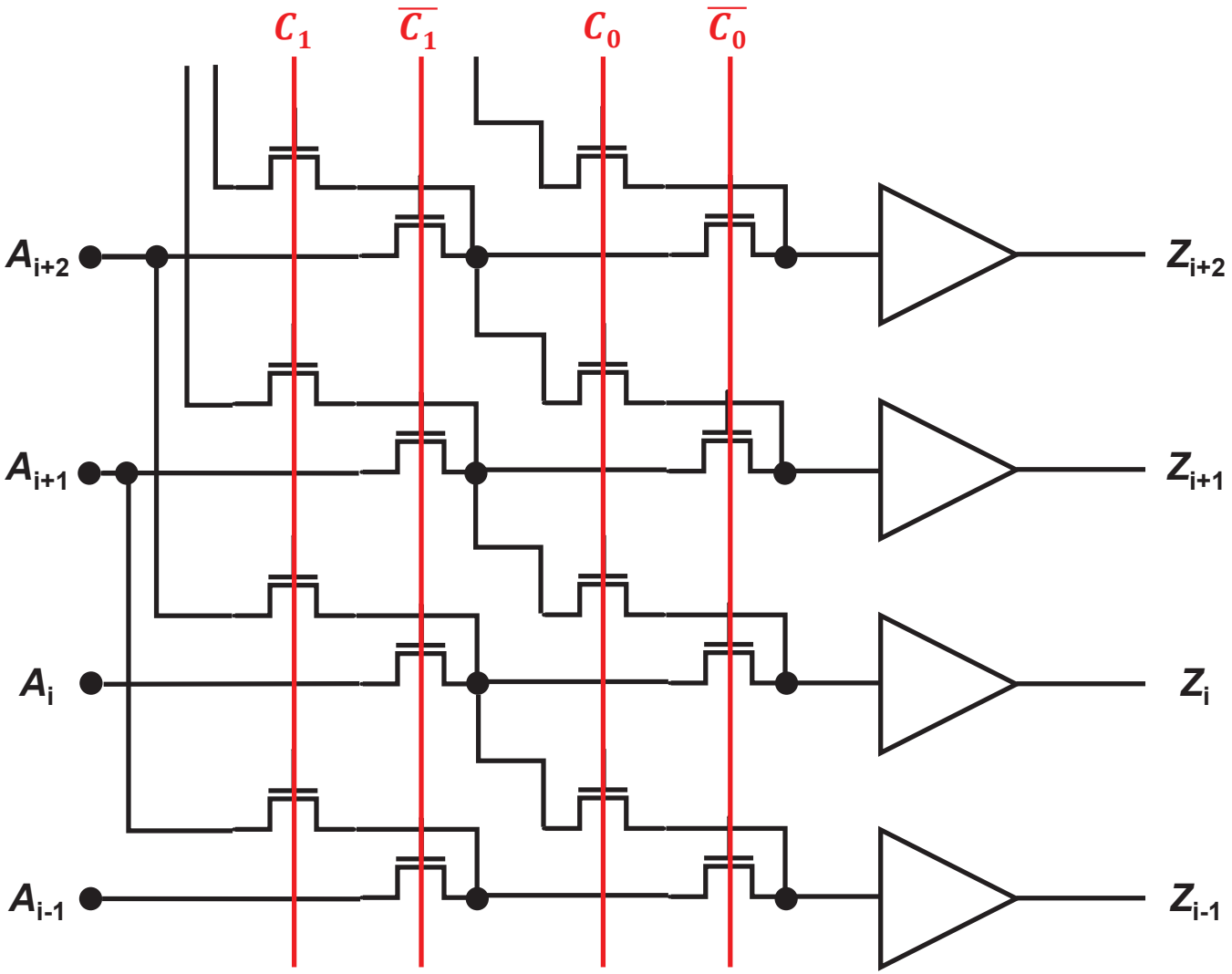
Optimized if shift count corresponds to the number of bits to shift by

Shift count:	C_5	C_4	C_3	C_2	C_1	C_0
Shift by:	2^5	2^4	2^3	2^2	2^1	2^0

The Logarithmic Shifter



The Logarithmic Shifter



Complementary signals for all control bits are needed

Reduced wiring complexity and area overhead