Advanced Computer-Aided VLSI System Design
Homework 3: Keccak Hash Function

TA: 徐以帆 r13943005@ntu.edu.tw

Due Tuesday, Apr. 29, 13:59

TA: 蔡岳峰 f12943014@ntu.edu.tw

Data Preparation

1. Decompress 1132_hw3.tar with following command

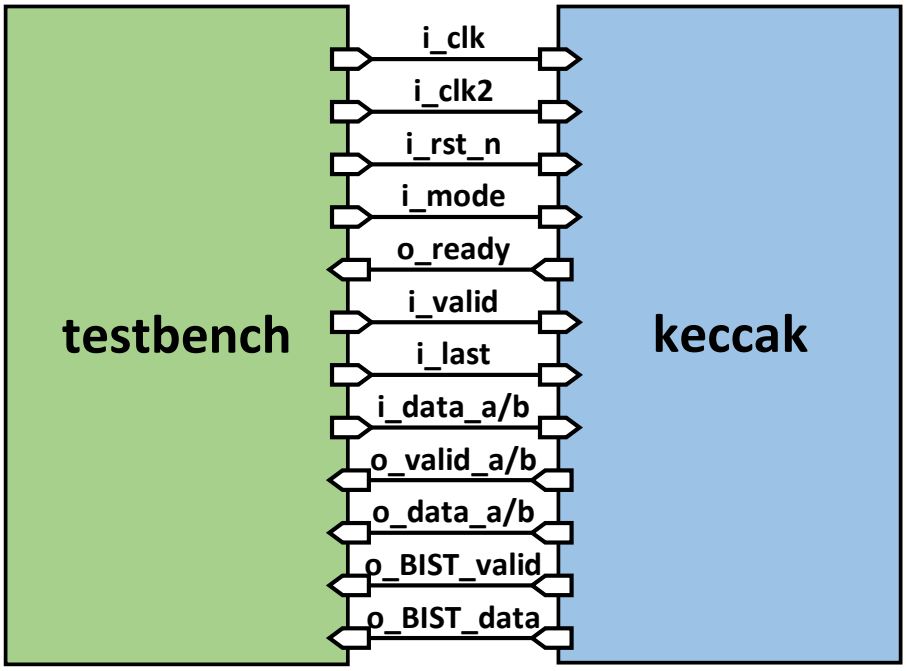
```
tar -xvf 1132_hw3.tar
```

Folder	File	Description
00_TESTBED	testbench.v	Testbench for the design
00_TESTBED/pattern	pattern*x*.dat	Input pattern
00_TESTBED/pattern	golden*x*.dat	Output golden pattern
00_TESTBED/pattern	keccak.py	Python implementation
01_RTL	01_run	RTL simulation bash file
01_RTL	rtl.f	File list for RTL simulation
01_RTL	keccak.v	Your RTL design
01_RTL	xor2.v	Template for xor2 module
02_SYN	02_run	Synthesis bash file
02_SYN	filelist.v	File list for synthesis
02_SYN	syn.tcl	TCL script for synthesis
02_SYN	keccak.sdc	Constraint for synthesis
03_GATE	03_run	Gate level simulation bash file
03_GATE	gate.f	File list for gate level simulation
04_PWR	04_run	Power measurement bash file
04_PWR	pt.tcl	PrimeTime script for power measurement

Introduction

In this homework, you are required to design an accelerator for **Keccak hash function**, the precursor to the well-known SHA-3 hash function. Cryptographic hash functions process input data of arbitrary length and generate a fixed-length, deterministic, collision-resistant and irreversible output. These properties make them ideal for secure applications such as encryption, authentication, and digital signature. Additionally, your design must integrate two DFT methods: BIST and scan chain.

Block Diagram



Specifications

1. Top module name: **keccak**
2. Input/output description:

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
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
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

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

Design Description

1. Operation Mode

   There are three operation modes:

   a. 2'b00: Compute Keccak hash for only input A.

   b. 2'b01: Compute Keccak hash for both input A and B separately.

   c. 2'b11: Perform built-in self-test.

2. Keccak Hash Function

   In this homework, you are asked to implement a variant of Keccak hash function – Keccak[272, 128], which adopts Keccak-*f*[400] as the underlying permutation function. The parameters of the chosen Keccak function are as:

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

   The process of computing Keccak hash function is:

   a. Padding: Pad the raw input to an input sequence whose length is aligned with *r*. This is done by the software and the input to your design are padded input sequences. To align input sequence to the input port width, the input sequence is then partitioned (see Fig. 1)

   b. Initialization: Initialize the state S as an empty $5 \times 5 \times w$ array.

   c. Absorbing: For each block of the input sequence, compute

   $$S[x,y] = S[x,y] \oplus \text{Block}_i[(x + 5y) \times w + : w]$$

   for each (x, y) such that $x + 5y < r/w$ (see Fig. 2) and perform one pass of permutation

   $$\text{S} = \text{Keccak} - f[400](S)$$

   d. Squeezing: For 64 bit fixed-length Keccak hash function, the output is simplified to be the lowest 64 bits of S. (see Fig. 3)

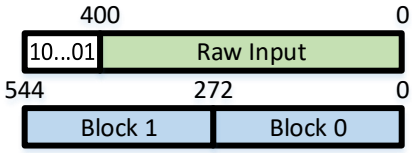


Fig. 1 Input sequence and padding

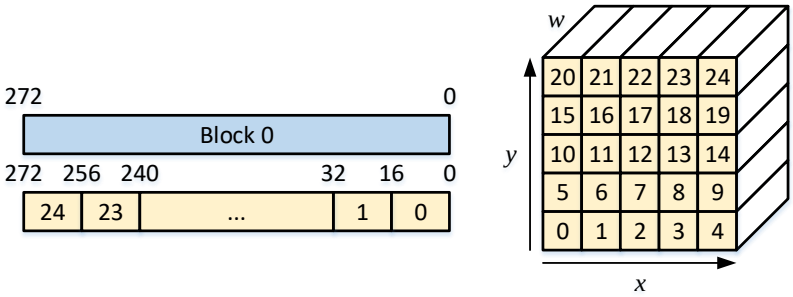


Fig. 2 Mapping between input sequence and state array

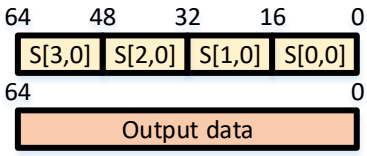


Fig. 3 Output data format

Keccak-*f*[400] consists of 20 identical rounds of computation, except that the round constant (RC) is different for each round. The process of each round is:

a. θ step: Diffusion across the state array. First, diffuse elements within each column:

$$C[x] = S[x,0] \oplus S[x,1] \oplus S[x,2] \oplus S[x,3] \oplus S[x,4]$$

Then, diffuse elements from nearby columns:

$$D[x] = C[(x-1)\ mod\ 5] \oplus \mathrm{ROT}(C[(x+1)\ mod\ 5], 1)$$

where $\mathrm{ROT}(x,n)$ means cyclic shift left x by n bits. Finally, XOR D with the original state array:

$$\mathrm{S}'[x,y] = S[x,y] \oplus D[x]$$

b. ρ step: Bitwise rotation. Rotate each entry of the state array by a predefined amount:

$$\mathrm{S}'[x,y] = \mathrm{ROT}(S[x,y], \mathrm{OFFSET}[x,y])$$

where the amount of OFFSET for Keccak-*f*[400] is given by the following table.

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

c. π step: Dispersion for long-term diffusion. Permute the state array as:

$$S'[y, (2x + 3y)\ mod\ 5] = S[x, y]$$

d. χ step: Nonlinear transformation. Compute

$$S'[x, y] = S[x, y] \oplus \left[(\sim S[(x + 1)\ mod\ 5, y]) \wedge S[(x + 2)\ mod\ 5, y]\right]$$

e. ι step: Disrupt symmetry. XOR $S[0,0]$ with the round constant. The round constant for each round is given by the following table.

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

3. BIST

In this homework, you are asked to integrate BIST for XOR operations. All XOR operations must be done through instantiating the given xor2 module. Each instance of the xor2 module must be assigned a unique ID through module parameter (see xor2.v for example). The ID should be in the range from 1 to 2^{64}-1. In mode 11, you should check the correctness of each xor2 instance. When the BIST is completed, pull o_BIST_valid high and assign the ID of the faulty instance to o_BIST_data. If all the instances are fault-free, assign all 0's to o_BIST_data. Assum there will only be at most one faulty instance, and the faulty instance will produce wrong output for at least one combination of input (total 2^4 possible combinations). To check the functionality of the BIST, TA will replace the xor2.v file to inject at most one faulty instance. You do NOT have to make any changes to your synthesis script for TA to test the circuit. **Do NOT deliberately replace XOR operations with other combinations of operations or you might get zero points.**

4. Scan Chain

You must insert **at least one scan chain** in your design, and all registers must be on the scan chain(s). **Use dedicate scan out pin**.

5. Input / Output

The operation mode (i_mode) will be stable throughout the entire simulation.

For mode 00 and 01,

a. There will be multiple input sequences in one simulation.

b. Each input sequence will be partitioned into $\lceil seq\_len/128 \rceil$ input data of 128 bits. (see Fig. 4)

c. i_last indicates the last part of the current input sequence, and will only be pulled high if i_valid is pulled high.

d. If i_valid is low and o_ready is high, i_valid will be randomly pulled high in the next cycle.

e. If i_valid is high or o_ready is low, i_valid will be low in the next cycle.

f. Output set A and output set B are independent.

g. Output data must be in the same order of input data, that is, the output of input sequence n must appear earlier than that of input sequence n+1.

h. o_valid_a/b can be pulled high independently at any positive edge of i_clk.

For mode 11, all inputs from testbench will be stable 0's, and o_BIST_valid can be pulled high at any positive edge of i_clk.

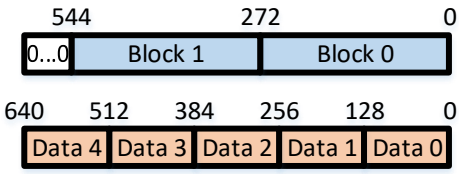
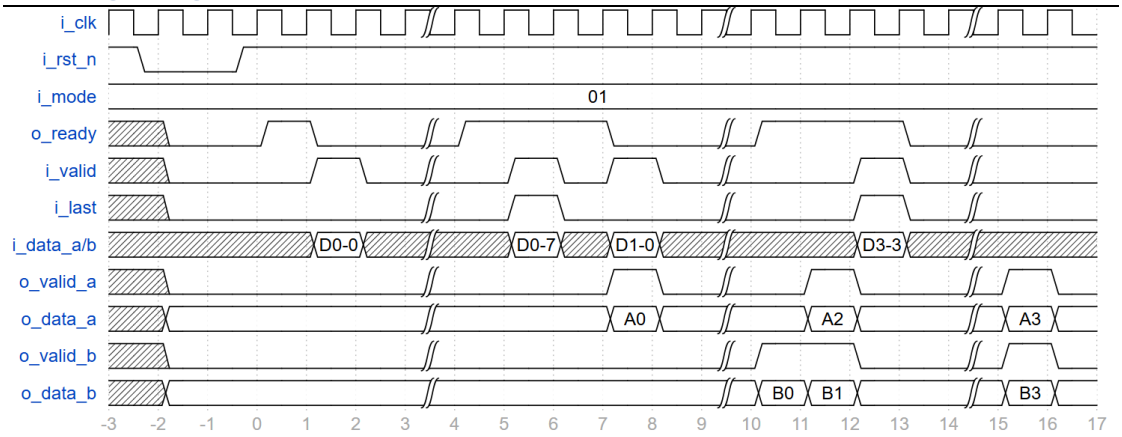
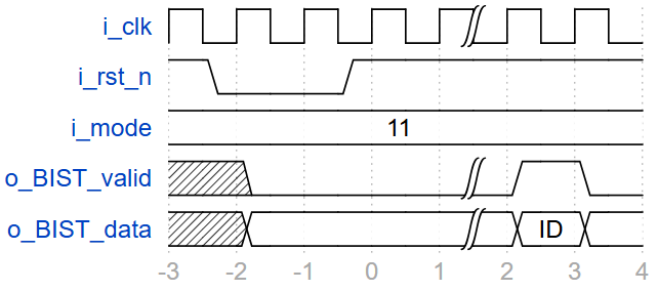


Fig. 4 Input data partition

Timing Diagram



1. At label 0, the simulation starts. At label 16, the simulation ends. The total simulation time is computed as $\text{time}(L = 16) - \text{time}(L = 0)$.
2. At lable 0-1, o_ready is high and i_valid is low. Therefore, i_valid may be pulled high in the next cycle. The first input data is also available.
3. At lebel 5-6, both i_valid and i_last are high, indicating D0-7 is the last part of data of input sequence 0. Also, both o_ready and i_valid are high. Therefore, i_valid will be low in the next cycle.
4. At label 5-6, o_valid_a is high, and the output data of the first input sequence is presented on o_data_a. Note that o_valid_a and o_valid_b do not have to be synchronized.



5. At label 2-3, o_BIST_valid is high, and the ID of the faulty instance is presented on o_BIST_data. If all instances are fault-free, the ID should be set to 0.

Hint

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

Submission

1. 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
```

Note: Use **lower case** for the letter in your student ID. (Ex. r13943000_hw3)

2. Content of the report.txt

   Record (1) total number of xor2 instances in your design, (2) clock period for i_clk and i_clk2, (3) area after scan chain insertion, and (4) gate-level (including scan chain) runtime and power for each public pattern

```
1   Student ID: r13943xxx
2   Num. of xor2 inst.: xxx
3   Clock period of i_clk : x.x ns
4   Clock period of i_clk2: x.x ns
5   Area: xxx.xxxxxx um^2
6   ------------------------------
7   m00 time: xxx.xxx ns
8   m00 power: xxx.xxx mW
9   ------------------------------
10  m01 time: xxx.xxx ns
11  m01 power: xxx.xxx mW
12  ------------------------------
13  m11 time: xxx.xxx ns
14  m11 power: xxx.xxx mW
```

3. Pack the folder **studentID_hw3** into a **tar file** named **studentID_hw3_v*k*.tar (*k* is the number of the version, *k* =1,2,…)**. TA will only check the last version.

```
tar -cvf studentID_hw3_vk.tar studentID_hw3
```

4. **Submit to NTU Cool and place the tared file at your home directory**.

Grading Policy

1. 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.

2. Hash function simulation (mode 00 and 01) (50%)

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

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

4. Scan chain insertion (10%)
   Report files generated by design compiler.

5. Performance ranking (30%)
   You may get this score only if you get full points in all the above criteria.

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

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)

Reference

[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.