Keccak RTL Design Specification

Introduction

The provided RTL design implements a Keccak permutation, which is a core component of the SHA-3 cryptographic hash function. The design consists of several interconnected modules that perform data padding, permutation, and round transformations. The design adheres to the Keccak-f[1600] permutation standard, which is part of the NIST SHA-3 specification.

Architecture

The design is structured into several modules, each responsible for a specific part of the Keccak permutation process:

1. **Top-Level Module (`keccak`)**: This module orchestrates the overall process, managing input data, padding, and invoking the permutation function.
2. **Padding Module (`padder`)**: Prepares the input data by adding necessary padding to conform to the Keccak block size requirements.
3. **Permutation Module (`f_permutation`)**: Executes the Keccak-f[1600] permutation, which involves multiple rounds of transformations.
4. **Round Constants Module (`rconst2in1`)**: Generates round constants used in the permutation rounds.
5. **Round Transformation Module (`round2in1`)**: Implements the core transformation steps of the Keccak permutation, including theta, rho, pi, chi, and iota steps.

Key Operations

- **Data Flow**: Input data is padded and then processed through the permutation function, which applies a series of transformations over multiple rounds.
- **State Machines**: The design uses state machines to manage the flow of data and control signals, ensuring correct sequencing of operations.
- **Clock Domains**: The design operates synchronously with a single clock domain.

Interface

`f_permutation` Module

Signal	Width	In/Out	Description
clk	1	In	Clock signal for synchronous operations
reset	1	In	Asynchronous reset signal
in	576	In	Input data for permutation
in_ready	1	In	Indicates input data is ready
ack	1	Out	Acknowledgment signal for input data usage
out	1600	Out	Output data after permutation
out_ready	1	Out	Indicates output data is ready

`keccak` Module

Signal	Width	In/Out	Description
clk	1	In	Clock signal for synchronous operations
reset	1	In	Asynchronous reset signal
in	64	In	Input data chunk
in_ready	1	In	Indicates input data chunk is ready
is_last	1	In	Indicates the last input chunk
byte_num	3	In	Number of valid bytes in the last chunk
buffer_full	1	Out	Indicates the buffer is full
out	512	Out	Output data after processing
out_ready	1	Out	Indicates output data is ready

`padder` Module

Signal	Width	In/Out	Description
clk	1	In	Clock signal for synchronous operations
reset	1	In	Asynchronous reset signal
in	64	In	Input data chunk
in_ready	1	In	Indicates input data chunk is ready
is_last	1	In	Indicates the last input chunk
byte_num	3	In	Number of valid bytes in the last chunk
buffer_full	1	Out	Indicates the buffer is full
out	576	Out	Padded output data
out_ready	1	Out	Indicates padded data is ready

`padder1` Module

Signal	Width	In/Out	Description
in	64	In	Input data chunk
byte_num	3	In	Number of valid bytes in the input chunk
out	64	Out	Padded output data chunk

`rconst2in1` Module

Signal	Width	In/Out	Description
i	12	In	Index for round constant generation
rc1	64	Out	First round constant
rc2	64	Out	Second round constant

`round2in1` Module

Signal	Width	In/Out	Description
in	1600	In	Input state for round transformation
round_const_1	64	In	First round constant
round_const_2	64	In	Second round constant
out	1600	Out	Output state after round transformation

Timing

- **Latency**: The design processes input data in multiple clock cycles, with the permutation function executing over several rounds.
- **Signal Behavior**: All operations are synchronous to the rising edge of the clock signal. The `out_ready` signal indicates when the output data is valid.

Usage

1. **Initialization**: Reset the design using the `reset` signal.
2. **Data Input**: Provide input data chunks through the `in` signal, asserting `in_ready` when data is available.
3. **Padding**: The `padder` module automatically handles data padding based on the `is_last` and `byte_num` signals.
4. **Permutation**: The `f_permutation` module processes the padded data, generating the permuted output.
5. **Output**: Monitor the `out_ready` signal to determine when the output data is valid and can be read from the `out` signal.

This specification provides a comprehensive overview of the Keccak RTL design, detailing its architecture, interface, timing, and usage to facilitate implementation and integration by hardware engineers.

Functional Description (Generated by funcgen)

Verilog Design Modules Functional Description

Module: f_permutation (File: f_permutation.v)

Purpose

The `f_permutation` module implements a permutation function as part of a cryptographic algorithm. It processes input data in rounds, controlled by ready and acknowledgment signals, to produce a transformed output.

Ports

- **clk**: Input, 1-bit - The clock signal for synchronous operations.
- **reset**: Input, 1-bit - Active-high reset signal to initialize the module state.
- **in**: Input, 576-bit - Data input to be permutated.
- **in_ready**: Input, 1-bit - Indicates that input data is valid and ready to be processed.
- **ack**: Output, 1-bit - Acknowledgment signal, high when the module accepts the input.
- **out**: Output, 1600-bit - The permutated output data.
- **out_ready**: Output, 1-bit - Indicates that the output data is valid.

Internal Signals

- **i**: Register, 11-bit - Counter indicating the current round of processing.
- **update**: Wire, 1-bit - Signal indicating whether to update the output in this cycle.
- **accept**: Wire, 1-bit - High when the module is ready to accept new input data.
- **calc**: Register, 1-bit - Flag indicating whether the module is in calculation mode.

Functionality

- **Sequential Logic**:
  - On the rising edge of `clk`, the `i` counter increments if `accept` is high, indicating new input processing.
  - The `calc` signal tracks whether the module is actively processing.
  - Updates `out` and `out_ready` based on the state of `i` and `accept`.
- **Combinational Logic**:
  - `accept` is derived from `in_ready` and the negation of `calc`.

    - `ack` is set directly by `accept`.
    - Generates the `update` signal based on `calc` and `accept`.
    - Determines the `round_in` input for submodules.

Instantiations

- **rconst2in1**: The `rconst_` instance generates round constants used for permutation.
- **round2in1**: The `round_` instance performs permutation transformation in each round.

Module: keccak (File: keccak.v)

Purpose

The `keccak` module implements the Keccak hashing algorithm, orchestrating the padding, permutation, and result generation stages.

Ports

- **clk**: Input, 1-bit - The clock signal.
- **reset**: Input, 1-bit - Active-high reset to clear states.
- **in**: Input, 64-bit - Input data segment for hashing.
- **in_ready**: Input, 1-bit - Signal indicating new input data availability.
- **is_last**: Input, 1-bit - Signals the last segment of input data.
- **byte_num**: Input, 3-bit - Indicates the number of bytes available in the data segment.
- **buffer_full**: Output, 1-bit - High when the internal buffer is full.
- **out**: Output, 512-bit - Hash output of the processed input data.
- **out_ready**: Output, 1-bit - Indicates when the output hash is valid.

Internal Signals

- **state**: Register, 1-bit - Tracks whether more input data is expected.
- **padder_out_ready**: Wire, 1-bit - Indicates when padding is complete.
- **f_ack**: Wire, 1-bit - Acknowledgment from `f_permutation` indicating input acceptance.
- **f_out**: Wire, 1600-bit - Output from `f_permutation`.
- **f_out_ready**: Wire, 1-bit - Indicates when the permutation output is ready.
- **i**: Register, 11-bit - Counter to synchronize hash output readiness.

Functionality

- **Sequential Logic**:
    - Updates internal state and output readiness counters based on reset and control signals.
    - Controls switching between input acceptance and result generation modes.
- **Combinational Logic**:
    - Generates reordered byte output from the permutation result.
    - Utilizes generate blocks for bit manipulation across data slices.

Instantiations

- **padder**: Handles padding of input data to meet algorithm requirements.
- **f_permutation**: Executes the core permutation logic on padded data.

Module: padder (File: padder.v)

Purpose

The `padder` module prepares input data by adding necessary padding before permutation, ensuring data meets cryptographic processing requirements.

Ports

- **clk**: Input, 1-bit - Clock for synchronization.
- **reset**: Input, 1-bit - Resets the internal state.
- **in**: Input, 64-bit - Data input segment.
- **in_ready**: Input, 1-bit - Data availability indicator.
- **is_last**: Input, 1-bit - Last data segment indicator.
- **byte_num**: Input, 3-bit - Number of valid bytes in `in`.
- **buffer_full**: Output, 1-bit - Indicates if the buffer is full.
- **out**: Output, 576-bit - Padded output data.
- **out_ready**: Output, 1-bit - Indicates when padded output is ready.
- **f_ack**: Input, 1-bit - Acknowledgment from `f_permutation`.

Internal Signals

- **state**: Register, 1-bit - Indicates if more data is expected.
- **done**: Register, 1-bit - Marks the completion of data padding.
- **i**: Register, 9-bit - Buffer length tracker.
- **v0**: Wire, 64-bit - Output from `padder1` handling individual byte padding.
- **v1**: Register, 64-bit - Shifted data to be added to the output buffer.

Functionality

- **Sequential Logic**:
  - Manages `out` buffer updates and controls state transitions based on input status and readiness.
  - Tracks buffer length and whether padding has been completed.
- **Combinational Logic**:
  - Manages acceptance of new input data if not final input and the buffer has space.
  - Uses the `padder1` module to construct the padded data block.

Instantiations

- **padder1**: A submodule which handles padding for a single data word based on the number of valid bytes.

Module: padder1 (File: padder1.v)

Purpose

The `padder1` module performs byte-based padding, extending partial byte values to a complete word as needed.

Ports

- **in**: Input, 64-bit - Data to be padded.
- **byte_num**: Input, 3-bit - Number of valid bytes in `in`.
- **out**: Output, 64-bit - Padded output word.

Internal Signals

- **out**: Register, 64-bit - Stores the padded output value.

Functionality

- **Combinational Logic**:

    - Uses a case statement to append padding bits to words with less than 8 bytes, creating fully qualified words.

Module: rconst2in1 (File: rconst2in1.v)

Purpose

`rconst2in1` generates two different round constants needed for the permutation process across multiple rounds.

Ports

- **i**: Input, 12-bit - Current round index for constant selection.
- **rc1**: Output, 64-bit - First round constant.
- **rc2**: Output, 64-bit - Second round constant.

Functionality

- **Combinational Logic**:
    - Produces round constants `rc1` and `rc2` based on specific control bits from `i`, applied through hardcoded bitwise operations.

Module: round2in1 (File: round2in1.v)

Purpose

The `round2in1` module performs two rounds of transformation according to the Keccak permutation function, using the provided round constants to process a 1600-bit input.

Ports

- **in**: Input, 1600-bit - Input to be transformed.
- **round_const_1**: Input, 64-bit - First round constant.
- **round_const_2**: Input, 64-bit - Second round constant.
- **out**: Output, 1600-bit - Resulting transformed output.

Functionality

- **Combinational Logic**:
    - Implements two complete rounds of Keccak's permutation, replicating the theta, rho, pi, chi, and iota steps using combinational logic blocks for both rounds.

Inter-Module Connections

The design is hierarchically structured, with `keccak` as the top-level module. It connects with: - **`padder`** for input preparation. - **`f_permutation`** for the main cryptographic permutation, instantiating `rconst2in1` and `round2in1` for round operations and constant generation. - **`padder1`** within `padder` for single-word padding. The control signals (`in_ready`, `ack`, `out_ready`) synchronize data flow across modules, ensuring correct processing and output production timelines within the cryptographic algorithm.