Azadi-SoC Design Specification

https://github.com/merledu/azadi-soc.git

Version 0.2-draft-20221223

MERL

Accelerating Engineering Innovation

Table of contents

1. Introduction

Azadi SoC is based on a 32-bit three-stage pipelined RISC-V processor (lowrisc/ibex [1] + F ext) with a Tile-Link (UL) bus protocol to enable peripherals/memory communication with the core. The RISC-V processor (RV32IMFC) supports I base ISA, M multiply and divide F single precision floating point, and C compressed instructions. Below is the top-level diagram of the SoC.

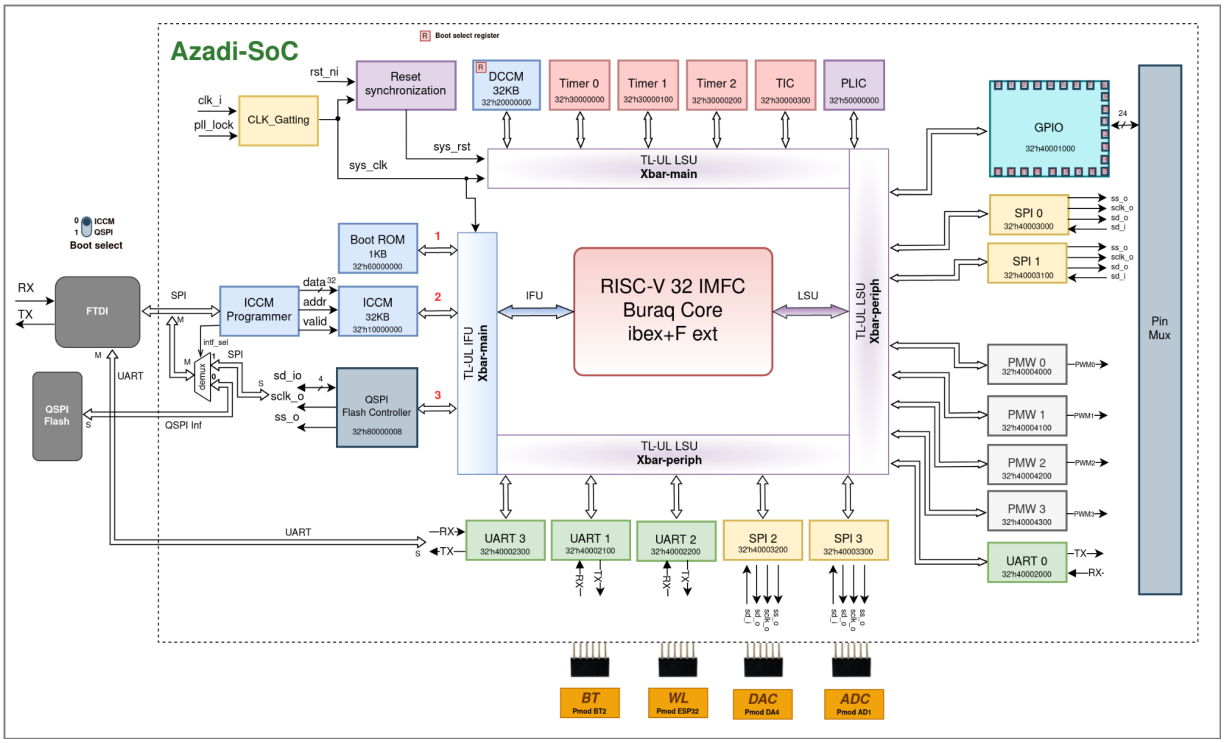

Figure 1.1. Azadi-SoC Block level diagram

2. SoC Overview

2.1. Reset and Clock Management

There is a single domain for clock and reset in the SoC and active low asynchronous reset is used inside the whole chips.
There is no PLL inside the core hence an input clock along with a pll_lock signal is coming into the SoC. To provide a glitches-free clock we are using clock gating at the input that uses the pll_lock signal to enable the clock gating cell. Once the pll_lock is enabled the output clock is provided to the whole SoC. The CG schematic is shown in figure 2.1.

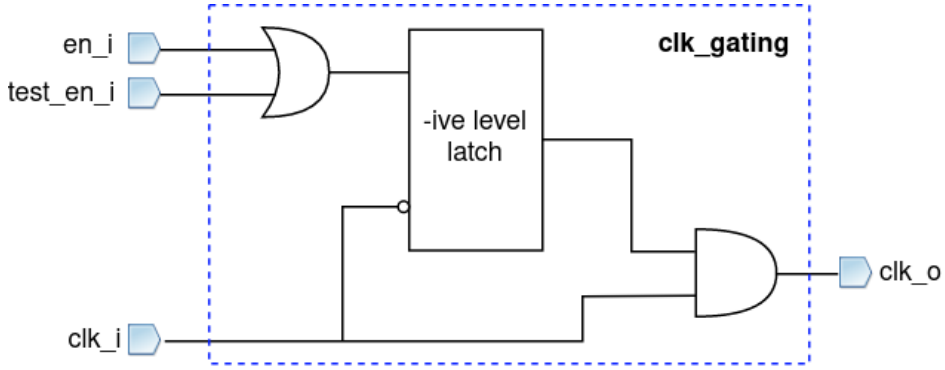

Figure 2.1. Clock-gating cell schematic

To handle the asynchronous reset we are using the dual flip-flop reset synchronization scheme that allows resetting assertion asynchronously and de-assertion synchronously. This scheme will be useful to escape from the metastability state occurring issue among flip flops.

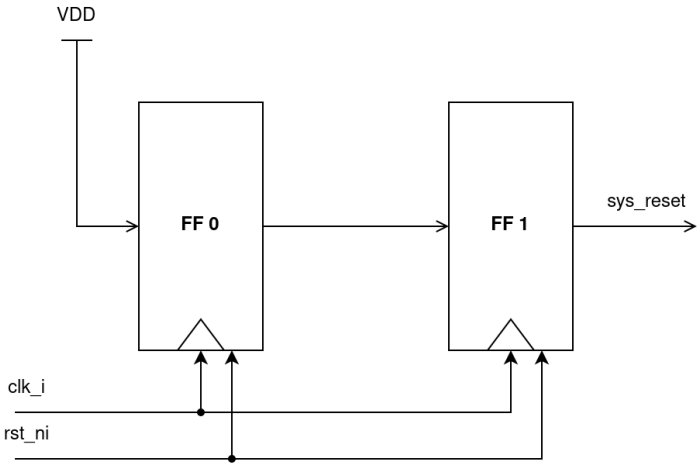

Figure 2.2. Dual flip-flop-based rest synchronizer

2.2. Memories

The Azadi-SoC provides 4 interfaces for memory integration including 2x 32KB memories each for Instruction and Data memory and a 1KB ROM integrated with the SoC which helps SoC in

the booting process. A QSPI interface is provided to use the external flash memory with the SoC for storing large programs in it.

2.3. Peripherals

The Azadi-SoC supports a vast number of peripherals that include Timers x3, PWM x4, UART x4, SPI x4, GPIO x24 pins, and PLIC to control external peripherals interrupts. SPI 2 and SPI 3 are dedicated to the use of DAC and ADC respectively. Whereas UART 1 and UART 2 are for Bluetooth and Wifi PMOD connections respectively. UART 3 is dedicated to the use of off-chip communication with the Host using FTDI and can also be used as a peripheral as well.
The remaining peripherals including SPI 0, SPI 1, UART 0, and PWM 0-3 are pin muxed with the first 18 GPIO pins, and the remaining GPIO pins from 19-23 can be used directly. The pin muxing is configurable through software.

2.4. Programming Interface

The SoC can be programmed through the SPI interface and there are two modules named iccm_programmer and qspi_flash_controller used to receive the input data from the master SPI and transmit it to the iccm_adapter and qspi_flash in a packed array of 32 bit, respectively.
The external toggle switch will tell which interface should be enabled for loading the instructions.

3. Ibex Core Specification

The specification of Ibex can be found on the official documentation site.

3.1 Changes made in Ibex for Azadi-SoC

We have added a Floating Point Unit with ibex and changed the PC starting offset address in the instruction fetch unit using a Verilog define statement (`ifdef AZADI) because we are handling interrupts in direct mode whereas ibex handles in vector mode.
Ibex comes with vast mico-architectural features but we enabled a few of them shown in table 3.1.

Table 3.1. Supported features of Ibex in Azadi-SoC

Ibex Features	**Support in Azadi**
Physical Memory Protection	
RV32E	
RV32M	✔
RV32B	
BranchTargetALU	✔
WriteBackStage	✔
ICahche	
ICacheECC	
BranchPredictor	✔
RISC-V Debug	
SecureIbex	

The Floating Point Unit (FPU) is integrated with ibex as a separate pipeline and the communication is done using ready/valid protocol. We support only Single Precision or F extension in Azadi and FPU is integrated with Ibex through a parameter that means it can simply be turned on and off without affecting the other implementations. In ibex 3 additional Control and Status Registers (CSR) are added in the ibex_cs_rsgister module along with one additional floating point flip-flop-based (32x32) register file.

4. FPU Documentation

https://github.com/openhwgroup/cvfpu
FPnew is a parametric floating-point unit that supports standard RISC-V operations as well as trans precision formats, written in SystemVerilog.

4.1. Current Configurations of FPU

Table 4.1 shows the current FPU configuration used in Azadi.

Table 4.1. FPU supported configuration in Azadi

Parameter Name	**Value**
Features	RV32F
Width	*32*
EnableVectors	*0*
EnableNaNBox	*1*
FpFmtMask	*5'b10000*
IntFmtMask	*4'b0010*
Implementation	DEFAULT_NOREG
PipeRegs	*0*

4.2. Top-Level Interface

The top-level module of the FPU is `fpnew_top` and its interface is further described in this section. FPnew uses a synchronous interface using handshaking to transfer data into and out of the FPU. All array types are packed due to poor support of unpacked arrays in some EDA tools. SystemVerilog interfaces are not used due to poor support in some EDA tools.

4.3. Parameters

The configuration parameters define the behavior of the FPU. These parameters are based on the structure data types present in the fpnew_pkg.

Table 4.2. Available parameters in fpnew

Parameter Name	**Description**
Features	Specifies the features of the FPU. i.e. Supported formats and operations.

Implementation	Allows control of how the above features are implemented, such as the number of pipeline stages and architecture of subunits
TagType	The SystemVerilog data type of the operation tag

4.4. Ports

FPnew ports are declared using custom data types that are made using structures and enums. Table 4.3 show the fpnew_top (the top-module) module's IO ports where W represents the datapath width of the FPU.

Table 4.3. Ports list of the top module of fpnew

Port Name	**Direction**	**Type**	**Description**
clk_i	in	logic	Clock, rising-edge triggered
rst_ni	In	logic	Asynchronous active low reset
operands_i	in	logic [2:0][W-1:0]	Three operands
rnd_mode_i	in	roundmode_e	Floating-point rounding mode
op_i	in	operation_e	Operation select
op_mod_i	in	logic	Operation modifier
src_fmt_i	in	fp_format_e	Source FP format
dst_fmt_i	in	fp_format_e	Destination FP format
int_fmt_i	in	int_format_e	Integer format
vectorial_op_i	in	logic	Vectorial operation select
tag_i	in	TagType	Operation tag input
in_valid_i	in	logic	Input data valid
in_ready_o	out	logic	Input interface ready
flush_i	in	logic	Synchronous pipeline reset
result_o	out	logic[W-1:0]	Result
status_o	out	status_t	RISC-V floating-point status flags fflags
tag_o	out	TagType	Operation tag output

out_valid_o	out	logic	Output data valid
out_ready_i	in	logic	Output interface ready
busy_o	out	logic	FPU operation in flight

4.5. Data Types

The following custom data types and enumerations are used in ports of the FPU and are defined in fpnew_pkg. Default values from the package are listed.

- roundmode_e - FP Rounding Mode

Enumeration of type logic [2:0] holding available rounding modes, encoded for use in RISC-V cores

Table 4.4. Supported rounding modes

Enumerator	**Value**	**Rounding Mode**
RNE	3'b000	To nearest, tie to even
RTZ	3'b001	Toward zero
RDN	3'b010	Toward negative infinity
RUP	3'b011	Toward positive infinity
RMM	3'b100	To nearest, tie away from zero
DYN	3'b111	RISC-V Dynamic RM, invalid if passed to operations

- operation_e - FP Operation

Enumeration of type logic [3:0] holding the FP operation. The operation modifier op_mod_i can change the operation carried out. Unless noted otherwise, the first operand `op[0]` is used for the operation.

Table 4.5. Supported operations

Enumerator	**Modifier**	**Operation**
FMADD	0	Fused multiply-add ((op[0] * op[1]) + op[2])
FMADD	1	Fused multiply-subtract ((op[0] * op[1]) - op[2])
FMNSUB	0	Negated fused multiply-subtract (-(op[0] * op[1]) + op[2])
FMNSUB	1	Negated fused multiply-add (-(op[0] * op[1]) - op[2])

ADD	0	Addition (op[1] + op[2]) *note the operand indices*
ADD	1	Subtraction (op[1] - op[2]) *note the operand indices*
MUL	0	Multiplication (op[0] * op[1])
DIV	0	Division (op[0] / op[1])
SQRT	0	Square root
SGNJ	0	Sign injection, operation encoded in rounding mode RNE: op[0] with sign(op[1]) RTZ: op[0] with ~sign(op[1]) RDN: op[0] with sign(op[0]) ^ sign(op[1]) RUP: op[0] (passthrough)
SGNJ	1	As above, but the result is sign-extended instead of NaN-Boxed
MINMAX	0	Minimum / maximum, operation encoded in rounding mode RNE: minimumNumber(op[0], op[1]) RTZ: maximumNumber(op[0], op[1])
CMP	0	Comparison, operation encoded in rounding mode RNE: op[0] <= op[1] RTZ: op[0] < op[1] RDN: op[0] == op[1]
CLASSIFY	0	Classification, returns RISC-V classification block
F2F	0	FP to FP cast, formats given by src_fmt_i and dst_fmt_i
F2I	0	FP to signed integer cast, formats given by src_fmt_i and int_fmt_i
F2I	1	FP to unsigned integer cast, formats given by src_fmt_i and int_fmt_i
I2F	0	Signed integer to FP cast, formats given by int_fmt_i and dst_fmt_i
I2F	1	Unsigned integer to FP cast, formats given by int_fmt_i and dst_fmt_i

- fp_format_e - FP Formats

Enumeration of type logic [2:0] holding the supported FP formats.

Table 4.6. Supported floating point formats

Enumerator	Format	Width	Exp. Bits	Man. Bits
FP32	IEEE binary32	32 bit	8	23
FP64	IEEE binary64	64 bit	11	52
FP16	IEEE binary16	16 bit	5	10

The following global parameters associated with FP formats are set in `fpnew_pkg`:

```
localparam int unsigned NUM_FP_FORMATS = 5;
localparam int unsigned FP_FORMAT_BITS = $clog2(NUM_FP_FORMATS);
```

- int_format_e - Integer Formats

Enumeration of type `logic [1:0]` holding the supported integer formats.

Table 4.7. Supported Integerformats

Enumerator	Width
INT16	16 bit
INT32	32 bit
INT64	64 bit

The following global parameters associated with integer formats are set in `fpnew_pkg`:

```
localparam int unsigned NUM_INT_FORMATS = 4;
localparam int unsigned INT_FORMAT_BITS = $clog2(NUM_INT_FORMATS);
```

- status_t - FP Status Flags

Packed struct containing the five FP status flags as `logic` in order MSB to LSB:

Table 4.8. Floating point exceptional flags

Member	Description
NV	Invalid operation
DZ	Division by zero
OF	Overflow
UF	Underflow

NX	Inexact operation

4.6. NaN-Boxing

RISC-V mandates so-called NaN-boxing of all FP values in formats that are narrower than the widest available format in the system. This means that all unused high-order bits of narrow formats must be set to `'1`, otherwise the value is considered invalid (a NaN).
Checks for whether input values are properly NaN-boxed are enabled by default but can be turned off. Narrow FP output values from the FPU are always NaN-boxed. Narrow integer output values from the FPU are sign-extended, even if unsigned.

4.7 Handshake Interface

Both the input and output side of FPnew feature a `valid`/`ready` handshake interface which controls the flow of data into and out of the FPU. The handshaking protocol is similar to the ones used in common protocols such as AXI:

- An asserted `valid` signal that data on the corresponding interface is valid and stable.
- Once `valid` is asserted, it must not be disasserted until the handshake is complete.
- An asserted `ready` signal that the interface can process data on the following rising clock edge.
- Once `valid` and `ready` are asserted during a rising clock edge, the transaction is complete.
- After a completed transaction, `valid` may remain asserted to provide new data for the next transfer.
- The protocol direction is top-down. `ready` may depend on `valid` but `valid` must not depend on `ready`.

4.8 Operation Tags

Operation tags are metadata accompanying an operation and can be used to link results back to the operations that produced them. Tags traverse the FPU without being modified, but always stay in sync with the operation they were issued with. Tags are an optional feature of FPnew and can be controlled by setting the TagType parameter as needed (usually a packed vector of logic, but can be any type). In order to disable the use of tags, set TagType to logic (the default value), and bind the tag_i port to a static value. Furthermore, ensure that your synthesis tool removes static registers.

4.9 Configuration

Main configuration of the FPU is done through parameters on the `fpnew_top` module.
A default selection of formats and features is defined in the package and can be controlled through these parameters.
Furthermore, the project package fpnew_pkg can be modified to provide even more custom formats to the FPU.

4.9.1 Configuration Parameters

Features - Feature set of the FPU
The Features parameter is used to configure the available formats and special features of the FPU. It is of type fpu_features_t which is defined as

```
typedef struct packed {
  int unsigned Width;
  logic        EnableVectors;
  logic        EnableNanBox;
  fmt_logic_t  FpFmtMask;
  ifmt_logic_t IntFmtMask;
} fpu_features_t;
```

The fields of this struct behave as follows:

4.9.2. Width - Datapath Width

Specifies the width of the FPU datapath and of the input and output data ports (operands_i/result_o). It must be larger or equal to the width of the widest enabled FP and integer format.
Default: 64

4.9.3. EnableVectors - Vectorial Hardware Generation

Controls the generation of packed-SIMD computation units in the FPU. If set to 1, vectorial execution units will be generated for all FP formats that are narrower than Width in order to fill up the datapath width. For example, given Width = 64, there will be four execution units for every operation on 16-bit FP formats.
Default: 1'b1

4.9.4. EnableNanBox - NaN-Boxing Check Control

Controls whether input value NaN-boxing is enforced. If set to 1, all values of FP formats that are narrower than `Width` will be considered NaN unless all unused high-order bits are set to 1. Output FP values are always NaN-boxed, regardless of this setting.
Default: 1'b1

4.9.5. FpFmtMask - Enabled FP Formats

The FpFmtMask parameter is of type fmt_logic_t which is an array holding one logic bit per FP format from fp_format_e, in ascending order.

```
typedef logic [0:NUM_FP_FORMATS-1] fmt_logic_t; // Logic indexed by FP
format
```

If a bit in FpFmtMask is set, FPU hardware for the corresponding format is generated.

Otherwise, synthesis tools can optimize away any logic associated with this format, and operations on the format yield undefined results.
Default: '1 (all enabled)

4.9.6. IntFmtMask - Enabled Integer Formats

The IntFmtMask parameter is of type ifmt_logic_t which is an array holding one logic bit per integer format from int_format_e, in ascending order.

```
typedef logic [0:NUM_INT_FORMATS-1] ifmt_logic_t; // Logic indexed by
integer format
```

If a bit in IntFmtMask is set, FPU hardware for the corresponding format is generated. Otherwise, synthesis tools can optimize away any logic associated with this format, and operations on the format yield undefined results.
Default: '1 (all enabled)

4.10. Implementation - Implementation Options

The FPU is divided into four operation groups, ADDMUL, DIVSQRT, NONDOMP, and CONV. The Implementation parameter controls the implementation of these operation groups.
It is of type fpu_implementation_t which is defined as

```
typedef struct packed {
  opgrp_fmt_unsigned_t   PipeRegs;
  opgrp_fmt_unit_types_t UnitTypes;
  pipe_config_t          PipeConfig;
} fpu_implementation_t;
```

The fields of this struct behave as follows:

4.10.1. PipeRegs - Number of Pipelining Stages

The PipeRegs parameter is of type opgrp_fmt_unsigned_t which is an array of arrays, holding for each operation group for each format an unsigned value, in ascending order.

```
typedef logic [0:NUM_FP_FORMATS-1][31:0] fmt_unsigned_t; // Array of
unsigned indexed by FP format
typedef fmt_unsigned_t [0:NUM_OPGROUPS-1] opgrp_fmt_unsigned_t; // Array of
format-specific unsigned indexed by operation group
```

This parameter sets a number of pipeline stages to be inserted into the computational units per operation group, per FP format. As such, latencies for different operations and different formats can be freely configured.
Default: '{default: 0} (no pipelining - all operations combinatorial)

4.10.2. UnitTypes - HW Unit Implementation

The UnitTypes parameter is of type opgrp_fmt_unit_types_t which is an array of arrays, holding for each operation group for each format an enumeration value, in ascending order.

```
typedef unit_type_t [0:NUM_FP_FORMATS-1] fmt_unit_types_t; // Array of unit
types indexed by format
typedef fmt_unit_types_t [0:NUM_OPGROUPS-1] opgrp_fmt_unit_types_t; //
Array of format-specific unit types indexed by opgroup
```

The unit type unit_type_t is an enumeration of type logic [1:0] holding the following implementation options for a particular hardware unit:

Table 4.9. Parameters to configure operational blocks

Enumerator	**Description**
DISABLED	No hardware units will be generated for this format
PARALLEL	One hardware unit per format will be generated
MERGED	One combined multi-format hardware unit will be generated for all formats selecting MERGED

The `UnitTypes` parameter allows control resources used for the FPU by either removing operation units for certain formats and operations or merging multiple formats into one. Currently, the following unit types are available for the FPU operation groups:

Table 4.10. Available parameters for hardware block generation

	ADDMUL	**DIVSQRT**	**NONCOMP**	**CONV**
PARALLEL	✔		✔	
MERGED	✔	✔		✔

```
Default:
'{'{default: PARALLEL}, // ADDMUL
  '{default: MERGED},   // DIVSQRT
  '{default: PARALLEL}, // NONCOMP
  '{default: MERGED}}   // CONV
```

(all formats within operation group use the same type)

4.10.3. PipeConfig - Pipeline Register Placement

The PipeConfig parameter is of type pipe_config_t and controls register placement in operational units. The requested number of registers is placed in predefined locations within the units according to the PipeConfig parameter. For best results, we **strongly** encourage the use of automatic retiming options in synthesis tools to optimize the pre-placed pipeline registers.

The configuration pipe_config_t is an enumeration of type logic [1:0] holding the following implementation options for the pipelines in operational units:

Table 4.11. Pipeline configuration options

Enumerator	Description
BEFORE	All pipeline registers are inserted at the inputs of the operational unit
AFTER	All pipeline registers are inserted at the outputs of the operational unit
INSIDE	All registers are inserted at roughly the middle of the operational unit (if not possible, BEFORE)
DISTRIBUTED	Registers are evenly distributed to INSIDE, BEFORE, and AFTER (if no INSIDE, all BEFORE)

4.11. Architecture

The exact architecture of FPnew depends on the configuration through parameters. The main architectural traits as well as the effect of some parameters are described henceforth. The design philosophy behind FPnew is that the "plumbing" of the architecture is quite regular and generic and the actual operations that handle the data are located at the lowest level. Handshaking is used to pass data through the hierarchy levels. As such, very fine-grained clock-gating can be applied to silence all parts of the architecture that are not actively contributing to useful work, significantly increasing energy efficiency.

4.11.1. Top-Level

The topmost level of hierarchy in FPnew is host to several operation group blocks as well as an output arbiter. The operation group is the highest level of grouping within FPnew and signifies a class of operations that can usually be executed on a single hardware unit - such as additions and multiplications usually being mapped to an FMA unit.

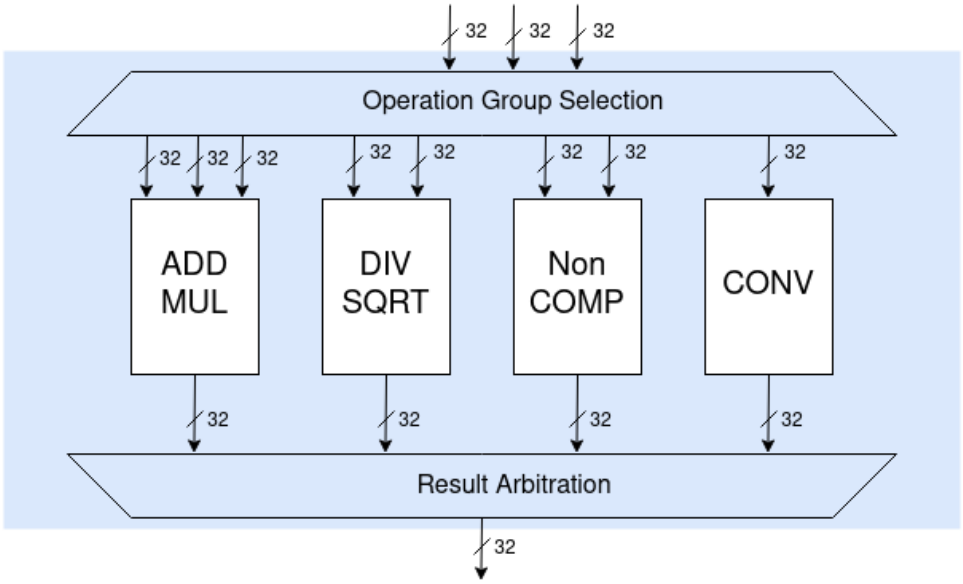

Figure 4.1. FPU block diagram

There are currently four operation groups in FPnew which are enumerated in `opgroup_e` as outlined in the following table:

Table 4.12. Supported hardware operation block

Enumerator	**Description**	**Associated Operations**
ADDMUL	Addition and Multiplication	FMADD, FNMSUB, ADD, MUL
DIVSQRT	Division and Square Root	DIV, SQRT
NONCOMP	Non-Computational Operations like Comparisons	SGNJ, MINMAX, CMP, CLASS
CONV	Conversions	F2I, I2F, F2F

Most architectural decisions for FPnew are made at very fine granularity. The big exception to this is the generation of vectorial hardware which is decided at the top level through the EnableVectors parameter.

4.11.2. Pipelining

Pipeline registers are inserted into the operational units directly, according to the settings in the Implementation parameter. As such, each slice in the system can have a different latency. Merged slices are bound to have the largest latency of the included formats. All pipeline registers are inserted as shift registers at predefined locations in the FPU.

For optimal mapping, retiming functionality of your synthesis tools should be used to balance the paths between registers. Data traverses the pipeline stages within the operational units using the same handshaking mechanism that is also present at the top-level FPU interface.
An individual pipeline stage is only stalled if its successor stage is stalled and cannot proceed in the following cycle. In general, different operations can overtake each other in the FPU if their latencies differ or significant back pressure exists in one of the paths.
Hence, the use of operation tags is required to identify the existing data if more than one operation is allowed to enter the FPU.

4.11.3. Output Arbitration

The round-robin arbiters are used on the output of the slices and operation group blocks that handles the multiple requests from the modules. The fair arbitration scheme is used to allow every unit to get a chance to write on the output with equal access in other words a unit cannot write the outputs twice in a row if other units are also contending for the output.

5. System Bus (TileLink-UL)

Tilelink is an open standard chip interconnect which came into existence along with the evolution of RISCV ISA (instruction set architecture). RISCV ISA is unique in terms of simplicity and efficiency. That is the reason for developing an independent bus architecture for RISC-V based systems. Tilelink provides some additional features over various available bus architectures as shown in table 5.1.

Table 5.1. Features of TileLink Bus Protocol

Feature	**AHB**	**Wishbone**	**AXI4**	**ACE**	**CHI**	**Tilelink**
Open standard	Yes	Yes	Yes	Yes	No	Yes
Easy to implement	No	Yes	-	No	?	Yes
Cache block motion	No	No	No	Yes	Yes	Yes
Multiple cache layers	-	-	-	No	?	Yes
Reusable off-chip	No	-	-	-	?	Yes
High performance	No	Yes	Yes	-	?	Yes

Tilelink is designed for RISCV ISA but also supports other ISAs as well. It provides deadlock freedom and optionally out-of-order transaction completion to improve throughput.

5.1. TitleLink conformal level

Tilelink has three conformance levels based on the design requirements. All TileLink conformance levels follow the same decoupled host and device protocol (officially it is named master-slave protocol) communication shown in figure 5.1.

1. TL-C (TileLink cached)
2. TL-UH (TileLink uncached heavyweight)
3. TL-UL (TileLink uncached lightweight)

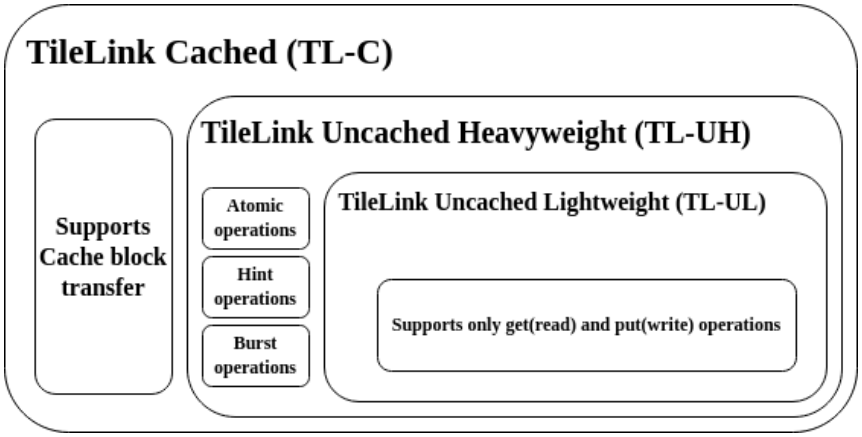

Figure 5.1. TileLink block diagram

For this project TL-UL is the best choice because there is no atomic and burst data transfer operation supported. For that reason, this report will only discuss the details of the TL-UL protocol.

5.2. TileLink Channel

Tilelink protocol comprises five channels (A, B, C, D & E) out of which two channels (A & D) are mandatory for all conformance levels. The other three channels (B, C & E) are only needed for TL-C implementation. To avoid deadlock, TileLink defines the priorities for all five channels as.

A << B << C << D << E

5.3. TileLink Architecture

Tilelink supports multiple hosts and device communication using decoupled interfaces. It is defined in terms of a graph of connected agents that send and receive messages over point-to-point channels within a link to perform operations on a shared address space.

Operation: Operation is defined as the change in data values, permission, or location in the memory hierarchy of an address range.
Agent: The agent is defined as the active module participating in sending or receiving messages over the TL-UL protocol.
Channel: Channel is a unidirectional communication connection between the host and the device.
Message: Messages are a set of control and data values communicated over a channel.
Link: The set of channels required to complete operations between two ends.
Network topology:
In a TileLink network Multiple agents are connected by links. One end of each link is connected to the host interface and the other end is connected to the device interface. The agent with the host interface can send the request over channel A and the agent with the device interface can process the request and send feedback over channel D. As shown in figure 5.2.

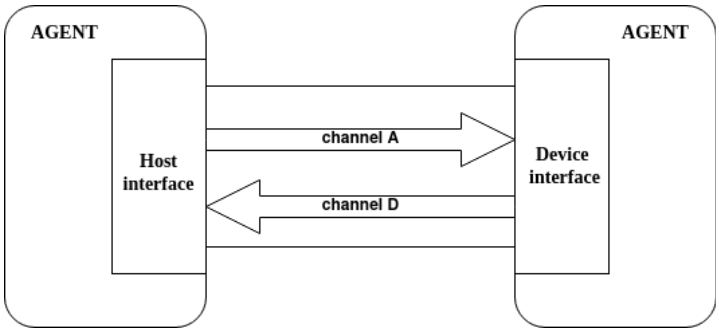

Figure 5.2. TitleLink (UL) network topology

This is the simplest TileLink network, but TileLink also supports a more complex network topology where multiple hosts and devices are connected, and a hierarchical network model is also supported.

5.4. TL-UL component

Current implementation contains the following communication blocks.

5.4.1. TL-Host Adapter

Connected with the host's external interface and manages and converts host signals to TLUL protocol.

5.4.2. TL-SRAM Adapter

Connected with SRAM/memory devices. It manages read/write requests for SRAMs and converts SRAM signals to TLUL protocol. It has responses and requests FIFOs for dealing with multiple read/write access. It optionally converts the host byte mask to a bit mask for SRAM having bit-masked access. It is done by the byte mask parameter.

5.4.3. TL-Reg Adapter

Connected with Register based devices like GPIO, UART, etc. Just like a host adapter it manages and converts device signals to TLUL protocol.

5.4.4. TL-cross bar

Cross bar performs address decoding and signal routing in Tilelink communication. Azadi SoC has two crossbars, one for internal devices like SRAMS, timers, interrupt controllers, etc. And one for devices with external interfaces like, GPIO, UART, SPI, etc.

5.4.5. TL-Socket 1N

1xN socket accepts single input from one host and switches it to one out of N targeted devices based on address. It has default synchronous behavior. 1xN socket is used for the arbitration of single host signals to multiple devices. It also checks TL-UL error conditions and establishes communication accordingly. Figure 5.3 represents the TL-UL socket 1-N.

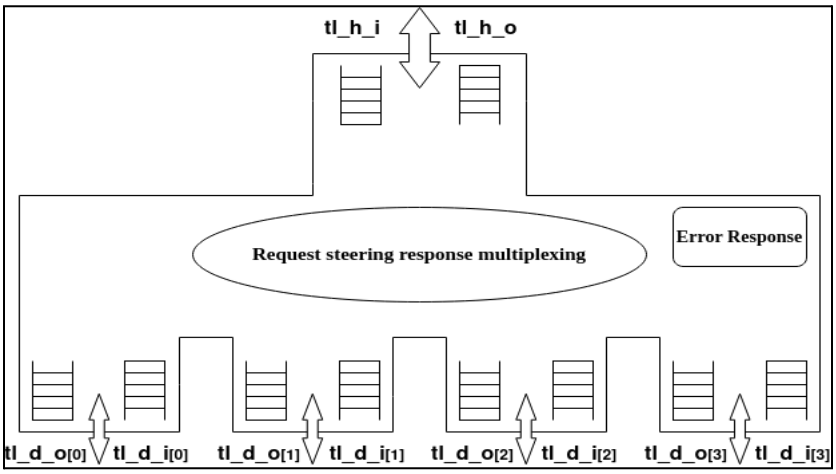

Figure 5.3. TL-UL 1-N socket

5.4.6. TL-Socket M1

Mx1 socket has M number of inputs and a single output channel. It is used on the device side where a single device can be requested by multiple hosts. So, there is a need for checking the priority of the request. Thus Mx1 socket performs priority arbitration when it gets multiple requests at the same instant. Figure 19 represents the TLUL socket M-1.

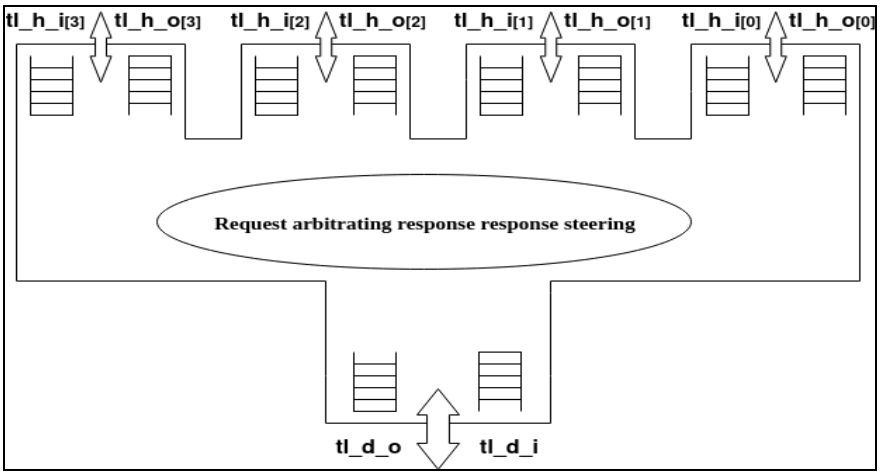

Figure 5.4. TL-UL M-1 socket

5.5. Operations

Tilelink follows a ready valid interface and provides operation code for ongoing transactions. A request is only accepted when it is valid and the device is not busy. As shown in figure 5.5.

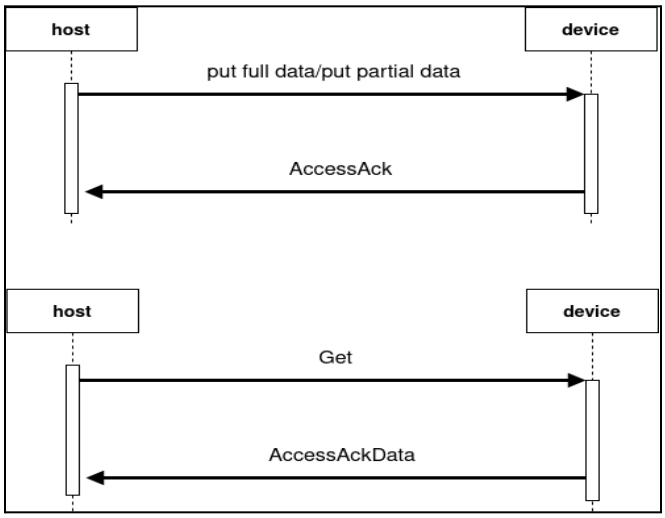

Figure 5.5. TL-UL send/receive operations

5.6. Azadi Bus Hierarchy

Azadi SoC has two levels of Tilelink Crossbar for peripherals with no external interfaces (*tlul_xbar_main*) like Timers, memories, and interrupt controllers and for peripherals with external interfaces (*tlul_xbar_periph*) like PWM, UART, SPI, and GPIOs.

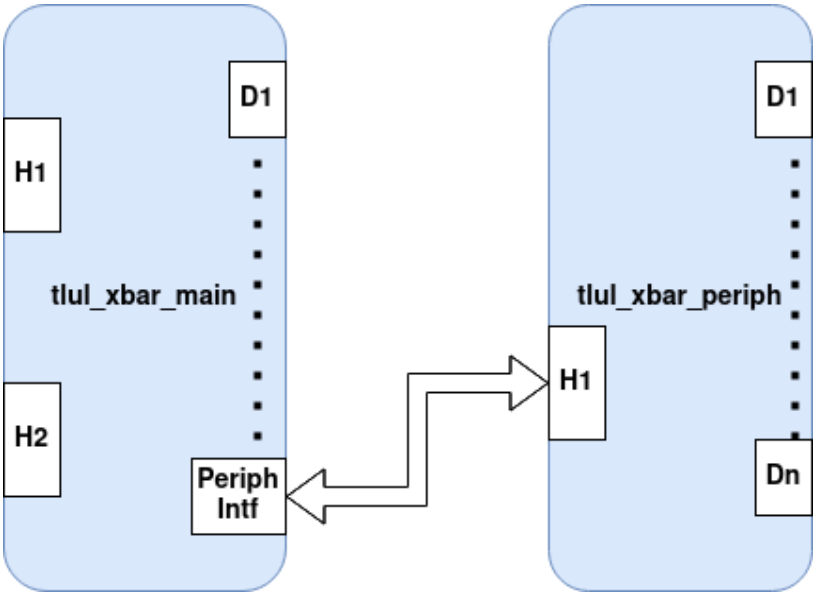

Figure 5.6. Azadi Bus Structure

5.6.1. Memory Map

The Memory map of Azadi-SoC is shown in table 5.1.

Table 5.1. System Memory Map

Host	Peripheral	Base Address	Max Address	Address Space
Host0 (IFU)	QSPI Flash Controller	32'h80000000	32'h80FFFFFF	2 MBytes
	ICCM (32KB)	32'h10000000	32'h10001FFF	1 KBytes
Host1 (LSU)	DCCM (32KB)	32'h20000000	32'h20001FFF	1 KBytes
	Boot Register	32'h20002000	32'h20002000	4 Bytes
	Timer0	32'h30000000	32'h30000FFF	512 Bytes
	Timer1	32'h30001000	32'h30001FFF	512 Bytes
	Timer2	32'h30002000	32'h30002FFF	512 Bytes
	TIC	32'h30003000	32'h300030FF	32 Bytes
	Periph	32'h40000000	32'h4000FFFF	8 KBytes
	PLIC	32'h50000000	32'h50000FFF	512 Bytes
	ROM	32'h60000000	32'h500000FF	256 Bytes
Periph				

(Xbar-peripheral)				
LSU -> periph	GPIO	32'h40001000	32'h400010FF	32 Bytes
	UART0	32'h40002000	32'h400020FF	32 Bytes
	UART1	32'h40002100	32'h400021FF	32 Bytes
	UART2	32'h40002200	32'h400022FF	32 Bytes
	UART3	32'h40002300	32'h400023FF	32 Bytes
	SPI0	32'h40003000	32'h400030FF	32 Bytes
	SPI1	32'h40003100	32'h400031FF	32 Bytes
	SPI2	32'h40003200	32'h400032FF	32 Bytes
	SPI3	32'h40003300	32'h400033FF	32 Bytes
	PWM0	32'h40004000	32'h400040FF	32 Bytes
	PWM1	32'h40004100	32'h400041FF	32 Bytes
	PWM2	32'h40004200	32'h400042FF	32 Bytes
	PWM3	32'h40004300	32'h400043FF	32 Bytes

6. Peripherals

6.1. PLIC

Platform Level Interrupt Controller (PLIC) is designed to handle various external and software interrupts. According to RISC-V PLIC spec, it can handle up to 1024 interrupts for multiple targets. In the current implementation, up to 48 external and software interrupt support is available. Plic can be used with any RISC-V-based system, shown in figure 6.1.

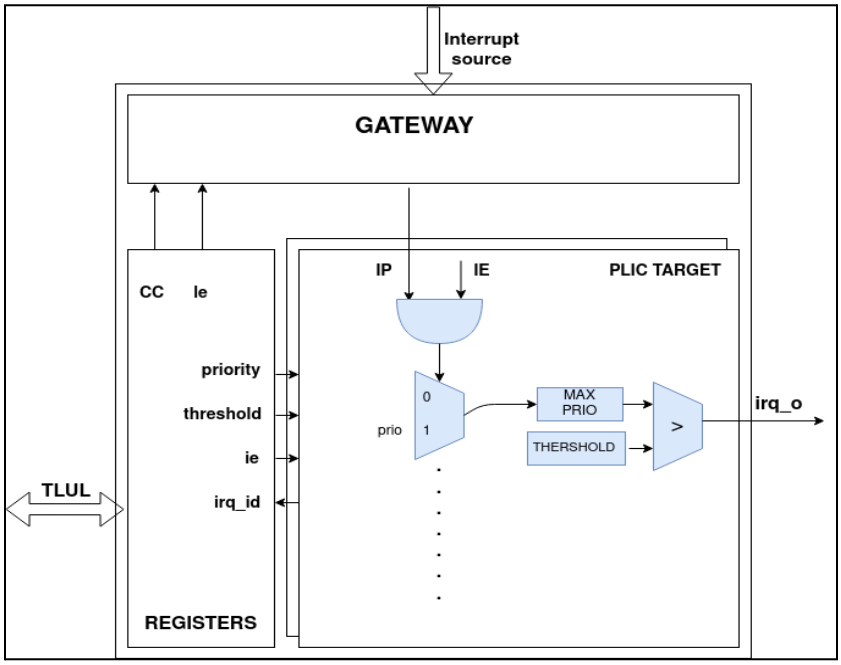

Figure 6.1. PLIC block diagram

6.1.1. rv_plic_gateway

Gateway is the entry point for all external interrupts. Plic gateway accepts interrupts up to 48 in the form of an interrupt vector. Then it converts the external interrupt signal into plic-specific interrupt signals. Once an interrupt is claimed by the target, the gateway will not accept any other interrupt until the claimed interrupt is completed. The diagram below shows the structure of the PLIC gateway for a single interrupt.

6.1.2. rv_plic_target

PLIC target, also known as PLIC core, is responsible for interrupt selection based on priorities and threshold. The module uses a binary tree for finding the interrupt with maximum priority if multiple interrupts are triggered at the same instant.

6.1.3. PLIC priority

Increasing the max priority will also increase hardware to find the max priority. In the current implementation, the max priority is set to 32. Every interrupt has a unique ID based on its index number in the interrupt source vector. An interrupt with the lowest interrupt ID will have the highest priority.

6.1.4. PLIC threshold

The threshold is specific to every target. Since PLIC can support multiple targets(harts) so every target has a specific interrupt acceptance threshold. The interrupts with priority below the target threshold value will not be serviced by that target.

6.1.5. rv_plic_reg_top

PLIC reg top contains all the memory-mapped registers defined in the PLIC specification. Table 6 represents the Plic register map.

Table 6.1. PLIC memory mapped registers

Address	Name	Reset value	Access
0x0 - 0x4	Interrupt pending	0x0	Read only
0x8 - 0xc	level/edge	0x0	read/write
0x10 - 0xd0	Interrupt source priority	0x0	read/write
0xd4 - 0xd8	Interrupt enable	0x0	read/write
0xdc	Threshold for target	0x0	read/write
0xe0	claim/complete	0x0	read/write
0xe4	Machine software interrupt	0x0	read/write

6.1.6. Interrupt enable

For servicing an interrupt the interrupt enable bit for that interrupt must be set.
Interrupt enable bit: index value in interrupt source vector.
Interrupt pending: Any external interrupt will raise the dedicated interrupt pending bit in the interrupt pending register.

6.1.7. Interrupt claim

Target can claim interrupt by reading the interrupt ID from the claim register.

6.1.8. Interrupt completion

After handling the interrupt target should write the interrupt ID back to the complete register for indicating PLIC about the completion of the interrupt.

6.2. Timer

The module has 64-bit counters which increment by a step value whenever prescale reg overflows. There are two counters mtime and *mtimecmp* as described in the RISC-V privilege spec. mtimecmp holds the timeout value while mtime increments itself by a step value after every prescale value overflow, the block diagram shown in the figure 6.2.

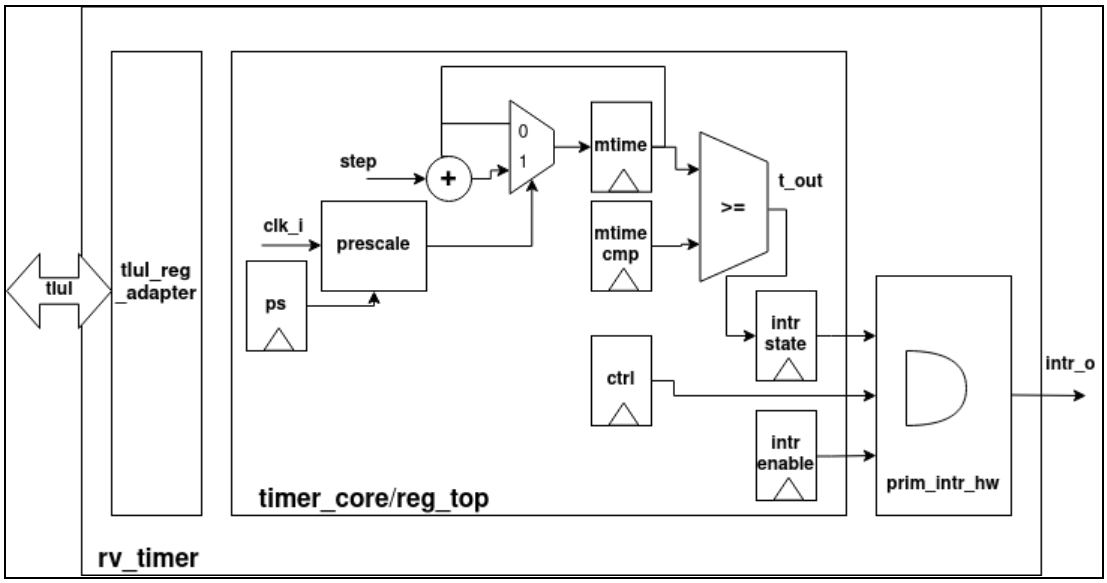

Figure 6.2. Block diagram of Timer

6.2.1. prim_intr_hw

Interrupt hardware is a generic interrupt generation hardware that uses interrupt status and interrupt enables registers for raising interrupts.

6.2.2. timer_core

Timer core module counts the configured timer value based on prescale and steps values for raising time-out events. mtimecmp register is loaded with the timeout value and mtime register counts the value until it becomes equal or greater than *mtimecmp* value. Both registers are 64bit wide.

6.2.3. rv_timer_reg_top

Reg top contains all the registers and their read/write access. Table 6.2 shows the register map of the timer module.

Table 6.2. Timer memory mapped registers

Offset	Name	Reset value	Access
0x0	Control reg	0x0	read/write
0x100	Configuration reg	0x0	read/write
0x104	mtime lower	0x0	read/write
0x108	mtime upper	0x0	read/write
0x10c	mtimecmp lower	0xffffffff	read/write
0x110	mtimecmp upper	0xffffffff	read/write
0x114	Interrupt enable	0x0	read/write

0x118	Interrupt status	0x0	read/write,1clear

The reset value of mtimecmp is kept 0xffffffff_ffffffff for avoiding unnecessary interrupts, therefore programmers should take care of the mtimecmp value while reading or writing it. Both registers are 64bit wide but the system bus only supports 32bit data transfer therefore reading and writing the 64bit value will take 2 cycles. This can lead to reading the incorrect 64-bit value. Therefore it is the responsibility of the programmer to take care of the reading pattern for reading the correct value.

6.3. GPIO

General Purpose Input Output (GPIO) module allows SoC to communicate with various devices through 32 general purpose IOs. all 32 pins can be written as direct out and can also be written as masked write for upper and lower 16 bits. There is also an input filter for noise reduction, the block diagram shown in figure 6.3.

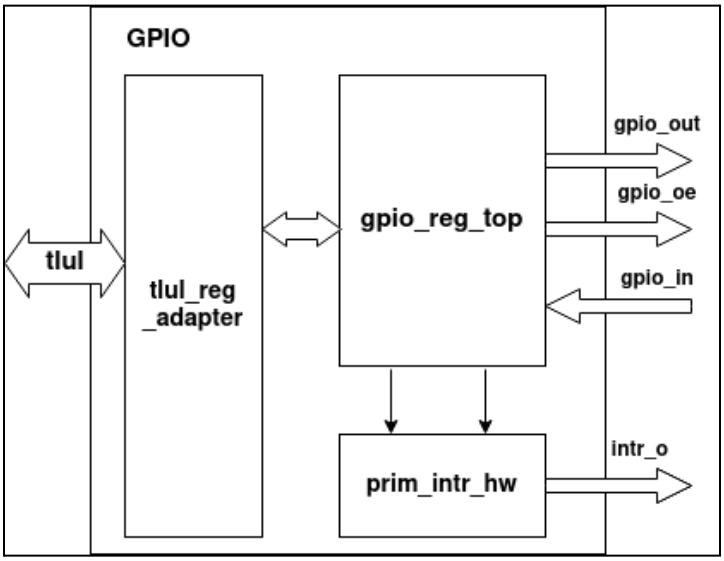

Figure 6.3. GPIO block diagram

6.3.1. gpio_reg_top

Reg top module contains all GPIO registers and their privileges. The register map of the GPIO module is shown in table 6.3.

Table 6.3. GPIO memory mapped registers

Address	Name	Reset value	Access
0x0	Interrupt status	0x0	read/write,1clear
0x4	Interrupt enable	0x0	read/write
0xc	Input data read	0x0	Read only
0x10	Direct out data	0x0	read/write

0x14	Masked write lower	0x0	[15:0]rw, [31:16]wo
0x18	Masked write upper	0x0	[15:0]rw, [31:16]wo
0x1c	Output enable	0x0	read/write
0x20	Output enable for masked write lower	0x0	read/write
0x24	Output enable for masked write upper	0x0	read/write
0x28	Interrupt enable rising edge	0x0	read/write
0x2c	Interrupt enable falling edge	0x0	read/write
0x30	Interrupt enable level high	0x0	read/write
0x34	Interrupt enable level low	0x0	read/write
0x38	Filter enable for input	0x0	read/write

GPIO module provides an optional masked write feature for both lower and pins. For masked write lower, the value to be written on the pins is provided in lower 16 bits and the upper 16 bits are filled with the mask. It is the same for masked write upper case but the value will be written on the upper 16 pins.

6.3.2. Interrupts

GPIO module has four types of interrupt configuration.

- Level high: raise interrupt when a constant high value(1) is detected on the pin.
- Level low: raise interrupt when a constant low value(0) is detected on the pin.
- Rising edge: raise interrupt when the value changes from low to high on the pin.
- Falling edge: raise interrupt when the value changes from high to low on the pin.

Whenever a configured interrupt is detected on the pin it is registered in the interrupt status register which is then checked along with the interrupt enable bit of that pin in the prim_intr_hw module. If the condition satisfies an interrupt will be raised for the configured pin. There are 32 pins and each pin can be configured and raised interrupt independently.

6.4. PWM

Pulse width modulation is widely used in controlling analog circuits using digital logic. It is also used in various control applications like DC motor speed and direction control and brightness control of LEDs. Azadi SoC provides a dual-channel PWM module with independent control over each channel, figure 6.4 represents the system diagram of the PWM module.

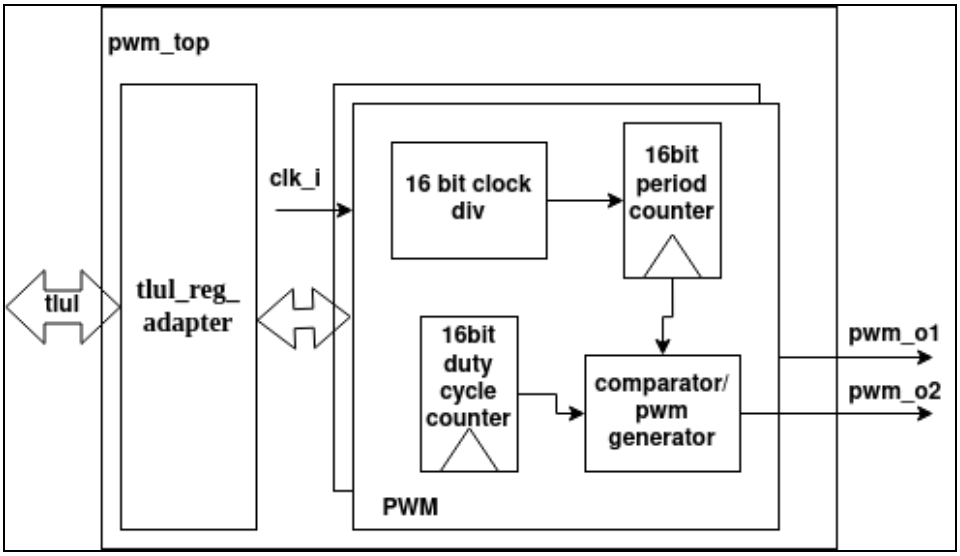

Figure 6.4. PWM block diagram

6.4.1. down_clocking

The ‘down_clocking’ is used for clock division for variable frequency of PWM output. There are two separate clock dividers for each PWM channel. Table 6.4 shows the register map of the PWM module.

$$f_{pwm} = f_{clki} / divisor$$

Table 6.4. PWM memory mapped registers

Offset	Name	Reset value	Access
0x0	ctrl	0x0	read/write
0x4	Divisor_1	0x0	read/write
0x8	Period_1	0x0	read/write
0xc	Duty cycle (DC_1)	0x0	read/write
0x10	Divisor_2	0x0	read/write
0x14	Period_2	0x0	read/write
0x18	Duty cycle (DC_2)	0x0	read/write

6.5. UART

The peripheral UART enables the communication of SoC with the outside world using two pins RX to receive the data and TX to transmit the data. The UART protocol is being used in many embedded devices due to its simplicity and less hardware. The protocol works on a transmitter and receiver rather than a master and slave. The device that is sending the data would be the transmitter and receiving device would be the receiver. The transmission is done by the

transmission of frames from the transmitter to the receiver. The frame size can be 10 or 11 bits, without parity bit and with parity bit respectively. The communication starts with the start bit and ends with a stop bit. There is no acknowledgment signal received from the receiver which ensures the successful transmission of the data packet. To communicate properly both the transmitter and receiver should be set on the same baud rate. The current implementation of UART supports the following features.

- full duplex data transmission.
- configurable baud rate.
- 128 bytes TX buffer.
- 128 bytes RX buffer.
- timeout-based RX interrupts for receiving multiple consecutive bytes and optional RX interrupts per byte.

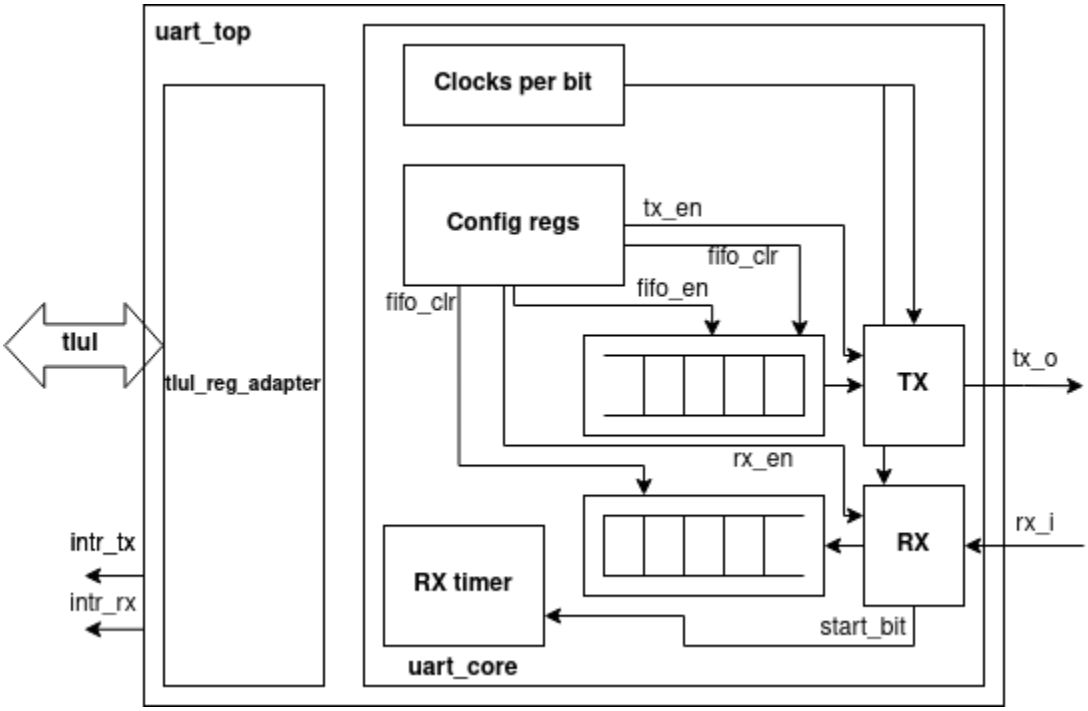

Figure 6.5. UART Block Diagram

6.5.1. Baud Rate

Baud rate simply means the number of changes of signal in one second while the bit rate is the number of transmissions of bits in a second. The baud rate has some standard values like 1200, 2400, 9600, 19200, 38400, 57600, 115200, and 230400.
The transmitter and receiver modules require a number of clocks per bit for the operation which can be calculated as

$$Clocks\ per\ bits\ = \frac{f_{sys\ clock}}{baud\ rate}\ +\ 1$$

6.5.2. uart_core

uart_core is a top-level wrapper that contains a register map and uart_tx and uart_rx modules. It also has relevant logic for TX and RX interrupt generation.

Table 6.5. UART memory mapped registers

Offset	Name	Reset value	Access
0x0	clocks_per_bit	0x0	Write only
0x4	tx	0x0	Write only
0x8	rx_val	0x0	Read only
0xc	rx_en	0x0	Write only
0x10	tx_en	0x0	Write only
0x14	rx_status	0x0	Read only
0x18	rx_sc (rx status clear)	0x0	Write 1 clear
0x1c	tx_fifo_en	0x0	Write only
0x20	tx_fifo_clr	0x0	Write only
0x28	rx_timeout	0x0	Write only
0x30	rx_fifo_clr	0x0	Write only
0x34	rx_buffer_size	0x0	Read only
0x38	rx_byte_en	0x0	Write only

6.6. Programming UART

Programmers should consider the following points when programming UART.

- Calculate clocks per bit with desired baud rate using the given formula and write it into the *clocks_per_bit* register.
- In the case of TX operation write all required data into TX buffer (on offset 0x4). Then first enable tx FIFO by writing 1 to *tx_fifo_en* register and at last enable transmitter by writing 1 to *tx_en* register. the device will raise TX interrupt once all the data in the buffer is transmitted.
- In the case of RX operation write the appropriate clocks per bit value in the *clocks_per_bit* register. The status of received data can be checked by polling the *rx_status* register or it could be an interrupt-driven receive operation. For RX interrupts, there are two options: enable interrupts for every received byte by writing 1 to the *rx_byte_en* register or set an RX timeout value in the *rx_timeout* register for raising interrupt on multiple consecutively received bytes. At last, enable the receiver module by writing 1 to the *rx_en* register.

6.7. TIC

Timer Interrupt Controller (TIC) is designed for dealing with interrupts of three timers in Azadi SoC. In the current implementation, only three interrupts are supported. Each interrupt is assigned a unique id. Id values are stored in the *irq_id* register based on interrupt priority.
Interrupt priority order: intr_src[3] >> intr_src[2] >> intr_src[1] >> intr_src[0]

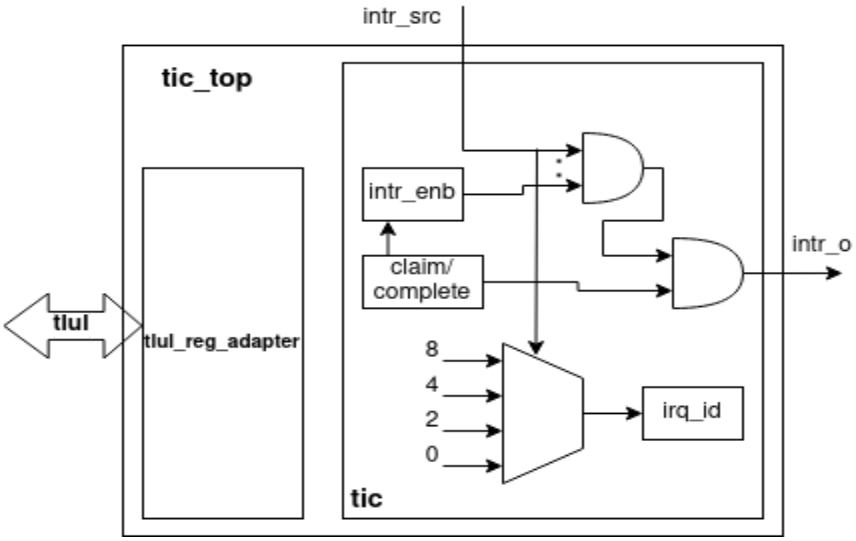

Figure 6.6. TIC Block Diagram

Table 6.6. TIC memory mapped registers

Offset	Name	Reset value	Access
0x4	intr_enb	0x0	Write only
0x8	claim / complete	0x0	Read/Write

6.7.1. TIC interrupt handling

Programmers should consider the following points when dealing with timer interrupts with tic.

- TIC generates an interrupt for core when interrupts are enabled in TIC. For enabling interrupts write 1 to the *intr_enb* register.
- TIC deals with one interrupt at a time. In case of multiple interrupts, the interrupt with the highest priority will be serviced.
- Programmers should take care in writing ISR for TIC.
    - The first thing to do in tic ISR is to claim the interrupt by reading the interrupt id from address 0x8. Once the interrupt is claimed no further interrupts will be accepted by TIC.
    - Once the required action is performed against the interrupt, ISR should mark the interrupt as complete by clearing the irq_id at address 0x8. The flow is described in waveforms.

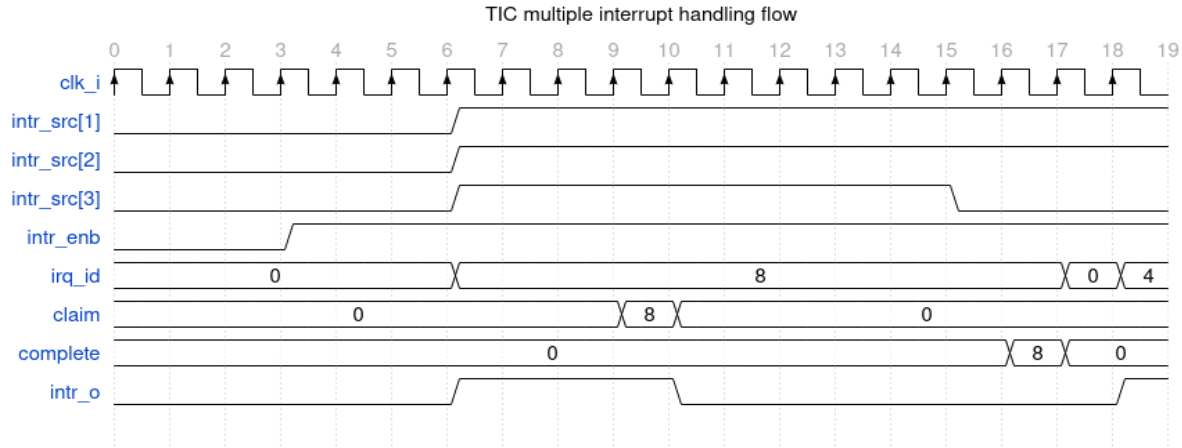

Figure 6.7. Timing diagram of interrupt handling with TIC

6.8. SPI

Serial Peripheral Interface is a synchronous communication protocol used for short-distance communication and is widely used in different devices and sensors today. Spi uses host and device network topology. a single host can communicate with multiple devices based on the device's select signal. The current implementation of SPI protocol provides the following features. Figure 6.8 shows the system diagram of the SPI module.

- Full duplex data transfer.
- Optional data transfer with LSB or MSB first.
- Data transfer of up to 64 bits.
- Supports all four combinations of CPOL and CPHA.
- Supports clock division up to 16 bits.

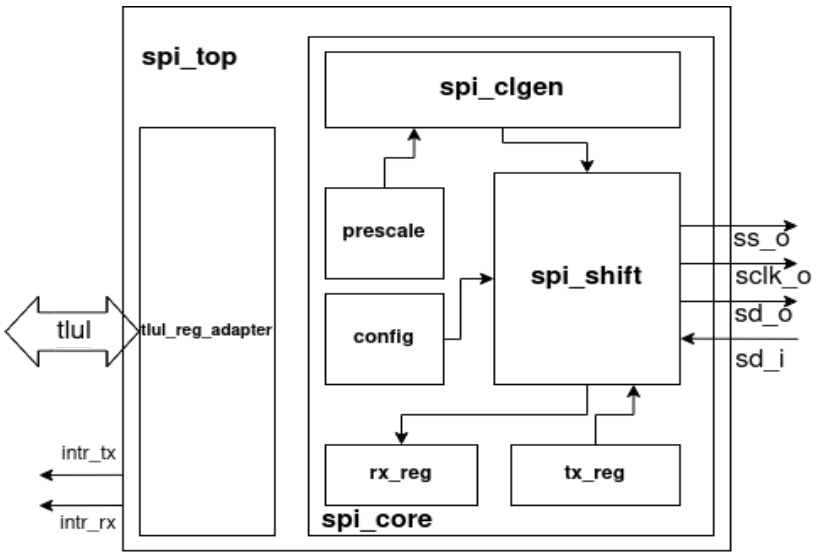

Figure 6.8. Block diagram of SPI

6.8.1. spi_clgen

spi_clgen is a clock divider that generates clock ticks for running SPI at a lower frequency. In default condition the out frequency of *sclk_o* is sys_clk/2. The divider has 16 bits resolution and divides the clock in the power of 2 i.e 2^n.

$$f_{sclk} = f_{clki} / (2^n - 1)$$

6.8.2. spi_core

SPI core contains all the registers of the SPI module and an interface to the spi_top module. Table 7 represents the register map of the SPI module.

Table 6.7 SPI memory mapped registers

Offset	**Name**	**Reset value**	**Access**
0x0	configuration	0x0c000	Write only
0x4	prescale	0x0	Write only
0x8	tx_reg[31:0]	0x0	Write only
0xc	tx_reg[63:32]	0x0	Write only
0x10	rx_reg[31:0]	0x0	Read only
0x14	rx_reg[63:32]	0x0	Read only
0x18	control	0x0	Write only

Table 6.8. Configuration Register

Configuration Register						
rlsb	**tlsb**	**tx_en**	**cpha**	**cpol**	**num_rx_bits**	**num_tx_bits**
20	19	18	17	16	15 - 8	7 - 0
num_tx_bits	Specifies the number of bits to transmit. Max 64 bits are supported in the current implementation.					
num_rx_bits	Specifies the number of bits to receive. Max 64 bits are supported in the current implementation.					
cpol	Clock polarity can be 1 or 0. default is 1.					
cpha	Clock phase can be 1 or 0. default is 1.					
tx_en	Set 1 for write-only operation.					
tlsb	Set 1 for transmitting data from LSB.					

rlsb	Set 1 for receiving data from LSB.

Table 6.9. Prescale Register

Prescale	
prescale value	**ps_en**
16 - 1	0
ps_en	Set 1 to enable clock division.
prescale value	16-bit clock division value.

6.8.3. spi_shift

SPI shift has a state machine with four states (IDLE, TRANSMIT, TOGGLE, and RECEIVE). The data is shifted out on the basis of the configuration register in TRANSMIT state. TOGGLE state inverts the value of *sclk_o* for generating a stable output SPI clock. It also shifts the incoming data on the rising edge of the system clock. RECEIVE state marks the end of rx operation. The current implementation of the spi_shift module supports all four combinations of CPOL and CPHA timing specification of CPOL and CPHA is described in the diagram below.

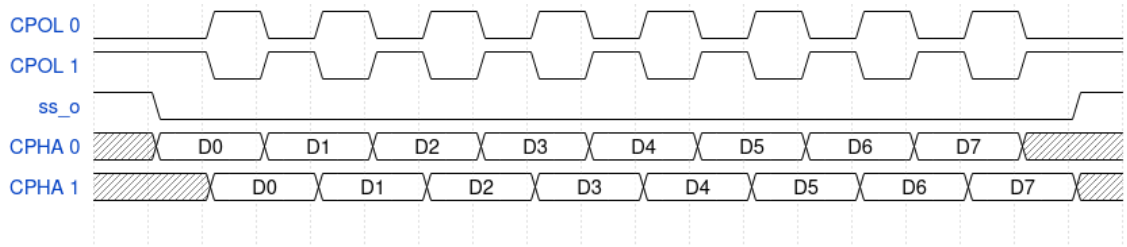

Figure 6.9. SPI modes based on CPOL and CPHA

6.8.4. Interrupts

SPI module has 2 interrupts for tx and rx after every data transfer or reception an interrupt is raised to indicate the completion of the operation.

6.9. Programming SPI

Programmers should consider following instructions while programming the SPI module.

- Current implementation supports up to 64 bits of data transmission and reception. but the write and read interface of the module is 32 bits so data (read or write) greater than 32 bits should be carried out in two cycles as per the given register map.
- Data transmission or reception is supported for both LSB and MSB configs. but when data transmission is less than 64 bits then two cases should be considered.
    - If shift out mode(tlsb) is selected for MSB then data should be written to the MSBs of *tx_reg*.
    - If a shift in mode(rlsb) is selected for MSB then data should be read out from the lower bits of the *rx_reg* and if the mode is selected for LSB then data should be read from the upper bits of the *rx_reg*.

- *Control* reg should be written at last after writing *tx_reg*, *prescale,* and *configuration* registers.

6.10. QSPI

Azadi SoC has a dedicated quad SPI device for executing programs from flash. In the current implementation, the controller only deals with the execute-in-place (XIP) mode of flash. So it is mandatory to set the flash into XIP through an external SPI programming interface. There is an optional configuration feature provided for running the device in SPI mode, exiting the XIP mode, setting dummy cycles, and loading the program from a configurable address. This feature is accessible through an external SPI programming interface.

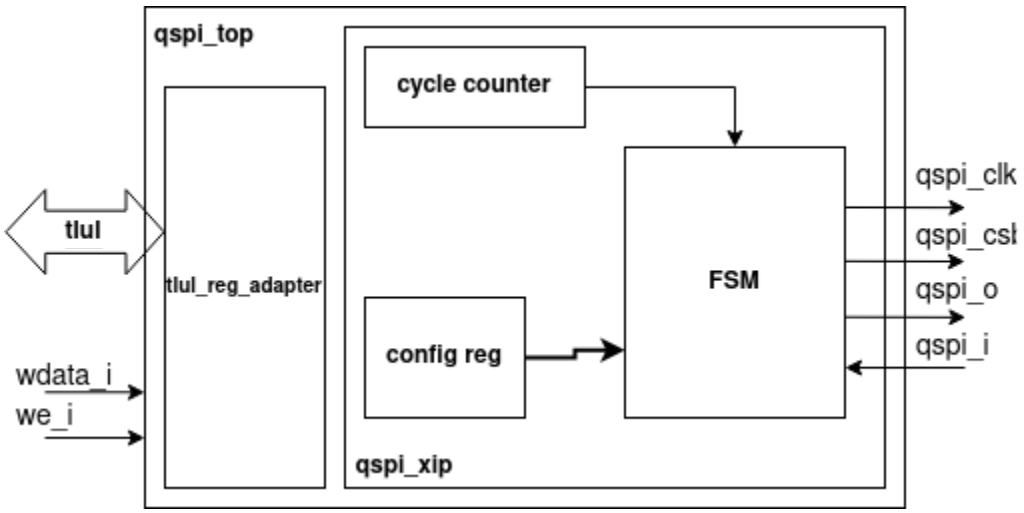

Figure 6.10. QSPI block diagram

Table 6.10. QSPI configuration register

Dummy cycles	quad mode	exit xip	addr
31 - 26	25	24	23 - 0

addr	Optional for loading program from configurable address
exit xip	Set 1 to exit flash from XIP mode. Details could be found in the flash datasheet.
quad mode	Set 1 to run in quad mode (default).
dummy cycles	Optional for setting variable dummy cycles.

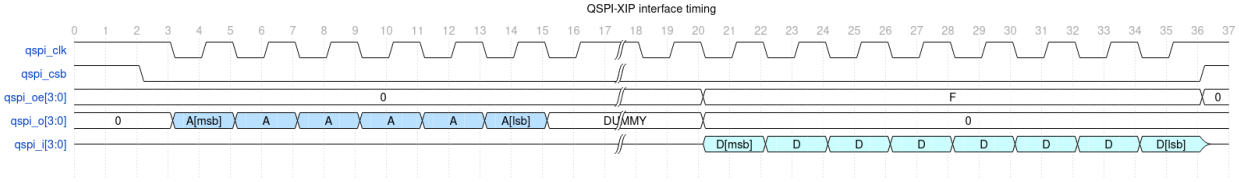

Figure 6.11. QSPI interface timing

6.10.1. Programming Interface and Targets

Azadi SoC has an SPI slave interface (for loading programs into either tightly coupled memory ICCM or xSPI flash) which receives data in bytes and writes to the selected target. There are three targets that can be programmed externally through the SPI slave interface.

1. ICCM
2. xSPI Flash
3. Quad SPI Flash Controller

6.10.2. Selecting Targets

SPI programming interface has command-based target selection. Therefore writing to any programmable target is followed by a command for writing to that target.

1. B1h: command for writing program to ICCM.
2. B2h: command for writing program to SPI Flash.
3. B3h: command for writing configuration string to Quad SPI Flash controller.

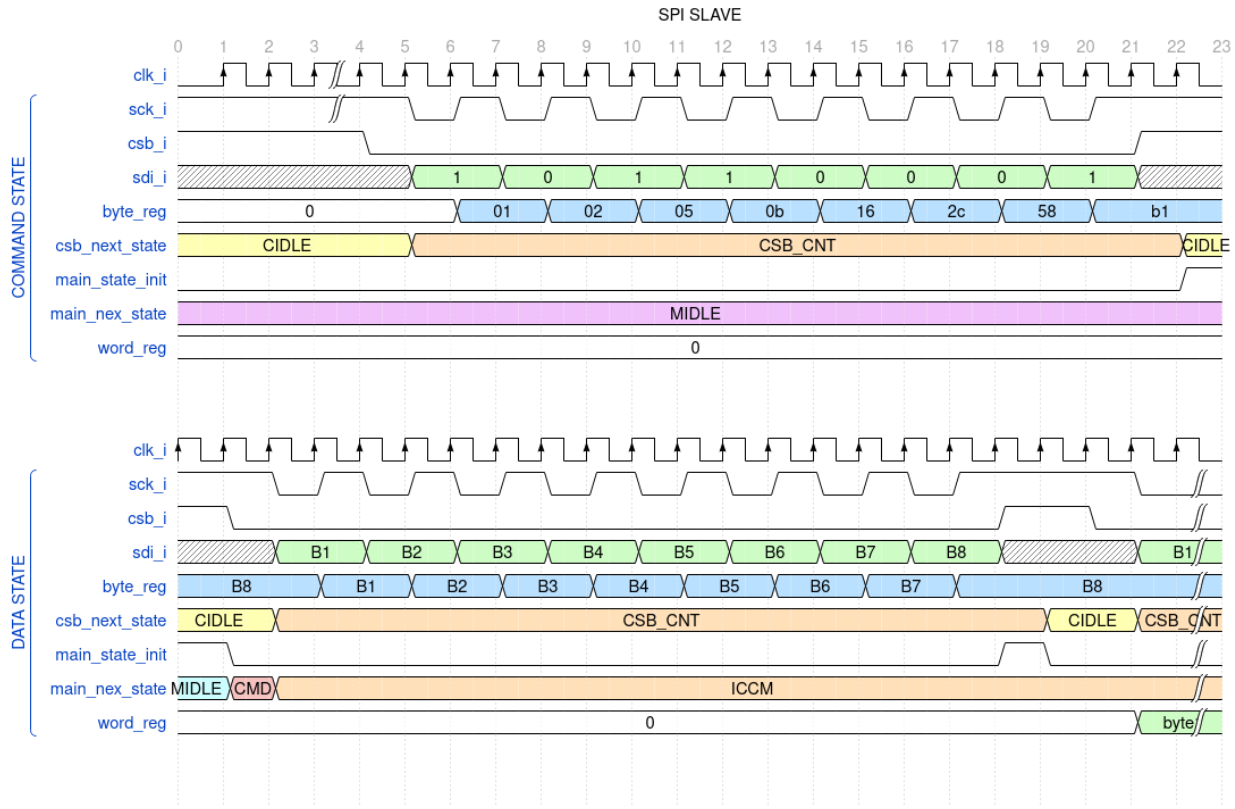

Figure 6.12. Timing diagram of SPIprogrammer

In the above figure an example is shown for programming ICCM. First, a command B1h is sent which can be seen in ***byte_reg*** once the SPI transaction is completed the signal ***main_state_init*** is set to invoke the main state machine which then checks the command and sets the appropriate state for subsequent SPI transactions.

6.10.3. State Transitions

Figure 6.13 shows the CSB state diagram.

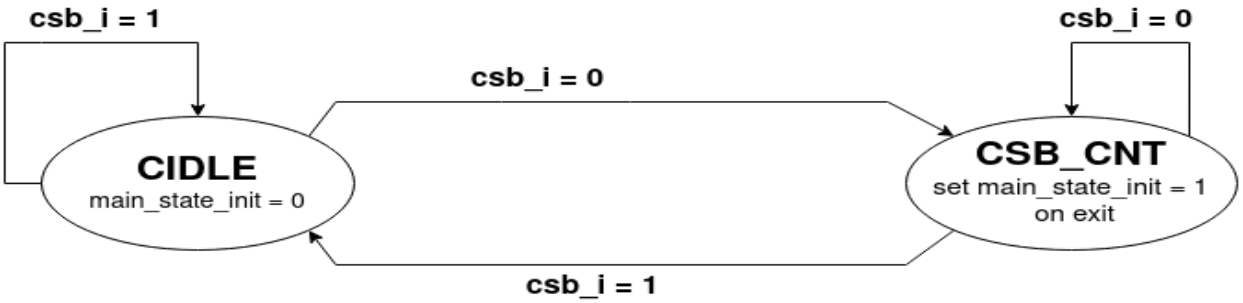

Figure 6.13. SPI programmer csb state machine

The diagram above shows the states of the ***csb_i*** signal which marks the beginning and end of SPI transactions. A detailed state flow can be seen in the flow diagram in figure 6.14.

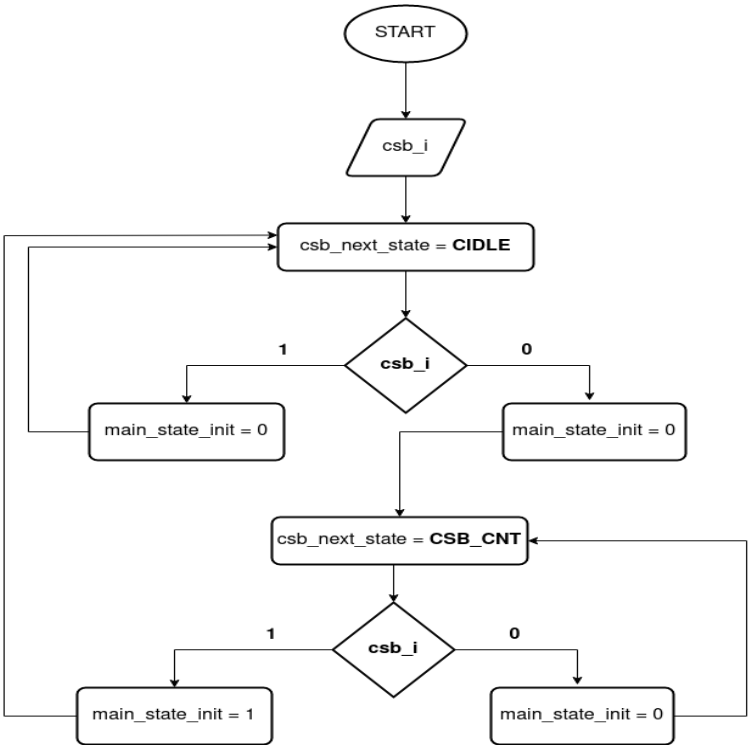

Figure 6.14. CSB control flow diagram for SPI programmer

The main state machine diagram is shown in figure 6.15.

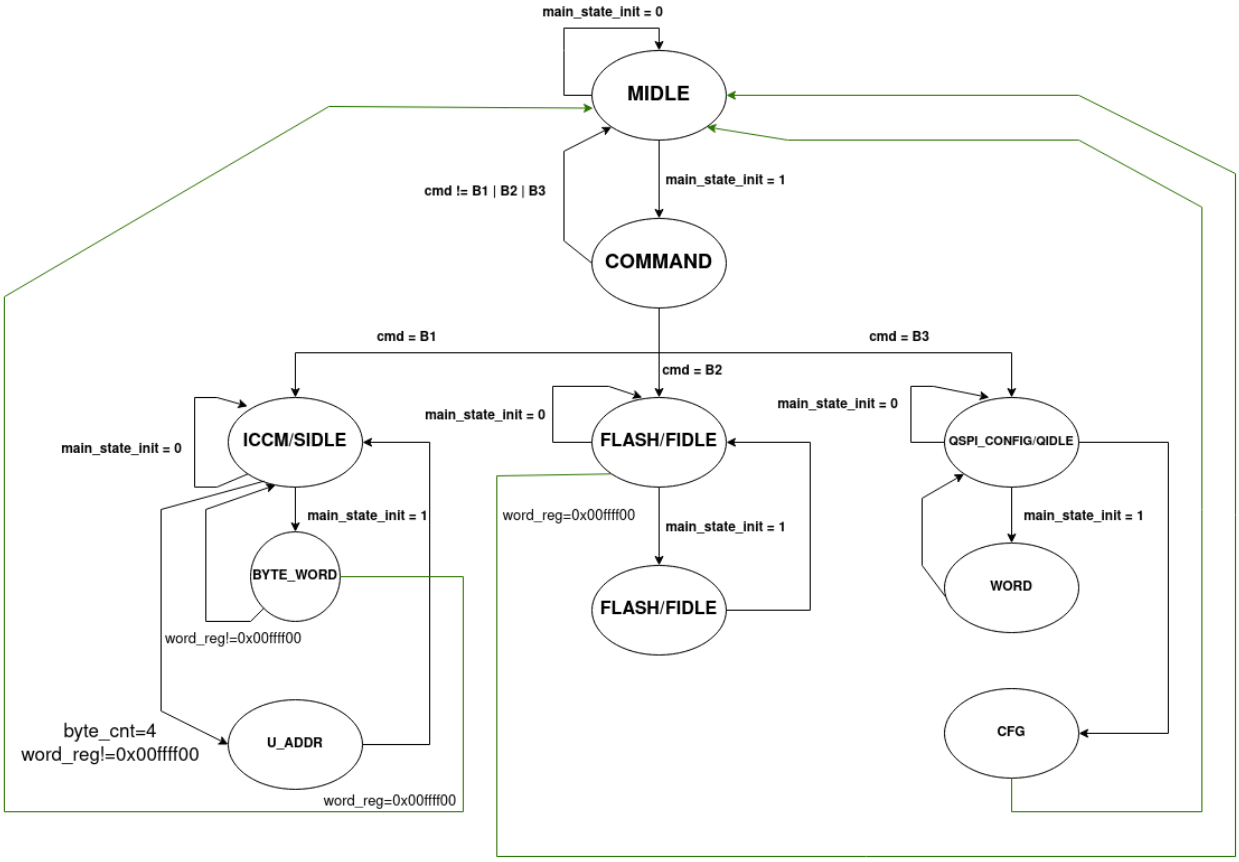

Figure 6.15. State diagram of SPI programmer

The main state machine is set to command state on ***main_state_init*** and when a command is detected for a particular target it sets the state to the appropriate sub-state (ICCM, FLASH, or QSPI_CONFIG). The sub-states ICCM and FLASH require a 32-bit exit command(0x00ffff00) once the program is loaded completely. The details of state transitions can be seen in the flow diagram in figure 6.15 and the flow diagram is shown in figure 6.16.

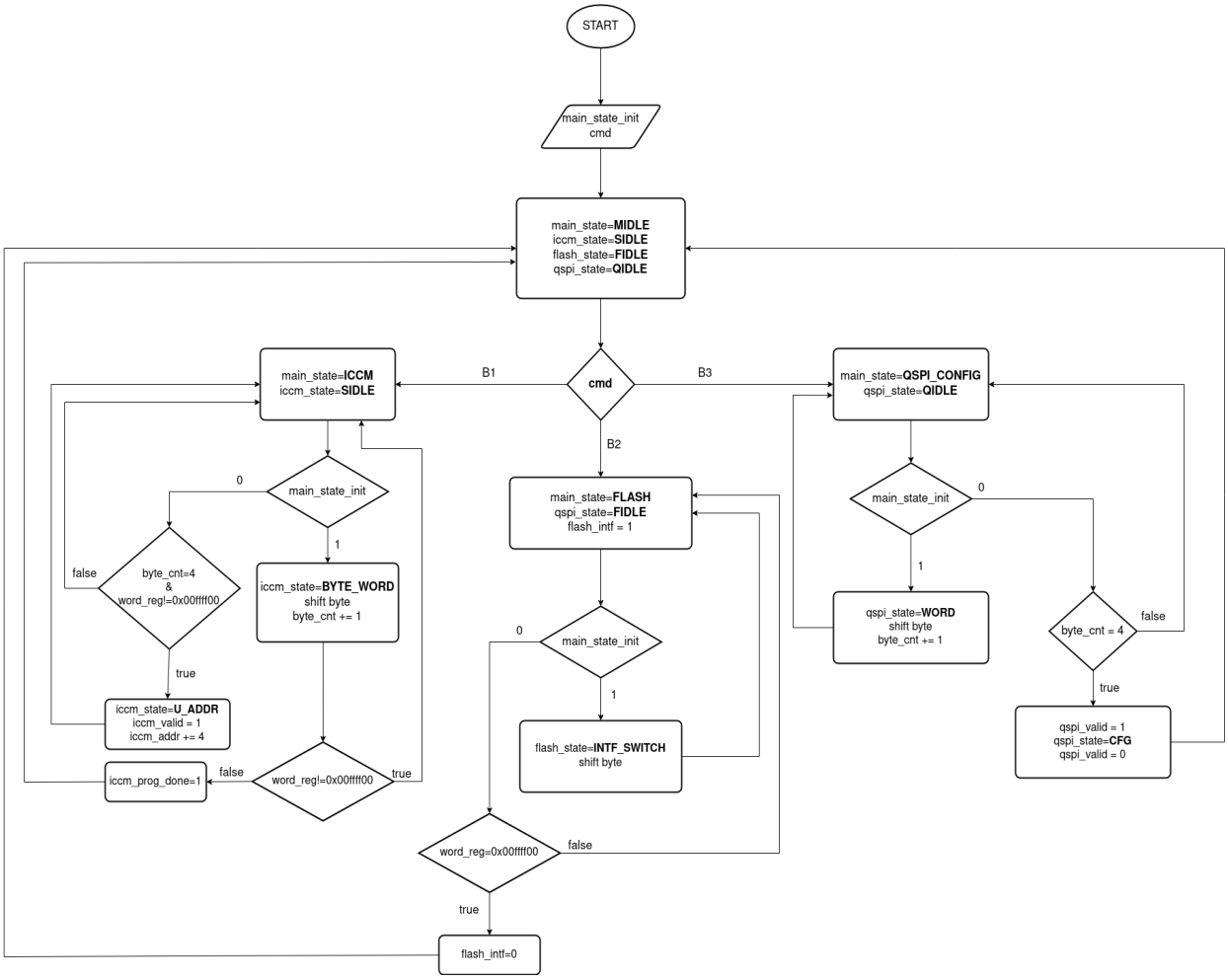

Fig 6.16. Flow diagram of SPI programmer

References

1. Ibex, lowRISC, (2023), GitHub repository, https://github.com/lowrisc/ibex
2. Mach, S., Schuiki, F., Zaruba, F., & Benini, L. (2020). FPnew: An open-source multiformat floating-point unit architecture for energy-proportional transprecision computing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, *29*(4), 774-787.

Others:

1. RISC-V unprivileged ISA: riscv-spec-20191213
2. RISC-V privileged ISA: riscv-privileged-20211203
3. TileLink specification: tilelink-spec-1.7.1
4. IEEE-754 Floating Point standard: IEEE-754-2008