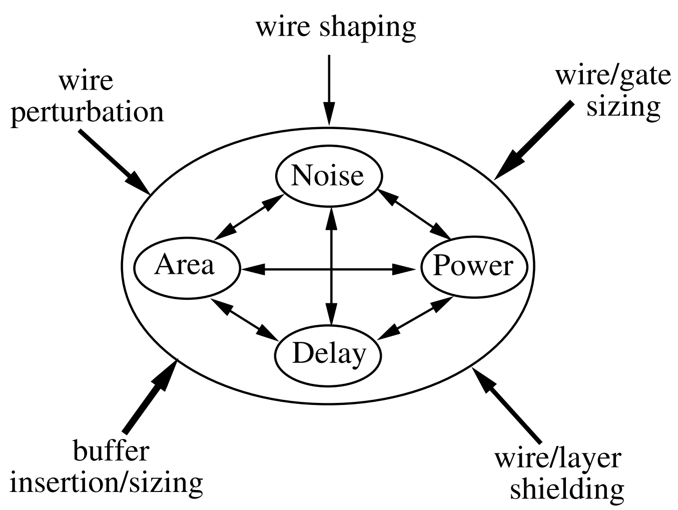
Chapter 6: Subsystem Design

- **Reading: Wolf**
  - Chapter 6.1–6.2: Subsystem design principles
  - Chapter 6.3: Combinational shifters
  - Chapter 6.4: Adders
  - Chapter 6.5: ALUs
  - Chapter 6.6: Multipliers
  - Chapter 6.7: High-density memory
  - Chapter 6.8–6.9: Field-programmable gate arrays

Subsystem Design

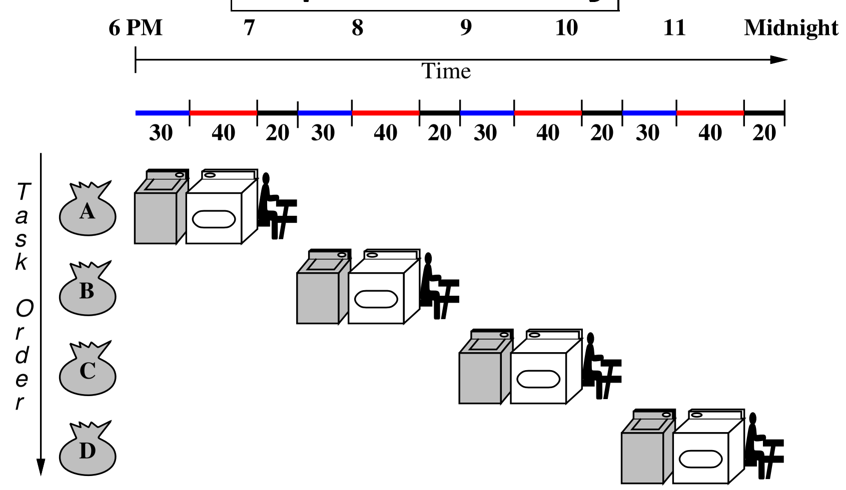
- Major contemporary design metrics: area, delay, power, noise
- Popular techniques: microscopic changes, sizing, buffer insertion, logic redesign, memory element placement, etc.



Optimization at Each Level of Abstraction

- **Layout**
  - Change microscopic layout: parasitics? gate/wire placement? wire shapes? wire shielding?
- **Circuit**
  - Size transistors: delay? area? power? noise?
  - Use precharged gates: delay? noise? power? area?
- **Logic**
  - Redesign logic: gate depth? area?
- **Register-transfer and above**
  - Place memory elements: clock period?
  - Encode signals/states: logic delay? area?
  - Pipeline circuits: tradeoffs between clock period and latency

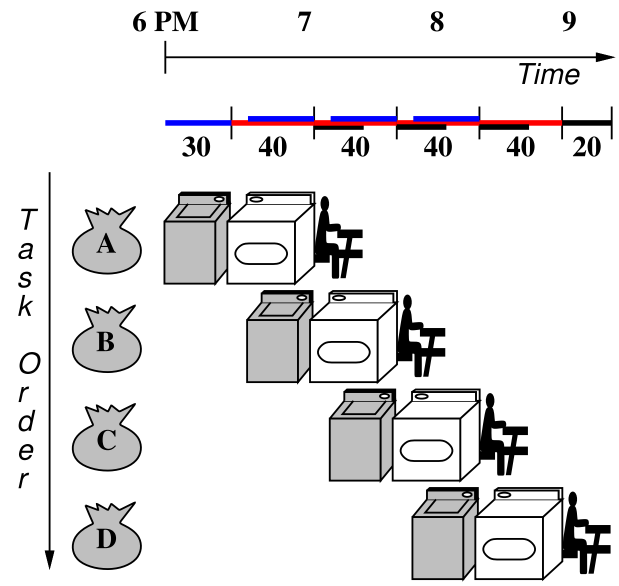
Sequential Laundry



- Washer takes 30 min; Dryer takes 40 min; “Folder” takes 20 min.
- Sequential laundry takes 6 hours for 4 loads.
- If they learned pipelining, how long would laundry take?

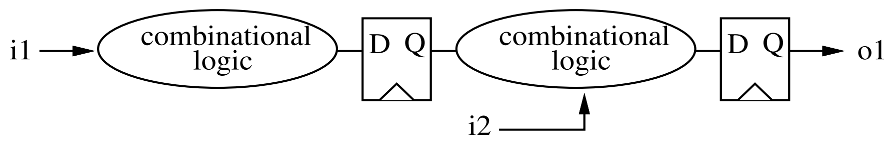
Pipelined Laundry & Lessons

- Pipelining doesn't help latency of single task. It helps throughput of entire workload.
- Pipeline rate limited by the slowest pipeline stage.
- Multiple tasks operating simultaneously.
- Potential speedup = # of pipe stages.
- Unbalanced lengths of pipe stages reduces speedup.
- Time to "fill" pipeline and time to "drain" it reduce speedup (data load, store, clock setup, hold, skew, etc).

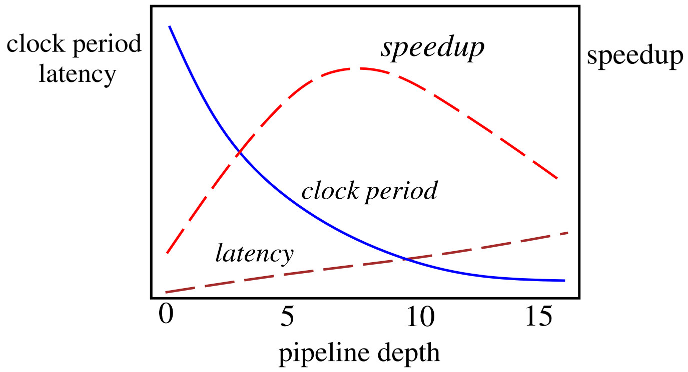


Pipelining Effects

- Pipelining allows large combinational functions to be broken up pieces whose delays are in balance with other system components.

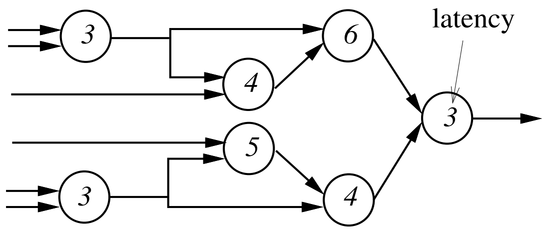


- Pipeline depth v.s. clock period, latency, performance



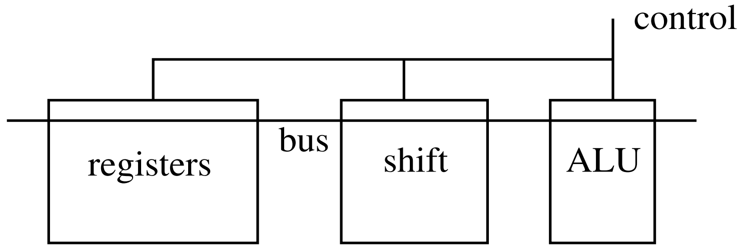
Pipelined vs. Unpipelined Performance

- Assumptions:
  - No delays (including wire delays) except components' latencies.
  - A fixed clock overhead: 2 ns (setup & hold times, clock skew).
- How to partition a circuit to maximize pipelined performance?
- What is the cycle time for the pipelined version of the circuit that maximizes performance without allocating multiple cycles to a stage?
- What is the total time to execute the circuit through all stages?
- What is the speedup versus a single-cycle, unpipelined design?



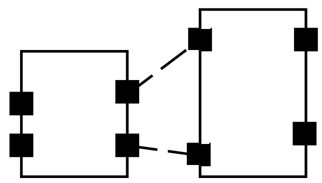
Data Path

- A data path is a logical and a physical structure
  - Bitwise logical organization
  - Bitwise physical design
- Datapath often has ALUs/adders, registers, shifters, counters
- Data is generally passed via buses
- Bit-slice is a one-bit version of the complete datapath; n-bit datapath is composed of n bit-slices connected by buses.



Bit-slice Structure

- Many arithmetic and logical functions can be defined recursively on bits of word.
- A bit-slice is a one-bit (or n-bit) segment of an operation of minimum size to ensure regularity.
- Regular logical structure allows regular physical structure.
- Cells in bit-slice may be abutted together, which requires matching positions on terminals.
- Pitch-matching is designing cells to ensure that pins are proper positions for abutting.



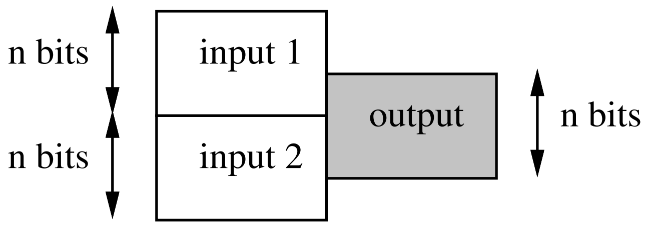
abutting cells may require
moving pins or stretching

Registers and Buses for Datapaths

- Registers and buses are unique to datapath design.
- Registers
  - If only a few registers are needed, use standard latches or flip-flops.
  - If many registers are required, choose an n-port SRAM to save area.
- Buses
  - Datapath cannot afford full connectivity between all pairs of slices. (routing cost?)
  - Datapaths are usually made with buses that provide fewer connections at much less cost in area.
  - A bus circuit is a specialized multiplexer circuit.

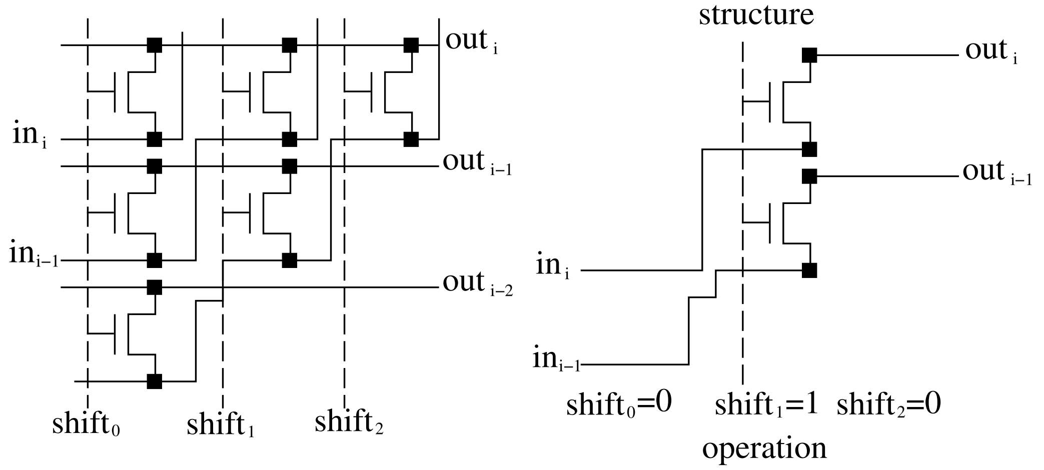
Shifters

- Useful for arithmetic operations, bit field extraction, etc.
- Latch-based shift registers can shift only one bit per clock cycle.
- A **barrel shifter** can perform n-bit shifts in a single cycle with efficient layout.
  - Accepts $2n$ data inputs and n control signals, producing n data outputs.
  - Selects arbitrary contiguous n bits out of $2n$ input bits.
  - Right shift: data into top, 0 into bottom.
  - Left shift: 0 into top, data into bottom.
  - Rotate: data into top and bottom.



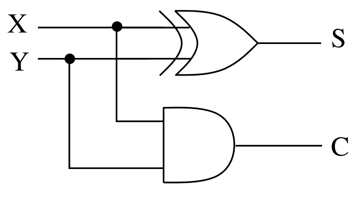
Barrel Shifter Layout

- 2-D array of $2n$ vertical $\times$ n horizontal cells.
- Input data travels diagonally upward; output travels horizontally.
- Control signals run vertically. Exactly one control signal is set to 1, turning on all transmission gates in that column.
- Area: large number of cells, but each one is small.
- Delay: large due to long wires and transmission gates.

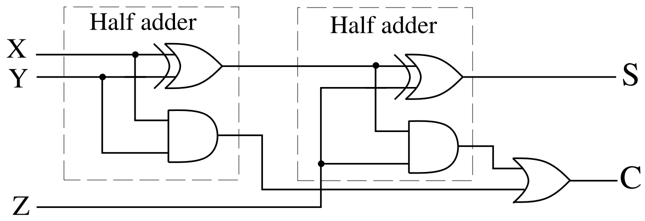


Adder

- Adder delay is dominated by carry chain.
- Carry chain analysis must consider transistor, wiring delay.
- Modern VLSI favors adder designs which have compact carry chains.
- A **half adder** adds two bits, X and Y, to produce a 1-bit **sum**, S, and a 1-bit **carry**, C.
  - $S = (\bar{X}Y + X\bar{Y}) = X \oplus Y$; $C = XY$.
- A **full adder** adds two bits, X and Y, and one **carry input**, Z, to produce a 1-bit **sum**, S, and a 1-bit **carry**, C.
  - $S = (\bar{X}\bar{Y}Z + \bar{X}Y\bar{Z} + X\bar{Y}\bar{Z} + XYZ) = X \oplus Y \oplus Z$;
    $C = XY + XZ + YZ = XY + Z(X \oplus Y)$ (check this!)



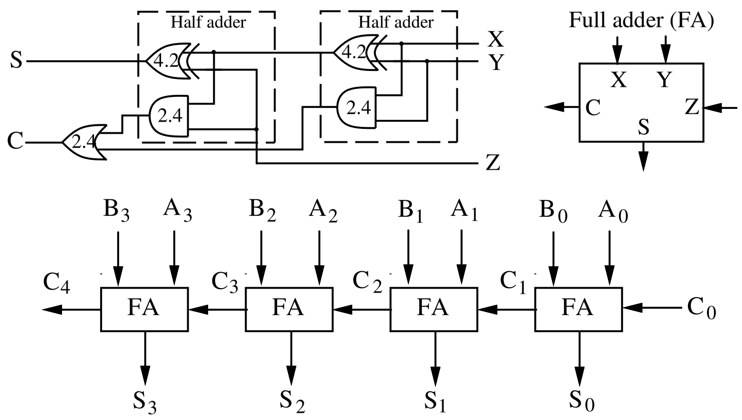
Half adder



Full adder

Ripple Carry Adder

- 4-bit ripple carry adder example: Inputs: C_0, $A = A_{n-1} \ldots A_0$, $B = B_{n-1} \ldots B_0$; Outputs: $s_{n-1} \ldots s_0$ and $c_n \ldots c_1$.



- Propagation delays: $C_0(A_0, B_0) \rightarrow C_1$: 4.8(9.0) ns; $C_0(A_0, B_0) \rightarrow C_2$: 9.6(13.8) ns; $C_0(A_0, B_0) \rightarrow C_3$: 14.4(18.6) ns; $C_0(A_0, B_0) \rightarrow C_4$: 19.2(23.4) ns.
- **Question:** Can we improve the delays?
  - The carries are the bottleneck!

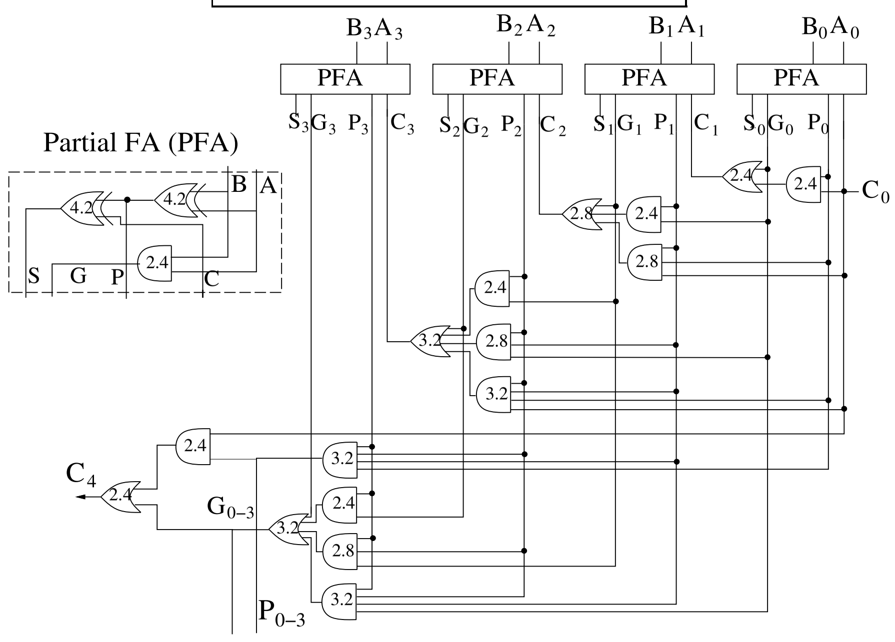
Carry Lookahead Adder

- **Observation:** Major part of the expressions for each carry in the chain can be **precomputed**.
  - Exp: Add $A_3A_2A_1A_0$ and $B_3B_2B_1B_0$.
  - **Carry-generate function:** $G_i = A_iB_i$;
    carry-propagate function: $P_i = A_i \oplus B_i$.
    Then, the 4 carries C_1, C_2, C_3, C_4 can be expressed as follows:

$$\begin{aligned}
C_1 &= G_0 + P_0C_0 && (1)\\
C_2 &= G_1 + P_1C_1 && (2)\\
&= G_1 + P_1G_0 + P_1P_0C_0 \\
C_3 &= G_2 + P_2C_2 && (3)\\
&= G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0 \\
C_4 &= G_3 + P_3C_3 && (4)\\
&= G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0
\end{aligned}$$

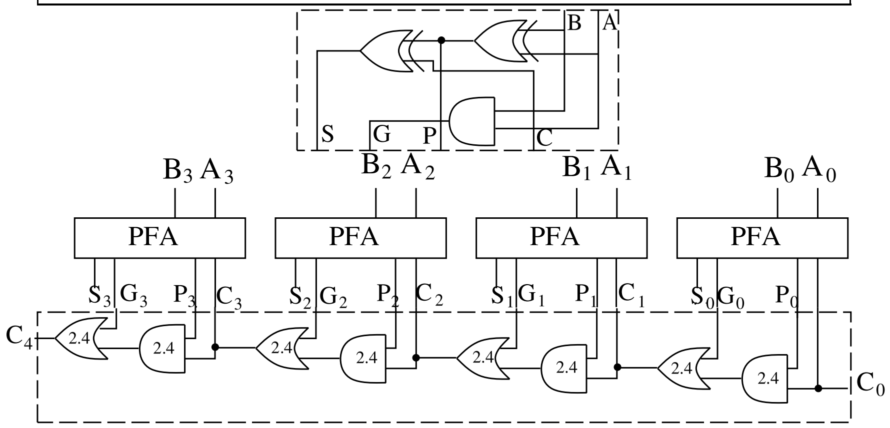
- **Carries can be computed directly from input bits A_i, B_i and input carry C_0.** No ripple effect!
- Extend to n-bit adders?
  - **Problem: Fan-in limitation?**

Carry Lookahead Adder



- **Group propagate function:** $P_{0-3} = P_3P_2P_1P_0$;
  Group generate function: $G_{0-3} = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0$;
  $C_4 = G_{0-3} + P_{0-3}C_0$.

Carry Lookahead vs. Ripple Carry Adders

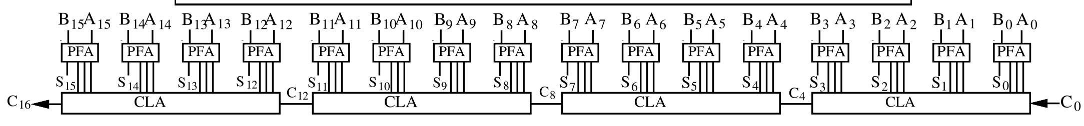


Ripple carry adder

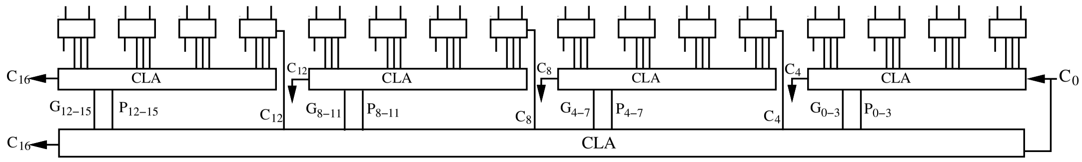
Carry chain	Ripple delay	CLA delay
$C_0(A_0, B_0)$ to C_1	4.8 (9.0)	4.8 (9.0)
$C_0(A_0, B_0)$ to C_2	9.6 (13.8)	5.6 (9.8)
$C_0(A_0, B_0)$ to C_3	14.4 (18.6)	6.4 (10.6)
$C_0(A_0, B_0)$ to C_4	19.2 (23.4)	4.8 (13.0)

- CLA is faster, but much more complex layout.

Carry Lookahead Adder Construction



Single–level CLA generator



Two–level CLA generator

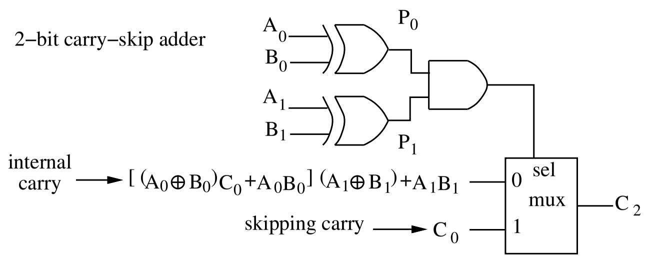
- Carry chain delays for 16-bit adders:

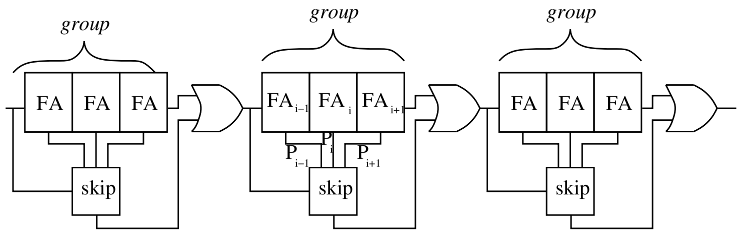
Carry chain	Ripple delay	One-level CLA	Two-level CLA
$C_0(A_0, B_0)$ to C_4	19.2 (23.4)	4.8 (13.0)	4.8 (13.0)
$C_0(A_0, B_0)$ to C_8	38.4 (42.6)	9.6 (17.8)	5.6 (16.2)
$C_0(A_0, B_0)$ to C_{12}	57.6 (61.8)	14.4 (22.6)	6.4 (17.0)
$C_0(A_0, B_0)$ to C_{16}	76.8 (81.0)	19.2 (27.4)	4.8 (19.4)

- n-bit ripple-carry adder has $2n$ gate delays; n-bit adder with $\lceil \log_4 n \rceil$ levels of CLA needs $2\lceil \log_4 n \rceil$ gate delays to generate top-level carries.

Carry-skip Adder

- Looks for cases in which carry out of a set of bits is identical to carry in.
- Typically organized into m-bit stages.
- If $A_i \neq B_i$ for every bit in stage, then bypass gate sends stage's carry input directly to carry output.

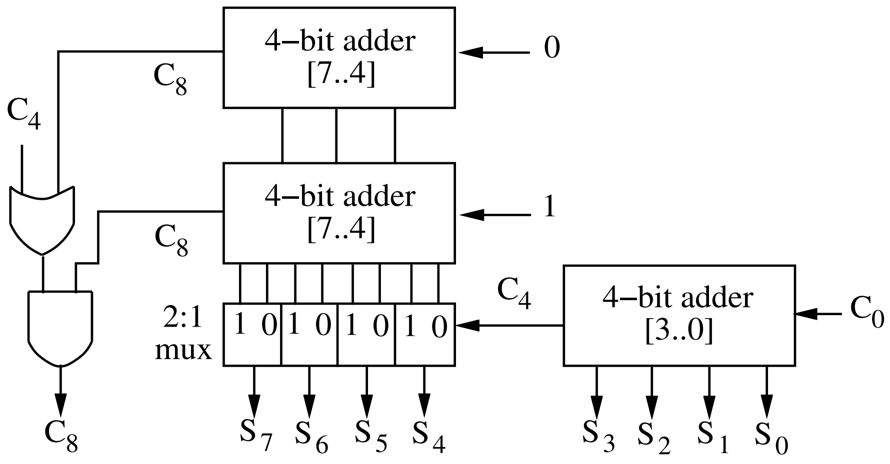




organization of groups in a carry–skip adder

Carry-select Adder

- Compute two results in parallel, each for different carry input assumptions.
- Use actual carry in to select correct result.
- Reduces delay to multiplexer.



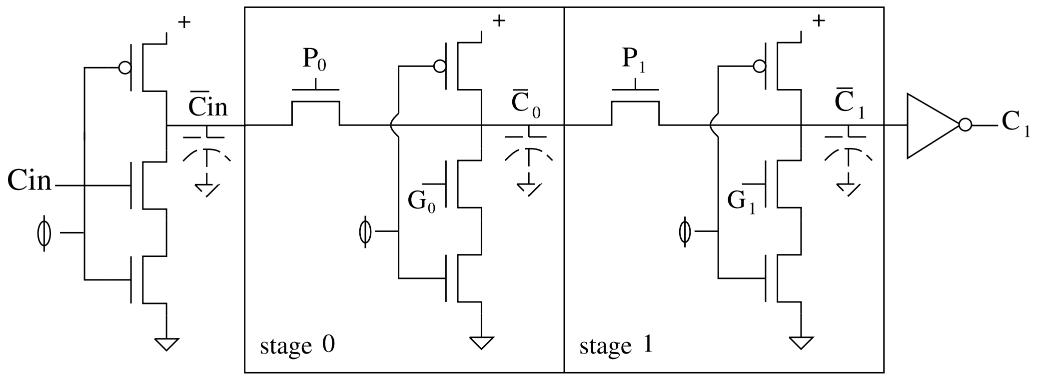
Comparison

- Asymptotic time (speed) and space (area) requirements
- Empirical average power consumption of 16-bit adders

Adder	Time	Space	Power (mW)
Ripple carry adder	$O(n)$	$O(n)$	0.117
Carry look-ahead adder	$O(\lg n)$	$O(n \lg n)$	0.171
Carry skip adder	$O(\sqrt{n})$	$O(n)$	0.109
Carry select adder	$O(\sqrt{n})$	$O(n)$	0.216

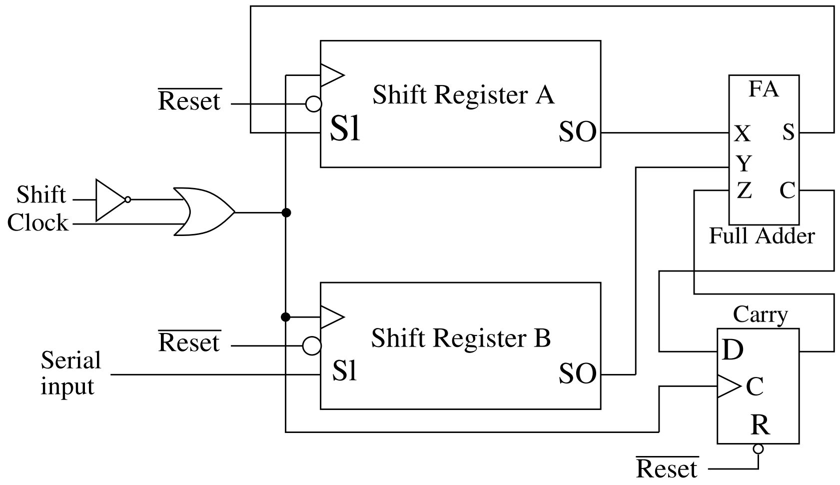
Manchester Adder

- Precharged carry chain uses carry generate and propagate functions $G_i = A_iB_i$ and $P_i = A_i \oplus B_i$.
- Generate signal A_iB_i discharges carry bit.
- Propagate signal $A_i \oplus B_i$ connects adjacent carry.
- Worst-case discharge path goes through entire carry chain.



Serial Adder

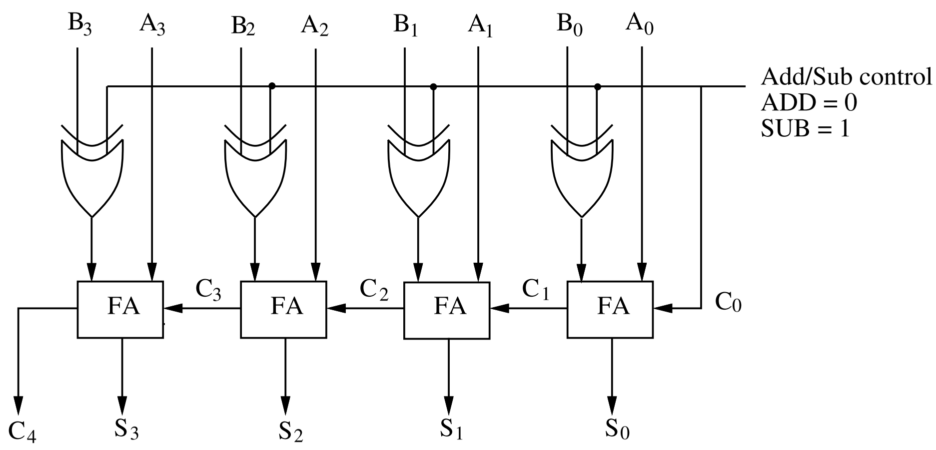
- Add two numbers (held in registers A and B) serially, starting at the least significant bit.
- The serial adder is slower but smaller than the parallel (e.g., ripple-carry) adder.
- May be used in signal-processing arithmetic where fast computation is important but latency is unimportant.



Binary Adder/Subtractor

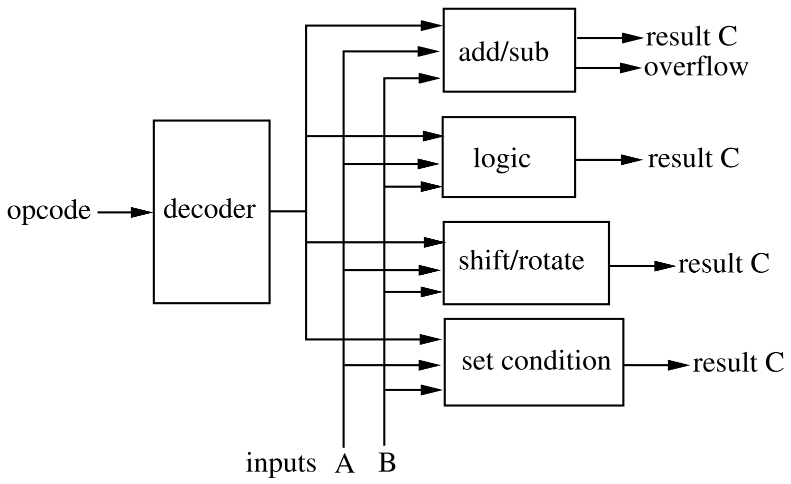
- Subtraction can be achieved by adding 2's complement.
  - Complement each bit (1's complement) and add 1 to the result.
- $C_0 = 0$: addition; $C_0 = 1$: Form the 1's complement of B using EXOR gates and add one.
- Inputs $A[0..n-1], B[0..n-1]$, Sum $S[0..n-1]$, sign bit: $(n-1)$-th

$$Overflow \ = \ A_{n-1}B_{n-1}\overline{S_{n-1}} + \overline{A_{n-1}}\ \overline{B_{n-1}}S_{n-1}.$$



ALU

- The arithmetic logic unit (ALU) computes a variety of logical and arithmetic based on opcode.
- May offer complete set of functions of two variables or a subset.
- ALU built around an adder, since carry chain determines delay.



Binary Multiplication

- **Compute partial products**, and **justify and sum** the partial products.
  - $m \times n$ digit multiply generates up to an $(m + n)$-digit result.
- **Compute partial products:**
  - n-digit, base-2 numbers require adding up to n digits (with carries) in a column.
  - Multiply the row of multiplicand digits by each multiplier digit, one at a time.
  - If the multiplier digit is 0 $\Longrightarrow$ all zero; if the multiplier digit is 1 $\Longrightarrow$ the same as the multiplicand.

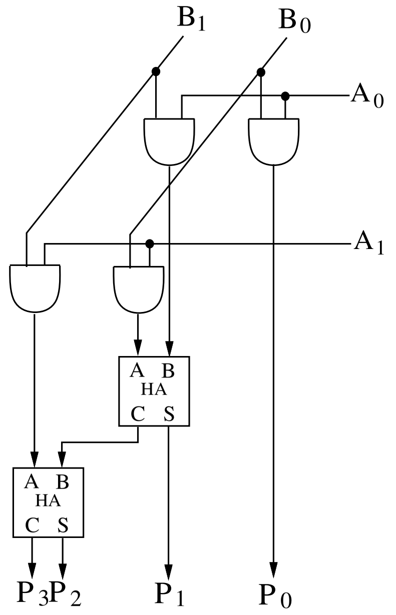
$$
\begin{array}{r}
1\ 0\ 1 \\
\times\ \ 0\ 1\ 1 \\
\hline
1\ 0\ 1 \\
1\ 0\ 1\ \ \ \\
0\ 0\ 0\ \ \ \ \ \ \\
\hline
0\ 0\ 1\ 1\ 1\ 1
\end{array}
$$

2×2 Multiplier

- For $m \times n$ digit multiplier, we need $m \times n$ AND gates and $(n-1)$ m-bit adders to produce a product of $m+n$ bits.

$$
\begin{array}{cccc}
 & & B_1 & B_0 \\
 & & A_1 & A_0 \\
\cline{3-4}
 & & A_0B_1 & A_0B_0 \\
 & A_1B_1 & A_1B_0 & \\
\hline
P_3 & P_2 & P_1 & P_0
\end{array}
$$

$$
\begin{aligned}
P_0 &= A_0B_0 \\
P_1 &= A_0B_1 \oplus A_1B_0 \\
P_2 &= A_1B_1 \oplus (A_0B_1 \bullet A_1B_0) \\
P_3 &= A_1B_1 \bullet (A_0B_1 \bullet A_1B_0)
\end{aligned}
$$



Signed Multiplication

- Case: negative multiplicand, positive multiplier $\Rightarrow$ signed extension works

$$
\begin{array}{r cccccccccc l}
 & & & & & & 1 & 1 & 0 & 1 & 0 & (-6) \\
 & & & & & \times & 0 & 1 & 1 & 0 & 1 & (+13) \\
\hline
\text{sign extension}\left\{\begin{array}{l} \\ \\ \\ \\ \\ \end{array}\right. & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} & 1 & 1 & 0 & 1 & 0 & \\
 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & 0 & 0 & 0 & 0 & 0 & & \\
 & \mathbf{1} & \mathbf{1} & \mathbf{1} & 1 & 1 & 0 & 1 & 0 & & & \\
 & \mathbf{1} & \mathbf{1} & 1 & 1 & 0 & 1 & 0 & & & & \\
 & \mathbf{0} & 0 & 0 & 0 & 0 & 0 & & & & & \\
\hline
 & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{1} & 1 & 0 & 0 & 1 & 0 & (-78)
\end{array}
$$

- Other cases can be handled by applying 2's complement to appropriate numbers.
- How to reduce the number of operations?
- Observation: A multiplier with a single block of 1s can be regarded as the difference of two numbers:

$$
\begin{array}{rccccccc l}
 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & (32) \\
- & 0 & 0 & 0 & 0 & 0 & 1 & 0 & (\ 2) \\
\hline
 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & (30)
\end{array}
$$

Booth Algorithm

- Encoding scheme based on the previous observation to reduce number of stages in multiplication.
- Performs two bits of multiplication at one time $\Rightarrow$ requires half the stages.
- Treat positive and negative numbers uniformly.
- Each stage is slightly more complex than simple multiplier.
- Note that adder/subtracter is almost as small/fast as adder alone.
- Booth encoding:

```
    0  1  0  0  0  0  0
 –  0  0  0  0  0  1  0
 ----------------------
    0  0  1  1  1  1  0  (0)

    0 +1  0  0  0 –1  0
  └───┘ └──┘ └──┘ └──┘
  0     +2    0    –2
```


Booth Algorithm Examples

- Attention: sign extension, 2’s complement, 1-bit left shift

```
          1 1 0 1 0   (–6)
        × 0 1 1 0 1   (+13)
---------------------
1 1 1 1 1|1 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 0 1 0
0|0 0 0 0 0
---------------------
1 1 1 0 1 1 0 0 1 0   (–78)
```

```
          1 1 0 1 0
        +1 0 –1 +1 –1
---------------------
0 0 0 0 0|0 0 1 1 0
1 1 1 1|1 1 0 1 0
0 0 0|0 0 1 1 0
0 0|0 0 0 0 0
1|1 1 0 1 0
---------------------
1 1 1 0 1 1 0 0 1 0
```

```
          1 1 0 1 0
         +1   –1   +1
---------------------
1 1 1 1 1|1 1 0 1 0
0 0 0|0 0 1 1 0  ←—— 2’s complement
1|1 1 0 1 0
---------------------
1 1 1 0 1 1 0 0 1 0
```

```
          0 1 1 0 1   (+13)
        × 1 1 0 1 0   (–6)
---------------------
```

```
          0 1 1 0 1
          0 –1 +1 –1 0
---------------------
0 0 0 0 0|0 0 0 0 0
1 1 1 1|1 0 0 1 1
0 0 0|0 1 1 0 1
1 1|1 0 0 1 1
0|0 0 0 0 0
---------------------
1 1 1 0 1 1 0 0 1 0
```

```
          0 1 1 0 1
          0   –1   –2
---------------------
1 1 1 1 1|0 0 1 1 0  ←—— 2’s complement & shift 1–bit left
1 1 1|1 0 0 1 1  ←—— 2’s complement
0|0 0 0 0 0
---------------------
1 1 1 0 1 1 0 0 1 0
```

Booth Encoding

- 2's complement form of multiplier

$$y = -2^n y_n + 2^{n-1} y_{n-1} + 2^{n-2} y_{n-2} + \dots$$

- Rewrite using $2^n = 2^{n+1} - 2^n$:

$$y = 2^n(y_{n-1} - y_n) + 2^{n-1}(y_{n-2} - y_{n-1}) + 2^{n-2}(y_{n-3} - y_{n-2}) + \dots$$

- Consider the first two terms by looking at three bits of y, we can determine whether to add 0, x, $2x$, $-x$, or $-2x$ to partial product (x: multiplicand).

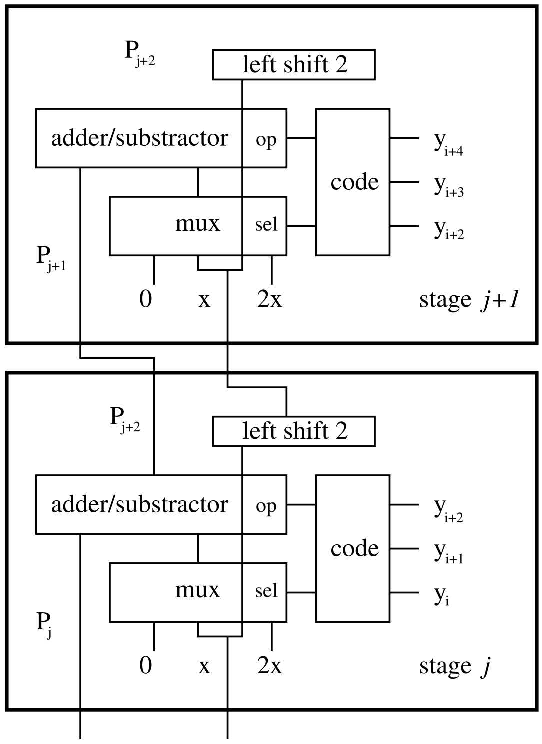
$y_i y_{i-1} y_{i-2}$	Increment
000	0
001	x
010	x
011	$2x$
100	$-2x$
101	$-x$
110	$-x$
111	0

More Booth Example

$y_i y_{i-1} y_{i-2}$	Increment
000	0
001	x
010	x
011	$2x$
100	$-2x$
101	$-x$
110	$-x$
111	0

- $x = 011001\ (25), y = 101110\ (-18)$
- P_i: ith partial product; $P_0 = 00000000000$.
- $y_1 y_0 y_{-1} = 100, P_1 = P_0 - (10 \cdot 011001) = 11111001110$.
- $y_3 y_2 y_1 = 111, P_2 = P_1 + 0 = 11111001110$.
- $y_5 y_4 y_3 = 101, P_3 = P_2 - 0110010000 = 11000111110$ (final product).

Booth Structure



High-Density Memory

- **Memory:** A collection of storage cells together with the necessary circuits to transfer information to and from them.
- **Memory address:** A collection of binary digits interpreted as a number that refers to a basic memory storage element.
- **Memory organization:** The structure of memory in terms of how data are accessed.
  - Organized as an indexed **array of words.** Index $\longleftrightarrow$ Memory address.
  - Address is divided into row, column; row can contain more than one word.