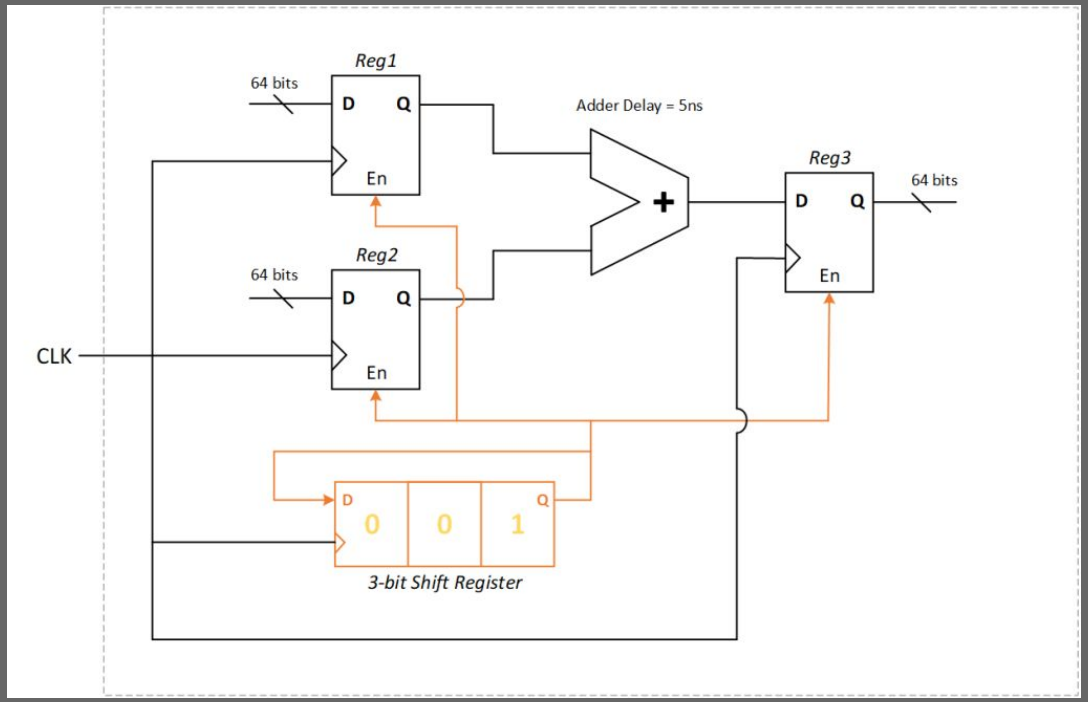
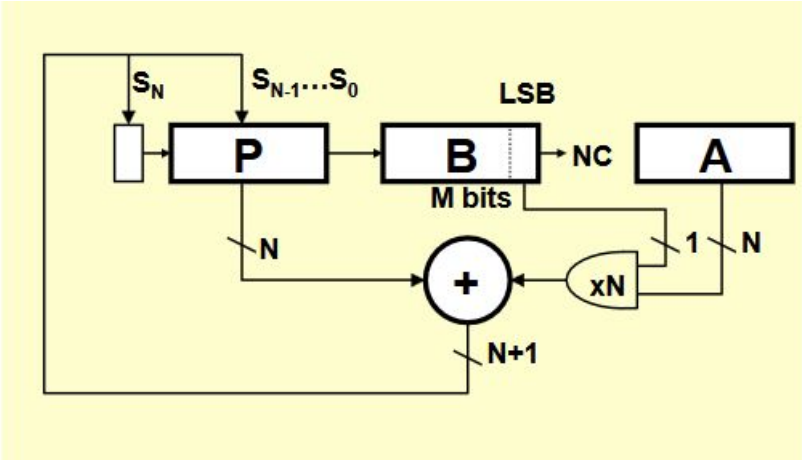
CPE 470 - Accelerator Optimization



Sequential Logic

- Many operations can be expressed as smaller sequential steps
  - Ex: Multiplication as multiple steps of addition

- Can use sequential algorithms to minimize logic usage and critical path
  - Sequential Multiplier uses one adder → shorter critical path
  - Much less area (one adder compared to N adders)
  - Tradeoff: now completes in a lot more cycles
  - 32 bit multiplier takes 32 cycles

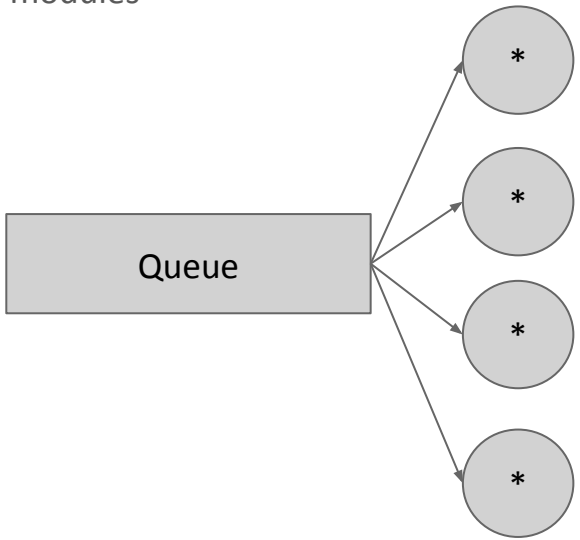


Parallel Sequential

Glossary

Throughput: how often an operation completes
Latency: how long an operation takes to complete from being started

- Sequential Algorithms often take much less area than their unrolled versions
  - Multiplication Example: 1 vs n adder modules for n bits
- Smaller area enables more parallelization: instantiate more of them
  - Queue and Dispatch operations as they come to available modules
  - Modules marked busy as they compute
- With enough parallel modules, throughput can scale
  - Example: With 1 32-bit sequential multiplier:
    - **Throughput** is 1 multiply every 32 cycles
    - **Latency** is 32 cycles (from dispatch to completion)
  - Example: With 32 32-bit sequential multipliers:
    - **Throughput** is 1 multiply every cycle
    - **Latency** is still 32 cycles

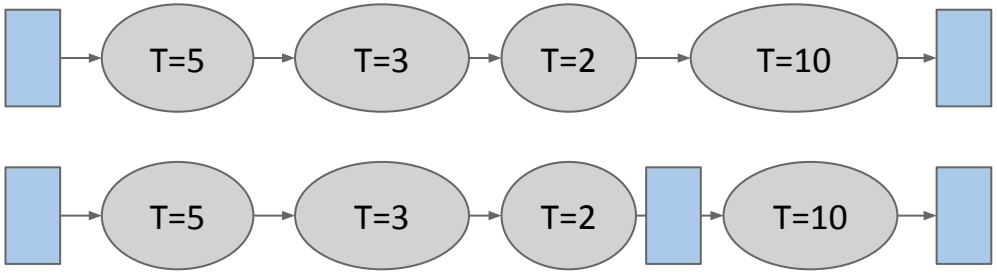


Pipelining

Glossary

Pipelining: breaking a combinational path into steps by inserting registers

- What if we want to improve logic without having to switch to a purely sequential algorithm?
- Pipelining takes existing unrolled combinational block and divide it into steps
  - Each step should have a similar delay
  - Each step gets a register on its output
  - Each step's output register is the input to the next step



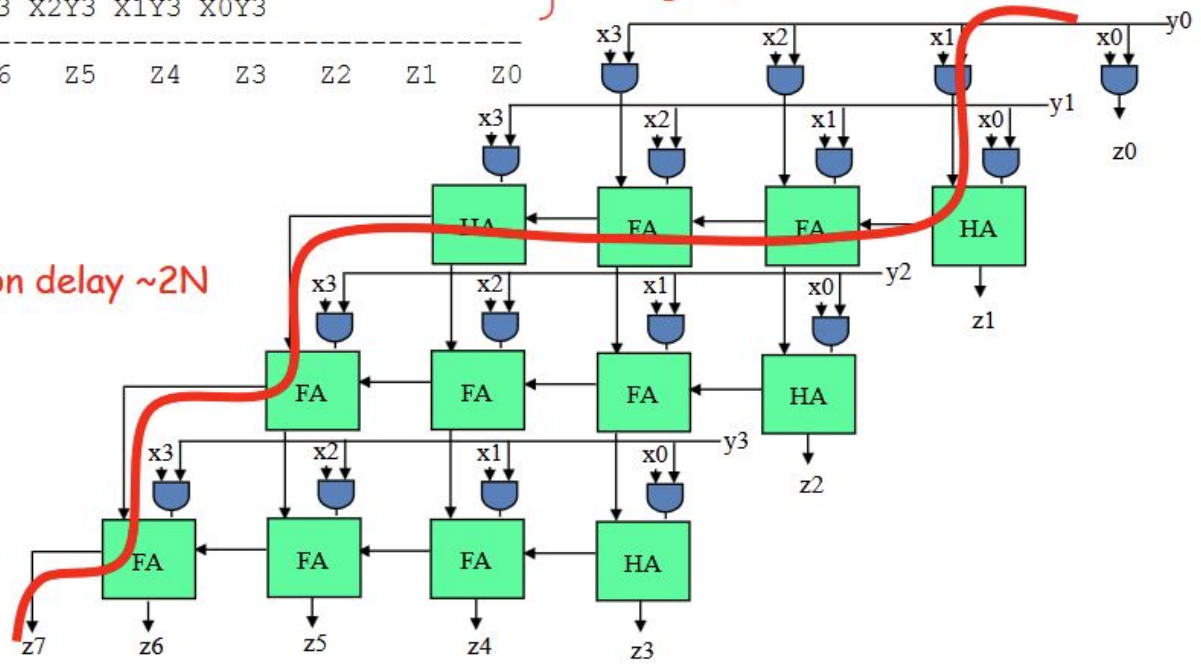
Breaking this logic into two stages brings total logic delay from 20 down to 10

- Pros:
  - Can introduce new data much faster, throughput is improved
  - With a balanced pipeline where every stage is equal delay, n-stages gives n-times the throughput
- Cons:
  - Latency is not improved
  - Registers cost area

Example: Multiplication

```
                        X3   X2   X1   X0   <- multiplicand
                    *   Y3   Y2   Y1   Y0   <- multiplier
                    ------------------------
                       X3Y0 X2Y0 X1Y0 X0Y0
                  + X3Y1 X2Y1 X1Y1 X0Y1
                  + X3Y2 X2Y2 X1Y2 X0Y2
                  + X3Y3 X2Y3 X1Y3 X0Y3
                  ------------------------------------
                  Z7   Z6   Z5   Z4   Z3   Z2   Z1   Z0
```

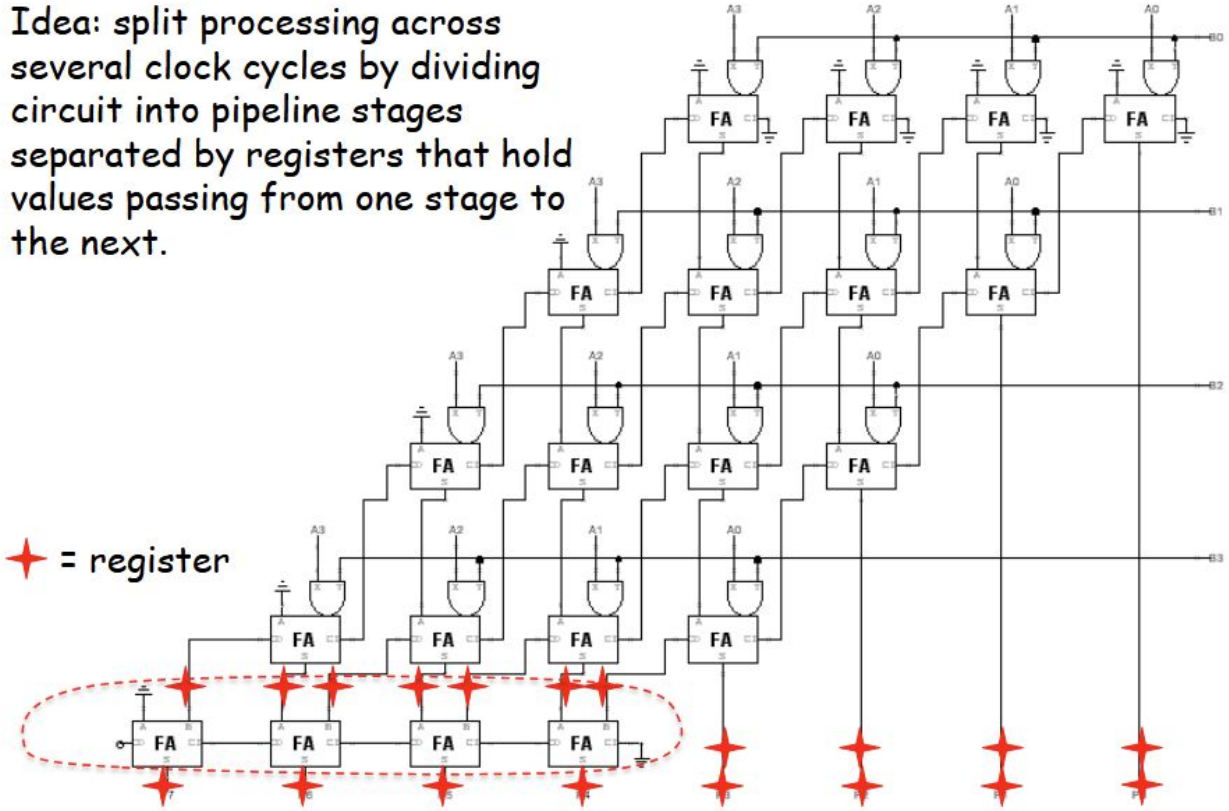
Partial products, one for each bit in multiplier (each bit needs just one AND gate)



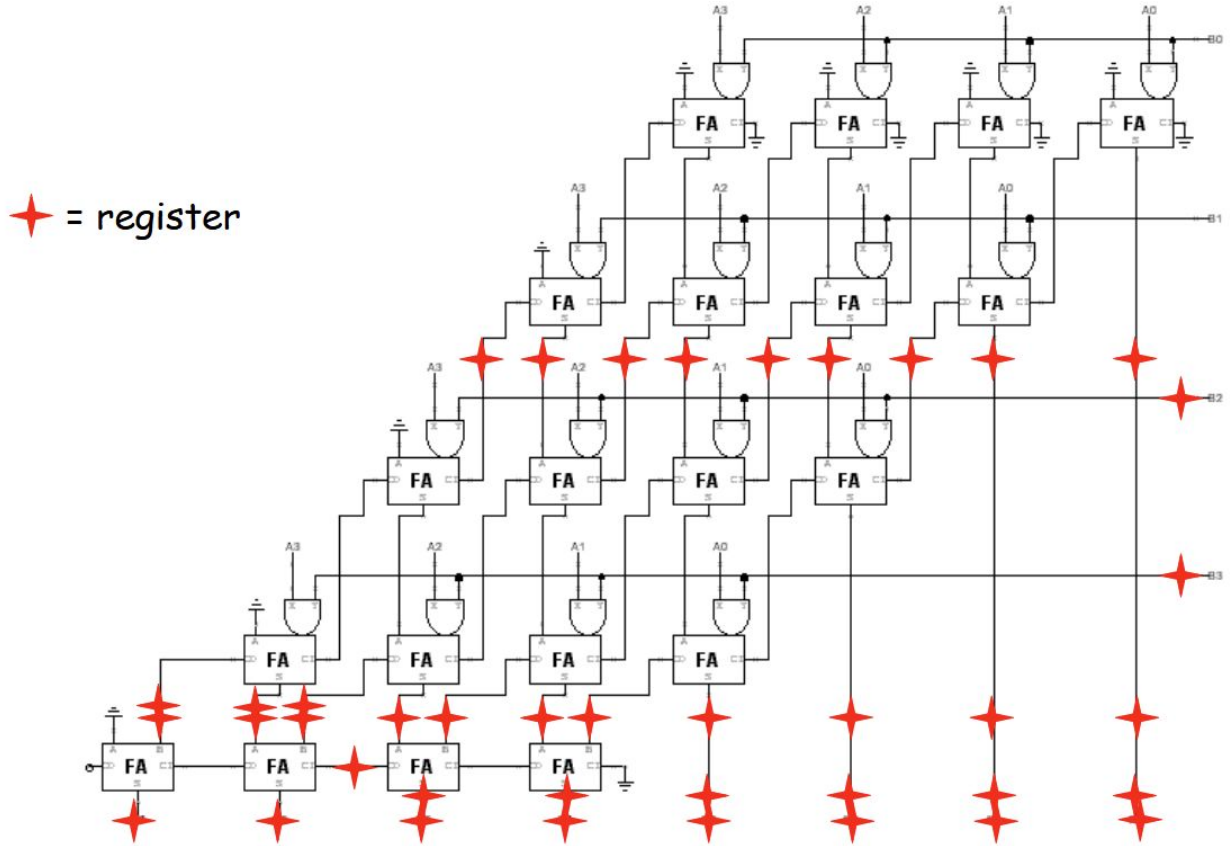
- Propagation delay ~2N

Pipelined Multiplication

Idea: split processing across several clock cycles by dividing circuit into pipeline stages separated by registers that hold values passing from one stage to the next.



Pipelined Multiplication 2

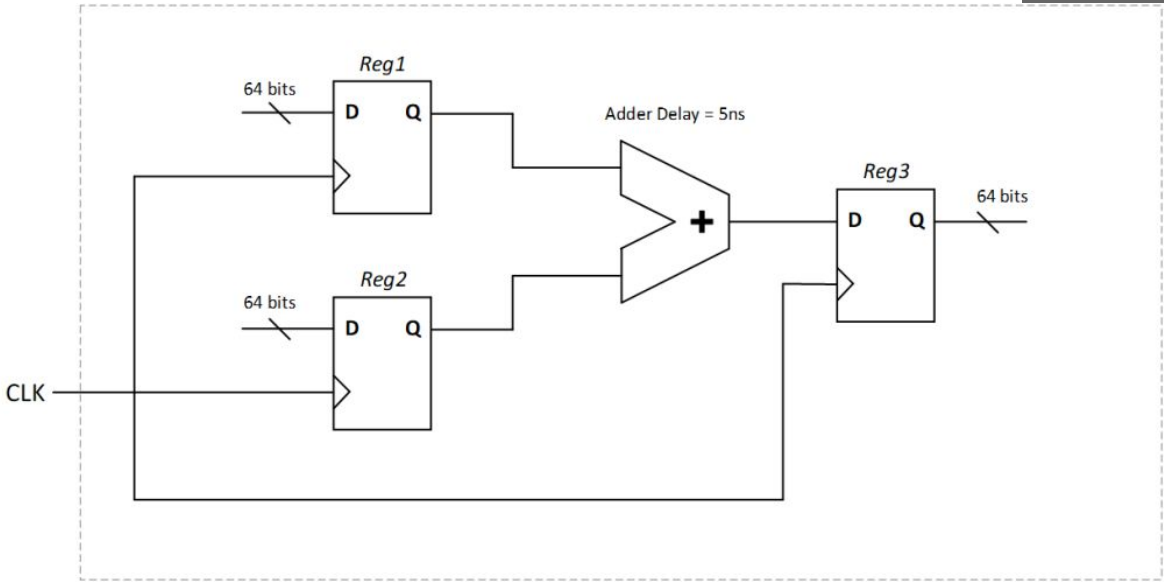


Multi Cycle Paths

Glossary

Multi Cycle Path: combinational logic path that occurs over multiple clock cycles

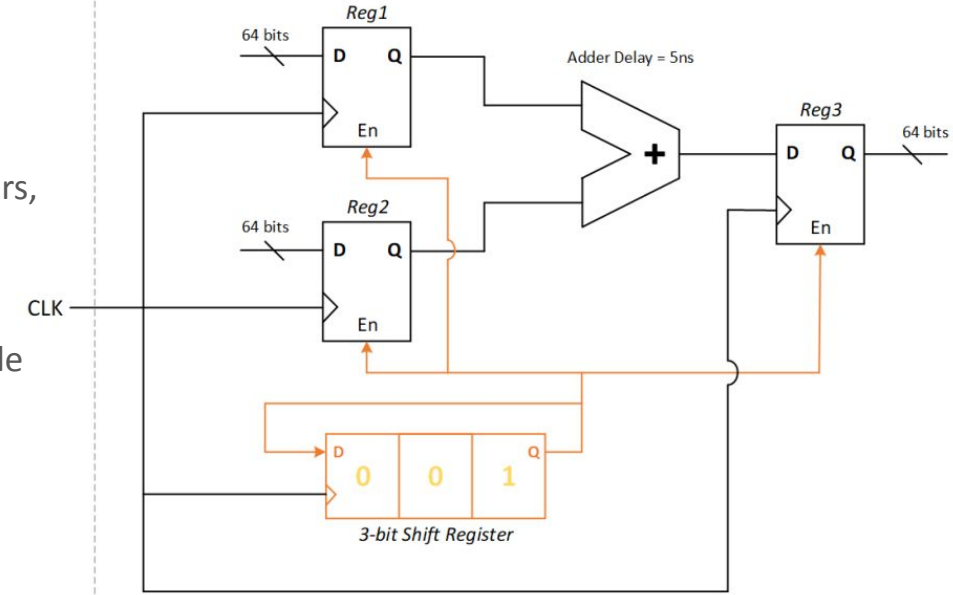
- What if cannot break up an operation cleanly with flip flops?
  - Can't pipeline or sequentialize
- Use a **Multi Cycle Path**!
  - For an n-cycle path, only enable the input and output registers every n cycles
  - Benefits:
    - Can use logic with more delay than your clock period
    - Uses less synchronization registers
  - Drawbacks:
    - Does not get the improved throughput of pipelining
    - One computation at a time
    - Difficult to set up in STA
    - Easy to accidentally hide mistakes
- Example:
  - We want 2ns clock
  - Critical Path is 5ns
  - Use a 3-cycle path



Multi Cycle Path Clocking

- Multi Cycle Paths REQUIRE only activating registers every n cycles
  - Deciding something is a multi-cycle path is not enough
  - Have to manage timing manually with FSM, counters, or shift registers

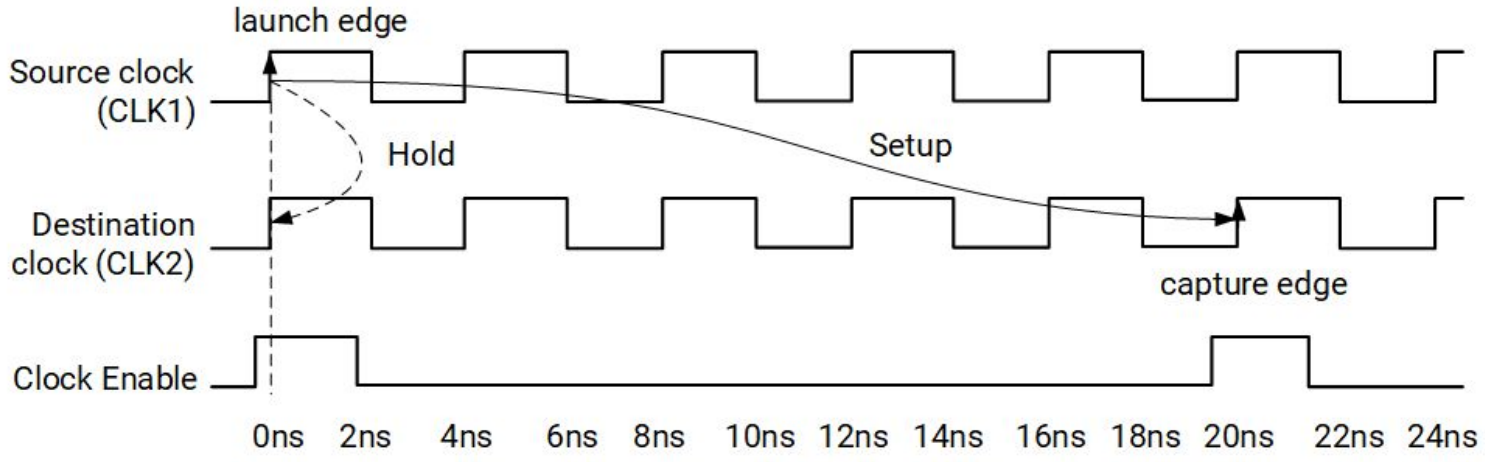
- Example: 3 bit shift register activates flip flops every third cycle



Multicycle Path Timing

- Setup violations should only be checked N cycles out, when the output register is actually enabled
- Hold violations should only be checked during the launch cycle

Figure: Setup=5, Hold=4



Advanced Timing with SDCs

Glossary

SDC: Synopsys Design Constraints

- Basic Setup and Hold time violations are already handled by Openlane
- More advanced timing requires use of **SDC** Files
  - Originally from Synopsys, but now an industry standard
    - Xilinx's XDC format is very similar
  - Uses **TCL** language
- Allows defining special timing paths.
- Two main ways of interacting with design:
  - `[get_ports portname]`
    - gets one or more ports from the top level module of name
  - `[get_pins modulename/pinname]`
    - gets one or more pins, which is an input/output of an instantiated submodule

Create a clock signal named "clk" with T=10ns on the top-level port "clk"
Simulate the propagation delay of the clock more accurately

```
create_clock [get_ports clk] -name clk -period 10
set_propagated_clock clk
```

Multi Cycle Paths in SDCs

- Defined from the **Clock Pin** of the starting register to the **Data Input** pin of the ending register
- Setup path sets how many cycles
  - Example: Division takes ~165 ns, we want a clock period of 10ns
  - Set to a 17 cycle path
- Hold path should be one less than the setup path
- **Problem**: timing is done AFTER synthesis, on the gate-level netlist
  - Have to find signal names from your netlist (.nl.v file)
  - Use wildcard * to match all signals of a multi-bit vector

```
div Comb (.a_reg(a_reg), .b_reg(b_reg), .comb_out(comb_out));
```

```
set_multicycle_path -setup 17 -from [get_pins Comb.a_reg*/CLK ] -to  [get_pins  multicycle_out*df*/D]
set_multicycle_path -hold  16 -from [get_pins Comb.a_reg*/CLK ] -to  [get_pins  multicycle_out*df*/D]
```

17 cycle path → 17 setup, 16 hold

Path starts at the clock pin of the Comb module's a_reg pin

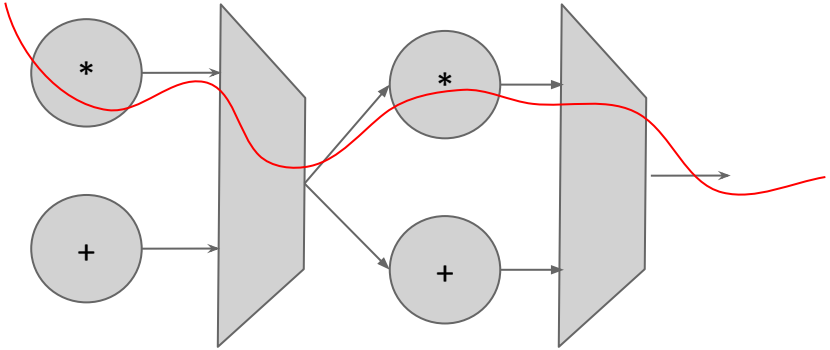
Path ends at the data pin of a register outside the Comb module

False Paths

Glossary

False Path: timing path that is never utilized

- Sometimes designs will have critical paths that are not every actually used
- Timing tools don't know the state of muxes or logic; assume all paths are taken
  - **False Paths** are found by tools even though they are never reached at runtime
  - `set_false_path`
    - Similar syntax to multicycle paths from previous slides



- Example: design a module that multiply-adds or add-multiplies
  - Never double multiplies
- Critical path appears to be double multiply
  - Timing tool doesn't know which muxes are active or mutually exclusive
- Use false path to disable the unused path
  - Total delay is now delay(+,*) rather than delay(*,*)

References

- https://vlsitutorials.com/constraining-multi-cycle-path-in-synthesis/
- https://courses.csail.mit.edu/6.111/f2008/handouts/L09.pdf