Computer-Aided VLSI System Design

Chap.4-1 Architecture Improvement of Timing, Area, and Power

Lecturer: 孫振庭 (Brian)

Graduate Institute of Electronics Engineering, National Taiwan University

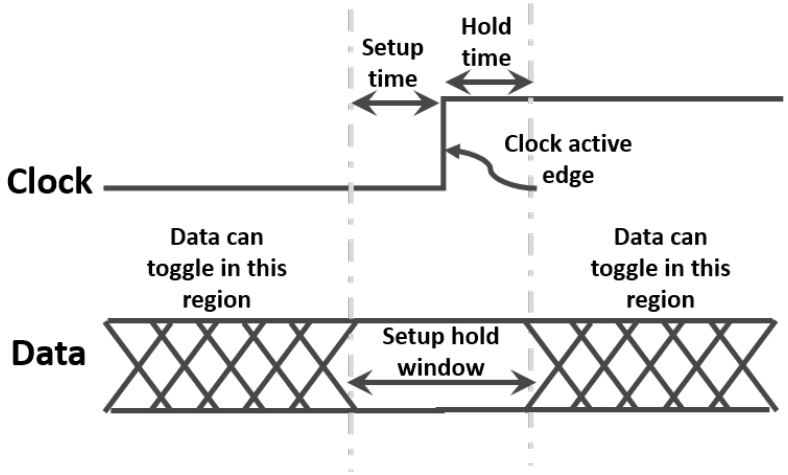
NTU GIEE

Outline

- Introduction
- Register Timing
- Timing Improvements
  - Pipeline
  - Retiming
  - Parallel
- Area and Power

Introduction to Hardware Performance

- Designers should be aware of common **performance metrics**
  - Timing
  - Area
  - Power
- **Meeting timing requirements is the fundamental goal**
  - A digital circuit may not work if it has timing violations
  - Optimize area and power only after timing is met



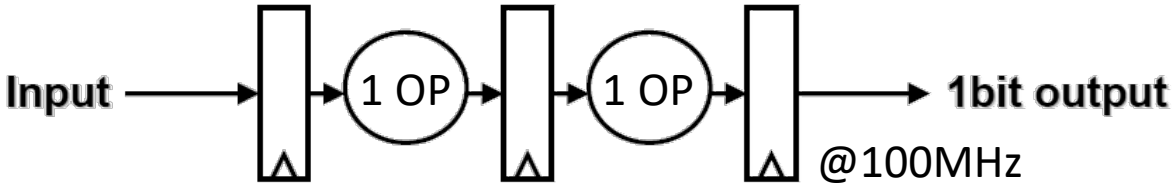
Performance Optimization

- Limitation of synthesis tools
  - Cannot automatically infer hardware constraints (e.g. clock speed, input/output delay)
  - Cannot resolve all timing, area, and power issues
- Performance optimization is done better at the algorithmic and architecture level
  - Cannot rely on synthesis tool all the time
  - Plan and analyze your design for better performance

Latency and Throughput

- **Latency**
  - The *time* taken to complete an operation
  - e.g. cycles, seconds
- **Throughput**
  - The *rate* of operations being completed
  - e.g. OP/cycle, x-bit width/seconds)

- **Example**



  - **Latency**: 3 cycles, or 30 ns
  - **Throughput**: 2 OP/cycle, 100 Mb/s, or 200 MOPS

Timing Requirements

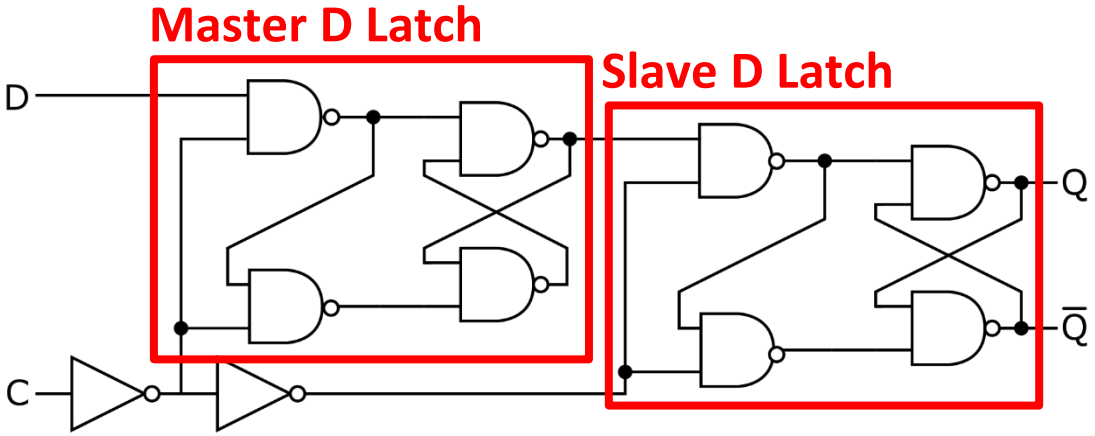
- To meet throughput requirements in system specifications, the clock cycle must be smaller than some values
- The design must meet timing with margin, and use worst-case library model in synthesis
- If the design cannot deliver certain throughput in post-synthesis simulation, we need to improve timing with the methods below:
  - Pipeline
  - Retiming
  - Parallel

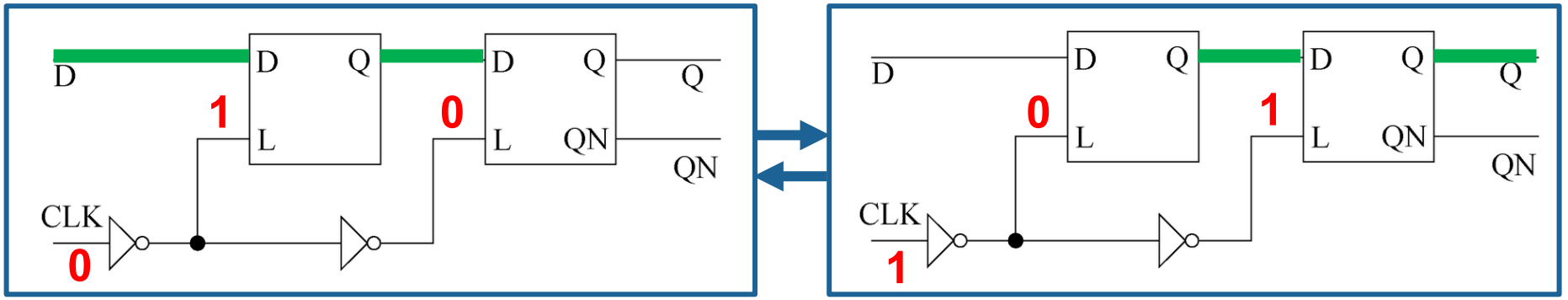
Outline

- Introduction
- Register Timing
- Timing Improvements
  - Pipeline
  - Retiming
  - Parallel
- Area and Power

Register Architecture

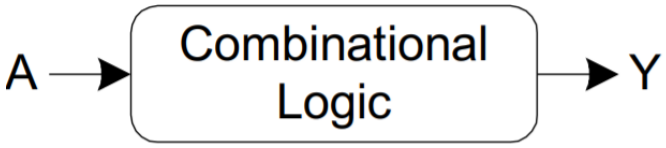
- Positive edge-triggered D flip-flop

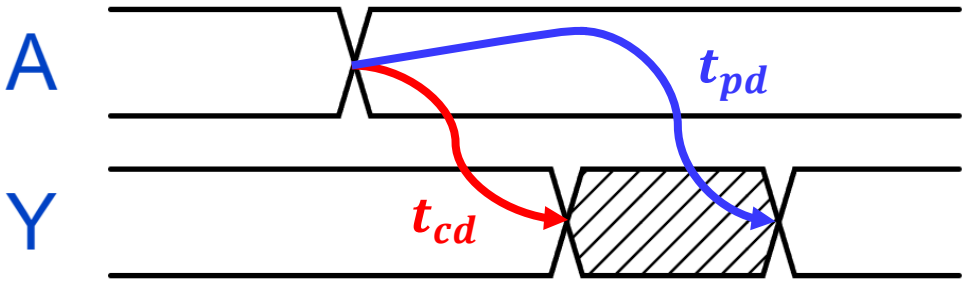




Register Timing: Notation (1/2)

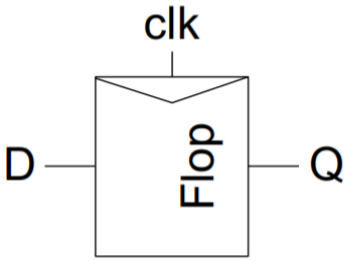
Name	Definition
t_{pd}	Logic propagation delay (maximum logic delay)
t_{cd}	Logic contamination delay (minimum logic delay)

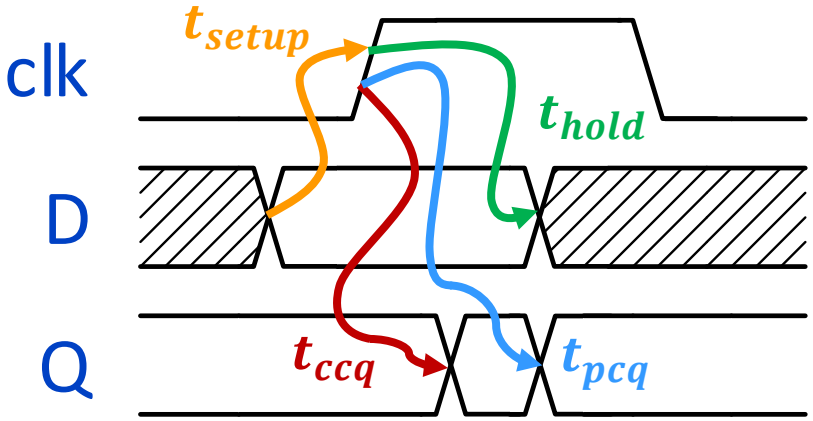




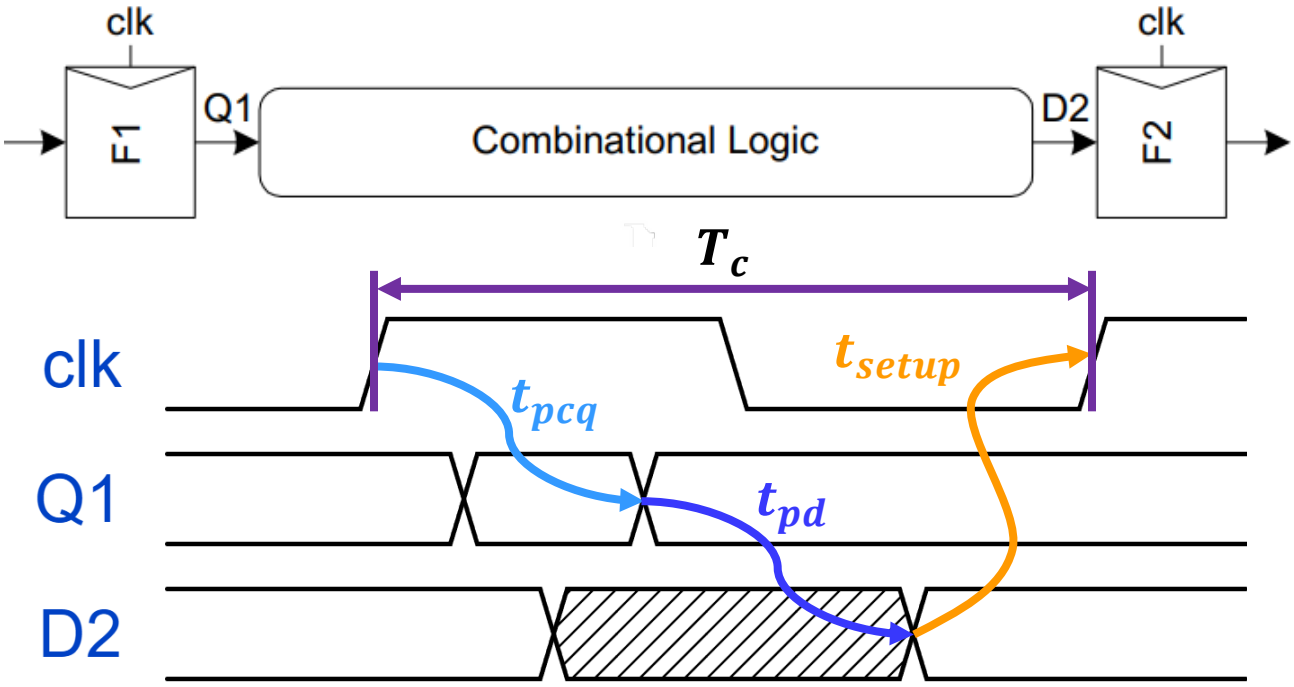
Register Timing: Notation (2/2)

Name	Definition
t_{pcq}	Clock-to-Q propagation delay (maximum clk-Q delay)
t_{ccq}	Clock-to-Q contamination delay (minimum clk-Q delay)
t_{setup}	Setup time (D must be stable for t_{setup} **before** posedge clock)
t_{hold}	Hold time (D must be stable for t_{hold} **after** posedge clock)
T_c	Clock period





Register Timing: Max Delay



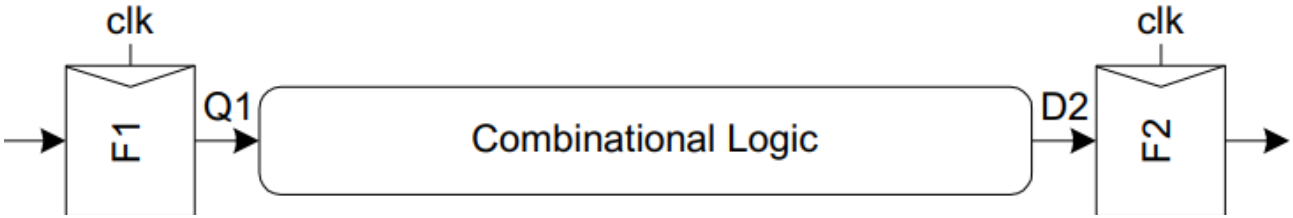
Requirement:

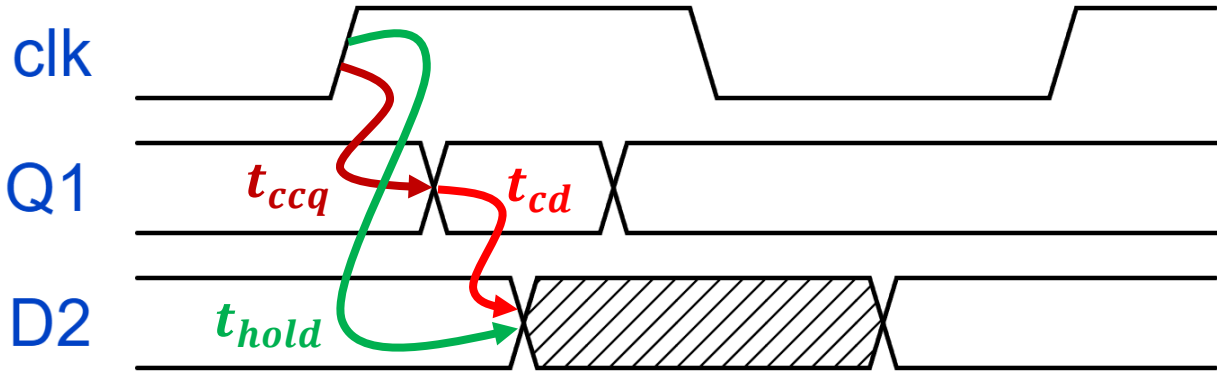
$$t_{pd} \leq T_c - \underbrace{\left(t_{\text{setup}} + t_{pcq}\right)}_{\text{sequencing overhead}}$$

Setup time violation:

$$T_c - t_{pd} - t_{pcq} < t_{setup}$$

Register Timing: Min Delay





Requirement:

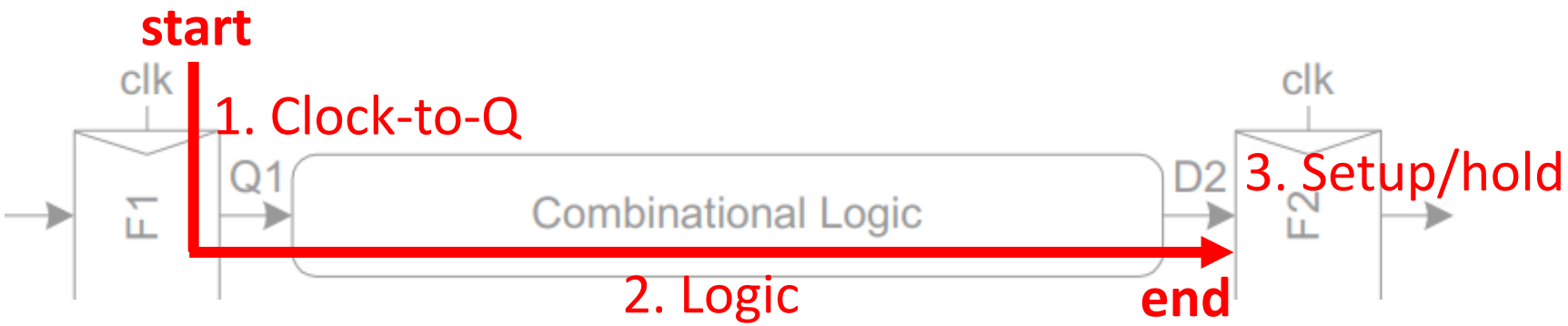
$$t_{cd} \geq t_{\text{hold}} - t_{ccq}$$

Hold time violation:

$$t_{cd} + t_{ccq} < t_{hold}$$

Calculate Register Timing

- Note: register timing calculation starts from the **clock** signal



Timing Violations

- **Slack** should be non-negative, otherwise it is a timing violation
  - Setup slack : $T_c - t_{pd} - t_{pcq} - t_{setup}$
  - Hold slack : $t_{cd} + t_{ccq} - t_{hold}$

- **Fixing setup time violation**
  1. Larger T_c (lower frequency)
  2. Lower t_{pd} (pipelining, retiming)

- **Fixing hold time violation**
  1. Larger t_{cd}
  - **Note**: hold time violation cannot be fixed by adjusting T_c

Outline

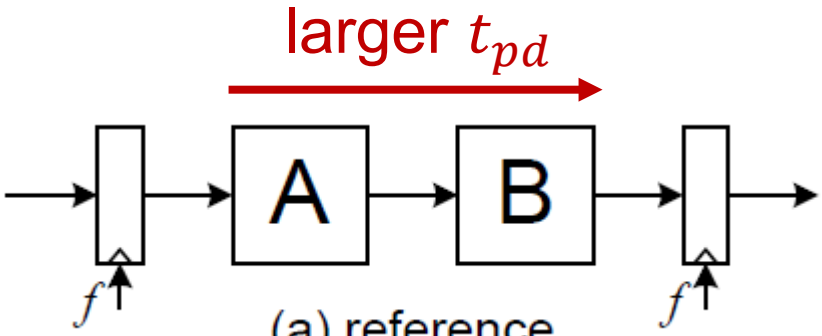
- Introduction
- Register Timing
- Timing Improvements
  - Pipeline
  - Retiming
  - Parallel
- Area and Power

Improve Timing in a Design (1/2)

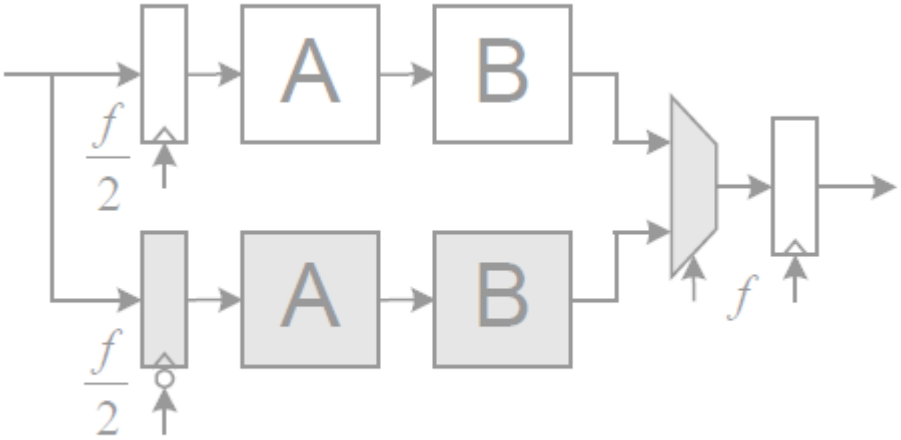
- **Pipelining**: exploits **temporal** parallelism
  - Insert pipeline registers without changing functionality
  - Shorter critical path -> faster achievable clock
  - Trade off latency (in cycles) to improve throughput

- **Parallelizing**: exploits **spatial** parallelism
  - Duplicate function units, works in parallel
  - With slower clock : achieve same throughput
    With same clock : achieve higher throughput
  - Trade off area to improve throughput

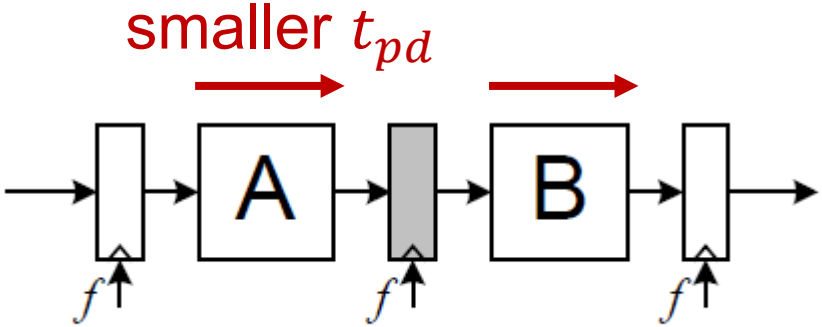
Improve Timing in a Design (2/2)



(a) reference



(b) parallel



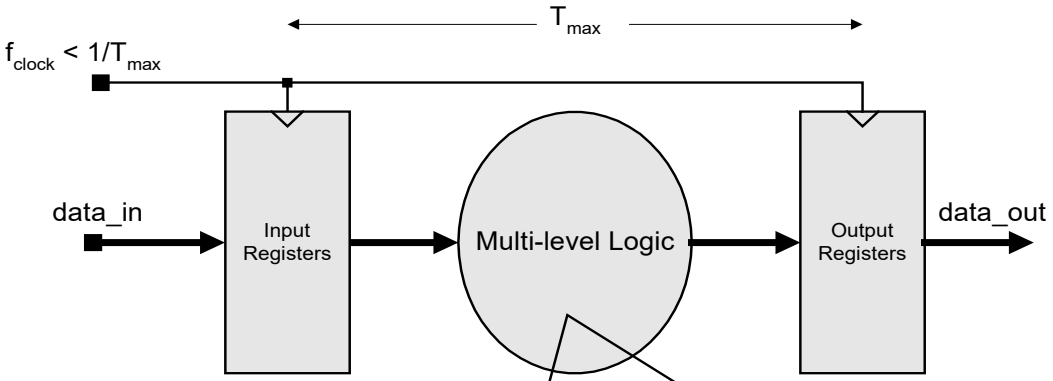
(c) pipeline

Outline

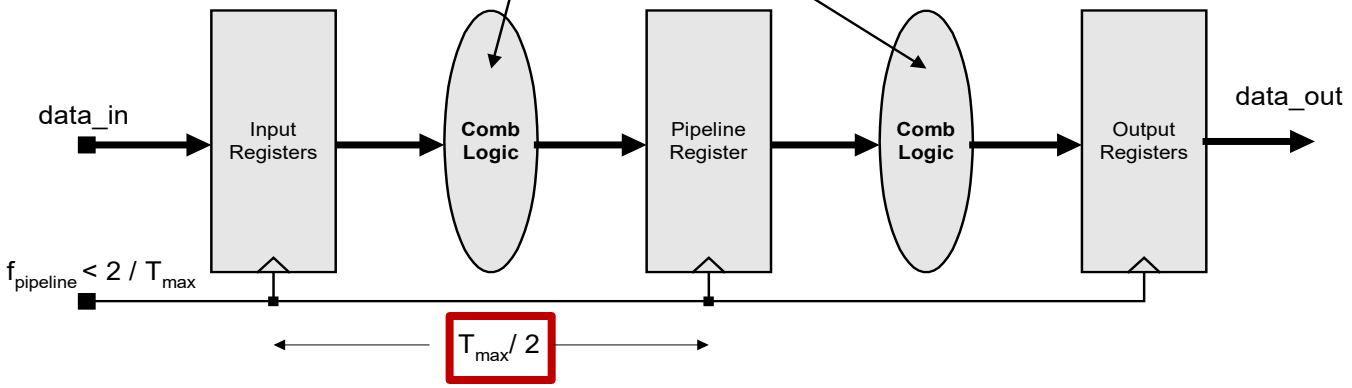
- Introduction
- Register Timing
- Timing Improvements
  - Pipeline
  - Retiming
  - Parallel
- Area and Power

Pipelining (1/2)

- Original



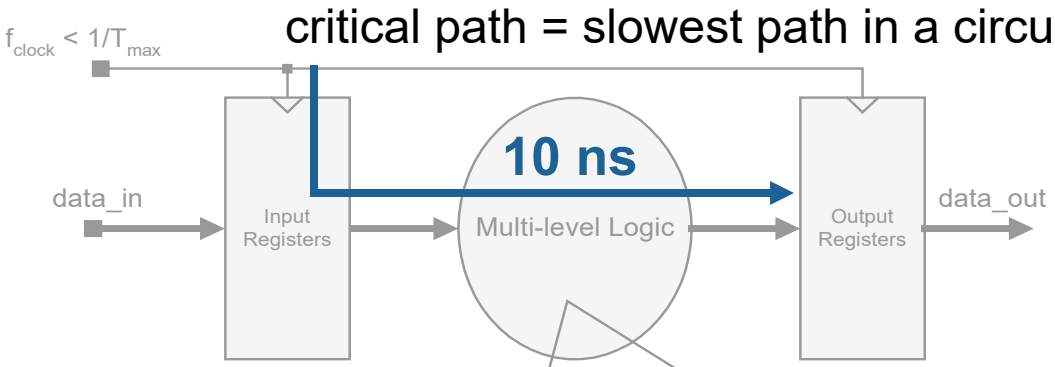
- 2-stage pipelining



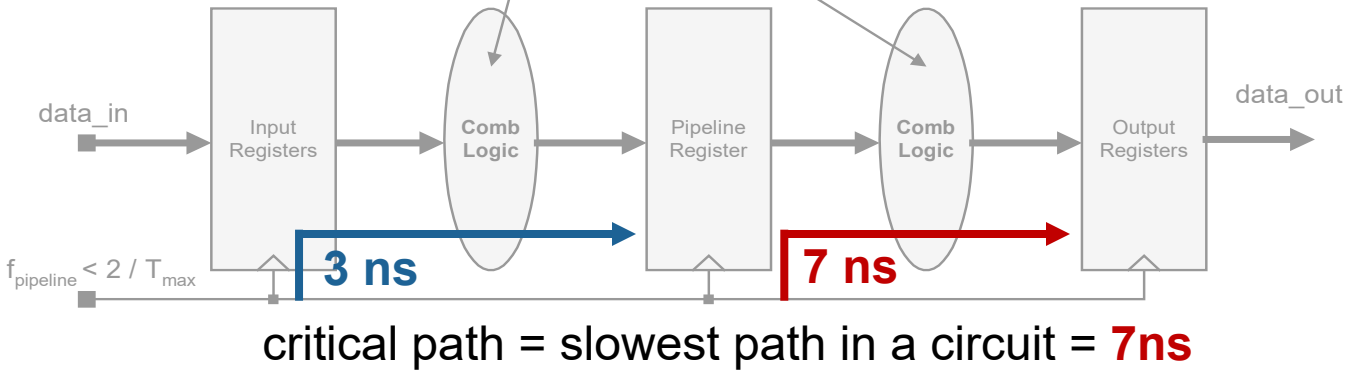
2x faster clock

Pipelining (2/2)

- **Original**



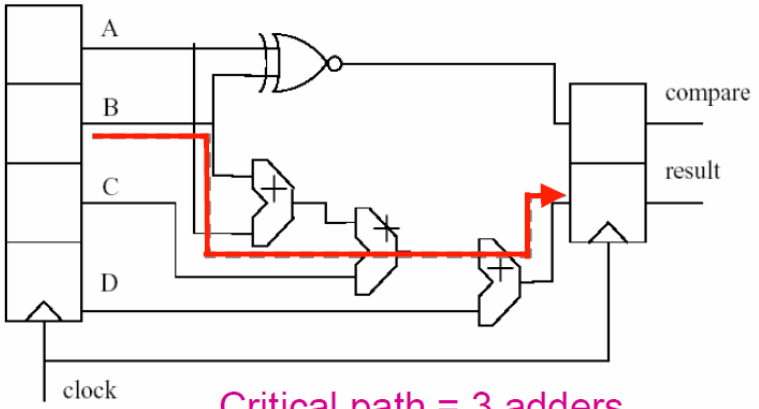
- **2-stage pipelining**



Example: A Simple Circuit

Original

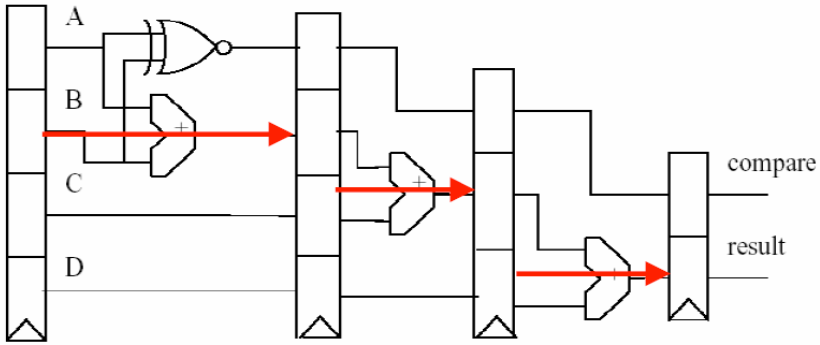
```
always @ (posedge clk) begin
    compare <= A ~^ B;
    result  <= A + B + C + D;
end
```



Critical path = 3 adders

3-stage pipelining

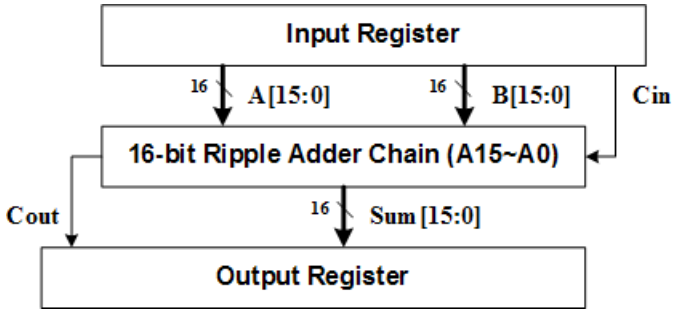
```
always @ (posedge clk) begin
    stage1_1 <= A ~^ B;
    stage1_2 <= A + B;
    stage1_3 <= C;
    stage1_4 <= D;
    stage2_1 <= stage1_1;
    stage2_2 <= stage1_2 + stage1_3;
    stage2_3 <= stage1_3;
    compare  <= stage2_1;
    result   <= stage2_2 + stage2_3;
end
```



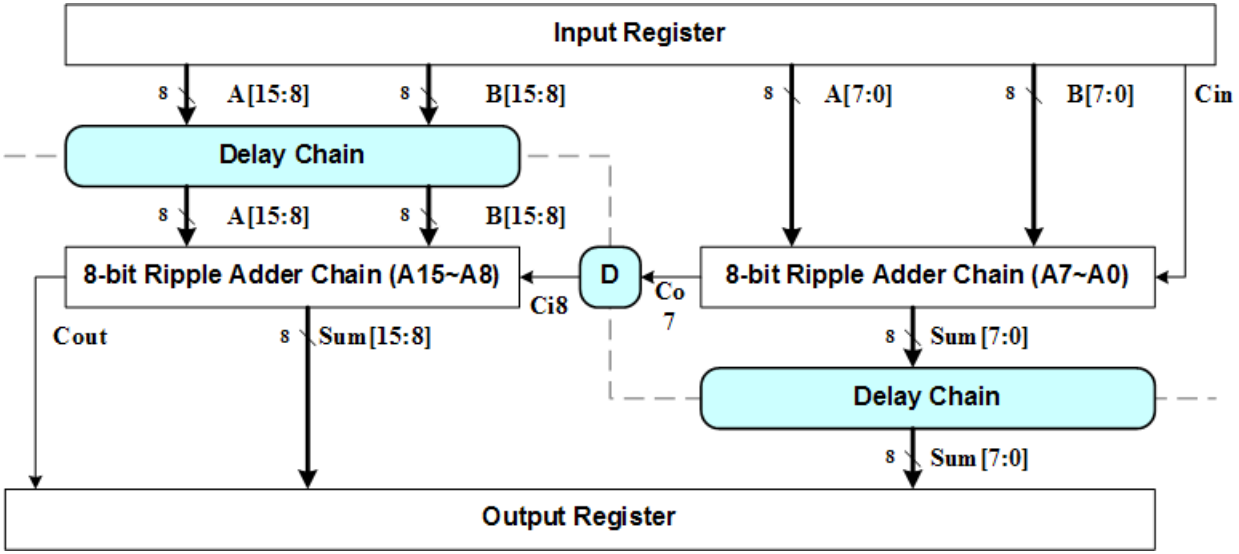
Critical path = 1 adders

Example: Pipelined 16-bit Adder

- Original



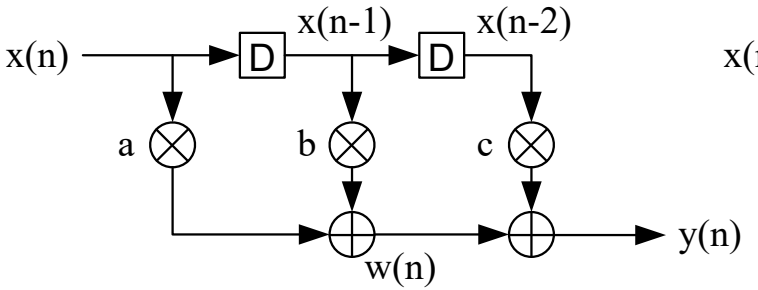
- 2-stage pipelining



Example: FIR Filter

Original

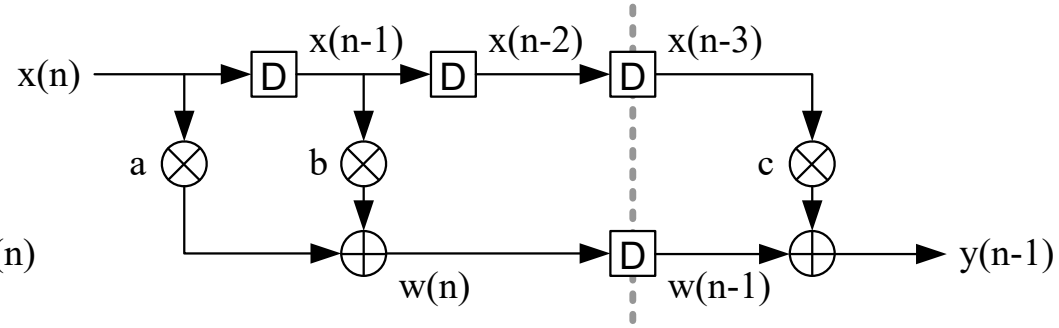
$$w(n) = ax(n) + bx(n-1)$$
$$y(n) = cx(n-2) + w(n)$$



critical path = $T_{MUL} + 2T_{ADD}$

2-stage pipelining

$$w(n-1) = ax(n-1) + bx(n-2)$$
$$y(n-1) = cx(n-3) + w(n-1)$$

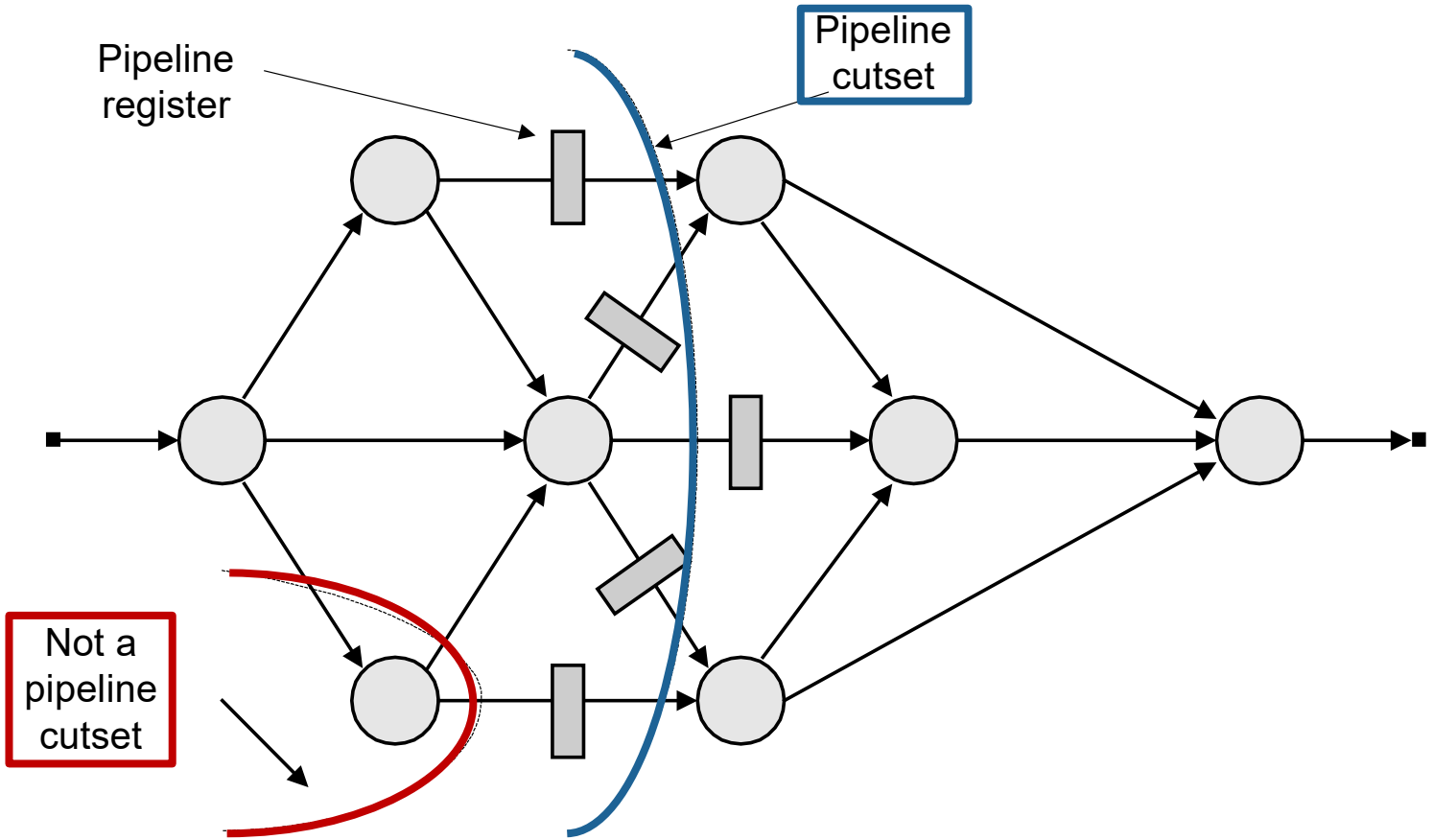


critical path = $T_{MUL} + T_{ADD}$

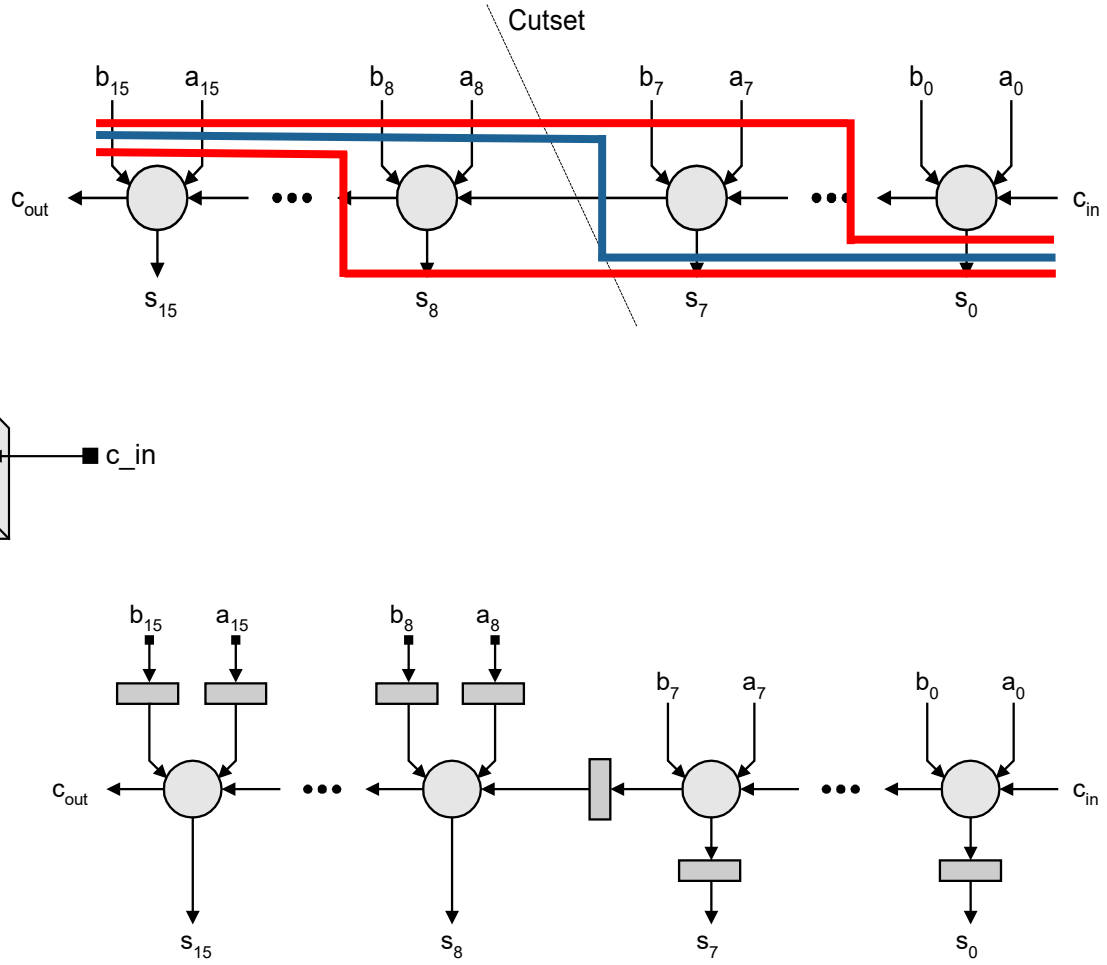
Pipelining a Design

- Draw the circuit diagram as a **directed graph**
- Put pipeline registers on **feed-forward cutset** of the graph
  - **Cutset**: a set of edges such that if they are removed from the graph, the graph becomes two disjoint sets
  - **Feed-forward**: all the edges have the same direction from one disjoint set to the other
- A **pipeline cutset** is a feed-forward cutset and **all inputs and outputs are in different disjoint sets**

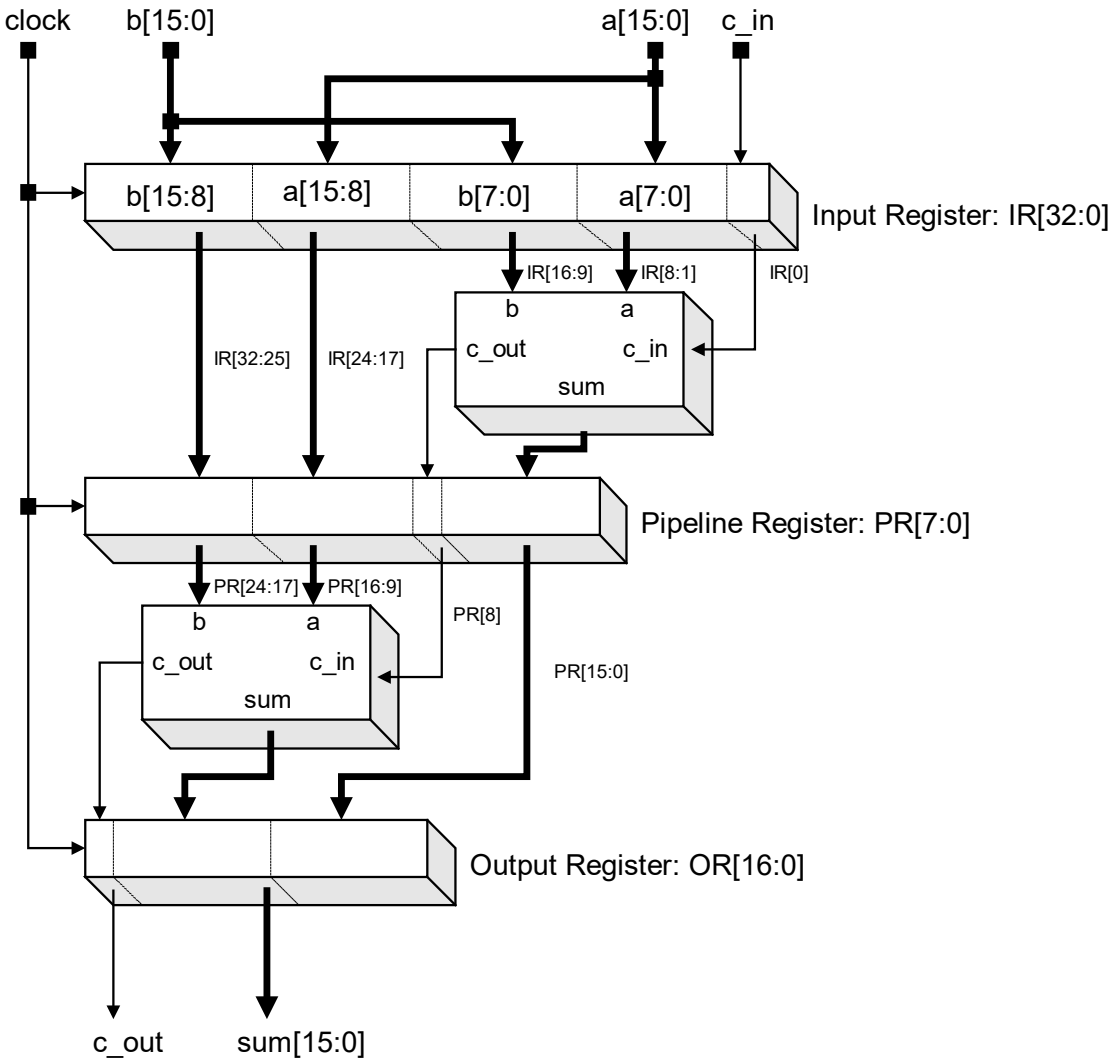
Feed-Forward Cutset



Balancing Performance between Stages



Example: Pipelined 16-bit Adder



Pipeline Overhead

- **Area overhead** from pipeline registers
- **Latency overhead** from pipeline registers
  - N-stage pipeline registers -> N-cycle latency
- Some additional overhead
  - Register setup time
  - Non-ideal separation-> not exactly 1/N period
- **Retiming** for more balanced pipeline stage separation

Outline

- Introduction
- Register Timing
- Timing Improvements
  - Pipeline
  - Retiming
  - Parallel
- Area and Power

Retiming (1/2)

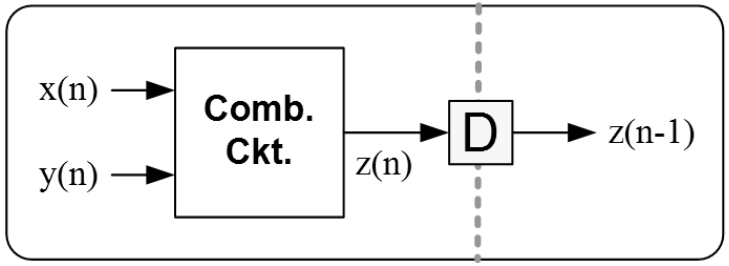
- **Changing location of registers without affecting functionality**
  - Balance latency on different combinational path
  - Possible reduction on clock period

- **Note**
  - May change the number of registers in a design
  - Retiming can be applied by synthesis tool

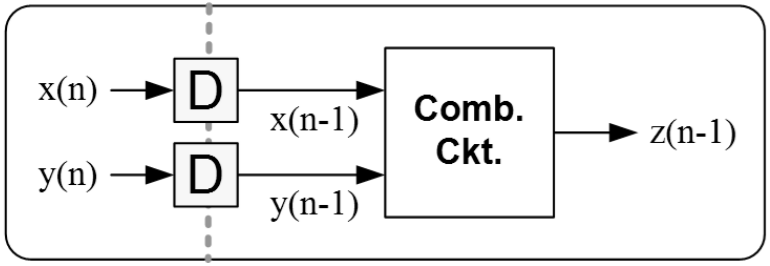
Retiming (2/2)

Cutset retiming

- Moving/adding registers on a feed-forward cutset



1 register



2 registers

In Design Compiler:

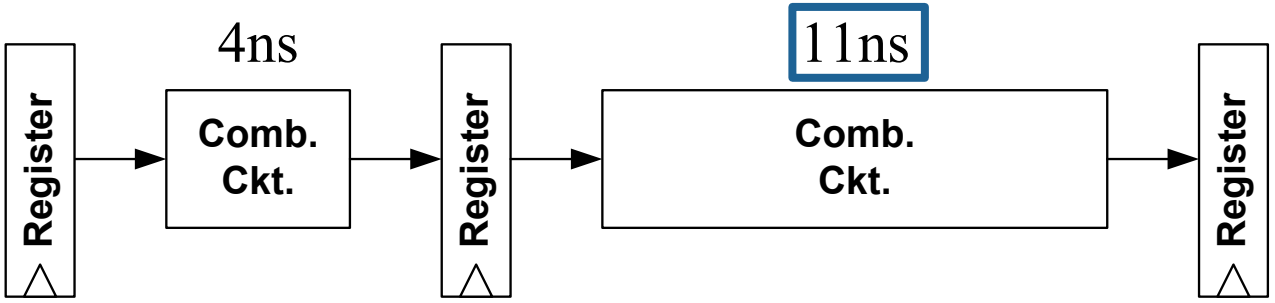
- `compile_ultra -retime`

  (no retiming by default)

Example: Retiming (1/2)

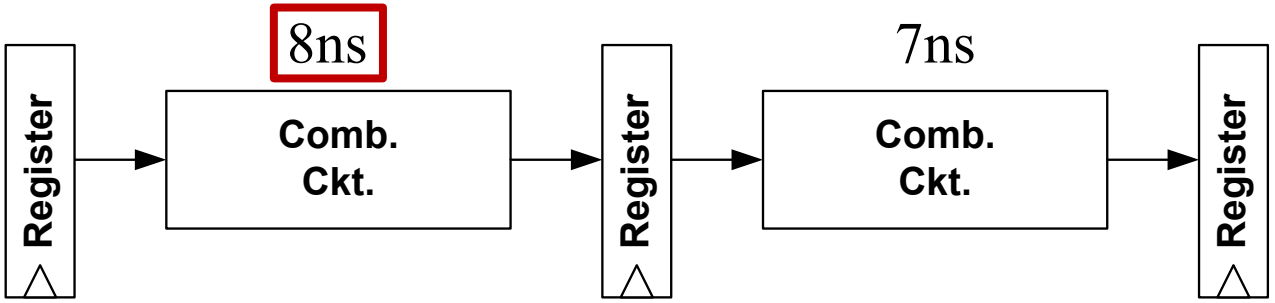
- **Original**

Clock period > 11ns



- **After Retiming**

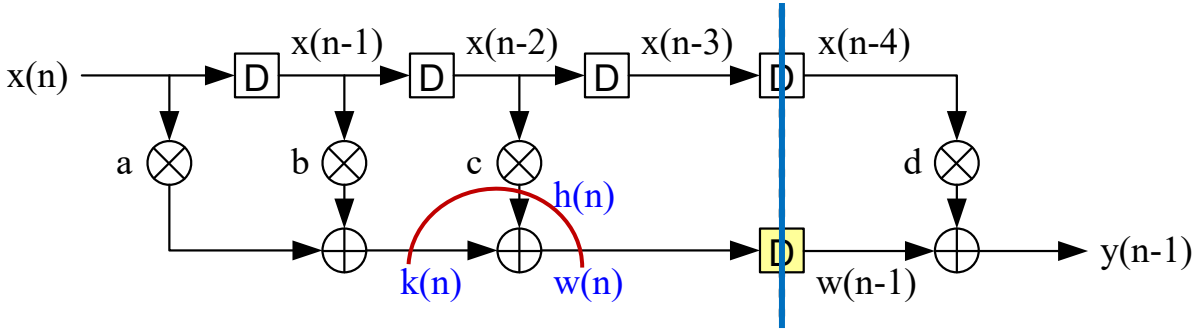
Clock period > 8ns



Example: Retiming (2/2)

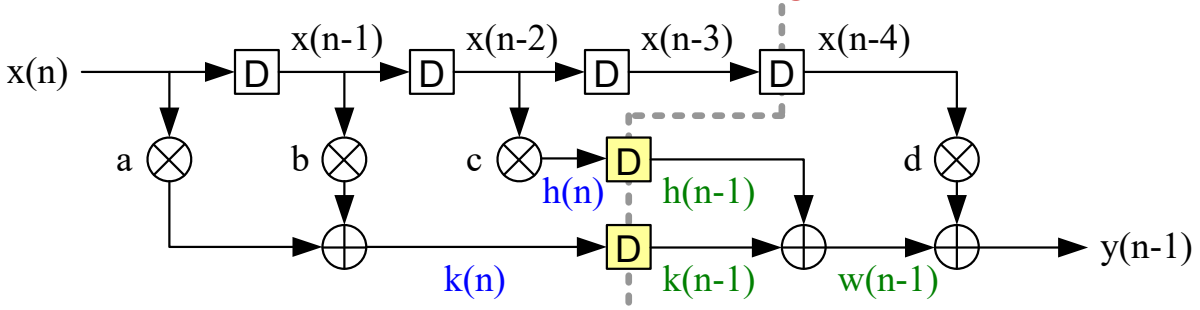
- **Original**

Clock period > $T_{MUL} + 2T_{ADD}$



- **After Retiming**

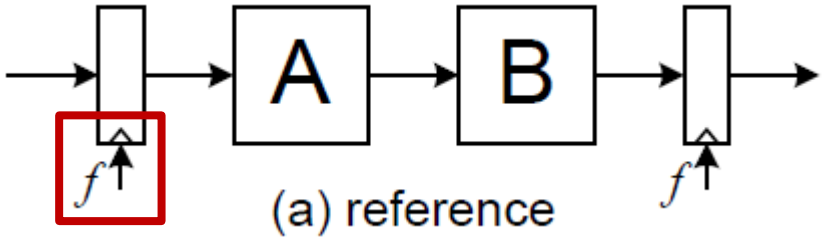
Clock period > $T_{MUL} + T_{ADD}$



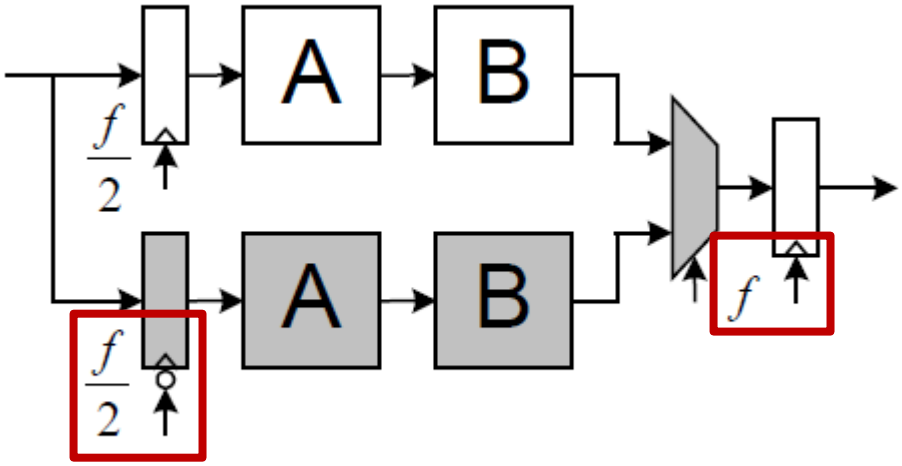
Outline

- Introduction
- Register Timing
- Timing Improvements
  - Pipeline
  - Retiming
  - Parallel
- Area and Power

Parallel Architecture

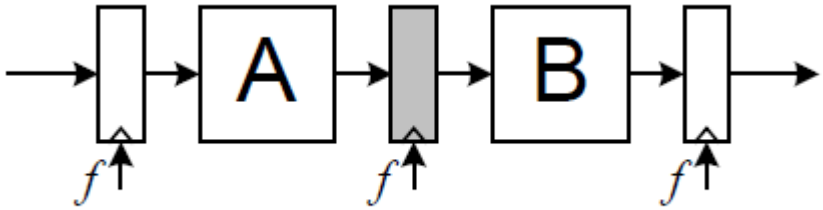


(a) reference



larger T_c (b) parallel

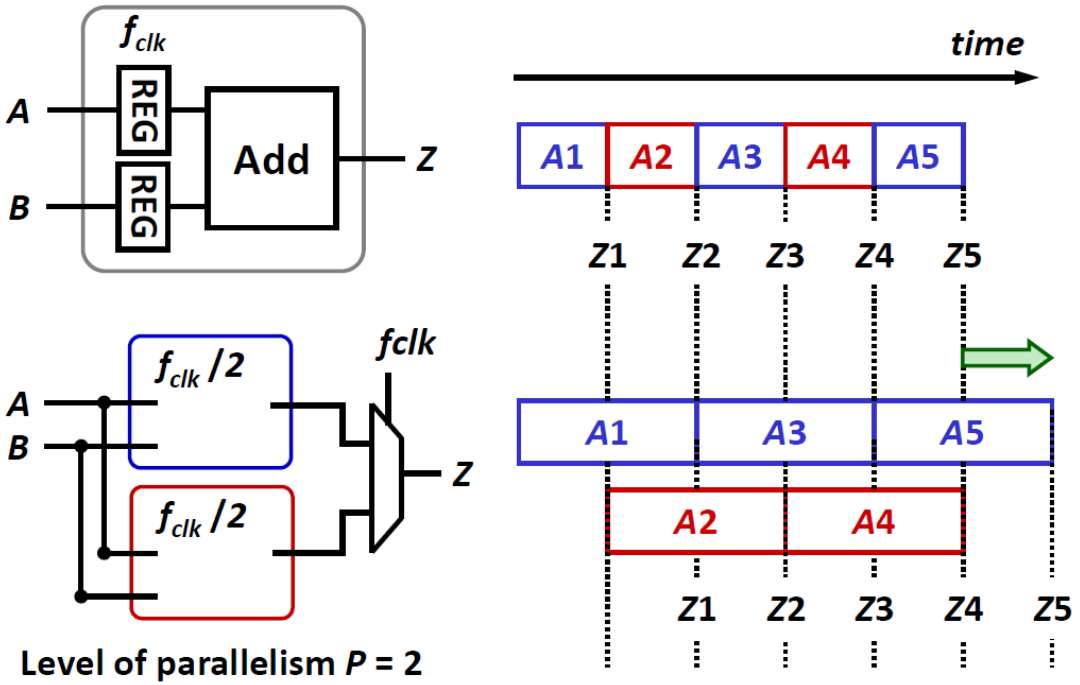
same throughput



(c) pipeline

Parallel Architecture: Timing

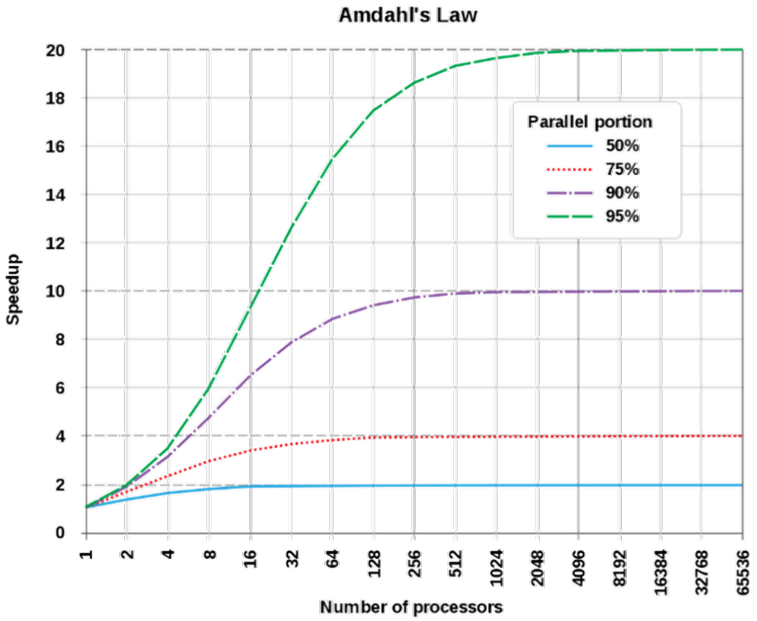
- For circuits with **same throughput**, parallel architecture will have:
  - Lower frequency
  - Larger area
  - Lower dynamic power
  - Longer latency



Parallel Architecture: Performance

- For circuits with **same frequency**, parallel architecture can also be used to increase performance

- Amdahl's Law: number of parallel architecture may not speed up along with parallelizing factor
  - Non-parallelizable operations
  - Dependency between data
  - Dependency between operations
  - Limitation of I/O bottleneck (communication bounded)



Outline

- Introduction
- Register Timing
- Timing Improvements
  - Pipeline
  - Retiming
  - Parallel
- Area and Power

Area Issues

- **Area is cost**
  - During design process, designers should be aware of area
  - Basic approach: hardware sharing

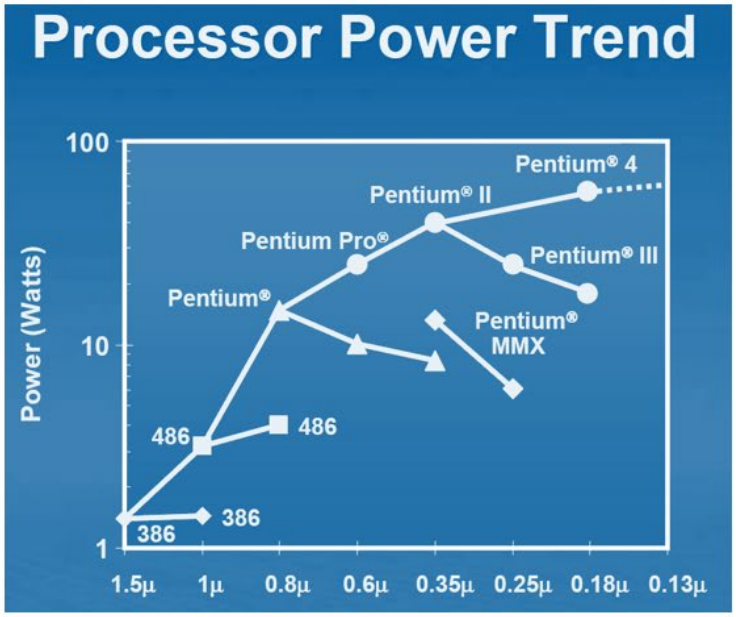
```
wire [15:0] A, B, C;
wire [ 7:0] P, Q, R;
wire        mult_sel;

assign A = P * Q;
assign B = P * R;
assign C = mult_sel ? A : B;
```

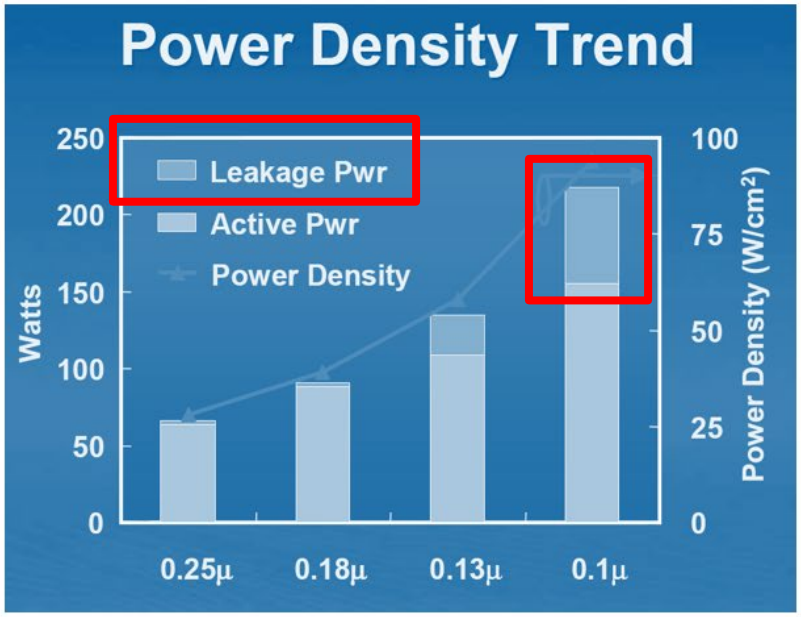


```
wire [15:0] C;
wire [ 7:0] P, Q, R, S;
wire        mult_sel;

assign S = mult_sel ? Q : R;
assign C = P * S;
```

- There are tradeoffs between area and timing
  - Can be specified in synthesis constraints

Power Issues





- Low power design is more important in modern chips due to heat dissipation, packaging, and portability requirements

Dynamic Power in CMOS

$$\text{Dynamic power, } P = \sum \alpha\, C\, V_{dd}^2\, f$$

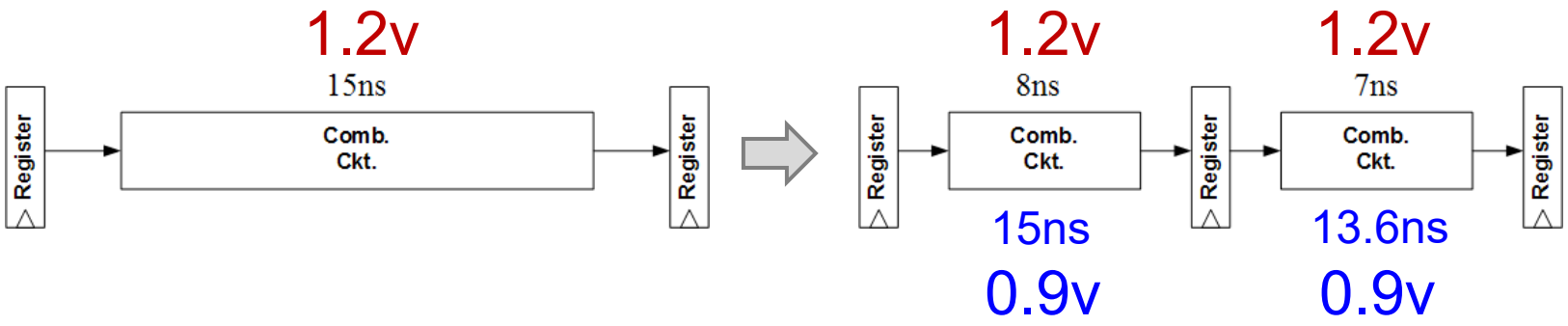
- α : switching activity
- f : clock frequency
- C : node capacitance
- V_{dd} : power supply voltage

Strategies for Low Power Design

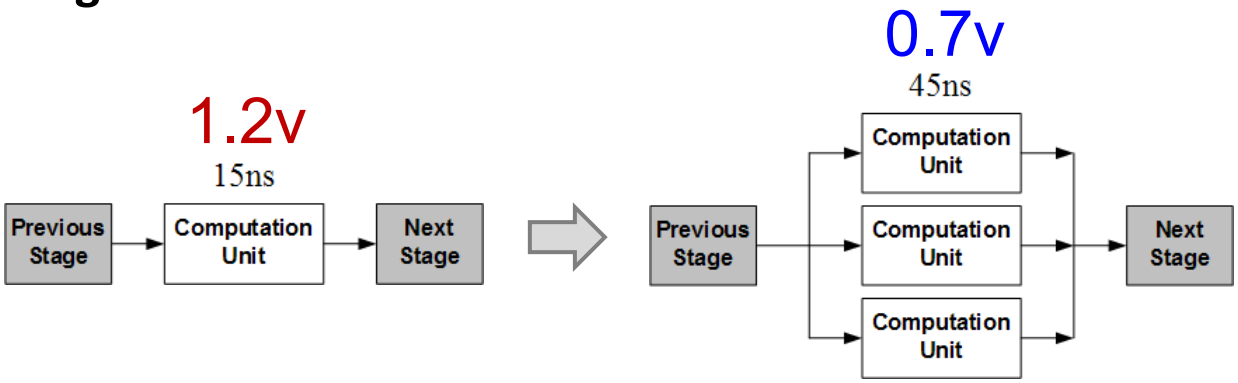
- **Reducing Clock Frequency**
  - Slower clock in power saving mode
  - Lower frequency with voltage scaling
- **Reducing Switching Activity**
  - Avoid unnecessary circuit switching
  - Reducing switching activity at I/O pins
  - Clock gating
- **Note**: these techniques can reduce dynamic power, but in modern technologies, leakage power is also important

Voltage Scaling

- Assume maintaining the same throughput
  - **Pipelining**:

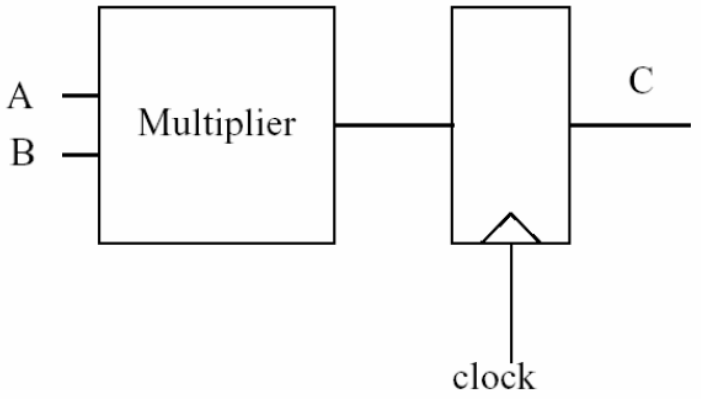


  - **Parallelizing**:



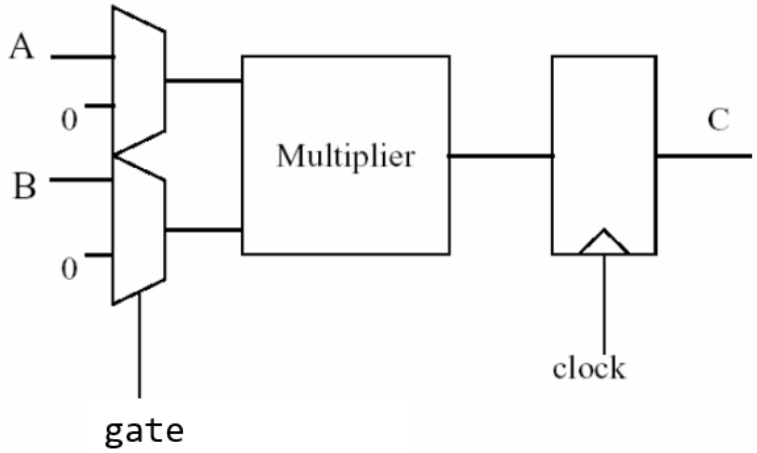
Example: Input Isolation for Multiplier

```
reg [31:0] C;
reg [15:0] A, B;

always @ (posedge clk) begin
    C <= A * B;
end
```



Gated

```
reg  [31:0] C;
reg  [15:0] A, B;
wire [15:0] A_, B_;
assign A_ = gate ? 0 : A;
assign B_ = gate ? 0 : B;

always @ (posedge clk) begin
    C = A_ * B_;
end
```



Computer-Aided VLSI System Design

Chap.4-2 Digital Design Guidelines: From Specification to Circuit

Lecturer: 孫振庭 (Brian)

Graduate Institute of Electronics Engineering, National Taiwan University

NTU GIEE

Outline

- Design Planning
- Design Structure
- Finite State Machines
- More on Debugging
- Tools for Design Planning

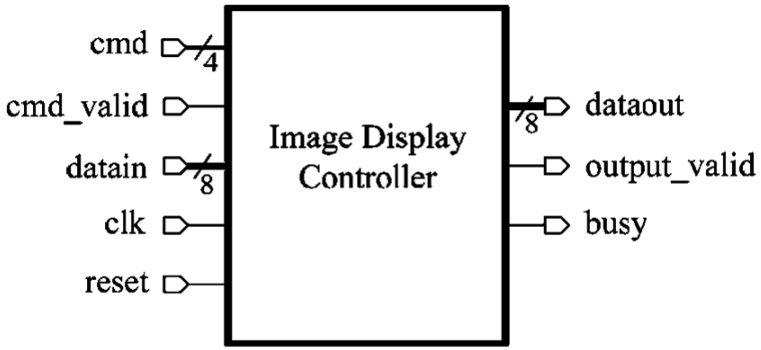
Planning Your Design

1. **Specifications**
   - Module interface
   - Timing diagram
   - Control flow and protocol
2. **Control Flow**
   - Design a finite state machine (FSM)
   - Determine transition conditions
   - Determine state outputs
3. **Datapath**
4. **Others**
   - Reset strategy (sync/async, active high/low)
   - Verify on paper before coding

Datasheet & Timing Diagram

- At the beginning of digital circuit design:
  - I/O specification: bit width, active high/low, input/output, …
  - Timing specification: operating frequency, input/output delay, …

- Example: documenting the signals of your design

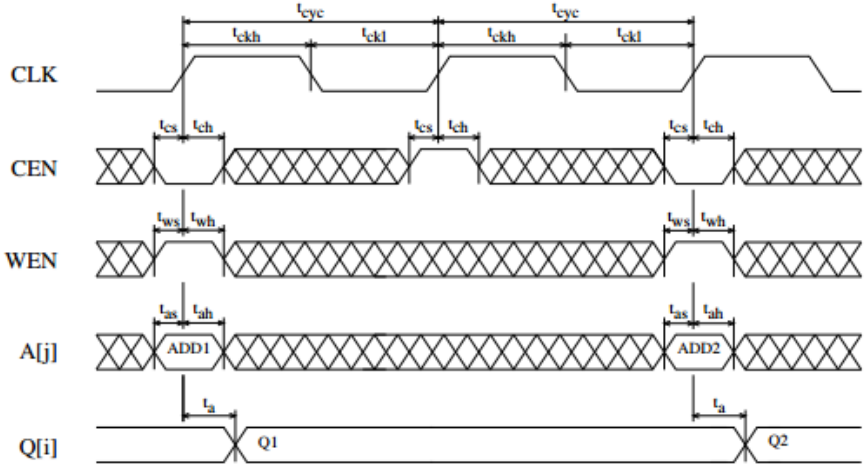


信號名稱	輸/出入	位元寬度	說明
reset	input	1	高位準非同步(active high asynchronous)之系統重置信號。 說明:本信號應於系統啟動時送出。
clk	input	1	時脈信號。 說明:此系統為同步於時脈正緣(posedge)之同步設計。
cmd	input	4	指令輸入信號。 說明:本控制器共有九種指令輸入,相關指令說明請參考表二。指令輸入只有在 cmd_valid 為 high 及 busy 為 low 時,為有效指令。
cmd_valid	input	1	有效指令輸入信號。 說明:當本信號為 high 時表示 cmd 指令為有效指令輸入。
datain	input	8	八位元影像資料輸入埠。
dataout	output	8	八位元影像資料輸出埠。
output_valid	output	1	有效資料輸出信號。 說明:當本信號為 high 時表示 dataout 為有效資料輸出。
busy	output	1	系統忙碌信號。 說明:當本信號為 high 時,表示此控制器正在執行現行(current)指令,而無法接收其他新的指令輸入。

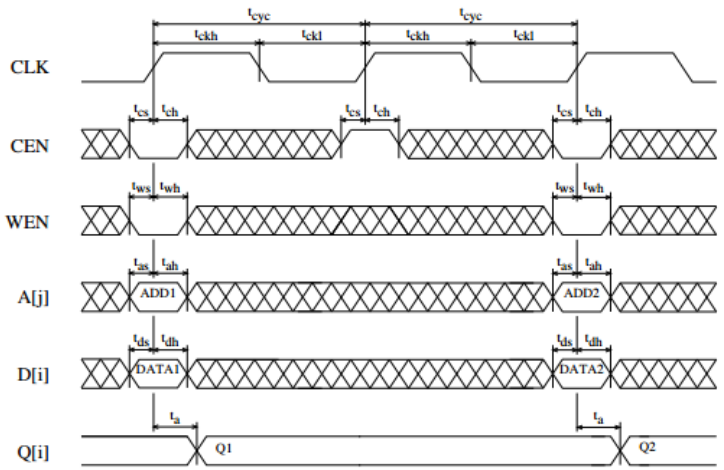
Example: SRAM (1/2)

- Control address for data read
- Control address, input data, write enable for data write

Read timing



Write timing

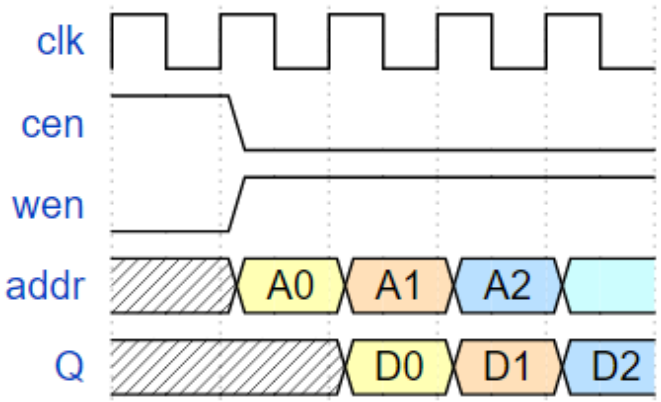


Example: SRAM (2/2)

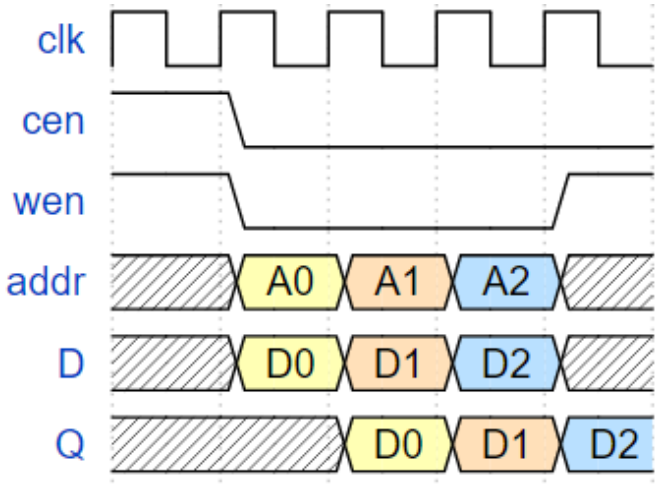
- Control address for data read
- Control address, input data, write enable for data write

- Draw timing diagrams for sequential circuits based on specs

Read timing



Write timing



Reset Strategy

- Reset is a global signal distributed across the entire chip
- Initialize the chip to the idle state
- Be aware of reset behavior
  - Timing: synchronous/asynchronous
  - Trigger: active low/active high

```
always @ (posedge clk or negedge rst_n) begin
    if (~rst_n) begin
          // Reset assignment
    end
    else begin
          // Next state assignment
    end
end
```

Asynchronous active-low reset

```
always @ (posedge clk) begin
    if (rst) begin
          // Reset assignment
    end
    else begin
          // Next state assignment
    end
end
```

Synchronous active-high reset

Outline

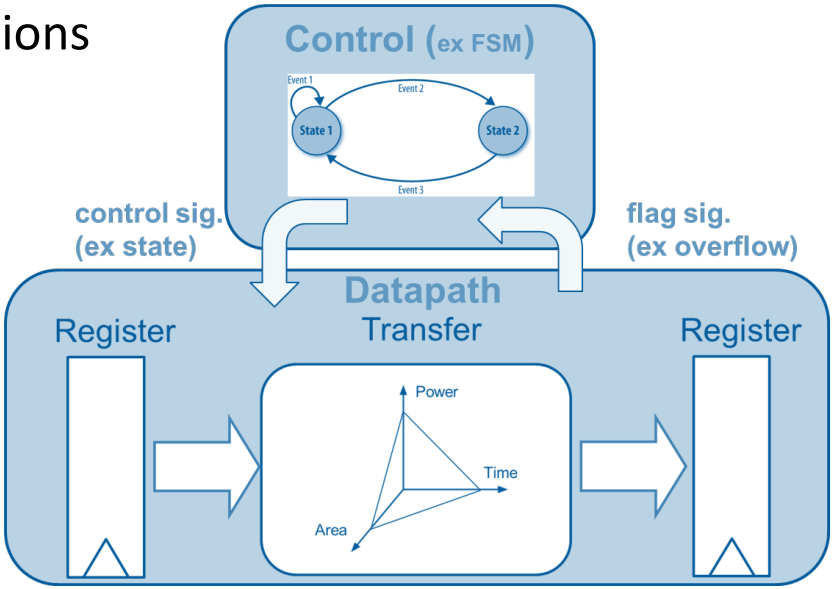
- Design Planning
- Design Structure
- Finite State Machines
- More on Debugging
- Tools for Design Planning

Controller & Datapath (1/2)

- Separate controller and datapath
- **Controller**
  - Controls the operations of datapath modules
  - e.g.: **FSM**, counter
- **Datapath**
  - Performs computation with the input data
  - e.g. arithmetic & logic units, data registers, MUX

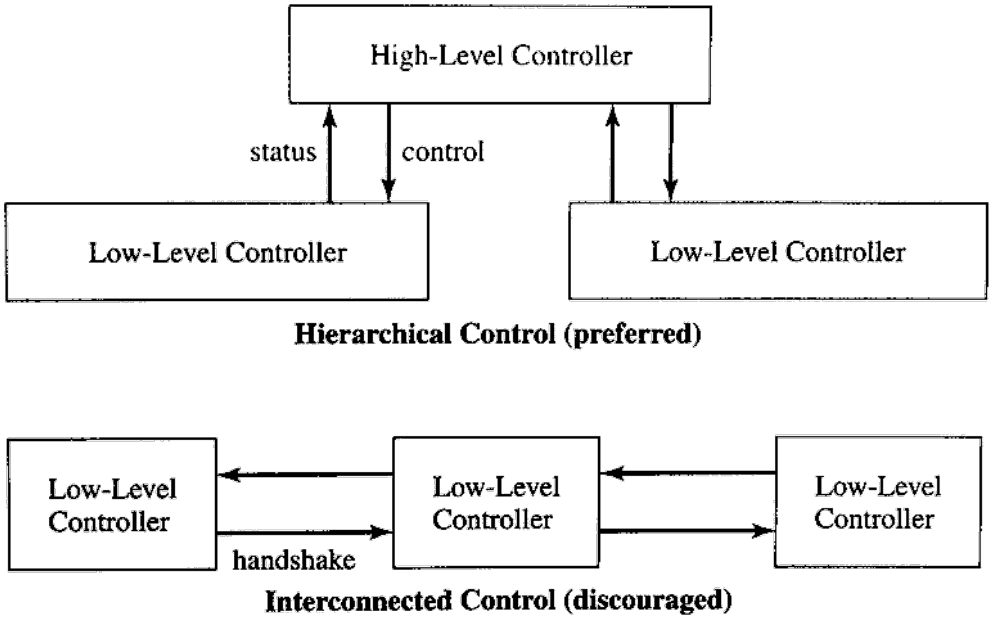
Controller & Datapath (2/2)

- **Control signals**
  - Determine the detailed operations to be performed on the datapath
  - e.g.: FSM states, start, select signals
- **Status signals**
  - Indicate the status of datapath modules
  - e.g. flags, finish, overflow, execptions



Control Strategy

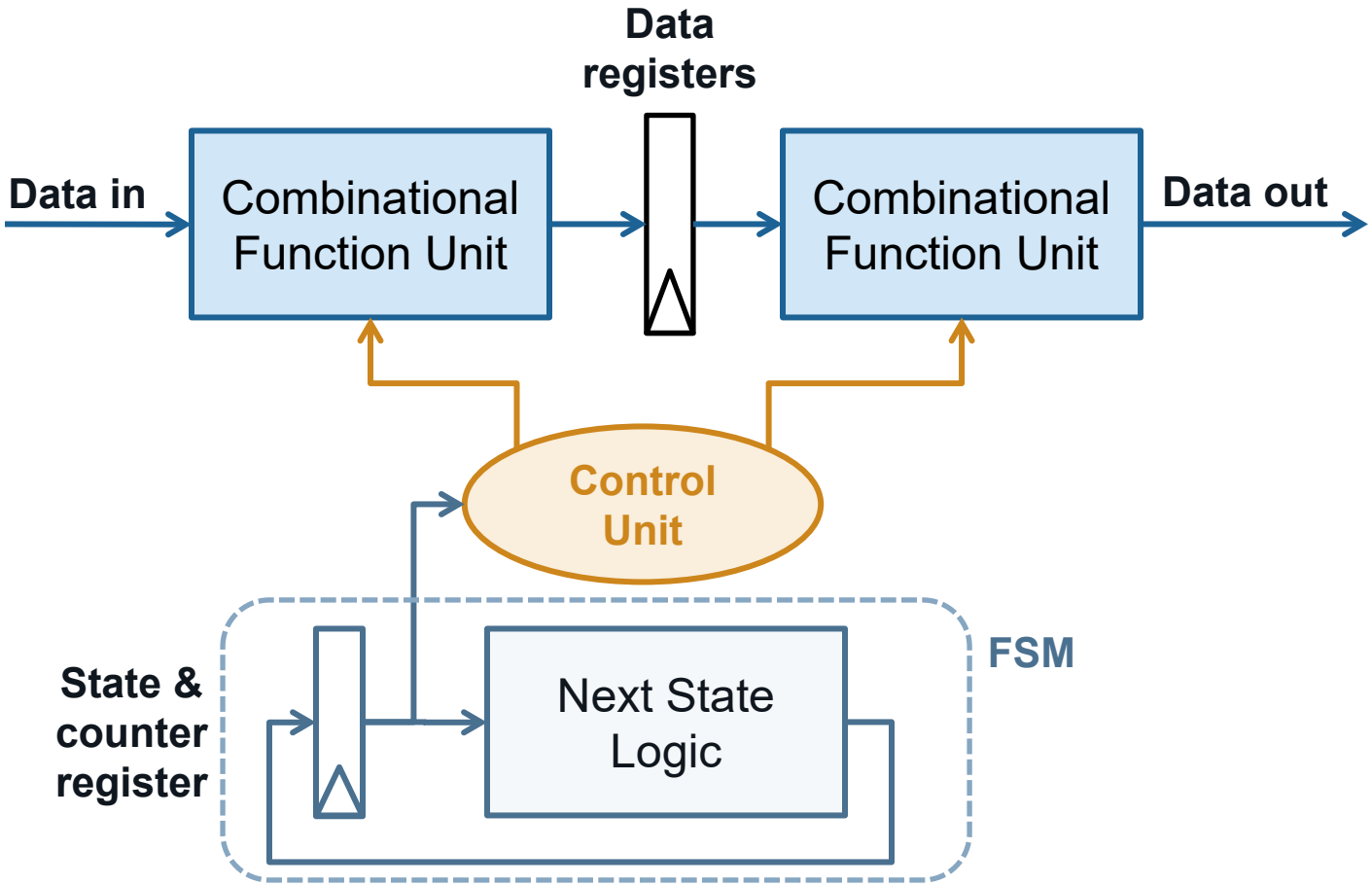
- **Make sure the behavior of control signals are correct**
- Generally, control sequence are generated by one or a combination of the following strategies:
  - **Finite state machines (FSMs)**
  - Top-down controllers
  - Counters
  - Software



Structuring the Datapath

- Begin with block diagrams
- Determine the functional units and their connection
  - Low-level instances (e.g. arithmetic, logic)
  - Memory instances (SRAM)
  - Submodules
- Determine the design strategy
  - Pipelining stages
  - Parallelization degree

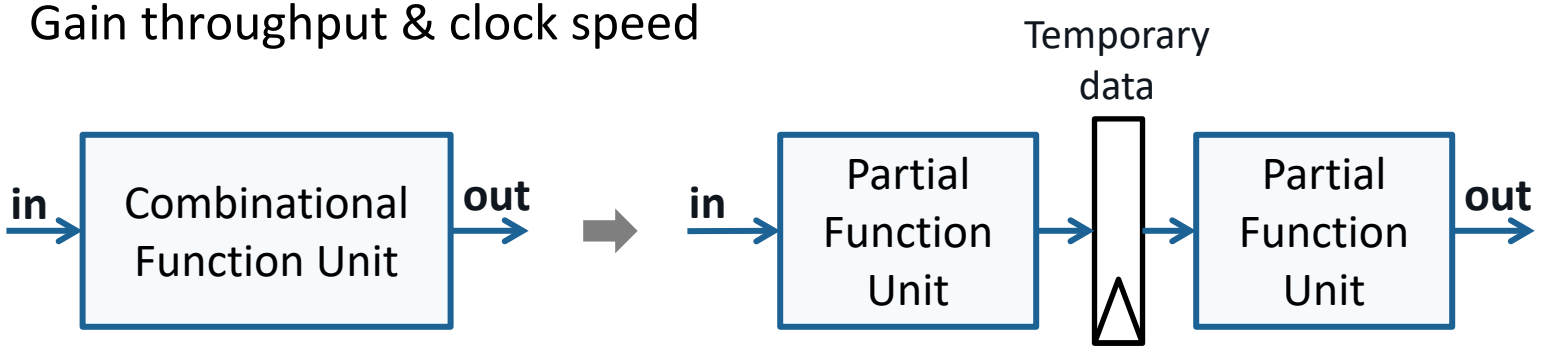
Datapath Design (1/3)



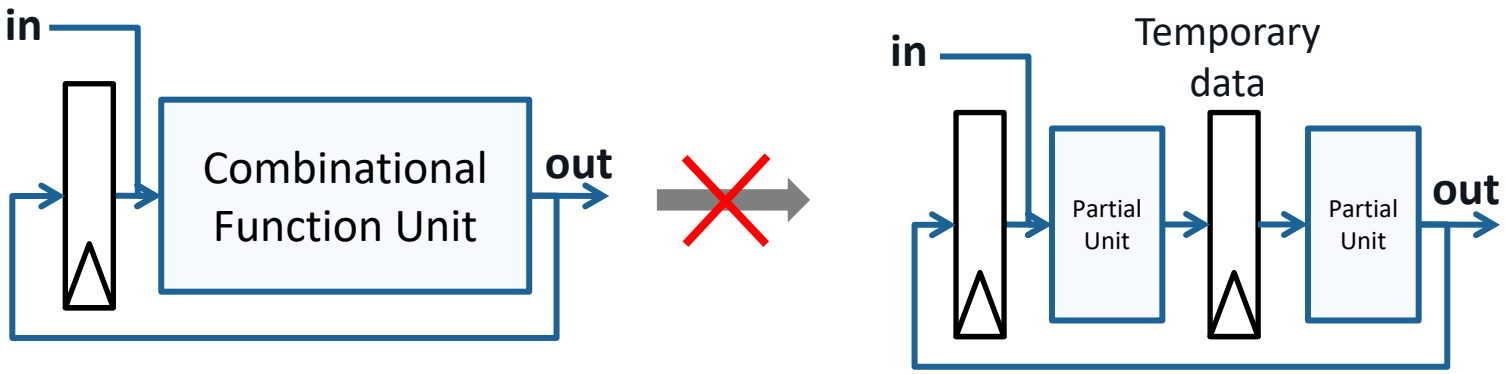
Datapath Design (2/3)

Pipeline

- Gain throughput & clock speed



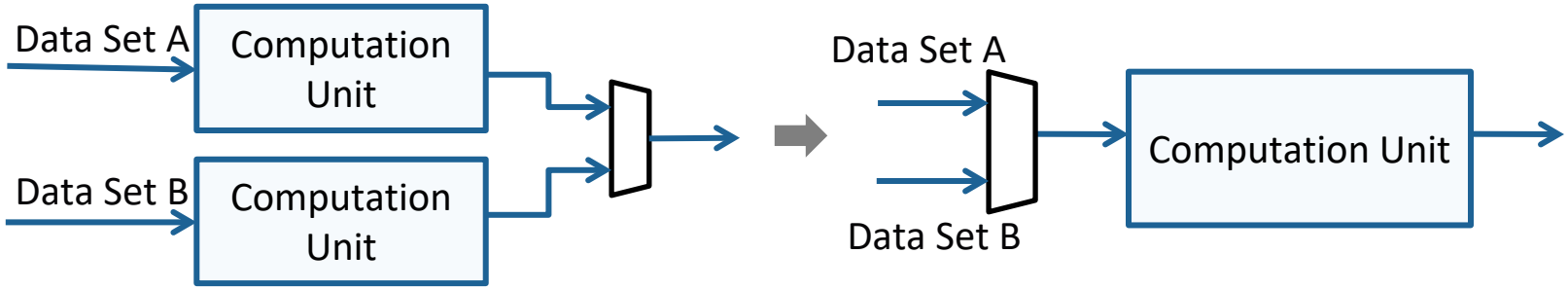
- Feedback loops cannot be pipelined



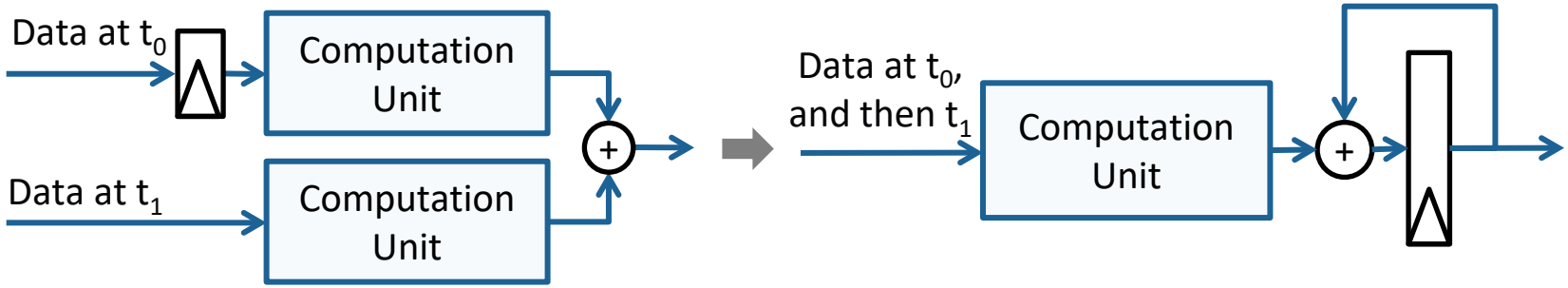
Datapath Design (3/3)

Functional block reusing

- Reduce combinational area

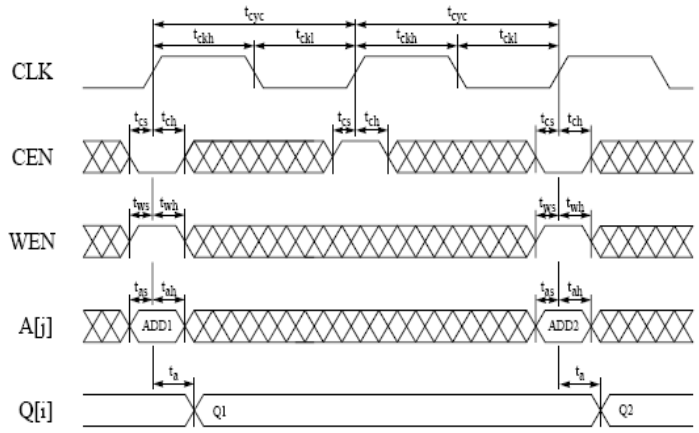


- Reduce area when input comes sequentially

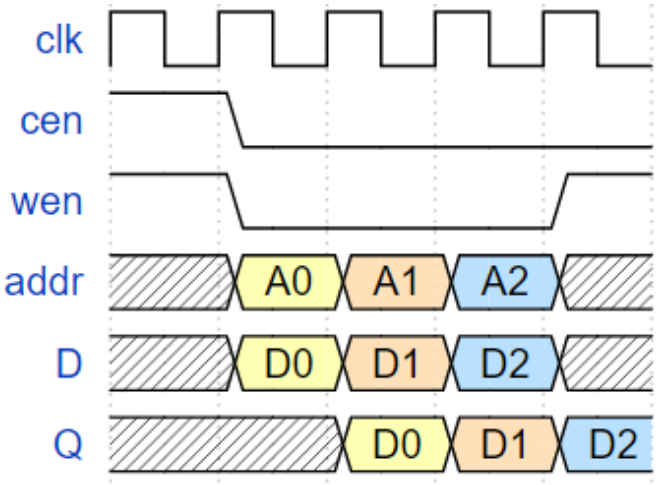


Note: Verify before Coding

- It is a useful practice to hand-simulate the design before coding
- Draw a timing diagram capturing critical events (e.g. state transitions and signal changes)



specification



timing in your design

Conclusion: Partition for Synthesis

- Separate combinational and sequential logic (Chap. 3)
- Separate control and datapath (Chap. 4-2)
- Register at hierarchical output (Chap. 3)

- Keep major blocks separated (draw block diagrams)
- Avoid glue logic (logic between module connections)
- Avoid aysynchronous logic, false path, and multi-cycle path

Outline

- Design Planning
- Design Structure
- Finite State Machines
- More on Debugging
- Tools for Design Planning

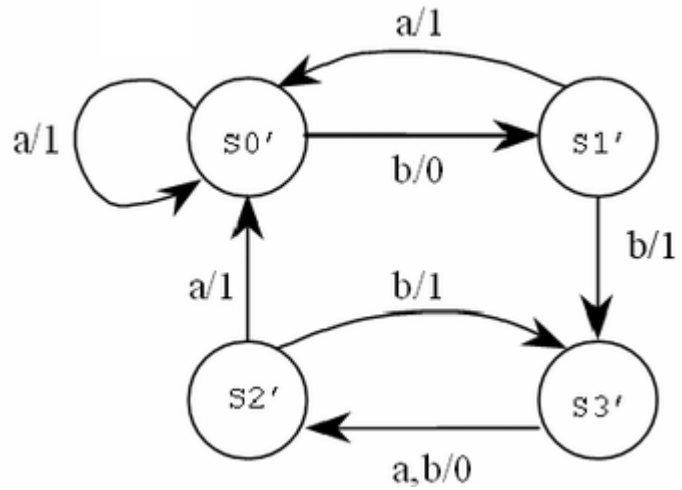
Finite State Machines (FSMs)

- An abstract machine that can be in exactly one of a finite number of states at any given time
  - **Input**: data fed into the machine
  - **Transition**: state change based on current state and input
  - **Output**: data output based on current state and/or input

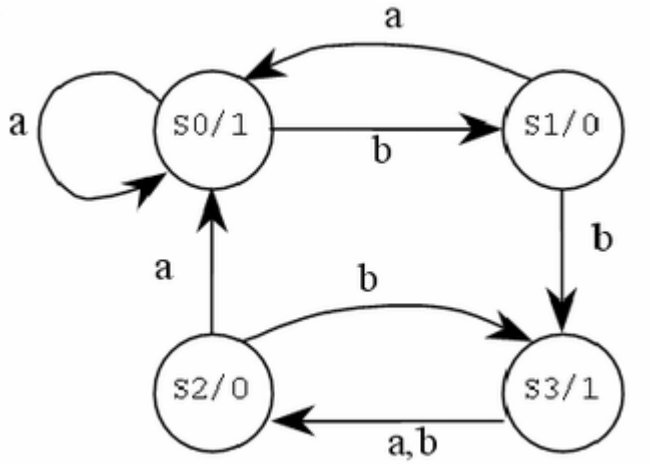
- **Types of FSMs**
  - **Mealy**: output determined by **its current state** and **inputs**
  - **Moore**: output determined only by **its current state**

Mealy and Moore FSM (1/2)

Mealy

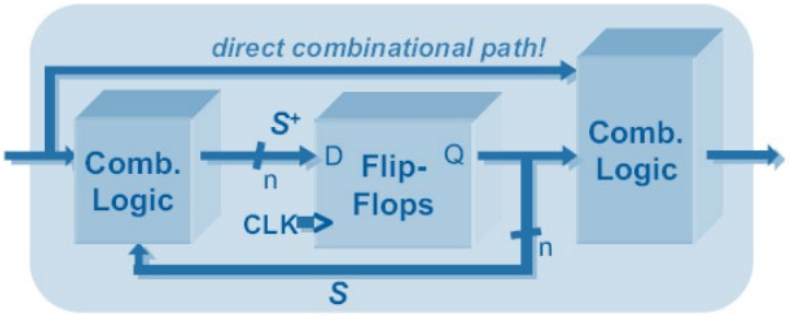


Moore

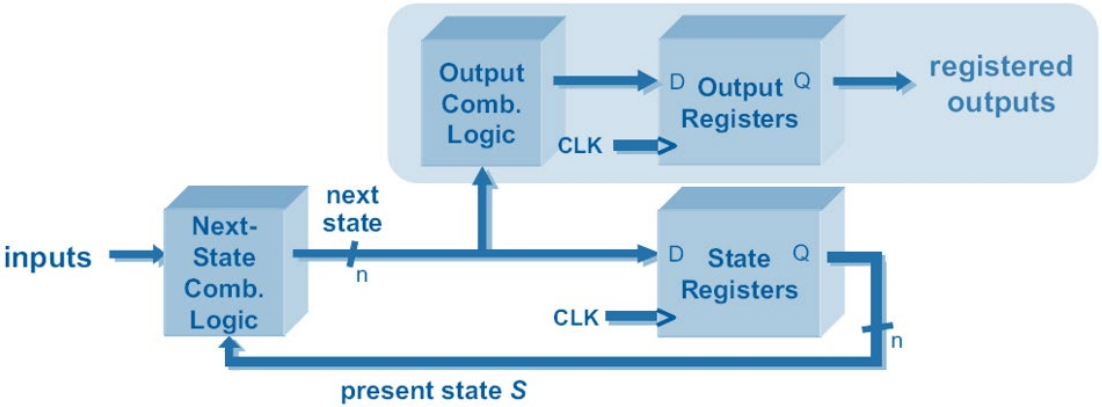


Mealy and Moore FSM (2/2)

- **Mealy** (combinational output)



- **Moore** (sequential output)



Example: FSM in Verilog

```
localparam S_IDLE  = 2'd0;
localparam S_READ  = 2'd1;
localparam S_WRITE = 2'd2;
localparam S_DONE  = 2'd3;

reg [1:0] state_r, state_w;

always @ (*) begin
    state_w = state_r;
    case (state_r)
        S_IDLE:  begin ... end
        S_READ:  begin ... end
        S_WRITE: begin ... end
        S_DONE:  begin ... end
    endcase
end
```

Use `localparam` or `enum` for state constants

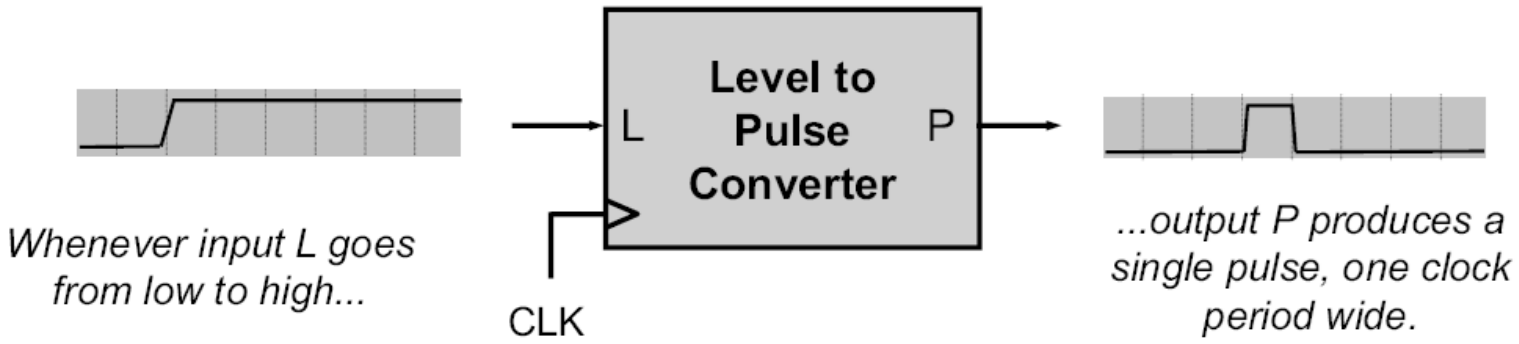
State register

State transitions

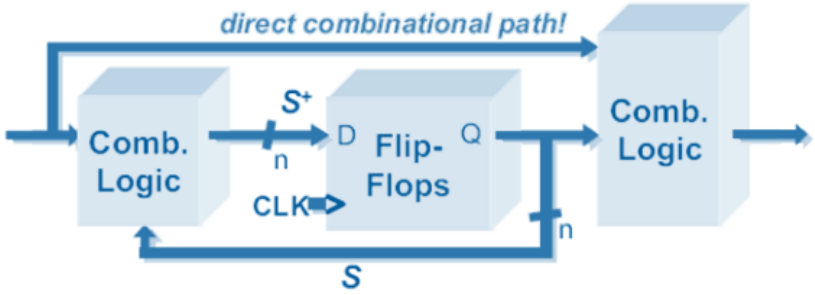
```
always @ (posedge clk or posedge rst) begin
    if (rst) begin
        state_r <= S_IDLE;
    end
    else begin
        state_r <= state_w;
    end
end
```

Example: Level-to-Pulse Converter (1/2)

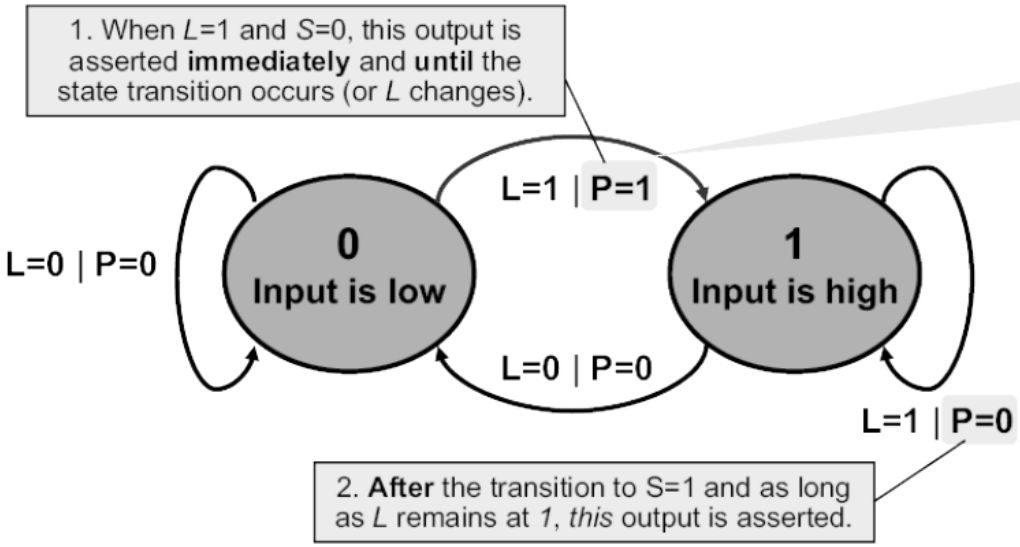
- Produces a single-cycle pulse every time when input goes from low to high
  - e.g. a synchronous rising edge detector
- Applications
  - Button and switches
  - Single-cycle enable signal

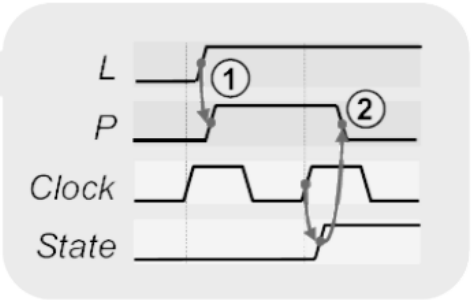


Example: Level-to-Pulse Converter (2/2)



- Since outputs are determined by state *and* inputs, Mealy FSMs may need fewer states than Moore FSM implementations





Output transitions immediately.

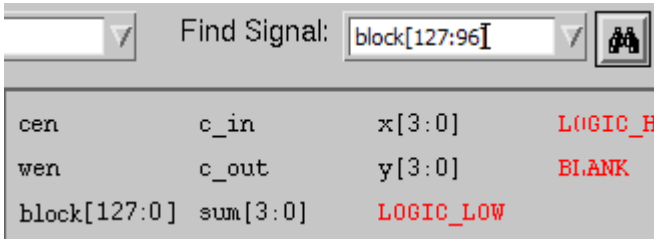
State transitions at the clock edge.

Outline

- Design Planning
- Design Structure
- Finite State Machines
- More on Debugging
- Tools for Design Planning

RTL Debugging

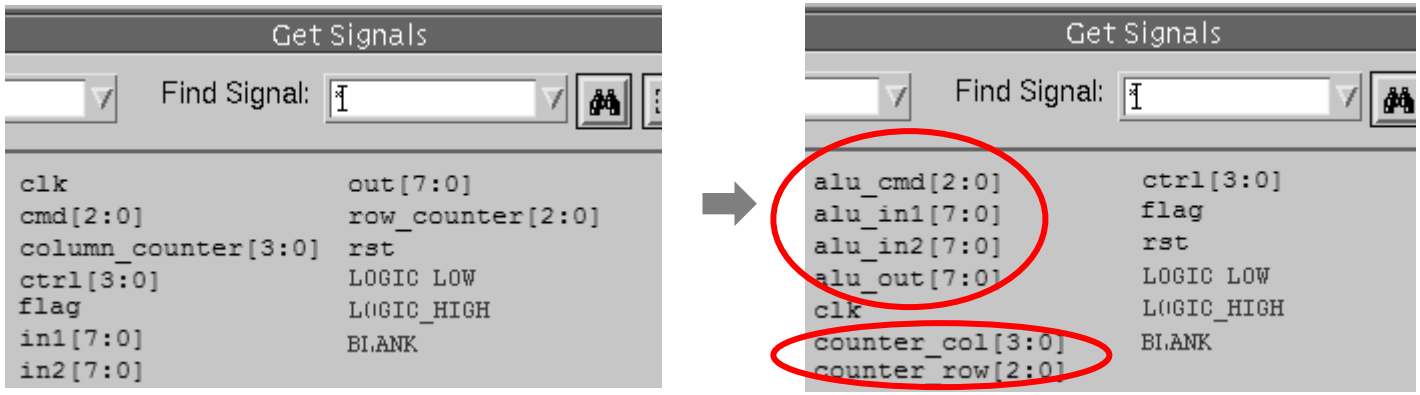
- **Tracing signals**
  - When *S* is incorrect, check what drives *S* first
  - Backward trace until the error source is found
  - Tool: nTrace
- **Check control signals first**
  - FSM state transitions
  - Start/done, valid/ready handshake signals
- **Note**: Partial vector signal in nWave



Naming Conventions for Debugging

- Use suffix in signal naming to show the attributes of signals
  - `signal_n`: active-low
  - `signal_w`: wire
  - `signal_r`: register
  - `signal_next`: next FSM state
  - `signal_cur`: current FSM state

- Alphabetical naming for debugging



Timing Checks in Post-Synthesis Simulation (1/2)

- **SDF annotation**
  - Make sure that the SDF file exists, and the path is correct
  - Make sure that the SDF file is parsed correctly during simuation

```
$sdf_annotate("design.sdf", testbench)
```

- **Input delay and output delay**
  - Synthesis tool does not know when the inputs come, and when the outputs are captured
  - Default: No input and output delay during synthesis
  - Set the constraints properly (Chap. 5)
  - Model input and output delay correctly in testbench

Timing Checks in Post-Synthesis Simulation (2/2)

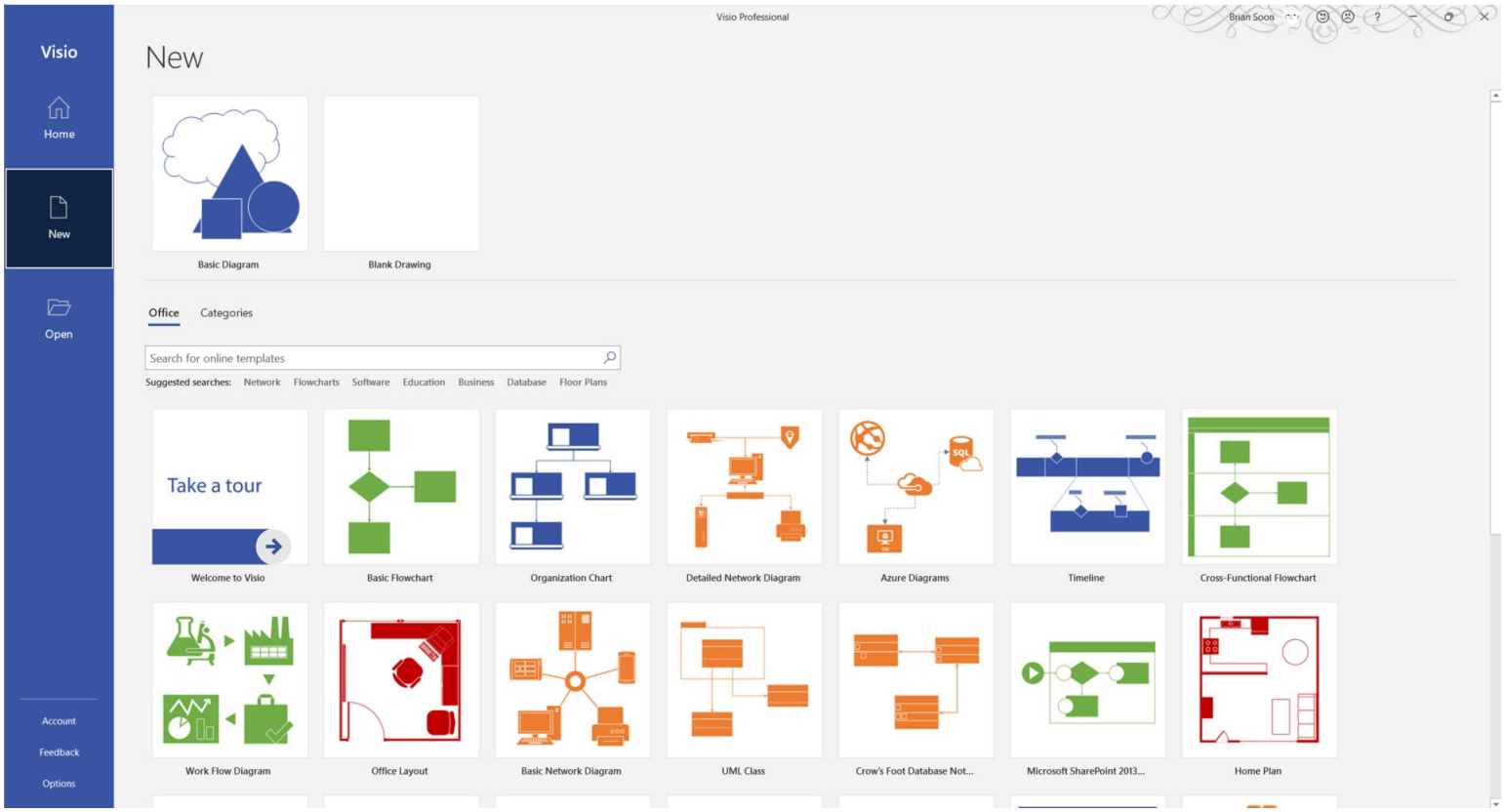
- **Setup-time violation**
  - Input of flip-flop not state during clock trigger
    - Unexpected long combinational delay
    - Combinational loop
    - Improper input delay constraints
- **Hold-time violation**
  - Should be fixed in place-and-route stage (insert buffers)
- Also check if the timing violations are caused by the reset signal, which can be fixed by using correct reset timing

Outline

- Design Planning
- Design Structure
- Finite State Machines
- More on Debugging
- Tools for Design Planning

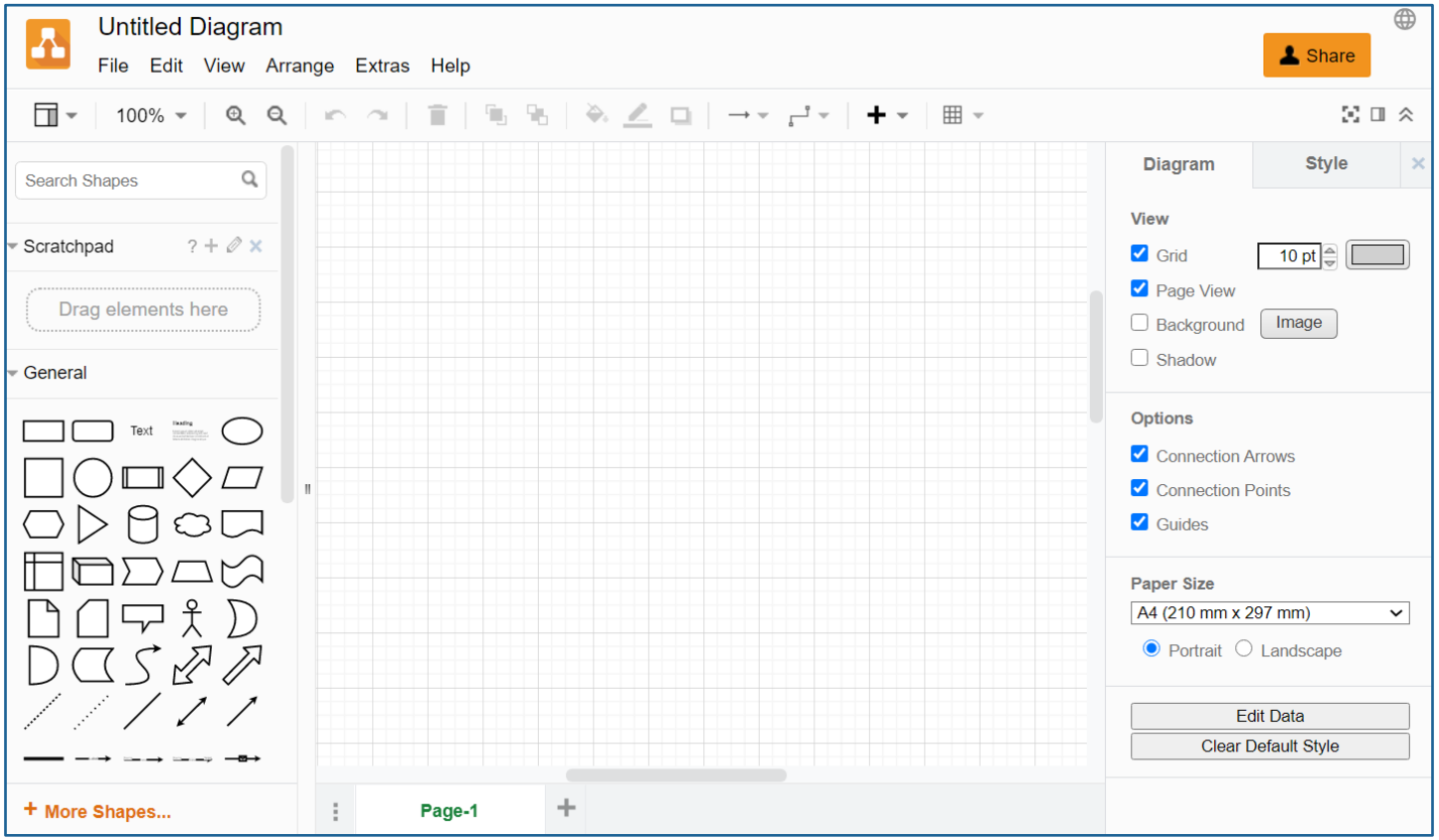
Block Diagram: Microsoft Visio

- A powerful tool for block diagram and flowchart



Block Diagram: draw.io

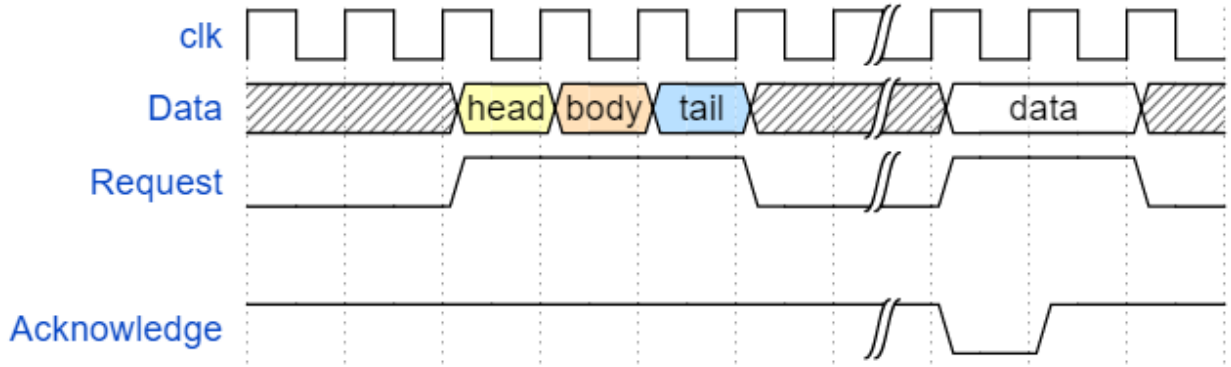
- An open-source diagram software
- Website: https://app.diagrams.net/



Timing Diagram: WaveDrom

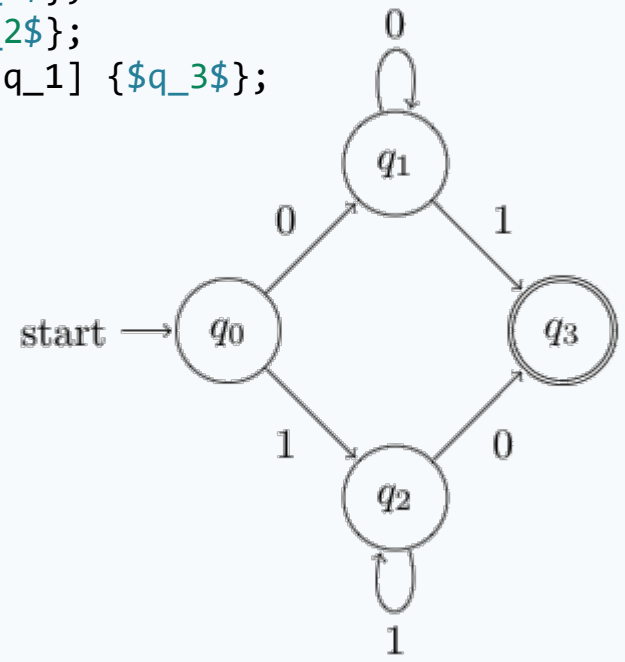
- An open-source timing diagram software
- Website: https://wavedrom.com/

```
{ signal: [
  { name: "clk",          wave: "p.....|..." },
  { name: "Data",         wave: "x.345x|=.x",
                         data: ["head", "body", "tail", "data"] },
  { name: "Request",      wave: "0.1..0|1.0" },
  {},
  { name: "Acknowledge", wave: "1.....|01." }
]}
```

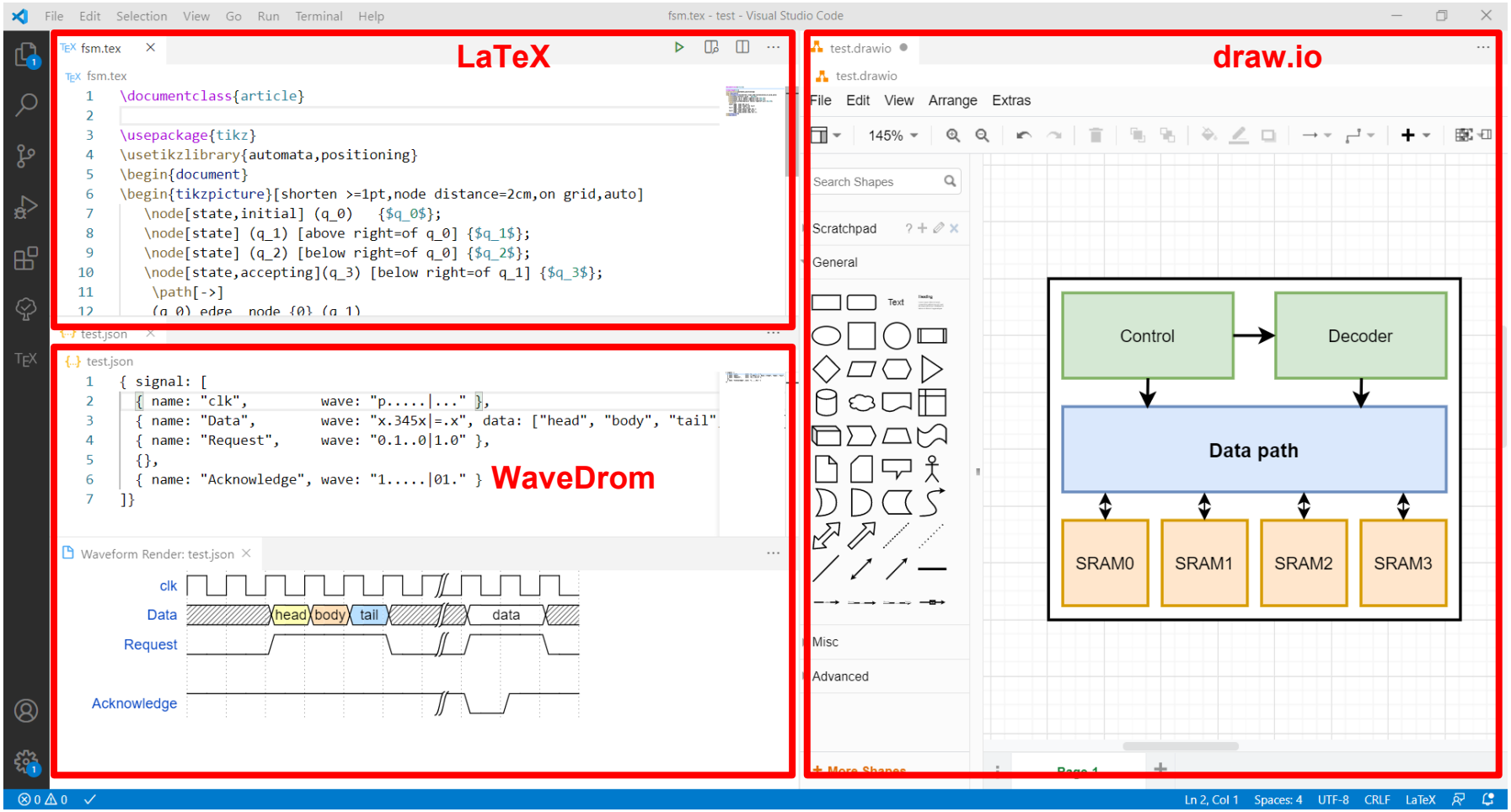


FSM: LaTeX + TikZ

```
\documentclass{article}

\usepackage{tikz}
\usetikzlibrary{automata,positioning}
\begin{document}
\begin{tikzpicture}[shorten >=1pt,node distance=2cm,on grid,auto]
   \node[state,initial] (q_0)   {$q_0$};
   \node[state] (q_1) [above right=of q_0] {$q_1$};
   \node[state] (q_2) [below right=of q_0] {$q_2$};
   \node[state,accepting](q_3) [below right=of q_1] {$q_3$};
    \path[->]
    (q_0) edge  node {0} (q_1)
          edge  node [swap] {1} (q_2)
    (q_1) edge  node  {1} (q_3)
          edge [loop above] node {0} ()
    (q_2) edge  node [swap] {0} (q_3)
          edge [loop below] node {1} ();
\end{tikzpicture}
\end{document}
```



Visual Studio Code Integration



Extensions: LaTeX Workshop, Draw.io Integration, Waveform Render