Chair for Integrated Systems
Department of Electrical and Computer Engineering
Technical University of Munich

Verification Of Computing Board For Data Processing

Master thesis

Author:	Muhammad Farhan
Carried out at:	Max Planck Institute for Extraterrestial Physics
External advisor:	Dr. sc. Sabine Ott
Advisor:	Dr. sc. Sabine Ott
Supervisor:	Prof. Dr. sc. techn. Markus Plattner
Submission date:	October 20, 2020

Abstract

Field Programmable Gate Array based System on Chips are popular in space industry owing to runtime reconfigurability and instantaneous synthesis of the desired data processing architecture. Over the course of this thesis, an exhaustive theoretical study as well as practical implementation is performed on Polarfire FPGA based System-on-chip to create design framework for a custom board fabricated by Max Planck Institute for Extraterrestrial Physics which can act as a potential substitute for the Instrumentation Control Unit. All data processing components on the board including memories, data converters, interfaces connecting them and microprocessor based processing unit based on RISC-V are designed, tested and evaluated. At the end, a comprehensive board support package as well as a solid theoretical documentation is created to facilitate any data processing design to be deployed on the manufactured board.

Contents

List of Figures

List of Tables

1 Introduction

The work performed at Max Planck Institute and on the FPGA based board broadly deals with the development of potential replacement for the triple stack Instrumentation board based on Field Programmable Gate Array technology. It includes several interfaces, memory components, data converters as well as a system-on-chip for efficient designing for any data processing system which might be eventually deployed on it. In this section, we will present an overview of motivation of the project as well as the outline of the thesis including a brief overview of our approach.

1.1 Motivation

WFI Athena [43] is a mission from Max Planck for The WFI (Wide Frame Imager) is one of the two scientific instruments proposed for Athena, the mission selected to address the "Hot and Energetic Universe" science theme identified by ESA for its L2 large satellite mission with launch in early 2030.

The on-board electronics on the spacecraft is highly complex with several core systems but the main portion comprises of Sensor interface unit, Instrumentation control unit and Power supply unit bundled together in form of triple stack card. The purpose of this thesis is to successfully perform a feasibility study and create a redundancy for the triple stack card in the form of custom computing board which can be deployed in case of severe malfunctioning in the main board which is being created by the Electronics Department at the Max Planck Institute for Extraterrestrial Physics along with its consortium partners.

1.2 Problem

As with every engineering project, there is a need to create a redundancy for the main circuitry and hence with this motivation, the idea to fabricate this board was conceived. The board contains several components including SRAMs, FRAM, QDR II+, LPDDR3, Flash, UART, FPGA/SoC, CAN, GPIO, ADC and Power monitoring circuitry.

However, owing to time constraint, over the course of this thesis, only FPGA/SoC, ADC, SRAM, FRAM and UART circuitry have been verified, with complete documentation highlight the function and purpose of every component on the board.

A brief overview of the board with all the relevant chips and circuitry is as shown below:

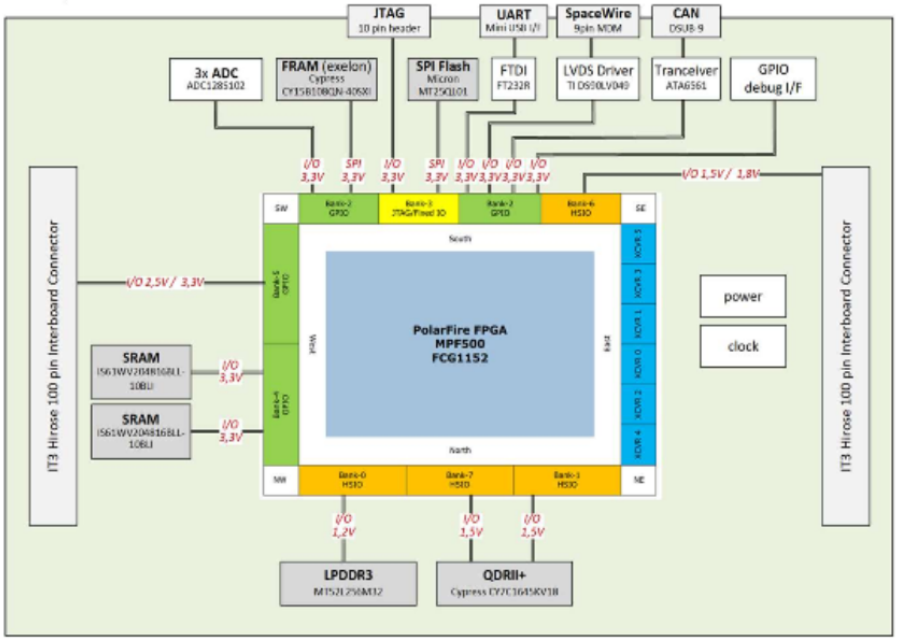


Figure 1.1: Board Block Diagram

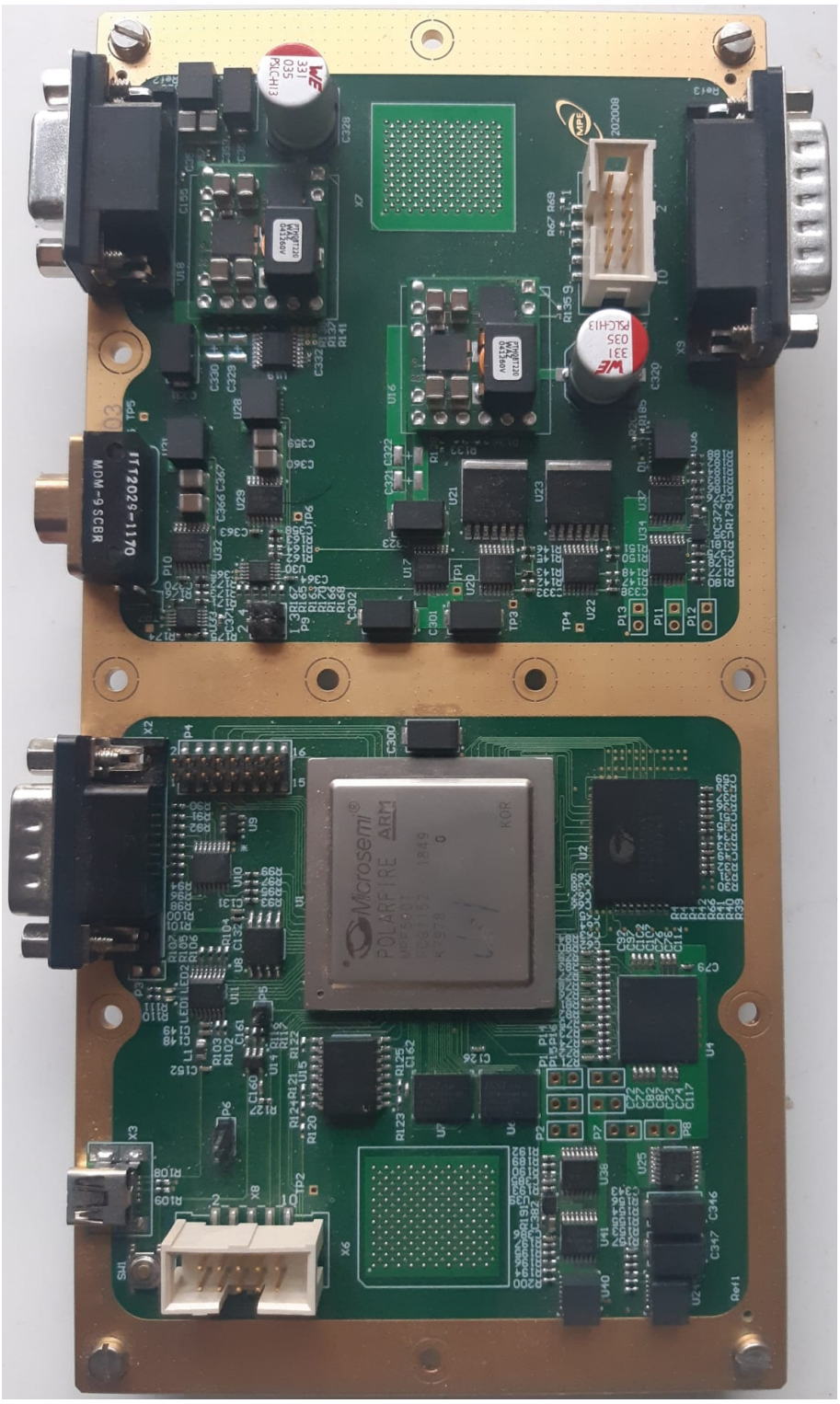


Figure 1.2: Board Top View

1.3 Goals of Thesis

The goal of this thesis is to verify the communication interface as well as memory components on the board and as such create custom drivers for them to facilitate the debugging and further system development.

The outcome will be a comprehensive documentation as well as verified functional Verilog models which can then be utilized for debugging, development or integration into any new design created on the board. It will also include a soft-core processor with a simple housekeeping routine to further facilitate any firmware based development on the manufactured board.

1.4 Approach

The steps followed to develop the system as well its documentation are as listed below:

- Understanding the Interface and Components: In this step, we try to analyse the board with regards to all the chips placed on it including memories, interfaces and the field programmable gate array.
- Literature Review: In this step, we review all the documentation of the integrated circuits as well as the field programmable array with regards to interfaces and resources available therein to ensure suitable designs later on.
- Design: In this step, suitable designs were created in form of the top wrapper to be later deployed onto the board alongside the state machines describing their behaviour. It involved selecting optimum number of pins describing the interfaces to and from the modules and included synchronization elements between the designs as well. There were also Modelsim models created as test-bench to verify the functionality of the design with wave viewer.
- Implementation: The designs were implemented on the platform in the form of firmware or hardware via expressing them in a hardware description language. Syntax as well as pre-requisites of the platform in form of memory map, standard HDL description or the HAL of the processor were adhered to.
- Evaluation and Testing: In this step, the desired results were compared to the designed results to ensure proper functioning of the designs. The resources utilized by the designs on the platform were also recorded.

1.5 Outline

The work is structured as follows. In Chapter 1 a brief introduction of the problem is given along with the motivation to solve it and then a brief overview of the solution is given. In chapter 2, we discuss the theoretical fundamentals of digital designing as well as processor based systems, it also reviews the major concepts of computer engineering along with firmware based development as well the details of FPGA architectures, ADCs, Interconnects as well as Memory Technologies. In chapter 3, the design automation techniques for digital designing as well as Integrated development environments for the design will be discussed, there will also be a brief overview of the Polarfire FPGA SoC family. In chapter 4, the designing as well as probing function of the UART core is developed and implemented. In chapter 5, the ADC subsystem, its development as well as its integration in the probing path will be developed and discussed. In chapter 6, there is a comprehensive discussion as well as development of memory subsystems on the board including SRAM, FRAM as well as foundational discussion on LPDDR and QDR II+. In chapter 7, a concept study is performed for the I2C subsystem present on the board which can be developed and tested in the future. In chapter 8, a housekeeping procedure based on MiV core is deployed and documented for further firmware development based on it for the future. Finally, in conclusion, future goals are decided and suggestions for further work are provided.

2 Theoretical Fundamentals

In this chapter, the necessary background information is introduced in order to better document and understand the thesis. The section will mostly introduce digital technology with special emphasis on Field Programmable Gate Array technology and Processor based system. As it is necessary to know about about logic families as well as the fundamentals of digital circuit design and technology, a brief treatment of those topics shall also be meted out. Also the basics of modern computer engineering including the concepts of API (Application Program Interface), ABI (Application Binary Interface), Shared libraries and Compiler technology shall also be covered as modern embedded system design is increasingly being steered towards software development and RTOS (Real-Time Operating System) based development making it integral to have good grasp over these concepts as well.

2.1 Digital Electronics and Logic families

Digital electronics utilizes the flipping and storage of bits to create functional machines. The building block of all the modern digital circuits is the transistor based on semiconductor materials and can be broadly classified into being current controlled or voltage controlled device with given voltage and current characteristics [2]. The modern physical layer electronics mostly utilizes the CMOS based voltage controlled device to create switching circuitry owing to its efficient nature, ease of dimension reduction, less static power dissipation and high packing density, although the fastest logic family present is considered to be Bipolar Junction Transistor (BJT) based Emitter Coupled Logic (ECL). In modern day electronics, a discrete transistor as a switch is never utilized but rather transistor based structures fulfilling certain criterion are created which are known as the logic families.

In this section, we will discuss the large signal transistor model, basics of logic families, CMOS logic family and inverter chain based digital logic storage elements.

2.1.1 Transistor, Switch and Large Signal Model

Various transistor models have been proposed to quantify the exact behaviour of the transistor as per the given conditions and requirements. Broadly, these models can be classified as being Large signal models in which the movement can be described as happening across the whole of V-I curve of the device or the small signal model

wherein we take one bias point on the device V-I curve and linearise it about it. As is shown in the figure below.

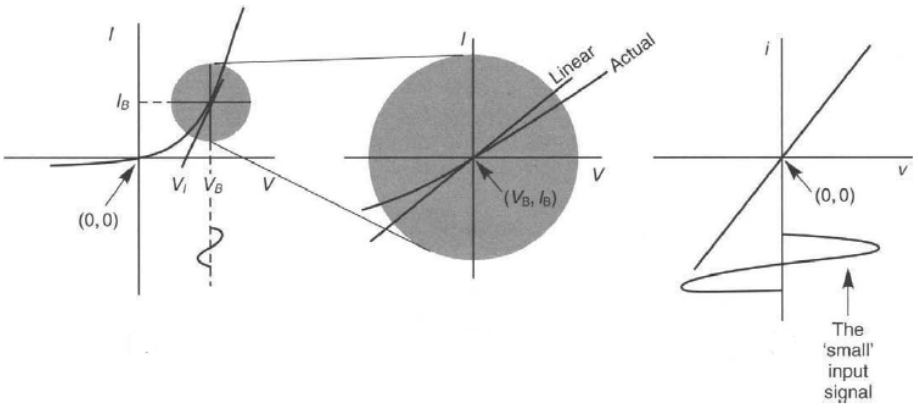


Figure 2.1: Small Signal Model of Transistor [1]

As logic families and switching in general is concerned with turning on and off the transistor, we will only consider the large signal model whereas people designing analogue circuit generally consider the small signal models in their designs.

The figure below shows the schematic as well as V-I characteristic of the Bipolar junction transistor being used as a switch.

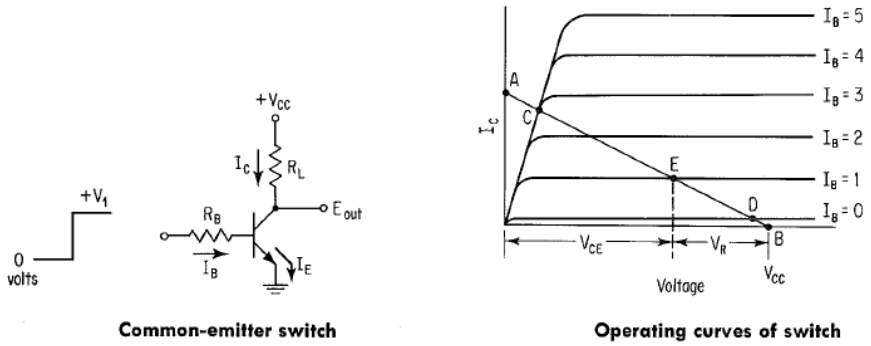


Figure 2.2: Large Signal Model of Transistor[2]

The load line from A to B specifies the output voltage of the structure with the switch being turned on at C and turning off or going to zero voltage at point D.

2.1.2 Logic Families

The requirements in terms of area, speed, interconnections and power led to a surge in development of transistor based structures fulfilling the required criterion and as such discrete transistor is never used for the design. Such transistor structures are known as the logic families and they are designed with certain figures of merit in mind. The table below depicts some of these characteristics:

Logic Family (Silicon Technology)			Introduction	Features	Limitations
Transistor Logic Families (Bipolar Transistor Technology)	Saturated Logic Families (ON – Saturation Mode, OFF – Cut Off Mode)	1. RTL (Resistor Transistor Logic)	- In common use before the development of ICs. Common Emitter Configuration. - Logic 1: 1-3.6 V and Logic 0: 0.2V	- First logic family, require minimum number of transistors.	- Low speed, high power dissipation - Low fan out, poor noise immunity - Operating speed <4MHz.
		2. DCTL (Direct Coupled Transistor Logic)	- Direct coupled transistors. - Base resistors of RTL are removed.	- Simpler than RTL, easy to fabricate. - Fewer components hence economical.	- Small logic swing, poor noise margin. - Current hogging
		3. DTL (Diode Transistor Logic)	- Use diodes and transistors. - Input is fed through diodes followed by transistor at the output side.	- First circuit configuration designed into IC. - Very small in size and high reliability at very low price. - Greater fan out and improved noise margins.	- No low and constant output impedance in both states.
		4. TTL (Transistor-Transistor Logic)	- Use all transistors totem pole output. - Function of diodes in DTL is performed by multi-emitter transistor at input	- Fast switching time, larger fan out. - Reduced silicon chip area. - Easy to interface with other logic families.	- Large current spike when switching from low to high. - Less noise immunity (0.4V)
		5. IIL (Integrated Injection Logic)	- Merged Transistor Logic (MTL). - Both PNP and NPN transistors are used. - Designed around multi-collector inverting transistors.	- High component density, less power dissipation. - Low metal interconnection. - Used in MSI and LSI designs.	- Poor noise immunity.
	Non-Saturated Logic (ON – Active Mode, OFF – Cut Off Mode)	6.ECL (Emitter Coupled Logic)	- Non saturated logic/Current mode logic. - Compliment output/eliminates the need of inverter. - Logic 1: -0.8 and Logic 0: -1.7	- Fastest logic family - Used in very high frequency applications. - No noise spikes, large fan out.	- Require large silicon area, high power dissipation (high cost). - Inconvenient voltage levels. - Low noise margins.
MOS Logic Families (Unipolar Transistor Technology)		7.MOS Logic (Metal Oxide Semiconductor Logic)	- Use pMOS, nMOS or both with high packaging density. - Easy to design and fabricate - Less power drawn due to gate dielectric.	- Lower power dissipation. - Shorter rise and fall times. - Large fan-out.	- Larger propagation due to high output impedance. - Noise margin is around 1V.

Parameter	RTL	IIL	DTL	HTL	TTL	ECL	MOS	CMOS
Basic Gate	NOR	NOR	NAND	NAND	NAND	OR-NOR	NAND	NOR-NAND
Fan Out	5	Depends on Injector Current	8	10	10-20	25	20	20-50
Power Dissipation	12 mW	6 nW – 70 uW	8-12 mW	55 mW	10 mW	40-55 mW	0.2-10 mW	0.025-1.01 mW
Noise Immunity	Nominal	Poor	Good	Excellent	Very Good	Poor	Good	Very Good
Propagation Delay	12 nSec	25-30 nSec	30 nSec	4 nSec	10 nSec	1-2 nSec	300 nSec	70 nSec
Clock Rate	8 MHz	-	72 MHz	4 MHz	35 MHz	+60 MHz	2 MHz	10 MHz
Speed X Power	144	Less than 1	300	-	100	100	60	70

Figure 2.3: Logic Families Comparison [3]

Normally, at present, not much emphasis has been attached to the transistor based switching circuitry since the advent of VLSI has shrunken the transistor very rapidly leading to reduction in emphasis towards applied aspects of physical layer and more towards the design itself. Some other transistor level design with regards to logic manipulating structures include transistor sizing to manage speed and fanout and inclusion of Variable Power Supplies and Transistor Thresholds into the design as well. [44] Similarly, the simulation and verification of analog circuitry is also of great relevance with multiple benchmarks and SPICE based simulators designed for verification. [45]

2.1.3 CMOS Logic Gates and Gate Standard

As the CMOS based logic circuitry predominates, we shall use it as a reference to create the logic gates based on it. A basic CMOS based logic gate structure is as shown below:

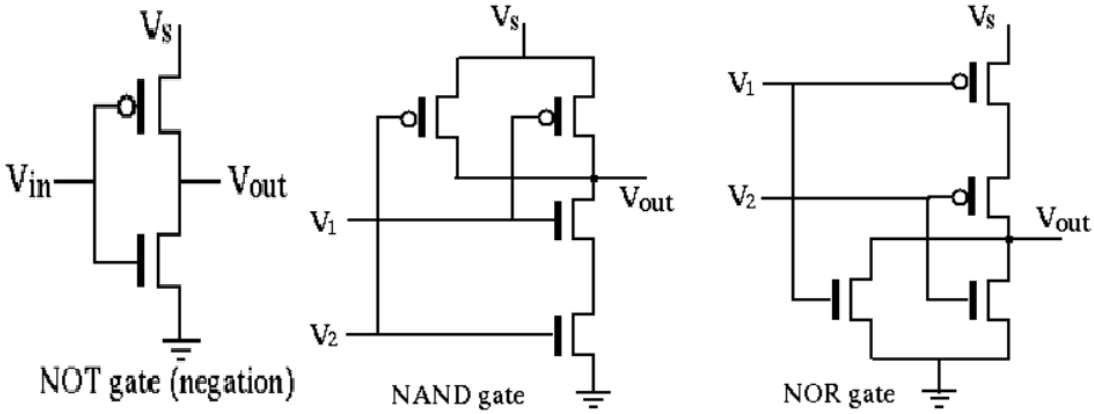


Figure 2.4: Cmos Based Logic Gates

Normally each combinational logic element has a delay associated with it and it is specified in a document explaining the implementation of the cell called the delay lookup table specifying expected delay at each fanout and connected capacitance.

	Output capacitance (fF)						
Input slew	0.4	0.8	1.6	3.2	6.4	12.8	25.6
Output rise delay							
0.0075	0.018	0.022	0.032	0.051	0.089	0.164	0.315
0.0375	0.032	0.038	0.048	0.066	0.104	0.179	0.330
0.1500	0.059	0.069	0.087	0.117	0.164	0.239	0.388
0.6000	0.129	0.145	0.174	0.224	0.305	0.433	0.628
Output rise transition							
0.0075	0.012	0.016	0.025	0.043	0.079	0.151	0.294
0.0375	0.018	0.021	0.027	0.043	0.079	0.151	0.294
0.1500	0.036	0.042	0.052	0.067	0.092	0.152	0.294
0.6000	0.095	0.100	0.114	0.139	0.183	0.252	0.353

Figure 2.5: Sample logic delay table[4]

2.1.4 Digital Storage device

Ideally, all the digital circuitry should be combinational or direct logic switching circuitry based, however, owing to hazards as well as a need for a need to store the intermediate logic so as to slow down the circuit and prevent the logic hazards nearly all modern day circuits are sequential in nature [4].

All the storage elements in modern circuits are based on the inverter chain as is shown in the figure below, wherein the two stable point A and B are used to store the logical bit whereas point C being metastable is never reached.

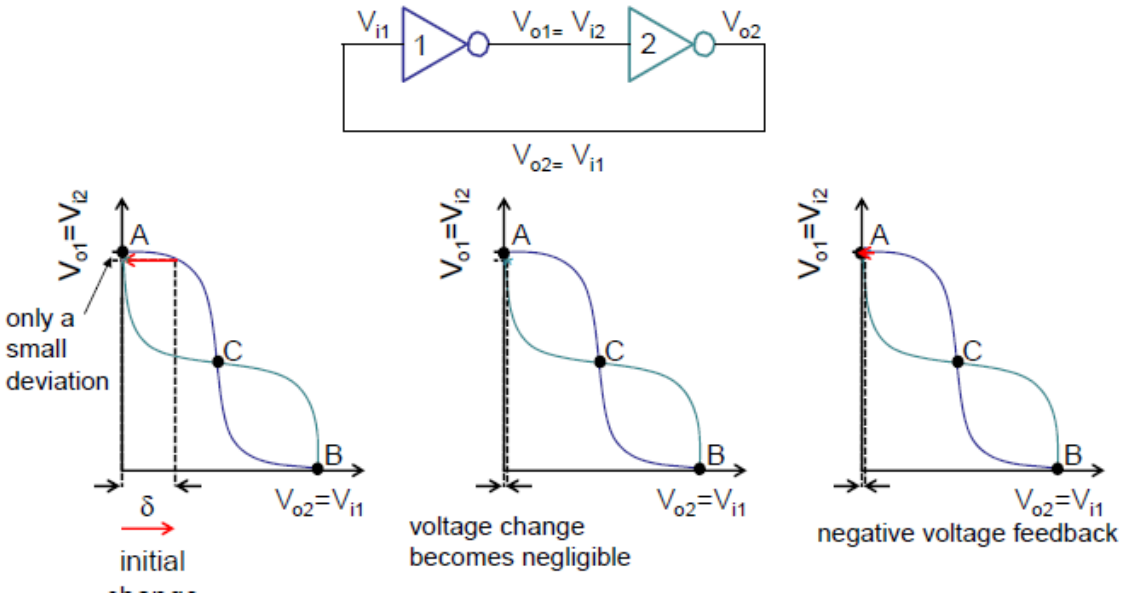


Figure 2.6: The Inverter Chain[4]

The first basic element, derived from the inverter chain, which stores the data and can be made to switch between two logic levels is an SR-Latch which also happens to be level triggered, i.e, the logic is triggered as soon as the input changes as opposed to waiting for an event like the clock edge. Latches are generally avoided in the designs as they bring about an element of uncertainty as the direct latency of the logic element comes into play and it leaves a very tight margin for the overall logic to synchronize with each other, Only very experienced designers with very deep knowledge of semiconductor process and proper characterizations of their designs utilize latches in their designs. A hardware description of the level latch is as show:

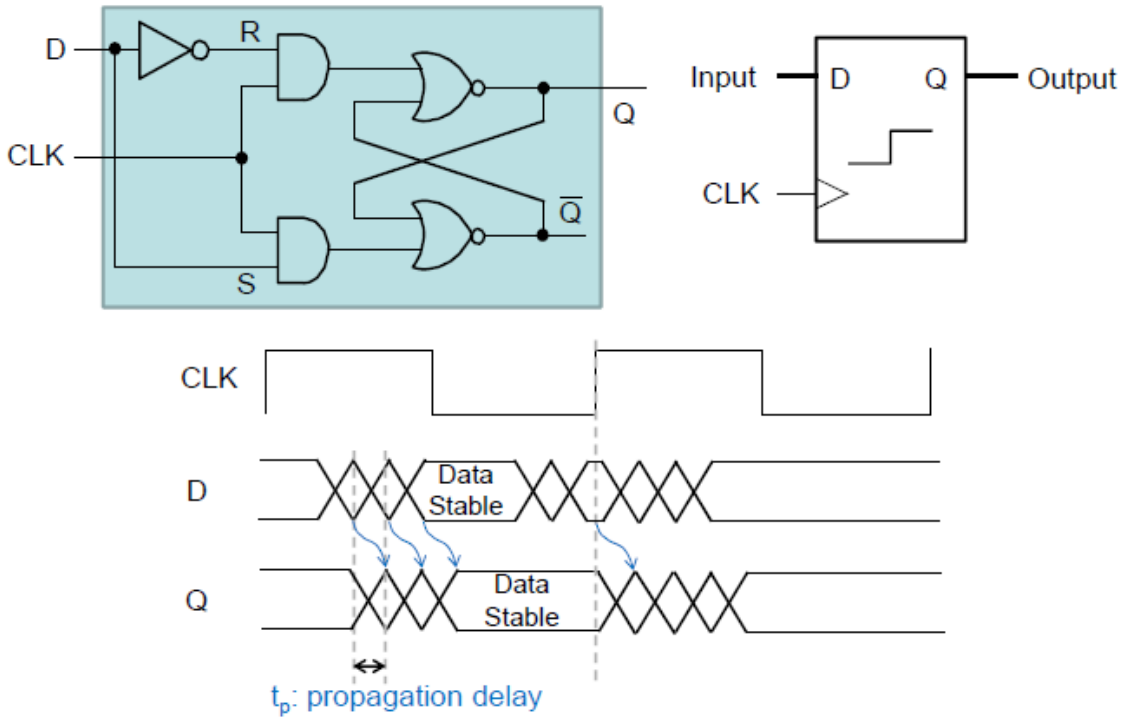


Figure 2.7: Level Triggered Latch[4]

The second element derived from the latch is an event-triggered flip-flop wherein the logic state is changed only at the edge of a clock pulse and remains stable otherwise. Nearly all digital circuits are based on the Flip-Flops as the timing structure introduced therein is easy to synchronize and scale. A hardware description of the edge triggered Flip-Flop is as shown below:

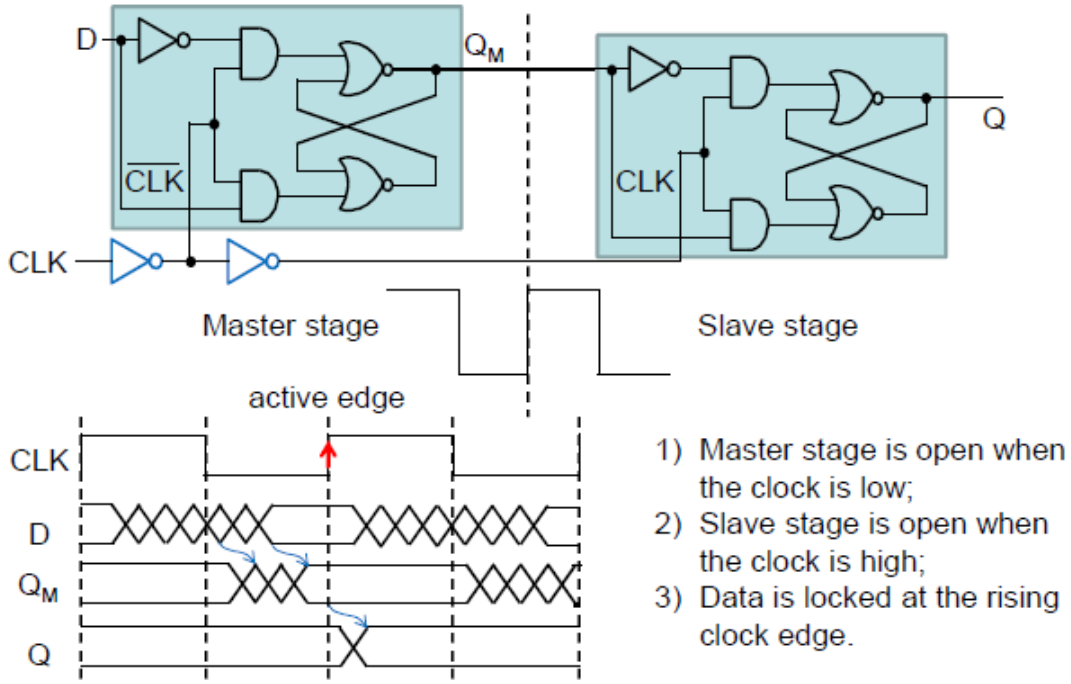


Figure 2.8: The Edge Triggered Flip-Flop[4]

2.2 Digital Designing with Hardware Description Languages

Modern front-end digital circuit design mostly deals with the utilization of hardware description languages to express the design in a programming language. Although at synthesis level, it is possible to use multiple high level languages to synthesize the design, for the duration of this thesis we will only consider the hardware description language Verilog to document and express our designs.

In broad sense, Verilog makes it possible to express designs in terms of five different levels[46] of abstraction including:

- Switch Level Modelling : At this level of abstraction, the basic entity for the design is a switch mirroring the basic CMOS switch. However, as at this level, the people who design ASICs work, hence we will not consider this abstraction within the scope of this thesis
- Gate Level Modelling : At this level of abstraction, the basic gate primitives stored in the library of Verilog are used to model the behaviour of the circuit. Additionally, the user may introduce user defined primitives and further utilize them in the design as well.
- Data Flow Modelling : Data Flow modelling is based on utilizing the assign keyword of Verilog and using it to create a design. In essence, it is equivalent to flow of data within the design to bring about the desired functionality.
- Behavioural Modelling : At this stage of abstraction, the always block is used to explain the behaviour of the module. Most clocking events or the sequential circuits utilize this level of abstraction for their designs.
- Structural Modelling : At this stage of abstraction, entire designs are instantiated in the code as an IP and input/outputs can be assigned to the ports of the structure.

2.2.1 Combinational and Sequential Logic

All the modern digital circuits can be classified as being either combinational or expressing memory wherein they are classified as being sequential circuits [47].

Combinational Circuit

Combinational Circuits do not exhibit memory and are modelled as time independent elements in the design, atleast during the designing phase. There are multiple equivalent ways of modelling the functionality of combinational circuit in Verilog including Gate Level modelling, Data Flow Modelling and Behavioural modelling as is shown below.

```
module Combo(
input a,
input b,
input c,
output d );

and(d,a,b);

assign d = a & b;

always @(*)
d = a & b;

endmodule
```

Figure 2.9: Verilog Description of Combinational Circuit

Sequential Circuit

Sequential Circuits are memory elements in which the current output explicitly depends on all the previous inputs to the circuit. Similar to combinational circuits, sequential circuits can also be modelled at Gate Level, Data Flow and Behavioural levels of abstraction. However, since we mostly use clock signal in digital designs, we will only consider the behavioural description of such a circuit as is shown below.

```
module Seq(
(
input clk,
input arst_n,
input a,
input b,
output d );

reg  State, Nxt_State;
localparam IDLE = 1'b0, ACT = 1'b1;

always @(clk, arst_n)
if (arst_n)
State <= 1'b0;
else
State <= Nxt_State;

assign =

endmodule
```

Figure 2.10: Verilog Description of Sequential Circuit

2.2.2 Flip-Flops and Latches

The two major sequential elements used in design of digital circuits are Flip-Flop and Latches. The major difference between then being the instant at which they are triggered with flip-flops triggered at the edges of the clock whereas latches are triggered at any change at the input of the circuit.

```
always @(posedge clk, arst_n)
if (arst_n)
State <= 1'b0;
else
State <= Nxt_State;
```

Figure 2.11: Verilog Description of Flip-Flop

Generally Latch based designs are fast as there is no delay in waiting for clock signal to trigger where as Flip-Flop are slow as they wait for clock signal to trigger. However, in most designs, Latches are generally avoided as they often lead to an unpredictable behaviour in the circuits [48].

```
always @(clk, arst_n)
if (arst_n)
State <= 1'b0;
else
State <= Nxt_State;
```

Figure 2.12: Verilog Description of a Latch

2.2.3 Control Signals - Clock and Reset

The two basic signals used in modern digital circuits to control and manipulate the logic are clock and reset.

Synchronous and Asynchronous Reset

The reset circuitry [49] in digital designing is used to bring back the digital circuit to its initial state. Broadly Speaking, there are two types of reset within the scope of digital designing including synchronous and asynchronous reset.

In the synchronous reset, the reset circuitry is triggered at the positive edge of the clock and as such the reset mechanism is in sync with the clocking mechanism of the design.

```
always @(posedge clk, negedge arst_n)
if (!arst_n)
State <= 1'b0;
else
State <= Nxt_State;
```

Figure 2.13: Verilog Description of a Synchronous Reset

In the asynchronous case, the reset and clocking circuitry is decoupled from each other and as such the reset action function separately from the clocking mechanism of the design. This is the more preferred form of design as clocking and reset are totally detached from one another. This particular reset design is also used over the course of this project and thesis.

```
always @(posedge clk, arst_n)
if (arst_n)
State <= 1'b0;
else
State <= Nxt_State;
```

Figure 2.14: Verilog Description of an Asynchronous Reset

Clock Signal

As the clock is the most integral signal in the design of digital circuits, this section will discuss the basics of clock generation, clock distribution and the figures of merit for clock and clock generation circuitry.

Clock Generation: The clock signal can be generated [50] in two distinct but related ways including via crystal oscillator or a clock generator which combines combines an oscillator with one or more PLLs, output dividers and output buffers. Clock generators and clock buffers are useful when several frequencies are required and the target ICs are all on the same board or in the same FPGA. Clocks can also be classified as being free-running or synchronous [51].

- Free Running applications require one or more independent clocks without any special phase-lock or synchronization requirements. Example applications are standard processors, memory controllers, SoCs and peripheral components
- Synchronous applications require continuous communication and network-level synchronization. Examples are Optical Transport Networking (OTN), mobile backhaul, synchronous Ethernet and HD SDI video transmission. These applications require transmitters and receivers to operate at the same frequency.

Another division could be in terms of being internal or external clock [52]:

- Internal oscillators are commonly used to provide timing for MCUs that don't require accurate timing. Internal oscillators are good enough for low-baud UART communication.
- external crystals and oscillators are required for communication protocols such as CAN, USB or Ethernet which have stricter timing accuracy requirements. Using an external oscillator allows a wider range of frequencies where the internal oscillator(s) are typically one frequency with a handful of clock prescaler options.

Clock Clock Quality: The main factor impacting quality of the clock circulating on a die is the jitter which can be quantified in three distinct ways [53] including:

- Cycle-to-cycle jitter measures the maximum change in the clock period between any two adjacent clock cycles, typically measured over 1,000 cycles.
- Period jitter is the maximum deviation in clock period with respect to an ideal period over a large number of cycles (10,000 is typical).
- Phase jitter is the figure of merit for demanding, high-speed SerDes applications. It is a ratio of noise power to signal power calculated by integrating the clock single sideband phase noise across a range of frequencies offset from a carrier signal.

Clock Distribution: The clock distribution on a die is concerned more with physical design it includes the logic family of the transistor structures as well as the overall routing structures of the clock tree on the die.

The logic structures [54] for interaction with the clock tree I/Os include

- LVPECL stands for Low-Voltage Positive Emitter-Coupled Logic, and it is a power optimized version of PECL or Positive Emitter-Coupled Logic. It uses a positive 3.3 V power supply.
- LVDS is Low-Voltage Differential Signaling, and it is only a physical layer specification, but a data link layer is often added by communication standards and applications.
- Current Mode Logic transmits data at speeds between 312.5 Mbit/s and 3.125 Gbit/s across standard circuit boards.
- High-Speed Current Steering Logic is differential logic with two output pins that switch between 0 and 14 mA.

It is also possible to switch between these logic levels and there several specialized circuits designed for such task.

The clock tree synthesis [55] is a physical design aspect and as such in-depth explanation of it is beyond the scope of this thesis. Some common VLSI structures for the distribution of the clocking resources on a semiconductor die include H-Tree, X-Tree, Multi level clock tree and Fish bone.

2.2.4 Building Blocks of Digital Circuits

Almost all the digital circuits following the paradigm of clock or synchronous design integrate some basic building blocks in their design. some of the most common such digital blocks are as listed below:

Counters

Counter [56] are used in digital logic to record the number of clock. There can be broadly two distinct types of such counter including up and down counter. Counters in Verilog are often modelled at behavioural level of abstraction as they are usually triggered at positive edge of the clock. Counters are also broadly classified as being up-counters and down-counters. A Verilog description of the counter is as shown below:

```
reg [3:0] bit_count;
always @(posedge ref_clk, negedge arst_n)
if (( state != nxt_state ) || (!arst_n))
 bit_count <= 3'd0;
else
 bit_count <= bit_count + 1'b1;
```

Figure 2.15: Verilog Description of a Counter

Shift registers

The methods by which the data moves in and out of a register or a register bank. A simple verilog description of the shift register [57] is as given below:

```
if(state == DATA_HIGH)
    begin
    data_high <= {data_high[6:0],SDA_in};
end
```

Figure 2.16: Verilog Description of a Shift Register

One shot timer

Often there is a need to trigger a certain event at time in digital circuit and hence timers come in handy when such a situation is desired. One such commonly used timer is the one-shot timer [58] which is also often used in baud-rate generation or event triggering. A simple verilog description one shot timer is as given below:

```
module oneshot
   #(parameter Clkfrequency=50000000,
     parameter Baudrate = 115200,//9600,
     parameter cntrlimit = (Clkfrequency/Baudrate)
    )
     (input clk,
      input arst_n,
      output tick);

      reg [13:0] cntr;

      always @(posedge clk,negedge arst_n)
          if(!arst_n)
              cntr<=14'd0;
          else if(cntr==cntrlimit)//(tick)
              cntr<=14'd0;
          else
                  cntr<=cntr+1'b1;

    assign tick=(cntr<cntrlimit)? 1'b0:1'b1;//tick = (cntr==cntrlimit)

endmodule
```

Figure 2.17: Verilog Description of One-shot Counter

Edge detection circuit

The Edge detection circuit [59] samples the incoming signal and determines whether the edge condition is present or not. The edge detection circuit can be realized in a number of equivalent ways in the hardware description language. Two common HDL definitions of such a circuitry are as given below:

```
//edge detection of the data_Valid
always @(posedge clk) data_valid_in_reg <= data_valid_in;
assign data_valid_reg = data_valid_in_reg && ~data_valid_in;
```

Figure 2.18: Verilog description of edge detection circuitry 1

```
/* Handle address incrementing to cycle through reading
   bytes from the ADC device input pins */
always @ (posedge addr_inc or negedge rst)
  if (!rst)
    dout_addr <= 2'b00;
  else
    dout_addr <= dout_addr + 1'b1;
```

Figure 2.19: Verilog description of edge detection circuitry 2

Mux and Demux

The utilization of multiplexer and demultiplexers [60] is very frequent in digital designing. The circuitry for multiplexing and demultiplexing can be realized in number of ways and at different levels of abstraction. Although it is possible to use data-flow model to express a multiplexing circuitry, here, we will only consider behavioural model followed by a case statement. A Verilog description of such a circuit is given below

```
always @ (*)
  case (sclk_count)
    4'd3: dout = dout_addr[1];
    4'd4: dout = dout_addr[0];
    default: dout = 1'b0;
  endcase
```

Figure 2.20: Verilog Description of Multiplexing Circuitry

Clock shifting circuit

Often times there is a need to create a multiple phased clock signal [5] in digital design, although such signal should generally be created using a phased lock loop, it is possible to use logic fabric to generate such clocks as well. As program for such operation would be very long and be based on counters, here we only present phase shift diagram needed to carry out such an operation. Note that the clock division circuit is also similar but utilizes the counters mentioned above.

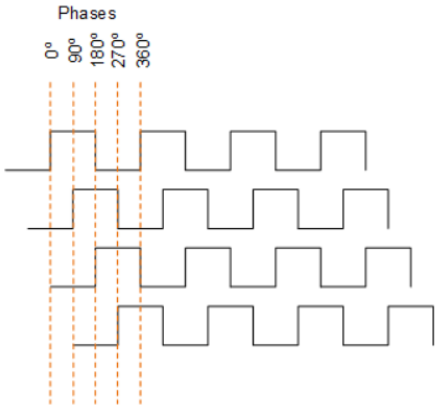
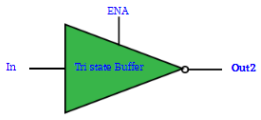


Figure 2.21: Clock Phase Shift Circuit[5]

Pin Multiplexing

The Pin multiplexing [6] is one of the most frequent operations performed in the Field Programmable Gate Array circuitry and as such the buffer construct that it utilizes shall be discussed here briefly:

ENA	IN	OUT2
0	0	1
0	1	0
1	0	Z
1	1	Z




```
assign data_sram = (State == Start_Write)? data_in: 21'bzzzzzzzzzzzzzzzzzzzzz ;
data_received = data_sram;
```

Figure 2.22: Verilog description of tri-state buffer[6]

In the above tri-state buffer, State signal is used to turn the otherwise input into an output signal to be relayed to the circuit in succession. when the state signal is not triggered, it is possible to treat the data sram as input signal and read data from it.

Finite State Machine

Finite state machines m refers to the abstract machines which are used to trigger certain logic or control a digital design. Broadly, Finite State Machines can be divided into two distinct categories including:

- Moore State Machine in which the output logic is only the function of the present state of the state machine.
- Mealy State Machine in which the output logic is function of both present state and the input to the circuit.

$$\mathbf{FSM} = (S, I, O, \delta, \lambda, S^0) \qquad \text{6-tuple}$$

1) $S = \{\underline{\hat{s}}_0, \underline{\hat{s}}_1, \ldots, \underline{\hat{s}}_{2^r-1}\} = \mathcal{B}^r$: finite set of states, $0 < |S| < \infty$
   $= \{S_0, S_1, \ldots, S_{2^r-1}\}$
2) $I = \{\underline{\hat{x}}_0, \underline{\hat{x}}_1, \ldots, \underline{\hat{x}}_{2^n-1}\} = \mathcal{B}^n$: set of input patterns
   $= \{X_0, X_1, \ldots, X_{2^n-1}\}$ input alphabet
3) $O = \{\underline{\hat{y}}_0, \underline{\hat{y}}_1, \ldots, \underline{\hat{y}}_{2^m-1}\} = \mathcal{B}^m$: set of output patterns
   $= \{Y_0, Y_1, \ldots, Y_{2^m-1}\}$ output alphabet
4) $\delta : S \times I \to S, \quad (\underline{s}, \underline{x}) \to \underline{z}$: (state-) transition function
5) $\lambda : S \times I \to O, \quad (\underline{s}, \underline{x}) \to \underline{y}$: output function
6) $S^0 \in S$: initial state

Types of machines:

Mealy automaton $\lambda : S \times I \to O, \quad \underline{y} = \lambda(\underline{s}, \underline{x})$
Moore automaton $\lambda : S \to O, \quad \underline{y} = \lambda(\underline{s})$

Figure 2.23: The Mathematical Description of FSM[7]

2.2.5 Control and Datapath Division

The previous section described some of the most basic design structures which can be used to construct a digital design, although on a system level, we need to divide the design as such it can be used in the most efficient method with regards to area, speed and power requirements. In such cases, the design can be divided into two logical sections including control path and data-path [8], with control path describing the logical operations being performed on the datapath of the circuitry.

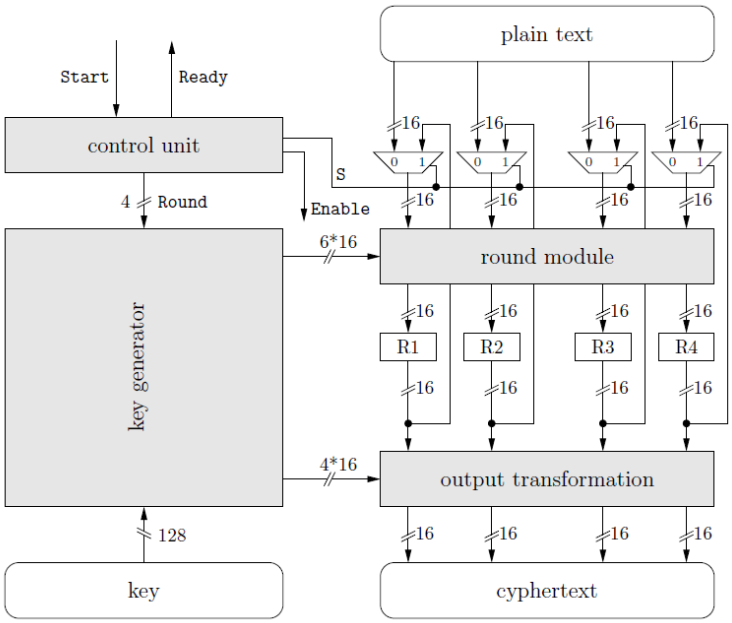


Figure 2.24: Control and Datapath division of the design[8]

The diagram above shows one division in terms of control and datapath in which the control unit multiplexes the round module at each round thereby efficiently using the fabric of the digital design.

2.3 Verification Fundamentals

Verification of digital circuit in concerned with ensuring that the desired behaviour is exhibited by the designed circuitry. Broadly classifying, the verification can be divided into four distinct categories including:

2.3.1 Logic Tracing

The simplest form of verifying the functionality of a digital circuit is by tracing through the input and output logic combinations and viewing the waveform on the waveform viewer [61]. A simple waveform from Modelsim simulator for the AND gate is as shown:

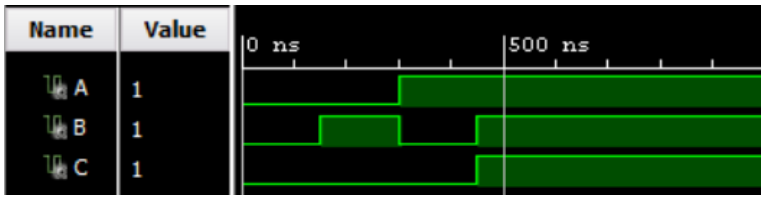


Figure 2.25: Logic Tracing with Modelsim

Over the course of this thesis, the simplest form of verification which is logic tracing will be utilized to ascertain the functionality of the design, but a brief discussion of other verification types shall be given.

2.3.2 Assertion Based Verification

Assertion [62] based verification can be considered as being the next step towards automated testing wherein we place assertions or automated test on the netlist of the circuits. One instance of an assertion is as shown below:

```
//////////////////////////////////////////Test-Bench Verification routine//////////////////

always @(posedge clk)
 begin

  //With the below statement, module waits for the transaction between them to complete
  @(posedge cs);

  if ( mosi_t == data_out )
    t = t + 1'b1;

  if ( data_in == miso_t )
    r = r + 1'b1;

 end


//////////////////////////////////////////Test-Bench Display routine////////////////////////

always @(posedge clk)
begin
    if(RUN == t) begin
    $display("The data received sucessfully by SPI Module");
    t= 1'b0;
    end

    if(RUN == r) begin
    $display("The data Transmitted sucessfully by SPI Module");
    r=1'b0;
    end

end
```

Figure 2.26: Assertion Based Verification of Design

2.3.3 Model Based Verification

Model based verification [63] is concerned with mimicking the behaviour of the input and output circuitry to the design under test or DUT by developing software models of those structures. As over the course of this thesis Model Based Verification has not been carried out, we will only present a brief overview of the such a verification process:

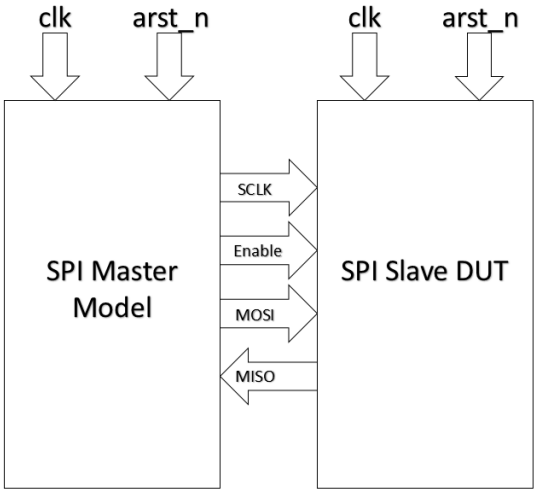


Figure 2.27: An overview of Model Based Verification

In the above diagram, a fully functional model of SPI master is created using C language in the test bench and a Slave SPI core is instantiated. A connection between the two is made in the test bench with assertions and verifications on the master side to ensure proper functioning of the Design under Test module.

2.3.4 UVM Based Verification

The UVM [9] is a simulation based verification methodology . It is the defacto standard for the verification of the digital circuits in the industry. The UVM provides a base class library written in SystemVerilog, an object oriented Hardware Description and Verification Language (HDVL) which is an extension of the original Verilog HDL. A classical UVM testbench consists of reusable verification components (VCs). Each of those VCs has a defined architecture containing the following elements:

- A Sequencer is an advanced input stimulus generator responsible to generate input sequences.
- The Driver adds control signals to the input stimuli provided by the Sequencer.
- The Monitor samples the output sequences transmitted from the Device Under Test (DUT) to perform validation with regards to the specification. The monitor also collects coverage information.

Sequencer, Driver as well as Monitor are in-built classes of the UVM library.

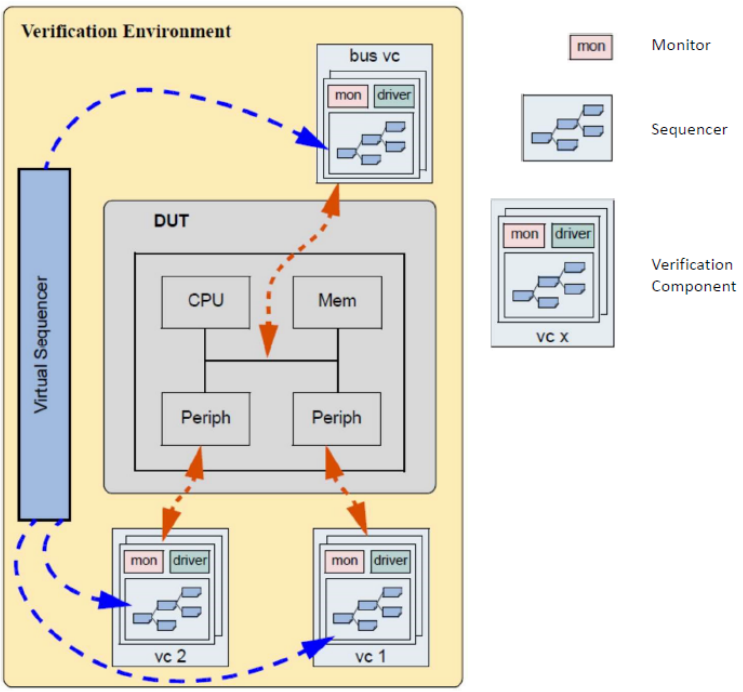


Figure 2.28: Sample UVM verification environment[9]

2.3.5 Formal Method of Verification

Formal Verification [64] methods utilize powerful mathematical techniques to check the functionality of design, unlike developing a simulation model of the system, under test usually because there is a state space explosion in terms of verification of the design verification with input pattern generation. Such features are generally built into powerful simulators and logic checkers, and as such the mathematical models constructed for their checking are beyond the scope of this thesis, but we will give a basic overview of the techniques used here.

- Equivalence checking is to show whether two designs are equivalent. We call two designs equivalent if for all input combinations the output is exactly the same. Equivalence checking is needed in mostly all synthesis design flows when moving vertically down in terms of abstraction layers, in particular to show whether the implementation is equivalent on RTL and gate level. But it is also needed for proving that horizontal transformations, namely optimization steps, are valid transformations and did not alter the system's behaviour.

- The intention of property checking, also known as model checking, is to prove a model's compliance with given temporal logic expressions using fully automated methods. There exist mainly two decidable formalisms to characterize temporal logic expressions, namely Computational Tree Logic (CTL) and Linear Temporal Logic (LTL).

- In bounded model checking the state search problem is mapped to a Boolean Satisfiability (SAT) problem to cope with the state explosion problem. SAT solvers require less memory usage and show increased performance such that bounded model checking can handle systems with a higher number of state variables.

There can be several formal methods of verification but the scale and applicability of them are beyond the scope of this thesis.

2.4 Digital IC design flow

The first Integrated circuit was developed in 1958 by Jack Kilby [65] [66], where upon there was an explosion in terms of automation and algorithm development for Logic Synthesis, SPICE model development, Layout Synthesis and Physical Design Automation as well as efficient Simulation engines for circuit and design verification. As the eventual realization of the digital design is a Digital IC, the following section will cover the basics of the modern digital IC designing. A sample digital IC design flow is as shown below:

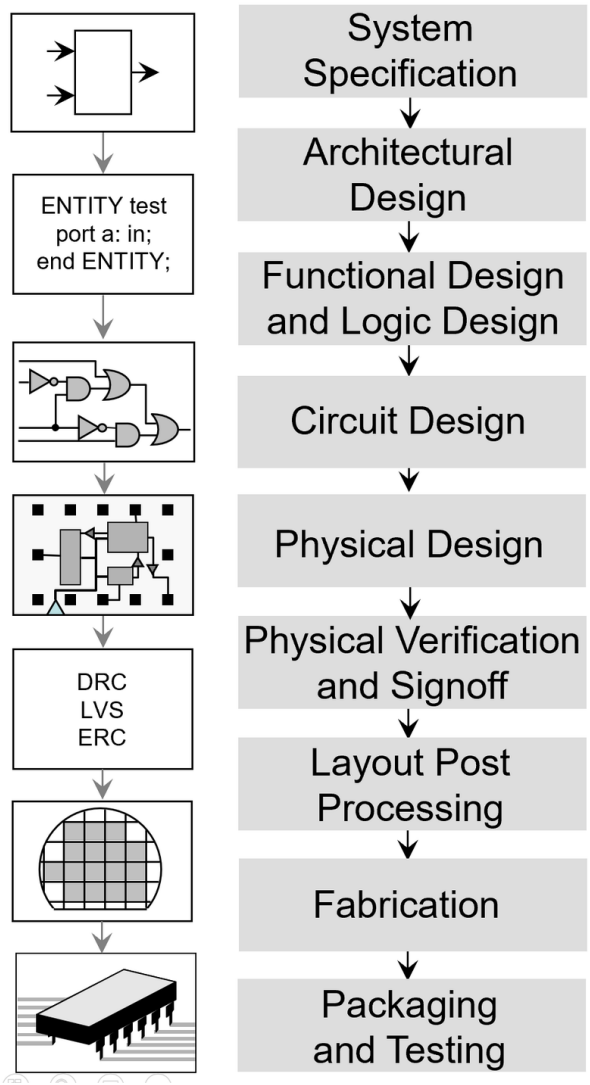


Figure 2.29: Digital IC Design Flow[10]

Digital IC design flow comprises of broadly three steps which corroborate with the diagram above:

1. Front-End Design
2. Back-End Design
3. Fabrication and Packaging

2.4.1 Front-End Design

In the front-end design, system and architectural details are specified and circuits are developed. In terms of digital circuits this corresponds to digital circuit design and verification, integration of Design for Test (DFT) structures [67], whereas for the analog circuitry design this would translate into topology selection [68] and subsequent simulation and analysis of the SPICE model [69] [**?**] in the simulator.

2.4.2 Back-End Design

The back end is concerned chiefly with Physical design and Sign-off checks. Physical design, atleast for digital IC, is concerned with Partitioning, Chip-Planning, Placement, Clock Tree Synthesis, Signal Routing and finally the Timing Closure [70]. The sign-off checks on the other hand is the collective name given to a series of verification steps that the design must pass before it can be taped out. It includes Layout versus Schematic (LVS) ensures that there is a match between netlist in present in both layout and the schematic. Second major check is the Design Rule Check (DRC) which determines whether the layout satisfies certain rules specified by the layout team or not. [71]

2.4.3 Fabrication and Packaging

The final stage after physical synthesis and verification is the manufacturing of the IC, which entails semiconductor wafer production followed by creation of design structures on the semiconductor die. Some aspects here include placement of Process control monitor [72] and Reliability control monitor [73] onto semiconductor wafer die to monitor the process and yield of semiconductor.following fabrication the next step is semiconductor packaging and test performed on it including moisture, stress and solder-ability checks [74]. This is the final step after which semiconductor is shipped to the customer.

2.5 FPGA technology and Design Flow

Th field programmable gate array technology emerged in 1984 when Xilinx introduced XC2064 [75]. FPGA have become fundamental to modern electronics industry wherein they act as substitute for a full fledged ASIC design thereby saving on non-recurring engineering cost and the need for ASIC development expertise. This section will give a brief overview of FPGA technology as well as a System-on-chip which couples reconfigurable fabric with a processing unit and has become predominant architecture for embedded computing systems.

2.5.1 Generations of FPGA

The development and emergence can be broadly classified into four broad generations [76] including:

1. Age of Invention 1984–1991 : FPGAs were small and automated placement and routing were not considered essential. Manual design was often necessary. FPGAs were much smaller than the applications that users wanted to put into them and hence multiple-FPGA systems became popular while automated multi-chip partitioning software was identified as an important component of an FPGA design suite.

2. Age of Expansion 1992–1999 : Area Became Less Precious and Design Automation Became Essential. SRAM emerged as technology of choice with LUT as logic cell of choice.

3. Age of Accumulation 2000–2007 : Emergence of communication technology led to its fusion with FPGA technology in form of integration of high speed transceivers among other blocks. Moores law brought about an improvement terms of area, power consumption and performance.

4. Present Age : Mounting ASIC costs led to adoption of FPGA as a system-on-chip integrating all necessary components needed to develop a custom processing system. Sophistication of design tools including tools for network on chip design and monitoring temperature of SoC were introduced. FPGA in its reconfigurability has started to compete with other compute technologies including GPUs and mutli-core processor as platform of choice which are deemed as more economically feasible and easier to design with.

2.5.2 FPGA architecture fundamentals

Reconfigurable fabrics and their design is active area of research, most of the details of the FPGA fabric are classified and proprietary to the vendor but some common shared features can be highlighted including [77] :

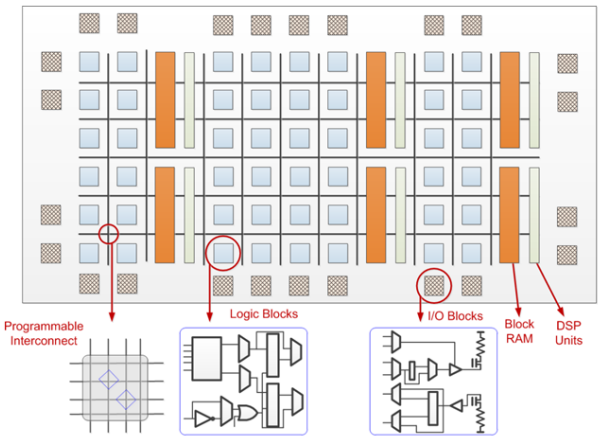


Figure 2.30: Generic FPGA Architecture[11]

Look-up Table

LUT [13] is the technology which mimics the function of a logic gate, more precisely the most fundamental logic cell in an FPGA is a LUT. A LUT consists of some number of inputs and one output. What makes a LUT powerful is that you can program what the output should be for every single possible input.

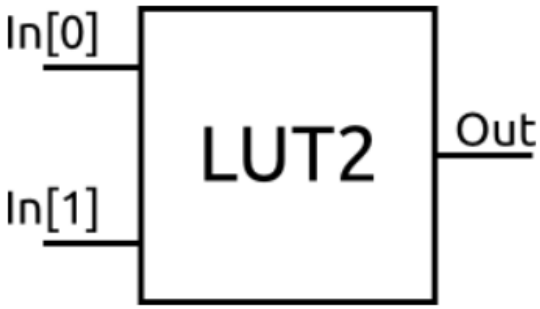


Figure 2.31: Look-up Table (LUT)[12]

A LUT consists of a block of RAM that is indexed by the LUT's inputs. The output of the LUT is whatever value is in the indexed location in it's RAM.

Configure Logic Block

These blocks contain the logic for the FPGA. In the large-grain architecture used by all FPGA vendors today, these CLBs contain enough logic to create a small state machine. The block contains RAM for creating arbitrary combinatorial logic functions, also known as lookup tables (LUTs). It also contains flip-flops for clocked storage elements, along with multiplexers in order to route the logic within the block and to and from external resources. The multiplexers also allow polarity selection and reset and clear input selection.

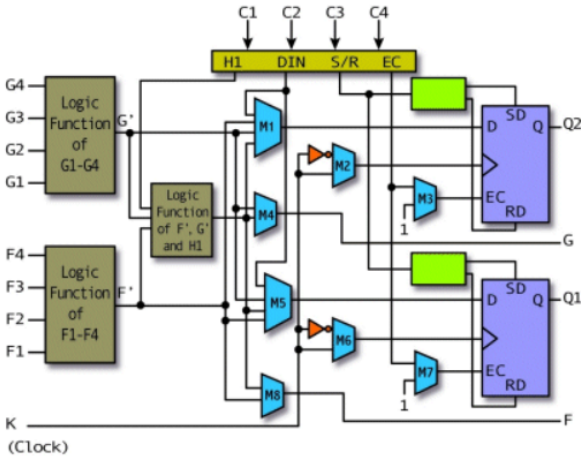


Figure 2.32: Configure Logic Block[13]

Interconnects and Routing Matrix

Programmable Interconnect are long lines that can be used to connect critical CLBs that are physically far from each other on the chip without inducing much delay. Theses long lines can also be used as buses within the chip.

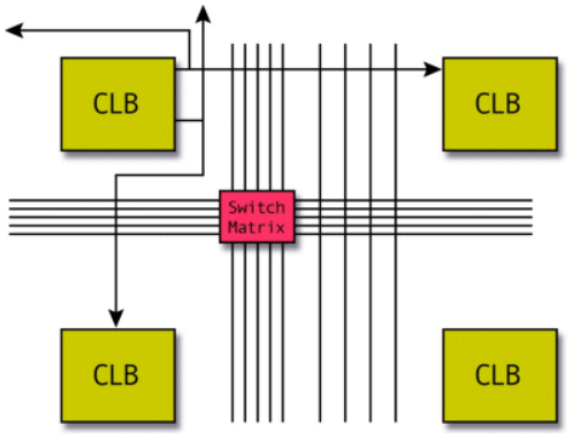


Figure 2.33: Interconnect and Routing matrix[13]

There are also short lines that are used to connect individual CLBs that are located physically close to each other. Transistors are used to turn on or off connections between different lines. There are also several programmable switch matrices in the FPGA to connect these long and short lines together in specific, flexible combinations.

I/O Block

A Configurable input/output (I/O) Block is used to bring signals onto the chip and send them back off again. It consists of an input buffer and an output buffer with three-state and open collector output controls.

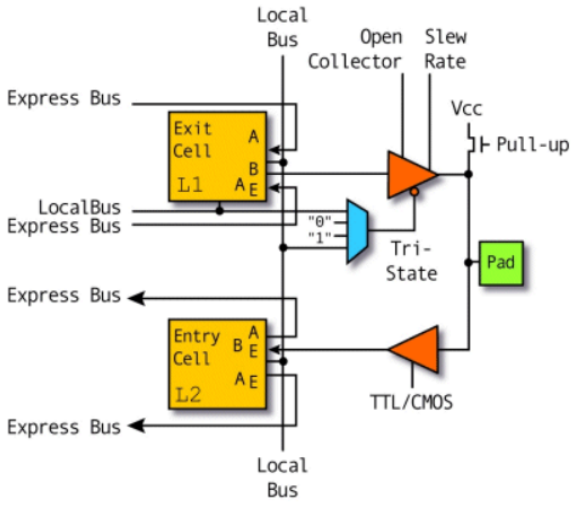


Figure 2.34: FPGA I/O Blocks[13]

Other Logic Features

Some other common FPGA features include [78], Digital Clock Manager DCM is used to perform Clock Phase shift, De skew, Clock divider and frequency synthesis. Multiplier Block implement dedicated 18×18 multiplier with Signed and unsigned operation. Block RAM is a dedicated memory implement dual port 16kb memory. Some FPGAs for specialized applications also include PCI terminations and DSP blocks for efficient computations.

2.5.3 FPGA design flow

The FPGA design flow [79] can be summarized by the diagrams below:

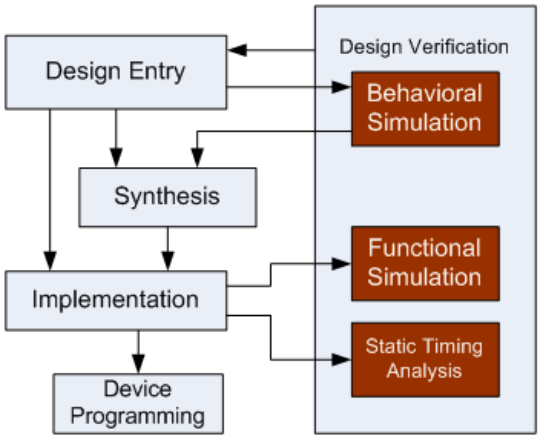


Figure 2.35: FPGA Design Flow[14]

Design Entry

There are different techniques for design entry including Schematic based, Hardware Description Language or a combination of both. Selection of a method depends on the design and designer. If the designer wants to deal more with Hardware, then Schematic entry is the better choice. When the design is complex or the designer thinks the design in an algorithmic way then HDL is the better choice. Language based entry is faster but lag in performance and density.

Synthesis

The process which translates VHDL or Verilog code into a device netlist formate. i.e a complete circuit with logical elements(gates, flip flops) for the design. Synthesis process will check code syntax and analyze the hierarchy of the design which ensures that the design is optimized for the design architecture, the designer has selected. The resulting netlist(s) is saved to an NGC(Native Generic Circuit) file (for Xilinx® Synthesis Technology (XST)).

Implementation

This process consists of three steps:

1. Translate : process combines all the input netlists and constraints to a logic design file. This information is saved as a NGD (Native Generic Database)

file. This can be done using NGD Build program. Here, defining constraints is nothing but, assigning the ports in the design to the physical elements (ex. pins, switches, buttons etc) of the targeted device and specifying time requirements of the design. This information is stored in a file named UCF (User Constraints File).Tools used to create or modify the UCF are PACE, Constraint Editor etc.

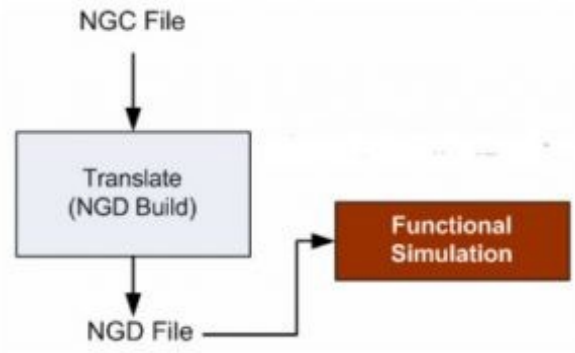


Figure 2.36: FPGA Design Flow[14]

2. Map process divides the whole circuit with logical elements into sub blocks such that they can be fit into the FPGA logic blocks. That means map process fits the logic defined by the NGD file into the targeted FPGA elements (Combinational Logic Blocks (CLB), Input Output Blocks (IOB)) and generates an NCD (Native Circuit Description) file which physically represents the design mapped to the components of FPGA. MAP program is used for this purpose.

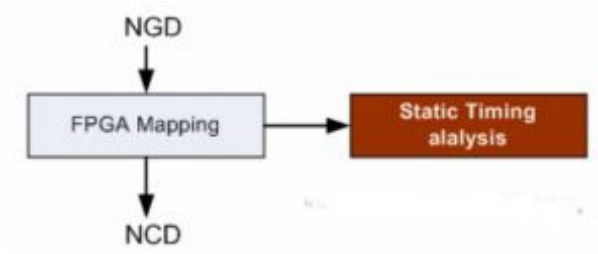


Figure 2.37: FPGA Design Flow[14]

3. Place and Route PAR program is used for this process. The place and route process places the sub blocks from the map process into logic blocks according to the constraints and connects the logic blocks. Ex. if a sub block is placed in a logic block which is very near to IO pin, then it may save the time but it may

effect some other constraint. So trade off between all the constraints is taken account by the place and route process. The PAR tool takes the mapped NCD file as input and produces a completely routed NCD file as output. Output NCD file consists the routing information.

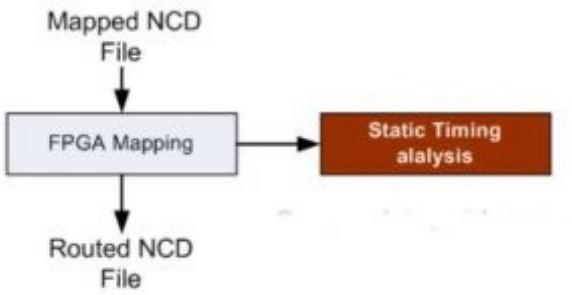


Figure 2.38: FPGA Design Flow[14]

Device Programming

Now the design must be loaded on the FPGA. But the design must be converted to a format so that the FPGA can accept it. BITGEN program deals with the conversion. The routed NCD file is then given to the BITGEN program to generate a bit stream (a .BIT file) which can be used to configure the target FPGA device. This can be done using a cable. Selection of cable depends on the design.

Design Verification

Verification can be done at different stages of the process steps:

1. Behavioral Simulation (RTL Simulation): This is first of all simulation steps; those are encountered throughout the hierarchy of the design flow. This simulation is performed before synthesis process to verify RTL (behavioral) code and to confirm that the design is functioning as intended. Behavioral simulation can be performed on either VHDL or Verilog designs. In this process, signals and variables are observed, procedures and functions are traced and breakpoints are set. This is a very fast simulation and so allows the designer to change the HDL code if the required functionality is not met with in a short time period. Since the design is not yet synthesized to gate level, timing and resource usage properties are still unknown.

2. Functional simulation(Post Translate Simulation) : Functional simulation gives information about the logic operation of the circuit. Designer can verify the functionality of the design using this process after the Translate process. If

the functionality is not as expected, then the designer has to made changes in the code and again follow the design flow steps.

3. Static Timing Analysis : This can be done after MAP or PAR processes Post MAP timing report lists signal path delays of the design derived from the design logic. Post Place and Route timing report incorporates timing delay information to provide a comprehensive timing summary of the design.

2.5.4 System-on-chip

A System-on-chip [80], is essentially an integrated circuit or an IC that takes a single platform and integrates an entire electronic or computer system onto it. It is, exactly as its name suggests, an entire system on a single chip. The components that an SoC generally looks to incorporate within itself include a central processing unit, input and output ports, internal memory, as well as analog input and output blocks among other things. Depending on the kind of system that has been reduced to the size of a chip, it can perform a variety of functions including signal processing, wireless communication, artificial intelligence and more.

System-on-chip Elements

- To begin with, a system on chip must have a processor at its core which will define its functions. Normally, an SoC has multiple processor cores. It can be a microcontroller, a microprocessor, a digital signal processor, or an application specific instruction set processor.

- Secondly, the chip must have its memories which will allow it to perform computation. It may have RAM, ROM, EEPROM, or even a flash memory.

- The next thing an SoC must possess are external interfaces which will help it comply with industry standard communication protocols such as USB, Ethernet, and HDMI. It can also incorporate wireless technology and involve protocols pertaining to WiFi and Bluetooth.

- It will also need a GPU or a Graphical Processing Unit in order to help visualize the interface.

- Other stuff that an SoC may have includes voltage regulators, phase lock loop control systems and oscillators, clocks and timers, analog to digital and digital to analog converters, etc.

- Internal interface bus or a network to connect all the individual blocks

System-on-Chip Advantages

- System-on-Chip are: power saving, space saving and cost reduction
- System-on-Chip are also much more efficient as systems as their performance is maximized per watt
- Systems on chip also tend to minimize the latency provided the various elements are strategically placed on the motherboard in order to minimize interference and interconnection delays as well as speed up the data transmission process

2.6 Processor Technology

The processor technology traces its origins back to 1971 when Intel introduced the processor 4004 [81]. In essence, processor can be though as being an programmable state machine as such the control unit becomes programmable and the operations performed on the data-path can be adjusted, modified and altered via the memory or instructions stored in the control circuitry [82].

2.6.1 Computer Classification and Organization

Computer Classification

The most simple classification of the computing structure can be in terms of being control oriented or data-flow oriented [83]. The most obvious instance of control oriented architecture is that of the Von Neumann machine wherein the program counter keeps track of the instructions being processed by the machine [84], indirect contrast is the data-flow or stream architecture which completely lacks the program or arbitrating logic and hence the input to the machine drives the logic itself resulting in non-determinism in execution [85], note that most signal processing architectures at hardware level are stream processing based for example the digital filters. The control oriented paradigm can again be classified in terms of data processing unit and central processing unit and the reconfigurability in the individual units of the design, wherein the Data Processing Unit features reconfigurability of electronic circuitry as per the instruction provided [86], Systolic array is classic example of a data processing architecture [87] [88] while the Central processing unit is more of state machine implementing instructions in a certain sequence.

A more general classification commonly used is that of the Flynn's taxonomy [89] wherein all the computational machines can be thought to belong to one of the four categories including:

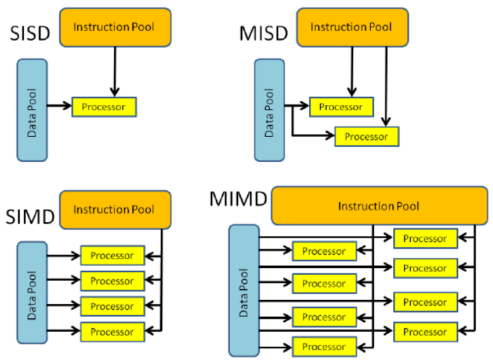


Figure 2.39: Flynns Taxonomy[15]

Since the most common and prevalent form of computing architecture is the Von Neumann paradigm with the program counter and control circuitry present to control the execution of computation, within the scope of this thesis we will only consider the distinct organizations of the Von Neumann machine or the computer organization. Note that one reason for the popularity of CPU is that they are general purpose and as such most real-life computation is assumed to be control oriented as a result.

Computer Organization

Computer Organization is concerned with the structure and behaviour of a computer system as seen by the user [90]. In total, there are three distinct modes and methods into which the processor organization can be classified, namely, Accumulator machine, Stack machine or the Register machine. [91]

An accumulator machine, also called a 1-operand machine, or a CPU with accumulator based architecture, is a kind of CPU where, although it may have several registers, the CPU mostly stores the results of calculations in one special register, typically called "the accumulator". Almost all early[clarification needed] computers were accumulator machines with only the high-performance "supercomputers" having multiple registers. [92] Most traditional accumulator machines have very few registers, and many only have a single 'general-purpose' register (by today's standards). On the PDP-8, for example, there are a few registers, but the only one you can access directly is the AC. The vast majority of instructions (not that 'vast' applies to a design with eight instructions) operate on the AC. The only other object to operate on is memory, so most instructions have a single field (a 7-bit address). The data transfer direction is implicit in the instruction (think load/store). [93]

In the stack machine, data is available at the top of the stack by default. The stack acts as a source and destination, push and pop instructions are used to access instructions and data from the stack. There is no need to pass the source and destination address because the default address is top of the stack. In the stack machine, there is no need to pass explicit addresses in the instruction. Therefore the instruction format consists only of the OPCODE (Operation Code) field. This instruction format is known as Zero address instruction. PDP-11, Intel's 8085 and HP 3000 are some of the examples of the stack organized computers. [94] Note that some virtual machines are totally stack based due to ease of programming and writing a compiler. [95] [96]

When we are using multiple general-purpose registers, instead of a single accumulator register, in the CPU Organization then this type of organization is known as General register-based CPU Organization. In this type of organization, the computer uses two or three address fields in their instruction format. Each address field may specify a general register or a memory word. If many CPU registers are available for heavily used variables and intermediate results, we can avoid memory

references much of the time, thus vastly increasing program execution speed, and reducing program size. [97]

Note all of the machines discussed in this section are Turing complete [98] and as such modern processor can be considered as being a fusion of all the above three configurations but is predominately referred to as being register machine. Hence a s result the stack functionality of the code directly comes from the processor itself making it difficult for example to change the direction of growth of the stack whereas the heap functionality is generally provided by runtime system.

2.6.2 Processor as Register Machine

The modern computer organization which predominates the market is that of register machine, in which the control logic performs its operations on the registers present on the data-path and hence the register machine model predominates. Almost all micro-controller based circuitry is based on the Von Neumann paradigm. On a simple register machine, any register can be used for any purpose with the exception of a special program counter which is used to track the progression of the program [99], although modern processors have a given ABI which specifies the sequence of the rules with which the registers can be manipulated or accessed. Some common registers in the processor include:

- Program Counter: A program counter (PC) is a CPU register in the computer processor which has the address of the next instruction to be executed from memory. It is a digital counter needed for faster execution of tasks as well as for tracking the current execution point. A program counter is also known as an instruction counter, instruction pointer, instruction address register or sequence control register. [100]

- Link Register: A link register is a special-purpose register which holds the address to return to when a function call completes. This is more efficient than the more traditional scheme of storing return addresses on a call stack, sometimes called a machine stack. The link register does not require the writes and reads of the memory containing the stack which can save a considerable percentage of execution time with repeated calls of small subroutines. [101]

- Shadow Register: Shadow register is a register devised within the micro-controller for purpose of holding certain data to be used later. The name "Shadow" implies to duplicate some value and use it again - so it wont get lost.[102]

- Stack Pointer: The Stack Pointer is a register which holds the address of the next available spot on the stack. The stack is a area in memory which is reserved to store a stack, that is a LIFO (Last In First Out) type of container,

where we store the local variables and return address, allowing a simple management of the nesting of function calls in a typical program. [103]

- Invisible Register: Since these types of register cannot be accessed directly by a program they are called invisible registers. The program invisible registers are used to access and specify the address tables of global and local descriptor tables. The Global Descriptor table register contains the limit and the base addresses for the descriptor table. The same applies for the Interrupt descriptor table register.although some of these registers are accessed by the system software. [104] [105]

All the Register machines which are used for computation have a format of ordering the bits in byte, word or nibble and hence can be classified as being Little Endian or Big Endian. Most modern processors give an option to switch between the two options by setting the internal processor flag. [106]

2.6.3 Computer Architecture Fundamentals

The modern computer architecture, based on register machine following predominately the Von Neumann model of computer, can be broadly classified into the three distinct elements including Instruction Set Architecture, Micro-Architecture and Macro-Architecture (which has emerged due to power, memory wall faced by the modern computing) [107]. Note that modern processor architectures also integrate some other crucial elements and systems including Interrupt mechanism [108], Processor Timing Subsystem[109], Hardware Context Switching Support[110] and MMU based Virtual memory subsystem [111] (utilizing special table or hardware features within the processor[112]) in parallel to code but it is beyond the scope of discussion here. Some basic aspects of the modern computer architecture are as follows:

Instruction set Architecture

Instruction set specifies the instructions available for the processor to manipulate the information in the registers and the main memory via the use of program stored in the memory. It generally covers the following details including: [113] [114]

1. Instruction types and encodings: Instruction types refers to the various constructs available to alter the flow of information in program or store intermediate values, they include Arithmetic and logic instructions, Data transfer instructions and Control flow instructions. Instruction encoding refers to the way in which operands and their addresses are presented to a processor, for example, in 3 address machine, the encoding has to specify destination, first source operand, base and index.

2. Number of Operands: Operand is any element capable of being manipulated. Overall instruction set specifies the number of operands that can be there in a machine with there being 0-operand machine (also known as stack machine) , 1-operand machine (also known as accumulator machine), 2-operand machine and 3-operand machines.

3. Addressing mode: It refers to the different ways the address of an operand in memory is specified and calculated. Some common addressing modes include Register, Immediate, Direct, Register Indirect, Displacement, Memory Indirect, Indexed, Auto-increment and Auto-decrement. [115]

4. Size of register file: The number of registers in a register file of a processor is also decided by ISA, for example too few register will result in a high register pressure leading to highly complex compilers where as too many registers will result in wastage of die area as well as power in terms of storage element used to develop the memory elements.

5. Instruction length: The size or length of an instruction varies widely, from as little as four bits in some micro-controllers to many hundreds of bits in some VLIW systems. A RISC instruction set normally has a fixed instruction length, whereas a typical CISC instruction set may have instructions of widely varying length.

The two classical paradigms of the instruction set architecture categorization include RISC (Reduced Instruction Set Computing) and CISC (Complex Instruction Set Computing) [116], wherein the RISC machines have single instruction for a single operation resulting in faster computation but large program size with CISC lumping up multiple instructions into one to safe on program memory at the expense of complicated instruction set. Note that RISC architectures are also load-store, have fix instruction size and enable pipeling (since every instruction take equal cycles or time) whereas CISC architectures generally lack such features, although this simple classification is getting increasingly redundant. Note another classification can be in terms of the ways data moves from one register to the other including register-register (also known as load-store architecture which predominates RISC machines), register-memory or memory-memory.

Abstract machine which don't have any hardware like the Java Virtual Machine also have Instruction set with which they can be programmed. They usually utilize the stack computer organization as it then becomes easy to create a compiler for such a machine.

Microarchitecture

A micro-architecture (sometimes written as "micro-architecture") is the digital logic that allows an instruction set to be executed. It is the combined implementation of

registers, memory, arithmetic logic units, multiplexers, and any other digital logic blocks. All of this, together, forms the processor. [117]

A micro-architecture combined with an instruction set architecture (ISA) makes up the system's computer architecture as a whole. Different micro architectures can implement the same ISA, but with trade-offs in things like power efficiency or execution speed. For Instance, Intel, AMD and ARM have multiple families of micro-architectures with which to implement a particular instruction set. [118]

Although micro-architecture covers a massive scope with multiple topics, some cardinal aspects include:

1. Cache: A CPU cache is a hardware cache used by the central processing unit (CPU) of a computer to reduce the average cost (time or energy) to access data from the main memory. A cache is a smaller, faster memory, located closer to a processor core, which stores copies of the data from frequently used main memory locations. Most CPUs have a hierarchy of multiple cache levels (L1, L2, often L3, and rarely even L4), with separate instruction-specific and data-specific caches at level 1. There can be many types of caches depending on requirements including direct mapped and set associative. [119]

2. Pipeline: Pipe-lining is a technique for implementing instruction-level parallelism within a single processor. Pipe-lining attempts to keep every part of the processor busy with some instruction by dividing incoming instructions into a series of sequential steps performed by different processor units with different parts of instructions processed in parallel. The classical RISC pipeline consists of 5 stages including, Instruction fetch, Instruction execute, store in Memory and then write back. [120]

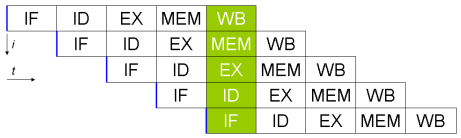


Figure 2.40: Classic RISC Pipeline

3. Branch Prediction: A branch predictor is a digital circuit that tries to guess which way a branch (e.g., an if–then–else structure) will go before this is known definitively. The purpose of the branch predictor is to improve the flow in the instruction pipeline. Branch predictors play a critical role in achieving high effective performance in many modern pipelined microprocessor architectures such as x86. [121]

4. Order-of-order Execution: out-of-order execution is a paradigm used in most high-performance central processing units to make use of instruction cycles

that would otherwise be wasted. In this paradigm, a processor executes instructions in an order governed by the availability of input data and execution units, rather than by their original order in a program. In doing so, the processor can avoid being idle while waiting for the preceding instruction to complete and can, in the meantime, process the next instructions that are able to run immediately and independently. [122]

5. Speculative Execution: Speculative execution is an optimization technique where a computer system performs some task that may not be needed. Work is done before it is known whether it is actually needed, so as to prevent a delay that would have to be incurred by doing the work after it is known that it is needed. If it turns out the work was not needed after all, most changes made by the work are reverted and the results are ignored. [123]

6. Register Renaming: Register renaming is a technique that abstracts logical registers from physical registers. Every logical register has a set of physical registers associated with it. While a programmer in assembly language refers for instance to a logical register accu, the processor transposes this name to one specific physical register on the fly. The physical registers are opaque and cannot be referenced directly but only via the canonical names. [124]

Note that although the above mentioned structures have all been abstracted, there is still a lot of interest in micro-architecture recently as the program as well operating system performance ultimately depend on how efficiently these were utilized over the course of writing a program. [125] Also modern vulnerabilities in the processor including Meltdown and Spectre frequently utilize shortcoming in implementation of speculative execution to mount an attack. [126] [127]

CPU Modes

Modern CPUs have multiple modes to support the multiple functions they perform and as such they have to be switched from one to another to fulfil a certain criterion. In this section, we will review some common CPU modes of Intel and ARM Architectures which are most predominant in the market.

The purpose of processor modes is to regulate access to memory and hardware resources so that a process initiated by a specific user can't access the memory of other processes or access hardware they don't have permission for. The operating system can add quite refined permissions, so users only have access to certain files, read-only access to certain files, or other granular rights. Further you might set up specific users and groups to grant the exact rights to processes like web servers to help protect you system from malicious hackers or program bugs causing havoc. [128]

The Intel Processor Modes [129] include:

1. Real mode (16 Bit, segmented) : This mode that's present since the old days of the 8086 is still the mode processors are booting in. At least BIOS/EFI code runs in it and when booting in legacy mode also bootloaders are running in 16 bit mode. Adresses are calculated as a pair of segment address and offset. A segment address is just a 16 bit value inside one of the segment registers (CS, DS, ES, FS, GS, SS) that simply gets multiplied by 16 - then the offset is added. The segments in use are:
   - The codesegment CS that's used in conjunction with the instruction pointer IP to address the next instruction to be fetched and executed. Normally CS is just updated by either a far-call, far-jump, interrupt or interrupt return.
   - The datasegment DS that's used when accessing memory without segment override except for some string functions that are using ES as destination and since the 80486 the additional segments FS and GS that might be used to access some memory locations using overrides.
   - The stack segment SS that's used in conjunction with the stack pointer SP to provide a downwards growing stack.

   These memory models have been important for compiler developers. On current compilers most of the time it's expected that code, data and especially stack segments overlap - on some machines code might be separate but stack and data is assumed to be flat. This is required to intermix local variables that are normally allocated on the stack and data on the heap. Since loading segment registers normally flushes some internal caches and takes longer than simply loading a general purpose register far pointer usage is also minimized by compilers whenever possible. There have been some differnt definition for memory models used by compilers in this mode:
   - tiny model used a single overlapping 64 KByt
   - data and code area (i.e. CS equal to DS, ES, FS and GS)
   - small model used one data segment and one code segment (non overlapping).
   - compact model had a single code but multiple data segments
   - medium used multiple code and a single data segment
   - large model used multiple code and multiple data segments that are swapped as required.
   - huge was the large memory model but single data structures have been larger than a segment and crossed segment boundaries.

   There are not many modern compilers that are capable of targeting real mode (which is especially a problem when it comes to development of firmware or

bootloaders). To circumvent these problems solutions like DOS4GW already switches into protected mode. One major compilers that support 16 bit mode was the Watcom C compiler.

2. Unreal mode (16 Bit, 4 GB data segment, 16 Bit code segment) : This is an undocumented 16 bit mode. It's entered by first entering 32 bit protected mode, loading a 4 GB spawning 32 bit data segment into DS,ES,FS or GS (or all of them) and switching back to real mode. This keeps the large data segments in place and allows (using 32 bit address overrides) to access the whole 32 bit memory space from 16 bit code. This mode is called that way since it has been used in the popular Unreal game.

   In the beginning this behaviour was a bug that has been exploited by developers but since it got somewhat popular it was also implemented on modern processors even up until today.

3. Protected mode (32 Bit) : This is the 32 bit mode people normally run their 32 bit systems in. The segment registers are just selectors into the global or local descriptor table (GDT / LDT). These descriptors might represent base, limit and a protection level as well as selector type (data, stack or code). Code can only be executed in code segments, etc. Normally modern compilers assume at least overlapping data and stack segments in a single flat address space. This allows them to use simple registers to contain data pointers onto the heap and local variables on the stack - as well as function pointers. There is no need for any register override. Some systems use registers FS and GS to reference into special areas like thread local storage (MS Windows) or a global offset table (Linux).

   Code segments (and also data segments) can be assigned one of four protection levels. Ring 0 is traditionally the kernel protection level. Code running in ring 0 can do anything and execute all operations (except when running inside a virtualization hypervisor). Ring 1 and 2 are not used really often and cannot execute privileged operations. Ring 3 is the typical user code level and can also not execute privileged instructions. Switching into other protection levels is normally done via an interrupt or an interrupt return. Also the sysenter/sysleave or syscall/sysret functions might be used for an protection level transition. Transition from a numerical lower level to higher level is also possible via a far jump.

   Additionally protected mode supports the usage of paging where an additional indirection layer is introduced. Using paging all virtual adresses can be remapped to other physical adresses using a page table using 4 kByte (default), 2 MByte or 4 MByte pages. Using this mechanisms and page address extensions one can even use a virtual 32 bit adress space to reference into a 36 bit physical address space. Paging also allows to implement virtual memory

by providing an pagefault interrupt in case a disabled page is accessed. It also supports two protection levels (ring 0 as supervisor mode and and rings 1-3 as user mode).

Protected mode might also be used segmented with different memory models as described above but since it makes the lives of compiler developers way easier the flat memory model where CS equals DS,ES,FS,GS and SS has become the most frequently used. This of course also requires some additional mechanisms to provide stack overflow bugs (that would be no problem when using a limited stack segment that would not overlap code or data regions - but then one would have to use different pointers for local variables and heap variables / constants when passing through code or perform some special arithmetic in such cases.

4. Virtual 8086 mode : Since some legacy code might be required on a system running in 32 bit mode there is a special mode, virtual 8086 mode. It runs 16 bit code in a 16 bit code segment under a 32 bit ring 0 implementation. Its sometimes used to run legacy programs on modern operating systems - and also sometimes used in bootloaders like FreeBSD BTX to run 16 bit BIOS code from 32 bit protected mode (one has to take some precautions there because some BIOS routines also do some tricks to access high memory regions).

5. Long mode (64 Bit) : This is the native 64 bit mode. Its similar to 32 bit mode and one has to switch through 32 bit protected mode to enter 64 bit mode. The main difference is that the flat adressing has been made the only choice for CS,DS and ES. Only FS and GS can use descriptors with a base that's not zero and a limit that's not the maximum. One cannot run Virtual 8086 tasks when running the processor in long mode but one can use 32 bit code segments for 32 bit protected mode applications (so a processor running in long mode can concurrently run 32 bit and 64 bit code).

6. System management mode (SMM) : This is a special mode that's normally only used by firmware developers. It's some kind of a special 32 bit protected mode - without any protection. It also triggers some hardware changes like removing mapping of some PCI device address space, etc. The code running in SMM normally hides from "normal" memory regions - they are not accessible in any way from normal 16 bit or 32/64 bit modes. One enters SMM only via triggering of an system management interrupt. Some code running in SMM might be a management engine, thermal control, remote management, etc. Normally one is not capable of injecting code into the SMM since it's thought of being part of the trusted and unmodifyable base of the system.

The ARM Processor Modes:

1. User – regular programs that can access the resources they have permission for.

2. FIQ – the processor goes into this mode when handling as fast interrupt. The operating system provides this code and it has access to all operating system resources.

3. IRQ – the processor goes into this mode when handling a regular interrupt. The operating system provides this code and it has access to all operating system resources.

4. Supervisor – when a user mode program makes an SVC Assembly instruction which calls an operating system services, the program switches to this mode, which allows the program to operate at a privileged level for the duration of the code.

5. Monitor – if you have an ARM processor with security extensions then this mode is used to monitor the system.

6. Abort – if a user mode program tries to access memory it isn't allowed, then this mode is entered to let the operating system intervene and either terminate the program, or send the program a signal.

7. Hyp – this is hypervisor mode that is an optional ARM extension. This allows the virtual hypervisor run at a more secure level than the operating systems it is virtualizing.

8. Undefined – is a user mode program tries to execute an undefined or illegal Assembly instruction then this mode is entered and the operating system can terminate the program or send it a signal.

9. System – this is the mode that the operating system runs at. Processes that the operating system considers part of itself run at this level.

Microcode

Microcode is the lowest specified level of processor and machine instructions sets. It is a layer comprised of small instruction sets, which are derived from machine language. Microcode performs short, control-level register operations, including multiple micro instructions, each of which performs one or more micro operations. [130]

Microcode and machine language differ. Machine language operates at the hardware abstractions upper layer. However, microcode deals with lower-level or circuit-based operations. Because microcode is usually embedded in hardware, it generally cannot be altered. Processor microcode is akin to processor firmware. The kernel is able to update the processor's firmware without the need to update it via a BIOS update. A microcode update is kept in volatile memory, thus the BIOS/UEFI or kernel updates the microcode during every boot. Processors from Intel and AMD may

need updates to their microcode to operate correctly. These updates fix bugs/errata that can cause anything from incorrect processing, to code and data corruption, and system lockups. It is very difficult to know for sure whether you need a microcode update or not, but it is not safe at all to just ignore them. You might not notice their effect and have precious data silently corrupted, or an important program silently misbehave. Or you could experience one of those unexplainable and infrequent software issues (such as kernel oops, application segfaults) or hardware issues (including sudden reboots and hangs). Releases of new microcode updates are more frequent on young processors, but the release of new microcode updates for older processors do happen. [131] Note that vulnerabilities like meltdown and spectre have also been dealt with micro-code updates as well. [132] Note that some RISC machines have eliminated the utilization of the microcodes in their design for speed-up purposes. [133]

Two common principles exist to pack control signals into microinstructions. This choice greatly influences the whole microarchitecture and has a huge impact on the size of microcode programs, including: [134]

- Horizontal Encoding. The horizontal encoding designates one bit position in the microinstruction for each control signal of all functional units. For the sake of simple logic and speed, no further encoding or compression is applied. This results in broad control words, even for small processors.

- Vertical Encoding. Vertically encoded microcode may look like a common RISC instruction set. The microinstruction usually contains an opcode field that selects the operation to be performed and additional operand fields. The operand fields may vary in number and size depend- ing on the opcode and specific flag fields. Bit positions can be reused efficiently, thus the microinstructions are more compact.

Macroarchitecture

In the recent era in computing, there has been a rapid shrinking in the die leading to power and memory walls being encountered by the system leading to diminishing gains in terms of speed and power efficiency. Hence there has been a move towards Multi-core or an emphasis on macro-architecture for the system to take advantage of inherent parallelism present in the program to speed up the computations. The predominance of multi-core system is as such that it is impossible to find a single core implementation of any design and there is even an interest in implementing traditionally ASIC implementations of design via its multi-core counterpart for example the 802.11 Wi-fi Receiver[135] and FFT Algorithm [136]. In this section, there will be a very brief overview of certain aspects of multi-core designs.

The basic laws which govern the multi-core system modelling include:

1. Amdahl's Law: Formula which gives the theoretical speed up in latency of the execution of a task at fixed workload that can be expected of a system whose resources are improved. [137]

2. Gustafson's Law: Gustafson's law (or Gustafson–Barsis's law) gives the theoretical speed up in latency of the execution of a task at fixed execution time that can be expected of a system whose resources are improved. [138]

3. Sun and Ni's Law: a memory-bounded speedup model which states that as computing power increases the corresponding increase in problem size is constrained by the system's memory capacity. In general, as a system grows in computational power, the problems run on the system increase in size. It is a generalization of Amdahl's and Gustafson's law. [139]

Homogeneous and heterogeneous cores : There are two basic paradigms of design in multi-core system including heterogeneous and homogeneous core system. A device that contains multiple cores with different types of instruction sets is referred to as heterogeneous. In contrast, homogeneous multi-core devices implement multiple identical cores. The current trend is to create homogeneous multi-core devices so as to keep the overall programming model simple [140], but a significant performance advantage can be obtained by using specialized cores and accelerators to offload the main cores. [141]

SMP (with and without NUMA) : The symmetric multi-processing (SMP) model is the classic model for designing multicore operating systems, such as Linux, where data is shared to a large extent and where a number of different locking mechanisms and atomic operations are used frequently for synchronization. The SMP model is a very simple model from the application perspective, but its implementation is complex and difficult to make correct at the operating system level. Moreover, it does not scale very well to more than 4–8 cores, especially not on an application level. A Non Uniform Memory Access System uses a design that has multiple bus/memory subsystems in an attempt to reduce the bottleneck of a single bus/memory system. In a NUMA design, every processor has access to all memory; however, each processor has faster access to a portion of the memory, and slower access to the rest. The difference in speed is normally no more than 1:3. In stark contrast is the UMA (uniform memory access) wherein the system slows down as the cores attempt to access the same resources leading to a bottleneck.

AMP Architecture: The asymmetric multi-processing (AMP) model uses an approach where each core runs its own isolated software system. These may even use different RTOSes. The advantages of an AMP system are that high performance is achieved locally and that it scales well. The trade off is that the system becomes difficult to manage and can become fairly static.

Processor affinity: Processor Affinity [142] is the term used for describing the rules for associating certain threads and certain processors. The operating system uses

Processor Affinity when the OS task scheduler assigns threads to run in processors. By default each process/ thread has affinity for every processor in the system.

Hyper-Threading: It is Intel's term for simultaneous multithreading (SMT). [143] This is a process where a CPU splits each of its physical cores into virtual cores, which are known as threads. For example, most of Intel's CPUs with two cores use hyper-threading to provide four threads, and Intel CPUs with four cores use hyper-threading to provide eight threads. Hyper-Threading allows each core to do two things simultaneously. It increases CPU performance by improving the processor's efficiency, thereby allowing you to run multiple demanding apps at the same time or use heavily-threaded apps without the PC lagging.

Generally, When designing a multi-core processing system, the following aspects are considered in depth: [144]

1. How do they boot? one core is chosen as the boot processor and sets up the system so that the other processors can use a shorter initialization sequence and to keep the different processors from not interfering with each other during boot.

2. How do they communicate through memory? different cores mostly communicate through shared memory. Most multiprocessors these days have coherent shared memory, so that changes to memory made by one core will be visible to the others. This is convenient, but sometimes the changes are not seen in the same order by different cores, so an elaborate system of locks and fences and complexity under the name of "memory ordering rules" is needed. The different cores are really co-equal and just run programs independently. Any coordination is a matter for software, either the OS or application code. Modern multi-core systems have mainly two models of IPC, the Shared Memory model and the Message Passing (or message queue) model. Since any core can have full access to the device memory, senders and receivers can also communicate data by exchanging pointers only without actually exchanging data. In this case, the sender writes data to a specific memory, notifies the receiver and sends the pointer. The receiver then accesses the memory and when it finishes with it, the receiver notifies the sender. In general, there are many mechanisms to implement an IPC. The most common are, Shared memory, TCP, Named pipe, File mapping, Mail slots and MSMQ (Microsoft Queue Solution). [145]

3. How do they interrupt each other? interprocessor interrupts, so that one processor can get the attention of another. These are used by modern operating systems for many purposes, but as an easy example, if an application running on one core has gotten wedged into a tight loop, there has to be a way for another core to break it loose. IPIs are also used for scheduling, cache flushes, TLB shootdowns, and many other purposes. On Intel processors, interprocessor interrupts are provided by the APIC (advanced programmable interrupt controller IIRC) that is on-chip.

> It is certainly possible to run a multiprocessor without IPI, you can just have a periodic local timer interrupt on every core that checks a communications area in memory, but it would slow things down.

As seen from the enumeration and design considerations above, there are multiple ways of designing a multi-core system, hence this can result in scalability issues and create problem with operating system and software deployments.Intel, as a result, has introduced multi-core processor specifications unifying the design aspects of multiple core systems thereby making it easy to port Operating Systems and System Software from one architectural design to the other. [146]

Memory Management Unit (MMU)

Modern Processors also integrate Memory Management Unit (MMU) [147] [148] in their design which is also a requirement for booting most Operating Systems and enables separation of processes into distinct address spaces. Since the Processor utilized over the course of this thesis is a simple Mi-V microprocessor with absence of MMU, we will not consider it in detail. Note that one major reason that GPUs are not suited for general purpose computing and will hence be restricted to acceleration niche is that they lack the presence of optimized MMU based virtual memory subsystems among other things which is integral for modern operating systems as well as general purpose computing. [149]

2.6.4 Processor Performance and Simulation

In this section, we will review the basics basics of quantification for the Processor based systems as well as simulation of System-on-chips for their performance analysis.

Processor and Instruction Set Performance metrics

The instruction set performance of a machine can be measured in a number of ways including: [150] [151] [152]

- Execution and cycle counts of a benchmark program to determine how long does it take for different instruction set to processor a given program.
- System, Board and Processor powers are other measures to determine the efficiency of the processors.
- Instruction count(number of instructions that get executed for a particular task, algorithm, workload, or program) , mix(refers to The blend of instruction types in a program) and length(the number of machine code instructions required to execute a section of a computer program.) are other measures which determine the efficiency of a given instruction set.

- Other means of measuring a processor is by using performance metrics including: MIPS, MFLOPS, SPEC and QUIPS, with which a processors performance is quantified.

Note, in all the above measures we assume that micro-architectural details are either constant or don't significantly impact the processor otherwise the process becomes more complex with multiple variables.

Processor Simulation Methods

In this section we will present an overview of two major processor simulation methods including:

- Instruction set simulator based model utilize SystemC or Gem5 to generate a transaction level model of the system to be utilized for cycle accurate simulation of the system. The resulting system is called a virtual prototype which can then be used for simulation, verification or performance analysis. [153]
- Source code annotation method on the other hand relies annotating the timing properties of the processor in form of WCET of the basic blocks [154], resulting in very fast simulation and verification of the system provided that a proper timing model of the processor as well as its micro-architecture and interconnects can be generated. [155]

2.6.5 Programming Model of Processor

The programmers view of the processor is generally considered the memory model of the processor. In this section, we will review the classical address spaces of the processor, memory models introduced by the programmers as well as more recent division of programming models

Classical Address Spaces of Processor

Address space refers to the addressable memory accessible to the processor as simple manipulation of the processor register files is not sufficient to program a system and an external addressable memory is needed. They can be broadly classified into three primary types including Von Neumann, Harvard and modified Harvard address spaces, Although some special hardware platforms like DSPs might make distinction like reserving separate address space for DSP coefficients, thereby inducing an implicit memory model at hardware level. The explanation of three basic address space models are as given below:

1. Harvard Architecture : In Harvard architecture there are physically separate memories for the instruction and data sections of the program,i.e. there are two

distinct address space , the prime example of this are AVR micro controllers. A self modifying code is not possible in Harvard machines [156]

2. Von Neumann Architecture :we have one physical memory, which contains both data and code; we also have one bus from our CPU to our memory, which technically means that we cannot read both data and code at the same time (though from developer's perspective, the latter effect is not really observable). On the other hand, we have one address space, so we can have pointers to the data, and pointers to the code; also we can write the code as data, and then execute it as a code, the prime example of Von Neumann architecture is 8088 .A self modifying code is possible in Von Neumann architecture. [157]

3. Modified Harvard Architecture : A modified Harvard architecture machine is very much like a Harvard architecture machine, but it relaxes the strict separation between instruction and data while still letting the CPU concurrently access two (or more) memory buses. The Modified Harvard Architecture is a variation of the Harvard computer architec- ture that allows the contents of the instruction memory to be accessed as if it were data. The most common modification includes separate instruction and data caches backed by a common address space. The prime example of modified Harvard architecture are 8051 and TMS320C24x. A self-modifying code is possible in modified Harvard architecture as entire address space is available for addressing. [158]

Note that the earliest computers had a fixed program rather than a Stored Program model, hence as a result there is a possible of self-modification of program although it has largely fallen into disuse but still modern stored programs are liable to mischievous manipulation like control flow hijacking as will be seen in later sections.[158]

Memory Model of Processor

In classical age, there was just an address space available to the programmer and he would draw and place code structures on to it himself and there was only a little notion of a standard memory model [159] [160], with the basic idea being a pointer in form of program counter to the starting address of program in memory. In this manner we can have multiple memory models including: [161]

1. flat address space : A "flat" address space of undistinguished words (not bytes until very late in the game) that began at address zero and continued upward to the "top" of available memory. Some portion of memory at the "bottom" or "top" of memory would be reserved for some sort of "loader", with the rest available to the single executing program. One program at a time ran, with the program able to use all but that small reserved RAM area.Note extended machines like Operating system have MMU virtual memory allow compiler to

place object as per the memory model of language and executable, all while providing a flat memory model for the compiler or runtime to place program memory objects on.

2. Paged Memory Model : A "paged" model assumes the address space is divided into "pages" of a certain size (though in some cases several different page sizes are supported). Generally the page size is somewhere between 256 bytes and 64k bytes (always a power of 2). It is also assumed that the hardware contains some sort of address translation support so that "logical" addresses (addresses in the program's "address space") can be mapped to "physical" addresses (addresses in RAM).

3. Segmented memory model [162] : The original memory model of 8088 and x86 machines, A segmented memory model divides the system memory into groups of independent segments referenced by pointers located in the segment registers. Each segment is used to contain a specific type of data. One segment is used to contain instruction codes, another segment stores the data elements, and a third segment keeps the program stack. Some segments include:
   a) Data segment It is represented by .data section and the .bss. The .data section is used to declare the memory region, where data elements are stored for the program. This section cannot be expanded after the data elements are declared, and it remains static throughout the program. The .bss section is also a static memory section that contains buffers for data to be declared later in the program. This buffer memory is zero-filled.
   b) Code segment It is represented by .text section. This defines an area in memory that stores the instruction codes. This is also a fixed area.
   c) Stack This segment contains data values passed to functions and procedures within the program.

4. C memory model : The modern standard upon which all the system programming level is performed. Although mentioned here as a memory model, it can in fact be considered as a programming model [163], the sections of which are handled by the runtime of a given system as per the system ABI.

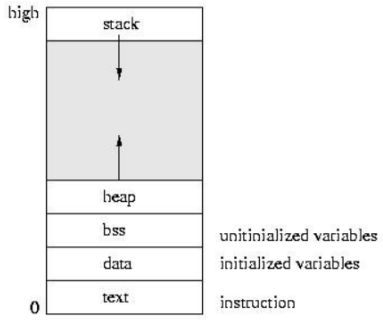


Figure 2.41: C Memory Model

Programming Model, Execution Model and Memory Model

With the advent of high level languages as well as persistence of classical computing concepts regarding the memory, there can be there three broad categories of programming memory models.

1. Memory Model [164] : The term is more of a classical explanation for the memory and as such refers to the actual physical memory access and organisation methods that exist in the system. Note at time we refer to C programming language has having a memory model as it is very close to hardware, however, the memory model of C is actually a programming model the execution model for which is provided by runtime of the language. In the classical age, all three model including Memory Model, Programming Model and the Execution Model were equivalent.

2. Programming Model : Programming model [165] generally refers to users abstractions developer atop a more concrete execution model, fro example operating system FreeRTOS can be classified as being a Programming Model built a top C Programming language execution model. Normally these days, a programming language has both a programming model and an execution model, although execution model and programming models can differ in cases when a library call is providing a distinct programming model as in the case of Pthreads.

3. Execution Model : Execution model refers to how the program which in programmers perspective might me executing differently actually executes in the hardware. Execution model is often provided either by runtime or by an extended machine in form of operating system, for example, the code in Linux executes sequentially unless the programming is mapped to multiple kernel threads wherein there is actual parallelism in code with regards to execution model.[166] Also much like Pthreads example before, the resulting programming model is sequential although its execution is still sequential, although

C++11 has integrated parallelism in its abstract machine whereby making both programming and execution model parallel without taking into account operating system. [167]

2.6.6 Hardware Abstraction Layer and Device Drivers

A device driver is a software module responsible for managing low-level I/O operations for a particular hardware device. Device drivers can also be software-only, emulating a device that exists only in software, such as a RAM disk or a pseudo-terminal. Such device drivers are called pseudo device drivers and cannot perform functions requiring hardware (such as DMA).

A device driver contains all the device-specific code necessary to communicate with a device and provides a standard I/O interface to the rest of the system.

Hardware Abstraction Layer

The increasing sophistication of the processor technology as well as the motherboard has introduced the utilization of pre-existing software to control the peripherals on the motherboard as well as the components placed on the chip itself in the form of firmware know as Hardware Abstraction Layer which plays role of device driver for on-board components. The role of the HAL[168] can be classified in three distinct ways:

The architecture HAL abstracts the basic CPU architecture and includes things like interrupt delivery, context switching, CPU startup etc.

The platform HAL abstracts the properties of the current platform and includes things like platform startup, timer devices, I/O register access and interrupt controllers.

The implementation HAL abstracts properties that lie between these two, such as architecture variants and on-chip devices.

The boundaries between these three HAL levels are necessarily blurred.

Device Driver

The device driver on the other hand generally refers to the interfaces to the components off the board like any device connected to main board via the UART. The Device drivers can be broadly classified [169] into two distinct categories including:

A Character Device is one with which the Driver communicates by sending and receiving single characters (bytes, octets).Examples for Character Devices: serial ports, parallel ports, sounds cards.

A Block Device is one with which the Driver communicates by sending entire blocks of data. Examples for Block Devices: hard disks, USB cameras, Disk-On-Key.

Note that given the criticality of device driver interface software, it is an active area of research with a lot of formal methods devised to ensure integrity of the software. [170]

2.7 Program and Compiler Theory

Program refers to a set of instructions which instructs the processors to implement a certain functionality. In this section, all the basic elements needed to place the desired program into the address space of the processor to be executed will be introduced, also there will be a discussion on the structure of the programs as modern software systems often require strict latency requirements as well as security assurance in their design. In modern program development the primary objective is to create a source code which is seamlessly portable across multiple platforms, it is with this aim that modern technologies like Java Virtual Machine as well as containerization have permeated. In this section, the bare basic technologies for creating a firmware for a system level program shall be discussed including their terminologies and details. Compiler, on the other hand, can be simply described as a program that merely translates program form one language to the other and as such there can be source to source compilers as well.

The concept of program is ubiquitous in modern computing as such Compiler, linkers and even entire Operating systems are merely programs running or either creating other programs. [171]

2.7.1 Source Code, ABI, API and Shared/Dynamic Libraries

The firmware or the bare metal program as well as Operating System level System programs utilize some basic software elements to bring about desired functionality i.e timer, mutexes or I/O Scanning. In this Section, we will discuss some of these elements including Source Code, Application Peripheral Interface, Application Binary Interface, Runtime System and concept of Libraries.

In modern programming system, the abstraction of source code predominates, i.e, when we program, we program for an abstract machine with a given memory model [172]. The source code also in itself should be highly portable form one architecture to another, hence as such architectural dependencies are to be handled by programmer abstract structures like the runtime libraries or the compiler ABI provided.

Source Code

The basic element of the software is the source code which is written in a high level language and attempts to describe a given functionality in terms of the syntax of the language being utilized. Some common procedures which are performed on the source code include the code re-factoring to make code more maintainable, Unit testing to ensure proper functionality or Test-Driven Development to create an optimum software [173]. Programmers use syntax and semantics of the programming language to create a control and data flow structures in their code to attain a desired goal or functionality in their code [174]. Programming languages can be classified in

a number of ways based on their paradigm and semantics. Normally, Software also comprises of multiple source code files whose architecture is documented in UML with common design patterns like the MCV being used to describe the functionality of the design [175]. Generally, Programming languages, much like a processor with infinite memory are considered as being Turing complete. [176]

Also when designing source code, we program for an abstract machine with a given memory model and as such even simple constructs like pointers need not to have any address or physical significance. In fact, the standard documents for programming languages like C++ only mentions an abstract machine [177] without any hardware significance as such even structures like points need not have any hardware significance attached to them.

Here in this section, we only presented a brief overview of the source code, later on, in the other sections, a more complete and comprehensive treatment shall be meted out with regards to source code structure and analysis.

Application Programming Interface

Application Program Interface [178] [179] refers to the methods by which to source code or program interact with one another. It specifies arguments as well as return types for with which two software entities interact. API should not be confused with other software architectures like sockets and pipes which are used to communicate betwen software entities but will also have their APIs. API concerns the source code itself.

Application Binary Interface

The Application binary interface specifies how the binary or code level entities are mapped into the memory and interact with one another. Note the concept of ABI extends form processor till the extended machines, like the processors have certain calling conventions and Endianess in them which has to be accounted for when programming them, similarly Operating system has certain system call conventions that the program has to account for. Generally ABI accounts for things like [180]:

- data type, size, and alignment
- the calling convention, which controls how functions' arguments are passed and return values retrieved
- the system call numbers and how an application should make system calls to the operating system

Some other compiler standardization of the ABI include:

- the C++ name mangling

- exception propagation, and calling convention between compilers on the same platform, but do not require cross-platform compatibility

Overall, the ABI can be organized into three distinct hierarchies [181]:

1. processor level - the instruction set, the calling convention

2. kernel level - the system call convention, the special file path convention (e.g. the /proc and /sys files in Linux), etc.

3. OS level - the object format, the runtime libraries, etc.

Although programming languages generally don't consider ABI and generally the task of selecting an ABI is responsibility of the compiler [182] , since we work on an abstract machine while using them, ABI stability [183] is a important concept since all the memory layout of data structures as well as expression of programming structures on the memory is specified by the ABI and any non- transferable changes in it will cause the program to not function properly. ABI hence is a binary concept instead of source code level.

Shared and Dynamic Libraries

A software library [184] is a suite of data and programming code that is used to develop software programs and applications. It is designed to assist both the programmer and the programming language compiler in building and executing software. A software library generally consists of pre-written code, classes, procedures, scripts, configuration data and more. Typically, a developer might manually add a software library to a program to achieve more functionality or to automate a process without writing code for it. For example, when developing a mathematical program or application, a developer may add a mathematics software library to the program to eliminate the need for writing complex functions. All of the available functions within a software library can just be called/used within the program body without defining them explicitly. Similarly, a compiler might automatically add a related software library to a program on run time.

The first modularization introduced in the development of the system level software is that of libraries [185]. In terms of division they can be divided into being Static and Dynamic Libraries. The Static libraries are compiled and merged into the code at compile time by the linker, whereas the Dynamic libraries are linked at runtime into the code and might be shared between the modules [186]. Dynamic Libraries are mostly an Operating System concept whereas the Static libraries are commonly found in embedded system where in the system functionality such as embedded networks stacks, is generally implemented in terms of a static library .

Run-time System

The runtime system [187] can be considered as being first step towards virtualization and abstraction of the hardware. Runtime library provides execution environment for the language features. Note in some cases, Run-time Systems are considered as an extension of ABI. A Runtime library [188] is the library with the features follow:

1. Runtime library consists a collection of functions specialize in a specific target hardware. For example: behavior of printf function in each different target hardware is different. In Intel x86 PC platform , printf will out a string to console, but in many embedded ARM boards, may be printf will out a string to serial port. So printf is a part of the runtime library.

2. Runtime library consists a collection of functions which are only used by compiler, application developer cannot to use it. For example: in compiling processing, compiler creates some internal functions automatically. These functions are used for some tasks as initialize zero data, main()e.t.c. These functions cannot call by application developer.

3. Runtime library consists functions that can be performed only (or are more efficient or accurate) at runtime are implemented in the runtime library. For example: some logic errors, array bounds checking, dynamic type checking, e.t.c.

More precisely, for example, the execution model for the implementation of malloc() would be provided by the runtime and as such would differ when the program is generated for an Operating system or for an Embedded baremetal platform.Similarly, Heap functionality of the program is provided by the runtime whereas stack is mostly a hardware aspect based on a processor (more precisely direction of growth of stack). Note Desktop based binary applications have some runtime libraries in the form of crt0 which are executed before the program main() entry path is reached, similarly baremetal micro-controllers have a concept of startup file [189] which carries out the initialization of the platform before the main() function [190] is reached. Note that Runtime library shouldn't be confused with standard C library which is in fact set of functions which are directly available to user or programmer. [191]

2.7.2 Program Analysis

In this section, we will examine the analysis of programs as well as various methodologies which are used for their verification, security analysis and execution time analysis whether on bare-metal platform or the extended machine in the form of Operating System.

Every Program, much like the processor or any other technology within the digital industry can be broadly divided into Control and Data-flow paths [192]. Hence, regardless of whether the program or its binary will run eventually bare-metal or on an operating system, the analysis can be done in terms of these two structures present therein.

The basic process behind program design and development in modern computing is its eventual translation from source to binary format and as such translation of source to binary can be classified as being a sub-category of Electronic Design Automation, although there is an active research community dedicated towards creating compact binaries directly for execution [193]. The concept of comprehending the program is important as all the modern computing structures are software including operating systems and modularized libraries.

In classical times, much like the absence of any memory model discussed in the section before, programs were also created with an expertise of the programming without any unified design methodologies towards these concepts. Programmers would manually create control flow graph to perform operations on the data path and then would code or express them in assembly language to be executed eventually on a platform, however, these days, generally compiler creates and optimizes both the control and data flow models in all of its stages, as such the entire process has become highly automated. In this section, we will not discuss the internals of compliers and representation but the main focus would be on the Structure of the program with regards to control and data flow in various stages and the aspects of Vulnerability, Performance and Verification which incur as a result of these program representations.

Note that control and data flow abstraction is similar to many other dualities within electrical engineering . Also the abstraction of control and data flow division exists irrespective of the language used as such a scheme is tantamount to the topology of a circuit as well.

Program Terminologies

In this section, we will review the all the basic terminologies associated with program analysis and synthesis. Some of the cardinal structures include:

1. Translation Unit: A translation unit (TU) is the post-processed source code that is used as input to the compiler, the output of which is an object file. A TU comprises a single source file (C or CPP), with all header files included, and after all preprocessor statements have been evaluated. All names within a translation unit have internal linkage and can be resolved by the compiler. Names in other translation units have external linkage must be resolved by the linker. A translation unit is sometimes called a compilation unit. A C++ program consists of a number of translation units. [194]

2. Basic Block: Basic Block is a straight line code sequence which has no branches in and out branches except to the entry and at the end respectively. Basic Block is a set of statements which always executes one after other, in a sequence. [195]

3. Super Block : A Superblock is a trace which has no side entrances. Control may only enter from the top but may leave at one or more exit points. So single entry at top, but multiple exits. We use profile information to build a Superblock from common path which includes multiple basic blocks. We then apply Superblock optimization. [196]

4. Hyper Block : A hyperblock is a set of predicted basic blocks in which control may only enter from the top, but may exit from one or more locations. Hyperblocks are formed using a modified version of if-conversion. Basic blocks are included in a hyperblock based on their execution frequency, size, and instruction characteristics. Speculative execution is provided by performing predicate promotion within a hyperblock. Superscalar optimization, scheduling, and register allocation may also be effectively applied to the resultant hyperblocks. [197]

5. Control flow graph: A control-flow graph (cfg) models the flow of control between the basic blocks in a program. A cfg is a directed graph, G = (N, E). Each node n N corresponds to a basic block. Each edge e = (ni, nj) E corresponds to a possible transfer of control from block ni to block nj. A cfg has a node for every basic block and an edge for each possible control transfer between blocks. [198]

6. Loops: A natural loop has exactly one start node which is executed every time the loop iterates. This node is called header. A natural loop is repeatable, i. e., there is a path back to the header. [199]

7. Data flow graph : A data flow graph is a model of a program with no conditionals. In a high-level programming language, a code segment with no conditionals more precisely, with only one entry and exit point is known as a basic block. [200]

8. Inter-procedural control flow graph : An approach that avoids the burden of annotations, and can capture what a procedure actually does as used in a particular program, is building a control flow graph for the entire program, rather than just one procedure. We add additional edges to the control flow graph. For every call to function g, we add an edge from the call site to the first instruction of g, and from every return statement of g to the instruction following that call. [201] CFG (Control Flow Graph): representsw control flow for a single procedure whereas IPCFG (Inter-Procedural Control Flow

Graph) represents control flow for a program. [202] Call graphs (CG) and control flow graphs (CFG) consist of nodes and edges. CG is interprocedural, nodes represent subroutines (methods, functions, ...), while edges represent the relationship caller-called between two subroutines (e.g., A-¿B, "A" is the caller subroutine, while "B" is the called subroutine). CFG is intraprocedural, nodes represent program statements, including called subroutines but also conditionals, while edges represent the flow of the program. When CG and CFG are combined, it is called interprocedural control flow graph (ICFG). CFG generation is more resource-intensive than CG but more detailed.[203]

9. Call Graph: A call graph (also known as a call multigraph) is a control-flow graph, which represents calling relationships between subroutines in a computer program. Each node represents a procedure and each edge (f, g) indicates that procedure f calls procedure g. Thus, a cycle in the graph indicates recursive procedure calls. Call graphs can be dynamic or static. A dynamic call graph is a record of an execution of the program, for example as output by a profiler. Thus, a dynamic call graph can be exact, but only describes one run of the program. A static call graph is a call graph intended to represent every possible run of the program. The exact static call graph is an undecidable problem, so static call graph algorithms are generally over approximations. That is, every call relationship that occurs is represented in the graph, and possibly also some call relationships that would never occur in actual runs of the program. [204]

10. Source Code Annotation : Source code annotation refer to placing the information of certain basic blocks into the source code of a program to attain a certain aim which can either be performance information when a virtual prototype of a certain computing platform is being created or some debugging information annotation so as to observe in real-time the progress of a certain program. [205]

Program Generation Flow

This section will deal with the generation of program that is the morphing of source code into binary form. Although there is a considerable overlap with the compiler technology, here we will only consider the process in therms of generation of the program, with regards to control and data-flow, to be eventually executed on an architectural platform and not the optimizations involved in the process. A sample program generation flow is as shown below, wherein, it can be seen that two cardinal entities involved in program analysis are the source code and binary code:

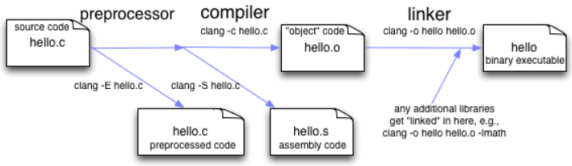


Figure 2.42: Program Generation Flow[16]

Program Verification

The most fundamental operation to be performed on any program is its integrity and functioning. As over the course of this thesis, only simple housekeeping operations were performed, we will present only a brief overview of the techniques involved n the whole process and not the mathematical intricacies involved therein.

The concept of program verification at its core involves developing a mathematical model of both the control flow and the data-flow within its structure to formally prove the proper functioning of the design. Note that effectiveness of formal methods with regards to confirming the validity of the program is at par to other modelling methods in engineering wherein the model they can only assure the correctness provided that the methods used to create the model were effective and accurate [206].

Some good literature on the subject can be read in [207] [208] [209]. Note that the mathematical optimization techniques in form of discrete mathematics are the same as those utilized for the hardware verification of the digital circuits discussed in the sections before. A sample formal verification flow is a s shown below:

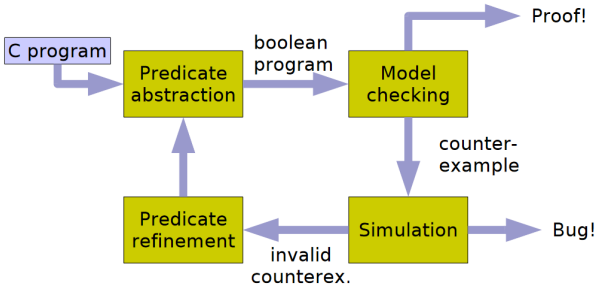


Figure 2.43: Formal Verification Flow[17]

As devising Formal Verification techniques encompassing both control and data flow path is a tedious procedure which does scale well and suffers from same state space explosion problem discussed before, generally Industrial Projects avoid it and alternative simulation or analysis solutions are used to test the programs tantamount to UVM library used in testing of digital designs. One such commonly used architecture and library is the xUnit, which is used for functional verification of the software, and its architecture is as shown below:

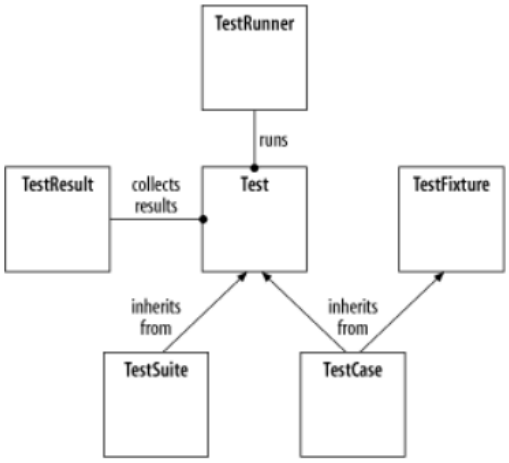


Figure 2.44: xUnit Architecture[18]

The tools utilized for "Informal" or approximate verification of the generated programs can be broadly divided into static and dynamic analysis tools [210], note that when formal methods are used for verification, it is possible to cover both such analysis simultaneously although the process is rather tedious but the human error is completely eliminated. Although the modelling error devised for formal verification still persists

In static code analysis, the code is parsed and the tool attempts to determine it functionality, for example, Linting attempts to carry out static analysis of the code to determine the short-coming and errors that exist in it. In dynamic code analysis, the code is examined at runtime via executing it. It entails running only those paths that are executable and accounts for the inadequacy of the Static code analysis. Valgrind, for example, is a dynamic analysis tool which instruments the code and creates a virtual machine at runtime to examine its behaviour in terms of memory leakage for example among other things. Also software testing techniques like unit testing with xUnit also utilize dynamic testing to carry out code coverage and functional verification of the design.

Program Vulnerability Analysis

Program vulnerability refers to utilization of loopholes within the control and data flow structures as such to alter the normal flow of the program execution on a given platform. As security is not chief concern within this thesis, a brief glimpse of such vulnerabilities will be given in this section. Overall, at program level, there can be the three basic types of vulnerabilities including [211]:

1. Control Flow Vulnerabilities : Also called control-based attack, in which an attacker uses a memory corruption error, such as a buffer overflow or use-after-

free, to overwrite control-data such as a return address or function pointer and thereby modifies the control-flow of the program.

2. Data Flow Vulnerabilities : Also called non-control data attacks. These attacks do not modify the control-flow of programs, but instead corrupt user identity data, configuration data, user input data or decision-making data to achieve the attacker's ends

3. Platform Vulnerabilities : The platform vulnerabilities result from the natures of the platform on which the program is being executed, they include things like Differential Power Attack [212], Spectre and Meltdown resulting from micro-architecture of the platform and also Operating system vulnerabilities like attacking the system call handler [213] or the Directory traversal attack[214].

Note that to successfully execute these attacks, the underlying machines need to have an appropriate models for example, it is very hard to impossible to launch a control flow attack on Harvard architecture as program code resides in a separate address space. [215] [216]

Program Worse Case Execution Time (WCET) Analysis

This section will deal with analysis of the programs with regards to their execution time on a given platform. Execution time analysis is increasingly becoming of great significance as there is more and more importance being attached to real-time performance of application especially with regards to embedded system design industry [217]. Overall WCET or Worst Case Execution Time attempts to quanitfy all possible execution paths ina program and can be broadly be divided into two major types [218]:

1. Static WCET Analysis: Static analysis methods attempt to obtain a WCET estimate without executing or simulating a program.Instead,an estimation is made by statically analysing the software together with the hardware.Static analysis most often aims to derive safe estimates,in contrast to dynamic analyses.However,due to the complexity of software and hardware,and due to the requirement on safety,a static safe static analysis occasionally has to introduce safe approximations resulting in a less precise estimate.Often such approximations arise from simplifications,with the result that the analysis considers potential executions which actually never occur in the program.

2. Dynamic WCET Analysis: Dynamic(or measurement based) approaches,which have long been the industrial standard, are methods that rely on running or simulating a program to find information about the execution time.In order to find the WCET then,the program would have to be run on its worst-case input and its worst-case hardware state.However,it is typically very difficult to know

which is the worst-casein- put and hardware state.Thus,the dynamic approach has to execute a program end-to-end a large number of times with a large number of inputs and observe the longest execution time.However,unless it is a very simple program run- ning on very simple hardware,the measured executions will constitute a subset of the actual set of possible executions.Thus,there are no guarantees that the worst-case has been observed.This means that results from measurement based approaches typically cannot be considered as safe estimates

Code Complexity Measures

In this section, we review the code quality metrics which can expressed and explained via the control and data flow structures introduced before. The metrics, of significant value in software industry, include [219]:

- Source Lines of Code (SLOC) – It counts the number of lines in the source code. It is the most straightforward metric used to measure the size of the program. However, functionality and complexity do not relate that well as a skilled developer might be able to deliver the same functionality with a significantly smaller code.

- Cyclomatic Complexity – This measures how much control flow exists in a program. Since programs with more conditional logic are more complex, measuring the level of complexity tells the developers how much needs to be managed. This is done by directly measuring the number of paths through the code. Say a program is a graph of all possible operations. Then, complexity measures the number of unique paths through that graph. Every if, while, or for statement creates a new branch. It is even possible for one branch to double the total number of paths. But the results from this method can sometimes be very deceptive.

- Halstead Volume – This metric measures how much information is in the source code. It looks at how many variables are used and how often they are used in programs and functions. Since these additional pieces of data affect data flow, programmers must learn them.

- Maintainability Index – It calculates the overall score of the maintainability of a program. Maintainability index is more of an empirical measure, unlike Cyclomatic complexity and Halstead volume. This index has been developed over the years with consultants from Hewlett-Packard and its software teams. It weighs Cyclomatic complexity and Halstead volume against the number of lines of code in the program. This analysis gives an overall picture of software complexity.

2.7.3 Compiler

The compiler at its very heart can be considered as a program transforming code from one language to another as per the given inputs and requirements. In this section, we will give a glimpse of workings and design of the compiler.

Compiler Drivers

Compiler driver [220] [221] is the concatenation and manager of the compilation process. contrary to popular believe GCC is actually a compiler driver, gcc and clang are both known to be compiler drivers. As such, the gcc executable does not compile anything itself. Rather, it calls the compiler (cc1), assembler (as) and linker (ld) with the right flags as needed. The behavior is controlled by spec strings, which are provided by a plain-text spec file. You can run gcc -dumpspecs to dump the built-in spec file. It is complex but the main idea is construction of cc1/assembler/linker command lines. Note: the interaction with the assembler/the linker should be clear from the output. The g++ program is another compiler driver. It uses -x c++ by default and additionally links against the C++ library. The two programs are otherwise equivalent.[222]

Compiler Characteristics

Some characteristics of a good compiler [223] include (in order of importance):

1. the compiler itself must be bug-free
2. it must generate correct machine code
3. the generated machine code must run fast
4. the compiler itself must run fast (compilation time must be proportional to program size)
5. the compiler must be portable (i.e, modular, supporting separate compilation)
6. it must print good diagnostics and error messages
7. the generated code must work well with existing debuggers
8. the generated code must work well with existing debuggers
9. must have consistent and predictable optimization.

Compiler Architecture

Compiler are generally divided into stages and encompass aspects like Parser, Semantic analyzer, AST synthesizer, Register allocation algorithms, Program block analyzer, Tokenizer, Instruction scheduler, Type checking algorihms, Symbol table manager, Alias analyzer, Loop analyzer, dead code eliminator, strength reduction algorithm and control/dataflow analysis algorithms for the code. Although it is possible to architect a compiler as a monolithic architecture and even merge it with a linker to create executable directly as was done with pascal compilers in the past, or treat the architecture of the compiler in terms of distributed architecture wherein there are multiple stages in compiler with their own unique architecture, the two most pre-dominant styles of compiler construction model include Aho Ullman approach and the Davidson Fraser approach.

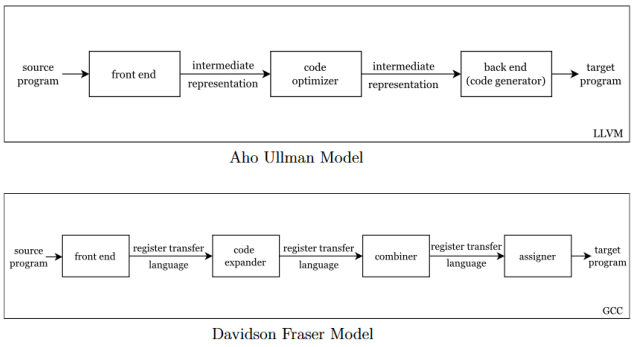


Figure 2.45: Compiler Architectures[19]

The Aho Ullman model places a large focus on having a target-independent intermediate representation (IR) language for a bulk of the optimization before the backend which allows the instruction selection process to use a cost-based approach. The Davidson Fraser model focuses on transforming the IR into a type of target-independent register transfer language (RTL).1 The RTL then undergoes an expansion process followed by a recognizer which selects the instructions based on the expanded representation. LLVM follows Aho Ullman model whereas GCC is based on Davidson Fraser model.

Note that often a times compiler is treated as a black box in case of carrying out its performance analysis for example and a sample program is executed with sample benchmarks and compiled with it to check its integrity or performance. [224]

Compiler Structure

Compiler structure can be broadly divided into three distinct phases including: front-end, middle-end and back-end. [225]

1. The front end scans the input and verifies syntax and semantics according to a specific source language. For statically typed languages it performs type checking by collecting type information. If the input program is syntactically incorrect or has a type error, it generates error and/or warning messages, usually identifying the location in the source code where the problem was detected; in some cases the actual error may be (much) earlier in the program. Aspects of the front end include lexical analysis, syntax analysis, and semantic analysis. The front end transforms the input program into an intermediate representation (IR) for further processing by the middle end. This IR is usually a lower-level representation of the program with respect to the source code.

2. The middle end performs optimizations on the IR that are independent of the CPU architecture being targeted. This source code/machine code independence is intended to enable generic optimizations to be shared between versions of the compiler supporting different languages and target processors. Examples of middle end optimizations are removal of useless (dead code elimination) or unreachable code (reachability analysis), discovery and propagation of constant values (constant propagation), relocation of computation to a less frequently executed place (e.g., out of a loop), or specialization of computation based on the context. Eventually producing the "optimized" IR that is used by the back end.

3. The back end takes the optimized IR from the middle end. It may perform more analysis, transformations and optimizations that are specific for the target CPU architecture. The back end generates the target-dependent assembly code, performing register allocation in the process. The back end performs instruction scheduling, which re-orders instructions to keep parallel execution units busy by filling delay slots. Although most optimization problems are NP-hard, heuristic techniques for solving them are well-developed and currently implemented in production-quality compilers. Typically the output of a back end is machine code specialized for a particular processor and operating system.

Note that although, most modern compilers are considered in three stages, it is very much possible to study compiler structure just by studying two stages only [226], wherein:

1. The front end consists of the following phases:
   - scanning: a scanner groups input characters into tokens;
   - parsing: a parser recognizes sequences of tokens according to some grammar and generates Abstract Syntax Trees (ASTs);
   - semantic analysis: performs type checking (ie, checking whether the variables, functions etc in the source program are used consistently with

their definitions and with the language semantics) and translates ASTs into IRs;

- optimization: optimizes IRs.

2. The back end consists of the following phases:
   - instruction selection: maps IRs into assembly code;
   - code optimization: optimizes the assembly code using control-flow and data-flow analyses, register allocation, etc;
   - code emission: generates machine code from assembly code.

The generated machine code is written in an object file. This file is not executable since it may refer to external symbols (such as system calls). The operating system provides the following utilities to execute the code:

- linking: A linker takes several object files and libraries as input and produces one executable object file. It retrieves from the input files (and puts them together in the executable object file) the code of all the referenced functions/procedures and it resolves all external references to real addresses. The libraries include the operating sytem libraries, the language-specific libraries, and, maybe, user-created libraries.

- loading: A loader loads an executable object file into memory, initializes the registers, heap, data, etc and starts the execution of the program. Note that Initialization here refers to platform initialization wherein loader initializes the processes heap and segments whereas the runtime is responsible for the memory segments management of the program itself.

GCC and LLVM Structure

As the two open source and commonly utilized compilers are GCC and LLVM, in this section, we will review and present a diagrammatic representation of their inner structure:

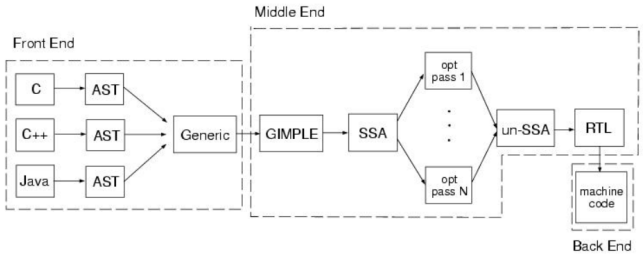


Figure 2.46: Structure of GCC[20]

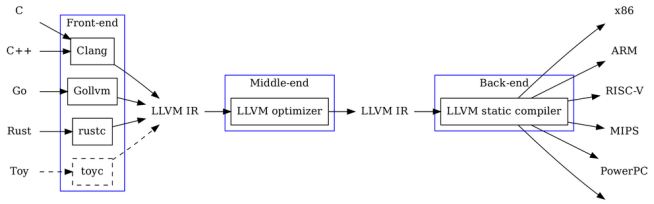


Figure 2.47: Structure of LLVM[21]

Note that in each of its stages, the compiler utilizes Symbol Table,Literal Table and error handler to keep track of the representational features of the program. [227]. Note that Global offset table and Procedure linkage table also track the address schemes and object modules linkage of external libraries and individual objects. The function, particularly that of the GOT, is little different in cases when the code is compiled in form of position independent code or for Load-time relocation. [228] [229]

Compiler Types

Compiler can be broadly divided into several types including: [230]

1. Single pass : If we combine or group all the phases of compiler design in a single module known as single pass compiler. A one pass/single pass compiler is that type of compiler that passes through the part of each compilation unit exactly once. Single pass compiler is faster and smaller than the multi pass compiler. As a disadvantage of single pass compiler is that it is less efficient in comparison with multi-pass compiler.

2. Multi pass Compilers : A Two pass/multi-pass Compiler is a type of compiler that processes the source code or abstract syntax tree of a program multiple times.

Object File

The end result of a compilation operation if the object file which can be grouped into three distinct types [231]:

1. Relocatable object file, which contains binary code and data in a form that can be combined with other relocatable object files at compile time to create an executable object file.

2. Executable object file, which contains binary code and data in a form that can be directly loaded into memory and executed.

3. Shared object file, which is a special type of relocatable object file that can be loaded into memory and linked dynamically, either at load time or at run time.

Object File Format

Although at bare metal level, there is a little distinction between memory model and executable file format, since the program at bare metal level in micro-controller mirrors its memory model at programming level, modern operating systems include distinct file format for program loading and execution. note that primary reason that such high-level formats were introduced was to completely abstract and visualize the memory address space in terms of sections for the compiler and as such there are several propriety file formats and their respective loaders as well. As our application is mostly geared towards single address spaces and bare metal programming, we will only give a brief overview of basic file formats for executable files, object code, shared libraries, and core dumps. including [232]:

1. A.out : A.out is the output file of the assembler and the link editor. In both cases, a.out is executable provided there were no errors and no unresolved external references. This file has four sections: a header, the program text, asymbol table, and relocation bits. a.out remains the default output file name for executables created by certain compilers and linkers when no output name is specified, even though the created files actually are not in the a.out format. [233]

2. COFF : The PE/COFF file headers consist of a MS-DOS stub, file signature, COFF Header, and Optional Header. An object file contains only the COFF Header, but an image file contains all the headers described above. The

most important of these headers is the COFF header. It was introduced to replace the previously used a.out format, and formed the basis for extended specifications such as XCOFF and ECOFF. [234]

3. COM : The COM format is the original binary executable format used in DOS. It is very simple; it has no header and contains no standard metadata, only code and data. This simplicity exacts a price: the binary has a maximum size of 65,280 (FF00h) bytes (256 bytes short of 64 KB) and stores all its code and data in one segment. [235]

4. ELF : ELF [236] was introduced as main binary format for linux following problems with previous formats. ELF format is flexible, extensible, and cross-platform. For instance it supports different endiannesses and address sizes so it does not exclude any particular central processing unit (CPU) or instruction set architecture. This has allowed it to be adopted by many different operating systems on many different hardware platforms.The ELF format is defined by the generic specification (gABI) to which processor specific extensions(psABI) are added. It comprises of [237]
    - Section: tells the linker if a section is either: raw data to be loaded into memory, e.g. .data, .text, etc. or formatted metadata about other sections, that will be used by the linker, but disappear at runtime e.g. .symtab, .srttab, .rela.text
    - Segment: tells the operating system: where should a segment be loaded into virtual memory and what permissions the segments have (read, write, execute).

2.7.4 Linker

Linker is a program concerned with combining or linking multiple object files together. Overall, Linker performs three fundamental tasks including [238]:

1. Linking together, arranging or mapping multiple sections of the object file onto new object file or an executable in form of the segments (which are classical .text, .data and .stack) as per the binary format described which is generally ELF.

2. Linking multiple object files into one single executable file

3. Placing the sections of the object file into the specified positions of the executable as per the given linker scipt.

Generally, there are three broad types of commands used by the linkers including [239]:

1. Memory command : Describes the location and size of blocks of memory in the target.

2. PHDRS command : The ELF object file format uses program headers, aka segments. The program headers describe how the program should be loaded into the target memory.

3. Sections command : Tells the linker how to map input sections into output sections, and how to place the output sections in memory.

The output of the linker is an executable file or an executable format which can be loaded and run on a processor. Hex is the most common program format for all the micro-controller based systems where modern Linux based system generally utilize the ELF file format.

Although in modern computing systems, mostly the compiler and linker are considered as being separate concepts.In reality, it is very much possible to combine the linker and compiler together as was done in the past with some old Pascal compilers as was discussed before. [240]

2.7.5 Loader

Loading is concerned with placing or loading the program into the address space of the processor to be executed. Loader can be classified in different ways for different execution platforms:

1. In simple micro-controller based systems, an SPI based bootloader is used to place program into the flash memory of the micro-controller for it to be executed. [241]

2. Other Micro-controller based systems also have a bootloader amenity, which is a program running in an alternative memory space and as such it is possible to load the program into the address space at runtime as well as opposed to in-system programming. [242]

3. In some Real-time Operating systems like the FreeRTOS, there is a possibility of loading and removing the modules at runtime and as such this can be classified as being loading. Although RTOS are generally single address space operating systems and lack program loading capability. [243]

4. In modern operating systems like Linux, loading is often carried out by the exec family of system calls which create replicate a current process followed by binfmtmisc.c which acts a static loader and parses and places the executable instructions from executbale file format onto the process followed by dynamic linker in form of ld-linux-x86-64.so which resolves the shared library symbols at runtime. [244]

In modern sense, loader usually refers to both loading and dynamic linking and as such when used it refers to both dynamic linking as well as process of loading file into memory.

Loader types

Program loaders can be classified in multiple ways in system programming level. A lot of these classifications can be considered as being classical, for example, dynamic loading at run-time, however they are still included for the sake of completion. There are some common alternatives for organizing the loading functions, including relocation and linking. Linking Loaders – Perform all linking and relocation at load time. The Other Alternatives are Linkage editors, which perform linking prior to load time and, Dynamic linking, in which linking function is performed at execution time .Some fundamental types of loaders are as listed below:

- Compiler-And-Go Loader : A classical loading scheme in which an assembler run in one part of memory and places assembled machine instructions, data as they are assembled, directly into their assigned memory locations. This type of loader was found in early computer and is the main loader type in TempleOS.
- Absolute Loader : The operation of absolute loader is very simple. The object code is loaded to specified locations in the memory. At the end the loader jumps to the specified address to begin execution of the loaded program.
- Boot-Strap Loader : When a computer is first turned on or restarted, a special type of absolute loader, called bootstrap loader is executed. This bootstrap loads the first program to be run by the computer – usually an operating system. The bootstrap itself begins at address 0. It loads the OS starting address 0x80. No header record or control information, the object code is consecutive bytes of memory.
- Relocating Loader : The concept of program relocation is, the execution of the object program using any part of the available and sufficient memory. The object program is loaded into memory wherever there is room for it. The actual starting address of the object program is not known until load time. Relocation provides the efficient sharing of the machine with larger memory and when several independent programs are to be run together. It also supports the use of subroutine libraries efficiently. Loaders that allow for program relocation are called relocating loaders or relative loaders.
- Dynamic linking loader : The scheme that postpones the linking functions until execution. A subroutine is loaded and linked to the rest of the program when it is first called – usually called dynamic linking, dynamic loading or

load on call. The advantages of dynamic linking are, it allow several executing programs to share one copy of a subroutine or library. In an object oriented system, dynamic linking makes it possible for one object to be shared by several programs. Dynamic linking provides the ability to load the routines only when (and if) they are needed. The actual loading and linking can be accomplished using operating system service request.

2.7.6 Build Tools and Automation

As mentioned in the section before, the compiler driver can be used to drive the overall program generation process via command line although such a setup does not scale properly in large projects, hence we require a build system to keep proper tracking all dependencies involved in the build process [245]. A brief overview of the build process is as shown below:

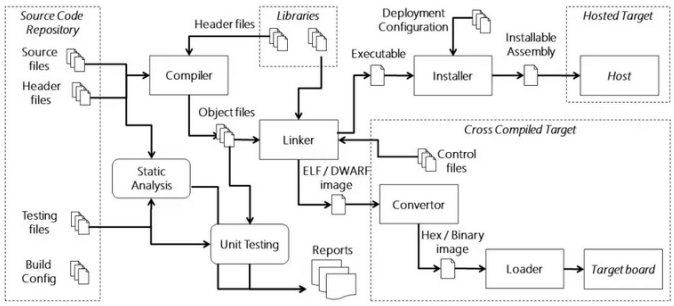


Figure 2.48: Build Process Overview[22]

It can be seen from the above flow that the ultimate result of a build process is an executable which can run on a machine. The most common and prevalent build system used is make [246] which is configured and driven via the makefiles. Other build systems include Maven and CMake, which is also build system automation, i.e a build system that creates file for other build systems [247]. Note that there is a distinction between build tool and a build system, wherein a build tool operates on a set of packages. It determines the dependency graph and invokes the specific build system for each package in topological order. The build tool itself should know as little as possible about the build system used for a specific package. Just enough in order to know how to setup the environment for it, invoke the build, and setup the environment to use the built package whereas build system on the other hand operates on a single package [248]. Hence tools like Yocto and OpenWRT can be better classified as being build tools rather than build systems.

Determinism in Build systems

Modern build systems are required to have a determinism in the binary or executable that they output. A build is said to be "Reproducible" [249] when, given a set of source code, the exact same binary can be generated by different people on different computers. Specifically, this means:

1. Code size and placement of sections are the same
2. Filepaths emitted in the binary are the same

3. When a binary diff is computed between the build artifacts (e.g the ELF or .bin), there will be zero differences

Many open source projects (such as Debian GNU/Linux) are working towards making the entire build reproducible. Some reasons why the build might not be deterministic include [250]:

1. Compressing files with different levels of parallelism could result in different output

2. Dates, times and build paths can be embedded in the built objects

3. Dates and times of build objects changes depending on when the build was run and which tasks happened in parallel

Some possible ways to make a build deterministic is by ensuring that it picks from its environment as little as possible:

1. Build on the same distro version with the same installed packages

2. Build in the same path

3. Use the same build hardware

Build Types

The various build types [251] can be classified into several categories including:

1. Full Build: In the full build process we perform the build process from scratch, it treats all the resources of the project never seen by the build server/build tool. It takes the full project as an input, checks the dependencies, compiles all the source files and builds all the parts in order. After that assemble it into build artifact.

2. Incremental Build: In the full build process we perform the build process from scratch, it treats all the resources of the project never seen by the build server/build tool. It takes the full project as an input, checks the dependencies, compiles all the source files and builds all the parts in order. After that assemble it into build artifact.

3. Manual build trigger: This is the most common software building trigger. Whenever you are ready with the code/change, you can go to the build server using build tool and trigger the build.

4. Scheduled build trigger: Scheduled build triggers are configured to run the build at specific time of the day or when a specific event occurs. This triggers can be configured to force builds regardless of whether source changes occurred. One of the best example of it is scheduled nightly full build.

5. Scheduled build trigger: Scheduled build triggers are configured to run the build at specific time of the day or when a specific event occurs. This triggers can be configured to force builds regardless of whether source changes occurred. One of the best example of it is scheduled nightly full build.

6. Source code repository build trigger: This trigger initiates builds whenever any source code change occurs in the repository. When a developer commits any change in the version control system, this trigger gets initiated. It can be customised whether you want to trigger a build when a any single file get changed or any specific file(s) get modified/checked-in.

7. Post-Process build trigger : Post-processing build triggers listen for post-processing events. When an event is detected, it can trigger other events or any upstream/downstream build.

2.7.7 Instrumentation, Debugging, Tracing and Profiling

Another significant aspect of embedded system development is the debugging of design via simulation in software or on hardware. In this section, we will review some major design testing and verification present and utilized in modern embedded systems. Note that concepts of Instrumentation, Debugging, Tracing and Profiling [252] can be applied to all the concepts and computing structures discussed before including Processor, FPGA, Digital I.C, Programs and Operating Systems with their own unique definition and tools.

Instrumentation

Instrumentation can be more precisely described as being means by which debugging, tracing and profiling can be carried out. Some instances of instrumentation include:

- On-chip instrumentation : System-on-chips frequently offer JTAG based Run Debug capability enabling us to trigger and probe internal device registers.
- Linux Kernel Instrumentation : Kernel Function Instrumentation (KFI) is a kernel function tracing system, which uses the "-finstrument-functions" capability of the gcc compiler to add instrumentation callouts to every function entry and exit. The KFI system provides for capturing these callouts and generating a trace of events, with timing details. KFI is excellent at providing a good timing overview of kernel procedures, allowing you to see where time is spent in functions and sub-routines in the kernel.
- Source code instrumentation and binary code instrumentation : The program source and binary can also be instrumented using tools like valgrind to perform profiling, tracing or debugging.
- JTAG based instrumentation on micro-controllers : Micro-controllers also offer tun and debug capability allowing profiling, tracing as well as debugging.

Debugging

Debugging with respect to hardware and software refers to placing breakpoints or probing points into the the design. Typically a debugger is used that can pause the execution of an application, examine variables and manipulate them.

Tracing

Trace is a log of events within to the program. Those events can be ordered chronologically, Tracing is the process of generating and collecting those events while the use case is typically flow analysis.

Profiling

Profiling is a dynamic analysis process that collects information about the execution of an application the type of information that is collected depends on your use case, e.g. the number of requests, the result of profiling is a profile with the collected information. The source for a profile can be exact events or a sample of events that occurred because the data is aggregated in a profile it is irrelevant when and in which order the events happened.

2.8 ADC Technology

The analog-to-digital technology is concerned with quantifying as well as discretizing the analog waveform making it suitable for a digital representation.

2.8.1 ADC Types

Although there can be several types of analog-to-digital converters based on the requirement of the application at hand here we will only discuss the two cardinal types [253] which are:

Flash ADC

A flash ADC is a type of Analog-to-digital converter that uses a linear voltage ladder with a comparator at each "rung" of the ladder to compare the input voltage to successive reference voltages. Often these reference ladders are constructed of many resistors; however, modern implementations show that capacitive voltage division is also possible. The output of these comparators is generally fed into a digital encoder, which converts the inputs into a binary value

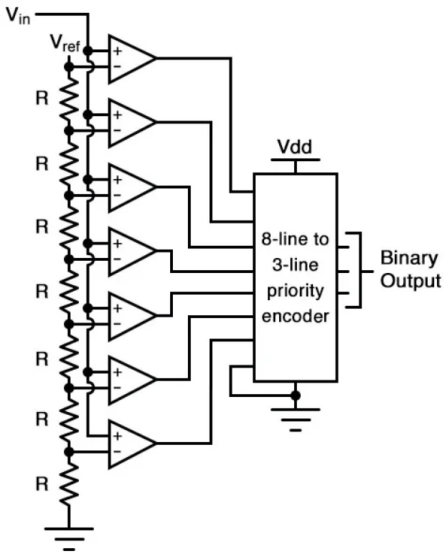


Figure 2.49: Flash ADC

They have the advantage of lower hardware effort to design, less power dissipation but generally have a slow response.

Successive Approximation ADC

A successive-approximation ADC is a type of analog-to-digital converter that converts a continuous analog waveform into a discrete digital representation using a binary search through all possible quantization levels before finally converging upon a digital output for each conversion.

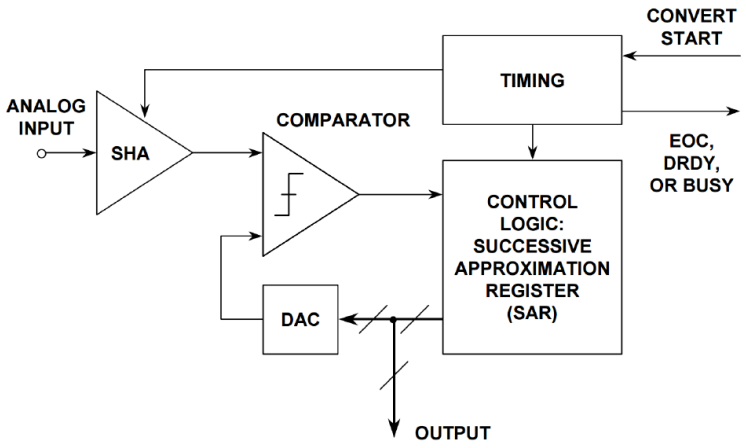


Figure 2.50: Successive-Approximation ADC [23]

These ADCs require lower hardware effort to create and maintain, have low power dissipation but are comparatively slower than flash ADCs.

2.8.2 ADC Figures of Merit

A figure of merit (FoM) [254] is a useful tool for comparing the conversion efficiency of A/D converters. Come common measures [255] include Sampling frequency, Resolution in terms of number of bitd, effective number of bits, signal to noise ratio (ENOB, SNDR), Power consumption, power supply requirements while others also include Differential Non-Linearity and Integral Non-Linearity. Among more famous data converter FOMs suggested [256], the ones by schreier and carter happen to be the most famous one and are frequently used in assessing the performance of an ADC.

2.8.3 Noise Shaping and Dithering

Another significant operation in data converter system is that of noise shaping and dithering. Dither [257] is an intentionally applied form of noise used to randomize quantization error, whereas noise shaping [258] refers to altering the spectral shape of the error that is introduced by dithering and quantization; such that the noise power is at a lower level in frequency bands at which noise is considered to be less

desirable and at a correspondingly higher level in bands where it is considered to be more desirable.

2.9 Interconnects

The interconnects are basic elements of digital or any hardware level design and as such can be grouped in two broad categories of on-chip interconnect or off-chip interconnect.

2.9.1 Off-Chip Interconnects

The off-chip interconnects are used by the ICs to communicate with one another and include:

UART

UART [259] transmit data asynchronously, and also utilizes two pins namely Rx and Tx to do so. The structures of UART packet, as can be surmised from the packet below, comprises of Start bit, Data bits and then finally the Stop bit. UART detects a start bit, it starts to read the incoming bits at a specific frequency known as the baud rate. Baud rate is a measure of the speed of data transfer, expressed in bits per second (bps). Both UART must operate at about the same baud rate. The baud rate between the transmitting and receiving UART can only differ by about 10 percent before the timing of bits gets too far off. A simple UART packet is as shown below:

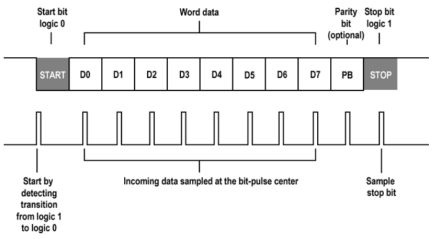


Figure 2.51: UART Packet Structure[24]

UART is a simple protocol that can be used to configure devices or send a command and is not suited to heavy data processing or transfer operations. Maximum data rate supported is about 230 Kbps to 460kbps. [260]

SPI

SPI [261] is also known as four wire serial bus and is a synchronous protocol. The SPI bus specifies four logic signals including:

- SCLK: Serial Clock (output from master)
- MOSI: Master Out Slave In (data output from master)
- MISO: Master In Slave Out (data output from slave)
- CS /SS: Chip/Slave Select (often active low, output from master to indicate that data is being sent)

SPI has four modes (0,1,2,3) [262] that correspond to the four possible clocking configurations.

With non-inverted clock polarity (i.e., the clock is at logic low when slave select transitions to logic low):

- Mode 0: Clock phase is configured such that data is sampled on the rising edge of the clock pulse and shifted out on the falling edge of the clock pulse. This corresponds to the first blue clock trace in the above diagram. Note that data must be available before the first rising edge of the clock.
- Mode 1: Clock phase is configured such that data is sampled on the falling edge of the clock pulse and shifted out on the rising edge of the clock pulse. This corresponds to the second blue clock trace in the above diagram.

With inverted clock polarity (i.e., the clock is at logic high when slave select transitions to logic low):

- Mode 2: Clock phase is configured such that data is sampled on the falling edge of the clock pulse and shifted out on the rising edge of the clock pulse. This corresponds to the first orange clock trace in the above diagram. Note that data must be available before the first falling edge of the clock.
- Mode 3: Clock phase is configured such that data is sampled on the rising edge of the clock pulse and shifted out on the falling edge of the clock pulse. This corresponds to the second orange clock trace in the above diagram.

A typical SPI packet structure is as shown below:

SPI is more data-intensive protocol and is used in for example retrieving data from a storage device or peripherals as it supports high data-rate. Maximum data rate limit is not specified in SPI interface. Usually supports about 10 Mbps to 20 Mbps.[260]

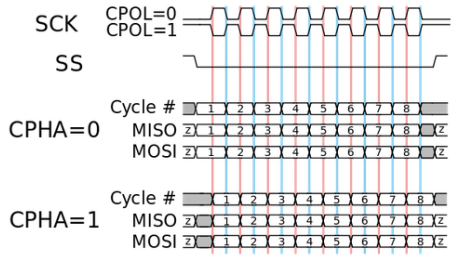


Figure 2.52: SPI Packet Structure[25]

I2C

The Inter-Integrated Circuit (I2C) [26] Protocol is a protocol intended to allow multiple "peripheral" digital integrated circuits ("chips") to communicate with one or more "controller" chips. Because serial ports are asynchronous (no clock data is transmitted), devices using them must agree ahead of time on a data rate. The two devices must also have clocks that are close to the same rate, and will remain so–excessive differences between clock rates on either end will cause garbled data. Each I2C bus consists of two signals: SCL and SDA. SCL is the clock signal, and SDA is the data signal. The clock signal is always generated by the current bus controller; some peripheral devices may force the clock low at times to delay the controller sending more data (or to require more time to prepare data before the controller attempts to clock it out). This is called "clock stretching" and is described on the protocol page. A typical I2C packet structure is as shown below:

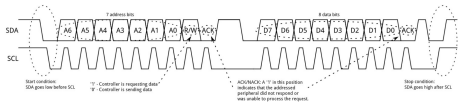


Figure 2.53: I2C Packet Structure[26]

To initiate the address frame, the controller device leaves SCL high and pulls SDA low. This puts all peripheral devices on notice that a transmission is about to start. If two controllers wish to take ownership of the bus at one time, whichever device pulls SDA low first wins the race and gains control of the bus. The address frame is always first in any new communication sequence. For a 7-bit address, the address is clocked out most significant bit (MSB) first, followed by a R/W bit indicating whether this is a read (1) or write (0) operation.After the address frame has been sent, data can begin being transmitted. The controller will simply continue generating clock pulses at a regular interval, and the data will be placed on SDA by either the controller or the peripheral, depending on whether the R/W bit indicated a read or write operation. Once all the data frames have been sent, the controller will generate a stop condition. Stop conditions are defined by a 0-¿1 (low to high)

transition on SDA after a 0-¿1 transition on SCL, with SCL remaining high.

I2C is generally used for housekeeping for example controlling the opening of a camera shutter or retrieving vales of power from the board chips as was the case in this board, also the possibility of creating a network of I2C on the board also makes its use attractive. I2C supports 100 kbps, 400 kbps, 3.4 Mbps. Some variants also supports 10 Kbps and 1 Mbps. [260]

PCI

Although PCI has not been utilized as an interconnect over the course of this thesis, we will have a brief discussion of it here as it permeates all modern computing boards. PCI Express® or PCIe® [27] is a high performance, high bandwidth serial communications interconnect standard that has been devised by the Peripheral Component Interconnect Special Interest Group (PCI-SIG) to replace bus-based communication architectures, such as PCI, PCI Extended (PCI-X), and the accelerated graphics port (AGP).

The topology of a simplified PCI system, consisting of the four function types the root complex, switch, endpoints and bridge.

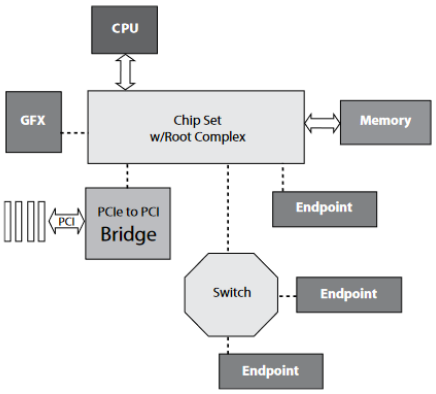


Figure 2.54: Hierarchy of simulation methods[27]

1. The root complex initializes the whole PCIe fabric and configures each of the links. It normally connects the central processor unit (CPU) to one or more of the other three functions – PCIe switches, PCIe endpoints and PCIe-to-PCI bridges

2. The PCIe switch routes data downstream to multiple PCIe ports, and from each of these individual ports upstream to a single root complex. PCIe switches may also route traffic flexibly from one downstream port to another (peer-to-peer), eliminating the restrictive tree structure required by traditional PCI systems.

3. Endpoints normally reside in the end applications connecting the application to the PCIe network in the system. The Endpoint requests and completes PCIe transactions. Generally, there are more endpoints in the system than any other type of PCIe component.

4. The bridge connects PCIe to other PCI bus standards, such as PCI/PCI-X, in systems that employ those bus architectures as well as PCIe.

The protocol as defined in the PCIe specification adheres to the Open Source Initiative (OSI) model. It is partitioned into five principal layers, as shown on the left side of figure 2. This section provides a general overview of the Mechanical and Physical Layers; subsequent sections will address Link, Transaction and Application Layers. A snapshot of the protocols involved in the PCI can be seen below.

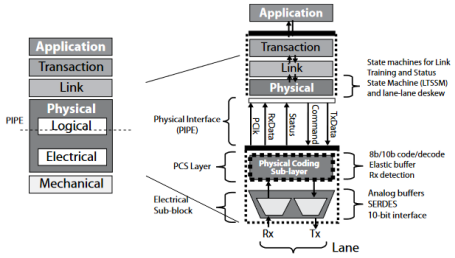


Figure 2.55: Hierarchy of simulation methods[27]

2.9.2 On-Chip Interconnects

With the advent of the system on chips, there have been multiple on-chip interconnects that have been introduced. In this section, we will discuss the ones which are utilized most frequently. The most common utilized architectures for the on-chip interconnect between the devices are:

1. CoreConnect on-chip bus system from IBM Microelectronics.

2. The Wishbone Bus Architecture by Silicore Corporation

3. Avalon architecture from Altera

4. AMBA from ARM

CoreConnect

The CoreConnect architecture [263] comprises of three distinct elements including:

1. Processor Local Bus: The PLB is the main on-chip system bus. It links the processor with on-chip memory, memory controllers, and other high-speed peripherals, including DMA controllers. It's a synchronous, multimaster, arbitrated bus.

2. On-Chip Peripheral Bus: Designed to support slower peripherals, the OPB is implemented as a straightforward multimaster, arbitrated bus. It's a synchronous bus with a common clock, but its devices can run with slower clocks, as long as all of the clocks' rising edges are in sync with the rising edge of the main clock. This bus uses a distributed multiplexer implementation.

3. Device Control Register Bus: The DCR bus provides an alternative path to the system for setting the individual device control registers. With it, the host CPU can set up the device-control-register sets without loading down the main PLB. This bus has a single master, the CPU interface, which can Read or Write to the individual device control registers.

Wishbone

Wishbone [264] is a slave and master architecture that supports three distinct topologies including:

1. Point-to-Point Interconnection : Point to point is the simplest way of connecting two IP Cores to each other where a single Master interface is connected to a single Slave interface. For example, a microprocessor as Master interface can be connected to a serial input output port Slave interface. The data transaction is controlled by handshaking signals.

2. Data Flow Interconnection : The dataflow interconnection processes the data in a sequential manner. Each IP core contains both a Master and a Slave interface. An IP core appears as a Master to the next IP core and also appear as a Slave to the core prior to it.

3. Share Bus Interconnection : Shared bus interconnection allows multiple Masters to connect with multiple Slaves over a single shared bus. Only one Master can access the bus at a time. Master starts the bus cycle to Slave, in turn Slave participates in bus transactions with the Master.

4. Cross Bar Switch Interconnection : Unlike shared bus interconnection where a Master has to wait for the access of bus depending on the availability of bus, a crossbar switch interconnection allows multiple Masters to use the same interconnection as long as two Masters are not accessing the same Slave at a time.

Avalon

Avalon [265] is a synchronous interface and specifies the port connections between master and slave components and specifies the timing by which these components communicate. Basic Avalon bus transactions transfer one data item 8-, 16-, 32-, 64-, or 128-bits wide. Avalon uses separate address, data and control lines. This bus supports multiple bus masters. Masters and slaves interact with each other based on a technique called slave-side (distributed) arbitration. The Avalon bus model (switch fabric) provides the following services to Avalon peripherals connected to the bus: data-path multiplexing, address decoding, wait- state generation, dynamic bus sizing, interrupt priority assignment, latent transfer capabilities, and a streaming Read and Write capabilities.

There are point-to-point connections between master and slave pairs which need to be connected. [266] There is no any conflict when the different master port or slave port communicates with each other at the same time. If multiple master components need to access the same slave port, Avalon bus would automatically add the arbitration logic on the slave port. All of these improve the performance of Avalon bus.

Avalon bus offers many services for those peripheral components which are connected to it, including data channel multiplexing, address encoding, pipelining transmission capability, offering wait status, bus width dynamic adjustment, interrupt assignment, delay transfer mode and streaming transfer mode, etc.

AMBA

AMBA [267] is an open-standard that outlines how to connect and manage the different components or blocks within an SoC. The AMBA specification was developed by ARM and has become the de facto standard for interfacing components in an SoC.

There are three distinct buses described for facilitating on-chip communications. These are the Advanced High-Performance Bus (AHB), the Advanced System Bus (ASB), and the Advanced Peripheral Bus (APB). The AHB is the backbone of the system and is designed specifically for high performance, high-frequency components. This includes the connections of processors, on-chip memories, and memory interfaces among others.

The ASB is an alternative to the AHB where some high-performance features are not needed. The APB is a simplified interface designed for low bandwidth peripherals that do not require the high performance of the AHB or the ASB. These include components like a UART, low-frequency GPIO, and timers.

AMBA has also introduced the AXI protocol. AXI stands for Advanced eXtensible Interface and offers even higher performance than the AHB, implemented through a point-to-point connection scheme. Instead of a system bus, the AXI interconnect allows transactions between masters and slaves using only a few well-defined

interfaces. As the interconnects were not developed in detail over the course of this thesis, we will not go into details of their specifications and workings.

2.9.3 Realizations of Interconnect Points

Most of the functional system can be realized either as hardware or in the software and there can be equivalence checking for them as well. For instance, the Linux kernel has software structures like the POSIX timer which also provides the timing functionality equivalent to that of the hardware timers in the processors. Similar to any other hardware circuitry, it is also possible to realize drivers like UART, SPI and I2C entirely in software provided that the interfaced pins as well as hardware timers provide the means of realizing such a structure in the firmware, this processes is known as the bit banging [268] realization of the interfaces. Equivalently, it is also possible to realize other hardware constructs like the interrupts entirely in software as is done in operating systems like Linux.

2.10 Memory technologies

The primary hardware component which stores the data and retrieves it back is the memory. Given the existence of the memory wall, there has been renewed interest in reinventing the wheel for memory technologies and architectures, as such memory technologies have become a very hot research topic. There are many distinct types of memories present on whole where they serve their own distinct purposes including more unique types such as the phase shift memory, however, in the course of this thesis, we will restrict ourselves to those memories present on the board only or are ubiquitous in implementation.

2.10.1 Memory Classifications

Memories in an embedded system can be classified in a number of distinct ways, including:

- Random access and Non-Random access [269], wherein the data is accessed sequentially or it can be accessed in a random fashion.
- Volatile [270] in which the memory contents are lost upon supply voltage cut-off or non-volatile in which the memory content or data is retained after power cut-off.
- synchronous [271] wherein the internal operations of the memory are in sync with the supplied clock by the board or asynchronous wherein there is no external clock and memory element functions asynchronously. Generally, synchronous memories are faster as there is a possibility of pipeline therein.

2.10.2 Memory Types

SRAM

Static Random Access memory (SRAM) [272] is a type of semiconductor memory. It is static and volatile, implying data retention persists for as long as the device is powered without any form of a refresh, however, once the power is cut, data will be lost. It is random access, meaning the next memory location that can be read or written to does not depend on the last access location. The static property of SRAM comes from its use of some sort of a feedback mechanism to maintain the stored bit state. This is in contrast to other forms of memory, such as Dynamic RAM, where the stored state of the bit is kept in the form of a charge that leaks over time thereby requiring the data to be refreshed (i.e, read and re-written back). A typical SRAM cell is as shown below:

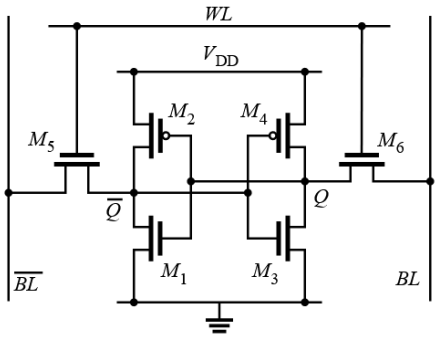


Figure 2.56: An SRAM Cell[28]

Large blocks of SRAM memory comprise of arrays of individual SRAM blocks called cells. An SRAM cell is capable of storing a single bit of data for as long as there is power. Likewise, an array of eight SRAM cells can store 1 byte of data. Arrays of SRAM form the foundation for every digital SRAM memory. A typical SRAM cell array is as shown below:

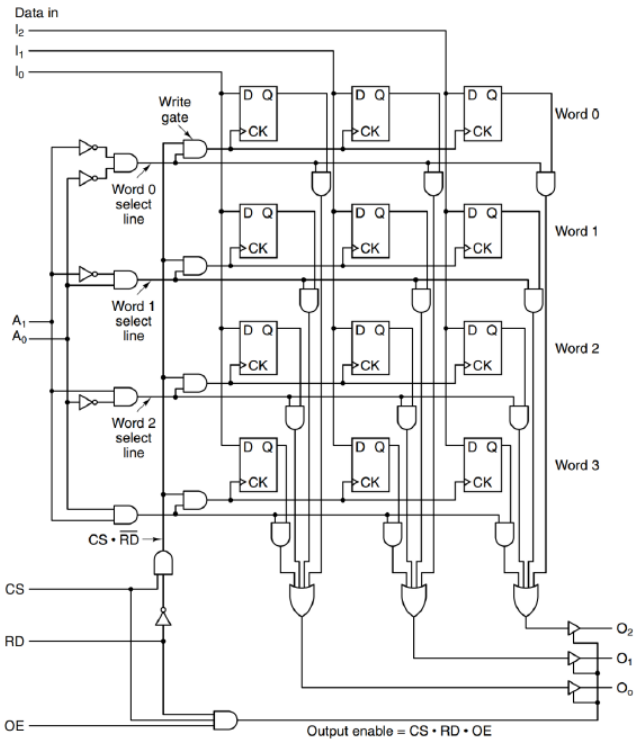


Figure 2.57: An SRAM Cell Array[29]

It is generally considered as being an expensive memory and as such generally internal and fast memory structures like cache and register file utilize it for storing or manipulating the data.

DRAM

Dynamic Random Access Memory (DRAM) [273] is widely used as a computers main memory. Each DRAM memory cell is made up of a transistor and a capacitor within an integrated circuit, and a data bit is stored in the capacitor. Since transistors always leak a small amount, the capacitors will slowly discharge, causing information stored in it to drain; hence, DRAM has to be refreshed (given a new electronic charge) every few milliseconds to retain data. The main advantages of DRAM are its simple design and low cost in comparison to alternative types of memory. The main disadvantages of DRAM are its high volatility and high power consumption relative to other options. [274]. A typical DRAM cell is as shown below:

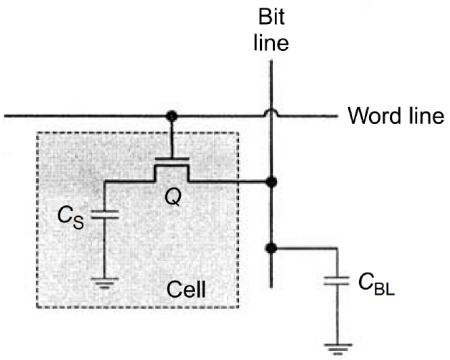


Figure 2.58: A DRAM Cell[30]

DRAM [275] is usually arranged in a rectangular array of charge storage cells consisting of one capacitor and transistor per data bit. Some DRAM matrices are many thousands of cells in height and width. The long horizontal lines connecting each row are known as word-lines. Each column of cells is composed of two bit-lines, each connected to every other storage cell in the column. They are generally known as the "+" and "" bit lines. A typical DRAM array is as shown below:

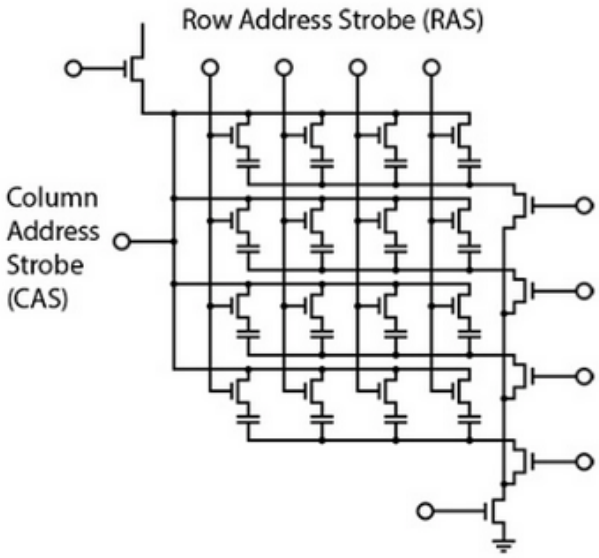


Figure 2.59: a DRAM Cell Array[31]

It is a bit more affordable memory than the SRAM [276] and as such is utilized frequently in external memories like the RAM on the motherboard.

FRAM

FRAM, [277] an acronym for ferroelectric random access memory, is a non-volatile memory that can hold data even after it is powered off. In spite of the name, FRAM is a ferroelectric memory and is not affected by magnetic fields as there is no ferrous material (iron) in the chip. Ferroelectric materials switch polarity in an electric field, but are not affected by magnetic fields. It combines the fast read and write access of dynamic RAM, DRAM whilst also providing non-volatile capability. An FRAM cell is as shown below:

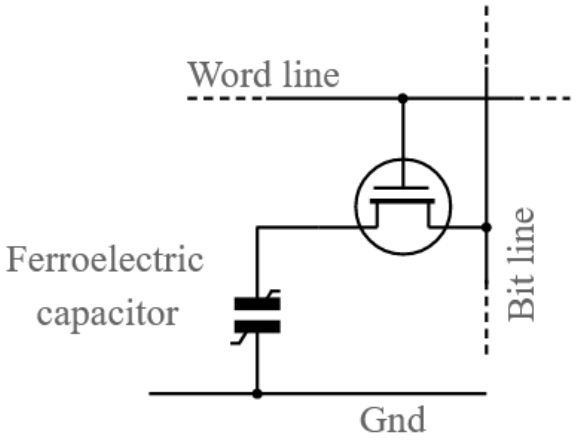


Figure 2.60: An FRAM Cell[32]

FRAM has fast write times. Beyond all the other operations, the actual write time to an FRAM memory cell is less than 50ns. That is up to 1000x faster than Flash/EEPROM. Writes to the FRAM cell occur at low volt-age and very little current is needed to change the data. With Flash, high voltages are needed. only a small amount of energy is required, all the necessary power for FRAM is front-loaded at the beginning of data write. Unified memory means it's the only technology to elimi-nate boundaries between variable and constant data, which simplifies data handling, in-system programming and firmware image.

FLASH

Flash memory [278], also known as flash storage, is a type of nonvolatile memory that erases data in units called blocks and rewrites data at the byte level. Flash memory is widely used for storage and data transfer in consumer devices, enterprise systems and industrial applications. Flash memory retains data for an extended period of time, regardless of whether a flash-equipped device is powered on or off. Flash is also known as Configuration memory as most of instructions in harvard architecture as well as extended harvard architecture are stored in it.

There are two types of flash memory: NOR and NAND. [33]

NOR and NAND flash memory differ in architecture and design characteristics. NOR flash uses no shared components and can connect individual memory cells in parallel, enabling random access to data. A NAND flash cell is more compact and has fewer bit lines, stringing together floating gate transistors to increase storage density.

NAND is better suited to serial rather than random data access. NAND flash process geometries were developed in response to planar NAND reaching its practical scaling limit.

NOR flash is fast on data reads, but it is typically slower than NAND on erases and writes. NOR flash programs data at the byte level. NAND flash programs data in pages, which are larger than bytes, but smaller than blocks. For instance, a page might be 4 kilobytes (KB), while a block might be 128 KB to 256 KB or megabytes in size. NAND flash consumes less power than NOR flash for write-intensive applications.

NOR flash is more expensive to produce than NAND flash and tends to be used primarily in consumer and embedded devices for boot purposes and read-only applications for code storage. NAND flash is more suitable for data storage in consumer devices and enterprise server and storage systems due to its lower cost per bit to store data, greater density and higher programming and erase (P/E) speeds.

Devices, such as camera phones, may use both NOR and NAND flash, in addition to other memory technologies, to facilitate code execution and data storage.

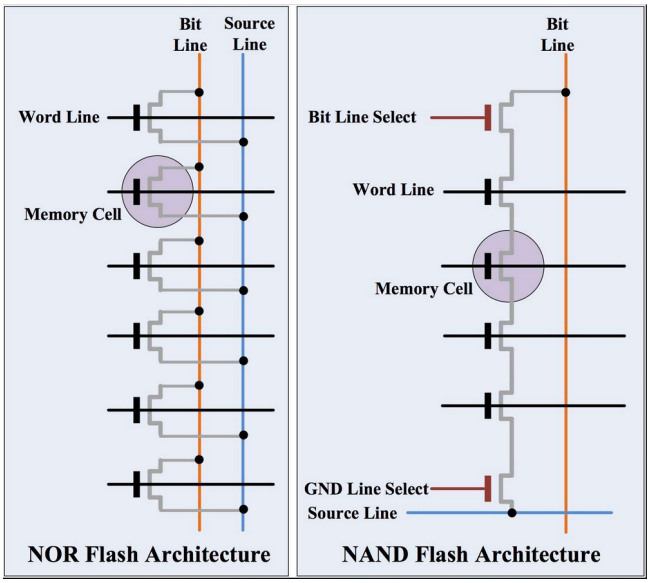


Figure 2.61: A Flash Cell[33]

Flash memory is also used for in-memory computing to help speed performance and scalability of systems that manage and analyze large sets of data. NAND flash-

based solid-state drives are often used to accelerate the performance of I/O-intensive applications. NOR flash memory is often used to hold control code, such as the basic input/output system (BIOS), in a PC.

EEPROM

EEPROM stands for electrically erasable programmable read-only memory. It is a non-volatile flash memory device, that is, stored information is retained when the power is removed. EEPROM generally offers excellent capabilities and performance. EEPROM is also known as the book-keeping memory and generally is used to small simple data like key of a cipher.

There are two distinct EEPROM [34] families: serial and parallel access. The serial access represents 90 percent of the overall EEPROM market, and parallel EEPROMs about 10 percent. Parallel devices are available in higher densities (256Kbit), are generally faster, offer high endurance and reliability, and are found mostly in the military market. Serial EEPROMs are less dense (typically from 256 bit to 256Kbit) and are slower than parallel devices. They are much cheaper and used in more "commodity" applications.

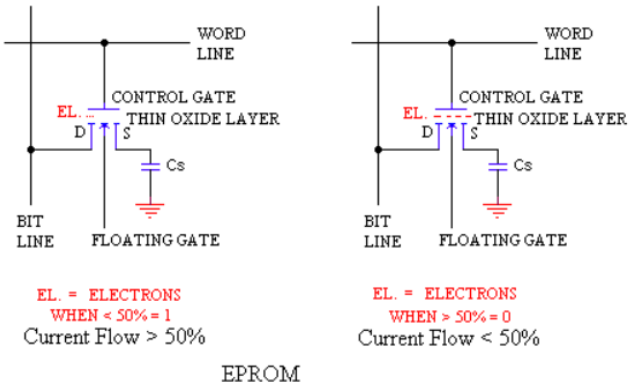


Figure 2.62: An EEPROM Cell[34]

There are several different types of EPROM's [279]:

- UV-EPROM: UV EPROM's are programmed at high voltages (usually 28V) and are erased by shining Ultra-Violet (high energy) light at them through a window. They have no limit to read cycles and can store data for over 20 years.

- OTP: One Time Programmable ROMS are generally identical to UV-EPROM's with one exception, they do not have the window for UV light and thus are not readily erasable. This was done to save money, as they could be packaged

in a plastic package instead of ceramic/glass. There has been some success in erasing these using X-Rays.

- EEPROM: These are Electronically erasable. Again it requires a higher voltage (12.5V-28V) which can be generated on chip by a charge pump. Write times are usually rather slow and power requirements are hard to manage as writes take MUCH more power then reads. EEPROM's also have a limited number of write cycles due to electron tunnel oxide degradation, a result of quantum mechanics.
- Flash: Flash is very similar to EEPROM's except they are built to write/erase entire blocks (multi-byte) at a time (EEPROM writes a byte at a time). This makes writes faster, and saves transistors.

3 Integrated Development Environment

Modern Digital as well as firmware based development relies on Integrated Development environment in addition to the more usual command line interface based on scripting languages like TCL and Bash Script. The IDE with which we will work is the Microsemi Libero which is used to program the FPGA system. With the predominance of electronic design automation software and algorithms, engineers usually only need to specify the design entry and constraints with the design synthesis and optimization being carried out by the software as opposed to entire schematic entry into the software, although all modern design entry provide option for manual entry. In this section, there will be brief discussion on integrated development environments for system on chip designs as well as the design automation therein.

3.1 Libero IDE and its features

Microsemi's Libero ® IDE software release for designing with Microsemi Rad-Tolerant FPGAs, Antifuse FPGAs and Legacy Discontinued Flash FPGAs and managing the entire design flow from design entry, synthesis and simulation, through place-and-route, timing and power analysis. [35]

The core feature of the Software IDE include:

- Powerful project and design flow management
- Full suite of integrated design entry tools and methodologies:
    1. SmartDesign graphical SoC design creation with automatic abstraction to HDL
    2. IP Core Catalog and configuration
    3. User-defined block creation flow for design re-use
- Synplify Pro ME synthesis fully optimizes Microsemi FPGA device performance and area utilization
- Synphony Model Compiler ME performs high-level synthesis optimizations within a Simulink® environment

- Modelsim ME VHDL or Verilog behavioral, post-synthesis and post-layout simulation capability
- Physical design implementation, floorplanning, physical constraints, and layout
- Timing-driven and power-driven place-and-route
- SmartTime environment for timing constraint management and analysis
- SmartPower provides comprehensive power analysis for actual and "what if" power scenarios
- Interface to FlashPro programmers
- Post-route On Chip Debug Tools and Identify ME debugging software for Microsemi flash designs
- Silicon Explorer II debugging software for Microsemi antifuse designs

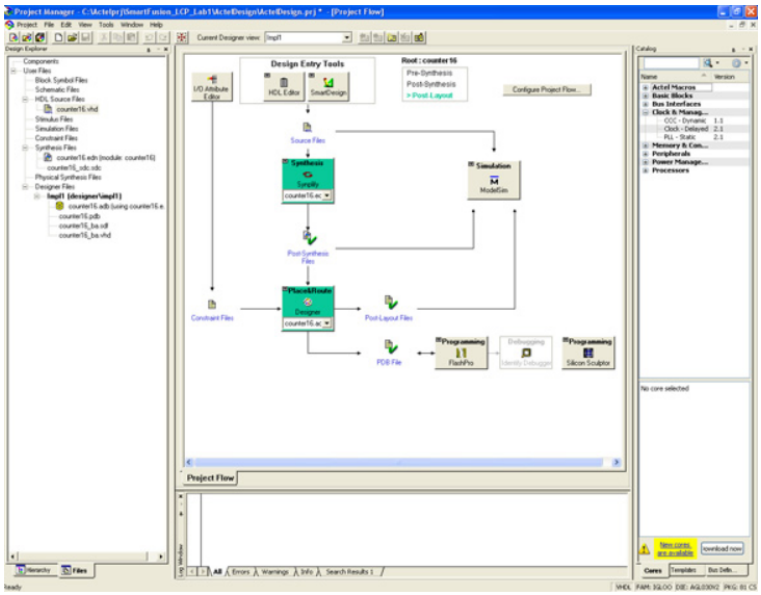


Figure 3.1: Libero IDE Project Manager[35]

3.2 Software licensing and its types

Overall the licenses for software or electronic design automation systems can be broadly divided into two broad categories including Fixed (or node based) or floating (or server based) licensing schemes. [280]

A node-locked license is locked to a specific hard disk ID or movable USB hardware key dongle. A USB dongle with the accompanying license file allows the software to operate on any PC to which the dongle is attached and the license file and software are installed.

A floating license is typically installed on a network server (Windows, Linux, or solaris) and allows networked client PCs to access the license from the server. The Client PCs can be Windows or Linux OS. Client seats can be purchased to allow up to 999 users to run the Libero software simultaneously.

3.3 TCL scripting for tool automation

The most dominant language used for manipulating and operating electronic design automation tools is TCL which is a scripting language. Although over the course of this thesis, GUI based IDE is used for interacting with the Microsemi libero, it is equally possible to use TCL scripting language to command the tool as well. In this section we will review basic command set and script used to interact with Libero software.

Some common example of TCL scripts are as shown below [36]:

```
# ------------------------------------------------------------
# 1.1 Create a new project
# ------------------------------------------------------------
new_project\
        -location {<project location path>/<project>} \
        -name {<project name>} \
        -project_description {} \
        -hdl {VERILOG} \
        -family {SmartFusion2} \
        -die {M2S050T} \
        -package {896 FBGA} \
        -speed {-1} \
        -die_voltage {1.2} \
        -adv_options {DSW_VCCA_VOLTAGE_RAMP_RATE:100_MS} \
        -adv_options {IO_DEFT_STD:LVCMOS 2.5V} \
        -adv_options {RESTRICTPROBEPINS:1} \
        -adv_options {TEMPR:COM} \
        -adv_options {VCCI_1.2_VOLTR:COM} \
        -adv_options {VCCI_1.5_VOLTR:COM} \
        -adv_options {VCCI_1.8_VOLTR:COM} \
        -adv_options {VCCI_2.5_VOLTR:COM} \
        -adv_options {VCCI_3.3_VOLTR:COM} \
        -adv_options {VOLTR:COM}
```

Figure 3.2: TCL script to create a project[36]

The following TCL script can be used to run the synthesis process.

```
# ------------------------------------------------------------
# 4. Running Synthesis
# The command also shows how to organize constraint files for Synthesis
#  I in step # 2 the file that was imported is an .edn, then the synthesis
#  step is not required.
# ------------------------------------------------------------
organize_constraints -file {./synthesis/<file_name>.sdc} \
      -mode {new} -designer_view {Impl1} \
      -module {<design_name>::work} \
      -tool {synthesis}


organize_tool_files -tool {SYNTHESIZE} \
      -file {./synthesis/<file_name>.sdc} \
      -module {<design_name>::work} \
      -input_type {constraint}

run_tool -name {SYNTHESIZE}
```

Figure 3.3: TCL script to run synthesis[36]

To save and close the project, we can utilize the below given TCL script.

```
# ------------------------------------------------------------
# 9. Save and close project
# ------------------------------------------------------------
close_project -save 1
```

Figure 3.4: TCL script to save and close project[36]

3.4 Directives, Constraints and Mapping

When HDL code is used at front-end digital design, there can be broadly two distinct types of command and syntax structures that can be used for optimum resource allocation and mapping onto the FPGA fabric including (Note that their definition may change from one environment to the other) [281] [37]:

3.4.1 Directives

Directives control compiler optimizations and hence have to be entered directly in the HDL source code. One instance is syn state machine directive which enables/disables state-machine optimization on individual state registers in the design. When you disable the FSM Compiler, state-machines are not automatically extracted. To extract some state machines, use this directive with a value of 1 on just those individual state-registers to be extracted. Conversely, when the FSM Compiler is enabled and there are state machines in your design that you do not want extracted, use syn state machine with a value of 0 to override extraction on just those individual state registers.

3.4.2 Attributes

Attributes control mapping optimizations whereas Attributes can be entered either in the source code, in the SCOPE Attributes tab, or manually in a constraint file. One example of synthesis attribute is as given below which is used to specify the type of encoding of a finite state machine to be utilized in a design, and forces the user specified specification to be used instead. Attributes are often used interchangeably with Directives as well.

```
(* fsm_encoding = "user" *)
reg [5:0] state;
```

Figure 3.5: Example of an Attribute[37]

3.4.3 Constraints

Constraints [282] are used to influence the FPGA design implementation tools including the synthesizer, and place-and-route tools. They allow the design team to specify the design performance requirements and guide the tools toward meeting those requirements. The implementation tools prioritize their actions based on the optimization levels of synthesis, specified timing, assignment of pins, and grouping of logic provided to the tools by the design team. The four primary types of constraints include synthesis, I/O, timing and area/location constraints:

1. Synthesis constraints influence the details of how the synthesis of HDL code to RTL occurs. There are a range of synthesis constraints and their context, format and use typically vary between different tools.

2. I/O constraints (also commonly referred to as pin assignment), are used to assign a signal to a specific I/O (pin) or I/O bank. I/O constraints may also be used to specify the user configurable I/O characteristics for individual I/Os and I/O banks.

3. Timing constraints are used to specify the timing characteristics of the design. Timing constraints may affect all internal timing interconnections, delays through logic and LUTs and between flip-flops or registers. Timing constraints can be either global or path-specific.

4. Area constraints are used to map specific circuitry to a range of resources within the FPGA. Location constraints specify the location either relative to another design element or to a specific fixed resource within the FPGA.

3.4.4 Mapping

Mapping is considered as being an attribute with which it becomes possible to ascribe the design elements directly onto the fabric of FPGA. Note that it is possible to specify Mapping constraints both in HDL source code as well as UCF (User Constrained File). The following pictures show an Implementation of 2 input NAND gate with LUT6 of FPGA architecture.

```
Verilog Instantiation Template
// LUT6: 6-input Look-Up Table with general output
// 7 Series
// Xilinx HDL Libraries Guide, version 2012.2
LUT6 #(
.INIT(64'h0000000000000007) // Specify LUT Contents
) LUT6_inst (
.O(y), // LUT general output
.I0(a), // LUT input
.I1(b), // LUT input
.I2(0), // LUT input
.I3(0), // LUT input
.I4(0), // LUT input
.I5(0) // LUT input
);
// End of LUT6_inst instantiation
```

Figure 3.6: 2 Input NAND gate with LUT6[38]

3.5 Logic Simulator and its Types

The logic simulator [39] is used to emulate the behaviour of digital circuits and as such is integral in verification of the designs. The hierarchy of simulators is as shown below.

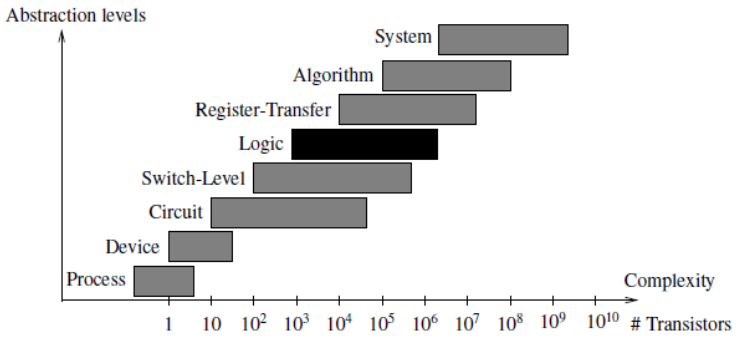


Figure 3.7: Hierarchy of simulation methods[39]

Note as discussed in the sections before, there is an alternative in form of formal methods wherein mathematics as opposed to simulation is used to verify the designs.

The inner structure and organization of a simulation tool is as shown below:

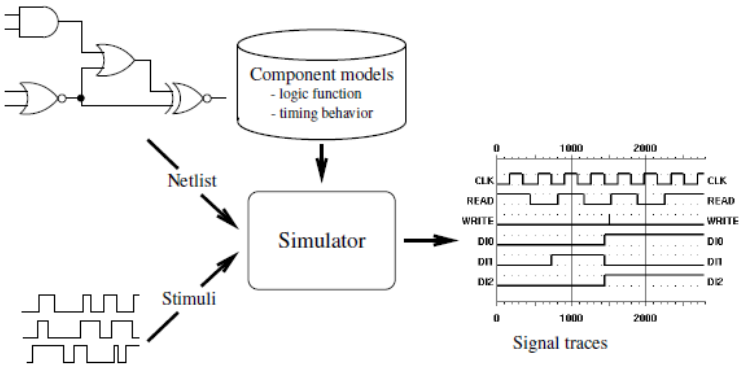


Figure 3.8: Overview of Simulation Engine[39]

The simulator utilizes netlist from design, component models from the library and get stimulus from the user to present the complete simulation model of the design. The user can generate models in high language like C to create stimulus for the simulator as was discussed in the sections before.

Broadly the logic simulator can be divided into two broad types including:

1. Compiler Driven Simulation : The circuit under test is compiled into an executable program. In each simulation cycle, the logic function of the complete circuit is evaluated, based on the input signals and the internal circuit states. (registers, feedback signals). Very fast, usually no timing analysis (zero delay simulation).

2. Event driven simulation : It is based on signal changes within the circuit. In each simulation step, components are evaluated only if a signal change occurs on their corresponding inputs. Signal changes are coded as events.

Over the course of the thesis, Modelsim simulator [40] was used to simulate and verify the designs. It is a compiler driven simulator and hence is very fast and is preferred tool for most simulations unlike the Isim from Xilinx which happens to be event driven simulator. As the verification model details were discussed in the section before, here we will only discuss the internals of the the Modelsim simulator as well as simple design flow therein. Also Modelsim can be controlled via both GUI or with TCL however the functions are all same either way.

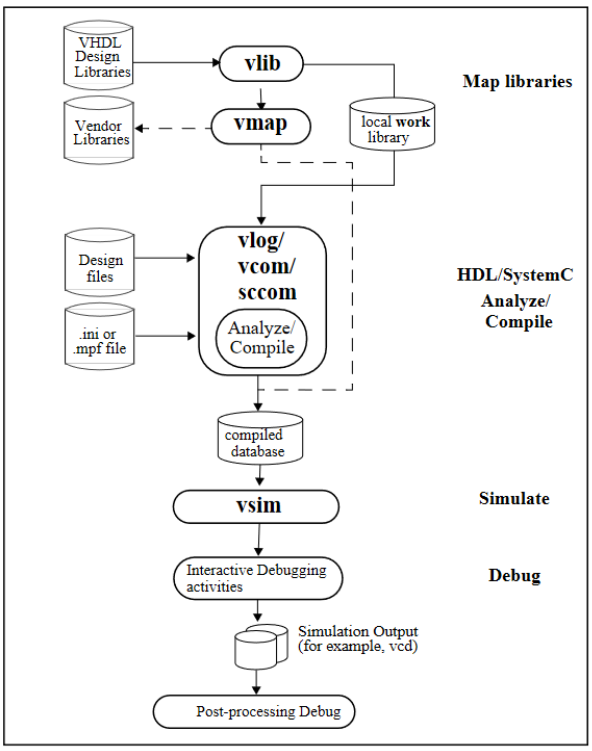


Figure 3.9: Modelsim Simulation Engine[40]

The design flow of ModelSim is as shown below, as mentioned before, as we are interested in the tools and verification capabilities of the simulator itself, we will not go into the details of the verification models themselves.

Task	Example Command Line Entry	GUI Menu Pull-down	GUI Icons
Step 1: Map libraries	**vlib** <library_name> **vmap** work <library_name>	a. **File** > **New** > **Project** b. Enter library name c. Add design files to project	N/A
Step 2: Compile the design	**vlog** file1.v file2.v ... (Verilog) **vcom** file1.vhd file2.vhd ... (VHDL)	a. **Compile** > **Compile** or **Compile** > **Compile All**	**Compile** or **Compile All**
Step 3: Load the design into the simulator	**vsim** <top>	a. **Simulate** > **Start Simulation** b. Click on top design module or optimized design unit name c. Click OK This action loads the design for simulation	**Simulate** icon:
Step 4: Run the simulation	run step	**Simulate** > **Run**	**Run**, or **Run continue**, or **Run -all**
Step 5: Debug the design	Common debugging commands: bp describe drivers examine force log show	**N/A**	N/A

Figure 3.10: Modelsim Design Flow[40]

The most significant commands for controlling and debugging the design, after adding the objects to the waveform window are:

- Run which is used to start the simulation
- Step which is used to advance simulation by a particular time interval specified to the simulator
- Stop which is used to stop the simulator
- Force in the debugging mode which is used to force a particular waveform on a netlist.

Other than this, ModelSim also features several windows in its GUI form including Project Window(window that shows all the files present in current project), Source Window(to show the source code in text form or more precisely it is a text editor), Wave Window (to display the waveform obtained from the design by the simulator tool) and FSM Viewer Window(to display all FSM extracted by the tool). These are the most significant display options in the GUI

3.6 SMART Design and Block Designs

Libero IDE offers an alternative design view in terms of implementing the digital designs. SmartDesign [283] is a visual block-based design creation/entry tool for the instantiation, configuration, and connection of Microsemi IPs, user-generated IPs, and custom/glue-logic HDL modules. This tool provides a canvas, which is analogous to a breadboard, for stitching together the different components in the design. The final result from SmartDesign is a design-rule-checked and automatically abstracted synthesis-ready HDL file. A generated SmartDesign can be the entire FPGA design or a component subsystem to be reused in a larger design.

The following design objects can be instantiated in the SmartDesign Canvas:

- Microsemi IP Cores
- User-generated or third-party IP Cores
- HDL design files
- HDL + design files
- Basic macros
- Other SmartDesign components (*.cxf files) generated from SmartDesign in the current Libero SoC project or imported from other Libero SoC projects
- Other SmartDesign components (*.cxf files) generated from SmartDesign in the current Libero SoC project or imported from other Libero SoC projects

The smart design is a great aid when designing system on chips form a system level point of view.

3.7 The Design Reports

Over the course the design synthesis process, several reports are generated, the most important of which are [284]:

3.7.1 Synthesis Report

This reports various (potential) problems in our design. Example unconnected outputs. Undriven inputs. Latch inference, for example, usually occurs because of bad coding rather than intentional design to use latches. It lists signals driven with constants to remind us whether it is necessary to have such signals or whether you really meant to do so. Also nets which are driven by more than one source are listed to let us review if that's what we meant to do to create a shared tri-state bus or a open-collector bus.

FSM (Finite State Machine) coding used (i.e. whether the state memory FFs were chosen based on encoded state assignment method or one-hot method, etc.) is revealed in this report. Note to change the type of finite state machine generated, we can pass an attribute to modify its creation.

3.7.2 Timing Report

Timing Reports are generated in various phases of the tool flow, but the most accurate report is the one produced after place and route. Place-and-Route is the final step before the tool generates a configuration file (.bit file) for the FPGA. In this step the Xilinx tool maps the circuit to physical locations in the FPGA and creates the signal routes that connect various logic elements. Routes contribute almost half of the latency in the circuit. So only after Place-and-Route is complete the tools can compute the precise delay of each path. The portion of the Place-and-Route report explaining the timing constraints is shown below. This shows the WNS (Worst Negative slack or Worst Case Setup time Slack). WNS is the difference between the clock period and the delay between a pair of registers. A positive worst case setup time slack means the constraint is met and a negative slack means that the longest path has a path delay longer than the clock period of the circuit. The longest path delay determines the maximum frequency at which the design can operate. The report contains more details about the timing of these paths. A timing path originates at a given starting point (usually the Q output of a source FF, but more accurately the starting point is the clock trigger at the clock input to the source FF, so that we can take into account the tffpd of the source FF), goes through some combinational logic, and then to a certain terminal point (usually the D input to the destination FF).

In the timing report, we get worst path information by default, so that we do not reduce the delay of one path of the design only to have another path slowed to the point where it becomes the longest path. Following is an excerpt from the timing report that details the delay of a single path. While we can easily identify the source and destination in this path, the intermediate signal names are obfuscated. Your report values may be different. Synthesis results may vary from run to run as optimization is somewhat heuristic and non-deterministic! You need to look for similar lines in your report.

3.8 PolarFire FPGA SoC Family

The Polar Fire FPGA [285] from Microsemi is a System-on-chip. PolarFire® FPGAs are the fifth-generation family of non-volatile FPGA devices from Microchip, built on state-of-the- art 28 nm non-volatile process technology. Cost-optimized PolarFire FPGAs deliver the lowest power at mid-range densities. Some of the features include:

- Up to 481K logic elements consisting of a 4-input look-up table (LUT) with a fractureable D-type flipflop
- The non-volatile FPGA fabric is built on state-of-the-art 28 nm low power non-volatile process technology. The PolarFire FPGA fabric is composed of the following building blocks: Logic elements, On-chip memory (LSRAM, SRAM, sNVM, and PROM) and Math block
- In each PolarFire FPGA, there are eight DLLs and eight PLLs to provide flexible clock generation and management capabilities. In addition to these DLLs and PLLs, up to 15 transceiver lane transmit PLLs are also available.
- PolarFire device user I/Os support multiple I/O standards while providing the high bandwidth needed to maximize the internal logic capabilities of the device and achieve the required system-level performance.
- ach PolarFire FPGA integrates two low-power built-in PCIe Gen2 controllers, allowing seamless and easy connectivity to one or more host processors
- The PolarFire FPGA system controller is based on the industry-standard ARM Cortex-M3 and is only used for FPGA powerup, secure DPA safe FPGA programming, and executing and responding to system services. All internal memories are SECDED protected with background scrubbing capabilities to remove single bit error
- Two specified user I/Os can be configured (at design capture stage) as either two single-ended live probes or one differential live probe. These live probes can provide read access to any register in the FPGA fabric, to the output pipeline registers in the LSRAMs, and to all the registers in the math block in real-time without having to re-instrument the code. A snapshot of all internal probe points can be created and read out asynchronously.

4 UART Sub-System Verification

This section discusses the design as well as the implementation of the UART controller sub-system designed and deployed on the board as a means for on-chip probing, communications and debugging.

UART being a common interface and protocol is frequently utilized in processing systems and digital electronics as a means of simple command based communication.

4.1 Logic Wrappers

The logic wrappers are a black box or top level view of the functionality to be implemented in the hardware eventually.

4.1.1 Top Wrapper

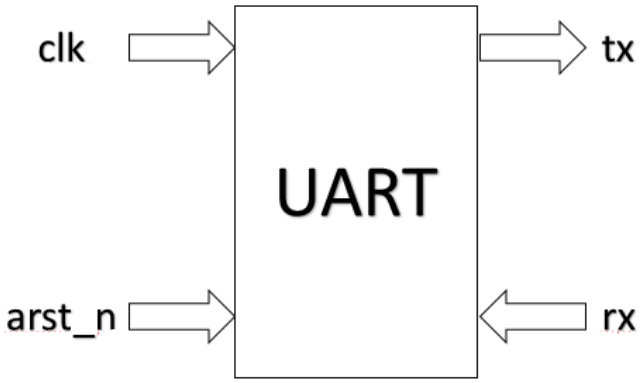


Figure 4.1: UART Top Level Wrapper

Pin	Direction	Bits	Summary
clk	input	1	The clock signal is basic signal at the positive edge of which state machine moves forward and events are triggered.
arstn	input	1	the reset signal is used to bring back the state machine to its idle state and refresh all registers to initial value.
tx	output	1	the tx pin is used to transmit the data from the UART module.
rx	input	1	the rx pin is used to transmit the data from the UART module.

Table 4.1: UART Signals Description

4.1.2 Receive (Rx) Wrapper

In this section, we will introduce a sub-system of the UART wrapper which is the Receive Wrapper of the UART design.

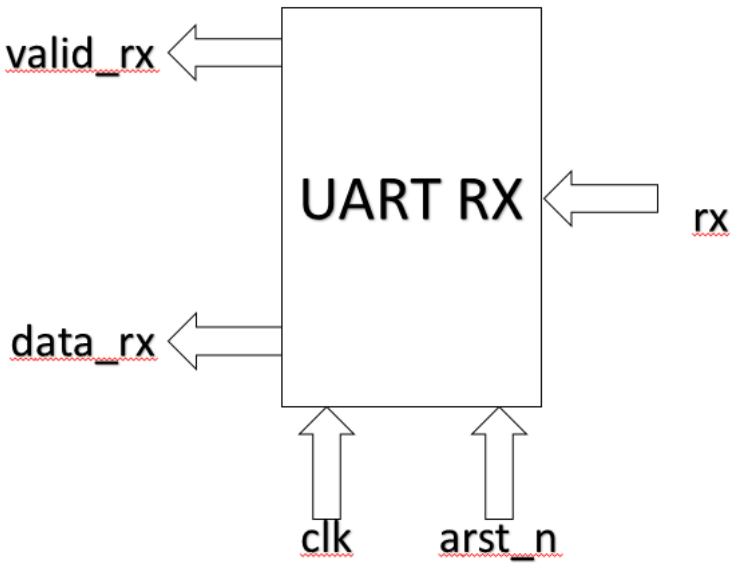


Figure 4.2: UART Rx Top Level Wrapper

Pin	Direction	Bits	Summary
clk	input	1	The clock signal is basic signal at the positive edge of which state machine moves forward and events are triggered.
arstn	input	1	the reset signal is used to bring back the state machine to its idle state and refresh all registers to initial value.
rx	output	1	the rx pin is used to transmit the data from the UART module.
datarx	output	8	the 8 bit wide data register is the amalgam is all the data received from the rx pin
validrx	output	1	the valid rx is triggered high for a single cycle to signal the successive module regarding the validity of the data

Table 4.2: UART Receive Signals Description

4.1.3 Transmit (Tx) Wrapper

In this section, we will introduce a sub-system of the UART wrapper which is the Transmit Wrapper of the UART design.

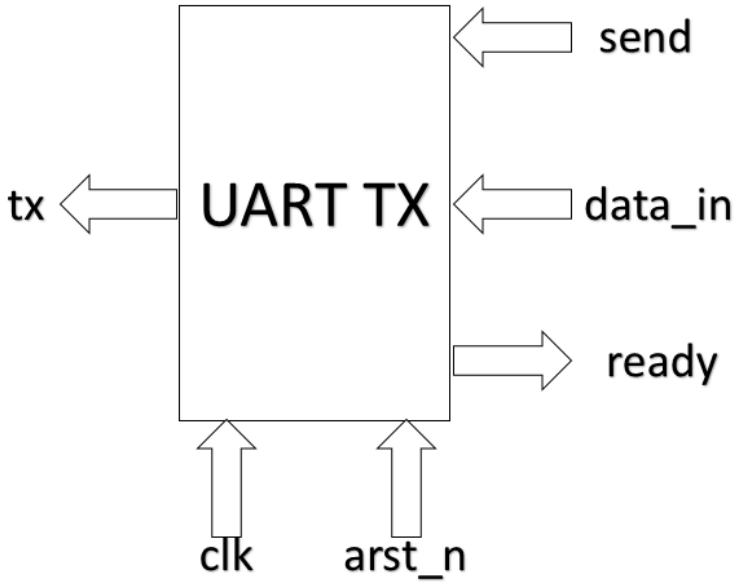


Figure 4.3: UART Tx Top Level Wrapper

Pin	Direction	Bits	Summary
clk	input	1	The clock signal is basic signal at the positive edge of which state machine moves forward and events are triggered.
arstn	input	1	the reset signal is used to bring back the state machine to its idle state and refresh all registers to initial value.
send	input	1	the send pin is kept high to indicate that data-in is valid and trigger the module. It is an atomic operation.
datain	input	8	the 8 bit wide datain register is used to send the data through UART vuia tx pin. Data is send via the send trigger operation.
tx	output	1	the tx pin is used to transmit the data from the UART module.

Table 4.3: UART Transmit Signals Description

4.2 UART Subsystem and Chipset

The UART subsystem and all its relevant chipsets can be shown below, the main chipset therein is the FTDI [286] which can also be configured. However, over the course of this thesis, the default configuration was utilized.

It can be seen that there are Leds present on the board which indicate the the transmission and reception operating to and from the FPGA circuitry.

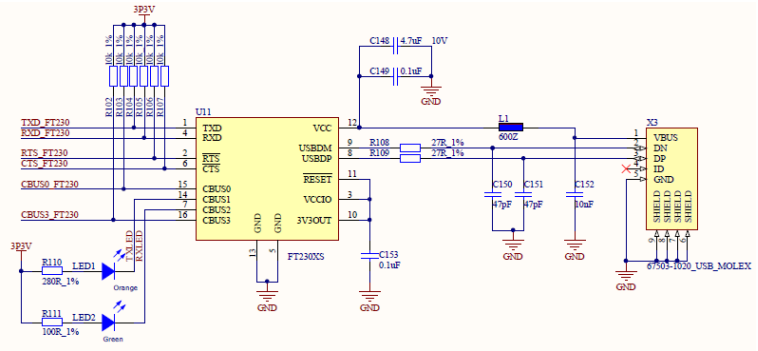


Figure 4.4: UART Power Monitoring Circuitry

4.3 Probing Path Design

The goal of design for this section is the UART based probing path and as such this section will introduce the top level view of the probing path designed to probe and check the design under test on the FPGA. Broadly speaking, there are four distinct sub-modules present within the design including:

1, UART RX: This module is responsible for receiving the data from th computer or external device

2, Hot Key Mux: This module is responsible for detecting the right sequence and triggering an event

3, DUT: This is the module which is to be tested and can include memory elements or the ADC.

4, UART TX: This module is responsible for transmitting the data from the probing subsystem back to the computer or any attached external device.

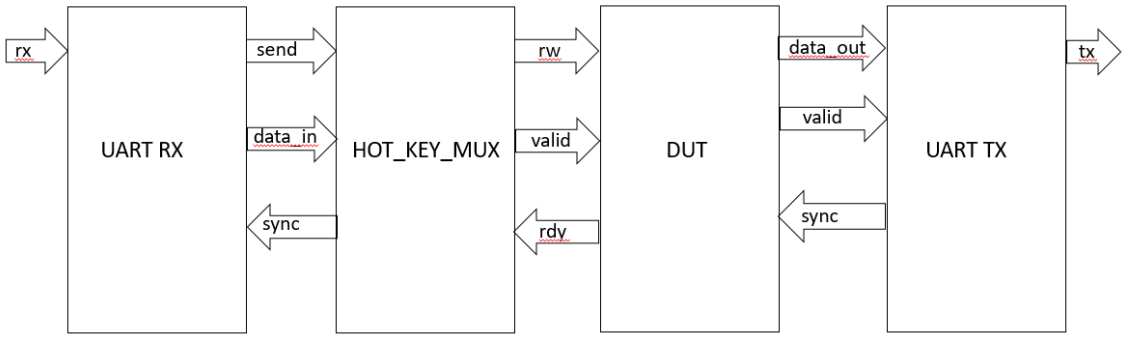


Figure 4.5: UART based Probing Path

Day	Direction	Bits	Summary
rx	input	1	the rx pin is used to transmit the data from the UART module.
send	output	1	the send signal signals the successive module that datain is valid.
datain	output	1	the 8-bit wide data in send the received signal into the hot key multiplexer.
sync	input	1	the rx pin is used to transmit the data from the UART module.
rw	output	1	used when interfacing memory device drivers and indicates whether action to be performed is read or write.
valid	output	1	signal which indicates that the successive module to start its operation. It is to be kept high for signal cycle and is atomic in nature.
rdy	input	1	the rdy signal indicates that the successive DUT module is ready to perform action.

Table 4.4: DUT Handshaking Signals Description

Day	Direction	Bits	Summary
rw	input	1	The read write signal indicates the DUT whether the operation to be performed is read or write. It is used in memory drivers.
valid	input	1	the valid signal indicates the successive module to start its operation. It is to be kept high for a single cycle and is atomic in nature.
rdy	output	1	the rdy signal indicates that DUT is ready to receive and process signals.
valid	output	1	the rx pin is used to transmit the data to the UART module, to be send to the computer.
dataout	output	1	the 8 bit wide data out is used to transmit data to the UART to be send to the computer.
sync	input	1	the sync signal indicates that UART is ready to receive the data from the DUT or any other preceding module.

Table 4.5: Hot Key Handshaking Signals Description

4.4 Design Considerations

The state machines designed over the course of this thesis follow mostly the Moore paradigm wherein the output is assigned at the assumed state.

4.4.1 State Machines

The state machine for the transmitter part of the UART can be seen below, the transmitter simply creates a start condition and then send the bits as per the given baud rate.

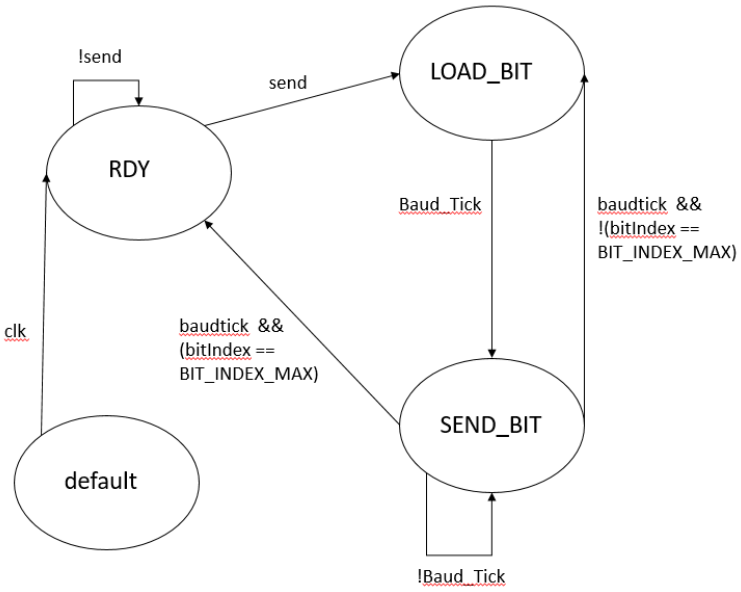


Figure 4.6: UART Transmitter Finite State Machine

The state machine for the receiving part is as shown below, the receive detects the start condition followed by asynchronous sampling of 16 times the baud rate wherein the 8 sample is taken which is in turn synced with the start bit detection condition. the sampling is done as the setup is asynchronous and self-synchronization mechanism is needed for the design.

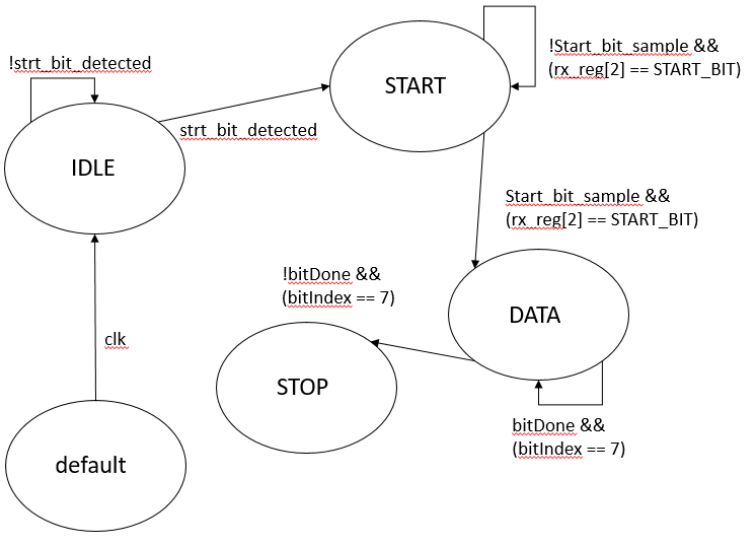


Figure 4.7: UART Receiver Finite State Machine

5 ADC Subsystem Verification

This chapter will deal with implementation as well as verification of SPI based interface controlling the ADC on the board. The state machine, wave snapshot as well oscilloscope result is documented in this chapter.

5.1 Logic Wrapper

The top level module or the main logic wrapper is as shown below along with the description of all the signals going into and coming out of the design.

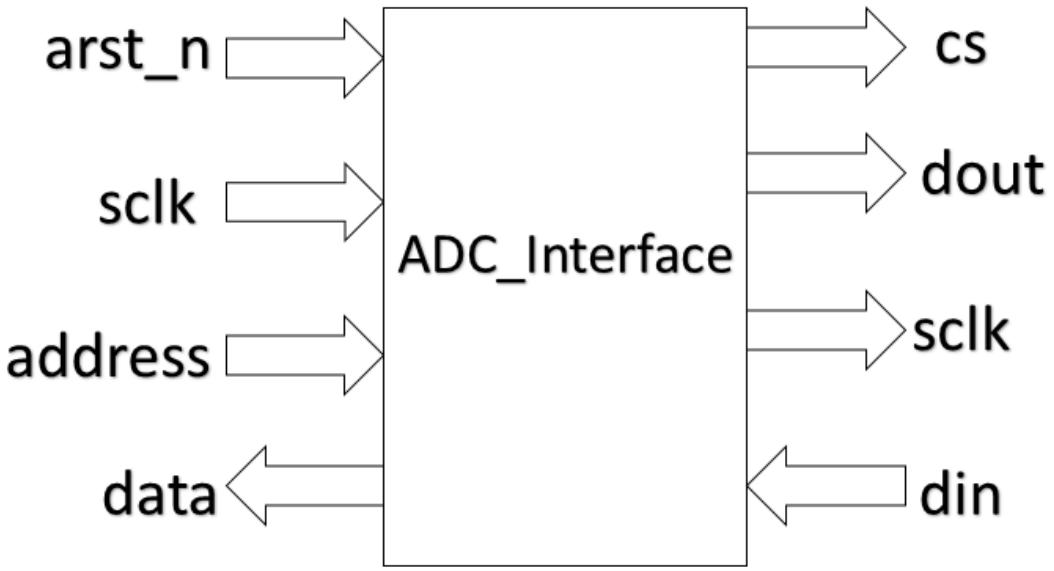


Figure 5.1: ADC Top Level Wrapper

Pin	Direction	Bits	Summary
sclk	input	1	The clock signal is basic signal at the positive edge of which state machine moves forward and events are triggered.
arstn	input	1	the reset signal is used to bring back the state machine to its idle state and refresh all registers to initial value.
address	input	3	the address pin is used select 4 of the 8 channels of the ADC over the course of this thesis.
data	output	12	the data sampled by the ADC is send to and displayed via this 12 bit wide register.
cs	output	1	this pin is pulled low to activate the ADC and start sending the sclk to clock the output.
dout	output	1	this signal is one bit part of SPI output from the FPGA to the ADC and is in-sync with the sclk
address	input	2	the address pin is used here to directly select the required channel and bypass the sampling and directly trigger the right channel.
sclk	output	1	the output sclk is directly connected to input sclk adn is received onto ADC chip via cs signal.
din	input	1	this signal is 1 bit input from the ADC and is in-sync with the sclk clock from the FPGA

Table 5.1: ADC Controller Signals Description

5.2 Chip Performance, Timing and Operational Requirements

The ADC128S102 is a low-power, eight-channel CMOS 12-bit analog-to-digital converter specified for conversion throughput rates of 500 ksps to 1 MSPS. The converter is based on a successive-approximation register architecture with an internal track-and-hold circuit. It can be configured to accept up to eight input signals at inputs IN0 through IN7.

The performance conditions including power supply requirements and the allowable clock frequencies are as given in the table below:

	MIN	MAX	UNIT
Operating Temperature, T_A	−40	105	°C
V_A Supply Voltage	2.7	5.25	V
V_D Supply Voltage	2.7	V_A	V
Digital Input Voltage	0	V_A	V
Analog Input Voltage	0	V_A	V
Clock Frequency	8	16	MHz

Figure 5.2: Performance Conditions of the ADC Circuit

The timing diagram including the command structure of the system is as shown in the diagram below:

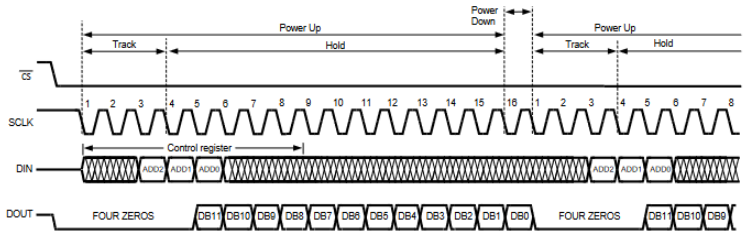


Figure 1. ADC128S102 Operational Timing Diagram

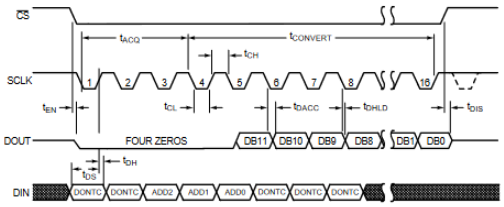


Figure 5.3: Timing Diagram of the ADC Chipset

5.3 Driver Design Considerations

The driver for the ADC module comprises of RAM module and a sampling module. All the 8 channel are sampled and stored in the RAM. Hence each channel takes and overall 16 cycles and then at end for all the 8 channels we have a cumulative of 128 cycles,i.e, the reading for even a single channel are obtained after 128 cycles. Note that this setup can also be modified as such we get the readings per channel within the 16 cycles, however, all other channels will be skipped in this case and only the designated channel will be obtained.

5.4 Probing Path integration

The probing path for the ADC can be seen below. The user sends a signal to the ADC module via the UART or the selected channel specified in the Hot Key Mux, to be forwarded and displayed onto the Hterminal. The binary 12 bit value obtained can be displayed both on Hterminal as well as the oscilloscope to be verified as per the injected DC signal into the ADC.

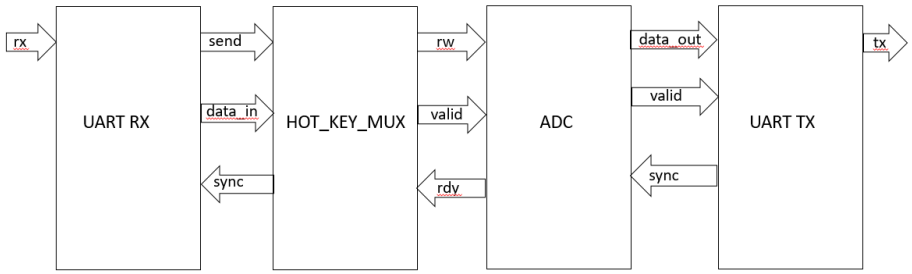


Figure 5.4: ADC Probing Path Design

5.5 Hardware Test, Hterminal and Oscilloscope

The setup was connected with the computer via the UART and Hterm (the snapshot of which is attached in this section) was used to interact with the system, while the DC voltage was injected and and displayed on the oscilloscope present on the electronics test bench.

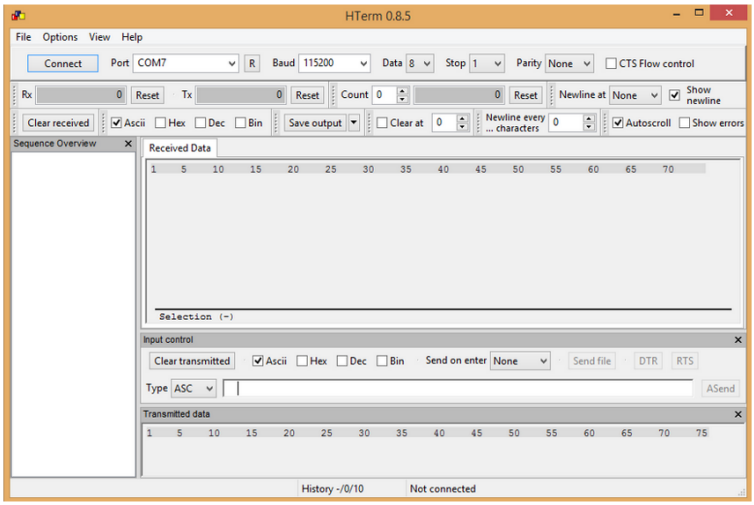


Figure 5.5: Terminal Snapshot of received binary signal

5.6 Measurement Results

The following table gives the values received from the board with any keystroke stimuli given to the system to acquire a reading from the ADC. Two input voltages were provided and the last 8 bits received from the board are as shown in the table.

Stimuli	Input (Voltage)	Output (Binary last 8 bits)
keystroke	0	0000 0000
keystroke	2.5	0001 1111

Table 5.2: ADC Measurement Result

6 Memory Subsystem Verification

The memory subsystem on the board comprises of two SRAM memories which are asynchronous, one SPI based FRAM memory, low power DDR memory followed by one SRAM in configuration of QDR I+.

6.1 SRAM

The two SRAMs are a volatile memory from ISSI IS61/64WV204816ALL/BLL which are present on the board with the aim to store operations that require high speed.

6.1.1 Logic Wrapper

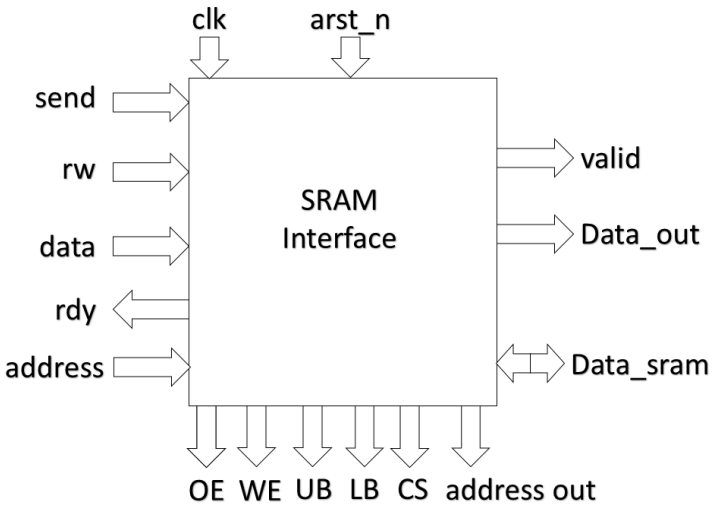


Figure 6.1: SRAM Top Level Wrapper

Pin	Direction	Bits	Summary
clk	input	1	The clock signal is basic signal at the positive edge of which state machine moves forward and events are triggered.
arstn	input	1	the reset signal is used to bring back the state machine to its idle state and refresh all registers to initial value.
send	input	1	the send pin is kept high when the user wishes to send the signal to SRAM. It is an atomic operation.
rw	input	1	the pin to indicate whether the operation to be performed should be read or write.
data	input	16	the datas to be transmitted and eventually stored on the SRAM is send via this 16 bit wide register.
rdy	output	1	the output pin to indicate that the SRAM module is not busy and is ready to receive signals. It is part of handshaking mechanism of design.
address	input	21	the 21 bit wide register is used to address the row of the SRAM for storing or data retrieving operation.
valid	output	1	the signal on the valid pin is kept high for one cycle to trigger the successive module and indicate the validity of the data.
dataout	output	1	an 8 bit wide register that is used to send the data from the module onto the Hterminal for display on the computer.
datasram	in/out	1	a bi-directional data bus that is used to send and retrieve data from the SRAM and is controlled by tri-state buffer in the design.
OE	output	1	the output enable signal that is used to activate the memory for read write operations.
WE	output	1	the write enable pin with which it becomes possible to write the data onto the desired memory address.
UB	output	1	the pin used to select upper bits of the data to be addressed.
LB	output	1	the pin used to select upper bits of the data to be addressed.
CS	output	1	the pin used to activate the internal power and data powering circuitry on the chip.
addressout	output	1	the register pins with which address is provided to the SRAM chipset.

Table 6.1: SRAM Controller Signals Description

6.1.2 Chip Performance and Operational Requirements

The ISSI IS61/64WV204816ALL/BLL are high-speed, 32M bit static RAMs organized as 2048K words by 16 bits. It is fabricated using ISSI's high-performance CMOS technology. This highly reliable process coupled with innovative circuit design techniques, yields high-performance and low power consumption devices. When CS is HIGH (deselected), the device assumes a standby mode at which the power dissipation can be reduced down with CMOS input levels. Easy memory expansion is provided by using Chip Enable and Output Enable inputs. The active LOW Write Enable (WE) controls both writing and reading of the memory. A data byte allows Upper Byte (UB) and Lower Byte (LB) access.

Symbol	Parameter	Value	Unit
Vterm	Terminal Voltage with Respect to VSS	–0.5 to V_{DD} + 0.5V	V
V_{DD}	V_{DD} Related to VSS	–0.3 to 4.0	V
tStg	Storage Temperature	–65 to +150	°C
P_T	Power Dissipation	1.0	W

Figure 6.2: SRAM Operating Conditions

Parameter	Symbol	-10(1)		-12(1)		unit	notes
		Min	Min	Min	Max		
Read Cycle Time	tRC	10	-	12	-	ns	
Address Access Time	tAA	-	10	-	12	ns	
Output Hold Time	tOHA	2.5	-	2.5	-	ns	
CS# Access Time	tACE	-	10	-	12	ns	
OE# Access Time	tDOE	-	6	-	7	ns	
OE# to High-Z Output	tHZOE	0	5	0	6	ns	2
OE# to Low-Z Output	tLZOE	0	-	0	-	ns	2
CS# to High-Z Output	tHZCE	0	5	0	6	ns	2
CS# to Low-Z Output	tLZCE	3	-	3	-	ns	2
UB#, LB# Access Time	tBA	-	6	-	7	ns	
UB#, LB# to High-Z Output	tHZB	0	5	0	6	ns	2
UB#, LB# to Low-Z Output	tLZB	0	-	0	-	ns	2

Figure 6.3: SRAM Read Cycle Latencies

Parameter	Symbol	-10(1)		-12(1)		unit	notes
		Min	Max	Min	Max		
Write Cycle Time	tWC	10	-	12	-	ns	
CS# to Write End	tSCS	8	-	9	-	ns	
Address Setup Time to Write End	tAW	8	-	9	-	ns	
UB#,LB# to Write End	tPWB	8	-	9	-	ns	
Address Hold from Write End	tHA	0	-	0	-	ns	
Address Setup Time	tSA	0	-	0	-	ns	
WE# Pulse Width	tPWE1	8	-	9	-	ns	
WE# Pulse Width (OE# = LOW)	tPWE2	10	-	12	-	ns	2
Data Setup to Write End	tSD	6	-	7	-	ns	
Data Hold from Write End	tHD	0	-	0	-	ns	
WE# LOW to High-Z Output	tHZWE	-	4	-	5	ns	
WE# HIGH to Low-Z Output	tLZWE	2	-	2	-	ns	

Figure 6.4: SRAM Write Cycle Latencies

6.1.3 FSM and timing design considerations

The Finite State Machine for the SRAM can be seen below. As the system is asynchronous, we rely of the time period between the two cycles of the clock to ensure that correct operation is performed. In this thesis, a large margin was maintained which ensured that always correct operation was performed and the time was a lot more than safety margin.

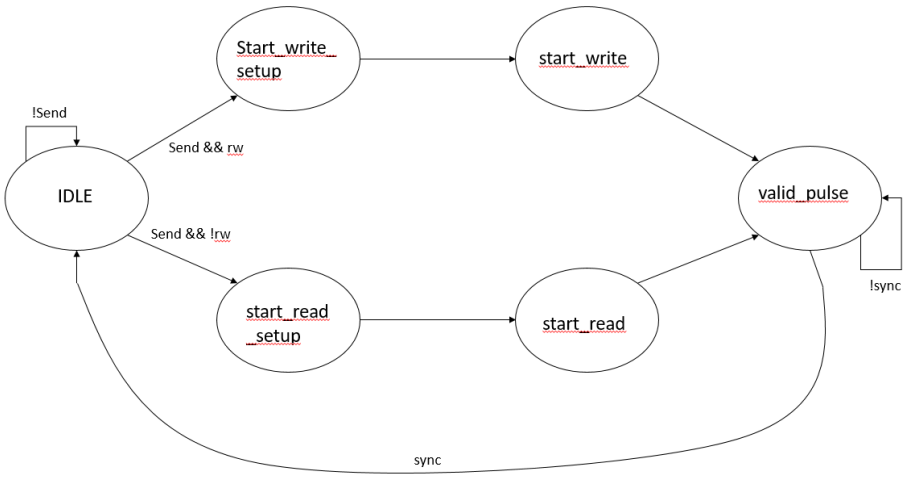


Figure 6.5: SRAM Finite State Machine

6.1.4 Probing Path and Test Results

The Probing Path based testing mechanism for the SRAM can be seen below. The user sends a signal via the UART to perform read or write operation on the SRAM via the Hot Key Mux with the given data stored in the same module. By performing write and read operation on a memory location, the data outputted can be verified via the reading shown on the Hterminal.

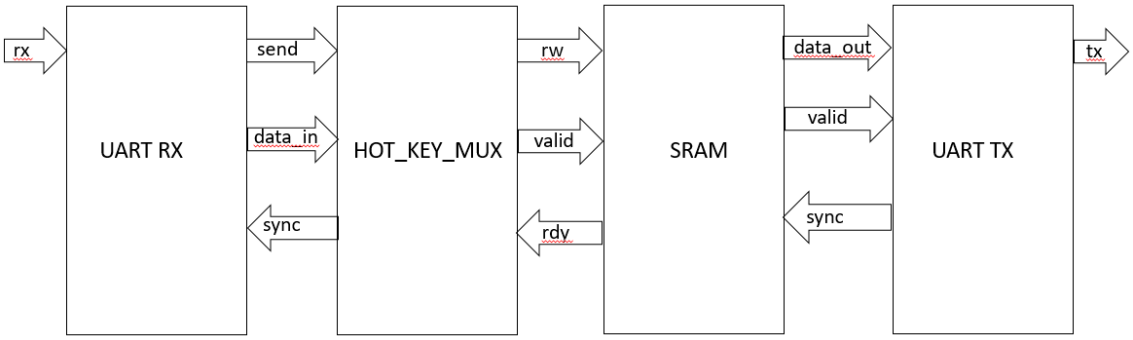


Figure 6.6: SRAM Probing Path Setup

6.1.5 Measurement Results

The table below shows the readings obtained form the SRAM stimuli. the read and write stimuli were given to the board and the power supply was also checked to check the volatile nature of the SRAM.

Stimuli	Power supply	Input	Output (last 8 bits)
w	On	0000 0000	xxxx xxxx
r	On	0001 1111	0000 0000
w	On	1100 1100	0000 0000
r	On	0001 1111	1100 1100
-	Off	xxxx xxxx	xxxx xxxx
r	On	0001 1111	xxxx xxxx
w	On	1101 1101	xxxx xxxx
r	On	1100 1100	1101 1101

Table 6.2: SRAM Measurement Result

6.2 FRAM

The FRAM from Cypress Semiconductor (Infineon) is a non-volatile memory with the aim of storing information which is needed for high retention and retrieval in case of power failure. Also given the long retention time, it is possible to introduce am embedded file system based on this storage as well.

6.2.1 Logic Wrapper

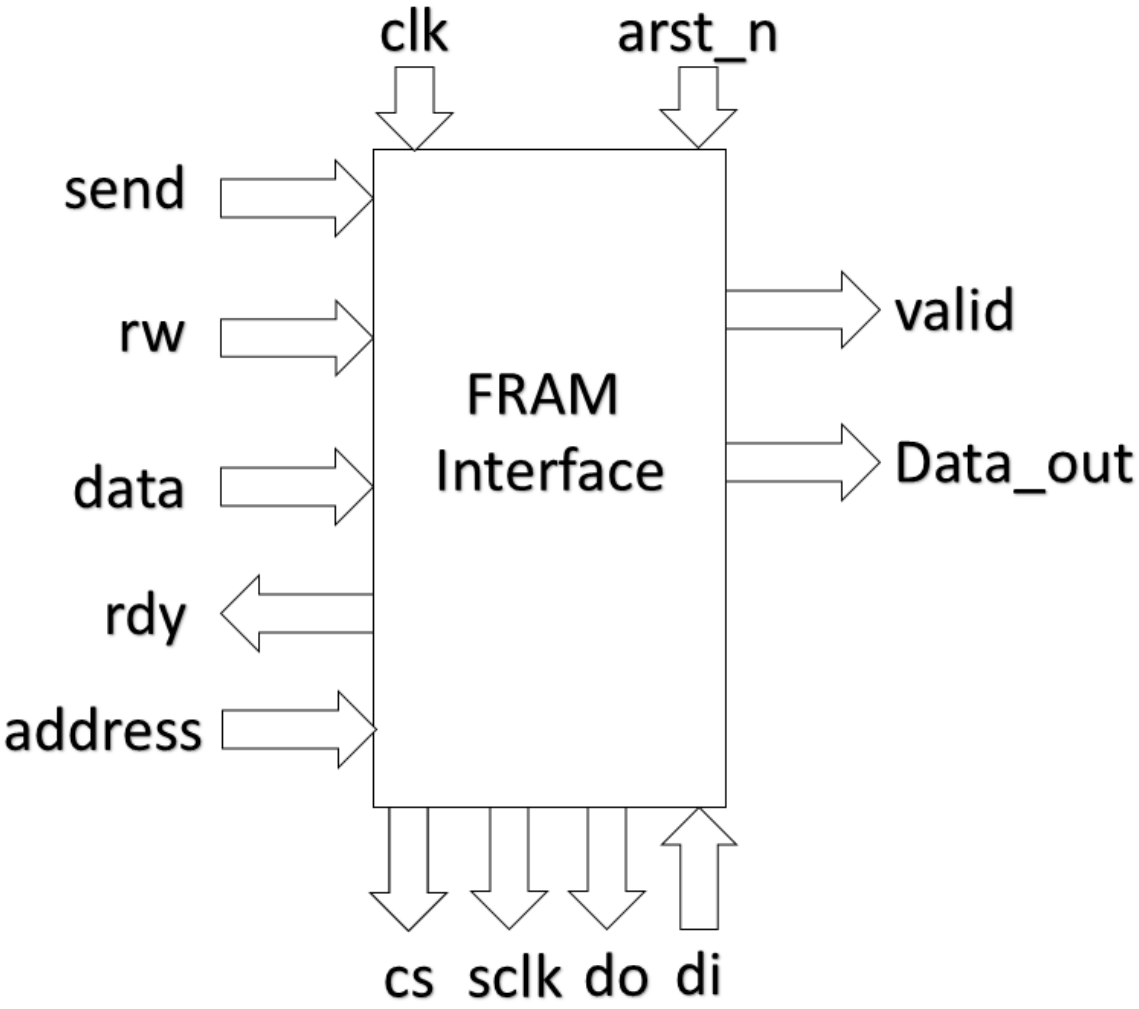


Figure 6.7: FRAM Top Level Wrapper

Pin	Direction	Bits	Summary
clk	input	1	The clock signal is basic signal at the positive edge of which state machine moves forward and events are triggered.
arstn	input	1	the reset signal is used to bring back the state machine to its idle state and refresh all registers to initial value.
send	input	1	the send pin is kept high when the user wishes to send the signal to FRAM. It is an atomic operation.
rw	input	1	the pin to indicate whether the operation to be performed should be read or write.
data	input	8	the data to be transmitted and eventually stored on the FRAM is send via this 8 bit wide register.
rdy	output	1	the output pin to indicate that the FRAM module is not busy and is ready to receive signals. It is part of handshaking mechanism of design.
address	input	24	the 24 bit wide register is used to address the row of the FRAM for storing or data retrieving operation.
valid	output	1	the signal on the valid pin is kept high for one cycle to trigger the successive module and indicate the validity of the data.
dataout	output	1	an 8 bit wide register that is used to send the data from the module onto the Hterminal for display on the computer.
cs	output	1	this pin is pulled low to activate the FRAM and start sending the sclk to clock the output.
sclk	output	1	the output sclk is directly connected to input sclk adn is received onto FRAM chip via cs signal.
do	output	1	this signal is one bit part of SPI output from the FPGA to the FRAM and is in-sync with the sclk
di	input	1	this signal is 1 bit input from the FRAM and is in-sync with the sclk clock from the FPGA

Table 6.3: FRAM Signals Description

6.2.2 Chip Performance and Operational Requirements

The Excelon LP CY15X108QN is a low power, 8-Mbit nonvolatile memory employing an advanced ferroelectric process. A ferro-electric random access memory or F-RAM is nonvolatile and performs reads and writes similar to a RAM. It provides reliable data retention for 151 years while eliminating the complexities, overhead, and system-level reliability problems caused by serial flash, EEPROM, and other nonvolatile memories.

The CY15X108QN will power up with writes disabled. The WREN command must be issued before any write operation. Sending the WREN opcode allows the user to issue subsequent opcodes for write operations. These include writing to the Status Register (WRSR), the memory (WRITE), Special Sector (SSWR), and Write Serial Number (WRSN).

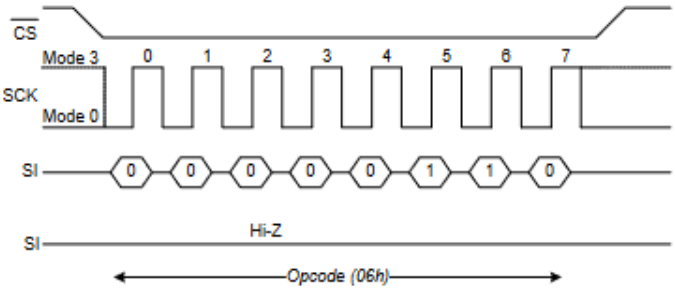


Figure 6.8: FRAM WREN Command

All writes to the memory begin with a WREN opcode with CS being asserted and deasserted. The next opcode is WRITE. The WRITE opcode is followed by a three-byte address containing the 20-bit address (A19–A0) of the first data byte to be written into the memory. The upper four bits of the three-byte address are ignored. Subsequent bytes are data bytes, which are written sequentially. Addresses are incremented internally as long as the bus master continues to issue clocks and keeps CS LOW. If the last address of FFFFFh is reached, the internal address counter will roll over to 00000h. Every data byte to be written is transmitted on SI in 8-clock cycles with MSb first and the LSb last. The rising edge of CS terminates a write operation.

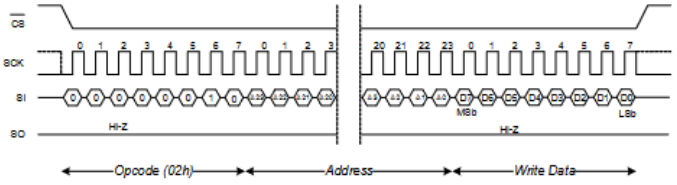


Figure 6.9: FRAM Write Timing Diagram

After the falling edge of CS, the bus master can issue a READ opcode. Following the READ command is a three-byte address containing the 20-bit address (A19–A0) of the first byte of the read operation. The upper four bits of the address are ignored. After the opcode and address are issued, the device drives out the read data on the next eight clocks. The SI input is ignored during read data bytes. Subsequent bytes are data bytes, which are read out sequentially. Addresses are incremented internally as long as the bus master continues to issue clocks and CS is LOW. If the last address of FFFFFh is reached, the internal address counter will roll over to 00000h. Every read data byte on SO is driven in 8-clock cycles with MSb first and the LSb last. The rising edge of CS terminates a read operation and tristates the SO pin.

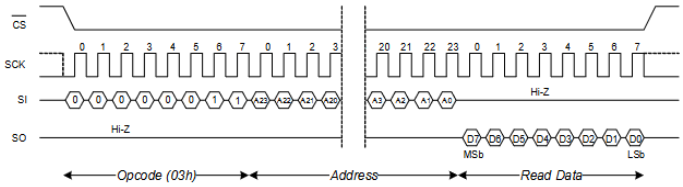


Figure 6.10: FRAM Read Timing Diagram

6.2.3 FSM and timing design considerations

The finite state machine representation of the FRAM can be seen below. The state machine accounts for both read and write operations which can be selected by sending the relevant read or write command respectively. Note that FRAM is using and SPI based protocol and as such everything is in sync with the positive edge of the clock, and hence the setup works fine provided that we keep the system within the appropriate operating frequency range.

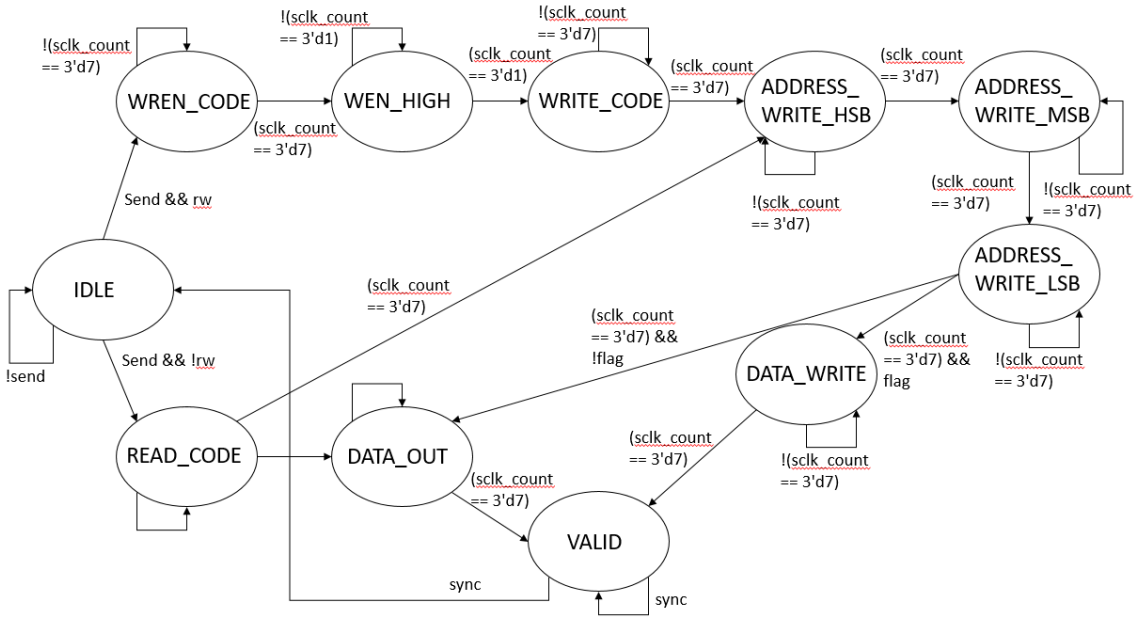


Figure 6.11: FRAM Finite State Machine

6.2.4 Probing Path and Test Results

The Probing path based verification circuitry for the FRAM can be seen below. As in the case of SRAM, the user sends a signal to perform read or right on the FRAM memory location with the data already stored in the Hot Key Mux module. The reading can then be displayed on the Hterminal which can be used to verify if the data was correctly stored or not.

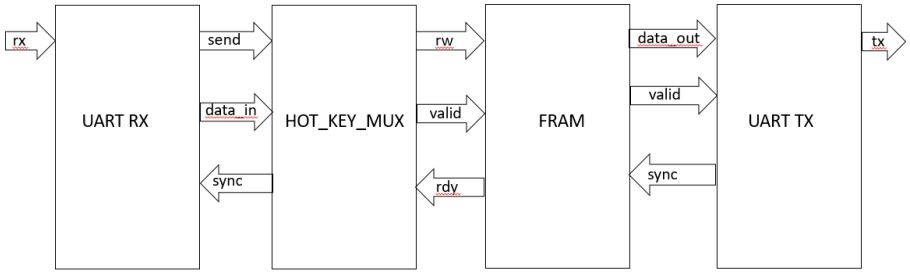


Figure 6.12: FRAM Finite State Machine

6.2.5 Measurement Results

The table below shows the readings obtained form the FRAM stimuli. the read and write stimuli were given to the board and the power supply was also checked to check the volatile nature of the FRAM.

Stimuli	Power supply	Input	Output (last 8 bits)
w	On	0000 0000	xxxx xxxx
r	On	0001 1111	0000 0000
w	On	1100 1100	0000 0000
r	On	0001 1111	1100 1100
-	Off	xxxx xxxx	xxxx xxxx
r	On	0001 1111	1100 1100
w	On	1101 1101	xxxx xxxx
r	On	1100 1100	1101 1101

Table 6.4: FRAM Measurement Result

6.3 DDR Memories

The DDR memories are a category of synchronous DRAM wherein the banks are organized as such it becomes possible to read multiple data at a single clock cycle. On this board, the QDR II+ is instead based on SRAM technology wherein also it is possible to read multiple data within single clock cycle. Note that the Polarfire contains controller for these memories within is fabric and as such the logic for the control of the embedded controllers has been developed over the course of this thesis.

6.3.1 LPDDR/DDR

Logic Wrapper

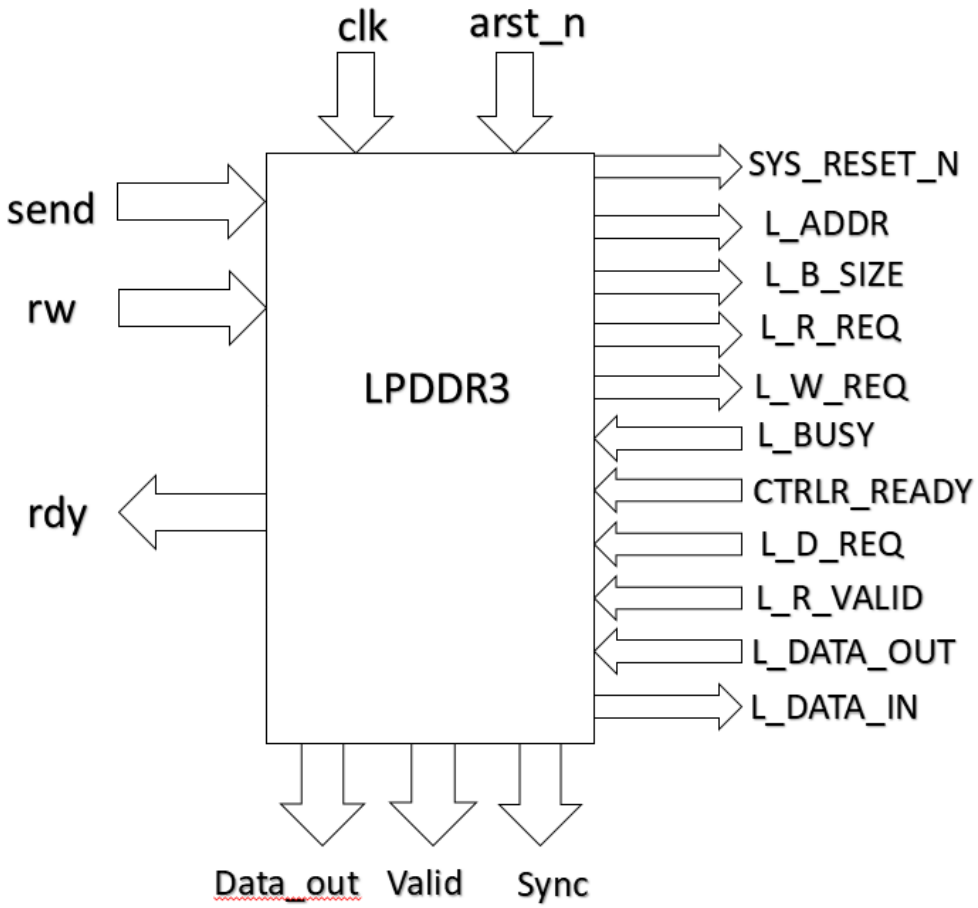


Figure 6.13: LPDDR Top Level Wrapper

Pin	Direction	Bits	Summary
clk	input	1	The clock signal is basic signal at the positive edge of which state machine moves forward and events are triggered.
arstn	input	1	the reset signal is used to bring back the state machine to its idle state and refresh all registers to initial value.
send	output	1	the send pin is kept high when the user wishes to send the signal to LPDDR3. It is an atomic operation.
rw	input	1	the pin to indicate whether the operation to be performed should be read or write.
rdy	input	1	the output pin to indicate that the LPDDR3 module is not busy and is ready to receive signals. It is part of handshaking mechanism of design.
Dataout	output	8	an 8 bit wide register that is used to send the data from the module onto the Hterminal for display on the computer.
Valid	input	1	the signal on the valid pin is kept high for one cycle to trigger the successive module and indicate the validity of the data. It is an atomic operation.
Sync	input	1	It is used by successive module that it is now ready to receive signal or data, which in this case would be UART transmitter to the PC.

Table 6.5: LPDDR3 Controller Side Signals Description

Pin	Direction	Bits	Summary
SYSRESETN	output	1	Active-low asynchronous system reset.
LADDR	output	38	Native interface address sizes: DDR4 = 39 bits, DDR3 = 36 bits, and LPDDR3 = 36 bits.
LBSIZE	output	11	Native interface burst length in terms of bytes. It must be in multiples of the native interface bus width.
LRREQ	output	1	Native interface read request
LWREQ	output	1	Native interface write request.
LBUSY	input	1	Specifies that the subsystem is busy and is not accepting new requests. A command is accepted on any clock cycle where LRREQ or LWREQ is set, and LBUSY is low. If LBUSY is high when LRREQ or LWREQ is set, the request may be kept asserted (along with the desired LADDR, LBSIZE and LAUTOPCH values) until LBUSY goes low.
CTRLRREADY	input	1	Causes the subsystem to reissue the initialization sequence to the SDRAM. The initialization begins when a low-to-high transition is detected on the input.
LDREQ	input	1	Requests data on the native interface write data bus (LDATAIN) during a write transaction. Asserts one clock cycle prior to when data is required.
LRVALID	input	1	Data-valid indication for data on the native interface read data bus (LDATAOUT).
LDATAIN	input	256	Input data bus. This data bus is eight times the width of the SDRAM device data bus.
LDATAOUT	input	256	Output data bus. This data bus is twice the width of the SDRAM device data bus.

Table 6.6: LPDDR3 Memory Side Signals Description

Chip Performance and Operational Requirements

The smart design based diagram for generating clock, reset as well as calibration signals for the memory attached can be seen in the diagram below.

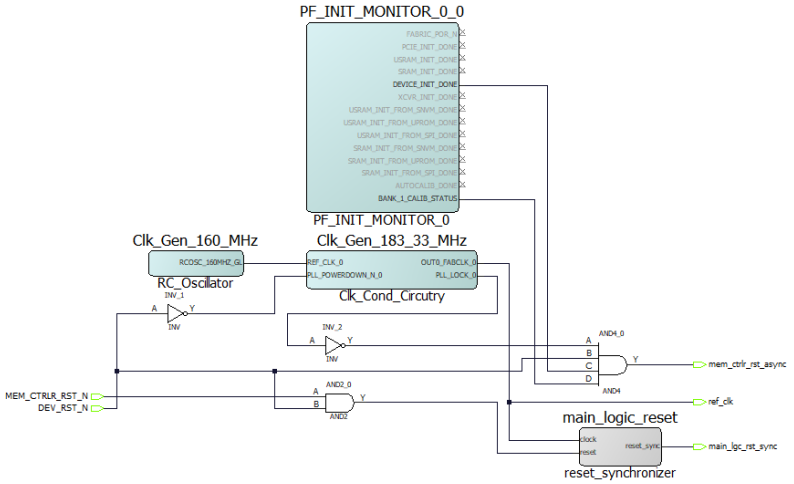


Figure 6.14: System Interconnect view for Clock and Reset Circuitry

The smart design based diagram for controlling the LPDDR3 RAM can be seen in the diagram below. Note that to test this picture is from a sample project which was used to test the sample system on the prototyping board from microsemi and as such utilized the AXI4 interconnects.

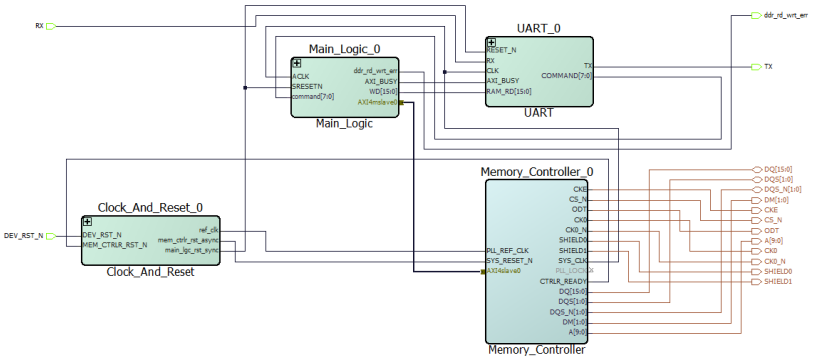


Figure 6.15: System Interconnect view for LPDDR3 Controller

FSM and timing design considerations

The finite state machine for controlling the LPDDR3 can be seen below. It can be seen that before every read or write the system is first reset, in order to ensure proper functioning followed by waiting for the controller to respond. Eventually valid is asserted for as long as the sync signal is not generated by the following module for the controller to be read where upon the system returns to the idle state.

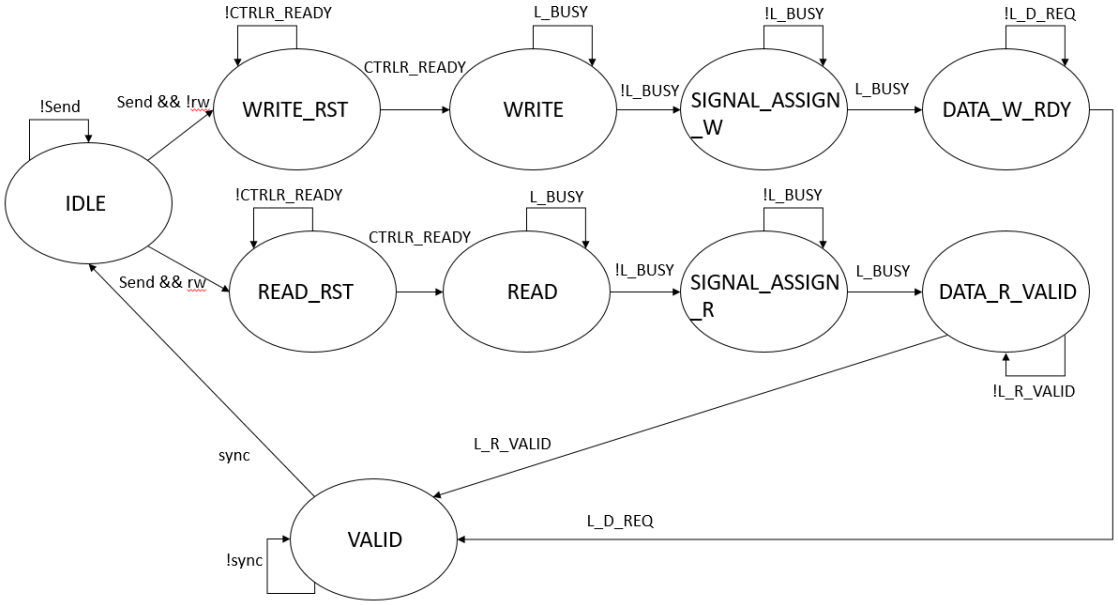


Figure 6.16: LPDDR3 Finite State Machine

6.3.2 QDR II+

Logic Wrapper

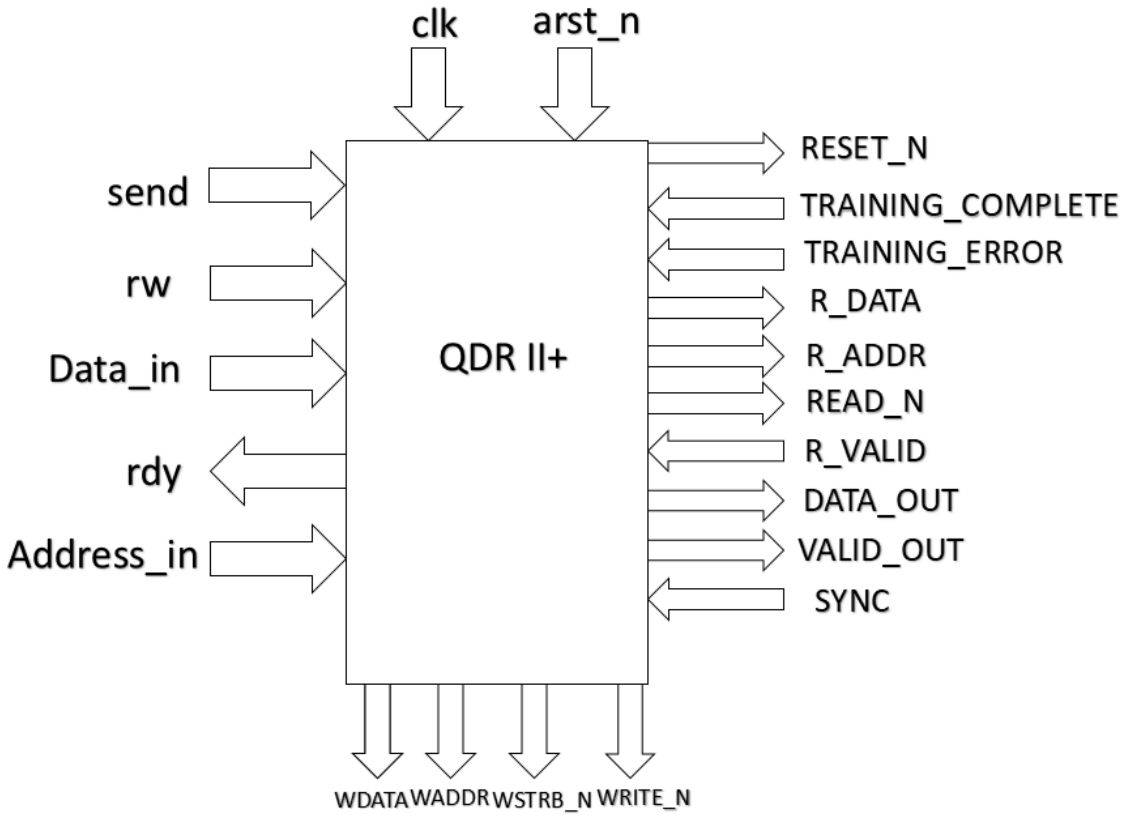


Figure 6.17: QDR II+ Top Level Wrapper

Pin	Direction	Bits	Summary
clk	input	1	The clock signal is basic signal at the positive edge of which state machine moves forward and events are triggered.
arstn	input	1	the reset signal is used to bring back the state machine to its idle state and refresh all registers to initial value.
send	output	1	the send pin is kept high when the user wishes to send the signal to QDR II+. It is an atomic operation.
rw	input	1	the pin to indicate whether the operation to be performed should be read or write.
rdy	input	1	The output pin to indicate that the QDR II+ module is not busy and is ready to receive signals. It is part of handshaking mechanism of design.
DATAOUT	output	8	an 8 bit wide register that is used to send the data from the module onto the Hterminal for display on the computer.
VALIDOUT	output	1	the signal on the valid pin is kept high for one cycle to trigger the successive module and indicate the validity of the data. It is an atomic operation.
SYNC	input	1	It is used by successive module that it is now ready to receive signal or data, which in this case would be UART transmitter to the PC.

Table 6.7: QDR II+ Controller Side Signals Description

Pin	Direction	Bits	Summary
RESETN	input	1	Active-low asynchronous system reset.
TRAININGCOMPLETE	input	1	Indicates that Training is complete. The user logic must check for TRAININGCOMPLETE = 1 and TRAININGERROR = 0 before accessing the QDR memory.
TRAININGERROR	input	3	– Bit 0: Reserved – Bits [2:1]: 0: No error. 1: Timeout error, data VALID is received during Q training. 2: No solution found during training. If no solution is found, restart the Training process described.
RDATA	output	288	Read data, valid when RVALID is asserted.
RADDR	output	40	Read addresses for each beat. Width is (QDR SRAM Address Width)*4 if burst size is 2, (QDR SRAM Address Width)*2 if burst size is 4 In Burst of 2: ADDR3, ADDR2, ADDR1, ADDR0 In Burst of 4: ADDR1, ADDR0
READN	input	2	Read request with address specified on RADDR bus. 4-bit Wide if Burst Size is 2. 2-bit Wide if Burst Size is 4.
RVALID	input	1	Read data valid.
WDATA	input	288	Write data.
WADDR	input	40	Write addresses for each beat. Width is (QDR SRAM Address Width)*4 if burst size is 2, (QDR SRAM Address Width)*2 if burst size is 4 In Burst of 2: ADDR3, ADDR2, ADDR1, ADDR0 In Burst of 4: ADDR1, ADDR0.
WSTRBN	input	32	Write strobe for write transaction.
WRITEN	input	2	Write request with address specified on WADDR bus. 4-bit Wide if Burst Size is 2. 2-bit Wide if Burst Size is 4.

Table 6.8: QDR II+ Memory Side Signals Description

Chip Performance and Operational Requirements

The smart design based diagram for generating clock, reset as well as calibration signals for the memory attached can be seen in the diagram below. The calibration part ensures that voltage signal and banks within FPGA are setup for the smooth functioning of the system.

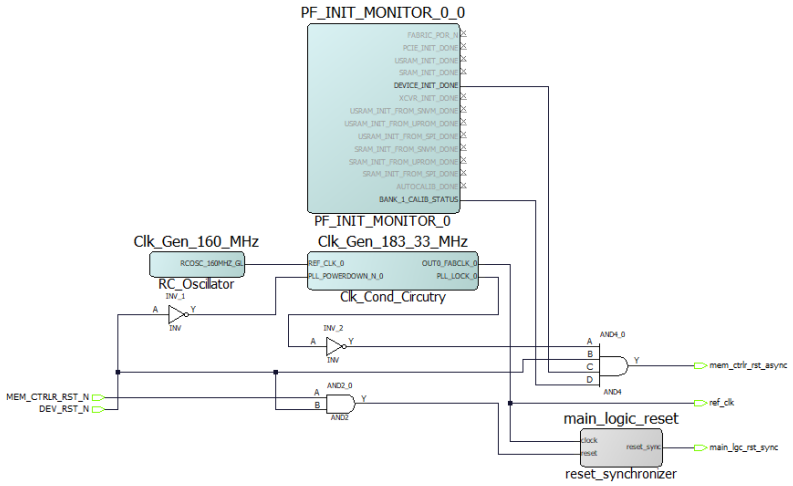


Figure 6.18: System Interconnect view of Clock and Reset Circuitry

The smart design based diagram for controlling the QDR II+ RAM can be seen in the diagram below. In this case, unlike for the LPDDR3, we have used the native interface to interact with QDR II + memory controller.

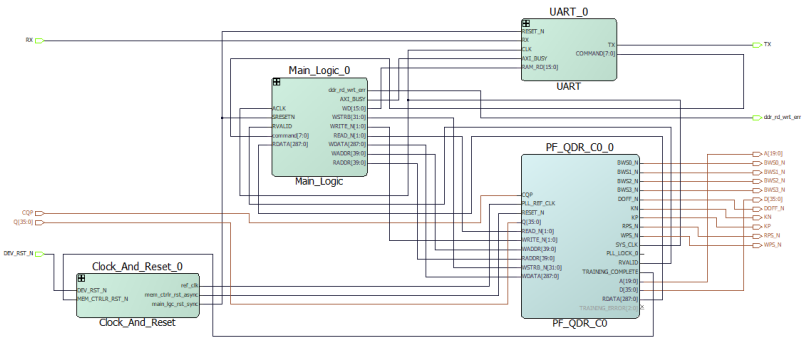


Figure 6.19: System Interconnect view of QDR II+ Memory Controller

FSM and timing design considerations

The finite state machine for controlling the QDR II+ module can be seen below. The system is first adjusted and trained for properly setting up voltages for the board followed by the read or write operations respectively.

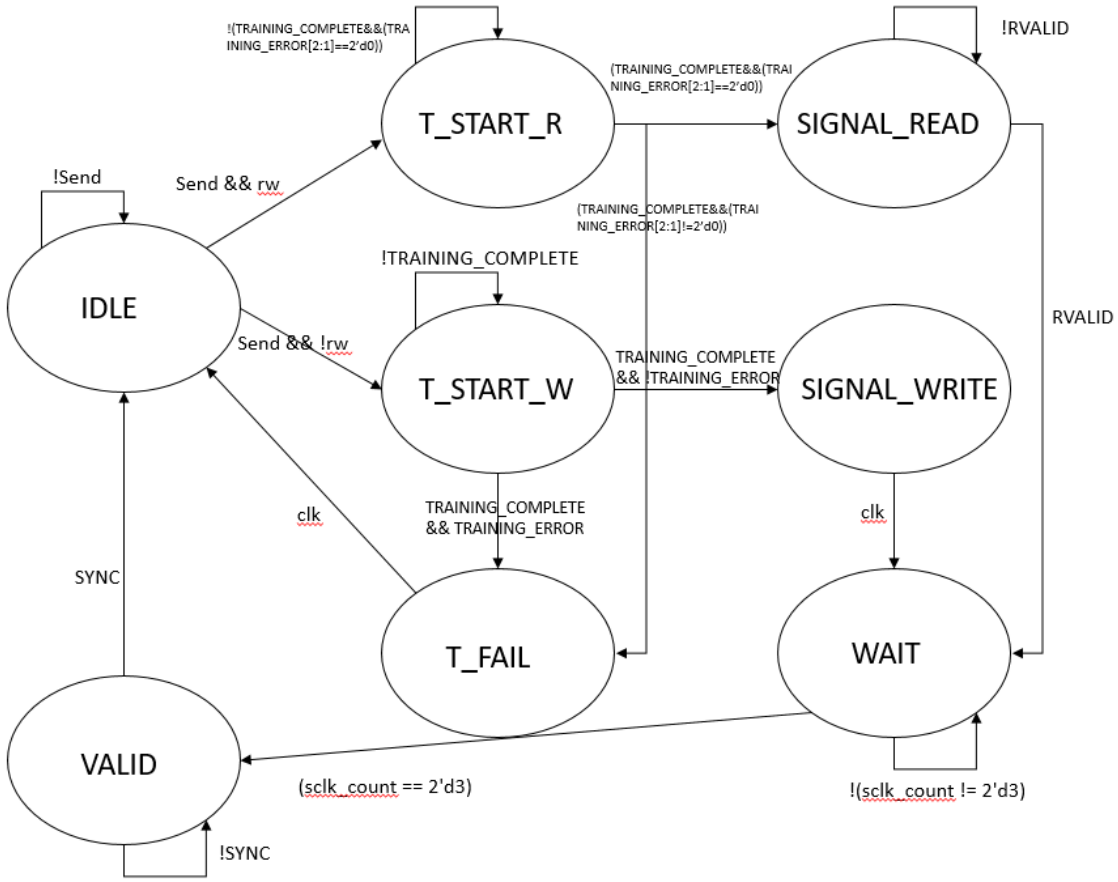


Figure 6.20: QDR II+ Finite State Machine

7 Concept study of I2C bus sub-system Verification

This chapter will summarize the work done in the thesis with regards to the creation of I2C sub-system and will give insights into how it can be used in future research for activating and using I2C based power monitoring system on the board.

7.1 Power Monitoring Chipsets and Addressing

The board comes equipped with several power monitoring intergated chips which can be used to sense current, voltage as well as power consumption and are accessible via the I2C protocol. The INA260 is a digital-output, current, power, and voltage monitor with an I2C and SMBus™-compatible interface with an integrated precision shunt resistor. It enables high-accuracy current and power measurements and over-current detection at common-mode voltages that can vary from 0 V to 36 V, independent of the supply voltage. A snap-shot of the chipset configuration can be seen below:

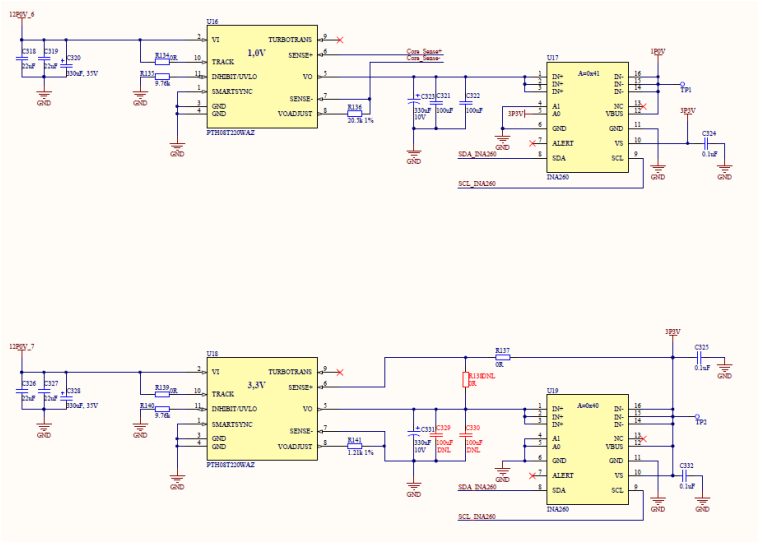


Figure 7.1: I2C based Power Monitoring Circuit

7.2 Top level Wrapper module

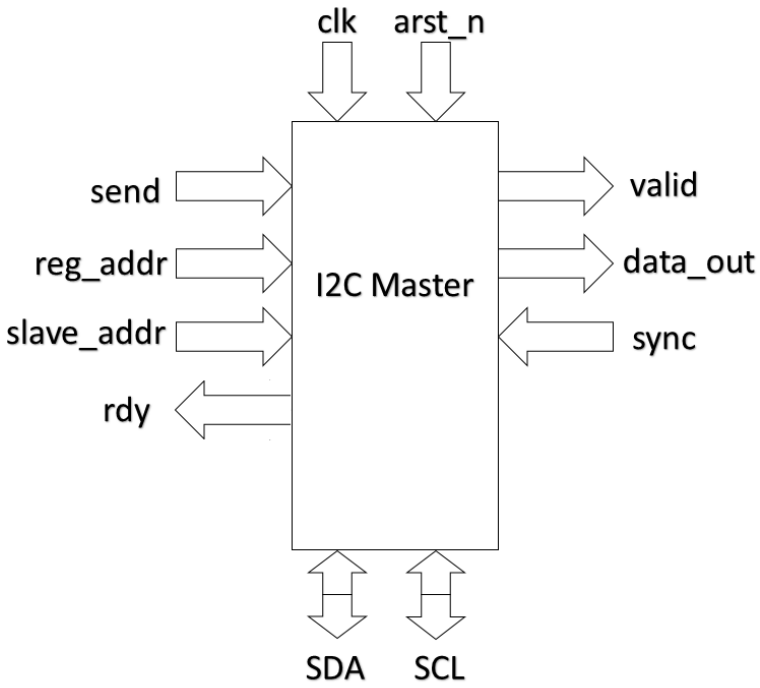


Figure 7.2: I2C Master Top Level Wrapper

Pin	Direction	Bits	Summary
clk	input	1	The clock signal is basic signal at the positive edge of which state machine moves forward and events are triggered.
arstn	input	1	the reset signal is used to bring back the state machine to its idle state and refresh all registers to initial value.
send	input	1	the send pin is kept high when the user wishes to send the signal to I2C module. It is an atomic operation.
regaddr	input	8	the 8 bit wide address register with which individual registers within the slaves chips can be addressed.
slaveaddr	input	8	the 8 bit wide address register with which the slaves power monitoring chips on the board can be addressed.
rdy	output	1	the rx pin is used to transmit the data from the UART module.
valid	output	1	the signal on the valid pin is kept high for one cycle to trigger the successive module and indicate the validity of the data. It is an atomic operation.
dataout	output	8	an 8 bit wide register that is used to send the data from the module onto the Hterminal for display on the computer.
sync	output	1	It is used by sucessive module that it is now ready to receive signal or data, which in this case would be UART transmitter to the PC.

Table 7.1: I2C Master Signals Description

7.3 Design Considerations

Note that I2C is network based protocol with multiple masters, however, over the course of this thesis, feasibility study with only a single master has been performed, since the power monitoring circuitry is all expected to be in slave configuration. The assign addresses of individual chips are configured using the combination of values applied to A1 and A2 in the form of SCL, SDA, VCC and GND. The following table gives the description of sixteen possible address values with this setup.

A1	A0	SLAVE ADDRESS
GND	GND	1000000
GND	VS	1000001
GND	SDA	1000010
GND	SCL	1000011
VS	GND	1000100
VS	VS	1000101
VS	SDA	1000110
VS	SCL	1000111
SDA	GND	1001000
SDA	VS	1001001
SDA	SDA	1001010
SDA	SCL	1001011
SCL	GND	1001100
SCL	VS	1001101
SCL	SDA	1001110
SCL	SCL	1001111

Figure 7.3: I2C Power Chipset Addressing

7.3.1 State Machines

The state machine below shows the chronology of the I2C master interaction with the slave power ICs on the board wherein the master first generated the slave address, awaits an acknowledgement and then proceeds to read relevant regsiters in the slave IC and further on proceeds to terminate the connection at the completion of the transaction.

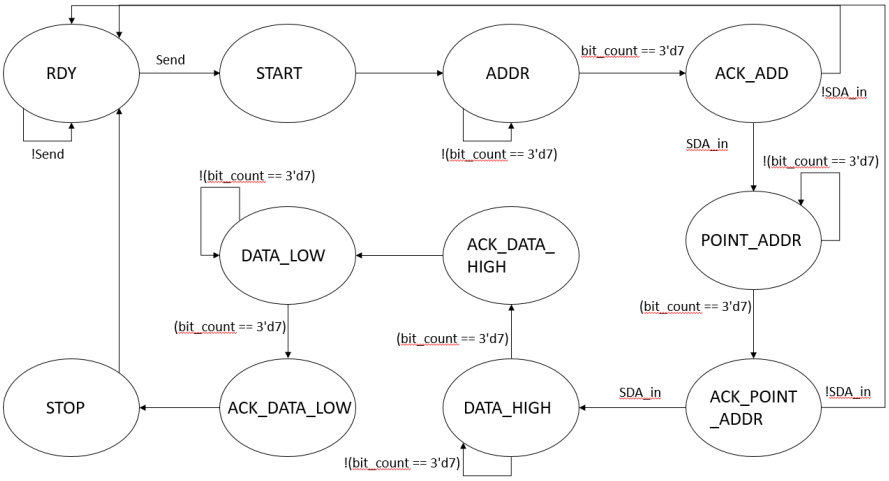


Figure 7.4: I2C Master Finite State Machine

7.3.2 Probing Path integration

The probing path with all relevant handshake signals as before is shown below.

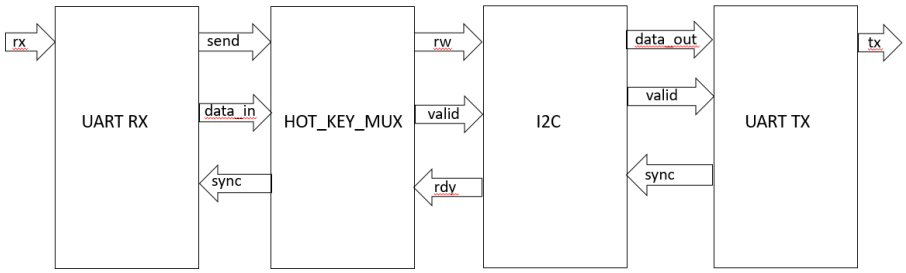


Figure 7.5: I2C Probing Path Setup

8 Mi-V Based System Design

This section deals with the implementation of softcore processor with emphasis to house keeping procedure as a template for a possible further development.

8.1 Processor Organization

The block diagram view of the MiV sub-system can be seen below. The RISC-V core can been seen to have a five stage pipeline coupled with on chip hardware based multiply and divide facilities. In terms of micro-architecture, there is a separate instruction and data cache indicating a Harvard based architecture. A hardware interrupt based circuitry is also integrated with the processing subsystem, allowing for interrupt based system to also function in parallel to main execution logic. A JTAG based debugging system including single step debugging has also been integrated into the whole scheme, to trace the registers in the system. Note that a bootloader based firmware up-gradation is also possible within this processor based system.

Also the entire processing system has interconnection facilities making it possible to developed an entire processing system with timers and memory interfaces as is shown in the diagram.

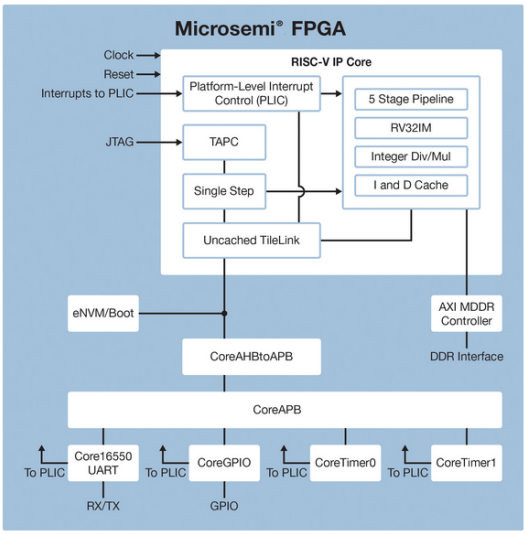


Figure 8.1: MiV Sub-system Block Diagram[41]

8.2 System Interconnect View

A sample interconnect view of the Mi-V processor based system can be seen below. The system controller init block is responsible for generating reset signal for all the components o the chip and does so by monitoring the state of the system-on-chip. There is a CCC block which si a phase lock loop responsible for genrating clcok signals for all the components on the board incluing reset synchronization circuitry, on-board memories, UART Controller, GPIO controller, SPI controller and JTAG debugging system. The UART controller is used to interface system with a terminal, GPIO to the pins on the board, SPI is used to control any SPI based controller on the board. The DDR memory is used to hold the firmware or more precisely the 3 three classical sections of data, text and stack of the program. JTAG based system is used to upload program as well as carry out any debugging on the processor program.

Note that all these individual components are connected via the use of a system-on-chip interconnect, AXI based on AMBA in this case, which can also be seen in the diagram.

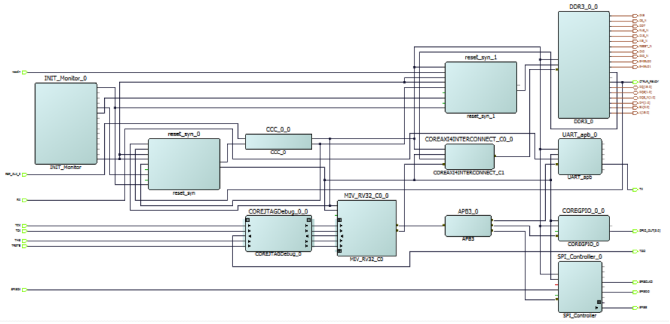


Figure 8.2: MiV System Interconnect View [42]

8.3 Design Flow

A sample design flow is as shown which was also followed over the curse of developing a custom RISC-V subsystem. The architecture was borrowed from the sample design provided by the Microsemi corporation.

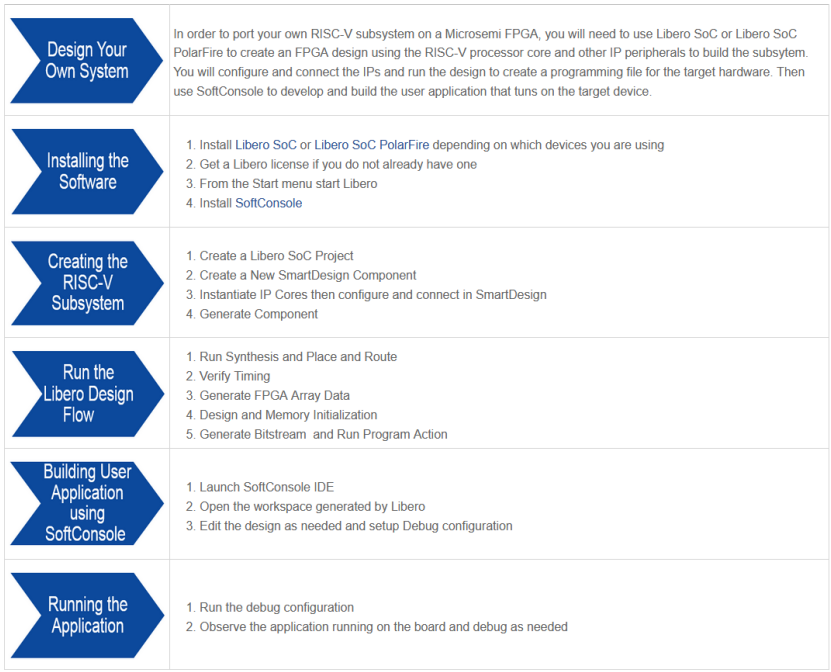


Figure 8.3: Design flow for MiV based System[41]

8.4 Implementation

The implementation steps and their details are enumerated as follows:

1. A sample system-on-chip with the configuration as given above was created from scratch and integrated all of the above IPs and features.

2. The source code was utilized from the example project. It was understood in terms of its interaction with HAL and drivers associated with it.

3. The linker script was also utilized form the sample project and memory locations of every code section was verified. It was ensured that the program code section in the linker file corresponded to the program counter of the micro-controller.

4. The UART driver header file was modified as such the baud rate was modified to match the required 115200.

5. the program was loaded using JTAG connection and the function of UART interaction with the on-board leds was verified.

9 Conclusion

9.1 Summary

A complete setup for the board as well as all the components on it was devised. The components tested and verified included UART subsystem, ADC subsystem, SRAM and FRAM subsystems whereas the DDR memory subsystem couldn't be verified on the board. There was also a feasibility study for developing and I2C based subsystem for power monitoring as well. A complete firmware based system based on MiV softcore processor w as also developed and verified for housekeeping procedures and as a template for further firmware development.

9.2 Future Work

In the future and as an extension of this project, the following tasks can be performed including:

- An Improved ADC testing scheme can be adopted by injecting a sinusoidal wave instead of a fixed DC voltage into the system. Also the fill-scale deflection can be measured to see the accuracy of the ADC.
- In case of memory controller, at the moment simple read and write operation has been carried out with a very large safety margin in terms of operation characteristics. hence in future a randomized read and write operations can be performed coupled with the shifting power and clock frequency margins to verify the integrity of the system.
- A counter based timer debugging circuitry can be integrated into the system as such that in case of any anomalous and undesired behaviour, the timer signals an error generated within the circuit.
- At the moment, a simple single core baremetal solution has been implemented for the firmware part. In future a possible implementation of an RTOS can be done coupled with extension of the system to multiple core. Also further operating system support in the form of embedded Linux can also be explored.

10 Codes and Programs

UART INTERFACE

```
`timescale 1ns / 1ps
module UART_INTERFACE(
        //input       clk   ,//input 50Mhz clock
        input       arst_n,//asynchronous active low reset
        input               rx    ,//rx uart bit
        //output [7:0] led   ,//data received output to leds..can be compared using ascii table
        output      tx     //tx uart bit
);

        wire          valid_rx        ;//valid flag output from Rx Core
        wire     [7:0] data_rx         ;//Vaild Rx data
        reg      [7:0] data_reg        ;//Intermediate data register
        reg      [7:0] data_tx         ;//Valid tx data input to Tx core
        reg           valid_tx        ;//Valid tx flag
        reg           valid_rx_reg    ;//Registered for oneshot generation
        wire                          valid_transHtoL   ;//Valid transition from High to Low flag
        wire          clk;
  wire          ready_tx;

        //UART RX Core
        UART_DRIVER_RX rx_inst(
                .clk        (clk),
                .arst_n (arst_n  ),
                .rx         (rx     ),
                .data   (data_rx ),
                .valid  (valid_rx)
        );

  //Clock Core
  PF_OSC_C1 uut1(
  // Outputs
  .RCOSC_2MHZ_GL(clk)
);


//Rx Valid One Shot Generation to csample the signal at just one clock edge
always @ (posedge clk)
        valid_rx_reg <= valid_rx;

assign valid_transHtoL = valid_rx_reg && ~valid_rx;

//Recieved data is stored until next data is received
```

```
always @(posedge clk,negedge arst_n)
        if (!arst_n)
                data_reg <= {8{1'b0}};
        else if (valid_rx)
                data_reg <= data_rx  ;

//data stored is output to led
//assign led = data_reg;

//data tx registered when tx is ready and valid rx data is received
always @(posedge clk,negedge arst_n)
        if (!arst_n)
                data_tx <= {8{1'b0}};
        else if (valid_transHtoL && ready_tx)
                data_tx <= data_reg+1'b1 ;

//Valid Tx generation when tx is ready and valid rx data is received for one cycle only
always @(posedge clk,negedge arst_n)
        if (!arst_n)
                valid_tx <= 1'b0;
        else if (valid_transHtoL && ready_tx)
                valid_tx <= 1'b1   ;
        else
                valid_tx <= 1'b0   ;

        // UART Tx Control
        UART_DRIVER_TX tx_inst(
                .clk    (clk      ),
                .arst_n  (arst_n    ),
                .send    (valid_tx  ),
                .data_in (data_tx   ),
                .ready   (ready_tx  ),
                .tx     (tx       )
  );

Endmodule
```

UART Transmitter

```
`timescale 1ns / 1ps
module UART_DRIVER_TX#(
	parameter BAUD_RATE = 9600		//1200/2400/4800/9600/19200/115200
)(
	input	clk	,
	input	arst_n ,
	input	send	,
	input	[7:0] data_in,
	output	ready	,
	output	tx
);
//States
localparam	[1:0] RDY	= 2'd0;
localparam	[1:0] LOAD_BIT = 2'd1;
localparam	[1:0] SEND_BIT = 2'd2;
//Other parameters
localparam	BIT_INDEX_MAX = 10;
localparam	BIT_TMR_MAX	=14'd10416;
localparam	START_BIT = 1'b0;
localparam	STOP_BIT	= 1'b1;

wire	baudTick	;//combinatorial logic that goes high when bitTmr has counted to the proper
value to ensure a 9600 baud rate
reg	[ 3:0] bitIndex	;//Contains the index of the next bit in txData that needs to be transferred
(txData is 10 bits so bitIndex ranges between 0 - 9)
reg	txBit	;//a register that holds the current data_in being sent over the UART TX line
reg	[ 9:0] txData	;//A register that contains the whole data_in packet to be sent, including start and
stop bits.
reg	[ 1:0] txState_next	;
reg	[ 1:0] txState_reg	;
reg	[13:0] bitTmr	;

always @(posedge clk,negedge arst_n)
	if(!arst_n)
		txState_reg <= RDY;
	else
		txState_reg <= txState_next;

//Next state logic

always @ (*)
	case (txState_reg)
```

```
                RDY       :         txState_next = send ? LOAD_BIT : RDY;
                LOAD_BIT   :    txState_next = SEND_BIT;
                SEND_BIT   : txState_next = baudTick ? ((bitIndex == BIT_INDEX_MAX) ? RDY :
LOAD_BIT) : SEND_BIT;
                default    : txState_next = RDY;
        endcase

//BaudRateGenerator to generate 9600 counter

        BaudRateGenerator #(
          .BAUD_RATE (BAUD_RATE)//
        )baudTick_gen (
                .clk    (clk    ),
                .arst_n  (!(txState_reg == RDY)),
                .baudTick(baudTick)
        );

always @(posedge clk)
        if (txState_reg == RDY)
                bitTmr <= 0;
        else if (baudTick)
                bitTmr <= 0;
        else
                bitTmr <= bitTmr + 1'b1;


//
always@(posedge clk)
        if (txState_reg == RDY)
                bitIndex <= 4'd0;
        else if (txState_reg == LOAD_BIT)
                bitIndex <= bitIndex + 1'b1;

//Latch txData
always@(posedge clk,negedge arst_n)
        if (!arst_n)
                txData <= 10'd0;
        else if (send)
                txData <= {STOP_BIT,data_in,START_BIT};

always@(posedge clk)
        if (txState_reg == RDY)
                txBit <= 1'b1;
        else if (txState_reg == LOAD_BIT)
```

```
                    txBit <= txData[bitIndex];

//Driving Output Signal
assign tx      = txBit;
assign ready   = (txState_reg == RDY);// && !send;


endmodule
```

UART Receiver

```
`timescale 1ns / 1ps
module UART_DRIVER_RX #(
        parameter BAUD_RATE = 9600        //1200/2400/4800/9600/19200/115200
)(
        //Input
        input         clk   ,
        input         arst_n ,
    input         rx    ,
        //Output
        output reg [7:0] data   ,
        output        valid
);

//States
localparam       IDLE = 2'd0;
localparam       START= 2'd1;
localparam       DATA = 2'd2;
localparam       STOP = 2'd3;
//Other parameters
localparam  BIT_INDEX_MAX = 10  ;
localparam  START_BIT     = 1'b0;
localparam  STOP_BIT      = 1'b1;


reg  [ 3:0] bitCntr        ;//Counter that keeps track of the number of clock cycles the current bit has been
held stable over the UART TX line.
wire       bitDone        ;//combinatorial logic that goes high when bitTmr has counted to the proper
value to ensure a 9600 baud rate
wire       start_bit_sample ;
reg  [ 3:0] bitIndex       ;//Contains the index of the next bit in txData that needs to be transferred
(txData is 10 bits so bitIndex ranges between 0 - 9)
reg  [ 1:0] rxState_next    ;
reg  [ 1:0] rxState_reg     ;
reg  [ 2:0] rx_reg         ;
wire       strt_bit_detected;



//BaudRateGenerator to generate 9600/16=600 counter
        BaudRateGenerator #(
```

```
        .BAUD_RATE (BAUD_RATE*16)//
    )baudTick_gen (
            .clk    (clk    ),
            .arst_n  (arst_n  ),
            .baudTick(baudTick)
    );



always@(posedge clk,negedge arst_n)
    if (!arst_n)
            bitCntr <= 4'd0;
    else if ((rxState_reg == START) && start_bit_sample)
            bitCntr <= 4'd0;
    else if (baudTick)
            bitCntr <= bitCntr + 1'b1;

assign bitDone        = (bitCntr == 15) && (rxState_reg == DATA ) && baudTick;
assign start_bit_sample = (bitCntr == 7 ) && (rxState_reg == START) && baudTick;



//reg the recieved asynchronous serial data
always@(posedge clk,negedge arst_n)
    if (!arst_n)
            rx_reg  <= 3'd0;
    else
            rx_reg  <= {rx_reg[1:0],rx};


//Detection of Start Bit
assign strt_bit_detected = (rxState_reg == IDLE) && rx_reg[2] && !rx_reg[1];



//Next state logic(mealy machine
always@(posedge clk,negedge arst_n)
    if (!arst_n)
            rxState_reg <= IDLE;
    else
      rxState_reg <= rxState_next;

always@(*)
    case (rxState_reg)
```

```
                IDLE    :       rxState_next = strt_bit_detected                         ? START : IDLE  ;
                START   :       rxState_next = (start_bit_sample && (rx_reg[2] == START_BIT)     ) ?
DATA  : START ;
                DATA    : rxState_next = (bitDone && (bitIndex == 7)                    ) ? STOP  : DATA  ;
                STOP            :       rxState_next = ((bitCntr == 15) && baudTick && (rx_reg[2] ==
STOP_BIT)) ? IDLE  : STOP  ;
                default : rxState_next = IDLE;
        endcase




//Data
always@(posedge clk,negedge arst_n)
        if (!arst_n)
                data <= 8'd0;
        else if((rxState_reg == DATA) && bitDone)
                data <= {rx_reg[2],data[7:1]};

assign valid = (bitCntr == 15) && (rxState_reg == STOP) && baudTick;

always@(posedge clk,negedge arst_n)
        if (!arst_n)
                bitIndex <= 3'd0;
        else if (rxState_reg == STOP)
                bitIndex <= 3'd0;
        else if( bitDone && (rxState_reg == DATA) )
                bitIndex <= bitIndex + 1'b1;

endmodule
```

ADC Interface Controller

```
**
 * This ADC interface provides an serial to parallel interface for
 * a ADC chip.
 * After reset the interface just loops collecting 1 reading from the
 * ADC chip every 16 sclk cycles and stores them into its internal memory
 * at one of 4 address spaces.  The data will be refreshed every 4x16 (64 cycles)
 * So a consumer should be setup to read the data before its gone.
 */
module adc_interface  (
  addr, data,
  din, dout,
  sclk, rst);

input [1:0]    addr;
output [11:0]  data;
input          sclk;
input          rst;
input          din;
output         dout;

reg [11:0]    data;
reg           dout;

reg [3:0]     sclk_count;
reg [1:0]     dout_addr; // We only want to select 4 of the 8 analog ports
wire          addr_inc;
reg  [11:0]   din_ff;
reg  [11:0]   data_ram [0:2];

assign addr_inc = (sclk_count == 4'b0001);

/* Handle clock counting */
always @ (posedge sclk or negedge rst)
  if (!rst)
    sclk_count <= 4'b0000;
  else
    sclk_count <= sclk_count + 1'b1;

/* Handle address incrementing to cycle through reading
   bytes from the ADC device input pins */
always @ (posedge addr_inc or negedge rst)
  if (!rst)
```

```
   dout_addr <= 2'b00;
  else
   dout_addr <= dout_addr + 1'b1;

/* Serial DOUT, based on sclk count, send the current address bit MSB first.
 * Note: since we are only selecting 4 Analog ports we just have 2 bits to send
 * during the 2nd clock cycle we went a zero by defualt.
 */
always @ (*)
  case (sclk_count)
    4'd3: dout = dout_addr[1];
    4'd4: dout = dout_addr[0];
    default: dout = 1'b0;
  endcase

/* DeSerialize DIN, use a shift register to move DIN into a 12 bit register during
 * clock cycles 4 -> 15
 */
always @ (posedge sclk or negedge rst)
  if (!rst)
     din_ff <= 12'd0;
  else
    casez (sclk_count)
     4'b01??, 4'b1???: din_ff <= {din_ff[10:0],din};
    endcase

/* Return static ram on read interface
 * Write shift register to static ram on first clock
 */
always @ (negedge sclk) begin
  if (sclk_count == 4'b0000) begin
    data_ram[dout_addr] <= din_ff;
  end
  data <= data_ram[addr];
end

endmodule



module clkdiv (
  input clk_in,
  output clk_out,
```

```
 input rst
);

/* div by 16 * 4 = 64
   div by = 2 ^ (bits - 1)
*/
  // 1 bit - no delay
  // 2 bit - div by 2
  // 3 bit - div by 4
  // 4 bit - div by 8
  // 5 bit - div by 16
  // 6 bit - div by 32
  // 7 bit - div by 64

reg [6:0] div_counter;

assign clk_out = div_counter[6];

always @ (posedge clk_in or negedge rst)
 if (!rst)
   div_counter <= 7'b0;
 else
   div_counter <= div_counter + 1'b1;

endmodule


/**
 * The LED driver takes the 11:0 input as a magnitude and graphs
 * the value on an LED (like a recording mixer level display).
 * the magnitude is update 85 times per second
 * it will output 16 PWM phases per LED
 * each PWM cycle wave will be 0 to 50% duty cycle depending on the
 * input.
 *
 * Because the input wave is centered at half the 12 bit dinput
 * we only need to consider the top 6 bits (assuming the bottom
 * is a close mirror).  The 6 bits are dithered to 8 bits by the
 * following table
 *
 */

module led_driver(leds, rst, dclk, dinput);
```

```
output [7:0] leds;
input       rst;
input       dclk;   // 44100 Hz
input  [11:0] dinput; // 85Hz (sync to 44100Hz)

reg    [7:0] leds;

//wire   [7:0] din_high;

//assign din_high = dinput[10:3];

always @ (posedge dclk or negedge rst)
begin
 if (!rst)
   leds <= 8'b0000_0000;
 else
   leds <= dinput [11:4];
   /* This is base 2 logarithm of the input
   casez (din_high)
     8'b0000_0000: leds <= 8'b0000_0000;
     8'b0000_0001: leds <= 8'b0000_0001;
     8'b0000_001?: leds <= 8'b0000_0011;
     8'b0000_01??: leds <= 8'b0000_0111;
     8'b0000_1???: leds <= 8'b0000_1111;
     8'b0001_????: leds <= 8'b0001_1111;
     8'b001?_????: leds <= 8'b0011_1111;
     8'b01??_????: leds <= 8'b0111_1111;
     8'b1???_????: leds <= 8'b1111_1111;
   endcase */
end

endmodule


/**
 * The max module scans the input every for a for a value and outputs
 * a new value every n clock cycles. The max value is latched to the max
 * output when the cycle is up.
 */
module max (din, dclk, rst, maxout);

parameter BUS_WIDTH = 12;

output [BUS_WIDTH-1:0]  maxout;
```

```
input  [BUS_WIDTH-1:0]  din;
input                   dclk;
input                   rst;

reg [BUS_WIDTH-1:0]     max_current;
reg [BUS_WIDTH-1:0]     maxout;
reg [8:0]                sample_count; // We take the max of 512 samples
                        // 44100 hz / 512 -> 85 max samples per second

wire                    sample_done;
wire [BUS_WIDTH-1:0]    max_next;

assign sample_done = (sample_count == 9'b0);
/* Next max is either current input or current max */
/* if we are done sampling then we throw out current max */
assign max_next = ((din > max_current) || sample_done)  ? din : max_current;

/* Handle sample counting */
always @ (posedge dclk or negedge rst)
 if (!rst)
   sample_count <= 9'b0;
 else
   sample_count <= sample_count + 1'b1;

/* Handle max capturing */
always @ (posedge dclk or negedge rst)
 if (!rst)
   max_current <= {BUS_WIDTH{1'b0}};
 else
   max_current <= max_next;

/* Handle max output when sampling is done */
always @ (posedge sample_done or negedge rst) begin
 if (!rst)
  maxout <= {BUS_WIDTH{1'b0}};
 else
  maxout <= max_current;
end

endmodule



/*
```

```
 * Rectifier circuit for the ADC output bits.
 * Since the input wave ranges from 0 to 4096
 * and is centered around 2048, we need to measure
 * how much the signal moves from the center.
 *
 * This circuit creates a half rectifier.
 */
parameter BUS_WIDTH = 12;

module rectifier(
  input  [BUS_WIDTH-1:0] din,
  output reg [BUS_WIDTH-1:0] dout
);

/* if high bit is set allow output otherwise nothing */
always @ (*)
if (din[BUS_WIDTH-1])
 dout = din;
else
 dout = {BUS_WIDTH{1'b0}};

endmodule


/*
 * Top level module that interfaces with and ADC
 * and outputs the result on 7 inline LEDs like a VU
 * meter.
 */
module signal_display (
  input       adc_sdat,  // \
  output      adc_saddr, // | - ADC interface
  output      adc_sclk,  // |
  output      adc_cs_n,  ///
  input       rst_n,
  input       clk,      // 44,000hz x 16 x 4
  output [7:0] leds
);

wire       dclk;        // divided clock (44,000hz)
wire       rst;
wire [1:0]  adc_addr;
wire [11:0] adc_data;
wire [11:0] rectified_data;
```

```
wire [11:0] max;

assign adc_sclk = clk;
assign adc_addr = 2'b00;   // always select ADC input 0
//assign rst = ~rst_n;      // rst_n is usually high, reset when low
assign rst = rst_n;      // rst_n is usually high, reset when low
//assign adc_cs_n = rst;    // rst is usually low, select adc then

assign adc_cs_n = 1'b0;    // rst is usually low, select adc then

// Interface with ADC
//   - addr and data are parrallel interface
//   - din/dout are serial interface with ADC chip
adc_interface adc_interfacei (
  .addr(adc_addr), // hard coded to 00 for now
  .data(adc_data),
  .sclk(clk),     // signal clock
  .rst(rst),
  .din(adc_sdat),
  .dout(adc_saddr));


// Clock divider by 64 (bring clk down to 44Khz)
clkdiv clkdivi (
 .clk_in(clk),
 .clk_out(dclk),
 .rst(rst));
/*
// Just output values that are above 2048 (midpoint)
rectifier rectifieri (
  .din(adc_data),
  .dout(rectified_data));

// Max get the max adc value every 85hz
max maxi (
  .din(rectified_data),
  .dclk(dclk),
  .rst(rst),
  .maxout(max));
*/

// Output the max value onto 7 inline leds
led_driver led_driveri (
  .leds(leds),
```

```
  .rst(rst),
  .dclk(dclk),
  //.dinput(max));
  .dinput(adc_data[11:0]));

endmodule
```

SRAM Controller

```
//`timescale <time_units> / <precision>
module SRAM(
input clk, arst_n,

input send, rw,
input [15:0] data_in,
input [20:0] address_in,
output rdy,

output valid,
output reg [7:0] data_out,
input sync,

output [20:0] address_out,
inout [15:0] data_sram,
output reg OE,              // output enable - low to enable
output reg WE,              // write enable - low to enable
output reg UB,              // upper bit active select - low to enable
output reg LB,              // lower bit active select - low to enable
output reg CS               // chip select - low to enable

);

reg [15:0] data_received;

reg [2:0] State, Nxt_State;
localparam IDLE = 3'd0, Start_Write_Setup = 3'd1, Start_Write = 3'd2, Start_Read_Setup = 3'd3,
Start_Read = 3'd4, Valid_Pulse = 3'd5;

assign valid = (State == Valid_Pulse);
//assign data_sram = (State == Start_Write_Setup)? data_in: 21'bzzzzzzzzzzzzzzzzzzzzz ;
assign data_sram = (State == Start_Write)? data_in: 21'bzzzzzzzzzzzzzzzzzzzzz ;
//assign data_received = (State == Start_Read)? data_sram: 21'b000000000000000000000 ;
//assign data_received = data_sram;

assign rdy = (State == IDLE);
assign address_out = address_in;
assign data_out = data_received[15:8];

always @(*)
case(State)
IDLE : begin
```

```
    if(send && rw) Nxt_State = Start_Write_Setup;
    else if (send && !rw) Nxt_State = Start_Read_Setup;
    else Nxt_State = IDLE;
    OE = 1'b1;
    WE = 1'b1;
    CS = 1'b1;
    UB = 1'b1;
    LB = 1'b1;
    end

Start_Write_Setup : begin
    Nxt_State = Start_Write;
    OE = 1'b0;
    WE = 1'b1;
    CS = 1'b0;
    UB = 1'b1;
    LB = 1'b1;
    end

Start_Write : begin
Nxt_State = Valid_Pulse;
    OE = 1'b0;
    WE = 1'b0;
    CS = 1'b0;
    UB = 1'b0;
    LB = 1'b0;
    end

Start_Read_Setup : begin
Nxt_State = Start_Read;
   OE = 1'b0;
   WE = 1'b1;
   CS = 1'b0;
   UB = 1'b0;
   LB = 1'b0;
   end

Start_Read : begin
   Nxt_State =  Valid_Pulse;
   OE = 1'b0;
   WE = 1'b1;
   CS = 1'b0;
   UB = 1'b0;
   LB = 1'b0;
```

```
    data_received = data_sram;
    end

Valid_Pulse : begin
    if(sync) begin
    Nxt_State = IDLE;
    end
    OE = 1'b1;
    WE = 1'b1;
    CS = 1'b1;
    UB = 1'b0;
    LB = 1'b0;
    end

endcase

always @(posedge clk, negedge arst_n)
if (!arst_n)
State <= IDLE;
else
State <= Nxt_State;

Endmodule
```

FRAM

```
//`timescale <time_units> / <precision>
module FRAM(
input clk, arst_n,

input send, rw,
input [7:0] data_in,
input [23:0] address_in,
output rdy,

output valid,
output reg [7:0] data_out,
input sync,

output reg cs,
output sclk,
output reg dout,
input din,
output WP
);

reg [3:0] State, Nxt_State;
localparam IDLE = 4'd0, WREN_CODE = 4'd1, WEN_HIGH = 4'd2, WRITE_CODE = 4'd3,
ADDRESS_WRITE_HSB = 4'd4,
ADDRESS_WRITE_MSB = 4'd5, ADDRESS_WRITE_LSB = 4'd6, DATA_WRITE = 4'd7, VALID = 4'd8,
READ_CODE = 4'd9, DATA_OUT = 4'd10;

reg [7:0] WRITE_OPCODE = 8'b01000000;
reg [7:0] WREN_OPCODE  = 8'b01100000;
reg [7:0] READ_OPCODE  = 8'b11000000;

reg [2:0] sclk_count;

assign valid = (State == VALID);
assign rdy = (State == IDLE);
assign WP = 1'b1;
assign sclk = clk;
reg flag;

always @(posedge sclk, negedge arst_n)
if (!arst_n)
State <= IDLE;
else
```

```
State <= Nxt_State;

always @(posedge sclk, negedge arst_n)
if (( State != Nxt_State ) || (!arst_n))
 sclk_count <= 3'd0;
else
 sclk_count <= sclk_count + 1'b1;

always @(*)
begin
case(State)

IDLE :
dout = 1'b0;

WREN_CODE :
case(sclk_count)
3'd0: dout = WREN_OPCODE[0];
3'd1: dout = WREN_OPCODE[1];
3'd2: dout = WREN_OPCODE[2];
3'd3: dout = WREN_OPCODE[3];
3'd4: dout = WREN_OPCODE[4];
3'd5: dout = WREN_OPCODE[5];
3'd6: dout = WREN_OPCODE[6];
3'd7: dout = WREN_OPCODE[7];
default : dout = 1'b0;
endcase


WRITE_CODE :
case(sclk_count)
3'd0: dout = WRITE_OPCODE[0];
3'd1: dout = WRITE_OPCODE[1];
3'd2: dout = WRITE_OPCODE[2];
3'd3: dout = WRITE_OPCODE[3];
3'd4: dout = WRITE_OPCODE[4];
3'd5: dout = WRITE_OPCODE[5];
3'd6: dout = WRITE_OPCODE[6];
3'd7: dout = WRITE_OPCODE[7];
default : dout = 1'b0;
endcase

READ_CODE :
case(sclk_count)
```

```
3'd0: dout = READ_OPCODE[0];
3'd1: dout = READ_OPCODE[1];
3'd2: dout = READ_OPCODE[2];
3'd3: dout = READ_OPCODE[3];
3'd4: dout = READ_OPCODE[4];
3'd5: dout = READ_OPCODE[5];
3'd6: dout = READ_OPCODE[6];
3'd7: dout = READ_OPCODE[7];
default : dout = 1'b0;
endcase

ADDRESS_WRITE_HSB :
case(sclk_count)
3'd0: dout = address_in[23];
3'd1: dout = address_in[22];
3'd2: dout = address_in[21];
3'd3: dout = address_in[20];
3'd4: dout = address_in[19];
3'd5: dout = address_in[18];
3'd6: dout = address_in[17];
3'd7: dout = address_in[16];
default : dout = 1'b0;
endcase

ADDRESS_WRITE_MSB :
case(sclk_count)
3'd0: dout = address_in[15];
3'd1: dout = address_in[14];
3'd2: dout = address_in[13];
3'd3: dout = address_in[12];
3'd4: dout = address_in[11];
3'd5: dout = address_in[10];
3'd6: dout = address_in[9];
3'd7: dout = address_in[8];
default : dout = 1'b0;
endcase

ADDRESS_WRITE_LSB :
case(sclk_count)
3'd0: dout = address_in[7];
3'd1: dout = address_in[6];
3'd2: dout = address_in[5];
3'd3: dout = address_in[4];
3'd4: dout = address_in[3];
```

```
3'd5: dout = address_in[2];
3'd6: dout = address_in[1];
3'd7: dout = address_in[0];
default : dout = 1'b0;
endcase

DATA_WRITE :
case(sclk_count)
3'd0: dout = data_in[7];
3'd1: dout = data_in[6];
3'd2: dout = data_in[5];
3'd3: dout = data_in[4];
3'd4: dout = data_in[3];
3'd5: dout = data_in[2];
3'd6: dout = data_in[1];
3'd7: dout = data_in[0];
default : dout = 1'b0;
endcase
/*
DATA_OUT :
   casez (sclk_count_neg)
     3'b???: data_out <= {data_out[6:0],din};
   endcase
 */
default:
begin
dout = 1'b0;
end

endcase
end


always @(posedge sclk, negedge arst_n)
if (( State == READ_CODE ) || (!arst_n))
 data_out <= 8'd0;
else if(State == DATA_OUT)
   begin
   data_out <= {data_out[6:0],din};
   end
else
  data_out <= data_out;

always @(*)
```

```
begin
case(State)
IDLE : begin
    if(send)
    begin
      if(rw)
      begin
      Nxt_State = WREN_CODE;
      flag = 1'b1;
      cs = 1'b0;
      end
      else
      begin
      Nxt_State = READ_CODE;
      flag = 1'b0;
      cs = 1'b0;
      end
    end
    else begin
    Nxt_State = IDLE;
    flag = 1'b0;
    end

    cs = 1'b1;
    end

WREN_CODE : begin
    if (sclk_count == 3'd7)
    Nxt_State = WEN_HIGH;
    else
    Nxt_State = WREN_CODE;
    cs = 1'b0;
    end

WEN_HIGH : begin
    if(sclk_count == 3'd1) //WEN_HIGH pulled high for 2 cycles
    begin
    Nxt_State = WRITE_CODE;
    cs = 1'b1;
    end
    else
    Nxt_State = WEN_HIGH;
    cs = 1'b1;
    end
```

```
WRITE_CODE : begin
    if(sclk_count == 3'd7)
    Nxt_State = ADDRESS_WRITE_HSB;
    else
    Nxt_State = WRITE_CODE;
    cs = 1'b0;
    end

ADDRESS_WRITE_HSB : begin
    if(sclk_count == 3'd7)
    Nxt_State = ADDRESS_WRITE_MSB;
    else
    Nxt_State = ADDRESS_WRITE_HSB ;
    cs = 1'b0;
    end

ADDRESS_WRITE_MSB : begin
    if(sclk_count == 3'd7)
    Nxt_State = ADDRESS_WRITE_LSB;
    else
    Nxt_State = ADDRESS_WRITE_MSB;
    cs = 1'b0;
    end

ADDRESS_WRITE_LSB : begin
    if(sclk_count == 3'd7)
    begin
    if(flag== 1'b1)
    Nxt_State = DATA_WRITE;
    else if(flag== 1'b0)
    Nxt_State = DATA_OUT;
    end
    else
    Nxt_State = ADDRESS_WRITE_LSB;
    cs = 1'b0;
    end

DATA_WRITE : begin
    if(sclk_count == 3'd7)
    Nxt_State = VALID;
    else
    Nxt_State = DATA_WRITE;
    cs = 1'b0;
```

```
    end

VALID : begin
    if(sync)
    Nxt_State = IDLE;
    else
    Nxt_State = VALID;
    cs = 1'b1;
    end

READ_CODE : begin
    if (sclk_count == 3'd7)
    Nxt_State = ADDRESS_WRITE_HSB;
    else
    Nxt_State = READ_CODE;
    cs = 1'b0;
    end

DATA_OUT : begin
    if(sclk_count == 3'd7)
    Nxt_State = VALID;
    else
    Nxt_State = DATA_OUT;
    cs = 1'b0;
    end

default : begin
   Nxt_State = IDLE;
   cs = 1'b1;
end

endcase
end


Endmodule
```

LPDDR3

```
//`timescale <time_units> / <precision>

module LPDDR_Driver(
input clk,
input arst_n,

input send,
input rw,
output rdy,

output [7:0] data_out,
output valid,
input sync,

output reg SYS_RESET_N,
output [37:0] L_ADDR,
output [10:0] L_B_SIZE,
output reg L_R_REQ,
output reg L_W_REQ,
input L_BUSY,
input CTRLR_READY,
input L_D_REQ,
input L_R_VALID,
output [255:0] L_DATA_IN,
input [255:0] L_DATAOUT

);
// If this doesnt work try integrating the reset signal in design aswell

reg [3:0] STATE, NXT_STATE;
reg [255:0] L_DATAOUT_READ;

localparam IDLE = 4'd0, WRITE = 4'd1, READ = 4'd2, SIGNAL_ASSIGN_W = 4'd3,
SIGNAL_ASSIGN_R = 4'd4, DATA_W_RDY = 4'd5, DATA_R_VALID = 4'd6, VALID = 4'd7, WRITE_RST = 4'd8,
READ_RST = 4'd9;

assign  L_B_SIZE = 11'd8;
assign L_ADDR = 38'd0;
assign L_DATA_IN = 256'd0;
assign data_out = L_DATAOUT_READ[7:0];
assign valid = (STATE == VALID);
assign rdy = (STATE == IDLE);
```

```
always @(posedge clk, negedge arst_n)
if(!arst_n)
STATE <= IDLE;
else
STATE <= NXT_STATE;

always @(*)
case(STATE)

IDLE : begin
if(send && !rw) begin
NXT_STATE <= WRITE_RST;
SYS_RESET_N <= 1'b0;
end
else if (send && rw) begin
NXT_STATE <= READ_RST;
SYS_RESET_N <= 1'b0;
end
else begin
NXT_STATE <= IDLE;
SYS_RESET_N <= 1'b1;
end
end

WRITE_RST : begin
if(CTRLR_READY)
NXT_STATE <= WRITE;
else
NXT_STATE <= WRITE_RST;
SYS_RESET_N <= 1'b1;
end

READ_RST : begin
if(CTRLR_READY)
NXT_STATE <= READ;
else
NXT_STATE <= READ_RST;
SYS_RESET_N <= 1'b1;
end

WRITE : begin
if(!L_BUSY)
NXT_STATE <= SIGNAL_ASSIGN_W;
```

```
else
NXT_STATE <= WRITE;
end

READ : begin
if(!L_BUSY)
NXT_STATE <= SIGNAL_ASSIGN_R;
else
NXT_STATE <= READ;
end

SIGNAL_ASSIGN_W : begin
if(L_BUSY) begin
L_W_REQ <= 1'b0;
NXT_STATE <= DATA_W_RDY;
end
else begin
NXT_STATE <= SIGNAL_ASSIGN_W;
L_W_REQ <= 1'b1;
end
end

DATA_W_RDY : begin
if(L_D_REQ)
NXT_STATE <= VALID;
else
NXT_STATE <= DATA_W_RDY;
end

SIGNAL_ASSIGN_R : begin
if(L_BUSY) begin
NXT_STATE <= DATA_R_VALID;
L_R_REQ <= 1'b0;
end
else begin
NXT_STATE <= SIGNAL_ASSIGN_R;
L_R_REQ <= 1'b1;
end
end

DATA_R_VALID : begin
if(L_R_VALID) begin
NXT_STATE <= VALID;
L_DATAOUT_READ <= L_DATAOUT;
```

```
end
else
NXT_STATE <= DATA_R_VALID;
end

VALID : begin
if(sync)
NXT_STATE <= IDLE;
else
NXT_STATE <= VALID;
end

default : NXT_STATE <= IDLE;
endcase

endmodule
```

QDR II+

```
//`timescale <time_units> / <precision>

module QDR_Controller(
input clk, arst_n,

input send, rw,
input data_in,
output rdy,
input address_in,

output RESET_N,
input TRAINING_COMPLETE,
input [2:0] TRAINING_ERROR,
output [287:0] RDATA,
output [39:0] RADDR,
output [1:0] READ_N,
input RVALID,

output [287:0] WDATA,
output [39:0] WADDR,
output [31:0] WSTRB_N,
output [1:0] WRITE_N,

output [7:0] DATA_OUT,
output VALID_OUT,
input SYNC
);

reg [3:0] STATE, NXT_STATE;
reg [1:0] sclk_count;
wire [287:0] RDATA_IN;

localparam IDLE = 4'd0, T_START_R = 4'd1, T_START_W = 4'd2,
T_FAIL = 4'd3, SIGNAL_READ = 4'd4, SIGNAL_WRITE = 4'd5, WAIT = 4'd6, IDLE = 4'd7;

assign READ_N = 2'b11;
assign WRITE_N = 2'b11;
assign WSTRB_N = (STATE == SIGNAL_WRITE) ? 32'b1111_1111_1111_1111_1111_1111_1111_1111:
32b'0000_0000_0000_0000_0000_0000_00000_0000;
assign WADDR = 40'b111111111_1111111111_1111111111_1111111111;
assign RADDR = 40'b111111111_1111111111_1111111111_1111111111;
```

```
assign WDATA =
288'b1111111111111111_1111111111111111_1111111111111111_1111111111111111_1111111111
11111_1111111111111111_1111111111111111_1111111111111111_1111111111111111_1111111111
111111_1111111111111111_1111111111111111_1111111111111111_1111111111111111_1111111111
1111111_1111111111111111_1111111111111111_1111111111111111;
assign VALID_OUT = (STATE == VALID)
assign DATA_OUT = RDATA_IN[7:0]

always @(posedge clk, negedge arst_n)
if (( STATE != NXT_STATE ) || (!arst_n))
 sclk_count <= 2'd0;
else
 sclk_count <= sclk_count + 1'b1;



always @(posedge clk, negedge arst_n)
if(!arst_n)
NXT_STATE <= IDLE;
else
STATE <= NXT_STATE;


always @(*)
case (STATE)

IDLE : begin
if(send && rw) NXT_STATE <= T_START_R;
else if (send && !rw) NXT_STATE <= T_START_W;
else NXT_STATE <= IDLE;
end

T_START_R : begin
if(TRAINING_COMPLETE && (TRAINING_ERROR[2:1] == 2'd0))
NXT_STATE <= SIGNAL_READ;
else if(TRAINING_COMPLETE && (TRAINING_ERROR[2:1] != 2'd0 ))
NXT_STATE <= T_FAIL;
else
NXT_STATE <= T_START_R;
end

T_START_W : begin
if(TRAINING_COMPLETE && !TRAINING_ERROR)
NXT_STATE <= SIGNAL_WRITE;
```

```
else if(TRAINING_COMPLETE && TRAINING_ERROR)
NXT_STATE <= T_FAIL;
else
NXT_STATE <= T_START_W;
end

T_FAIL : begin
NXT_STATE <= IDLE;
end

SIGNAL_READ : begin
if(RVALID)
NXT_STATE <= WAIT;
RDATA_IN <= RDATA;
else
NXT_STATE <= SIGNAL_READ;
end

SIGNAL_WRITE : begin
NXT_STATE <= WAIT;
end

WAIT : begin
if(sclk_count == 2'd3)
NXT_STATE <= VALID;
else
NXT_STATE <= WAIT;
end

VALID : begin
if(SYNC)
NXT_STATE <= IDLE;
else
NXT_STATE <= VALID;
end

endcase

endmodule
```

I2C Master Controller

```
//`timescale <time_units> / <precision>

module I2C_Master(
input clk, arst_n,

input send,
input [7:0] reg_addr, slave_addr,
output rdy,

output valid,
output reg [7:0] data_out,
input sync,

inout SDA,
inout SCL
);



parameter RDY = 4'd0, START = 4'd1, ADDR = 4'd2, ACK_ADD = 4'd3, POINT_ADDR = 4'd4,
ACK_POINT_ADDR = 4'd5, DATA_HIGH = 4'd6, ACK_DATA_HIGH = 4'd7, DATA_LOW = 4'd8,
ACK_DATA_LOW = 4'd9, STOP = 4'd10;



reg [2:0] state, nxt_state;

reg [7:0] reg_addr_store, slave_addr_store;
reg [7:0] data_high, data_low;
assign data_out = data_low;

reg [3:0] ref_clk_reg;
wire ref_clk;

reg SDA_out;
wire SDA_in;
assign SDA = SDA_out;
assign SDA_in = SDA;


always @(posedge ref_clk, negedge arst_n)
if (( state == ACK_POINT_ADDR ) || (!arst_n) || (state == ACK_DATA_HIGH) )
```

```
 begin
 data_out <= 8'd0;
 data_high <= 8'd0;
 data_low <= 8'd0;
 end
 else if(state == DATA_HIGH)
   begin
   data_high <= {data_high[6:0],SDA_in};
   end
else if(state == DATA_LOW)
   begin
   data_low <= {data_low[6:0],SDA_in};
   end
else
  begin
  data_high <= data_high;
  data_low <= data_low;
  data_out <= data_out;
  end



// counter resets when sttae is not equal to the next state
always @(posedge clk)
 ref_clk_reg <= ref_clk_reg + 1'b1;

assign ref_clk = ref_clk_reg[3];

reg [3:0] bit_count;

always @(posedge ref_clk, negedge arst_n)
if (( state != nxt_state ) || (!arst_n))
 bit_count <= 3'd0;
else
 bit_count <= bit_count + 1'b1;


always @(*)
case(state)

RDY : begin
SDA_out = 1'bz;
end
```

```
START : begin
SDA_out =  1'b1;
end

ADDR : begin
case(bit_count)
3'd0 : SDA_out = slave_addr_store[0];
3'd1 : SDA_out = slave_addr_store[1];
3'd2 : SDA_out = slave_addr_store[2];
3'd3 : SDA_out = slave_addr_store[3];
3'd4 : SDA_out = slave_addr_store[4];
3'd5 : SDA_out = slave_addr_store[5];
3'd6 : SDA_out = slave_addr_store[6];
3'd7 : SDA_out = slave_addr_store[7];
default : SDA_out = 1'bz;
endcase
end

ACK_ADD : begin
SDA_out = 1'bz;
end


POINT_ADDR : begin
case(bit_count)
3'd0 : SDA_out = reg_addr_store[0];
3'd1 : SDA_out = reg_addr_store[1];
3'd2 : SDA_out = reg_addr_store[2];
3'd3 : SDA_out = reg_addr_store[3];
3'd4 : SDA_out = reg_addr_store[4];
3'd5 : SDA_out = reg_addr_store[5];
3'd6 : SDA_out = reg_addr_store[6];
3'd7 : SDA_out = reg_addr_store[7];
default : SDA_out = 1'bz;
endcase
end

ACK_POINT_ADDR : begin
SDA_out = 1'bz;
end


DATA_HIGH : begin
```

```
SDA_out = 1'bz;
end

ACK_DATA_HIGH : begin
SDA_out = 1'bz;
end


DATA_LOW : begin
SDA_out = 1'bz;
end


ACK_DATA_LOW : begin
SDA_out = 1'bz;
end

default: begin
SDA_out = 1'bz;
end
endcase;



always @(*)
begin
case(state)

RDY : begin

 if(send)
 nxt_state <= START;
 else
 nxt_state <= RDY;

end

START : begin
  nxt_state <= ADDR;
  reg_addr_store <= reg_addr;
  slave_addr_store <= slave_addr;
end
```

```
ADDR : begin
if(bit_count == 3'd7)
nxt_state <= ACK_ADD;
else
nxt_state <= ADDR;
end

ACK_ADD : begin
if(SDA_in)
nxt_state <= POINT_ADDR;
else
nxt_state <= RDY;
end

POINT_ADDR : begin
if(bit_count == 3'd7)
nxt_state <= ACK_POINT_ADDR;
else
nxt_state <= POINT_ADDR;
end


ACK_POINT_ADDR: begin
if(SDA_in)
nxt_state <= DATA_HIGH;
else
nxt_state <= RDY;
end

DATA_HIGH: begin
if(bit_count == 3'd7)
nxt_state <= ACK_DATA_HIGH;
else
nxt_state <= DATA_HIGH;
end

ACK_DATA_HIGH: begin
nxt_state <= DATA_LOW;
end

DATA_LOW: begin
if(bit_count == 3'd7)
nxt_state <= ACK_DATA_LOW;
else
```

```
nxt_state <= DATA_LOW;
end

ACK_DATA_LOW: begin
nxt_state <= STOP;
end

STOP: begin
nxt_state <= RDY;
end

endcase

end;


always @(posedge ref_clk,negedge arst_n)
if(!arst_n)
 state <= RDY;
else
 state <= nxt_state;

endmodule
```

Linker Script

```
/*******************************************************************************
 * Copyright 2019-2021 Microchip FPGA Embedded Systems Solutions.
 *
 * SPDX-License-Identifier: MIT
 *
 * file name : miv-rv32-ram.ld
 * Mi-V soft processor linker script for creating a SoftConsole downloadable
 * debug image executing in SRAM.
 *
 * This linker script assumes that a RAM is connected at on the Mi-V soft
 * processor memory space. The start address and size of the memory space must
 * be correct as per the Libero design.
 *
 * Supports MIV_RV32 as well as the legacy RV32 cores with appropriate memory
 * section addresses as per your design.
 *
 * SVN $Revision: 13158 $
 * SVN $Date: 2021-01-31 10:57:57 +0530 (Sun, 31 Jan 2021) $
 */

OUTPUT_ARCH( "riscv" )
ENTRY(_start)

MEMORY
{
   ram (rwx) : ORIGIN = 0x80000000, LENGTH = 64k
}

RAM_START_ADDRESS   = 0x80000000;      /* Must be the same value MEMORY region ram ORIGIN
above. */
MTVEC_OFFSET       = 0x100;
RAM_SIZE           = 64k;           /* Must be the same value MEMORY region ram LENGTH above. */
STACK_SIZE         = 2k;            /* needs to be calculated for your application */
HEAP_SIZE          = 2k;           /* needs to be calculated for your application */

SECTIONS
{
  .entry : ALIGN(0x10)
  {
    KEEP (*(SORT_NONE(.entry)))
```

```
 . = ALIGN(0x10);
} > ram

.text : ALIGN(0x10)
{
 *(.text .text.* .gnu.linkonce.t.*)
 *(.plt)
 . = ALIGN(0x10);

 KEEP (*crtbegin.o(.ctors))
 KEEP (*(EXCLUDE_FILE (*crtend.o) .ctors))
 KEEP (*(SORT(.ctors.*)))
 KEEP (*crtend.o(.ctors))
 KEEP (*crtbegin.o(.dtors))
 KEEP (*(EXCLUDE_FILE (*crtend.o) .dtors))
 KEEP (*(SORT(.dtors.*)))
 KEEP (*crtend.o(.dtors))

 *(.rodata .rodata.* .gnu.linkonce.r.*)
 *(.gcc_except_table)
 *(.eh_frame_hdr)
 *(.eh_frame)

 KEEP (*(.init))
 KEEP (*(.fini))

 PROVIDE_HIDDEN (__preinit_array_start = .);
 KEEP (*(.preinit_array))
 PROVIDE_HIDDEN (__preinit_array_end = .);
 PROVIDE_HIDDEN (__init_array_start = .);
 KEEP (*(SORT(.init_array.*)))
 KEEP (*(.init_array))
 PROVIDE_HIDDEN (__init_array_end = .);
 PROVIDE_HIDDEN (__fini_array_start = .);
 KEEP (*(.fini_array))
 KEEP (*(SORT(.fini_array.*)))
 PROVIDE_HIDDEN (__fini_array_end = .);
 . = ALIGN(0x10);

} > ram

/* short/global data section */
.sdata : ALIGN(0x10)
{
```

```
  __sdata_load = LOADADDR(.sdata);
  __sdata_start = .;
  PROVIDE( __global_pointer$ = . + 0x800);
  *(.srodata.cst16) *(.srodata.cst8) *(.srodata.cst4) *(.srodata.cst2)
  *(.srodata*)
  *(.sdata .sdata.* .gnu.linkonce.s.*)
  . = ALIGN(0x10);
  __sdata_end = .;
} > ram

/* data section */
.data : ALIGN(0x10)
{
  __data_load = LOADADDR(.data);
  __data_start = .;
  *(.got.plt) *(.got)
  *(.shdata)
  *(.data .data.* .gnu.linkonce.d.*)
  . = ALIGN(0x10);
  __data_end = .;
} > ram

/* sbss section */
.sbss : ALIGN(0x10)
{
  __sbss_start = .;
  *(.sbss .sbss.* .gnu.linkonce.sb.*)
  *(.scommon)
  . = ALIGN(0x10);
  __sbss_end = .;
} > ram

/* sbss section */
.bss : ALIGN(0x10)
{
  __bss_start = .;
  *(.shbss)
  *(.bss .bss.* .gnu.linkonce.b.*)
  *(COMMON)
  . = ALIGN(0x10);
  __bss_end = .;
} > ram

/* End of uninitialized data segment */
```

```
  _end = .;

  .heap : ALIGN(0x10)
  {
    __heap_start = .;
    . += HEAP_SIZE;
    __heap_end = .;
    . = ALIGN(0x10);
    _heap_end = __heap_end;
  } > ram

  .stack : ALIGN(0x10)
  {
    __stack_bottom = .;
    . += STACK_SIZE;
    __stack_top = .;
  } > ram
}
```

Main RISC V Program (Drivers and HAL not included)

```
/*******************************************************************************
 * (c) Copyright 2016-2017 Microsemi SoC Products Group. All rights reserved.
 *
 * This simple example demonstrates how to use the CoreUARTapb driver and
 * CoreGPIO driver. This example application prints Hello World! on Serial
 * Terminal program and blinks LEDs on PolarFire Evaluation Kit.
 *
 */
#include "miv_rv32_hal.h"
#include "core_uart_apb.h"
#include "core_gpio.h"
#include "hw_platform.h"
#include "hw_reg_access.h"

#if 0

#define BUFFER_A_SIZE   512

/* Manufacture and device IDs for Micron MT25QL01GBBB SPI Flash. */
#define FLASH_MANUFACTURER_ID   (uint8_t)0x20
#define FLASH_DEVICE_ID         (uint8_t)0xBB

/*
 * Static global variables
 */
static uint8_t g_flash_wr_buf[BUFFER_A_SIZE];
static uint8_t g_flash_rd_buf[BUFFER_A_SIZE];

/* Local Function. */
static uint8_t verify_write(uint8_t* write_buff, uint8_t* read_buff, uint16_t size);

/*******************************************************************************//**
 * Read the date from SPI FLASH and compare the same with write buffer.
 */
static uint8_t verify_write(uint8_t* write_buff, uint8_t* read_buff, uint16_t size)
{
   uint8_t error = 0;
   uint16_t index = 0;

   while(size != 0)
```

```
    {
        if(write_buff[index] != read_buff[index])
        {
            error = 1;
            break;
        }
        index++;
        size--;
    }

    return error;
}

#endif


/*
 * Delay loop down counter load value.
 */
#define DELAY_LOAD_VALUE     0x00080000


/*******************************************************************************
 * Instruction message. This message will be transmitted over the UART to
 * HyperTerminal when the program starts.
 *******************************************************************************/

uint8_t testmsg[] = {"\r\n\r\nHello World!\0"};

/*-------------------------------------------------------------------------
 * UART instance data.
 */
UART_instance_t g_uart;

/*-------------------------------------------------------------------------
 * GPIO instance data.
 */
gpio_instance_t g_gpio_out;


/*-------------------------------------------------------------------------
 * main
 */
int main()
```

```
{
  volatile int32_t delay_count = 0;

#if 1
  /*******************************************************************************
   * Initialize the CoreGPIO driver with the base address of the CoreGPIO
   * instance to use and the initial state of the outputs.
   ******************************************************************************/
  GPIO_init(&g_gpio_out, COREGPIO_OUT_BASE_ADDR, GPIO_APB_32_BITS_BUS);

  /*******************************************************************************
   * Configure the GPIOs.
   ******************************************************************************/
//  GPIO_config( &g_gpio_out, GPIO_0, GPIO_OUTPUT_MODE );
//  GPIO_config( &g_gpio_out, GPIO_1, GPIO_OUTPUT_MODE );
//  GPIO_config( &g_gpio_out, GPIO_2, GPIO_OUTPUT_MODE );
//  GPIO_config( &g_gpio_out, GPIO_3, GPIO_OUTPUT_MODE );

  /*******************************************************************************
   * Set the GPIO outputs.
   ******************************************************************************/
  GPIO_set_output( &g_gpio_out, GPIO_0,1 );
  GPIO_set_output( &g_gpio_out, GPIO_1,0 );
  GPIO_set_output( &g_gpio_out, GPIO_2,1 );
  GPIO_set_output( &g_gpio_out, GPIO_3,0 );


  /*******************************************************************************
   * Initialize CoreUARTapb with its base address, baud value, and line
   * configuration.
   ******************************************************************************/
  UART_init(&g_uart,
                  COREUARTAPB0_BASE_ADDR,
                        BAUD_VALUE_115200,
                        (DATA_8_BITS | NO_PARITY));

  /*******************************************************************************
   * Send the instructions message.
   ******************************************************************************/
  UART_polled_tx_string(&g_uart, (const uint8_t *)&testmsg);

#endif
#if 1
  /*
```

```
{
  volatile int32_t delay_count = 0;

#if 1
  /*******************************************************************************
   * Initialize the CoreGPIO driver with the base address of the CoreGPIO
   * instance to use and the initial state of the outputs.
   ******************************************************************************/
  GPIO_init(&g_gpio_out, COREGPIO_OUT_BASE_ADDR, GPIO_APB_32_BITS_BUS);

  /*******************************************************************************
   * Configure the GPIOs.
   ******************************************************************************/
//  GPIO_config( &g_gpio_out, GPIO_0, GPIO_OUTPUT_MODE );
//  GPIO_config( &g_gpio_out, GPIO_1, GPIO_OUTPUT_MODE );
//  GPIO_config( &g_gpio_out, GPIO_2, GPIO_OUTPUT_MODE );
//  GPIO_config( &g_gpio_out, GPIO_3, GPIO_OUTPUT_MODE );

  /*******************************************************************************
   * Set the GPIO outputs.
   ******************************************************************************/
  GPIO_set_output( &g_gpio_out, GPIO_0,1 );
  GPIO_set_output( &g_gpio_out, GPIO_1,0 );
  GPIO_set_output( &g_gpio_out, GPIO_2,1 );
  GPIO_set_output( &g_gpio_out, GPIO_3,0 );


  /*******************************************************************************
   * Initialize CoreUARTapb with its base address, baud value, and line
   * configuration.
   ******************************************************************************/
  UART_init(&g_uart,
                  COREUARTAPB0_BASE_ADDR,
                        BAUD_VALUE_115200,
                        (DATA_8_BITS | NO_PARITY));

  /*******************************************************************************
   * Send the instructions message.
   ******************************************************************************/
  UART_polled_tx_string(&g_uart, (const uint8_t *)&testmsg);

#endif
#if 1
  /*
```

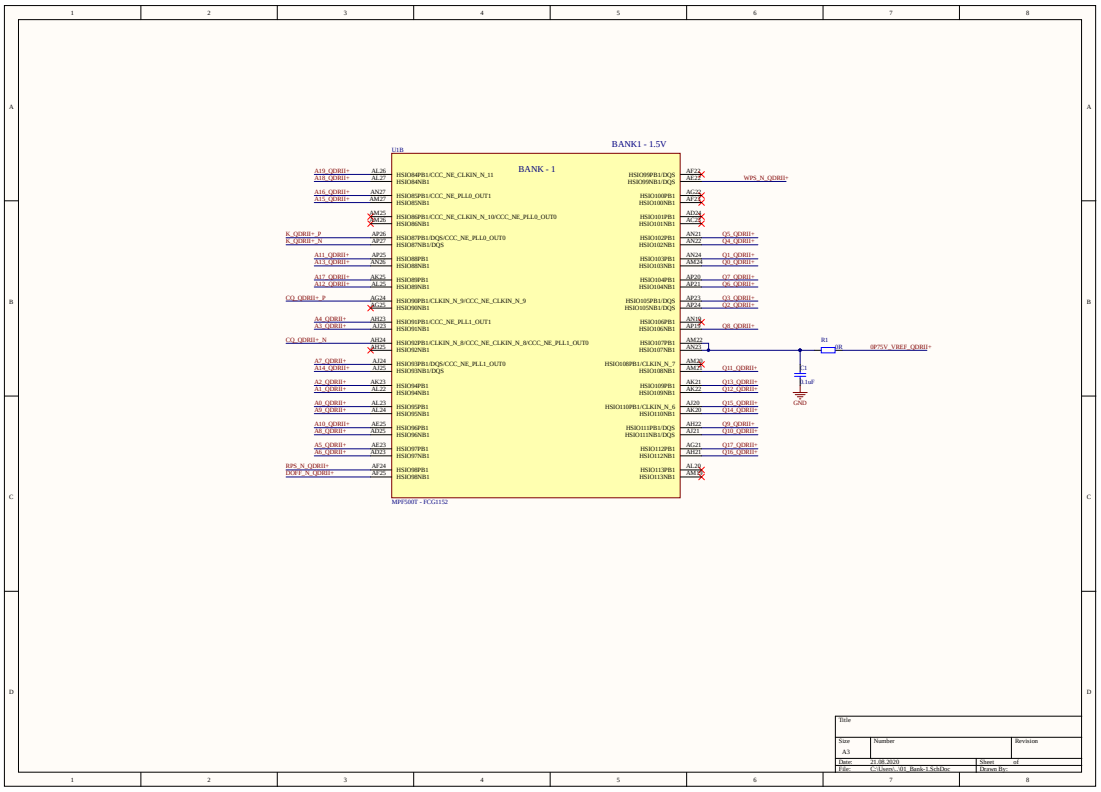
```
    * Set initial delay used to blink the LED.
    */
   delay_count = DELAY_LOAD_VALUE;

   /*
    * Infinite loop.
    */
   for(;;)
   {
           uint32_t gpio_pattern;
           /*
            * Decrement delay counter.
            */
           --delay_count;

           /*
            * Check if delay expired.
            */
           if ( delay_count <= 0 )
           {
                   /*
                    * Reload delay counter.
                    */
                   delay_count = DELAY_LOAD_VALUE;

                   /*
                    * Toggle GPIO output pattern by doing an exclusive OR of all
                    * pattern bits with ones.
                    */
                   gpio_pattern = GPIO_get_outputs( &g_gpio_out );
                   gpio_pattern ^= 0x0000000F;
                   GPIO_set_outputs( &g_gpio_out, gpio_pattern );
           }
   }

#endif

   return 0;

}
```

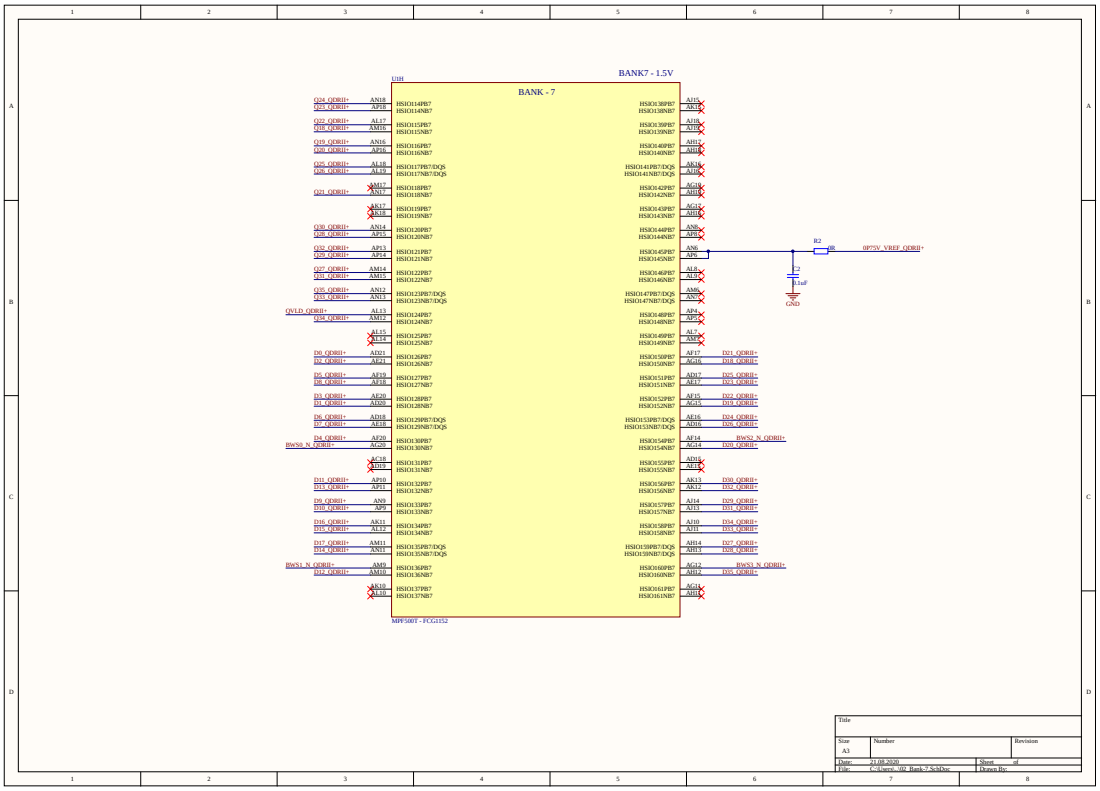
Make File wth GNU C Toolchain

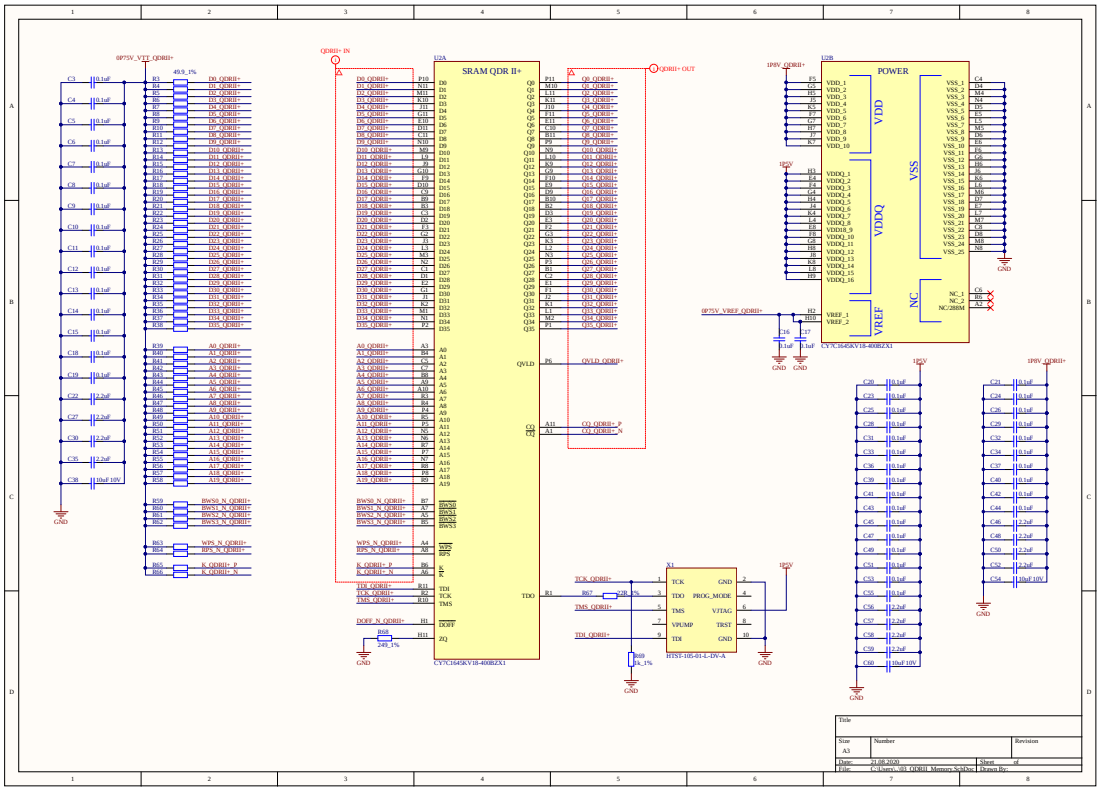
```
################################################################################
# Automatically-generated file. Do not edit!
################################################################################

-include ../makefile.init

RM := rm -rf

# All of the sources participating in the build are defined here
-include sources.mk
-include miv_rv32_hal/subdir.mk
-include hal/subdir.mk
-include drivers/CoreUARTapb/subdir.mk
-include drivers/CoreSPI/subdir.mk
-include drivers/CoreGPIO/subdir.mk
-include .metadata/.plugins/org.eclipse.cdt.make.core/subdir.mk
-include subdir.mk
-include objects.mk

ifneq ($(MAKECMDGOALS),clean)
ifneq ($(strip $(ASM_DEPS)),)
-include $(ASM_DEPS)
endif
ifneq ($(strip $(S_UPPER_DEPS)),)
-include $(S_UPPER_DEPS)
endif
ifneq ($(strip $(C_DEPS)),)
-include $(C_DEPS)
endif
endif

-include ../makefile.defs

# Add inputs and outputs from these tool invocations to the build variables
SECONDARY_FLASH += \
MiV_uart_blinky.hex \

SECONDARY_SIZE += \
MiV_uart_blinky.siz \
```

```
# All Target
all: MiV_uart_blinky.elf secondary-outputs

# Tool invocations
MiV_uart_blinky.elf: $(OBJS) $(USER_OBJS)
	@echo 'Building target: $@'
	@echo 'Invoking: GNU RISC-V Cross C Linker'
	riscv64-unknown-elf-gcc -march=rv32im -mabi=ilp32 -msmall-data-limit=8 -mno-save-restore -
Os -fmessage-length=0 -fsigned-char -ffunction-sections -fdata-sections  -g -T
"C:\WFH_Tasks\SoftConsole\MiV_uart_blinky\miv_rv32_hal\miv-rv32-ram.ld" -nostartfiles -Xlinker --gc-
sections -Wl,-Map,"MiV_uart_blinky.map" -o "MiV_uart_blinky.elf" $(OBJS) $(USER_OBJS) $(LIBS)
	@echo 'Finished building target: $@'
	@echo ' '

MiV_uart_blinky.hex: MiV_uart_blinky.elf
	@echo 'Invoking: GNU RISC-V Cross Create Flash Image'
	riscv64-unknown-elf-objcopy -O ihex --change-section-lma *-0x80000000 "MiV_uart_blinky.elf"
"MiV_uart_blinky.hex"
	@echo 'Finished building: $@'
	@echo ' '

MiV_uart_blinky.siz: MiV_uart_blinky.elf
	@echo 'Invoking: GNU RISC-V Cross Print Size'
	riscv64-unknown-elf-size --format=berkeley "MiV_uart_blinky.elf"
	@echo 'Finished building: $@'
	@echo ' '

# Other Targets
clean:
	-$(RM)
$(OBJS)$(SECONDARY_FLASH)$(SECONDARY_SIZE)$(ASM_DEPS)$(S_UPPER_DEPS)$(C_DEPS)
MiV_uart_blinky.elf
	-@echo ' '

secondary-outputs: $(SECONDARY_FLASH) $(SECONDARY_SIZE)

.PHONY: all clean dependents

-include ../makefile.targets
```

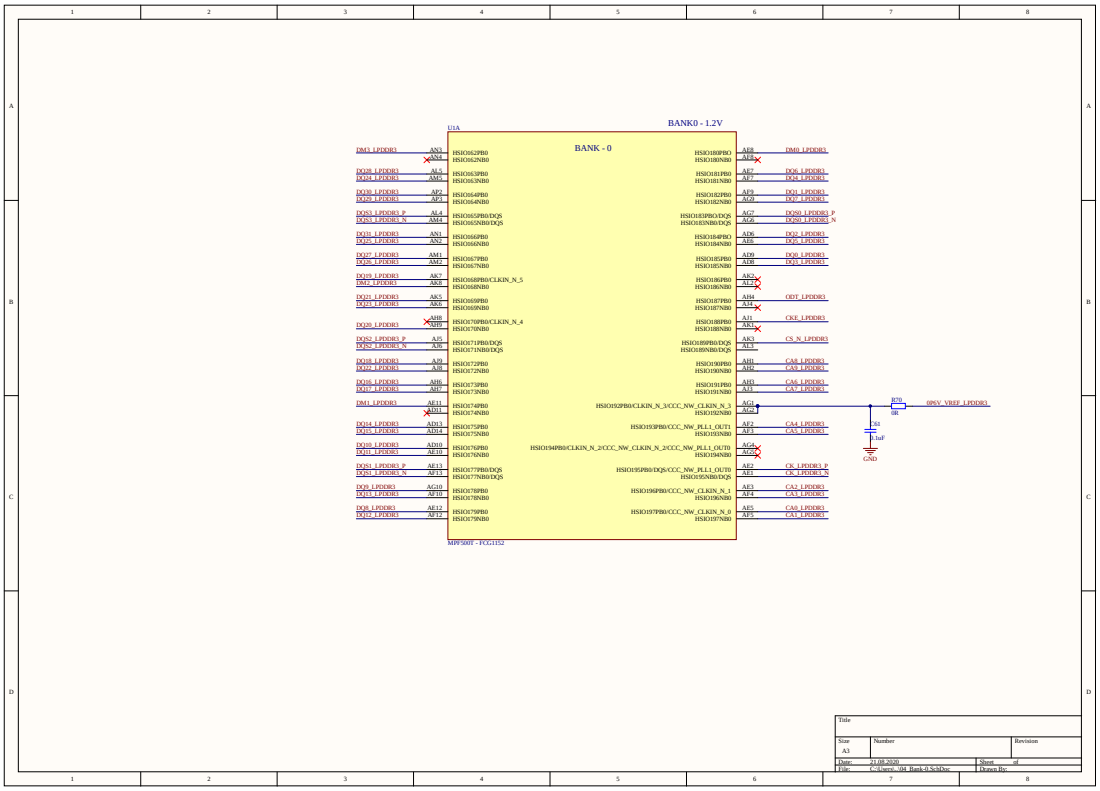
11 Board Schematics

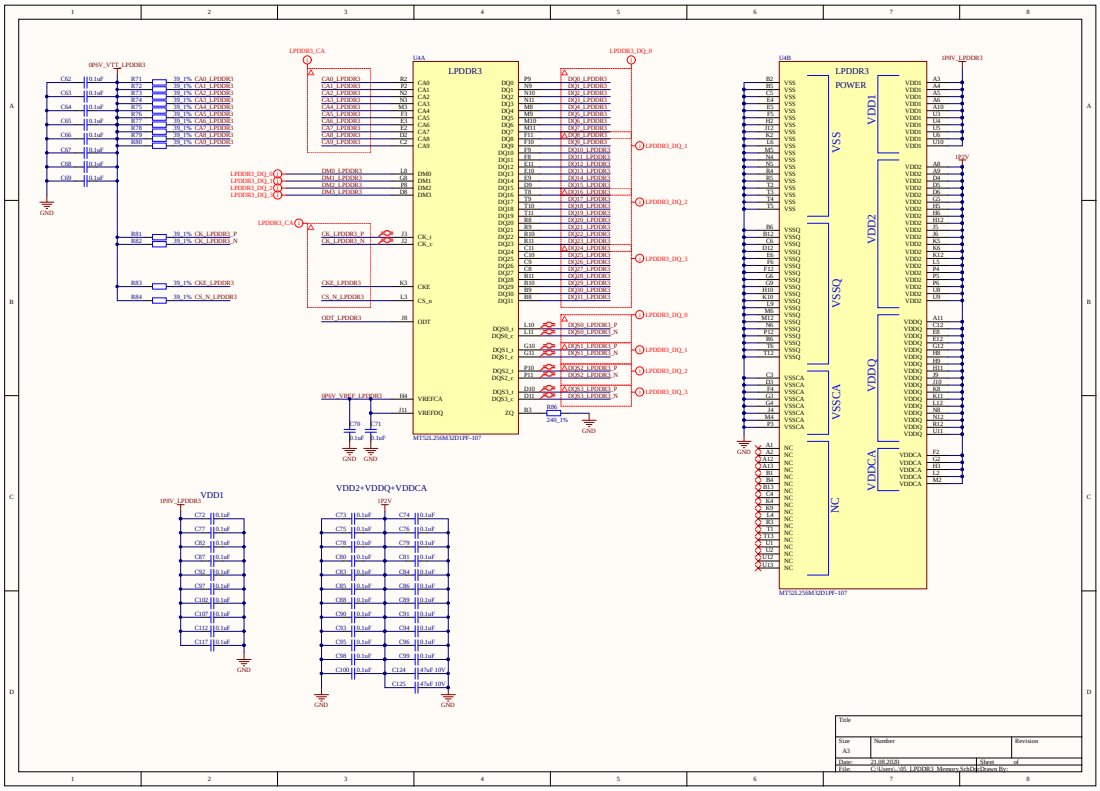


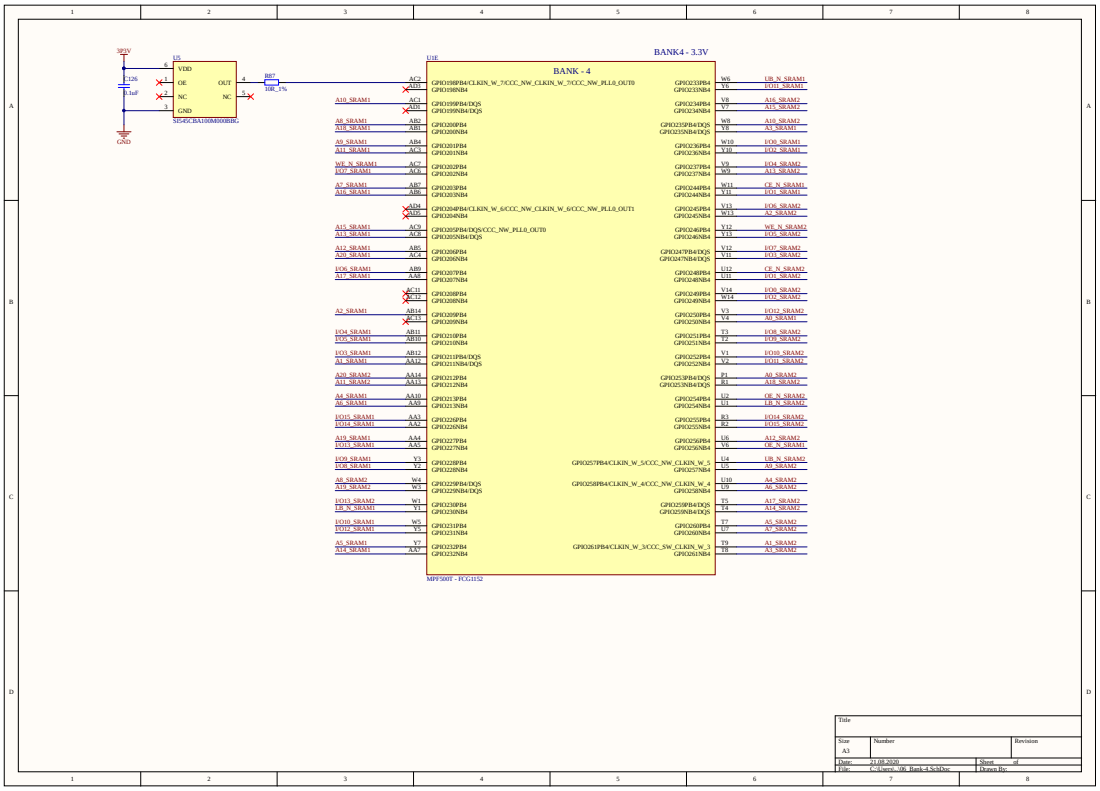

BANK7 - 1.5V
U1H
BANK - 7
MPF500T - FCG1152
0P75V_VREF_QDRII+
R2
C2
0.1uF
GND
Title
Size
A3
Number
Revision
Date: 21.08.2020
Sheet of
File: C:\Users\...\02_Bank-7.SchDoc
Drawn By:

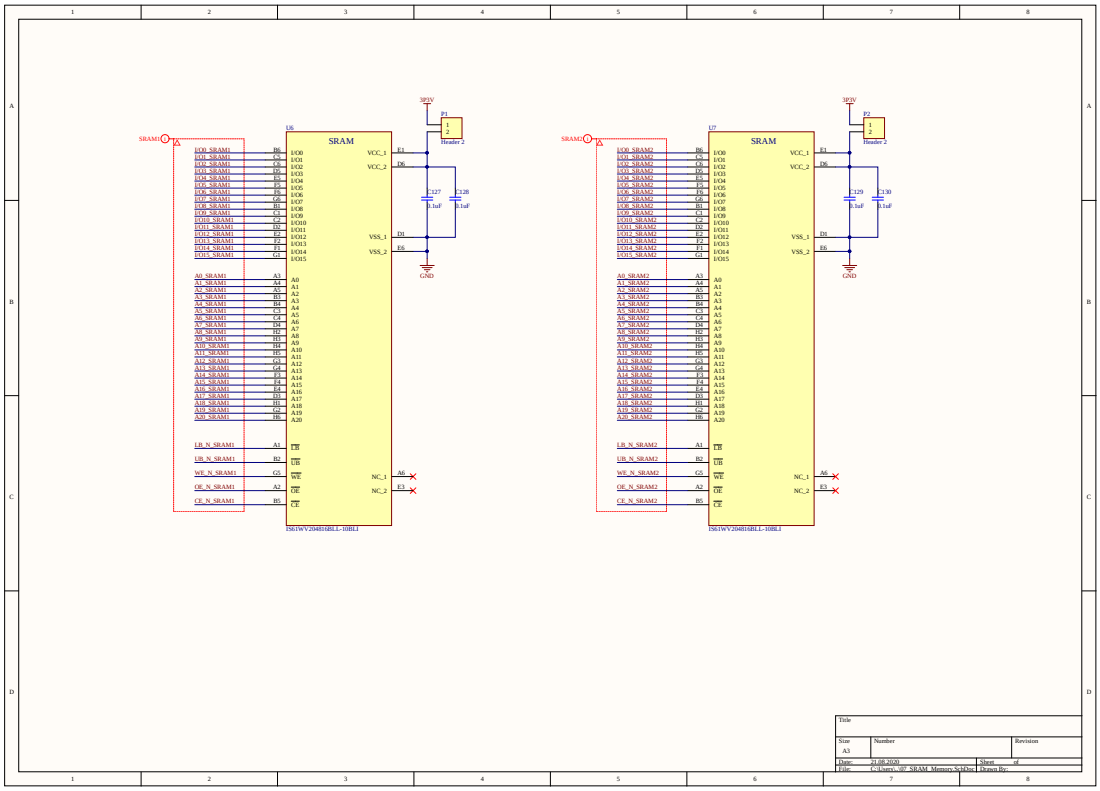

SRAM QDR II+
POWER
VDD
VSS
VDDQ
NC
VREF
CY7C1645KV18-400BZX1
HTST-105-01-L-DV-A
QDRII+ IN
QDRII+ OUT
0P75V_VTT_QDRII+
1P8V_QDRII+
1P5V
0P75V_VREF_QDRII+
GND

11 Board Schematics

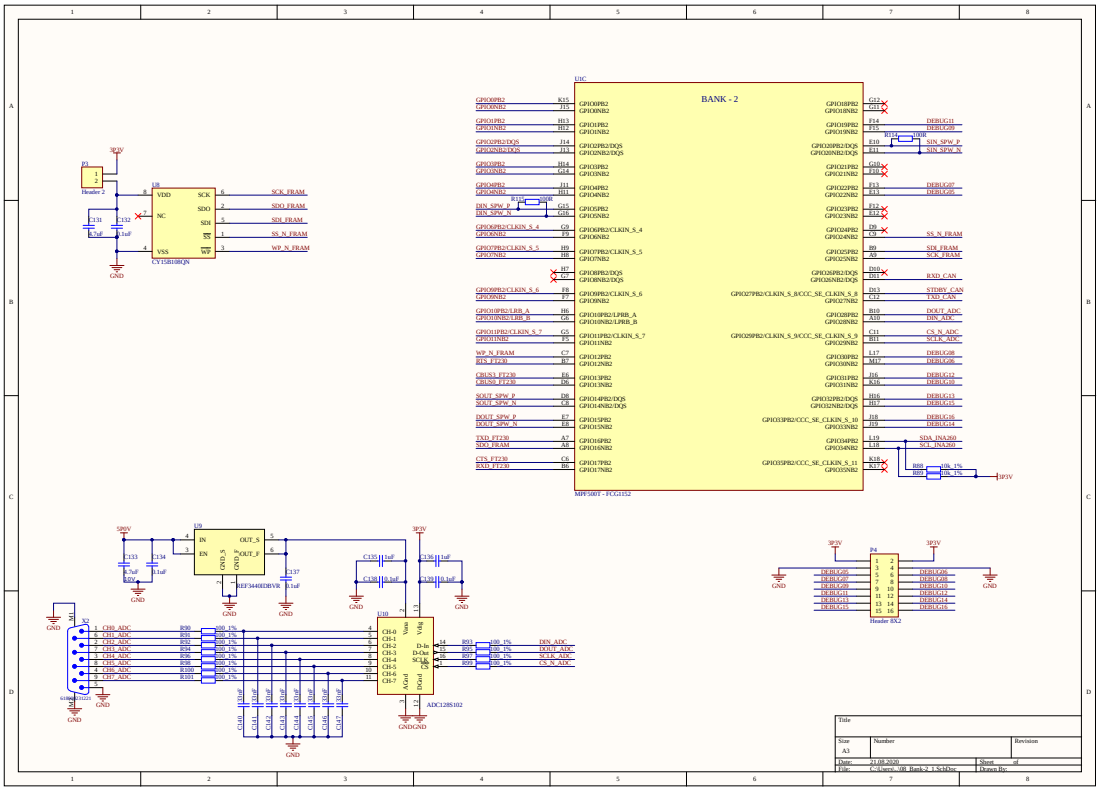


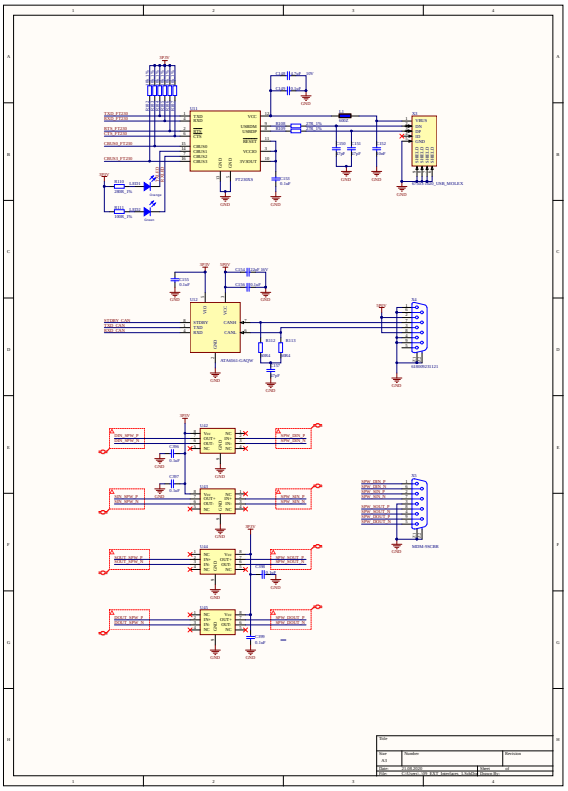

LPDDR3
POWER
VSS
VDD1
VDD2
VSSQ
VDDQ
VSSCA
VDDCA
NC
MT52L256M32D1PF-107
VDD1
VDD2+VDDQ+VDDCA

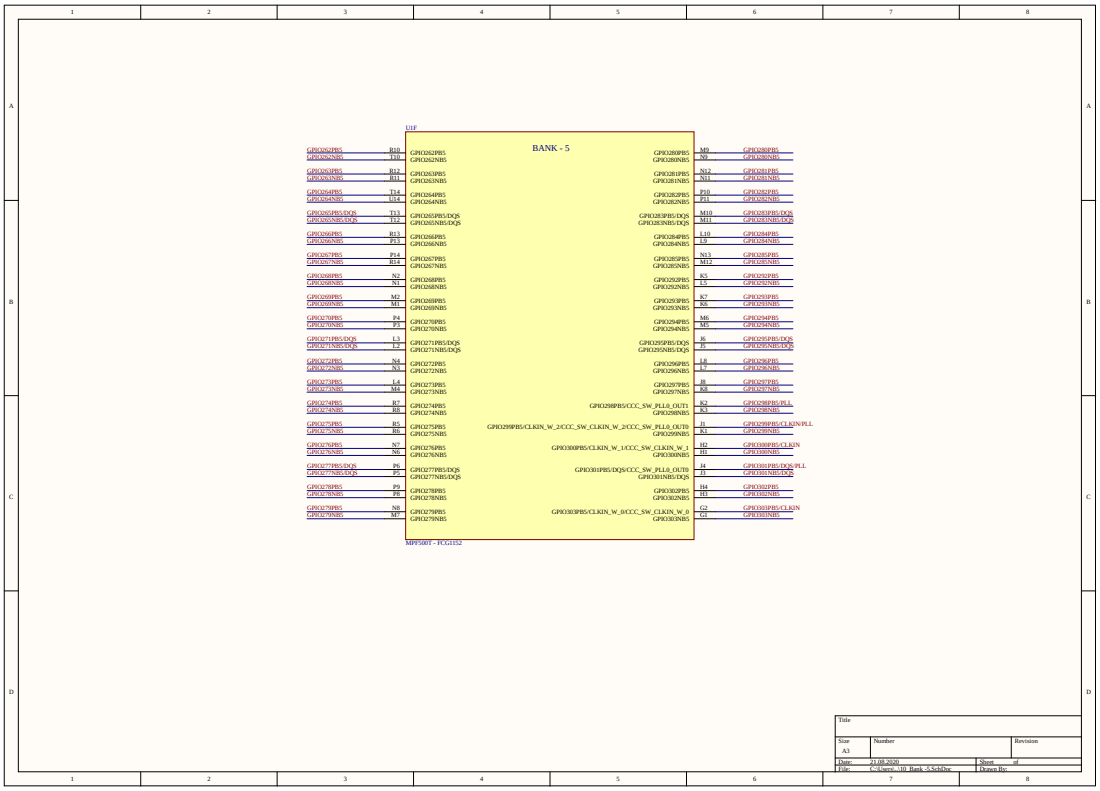

BANK4 - 3.3V
BANK - 4
U1E
U5
3P3V
GND
B87
49R_1%
C126
0.1uF
MS45C-BA100M00BR-
MPF500T - FCG1152
Title
Size
A3
Number
Revision
Date: 21.08.2020
Sheet of
File: C:\Users\...\09_Bank-4.SchDoc
Drawn By:

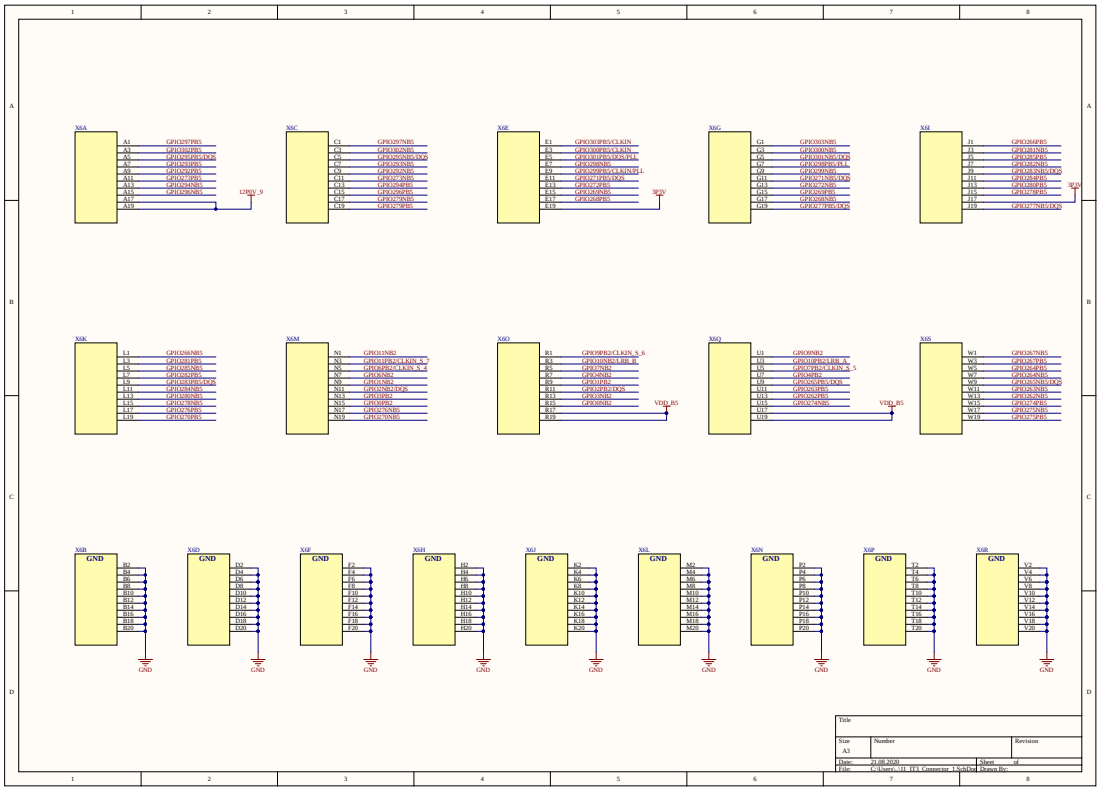

SRAM1
U6
SRAM
3P3V
P1
Header 2
C127
0.1uF
C128
0.1uF
GND
IO0_SRAM1
IO1_SRAM1
IO2_SRAM1
IO3_SRAM1
IO4_SRAM1
IO5_SRAM1
IO6_SRAM1
IO7_SRAM1
IO8_SRAM1
IO9_SRAM1
IO10_SRAM1
IO11_SRAM1
IO12_SRAM1
IO13_SRAM1
IO14_SRAM1
IO15_SRAM1
A0_SRAM1
A1_SRAM1
A2_SRAM1
A3_SRAM1
A4_SRAM1
A5_SRAM1
A6_SRAM1
A7_SRAM1
A8_SRAM1
A9_SRAM1
A10_SRAM1
A11_SRAM1
A12_SRAM1
A13_SRAM1
A14_SRAM1
A15_SRAM1
A16_SRAM1
A17_SRAM1
A18_SRAM1
A19_SRAM1
A20_SRAM1
LB_N_SRAM1
UB_N_SRAM1
WE_N_SRAM1
OE_N_SRAM1
CE_N_SRAM1
VCC_1
VCC_2
VSS_1
VSS_2
NC_1
NC_2
IS61WV204816BLL-10BLI
SRAM2
U7
SRAM
3P3V
P2
Header 2
C129
0.1uF
C130
0.1uF
GND
IO0_SRAM2
IO1_SRAM2
IO2_SRAM2
IO3_SRAM2
IO4_SRAM2
IO5_SRAM2
IO6_SRAM2
IO7_SRAM2
IO8_SRAM2
IO9_SRAM2
IO10_SRAM2
IO11_SRAM2
IO12_SRAM2
IO13_SRAM2
IO14_SRAM2
IO15_SRAM2
A0_SRAM2
A1_SRAM2
A2_SRAM2
A3_SRAM2
A4_SRAM2
A5_SRAM2
A6_SRAM2
A7_SRAM2
A8_SRAM2
A9_SRAM2
A10_SRAM2
A11_SRAM2
A12_SRAM2
A13_SRAM2
A14_SRAM2
A15_SRAM2
A16_SRAM2
A17_SRAM2
A18_SRAM2
A19_SRAM2
A20_SRAM2
LB_N_SRAM2
UB_N_SRAM2
WE_N_SRAM2
OE_N_SRAM2
CE_N_SRAM2
VCC_1
VCC_2
VSS_1
VSS_2
NC_1
NC_2
IS61WV204816BLL-10BLI
Title
Size
A3
Number
Revision
Date:
21.08.2020
Sheet
of
File:
C:\Users\...\07_SRAM_Memory.SchDoc
Drawn By:

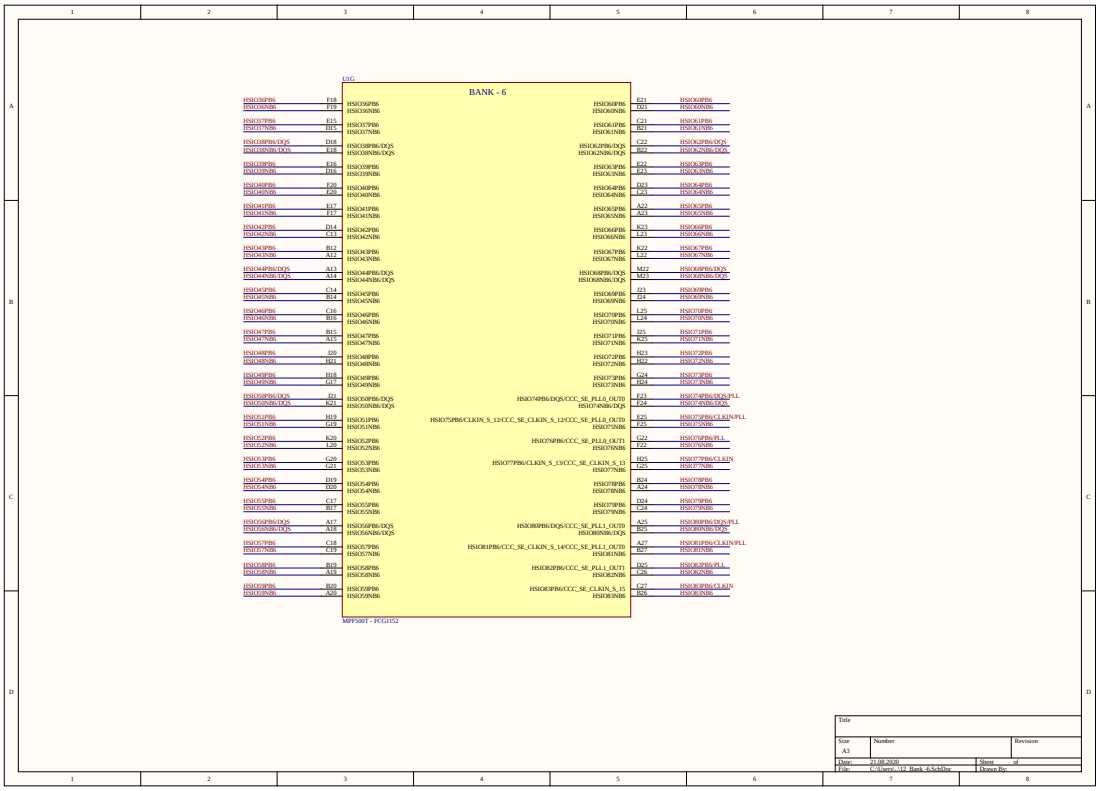
11 Board Schematics

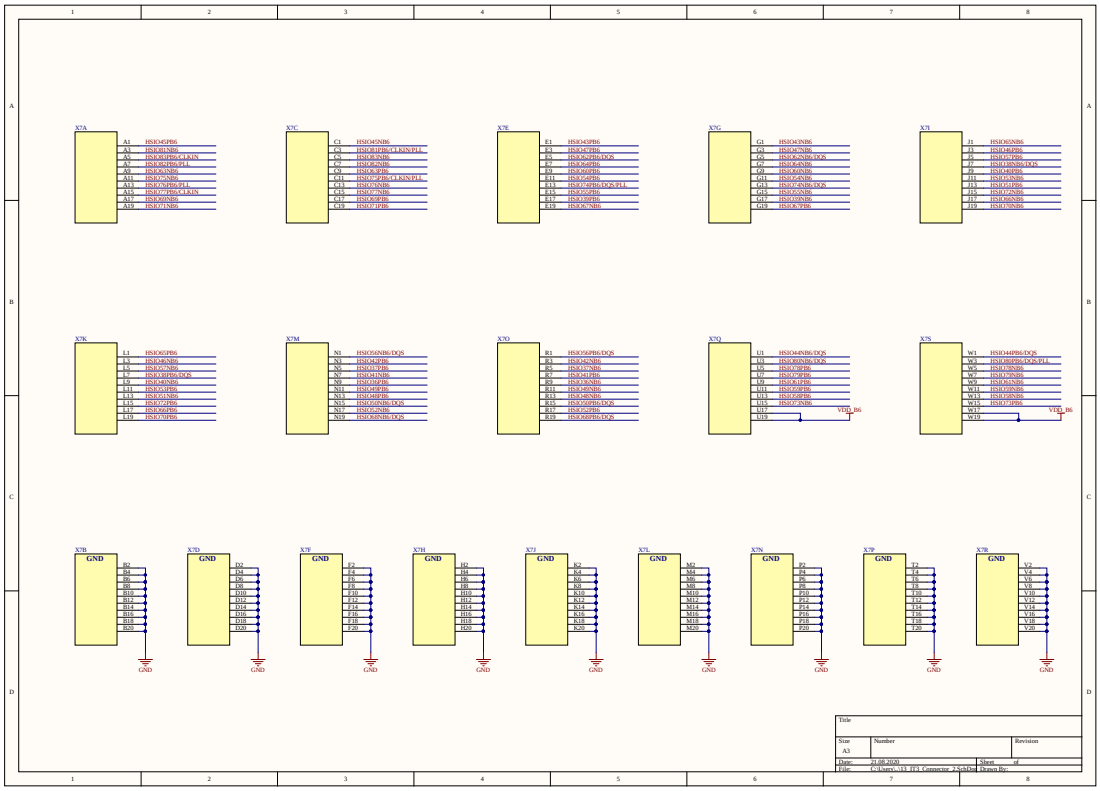




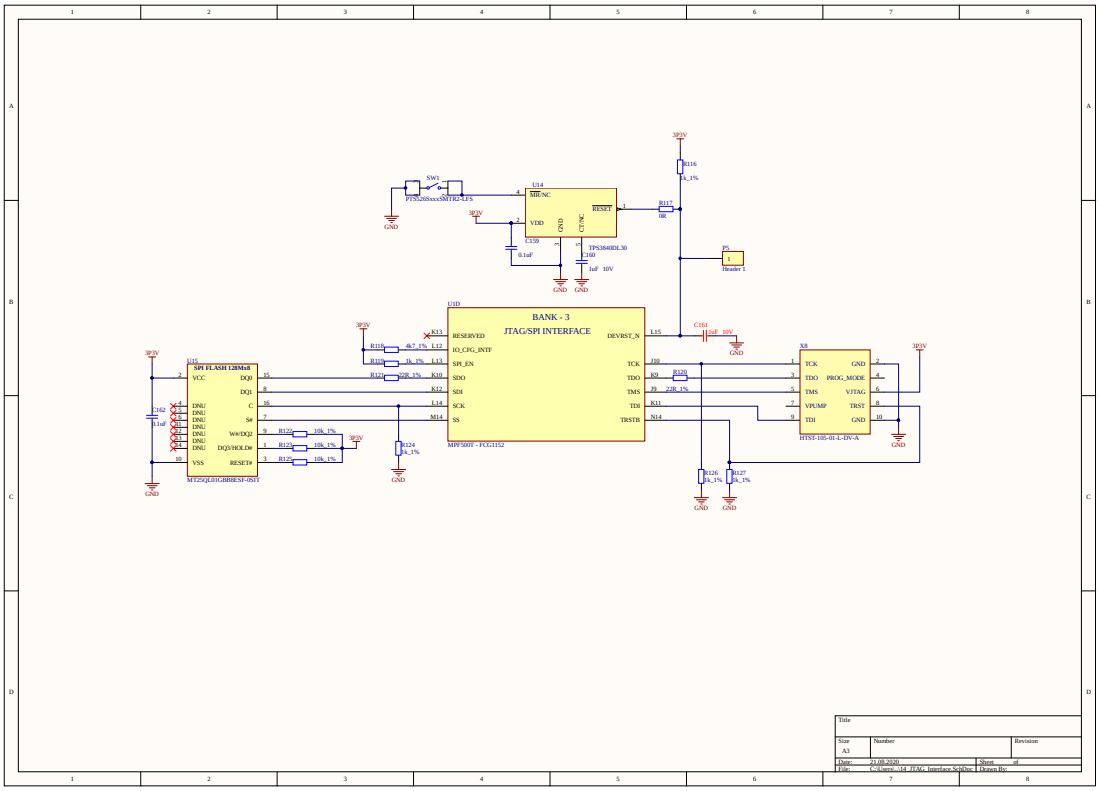

U1F
BANK - 5
MPF500T - FCG1152
GPIO262PB5 R10
GPIO262NB5 T10
GPIO263PB5 R12
GPIO263NB5 R11
GPIO264PB5 T14
GPIO264NB5 U14
GPIO265PB5/DQS T13
GPIO265NB5/DQS T12
GPIO266PB5 R13
GPIO266NB5 P13
GPIO267PB5 P14
GPIO267NB5 R14
GPIO268PB5 N2
GPIO268NB5 N3
GPIO269PB5 M2
GPIO269NB5 M1
GPIO270PB5 P4
GPIO270NB5 P3
GPIO271PB5/DQS L3
GPIO271NB5/DQS L2
GPIO272PB5 N4
GPIO272NB5 N3
GPIO273PB5 L4
GPIO273NB5 M4
GPIO274PB5 R7
GPIO274NB5 R8
GPIO275PB5 R5
GPIO275NB5 R6
GPIO276PB5 N7
GPIO276NB5 N6
GPIO277PB5/DQS P6
GPIO277NB5/DQS P5
GPIO278PB5 P9
GPIO278NB5 P8
GPIO279PB5 N8
GPIO279NB5 M7
GPIO262PB5/PCIE_1_PERST_N
GPIO262NB5
GPIO263PB5
GPIO263NB5
GPIO264PB5
GPIO264NB5
GPIO265PB5/DQS
GPIO265NB5/DQS
GPIO266PB5
GPIO266NB5
GPIO267PB5
GPIO267NB5
GPIO268PB5
GPIO268NB5
GPIO269PB5
GPIO269NB5
GPIO270PB5
GPIO270NB5
GPIO271PB5/DQS
GPIO271NB5/DQS
GPIO272PB5
GPIO272NB5
GPIO273PB5
GPIO273NB5
GPIO274PB5
GPIO274NB5
GPIO275PB5
GPIO275NB5
GPIO276PB5
GPIO276NB5
GPIO277PB5/DQS
GPIO277NB5/DQS
GPIO278PB5
GPIO278NB5
GPIO279PB5
GPIO279NB5
GPIO282PB5 M9
GPIO282NB5 N9
GPIO281PB5 N12
GPIO281NB5 N11
GPIO282PB5 P10
GPIO282NB5 P11
GPIO283PB5/DQS M10
GPIO283NB5/DQS M11
GPIO284PB5 L10
GPIO284NB5 L9
GPIO285PB5 N13
GPIO285NB5 M12
GPIO292PB5 K5
GPIO292NB5 L5
GPIO293PB5 K7
GPIO293NB5 K6
GPIO294PB5 M6
GPIO294NB5 M5
GPIO295PB5/DQS J6
GPIO295NB5/DQS J5
GPIO296PB5 L8
GPIO296NB5 L7
GPIO297PB5 J8
GPIO297NB5 K8
GPIO298PB5/PLL_2_OUT1 K2
GPIO298NB5 K3
GPIO299PB5/CLKIN_W_2/CCC_SW_PLL0_OUT0 J1
GPIO299NB5 K1
GPIO300PB5/CLKIN_W_1/CCC_SW_CLKIN_W_1 H2
GPIO300NB5 H1
GPIO301PB5/DQS/CCC_SW_PLL0_OUT6 J4
GPIO301NB5/DQS J3
GPIO302PB5 H4
GPIO302NB5 H3
GPIO303PB5/CLKIN_W_0/CCC_SW_CLKIN_W_0 G2
GPIO303NB5 G1
Title
Size A3
Number
Revision
Date: 21-08-2020
File: C:\Users\...\10_Bank_5.SchDoc
Sheet of
Drawn By:

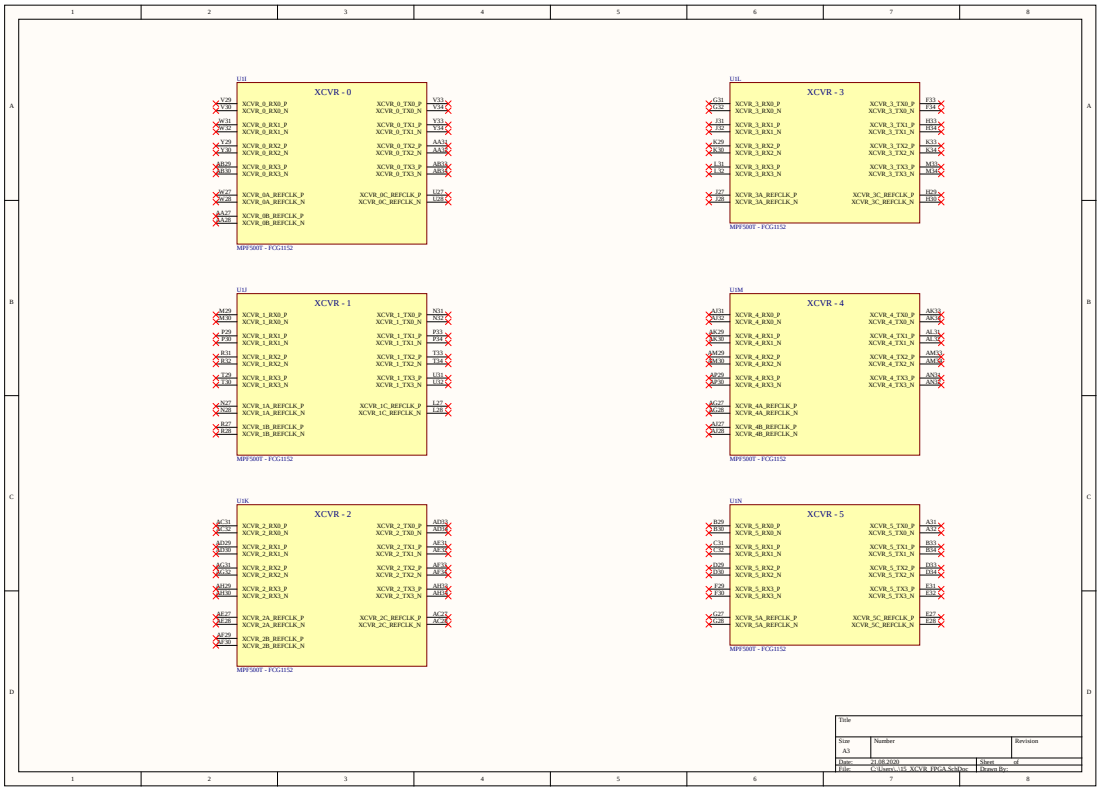

X6A
X6C
X6E
X6G
X6I
X6K
X6M
X6O
X6Q
X6S
X6B
X6D
X6F
X6H
X6J
X6L
X6N
X6P
X6R
GND
12PDV_9
3PDV
VDD_BS
Title
Size
A3
Number
Revision
Date:
21.08.2020
Sheet
of
File:
C:\Users\...\11_1T3_Connector_1.SchDoc
Drawn By:

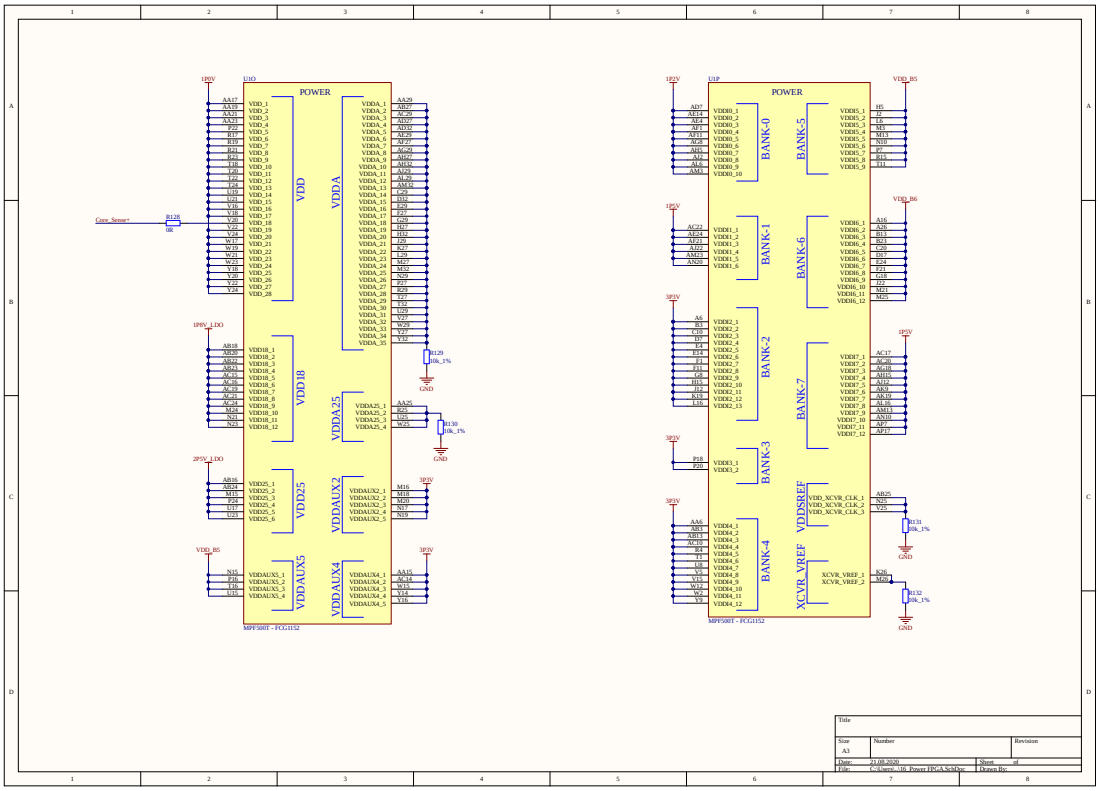

U1G
BANK - 6
MPF500T - FCG1152
Title
Size
A3
Number
Revision
Date: 21.08.2020
Sheet of
File: C:\Users\...\12_Bank_6.SchDoc
Drawn By:

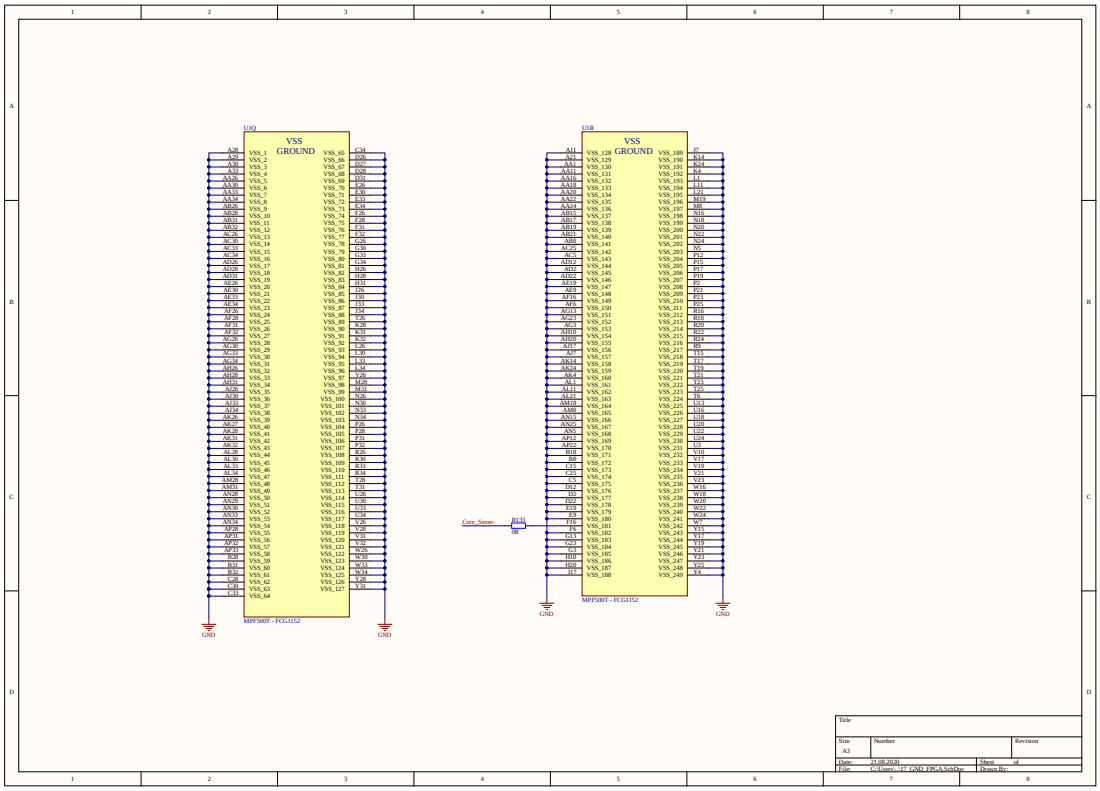

X7A
X7C
X7E
X7G
X7I
X7K
X7M
X7O
X7Q
X7S
X7B
X7D
X7F
X7H
X7J
X7L
X7N
X7P
X7R
GND
Title
Size
A3
Number
Revision
Date: 21.08.2020
File: C:\Users\...\13_IT3_Connector_2.SchDoc
Sheet of
Drawn By:

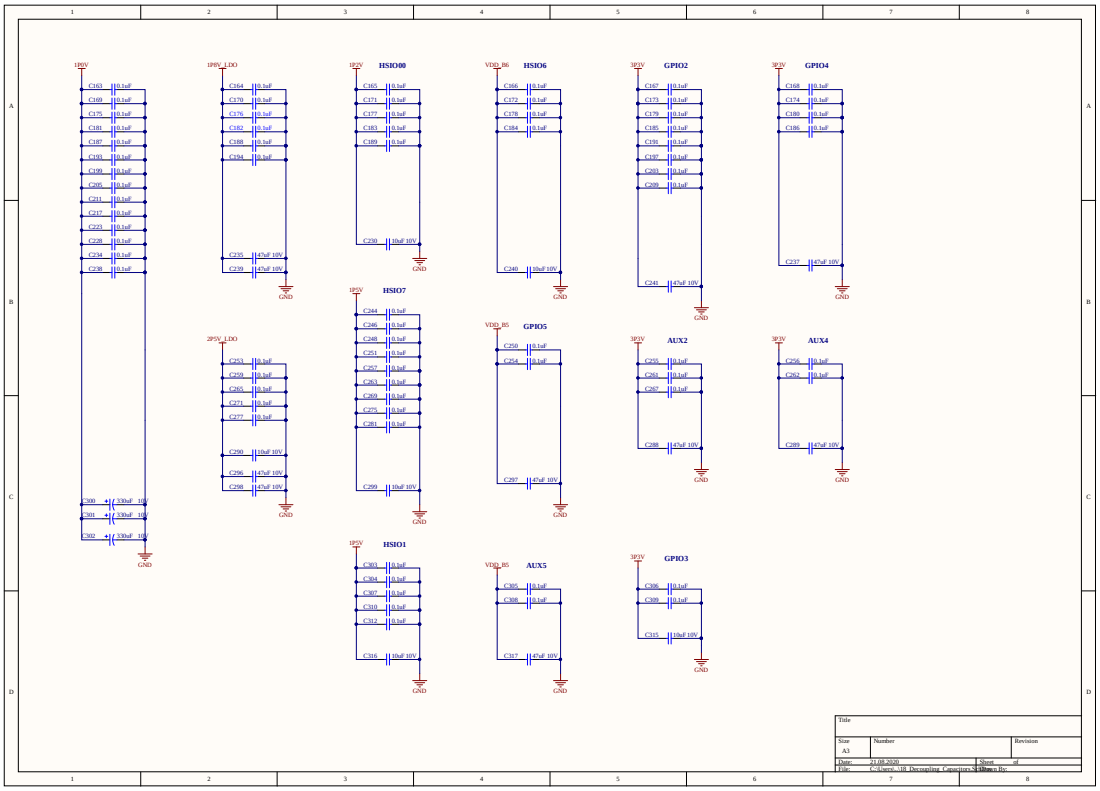
11 Board Schematics

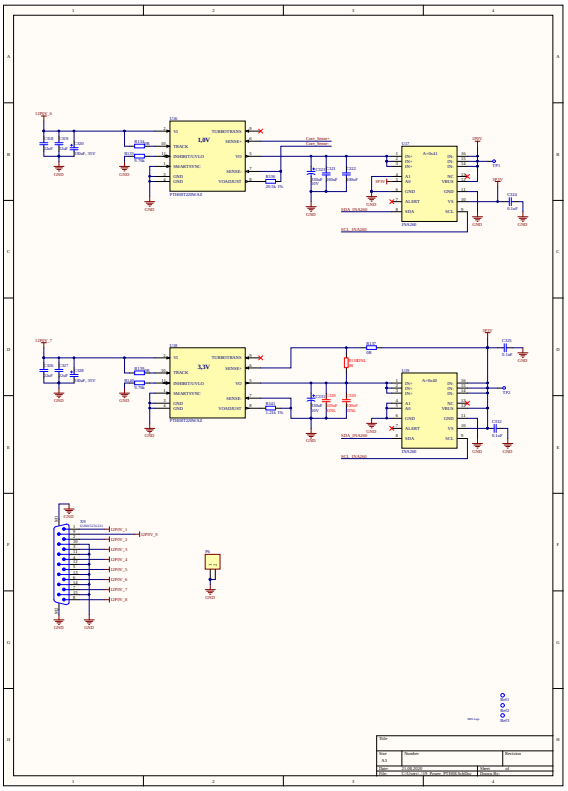


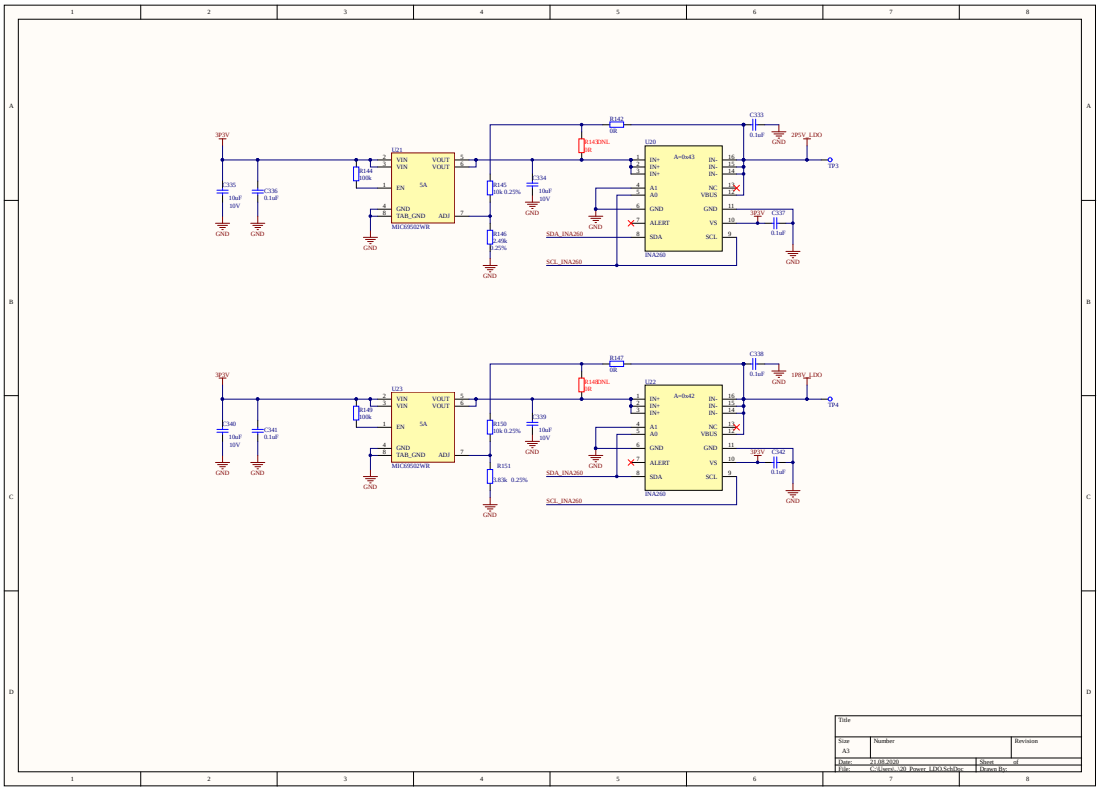

U1I
XCVR - 0
MPF500T - FCG1152
U1L
XCVR - 3
MPF500T - FCG1152
U1J
XCVR - 1
MPF500T - FCG1152
U1M
XCVR - 4
MPF500T - FCG1152
U1K
XCVR - 2
MPF500T - FCG1152
U1N
XCVR - 5
MPF500T - FCG1152
Title
Size
A3
Number
Revision
Date: 21.08.2020
Sheet of
File: C:\Users\...\35_XCVR_FPGA.SchDoc
Drawn By:

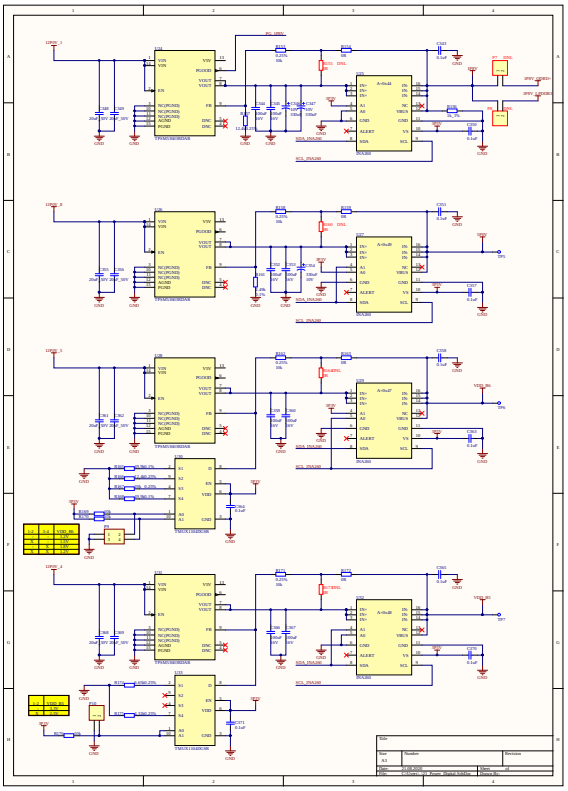

POWER
VDD
VDDA
VDD18
VDDA25
VDDAUX2
VDD25
VDDAUX5
VDDAUX4
BANK-0
BANK-1
BANK-2
BANK-3
BANK-4
BANK-5
BANK-6
BANK-7
VDDSREF
XCVR_VREF
MPF500T - FCG1152
Core_Sense+
R128
0R
R129
10k_1%
R130
10k_1%
R131
10k_1%
R132
10k_1%
GND
1P0V
1P2V
1P5V
2P5V_LDO
1P8V_LDO
3P3V
VDD_B5
VDD_B6
Title
Size
A3
Number
Revision
Date:
21.08.2020
Sheet
of
File:
C:\Users\...\10_Power FPGA.SchDoc
Drawn By:

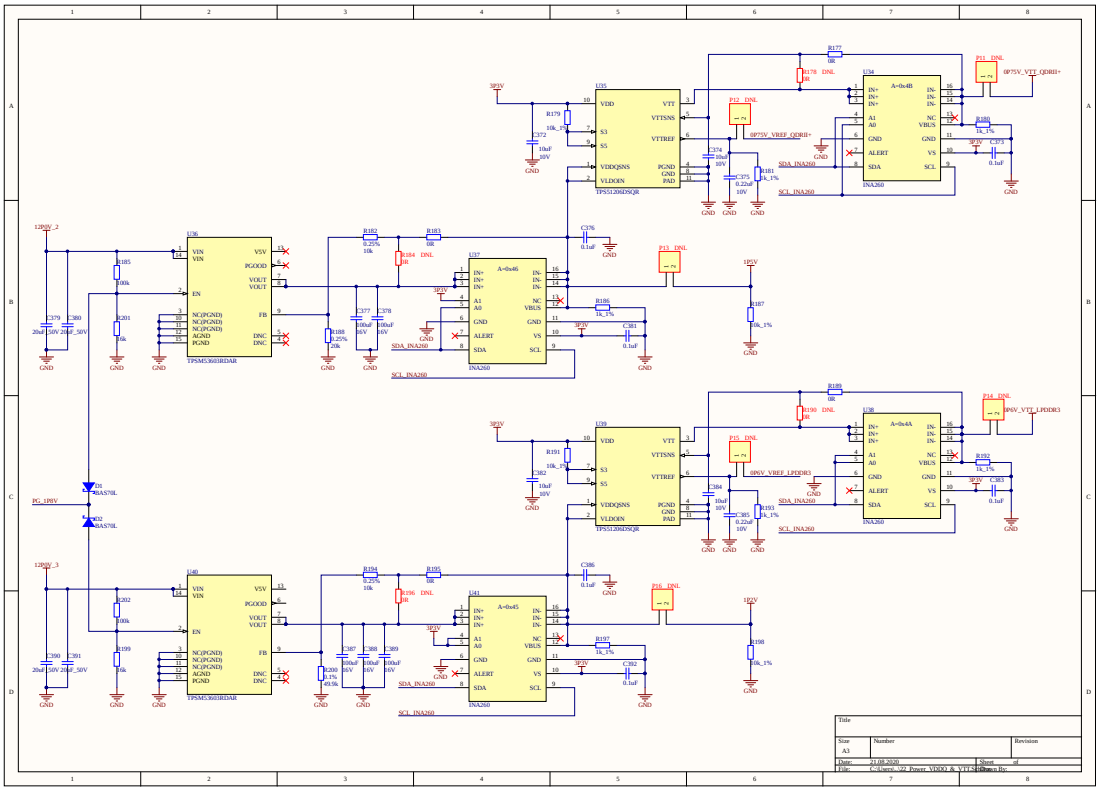

U1Q
VSS
GROUND
MPF500T - FCG1152
GND
U1R
VSS
GROUND
MPF500T - FCG1152
GND
Core_Sense
R133
0R
Title
Size
A3
Number
Revision
Date:
21.08.2020
Sheet
of
File:
C:\Users\...\37_GND_FPGA.SchDoc
Drawn By:

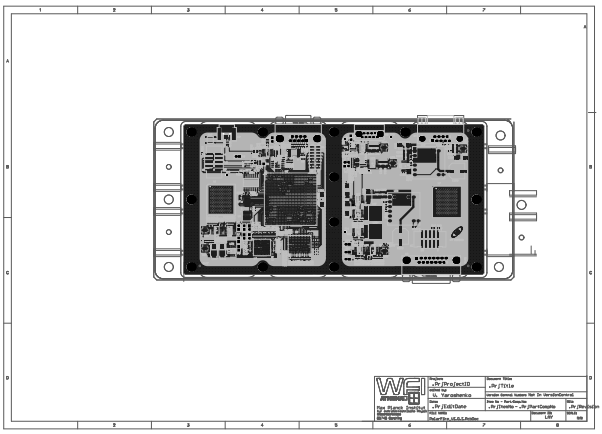

1P8V
1P8V_LDO
2P5V_LDO
1P2V
HSIO00
VDD_B6
HSIO6
3P3V
GPIO2
GPIO4
1P5V
HSIO7
VDD_B5
GPIO5
AUX2
AUX4
HSIO1
AUX5
GPIO3
GND
Title
Size
A3
Number
Revision
Date: 21.08.2020
File: C:\Users\...\IP_Decoupling_Capacitors.SchDoc
Sheet of
Drawn By:




3P3V
U21
MIC69502WR
VIN
VOUT
EN
5A
GND
TAB_GND
ADJ
R144
100k
C335
10uF
10V
C336
0.1uF
R145
10k 0.25%
R146
2.49k
0.25%
C354
10uF
10V
R143
0R
R143NL
DNI
U20
INA260
SDA_INA260
SCL_INA260
ALERT
SDA
SCL
C333
0.1uF
2P5V_LDO
TP3
3P3V
C337
0.1uF
U23
R149
100k
C340
10uF
10V
C341
0.1uF
R150
10k 0.25%
R151
1.83k 0.25%
C339
10uF
10V
R148
0R
R148NL
DNI
U22
C338
0.1uF
1P8V_LDO
TP4
C342
0.1uF
Title
Size
A3
Number
Revision
Date:
21.08.2020
Sheet
of
File:
C:\Users\...\20_Power_LDO.SchDoc
Drawn By:







Bibliography

[1] Stephen Gergely David Crecraft. *Analog Electronics Circuits, Systems and Signal Processing*, chapter 1. Butterworth-Heinemann, 2002.

[2] Sedra Smith. *Microelectronic Circuits*. Oxford University Press, 2010.

[3] Shree Krishna Khadka. *Comparison of IC Logic Families*, page 1. 04 2017.

[4] Bing Li. *Lecture notes in Timing of Digital Circuits*, chapter 3, page 100. Technical University of Munich (TUM), Munich, 2019.

[5] chipverify. *Verilog Clock Generator*. https://www.chipverify.com/verilog/verilog-clock-generator, 2019.

[6] nandland. *What is a Tri-State Buffer*. https://www.nandland.com/articles/tri-state-buffer-half-full-duplex.html, 2019.

[7] Ulf Schlichtmann. *Electronic Design Automation - Lecture Notes*. Technical University of Munich (TUM), 2019.

[8] Ulf Schlichtmann. *VHDL System Design Lab*. Technical Univeristy of Munich (TUM), 2019.

[9] Valentin Hiltl. *Generation of Properties for Design Verification from an Abstract FSM Model*, page 6. Technical University of Munich (TUM), 12.04.2019.

[10] Wikiwand. *Integrated circuit design*. https://www.wikiwand.com/en/Integrated$_c ircuit_d esign$.

[11] Bob Zeidman. *All about FPGAs*. www.eetimes.com/all-about-fpgas/.

[12] Ell C Alchitry. *How Does an FPGA Work?* learn.sparkfun.com/tutorials/how-does-an-fpga-work/look-up-tables.

[13] Hardwarebee. *Overview of Lookup Tables (LUT) in FPGA Design*. https://hardwarebee.com/overview-of-lookup-tables-in-fpga-design/.

[14] Shahul Akthar. *FPGA Design Flow*. allaboutfpga.com/fpga-design-flow/, April 8, 2014.

[15] Tv Tropes. *Useful Notes / Flynn's Taxonomy.* tvtropes.org/pmwiki/pmwiki.php/UsefulNotes/FlynnsTaxonomy.

[16] Joel Sommers. *Program structure and compilation.* https://jsommers.github.io/cbook/programstructure, 2018.

[17] Bjoern Doebel. *Robustness.* TU Dresden, 22.01.2008.

[18] Paul Hamill. *Unit Test Frameworks.* O'Reilly, 2006.

[19] Connor Jan Goldberg. *The Design of a Custom 32-bit RISC CPU and LLVM Compiler Backend Backend.* Rochester Institute of Technology, 8.2017.

[20] Wiki Books. *GNU C Compiler Internals/GNU C Compiler Architecture.* en.wikibooks.org/wiki/GNU$_{CC}ompiler_Internals/GNU_{CC}ompiler_Architecture, 30April2020.$

[21] Robin Eklind. *LLVM IR and Go.* blog.gopheracademy.com/advent-2018/llvm-ir-and-go/, Dec 19, 2018.

[22] Martin Bond. *Why We Need Build Systems.* blog.feabhas.com/2021/06/why-we-need-build-systems/, June 4, 2021.

[23] Walt Kester. *Analog-Digital Conversion*, chapter 3, page 25. Analog Devices, USA, 2000.

[24] Twilio. *UART Explained - An Introduction To The Standard Serial Bus.* https://developer.electricimp.com/resources/uart, USA.

[25] Colin M.L. Burnett. *SPI bus timing diagram.* https://commons.wikimedia.org/wiki/File:SPI$_{t}iming_{d}iagram2.svg, USA, 7September2010.$

[26] Sparkfun. *I2C.* https://learn.sparkfun.com/tutorials/i2c/all, USA, 13 September 2015.

[27] Scott Knowlton. *Understanding the Fundamentals of PCI Express*, pages 2–13. Synopsis White Paper, USA, September 2007.

[28] Wikimedia. *Static Random-Access Memory (SRAM).* https://commons.wikimedia.org/wiki/File:SRAM$_{C}ell_{(}6_{T}ransistors).svg, USA.$

[29] Andrew S. Tanenbaum. *Structured Computer Organization*, chapter 3.3, page 166. Pearson Prentice Hall Pearson Education, Inc., USA, 2006.

[30] Wai-Kai Chen. *The VLSI handbook.* CRC press, USA, 2016.

[31] Mark LaPedus. *DRAM Scaling Challenges Grow.* https://semiengineering.com/dram-scaling-challenges-grow/, USA, 21 November, 2019.

[32] Electronics Notes. *What is FRAM Memory: ferroelectric RAM*, page 1. https://www.electronics-notes.com/articles/electronic$_c$*omponents*/*semiconductor* − *ic* − *memory*/*fram* − *ferroelectric* − *ram* − *memory.php*, *USA*.

[33] Avinash Aravindan. *Flash 101: NAND Flash vs NOR Flash*, page 1. embedded.com, India.

[34] Prabal Dutta. *ROM, EPROM, AND EEPROM TECHNOLOGY*, chapter 9, page 9. University of Michigan, Michigan,USA.

[35] Microsemi. *Libero IDE*. www.microsemi.com/product-directory/design-resources/1751-libero-ide, 2020.

[36] Microsemi. *Sample Tcl script to run Libero SoC flow in batch mode for SmartFusion2*. www.actel.com/kb/article.aspx?id=SL5619, 2021.

[37] Xilinx. *Constraints Guide*. www.xilinx.com/support/documentation, April 1, 2013.

[38] Sudarshan Sharma. *Writing LUT level design*. www.sudarshan-sh.com/posts/2019/09/writing$_l$*ut*$_l$*evel*$_d$*esign*/, *September*28, 2019.

[39] Ulf Schlichtmann. *Electronic Design Automation - Lecture Notes*. Technical University of Munich (TUM), 2019.

[40] Mentor Graphics. *ModelSim® PE User's Manual*. wikis.ece.iastate.edu/cpre584/images/3/3c/Modelsim$_p$*e*$_u$*ser*$_1$0.0*d.pdf*, 2011.

[41] Microsemi. *RISC-V CPUs*. www.microsemi.com/product-directory/mi-v-risc-v-ecosystem/4406-risc-v-cpus, 2020.

[42] Microsemi. *PolarFire FPGA: Building a Mi-V Processor Subsystem*. Microsemi Corporation, 2021.

[43] Max Planck Institute for extraterrestrial Physics. *Wide Field Imager Overview*. www.mpe.mpg.de/ATHENA-WFI/.

[44] Don Bouldin. *ASIC by Design - Automated design of digital signal processing application-specific integrated circuits*, page 4. 09 2004.

[45] Peter D. Hiscocks. *Analog Circuit Design*. 03 2011.

[46] Vrit Raval. *Abstraction Levels in Verilog*. https://medium.com/verilog-novice-to-wizard/abstraction-levels-in-verilog-2f663786b03f, 2019.

[47] geeksforgeeks. *Combinational and Sequential Circuits*. https://www.geeksforgeeks.org/combinational-and-sequential-circuits/, 2018.

[48] Jonas Julian Jensen. *Why are Latches Bad and How to Avoid Them.* https://vhdlwhiz.com/why-latches-are-bad/, 2018.

[49] Embedded.com. *Asynchronous adn Synchronous Reset.* https://www.embedded.com/asynchronous-reset-synchronization-and-distribution-challenges-and-solutions/, 2019.

[50] Tyler Wojciechowicz. *How are clock signals produced?* https://www.semiconductorstore.com/blog/2018/What-Are-Clock-Signals-in-Digital-Circuits-and-How-Are-They-Produced-Symmetry-Blog/3322/, 2019.

[51] Tyler Wojciechowicz. *Synchronous and free-running designs.* https://www.semiconductorstore.com/blog/2018/What-Are-Clock-Signals-in-Digital-Circuits-and-How-Are-They-Produced-Symmetry-Blog/3322/, 2019.

[52] Tyler Wojciechowicz. *Internal and external oscillators.* https://www.semiconductorstore.com/blog/2018/What-Are-Clock-Signals-in-Digital-Circuits-and-How-Are-They-Produced-Symmetry-Blog/3322/, 2019.

[53] Linda Lua. *Clock Tree 101.* https://www.mouser.com/pdfdocs/clock-tree-101-timing-basics.pdf, 2018.

[54] Inc. Micrel. *Differential Clock Translation.* https://www.microchip.com/content/, 2018.

[55] VLSI Basic. *Clock Tree Synthesis.* https://vlsibasic.blogspot.com/2014/01/clock-tree-synthesis.html, 2018.

[56] Umair Hussaini. *Counters – Synchronous, Asynchronous, up, down Johnson ring counters.* https://technobyte.org/counters-up-down-synchronous-asynchronous/ /, 2018.

[57] Electronics Tutorial. *The Shift Register.* https://www.electronics-tutorials.ws/sequential/seq$_5$.*html*, 2018.

[58] arm mbed. *Using Hardware Timers.* https://os.mbed.com/users/4180$_1$/*notebook*/*using*−*hardware* − *timers*/, 2018.

[59] Cypress Infineon Semiconductors. *Edge Detection Circuit.* https://www.cypress.com/file/133046/download, 2019.

[60] elprocus. *Multiplexer and Demultiplexer : Types and Their Differences.* https://www.elprocus.com/what-is-multiplexer-and-demultiplexer-types-and-differences/, 2019.

[61] Nandland. *Using Modelsim for Simulation, for Beginners.* https://www.nandland.com/vhdl/tutorials/tutorial-modelsim-simulation-walkthrough.html.

[62] Deepak Kumar Tala. *Assertions In Verilog.* ASIC-World), 2014.

[63] Arthur Freitas publisher = Hyperstone GmbH). *Hardware/Software Co-Verification Using the SystemVerilog DPI.*

[64] Valentin Hiltl. *Generation of Properties for Design Verification from an Abstract FSM Model*, page 8. Technical University of Munich (TUM), 12.04.2019.

[65] Srinivas Devadas Jie-Hong Roland Jiang. *Logic Synthesis in a Nutshell.* Massachusetts Institute of Technology, Cambridge, Massachusetts), October 16, 2008.

[66] J. Vygen S. Held, S. Hougardy. *Chip Design.* Technical Univeristy of Munich (TUM), 2019.

[67] Edward J. McCluskey. *Built-In Self-Test Structures.* IEEE Design Test of Computers, April 1985.

[68] Stackexchange. *How is circuit topology chosen in analog designs?* electronics.stackexchange.com/questions/463491/how-is-circuit-topology-chosen-in-analog-designs.

[69] Allaboutcircuits. *SPICE Models.* www.allaboutcircuits.com/textbook/semiconductors/chpt-3/spice-models/.

[70] Igor L. Markov Jin Hu Andrew B. Kahng, Jens Lienig. *VLSI Physical Design : From Graph Partitioning to Timing Closure.* Springer, 9. February 2011.

[71] VLSI Guide. *Sign Off Checks.* https://www.vlsiguide.com/2019/02/sign-off-checks.html, 9. August 2019.

[72] L. R. Snowden. *Process Control Monitor to Device Data Correlation.* 45th ARFTG Conference Digest, 1995.

[73] C. S. Wang. *Reliability control monitor guideline of negative bias temperature instability for 0.13 /spl mu/m CMOS technology.* Proceedings of the 11th International Symposium on the Physical and Failure Analysis of Integrated Circuits. IPFA 2004, 2004.

[74] G. Bednarz D. Edwards, G. Heinen and W. Schroen. *Test Structure Methodology of IC Package Material Characterization.* IEEE Transactions on Components, Hybrids, and Manufacturing Technology, December 1983.

[75] Ken Shirriff. *Reverse-engineering the First FPGA Chip Xilinx XC2064.* semiwiki.com/fpga/290990-reverse-engineering-the-first-fpga-chip-xilinx-xc2064/, 09-16-2020.

[76] Stephen M. Trimberger. *Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology.* Proceedings of the IEEE, 2015.

[77] Priyabrata Biswas. *Introduction to FPGA and its Architecture.* https://towardsdatascience.com/introduction-to-fpga-and-its-architecture-20a62c14421c.

[78] Shahul Akthar. *FPGA Architecture.* allaboutfpga.com/fpga-architecture/.

[79] Priyabrata Biswas. *Introduction to FPGA and its Architecture.* towardsdatascience.com/introduction-to-fpga-and-its-architecture-20a62c14421c, 20 Nov 2019.

[80] Anysilicon. *What is a System on Chip (SoC)?* https://anysilicon.com/what-is-a-system-on-chip-soc/, 2015.

[81] Intel Corporation. *The Story of the Intel® 4004.* www.intel.de/content/www/de/de/history/museum-story-of-intel-4004.html.

[82] UC Berkeley. *Finite State Machines for Simple CPUs.* inst.eecs.berkeley.edu/ cs150/fa05/CLD$_{S}upplement/chapter$11/$chapter$11.doc3.$html$, 16/07/96.

[83] Reiner Hartenstein. *THE VON NEUMANN SYNDROME.* TU Kaiserslautern.

[84] Reiner Hartenstein. *The Expensive von Neumann Paradigm: is its Predominance still Tolerable?* hartenstein.de/Hartenstein-CPU-centric.pdf.

[85] Scott Rixner. *Stream Processor Architecture.* Rice University, 16/07/96.

[86] Kevin Deierling. *What Is a DPU?* https://blogs.nvidia.com/blog/2020/05/20/whats-a-dpu-data-processing-unit/, 16/07/96.

[87] Reiner Hartenstein. *Generalization of the Systolic Array.* http://www.fpl.uni-kl.de/staff/hartenstein/GeneralizationOfTheSystolicArray.pdf, Oct 2003.

[88] Abhishek Damle Kyle E. Berger and Jack Retcher. *Construction of a 4x4 Systolic Array.* www.a-damle.com/project$_{f}iles/systolic_{a}rray/systolic_{a}rray.pdf$.

[89] Mark Ilg. *Projectile Monte-Carlo Trajectory Analysis Using a Graphics Processing Unit.* https://arc.aiaa.org/doi/10.2514/6.2011-6266, 14 Jun 2012.

[90] Java Point. *Computer Architecture VS Computer Organization.* www.javatpoint.com/computer-architecture-vs-computer-organization.

[91] Geeksforgeeks. *Computer Architecture VS Computer Organization.* www.geeksforgeeks.org/difference-between-memory-based-and-register-based-addressing-modes/.

[92] University of Surabaya. *Accumulator (computing).* p2k.um-surabaya.ac.id/IT/en/3045-2942/accumulators$_3$23$_p$2$k - um - surabaya.html$.

[93] Itectec. *Electronic – early accumulator based machines.* itectec.com/electrical/electronic-early-accumulator-based-machines/.

[94] Geeksforgeeks. *Electronic – early accumulator based machines.* www.geeksforgeeks.org/stack-machine-in-computer-organisation/.

[95] Phil Koopman. *1.3 WHY ARE STACK MACHINES IMPORTANT?* users.ece.cmu.edu/ koopman/stack$_c$$omputers/sec1_3$.$html$.

[96] Stackoverflow. *Why is the JVM stack-based and the Dalvik VM register-based?* stackoverflow.com/questions/2719469/why-is-the-jvm-stack-based-and-the-dalvik-vm-register-based.

[97] Geeksforgeeks. *Introduction of General Register based CPU Organization.* www.geeksforgeeks.org/introduction-of-general-register-based-cpu-organization/.

[98] Chortle. *Turing Complete.* chortle.ccsu.edu/StructuredC/Chap01/struct01$_5$.$html$.

[99] Justin Rajewski. *Basic CPU Design.* https://alchitry.com/blogs/tutorials/basic-cpu, 2018.

[100] Techopedia. *Program Counter (PC).* www.techopedia.com/definition/13114/program-counter-pc, June 16, 2017.

[101] Wikipedia. *Link Register.* https://en.wikipedia.org/wiki/Link$_r$$egister$.

[102] Newbedev. *What actually is a shadow register?* newbedev.com/what-actually-is-a-shadow-register.

[103] Titanwolf. *What is a stack pointer used for in microprocessors?* www.titanwolf.org/Network/q/4f72680c-1009-4d31-bc76-b31b5b63d2ff/y.

[104] Career Ride. *Explain briefly the characteristics of the program invisible registers.* www.careerride.com/view/explain-briefly-the-characteristics-of-the-program-invisible-registers-intel-microprocessor-3134.aspx.

[105] Barry B. Brey. *Chapter – 2 The Microprocessor its Architecture.* www.byclb.com/TR/Tutorials/microprocessors/ch2$_1$.htm.

[106] Crystal Bedell. *big-endian and little-endian*. www.techtarget.com/searchnetworking/definition/big-endian-and-little-endian.

[107] Christian Märtin. *Multicore Processors: Challenges, Opportunities, Emerging Trends*. www.hs-augsburg.de/Binaries/Binary20964/Multicore-Embeddedfinal-revised.pdf, 2014.

[108] Osdev. *Interrupts*. wiki.osdev.org/Interrupts.

[109] Andre Dos Santos Oliveira. *Clock Synchronization For Modern Multiprocessors*. University of Porto, May 2015.

[110] Os Dev. *Context Switching*. https://wiki.osdev.org/Context$_S$*witching*.

[111] Os Dev. *GDT Tutorial*. https://wiki.osdev.org/GDT$_T$*utorial*.

[112] Arthur Chunqi Li. *What is GDT/LDT/IDT/TSS*. https://chunqili.blogspot.com/2013/06/what-is-gdtldtidttss.html.

[113] Virgil Bistriceanu. *3. Instruction set design*. http://www.cs.iit.edu/ virgil/cs470/Book/chapter3.pdf, Sep 6, 2011.

[114] Virgil Bistriceanu. *4. Addressing modes*. www.cs.iit.edu/ virgil/cs470/Book/chapter4.pdf, Sep 6, 2011.

[115] Igor Kholodov. *Modes of Memory Addressing on x86*. http://www.c-jump.com/CIS77/ASM/Memory/lecture.html.

[116] Ra'ul Rojas Margarita Esponda. *The RISC Concept - A Survey of Implementations*. www.inf.fu-berlin.de/lehre/WS94/RA/RISC-9.html, September 1991.

[117] Stephen St. Michael. *What Is a Microarchitecture? Understanding Processors and Register Files in an ARM Core*. www.allaboutcircuits.com/technical-articles/what-is-a-microarchitecture-processor-register-files-ARM-core/, February 07, 2019.

[118] Agner Fog. *The microarchitecture of Intel, AMD, and VIA CPUs An optimization guide for assembly programmers and compiler makers*. Technical University of Denmark, 2021-08-17.

[119] Wikipedia. *CPU cache*. en.wikipedia.org/wiki/CPU$_c$*ache*.

[120] Wikipedia. *Instruction pipelining*. en.wikipedia.org/wiki/Instruction$_p$*ipelining*.

[121] Wikipedia. *Branch Predictor*. en.wikipedia.org/wiki/Branch$_p$*redictor*.

[122] Wikipedia. *Out-of-order execution*. en.wikipedia.org/wiki/Out-of-order$_e$*xecution*.

[123] Wikipedia. *Speculative Execution*. en.wikipedia.org/wiki/Speculative$_e$*xecution*.

[124] Wikipedia. *Register renaming*. en.wikipedia.org/wiki/Register$_r$*enaming*.

[125] Alex Blewitt. *Understanding CPU Microarchitecture to Increase Performance*. www.infoq.com/presentations/microarchitecture-modern-cpu/, 2020.

[126] Dmitry Evtyushkin Nael Abu-Ghazaleh, Dmitry Ponomarev. *How the Spectre and Meltdown Hacks Really Worked*. IEEE Spectrum, 28 Feb 2019.

[127] Nica Latto. *What Are Meltdown and Spectre?* www.avast.com/c-meltdown-spectre, September 23, 2021.

[128] Stephen Smith. *ARM Processor Modes*. smist08.wordpress.com/2019/12/02/arm-processor-modes/.

[129] Dipl.-Ing. Thomas Spielauer. *x86 and x86-64 memory models*. www.tspi.at/2019/11/03/x86memmodels.html, 03 Nov 2019.

[130] Techopedia. *Microcode*. www.techopedia.com/definition/8332/microcode, January 4, 2017.

[131] Debian. *Microcode*. wiki.debian.org/Microcode, 14.06.2020.

[132] Alert Logic Staff. *A Clear Guide to Meltdown and Spectre Patches*. www.alertlogic.com/blog/a-clear-guide-to-meltdown-and-spectre-patches/, Mar 20, 2019.

[133] Scott Fulton III. *Arm processors: Everything you need to know now*. www.zdnet.com/article/arm-processors-everything-you-need-to-know-now/, March 30, 2021.

[134] Marc Fyrbiak Christian Kison Robert Gawlik Christof Paar Philipp Koppe, Benjamin Kollenda and Thorsten Holz. *Reverse Engineering x86 Processor Microcode*. Ruhr-Universitsat Bochum, 01.10.2019.

[135] Y. Voronenko F. Franchetti C. R. Berger, V. Arbatov and M. Püschel. *Real-time software implementation of an IEEE 802.11a baseband receiver on Intel multicore*. IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2011.

[136] Yevgen Voronenko Franz Franchetti and Markus Puschel. *FFT Program Generation for Shared Memory: SMP and Multicore*. SC 2006 Conference, Proceedings of the ACM/IEEE, December 2006.

[137] Wikipedia. *Amdahl's law*. en.wikipedia.org/wiki/Amdahl

[138] Wikipedia. *Gustafson's law*. en.wikipedia.org/wiki/Gustafson

[139] Prof. Taek Kwon. *Sun and Ni's Law.* www.d.umn.edu/ tkwon/course/5315/HW/MultiprocessorLaws.pdf.

[140] Rick Merritt. *CPU designers debate multi-core future.* www.edn.com/cpu-designers-debate-multi-core-future/, February 6, 2008.

[141] NXP. *Embedded Multicore: An Introduction.* www.nxp.com/files-static/32bit/doc/ref$_m$*anual*/*EMBMCRM.pdf*/, 2009.

[142] TMurgent Technologies. *White Paper Processor Affinity Multiple CPU Scheduling.* White Paper TMurgent Technologies, November 3, 2003.

[143] Scharon Harding. *What Is Hyper-Threading? A Basic Definition.* www.tomshardware.com/reviews/hyper-threading-intel-definition,5746.html, August 23, 2018.

[144] Lawrence Stewart. *How does multi-core CPU work? In terms of "communication". How do they "communicate" with each other. Do one core control others or they use another method for connection?* www.quora.com/How-does-multi-core-CPU-work-In-terms-of-communication-How-do-they-communicate-with-each-other-Do-one-core-control-others-or-they-use-another-method-for-connection.

[145] Naim Dahnoun. *Multicore DSP From Algorithms to Real-time Implementation on the TMS320C66x SoC. 9.Inter-Processor Communication (IPC).* 2018 John Wiley Sons Ltd.

[146] Erich Boleyn. *Intel MP Specification compatibility.* www.uruk.org/mps/, 30.04.1997.

[147] Konrad Eisele. *Design of a Memory Management Unit for System-on-a-Chip Platform "LEON".* University of Stuttgart, 14.11.2002.

[148] Milan Milenkovic. *Microprocessor Memory Management Units.* IEEE MicroVolume 10 Issue 2, 01 March 1990.

[149] Reddit. *Could a GPU alone run a complete system?* www.reddit.com/r/hardware/comments/10bph4/could$_{ag}$*pu*$_a$*lone*$_r$*un*$_{ac}$*omplete*$_s$*ystem*/.

[150] Wikibooks. *Microprocessor Design/Performance Metrics.* en.wikibooks.org/wiki/Microprocessor$_D$*esign*/*Performance*$_M$*etrics*.

[151] Thiruvengadam Vijayaraghavan Emily Blem, Jaikrishnan Menon and Karthikeyan Sankaralingam. *ISA Wars: Understanding the Relevance of ISA being RISC or CISC to Performance, Power, and Energy on Modern Architectures.* CM Trans. Comput. Syst. 33, 1, Article 3.

[152] David J. Lilja. *Measuring Computer Performance - A Parctitioners Guide.* Cambridge University Press 2004.

[153] Roman Plyaskin. *Fast And Accurate Performance Simulation Of Out-Of-Orderprocessing Cores In Embedded Systems*. Technical University of Munich - Lehrstuhl für Integrierte Systeme, 23.06.2014.

[154] Reinhard Wilhelm. *Formal Analysis of Processor Timing Models*. Conference Paper in Lecture Notes in Computer Science, April 2004.

[155] K. Lu D. Mueller-Gritschneder and U. Schlichtmanns. *Control-Flow-Driven Source Level Timing Annotation for Embedded Software Models on Transaction Level*. 2011 14th Euromicro Conference on Digital System Design, 2011.

[156] Wikiwand. *Self-modifying code*. www.wikiwand.com/en/Self-modifying$_c$*ode*.

[157] 101Computing. *Self-modifying code in LMC*. www.101computing.net/self-modifying-code-in-lmc/, January 6, 2021.

[158] 101Computing. *Harvard or von Neumann?* www.101computing.net/self-modifying-code-in-lmc/, January 6, 2021.

[159] Alan Creak. *Memory Models*. www.cs.auckland.ac.nz/ alan/courses/os/book/4 How 02 memorymod.pdf, 2007 December.

[160] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *The Abstraction: Address Spaces*. pages.cs.wisc.edu/ remzi/OSTEP/vm-intro.pdf, August 15, 2013.

[161] Superuser. *Understanding "Flat memory model" and "Segmented memory model"*. superuser.com/questions/318804/understanding-flat-memory-model-and-segmented-memory-model.

[162] TutorialsPoint. *Assembly - Memory Segments*. www.tutorialspoint.com/assembly$_p$*rogramming*/*assembly*$_m$*emory*$_s$*egments.htm*.

[163] Aticleworld. *Memory Layout of C program*. aticleworld.com/memory-layout-of-c-program/.

[164] Boehm H.J. Adve S.V. *Memory Models. In: Padua D. (eds) Encyclopedia of Parallel Computing*. Springer, Boston, MA.

[165] Wikipedia. *Programming Models*. en.wikipedia.org/wiki/Programming$_m$*odel*.

[166] Stackoverflow. *Threads: Why must all user threads be mapped to a kernel thread?* stackoverflow.com/questions/14791278/threads-why-must-all-user-threads-be-mapped-to-a-kernel-thread.

[167] Stackoverflow. *C++11 introduced a standardized memory model. What does it mean? And how is it going to affect C++ programming?* stackoverflow.com/questions/6319146/c11-introduced-a-standardized-memory-model-what-does-it-mean-and-how-is-it-g.

[168] eCos. *Hardware Abstraction Layer.* http://www.ecos.sourceware.org/docs-1.3.1/ref/ecos-ref.b.html, 2018.

[169] stackexchange. *Bock and Character Device.* https://unix.stackexchange.com, 2019.

[170] Constantin Sarbu. *Operational Profiling of OS Drivers.* Vom Fachbereich Informatik der Technischen Universit at Darmstad, 25. May 2009.

[171] David R.O'Hallaron Randal E.Bryant. *Computer Systems: A Programmer's Perspective.* Pearson Education Inc., 2016.

[172] Harvard University Press. *High Level program Abstraction.* https://news.harvard.edu/gazette/story/2014/12/grace-hopper-computing-pioneer/, 2018.

[173] James W. Grenning. *Test-Driven Development for Embedded C.* Pragmatic Bookshelf, 2011.

[174] Enrique I. Oviedo. *Control flow, Data flow and Program Complexity.* Software engineering metrics I: measures and validations, 04 October 1993.

[175] Chetan Giridhar. *Learning Python Design Patterns - Second Edition.* Packt, February 2016.

[176] Niklas Gruhn. *What makes a programming language Turing complete?* dev.to/gruhn/what-makes-a-programming-language-turing-complete-58fl, Jan 29, 2020.

[177] Bryce Adelstein Lelbach. *The C++ Execution Model.* Nividia Inc., 2019.

[178] Kin Lane. *Intro to APIs: What Is an API?* Postman Blog, October 5, 2020.

[179] Genera Codice. *Difference between API and ABI.* www.generacodice.com/en/articolo/942960/difference-between-api-and-abi, 2019.

[180] *Application binary interface.* en.wikipedia.org/wiki/Application$_b inary_i nter face$.

[181] ZhouZhuo. *Difference between API and ABI.* stackoverflow.com/questions/3784389/difference-between-api-and-abi/3784724, Oct 21 2017.

[182] GNU-GCC. *ABI Policy and Guidelines.* gcc.gnu.org/onlinedocs/libstdc++/manual/abi.html.

[183] Ray Kinsella Thomas Monjalon. *Why is ABI Stability Important?* www.dpdk.org/blog/2019/10/10/why-is-abi-stability-important/, October 10, 2019.

[184] Techopedia. *Software Library.* www.techopedia.com/definition/3828/software-library, December 5, 2016.

[185] Ulrich Drepper. *How To Write Shared Libraries.* Ulrich Drepper, December 10, 2011.

[186] Niall Cooling. *Static and Dynamic Libraries on Linux.* blog.feabhas.com/2014/04/static-and-dynamic-libraries-on-linux/, April 4, 2014.

[187] Segger. *C runtime library.* wiki.segger.com/C$_r untime_l ibrary, 4November$2019.

[188] Le Trung Thang. *Runtime Library in C/C++.* letrungthang.wordpress.com/2011/04/12/runtime-library-in-cc/, April 12, 2011.

[189] Micro-Controller Labs. *What is Microcontrollers startup file – Understand its various Functions.* microcontrollerslab.com/microcontrollers-startup-file-arm-cortex-m4-mcu/.

[190] Phillip Johnston. *A General Overview of What Happens Before main().* embeddedartistry.com/blog/2019/04/08/a-general-overview-of-what-happens-before-main/, 8 April 2019.

[191] Stackoverflow. *Difference between C/C++ Runtime Library and C/C++ Standard Library.* stackoverflow.com/questions/424549/difference-between-c-c-runtime-library-and-c-c-standard-library.

[192] Keshav Pingali Micah Beck, Richard Johnson. *From Control Flow to Dataflow.* Cornell University, 1989.

[193] Muppetlabs. *A Whirlwind Tutorial on Creating Really Teensy ELF Executables for Linux.* www.muppetlabs.com/ breadbox/software/tiny/teensy.html.

[194] Zuga. *C++ - What is a translation unit?* zuga.net/articles/cpp-what-is-a-translation-unit/.

[195] Geeksforgeeks. *Basic Blocks in Compiler Design.* www.geeksforgeeks.org/basic-blocks-in-compiler-design/, 03 Jul, 2020.

[196] Scott A. Mahlke Wen-mei W. Hwu. *The Super block: An Effective Technique for VLIW and Superscalar Compilation.* Journal of Sup ercomputing, 1993, 1993.

[197] Scott A. Mahlke. *Effective Compiler Support for Predicated Execution Using the Hyperblock.* Proceedings the 25th Annual International Symposium on Microarchitecture MICRO 25, 1-4 Dec. 1992.

[198] Linda Torczon Keith D. Cooper. *Engineering a Compiler (Second Edition).* 2011 Elsevier Inc, 2012.

[199] Henrik Theiling. *Control Flow Graphs for Real-Time System Analysis.* University of Saarland, 2002.

[200] Marilyn Wolf. *Computers as Components Principles of Embedded Computing System Design.* The Morgan Kaufmann Series, 2017.

[201] Claire Le Goues Jonathan Aldrich and Rohan Padhye. *Program Analysis.* Jonathan Aldrich, April 13, 2021.

[202] Stackexchange. *What is the difference between control flow graph interprocedural control flow graph?* cs.stackexchange.com/questions/7054/what-is-the-difference-between-control-flow-graph-interprocedural-control-flow.

[203] Newbedev. *Practical differences between control flow graph and call (flow?) graph?* newbedev.com/practical-differences-between-control-flow-graph-and-call-flow-graph.

[204] Galbox. *Call Graph for Mac.* galbox.weebly.com/call-graph-for-mac.html.

[205] Wikipedia. *Source-code annotation.* en.wikipedia.org.

[206] Dmitrijs Zaparanuks Matthias Hauswirth. *Algorithmic Profiling*, pages 67–76. In Proc. PLDI, 2012.

[207] Hanne Nielsen Flemming Nielsen and Chris Hankin. *Principles of Program Analysis.* Springer, 2005.

[208] Amitabha Sanyal Uday Khedker and Bageshri Karkara. *Data Flow Analysis: Theory and Practice.* CRC Press, 2009.

[209] C. Baier and J-P. Katoen. *Principles of Model Checking.* MIT Press, 2007.

[210] Elise Hewett. *A Survey of Static and Dynamic Analyzer Tools.* IEEExplore, 2008.

[211] Cole Schlesinger David Walker Benjamin Zorn Nikhil Swamy, Karthik Pattabiraman. *Modular Protections against Non-control Data Attacks.* IEEE 24th Computer Security Foundations Symposium, 27-29 June 2011.

[212] Xiangqun Chen Hong Mei Lin Yan, Yao Guo. *A Study on Power Side Channels on Mobile Devices.* arXiv, 2015.

[213] Martin Drahanský Boris Procházka, Tomáš Vojnar. *Hijacking the Linux Kernel.* Brno University of Technology, 2010.

[214] Acutenix. *Directory Traversal Attacks.* www.acunetix.com/websitesecurity/directory-traversal/, 2021.

[215] Aurélien Francillon. *Attacking and Protecting Constrained Embedded Systems from Control Flow Attacks.* Institut Polytechnique De Grenoble, 26 Nov 2010.

[216] Aurélien Francillon. *Code Injection Attacks on Harvard-Architecture Devices*. Institut Polytechnique De Grenoble, October 27–31, 2008.

[217] Christian Ferdinand Reinhold Heckmann. *Worst-Case Execution Time Prediction by Static Program Analysis*. AbsInt Angewandte Informatik GmbH.

[218] Stefan Bygde. *Parametric WCET Analysis*. Mälardalen University Press Dissertations, 2013.

[219] Codegrip. *A Simple Understanding of Code Complexity*. www.codegrip.tech/productivity/a-simple-understanding-of-code-complexity/.

[220] Advanced Data Controls Corp. *Compiler Driver*. adac.jp/eng/support/ghs/ccvxx.pdf.

[221] Stackoverflow. *Difference between compiler and compiler driver in LLVM?* stackoverflow.com/questions/32595607/difference-between-compiler-and-compiler-driver-in-llvm.

[222] Zhuanlan Zhihu. *Compiler Driver And Cross Compilation*. zhuanlan.zhihu.com/p/360482939.

[223] Leonidas Fegaras. *CSE 5317/4305: Design and Construction of Compilers*. lambda.uta.edu/cse5317/notes/node4.html, 2015-01-20.

[224] Vishal Kasliwal and Andrey Vladimirov. *A Performance-Based Comparison Of C/C++ Compilers*. Colfax International, November 19, 2017.

[225] Wikipedia. *Compiler*. en.wikipedia.org/wiki/Compiler.

[226] Leonidas Fegaras. *CSE 5317/4305: Design and Construction of Compilers*. lambda.uta.edu/cse5317/notes/node5.html, 2015-01-20.

[227] Kenneth C.Louden. *Compiler Construction Priniciples and Practice*. csunplugged.files.wordpress.com/2012/12/compiler-construction-principles-and-practice-k-c-louden-pws-1997-cmp-2002-592s.pdf, 1997.

[228] Eli Bendersky. *Load-time relocation of shared libraries*. eli.thegreenplace.net/2011/08/25/load-time-relocation-of-shared-libraries/, August 25, 2011.

[229] Stackoverflow. *Difference between position dependent and position independent code?* stackoverflow.com/questions/59691349/difference-between-position-dependent-and-position-independent-code, January 11, 20202.

[230] Geeksforgeeks. *Single pass, Two pass, and Multi pass Compilers.* www.geeksforgeeks.org/single-pass-two-pass-and-multi-pass-compilers/, 21 Nov, 2019.

[231] Oracle Corporation. *Chapter 7 Object File Format.* docs.oracle.com/cd/E19683-01/817-3677/chapter6-46512/index.html, 2010.

[232] Wiki Osdev. *Executable Formats.* wiki.osdev.org/Executable$_{F}ormats, 21July2016.$

[233] retro11. *2.11BSD431.* www.retro11.de/ouxr/211bsd/usr/man/cat5/a.out.0.html, January 9, 1994.

[234] Microsoft. *PE Format.* docs.microsoft.com/en-us/windows/win32/debug/pe-format, 03.31.2021.

[235] Alchetron. *COM File.* alchetron.com/COM-file, Feb 11, 2018.

[236] Owlapps. *Executable and Linkable Format.* next.owlapps.net/owlapps$_{a}pps/articles?id =$ $9914lang = en, 2012.$

[237] newbedev. *What's the difference of section and segment in ELF file format.* newbedev.com/what-s-the-difference-of-section-and-segment-in-elf-file-format, 2021.

[238] Govind Mukundan. *Analyzing the Linker Map file with a little help from the ELF and the DWARF.* Embeddedrelated, December 27, 2015.

[239] Microchip. *AVR32795: Using the GNU Linker Scripts on AVR UC3 Devices.* Microchip Application Notes.

[240] Stackoverflow. *Why do we need compiling and linking separately?* stackoverflow.com/questions/19225453/why-do-we-need-compiling-and-linking-separately/19227350.

[241] Microchip. *AVR910: In-System Programming.* Microchip Applicaton Notes.

[242] Microchip Corporation. *Atmel AVR2054: Serial Bootloader User Guide.* ww1.microchip.com/downloads/en/AppNotes/Atmel-8390-WIRELESS-AVR2054-Serial-Bootloader-User-Guide$_{A}pplication - Note.pdf.$

[243] FreeRTOS API Reference. *Task Creation.* web.ist.utl.pt/ ist11993/FRTOS-API/group$_{tasks.html,Apr}$192011.

[244] nachoparker. *The real power of Linux executables.* ownyourbits.com/2018/05/23/the-real-power-of-linux-executables/, May 23, 2018.

[245] Simon Peyton Jones Andrey Mokhov, Neil Mitchell. *Build Systems à la Carte*. Proc. ACM Program.Lang. 2, 2018.

[246] Robert Mecklenburg. *Managing Projects with GNU Make, 3rd Edition*. O'Reilly, November 2004.

[247] Peter Smith. *Software Build Systems Principles and Experience*. Addison-Wesley, 2011.

[248] Dirk Thomas. *A universal build tool*. design.ros2.org/articles/build$_t$*ool.html*.

[249] Chris Coleman. *Reproducible Firmware Builds*. interrupt.memfault.com/blog/reproducible-firmware-builds, 11 Dec 2019.

[250] Yocto Project. *Reproducible Builds*. wiki.yoctoproject.org, 9 February 2021.

[251] Ravi Verma. *Software Build Process – All You Need to Know!* scmquest.com/software-build-knowledge/, July 13, 2020.

[252] Stackoverfloe. *Definition of debugging, profiling and tracing*. Stackoverflow, 16 June 2021.

[253] Sebastian Steinhorst. *Lecture notes in Internet of Things*, chapter 3, page 6. Technical University of Munich (TUM), Munich, 2020.

[254] Florian Erdinger. *Analog to Digital Conversion*, chapter 14, page 6. Universitat Heidelburg, Heidelburg, 2020.

[255] Jorg Vollrath. *Summary of Interface Electronics*, chapter 14s, page 6. TH Nurmburg, Munich, 2020.

[256] B. E. Jonsson. *A survey of ADC surveys*, chapter 0, page 1. converterpassion.wordpress.com/2012/08/28/a-survey-of-adc-surveys/, Munich, August 28, 2012.

[257] Richard Lyons Randy Yates. *Reducing ADC Quantization Noise*, chapter 0, page 1. Microwaves and RF, USA, 17 June 2005.

[258] Stefan Wabnik. *Different Quantisation Noise Shaping Methods For Predictive Audio Coding*, page 2. Fraunhofer Institute for Digital Media Technology, Ilmenau, Germany, Ilmenau, 14 - 19 May 2006.

[259] Scott Campbell. *Basics of UART Communication*, page 1. https://www.circuitbasics.com/basics-uart-communication/, USA, 02 June, 2016.

[260] RFwireless World. *UART vs SPI vs I2C — Difference between UART, SPI and I2C.* https://www.rfwireless-world.com/Terminology/UART-vs-SPI-vs-I2C.html, USA.

[261] Wikipedia. *Serial Peripheral Interface.* USA.

[262] Mark Hughes. *Back to Basics: SPI (Serial Peripheral Interface).* https://www.allaboutcircuits.com/technical-articles/spi-serial-peripheral-interface/, USA, 13 February, 2017 2007.

[263] Ray Weiss. *CoreConnect: The On-Chip Bus System*, page 1. https://www.electronicdesign.com/technologies/embedded-revolution/article/21766469/coreconnect-the-onchip-bus-system, USA.

[264] Mohandeep Sharma and Dilip Kumar. *Wishbone Bus Architecture – A Survey And Comparison*, page 110. nternational Journal of VLSI design Communication Systems (VLSICS) Vol.3, India, 2 April 2012.

[265] Mile Stojcev Milica Miti. *An Overview of On-Chip Buses*, pages 405–428. Electronics and Energetics 2006 Volume 19, Issue 3, Serbia, January 2006.

[266] Arthur. *Avalon Bus Architecture Overview*, page 1. https://www.element14.com/community/blogs/spice$_e$ye/2008/11/26/$avalon - bus - architecture - overview, USA, 26November$, 2008.

[267] Stephen St. Michael. *The Advanced Microcontroller Bus Architecture: An Introduction*, page 1. All About Circuits, USA, 02 June, 2019.

[268] Digital Cerebrum. *Bit Banging*, page 1. digitalcerebrum.wordpress.com/random-tech-info/bit-banging/, USA.

[269] Stacey Peterson. *RAM (Random Access Memory)*, page 1. searchstorage.techtarget.com/definition/RAM-random-access-memory, USA.

[270] Geeksforgeeks. *Difference between Volatile Memory and Non-Volatile Memory*, page 1. www.geeksforgeeks.org/difference-between-volatile-memory-and-non-volatile-memory/, USA, 28 June 2020.

[271] Intel Altera. *Converting Memory from Asynchronous to Synchronous for Stratix Stratix GX Designs*, page 1. Altera Corporation, USA.

[272] WikiChip. *Static Random-Access Memory (SRAM).* https://en.wikichip.org/wiki/static$_r$andom - access$_m$emory, USA.

[273] ATP. *Understanding RAM and DRAM Computer Memory Types.* https://www.atpinc.com/blog/computer-memory-types-dram-ram-module, USA, 25 August 2018.

[274] Bauman National Library. *DRAM (Dynamic Random Access Memory).* https://en.bmstu.wiki/DRAM$_(Dynamic_Random_Access_Memory)$, *Russia.*

[275] Vskills. *DRAM.* https://www.vskills.in/certification/tutorial/dram/, India.

[276] Matthew Martin. *SRAM vs DRAM: Know the Difference.* https://www.guru99.com/sram-vs-dram-difference.html, USA, August 27, 2021.

[277] Texas Instruments. *FRAM FAQs*, page 1. Texas Instruments TI, USA.

[278] Garry Kranz. *Flash Memory*, page 1. searchstorage.techtarget.com/definition/flash-memory, USA.

[279] CPU Schack. *EPROM*, page 1. https://www.cpushack.com/EPROM.html, USA, 19 May 2006.

[280] Microsemi. *Software Installation and Licensing Guide Libero SoC.* coredocs.s3.amazonaws.com/Libero, 2020.

[281] Microsemi. *Synopsys FPGA Synthesis Synplify Pro for Microsemi Edition Attribute Reference.* Microsemi Corporation, May 2015.

[282] Benjamin F. Harding R.C. Cofer. *Rapid System Prototyping with FPGAs.* Elsevier Inc., 2006.

[283] Microsemi. *SmartDesign User Guide.* coredocs.s3.amazonaws.com/Libero, 2021.

[284] EE354L Introduction to Digital Circuits. *Timing Analysis and Timing Constraints.* www-classes.usc.edu/engr/ee-s/254/ee254l$_l ab_m anual/Timing/handout_f iles/ee254l_t iming.pdf$, 10/13/20.

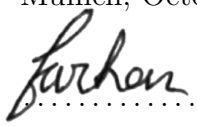
[285] Microsemi. *PolarFire® FPGA Product Overview.* ww1.microchip.com/downloads/en/DeviceDoc/PolarFire FPGA Product Overview.pdf, 2020.

[286] FTDI Chip. *User Guide for FTDI FT PROG Utility.* www.ftdichip.com/Support/Documents/AppNotes, 2016-04-06.

Confirmation

Herewith I, Muhammad Farhan, confirm that I independently prepared this work. No further references or auxiliary means except those declared in this document have been used.

Munich, October 20, 2020



.....

Muhammad Farhan