Comparative Analysis for Power Computation and Simulation Time Between RTL and TLM

By

Ahmed Mohamed Mahmoud Abdulhamid Khalil

Ahmed Mohamed Mahmoud Aly

Ahmed Nagy Mohamed Mohamed El-Zeiny

Adham Hassan Rageh

AbdelRahman Saeed Anwar

Karim Mohamed Abo El-Makarem

Under supervision of

Dr, Hassan Mostafa

Dr, Magdy El-Moursy

A Graduation Project Report Submitted to
the Faculty of Engineering at Cairo University
in Partial Fulfillment of the Requirements for the
Degree of
Bachelor of Science
in
Electronics and Communications Engineering

Faculty of Engineering, Cairo University

Giza, Egypt

July 2017

Table of Contents

Chapter 1: Contents

List of Tables v

List of Figures vi

List of Symbols and Abbreviations viii

Acknowledgments ix

Abstract x

Chapter 2: Introduction 1

2.1 Problem Overview 2

2.2 Solution 3

2.3 Work Flow 4

Chapter 3: Zynq-7000 All Programmable SoC 5

3.1 Introduction 5

3.2 Block Diagram 6

3.3 GPIO 7

3.3.1 Features 7

3.3.2 Block Diagram 7

3.3.3 Functional Description 9

3.3.4 EMIO Signals 10

3.3.5 Interrupts 11

3.4 GIC 12

3.4.1 Block Diagram 13

3.4.2 Functional Description 14

3.4.3 Interrupt Sensitivity and Targeting 15

3.4.4 Interrupt Prioritization and Handling 16
3.5 UART 17
3.5.1 Features 17
3.5.2 Block Diagram 18
3.5.3 Functional Description 19
3.5.4 Status and Interrupts 25
3.5.5 Block Flow Building 27
3.6 SPI 32
3.6.1 Features 32
3.6.2 Block Diagram 33
3.6.3 Functional Description 35
3.7 I2C 40
3.7.1 Features 41
3.7.2 Block Diagram 42
3.7.3 Functional Description 44
3.7.4 Modes of Operation 47
3.8 DMA 52
3.8.1 Features 55
3.8.2 Block Diagram 56
3.8.3 Functional Description 58
3.8.4 Events and Timing Diagram 65
Chapter 4: Simulation, Synthesis, and Integration 72
4.1 Simulation 72
4.1.1 GPIO 72
4.1.2 GIC 75
4.1.3 UART 76
4.1.4 SPI 77

4.1.5 I2C 78

4.2 Synthesis 78

4.2.1 Logic Synthesis 79

4.2.2 Synthesis Problems 81

4.3 Integration 82

4.3.1 Zynq Blocks Integration 83

4.3.2 Scenarios 84

4.3.3 Integration Problems 85

Chapter 5: Transaction Level Power Modeling 87

5.1 Power Modeling for TLM 87

5.2 TLPM Methodology 88

5.3 Correlation Matrix 90

5.4 Vista 92

Chapter 6: Results and Conclusion 95

6.1 Results of each block 95

6.2 Overall System results 97

6.3 Conclusion 99

References 100

List of Tables

Table 1 GPIO Interrupt signals 12
Table 2 PPI Interrupt Requests IDs 14
Table 3 SPI Interrupt Requests IDs 15
Table 4 Auto/Manual SS and Start 36
Table 5 Clock Phase and Polarity Parameters 39
Table 6 Operations of the DMA state machine 63
Table 7 UART Scenarios 76
Table 8 GPIO results 95
Table 9 SPI results 95
Table 10 I2C results 96
Table 11 Power Calculation results of overall system 99
Table 12 Simulation Time results of overall system 99

List of Figures

Figure 2-1. Flow of the extraction of power parameters ... 3
Figure 3-1. Block Diagram of Zynq-7000 ... 6
Figure 3-2. GPIO main banks ... 8
Figure 3-3. GPIO Functional Block Diagram ... 8
Figure 3-4 GIC Block Diagram ... 13
Figure 3-5 UART System block ... 19
Figure 3-6 UART Block Diagram ... 19
Figure 3-7 Baud Rate Generator ... 20
Figure 3-8 Default BDIV Receiver Data Stream ... 22
Figure 3-9 UART Mode Switch for TxD and RxD ... 24
Figure 3-10 Interrupts and status signals ... 25
Figure 3-11 UART RxFIFO and TxFIFO Interrupt ... 27
Figure 3-12 TX state diagram ... 29
Figure 3-13 RX state diagram ... 29
Figure 3-14 Controllers, FIFOs and Baud Rate generator ... 31
Figure 3-15 SPI system block diagram ... 33
Figure 3-16 SPI functional block diagram ... 34
Figure 3-17 Rx and Tx FIFO interrupts ... 39
Figure 3-18 The clock phase parameter and the state of the SS ... 40
Figure 3-19 I2C Block Diagram ... 42
Figure 3-20 I2C Device Connections ... 44
Figure 3-21 I2C Transaction waveform ... 46
Figure 3-22 I2C Clock Division ... 47
Figure 3-23 Programmed I/O to UART ... 53
Figure 3-24 DMA Transfer to UART ... 54
Figure 3-25 DMA Interface ... 56
Figure 3-26 DMA Block Diagram ... 57
Figure 3-27 Round Robin Scheduling ... 59
Figure 3-28 DMA APB Interface ... 59
Figure 3-29 DMA AXI Interface ... 60

Figure 3-30 Peripheral request interfaces 61
Figure 3-31 Operating state in DMAC 62
Figure 3-32 DMA Write Transfer 63
Figure 3-33 DMA Read Transfer 64
Figure 3-34 Single and Burst request timing diagram 65
Figure 3-35 Burst request timing diagram 66
Figure 3-36 DMA Abort handling 68
Figure 3-37 Even Divider Timing Diagram 69
Figure 3-38 Even Diagram Flow Chart 69
Figure 3-39 Odd Divider Timing Diagram 70
Figure 3-40 Optimized Design Flow Chart 71
Figure 4-1 GPIO Simulation results 1 72
Figure 4-2 GPIO Simulation results 2 73
Figure 4-3 GPIO Simulation results 3 74
Figure 4-4 GPIO Simulation results 4 74
Figure 4-5 GIC Simulation Results 75
Figure 4-6 FIFO containing the 16 bytes to be sent 76
Figure 4-7 interrupt is raised and 0the data is totally sent 76
Figure 4-8 SPI Simulation Results 77
Figure 4-9 SPI Memory results 78
Figure 4-10 I2C Simulation results 78
Figure 4-11 Synthesis diagram 79
Figure 4-12 Logic Synthesis steps 79
Figure 4-13 Static Timing Analysis 80
Figure 4-14 Negative slack 81
Figure 4-15 Synthesis Problem 1 81
Figure 4-16 Integration 82
Figure 4-17 Zynq Blocks Integration 83
Figure 4-18 GPIO Scenarios 84
Figure 4-19 I2C, SPI, UART Scenarios 84
Figure 4-20 Integration Problem 1 85
Figure 4-21 Interrupts implementation Problem 86
Figure 5-1 Power Characterization flow 87
Figure 5-2 Power Characterization in TLPM 89

List of Symbols and Abbreviations

RTL	Register Transfer Level
TLM	Transaction Level Modeling
DMA	Direct Memory Access
GPIO	General Purpose Input/ Output
GIC	Generic Interrupt Controller
I2C	Inter-Integrated Circuit
SPI	Serial Peripheral Interface
UART	Universal Asynchronous Receiver Transmitter
FIFO	First In First Out
Rx	Receiver
Tx	Transmitter
PPI	Private Peripheral Interrupt
SPI	Shared Peripheral Interrupt
SGI	Software Generated Interrupt
APB	Advanced Peripheral Bus

Acknowledgments

We have the opportunity to present our appreciation to all of those who helped and supported us along the year work.

First, to our advisors, Dr. Hassan Mostafa and Dr. Magdy El-Morsy, we would like to express our sincere gratefulness and appreciation for their excellent guidance, caring, patience, and immense help in planning and executing the work in a timely manner. Their great personality and creativity provided us with an excellent atmosphere for work, while their technical insight and experience helped us a lot in our research. Their support at the time of crisis will always be remembered.

Of course, we will never find words enough to express the gratitude and appreciation that we owe to our families. Their tender love and support have always been the cementing force for building what we achieved. The all-round support rendered by them provided the much needed stimulant to sail through the phases of stress and strain.

We would like also to thank Eng. Amr Baher, who was always there for any research questions, providing precious advice and sharing his time and experience.

We would like to thank Eng. Zyad Abd El Tawab and Eng. Amr Hany for their assistance and contribution in the tools support.

Finally, Thanks are due to previous GP members. Their theses were supportive, informative and guided us in many critical issues.

Abstract

Designing a system on chip (SoC) is one of the main challenges nowadays in the market. A wide introducing of the TLM in implementing SoCs is proposed to prove its uniqueness in getting faster validation for the designers. This paper introduces a methodology that using TLM to accelerate the simulation time than RTL with accurate power estimation. Some techniques (RTL simulation, Design synthesis and SystemC prototyping) are used to achieve the methodology purpose. ZYNQ-7000 FPGA blocks (GPIO, SPI, I2C and UART) are implemented on RTL and TLM to validate the methodology.

The validation of the functionality is obtained through similar scenarios on both TLM and RTL. Experimental results reveal the efficiency and accuracy of the methodology. This methodology has appeared to speed up the simulation time up to 636 times than RTL while the error in power estimation is on average 1.2%.

In our study, we show that by raising the level of abstraction, one not only achieves better simulation speed, flexibility, ease of verification but also reduces time to develop and shorten code length. All this is achieved while being able to maintain almost the same accuracy.

Evaluating power consumption at early phase of product life cycle is important to decrease the number of the expensive design iterations. A methodology is proposed in our study for dynamic power estimation using Transaction Level Modelling (TLM). The methodology exploits the existing tools for RTL simulation, design synthesis and SystemC prototyping to provide fast and accurate power estimation using Transaction Level Power Modelling (TLPM).

Chapter 2: Introduction

SoCs are widely used nowadays in many applications like mobiles, cameras, FPGAs,...etc. Trying to implement SoC on RTL reveals some restrictions. The main restriction is the simulation time. As the SoC complexity increases, the simulation time increases or simulation crashes. As a result, another restriction appeared which is power estimation.

Trying to get an estimation of the dynamic power on RTL has become an impossible target due to the large simulation time. To overcome the drawbacks of RTL, different methodologies and modeling techniques have been implemented to keep the design life cycle short. The trials show that implementing SoCs using TLM has a great impact on accelerating design life cycle and introducing it to the market in a short time.

Whenever the estimation of the power was in early stages, the release of the design will be accelerated. This will save the expensive costs paid on the huge computations of the long iterations which elongate the design life cycle. SoC design brings with it new challenges and difficulties.

The designs are large, complicated, and involve software and hardware components. These designs have to be modeled at a high level of abstraction before partitioning into hardware and software components for final implementation. From the hardware (HW) design point of view, hardware description languages (HDLs) such as Verilog and VHDL, in conjunction with hardware simulation and synthesis tools, have proven highly valuable. Efforts have been made to extend these approaches to work for multiprocessor-based SoC design, but these tend to require a significant amount of information about the partitioning of the system into coarse-grained blocks at the start. With the increased complexity new approaches seem inevitable.

We are at a transition stage similar to the move from schematic-based design to hardware description language (HDL)-based design. A few years ago, just the notion of a new language (such as C) for hardware design, raised technical and suitability issues. Hardware engineers accustomed to using HDLs find it difficult to accept that a

software development and modeling language such as C/C++ could be useful for hardware design. To accelerate hardware design, designers use software models of the hardware that they build. These models developed at high level of abstraction are used to validate the functionality and evaluate performance.

SystemC a relatively new language, is aimed at facilitating model development above the register transfer level (RTL) [1,2]. SystemC 2.0 provides the ability to capture designs at various levels of abstraction. The single language solution to express designs at different levels of abstraction makes SystemC language a strong contender for system level design language. Built around the C++ language, SystemC inherits the reputation of C++ as a multiparadigm language, and offers additional capabilities for HW design. SystemC is close enough to C++ to develop SW intellectual property (IP) blocks, and is well suited for developing HW soft IP's.

Earlier to SystemC, a typical design at system level was created using C/C++, Matlab, Spreadsheet or some variant. Next, these designs had to be manually translated into RTL to capture the architecture. This conversion would typically lead to many design errors making previous work unfit for reuse.

SystemC provides various features to perform system level modeling and simulation, which are missing in the generic HDL's such as VHDL and Verilog. But SystemC is not intended to replace Verilog or VHDL. Supporting models at various levels of abstraction is a key feature in SystemC. Synthesizability of the SystemC models currently lacks commercial tool support.

2.1 Problem Overview

Determining the power of large System-on-Chip (SoC) designs is very computational and time expensive on Register Transfer Level (RTL), if even possible. Power should be addressed in early stages of the design process to prevent long expensive iterations, which hinders releasing a design. Integration of power computation with TLM is gaining interest in both research and industry. TLM for power is needed to overcome any unexpected drawback in the design at early stages. In this paper, estimating power dissipation on TLM is performed in two stages: Characterization and Implementation. Characterization represents the extraction of power parameters from the existing design. Implementation represents the addition of power model to TLM and the

execution of this model to have reliable numbers of power consumption. The flow is introduced in Figure 2-1.

2.2 Solution

Several approaches have been developed to estimate the power using different methodologies at various abstraction levels of the design. The presented approach misses important design elements such as power management components, which affects accuracy for the resulting power numbers versus actual on-chip ones. The flow in Power Kernel tool adopts cycle-accurate models in TLM. This leads to large simulation time.

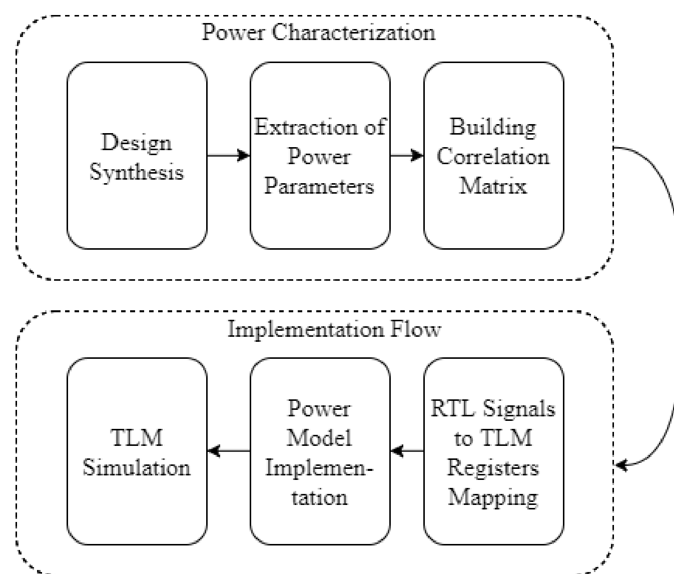


Figure 2-1. Flow of the extraction of power parameters

Transaction Level Power Modeling (TLPM) Flow Characterizing power parameters has been addressed in previous works using extracted Gate level parasitics as introduced in [3,5]. The parasitics are used to build database in order to calculate the power equation. This process of parasitics extraction and Gate level simulation is complicated and time consuming. Simulation overhead is very high in [8,9]. Extending every SystemC module with add-on library leads to undesirable additional simulation time for power instrumentation.

Previous approaches are not suitable for large designs of SoC. They are either too slow or too inaccurate. In this thesis, approximately-timed modeling at high level of abstraction of the design components is adopted to achieve fast simulation with high

precision. A methodology for estimating power dissipation on TLM (TLPM) is developed. The simulation overhead of power instrumentation is minimized.

2.3 Work Flow

In this document, we discuss the effect of modeling at different levels of abstractions using SystemC and Verilog. We start with an introduction to Zynq-7000 SoC and the implemented blocks (GPIO, GIC, SPI, I2C, and UART) in Chapter 3. Then, we discuss the Simulation of each block along with Synthesis and Integration in Chapter 4.

In Chapter 5, we introduce the Transaction Level Power Modeling methodology and power characterization flow. Finally, in Chapter 6, we discuss the results obtained and provide conclusions.

Chapter 3: Zynq-7000 All Programmable SoC

3.1 Introduction

The System-on-Chip (SoC) we are basing our comparative analysis on is the Zynq-7000 family platform. The Zynq®-7000 family is based on the Xilinx® All Programmable SoC (AP SoC) architecture. These products integrate a feature-rich dual-core ARM® Cortex™-A9 MPCore™ based processing system (PS) and Xilinx programmable logic (PL) in a single device, built on a state-of-the-art, high-performance, low-power (HPL), 28 nm, and high-k metal gate (HKMG) process technology. The ARM Cortex-A9 MPCore CPUs are the heart of the PS which also includes on-chip memory, external memory interfaces, and a rich set of I/O peripherals.

The Zynq-7000 family offers the flexibility and scalability of an FPGA, while providing performance, power, and ease of use typically associated with ASIC and ASSPs. The range of devices in the Zynq-7000 AP SoC family enables designers to target cost-sensitive as well as high-performance applications from a single platform using industry-standard tools. While each device in the Zynq-7000 family contains the same PS, the PL and I/O resources vary between the devices. As a result, the Zynq-7000 AP SoC devices can serve a wide range of applications including:

- Automotive driver assistance, driver information, and infotainment
- Broadcast camera
- Industrial motor control, industrial networking, and machine vision
- IP and smart camera
- LTE radio and baseband
- Medical diagnostics and imaging
- Multifunction printers
- Video and night vision equipment

The Zynq-7000 architecture conveniently maps the custom logic and software in the PL and PS respectively. It enables the realization of unique and differentiated system

functions. The integration of the PS with the PL provides levels of performance that two-chip solutions (for example, an ASSP with an FPGA) cannot match due to their limited I/O bandwidth, loose-coupling and power budgets.

3.2 Block Diagram

Figure 3-1 illustrates the functional blocks of the Zynq-7000 AP SoC.

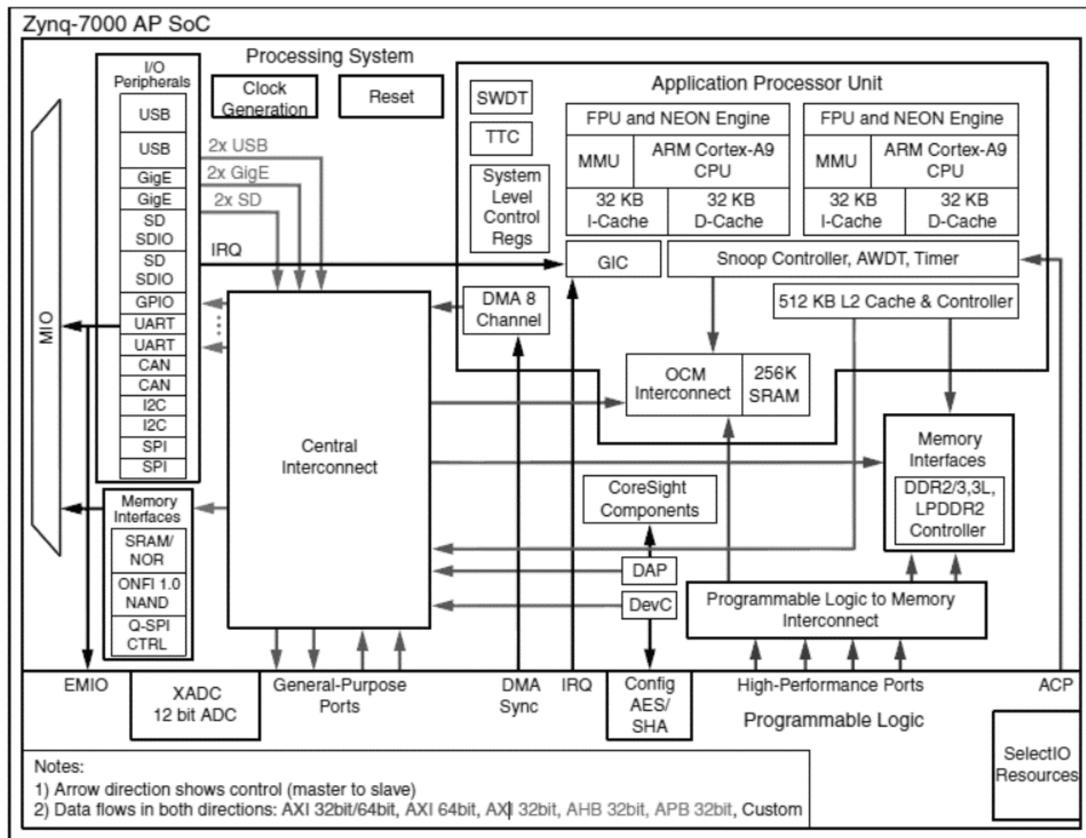


Figure 3-1. Block Diagram of Zynq-7000

It is obvious that the System is too huge and could never be completely built within neither the available time nor the allowed team number. Hence, only six blocks were chosen to build our analysis on. These blocks are GPIO, SPI, UART, GIC, I2C, and DMAC. In the next sections in this chapter, we will talk about each one of these blocks in details.

3.3 GPIO

The general purpose I/O (GPIO) peripheral provides software with observation and control of up to 54 device pins via the MIO module. It also provides access to 64 inputs from the Programmable Logic (PL) and 128 outputs to the PL through the EMIO interface. The GPIO is organized into four banks of registers that group related interface signals. Each GPIO is independently and dynamically programmed as input, output, or interrupt sensing.

Software can read all GPIO values within a bank using a single load instruction, or write data to one or more GPIOs (within a range of GPIOs) using a single store instruction. The GPIO control and status registers are memory mapped at base address 0xE000_A000.

3.3.1 Features

Key features of the GPIO peripheral are summarized as follows:

- 54 GPIO signals for device pins (routed through the MIO multiplexer).
- Outputs are 3-state capable.
- 192 GPIO signals between the PS and PL via the EMIO interface.
- 64 Inputs, 128 outputs (64 true outputs and 64 output enables).
- The function of each GPIO can be dynamically programmed on an individual or group basis.
- Enable, bit or bank data write, output enable and direction controls.
- Programmable interrupts on individual GPIO basis.
- Status read of raw and masked interrupt.
- Selectable sensitivity: Level-sensitive (High or Low) or edge-sensitive (positive, negative, or both).

3.3.2 Block Diagram

As shown in Figure 3-2, the GPIO module is divided into four banks:

1. Bank0: 32-bit bank controlling MIO pins [31:0].
2. Bank1: 22-bit bank controlling MIO pins [53:32].
3. Bank2: 32-bit bank controlling EMIO signals [31:0].
4. Bank3: 32-bit bank controlling EMIO signals [63:32].

Note: Bank1 is limited to 22 bits because the MIO has a total of 54 pins.

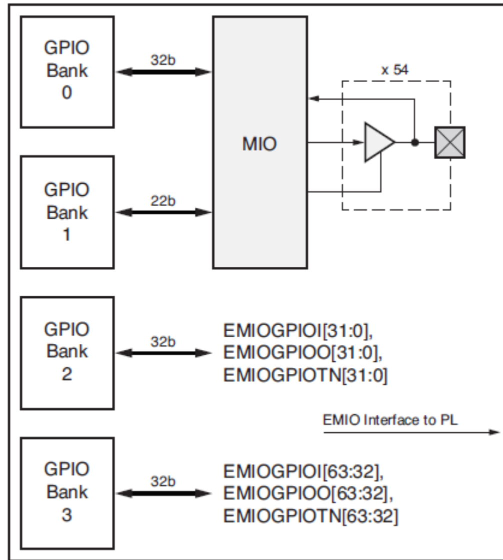


Figure 3-2. GPIO main banks

Each bank consists of several registers and other functional hardware as shown in Figure 3-3.

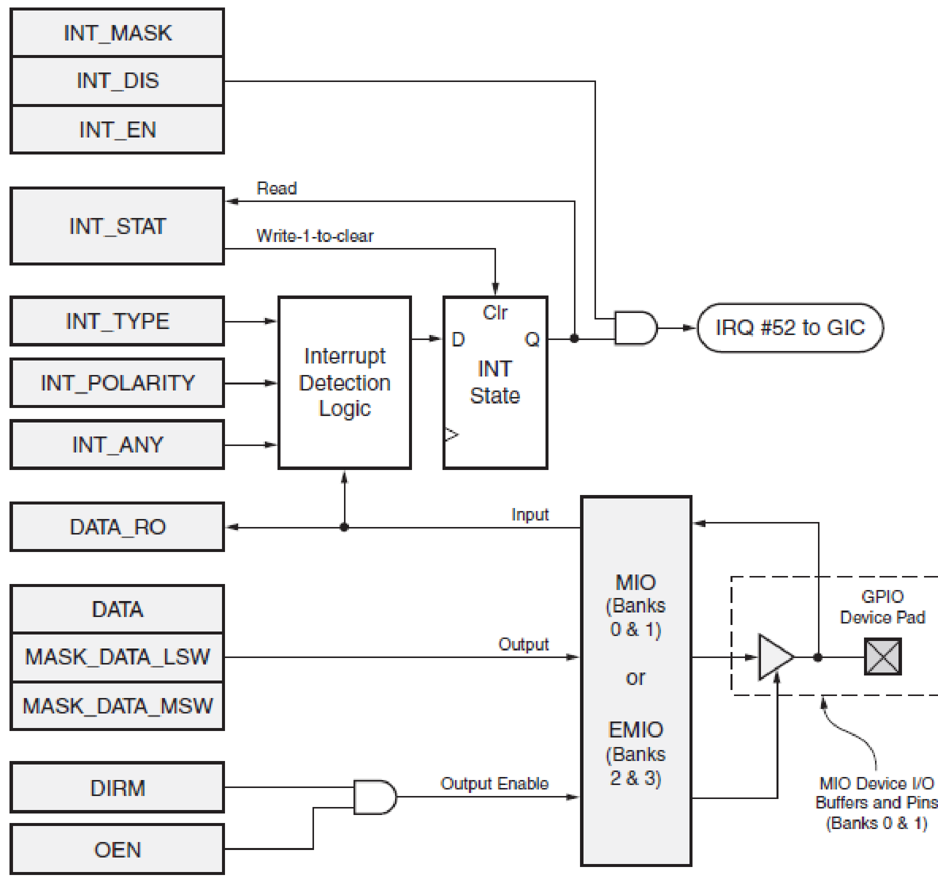


Figure 3-3. GPIO Functional Block Diagram

3.3.3 Functional Description

The GPIO is controlled by software through a series of memory-mapped registers. The control for each bank is the same, although there are minor differences between the MIO and EMIO banks due to their differing functionality.

Software configures the GPIO as either an output or input. The DATA_RO register always returns the state of the GPIO pin regardless of whether the GPIO is set to input (OE signal false) or output (OE signal true). To generate an output waveform, software repeatedly writes to one or more GPIOs (usually using the MASK_DATA register).

Applications might need to switch more than one GPIO at the same time (less a small amount of inherent skew time between two I/O buffers). In this case, all the GPIOs that need to be switched simultaneously must be from the same 16-bit half-bank (i.e., either the most-significant 16 bits or the least-significant 16 bits) of GPIOs to enable the MASK_DATA register to write to them in one store instruction.

GPIO bank control (for Bank0 and Bank1) is summarized as follows:

- **DATA_RO:** This register enables software to observe the value on the device pin. If the GPIO signal is configured as an output, then this would normally reflect the value being driven on the output. Writes to this register are ignored. If the MIO is not configured to enable this pin as a GPIO pin, then DATA_RO is unpredictable because software cannot observe values on non-GPIO pins through the GPIO registers.
- **DATA:** This register controls the value to be output when the GPIO signal is configured as an output. All 32 bits of this register are written at one time. Reading from this register returns the previous value written to either DATA or MASK_DATA {LSW, MSW}; it does not return the current value on the device pin.
- **MASK_DATA_LSW:** This register enables more selective changes to the desired output value. Any combination of up to 16 bits can be written. Those bits that are not written are unchanged and hold their previous value. Reading from this register returns the previous value written to either DATA or MASK_DATA {LSW, MSW}; it does not return the current value on the device pin. This register avoids the need for a read-modify-write sequence for unchanged bits.

- **MASK_DATA_MSW:** This register is the same as MASK_DATA_LSW, except it controls the upper16 bits of the bank.
- **DIRM:** Direction Mode. This controls whether the I/O pin is acting as an input or an output. Since the input logic is always enabled, this effectively enables/disables the output driver. When DIRM[x]==0, the output driver is disabled.
- **OEN:** Output Enable. When the I/O is configured as an output, this controls whether the output is enabled or not. When the output is disabled, the pin is 3-stated. When OEN[x]==0, the output driver is disabled. If MIO TRI_ENABLE is set to 1, enabling 3-state and disabling the driver, then OEN is ignored and the output is 3-stated.

3.3.4 EMIO Signals

This section describes the operation of Bank2 and Bank3. The register interface for the EMIO banks is the same as for the MIO banks described in the previous section. However, the EMIO interface is simply wires between the PS and the PL, so there are a few differences:

- The inputs are wires from the PL and are unrelated to the output values or the OEN register. They can be read from the DATA_RO register when DIRM is set to 0, making it an input.
- The output wires are not 3-state capable, so they are unaffected by OEN. The value to be output is programmed using the DATA, MASK_DATA_LSW, and MASK_DATA_MSW registers. DIRM *must* be set to 1, making it an output.
- The output enable wires are simply outputs from the PS. These are controlled by the DIRM/OEN registers as follows: EMIOGPIOTN[x] = DIRM[x] & OEN[x].
- The EMIO I/Os are not connected to the MIO I/Os in any way. The EMIO inputs cannot be connected to the MIO outputs and the MIO inputs cannot be connected to the EMIO outputs. Each bank is independent and can only be used as software observable/controllable signals.

GPIO bits [8:7] of Bank0 correspond to package pins that are used to control the voltage mode of the I/O buffers themselves during reset. These pins are called the VMODE pin straps for the MIO. They must be driven by the external system

according to the proper voltage mode. To prevent them from being driven by other system logic, they cannot be used as general-purpose inputs.

These bits can be used as general-purpose outputs since the output driver is disabled at reset. The system can start using these as outputs after the voltage mode has been read during system boot.

3.3.5 Interrupts

The interrupt detection logic monitors the GPIO input signal. The interrupt trigger can be a positive edge, negative edge, either edge, Low-level or High-level. The trigger sensitivity is programmed using the INT_TYPE, INT_POLARITY and INT_ANY registers as shown in Table 1.

If an interrupt is detected, the GPIO's INT_STAT state is set true by the interrupt detection logic. If the INT_STAT state is enabled (unmasked), then the interrupt propagates through to a large OR function. This function combines all interrupts for all GPIOs in all four banks to one output (IRQ ID#52) to the interrupt controller. If the interrupt is disabled (masked), then the INT_STAT state is maintained until cleared, but it does not propagate to the interrupt controller unless the INT_EN is later written to disable the mask. As all GPIOs share the same interrupt, software must consider both INT_MASK and INT_STAT to determine which GPIO is causing an interrupt.

The interrupt mask state is controlled by writing a 1 to the INT_EN and INT_DIS registers. Writing a 1 to the INT_EN register disables the mask allowing an active interrupt to propagate to the interrupt controller. Writing a 1 to the INT_DIS register enables the mask. The state of the interrupt mask can be read using the INT_MASK register.

If the GPIO interrupt is edge sensitive, then the INT state is latched by the detection logic. The INT latch is cleared by writing a 1 to the INT_STAT register. For level-sensitive interrupts, the source of the interrupt input to the GPIO must be cleared to clear the interrupt signal. Alternatively, software can mask that input using the INT_DIS register.

The state of the interrupt signal going to the interrupt controller can be inferred by reading the INT_STAT and INT_MASK registers. This interrupt signal is asserted if INT_STAT=1 and INT_MASK=0. GPIO bank control is summarized as follows:

- **INT_MASK:** This register is read-only and shows which bits are currently masked and which are un-masked/enabled.
- **INT_EN:** Writing a 1 to any bit of this register enables/unmasks that signal for interrupts. Reading from this register returns an unpredictable value.
- **INT_DIS:** Writing a 1 to any bit of this register masks that signal for interrupts. Reading from this register returns an unpredictable value.
- **INT_STAT:** This register shows if an interrupt event has occurred or not. Writing a 1 to a bit in this register clears the interrupt status for that bit. Writing a 0 to a bit in this register is ignored.
- **INT_TYPE:** This register controls whether the interrupt is edge sensitive or level sensitive.
- **INT_POLARITY:** This register controls whether the interrupt is active-Low or active High (or falling-edge sensitive or rising-edge sensitive).
- **INT_ON_ANY:** If INT_TYPE is set to edge sensitive, then this register enables an interrupt event on both rising and falling edges. This register is ignored if INT_TYPE is set to level sensitive.

Type	gpio.INT_TYPE_0	gpio.INT_POLARITY_0	gpio.INT_ANY_0
Rising edge-sensitive	1	1	0
Falling edge-sensitive	1	0	0
Both rising and falling edge-sensitive	1	X	1
Level sensitive, asserted High	0	1	X
Level sensitive, asserted Low	0	0	X

Table 1 GPIO Interrupt signals

3.4 GIC

The generic interrupt controller (GIC) is a centralized resource for managing interrupts sent to the CPUs from the Processing System (PS) and the Programmable Logic (PL). The controller enables, disables, masks, and prioritizes the interrupt sources and sends them to the selected CPU (or CPUs) in a programmed manner as the CPU interface accepts the next interrupt.

The controller is based on the ARM Generic Interrupt Controller Architecture version 1.0 (GIC v1), non-vectored. The registers are accessed via the CPU private bus for fast read/write response by avoiding temporary blockage or other bottlenecks in the interconnect.

The interrupt distributor centralizes all interrupt sources before dispatching the one with the highest priority to the individual CPUs. The GIC ensures that an interrupt targeted to several CPUs can only be taken by one CPU at a time. All interrupt sources are identified by a unique interrupt ID number. All interrupt sources have their own configurable priority and list of targeted CPUs.

3.4.1 Block Diagram

There are three main sources of interrupts controlled by the GIC; Software Generated Interrupts (SGI), CPU Private Peripheral Interrupts (PPI), and Shared Peripheral Interrupts (SPI). The shared peripheral interrupts are generated from various subsystems that include the I/O peripherals in the PS and logic in the PL. The interrupt sources are illustrated in Figure 3-4.

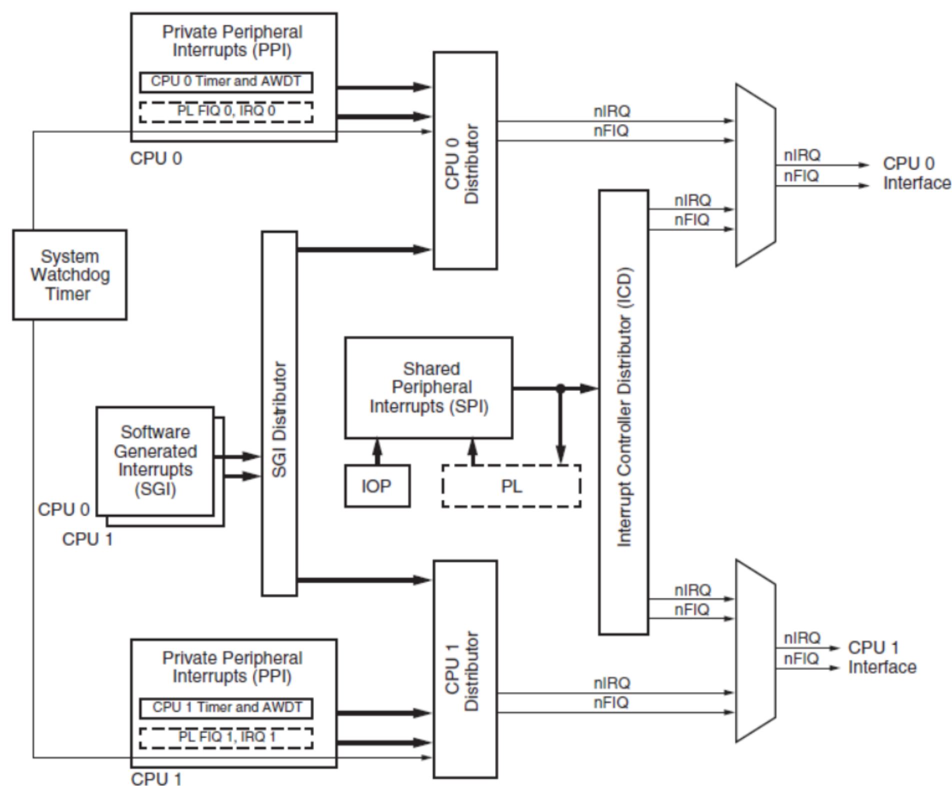


Figure 3-4 GIC Block Diagram

3.4.2 Functional Description

Software Generated Interrupts (SGI)

Each CPU can interrupt itself, the other CPU, or both CPUs using a software generated interrupt (SGI). There are 16 software generated interrupts whose IDs are from 0 to 15. An SGI is generated by writing the SGI interrupt number to the ICDSGIR register and specifying the target CPU(s). This write occurs via the CPU's own private bus.

Each CPU has its own set of SGI registers to generate one or more of the 16 software generated interrupts. All SGIs are edge triggered. The sensitivity types for SGIs are fixed and cannot be changed; the ICDICFR0 register is read-only, since it specifies the sensitivity types of all the 16 SGIs.

CPU Private Peripheral Interrupts (PPI)

Each CPU connects to a private set of five peripheral interrupts whose IDs are from 7 to 31 since IDs from 16 to 26 are reserved as shown in Table 2. The sensitivity types for PPIs are fixed and cannot be changed.

PPI #	IRQ ID #	Source
~	16:26	Reserved
0	27	Global Timer
1	28	nFIQ
2	29	CPU Private Timer
3	30	AWDT (Watchdog)
4	31	nIRQ

Table 2 PPI Interrupt Requests IDs

Shared Peripheral Interrupts (SPI)

A group of approximately 60 interrupts from various modules can be routed to one or both CPUs or the PL. The interrupt controller manages the prioritization and reception of these interrupts for the CPUs. Except for IRQ #61 through #68 and #84 through #91 (PL), all interrupt sensitivity types are fixed by the requesting sources and cannot be changed.

The SPI interrupts are listed in Table 3.

SPI #	IRQ ID #	Source
0:3	32:35	APU
~	36	Reserved
4:5	37:38	PMU
6	39	XADC
7	40	DevC
8	41	SWDT
9:11	42:44	TTC0
12:16	45:49	DMAC
17:18	50:51	Memory
19:27	52:60	IOP (0)
28:35	61:68	PL (0)
36:38	69:71	TTC1
39:42	72:75	DMAC
43:50	76:83	IOP (1)
51:58	84:91	PL (1)
59	92	SCU
~	93:95	Reserved

Table 3 SPI Interrupt Requests IDs

3.4.3 Interrupt Sensitivity and Targeting

There are three types of interrupts that come into the GIC as explained before: SPI, PPI and SGI. In a general sense, the interrupt signals include a sensitivity setting, whether one or both CPUs handle the interrupt, and which CPU or CPUs are targeted: zero, one, or both. However, the functionality of most interrupt signals includes fixed settings, while others are partially programmable.

<u>Shared Peripheral Interrupts (SPI)</u>

The SPI interrupts can be targeted to any number of CPUs, but only one CPU handles the interrupt. If an interrupt is targeted to both CPUs and they respond to the GIC at the same time, the MPcore ensures that only one of the CPUs reads the active interrupt ID#. The other CPU receives the Spurious ID# 1023 interrupt or the next pending interrupt, depending on the timing. This removes the requirement for a lock in the interrupt service routine. Targeting the CPU is done by the ICDIPTR [23:8] registers.

Private Peripheral Interrupts (PPI)

Each CPU has its own separate PPI interrupts with fixed functionality; the sensitivity, handling, and targeting of these interrupts are not programmable. Each interrupt only goes to its own CPU and is handled by that CPU.

Software Generated Interrupts (SGI)

The SGI interrupts are always edge sensitive and are generated when software writes the interrupt number to ICDSGIR register. All the targeted CPUs defined in the ICDIPTR [23:8] must handle the interrupt to clear it.

3.4.4 Interrupt Prioritization and Handling

All of the interrupt requests (PPI, SGI and SPI) are assigned a unique ID number. The controller uses the ID number to arbitrate. The interrupt distributor holds the list of pending interrupts for each CPU, and then selects the highest priority interrupt before issuing it to the CPU interface. Interrupts of equal priority are resolved by selecting the lowest ID.

The prioritization logic is physically duplicated to enable the simultaneous selection of the highest priority interrupt for each CPU. The interrupt distributor holds the central list of interrupts, processors and activation information, and is responsible for triggering software interrupts to the CPUs.

SGI and PPI distributor registers are banked to provide a separate copy for each connected processor. Hardware ensures that an interrupt targeting several CPUs can only be taken by one CPU at a time. The interrupt distributor transmits to the CPU interfaces the highest pending interrupt. It receives back the information that the interrupt has been acknowledged, and can then change the status of the corresponding interrupt. Only the CPU that acknowledges the interrupt can end that interrupt.

If the interrupt is pending in the GIC and IRQ is de-asserted, the interrupt in the GIC becomes inactive (and the CPU never sees it). If the interrupt is active in the GIC (because the CPU interface has acknowledged the interrupt), then the software ISR determines the cause by checking the GIC registers first and then polling the I/O Peripheral interrupt status registers.

3.5 UART

UART stands for universal asynchronous receiver and transmitter. It is one of the peripheral blocks found in any SoC. It supports full-duplex transmission and reception of data bits through different modes. The main usage of the UART is going from parallel to serial communication and vice versa. Also, as it is asynchronous, the receiver has to wait for the start bit and start reception. The UART implemented according to the specifications extracted from the Zynq-7000 FPGA.

UART supports a wide range of programmable baud rates and I/O signal formats. The controller can accommodate automatic parity generation and multi-master detection mode. The UART operations are controlled by the configuration and mode registers. The state of the FIFOs, modem signals and other controller functions are read using the status, interrupt status and modem status registers.

The controller is structured with separate Rx and Tx data paths. Each path includes a 64-byte FIFO. The controller serializes and de-serializes data in the Tx and Rx FIFOs and includes a mode switch to support various loopback configurations for the RxD and TxD signals. The FIFO interrupt status bits support a polling or interrupt driven handler. Software reads and writes data bytes using the Rx and Tx data port registers.

When the UART is being used in a modem-like application, the modem control module detects and generates the modem handshake signals and controls the receiver and transmitter paths according to the handshaking protocol. There are 2 UARTs inside the Zynq-7000. Both can work independently and there is an option to make them transmit and receive to each other.

3.5.1 Features

The UART supports a lot of features. It includes features of the frame form. The number of bits can be adopted and the parity bit of the frame. Also, the Number of bytes and the modes supported in the UART is added. Besides, the interrupts are generated to the GIC. The option of setting the two UARTs together to transmit and receive between each other is supported but is performed by the main processor of the system.

Each UART controller (UART 0 and UART 1) has the following features:

- Programmable baud rate generator
- 64-byte receive and transmit FIFOs
- Programmable protocol
  - 6, 7, or 8 data bits
  - 1, 1.5, or 2 stop bits
  - Odd, even, space, mark, or no parity
- Parity, framing and overrun error detection
- Line-break generation
- Interrupts generation
- RxD and TxD modes: Normal/echo and diagnostic loopbacks using the mode switch
- Loop UART 0 with UART 1 option
- Modem control signals: CTS, RTS, DSR, DTR, RI and DCD are available only on the EMIO interface

3.5.2 Block Diagram

The overall system of the UART works according to Figure 3-5. The figure explains the block on system level. This overview shows how the main signals entering to the block and its interface with the I/O pins, the interface with the ABP bus and the interface with the GIC when interrupts are generated. The block has two different clocks coming from two different places (UART-REF-CLK and CPU-1X-CLK).

The pins of the Tx, Rx, GIC and modem signals are shown. Also, the interface of the controller with the registers appeared to control the features of the frame to be sent or received. The main brain of the UART is the controller that works and manages the other modules inside.

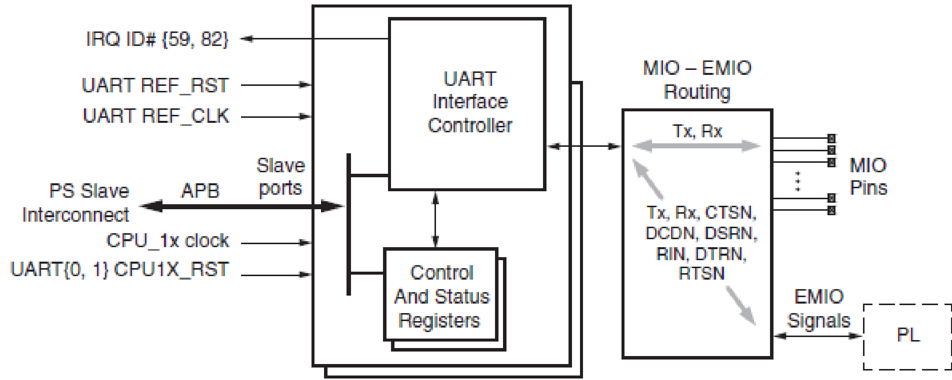


Figure 3-5 UART System block

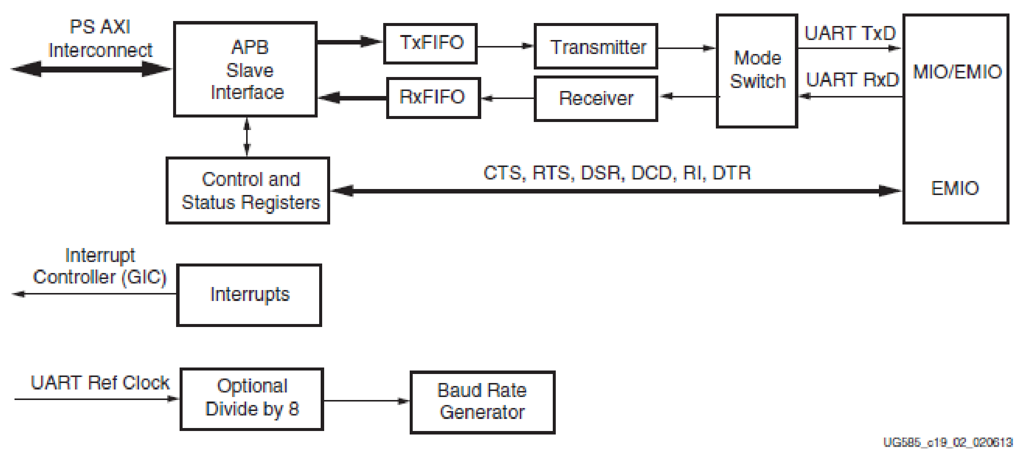


Figure 3-6 UART Block Diagram

The block diagram shows the internal structure of the UART block. And it is divided into several internal blocks. It is explained in the following section.

3.5.3 Functional Description

Control Logic

The control logic contains the Control register and the Mode register, which are used to select the various operating modes of the UART. The Control register enables, disables, and issues soft resets to the receiver and transmitter modules. In addition, it restarts the receiver timeout period, and controls the transmitter break logic. Receive line break detection must be implemented in Software. It will be indicated by a Frame Error followed by one or more zero bytes in the RxFIFO.

The Mode register selects the clock used by the baud rate generator. It also selects the bit length, parity bit and stop bit to be used by transmitted and received data. In addition, it selects the mode of operation of the UART, switching between normal UART mode, automatic echo, local loopback, or remote loopback, as required.

Baud Rate Generator

The baud rate generator furnishes the bit period clock, or baud rate clock, for both the receiver and the transmitter. The baud rate clock is implemented by distributing the base clock uart_clk and a single cycle clock enable to achieve the effect of clocking at the appropriate frequency division. The effective logic for the baud rate generation is shown in Figure 3-7.

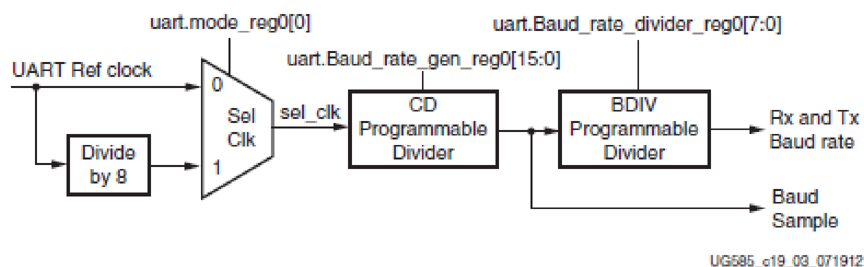


Figure 3-7 Baud Rate Generator

The baud rate generator can use either the master clock signal, Uart_ref_clk, or the master clock divided by eight, uart_ref_clk/8. The clock signal used is selected according to the value of the CLKS bit in the Mode register (uart.mode_reg0). The resulting selected clock is termed sel_clk in the following description.

The sel_clk clock is divided down to generate three other clocks: baud_sample, baud_tx_rate, and baud_rx_rate. The baud_tx_rate is the target baud rate used for transmitting data. The baud_rx_rate is nominally at the same rate, but gets resynchronized to the incoming received data. The baud_sample runs at a multiple ([BDIV] + 1) of baud_rx_rate and baud_tx_rate and is used to over-sample the received data.

The sel_clk clock frequency is divided by the CD field value in the Baud Rate Generator register to generate the baud_sample clock enable. This register can be programmed with a value between 1 and 65535. The baud_sample clock is divided by [BDIV] plus 1. BDIV is a programmable field in the Baud Rate

Divider register and can be programmed with a value between 4 and 255. It has a reset value of 15, inferring a default ratio of 16 baud_sample clocks per baud_tx_clock /

baud_rx_rate. Thus, the frequency of the baud_sample clock enable is shown in the following equation:

$$baud\ sample = (sel_clk)/CD$$

The frequency of the baud_rx_rate and baud_tx_rate clock enables is shown in the following equation:

$$baud_rate=(sel_clk)/(CD*(BDIV+1))$$

Transmit FIFO

The transmit FIFO (TxFIFO) stores data written from the APB interface until it is removed by the transmit module and loaded into its shift register. The TxFIFO maximum data width is eight bits. Data is loaded into the TxFIFO by writing to the TxFIFO register. When data is loaded into the TxFIFO, the TxFIFO empty flag is cleared and remains in this Low state until the last word in the TxFIFO has been removed and loaded into the transmitter shift register. This means that host software has another full serial word time until the next data is needed, allowing it to react to the empty flag being set and write another word in the TxFIFO without loss in transmission time.

The TxFIFO full interrupt status (TFULL) indicates that the TxFIFO is completely full and prevents any further data from being loaded into the TxFIFO. If another APB write to the TxFIFO is performed, an overflow is triggered and the write data is not loaded into the TxFIFO. The transmit FIFO nearly full flag (TNFULL) indicates that there is not enough free space in the FIFO for one more write of the programmed size, as controlled by the WSIZE bits of the Mode register.

The TxFIFO nearly-full flag (TNFULL) indicates that there is only byte free in the TxFIFO. A threshold trigger (TTRIG) can be setup on the TxFIFO fill level. The Transmitter Trigger register can be used to setup this value, such that the trigger is set when the TxFIFO fill level reaches this programmed value.

Transmitter Data Stream

The transmit module removes parallel data from the TxFIFO and loads it into the transmitter shift register so that it can be serialized. The transmit module shifts out the start bit, data bits, parity bit, and stop bits as a serial data stream. Data is transmitted, least significant bit first, on the rising edge of the transmit baud clock enable (baud_tx_rate). The uart.mode_reg0[CHRL] register bit selects the character length, in terms of the number of data bits. The uart.mode_reg0[NBSTOP] register bit selects the number of stop bits to transmit.

Receiver FIFO

The RxFIFO stores data that is received by the receiver serial shift register. The RxFIFO maximum data width is eight bits. When data is loaded into the RxFIFO, the RxFIFO empty flag is cleared and this state remains Low until all data in the RxFIFO has been transferred through the APB interface. Reading from an empty RxFIFO returns zero.

The RxFIFO full status (Chnl_int_sts_reg0 [RFUL] and Channel_sts_reg0 [RFUL] bits) indicates that the RxFIFO is full and prevents any further data from being loaded into the RxFIFO. When a space becomes available in the RxFIFO, any character stored in the receiver will be loaded. A threshold trigger (RTRIG) can be setup on the RxFIFO fill level. The Receiver Trigger Level register (Rcvr_FIFO_trigger_level0) can be used to setup this value, such that the trigger is set when the RxFIFO fill level transitions this programmed value. The Range is 1 to 63.

Receiver Data Capture

The UART continuously over-samples the UARTx_RxD signal using UARTx REF_CLK and the clock enable (baud sample). When the samples detect a transition to a Low level, it can indicate the beginning of a start bit. When the UART senses a Low level at the UART_RxD input, it waits for a count of half of BDIV baud rate clock cycles, and then samples three more times. If all three bits still indicate a Low level, the receiver considers this to be a valid start bit, as illustrated in Figure 3-8 for the default BDIV of 15.

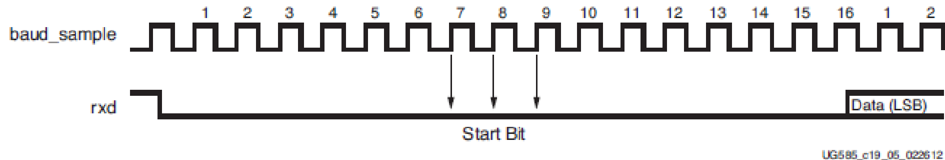


Figure 3-8 Default BDIV Receiver Data Stream

The logic value is determined by majority voting; two samples having the same value define the value of the data bit. When the value of a serial data bit has been determined, it is shifted to the receive shift register. When a complete character has been assembled, the contents of the register are then pushed to the RxFIFO. The Receiver has some errors and interrupts which are illustrated as follows:

- **Receiver Parity Error:**

  Each time a character is received, the receiver calculates the parity of the received data bits in accordance with the uart.mode_reg0 [PAR] bit field. It then compares the result with the received parity bit. If a difference is detected, the parity error bit is set = 1, uart.Chnl_int_sts_reg0 [PARE]. An interrupt is generated, if enabled.

- **Receiver Framing Error**:

  When the receiver fails to receive a valid stop bit at the end of a frame, the frame error bit is set =1, uart.Chnl_int_sts_reg0 [FRAME]. An interrupt is generated, if enabled.

- **Receiver Overflow Error:**

  When a character is received, the controller checks to see if the RxFIFO has room. If it does, then the character is written into the RxFIFO. If the RxFIFO is full, then the controller waits. If a subsequent start bit on RxD is detected and the RxFIFO is still full, then data is lost and the controller sets the Rx overflow interrupt bit, uart.Chnl_int_sts_reg0 [ROVR] = 1. An interrupt is generated, if enabled.

- **Receiver Timeout Mechanism:**

  The receiver timeout mechanism enables the receiver to detect an inactive RxD signal (a persistent High level). The timeout period is programmed by writing to the uart.Rcvr_timeout_reg0 [RTO] bit field. The timeout mechanism uses a 10-bit decrementing counter. The counter is reloaded and starts counting down whenever a new start bit is received on the RxD signal, or whenever software writes a 1 to uart.Control_reg0 [RSTTO] (regardless of the previous [RSTTO] value).

  If no start bit or reset timeout occurs for 1,023-bit periods, a timeout occurs. The Receiver timeout error bit [TIMEOUT] will be set in the interrupt status register, and the [RSTTO] bit in the Control register should be written with a 1 to restart the timeout counter, which loads the newly programmed timeout value. It is built as decrementing counter that moves downwards towards zero.

  When the decrementing counter reaches 0, the receiver timeout occurs and the controller sets the timeout interrupt status bit uart.Chnl_int_sts_reg0 [TIMEOUT] = 1. If the interrupt is enabled (uart.Intrpt_mask_reg0 [TIMEOUT] = 1), then the IRQ signal to the PS interrupt controller is asserted. Whenever the timeout interrupt occurs, it is cleared with a write back of 1 to the Chnl_int_sts_reg0 [TIMEOUT] bit. Software must set uart.Control_reg0 [RSTTO] = 1 to generate further receive timeout interrupts.

I/O Mode Switch

The mode switch controls the routing of the RxD and TxD signals within the controller as shown in Figure 3-9. The loopback using the mode switch occurs regardless of the MIO-EMIO routing of the UART TxD/RxD I/O signals. There are four operating modes as shown in Figure 3-9. The mode is controlled by the uart.mode_reg0 [CHMODE] register bit field: normal, automatic echo, local loopback and remote loopback.

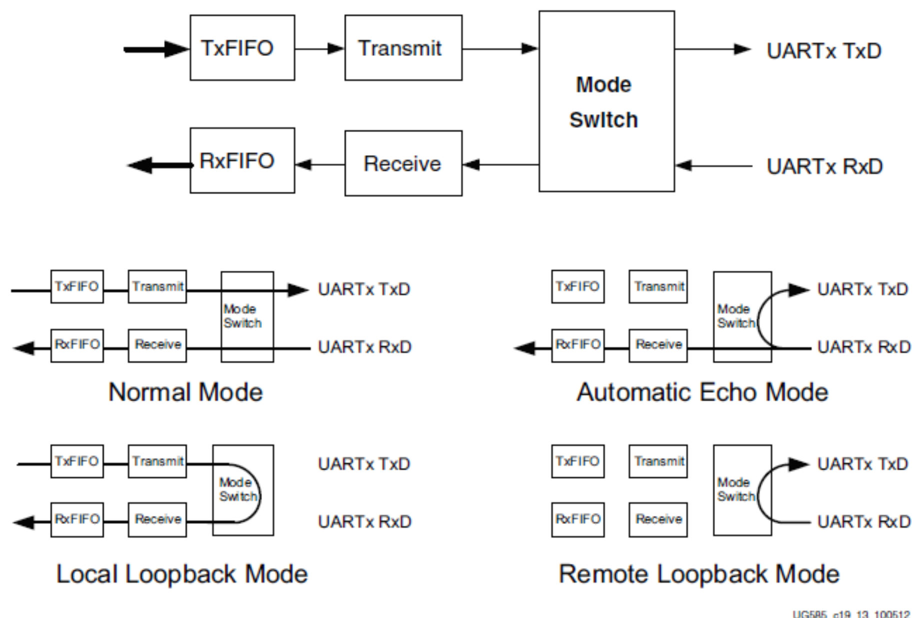


Figure 3-9 UART Mode Switch for TxD and RxD

- **Normal Mode:** Normal mode is used for standard UART operations.
- **Automatic Echo Mode:** Echo mode receives data on RxD and the mode switch routes the data to both the receiver and the TxD pin. Data from the transmitter cannot be sent out from the controller.
- **Local Loopback Mode:** Local loopback mode does not connect to the RxD or TxD pins. Instead, the transmitted data is looped back around to the receiver.
- **Remote Loopback Mode:** Remote loopback mode connects the RxD signal to the TxD signal. In this mode, the controller cannot send anything on TxD and the controller does not receive anything on RxD.

UART0-to-UART1 Connection

The I/O signals of the two UART controllers in the PS can be connected. In this mode, the RxD and CTS input signals from one controller are connected to the TxD and RTS output signals of the other UART controller by setting the slcr.LOOP [UA0_LOOP_UA1] bit = 1. The other flow control signals are not connected. This UART-to-UART connection occurs regardless of the MIO-EMIO programming.

3.5.4 Status and Interrupts

Interrupt and Status Registers

There are two status registers that can be read by software. Both show raw status. The Chnl_int_sts_reg0 register can be read for status and generate an interrupt. The Channel_sts_reg0 register can only be read for status. The Chnl_int_sts_reg0 register is sticky; once a bit is set, the bit stays set until software clears it. Write a 1 to clear a bit. This register is bit-wise ANDed with the Intrpt_mask_reg0 mask register. If any of the bit-wise AND functions have a result = 1, then the UART interrupt is asserted to the PS interrupt controller.

- **Channel_sts_reg0:** Read-only raw status of the UART. Writes are ignored

The various FIFO and system indicators are routed to the uart.Channel_sts_reg0 register and/or the uart.Chnl_int_sts_reg0 register as shown in Figure 3-10.

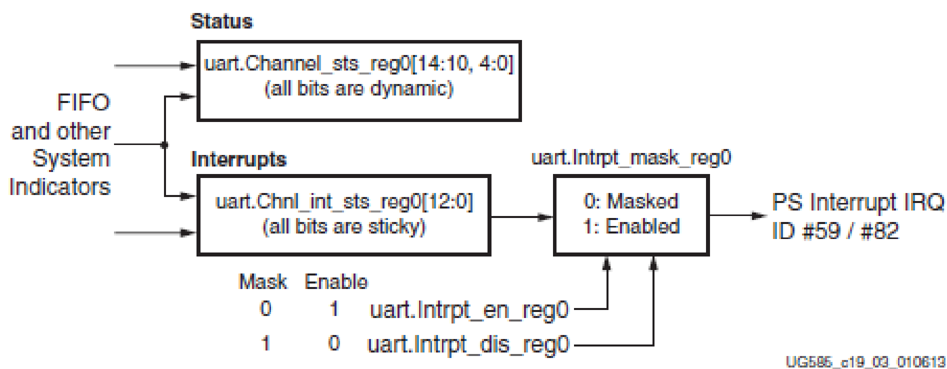


Figure 3-10 Interrupts and status signals

Interrupt Mask Register

Intrpt_mask_reg0 is a read-only interrupt mask/enable register that is used to mask individual raw interrupts in the Chnl_int_sts_reg0 register:

- If the mask bit = 0, the interrupt is masked.
- If the mask bit = 1, the interrupt is enabled.

This mask is controlled by the write-only Intrpt_en_reg0 and Intrpt_dis_reg0 registers. Each associated enable/disable interrupt bit should be set mutually exclusive (e.g., to enable an interrupt, write 1 to Intrpt_en_reg0[x] and write 0 to Intrpt_dis_reg0[x]).

Channel Status

These status bits are in the Channel_sts_reg0 register.

- **TACTIVE:** Transmitter state machine active status. If in an active state, the transmitter is currently shifting out a character.
- **RACTIVE:** Receiver state machine active status. If in an active state, the receiver is has detected a start bit and is currently shifting in a character.
- **FDELT:** Receiver flow delay trigger continuous status. The FDELT status bit is used to monitor the RxFIFO level in comparison with the flow delay trigger level.

Non-FIFO Interrupts

These interrupt status bits are in the Chnl_int_sts_reg0 register.

- **TIMEOUT:** Receiver Timeout Error interrupt status. This event is triggered whenever the receiver timeout counter has expired due to a long idle condition.
- **PARE:** Receiver Parity Error interrupt status. This event is triggered whenever the received parity bit does not match the expected value.
- **FRAME:** Receiver Framing Error interrupt status. This event is triggered whenever the receiver fails to detect a valid stop bit.
- **DMSI:** indicates a change of logic level on the DCD, DSR, RI or CTS modem flow control signals.

This includes High-to-Low and Low-to-High logic transitions on any of these signals.

FIFO Interrupts

The status bits for the FIFO interrupts are illustrated in Figure 3-11. These interrupt status bits are in the Channel Status (uart.Channel_sts_reg0) and Channel Interrupt Status (uart.Chnl_int_sts_reg0) registers with the exception that the [TOVR] and [ROVR] bits are not part of the uart.Channel_sts_reg0 register.

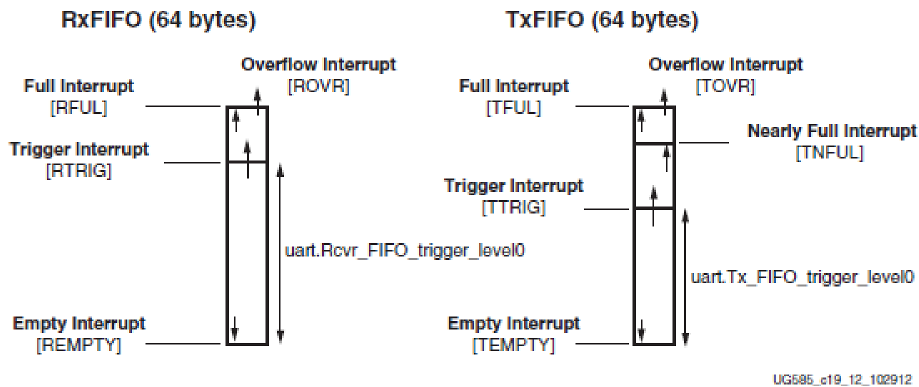


Figure 3-11 UART RxFIFO and TxFIFO Interrupt

The FIFO trigger levels are controlled by these bit fields:

- uart.Rcvr_FIFO_trigger_level0[RTRIG], a 6-bit field
- uart.Tx_FIFO_trigger_level0[TTRIG], a 6-bit field

3.5.5 Block Flow Building

Building of the UART block of the Zynq-7000 board has passed through many phases. After reading the manual, the UART is set to be divided into smaller blocks which are FIFO-TX, FIFO-RX, TX-controller, RX-controller, Sampler, Baud-rate-generator, Clock-divider, Main-controller and APB-interface. Each small block is concerned about doing one thing. Division of the functionality is very important and will be revealed later in the integration phase. It helps in detection of the errors and the problems that appears during the testing phase. Also, the interrupts are so important. There are two ways to add them to the functionality either from the beginning and test the functionality or build the block in normal case then add the interrupts to it later and test the functionality at the end. The first is the hardest to think about the interrupts while building but we chose it as to think of each smaller block as one big picture.

Moving to the building flow, the start comes from building the TX and RX controllers. It was a great challenge. They are responsible for the formation of the frame to be transmitted and the reception of the frame while receiving. The main challenge in those two blocks is choosing the most suitable states for those blocks as mentioned in the state diagrams.

The TX-controller is divided into 6 main states which are Idle, Start, Data, Parity, Stop, Break. The Idle state is the state where the TX keeps the bus idle by sending 1 on it. The Start state is responsible for getting down from 1 to 0 this trigger represents the start of data frame. The data state is the state where the data is sent bit by bit according to their size determined by frame characters either 6, 7 or 8. The parity state is adding a parity bit to the frame which is one of the following (even parity, odd parity, zero, mark, or no parity).

The Stop state which puts the number of stop bits added to the frame. The break state is recently added to halt the transmitter when it works as modem. The following state diagram in fig-x illustrates the previous explanation. Moving to the receiver state machine it is almost like the TX one except for some new things. The RX state machine works as follows with the following states (Idle, Start, Data, Parity, Stop, Break, Timeout, Parity, Frame, Save). First, IDLE state is the state where the RX is idle. Start state here unlike the receiver differs somehow through having timeout counter to detect the start bit but if it is not found it goes to Timeout state then to Idle again.

Moving to Data state, it ensures the reception of the data bit by bit and samples it using sampler block. Then, the Parity state appears to have different role in the RX as the block calculates the parity of the received data and compare them to the received parity bit. If it is true, it proceeds to the next state. But if it is false, it goes to the Parity state where parity interrupt is raised then moves to idle state. Detecting the stop bits is the last part in receiving the frame. If they are correctly received, it will go to save state to save data. If not, it will go to Frame state triggering Frame interrupt then to Idle state.

The Save state is to save the data to the FIFO-RX and return back to Idle state. The state diagram of the receiver is illustrated in Figure 3-12.

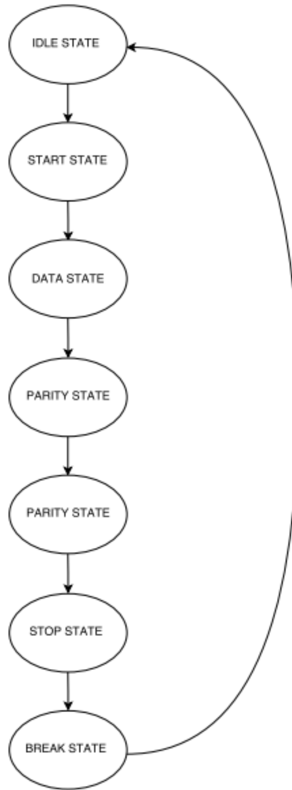


Figure 3-12 TX state diagram

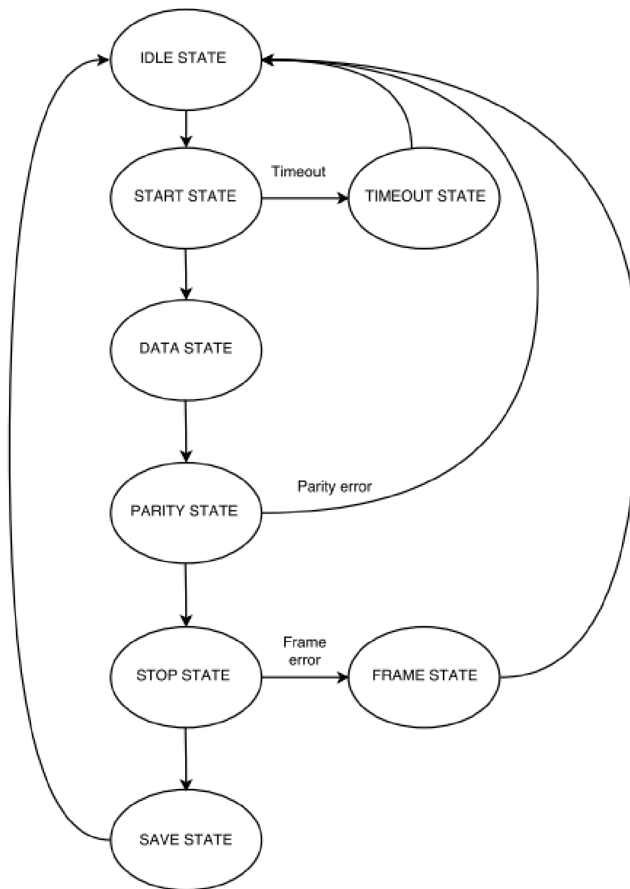


Figure 3-13 RX state diagram

After setting those controllers, it is the role of the testing to ensure that those two blocks are achieving the required functionality. After building the TX and RX controllers, the two FIFOs are then built. The FIFOs are like a small memory to save the data. Each FIFO can hold till 64 bytes. The challenge is how to get round in the FIFO when the pointers reach the end. It is solved through a trick of setting the writer and reader pointers to move in double the size of the FIFO (128 bytes). It is abstracted from the pointers in data structure algorithms. The FIFOs have to be interfaced from the two sides. The first with the TX,RX controllers mentioned above. The other side with the main controller to get the bytes in TX_FIFO and out from the RX_FIFO will be discussed later. The interrupts concerning the FIFO are set inside the block.

Besides, building two of the important blocks that will help in reaching the results of the long simulation time which are the Baud-rate-generator and the Clock-divider. These two blocks are concerned with generating the baud rate and baud sample from the main clock coming to UART. We use hierarchal reference in those two blocks. The upper is the Baud-rate-generator and inside which embedded the Clock-divider. The value of the division will play an important role in proving the point as increasing the value of the division will make the simulation takes long time to show the results that will be mentioned later. The TX and RX controllers works on the baud rate generated form Baud-rate-generator. They are all connected through the main controller. The division values are defined through registers and they have reset default values. There is a small block which is the sampler, it is one of the blocks used in the receiver. The sampler is based on Majority check. It samples the bit three times and if it has majority of 2 or more '1's' it will give one otherwise will give zero. It is used in the receiver controller to detect whether the received bit is 1 or 0. It is built as the specifications mentioned in the manual with getting three samples.

Then, the first integration occurs between the previous blocks to be tested together that they meet the functionality required. The integration requires tracking all signals of the blocks to be totally connected and passing the tests on sending and receiving bits. It was a challenge to generate many tests in order to edit those blocks on that small scale. So that to be ready in further building of the coming blocks.

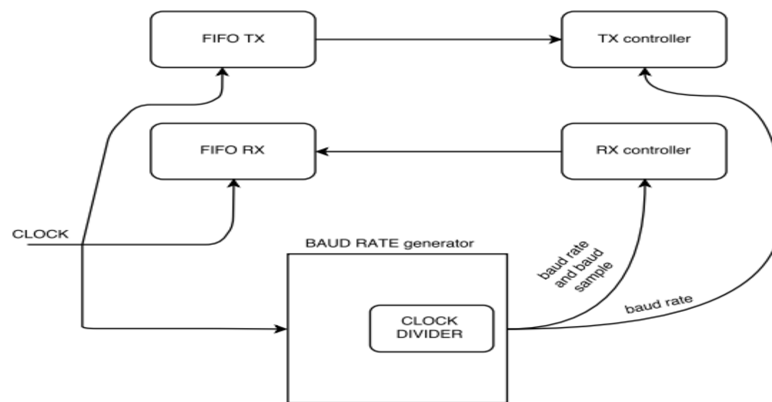


Figure 3-14 Controllers, FIFOs and Baud Rate generator

Moving to building the main controller, the master block that controls all the previous blocks and connect them together. It contains the registers and the connecting signals between the blocks. This block connects the two FIFOs with the TX,RX controllers and managing the clock for the blocks. Besides, it connects the FIFOs to the registers that are filled by the APB-interface through the databus. It handles the control of the overall system to meet the functionality with handling the interrupts. In addition to that this block is responsible to connect the internal blocks with the outside pins where the input is entered and the output is passed to. It is the brain of the UART where all the decisions are taken.

After the main controller is built and connecting all blocks under it, it is the time for validating this new integration. A rough test passes on the signals to ensure meeting the specifications. It is one of the long phases as it requires to do many edits to the blocks and the connections between them. It was one of the hardest phases and it took a very long time. After ensuring that everything is working as expected through the UART. It is time to go to the last part in the building of the UART.

Building the APB was the end of the building phase of the UART. The APB-interface includes all the interfaces of the UART with the external system. It has the connections with the APB and the external I/O pins. The APB-interface block is concerned with writing and reading from the databus and getting the address from the address bus. Besides, getting the inputs and passing the outputs to the pins. It acts as the link between the main-controller and the external system. The testing phase on the new integration to ensure that the block is fully connected and is meeting all the specifications.

3.6 SPI

The Serial Peripheral Interface (SPI) bus controller enables communications with a variety of peripherals such as memories, temperature sensors, pressure sensors, analog converters, real-time clocks, displays, and any SD card with serial mode support. The SPI controller can function in master mode and in slave mode.

In master mode, the controller drives the serial clock and slave selects with an option to support SPI's multi-master mode. The serial clock is derived from the PS clock subsystem. The controller initiates messages using up to 3 individual slave select (SS) output signals that can be externally expanded.

The controller reads and writes to slave devices by writing bytes to the 32-bit read/write data port register. In slave mode, the controller receives the serial clock from the external device and uses the SPI reference clock to synchronize data capture. The slave mode includes a programmable start detection mechanism when the controller is enabled while the SS is asserted.

The read and write FIFOs provide buffering between the SPI I/O interface and the software servicing the controller via the APB slave interface. The FIFO are used for both slave and master I/O modes.

3.6.1 Features

Each SPI controller is configured and controlled independent, they include the following features:

- Four wire bus – MOSI, MISO, SCLK, and SS
- up to 3 slave selects for master mode
- Full-duplex operation offers simultaneous receive and transmit
- 32-bit register programming via APB slave interface
- Memory mapped read/write data ports for Rx/Tx-FIFOs (byte-wide)
- 128-byte read and 128-byte write FIFOs
- Programmable FIFO thresholds status and interrupts
- Master I/O mode
- Manual and auto start transmission of data
- Manual and auto slave select (SS) mode

- Slave select signals can be connected directly to slave devices or expanded externally
- Programmable SS and MOSI delays
- Slave I/O mode
- Programmable start detection mode
- Programmable clock phase and polarity (CPHA, CPOL)
- Programmable interrupt-driven device or poll status

3.6.2 Block Diagram

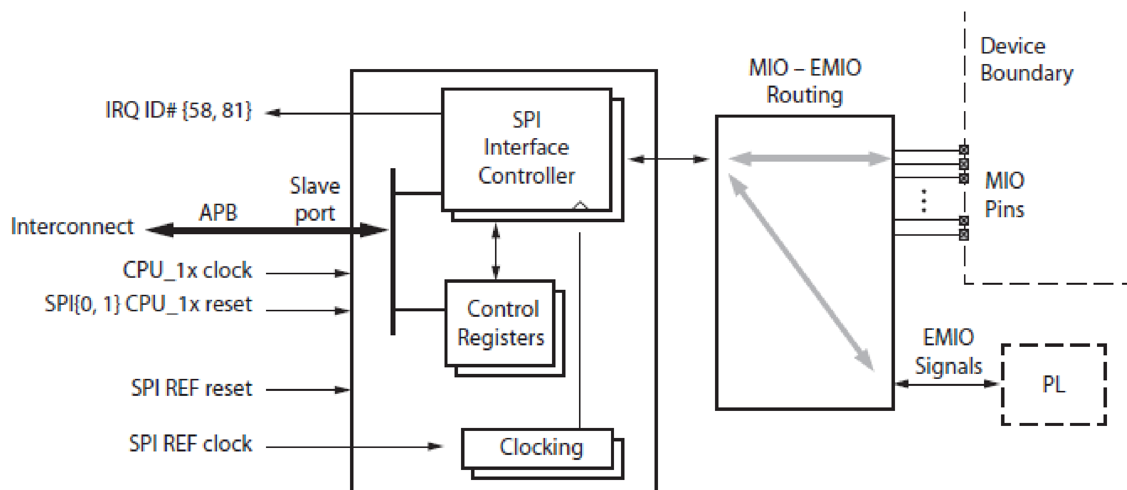


Figure 3-15 SPI system block diagram

The system block diagram of the SPI is shown in Figure 3-15 and it consists of the following main blocks:

1. **SPI Interface Controller:** The I/O signals of controller can be routed to MIO pins or the EMIO interface. Each controller also has individual interrupts signals to the PS interrupt controller and a separate reset signal. Each controller has its own set of control and status registers.

2. **Clocking:** The PS clock subsystem provides a reference clock to the SPI controller. The SPI reference clock is used for the controller logic and by the baud rate generator to create the SCLK clock for master mode.

3. **MIO-EMIO:** The SPI I/O signals can be routed to the MIO pins or the EMIO.

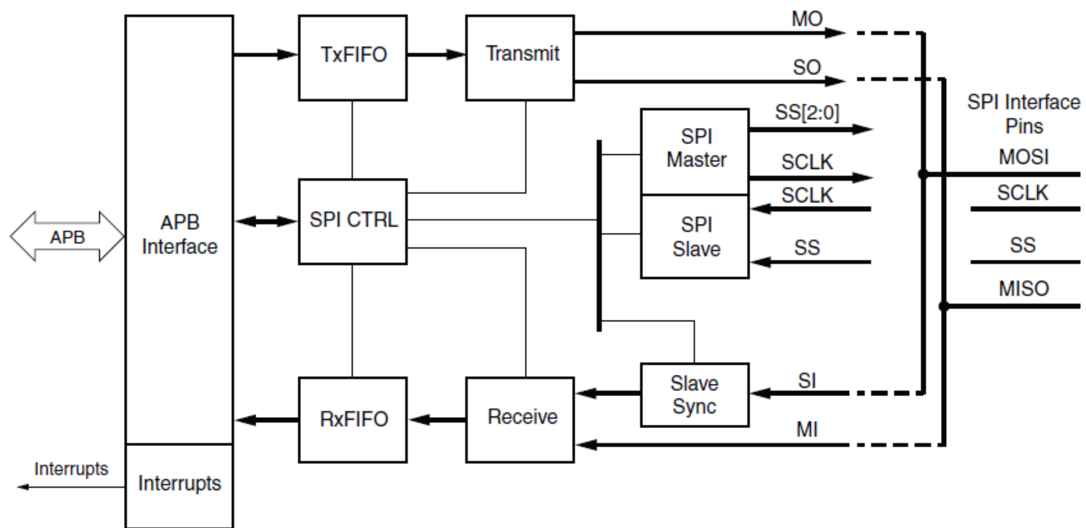


Figure 3-16 SPI functional block diagram

The functional block diagram of the SPI is shown in Figure 3-16 and it consists of the following main blocks:

1. **APB Slave Interface:** The 32-bit APB slave interface responds to register reads and writes, including data ports for reading and writing commands and data from and to the FIFOs. All registers transactions are 32 bits. The data ports use bits [7:0] of these ports

2. **SPI Master Mode:** When the controller operates in master mode, it drives the SCLK clock and up to 3 slave select output signals. The SS and start of transmission on MOSI can be manually controlled by software or automatically controlled by the hardware.

3. **SPI Slave Mode:** When the controller operates in slave mode, it uses a single slave select input (SS 0). The SCLK is synchronized to the controller reference clock (SPI_Ref_Clk).

4. **Tx and Rx FIFOs:** Each FIFO is 128 bytes. Software reads and writes these FIFOs using the register mapped data port registers. The FIFOs bridge two clock domains; APB interface and the controller's SPI_Ref_Clk. Software writes to the TxFIFO in the APB clock domain and the controller reads the TxFIFO in the SPI_Ref_Clk domain. The controller fills the RxFIFO in the SPI_Ref_Clk domain and software reads the RxFIFO in the APB clock domain.

3.6.3 Functional Description

Master Mode

In master mode, the SPI I/O interface can transmit data to a slave or initiate a transfer to receive data from a slave. The controller selects one slave device at the time using one of the three slave select lines.

Data Transfer

The SCLK clock and MOSI signals are under control of the master. Data to be transmitted is written into the TxFIFO by software using register writes and then unloaded for transmission by the controller hardware in a manual or automatic start sequence. Data is driven onto the master output (MOSI) data pin. Transmission is continuous while there is data in the TxFIFO.

Data is received serially on the MISO data pin and is loaded 8 bits at a time into the RxFIFO. Software reads the RxFIFO using register reads. For every "n" bytes written to the TxFIFO, there will be "n" bytes stored in RxFIFO that must be read by software before starting the next transfer.

Auto/Manual SS and Start

Data transfers on the I/O interface can be manually started using software or automatically started by the controller hardware. In addition, the slave select assertion/de-assertion can be done by the controller hardware or from software. These four combinations are shown in Table 4.

Slave Select Control	Data Transfer Start Control	Manual Slave Select	Manual Start Enable & Command	Operation
Manual SS (software)	Manual Start	1	1	Software controls the slave select and must issue the start command to serialize data.
Manual SS (software)	Auto Start	1	0	Software controls the slave select, but the controller hardware automatically starts to serialize data when there is data in the TxFIFO (recommended for general use).

Auto SS (controller)	Manual Start	0	1	Controller hardware controls the slave select, but the software must issue the start command to serialize data in the TxFIFO. This mode is applicable for specific use cases such as sending small chunks of data that fit into the SPI controller FIFO.
Auto SS (controller)	Auto Start	0	0	Controller hardware controls the slave select and serializes data when there is data in the TxFIFO.

Table 4 Auto/Manual SS and Start

Manual SS

Software selects the manual slave select method by setting the spi.Config_reg0 [Manual_CS] bit = 1. In this mode, software must explicitly control the slave select assertion/de-assertion. When the [Manual_CS] bit = 0, the controller hardware automatically asserts the slave select during a data transfer.

Automatic SS

Software selects the auto slave select method by programming the spi.Config_reg0 [Manual_CS] bit = 0. The SPI controller asserts/de-asserts the slave select for each transfer of TxFIFO content on to the MOSI signal. Software writes data to the TxFIFO and the controller asserts the slave select automatically, transmits the data in the TxFIFO and then de-asserts the slave select. The slave select gets de-asserted after all the data in the Tx FIFO is transmitted. This is the end of the transfer. Software ensures the following in automatic slave select mode.

- Software continuously fills the TxFIFO with the data bytes to be transmitted, without the TxFIFO becoming empty, to maintain an asserted slave select.
- Software continuously reads data bytes received in the RxFIFO to avoid overflow.

Software uses the TxFIFO and RxFIFO threshold levels to avoid FIFO under- and over-flows. The TxFIFO Not Full condition is flagged when the number of bytes in TxFIFO is less than the TxFIFO threshold level. The RxFIFO full condition is flagged when the number of bytes in RxFIFO is equal to 128.

Manual Start

- **Enable:** Software selects the manual transfer method by setting the spi.Config_reg0 [Man_start_en] bit = 1. In this mode, software must explicitly start the data transfer using manual start command mechanism. When the [Man_start_en] bit = 0, the controller hardware automatically starts the data transfer when there is data available in the TxFIFO.

- **Command:** Software starts a manual transfer by writing a 1 to the spi.Config_reg0 [Man_start_com] bit. When the software writes the 1, the controller hardware starts the data transfer and transfers all the data bytes present in the TxFIFO. The [Man_start_com] bit is self-clearing. Writing a 1 to this bit is ignored if [Man_start_en] = 0. Writing a 0 to [Man_start_com] has no effect, regardless of mode.

Slave Mode

In slave mode, the controller receives messages from the external master and outputs a simultaneous reply. The controller enters slave mode when spi.Config_reg0 [MODE_SEL] = 0 and spi.En_reg0 [SPI_EN] = 1.

The SCLK latches data on the MOSI input. If the slave select input signal is High (inactive), the controller ignores the MOSI input. When the slave select is asserted, it must be held active for the duration of the transfer. If SS de-asserts during the transfer, the controller sets the spi.Intr_status_reg0 [MODE_FAIL] interrupt bit. The software receives the [MODE_FAIL] interrupt so it can abort the transfer, reset the controller, and re-send the transfer.

The error mechanism is enabled by the [Modefail_gen_en] bit. Data to be sent to the master is written into the TxFIFO by software and then serialized onto the master input (MISO) signal by the controller. Transmission continues while there is still data in the TxFIFO and the slave select signal remains asserted (active Low).

Clocking

The slave select input pin must be driven synchronously to the SCLK input. The controller operates in the SPI_Ref_Clk clock domain. The input signals are synchronized and analyzed in the SPI_Ref_Clk domain.

Word Detection

The start of a word is detected in the SPI_Ref_Clk clock domain.

- **Detection when Controller is enabled:** If the controller is enabled (from a disabled state) at a time when SS is Low (active), the controller will ignore the data and wait for the SCLK to be inactive (a word boundary) before capturing data. The controller counts SCLK inactivity in the SPI_Ref_Clk domain. A new word is assumed when the SCLK idle count reaches the value programmed into the [Slave_Idle_coun] bit field.

- **Detection when SS asserts:** With the controller enabled and SS is detected High (inactive), the controller will assume the start of the word occurs on the next active edge of SCLK after SS transitions Low (active).

FIFOs

The Rx and Tx FIFOs are each 128 bytes deep.

- **RxFIFO:** If the controller attempts to push data into a full RxFIFO then the content is lost and the sticky overflow flag is set. No data is added to a full RxFIFO. Software writes a 1 to the bit to clear the [RX_OVERFLOW] bit.

- **TxFIFO:** If software attempts to write data into a full TxFIFO then the write is ignored. No data is added to a full TxFIFO. The [TX_FIFO_full] bit is asserted until the TxFIFO is read and the TxFIFO is no longer full. If the TxFIFO overflows, the sticky [RX_OVERFLOW] bit is set = 1.

FIFO Interrupts

The interrupt status bits (sticky and dynamic) are filtered by the mask register and then sent to the system interrupt controller. The mask register is controlled by the en/dis interrupt control registers. The Rx and Tx FIFO interrupts are illustrated in Figure 3-17.

Master Mode SCLK

The SCLK is driven by the controller in master mode. It is generated by the a divided-down SPI_Ref_Clk using the spi.Config_reg0 [BAUD_RATE_DIV] bit field.

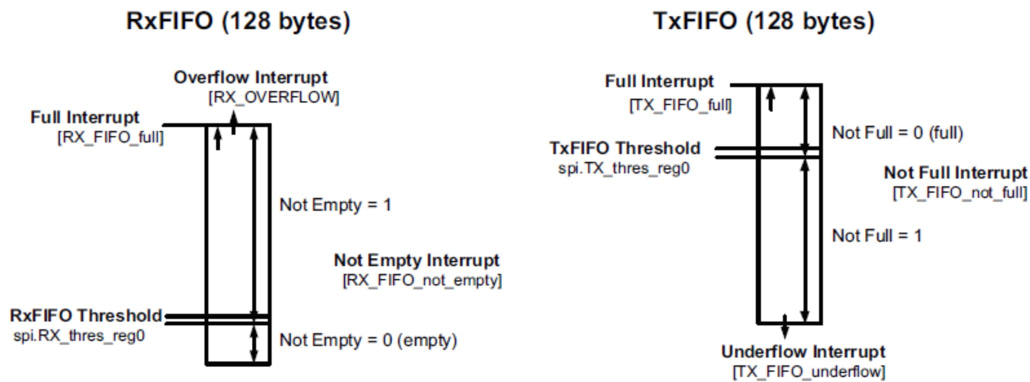


Figure 3-17 Rx and Tx FIFO interrupts

Frequency Ratio

The range of the baud rate divider is from a minimum of 4 to a maximum of 256 in binary steps (i.e., divide by 4, 8, 16,... 256). The controller supports various I/O signaling relationships for master mode. There are four combinations for setting the phase and polarity control bits, spi.Config_reg0 [CLK_PH] and [CLK_POL]. These parameters affect the active edge of the serial clock, the assertion of the slave select and the idle state of the SCLK. The clock phase parameter defines the state of the SS between words and the state of SCLK when the controller is not transmitting bits. The phase and polarity parameters are summarized in Table 5.

	CLK_PH = 0		**CLK_PH = 1**	
	CLK_POL = 0	**CLK_POL = 1**	**CLK_POL = 0**	**CLK_POL = 1**
Driving Edge	negative	positive	positive	negative
Sampling Edge	positive	negative	negative	positive
SS State between Words	active		inactive	
SCLK State outside of Word	active		inactive	

Table 5 Clock Phase and Polarity Parameters

1. CLK_PH = 0

- **SS Activity:** The master automatically drives the SS outputs inactive (High) for a the time programmed into the spi.Delay_reg0 [d_nss] bit field: Time = (1 + [d_nss]) * SPI_Ref_Clk clock period. The minimum time is 2 SPI_Ref_Clk clock periods.

- **Delay between Words:** The delay between the last bit period of the current word and the first bit period on the next word:

$$Time = (2 + [d_btwn]) * SPI_Ref_Clk$$

The minimum time is 3 SPI_Ref_Clk clock periods. This delay enables the TxFIFO to be unloaded and ready for the next parallel-to-serial conversion and to toggle slave select inactive High.

2. **CLK_PH = 1**

- **SS Activity:** The SS output signals are not driven inactive between words.
- **Delay between Words:** The minimum delay between the last bit period of the current word and the first bit period on the next word is, by default, one SPI_Ref_Clk cycles (configurable by the spi.Delay_reg0 register). This allows the TxFIFO to be unloaded and ready for the next parallel-to serial-conversion.

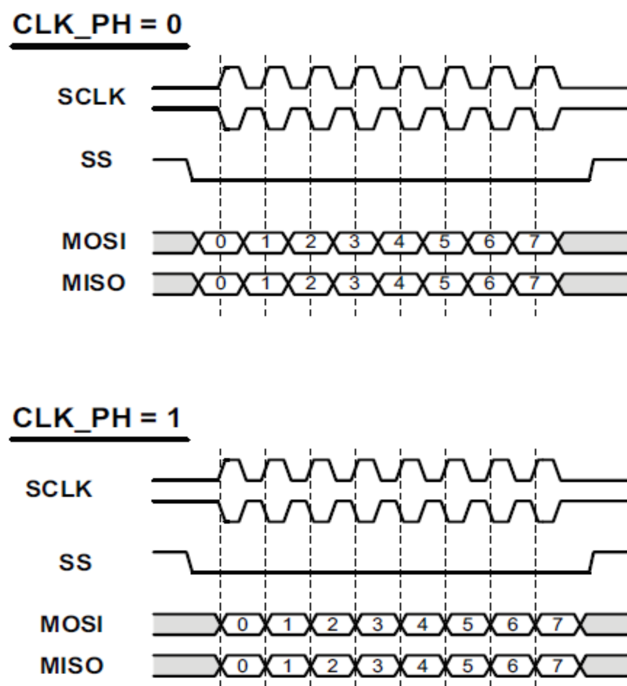


Figure 3-18 The clock phase parameter and the state of the SS

3.7 I2C

The Inter Integrated Circuit (I2C) is a semi-synchronous serial protocol for communication between components on the same circuit board.

3.7.1 Features

Zynq-7000 supports two I2C devices with these features:

- Two signals are needed for communication: SCL (clock) and SDA (data)
- Master generates the clock (SCL). The receiver samples the data from SDA when SCL is high and the transmitter writes when SCL is low.
- No strict baud rate is needed, master generates SCL and the slave acts accordingly.
- The only constraint is that the slave sampling rate must be greater than or equal to the master baud rate.
- Multi slave
- Each slave has an address. And it starts reacting when the master sends its address.
- Multi master
- Any master can take the bus as long as it is idle. Arbitration will be discussed in the following section.
- I2C bus specification version 2
- Supports 16-byte FIFO
- Programmable normal and fast bus data rates
- Master mode
    - Write transfer
    - Read transfer
    - Extended address support
    - Support HOLD for slow processor service
    - Supports TO interrupt flag to avoid stall condition
- Slave mode
    - Slave transmitter
    - Slave receiver
    - Fully programmable slave response address
    - Supports HOLD to prevent overflow condition
    - Supports TO interrupt flag to avoid stall condition
    - Software can poll for status or function as interrupt-driven device
    - Programmable interrupt generation

3.7.2 Block Diagram

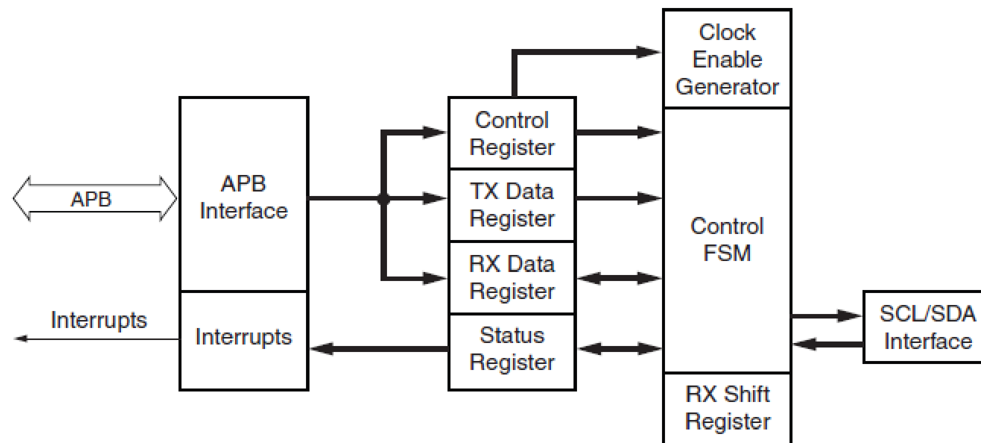


Figure 3-19 I2C Block Diagram

The module, as shown in Figure 3-19, mainly consists of:

- APB interface that writes to or read from the module registers
- Registers that are used to configure the module and exchange data.
- Clock enable generator. The communication mainly depends on the output clock from this block.
- Shift register which converts from serial to parallel during reception and parallel to serial during transmission.
- Interface to the SCL and SDA lines.
- Interrupt block to manage interrupt masking.

The most important registers located inside the I2C are:

- **Control register:** configure the mode of operation and the speed of the communication
  - DIV_A(15:14): 0-3: divisor_a for stage A clock divider.
  - DIV_B(13:8): 0-63: divisor_b for stage B clock divider
  - CLR_FIFO(6): initialize the FIFO to zeros and clears the transfer size register. Automatically gets cleared APB clock.
  - SLVMON(5): enables slave monitor mode.
  - HOLD(4): enables holding SCL line low when no data is available for transmission or no data can be received.
  - ACKEN(3): this bit needs to be set so than the module sends ACK when necessary.

- NEA(2): If zero then normal (7-bit) address is used.
- MS(1): 1 for master and 0 for slave.
- RW(0): 1 for master receiver, 0 for master transmitter.

- Status register: contains information about the current state of the module and communication
  - BA(8): bus active.
  - RXOVF(7): receiver overflow.
  - TXDV(6): transmit data valid.
  - RXDV(5): receiver data valid.
  - RXRW(3): mode of operation received from master (slave transmitter, slave receiver).
- Address register:
  - In slave mode: it contains the address of the module
  - In master mode: it contains the slave address to communicate with.
- Data register:
  - In transmitter mode: APB writes to FIFO through the data register
  - In receiver mode: APB reads from FIFO through data register.
- Transfer size register: the meaning of its content varies according to the mode of operation
  - Master transmitter: number of data bytes still not transmitted minus one
  - Master receiver mode: number of data bytes that are still expected to be received
  - Slave transmitter mode: number of bytes remaining in the FIFO after the master terminates the transfer
  - Slave receiver mode: number of valid data bytes in the FIFO
- Slave mod pause register: pause interval in slave monitor mode.
- Time out register: maximum interval to wait before issuing time out.
- Interrupt status register: contains the status of the interrupts.
  - ARB_LOST(9): master lost arbitration
  - RX_UNF(7): receive underflow (APB reads from FIFO while it is empty)
  - TX_OVF(6): transmitter overflow (APB writes to FIFO while it is full)

- RX_OVF(5): receive overflow (a new bit is received while the FIFO is full)
- SLV_RDY(4): monitored slave is ready
- TO(3): transfer time out
- NACK(2): transfer is not acknowledged
- DATA(1): in transmitter mode, means FIFO needs more data, in receiver mode means more data nead to be read from FIFO
- COMP(0): transfer is complete

- Interrupt mask register: masks for module interrupts. If a bit is set that means the corresponding interrupt is masked. It is read only register. But can be modified through interrupt enable and interrupt disable registers.
- Interrupt enable register: if a bit in this register is set, the corresponding bit in the mask register is cleared.
- Interrupt disable register: if a bit in this register is set, the corresponding bit in the mask register is set.

3.7.3 Functional Description

Device connections

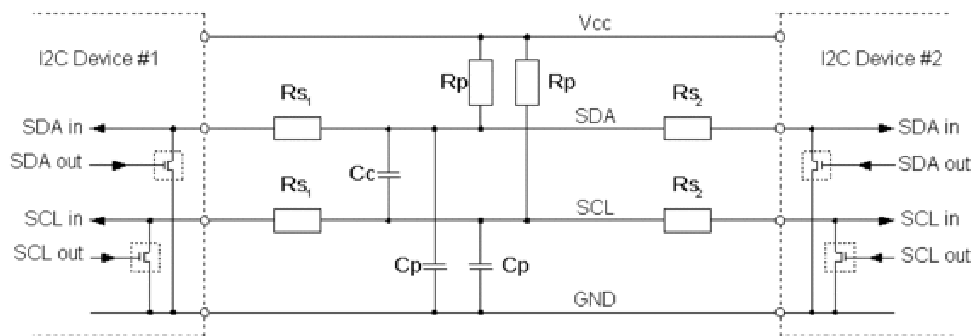


Figure 3-20 I2C Device Connections

As shown Figure 3-20, each device is connected to the bus through a transistor to ground. To write zero, the device pulls the line (SDA or SCL) to ground by turning the transistor on. Writing one is done by turning the transistor off and the pull up resistance (Rp) pulls the line up to Vcc. This technique is usually called open drain.

There are some points to note about the previous technique.

- There is no difference in the connection between the master and the slave. Thus, any connected device can be master or slave. This is decided by the device software.
- If all devices write '1' and one device writes '0', it wins. This is used in arbitration in case of multi masters as discussed in the following point.
- If the bus is idle and two masters started sending at the same time, the first device that writes '0' wins and the other device stops and wait until the bus is idle again.
- Every master must read SDA line after it has written '1' on the bus to check if another master is using the bus.
- Clock stretching can be used by slow slaves to slow down the communication. This is done by holding SCL low until the slave is ready to receive more data. This means the master must also read SCL after writing '1' on it to check if the slave is ready or not.

How it works

- The transaction goes through multiple steps as shown in Figure 3-21.
- Start condition: SDA transition from high to low while SCL is high. It is issued by the master at the start of the transaction.
- Addressing: after the master generates the start condition it sends the address of the desired slave. Address can be 10 bits or 7 bits.
- R/W: after sending the address, the master ends one more bit to tell the slave if it is a read or write transaction.
- ACK: master waits for slave to ACK to make sure the slave is ready for the transaction. ACK bit is also sent by the receiver after each successful reception of one byte of data.
- DATA: after the master sends the address and receives ACK the data starts to flow from TX to RX.
- Stop condition: SDA transition from low to high while SCL is high. It is issued by the master after the transaction ends to tell other masters that the bus is idle.

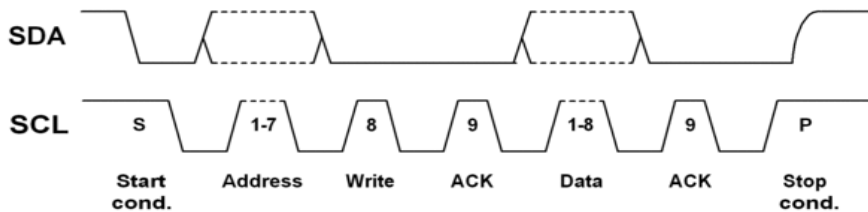


Figure 3-21 I2C Transaction waveform

In some cases, when the slave doesn't ACK on the address, the master may resend a start condition and send the address again instead of sending stop.

Zynq-7000 I2C controller

Zynq-7000 has an I2C module. It is a bus controller that can function as a master or a slave in a multi-master design.

- In master mode, a transfer can only be initiated by the processor writing the slave address into the I2C address register. The processor is notified of any available received data by a data interrupt or a transfer complete interrupt. If the HOLD bit is set, the I2C interface holds the SCL line Low after the data is transmitted to support slow processor service. The master can be programmed to use both normal (7-bit) addressing and extended (10-bit) addressing modes.

- In slave monitor mode, the I2C interface is set up as a master and continues to attempt a transfer to a slave until the slave device responds with an ACK. The HOLD bit can be set to prevent the master from continuing with the transfer, preventing an overflow condition in the slave. A common feature between master mode and slave mode is the timeout (TO) interrupt flag. If at any point the SCL line is held Low by the master or the accessed slave for more than the period specified in the Timeout register, a timeout (TO) interrupt is generated to avoid stall conditions.

I2C speed

The main clock used within the I2C interface is the clock enable signal as shown in Figure 3-22.

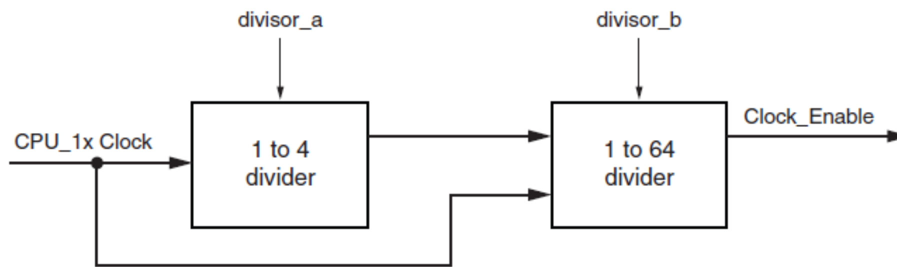


Figure 3-22 I2C Clock Division

- In slave mode, the clock enable is used to extract synchronization information for correct sampling of the SDA line.
- In master mode, the clock enable is used to establish a time base for generating the desired SCL frequency.

The frequency of the clock enable is defined by the frequency of the CPU_1x clock and the values of divisor_a and divisor_b using the following equation:

$$Clock_enable = \frac{\text{CPU_1x clock}}{22 * (divisor_a + 1) * (divisor_b + 1)}$$

3.7.4 Modes of Operation

Master mode

- Write transfer:

  The following steps must be done to accomplish an I2C write transfer:

  1. Write to the control register to set up SCL speed and addressing mode.
  2. Set the MS, ACKEN, and CLR_FIFO bits and clear the RW bit in the Control register.
  3. If required, set the HOLD bit. Otherwise write the first byte of data to the I2C Data register.
  4. Write the slave address into the I2C address register. This initiates the I2C transfer.
  5. Continue to load the remaining data to be sent to the slave by writing to the I2C Data register. The data is pushed in the FIFO each time the host writes to the I2C Data register.

When all data is transferred successfully, the COMP bit is set in the interrupt status register. A data interrupt is generated whenever there are only two bytes left for transmission in the FIFO.

When all data is transferred successfully, If the HOLD bit is not set, the I2C interface generates a STOP condition and terminates the transfer. If the HOLD bit is set, the I2C interface holds the SCL line Low

After the data is transmitted. The host is notified of this event by a transfer complete interrupt (COMP bit set) and the TXDV bit in the status register is cleared. At this point, the host can proceed in two ways:

1. Clear the HOLD bit. This causes the I2C interface to generate a STOP condition.
2. Supply more data by writing to the I2C address register. This causes the I2C interface to continue with the transfer, writing more data to the slave.

If at any point the slave responds with a NACK, the transfer automatically terminates and a transfer NACK interrupt is generated (the NACK bit set). When a NACK is received the Transfer Size register indicates the number of bytes that still need to be sent minus one. Unless the very last byte written by the host into the FIFO was a NACK byte, TXDV remains High. In this case, the host must clear the FIFO by setting the CLR_FIFO bit in the Control register.

If at any point the SCL line is held Low by the master or the accessed slave for more than the period specified in the Timeout register, a TO interrupt is generated and the outstanding amount of data minus one is then read from the Transfer Size register.

- Read Transfer:

  The following steps must be done to accomplish an I2C read transfer:

  1. Write to the Control register to set up the SCL speed and addressing mode.
  2. Set the MS, ACKEN, CLR_FIFO bits, and the RW bit in the Control register.
  3. If the host wants to hold the bus after the data is received, it must also set the HOLD bit.
  4. Write the number of requested bytes in the Transfer Size register.
  5. Write the slave address in the I2C Address register. This initiates the I2C transfer.

  The host is notified of any available received data in two ways:

1. If an outstanding transfer size is more than the FIFO size -2, a data interrupt is generated (DATA bit set) when there are two free locations available in the FIFO.
2. If an outstanding transfer size is less than FIFO size -2, a transfer complete interrupt is generated (COMP bit set) when the outstanding transfer size bytes are received.

In both cases, the RXDV bit in the status register is set.

If at any point the slave responds with NACK while the master transmits a slave address for a master read transfer, the transfer automatically terminates and a transfer NACK interrupt is generated (NACK bit is set). The outstanding amount of data can be read from the Transfer Size register. If at any point the SCL line is held Low by the master or the accessed slave for more than the period specified in the Timeout register, a TO interrupt is generated.

Slave monitor mode

This mode is meaningful only when the module is in master mode and bit SLVMON in the control register is set. The host must set the MS and SLVMON bits and clear the RW bit in the Control register. It must initialize the Slave Monitor Pause register.

The master attempts a transfer to a particular slave whenever the host writes to the I2C Address register. If the slave returns a NACK when it receives the address, the master waits for the time interval established by the Slave Monitor Pause register and attempts to address the slave again. The master continues this cycle until the slave responds with an ACK to its address or until the host clears the SLVMON bit in the Control register. If the addressed slave responds with an ACK, the I2C interface terminates the transfer by generating a STOP condition and a SLV_RDY interrupt.

The operation of the module in master mode is summarized in the **Chapter 5**.

Slave mode

The I2C interface is set up as a slave by clearing the MS bit in the Control register. The I2C slave must be given a unique identifying address by writing to the I2C address register. The SCL speed must also be set up at least as fast as the fastest SCL frequency expected to be seen. When in slave mode, the I2C interface operates as either a slave transmitter or a slave receiver

Slave transmitter

The slave becomes a transmitter after recognizing the entire slave address sent by the master and when the R/W field in the last address byte sent is High. This means that the slave has been requested to send data over the I2C bus and the host is notified of this through an interrupt through an interrupt to the GIC. At the same time, the SCL line is held Low to allow the host to supply data to the I2C slave before the I2C master starts sampling the SDA line. The host is notified of this event by setting the DATA interrupt flag.

At the same time, the SCL line is held Low to allow the host to supply data to the I2C slave before the I2C master starts sampling the SDA line. The host must supply data for transmission through the I2C data register so that the SCL line is released and transfer continues. If it does not write to the I2C data register before the timeout period expires, an interrupt is generated and a TO interrupt flag is set.

After the host writes to the I2C data register, the transfer continues by loading data in the FIFO while the transfer is in progress. The amount of data loaded in the FIFO might be a known system parameter or communicated in advance through a higher-level protocol using the I2C bus.

When there are only two valid bytes left in the FIFO for transmission, an interrupt is generated and the DATA interrupt flag is set. If the I2C master returns a NACK on the last byte transmitted from the FIFO, an interrupt is generated, and the COMP interrupt flag is set as soon as the I2C master generates a STOP condition.

The transfer must continue if the master acknowledges on the last byte sent out from the FIFO. At that moment, the DATA interrupt flag is set. The TXDV flag in the Status register is cleared because the FIFO is empty. If the I2C master terminates the transfer before all of the data in the FIFO is sent by the slave, the host is notified, and the NACK interrupt flag is set while TXDV remains set and the Transfer Size register indicates the remaining bytes in the FIFO. The host must set the CLR_FIFO bit in the Control register to clear the FIFO and the TXDV bit.

Slave Receiver

The slave becomes a receiver after recognizing the entire slave address sent by the master and when the R/W bit in the first address byte is Low. This means that the master is about to send one or more data bytes to the slave over the I2C bus.

After a byte is acknowledged by the I2C slave, the RXDV bit in the Status register is set, indicating that new data has been received. The host reads the received data through the I2C Data register. An interrupt is generated and the DATA interrupt flag is set when there are only two free locations left in the FIFO.

Whenever the I2C master generates a STOP condition, an interrupt is generated and the COMP interrupt flag is set. The Transfer Size register then contains the number of bytes received that are available in the FIFO. This number is decremented by the host on each read of the I2C Data register.

If the FIFO is full when one or more bytes are received by the I2C interface, an interrupt is generated and the RX_OVF interrupt flag is set. The last byte received is not acknowledged and the data in the FIFO is kept intact.

The HOLD bit can be set in the Control register to avoid overflow conditions when it is impossible to respond to interrupts in a reasonable time. If the HOLD bit is set, the I2C interface keeps the SCL line Low until the host clears resources for data reception. This prevents the master from continuing with the transfer, causing an overflow condition in the slave. The host clears resources for data reception by reading the data register.

If the HOLD bit is set and the I2C interface keeps the SCL line Low for longer than the timeout period, an interrupt is generated and the TO interrupt flag is set. The operation of the module in slave mode is summarized in **Chapter 5**.

3.8 DMA

Direct Memory Access (DMA) is a feature of computer systems that allows certain hardware subsystems to access main system memory (RAM), independent of the central processing unit (CPU).

Without DMA, when the CPU is using programmed input/output, it is typically fully occupied for the entire duration of the read or write operation, and is thus unavailable to perform other work. With DMA, the CPU first initiates the transfer, then it does other operations while the transfer is in progress, and it finally receives an interrupt from the DMA controller when the operation is done. This feature is useful at any time that the CPU cannot keep up with the rate of data transfer, or when the CPU needs to perform useful work while waiting for a relatively slow I/O data transfer. Many hardware systems use DMA. DMA is also used for intra-chip data transfer in multi-core processors. Computers that have DMA channels can transfer data to and from devices with much less CPU overhead than computers without DMA channels. Similarly, a processing element inside a multi-core processor can transfer data to and from its local memory without occupying its processor time, allowing computation and data transfer to proceed in parallel.

DMA can also be used for "memory to memory" copying or moving of data within memory. DMA can offload expensive memory operations, such as large copies or scatter-gather operations, from the CPU to a dedicated DMA engine.

Many systems have internal and external interfaces that produce or consume data. These can be simple UARTs, external bus devices such as PCI-based Ethernet controllers, or complicated video and graphics devices. A key part of any system is ensuring that data flowing in to and out of these interfaces is handled properly and not lost or corrupted.

In order for the system to operate on the data (decode a message, produce the next frame of a video, send data to a remote server, etc.) the data must normally be in the system main memory to allow the main processor to access it. A simple way of moving the data between the peripheral device and main memory is to use the main processor to perform load or store operations for each byte or word of data to be moved. The processor must wait for the peripheral to be ready before transferring

each byte or word, which can be done by polling a status register or by handling a "ready" interrupt from the device as shown in Figure 3-23. This is referred to as Programmed I/O.

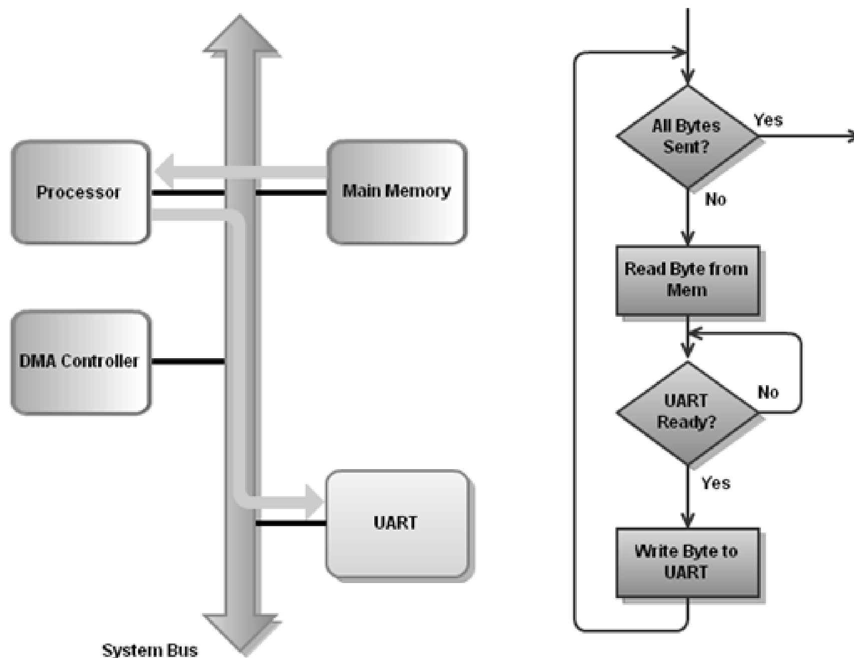


Figure 3-23 Programmed I/O to UART

If the device produces or consumes a small amount of data at low data rates, this may be the best way of managing the data transfer in and out of main memory; in fact, for some devices such as simple UARTs this may be the only way. However, using programmed I/O has some limitations: first, the fact that programmed I/O involves a software loop may limit its performance. Secondly, for many systems this is not a good use of processing time as the processor may be spending more time than is necessary or available moving data between main memory and its external interfaces.

The alternative is to set up a DMA transfer that gives the job of moving the data to a special-purpose device in the system. Once the processor has set up the transfer it can do something else while the transfer is in progress or wait to be notified when the transfer has finished. For example, the same transfer to the UART that was done using programmed I/O in the example above could be done using the system's DMA controller, which would free up the processor to do other work while the transfer was in progress.

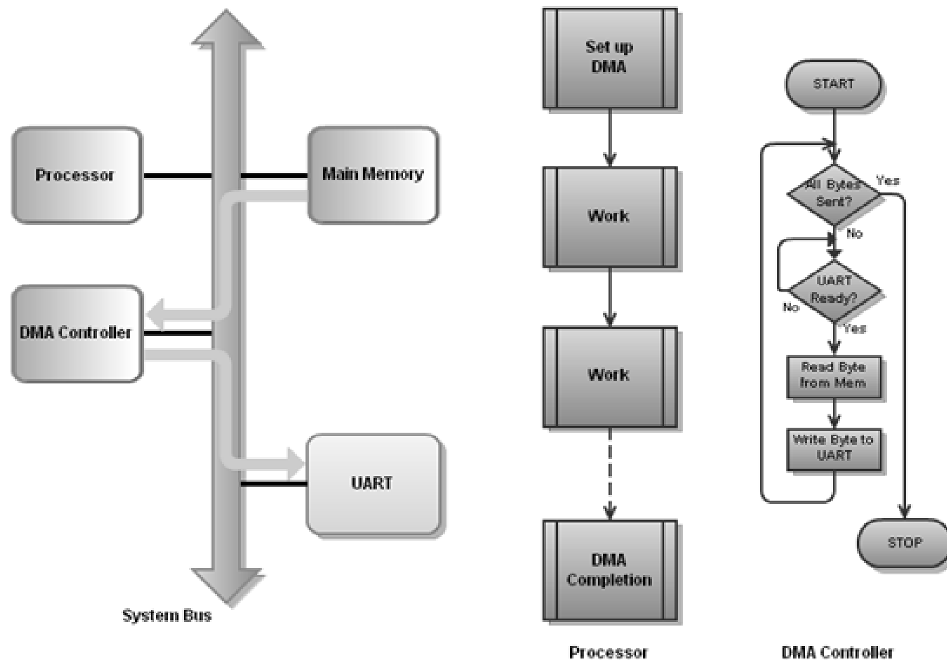


Figure 3-24 DMA Transfer to UART

DMA transfers can take different forms depending on the hardware design and the peripheral devices involved. The simplest is a known as a single-cycle DMA transfer and is typically used to transfer data between devices such as UARTs. In this situation, the peripheral device uses a control line to signal that it has data to transfer or requires new data. The DMA controller obtains access to the system bus, transfers the data, and then releases the bus. Access to the bus is granted when the processor, or another bus master, is not using the bus. Single-cycle DMA transfers are therefore interleaved with other bus transactions and do not much affect the operation of the processor.

Another type of transfer is a burst transfer. This is used to transfer a block of data in a series of back-to-back accesses to the system bus. The transfer starts with a bus request; when this is granted, the data is transferred in bursts, for example 128 bytes at a time. The burst size depends on the processor architecture and the peripheral, and may be programmable depending on the details of the hardware. While a burst transaction is occurring the processor will not be able to access the system bus. However, preventing the processor from accessing the system bus – for example to fetch new instructions or data from external memory – may cause it to stall, which can reduce the system performance. To minimize the effects of this problem, the DMA

controller may release the bus after a fixed number of burst transactions or when a pre-determined bandwidth limit has been reached.

In this documentation, DMA-330 will take place as it is an essential block from Zynq-7000 FPGA which will be designed to prove some simulation time and power concepts. Power and simulation time will be discussed in the later sections.

3.8.1 Features

Core-Link™ DMA-330 Controller provides the following features:

- AXI master interface to perform the DMA transfers.
- Two APB slave interfaces that control its operation.
- Trust-Zone secure technology with one APB interface operating in the secure state and the other operating in the Non-secure state.
- A small instruction set that provides a flexible method of specifying the DMA operations.
- Variable-length instructions to minimize the program memory requirements.
- Configurable RTL that enables you to optimize the DMAC for the application.
- Programmable security state for each DMA channel.
- Signals the occurrence of various DMA events using the interrupt output signals.
- It supports multiple transfer types:
- Memory-to-memory.
- Memory-to-peripheral.
- Peripheral-to-memory.
- Scatter-gather.

When a transfer is set up and initiated by programming all the transfer attributes into device registers it is known as a direct DMA transfer. Many devices also provide support for chained DMA transfer. Here the transfer attributes are specified in a device-specific data structure, often referred to as a DMA descriptor, which is similar to the direct DMA register set. Each descriptor in the chain has a pointer to the next. The address of the first descriptor in the chain is usually programmed into a register in the device.

Each descriptor points to a single block of memory or data buffer in the processor memory that is the source or destination of the transfer. There may be restrictions on how the memory is aligned and what the maximum block size is. For example, each buffer may have to be aligned to a 32- or 64-byte address boundary and be limited to 64 Kbytes. The memory blocks generally do not have to be contiguous but the memory inside each buffer must be made up of contiguous physical pages. This configuration is often referred to as scatter-gather DMA and operating systems may provide routines to help device drivers to build and manage scatter-gather memory buffers.

3.8.2 Block Diagram

DMAC is considered as a master on AXI bus. It asks for arbitration from the ARM processor which gives the bus to it when it is free. It is considered as a slave on APB bus. Configuration data are being delivered to DMAC through AXI-APB Bridge which is the master for the APB bus. It deals with UART, GPIO and Timer through peripheral request interface as shown in Figure 3-25.

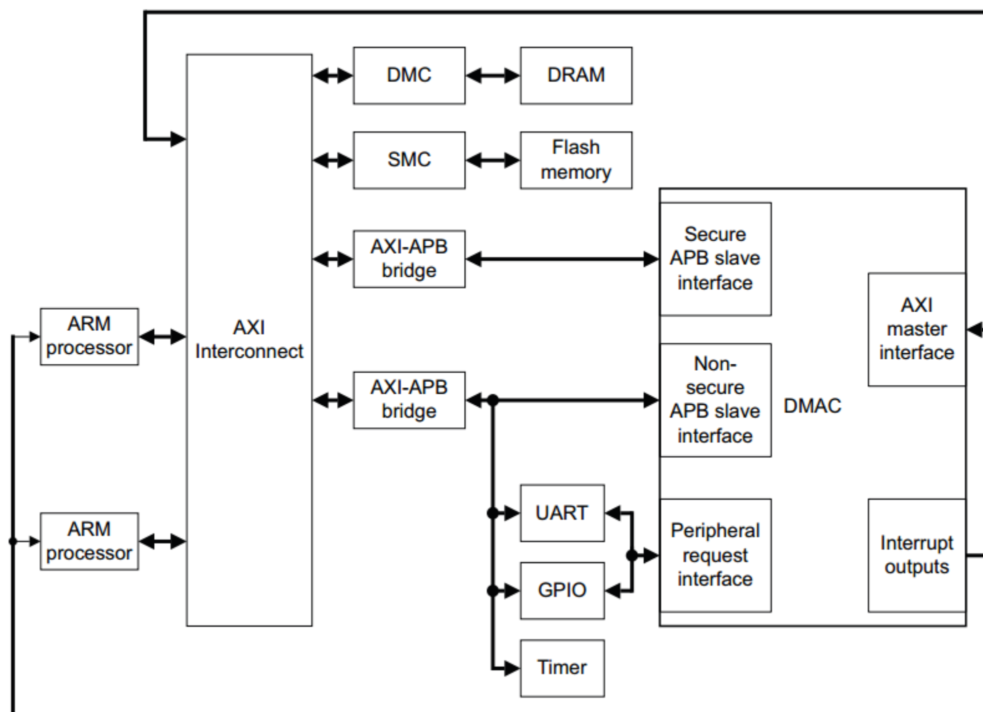


Figure 3-25 DMA Interface

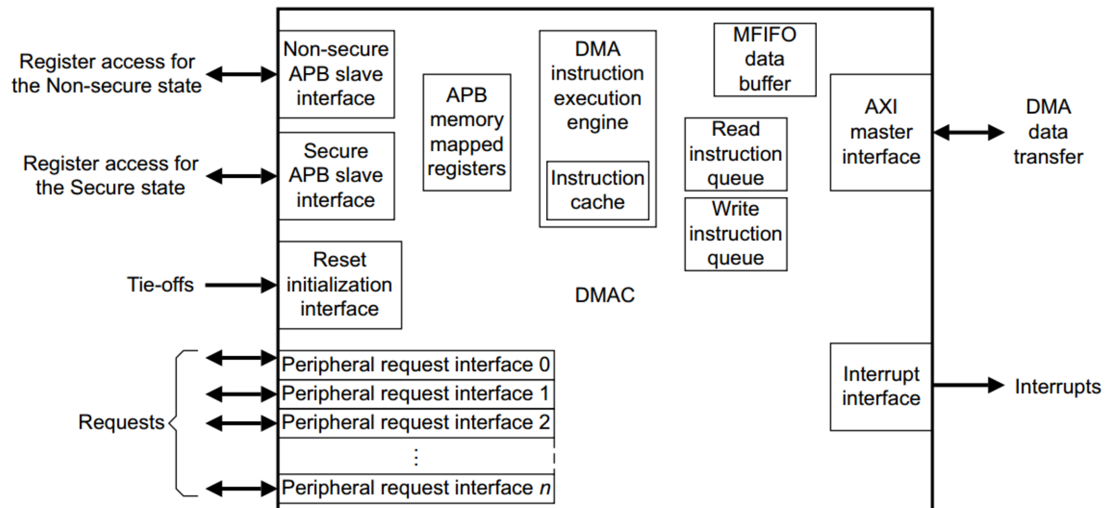


Figure 3-26 DMA Block Diagram

The DMA mainly consists, as shown in, of the following main functional blocks:

- Instruction execution engine: enables DMAC to process program code that controls a DMA transfer.
- AXI master interface is used to access the program code which stored in a region in system memory.
- Instructions are temporarily stored in cache.
- Variable length instructions are used from one byte to six bytes.
- There are eight channels in DMA each one capable of supporting a single concurrent thread of DMA operation.
- There is a separate state machine for each thread.
- There is a DMA manager which initialize DMA channel threads.
- DMAC executes only one instruction for AXI clock cycle.
- It uses a round-robin process when selecting the next active DMA channel thread to execute.
- Each channel has its separate *Program Counter* (PC) register.
- When a thread requests an instruction from an address, the cache performs a look-up. If a cache hit occurs, then the cache immediately provides the data. Otherwise, the thread is stalled while the DMAC uses the AXI master interface to perform a cache line fill. During cache line fill is in progress, the DMAC enables other threads to access the cache, but if another cache miss occurs, this stalls the pipeline until the first line fill is complete. If an

instruction is greater than 4 bytes, or spans the end of a cache line, the DMAC performs multiple cache accesses to fetch the instruction.

- When a DMA channel thread executes a load or store instruction, the DMAC adds the instruction to the relevant read or write queue. The DMAC uses these queues as an instruction storage buffer prior to it issuing the instructions on the AXI bus. The DMAC also contains a *Multi First-In-First-Out* (MFIFO) data buffer that it uses to store data that it reads, or writes, during a DMA transfer.
- The DMAC provides multiple interrupt outputs to enable efficient communication of events to external microprocessors.
- The peripheral request interfaces support the connection of DMA-capable peripherals to enable memory-to-peripheral and peripheral-to-memory DMA transfers to occur, without intervention from a microprocessor.
- Dual APB interfaces enable the operation of the DMAC to be partitioned into the secure state
and non-secure state.
- APB interfaces can be used to access status registers and also directly execute instructions in the DMAC.

3.8.3 Functional Description

Round-Robin Scheduling

Round Robin is the preemptive, can be splitted or interrupted, process scheduling algorithm.

- Each process is provided a fix time to execute, it is called a **quantum**.
- Once a process is executed for a given time period, it is preempted and other process executes for a given time period.
- Context switching is used to save states of preempted processes.
- Round-Robin example is expressed in the Fig.x

Process	Arrival Time	Execute Time	Waiting Time
P0	0	5	9
P1	1	3	2
P2	2	8	12
P3	3	6	11

Figure 3-27 Round Robin Scheduling

Average waiting time is the one of the most important factor in scheduling techniques. It gives us an impression if this technique suitable for these processes or not. It is also a comparison factor between multiple scheduling techniques.

Average Wait Time: (9+2+12+11) / 4 = 8.5

DMAC Interfaces

1- APB slave interfaces:

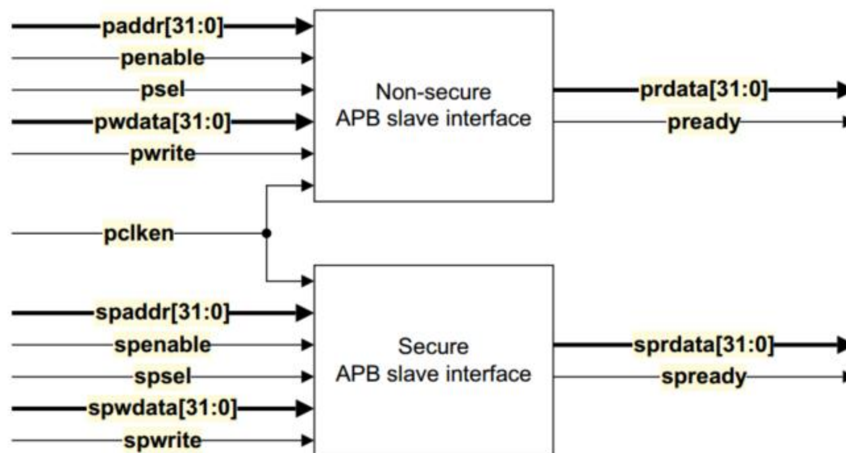


Figure 3-28 DMA APB Interface

- The DMAC allocates 4KB of memory for each APB interface.
- The same clock as the AXI domain clock, **aclk** clock the APB interfaces.
- As shown in Figure 3-28, DMAC provides a clock enable signal, **pclken**, that enables both APB interfaces to operate at a slower clock rate. The clock enable signal must be an integer divisor of **aclk**.
- To achieve that a clock divider is implemented to divide **aclk** to smaller values.

2- AXI interface:

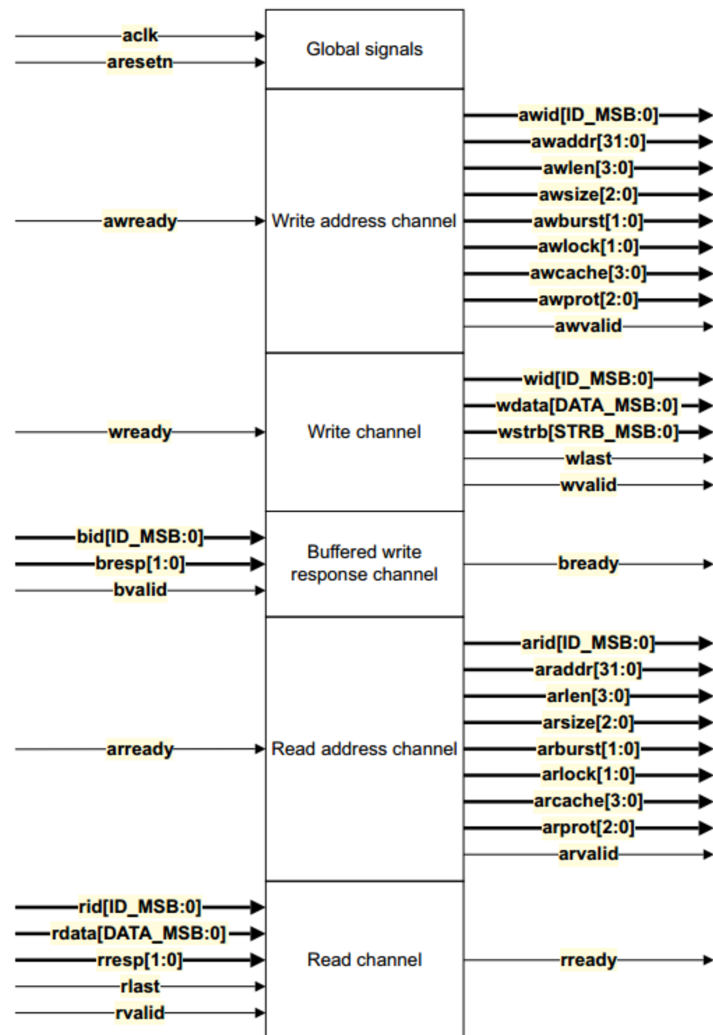


Figure 3-29 DMA AXI Interface

- The DMAC contains a single AXI master interface that enables it to transfer data from a source
  AXI slave to a destination AXI slave.
- The DMAC does not support locked or exclusive accesses.
- The DMAC does not generate wrapping address bursts.
- The value of ID_MSB depends on the number of DMA channels in the configured
  DMAC.
- The values of DATA_MSB and STRB_MSB depend on the data width of the configured
  DMAC.

- When a DMA channel thread accesses the AXI master interface, the DMAC signals the AXI identification tag to be the same number as the DMA channel. For example, when the program thread for DMA channel 5 performs a DMA store operation, the DMAC sets **AWID[2:0]** and **WID[2:0]** to 0b101.
- When the DMA manager thread accesses the AXI master interface, the DMAC signals the AXI identification tag to be the same number as the number of DMA channels that the DMAC provides. For example, if the DMAC is configured to provide eight DMA channels, when the DMA manager performs a read operation, the DMAC sets **ARID[3:0]** to 0b1000.
- Burst transfers are incremental.
- Cache write technique is write-through, Allocate on reads only.
- **ARLEN and ARSIZE for instruction fetches**

  When performing an instruction fetch, the DMAC sets **ARLEN** and **ARSIZE** as follows:

  Instruction cache length ≤ AXI data bus width

  • **ARLEN** = 1.

  • **ARSIZE** = length of instruction cache in bytes.

  Instruction cache length > AXI data bus width

  • **ARLEN** = ratio of the length of an instruction cache line in bytes to the width of the AXI data bus in bytes.

  • **ARSIZE** = width of AXI data bus in bytes.

3- Peripheral request interfaces:

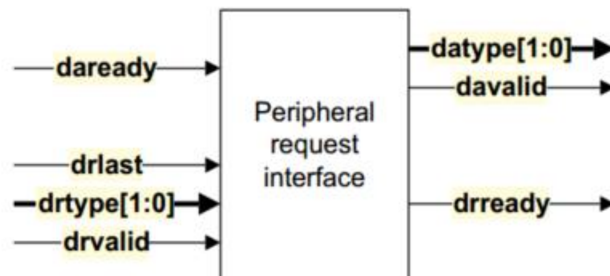


Figure 3-30 Peripheral request interfaces

DMAC operating states

Figure 3-31 illustrates the existing operating state in DMAC which DMA manager thread and DMA channel thread move between them when certain actions occurs.

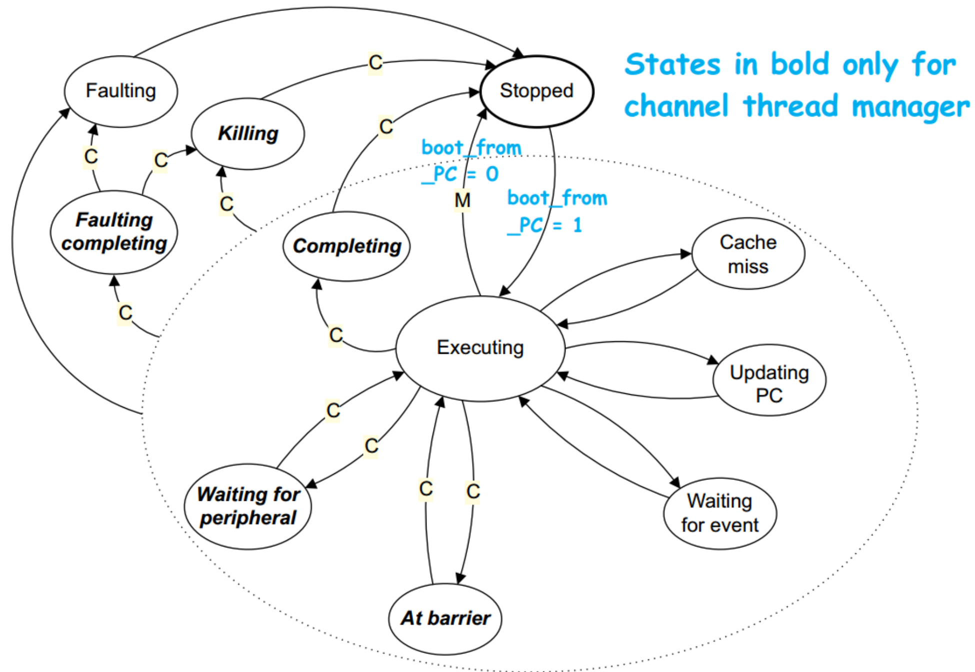
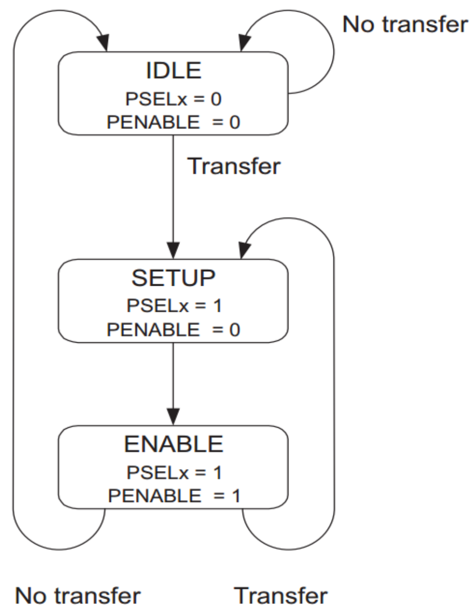


Figure 3-31 Operating state in DMAC

Using the APB slave interfaces

- **State diagram:**



Operations of the state machine is through the three states described in Ta

State	Description
IDLE	The default state for the peripheral bus.
SETUP	When a transfer is required the bus moves into the SETUP state, where the appropriate select signal, **PSELx**, is asserted. The bus only remains in the SETUP state for one clock cycle and will always move to the ENABLE state on the next rising edge of the clock.
ENABLE	In the ENABLE state the enable signal, **PENABLE** is asserted. The address, write and select signals all remain stable during the transition from the SETUP to ENABLE state. The ENABLE state also only lasts for a single clock cycle and after this state the bus will return to the IDLE state if no further transfers are required. Alternatively, if another transfer is to follow then the bus will move directly to the SETUP state. It is acceptable for the address, write and select signals to glitch during a transition from the ENABLE to SETUP states.

Table 6 Operations of the DMA state machine

- **Write Transfer:**

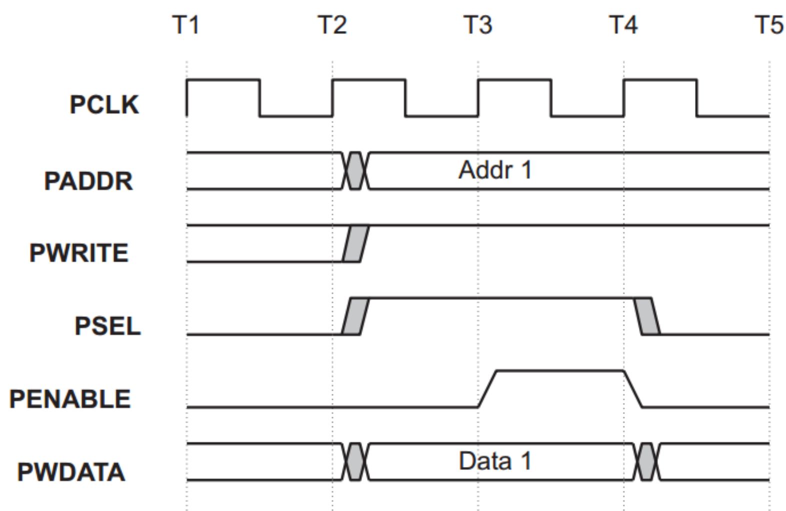


Figure 3-32 DMA Write Transfer

The write transfer starts with the address, write data, write signal and select signal all changing after the rising edge of the clock. The first clock cycle

of the transfer is called the SETUP cycle. After the following clock edge the enable signal **PENABLE** is asserted, and this indicates that the ENABLE cycle is taking place. The address, data and control signals all remain valid throughout the ENABLE cycle. The transfer completes at the end of this cycle. The enable signal, **PENABLE**, will be de-asserted at the end of the transfer. The select signal will also go LOW, unless the transfer is to be immediately followed by another transfer to the same peripheral. In order to reduce power consumption the address signal and the write signal will not change after a transfer until the next access occurs. The protocol only requires a clean transition on the enable signal. It is possible that in the case of back to back transfers the select and write signals may glitch.

- **Read Transfer:**

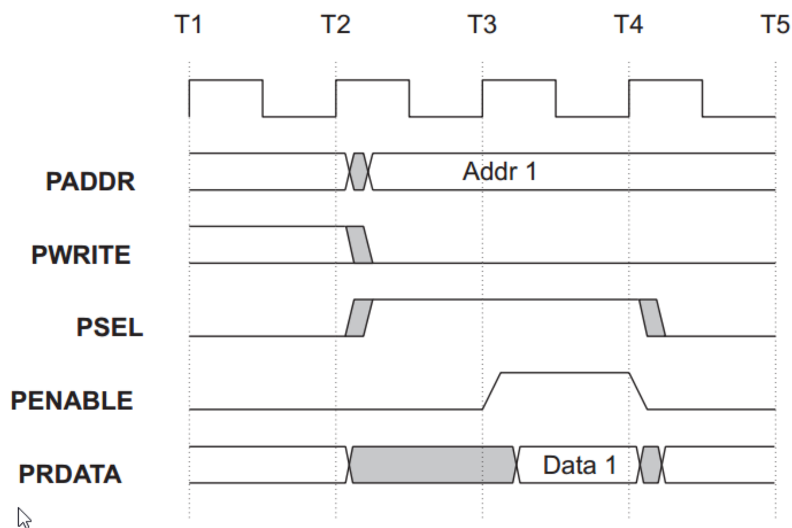


Figure 3-33 DMA Read Transfer

The timing of the address, write, select and strobe signals are all the same as for the write transfer. In the case of a read, the slave must provide the data during the ENABLE cycle. The data is sampled on the rising edge of clock at the end of the ENABLE cycle.

3.8.4 Events and Timing Diagram

1. Single and burst request:

DMA request timing when some peripheral requests a single and a burst transfer.

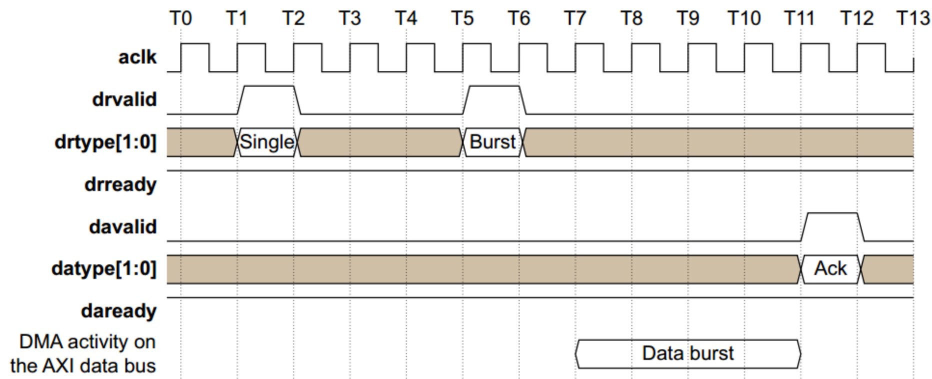


Figure 3-34 Single and Burst request timing diagram

T1	The DMAC detects a request for a single transfer.
T3	The DMAC ignores the single transfer request because the DMA channel thread had executed a DMAWFP burst instruction.
T5	The DMAC detects a request for a burst transfer.
T7 – T10	The DMAC performs a burst transfer.
T1	The DMAC sets **davalid** HIGH and sets **datype[1:0]** to indicate that the burst transfer is complete.

2. DMAC performs single transfers for a burst request:

DMA request timing when some peripheral requests a burst transfer, but the DMAC has insufficient data remaining in the MFIFO to generate a burst and therefore completes the request using single transfers.

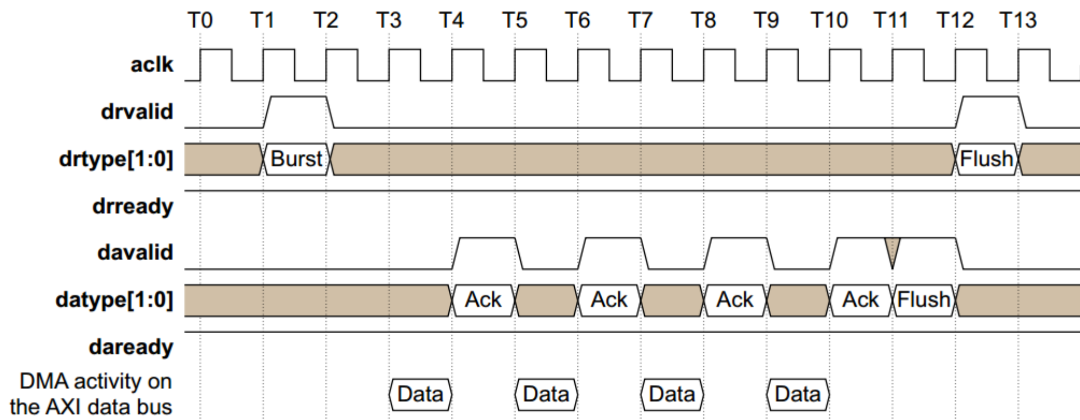


Figure 3-35 Burst request timing diagram

T1	The DMAC detects a request for a burst transfer.
T3	The MFIFO contains insufficient data for the DMAC to generate a burst transfer and therefore, the DMAC performs a single transfer.
T4	The DMAC signals **davalid** and **datype[1:0]** to indicate completion of a single transfer.
T5 – T10	The DMAC performs the remaining three single transfers.
T11	The DMAC signals **davalid** and **datype[1:0]** to request the peripheral to flush the contents of any control registers that are associated with the current DMA cycle.
T12	The peripheral signals **drvalid** and **drtype[1:0]** to acknowledge the flush request.

Peripheral length management

The peripheral request interface enables a peripheral to control the quantity of data that a DMA cycle contains, without the DMAC being aware of how many data transfers it contains. The peripheral controls the DMA cycle by using: **drtype[1:0]** to select a single or burst transfer. **drlast** to notify the DMAC when it commences the final request in the current series.

DMAC length management

DMAC length management is when the DMAC controls the total amount of data to transfer. The peripheral uses the peripheral request interface to notify the DMAC when it requires the DMAC to transfer data to or from the peripheral. The DMA channel thread controls how the DMAC responds to the peripheral requests.

Events and interrupts

The number of events and interrupts that the DMAC can support is configurable. When the configured number of event-interrupt resources is set then you must program the INTEN Register to control if each event-interrupt resource is either an event or an interrupt. When the DMAC executes a DMASEV instruction it modifies the event-interrupt resource that you specify. If the INTEN register sets the event-interrupt resource to be an:

- **Event:** the DMAC generates an event for the specified event-interrupt resource. When the DMAC executes a DMAWFE instruction for the same event-interrupt resource then it clears the event.
- **Interrupt:** The DMAC sets **irq<event_num>** HIGH, where event_num is the number of the specified event-resource. To clear the interrupt you must write to the INTCLR Register.

Therefore, if you require a DMAC to be able to signal two interrupt requests and generate five events then the DMAC must be configured to support seven event-interrupt resources. In this example, the DMAC provides **seveninterrupt signals**, **irq[6:0]**, and therefore five of these signals are not used.

Watchdog abort

The DMAC can lock up if one or more DMA channel programs are running and the MFIFO is too small to satisfy the storage requirements of the DMA programs.

The DMAC detects a lock up when all of the following conditions occur:

- Load queue is empty.
- Store queue is empty.
- All of the running channels are prevented from executing a DMALD

instruction either because the MFIFO does not have sufficient free space or another channel owns the load-lock. When the DMAC detects a lockup it signals an interrupt.

Abort handling

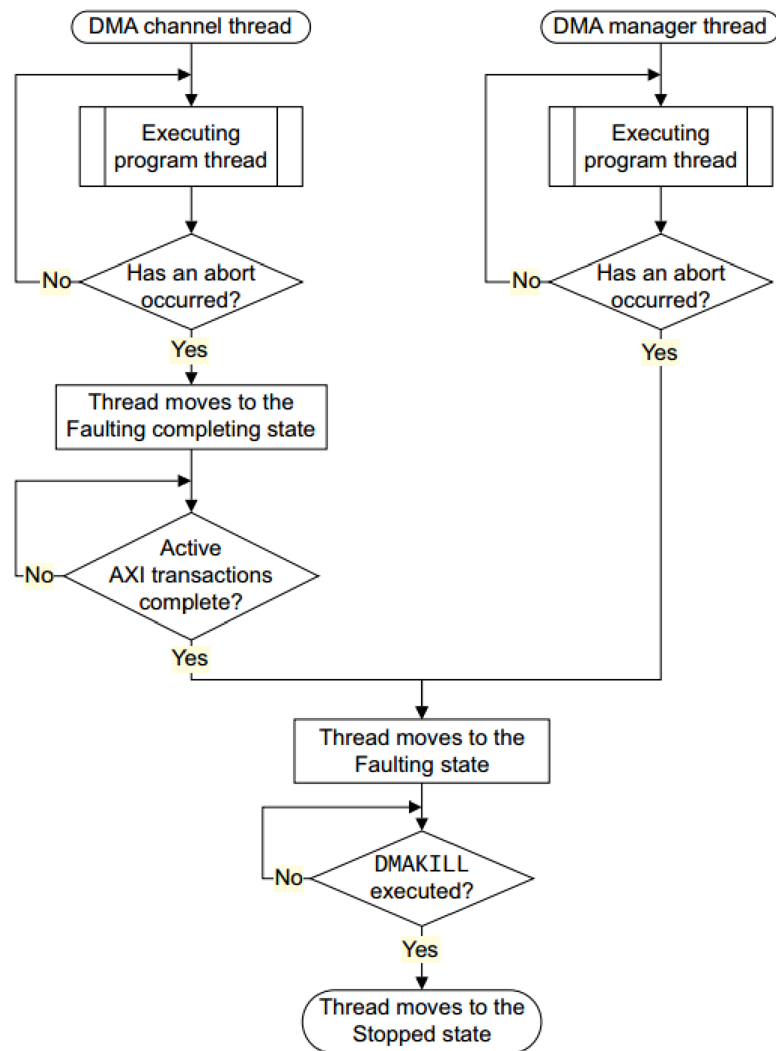


Figure 3-36 DMA Abort handling

Work Steps

At first, APB interface is designed according to the state diagram, write transfer and read transfer mentioned before. It consists only some 3 states which moves between them according to specific signals **PSEL, PREADY** and **PENABLE**.

Secondly, a divider is designed to divide any signal by even and odd integer numbers as the clock of APB interface can be a divided version from **aclk** which is the clock of AXI interface.

There are multiple versions of the divider designed:

1- Even Divider

Timing Diagram:

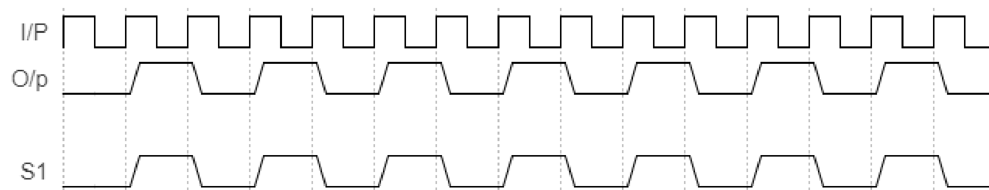


Figure 3-37 Even Divider Timing Diagram

Flow Chart:

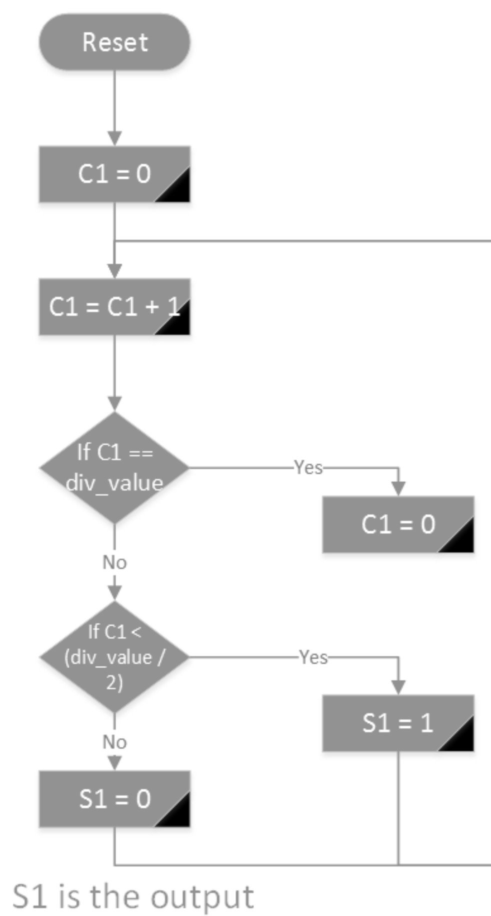


Figure 3-38 Even Diagram Flow Chart

2- Odd Divider

Timing Diagram:

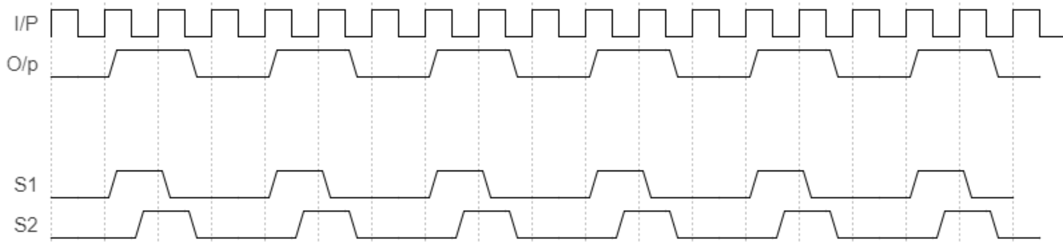
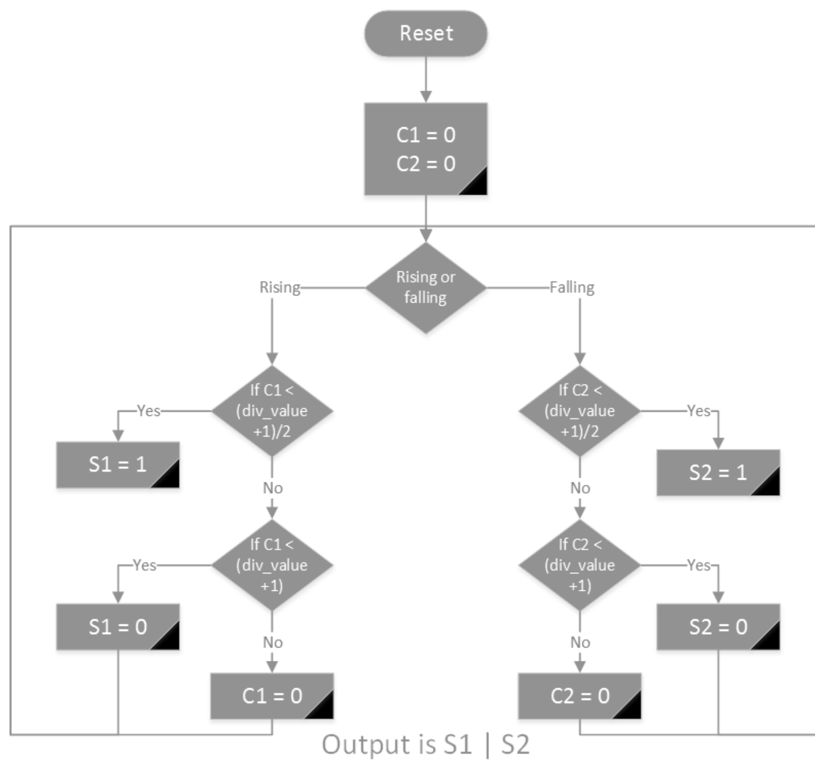


Figure 3-39 Odd Divider Timing Diagram

Flow Chart:



3- Optimized Design

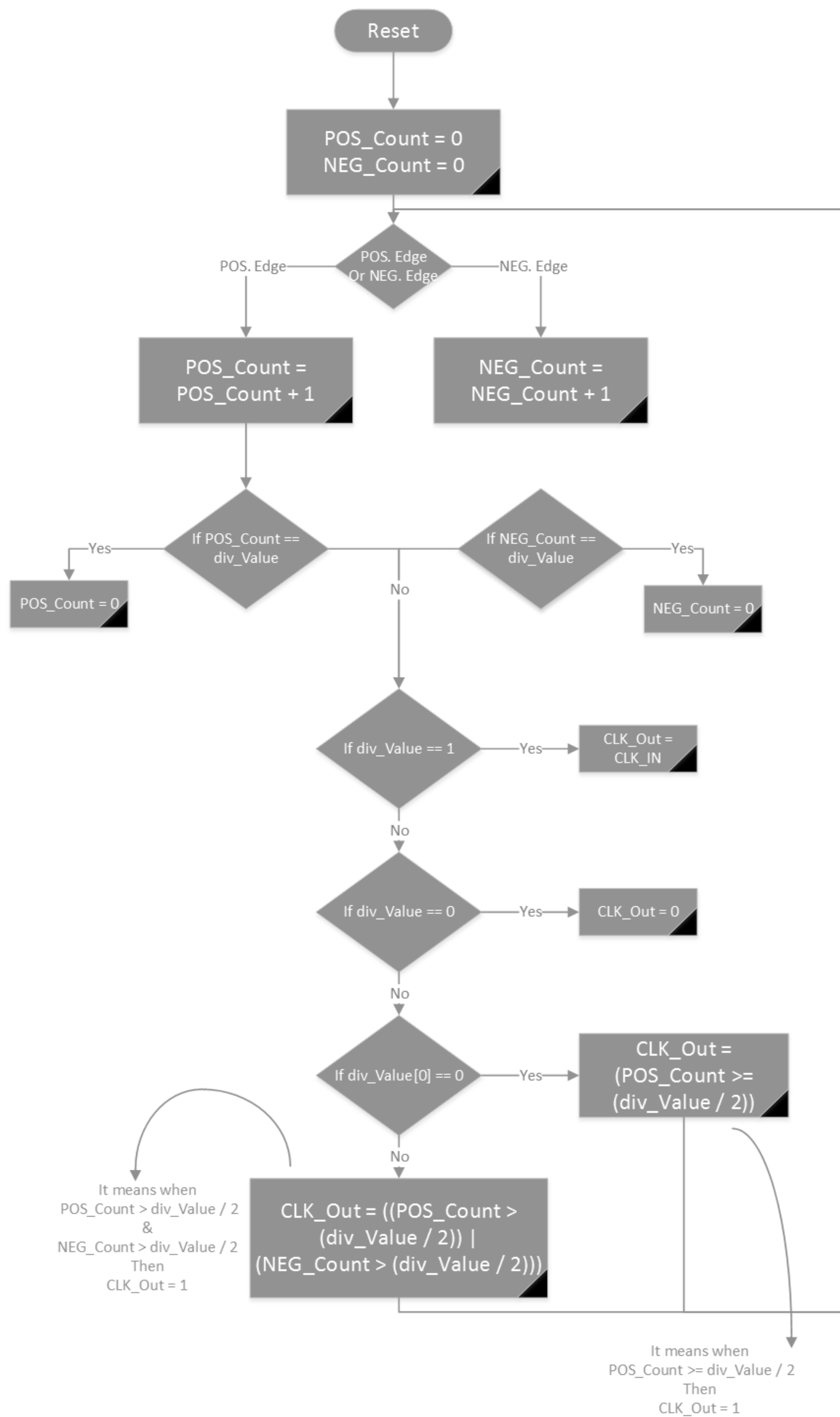


Figure 3-40 Optimized Design Flow Chart

Chapter 4: Simulation, Synthesis, and Integration

4.1 Simulation

4.1.1 GPIO

- **GPIO Pin Configurations**

  Each individual GPIO pin can be configured as input/output. However, bank0 [8:7] pins must be configured as outputs.

- **Writing Data to GPIO Output Pins**

  For GPIO pins configured as outputs, there are two options to program the desired value.

a) Option 1: Read, and update the GPIO pin using the gpio.DATA_0 register.

Example: Configure MIO pin 10 as an output and write 1 to pin 10:

1. **Set the direction as output:** Write 0x0000_0400 to the gpio.DIRM_0 register.
2. **Set the output enable**: Write 0x0000_0400 to the gpio.OEN_0 register.

Note: The output enable has significance only when the GPIO pin is configured as an output.

3. **Write updated value to output pin 10:** Write 0x0000_0400 to the gpio.DATA_0 register.

So, the bank will be directly updated with the new value as shown in Figure 4-1.

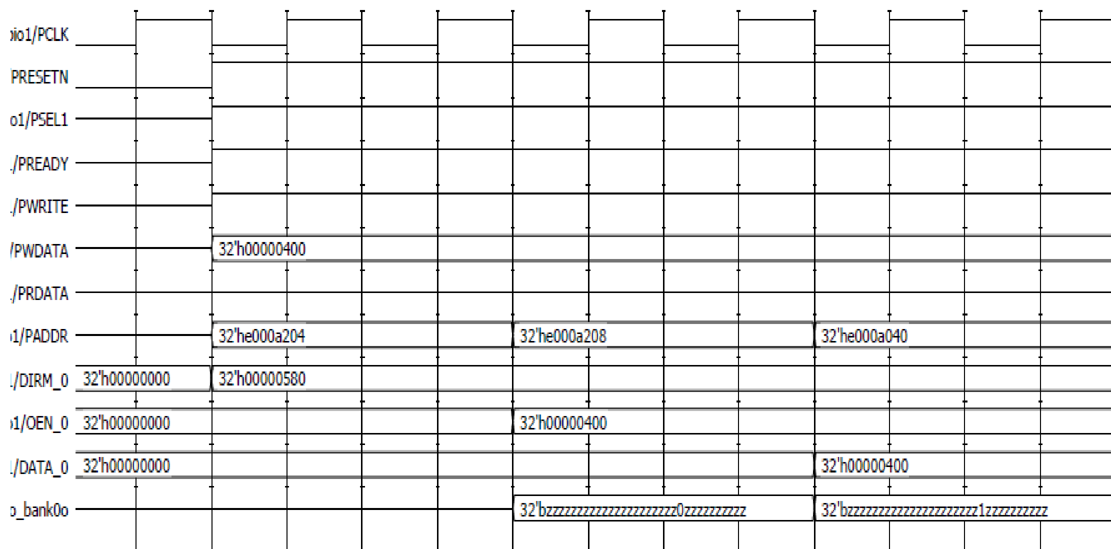


Figure 4-1 GPIO Simulation results 1

b) Option 2: Use the MASK_DATA_x_MSW/LSW registers to update one or more GPIO pins.

Example: Set output pins 20, 25, and 30 to 1 using the MASK_DATA_0_MSW register.

1. **Generate the mask value for pins 20, 25, and 30:** To drive pins 20, 25 and 30, 0xBDEF is the mask value for gpio.MASK_DATA_0_MSW [MASK_0_MSW].
2. **Generate the data value for pins 20, 25, 30:** To drive 1 on pins 20, 25, and 30, 0x4210 is the data value for gpio.MASK_DATA_0_MSW [DATA_0_MSW].
3. **Write the mask and data to the MASK_DATA_x_MSW register:** Write 0xBDEF_4210 to the gpio.MASK_DATA_0_MSW register.

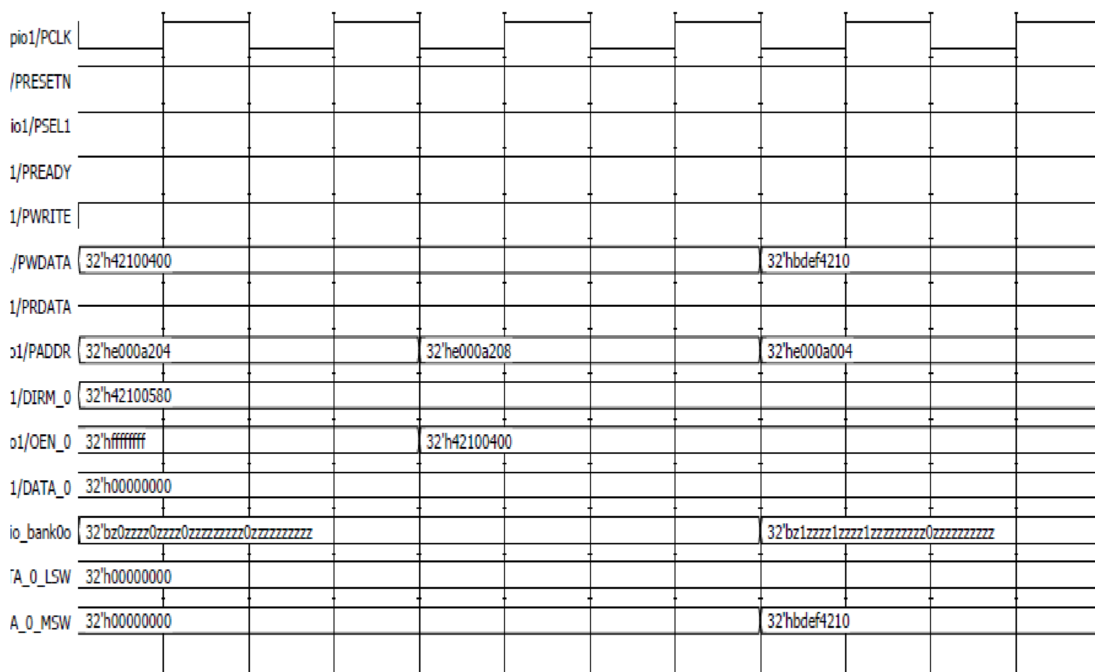


Figure 4-2 GPIO Simulation results 2

- **Reading Data from GPIO Input Pins**

For GPIO pins configured as inputs, there are two options to monitor the input.

Option 1: Use the gpio.DATA_RO_x register of each bank.

Example: Read the state of all GPIO input pins in bank 0 using the DATA_RO_0 register.

1. **Read Input Bank 0:** Read the gpio.DATA_RO_0 register.

   So here what is written in input bank will directly update DATA_RO_0 which can be easily read through APB Bus.

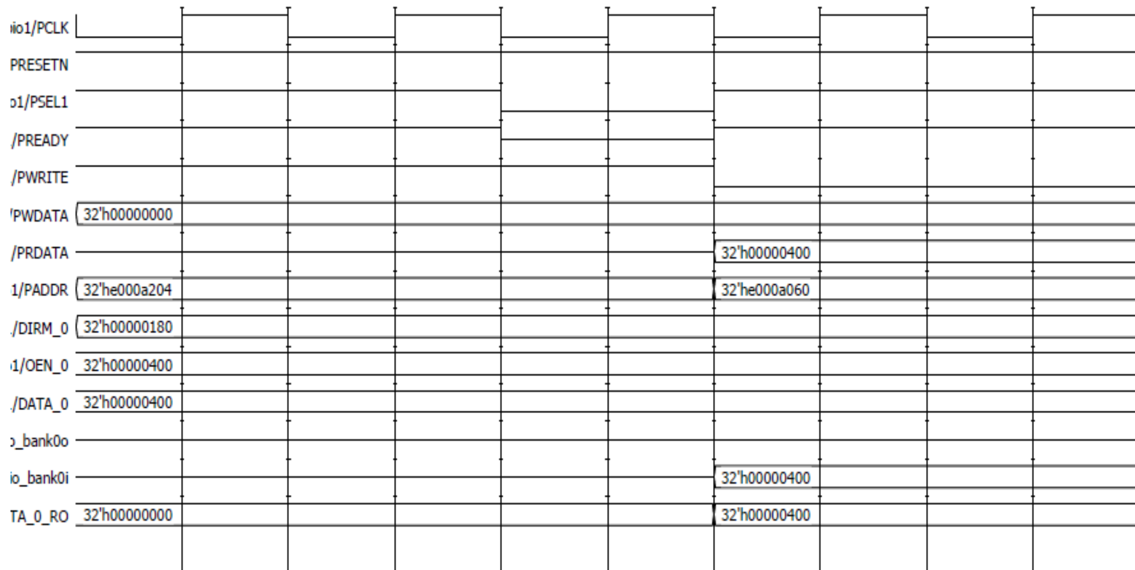


Figure 4-3 GPIO Simulation results 3

Option 2: Use interrupt logic on input pins

Example: Configure MIO pin 12 to be triggered as rising edge.

1. **Set the trigger as a rising edge:** Write 1 to gpio.INT_TYPE_0 [12]. Write 1 to gpio.INT_POLARITY_0 [12]. Write 0 to gpio.INT_ANY_0 [12].

2. **Enable interrupt:** Write 1 to gpio.INT_EN_0 [12].

3. **Status of Input pin:** gpio.INT_STAT_0 [12] =1 implies that an interrupt event occurred.

4. **Disable interrupt**: Write 1 to gpio.INT_DIS_0 [12].

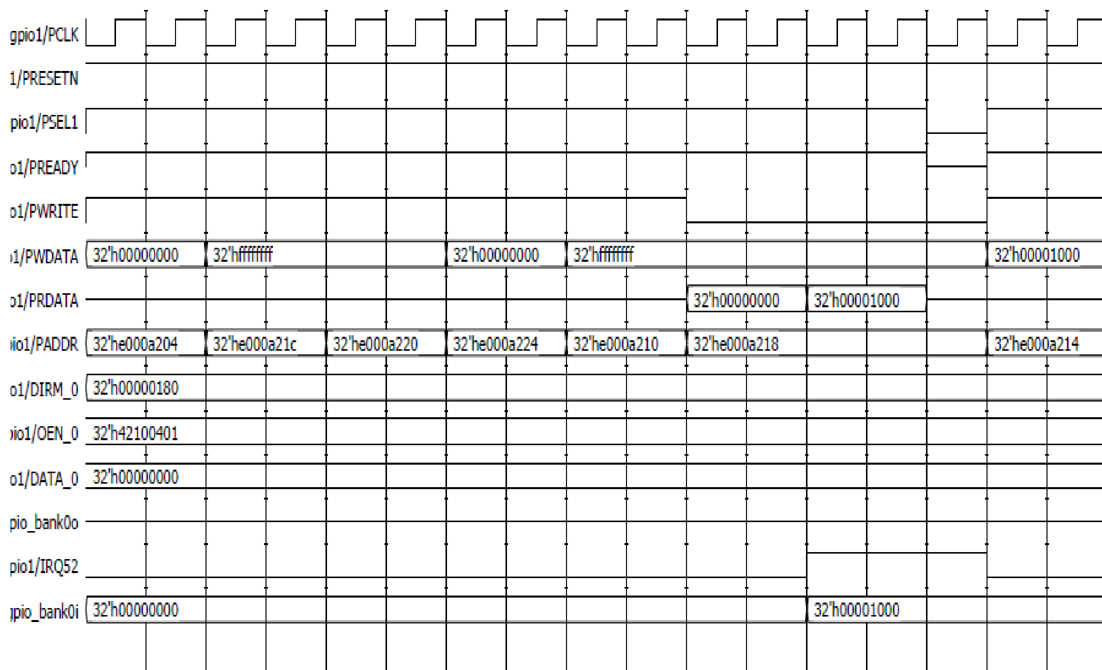


Figure 4-4 GPIO Simulation results 4

4.1.2 GIC

Considering all the last functional description in **Chapter 3**, we managed to build the RTL model of the GIC and getting out the correct simulation results and matching it with the TLM model successfully as shown in Figure 4-5.

The case shown in the simulation is: interrupt signal is received from the UART then another one from the SPI. The GIC managed to handle both signals and send Interrupt signal to the targeted CPU.

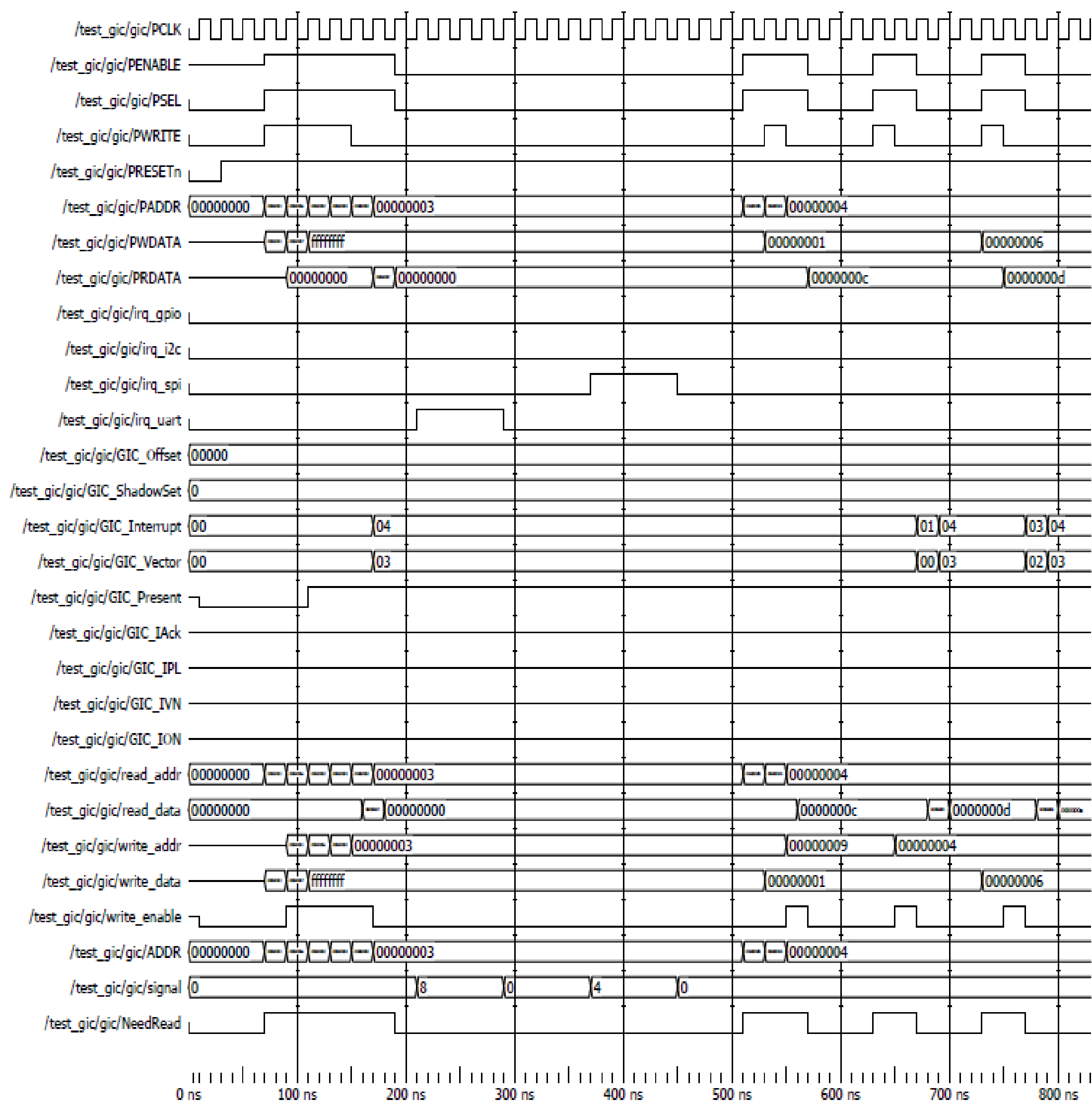


Figure 4-5 GIC Simulation Results

4.1.3 UART

Many cases were simulated to validate that the UART functionality is working according to the specifications. The following case is sending 16 bytes of data at the normal mode at the transmitter.

The interrupt, passed to the GIC, is raised to indicate the end of the simulation. Figure 4-6 shows the bytes inside the FIFO. Figure 4-7 shows that the data is totally sent and interrupt is raised announcing the end of the simulation.

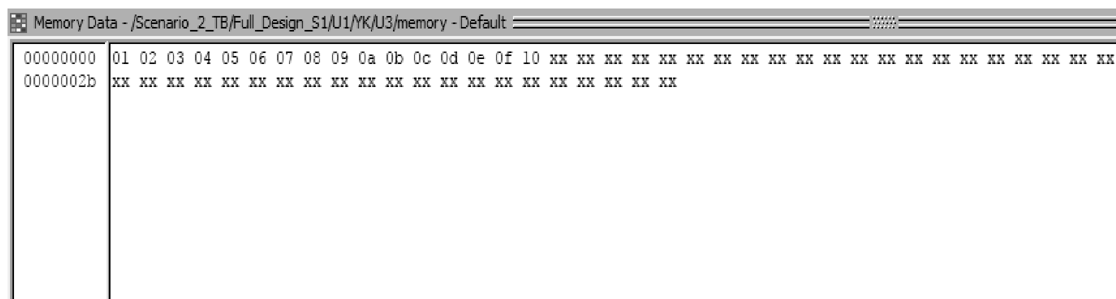


Figure 4-6 FIFO containing the 16 bytes to be sent

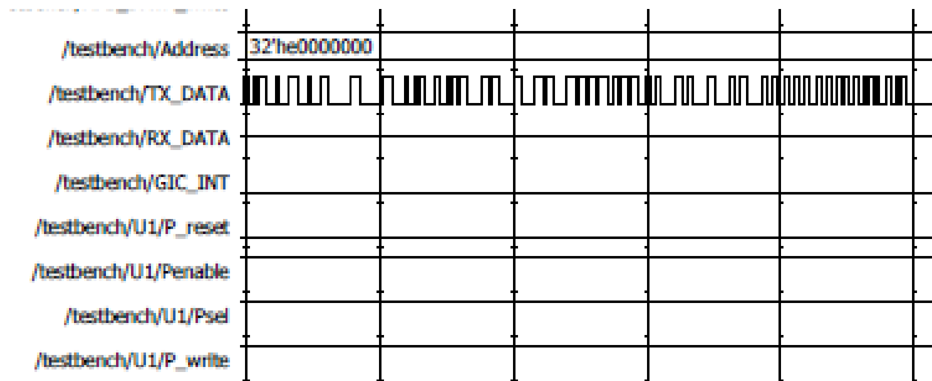


Figure 4-7 interrupt is raised and 0the data is totally sent

In the design, we have three divisions to be taken into consideration the first is dividing by eight, the second is dividing up to 65535 and the third is dividing up to 255. The number of bytes is up to 64 bytes as shown in Table 7.

# of bytes to be sent	First division	Second division	Third division	Simulation time (sec.)
8	✓	F	FF	15.23
8	✓	FF	FF	263.59
16	-	3FF	FF	454.88
2	✓	FFFF	FF	1670

Table 7 UART Scenarios

4.1.4 SPI

In Figure 4-8, a simulation of master mode is illustrated with a clock division equals four, clock phase=0 and a clock polarity=1this is obvious in (sck) signal as the duration of the (sck) is four times the reference clock (PCLK) also the sck starts with low state as clock polarity=1 and this combination of clock polarity and clock phase values make the driving edge is positive and a negative sampling edge.

Also the flow of the byte from entering the TxFIFO from the PWDATA signal according to the write pointer of the FIFO with the reference clock domain then this data is transmitted out the FIFO according to the write pointer to the master block in (tdata1) signal with the sck domain then each byte is moved into the (treg) that starts to shift the data bit by bit to the (mosi) signal which is an input to the salve the signal (done1) is activated showing that the whole byte is successfully transmitted.

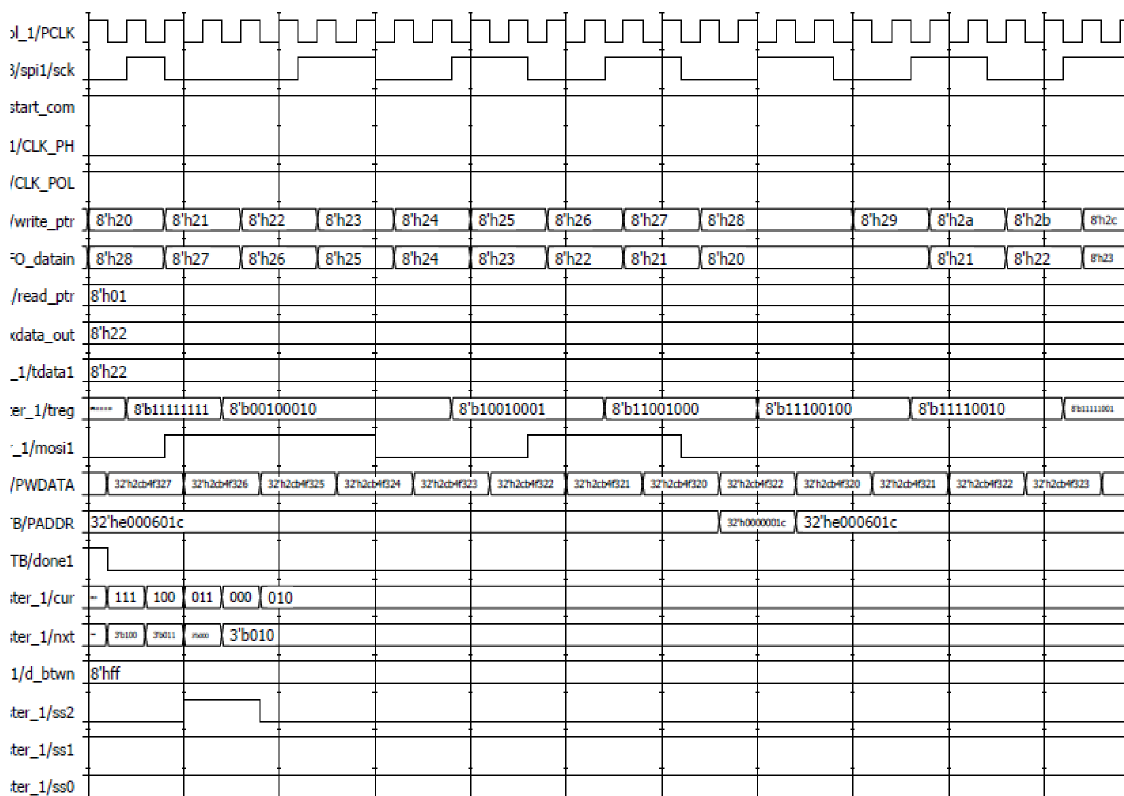


Figure 4-8 SPI Simulation Results

Also in this simulation the slave select of the third slave (ss2) is activated is activated along the time of transmission as it is active low then after the transmission is finished the (ss1) is deactivated (changed to 1) waiting for the new byte to be transmitted, the waiting time between the transmission of the bytes is determined according to the (d_btwn) signal according to the equation stated previously.

Figure 4-9 shows the TxFIFO after the end of simulation where all the FIFO is full of data.

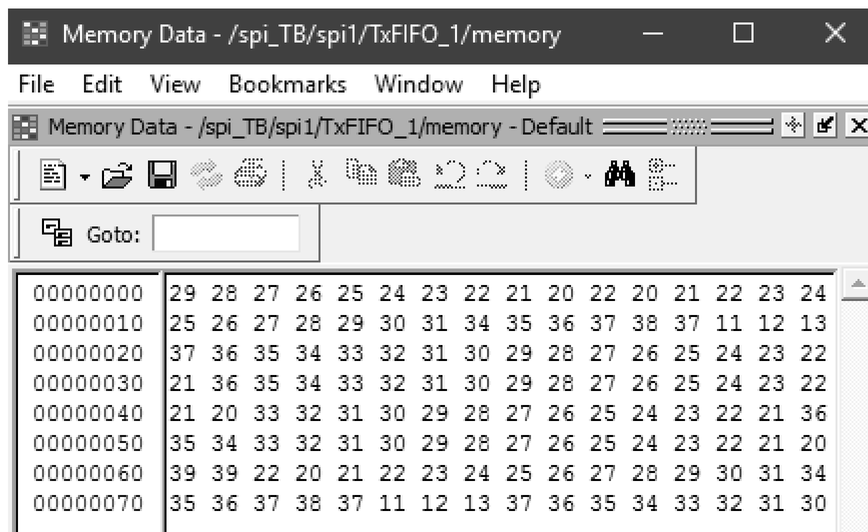


Figure 4-9 SPI Memory results

4.1.5 I2C

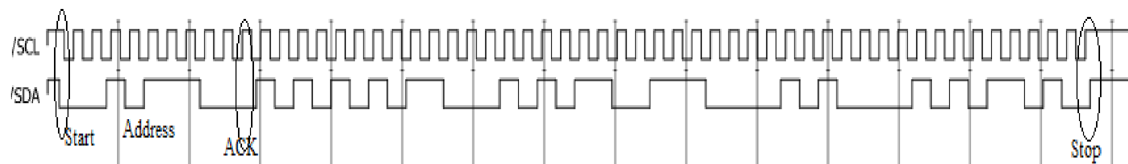


Figure 4-10 I2C Simulation results

Figure 4-10 shows SCL and SDA lines changes for a master transmitter mode, in this waveform 5 bytes were sent. First the master sends a start condition. Then the slave address and the R/W bit. The slave ACKs then the master starts sending 5 bytes. The slave also ACKs after each byte sent by the master. After that the master terminates the transaction by a stop condition.

4.2 Synthesis

Synthesis is Combining primitive logic functions to form a design netlist that meets functional and design goals. Synthesis is a very important step in digital design. It saves a lot of time and efforts. In HDL level complexity is very low it is just a written code. On the other hand, when the design goes through steps to layout it will be very complex. Figure 4-11 shows how complex the design will be.

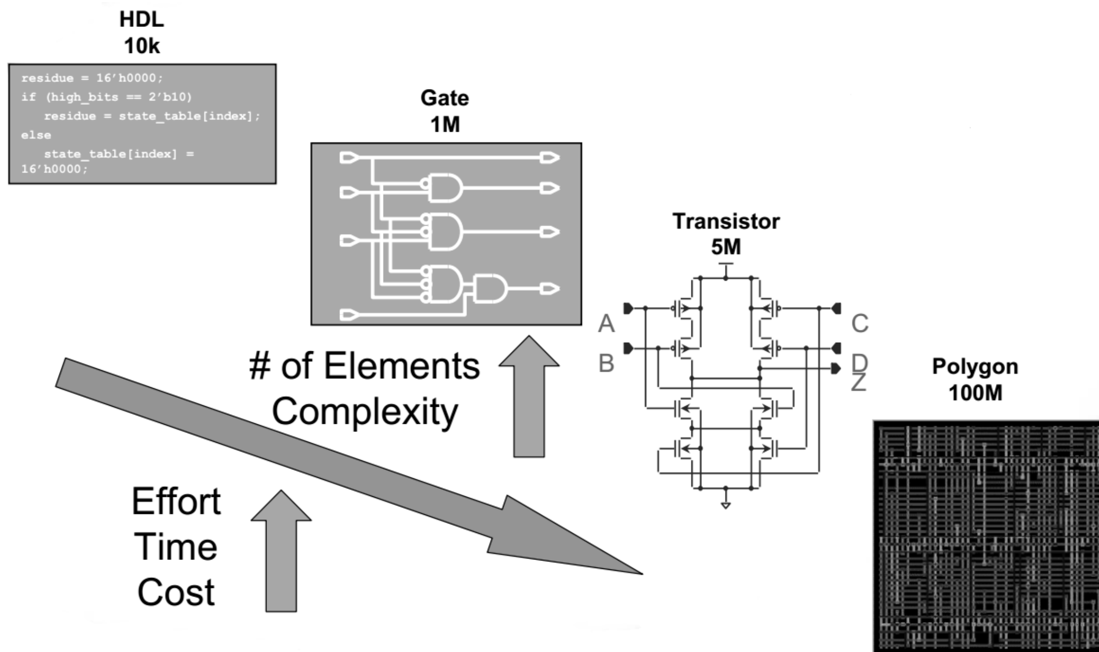


Figure 4-11 Synthesis diagram

4.2.1 Logic Synthesis

It consists, as shown in Figure 4-12, of three main steps:

Logic Synthesis = Translation + Mapping + Optimization.

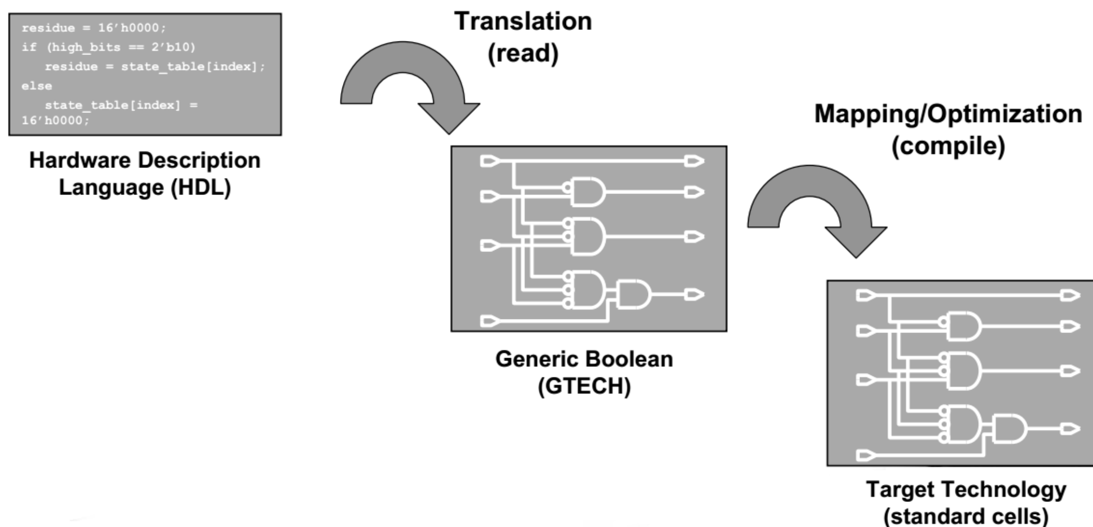


Figure 4-12 Logic Synthesis steps

Hardware Description Language (HDL) can be Verilog or VHDL. The way, the code is written, will affect all results especially in Verilog. Verilog is a synthesizer dependent as each synthesis tool translates the code to a different hardware depending on its implementation. Designers must take care of previous notes when designing a

large system. Register transfer level (RTL) gives optimal results. It makes the design so easy and decrease complexity of the system.

- Translation: It converts HDL to its functional Boolean equivalent
- HDL syntax/rule checks
- Optimizes HDL
- Arithmetic function mapping
- Sequential function mapping
- Combinational function mapping
- Mapping and Optimization:
  1) Maps Boolean functions to technology specific primitive functions.
  2) Modifies mapping to meet design goals:
     - Design Rules.
     - Timing.
     - Area.
     - Power.
- Static Timing Analysis (STA): It breaks the design into sets of signal paths as shown in Figure 4-13. Each path has a start point and an end point:

  - Start points:
    • Input ports.
    • Clock pins of Flip-Flops or registers.
  - End points:
    • Output ports.
    • All input pins of sequential devices (except clock pins).

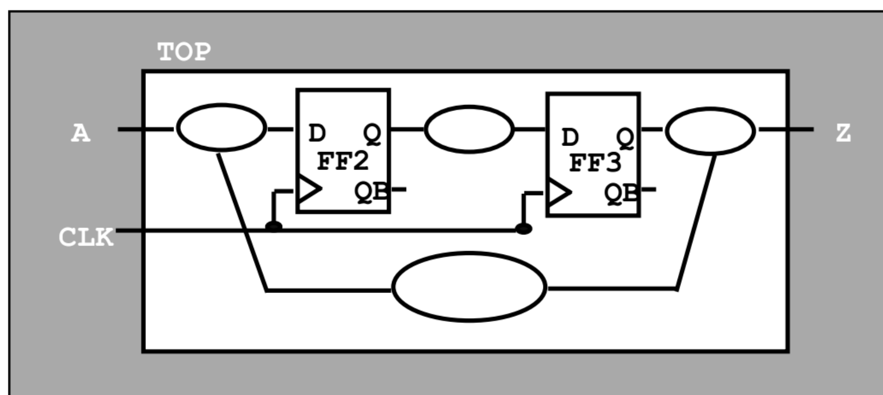


Figure 4-13 Static Timing Analysis

Timing is very important. Determining the critical path and its delay helps in avoiding negative slack. Negative Slack means that the clock period is not enough for the data to reach the output because it is smaller than the critical path. Negative slack is shown in Figure 4-14.

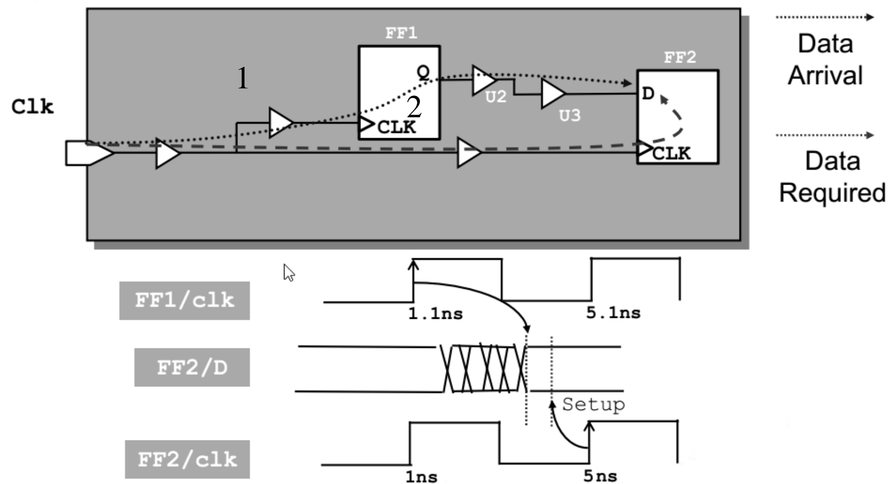


Figure 4-14 Negative slack

4.2.2 Synthesis Problems

1- The main problem is that the same signal is derived in more than always block. The problem is due to the two clock edges will never arrive simultaneously, the synthesizer cannot assume that.

- It is solved by merging the always in only one depending on if conditions which will be translated as a multiplexer as in Figure 4-15.

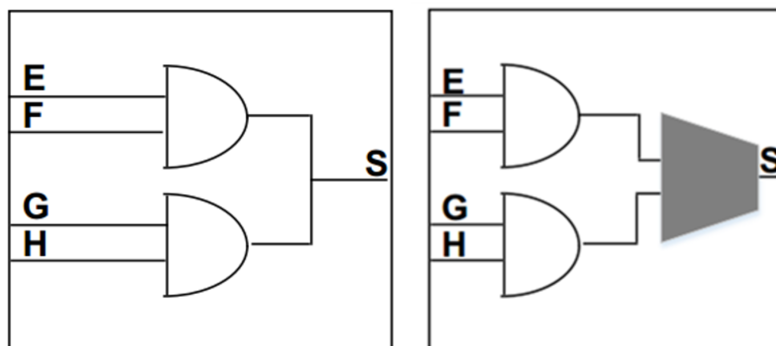


Figure 4-15 Synthesis Problem 1

2- Triggering signal on both edges of the clock**.** When register is assigned in an edge-sensitive always block, it defines a flip-flop. FPGAs do not have flip-flops that can trigger on both edges of a clock. Triggering in both edges is

double-clocked logic, which is rarely supported by synthesis tools since cell/vendor libraries rarely contain these varieties of flops.

- Some designs may employ double-clocked logic; often using custom cells, with a specialized synthesis/timing flow. In order to do that, it is needed to have two separate always blocks, one for each edge of the clock, and then figure out a way to combine the outputs of the two blocks without creating glitches.

- Case statement must have a default value. This may lead to problems in hardware level. It may hang the system due to any error as it will reach a state which is not defined in the code. Designers must take care when designing systems to avoid this problem.

4.3 Integration

Integrating the implemented blocks to build the whole system maybe a hard or an easy task depending on the designs. It will be very hard if each designer builds his system without awareness of what other designers do. It can be emulated to the game which some cubes are put above each other to build a specific construction. If these cubes are not well-organized then the overall structure will tear down. On other hand, if the system is well-organized then the overall structure will be very strong and never tear down as in Figure 4-16.

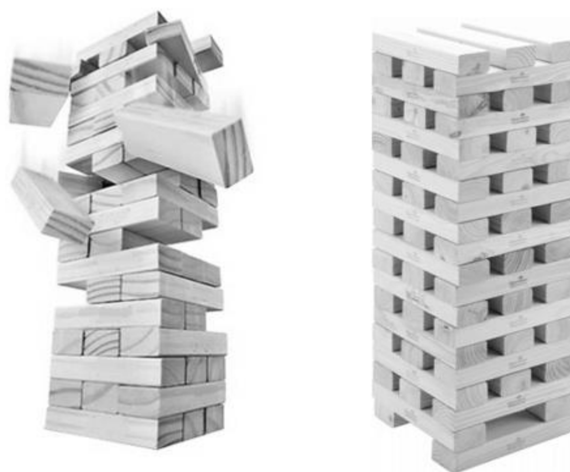


Figure 4-16 Integration

Due to what mentioned before designers must take care during designing the system. They must completely agree with each other from the beginning with specific interfaces which will make the integration so easy.

4.3.1 Zynq Blocks Integration

In this project, interfaces are already specified by the designed board. All blocked are interfaced with APB bus. This bus has specific interfaces which are constant for all blocks. So, integration becomes as easy as in Figure 4-17.

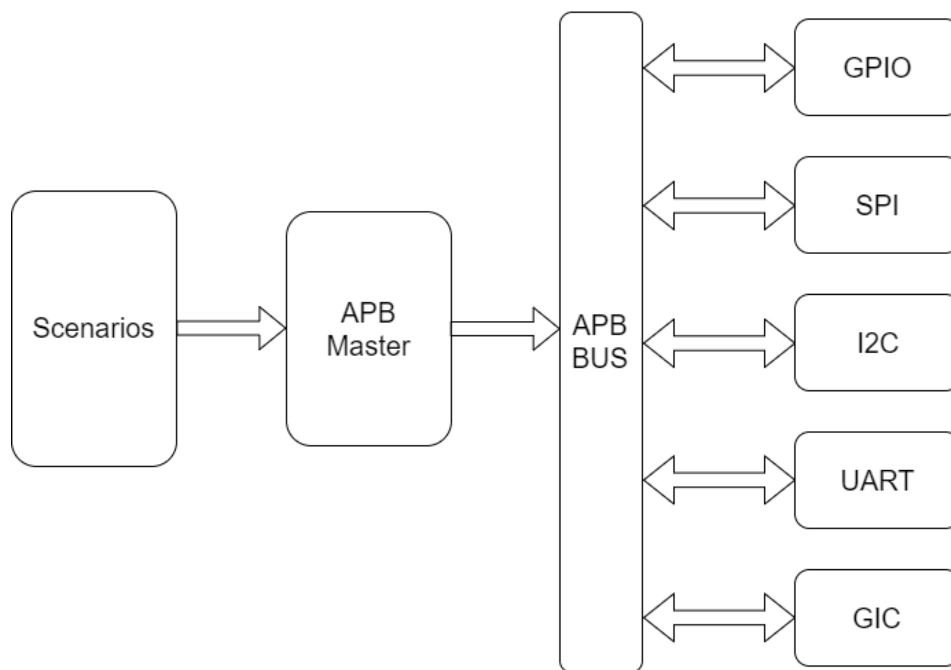


Figure 4-17 Zynq Blocks Integration

Implemented blocks, GIO, SPI, I2C, UART and GIC, are considered as slaves to APB bus. A master must exit to accommodate between blocks' transactions. Depending on address margin the master chooses the required block. APB Master is the designed to do that job. But a problem appeared, there is no processor (CPU) which will give stimulus to these blocks. A rough testbench is implemented to replace the CPU. This testbench consists of some scenarios which will achieve the required target of the project. To achieve this target RTL simulation time must be very large which reaches 80 hours. The dominant block in simulation time is the UART as its clock can be divided million times from its reference value.

4.3.2 Scenarios

Four scenarios are implemented in the testbench to achieve the required results which will be discussed in Chapter 5.

GPIO scenario is fixed in all implemented scenarios. This scenario runs as the next step in each scenario. This scenario is implemented as shown in Figure 4-18.

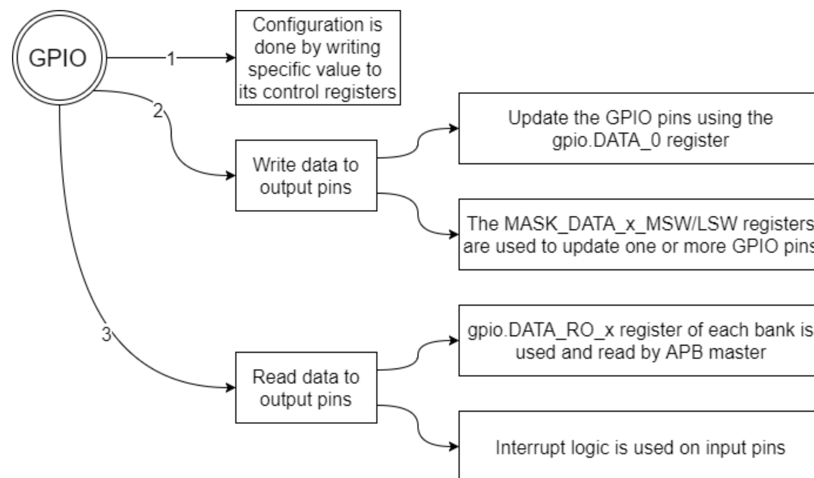


Figure 4-18 GPIO Scenarios

Different scenarios for other blocks these scenarios are shown in Figure 4-19.

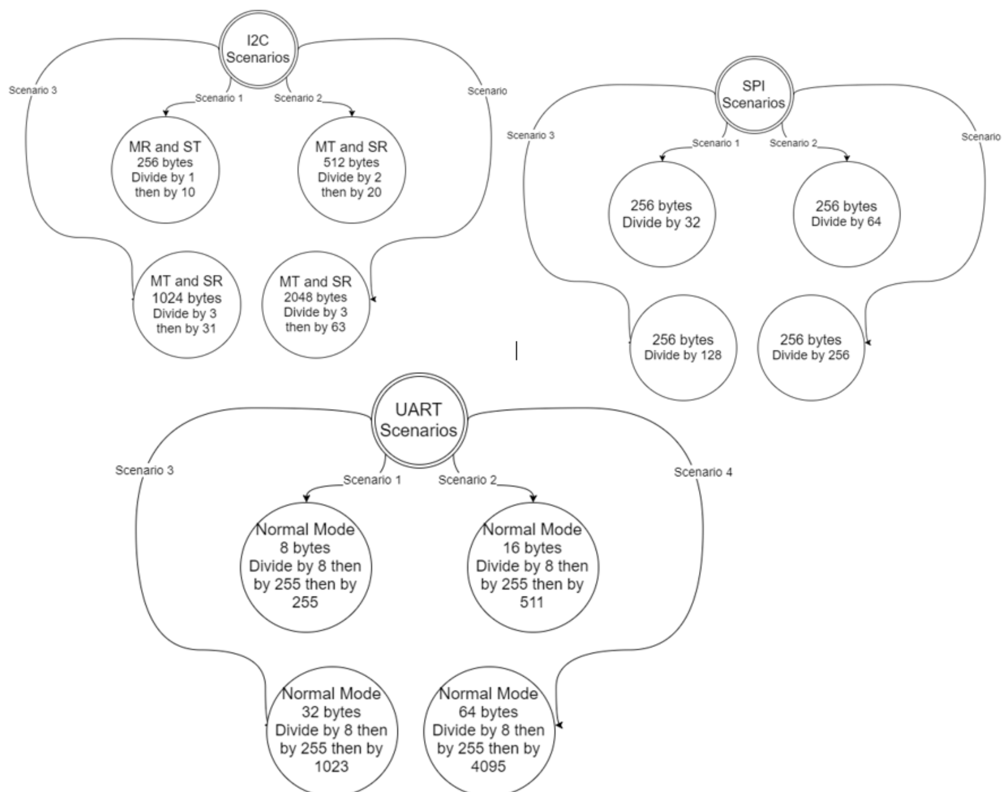


Figure 4-19 I2C, SPI, UART Scenarios

4.3.3 Integration Problems

Nothing becomes perfect from the first iteration so some problems appeared. These problems are as follow:

1) Two blocks access a signal at the same time with different values. Let's take an example, I2C writes (0) as a default value on the PRDATA which is an APB interface. On the other hand, UART writes (10) on the same signal so that hardware will be dispersed and un-known value will appear. This problem is shown in Figure 4-20 x and y. It is solved by making the default value for each block as a high impedance value (Z).

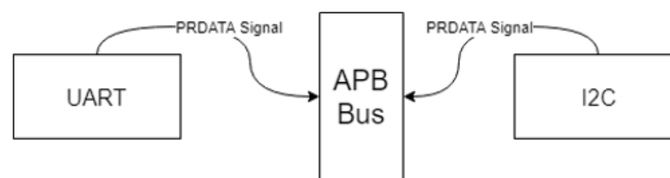


Fig. x

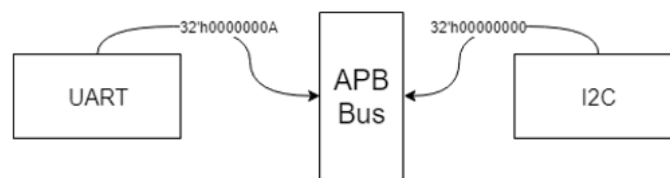


Fig. y

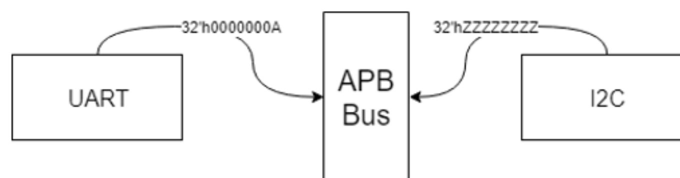


Fig. z

Figure 4-20 Integration Problem 1

2) There is no master on the bus so APB master is implemented.
3) There is no CPU, so a testbench is implemented to give stimulus to the blocks.
4) Interrupts implementation is so hard using Verilog. Interrupt comes out from each block and enter GIC. Some prioritization is done on these interrupts then the highest priority interrupt comes out from the GIC to the processor which will send a respond to the block depending on the interrupt.

This implementation can't be built using Verilog. So, the testbench is implemented using System-Verilog. Some like Interrupt Service Routine (ISR) is implemented. As when interrupt occurs, the block stops executing its main functionality. After processor responds to the interrupt, it executes another function. When the block finishes executing this aside function it returns to its main function again. Interrupts execution flow is described in Figure 4-21.

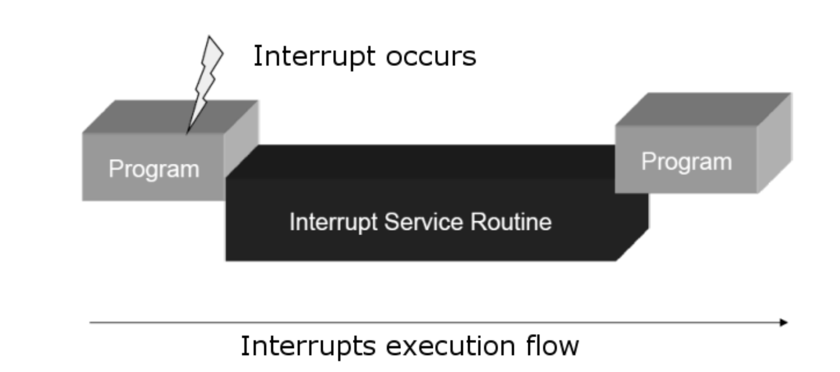


Figure 4-21 Interrupts implementation Problem

5) Synchronization between the blocks. It is solved by adding some delays in the code to achieve synchronization.

Chapter 5: Transaction Level Power Modeling

Power consumption is a key specification in any design. Determining the power of large System-on-Chip (SoC) designs is very computational and time expensive on Register Transfer Level (RTL), if even possible. Power should be addressed in early stages of the design process to prevent long expensive iterations, which hinders releasing a design. Integration of power computation with TLM is gaining interest in both research and industry. TLM for power is needed to overcome any unexpected drawback in the design at early stages. In this paper, estimating power dissipation on TLM is performed in two stages: Characterization and Implementation.

Characterization represents the extraction of power parameters from the existing design. Implementation represents the addition of power model to TLM and the execution of this model to have reliable numbers of power consumption. The flow is introduced in Figure 5-1.

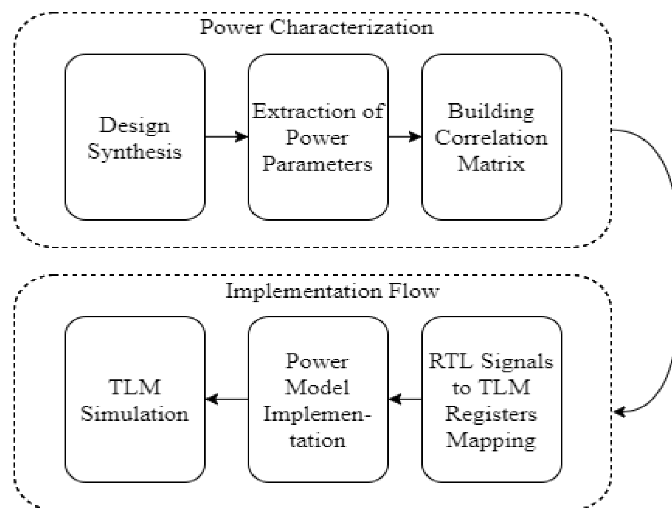


Figure 5-1 Power Characterization flow

5.1 Power Modeling for TLM

Simulating Gate Level (GL) netlist is needed to estimate power. The netlist contains timing and parasitic information about interconnects and gates. On large SoC, simulation on GL netlist to perform power estimation is not feasible. A fast simulation

on TLM is needed. Bottom-up approach is adopted in this paper to perform power estimation of SoC on TLM. A power model on TLM is created for different units/components. Those components are integrated to create SoC. This allows estimating power of large designs in reasonable simulation time.

Building power model on TLM needs power characterization of the design. Power characterization is the extraction of the design power parameters, which enclose the exerted energy by different elements in the design.

To perform power characterization," Power Analysis" flow [10] is adopted. This flow is normally used for power calculation on RTL designs using the following steps:

1) Generation of signals switching activity: RTL simulation is performed to obtain switching activity of every signal in the design. This activity is saved in Switching Activity Interchange Format (SAIF) file, which encloses toggling information of signals.

2) Design Synthesis: RTL design is synthesized into netlist, which contains complete information of different GL components.

3) Power Calculation: The SAIF file is attached to the synthesized netlist. Total power of the design is calculated for the attached switching activity.

5.2 TLPM Methodology

In TLPM methodology, Power Analysis flow is manipulated to suit power characterization on TLM. This is achieved by calculating exerted energy by every design element, which has its own contribution for the total power. Using superposition concept, the total power is divided among its contributors of the design elements. In the implementation phase, those elements are mapped to the corresponding registers in TLM side. Then, a comprehensive estimation is created for the total power. In order to achieve power characterization, the following steps are performed. The steps are illustrated in Figure 5-2 as follows...

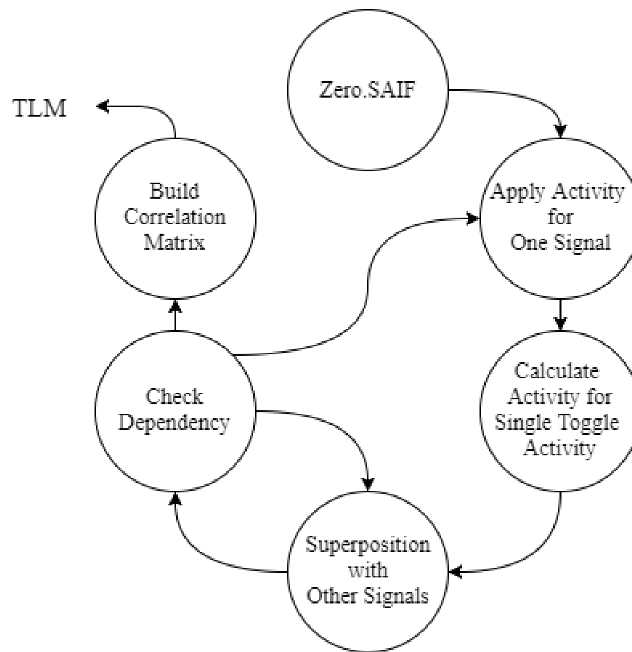


Figure 5-2 Power Characterization in TLPM

1) Create Zero.SAIF file, this file contains Zero activity information for all RTL signals.

2) Adopt superposition theory for the elements. Enable the activity of only one element at a time, to evaluate contribution of each element in total power.

3) Apply switching activity on a single element in Zero.SAIF file. Power information of every signal is evaluated in RTL side, and energy of a single toggle count is determined.

4) In order to overcome dependencies among signals, activity of signals is added incrementally into Zero.SAIF file. The dependency scheme among RTL signals is completed upon performing this step for all signals.

5) The dependency scheme is divided into three correlation factors:

   a) Zero correlation factor: The signal has its own distinct contribution in total dynamic power, and no other signal shares its contribution in total power value. Clock tree signals are good examples of those.

   b) One correlation factor (total dependency): In this case, the signals have same contribution with respect to power. They are grouped in one bundle. Signals on same data path match this category.

c) Non-Zero, Non-One correlation factor: In this case, the signals have effect on other signals. Superposition cannot be applied and conditional correlation is applied.

6) The dependency scheme between signals constitutes the" Correlation Matrix" of the design. Correlation matrix is used to build the power model on TLM.

5.3 Correlation Matrix

This section elaborates the mapping mechanism of the correlation matrix of RTL signals to the TLM registers. Power model development is also described. The implementation phase of TLPM methodology is performed in two steps:

A. Signal to Register Mapping

TLM of a design contains the design registers. Registers are accessed with read/write operations. Programmer's View/Timing (PVT) models are used here for design units where PV represents the functional behavior of the device [15]. For power estimation, registers need to be mapped to the relevant power contributors in RTL side. This is performed according to the correlation matrix that has been extracted earlier. For signal to register mapping scheme, signals tracking technique is adopted. The technique is performed by tracking the drivers and the receivers of signals to the effective design ports. The debugging utilities as in [11] facilitate the tracking technique.

B. Power Evaluation

Signals in the design can be categorized as:

1) Signals exist in RTL and do exist in TLM side. These are represented in registers and ports. For these signals, power evaluation is done on read/write operations of the registers. This is performed using the initiator or debug ports as mentioned later.

2) Signals exist in RTL, but implicitly implemented in TLM side. Clock signal is example for that. There are signals/nets for clocks in RTL, but there is no explicit registers or ports on TLM side.

3) Signals exist in RTL, with no implementation at all in TLM side. Signals of power management components or internally generated control signals are examples for those. Those signals are not used during the normal operation of TLM. But, TLPM methodology considers their activities.

For signals with no implementation or with implicit implementation in TLM, additional function is implemented to evaluate power of those signals according to their contribution factors. These factors are extracted from the correlation matrix which is generated from power characterization step.

There are two types of ports in TLM: Initiator ports and Debug ports. Both types are used in TLPM methodology to evaluate the power of the signals.

1) Initiator Ports: These represent the real behavior of the Model. They are the interface between the design unit and SoC. The communication between TLM of the device and SoC is done through callback functions. There are callback functions for every read/write operation of each register. To build the power model, power equations are added to these callback functions to evaluate the power upon every read/write operation. After annotating the RTL contributors to the model, the power model is built using the following steps:

   (a) Track the activity of TLM registers.

   (b) Calculate the exerted energy by every operation.

   (c) Accumulate the energy result to the previously calculated values.

   As simulation time advances, the model gathers all exerted energy across the design.

2) Debug Ports: They represent the back-door access to the TLM of the design where the user can access information of all registers without interfering the blocking transport calls on initiator port. Power model could be added to debug ports.

However, they need explicit calls in the stimulus function beside the initiator ports. TLPM methodology uses both initiator and debug ports in power estimation. Initiator ports track activity of one register at a time. However, multiple registers need to be tracked at same instant to estimate power accurately. The debug ports are needed along with the initiator ports to track multiple registers.

5.4 Vista

Simulation of TLM ZYNQ-7000 model has passed through a sequence of phases. First phase a TLM model is needed which contains the five blocks (I2C, UART, SPI, GPIO, GIC) with the same functionality of the RTL blocks built before, then generating a generic CPU by using VISTA to generate a stimulus that models the same test bench generated in RTL with the same scenarios in order to have a fair comparison between TLM and RTL including the simulation time, the error in power estimation.

During the simulation of the TLM model many difficulties were faced as some features in the model were different from the RTL so some modifications in the RTL code were made to get all the required features, another problem that need some modifications in the RTL that when simulating the blocks that contains FIFOs after all the bytes in the FIFO is sent the TLM model cannot wait for the interrupts to refill the FIFO as the TLM generate the interrupts after the whole transaction is finished unlike the RTL which can generate the interrupts during transactions, so larger FIFOs are needed in the RTL to have the size of the whole number of bytes needed to be transmitted .

Then moving to the next phase of power estimation mapping from RTL to TLM to apply the methodology explained previously that maps the energy generated from all the signals in the RTL block to the full integrated TLM blocks in order to simulate the full integrated design in a short time with almost the same accuracy in power estimation from RTL while the same stimulus may take a long time or crashes while simulating the RTL.

The power model in the TLM model is generated by adding some functions to the main TLM stimulus, first function to count the number of toggles in each signal by comparing the current value of the signal with previous value then it multiply the

number of toggles to the energy of each signal acquired from the RTL using the design compiler then this function calls another function which adds the power of all the signals to obtain the total power and the following are the codes of the two SystemC functions .

First a Zero.SAIF file was generated for each block. In this file there is no switching activity for any signal. Then this file was used with the netlist to make sure the dynamic power is zero. Then single toggle was added to every signal in the design individually (all other signals have zero switching activity). The power obtained for each signal was saved. After this the dependency between signals was checked by adding activity to more than one signal at the same time and check if the resultant power is the addition of the two numbers obtained from each signal individually. It was noted that most of the signal are independent except for the signals on the same path or that depend on each other in operation such as APB signals. For the dependent signals, their contribution was modeled by adding switching activity to all of them at once and calculating their total energy.

For the linear registers (registers that the power of their bits adds up), switching activity was added to all the pins of the register. Then the power was calculated and averaged to obtain the energy per unit toggle per pin for each register. Then in Vista, a function was added to the model to calculate the number of toggles when a register value changes. Then the number of toggles was multiplied by the energy per unit toggle per bin to get the energy exerted by this register.

For the Blocks that their operation totally depends on the clock, most of their power came from the clock. As the clock signal always toggles at a constant rate, its power was modeled as a constant number that is added to the total power calculated from the power obtained from registers modeling.

For the blocks that contain clock division, the counters of the dividers contribute significant power. However their power is not constant as the clock power. Also the values of these counters can't be obtained in the TLM model as TLM doesn't have clock dividers. It was noted that the contribution of the clock dividers to the power mainly depends on the value by which the clock is divided. Several trials were done for different values of the clock divisor and a relation was obtained.

Scripts were written to automate saif file modification during power characterization. TCL was used to interface easily with design compiler. Perl was used for files modification.

A function was added to accumulate the total energy exerted during simulation. And calculate the total power at the end.

Chapter 6: Results and Conclusion

6.1 Results of each block

This section aims to show the effective parameters in each block. Due to change in these parameters, power is changed. The main parameter which affects the power is the clock division.

1) GPIO: It has only fixed scenario which was discussed previously in chapter 4. Power computation results are shown in Table 8.

RTL(uW)	TLM(uW)	Error%
258.0196	253.230344	1.85%

Table 8 GPIO results

2) SPI: Experiments were developed along changes in clock division. Power is calculated in RTL and TLM to see how much the power of the TLM model matches that of the RTL design. Some iterations are built before reached this TLM model to decrease the error. Power computation results are shown in Table 9.

Clock division	RTL(uW)	TLM(uW)	Error%
4	1.1451	1.298	1.33
8	1.0247	1.0215	0.312
16	0.9404	0.9408	-0.0398
32	0.8902	0.8923	-0.231
64	0.8568	0.8637	-0.803
128	0.8394	0.8456	-0.736
256	0.8303	0.8378	-0.896

Table 9 SPI results

3) I2C: To test its power model, two modules of I2C were used. One was configured as a master and the other was configured as a slave. Table 10 shows the power obtained from both RTL and TLM for different configuration.

Clock Division	Number of Bytes	RTL(uW)	TLM(uW)	Error%
144	10	177.021	176	0.58
3	50	179.296	176.8	1.4
1	30	182.5	178.5	2.2

Table 10 I2C results

4) UART: For its module, a normal mode for transmitter was tested with different configurations through changing number of bytes and the clock division as shown in Table 11.

Clock Division	Number of Bytes	RTL(uW)	TLM(uW)	Error%
8	15	142.439	143	-0.39
15	15	143.931	144.4	-0.33
255	5	143.182	143.1	0.57

The above results show that validation in RTL is very difficult or sometimes impossible. Instead, TLM provide faster validation with approximate results in power. TLM models can be enhanced to reduce error. In the previous section, power is calculated for each block and error between RTL and TLM is

obtained. Some scenarios are developed to prove that TLM is the better choice for companies in faster validation. Manual comparison is used between two models to validate the functionality.

6.2 Overall System results

GPIO scenario is used in all coming scenarios but it will be mentioned here only one time as it is fixed. Configuration of MIO pins is done by storing specific values in GPIO registers. To write data to output pins two options are used.

Four scenarios are applied as follows:

1) Scenario A:

SPI master is configured to transmit 256 bytes and slave is configured to receive these bytes. APB clock is divided by 32. I2C is set to master receiver and slave transmitter mode. Slave is configured to transmit 256 bytes. Clock division is done on 2 stages.

- Stage 1: APB clock is divided by 1.
- Stage 2: stage 1 output clock is divided by 10.

So, the clock is totally divided by 10. UART is configured in normal mode. It is configured to transmit 8 bytes. Clock division is done into 3 stages.

- Stage 1: UART ref clock is divided by 8.
- Stage 2: stage 1 output is then divided by 255.
- Stage 3: stage 2 output is then divided by 255.

So, the clock is totally divided by 520200.

2) Scenario B:

SPI master is configured to transmit 256 bytes and slave is configured to receive these bytes. APB clock is divided by 64. I2C is set to master transmitter and slave receiver mode. Master is configured to transmit 512 bytes. Clock division is done on 2 stages.

- Stage 1: APB clock is divided by 2.
- Stage 2: stage 1 output clock is divided by 20.

So, the clock is totally divided by 40. UART is configured in normal mode. It is configured to transmit 16 bytes. Clock division is done into 3 stages.

- Stage 1: UART ref clock is divided by 8.
- Stage 2: stage 1 output is then divided by 255.
- Stage 3: stage 2 output is then divided by 511.

So, the clock is totally divided by 260610.

3) Scenario C:

SPI master is configured to transmit 256 bytes and slave is configured to receive these bytes. APB clock is divided by 128. I2C is set to master transmitter and slave receiver mode. Master is configured to transmit 1024 bytes. Clock division is done on 2 stages.

- Stage 1: APB clock is divided by 3.
- Stage 2: stage 1 output clock is divided by 31.

So, the clock is totally divided by 93. UART is configured in normal mode. It is configured to transmit 32 bytes. Clock division is done into 3 stages.

- Stage1: UART ref clock is divided by 8.
- Stage 2: stage 1 output is then divided by 255.
- Stage 3: stage 2 output is then divided by 1023.

So, the clock is totally divided by 2086920.

4) Scenario D:

SPI master is configured to transmit 256 bytes and slave is configured to receive these bytes. APB clock is divided by 256. I2C is set to master transmitter and slave receiver mode. Master is configured to transmit 2048 bytes. Clock division is done on 2 stages.

- Stage 1: APB clock is divided by 3.
- Stage 2: stage 1 output clock is divided by 63.

So, the clock is totally divided by 189. UART is configured in normal mode. It is configured to transmit 64 bytes. Clock division is done into 3 stages.

- Stage 1: UART ref clock is divided by 8.
- Stage 2: stage 1 output is then divided by 255.
- Stage 3: stage 2 output is then divided by 4095.

So, the clock is totally divided by 8353800. Total power is calculated for these scenarios in RTL and TLM as shown in Table 11. Simulation time is calculated also in RTL and TLM as shown in Table 12. It is obvious that UART is the dominant in simulation time as the clock is divided by a huge number so transmission will be very slow.

Note: The machine used to run scenarios has a third-generation core i7 processor which works on (2.00-2.5) GHz.

	RTL(mW)	TLM(mW)	Error%
Scenario A	1.2865	1.26	2
Scenario B	1.2718	1.259	1
Scenario C	1.2650	1.2589	0.5
Scenario D	-	-	-
Average	1.22744	1.2593	1.67

Table 11 Power Calculation results of overall system

	RTL(sec)	TLM(sec)	RTL/TLM Ratio
Scenario A	1616	66	24.4848
Scenario B	14400	66	216.181
Scenario C	41437	66	627.833
Scenario D	-	-	-
Average	19151	66	289.499

Table 12 Simulation Time results of overall system

The power model of **Scenario D** could not be calculated as the simulation needed a huge memory exceeding 1TB so it could not be achieved on normal PC.

6.3 Conclusion

A new methodology (TLPM) is proposed for power estimation using Transaction Level Modeling (TLM). The product life cycle is enhanced through the estimation of the power consumption at early design phase by using this methodology. Integration of the implementation with the commercial tools and flows is performed. The effectiveness and efficiency of the methodology are revealed with respect to accuracy along with the simulation time. The methodology speeds up the simulation time of different scenarios on TLM using (TLPM) up to 636 times than RTL simulation while keeping the error in power estimation is on average 1.2%.

References

[1] ”Transaction Level Modeling”, http://www.accellera.org.

[2] Rose, A., Swan, S., Pierce, J. and Fernandez, J.M. ”Transaction level modeling in SystemC.” Open SystemC Initiative 1(1.297), January 2005.

[3] Dhanwada, Nagu, Ing-Chao Lin, and Vijay Narayanan. ”A Power Estimation Methodology for SystemC Transaction Level Models.” Proceedings of IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis. pp. 142-147, September 2005.

[4] ”PKtool Power Kernel tool”, http://pktool.sourceforge.net.

[5] Vece, Giovanni B., and Massimo Conti. ”Power Estimation in Embedded Systems within a SystemC-based Design Context: the PKtool Environment.” Intelligent solutions in Embedded Systems. pp. 179-184, June2009.

[6] Greaves, D., and Yasin, M. ”TLM POWER3: Power Estimation Methodology for SystemC TLM 2.0.” Models, Methods, and Tools for Complex Chip Design. Springer International Publishing, pp. 53-68, August 2014.

[7] Cai, Lukai, and Daniel Gajski. ”Transaction Level Modeling: An Overview.” Proceedings of IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis. pp. 19-24, October 2003.

[8] Helmstetter, Claude, Tayeb Bouhadiba, Matthieu Moy, and Florence Maraninchi. ”Fast and Modular Transaction-Level-Modeling.” IEEE Transactions on VLSI Systems. pp. 501-513, 2006.

[9] Bouhadiba, Tayeb, Matthieu Moy, and Florence Maraninchi. ”System- Level Modeling of Energy in TLM for Early Validation of Power and Thermal Management.” Proceedings of the Conference on Design, Automation and Test in Europe. pp. 1609-1614, March 2013.

[10] ”Power Compiler User Guide”, http://www.synopsys.com.

[11] ”QuestaSim User Guide”, https://www.mentor.com.

[12] Chadha, Rakesh, and Jayaram Bhasker. An ASIC Low Power Primer: Analysis, Techniques and Specification. Springer Science and Business Media, 2012.

[13] ”Design Compiler User Guide”, http://www.synopsys.com.

[14] ”Universal Verification Methodology”, http://www.accellera.org.

[15] ”Vista User Guide”, https://www.mentor.com.

[16] ”ARM Dual-Timer Module (SP804) Technical Reference”, http://www.arm.com.

[17] ”Zynq-7000 All Programmable SoC Technical Reference Manual”,

[18] https://www.xilinx.com.