GPIO
Product User Guide

rocksavagetech.chiselWare.GPIO

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

Abdelrahman Abbas, Ahmed Elmenshawi, Nick Allison, Jimmy Bright

October 5, 2024

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 recieved from the GPIO
gpioOutputEnable	*dataWidth*	Output	Enable data to be recieved from the GPIO
irqOutput	1	Output	Sent when interrupt is triggered on the GPIO

Table 1: 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 2: 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 3: Parameter Descriptions

The GPIO is instantiated into a design as follows:

```
// Valid GPIO Instantiation Example
val myGPIO = new GPIO(
  dataWidth = 32,
  addrWidth = 32 )
```

4 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	DESC TODO
OUTPUT	dataWidth	DESC TODO
INPUT	dataWidth	DESC TODO
MODE	dataWidth	DESC TODO
ATOMIC_OPERATION	4	DESC TODO
ATOMIC_MASK	p.dataWidth	DESC TODO
ATOMIC_SET	1	DESC TODO
VIRTUAL_PORT_MAP	sizeOfVirtualPorts	DESC TODO
VIRTUAL_PORT_OUTPUT	numVirtualPorts	DESC TODO
VIRTUAL_PORT_ENABLE	1	DESC TODO
TRIGGER_TYPE	dataWidth	DESC TODO
TRIGGER_LO	dataWidth	DESC TODO
TRIGGER_HI	dataWidth	DESC TODO
TRIGGER_STATUS	dataWidth	DESC TODO
IRQ_ENABLE	dataWidth	DESC TODO

4.1 Register Operation

4.1.1 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

DIRECTION[n]	Direction
0	Input
1	Output

Table 5: DIRECTION Register

4.1.2 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.

4.1.3 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.

4.1.4 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 6: MODE Register

4.1.5 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.

For some bit string $p_3p_2p_1p_0$, the operation is as follows:

$$\begin{aligned} \mathrm{OUTPUT[i]} \leftarrow\ & \mathrm{OUTPUT[i]\ \&\ MASK[i]\ \&\ } p_2 \\ & \mathrm{OUTPUT[i]\ \&\ !MASK[i]\ \&\ } p_1 \\ & \mathrm{!OUTPUT[i]\ \&\ MASK[i]\ \&\ } p_3 \\ & \mathrm{!OUTPUT[i]\ \&\ !MASK[i]\ \&\ } p_0 \end{aligned}$$

4.1.6 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.

4.1.7 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.

4.1.8 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 7: TRIGGER_TYPE Register

4.1.9 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 8: TRIGGER_LO Register

4.1.10 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 9: TRIGGER_HI Register

4.1.11 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 10: TRIGGER_STATUS Register

4.1.12 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 11: IRQ_ENABLE Register

4.2 Register Addresses

4.2.1 dataWidth: 8

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 12: 8-bit Register Addressing

4.2.2 dataWidth: 16

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 13: 16-bit Register Addressing

4.2.3 dataWidth: 32

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 14: 32-bit Register Addressing

5 Virtual Ports

When a virtual port is mapped to a physical pin in your GPIO module, the behavior of the virtual port should directly correspond to the mode (input or output) of the physical pin it is mapped to. Here's a breakdown of how the virtual port should behave in each scenario:

1. Physical Pin Configured as Output

- **Data Flow**: When the physical pin is configured as an output, the virtual port should mirror the behavior of the physical pin in the output direction.
  - The virtual port **writes** data to the same physical pin.
  - Any **write** to the virtual port should directly translate into setting the output value of the physical pin.
  - The direction of the virtual port is **implicitly output**, since it is attached to a physical output pin.
- **Enable Behavior**: If virtual ports are supported and enabled, writing to the virtual port should behave as if you are writing directly to the physical pin.
  - The virtual port output should be enabled when the corresponding physical pin's output is enabled.

Example:

- Physical pin p is configured as an output.
- Virtual port v is mapped to pin p.
- Writing 1 to virtual port v should output 1 on physical pin p.

2. Physical Pin Configured as Input

- **Data Flow**: When the physical pin is configured as an input, the virtual port should reflect the data coming **from** the physical pin.
  - The virtual port can **read** the value of the physical pin but cannot write to it.
  - Any **read** from the virtual port should return the current value of the physical pin.
  - The virtual port direction is implicitly **input**, since it is attached to a physical input pin.
- **Enable Behavior**: If virtual ports are supported and enabled, reading from the virtual port should behave as if you are reading directly from the physical pin.
  - The virtual port input should be enabled when the physical pin's input is enabled.

Example:

- Physical pin p is configured as an input.
- Virtual port v is mapped to pin p.
- Reading from virtual port v should return the current state of physical pin p (either 0 or 1).

3. Physical Pin Reconfiguration (Dynamic Behavior)

- If the direction of the physical pin changes dynamically during runtime, the virtual port's behavior should immediately reflect this change.

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

- The virtual port should also respect any changes to the physical pin's enable signal (e.g., when a pin is disabled or tri-stated).

Summary of Correspondence

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

Additional Considerations

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

By maintaining this mapping behavior, you can ensure that virtual ports act as an abstraction over physical pins, seamlessly extending the functionality of the GPIO without altering the underlying physical behavior.

6 Theory of Operations

6.1 Introduction

The **DynamicFifo** is a highly parameterized FIFO and FIFO controller. It is configurable as a full self-contained FIFO with internal memory being constructed from flip-flops, or a FIFO controller that uses an external SRAM for memory.

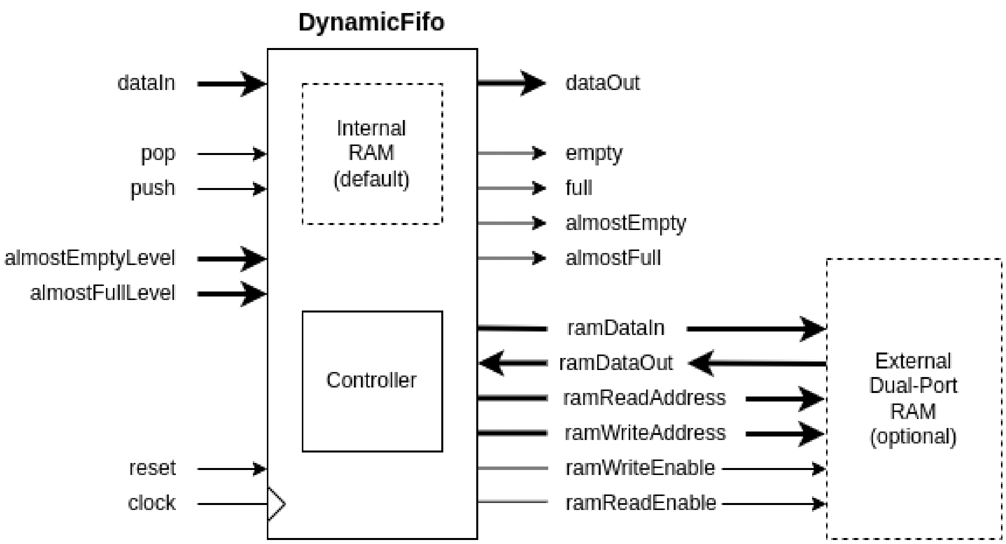

Figure 1: Block Diagram

It features the following status flags which are described in Table 1.

- empty
- full
- almostEmpty
- almostFull

When *push* is asserted, the data on the *dataIn* port is enqued on the next rising edge of *clock*. When *pop* is asserted, the top of the FIFO is dequed and immediately available on the *dataOut* port. Pop and Push operations can be simulataneous.

There are two error conditions which produce the following effects:

- When *pop* is asserted and the FIFO is empty (*empty* is active), *dataOut* will contain the last valid data held in the FIFO.
- When *push* is asserted and the FIFO is full (*full* is active), *dataIn* will be ignored and not enqued.

The *almostEmpty* and *almostFull* flags allow for additional feedback to the system that is useful for optimizing data flow control. The levels of these flags can be programmed dynamically through the *almostEmptyLevel* and *almostFullLevel* ports.

6.2 Interface Timing

GPIO has a synchronous APB3 interface, and a GPIO interface. The timing diagram shown below in Figure 2 represents an instantiation with the following parameters.

```
val myGPIO = new GPIO(
  dataWidth = 16,
  addrWidth = 16 )
```

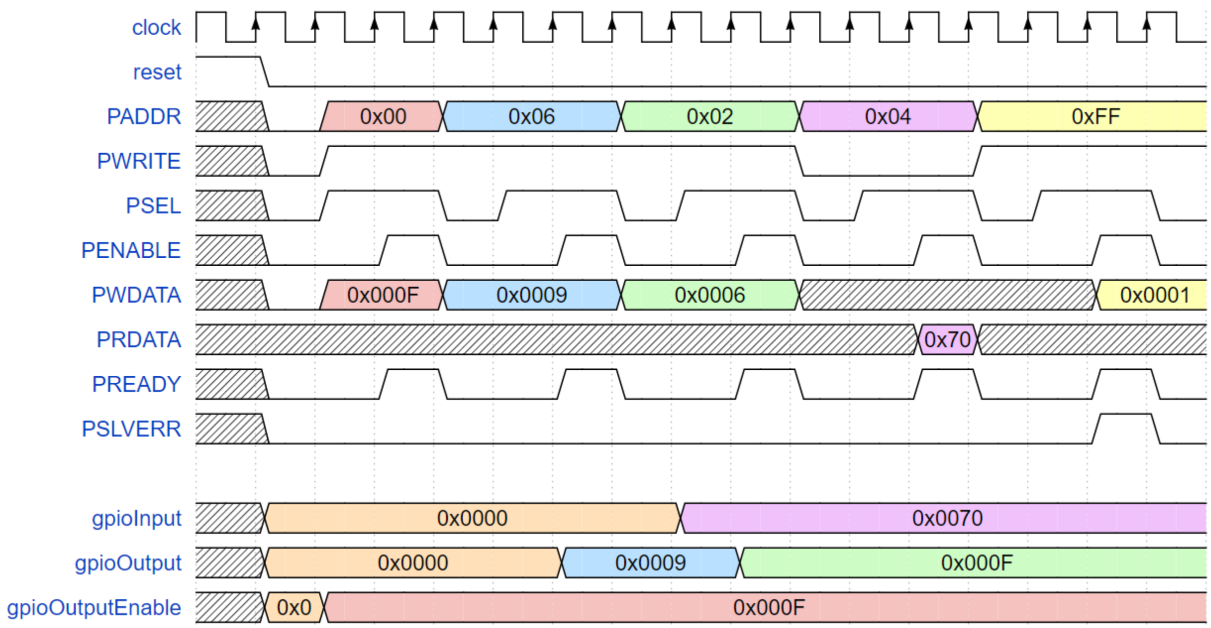

Figure 2: Timing Diagram

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.

7 Simulation

7.1 Tests

The test bench generates a number (default is 50) configurations of the DynamicFifo that are highly randomized. There are two flavors of tests:

- Directed tests that fill the FIFO with random data and then read back the results to verify that the read data matches the writted data.
- Lengthy random tests that are used to check odd combinations of configurations and to compile code coverage data.

7.2 Code coverage

All inputs and outputs are checked to insure each toggle at least once. An error will be thrown in case any port fails to toggle.

The only exception are the *almostEmptyLevel* and *almostFullLevel* which are intended to be static during each simulation. These signals are excluded from coverage checks.

7.3 Running simulation

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

```
$ sbt "test"
```

or from make as follows:

```
$ make test
```

7.4 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 ./generated/GPIO.fst
```

8 Synthesis

8.1 Area

The DynamicFifo 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	externalRAM	dataWidth	fifoDepth	Gates
small_false_8_8	false	8	8	769
medium_false_32_64	false	32	64	19,283
large_false_64_256	false	64	256	152,808
small_true_64_256	true	64	256	355
medium_true_128_128	true	128	128	477
large_true_256_2048	true	256	2048	502

Table 15: Synthesis results

8.2 SDC File

An `.sdc` file is generated to provide synthesis and static timing analysis tools guidance for synthesis.

The `DynamicFifo.sdc` file is emitted and found in the `./syn` directory.

8.3 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
small_false_8_8	5ns	50%	20%	20%	2.93 (MET)
medium_false_32_64	5ns	50%	20%	20%	2.69 (MET)
large_false_64_256	5ns	50%	20%	20%	2.80 (MET)
small_true_64_256	5ns	50%	20%	20%	2.80 (MET)
medium_true_128_128	5ns	50%	20%	20%	2.70 (MET)
large_true_256_2048	5ns	50%	20%	20%	2.77 (MET)

Table 16: Static Timing Analysis results

8.4 Multicycle Paths

None.