Datapath Design

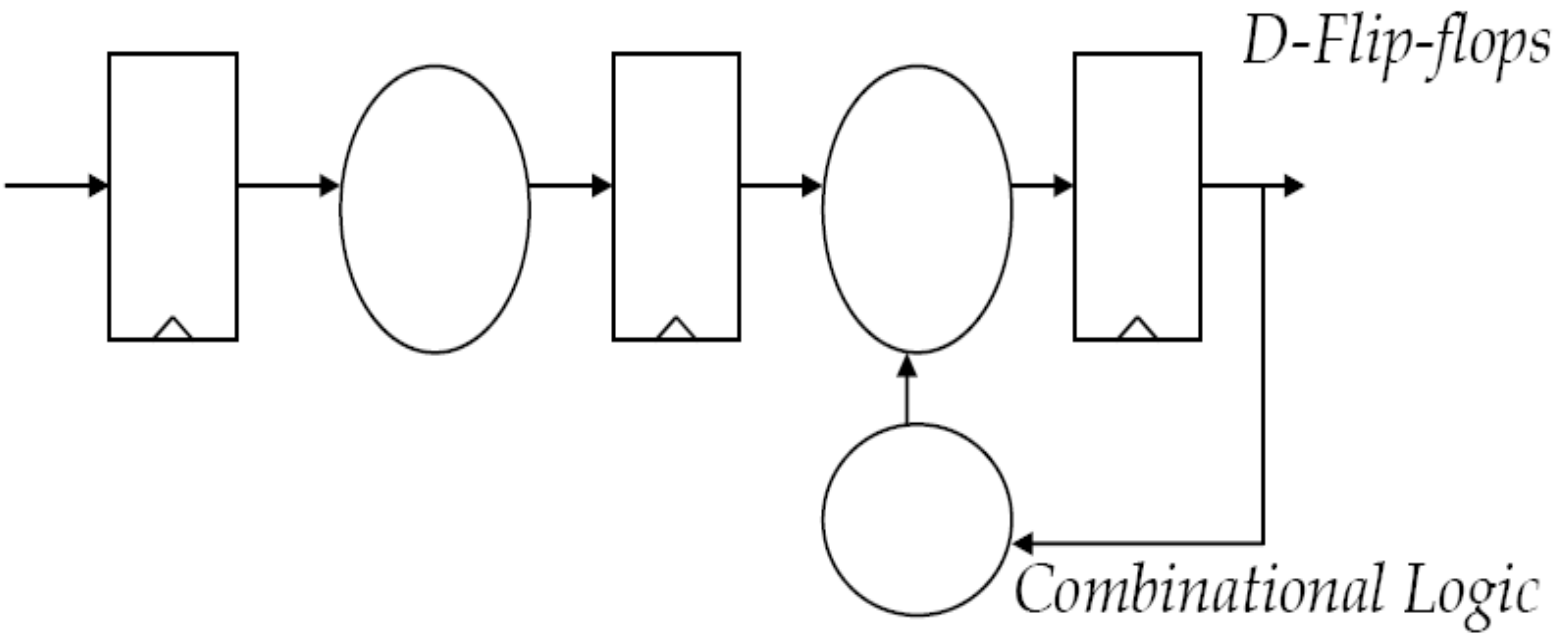
Timing Design in Digital Systems

One clock

- One clock, one edge; Flip-flops only
  - For your design (at least for each module) use one clock source and only one edge of that clock
  - Only use edge-triggered flip-flops
- Why?
  - Moving data between different clock domains requires careful timing design and synthesis “scripting”
- If you need multiple clocks in your design
  - Make them related by a powers of 2
    - E.g 50, 100 and 200 MHz
  - Consider one clock per module
- Consider resynchronizing using flip-flops between clock domains

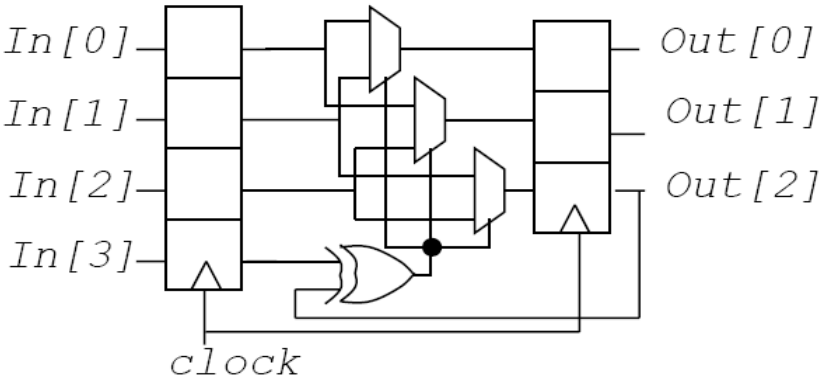
General Approach to Timing Design

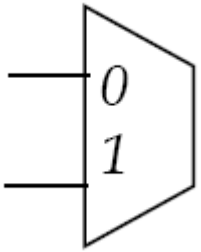
- In general, all signals start and end in registers every clock period

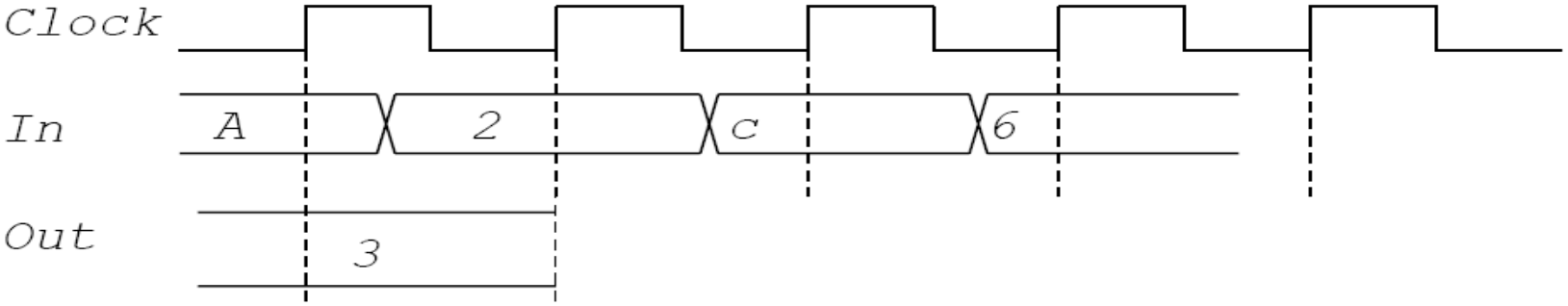


Clock Level Timing

- Example



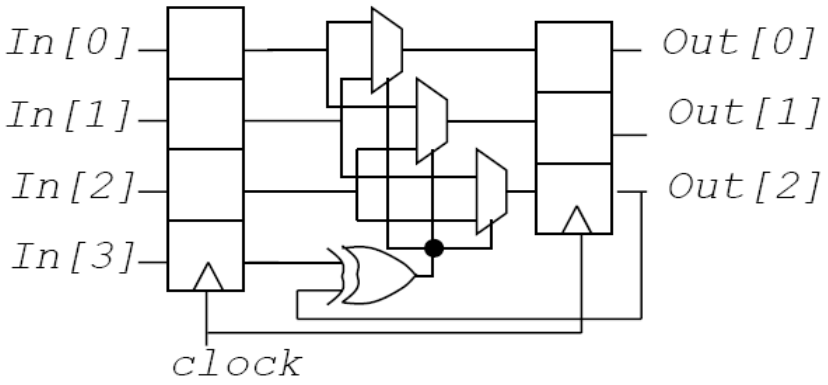


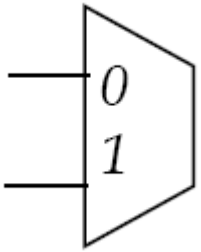


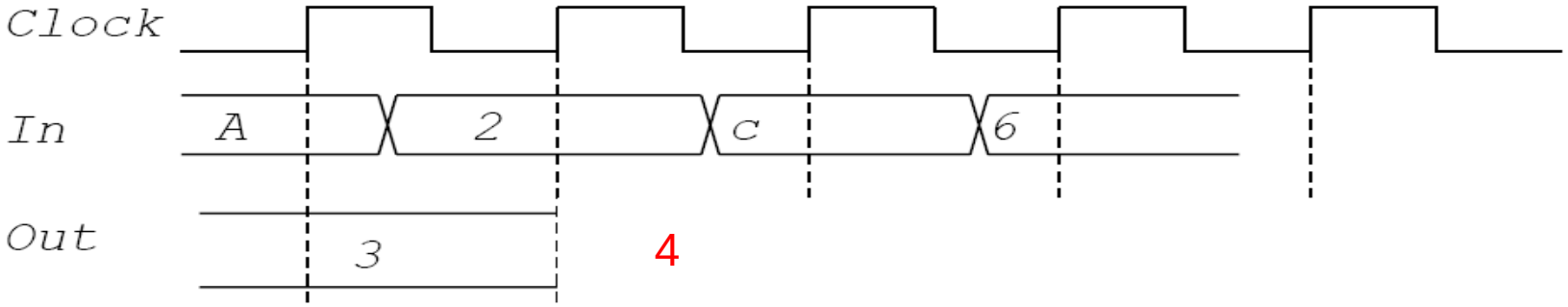
SYNOPSYS®

Clock Level Timing

- Example



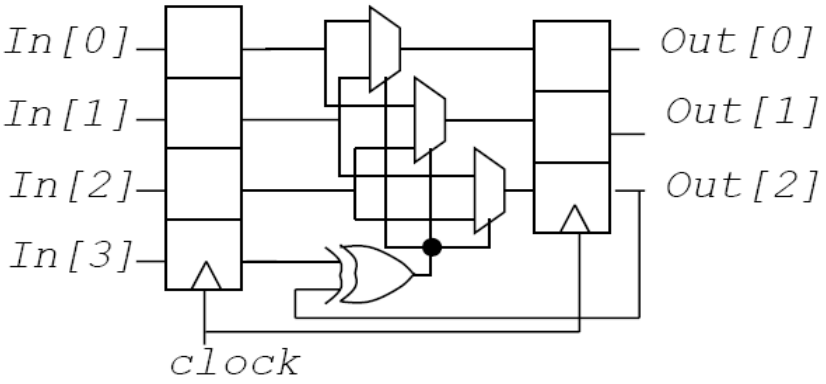


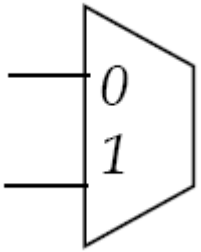


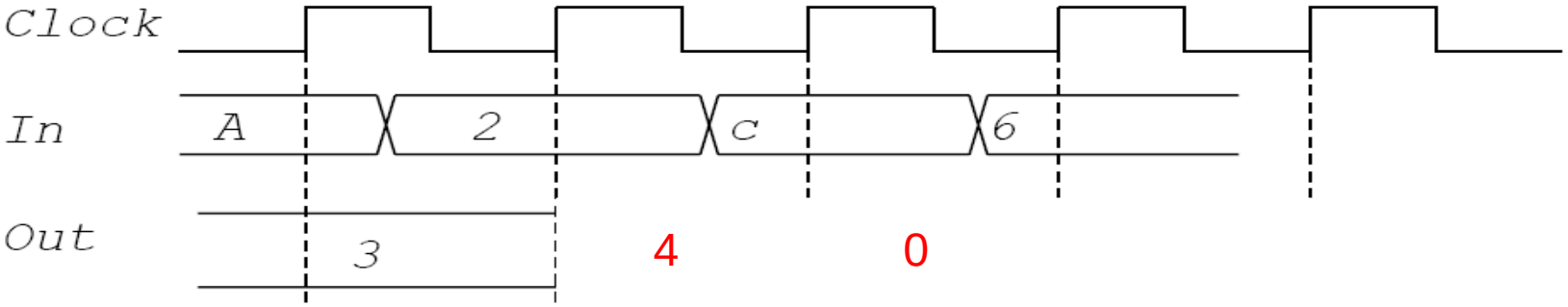
SYNOPSYS®

Clock Level Timing

- Example



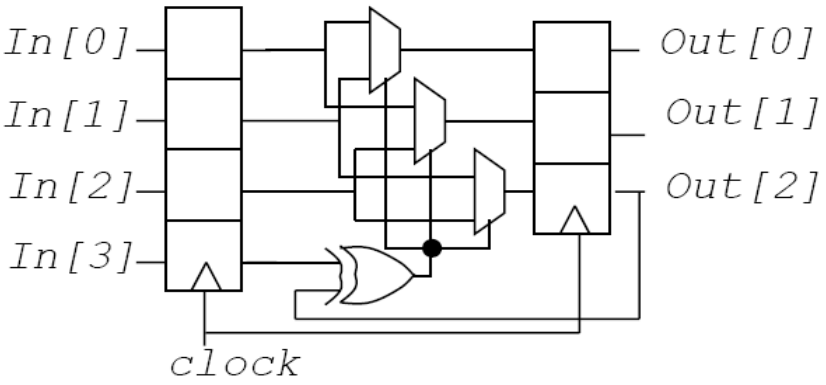


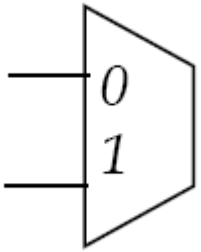


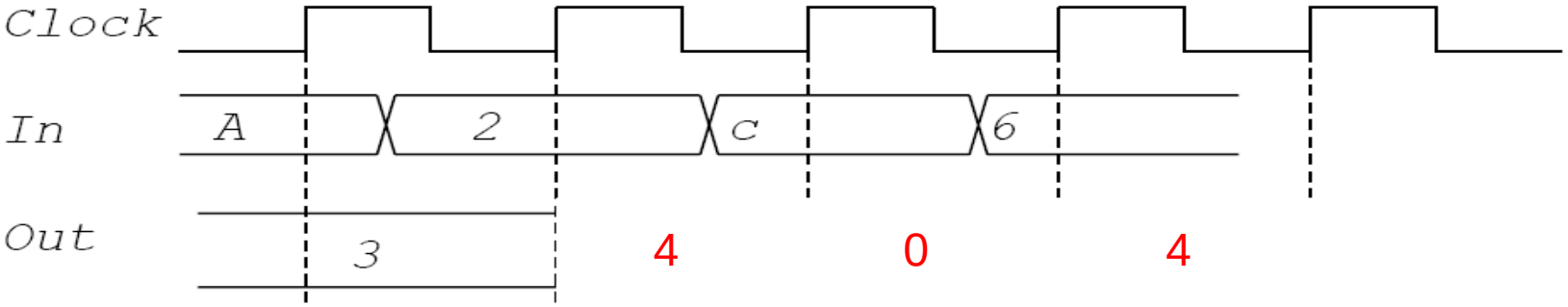
SYNOPSYS®

Clock Level Timing

- Example



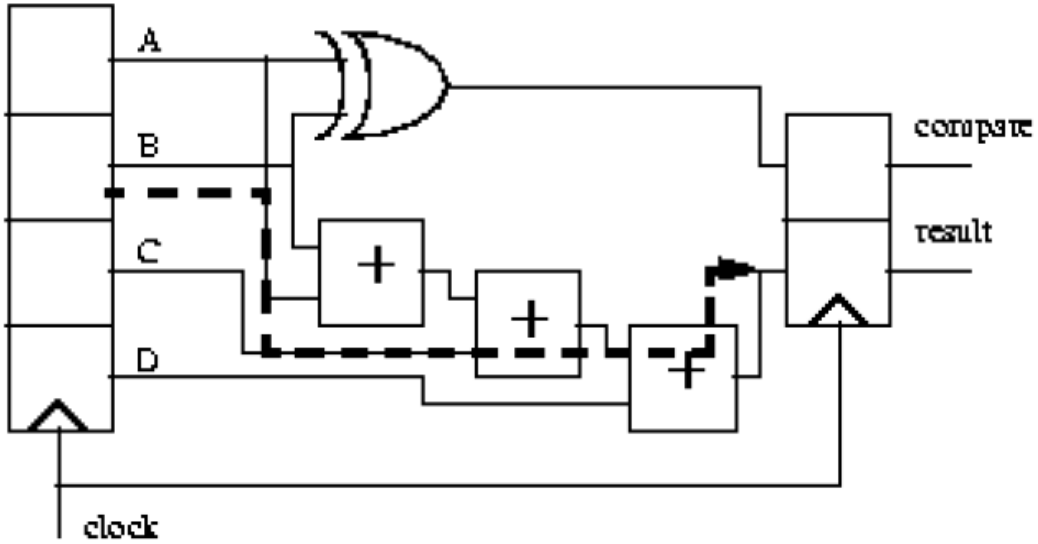




SYNOPSYS®

Critical Path

- The clock speed is determined by the slowest feasible path between registers in the design
  - Often referred to as “the critical path”



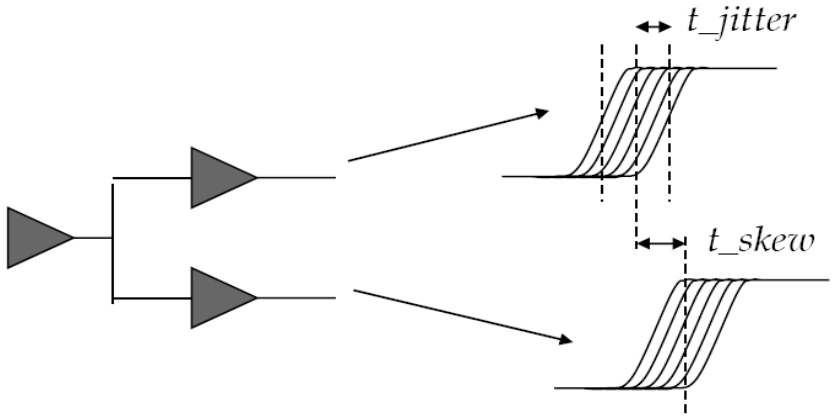
Critical path is longer with increased *logic depth* (# gates in series)

Synchronous Clock Distribution

- The goal of clock tree is for the clock to arrive at every leaf node at the same time
- Usually designed after synthesis: Matched buffer, matched capacitance loads
- Common design method:
  - “H tree”
  - Clock tree done after design
  - Balance RC delays
  - Balance buffer delays

Clock skew and jitter

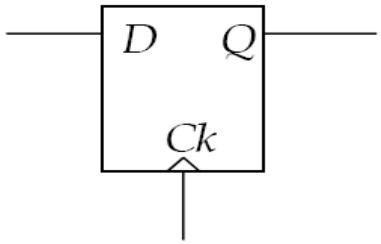
- Clock skew = systematic clock edge variation between sites
  - Mainly caused by delay variations introduced by manufacturing variations
  - Random variation
- Clock jitter = variation in clock edge timing between clock cycles
  - Mainly caused by noise

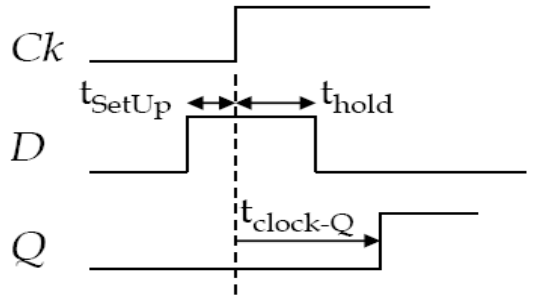


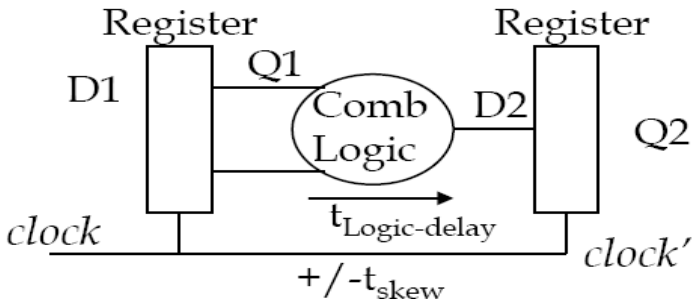
SYNOPSYS®

Flip-Flop based design

- *Edge triggered D-flip-flop*
  - Q becomes D after clock edge
- *Set-up time*
  - Data can not change no later than this point before the clock edge.
- *Hold time*
  - Data can not change during this time after the clock edge.
- *t_clock-Q*
  - Delay on output (Q) changing from positive clock edge



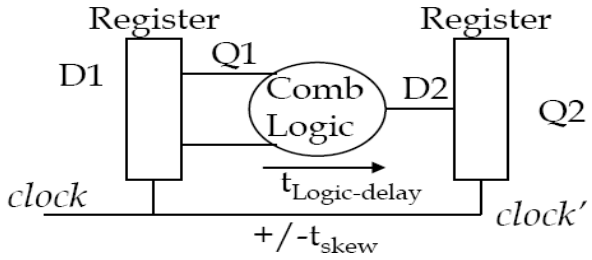




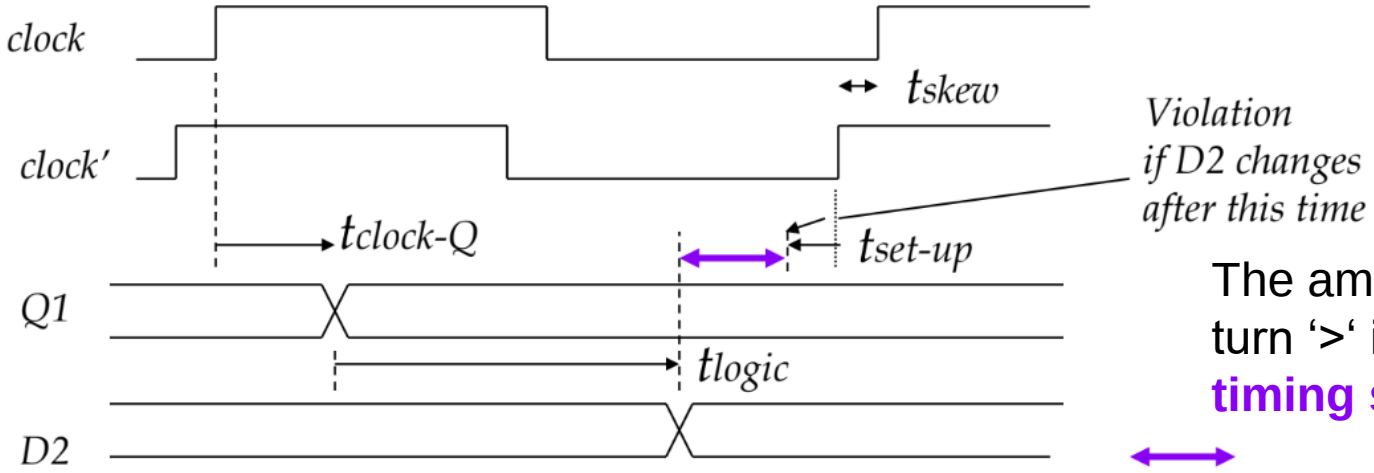
SYNOPSYS®

Preventing Set-Up Violations

- **Set-up violation:** Logic is too slow for the correct logic value to arrive at the inputs to the register on the right before one set-up time before the clock edge
- Constraint to prevent this:



$$t_{clock} \geq t_{clock-Q-\max} + t_{logic-\max} + t_{set-up} + t_{skew}$$



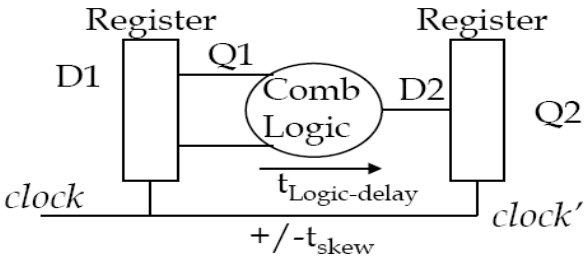
The amount of time required to turn '>' into '=' is referred to as **timing slack**

SYNOPSYS®

Preventing hold Violations

- Hold violations occur when race-through is possible
  - What is the worst case for clock'?
    - Clock' arrives after clock → more time for D2 to change
  - sometimes we have to insert logic to prevent hold violations
- Constraint to prevent hold violations:

$$t_{hold} + t_{skew} \leq t_{clock-Q-\min} + t_{\log ic-\min}$$

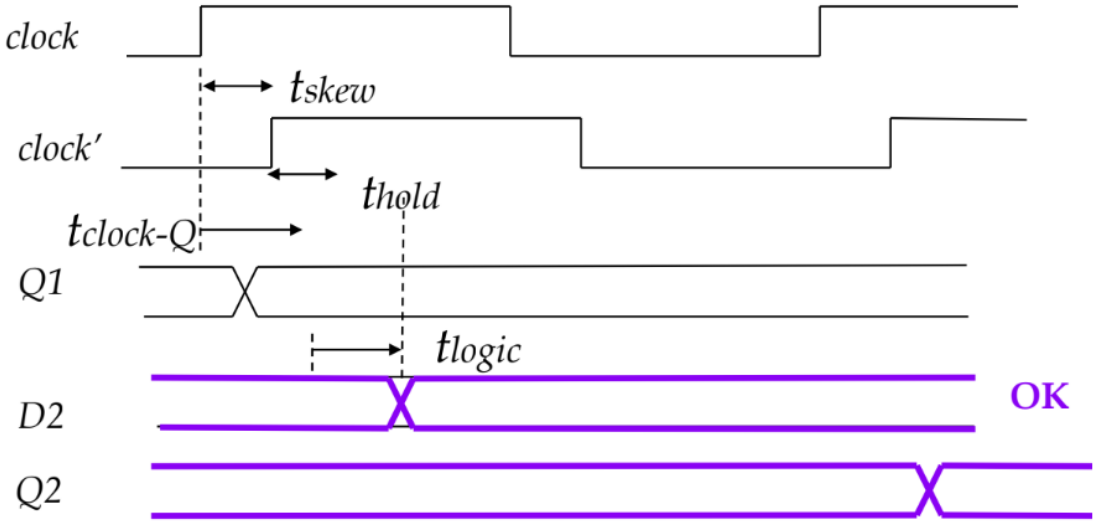


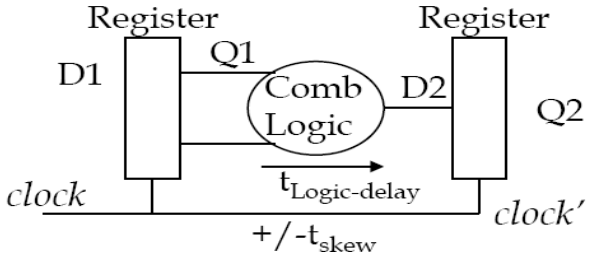
SYNOPSYS

Preventing hold Violations

- Hold violations occur when race-through is possible
  - What is the worst case for clock'?
    - Clock' arrives after clock → more time for D2 to change
  - sometimes we have to insert logic to prevent hold violations
- Constraint to prevent hold violations:

$$t_{hold} + t_{skew} \leq t_{clock-Q-\min} + t_{logic-\min}$$



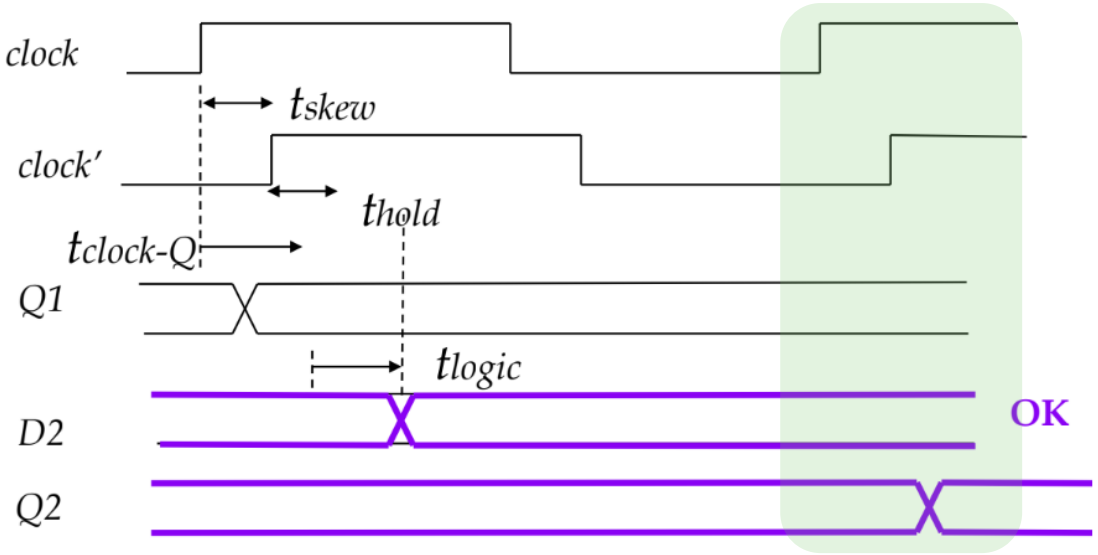


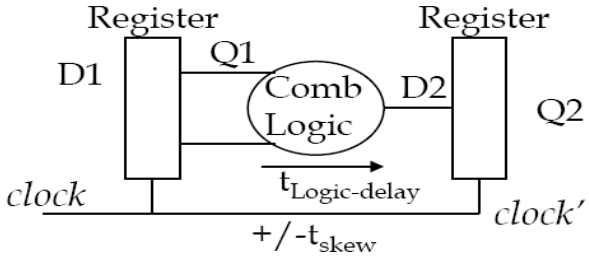
SYNOPSYS

Preventing hold Violations

- Hold violations occur when race-through is possible
  - What is the worst case for clock'?
    - Clock' arrives after clock → more time for D2 to change
  - sometimes we have to insert logic to prevent hold violations
- Constraint to prevent hold violations:

$$t_{hold} + t_{skew} \leq t_{clock-Q-\min} + t_{logic-\min}$$



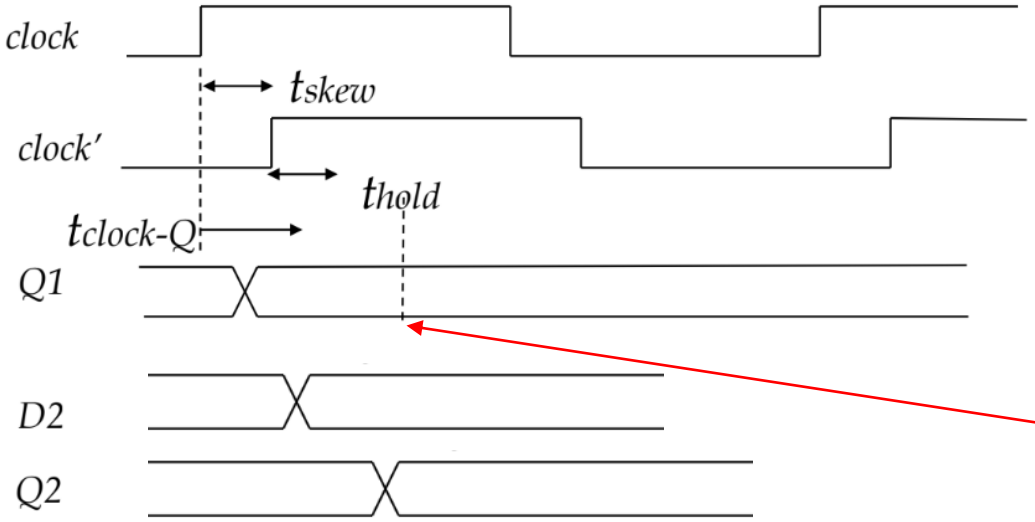


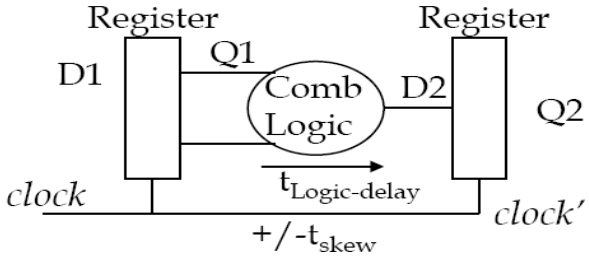
SYNOPSYS

Preventing hold Violations

- Hold violations occur when race-through is possible
  - What is the worst case for clock'?
    - Clock' arrives after clock → more time for D2 to change
  - sometimes we have to insert logic to prevent hold violations
- Constraint to prevent hold violations:

$$t_{hold} + t_{skew} \leq t_{clock-Q-\min} + t_{logic-\min}$$



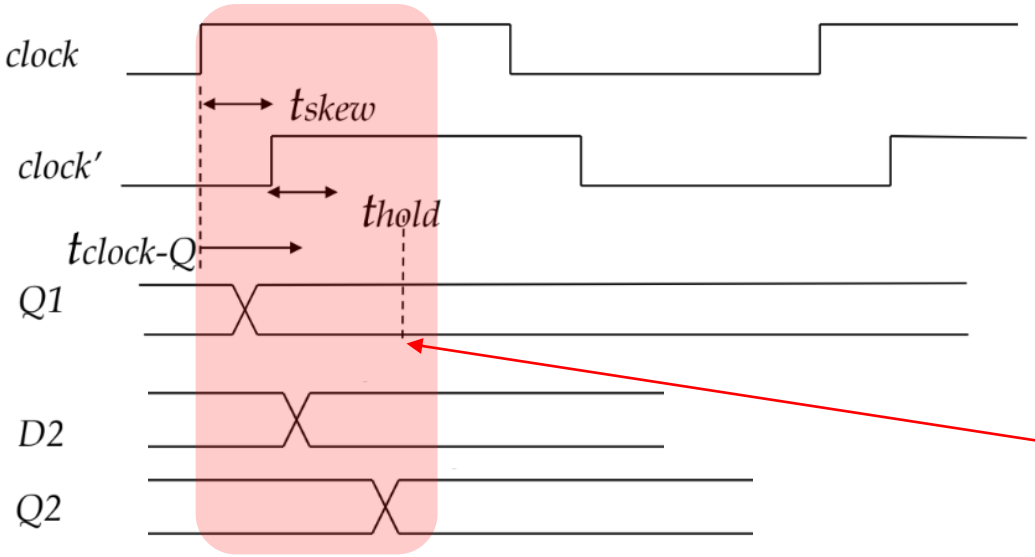


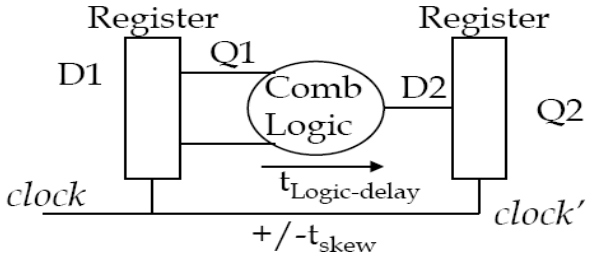
If D2 changes before this time:
Incorrect Q2 or race through

Preventing hold Violations

- Hold violations occur when race-through is possible
  - What is the worst case for clock'?
    - Clock' arrives after clock → more time for D2 to change
  - sometimes we have to insert logic to prevent hold violations
- Constraint to prevent hold violations:

$$t_{hold} + t_{skew} \leq t_{clock-Q-\min} + t_{logic-\min}$$

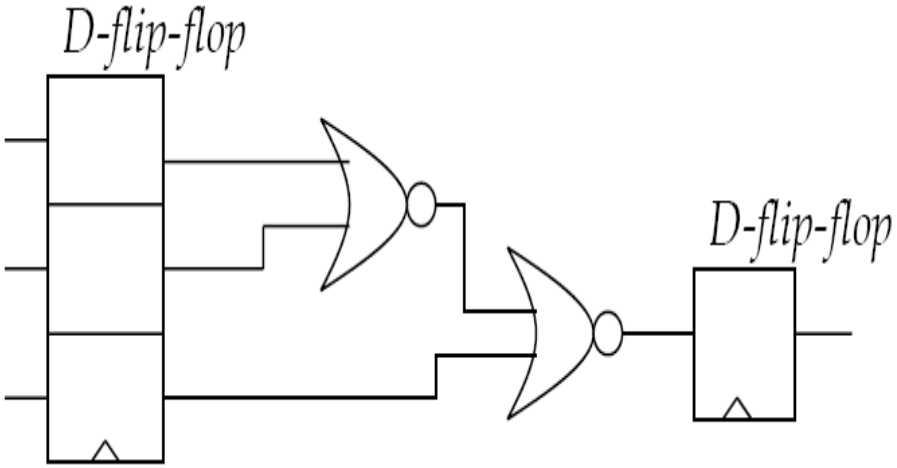




If D2 changes before this time:
Incorrect Q2 or race through

Preventing set-up/hold Violations

- Example: what is the fastest clock frequency? Is there potential for a hold violation?



min : typ : max

Tclock-Q = 3 : 4 : 5

TNOR = 1 : 2 : 3

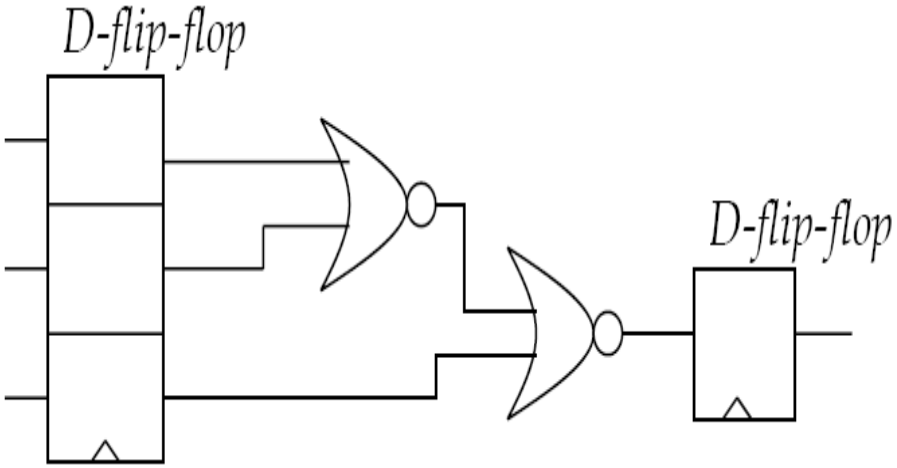
Tsu_max = 1

Thold_max = 2

Tskew = 1 ns

Preventing set-up/hold Violations

- Example: what is the fastest clock frequency? Is there potential for a hold violation?

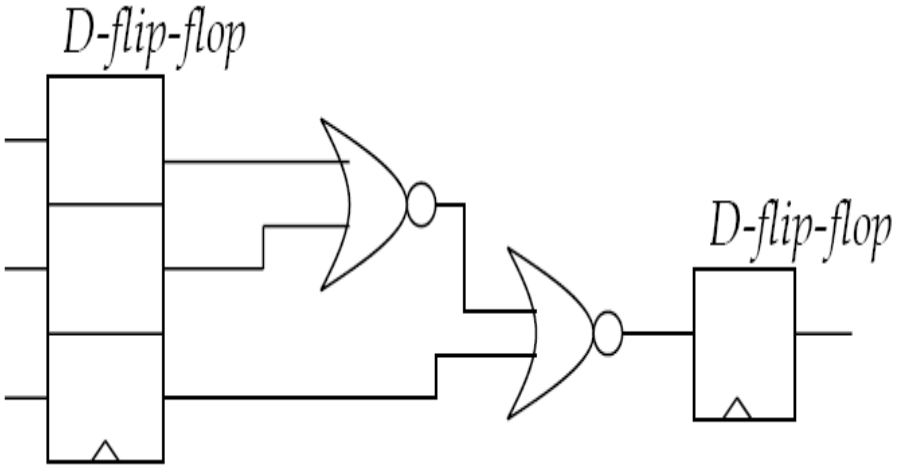


min : typ : max
Tclock-Q = 3 : 4 : 5
TNOR = 1 : 2 : 3
Tsu_max = 1
Thold_max = 2
Tskew = 1 ns

Setup: $t_{clock} \geq t_{clock\text{-}Q\text{-}max}+t_{logic\text{-}max}+t_{setup}+t_{skew}$
$= 5+(3+3)+1+1$ ns $= 13$ ns → 76MHz

Preventing set-up/hold Violations

- Example: what is the fastest clock frequency? Is there potential for a hold violation?



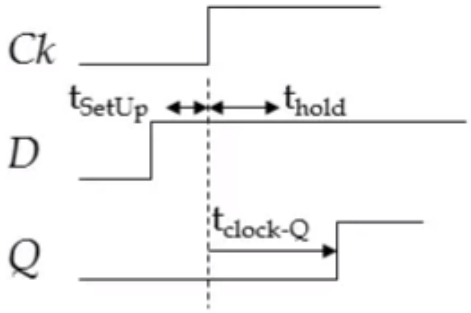
min : typ : max
Tclock-Q = 3 : 4 : 5
TNOR = 1 : 2 : 3
Tsu_max = 1
Thold_max = 2
Tskew = 1 ns

Setup: $t_{clock} \geq t_{clock\text{-}Q\text{-}max} + t_{logic\text{-}max} + t_{setup} + t_{skew}$
$= 5+(3+3)+1+1$ ns $= 13$ ns → 76MHz

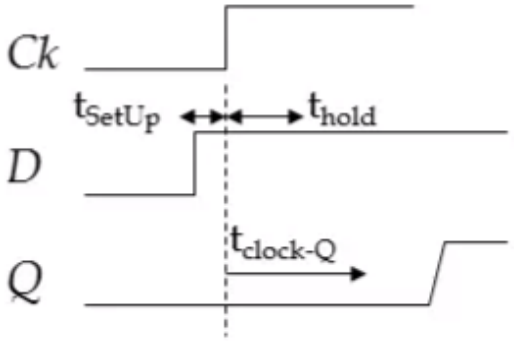
Hold: Is $t_{hold} + t_{skew} \leq t_{clock\text{-}Q\text{-}min} + t_{logic\text{-}min}$?
$2 + 1 \leq 3 + 1$ → No we don't have a hold violation

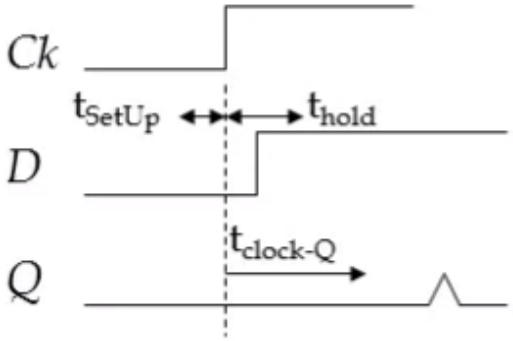
What can happen with a Timing Violation

- D changes outside setup & hold → $t_{clock\text{-}Q}$ is correct



- D changes during setup & hold → $t_{clock\text{-}Q}$ is longer than specified, or Q doesn't transition correctly

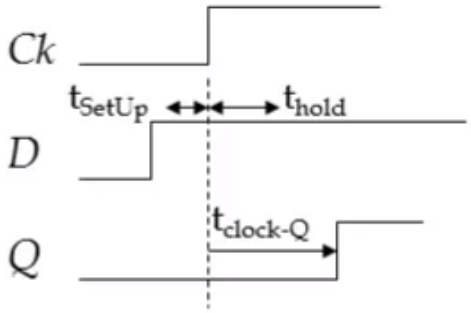




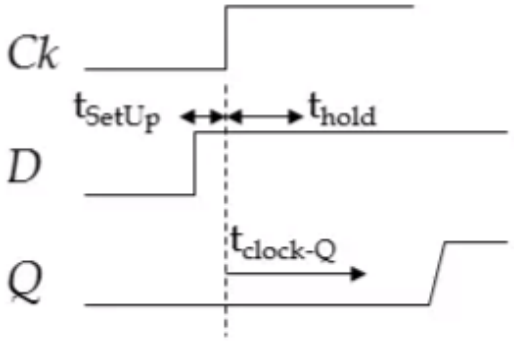
SYNOPSYS®

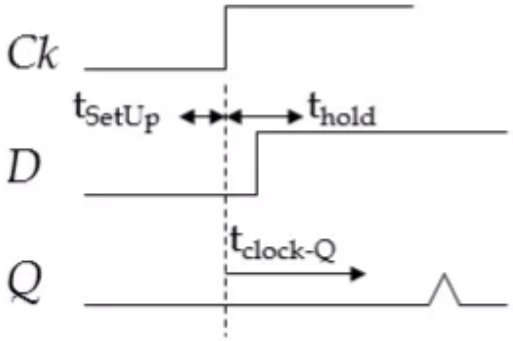
What can happen with a Timing Violation

- D changes outside setup & hold → $t_{clock\text{-}Q}$ is correct



- D changes during setup & hold → $t_{clock\text{-}Q}$ is longer than specified, or Q doesn't transition correctly

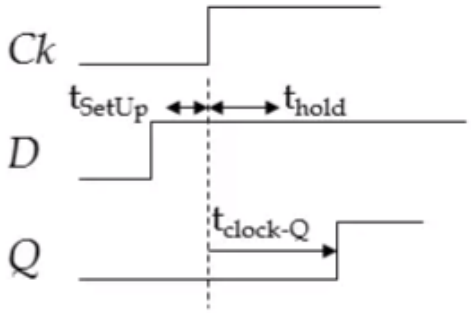




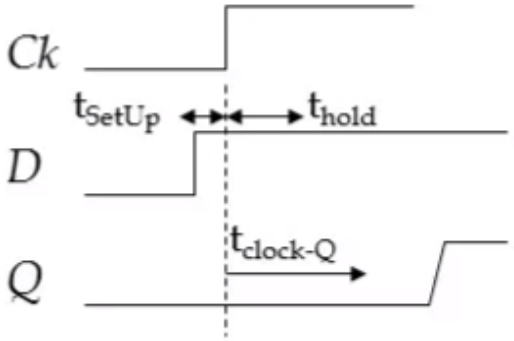
Incorrect timing in the next logic stage

What can happen with a Timing Violation

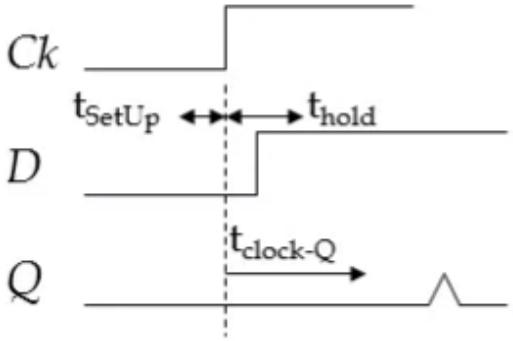
- D changes outside setup & hold → $t_{clock\text{-}Q}$ is correct



- D changes during setup & hold → $t_{clock\text{-}Q}$ is longer than specified, or Q doesn't transition correctly



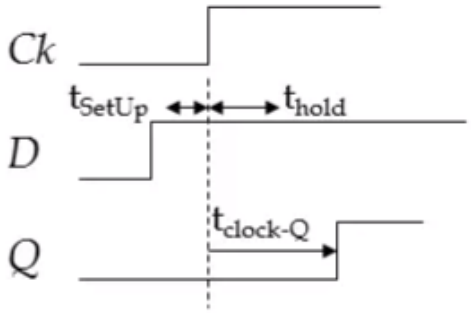
Incorrect timing in the next logic stage



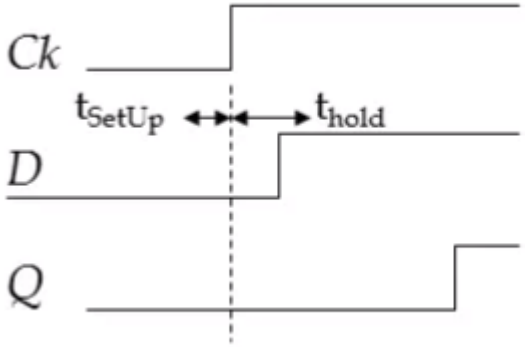
Logic Failure

What can happen with a Timing Violation

- D changes outside setup & hold → $t_{clock\text{-}Q}$ is correct



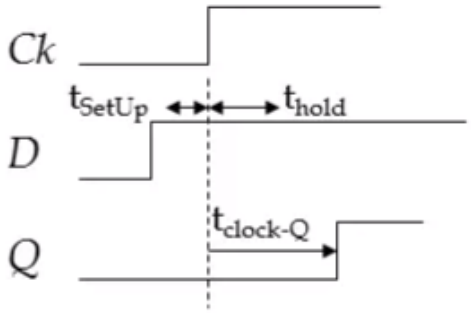
- D changes during hold



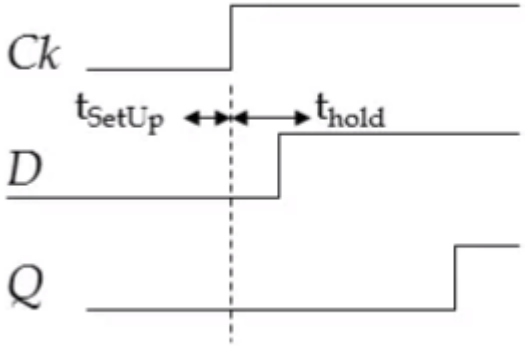
SYNOPSYS®

What can happen with a Timing Violation

- D changes outside setup & hold → $t_{clock\text{-}Q}$ is correct



- D changes during hold



Q correct, wrong cycle
Racethrough

Timing Tables

Cell Description

The DFF cell is a positive-edge triggered, static D-type flip-flop.

Function Table

D	CK	Q[n+1]	QN[n+1]
0	↑	0	1
1	↑	1	0
x	↓	Q[n]	QN[n]

Timing Tables

Timing Constraints at 25°C, 1.8V, Typical Process

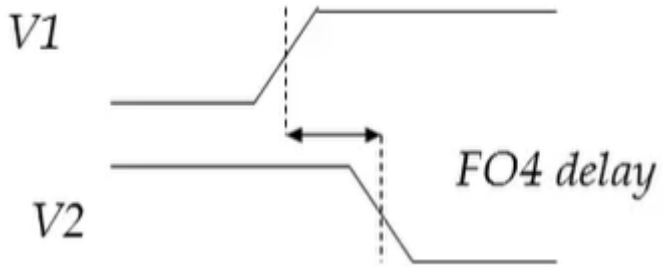
Pin	Requirement	Interval (ns)			
		XL	X1	X2	X4
D	setup↑ → CK	0.04	0.06	0.06	0.06
	setup↓ → CK	0.10	0.15	0.15	0.14
	hold↑ → CK	-0.02	-0.04	-0.04	-0.03
	hold↓ → CK	0.01	-0.04	-0.04	-0.04
CK	minpwh	0.18	0.18	0.18	0.18
	minpwl	0.25	0.25	0.25	0.25

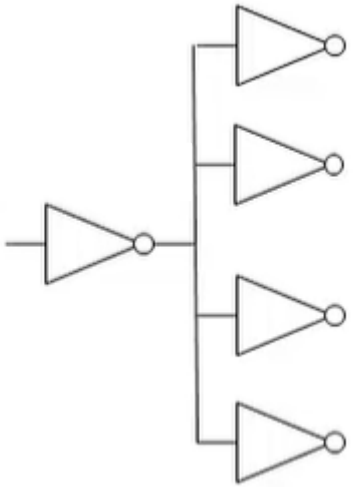
Delays at 25°C, 1.8V, Typical Process

Description	Intrinsic Delay (ns)				K_{load} (ns/pF)			
	XL	X1	X2	X4	XL	X1	X2	X4
CK → Q↑	0.219	0.197	0.180	0.155	5.696	4.288	2.261	1.139
CK → Q↓	0.179	0.166	0.153	0.132	3.406	2.469	1.234	0.611
CK → QN↑	0.238	0.221	0.196	0.179	6.239	4.518	2.259	1.138
CK → QN↓	0.304	0.280	0.251	0.219	3.342	2.444	1.219	0.605

Delay Metric

- Usual Metric for delay:
  - Fanout of 4 inverter delay: FO4
- Estimating FO4:
  - Typical ~ 360 x L_{gate} (ps)
  - Worst Case ~ 600 x L_{gate} (ps)
  - E.g. In a 28nm process FO-4 <= 16.8 ps





SYNOPSYS®

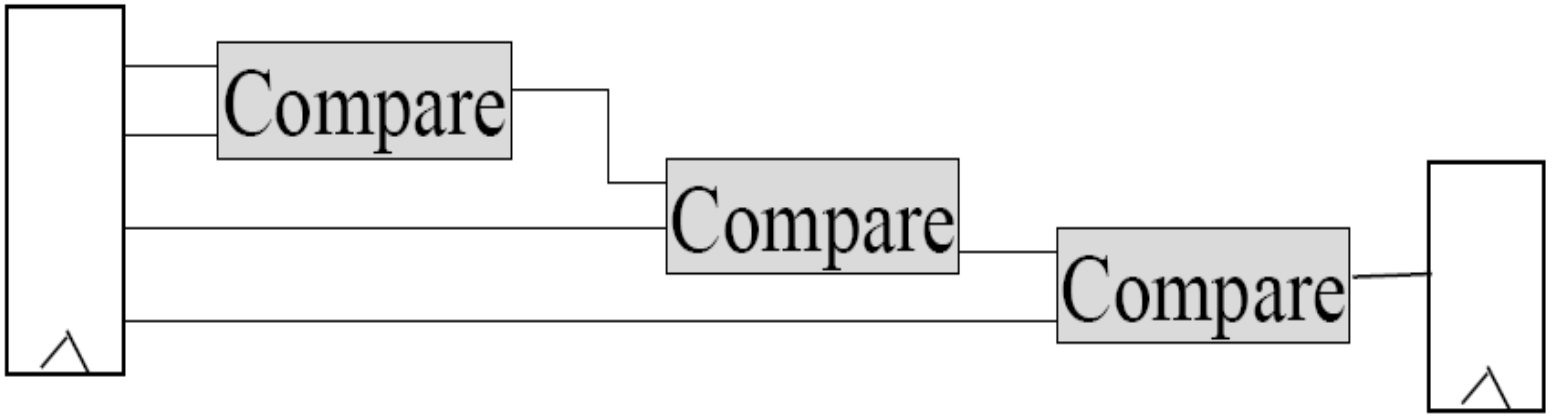
Delay Metric

- Usual Metric for delay:
  - Fanout of 4 inverter delay: FO4
- Estimating FO4:
  - Typical ~ 360 x L_{gate} (ps)
  - Worst Case ~ 600 x L_{gate} (ps)
  - E.g. In a 28nm process FO-4 <= 16.8 ps
- Example Delays:

Inverter = FO-4
2-input NAND gate = 2FO4
1-bit adder = 10 FO4
2-input Multiplexer = 4 FO4
Flip-flop t_cp-Q = 4 FO4
Flip-flop t_su / t_h = 2 FO4
Clock skew = 4 FO4
Clock jitter = 2 FO4

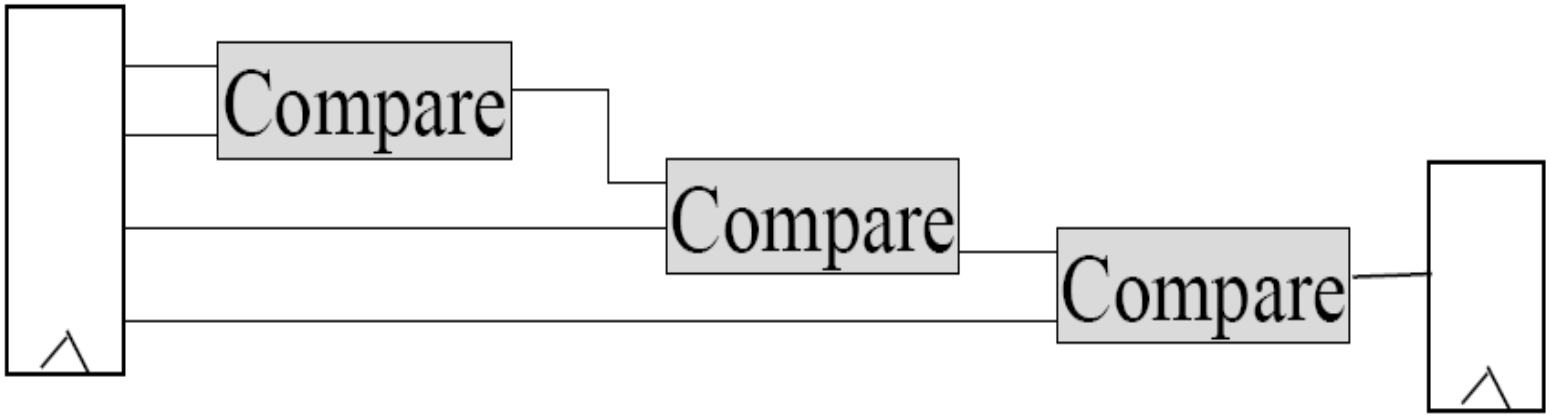
Improving Timing Performance

- Example 1 : Benefits of Pipelining and Parallelism
  - If t_comparator = 20 FO4, what is the clock period?
    (Use values on previous page)



Improving Timing Performance

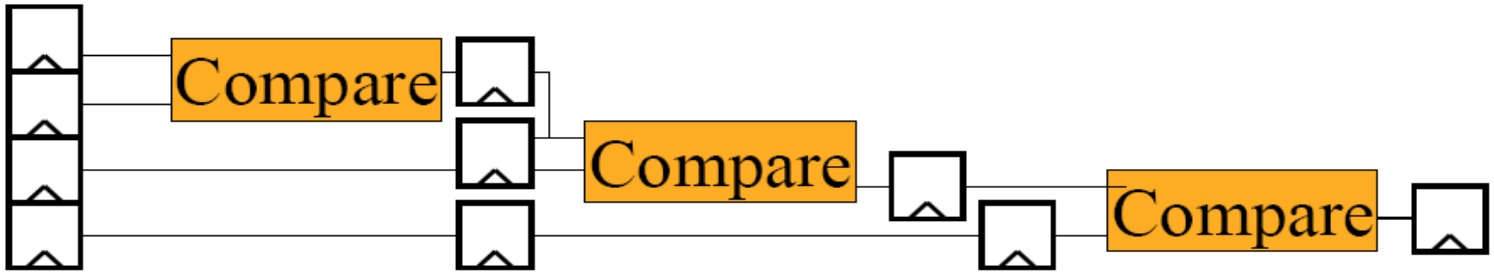
- Example 1 : Benefits of Pipelining and Parallelism
  - If t_comparator = 20 FO4, what is the clock period? (Use values on previous page)



$$T_{cp} = t_{ck\text{-}Q} + t_{logic} + t_{su} + t_{skew}$$
$$= 4 + 3*20 + 2 + 4$$
$$= 70 \text{ FO4}$$

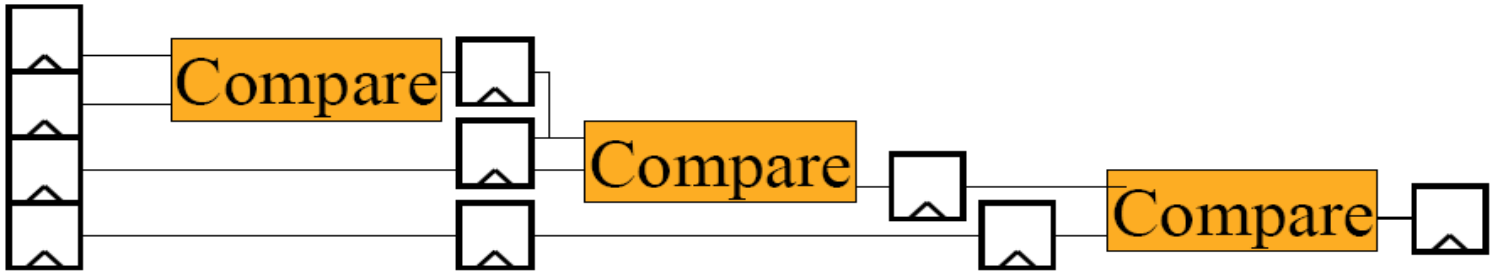
Improving Timing Performance

- Replace with a pipelined version
- Operation
  - Adding pipeline stages reduces the clock period
  - Overlapping computations at the same time
- Delay improvement?
- What is the drawback?



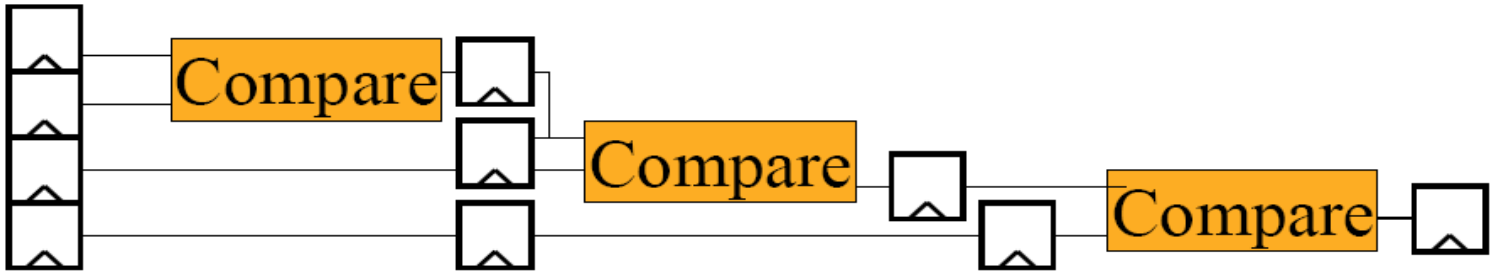
Improving Timing Performance

- Replace with a pipelined version
- Operation
  - Adding pipeline stages reduces the clock period
  - Overlapping computations at the same time
- Delay improvement? 4+20+2+4=30FO4 (2.3x not 3x)
- What is the drawback?



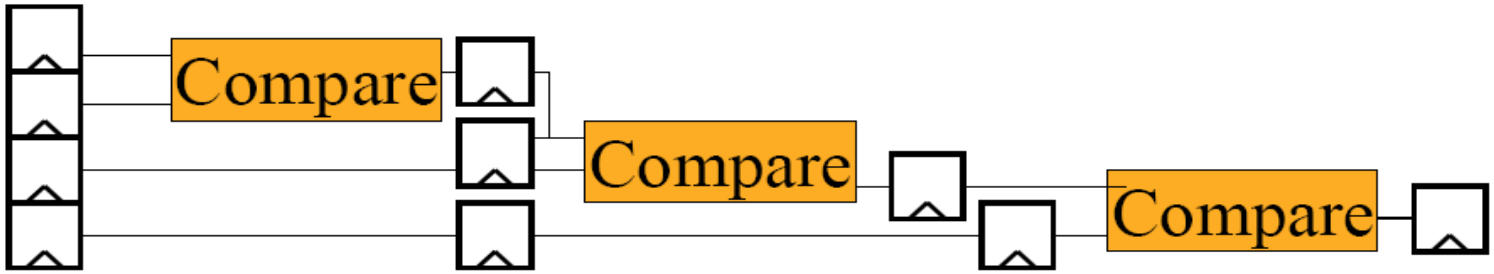
Improving Timing Performance

- Replace with a pipelined version
- Operation
  - Adding pipeline stages reduces the clock period
  - Overlapping computations at the same time
- Delay improvement? 4+20+2+4=30FO4 (2.3x not 3x)
- What is the drawback? Increased latency, additional circuit



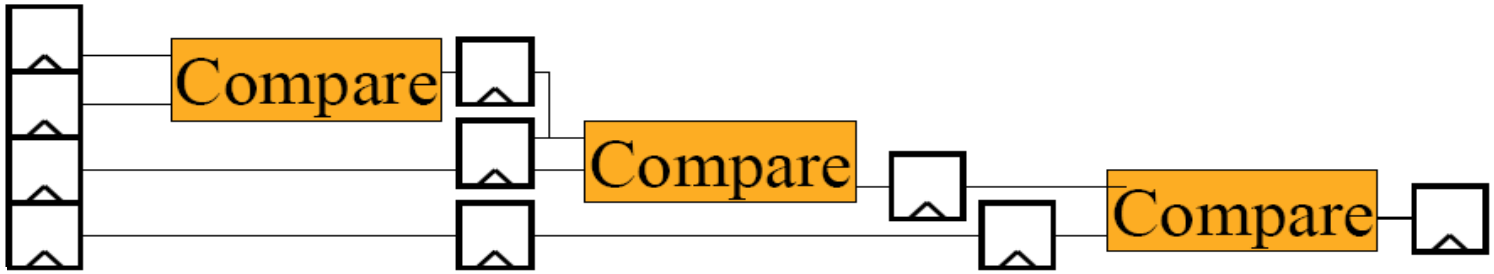
Improving Timing Performance

- Replace with a pipelined version
- Operation
  - Adding pipeline stages reduces the clock period
  - Overlapping computations at the same time
- Delay improvement? 4+20+2+4=30FO4 (2.3x not 3x)
- What is the drawback? Increased latency, additional circuit



Improving Timing Performance

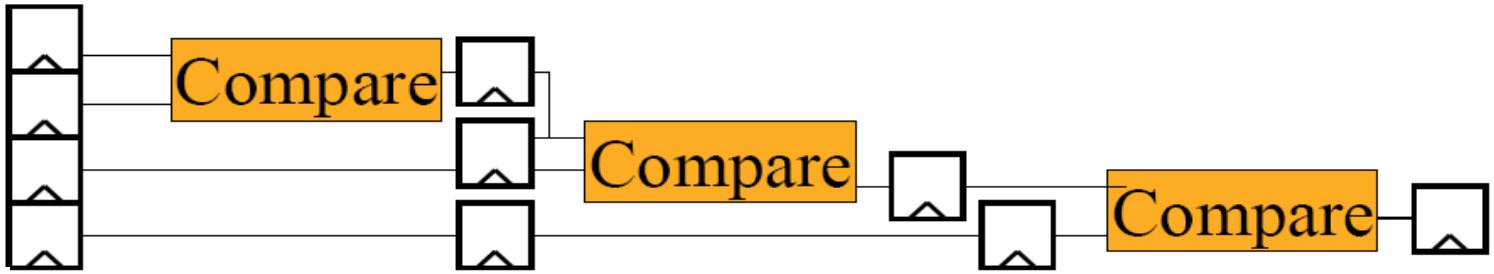
- Replace with a pipelined version
- Operation
  - Adding pipeline stages reduces the clock period
  - Overlapping computations at the same time
- Delay improvement? 4+20+2+4=30FO4 (2.3x not 3x)
- What is the drawback? Increased latency, additional circuit



A	1
1	2
C	3
D	4

Improving Timing Performance

- Replace with a pipelined version
- Operation
  - Adding pipeline stages reduces the clock period
  - Overlapping computations at the same time
- Delay improvement? 4+20+2+4=30FO4 (2.3x not 3x)
- What is the drawback? Increased latency, additional circuit

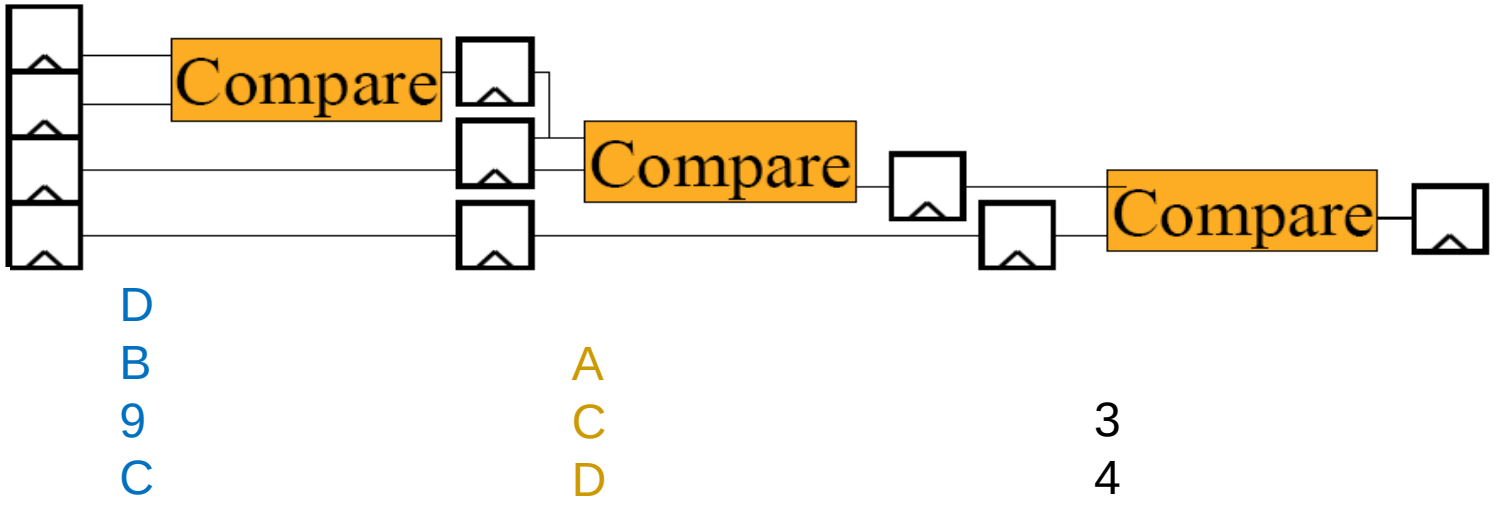


D	A	
B	1	2
9	C	3
C	D	4

SYNOPSYS®

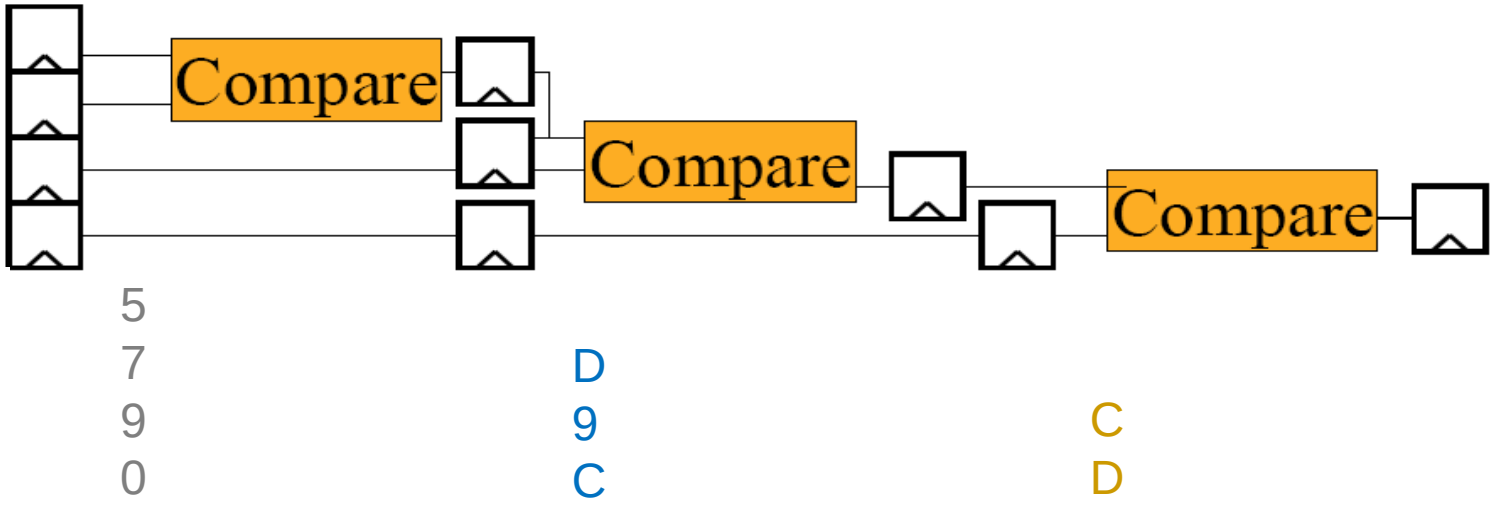
Improving Timing Performance

- Replace with a pipelined version
- Operation
  - Adding pipeline stages reduces the clock period
  - Overlapping computations at the same time
- Delay improvement? 4+20+2+4=30FO4 (2.3x not 3x)
- What is the drawback? Increased latency, additional circuit



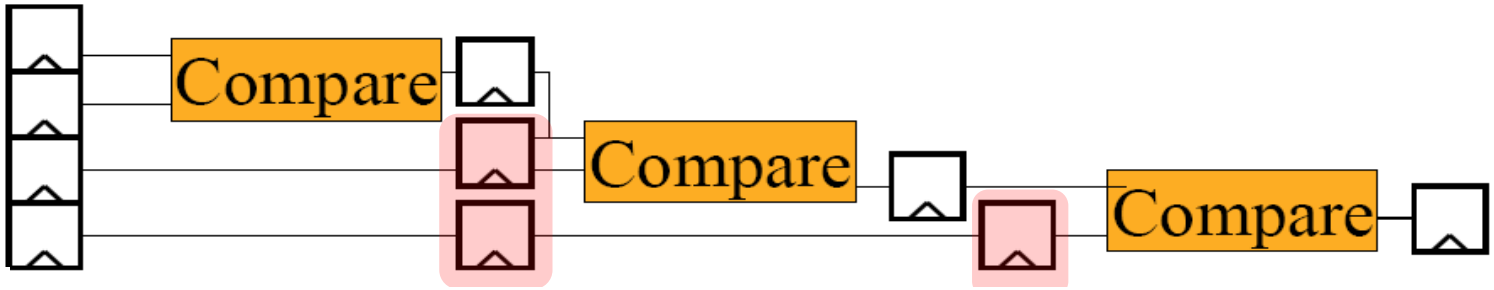
Improving Timing Performance

- Replace with a pipelined version
- Operation
  - Adding pipeline stages reduces the clock period
  - Overlapping computations at the same time
- Delay improvement? 4+20+2+4=30FO4 (2.3x not 3x)
- What is the drawback? Increased latency, additional circuit



Improving Timing Performance

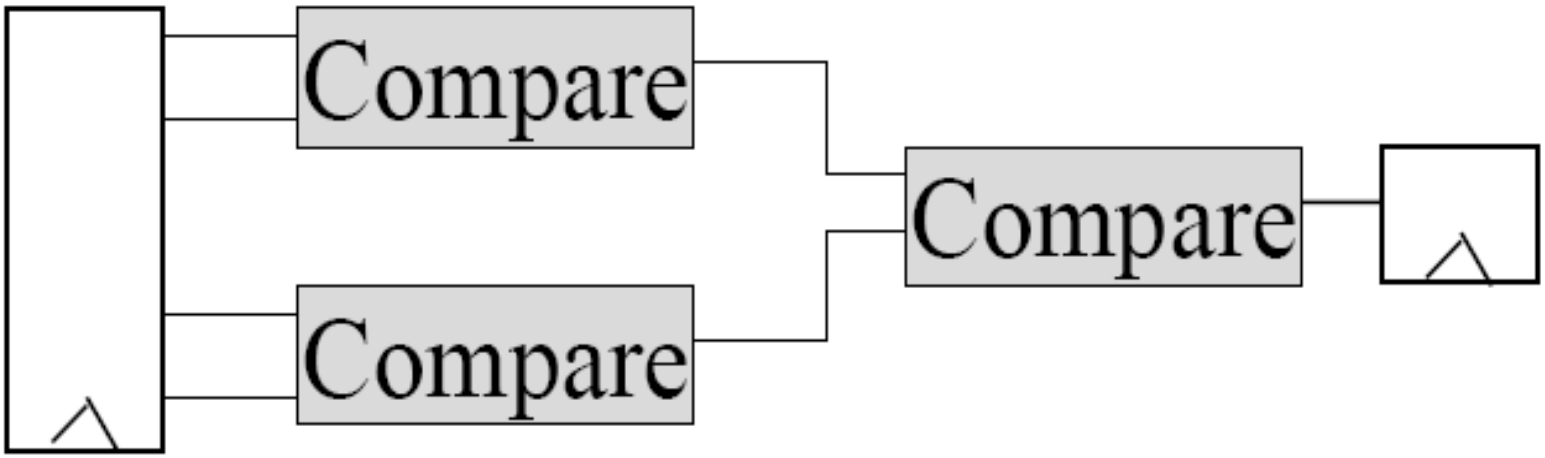
- Replace with a pipelined version
- Operation
  - Adding pipeline stages reduces the clock period
  - Overlapping computations at the same time
- Delay improvement? 4+20+2+4=30FO4 (2.3x not 3x)
- What is the drawback? Increased latency, additional circuit



Keep synchronicity at each stage they keep the different waves (colors) together

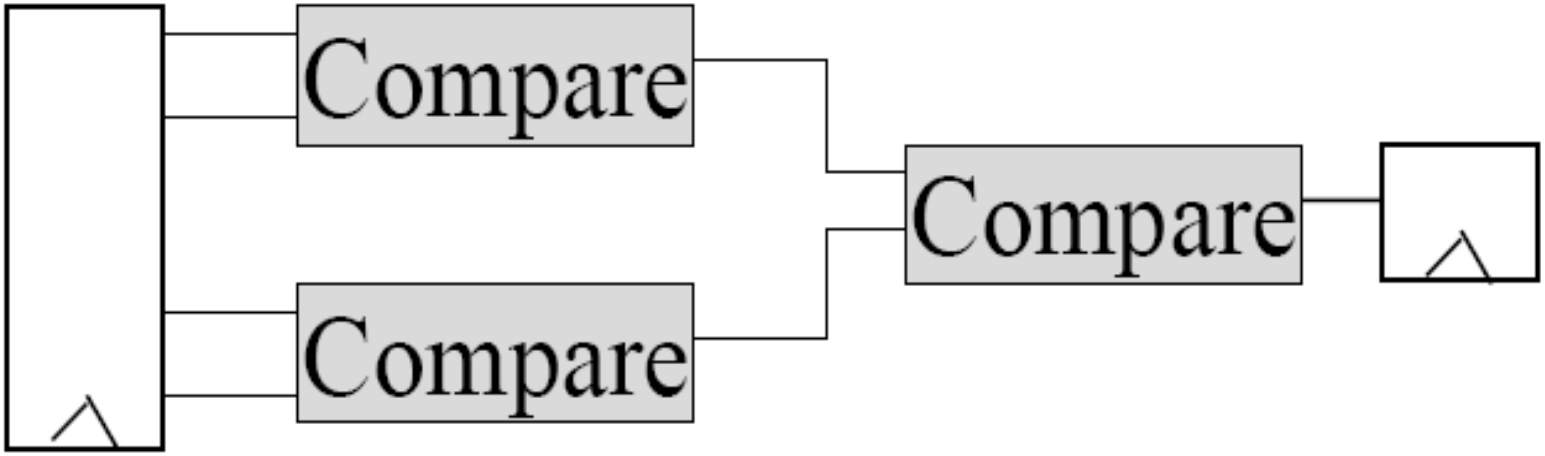
Improving Timing Performance

- Replace with:
  - Parallelism at logic level
- Clock Period?
  -



Improving Timing Performance

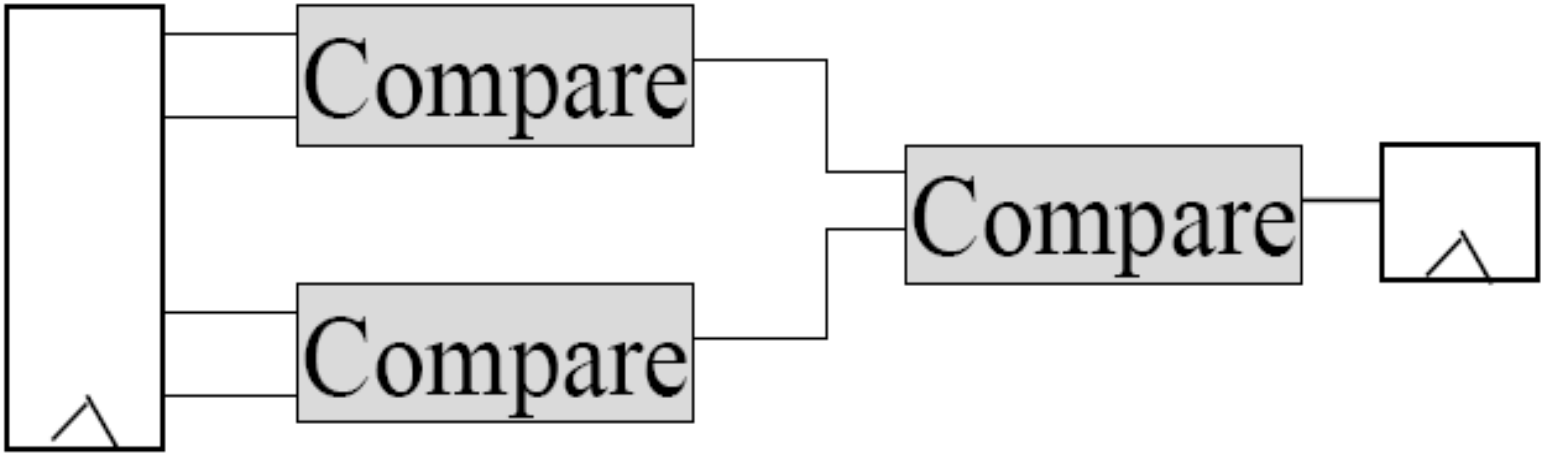
- Replace with:
  - Parallelism at logic level
- Clock Period? 4 + 2*20 + 2 + 4 = 50 FO4
  -



SYNOPSYS®

Improving Timing Performance

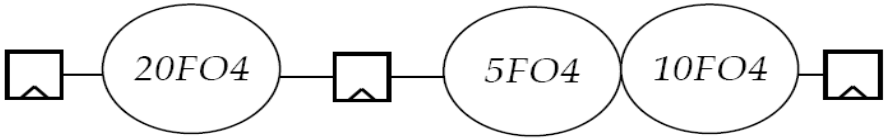
- Replace with:
  - Parallelism at logic level
- Clock Period? 4 + 2*20 + 2 + 4 = 50 FO4
  - No increase in area



Retiming

- Impact of critical paths can often be reduced by retiming or rebalancing a design
  - Before:

  - After:

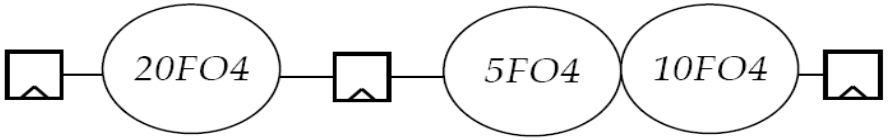


Retiming

- Impact of critical paths can often be reduced by retiming or rebalancing a design
  - Before:

$T_{cp} = 4 + (20 + 5) + 2 + 4 = 35$ FO4

  - After:

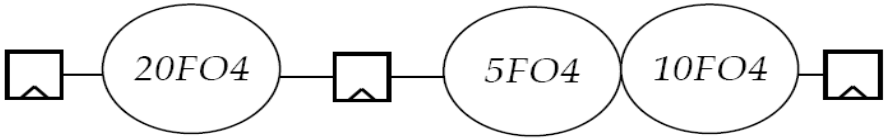


Retiming

- Impact of critical paths can often be reduced by retiming or rebalancing a design
  - Before:

$T_{cp} = 4 + (20 + 5) + 2 + 4 = 35$ FO4

  - After:



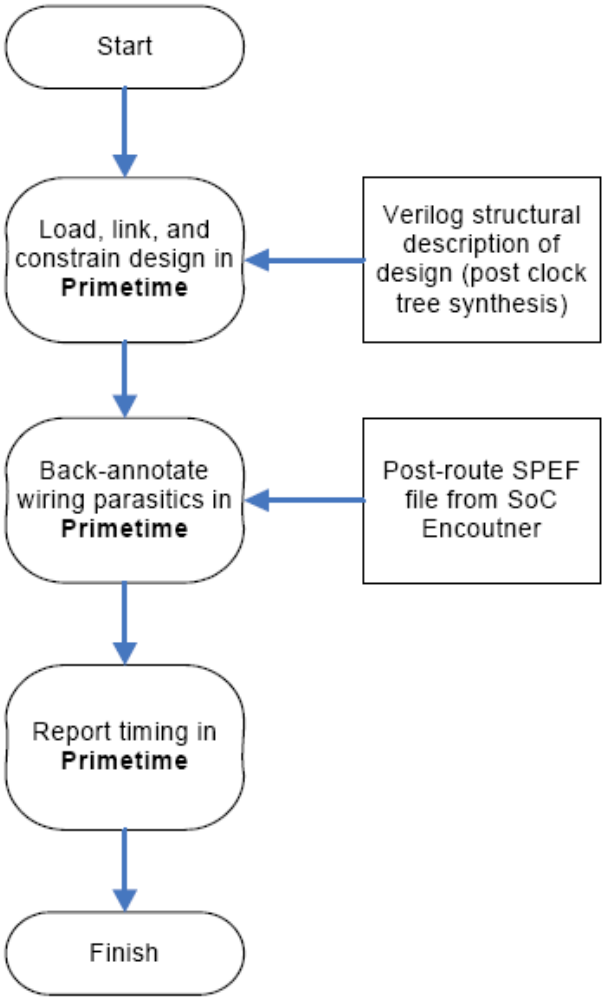
$T_{cp} = 4 + (20) + 2 + 4 = 30$ FO4

Timing in CAD Flow

- During synthesis
  - Tool calculates path delays under worst-case delay conditions
  - Determines critical path
  - Moves logic to a faster path if setup violation predicted
- After synthesis:
  - Perform Static Timing Analysis
  - Determine no setup violations exist under worst case conditions
  - Determine no hold violations exist under best case conditions
- After place and route:
  - Perform Timing Analysis
  - i.e. Run timing verification tools on netlist with actual delays
  - Back-annotate actual delays to netlist from later tools

Initial Delay Estimation Flow

- Primetime: Gate-level static timing analysis tool
  - Report timing for critical path
- Back-annotate wire parasitics for more accuracy
  - SPEF file from place and route tool



How to Design Complex Digital Systems

Steps in High Level Design

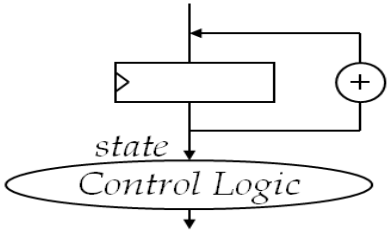
- Determine MicroOperations to be performed on datapath units
  - e.g. adds, subtracts, multiplies, memory references, etc.
- Design datapath units to perform these operations efficiently
  - Design to RTL level
- Identify control points
  - Control lines
  - Status lines
- Determine reset/start/stop/transition actions
  - Especially global reset strategy

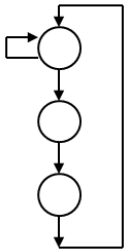
Steps in High Level Design

- Determine control sequence
  - Generally MicroOp sequence required to perform overall task
  - Gives sequence of control events and status line responses
- Determine control strategy
  - Mix of FSMs and/or counters
- Verify before coding

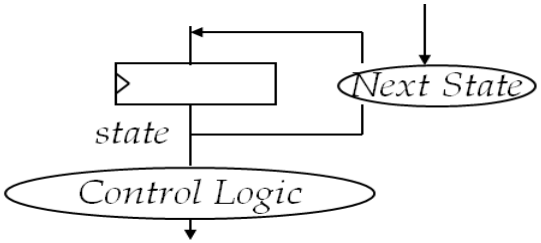
Control Strategies

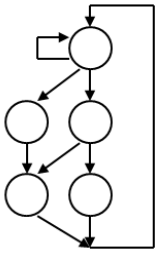
- Counters:
  - Takes machine through a linear sequence of states with few decisions along the way





- FSMs
  - Permits branches in control decision chain

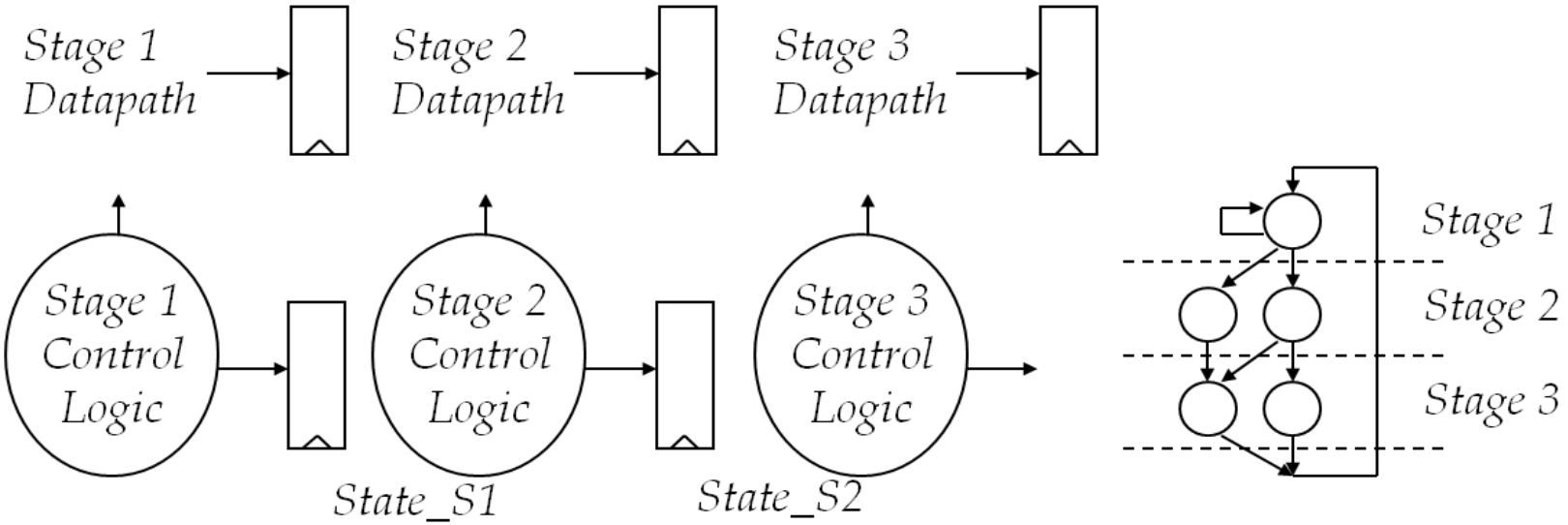




FSD

Control Strategies

- Pipeline Control
  - In a sense, an “unrolled” FSM – each stage does one step (or one of several parallel steps) in an FSM; state information communicated between stages



SYNOPSYS®

Reset

- Reset is a global signal that the designer can not modify
- It is generally asserted on power up or a “hard” reset
- It is used to start the machine in a “known” state
- Thus it must be distributed to
  - All FSMs
  - Selected counters
  - Selected status registers

Achieving Efficiency

- High level tradeoffs:
  - Parallelism
  - Pipelining
  - Optimizing the critical resource
    - E.g. Memory bandwidth
  - Keep resources busy
    - If a resource is idle can it be shared?
    - Goal : Everything is used every clock cycle
- Algorithmic Optimizations
  - E.g. Algorithms that avoid DRAM accesses
    - e.g. Compress table onto SRAM
  - Exploiting common algorithms in Computer Science
    - e.g. Hash tables for matches
    - e.g. Shift instead of *2 /2

Achieving Efficiency

- Think hardware (area and delay):
  - Avoid large FSMs
  - Count the large units (* + memories, etc.)
  - Avoid high-fanout signals
  - Avoid priority logic
  - Structure arithmetic for speed
    - E.g. CLA instead of ripple carry
- Exploit existing Intellectual Property

Achieving Efficiency

- Synopsys, and others, provide libraries of carefully optimized design blocks to use – called “Design Ware”
  - Libraries include: Arithmetic, Advanced Math, DSP, Control, Sequential, and Fault Tolerant
  - For +,-,*, >=, <=, >, and <, design ware is automatically used
  - More complex cells must be inferred via a procedure call, e.g. cosine

Achieving Efficiency

- Synopsys, and others, provide libraries of carefully optimized design blocks to use – called “Design Ware”

```
module trigger (angle, cos_out);
    parameter wordlength1 = 8, wordlength2 = 8;
    input [wordlength-1:0] angle;
    output [wordlength-1:0] cos_out;
    // passes the widths to the cos function
    parameter angle_width = wordlength1, cos_width = wordlength2;
    `include “synopsys_syn/dw/sim_ver/DW02_cos_function.inc”
    wire [wordlength2-1:0] cos_out;
    // infer DW02_cos
    assign cos_out = cos(angle);
endmodule
```

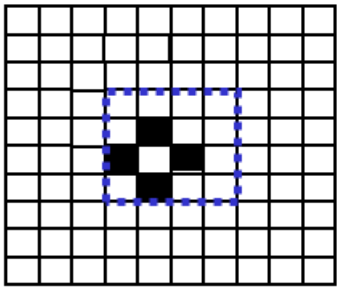
Complex Design Example

“Mantras”

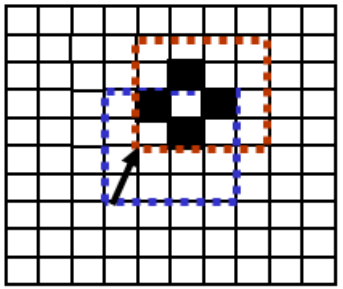
1. One clock, one edge, flip-flops only
2. Design BEFORE coding
3. Behavior implies function
4. Clearly separate control and datapath

Design Example

- Motion Estimator
- Task: Detect blocks of video data in successive frames that are related only via a translation
  - Digital Video is captured as blocks of 16x16 pixels
  - Want to determine if block has moved largely unchanged
    - If true can transmit motion vector rather than block
    - Permits high level of compression
  - Example (4x4 block)



Reference Block in Frame 1



“Draw block” with motion vector (1,2) in frame 2

Search Algorithm

Describe for 16x16 reference block:

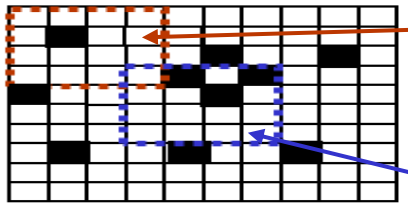
1. Move a window the size of the reference block over search space in the second frame
2. For each window location (i,j) determine the distortion vector

$$D(i,j) = \sum_{m=0}^{15}\sum_{n=0}^{15} | r_{m,n} - S_{m+i,N+j} |$$

3. Maintain the best distortion and appropriate motion vector produced so far.

- For Example (4x4 block):

Reference Block in Frame 1



Search window in frame 2

Search Block Location (i,j)=(-3,3)
D=3 (3 pixels different in this B&W example)

Original Location of Reference Block in Frame 1

System Requirements

- System Requirements:
- 16x16 Reference Block
- 31x31 Search Window
- Each stored in one two-read-ported memory
  - In reality one memory per frame
- Grey-scale coded pixels (8 bits/block)
- 4096 reference blocks in a frame
- Conduct search at 15 frames per second
  - (Encoding does not have to be real time)
- Clocks available : 130, 260 MHz
- 0.25 μm CMOS library

Step 1 : System Design

Elements to thinking:

- Bottom-up design
  - Determine critical bottlenecks (paths & other bottlenecks)
- Top-down design
  - Determine use of pipelining and parallelism to meet performance constraints

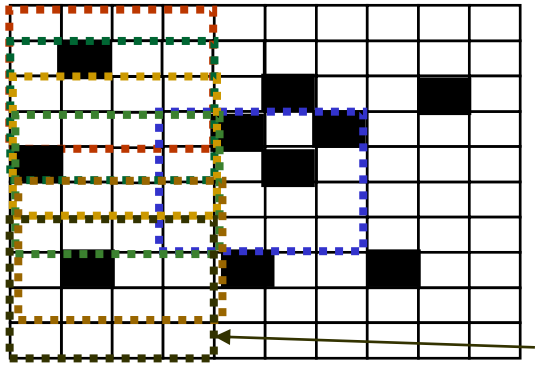
$$D = D + \mid r_{mn} - S_{m+i,N+j} \mid$$

Critical Bottlenecks:

- Elemental Arithmetic Operation (add-accumulate):
  - Design, synthesize ➔ Can operate at 260 MHz with some timing margin left over
- Memories:
  - Single access per clock cycle

… System Design

- Top Down Design
- Number of add-accumulates per clock cycle:
  - 4096 blocks per 1/15 of a second
  - (31-15)x(31-15)=256 searches/block
  - 16x16=256 add-accumulates per search
  - ➔ 4096*15*256*256 = 4.027E9 add-accumulates/second
  - At 260 MHz ➔ At least 16 adders in parallel (4027/260=15.5)
- Searches/block [(4x4) on (10x10) example]:

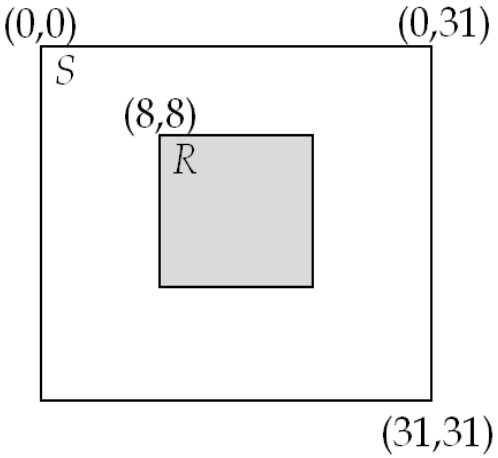


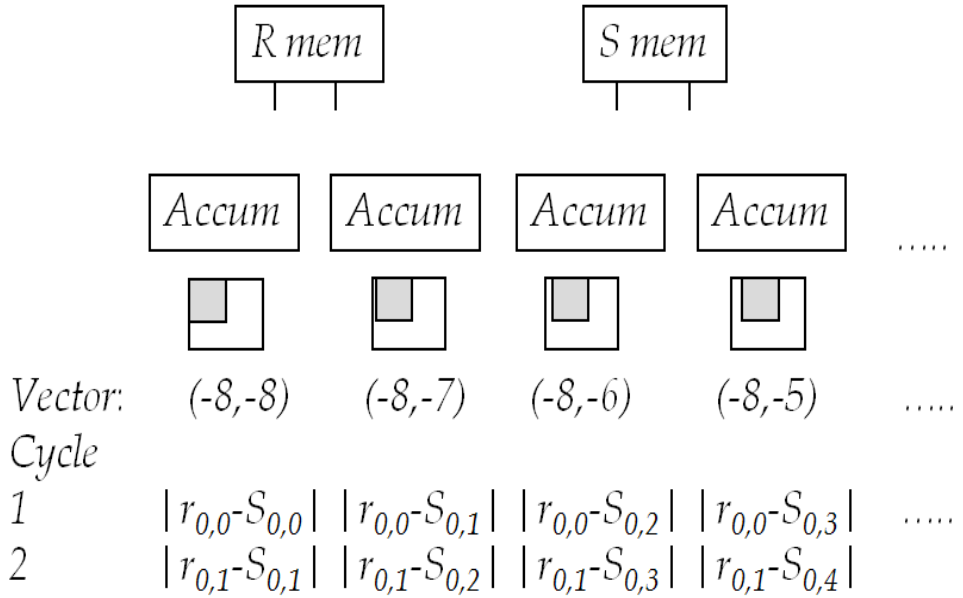
Search (-3,-3)
Search (-2,-3)
Search (-1,-3)
Search (0,-3)
Search(1,-3)
Search(2,-3)
Search(3,-3)

7 searches per column
7 searches per row
(10-3)x(10-3) total searches

…System Design

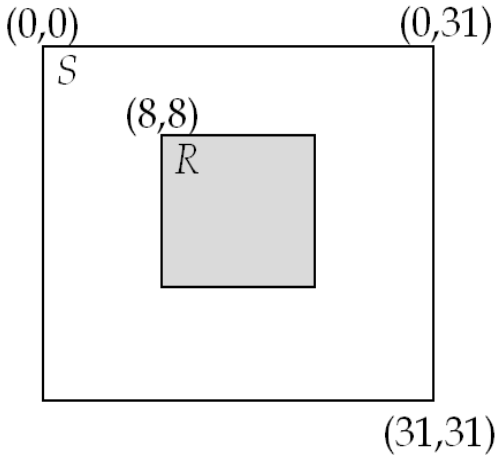
- First Attempt
- Assign one search per Accumulator

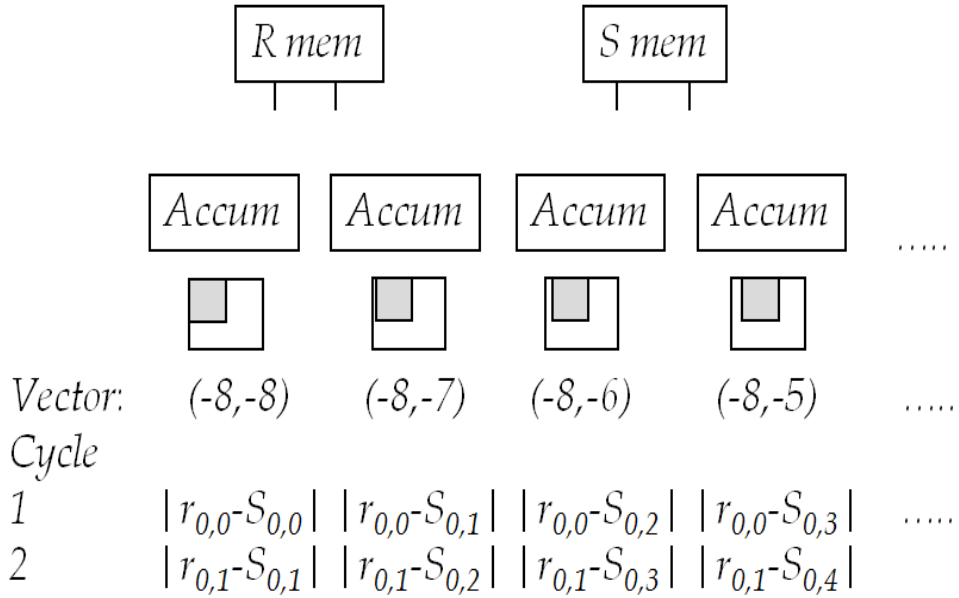




…System Design

- First Attempt
- Assign one search per Accumulator

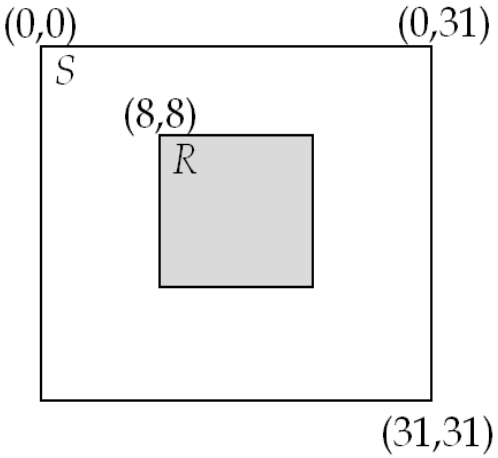


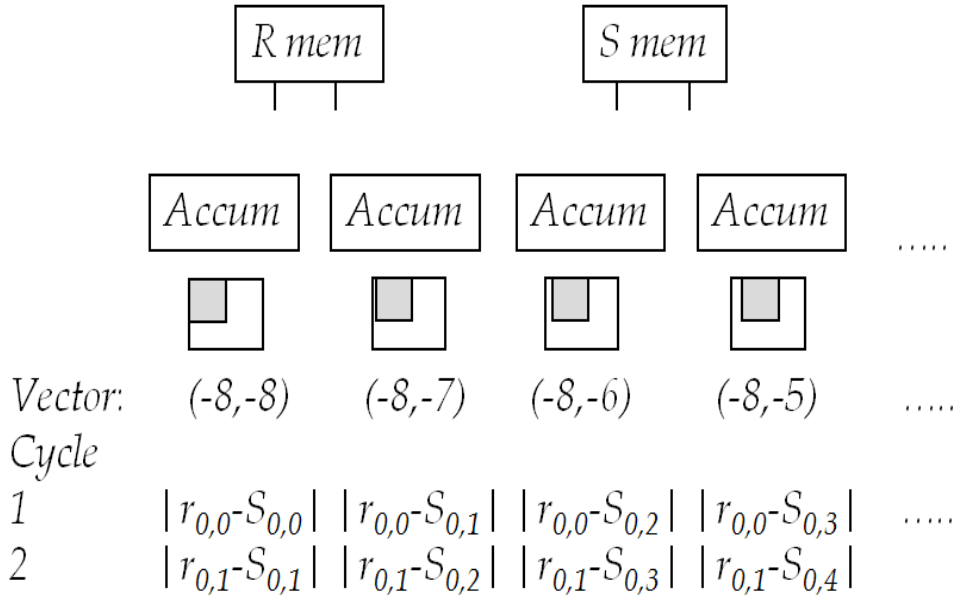


Problem: Requires 16 port S memory

… System Design

- Second Attempt:
- Stagger Startup of Accumulators

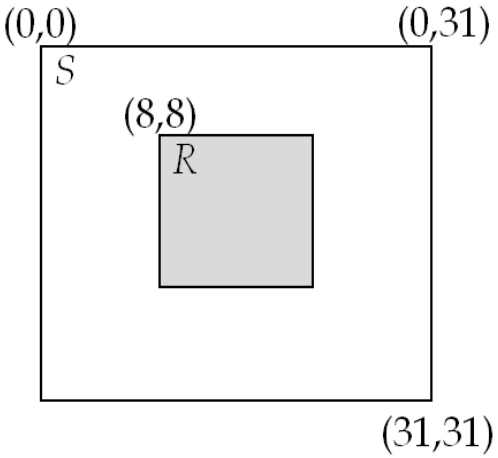


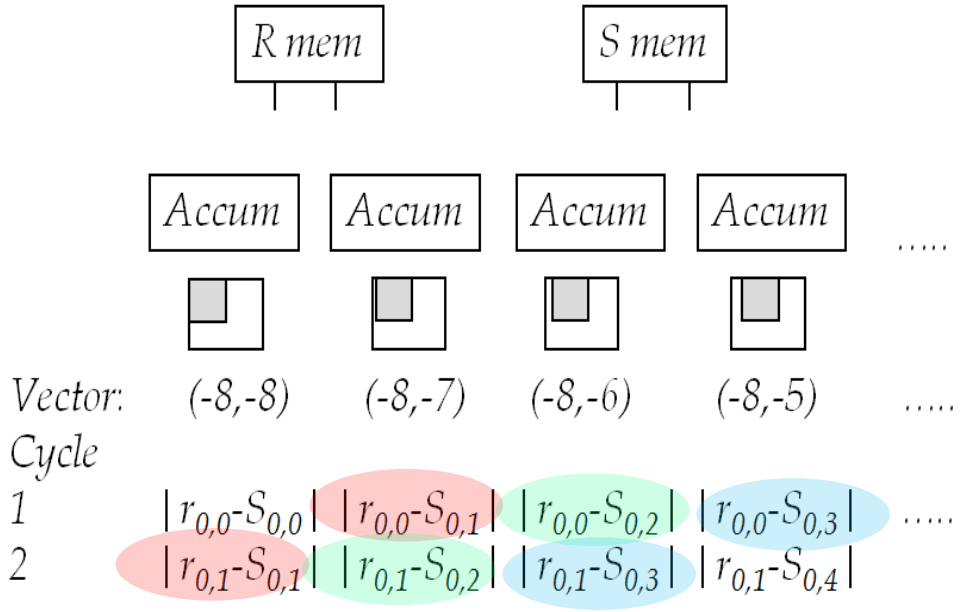


Problem: Requires 16 port S memory

… System Design

- Second Attempt:
- Stagger Startup of Accumulators

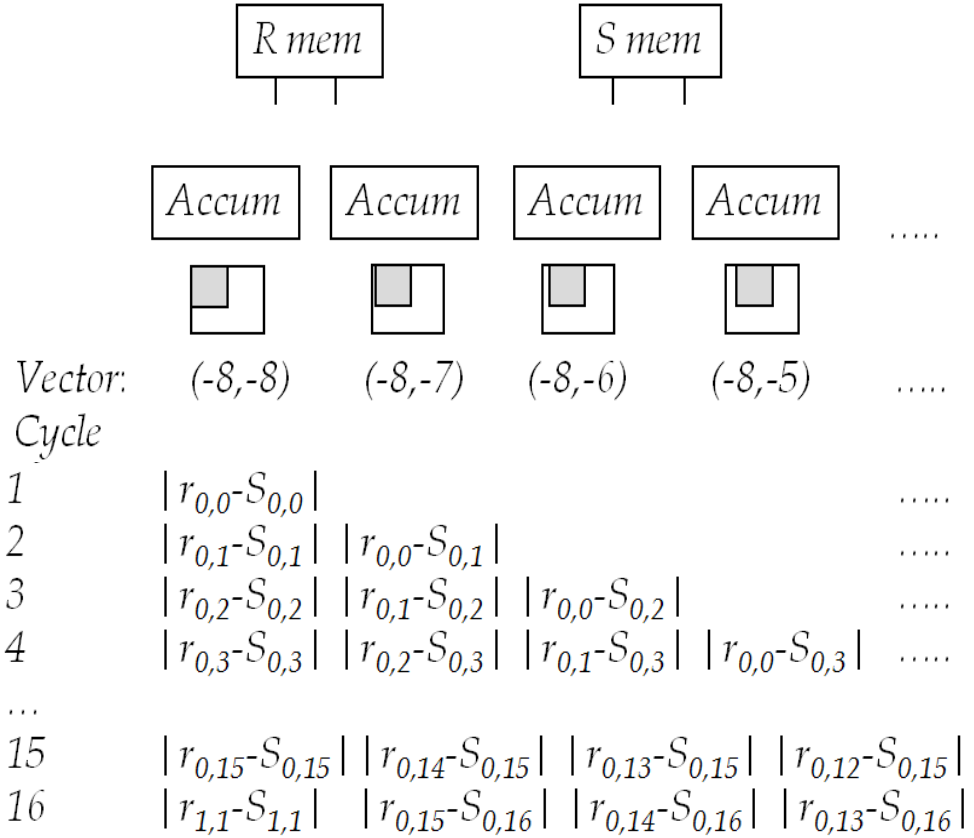




Problem: Requires 16 port S memory

… System Design

- Second Attempt:
- Stagger Startup of Accumulators

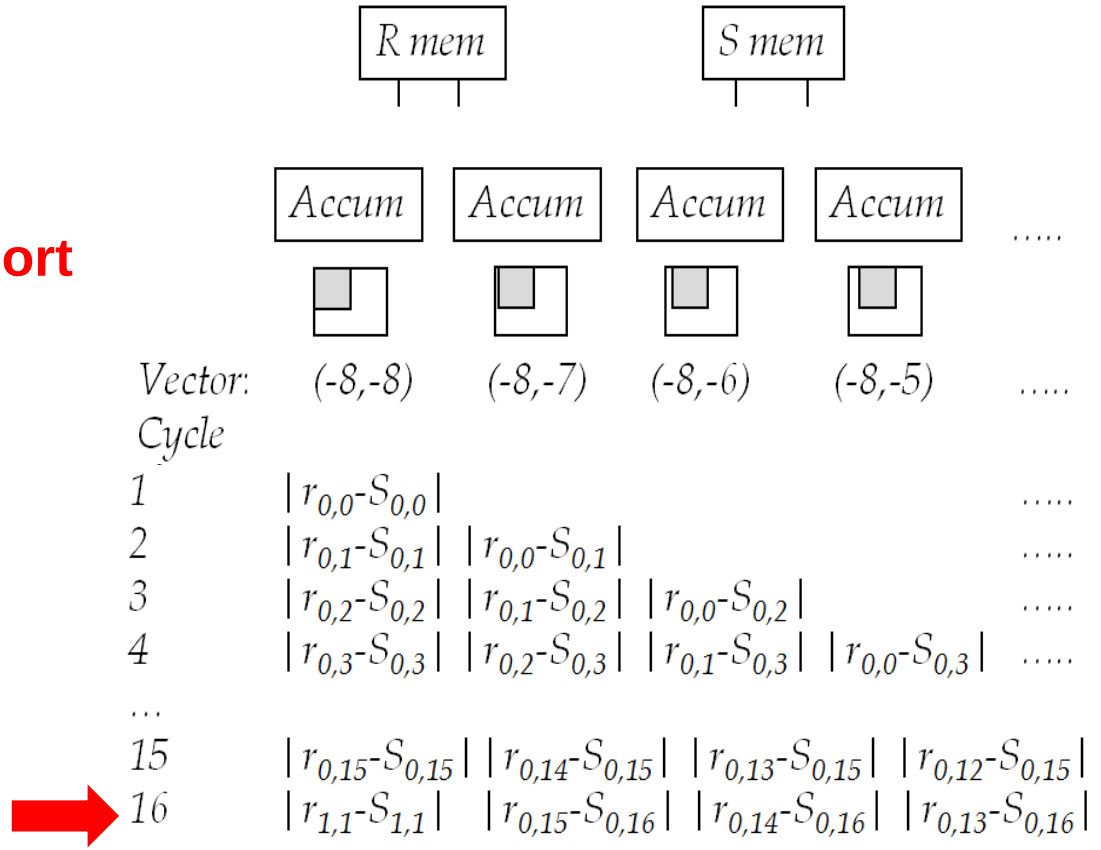


SYNOPSYS®

… System Design

- Second Attempt:
- Stagger Startup of Accumulators

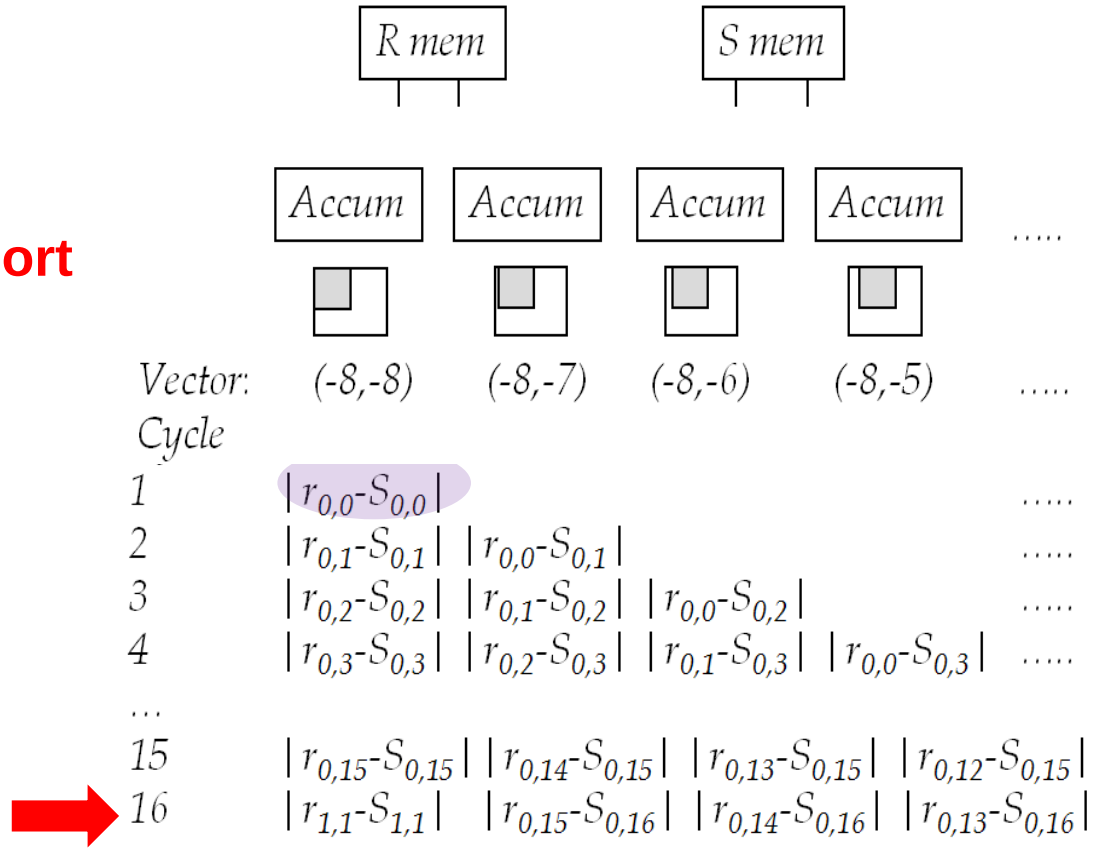
Problem: Requires 16 port R memory



… System Design

- Second Attempt:
- Stagger Startup of Accumulators

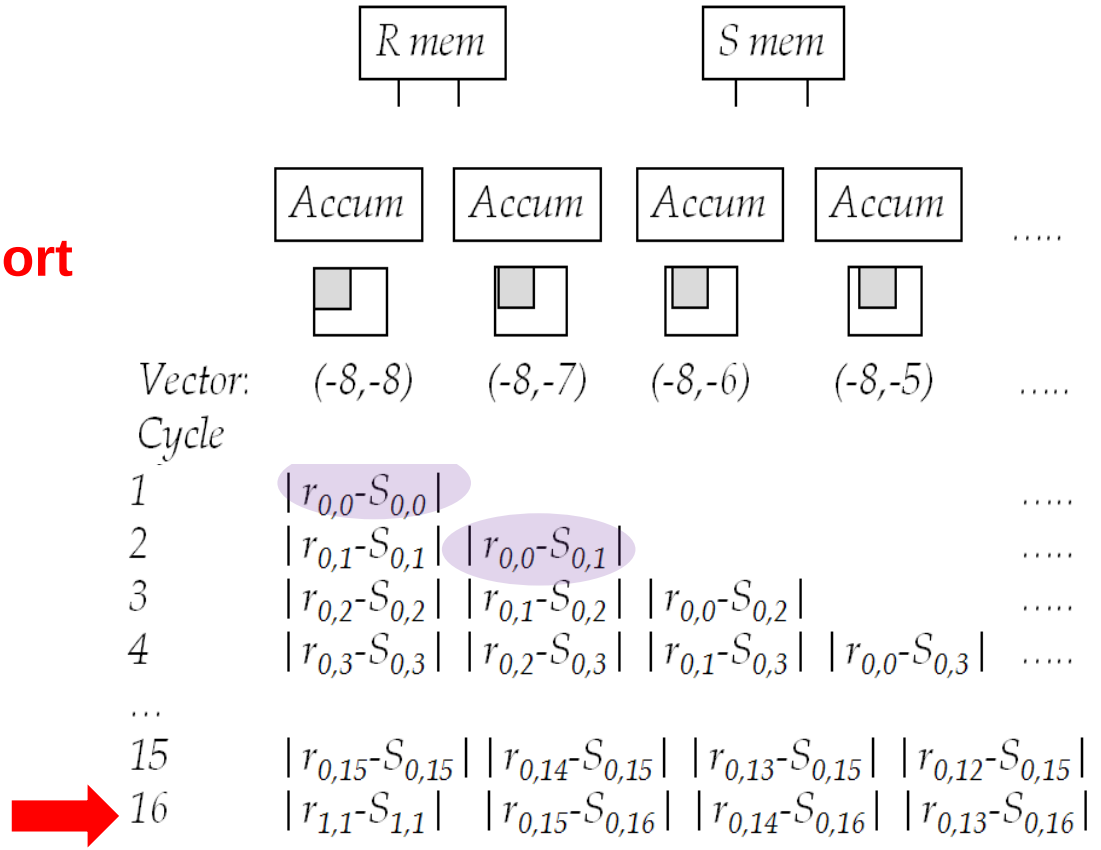
Problem: Requires 16 port R memory



… System Design

- Second Attempt:
- Stagger Startup of Accumulators

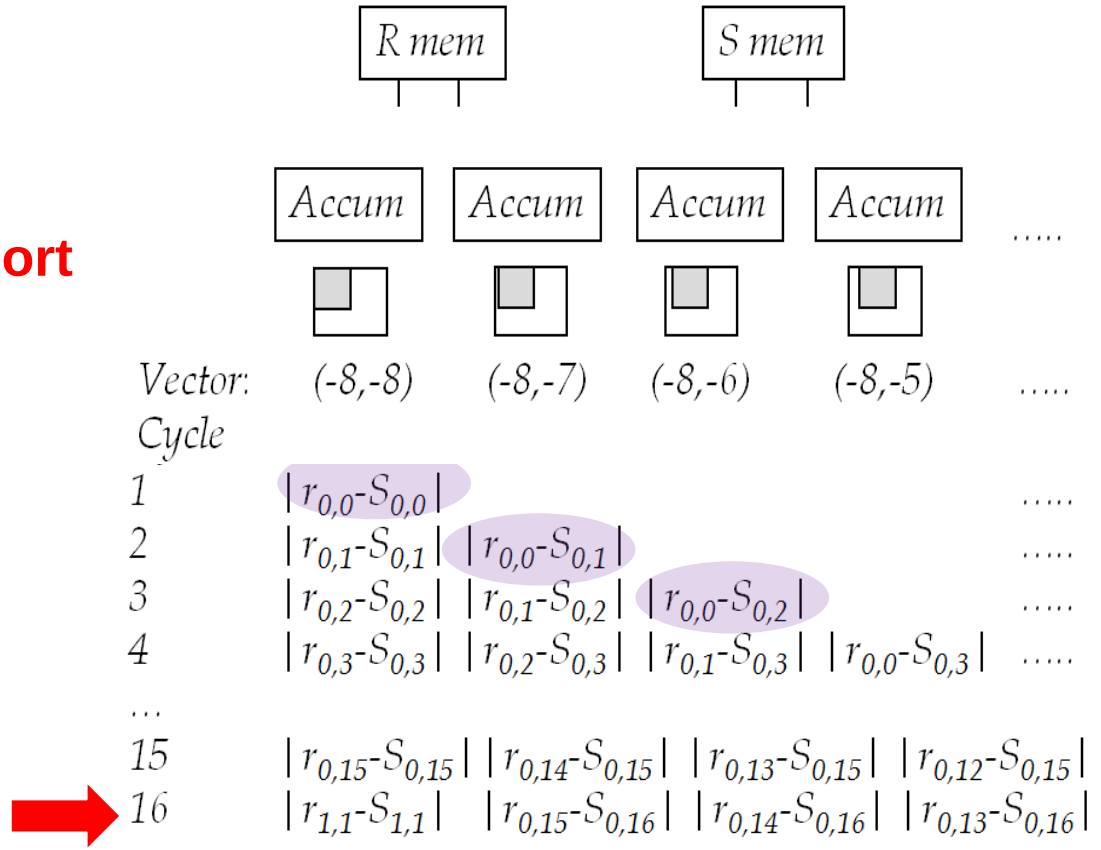
Problem: Requires 16 port R memory



… System Design

- Second Attempt:
- Stagger Startup of Accumulators

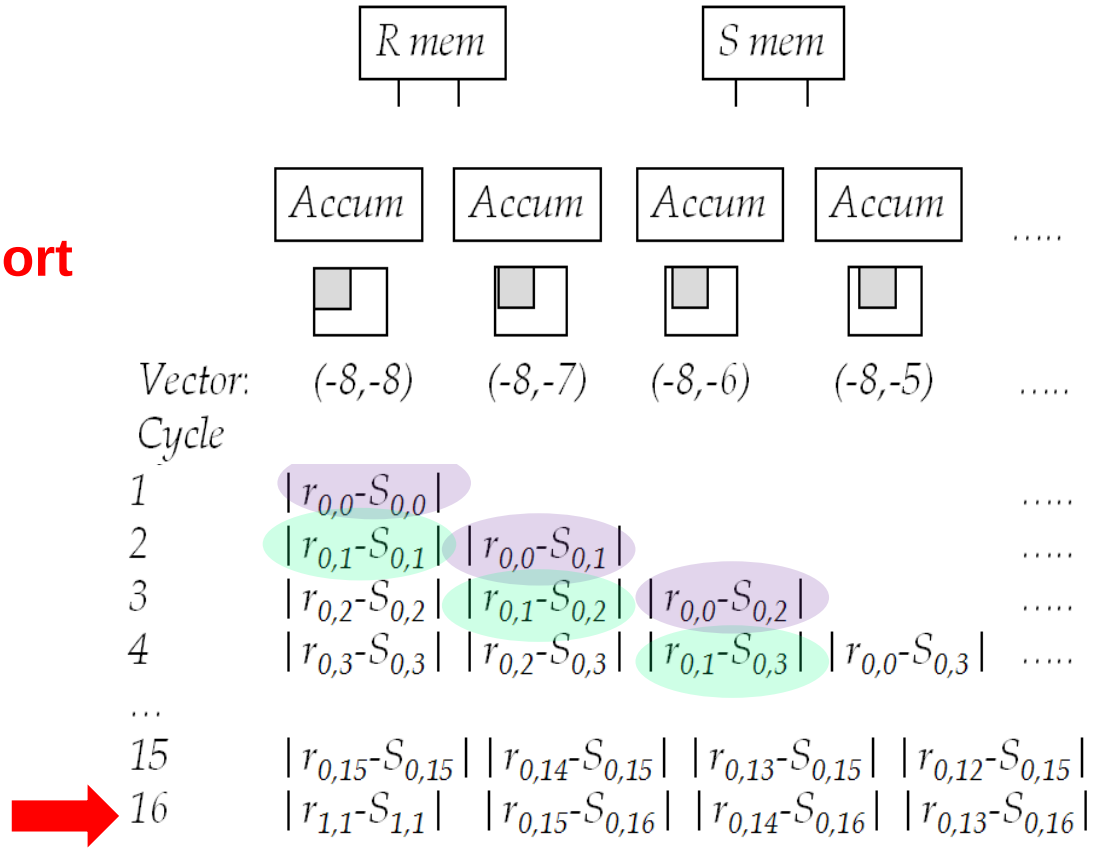
Problem: Requires 16 port R memory



… System Design

- Second Attempt:
- Stagger Startup of Accumulators

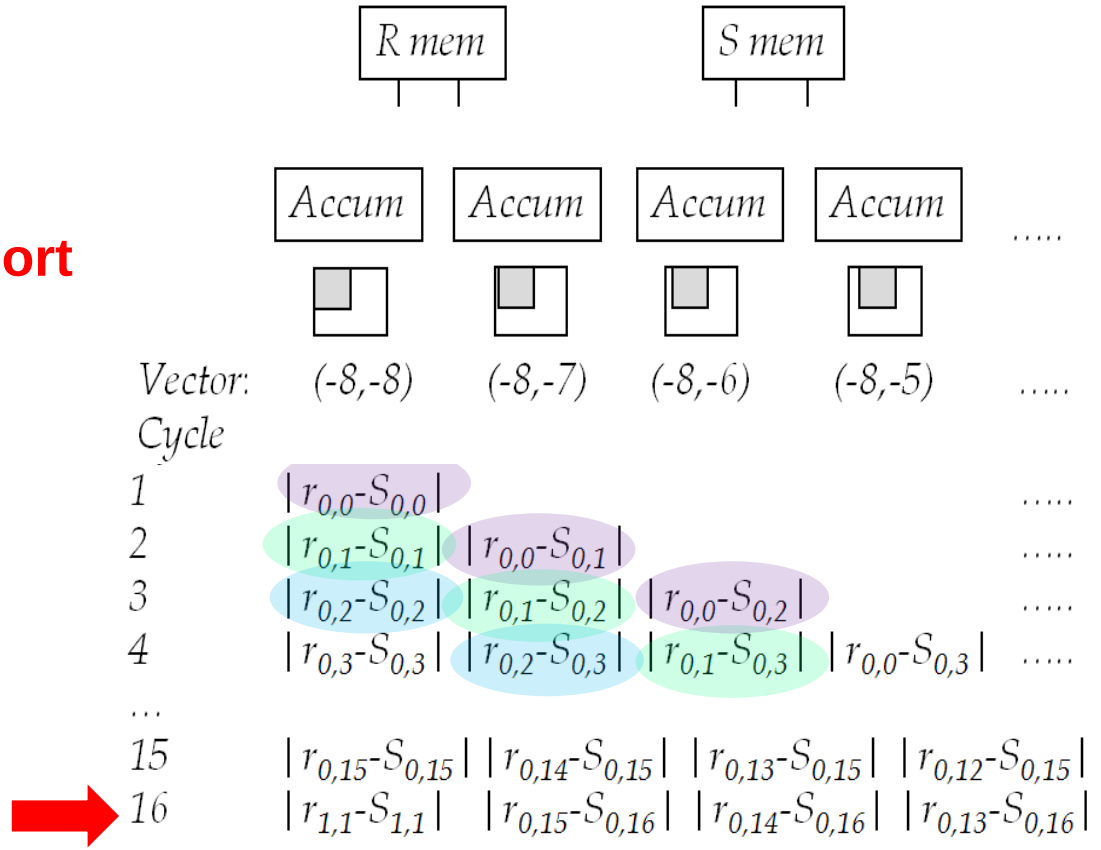
Problem: Requires 16 port R memory



… System Design

- Second Attempt:
- Stagger Startup of Accumulators

Problem: Requires 16 port R memory

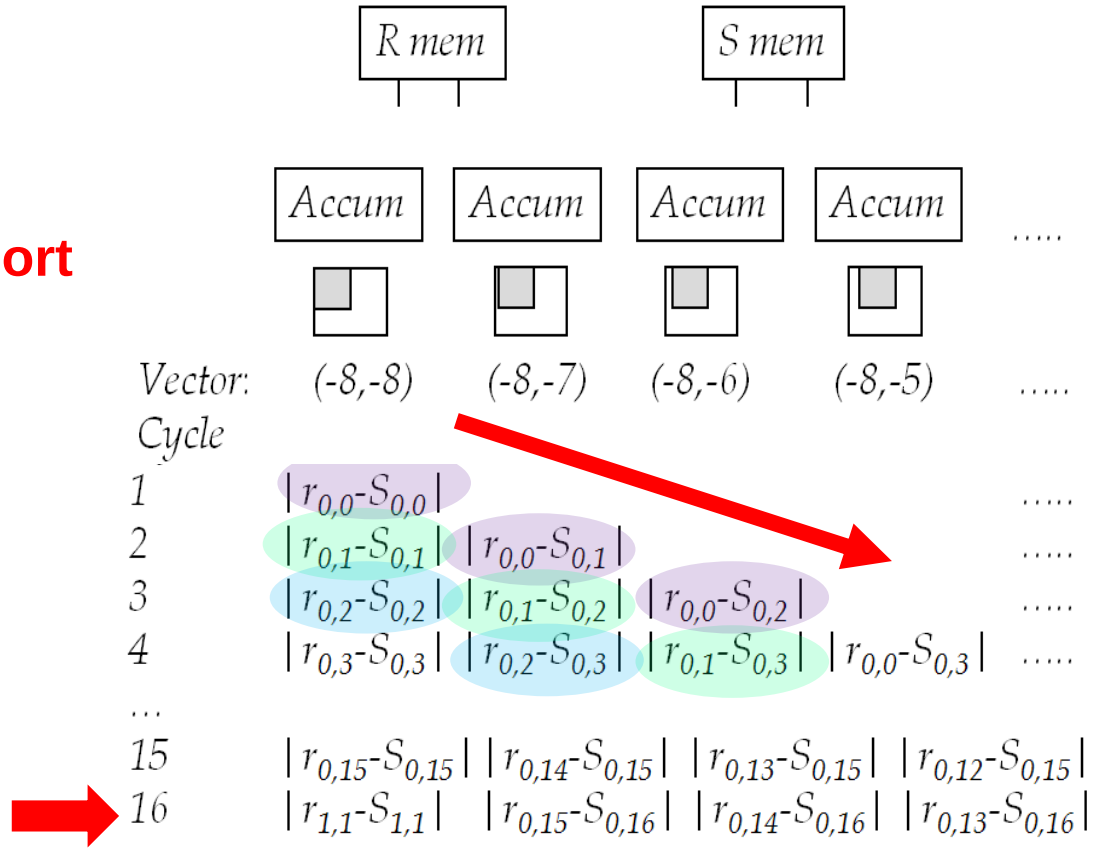


… System Design

- Second Attempt:
- Stagger Startup of Accumulators

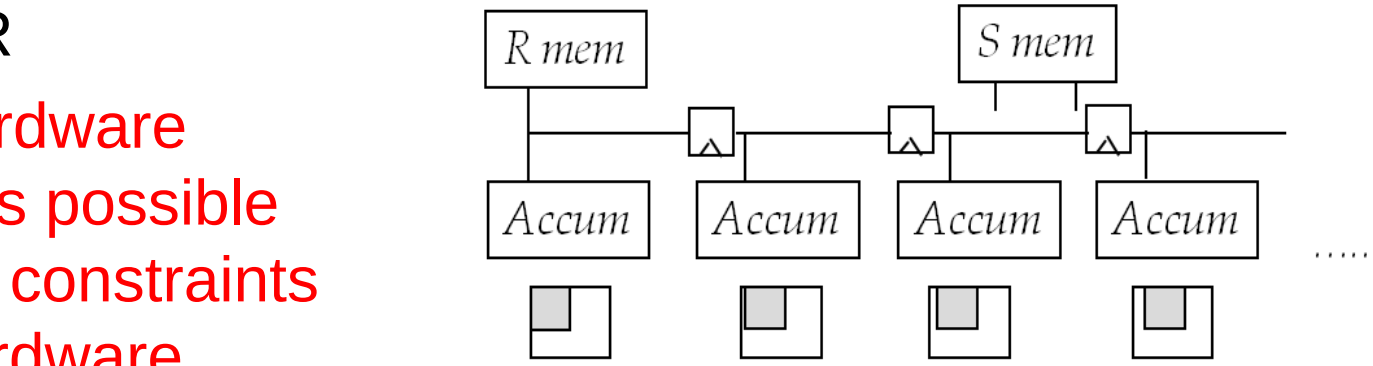
Problem: Requires 16 port R memory

Problem: R Pattern



… System Design

- Final Solution:
- Pipeline R
- Keeps hardware as busy as possible within the constraints of that hardware



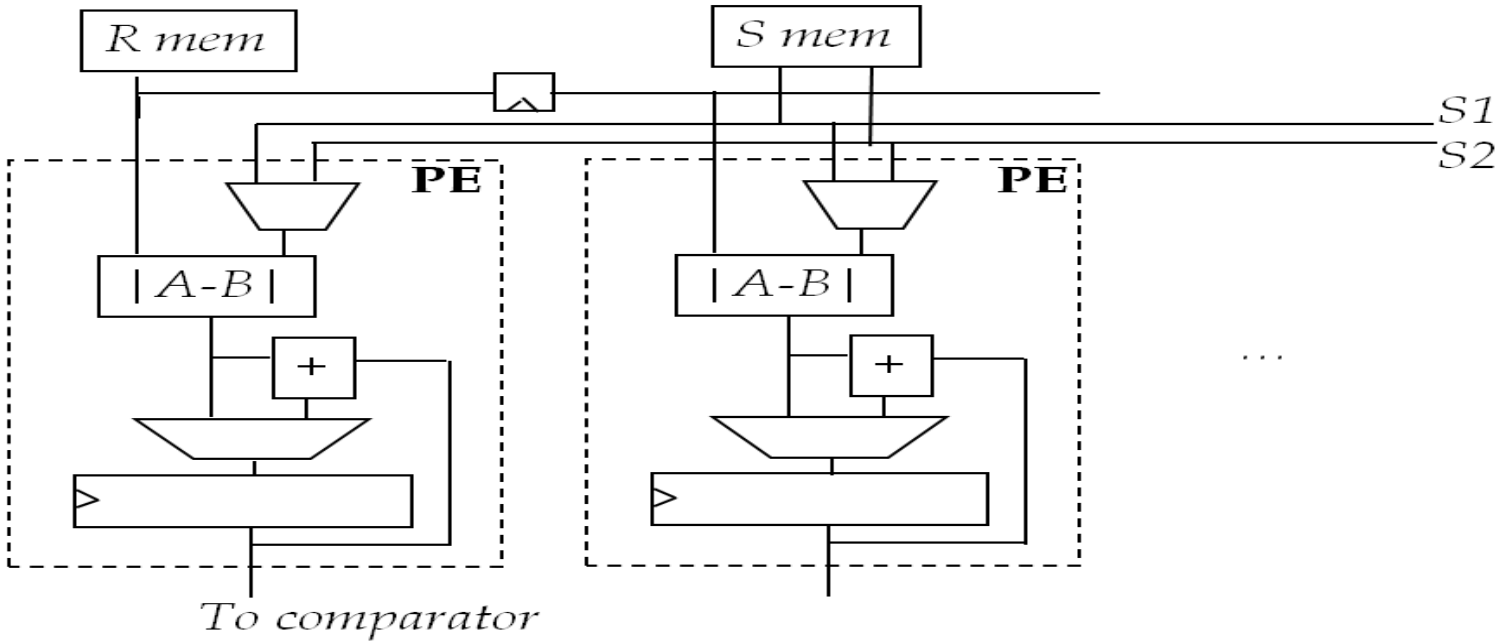
Vector:	(-8,-8)	(-8,-7)	(-8,-6)	(-8,-5)	.....
Cycle					
1	$\lvert r_{0,0}-S_{0,0}\rvert$				.....
2	$\lvert r_{0,1}-S_{0,1}\rvert$	$\lvert r_{0,0}-S_{0,1}\rvert$			.....
3	$\lvert r_{0,2}-S_{0,2}\rvert$	$\lvert r_{0,1}-S_{0,2}\rvert$	$\lvert r_{0,0}-S_{0,2}\rvert$		.....
4	$\lvert r_{0,3}-S_{0,3}\rvert$	$\lvert r_{0,2}-S_{0,3}\rvert$	$\lvert r_{0,1}-S_{0,3}\rvert$	$\lvert r_{0,0}-S_{0,3}\rvert$	.....
…					
15	$\lvert r_{0,15}-S_{0,15}\rvert$	$\lvert r_{0,14}-S_{0,15}\rvert$	$\lvert r_{0,13}-S_{0,15}\rvert$	$\lvert r_{0,12}-S_{0,15}\rvert$	
16	$\lvert r_{1,1}-S_{1,1}\rvert$	$\lvert r_{0,15}-S_{0,16}\rvert$	$\lvert r_{0,14}-S_{0,16}\rvert$	$\lvert r_{0,13}-S_{0,16}\rvert$	

2 S mem ports required →

Step 2 : Design Datapath

- Datapath Details:
- Detailed hardware required to implement above

PE = Processing Element



SYNOPSYS®

Coding Datapath

- PE: Note, accumulator cant overflow – saturate at FF

```
module PE (clock, R, S1, S2, S1S2mux, newDist, Accumulate, Rpipe);
input clock;
input [7:0] R, S1, S2;// memory inputs
input S1S2mux, newDist;// control inputs
output [7:0] Accumulate, Rpipe;
reg [7:0] Accumulate, AccumulateIn, Difference, Rpipe;
reg       Carry;

always @(posedge clock) Rpipe <= R;
always @(posedge clock) Accumulate <= AccumulateIn;

always @(R or S1 or S2 or S1S2mux or newDist or Accumulate)
  begin  // capture behavior of logic
   difference = R - S1S2mux ? S1 : S2;
   if (difference < 0) difference = 0 - difference;
// absolute subtraction
    {Carry,AccumulateIn} = Accumulate + difference;
    if (Carry == 1) AccumulateIn = 8'hFF;// saturated
    if (newDist == 1) AccumulateIn = difference;
// starting new Distortion calculation
  end
endmodule
```

Motion Estimator Processing Element (PE).

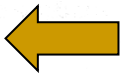
Coding Datapath

- PE: Note, accumulator cant overflow – saturate at FF

```
module PE (clock, R, S1, S2, S1S2mux, newDist, Accumulate, Rpipe);
input clock;
input [7:0] R, S1, S2;// memory inputs
input S1S2mux, newDist;// control inputs
output [7:0] Accumulate, Rpipe;
reg [7:0] Accumulate, AccumulateIn, Difference, Rpipe;
reg       Carry;

always @(posedge clock) Rpipe <= R;
always @(posedge clock) Accumulate <= AccumulateIn;

always @(R or S1 or S2 or S1S2mux or newDist or Accumulate)
  begin  // capture behavior of logic
   difference = R - S1S2mux ? S1 : S2;
   if (difference < 0) difference = 0 - difference;
// absolute subtraction
    {Carry,AccumulateIn} = Accumulate + difference;
    if (Carry == 1) AccumulateIn = 8'hFF;// saturated
    if (newDist == 1) AccumulateIn = difference;
// starting new Distortion calculation
  end
endmodule
```



Motion Estimator Processing Element (PE).

Coding Datapath

- PE: Note, accumulator cant overflow – saturate at FF

```
module PE (clock, R, S1, S2, S1S2mux, newDist, Accumulate, Rpipe);
input clock;
input [7:0] R, S1, S2;// memory inputs
input S1S2mux, newDist;// control inputs
output [7:0] Accumulate, Rpipe;
reg [7:0] Accumulate, AccumulateIn, Difference, Rpipe;
reg       Carry;

always @(posedge clock) Rpipe <= R;
always @(posedge clock) Accumulate <= AccumulateIn;

always @(R or S1 or S2 or S1S2mux or newDist or Accumulate)
  begin  // capture behavior of logic
   difference = R - S1S2mux ? S1 : S2;
   if (difference < 0) difference = 0 - difference;
// absolute subtraction
    {Carry,AccumulateIn} = Accumulate + difference;
    if (Carry == 1) AccumulateIn = 8'hFF;// saturated
    if (newDist == 1) AccumulateIn = difference;
// starting new Distortion calculation
  end
endmodule
```

I don't need large distortions

Motion Estimator Processing Element (PE).

Coding Datapath

- PE: Note, accumulator cant overflow – saturate at FF

```
module PE (clock, R, S1, S2, S1S2mux, newDist, Accumulate, Rpipe);
input clock;
input [7:0] R, S1, S2;// memory inputs
input S1S2mux, newDist;// control inputs
output [7:0] Accumulate, Rpipe;
reg [7:0] Accumulate, AccumulateIn, Difference, Rpipe;
reg       Carry;

always @(posedge clock) Rpipe <= R;
always @(posedge clock) Accumulate <= AccumulateIn;

always @(R or S1 or S2 or S1S2mux or newDist or Accumulate)
  begin  // capture behavior of logic
   difference = R - S1S2mux ? S1 : S2;
   if (difference < 0) difference = 0 - difference;
// absolute subtraction
    {Carry,AccumulateIn} = Accumulate + difference;
    if (Carry == 1) AccumulateIn = 8'hFF;// saturated
    if (newDist == 1) AccumulateIn = difference;
// starting new Distortion calculation
  end
endmodule
```

I don't need large distortions

NO else? Latch?

Motion Estimator Processing Element (PE).

Coding Datapath

- PE: Note, accumulator cant overflow – saturate at FF

```
module PE (clock, R, S1, S2, S1S2mux, newDist, Accumulate, Rpipe);
input clock;
input [7:0] R, S1, S2;// memory inputs
input S1S2mux, newDist;// control inputs
output [7:0] Accumulate, Rpipe;
reg [7:0] Accumulate, AccumulateIn, Difference, Rpipe;
reg       Carry;

always @(posedge clock) Rpipe <= R;
always @(posedge clock) Accumulate <= AccumulateIn;

always @(R or S1 or S2 or S1S2mux or newDist or Accumulate)
  begin  // capture behavior of logic
   difference = R - S1S2mux ? S1 : S2;
   if (difference < 0) difference = 0 - difference;
// absolute subtraction
    {Carry,AccumulateIn} = Accumulate + difference;
    if (Carry == 1) AccumulateIn = 8'hFF;// saturated
    if (newDist == 1) AccumulateIn = difference;
// starting new Distortion calculation
  end
endmodule
```

I don't need large distortions

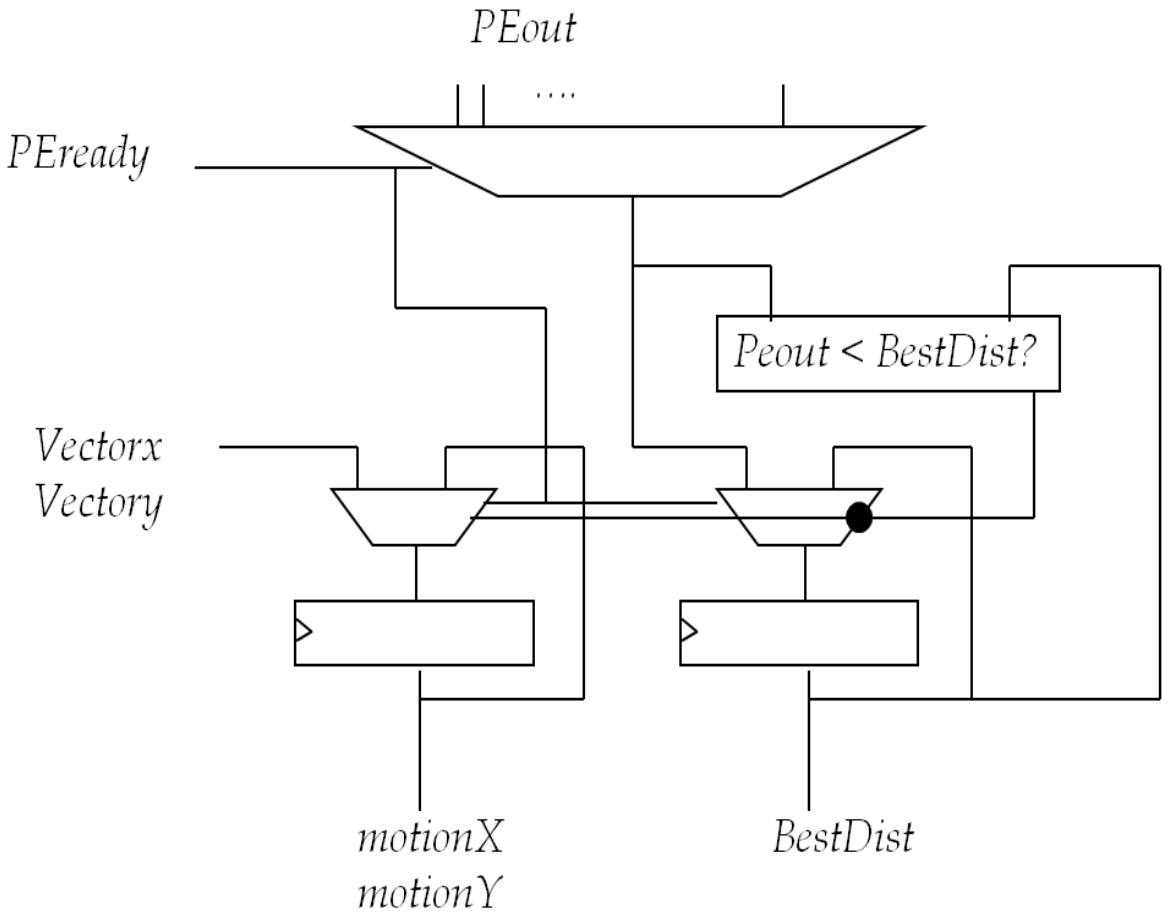
NO else? Latch?

No! It always takes a value!!

Motion Estimator Processing Element (PE).

… Datapath

- Comparator:



Datapath

```
module Comparator (clock, CompStart, PEout, PEready, vectorX,
                        vectorY, BestDist, motionX, motionY);
input clock;
input CompStart; // goes high when distortion calculations start
input [8*16:0] PEout; // Outputs of PEs as one long vector
input [15:0] PEready; // Goes high when that PE has a new distortion
input [3:0] vectorX, vectorY; // Motion vector being evaluated
output [7:0] BestDist; // Best Distortion vector so far
output [3:0] motionX, motionY; // Best motion vector so far
reg [7:0] BestDist, newDist;
reg [3:0] motionX, motionY;
reg       newBest;

always @(posedge clock)
  if (CompStart == 0) BestDist <= 8'hFF;  //initialize to highest value
  else if (newBest == 1)
    begin
      BestDist <= newDist;
      motionX <= vectorX;
      motionY <= vectorY;
    end

always @(BestDist or PEout or PEready)
  begin
   newDist = PEout [PEready*8+7 : PEready*8];
   if ((|PEready == 0) || (start == 0)) newBest = 0; // no PE is ready
   else if (newDist < BestDist) newBest = 1;
   else newBest = 0;
  end

endmodule
```

Comparator Module.

Step 3. Identify Control Points

PE control lines:

- S1S2mux [15:0]; // S1-S2 mux control
- NewDist [15:0] ; // =1 when PE is starting a new distortion calculation

Comparator control lines:

- CompStart;11 = // when PEs running
- PEready [15:0]; // PEready[I]=1 when PEi has a new distortion vector
- VectorX [3:0] ;
- VectorY [3:0];// Motion vector being evaluated

Memory control lines:

- Memories organization
- AddressR [7:0]; // address for Reference memory (0,0). ..(15,15)
- AddressS1 [9:0] ; // address for first read port of Search mem
- AddressS2 [9:0] ; // second read port of Search mem (0,0)-(30,30)

Step. 4 Design Controller

- Best Strategy : Counter
- Reset Strategy
- Reset needed to initialize entire chip in known state
  - Does not apply here, as long as “start” comes from a unit that does use a reset

Step. 4 Design Controller

- Best Strategy : Counter

```
always @(posedge clock)
  if (start == 0) count <= 12'b0;
  else if (completed == 0) count <= count + 1'b1;

always @(count)
  begin
    for (i=0; i<15; i = i+1)
      begin
       NewDist[i] = (count[7:0] == i);
       PEready[i] = (NewDist[i] && !(count < 9'd256));
       S1S2mux[i] = (count[3:0] > i);
      end
    AddressR = count[7:0];
    AddressS1 = (count[11:8] + count[7:4]>>4)*5'd32 + count[3:0];
    AddressS2 = (count[11:8] + count[7:4]>>4)*4'd16 + count[3:0];
    VectorX = count[3:0] - 4'd7;
    VectorY = count[11:8]>>4 - 4'd7;
    complete = (count = 4'd16 * (9'd256 + 1));
  end

endmodule
```

Step. 4 Design Controller

- Best Strategy : Counter

```
module control (clock, start, S1S2mux, NewDist, CompStart, PEready,
                VectorX, VectorY, AddressR, AddressS1, AddressS2);
input clock;
input start; // = 1 when`going´
output [15:0] S1S2mux;
output [15:0] NewDist;
output  CompStart;
output [15:0] PEready;
output [3:0]  VectorX, VectorY;
output [7:0]  AddressR;
output [9:0]  AddressS1, AddressS2;
reg [15:0] S1S2mux;
reg [15:0] NewDist;
reg        CompStart;
reg [15:0] PEready;
reg [3:0]  VectorX, VectorY;
reg [7:0]  AddressR;
reg [9:0]  AddressS1, AddressS2;
reg [12:0] count;
reg     completed;
integer    i;
```