Contents

List of Figures

List of Tables

1 Introduction

This chapter will introduce the basics of a Central Processing Unit (CPU), SoC and widely used types of computer architectures in computer systems, before diving into the design elements of a SoC built around a RISC-V core.

1.1 Central Processing Unit

The CPU is a hardware unit consisting of a control and processor unit, responsible for processing data based on a predefined set of instructions known as an ISA. In simple terms, it is analogous to a human brain, in the sense that it operates the computer system by providing necessary control signals to supporting hardware units to convert raw-data into useful information. A *Microprocessor* is an Integrated-Circuit (IC) that integrates all the functional blocks of a CPU onto a single chip, resulting in smaller footprint and higher achievable clock frequencies.

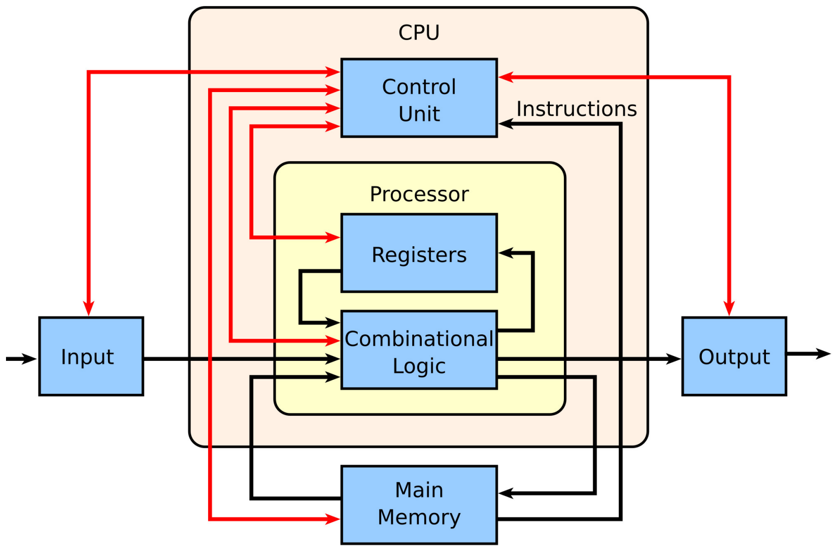


Figure 1.1: Block diagram of a basic computer with uniprocessor CPU. Black lines indicate data flow, whereas red lines indicate control flow. Arrows indicate the direction of flow. (By User:Lambtron - File:ABasicComputer.gif, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=123099855)

1.2 System-On-Chip

A SoC combines the CPU with memory, Memory-Management-Unit (MMU) and appropriate functional units such as a *Bus-Interconnect*, *Radio-Modem*, *Graphical-Processing-Unit* and other application specific hardware. Thus, providing a higher level of system integration and abstraction, resulting in better system design and performance in multi-system designs.

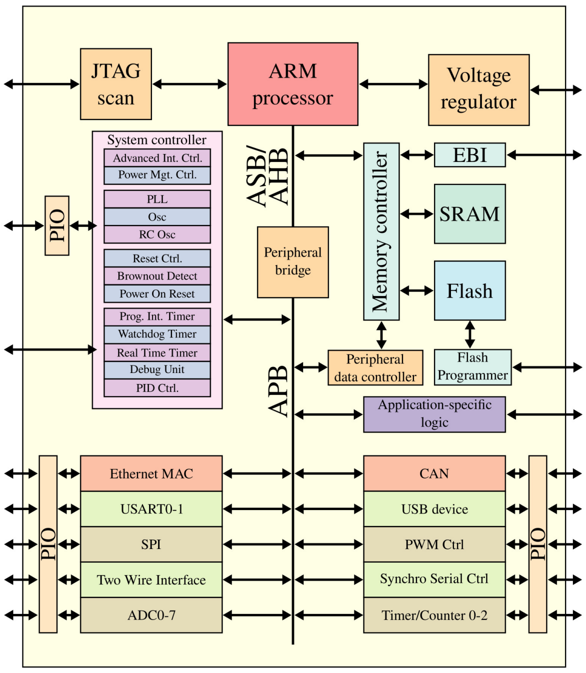


Figure 1.2: Block diagram of an ARM based SoC. (By en:User:Cburnett - Own work in Inkscape based on en:Image:ARMSoCBlockDiagram.gif, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=2866881)

1.3 Computer Architecture

A computer architecture describes the relation between different modules of a computer system and how they operate in tandem to achieve some goal. There are three widely used computer architectures, namely, *Von-Neumann*, *Harvard* and *Super-Harvard* also commonly known as SHARC.

1.3.1 Von-Neumann Architecture

The *Von-Neumann* consists of the CPU and a single memory unit to hold both data and instructions. This gives the advantage of design simplicity, resulting in lower *Design-Complexity*, *Area* and *Power* consumption at the expense of *Speed* due to its limited memory access bandwidth, i.e., the CPU can fetch either data or instruction from memory at any given time.

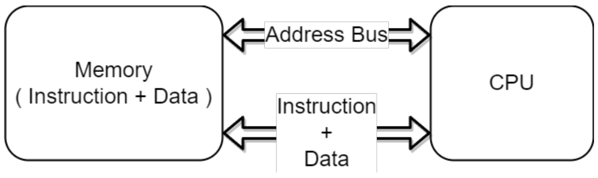


Figure 1.3: Von-Neumann architecture with shared *Instruction* and *Data* bus.

1.3.2 Harvard Architecture

To provide wider memory bandwidth, the *Harvard* architecture provides separate *Instruction* and *Data* memory bus, so that the CPU can access both data and instruction at the same time. Thus, resulting in higher *Design-Complexity*, *Area*, *Power* and *Speed* compared to *Von-Neumann* architecture.

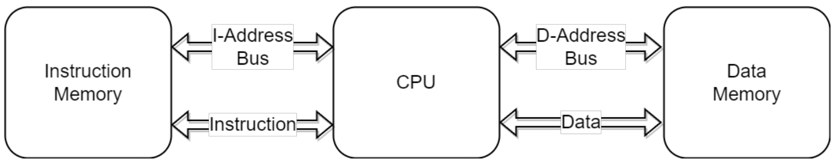


Figure 1.4: Harvard architecture with separate *Instruction* and *Data* bus.

1.3.3 Super-Harvard Architecture

Also commonly known as *SHARC*, this architecture improves upon the standard *Harvard* architecture by utilizing an *Instruction-Cache* within the CPU to reduce memory access operations for frequently used instructions such as loops. In-addition, it also provides an *I/O-Controller* that can perform *DMA* to data memory to reduce the burden of external memory access by the CPU. Thus vastly improving performance by improving throughput of the system. A common use-case of *SHARC* is in Digital-Signal-Processors (DSPs).

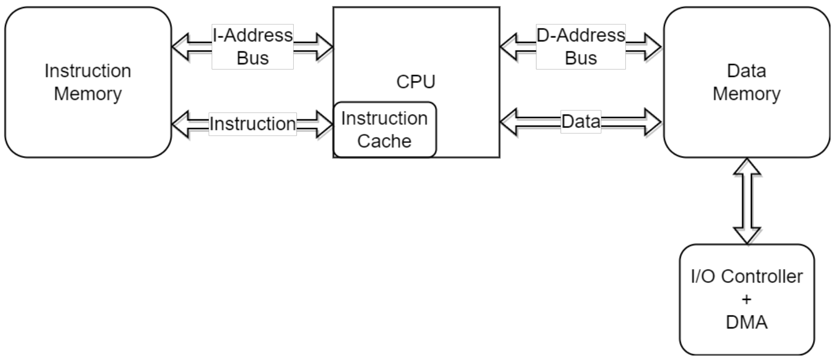


Figure 1.5: Super-Harvard architecture with instruction cache and DMA in addition to separate *Instruction* and *Data* bus.

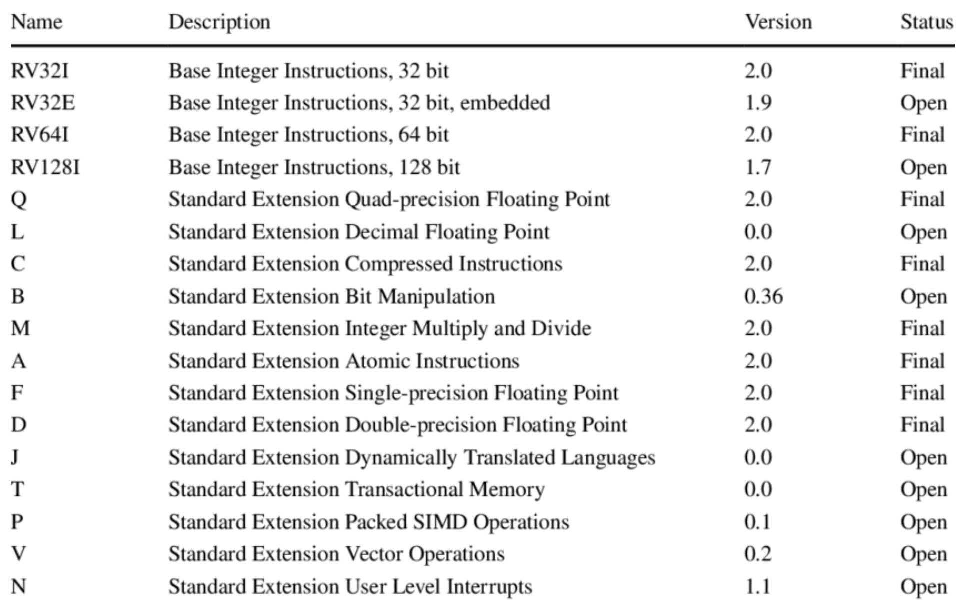
1.4 Instruction-Set-Architecture

Any program when compiled is broken down and represented by a pre-defined set of instructions, which take advantage of the underlying hardware architecture discussed in section 1.3. An ISA provides a well documented structure for such base instructions, that a program may be broken down into, to fit the target hardware architecture. Hence one could think of the ISA as an abstract model of the computer, which generally defines how the software controls the CPU.

The ISA is widely classified into two major types namely, Reduced Instruction Set Computing (RISC) and Complex Instruction Set Computing (CISC). They differ in the way the two ISAs represent a software program when compiled. In simple terms, CISC breaks down a program into complex instructions which may require complex hardware or multi-cycle operation to perform a single instruction in hardware, whereas, RISC represents the program using the most frequently used simple operations, with more complex routines broken down into sub-routines made up of these simple instructions.

1.4.1 Reduced Instruction Set Computing: RISC-V

RISC-V [2, 1] is an open standard ISA based on established RISC principles. Unlike most other ISA designs, RISC-V is provided under royalty-free open-source licenses. RISC-V is a register-register architecture (source and destination must be registers) in contrast to the CISC which is a register-memory architecture (one of source or destination can be from memory). A notable feature of RISC-V is the carefully placed choice of instruction bit-fields to facilitate easier use of multiplexers in CPUs. The ISA is designed to support a wide range of uses. The base instruction set has a fixed length of 32-bit naturally aligned instructions, with added extensions (figure 1.6) which support designs from small embedded systems to supercomputers with vector processors, and warehouse-scale parallel computers.



Name	Description	Version	Status
RV32I	Base Integer Instructions, 32 bit	2.0	Final
RV32E	Base Integer Instructions, 32 bit, embedded	1.9	Open
RV64I	Base Integer Instructions, 64 bit	2.0	Final
RV128I	Base Integer Instructions, 128 bit	1.7	Open
Q	Standard Extension Quad-precision Floating Point	2.0	Final
L	Standard Extension Decimal Floating Point	0.0	Open
C	Standard Extension Compressed Instructions	2.0	Final
B	Standard Extension Bit Manipulation	0.36	Open
M	Standard Extension Integer Multiply and Divide	2.0	Final
A	Standard Extension Atomic Instructions	2.0	Final
F	Standard Extension Single-precision Floating Point	2.0	Final
D	Standard Extension Double-precision Floating Point	2.0	Final
J	Standard Extension Dynamically Translated Languages	0.0	Open
T	Standard Extension Transactional Memory	0.0	Open
P	Standard Extension Packed SIMD Operations	0.1	Open
V	Standard Extension Vector Operations	0.2	Open
N	Standard Extension User Level Interrupts	1.1	Open

Figure 1.6: RISC-V base ISA and its extensions [7].

1.4.2 Complex Instruction Set Computing: Intel x86

In contrast to RISC, CISC is an ISA that can execute several low-level operations (memory load, store and arithmatic) in a single instruction. It was introduced by *Intel Corporation* when the earliest computers focused on enhancing CPU speed by minimizing the number of instructions per program. This objective was achieved by combining multiple simple commands into one complex instruction. Hence, in CISC architectures, the CPU is required to perform additional work to decode and execute a single instruction. As the decoding and execution logic of complex instruction takes more time, resulting in lower clock speeds.

1.5 Task Description

The aim is to implement the *Dobby* [11] microcontroller, which is a RISC-V ISA based processor aimed at general processing tasks, optimized either in *Area*, *Speed* or *Power*. As part of this task, The following features were implemented and verified:

- RISC-V ISA RV32IM [Base instruction format(I) and Extension M which includes multiplication and division].
- 16 kB built-in program memory (PRAM).
- 2 general purpose interrupts, high-speed bus interface for peripheral device and/or external memory attachment.
- PRAM initialization after reset.
- A memory-controller giving the processor core and init-controller access to on-chip memory and the external bus interface based on the memory layout.

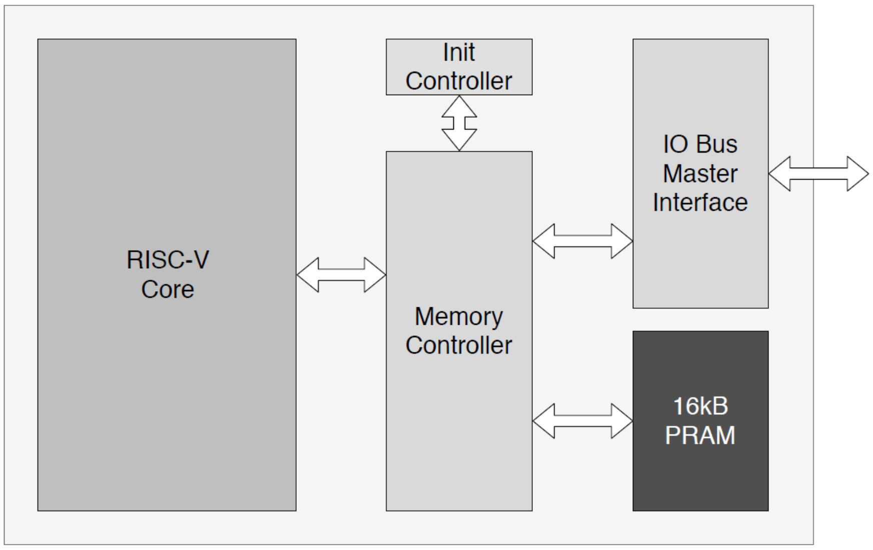


Figure 1.7: Dobby SoC base architecture [11].

2 Background and Related Work

In this chapter we will explore what a uARCH is and some of the most widely used processor uARCHs and briefly discuss the components of a uARCH. In-addition, a brief about *Comet-RISCV* and *PicoRV32*, RISC-V processor architectures will be provided, whcih were used to study and understand RISC-V ISA.

2.1 Micro-Architecture

A uARCH describes the arrangement of *digital-blocks/modules* such as *ALU*, *Data-path*, *Registers/Memory* and *Control-Unit*, to implement a given ISA in a processor. An ISA may be implemented using different uARCHs depending on target design goals (specifications) and fabrication technology (technology node). The complete implementation of an ISA on a uARCH results in the *Computer-Architecture* described previously in section 1.3. According to [9, 3], different uARCHs can be broadly classified as follows:

1. **Single-Cycle** uARCH executes an instruction in a single *clock cycle*. Thus, it does not need intermediate registers to store operational state, making it simple to design and power efficient. However, due to its simplicity, the *critical path* (cycle time) is determined by the execution time of the longest instruction.
2. **Multi-Cycle** uARCH executes an instruction in multiple, short cycles, with the number of cycles per instructions determined by the complexity of the instruction. Moreover, uARCH reduces the hardware cost by reusing expensive hardware blocks such as adders and memories, at the expense of adding additional registers to hold intermediary states/results. Therefore, multi-cycle execution of instructions reduce the *critical path* of longer instructions.
3. **Pipelined** uARCH splits the execution of an instruction into multiple phases, allowing different instructions to be processed in different phases simultaneously, leading to improved Instruction-Level-Parallelism (ILP). Each phase/stage may be implemented as *Single-Cycle* or *Multi-Cycle* depending on the complexity of the *control-unit* and other design goals.
4. **Out-of-Order Exectuion** uARCH can execute instructions out-of program order, resulting in higher ILP at the cost of hardware complexity compared to in-order execution architectures.

5. **Super-Scalar** uARCH can execute multiple instructions in a single pipeline stage, enabling the processor to achieve a throughput greater than 1 instruction per cycle for any program.

2.2 Building Blocks of a Micro-Architecture

As previously discussed in section 2.1, a uARCH is built using modules that have specific functionalities and structure. Shown in figure 2.1 is a general block structure of a uARCH. In this section, we shall elaborate on some of the most important and common modules, and their impact on uARCH design.

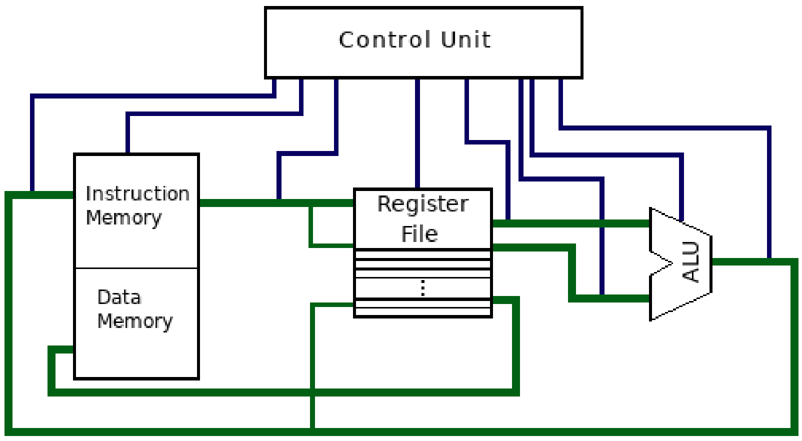


Figure 2.1: A overly simplified uARCH with, *data-path* shown in green and *control-path* shown in dark blue (By: Stephen St. Michael - Technical Article on Microarchitecture in https://www.allaboutcircuits.com/).

2.2.1 Arithmetic-Logic-Unit

As the name suggests, an ALU is responsible for all arithmetic and logical operations carried out in the processor. Any instruction that requires some form of arithmetic or logical operation, is sent to the ALU along with the corresponding operands, so that the ALU may process the data and produce the required result.

2.2.2 Register-File

The register file is a collection of registers that are used to hold operands and their results from the ALU. The register file has the fastest access time compared to other memory structures in the hierarchy and therefore is used to store values that are currently required by the processor for operation.

2.2.3 Data-path

The data-path acts as the highway that connects all the functional units of the processor together, for transfer of data amongst them. All the operands required by the ALU are brought in through the data-path, and the result of the ALU is once again carried by the data-path to its destination.

2.2.4 Control-Unit

The control-unit is responsible for providing timely signals to all the functional blocks of a uARCH to facilitate proper and deterministic functioning of the system. A control-unit's complexity is proportional to the overall complexity of the uARCH.

2.2.5 Memory

The memory lies directly above the register-file in terms of hierarchy and is therefore slower. But, it is comparatively much larger compared to a register-file, and is hence predominantly used to store program instructions and data. A uARCH may have multiple levels of memory, such as, *L1*, *L2* and *L3* caches, depending on the performance goals and design complexity.

2.3 Comet

Comet [14] is a 5-stage pipelined RISC-V core (processor), as shown in figure 2.2. Comet aims to provide a novel methodology, to generate efficient hardware from *C/C++* using High Level Synthesis (HLS), whose resulting hardware is comparable to *ASIC* implementations targeting the 28nm technology node. While reducing design flow complexity by taking advantage of HLS and relevant toolchains.

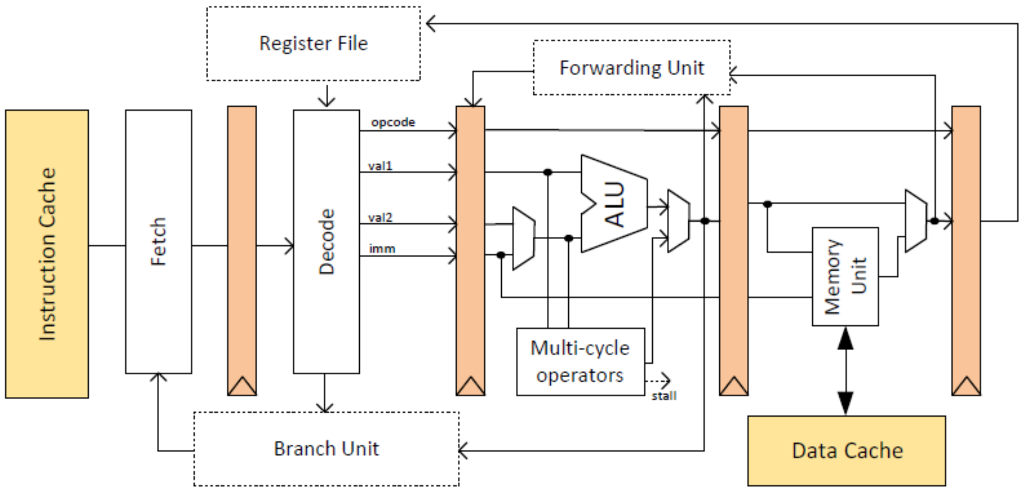


Figure 2.2: Internal organization of a processor with a standard RISC 5-stage pipeline, forward mechanisms, multi-cycle operators, and data and instruction caches [14].

Comet is a good starting point to understand the RISC-V ISA, due to its simplicity of description using C/C++ as a language. It describes the control-path, data-path and other modules of a 5-stage pipelined uARCH well in terms of functionality and data-flow sequencing. However, this high level of abstraction presented by HLS hinders the understanding of *concurrent* operations of all functional units, which is essential for designs aiming to leverage the advantages of using Hardware-Description-Languages (HDLs).

2.4 PicoRV32

Pico [16] is another RISC-V ISA based core, written in *Verilog*, focusing on *Area* optimization. It implements the *RV32I* standard with support for *M* standard extension provided through the *Pico Co-Processor Interface* (PCPI), which connects to co-processors performing multiplication and division operations. The Pico core was intended by designers to be a turnkey solution for simple control tasks in ASIC and FPGA designs [16].

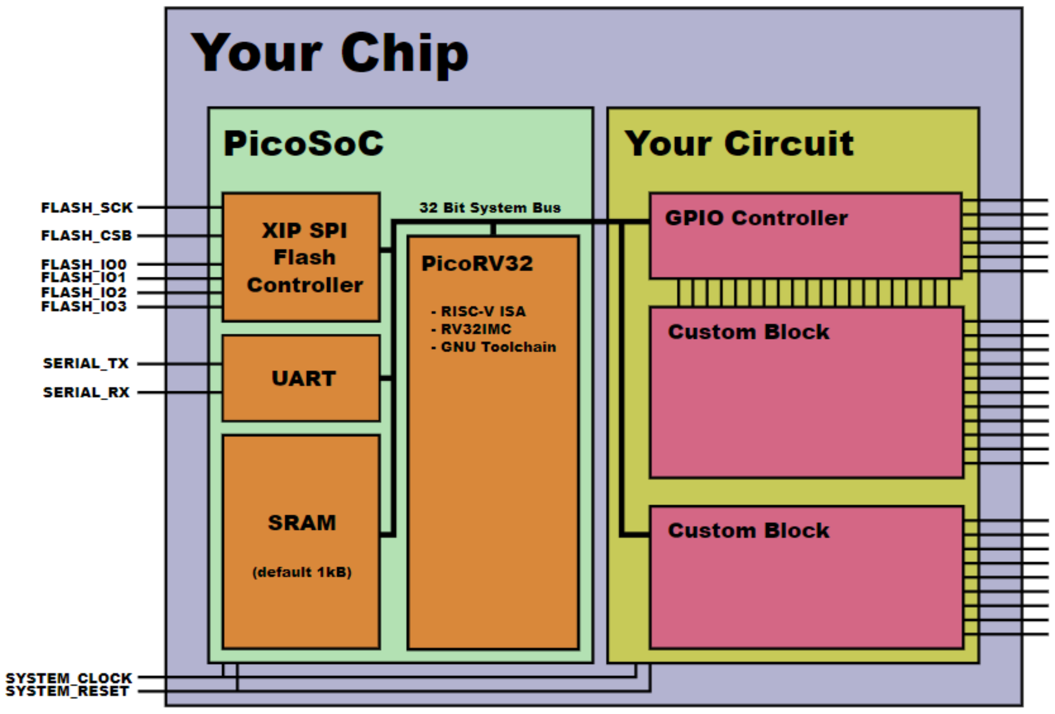


Figure 2.3: Visualization of intended Pico Core usage [16].

3 Design Methodology

In this chapter we will discuss the base constructs, assumptions and the factors backing design decisions. First, we re-iterate over the goal for this project, it is to build a SoCs surrounding a RISC-Vs core supporting the *RV32IM* ISA. Furthermore, the SoC must be optimized for one of the three design factors, *Area*, *Speed* or *Power*. Following the bottom-up paradigm, the problem is divided into parts and addressed individually in the upcoming sections while system integration and implementation is explored in chapter 4.

3.1 Optimization Goals: Power-Performance-Area

Power-Performance-Area (PPA) are the three major components of a process technology, characterizing the physical constraints and available design options for ICs fabricated on that process node. Different trade-offs between the three variables allow for different circuit optimizations and section 1.5 states, the SoC must be optimized for either *Area*, *Speed* or *Power*. But, a unilateral optimization on one of the fore-mentioned factors does not make sense without a standard factor to measure the optimization values against. Hence, we choose a operating frequency of 100MHz to be the target speed of the SoC design, around which the SoC will be optimized for *Power* consumption.

3.2 Micro-Architecture Selection

Based on section 2.1 and section 3.1, an outline of the intended uARCH is created, starting from the Multiply-Divide-Unit (MDU) which is responsible for computing multiply and division instructions. We start with the MDU due to its expected, high combinational logic complexity, assuming it will be the major bottleneck in determining the processor's speed and power consumption. A simple test MDU is created in *Verilog* to determine the maximum achievable speed and corresponding power consumption, of a Standard-Cell-Library (SCL) implementation of the MDU. Shown in table 3.1 are the speed, power and area metrics of the SCL implementation.

An obvious choice of uARCH when optimizing for power is a *Single-Cycle* processor design due to the absence of intermediary registers and design simplicity. However, equally obvious from table 3.1 is the disadvantage of a *Single-Cycle* design when trying to implement a *RV32IM* ISA, in a *28nm* technology node SCL provided [6], at an operating frequency of 100MHz. In-order

Frequency (MHz)	Power (uW)	Area (um^2)	Slack (ns)
10	12.879	5071.247	+5.27
25	37.735	5956.587	+0.01
50	193.407	10708.308	-0.36
100	269.927	10523.097	-10.55

Table 3.1: Resource Utilization of a Multiply-Divide-Unit using a Standard-Cell-Library implementation at different operation frequencies.

to achieve *Single-Cycle* computation, not only is the critical-path of the MDU long (negative slack), but its power consumption is also very large at higher frequencies.

Therefore, a *Pipelined* uARCH is chosen ensuring each phase/stage of the pipeline is given the flexibility to be implemented as *Single-Cycle* or *Multi-Cycle* depending on its complexity, to achieve the target operating frequency of 100MHz. Furthermore, in a *Pipelined* uARCH, a large portion of power can be optimized by carefully pipelining the system to use less number of pipeline registers, and by solving for the power hungry MDU logic.

3.3 Multiplication-Division-Unit

As seen from table 3.1, the MDU becomes quite power hungry as frequency increases and needs to be pipelined to achieve the target frequency of 100MHz. Rather than use an array based solution [15, 12] for multiplication, we explore sequential approaches such as *Modified Booth's Algorithm* for multiplication and *Restoring-Division Algorithm* for division. The advantage of using sequential based multiplication and division algorithm is that, the number of modules required to carry out the operation can be reduced by performing both operations using the same logic, thus reducing power consumption.

3.3.1 Modified Booth's Algorithm

The *Booth's Algorithm* [4] is a multiplication algorithm for both signed and unsigned binary numbers. The algorithm aims to reduce the number of steps required to accumulate the partial-products produced, by optimizing away strings of 1's or 0's in the multiplier. The algorithm can be implemented in a sequential-iterative fashion as shown in figure 3.1. The *Modified* version of the *Booth's Algorithm* performs a *radix-4* encoding of the multiplicand, based on an overlapping grouping of 3 LSB bits of the multiplier [10, 8]. This essentially cuts down on the number of partial products generated and computations needed by half.

3.3.2 Restoring-Division Algorithm

Unlike multiplication, computers find it difficult to perform division operations. Algorithms that perform division can be classified into slow (restoring, non-restoring and SRT) and fast (Newton–Raphson and Goldschmidt) algorithms. This work utilizes the *Restoring-Division* algorithm due to its simplicity and shared similarities in digital-logic between *Restoring-Division*

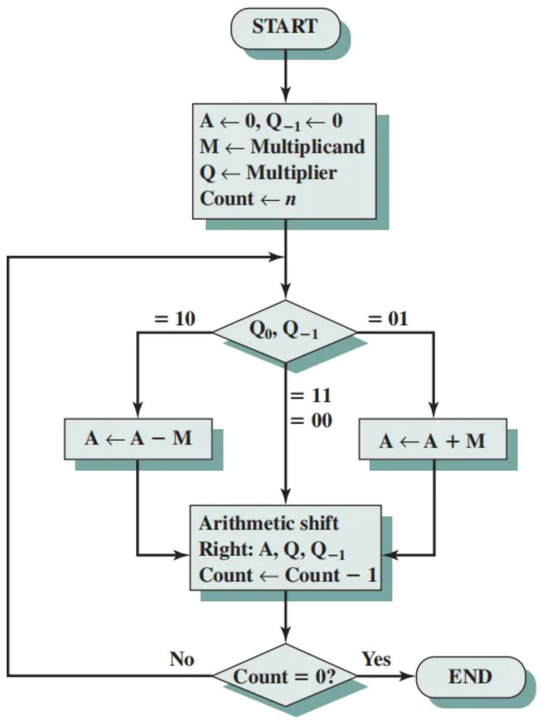


Figure 3.1: Flowchart of Booth's Algorithm (By: Jetnipit Kulrativid - Technical Article on Booth's Algorithm in https://medium.com/).

and *Booth's* algorithm. Shown in figure 3.2 is a flowchart representation of *Restoring-Division* algorithm. Since this algorithm works on *unsigned* integers, every operand is converted to unsigned before division and the result is converted once again to its appropriate sign.

3.4 Power Optimization Techniques

In this section, we will establish how power is generally dissipated in an IC and explore the different techniques commonly used in Very Large-Scale Integration (VLSI) domain to design ICs optimized for power consumption. The power consumption of a digital *CMOS* circuit is characterized by these three components:

- ***Dynamic-Power*** – This component is related to the charging and discharging of the load capacitance seen at the output of a circuit. It is described mathematically as $P_{dynamic} = \alpha \cdot C_L \cdot V_{dd}^2 \cdot f_{op}$. Where, α is the switching activity, C_L is the load capacitance, V_{dd} is the supply voltage and f_{op} is the operating frequency.
- ***Static-Power*** – Even when there is no switching activity, the circuit will consume some power by virtue of its construction. This is known as static power and it is mainly due to the leakage currents that flows, when the transistor is in off-state. It is represented as $P_{static} = V_{dd} \cdot I_{leak}$. Where, I_{leak} is the leakage current.
- ***Short-Circuit-Power*** – When a CMOS gate transitions between logic states (0 -> 1 or 1 -> 0), both NMOS and PMOS transistors are momentarily turned on at the same

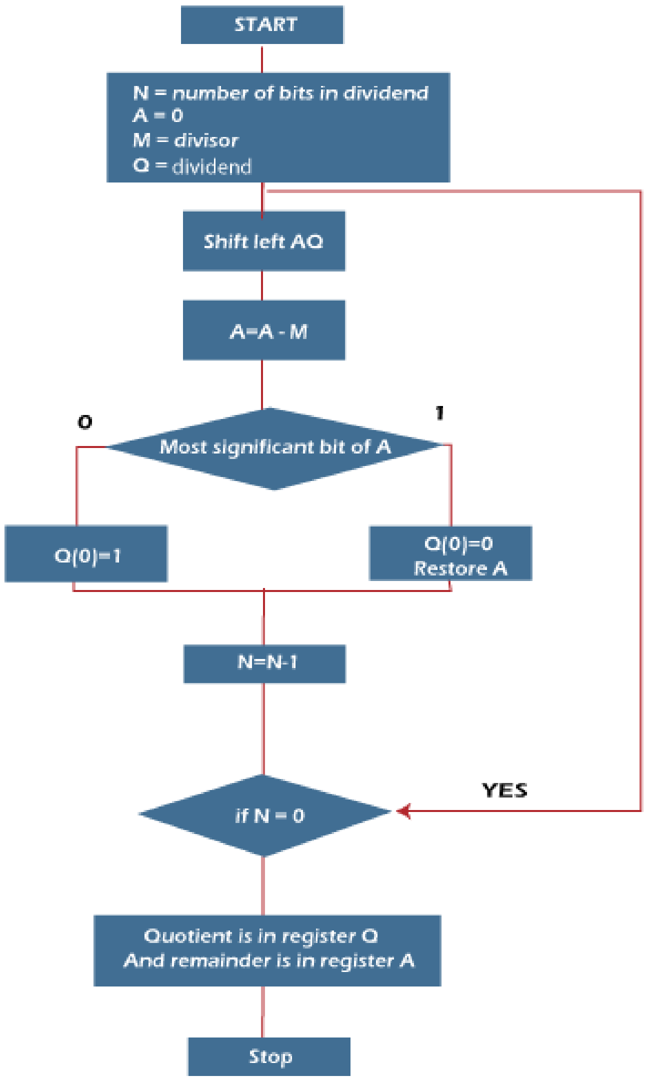

START
N = number of bits in dividend
A = 0
M = divisor
Q = dividend
Shift left AQ
A=A - M
0
1
Most significant bit of A
Q[0]=1
Q[0]=0
Restore A
N=N-1
YES
if N = 0
Quotient is in register Q
And remainder is in register A
Stop

Figure 3.2: Flowchart of Restoring-Division Algorithm (By: https://www.javatpoint.com/restoring-division-algorithm-for-unsigned-integer)

time, causing a short-circuit from V_{dd} to GND. This momentary short-circuit causes the *Short-Circuit-Power* P_{short}

Hence the total power consumption of an IC can be described as:

$$P_{total} = \underbrace{\alpha \cdot C_L \cdot V_{dd}^2 \cdot f_{op}}_{P_{dynamic}} + \underbrace{V_{dd} \cdot I_{leak}}_{P_{static}} + \underbrace{V_{dd} \cdot I_{short}}_{P_{short}}$$

The terms P_{static} and P_{short} are determined by the CMOS technology (technology node) and supply voltage V_{dd}. In-addition, for a fixed V_{dd}, the $P_{dynamic}$ dominates in the case of high switching activity, high fanout C_L and high operating frequency f_{op}. Therefore, we attempt to optimize the design by controlling $P_{dynamic}$.

3.4.1 Clock Gating

Majority of the dynamic power consumption in an IC stems from the switching activity of *sequential-elements* such as registers. Hence, the clock of a *sequential-element* can be deactivated temporarily to save power when the element (register) is not in use. Generally, there are two different techniques of implementing clock gating:

1. ***Intent based Clock-Gating*** – This kind of clock gating is inserted into a design as part of functionality, through Register-Transfer-Level (RTL) code.
2. ***Tool generated Clock Gating*** – This kind of clock gating is inserted by EDA tools during synthesis by identifying Flip-Flops (FFs) that satisfy certain conditions for clock gating.

Throughout the implementation phase, we try to find scenarios where FFs can be gated and apply various clock-gating techniques [5] in RTL.

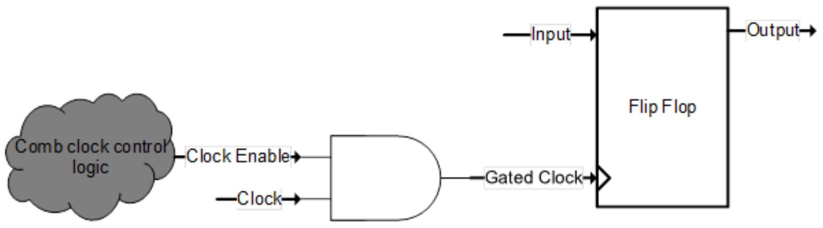


Figure 3.3: Rudimentary AND gate based Clock-Gating logic for a Flip-Flop (By: https://anysilicon.com/the-ultimate-guide-to-clock-gating/)

3.4.2 Datapath Gating

Datapath gating involves reduction of switching activity along the *data-path*. This can be achieved in multiple ways, such as by *Serializing* the data-path to avoid parallel wires. This results in the reduction of parasitic capacitance between wires, reduction in fanout and load capacitance, thus, reducing dynamic power consumption. If necessary, the uARCH can be meticulously studied and datapath gating can be applied to *MUX* select lines to prevent them from unnecessarily switching during inactivity of the data-bus.

3.4.3 Power Gating

This technique is used to temporarily disconnect a module from its supply-rails to mitigate the overall leakage power (P_{static}) of the IC. Since this technique involves switching on-off the power-supply, it requires additional hardware which is capable of handling and monitoring supply voltage and current, while not negatively impacting the supply parameters during normal operation. A power-manager unit is implemented as part of the SoC to perform power-gating where extreme low power operation is intended.

4 Implementation

This chapter will describe the implementation of the uARCH designed in chapter 3, complications faced during implementation and their solutions. The chapter is sectioned, starting with the stages of the uARCH pipeline, touching on *Hazards*, *Racing* and *Metastability*.

4.1 Processor Core

Shown in figure 4.9 is the complete uARCH of the RISC-V core with support for a select few machine level privileged instructions as stated in [11]. Each sub-module will have its own section describing its function and implementation. We start the implementation processes by implementing *Decoder-Unit* and ALU in tandem. This is done to speed up the power optimization process at an early stage to reduce the number of registers (FFs) and datapath signals from the *Decoder* to the ALU, through the pipeline boundary. As previously discussed in section 1.4.1, the RISC-V ISA bitfields are chosen for flexiblity during *MUX* placement. Therefore, we start by implementing the *Decoder* which is responsible for decoding the incoming instructions and setting the corresponding select lines of *MUXs*, which control the operation and operands of the *Execute* stage.

4.1.1 Decode-Phase

The implementation of this unit is simple, with most of the complication arising from *MUX* placement and datapath arrangement to reduce delay along the datapath before entering *DE2EX* FFs. The main part of the decoder consists of a Finite-State-Machine (FSM) (figure 4.1), present as part of the *Control-Unit* to decode incoming instructions and assert the appropriate control signals. In-order to assist the decoder, an *Immediate-Generate-Unit* is included to build the various *Immediates* stipulated in the RISC-V ISA (figure 4.2b) as part of the instruction being decoded.

4.1.2 Execute-Phase

Construction of the *Execution-Unit* starts around the ALU [13]. We transform the generic ALU implementation by optimizing the control signals and datapath to reduce the number of wires. To further optimize for power, the multiplication and division algorithms described in section 3.3 are implemented by re-using the ALU components, and the 64-bit *Accumulator*

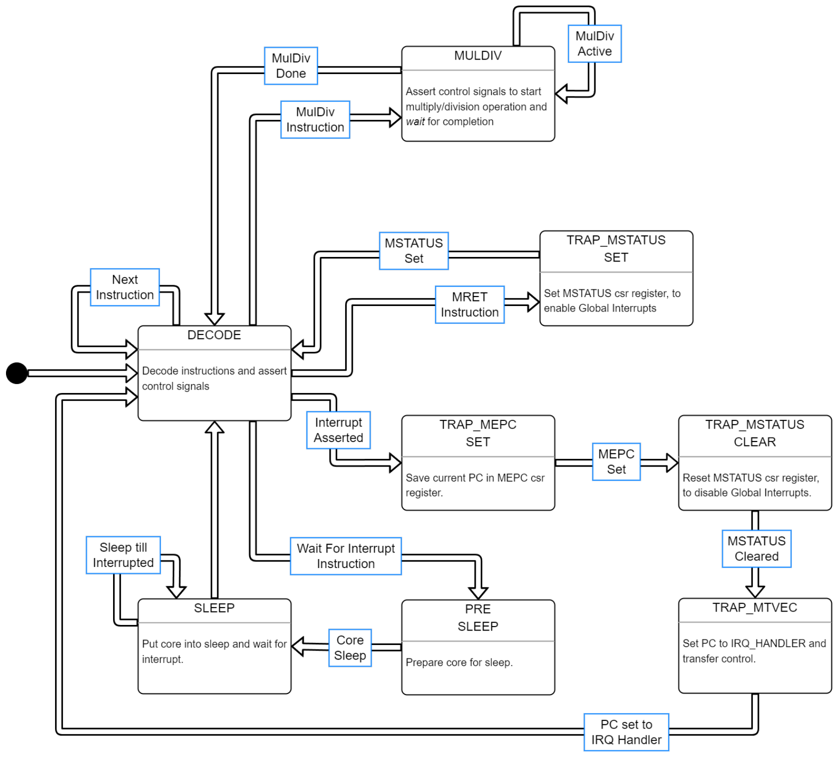

MULDIV
Assert control signals to start multiply/division operation and *wait* for completion
MulDiv Active
MulDiv Done
MulDiv Instruction
MSTATUS Set
TRAP_MSTATUS SET
Set MSTATUS csr register, to enable Global Interrupts
MRET Instruction
Next Instruction
DECODE
Decode instructions and assert control signals
Interrupt Asserted
TRAP_MEPC SET
Save current PC in MEPC csr register.
MEPC Set
TRAP_MSTATUS CLEAR
Reset MSTATUS csr register, to disable Global Interrupts.
MSTATUS Cleared
Sleep till Interrupted
Wait For Interrupt Instruction
SLEEP
Put core into sleep and wait for interrupt.
Core Sleep
PRE SLEEP
Prepare core for sleep.
TRAP_MTVEC
Set PC to IRQ_HANDLER and transfer control.
PC set to IRQ Handler

Figure 4.1: Decoder Control-Unit's Finite-State-Machine.

(used to hold the results of multiplication and division) is first, split into two 32-bit registers with the lower half merged into the *LHS* register (part of *DE2EX REGS*). This is done through the *Accum_shifter* unit (figure 4.9) to reduce the register count, thus, reducing overall power consumption.

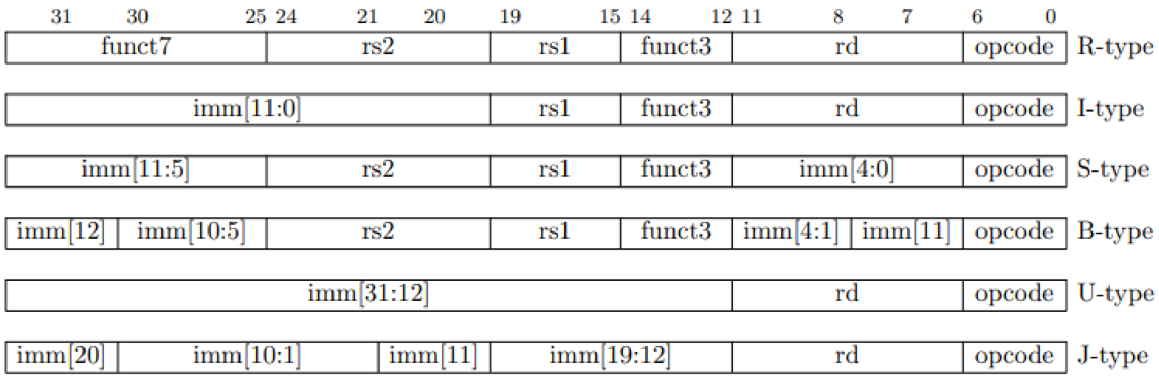
4.1.3 Memory and Writeback Phase

Unlike the generic 5-stage pipeline (figure 2.2), this work's uARCH utilizes a 4-stage pipeline by combining the *Memory* and *Writeback* phases into one. This is due to two reasons:

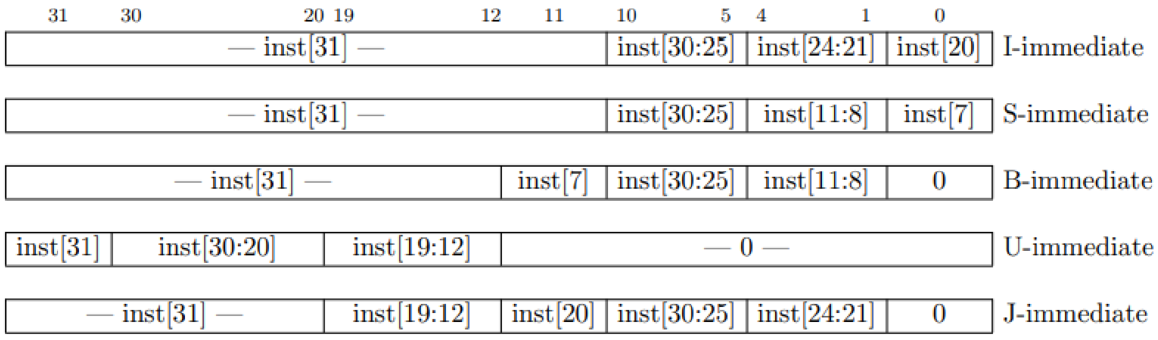
1. First, the *Memory* phase being very simple and lightweight, which is made possible by implementing calculated *stall* cycles before, during and after memory bus access through the MMU, resulting in enough time for the signals to propagate through the *Critical-Path*. Thus, any data/result that flows into the *Memory* phase can be directly passed onto the respective *Register-Files*.
2. Secondly, the register files are designed to perform both *Writes* during the negative edge of the clock and *Reads* asynchronously, enabling same cycle writes and reads.

4.1.4 Fetch-Phase

This phase is rather straightforward in implementation, it simply selects the appropriate *Program-Counter* or instruction address from a *MUX* that has the *Branch* target from *Execution-*


31 30 25 24 21 20 19 15 14 12 11 8 7 6 0
funct7 | rs2 | rs1 | funct3 | rd | opcode | R-type
imm[11:0] | rs1 | funct3 | rd | opcode | I-type
imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | opcode | S-type
imm[12] | imm[10:5] | rs2 | rs1 | funct3 | imm[4:1] | imm[11] | opcode | B-type
imm[31:12] | rd | opcode | U-type
imm[20] | imm[10:1] | imm[11] | imm[19:12] | rd | opcode | J-type

(a)


31 30 20 19 12 11 10 5 4 1 0
— inst[31] — | inst[30:25] | inst[24:21] | inst[20] | I-immediate
— inst[31] — | inst[30:25] | inst[11:8] | inst[7] | S-immediate
— inst[31] — | inst[7] | inst[30:25] | inst[11:8] | 0 | B-immediate
inst[31] | inst[30:20] | inst[19:12] | — 0 — | U-immediate
— inst[31] — | inst[19:12] | inst[20] | inst[30:25] | inst[24:21] | 0 | J-immediate

(b)

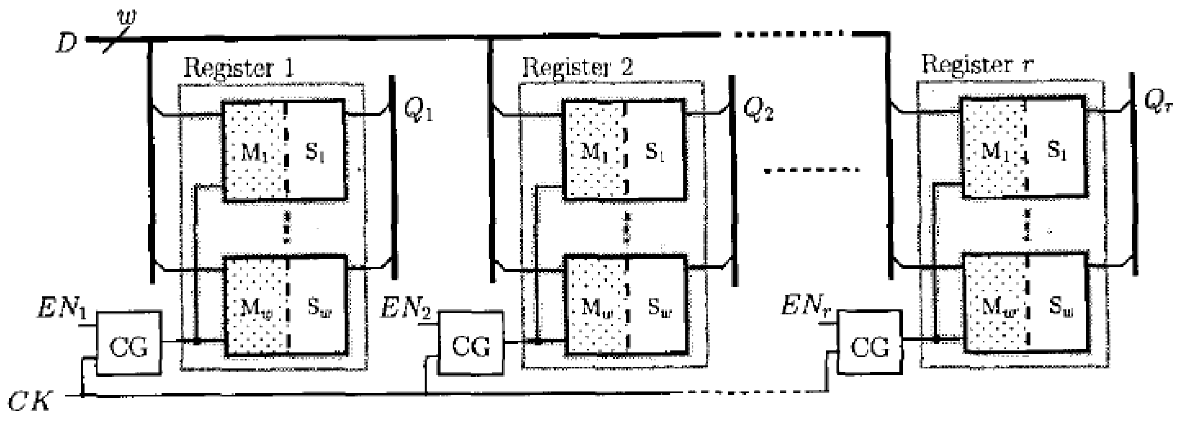
Figure 4.2: (a) RISC-V instruction types. (b) RISC-V Immediate types.

Phase, *Jump* target from *Decode-Phase* and the next immediate instruction (PC+4) from the *Fetch-Phase*.

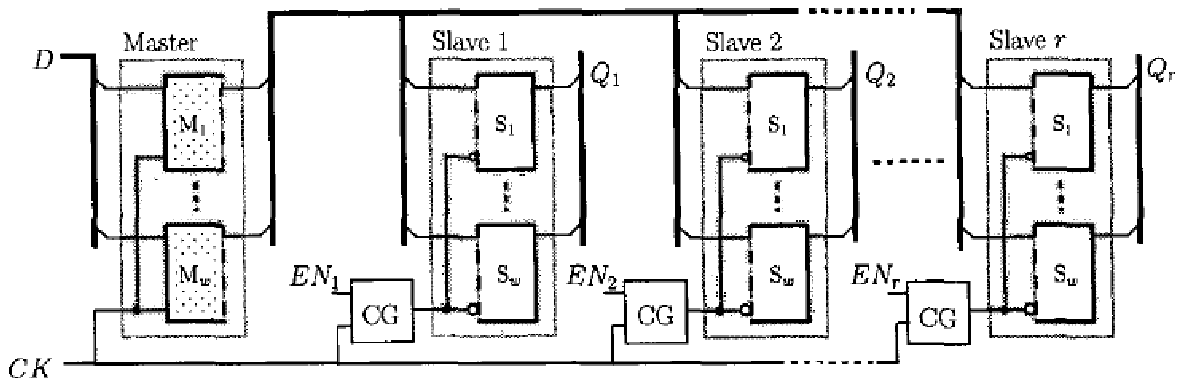
4.1.5 Register-File

As previously described, the *Register-File* is implemented with *Writes* occurring on the negative edge of clock and *Reads* happening asynchronously. The reads are asynchronous since any value read from the *Register-File* will end up in a positive edge clocked register in the *Decoder-Phase*. Thus, ultimately making the *Reads* happen on the positive edge of clock as seen from the *Execute-Phase*, where the data is put to use. This ensures there are no Write-After-Read (WAR) hazards, since a write (negedge) will always happen after a read (posedge).

After the first version of the core's implementation, the power reports exhibit a large energy consumption due to the operation of register files. Hence, different options were explored to reduce the power consumed by the register file. One such solution applied in this work stems from [17], where the register file is implemented with one *Master-Latch* and several *Slave-Latches* to form a register file. A single FF can be implemented using a *Master-Slave* latch combination as shown in figure 4.3a, this leads to the concept of using a single *Master-Latch* for the whole register file as shown in figure 4.3b. The arrangement of single *Master-Latch* with multiple *Slave-Latch* predominantly saves on power because of fewer memory resources (latches) compared to the traditional method. The effect of the type of register-file implementations, on power consumption is discussed in chapter 5.



(a)



(b)

Figure 4.3: (a) A conventional register-file implemented by the combination of Master-Latch (M_i) and Slave-Latch (S_i). (b) The proposed single Master multiple Slave latch model. [17]

4.1.6 Branch-Unit

The Branch-Unit is responsible for asserting branch signals to the *Fetch-Phase* and also avoids *Control-Hazards* buy flushing the pipeline. The *Decode-Phase*, *Execute-Phase* and *Branch Comparator* (figure 4.9) collectively compute the branch destination address and whether a branch should be taken or not, in the case of a conditional branch statement. To save on power, the branch resolution mechanism is kept simple and does not contain any complicated branch prediction logic. By default the branch is assumed to be *not taken*, and when taken, the pipeline is simply flushed.

4.1.7 Stall and Forward Unit

Due to heavy focus on power optimization, the *Stall-Unit* plays a major role in (de)asserting the stall signals to various pipeline registers and stages, at the right moment. Improper timing on the stall signals will cause the pipeline to break and behave in a nondeterministic fashion. The *Forward-Unit*, which is a part of the *Stall-Unit*, is responsible for resolving *Data-Hazards* by forwarding the appropriate result to the *Execute-Phase* during Read-After-Write (RAW) hazard.

Due to the guaranteed path-delay between databus-input and *Register-File* in the *Memory-Phase*, the uARCH does not employ a *ME2WB* pipeline register, thus reducing on a couple registers and saving on power consumption. Furthermore, after repeated analysis on the pipeline timing and operation, *Clock-Gating* is applied to all possible registers to reduce power consumption. For example, during a *Multi-Cycle* divide or multiply operation, the pipeline is stalled, meaning certain pipeline registers can have their clocks gated under certain conditions to save energy.

4.2 Init-Controller, Memory-Manager and External-Bus Interface

We build the MMU around the uARCH of the core, while adhering to the memory specifications from section 1.5. The specification states a 16KB SRAM usage for program memory, which can also be used to store data, insinuating we can use a *Von-Neumann* architecture for memory. Therefore, the datapath to and from the internal memory can be serialized, along with disabling read registers of SRAM blocks not being accessed, to achieve higher gains in power reduction. Furthermore, the *Init-Controller*, responsible for the initialization of the internal SRAM modules upon reset, is integrated into the MMU by making the initiation process a part of the MMU's FSM.

In tandem with a simple MMU, a simple External-Bus Interface (EBI) is implemented to access external peripherals and memory. The EBI does not contain any extra buffers or complicated logic for burst transmission, hence saving on power consumption. Both the MMU and EBI have their buffers *Clock-Gated* to conserve energy when not in use.

4.3 Test Environment

The test environment for this project is built using the *RISCV-GNU* toolchain and scripts provided within the *ICPRO* environment, along with a custom *Testbench* to test the Device-Under-Test (DUT). This section aims to provide a brief overview of the testing and validation process used to verify the SoC design.

4.3.1 Testbench

A comprehensive *Testbench* is built using *Verilog* to test the SoC design, and validate its functionalities after various design iterations through *Synthesis* and *Place-and-Route*. The *Testbench* is responsible for simulating the external environment of the DUT, which in our case is the SoC. As shown in figure 4.4, the *Testbench* simulates an external *Slave* device and *External SRAM*, to provide program memory to SoC upon reset and emulate external I/O and bus transactions.

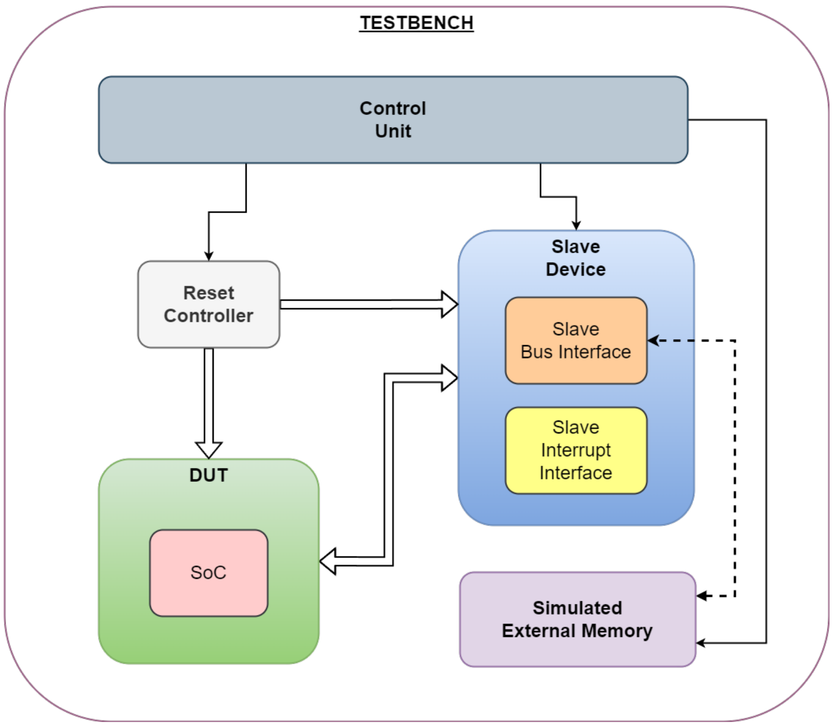


Figure 4.4: Block diagram of Testbench.

The external stimuli are broken down into concrete parts and implemented as standalone *Tasks* (functions), so that they may be called in any order to simulate the external environment. Each task has built-in timing delays to ensure the external signals are asserted at the correct intervals with respect to the global clock. In-addition, various *parameters* have been defined to help control the behaviour of the *Testbench*, thus making the *Testbench* comprehensive and flexible.

4.3.2 Test Programs and Compiler

The *ICPRO* project management environment provides the necessary support software, such as *riscv-gnu-toolchain*, linker scripts and other related program files to make the process of compilation relatively flexible. To test the SoC, a simple C program is written to perform basic ALU operations, multiplication, division and *Bubble-Sort*.

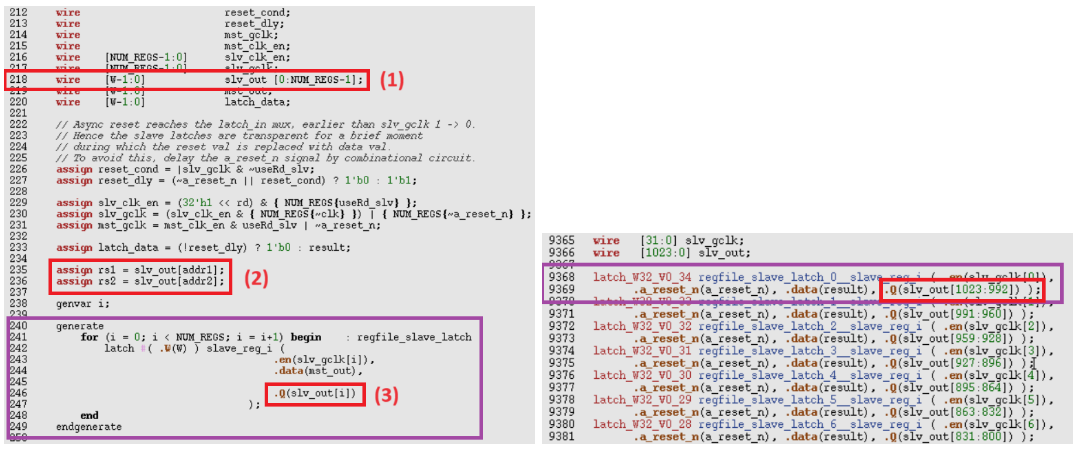
To test more complex operations such as *Illegal-Instruction-Handling* and *Privileged* instructions like *Wait-For-Interrupt*, custom *assembly* instructions are injected from the *Reset-Handler* routine upon start-up of the SoC.

4.4 Complications

In this section, we shed light on some of the complications faced during implementation and synthesis of the design. The complications are varied, with some related to the synthesized design not meeting Static-Timing-Analysis (STA) requirements, or the synthesis tool breaking design functionality due to optimizations.

4.4.1 Synthesis of Packed and Unpacked Arrays

During the implementation of the custom *Master-Slave* latch register file, synthesis presented some problems due to misinterpretation of the code used to describe the register file. The initial implementation of the register file is done as shown in figure 4.5a, where the master and slave latches are implemented using *generate* blocks, while reads are implemented using a decoder. The problem arises during synthesis, when the 32-bit vector *'slv_out'* unpacked array containing 32 elements, is decomposed into a single packed vector of 1024-bits and assigned to the outputs of individual slave latches. Although the decomposition itself does not present problems, the assignment order of the bits is inverted as shown in figure 4.5b, resulting in the decoder reading from *register_31* when the intended read address is *register_0*.



```
wire                        reset_cond;
wire                        reset_dly;
wire                        mst_gclk;
wire                        mst_clk_en;
wire    [NUM_REGS-1:0]      slv_clk_en;
wire    [NUM_REGS-1:0]      slv_gclk;
wire    [W-1:0]             slv_out [0:NUM_REGS-1];
wire    [W-1:0]             mst_out;
wire    [W-1:0]             latch_data;

// Async reset reaches the latch_in mux, earlier than slv_gclk 1 -> 0.
// Hence the slave latches are transparent for a brief moment
// during which the reset val is replaced with data val.
// To avoid this, delay the a_reset_n signal by combinational circuit.
assign reset_cond = |slv_gclk & ~useRd_slv;
assign reset_dly = (~a_reset_n || reset_cond) ? 1'b0 : 1'b1;

assign slv_clk_en = (32'h1 << rd) & { NUM_REGS{useRd_slv} };
assign slv_gclk = (slv_clk_en & { NUM_REGS{~clk} }) | { NUM_REGS{~a_reset_n} };
assign mst_gclk = mst_clk_en & useRd_slv | ~a_reset_n;

assign latch_data = (!reset_dly) ? 1'b0 : result;

assign rs1 = slv_out[addr1];
assign rs2 = slv_out[addr2];

genvar i;

generate
    for (i = 0; i < NUM_REGS; i = i+1) begin    : regfile_slave_latch
        latch #( .W(W) ) slave_reg_i (
                                .en(slv_gclk[i]),
                                .data(mst_out),

                                .Q(slv_out[i])
                            );
    end
endgenerate
```

```
wire   [31:0] slv_gclk;
wire   [1023:0] slv_out;

latch_W32_V0_34 regfile_slave_latch_0__slave_reg_i ( .en(slv_gclk[0]),
        .a_reset_n(a_reset_n), .data(result), .Q(slv_out[1023:992]) );
latch_W32_V0_33 regfile_slave_latch_1__slave_reg_i ( .en(slv_gclk[1]),
        .a_reset_n(a_reset_n), .data(result), .Q(slv_out[991:960]) );
latch_W32_V0_32 regfile_slave_latch_2__slave_reg_i ( .en(slv_gclk[2]),
        .a_reset_n(a_reset_n), .data(result), .Q(slv_out[959:928]) );
latch_W32_V0_31 regfile_slave_latch_3__slave_reg_i ( .en(slv_gclk[3]),
        .a_reset_n(a_reset_n), .data(result), .Q(slv_out[927:896]) );
latch_W32_V0_30 regfile_slave_latch_4__slave_reg_i ( .en(slv_gclk[4]),
        .a_reset_n(a_reset_n), .data(result), .Q(slv_out[895:864]) );
latch_W32_V0_29 regfile_slave_latch_5__slave_reg_i ( .en(slv_gclk[5]),
        .a_reset_n(a_reset_n), .data(result), .Q(slv_out[863:832]) );
latch_W32_V0_28 regfile_slave_latch_6__slave_reg_i ( .en(slv_gclk[6]),
        .a_reset_n(a_reset_n), .data(result), .Q(slv_out[831:800]) );
```

Figure 4.5: (a) (1) Shows the multi-dimensional unpacked array used to read from the slave latches, (2) shows the implicit decoders, and (3) shows the assignment of 'slv_out' array to corresponding slave latches. (b) Result of synthesis, decomposing the array to a single vector of 1024-bits wide and assigning slice 'slv_out[1023:992]' (which is slv_out[31] in unpacked format) to slave latch 0.

The issue was resolved by implementing *'slv_out'* as a 1024-bit wide vector and assigning individual 32-bit slices to the correct latch, through the generate block.

4.4.2 Post-Synthesis Master-Slave Latch Timing

A problem with regard to timing arises during *post-synthesis* simulation, with the utilization of latches. The latch based register file operates by turning the *Master-Latch* transparent during the positive clock and the *Slave-Latch* transparent during the negative clock. The problem occurs during the transition from negative half to positive half of the clock period. Due to the delay in the clock gating circuitry, the slave latches stay transparent for a brief moment during the positive clock cycle, while the master latches are transparent as shown in figure 4.6. This causes the previous slave's data to be overwritten with the masters data, which is to be written in a new slave.

Lets take a look at figure 4.6, the red box corresponds to the transition of slave gated clock ('slv_gclk') during negative half of clock and violet box is for the 'slv_gclk' transition during

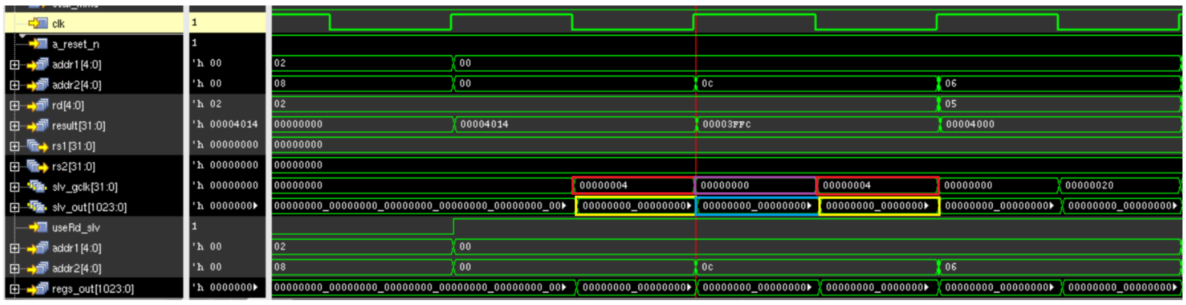


Figure 4.6: Depiction of Master-Slave Latch timing after synthesis.

positive half. Similarly, the yellow box marks the transition of slave latches during negative cycle and the blue marks the transition during positive cycle. Clearly the blue box transition should not occur (the yellow value must be held till next negative cycle), but it happens due to both master and slave latches being transparent at the same time. The workaround for this is to introduce additional logic, ensuring the master's clock is enabled only when all the slaves have their clocks disabled. This ensures both the master and slave latches are not transparent at the same time.

4.4.3 Explicit Clock-Gating

Adding clock gating explicitly through RTL using enable latches and gates on the clock tree, messes with the STA of the synthesis tool, resulting in *setup* and *hold* timing faults on the FF's clock pin. This happens due to the extra overhead added by the gating logic to the clock tree, resulting in a delayed clock and therefore harder STA for the synthesis tool.

The solution is to use Integrated Clock Gating Cell (ICGC), which is a library cell designed with tightly coupled logic and buffers to ensure signals on the clock tree meet their strict timing constraints, thus facilitating better STA. In-order to insert ICGCs, explicit gating logic must be removed and made implicit by adding enable signals to the FFs, and directing the synthesis tool to perform clock gating on FFs. This results in the synthesis tool inserting ICGCs before the concerned FF.

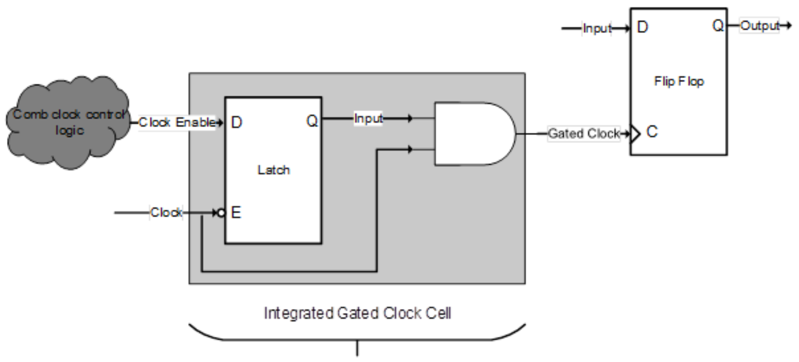
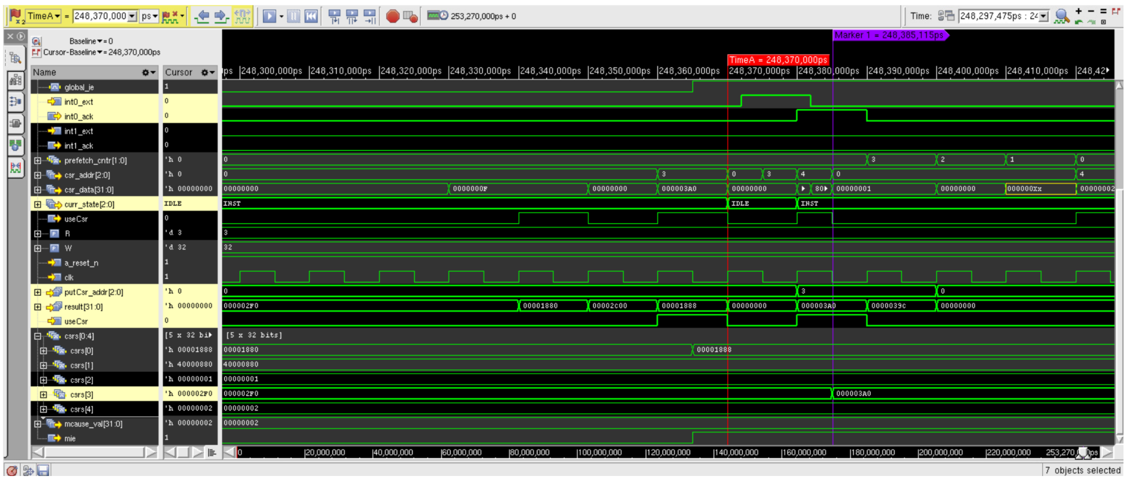


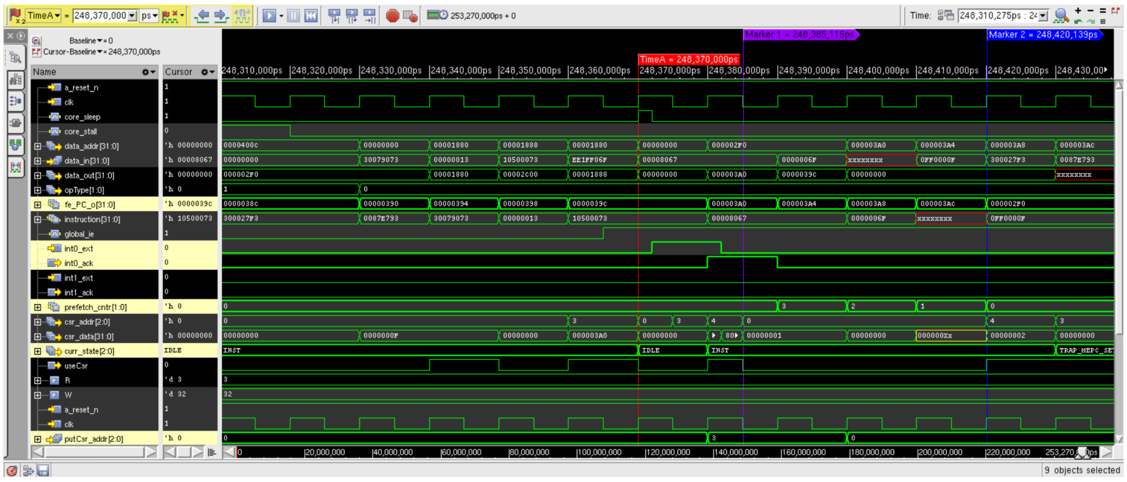
Figure 4.7: Clock Gating performed by integrated clock gating cell to meet STA constraints (By: https://anysilicon.com/the-ultimate-guide-to-clock-gating/).

4.4.4 Pipeline Timing

In the presence of an illegal instruction, the current PC is stored in *MEPC* csr register and program control is transferred to the *ILLEGAL_INSTRUCTION_HANDLER*, which prepares the core for sleep by calling the *Wait-For-Interrupt* (WFI) instruction. The *WFI* instruction sets the *MEPC* to the next_PC of the current instruction and puts the core in *IDLE/SLEEP* mode where it waits for an external interrupt to wake it up, while it takes 2 cycles for the new *MEPC* value to be stored in *MEPC* csr register due to pipelining.



(a)



(b)

Figure 4.8: (a) The red line indicates when the core enters IDLE mode ('curr_state'), the violet (marker_1) indicates the cycle in which the new MEPC value is stored ('csrs[3]'). (b) The blue line (marker_2) indicates the start of execution at the old MEPC value ('fe_PC_o').

The problem arises, when the external interrupt appears immediately after the core enters sleep, which causes the core to start execution at the old *MEPC* pointing to the illegal instruction, resulting in a loop as shown in figure 4.8. A fix for this situation is applied by flushing the pipeline for 3 cycles, which causes the core to disable global interrupts during these 3 cycles (by virtue of uARCH design), thus providing the pipeline enough time to reflect the change into *MEPC* csr register.

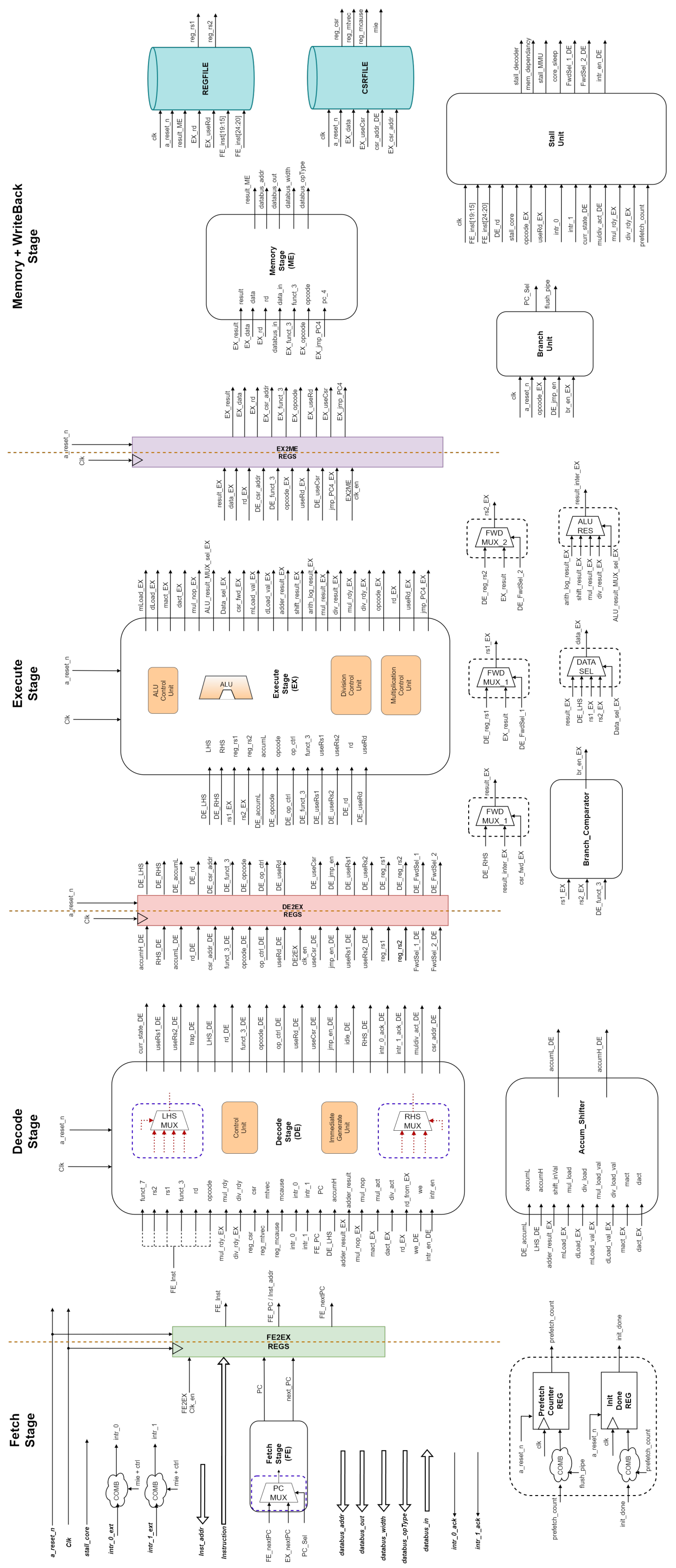


Figure 4.9: Microarchitecture block diagram of four stage pipelined core. Signals shown in ***bold-italic*** fonts depict the core's I/O signals.

5 Results and Conclusion

This chapter will discuss the results obtained from the evaluations made during the testing of the SoC. We first explore the results from the power optimization techniques implemented in the RISC-V core's design before validation of its functionality using a simple *C program* implementing external-bus transactions, simple ALU operations, and a *Bubble-Sort* algorithm.

5.1 Register-File Power Optimization

We start the evaluating the power consumption of a *Generic-Register-File* (GRF) implementation and the *Master-Slave-Latch* (MSL) based implementation. The results of various register-file sizes, clocked at 100MHz (target operation frequency) is shown in table 5.1.

Register-File Size (No. 32bit Regs)	GRF Power (uW)	MSL Power (uW)	Power Saved by MSL (%)
5	3.381	2.793	17.391
8	9.704	5.856	39.654
16	13.406	7.859	41.377
32	20.936	14.005	33.106
64	35.732	23.011	35.601
128	66.471	38.095	42.689

Table 5.1: Power consumption of the two register file implementations (GRF and MSL) measured at 100MHz operating frequency.

The *Master-Slave-Latch* based register file implementation reduces power consumption by 33.106% compared to the generic implementation of 32$\times$32-bit registers. Furthermore, a trend can be noticed where the power reduction rises and falls within a certain range and does not continue to keep rising, this is due to the increase in:

1. ***Dynamic Power*** caused by the increase in load capacitance seen at the output of the *Master-Latch* due to high fanout to slaves. A workaround for this is to use memory bank based system, where each bank is driven by the same master, helping control the max fanout per master latch.
2. ***Auxiliary Power*** which is consumed by the clock-gating logic and other supporting logic such as, the timing logic added to ensure proper master-slave latch operation as previously discussed in section 4.4.2.

5.2 SoC Power Optimization

The SoC is heavily clock-gated to optimize for power whenever possible during normal execution. To test the results of power-optimization post-synthesis, a baseline version of the SoC is synthesized without any clock-gating circuitry insertion. By removing the *-gate_clock* option from the *compile_ultra* command in the *Synopsis Design Compiler* compilation script, we indicate to the synthesis tool, to not perform clock-gating on any FFs. A plot of the overall power consumption of baseline model and power-optimized SoC is shown in figure 5.1 for different operating frequencies.

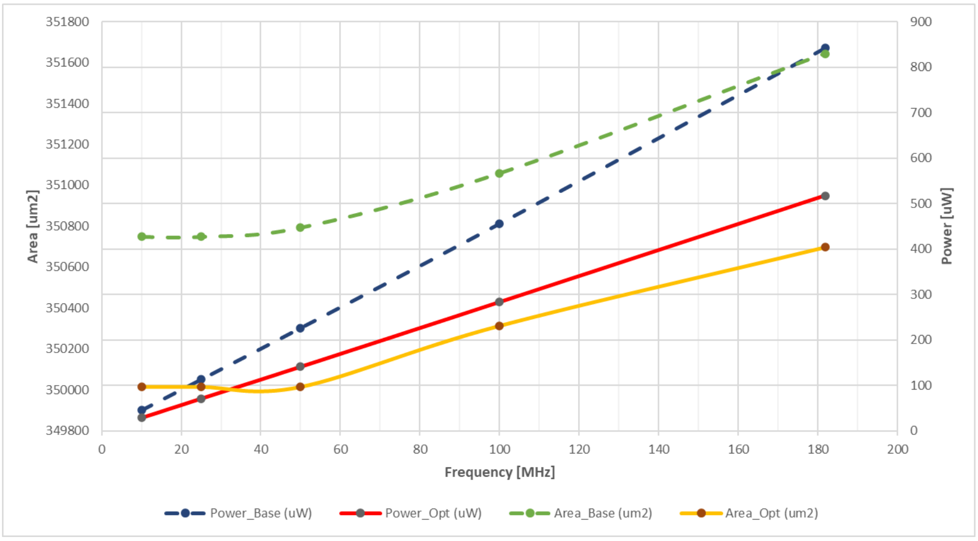


Figure 5.1: Power and Area against different operating frequencies (F_{max} = 181.818*MHz*).

This work has a maximum operating frequency (F_{max}) of 181.818*MHz* and consumes 37.942% less energy compared to the unoptimized version (baseline) at the target operating frequency of 100*MHz*. The growth of power consumption with respect to frequency, for different class of components (I/O Pad, Registers, Clock-Network, etc) is shown in figure 5.2. The *IO_Pads* have the largest slope, suggesting a lower SoC power consumption for a target frequency can be achieved by designing the *IO_Pads* to operate at a much lower rate (frequency) than that of the RISC-V core's.

Comparing the core's total power consumption (table 5.2) with the synthesis tool's ALU implementation (table 3.1) at the target operating frequency of 100*MHz*, results in the conclusion that the *Modified-Booths-Algorithm* and *Restoring-Division-Algorithm* greatly reduce power consumption for multiply and divide operations, at the cost of higher cycle count per operation.

Module	Power (uW)	Area (um2)
RISCV-Core	69.799	6823.557
MMU+EBI	51.080	36237.084
IO_Pads	157.933	307252.594
SoC_Top	283.025	350313.235

Table 5.2: Power consumption of sub-modules measured at 100MHz operating frequency.

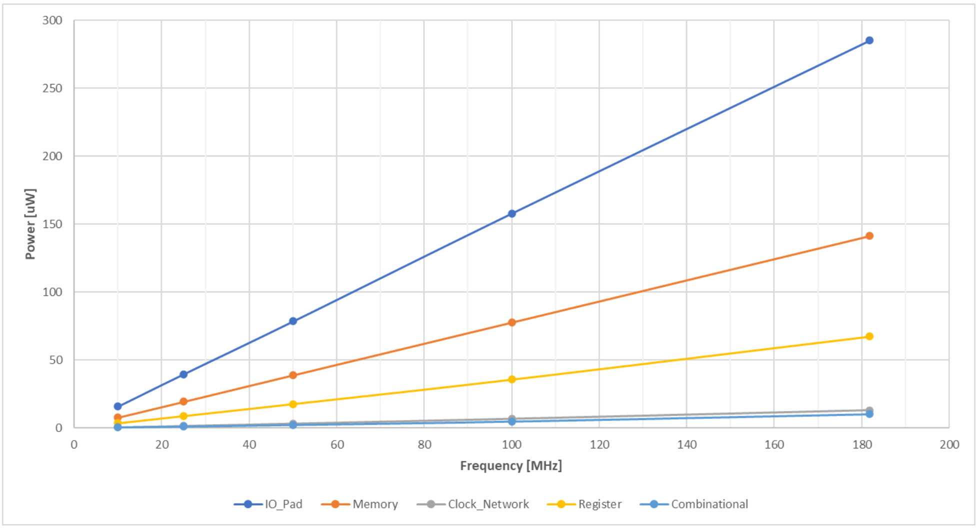
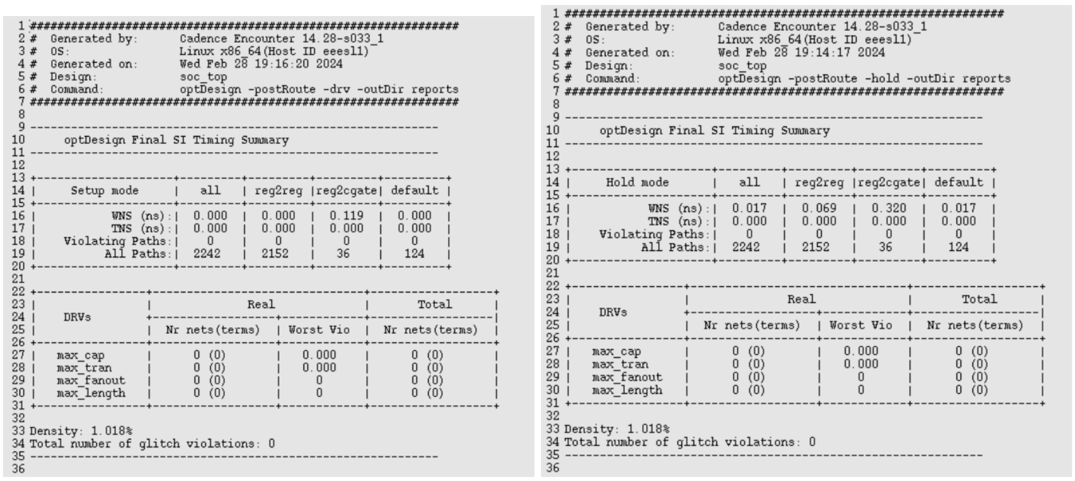


Figure 5.2: Power of sub-components against different operating frequencies (F_{max} = 181.818*MHz*).

5.3 SoC Implementation

After synthesis using *Synopsis Design Compiler*, we implement the design at the target frequency (F) of 100*MHz* by performing PnR of the synthesized design. All the PnR workload is offset to the *Innovus* tool which performs PnR. Marginally better results maybe obtained by manually guiding the tool through modified scripts, into achieving the required results. Shown in figure 5.4 is the final result of PnR and in figure 5.3 the overall timing reports post-route (setup and hold), taking Signal-Integrity (SI) (section 6.1) into account.



```
###################################################################
#   Generated by:       Cadence Encounter 14.28-s033_1
#   OS:                 Linux x86_64(Host ID eeesl1)
#   Generated on:       Wed Feb 28 19:16:20 2024
#   Design:             soc_top
#   Command:            optDesign -postRoute -drv -outDir reports
###################################################################

------------------------------------------------------------
     optDesign Final SI Timing Summary
------------------------------------------------------------

+--------------------+---------+---------+---------+---------+
|     Setup mode     |   all   | reg2reg |reg2cgate| default |
+--------------------+---------+---------+---------+---------+
|           WNS (ns):|  0.000  |  0.000  |  0.119  |  0.000  |
|           TNS (ns):|  0.000  |  0.000  |  0.000  |  0.000  |
|    Violating Paths:|    0    |    0    |    0    |    0    |
|          All Paths:|  2242   |  2152   |   36    |   124   |
+--------------------+---------+---------+---------+---------+

+----------------+-------------------------------+------------------+
|                |              Real             |       Total      |
|    DRVs        +------------------+------------+------------------|
|                | Nr nets(terms)   | Worst Vio  |  Nr nets(terms)  |
+----------------+------------------+------------+------------------+
|   max_cap      |      0 (0)       |   0.000    |      0 (0)       |
|   max_tran     |      0 (0)       |   0.000    |      0 (0)       |
|   max_fanout   |      0 (0)       |     0      |      0 (0)       |
|   max_length   |      0 (0)       |     0      |      0 (0)       |
+----------------+------------------+------------+------------------+

Density: 1.018%
Total number of glitch violations: 0
------------------------------------------------------------

```

(a)

```
###################################################################
#   Generated by:       Cadence Encounter 14.28-s033_1
#   OS:                 Linux x86_64(Host ID eeesl1)
#   Generated on:       Wed Feb 28 19:14:17 2024
#   Design:             soc_top
#   Command:            optDesign -postRoute -hold -outDir reports
###################################################################

------------------------------------------------------------
     optDesign Final SI Timing Summary
------------------------------------------------------------

+--------------------+---------+---------+---------+---------+
|     Hold mode      |   all   | reg2reg |reg2cgate| default |
+--------------------+---------+---------+---------+---------+
|           WNS (ns):|  0.017  |  0.069  |  0.320  |  0.017  |
|           TNS (ns):|  0.000  |  0.000  |  0.000  |  0.000  |
|    Violating Paths:|    0    |    0    |    0    |    0    |
|          All Paths:|  2242   |  2152   |   36    |   124   |
+--------------------+---------+---------+---------+---------+

+----------------+-------------------------------+------------------+
|                |              Real             |       Total      |
|    DRVs        +------------------+------------+------------------|
|                | Nr nets(terms)   | Worst Vio  |  Nr nets(terms)  |
+----------------+------------------+------------+------------------+
|   max_cap      |      0 (0)       |   0.000    |      0 (0)       |
|   max_tran     |      0 (0)       |   0.000    |      0 (0)       |
|   max_fanout   |      0 (0)       |     0      |      0 (0)       |
|   max_length   |      0 (0)       |     0      |      0 (0)       |
+----------------+------------------+------------+------------------+

Density: 1.018%
Total number of glitch violations: 0
------------------------------------------------------------

```

(b)

Figure 5.3: (a) Post PnR setup timing summary, (b) Post PnR hold timing summary of implemented SoC design.

The *Core-Utilization Density* (section 6.2) of the implementation is 1.018%, suggesting there is a lot more room for improvement in the PnR department. In our case, the implemented SoC design consumes very little area compared to the *IO_Pads*, forcing the aspect-ratio of the core

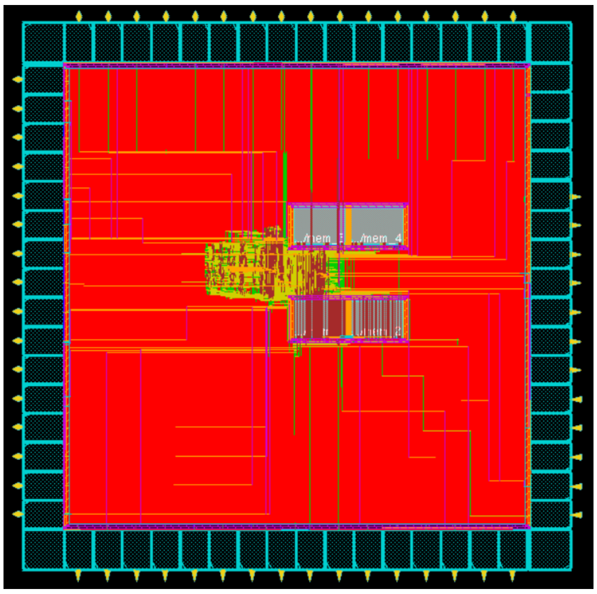
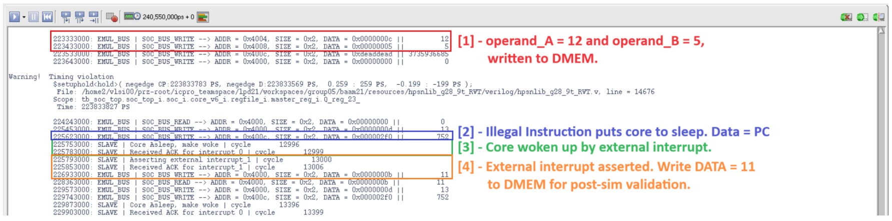


Figure 5.4: Visualization of implemented SoC design at target frequency of 100*MHz*.

to remain higher, resulting in a lot of under-utilized space and low *Core-Utilization Density*.

The resulting implementation of the SoC was verified by testing its functionality at the target frequency (F_{op} = 100*MHz*) using the the testbench and program described in section 4.3. The console output of the test-program is shown in figure 5.5, the implemented SoC successfully manages to perform, basic ALU operations, multiplication, division and bubble-sort on an array.



```
240,550,000ps + 0

          223333000: EMUL_BUS | SOC_BUS_WRITE --> ADDR = 0x4004, SIZE = 0x2, DATA = 0x0000000c ||         12
          223433000: EMUL_BUS | SOC_BUS_WRITE --> ADDR = 0x4008, SIZE = 0x2, DATA = 0x00000005 ||          5
          223533000: EMUL_BUS | SOC_BUS_WRITE --> ADDR = 0x400c, SIZE = 0x2, DATA = 0xdeaddead || 3735936685
          223643000: EMUL_BUS | SOC_BUS_WRITE --> ADDR = 0x4000, SIZE = 0x2, DATA = 0x00000000 ||          0

Warning!  Timing violation
          $setuphold<hold>( negedge CP:223833783 PS, negedge D:223833569 PS,  0.259 : 259 PS,  -0.199 : -199 PS );
           File: /home2/vlsi00/prz-root/icpro_teamspace/lpd21/workspaces/group05/baam21/resources/hpsnlib_g28_9t_RVT/verilog/hpsnlib_g28_9t_RVT.v, line = 14676
          Scope: tb_soc_top.soc_top_i.soc_i.core_v6_i.regfile_i.master_reg_i.Q_reg_23_
           Time: 223833827 PS

          224243000: EMUL_BUS | SOC_BUS_READ --> ADDR = 0x4000, SIZE = 0x2, DATA = 0x00000000 ||          0
          225453000: EMUL_BUS | SOC_BUS_WRITE --> ADDR = 0x4000, SIZE = 0x2, DATA = 0x0000000d ||         13
          225623000: EMUL_BUS | SOC_BUS_WRITE --> ADDR = 0x400c, SIZE = 0x2, DATA = 0x000002f0 ||        752
          225753000: SLAVE | Core Asleep, make woke | cycle        12996
          225783000: SLAVE | Received ACK for interrupt 0 | cycle        12999
          225793000: SLAVE | Asserting external interrupt_1 | cycle        13000
          225853000: SLAVE | Received ACK for interrupt_1 | cycle        13006
          226933000: EMUL_BUS | SOC_BUS_WRITE --> ADDR = 0x4000, SIZE = 0x2, DATA = 0x0000000b ||         11
          228363000: EMUL_BUS | SOC_BUS_READ --> ADDR = 0x4000, SIZE = 0x2, DATA = 0x0000000b ||         11
          229573000: EMUL_BUS | SOC_BUS_WRITE --> ADDR = 0x4000, SIZE = 0x2, DATA = 0x0000000d ||         13
          229743000: EMUL_BUS | SOC_BUS_WRITE --> ADDR = 0x400c, SIZE = 0x2, DATA = 0x000002f0 ||        752
          229873000: SLAVE | Core Asleep, make woke | cycle        13396
          229903000: SLAVE | Received ACK for interrupt 0 | cycle        13399
```

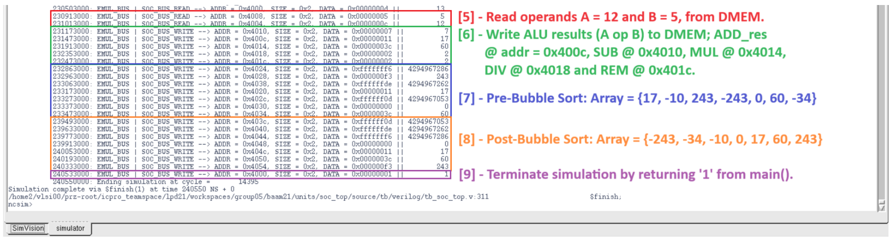
[1] - operand_A = 12 and operand_B = 5, written to DMEM.

[2] - Illegal Instruction puts core to sleep. Data = PC

[3] - Core woken up by external interrupt.

[4] - External interrupt asserted. Write DATA = 11 to DMEM for post-sim validation.

(a)



```
          230503000: EMUL_BUS | SOC_BUS_READ --> ADDR = 0x4000, SIZE = 0x2, DATA = 0x0000000d ||         13
          230913000: EMUL_BUS | SOC_BUS_READ --> ADDR = 0x4008, SIZE = 0x2, DATA = 0x00000005 ||          5
          231013000: EMUL_BUS | SOC_BUS_READ --> ADDR = 0x4004, SIZE = 0x2, DATA = 0x0000000c ||         12
          231173000: EMUL_BUS | SOC_BUS_WRITE --> ADDR = 0x4010, SIZE = 0x2, DATA = 0x00000007 ||          7
          231473000: EMUL_BUS | SOC_BUS_WRITE --> ADDR = 0x400c, SIZE = 0x2, DATA = 0x00000011 ||         17
          231913000: EMUL_BUS | SOC_BUS_WRITE --> ADDR = 0x4014, SIZE = 0x2, DATA = 0x0000003c ||         60
          232353000: EMUL_BUS | SOC_BUS_WRITE --> ADDR = 0x4018, SIZE = 0x2, DATA = 0x00000002 ||          2
          232473000: EMUL_BUS | SOC_BUS_WRITE --> ADDR = 0x401c, SIZE = 0x2, DATA = 0x00000002 ||          2
          232863000: EMUL_BUS | SOC_BUS_WRITE --> ADDR = 0x4024, SIZE = 0x2, DATA = 0xfffffff6 || 4294967286
          232963000: EMUL_BUS | SOC_BUS_WRITE --> ADDR = 0x4028, SIZE = 0x2, DATA = 0x000000f3 ||        243
          233063000: EMUL_BUS | SOC_BUS_WRITE --> ADDR = 0x4038, SIZE = 0x2, DATA = 0xffffffde || 4294967262
          233173000: EMUL_BUS | SOC_BUS_WRITE --> ADDR = 0x4020, SIZE = 0x2, DATA = 0x00000011 ||         17
          233273000: EMUL_BUS | SOC_BUS_WRITE --> ADDR = 0x402c, SIZE = 0x2, DATA = 0xffffff0d || 4294967053
          233373000: EMUL_BUS | SOC_BUS_WRITE --> ADDR = 0x4030, SIZE = 0x2, DATA = 0x00000000 ||          0
          233473000: EMUL_BUS | SOC_BUS_WRITE --> ADDR = 0x4034, SIZE = 0x2, DATA = 0x0000003c ||         60
          239493000: EMUL_BUS | SOC_BUS_WRITE --> ADDR = 0x403c, SIZE = 0x2, DATA = 0xffffff0d || 4294967053
          239633000: EMUL_BUS | SOC_BUS_WRITE --> ADDR = 0x4040, SIZE = 0x2, DATA = 0xffffffde || 4294967262
          239773000: EMUL_BUS | SOC_BUS_WRITE --> ADDR = 0x4044, SIZE = 0x2, DATA = 0xfffffff6 || 4294967286
          239913000: EMUL_BUS | SOC_BUS_WRITE --> ADDR = 0x4048, SIZE = 0x2, DATA = 0x00000000 ||          0
          240053000: EMUL_BUS | SOC_BUS_WRITE --> ADDR = 0x404c, SIZE = 0x2, DATA = 0x00000011 ||         17
          240193000: EMUL_BUS | SOC_BUS_WRITE --> ADDR = 0x4050, SIZE = 0x2, DATA = 0x0000003c ||         60
          240333000: EMUL_BUS | SOC_BUS_WRITE --> ADDR = 0x4054, SIZE = 0x2, DATA = 0x000000f3 ||        243
          240533000: EMUL_BUS | SOC_BUS_WRITE --> ADDR = 0x4000, SIZE = 0x2, DATA = 0x00000001 ||          1
          240550000: Ending simulation at cycle =        14395
Simulation complete via $finish(1) at time 240550 NS + 0
/home2/vlsi00/prz-root/icpro_teamspace/lpd21/workspaces/group05/baam21/units/soc_top/source/tb/verilog/tb_soc_top.v:311             $finish;
ncsim>
```

[5] - Read operands A = 12 and B = 5, from DMEM.

[6] - Write ALU results (A op B) to DMEM; ADD_res @ addr = 0x400c, SUB @ 0x4010, MUL @ 0x4014, DIV @ 0x4018 and REM @ 0x401c.

[7] - Pre-Bubble Sort: Array = {17, -10, 243, -243, 0, 60, -34}

[8] - Post-Bubble Sort: Array = {-243, -34, -10, 0, 17, 60, 243}

[9] - Terminate simulation by returning '1' from main().

(b)

Figure 5.5: Console output of post PnR simulation of SoC design (a) Part-1 testing, memory store/load, illegal_instruction, reset and interrupt vectors. (b) Part-2 testing ALU and core functionality.

A summary of the implemented SoC operating at F_{op} = 100*MHz*, using the *RV32IM* RISC-V ISA, is shown in table 5.3.

Parameter	Value	Unit
Max. Frequency (F_{max})	181.818	MHz
Total Power (P_{total}) [@ F_{op} = 100*MHz*]	283.025	uW
Total Area (A_{total}) [@ F_{op} = 100*MHz*]	350313.235	um^2
Multiplication	18	Latency (Cycles)
Division	35	Latency (Cycles)
Internal Load/Store	2	Latency (Cycles)
External Load/Store (min)	7	Latency (Cycles)

Table 5.3: Summary of the SoC's operational parameters.

5.4 Conclusion and Future work

A SoC is designed to implement the RISC-V ISA - *RV32IM*, while meeting the requirements specified in section 1.5. The implementation is optimized for power consumption and achieves a 37.942% power reduction at F_{op} = 100*MHz* compared to the baseline version with no clock-gating applied. This work obtains a low power consumption at the cost of increased latency during multiplication, division and external-bus access.

The implementation shows a lot of room for improvement in the IO_Pads and Memory department when concerned with power utilization. In-addition, the core-utilization of this design is a concern since there is a lot of un-utilized (free) space in the core, this could be improved by using smaller IO_Pads (aspect ratio reduction), or by implementing more logic within the core (increase core functionality).

Bibliography

[1] et.al Andrew Waterman Krste Asanovi. “The RISC-V Instruction Set Manual Volume II: Privileged ISA”. In: CS Division, EECS Department, University of California, Berkeley: SiFive Inc., 2016.

[2] Krste Asanovi Andrew Waterman. “The RISC-V Instruction Set Manual Volume I: User-Level ISA”. In: CS Division, EECS Department, University of California, Berkeley: SiFive Inc., 2017.

[3] Grigorios Magklis Antonio González Fernando Latorre. “Processor Microarchitecture”. In: Springer Nature Switzerland AG: Springer Cham, 2022. DOI: https://doi.org/10.1007/978-3-031-01729-2.

[4] ANDREW D. BOOTH. “A SIGNED BINARY MULTIPLICATION TECHNIQUE”. In: *The Quarterly Journal of Mechanics and Applied Mathematics* 4.2 (Jan. 1951), pp. 236–240. ISSN: 0033-5614. DOI: 10.1093/qjmam/4.2.236. eprint: https://academic.oup.com/qjmam/article-pdf/4/2/236/5301697/4-2-236.pdf.

[5] Tamil Chindhu S. and N. Shanmugasundaram. “Clock Gating Techniques: An Overview”. In: *2018 Conference on Emerging Devices and Smart Systems (ICEDSS)*. 2018, pp. 217–221. DOI: 10.1109/ICEDSS.2018.8544281.

[6] TU-Dresden. “HPSNLIB: Standard-cell Library for synthesis.” In: *"https://tu-dresden.de/"* (2020).

[7] Fabrizio Gagliardi et al. “The international race towards Exascale in Europe”. In: *CCF Transactions on High Performance Computing* 1 (Apr. 2019). DOI: 10.1007/s42514-019-00002-y.

[8] Anantha Sri Purnima Gunturu. “Analysis of Booth&#x2019;s Multiplier Algorithm vs Array Multiplier Algorithm and their FPGA Implementation”. PhD thesis. 2019.

[9] Sarah L. Harris and David Money Harris. “7 - Microarchitecture”. In: *Digital Design and Computer Architecture*. Ed. by Sarah L. Harris and David Money Harris. Boston: Morgan Kaufmann, 2016, pp. 384–484. ISBN: 978-0-12-800056-4. DOI: https://doi.org/10.1016/B978-0-12-800056-4.00007-8.

[10] Sukhmeet Kaur, Suman, and M. S. Manna. “Implementation of Modified Booth Algorithm (Radix 4) and its Comparison with Booth Algorithm (Radix-2)”. In: 2013.

[11] Processor Design Lab. “Dobby RISC-V SoC Task Description”. In: Technical University, Dresden: Chair of Highly-Parallel VLSI Systems and Neuro-Microelectronics, 2020.

[12] Dadda. Luigi. “Some schemes for parallel multipliers”. In: *Alta Frequenza* 34.5 (1965), pp. 349–356.

[13] Paul Metzgen. “A high performance 32-bit ALU for programmable logic”. In: *Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*. FPGA ’04. Monterey, California, USA: Association for Computing Machinery, 2004, pp. 61–70. ISBN: 1581138296. DOI: 10.1145/968280.968291.

[14] Simon Rokicki et al. “What You Simulate Is What You Synthesize: Designing a Processor Core from C++ Specifications”. In: *ICCAD 2019 - 38th IEEE/ACM International Conference on Computer-Aided Design*. Westminster, CO, United States: IEEE, Nov. 2019, pp. 1–8.

[15] C. S. Wallace. “A Suggestion for a Fast Multiplier”. In: *IEEE Transactions on Electronic Computers* EC-13.1 (1964), pp. 14–17. DOI: 10.1109/PGEC.1964.263830.

[16] C. Wolf. “PicoRV32: A Size-Optimized RISC-V CPU”. In: ().

[17] M. Wroblewski et al. “A power efficient register file architecture using master latch sharing”. In: *Proceedings of the 2003 International Symposium on Circuits and Systems, 2003. ISCAS ’03.* Vol. 5. 2003, pp. V–V. DOI: 10.1109/ISCAS.2003.1206291.

6 Appendix

Definitions of custom metrics used in evaluation of this work, related software and methodology used to produce said results are described below.

6.1 Signal Integrity (SI)

SI affects the delay through a net by either increasing its delay (setup violation) or reducing its delay (hold violation).

To better understand the effect of SI on STA, lets imagine a net with a weak driver (victim net) surrounded by nets with stronger drivers (aggressor nets). Now, consider two scenarios in which:

- All nets (victim and aggressor) switch from low to high around the same time. Then, the aggressor nets can help pull the victim nets high, faster than normal, decreasing the victim net's transmission delay.
- All aggressor nets switch from low to high while the victim is switching from high to low. Then, the aggressors can fight against the victim net making it harder to switch, resulting in increased transmission delay through the victim net.

Now, how much an aggressor net can influence the victim is determined from their drive strengths and coupling capacitance between the nets.

6.2 Core-Utilization Density

Core-Utilization Density defines the area occupied by standard cells, macros and blockages. In general 70 8̃0% of the core utilization is fixed because more number of inverters and buffers will be added during the process of *Clock-Tree-Synthesis* (CTS) section 6.3 in order to maintain minimum skew on the clock tree.

$$CoreUtilization = \frac{StandardCellArea + MacroCellsArea}{TotalCoreArea}$$

Hence, a core utilization of 0.8 means that 80% of the area is available for placement of cells and macros, whereas 20% is left free for routing.

6.3 Clock-Tree-Synthesis (CTS)

Clock-Tree-Synthesis is the technique of balancing *clock-delay* to all clock inputs by inserting *buffer/inverters* along the clock routes. It is used to balance *clock-skew* and *insertion-latency* along the clock path.