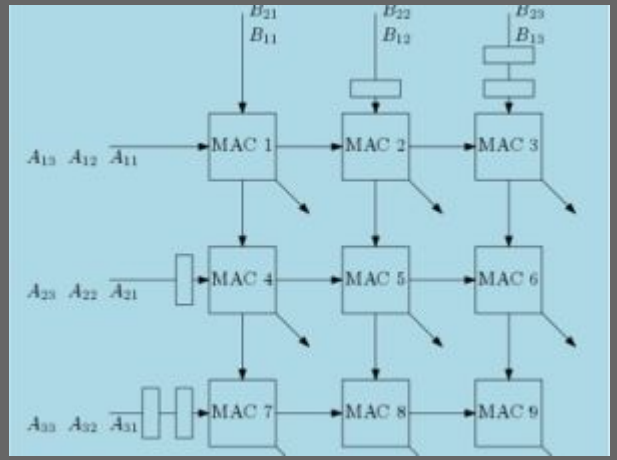
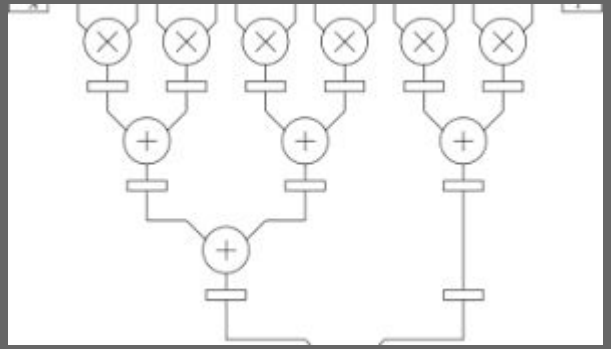
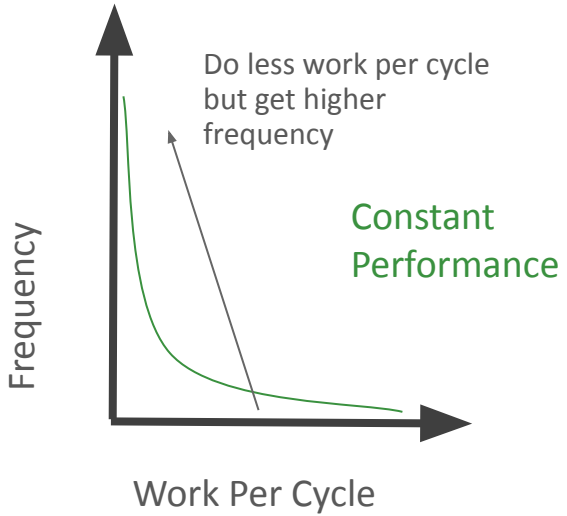
CPE 470 - Accelerator Design





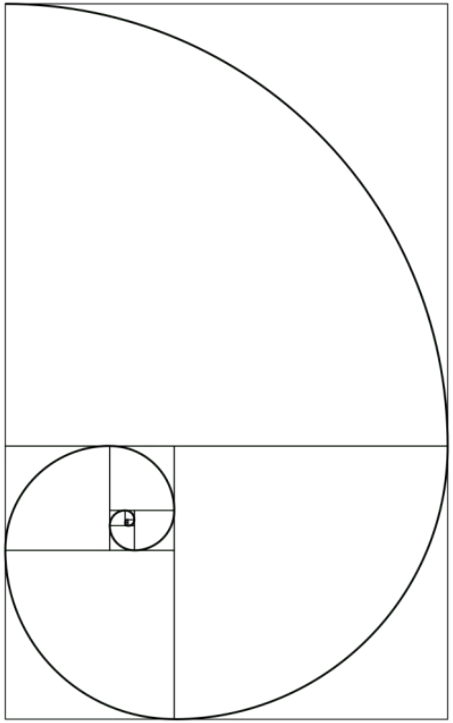
ASIC Design Goals

- Performance = Frequency * Work Per Cycle
- Two opposing constraints to reach higher performance:
  - Maximize Frequency
    - Optimizing critical path often means having to do less per cycle
  - Maximize Work Per cycle
    - Adding logic often increases critical path delay
- Logic Area is usually not main limitation
  - Parallel Logic -
    - Adds work per cycle without increasing critical path
  - Serial Logic - step by step
    - Adds work per cycle at the cost of delay
    - Necessary for Most Problems
    - Consider Splitting across multiple cycles



Fibonacci Accelerator: The Good

- High Frequency
  - Critical path is only one adder deep generally
  - Allows frequencies around 100 Mhz
  - Biggest Calculation
- Solves easy problems quickly
  - Gets low numbered Fibonacci quickly
- Good practice for variable delay systems
  - Have to set up & wait for handshake
  - Results not guaranteed immediately



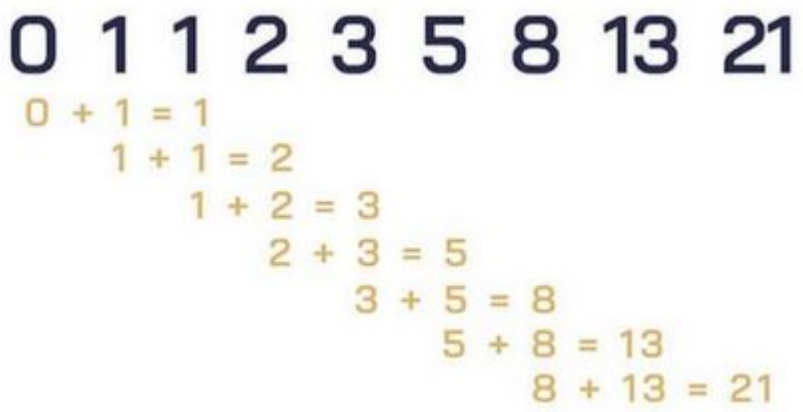
Fibonacci Accelerator: The Bad

- **Data Dependency** Chain
  - Difficult to parallelize this algorithm
  - Each computation step relies on the previous result
- Math solutions are overly complicated
  - Requires floating point, square root, and and division → all super expensive in hardware
- Variable Latency
  - Bigger inputs can take a long time

Glossary

Data Dependency: computation step relies on previous step

$$F_n = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}}$$



Improving Fibonacci: Caching

- Fibonacci involves recomputing the same series
  - Why recompute if we don't have to?
  - Solution ⟶ Save computation states in a cache
- When new input is introduced, start its computation at the closest cached input

Cached Input	Cached Results F_{n-1}, F_n
8	13, 21
15	377, 610
_	_

- Example: New Input of 10
  - Find Closest Cached Input: 8
  - Can resume from 8 and compute for 2 cycles
- Problem ⟶ Cache Lookup
  - Don't want to have to traverse cache
  - Use some bits of input as an address

Input Address Range	Cached Result
0-127	_ : _ , _
128-255	_ : _ , _

- Speeds up average case by # of cache lines, but only once caches are populated

Improving Fibonacci: Look Up Tables

- Similar idea to caching but hard coded in
  - Why wait for cache to fill if we know the result at synthesis time
- Hard code some amount of starting points
  - Example: If input is over 32, start
- Real Life examples could be:
  - sin, cos, trigonometry calcs
  - log, ln, e^x
  - Any math function that is often iteratively approximated

Cached Input	Cached Result
8	21
15	610
22	17711

Improving Fibonacci: Algorithm

- Deceptive Dependency Chain
  - Does every result depend ONLY on the previous result?
  - No! It can also be expressed as a combination of data we already have

- Can calculate 2 fibonacci numbers at a time with no logic depth cost
  - Calculating **c** and **d** both only require one addition each
  - Bit shifts are basically free
    - multiply or divide by a power of 2
  - **2x speedup** in every case!

- Can calculate 4 fib #s at a time with some logic depth cost

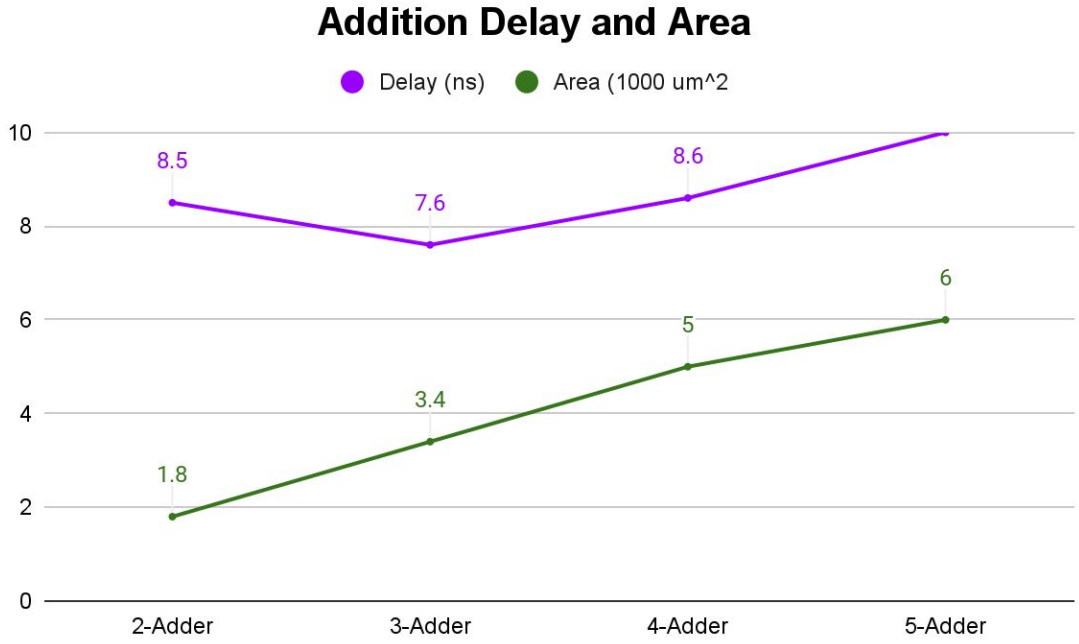
- Take Fibonacci numbers a, b, c, d, e, f
  - **c = b + a**
  - d = c + b = b + a + b ⟶ **d = a + 2b**
    - = a + b<<1
  - e = d + c ⟶ **e = 2a + 3b**
    - = a << 1 + b <<1 + b
  - f = e + d = **f = 3a + 5b**
    - **= a << 1 + a + b << 2 + b**

0 1 1 2 3 5 8 13 21

0 + 1 = 1
1 + 1 = 2
1 + 2 = 3
2 + 3 = 5
3 + 5 = 8
5 + 8 = 13
8 + 13 = 21

Addition

- It turns out adding 3 numbers is not twice as hard as adding 2 number
  - Takes ~twice the area but only a bit more delay
- Delay is dominated by bit width of the add, not by # of operands
  - So calculating 4 Fib #s at a time would be viable!




```
always_comb begin
    z = a + b;
```

```
always_comb begin
    z = a + b + c;
```

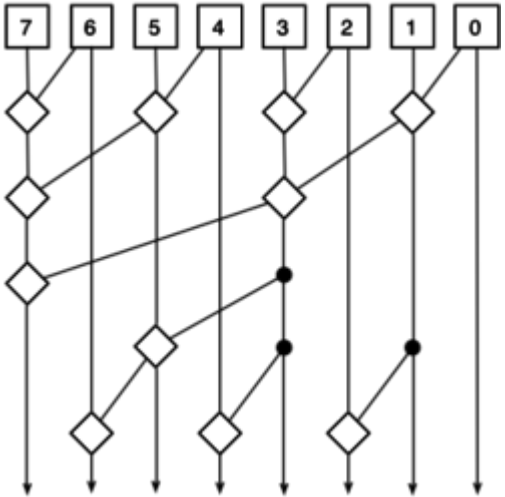
```
always_comb begin
    z = a + b + c + d;
```

```
always_comb begin
    z = a + b + c + d + e;
```

Adder Types

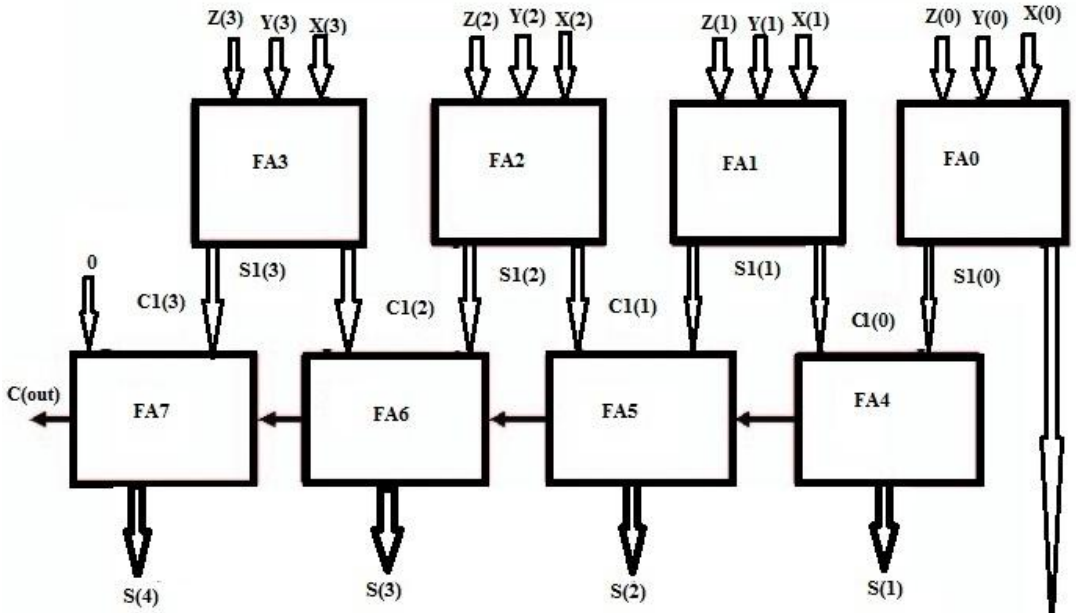
Brent-Kung Adder: add 2 numbers, Yosys default
With a bit width **n**

- Delay ~= $\log_2(n)$
- Area ~= n



Carry Save Adder: can add 3 numbers with minimal extra delay cost
With a bit width **n** and a number of operands 2+**m**

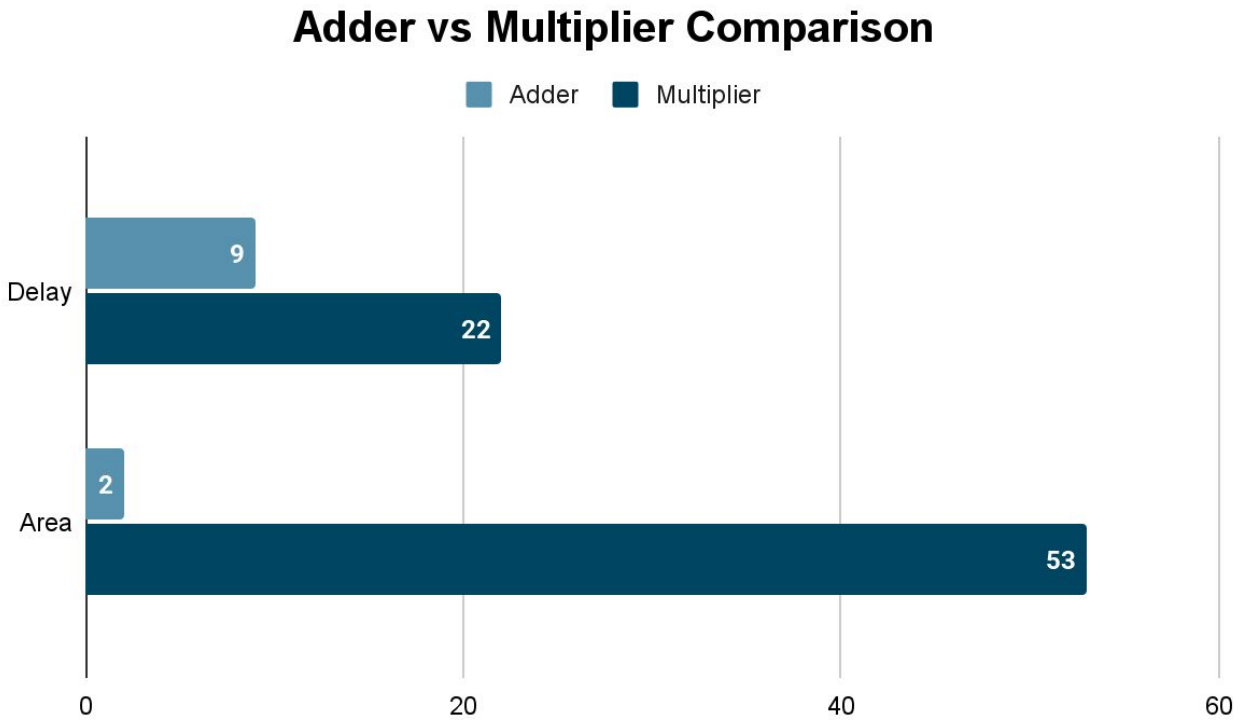
- Delay ~= n + m
- Area ~= n * m



Multiplication

```
always_comb begin
    z = a * b;
```

- Delay several times worse than addition
  - Usually would dominate critical path if left in one cycle
- 32 bit x 32 bit = 64 bit
  - Extreme area usage
- Use bit shifts where possible!
  - Bit shifts are basically free



Division

```
always_comb begin
    z = a / b;
```

- Division is hard
  - Really bad delay
  - Profiled with 32 bit / 32 bit
- If division is necessary:
  - use powers of 2 and bit shifting
  - Write your own multi-cycle division
  - Multi-cycle paths!