# SiFive E300 Platform Reference Manual

Version 1.0.1

# SiFive E300 Platform Reference Manual

## Proprietary Notice



## Release Information

| Version | Date | Changes |
|---|---|---|
| 1.0.1 | Dec 19, 2016 | Minor clarifications on PWM, SPI, AON register fields |
| 1.0 | November 29, 2016 | Initial release for HiFive1 release. |

# Contents


**SiFive E300 Platform Reference Manual** i

**1 Introduction** 1

1.1 Block Diagram . . . 1
1.2 Configurable E31 RISC-V Coreplex . . . 1
1.3 Custom Accelerators . . . 2
1.4 On-Chip Memory . . . 2
1.5 Execute-in-Place Quad-SPI Flash controller . . . 2
1.6 Peripheral Devices . . . 3
1.7 Platform-Level Interrupt Controller . . . 3
1.8 Always-On Block and Power Management . . . 3
1.9 Debug Support . . . 3
1.10 Software Tools . . . 3

**2 E300 Platform Memory Map** 5

**3 E300 Power Modes** 7

3.1 Run Mode . . . 7
3.2 Wait Mode . . . 7
3.3 Sleep Mode . . . 7

**4 E300 Clock Generation** 9

4.1 Clock Generation Overview . . . 9
4.2 Internal Trimmable Programmable 72 MHz Oscillator (HFROSC) . . . 9
4.3 External 16 MHz Crystal Oscillator (HFXOSC) . . . 11
4.4 Internal High-Frequency PLL (HFPLL) . . . 11
4.5 PLL Output Divider . . . 13
4.6 Internal Low-Frequency Oscillator (LFRCOSC) . . . 13
4.7 External 32.768 kHz Low-Frequency Crystal Oscillator (LFXOSC) . . . 13

**5 E300 Always-On (AON) Domain 15**

5.1 AON Power Source . . . 15
5.2 AON Clocking and Tilelink Slave Port . . . 15
5.3 AON Reset Unit . . . 15
5.3.1 Power-On Reset Circuit . . . 16
5.3.2 External Reset Circuit . . . 16
5.3.3 Reset Cause . . . 17
5.4 Watchdog Timer (WDT) . . . 17
5.5 Real-Time Clock (RTC) . . . 17
5.6 Backup Registers . . . 17
5.7 Power-Management Unit (PMU) . . . 17
5.8 AON Memory Map . . . 17

**6 E300 Power-Management Unit (PMU) 19**

6.1 PMU Overview . . . 19
6.2 PMU Key Register (`pmukey`) . . . 19
6.3 PMU Program . . . 19
6.4 Initiate Sleep Sequence Register (`pmusleep`) . . . 20
6.5 Wakeup Signal Conditioning . . . 21
6.6 PMU Interrupt Enables (`pmuie`) and Wakeup Cause (`pmucause`) . . . 21
6.7 Memory Map . . . 22

**7 E300 Power, Reset, Clock, Interrupt (PRCI) Control and Status Registers 23**

7.1 PRCI Address Space Usage . . . 23

**8 E300 Watchdog Timer (WDT) 25**

8.1 Watchdog Count Register (`wdogcount`) . . . 25
8.2 Watchdog Clock Selection . . . 26
8.3 Watchdog Configuration Register `wdogcfg` . . . 26
8.4 Watchdog Compare Register (`wdogcmp`) . . . 27
8.5 Watchdog Key Register (`wdogkey`) . . . 27
8.6 Watchdog Feed Address (`wdogfeed`) . . . 27
8.7 Watchdog Configuration . . . 27
8.8 Watchdog Resets . . . 27
8.9 Watchdog Interrupts (`wdogcmpip`) . . . 28

**9 E300 Real-Time Clock (RTC) 29**

9.1 RTC Count Registers `rtchi`/`rtclo` . . . 29

9.2 RTC Configuration Register rtccfg . . . . . . . . . . 29
9.3 RTC Compare Register rtccmp . . . . . . . . . . 30

**10 E300 Backup Registers 31**

**11 General Purpose Input/Output Controller (GPIO) 33**
11.1 Memory Map . . . . . . . . . . 33
11.2 Input / Output Values . . . . . . . . . . 33
11.3 Interrupts . . . . . . . . . . 33
11.4 Internal Pull-Ups . . . . . . . . . . 35
11.5 Drive Strength . . . . . . . . . . 35
11.6 Output Inversion . . . . . . . . . . 35
11.7 HW I/O Functions (IOF) . . . . . . . . . . 35
11.8 Behavior During Sleep Mode . . . . . . . . . . 36

**12 Universal Asynchronous Receiver/Transmitter (UART) 37**
12.1 UART Overview . . . . . . . . . . 37
12.2 Memory Map . . . . . . . . . . 37
12.3 Transmit Data Register (txdata) . . . . . . . . . . 38
12.4 Receive Data Register (rxdata) . . . . . . . . . . 38
12.5 Transmit Control Register (txctrl) . . . . . . . . . . 38
12.6 Receive Control Register (rxctrl) . . . . . . . . . . 39
12.7 Interrupt Registers (ip and ie) . . . . . . . . . . 39
12.8 Baud Rate Divisor Register (div) . . . . . . . . . . 39

**13 Serial Peripheral Interface (SPI) 41**
13.1 SPI Overview . . . . . . . . . . 41
13.2 Memory Map . . . . . . . . . . 41
13.3 Serial Clock Divisor Register (sckdiv) . . . . . . . . . . 41
13.4 Serial Clock Mode Register (sckmode) . . . . . . . . . . 42
13.5 Chip Select ID Register (csid) . . . . . . . . . . 43
13.6 Chip Select Default Register (csdef) . . . . . . . . . . 43
13.7 Chip Select Mode Register (csmode) . . . . . . . . . . 43
13.8 Delay Control Registers (delay0 and delay1) . . . . . . . . . . 44
13.9 Frame Format Register (fmt) . . . . . . . . . . 44
13.10 Transmit Data Register (txdata) . . . . . . . . . . 45
13.11 Receive Data Register (rxdata) . . . . . . . . . . 45
13.12 Transmit Watermark Register (txmark) . . . . . . . . . . 46

13.13 Receive Watermark Register (rxmark) . . . . . 46
13.14 Interrupt Registers (ie and ip) . . . . . 46
13.15 SPI Flash Interface Control Register (fctrl) . . . . . 47
13.16 SPI Flash Instruction Format Register (ffmt) . . . . . 47

**14 One-Time Programmable Memory (OTP) Peripheral 49**
14.1 Memory Map . . . . . 49
14.2 Programmed-I/O lock register (otp_lock) . . . . . 49
14.3 Programmed-I/O Sequencing . . . . . 51
14.4 Read sequencer control register (otp_rsctrl) . . . . . 51

**15 E300 Pulse-Width Modulation (PWM) Peripheral 53**
15.1 PWM Overview . . . . . 53
15.2 PWM Memory Map . . . . . 53
15.3 PWM Count Register (pwmcount) . . . . . 53
15.4 PWM Configuration Register (pwmcfg) . . . . . 54
15.5 PWM Compare Registers (pwmcmp0–pwmcmp3) . . . . . 55
15.6 Deglitch and Sticky circuitry . . . . . 56
15.7 Generating Left- or Right-Aligned PWM Waveforms . . . . . 56
15.8 Generating Center-Aligned (Phase-Correct) PWM Waveforms . . . . . 57
15.9 Generating Arbitrary PWM Waveforms using Ganging . . . . . 58
15.10 Generating One-shot Waveforms . . . . . 58
15.11 PWM Interrupts . . . . . 58

# Chapter 1

# Introduction

The E300 platform is the first member of SiFive's Freedom Everywhere family of customizable RISC-V SoCs. By combining a highly configurable base platform with customer-specific hardware extensions, the Freedom Everywhere family provides low-NRE and rapid time-to-market solutions for performance, cost, and power-sensitive embedded and IoT markets.

Each E300 SoC includes a SiFive E3 series RISC-V Coreplex with integrated instruction and data memories, a platform-level interrupt controller, on-chip debug unit, and an extensive selection of peripheral devices. This manual should be read together with the E3 Coreplex manual.

All aspects of the base E300 platform can be flexibly configured. In addition, the platform can be readily extended with customer-specific instruction-set extensions, custom coprocessors, custom accelerators, custom I/O, and custom always-on blocks. The resulting application-specific E300 SoC is optimized for manufacture in a TSMC 180nm process, and delivered as packaged tested parts by SiFive.

### Block Diagram

Figure 1.1 shows the top-level block diagram of the E300 platform. The heart of the current E300 platform is an E31 Coreplex, which contains an E31 RISC-V processor, instruction and data memories, the platform-level interrupt controller (PLIC), a central DMA controller, and a debug module.

### Configurable E31 RISC-V Coreplex

The configurable E31 RISC-V Coreplex provides a high-performance single-issue in-order 32-bit execution pipeline, with a peak sustained execution rate of one instruction per clock cycle. The Freedom E300 platform supports most configuration options of the E31 core as described in the E3 Coreplex manual, except for the following:

- Where present, the instruction cache line size is 32 bytes.
- The data cache is not supported.

The E3 Coreplex exports two TileLink attachments; a TileLink master port which can be used to attach a custom accelerator, and a TileLink slave port to drive the platform bus. Both ports support 32-byte burst accesses over a 32-bit datapath.

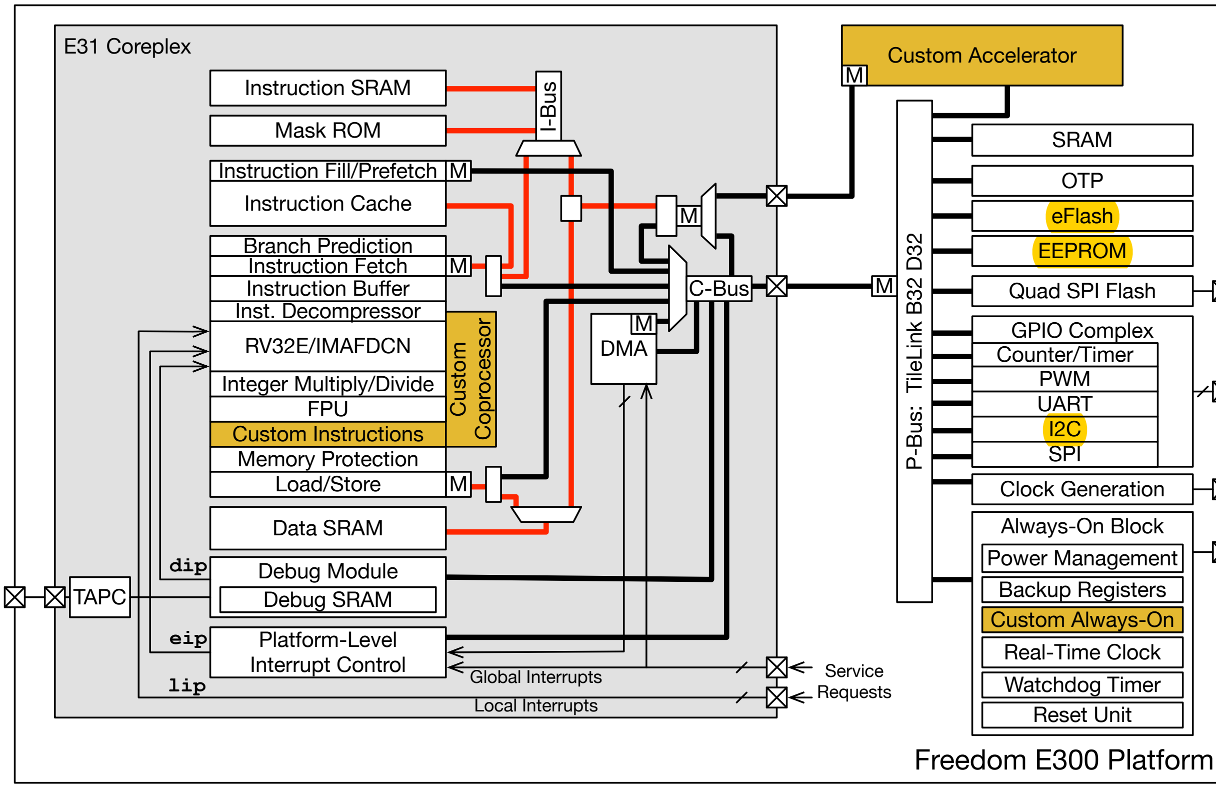


Figure 1.1: Top-Level Block Diagram of the E300 platform.

## Custom Accelerators

Custom autonomous accelerators can be added to provide application-specific processing. The custom accelerators can directly access on-chip memories and peripheral devices, and can generate and receive interrupts from the platform-level interrupt controller.

## On-Chip Memory

The on-chip memory system can be flexibly configured to include ROM, OTP, eFLASH, NVM/EEPROM, and/or SRAM of various sizes.

## Execute-in-Place Quad-SPI Flash controller

A dedicated Quad-SPI flash controller can be added with support for a a memory-mapped burst-read interface to support processor instruction cache or data cache refills from an external SPI flash memory. Memory burst writes are not supported. The external SPI flash has a set of control registers mapped into I/O space through which the external flash can be written under software control.

## Peripheral Devices

Peripheral devices can be selected from a large catalog of standard components, including counter/timers, watchdogs, PWM, GPIO, UART, I2C, SPI, ADC, DAC, SD/eMMC, USB 1.1/2.0 OTG, and 10/100/1000 Ethernet. The autonomous Coreplex DMA engine can be added to reduce processor overhead in servicing I/O transfers to and from data memory. Third-party peripheral IP can be attached via industry-standard SoC buses or TileLink. Please contact SiFive for details on the available peripheral offerings, or on how to connect to existing IP.

## Platform-Level Interrupt Controller

The configurable platform-level interrupt controller (PLIC) supports a large number of inputs and programmable priority levels, and with the addition of the N extension can also support nested interrupt handling for fast interrupt response.

## Always-On Block and Power Management

E300 SoCs can be configured with active power management to reduce leakage current in sleep mode. The Always-On Block (AON) supports low-power sleep with wakeup from an internal real-time clock interrupt or external I/O stimulus, or custom always-on circuitry.

## Debug Support

Each E300 system includes extensive platform-level debug facilities including hardware breakpoints, watchpoints, and single-step execution accessed via an industry-standard JTAG interface and supported by a full set of open-source debug tools. All components in the system, including the processor, accelerators, memories, peripheral devices, and interrupt controller, can be controlled and monitored over the debug port.

## Software Tools

SiFive provides a full open-source RISC-V embedded software development toolchain for E300 SoCs, including modern C and C++ compilers with soft-floating-point support, standard libraries, assemblers, linkers, and the FreeRTOS real-time operating system, together with debug tools to drive the on-chip debug hardware.

# Chapter 2

# E300 Platform Memory Map

The overall memory map of E300 is shown in Table 2.1.

| Base | Top | Description | |
|---|---|---|---|
| 0x0000_0000 | 0x0FFF_FFFF | *(see E3 Coreplex Manual)* | E3 Coreplex (256 MiB) |
| 0x1000_0000 | 0x1000_7FFF | Always-On (AON) (≤32 KiB) | Off-Coreplex I/O (1.75 GiB) |
| 0x1000_8000 | 0x1000_FFFF | Power, Reset, Clock, Interrupts (PRCI) (≤32 KiB) | |
| 0x1001_0000 | 0x1001_0FFF | On-chip OTP control | |
| 0x1001_1000 | 0x1001_1FFF | On-chip eFlash control | |
| 0x1001_2000 | 0x1001_2FFF | GPIO0 | |
| 0x1001_3000 | 0x1001_3FFF | UART0 | |
| 0x1001_4000 | 0x1001_4FFF | QSPI0 | |
| 0x1001_5000 | 0x1FFF_FFFF | Additional Peripherals (<256MiB) | |
| 0x2000_0000 | 0x3FFF_FFFF | Off-chip QSPI0 flash read (512 MiB) | |
| 0x4000_0000 | 0x7FFF_FFFF | Additional I/O or RAM (1 GiB) | |
| 0x8000_0000 | 0x8001_FFFF | Instruction and Data RAM (≤ 128 KiB) | Memory (2 GiB) |
| 0x8002_0000 | 0xFFFF_FFFF | Additional RAM | |

Table 2.1: E300 Physical Memory Map.

# Chapter 3

# E300 Power Modes

This chapter describes the different power modes available on E300 systems. E300 systems currently support three power modes: Run, Wait, and Sleep.

### Run Mode

Run mode corresponds to regular execution where the processor is running. Power consumption can be adjusted by varying the clock frequency of the processor and peripheral bus, and by enabling or disabling individual peripheral blocks. The processor exits run mode by executing a "Wait for Interrupt" (WFI) instruction.

### Wait Mode

When the processor executes a WFI instruction it enters Wait mode, which halts instruction execution and gates the clocks driving the processor pipeline. All state is preserved in the system. The processor will resume in Run mode when there is a local interrupt pending or when the PLIC sends an interrupt notification. The processor may also exit wait mode for other events, and software must check system status when exiting wait mode to determine the correct course of action.

### Sleep Mode

Sleep mode is entered by writing to a memory-mapped register `pmusleep` in the power-management unit (PMU). The `pmusleep` register is protected by the `pmukey` register which must be written with a defined value before writing to `pmusleep`.

The PMU will then execute a power-down sequence to turn off power to the processor and main pads. All volatile state in the system is lost except for state held in the AON domain. The main output pads will be left floating.

Sleep mode is exited when an enabled wakeup event occurs, whereupon the PMU will initiate a wakeup sequence. The wakeup sequence turns on the core and pad power supplies while asserting reset on the clocks, core and pads. After the power supplies stabilize, the clock reset is deasserted to allow the clocks to stabilize. Once the clocks are stable, the pad and processor resets are deasserted, and the processor begins running from the reset vector.

Software must reinitialize the core and can interrogate the PMU `pmucause` register to determine the cause of reset, and can recover pre-sleep state from the backup registers. The processor always initially runs from the HFROSC at the default setting, and must reconfigure clocks to run from an alternate clock source (HFXOSC or PLL) or at a different setting on the HFROSC.

# Chapter 4

# E300 Clock Generation

The Freedom E300 platform supports many alternative clock-generation schemes to match application needs. This chapter describes the basic structure of E300 clock generation. The various clock configuration registers live either in the AON block (Chapter 5) or the PRCI block (Chapter 7).

## Clock Generation Overview

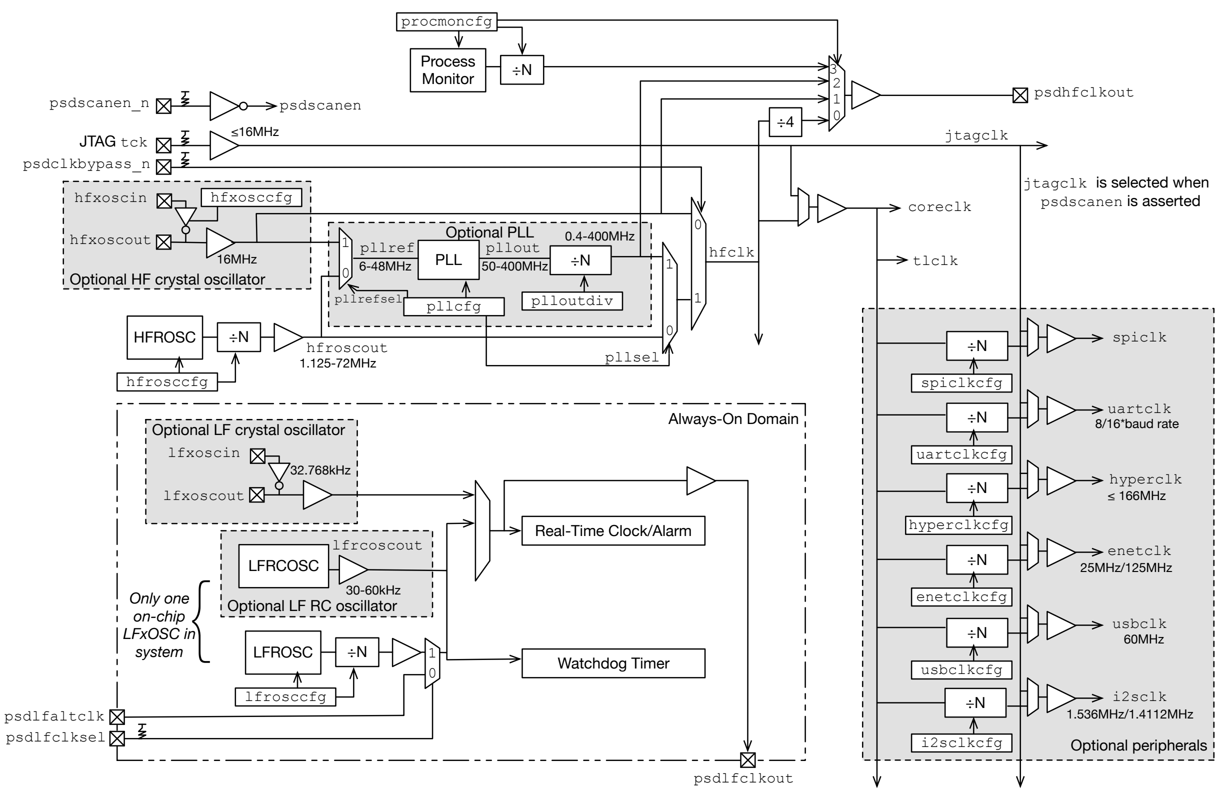


Figure 4.1: E300 clock generation scheme.

Figure 4.1 shows an overview of the E300 clock generation scheme. Most digital clocks on the chip are divided down from a central high-frequency clock `hfclk` produced from either the PLL or an on-chip trimmable oscillator. The PLL can be driven from either the on-chip oscillator or an off-chip crystal oscillator. In systems without a PLL, the off-chip oscillator can drive the high-frequency clock directly.

For the FE310-G000, the TileLink bus clock (`tlclk`) is fixed to be the same as the processor core clock (`coreclk`). As shown, each peripheral may also generate local divided clocks from `tlclk`.

The Always-On block includes a real-time clock circuit that is driven from one of three possible low-frequency clock sources: an off-chip 32 kHz crystal oscillator, an on-chip low-frequency RC oscillator, or a clock divided down from `hfclk`.

Test mode can select the JTAG test clk (TCK) to be driven into all clock trees to support scan.

## Internal Trimmable Programmable 72 MHz Oscillator (HFROSC)

An internal trimmable high-frequency ring oscillator (HFROSC) is used to provide the default clock after reset, and can be used to allow operation without an external high-frequency crystal or a PLL.

The oscillator is controlled by the `hfrosccfg` register, which is memory-mapped in the PRCI address space, and whose format is shown in Figure 4.1.

| 31 | 30 | 29 — 21 | 20 — 16 | 15 — 6 | 5 — 0 |
|---|---|---|---|---|---|
| `hfroscrdy` | `hfroscen` | 0 | `hfrosctrim` | 0 | `hfroscdiv` |
| 1 | 1 | 9 | 5 | 10 | 6 |

Table 4.1: The HFROSC config register, `hfrosccfg`.

The frequency can be adjusted in software using a 5-bit trim value in the `hfrosctrim`. The trim value (from 0–31) adjusts which tap of the variable delay chain is fed back to the start of the ring. A value of 0 corresponds to the longest chain and slowest frequency, while higher values correspond to shorter chains and therefore higher frequencies.

The HFROSC oscillator output frequency can be divided by an integer between 1 and 64 giving a frequency range of 1.125 MHz–72 MHz assuming the trim value is set to give a 72 MHz output. The value of the divider is given in the `hfroscdiv` field, where the divide ratio is one greater than the binary value held in the field (i.e., `hfroscdiv=0` indicates divide by 1, `hfroscdiv=1` indicates divide by 2, etc.). The value of the divider can be changed at any time.

The HFROSC is the default clock source used for the system core at reset. After a reset, the `hfrosctrim` value is reset to 16, the middle of the adjustable range, and the divider is reset to $\div$5 (`hfroscdiv=4`), which gives a nominal 13.8 MHz ($\pm$50%) output frequency.

The value of `hfrosctrim` that most closely achieves an 72 MHz clock output at nominal conditions (1.8 V at 25 C) is determined by manufacturing-time calibration and is stored in on-chip OTP storage. Upon reset, software in the processor boot sequence can write the calibrated value into the `hfrosctrim` field, but the value can be altered at any time during operation including when the processor is running from HFROSC.

To save power, the HFROSC can be disabled by clearing `hfroscen`. The processor must be running from a different clock source (the PLL, external crystal, or external clock) before disabling HFROSC. HFROSC can be explicitly renabled by setting `hfroscen`. HFROSC will be automatically re-enabled at every reset.

The status bit `hfroscrdy` indicates if the oscillator is operational and ready for use as a clock source.

## External 16 MHz Crystal Oscillator (HFXOSC)

An external high-frequency 16 MHz crystal oscillator can be used to provide a precise clock source. The crystal oscillator should have a capacitive load of $\leq$12 pF and an ESR $\leq$80 $\Omega$.

When used to drive the PLL, the 16 MHz crystal oscillator output frequency must be divided by two in the first-stage divider of the PLL (i.e., $R = 2$) to provide an 8 MHz reference clock to the VCO.

The input pad of the HFXOSC can also be used to supply an external clock source, in which case, the output pad should be left unconnected.

The HFXOSC input can be used to generate `hfclk` directly if there is no PLL present in the system, or if the PLL is set to bypass.

The HFXOSC is controlled via the memory-mapped `hfxosccfg` register.

| 31 | 30 | 29 — 0 |
|---|---|---|
| hfxoscrdy | hfxoscen | 0 |
| 1 | 1 | 30 |

Table 4.2: The HFXOSC config register, `hfxoscccfg`.

The `hfxoscen` bit turns on the crystal driver and is set after wakeup reset, but can be cleared to turn off the crystal driver and reduce power consumption. The `hfxoscrdy` bit indicates if the crystal oscillator output is ready for use.

The `hfxoscen` bit must also be turned on to use the HFXOSC input pad to connect an external clock source.

## Internal High-Frequency PLL (HFPLL)

The PLL generates a high-frequency clock by multiplying a mid-frequency reference source clock, either the HFROSC or the HFXOSC. The input frequency to the PLL can be in the range 6–48 MHz. The PLL can generate output clock frequencies in the range 48–384 MHz.

The PLL is controlled by a memory-mapped read-write `pllcfg` register in the PRCI address space. The format of `pllcfg` is shown in Figure 4.3.

Figure 4.2 shows how the PLL output frequency is set using a combination of three read-write fields: `pllr[2:0]`, `pllf[2:0]`, `pllq[1:0]`. The frequency constraints must be observed between each stage for correct operation.

The `pllr[1:0]` field encodes the reference clock divide ratio as a 2-bit binary value, where the value is one less than the divide ratio (i.e., `00`=1, `11`=4). The frequency of the output of the

| 31 | 30–19 | 18 | 17 | 16 | 15–12 | 11–10 | 9–4 | 3 | 2–0 |
|---|---|---|---|---|---|---|---|---|---|
| `plllock` | 0 | `pllbypass` | `pllrefsel` | `pllsel` | 0 | `pllq` | `pllf` | 0 | `pllr` |
| 1 | 12 | 1 | 1 | 1 | 4 | 2 | 6 | 1 | 3 |

Table 4.3: The PLL config register, `pllcfg`.

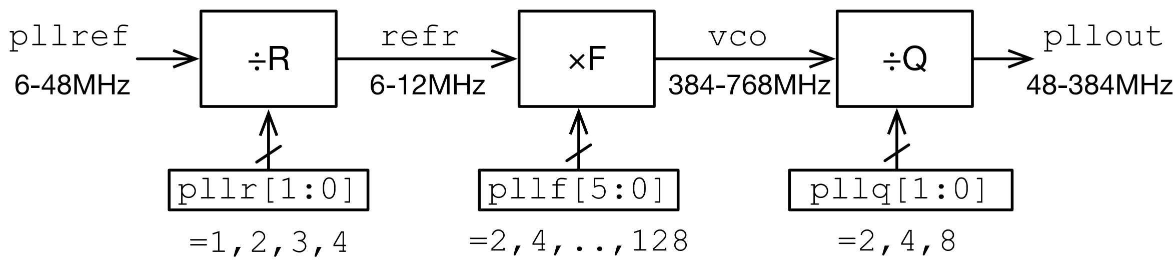


Figure 4.2: Controlling the E300 PLL output frequency.

reference divider (`refr`) must lie between 6–12 MHz.

The `pllf[5:0]` field encodes the PLL VCO multiply ratio as a 6-bit binary value, $N$, signifying a divide ratio of $2 \times (N + 1)$ (i.e., `000000`=2, `111111`=128). The frequency of the VCO output (`vco`) must lie between 384–768 MHz. Table 4.4 summarizes the valid settings of the multiply ratio.

| `refr` (MHz) | Legal `pllf` multiplier | | `vco` frequency (MHz) | |
|---|---|---|---|---|
| | Min | Max | Min | Max |
| 6 | 64 | 128 | 384 | 768 |
| 8 | 48 | 96 | 384 | 768 |
| 10 | 39 | 76 | 390 | 760 |
| 12 | 32 | 64 | 384 | 768 |

Table 4.4: Valid PLL multiply ratios. The multiplier setting in the table is given as the actual multiply ratio; the binary value stored in `pllf` field should be $(M/2) - 1$ for a multiply ratio $M$.

The `pllq[1:0]` field encodes the PLL output divide ratio as follow, `01`=2, `10`=4, `11`=8. The value `00` is not supported. The final output of the PLL must have a frequency that lies between 48–384 MHz.

The one-bit read-write `pllbypass` field in the `pllcfg` register turns off the PLL when written with a 1 and then `pllout` is driven directly by the clock indicated by `pllrefsel`. The other PLL registers can be configured when `pllbypass` is set. The agent that writes `pllcfg` should be running from a different clock source before disabling the PLL. The PLL is also disabled with `pllbypass`=1 after a wakeup reset.

The `pllsel` bit must be set to drive the final `hfclk` with the PLL output, bypassed or otherwise.

 

When `pllsel` is clear, the `hfroscclk` directly drives `hfclk`. The `pllsel` bit is clear on wakeup reset.

The `pllcfg` register is reset to: bypass and power off the PLL `pllbypass=1`; input driven from external HFXOSC oscillator `pllrefsel=1`; PLL not driving system clock `pllsel=0`; and the PLL ratios are set to R=2, F=64, and Q=8 (`pllr=01`, `pllf=011111`, `pllq=11`).

The PLL provides a lock signal which is set when the PLL has achieved lock, and which can be read from the most-significant bit of the `pllcfg` register. The PLL requires up to 100 $\mu$s to regain lock once enabled, and the lock signal will not necessarily be stable during this initial lock period so should only be interrogated after this period. The PLL may not achieve lock and the lock signal might not remain asserted if there is excessive jitter in the source clock.

The PLL requires dedicated 1.8 V power supply pads with a supply filter on the circuit board. The supply filter should be a 100 $\Omega$ resistor in series with the board 1.8 V supply decoupled with a 100 nF capacitor across the VDDPLL/VSSPLL supply pins. The VSSPLL pin should not be connected to board VSS.

## PLL Output Divider

The `plloutdiv` register controls a clock divider that divides the output of the PLL.

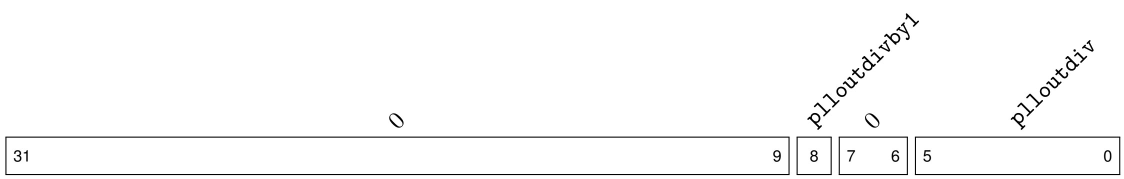


Figure 4.3: PLL Output Divider Register `plloutdiv`

If the `plloutdivby1` bit is set, the PLL output clock is passed through undivided. If `plloutdivby1` is clear, the value $N$ in `plloutdiv` sets the clock-divide ratio to $2\times(N+1)$ (between 2–128). The output divider expands the PLL output frequency range to 0.375–384 MHz.

The `plloutdivby1` register is reset to divide-by-1 (`plloutdivby1=1`)..

## Internal Low-Frequency Oscillator (LFRCOSC)

An untrimmed internal low-frequency RC oscillator can be provided with an operating frequency of 40-80 kHz. The internal low-frequency oscillator can be used to clock the always-on domain in lieu of an external crystal. A programmable prescalar is provided to allow runtime calibration of the low-frequency output to improve timing accuracy.

## External 32.768 kHz Low-Frequency Crystal Oscillator (LFXOSC)

A 32.768 kHz external crystal oscillator can be attached to provide a precise real-time clock. The oscillator can be turned off to save power but can require up to 1 s to stabilize.

# Chapter 5

# E300 Always-On (AON) Domain

The E300 platform supports an always-on (AON) domain that includes real-time counters, watchdog timers, backup registers, and reset and power-management circuitry for the rest of the system. Figure 5.1 shows an overview of the AON block.

### AON Power Source

The AON domain is continuously powered from an off-chip power source, either a regulated power supply or a battery.

### AON Clocking and Tilelink Slave Port

The AON block has a TileLink slave port to allow an external master to read and write registers inside the AON block. The AON domain is clocked by the low-frequency clock, `lfclk`. The core domain's Tilelink peripheral bus uses the high-frequency `tlclk`. A HF-LF power-clock-domain crossing (VCDC) bridges TileLink between the two power and clock domains.

### AON Reset Unit

An AON reset is the widest reset on an E300 system, and resets all state except for the JTAG debug interface.

An AON reset can be triggered by an on-chip power-on reset (POR) circuit when power is first applied to the AON domain, an external active-low reset pin (`erst_n`), or expiration of the watchdog timer (`wdogrst`).

These sources provide a short initial reset pulse which is extended by the reset stretcher to provide a shorter LFROSC reset signal `lfroscrst` and a longer stretched internal reset, `srst`.

The `lfroscrst` signal is used to initialize the ring oscillator in the LFROSC. This oscillator provides `lfclk`, which is used to clock the AON.

The `srst` strobe is passed to a reset synchronizer clocked by `lfclk` to generate `aonrst`, an asychronous-onset/synchronous-release reset signal used to reset most of the AON block.

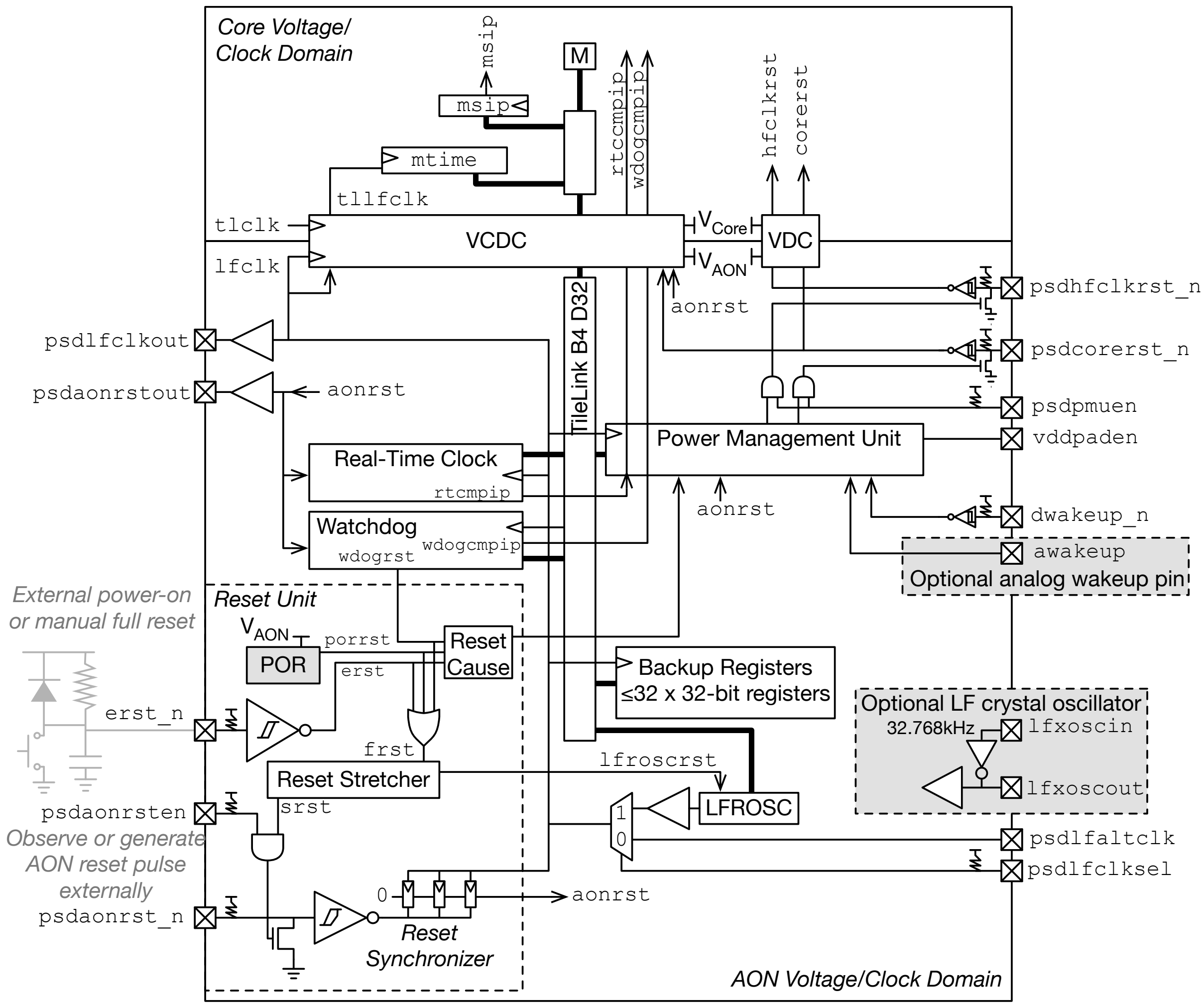


Figure 5.1: E300 Always-On Domain.

## Power-On Reset Circuit

This optional circuit holds it's output low until the voltage in the AON block rises above a design-time configurable preset threshold.

## External Reset Circuit

The E300 can be reset by pulling down on the external reset pin (`erst_n`), which has a weak pullup. An external power-on reset circuit consisting of a resistor and capacitor can be to provided to generate a sufficiently long pulse to allow supply voltage to rise and then initiate the reset stretcher.

The external reset circuit can add a diode as shown to quickly discharge the capacitor after the supply is removed to rearm the external power-on reset circuit.

A manual reset button can be connected in parallel over the capacitor.

### Reset Cause

The cause of an AON reset is latched in the Reset Unit and can be read from the `pmucause` register in the PMU.

### Watchdog Timer (WDT)

The watchdog timer can be used to provide a watchdog reset function, or a periodic timer interrupt. The watchdog is described in detail in Chapter 8.

### Real-Time Clock (RTC)

The real-time clock maintains time for the system and can also be used to generate interrupts for timed wakeup from sleep-mode or timer interrupts during normal operation. The Real-Time Clock is described in detail in Chapter 9.

### Backup Registers

The backup register provide a configurable number of 32-bit data registers that hold state during sleep. The FE310-G000 has 16×32-bit backup registers. The backup registers are described in detail in Chapter 10.

### Power-Management Unit (PMU)

The power-management unit (PMU) sequences the system power supplies and reset signals when transitioning into and out of sleep mode. The PMU also monitors AON signals for wakeup conditions. The PMU is described in detail in Chapter 6.

### AON Memory Map

Table 5.1 shows the memory map of the AON block.

| Address | Description | |
|---|---|---|
| 0x1000_0000 | wdogcfg | Watchdog Timer Registers |
| 0x1000_0004 | *Reserved* | |
| 0x1000_0008 | wdogcount | |
| 0x1000_000C | *Reserved* | |
| 0x1000_0010 | wdogs | |
| 0x1000_0014 | *Reserved* | |
| 0x1000_0018 | wdogfeed | |
| 0x1000_001C | wdogkey | |
| 0x1000_0020 | wdogcmp | |
| ... | | |
| 0x1000_0040 | rtccfg | Real-Time Clock Registers |
| 0x1000_0044 | *Reserved* | |
| 0x1000_0048 | rtclo | |
| 0x1000_004C | rtchi | |
| 0x1000_0050 | rtcs | |
| 0x1000_0054 | *Reserved* | |
| 0x1000_0058 | *Reserved* | |
| 0x1000_005C | *Reserved* | |
| 0x1000_0060 | rtccmp | |
| ... | | |
| 0x1000_0070 | lfrosccfg | AON Clock Configuration Registers |
| ... | | |
| ... | | |
| 0x1000_0080 | backup0 | Backup Registers |
| 0x1000_0084 | backup1 | |
| ... | | |
| 0x1000_00FC | backup31 | |
| 0x1000_0100 | PMU wakeup program memory | Power Management Unit |
| 0x1000_0120 | PMU sleep program memory | |
| 0x1000_0140 | pmuie | |
| 0x1000_0144 | pmucause | |
| 0x1000_0148 | pmusleep | |
| 0x1000_014C | pmukey | |

Table 5.1: SiFive AON Memory Map.

# Chapter 6

# E300 Power-Management Unit (PMU)

The E300 power-management unit (PMU) is implemented within the AON domain and sequences the system's power supplies and reset signals during power-on reset and when transitioning the "mostly off" (MOFF) block into and out of sleep mode.

## PMU Overview

The PMU is a synchronous unit clocked by the `lfclk` in the AON domain. The PMU handles reset, wakeup, and sleep actions initiated by power-on reset, wakeup events, and sleep requests. When the MOFF block is powered off, the PMU monitors AON signals to initiate the wakeup sequence. When the MOFF block is powered on, the PMU awaits sleep requests from the MOFF block, which initiate the sleep sequence. The PMU is based around a simple programmable microcode sequencer that steps through short programs to sequence output signals that control the power supplies and reset signals to the clocks, core, and pads in the system.

## PMU Key Register (`pmukey`)

The `pmukey` register has one bit of state. To prevent spurious sleep or PMU program modification, all writes to PMU registers must be preceded by an unlock operation to the `pmukey` register location, which sets `pmukey`. The value `0x51F15E` must be written to the `pmukey` register address to set the state bit before any write access to any other PMU register. The state bit is reset at AON reset, and after any write to a PMU register.

PMU registers may be read without setting `pmukey`.

## PMU Program

The PMU is implemented as a programmable sequencer to support customization and tuning of the wakeup and sleep sequences. A wakeup or sleep program comprises eight instructions. An instruction consists of a delay, encoded as a binary order of magnitude, and a new value for all of the PMU output signals to assume after that delay. The PMU instruction format is shown in Figure 6.2. For example, the instruction `0x108` delays for $2^8$ clock cycles, then raises `hfclkrst` and lowers all other output signals.

The PMU output signals are registered and only toggle on PMU instruction boundaries. The output registers are all asynchronously set to 1 by `aonrst`.

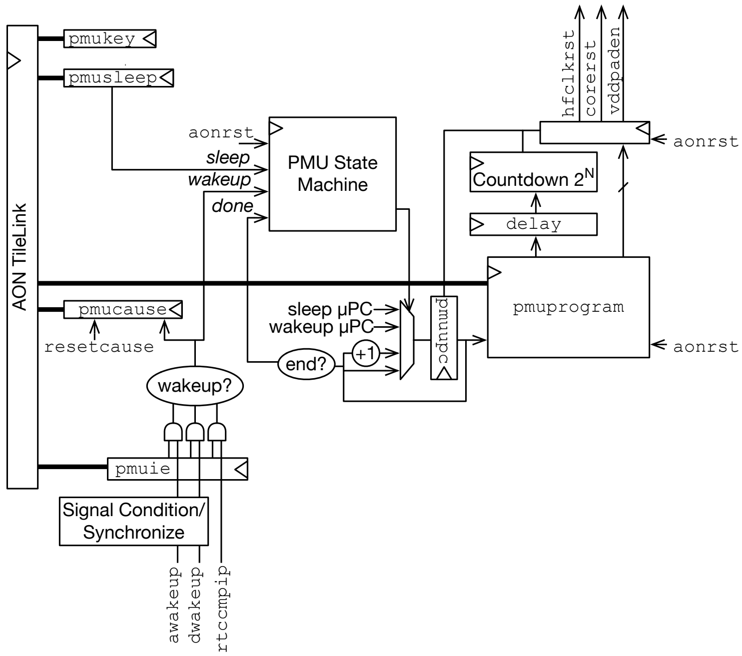


Figure 6.1: E300 Power-Management Unit.

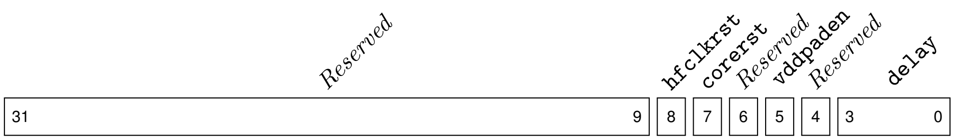


Figure 6.2: PMU instruction format.

At power-on reset, the PMU program memories are reset to conservative defaults. Table 6.1 shows the default wakeup program, and Table 6.2 shows the default sleep program.

| Index | Value | Meaning |
|---|---|---|
| 0 | 0x1f0 | Assert all resets and enable all power supplies |
| 1 | 0x0f8 | Idle $2^8$ cycles, then deassert `hfclkrst` |
| 2 | 0x030 | Deassert `corerst` and `padrst` |
| 3–7 | 0x030 | *Repeats* |

Table 6.1: Default PMU wakeup program.

### Initiate Sleep Sequence Register (`pmusleep`)

Writing any value to the `pmusleep` register initiates the sleep sequence stored in the sleep program memory. The MOFF block will sleep until an event enabled in the `pmuie` register occurs.

| Index | Value | Meaning |
|---|---|---|
| `0` | 0x0f0 | Assert `corerst` |
| `1` | 0x1f0 | Assert `hfclkrst` |
| `2` | 0x1d0 | Deassert `vddpaden` |
| `3` | 0x1c0 | Deassert *Reserved* |
| `4-7` | 0x1c0 | *Repeats* |

Table 6.2: Default PMU sleep program.

### Wakeup Signal Conditioning

The PMU can be woken by external signals, `dwakeup` and `awakeup`, which are preconditioned by the signal conditioning block.

Currently, the `dwakeup` signal has a fixed deglitch circuit that requires the `dwakeup` signal remain asserted for two AON clock edges before being accepted. The conditioning circuit also resynchronizes the `dwakeup` signal to the AON `lfclk`.

The `awakeup` analog intput is not yet supported on E300 systems.

### PMU Interrupt Enables (`pmuie`) and Wakeup Cause (`pmucause`)

The `pmuie` register indicates which events can wake the MOFF block from sleep. The `awakeup` bit indicates that the `awakeup` pin can rouse MOFF. The `dwakeup` bit indicates that a logic 0 on the `dwakeup_n` pin can rouse MOFF. The `rtc` bit indicates that the RTC comparator can rouse MOFF.

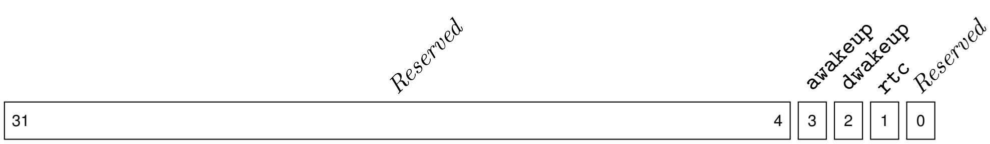


Figure 6.3: Format of `pmuie` register.

Following a wakeup, the `pmucause` register indicates which event caused the wakeup. The value in the `wakeupcause` field corresponds to the bit position of the event in `pmuie`, e.g., a value of 2 indicates `dwakeup`. The value 0 indicates a wakeup from reset.

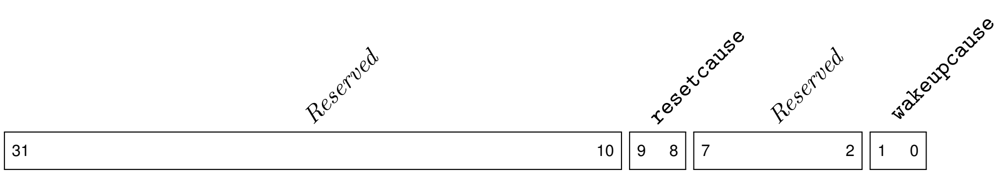


Figure 6.4: Format of `pmucause` register.

In the event of a wakeup from reset, the `resetcause` field indicates which reset source triggered the wakeup. Table 6.3 lists the values the `resetcause` field may take. The value in `resetcause` persists until the next reset.

| Index | Meaning |
|---|---|
| 0 | Power-on reset |
| 1 | External reset |
| 2 | Watchdog timer reset |

Table 6.3: Reset cause values.

### Memory Map

The memory map for the PMU is shown in Table 6.4. The memory map has been designed to only require naturally aligned 32-bit memory accesses.

| Address | Name | Description |
|---|---|---|
| 0x100 | pmuwakeupi0 | Wakeup program instruction 0 |
| 0x104 | pmuwakeupi1 | Wakeup program instruction 1 |
| 0x108 | pmuwakeupi2 | Wakeup program instruction 2 |
| 0x10c | pmuwakeupi3 | Wakeup program instruction 3 |
| 0x110 | pmuwakeupi4 | Wakeup program instruction 4 |
| 0x114 | pmuwakeupi5 | Wakeup program instruction 5 |
| 0x118 | pmuwakeupi6 | Wakeup program instruction 6 |
| 0x11c | pmuwakeupi7 | Wakeup program instruction 7 |
| 0x120 | pmusleepi0 | Sleep program instruction 0 |
| 0x124 | pmusleepi1 | Sleep program instruction 1 |
| 0x128 | pmusleepi2 | Sleep program instruction 2 |
| 0x12c | pmusleepi3 | Sleep program instruction 3 |
| 0x130 | pmusleepi4 | Sleep program instruction 4 |
| 0x134 | pmusleepi5 | Sleep program instruction 5 |
| 0x138 | pmusleepi6 | Sleep program instruction 6 |
| 0x13c | pmusleepi7 | Sleep program instruction 7 |
| 0x140 | pmuie | PMU interrupt enables |
| 0x144 | pmucause | PMU wakeup cause |
| 0x148 | pmusleep | Initiate sleep sequence |
| 0x14c | pmukey | PMU key register |

Table 6.4: SiFive PMU register offsets within AON memory map. Only naturally aligned 32-bit memory accesses are supported.

# Chapter 7

# E300 Power, Reset, Clock, Interrupt (PRCI) Control and Status Registers

PRCI is an umbrella term for platform non-AON memory-mapped control and status registers controlling component power states, resets, clock selection, and low-level interrupts, hence the name. The PRCI registers are generally only made visible to machine-mode software. The AON block contains registers with similar functions, but only for the AON block units.

## PRCI Address Space Usage

Table 7.1 shows the memory map for PRCI on SiFive systems.

| Address | Description | |
|---|---|---|
| 0x1000_8000 | hfrosccfg | Clock Configuration Registers |
| 0x1000_8004 | hfxosccfg | |
| 0x1000_8008 | pllcfg | |
| 0x1000_800c | plloutdiv | |
| 0x1000_8010 | coreclkcfg | |

Table 7.1: SiFive E300 PRCI Memory Map.

# Chapter 8

# E300 Watchdog Timer (WDT)

The watchdog timer (WDT) is used to cause a full power-on reset if either hardware or software errors cause the system to malfunction. The WDT can also be used as a programmable periodic interrupt source if the watchdog functionality is not required. The WDT is implemented as an upcounter in the Always-On domain that must be reset at regular intervals before the count reaches a preset threshold, else it will trigger a full power-on reset. To prevent errant code from resetting the counter, the WDT registers can only be updated by presenting a WDT key sequence.

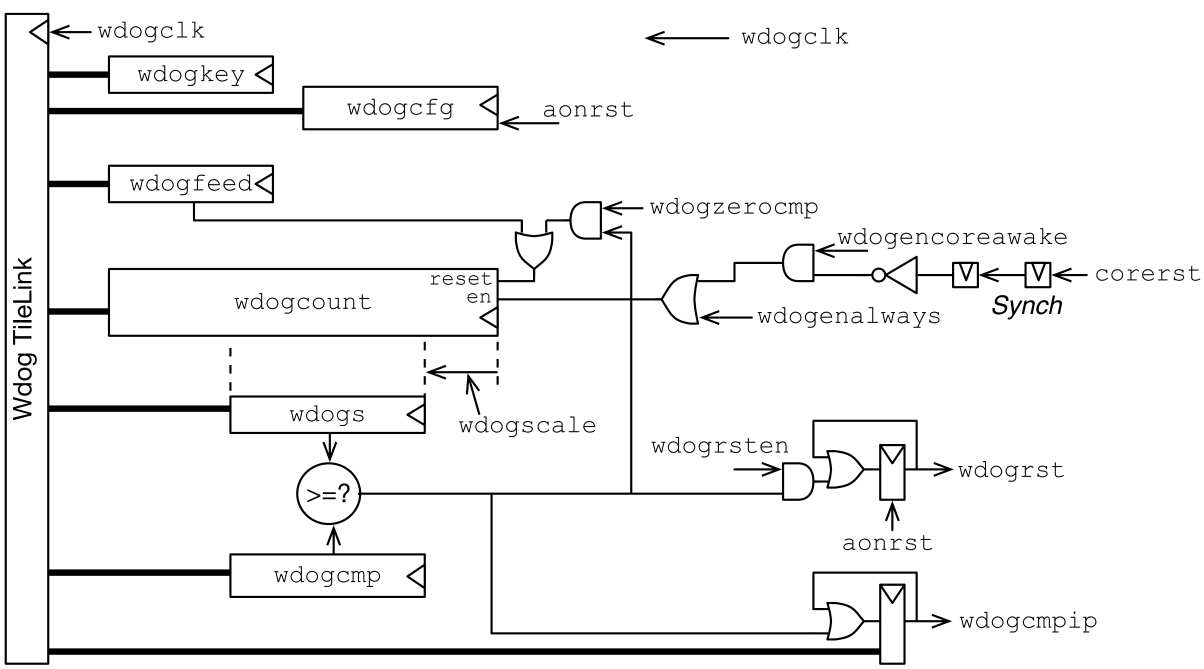


Figure 8.1: E300 Watchdog Timer.

### Watchdog Count Register (`wdogcount`)

The WDT is based around a 31-bit counter held in `wdogcount[30:0]`. The counter can be read or written over the TileLink bus. Bit 31 of `wdogcount` returns a zero when read.

The counter is incremented at a maximum rate determined by the watchdog clock selection. Each cycle, the counter can be conditionally incremented depending on the existence of certain conditions, including always incrementing or incrementing only when the processor is not asleep.

The counter can also be reset to zero depending on certain conditions, such as a successful write to `wdogfeed` or the counter matching the compare value.

### Watchdog Clock Selection

The WDT unit clock, `wdogclk`, is either driven from LFXOSC or LFRCOSC and runs at approximately 32 kHz.

### Watchdog Configuration Register `wdogcfg`


Figure 8.2: Watchdog configuration register `wdogcfg`

The `wdogen*` bits control the conditions under which the watchdog counter `wdogcount` is incremented. The `wdogenalways` bit if set means the watchdog counter always increments. The `wdogencoreawake` bit if set means the watchdog counter increments if the processor core is not asleep. The WDT uses the `corerst` signal from the wakeup sequencer to know when the core is sleeping. The counter increments by one each cycle only if any of the enabled conditions are true. The `wdogen*` bits are reset on AON reset.

The 4-bit `wdogscale` field scales the watchdog counter value before feeding it to the comparator. The value in `wdogscale` is the bit position within the `wdogcount` register of the start of a 16-bit `wdogs` field. A value of 0 in `wdogscale` indicates no scaling, and `wdogs` would then be equal to the low 16 bits of `wdogcount`. The maximum value of 15 in `wdogscale` corresponds to dividing the clock rate by $2^{15}$, so for an input clock of 32.768 kHz, the LSB of `wdogs` will increment once per second.

The value of `wdogs` is memory-mapped and can be read as a single 16-bit value over the AON TileLink bus.

The `wdogzerocmp` bit, if set, causes the watchdog counter `wdogcount` to be automatically reset to zero one cycle after the `wdogs` counter value matches or exceeds the compare value in `wdogcmp`. This feature can be used to implement periodic counter interrupts, where the period is independent of interrupt service time.

The `wdogrsten` bit controls whether the comparator output can set the `wdogrst` bit and hence cause a full reset.

The `wdogcmpip` interrupt pending bit can be read or written.

 

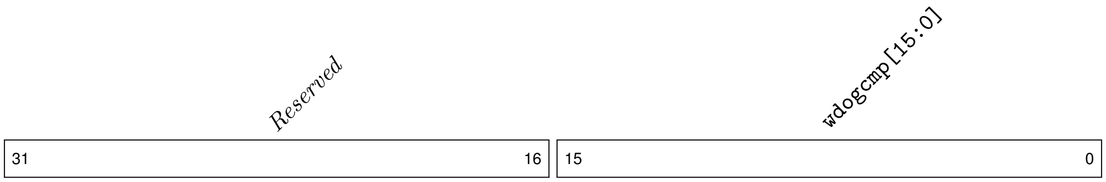


Figure 8.3: Watchdog compare register `wdogcmp`

```
li t0, 0x51F15E  # Obtain key.
sw t0, wdogkey   # Unlock kennel.
li t0, 0xD09F00D # Get some food.
sw t0, wdogfeed  # Feed the watchdog.
```

Figure 8.4: Sequence to reinitialize watchdog.

### Watchdog Compare Register (`wdogcmp`)

The compare register is a 16-bit value against which the current `wdogs` value is compared every cycle. The output of the comparator is asserted whenever the value of `wdogs` is greater than or equal to `wdogcmp`.

### Watchdog Key Register (`wdogkey`)

The `wdogkey` register has one bit of state. To prevent spurious reset of the WDT, all writes to `wdogcfg`, `wdogfeed`, `wdogcount`, `wdogs`, `wdogcmp` and `wdogcmpip` must be preceded by an unlock operation to the `wdogkey` register location, which sets `wdogkey`. The value `0x51F15E` must be written to the `wdogkey` register address to set the state bit before any write access to any other watchdog register. The state bit is reset at AON reset, and after any write to a watchdog register.

Watchdog registers may be read without setting `wdogkey`.

### Watchdog Feed Address (`wdogfeed`)

After a successful key unlock, the watchdog can be fed using a write of the value `0xD09F00D` to the `wdogfeed` address, which will reset the `wdogcount` register to zero. The full watchdog feed sequence is shown in Figure 8.4.

Note there is no state associated with the `wdogfeed` address. Reads of this address return 0.

### Watchdog Configuration

The WDT provides watchdog intervals of up to over 18 hours (≈65,535 seconds).

### Watchdog Resets

If the watchdog is not fed before the `wdogcount` register exceeds the compare register zero while the WDT is enabled, a reset pulse is sent to the reset circuitry, and the chip will go through a complete power-on sequence.

The WDT will be initalized after a full reset, with mode bit cleared.

### Watchdog Interrupts (`wdogcmpip`)

The WDT can be configured to provide periodic counter interrupts by disabling watchdog resets (`wdogrsten`=0) and enabling auto-zeroing of the count register when the comparator fires (`wdogzerocmp`=1).

The sticky single-bit `wdogcmpip` register captures the comparator output and holds it to provide an interrupt pending signal. The `wdogcmpip` register resides in bit 28 of the `wdogcfg` register, and can be read and written over TileLink to clear down the interrupt.

# Chapter 9

# E300 Real-Time Clock (RTC)

The E300 real-time clock (RT) is located in the always-on domain, and is clocked by a selectable low-frequency clock source. For best accuracy, the RTC should be driven by an external 32.768 kHz watch crystal oscillator (LFXOSC), but to reduce cost, can be driven by a factory-trimmed on-chip oscillator.

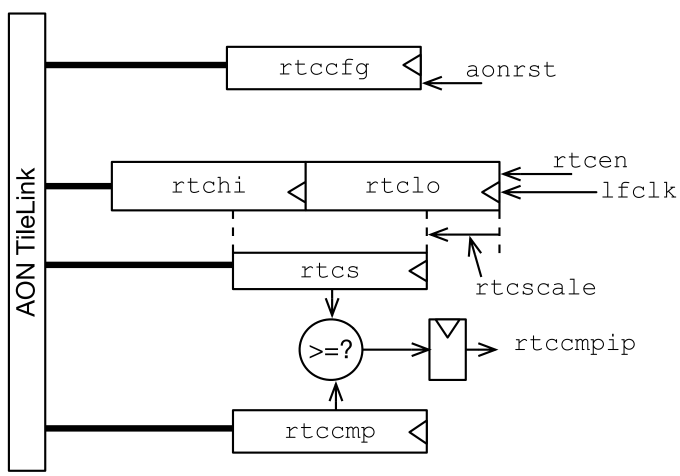


Figure 9.1: E300 Real-Time Clock.

### RTC Count Registers `rtchi`/`rtclo`

The real-time counter is based around the `rtchi`/`rtclo` register pair, which increment at the low-frequency clock rate when the RTC is enabled. The `rtclo` register holds the low 32 bits of the RTC, while `rtchi` holds the upper 16 bits of the RTC value. The total ≥48-bit counter width ensures there will no counter rollover for over 270 years assuming a 32.768 kHz low-frequency real-time clock source. The counter registers can be read or written over the TileLink bus.

### RTC Configuration Register `rtccfg`

The `rtcenalways` bit controls whether the RTC is enabled, and is reset on AON reset.

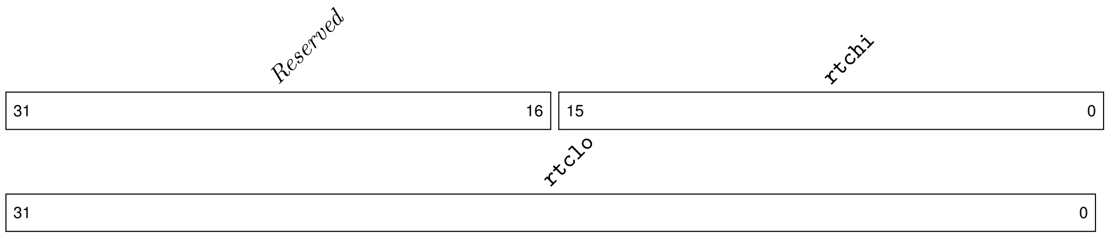


Figure 9.2: RTC counter register pair `rtchi`/`rtclo`

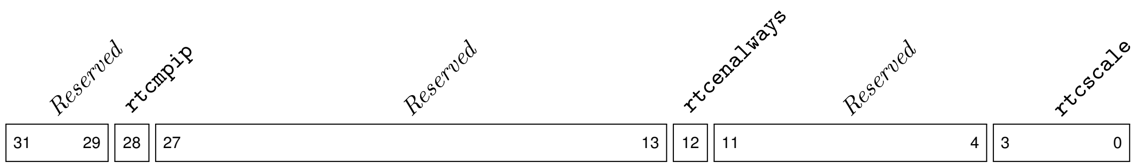


Figure 9.3: RTC configuration register `rtccfg`

The 4-bit `rtcscale` field scales the real-time counter value before feeding to the real-time interrupt comparator. The value in `rtcscale` is the bit position within the `rtclo`/`rtchi` register pair of the start of a 32-bit field `rtcs`. A value of 0 in `rtcscale` indicates no scaling, and `rtcs` would then be equal to `rtclo`. The maximum value of 15 in `rtcscale` corresponds to dividing the clock rate by $2^{15}$, so for an input clock of 32.768 kHz, the LSB of `rtcs` will increment once per second. The value of `rtcs` is memory-mapped and can be read as a single 32-bit register over the AON TileLink bus.

The `rtccmpip` interrupt pending bit is read-only.

### RTC Compare Register `rtccmp`

The `rtccmp` register holds a 32-bit value that is compared against `rtcs`, the scaled real-time clock. If `rtcs` is greater than or equal to `rtccmp`, the `rtccmpip` interrupt pending bit is set. The `rtccmpip` bit can be cleared down by writing a value to `rtccmp` that is greater than `rtcs`.

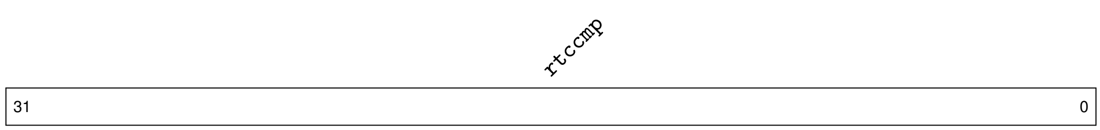


Figure 9.4: RTC counter compare register `rtccmp`

# Chapter 10

# E300 Backup Registers

The backup registers live in the Always-On domain, and provide a place to store critical data during sleep. Each register is 32-bits wide, and the number of backup registers is a configurable option.

# Chapter 11

# General Purpose Input/Output Controller (GPIO)

This chapter describes the operation of the General Purpose Input/Output Controller (GPIO) on SiFive systems. The SiFive GPIO controller is a peripheral device mapped in the internal memory map, discoverable in the Configuration String. It is responsible for low-level configuration of the actual GPIO pads on the device (direction, pull up-enable, and drive value), as well as selecting between various sources of the controls for these signals. The GPIO controller allows seperate configuration of each of $N$ GPIO bits. Figure 11.1 shows the control structure for each pin.

Atomic operations such as toggles are natively possible with the RISC-V 'A' extension.

### Memory Map

The memory map for the SiFive GPIO control registers is shown in Table 11.1. The GPIO memory map has been designed to only require naturally aligned 32-bit memory accesses.

### Input / Output Values

The same `port` register can be configured on a bitwise fashion to represent either inputs or outputs, as set by the `direction` register. Writing to the `port` register will update the bits regardless of the tristate value. Reading the `port` register will return the written value. Reading the `value` register will return the actual value of the pin.

In other words, on a read:

```
value = (input & direction) | (output & ~direction)
```

### Interrupts

A single interrupt bit can be generated for each GPIO bit. The interrupt can be driven by rising or falling edges, or by level values, and each can be enabled individually.

Inputs are synchronized before being sampled by the interrupt logic, so the input pulse width must be long enough to be detected by the synchronization logic.

To enable an interrupt, set the corresponding bit in the `rise_ie` and/or `fall_ie` to 1. If the correpsonding bit in `rise_ip` or `fall_ip` is set, an interrupt pin will be raised.

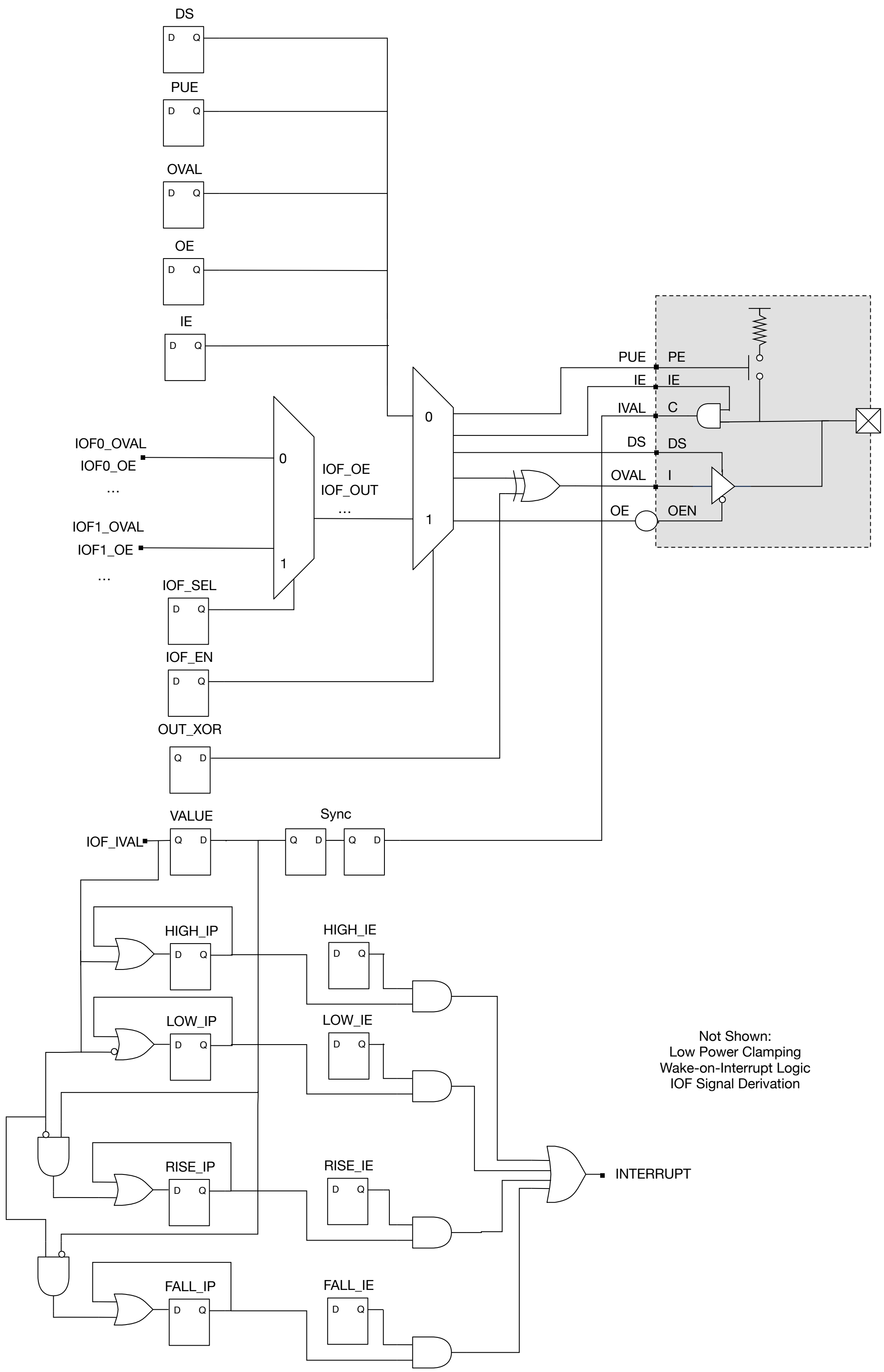


Figure 11.1: Structure of a single GPIO Pin with Control Registers. This structure is repeated for each pin.

| Address | Name | Description |
|---|---|---|
| 0x000 | value | pin value |
| 0x004 | input_en | * pin input enable |
| 0x008 | output_en | * pin output enable |
| 0x00C | port | output port value |
| 0x010 | pue | * internal pull-up enable |
| 0x014 | ds | Pin Drive Strength |
| 0x018 | rise_ie | rise interrupt enable |
| 0x01C | rise_ip | rise interrupt pending |
| 0x020 | fall_ie | fall interrupt enable |
| 0x024 | fall_ip | fall interrupt pending |
| 0x028 | high_ie | high interrupt enable |
| 0x02C | high_ip | high interrupt pending |
| 0x030 | low_ie | low interrupt enable |
| 0x034 | low_ip | low interrupt pending |
| 0x038 | iof_en | * HW I/O Function enable |
| 0x03C | iof_sel | HW I/O Function select |
| 0x040 | out_xor | Output XOR (invert) |

Table 11.1: SiFive GPIO Register Offsets. Only naturally aligned 32-bit memory accesses are supported. Registers marked with an * are asynchronously reset to 0. All other registers are synchronously reset to 0.

Once the interrupt is pending, it will remain set until a 1 is written to the `*_ip` register at that bit.

The interrupt pins may be routed to the PLIC, or directly to local interrupts.

### Internal Pull-Ups

When configured as inputs, each pin has an internal pull-up which can be enabled by software. At reset, all pins are set as inputs and pull-ups are disabled.

### Drive Strength

When configured as output, each pin has a SW-controllable Drive Strength.

### Output Inversion

When configured as an output (either SW or IOF controlled), the SW-writable `out_xor` register is combined with the output to invert it.

### HW I/O Functions (IOF)

Each GPIO pin can implement up to 2 HW-Driven functions (IOF) enabled with the `iof_en` register. Which IOF is used is selected with the `iof_sel` register.

When a pin is set to perform an IOF, it is possible that the software registers `port`, `output_en`, `pullup`, `ds`, `input_en` may not be used to control the pin directly. Rather, the pins may be controlled by hardware driving the IOF. Which functionalities are controlled by the IOF and which are controlled by the software registers are fixed in the hardware on a per-IOF basis. Those that are not controlled by the hardware continue to be controlled by the software registers.

If there is no IOFx for a pin configured with IOFx, the pin reverts to full software control.

### Behavior During Sleep Mode

# Chapter 12

# Universal Asynchronous Receiver/Transmitter (UART)

This chapter describes the operation of the SiFive Universal Asynchronous Receiver/Transmitter (UART).

## UART Overview

The UART peripheral supports the following features:

- 8-N-1 and 8-N-2 formats: 8 data bits, no parity bit, 1 start bit, 1 or 2 stop bits
- 8-entry transmit and receive FIFO buffers with programmable watermark interrupts
- 16$\times$ Rx oversampling with 2/3 majority voting per bit

The UART peripheral does not support hardware flow control or other modem control signals, or synchronous serial data tranfesrs.

## Memory Map

The memory map for the UART control registers is shown in Table 12.1. The UART memory map has been designed to only require naturally aligned 32-bit memory accesses.

| Address | Name | Description |
|---|---|---|
| `0x000` | `txdata` | Transmit data register |
| `0x004` | `rxdata` | Receive data register |
| `0x008` | `txctrl` | Transmit control register |
| `0x00C` | `rxctrl` | Receive control register |
| `0x010` | `ie` | UART interrupt enable |
| `0x014` | `ip` | UART Interrupt pending |
| `0x018` | `div` | Baud rate divisor |

Table 12.1: Register offsets within UART memory map.

### Transmit Data Register (`txdata`)

Writing to the `txdata` register enqueues the character contained in the `data` field to the transmit FIFO if the FIFO is able to accept new entries. Reading from `txdata` returns the current value of the `full` flag and zero in the `data` field. The `full` flag indicates whether the transmit FIFO is able to accept new entries; when set, writes to `data` are ignored. A RISC-V `amoswap` instruction can be used to both read the `full` status and attempt to enqueue data, with a non-zero return value indicating the character was not accepted.

| full | 0 | data |
|---|---|---|
| 31 | 30 ... 8 | 7 ... 0 |

Figure 12.1: Format of `txdata` register.

### Receive Data Register (`rxdata`)

Reading the `rxdata` register dequeues a character from the receive FIFO, and returns the value in the `data` field. The `empty` flag indicates if the receive FIFO was empty; when set, the `data` field does not contain a valid character. Writes to `rxdata` are ignored.

| empty | 0 | data |
|---|---|---|
| 31 | 30 ... 8 | 7 ... 0 |

Figure 12.2: Format of `rxdata` register.

### Transmit Control Register (`txctrl`)

The read-write `txctrl` register controls the operation of the transmit channel. The `txen` bit controls whether the Tx channel is active. When cleared, transmission of Tx FIFO contents is suppressed, and the `txd` pin is driven high.

The `nstop` field specifies the number of stop bits: `0` for one stop bit and `1` for two stop bits.

The `txcnt` field specifies the threshold at which the Tx FIFO watermark interrupt triggers.

The `txctrl` register is reset to `0`.

| *Reserved* | txcnt | *Reserved* | nstop | txen |
|---|---|---|---|---|
| 31 ... 19 | 18 ... 16 | 15 ... 2 | 1 | 0 |

Figure 12.3: Format of `txctrl` register.

### Receive Control Register (`rxctrl`)

The read-write `rxctrl` register controls the operation of the receive channel. The `rxen` bit controls whether the Rx channel is active. When cleared, the state of the `rxd` pin is ignored, and no characters will be enqueued into the Rx FIFO.

The `rxcnt` field specifies the threshold at which the Rx FIFO watermark interrupt triggers.

The `rxctrl` register is reset to `0`.

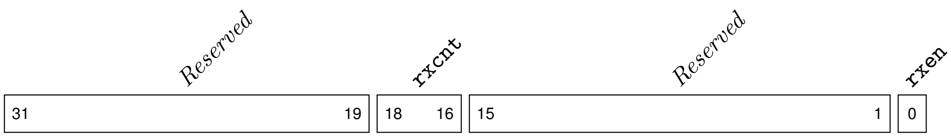


Figure 12.4: Format of `rxctrl` register.

### Interrupt Registers (`ip` and `ie`)

The `ip` register is a read-only register indicating the pending interrupt conditions, and the read-write `ie` register controls which UART interrupts are enabled. `ie` is reset to `0`.

The `txwm` condition becomes raised when the number of entries in the transmit FIFO is strictly less than the count specified by the `txcnt` field of the `txctrl` register. The pending bit is cleared when sufficient entries have been enqueued to exceed the watermark.

The `rxwm` condition becomes raised when the number of entries in the receive FIFO is strictly greater than the count specified by the `rxcnt` field of the `rxctrl` register. The pending bit is cleared when sufficient entries have been dequeued to fall below the watermark.

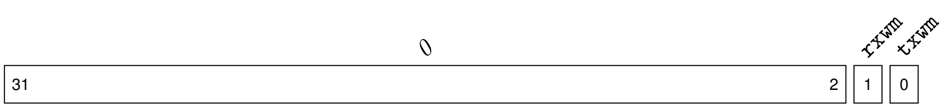


Figure 12.5: Format of `ie` and `ip` registers.

### Baud Rate Divisor Register (`div`)

The read-write `div` register specifies the divisor used by baud rate generation for both Tx and Rx channels. The relationship between the input clock and baud rate is given by the following formula:

$$f_{\text{baud}} = \frac{f_{\text{in}}}{\text{div} + 1}$$

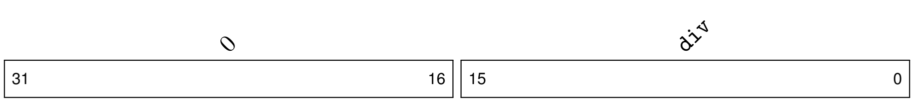


Figure 12.6: Format of `div` register.

The input clock is the bus clock `tlclk`. Table 12.2 shows divisors for some common core clock rates and commonly used baud rates. Note the table shows the divide ratios, which are one greater than the value stored in the `div` register.

| `tlclk` (MHz) | Target Baud (Hz) | Divisor | Actual Baud (Hz) | Error (%) |
|---:|---:|---:|---:|---|
| 2 | 31250 | 64 | 31250 | 0 |
| 2 | 115200 | 17 | 117647 | 2.12 |
| 16 | 31250 | 512 | 31250 | 0 |
| 16 | 115200 | 139 | 115108 | 0.08 |
| 16 | 250000 | 64 | 250000 | 0 |
| 200 | 31250 | 6400 | 31250 | 0 |
| 200 | 115200 | 1736 | 115207 | 0.0064 |
| 200 | 250000 | 800 | 250000 | 0 |
| 200 | 1843200 | 109 | 1834862 | 0.45 |
| 384 | 31250 | 12288 | 31250 | 0 |
| 384 | 115200 | 3333 | 115212 | 0.01 |
| 384 | 250000 | 1536 | 250000 | 0 |
| 384 | 1843200 | 208 | 1846154 | 0.16 |

Table 12.2: Common baud rates (MIDI=31250, DMX=250000) and required divide values to achieve them with given bus clock frequencies. The divide values are one greater than the value stored in the `div` register.

The receive channel is sampled at 16$\times$ the baud rate, and a majority vote over 3 neighboring bits is used to determine the received value. For this reason, the divisor must be $\geq$16 for a receive channel.

# Chapter 13

# Serial Peripheral Interface (SPI)

This chapter describes the operation of the SiFive Serial Peripheral Interface (SPI) controller.

## SPI Overview

The SPI controller supports master-only operation over the single-lane, dual-lane, and quad-lane protocols. The baseline controller provides a FIFO-based interface for performing programmed I/O. Software initiates a transfer by enqueuing a frame in the transmit FIFO; when the transfer completes, the slave response is placed in the receive FIFO.

In addition, the dedicated SPI0 controller implements a SPI flash read sequencer, which exposes the external SPI flash contents as a read/execute-only memory-mapped device. The SPI0 controller is reset to a state which allows memory-mapped reads, under the assumption that the input clock rate is less than 100 MHz and the external SPI flash device supports the common Winbond/Numonyx serial read (`0x03`) command. Sequential accesses are automatically combined into one long read command for higher performance.

The `fctrl` register controls switching between the memory-mapped and programmed-I/O modes. While in programmed-I/O mode, memory-mapped reads do not access the external SPI flash device and instead return `0` immediately. Hardware interlocks ensure that the current transfer completes before mode transitions and control register updates take effect.

## Memory Map

The memory map for the SPI control registers is shown in Table 13.1. The SPI memory map has been designed to only require naturally aligned 32-bit memory accesses.

## Serial Clock Divisor Register (`sckdiv`)

The `sckdiv` register specifies the divisor used for generating the serial clock (SCK). The relationship between the input clock and SCK is given by the following formula:

$$f_{\mathsf{sck}} = \frac{f_{\mathsf{in}}}{2(\mathsf{div}+1)}$$

The input clock is the bus clock `tlclk`. The reset value of the `div` field is 0x003.

| Address | Name | Description |
|---|---|---|
| `0x000` | sckdiv | Serial clock divisor |
| `0x004` | sckmode | Serial clock mode |
| `0x010` | csid | Chip select ID |
| `0x014` | csdef | Chip select default |
| `0x018` | csmode | Chip select mode |
| `0x028` | delay0 | Delay control 0 |
| `0x02C` | delay1 | Delay control 1 |
| `0x040` | fmt | Frame format |
| `0x048` | txdata | Tx FIFO data |
| `0x04C` | rxdata | Rx FIFO data |
| `0x050` | txmark | Tx FIFO watermark |
| `0x054` | rxmark | Rx FIFO watermark |
| `0x060` | fctrl* | SPI flash interface control |
| `0x064` | ffmt* | SPI flash instruction format |
| `0x070` | ie | SPI interrupt enable |
| `0x074` | ip | SPI interrupt pending |

Table 13.1: Register offsets within the SPI memory map. Registers marked * are present only on controllers with the direct-map flash interface, i.e., SPI0.

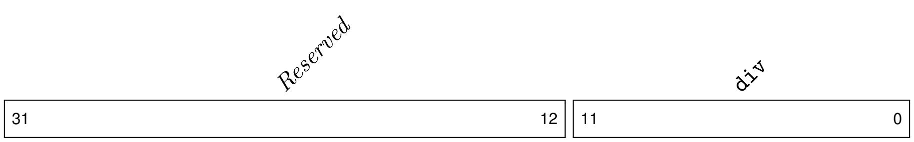


Figure 13.1: Format of `sckdiv` register.

## Serial Clock Mode Register (`sckmode`)

The `sckmode` register defines the serial clock polarity and phase. Tables 13.2 and 13.3 describe the behavior of the `pol` and `pha` fields, respectively. The reset value of `sckmode` is `0`.

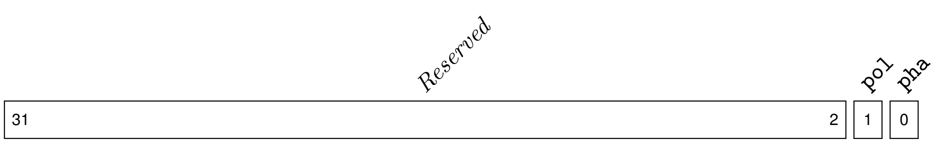


Figure 13.2: Format of `sckmode` register.

| Value | Description |
|---|---|
| 0 | Inactive state of SCK is logical 0 |
| 1 | Inactive state of SCK is logical 1 |

Table 13.2: Serial clock polarity.

| Value | Description |
|---|---|
| 0 | Data is sampled on the leading edge of SCK and shifted on the trailing edge of SCK |
| 1 | Data is shifted on the leading edge of SCK and sampled on the trailing edge of SCK |

Table 13.3: Serial clock phase.

### Chip Select ID Register (`csid`)

The `csid` register encodes the index of the CS pin to be toggled by hardware chip select control. The reset value is `0`.

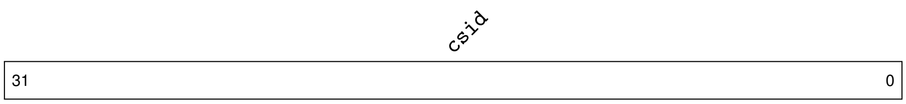


Figure 13.3: Format of `csid` register.

### Chip Select Default Register (`csdef`)

The `csdef` register specifies the inactive state (polarity) of the CS pins. The reset value is `0xFFFF`.

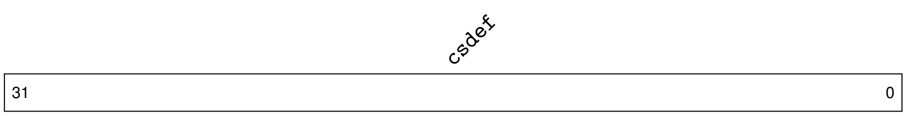


Figure 13.4: Format of `csdef` register.

### Chip Select Mode Register (`csmode`)

The `csmode` register defines the hardware chip select behavior as described in Table 13.4. The reset value is `0` (AUTO). In HOLD mode, the CS pin is de-asserted only when one of the following conditions occur:

- A different value is written to `csmode` or `csid`.
- A write to `csdef` changes the state of the selected pin.
- Direct-mapped flash mode is enabled.


Figure 13.5: Format of `csmode` register.

| Value | Name | Description |
|---|---|---|
| 0 | AUTO | Assert/de-assert CS at the beginning/end of each frame |
| 2 | HOLD | Keep CS continuously asserted after the initial frame |
| 3 | OFF | Disable hardware control of the CS pin |

Table 13.4: Chip select modes.

## Delay Control Registers (`delay0` and `delay1`)

The `delay0` and `delay1` registers allow for the insertion of arbitrary delays specified in units of one SCK period.

The `cssck` field specifies the delay between the assertion of CS and the first leading edge of SCK. When $\texttt{sckmode.pha} = 0$, an additional half-period delay is implicit. The reset value is `0x01`.

The `sckcs` field specifies the delay between the last trailing edge of SCK and the de-assertion of CS. When $\texttt{sckmode.pha} = 1$, an additional half-period delay is implicit. The reset value is `0x01`.

The `intercs` field specifies the minimum CS inactive time between de-assertion and assertion. The reset value is `0x01`.

The `interxfr` field specifies the delay between two consecutive frames without de-asserting CS. This is applicable only when `sckmode` is HOLD or OFF. The reset value is `0x00`.

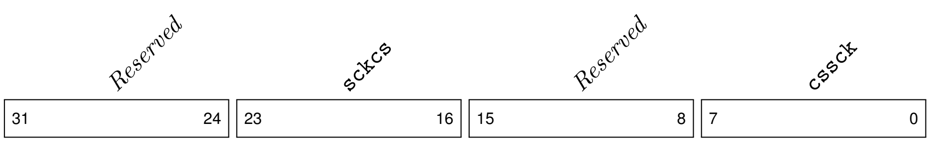


Figure 13.6: Format of `delay0` register.

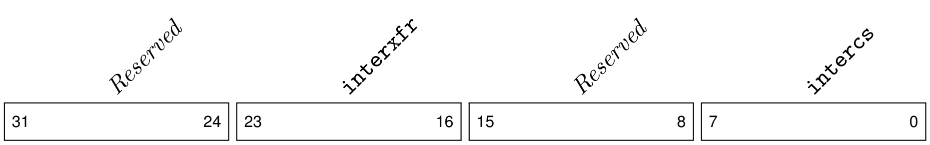


Figure 13.7: Format of `delay1` register.

## Frame Format Register (`fmt`)

The `fmt` register defines the frame format for transfers initiated through the programmed-I/O (FIFO) interface. Tables 13.5, 13.6, and 13.7 describe the `proto`, `endian`, and `dir` fields, respectively. The `len` field defines the number of bits per frame, where the allowed range is 0 to 8 inclusive.

The reset value is 0x80000, corresponding to `proto` = single, `dir` = Rx, `endian` = MSB, and $\texttt{len} = 8$.

 

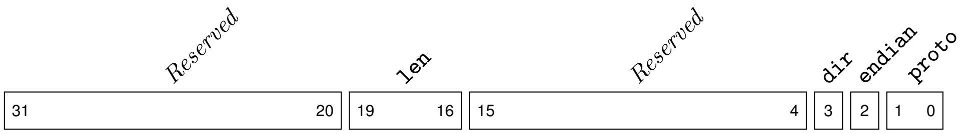


Figure 13.8: Format of `fmt` register.

| Value | Description | Data Pins |
|---|---|---|
| 0 | Single | DQ0 (MOSI), DQ1 (MISO) |
| 1 | Dual | DQ0, DQ1 |
| 2 | Quad | DQ0, DQ1, DQ2, DQ3 |

Table 13.5: SPI protocol. Unused DQ pins are tri-stated.

| Value | Description |
|---|---|
| 0 | Transmit most-significant bit (MSB) first |
| 1 | Transmit least-significant bit (LSB) first |

Table 13.6: SPI endianness.

| Value | Description |
|---|---|
| 0 | Rx: For dual and quad protocols, the DQ pins are tri-stated. For the single protocol, the DQ0 pin is driven with the transmit data as normal. |
| 1 | Tx: The receive FIFO is not populated. |

Table 13.7: SPI I/O direction.

### Transmit Data Register (`txdata`)

Writing to the `txdata` register loads the transmit FIFO with the value contained in the `data` field. For `fmt.len` $< 8$, values should be left-aligned when `fmt.endian` = MSB and right-aligned when `fmt.endian` = LSB.

The `full` flag indicates whether the transmit FIFO is ready to accept new entries; when set, writes to `txdata` are ignored. The `data` field returns `0x00` when read.

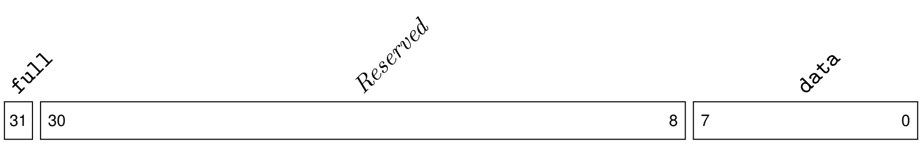


Figure 13.9: Format of `txdata` register.

### Receive Data Register (`rxdata`)

Reading the `rxdata` register dequeues a frame from the receive FIFO. For `fmt.len` $< 8$, values are left-aligned when `fmt.endian` = MSB and right-aligned when `fmt.endian` = LSB.

The `empty` flag indicates whether the receive FIFO contains new entries to be read; when set, the `data` field does not contain a valid frame. Writes to `rxdata` are ignored.

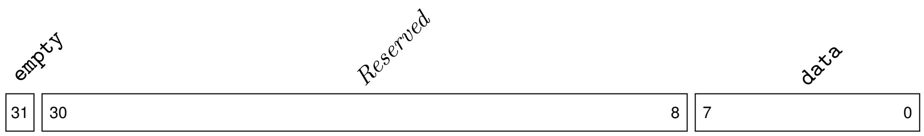


Figure 13.10: Format of `rxdata` register.

### Transmit Watermark Register (`txmark`)

The `txmark` register specifies the threshold at which the Tx FIFO watermark interrupt triggers. The reset value is `0`.


Figure 13.11: Format of `txmark` register.

### Receive Watermark Register (`rxmark`)

The `rxmark` register specifies the threshold at which the Rx FIFO watermark interrupt triggers. The reset value is `0`.

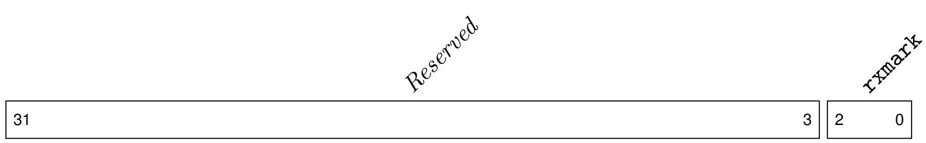


Figure 13.12: Format of `rxmark` register.

### Interrupt Registers (`ie` and `ip`)

The `ie` register controls which SPI interrupts are enabled, and `ip` is a read-only register indicating the pending interrupt conditions. `ie` is reset to zero.

The `txwm` condition becomes raised when the number of entries in the transmit FIFO is strictly less than the count specified by the `txmark` register. The pending bit is cleared when sufficient entries have been enqueued to exceed the watermark.

The `rxwm` condition becomes raised when the number of entries in the receive FIFO is strictly greater than the count specified by the `rxmark` register. The pending bit is cleared when sufficient entries have been dequeued to fall below the watermark.

 

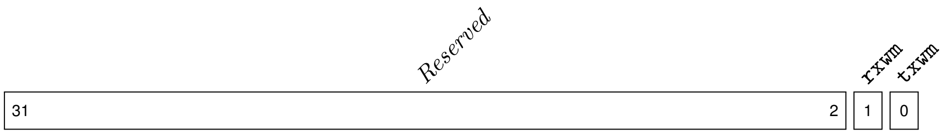


Figure 13.13: Format of `ie` and `ie` registers.

### SPI Flash Interface Control Register (`fctrl`)

When the `en` bit of the `fctrl` register is set, the controller enters SPI flash mode. Accesses to the direct-mapped memory region causes the controller to automatically sequence SPI flash reads in hardware. The reset value is `0x1`.

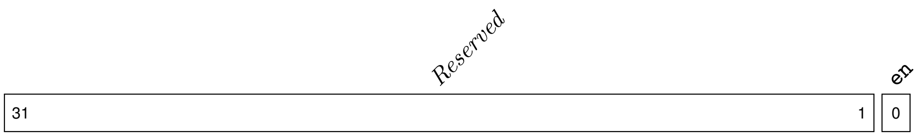


Figure 13.14: Format of `fctrl` register.

### SPI Flash Instruction Format Register (`ffmt`)

The `ffmt` register defines the format of the SPI flash read instruction issued by the controller when the direct-mapped memory region is accessed while in SPI flash mode.

An instruction consists of a command byte followed by a variable number of address bytes, dummy cycles (padding), and data bytes. Table 13.8 describes the function and reset value of each field.

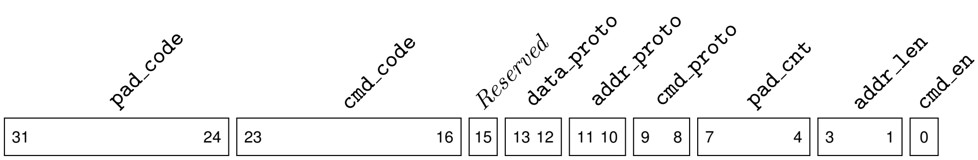


Figure 13.15: Format of `ffmt` register.

| Field | Description | Reset Value |
|---|---|---|
| `cmd_en` | Enable sending of command | `0x1` |
| `cmd_code` | Value of command byte | `0x03` |
| `cmd_proto` | Protocol for transmitting command | `0x0` |
| `addr_len` | Number of address bytes (0 to 4) | `0x3` |
| `addr_proto` | Protocol for transmitting address and padding | `0x0` |
| `pad_cnt` | Number of dummy cycles | `0x0` |
| `pad_code` | First 8 bits to transmit during dummy cycles | `0x00` |
| `data_proto` | Protocol for receiving data bytes | `0x0` |

Table 13.8: Instruction format fields. The protocol values follow the same definition as Table 13.5.

# Chapter 14

# One-Time Programmable Memory (OTP) Peripheral

This chapter describes the operation of the One-Time Programmable Memory (OTP) Controller on SiFive systems.

Device configuration and power-supply control is principally under software control. The controller is reset to a state that allows memory-mapped reads, under the assumption that the controller's clock rate is between 1 MHz and 37 MHz. `vrren` is asserted during synchronous reset; it is safe to read from OTP immediately after reset if reset is asserted for at least ??? $\mu$s while the controller's clock is running.

Programmed-I/O reads and writes are sequenced entirely by software.

## Memory Map

The memory map for the OTP control registers is shown in Table 14.1. The control-register memory map has been designed to only require naturally aligned 32-bit memory accesses. The OTP controller also contains a read sequencer, which exposes the OTP's contents as a read/execute-only memory-mapped device.

## Programmed-I/O lock register (`otp_lock`)

The `otp_lock` register supports synchronization between the read sequencer and the programmed-I/O interface. When the lock is clear, memory-mapped reads may proceed. When the lock is set, memory-mapped reads do not access the OTP device, and instead return `0` immediately.

The `otp_lock` should be acquired before writing to any other control register. Software can attempt to acquire the lock by storing `1` to `otp_lock`. If a memory-mapped read is in progress, the lock will not be acquired, and will retain the value `0`. Software can check if the lock was successfully acquired by loading `otp_lock` and checking that it has the value `1`.

After a programmed-I/O sequence, software should restore the previous value of any control registers that were modified, then store `0` to `otp_lock`.

Figure 14.1 shows the synchronization code sequence.

| Address | Name | Description |
|---|---|---|
| 0x00 | otp_lock | Programmed-I/O lock register |
| 0x04 | otp_ck | OTP device clock signal |
| 0x08 | otp_oe | OTP device output-enable signal |
| 0x0c | otp_sel | OTP device chip-select signal |
| 0x10 | otp_we | OTP device write-enable signal |
| 0x14 | otp_mr | OTP device mode register |
| 0x18 | otp_mrr | OTP read-voltage regulator control |
| 0x1c | otp_mpp | OTP write-voltage charge pump control |
| 0x20 | otp_vrren | OTP read-voltage enable |
| 0x24 | otp_vppen | OTP write-voltage enable |
| 0x28 | otp_a | OTP device address |
| 0x2c | otp_d | OTP device data input |
| 0x30 | otp_q | OTP device data output |
| 0x34 | otp_rsctrl | OTP read sequencer control |

Table 14.1: SiFive OTP Register Offsets. Only naturally aligned 32-bit memory accesses are supported.

```
      la t0, otp_lock
      li t1, 1
loop: sw t1, (t0)
      lw t2, (t0)
      beqz t2, loop
      #
      # Programmed I/O sequence goes here.
      #
      sw x0, (t0)
```

Figure 14.1: Sequence to acquire and release `otp_lock`.

### Programmed-I/O Sequencing

The programmed-I/O interface exposes the OTP device's and power-supply's control signals directly to software. Software is responsible for respecting these signals' setup and hold times.

The OTP device requires that data be programmed one bit at a time and that the result be re-read and retried according to a specific protocol.

See the OTP device and power supply data sheets for timing constraints, control signal descriptions, and the programming algorithm.

### Read sequencer control register (`otp_rsctrl`)

The read sequence consists of an address-setup phase, a read-pulse phase, and a read-access phase. The duration of these phases, in terms of controller clock cycles, is set by a programmable clock divider. The divider is controlled by the `otp_rsctrl` register, the layout of which is shown in Figure 14.2

The number of clock cycles in each phase is given by $2^{scale}$, and the width of each phase may be optionally scaled by 3. That is, the number of controller clock cycles in the address-setup phase is given by the expression $2^{scale}(1 + 2t_{AS})$; the number of clock cycles in the read-pulse phase is given by $2^{scale}(1 + 2t_{RP})$; and the read-access phase is $2^{scale}(1 + 2t_{RACC})$ cycles long. The reset value of `scale` is 1.

Software should acquire the `otp_lock` prior to modifying `otp_rsctrl`.

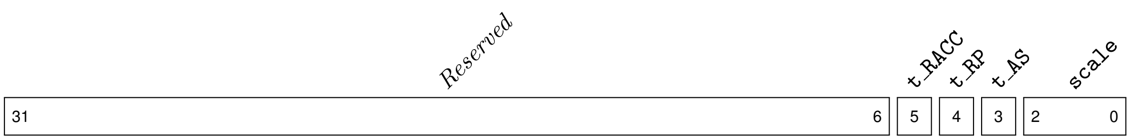


Figure 14.2: Read sequencer control register `otp_rsctrl`

# Chapter 15

# E300 Pulse-Width Modulation (PWM) Peripheral

This chapter describes the operation of the E300 Pulse-Width Modulation peripheral (PWM).

## PWM Overview

Figure 15.1 shows an overview of the PWM peripheral. The default configuration described here has four independent PWM comparators (`pwmcmp0`–`pwmcmp3`), but custom configurations with `ncmp` comparators are available on request. The PWM block can generate multiple types of waveform on GPIO output pins (`pwm`$X$`gpio`), and can also be used to generate several forms of internal timer interrupt. The comparator results are captured in the `pwmcmp`$X$`ip` flops and then fed to the PLIC as potential interrupt sources. The `pwmcmp`$X$`ip` outputs are further processed by an output ganging stage before being fed to the GPIOs.

The PWM unit can be provided in different comparator precisions up to 16 bits, with the version described here having the full 16 bits. To support clock scaling, the `pwmcount` register is 15 bits wider than the comparator precision, `cmpwidth`.

## PWM Memory Map

The memory map for the PWM peripheral is shown in Table 15.1.

## PWM Count Register (`pwmcount`)

The PWM unit is based around a counter held in `pwmcount`. The counter can be read or written over the TileLink bus. The `pwmcount` register is $(15 + cmpwidth)$ bits wide. For example, for `cmpwidth` of 16 bits, the counter is held in `pwmcount[30:0]`, and bit 31 of `pwmcount` returns a zero when read.

When used for PWM generation, the counter is normally incremented at a fixed rate then reset to zero at the end of every PWM cycle. The PWM counter is either reset when the scaled counter `pwms` reaches the value in `pwmcmp0`, or is simply allowed to wraparound to zero.

The counter can also be used in one-shot mode, where it disables counting after the first reset.

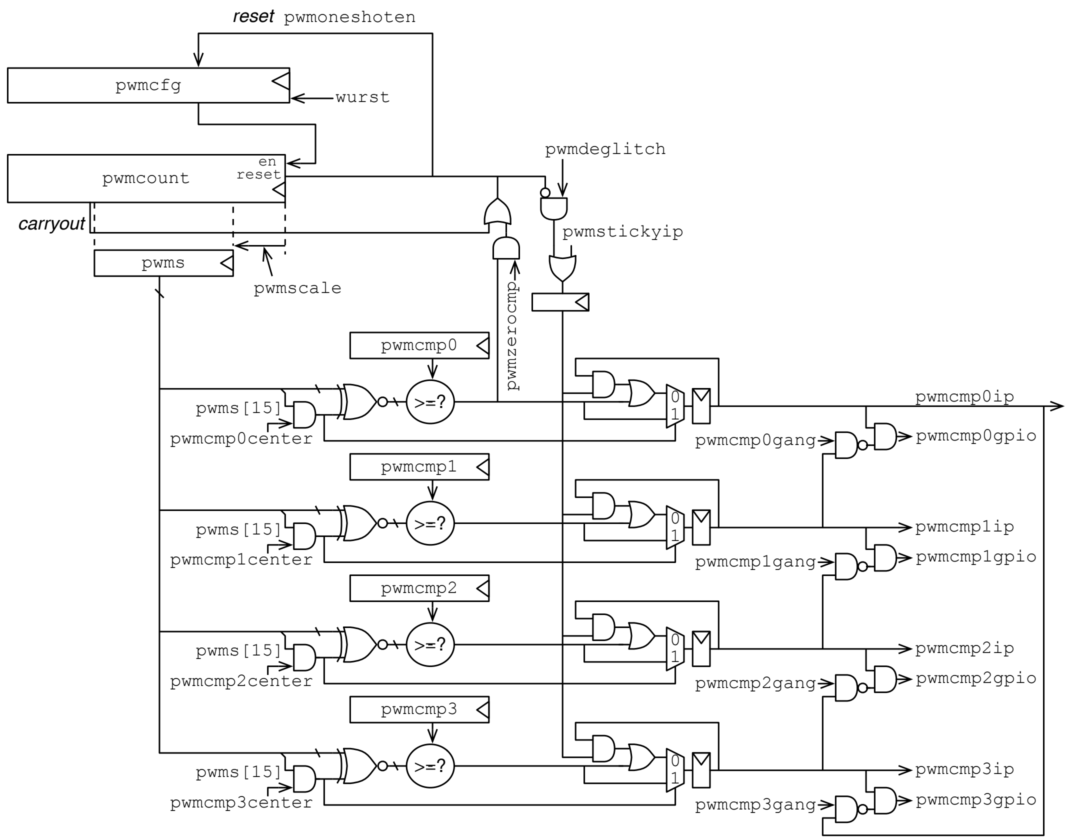


Figure 15.1: E300 PWM Peripheral.

## PWM Configuration Register (`pwmcfg`)

| | |
|---|---|
| 31 | pwmcmp3ip |
| 30 | pwmcmp2ip |
| 29 | pwmcmp1ip |
| 28 | pwmcmp0ip |
| 27 | pwmcmp3gang |
| 26 | pwmcmp2gang |
| 25 | pwmcmp1gang |
| 24 | pwmcmp0gang |
| 23–20 | *Reserved* |
| 19 | pwmcmp3center |
| 18 | pwmcmp2center |
| 17 | pwmcmp1center |
| 16 | pwmcmp0center |
| 15–14 | *Reserved* |
| 13 | pwmenoneshot |
| 12 | pwmenalways |
| 11 | *Reserved* |
| 10 | pwmdeglitch |
| 9 | pwmzerocmp |
| 8 | pwmsticky |
| 7–4 | *Reserved* |
| 3–0 | pwmscale |

Figure 15.2: PWM configuration register `pwmcfg`

The `pwmcfg` register contains various control and status information regarding the PWM peripheral, as shown in Figure 15.2.

The `pwmen*` bits control the conditions under which the PWM counter `pwmcount` is incremented. The counter increments by one each cycle only if any of the enabled conditions are true.

If the `pwmenalways` bit is set, the PWM counter increments continuously. When `pwmenoneshot` is set, the counter can increment but `pwmenoneshot` is reset to zero once the counter resets, disabling further counting (unless `pwmenalways` is set). The `pwmenoneshot` bit provides a way for software to generate a single PWM cycle then stop. Software can set the `pwnenoneshot` again at any time

 

| Address | Description |
|---|---|
| 0x00 | pwmcfg |
| 0x04 | *Reserved* |
| 0x08 | pwmcount |
| 0x0C | *Reserved* |
| 0x10 | pwms |
| 0x14 | *Reserved* |
| 0x18 | *Reserved* |
| 0x1C | *Reserved* |
| 0x20 | pwmcmp0 |
| 0x24 | pwmcmp1 |
| 0x28 | pwmcmp2 |
| 0x2C | pwmcmp3 |

Table 15.1: SiFive PWM memory map, offsets relative to PWM peripheral base address.

to replay the one-shot waveform. The `pwmen*` bits are reset at wakeup reset, which disables the PWM counter and saves power.

The 4-bit `pwmscale` field scales the PWM counter value before feeding it to the PWM comparators. The value in `pwmscale` is the bit position within the `pwmcount` register of the start of a `cmpwidth`-bit `pwms` field. A value of 0 in `pwmscale` indicates no scaling, and `pwms` would then be equal to the low `cmpwidth` bits of `pwmcount`. The maximum value of 15 in `pwmscale` corresponds to dividing the clock rate by $2^{15}$, so for an input bus clock of 16 MHz, the LSB of `pwms` will increment at 488.3 Hz.

The value of `pwms` is memory-mapped and can be read as a single `cmpwidth`-bit value over the TileLink bus.

The `pwmzerocmp` bit, if set, causes the PWM counter `pwmcount` to be automatically reset to zero one cycle after the `pwms` counter value matches the compare value in `pwmcmp0`. This is normally used to set the period of the PWM cycle. This feature can also be used to implement periodic counter interrupts, where the period is independent of interrupt service time.

### PWM Compare Registers (`pwmcmp0`–`pwmcmp3`)

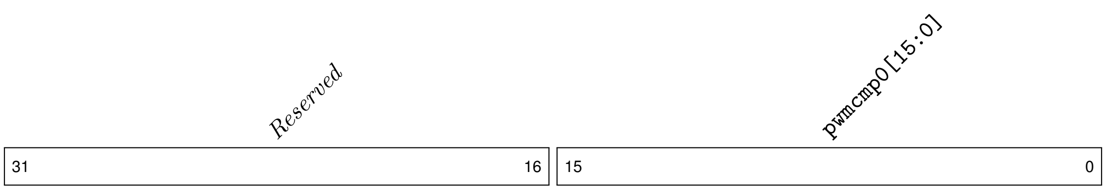


Figure 15.3: PWM compare register `pwmcmp0`. Registers `pwmcmp1`–`pwmcmp3` have the same format. This diagram assumes that `cmpwidth` of 16. The actual width each register is `cmpwidth`.

The primary use of the `ncmp` PWM compare registers is to define the edges of the PWM waveforms within the PWM cycle.

Each compare register is a `cmpwdith`-bit value against which the current `pwms` value is compared every cycle. The output of each comparator is high whenever the value of `pwms` is greater than or equal to the corresponding `pwmcmp`$X$.

If the `pwmzerocomp` bit is set, when `pwms` reaches or exceeds `pwmcmp0`, `pwmcount` is cleared to zero and the current PWM cycle is completed. Otherwise, the counter is allowed to wrap around.

### Deglitch and Sticky circuitry

To avoid glitches in the PWM waveforms when changing `pwmcmp`$X$ register values, the `pwmdeglitch` bit in `pwmcfg` can be set to capture any high output of a PWM comparator in a sticky bit (`pwmcmp`$X$`ip` for comparator $X$) and prevent the output falling again within the same PWM cycle. The `pwmcmp`$X$`ip` bits are only allowed to change at the start of the next PWM cycle.

*Note the `pwmcmp0ip` bit will only be high for one cycle when `pwmdeglitch` and `pwmzerocmp` are set where `pwmcmp0` is used to define the PWM cycle, but can be used as a regular PWM edge otherwise.*

If `pwmdeglitch` is set, but `pwmzerocmp` is clear, the deglitch circuit is still operational but is now triggered when `pwms` contains all 1s and will cause a carry out of the high bit of the `pwms` incrementer just before the counter wraps to zero.

The `pwmsticky` bit will disallow the `pwmcmp`$X$`ip` registers from clearing if they're already set, and is used to ensure interrupts are seen from the `pwmcmp`$X$`ip` bits.

### Generating Left- or Right-Aligned PWM Waveforms

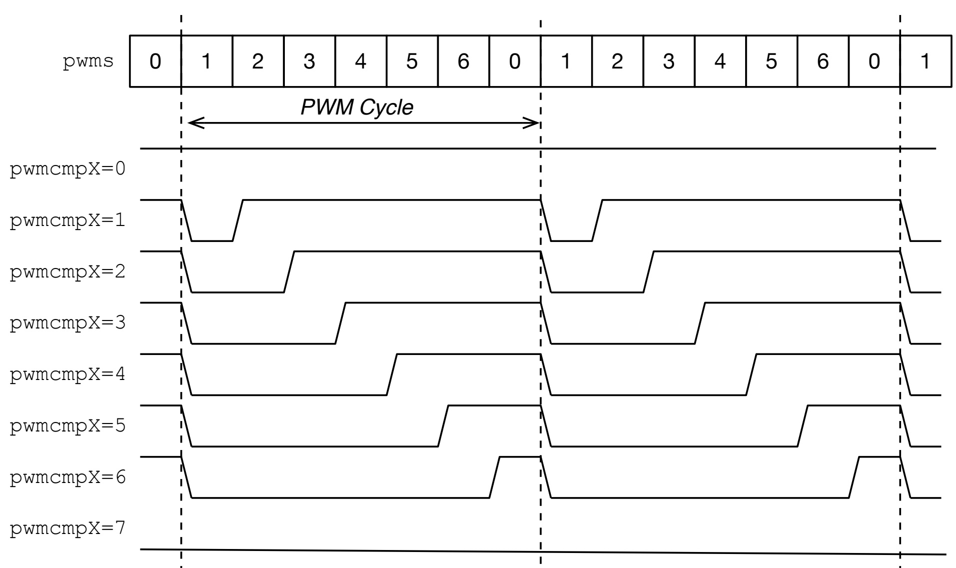


Figure 15.4: E300 basic right-aligned PWM waveforms. All possible base waveforms are shown for a 7-clock PWM cycle (`pwmcmp0`=6). The waveforms show the single cycle delay caused by registering the comparator outputs in the `pwmcmp`$X$`ip` bits. The signals can be inverted at the GPIOs to generate left-aligned waveforms.

Figure 15.4 shows the generation of various base PWM waveforms. The Figure illustrates that if `pwmcmp0` is set to less than the maximum count value (6 in this case), it is possible to generate

| pwms | pwmscenter |
|---|---|
| 000 | 000 |
| 001 | 001 |
| 010 | 010 |
| 011 | 011 |
| 100 | 011 |
| 101 | 010 |
| 110 | 001 |
| 111 | 000 |

Figure 15.5: Illustration of how count value is inverted before presentation to comparator when `pwmcmp`$X$`center` is selected, using a 3-bit `pwms` value.

both 100% (`pwmcmp`$X = 0$) and 0% (`pwmcmp`$X >$ `pwmcmp0`) right-aligned duty cycles using the other comparators. The `pwmcmp`$X$`ip` bits are routed to the GPIO pads, where they can be optionally and individually inverted thereby creating left-aligned PWM waveforms (high at beginning of cycle).

### Generating Center-Aligned (Phase-Correct) PWM Waveforms

The simple PWM waveforms above shift the phase of the waveform along with the duty cycle. A per-comparator `pwmcmp`$X$`center` bit in `pwmcfg` allows a single PWM comparator to generate a center-aligned symmetric duty-cycle as shown in Figure 15.6 The `pwmcmp`$X$`center` bit changes the comparator to compare with the bitwise inverted `pwms` value whenever the MSB of `pwms` is high.

This technique provides symmetric PWM waveforms but only when the PWM cycle is at the largest supported size. At a 16 MHz bus clock rate with 16-bit precision, this limits the fastest PWM cycle to 244 Hz, or 62.5 kHz with 8-bit precision. Higher bus clock rates allow proportionally faster PWM cycles using the single comparator center-aligned waveforms. This technique also reduces the effective width resolution by a factor of 2.

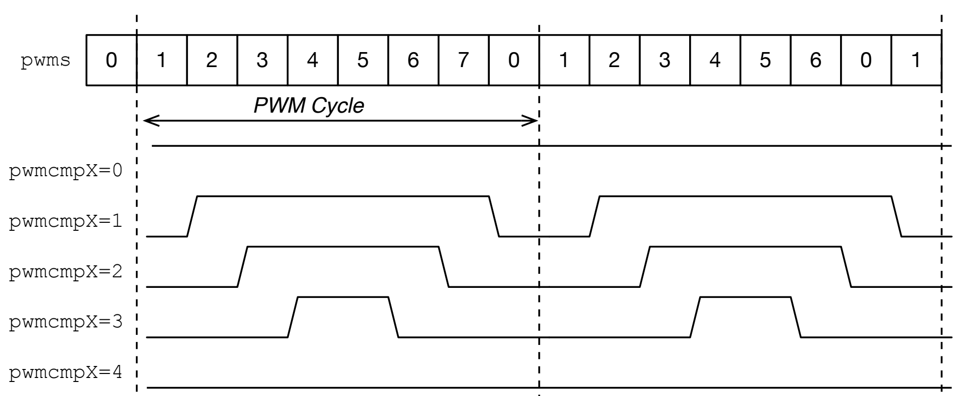


Figure 15.6: E300 center-aligned PWM waveforms generated from one comparator. All possible waveforms are shown for a 3-bit PWM precision. The signals can be inverted at the GPIOs to generate opposite-phase waveforms.

When a comparator is operating in center mode, the deglitch circuit allows one 0-1 transition during the first half of the cycle, and one 1-0 transition on the second half of the cycle.

### Generating Arbitrary PWM Waveforms using Ganging

A comparator can be ganged together with its next-highest-numbered neighbor to generate arbitrary PWM pulses. When the `pwmcmp`$X$`gang` bit is set, comparator $X$ fires and raises its `pwm`$X$`gpio` signal. When comparator $X + 1$ (or `pwmcmp0` for `pwmcmp3`) fires, the `pwm`$X$`gpio` output is reset to zero.

### Generating One-shot Waveforms

The PWM peripheral can be used to generate precisely timed one-shot pulses by first initializing the other parts of `pwmcfg` then writing a 1 to the `pwmenoneshot` bit. The counter will run for one PWM cycle, then once a reset condition occurs, the `pwmenoneshot` bit is reset in hardware to prevent a second cycle.

### PWM Interrupts

The PWM can be configured to provide periodic counter interrupts by enabling auto-zeroing of the count register when a comparator 0 fires (`pwmzerocmp`=1). The `pwmsticky` bit should also be set to ensure interrupts are not forgotten while waiting to run a handler.

The interrupt pending bits `pwmcmp`$X$`ip` can be cleared down using writes to the `pwmcfg` register.

The PWM peripheral can also be used as a regular timer with no counter reset (`pwmzerocmp`=0), where the comparators are now used to provide timer interrupts.