luna-soc

Great Scott Gadget

May 07, 2024

DATASHEETS

CHAPTER

ONE

EPTRI - SOC CONTROLLER FOR THE LUNA USB DEVICE

General Description

`eptri` (endpoint-tri) is a three-interface CSR controller that allows a CPU or Wishbone design to control the endpoints of a LUNA USB Device.

Features

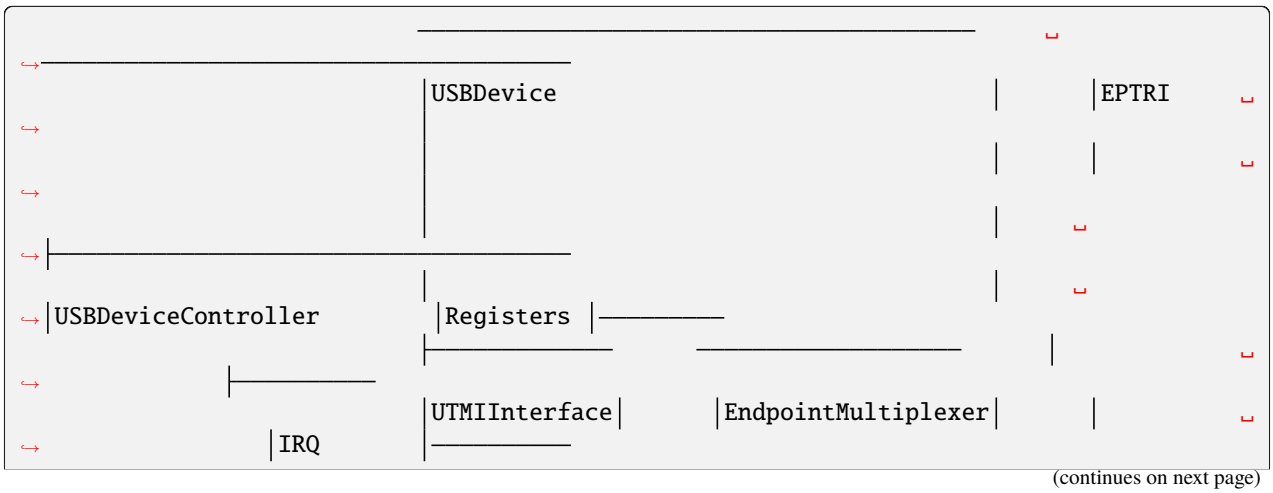
- CONTROL peripheral for managing device connection, reset and connection speed.
- SETUP interface peripheral for reading control transactions from the host.
- OUT interface peripheral for reading data transfers from the host.
- IN interface peripheral for writing data transactions to the host.

1.1 Introduction

1.1.1 Definitions

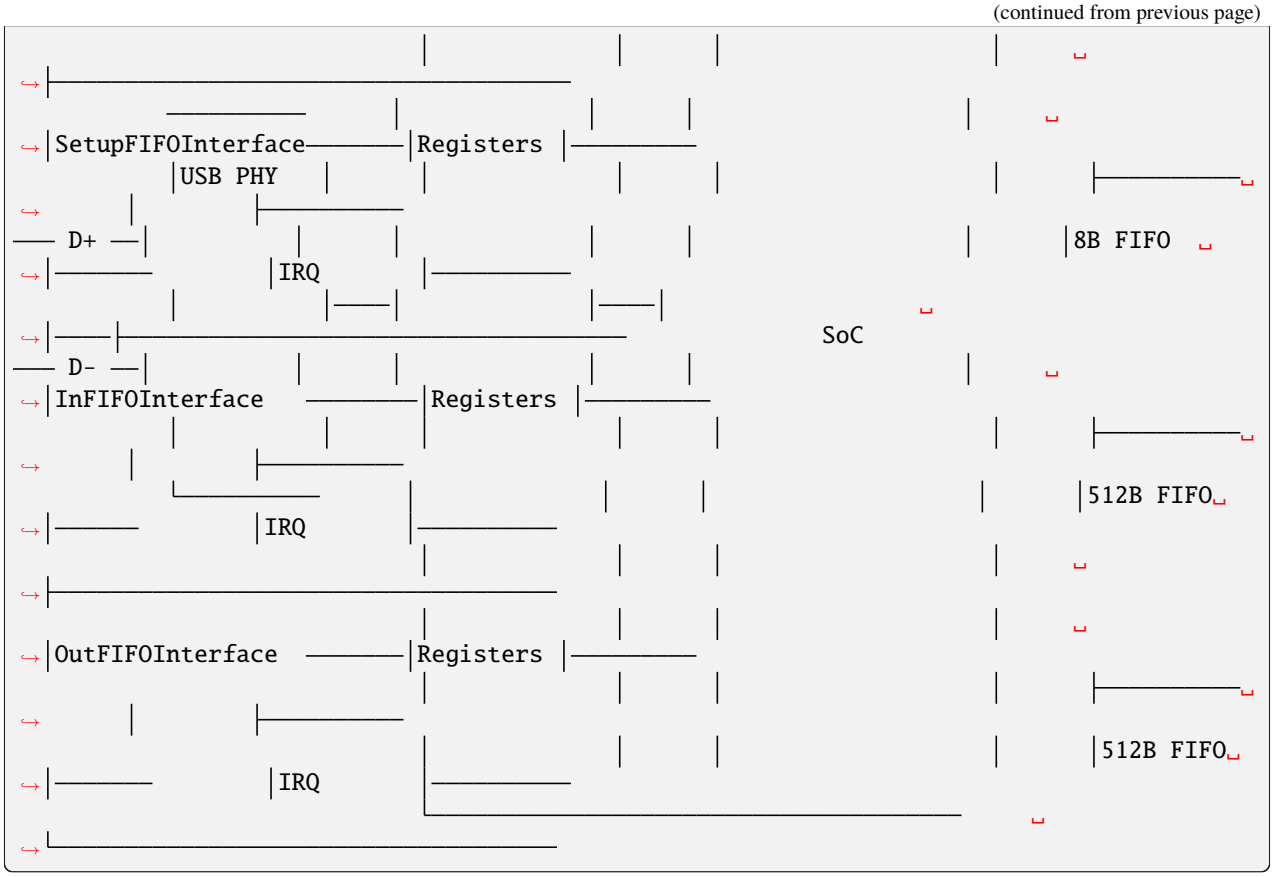
- *Controller* refers to the `eptri` controller as a whole, including all peripherals.
- *Peripheral* refers to `USBDeviceController`, `SetupFIFOInterface`, `InFIFOInterface` or `OutFIFOInterface`.

1.2 Block Diagram



(continues on next page)

(continued from previous page)



1.3 Usage

The exact details for integrating a LUNA eptri peripheral into a design will depend on the SoC platform you are using.

For example, using `luna-soc`:

```
class Top(Elaboratable):
    def __init__(self):
        # instantiate the SoC
        self.soc = LunaSoC(
            cpu=VexRiscv(reset_addr=0x00000000, variant="cynthion"),
            clock_frequency=int(60e6),
        )

        # instantiate the eptri device controller peripheral
        self.usb0 = USBDeviceController()

        # instantiate the three endpoint interface peripherals
        self.usb0_ep_control = SetupFIFOInterface()
        self.usb0_ep_in = InFIFOInterface()
        self.usb0_ep_out = OutFIFOInterface()

        # add the peripherals to the SoC
        self.soc.add_peripheral(self.usb0)
```

(continues on next page)

(continued from previous page)

```
        self.soc.add_peripheral(self.usb0_ep_control, as_submodule=False)
        self.soc.add_peripheral(self.usb0_ep_in, as_submodule=False)
        self.soc.add_peripheral(self.usb0_ep_out, as_submodule=False)

def elaborate(self, platform):
    m = Module()

    m.submodules.soc = self.soc

    # instantiate a LUNA USB device
    usb0_bus = platform.request(platform.default_usb_connection)
    usb0_device = USBDevice(bus=usb0_bus)

    # add the eptri endpoint interface peripherals to the LUNA USB device
    # as endpoint handlers
    usb0_device.add_endpoint(self.usb0_ep_control)
    usb0_device.add_endpoint(self.usb0_ep_in)
    usb0_device.add_endpoint(self.usb0_ep_out)

    # connect the eptri device controller to the LUNA USB device
    m.d.comb += self.usb0.attach(usb0_device)

    m.submodules.usb0_device = usb0_device

    return m
```

1.4 Registers

The `eptri` controller provides four sets of registers corresponding to each peripheral.

1.4.1 USBx - USBDeviceController

Table 1: USB0 Register Map

Offset	Rang	Access	Name	Description
0x00(	[0:0]	read-write	connect	Set this bit to '1' to allow the associated USB device to connect to a host.
0x00(	[1:0]	read-only	speed	Indicates the current speed of the USB device. 0 indicates High; 1 => Full, 2 => Low, and 3 => SuperSpeed (incl SuperSpeed+).
0x00(	[0:0]	read-write	low_speed_	Set this bit to '1' to force the device to operate at low speed.
0x00(	[0:0]	read-write	full_speed	Set this bit to '1' to force the device to operate at full speed.
0x001	[0:0]	read-only	status	usb0 status register field
0x001	[0:0]	read-write	pending	usb0 pending register field
0x001	[0:0]	read-write	enable	usb0 enable register field

1.4.2 USBx_EP_CONTROL - SetupFIFOInterface

Table 2: USB0_EP_CONTROL Register Map

Offset	Rang	Access	Nam	Description
0x00(	[7:0]	read-only	data	A FIFO that returns the bytes from the most recently captured SETUP packet. Reading a byte from this register advances the FIFO. The first eight bytes read from this conain the core SETUP packet.
0x00(	[0:0]	write-only	rese	Local reset control for the SETUP handler; writing a '1' to this register clears the handler state.
0x00(	[3:0]	read-only	epno	The number of the endpoint associated with the current SETUP packet.
0x00(	[0:0]	read-only	have	*1* iff data is available in the FIFO.
0x00	[0:0]	read-only	pend	*1* iff an interrupt is pending
0x00	[7:0]	read-write	addr	Controls the current device's USB address. Should be written after a SET_ADDRESS request is received. Automatically resets back to zero on a USB reset.
0x00	[0:0]	read-only	stat	usb0_ep_control status register field
0x00	[0:0]	read-write	pend	usb0_ep_control pending register field
0x00	[0:0]	read-write	enab	usb0_ep_control enable register field

1.4.3 `USBx_EP_IN` - InFIFOInterface

Table 3: USB0_EP_IN Register Map

Offset	Rang	Access	Nam	Description
0x00	[7:0]	write only	`data`	Write-only register. Each write enqueues a byte to be transmitted; gradually building a single packet to be transmitted. This queue should only ever contain a single packet; it is the software's responsibility to handle breaking requests down into packets.
0x00	[3:0]	read-write	`epno`	Contains the endpoint the enqueued packet is to be transmitted on. Writing this register marks the relevant packet as ready to transmit; and thus should only be written after a full packet has been written into the FIFO. If no data has been placed into the DATA FIFO, a zero-length packet is generated. Note that any IN requests that do not match the endpoint number are automatically NAK'd.
0x00	[0:0]	write only	`rese`	A write to this register clears the FIFO without transmitting.
0x00	[0:0]	read-write	`stal`	When this register contains '1', any IN tokens targeting *epno* will be responded to with a STALL token, rather than DATA or a NAK. For EP0, this register will automatically be cleared when a new SETUP token is received.
0x00	[0:0]	read-only	`idle`	This value is *1* if no packet is actively being transmitted.
0x00	[0:0]	read-only	`have`	This value is *1* if data is present in the transmit FIFO.
0x00	[0:0]	read-only	`pend`	*1* iff an interrupt is pending
0x00	[0:0]	read-write	`pid`	Contains the current PID toggle bit for the given endpoint.
0x00	[15:0	read-only	`nak`	Read-only register. Contains a bitmask of endpoints that have responded with a NAK since the last read of this register.
0x00	[0:0]	read-only	`stat`	usb0_ep_in status register field
0x00	[0:0]	read-write	`pend`	usb0_ep_in pending register field
0x00	[0:0]	read-write	`enab`	usb0_ep_in enable register field

1.4.4 USBx_EP_OUT - OutFIFOInterface

Table 4: USB0_EP_OUT Register Map

Offset	Rang	Access	Nam	Description
0x00	[7:0]	read-only	data	A FIFO that returns the bytes from the most recently captured OUT transaction. Reading a byte from this register advances the FIFO.
0x00	[3:0]	read-only	data	Register that contains the endpoint number associated with the data in the FIFO – that is, the endpoint number on which the relevant data was received.
0x00	[0:0]	write only	rese	Local reset for the OUT handler; clears the out FIFO.
0x00	[3:0]	read-write	epno	Selects the endpoint number to prime. This interface only allows priming a single endpoint at once– that is, only one endpoint can be ready to receive data at a time. See the *enable* bit for usage.
0x00	[0:0]	read-write	enab	Controls whether any data can be received on any primed OUT endpoint. This bit is automatically cleared on receive in order to give the controller time to read data from the FIFO. It must be re-enabled once the FIFO has been emptied.
0x00	[0:0]	write only	prim	Controls "priming" an out endpoint. To receive data on any endpoint, the CPU must first select the endpoint with the *epno* register; and then write a '1' into the prime and enable register. This prepares our FIFO to receive data; and the next OUT transaction will be captured into the FIFO. When a transaction is complete, the *enable* bit is reset; the *prime* is not. This effectively means that *enable* controls receiving on _any_ of the primed endpoints; while *prime* can be used to build a collection of endpoints willing to participate in receipt. Only one transaction / data packet is captured per *enable* write; repeated enabling is necessary to capture multiple packets.
0x00	[0:0]	read-write	stal	Controls STALL'ing the active endpoint. Setting or clearing this bit will set or clear STALL on the provided endpoint. Endpoint STALLs persist even after *epno* is changed; so multiple endpoints can be stalled at once by writing their respective endpoint numbers into *epno* register and then setting their *stall* bits.
0x00	[0:0]	read-only	have	*1* iff data is available in the FIFO.
0x00	[0:0]	read-only	pend	*1* iff an interrupt is pending
0x00	[7:0]	read-write	addr	Controls the current device's USB address. Should be written after a SET_ADDRESS request is received. Automatically resets back to zero on a USB reset.
0x00	[0:0]	read-write	pid	Contains the current PID toggle bit for the given endpoint.
0x00	[0:0]	read-only	stat	usb0_ep_out status register field
0x00	[0:0]	read-write	pend	usb0_ep_out pending register field
0x00	[0:0]	read-write	enab	usb0_ep_out enable register field

1.5 Interrupts

Each of the `eptri` peripherals can generate the following interrupts:

Table 5: USBDeviceController Registers

Interrupt	Peripheral	Description
USBx	USBDevice-Controller	Interrupt that triggers when the host issued a USB bus reset.
USBx_EP_CON	SetupFI-FOInterface	Interrupt that triggers when the host wrote a new SETUP packet to the bus.
USBx_EP_IN	InFIFOInter-face	Interrupt that triggers after the peripheral has written a data packet to the bus and read back a PID ACK response from the host.
USBx_EP_OUT	OutFIFOInt-erface	Interrupt that triggers when the peripheral has read a data packet from the host.

1.6 Programming Guide

The programming guide provides sequence diagrams that detail the steps required to perform the various operations supported by the `eptri` controller.

The following pseudo-code is used through-out to indicate register operations:

Table 6: Register Operations

Operation	Description
`r(PERIPHERAL.register)`	Read a value from the `register` belonging to `PERIPHERAL`.
`w(PERIPHERAL.register, bits)`	Write `bits` to the `register` belonging to `PERIPHERAL`.

1.6.1 Device Connection

1.6.2 Bus Reset

1.6.3 Control OUT Transfers

1.6.4 Control IN Transfers

1.6.5 Bulk OUT Transfers

1.6.6 Bulk IN Transfers

CHAPTER

TWO

LUNA_SOC

2.1 luna_soc package

2.1.1 Subpackages

luna_soc.gateware package

Subpackages

luna_soc.gateware.csr package

Subpackages

luna_soc.gateware.csr.usb2 package

Subpackages

luna_soc.gateware.csr.usb2.interfaces package

Submodules

luna_soc.gateware.csr.usb2.interfaces.eptri module

Implementation of a Triple-FIFO endpoint manager.

Equivalent (but not binary-compatbile) implementation of ValentyUSB's `eptri`.

For an example, see `examples/usb/eptri` or TinyUSB's `luna/dcd_eptri.c`.

class luna_soc.gateware.csr.usb2.interfaces.eptri.**InFIFOInterface**(**args*, *src_loc_at=0*, ***kwargs*)

Bases: *Peripheral*, Elaboratable

IN component of our *eptri*-equivalent interface.

Implements the FIFO that handles *eptri* IN requests. This FIFO collects USB data, and transmits it in response to an IN token. Like all *eptri* interfaces; it can handle only one pending packet at a time.

Attributes

interface: EndpointInterface
Our primary interface to the core USB device hardware.

`__init__`(*max_packet_size=512*)

Parameters

max_packet_size: int, optional
Sets the maximum packet size that can be transmitted on this endpoint. This should match the value provided in the relevant endpoint descriptor.

`elaborate`(*platform*)

`class luna_soc.gateware.csr.usb2.interfaces.eptri.OutFIFOInterface`(**args*, *src_loc_at=0*, ***kwargs*)

Bases: *Peripheral*, `Elaboratable`

OUT component of our *eptri*

Implements the OUT FIFO, which handles receiving packets from our host.

Attributes

interface: EndpointInterface
Our primary interface to the core USB device hardware.

`__init__`(*max_packet_size=512*)

Parameters

max_packet_size: int, optional
Sets the maximum packet size that can be transmitted on this endpoint. This should match the value provided in the relevant endpoint descriptor.

`elaborate`(*platform*)

`class luna_soc.gateware.csr.usb2.interfaces.eptri.SetupFIFOInterface`(**args*, *src_loc_at=0*, ***kwargs*)

Bases: *Peripheral*, `Elaboratable`

Setup component of our *eptri*-equivalent interface.

Implements the USB Setup FIFO, which handles SETUP packets on any endpoint.

This interface is similar to an *OutFIFOInterface*, but always ACKs packets, and does not allow for any flow control; as a USB device must always be ready to accept control packets. [USB2.0: 8.6.1]

Attributes

interface: EndpointInterface
Our primary interface to the core USB device hardware.

`elaborate`(*platform*)

Module contents

Submodules

luna_soc.gateware.csr.usb2.device module

Contains the organizing hardware used to add USB Device functionality to your own designs; including the core `USBDevice` class.

`class luna_soc.gateware.csr.usb2.device.USBDeviceController`(**args*, *src_loc_at=0*, ***kwargs*)

Bases: *Peripheral*, `Elaboratable`

SoC controller for a USBDevice.

Breaks our USBDevice control and status signals out into registers so a CPU / Wishbone master can control our USB device.

The attributes below are intended to connect to a USBDevice. Typically, they'd be created by using the .controller() method on a USBDevice object, which will automatically connect all relevant signals.

Attributes

connect: Signal(), output
High when the USBDevice should be allowed to connect to a host.

`attach`(*device: USBDevice*)

Returns a list of statements necessary to connect this to a USB controller.

The returned values makes all of the connections necessary to provide control and fetch status from the relevant USB device. These can be made either combinationally or synchronously, but combinational is recommended; as these signals are typically fed from a register anyway.

Parameters

device: USBDevice
The `USBDevice` object to be controlled.

`elaborate`(*platform*)

Module contents

Submodules

luna_soc.gateware.csr.base module

Peripheral helpers for LUNA devices.

class luna_soc.gateware.csr.base.**CSRBank**(*, *name=None*)

Bases: CSRBank

csr(*width*, *access*, *, *addr=None*, *alignment=None*, *name=None*, *src_loc_at=0*, *desc=None*)

Request a CSR register.

Parameters

width
[int] Width of the register. See amaranth_soc.csr.Element.

access
[Access] Register access mode. See amaranth_soc.csr.Element.

addr
[int] Address of the register. See amaranth_soc.csr.Multiplexer.add().

alignment
[int] Register alignment. See amaranth_soc.csr.Multiplexer.

name
[str] Name of the register. If None (default) the name is inferred from the variable name this register is assigned to.

desc
[str] Optional. Documentation for the given register. Used to generate register documentation automatically.

Return value

An instance of amaranth_soc.csr.Element.

class luna_soc.gateware.csr.base.**EventSource**(*, *mode='level'*, *name=None*, *src_loc_at=0*)

Bases: object

Event source.

Parameters

mode
["level", "rise", "fall"] Trigger mode. If "level", a notification is raised when the stb signal is high. If "rise" (or "fall") a notification is raised on a rising (or falling) edge of stb.

name
[str] Name of the event. If None (default) the name is inferred from the variable name this event source is assigned to.

Attributes

name
[str] Name of the event

mode
["level", "rise", "fall"] Trigger mode.

stb
[Signal, in] Event strobe.

class luna_soc.gateware.csr.base.**Peripheral**(*name=None*, *src_loc_at=1*)

Bases: Peripheral

csr_bank(*, *name=None*, *addr=None*, *alignment=None*, *desc=None*)

Request a CSR bank.

Arguments

name
[str] Optional. Bank name.

addr
[int or None] Address of the bank. If None, the implicit next address will be used. Otherwise, the exact specified address (which must be a multiple of 2 ** max(alignment, bridge_alignment)) will be used.

alignment
[int or None] Alignment of the bank. If not specified, the bridge alignment is used. See amaranth_soc.csr.Multiplexer for details.

desc
[str] Optional. Documentation for the given CSR bank.

Return value

An instance of *CSRBank*.

event(*, *mode='level'*, *name=None*, *src_loc_at=0*, *desc=None*)

Request an event source.

Arguments

desc
[str] Optional. Documentation for the given event.

See *EventSource* for details.

Return value

An instance of *EventSource*.

class luna_soc.gateware.csr.base.**PeripheralBridge**(**args*, *src_loc_at=0*, ***kwargs*)

Bases: Elaboratable

Peripheral bridge.

A bridge providing access to the registers and windows of a peripheral, and support for interrupt requests from its event sources.

Event managment is performed by an InterruptSource submodule.

Parameters

periph
[*Peripheral*] The peripheral whose resources are exposed by this bridge.

data_width
[int] Data width. See amaranth_soc.wishbone.Interface.

granularity
[int or None] Granularity. See amaranth_soc.wishbone.Interface.

features
[iter(str)] Optional signal set. See amaranth_soc.wishbone.Interface.

alignment
[int] Resource alignment. See amaranth_soc.memory.MemoryMap.

Attributes

bus
[amaranth_soc.wishbone.Interface] Wishbone bus providing access to the resources of the peripheral.

irq
[IRQLine, out] Interrupt request. It is raised if any event source is enabled and has a pending notification.

elaborate(*platform*)

luna_soc.gateware.csr.gpio module

class luna_soc.gateware.csr.gpio.**GpioPeripheral**(**args*, *src_loc_at=0*, ***kwargs*)

Bases: *Peripheral*, Elaboratable

GPIO peripheral.

elaborate(*platform*)

luna_soc.gateware.csr.led module

class luna_soc.gateware.csr.led.**LedPeripheral**(**args*, *src_loc_at=0*, ***kwargs*)

Bases: *Peripheral*, Elaboratable

Example peripheral that controls the board's LEDs.

elaborate(*platform*)

luna_soc.gateware.csr.sram module

luna_soc.gateware.csr.uart module

class luna_soc.gateware.csr.uart.**UARTPeripheral**(**args*, *src_loc_at=0*, ***kwargs*)

Bases: *Peripheral*, Elaboratable

Asynchronous serial transceiver peripheral.

See amaranth_stdio.serial.AsyncSerial for details.

CSR registers

divisor
: [read/write] Clock divisor.

rx_data
: [read-only] Receiver data.

rx_rdy
: [read-only] Receiver ready. The receiver FIFO is non-empty.

rx_err
: [read-only] Receiver error flags. See amaranth_stdio.serial.AsyncSerialRX for layout.

tx_data
: [write-only] Transmitter data.

tx_rdy
: [read-only] Transmitter ready. The transmitter FIFO is non-full.

Events

rx_rdy
: [level-triggered] Receiver ready. The receiver FIFO is non-empty.

rx_err
: [edge-triggered (rising)] Receiver error. Error cause is available in the `rx_err` register.

tx_mty
: [edge-triggered (rising)] Transmitter empty. The transmitter FIFO is empty.

Parameters

rx_depth
: [int] Depth of the receiver FIFO.

tx_depth
: [int] Depth of the transmitter FIFO.

divisor
: [int] Clock divisor reset value. Should be set to `int(clk_frequency // baudrate)`.

divisor_bits
: [int] Optional. Clock divisor width. If omitted, `bits_for(divisor)` is used instead.

data_bits
: [int] Data width.

parity
: ["none", "mark", "space", "even", "odd"] Parity mode.

pins
: [Record] Optional. UART pins. See `amaranth_boards.resources.UARTResource`.

Attributes

bus
: [`amaranth_soc.wishbone.Interface`] Wishbone bus interface.

irq
: [IRQLine] Interrupt request line.

`elaborate`(*platform*)

Module contents

Submodules

luna_soc.gateware.soc module

Module contents

luna_soc.generate package

Submodules

luna_soc.generate.genc module

Generate a C library for SoC designs.

class luna_soc.generate.genc.**GenC**(*soc: LunaSoC*)

Bases: object

generate_c_header(*macro_name='SOC_RESOURCES'*, *file=None*, *platform_name='Generic Platform'*)

Generates a C header file that simplifies access to the platform's resources.

Parameters

- **macro_name – Optional. The name of the guard macro for the C header, as a string without spaces.**
- **file – Optional. If provided, this will be treated as the file= argument to the print() –** function. This can be used to generate file content instead of printing to the terminal.

generate_ld_script(*file=None*)

Generates an ldscript that holds our primary RAM and ROM regions.

Parameters

file – Optional. If provided, this will be treated as the file= argument to the print() – function. This can be used to generate file content instead of printing to the terminal.

luna_soc.generate.generate module

Generate programming support files from SoC designs.

class luna_soc.generate.generate.**Generate**(*soc: LunaSoC*)

Bases: object

c_header(*macro_name='SOC_RESOURCES'*, *file=None*, *platform_name='Generic Platform'*)

Generates a C header file that simplifies access to the platform's resources.

Parameters

- **macro_name** – Optional. The name of the guard macro for the C header, as a string without spaces.
- **file** – Optional. If provided, this will be treated as the file= argument to the print() function. This can be used to generate file content instead of printing to the terminal.

ld_script(*file=None*)

Generates an ldscript that holds our primary RAM and ROM regions.

Parameters

file – Optional. If provided, this will be treated as the file= argument to the print() function. This can be used to generate file content instead of printing to the terminal.

memory_x(*file=None*)

Generates a svd file for the given SoC that can be used by external tools such as 'svdrust'.

Parameters

file – Optional. If provided, this will be treated as the file= argument to the print() function. This can be used to generate file content instead of printing to the terminal.

svd(*file=None*)

Generates a svd file for the given SoC that can be used by external tools such as 'svdrust'.

Parameters
file – Optional. If provided, this will be treated as the file= argument to the print() function. This can be used to generate file content instead of printing to the terminal.

luna_soc.generate.genrust module

Generate Rust support files for SoC designs.

class luna_soc.generate.genrust.**GenRust**(*soc: LunaSoC*)

Bases: object

generate_memory_x(*file=None*)

Generate a memory.x file for the given SoC

luna_soc.generate.gensvd module

Generate a SVD file for SoC designs.

class luna_soc.generate.gensvd.**GenSVD**(*soc: LunaSoC*)

Bases: object

generate_svd(*file=None*, *vendor='amaranth-soc'*, *name='soc'*, *description=None*)

Generate a svd file for the given SoC

luna_soc.generate.introspect module

Introspection tools for SoC designs.

class luna_soc.generate.introspect.**Introspect**(*soc: LunaSoC*)

Bases: object

irq_for_peripheral_window(*target_peripheral_window: MemoryMap*)

Returns any interrupt associated with the given peripheral.

Returns
if the given peripheral has an interrupt; or None, None if not

Return type
irqno, peripheral

log_resources()

Logs a summary of our resource utilization to our running logs.

main_ram_address()

Returns the address of the main system RAM.

range_for_peripheral(*target_peripheral: Peripheral*)

Returns size information for the given peripheral.

Returns
if the given size is known; or None, None if not

Return type
addr, size

`resources()`

Creates an iterator over each of the device's addressable resources.

Yields (MemoryMap, ResourceInfo, address, size) for each resource.

Module contents

luna_soc.util package

Submodules

luna_soc.util.readbin module

luna_soc.util.readbin.**get_boot_address**(*filename_or_regions*, *offset=0*)

luna_soc.util.readbin.**get_mem_data**(*filename_or_regions*, *data_width=32*, *endianness='big'*, *mem_size=None*, *offset=0*)

luna_soc.util.readbin.**get_mem_regions**(*filename_or_regions*, *offset*)

Module contents

2.1.2 Submodules

2.1.3 luna_soc.top_level_cli module

luna_soc.top_level_cli.**build**(*args*, *fragment*, *platform*, *build_dir*)

Top-level build command. Invokes the build steps for each artifact to be generated.

luna_soc.top_level_cli.**top_level_cli**(*fragment*, **pos_args*, ***kwargs*)

Runs a default CLI that assists in building and running SoC gateware.

If the user's options resulted in the board being programmed, this returns the fragment that was programmed onto the board. Otherwise, it returns None.

Parameters
fragment – The design to be built; or a callable that returns a fragment, – such as a Elaborable type. If the latter is provided, any keyword or positional arguments not specified here will be passed to this callable.

2.1.4 Module contents

PYTHON MODULE INDEX

l

INDEX

Symbols

A

B

C

E

G

I

L

M

O

P

R

S

T

U