PicoNut Manual

Release 0.0.1-107-g00d4609

Gundolf Kiefer Michael Schäferling
Marco Milenkovic Lorenz Sommer Lukas Bauer

Oct 11, 2024

PICONUT MANUAL

Document History

Version	Date	Description
0.0.1	2024-04-03	Initial version (how to document, welcome page)

Introduction

Welcome to the PicoNut project! The PicoNut processor is a RISC-V processor being developed by the Efficient Embedded Systems Group (EESG) at the technical university of applied sciences Augsburg. The main goal of this project in its first phase is to provide a minimal RISC-V compatible architecture that is well documented and built to be expanded upon. Future prospects include the ability to run the Linux operating system and to support the RISC-V matrix extension to facilitate AI applications.

CHAPTER ONE

THE PICONUT

The PicoNut project at the Augsburg University of Technology aims to develop a minimal and at the same time flexibly expandable RISC-V processor that works on common FPGA hardware and provides a complete simulator. The processor is intended to be used in teaching and research, in order to examine different computer architectures, the interaction between hardware and operating systems (Linux, FreeRTOS) and the integration of hardware accelerators, for example for AI applications.

PicoNut …

- … is a minimal, yet extendable RISC-V processor as an open source project.
- … runs on inexpensive FPGA boards, e.g. OrangeCrab or ULX03S.
- … is expandable with memory protection/MMU (Linux), AI acceleration, various RISC-V extensions.
- The hardware is modeled in SystemC (C++) in order to be able to build a powerful simulator from the same source code.
- Good software support is provided by RISC-V compatibility: GNU toolchain (GCC/GDB), newlib, FreeRTOS, Linux.
- Solid project management includes automated testing, project website and CI/CD techniques.

The project started at the beginning of 2024 and currently involves several enthusiastic students who are working on project work or theses.

CHAPTER

TWO

THE PICONUT PROJECT

The PicoNut processor is an implementation of the RISC-V instruction set architecture (ISA) written in SystemC using the ICSC (Intel Compiler for SystemC) and YOSYS toolchains for FPGA deployment. The main motivation of the project as a whole is to provide an expandable and approachable platform for developers and users alike. For this purpose, the entire processor can be simulated using the powerful simulation capabilities of the SystemC C++ library. The modular nature of the PicoNut allows for an easy exchange of individual submodules for rapid prototyping.

With the initial release of this project the processor supports the RV32I subset of the RISC-V ISA and features a fully fledged simulation environment capable of running any C programs compiled using the standard gcc RISC-V toolchain. Future plans include implementing RISC-V extensions such as the A,F,M and V extensions with the aim of running the Linux operating system as well as enabling AI applications.

The following sections will give a short overview of the system architecture, hardware, software and simulation aspects of the project in its current iteration.

2.1 System architecture

The PicoNut processor itself is part of a larger system built around it. The processor core, the “memory unit” (MemU), a Wishbone bus and any kind of memory make up the required minimal system.

Memory is to be byte addressable and data is to be interpreted in LSB order by all components.

2.1.1 Processor Core

The processor core is the main component of the system. It is responsible for executing instructions, accessing memory and other devices via the memory unit in the process. Communication between the processor core and the memory unit is done via the *IPort/DPort interfaces*.

2.1.2 Memory Unit

The memory unit serves as the processor cores interface to the wishbone bus. It handles all read and write requests taken by the processor core and forwards them to the wishbone bus. It does not equate to a memory management unit (MMU) at the time of release. Currently its primary function is address space separation to allow various peripherals connected to the wishbone bus to be addressed by the processor core. Virtual memory, paging and cache functionality are future prospects.

2.1.3 Main system bus

The main system bus is an implementation of the Wishbone bus protocol. It connects the memory unit (which is connected to the processor core), the main memory and all peripherals.

2.2 Software and Simulation

To facilitate testing, debugging and rapid prototyping, the project includes a comprehensive simulator. It allows the user to provide the system with a gcc compiled program which is then run. Beyond allowing for standard output print statements, a `.vcd` trace file containing all processor core signal states, a core trace and a memory dump are generated alongside. To enable rapid integration of new peripherals in software only a simulation only software peripheral interface class is provided. The peripheral interface class "CSoftPeripheral" allows software models to interface with hardware models in simulation. This approach unlocks the speed, resources and features of the C++ language for the verification of hardware modules during the prototyping process, especially when potentially complex modules are required for the DUT to function.

TODO: Link zu Marcos csoft

CHAPTER

THREE

THE MINIMAL NUCLEUS

nucleus submodules

This chapter gives an overview of the minimal nucleus' capabilities and details its submodules.

3.1 Overview

The main purpose of the minimal nucleus is to provide a well documented and minimal implementation of the RV32I RISC-V subset.

3.1.1 Capabilities

The minimal nucleus largely supports running of programs compiled by the RISC-V GCC toolchain. CSR instructuions are unsupported with this initial implementation. The nucleus is capable of running programs that do not rely on external peripherals or interrupts. It does not contain any pipelines and does not support multi-nucleus architectures.

3.1.2 Memory

The minimal nucleus is intended for use with byte-addressable memory. It interfaces with a so called "memory unit" to access memory. It does not access memory directly by itself. Communication with the memory unit is split into two ports, the instruction port for instruction fetching, and the data port for `load` and `store` operations. For these ports, a *custom communication protocol* has been created. The memory unit is connected to the internal system bus and acts as a wishbone interface for the cores requests.

3.1.3 Working principle

The minimal nucleus utilizes a simple fetch-decode-execute workflow. The `decode` step is always executed within one clock cycle. All instructions that are not `load` or `store` instructions, also execute within one clock cycle. The `fetch` step is a IPort memory transaction and takes 3 clock cycles to complete. `load` operations require 4 clock cycles, store operations require 3. For the time being, the ECALL and EBREAK operations halt the processor. The FENCE operation is implemented as a NOP operation.

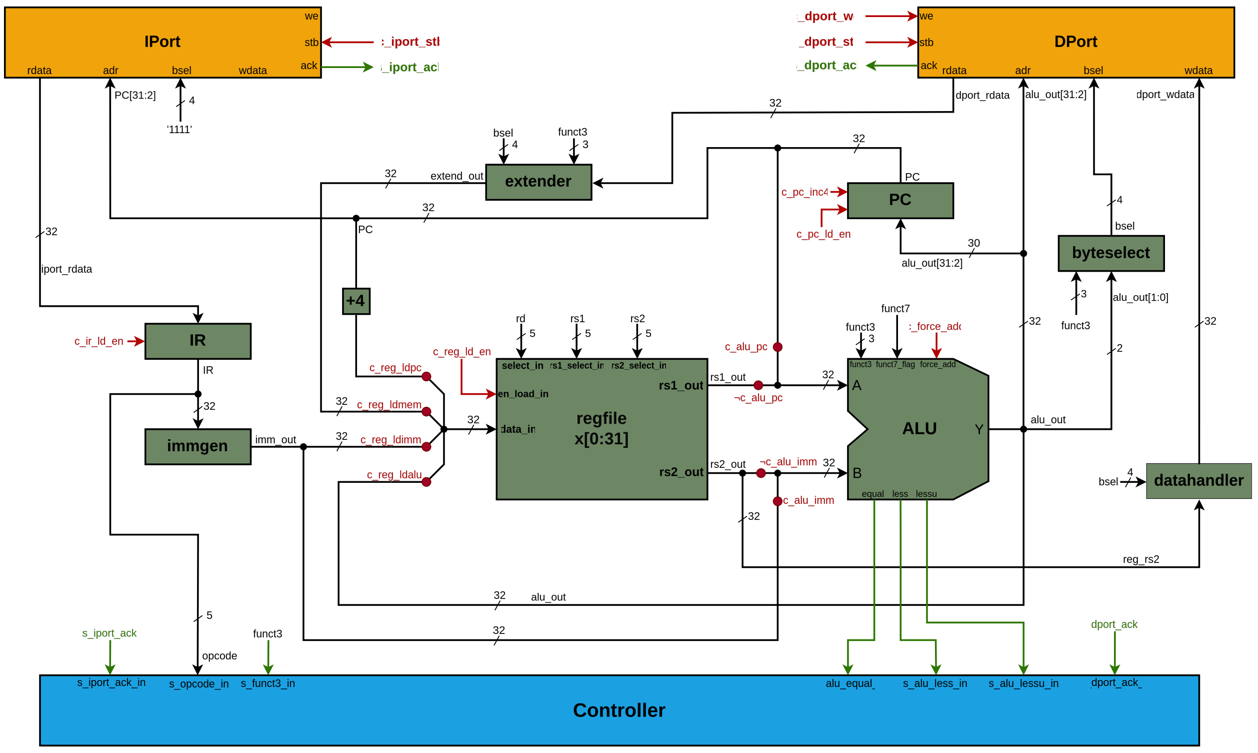


Fig. 3.1: Minimal nucleus block diagram

3.2 Arithmetic Logic Unit (ALU)

The ALU (Arithmetic Logic Unit) performs arithmetic and logical operations on the two 32-bit operands A and B. It is implemented using combinatorics and is not clock synchronous.

Table 3.1: ALU input ports

Port name	Width in bits	Description
A	32	Operand A.
B	32	Operand B.
select_op Derived from IR[14:12].	3	3-bit control signal that selects the operation to be performed.
funct7_flag Derived from IR[30].	1	Decides between ADD/SUB, SRA/SRL operations.
force_add	1	If set, forces the ALU to add the operands.

Table 3.2: ALU output ports

Port name	Width in bits	Description
Y	32	Result of the logical operation performed on operands A and B.
equal	1	Status signal, 1 if a == b, else 0.
less	1	Status signal, 1 if a < b, else 0.
lessu	1	Status signal, 1 if a < b, else 0 (unsigned).

Given the input signals `funct3` and `funct7_flag`, the ALU performs the following operations on the operands A and B.

Table 3.3: ALU operations

Operation	Shorthand	funct3	funct7	Description
ADD	y = a + b	0x0	0	Regular addition
SUB R-Type only.	y = a - b	0x0	1	Two's complement of B is added to A.
AND	y = a & b	0x7	0	Bitwise AND
OR	y = a | b	0x6	0	Bitwise OR
XOR	y = a ^ b	0x4	0	Bitwise XOR
SLL	y = a << b	0x1	0	Shift left logical
SRL	y = a >> b	0x5	0	Shift right logical
SRA	y = a >> b	0x5	1	Shift right arithmetic (sign extends)
SLT	y = a < b ? 1,0	0x2	0	Set less than
SLTU	y = a < b ? 1,0	0x3	0	Set less than unsigned (zero extends)

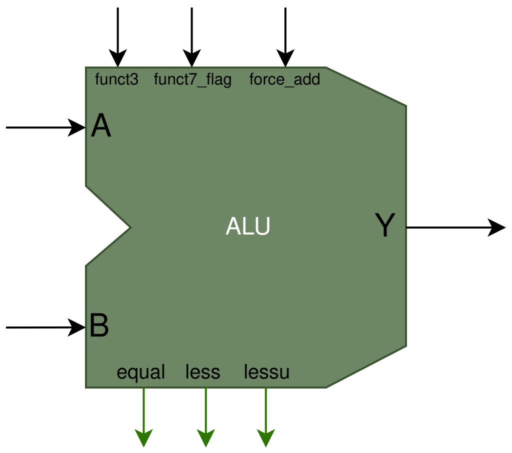


Fig. 3.2: ALU module symbol

The ALU can handle R- and I-Type instructions. In order to correctly implement the few differences the two instruction types have, a few distinctions have to be made. This concerns the fact, that there is no I-Type `subi` instruction. As seen in *Table 3*, `funct7` decides whether subtraction is performed instead of addition and whether to shift logically or arithmetically. According to the RISC-V specification the only relevant values of the `funct7` block are `0x0` and `0x2`. This allows for the whole block to be reduced to its 6th bit, which equals the 30th bit of the whole instruction and thus `IR[30]`. Additionally, even though I-Type instructions get decoded into their internal immediate value, the position of the `funct7` block remains the same.

Because there is no specified `subi` instruction in the specification, a simple forwarding of `IR[30]` to the ALU input `funct7_flag` would lead to erroneous results. This leads to a necessary separation of the I-Type instructions into the ones that care about the `funct7` block and the ones that do not. More specifically, the I-Type shift operations must be separated. For this purpose the controller outputs a control signal `c_funct7_flag`. It decides whether the `funct7_flag` input of the ALU is set to `0x0` or the bit `IR[30]`, depending on the `funct3` block.

Table 3.4: funct7_flag truth table

Instruction type	funct3	funct7_flag (ALU input)	c_funct7_flag (Control signal)
R-Type	0x0	IR[30]	0x1
R-Type	0x5	IR[30]	0x1
I-Type	0x0	0x0	0x0
I-Type	0x5	IR[30]	0x1

Note: This decision logic is implemented in the top level module nucleus.cpp

```
if (signal_c_funct7_flag == 0x1)
{
    signal_funct7 = signal_ir_out.read().range(30, 30);
}
else
{
    signal_funct7 = 0x0;
}
```

signal_funct7 is then forwarded to the ALU input funct7_flag.

Next to R- and I-Type instructions, the ALU is also indirectly used to perform address calculations for the load, store, branch and jump instructions. It also performs the additions necessary for the LUI and AUIPC instructions. All of the previously named instructions use the force_add flag to force the ALU to perform addition. Depending on the instruction, the operands are selected accordingly by the control unit (controller). Possible sources for operand A are rs1 and pc_out (current program counter). Possible operands for operand B are rs2 and imm_out (immediate value).

3.3 Regfile

The regfile component of the nucleus is used to store temporary values that are needed for calculations and program flow. It contains 32 32-bit registers.

Table 3.5: Regfile input ports

Port name	Signal width	Description
data_in	32	Data input
select_in	5	Selects the register in which input data is stored
rs1_select_in	5	Selects which registers value gets output to output port rs1_out
rs2_select_in	5	Selects which registers value gets output to output port rs2_out
en_load_in	1	Control signal to enable/disable storing of input data with the next rising edge.

Table 3.6: Regfile output ports

Port name	Signal width	Description
rs1_out	32	Output of the register selected by rs1_select_in
rs2_out	32	Output of the register selected by rs2_select_in

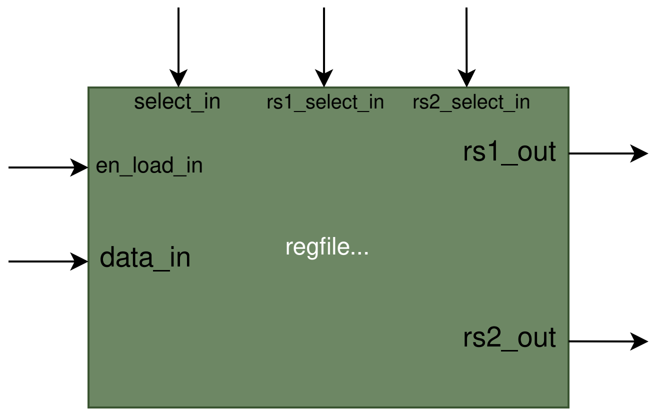


Fig. 3.3: Regfile module symbol

3.4 Program Counter (PC)

The program counter module is used to store the address of the current instruction at any given time. Its content can be thought of as the current position within the program. The program counter is incremented by `0x4` with the next rising edge, once the controller sets the `s_pc_inc4` control signal. Its internal value is be overwritten with the value at its input port with the next rising edge if `s_pc_ld_en` is set. This is used to perform jump and branch instrucitons.

Table 3.7: Program counter input ports

Port name	Signal width	Description
pc_in	30	Data input for when the PC is to be manipulated by jump and branch instructions.
inc_in	1	Control signal. When set, the PC is incremented by `0x4` at the next rising edge.
en_load_in	1	Control signal. When set, the internal register takes on the value present at the
`pc_in` port with the next rising edge.		

Note: Program counter internal register The internal register of the program counter is **30 bits wide**. The lowest two bits of any given address pertaining to the main program in memory can be omitted because every instruction is naturally word aligned and 4 bytes in size. The input port of the program counter is also 30 bits wide for this reason. To maintain a level of consistency for the program counters interactions with other modules and the instruction port interface, the output is 32 bits wide again. The lowest two bits are simply set to a constant `0`. Instead of every connected module having to append these two zeroes at their inputs, it is done once at the PCs output.

Table 3.8: Program counter output ports

Port name	Signal width	Description
pc_out	32	Constant output of the internal register.

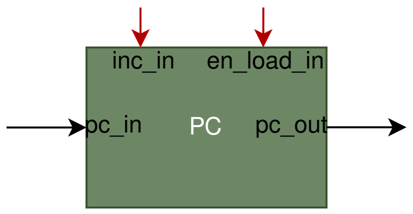


Fig. 3.4: Program counter module symbol

3.5 Instruction Register (IR)

The instruction register module stores the current instruction in its internal register and provides its content to the rest of the nucleus via a constant output.

Table 3.9: Instruction register input ports

Port name	Signal width	Description
ir_in	32	Input data. Internal register takes on this value at the next rising edge.
en_load_in	1	Control signal. When set, the internal register takes on the value present at the
`ir_in` port with the next rising edge.		

Table 3.10: Instruction register output ports

Port name	Signal width	Description
ir_out	32	Constant output of the internal register.

The IR input port `data_in` is connected to the IPort `rdata` signal.

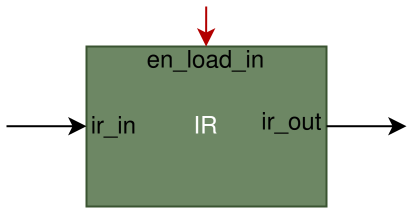


Fig. 3.5: Instruction register module symbol

3.6 Immediate Generator (immgen)

The immediate generator (immgen) module decodes the current instruction and generates an immediate value from it. All but the R-Type instruction type carry an immediate value. This immediate value is a simple constant built into an instruction.

Table 3.11: Immediate generator input ports

Port name	Signal width	Description
data_in	32	Instruction word from which an immediate value is to be generated from.

Table 3.12: Immediate generator output ports

Port name	Signal width	Description
imm_out	32	Immediate value generated from an instruction word.

Immediate values are decoded differently and serve a different purpose depending on the instruction and instruction type. Decoding and generation are executed in accordance with the RISC-V specification page 44, section "Immediate Encoding Variants.

Table 3.13: Immediate encoding variants

Instruction Type	Resulting immediate value
I-Type	`(inst[31])[31:11] + inst[30:25] + inst[24:21] + inst[20]`
S-Type	`(inst[31])[31:11] + inst[30:25] + inst[11:8] + inst[7]`
B-Type	`(inst[31])[31:12] + inst[7] + inst[30:25] + inst[11:8] + 0`
U-Type	`inst[31] + inst[30:20] + inst[19:12] + 0[11:0]`
J-Type	`(inst[31])[31:20] + inst[19:12] + inst[20] + inst[30:25] + inst[24:21] + 0`

How to read the immediate encoding table

In the table above, `inst` stands for the current instruction word from which an immediate value is to be generated from. The operator + in this case stands for concatenation of bits/bit-ranges. Additionally, syntax like `(inst[31])[31:12]` resolves as: "fill the range `[31:12]` in the output value with `inst[31]`".

The immediate generator module detects the correct decoding variant by evaluating the `opcode` field of the current instruction. Groups of instructions in the RV32I subset (load/store/immediate/branch/jump/upper immediate) do not always use the same instruction type format across the board.

Table 3.14: Instructions by instruction type

Instruction type	Instructions
I-Type	All "immediate arithmetic" instructions, all `load` instructions, **`jalr`**
S-Type	All `store` instructions
B-Type	All `branch` instructions
U-Type	`lui`, `auipc`
J-Type	`jal`

Note: The R-Type instruction format is not listed, since it does not contain an immediate value.

data_in
Immgen
imm_out

Fig. 3.6: Immediate generator module symbol

3.7 Byteselector

The byteselector module generates the `bsel` signal. This signal is forwarded to the data port interface as well as modules that handle memory access. It is necessary for the implementation of byte and halfword load/ store commands.

Table 3.15: Byteselector input ports

Port name	Signal width	Description
adr_in	2	Lowest two bits of the `alu_out` signal which represents the alignment of a calculated address.
funct3_in	3	`funct3` block of the current instruction.

Table 3.16: Byteselector output ports

Port name	Signal width	Description
byteselect_out	4	Outgoing byteselect signal.

Because memory accesses always use a full 32-bit word aligned address (lowest two bits are `0`) it is necessary to generate an additional signal to allow for loading/storing of individual bytes and halfwords. This `bsel` signal signifies which of the 4 bytes of a 32-bit word at a given address are to be loaded from or stored to. The `bsel` signal is 4 bits wide and each bit represents a single byte of a 32-bit word.

When an address is calculated by the ALU, only the upper 30 bits are forwarded to the data port interface. Memory accesses must always be word aligned and thus the lowest two bits of any address must always be `00`. This is because the main memory is byte-addressable and therefore 32-bit words sit in memory in intervals of `0x4`, leading to the lowest to bits always being `00`.For the purpose of generating the `bsel` signal, they - along with the `funct3` block of the current instruction - lead to the following `bsel` signal assignments.

Table 3.17: Byteselect generation table

adr_in	funct3_in	bsel_out
`00`	lb/lbu/sb	`0001`
`01`	lb/lbu/sb	`0010`
`10`	lb/lbu/sb	`0100`
`11`	lb/lbu/sb	`1000`
`00`	lh/lhu/sh	`0011`
`10`	lh/lhu/sh	`1100`
`01`	lh/lhu/sh	invalid
`11`	lh/lhu/sh	invalid
`--`	lw	`1111`

Note: The nucleus currently only supports fully 32-bit aligned loads and stores. It is not possible to load/store data beyond the 32-bit boundary of any address. Additionally, disjointed loads are also not supported. (for example `bsel 0101`).

3.7.1 Examples

To further illustrate the interaction between the effective address and the `bsel` signal, consider the following examples:

Assume the following memory layout.

Address	Value
`0x00000000`	`0x12 0x34 0x56 0x78`
`0x00000004`	`0xCA 0xFE 0xBA 0xBE`

Example 1

1. Let the registers `x1` and `x2` contain the value `0`.
2. Let the current instruction be `lb x1, 1(x2)`.
3. This reads as "load the value of the byte at address `0x00000000` offset by `0x1` into `x1`.
4. This leads to an effective target address of `0x00000001`.
5. The targeted byte is therefore `0x34`.
6. The address at the dport interface wil be `0x00000000`. (Lowest two bits always `00`!)
7. The byteselector will evaluate the `funct3` block (here: `0x0`) and the lowest two bits of the effective address (here: `0x1`), resulting in a `bsel` value of `0010`.
8. After the memory access transaction is complete, the register `x1` will contain the value `0x00000034`.

Example 2

1. Let the register `x1` contain the value `0x0` and register `x2` contain the value `0x00000004`.
2. Let the current instruction be `lh x1, 2(x2)`.
3. This reads as "load the value of the halfword at address `0x00000004` offset by `0x2` into `x1`.
4. This leads to an effective address of `0x00000006`.
5. The targeted halfword is therefore `0xBABE`.
6. The address at the dport interface wil be `0x00000004`. (Lowest two bits always `00`!)
7. The byteselector will evaluate the `funct3` block (here: `0x1`) and the lowest two bits of the effective address (here: `0x2`), resulting in a `bsel` value of `1100`.
8. After the memory transaction is complete. the register `x1` will contain the value `0x0000BABE`.

Note: The examples above only demonstrate the generation and function of the `bsel` signal. A full memory transaction involves additional modules.

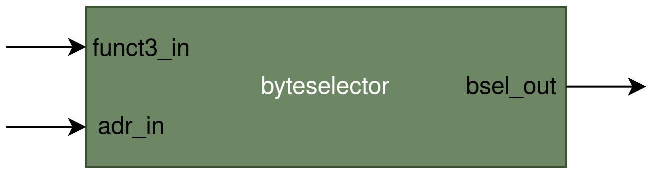


Fig. 3.7: Byteselector module symbol

3.8 Extender

The extender module prepares **incoming** data requested by the nucleus via load instructions. It ensures relevant data is in the correct position within the data word and sign extends it, if needed.

Table 3.18: Extender input ports

Port name	Signal width	Description
data_in	32	Incoming data word to be processed
funct3_in	3	funct3 block of the current instruction
bsel_in	4	Byteselect signal generated by the byteselector.

Table 3.19: Extender output ports

Port name	Signal width	Description
extend_out	32	Processed data word

When the nucleus requests data from memory via a load instruction, the memory unit will always provide a 32-bit data word. If the load instruction requests a full 32-bit word (lw instruction) no further steps have to be taken and the full data word is stored in the target internal register. This changes, when the nucleus requests to load a byte or halfword via the lb or lh instructions respectively. The memory unit will still provide a 32-bit word with the relevant data occupying either any of the single 4 bytes within the full word (lb) or two consecutive bytes (lh). The purpose of this module is to resolve this arrangement and to reorient it to form a new 32-bit word that represents this data. For this, the extender module uses the `bsel` signal already provided by the *Byteselector*. The `bsel` signal represents a mask where each of its bits corresponds to a byte in the incoming data word.

Table 3.20: Extender output table

Incoming data	`bsel`	Output
`0x12345678`	`0001`	`0x00000078`
`0x12345678`	`0010`	`0x00000056`
`0x12345678`	`0100`	`0x00000034`
`0x12345678`	`1000`	`0x00000012`
`0x12345678`	`0011`	`0x00005678`
`0x12345678`	`1100`	`0x00001234`

Additionally, the `lb` and `lh` instructions require sign extensions, if need be. For this purpose, the extender module evaluates the `funct3` block of the current instruction, to determine whether to zero extend (`lbu/lhu`) or to sign extend. In the `lb/lh` case, the most significant bit of the already previously rearranged data is evaluated. Should it read `1`, the remaining bits of the whole word are filled with `1`. Should this not apply or the instruction is `lbu` or `lhu`, the word will simply be zero extended.

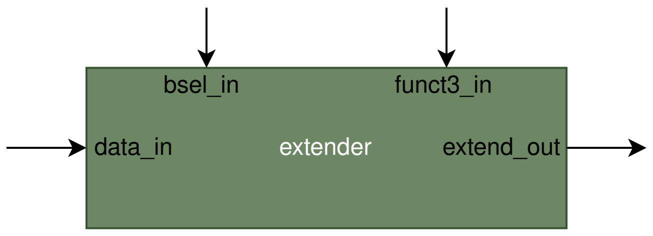


Fig. 3.8: Extender module symbol

3.9 Datahandler

The datahandler module is prepares **outgoing** data to the Dport interface. It ensures data within the outgoing 32-bit word is in the correct position.

Table 3.21: Datahandler input ports

Port name	Signal width	Description
data_in	32	Incoming data word to be processed
bsel_in	4	Byteselect signal generated by the byteselector

Table 3.22: Datahandler output ports

Port name	Signal width	Description
data_out	32	Processed data word

Data stored in any internal register (the regfile) occupies the low bits of the register according to the RISC-V specification. However, since `sb` and `sh` instructions enable the storing of individual bytes and halfwords to a specific, byte or halfword aligned position within any given memory address, it is necessary to rearrange outgoing data accordingly. Essentially, the datahandler module performs the inverse operation of the *extender module*. It uses the `bsel` signal to determine which position within the outgoing data word the incoming data should occupy. Unlike the *extender module*, it does not perform any sign or zero extension.

Table 3.23: Datahandler output table

Incoming data	`bsel`	Output
`0x00000012`	`1000`	`0x12000000`
`0x00000012`	`0100`	`0x00120000`
`0x00000012`	`0010`	`0x00001200`
`0x00000012`	`0001`	`0x00000012`
`0x00001234`	`1100`	`0x12340000`
`0x00001234`	`0011`	`0x00001234`
`0x12345678`	`1111`	`0x12345678`

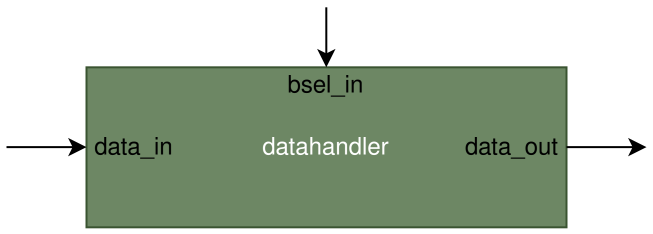


Fig. 3.9: Datahandler module symbol

3.10 Controller

The controller (also known as the control unit) is a finite state machine that sets a number of control signals which control signal flow and toggle functions of other submodules. It reads status signals from various parts of the nucleus and evaluates them at specific decision points to facilitate the overall function of executing instructions by setting control signals accordingly.

Table 3.24: Controller status signals (inputs)

Signal name	Signal width	Description
s_opcode	5	Opcode of the current instruction
s_funct3	3	funct3 block of the current instruction
s_alu_less	1	Status signal from the ALU. 1 if A < B, else 0.
s_alu_lessu	1	Status signal from the ALU. 1 if A < B (unsigned), else 0.
s_alu_equal	1	Status signal from the ALU. 1 if A == B, else 0.
s_dport_ack	1	Acknowledge signal from the data port interface.
s_iport_ack	1	Acknowledge signal from the instruction port interface.

Table 3.25: Controller control signals (outputs)

Signal name	Description
c_iport_stb	Instruction port strobe signal. Begins a transaction.
c_dport_stb	Data port strobe signal. Begins a transaction.
c_dport_we	Data port write enable signal.
c_reg_ld_en	Regfile load enable signal.
c_reg_ldpc	If set, regfile input is program counter.
c_reg_ldmem	If set, regfile input is a value from memory (via Dport).
c_reg_ldimm	If set, regfile input is the current immediate value.
c_reg_ldalu	If set, regfile input is the current ALU output.
c_alu_pc	If set, ALU operand A is the program counter.
c_alu_imm	If set, ALU operand B is the current immediate value.
c_force_add	If set, forces the ALU to perform an addition operation.
c_funct7_flag	Control signal for the ALU. Decides between ADD/SUB, SRA/SRL operations.
c_pc_inc4	If set, the program counter is incremented by `0x4` at the next rising edge.
c_pc_ld_en	If set, the program counter takes on the value present at the `pc_in` port at the next rising edge.
c_ir_ld	If set, the instruction register takes on the value present at the `ir_in` port at the next rising edge.

The controller has 22 internal states which it enters/leaves depending on the current status signals.

Table 3.26: Controller logical states

State name	Description
`RESET`	Initial state. Resets all control signals.
`IPORT_STB`	Instruction port strobe is set high.
`AWAIT_IPORT_ACK`	Await the instruction port acknowledge signal.
`DECODE`	Instruction fetched. Decode the current instruction.
`ALU`	Execute ALU instructions.
`ALU_IMM`	Execute immediate ALU operations (I-Type).
`ALU_IMM_SHIFT`	Execute immediate ALU shift instructions.
`LUI`	Execute upper immediate instructions.
`AUIPC`	Execute add upper immediate to PC instructions.
`NOP`	No operation, PC advances by `0x4`.
`BRANCH`	Calculate branch target address and set PC if condition is met.
`DONT_BRANCH`	Do not branch, PC advances by `0x4`.
`JAL`	Jump and link instruction. Calculate address and set PC.
`JALR`	Jump and link register instruction. Calculate address and set PC.

The load and store procedures (and related states) are subject to optimization and currently focus on on eliminating potential edge cases to ensure proper function.

Table 3.27: Controller load/store states

State name	Description
`LOAD1` Then go to `LOAD2` else remain.	Await data port acknowledge signal is low to ensure no conflict with an ongoing transaction.
`LOAD2` and increment PC. Go to `LOAD3`.	Set data port strobe signal high to begin transaction. Calculate the target address
`LOAD3`	Await data port acknowledge signal. If high, go to `LOAD4`, else remain.
`LOAD4`	Data port acknowledge signal is high. Enable regfile to load data and go to `IPORT_STB`.
`STORE1` Then go to `STORE2` else remain.	Await data port acknowledge signal is low to ensure no conflict with an ongoing transaction.
`STORE2` and increment PC. Set write enable signal high. Go to `STORE3`.	Set data port strobe signal high to begin transaction. Calculate the target address
`STORE3`	Await data port acknowledge signal. If high, go to `IPORT_STB`, else remain.

Note: While the load and store procedures and essentially the same, the load procedure required an additional state to avoid erroneous behavior when the target and source register (that holds the target address) are the same. The specific solution was to delay the signals that allow the regfile to load a new value (`c_reg_ldmem`, `c_reg_ld_en`) and enable them only after the bus transaction is complete and when the data present at the regfile input signal is valid.

There is potential for skipping over states when conditions are met, this however has not been explored with this implementation.

See the following diagram to get an overview of the state machine.

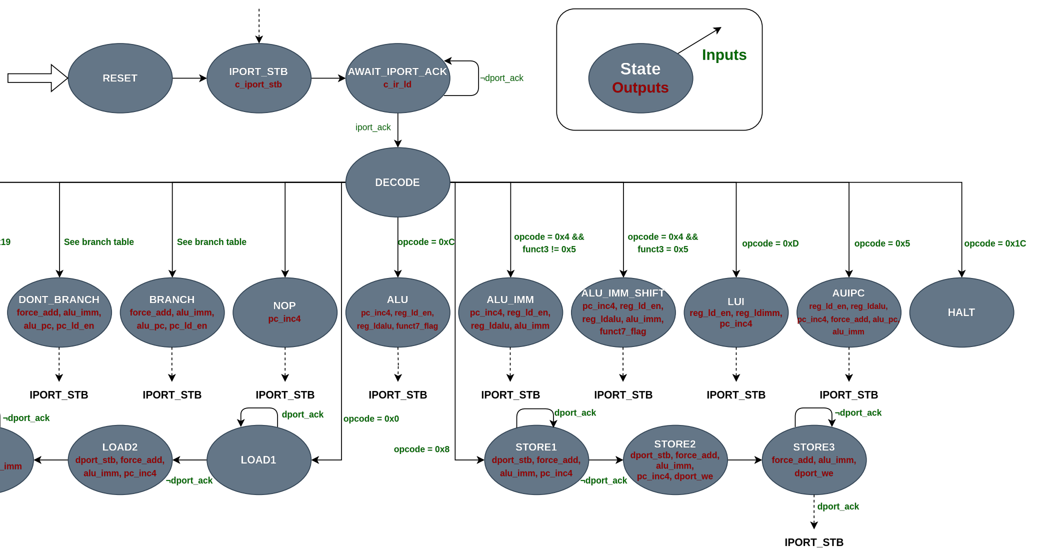

RESET
IPORT_STB
c_iport_stb
AWAIT_IPORT_ACK
c_ir_ld
¬dport_ack
State
Outputs
Inputs
iport_ack
DECODE
opcode = 0x1B
opcode = 0x19
See branch table
See branch table
opcode = 0xC
opcode = 0x4 && funct3 != 0x5
opcode = 0x4 && funct3 = 0x5
opcode = 0xD
opcode = 0x5
opcode = 0x1C
JAL
force_add, alu_imm, alu_pc, pc_ld_en, reg_ldpc, reg_ld_en
JALR
force_add, alu_imm, pc_ld_en, reg_ldpc, reg_ld_en
DONT_BRANCH
force_add, alu_imm, alu_pc, pc_ld_en
BRANCH
force_add, alu_imm, alu_pc, pc_ld_en
NOP
pc_inc4
ALU
pc_inc4, reg_ld_en, reg_ldalu, funct7_flag
ALU_IMM
pc_inc4, reg_ld_en, reg_ldalu, alu_imm
ALU_IMM_SHIFT
pc_inc4, reg_ld_en, reg_ldalu, alu_imm, funct7_flag
LUI
reg_ld_en, reg_ldimm, pc_inc4
AUIPC
reg_ld_en, reg_ldalu, pc_inc4, force_add, alu_pc, alu_imm
HALT
IPORT_STB
opcode = 0x0
opcode = 0x8
LOAD4
reg_ldmem, reg_ld_en, alu_imm, force_add
dport_ack
LOAD3
force_add, alu_imm
¬dport_ack
LOAD2
dport_stb, force_add, alu_imm, pc_inc4
¬dport_ack
LOAD1
dport_ack
STORE1
dport_stb, force_add, alu_imm, pc_inc4
dport_ack
¬dport_ack
STORE2
dport_stb, force_add, alu_imm, pc_inc4, dport_we
STORE3
force_add, alu_imm, dport_we
¬dport_ack
dport_ack

Fig. 3.10: Controller state machine diagram

Because the `BRANCH` and `DONT_BRANCH` transition edges would clutter the diagram unnecessarily, their conditions are detailed in the following table

Note: The table is meant to be read such that if the conditions in the "Conditions" column are met, the controller transitions into the `BRANCH` state, else it transitions into the `DONT_BRANCH` state.

Table 3.28: Branch conditions

Branch Type and opcode	Condition
`BEQ`, `0x0`	`s_alu_equal == 1`
`BNE`, `0x1`	`s_alu_equal == 0`
`BLT`, `0x4`	`s_alu_less == 1`
`BGE`, `0x5`	`s_alu_less == 0`
`BLTU`, `0x6`	`s_alu_lessu == 1`
`BGEU`, `0x7`	`s_alu_lessu == 0`

CHAPTER

FOUR

IPORT/DPORT INTERFACE

4.1 Data Port (DPort)

The data port is an interface between the nucleus and the memory unit. It utilizes a simple bus protocol to transfer data between them.

4.2 Instruction Port

The instruction port, similarly to the data port, is an interface between the nucleus and the memory unit. It is used to transfer instructions read from memory to the CPU. In contrast to the data port, the instruction port is a one-way interface. Instructions only need to be read by the CPU.

4.3 Data port and instruction port signals

The data port/instruction port interface has the following input/output signals.

Name	Driven by	Optional	Width	Description
stb	Nucleus	-	1	Strobe
we	Nucleus	DPort only	1	Write enable
bsel[n]	Nucleus	-	4	Byte Select
adr[n]	Nucleus	-	<=30	Address
wdata[n]	Nucleus	DPort only	32/64/128	Write data
ack	MemU	-	1	Acknowledge
rdata[n]	MemU	-	32/64/128	Read data

4.3.1 Strobe (stb)

The strobe signal begins a bus transaction. In single mode it is to be held high for no longer than a single clock cycle.

4.3.2 Write enable (we)

The write enable signal is exclusive to the data port interface. It signals that the nucleus is writing data to the memory unit. Similarly to the strobe signal, it is to be held high for no longer than a single clock cycle synchronous to the strobe signal at the beginning of a transaction cycle. It must be set by the nucleus to either 0 or 1. A signal vlaue of we=1 and begins a write transaction, we=0 begins a read transaction.

4.3.3 Byte select (bsel)

The byte select signal is used to select the range of bytes that is to be read from the target address. It can have a maximum width of 16.

Width	RISC-V ISA
4	32 bit
8	64 bit
16	128 bit

For example, with a width of 4, a bsel value of `0001` would select the 8 lowest bits (one byte) of the target address data content.

For the PicoNut processor in its first iteration, a width of 4 is sufficient.

Note: Frage: Wie lange valide und muss wann stabil anliegen?

4.3.4 Address (adr)

The address signal holds the value of the address which is to be accessed. For the RV32 ISA its width is equal or less than 30. For RV64 its width is less than or equal to 61.

Note: Frage: Wie lange valide und muss wann stabil anliegen?

4.3.5 Write data (wdata)

The write data signal contains the actual data to be written by the nucleus to an address managed by the memory unit. This signal, similarly to the write enable signal, is exclusive to the DPort interface. Valid signal widths are 32, 64 and 128.

Note: Frage: Wie lange valide und muss wann stabil anliegen?

4.3.6 Acknowledge (ack)

The acknowledge signal, set by the memory unit, signals to the core that the previous bus transaction was successful. The acknowledge signal stalls the bus until it is set high (a response occurs).

Note: Was tun im Fehlerfall?

4.3.7 Read data (rdata)

The read data signal is driven by the memory unit and holds the data is has read from the core during a bus transaction. The data is valid when ack=1 and thus valid for one clock cycle. Valid signal widths are 32, 64 and 128. The width of this signal (rdata) and the write data signal (wdata) are to be identical.

4.4 Modes of operation

4.4.1 Single mode

In default operation mode, the strobe (stb) signal may only stay high for a single clock cycle. The corresponding ack-pulse response sent by the memory unit stalls the bus until it is set. There is no constraint on how many clock cycles may pass before the memory unit sets the acknowledge signal high.

4.4.2 Overlap mode

The protocol also supports overlap mode. In overlap mode, one additional transaction may be initiated, before the previous one is complete (terminated by receiving a pulse of the `ack` signal). This means that at most two transactions may occupy the bus at any given time and their response windows may overlap. Following that, if two transactions have been initiated and not completed, at least one pulse of the `ack` signal has to be awaited, before a new transaction may be initiated.

4.5 Timing diagrams

4.5.1 Single mode

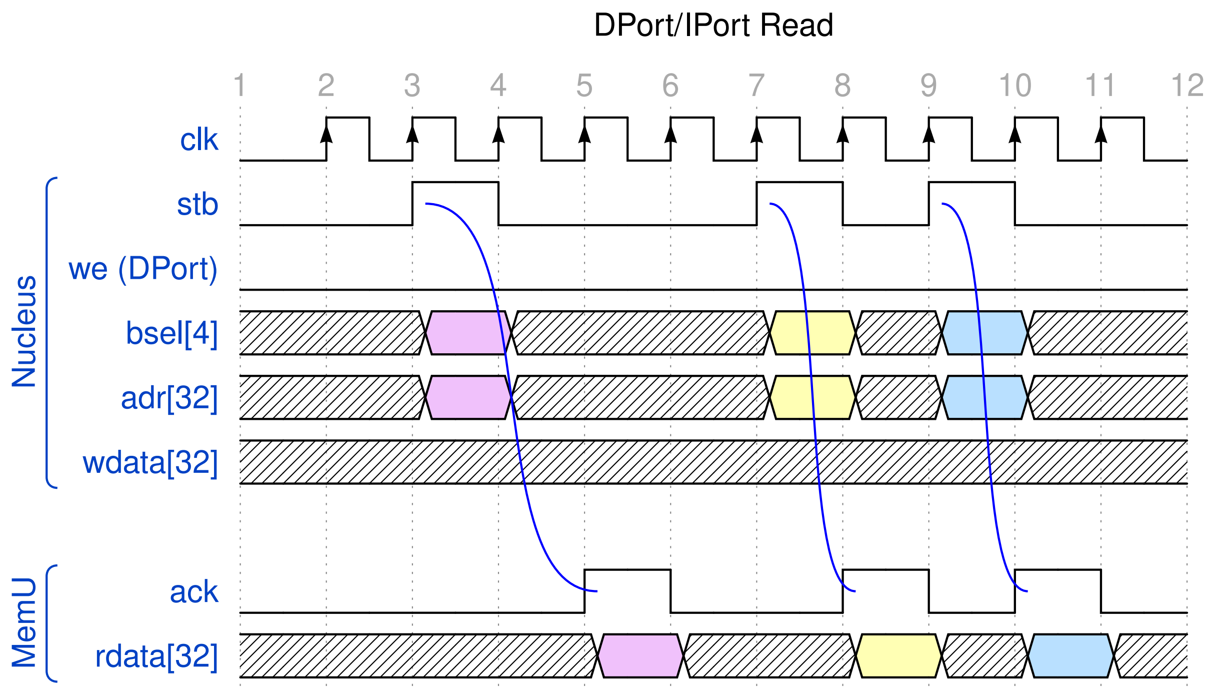


Fig. 4.1: DPort/IPort read cycle

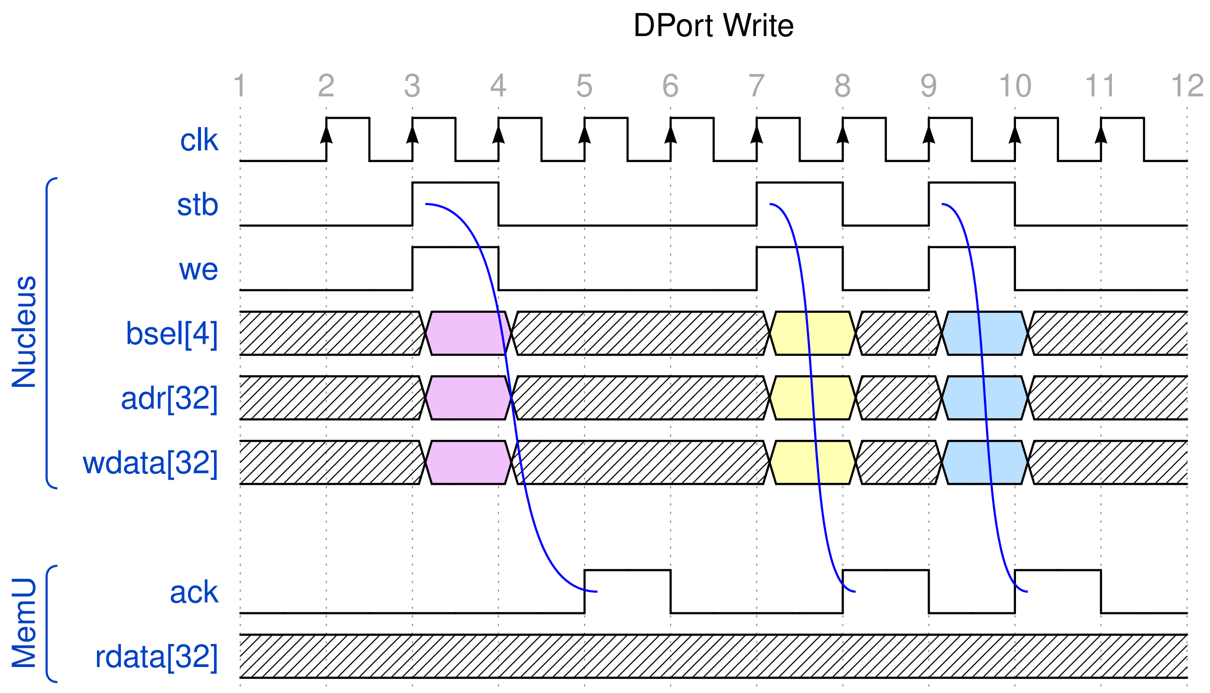


Fig. 4.2: DPort write cycle

4.5.2 Overlap mode

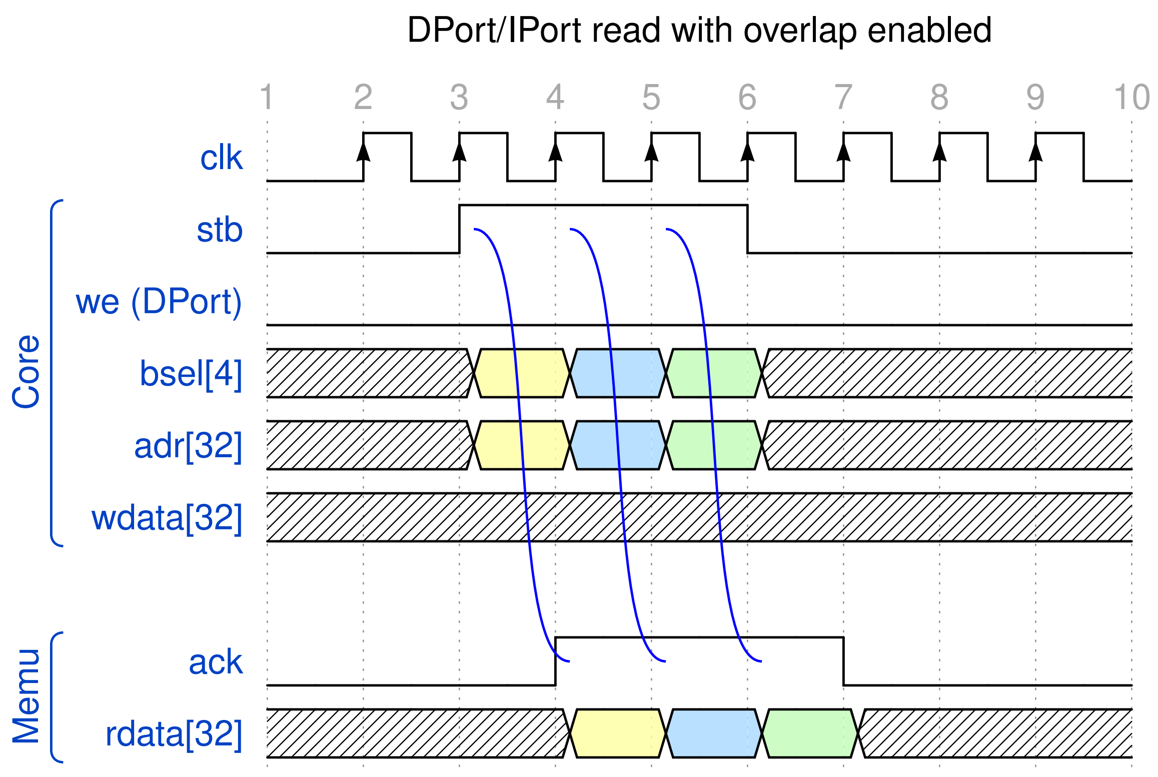


Fig. 4.3: DPort/Iport write cycle with overlap enabled

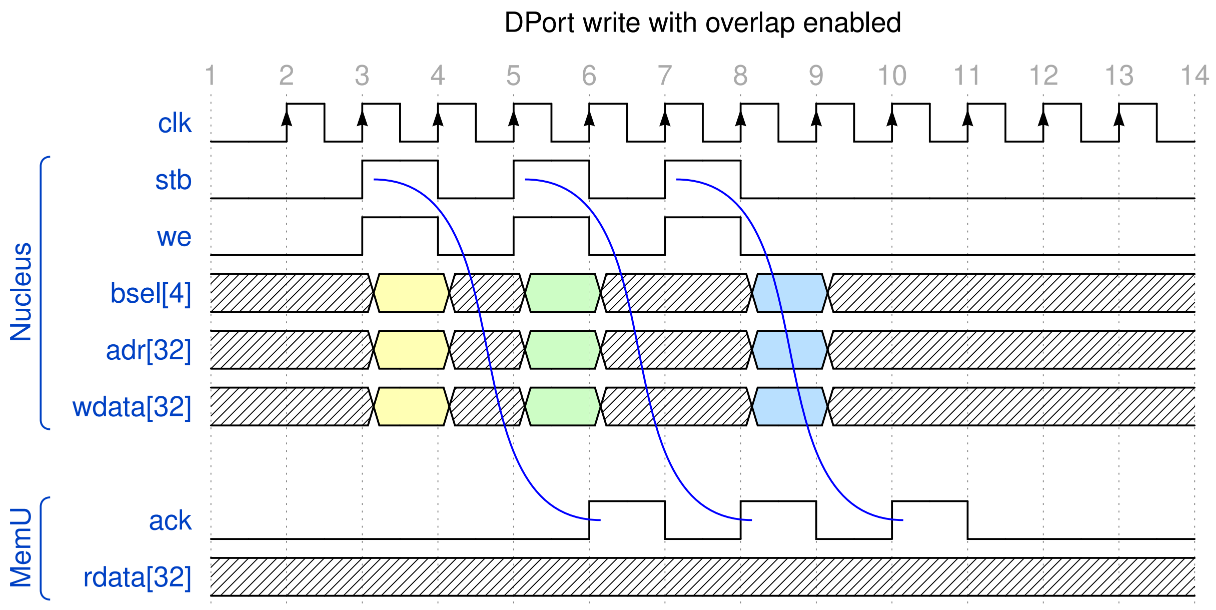


Fig. 4.4: DPort read cycle with overlap enabled

CHAPTER FIVE

UART

The UART module allows the PicoNut processor to communicate with the outside world. It is a simple module that can be used to send and receive data between the PicoNut and a user's computer. The module is based on the SiFive UART (SiFive FE310-G000 Manual), which has been implemented in the PicoNut project.

There are two implementations of the UART module in the PicoNut project. The first is the Wishbone UART, a UART module connected to the Wishbone bus, primarily intended for use on an FPGA. The second implementation is the SimUART, a UART module that can be connected to the `CSoftPeripheral` interface of the PicoNut processor, designed for use in the simulator.

5.1 UART Registers

A list of the registers for the UART module can be found in the documentation (SiFive FE310-G000 Manual) of the SiFive UART module.

5.2 Wishbone UART

The following figure provides an overview of the components of the Wishbone UART module. All subsequent diagrams omit the `clk` and `rst` signals for clarity.

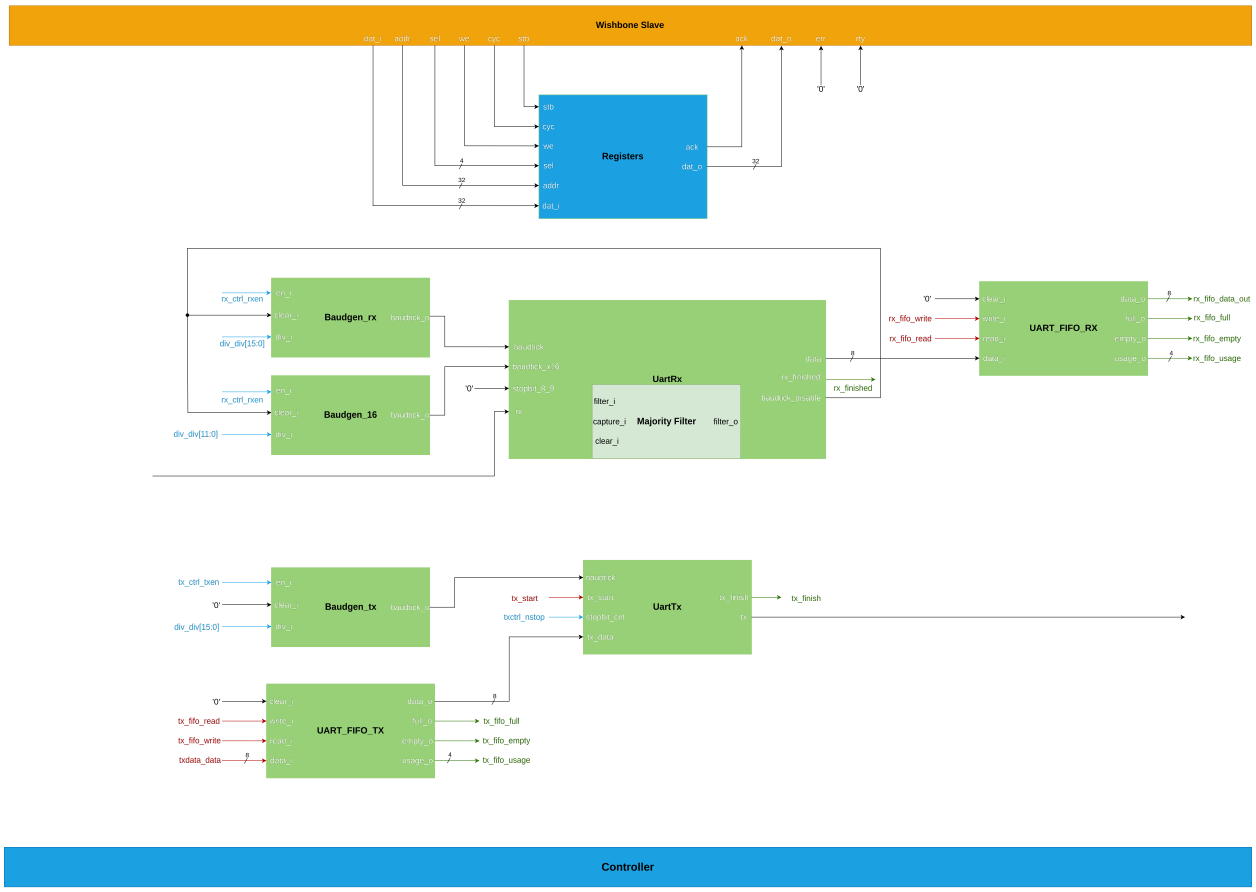


Fig. 5.1: Overview of the Wishbone UART module.

SC_MODULE(m_wb_uart)

#define

#define *

#define #define * following submodules:

- `baudgen_x16`: Baud rate generator for the 16x baud rate #define * - `baudgen_rx`: Baud rate generator for the receive (rx) baud rate
- `baudgen_tx`: Baud rate generator for the transmit (tx) baud rate
- `uart_rx`: Receiver module
- `uart_tx`: Transmitter module
- `uart_rx_fifo`: Receiver FIFO
- `uart_tx_fifo`: Transmitter FIFO

Author

Lukas Bauer

It connects the submodules and handles the Wishbone interface. It provides the registers as specified for the SiFive UART. The registers are:

- `txdata`: Transmit data register
- `rxdata`: Receive data register
- `txctrl`: Transmit control register
- `rxctrl`: Receive control register
- `ie`: Interrupt enable register
- `ip`: Interrupt pending register
- `div`: Baud rate divisor register

The module provides control functionality for the FIFOs, receiver, and transmitter. It initiates UART transmissions when data is written to the FIFOs, or writes the received data to the `rx_fifo`. It also controls whether the receiver or transmitter are enabled and at what baud rate they operate. Interrupts are also activated and deactivated by this module. A FIFO state machine handles Wishbone access to both FIFOs to prevent invalid transactions of data to and from the UART module.

There are two configuration options in the `config.mk` file:

- `CFG_WB_UART_DISABLE_FIFO`: This disables the FIFOs of the UART module. In this mode, only the `rx` and `tx` data registers are usable, meaning the `rxdata_empty` and `txdata_full` flags are set when the registers are in use. This also changes the behavior of the interrupts, where the `txctrl_txcnt` and `rxctrl_rxcnt` watermark levels have no effect. The watermark levels are set to 1, and the interrupts are triggered when the `txdata` and `rxdata` registers exceed the threshold as described in the SiFive UART specification.
- `CFG_WB_UART_BASE_ADDRESS`: This sets the base address of the UART module in the Wishbone bus address space.

Ports:

Parameters

- **`clk`** – **[in]** clock of the module
- **`reset`** – **[in]** reset of the module
- **`stb_i`** – **[in]** wb strobe
- **`cyc_i`** – **[in]** wb cycle
- **`we_i`** – **[in]** wb write enable
- **`sel_i`** – **[in]** wb byte select
- **`ack_o`** – **[out]** wb acknowledge
- **`err_o`** – **[out]** wb error
- **`rty_o`** – **[out]** wb retry
- **`addr_i`** – **[in]** wb address
- **`dat_i`** – **[in]** wb data in
- **`dat_o`** – **[out]** wb data out
- **`rx`** – **[in]** rx line
- **`tx`** – **[out]** tx line

The Wishbone UART module consists of the following components:

- *Baudgen*: Multiple Baudgenerators that divide the system clock to generate the baudrate for the UART module.
- *Majority Filter*: The majority filter is used to filter the received data. To ensure that the bit is received correctly.
- *FIFOs*: The FIFOs are used to buffer the data that is sent and received by the UART module.
- *uart_rx*: The receiver receives data from the outside world.
- *uart_tx*: The transmitter sends data to the outside world.

5.2.1 Baudgen

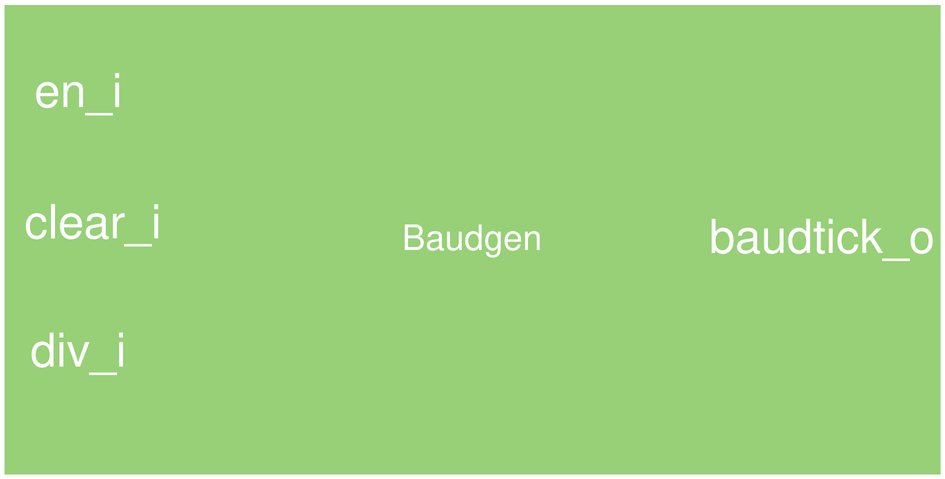


Fig. 5.2: Diagramm of the Baudrate Generator.

`SC_MODULE`(m_baudgen)

clockdivider to generate the baudtick for the UART

This clock divider can be configured by setting a divider value in the 16-bit `div` register. If the module is enabled, the baud generator counts up until the divider value is reached. Then, the `baudtick` is set to high, and the counter is reset.

Author

Lukas Bauer

Ports:

Parameters

- **`clk`** – **[in]** clock of the module
- **`reset`** – **[in]** reset of the module
- **`en_i`** – **[in]** enable for baudgen module
- **`clear_i`** – **[in]** clearing the baudgen (synchronous reset)
- **`div_i`** – **[in]** <16> divider value for baudgen
- **`baudtick_o`** – **[out]** generated baudtick

5.2.2 Majority Filter

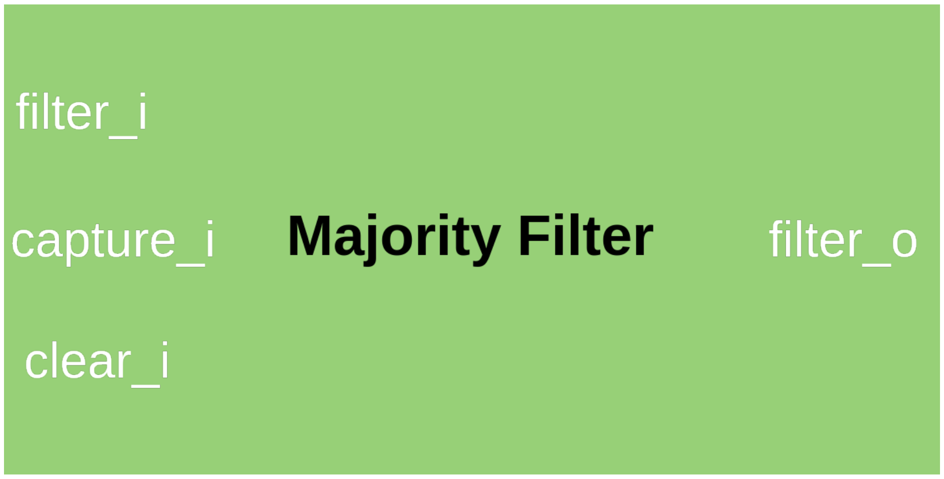


Fig. 5.3: Diagramm of the Majority Filter.

`SC_MODULE`(m_majority_filter)

majority filter to filter a input signal

This module allows checking if a signal remains stable for a certain amount of time. If the module is enabled, the input signal is checked every clock cycle. When the signal is low, the counter is incremented. Once the threshold is reached, the output signal is set to low. Otherwise, the output signal remains high.

Author

Lukas Bauer

Ports:

Parameters

- **`clk`** – **[in]** clock of the module
- **`reset`** – **[in]** reset of the module
- **`filter_i`** – **[in]** Signal to be filtered
- **`capture_i`** – **[in]** Enable Signal to capture the filter signal
- **`clear_i`** – **[in]** Clear the filter signal
- **`filter_o`** – **[out]** Filtered Signal

5.2.3 FIFOs

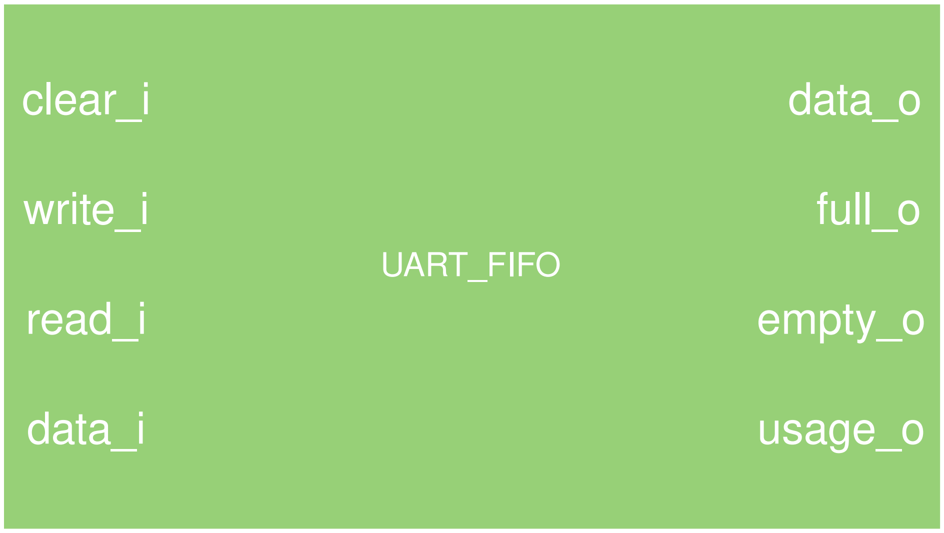


Fig. 5.4: Diagramm of the FIFOs.

SC_MODULE(m_uart_fifo)

first in first out buffer for UART

This module is a simple first-in, first-out (FIFO) buffer for the UART RX and TX. If the read and write pointers are equal, the buffer is empty. If the write pointer and the addresses are as far apart as possible, the buffer is full. The `usage` signal is a counter for the number of elements in the buffer. The `clear` signal resets the read and write pointers, as well as the usage counter. If `write` is set and the buffer is not full, the data is written to the buffer. If `read` is set and the buffer is not empty, the data is read from the buffer. The width of the data can be set using the `UART_FIFO_WIDTH` parameter. The size of the buffer can be set using the `UART_FIFO_SIZE_2E` parameter, which is the log2 of the size of the buffer. The buffer is implemented as a circular buffer.

Author

Lukas Bauer

Ports:

Parameters

- **`clk`** – **[in]** clock of the module
- **`reset`** – **[in]** reset of the module
- **`clear_i`** – **[in]** clear the content of the FIFO
- **`write_i`** – **[in]** write enable for the FIFO
- **`read_i`** – **[in]** read enable for the FIFO
- **`data_i`** – **[in]** <8> data to be written to the FIFO
- **`data_o`** – **[out]** <8> data to be read from the FIFO
- **`full_o`** – **[out]** FIFO is full

- **empty_o** – **[out]** FIFO is empty
- **usage_o** – **[out]** <4> usage of the FIFO

5.2.4 uart_rx

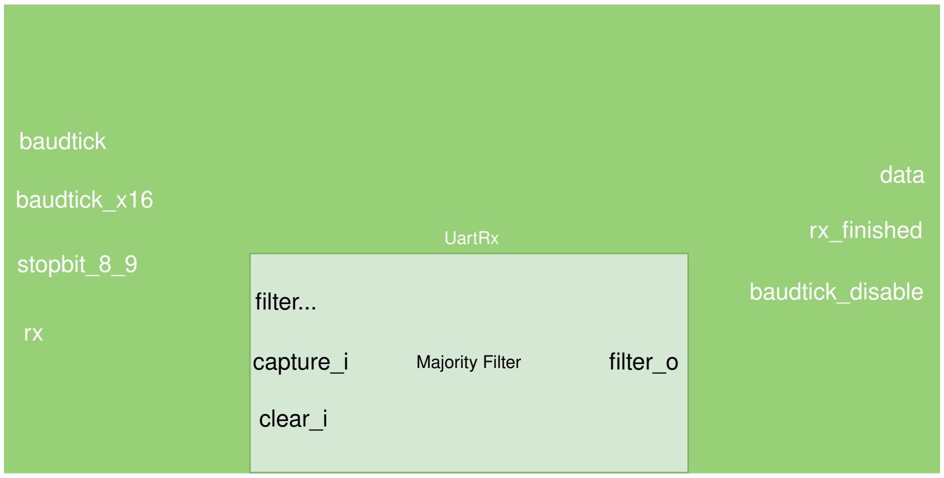


Fig. 5.5: Diagramm of the Receiver.

SC_MODULE(m_uart_rx)

Implementation of the UART receiver.

The UART receiver is a state machine that waits for the start bit of a UART frame. It contains the following submodule:

- Majority Filter Once the receiver detects the start bit, it starts the baudtick generation and waits for the next baudtick to sample the RX line. The received bits are stored in a shift register. They are then filtered by a majority filter to reduce noise. When the stop bit is received, the data is stored in the data register, and the `rx_finished` signal is set. The number of stop bits can be configured to 1 or 2.

Author

Lukas Bauer

Ports:

Parameters

- **clk** – **[in]** clock of the module
- **reset** – **[in]** reset of the module
- **baudtick_i** – **[in]** baudtick input
- **baudtick_x16_i** – **[in]** baudtick x16 input
- **stopbit_8_9_i** – **[in]** unset for 1 stop bit, set for 2 stop bits
- **rx_i** – **[in]** UART rx line

- **data_o** – **[out]** <16> Data Output
- **rx_finished_o** – **[out]** RX finished output
- **baudtick_disable_o** – **[out]** disable the baudtick generation if needed

5.2.5 uart_tx

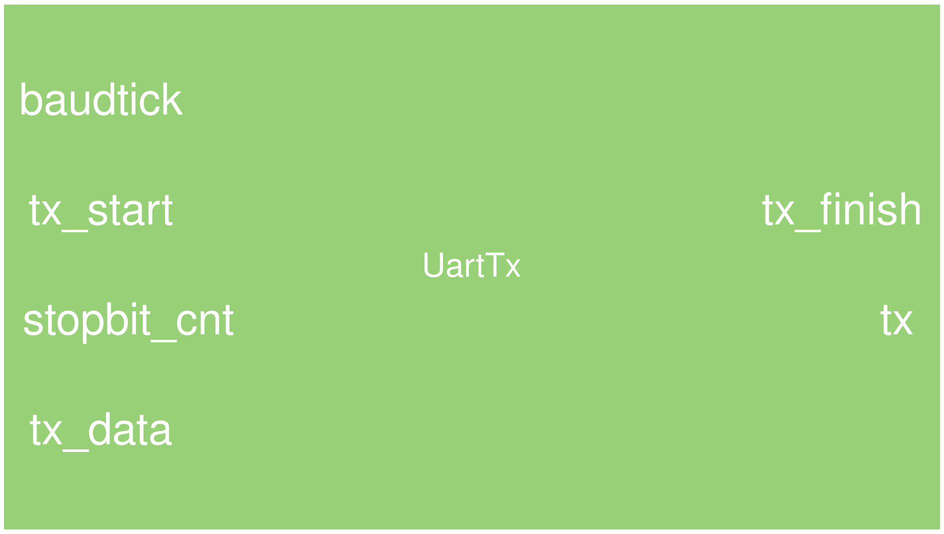


Fig. 5.6: Diagramm of the Transmitter.

SC_MODULE(m_uart_tx)

Implementation of the UART transmitter.

This module is used to transmit data over the UART interface. The data is provided to the module as an 8-bit word, and the module will transmit the data over the `tx_o` line. Transmission is initiated by setting the `tx_start_i` signal. The module will then transmit the data and set the `tx_finish_o` signal when the transmission is complete. The module also handles the stop bits. If the `stopbit_cnt_i` signal is set, the module will transmit 2 stop bits. If the signal is unset, the module will transmit 1 stop bit.

Author

Lukas Bauer

Ports:

Parameters

- **clk** – **[in]** clock of the module
- **reset** – **[in]** reset of the module
- **baudtick_i** – **[in]** baudtick input
- **tx_start_i** – **[in]** start signal for transmission
- **stopbit_cnt_i** – **[in]** unset: 1 stopbit, set: 2 stopbits
- **tx_data_i** – **[in]** <8> 8-bit word to transmit

- `tx_finish_o` – **[out]** transmission finished signal
- `tx_o` – **[out]** tx line signal

5.2.6 System for hardware test

To test the Wishbone UART module on hardware, a system has been created. This system contains a test module that can configure the UART module and echo the received data back. The system is synthesizable for the ULX3S board.

It contains two SystemC modules:

- *uart_wb_uart*: Contains the test hardware for the UART module.
- *top*: Contains a wrapper for synthesis

uart_test

`SC_MODULE`(m_uart_test)

a hardware module to test the wb_uart

This module initializes the wb_uart module via its Wishbone interface by setting the baud rate and enabling RX and TX. It then periodically checks the rxdata register to see if the RX FIFO is no longer empty. If that is the case, it saves the received data in a register and sends it to the txdata register to be sent back. This effectively echoes the received data back to the sender to check the basic functionality of the wb_uart module in hardware.

Author

Lukas Bauer

Ports:

Parameters

- `clk` – **[in]** the clock signal of the module
- `reset` – **[in]** the reset signal of the module
- `rx` – **[in]** the receive signal of the uart
- `tx` – **[out]** the transmit signal of the uart

top

`SC_MODULE`(m_top)

wrapper for synthesis

This module is just the top level wrapper for synthesis

Author

Lukas Bauer

Ports:

Parameters

- **`clk_25`** – **[in]** the clock signal comming form the board
- **`reset`** – **[in]** the reset signal of the module
- **`rx`** – **[in]** the receive signal of the uart
- **`tx`** – **[out]** the transmit signal of the uart

5.3 SimUART

The SimUART module implements a UART module that can be connected to the `CSoftPeripheral` interface of the PicoNut processor, which is described in Chapter *Software Library Documentation*.

The following figure provides an overview of the components of the SimUART module.

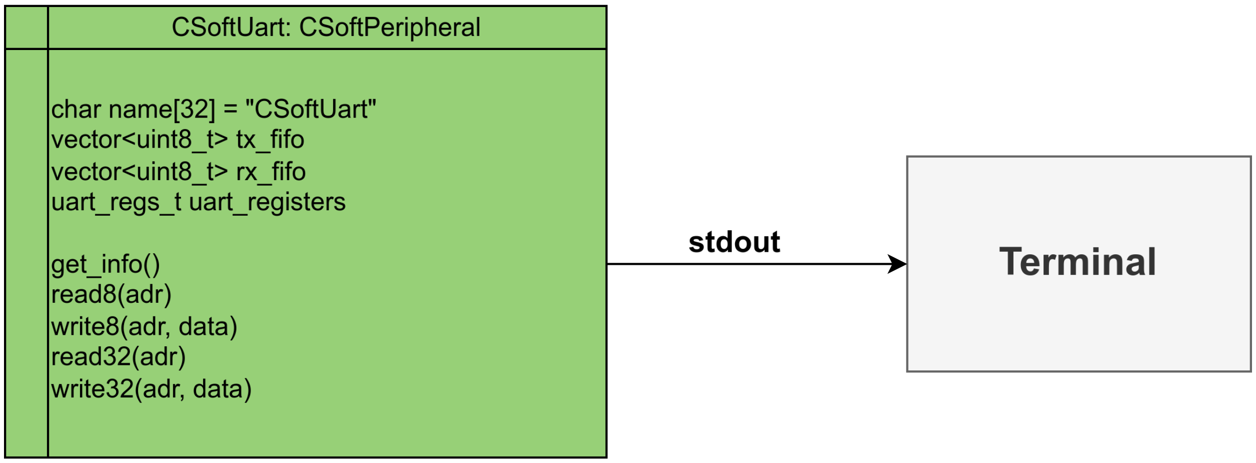


Fig. 5.7: Overview of the SimUART module.

group **`c_soft_uart`**

soft peripheral implementation of the sifive UART for simulation

This module is used to simulate the UART peripheral of the SiFive core. The module is implemented as a soft peripheral and can be used in the simulation environment. The module has the same registers as the SiFive UART. They are:

- `txdata`: 32-bit register for the transmit data
- `rxdata`: 32-bit register for the receive data
- `txctrl`: 32-bit register for the transmit control
- `rxctrl`: 32-bit register for the receive control
- `ie`: 32-bit register for the interrupt enable
- `ip`: 32-bit register for the interrupt pending
- `div`: 32-bit register for the baud rate divider

Author

Lukas Bauer

The module has a 32-bit memory interface and can be accessed by the soft peripheral interface.

Note: At the moment, the module only implements transmit functionality, meaning it can only display data on `stdout`, but cannot receive data on `stdin`.

Note: The module is not synthesizable and is only used for simulation purposes.

Note: The module does not simulate the baud rate because it is not needed for the simulation, meaning the `div` register has no effect on the module.

5.3.1 CSoftPeripheral Interface

The following methods are those implemented from the `CSoftPeripheral` interface. To enable the UART module to be connected to the `CSoftPeripheral` interface, the following methods must be implemented:

inline *c_soft_uart*::**c_soft_uart**(uint64_t size, uint64_t base_address)

Construct a new Soft Uart object.

Parameters

- **size** – address space of the peripheral
- **base_address** – base address of the peripheral in address space of the simulation

inline const char *`c_soft_uart::`**get_info**() override

inline uint8_t `c_soft_uart::`**read8**(uint64_t adr) override

inline void `c_soft_uart::`**write8**(uint64_t adr, uint8_t data) override

inline void `c_soft_uart::`**write32**(uint64_t adr, uint32_t data) override

inline uint32_t `c_soft_uart::`**read32**(uint64_t adr) override

inline bool `c_soft_uart::`**is_addressed**(uint64_t adr) override

5.3.2 UART Emulation

The following methods are implemented to emulate the functionality of the SiFive UART module. They handle data transmission, the FIFO, and interrupt generation.

void `c_soft_uart::`**handel_transmit**()

handels the transmit of the uart

Handles the transmission of the UART by checking if txen is enabled, and then sending the data from the tx_fifo if the UART is enabled and the tx_fifo is not empty. The data is sent to stdout.

void `c_soft_uart::`**update_txfull**()

updates the txfull flag in the txdata register

if the fifo is not full its set to false else to true

void `c_soft_uart::`**handel_interrupt**()

sets the values in the ip register according to the conditions

Checks if the interrupts are enabled and then sets the interrupt flag if the cnt register has the correct value.

void `c_soft_uart::`**`update_uart`**`()`

runs handel_interrupt and handel_transmit to update the uart

CHAPTER

SIX

SOFTWARE LIBRARY DOCUMENTATION

6.1 Overview

This documentation provides an overview and usage guide for the software library designed to manage peripherals and handle ELF file parsing for embedded system simulations. The library is structured to support easy integration and extension, with components dedicated to different aspects of system emulation, such as memory management, peripheral interfacing, and file handling.

6.2 Features

- **Peripheral Management**: Manages a variety of simulated peripherals.
- **Memory Simulation**: Simulates different memory scenarios and behaviors.
- **File Parsing**: Supports reading and interpreting ELF files.

6.3 Getting Started

6.3.1 Prerequisites

- A modern C++ compiler supporting C++17 or later.
- Basic understanding of computer architecture and peripheral management.
- Familiarity with the PicoNut project and its components.

6.4 Library Components

The library is divided into several key components, each handling a specific aspect of system simulation. Detailed documentation for each component is provided in the following sections.

6.5 c_soft_peripheral

The c_soft_peripheral class is the base class for all peripherals in the software library. It provides a common interface for peripheral operations and is designed to be extended for specific peripheral implementations.

6.5.1 Usage Example

```
class my_peripheral : public c_soft_peripheral {
public:
    my_peripheral(uint32_t base_address) : c_soft_peripheral(base_address) {}

    uint8_t read(uint32_t address) override {
        // Read from the peripheral
    }

    void write(uint32_t address, uint8_t data) override {
        // Write to the peripheral
    }
};
```

Refer to existing peripherals such as c_soft_memory and c_soft_uart for further examples.

6.6 c_soft_memory

The c_soft_memory class simulates memory within the software library. It supports basic read and write operations to access memory locations.

6.6.1 Usage Example

```
c_soft_memory memory(1024, 0x40000000);
memory.write(0x40000000, 0x12);
uint8_t data = memory.read(0x40000000);
```

The constructor takes the memory size and base address as parameters. It also provides methods to dump the memory contents to a file.

6.7 c_soft_uart

The c_soft_uart class simulates a UART interface. It allows data transmission and reception over a simulated UART connection and supports setting the baud rate, checking for data availability, and reading/writing data.

6.7.1 Usage Example

```
c_soft_uart uart(0x40001000);
uart.set_baud_rate(9600);
uart.write(0x40001000, 'A');
uint8_t data = uart.read(0x40001000);
```

6.8 Peripheral Interface

The `c_soft_peripheral_container` class manages the peripherals in a simulation-only environment. It allows adding, interacting with, and managing peripherals, typically used within a `m_soft_memu` or similar module.

6.8.1 Class: c_soft_peripheral_container

Public Methods

6.8.2 `void add_peripheral(uint64_t base_address, std::unique_ptr<c_soft_peripheral> peripheral)`

Description: Adds a new peripheral to the system at the specified base address.

Parameters:

- `base_address` (*uint64_t*): The base address of the peripheral.
- `peripheral` (*std::unique_ptr<c_soft_peripheral>*): A unique pointer to the peripheral.

Usage Example:

```
auto my_peripheral = std::make_unique<my_peripheral>();
peripheral_interface.add_peripheral(0x1000, std::move(my_peripheral));
```

6.8.3 `c_soft_peripheral* find_peripheral(uint64_t address)`

Description: Finds a peripheral by its memory address.

Parameters:

- `address` (*uint64_t*): The memory address to search for.

Returns:

- A pointer to the peripheral if found, otherwise `nullptr`.

Usage Example:

```
c_soft_peripheral* peripheral = peripheral_interface.find_peripheral(0x1000);
if (peripheral != nullptr) {
    // Peripheral found
}
```

6.8.4 void write_peripheral(uint64_t address, uint32_t data, uint8_t bsel)

Description: Writes data to a peripheral at the specified address, using the byte-select value.

Parameters:

- address (*uint64_t*): The peripheral address.
- data (*uint32_t*): Data to write.
- bsel (*uint8_t*): Byte-select value.

Usage Example:

```
peripheral_interface.write_peripheral(0x1000, 0xFF00FF00, 0x0F);
```

6.8.5 uint32_t read_peripheral(uint64_t address, uint8_t bsel)

Description: Reads data from a peripheral at the specified address, using the byte-select value.

Parameters:

- address (*uint64_t*): The address of the peripheral.
- bsel (*uint8_t*): Byte-select value.

Returns:

- The data read from the peripheral.

Usage Example:

```
uint32_t data = peripheral_interface.read_peripheral(0x1000, 0x01);
```

6.9 ELF Reader

The elf_reader class provides methods for reading and parsing ELF files in the software library. It reads the contents of an ELF file, extracts important information such as entry points and program headers, and supports symbol parsing for debugging purposes.

6.9.1 Example Usage

```
elf_reader elf("my_program.elf");
elf.initialize_from_file();
char* memory = elf.get_memory();
```

The ELF reader loads ELF files into memory and provides access to symbols for debugging and memory dumping.

6.10 Troubleshooting

Common issues and solutions:

- **Peripheral not found**: Ensure the peripheral's base address is correctly added to the system.
  - Is also intended to show up whenever a new peripheral is added to declare that there is not already a existing one at that address range.
- **Invalid memory access**: Check that the memory address falls within the bounds of the initialized memory.

CHAPTER

SEVEN

HOW TO DOCUMENT

7.1 Introduction

This page will detail how to document development on the PicoNut project using the setup provided.

7.1.1 Software dependencies

In order to build the documentation from the source directory, a few software packages have to be installed.

- pyhton3
- python3-sphinx
- python3-myst-parser
- python3-sphinx-rtd-theme
- python3-breathe
- python3-sphinxcontrib.svg2pdfconverter
- inkscape

Please install these packages with your package manager of choice.

7.1.2 MyST parser

This documentation uses the markdown language embedded into Sphinx through the extension myst-parser. For a full documentation of the features that myst provides, navigate to the official myst-parser documentation page.

Note: Ensure that the selected version of the documentation on the previously linked page is set to 0.18.1.

7.1.3 How to build

To build the documentation, navigate to `piconut/doc-src/pn-manuals` and invoke `make html`. This will create a file `index.html` in `piconut/doc-src/build/html` which can be opened with any web browser.

Additional targets for the `make` command include:

- `make clean` to clear the build files.
- `make latexpdf` to generate a PDF file.

To see all available options use `make help`.

7.1.4 Standard markdown syntax

This documentation is created using the markdown language.

A brief overview of standard markdown can be found markdownguide.org.

7.1.5 How to include markdown files

In order to add Markdown files to the documentation overall, it has to be included in `toctree` structure the file `index.md`. Simply listing the file by it's relative path is sufficient.

7.2 Additional syntax provided by MyST

The MyST-parser provides a range of extra features beyond standard markdown.

7.2.1 Comments

Comments can be made within source `.md` files. For this the `%` symbol is used.

```
% comment
```

7.2.2 Directives and Roles

MyST acts as a wrapper for RestructuredText roles and directives, simplifying the syntax.

Directives and roles allow the user to format text in a predefined manner. Directives are structured as a block surrounded by backticks, roles are constructed in-line.

According to the MyST manual, all docutils roles, docutils directives, sphinx roles and sphinx directives should be functional within MyST.

A short overview over the most commonly used directives will be provided here.

Admonitions

Admonitions such as warnings and notes are created as follows:

~~~
```{note}
    This is a note.
```
~~~

will render as:

Note:

```
This is a note.
```

Options include: `note`, `warning`, `attention`, `error`, `hint`, `important` and `tip`

Code

Code blocks can be constructed with the standard markdown syntax or the MyST provided syntax.

Standard Markdown code blocks

~~~
```c
int main()
    {
        printf("Hello World!\r");
    }
```
~~~

renders as

```c
int main()
    {
        printf("Hello World!\r");
    }
```

MyST directive

~~~
```{code-block} $language
    code
```
~~~

Is the default syntax.

~~~
```{code-block} c
    int main()
    {
        printf("Hello World!\r");
~~~

(continues on next page)

(continued from previous page)

```
    }
```
```
```

renders as

```
    int main()
    {
        printf("Hello World!\r");
    }
```

Tables and figures

Documentation on how to creates tables and how to include images and figures into the document can be found under *tables* and *figures and images*.

Roles

Roles in essence achieve the same things that directives do, but they are used in-line instead of requiring their own block.

The default syntax is:

```
{role-name}`role content`
```

For example:

```
    {math}`1+2 = 3`
```

renders as

$1 + 2 = 3$

References

References are useful when the user wants to create a quick and clickable reference to another section of the documentation. References can be created for headlines, images, tables and code blocks.

To begin with, a reference has to be declared at the target of the reference. The syntax for this is

```
(referencename)=
```

To create a clickable object to lead to a reference, the following snytax options can be used:

```
[text](referencename)
```

Example:

```
(testreference)=
# Some headline

This is an example.
```

(continues on next page)

(continued from previous page)

```

[This reference](testreference) will jump to the headline.
```

Alternative syntax for the clickable link is

```
{ref}`testreference`
```

7.2.3 Nesting of Backticks

When a block encased in backticks needs to be nested in another block encased in backticks, the outer block needs to have an additional backtick at the beginning and end of the block.

~~~
````
``` markdown
# hello
```
````
~~~

7.2.4 Footnotes

To add a footnote, use the following syntax:

```
Here is a footnote reference,[^1] and another.[^longnote]

[^1]: Here is the footnote.
[^longnote]: Here's one with multiple blocks.
```

This will render as:

Here is a footnote reference,[1] and another.[2]

7.2.5 Embed other files

To include links to other files e.g. documents in the pdf format, use the following syntax:

```
Here is a link to the technical report [zybo-z7_rm.pdf](doc/zybo-z7_rm.pdf).
```

This will render as:

Here is a link to the technical report zybo-z7_rm.pdf.

Note: The link will not be usable in the pdf version of the document.

[1] Here is the footnote.
[2] Here's one with multiple blocks.

7.3 Tables

To create tables the Github Markdown Syntax can be used.

Use pipes `|` to separate columns and hyphens `-` to create a header row.

```
| Header 1 | Header 2 | Header 3 |
| -------- | -------- | -------- |
| Cell 1   | Cell 2   | Cell 3   |
| Cell 4   | Cell 5   | Cell 6   |
```

This will render as:

Header 1	Header 2	Header 3
Cell 1	Cell 2	Cell 3
Cell 4	Cell 5	Cell 6

The table can be aligned to the left, right, or center by adding colons `:` to the header row.

```
| Left-aligned | Center-aligned | Right-aligned |
| :---         | :---:          | ---:          |
| Cell 1       | Cell 2         | Cell 3        |
| Cell 4       | Cell 5         | Cell 6        |
```

This will render as:

Left-aligned	Center-aligned	Right-aligned
Cell 1	Cell 2	Cell 3
Cell 4	Cell 5	Cell 6

It is also possible to add a caption and other options to the table.

~~~
```{table} Table Title
:align: center

| Header 1 | Header 2 | Header 3 |
| :------- | :------: | -------: |
| Cell 1   | Cell 2   | Cell 3   |
| Cell 4   | Cell 5   | Cell 6   |
```
~~~

This will render as:

Table 7.1: Table Title

Header 1	Header 2	Header 3
Cell 1	Cell 2	Cell 3
Cell 4	Cell 5	Cell 6

Note: The `table` directive is used to include tables with captions and additional options. These can be found in the MySt Parser Table Documentation

7.4 Images and Figures

Images and figures can be includes in multiple ways.

7.4.1 Supported Formats

The following image formats are supported:

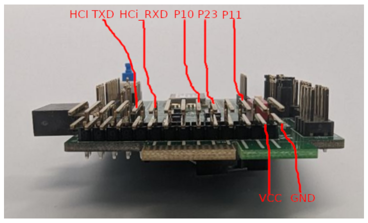
- png
- jpg
- jpeg
- pdf
- svg

7.4.2 Markdown Syntax

To include an image in a markdown file, use the following syntax:

```
![Alt text](figs/Pictail_BM70.jpeg)
```

this will render as:



7.4.3 MySt Syntax

To include an image in a MySt file, use the following syntax:

~~~
```{figure} figs/pn_logo_long.svg
:scale: 20
:alt: Alt text
:align: center

Caption text
```
~~~

this will render as:

Fig. 7.1: Caption text

Note: insead of the `figure` directive, the `image` directive can be used. The `figure` directive is used to include images with captions.
Additional options can be found in the MySt documentation

7.5 Using draw.io diagrams

When creating diagrams for the documentation with draw.io the file should be exported into either of the following formats:

- `.drawio.svg`
- `.drawio.png`

These files can still be edited with draw.io but can also be used in the documentation.

7.6 Doxygen

7.6.1 Usage

To include Doxygen code documentation into the Sphinx documentation, the `breathe` extension is used.

When adding a new file to the documentation, the following steps have to be taken:

1. Add the file to the `breathe_projects` and `breathe_projects_source` in the `conf.py` file.

Either add it to an existing project entry by adding the wanted file to the list of files, like so:

```
breathe_projects = {
    "how_to_document": ("../build/breathe/doxygen/how_to_document/xml")
}
breathe_projects_source = {
    "how_to_document": ("how_to_document/", ["paranut.h", "piconut.hpp"])
}
```

Or create a new project entry by adding a new key-value pair to the dictionary, like so:

```
breathe_projects = {
    "how_to_document": ("../build/breathe/doxygen/how_to_document/xml"),
    "<new_project>": ("../build/breathe/doxygen/<new_project>/xml")
}
breathe_projects_source = {
    "how_to_document": ("how_to_document/", ["paranut.h", "piconut.hpp"]),
    ">new_project>": ("<path to directory>", ["<list of filenames>"])
}
```

2. Use the commands of the `breathe` extension to add the documentation to the document. A list of available commands can be found in the breathe documentation.

7.6.2 Example

Documenting a function

To document a function, the following syntax is used:

~~~
```{doxygenfunction} start_sim
:project: how_to_document
```
~~~

This will render as:

void **pn_halt**(void)

Halt whatever core the function is executed on.

If executed on a core, it will be set to Mode 0 and stop operation. Causes reset of program counter to reset address on a Mode 2 capable CPU.

If executed on CePU, also halts all other CoPUs on system.

See documentation of Modes for more information.

This function returns nothing because it should not be possible for any core to leave this state on its own.

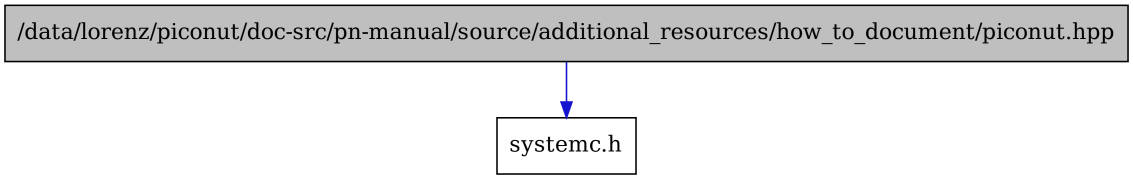
The name of the function is given as an option to the directive and the project specifes the project the function is part of.

Documenting a whole file

To document a whole file, the following syntax is used:

~~~
```{autodoxygenfile} piconut.hpp
:project: how_to_document
:allow-dot-graphs:
```
~~~

Include dependency graph for piconut.hpp:


/data/lorenz/piconut/doc-src/pn-manual/source/additional_resources/how_to_document/piconut.hpp
systemc.h

API of the libparanut.

Functions

void **`start_sim`**(sc_signal<bool> clk, int test)

Starts the simulation.

This function starts the simulation by taking a clock signal and a test number as input.

Parameters

- **`clk`** – The clock signal used for simulation.
- **`test`** – The test number to be used for simulation.

The name of the file is given as an option to the directive and the project specifes the project the file is part of. The `allow-dot-graphs` generates dot graphs to visualize the dependencies of the file.

7.7 Tips and Workarounds

7.7.1 Table Formatting

When creating tables, it is recommended to format them as shown in the *tables* section. This method allows the author to use line breaks within the cells of the table. Using other MyST table directives, such as `csv-table` or `list-table`, can lead to missing table lines and misplaced content when generating a PDF. Therefore, it is recommended to use standard Markdown table formatting, from this documentation.

CHAPTER

EIGHT

MAKESYSTEM

8.1 Integrate a Core/Memu into the piconut toplevel

When creating a new core or memory, it needs to be integrated into the piconut toplevel.

The makesystem provides a way to select different cores and memories to be used in the piconut. For this #ifdef statements are used in the piconut.h file to select the correct core and memory. The name of the define is generated by the makesystem and is the same as the folder name of the core or memory. The format is the folder name in uppercase surrounded by two underscores. For example if the core is in the folder hw/core/my_core the define will be __MY_CORE__.

In order to integrate a new core or memory into the piconut toplevel the following steps need to be done:

1. Add an #include statement to the piconut.h file. the #include statement should be inside an #ifdef statement with the name of the core or memory.

```
#ifdef __MY_CORE__
#include "my_core.h"
#endif
```

2. Add a Instance variable to the SC_MODULE in the piconut.h file. For cores use the varaible name core and for memories use the variable name memu. This also needs to be inside an #ifdef statement with the name of the core or memory.

```
#ifdef __MY_CORE__
my_core *core;
#endif
```

8.2 Adding a configuration option to the piconut

To add a configuration option to the piconut, it needs to be added to three different files:

- hw/piconut/config.mk
- hw/common/piconut-config.template.h
- hw/common/Makefile

8.2.1 hw/piconut/config.mk

In the config.mk file, the Option is added as a makefile variable. The option starts with CFG_ followed by the name of the option in uppercase and with underscores instead of spaces. After this a ?= followed by the default value of the option is added. Above the option, a comment must be added to describe the option.

```
# The option .....
CFG_MY_OPTION ?= 0
```

8.2.2 hw/common/piconut-config.template.h

This file is used to include the configuration options into the hardware source code. The options are included as #define statements. The name of the option must be the same as in the *config.mk* file. As a value the name of the option enclosed in curly braces is used. This is used to replace fill in the value of the option during the build process. Each option must have a doxygen comment to describe the option above it.

```
/**
 * @brief Brief description of the option
 *
 * The option .....
 */
#define CFG_MY_OPTION {CFG_MY_OPTION}
```

8.2.3 hw/common/Makefile

To use the configuration option in the hardware source code, the piconut-config.h needs to be generated with values from the config.mk file. To do this the sed command at the end of the Makefile in the hw/common needs to be modified. The sed command replaces the curly braces in the piconut-config.template.h file with the values from the config.mk file. For this you need to add a line to the sed command for each configuration option.

```
        @sed \
        -e 's#{CFG_REGFILE_SIZE}#$(CFG_REGFILE_SIZE)#g' \
    -e 's#{CFG_MY_OPTION}#$(CFG_MY_OPTION)#g' \
```

8.3 Creating a jenkins pipeline

When developing hardware or software for the piconut, it is important to test on a regular basis. To automate this process a jenkins server is used to run tests.

For this, a jenkins pipeline needs to be created. The pipelines are scripts written in groovy and define the steps that need to be executed.

8.3.1 Create a new pipeline file

The pipeline files are located in a separate repository, in which some configuration steps need to be taken.

The following steps need to be done to create a new pipeline file:

1. Clone the repository:

```
$ git clone https://ti-build.informatik.hs-augsburg.de:8443/piconut_developers/piconut_
↪jenkins-pipeline.git
```

2. Switch to the develop branch:

```
$ git checkout develop
```

3. Create a folder for the new pipeline:

The repository is structured for `build` and `test` pipelines. Build pipelines are used to build the hardware and software, while test pipelines are used to run tests e.g. testbenches for hardware or running the software in the simulator.

The `build` directory is divided further into the following folders:

- `doc`: for building the documentation
- `software`: for building software
- `sysc_synthesis`: for running systemc synthesis of modules
- `v_synthesis`: for running synthesis of whole systems. Generating bitstreams for different FPGA boards

The `test` folder currently does not have any subfolders.

The folders mentioned above contain the pipeline files. They mirror the structure of the source code repository.

For example, if a pipeline for the testbench of the `my_core` module which is located in the `hw/core/my_core` in the source code repository, the pipeline file would be located in the `test/hw/core/my_core` folder in the jenkins pipeline repository. If a pipeline for the ICSC SystemC synthesis of the `my_core` module is needed, the pipeline file would be located in the `sysc_synthesis/hw/core/my_core` folder in the jenkins pipeline repository.

4. Create the pipeline file:

The easiest way to create a new pipeline file is to copy an existing pipeline file of the same test type and modify it. Meaning if a pipeline for ICSC SystemC synthesis is needed, copy an existing pipeline from the `sysc_synthesis` folder and modify it. The name of the pipeline file needs to be changed accordingly.

5. Modify the pipeline file:

When modifying the pipeline file, two lines need to be changed

```
    args "-u jenkins --name test_hw_peripheral-bootram -v ${env.WORKSPACE}:/home/jenkins"
```

The `--name` argument needs to be adjusted to reflect the name of the hardware or software component that is being tested. It is required that the name is unique so all pipelines can be run in parallel.

```
    steps {
        sh '/bin/bash -c "cd /home/jenkins && cd hw/peripherals/bootram && make clean &&␣
↪\
        make run-tb"'
    }
```

The path that the `cd` command navigates to needs to be adjusted to the path of the hardware or software component that is being tested.

Important: If the pipeline is not of the same type (e.g. `testbench` or `sysc-synthesis`), the `make` command in the second line needs to be adjusted to run the correct make target.

8.3.2 Add the pipeline to the jenkins server

1. Log in to the jenkins server

The jenkins server is located at https://ti-build.informatik.hs-augsburg.de:8444/

Log into the jenkins server with your RZ credentials.

2. Add the pipeline

The jenkins webinterface uses the same structure as the pipeline repository.

To add a new pipeline, navigate to the folder where the pipeline file is located in the repository. If the folder does not already exist, create it by clicking on the `Create Element` button in the side menu. Then enter the folder name and select the `Folder` option. Click `OK`. In the next screen enter the folder name again and click `OK`.

When the folder structure is complete, click on the `New Item` button in the side menu. Select `free-style project` and use the `copy from` field at the bottom to assign a relative path to an already existing pipeline file. Then enter the name pipeline. The name must start with the appropriate pipeline type. The type starts with an uppercase letter. This is followed by the name of the hardware or software component that is being tested. E.g `Test_blockram_tb`.

At the bottom of the page change the path to the absolute path of the pipeline file in the repository.

Important: When creating a new pipeline, that tests code that is not on the develop branch, the parameter `BRANCH` at the top of the pipeline configuration needs to be adjusted to the correct branch name. When the merge to the develop branch is done, this value needs to be changed back to `develop`.

3. Test the pipeline

To ensure that the added pipeline works, click on the `build with parameters` button in the side menu and click `Build`. This will start the pipeline. To show the output of the pipeline click on the most current buildnumber in the bottom left of the screen. E.g. `#<number>`. Then click on the `Console Output` button in the side menu, to see the output of the pipeline. If the pipeline fails, the output will show `FAILED` otherwise it will show `SUCCESS`.

CHAPTER

NINE

BLOCK RAM TEMPLATES

This section provides an explanation of the Block RAM templates that are available in the PicoNut project. The templates are used to generate the Block RAMs that can be uses in the PicoNut project to e.g. store program code/data, caches or other data structures.

Firstly, the different types of Block RAMs that are available in the PicoNut project are explained. Secondly, a description of how to use one of the templates in a design is given.

The Types of Block RAMs that are available in the PicoNut project are:

- *Single Port Block RAM*
- *Dual Port Block RAM*
- *Dual Port Byte Enable Block RAM*

All the Block RAMs are synchronous RAMs that are write first.

They consist of two parts:

- The SystemC part that is used to simulate the RAM
- The Verilog part that is used to synthesize the RAM

This is done to ensure that the Block RAMs are not changed during the ICSC synthesis process. So that the synthesis tools for the boards e.g. Vivado or yosys can detect the RAMs and synthesize them correctly.

9.1 Single Port Block RAM

9.1.1 Description

The single port block RAM is a simple block RAM that has one port for reading and writing data. The RAM is a synchronous RAM that is write first.

9.1.2 Ports

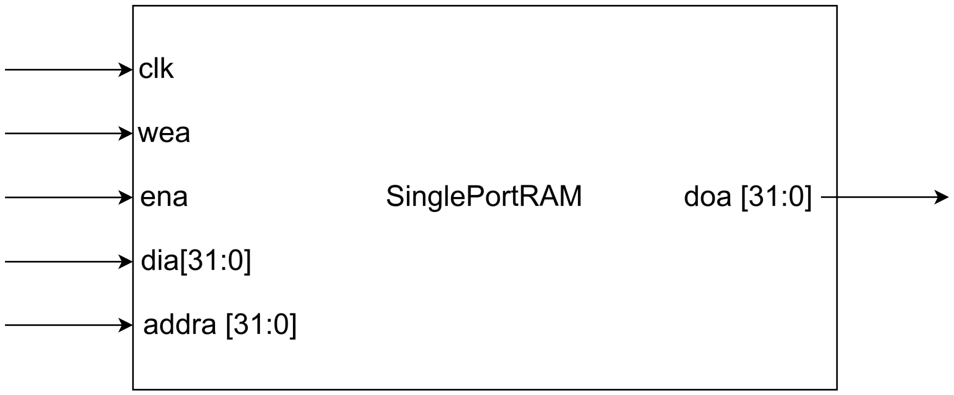


Fig. 9.1: single port block RAM ports

Table 9.1: Single Port Block RAM Ports

Port Name	Direction	Width	Description
clk	input	1	clock signal for the block RAM
wea	input	1	write enable signal for the block RAM **Important:** The ena signal must be high to write to the block RAM
ena	input	1	enable signal for the block RAM. Needs to be high for read and write operations
addra	input	32	address of the word to read or write
dia	input	32	data input for write operations
doa	output	32	data output for read operations

9.2 Dual Port Block RAM

9.2.1 Description

The dual port block RAM is a block RAM that has two ports for reading and writing data simultaneously. The RAM is a synchronous RAM that is write first.

9.2.2 Ports

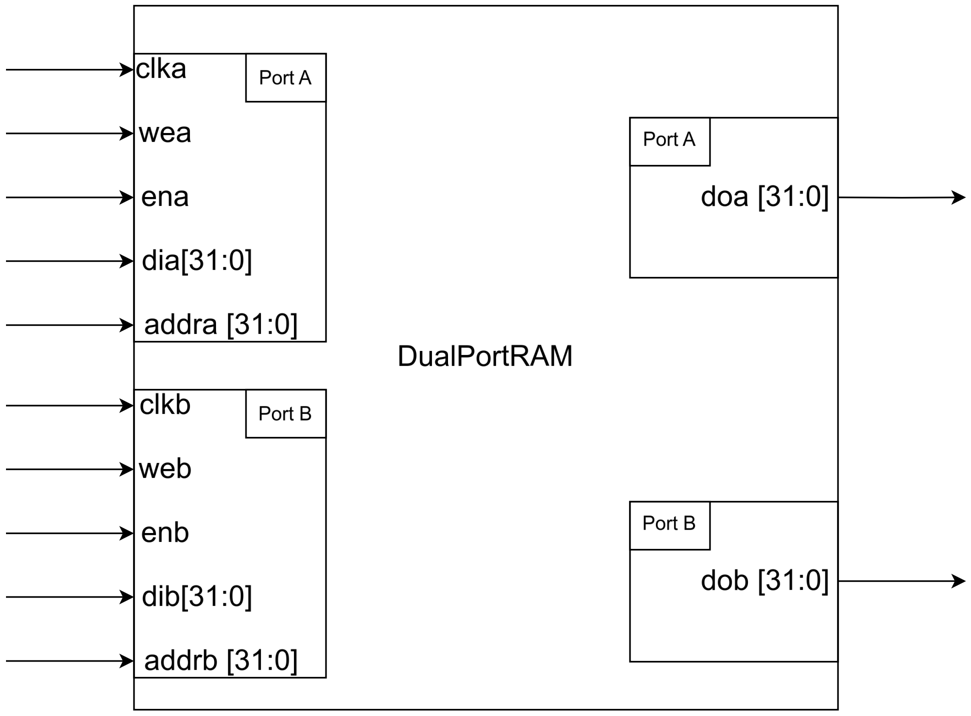


Fig. 9.2: dual port block RAM ports

Table 9.2: Dual Port Block RAM Ports

Port Name	Direction	Width	Description
Port A			
clka	input	1	clock signal for the first port of the block RAM
wea	input	1	write enable signal for the first port of the block RAM **Important:** The `ena` signal must be high to write to the block RAM
ena	input	1	enable signal for the first port of the block RAM. Needs to be high for read and write operations
addra	input	32	address of the word to read or write for the first port
dia	input	32	data input for write operations for the first port
doa	output	32	data output for read operations for the first port
Port B			
clkb	input	1	clock signal for the second port of the block RAM
web	input	1	write enable signal for the second port of the block RAM **Important:** The `enb` signal must be high to write to the block RAM
enb	input	1	enable signal for the second port of the block RAM. Needs to be high for read and write operations
addrb	input	32	address of the word to read or write for the second port
dib	input	32	data input for write operations for the second port
dob	output	32	data output for read operations for the second port

9.3 Dual Port Byte Enable Block RAM

9.3.1 Description

The dual port byte enable block RAM is a block RAM that has two ports for reading and writing data simultaneously. The RAM is a synchronous RAM that is write first. Additionally, the RAM can write to individual bytes of a word, reading is always done on a word basis.

9.3.2 Ports

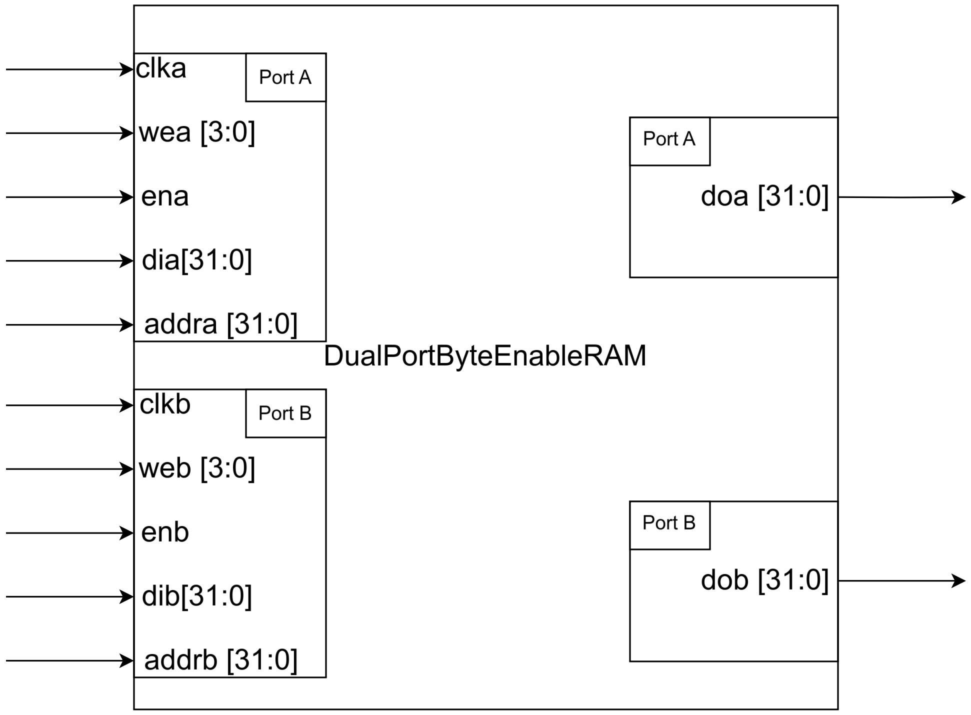


Fig. 9.3: dual port byte enable block RAM ports

Table 9.3: Dual Port Byte Enable Block RAM Ports

Port Name	Direction	Width	Description
Port A			
clka	input	1	clock signal for the first port of the block RAM
wea	input	4	write enable signal for the first port of the block RAM the write enable signal also works as a byte enable each bit represents a byte to write to **Important:** The `ena` signal must be high to write to the block RAM
ena	input	1	enable signal for the first port of the block RAM. Needs to be high for read and write operations
addra	input	32	address of the word to read or write for the first port
dia	input	32	data input for write operations for the first port
doa	output	32	data output for read operations for the first port
Port B			
clkb	input	1	clock signal for the second port of the block RAM
web	input	4	write enable signal for the second port of the block RAM the write enable signal also works as a byte enable each bit represents a byte to write to **Important:** The `enb` signal must be high to write to the block RAM
enb	input	1	enable signal for the second port of the block RAM. Needs to be high for read and write operations
addrb	input	32	address of the word to read or write for the second port
dib	input	32	data input for write operations for the second port
dob	output	32	data output for read operations for the second port

9.4 Using a template in a design

An implemented version of a BRAM can be found in the `hw/peripherals/bootram` directory.

To use a template in a design, the following steps need to be taken:

1. Copy the desired template from the `boards/bram_templates` directory to the cores, memus or peripherals directory of the design. The .h and .cpp files need to be copied into the top level directory of the module and the .v file needs to be copied into a `verilog` directory.
2. The blockram needs to be instantiated in the the SystemC Module in which it is used.
3. Add the .cpp file to the `MODULE_SRC` variable in the `Makefile` of the design.
4. Add the custom make targets needed for synthesis of the blockram to the `Makefile`

```
################ Variables ####################################################
VERILOG_TEMPLATE_PATH = $(SYSC_MODULE_DIR)/verilog
VERILOG_OUTPUT_PATH = $(SYSC_MODULE_DIR)/sc_build/verilog

VERILOG_TEMPLATES = $(wildcard $(VERILOG_TEMPLATE_PATH)/*_template.v)
VERILOG_TARGET_FILES = $(addprefix $(VERILOG_OUTPUT_PATH)/,$(subst _template.v,.v,
↪$(notdir $(VERILOG_TEMPLATES))))
```

(continues on next page)

(continued from previous page)

```
################ Special Targets ################################################


# extend the synthesis target from sysc-base
custom-verilog-copy:: $(VERILOG_TARGET_FILES)

$(VERILOG_OUTPUT_PATH)/%.v: $(VERILOG_TEMPLATE_PATH)/%_template.v $(PNS_CONFIG_MK)
↪$(CONFIG_MK)
    @mkdir --parents $(VERILOG_OUTPUT_PATH)
    @echo "### Generating $(notdir $@) ..."
    @sed \
    -e 's#{RAM_SIZE}#<variable>#g' \
    $< > $@
```

Note: The sed command needs to be adjusted to the specific template that is used. And the Variables that need to be replaced.

CHAPTER

TEN

APPENDIX

10.1 Building the GNU Toolchain

10.1.1 General Information

This chapter explains how to install a version of the GNU Toolchain that includes a version of GDB (GNU Debugger) in which the text user interface (tui) is enabled.

Prerequisites:

- libncurses-dev to be able to compile GDB with tui
- The packages listed here: RISC-V Collab Github
- a symbolic link "python" that points to "python3"

Information:

- GCC Version: `12.2.0`
- Buildtime with the `-j4` option: 22 minutes
- Size required for the repository: 11 GB
- Site required for installed Toolchain: 1.4 GB
- Installs the complete "riscv-gnu-toolchain" with an multilb newlib

10.1.2 Installation

1. Install libncurses (on debian):

```
$ sudo apt install libncurses-dev
```

2. If the "python" command is not available on your system you need to create a symbolic link named "python" pointing to the "python3" executable.
3. Download the source code from the GitHub repository.

```
$ git clone https://github.com/riscv/riscv-gnu-toolchain
```

4. Change directory into the newly downloaded repository.

```
$ cd riscv-gnu-toolchain
```

5. Checkout the Version "2023.01.04" (GCC 12.2.0) with the following command

```
$ git checkout 2023.01.04
```

6. Configuring the toolchain with the following command (remove the \from the command):

```
$ ./configure --prefix=/home/<username>/toolchain \
--with-multilib-generator="rv32i-ilp32--;rv32im-ilp32--;rv32ima-ilp32--"
```

Note: Note that the given prefix path is only a sugestion and can be changed by the user.

7. Compile and install the toolchain.

```
$ make
```

Note: Running the `make` command builds and installs the toolchain. If writing to the directory specified with `--prefix=` requires root privileges, these need to be given in order to successfully run the `make` command.

INDEX