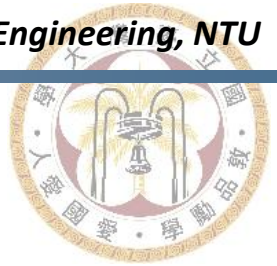
Advanced Computer-Aided VLSI System Design

Homework 3: Keccak Hash Function

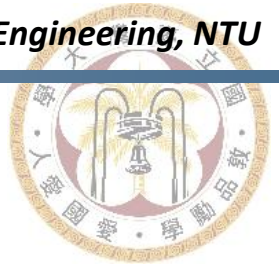
Graduate Institute of Electronics Engineering, National Taiwan University

NTU GIEE



Goals

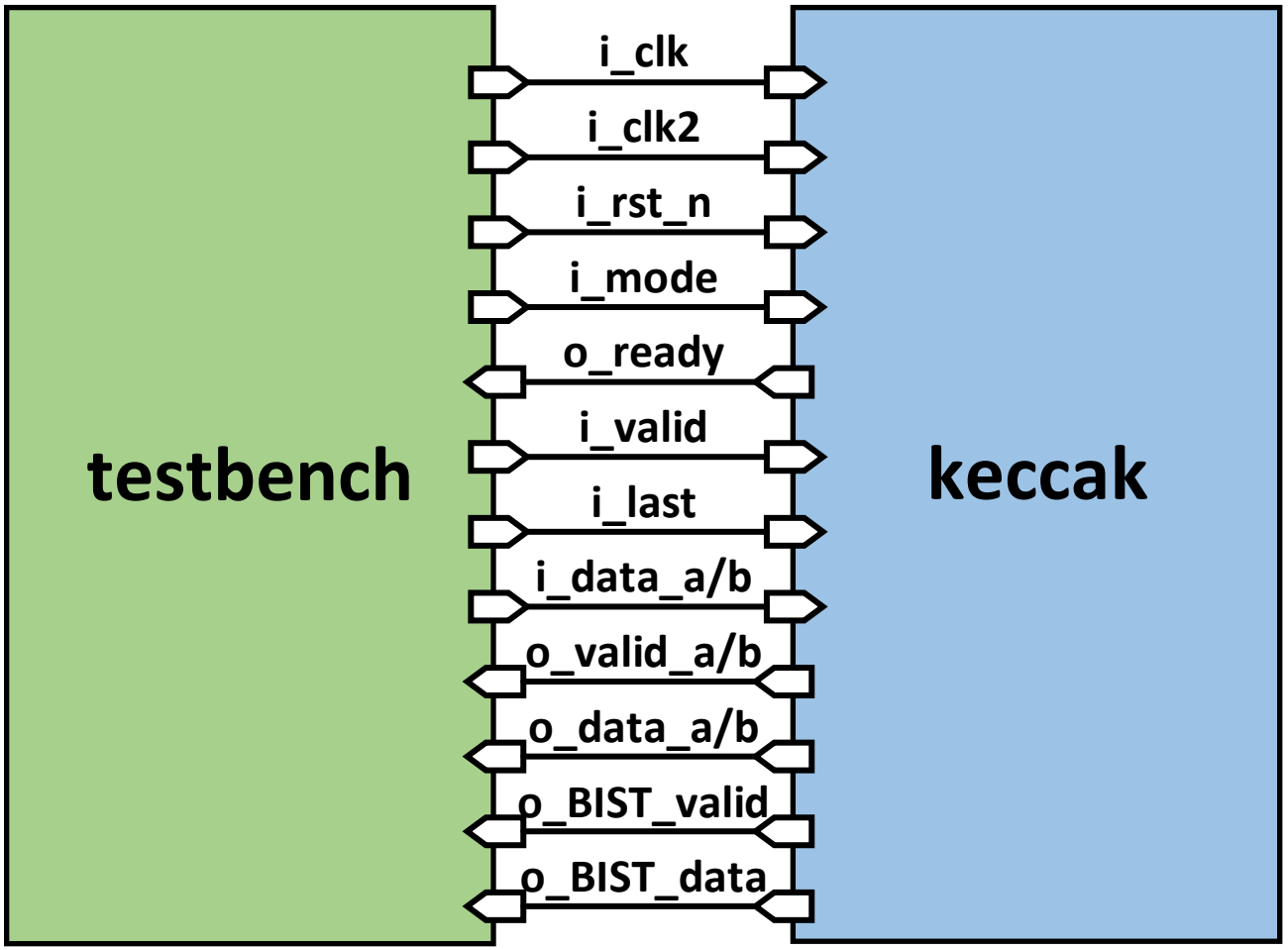
- In this homework, you will learn
  - How to insert scan chain
  - How to design and implement a small scale BIST
  - How to combine the learned techniques (optional)

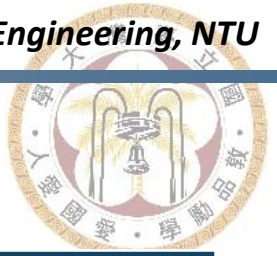


Introduction

- Keccak hash function is the precursor to the well-known SHA-3 hash function.
- Cryptographic hash functions like this can generate deterministic, collision-resistant and irreversible output from arbitrary input data.
- They are wildly used in secure applications such as encryption, authentication and digital signature.

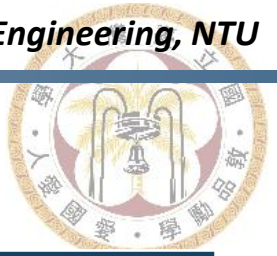
Block Diagram





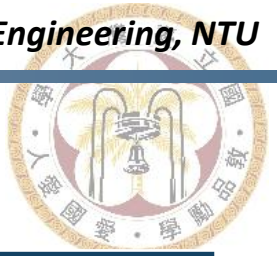
Input/Output

Signal Name	I/O	Width	Simple Description
i_clk	I	1	Clock for IO All inputs and outputs are synchronized with the positive edge with **0.5 ns** of input and output delay
i_clk2	I	1	Optional clock for computational core
i_rst_n	I	1	Active **low** asynchronous reset.
i_mode	I	2	Operation mode selection
o_ready	O	1	Pull **high** if ready for next input



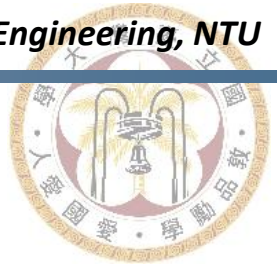
Input/Output

Signal Name	I/O	Width	Simple Description
i_valid	I	1	Pull **high** if input data is valid
i_last	I	1	Pull **high** if this is the last part of current input sequence
i_data_a	I	128	Input data A if i_valid is high; otherwise, all X's
i_data_b	I	128	Input data B if i_valid is high; otherwise, all X's
o_valid_a	O	1	Pull **high** if output data A is valid
o_data_a	O	64	Output data A



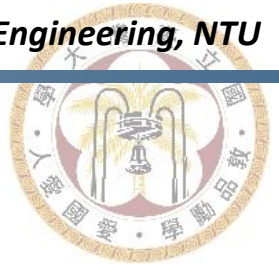
Input/Output

Signal Name	I/O	Width	Simple Description
o_valid_b	O	1	Pull **high** if output data B is valid
o_data_b	O	64	Output data B
o_BIST_valid	O	1	Pull **high** if BIST output is valid
o_BIST_data	O	64	Output of BIST mode



Specification

- Active **low** asynchronous reset is used only once
- There should be **at least one scan chain** in your design
- The total simulation time for each public case **must not exceed 6000 ns**
- The synthesis result should NOT include any latch
- There should be NO timing violations in the gate-level simulation after first positive edge of i_clk after reset



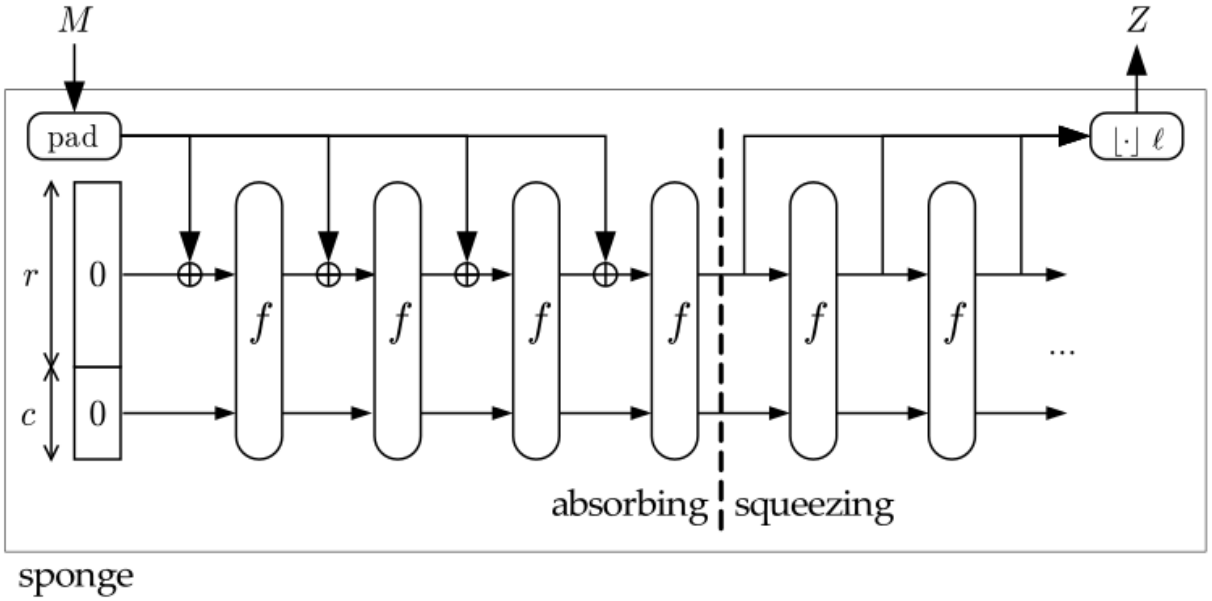
Operation Mode

- There are three operation modes

Mode	Code	Description
Single	2'b00	Compute Keccak hash for only input A
Double	2'b01	Compute Keccak hash for both input A and input B separately
BIST	2'b11	Perform built-in self-test

Keccak Hash Function

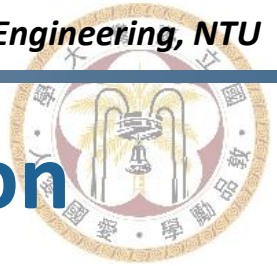
- Keccak is a family of hash functions that is based on sponge construction
- Sponge construction = absorbing phase + squeezing phase
- Variable-length input and arbitrary output length based on a fixed-length permutation function *f*



Keccak Hash Function

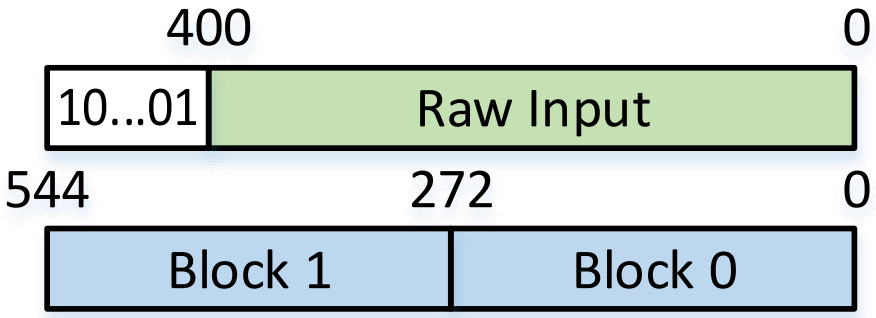
- In this homework, you are asked to implement a variant of the Keccak hash function with the following parameters

Parameter	Value	Parameter	Value
l	4	$w = 2^l$	16
$b = 25 \times w$	400	c	128
$r = b - c$	272	Rounds $= 12 + 2l$	20
Input length	Nr	Output length	64



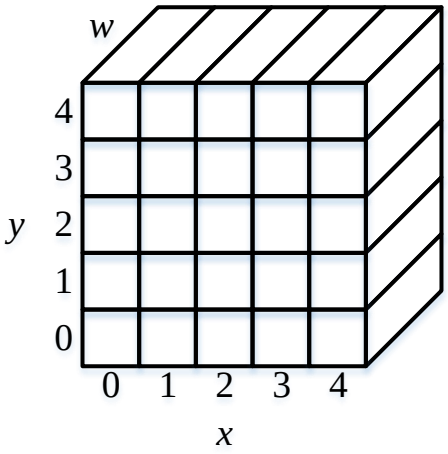
Process flow of Keccak Hash Function

1. Padding: Pad the raw input to an input sequence whose length is aligned with *r*
   - Padding rule: append a single bit 1, followed by zero or more 0's, and end with a single bit 1
   - Done by TA in software
   - Note: byte to int conversion adopt little-endian

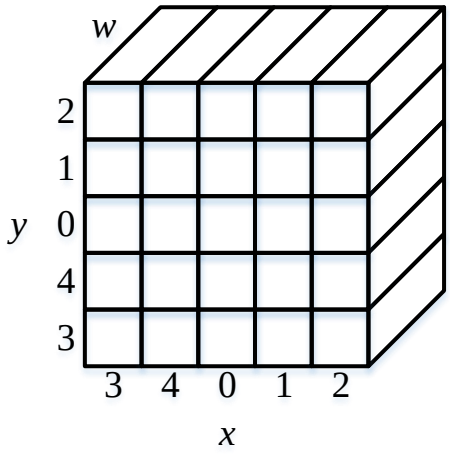


Process flow of Keccak Hash Function

2. Initialization: Initialize the state *S* as an empty array
   - Shape: 5 × 5 × *w*



Suitable for implementation

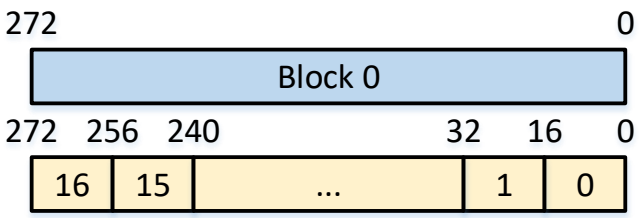


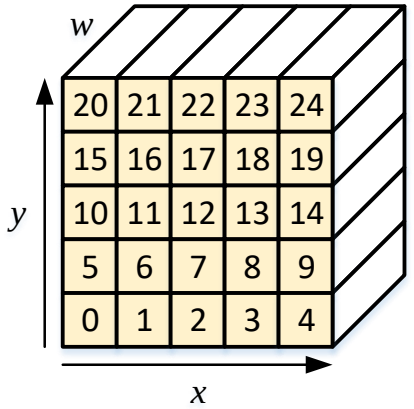
Suitable for understanding

Process flow of Keccak Hash Function

3. Absorbing: Consume input data

```
for B in Blocks
  for y in 0 … 4
    for x in 0 … 4
      if [(x+5*y) < r/w]
        S[x,y] = S[x,y] xor B[(x+5*y)*w +: w]
  S = Keccak-f(S)
```

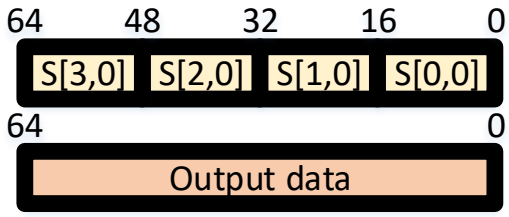


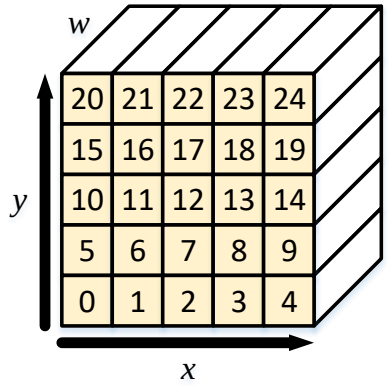


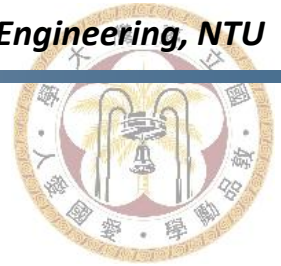
Process flow of Keccak Hash Function

4. Squeezing: Output hash result

```
Z = ""
while [output_len > 0]
  for y in 0 … 4
    for x in 0 … 4
      if [(x+5*y) < r/w] and [output_len > 0]
          Z = Z || S[x,y]
  if [output_len > 0]
    S = Keccak-f(S)
```

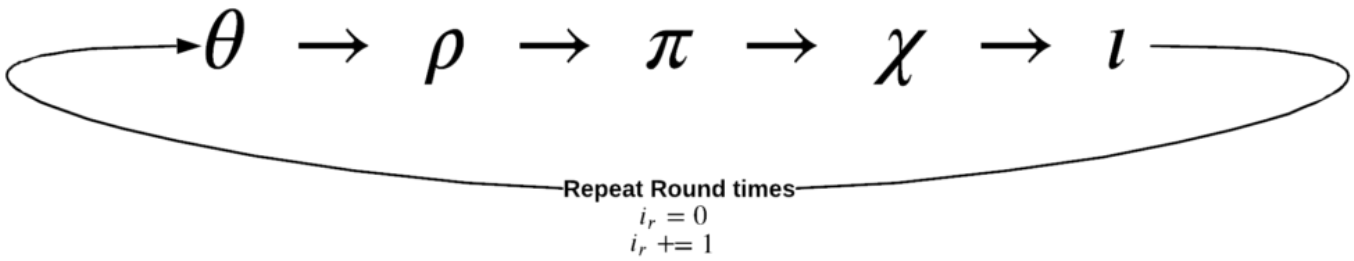






Keccak-*f*

- Keccak-*f* is the underlying permutation function for Keccak
- There are 20 rounds in the chosen variant
- Each round consists of 5 different steps

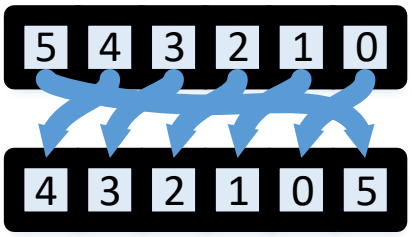


$$Rounds = 12 + 2\ell$$

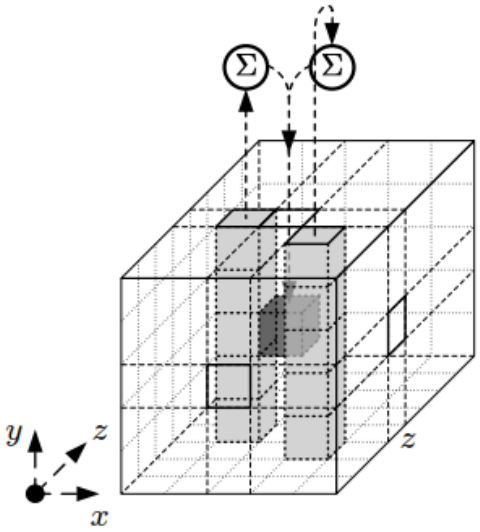
Process flow of Keccak-*f*

1. θ step: Diffusion across the state array

```
for x in 0 … 4
  C[x] = S[x,0] xor S[x,1] xor … S[x,4]
for x in 0 … 4
  D[x] = C[(x-1)%5] xor ROT(C[(x+1)%5], 1)
  for y in 0 … 4
    S[x,y] = S[x,y] xor D[x]
```



ROT(x, n=1)

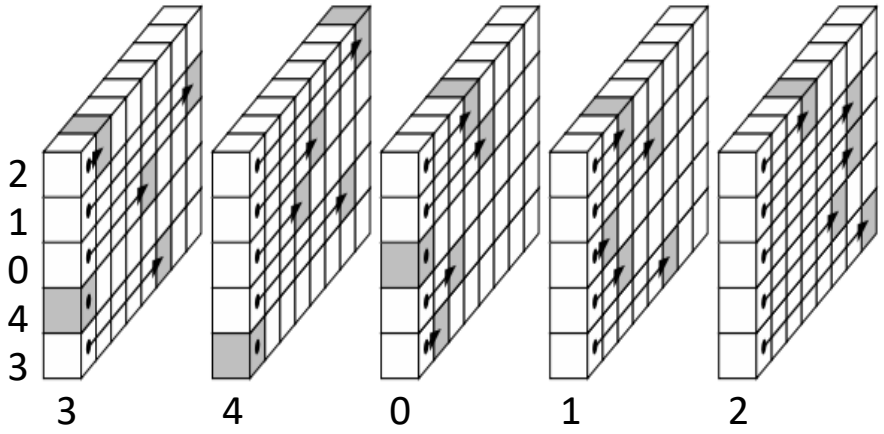


Process flow of Keccak-*f*

2. ρ step: Bitwise rotation

```
for y in 0 … 4
  for x in 0 … 4
    S[x,y] = ROT(S[x,y], OFFSET[x,y])
```

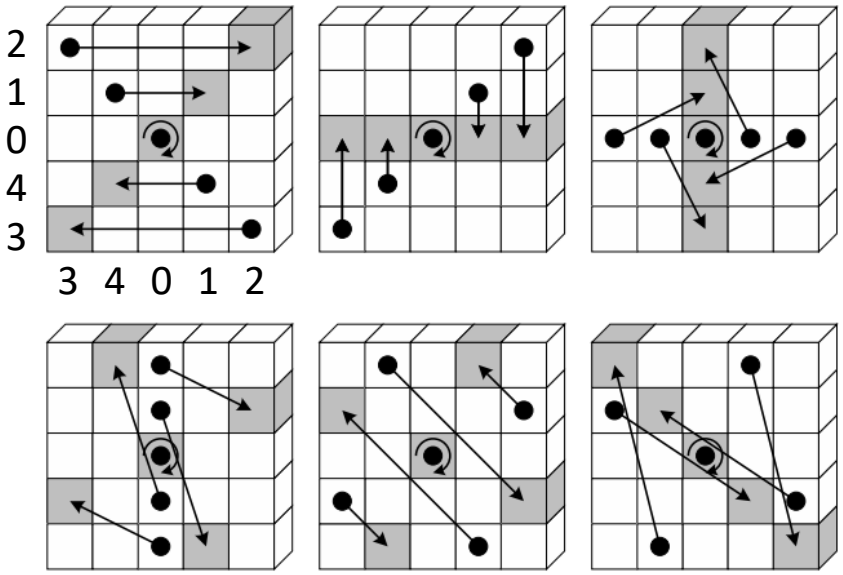
OFFSET	x = 0	x = 1	x = 2	x = 3	x = 4
y = 0	0	1	14	12	11
y = 1	4	12	6	7	4
y = 2	3	10	11	9	7
y = 3	9	13	15	5	8
y = 4	2	2	13	8	14

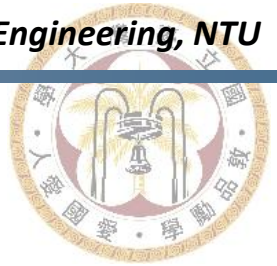


Process flow of Keccak-*f*

3. π step: Dispersion for long-term diffusion

```
for y in 0 … 4
  for x in 0 … 4
    S'[y,(2x+3)%5] = S[x,y]
S = S'
```

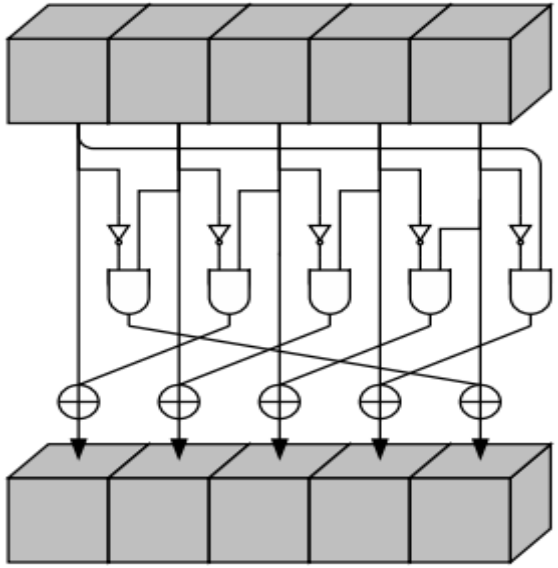


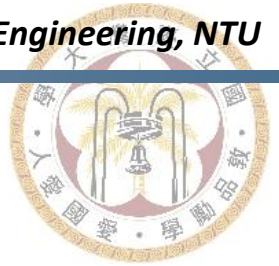


Process flow of Keccak-*f*

4. χ step: Nonlinear transformation

```
for y in 0 … 4
  for x in 0 … 4
    S’[y,x] = S[x,y] xor [(not S[(x+1)%5,y]) and S[(x+2)%5,y]]
S = S'
```



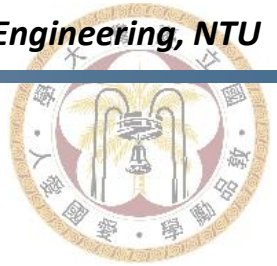


Process flow of Keccak-*f*

5. *ι* step: Disrupt symmetry

```
S[0,0] = S[0,0] xor RC[round_idx]
```

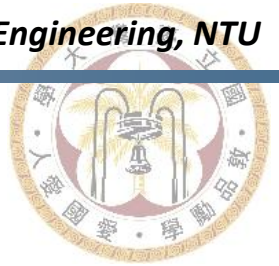
Round	RC	Round	RC	Round	RC	Round	RC
0	0x0001	5	0x0001	10	0x8009	15	0x8003
1	0x8082	6	0x8081	11	0x000A	16	0x8002
2	0x808A	7	0x8009	12	0x808B	17	0x0080
3	0x8000	8	0x008A	13	0x008B	18	0x800A
4	0x808B	9	0x0088	14	0x8089	19	0x000A



Built-In Self-Test

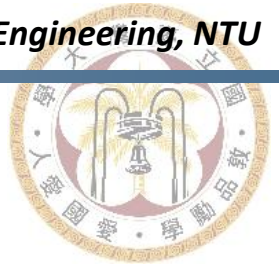
- BIST can check the correctness of the circuit without external test patterns
- You are asked to implement BIST for all the XOR operations in your design
- All XOR operation must be achieved by instantiating the *xor2* module provided by TA

```
1  module xor2 #(
2      parameter ID = 0
3  ) (
4      output [63:0] o_ID,
5      input  [ 1:0] a,
6      input  [ 1:0] b,
7      output [ 1:0] z
8  );
```



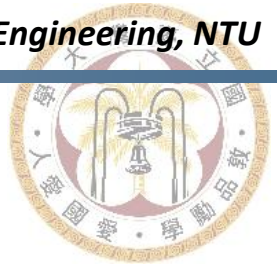
Built-In Self-Test

- Each *xor2* instance should be assigned a unique ID through module parameter, ranging from 1 to 2^{64}-1
- In BIST mode, the design should check the correctness of all the *xor2* instances, and output the ID of the faulty instance
- If all instances are fault-free, o_BIST_data should be assigned 0
- It is guaranteed that there will be at most one faulty instance
- It is guaranteed that the faulty instance will produce wrong output for at least one combination of inputs (total 2^4 possible combinations)



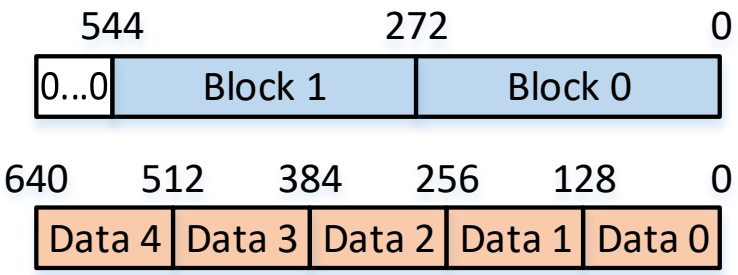
Scan Chain

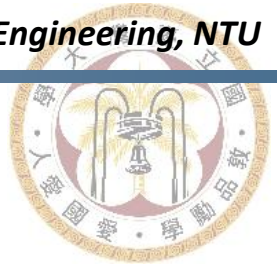
- You must insert **at least one scan chain** in your design
- All registers must be on the scan chain(s)
- **Use dedicate scan out pin**
- Check lab3 for details
- It is also encouraged to try ATPG on IC Lab server (optional)



Input / Output

- The operation mode (i_mode) will be stable throughout the entire simulation
- For mode 00 and 01,
  - There will be multiple input sequences in one simulation
  - Each input sequence will be partitioned into $\lceil \text{seq\_len}/128 \rceil$ input data of 128 bits
  - i_last indicates the last part of the current input sequence, and will only be pulled high if i_valid is pulled high

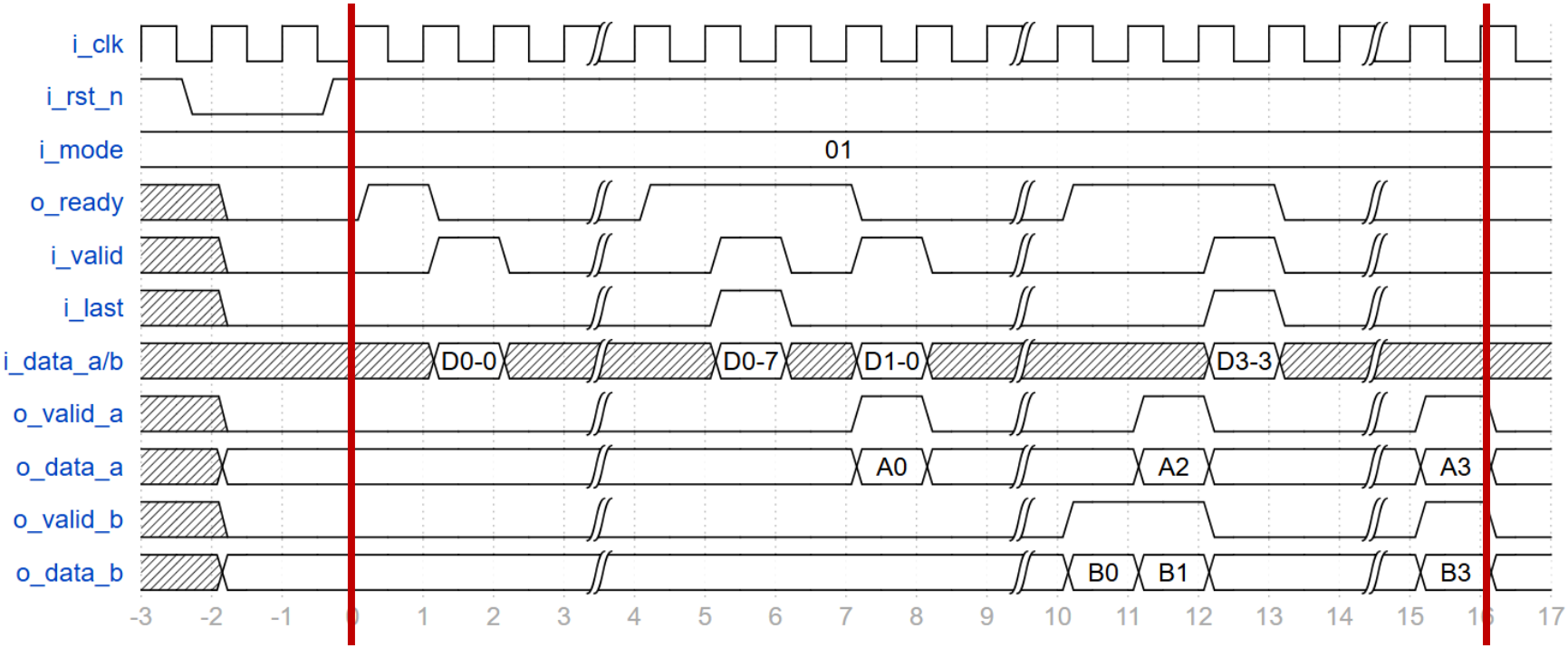




Input / Output

- For mode 00 and 01,
  - If i_valid is low and o_ready is high, i_valid will be randomly pulled high in the next cycle
  - If i_valid is high or o_ready is low, i_valid will be low in the next cycle
  - Output set A and output set B are independent
  - Output data must be in the same order of input data
  - o_valid_a/b can be pulled high independently at any positive edge of i_clk

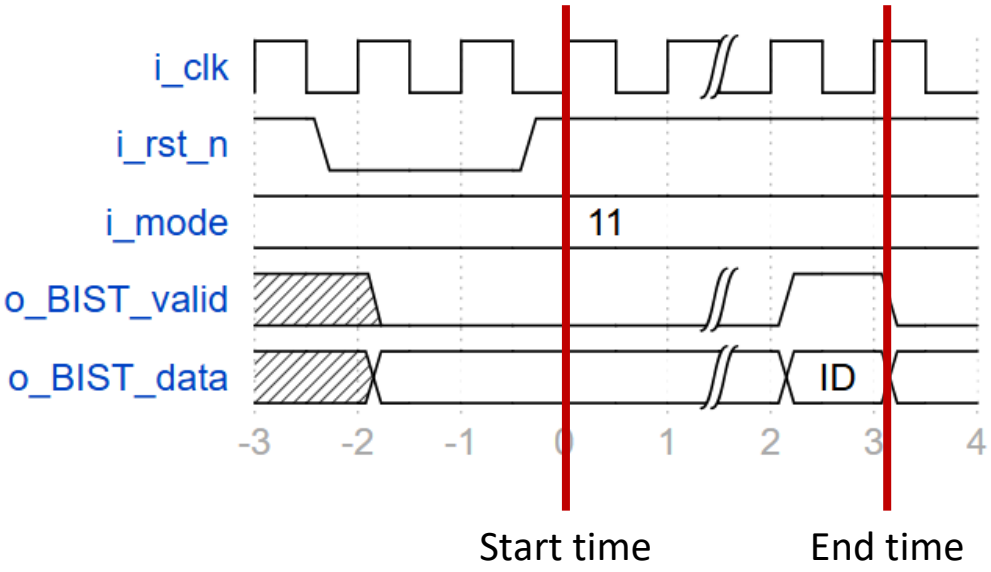
Timing Diagram

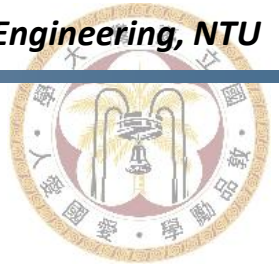


Start time

End time

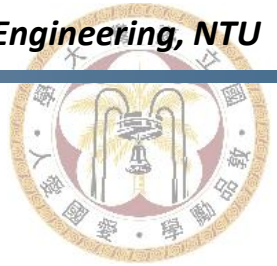
Timing Diagram





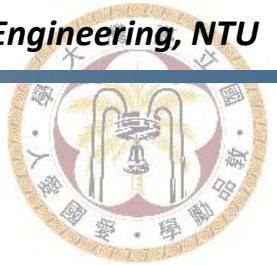
Hint

- You may adopt any techniques learned to gain better performance, including clock gating, power gating, multiple clock domain, etc.
- The scripts may not be complete. Remember to modify them.



keccak.v

```
1   module keccak #(
2       parameter IN_DATA_WIDTH  = 128,
3       parameter OUT_DATA_WIDTH = 64
4   ) (
5       input  i_clk,
6       input  i_clk2,
7       input  i_rst_n,
8
9       input  [               1:0] i_mode,
10      output                      o_ready,
11      input                       i_valid,
12      input                       i_last,
13      input  [ IN_DATA_WIDTH-1:0] i_data_a,
14      input  [ IN_DATA_WIDTH-1:0] i_data_b,
15      output                      o_valid_a,
16      output [OUT_DATA_WIDTH-1:0] o_data_a,
17      output                      o_valid_b,
18      output [OUT_DATA_WIDTH-1:0] o_data_b,
19      output                      o_BIST_valid,
20      output [OUT_DATA_WIDTH-1:0] o_BIST_data
21  );
```



xor2.v

```
1   module xor2 #(
2       parameter ID = 0
3   ) (
4       output [63:0] o_ID,
5       input  [ 1:0] a,
6       input  [ 1:0] b,
7       output [ 1:0] z
8   );
9
10      // `ifdef RTL
11      //     initial begin
12      //         $display("ID = %d", ID);
13      //     end
14      // `endif
15
16      assign o_ID = ID;
17      assign z = a ^ b;
18
19  endmodule
```

Report

- Total number of xor2 instances in your design
- Clock period for i_clk and i_clk2
- Area after scan chain insertion
- Gate-level (including scan chain) runtime and power for each public pattern

```
Student ID: r13943xxx
Num. of xor2 inst.: xxx
Clock period of i_clk : x.x ns
Clock period of i_clk2: x.x ns
Area: xxx.xxxxxx um^2
------------------------------
m00 time: xxx.xxx ns
m00 power: xxx.xxx mW
------------------------------
m01 time: xxx.xxx ns
m01 power: xxx.xxx mW
------------------------------
m11 time: xxx.xxx ns
m11 power: xxx.xxx mW
```

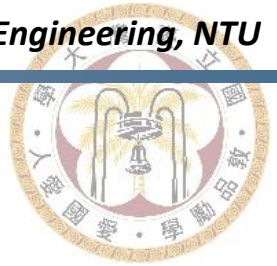
Submission

- Create a folder named studentID_hw3 and follow the hierarchy below

```
r13943xxx_hw3
    ├── 01_RTL
    │   ├── 01_run
    │   ├── rtl.f
    │   ├── keccak.upf (optional)
    │   ├── keccak.v
    │   ├── xor2.v
    │   └── xxx.v (other verilog files)
    ├── 02_SYN
    │   ├── syn.tcl
    │   ├── keccak_syn.area
    │   ├── keccak_syn.timing
    │   └── keccak_syn.scan_path
```

```
r13943xxx_hw3
    ├── 03_GATE
    │   ├── 03_run
    │   ├── gate.f
    │   ├── keccak_syn.upf (optional)
    │   ├── keccak_syn.v
    │   └── keccak_syn.sdf
    ├── 04_PWR
    │   ├── keccak_m00.power
    │   ├── keccak_m01.power
    │   └── keccak_m11.power
    └── report.txt
```

- Pack the folder **studentID_hw3** into a tar file named **studentID_hw3_v*k*.tar** (*k* is the number of version, *k* =1,2,…)
- Submit to NTU Cool and place the tared file at your home dir

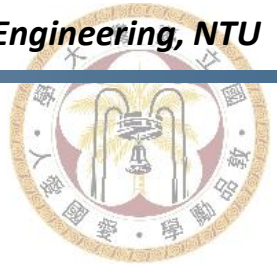


Grading Policy

- TA will use **01_run** and **03_run** to run your code at RTL and gate-level (including scan chain) simulation. Modify them to your needs.
- Hash function simulation (mode 00 and 01) (50%)
  - 0 point for the test case that takes more than **6000 ns**

	Score
RTL public	20%
Gate-level public	20%
Gate-level private	10%

- BIST simulation (mode 11) (10%)
  - Pass both RTL and gate-level simulation to get full points
  - Otherwise, 0 point



Grading Policy

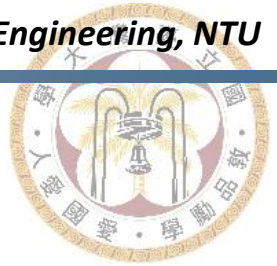
- Scan chain insertion (10%)
  - Report files generated by design compiler
- Performance ranking (30%)
  - You may get this score only if you get full points in all the above criteria

$$\text{Score} = A \times (T_{00} \times P_{00} + T_{01} \times P_{01} + T_{11} \times P_{11})$$

A: Total cell area after scan chain insertion (um^2)

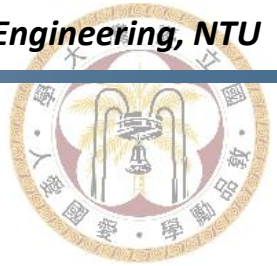
T_x: Simulation time of mode *x* reported by testbench (ns)

P_x: Average power of mode *x* reported by primetime (mW)



Grading Policy

- Do not memorize the answers directly in any way
- Lose **10 points** for any violation of design specifications
- Lose **5 points** for any wrong naming rule or format for submission
- **No delay submission is allowed**
- **No plagiarism**



References

[1] The Keccak Reference https://keccak.team/files/Keccak-reference-3.0.pdf

[2] In-depth Visual Breakdown of the SHA-3 Cryptographic Hashing Algorithm https://chemejon.io/sha-3-explained

[2] Synopsys ® Multivoltage Flow User Guide

[3] R. Ginosar, Metastability and Synchronizers: A Tutorial, 2011.

[4] Clifford E. Cummings, Clock Domain Crossing (CDC) Design & Verification Techniques Using SystemVerilog, 2008.