

Part 1: Datapath & Control Unit

Part 2: Pipeline

Computer Architecture
Dr. Guy Tel-Zur

Part 1: Datapath & Control Unit

Computer Architecture
Dr. Guy Tel-Zur

מטרת ההרצאה

במצגת קצרה זו נעמוד על עקרון תכן מרכזי והוא הפרדה בין מסלול הנתונים לבקר.

נושאי המצגת:

- א. החלוקה לבקר ולמסלול נתונים
- ב. בעיית דוגמה ראשונה. תרשים לוגי של מסלול הנתונים ותרשים FSM לבקר וכן טבלת מצבים מסכמת
- ג. בעיית דוגמה שנייה. כנ"ל עבור חישוב $N!$

This presentation is based on:

- MIT Open course ware "**Computation Structures**" by Dr. Christopher J. Terman.
- References:
 - [MIT Open course ware](https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-004-computation-structures-spring-2017/c9/c9s1/#1)
 - [MIT6.004 2020](https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-004-computation-structures-spring-2017/c9/c9s1/#1)
 - Slides by Prof. Shmuel Wimer Technion/BIU

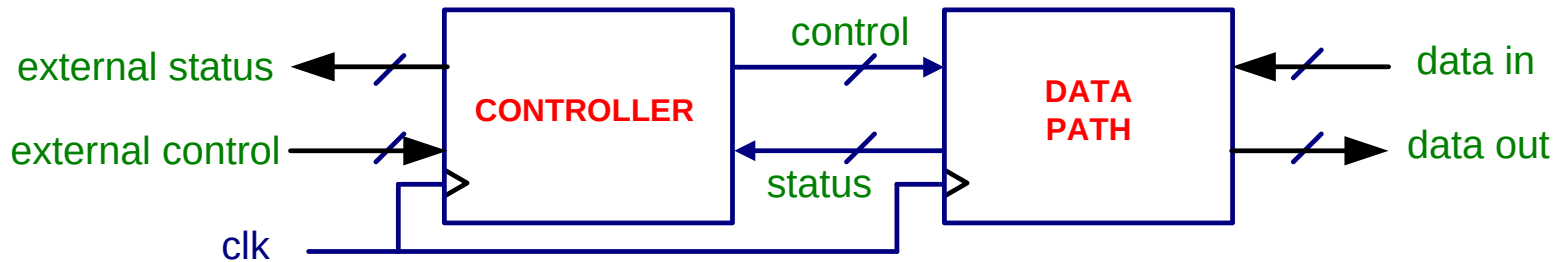
Chris Terman. 6.004 Computation Structures. Spring 2017. Massachusetts Institute of Technology: MIT OpenCourseWare, <https://ocw.mit.edu>. License: Creative Commons BY-NC-SA.

In particular we will discuss material from:

<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-004-computation-structures-spring-2017/c9/c9s1/#1>

See 9.2.1, 9.2.2 @ MIT open course ware site

חלוקה לבקר ומסלול נתונים



- מסלול נתונים:

- תפקיד – מאחסן נתונים, מעבד

- רכיבים אופייניים – רגיסטרים, מסכם, ALU, בוררים

- בקר:

- תפקיד – קובע מה יבצע מסלול הנתונים

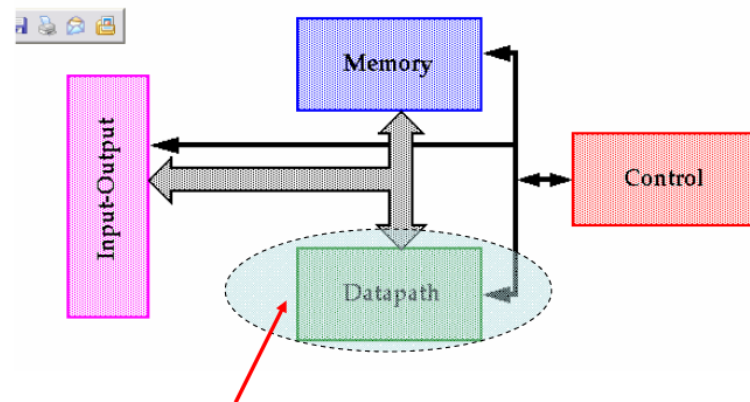
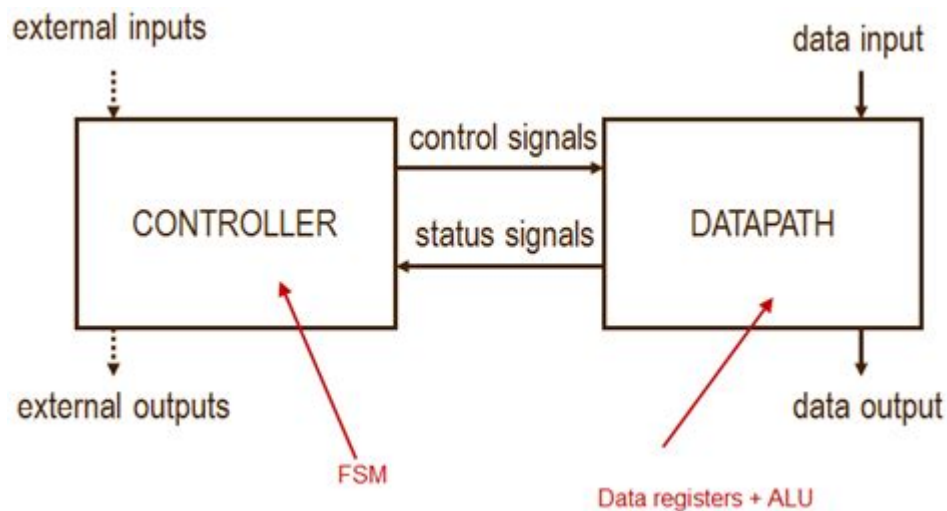
- מימוש - מערכת סדרתית כללית, למשל מימוש ע"י ROM ורגיסטר.

- קשר בין החלקים: אותות בקרה וסטטוס

- שני החלקים הם בדרך כלל מערכות סדרתיות

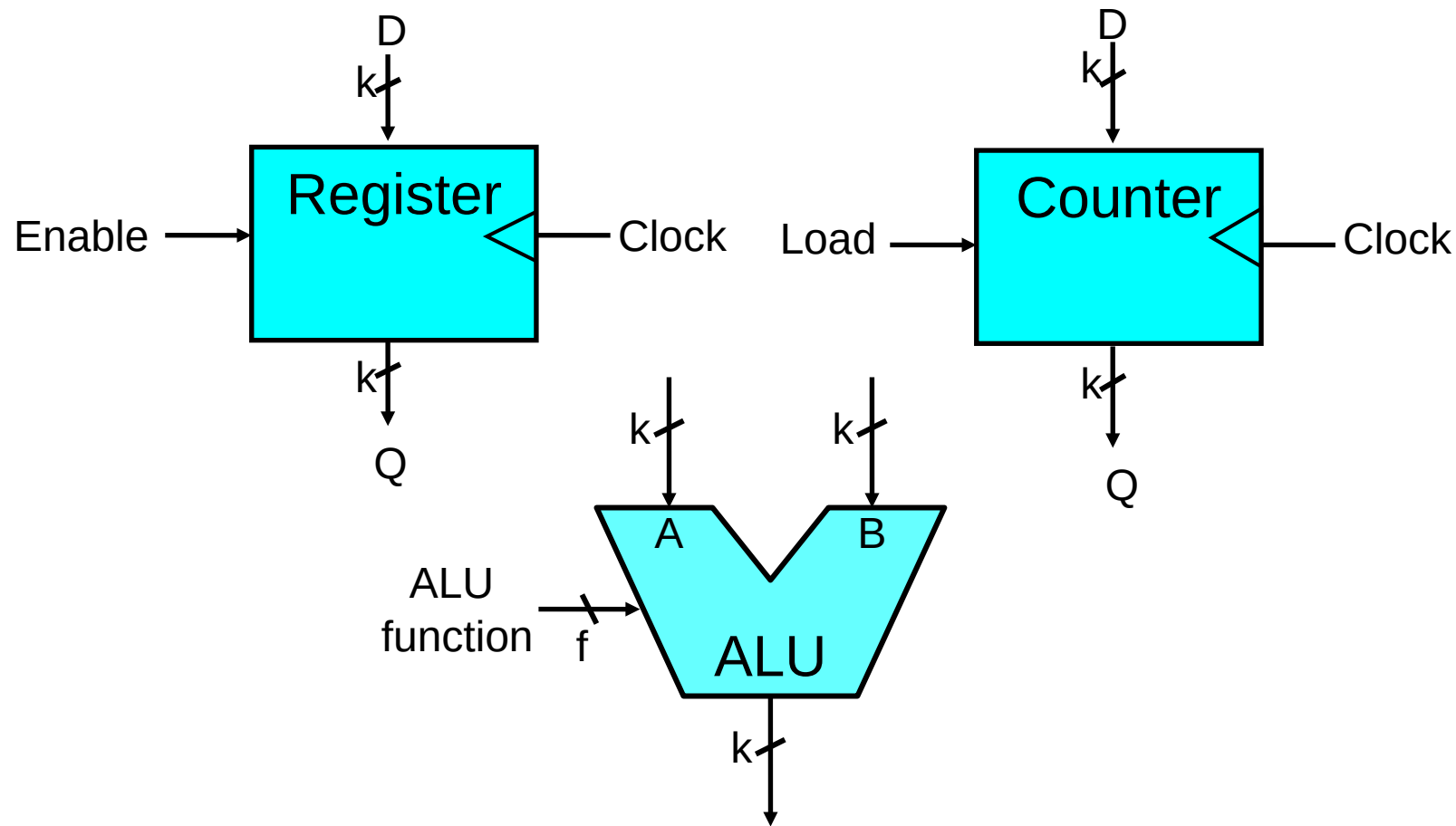
- החלוקה לא יחידה – יש מספר אפשרויות

בהתייחס לסעיף האחרון בשקף הקודם, להלן תרשימים אלטרנטיביים



- We shall concentrate first on this.

רכיבים סטנדרטיים למסלול נתונים פשוט



דוגמת תכנון: מסכם סדרות

- נדרש: מעגל לוגי המקבל סדרה טורית
 $\dots, 0, 0, n, x_1, x_2, \dots, x_n, m, y_1, y_2, \dots, y_m, 0, 0, \dots$
- ומחשב את הסכומים
 $\sum_n x_i, \sum_m y_i, \dots$
- שלבי התכנון:

1. תכנון מסלול נתונים בעזרת רכיבים סטנדרטיים
(מסכם, רגיסטר, מונה)
2. הגדרת קווי בקרה וסטטוס – דרישות של מסלול
הנתונים מהבקר
3. תכנון בקר בעזרת רכיבים סטנדרטיים (רגיסטר,
מסכם, מונה)

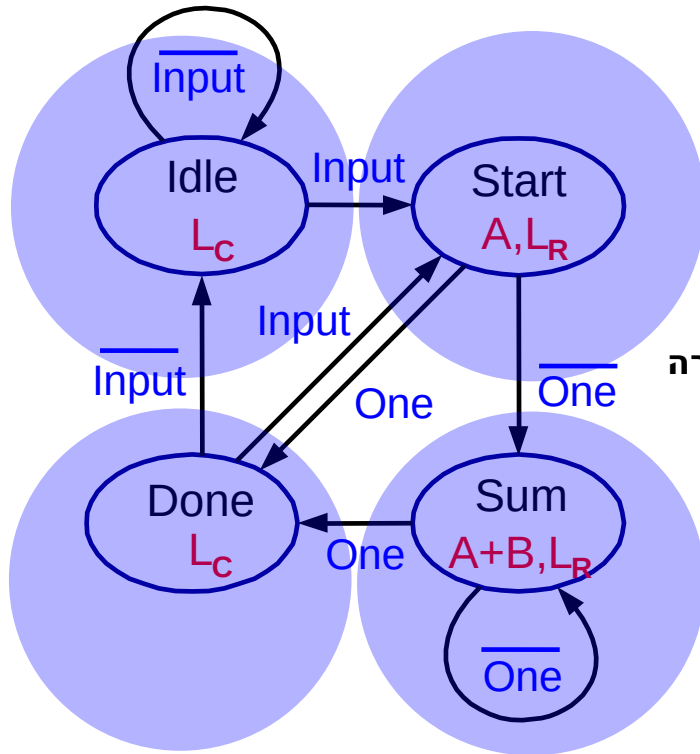
מסכם סדרות

מקרא:

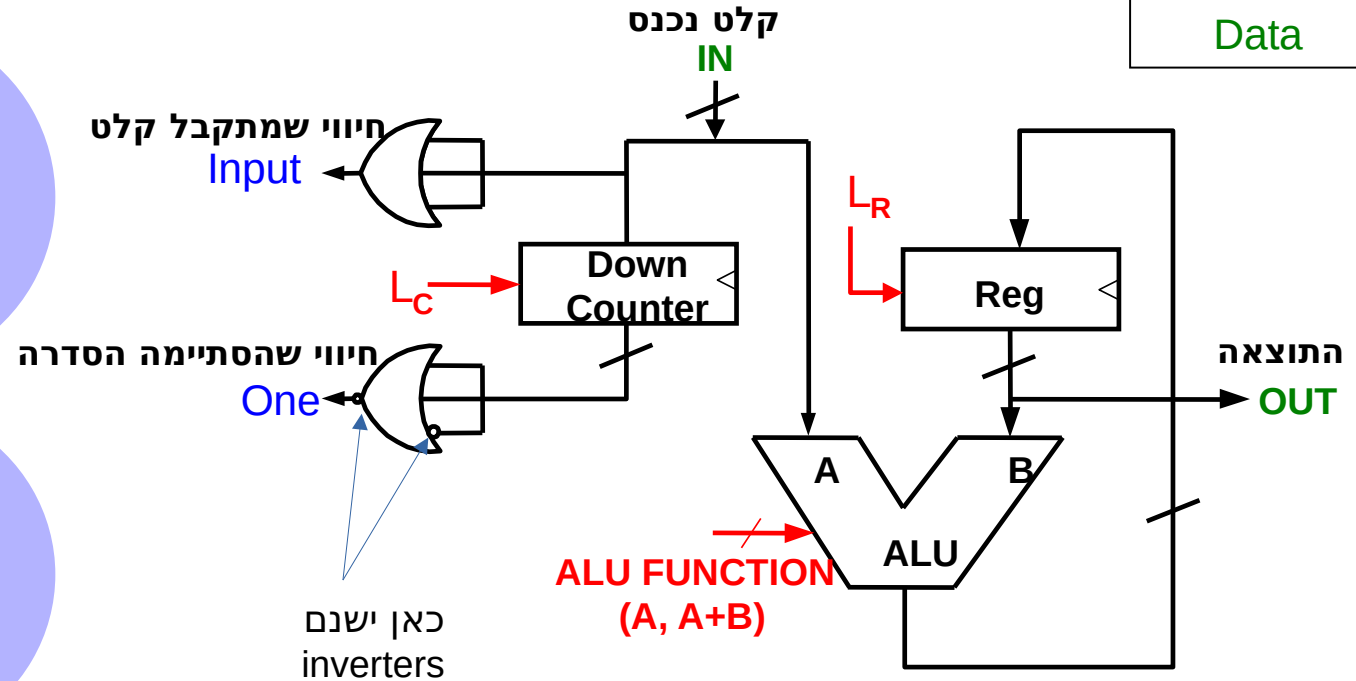
Control

Status

Data



בקר



מסלול נתונים

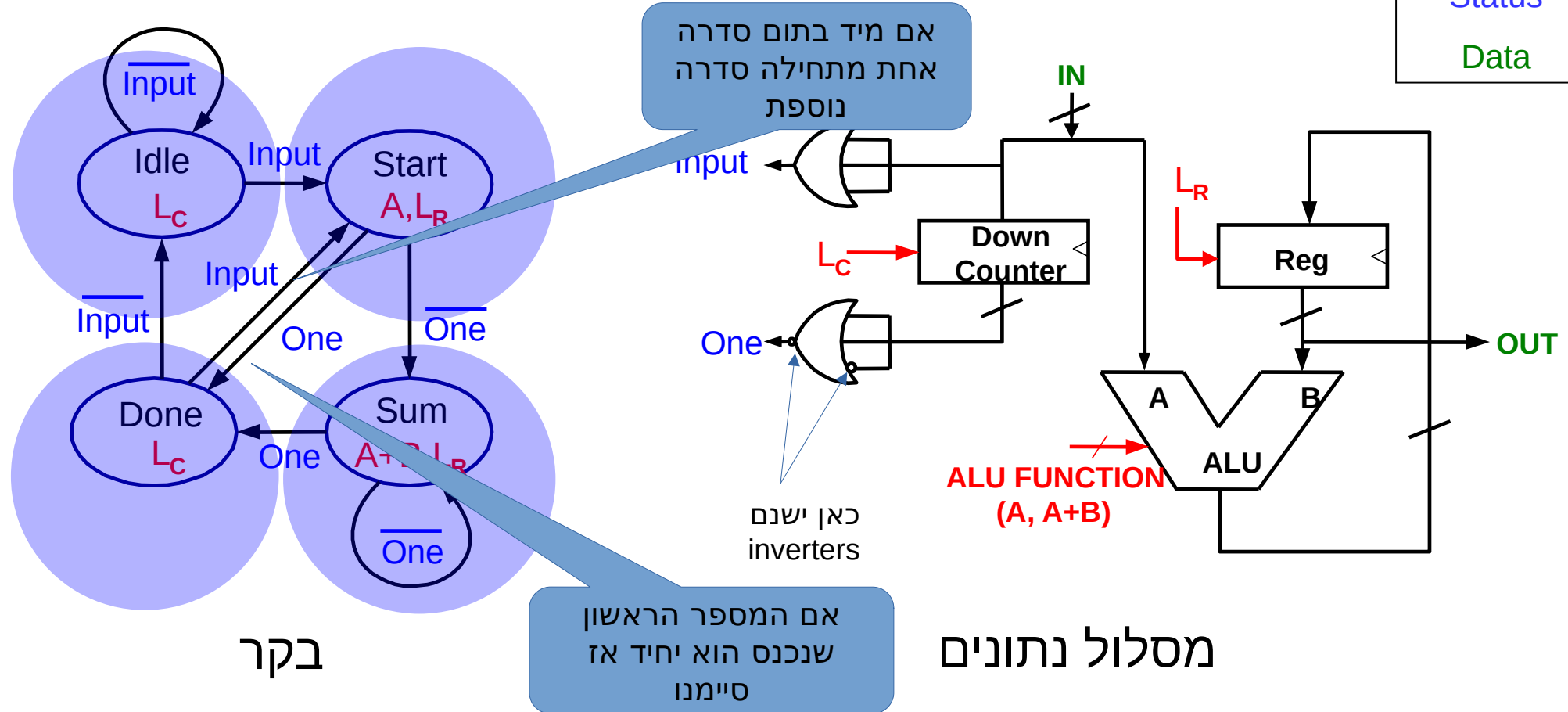
מסכם סדרות

מקרא:

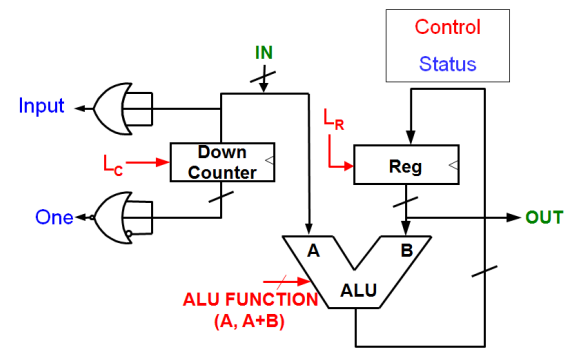
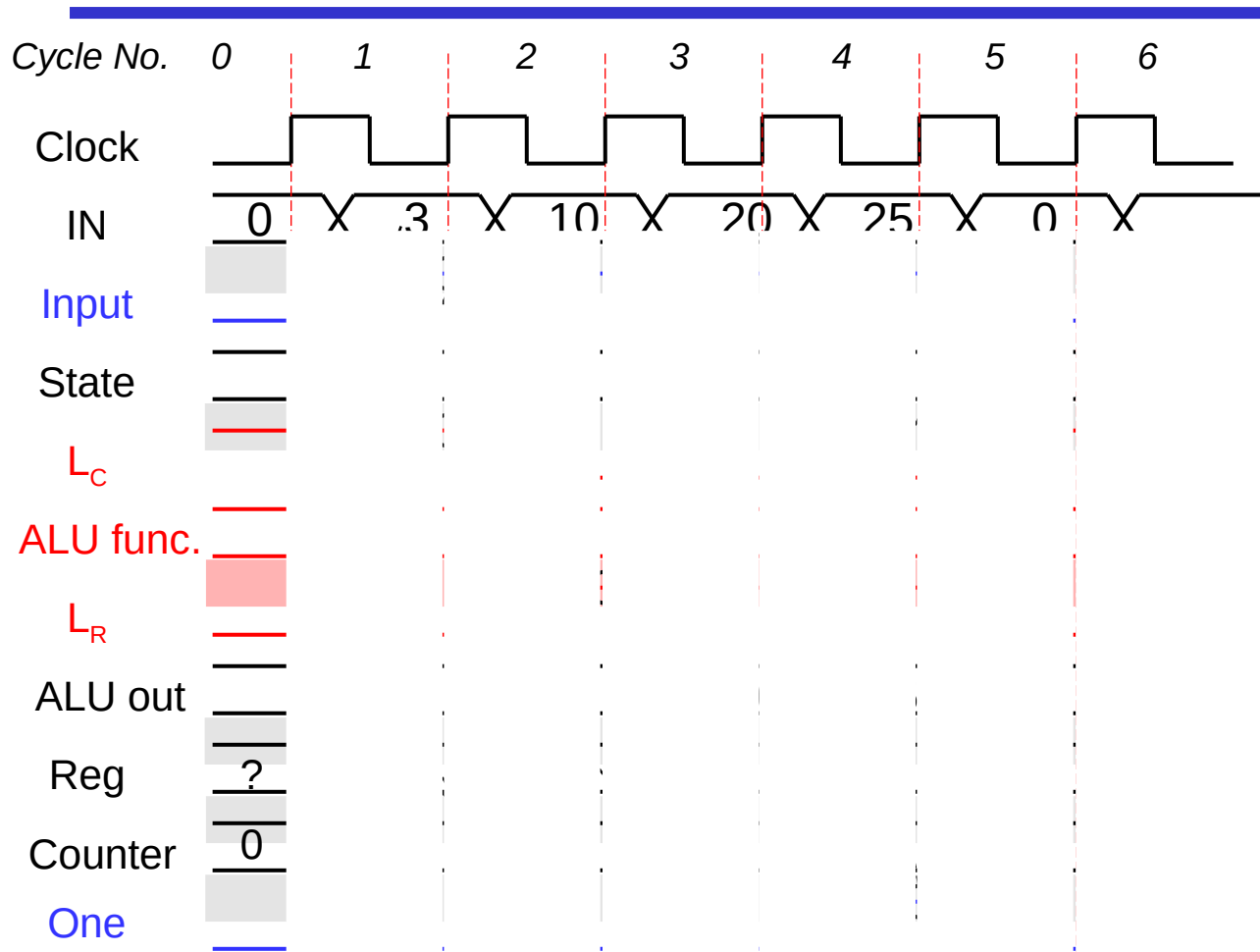
Control

Status

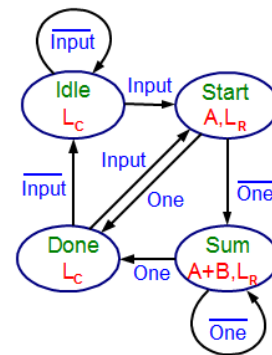
Data



דוגמא לסיכום סדרת 3 מספרים



מסלול נתונים



בקר

טבלת המצבים

S	Status		Control			S'	Out
Current state	input	one	L_c	L_R	ALU Func	Next state	Register output
IDLE	0	x	1	0	-	IDLE	x
IDLE	1	x	1	0	-	START	x
START	x	0	0	1	A	SUM	A+B
START	x	1	0	1	A	DONE	A
SUM	x	0	0	1	A+B	SUM	A+B
SUM	x	1	0	1	A+B	DONE	A+B
DONE	0	x	1	0	-	IDLE	A+B
DONE	1	x	1	0	-	START	A+B

מדוע החלוקה לבקר ומסלול נתונים יעילה?

העיקרון - הפרדת מערכת סדרתית עם מספר מצבים גדול לשתי מערכות סידרתיות:

– באחת (DP=DataPath) מספר מצבים גדול (רוב המצבים)

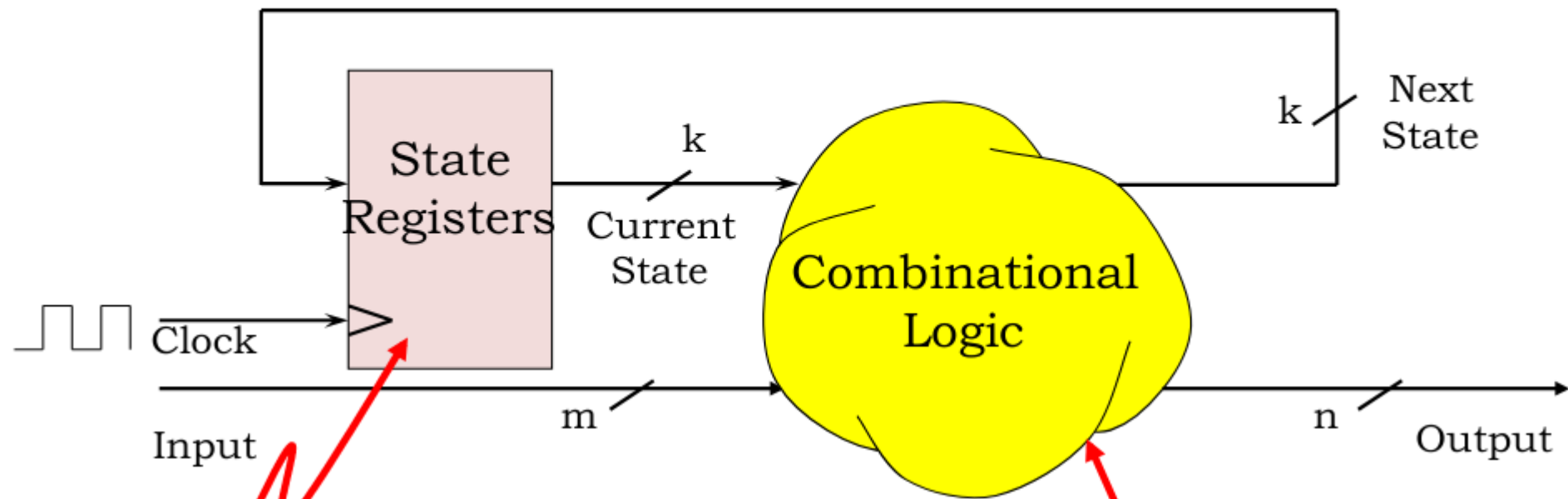
התכנון אינו בכלים מסורתיים ל-FSM

– באחרת (בקר) מספר המצבים קטן

ומתבצע תכנון FSM רגיל

- אם במסלול הנתונים n מצבים ובבקר k מצבים, כמה מצבים יש במכונה ללא פירוק?

- אחת ר-FSM יש K פליט-פלופים. מה סירוקיות התכנון (כמה מצבים יש)?

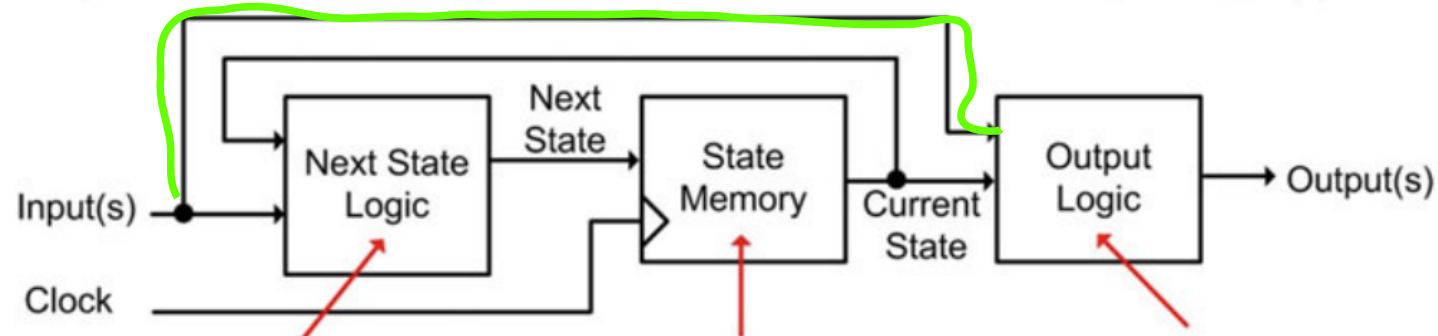


- Engineered cycles
- Works only if dynamic discipline obeyed
- Remembers k bits for a total of 2^k unique combinations

- Acyclic graph
- Obeys static discipline
- Can be exhaustively enumerated by a truth table of 2^{k+m} rows and $k+n$ output columns

Main Components of a Finite State Machine

Mealy Machine – The output(s) depend on both the current state and system input(s).

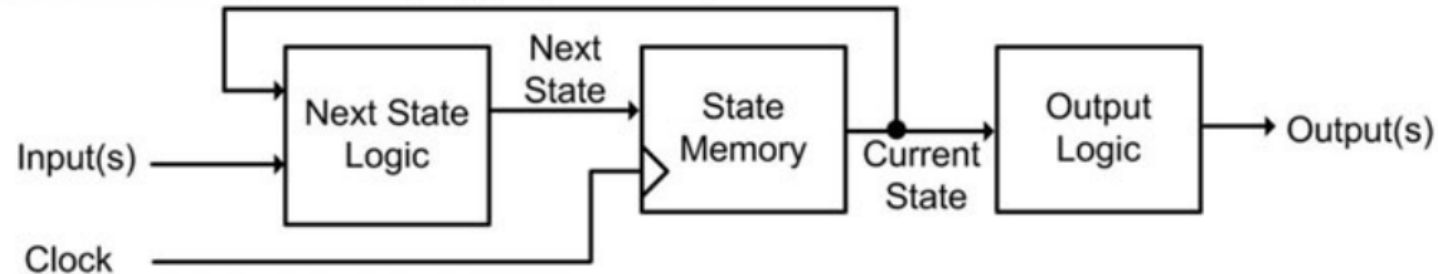


The next state logic creates the signal "next state" based on the current state and any system inputs. This block is implemented with combinational logic.

The state memory holds the current state. The current state is updated with "next state" on the rising edge of the clock. This block is implemented with D-Flip-Flops.

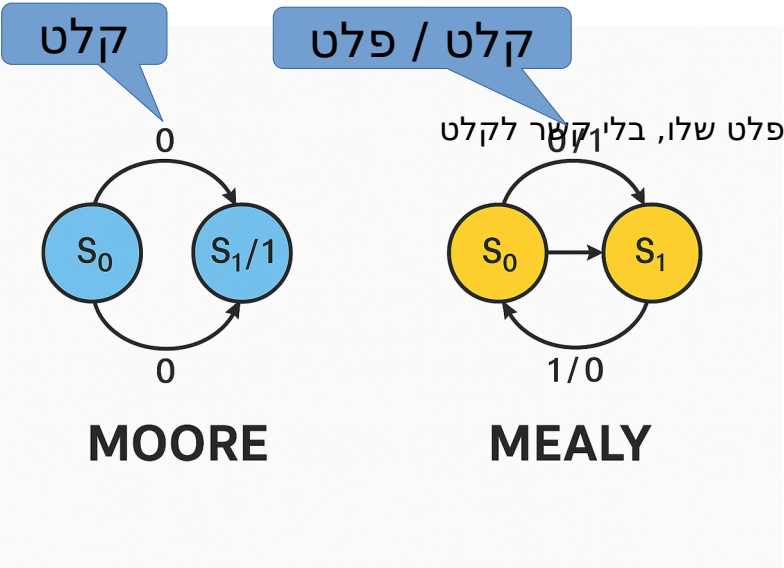
The output logic creates the system outputs. The output logic always depends on the current state of the machine and optionally (Mealy vs. Moore) the inputs of the system.

Moore Machine – The output(s) depends only on the current state.



הסבר פשוט אודות ההבדלים בין 2 סוגי מכונות המצבים הסופיים

מכונת Moore:



הפלט (output) תלוי רק במצב הנוכחי שבו נמצאת המכונה. כלומר, כל מצב "יודע" מראש מה הפלט שלו, בלי קשר לקלט שקיבל הרגע.

המעבר בין מצבים כן תלוי בקלט.

דוגמה פשוטה: נניח שיש מצב שנקרא $S1$ וכל עוד המכונה נמצאת בו, הפלט תמיד יהיה 1.

לא משנה איזה קלט היא תקבל, כל עוד היא לא עוזבת את $S1$, הפלט נשאר 1.

מכונת Mealy:

הפלט (output) תלוי גם במצב הנוכחי וגם בקלט הנוכחי. כלומר, אפילו אם המכונה באותו מצב, היא יכולה להוציא פלט שונה לפי הקלט שהיא מקבלת.

מאפיין	Moore	Mealy
הפלט תלוי ב...	רק במצב הנוכחי	מצב נוכחי וקלט נוכחי
תגובה לקלט	פחות מהירה (בשל שינוי המצב)	מהירה יותר (פלט מידי לקלט)
תכן	פשוט יותר	גמיש יותר. לפעמים חסכוני

השוואה בין שני המודלים.לא להתעכב על זה

BASIS OF COMPARISON	MEALY MACHINE	MOORE MACHINE
Description	Mealy machine changes its output based on its current input and present state.	Output of Moore machine only depends on its current state and not on the current input.
States	Mealy machine will have same or fewer states than Moore machine.	It has more states than Mealy machine.
Output	Output is placed on transition.	Output is placed on transition.
Value Of Output Function	The value of the output function is a function of the transitions and the changes, when the input logic on the present state is done.	The value of the output function is a function of the current state and the changes at the clock edges, whenever state changes occur.
Reaction To Inputs	Mealy machines react faster to inputs. They generally react in the same clock cycle.	More logic is required to decode the output resulting in more circuit delays. They generally react one clock cycle later.
Output And State	Asynchronous output generation through the state changes synchronous to the clock.	Both output and state change synchronous to the clock edge.
States Requirement	Generally requires fewer states for synthesis.	Generally requires more states for synthesis.
Hardware Requirement	Requires less hardware to design.	More hardware is required to design.
Counter	A counter is not a Mealy machine.	A counter is a Moore machine.
Design	Not necessarily easy to design.	Easy to design.

מספר המצבים

כמות החומרה הנדרשת

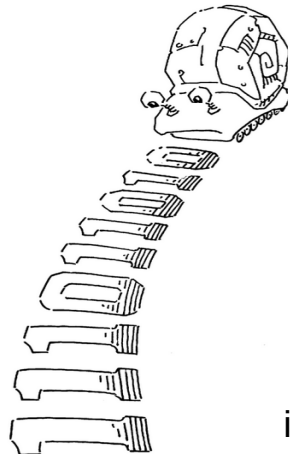
מונה הוא מכונת מור

קלות התכנון

FSM Example: Smiling Snail (H&H Ch. 3 and Mutlu L07, 2022)

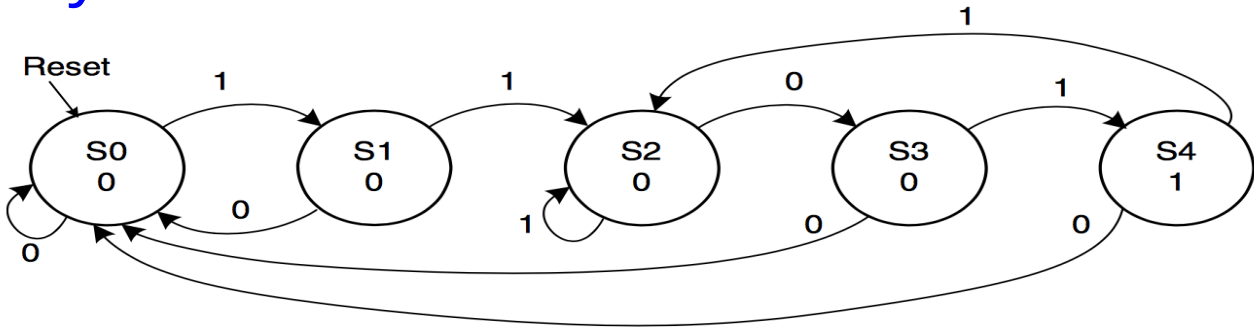
<https://safari.ethz.ch/digitaltechnik/spring2022/lib/exe/fetch.php?media=digitaldesign-comparch-2022-lecture7-hdl-verilog-afterlecture.pdf>

- A snail crawls down a paper tape with 1's and 0's on it
- The snail smiles whenever the last four digits it has crawled over are **1101**
- Design Moore and Mealy FSMs of the snail's brain



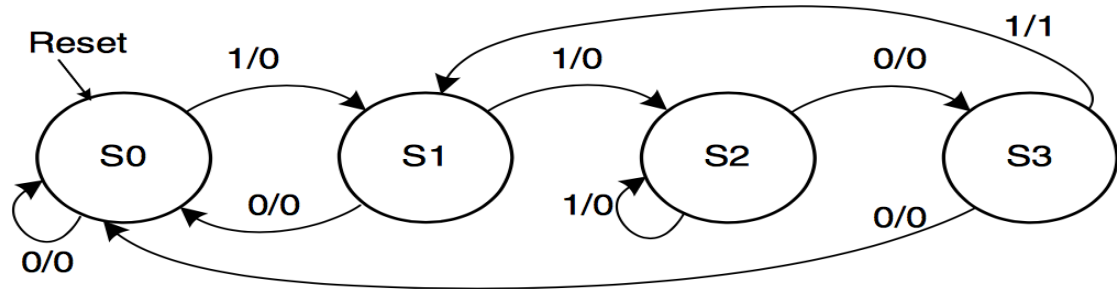
Moore

הקלט הוא על
החיצים.
המוצא הוא המספר
בתוך העיגולים

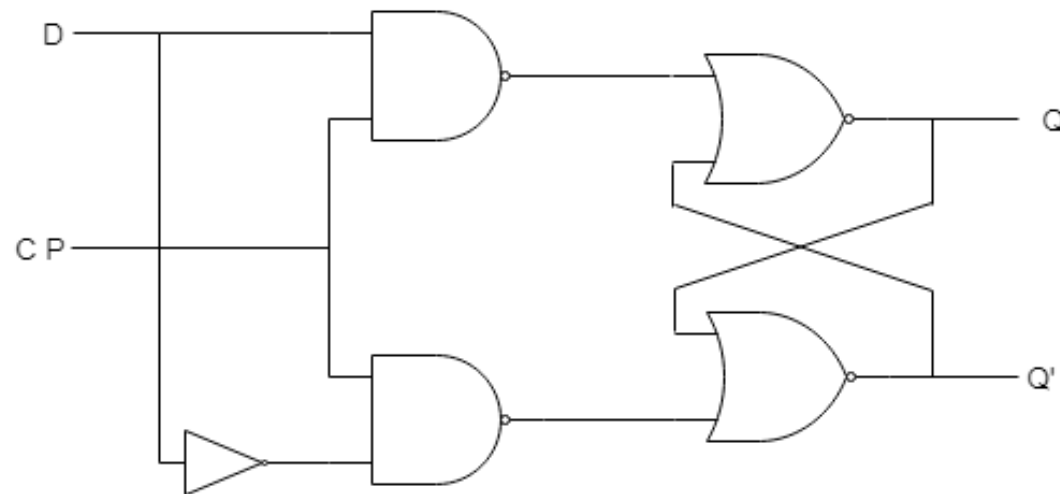


Mealy

על החיצים:
input/output
לכן המוצא תלוי בקלט
ובמצב



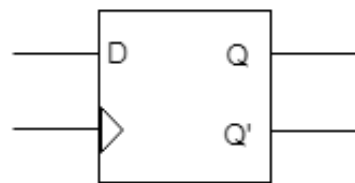
תזכורת D flip flop



(a) Logic diagram with Nand gates

Q	D	Q (t+1)
0	0	0
0	1	1
1	0	0
1	1	1

e



(b) Graphic Symbol

fig. Clocked D flip flop

Example: Factorial

$$\text{factorial}(N) = N! = N * (N-1) * \dots * 1$$

C:

```
int a = 1;
int b = N;
do {
    a = a * b;
    b = b - 1;
} while (b != 0)
```

a מכיל את התוצאה
b מונה כלפי מטה

```
initially:    a = 1, b = 5
after iter 1: a = 5, b = 4
after iter 2: a = 20, b = 3
after iter 3: a = 60, b = 2
after iter 4: a = 120, b = 1
after iter 5: a = 120, b = 0
Done!
```

Example: Factorial

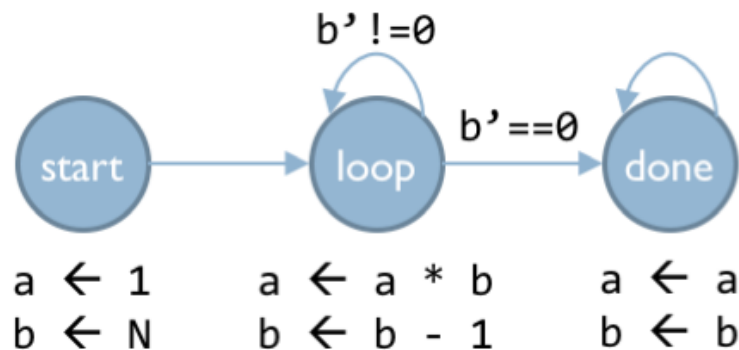
$$\text{factorial}(N) = N! = N * (N-1) * \dots * 1$$

C:

```
int a = 1;
int b = N;
do {
    a = a * b;
    b = b - 1;
} while (b != 0)
```

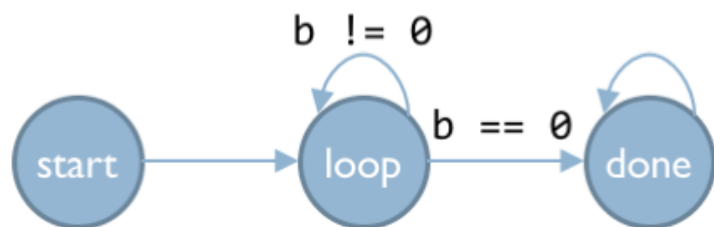
start: $a \leftarrow 1, b \leftarrow 5$
loop: $a \leftarrow 5, b \leftarrow 4$
loop: $a \leftarrow 20, b \leftarrow 3$
loop: $a \leftarrow 60, b \leftarrow 2$
loop: $a \leftarrow 120, b \leftarrow 1$
loop: $a \leftarrow 120, b \leftarrow 0$
done:

High-level FSM:



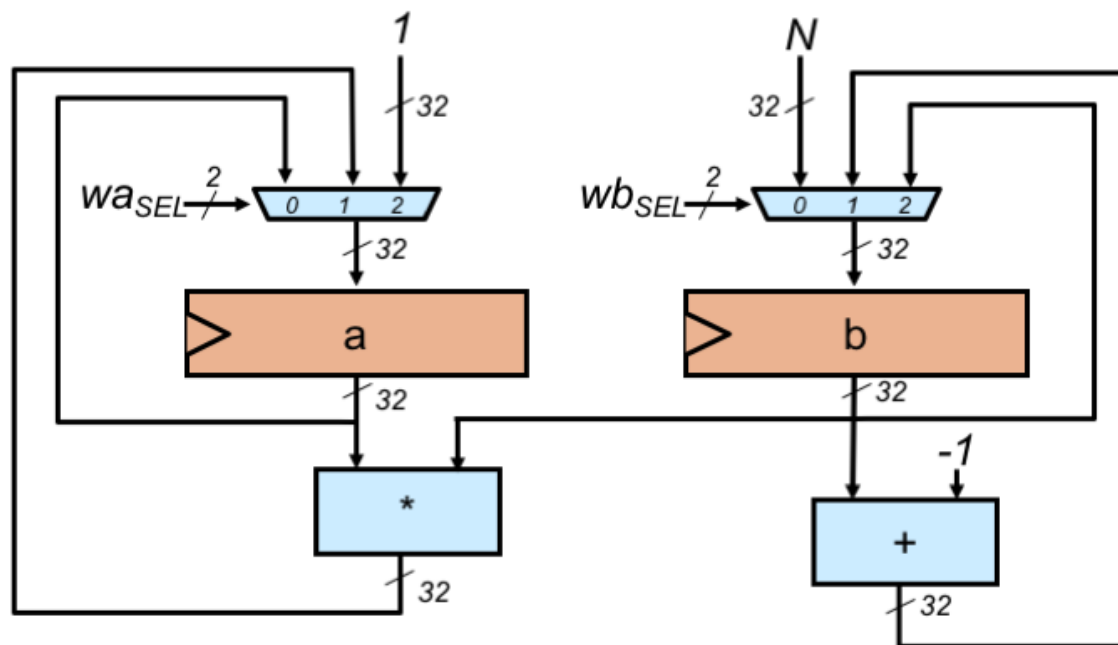
- Helpful to translate into hardware
- **D-registers** (a, b)
- 2-bits of state (start, loop, done)
- Boolean transitions ($b' == 0$, $b' \neq 0$)
- **Register assignments** in states (e.g., $a \leftarrow a * b$)

Datapath for Factorial

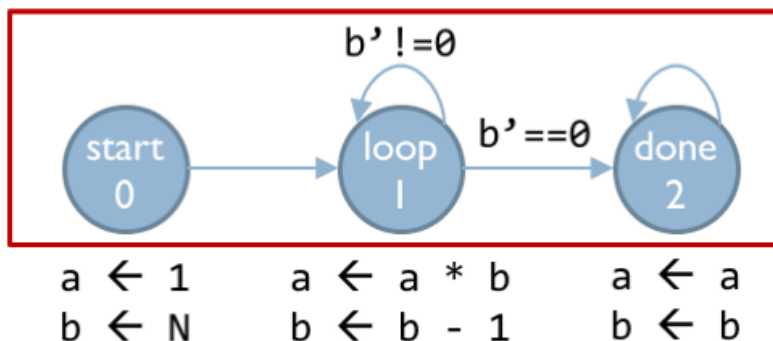


$a \leftarrow 1$	$a \leftarrow a * b$	$a \leftarrow a$
$b \leftarrow N$	$b \leftarrow b - 1$	$b \leftarrow b$

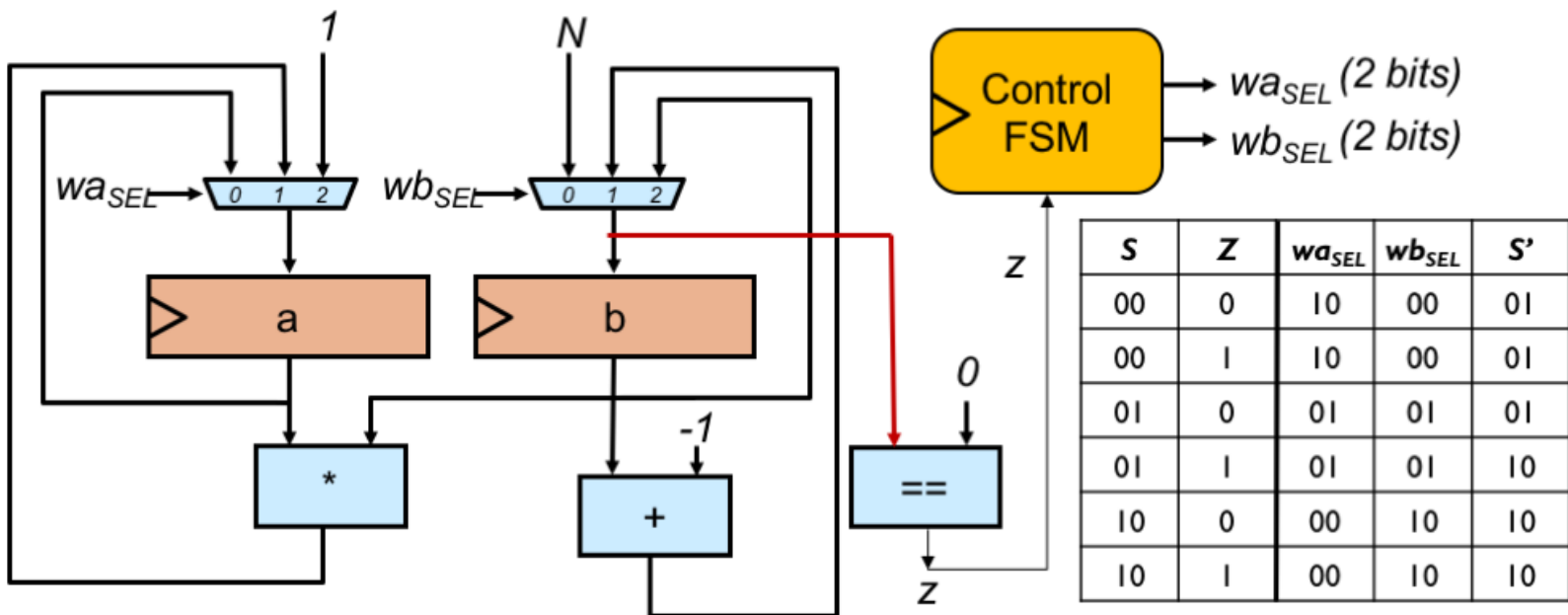
- Draw registers
- Draw combinational circuit for each assignment
- Connect to input muxes



Control FSM for Factorial



- Draw combinational logic for transition conditions
- Implement control FSM:
 - States: High-level FSM states
 - Inputs: Transition logic outputs
 - Outputs: Mux select signals



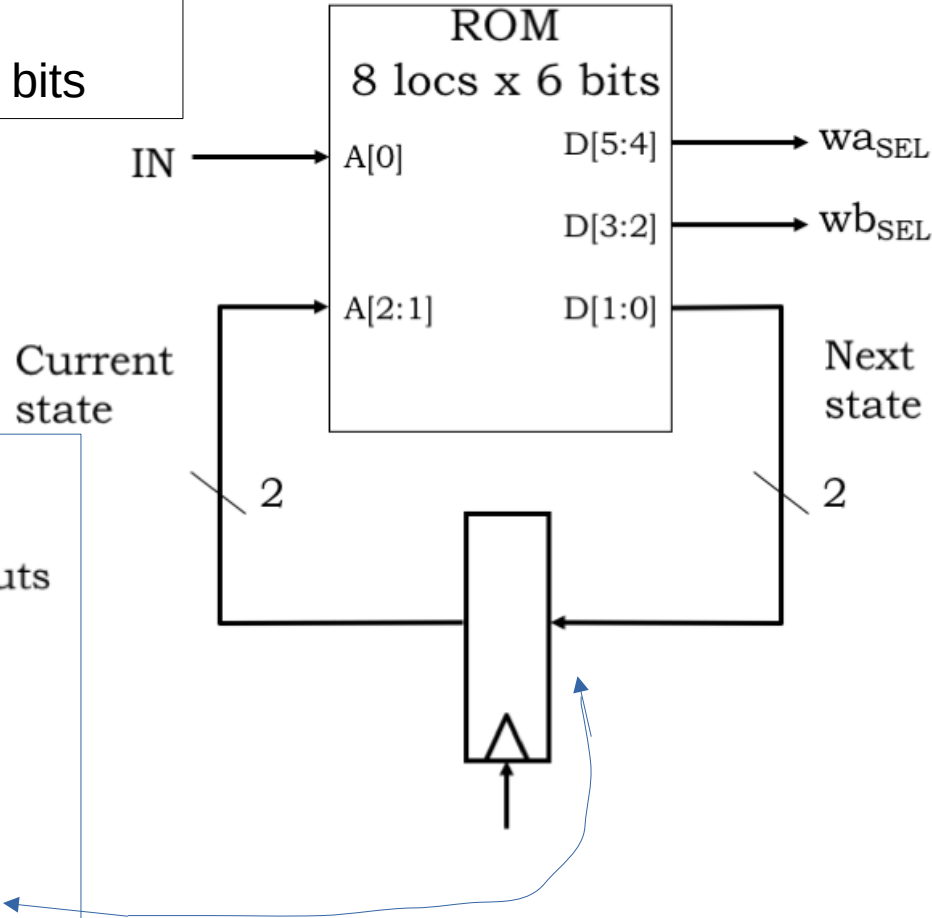
Z רלוונטי רק כאן כדי להעביר את המערכת למצב "2"

Control FSM Hardware

In: $2^3 = 8$

Out: $2 + 2 + 2 = 6$

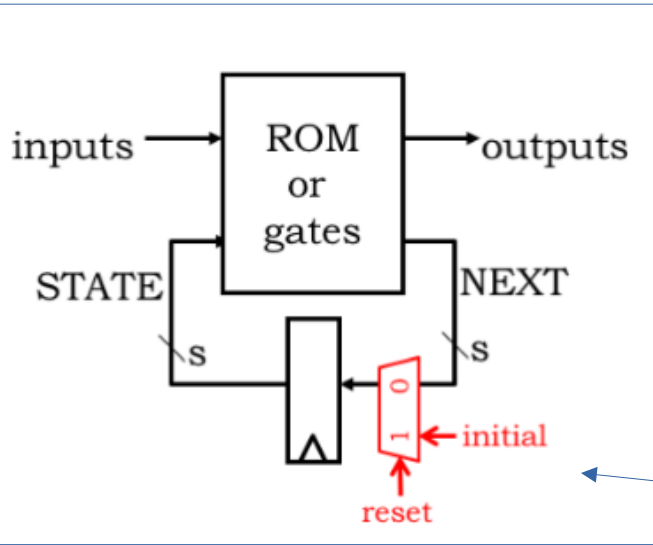
Total ROM size: 48 bits



ROM contents

A[2:0]	D[5:0]
000	10 00 01
001	10 00 01
010	01 01 01
011	01 01 10
100	00 10 10
101	00 10 10

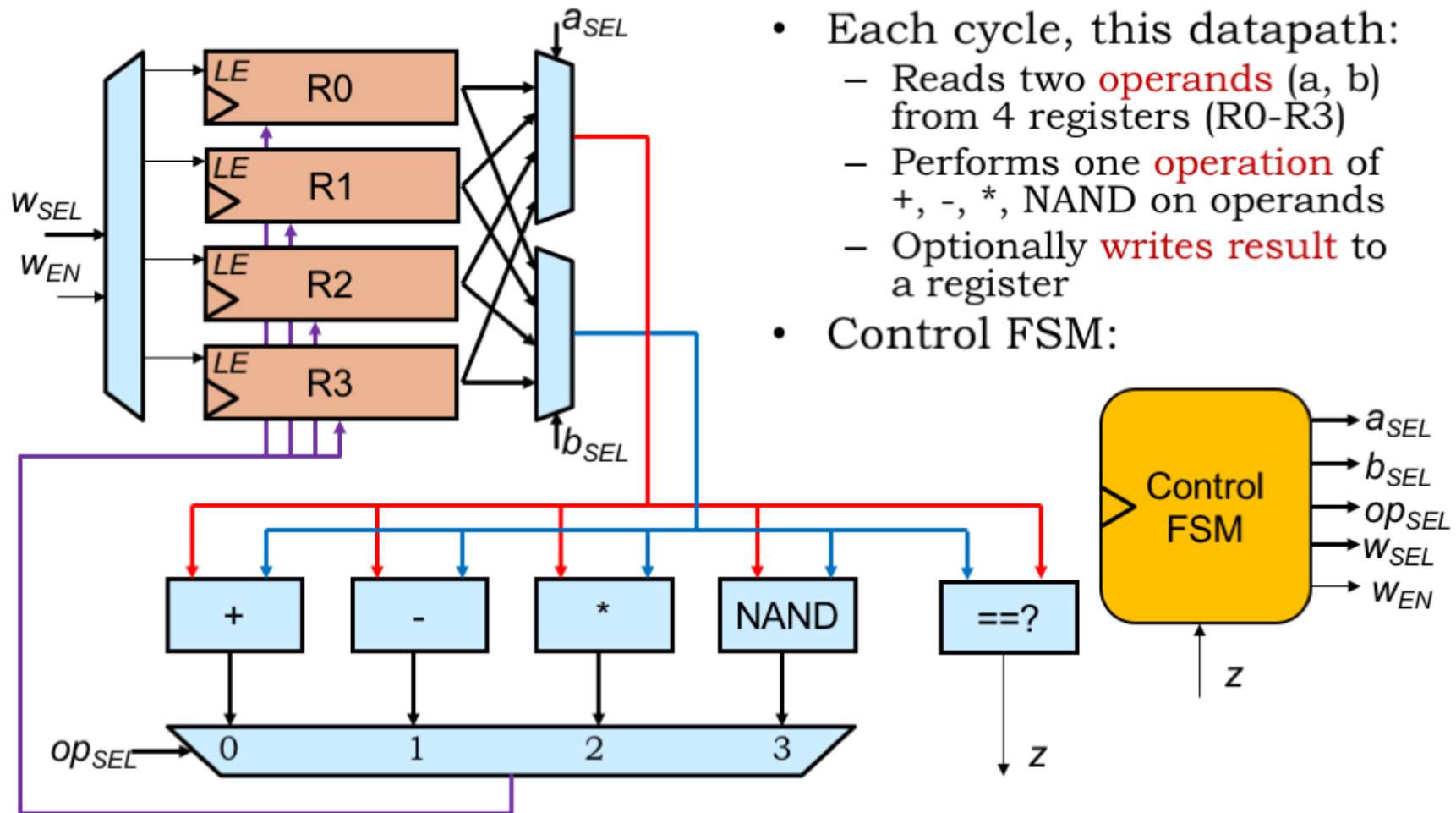
מיזוג של
העמודות
מהשקף הקודם



So Far: Single-Purpose Hardware

- Problem → Procedure (High-level FSM) → Implementation
- **Systematic way** to implement high-level FSM as a datapath + control FSM
 - Is this implementation an FSM itself?
 - If so, can you draw the truth table?
- How should we generalize our approach so we can solve many problems with one set of hardware?
 - More storage for operands and results
 - A larger repertoire of operations
 - General-purpose datapath

A Simple Programmable Datapath



A Control FSM for Factorial

- Assume initial register contents:

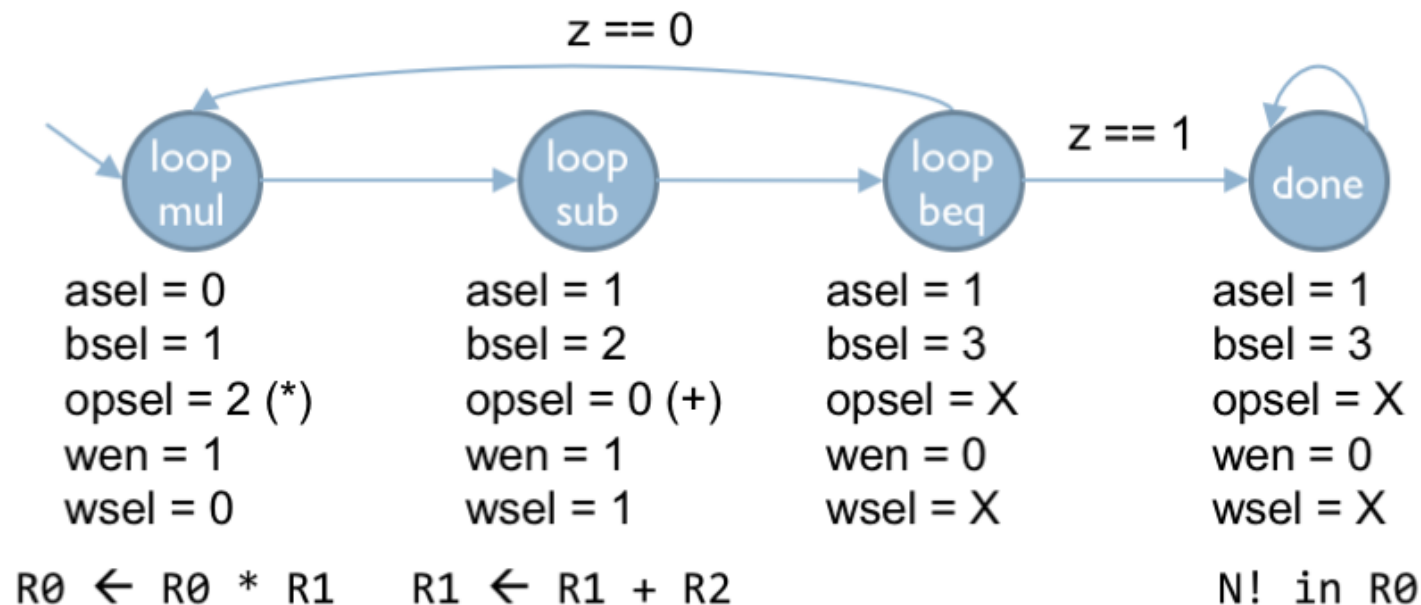
R0 value = 1

R1 value = N

R2 value = -1

R3 value = 0

- Control FSM:

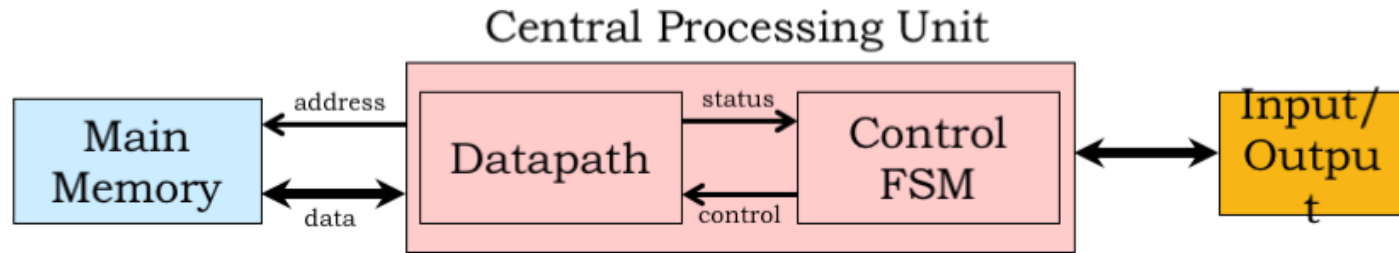


New Problem → New Control FSM

- You can solve many more problems with this datapath!
 - Exponentiation, division, square root, ...
 - But nothing that requires more than four registers
- By designing a control FSM, we are **programming the datapath**
- Early digital computers were programmed this way!
 - ENIAC (1943):
 - First general-purpose digital computer
 - Programmed by setting huge array of dials and switches
 - Reprogramming it took about 3 weeks

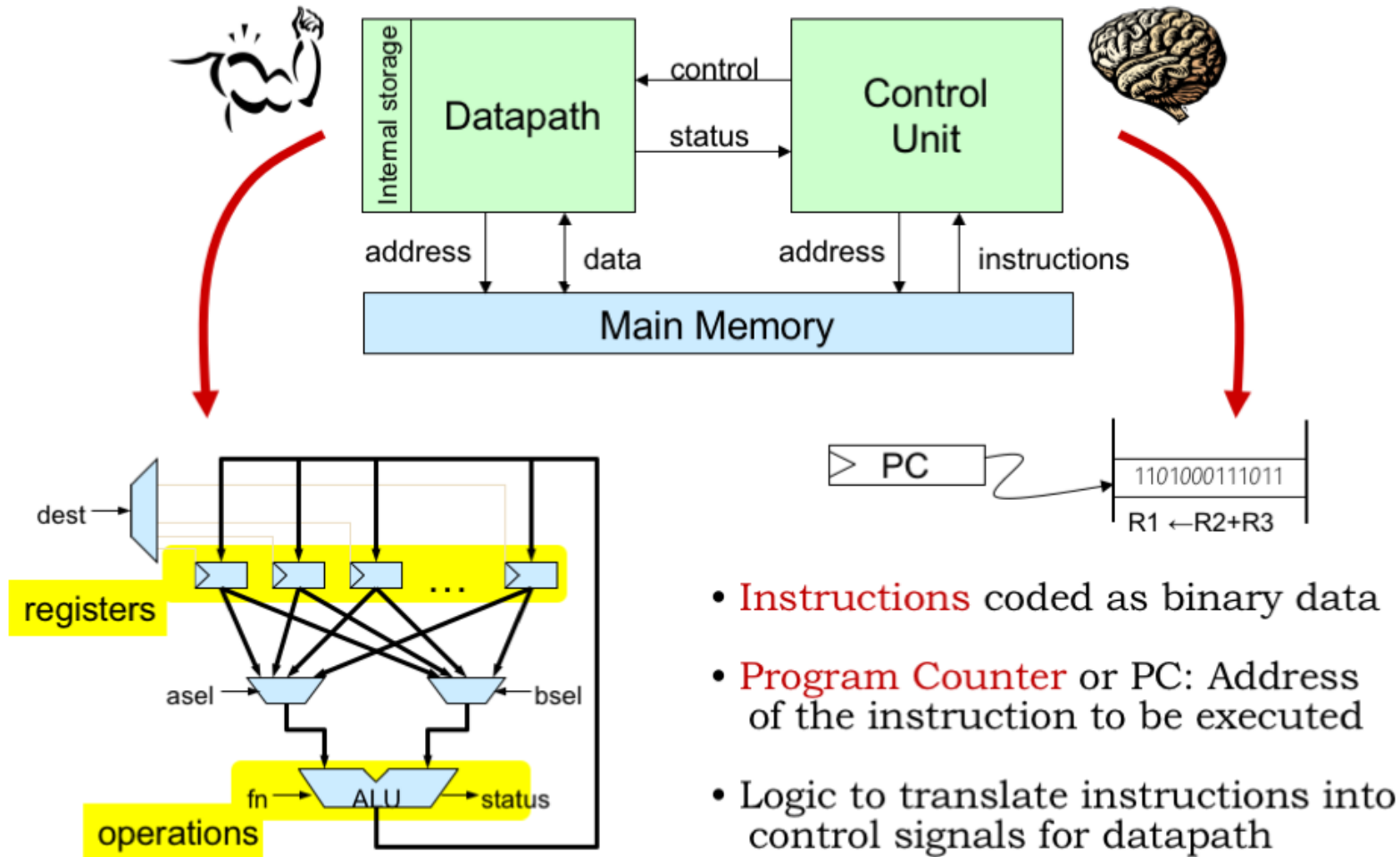
The von Neumann Model

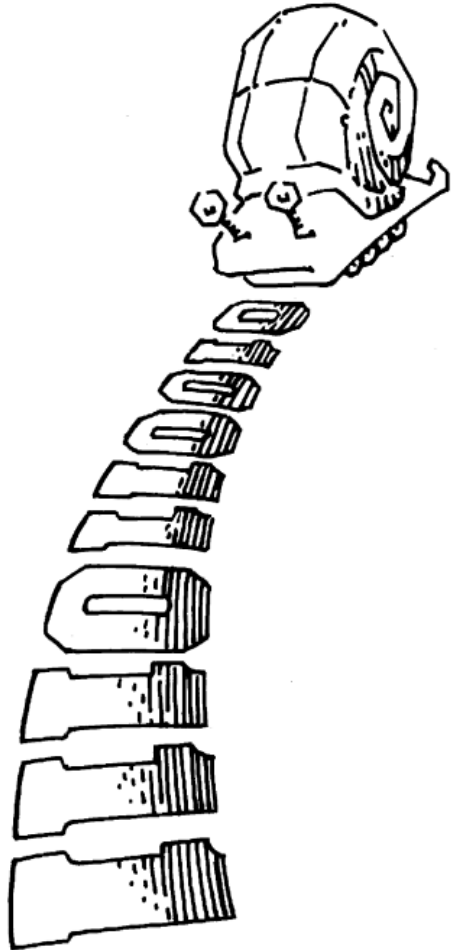
- Many approaches to build a general-purpose computer. Almost all modern computers are based on the von Neumann model (John von Neumann, 1945)
- Components:



- Central processing unit:
Performs operations on values in registers & memory
- Main memory:
Array of W words of N bits each
- Input/output devices to communicate with the outside world

Anatomy of a von Neumann Computer





הדגמה

The Snail's FSM

The Snail smiles if he detects the sequence: 01

- Reference: H&H Chapter 3, Example 3.7

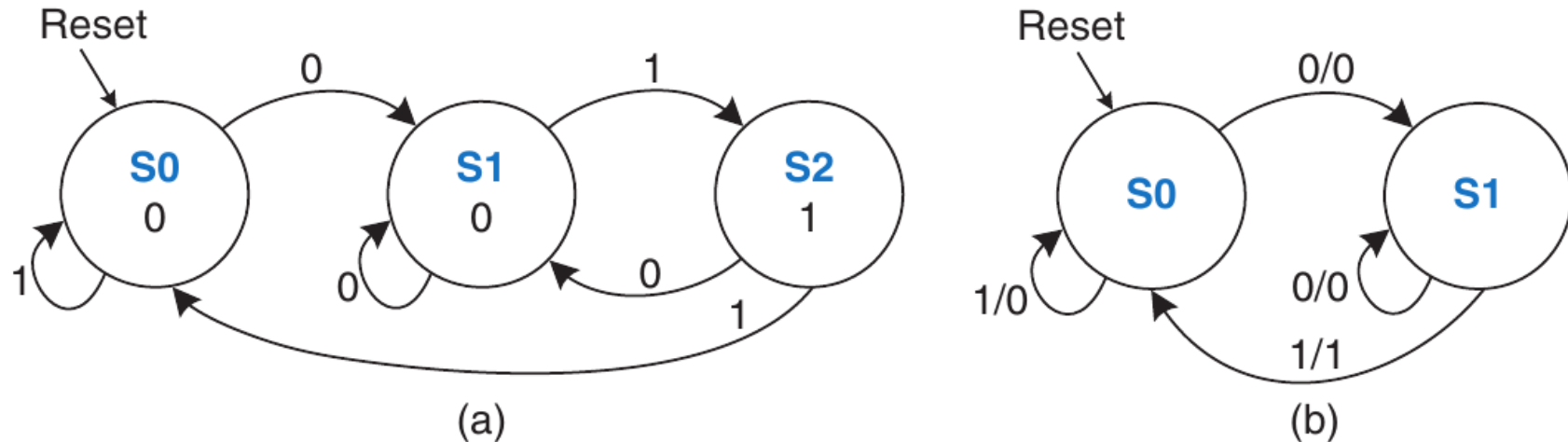


Figure 3.30 FSM state transition diagrams: (a) Moore machine, (b) Mealy machine

The Snail's truth tables

Table 3.11 Moore state transition table

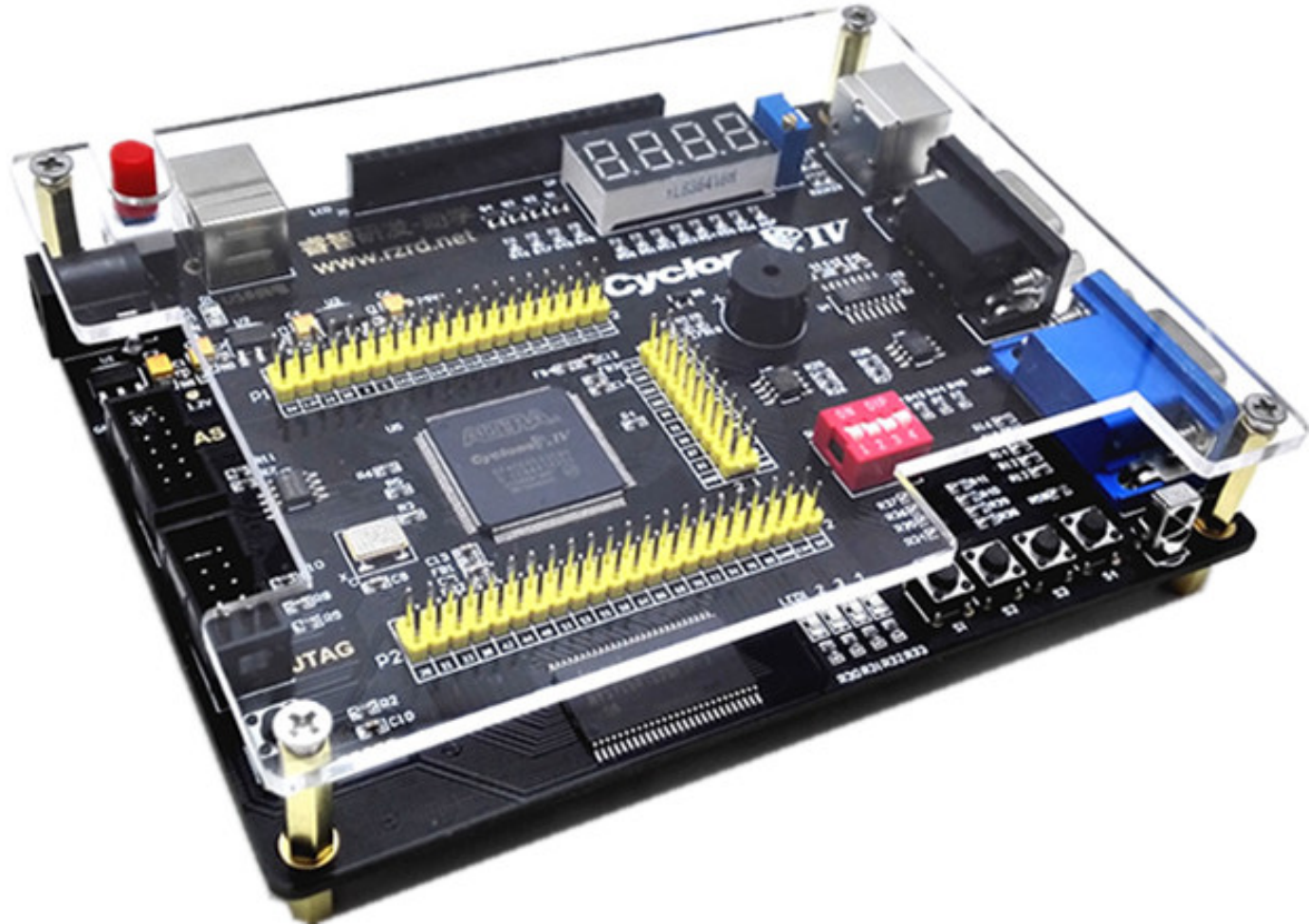
Current State S	Input A	Next State S'
S0	0	S1
S0	1	S0
S1	0	S1
S1	1	S2
S2	0	S1
S2	1	S0

Table 3.12 Moore output table

Current State S	Output Y
S0	0
S1	0
S2	1

Moore's FSM implementation in Verilog

Cyclone-IV FPGA dev. board



The happy Snail likes “1101”

Code development in *Quartus*

The screenshot displays the Intel Quartus Prime IDE interface. The main window shows a Verilog code file named `snail.v` with the following content:

```
1 // Smiling Snail FSM derived from David Harris & Sarah Harris chapter 3
2 // Ref: https://www.silicon-russia.com/2015/03/28/verilog-fpga-first-round-of-student-projects/
3 // Guy Tel-Zur, April 2023]
4 module pattern_fsm_moore
5 (
6     input clock,
7     input resetn,
8     input a, b, c, // a = input is 1, b = input is 0
9     output led_1, led_2, led_3, led_4
10 );
11
12 parameter [1:0] S0 = 0, S1 = 1, S2 = 2;
13 wire aa, bb, cc;
14
15 reg [1:0] state, next_state;
16
17 // state register
18
19 always @ (posedge clock or negedge resetn)
20 if (! resetn )
21     state <= S0;
22 else
23     state <= next_state;
24
25 assign aa = ~a; // click on 1
26 assign bb = ~b; // click on 0
27 assign cc = ~c; // a special click only for state S2
28
29 always @*
30
31 case (state)
```

The left sidebar shows the Project Navigator with the hierarchy: EntityInstance > Cyclone IV E: EP4CE6E22C8 > pattern_fsm_moore. The bottom-left pane shows the Messages window with the following content:

```
System (31) Processing (125)
Type ID Message
1 1 Quartus Prime Timing Analyzer was successful. 0 errors, 6 warnings
1 1 Running Quartus Prime EDA Netlist Writer
1 1 Command: quartus_eda --read_settings_files=off --write_settings_files=off snail -c snail
1 18236 Number of processors has not been specified which may cause overloading on shared machines. Set the global assignment NUM_PARALLEL_PROCESSORS in your QSF to an appropriate value for best performance.
1 204019 Generated file snail.vo in folder "/mnt/038d7f89-5713-462a-a96b-aa9afa3f6f93/opt/intelFPGA_lite/22.1std/simulation/questa/" for EDA simulation tool
1 1 Quartus Prime EDA Netlist Writer was successful. 0 errors, 1 warning
1 293000 Quartus Prime Full Compilation was successful. 0 errors, 21 warnings
```

The bottom-right pane shows the IP Catalog with the Project Directory and Library sections.

Flash the code to the device

File Edit View Processing Tools Window Help

Hardware Setup... USB-Blaster [1-3.3.3] Mode: JTAG Progress: 100% (Successful)

☐ Enable real-time ISP to allow background programming when available

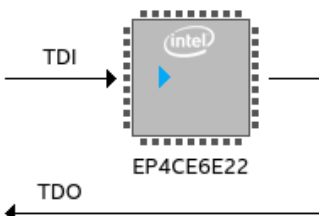
Start Stop Auto Detect Delete Add File... Change File... Save File Add Device... Up Down

File	Device	Checksum	Usercode	Program/Configure	Verify	Blank-Check	Examine
output_files/snail.sof	EP4CE6E22	000953C1	000953C1	☒	☐	☐	☐

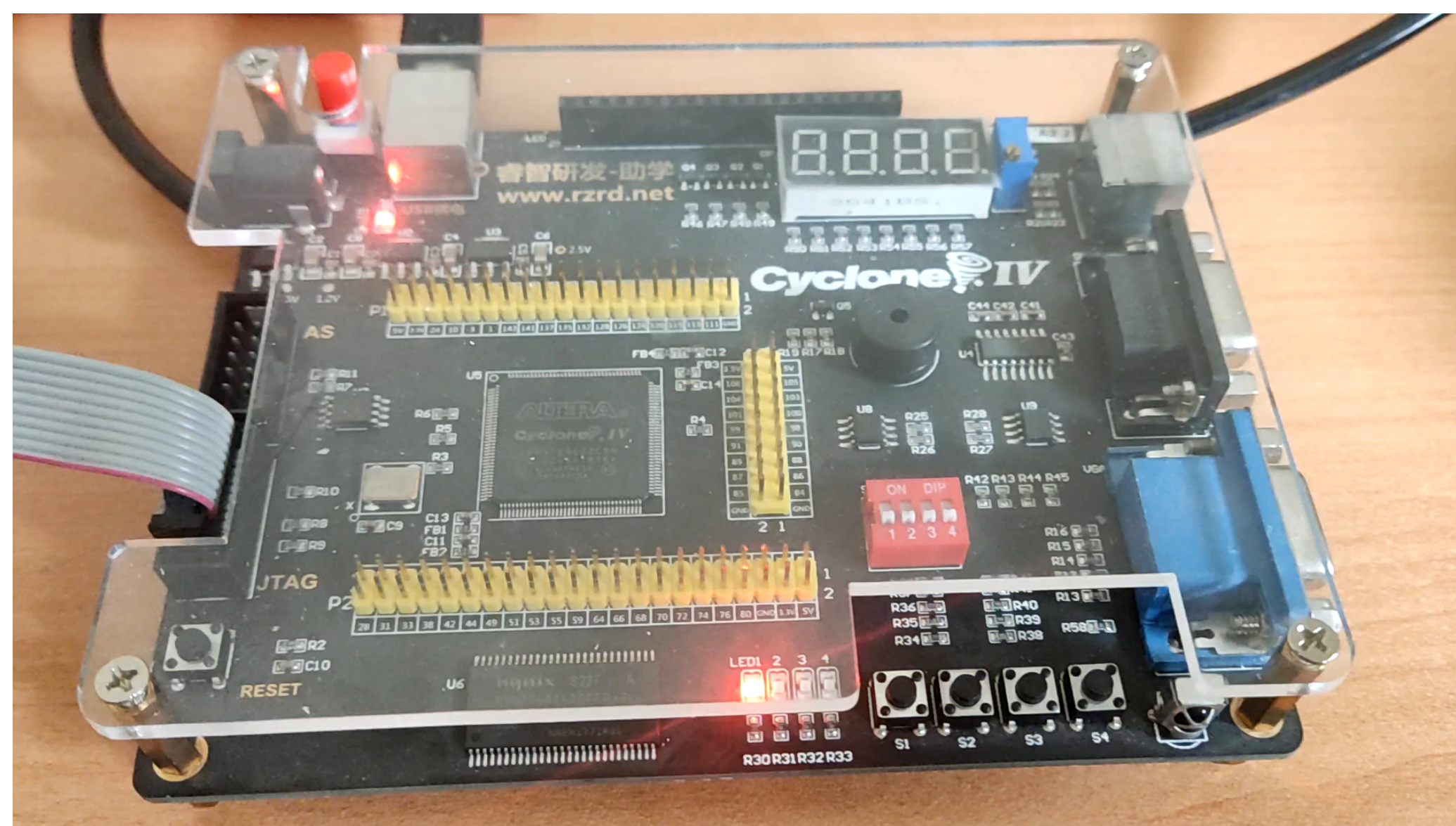
TDI

TDO

EP4CE6E22



The diagram shows an Intel EP4CE6E22 FPGA chip. A TDI (Test Data In) pin is connected to the top-left pin of the chip, and a TDO (Test Data Out) pin is connected to the bottom-left pin of the chip. The chip is labeled 'intel' and 'EP4CE6E22'.



**Watch the video on
Youtube:**

[https://youtu.be/K3Eri9pF
CKQ](https://youtu.be/K3Eri9pFCKQ)

Harris & Harris - The Snail Challenge

Exercise 3.25 Alyssa P. Hacker's snail from Section 3.4.3 has a daughter with a Mealy machine FSM brain. The daughter snail smiles whenever she slides over the pattern 1101 ~~or the pattern 1110~~. Sketch the state transition diagram for this happy snail using as few states as possible. Choose state encodings and write a combined state transition and output table using your encodings. Write the next state and output equations and sketch your FSM schematic.

Implementation on Tang Nano 9k FPGA

הרחבת הסקופ: חשיבה קצת מחוץ
לקופסת ה-FSM הקלאסית
(מדובר במכונת Moore)

This time we use a **shift register** that holds the a
'pattern' which is the last 4 values the snail traversed.
When the pattern is '1101' the LED lit.
The Verilog code is in the next slide.

```

module smiling_snail(
    input clk,
    input btn0,    // 0 input
    input btn1,    // 1 input
    output reg LED0, // 0 pressed
    output reg LED1, // 1 pressed
    output reg LED3 // snail smiles (1101 reached)
);
reg [3:0] pattern;
reg btn0_prev, btn1_prev;
initial begin
    pattern = 4'b0000; // pattern is a shift register
    LED0 = 1'b0;
    LED1 = 1'b0;
    LED3 = 1'b0;
    btn0_prev = 1'b0;
    btn1_prev = 1'b0;
end

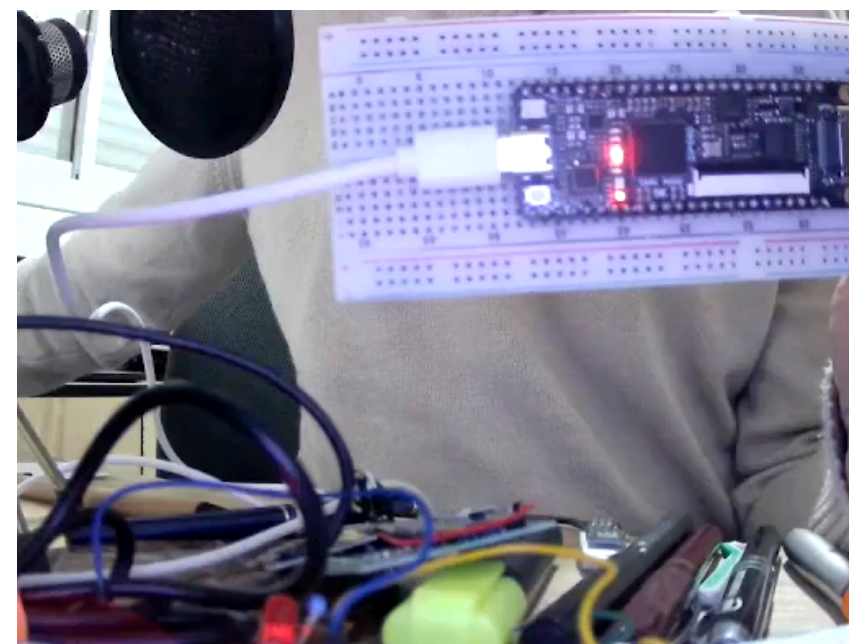
```

```

always @(posedge clk) begin
    // Detect button presses
    if (btn0 && !btn0_prev) begin
        pattern <= {pattern[2:0], 1'b0}; // לחיצה עם 0 מוסיפה 0 מימין
    end else if (btn1 && !btn1_prev) begin
        pattern <= {pattern[2:0], 1'b1}; // לחיצה על 1 מוסיפה 1 מימין
    end
    // Check for 1101 pattern
    LED3 <= ~(pattern == 4'b1101);
    // Update button states
    btn0_prev <= btn0;
    btn1_prev <= btn1;
    // Update LED0 and LED1
    LED0 <= btn0;
    LED1 <= btn1;
end
endmodule

```

a short video about that is in the next slide...



smiling_snail.v - v4_tang_nano - Visual Studio Code

File Edit Selection View Go Run Terminal Help

EXPLORER

- ✓ V4_TANG_NANO
 - Makefile
 - readme_guy.txt
 - smiling_snail.fs
 - smiling_snail.v
 - tangnano9k.cst

smiling_snail.v

```
1 module smiling_snail(  
2     input clk,  
3     input btn0,    // 0 input  
4     input btn1,    // 1 input  
5     output reg LED0, // 0 pressed  
6     output reg LED1, // 1 pressed  
7     output reg LED3 // snail smiles (1101 reached)  
8 );  
9  
10 reg [3:0] pattern;  
11 reg btn0_prev, btn1_prev;  
12  
13 initial begin  
14     pattern = 4'b0000; // pattern is a shift register  
15     LED0 = 1'b0;  
16     LED1 = 1'b0;  
17     LED3 = 1'b0;  
18     btn0_prev = 1'b0;  
19     btn1_prev = 1'b0;  
20 end  
21  
22 always @(posedge clk) begin  
23     // Detect button presses  
24     if (btn0 && !btn0_prev) begin  
25         pattern <= {pattern[2:0], 1'b0}; // 0 מימין  
26     end else if (btn1 && !btn1_prev) begin  
27         pattern <= {pattern[2:0], 1'b1}; // 1 מימין  
28     end  
29  
30     // Check for 1101 pattern  
31     LED3 <= ~(pattern == 4'b1101);  
32  
33     // Update button states  
34     btn0_prev <= btn0;  
35     btn1_prev <= btn1;  
36  
37     // Update LED0 and LED1  
38     LED0 <= btn0;  
39     LED1 <= btn1;  
40 end
```

OUTLINE
TIMELINE

Ln 29, Col 1 Spaces: 4 UTF-8 LF Verilog

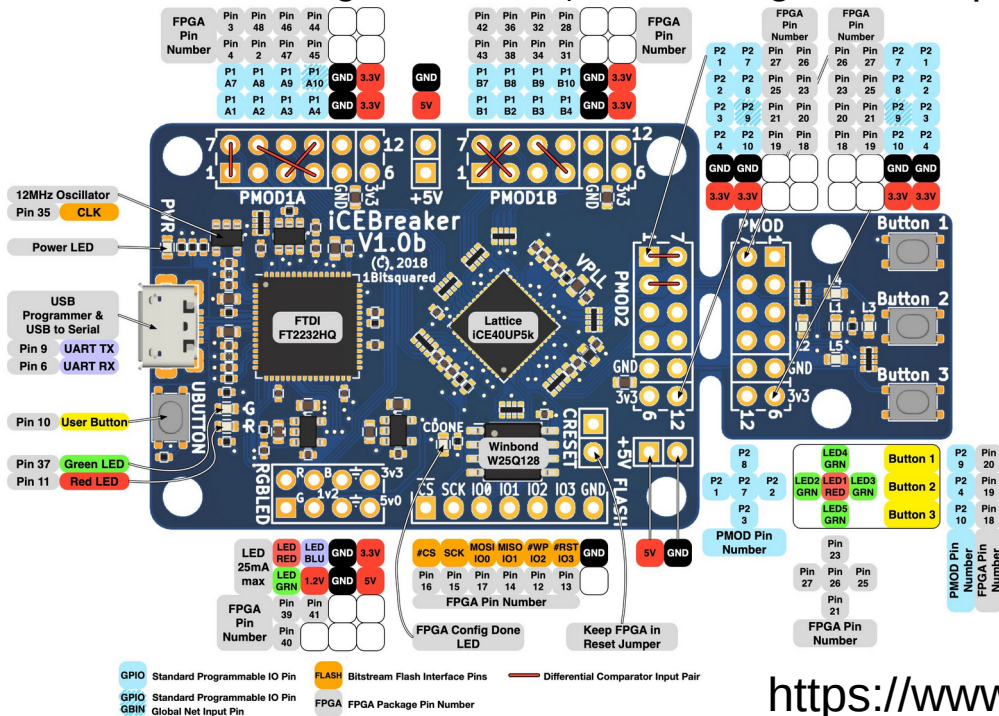
Similar code on an IceBreaker FPGA

Folder:

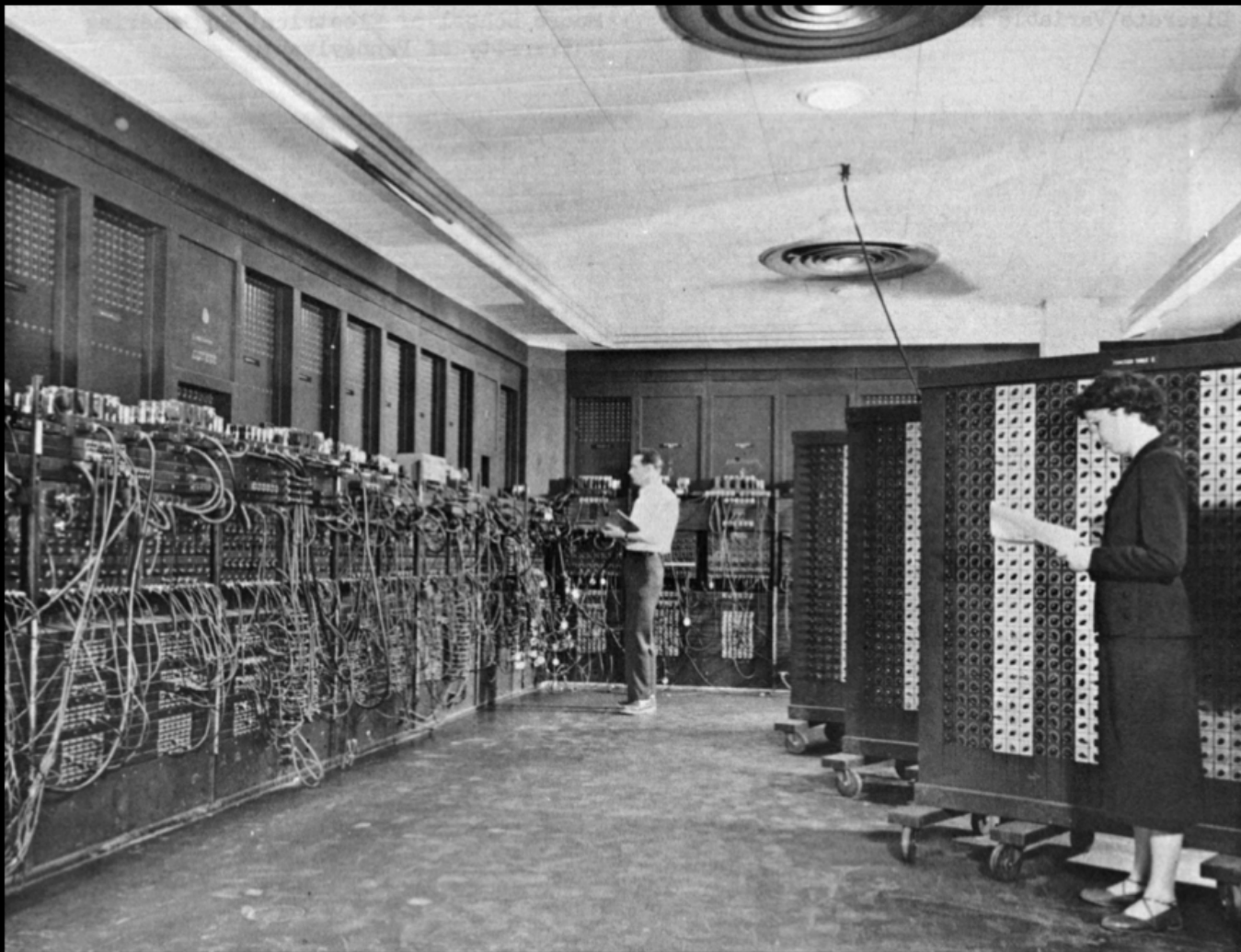
/home/telzur/science/Teaching/CPU/lectures/04/code/Harris_Snail/ICE40

Files:

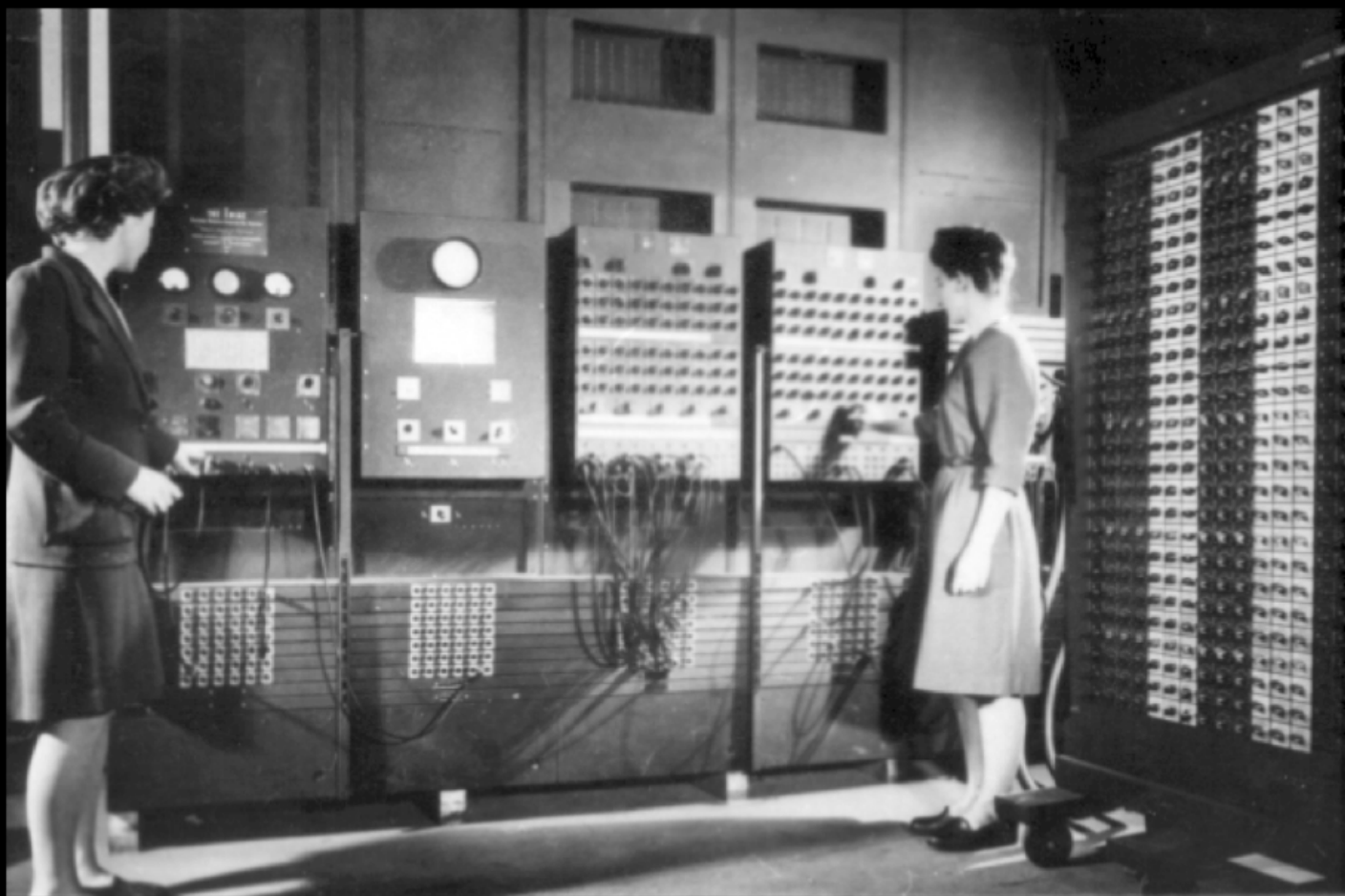
smiling_snail.v, smiling_snail.pcf, makeit.sh



<https://www.crowdsupply.com/1bitsquared/icebreaker-fpga>



"Eniac" by Unknown - U.S. Army Photo.



U.S. Army Photo.

Part 2: Pipeline Design

Computer Architecture

Dr. Guy Tel-Zur

מטרות המצגת

- ביסוס הבנת מושג ה-Pipeline
- מדדים המשפיעים על הביצועים
- מונחים מרכזיים:
- דרגות ה-Pipeline, חוקיו, עקרונות התכנון
- מתן דוגמה

This presentation is based on:

- MIT Open course ware "**Computation Structures**" by Dr. Christopher J. Terman.
- References:

- [MIT Open course ware](#)

MIT6.004 2017

- Slides 13-15 :

[Slides by Prof. Shmuel Wimer Technion/BIU](#)

In particular we will discuss material from:

<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-004-computation-structures-spring-2017/c7/c7s1/#1>

See 7.2.2, 7.2.3, 7.2.4 @ MIT open course ware site

Chris Terman. 6.004 Computation Structures. Spring 2017. Massachusetts Institute of Technology: MIT OpenCourseWare, <https://ocw.mit.edu>. License: Creative Commons BY-NC-SA.

Forget circuits... Let's Solve a Real Problem

INPUT:
dirty laundry



OUTPUT:
6 more weeks



Device: Washer

Function: Fill, Agitate, Spin

Washer_{PD} = 30 mins



Device: Dryer

Function: Heat, Spin

Dryer_{PD} = 60 mins

PD = Propagation delay, Period

One Load At a Time

Everyone knows that the real reason that we put off doing laundry so long is not because we procrastinate, are lazy, or even have better things to do.

The fact is, doing one load at a time is not smart.

Step 1:



Step 2:



$$\begin{aligned}\text{Total}_{\text{PD}} &= \text{Washer}_{\text{PD}} + \text{Dryer}_{\text{PD}} \\ &= \underline{\quad 90 \quad} \text{ mins}\end{aligned}$$

Doing N Loads of Laundry

Here's how they do laundry at Harvard, the “combinational” way.

Of course, this is just an urban legend. No one at Harvard actually *does* laundry. The butlers all arrive on Wednesday morning, pick up the dirty laundry and return it all pressed and starched in time for afternoon tea.



Step 1:



Step 2:



Step 3:



Step 4:



...

$$\begin{aligned} \text{Total}_{PD} &= N * (\text{Washer}_{PD} + \text{Dryer}_{PD}) \\ &= \underline{N*90} \text{ mins} \end{aligned}$$

Doing N Loads... The 6.004 Way

6.004 students
“pipeline” the laundry
process.

That’s why we wait!

Actually, it’s more like $N*60 + 30$ if we account for the startup transient correctly. When doing pipeline analysis, we’re mostly interested in the “steady state” where we assume we have an infinite supply of inputs.



$$\text{Total}_{PD} = N * \text{Max}(\text{Washer}_{PD}, \text{Dryer}_{PD})$$

$$= \underline{N*60} \text{ mins}$$

לפי האיטי מבין השניים

Performance Measures

Latency:

The delay from when an input is established until the output associated with that input becomes valid.

$$\text{Harvard Laundry} = \frac{90}{120} \text{ mins}$$

$$6.004 \text{ Laundry} = \frac{120}{120} \text{ mins}$$

Assuming that the wash is started as soon as possible and waits (wet) in the washer until dryer is available.

Throughput:

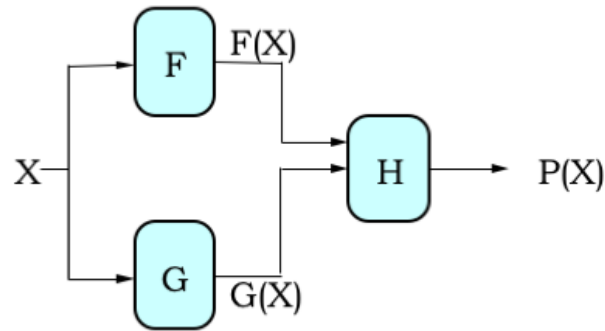
The *rate* at which inputs or outputs are processed.

$$\text{Harvard Laundry} = \frac{1/90}{1/60} \text{ outputs/min}$$

$$6.004 \text{ Laundry} = \frac{1/60}{1/60} \text{ outputs/min}$$

פריט יוצא כל
60 דקות

Okay, Back To Circuits...

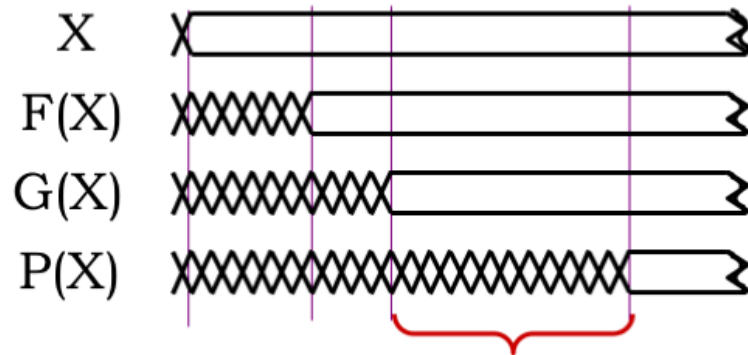


For combinational logic:

latency = t_{PD} ,

throughput = $1/t_{PD}$.

We can't get the answer faster,
but are we making effective use
of our hardware at all times?

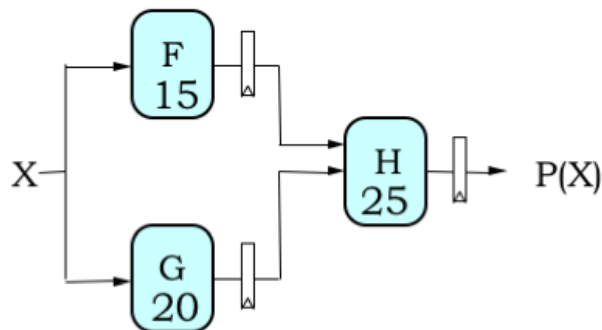


F & G are “idle”, just holding their outputs
stable while H performs its computation



Pipelined Circuits

use registers to hold H's input stable!

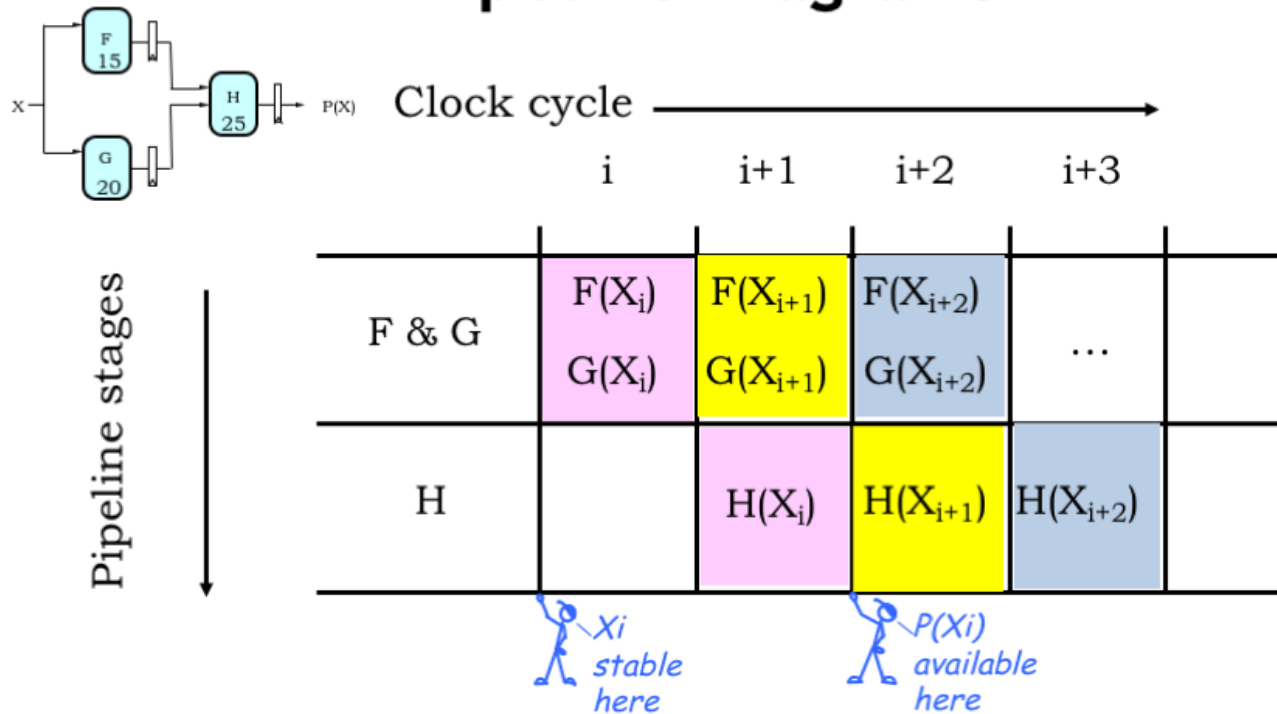


Now F & G can be working on input X_{i+1} while H is performing its computation on X_i . We've created a 2-stage *pipeline*: if we have a valid input X during clock cycle j, P(X) is valid during clock j+2.

Suppose F, G, H have propagation delays of 15, 20, 25 ns and we are using ideal zero-delay registers:

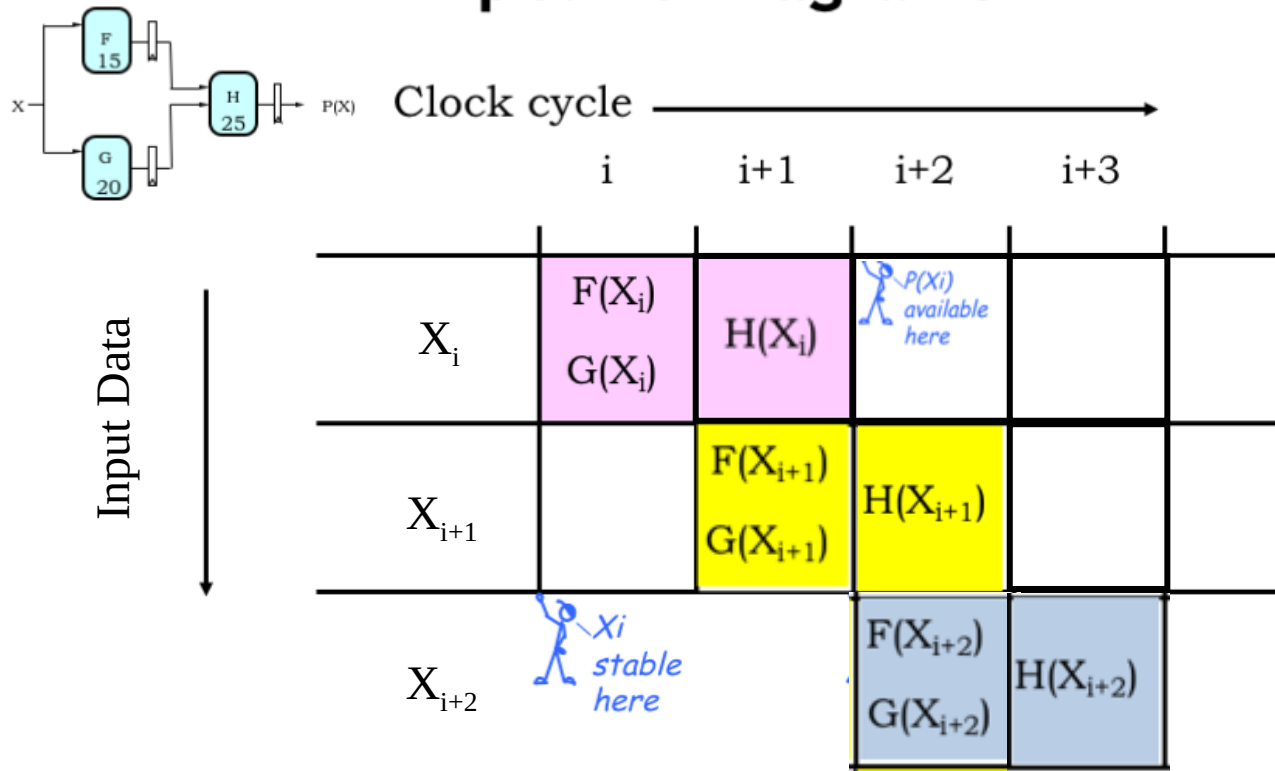
	<u>latency</u>	<u>throughput</u>
unpipelined	45	1/45
2-stage pipeline	<u>50</u> worse 	<u>1/25</u> better 

Pipeline Diagrams



The results associated with a particular set of input data moves *diagonally* through the diagram, progressing through one pipeline stage each clock cycle.

Pipeline Diagrams



The results associated with a particular set of input data moves *diagonally* through the diagram, progressing through one pipeline stage each clock cycle.

Pipeline Conventions

DEFINITION:

A well-formed *K-Stage Pipeline* (“K-pipeline”) is an acyclic circuit having exactly K registers on *every* path from an input to an output.

a COMBINATIONAL CIRCUIT is thus an 0-stage pipeline.

COMPOSITION CONVENTION:

Every pipeline stage, hence every K-Stage pipeline, has a register on its *OUTPUT* (not on its input).

ALWAYS:

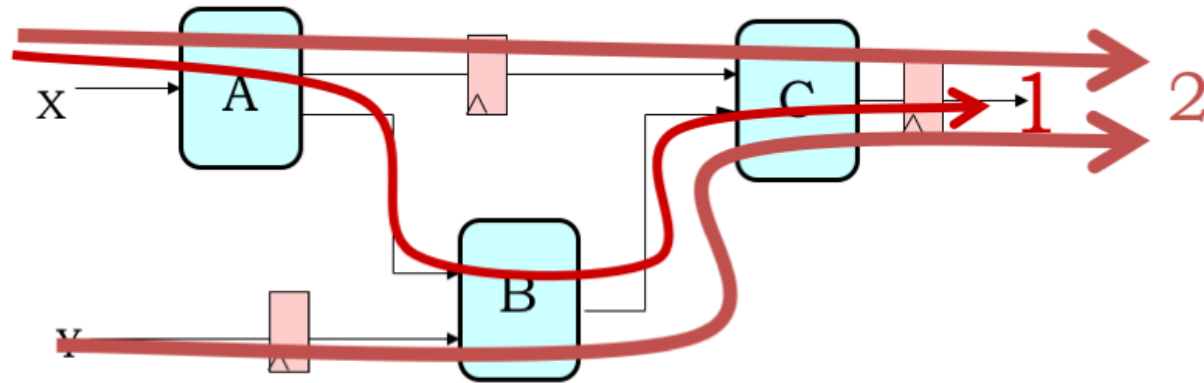
The CLOCK common to all registers must have a period sufficient to cover propagation over combinational paths PLUS (input) register t_{PD} PLUS (output) register t_{SETUP} .

The LATENCY of a K-pipeline is K times the period of the system's clock.

The THROUGHPUT of a K-pipeline is the frequency of the clock.

Ill-formed Pipelines

Consider a BAD job of pipelining:



For what value of K is the following circuit a K-Pipeline?

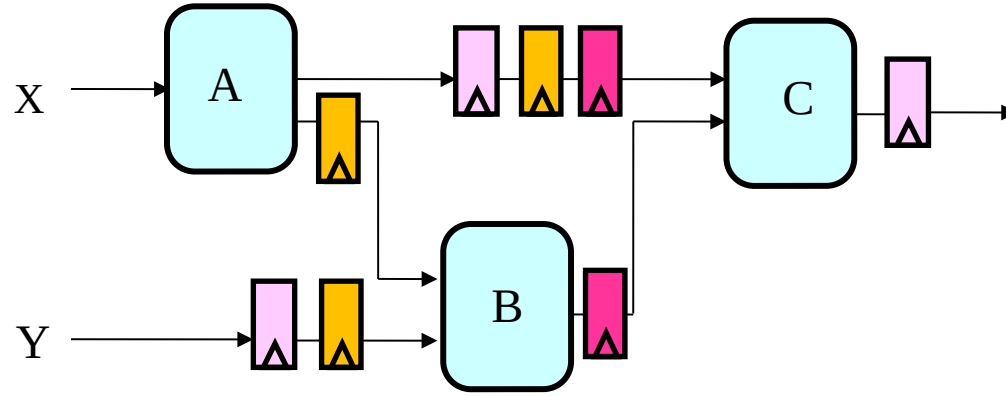
ANS: none

Problem:

Successive inputs get mixed: e.g., $B(A(X_{i+1}), Y_i)$. This happened because some paths from inputs to outputs have 2 registers, and some have only 1!

This CAN'T HAPPEN on a well-formed K pipeline!

fixed Pipeline



This is mainly a 4-pipeline with a twist of 3-pipeline to allow mixing $A(X_{i+1})$ and Y_i at the input of B.



תכנון שיטתי של Pipeline

- הגדרה: **K - Pipeline**

- מעגל לוגי ללא משוב
- כולל רכיבים צירופיים ואוגרים
- כל מסלול מכניסה ליציאה כולל בדיוק **K** אוגרים

A Pipelining Methodology

Step 1:

Draw a line that crosses every output in the circuit, and mark the endpoints as terminal points.

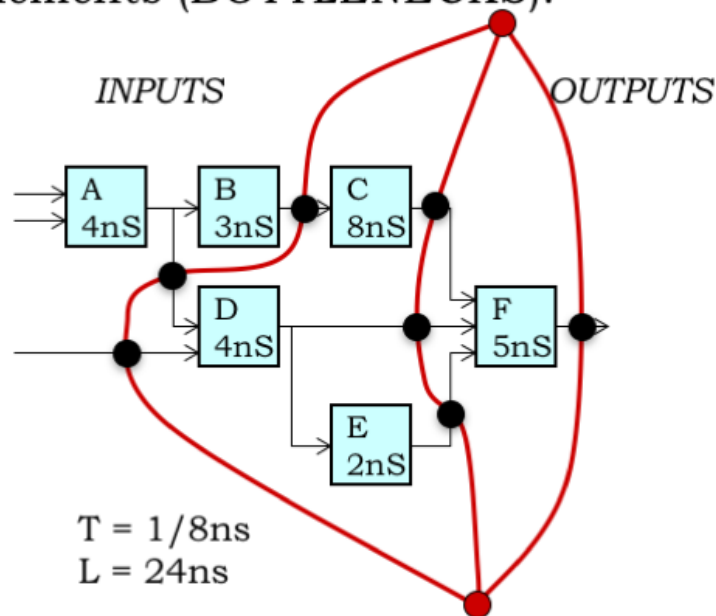
Step 2:

Continue to draw new lines between the terminal points across various circuit connections, ensuring that every connection crosses each line in the same direction. These lines demarcate *pipeline stages*.

Adding a pipeline register at every point where a separating line crosses a connection will always generate a valid pipeline.

STRATEGY:

Focus your attention on placing pipelining registers around the slowest circuit elements (BOTTLENECKS).



לתחום

A pipelining methodology

Step 1:

Add a register on each output.

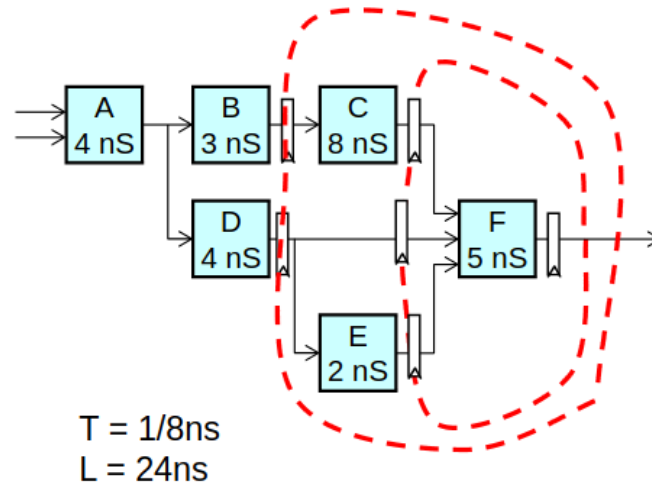
Step 2:

Add another register on each output. Draw a cut-set contour that includes all the new registers and some part of the circuit. Retime by moving regs from all outputs to all inputs of cut-set.

Repeat until satisfied with T.

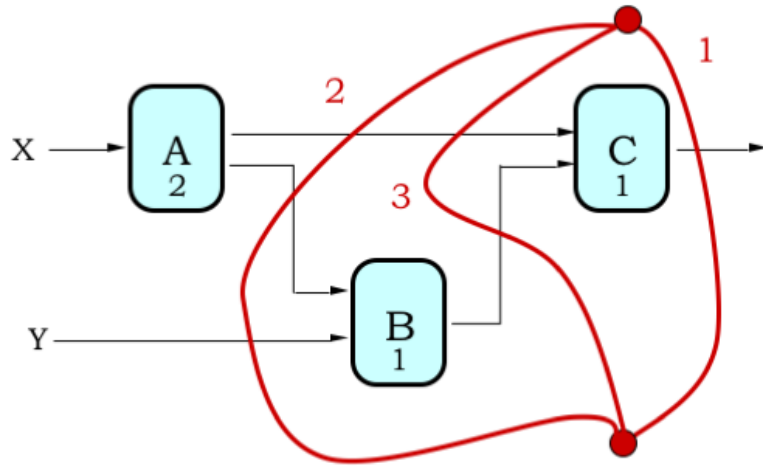
STRATEGY:

Focus your attention on placing pipelining registers around the slowest circuit elements (BOTTLENECKS).



Ref: <https://web.mit.edu/6.111/www/f2016/handouts/L09.pdf>

Pipeline Example



OBSERVATIONS:

- 1-pipeline improves neither L or T.
- T improved by breaking long combinational paths, allowing faster clock.
- Too many stages cost L, don't improve T.
- Back-to-back registers are often required to keep pipeline well-formed.

מערכת צרופים בלבד

קו 1

קווים 1 + 2

קוים

1 + 2 + 3

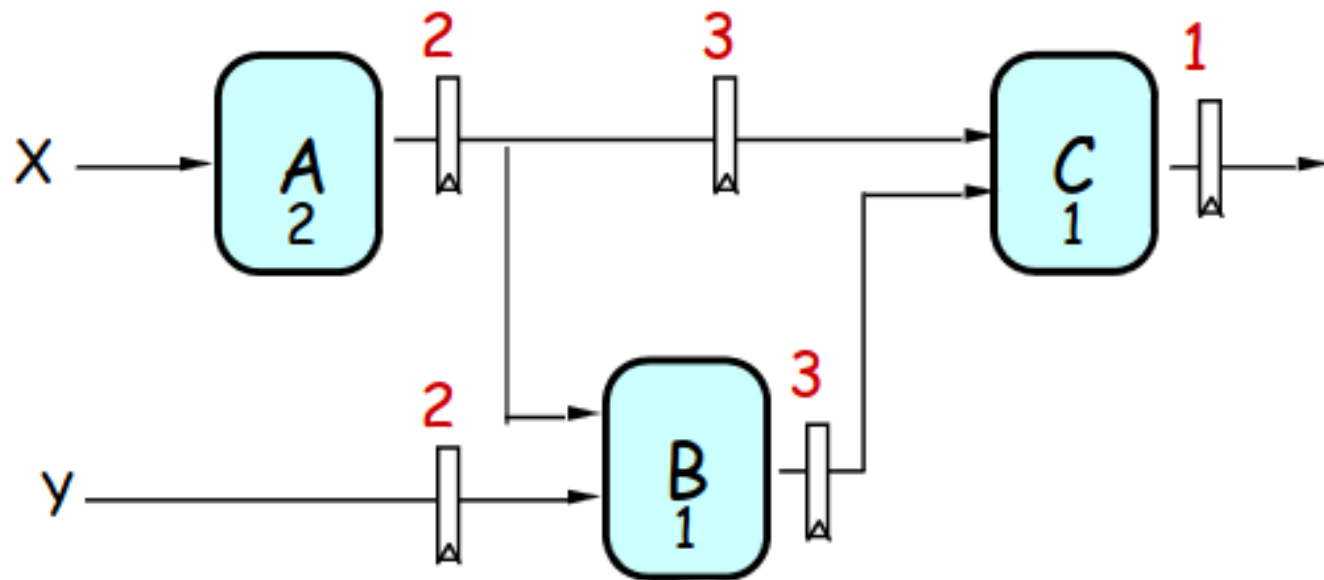
	LATENCY	THROUGHPUT
0-pipe:	4	1/4
1-pipe:	4	1/4
2-pipe:	4	1/2
3-pipe:	6	1/2

+ increase throughput
- increase latency
- "bottleneck" problem

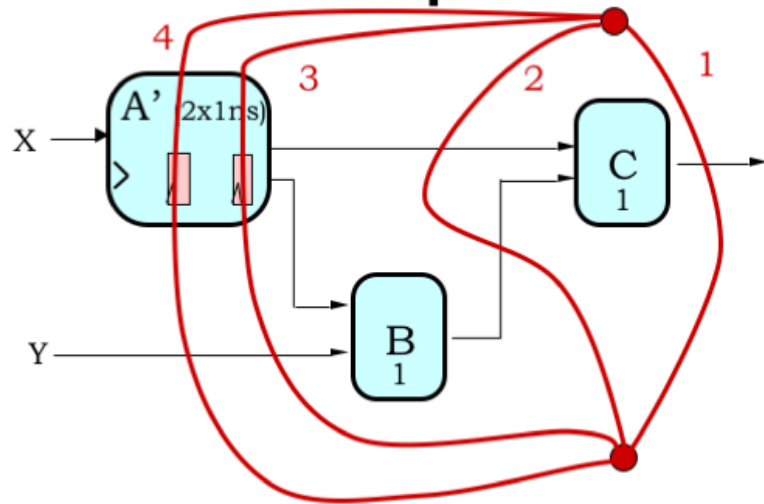
3 דרגות, כל אחת 2 נ"ש, אין שיפור

2 דרגות, כל אחת 2 נ"ש

אותו התרשים



Pipelined Components



4-stage pipeline, throughput=1

Pipelined systems can be hierarchical:

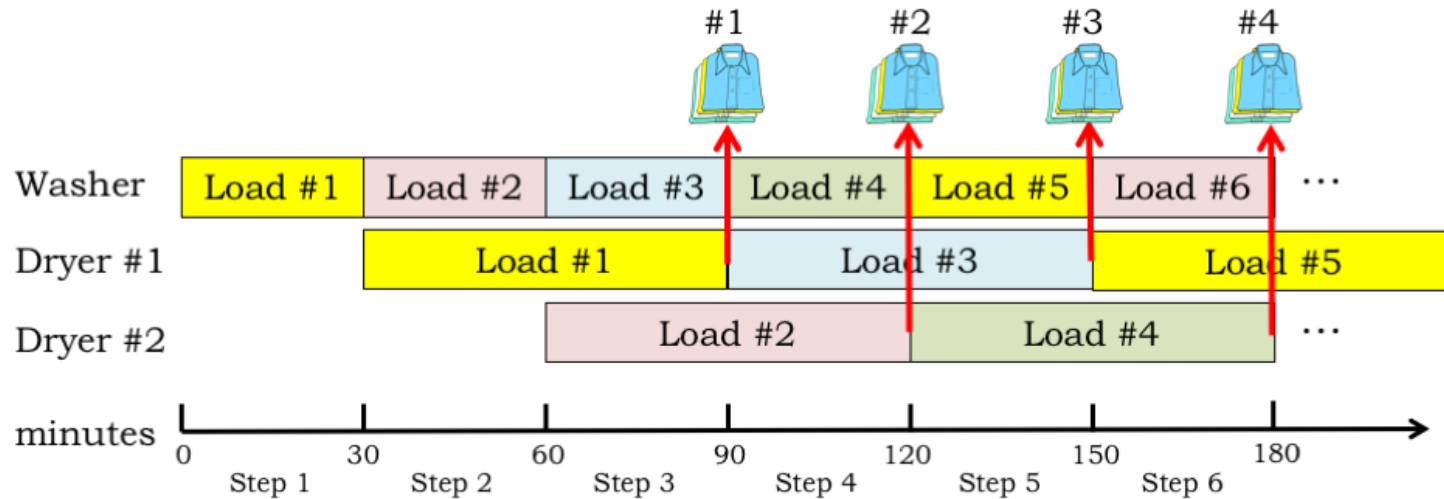
- Replacing a slow combinational component with a k-pipe version may let us decrease the clock period
- Must account for new pipeline stages in our plan



*but... but...
How can I pipeline
a clothes dryer???*

How Do 6.004 Students Do Laundry?

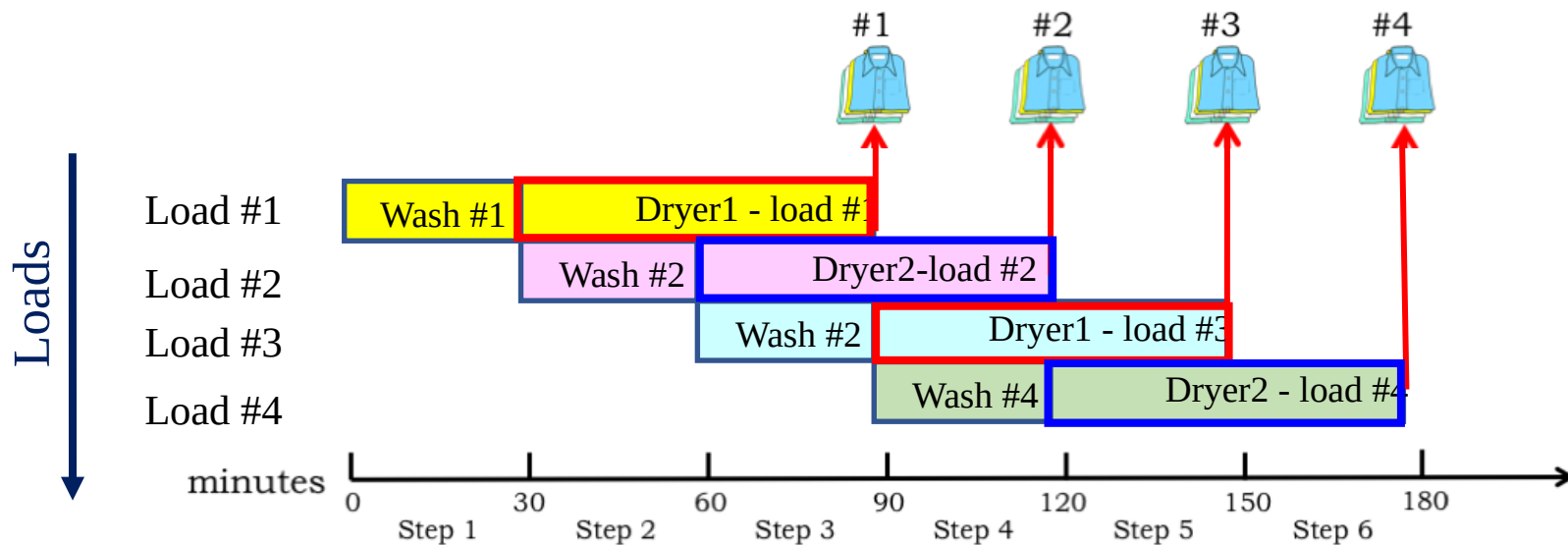
They work around the bottleneck. First, they find a laundromat with two dryers for every washer. Then they use dryer #1 for odd-numbered wash loads and dryer #2 for even-numbered wash loads.



Throughput = 1/30 loads/min, Latency = 90 mins/load

How Do 6.004 Students Do Laundry?

They work around the bottleneck. First, they find a laundromat with two dryers for every washer. Then they use dryer #1 for odd-numbered wash loads and dryer #2 for even-numbered wash loads.



Throughput = $\frac{1}{30}$ loads/min, Latency = 90 mins/load

Back To Our Bottleneck...

Recall our earlier example...

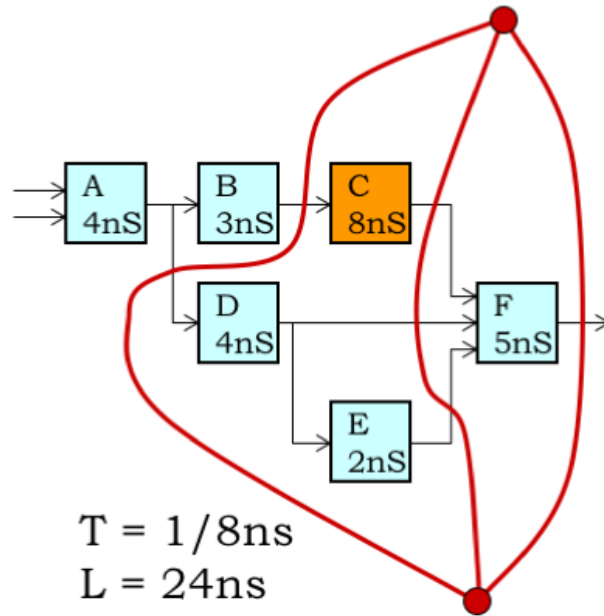
- C – the slowest component – limits clock period to 8 ns.
- HENCE throughput limited to $1/8\text{ns}$.

We could improve throughput by

- Finding a pipelined version of C;

OR ...

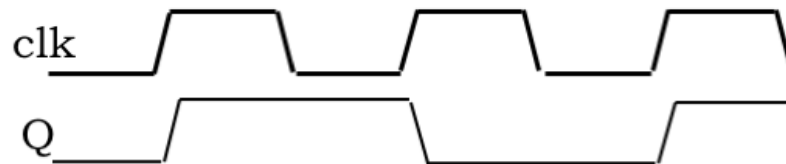
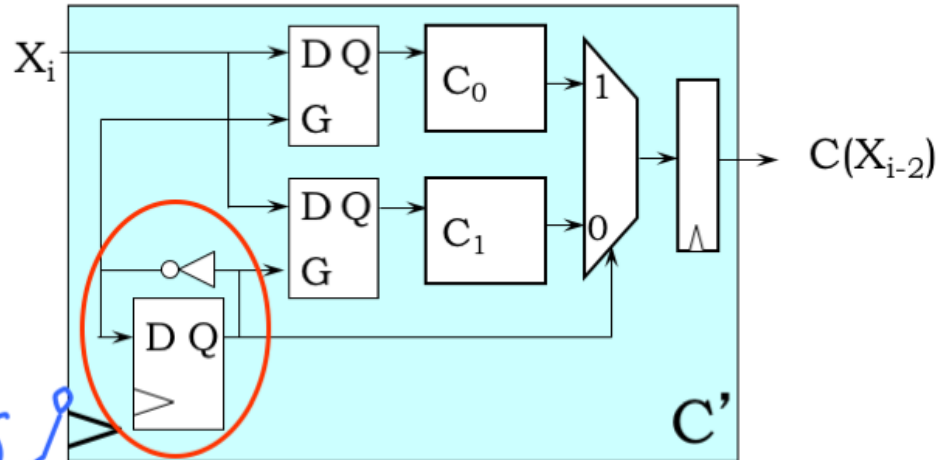
- *interleaving* multiple copies of C!



Circuit Interleaving

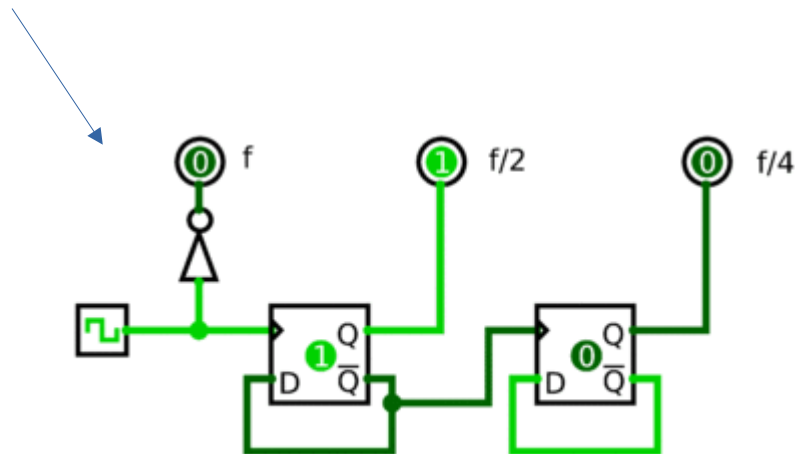
We can simulate a pipelined version of a slow component by replicating the critical element and alternate inputs between the various copies.

This is a simple 2-state FSM that alternates between 0 and 1 on each clock



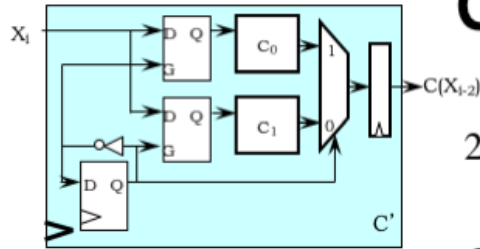
מחלק תדר

Click for animation

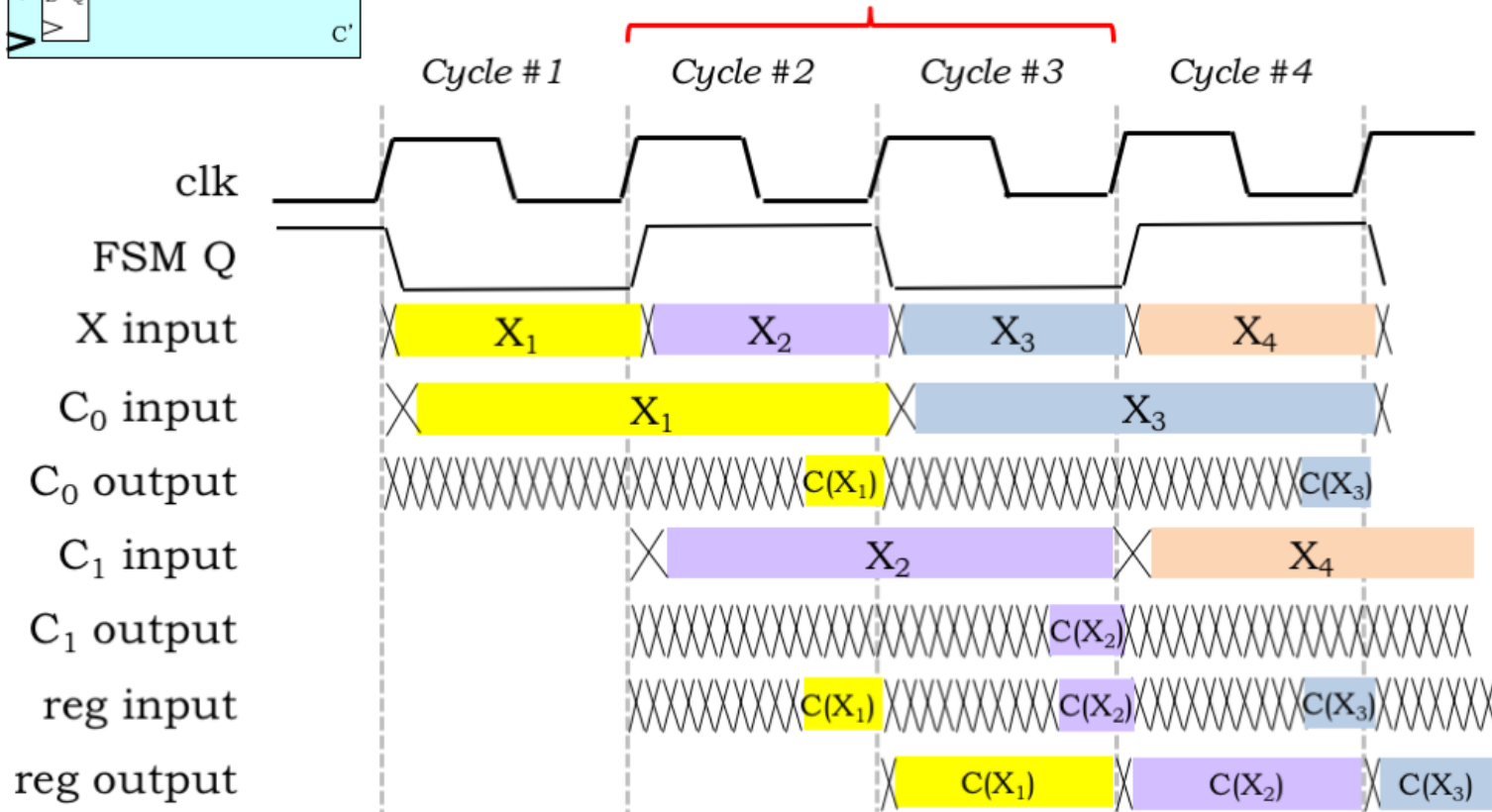


Credit: https://en.wikipedia.org/wiki/Frequency_divider

Circuit Interleaving



$$2 \cdot t_{CLK} \geq (t_{PD,upstreamREG} + t_{PD,LATCH} + t_{PC,C} + t_{PD,MUX} + t_{SETUP,REG})$$



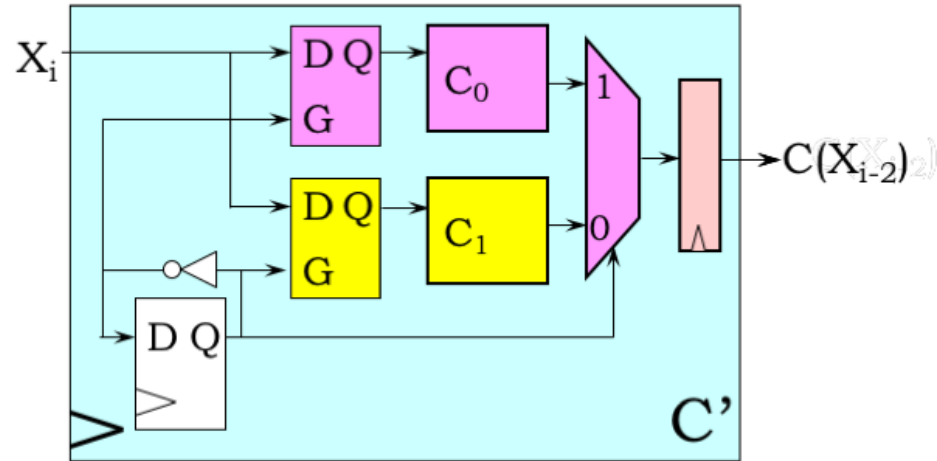
Circuit Interleaving

2-Clock Martinizing

“In by t_i , out by t_{i+2} ”

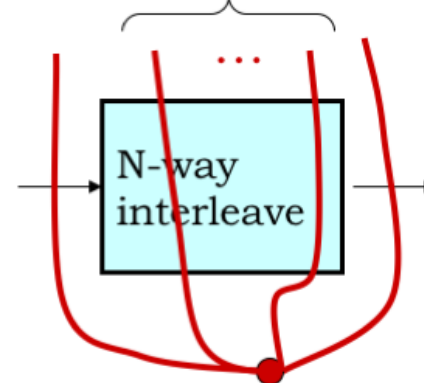
Throughput = 1/clock

Latency = 2 clocks



N registers

N-way interleaving
is equivalent to
N pipeline Stages...

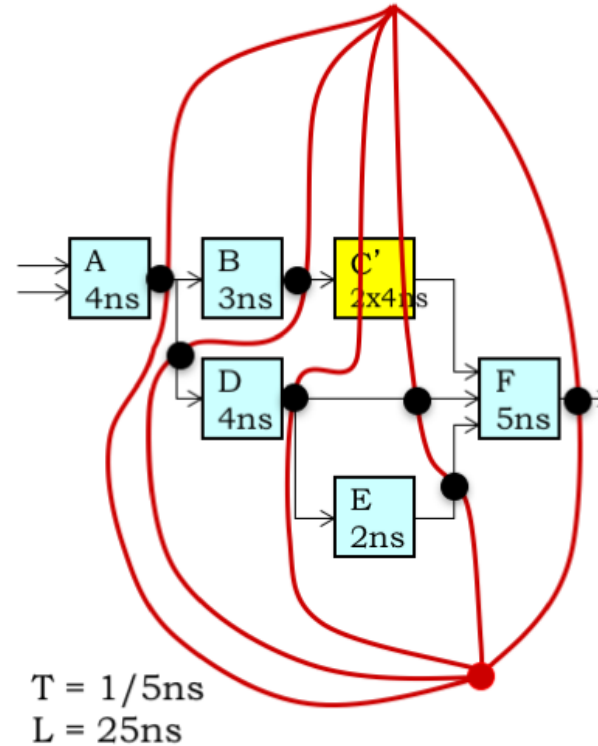


Combine Techniques

We can combine interleaving and pipelining. Here, C' interleaves two C elements and has an effective tCLK of 4 ns and a latency of 8 ns.

Since C' behaves as a 2-stage pipeline, two of our pipelining contours must pass through the C' component.

By combining interleaving with pipelining we move the bottleneck from the C element to the F element.



= כמות העיגולים כפול הזמן הכי ארוך
חמישה עיגולים (רגיסטרים) בכל נתיב כפול 5 נ"ש

Summary

Latency (L) = time it takes for given input to arrive at output

Throughput (T) = rate at which new outputs appear

For combinational circuits: $L = t_{PD}$ of circuit, $T = 1/L$ T=Throughput

For K-pipelines ($K > 0$):

- always have register on output(s)
- K registers on every path from input to output
- Inputs available shortly after clock i, outputs available shortly after clock (i+K)
- $t_{CLK} = t_{PD,REG} + t_{PD}$ of slowest pipeline stage + t_{SETUP}
- $T = 1/t_{CLK}$
 - more throughput $\Rightarrow$ split slowest pipeline stage(s)
 - use replication/interleaving if no further splits possible
- $L = K * t_{CLK} = K / T$
 - pipelined latency $\geq$ combinational latency

עד כאן מצגת זו

לקריאה נוספת אודות FSM ראו סעיף 3.4 ב-H&H