Prof. Dr.-Ing. A. Siggelkow

Specification - RWU-RV64I

Hochschule Ravensburg-Weingarten
University of Applied Sciences

B

Chapter 1

Requirements

Table 1.1: Functional Requirements

Requirement	ID	Importance	Verifiable	Description	Remarks
General					
Gen.: #persons	G01	High	VHDL-TB	The number of persons in a room must be counted.	
Gen.: max	G02	High	VHDL-TB	The number of persons in a room must not exceed a given limit.	
Gen.: only one pers.	G03	High	?	Only one person can either enter or leave the room at a time.	Check test
Gen.: three light sensors	G04	Medium	VHDL-TB	Along the doorway, there are three light-curtains to allow direction-tracking of possible visitors.	Why not only two?

Chapter 2

Document Overview

2.1 Validity of this document

This document is valid for RWU-RV64I version:

- V1.0: First samples
- V1.x: Revision

2.2 Target Specification Status

This RWU-RV64I target specification is initially derived from the RV64I specification. So readers being familiar with the RV64I architecture concepts and hardware building blocks will find most of them unchanged within this document.

This RWU-RV64I document represents and describes all features of the RWU-RV64I and is adequate as a standalone reference.

This release of the specification has the status of an target specification.

2.3 Major Changes since last Revision

Table 2.1: Spec Changes as compared to the previous RWU-RV64I Spec Revision

History		
Target Spec.	Current version : 1.0, 2022-11-22 Previous version : 0.0, 2020-10-12	Author: A. Siggelkow
Paragraph (in previous version)	Paragraph (in current version) / date	Subjects (major changes since last revision)
Current Chapter Version: C0.1; Date: 2022-11-22		
	2022-11-22	First draft (AS)

Contents

List of Figures

List of Tables

2.4 Glossary

Table 2.2: Glossary Type Descriptions

Type	Description
UART	Universal Asynchronous Receiver Transmitter
Bus	Binary Unit System
RGB	Red-Green-Blue. A color mixing system.
VGA	Video Graphics Array. A computer graphics standard.

2.5 Naming Conventions

Table 2.3: Alias Names for Cores and other Objects in the Spec

Object	Names also used in the spec
ARM926EJ-S ™	ARM, Controller, Micro controller unit, MCU, Processor, CPU
Bus	Binary Unit System

2.6 Document List

Table 2.4: History

History		
Target Spec.	Current version : 1.0, 2022-11-22 Previous version : 0.0, 2020-10-12	Author: A. Siggelkow
Paragraph (in previous version)	Paragraph (in current version) / date	Subjects (major changes since last revision)
Current Chapter Version: C0.1; Date: 2022-11-22		
	2022-11-22	First draft (AS)

The following documents are referred to in this document. They are either available in a separate attachment directory or are general standards.

Table 2.5: Document List

Nr	Doc Type	Version	Date	Filename	Attached
1	ARM engineering specification	A09	15.10.2024	ARM926EJ-S_EngSpec_A09	No
2	ARM926 technical reference manual	-	16.10.2024	DDI0198B_926	Yes

Chapter 3

Product Overview

3.1 Top Level View

Table 3.1: History

History		
Target Spec.	Current version : 1.0, 2025-02-22 Previous version : 0.0, 2020-10-12	Author: A. Siggelkow
Paragraph (in previous version)	Paragraph (in current version) / date	Subjects (major changes since last revision)
Current Chapter Version: C0.1; Date: 2025-02-22		
	2025-02-22	First draft (AS)

3.1.1 Introduction

The RWU-RV64I is a RISC-V processor with the integer 64 bit instruction set.

Figure 3.1 shows the top level of the chip.

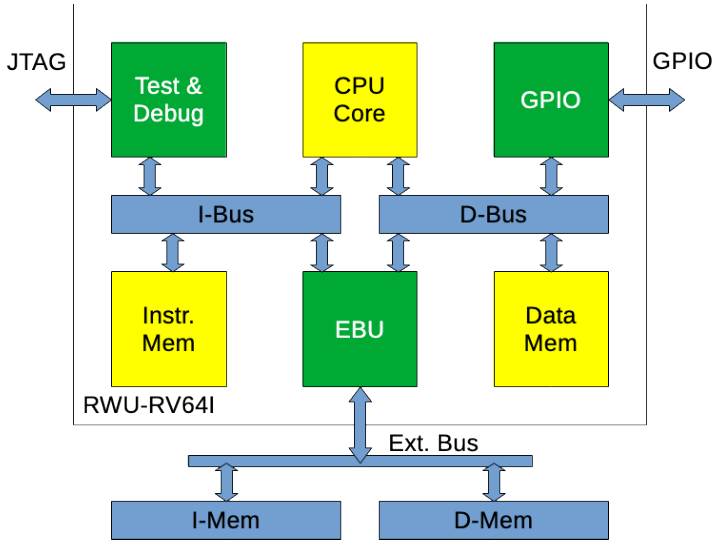

Figure 3.1: RWU-RV64I Overview

3.1.2 Key Features

First version, single cycle, on-chip memory only:

- RISC-V RV64I core
- Instruction memory: 65 kB
- Data memory: 65 kB
- JTAG, for loading the I-Mem

3.1.3 Functional Block Diagram

This block diagram shows only the most important architectural features. To maintain legibility some aspects have been simplified. More details of architecture, concepts and block integration are contained in other chapters of this specification.

Figure 3.2 shows a simplified block diagram of the chip.

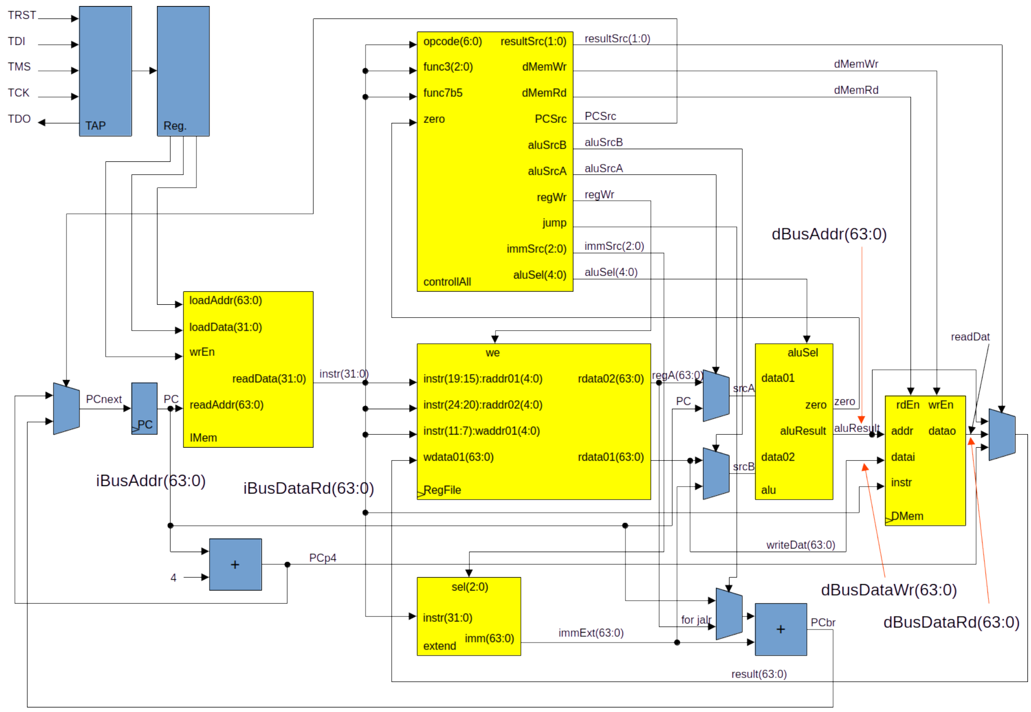

Figure 3.2: Simplified Block diagram

3.1.4 Package

The package is not defined yet.

3.2 System View

Table 3.2: History

History		
Target Spec.	Current version : 1.0, 2025-02-22 Previous version : 0.0, 2020-10-12	Author: A. Siggelkow
Paragraph (in previous version)	Paragraph (in current version) / date	Subjects (major changes since last revision)
Current Chapter Version: C0.1; Date: 2025-02-22		
	2025-02-22	First draft (AS)

The system is not defined yet.

3.3 Architecture Concepts Overview

Table 3.3: History

History		
Target Spec.	Current version : 1.0, 2025-02-22 Previous version : 0.0, 2020-10-12	Author: A. Siggelkow
Paragraph (in previous version)	Paragraph (in current version) / date	Subjects (major changes since last revision)
Current Chapter Version: C0.1; Date: 2025-02-22		
	2025-02-22	First draft (AS)

3.3.1 Technology

Europractice 90 nm technology.

Infineon's 90 nm (70 nm physical gate length) CMOS technology is designed for digital and analog circuit design. It offers advanced possibilities for low power design (triple well, low leakage transistor types). The technology has 4-9 layers of copper metallization. It offers an increase in peak performance of more than 50% and a reduction of active power consumption of more than a factor of 2. Compared to the predecessor technology twice the cell density for standard cells can be achieved in L90.

The nominal core voltage is 1.2 V, IO devices have a nominal supply voltage of 2.5 V or 1.8 V.

3.3.2 System Memory Concept

The first version has an on-chip instruction memory of ?? MB and an on-chip data memory of ?? MB. The technology is ??.

3.3.3 Software Architecture

The RISC-V compiler ??? is supported.

3.3.4 Interprocessor Communication Concept

Planned: UART and (Q)SPI.

3.3.5 Bus Concept

The main bus system is compliant to the Wishbone specification.

There is on WB bus for the instructions and one for the data.

3.3.6 Clock Concept

Not defined yet.

Until now, there is only one clock input. For a next generation, with external NOR flash memory for instruction and NAND flash for data, a QSPI interface will be needed. This requires a faster clock than the CPU clock.

3.3.7 System Control Concept

System control will be needed for:

- Reset and boot
- Interrupt

3.3.8 Interrupt Concept

Must be implemented.

3.3.9 Debug Concept

Must be implemented.

3.3.10 Power Management

Must be implemented.

3.3.11 Protection and Security Concept

Must be implemented.

3.4 Functional Block Overview

Table 3.4: History

History		
Target Spec.	Current version : 1.0, 2025-02-22 Previous version : 0.0, 2020-10-12	Author: A. Siggelkow
Paragraph (in previous version)	Paragraph (in current version) / date	Subjects (major changes since last revision)
Current Chapter Version: C0.1; Date: 2025-02-22		
	2025-02-22	First draft (AS)

3.4.1 RISC-V Core

The core is a standard compliant RISC-V with 64 bit and integer implementation only. It is single cycle, without cache and memory hierarchy.

3.4.2 Instruction Memory

Instruction memory:

3.4.3 Data Memory

Data memory:

3.4.4 Wishbone Peripherals

GPIO

There are 8 GPIO pins implemented. Together with the data, an address will be given to the pins. With this, 16 times 8 GPIOs can be driven with just one external decoder.

3.4.5 Debug

The test and debug will be done via a JTAG interface. Until now, the loading of the instruction memory is possible.

Chapter 4

Architecture Concepts

The logic style used here is as often as possible a "one pulse" logic.

Figure 4.1 shows the "pulsed" logic.

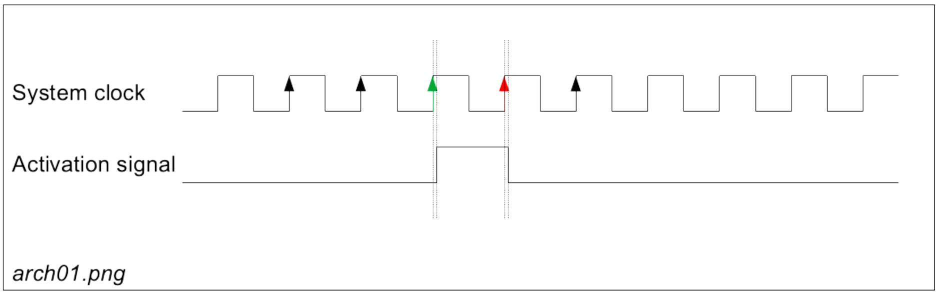

Figure 4.1: Enable with a Pulse

An "activation signal" will be generated synchronously with the "system clock" by its rising edge (green). This results in a delayed generation of the rising edge of the "activation signal". With the next rising edge of the "system clock" (red), the "activation signal" will be removed. Also here, the falling edge of the "activation signal" will have a small delay to the clock. So, the "activation signal" is exactly on "system clock" period high. This pair of signals can now be taken to start a following device (e.g. a counter), the counter will count only when the activation signal is high and the system clock will show a rising edge (red).

4.1 Technology

Table 4.1: History

History		
Target Spec.	Current version : 1.0, 2025-02-22 Previous version : 0.0, 2020-10-12	Author: A. Siggelkow
Paragraph (in previous version)	Paragraph (in current version) / date	Subjects (major changes since last revision)
Current Chapter Version: C0.1; Date: 2025-02-22		
	2025-02-22	First draft (AS)

4.2 CPU Concept

Table 4.2: History

History		
Target Spec.	Current version : 1.0, 2025-02-22 Previous version : 0.0, 2020-10-12	Author: A. Siggelkow
Paragraph (in previous version)	Paragraph (in current version) / date	Subjects (major changes since last revision)
Current Chapter Version: C0.1; Date: 2025-02-22		
	2025-02-22	First draft (AS)

The RWU-RV64I is a single cycle implementation without pipeline, memory hierarchy, MMU, etc. It is a students test chip.

Table 4.3 shows the notations which are used in the explanations for the instructions in tables 4.4 and 4.5.

Table 4.3: Notations for Instructions

Notation	Description
pc	program counter
rd	integer register destination
rsN	integer register source N
imm	immediate operand value
offs	immediate program counter relative offset
ux(reg)	unsigned XLEN-bit integer (32-bit on RV32, 64-bit on RV64)
sx(reg)	signed XLEN-bit integer (32-bit on RV32, 64-bit on RV64)
uN(reg)	zero extended N-bit integer register value
sN(reg)	sign extended N-bit integer register value
uN[reg + imm]	unsigned N-bit memory reference
sN[reg + imm]	signed N-bit memory reference

Table 4.4 lists all base integer instructions of a RISC-V 32 bit architecture according the specification [1]. A 64 bit architecture has the same instructions with an adapted length of some numbers and registers.

Table 4.4: RV32I Base Integer Instruction Set [2], [3], [1]

Instruction	Example	Meaning
Load Upper Immediate	*lui rd, imm*	$rd \leftarrow imm$
Add Upper Immediate to PC	*auipc rd, offs*	$rd \leftarrow pc + offs$
Jump and Link	*jal rd, offs*	$rd \leftarrow pc + len(instr)$, $pc \leftarrow pc + offs$
Jump and Link Register	*jalr rd, rs1, offs*	$rd \leftarrow pc + len(instr)$, $pc \leftarrow (rs1 + offs) \wedge -2$
Branch Equal	*beq rs1, rs2, offs*	$if\ rs1 = rs2\ then\ pc \leftarrow pc + offs$
Branch Not Equal	*bne rs1, rs2, offs*	$if\ rs1 \neq rs2\ then\ pc \leftarrow pc + offs$
Branch Less Than	*blt rs1, rs2, offs*	$if\ rs1 < rs2\ then\ pc \leftarrow pc + offs$
Branch Greater than Equal	*bge rs1, rs2, offs*	$if\ rs1 \geq rs2\ then\ pc \leftarrow pc + offs$

... next page

Instruction	Example	Meaning
Branch Less Than Unsigned	*bltu rs1, rs2, offs*	$if\ rs1 < rs2\ then\ pc \leftarrow pc + offs$
Branch Greater than Equal Unsigned	*bgeu rs1, rs2, offs*	$if\ rs1 \geq rs2\ then\ pc \leftarrow pc + offs$
Load Byte	*lb rd, offs(rs1)*	$rd \leftarrow s8[rs1 + offs]$
Load Half	*lh rd, offs(rs1)*	$rd \leftarrow s16[rs1 + offs]$
Load Word	*lw rd, offs(rs1)*	$rd \leftarrow s32[rs1 + offs]$
Load Byte Unsigned	*lbu rd, offs(rs1)*	$rd \leftarrow s8[rs1 + offs]$
Load Half Unsigned	*lhu rd, offs(rs1)*	$rd \leftarrow s16[rs1 + offs]$
Store Byte	*sb rs2, offs(rs1)*	$u8[rs1 + offs] \leftarrow rs2$
Store Half	*sh rs2, offs(rs1)*	$u16[rs1 + offs] \leftarrow rs2$
Store Word	*sw rs2, offs(rs1)*	$u32[rs1 + offs] \leftarrow rs2$
Add Immediate	*addi rd, rs1, imm*	$rd \leftarrow rs1 + sx(imm)$
Set Less Than Immediate	*slti rd, rs1, imm*	$rd \leftarrow sx(rs1) < sx(imm)$
Set Less Than Immediate Unsigned	*sltiu rd, rs1, imm*	$rd \leftarrow ux(rs1) < ux(imm)$
Xor Immediate	*xori rd, rs1, imm*	$rd \leftarrow ux(rs1) \oplus ux(imm)$
Or Immediate	*ori rd, rs1, imm*	$rd \leftarrow ux(rs1) \vee ux(imm)$
And Immediate	*andi rd, rs1, imm*	$rd \leftarrow ux(rs1) \wedge ux(imm)$
Shift Left Logical Immediate	*slli rd, rs1, imm*	$rd \leftarrow ux(rs1) << ux(imm)$
Shift Right Logical Immediate	*srli rd, rs1, imm*	$rd \leftarrow ux(rs1) >> ux(imm)$
Shift Right Arithmetic Immediate	*srai rd, rs1, imm*	$rd \leftarrow sx(rs1) >> ux(imm)$
Add	*add rd, rs1, rs2*	$rd \leftarrow sx(rs1) + sx(rs2)$
Subtract	*sub rd, rs1, rs2*	$rd \leftarrow sx(rs1) - sx(rs2)$
Shift Left Logical	*sll rd, rs1, rs2*	$rd \leftarrow ux(rs1) << rs2$
Set Less Than	*slt rd, rs1, rs2*	$rd \leftarrow sx(rs1) < sx(rs2)$
Set Less Than Unsigned	*sltu rd, rs1, rs2*	$rd \leftarrow ux(rs1) < ux(rs2)$
Xor	*xor rd, rs1, rs2*	$rd \leftarrow ux(rs1) \oplus ux(rs2)$
Shift Right Logical	*srl rd, rs1, rs2*	$rd \leftarrow ux(rs1) >> rs2$
Shift Right Arithmetic	*sra rd, rs1, rs2*	$rd \leftarrow sx(rs1) >> rs2$
Or	*or rd, rs1, rs2*	$rd \leftarrow ux(rs1) \vee ux(rs2)$

... next page

Instruction	Example	Meaning
And	*and rd, rs1, rs2*	$rd \leftarrow ux(rs1) \wedge ux(rs2)$
Fence	*fence pred, succ*	
Fence Instruction	*fence.i*	

End of table.

Table 4.5 lists all base integer instructions of a RISC-V 64 bit architecture, additional to the 32 bit integer instruction set, according the specification [1].

Table 4.5: RV64I Base Integer Instruction Set (in addition to RV32I) [2], [3], [1]

Instruction	Example	Meaning
Load Word Unsigned	*lwu rd, offs(rs1)*	$rd \leftarrow u32[rs1 + offs]$
Load Double	*ld rd, offs(rs1)*	$rd \leftarrow u64[rs1 + offs]$
Store Double	*sd rs2, offs(rs1)*	$u64[rs1 + offs] \leftarrow rs2$
Shift Left Logical Immediate	*slli rd, rs1, imm*	$rd \leftarrow ux(rs1) << sx(imm)$
Shift Right Logical Immediate	*srli rd, rs1, imm*	$rd \leftarrow ux(rs1) >> sx(imm)$
Shift Right Arithmetic Immediate	*srai rd, rs1, imm*	$rd \leftarrow sx(rs1) >> sx(imm)$
Add Immediate Word	*addiw rd, rs1, imm*	$rd \leftarrow s32(rs1) + imm$
Shift Left Logical Immediate Word	*slliw rd, rs1, imm*	$rd \leftarrow s32(u32(rs1) << imm)$
Shift Right Logical Immediate Word	*srliw rd, rs1, imm*	$rd \leftarrow s32(u32(rs1) >> imm)$
Shift Right Arithmetic Immediate Word	*sraiw rd, rs1, imm*	$rd \leftarrow s32(rs1) >> imm$
Add Word	*addw rd, rs1, rs2*	$rd \leftarrow s32(rs1) + s32(rs2)$
Subtract Word	*subw rd, rs1, rs2*	$rd \leftarrow s32(rs1) - s32(rs2)$
Shift Left Logical Word	*sllw rd, rs1, rs2*	$rd \leftarrow s32(u32(rs1) << rs2)$
Shift Right Logical Word	*srlw rd, rs1, rs2*	$rd \leftarrow s32(u32(rs1) >> rs2)$

... next page

Instruction	Example	Meaning
Shift Right Arithmetic Word	*sraw rd, rs1, rs2*	$rd \leftarrow s32(rs1) >> rs2$

End of table.

4.3 Bus Concept

Table 4.6: History

History		
Target Spec.	Current version : 1.0, 2025-02-22 Previous version : 0.0, 2020-10-12	Author: A. Siggelkow
Paragraph (in previous version)	Paragraph (in current version) / date	Subjects (major changes since last revision)
Current Chapter Version: C0.1; Date: 2025-02-22		
	2025-02-22	First draft (AS)

4.4 Interrupt Concept

Table 4.7: History

History		
Target Spec.	Current version : 1.0, 2025-02-22 Previous version : 0.0, 2020-10-12	Author: A. Siggelkow
Paragraph (in previous version)	Paragraph (in current version) / date	Subjects (major changes since last revision)
Current Chapter Version: C0.1; Date: 2025-02-22		
	2025-02-22	First draft (AS)

4.5 Clock Concept

Table 4.8: History

History		
Target Spec.	Current version : 1.0, 2025-02-22 Previous version : 0.0, 2020-10-12	Author: A. Siggelkow
Paragraph (in previous version)	Paragraph (in current version) / date	Subjects (major changes since last revision)
Current Chapter Version: C0.1; Date: 2025-02-22		
	2025-02-22	First draft (AS)

4.6 System Control Concept

Table 4.9: History

History		
Target Spec.	Current version : 1.0, 2025-02-22 Previous version : 0.0, 2020-10-12	Author: A. Siggelkow
Paragraph (in previous version)	Paragraph (in current version) / date	Subjects (major changes since last revision)
Current Chapter Version: C0.1; Date: 2025-02-22		
	2025-02-22	First draft (AS)

4.7 Power Concept

Table 4.10: History

History		
Target Spec.	Current version : 1.0, 2025-02-22 Previous version : 0.0, 2020-10-12	Author: A. Siggelkow
Paragraph (in previous version)	Paragraph (in current version) / date	Subjects (major changes since last revision)
Current Chapter Version: C0.1; Date: 2025-02-22		
	2025-02-22	First draft (AS)

4.8 Memory Concept

Table 4.11: History

History		
Target Spec.	Current version : 1.0, 2025-02-22 Previous version : 0.0, 2020-10-12	Author: A. Siggelkow
Paragraph (in previous version)	Paragraph (in current version) / date	Subjects (major changes since last revision)
Current Chapter Version: C0.1; Date: 2025-02-22		
	2025-02-22	First draft (AS)

4.9 System Protection and Security Concept

Table 4.12: History

History		
Target Spec.	Current version : 1.0, 2025-02-22 Previous version : 0.0, 2020-10-12	Author: A. Siggelkow
Paragraph (in previous version)	Paragraph (in current version) / date	Subjects (major changes since last revision)
Current Chapter Version: C0.1; Date: 2025-02-22		
	2025-02-22	First draft (AS)

4.10 Debug Concept

Table 4.13: History

History		
Target Spec.	Current version : 1.0, 2025-02-22 Previous version : 0.0, 2020-10-12	Author: A. Siggelkow
Paragraph (in previous version)	Paragraph (in current version) / date	Subjects (major changes since last revision)
Current Chapter Version: C0.1; Date: 2025-02-22		
	2025-02-22	First draft (AS)

4.11 Register Concept

Table 4.14: History

History		
Target Spec.	Current version : 1.0, 2025-02-22 Previous version : 0.0, 2020-10-12	Author: A. Siggelkow
Paragraph (in previous version)	Paragraph (in current version) / date	Subjects (major changes since last revision)
Current Chapter Version: C0.1; Date: 2025-02-22		
	2025-02-22	First draft (AS)

Chapter 5

RWU-RV64I Core

5.1 ALU

History

Regulations for author history:

- Only visible internally (by conditioning of the table, no other conditions needed)

Table 5.1: Authors

Author	Department	From
Andreas Siggelkow	Fac. E	February 2025

Regulations for history table:

- Last version on top.
- Dont use links in the history table. These may cause unexpected history changes in future versions.
- Name (or short sign) should be visible only internally.
- Create a new version after each (also limited) release of the document. Versioning does not depend on the overall document version.
- Use the form x.y for the version. x for the master versions, y for the project specific changes and updates.

Table 5.2: History

Date	Name	Content Changes
Version of this document: V0.1		
2025-02-13	Andreas Siggelkow	First draft (AS)

Functional Configuration of ALU0

This section should describe the product specific instantiation of this module's feature set (which may be a subset of the full feature set). It is intended also for use in the product overview section of the target spec/summary target spec.

The standard features of a ALU peripheral are described in its functional description in Section 5.1.2. Generic features, that can be decided for each specific instantiation of the peripheral are listed and completed for ALU0 in Table 5.4.

The RWU-RV64I has only one ALU.

Table 5.3: ALU0 Feature Set

	Generic GPIO Feature Set	Details
no	Seperate Kernel Clock	not needed
no	FIFO Data Buffering	not needed

HW Interfaces

The following table 5.4 lists all hardware interfaces which are relevant for the regular operation of the device. It is not a detailed list of all signals and does not include implementation specific informations.

The names used below show the functional connectivity. There is no guarantee that they are electrically compatible (e.g. voltage levels, clock domains etc. may be different). A full list is part of the design spec.

Table 5.4: HW Interfaces of Module ALU0

	Module internal name	Chip level name
Supply		
Clock	-	-
Bus	-	-
Interrupt	-	-
DMA	-	-
External Signals	-	-
Others	data01_i(63:0) data02_i(63:0) aluSel_i(4:0) aluZero_o aluNega_o aluCarr_o aluOver_o aluResult_o(63:0)	srcA_s(63:0) srcB_s(63:0) aluSel_s(4:0) zero_s nega_s carry_s overflow_s dBusAddr_s(63:0)

Bus Interface

None.

Registers

None.

Related Sections of this Spec

None.

5.1.1 History of ALU

Table 5.5: History

History		
Target Spec.	Current version : 1.0, 2025-02-13 Previous version : 0.0, 2020-10-12	Author: A. Siggelkow
Paragraph (in previous version)	Paragraph (in current version) / date	Subjects (major changes since last revision)
Current Chapter Version: C0.1; Date: 2025-02-13		
5.1	5.1 / 2025-02-13	First draft (AS)

5.1.2 Functional Overview

The ALU operates on 64 bit data and generates 64 bit data. The ALU indicates a zero result, a negative result, the carry of a result, and the overflow.

General Features:

- 64 bit integer addition
- 64 bit integer subtraction
- 64 bit AND
- 64 bit OR
- 64 bit XOR
- 64 bit set less than
- 64 bit set less than, unsigned
- 32 bit shift right arithmetic word
- 32 bit shift right logical word
- 32 bit shift left logical word
- 32 bit subtract word
- 32 bit add word

- 64 bit shift right arithmetic
- 64 bit shift right logical
- 64 bit shift left logical
- branch if equal
- branch if not equal
- branch if less than
- branch if greater or equal
- branch if less than, unsigned
- branch if greater or equal, unsigned

Synchronous Mode:

- -

Asynchronous Mode:

- -

5.1.3 Structural Overview

I/O of the ALU

Table 5.6 shows the input and outputs of the ALU0. This are all primary inputs and outputs. There is no bi-directional, tristate or open drain pin. All pins are high-active except the pins with a _n_ in its name.

Table 5.6: ALU0 I/O

Pin	Dir.	Wd.	Explanation
data01_i	in	64	Source data input. Comes either from the register file or the program conter (PC). Swiched by signal “aluSrcA”. Source one for all ALU operators.
data02_i	in	64	Source data input. Comes either from the register file or the immediate generator. Swiched by signal “aluSrcB”. Source two for all ALU operators.
aluSel_i	in	5	Selects the ALU operator.
aluZero_o	out	1	Zero flag of the ALU. Not buffered.
aluNega_o	out	1	Negative flag of the ALU. Not buffered.
aluCarr_o	out	1	Carry flag of the ALU. Not buffered.
aluOver_o	out	1	Overflow flag of the ALU. Not buffered.
aluResult_o	out	64	The result of each ALU operation.

Internals of the ALU

The following list (Table) shows the internal signals of the ALU. The table shows the signal name, the width, the driver, the target and a short explanation.

- Signal: aluSel_i[0]
  - Width: 1 bit
  - Source: aluSel_i
  - Destination: aluResult_o, overf_s
  - Explanation (special signal):
    * if aluSel_i[0] = 0, it is an ADD instruction
    * ... the second operand must be taken as it is
    * ... the carry-in of the addition is 0 (aluSel_i[0])
    * if aluSel_i[0] = 1, it is a SUB instruction
    * ... the second operand must be the 2-complement
    * ... the carry-in of the addition is 1 which acts as the +1 (aluSel_i[0])
- Signal: data01_s
  - Width: 64 bit, signed
  - Source: data01_i
  - Destination: aluResult_o, aluZero_o

  - Explanation: It is the signed interpretation of data01_i. It is needed for some branches (BLT, BGE) and SRA.

    logic signed [reg_width-1:0] data01_s;

- Signal: data02_s
  - Width: 64 bit, signed
  - Source: data02_i
  - Destination: aluResult_o, aluZero_o
  - Explanation: It is the signed interpretation of data02_i. It is needed for some branches (BLT, BGE) and SRA.

    logic signed [reg_width-1:0] data02_s;

- Signal: sum_s
  - Width: 64 bit
  - Source: data01_i, cinvb_s (data02_i), aluSel_i[0]
  - Destination: aluResult_o
  - Explanation: It is the sum of the two data inputs data01_i and data02_i. The second data input could be negative and so 2-complemented. cinvb_s is the inversion of data02_e when $aluSel\_i[0] = 1$, aluSel_i[0] also acts as the plus 1.

    assign {carry_s, sum_s} = data01_i + cinvb_s + aluSel_i[0];

- Signal: cinvb_s
  - Width: 64 bit
  - Source: data02_i, aluSel_i[0]
  - Destination: sum_s, carry_s
  - Explanation: cinvb_s is the inversion of data02_i when $aluSel\_i[0] = 1$. Together with aluSel_i[0] ($cinvb\_s+aluSel\_i[0]$) it is the 2-complement of data02_i.

    assign cinvb_s = aluSel_i[0] ? data02_i : data02_i;

- Signal: carry_s

- Width: 1 bit
- Source: data01_i, cinvb_s (data02_i), aluSel_i[0]
- Destination: aluCarr_o (carry flag)
- Explanation: It is the carry of sum_s.

  assign {carry_s, sum_s} = data01_i + cinvb_s + aluSel_i[0];

- Signal: sumw_s
  - Width: 32 bit
  - Source: data01_i, cinvb_s (data02_i), aluSel_i[0]
  - Destination: aluResult_o
  - Explanation: For add word, etc. It is the sum of the two data inputs data01_i(31:0) and data02_i(31:0. The second data input could be negative and so 2-complemented. cinvbw_s is the inversion of data02_e(31:0) when $aluSel\_i[0] = 1$, aluSel_i[0] also acts as the plus 1.

    assign {carryw_s, sumw_s} = data01_i[31:0] + cinvbw_s + aluSel_i[0];

- Signal: cinvbw_s
  - Width: 32 bit
  - Source: data02_e, aluSel_i[0]
  - Destination: sum_s, carry_s
  - Explanation: For add word, etc. cinvbw_s is the inversion of data02_i(31:0) when $aluSel\_i[0] = 1$. Together with aluSel_i[0] ($cinvbw\_s+aluSel\_i[0]$) it is the 2-complement of data02_i(31:0).

    assign cinvbw_s = aluSel_i[0] ? data02_i[31:0] : data02_i[31:0];

- Signal: carryw_s
  - Width: 1 bit
  - Source: data01_i, cinvb_s (data02_i), aluSel_i[0]
  - Destination: aluCarr_o (carry flag)
  - Explanation: For add word, etc. It is the carry of sumw_s.

    assign {carryw_s, sumw_s} = data01_i[31:0] + cinvbw_s + aluSel_i[0];

- Signal: sllw_s
  - Width: 32 bit
  - Source: data01_i, data02_i
  - Destination: aluResult_o
  - Explanation: SLLW (shift left logical word)
    Implementation:
    $sllw\_s = data01\_i(31:0) << data02\_i(4:0)$
    aluResult_o is sllw_s, sign extended.
- srlw_s
  - Width: 32 bit
  - Source: data01_i, data02_i
  - Destination: aluResult_o
  - Explanation: SRLW (shift right logical word)
    Implementation:
    $srlw\_s = data01\_i(31:0) >> data02\_i(4:0)$
    aluResult_o is srlw_s, sign extended.
- sraw_s
  - Width: 32 bit, signed
  - Source: data01_i, data02_i
  - Destination: aluResult_o
  - Explanation: SRAW (shift right algorithmic word)
    Implementation:
    $sraw\_s = data01w\_s >>> data02\_i(4:0)$
    aluResult_o is sraw_s, sign extended.
- data01w_s
  - Width: 32 bit, signed
  - Source: data01_i
  - Destination: sraw_s
  - Explanation: 32 bit signed representation of data01_i.
- sltiu_s
  - Width: 1 bit

  - Source: data01_i, data02_i
  - Destination: aluResult_o
  - Explanation: SLT, SLTU (set less than; signed, unsigned)
    Implementation:
    if(data01_i < data02_i)
    sltiu_s = 1;
    else
    sltiu_s = 0;
    aluResult_o is sltiu_s, zero extended.

- overf_s
  - Width: 1 bit
  - Source: data01_i, data02_i, sum_s, aluSel_i[0]
  - Destination: aluOver_o
  - Explanation: Overflow occurs when the result-value affects the sign:
    - overflow when adding two positives yields a negative
    - or, adding two negatives gives a positive
    - or, subtract a negative from a positive and get a negative
    - or, subtract a positive from a negative and get a positive

- aluResult_o
  - Width: 64 bit
  - Source: all signals
  - Destination: output
  - Explanation: The result of each ALU operation.
    If aluSel_i =

0	aluResult_o = sum_s;	ADD, ADDI, LB, LH, LW, LBU, LHU, LD, LWU, AUIPC, LUI, SB, SH, SW, SD, JALR, JAL
1	aluResult_o = sum_s;	SUB
2	aluResult_o = data01_i & data02_i;	AND, ANDI
3		OR
4		XOR
5		SLT
6		SLTU

5.1.4 Service Requests of the ALU

None.

5.1.5 The ALU Peripheral Kernel

The ALU calculates a result (aluResult_o) and the corresponding flags zero (aluZero_o), negative (aluNega_o), carry (asCarr_o) and overflow (asOver_o). Table 5.7 and table 5.8 are describing the function of the ALU for each instruction.

The zero flag is not only a zero flag but also the control for all branch instruction.

Table 5.7 lists all base integer instructions of a RISC-V 32 bit architecture according the specification [1]. A 64 bit architecture has the same instructions with an adapted length of some numbers and registers.

Table 5.7: RV32I Base Integer Instruction Set [2], [3], [1]

Instruction	Operation	Remarks
Load Upper Immediate	-	-
Add Upper Immediate to PC	-	-
Jump and Link	*jal rd, offs*	$rd \leftarrow pc + len(instr)$, $pc \leftarrow pc + offs$
Jump and Link Register	*jalr rd, rs1, offs*	$rd \leftarrow pc + len(instr)$, $pc \leftarrow (rs1 + offs) \wedge -2$
Branch Equal	*beq rs1, rs2, offs*	$if\ rs1 = rs2\ then\ pc \leftarrow pc + offs$
Branch Not Equal	*bne rs1, rs2, offs*	$if\ rs1 \neq rs2\ then\ pc \leftarrow pc + offs$
Branch Less Than	*blt rs1, rs2, offs*	$if\ rs1 < rs2\ then\ pc \leftarrow pc + offs$
Branch Greater than Equal	*bge rs1, rs2, offs*	$if\ rs1 \geq rs2\ then\ pc \leftarrow pc + offs$
Branch Less Than Unsigned	*bltu rs1, rs2, offs*	$if\ rs1 < rs2\ then\ pc \leftarrow pc + offs$
Branch Greater than Equal Unsigned	*bgeu rs1, rs2, offs*	$if\ rs1 \geq rs2\ then\ pc \leftarrow pc + offs$
Load Byte	*lb rd, offs(rs1)*	$rd \leftarrow s8[rs1 + offs]$
Load Half	*lh rd, offs(rs1)*	$rd \leftarrow s16[rs1 + offs]$
Load Word	*lw rd, offs(rs1)*	$rd \leftarrow s32[rs1 + offs]$

... next page

Instruction	Operation	Remarks
Load Byte Unsigned	*lbu rd, offs(rs1)*	$rd \leftarrow s8[rs1 + offs]$
Load Half Unsigned	*lhu rd, offs(rs1)*	$rd \leftarrow s16[rs1 + offs]$
Store Byte	*sb rs2, offs(rs1)*	$u8[rs1 + offs] \leftarrow rs2$
Store Half	*sh rs2, offs(rs1)*	$u16[rs1 + offs] \leftarrow rs2$
Store Word	*sw rs2, offs(rs1)*	$u32[rs1 + offs] \leftarrow rs2$
Add Immediate	*addi rd, rs1, imm*	$rd \leftarrow rs1 + sx(imm)$
Set Less Than Immediate	*slti rd, rs1, imm*	$rd \leftarrow sx(rs1) < sx(imm)$
Set Less Than Immediate Unsigned	*sltiu rd, rs1, imm*	$rd \leftarrow ux(rs1) < ux(imm)$
Xor Immediate	*xori rd, rs1, imm*	$rd \leftarrow ux(rs1) \oplus ux(imm)$
Or Immediate	*ori rd, rs1, imm*	$rd \leftarrow ux(rs1) \vee ux(imm)$
And Immediate	*andi rd, rs1, imm*	$rd \leftarrow ux(rs1) \wedge ux(imm)$
Shift Left Logical Immediate	*slli rd, rs1, imm*	$rd \leftarrow ux(rs1) << ux(imm)$
Shift Right Logical Immediate	*srli rd, rs1, imm*	$rd \leftarrow ux(rs1) >> ux(imm)$
Shift Right Arithmetic Immediate	*srai rd, rs1, imm*	$rd \leftarrow sx(rs1) >> ux(imm)$
Add	aluResult_o = sum_s = data01_i + cinvb_s + aluSel_i[0]	aluSel_i = 0; cinvb_s = aluSel_i[0] ? not data02_i : data02_i
Subtract	*sub rd, rs1, rs2*	$rd \leftarrow sx(rs1) - sx(rs2)$
Shift Left Logical	*sll rd, rs1, rs2*	$rd \leftarrow ux(rs1) << rs2$
Set Less Than	*slt rd, rs1, rs2*	$rd \leftarrow sx(rs1) < sx(rs2)$
Set Less Than Unsigned	*sltu rd, rs1, rs2*	$rd \leftarrow ux(rs1) < ux(rs2)$
Xor	*xor rd, rs1, rs2*	$rd \leftarrow ux(rs1) \oplus ux(rs2)$
Shift Right Logical	*srl rd, rs1, rs2*	$rd \leftarrow ux(rs1) >> rs2$
Shift Right Arithmetic	*sra rd, rs1, rs2*	$rd \leftarrow sx(rs1) >> rs2$
Or	*or rd, rs1, rs2*	$rd \leftarrow ux(rs1) \vee ux(rs2)$
And	*and rd, rs1, rs2*	$rd \leftarrow ux(rs1) \wedge ux(rs2)$
Fence	*fence pred, succ*	
Fence Instruction	*fence.i*	

End of table.

Table 5.8 lists all base integer instructions of a RISC-V 64 bit architecture, additional to the 32 bit integer instruction set, according the specification [1].

Table 5.8: RV64I Base Integer Instruction Set (in addition to RV32I) [2], [3], [1]

Instruction	Example	Meaning
Load Word Unsigned	*lwu rd, offs(rs1)*	$rd \leftarrow u32[rs1 + offs]$
Load Double	*ld rd, offs(rs1)*	$rd \leftarrow u64[rs1 + offs]$
Store Double	*sd rs2, offs(rs1)*	$u64[rs1 + offs] \leftarrow rs2$
Shift Left Logical Immediate	*slli rd, rs1, imm*	$rd \leftarrow ux(rs1) << sx(imm)$
Shift Right Logical Immediate	*srli rd, rs1, imm*	$rd \leftarrow ux(rs1) >> sx(imm)$
Shift Right Arithmetic Immediate	*srai rd, rs1, imm*	$rd \leftarrow sx(rs1) >> sx(imm)$
Add Immediate Word	*addiw rd, rs1, imm*	$rd \leftarrow s32(rs1) + imm$
Shift Left Logical Immediate Word	*slliw rd, rs1, imm*	$rd \leftarrow s32(u32(rs1) << imm)$
Shift Right Logical Immediate Word	*srliw rd, rs1, imm*	$rd \leftarrow s32(u32(rs1) >> imm)$
Shift Right Arithmetic Immediate Word	*sraiw rd, rs1, imm*	$rd \leftarrow s32(rs1) >> imm$
Add Word	*addw rd, rs1, rs2*	$rd \leftarrow s32(rs1) + s32(rs2)$
Subtract Word	*subw rd, rs1, rs2*	$rd \leftarrow s32(rs1) - s32(rs2)$
Shift Left Logical Word	*sllw rd, rs1, rs2*	$rd \leftarrow s32(u32(rs1) << rs2)$
Shift Right Logical Word	*srlw rd, rs1, rs2*	$rd \leftarrow s32(u32(rs1) >> rs2)$
Shift Right Arithmetic Word	*sraw rd, rs1, rs2*	$rd \leftarrow s32(rs1) >> rs2$

End of table.

5.1.6 Registers

None.

Chapter 6

Peripherals

6.1 GPIO

6.1.1 System Integration of GPIO0

History

Regulations for author history:

- Only visible internally (by conditioning of the table, no other conditions needed)

Table 6.1: Authors

Author	Department	From
Andreas Siggelkow	Fac. E	February 2025

Regulations for history table:

- Last version on top.
- Dont use links in the history table. These may cause unexpected history changes in future versions.
- Name (or short sign) should be visible only internally.
- Create a new version after each (also limited) release of the document. Versioning does not depend on the overall document version.
- Use the form x.y for the version. x for the master versions, y for the project specific changes and updates.

Table 6.2: History

Date	Name	Content Changes
Version of this document: V0.1		
2025-02-13	Andreas Siggelkow	First draft (AS)

Functional Configuration of GPIO0

This section should describe the product specific instantiation of this module's feature set (which may be a subset of the full feature set). It is intended also for use in the product overview section of the target spec/summary target spec.

The standard features of a GPOI peripheral are described in its functional description in Section 6.1.3. Generic features, that can be decided for each specific instantiation of the peripheral are listed and completed for GPIO0 in Table 5.3.

Table 6.3: GPIO0 Feature Set

	Generic GPIO Feature Set	Details
yes	Seperate Kernel Clock	125 MHz
no	FIFO Data Buffering	not yet

HW Interfaces

The following table 6.4 lists all hardware interfaces which are relevant for the regular operation of the device. It is not a detailled list of all signals and does not include implementation specific informations.

The names used below show the functional connectivity. There is no guarantee that they are electrically compatible (e.g. voltage levels, clock domains etc. may be different). A full list is part of the design spec.

Table 6.4: HW Interfaces of Module GPIO0

	Module internal name	Chip level name
Supply		
Clock	clk_i	clk_i
Bus	System Bus	Wishbone
Interrupt	None	-
DMA	None	-
External Signals	gpio_o gpioAdr_o cs_o	asGpio_s(7:0) asGpioAdr_s(3:0) asGpioCs_s

Bus Interface

Wishbone Slave BPI.

Registers

Table 6.5: Register List of Module GPIO0

Register Name(s) in Chip	Name used in module register description	Comment Comment
GPIO0_ID	ID	ID of GPIO0 peripheral

Related Sections of this Spec

- GPIO functional description starting with Section 6.1.3.
- Interrupt and DMA Concept Section “Architecture Concepts”
- Peripheral Architecture Concept Section “Architecture Concepts”
  - Slave AHB BPI Section “Architecture Concepts”
  - Peripheral ID Registers Section “Architecture Concepts”
  - Peripheral Clocking Concept Section “Architecture Concepts”
  - Clock Control Registers Section “Architecture Concepts”
  - Horizontal Interrupt and DMA Register Structure Section “Architecture Concepts”
  - Basic Interrupt and DMA Register Set Section “Architecture Concepts”
  - Interrupt Source Combining Register Set Section “Architecture Concepts”
  - Synchronization Block Section “Architecture Concepts”
  - Standardized Flip-Flop based FIFO Section “Architecture Concepts”
  - TX-FIFO Registers Section “Architecture Concepts”
  - RX-FIFO Registers Section “Architecture Concepts”

6.1.2 History of GPIO

Table 6.6: History

History		
Target Spec.	Current version : 1.0, 2025-02-13 Previous version : 0.0, 2020-10-12	Author: A. Siggelkow
Paragraph (in previous version)	Paragraph (in current version) / date	Subjects (major changes since last revision)
Current Chapter Version: C0.1; Date: 2025-02-13		
6.1	6.1 / 2025-02-13	First draft (AS)

6.1.3 Functional Overview

The GPIO peripheral allows to drive 8 bit data from the Wishbone data bus to GPIO pins asGpio_s(3:0) (output only). The address asGpioAdr_s(3:0) allows to decode up to 15 of such GPIO blocks (external decoder required). The address 0 is used for the GPIO ID register. A chip select (cs) is used to activate the GPIO output (dummy citation [4]).

General Features:

- Chip select
- FIFO data buffering (future)
- I/O (future)
- Interrupt (future)

Synchronous Mode:

- 8 data bits

Asynchronous Mode:

- -

6.1.4 Structural Overview

The GPIO is a Wishbone slave (asSlaveBPI) peripheral. This peripheral architecture aims for high bus performance by dividing peripherals into two clock domains. Figure 6.1 shows a simplified block diagram of the GPIO asSlaveBPI peripheral.

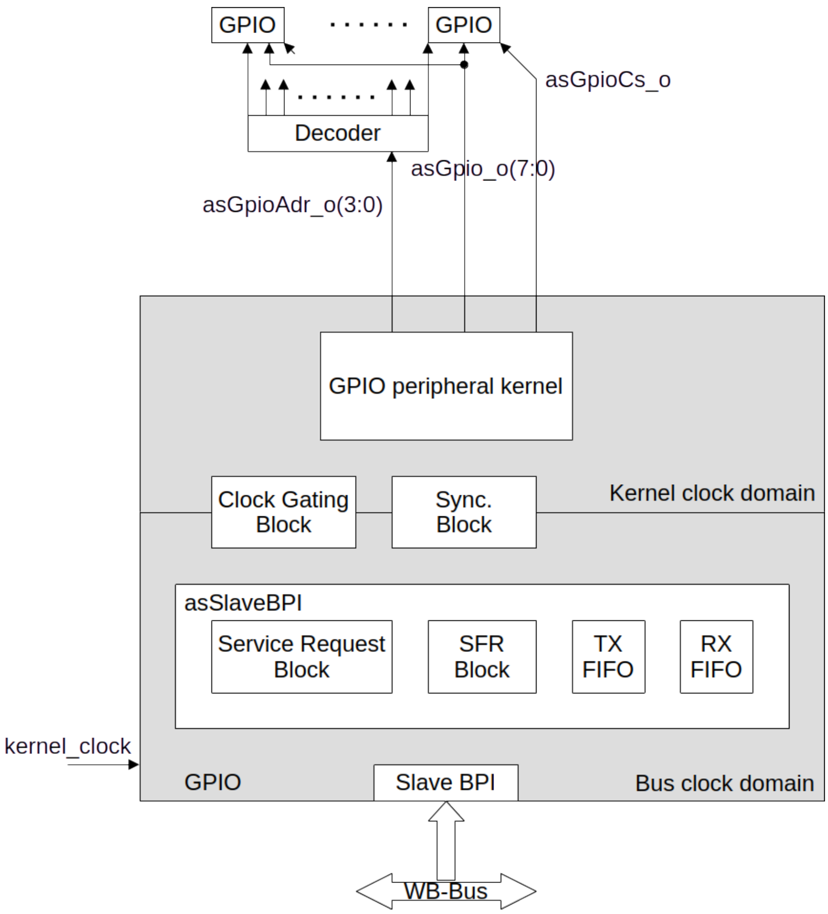

Figure 6.1: GPIO asSlaveBPI Peripheral

GPIO Peripheral Kernel: The GPIO kernel provides the specific functionality of the GPIO interface, that is described in Section 6.1.6. The peripheral kernel has its own kernel clock domain.

Bus Peripheral Interface BPI: The BPI is responsible for the interconnection of the GPIO registers with an on-chip bus. Also the peripheral clock gating is controlled by the BPI.

asSlaveBPI Block: The asSlaveBPI block lies within the bus clock domain and contains the following subblocks:

Special Function Register Block The Special Function Register Block contains all FIFO specific registers and all special function registers (SFR) of the peripheral kernel that are accessible by the SW via the system bus.

Service Request Block SRB The SRB is used to prepare the interrupt and data transfer requests for the Interrupt Control Unit (ICU) and the DMA Controller (DMAC), respectively.

TX FIFO The TX FIFO is used to buffer the transmission data from the bus, in order to adapt the character processing speed of the peripheral kernel to the transfer rate of the system bus.

RX FIFO The RX FIFO is used to buffer the received data from the kernel, in order to adapt the character processing speed of the peripheral kernel to the transfer rate of the bus system.

Clock Gating Block: The Clock Gating Block is used to derive the necessary clocks for the several blocks within the peripheral from the clocks provided by the system.

Synchronization Block: The Synchronization Block is used to synchronize the signals between the bus clock and kernel clock domains.

I/O of the GPIO Kernel

Table 6.7 shows the input and outputs of the GPIO kernel.

Table 6.7: GPIO I/O

Pin	Direction	Width	Explanation
clk_i	in	1	System clock
rst_i	in	1	System reset. Active high.
addr_i(3:0)	in	4	From GPIO-BPI. Address for external GPIO devices.
data_i(7:0)	in	8	From GPIO-BPI. GPIO outputs for one block.
en_i	in	1	From GPIO-BPI. Enables a write to a GPIO block.
gpio_o	out	8	To output pins. GPIO outputs for one block, delayed by one clock and enabled by en_i.
gpioAdr_o	out	4	To output pins. Address for external GPIO devices, delayed by one clock and enabled by en_i.
cs_o	out	1	To output pins. Chip select for outside GPIO blocks.

I/O of the GPIO BPI

Table 6.8 shows the input and outputs of the GPIO BPI.

Table 6.8: GPIO BPI

Pin	Direction	Width	Explanation
clk_i	in	1	System clock
rst_i	in	1	System reset. Active high.
addr_o(3:0)	out	4	To GPIO kernel. Address for external GPIO devices. addr_o = addr_i.
dat_from_core_i(7:0)	in	8	From GPIO kernel. Not used here, forced to zero.
dat_to_core_o(7:0)	out	8	To GPIO kernel. GPIO outputs for one block. dat_to_core_o = dat_i
wr_o	out	1	To GPIO kernel. Enable for GPIO data and address, and chip select for outside GPIO blocks. wr_o = we_i AND stb_i.
addr_i(3:0)	in	4	From WB-Bus. Peripherals addresses.
dat_i(63:0)	in	64	From WB-Bus. Peripherals data input.
dat_o(63:0)	out	64	To WB-Bus. Peripherals data output. For addr_i = 0 the GPIO ID register, else dat_from_core_i
we_i	in	1	From WB-Bus. Peripherals write enable.
sel_i(7:0)	in	8	From WB-Bus. Peripherals byte select. Not used here.
stb_i	in	1	From WB-Bus. Valid bus cycle. Combined (AND) with we_i for wr_o.
ack_o	out	1	To WB-Bus. Error free bus transaction. Here, ack_o = stb_i.
cyc_i	in	1	From WB-Bus. High for complete bus cycle. Not used here.

Internals of the GPIO BPI

Table 6.9 shows the internal signals and processes of the GPIO BPI.

Table 6.9: GPIO BPI Internals

Object	Type	Width	Explanation
id_reg_s	logic	64	GPIO ID register. ID set on reset. Read on address = 0. Overwritable on address = 0 and strobe and we.
dat_o	comb	1	Data output selector: • address = 0: GPIO ID • else: data from kernel (nothing here)

I/O of the GPIO Top

Table 6.10 shows the input and outputs of the GPIO Top.

Table 6.10: GPIO Top

Pin	Direction	Width	Explanation
clk_i	in	1	System clock
rst_i	in	1	System reset. Active high.
wbdAddr_i(3:0)	in	4	From WB-Bus. Peripherals addresses.
wbdDat_i(63:0)	in	64	From WB-Bus. Peripherals data input.
wbdDat_o(63:0)	out	64	To WB-Bus. Peripherals data output.
wbdWe_i	in	1	From WB-Bus. Peripherals write enable.
wbdSel_i(7:0)	in	8	From WB-Bus. Peripherals byte select.
wbdStb_i	in	1	From WB-Bus. Valid bus cycle.
wbdAck_o	out	1	To WB-Bus. Error free bus transaction.
wbdCyc_i	in	1	From WB-Bus. High for complete bus cycle.
gpio_o(7:0)	out	8	To chip pins. GPIO outputs for one block.
gpioAdr_(3:0)	out	4	To chip pins. Address for external GPIO devices.
cs_o	out	1	To chip pins. Chip select for outside GPIO blocks.

6.1.5 Service Requests of the GPIO

No service requests in the GPIO.

6.1.6 The GPIO Peripheral Kernel

As common for asSlaveBPI peripherals there are also two main states of the GPIO's kernel.

Before it can actually start operation, it has to be configured. This configuration state is left by setting bit RUN in register RUN_CTRL. Thereby the configuration registers are locked for write accesses, the functional blocks of peripheral kernel and asSlaveBPI block are reset (e.g. the FIFOs are flushed) and working state is entered. The different functional blocks of the GPIO peripheral kernel start operating as described in the following.

6.1.7 Registers

In the following the registers of a GPIO peripheral are listed and described. Some registers' layout or availability depends on the generic feature set implemented for that GPIO's instantiation. Such registers and concerned bit fields are denoted by the use of italic style.

GPIO Register Overview

Table 6.11: GPIO Register List

Register Group	Register Name	Register Symbol
System Registers	Clock Control Register	**CLC**
	Identification Register	**ID**
Special Function Registers	Run Control Register	**RUN_CTRL**
	Modemstatus Set Register	**MSS_SET**
	Modemstatus Clear Register	**MSS_CLR**

Read-Write Features

- w: writable bit, by SW
- r: readable bit, by SW
- h: bit will be set by HW only

GPIO_ID -: Identification register of this peripheral. A write will have no effect, a read will return the fixed ID. The ID will be defined on chip level.

Clock control register (in BPI) [Reset value: 00_H]

7	6	5	4	3	2	1	0
'-'	clk_ sel	'-'	'-'	kernel_ clk_ disable	'-'	'-'	bus_ clk_ disable
'-'	rw	'-'	'-'	rw	'-'	'-'	rw

ID register (in BPI) [Reset value: $DEADBEEFDEADBEEF_H$]

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48
id 63															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

47	46	45	44	43	42	41	41	39	38	37	36	35	34	33	32
id 47															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
id 31															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
id 15															
id 15															

Field	**Bits**	**Type**	**Description**
reserved	7:5,2:1	rw	not implemented, a write has no effect, a read returns a '0'
clk_sel	6	rw	Selects the clock source for the UART kernel: • '1': kernel_clk_i (kernel clock) drives the functional part • '0': bus_clk_i (bus clock) drives the functional part
kernel_clk_disable	3	rw	disables the kernel clock for the block.
bus_clk_disable	0	rw	disables the bus clock for the block.

Chapter 7

Test and Debug

Not yet

Chapter 8

Electrical Characteristics

Not yet

Chapter 9

Memory Map and Registers

Not yet

Bibliography

[1] The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2, https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf, Accessed: 2024-02-23, 2017.

[2] RISC-V Guide, https://marks.page/riscv/, Accessed: 2024-02-23, 2024.

[3] Five EmbedDev, https://five-embeddev.com/riscv-isa-manual/latest/instr-table.html, Accessed: 2024-02-23, 2024.

[4] G. Lehmann, B. Wunder, and M. Selz, Schaltungsdesign mit VHDL, Synthese, Simulation und Dokume
Poing: Franzis, 1994, ISBN: 978-3772361630. [Online]. Available: https://www.itiv.kit.edu/downloads/Buch_gesamt.pdf.