

Computer-Aided VLSI System Design

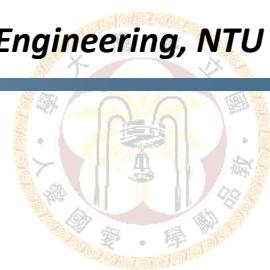
Chapter 4-1. Architecture Improvement of Timing, Area, and Power

Lecturer: Ching-Yen Lee

*Graduate Institute of Electronics Engineering,
National Taiwan University*

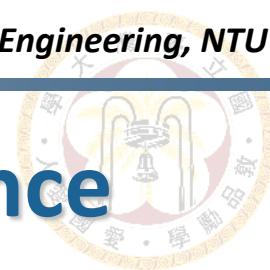


NTUGIEE



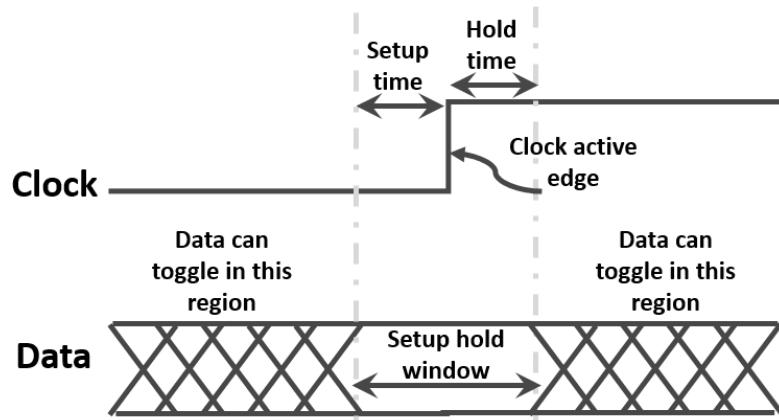
Outline

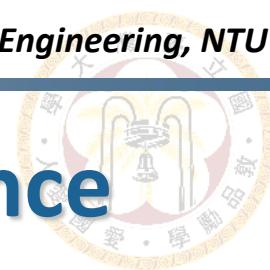
- **Introduction**
- **Register Timing**
- **Timing Improvements**
 - Pipeline
 - Retiming
 - Parallel
- **Area and Power**



Introduction to Hardware Performance

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

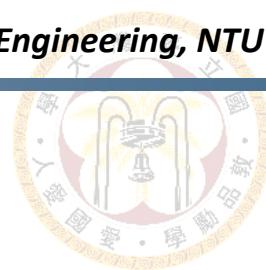




Introduction to Hardware Performance

- **Limitations 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 algorithm and architecture level
 - Cannot rely on synthesis tool all the time
 - Plan and analyze your design for better performance



Latency and Throughput

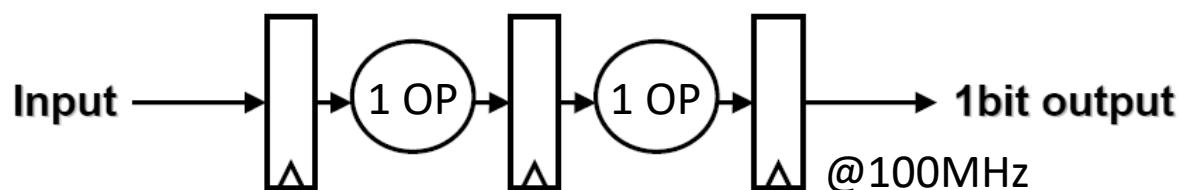
■ Latency

The *time* it takes to complete an operation
(e.g. cycles, seconds)

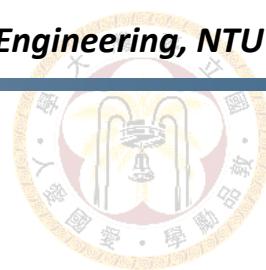
■ Throughput

The *rate* of operations being completed
(e.g., OP/cycle, x-bit width/seconds)

■ Example

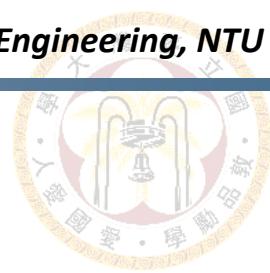


- **Latency:** 3 cycles, or 30ns
- **Throughput:** 2 OP/cycle, 100Mb/s, or 200MOPS



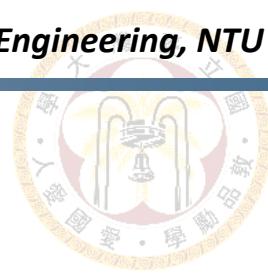
Timing Requirements

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



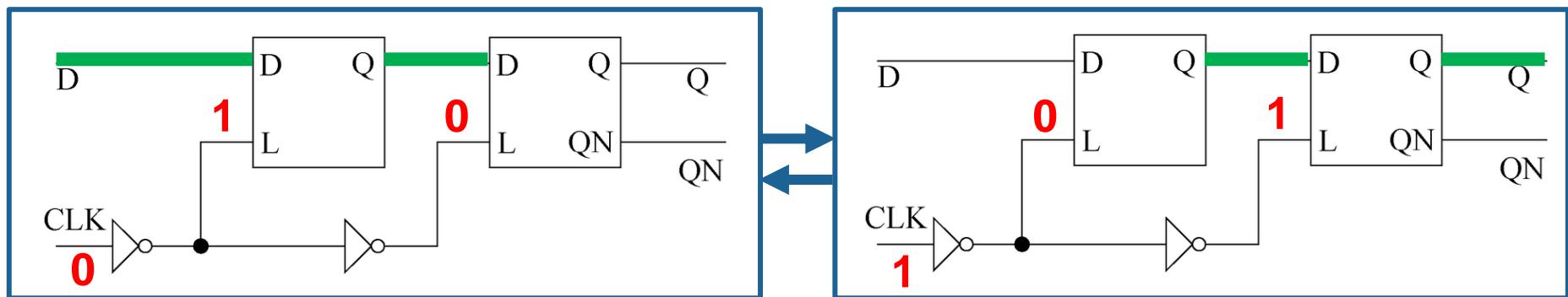
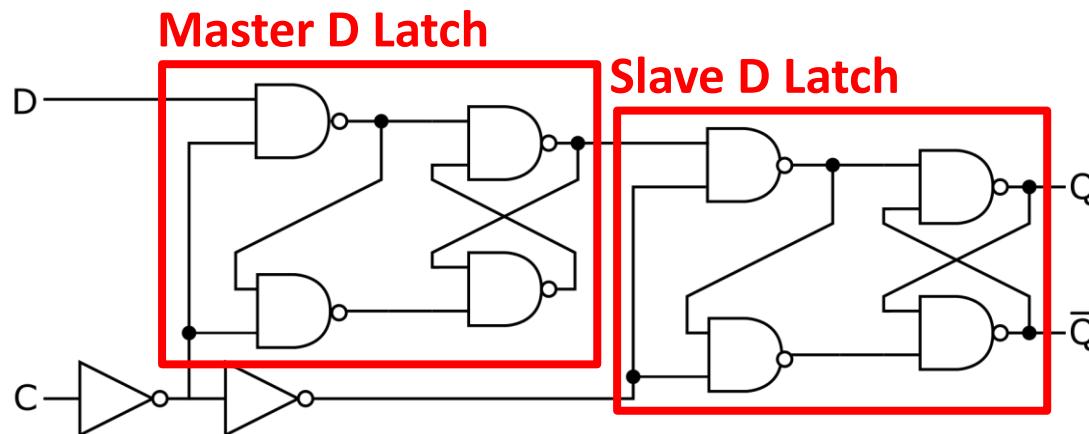
Outline

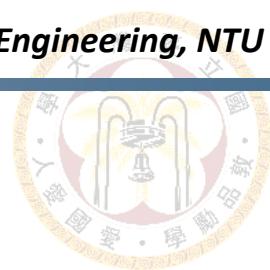
- Introduction
- Register Timing
- Timing Improvements
 - Pipeline
 - Retiming
 - Parallel
- Area and Power



Register Architecture

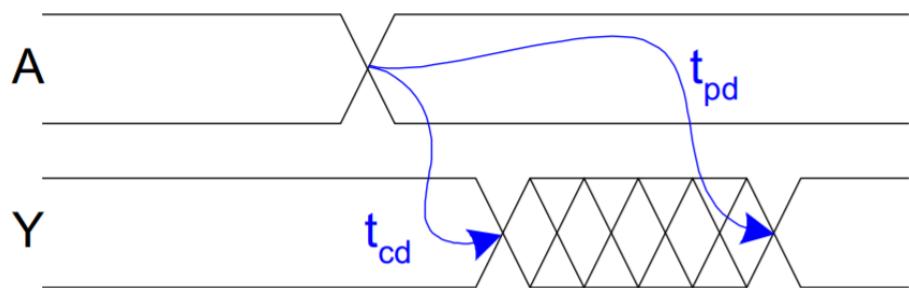
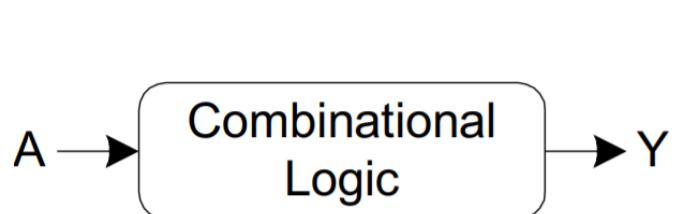
- Positive edge-triggered D flip-flop

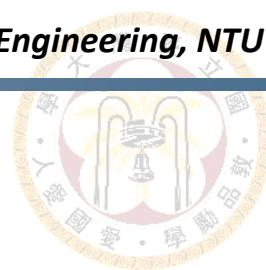




Register Timing: Notation

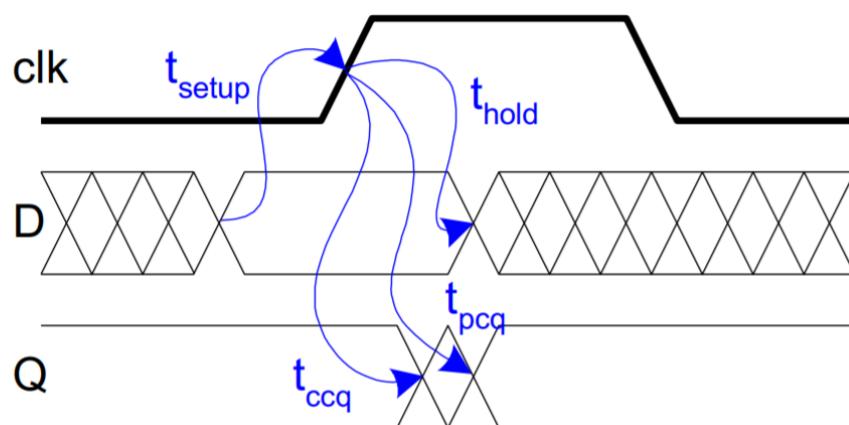
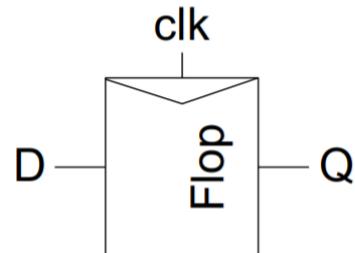
Name	Definition
t_{pd}	Logic propagation delay (maximum logic delay)
t_{cd}	Logic contamination delay (minimum logic delay)

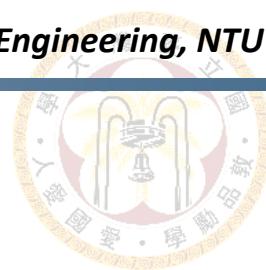




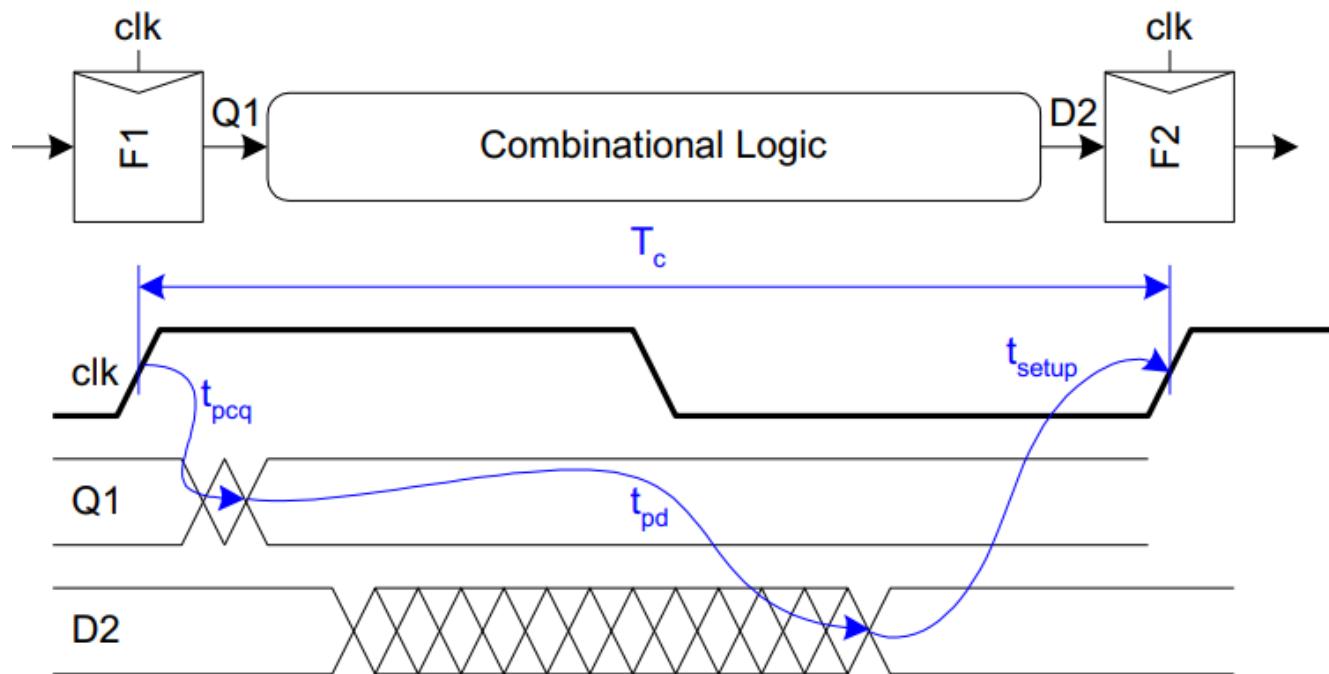
Register Timing: Notation

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





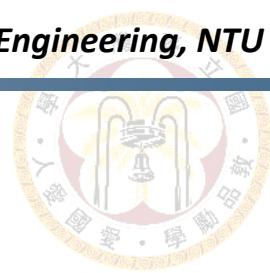
Register Timing: Max Delay



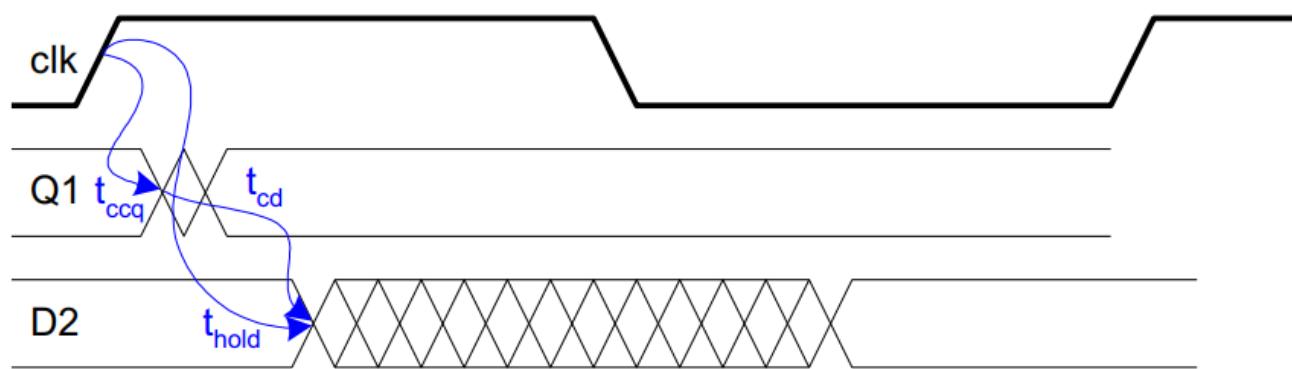
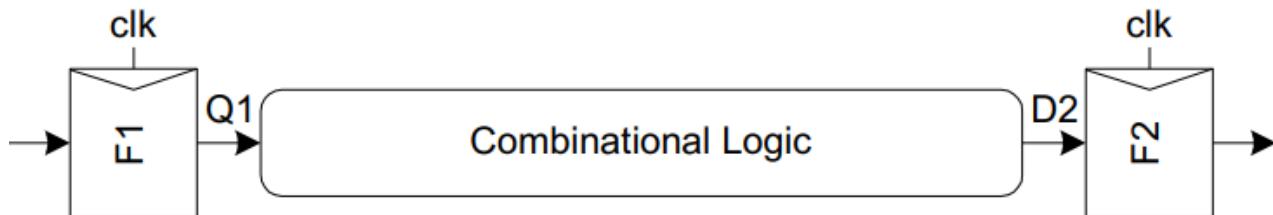
Requirement:

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

Setup time violation:
 $T_c - t_{pd} - t_{pcq} < t_{setup}$



Register Timing: Min Delay

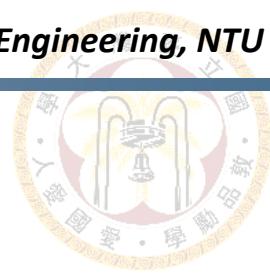


Requirement:

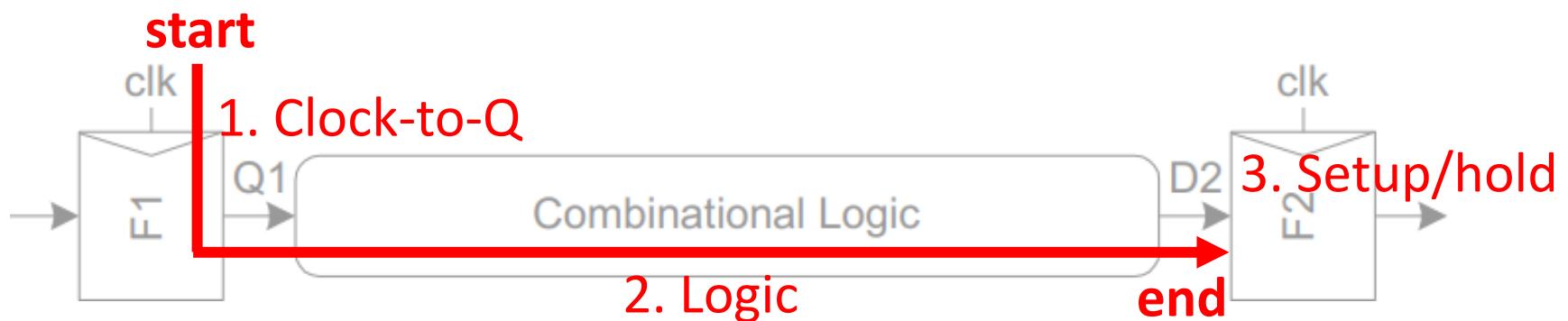
$$t_{cd} \geq t_{hold} - t_{ccq}$$

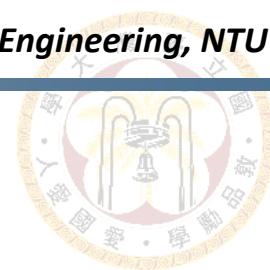
Hold time violation:

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



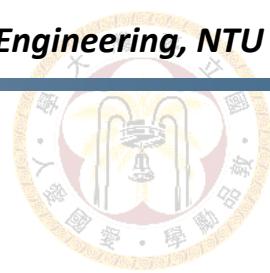
- **Note:** register timing calculation starts from the **clock** signal





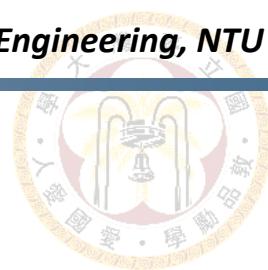
Timing Violations

- **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} (insert buffers)
 - **Note:** hold time violation cannot be fixed by adjusting T_c



Outline

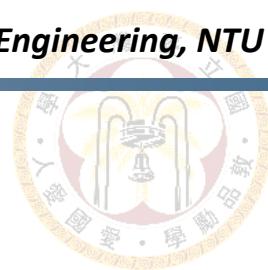
- Introduction
- Register Timing
- **Timing Improvements**
 - Pipeline
 - Retiming
 - Parallel
- Area and Power



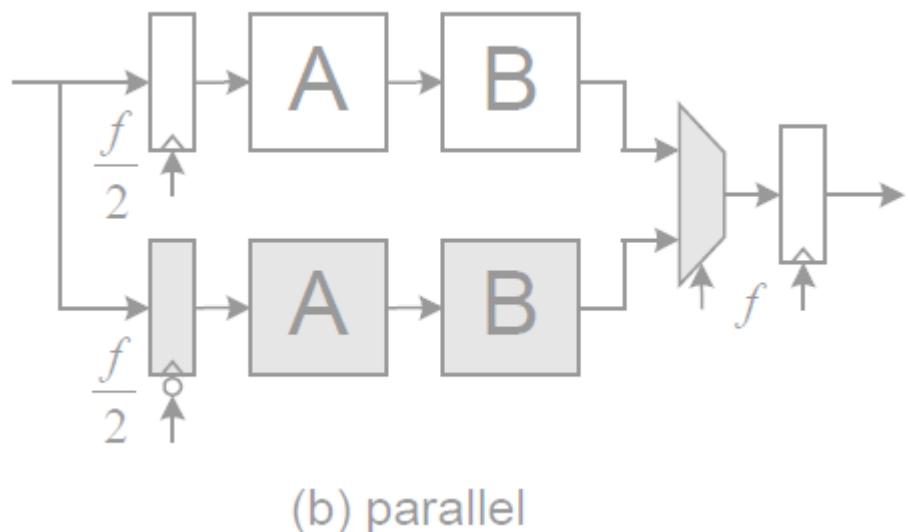
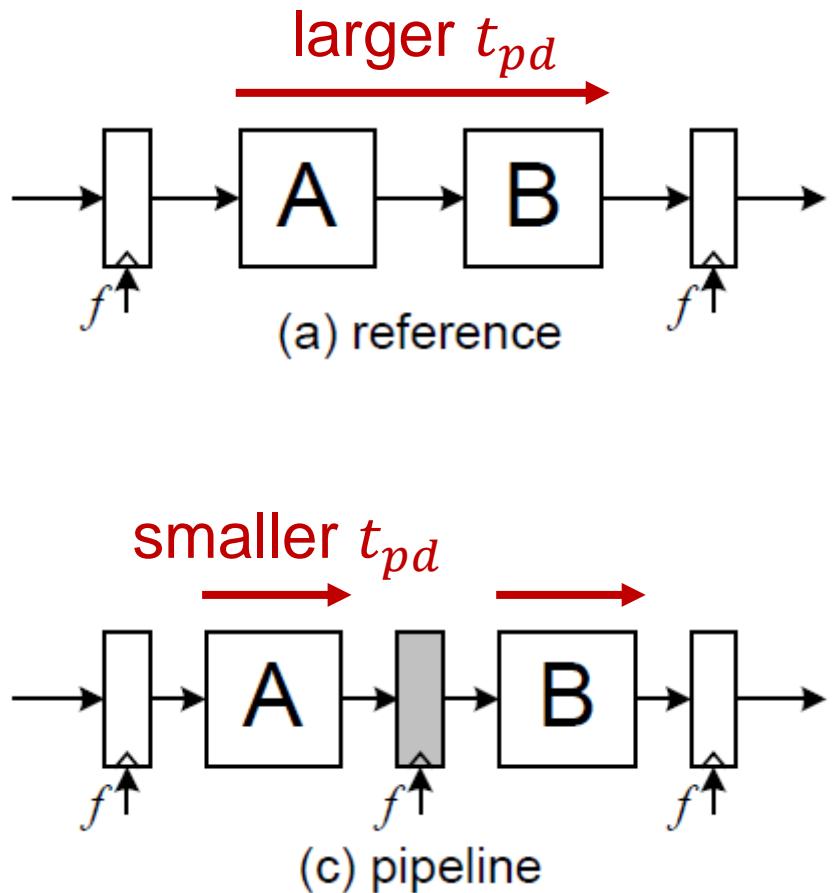
Improve Timing in a Design

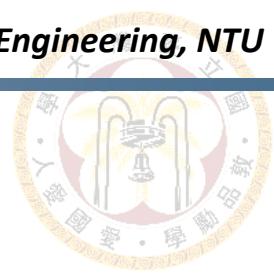
- **Pipelining:** exploits temporal parallelism
 - Insert pipeline registers without changing functionality
 - Shorter critical path -> faster achievable clock
 - Trade off latency (in cycles) to improve throughput

- **Parallelizing:** exploits spatial parallelism
 - Duplicate function units, working in parallel
 - Achieve same throughput using slower clock
(Achieve higher throughput using same clock)
 - Trade off area to improve throughput



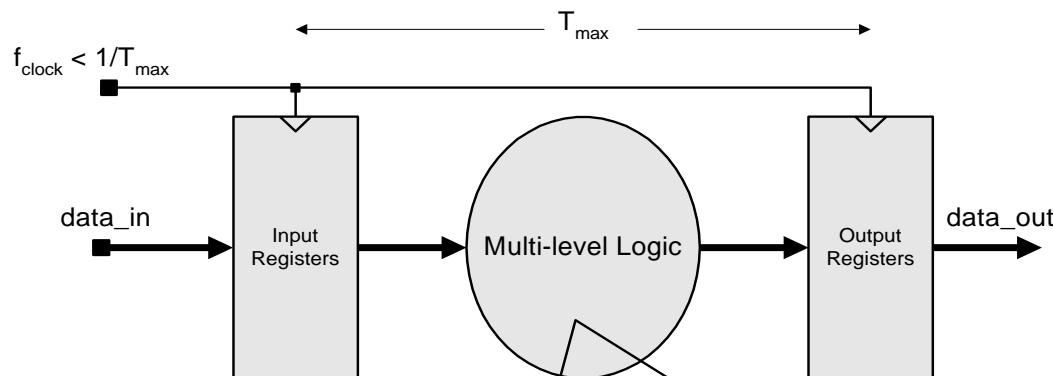
Improve Timing in a Design



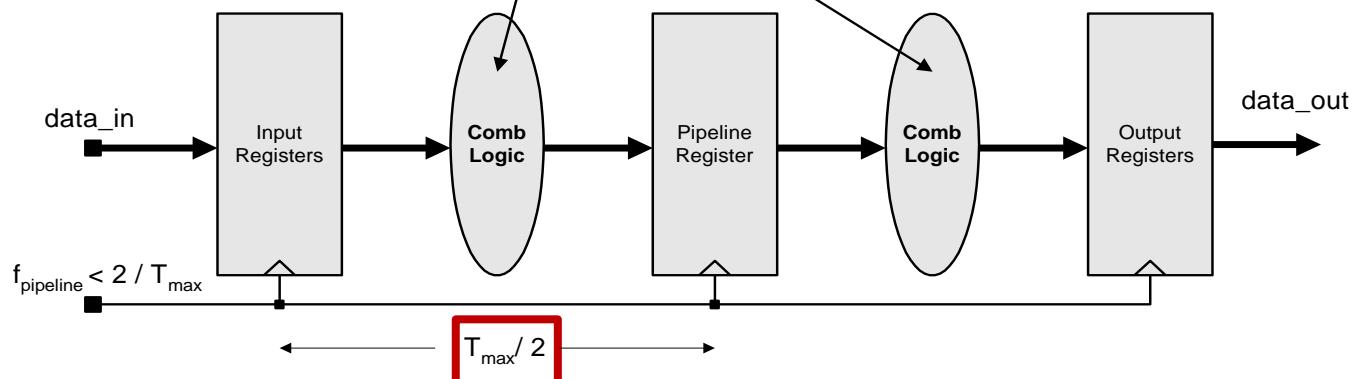


Pipelining

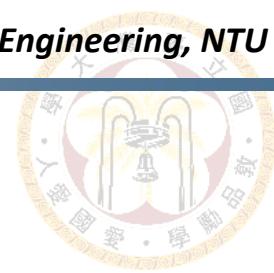
- Original



- 2-stage pipelining

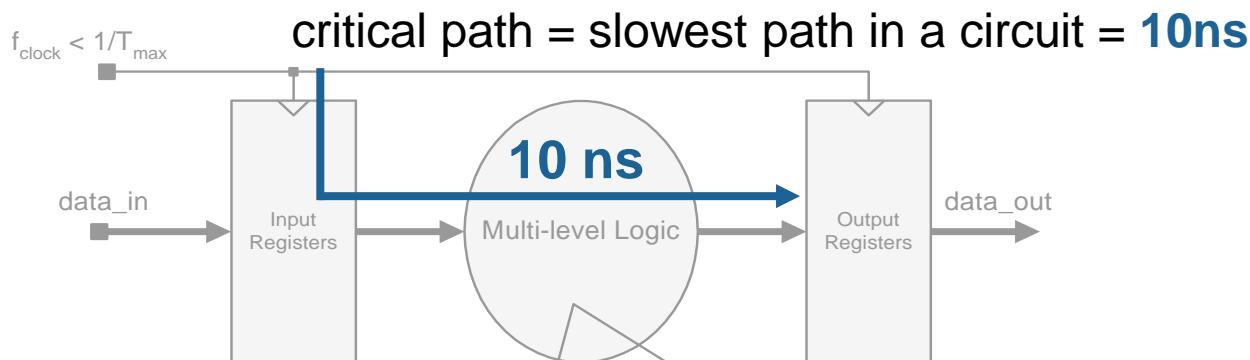


2x faster clock

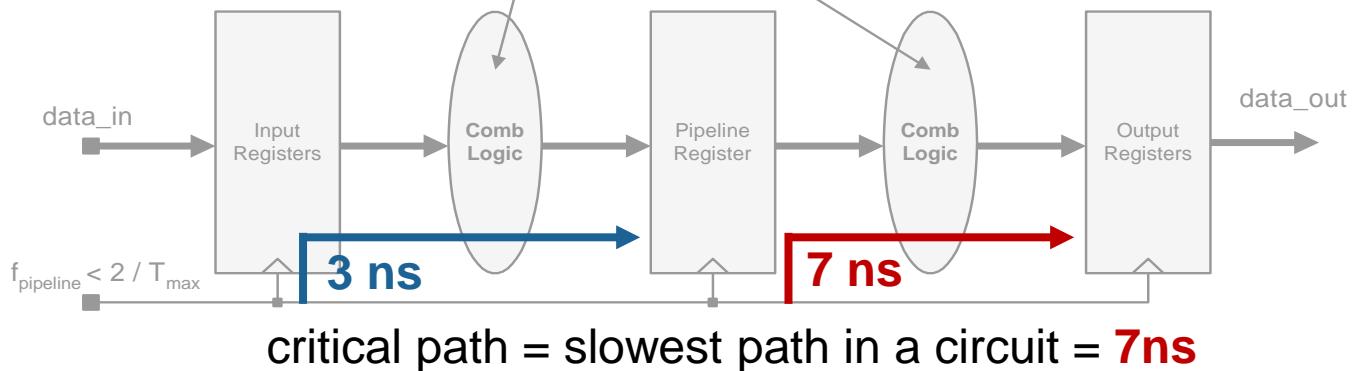


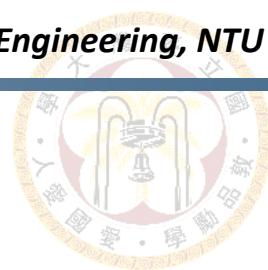
Pipelining

- Original



- 2-stage pipelining





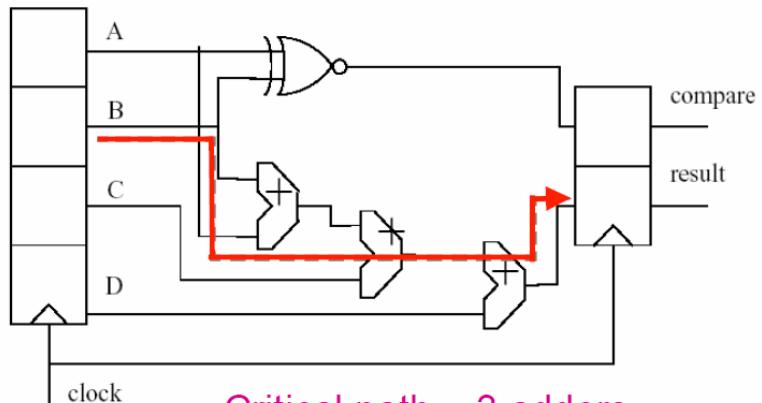
Example: A Simple Circuit

- Original

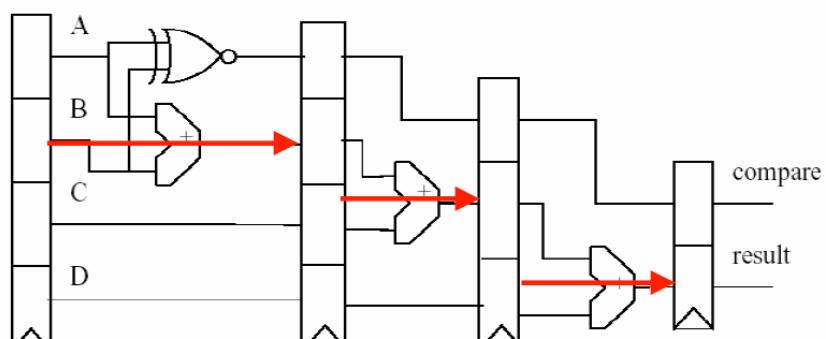
```
always @ (posedge clk) begin
    compare <= A ~^ B;
    result  <= A + B + C + D;
end
```

- 3-stage pipelining

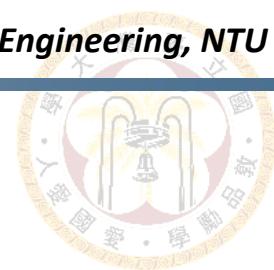
```
always @ (posedge clk) begin
    stage1_1 <= A ~^ B;
    stage1_2 <= A + B;
    stage1_3 <= C;
    stage1_4 <= D;
    stage2_1 <= stage1_1;
    stage2_2 <= stage1_2 + stage1_3;
    stage2_3 <= stage1_3;
    compare  <= stage2_1;
    result   <= stage2_2 + stage2_3;
end
```



Critical path = 3 adders

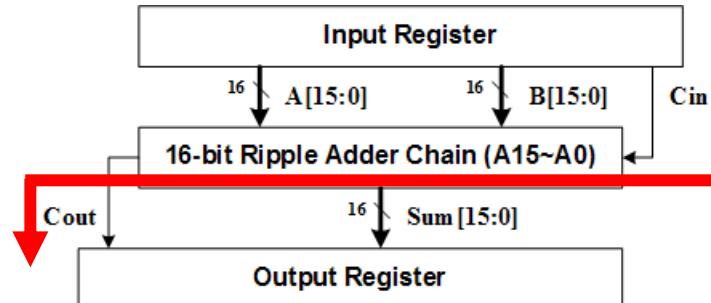


Critical path = 1 adders

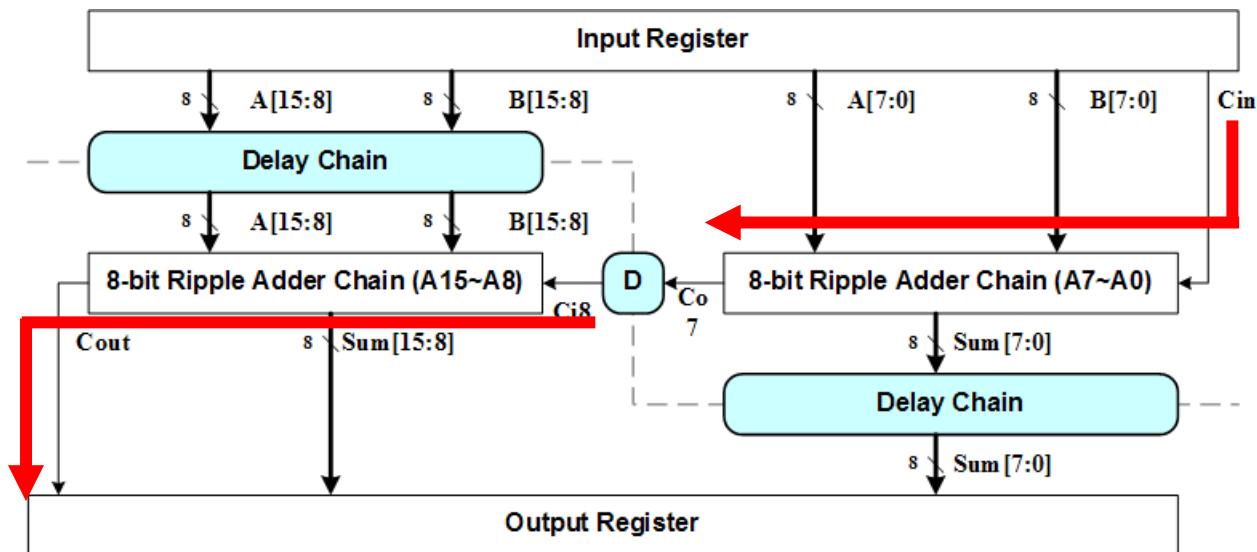


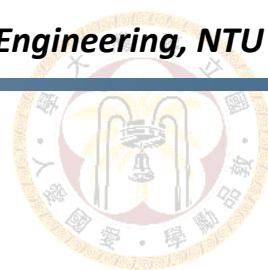
Example: Pipelined 16-bit Adder

- Original



- 2-stage pipelining

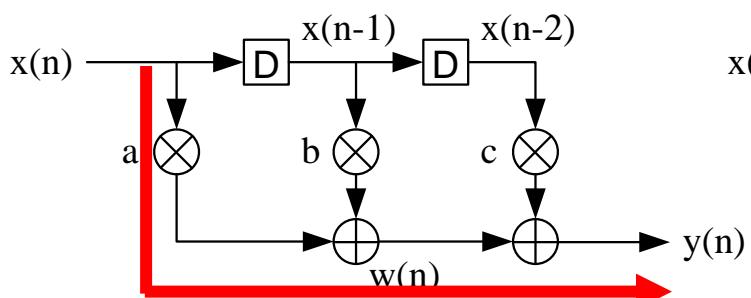




Example: FIR Filter

- Original

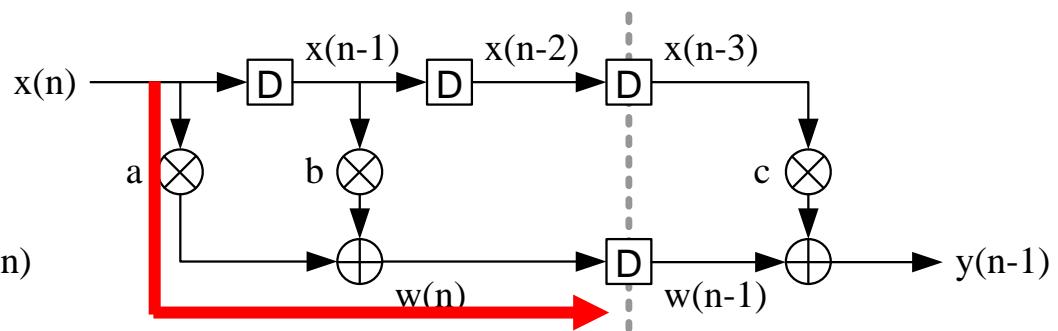
$$\begin{aligned} w(n) &= ax(n) + bx(n-1) \\ y(n) &= cx(n-2) + w(n) \end{aligned}$$



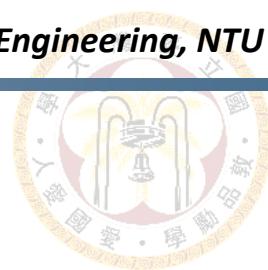
critical path = $T_{MUL} + 2T_{ADD}$

- 2-stage pipelining

$$\begin{aligned} w(n-1) &= ax(n-1) + bx(n-2) \\ y(n-1) &= cx(n-3) + w(n-1) \end{aligned}$$

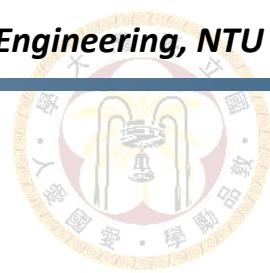


critical path = $T_{MUL} + T_{ADD}$

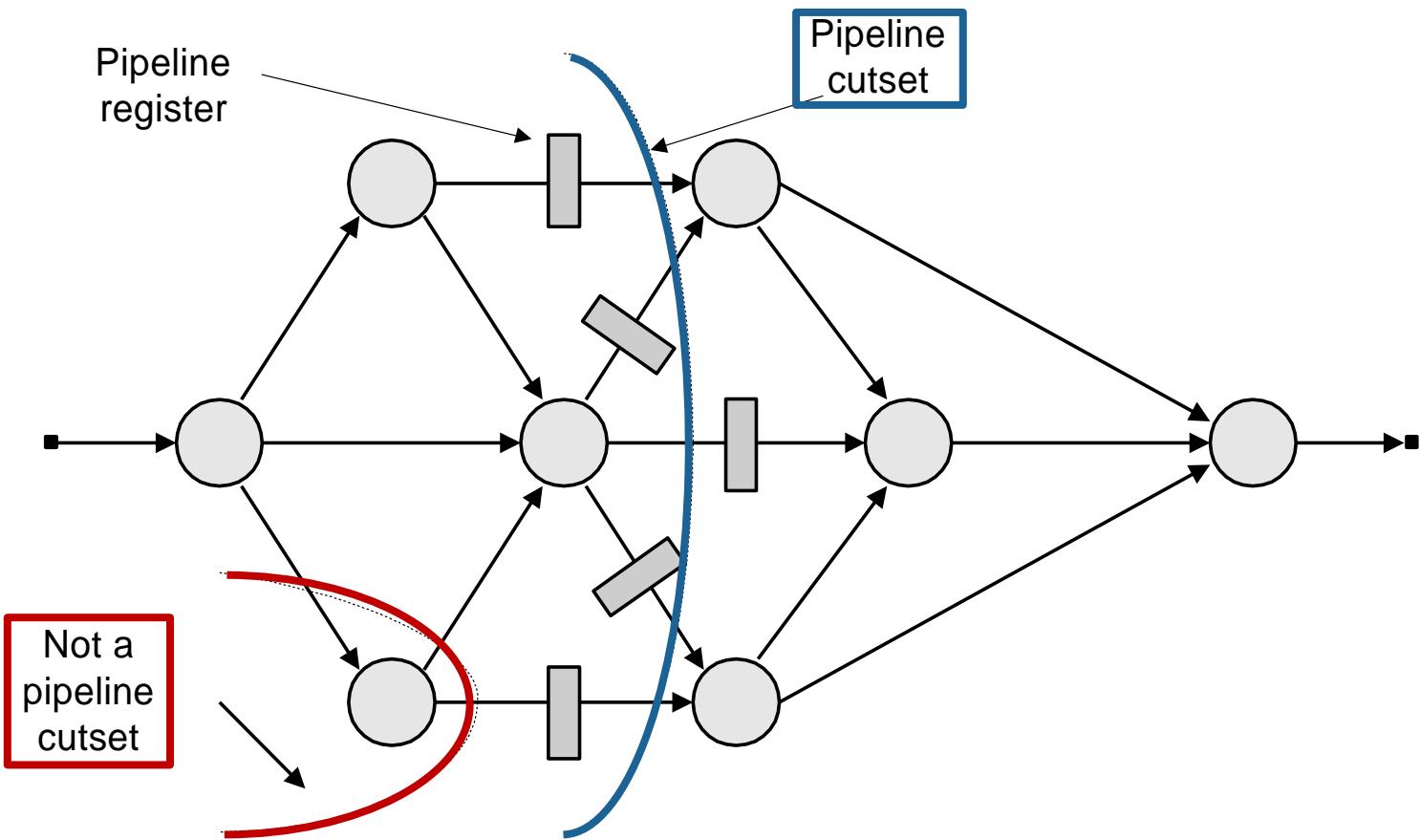


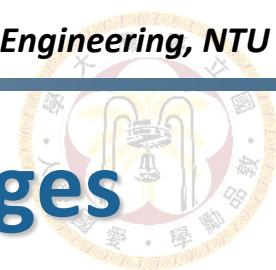
Pipelining a Design

- 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**

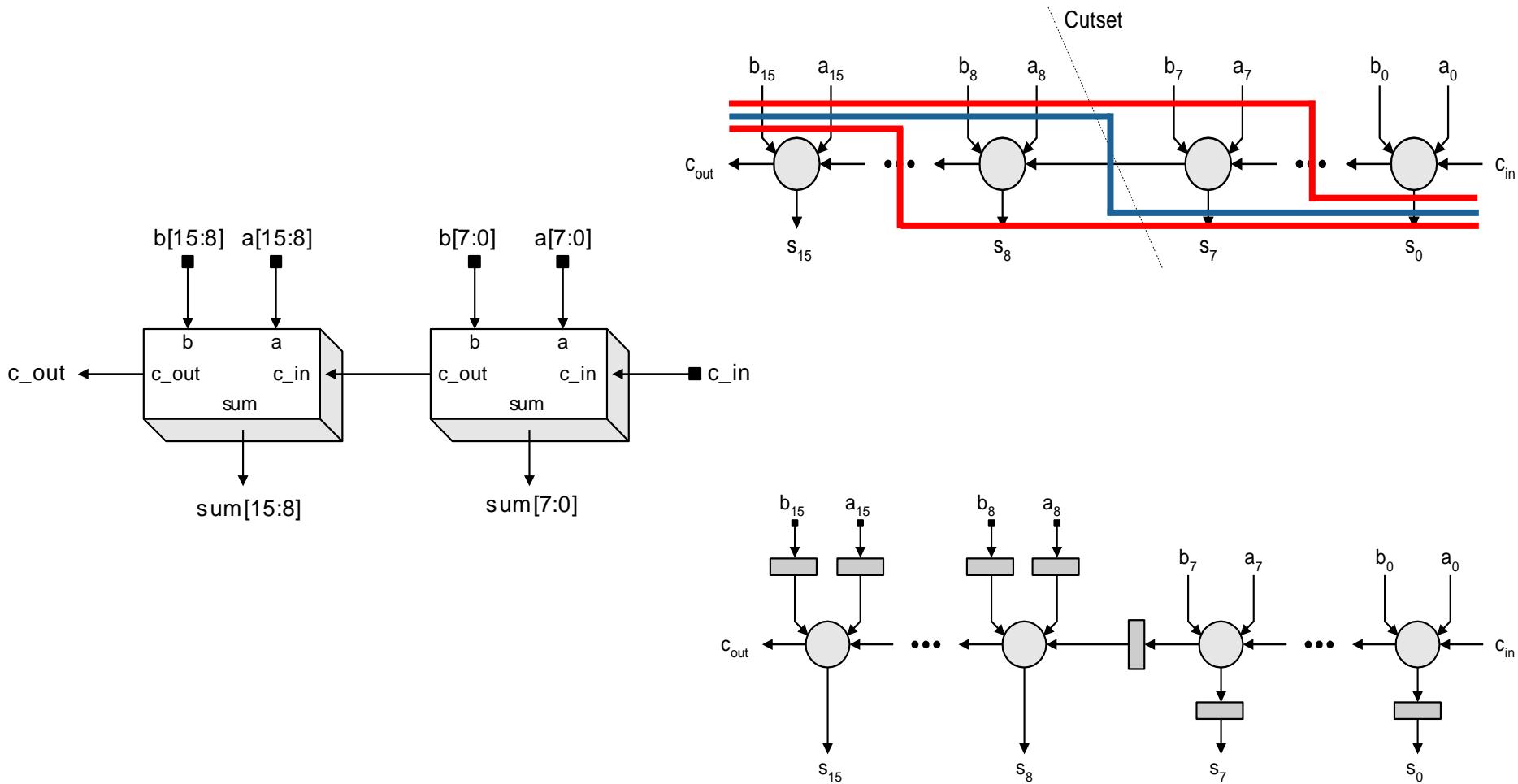


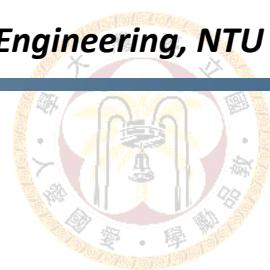
Feed-forward Cut Set



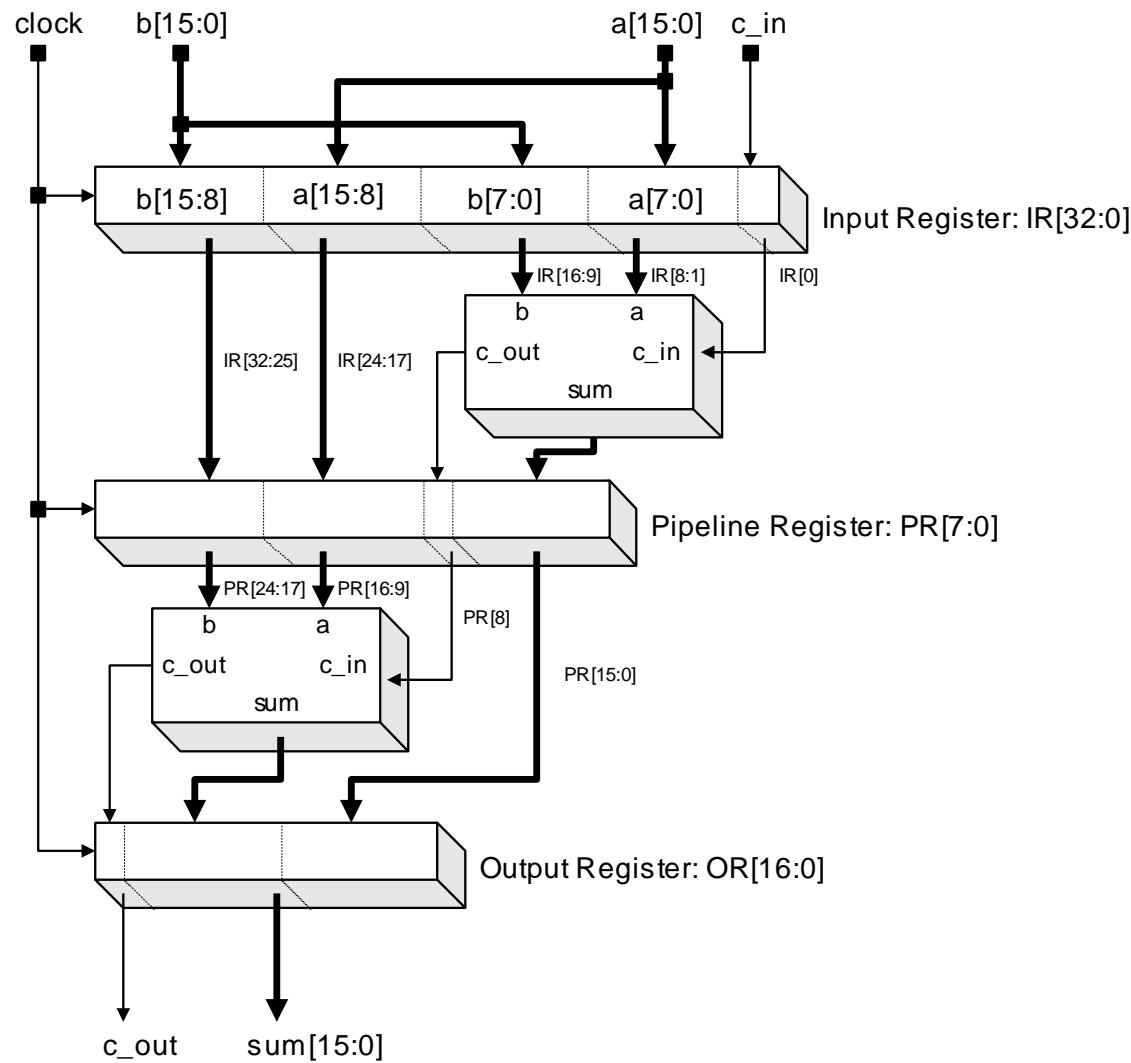


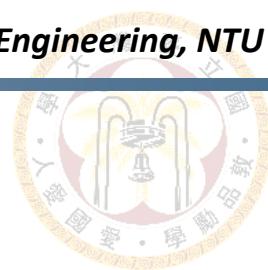
Balancing Performance between Stages





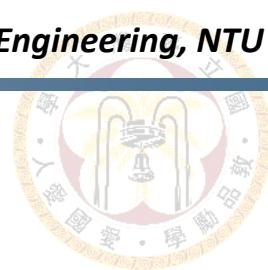
Example: Pipelined 16-bit Adder





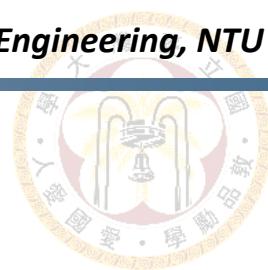
Pipeline Overhead

- **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



Retiming

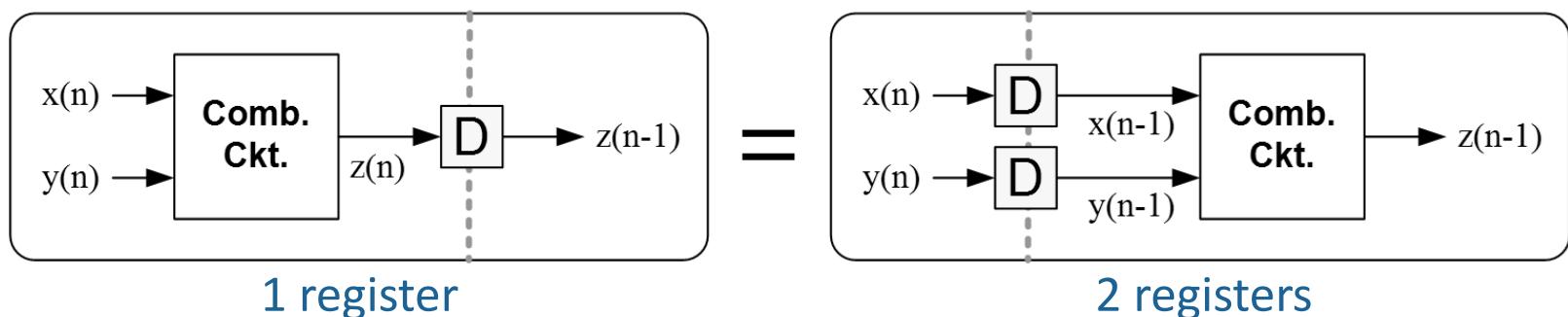
- **Changing location of registers without affecting functionality**
 - Balance latency on different combinational path
 - Possible reduction on clock period
- **Note**
 - May change the number of registers in a design
 - Retiming can be applied by synthesis tool



Retiming

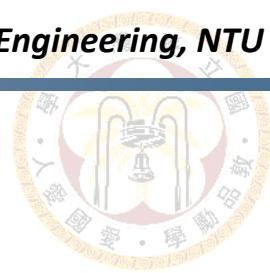
■ Cutset retiming

- Moving/adding registers on a feed-forward cutset



■ In Design Compiler:

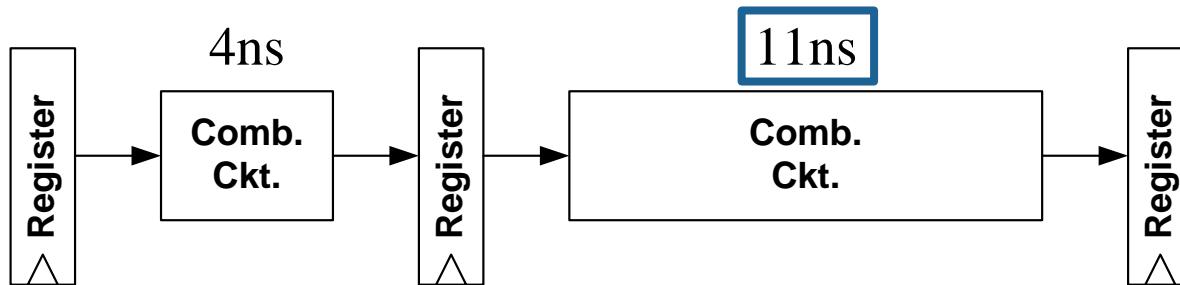
- `compile_ultra -retime`
(no retiming by default)



Example: Retiming

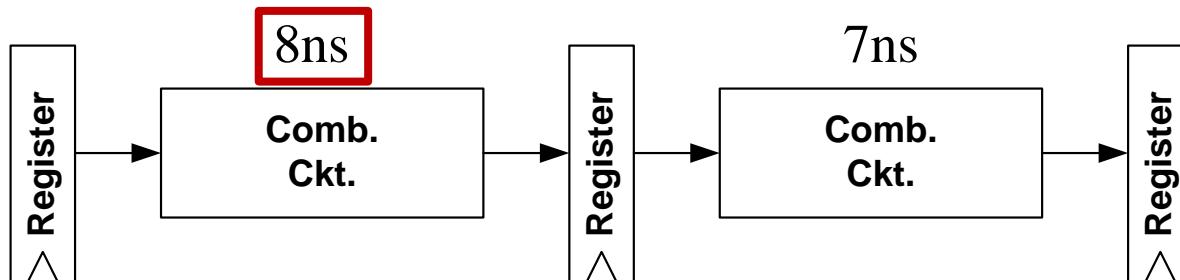
- **Original**

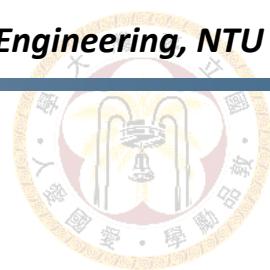
Clock period > 11ns



- **After Retiming**

Clock period > 8ns

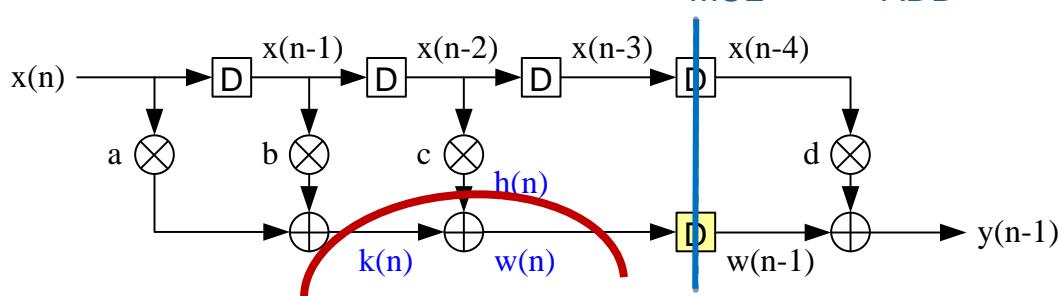




Example: Retiming

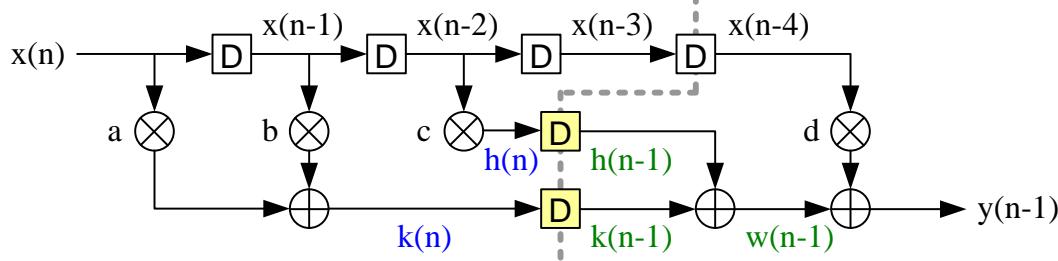
- Original

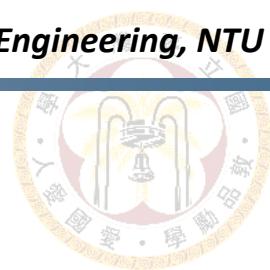
Clock period > $T_{MUL} + 2T_{ADD}$



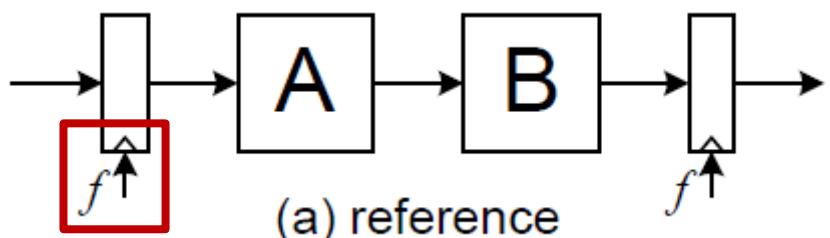
- After Retiming

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

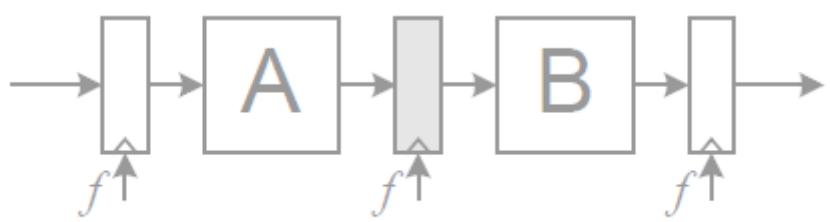




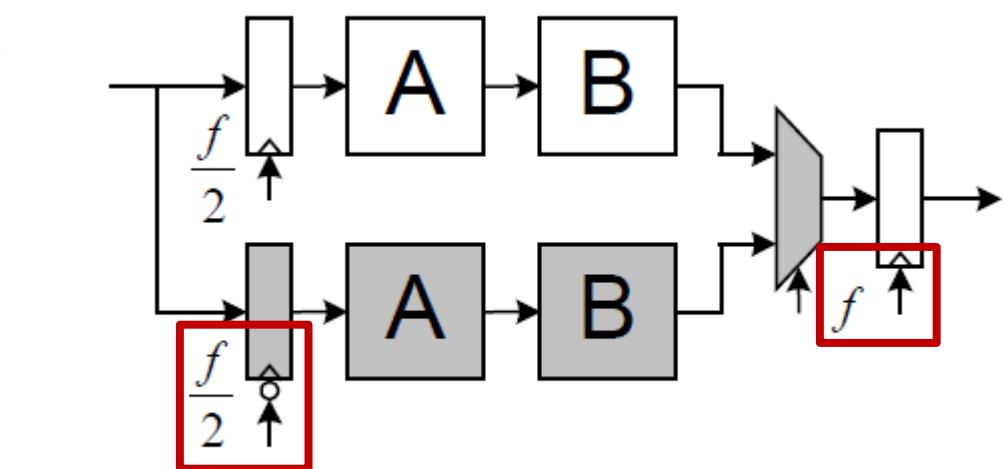
Parallel



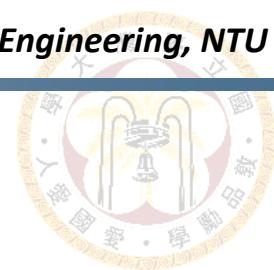
(a) reference



(c) pipeline

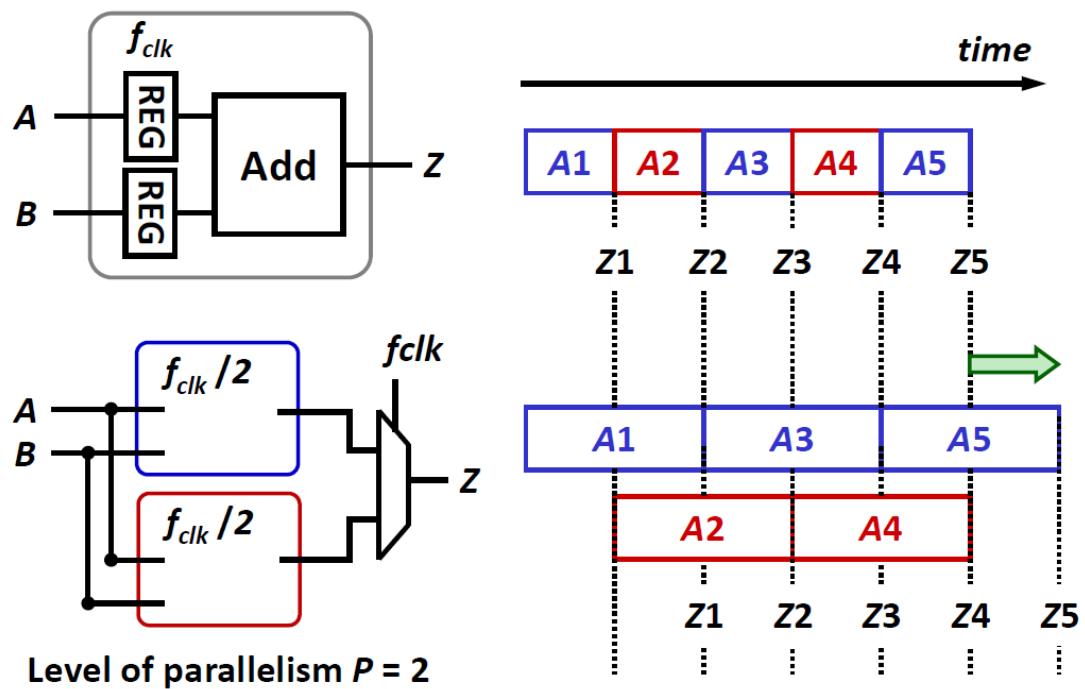


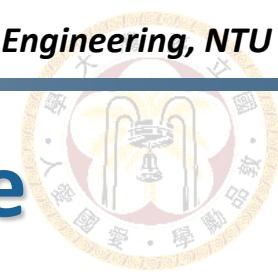
larger T_c (b) parallel
same throughput



Parallel Architecture: Timing

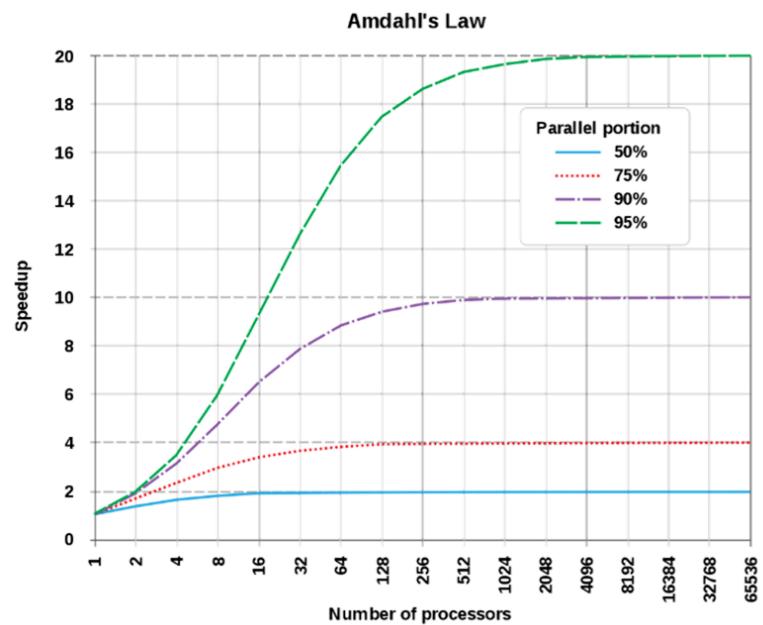
- For circuits with **same throughput**, parallel architecture will be:
 - Lower frequency
 - Larger area
 - Lower dynamic power
 - Longer latency

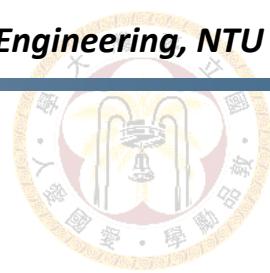




Parallel Architecture: Performance

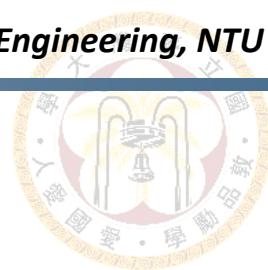
- For circuits with **same frequency**, parallel architecture can also be used to increase performance
- Parallel may not speed up as parallelizing factor (Amdahl's Law)
 - Non-parallelizable operations
 - Dependency between data
 - Dependency between operations
 - Limitation of I/O bottleneck
(communication bounded)





Outline

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



Area Issues

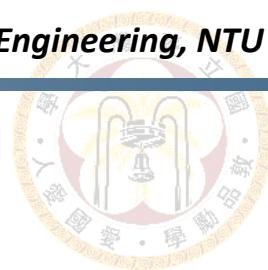
■ Area is cost

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

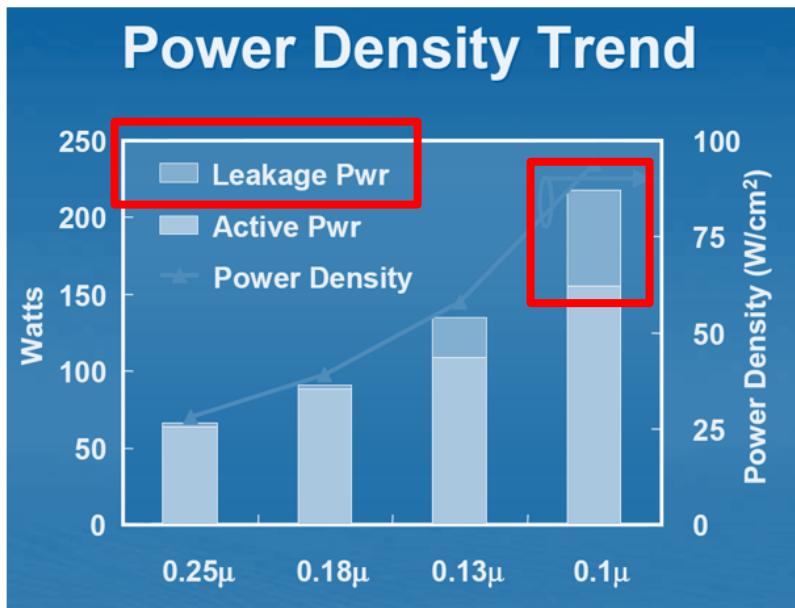
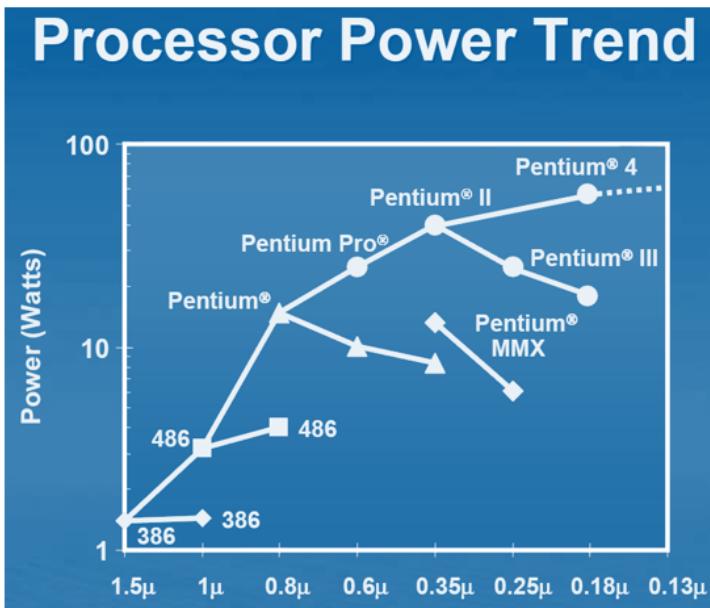
```
wire [15:0] A, B, C;  
wire [ 7:0] P, Q, R;  
wire mult_sel;  
  
assign A = P * Q;  
assign B = P * R;  
assign C = mult_sel ? A : B;
```

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

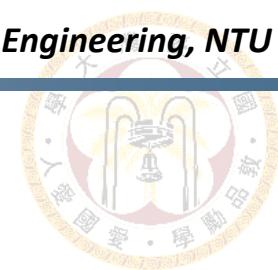
- There are tradeoffs between area and timing
 - Can be specified in synthesis constraints



Power Issues



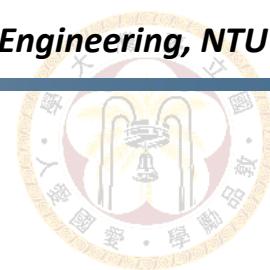
- Low power design is more important in modern chips due to heat dissipation, packaging, and portability requirements



Dynamic Power in CMOS

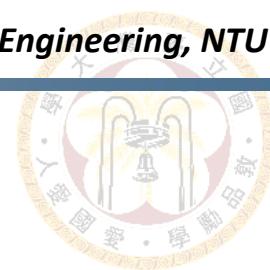
- α : switching activity
- f : clock frequency
- C : node capacitance
- V_{dd} : power supply voltage

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



Strategies for Low Power Design

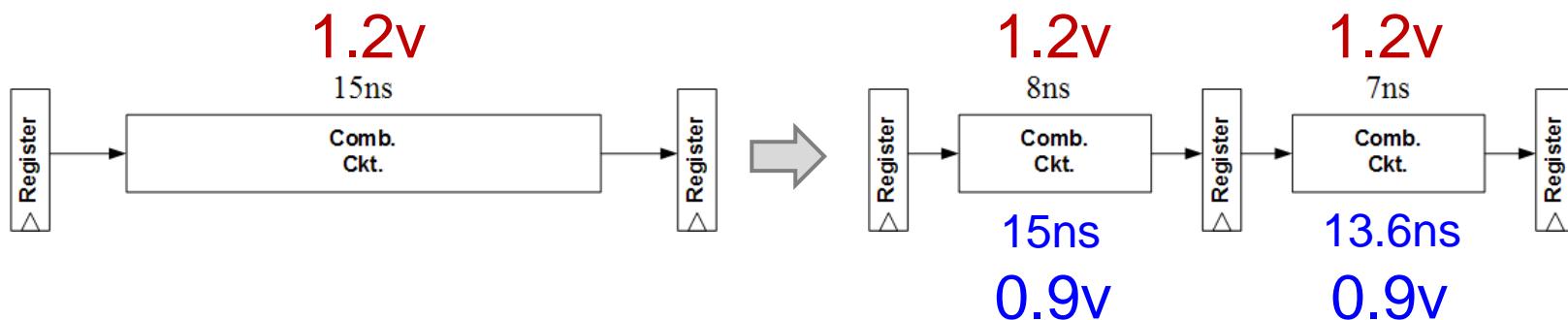
- Reducing Clock Frequency
 - Slower clock in power saving mode
 - Lower frequency with voltage scaling
- Reducing Switching Activity
 - Avoid unnecessary circuit switching
 - Reducing switching activity at I/O pins
 - Clock gating
- Note: these techniques can reduce dynamic power, but in modern technologies,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



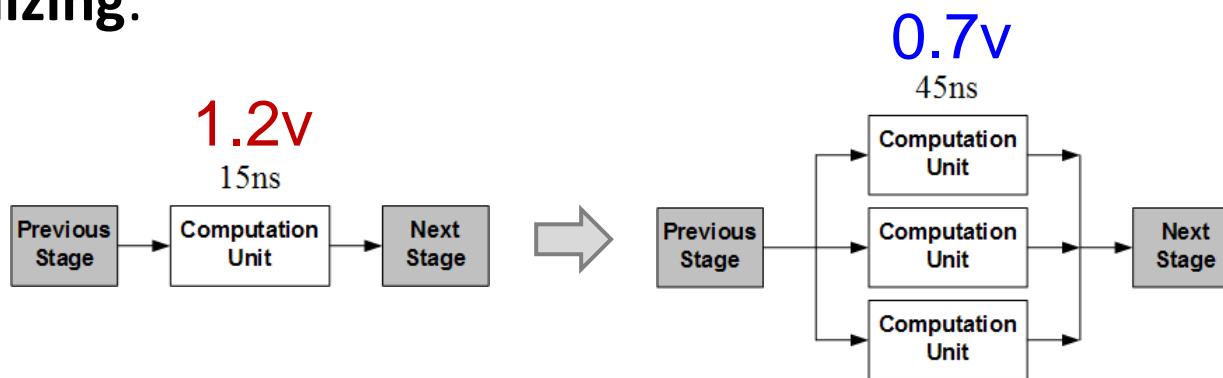
Voltage Scaling

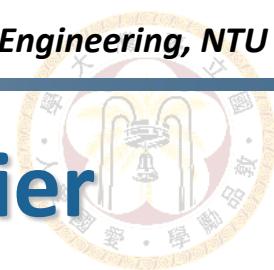
- Assume maintaining the same throughput

- Pipelining:



- Parallelizing:



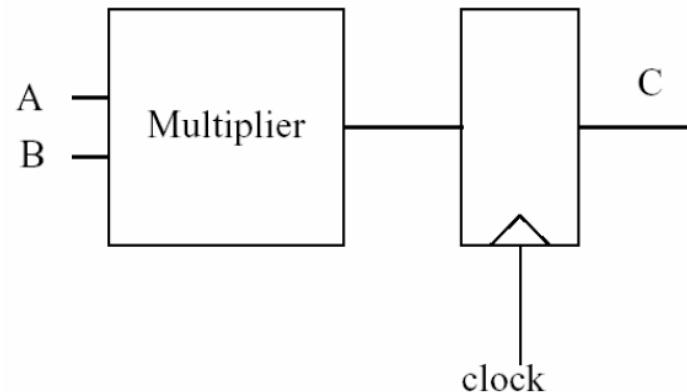


Example: Input Isolation for Multiplier

```

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

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

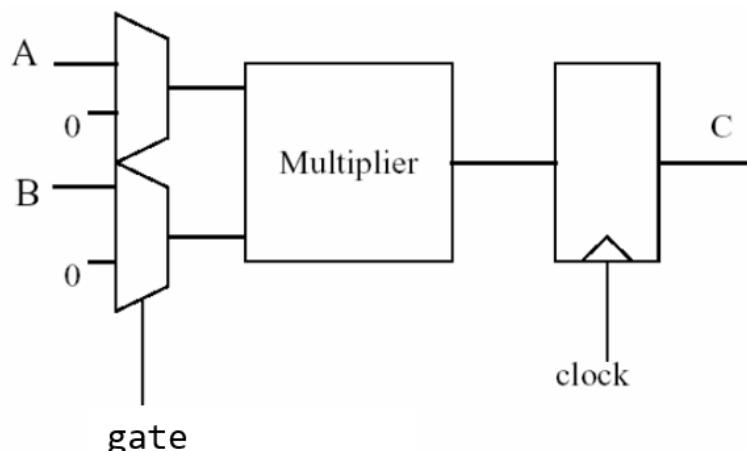


Gated

```

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

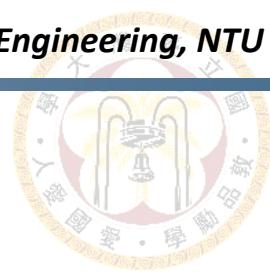
Chapter 4-2. Digital Design Guidelines: From Specification to Circuit

Lecturer: Ching-Yen Lee

*Graduate Institute of Electronics Engineering,
National Taiwan University*

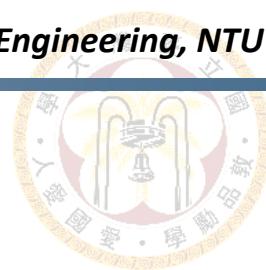


NTUGIEE



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

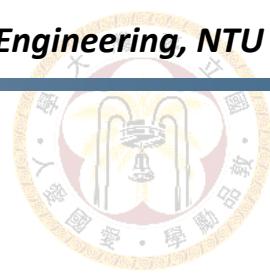
2. Control Flow

- Design a finite-state machine (FSM)
- Determine transition conditions
- Determine state outputs

3. Data path

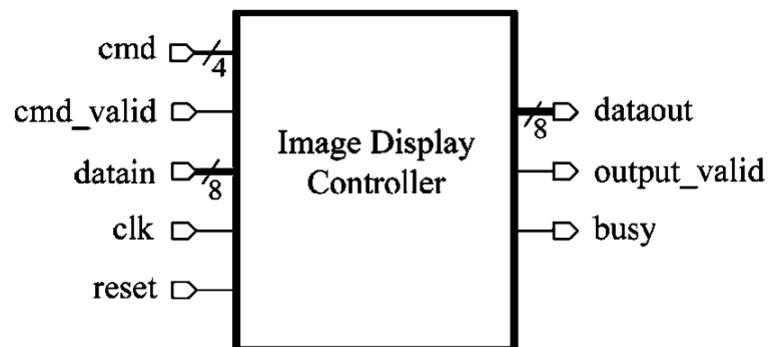
4. Others

- Reset strategy (sync/async, active high/low)
- Verify on paper before coding

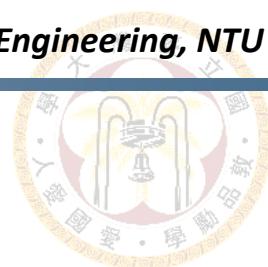


Datasheet & Timing Diagram

- At the beginning of digital circuit design
 - **I/O specification**: bit width, active high/low, input or output, ...
 - **Timing specification**: operating frequency, input/output delay, ...
- Example: documenting the signals of your design

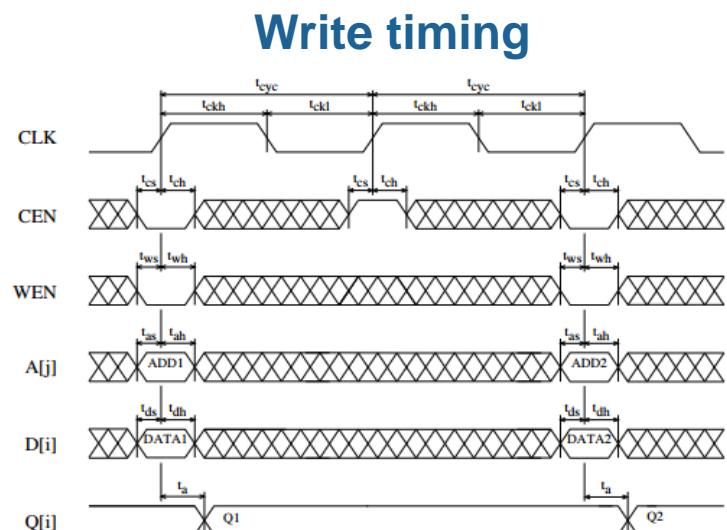
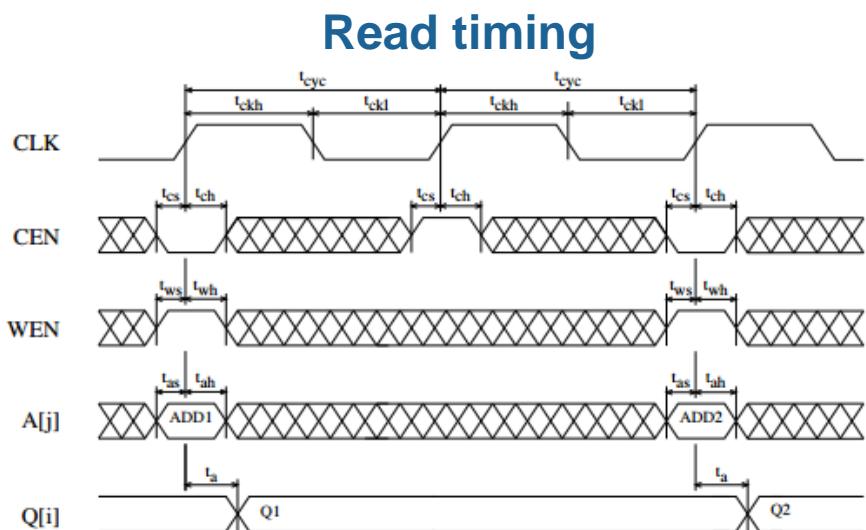


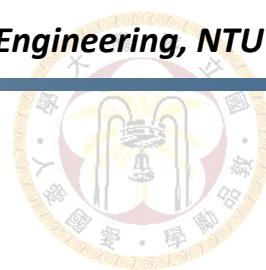
信號名稱	輸/出入	位元寬度	說明
reset	input	1	高位準非同步(active high asynchronous)之系統重置信號。 說明:本信號應於系統啟動時送出。
clk	input	1	時脈信號。 說明:此系統為同步於時脈正緣(posedge)之同步設計。
cmd	input	4	指令輸入信號。 說明:本控制器共有九種指令輸入，相關指令說明請參考表二。指令輸入只有在 cmd_valid 為 high 及 busy 為 low 時，為有效指令。
cmd_valid	input	1	有效指令輸入信號。 說明:當本信號為 high 時表示 cmd 指令為有效指令輸入。
datain	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

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

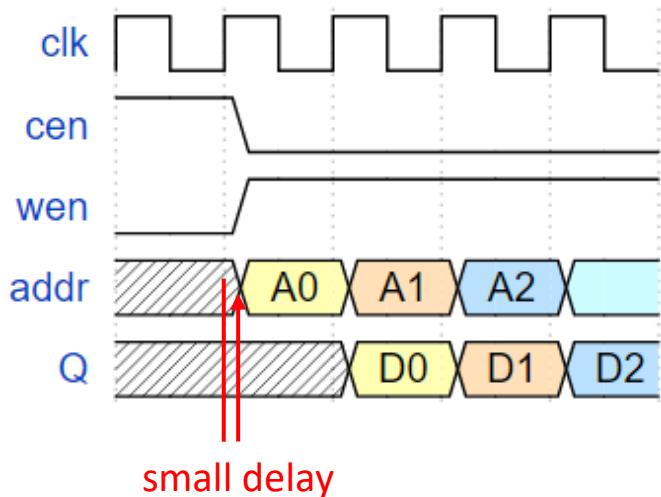




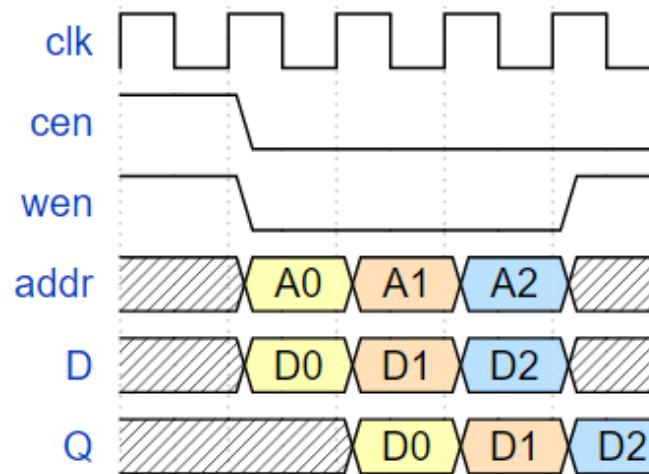
Example: SRAM

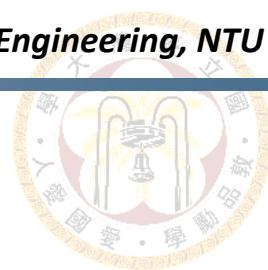
- Control address for data read
- Control address, input data, write enable for data write
- Draw timing diagrams for sequential circuits based on spec

Read timing



Write timing





Reset Strategy

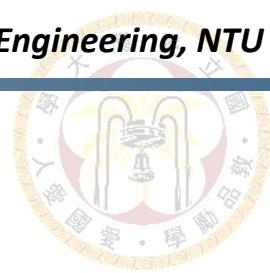
- Reset is a global signal distributed across the entire chip
- Initialize the chip to an idle state
- Be aware of the reset behavior
 - Synchronous/asynchronous
 - Active low/active high

```
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

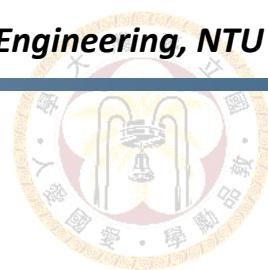
```
always @ (posedge clk) begin
    if (rst) begin
        // Reset assignment
    end
    else begin
        // Next state assignment
    end
end
```

Synchronous active-high reset



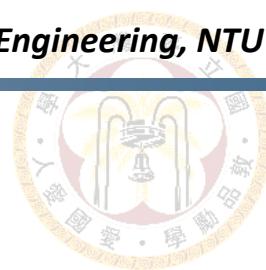
Outline

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



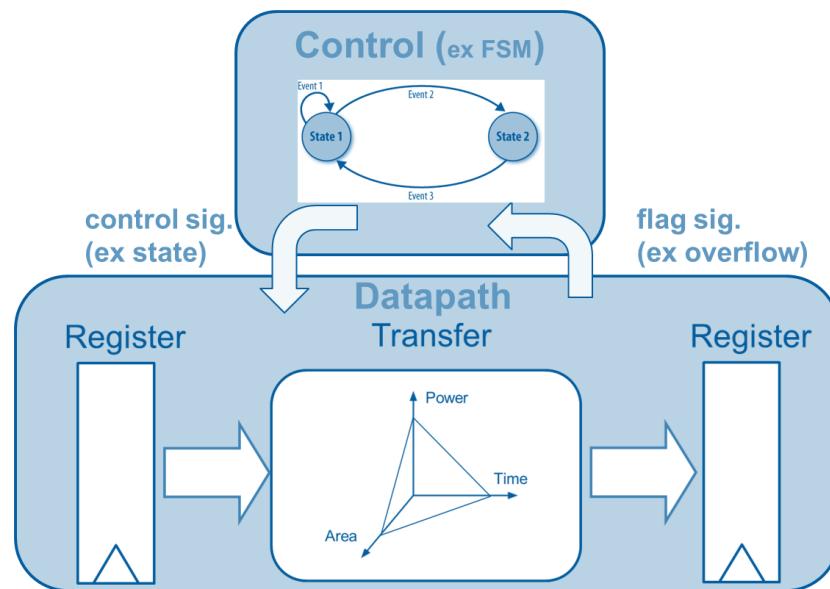
Controller & Datapath

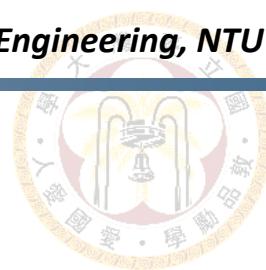
- Separated datapath and controller
- Controller
 - FSM, counter, ...
 - Controls the operations of datapath modules
- Datapath
 - Arithmetic & logic units, data registers, MUX, ...
 - Datapath performs computation on input data



Controller & Datapath

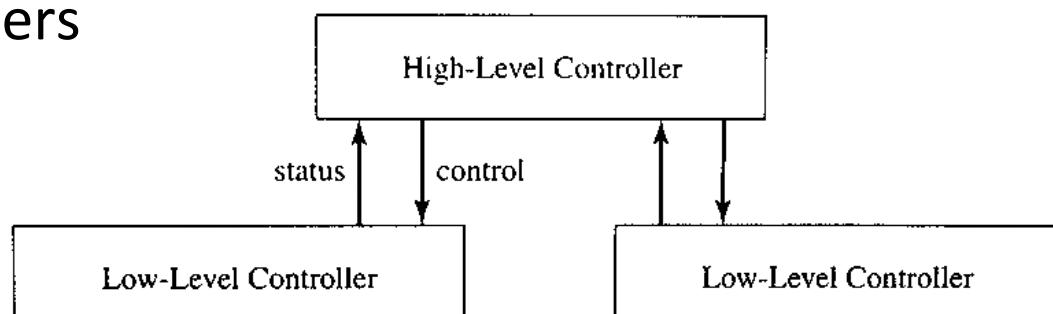
- **Control signals:** determine the detailed operations to be performed on the datapath
 - e.g., FSM states, start, select signals, ...
- **Status signals:** indicate the status of datapath modules
 - e.g., flags, finish, overflow, exceptions, ...



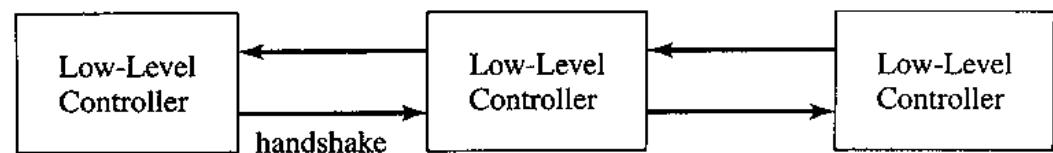


Control Strategy

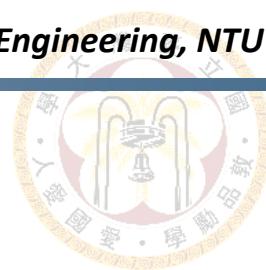
- Make sure the behavior of control signals are correct
- Generally, control sequence are generated by one or combination of the following:
 - Finite state machines (FSMs)
 - Top-down controllers
 - Counters
 - Software



Hierarchical Control (preferred)

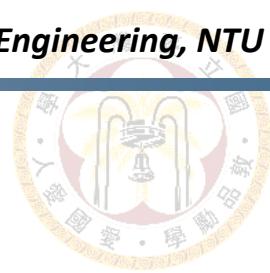


Interconnected Control (discouraged)

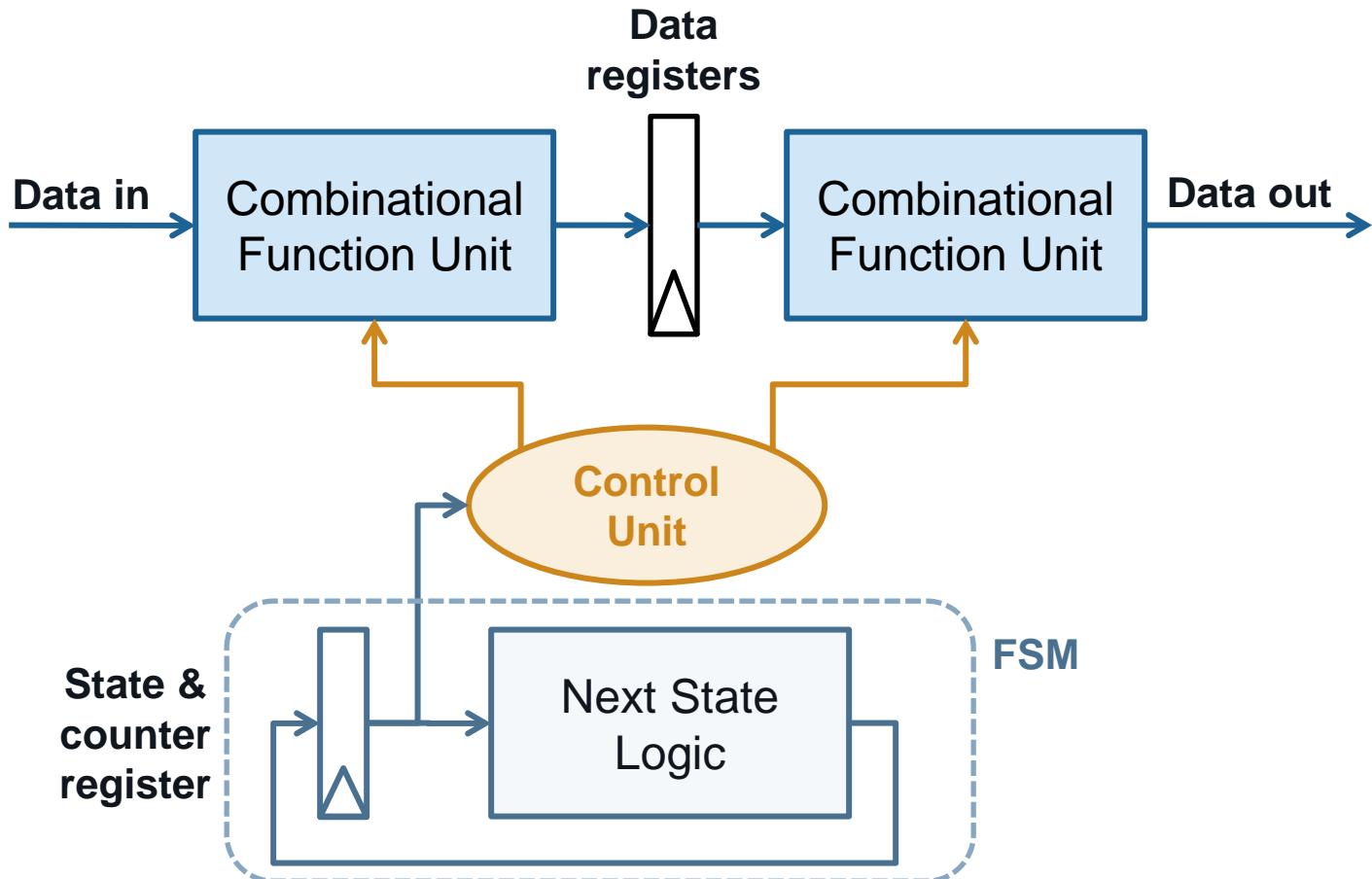


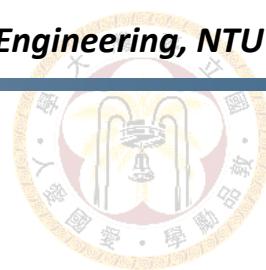
Structuring the Datapath

- Draw **block diagrams** first
- Determines function units and their connectivity
 - Low-level instances (arithmetic, logic, ...)
 - Memory instances (SRAM)
 - Submodules
- Determine the design parameters, for example:
 - Pipelining stages
 - Parallelization degree



Datapath Design

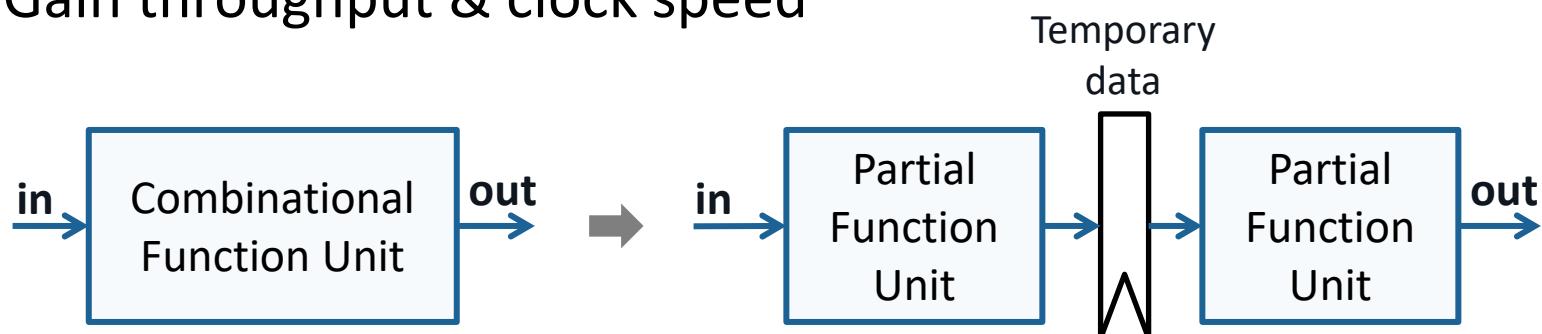




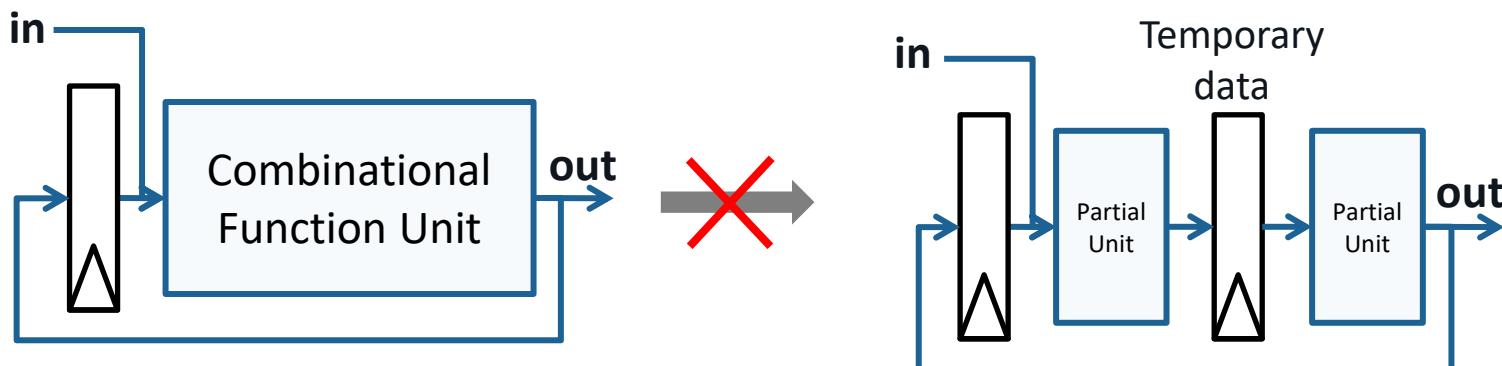
Datapath Design

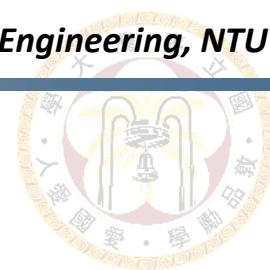
■ Pipeline

- Gain throughput & clock speed



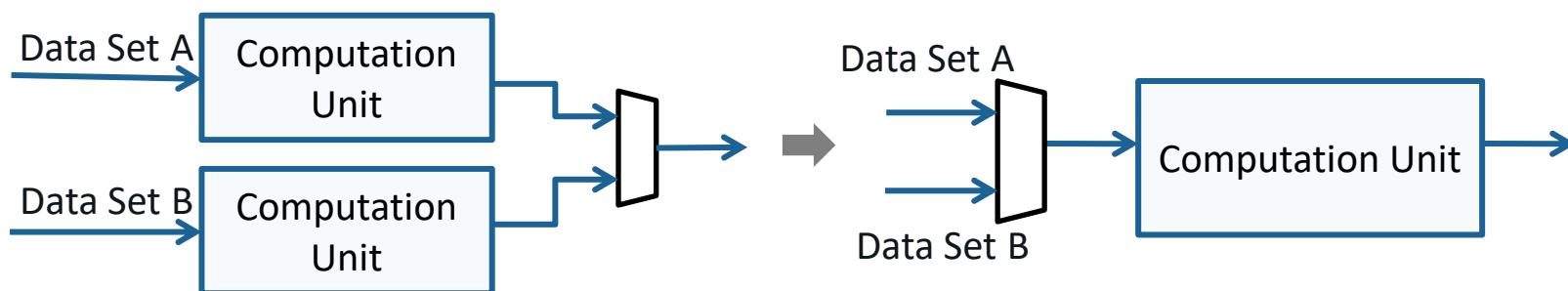
- Feedback loops can not be pipelined



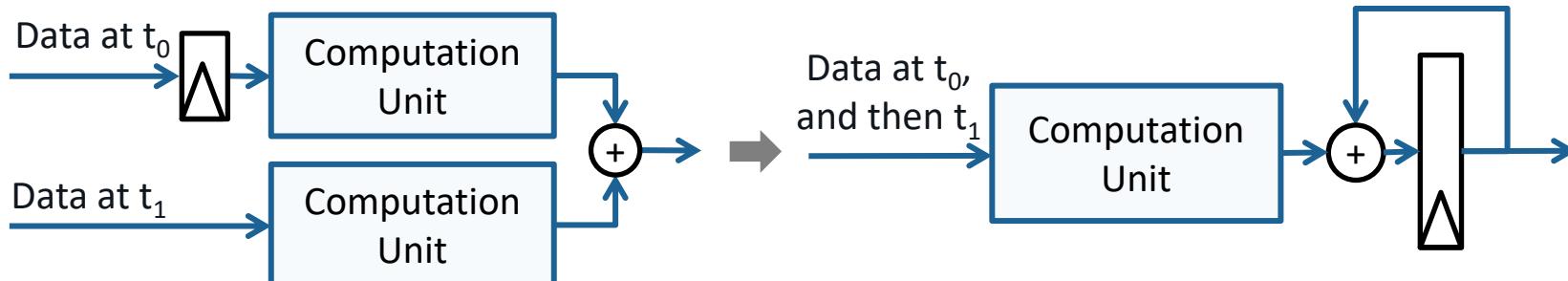


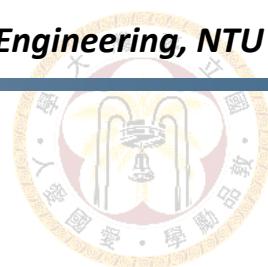
Datapath Design

- Functional block reusing
 - Reduce combinational area



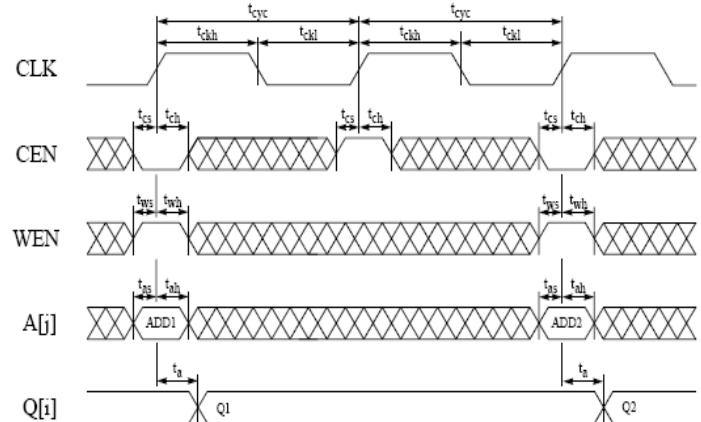
- Reduce area when input comes sequentially



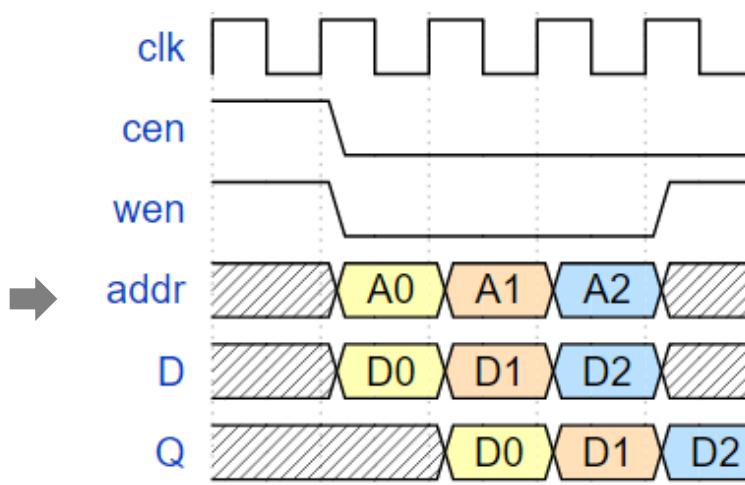


Note: Verify before Coding

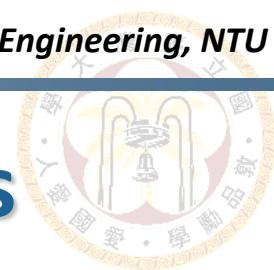
- It is a useful practice to hand-simulate the design before coding
- Draw a timing diagram capturing critical events (e.g., state transitions and signal changes)



specification

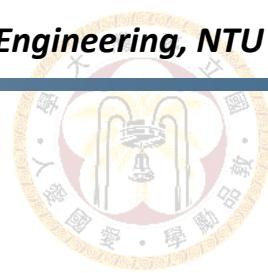


timing in your design



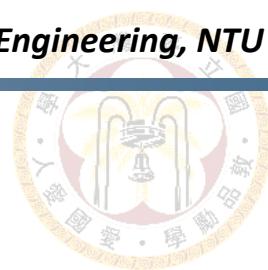
Conclusion: Partition for Synthesis

- Separate combinational and sequential (CH3)
 - Separate control and datapath (CH4-2)
 - Register at hierarchical output (CH3)
-
- Keep major blocks separate (draw block diagrams)
 - Avoid glue logic (logic between module connections)
 - Avoid asynchronous logic, false path, and multi-cycle path



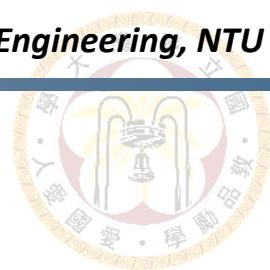
Outline

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



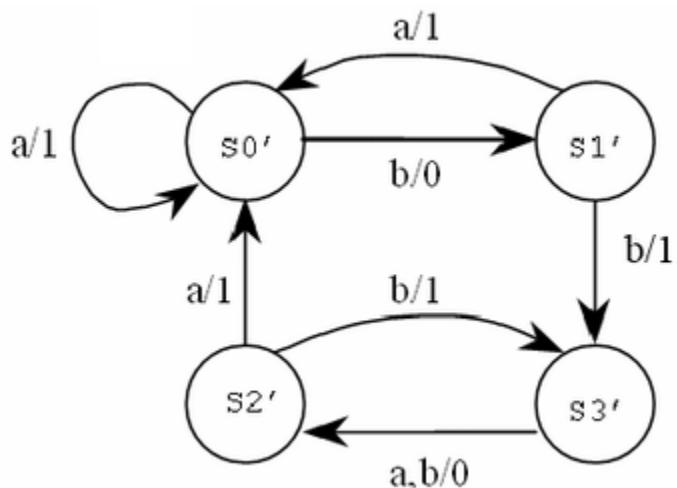
Finite State Machines

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

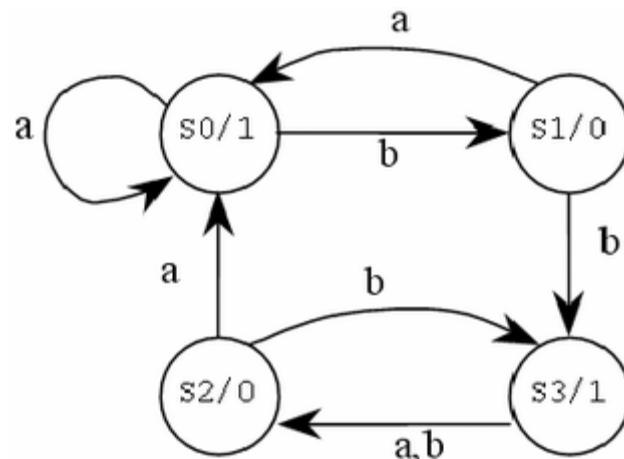


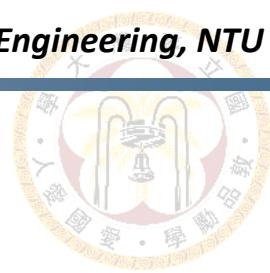
Finite State Machines

Mealy



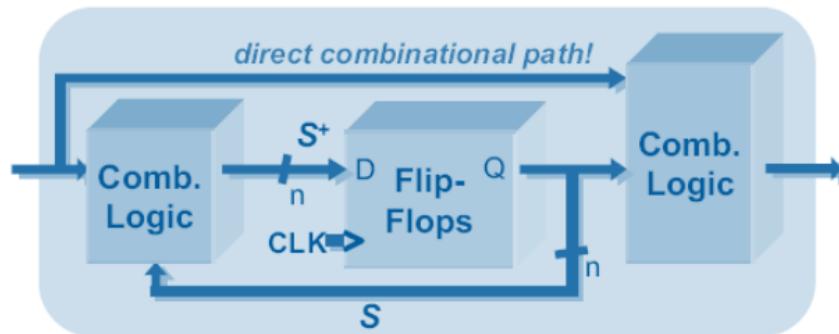
Moore



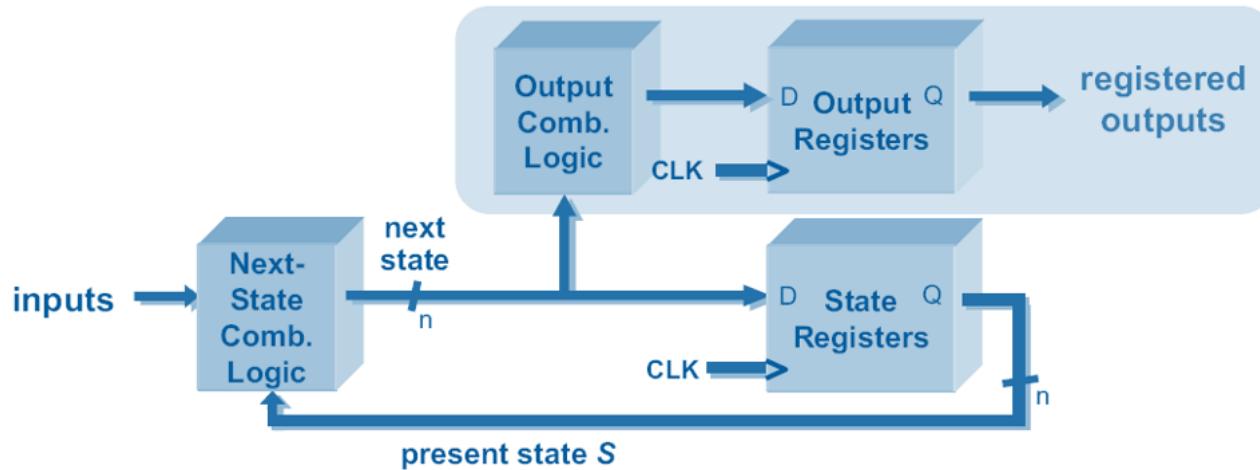


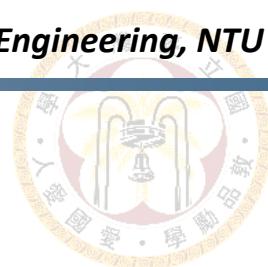
Mealy & Moore Machine

- 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;
```

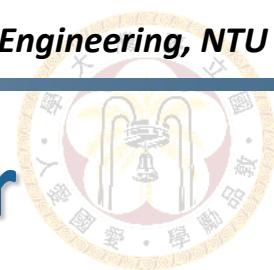
```
always @ (*) begin  
    state_w = state_r;  
    case (state_r)  
        S_IDLE: begin ... end  
        S_READ: begin ... end  
        S_WRITE: begin ... end  
        S_DONE: begin ... end  
    endcase  
end
```

Use localparam or enum
for state constants

State register

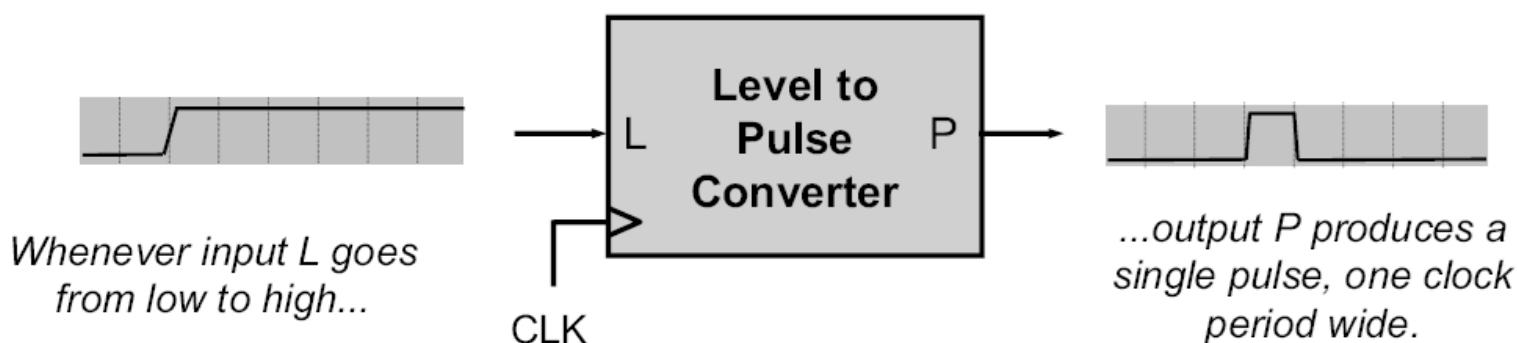
```
always @ (posedge clk or posedge  
rst) begin  
    if (rst) begin  
        state_r <= S_IDLE;  
    end  
    else begin  
        state_r <= state_w;  
    end  
end
```

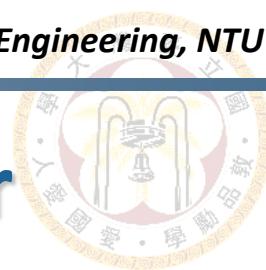
State transitions



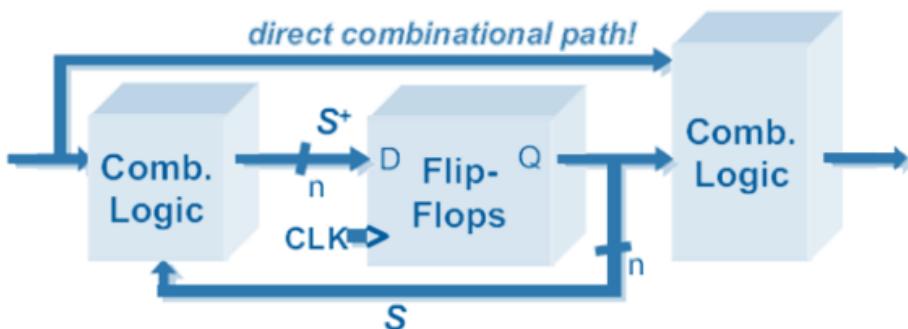
Example: Level-to-Pulse Converter

- A level-to-pulse converter produces a single-cycle pulse each time its input goes from low to high
 - I.e., a synchronous rising edge detector
- Applications
 - Button and switches
 - Single-cycle enable signal

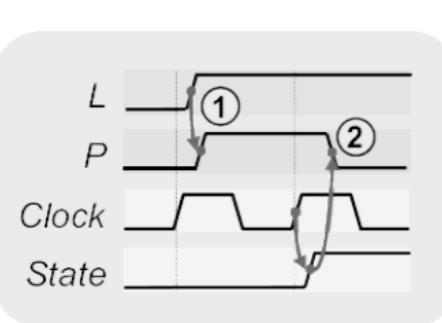
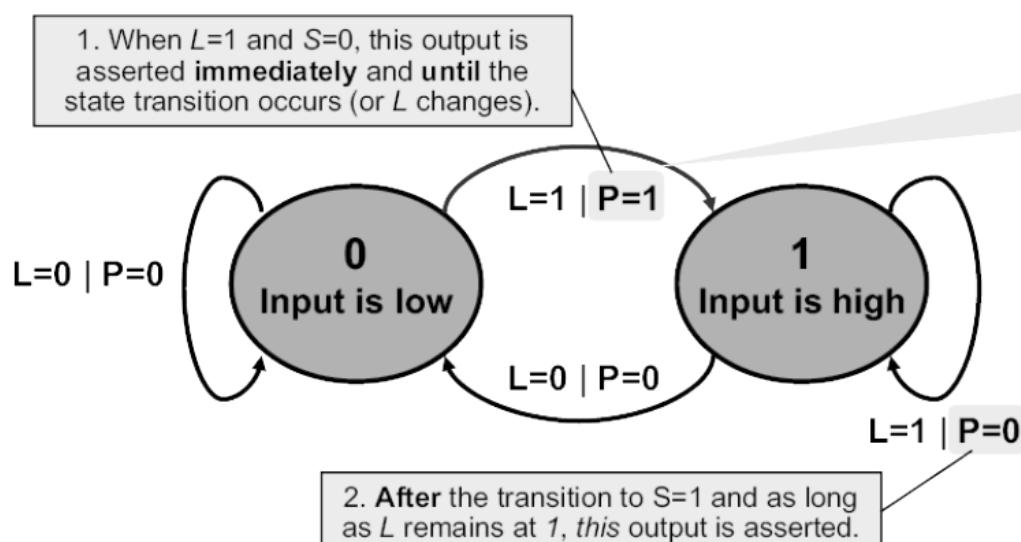




Example: Level-to-Pulse Converter

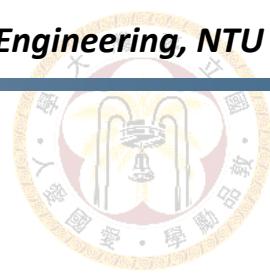


- Since outputs are determined by state *and* inputs, Mealy FSMs may need fewer states than Moore FSM implementations



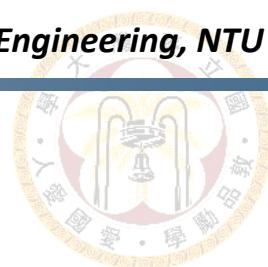
Output transitions immediately.

State transitions at the clock edge.



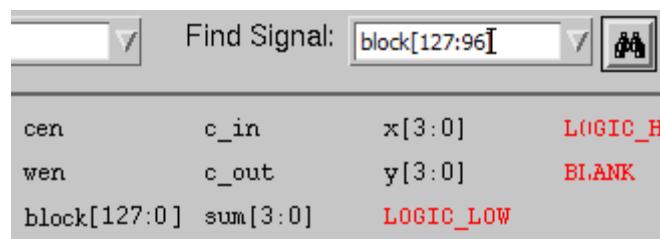
Outline

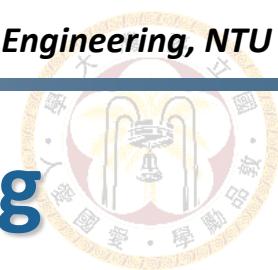
- Design Planning
- Design Structure
- Finite State Machines
- **More on Debugging**
- Tools for Design Planning



RTL Debugging

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





Naming Conventions for Debugging

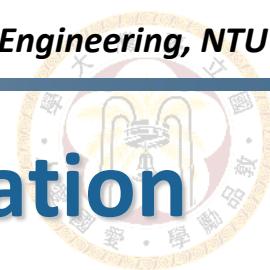
■ Signal naming suffix

- Utilizing suffix conventions to show the attributes of signals, for example:
- `signal_n`: active-low
- `signal_w`: wire
- `signal_r`: register
- `signal_next`: next-state of FSM
- `signal_cur`: current-state of FSM

■ Alphabetically naming for debugging

Get Signals	
Find Signal: <input type="text"/>	
clk	out[7:0]
cmd[2:0]	row_counter[2:0]
column_counter[3:0]	rst
ctrl[3:0]	LOGIC LOW
flag	LOGIC_HIGH
in1[7:0]	BLANK
in2[7:0]	

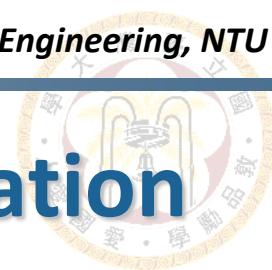
Get Signals	
Find Signal: <input type="text"/>	
alu_cmd[2:0]	ctrl[3:0]
alu_in1[7:0]	flag
alu_in2[7:0]	rst
alu_out[7:0]	LOGIC LOW
clk	LOGIC_HIGH
counter_col[3:0]	BLANK
counter_row[2:0]	



Timing Checks in Post-synthesis Simulation

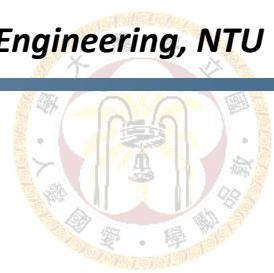
- **SDF annotation**
 - Make sure the SDF file exists, and the path is correct
 - Make sure the SDF file is parsed correctly during simulation

```
$sdf_annotation("design.sdf", testbench);
```
- **Input delay and output delay**
 - Synthesis tool does not know when your inputs come, and when the outputs are captured
 - No input and output delay during synthesis by default
 - Set the constraints properly (Chapter 5)
 - Model input and output delay correctly in testbench



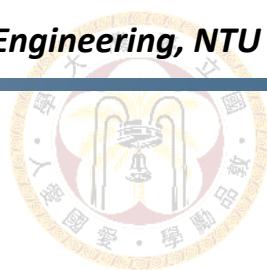
Timing Checks in Post-synthesis Simulation

- **Setup-time violation**
 - Input of flip-flop not stable when clock triggers
 - 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



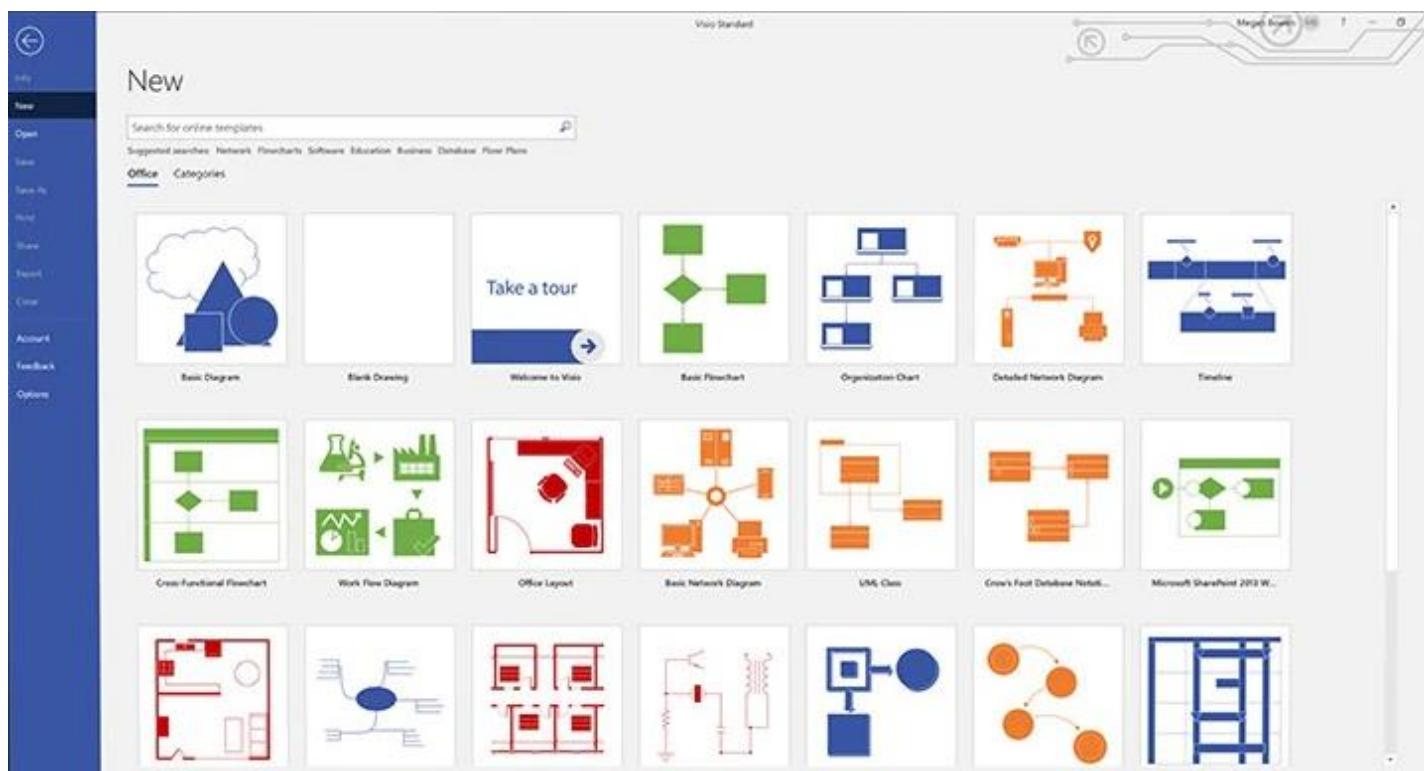
Outline

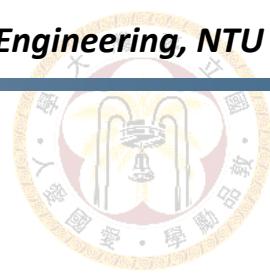
- Design Planning
- Design Structure
- Finite State Machines
- More on Debugging
- **Tools for Design Planning**



Block Diagram: Visio

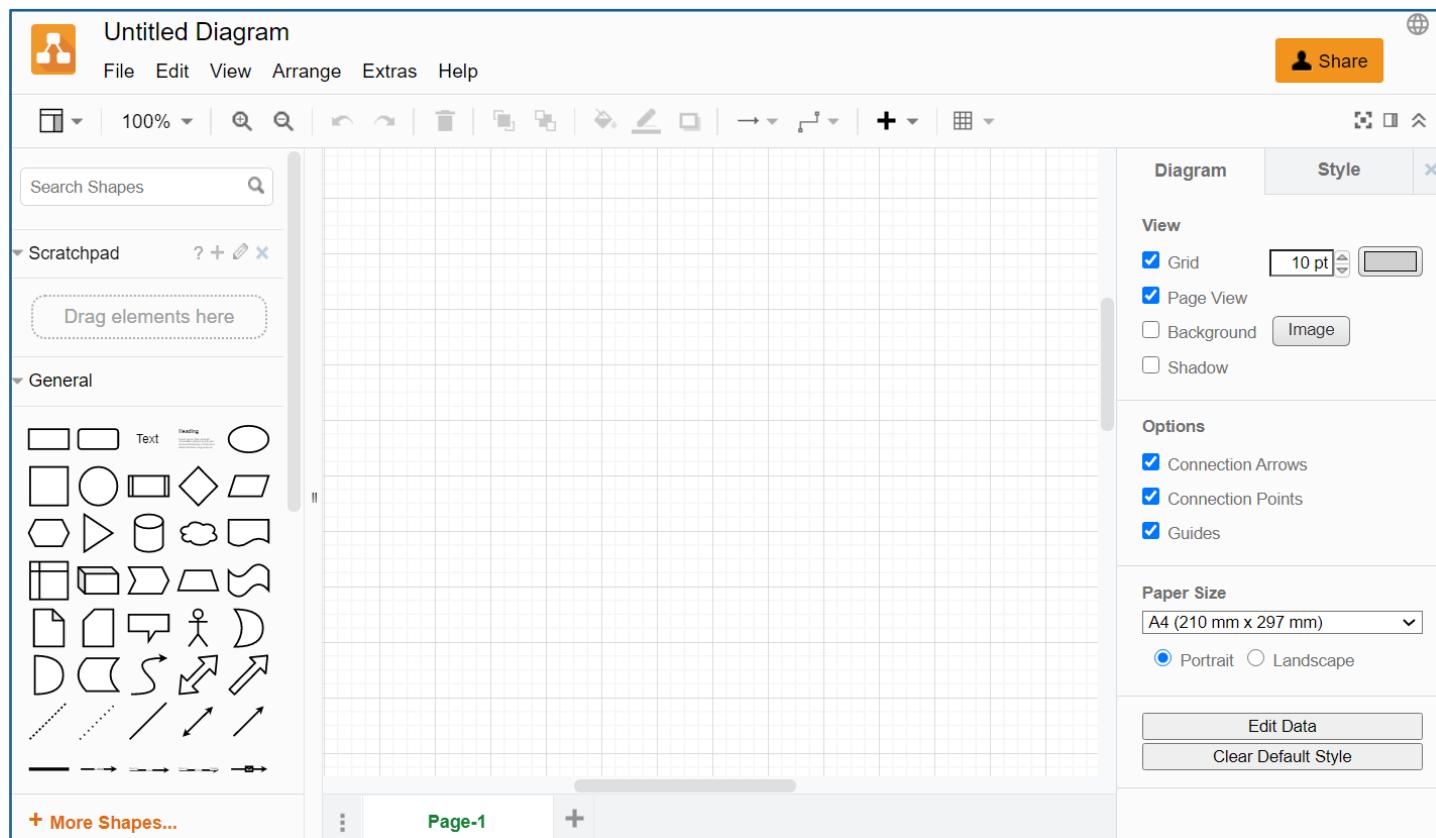
- Microsoft Office Visio
 - A powerful tool for block diagram, flow chart

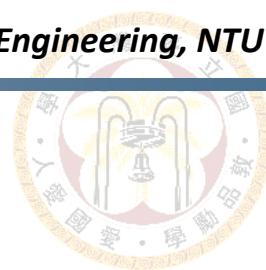




Block Diagram: draw.io

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

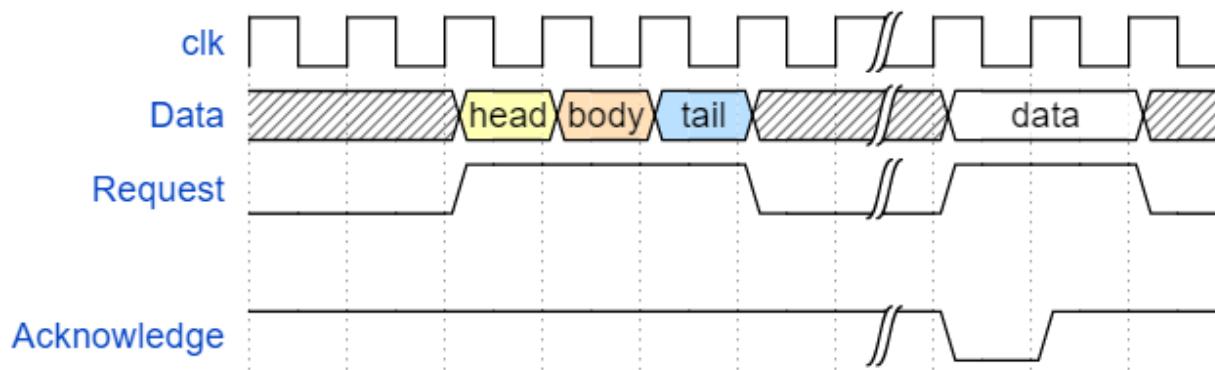


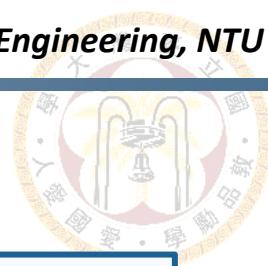


Timing Diagram: WaveDrom

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

```
{ 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." }  
]
```

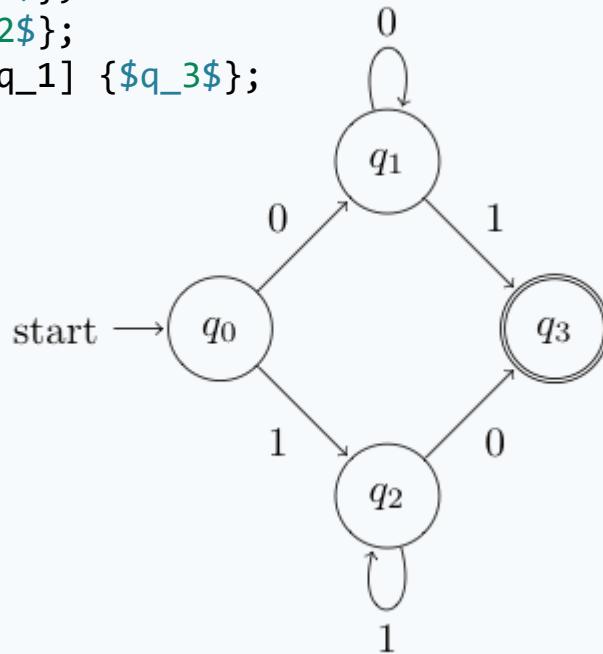


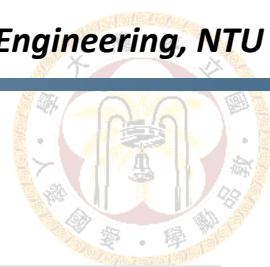


FSM: LaTeX + TikZ

```
\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

LaTeX

fsm.tex

```

1 \documentclass{article}
2
3 \usepackage{tikz}
4 \usetikzlibrary{automata,positioning}
5 \begin{document}
6 \begin{tikzpicture}[shorten >=1pt,node distance=2cm, on grid, auto]
7   \node[state,initial] (q_0)   {$q_0$};
8   \node[state] (q_1) [above right=of q_0] {$q_1$};
9   \node[state] (q_2) [below right=of q_0] {$q_2$};
10  \node[state,accepting](q_3) [below right=of q_1] {$q_3$};
11  \path[->]
12    (q_0) edge node {0} (q_1)
  
```

test.json

```

1 { signal: [
2   { name: "clk",      wave: "p.....|..." },
3   { name: "Data",     wave: "x.345x|=x", data: ["head", "body", "tail"] },
4   { name: "Request",  wave: "0.1..0|1.0" },
5   {},
6   { name: "Acknowledge", wave: "1.....|01." }
7 ]}
```

Waveform Render: test.json

draw.io

test.drawio

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