ROCKSAVAGE
TECHNOLOGY

Gpio
Product User Guide

rocksavagetech.chiselWare.Gpio

IPF certified to level: **0** of 5

Abdelrahman Abbas, Ahmed Elmenshawi, Nick Allison, Jimmy Bright

March 27, 2025

Contents

1 Errata and Known Issues

1.1 Errata

None.

1.2 Known Issues

None.

2 Port Descriptions

2.1 Gpio Interface

The ports for **Gpio** are shown below in Table 1. The width of several ports is controlled by the following input parameters:

dataWidth is the width of the gpioInput, gpioOutput, and gpioOutputEnable ports in bits

Port Name	Width	Direction	Description
gpioInput	*dataWidth*	Input	Data to be sent to the Gpio
gpioOutput	*dataWidth*	Output	Data to be received from the Gpio
gpioOutputEnable	*dataWidth*	Output	Enable data to be received from the Gpio
irqOutput	1	Output	Sent when interrupt is triggered on the Gpio

Table 2: Gpio Ports Descriptions

2.2 Apb3 Interface

The **Apb3 Interface** is a regular Apb3 Slave Interface. All signals supported are shown below in Table 2. See the *AMBA Apb Protocol Specifications* for a complete description of the signals. The width of several ports is controlled by the following input parameters:

- *dataWidth* is the width of PWDATA and PRDATA in bits
- *addrWidth* is the width of PADDR in bits

Port Name	Width	Direction	Description
PCLK	1	Input	Positive edge clock
PRESETN	1	Input	Active low reset
PSEL	1	Input	Indicates slave is selected and a data transfer is required
PENABLE	1	Input	Indicates second cycle of Apb transfer
PWRITE	1	Input	Indicates write access when HIGH and read access when LOW
PADDR	*addrWidth*	Input	Address bus
PWDATA	*dataWidth*	Input	Write data bus driven when PWRITE is HIGH
PRDATA	*dataWidth*	Output	Read data bus driven when PWRITE is LOW
PREADY	1	Output	Transfer ready
PSLVERR	1	Output	Transfer error

Table 4: Apb Ports Descriptions

3 Parameter Descriptions

The parameters for **Gpio** are shown below in Table 3.

Name	Type	Min	Max	Description
dataWidth	Int	1	≤ 32	The data width of Gpio ports, PWDATA, and PRDATA. Can be 8, 6, or 32 bits wide
addrWidth	Int	1	≤ 32	The Apb address bus width

Table 6: Parameter Descriptions

The Gpio is instantiated into a design as follows:

```
// Valid Gpio Instantiation Example
val myGpio = new Gpio(
  dataWidth = 32,
  addrWidth = 32 )
```

4 Operating Modes

4.1 Introduction

The Gpio core supports bidirectional IO pads, and each IO can be programmed to operate in push-pull or open drain mode, as defined by te MODE register.

Note: IO Pads are not implemented within the Gpio core - this is the responsiblity of the designer

4.2 Push-Pull Mode

In this mode, each bit of gpioOutput is driven from the internal OUTPUT register. The gpioOutputEnable bus is controlled via the DIRECTION register and specifies whether or not the pin will allow the IO Pad to see the corresponding OUTPUT data.

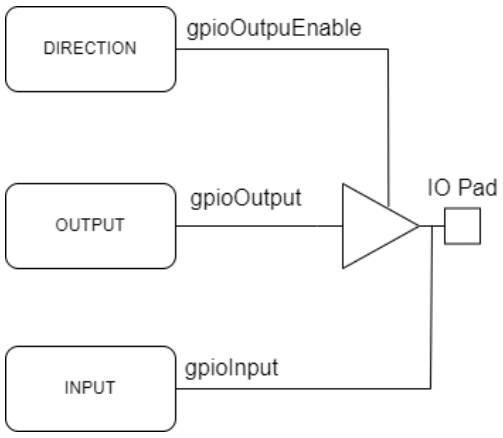

Figure 1: Push Pull Mode

4.3 Open Drain Mode

In this mode, each bit of gpioOutput is driven low when the corresponding OUTPUT register bit is low and corresponding DIRECTION register bit is high. This allows for multiple devices to pull the line low without conflicting high/low states on the output pin.

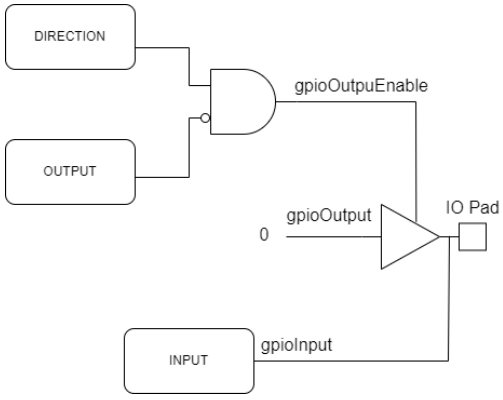

Figure 2: Open Drain Mode

5 Theory of Operations

5.1 Introduction

The **Gpio** module is configurable with parameters for setting data and address widths. It interfaces with the CPU via the Apb3 bus. The signals **gpioInput**, **gpioOutput** and **gpioOutputEnable** are designed to interface with external hardware, such as a tri-state buffer, to facilitate bi-directional communication on physical pins.

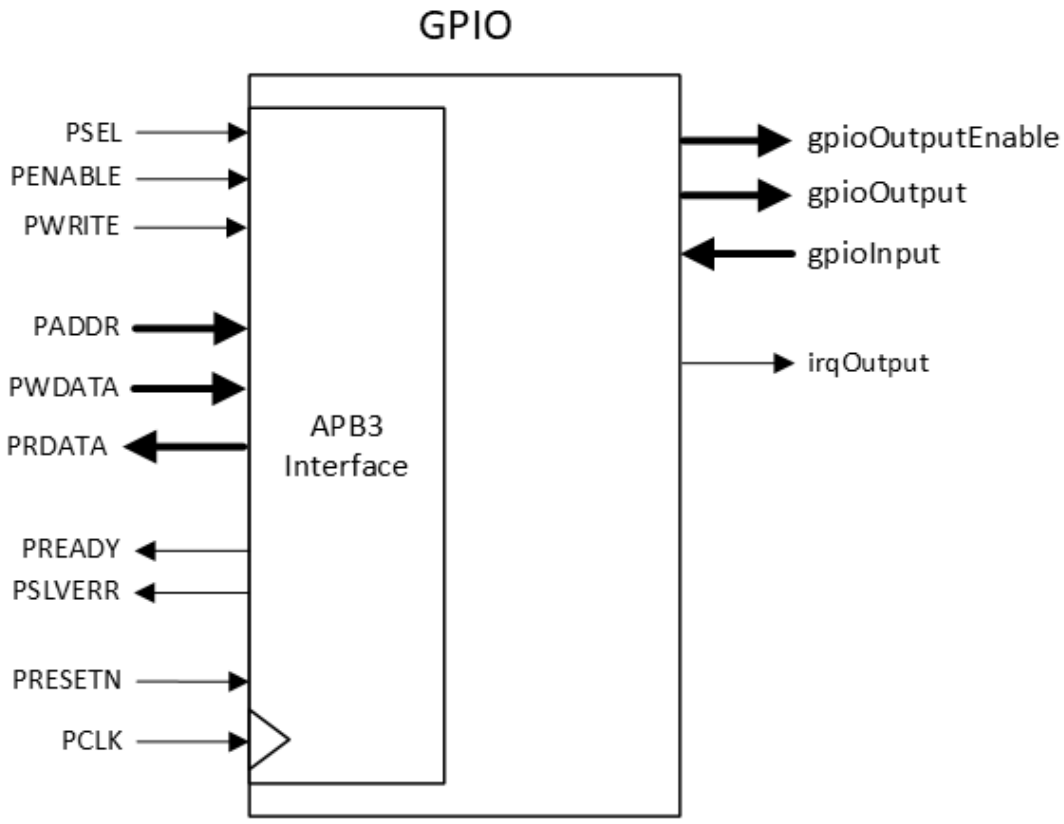

Figure 3: Gpio Block Diagram

5.2 Interface Timing

Gpio has a synchronous Apb3 interface, and a Gpio interface. The timing diagram shown below in Figure 4 represents an instantiation with the following parameters.

```
val myGpio = new Gpio(
  dataWidth = 16,
  addrWidth = 16 )
```

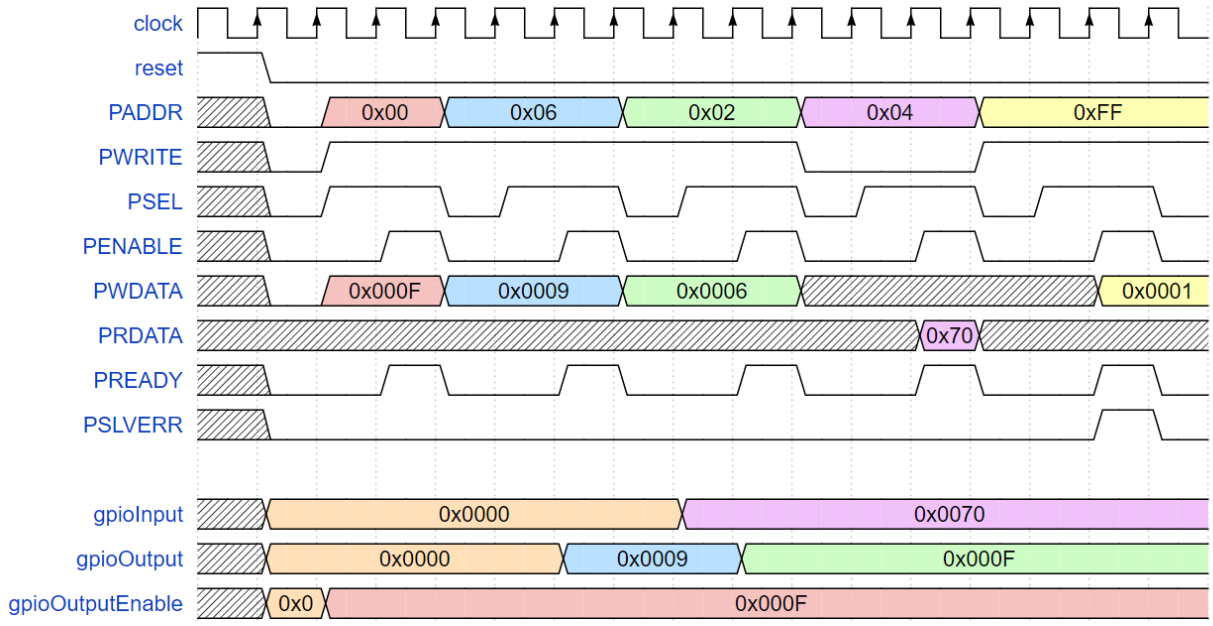

Figure 4: Timing Diagram

This shows the operation of the basic read/write register operations following the Apb protocol. Registers DIRECTION, MODE, and OUTPUT are written to, while register INPUT is read from.

The *gpioOutputEnable* port is driven to a value of 0x000F after 0x000F is written to the DIRECTION register at address 0x00. Next, the MODE register at address 0x06 is written a value of 0x0009. The *gpioOutput* port then has a value of 0x0009 because of the open-drain mode operation.

The OUTPUT register is then written a value of 0x0006 at address 0x02. *gpioOutput* takes on a value of 0x000F since the MSB and LSB are operating on open-drain mode, and the middle two bits are operating on push-pull mode.

The INPUT register is read from at address 0x04, which has a value of 0x0070 since *gpioInput* has a value fo 0x0070. On the final transaction, PSLVERR goes high because an invalid address is written to.

5.3 Register Interface

When programming registers, each register starts on a byte address, and the last bits it would take up in its final byte based on its size are unused. To find the size in bytes for any register, divide by the register size, and round up to the nearest whole number. For example, a 32-bit register would take up 4 bytes, and a 1-bit register would take up 1 byte.

Name	Size (Bits)	Description
DIRECTION	dataWidth	Controls whether the GPIO acts as an Input or Output
OUTPUT	dataWidth	Controls the output bus gpioOutput
INPUT	dataWidth	Controls the input bus gpioInput
MODE	dataWidth	Sets operating mode as push-pull or open drain mode
ATOMIC_OPERATION	4	Sets the atomic operation to be performed
ATOMIC_MASK	p.dataWidth	Masks the atomic operation on the OUTPUT register
ATOMIC_SET	1	Triggers the atomic operation on the OUTPUT register
VIRTUAL_PORT_MAP	sizeOfVirtualPorts	Maps virtual ports to physical ports
VIRTUAL_PORT_OUTPUT	numVirtualPorts	Sets output value of virtual ports
VIRTUAL_PORT_ENABLE	1	Enables the virtual ports
TRIGGER_TYPE	dataWidth	Configures gpio interrupts as edge or level triggered
TRIGGER_LO	dataWidth	Configures whether interrupt is triggered on level low or falling edge
TRIGGER_HI	dataWidth	Configures whether interrupt is triggered on level high or rising edge
TRIGGER_STATUS	dataWidth	Displays status of interrupt trigger condition on input bus
IRQ_ENABLE	dataWidth	Enables or disables interrupts

Table 8: Register Interface

5.3.1 Register Operation

DIRECTION: DIRECTION is a *dataWidth* bits wide active-high read/write register. This register controls the output enable bus *gpioOutputEnable*. Operation can be seen in Table

Table 9: Register Summary

DIRECTION[n]	Direction
0	Input
1	Output

OUTPUT: OUTPUT is a *dataWidth* bits wide read/write register. This register controls the output bus *gpioOutput*. Writing a '0' drives the pad low in both modes of operation. Writing a '1' drives the pad high in push-pull mode, or Hi-Z in open-drain mode.

INPUT: INPUT is a *dataWidth* bits wide read-only register. This register is written to from the input bus *gpioInput*. On the rising edge of the APB3 Bus Clock (PCLK), input data from *gpioInput* is synchronized using two flops and stored in the INPUT register. From there, it may be read via the APB3 Bus Interface through PRDATA.

MODE: MODE is a *dataWidth* bits wide read/write register. This register sets the operating mode for each bit of the *gpioOutput* and *gpioOutputEnable* busses as either push-pull or open drain mode. Operation can be seen in Table

MODE[n]	Operating Mode
0	Push-Pull
1	Open Drain

Table 10: MODE Register

ATOMIC_OPERATION: ATOMIC_OPERATION is a 4 bits wide read/write register. This register sets the atomic operation to be performed on the of gpio registers. The operation is performed on the *OUTPUT* register and is performed atomically.

ATOMIC_MASK: ATOMIC_MASK is a *dataWidth* bits wide read/write register. This register is used to mask the atomic operation on the *OUTPUT* register. The specific operation used is determined by the ATOMIC_OPERATION register seen in the above table.

ATOMIC_SET: ATOMIC_SET is a 1 bit wide read/write register. This register is used to trigger the atomic operation on the *OUTPUT* register. When ATOMIC_SET is written to, the operation specified in ATOMIC_OPERATION is performed on the *OUTPUT* register.

VIRTUAL_PORT_MAP: VIRTUAL_PORT_MAP is a *sizeOfVirtualPorts* bits wide read/write register. This register maps the virtual ports to the physical pins. Each bit in the register corresponds to a virtual port, and the value of the bit determines which physical pin the virtual port is mapped to according to the following table by default:

VIRTUAL_PORT_OUTPUT: VIRTUAL_PORT_OUTPUT is a *numVirtualPorts* bits wide read/write register. This register sets the output value of the virtual ports. Each bit in the register corresponds to a virtual port, and the value of the bit determines the output value of the virtual port.

VIRTUAL_PORT_ENABLE: VIRTUAL_PORT_ENABLE is a 1 bit wide read/write register. This register enables the virtual ports. When VIRTUAL_PORT_ENABLE is set to '1', the virtual ports are enabled. When VIRTUAL_PORT_ENABLE is set to '0', the virtual ports are disabled.

VIRTUAL_PORT_MAP[n]	Physical Pin
0	Pin 0
1	Pin 0
2	Pin 0
3	Pin 0
4	Pin 0
5	Pin 0
6	Pin 0
7	Pin 0

Table 11: VIRTUAL_PORT_MAP Register

TRIGGER_TYPE: TRIGGER_TYPE is a *dataWidth* bits wide read/write register. This register configures whether *gpioInput* is a level or edge sensitive interrupt trigger as seen below:

TRIGGER_TYPE[n]	Type
0	Level
1	Edge

Table 12: TRIGGER_TYPE Register

TRIGGER_LO: TRIGGER_LO is a *dataWidth* bits wide read/write register. This register configures whether the interrupt is triggered on a level low, or a falling edge, of *gpioInput* depending on how TRIGGER_TYPE is set. Operation can be see in Table:

TRIGGER_LO[n]	Level Trigger	Edge Trigger
0	No Trigger when Low	No Trigger on Falling Edge
1	Trigger when Low	Trigger on Falling Edge

Table 13: TRIGGER_LO Register

TRIGGER_HI: TRIGGER_HI is a *dataWidth* bits wide read/write register. This register configures whether the interrupt is triggered on a level high, or a rising edge, of *gpioInput* depending on how TRIGGER_TYPE is set. Operation can be see in Table:

TRIGGER_HI[n]	Level Trigger	Edge Trigger
0	No Trigger when High	No Trigger on Rising Edge
1	Trigger when High	Trigger on Rising Edge

Table 14: TRIGGER_HI Register

TRIGGER_STATUS: TRIGGER_STATUS is a *dataWidth* bits wide read/write register. This register sets a corresponding bit to '1' if a trigger condition is met on the corresponding *gpioInput[n]* according to the settings of TRIGGER_TYPE, TRIGGER_LO, and TRIGGER_HI.

TRIGGER_STATUS may be read on the PRDATA bus to determine if a trigger condition has occurred. Writing a '1' to TRIGGER_STATUS[n] will clear the status of the corresponding bit. If a new trigger is detected simulatenously during this write, the TRIGGER_STATUS[n] will remain set.

TRIGGER_STATUS[n]	Status
0	No Trigger Detected
1	Trigger Detected

Table 15: TRIGGER_STATUS Register

IRQ_ENABLE: IRQ_ENABLE is a *dataWidth* bits wide read/write register. This register determines if the *irqOutput* pin is asserted when a trigger condition occurs on the corresponding TRIGGER_STATUS[n]. IRQ_ENABLE is responsible for enabling interrupt generation from the Gpio core.

IRQ_ENABLE[n]	Definition
0	Disable IRQ Generation
1	Enable IRQ Generation

Table 16: IRQ_ENABLE Register

5.3.2 Register Addresses:

Register Name	Address Start	Address End
DIRECTION	0x0	0x0
OUTPUT	0x1	0x1
INPUT	0x2	0x2
MODE	0x3	0x3
ATOMIC_OPERATION	0x4	0x4
ATOMIC_MASK	0x5	0x5
ATOMIC_SET	0x6	0x6
VIRTUAL_PORT_MAP	0x7	0x7
VIRTUAL_PORT_OUTPUT	0x8	0x8
VIRTUAL_PORT_ENABLE	0x9	0x9
TRIGGER_TYPE	0xA	0xA
TRIGGER_LO	0xB	0xB
TRIGGER_HI	0xC	0xC
TRIGGER_STATUS	0xD	0xD
IRQ_ENABLE	0xE	0xE

Table 17: 8-bit Register Addressing

dataWidth: 8

Register Name	Address Start	Address End
DIRECTION	0x00	0x01
OUTPUT	0x02	0x03
INPUT	0x04	0x05
MODE	0x06	0x07
ATOMIC_OPERATION	0x08	0x09
ATOMIC_MASK	0x0A	0x0B
ATOMIC_SET	0x0C	0x0D
VIRTUAL_PORT_MAP	0x0E	0x0F
VIRTUAL_PORT_OUTPUT	0x10	0x20
VIRTUAL_PORT_ENABLE	0x30	0x40
TRIGGER_TYPE	0x50	0x60
TRIGGER_LO	0x70	0x80
TRIGGER_HI	0x90	0xA0
TRIGGER_STATUS	0xB0	0xC0
IRQ_ENABLE	0xD0	0xE0

Table 18: 16-bit Register Addressing

dataWidth: 16

Register Name	Address Start	Address End
DIRECTION	0x0000	0x0003
OUTPUT	0x0004	0x0007
INPUT	0x0008	0x00B0
MODE	0x00C0	0x00F0
ATOMIC_OPERATION	0x0010	0x0040
ATOMIC_MASK	0x0050	0x0080
ATOMIC_SET	0x0090	0x00C0
VIRTUAL_PORT_MAP	0x00D0	0x0100
VIRTUAL_PORT_OUTPUT	0x0200	0x0500
VIRTUAL_PORT_ENABLE	0x0600	0x0900
TRIGGER_TYPE	0x0A00	0x0D00
TRIGGER_LO	0x0E00	0x2000
TRIGGER_HI	0x3000	0x6000
TRIGGER_STATUS	0x7000	0xA000
IRQ_ENABLE	0xB000	0xE000

Table 19: 32-bit Register Addressing

dataWidth: 32

5.4 Virtual Ports

5.4.1 Introduction

Virtual ports in the Gpio module allow for the abstraction of physical pins, enabling flexible control without altering the underlying hardware behavior. A virtual port can be mapped to a physical pin, and its behavior will correspond to the mode (input or output) of the pin it is mapped to.
This section details how virtual ports should behave when mapped to physical pins, how data flows between them, and how enable signals are managed.

5.4.2 Physical Pin Configured as Output

When the physical pin is configured as an output, the virtual port should mirror this configuration.

- **Data Flow**: The virtual port writes data to the corresponding physical pin.
  - Writing to the virtual port sets the output value of the physical pin.
  - The virtual port is implicitly in output mode.
- **Enable Behavior**: Writing to the virtual port acts as if writing directly to the physical pin.
  - Virtual port output is enabled when the physical pin output is enabled.

Example:

- Physical pin p is set as output.
- Virtual port v is mapped to pin p.
- Writing a 1 to virtual port v sets the output of pin p to 1.

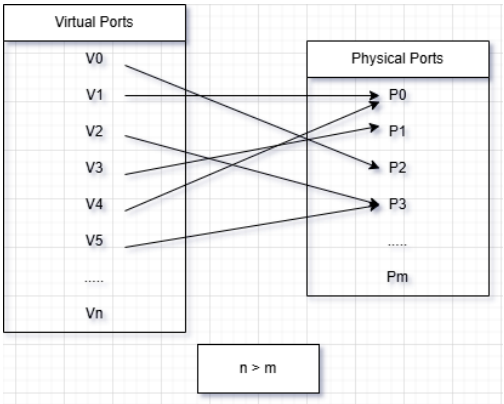

Figure 5: Mapping of virtual ports to physical pins. Virtual ports v1 and v4 overlap on Physical port p0, while Virtual ports v2 and v5 overlap on Physical port p1.

5.4.3 Physical Pin Configured as Input

When the physical pin is configured as an input, the virtual port should reflect the data from the physical pin.

- **Data Flow**: The virtual port reads the value of the physical pin.
  - Any read from the virtual port reflects the current value of the physical pin.
  - The virtual port is implicitly in input mode.
- **Enable Behavior**: Reading from the virtual port behaves as if reading directly from the physical pin.
  - Virtual port input is enabled when the physical pin input is enabled.

Example:

- Physical pin p is set as input.
- Virtual port v is mapped to pin p.
- Reading from virtual port v returns the current value of physical pin p (either 0 or 1).

5.4.4 Physical Pin Reconfiguration (Dynamic Behavior)

If the direction of the physical pin changes dynamically during runtime, the virtual port should reflect this change.

- When a physical pin changes from **input to output**, the virtual port should switch from **read-only** to **write-enabled**.

- When a physical pin changes from **output to input**, the virtual port should switch from **write-enabled** to **read-only**.
- The virtual port should respect any changes to the physical pin's enable signal (e.g., when a pin is disabled or tri-stated).

5.4.5 Summary of Correspondence

The following table summarizes the behavior of virtual ports when mapped to physical pins.

Physical Pin Mode	Virtual Port Behavior	Direction	Enable Behavior
Output	Writes to virtual port propagate to physical pin	Implicit Output	Enabled if physical pin output is enabled
Input	Reads from virtual port reflect the physical pin value	Implicit Input	Enabled if physical pin input is enabled

Table 20: Virtual Port Behavior Correspondence with Physical Pins

5.4.6 Additional Considerations

- **Virtual-to-Physical Map**: Ensure that the `virtualToPhysicalMap` correctly identifies which physical pin a virtual port is mapped to and maintains this mapping throughout the operation.
- **Enable Flag**: The virtual port enable flag should be checked to ensure that virtual ports are supported. If not enabled, virtual ports should not interact with physical pins.

By maintaining consistent mapping and handling between virtual and physical ports, you ensure that virtual ports act as an abstraction layer, extending Gpio functionality without altering the behavior of the physical pins.

5.5 Atomic Operations

5.5.1 Introduction

Atomic Operations allow the Gpio Module to set ports atomically without the need for multiple transactions. This section details how atomic operations work, how they are configured, and how they can be used to set multiple ports simultaneously.
For some bit string $p_3p_2p_1p_0$, the operation is as follows:

$$\begin{aligned} \text{OUTPUT[i]} \leftarrow\ & (\text{OUTPUT[i]} \ \&\ \text{MASK[i]} \ \&\ p_2)\ \text{OR} \\ & (\text{OUTPUT[i]} \ \&\ !\text{MASK[i]} \ \&\ p_1)\ \text{OR} \\ & (!\text{OUTPUT[i]} \ \&\ \text{MASK[i]} \ \&\ p_3)\ \text{OR} \\ & (!\text{OUTPUT[i]} \ \&\ !\text{MASK[i]} \ \&\ p_0) \end{aligned}$$

5.5.2 Diagram of Atomic Operations

Below is a diagram of the atomic operation.

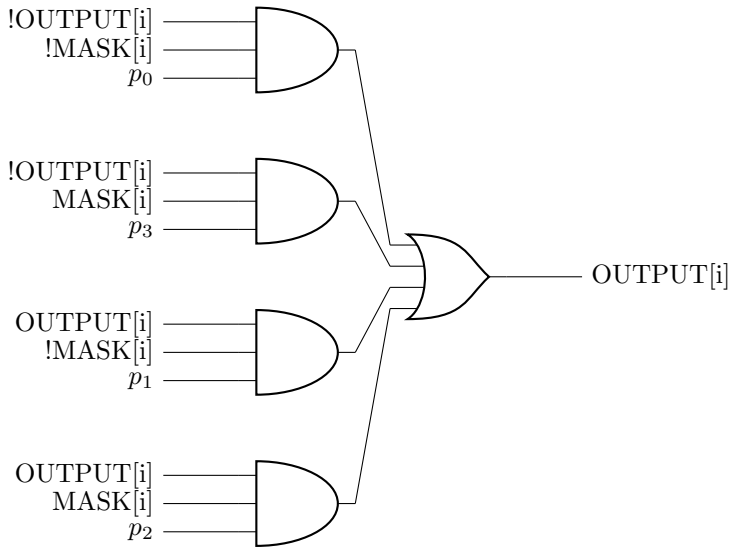

Figure 6: Atomic Operation Diagram

6 Simulation

6.1 Tests

The test bench generates a number (default is 2) configurations of the GPIO that are highly randomized. The user can run the full test-suite with n-configurations using "sbt "test" -DtestName="regression"".
User can also run individual tests or test groups by substituting "regression" with the relevant test name. A description of the tests is below:

Test Group Name	Test Name	Test Type	Test Description
basicRegisterRW	directionRegister	Directed	Writes and Reads Direction Register w/ Random Data
basicRegisterRW	modeRegister	Directed	Writes and Reads Mode Register w/ Random Data
basicRegisterRW	outputRegister	Directed	Writes and Reads Output Register w/ Random Data
basicRegisterRW	inputRegister	Directed	Writes and Reads Input Register w/ Random Data
basicRegisterRW	invalidAddress	Directed	Writes and Reads from Invalid Address Space
maskingRegisters	maskingAnd	Directed	Atomic Register "AND" Operation
maskingRegisters	maskingAndMode	Random	Set "AND" Atomic Register, Configure Mode, Test Output
interruptTriggers	triggerHigh	Directed	Test Interrupt Trigger on Level High
interruptTriggers	triggerLow	Directed	Test Interrupt Trigger on Level Low
interruptTriggers	triggerRising	Directed	Test Interrupt Trigger on Rising Edge
interruptTriggers	triggerFalling	Directed	Test Interrupt Trigger on Falling Edge
interruptTriggers	combinedTriggerLevel	Random	Test All Interrupt Triggers at Same Time on Different Ports
virtualPorts	virtualMapping	Directed	Test Virtual Port to Physical Port Mapping
virtualPorts	virtualWriting	Directed	Writing to a Virtual Port w/ Random Data
virtualPorts	virtualToPhysical	Directed	Verify Physcial Port Output after Virtual Port Write
virtualPorts	disableVirtual	Directed	Disable Virtual Port and Verify Physical Port Output
virtualPorts	invalidVirtual	Directed	Invalid Virtual Port Mapping
virtualPorts	disabledVirtualRead	Directed	Read from Disabled Virtual Port
virtualPorts	overlappingVirtualPorts	Random	Overlaping Virtual Ports Mapped to the Same Physical Port
virtualPorts	virtualInput	Directed	Virtual Port as Input
modeOperation	pushPullMode	Random	Write to random ports, set random ports to PPL, test output
modeOperation	drainMode	Random	Write to random ports, set random ports to open drain mode, test output

Table 21: Test Suite

6.2 Toggle Coverage

Current Score: 100%
All inputs and outputs are checked to insure each toggle at least once. If coverage is enabled during core instantiation, a cumulative coverage report of all tests is generated under ./out/cov/verliog.
Exceptions are higher bits of the *PADDR*, *PWDATA*, and *PRDATA* which are intended to be static during each simulation. These signals are excluded from coverage checks.

6.3 Code Coverage

Current Score: 82.17%
Code coverage for all tests can be generated as follows:

```
$ sbt coverage test -DtestName="allTests"
$ sbt coverageReport
$ python3 -m http.server 8000 --directory target/scoverage-report/
View report on local host: http://localhost:8000/index.html
```

6.4 Running simulation

Simulations can be run directly from the command prompt as follows:

```
$ sbt test -DtestName="allTests"
```
or from make as follows:
```
$ make test
```

6.5 Viewing the waveforms

The simulation generates an FST file that can be viewed using a waveform viewer. The command to view the waveform is as follows:

```
$ gtkwave ./out/test/Gpio.fst
```

7 Synthesis

7.1 Area

The Gpio has been tested in a number of configurations and the following results should be representative of what a user should see in their own technology.

Config Name	gpioWidth	numVirtualPorts	sizeOfVirtualPorts	Gates
8_8_8	8	8	8	6734
16_8_8	16	8	8	9325
32_8_8	32	8	8	14981

Table 23: Synthesis results

7.2 SDC File

An `.sdc` file is generated to provide synthesis and static timing analysis tools guidance for synthesis.
The `Gpio.sdc` file is emitted and found in the `./out/sta` directory.

7.3 Timing

The following timing was extracted using the generated .sdc files using the Nangate 45nm free library.

Config Name	Period	Duty Cycle	Input Delay	Output Delay	Slack
8_8_8	5ns	50%	20%	20%	1.79 (MET)
8_8_8	5ns	50%	20%	20%	1.79 (MET)
8_8_8	5ns	50%	20%	20%	1.81 (MET)

Table 25: Static Timing Analysis results

7.4 Multicycle Paths

None.

8 Usage Examples

8.1 Direction Register Configuration

The following example demonstrates how to configure GPIO pin directions:

```
// 1. Configure GPIO pins 0-7 as outputs, 8-15 as inputs
write(GPIO_DIRECTION_0, 0x00FF);    // Lower byte = outputs
write(GPIO_DIRECTION_1, 0xFF00);    // Upper byte = inputs

// 2. Verify configuration
uint16_t dir0 = read(GPIO_DIRECTION_0);
uint16_t dir1 = read(GPIO_DIRECTION_1);

if ((dir0 != 0x00FF) || (dir1 != 0xFF00)) {
    // Configuration error
    // ...
    return;
}
```

8.2 Output Register Operation

The following example demonstrates writing to output pins:

```
// 1. Set pins 0-3 high, others low (assuming pins are configured as outputs)
write(GPIO_OUTPUT_0, 0x000F);

// 2. Toggle output pins
uint16_t current = read(GPIO_OUTPUT_0);
write(GPIO_OUTPUT_0, ~current & 0xFFFF);

// 3. Write to specific pin (bit 5)
uint16_t new_value = (read(GPIO_OUTPUT_0) | (1 << 5));
write(GPIO_OUTPUT_0, new_value);
```

8.3 Input Register Operation

The following example demonstrates reading input pins:

```
// 1. Read all input pins (assuming pins are configured as inputs)
uint16_t inputs = read(GPIO_INPUT_0);

// 2. Check specific pin state (bit 7)
if (inputs & (1 << 7)) {
    // Pin is high
    // ...
} else {
    // Pin is low
    // ...
}

// 3. Wait for pin state change
uint16_t previous = read(GPIO_INPUT_0);
while ((read(GPIO_INPUT_0) & 0x0100) == (previous & 0x0100)) {
    // Wait for bit 8 to change
}
```

8.4 Mode Register Configuration

The following example demonstrates configuring pin modes:

```
// 1. Configure pins 0-3 as push-pull, 4-7 as open-drain
write(GPIO_MODE_0, 0x00F0);    // 00001111 (0-3: push-pull, 4-7: open-drain)

// 2. Configure pins 8-11 with pull-up, 12-15 with pull-down
write(GPIO_MODE_1, 0x0F00);    // 00001111 (8-11: pull-up, 12-15: pull-down)

// 3. Verify configuration
if ((read(GPIO_MODE_0) != 0x00F0) || (read(GPIO_MODE_1) != 0x0F00)) {
    // Handle configuration error
    // ...
}
```

8.5 Error Handling

The following example demonstrates invalid address handling:

```
// 1. Attempt to write to invalid address
uint32_t status = write(INVALID_ADDR, 0x1234);

// 2. Check for error
if (status & ERROR_BIT) {
    // Handle invalid address error
    printf("Error: Invalid address accessed\n");
    // ...
}

// 3. Attempt to read from invalid address
uint16_t data = read(INVALID_ADDR);
if (data != 0) {
    // Handle unexpected data
    // ...
}
```