End-to-End RTL to GDSII Design and Verification of SAP Processor Architectures

Submitted By

K.M. Amir Khasru	EEE2202026091
Md. Shahidul Haque Bhuiyan	EEE2202026003
Orin Gabrial Costa	EEE2202026065
Atiya Akter	EEE2202026004
Ripon Ahmed	EEE2001019006

Supervisor

Mirza Rabiul Hasan
Lecturer, Department of EEE
Sonargaon University

Department of Electrical & Electronics Engineering
Sonargaon University (SU)
147/I, Panthapath,
Dhaka-1215,

Date of Submission: January 2026

DECLARATION

We, the undersigned students of the Department of Electrical and Electronics Engineering (EEE), Sonargaon University, hereby declare that this project report entitled "**End-to-End RTL to GDSII Design and Verification of SAP Processor Architectures**" is the result of our own work carried out under the supervision of **Mirza Rabiul Hasan**, Lecturer, Department of Electrical and Electronics Engineering, Sonargaon University.

This project has not been submitted previously, in whole or in part, for the award of any degree or professional qualification at any university or institution. All sources of information used in this project have been properly acknowledged and referenced in accordance with academic standards.

We further declare that any resemblance to existing works is purely coincidental and unintentional. The work presented in this project is original and was conducted solely for academic purposes.

K.M. Amir Khasru
EEE2202026091

Md. Shahidul Haque Bhuiyan
EE2202026003

Orin Gabrial Costa
EEE2202026065

Atiya Akter
EE2202026004

Ripon Ahmed
EEE2001019006

CERTIFICATION

This is to certify that the project report entitled **"End-to-End RTL to GDSII Design and Verification of SAP Processor Architectures"** has been carried out by the following students of the Department of Electrical and Electronics Engineering (EEE), Sonargaon University, under my supervision:

K.M. Amir Khasru	EEE2202026091
Md. Shahidul Haque Bhuiyan	EEE2202026003
Orin Gabrial Costa	EEE2202026065
Atiya Akter	EEE2202026004
Ripon Ahmed	EEE2001019006

The work presented in this report is original and has been carried out in accordance with the rules and regulations of Sonargaon University. I believe it meets the standards required for submission and evaluation.

Supervisor

……………………………

Mirza Rabiul Hasan

Lecturer, Department of Electrical and Electronics Engineering

Sonargaon University

ACKNOWLEDGEMENT

First and foremost, we express our sincere gratitude to **Mirza Rabiul Hasan**, Lecturer, Department of Electrical and Electronics Engineering, Sonargaon University, for his invaluable guidance, encouragement, and continuous support throughout the course of this project. His expert advice and constructive feedback were instrumental in completing this work successfully.

We would like to thank our friends and course mates for their cooperation, helpful discussions, and support during the practical aspects of this project.

Finally, we extend our gratitude to all the faculty members of the Department of Electrical and Electronics Engineering for their guidance, inspiration, and encouragement, which contributed greatly to the successful completion of this project.

ABSTRACT

The main objective of this project is to design and verify SAP processor architectures through a complete RTL-to-GDSII design flow. This work demonstrates how a digital processor can be implemented starting from a high-level hardware description down to a physical layout ready for fabrication.

The SAP processor was developed using Verilog HDL, and its functionality was validated using comprehensive simulations and testbench verification. After confirming the design's correctness, the RTL was synthesized and transformed into a GDSII layout, highlighting the challenges of timing, area optimization, and functional accuracy in physical design.

This project provides practical exposure to processor design, hardware verification, and VLSI physical implementation, and serves as a foundation for further research on advanced processor architectures. The results show that the proposed methodology is effective for producing a reliable and verifiable processor design.

TABLE OF CONTENTS

Declaration…………………………………………………………………………..2
Certificateion……………………………………………………………………..3
Acknowledgement………………………………………………………………..4
Abstract……………………………………………………………………………...5
Chapter 1: Introduction……………………………………………………………...9
1.1 Background……………………………………………………………9
1.2 Objective of The Project……………………………………………….....9
1.3 Von Neumann Architecture……………………………………………10
1.4 Simple Computer Architectures………………………………………10-11
Chapter 2: SAP-1 Architecture………………………………………………………12
2.1 SAP-1 Block Diagram…………………………………………………12
2.2 SAP-1 Components…………………………………………………....13-17
2.3 SAP-1 Instruction Set…………………………………………………..17
2.4 SAP-1 Sub-Instructions (Micro-Operations) ……………………………..18
2.5 Clock………………………………………………………………….....19
Chapter 3: SAP-2 Architecture……………………………………………………….20
3.1 SAP-2 Block Diagram……………………………………………….…20
3.2 SAP-2 Components…………………………………………………....21-24
3.3 SAP-2 Instruction Set……………………………………………….…24-25
3.4 SAP-2 Addressing Modes……………………………………………..25-26
Chapter 4: SAP-3 Architecture………………………………………………….....27
4.1 SAP-3 Block Diagram…………………………………………………27
4.2 SAP-3 Components………………………………………………….....28-29
4.3 SAP-3 Instruction Set………………………………..…………...30-32
Chapter 5: Verification & Design…………………………………………32-47

Chapter 6: Discussion & Conclusion……………………………………………..38-39
6.1 Conclusion……………………………………………………………...38
6.2 Limitation……………………………………………………………38
6.3 Future Scopes………………………………………………………...39
References…………………………………………………………………...40-41
Appendix…………………………………………………………………42-91

LIST OF FIGURES

Figure 2.1: Block Diagram of SAP-1…………………………………………...12
Figure 3.1: Block Diagram of SAP-2…………………………………………...20
Figure 4.1: Block Diagram of SAP-3…………………………………………...27
Figure 5.1: SAP-1 RTL Verification……………………………………………….32
Figure 5.2: SAP-1 Netlist Simulation…………………………………………..….33
Figure 5.3: Simulation of Timing Power Check SAP-1……………………..….33
Figure 5.4: SAP-1 Physical Design……………………………………………..33
Figure 5.5: SAP-2 RTL Verification……………………………………………….34
Figure 5.6: SAP-2 Netlist Simulation…………………………………………..….35
Figure 5.7: Simulation of Timing Power Check SAP-2……………………..….35
Figure 5.8: SAP-2 Physical Design……………………………………………..35
Figure 5.9: SAP-3 RTL Verification……………………………………………….36
Figure 5.10: SAP-3 Netlist Simulation…………………………………..…...37
Figure 5.11: Simulation of Timing Power Check SAP-3……………….…...37
Figure 5.12: SAP-3 Physical Design…………………………………………….37

LIST OF TABLES

Table 2.3: SAP-1 Instruction Set……………………………………………..17
Table 2.4: SAP-1 Fetch and Execution Cycle……………………………………...18

Chapter 1: Introduction

1.1 Background

The continuous advancement of semiconductor technology has increased the demand for efficient and reliable digital integrated circuits. Modern VLSI design follows a structured design methodology known as the RTL to GDSII flow, which transforms a high-level functional description of a system into a physical layout ready for fabrication. This design flow involves multiple stages, including RTL design, functional verification, logic synthesis, physical implementation, and sign-off verification, each of which plays a critical role in ensuring the correctness and manufacturability of the final chip.

Processor architectures form the backbone of digital systems, and understanding their design and implementation is essential for students of electronics and VLSI engineering. The SAP (Simple As Possible) processor architecture, originally introduced for educational purposes, provides a simplified yet complete processor model that includes essential components such as the arithmetic logic unit, control unit, registers, and memory interface. Due to its modular and minimal design, the SAP processor serves as an ideal platform for learning core concepts of processor operation and hardware design.

While processor architectures are often studied at a theoretical or RTL level, practical exposure to the complete RTL to GDSII flow is limited in academic environments. Implementing the SAP processor through this full design flow allows students to gain hands-on experience with real-world ASIC design methodologies, industry-standard tools, and verification techniques. This project addresses the need to connect conceptual processor design with physical chip implementation by demonstrating an end-to-end VLSI design approach.

1.2 Objective of The Project

The primary objectives of this project are as follows:

- To design and model SAP processor architectures at the Register Transfer Level (RTL) using a hardware description language.
- To perform functional verification of the RTL design through simulation and ensure correct processor operation.
- To synthesize the verified RTL into a gate-level netlist while meeting timing and area constraints.
- To implement the physical design flow, including floorplanning, placement, clock tree synthesis, and routing.
- To generate a GDSII layout that is compliant with design rules and suitable for fabrication.
- To perform verification checks such as Static Timing Analysis (STA), Design Rule Check (DRC), and Layout Versus Schematic (LVS).
- To gain practical understanding of the complete RTL to GDSII design and verification flow for processor architectures.

1.3 Von Neumann Architecture

The Von Neumann architecture is a computer design model where both program instructions and data are stored in the same memory. It consists of five main components:

- Memory
- Arithmetic Logic Unit (ALU)
- Control Unit
- Input Unit
- Output Unit

In this architecture, instructions are fetched sequentially from memory, decoded, and executed. SAP-based computers follow the Von Neumann model, making them ideal examples for understanding classical computer organization.

1.4 Simple Computer Architectures

Simple computer architectures are designed to execute basic computational operations while maintaining minimal hardware complexity. These architectures typically include fundamental components such as the Arithmetic Logic Unit (ALU), control unit, registers, memory, and input/output interfaces. The instruction set is limited and easy to decode, allowing the processor to perform operations in a sequential and predictable manner.

Such architectures generally operate using a basic instruction cycle consisting of instruction fetch, decode, and execution stages. Due to their straightforward control logic and limited functionality, simple computer architectures are easy to design, simulate, and verify. This makes them highly suitable for educational applications, where the focus is on understanding core concepts rather than achieving high performance.

The SAP (Simple As Possible) processor is a representative example of a simple computer architecture. It demonstrates essential processor operations such as data movement, arithmetic computation, and control flow using a minimal instruction set. In this project, simple computer architectures serve as an effective platform for exploring the complete RTL to GDSII design flow, helping bridge the gap between theoretical processor concepts and practical VLSI implementation.

Chapter 2: SAP-1 Architecture

The SAP-1 computer is a bus-organized computer that follows the Von Neumann architecture. It utilizes a single 8-bit bidirectional central bus (commonly referred to as the W-bus) for all data transfers and consists of ten major components. A block diagram illustrating its architecture is shown in Figure 2.1 below.

2.1 SAP-1 Block Diagram

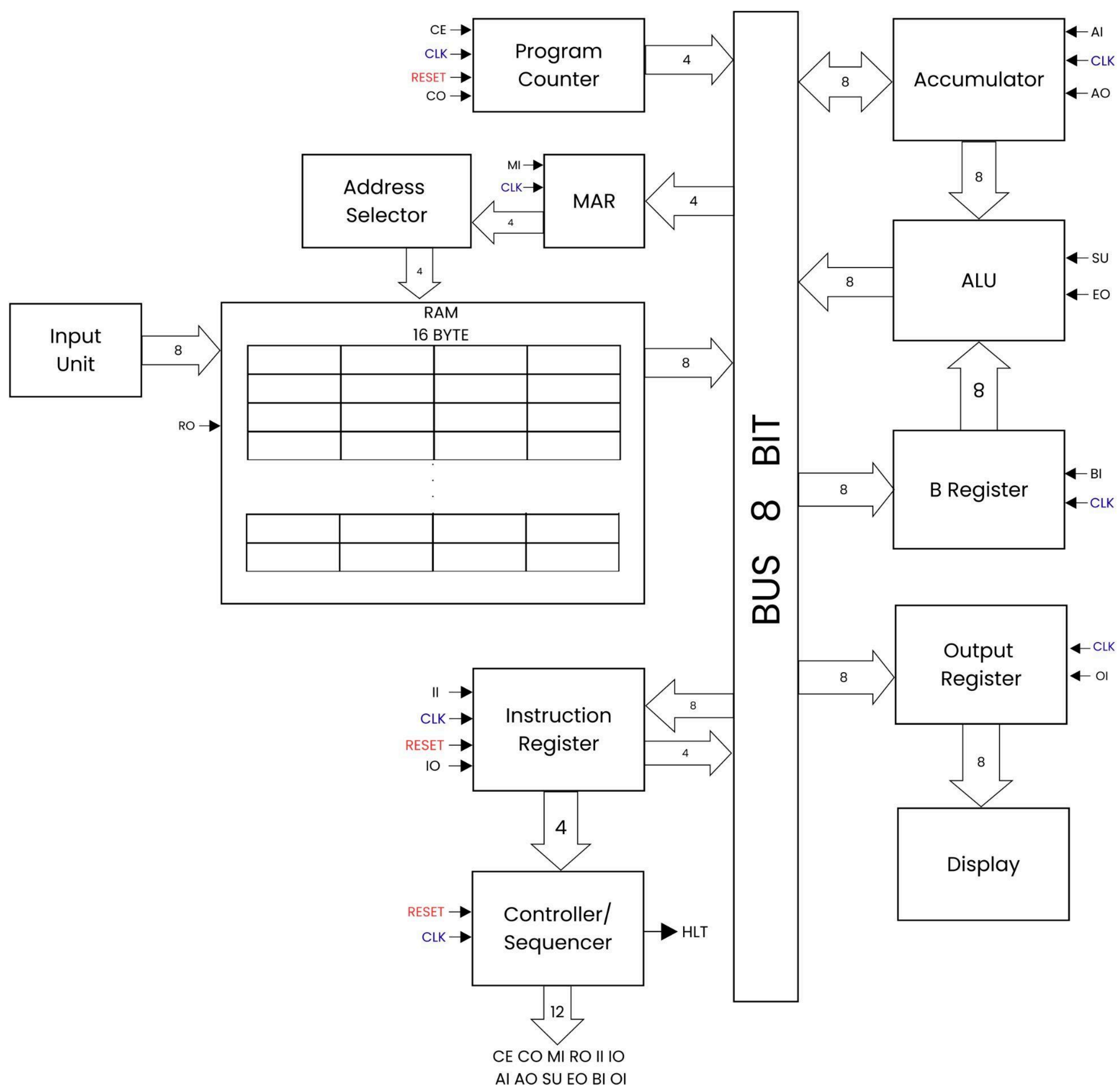


Figure 2.1: Block Diagram of SAP-1

2.2 SAP-1 Components

Program Counter:

The program counter (PC) stores and provides the memory address of the next instruction to be fetched and executed. It is part of the control unit and counts sequentially from 0000 to 1111, corresponding to the memory addresses where the SAP-1 program is stored. The first instruction is at address 0000, the second at 0001, the third at 0010, and so on. At the start of each run, the PC is reset to 0000. When the computer starts, the PC sends the address 0000 to memory, fetches the first instruction, and then increments by 1. After executing the first instruction, it sends 0001 for the next instruction, increments again, and continues this process throughout the program, ensuring the computer always knows which instruction to fetch next.

Pins/Signals:

- CE (Counter Enable): Increments the program Counter by 1 on the next clock cycle.
- CLK (Clock): Clock pulse input.
- RESET: Resets the program counter to 0000
- CO (Counter Out): Puts the current program counter value on the bus. Output is disconnected when CO is low.

Memory Address Register (MAR):

The Memory Address Register (MAR) is a 4-bit register. It stores the memory address of the RAM location from which data or instructions will be fetched. This 4-bit address is received via the bus from the Program Counter (PC) and then loaded into the MAR. The stored address is then sent to the RAM to select the specific memory location for reading data or instructions.

Pins/Signals:

- MI (MAR In): It loads (stores) the current value present on the bus into the Memory Address Register (MAR).
- CLK (Clock): Clock pulse input.

Address Selector:

The Address Selector is a multiplexer that decides which 4-bit address source will be sent to the Memory Address Register. It chooses between the output of the Program

Counter (used for fetching the next instruction) and the lower 4 bits of the Instruction Register (used when an instruction like LDA, ADD, or SUB needs to access a specific memory location for data). This selection is controlled by the timing signals from the Controller.

Input Unit:

The Input Unit allows the user to manually enter both program instructions and data into the RAM before the computer starts running. It typically consists of switches or a small keypad that provides an 8-bit data value and a 4-bit address. When the write signal is active, the Input Unit places the selected data into the specified memory location.

Switches of the input unit:

- Data Switch: 8 switches (D7-D0) for writing 8-bit data to RAM.
- R/W Switch: For switching between read and write.

Random Access Memory (RAM):

The RAM is the main memory of the SAP-1 computer. It contains 16 locations, each 8 bits wide, giving a total memory size of 16 bytes. Both program instructions and data are stored in the same memory space following the Von Neumann architecture. The RAM receives a 4-bit address from the MAR and can read data onto the bus or accept new data from the bus or the Input Unit, depending on the read/write control.

Pins/Signals:

- RO (RAM Out): When active, it places RAM data on the bus. Output is disconnected when RO is low.

Instruction Register (IR):

The Instruction Register (IR) is an 8-bit register that stores the complete 8-bit instruction fetched from the RAM. This 8-bit data is then split into two 4-bit nibbles. The upper nibble (most significant 4 bits), which contains the opcode, is sent to the controller-sequencer for decoding and generating the appropriate control signals. The lower nibble (least significant 4 bits), which usually represents the address or operand, is placed onto the bus when needed.

Pins/Signals:

- Instruction Register In (II): When active, it loads (writes) the current value present on the bus into the Instruction Register.
- Instruction Register Out (IO): When active, it sends only the stored lower nibble (the 4 least significant bits) from the Instruction Register onto the bus.

Controller / Sequencer:

The Controller-Sequencer is the central timing and coordination unit of the SAP-1 computer. It generates the necessary control signals for each block so that all actions occur in the desired sequence. It produces a total of 12 control bits that come out of the controller-sequencer block. These 12 control bits determine how all the other blocks will respond to the next positive clock edge. In addition, the controller-sequencer generates a special HLT (halt) signal, which can stop the operation of the computer by disabling the main clock when the HLT instruction is executed.

Pins/Signals:

- HLT (Halt): Input from HLT opcode; stops the clock or sequencer when active.
- Outputs: 12 control lines (CE, CO, MI, RO, II, IO, AI, AO, SU, EO, BI, OI) to all components.

Accumulator (A Register):

The Accumulator is the main 8-bit working register of the SAP-1. It holds the result of all arithmetic operations and is the primary location where data is processed. During execution of ADD or SUB instructions, one operand comes from the Accumulator while the other comes from memory via the B Register. The Accumulator can place its contents onto the bus when needed or load new values from the bus (usually from the ALU output).

Pins/Signals:

- AI (Accumulator In): Loads data from the bus into the Accumulator.
- AO (Accumulator Out): Accumulator sends its stored data to the bus.

ALU (Adder/Subtractor):

The ALU is a simple 8-bit arithmetic unit that can only perform addition or subtraction. It takes two 8-bit inputs: one from the Accumulator and one from the B Register. Depending on the SU control signal, it either adds the two values or subtracts

the B Register value from the Accumulator value. The result is placed on the bus only when the output enable signal is active.

Pins/Signals:

- SU (Subtract): Mode select; high for subtract, low for add.
- EO (Enable Out): Puts the ALU output onto the bus. The output of the ALU should be disconnected when the RO is low.

B Register:

The B Register is an 8-bit temporary storage register. During ADD and SUB instructions, it holds the second operand that was fetched from memory. The B Register loads its value directly from the bus when instructed and continuously feeds its contents to one input of the ALU. It does not have its own output enabled on the main bus.

Pins/Signals:

- BI (B Register In): Loads the current value from the bus into the B register.

Output Register:

The Output Unit consists of an 8-bit register known as the Output Register. Its main purpose is to store the result from the Accumulator for display when requested by the OUT instruction. When the output needs to be shown, the Output Register receives the data directly from the Accumulator via the bus. This register can be connected to various display modules, such as LEDs, to visually represent the output value in binary.

Control signal:

- OI (Output Register In): When active, loads the current value present on the bus into the Output Register.

Binary Display:

The Binary Display is simply an output device consisting of eight LEDs (or a small LED panel) connected directly to the Output Register. It visually represents the 8-bit value stored in the Output Register in binary form, allowing the user to see the final result of any program that uses the OUT instruction. It has no active control signals of its own.

2.3 SAP-1 Instruction Set

The instruction set of a computer is the basic operations it can perform. The instruction set of the SAP-1 is described in Table 2.3 below.

Operation	Description
LDD	Load RAM data into the accumulator
ADD	Add RAM data to the accumulator
SUB	Subtract RAM data from the accumulator
OUT	Load accumulator data into the output register
HLT	Stop processing

Table 2.3: SAP-1 Instruction Set

2.4 SAP-1 Sub-Instructions (Micro-Operations)

In the SAP-1 architecture, the high-level instructions (macro-instructions) like LDA, ADD, SUB, OUT, and HLT are not executed in a single step. Instead, each macro-instruction is broken down into a sequence of smaller, atomic steps called micro-operations or sub-instructions.

These micro-operations are executed one per clock cycle (T-state) and are controlled by the 12-bit control word generated by the Controller/Sequencer.
SAP-1 uses a fixed 6 T-state cycle (T1-T6) for every instruction:

Cycle	Operation	State
Fetch	All Operation	T1
		T2
		T3
Execution	Load	T4
		T5
		T6
	Add	T4
		T5
		T6
	Subtract	T4
		T5
		T6
	Output	T4
		T5
		T6

Table 2.4: SAP-1 Fetch and Execution Cycle

2.5 Clock (in SAP-1 Architecture)

The clock is a fundamental component of the SAP-1 computer that controls the timing of all operations. It generates a continuous sequence of clock pulses that synchronize the activities of all registers, the ALU, memory, and the control unit. Each clock pulse represents a fixed time interval during which a specific operation or sub-instruction is performed.

In SAP-1, instruction execution is divided into several steps called micro-operations, and each micro-operation is completed in one clock cycle. On every rising edge of the clock, data is transferred between registers or processed by the ALU as directed by the control unit. This ensures that all components operate in an orderly and coordinated manner.

The clock also plays an important role in sequencing instructions. It allows the control unit to move from the fetch cycle to the execute cycle and then to the next instruction. Without the clock, the SAP-1 system would not be able to maintain proper timing, and reliable instruction execution would not be possible.

Chapter 3: SAP-2 Architecture

The Simple-As-Possible-2 (SAP-2) architecture is an enhanced educational microcomputer model developed to overcome the limitations of SAP-1 while still maintaining simplicity and clarity. SAP-2 introduces additional registers, an expanded addressing mechanism, improved control logic, and greater instruction flexibility. These improvements allow SAP-2 to demonstrate more realistic CPU behavior while remaining easy to understand.

3.1 SAP-2 Block Diagram

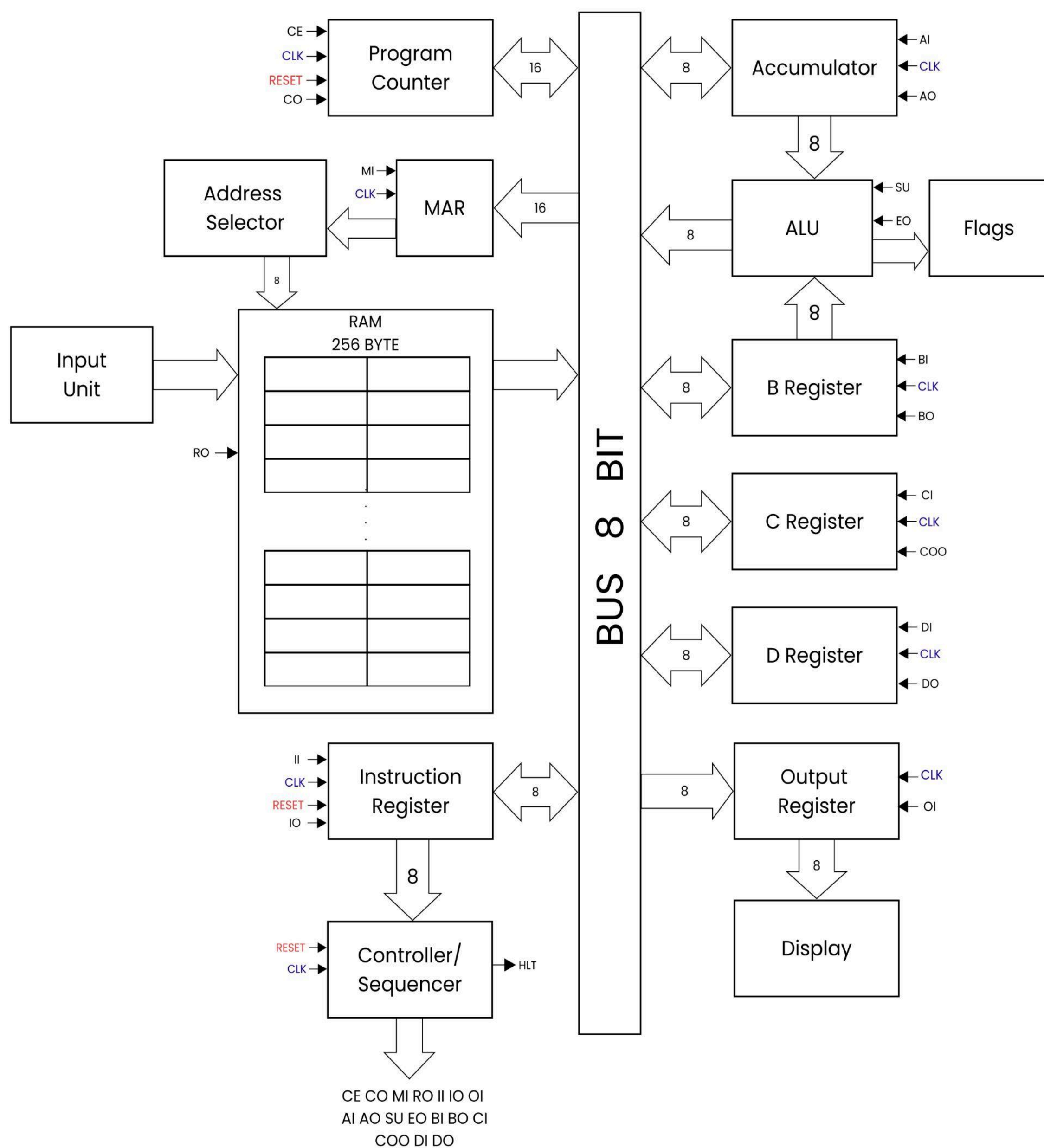


Figure 3.1: Block Diagram of SAP-2

3.2 SAP-2 Components

Program Counter:

The Program Counter is responsible for holding the address of the next instruction to be fetched from memory. In SAP-2, the Program Counter is a 16-bit register, allowing the processor to address a larger memory range compared to SAP-1. During normal execution, the Program Counter automatically increments after each instruction fetch, ensuring sequential execution of instructions. The contents of the Program Counter can be placed onto the address bus when required, enabling instruction fetching. A reset signal clears the Program Counter and returns it to the starting address, ensuring proper system initialization.

Memory Address Register (MAR):

The Memory Address Register temporarily stores the address of the memory location that the processor intends to access. This address may originate from the Program Counter during instruction fetch or from another internal source during data access operations. The MAR directly interfaces with the RAM address lines and isolates the address bus from the rest of the CPU. By using a dedicated address register, SAP-2 ensures stable and accurate memory access during both read and write operations.

Address Selector:

The Address Selector is a control block that determines the source of the memory address loaded into the MAR. In SAP-2, memory access is required not only for instruction fetching but also for data manipulation instructions. The Address Selector enables flexible address selection by choosing between the Program Counter and other internal address sources. This design improves instruction execution capability and supports more complex program behavior.

Random Access Memory (RAM):

SAP-2 includes 256 bytes of Random Access Memory, which stores both program instructions and data. Memory locations are accessed using the address provided by the MAR, and data transfer occurs through the 8-bit data bus. When a read operation is enabled, the contents of the selected memory location are placed onto the bus so that they can be loaded into registers such as the Instruction Register or Accumulator.

This unified memory organization simplifies the architecture and reflects the principles of basic computer design.

Instruction Register:

The Instruction Register stores the instruction currently being executed by the processor. Once an instruction is fetched from memory, it is loaded into the Instruction Register and remains there throughout the execution cycle. The opcode portion of the instruction is forwarded to the Controller for decoding, while the operand portion can be placed onto the data bus when required. The Instruction Register ensures that the instruction remains stable and unchanged during execution, enabling correct control signal generation.

Controller/Sequencer:

The Controller, also known as the Sequencer, is the central control unit of the SAP-2 architecture. It decodes the opcode received from the Instruction Register and generates a sequence of control signals based on the system clock. These control signals coordinate all internal operations, including register loading, bus enabling, ALU operation selection, memory access, and input/output control. The Controller also manages system reset and halt operations. By executing instructions through a series of timed micro-operations, the Controller ensures orderly and synchronized processor behavior.

Accumulator Register:

The Accumulator is the primary working register of the SAP-2 processor. It stores intermediate data and final results of arithmetic and logic operations. One input of the ALU is permanently connected to the Accumulator, making it central to data processing. The Accumulator can receive data from the system bus and can also place its contents onto the bus when required. Due to its frequent use, the Accumulator plays a vital role in almost every instruction execution.

General-Purpose Registers (B, C, and D Registers):

The SAP-2 architecture includes multiple general-purpose registers to enhance processing capability. The B Register is mainly used to hold the second operand for ALU operations, while the Accumulator provides the first operand. This separation improves clarity and efficiency during arithmetic and logic operations. The C and D

Registers serve as additional storage locations for temporary data, memory values, or intermediate results. The inclusion of these registers reduces dependency on memory access and allows more structured and flexible instruction execution.

Flags Register:

The Flags Register stores status information generated by the ALU after each operation. These status bits indicate conditions such as whether the result is zero or whether a carry has occurred. The stored flags are useful for implementing conditional operations and decision-making instructions. Although simple, the Flags Register adds significant functional value to the processor.

Input Unit:

The Input Unit allows external data to be introduced into the SAP-2 system. This unit is typically connected to switches or other manual input devices and places data onto the system bus when enabled. The Input Unit is primarily used for testing, experimentation, and demonstration purposes in educational environments.

Output Register and Display:

The Output Register stores data that is intended to be sent outside the CPU. Once data is loaded into the Output Register from the system bus, it remains stable until updated. The Display Unit connected to the Output Register presents the output in a visible form, such as LEDs or a seven-segment display. This subsystem allows users to observe the results of program execution clearly.

Clock and Reset System:

The Clock signal synchronizes all sequential operations in the SAP-2 architecture. Register loading, control signal execution, and instruction sequencing all depend on the clock. The Reset signal initializes the system by clearing registers and returning the processor to a known starting state. Together, the clock and reset system ensure stable, reliable, and predictable operation.

3.3 SAP-2 Instruction Set

Memory Reference Instructions:

- LDA-Load the Accumulator
- STA-Store the Accumulator
- MVI-Move Immediate for immediate data transfer

Register Instructions:

- MOV (Move) for register data transfer
- INR (Increment) and DCR (Decrement) for register manipulation
- ADD and SUB for arithmetic operations

Jump and Call Instructions:

- JMP (Jump) and JM (Jump if minus) for program flow control
- JZ (Jump if zero) and JNZ (Jump if not zero) based on flag conditions
- CALL (Call) and RET (Return) for subroutine handling

Logic Instructions:

- CMA-Complement the accumulator
- ANA-And the accumulator with specified register
- XRA-XOR the accumulator with specified register
- ORA-OR the accumulator with specified register
- ANI-And immediate
- ORI-OR immediate
- XRI-XOR immediate

Other Instructions:

- IN-Input
- OUT-Output
- HLT-Stop processing
- NOP-No operation
- RAL-Rotate the accumulator left
- RAR-Rotate the accumulator right

3.4 SAP-2 Addressing Modes

Addressing modes are techniques used by the CPU to determine the location of the operand(s) required to execute an instruction. These modes establish rules for interpreting the address field in an instruction, assisting the CPU in fetching the operands accurately.

- **Opcode** - Indicates the operation the CPU should perform (e.g., ADD, MOV).
- **Operands -** The data or addresses on which this operation is executed.

Direct Addressing Mode:

In direct addressing mode, the operand field of the instruction contains the memory address of the data to be accessed. The CPU directly uses this address to fetch the operand from memory.

Example Instruction: LDA 2050H (Load the contents of memory location 2050H into the accumulator A).

Immediate Addressing Mode:

The actual operand (data) is included within the instruction itself. This is fast because no memory reference other than the instruction fetch is required to obtain the operand.

Example Instruction: MVI A, 2ah (Move the immediate value 2ah into the accumulator A).

Register Addressing Mode:

The operand is located in a CPU register, and the instruction specifies which register to use. This is very fast as all operations occur within the CPU's registers.

Example Instruction: MOV A, B (Move the content of register B to register A).

Implied Addressing Mode:

The operand is implicitly defined by the opcode itself, and no address field is needed in the instruction. Instructions operating on the accumulator are often in this mode.

Example Instruction: CMA (Complement the accumulator A).

Chapter 4: SAP-3 Architecture

SAP-3 (Simple-As-Possible-3) is an advanced version of the SAP computer family, designed to overcome the limitations of SAP-1 and SAP-2. While SAP-1 is mainly intended for instructional purposes and SAP-2 adds basic programmability, SAP-3 introduces more realistic CPU features, such as a richer instruction set, improved control unit, better memory addressing, and enhanced data processing capabilities.

4.1 SAP-3 Block Diagram

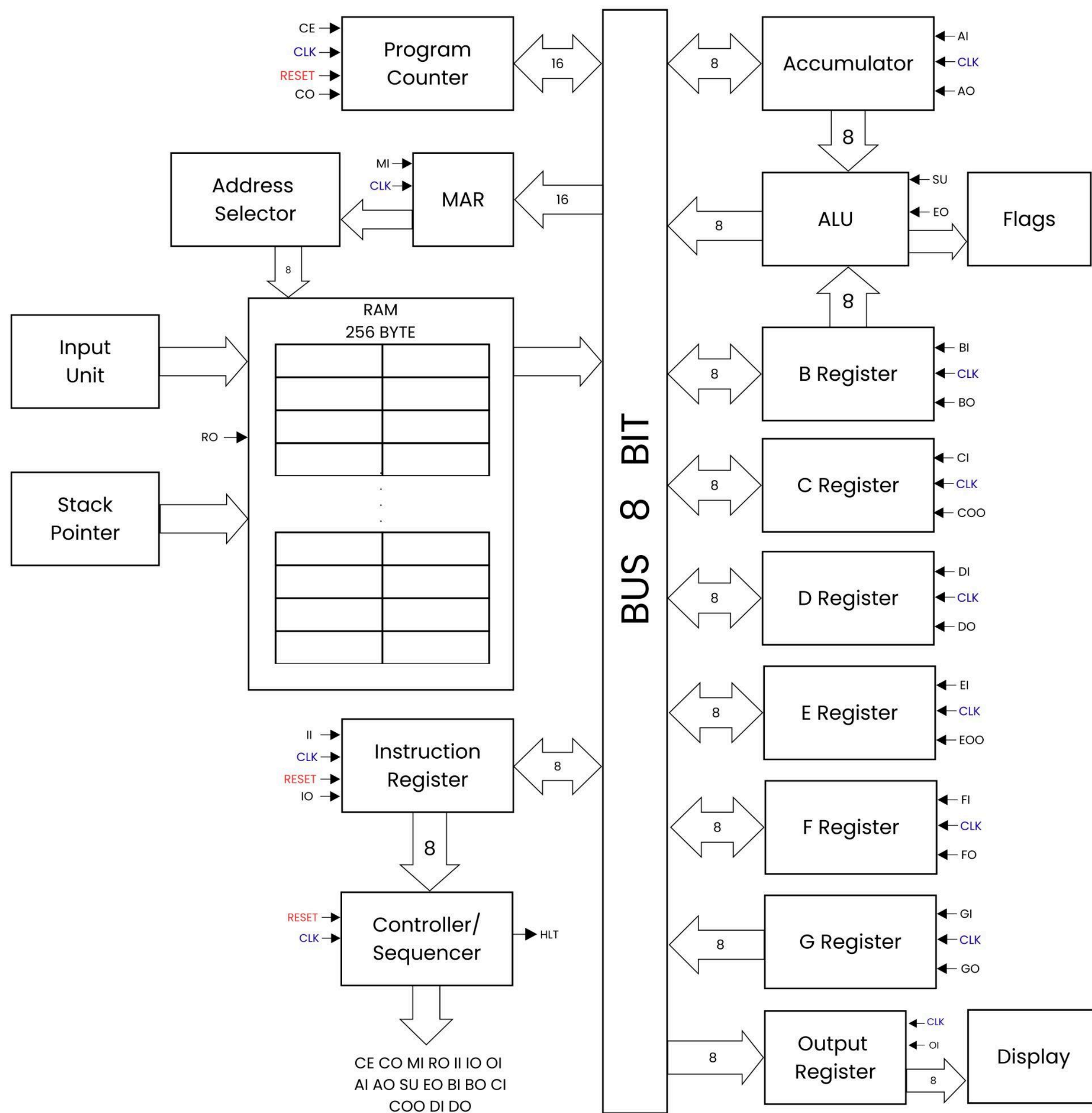


Figure 4.1: SAP-3 Block Diagram

4.2 SAP-3 Components

SAP-3 builds upon the foundation of SAP-1 and SAP-2 by introducing several new hardware blocks and extending existing ones. These additions significantly improve the processing capability, flexibility, and realism of the system. The following sections explain only the newly added or enhanced components that distinguish SAP-3 from earlier SAP architectures.

General-Purpose Registers:

One of the major improvements in SAP-3 compared to SAP-2 is the addition of several general-purpose registers. In SAP-2, the system mainly depends on the accumulator and a limited number of registers for data storage and operations. This restricts the way data can be handled during program execution.

SAP-3 overcomes this limitation by introducing multiple general-purpose registers such as C, D, E, F, and G along with the existing registers. Each of these registers is 8-bit wide and connected directly to the system bus. They can both receive data from the bus and place data onto the bus when required.

The presence of multiple registers allows intermediate results to be stored temporarily without immediately writing them back to memory. This reduces memory access, simplifies instruction execution, and improves overall efficiency. It also makes register-to-register data transfer possible, which was very limited in SAP-2. As a result, SAP-3 supports more flexible programming and better resembles real microprocessor register organization.

Stack Pointer:

Another important feature introduced in SAP-3 is the Stack Pointer, which is completely absent in SAP-2. The Stack Pointer is a special register that always holds the memory address of the top element of the stack.

The stack is a reserved area in memory used for temporary data storage, subroutine calls, and return addresses. During a PUSH operation, data is stored at the memory location pointed to by the Stack Pointer, and the pointer is updated accordingly. During a POP operation, data is retrieved from the stack, and the pointer moves in the opposite direction.

The inclusion of the Stack Pointer enables SAP-3 to support advanced instructions such as CALL and RET. These instructions allow programs to jump to subroutines and return back to the main program correctly. This feature makes it possible to write structured programs with reusable code blocks, which is not achievable in SAP-2.

Enhanced Controller / Sequencer

The Controller / Sequencer in SAP-3 is more advanced compared to the controller used in SAP-2. While the SAP-2 controller is capable of handling basic instruction fetch and execution, it is limited to a small and fixed set of operations. SAP-3 requires a stronger control unit due to the introduction of additional registers, stack operations, and conditional instructions.

In SAP-3, the controller is responsible for decoding a larger instruction set and generating a greater number of control signals. These signals manage register selection, stack pointer operations, flag evaluation, and register-to-register data transfers. The controller also coordinates the timing of each instruction by dividing execution into multiple clock cycles, ensuring that data moves correctly through the bus, registers, and ALU.

Another important improvement is the controller's ability to respond to status flags. After an ALU operation, flags such as Zero or Carry are updated, and the controller uses this information to decide whether conditional jump instructions should be executed or skipped. This allows SAP-3 programs to make decisions during execution, which was not possible in SAP-2.

4.3 SAP-3 Instruction Set

The instruction set of SAP-3 is more advanced than that of SAP-1 and SAP-2. It is designed to support register operations, stack handling, conditional branching, and input/output operations. Each instruction performs a specific task and is executed under the control of the controller/sequencer.

For clarity, the SAP-3 instruction set can be grouped into several categories based on functionality.

1. Data Transfer Instructions

These instructions are used to move data between memory, registers, and the accumulator.

- **LDA addr (Load Accumulator):** This instruction loads data from the specified memory address into the accumulator. It is mainly used to fetch data from memory for processing.
- **STA addr (Store Accumulator):** This instruction stores the contents of the accumulator into a specified memory location. It is used to save computation results.
- **MOV R1, R2 (Move Register Data):** This instruction copies data from one register to another. It allows register-to-register data transfer without involving memory, which improves efficiency.
- **LDI data (Load Immediate):** This instruction loads a constant value directly into a register or accumulator. It is useful for initializing registers with fixed values.

2. Arithmetic Instructions

Arithmetic instructions perform mathematical operations using the ALU.

- **ADD R (Add Register):**This instruction adds the contents of a selected register to the accumulator. The result is stored back in the accumulator.
- **SUB R (Subtract Register):** This instruction subtracts the contents of a register from the accumulator. It uses the subtraction control of the ALU.
- **INC R (Increment Register):** This instruction increases the value of a register by one. It is commonly used in loops and counters.

- **DEC R (Decrement Register):** This instruction decreases the value of a register by one. It is useful for countdown operations.

3. Logical Instructions

Logical instructions perform bit-wise operations.

- **AND R:** Performs a bitwise AND operation between the accumulator and a register.
- OR R: Performs a bitwise OR operation between the accumulator and a register.
- XOR R: Performs a bitwise exclusive OR operation between the accumulator and a register.

These instructions are mainly used for masking, comparison, and logic control operations.

4. Control Flow Instructions

Control flow instructions change the normal sequence of program execution.

- **JMP addr (Jump):** This instruction causes the program to jump to a specified memory address unconditionally.
- **JZ addr (Jump if Zero):** This instruction causes a jump only if the Zero flag is set. It is used for decision-making.
- **JC addr (Jump if Carry):** This instruction causes a jump if the Carry flag is set after an arithmetic operation.

These instructions allow SAP-3 programs to implement loops and conditional execution.

5. Stack Instructions

Stack instructions are supported due to the presence of the Stack Pointer in SAP-3.

- **PUSH R:** This instruction stores the contents of a register onto the stack and updates the stack pointer.
- **POP R:** This instruction retrieves data from the stack and loads it into a register.

These instructions are essential for temporary storage and subroutine handling.

6. Subroutine Instructions

SAP-3 supports structured program execution using subroutines.

- **CALL addr:** This instruction saves the return address onto the stack and jumps to the subroutine address.
- **RET (Return):** This instruction retrieves the return address from the stack and resumes execution from that point.

These instructions allow reusable code blocks and organized program structure.

7. Input and Output Instructions

These instructions handle communication with external devices.

- **IN:** Reads data from the input unit and places it onto the bus or into a register.
- **OUT:** Transfers data from the accumulator or bus to the output register for display.

8. Machine Control Instructions

These instructions control the overall operation of the processor.

- **HLT (Halt):** Stops program execution until the system is reset.
- **NOP (No Operation):** Does not perform any operation but consumes one clock cycle. It is useful for timing control.

Chapter 5: Verification & Design

SAP-1

```
==========================================
  SAP-1 Computer Simulation
==========================================

Program:
  0x0: LDA F  (Load 1)
  0x1: ADD E  (Add 3)
  0x2: SUB D  (Sub 5)
  0x3: OUT
  0x4: HLT
  Data: [D]=5, [E]=3, [F]=1

Expected: 1 + 3 - 5 = -1 (255)

Time | Cyc | PC |State| IR | ACC | B | ALU | OUT | INST
-----|-----|-------|-----|-------|-----|---|-----|-----|-----
  15 |   1 |   0 | T0  | 00 |   0 |   0 |   0 |   0 | inst=0
  25 |   2 |   0 | T1  | 00 |   0 |   0 |   0 |   0 | inst=0
  35 |   3 |   0 | T2  | 00 |   0 |   0 |   0 |   0 | inst=0
  45 |   4 |   1 | T6  | 1f |   0 |   0 |   0 |   0 | inst=0
  55 |   5 |   1 | T3  | 1f |   0 |   0 |   0 |   0 | inst=1
  65 |   6 |   1 | T4  | 1f |   0 |   0 |   0 |   0 | inst=1
  75 |   7 |   1 | T0  | 1f |   0 |   0 |   0 |   0 | inst=1
  85 |   8 |   1 | T1  | 1f |   1 |   0 |   1 |   0 | inst=1
  95 |   9 |   1 | T2  | 1f |   1 |   0 |   1 |   0 | inst=1
 105 |  10 |   2 | T6  | 2e |   1 |   0 |   1 |   0 | inst=1
 115 |  11 |   2 | T3  | 2e |   1 |   0 |   1 |   0 | inst=2
 125 |  12 |   2 | T4  | 2e |   1 |   0 |   1 |   0 | inst=2
 135 |  13 |   2 | T5  | 2e |   1 |   0 |   1 |   0 | inst=2
 145 |  14 |   2 | T0  | 2e |   1 |   3 |   4 |   0 | inst=2
 155 |  15 |   2 | T1  | 2e |   4 |   3 |   7 |   0 | inst=2
 165 |  16 |   2 | T2  | 2e |   4 |   3 |   7 |   0 | inst=2
 175 |  17 |   3 | T6  | 3d |   4 |   3 |   7 |   0 | inst=2
 185 |  18 |   3 | T3  | 3d |   4 |   3 |   7 |   0 | inst=3
 195 |  19 |   3 | T4  | 3d |   4 |   3 |   7 |   0 | inst=3
 205 |  20 |   3 | T5  | 3d |   4 |   3 |   7 |   0 | inst=3
 215 |  21 |   3 | T0  | 3d |   4 |   5 | 255 |   0 | inst=3
 225 |  22 |   3 | T1  | 3d | 255 |   5 |   4 |   0 | inst=3
 235 |  23 |   3 | T2  | 3d | 255 |   5 |   4 |   0 | inst=3
 245 |  24 |   4 | T6  | e0 | 255 |   5 |   4 |   0 | inst=3
 255 |  25 |   4 | T3  | e0 | 255 |   5 |   4 |   0 | inst=e
 265 |  26 |   4 | T0  | e0 | 255 |   5 |   4 |   0 | inst=e
 275 |  27 |   4 | T1  | e0 | 255 |   5 |   4 | 255 | inst=e
 285 |  28 |   4 | T2  | e0 | 255 |   5 |   4 | 255 | inst=e
 295 |  29 |   5 | T6  | f0 | 255 |   5 |   4 | 255 | inst=e
 305 |  30 |   5 | T3  | f0 | 255 |   5 |   4 | 255 | inst=f
 315 |  31 |   5 | T0  | f0 | 255 |   5 |   4 | 255 | inst=f

==========================================
HALTED - Final Output: 255 (0xff)
==========================================

*** TEST PASSED! ***

sap1.v:450: $finish called at 325 (1s)
 325 |  32 |   5 | T0  | f0 | 255 |   5 |   4 | 255 | inst=f
```

Figure 5.1: SAP-1 RTL Verification

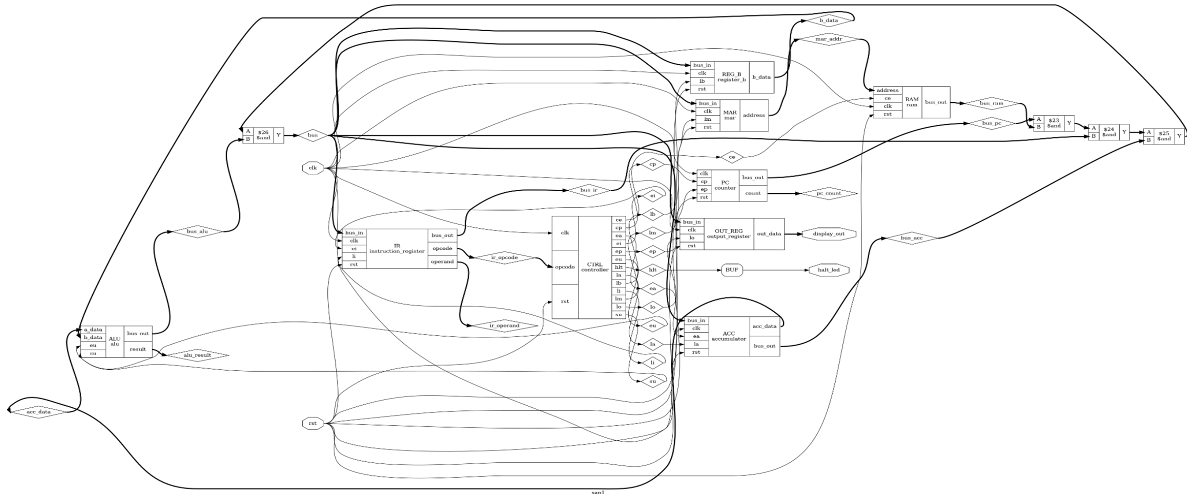


Figure 5.2: SAP-1 Netlist Simulation

Group	Internal Power	Switching Power	Leakage Power	Total Power (Watts)	
Sequential	1.27e-04	0.00e+00	1.16e-05	1.38e-04	86.2%
Combinational	5.30e-06	8.15e-07	1.60e-05	2.22e-05	13.8%
Clock	0.00e+00	0.00e+00	0.00e+00	0.00e+00	0.0%
Macro	0.00e+00	0.00e+00	0.00e+00	0.00e+00	0.0%
Pad	0.00e+00	0.00e+00	0.00e+00	0.00e+00	0.0%
Total	1.32e-04	8.15e-07	2.76e-05	1.60e-04	100.0%
	82.3%	0.5%	17.2%		

Figure 5.3: Simulation of Timing Power Check SAP-1

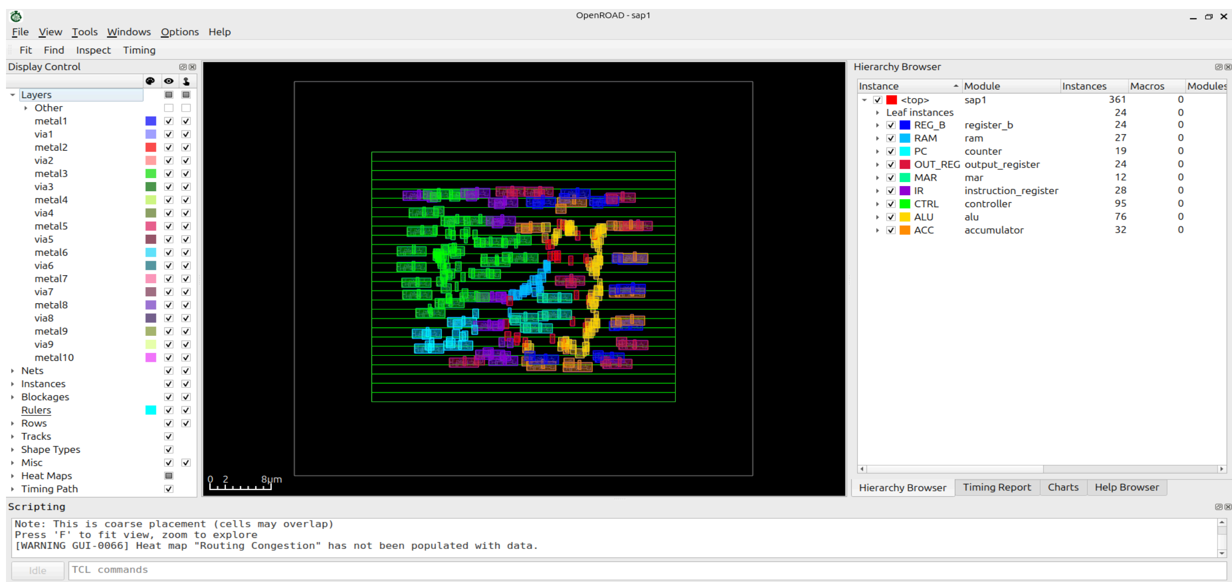


Figure 5.4: SAP-1 Physical Design

SAP-2

```
========================================
 SAP-2 COMPUTER TEST SUITE
========================================

>>> Test 1: LDA and OUT
Expected output: 42
Cyc | PC | T | IR | MAR | RegA | Bus | OUT
----|-------|---|-------|-----|------|-----|----
  0 |   0 | 1 | 00 |   0 |   0 |   0 |   0
  1 |   0 | 2 | 00 |   0 |   0 |  58 |   0
  2 |   1 | 3 | 3a |   0 |   0 |   z |   0
  3 |   1 | 4 | 3a |   0 |   0 |   z |   0
  4 |   1 | 5 | 3a |   0 |   0 |   1 |   0
  5 |   2 | 6 | 3a |   1 |   0 |  16 |   0
  6 |   2 | 0 | 3a |  16 |   0 |  42 |   0
  7 |   2 | 1 | 3a |  16 |  42 |   2 |   0
  8 |   2 | 2 | 3a |   2 |  42 | 211 |   0
  9 |   3 | 3 | d3 |   2 |  42 |   z |   0
 10 |   3 | 4 | d3 |   2 |  42 |   z |   0
 11 |   3 | 0 | d3 |   2 |  42 |  42 |   0
 12 |   3 | 1 | d3 |   2 |  42 |   3 |  42
 13 |   3 | 2 | d3 |   3 |  42 | 118 |  42
 14 |   4 | 3 | 76 |   3 |  42 |   z |  42
 15 |   4 | 4 | 76 |   3 |  42 |   z |  42
 16 |   4 | 0 | 76 |   3 |  42 |   z |  42
✓ PASS: Output = 42
```

```
>>> Test 2: LDA + ADD
Expected output: 40
Cyc | PC | T | IR | MAR | RegA | Bus | OUT
----|-------|---|-------|-----|------|-----|----
  0 |   0 | 1 | 00 |   0 |   0 |   0 |   0
  1 |   0 | 2 | 00 |   0 |   0 |  58 |   0
  2 |   1 | 3 | 3a |   0 |   0 |   z |   0
  3 |   1 | 4 | 3a |   0 |   0 |   z |   0
  4 |   1 | 5 | 3a |   0 |   0 |   1 |   0
  5 |   2 | 6 | 3a |   1 |   0 |  16 |   0
  6 |   2 | 0 | 3a |  16 |   0 |  15 |   0
  7 |   2 | 1 | 3a |  16 |  15 |   2 |   0
  8 |   2 | 2 | 3a |   2 |  15 | 134 |   0
  9 |   3 | 3 | 86 |   2 |  15 |   z |   0
 10 |   3 | 4 | 86 |   2 |  15 |   z |   0
 11 |   3 | 7 | 86 |   2 |  15 |   3 |   0
 12 |   4 | b | 86 |   3 |  15 |  17 |   0
 13 |   4 | c | 86 |  17 |  15 |  25 |   0
 14 |   4 | d | 86 |  17 |  15 |   z |   0
 15 |   4 | 0 | 86 |  17 |  15 |  40 |   0
 16 |   4 | 1 | 86 |  17 |  40 |   4 |   0
 17 |   4 | 2 | 86 |   4 |  40 | 211 |   0
 18 |   5 | 3 | d3 |   4 |  40 |   z |   0
 19 |   5 | 4 | d3 |   4 |  40 |   z |   0
 20 |   5 | 0 | d3 |   4 |  40 |  40 |   0
 21 |   5 | 1 | d3 |   4 |  40 |   5 |  40
 22 |   5 | 2 | d3 |   5 |  40 | 118 |  40
 23 |   6 | 3 | 76 |   5 |  40 |   z |  40
 24 |   6 | 4 | 76 |   5 |  40 |   z |  40
 25 |   6 | 0 | 76 |   5 |  40 |   z |  40
✓ PASS: Output = 40
```

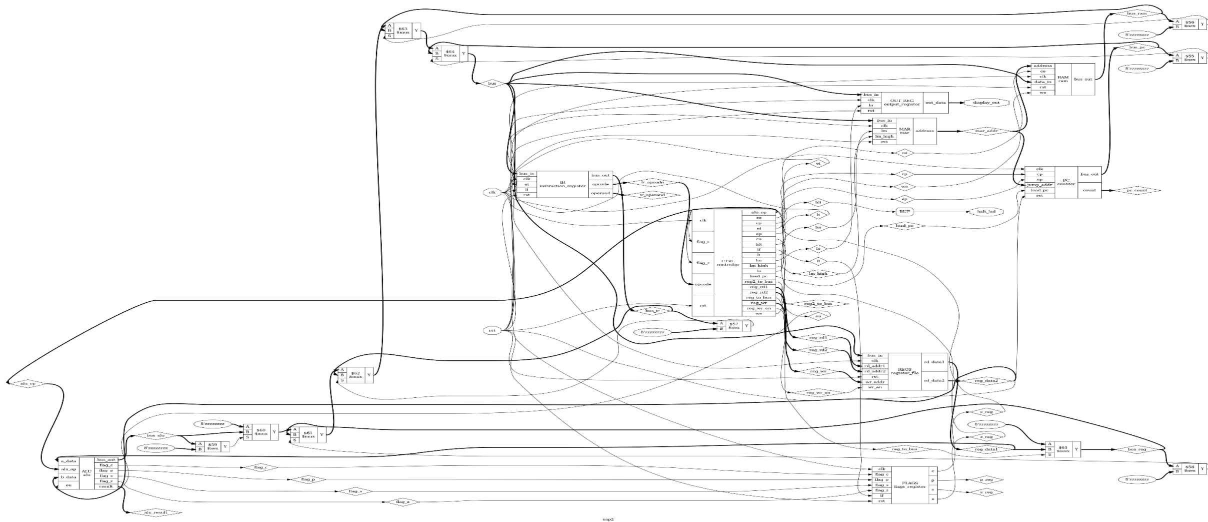
```
>>> Test 3: LDA + SUB
Expected output: 20
Cyc | PC | T | IR | MAR | RegA | Bus | OUT
----|-------|---|-------|-----|------|-----|----
  0 |   0 | 1 | 00 |   0 |   0 |   0 |   0
  1 |   0 | 2 | 00 |   0 |   0 |  58 |   0
  2 |   1 | 3 | 3a |   0 |   0 |   z |   0
  3 |   1 | 4 | 3a |   0 |   0 |   z |   0
  4 |   1 | 5 | 3a |   0 |   0 |   1 |   0
  5 |   2 | 6 | 3a |   1 |   0 |  16 |   0
  6 |   2 | 0 | 3a |  16 |   0 |  50 |   0
  7 |   2 | 1 | 3a |  16 |  50 |   2 |   0
  8 |   2 | 2 | 3a |   2 |  50 | 150 |   0
  9 |   3 | 3 | 96 |   2 |  50 |   z |   0
 10 |   3 | 4 | 96 |   2 |  50 |   z |   0
 11 |   3 | 7 | 96 |   2 |  50 |   3 |   0
 12 |   4 | b | 96 |   3 |  50 |  17 |   0
 13 |   4 | c | 96 |  17 |  50 |  30 |   0
 14 |   4 | d | 96 |  17 |  50 |   z |   0
 15 |   4 | 0 | 96 |  17 |  50 |  20 |   0
 16 |   4 | 1 | 96 |  17 |  20 |   4 |   0
 17 |   4 | 2 | 96 |   4 |  20 | 211 |   0
 18 |   5 | 3 | d3 |   4 |  20 |   z |   0
 19 |   5 | 4 | d3 |   4 |  20 |   z |   0
 20 |   5 | 0 | d3 |   4 |  20 |  20 |   0
 21 |   5 | 1 | d3 |   4 |  20 |   5 |  20
 22 |   5 | 2 | d3 |   5 |  20 | 118 |  20
 23 |   6 | 3 | 76 |   5 |  20 |   z |  20
 24 |   6 | 4 | 76 |   5 |  20 |   z |  20
 25 |   6 | 0 | 76 |   5 |  20 |   z |  20
✓ PASS: Output = 20
```

```
>>> Test 4: MVI (Move Immediate)
Expected output: 99
Cyc | PC | T | IR | MAR | RegA | Bus | OUT
----|-------|---|-------|-----|------|-----|----
  0 |   0 | 1 | 00 |   0 |   0 |   0 |   0
  1 |   0 | 2 | 00 |   0 |   0 |  62 |   0
  2 |   1 | 3 | 3e |   0 |   0 |   z |   0
  3 |   1 | 4 | 3e |   0 |   0 |   z |   0
  4 |   1 | f | 3e |   0 |   0 |   1 |   0
  5 |   2 | 0 | 3e |   1 |   0 |  99 |   0
  6 |   2 | 1 | 3e |   1 |  99 |   2 |   0
  7 |   2 | 2 | 3e |   2 |  99 | 211 |   0
  8 |   3 | 3 | d3 |   2 |  99 |   z |   0
  9 |   3 | 4 | d3 |   2 |  99 |   z |   0
 10 |   3 | 0 | d3 |   2 |  99 |  99 |   0
 11 |   3 | 1 | d3 |   2 |  99 |   3 |  99
 12 |   3 | 2 | d3 |   3 |  99 | 118 |  99
 13 |   4 | 3 | 76 |   3 |  99 |   z |  99
 14 |   4 | 4 | 76 |   3 |  99 |   z |  99
 15 |   4 | 0 | 76 |   3 |  99 |   z |  99
✓ PASS: Output = 99

========================================
 SAP-2 TEST SUITE COMPLETED
========================================

sap2.v:599: $finish called at 1266 (1s)
```

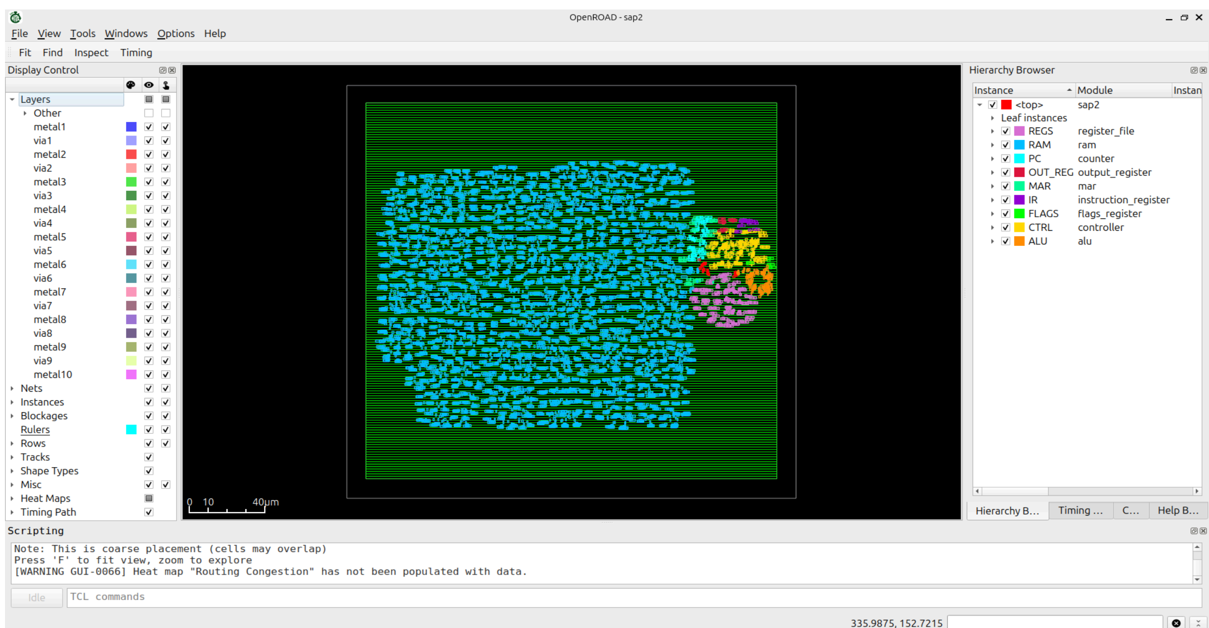
Figure 5.5: SAP-2 RTL Verification



5.6 *Figure: SAP-2 Netlist Simulation*

```
% report_power
Group                  Internal  Switching    Leakage      Total
                          Power      Power      Power      Power (Watts)
----------------------------------------------------------------
Sequential             3.63e-03   0.00e+00   4.14e-04   4.04e-03  90.4%
Combinational          3.20e-05   2.12e-07   3.97e-04   4.29e-04   9.6%
Clock                  0.00e+00   0.00e+00   0.00e+00   0.00e+00   0.0%
Macro                  0.00e+00   0.00e+00   0.00e+00   0.00e+00   0.0%
Pad                    0.00e+00   0.00e+00   0.00e+00   0.00e+00   0.0%
----------------------------------------------------------------
Total                  3.66e-03   2.12e-07   8.11e-04   4.47e-03 100.0%
                          81.9%       0.0%      18.1%
```

5.7 *Figure: Simulation of Timing Power Check SAP-2*



5.8 *Figure: SAP-2 Physical Design*

SAP-3

```
==========================================
 SAP-3 Computer Simulation (8080-like)
==========================================
Program:
 0x0: MVI A, 5
 0x2: INR A
 0x3: MVI B, 10
 0x5: ADD B
 0x6: OUT
 0x7: HLT

Expected: 5 + 1 + 10 = 16

Time | Cyc | PC |State| IR | ACC | B | C | OUT | Flags
-----|-----|-----|-----|----|----|----|----|-----|-------
  15 | 1 | 0000 | T0 | 00 | 0 | 0 | 0 | 0 | 1001
  25 | 2 | 0000 | T1 | 00 | 0 | 0 | 0 | 0 | 1001
  35 | 3 | 0000 | T2 | 00 | 0 | 0 | 0 | 0 | 1001
  45 | 4 | 0000 | T3 | 3e | 0 | 0 | 0 | 0 | 1001
  55 | 5 | 0001 | T7 | 3e | 0 | 0 | 0 | 0 | 1001
  65 | 6 | 0001 | T4 | 3e | 0 | 0 | 0 | 0 | 1001
  75 | 7 | 0001 | T0 | 3e | 0 | 0 | 0 | 0 | 1001
  85 | 8 | 0002 | T1 | 3e | 5 | 0 | 0 | 0 | 0001
  95 | 9 | 0002 | T2 | 3e | 5 | 0 | 0 | 0 | 0001
 105 | 10| 0002 | T3 | 3c | 5 | 0 | 0 | 0 | 0001
 115 | 11| 0003 | T7 | 3c | 5 | 0 | 0 | 0 | 0001
 125 | 12| 0003 | T0 | 3c | 5 | 0 | 0 | 0 | 0001
 135 | 13| 0003 | T1 | 3c | 6 | 0 | 0 | 0 | 0001
 145 | 14| 0003 | T2 | 3c | 6 | 0 | 0 | 0 | 0001
 155 | 15| 0003 | T3 | 06 | 6 | 0 | 0 | 0 | 0001
 165 | 16| 0004 | T7 | 06 | 6 | 0 | 0 | 0 | 0001
 175 | 17| 0004 | T4 | 06 | 6 | 0 | 0 | 0 | 0001
 185 | 18| 0004 | T0 | 06 | 6 | 0 | 0 | 0 | 0001
 195 | 19| 0005 | T1 | 06 | 6 | 10 | 0 | 0 | 0000
 205 | 20| 0005 | T2 | 06 | 6 | 10 | 0 | 0 | 0000
 215 | 21| 0005 | T3 | 80 | 6 | 10 | 0 | 0 | 0000
 225 | 22| 0006 | T7 | 80 | 6 | 10 | 0 | 0 | 0000
 235 | 23| 0006 | T0 | 80 | 6 | 10 | 0 | 0 | 0000
 245 | 24| 0006 | T1 | 80 | 16 | 10 | 0 | 0 | 0000
 255 | 25| 0006 | T2 | 80 | 16 | 10 | 0 | 0 | 0000
 265 | 26| 0006 | T3 | d3 | 16 | 10 | 0 | 0 | 0000
 275 | 27| 0007 | T7 | d3 | 16 | 10 | 0 | 0 | 0000
 285 | 28| 0007 | T0 | d3 | 16 | 10 | 0 | 0 | 0000
 295 | 29| 0007 | T1 | d3 | 16 | 10 | 0 | 16 | 0000
 305 | 30| 0007 | T2 | d3 | 16 | 10 | 0 | 16 | 0000
 315 | 31| 0007 | T3 | 76 | 16 | 10 | 0 | 16 | 0000
 325 | 32 | 0008 | T7 | 76 | 16 | 10 | 0 | 16 | 0000
 335 | 33| 0008 | T0 | 76 | 16 | 10 | 0 | 16 | 0000
==========================================
HALTED - Final Output: 16 (0x10)
Flags: Z=0 C=0 S=0 P=0
==========================================
*** TEST PASSED! ***
```

Figure 5.9: SAP-3 RTL Verification

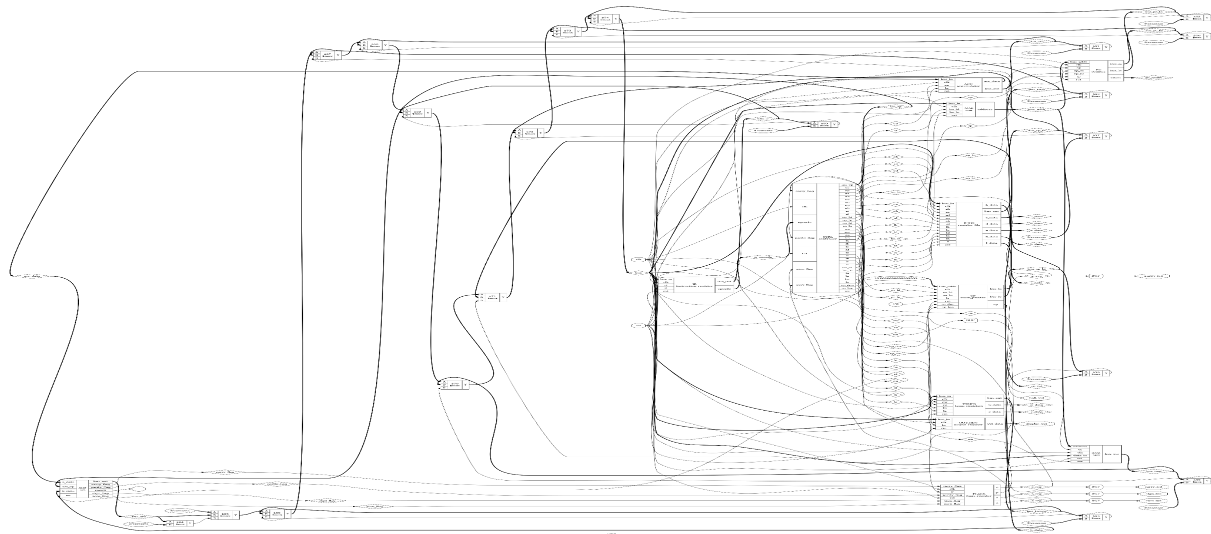


Figure 5.10: SAP-3 Netlist Simulation

Group	Internal Power	Switching Power	Leakage Power	Total Power (Watts)	
Sequential	2.57e-04	0.00e+00	4.33e-04	6.90e-04	61.7%
Combinational	3.56e-05	2.30e-07	3.93e-04	4.29e-04	38.3%
Clock	0.00e+00	0.00e+00	0.00e+00	0.00e+00	0.0%
Macro	0.00e+00	0.00e+00	0.00e+00	0.00e+00	0.0%
Pad	0.00e+00	0.00e+00	0.00e+00	0.00e+00	0.0%
Total	2.93e-04	2.30e-07	8.26e-04	1.12e-03	100.0%
	26.1%	0.0%	73.8%		

Figure 5.11: Simulation of Timing Power Check SAP-3

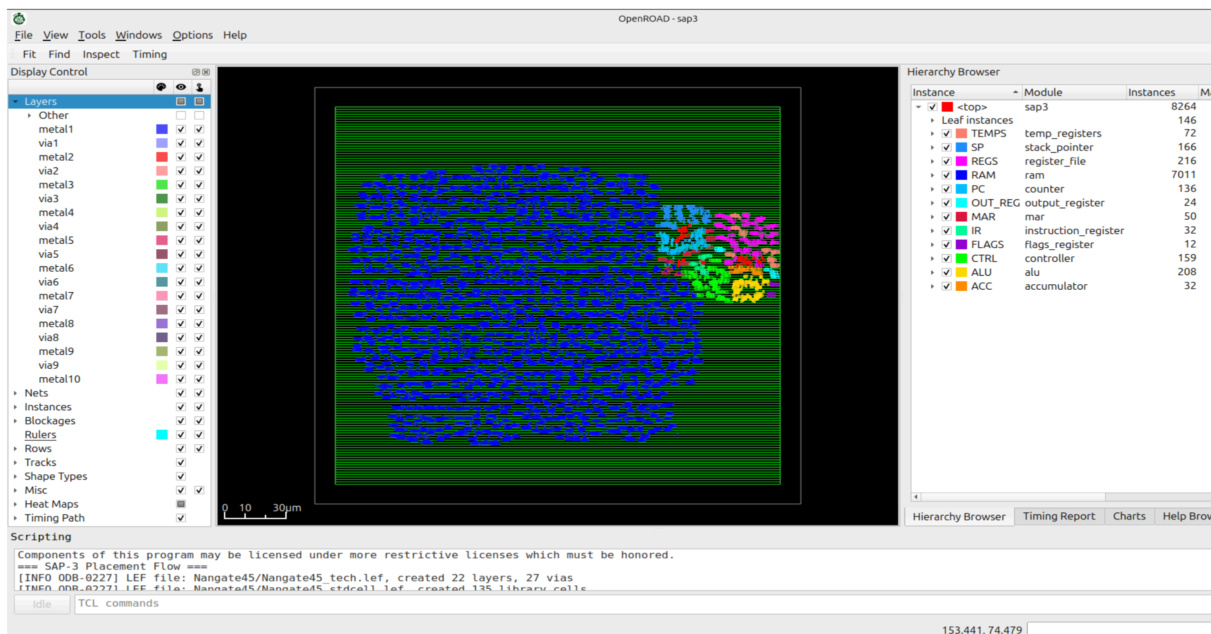


Figure 5.12: SAP-3 Physical Design

Chapter 6: Discussion & Conclusion

6.1 Conclusion

In this project, we successfully implemented and verified the design of SAP processor architectures, starting from RTL (Register Transfer Level) down to GDSII layout. The project demonstrates a complete hardware design flow, including design entry, simulation, synthesis, and physical layout generation.

Through this work, we were able to:

- Understand and implement the SAP processor architecture in a hardware description language.
- Verify functional correctness using simulation and testbench-based verification.
- Convert the RTL design into GDSII format, demonstrating the readiness of the design for fabrication.
- Compare the design steps with theoretical models to ensure accuracy and reliability.

Overall, this project enhanced our understanding of processor design, verification methodology, and VLSI design flow. It also provides a practical foundation for designing more complex processors in the future.

6.2 Limitations

Despite the success of the project, several limitations were identified:

1. **Limited Instruction Set:** The SAP processor architecture implemented supports a basic instruction set. More advanced instructions could not be implemented due to time and complexity constraints.
2. **Simulation Only:** Physical fabrication of the GDSII design was not performed, so real-world performance could not be validated.
3. **Resource Constraints:** Limited computational resources restricted the size and complexity of test cases for verification.
4. **Technology Node Limitation:** The GDSII layout was generated for an educational purpose, and advanced fabrication technologies (e.g., deep sub-micron) were not used.

6.3 Future Scopes

This project lays the foundation for several potential improvements and research directions:

1. **Enhanced Instruction Set:** Future work can implement a more complex instruction set and advanced features like interrupts, pipelining, or cache memory.
2. **FPGA or ASIC Implementation:** The design can be synthesized and tested on an FPGA or fabricated as an ASIC to validate real-world performance.
3. **Optimization:** Future designs can focus on area, speed, and power optimization, using advanced synthesis and layout tools.
4. **Support for Modern Architectures:** The methodology can be extended to RISC or more advanced processor architectures, building on the SAP model.
5. **Automated Verification:** Development of a more comprehensive automated verification environment can improve design reliability and reduce verification time.

References

1. A. P. Malvino and J. A. Brown, *Digital Computer Electronics*: An Introduction to Microcomputers, 3rd ed. New York, NY, USA: McGraw-Hill, 1993, pp. 140–212.
2. B. Eater, "*Build an 8-bit breadboard computer!*" YouTube playlist, 2016–2017. [Online]. Available: https://www.youtube.com/playlist?list=PLowKtXNTBypGqImE405J2565dvJajkVqP. (Accessed: Jan. 12, 2026).
3. A. Morlan, "*Building an FPGA Computer: SAP-1 → SAP-2 → SAP-3*," 2023. [Online]. Available: https://austinmorlan.com/posts/fpga_computer_sap1/. (Accessed: Jan. 12, 2026).
4. R. Electronics, "*The 8-bit SAP-3*," GitHub repository, 2023. [Online]. Available: https://github.com/rolf-electronics/The-8-bit-SAP-3. (Accessed: Jan. 12, 2026).
5. K. Ok, "*Designing and Implementing a SAP-1 Computer*," VHDL/FPGA project, 2023. [Online]. Available: https://karenok.github.io/SAP-1-Computer/. (Accessed: Jan. 12, 2026).
6. Wikipedia contributors, "*Simple-As-Possible computer*," Wikipedia, The Free Encyclopedia, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Simple-As-Possible_computer. (Accessed: Jan. 12, 2026).
7. D. Grieco, "*SAP-1 Processor Architecture*," GitHub documentation, 2022. [Online]. Available: https://dangrie158.github.io/SAP-1/. (Accessed: Jan. 12, 2026).
8. D. M. Harris and S. L. Harris, *Digital Design and Computer Architecture*, 2nd ed. Burlington, MA, USA: Morgan Kaufmann, 2013, pp. 1–120.
9. J. L. Hennessy and D. A. Patterson, *Computer Organization and Design: The Hardware/Software Interface*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann, 2014, pp. 45–180.

10. J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 6th ed. San Mateo, CA, USA: Morgan Kaufmann, 2019, pp. 1–150.
11. M. D. Ciletti, *Advanced Digital Design with the Verilog HDL*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall, 2011, pp. 65–210.
12. S. Palnitkar, *Verilog HDL: A Guide to Digital Design and Synthesis*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall, 2003, pp. 25–160.
13. V. Taraate, *ASIC Design and Synthesis: RTL Design Using Verilog*, Singapore: Springer, 2020, pp. 1–140.
14. S. Ramachandran, *Digital VLSI Systems Design*, New York, NY, USA: Springer, 2007, pp. 75–190.
15. N. H. E. Weste and D. Harris, *CMOS VLSI Design: A Systems Perspective*, 4th ed. Boston, MA, USA: Addison-Wesley, 2011, pp. 215–410.
16. K. J. Nowka and T. T. Arslan, *VLSI Circuit Design Methodologies*, New York, NY, USA: McGraw-Hill, 2009, pp. 90–240.
17. N. Sherwani, *Algorithms for VLSI Physical Design Automation*, 3rd ed. Boston, MA, USA: Kluwer Academic Publishers, 1999, pp. 120–310.
18. K. Golshan, *Physical Design Essentials: An ASIC Design Implementation Perspective*, New York, NY, USA: Springer, 2007, pp. 1–180.
19. D. Smith, *Static Timing Analysis for Nanometer Designs*, New York, NY, USA: Springer, 2010, pp. 35–200.
20. S. Sutherland, *RTL Modeling with SystemVerilog for Simulation and Synthesis*, Boston, MA, USA: Springer, 2013, pp. 60–190.
21. C. Spear and G. Tumbush, *SystemVerilog for Verification*, 3rd ed. New York, NY, USA: Springer, 2012, pp. 1–160.
22. B. Wile, J. Goss, and W. Roesner, *Comprehensive Functional Verification*, San Francisco, CA, USA: Morgan Kaufmann, 2005, pp. 95–260.
23. S. Furber, *ARM System-on-Chip Architecture*, 2nd ed. Boston, MA, USA: Addison-Wesley, 2000, pp. 30–140.
24. Y. Li, *Computer Principles and Design in Verilog HDL*, Beijing, China: Science Press, 2008, pp. 55–180.

Appendix

RTL Source Code (SAP-1)

```
// 1. PROGRAM COUNTER (4-bit)
module counter(
   input clk,
   input rst,
   input cp,  // count enable
   input ep,  // enable output to bus
   output reg [3:0] count,
   inout [7:0] bus
);
   always @(posedge clk or posedge rst) begin
      if (rst)
         count <= 4'b0000;
      else if (cp)
         count <= count + 1;
   end
   assign bus = ep ? {4'b0000, count} : 8'bzzzzzzzz;
endmodule
// 2. MEMORY ADDRESS REGISTER (4-bit)
module mar(
   input clk,
   input rst,
   input lm,  // load enable
    input [7:0] bus,
   output reg [3:0] address
);
   always @(posedge clk or posedge rst) begin
      if (rst)
```

```
            address <= 4'b0000;
        else if (lm)
            address <= bus[3:0];
    end
endmodule
// 3. RAM (16 bytes)
module ram(
    input clk,
    input [3:0] address,
    input ce,  // chip enable (output to bus)
    inout [7:0] bus
);
    reg [7:0] memory [0:15];
    // Initialize with empty program
    initial begin
        memory[0]  = 8'h00;
        memory[1]  = 8'h00;
        memory[2]  = 8'h00;
        memory[3]  = 8'h00;
        memory[4]  = 8'h00;
        memory[5]  = 8'h00;
        memory[6]  = 8'h00;
        memory[7]  = 8'h00;
        memory[8]  = 8'h00;
        memory[9]  = 8'h00;
        memory[10] = 8'h00;
        memory[11] = 8'h00;
        memory[12] = 8'h00;
        memory[13] = 8'h00;
        memory[14] = 8'h00;
```

```
        memory[15] = 8'h00;
    end
    assign bus = ce ? memory[address] : 8'bzzzzzzzz;
endmodule
// 4. INSTRUCTION REGISTER (8-bit)
module instruction_register(
    input clk,
    input rst,
    input li,  // load instruction
    input ei,  // enable instruction (lower 4 bits to bus)
    input [7:0] bus,
    output reg [3:0] opcode,
    output reg [3:0] operand,
    inout [7:0] bus_out
);
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            opcode <= 4'b0000;
            operand <= 4'b0000;
        end
        else if (li) begin
            opcode <= bus[7:4];
            operand <= bus[3:0];
        end
    end
    assign bus_out = ei ? {4'b0000, operand} : 8'bzzzzzzzz;
endmodule
// 5. CONTROLLER/SEQUENCER
module controller(
    input clk,
```

```
    input rst,
    input [3:0] opcode,
    output reg cp,  // counter enable
    output reg ep,  // counter output enable
    output reg lm,  // MAR load
    output reg ce,  // RAM chip enable
    output reg li,  // instruction register load
    output reg ei,  // instruction register output enable
    output reg la,  // accumulator load
    output reg ea,  // accumulator output enable
    output reg su,  // ALU subtract
    output reg eu,  // ALU output enable
    output reg lb,  // B register load
    output reg lo,  // output register load
    output reg hlt  // halt
);
    reg [2:0] t_state;  // Timing state
    reg [3:0] instruction;  // Latched instruction
    // Instruction opcodes
    parameter LDA = 4'h1;
    parameter ADD = 4'h2;
    parameter SUB = 4'h3;
    parameter OUT = 4'hE;
    parameter HLT = 4'hF;
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            t_state <= 3'b000;
            instruction <= 4'h0;
            cp <= 0; ep <= 0; lm <= 0; ce <= 0; li <= 0;
            ei <= 0; la <= 0; ea <= 0; su <= 0; eu <= 0;
```

```
        lb <= 0; lo <= 0; hlt <= 0;
    end
    else if (!hlt) begin
        // Default: clear all control signals
        cp <= 0; ep <= 0; lm <= 0; ce <= 0; li <= 0;
        ei <= 0; la <= 0; ea <= 0; su <= 0; eu <= 0;
        lb <= 0; lo <= 0
        case (t_state)
            // T0: Address state - PC to MAR
            3'b000: begin
                ep <= 1;  // PC to bus
                lm <= 1;  // Load MAR
                t_state <= 3'b001;
            end
            // T1: Increment state - Increment PC, RAM to IR
            3'b001: begin
                cp <= 1;  // Increment PC
                ce <= 1;  // RAM to bus
                li <= 1;  // Load IR
                t_state <= 3'b010;
            end
            // T2: Wait and latch instruction
            3'b010: begin
                // Do nothing, just wait for IR to update
                t_state <= 3'b110;  // Go to T6 (new latch state)
            end
            // T6: Latch state - now opcode is valid
            3'b110: begin
                instruction <= opcode;  // NOW opcode is stable
                t_state <= 3'b011;
```

```
end
// T3: Execute
3'b011: begin
    case (instruction)
        LDA: begin
            ei <= 1;  // IR operand to bus
            lm <= 1;  // Load MAR with address
            t_state <= 3'b100;
        end
        ADD: begin
            ei <= 1;
            lm <= 1;
            t_state <= 3'b100;
        end
        SUB: begin
            ei <= 1;
            lm <= 1;
            t_state <= 3'b100;
        end
        OUT: begin
            ea <= 1;  // ACC to bus
            lo <= 1;  // Load output register
            t_state <= 3'b000;  // Back to fetch
        end
        HLT: begin
            hlt <= 1;
            t_state <= 3'b000;
        end
        default: t_state <= 3'b000;
    endcase
```

```
end
// T4: Memory read/write
3'b100: begin
    case (instruction)
        LDA: begin
            ce <= 1;  // RAM to bus
            la <= 1;  // Load ACC
            t_state <= 3'b000;
        end
        ADD: begin
            ce <= 1;  // RAM to bus
            lb <= 1;  // Load B register
            t_state <= 3'b101;
        end
        SUB: begin
            ce <= 1;
            lb <= 1;
            t_state <= 3'b101;
        end
        default: t_state <= 3'b000;
    endcase
end
// T5: ALU operation
3'b101: begin
    case (instruction)
        ADD: begin
            eu <= 1;  // ALU to bus
            la <= 1;  // Load ACC
            su <= 0;  // Add mode
            t_state <= 3'b000;
```

```
                end
                SUB: begin
                    eu <= 1;
                    la <= 1;
                    su <= 1;  // Subtract mode
                    t_state <= 3'b000;
                end
                default: t_state <= 3'b000;
            endcase
        end
        default: t_state <= 3'b000;
    endcase
    end
  end
endmodule
// 6. ACCUMULATOR (8-bit)
module accumulator(
  input clk,
  input rst,
  input la,  // load enable
  input ea,  // enable output to bus
  input [7:0] bus,
  output reg [7:0] acc_data,
  inout [7:0] bus_out
);
  always @(posedge clk or posedge rst) begin
    if (rst)
      acc_data <= 8'b00000000;
    else if (la)
      acc_data <= bus;
```

```
    end
    assign bus_out = ea ? acc_data : 8'bzzzzzzzz;
endmodule
// 7. REGISTER B (8-bit)
module register_b(
    input clk,
    input rst,
    input lb,  // load enable
    input [7:0] bus,
    output reg [7:0] b_data
);
    always @(posedge clk or posedge rst) begin
        if (rst)
            b_data <= 8'b00000000;
        else if (lb)
            b_data <= bus;
    end
endmodule
// 8. ALU (8-bit)
module alu(
    input [7:0] a_data,
    input [7:0] b_data,
    input su,  // subtract control
    input eu,  // enable output to bus
    output [7:0] result,
    inout [7:0] bus
);
    assign result = su ? (a_data - b_data) : (a_data + b_data);
    assign bus = eu ? result : 8'bzzzzzzzz;
endmodule
```

```
// 9. OUTPUT REGISTER (8-bit)
module output_register(
   input clk,
   input rst,
   input lo,  // load enable
   input [7:0] bus,
   output reg [7:0] out_data
);
   always @(posedge clk or posedge rst) begin
      if (rst)
         out_data <= 8'b00000000;
      else if (lo)
         out_data <= bus;
   end
endmodule
// 10. TOP MODULE - SAP-1 COMPUTER
module sap1(
   input clk,
   input rst,
   output [7:0] display_out,
   output halt_led
);
   // 8-bit bus
   wire [7:0] bus;
   // Control signals
   wire cp, ep, lm, ce, li, ei, la, ea, su, eu, lb, lo, hlt;

   // Internal wires
   wire [3:0] pc_count;
   wire [3:0] mar_addr;
```

```
wire [3:0] ir_opcode;
wire [3:0] ir_operand;
wire [7:0] acc_data;
wire [7:0] b_data;
wire [7:0] alu_result;
// Instantiate modules
counter PC(
    .clk(clk), .rst(rst), .cp(cp), .ep(ep),
    .count(pc_count), .bus(bus)
);
mar MAR(
    .clk(clk), .rst(rst), .lm(lm),
    .bus(bus), .address(mar_addr)
);
ram RAM(
    .clk(clk), .address(mar_addr), .ce(ce),
    .bus(bus)
);
instruction_register IR(
    .clk(clk), .rst(rst), .li(li), .ei(ei),
    .bus(bus), .opcode(ir_opcode), .operand(ir_operand),
    .bus_out(bus)
);
controller CTRL(
    .clk(clk), .rst(rst), .opcode(ir_opcode),
    .cp(cp), .ep(ep), .lm(lm), .ce(ce), .li(li), .ei(ei),
    .la(la), .ea(ea), .su(su), .eu(eu), .lb(lb), .lo(lo), .hlt(hlt)
);
accumulator ACC(
    .clk(clk), .rst(rst), .la(la), .ea(ea),
```

```
        .bus(bus), .acc_data(acc_data), .bus_out(bus)
    );
    register_b REG_B(
        .clk(clk), .rst(rst), .lb(lb),
        .bus(bus), .b_data(b_data)
    );
    alu ALU(
        .a_data(acc_data), .b_data(b_data), .su(su), .eu(eu),
        .result(alu_result), .bus(bus)
    );
    output_register OUT_REG(
        .clk(clk), .rst(rst), .lo(lo),
        .bus(bus), .out_data(display_out)
    );
    assign halt_led = hlt;
endmodule
```

// TESTBENCH CODE

```
module sap1_tb;
    reg clk, rst;
    wire [7:0] display_out;
    wire halt_led;
    integer cycle_count;
    sap1 UUT(
        .clk(clk),
        .rst(rst),
        .display_out(display_out),
        .halt_led(halt_led)
    );
    initial begin
        clk = 0;
```

```
        forever #5 clk = ~clk;
    end
    initial begin
        $display("\n========================================");
        $display("  SAP-1 Computer Simulation");
        $display("========================================\n");
rst = 1;
        cycle_count = 0;
        // Load program after a delay
        #10;
        // Program: 1 + 3 - 5 = -1 (255)
        UUT.RAM.memory[0]  = 8'h1F;  // LDA F
        UUT.RAM.memory[1]  = 8'h2E;  // ADD E
        UUT.RAM.memory[2]  = 8'h3D;  // SUB D
        UUT.RAM.memory[3]  = 8'hE0;  // OUT
        UUT.RAM.memory[4]  = 8'hF0;  // HLT
        UUT.RAM.memory[13] = 8'd5;
        UUT.RAM.memory[14] = 8'd3;
        UUT.RAM.memory[15] = 8'd1;

        $display("Program:");
        $display("  0x0: LDA F  (Load 1)");
        $display("  0x1: ADD E  (Add 3)");
        $display("  0x2: SUB D  (Sub 5)");
        $display("  0x3: OUT");
        $display("  0x4: HLT");
        $display("  Data: [D]=5, [E]=3, [F]=1");
        $display("\nExpected: 1 + 3 - 5 = -1 (255)\n");
#5 rst = 0;
        $display("Time | Cyc | PC |State| IR | ACC | B | ALU | OUT | INST");
```

```
        $display("-----|-----|-------|-----|-------|-----|---|-----|-----|-----");
        wait(halt_led == 1);
        #20;
        $display("\n========================================");
        $display("HALTED - Final Output: %d (0x%h)", display_out, display_out);
        $display("========================================");
        if (display_out == 8'd255)
            $display("\n*** TEST PASSED! ***\n");
        else
            $display("\n*** FAILED! Expected 255, got %d ***\n", display_out);
        $finish;
    end
    always @(posedge clk) begin
        if (!rst) begin
            cycle_count = cycle_count + 1;
            $display("%4t | %3d | %3d | T%d  | %h | %3d | %d | %3d | %3d | inst=%h",
                     $time, cycle_count, UUT.pc_count, UUT.CTRL.t_state,
                     {UUT.ir_opcode, UUT.ir_operand},
                     UUT.acc_data, UUT.b_data, UUT.alu_result, display_out,
                     UUT.CTRL.instruction);
        end
    end
endmodule
```

Command for RTL Source Code

```
iverilog -o sap1 sap1.v
./sap1
```

Command for Netlist Simulation

```
# read modules from Verilog file
read_verilog sap1.v
#hierarchy top
```

```
hierarchy -top sap1

# translate processes to netlists

proc

# remove unused cells and wires

clean

# show the generic netlist

show sap1

#perform optimization

opt

#resource sharing optimization

share -aggressive

# show the generic netlist

show sap1

# mapping to internal cell library

techmap

# mapping flip-flops to toy.lib

dfflibmap -liberty NangateOCL.lib

# mapping logic to toy.lib

abc -liberty NangateOCL.lib

# remove unused cells and wires

clean

#report design statistics

stat -liberty NangateOCL.lib

# Write the current design to a Verilog file

write_verilog -noattr -noexpr -nohex -nodec netlist_sap1.v
```

Command for Timing Power Check

```
sta

read_liberty NangateOCL.lib

read_verilog netlist_sap1.v

link_design sap1
```

```
read_sdc top.sdc

report_checks -path_delay max

report_checks -path_delay min

report_power
```

Command for physical Design

```
openroad -gui place_and_view_sap3_simple.tcl
```

RTL Source Code (SAP-2)

```
//============================================================================
// SAP-2 (Simple As Possible - 2) Computer Implementation in Verilog
// FIXED VERSION - Controller state machine corrected
//============================================================================
// 1. PROGRAM COUNTER (16-bit for 64K addressing)
module counter(
  input clk,
  input rst,
  input cp,
  input ep,
  input [15:0] jump_addr,
  input load_pc,
  output reg [15:0] count,
  inout [7:0] bus
);
  always @(posedge clk or posedge rst) begin
    if (rst)
      count <= 16'h0000;
    else if (load_pc)
      count <= jump_addr;
    else if (cp)
      count <= count + 1;
  end

  assign bus = ep ? count[7:0] : 8'bzzzzzzzz;
endmodule

// 2. MEMORY ADDRESS REGISTER (16-bit)
module mar(
  input clk,
```

```
    input rst,
    input lm,
    input lm_high,
    input [7:0] bus,
    output reg [15:0] address
);
    always @(posedge clk or posedge rst) begin
        if (rst)
            address <= 16'h0000;
        else if (lm)
            address[7:0] <= bus;
        else if (lm_high)
            address[15:8] <= bus;
    end
endmodule

// 3. RAM (256 bytes)
module ram(
    input clk,
    input [15:0] address,
    input ce,
    input we,
    input [7:0] data_in,
    inout [7:0] bus
);
    reg [7:0] memory [0:255];

    integer i;
    initial begin
        for (i = 0; i < 256; i = i + 1)
            memory[i] = 8'h00;
    end

    always @(posedge clk) begin
        if (we)
            memory[address[7:0]] <= data_in;
    end

    assign bus = ce ? memory[address[7:0]] : 8'bzzzzzzzz;
endmodule
```

```
// 4. INSTRUCTION REGISTER
module instruction_register(
    input clk,
    input rst,
    input li,
    input ei,
    input [7:0] bus,
    output reg [7:0] opcode,
    output reg [7:0] operand,
    inout [7:0] bus_out
);
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            opcode <= 8'h00;
            operand <= 8'h00;
        end
        else if (li) begin
            opcode <= bus;
        end
    end

    assign bus_out = ei ? operand : 8'bzzzzzzzz;
endmodule

// 5. REGISTER FILE
module register_file(
    input clk,
    input rst,
    input [2:0] rd_addr1,
    input [2:0] rd_addr2,
    input [2:0] wr_addr,
    input wr_en,
    input [7:0] bus,
    output [7:0] rd_data1,
    output [7:0] rd_data2
);
    reg [7:0] regs [0:7];

    integer i;
```

```
    initial begin
        for (i = 0; i < 8; i = i + 1)
            regs[i] = 8'h00;
    end

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            for (i = 0; i < 8; i = i + 1)
                regs[i] <= 8'h00;
        end
        else if (wr_en)
            regs[wr_addr] <= bus;
    end

    assign rd_data1 = regs[rd_addr1];
    assign rd_data2 = regs[rd_addr2];
endmodule

// 6. ALU with Flags
module alu(
    input [7:0] a_data,
    input [7:0] b_data,
    input [3:0] alu_op,
    input eu,
    output reg [7:0] result,
    output reg flag_z,
    output reg flag_c,
    output reg flag_s,
    output reg flag_p,
    inout [7:0] bus
);
    parameter ADD = 4'h0, SUB = 4'h1, AND = 4'h2, OR = 4'h3;
    parameter XOR = 4'h4, INC = 4'h5, DEC = 4'h6, CMP = 4'h7;

    reg [8:0] temp_result;

    always @(*) begin
        flag_c = 0;
        case (alu_op)
            ADD: begin
```

```
            temp_result = a_data + b_data;
            result = temp_result[7:0];
            flag_c = temp_result[8];
        end
        SUB, CMP: begin
            temp_result = a_data - b_data;
            result = temp_result[7:0];
            flag_c = temp_result[8];
        end
        AND: result = a_data & b_data;
        OR:  result = a_data | b_data;
        XOR: result = a_data ^ b_data;
        INC: result = a_data + 1;
        DEC: result = a_data - 1;
        default: result = a_data;
    endcase

    flag_z = (result == 8'h00);
    flag_s = result[7];
    flag_p = ~^result;
  end

  assign bus = eu ? result : 8'bzzzzzzzz;
endmodule

// 7. FLAGS REGISTER
module flags_register(
  input clk,
  input rst,
  input lf,
  input flag_z,
  input flag_c,
  input flag_s,
  input flag_p,
  output reg z,
  output reg c,
  output reg s,
  output reg p
);
  always @(posedge clk or posedge rst) begin
```

```
        if (rst) begin
            z <= 0; c <= 0; s <= 0; p <= 0;
        end
        else if (lf) begin
            z <= flag_z;
            c <= flag_c;
            s <= flag_s;
            p <= flag_p;
        end
    end
endmodule

// 8. CONTROLLER - FIXED STATE MACHINE
module controller(
    input clk,
    input rst,
    input [7:0] opcode,
    input flag_z,
    input flag_c,
    output reg cp, ep, lm, lm_high, ce, we, li, ei,
    output reg [3:0] alu_op,
    output reg eu, lf,
    output reg [2:0] reg_rd1, reg_rd2, reg_wr,
    output reg reg_wr_en, reg_to_bus, reg2_to_bus, load_pc, lo, hlt
);
    reg [3:0] t_state;
    reg [7:0] instruction;

    parameter NOP=8'h00, LDA=8'h3A, STA=8'h32;
    parameter MVI_A=8'h3E, MVI_B=8'h06, MVI_C=8'h0E, MVI_D=8'h16, MVI_E=8'h1E;
    parameter ADD_M=8'h86, SUB_M=8'h96, ANA_M=8'hA6, ORA_M=8'hB6, XRA_M=8'hAE;
    parameter INR_A=8'h3C, INR_B=8'h04, INR_C=8'h0C, INR_D=8'h14, INR_E=8'h1C;
    parameter DCR_A=8'h3D, DCR_B=8'h05, DCR_C=8'h0D, DCR_D=8'h15, DCR_E=8'h1D;
    parameter JMP=8'hC3, JZ=8'hCA, JNZ=8'hC2, JC=8'hDA, JNC=8'hD2;
    parameter OUT_OP=8'hD3, HLT=8'h76;
    parameter REG_A=3'h0, REG_B=3'h1, REG_C=3'h2, REG_D=3'h3;
    parameter REG_E=3'h4, REG_H=3'h5, REG_L=3'h6;

    always @(posedge clk or posedge rst) begin
        if (rst) begin
```

```
    t_state <= 4'h0; instruction <= 8'h00;
    cp <= 0; ep <= 0; lm <= 0; lm_high <= 0; ce <= 0; we <= 0;
    li <= 0; ei <= 0; eu <= 0; lf <= 0; lo <= 0; hlt <= 0;
    reg_wr_en <= 0; reg_to_bus <= 0; reg2_to_bus <= 0; load_pc <= 0; alu_op <= 4'h0;
    reg_rd1 <= 0; reg_rd2 <= 0; reg_wr <= 0;
end
else if (!hlt) begin
    // Default: clear all control signals
    cp <= 0; ep <= 0; lm <= 0; lm_high <= 0; ce <= 0; we <= 0;
    li <= 0; ei <= 0; eu <= 0; lf <= 0; lo <= 0;
    reg_wr_en <= 0; reg_to_bus <= 0; reg2_to_bus <= 0; load_pc <= 0;

    case (t_state)
        // FETCH CYCLE
        4'h0: begin  // T0: PC → MAR
            ep <= 1; lm <= 1;
            t_state <= 4'h1;
        end

        4'h1: begin  // T1: RAM → IR, PC++
            cp <= 1; ce <= 1; li <= 1;
            t_state <= 4'h2;
        end

        4'h2: begin  // T2: Wait state
            t_state <= 4'h3;
        end

        4'h3: begin  // T3: Latch instruction
            instruction <= opcode;
            t_state <= 4'h4;
        end

        // EXECUTE CYCLE
        4'h4: begin
            case (instruction)
                NOP: begin
                    t_state <= 4'h0;
                end
```

```
LDA: begin  // T4: PC -> MAR, PC++
    ep <= 1; lm <= 1; cp <= 1;
    reg_wr <= REG_A;
    t_state <= 4'h5;
end

STA: begin  // Store Accumulator Direct
    ep <= 1; lm <= 1; cp <= 1;
    t_state <= 4'h8;
end

MVI_A, MVI_B, MVI_C, MVI_D, MVI_E: begin  // T4: PC -> MAR, PC++
    ep <= 1; lm <= 1; cp <= 1;
    // Decode which register from opcode
    case (instruction)
        MVI_A: reg_wr <= REG_A;
        MVI_B: reg_wr <= REG_B;
        MVI_C: reg_wr <= REG_C;
        MVI_D: reg_wr <= REG_D;
        MVI_E: reg_wr <= REG_E;
    endcase
    t_state <= 4'hF;  // Use TF for MVI read
end

ADD_M, SUB_M, ANA_M, ORA_M, XRA_M: begin
    ep <= 1; lm <= 1; cp <= 1;
    t_state <= 4'h7;
end

INR_A, INR_B, INR_C, INR_D, INR_E,
DCR_A, DCR_B, DCR_C, DCR_D, DCR_E: begin
    // Decode which register
    case (instruction)
        INR_A, DCR_A: begin reg_rd1 <= REG_A; reg_wr <= REG_A; end
        INR_B, DCR_B: begin reg_rd1 <= REG_B; reg_wr <= REG_B; end
        INR_C, DCR_C: begin reg_rd1 <= REG_C; reg_wr <= REG_C; end
        INR_D, DCR_D: begin reg_rd1 <= REG_D; reg_wr <= REG_D; end
        INR_E, DCR_E: begin reg_rd1 <= REG_E; reg_wr <= REG_E; end
    endcase
    // Decode operation
```

```
            case (instruction)
                INR_A, INR_B, INR_C, INR_D, INR_E: alu_op <= 4'h5;
                DCR_A, DCR_B, DCR_C, DCR_D, DCR_E: alu_op <= 4'h6;
            endcase
            t_state <= 4'h9;
        end

        JMP, JZ, JNZ, JC, JNC: begin  // Jump instructions
            ep <= 1; lm <= 1; cp <= 1;
            t_state <= 4'hA;
        end

        OUT_OP: begin  // Output
            reg_rd1 <= REG_A;
            reg_to_bus <= 1;
            lo <= 1;
            t_state <= 4'h0;
        end

        HLT: begin  // Halt
            hlt <= 1;
            t_state <= 4'h0;
        end

        default: t_state <= 4'h0;
    endcase
end

// EXECUTION STATES
4'h5: begin  // T5: LDA - RAM[MAR] -> MAR (get address)
    ce <= 1; lm <= 1;
    t_state <= 4'h6;
end

4'h6: begin  // T6: LDA - RAM[MAR] -> Register
    ce <= 1;
    reg_wr_en <= 1;
    t_state <= 4'h0;
end
```

```
4'h7: begin  // T7: ALU ops - RAM[MAR] -> MAR (get data address)
    ce <= 1; lm <= 1;
    t_state <= 4'hB;
end

4'h8: begin  // T8: STA - wait for address
    t_state <= 4'hE;
end

4'h9: begin  // T9: INR/DCR - execute
    eu <= 1;
    reg_wr_en <= 1;
    lf <= 1;
    t_state <= 4'h0;
end

4'hA: begin  // TA: Jump - get high byte (if needed)
    // For SAP-2 simplified jumps, just use low byte
    case (instruction)
        JMP:  load_pc <= 1;
        JZ:   load_pc <= flag_z;
        JNZ:  load_pc <= ~flag_z;
        JC:   load_pc <= flag_c;
        JNC:  load_pc <= ~flag_c;
    endcase
    t_state <= 4'h0;
end

4'hB: begin  // TB: ALU ops - read memory into B
    ce <= 1;
    reg_wr <= REG_B;
    reg_wr_en <= 1;
    t_state <= 4'hC;
end

4'hC: begin  // TC: ALU ops - setup ALU operation
    reg_rd1 <= REG_A;
    reg_rd2 <= REG_B;
    case (instruction)
        ADD_M: alu_op <= 4'h0;
```

```
            SUB_M: alu_op <= 4'h1;
            ANA_M: alu_op <= 4'h2;
            ORA_M: alu_op <= 4'h3;
            XRA_M: alu_op <= 4'h4;
          endcase
          t_state <= 4'hD;
        end

        4'hD: begin  // TD: ALU ops - execute and store result
          eu <= 1;
          reg_wr <= REG_A;
          reg_wr_en <= 1;
          lf <= 1;
          t_state <= 4'h0;
        end

        4'hE: begin  // TE: STA - write A to memory
          reg_rd1 <= REG_A;
          reg_to_bus <= 1;
          we <= 1;
          t_state <= 4'h0;
        end

        4'hF: begin  // TF: MVI - RAM[MAR] -> Register (immediate value)
          ce <= 1;
          reg_wr_en <= 1;
          t_state <= 4'h0;
        end

        default: t_state <= 4'h0;
      endcase
    end
  end
endmodule

// 9. OUTPUT REGISTER
module output_register(
  input clk,
  input rst,
  input lo,
```

```
    input [7:0] bus,
    output reg [7:0] out_data
);
    always @(posedge clk or posedge rst) begin
        if (rst)
            out_data <= 8'h00;
        else if (lo)
            out_data <= bus;
    end
endmodule

// 10. SAP-2 TOP MODULE
module sap2(
    input clk,
    input rst,
    output [7:0] display_out,
    output halt_led
);
    wire [7:0] bus;
    wire cp, ep, lm, lm_high, ce, we, li, ei, eu, lf, lo, hlt;
    wire [3:0] alu_op;
    wire [2:0] reg_rd1, reg_rd2, reg_wr;
    wire reg_wr_en, reg_to_bus, reg2_to_bus, load_pc;
    wire [15:0] pc_count, mar_addr;
    wire [7:0] ir_opcode, ir_operand;
    wire [7:0] reg_data1, reg_data2, alu_result;
    wire flag_z, flag_c, flag_s, flag_p, z_reg, c_reg, s_reg, p_reg;

    counter PC(.clk(clk), .rst(rst), .cp(cp), .ep(ep),
               .jump_addr(mar_addr), .load_pc(load_pc),
               .count(pc_count), .bus(bus));

    mar MAR(.clk(clk), .rst(rst), .lm(lm), .lm_high(lm_high),
            .bus(bus), .address(mar_addr));

    ram RAM(.clk(clk), .address(mar_addr), .ce(ce), .we(we),
            .data_in(bus), .bus(bus));

    instruction_register IR(.clk(clk), .rst(rst), .li(li), .ei(ei),
                            .bus(bus), .opcode(ir_opcode),
```

```
                        .operand(ir_operand), .bus_out(bus));

    register_file REGS(.clk(clk), .rst(rst),
                       .rd_addr1(reg_rd1), .rd_addr2(reg_rd2),
                       .wr_addr(reg_wr), .wr_en(reg_wr_en),
                       .bus(bus), .rd_data1(reg_data1),
                       .rd_data2(reg_data2));

    alu ALU(.a_data(reg_data1), .b_data(reg_data2), .alu_op(alu_op),
            .eu(eu), .result(alu_result),
            .flag_z(flag_z), .flag_c(flag_c),
            .flag_s(flag_s), .flag_p(flag_p), .bus(bus));

    flags_register FLAGS(.clk(clk), .rst(rst), .lf(lf),
                         .flag_z(flag_z), .flag_c(flag_c),
                         .flag_s(flag_s), .flag_p(flag_p),
                         .z(z_reg), .c(c_reg), .s(s_reg), .p(p_reg));

    controller CTRL(.clk(clk), .rst(rst), .opcode(ir_opcode),
                    .flag_z(z_reg), .flag_c(c_reg),
                    .cp(cp), .ep(ep), .lm(lm), .lm_high(lm_high),
                    .ce(ce), .we(we), .li(li), .ei(ei),
                    .alu_op(alu_op), .eu(eu), .lf(lf),
                    .reg_rd1(reg_rd1), .reg_rd2(reg_rd2),
                    .reg_wr(reg_wr), .reg_wr_en(reg_wr_en),
                    .reg_to_bus(reg_to_bus), .reg2_to_bus(reg2_to_bus),
                    .load_pc(load_pc), .lo(lo), .hlt(hlt));

    output_register OUT_REG(.clk(clk), .rst(rst), .lo(lo),
                        .bus(bus), .out_data(display_out));
    assign bus = reg_to_bus ? reg_data1 : 8'bzzzzzzzz;
    assign halt_led = hlt;
endmodule
```

// TESTBENCH CODE

```
module sap2_tb;
    reg clk, rst;
    wire [7:0] display_out;
    wire halt_led;
    integer test_num;
```

```
sap2 UUT(.clk(clk), .rst(rst), .display_out(display_out), .halt_led(halt_led));
initial begin
   clk = 0;
   forever #5 clk = ~clk;
end
task run_test;
   input [7:0] expected;
   input [200*8:1] description;
   integer cycle;
   begin
      rst = 1; cycle = 0; #15 rst = 0;
      $display("\n>>> Test %0d: %0s", test_num, description);
      $display("Expected output: %0d", expected);
      $display("Cyc | PC | T | IR | MAR | RegA | Bus | OUT");
      $display("----|-------|---|-------|-----|------|-----|----");

      while (!halt_led && cycle < 100) begin
         @(posedge clk); #1;
         $display("%3d | %3d | %X | %h | %3d | %3d | %3d | %3d",
                 cycle, UUT.pc_count[7:0], UUT.CTRL.t_state,
                 UUT.ir_opcode, UUT.mar_addr[7:0],
                 UUT.REGS.regs[0], UUT.bus, display_out);
         cycle = cycle + 1;
      end

      if (cycle >= 100) begin
         $display(" ✗ TIMEOUT: Simulation exceeded 100 cycles");
      end else if (display_out == expected) begin
         $display(" ✓ PASS: Output = %0d", display_out);
      end else begin
         $display(" ✗ FAIL: Expected %0d, got %0d", expected, display_out);
      end

      test_num = test_num + 1;
      #20;
   end
endtask

initial begin
   $display("\n========================================");
```

```
$display("  SAP-2 COMPUTER TEST SUITE");
$display("========================================");
test_num = 1;

// TEST 1: LDA and OUT
#10;
UUT.RAM.memory[0] = 8'h3A; UUT.RAM.memory[1] = 8'd16;
UUT.RAM.memory[2] = 8'hD3; UUT.RAM.memory[3] = 8'h76;
UUT.RAM.memory[16] = 8'd42;
run_test(42, "LDA and OUT");

// TEST 2: LDA, ADD, OUT
#10;
UUT.RAM.memory[0] = 8'h3A; UUT.RAM.memory[1] = 8'd16;
UUT.RAM.memory[2] = 8'h86; UUT.RAM.memory[3] = 8'd17;
UUT.RAM.memory[4] = 8'hD3; UUT.RAM.memory[5] = 8'h76;
UUT.RAM.memory[16] = 8'd15; UUT.RAM.memory[17] = 8'd25;
run_test(40, "LDA + ADD");

// TEST 3: LDA, SUB, OUT
#10;
UUT.RAM.memory[0] = 8'h3A; UUT.RAM.memory[1] = 8'd16;
UUT.RAM.memory[2] = 8'h96; UUT.RAM.memory[3] = 8'd17;
UUT.RAM.memory[4] = 8'hD3; UUT.RAM.memory[5] = 8'h76;
UUT.RAM.memory[16] = 8'd50; UUT.RAM.memory[17] = 8'd30;
run_test(20, "LDA + SUB");

// TEST 4: MVI
#10;
UUT.RAM.memory[0] = 8'h3E; UUT.RAM.memory[1] = 8'd99;
UUT.RAM.memory[2] = 8'hD3; UUT.RAM.memory[3] = 8'h76;
run_test(99, "MVI (Move Immediate)");

// TEST 5: INR
#10;
UUT.RAM.memory[0] = 8'h3E; UUT.RAM.memory[1] = 8'd10;
UUT.RAM.memory[2] = 8'h3C; UUT.RAM.memory[3] = 8'hD3;
UUT.RAM.memory[4] = 8'h76;
run_test(11, "INR (Increment)");
```

```
        $display("\n==========================================");
        $display("  SAP-2 TEST SUITE COMPLETED");
        $display("==========================================\n");
        $finish;
    end
endmodule
```

Command for RTL Source Code

```
iverilog -o sap2 sap2.v
./sap2
```

Command for Netlist Simulation

```
# read modules from Verilog file
read_verilog sap2.v
#hierarchy top
hierarchy -top sap2
# translate processes to netlists
proc
# remove unused cells and wires
clean
# show the generic netlist
show sap2
#perform optimization
opt
#this convers ram to flip-flops
memory
#resource sharing optimization
share -aggressive
# show the generic netlist
show sap2
# mapping to internal cell library
techmap
# mapping flip-flops to toy.lib
dfflibmap -liberty NangateOCL.lib
```

```
# mapping logic to toy.lib

abc -liberty NangateOCL.lib

# remove unused cells and wires

clean

#clean up optimization

opt_clean -purge

#report design statistics

stat -liberty NangateOCL.lib

# Write the current design to a Verilog file

write_verilog -noattr -noexpr -nohex -nodec netlist_sap2.v
```

Command for Timing Power Check

```
sta

read_liberty NangateOCL.lib

read_verilog netlist_sap2_fix.v

link_design sap2

read_sdc top.sdc

report_checks -path_delay max

report_checks -path_delay min

report_power
```

Command for Physical Design

```
openroad -gui place_and_view_sap3_simple.tcl
```

RTL Source Code (SAP-3)

```
//=============================================================
// SAP-3 (8080-compatible) Computer Implementation in Verilog
// - 8-bit data bus, 16-bit address bus
// - 6 general purpose registers: B, C, D, E, H, L (can pair: BC, DE, HL)
// - 16-bit PC and SP
// - Stack operations (PUSH/POP)
// - Subroutine CALL/RET
// - Flags: Zero, Carry, Sign, Parity
//=============================================================
```

```
// 1. PROGRAM COUNTER (16-bit)
module counter(
    input clk,
    input rst,
    input cp,
    input ep_lo,
    input ep_hi,
    input lp,
    input [15:0] bus_addr,
    output reg [15:0] count,
    inout [7:0] bus
);
    always @(posedge clk or posedge rst) begin
        if (rst)
            count <= 16'h0000;
        else if (lp)
            count <= bus_addr;
        else if (cp)
            count <= count + 1;
    end

    assign bus = ep_lo ? count[7:0] : (ep_hi ? count[15:8] : 8'hzz);
endmodule

// 2. STACK POINTER (16-bit)
module stack_pointer(
    input clk,
    input rst,
    input sp_inc,
    input sp_dec,
    input ls,
    input es_lo,
    input es_hi,
    input [15:0] bus_addr,
    output reg [15:0] sp,
    inout [7:0] bus
);
    always @(posedge clk or posedge rst) begin
        if (rst)
```

```
            sp <= 16'hFFFF;
        else if (ls)
            sp <= bus_addr;
        else if (sp_inc)
            sp <= sp + 1;
        else if (sp_dec)
            sp <= sp - 1;
    end

    assign bus = es_lo ? sp[7:0] : (es_hi ? sp[15:8] : 8'hzz);
endmodule

// 3. MEMORY ADDRESS REGISTER (16-bit)
module mar(
    input clk,
    input rst,
    input lm_lo,
    input lm_hi,
    input [7:0] bus,
    output reg [15:0] address
);
    always @(posedge clk or posedge rst) begin
        if (rst)
            address <= 16'h0000;
        else if (lm_lo)
            address[7:0] <= bus;
        else if (lm_hi)
            address[15:8] <= bus;
    end
endmodule

// 4. RAM (256 bytes)
module ram(
    input clk,
    input [15:0] address,
    input ce,
    input we,
    input [7:0] bus_in,
    inout [7:0] bus
);
```

```
    reg [7:0] memory [0:255];
    integer i;

    initial begin
        for (i = 0; i < 256; i = i + 1)
            memory[i] = 8'h00;
    end

    always @(posedge clk) begin
        if (we && address < 256)
            memory[address[7:0]] <= bus_in;
    end
    assign bus = (ce && address < 256) ? memory[address[7:0]] : 8'hzz;
endmodule

// 5. INSTRUCTION REGISTER (8-bit)
module instruction_register(
    input clk,
    input rst,
    input li,
    input ei,
    input [7:0] bus,
    output reg [7:0] opcode,
    inout [7:0] bus_out
);
    always @(posedge clk or posedge rst) begin
        if (rst)
            opcode <= 8'h00;
        else if (li)
            opcode <= bus;
    end

    assign bus_out = ei ? opcode : 8'hzz;
endmodule

// 6. REGISTER FILE (B, C, D, E, H, L)
module register_file(
    input clk,
    input rst,
    input lb, input lc, input ld, input le, input lh, input ll,
```

```
    input eb, input ec, input ed, input ee, input eh, input el,
    input [7:0] bus,
    output reg [7:0] b_data, c_data, d_data, e_data, h_data, l_data,
    inout [7:0] bus_out
);
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            b_data <= 8'h00; c_data <= 8'h00;
            d_data <= 8'h00; e_data <= 8'h00;
            h_data <= 8'h00; l_data <= 8'h00;
        end else begin
            if (lb) b_data <= bus;
            if (lc) c_data <= bus;
            if (ld) d_data <= bus;
            if (le) e_data <= bus;
            if (lh) h_data <= bus;
            if (ll) l_data <= bus;
        end
    end

    assign bus_out = eb ? b_data :
                    ec ? c_data :
                    ed ? d_data :
                    ee ? e_data :
                    eh ? h_data :
                    el ? l_data : 8'hzz;
endmodule

// 7. ACCUMULATOR
module accumulator(
    input clk,
    input rst,
    input la,
    input ea,
    input [7:0] bus,
    output reg [7:0] acc_data,
    inout [7:0] bus_out
);
    always @(posedge clk or posedge rst) begin
        if (rst)
```

```
            acc_data <= 8'h00;
        else if (la)
            acc_data <= bus;
    end

    assign bus_out = ea ? acc_data : 8'hzz;
endmodule

// 8. TEMP REGISTERS (W and Z for 16-bit operations)
module temp_registers(
    input clk,
    input rst,
    input lw, input lz,
    input ew, input ez,
    input [7:0] bus,
    output reg [7:0] w_data, z_data,
    inout [7:0] bus_out
);
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            w_data <= 8'h00;
            z_data <= 8'h00;
        end else begin
            if (lw) w_data <= bus;
            if (lz) z_data <= bus;
        end
    end

    assign bus_out = ew ? w_data : (ez ? z_data : 8'hzz);
endmodule

// 9. ALU (8-bit with flags)
module alu(
    input [7:0] a_data,
    input [7:0] b_data,
    input [3:0] alu_op,
    input eu,
    output reg [7:0] result,
    output reg zero_flag,
    output reg carry_flag,
```

```
    output reg sign_flag,
    output reg parity_flag,
    inout [7:0] bus
);
    reg [8:0] temp_result;
    integer i, parity_count;

    parameter ALU_ADD = 4'h0;
    parameter ALU_SUB = 4'h1;
    parameter ALU_AND = 4'h2;
    parameter ALU_OR  = 4'h3;
    parameter ALU_XOR = 4'h4;
    parameter ALU_CMP = 4'h5;
    parameter ALU_INC = 4'h6;
    parameter ALU_DEC = 4'h7;
    parameter ALU_RLC = 4'h8;
    parameter ALU_RRC = 4'h9;

    always @(*) begin
        carry_flag = 0;
        temp_result = 9'h0;
        result = 8'h00;

        case (alu_op)
            ALU_ADD: begin
                temp_result = a_data + b_data;
                carry_flag = temp_result[8];
                result = temp_result[7:0];
            end
            ALU_SUB: begin
                temp_result = a_data - b_data;
                carry_flag = temp_result[8];
                result = temp_result[7:0];
            end
            ALU_AND: result = a_data & b_data;
            ALU_OR:  result = a_data | b_data;
            ALU_XOR: result = a_data ^ b_data;
            ALU_CMP: begin
                temp_result = a_data - b_data;
                carry_flag = temp_result[8];
```

```
            result = temp_result[7:0];
        end
        ALU_INC: begin
            temp_result = a_data + 1;
            result = temp_result[7:0];
        end
        ALU_DEC: begin
            temp_result = a_data - 1;
            result = temp_result[7:0];
        end
        ALU_RLC: begin
            result = {a_data[6:0], a_data[7]};
            carry_flag = a_data[7];
        end
        ALU_RRC: begin
            result = {a_data[0], a_data[7:1]};
            carry_flag = a_data[0];
        end
        default: result = a_data;
      endcase

      zero_flag = (result == 8'h00);
      sign_flag = result[7];

      parity_count = 0;
      for (i = 0; i < 8; i = i + 1)
        if (result[i]) parity_count = parity_count + 1;
      parity_flag = (parity_count[0] == 0);
  end

  assign bus = eu ? result : 8'hzz;
endmodule

// 10. CONTROLLER/SEQUENCER
module controller(
  input clk,
  input rst,
  input [7:0] opcode,
  input zero_flag,
  input carry_flag,
```

```
    input sign_flag,
    input parity_flag,
    output reg cp,
    output reg ep_lo, ep_hi,
    output reg lp,
    output reg lm_lo, lm_hi,
    output reg ce, we,
    output reg li, ei,
    output reg la, ea,
    output reg lb, lc, ld, le, lh, ll,
    output reg eb, ec, ed, ee, eh, el,
    output reg lw, lz, ew, ez,
    output reg sp_inc, sp_dec,
    output reg es_lo, es_hi,
    output reg [3:0] alu_op,
    output reg eu,
    output reg lo,
    output reg hlt
);
    reg [3:0] t_state;
    reg [7:0] instruction;

    // Simplified instruction set
    parameter NOP   = 8'h00;
    parameter MVI_B = 8'h06;
    parameter MVI_C = 8'h0E;
    parameter MVI_A = 8'h3E;
    parameter MOV_AB = 8'h78;
    parameter MOV_AC = 8'h79;
    parameter ADD_B = 8'h80;
    parameter ADD_C = 8'h81;
    parameter SUB_B = 8'h90;
    parameter ANA_B = 8'hA0;
    parameter ORA_B = 8'hB0;
    parameter XRA_B = 8'hA8;
    parameter CMP_B = 8'hB8;
    parameter INR_A = 8'h3C;
    parameter DCR_A = 8'h3D;
    parameter OUT   = 8'hD3;
    parameter HLT   = 8'h76;
```

```
always @(posedge clk or posedge rst) begin
    if (rst) begin
        t_state <= 4'h0;
        instruction <= 8'h00;
        cp <= 0; ep_lo <= 0; ep_hi <= 0; lp <= 0;
        lm_lo <= 0; lm_hi <= 0; ce <= 0; we <= 0;
        li <= 0; ei <= 0; la <= 0; ea <= 0;
        lb <= 0; lc <= 0; ld <= 0; le <= 0; lh <= 0; ll <= 0;
        eb <= 0; ec <= 0; ed <= 0; ee <= 0; eh <= 0; el <= 0;
        lw <= 0; lz <= 0; ew <= 0; ez <= 0;
        sp_inc <= 0; sp_dec <= 0; es_lo <= 0; es_hi <= 0;
        alu_op <= 0; eu <= 0; lo <= 0; hlt <= 0;
    end
    else if (!hlt) begin
        // Clear all control signals
        cp <= 0; ep_lo <= 0; ep_hi <= 0; lp <= 0;
        lm_lo <= 0; lm_hi <= 0; ce <= 0; we <= 0;
        li <= 0; ei <= 0; la <= 0; ea <= 0;
        lb <= 0; lc <= 0; ld <= 0; le <= 0; lh <= 0; ll <= 0;
        eb <= 0; ec <= 0; ed <= 0; ee <= 0; eh <= 0; el <= 0;
        lw <= 0; lz <= 0; ew <= 0; ez <= 0;
        sp_inc <= 0; sp_dec <= 0; es_lo <= 0; es_hi <= 0;
        alu_op <= 0; eu <= 0; lo <= 0;

        case (t_state)
            // FETCH cycle
            4'h0: begin
                ep_lo <= 1;
                lm_lo <= 1;
                t_state <= 4'h1;
            end

            4'h1: begin
                ce <= 1;
                li <= 1;
                t_state <= 4'h2;
            end

            4'h2: begin
```

```
    cp <= 1;
    t_state <= 4'h3;
end

// DECODE cycle
4'h3: begin
    instruction <= opcode;
    t_state <= 4'h7;
end

// EXECUTE
4'h7: begin
    case (instruction)
        NOP: t_state <= 4'h0;

        MVI_B, MVI_C, MVI_A: begin
            ep_lo <= 1;
            lm_lo <= 1;
            t_state <= 4'h4;
        end

        MOV_AB: begin
            eb <= 1;
            la <= 1;
            t_state <= 4'h0;
        end

        MOV_AC: begin
            ec <= 1;
            la <= 1;
            t_state <= 4'h0;
        end

        ADD_B: begin
            alu_op <= 4'h0;
            eu <= 1;
            la <= 1;
            t_state <= 4'h0;
        end
```

```
ADD_C: begin
    alu_op <= 4'h0;
    eu <= 1;
    la <= 1;
    t_state <= 4'h0;
end

SUB_B: begin
    alu_op <= 4'h1;
    eu <= 1;
    la <= 1;
    t_state <= 4'h0;
end

ANA_B: begin
    alu_op <= 4'h2;
    eu <= 1;
    la <= 1;
    t_state <= 4'h0;
end

ORA_B: begin
    alu_op <= 4'h3;
    eu <= 1;
    la <= 1;
    t_state <= 4'h0;
end

XRA_B: begin
    alu_op <= 4'h4;
    eu <= 1;
    la <= 1;
    t_state <= 4'h0;
end

INR_A: begin
    alu_op <= 4'h6;
    eu <= 1;
    la <= 1;
    t_state <= 4'h0;
```

```
        end

        DCR_A: begin
            alu_op <= 4'h7;
            eu <= 1;
            la <= 1;
            t_state <= 4'h0;
        end

        OUT: begin
            ea <= 1;
            lo <= 1;
            t_state <= 4'h0;
        end

        HLT: begin
            hlt <= 1;
            t_state <= 4'h0;
        end

        default: t_state <= 4'h0;
    endcase
end

4'h4: begin
    cp <= 1;
    ce <= 1;
    case (instruction)
        MVI_A: la <= 1;
        MVI_B: lb <= 1;
        MVI_C: lc <= 1;
    endcase
    t_state <= 4'h0;
end

4'h5: begin
    alu_op <= 4'h0;
    eu <= 1;
    la <= 1;
    t_state <= 4'h0;
```

```
            end

            default: t_state <= 4'h0;
          endcase
      end
   end
endmodule

// 11. OUTPUT REGISTER
module output_register(
   input clk,
   input rst,
   input lo,
   input [7:0] bus,
   output reg [7:0] out_data
);
   always @(posedge clk or posedge rst) begin
      if (rst)
         out_data <= 8'h00;
      else if (lo)
         out_data <= bus;
   end
endmodule

// 12. TOP MODULE - SAP-3
module sap3(
   input clk,
   input rst,
   output [7:0] display_out,
   output halt_led,
   output zero_led,
   output carry_led,
   output sign_led,
   output parity_led
);
   wire [7:0] bus;
   wire [15:0] pc_count, sp_val, mar_addr;
   wire [7:0] ir_opcode;
   wire [7:0] acc_data, b_data, c_data, d_data, e_data, h_data, l_data;
   wire [7:0] w_data, z_data;
```

```
wire zero_flag, carry_flag, sign_flag, parity_flag;
wire cp, ep_lo, ep_hi, lp, lm_lo, lm_hi, ce, we, li, ei;
wire la, ea, lb, lc, ld, le, lh, ll, eb, ec, ed, ee, eh, el;
wire lw, lz, ew, ez, sp_inc, sp_dec, es_lo, es_hi, eu, lo, hlt;
wire [3:0] alu_op;
counter PC(
    .clk(clk), .rst(rst), .cp(cp), .ep_lo(ep_lo), .ep_hi(ep_hi),
    .lp(lp), .bus_addr(mar_addr), .count(pc_count), .bus(bus)
);

stack_pointer SP(
    .clk(clk), .rst(rst), .sp_inc(sp_inc), .sp_dec(sp_dec),
    .ls(1'b0), .es_lo(es_lo), .es_hi(es_hi),
    .bus_addr(16'h0000), .sp(sp_val), .bus(bus)
);
mar MAR(
    .clk(clk), .rst(rst), .lm_lo(lm_lo), .lm_hi(lm_hi),
    .bus(bus), .address(mar_addr)
);
ram RAM(
    .clk(clk), .address(mar_addr), .ce(ce), .we(we),
    .bus_in(bus), .bus(bus)
);
instruction_register IR(
    .clk(clk), .rst(rst), .li(li), .ei(ei),
    .bus(bus), .opcode(ir_opcode), .bus_out(bus)
);
controller CTRL(
    .clk(clk), .rst(rst), .opcode(ir_opcode),
    .zero_flag(zero_flag), .carry_flag(carry_flag),
    .sign_flag(sign_flag), .parity_flag(parity_flag),
    .cp(cp), .ep_lo(ep_lo), .ep_hi(ep_hi), .lp(lp),
    .lm_lo(lm_lo), .lm_hi(lm_hi), .ce(ce), .we(we),
    .li(li), .ei(ei), .la(la), .ea(ea),
    .lb(lb), .lc(lc), .ld(ld), .le(le), .lh(lh), .ll(ll),
    .eb(eb), .ec(ec), .ed(ed), .ee(ee), .eh(eh), .el(el),
    .lw(lw), .lz(lz), .ew(ew), .ez(ez),
    .sp_inc(sp_inc), .sp_dec(sp_dec), .es_lo(es_lo), .es_hi(es_hi),
    .alu_op(alu_op), .eu(eu), .lo(lo), .hlt(hlt)
);
```

```
    accumulator ACC(
        .clk(clk), .rst(rst), .la(la), .ea(ea),
        .bus(bus), .acc_data(acc_data), .bus_out(bus)
    );

    register_file REGS(
        .clk(clk), .rst(rst),
        .lb(lb), .lc(lc), .ld(ld), .le(le), .lh(lh), .ll(ll),
        .eb(eb), .ec(ec), .ed(ed), .ee(ee), .eh(eh), .el(el),
        .bus(bus),
        .b_data(b_data), .c_data(c_data), .d_data(d_data),
        .e_data(e_data), .h_data(h_data), .l_data(l_data),
        .bus_out(bus)
    );

    temp_registers TEMPS(
        .clk(clk), .rst(rst), .lw(lw), .lz(lz), .ew(ew), .ez(ez),
        .bus(bus), .w_data(w_data), .z_data(z_data), .bus_out(bus)
    );
    alu ALU(
        .a_data(acc_data), .b_data(b_data), .alu_op(alu_op), .eu(eu),
        .result(), .zero_flag(zero_flag), .carry_flag(carry_flag),
        .sign_flag(sign_flag), .parity_flag(parity_flag), .bus(bus)
    );
    output_register OUT_REG(
        .clk(clk), .rst(rst), .lo(lo),
        .bus(bus), .out_data(display_out)
    );
    assign halt_led = hlt;
    assign zero_led = zero_flag;
    assign carry_led = carry_flag;
    assign sign_led = sign_flag;
    assign parity_led = parity_flag;
endmodule
```

// TESTBENCH CODE

```
module sap3_tb;
    reg clk, rst;
    wire [7:0] display_out;
```

```
wire halt_led, zero_led, carry_led, sign_led, parity_led;
integer cycle_count;
sap3 UUT(
    .clk(clk),
    .rst(rst),
    .display_out(display_out),
    .halt_led(halt_led),
    .zero_led(zero_led),
    .carry_led(carry_led),
    .sign_led(sign_led),
    .parity_led(parity_led)
);
initial begin
    clk = 0;
    forever #5 clk = ~clk;
end
initial begin
    $display("\n========================================");
    $display("  SAP-3 Computer Simulation (8080-like)");
    $display("========================================\n");

    rst = 1;
    cycle_count = 0;
    #10;
    // Program: MVI A,5; INR A; MVI B,10; ADD B; OUT; HLT
    UUT.RAM.memory[0]  = 8'h3E;  // MVI A
    UUT.RAM.memory[1]  = 8'h05;  // 5
    UUT.RAM.memory[2]  = 8'h3C;  // INR A
    UUT.RAM.memory[3]  = 8'h06;  // MVI B
    UUT.RAM.memory[4]  = 8'h0A;  // 10
    UUT.RAM.memory[5]  = 8'h80;  // ADD B
    UUT.RAM.memory[6]  = 8'hD3;  // OUT
    UUT.RAM.memory[7]  = 8'h76;  // HLT

    $display("Program:");
    $display("  0x0: MVI A, 5");
    $display("  0x2: INR A");
    $display("  0x3: MVI B, 10");
    $display("  0x5: ADD B");
    $display("  0x6: OUT");
```

```
        $display("  0x7: HLT");
        $display("\nExpected: 5 + 1 + 10 = 16\n");
        #5 rst = 0;

        $display("Time | Cyc | PC |State| IR | ACC | B  | C  | OUT | Flags");
        $display("-----|-----|-----|-----|----|----|----|----|-----|-------");

        wait(halt_led == 1);
        #20;
        $display("\n========================================");
        $display("HALTED - Final Output: %d (0x%h)", display_out, display_out);
        $display("Flags: Z=%b C=%b S=%b P=%b", zero_led, carry_led, sign_led, parity_led);
        $display("========================================");

        if (display_out == 8'd16)
            $display("\n*** TEST PASSED! ***\n");
        else
            $display("\n*** FAILED! Expected 16, got %d ***\n", display_out);
        $finish;
    end

    always @(posedge clk) begin
        if (!rst) begin
            cycle_count = cycle_count + 1;
            $display("%4t | %3d | %3h | T%1h | %2h | %2d | %2d | %2d | %3d | %b%b%b%b",
                     $time, cycle_count, UUT.pc_count, UUT.CTRL.t_state,
                     UUT.ir_opcode, UUT.acc_data, UUT.b_data, UUT.c_data,
                     display_out, zero_led, carry_led, sign_led, parity_led);
        end
    end
endmodule
```

Command for RTL

```
iverilog -o sap3 sap3.v
./sap3
```

Command for Netlist Simulation

```
# read modules from Verilog file
read_verilog sap3.v
#hierarchy top
hierarchy -top sap3
# translate processes to netlists
```

```
proc
# remove unused cells and wires
clean
# show the generic netlist
show sap3
#perform optimization
opt
#this convers ram to flip-flops
memory
#resource sharing optimization
share -aggressive
# show the generic netlist
show sap3
# mapping to internal cell library
techmap
# mapping flip-flops to toy.lib
dfflibmap -liberty NangateOCL.lib
# mapping logic to toy.lib
abc -liberty NangateOCL.lib
# remove unused cells and wires
clean
#clean up optimization
opt_clean -purge
#report design statistics
stat -liberty NangateOCL.lib
# Write the current design to a Verilog file
write_verilog -noattr -noexpr -nohex -nodec netlist_sap3.v
```

Command for Timing Power Check

```
sta
read_liberty NangateOCL.lib
read_verilog netlist_sap3.v
link_design sap3
read_sdc top.sdc
report_checks -path_delay max
report_checks -path_delay min
report_power
```

Command for Physical Design

```
openroad -gui place_and_view_sap3_simple.tcl
```