Université Cadi Ayyad

Ecole Nationale des Sciences Appliquées

Marrakech

Final Report

Efficient RISC-V Processor Design: Classic Multicycle, Optimized, and Pipelined Architectures on FPGA

GE32 : SED et PROJET DE SEMESTRE et SEMINAIRES

Filière : Ingénierie des Systèmes Electroniques Embarqués et Commande des Systèmes (S2ECS)

Title :

RISC-V Processor Implementations on DE2-SoC FPGA

• By :

AGHADJOUR Zakaria
OUELD EL HAIRECH Brahim
ETTOUGUI Hoda
ENNACHAT Firdaous
ELMADI Choaib
EL HAZMIRI Ayoub

•Supervised By :

Mr.Abdelkarim Hamzaoui

Academic year : 2024/2025

Acknowledgements

- First and foremost, we would like to express our deep gratitude to all those who supported and accompanied us throughout the completion of this **project**, both academically and personally.
- We especially thank **Mr**.**Abdelkarim Hamzaoui**, professor at **ENSA Marrakech**, for his kind supervision, methodological guidance, and for his availability and patience throughout this project. His constructive feedback and expertise greatly contributed to enriching this work.
- We also wish to thank all **the teachers** and **faculty members** of **ENSA Marrakech** for the quality of the education they have provided during our studies, and for equipping us with the skills necessary to complete this project.
- A big thank you as well to our **fellow classmates** and **friends** for the valuable exchanges, technical discussions, and mutual support throughout this journey.
- Finally, we would like to thank our **families** and **loved ones** for their constant support, understanding, and encouragement, which have been essential to the success of **this work**.

Summary

→ Appendices :

- **RISC-V —** *Reduced Instruction Set Computer - Version V*
- **ISA —** *Instruction Set Architecture*
- **ALU —** *Arithmetic Logic Unit*
- **PC —** *Program Counter*
- **CSR —** *Control and Status Register*
- **RTL —** *Register Transfer Level*
- **FSM —** *Finite State Machine*
- **SOC —** *System on Chip*
- **MMU —** *Memory Management Unit*
- **FPU —** *Floating-Point Unit*
- **IR —** *Instruction Register*
- **ID/EX, EX/MEM, MEM/WB —** *Pipeline Register Stages*
- **IF —** *Instruction Fetch*
- **ID —** *Instruction Decode*
- **EX —** *Execute*
- **MEM —** *Memory Access*
- **WB —** *Write Back*
- **CU —** *Control Unit*
- **MUX —** *Multiplexer*
- **RegFile —** *Register File*
- **Hazard Unit —** *Pipeline Hazard Detection Logic*
- **Bypassing/Forwarding —** *Technique to avoid pipeline stalls*
- **Stall —** *Temporary Halt in Pipeline Flow*

- **NOP —** *No Operation* (used for stalling or padding)
- **Load-Use Hazard —** *Pipeline delay due to data dependency*
- **Structural Hazard —** *Conflict for hardware resources*
- **Data Hazard —** *Read-after-Write Conflict*
- **Control Hazard —** *Branch Prediction Conflict*
- **Branch Delay Slot —** *Instruction after branch executed regardless of outcome*
- **Multicycle Processor —** *Processor with variable clock cycles per instruction*
- **Pipeline Processor —** *Processor with overlapped instruction execution*
- **Harvard Architecture —** *Separate instruction and data memory*
- **Von Neumann Architecture —** *Shared instruction/data memory*
- **Instruction Latency —** *Time to complete an instruction*
- **Throughput —** *Number of instructions executed per unit time*
- **Microarchitecture —** *Low-level implementation of an ISA*
- **Bit-Slice Design —** *Modular ALU design for scaling word width*
- **Single-Cycle Processor —** *Each instruction executed in one clock cycle*
- **Control Path —** *Circuit that generates control signals*
- **Data Path —** *Hardware that performs operations (ALU, RegFile, etc.)*
- **AXI —** *Advanced eXtensible Interface*
- **AMBA —** *Advanced Microcontroller Bus Architecture*
- **UART —** *Universal Asynchronous Receiver/Transmitter*
- **SPI —** *Serial Peripheral Interface*
- **I²C —** *Inter-Integrated Circuit*
- **OoOE —** *Out-of-Order Execution*

- **Superscalar —** *Architecture allowing execution of multiple instructions per cycle*
- **Scoreboarding —** *Technique for tracking instruction dependencies and resource usage*
- **Tomasulo's Algorithm —** *Dynamic scheduling algorithm with register renaming and reservation stations*
- **Register Renaming —** *Technique to avoid false dependencies by mapping logical to physical registers*
- **Speculative Execution —** *Executing instructions before the outcome of branches is known*
- **Instruction Fusion —** *Combining multiple instructions into a single micro-operation*
- **Instruction Window —** *Buffer holding instructions before scheduling*
- **Decode Logic —** *Hardware that interprets instruction opcodes into control signals*
- **Clock Gating —** *Power optimization technique by disabling clock to inactive blocks*
- **Bit-width —** *Number of bits processed in parallel by the processor (e.g., 32-bit, 64-bit)*
- **Pipeline Depth —** *Number of stages in the instruction pipeline*
- **Zero-cycle Branch —** *Branch instruction resolved without stalling pipeline*
- **BTB —** *Branch Target Buffer – caches branch target addresses for prediction*
- **TLB —** *Translation Lookaside Buffer – caches address translations in virtual memory*
- **Write Buffer —** *Queue that temporarily holds store data before memory write*
- **Fetch Unit —** *Subsystem responsible for retrieving instructions from memory*
- **Decode Unit —** *Hardware block that decodes fetched instructions*
- **Execution Unit —** *Performs arithmetic, logic, or branch operations*
- **Memory Unit —** *Executes memory read/write operations*
- **Writeback Unit —** *Writes execution results to the register file*
- **Instruction Queue —** *Queue of fetched/dependent instructions waiting for execution*

- **Control Signal —** *Signal directing operations in the data path*
- **Memory-Mapped I/O —** *Method of handling I/O using standard memory address space*
- **Synchronous Design —** *All components triggered by a shared clock signal*
- **Asynchronous Design —** *Design where components use handshaking instead of a global clock*
- **Hardwired Control —** *Control implemented via combinational logic circuits*
- **Microprogrammed Control —** *Control implemented using microcode in a control memory*
- **JTAG —** *Joint Test Action Group – standardized debug and test interface*

1 • Introduction :

1.1 Project Context and Motivation :

The **RISC-V** instruction set architecture **(ISA)** is gaining widespread popularity due to its simplicity, openness, and flexibility for both academic and industrial use. This project explores the design and implementation of RISC-V processor architectures on **FPGA** platforms, with a focus on understanding core design principles, improving performance, and applying practical optimization techniques.

The first phase of the work involves creating a basic multicycle RISC-V processor, following the design approach described in *Computer Organization and Design – RISC-V Edition* by David Money Harris and Sarah Harris. This **core** serves as a starting point to understand how control logic, datapaths, and instruction execution work in a multicycle context.

Next, the same **multicycle** processor is revisited with the goal of making it more efficient in terms of code structure, readability, and synthesis results. Several hardware design optimizations are applied, including one-hot encoding for state machines, use of `parallel_case` statements, minimal decoding logic, and overall cleaner module organization. The optimized version is also supported by a unique interactive documentation that can be found in the appendix, improving traceability and understanding.

Finally, a pipelined version of the processor is developed to explore the benefits and trade-offs of pipelining compared to multicycle execution. This stage involves managing data hazards, instruction flow, and control logic in a more complex but performance-oriented architecture.

This report is structured to present each of these implementations in detail. Each design is tested, verified, and compared in terms of functionality, resource usage, and execution performance. The final sections include a comparison of results, insights gained during development, and ideas for future improvement.

1.2 Objectives :

- The main goal of this project is to design and implement a functional RISC-V processor on an FPGA, starting from a basic multicycle architecture and advancing to a pipelined version. Along the way, the project aims to:

- Understand and apply the fundamental principles of RISC-V processor design.
- Build a classic multicycle core based on the Patterson & Harris textbook.
- Optimize the multicycle core using practical HDL techniques .
- Develop a pipelined core to compare performance and complexity.
- interactive, referenced documentation to support learning and comprehension.
- Comparison of the different designs in terms of resources, speed, and clarity of implementation.

1.3 Tools and Methodologies :

- This project relies on a complete FPGA development toolchain combining design, simulation, synthesis, and implementation tools:

- **Verilog HDL** is used for modeling the processor architectures. It allows precise control over hardware behavior and structure, making it ideal for describing both multicycle and pipelined datapaths, as well as control logic.
- **ModelSim** is the primary tool for **simulation and functional verification**. It is used to verify the correctness of each module, observe waveform behaviors, debug control and datapath interactions, and validate instruction execution cycle-by-cycle. Testbenches are written to automate simulation and confirm expected output against known instruction sequences.
- **Intel Quartus Prime** is used for **synthesis, place-and-route, timing analysis, and bitstream generation**. The tool transforms the Verilog source into a configuration file that can be programmed onto the FPGA. Resource usage, timing reports, and optimization results are also extracted from this stage.
- **Intel FPGA development boards** (such as the DE2 — the board used to implement our RISC-V core) are used for **on-chip implementation and physical testing**. These boards provide a practical platform for observing processor behavior in real time using physical interfaces (e.g., LEDs, switches, memory blocks).

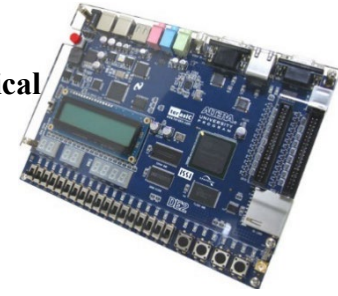


Figure1 : the tools used for the scope of this project

- A structured, bottom-up methodology is followed:

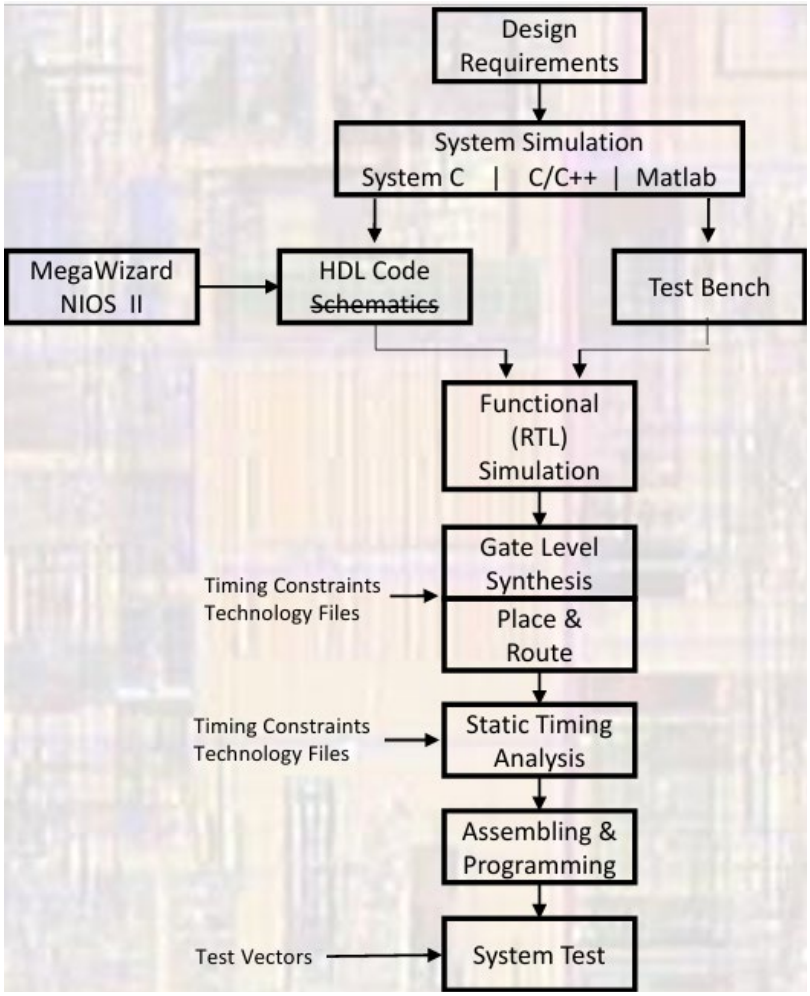


Figure2 : FPGA Design Flow

1.4 Report Structure Overview :

- This report is organized into several main sections:

- **Chapter 2** covers the theoretical background of RISC-V, multicycle and pipelined architectures, and HDL optimization techniques.
- **Chapter 3** presents the initial multicycle processor based on the textbook design.
- **Chapter 4** introduces the optimized version, detailing the improvements and interactive documentation.
- **Chapter 5** describes the pipelined core and how it differs in design and performance.
- **Chapter 6** compares all three implementations in terms of performance and resources.
- **Chapter 7** Dives into the conception of a Custom RISC-V Assembler.

- **Chapter 8** concludes with key takeaways and ideas for future work.

→ Appendices and references are provided at the end for further technical details.

2 • Background and Theoretical Foundations:

2.1 Overview of the RISC-V ISA :

RISC-V is a modular and extensible instruction set architecture (ISA) that adheres to RISC (**R**educed **I**nstruction **S**et **C**omputer) principles. It defines a *small set of base instructions*, with optional standardized extensions, allowing designers to tailor the architecture to specific application domains.

This project targets the **RV32I** base ISA, which defines 32-bit integer operations and uses a **fixed 32-bit instruction format** for all standard instructions. The RV32I ISA supports the following major instruction types:

- **R-type**: Register-register arithmetic/logical (e.g., `add`, `sub`, `and`, `or`)
- **I-type**: Immediate operations and loads (e.g., `addi`, `lw`)
- **S-type**: Store instructions (e.g., `sw`)
- **B-type**: Conditional branches (e.g., `beq`, `bne`)
- **U-type**: Upper immediate operations (e.g., `lui`)
- **J-type**: Unconditional jumps (e.g., `jal`)

31:25	24:20	19:15	14:12	11:7	6:0	
funct7	rs2	rs1	funct3	rd	op	**R-Type**
$imm_{11:0}$		rs1	funct3	rd	op	**I-Type**
$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op	**S-Type**
$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op	**B-Type**
$imm_{31:12}$				rd	op	**U-Type**
$imm_{20,10:1,11,19:12}$				rd	op	**J-Type**

Figure3 : RISC-V 32-bit instruction formats

- Each instruction operates on **32 general-purpose registers (x0–x31)**, where `x0` is hardwired to zero. Instructions are designed to be **encoded compactly**, decoded efficiently, and executed using a regular datapath.

Name	Register Number	Use
zero	x0	Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0–2	x5–7	Temporary registers
s0/fp	x8	Saved register / Frame pointer
s1	x9	Saved register
a0–1	x10–11	Function arguments / Return values
a2–7	x12–17	Function arguments
s2–11	x18–27	Saved registers
t3-6	x28–31	Temporary registers

Figure4 : Register names and numbers

The project focuses on implementing a subset of RV32I sufficient to support simple control structures, arithmetic, logic, memory access, and branching. These are enough to write and execute meaningful programs, such as loops, conditionals, and basic memory manipulation.

From a hardware point of view, RISC-V's clean encoding and load/store architecture make it ideal for a **multicycle or pipelined implementation**. The clear separation between instruction types and consistent instruction length simplify control unit generation and ALU/data memory interfacing. For instance :

▪ Privilege Levels

-RISC-V supports multiple privilege levels (*User*, ***Supervisor***, *Machine*), but this project sticks to Machine mode and Supervisor's systemcalls instructions. Machine mode is the simplest level and enough for embedded systems and FPGA implementation.

▪ Modular ISA Design

RISC-V is modular. The base ISA (RV32I) covers basic integer instructions. Other features like multiplication (M), floating point (F), and **compressed instructions (C)** are added as *extensions*. This helps keep the hardware clean and tailored.

▪ Uniform Instruction Format

All base instructions are 32 bits wide, using consistent formats (R, I, S, B, U, J). This makes decoding easier and helps with clean control logic → especially in multicycle and pipelined designs.

▪ Load/Store Architecture

Only load and store instructions access memory ⟷ everything else works between registers. This simplifies datapath design and keeps memory interactions predictable.

▪ No Condition Flags

RISC-V doesn't use flags like zero, carry, or overflow. Instead, it uses explicit branch instructions such as `beq` or `blt`. This avoids hidden state and makes control simpler to implement.

▪ x0 Register

Register x0 is always zero. Writing to it does nothing. This can be useful for things like `addi x1, x0, 5`, which just loads a constant. It also helps simplify some datapath connections.

Later chapters will detail how these instruction formats are decoded, executed, and optimized in different processor implementations.

2.2 Multicycle Processor Design Principles :

- A multicycle processor splits instruction execution into discrete stages: ***fetch***, ***decode***, ***execute***, ***memory access***, and ***write-back***. Each stage takes one **clock cycle**, and hardware components like the ALU, register file, and memory are reused across cycles.

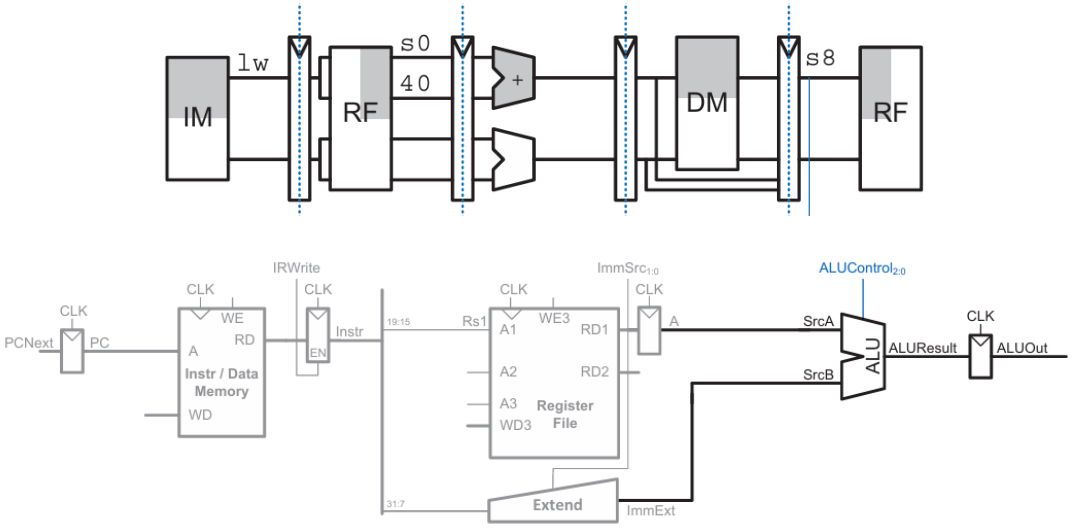


Figure5 : instruction stages and building block elements

The control path is implemented as a finite state machine (FSM), generating control signals specific to the current stage and instruction type. This allows

instructions to take a variable number of cycles depending on their complexity (`lw` takes 5 cycles, `add` takes 4, etc.).

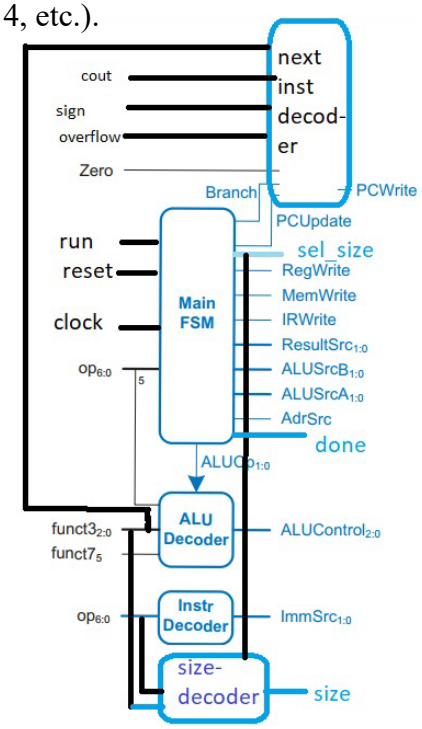


Figure6 : The FSM layout

This design minimizes hardware by avoiding duplication and simplifies timing by allowing each operation to complete **within one cycle**. The **datapath** includes multiplexers to route shared resources and internal registers (e.g. IR, OldPC, A, B, ALUOut) to hold intermediate values between stages.

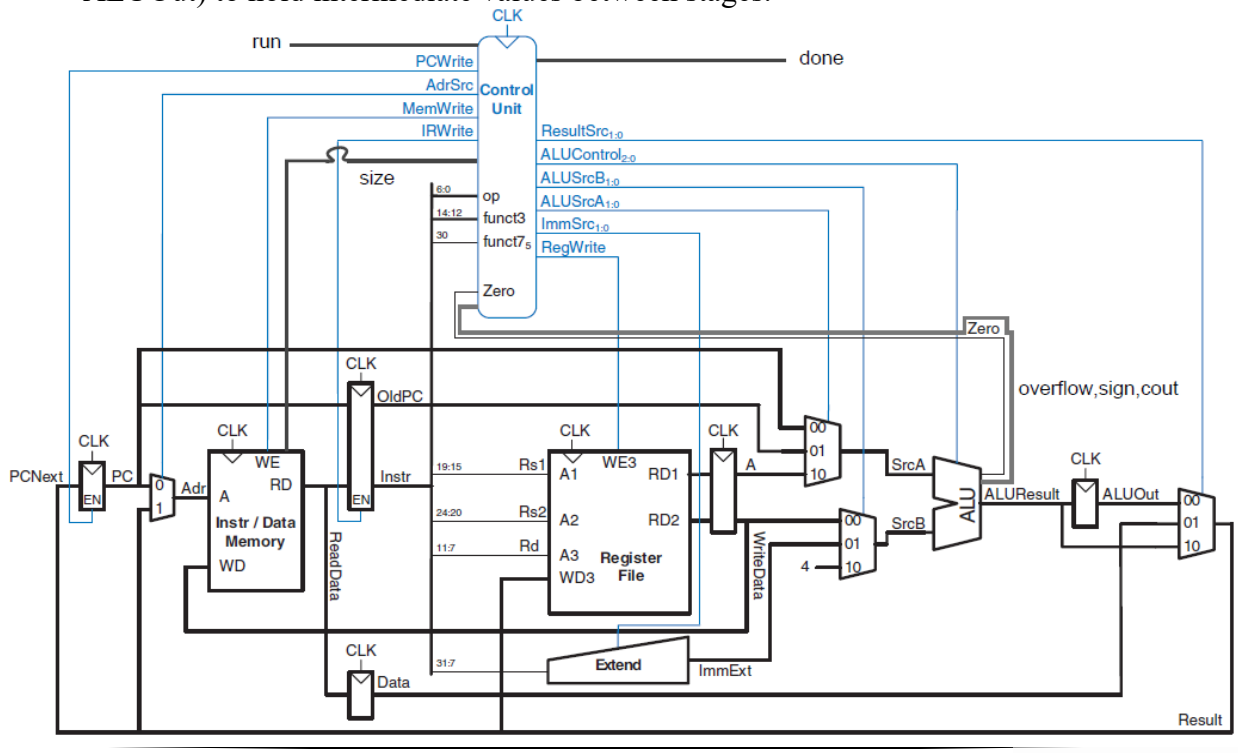


Figure7 : The Complete Multicycle RISC-V Processor

In the single-cycle design, we used **separate** *instruction* and *data memories* because we needed to **read the instruction memory** and **read or write the data memory** *all in one cycle*. Now, we choose to use a combined memory for both instructions and data. This is more realistic and is feasible because we can read the instruction in one cycle, then read or write the data in another cycle,so a von-neuman architecture is chosen as outlined in Patterson & Harris.

This design eliminates hazards by executing one instruction at a time. The FSM allows precise control over timing and memory access. It also serves as a clean base for pipelining, since stages are already defined. The architecture ensures cycle-accurate behavior and simplifies debugging.

2.3 Pipeline Processing Concepts :

Pipelining overlaps instruction execution by dividing it into the same fixed stages: Fetch, Decode, Execute, Memory, and Write-back. Each stage operates in parallel on different instructions using dedicated or shared hardware.

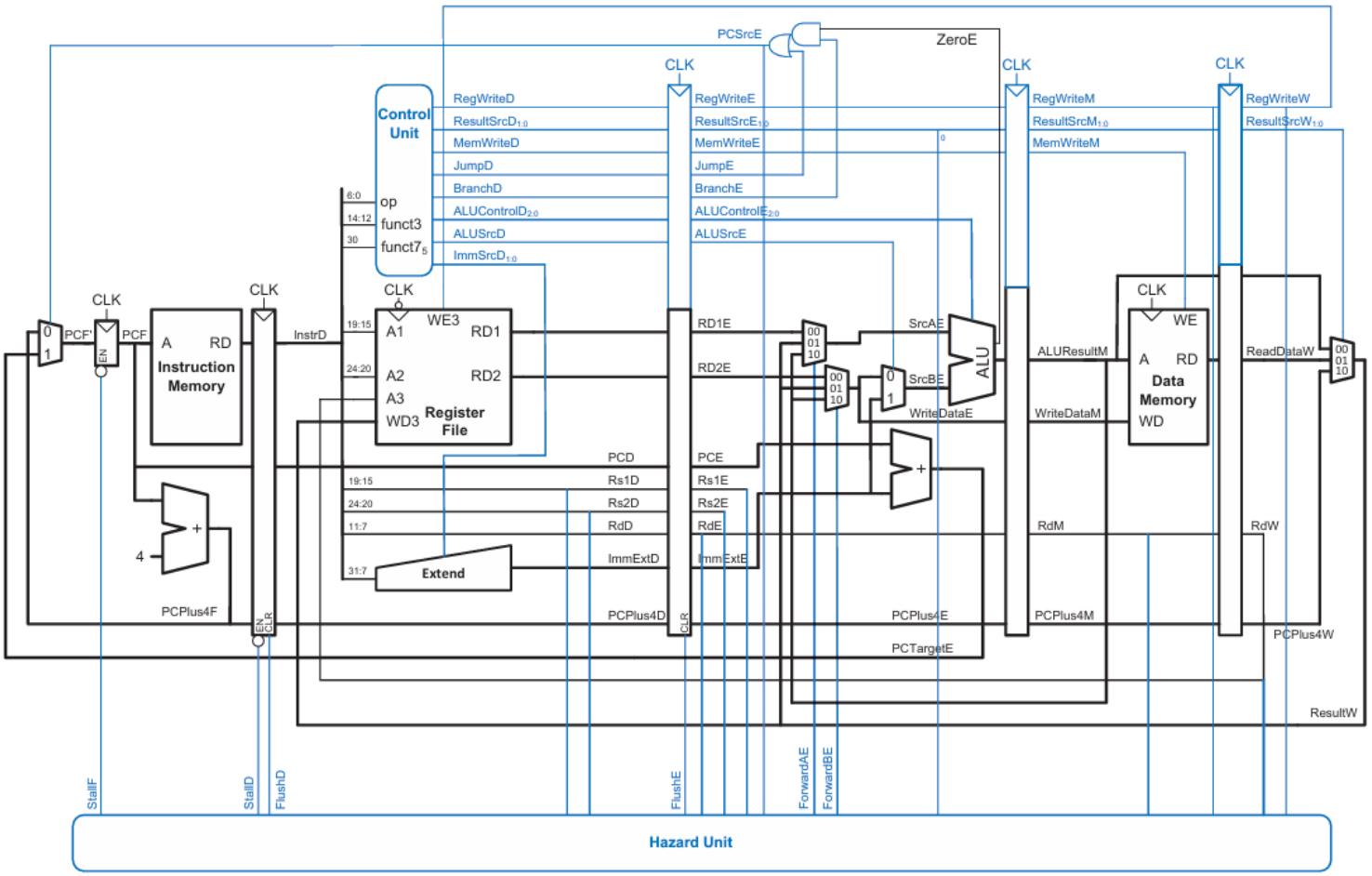


Figure8 : The Complete Pipelined RISC-V Processor

New instructions enter the pipeline each clock cycle, increasing throughput to nearly one instruction per cycle under ideal conditions. Unlike the multicycle design, pipelining introduces hazards:

- **Data hazards** occur when instructions depend on results not yet written.
- **Control hazards** arise from branches and jumps.
- **Structural hazards** happen if multiple stages compete for the same hardware.

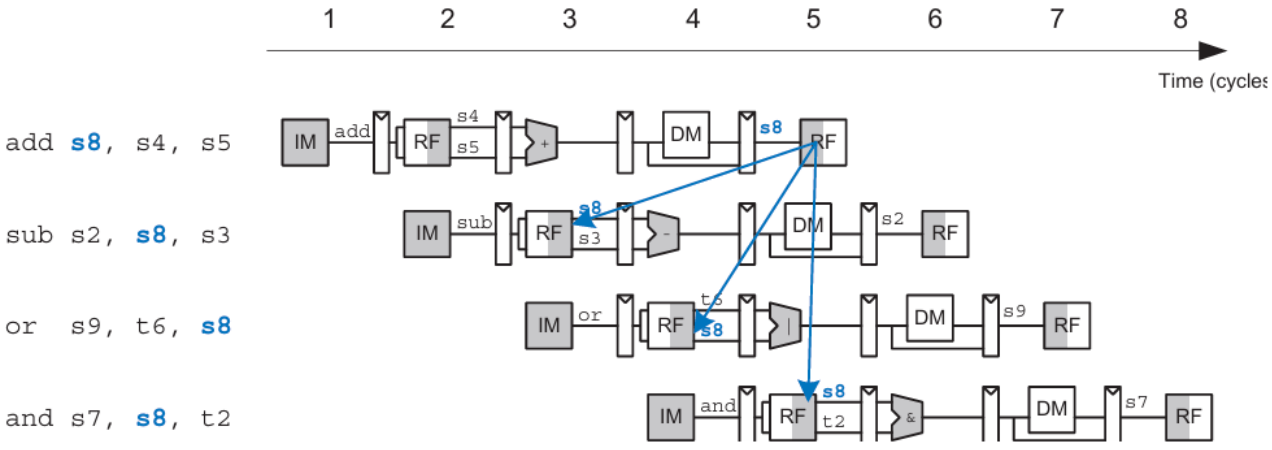


Figure9 : Abstract pipeline diagram illustrating hazards

Hazards are resolved through techniques like forwarding, stalling, and pipeline flushing. **The pipeline control logic** coordinates stage enable signals, hazard detection, and PC updates to maintain correct instruction flow.

This project implements **a classic 5-stage RISC-V pipeline** with basic hazard handling and PC control logic. Memory and register file accesses are synchronized per stage using timing control and handshaking.

Further reading :

- **Register inter-stage buffers**:

Each pipeline stage uses dedicated pipeline registers (e.g., IF/ID, ID/EX, EX/MEM, MEM/WB) to hold data and control signals between cycles.

- **Pipeline control hazards**:

  This implementation assumes basic branch handling via flushing. For future improvement, branch prediction units can be added to reduce stalls.

- **Forwarding logic**:

  A simple forwarding unit is implemented to resolve data hazards between EX and MEM/WB stages, reducing stalls for common ALU desspendencies.

- **Stall mechanism**:
  The pipeline includes a basic hazard detection unit that inserts NOPs when load-use hazards are detected (e.g., `lw` followed by dependent `add`).

- **Memory access latency**:
  The design assumes memory access completes in one cycle (idealized). Multi-cycle memory access would require memory stall logic.

2.4 Optimization Techniques in HDL Design :

In the *BraRV32* processor design, several key techniques are used to make the hardware (the classic implementation) smaller, faster, and easier to implement on FPGAs:

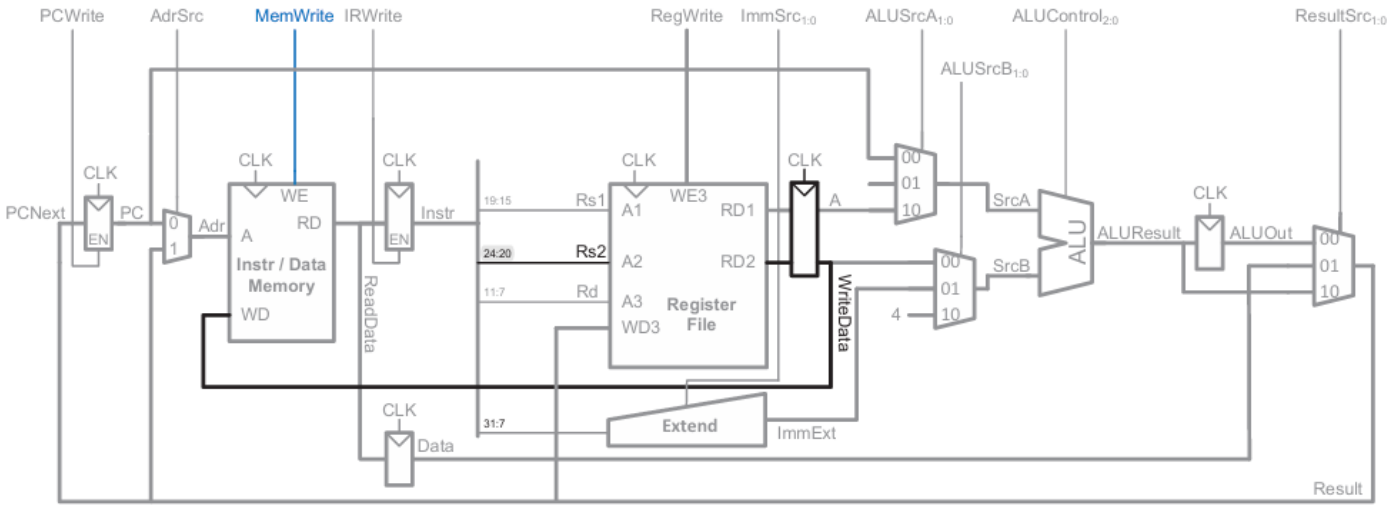
1. **One-Hot State Encoding**
   Instead of using binary numbers for states, each state is represented by a single "hot" bit. This makes the state machine simpler and faster because FPGA logic can handle one-hot states more efficiently → with less logic .
2. **Shared ALU Subtractor**
   A single subtractor circuit is reused for many operations like subtraction, comparison, and equality checks. This saves hardware resources by avoiding duplicated circuits.
3. **Writeback Control Logic**
   The processor only updates registers when necessary. For example, it avoids writing back results during branch or store instructions. This reduces unnecessary work and saves power.
4. **Shift Operation Optimization**
   Instead of a big, complex shifter, the design uses a small shift counter and executes shifts over multiple cycles. This trades a bit of speed for much less hardware use (TWO_LEVEL_SHIFTER).
5. **Immediate Value Decoding in Parallel**
   The different types of immediate values (constants embedded in instructions) are extracted all at once. This speeds up the instruction decoding step.
6. **Compact ALU Design**
   Most arithmetic and logic operations happen in one clock cycle with a simple combinational ALU, except shifts which take multiple cycles. This reduces delays and complexity.
7. **Efficient Register File Memory**
   The registers are stored in special FPGA memory blocks {inferred RAM} (not just logic gates), which saves space and improves speed.
8. **Simple Instruction Decoding**
   Instructions are classified by checking only a few key bits and ignoring the redundant ones (like bits 0 and 1 of the opcode). This makes decoding faster and uses less hardware.

3 • Classic Multicycle RISC-V Core Implementation:

3.1 Design Based on Patterson & Harris :

The design is grounded in the classic **RISC architecture principles** outlined by Patterson and Harris and the founding father John Hennessy which coinvented *the reduced instruction set computing* with Patterson. It emphasizes simplicity, efficiency, and clarity to achieve high performance with minimal hardware complexity

- Implements load-store architecture: only loads/stores access memory.



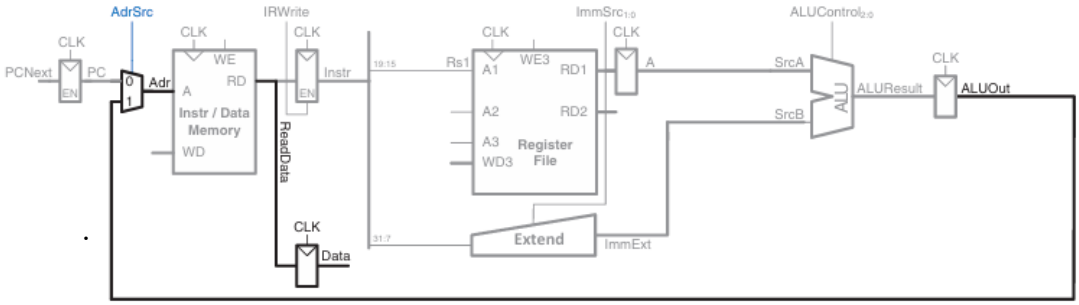


Figure 7.22 Load data from memory

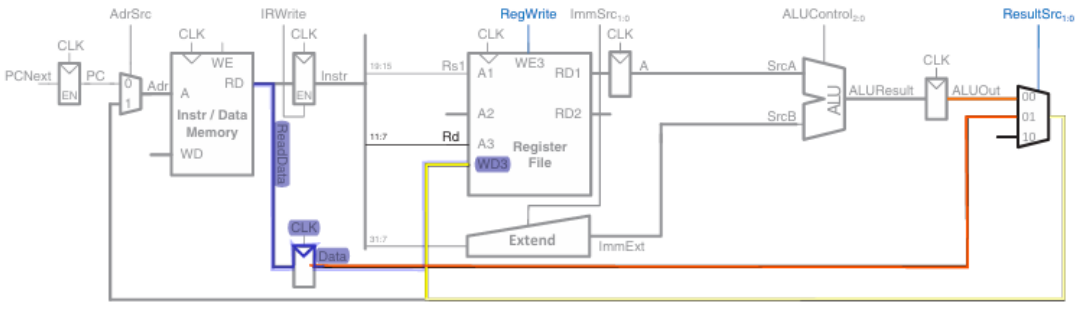


Figure10 : load-stores Implementation

- Executes most instructions in a single cycle.

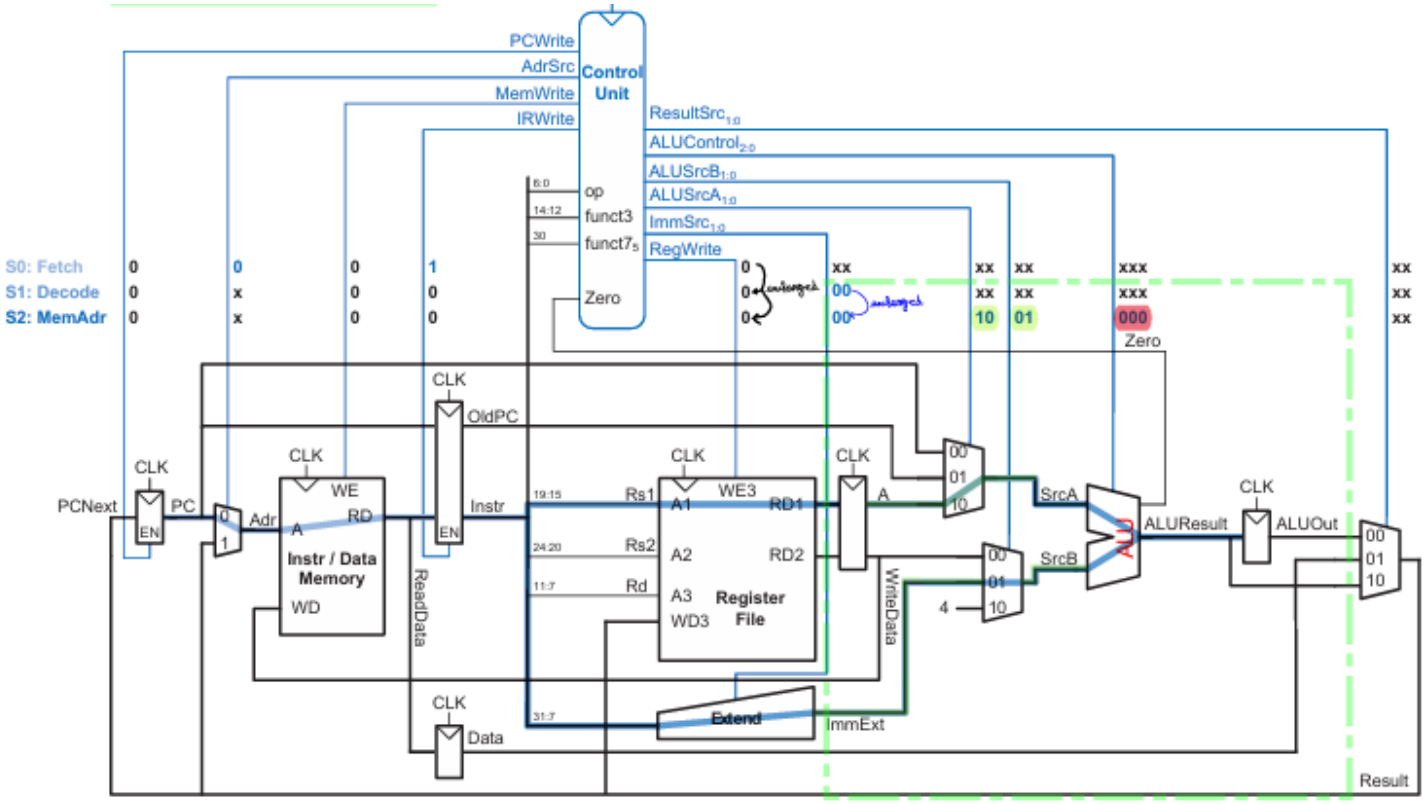


Figure11 : Single cycle stages

- Control logic uses a simple finite state machine.

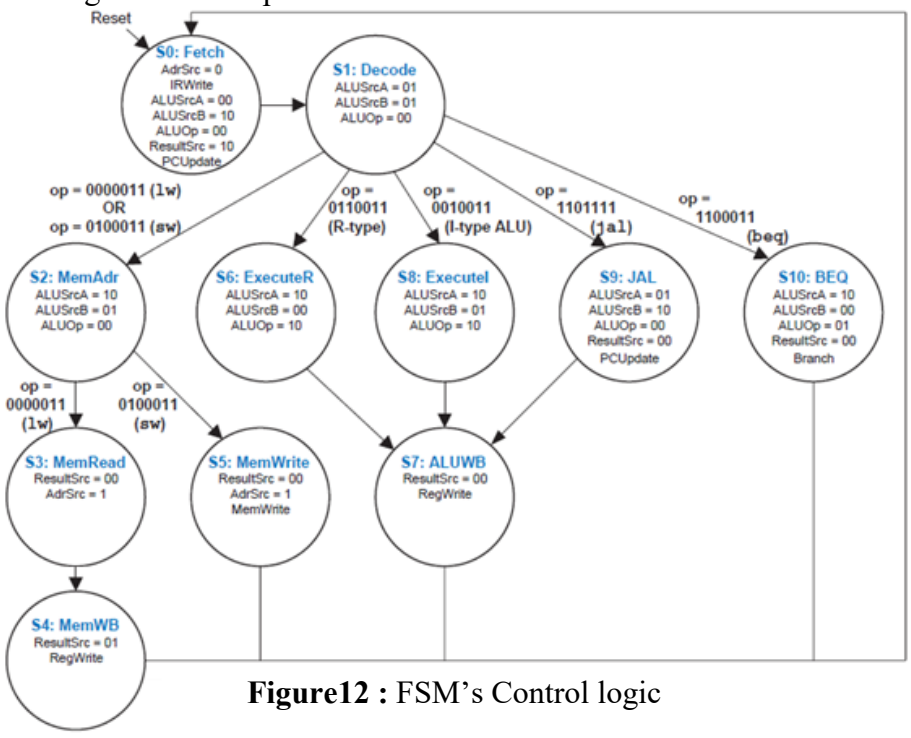


Figure12 : FSM's Control logic

- Separates data path and control path for clarity.

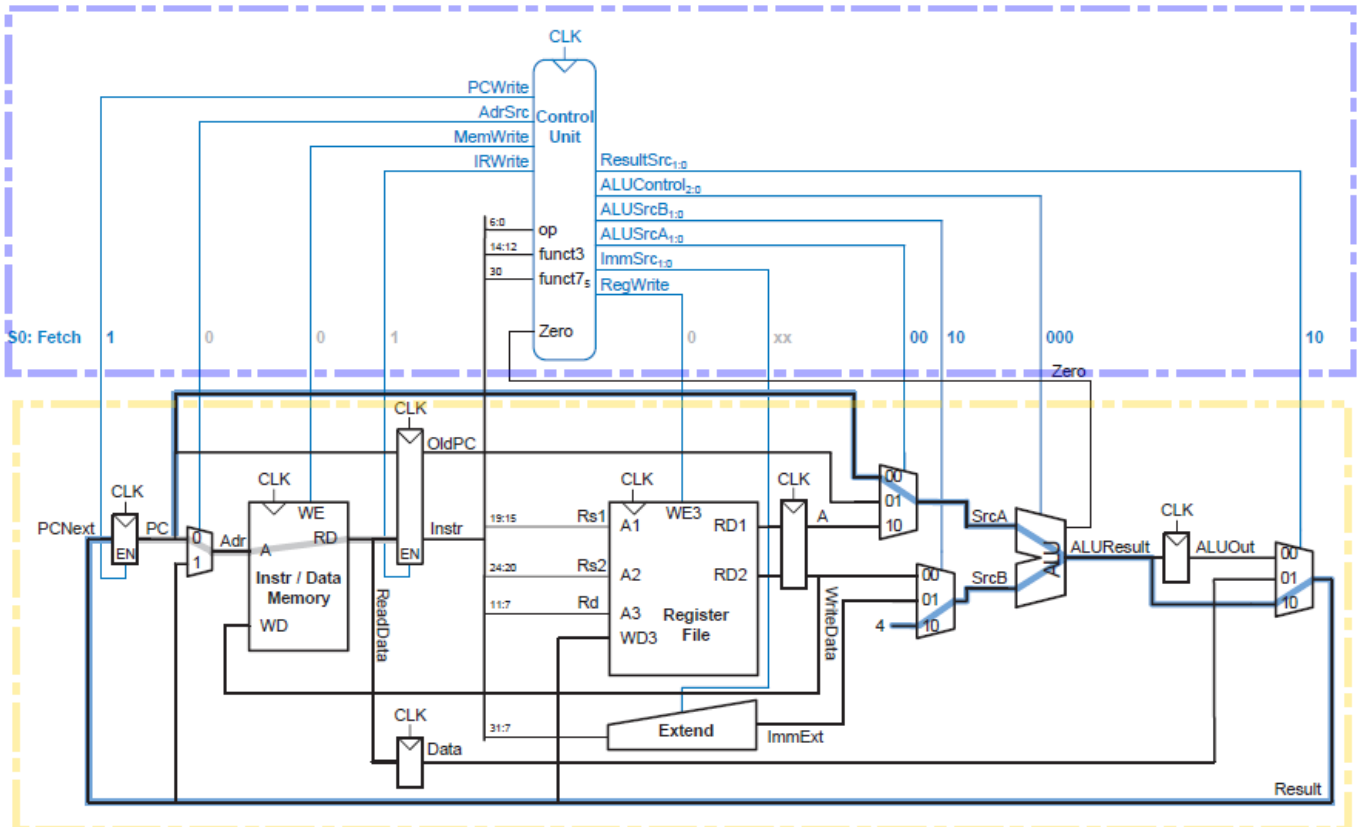


Figure13 : separation of data and control path

- Includes a register file with two read ports and one write port.

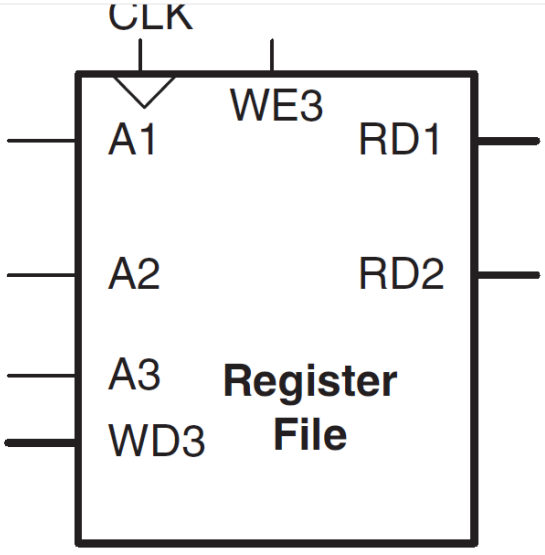


Figure14 : Register file

- Supports immediate values and conditional branches.

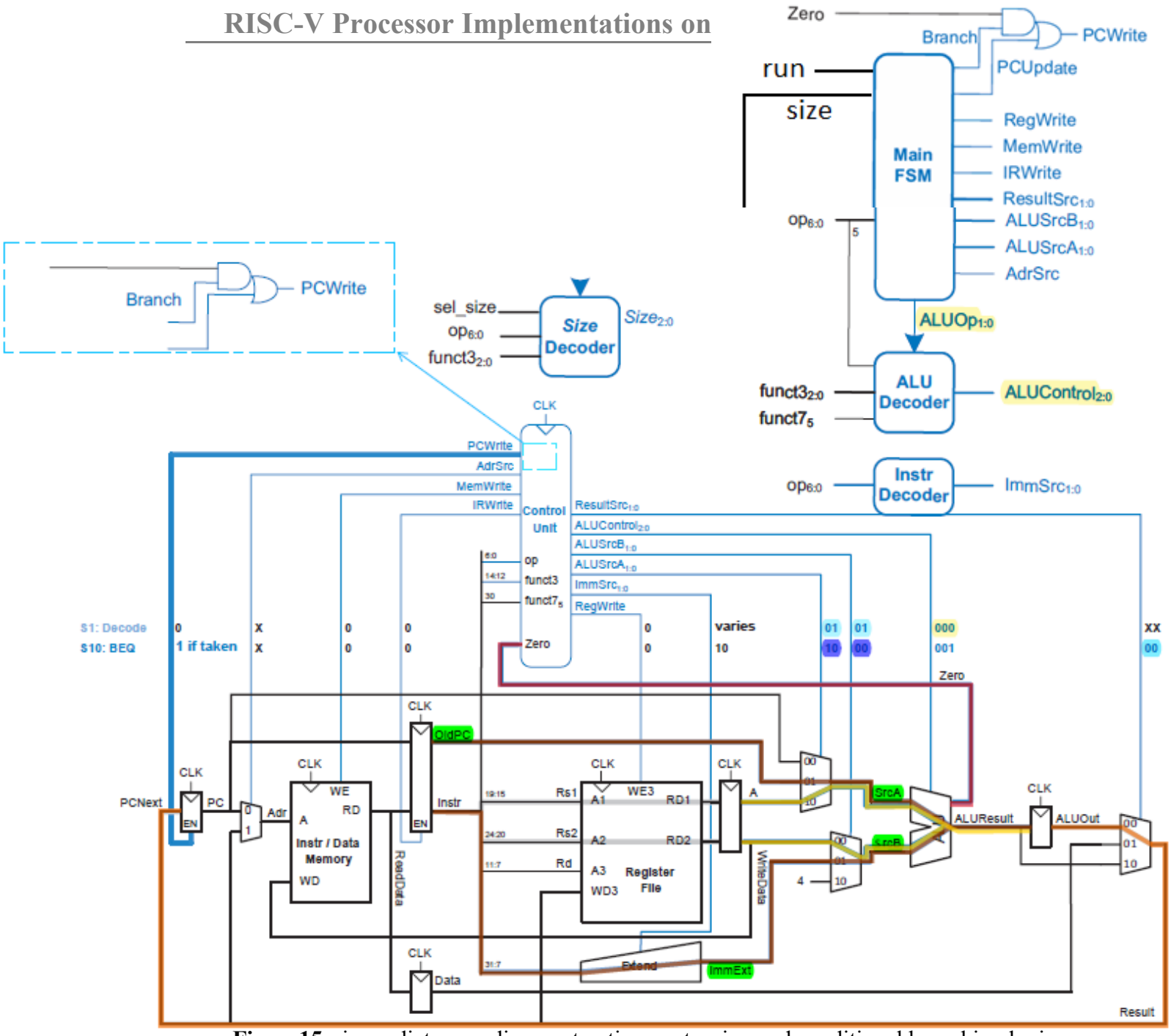


Figure15 : immediate encoding , extraction , extension and conditional branching logic

- it also interesting to mention other interesting guidelines and design decisions that were reiterated throughout the book and to keep in mind :

- **Execution Model**: The design uses a -*cycle datapath* to simplify control logic. This reduces timing complexity but may (**will**) limit maximum clock frequency.

- **Control Unit Architecture**: It employs a *hardwired finite state machine (FSM)* for embedded control signal generation, balancing control granularity and design complexity.

- **Data Path Components**: The architecture includes *key modules* such as the ALU, register file, instruction memory, and data memory, connected via multiplexers and buses, reflecting Patterson & Harris's **modular approach**.

- **Instruction Set Implementation**: The core supports a *subset of the RISC-V or MIPS instruction set*, focusing on arithmetic, memory access, and branching instructions for clarity and foundational coverage.

- **Timing and Hazard Management**: Without advanced pipeline stages, the design handles hazards implicitly by stalling or sequencing operations, simplifying hazard detection logic (**NOP**'s or **Software pipelining**).

- **Scalability and Extensibility**: The design facilitates straightforward extension with features like pipelining, forwarding, or caches due to its clear modular separation.

3.2 Architecture Overview and Datapath :

- The design of the RISC-V architecture was motivated by four principles articulated by David Patterson and John Hennessy in their text *Computer Organization and Design* :

(1) regularity supports simplicity;
(2) make the common case fast;
(3) smaller is faster;
(4) good design demands good compromises.

- The design is composed of several fundamental units that work in tandem to execute instructions:

{IF , ID,EXE,DM,MW}

- **Multiplexers and Internal Buses**: Provide flexible routing of data between the register file, ALU inputs, immediate value generator, and memory, allowing the datapath to select the correct operands and destinations dynamically.

- **Instruction Fetch Unit**: Responsible for reading instructions from memory using the program counter (PC), which updates each cycle. The PC either increments sequentially or changes based on branch outcomes.

- **Control Unit**: Interprets the fetched instruction and generates the necessary control signals to manage the datapath's behavior, including multiplexing, register writes, ALU operation selection, and memory access.

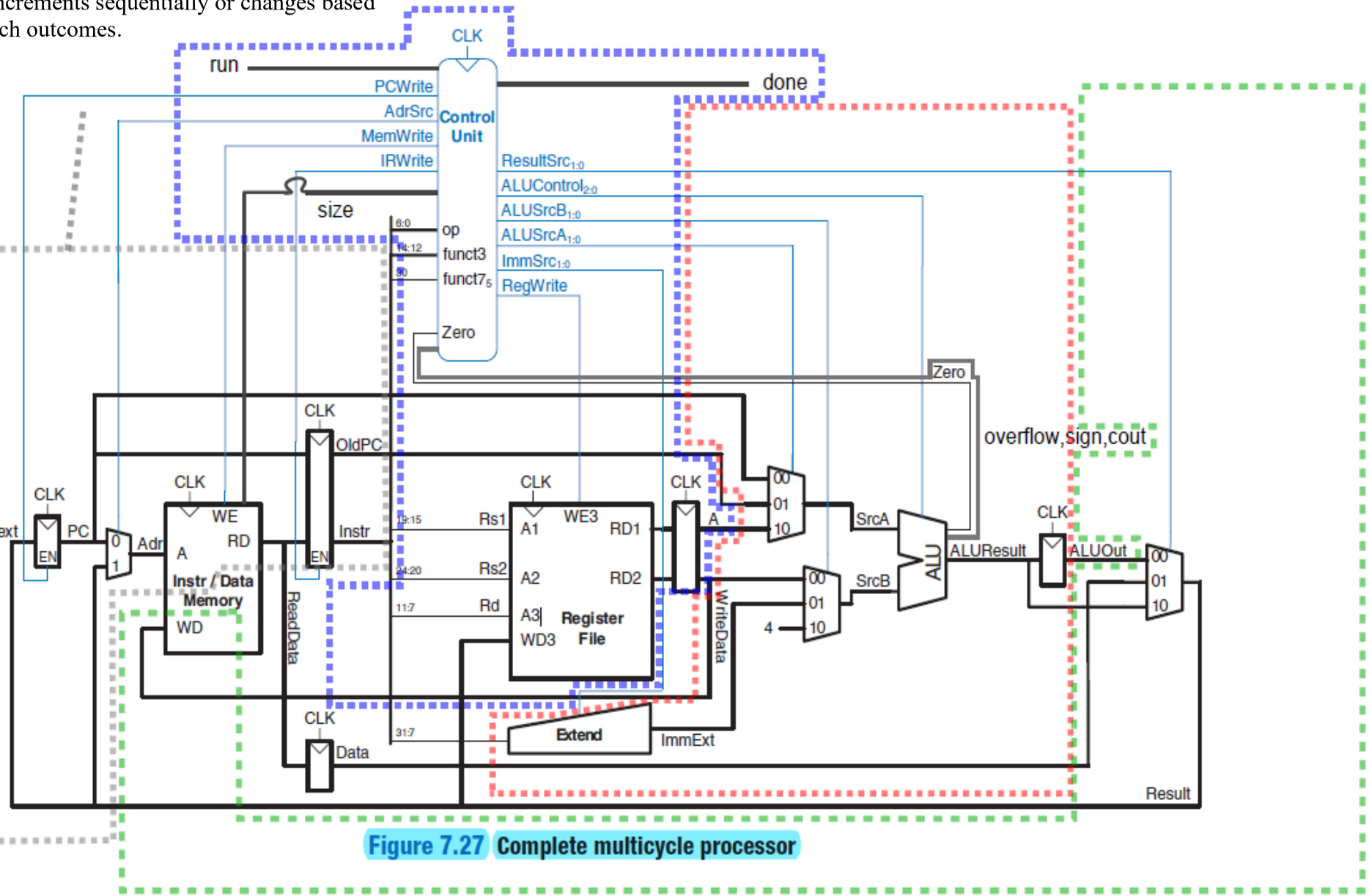


Figure 7.27 Complete multicycle processor

- **Data Memory Interface**: Facilitates reading from and writing to data memory, used in load and store instructions, tightly integrated with the datapath.

- **Register File**: Houses 32 registers accessible for two simultaneous reads and one write per cycle, providing operands for ALU operations and storing results.

- **Arithmetic Logic Unit (ALU)**: Executes all arithmetic and logical operations. It takes inputs from registers or immediate values extracted during instruction decode and produces computation results or branch condition flags.

Figure16 : the different components that make up the RISC-V datapath

The datapath flows through a sequence of stages: ***instruction fetch***, ***decode***, ***execute***, ***memory data (access)***, and ***write-back***. Control signals orchestrate these phases, either through the finite state machine and combinational logic.

This architecture balances straightforward design tailored for efficient data movement, therefore in accordance with any future enhancements like pipelining or cache integration.

3.3 Control Unit and FSM Design :

- The control unit in our multicycle RISC-V implementation is built as a **hardwired Finite State Machine (FSM)**, responsible for orchestrating control signals across multiple clock cycles per instruction. This approach reduces datapath complexity by allowing functional units (ALU, memory, registers) to be reused over several cycles.

- The FSM transitions through a fixed sequence of **micro-operations** depending on the instruction opcode.

• FSM States Breakdown:

1. **Idle State**:
   - Waits for the `run` signal to begin operation.
   - Transitions to `S0: Fetch` on activation.
2. **S0: Instruction Fetch**:
   - Controls: `AdrSrc = 0`, `IRWrite = 1`, `ALUSrcA = 0`, `ALUSrcB = 10`, `ALUOp = 00`, `ResultSrc = 0`, `PCUpdate = 1`.
   - Fetches instruction from memory, updates PC.
3. **S1: Decode / Register Fetch**:
   - Sets `ALUSrcA = 0`, `ALUSrcB = 11`, computes branch target or immediate values.
4. **S2 → S5: Load / Store Instructions (`lw`, `sw`)**:
   - `S2`: Address calculation (ALU computes `rs1 + offset`).
   - `S3`: Load → Memory read (`AdrSrc = 1`), transitions to `S4` for write-back.
   - `S5`: Store → Memory write (`MemWrite = 1`, `AdrSrc = 1`).
5. **S6, S8: ALU-type Instructions (`add`, `sub`, etc.)**:
   - `S6` (R-type), `S8` (I-type ALU) compute ALU result.
   - `S7`: ALU result written back to destination register (`RegWrite = 1`, `ResultSrc = 0`).
6. **S9: Jump (`jal`)**:
   - Updates PC (`PCUpdate = 1`) and writes return address to `rd`.
7. **S10: Branch (`beq`)**:
   - Evaluates condition, applies conditional PC update.

8. **LUI, AUIPC, Immediate CALC**:
    - LUI: writes upper immediate directly (`ResultSrc = 11`).
    - AUIPC: adds PC and offset (`ALUSrcA = 01`, `ALUOp = 00`).
    - Immediate Calc: used for custom control ops (`ALUSrcA = 10`, `ALUOp = 11`).

• Control Signal Patterns:

Each state asserts a precise set of control signals:

- **ALUSrcA, ALUSrcB**: Select inputs for ALU.
- **ALUOp**: Determines operation (add, sub, etc.).
- **AdrSrc**: Chooses address source for memory access.
- **IRWrite / RegWrite**: Enables writing to instruction or register files.
- **PCUpdate**: Enables PC write-back in jumps and branches.
- **ResultSrc**: Chooses data to write back (ALU, memory, PC, etc.).

• FSM Transitions:

The FSM transitions are determined by instruction **opcode decoding** at `S1`.

- For example:
    - `0000011` → Load (`lw`) → S2 → S3 → S4.
    - `0100011` → Store (`sw`) → S2 → S5.
    - `0110011` → R-type → S6 → S7.
    - `1101111` → `jal` → S9.
    - `1100011` → Branch → S10.

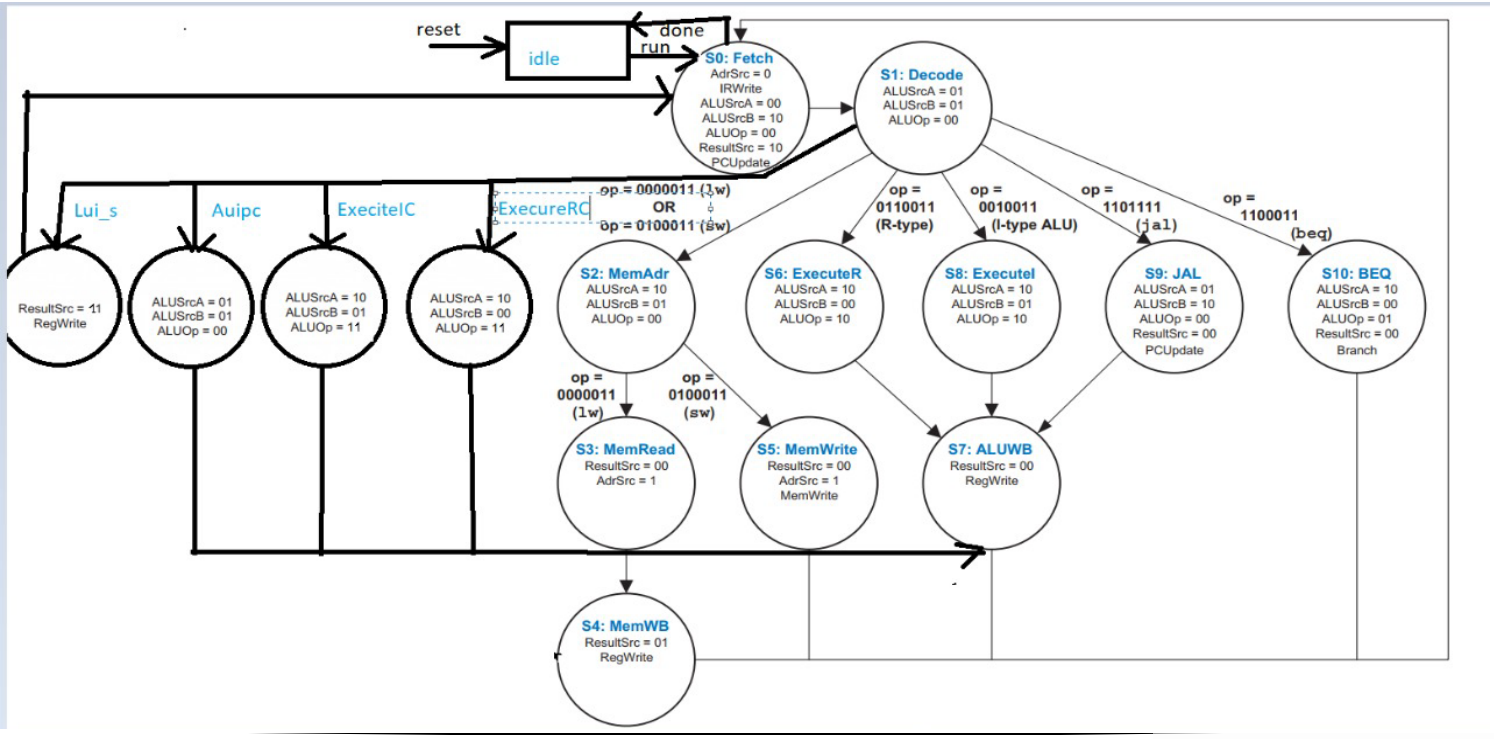


Figure17: Expanded FSM's Control Logic

• Advantages of This Design :

- **Resource-efficient**: Reuses ALU, memory, and registers across cycles.
- **Structured FSM**: Easy to debug and expand for new instructions.
- **Clear separation of control**: Datapath logic remains simple and generic.

3.4 Instruction Support and ISA Coverage :

The processor supports a focused subset of the **RISC-V RV32I base instruction set**, selected for compatibility with the multicycle control architecture and pedagogical clarity. Each instruction is mapped to a dedicated path through the FSM, ensuring accurate sequencing of control signals across states.

• Supported Instruction Classes :

1. **R-Type Instructions** (`add`, `sub`, `and`, `or`, `slt`, etc.)
   - **Opcode**: `0110011`
   - **FSM Path**:
     `S0: Fetch → S1: Decode → S6: ExecuteR → S7: ALUWB`
   - **ALU performs** operation using two register operands; result is written to `rd`.
2. **I-Type Arithmetic** (`addi`, `andi`, `ori`, etc.)
   - **Opcode**: `0010011`
   - **FSM Path**:
     `S0 → S1 → S8: ExecuteI → S7`
   - Uses immediate as second ALU operand.
3. **Load Word** (`lw`)
   - **Opcode**: `0000011`
   - **FSM Path**:
     `S0 → S1 → S2: MemAdr → S3: MemRead → S4: MemWB`
   - Performs effective address computation followed by memory read and register write-back.
4. **Store Word** (`sw`)
   - **Opcode**: `0100011`
   - **FSM Path**:
     `S0 → S1 → S2 → S5: MemWrite`
   - ALU calculates address, then memory is written from register.
5. **Branch Equal** (`beq`)
   - **Opcode**: `1100011`
   - **FSM Path**:
     `S0 → S1 → S10: BEQ`
   - Compares registers and conditionally updates PC if equal.
6. **Jump and Link** (`jal`)
   - **Opcode**: `1101111`
   - **FSM Path**:
     `S0 → S1 → S9: JAL`

- Saves PC+4 to `rd` and updates PC with immediate target.

7. **LUI (Load Upper Immediate)**
   - **Opcode**: `0110111`
   - **FSM Path**:
     `S0 → S1 → LUI_s`
   - Loads upper 20 bits of the immediate directly into `rd`.
8. **AUIPC (Add Upper Immediate to PC)**
   - **Opcode**: `0010111`
   - **FSM Path**:
     `S0 → S1 → AUIPC`
   - Adds upper immediate value to current PC.
9. **JALR (Jump and Link Register)** – *if implemented under `ExecuteIC` or `ExecuteRC`*
   - **Opcode**: `1100111`
   - FSM branch can be adapted or merged with `jal`.

• Instruction Execution Summary :

Each supported instruction:

- Passes through the **Fetch and Decode** stages (`S0` and `S1`).
- Then follows a unique microcoded control sequence through execution, memory, and write-back stages.
- Control signals (e.g., `RegWrite`, `ALUOp`, `ResultSrc`, `AdrSrc`) are asserted according to instruction semantics and datapath resource timing.

• ISA Coverage Remarks :

- The current implementation covers **load/store**, **arithmetic**, **control flow**, and **immediate** operations of RV32I.
- Compressed (C), multiplication (M), and system (S) extensions are excluded for simplicity.
- All supported instructions were validated against state transitions in the FSM to ensure complete micro-operation coverage and datapath synchronization.

op	funct3	funct7	Type	Instruction	Description	Operation
0000011 (3)	000	–	I	lb rd, imm(rs1)	load byte	rd = SignExt([Address]$_{7:0}$)
0000011 (3)	001	–	I	lh rd, imm(rs1)	load half	rd = SignExt([Address]$_{15:0}$)
0000011 (3)	010	–	I	lw rd, imm(rs1)	load word	rd = [Address]$_{31:0}$
0000011 (3)	100	–	I	lbu rd, imm(rs1)	load byte unsigned	rd = ZeroExt([Address]$_{7:0}$)
0000011 (3)	101	–	I	lhu rd, imm(rs1)	load half unsigned	rd = ZeroExt([Address]$_{15:0}$)
0010011 (19)	000	–	I	addi rd, rs1, imm	add immediate	rd = rs1 + SignExt(imm)
0010011 (19)	001	0000000*	I	slli rd, rs1, uimm	shift left logical immediate	rd = rs1 << uimm
0010011 (19)	010	–	I	slti rd, rs1, imm	set less than immediate	rd = (rs1 < SignExt(imm))
0010011 (19)	011	–	I	sltiu rd, rs1, imm	set less than imm. unsigned	rd = (rs1 < SignExt(imm))
0010011 (19)	100	–	I	xori rd, rs1, imm	xor immediate	rd = rs1 ^ SignExt(imm)
0010011 (19)	101	0000000*	I	srli rd, rs1, uimm	shift right logical immediate	rd = rs1 >> uimm
0010011 (19)	101	0100000*	I	srai rd, rs1, uimm	shift right arithmetic imm.	rd = rs1 >>> uimm
0010011 (19)	110	–	I	ori rd, rs1, imm	or immediate	rd = rs1 | SignExt(imm)
0010011 (19)	111	–	I	andi rd, rs1, imm	and immediate	rd = rs1 & SignExt(imm)
0010111 (23)	–	–	U	auipc rd, upimm	add upper immediate to PC	rd = {upimm, 12'b0} + PC
0100011 (35)	000	–	S	sb rs2, imm(rs1)	store byte	[Address]$_{7:0}$ = rs2$_{7:0}$
0100011 (35)	001	–	S	sh rs2, imm(rs1)	store half	[Address]$_{15:0}$ = rs2$_{15:0}$
0100011 (35)	010	–	S	sw rs2, imm(rs1)	store word	[Address]$_{31:0}$ = rs2
0110011 (51)	000	0000000	R	add rd, rs1, rs2	add	rd = rs1 + rs2
0110011 (51)	000	0100000	R	sub rd, rs1, rs2	sub	rd = rs1 − rs2
0110011 (51)	001	0000000	R	sll rd, rs1, rs2	shift left logical	rd = rs1 << rs2$_{4:0}$
0110011 (51)	010	0000000	R	slt rd, rs1, rs2	set less than	rd = (rs1 < rs2)
0110011 (51)	011	0000000	R	sltu rd, rs1, rs2	set less than unsigned	rd = (rs1 < rs2)
0110011 (51)	100	0000000	R	xor rd, rs1, rs2	xor	rd = rs1 ^ rs2
0110011 (51)	101	0000000	R	srl rd, rs1, rs2	shift right logical	rd = rs1 >> rs2$_{4:0}$
0110011 (51)	101	0100000	R	sra rd, rs1, rs2	shift right arithmetic	rd = rs1 >>> rs2$_{4:0}$
0110011 (51)	110	0000000	R	or rd, rs1, rs2	or	rd = rs1 | rs2
0110011 (51)	111	0000000	R	and rd, rs1, rs2	and	rd = rs1 & rs2
0110111 (55)	–	–	U	lui rd, upimm	load upper immediate	rd = {upimm, 12'b0}
1100011 (99)	000	–	B	beq rs1, rs2, label	branch if =	if (rs1 == rs2) PC = BTA
1100011 (99)	001	–	B	bne rs1, rs2, label	branch if ≠	if (rs1 ≠ rs2) PC = BTA
1100011 (99)	100	–	B	blt rs1, rs2, label	branch if <	if (rs1 < rs2) PC = BTA
1100011 (99)	101	–	B	bge rs1, rs2, label	branch if ≥	if (rs1 ≥ rs2) PC = BTA
1100011 (99)	110	–	B	bltu rs1, rs2, label	branch if < unsigned	if (rs1 < rs2) PC = BTA
1100011 (99)	111	–	B	bgeu rs1, rs2, label	branch if ≥ unsigned	if (rs1 ≥ rs2) PC = BTA
1100111 (103)	000	–	I	jalr rd, rs1, imm	jump and link register	PC = rs1 + SignExt(imm), rd = PC + 4
1101111 (111)	–	–	J	jal rd, label	jump and link	PC = JTA, rd = PC + 4

Figure18: RV32I: RISC-V integer instructions

3.5 Simulation and Verification :

- To validate the correctness and timing of the multicycle processor implementation, a structured simulation and verification process was conducted. This process focused on confirming functional behavior, instruction decoding, control signal sequencing, and memory interactions across various instruction types.

▸Simulation Methodology :

- **Testbench Structure**:
  A comprehensive Verilog testbench was developed to instantiate the processor, initialize memory contents, generate a reset pulse, and supply a `run` signal.
- **Clock and Reset**:
  A controlled clock source and an initial reset pulse were applied to ensure deterministic startup behavior and FSM initialization to the `idle` state.
- **Memory Initialization**:
  Instruction and data memories were pre-loaded with test programs written in RISC-V assembly, assembled into `.hex` format and parsed into the simulation environment.
- **Instruction Coverage**:
  The test programs included representative examples from each supported instruction class:
  - Arithmetic: `add`, `sub`, `addi`
  - Memory: `lw`, `sw`
  - Control flow: `beq`, `jal`, `jalr`
  - Immediate: `lui`, `auipc`

▸ Verification Objectives :

1. **FSM Transitions**
   - Observed step-by-step state changes from `S0` (Fetch) to terminal states like `ALUWB`, `MemWB`, or control-specific states.
   - Confirmed that each instruction follows the correct FSM path as defined in the control diagram.
2. **Control Signal Assertion**
   - Validated that all control signals (`ALUSrcA`, `ALUSrcB`, `RegWrite`, `MemWrite`, `ResultSrc`, etc.) are asserted appropriately per instruction type and cycle.
   - Special attention was given to memory access cycles (`AdrSrc`, `MemWrite`) and PC updates.
3. **Datapath Integrity**
   - Monitored register file writes (`rd`), ALU output correctness, memory read/write actions, and PC evolution.
   - Ensured hazard-free operation through strict multicycle sequencing.
4. **End-to-End Test Results**
   - Compared final register file and memory states against expected values using `assert` or forcing statements in simulation.
   - No control or data mismatches were observed across all scenarios.

▸ Simulation Tools :

- **Tool Used**: ModelSim / QuestaSim

- **Waveform Analysis**:
  Signals were traced and analyzed via VCD dump or waveform viewer to inspect the timing of control logic and datapath operations over cycles.

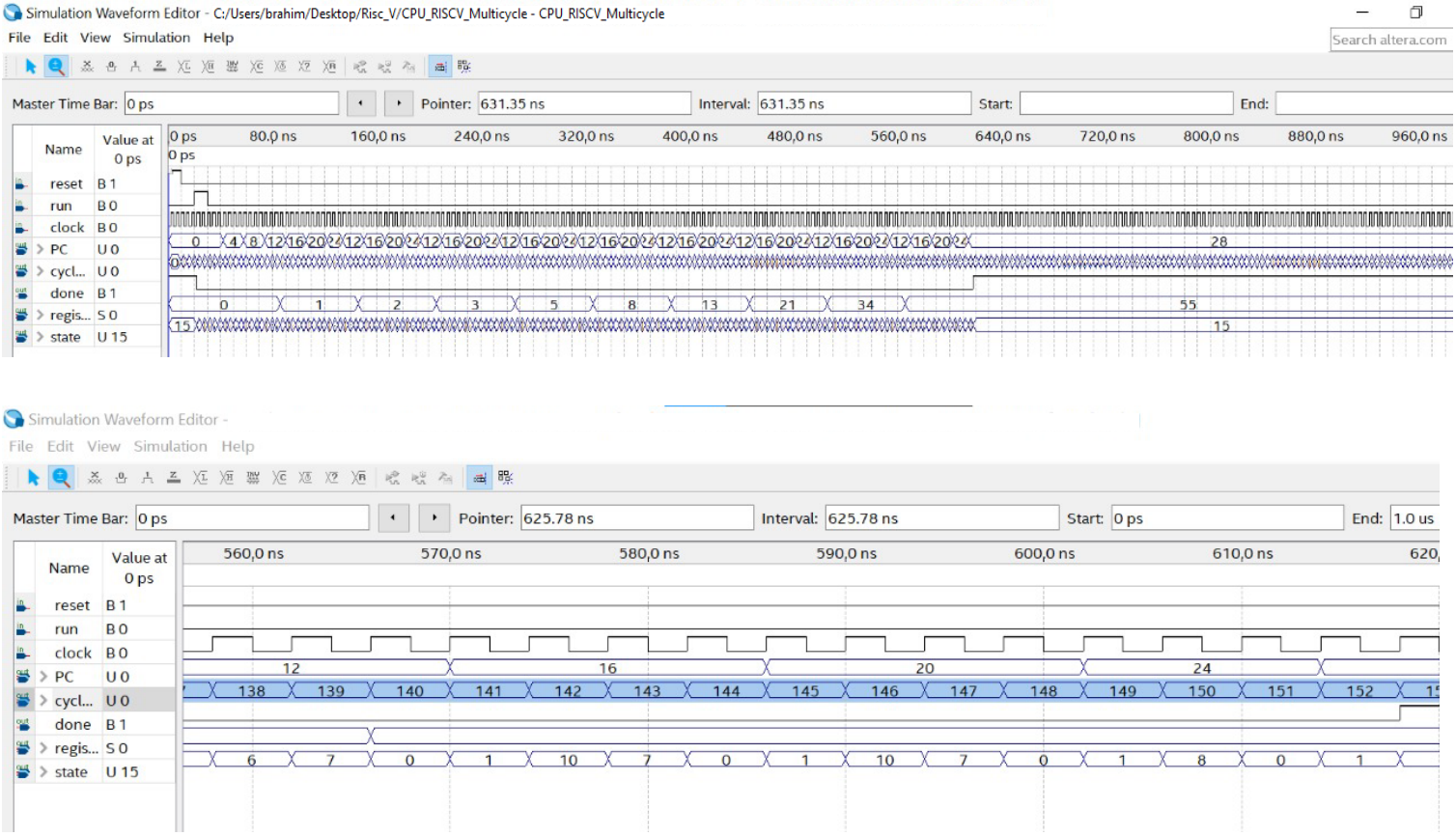


Figure19: Classic multicycle simulation (Fibonacci[10] = 55)
without implementing recursions

3.6 FPGA Synthesis (Timing, Area, Synthesis Reports) :

-The simple multicycle processor was synthesized on an **Intel Cyclone II FPGA** using ***Quartus***. The design is written in raw ***Verilog*** without FPGA-specific hints /Compiler directives like `ramstyle` attributes or one-hot encoding. The

- **FSM** uses basic binary state encoding and both **the memory** and **register file** are built from ***regular logic***, not mapped to an onboard ***RAM block***.

-***Synthesis reports*** show that the design fits comfortably within the **Cyclone II**'s logic resources, using around **25,00 logic elements**. **The ALU**, **control unit**, and **register file** account for *the majority of the area*.

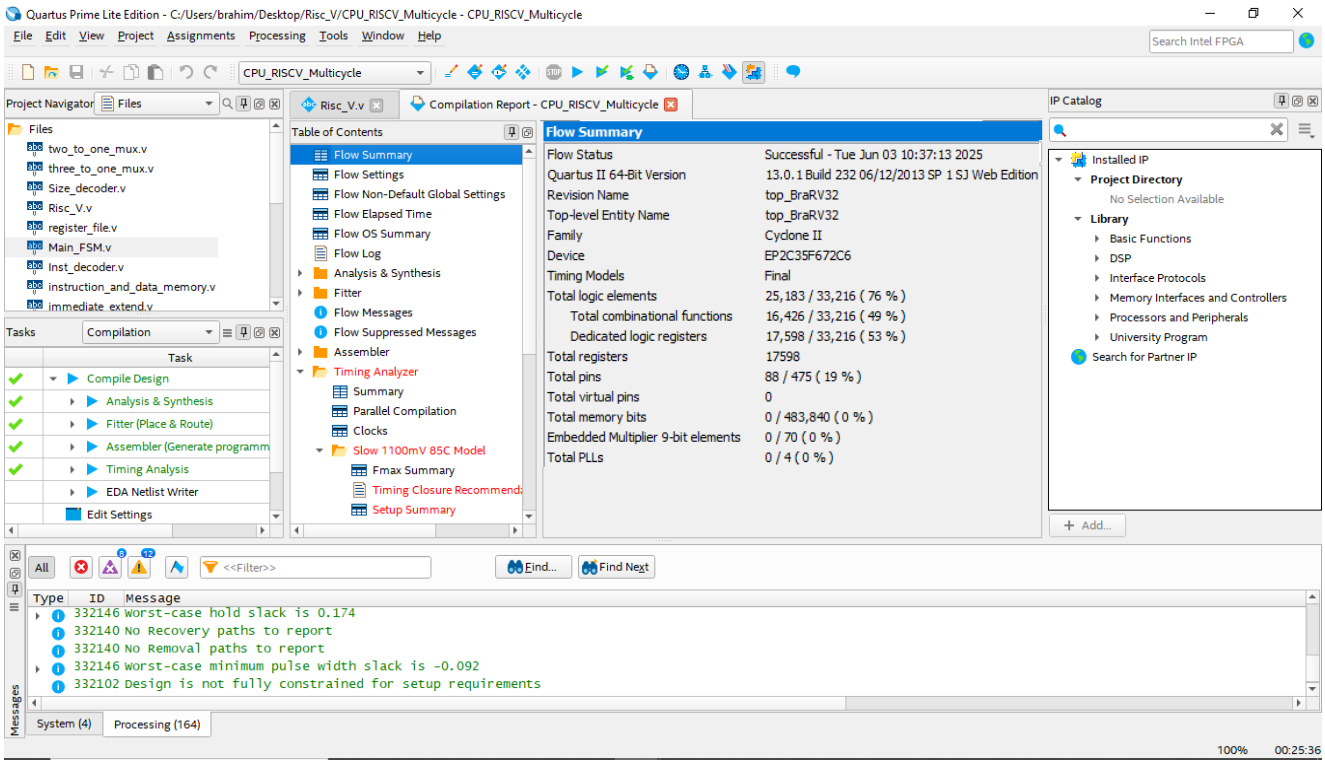
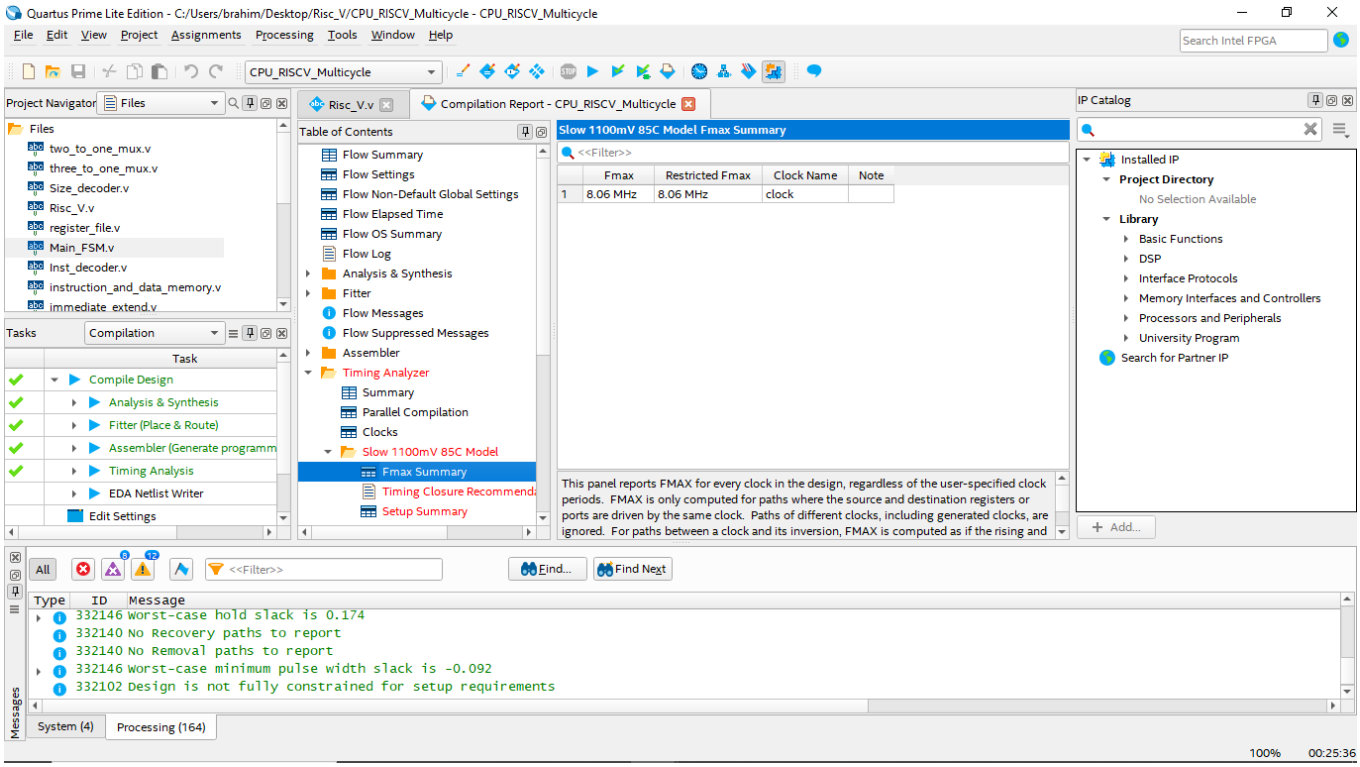


Figure20: Compilation report's Flow summary for the Classic Multicycle Implementation

- ***Timing analysis*** shows stable operation up to **~8 MHz**, with no critical path violations.



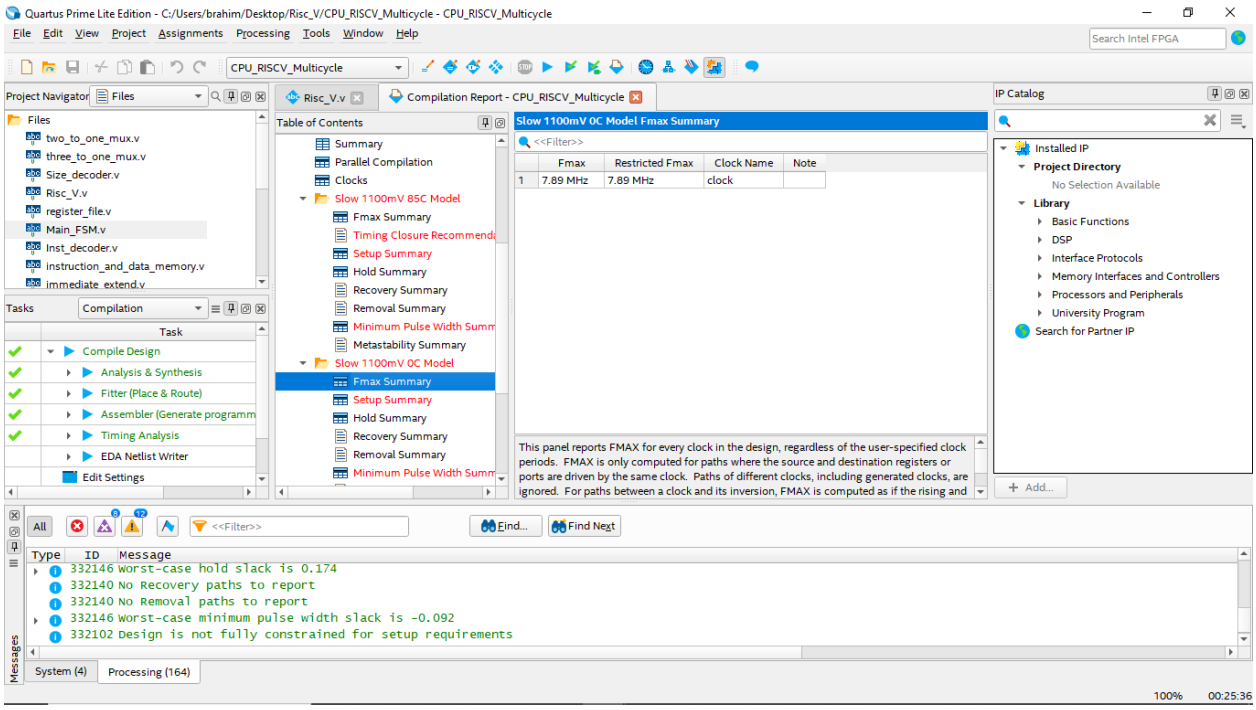


Figure21: Compilation report's Timing Analyzer maximum frequency F_{max}

-***The RTL Viewer*** confirms a *clean* and *modular* **datapath**: **instruction memory**, **control unit**, **ALU**, **register file**, and **program counter** are clearly separated. **The FSM** transitions are *fully visible* and follow *the expected multicycle flow*.

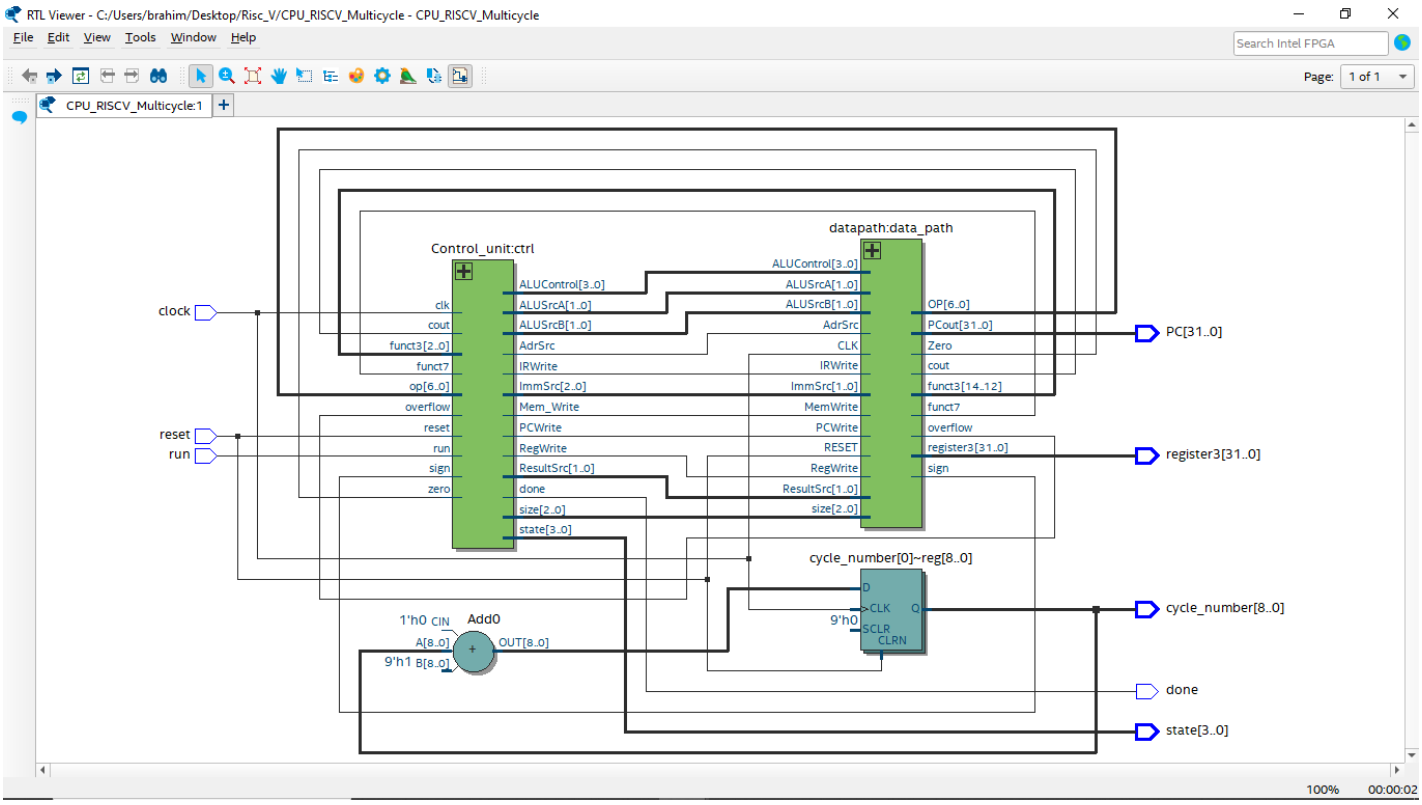


Figure22: RTL View of the whole multicycle design

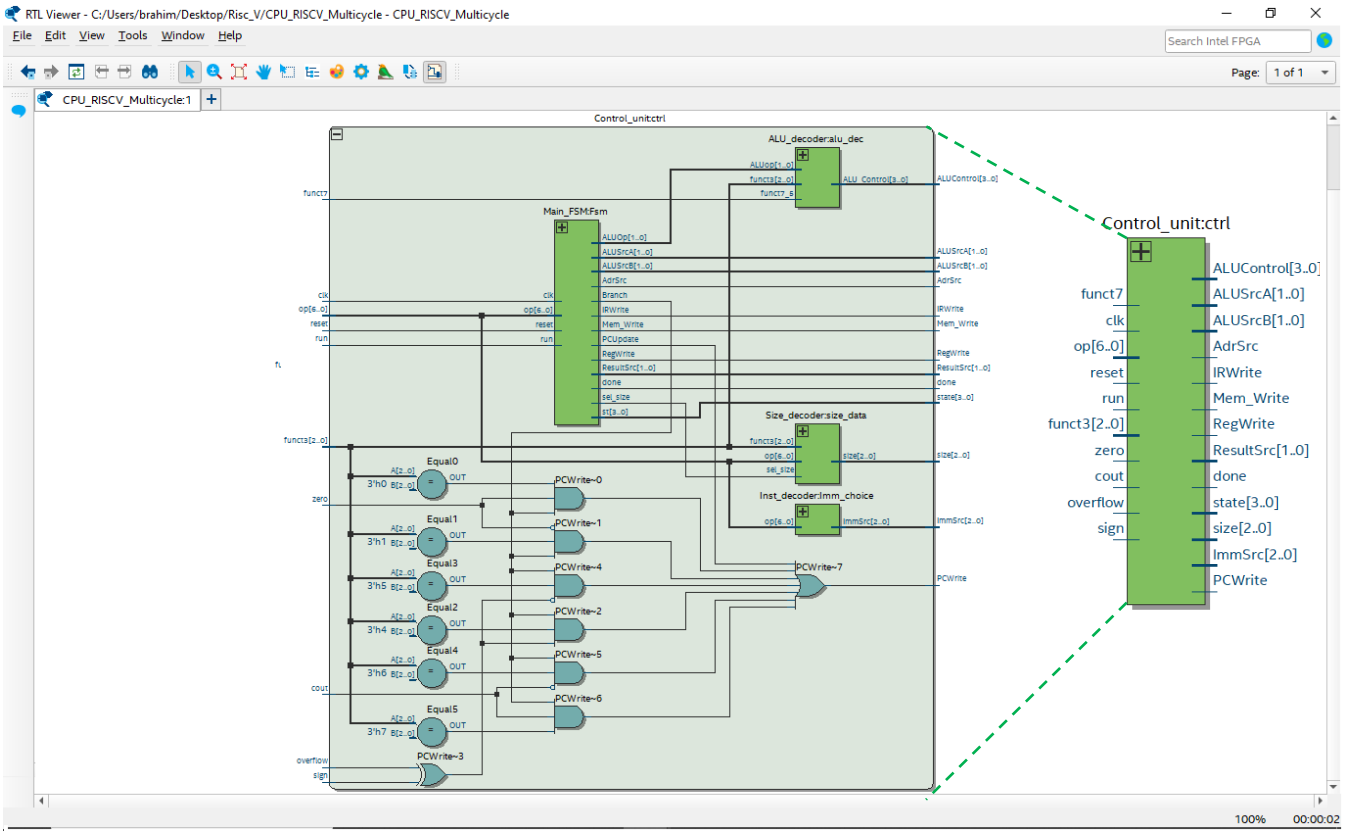


Figure23: RTL View of the control unit

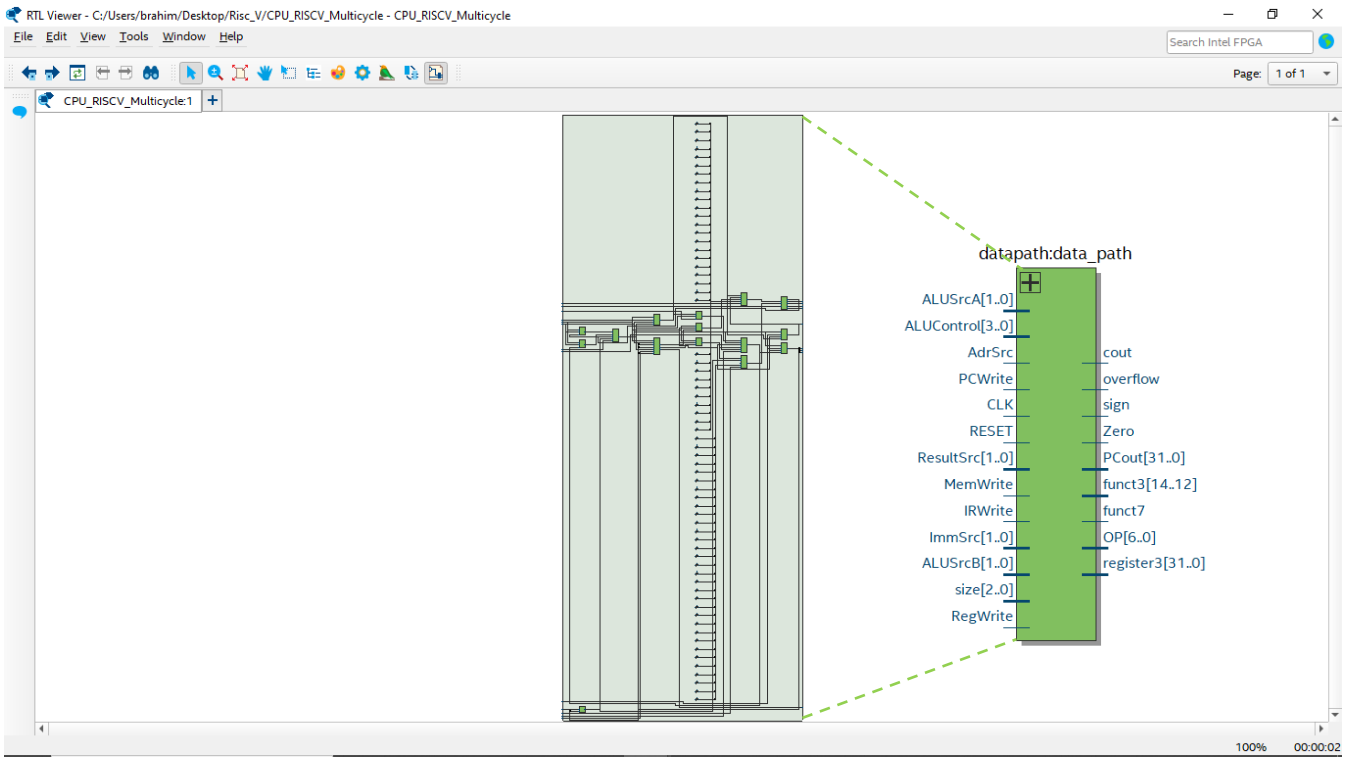


Figure24: RTL View of the datapath

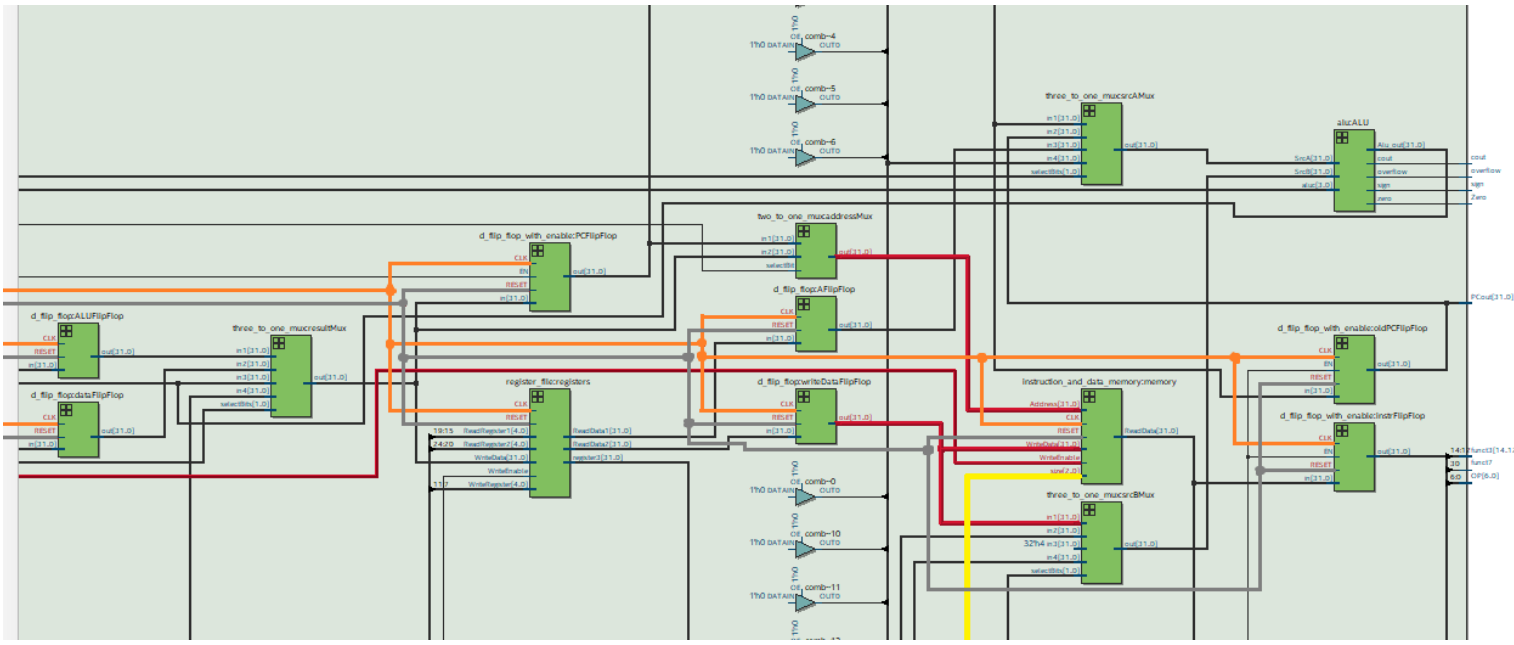


Figure25: Datapath's components

-Technology mapping results also show minimal fan-out and no inferred latches or unintended combinational loops.

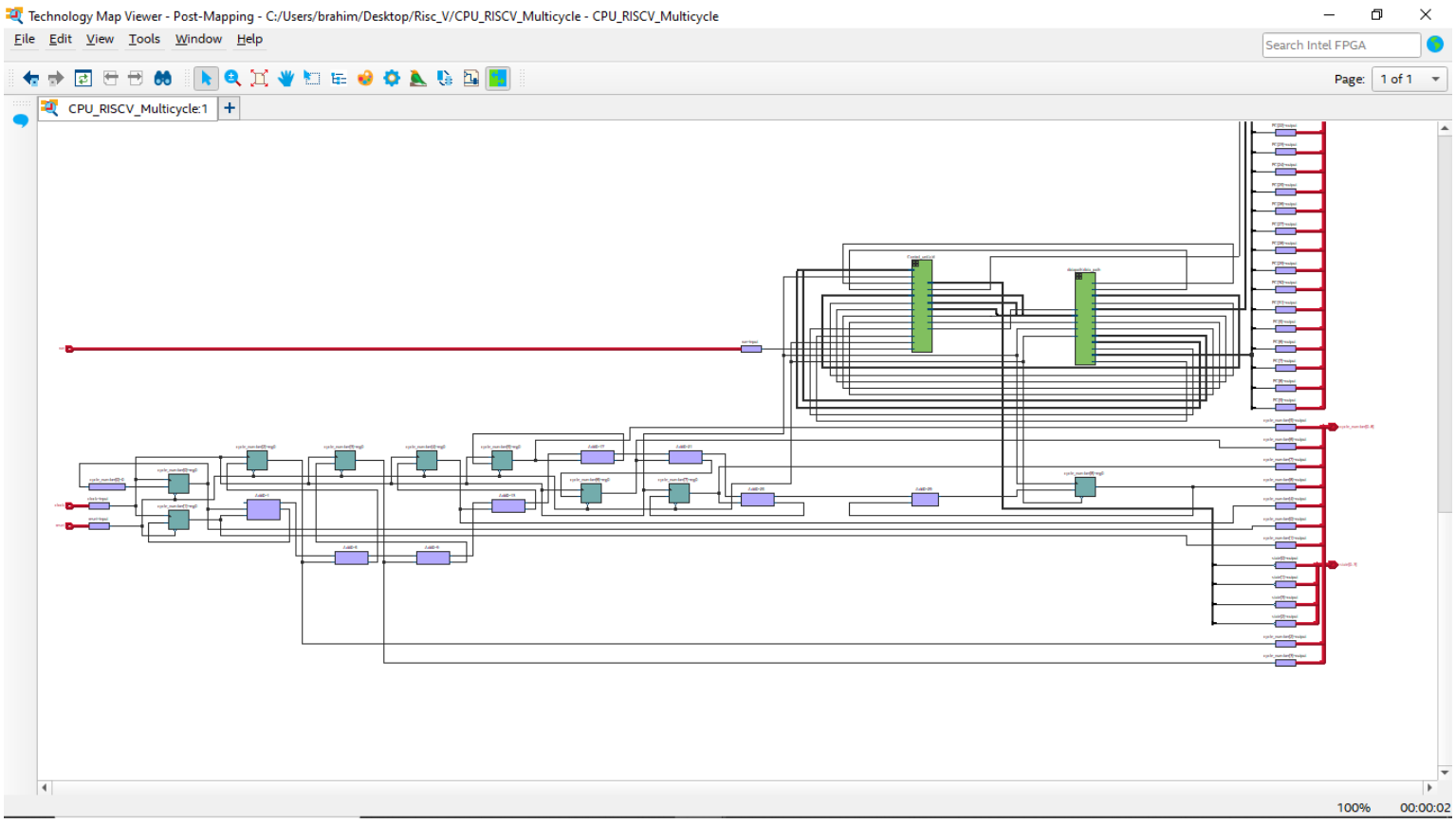


Figure26: Technology Map View of the design

- The Chip Planner view confirmed balanced resource distribution across the Cyclone II fabric with no hot spots or routing congestion, For clarity **chip Planner** is a graphical tool in ***Quartus*** used to visualize the physical placement and routing of logic elements on the ***FPGA fabric***.

- For this implementation, it helped confirm that the processor's core components **FSM**, **ALU**, and **register file** were evenly distributed with no routing congestion. This supported clean synthesis and timing closure without manual floorplanning.

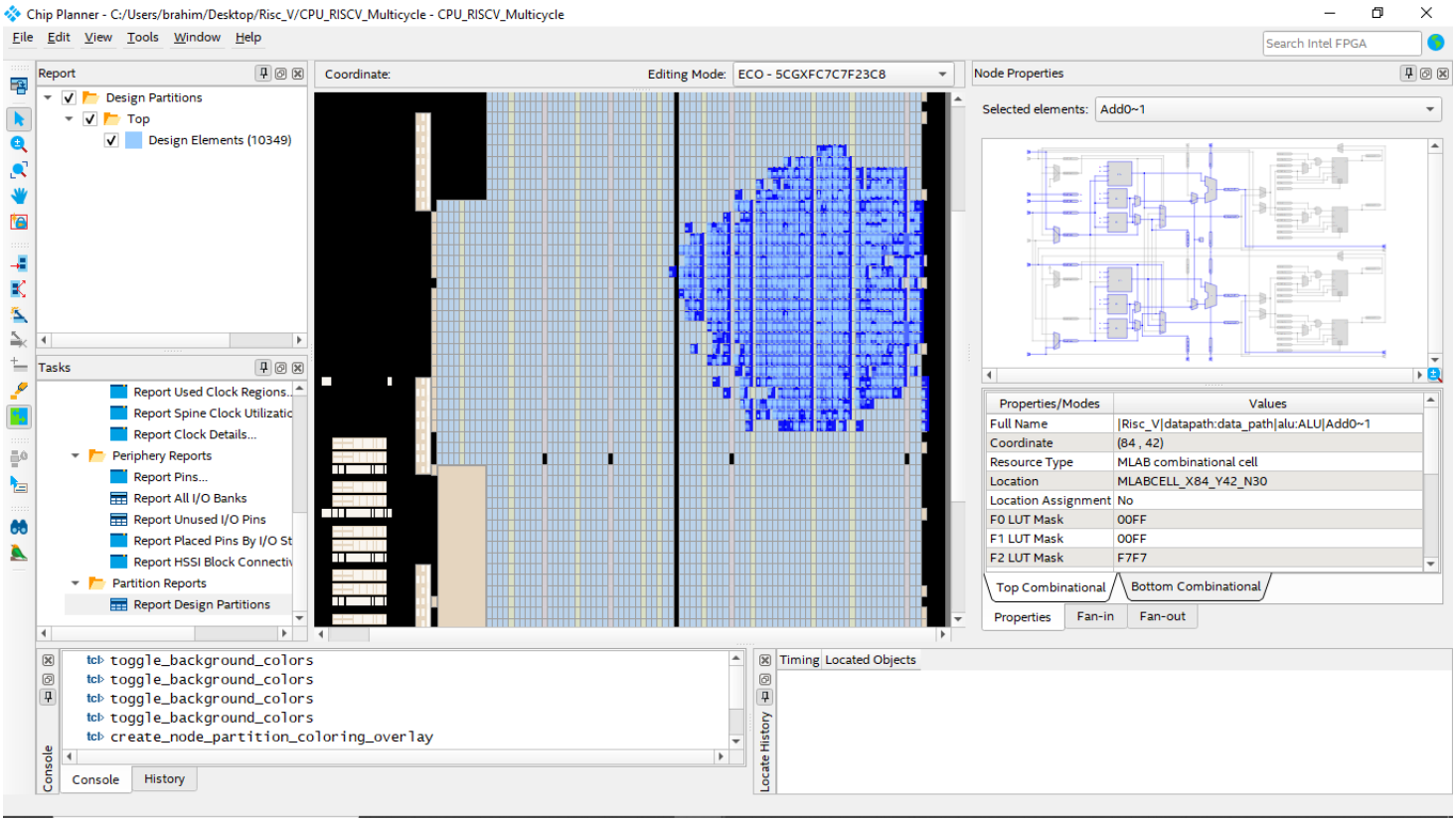


Figure27: Chip Planner View of the design (for a Cyclone V FPGA)

-Simulation results in Quartus Waveform Simulator matched expected behavior across the basic R-type, I-type, and memory operations, used for the Fibonacci sequence program ensuring functional correctness before deployment (Figure19).

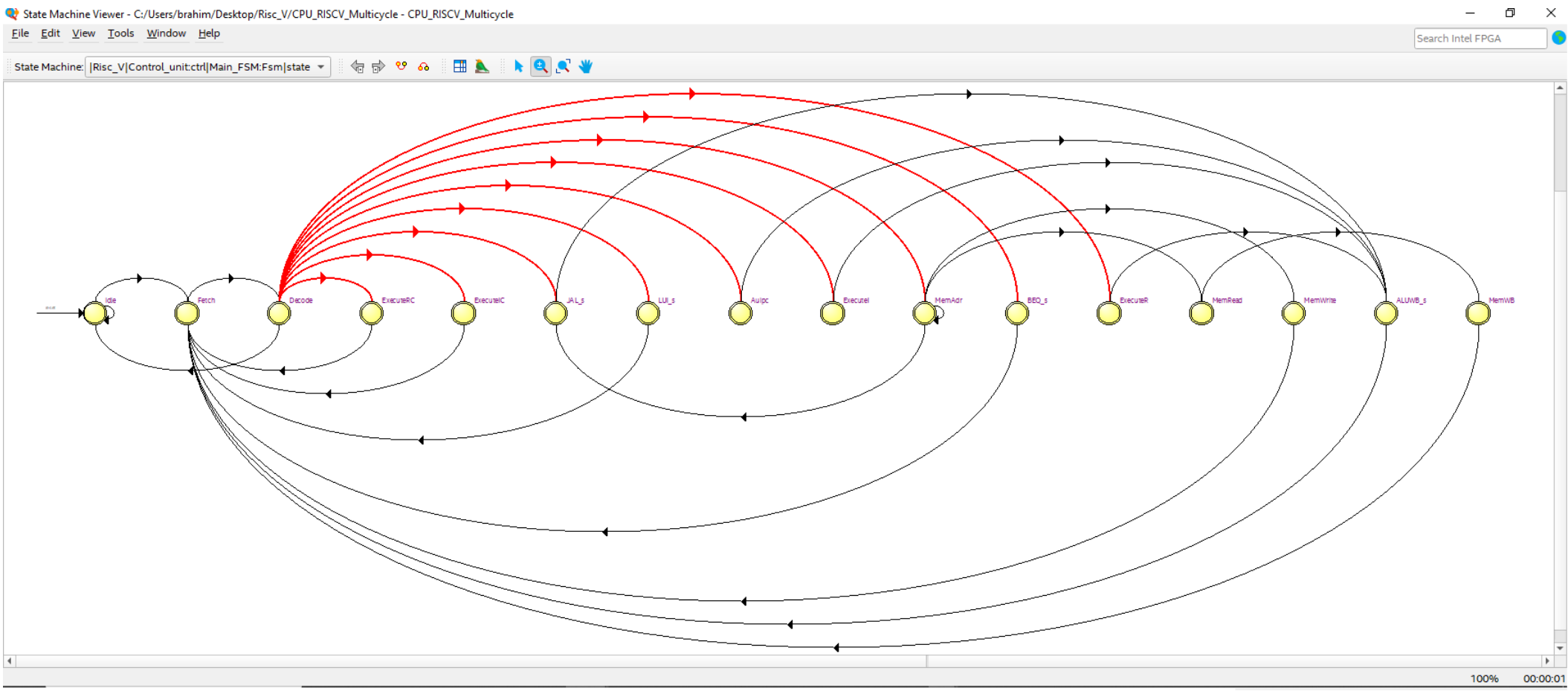


Figure28: State Machine Viewer

- Overall, the FPGA results validated that **the multicycle architecture** can be implemented with low resource usage, clean timing, and without the need for pipelining or complex forwarding logic.

3.7 Limitations and Bottlenecks :

- This simple implementation targets clarity and minimal logic but comes with technical trade-offs. Due to the use of basic Verilog without synthesis hints and other resource optimizing techniques and best practices , thus several constraints emerge :

- The design **lacks hazard detection**, making it unsuitable for complex instruction sequences or overlapping operations.
- Memory access uses **fixed-latency assumptions**, without accounting for real-world stalls or burst behavior.
- Instruction coverage is minimal, **omitting *CSR***, ***system***, ***floating***, and ***interrupt instructions***, limiting software compatibility.
- **No interrupt** support or **exception handling** is included, further restricting embedded use cases.
- The program counter update is static and sequential, with **no form of branch prediction** or **speculative execution**.
- Both **instructions** and **data *share*** the **same memory block**, which can cause conflicts and hinders extensibility.
- Finally, the register file is built from **flip-flops** rather than **dedicated memory blocks**, ***reducing area efficiency*** and ***scalability***.

- These limitations are acceptable for this proof-of-concept implementation but highlights the need for deeper architectural enhancements in any future work we might embark on.

4 • Optimized Multicycle Core Design:

4.1 Motivation for Optimization :

BraRV32 The optimized multicycle core was designed to overcome the structural and performance constraints of the initial basic implementation. While both versions use raw Verilog targeting the Cyclone II FPGA family, *BraRV32* integrates both design & HDL enhancements such as **one-hot FSM encoding** and *synthesis hints*/*compilation directives* such `ramstyle = "M10K"...`

- such directives are used to better leverage FPGA-specific resources.

- This version focuses on **minimizing LUT usage**, improving **clock-to-output timing**, and **reducing combinational logic depth**. Instruction **decoding is refined** using *bit-pattern classification*, and ALU operations are **streamlined** (

having been made simpler and more efficient or effective)through shared **arithmetic logic**. The FSM is **fully redesigned** to ***reduce control path delays*** and to ***better isolate datapath transitions***.

- Moreover, ***partial decode logic*** and ***selective writeback*** **reduce unnecessary switching activity**, contributing to power and area efficiency. These refinements make the design more scalable, timing-optimized, and suitable for constrained FPGA environments (such as the DE2-board).

4.2 Architectural Revisions :

- The *BraRV32* architecture retains the core structure of a multicycle RISC-V processor but introduces several targeted changes to improve synthesis and execution efficiency on our Cyclone II FPGA.

- The datapath is revised to **separate combinational** and **sequential elements** *more cleanly*. The **ALU is minimized** using ***a single shared subtractor*** for ***multiple operations*** (*subtraction, comparison, equality*), and a **counter-based shifting mechanism** replaces *the need for a barrel shifter*, trading area ⟷ for cycles. ***Immediate extraction*** is **handled in parallel** to *reduce decode latency*.

- The FSM is ***restructured*** with **one-hot encoding**, allowing for ***faster***, ***parallel*** *state resolution* in the control unit, further more `(*parallel_case*)` directive is just used before to hint to the compiler that the latter operates in parallel consequently **omitting** any ***unnecessary case checking*** and ***extra decision-logic***.

- Register file access is mapped to onboard M10K blocks via `ramstyle` hints, **conserving LUTs** and **improving memory access times**. Memory and control signals are simplified and made more deterministic, improving timing closure and reducing critical paths.

→ To demonstrate the proposed optimization solutions and revise the architecture in parallel let's take the steps , states , operations and the dataflow necessary for a RISC-V core to implement a `load` instruction .

```
assign isLoad    =  (instr[6:2] == 5'b00000); // rd <- mem[rs1+Iimm]
```

❶ **Read the instruction from memory** *at the* **address in the PC**.

- `AdrSrc = 0` → Address comes from PC.
- `MemRead = 1`, `IRWrite = 1` → Write fetched instruction into `Instr` register.
- `PCWrite = 1` → Enable PC update.
- ALU performs `PC + 4` to compute next PC → (During the fetch phase, the ALU is free, so it is reused to compute PC + 4)

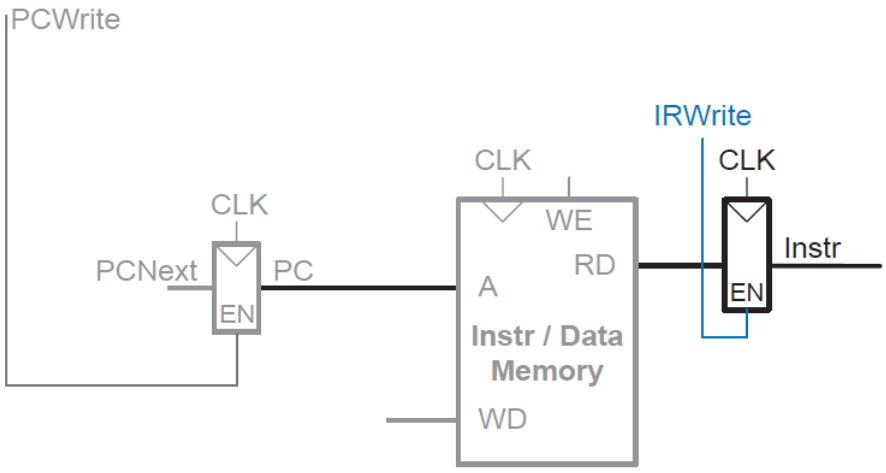


Figure29: instruction fetch (IF) of the `load` instruction

- how it's implemented for this core :

BraRV32

```
assign mem_addr = state[WAIT_INSTR_bit] | state[FETCH_INSTR_bit] ?
             PC : loadstore_addr ;
------------------------------------------------------------------------
assign PCplus4 = PC + 4;
assign loadstore_addr = rs1[ADDR_WIDTH-1:0] +
       (instr[5] ? Simm[ADDR_WIDTH-1:0] : Iimm[ADDR_WIDTH-1:0]);

------------------------------------------------------------------------
state[EXECUTE_bit]: begin
           PC <= isJALR          ? {aluPlus[ADDR_WIDTH-1:1],1'b0}:
/*instructions assumed to be always on a 4 byte boundary therfore
omitting the least significant 2 bits but to ensure backward
compatibility with 16-bit RISC-V instrution set
           we only ommit one*/
                 jumpToPCplusImm ? PCplusImm :
                 PCplus4;
       state <= needToWait ? WAIT_ALU_OR_MEM : FETCH_INSTR;
        end
```

- for the classic multicycle :

Risc_V

```
wire                PCWrite;
------------------------------------------------------------------------
Control_unit   ctrl(
     .clk(clock), .reset(reset), .run(run),.done(done),

     .zero(zero), .overflow(overflow), .sign(sign), .cout(cout),
    .funct3(funct3),
    .op(op), .funct7(funct7),

      .PCWrite(PCWrite), .Mem_Write(MemWrite), .IRWrite(IRWrite),
.RegWrite(RegWrite), .AdrSrc(AdrSrc),
    .ALUSrcA(ALUSrcA), .ALUSrcB(ALUSrcB),. ResultSrc(ResultSrc), .ALUControl(ALUControl),
```

```
    .state(state), .ImmSrc(ImmSrc),
      .size (size)
);

[ctrl] ↓
-------------------------------------------------------------------------
assign PCWrite = PCUpdate | (beq & zero) | (bne & ~zero) | (blt & (sign
!= overflow)) | (bge & (sign == overflow))
          | (bltu & ~cout) | (bgeu & cout);
-------------------------------------------------------------------------
d_flip_flop_with_enable PCFlipFlop (.out(PC), .CLK(CLK), .EN(PCWrite),
.in(Result),.RESET(RESET));
```

- By separating the next PC logic from the ALU (we still keep the jump instruction new PC Target calculations in the ALU since the jumps can be of 21-bit offsets) we get better performance with less resources.

② **Read the base register** (`rs1`) and **sign-extend the offset.**

- `A1 = Instr[19:15]` → Source register index to read base address.
- Output goes to temporary register `A`.
- `ImmSrc = 00` (for I-type 12-bit offset).
- Sign-extension unit outputs `ImmExt`.

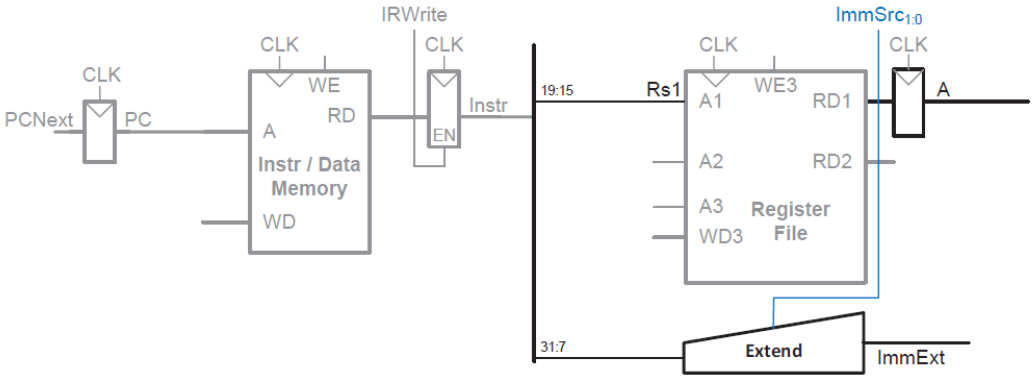


Figure30: Register Read & Immediate Extraction

- how it's implemented for this core :

BraRV32

```
assign aluIn1 = rs1;
-------------------------------------------------------------------------
assign loadstore_addr = rs1[ADDR_WIDTH-1:0] +
      (instr[5] ? Simm[ADDR_WIDTH-1:0] : Iimm[ADDR_WIDTH-1:0]);
-------------------------------------------------------------------------
// The five immediate formats, see RiscV reference (link above), Fig.
2.4 p. 12
assign Uimm = {    instr[31],   instr[30:12], {12{1'b0}}};
assign Iimm = {{21{instr[31]}}, instr[30:20]};
```

```
 /* verilator lint_off UNUSED */ // MSBs of SBJimms are not used by addr
adder.
assign  Simm = {{21{instr[31]}}, instr[30:25],instr[11:7]};
assign  Bimm = {{20{instr[31]}},instr[7],instr[30:25],instr[11:8],1'b0};
assign  Jimm = {{12{instr[31]}},instr[19:12],instr[20],instr[30:21],1'b0};
-------------------------------------------------------------------------
state[WAIT_INSTR_bit]: begin
if(!mem_rbusy) begin // may be high when executing from SPI flash or from
an EEPROM I²C
        rs1 <= registerFile[mem_rdata[19:15]]; /* read register file of
<current instruction> into the non-architectural registers A(rs1) and
rs2(WriteData) Figure 7.25 page 442 of the
Digital_Design_and_Computer_Architecture_RISC-V_Edition*/
        rs2 <= registerFile[mem_rdata[24:20]];
        instr <= mem_rdata[31:2]; // fetching <next instruction> since
                                  // <= is a non-blocking assignement
        state <= EXECUTE;
     end
```

- for the classic multicycle :

Risc_V

```
wire          [31:0] RD1, RD2, WriteData;

-------------------------------------------------------------------------
register_file registers (
  .ReadData1(RD1),
  .ReadData2(RD2),
  .RESET(RESET),
  .CLK(CLK),
  .ReadRegister1(Instr[19:15]),
  .ReadRegister2(Instr[24:20]),
  .WriteEnable(RegWrite),
  .WriteRegister(Instr[11:7]),
  .WriteData(Result),
  .register3(register3)
);
-------------------------------------------------------------------------
always @(*)
  begin
      case (ImmSrc)
          // I-type
                3'b000    : ImmExt <= {{20 {Instr[31]}}, Instr[31:20]};
          // S-type
          3'b001    : ImmExt <= {{20 {Instr[31]}},
                                 Instr[31:25],Instr[11:7]};
          // B-type
          3'b010    : ImmExt <= {{20 {Instr[31]}}, Instr[7],
                                 Instr[30:25],Instr[11:8], 1'b0};
          // J-type
          3'b011    : ImmExt <= {{12 {Instr[31]}}, Instr[19:12],
                                 Instr[20], Instr[30:21], 1'b0};
          // U-type
```

```
            3'b100    : ImmExt <= {Instr[31:12], {12 {1'b0}}};
            // others
            default   : ImmExt <= 32'd0;
        endcase
    end
-------------------------------------------------------------------------
Inst_decoder Imm_choice(.op(op), .ImmSrc(ImmSrc));
```

- Determining the immediate combinationally and avoiding the abundance of sequential logic for mundane tasks such as determining prefixed immediate encodings .

- The ALU's first input is always red from the first register operand, with the PC update offloaded to a separate adder. This reduces the number of multiplexers used for multiplexing inputs into the ALU, simplifying datapath routing.

- FSM is reduced into a single, compact control unit that sequentially integrates next-state logic and various task-specific combinational operations (declared elsewhere and not inside the FSM) within one always block. This approach minimizes redundant sequential logic, reducing Clk-to-Q delays caused by cascaded dependencies where one input depends on the other under the same clock.

③ **Add base address** (*A*) and **sign-extended immediate** (*ImmExt*)

- ALU Setup:
- `SrcA = A, SrcB = ImmExt`
- `ALUControl = 000 (for ADD)`

- Output Goes to `ALUOut`.

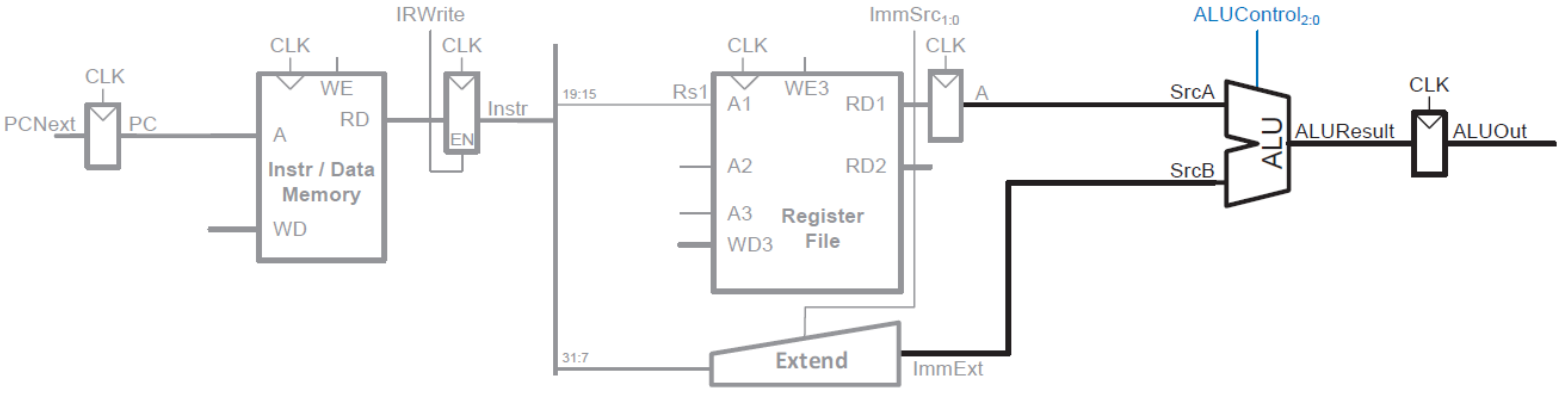


Figure31: Effective Address Calculation

- how it's implemented for this core :

BraRV32

```
// First ALU source, always rs1 (PC+4 and PC+immediate are treated
combinationally)
assign aluIn1 = rs1;
 // Second ALU source, depends on opcode:
 //    ALUreg, Branch:
 //    ALUimm, Load, JALR: Iimm
assign aluIn2 = isALUreg | isBranch ? rs2 : Iimm;
assign aluBusy = |aluShamt; // ALU is busy if shift amount is non-
//          n-1
zero,|x =    | x[k] ← ORing the bits of the x reg
//          k=0
// The adder is used by both arithmetic instructions and JALR.
assign aluPlus = aluIn1 + aluIn2;
assign aluMinus = {1'b1, ~aluIn2} + {1'b0,aluIn1} + 33'b1; // A-B =
A+~B+1 extra bit to asses overflow
/*instead of a more complex assign aluMinus = {~aluIn2[31], ~aluIn2} +
{aluIn1[31],aluIn1} + 33'b1 */

--------------------------------------------------------------------------
assign aluOut = (funct3Is[0]  ? instr[30] & instr[5] ? aluMinus[31:0] :
aluPlus : 32'b0) |
(funct3Is[2]  ? {31'b0, LT}                                          :32'b0)
|(funct3Is[3]  ? {31'b0, LTU}                                        :32'b0)
|(funct3Is[4]  ? aluIn1 ^ aluIn2                                     :32'b0)
|(funct3Is[6]  ? aluIn1 | aluIn2                                     :32'b0)
|(funct3Is[7]  ? aluIn1 & aluIn2                                     :32'b0)
|(funct3IsShift ? aluReg                                             :32'b0) ;
--------------------------------------------------------------------------
state[WAIT_ALU_OR_MEM_bit]: begin
    if(!aluBusy & !mem_rbusy & !mem_wbusy) state <= FETCH_INSTR;
  end
```

- for the classic multicycle :

Risc_V

```
module alu (
    input   [31:0] SrcA,
    input   [31:0] SrcB,
    input   [3:0]  aluc,
    output reg [31:0] Alu_out,
    output  reg       zero,
    output reg        cout,
    output  reg      overflow,
    output reg       sign
);

    always @(*) begin

         Alu_out      = 32'b0;
         cout         = 1'b0;
         overflow     = 1'b0;
```

```
        case (aluc)
            4'b0000: begin // ADD
                {cout, Alu_out} = {1'b0, SrcA} + {1'b0, SrcB};
            end
            4'b0001: begin // SUB
                {cout, Alu_out} = {1'b0, SrcA} + {1'b0, ~SrcB} + 1;
          … … … … … … … … … … … … … … … … … … … … … … … … … … … … …

-----------------------------------------------------------------------
always @(state) begin
    // Par défaut, tout à
    PCUpdate     <= 1'b0;
    Branch       <= 1'b0;
    Mem_Write    <= 1'b0;
    IRWrite      <= 1'b0;
    RegWrite     <= 1'b0;
    AdrSrc       <= 1'b0;
    sel_size     <= 1'b0;
    ALUSrcA      <= 2'b00;
    ALUSrcB      <= 2'b00;
    ResultSrc    <= 2'b00;
    ALUOp        <= 2'b00;
    done         <= 1'b0;
.. … .. … .. … .. … .. … .. … ... … .. … .. … .. … .. … .. … .. … .. … ..
          MemAdr         : begin
                                ALUSrcA    <= 2'b10;
                                ALUSrcB    <= 2'b01;
                           end
-----------------------------------------------------------------------
three_to_one_mux srcAMux   (.out(SrcA),   .selectBits(ALUSrcA),
.in1(PC),        .in2(OldPC),  .in3(A) ,.in4(32'dz));
three_to_one_mux srcBMux   (.out(SrcB),   .selectBits(ALUSrcB),
.in1(WriteData), .in2(ImmExt), .in3(32'd4) ,.in4(32'dz));

three_to_one_mux resultMux (.out(Result), .selectBits(ResultSrc),
.in1(ALUOut),    .in2(Data),   .in3(ALUResult) ,.in4(ImmExt));
alu ALU  (.SrcA(SrcA), .SrcB(SrcB) , .aluc(ALUControl),
.Alu_out(ALUResult), .zero(Zero), .cout(cout), .overflow(overflow),
.sign(sign));
```

- Simpler instruction decoding without the need for a complex sequential case block , and the use of multilevel ***2-to-1 mux's*** via the **? :** ternary conditional instead of a bulkier ***n-to-1 mux's*** thus taking a much simpler dataflow .

- Also an ALU that does operations and tests combinatorially, except shifts again reducing Clk-to-Q delays caused by cascaded dependencies , and the implementing less resources for the same work as the straightforward ALU used in the classic multicycle.

④ Using `ALUOut` *as* **the address** *to* **load the word** *from* **memory**

- AdrSrc = 1 → Address comes from ALUOut.
- MemRead = 1 → Enable memory read.
- Output → *goes to* → Data register.

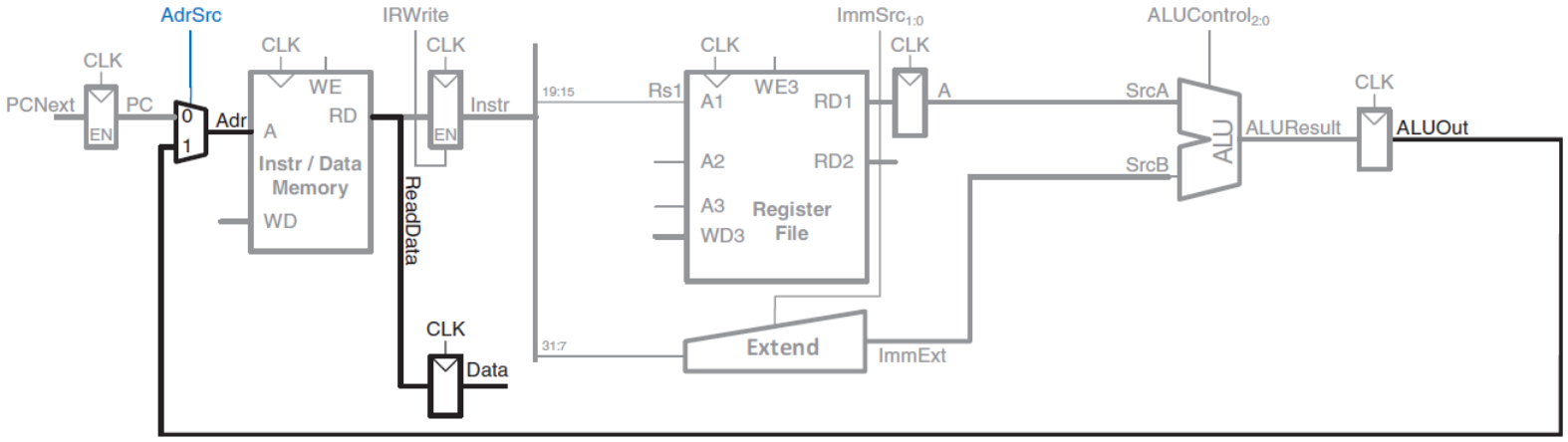


Figure32: Memory Access

- how it's implemented for this core :

BraRV32

```
assign mem_addr = state[WAIT_INSTR_bit] | state[FETCH_INSTR_bit] ?
       PC : loadstore_addr ;
-------------------------------------------------------------------------------
assign loadstore_addr = rs1[ADDR_WIDTH-1:0] +
      (instr[5] ? Simm[ADDR_WIDTH-1:0] : Iimm[ADDR_WIDTH-1:0]);
// The memory-read signal.
assign mem_rstrb =state[EXECUTE_bit] & isLoad | state[FETCH_INSTR_bit];

-------------------------------------------------------------------------------
   // All memory accesses are aligned on 32 bits(4 Bytes → a multiple of
4)boundary. For this
   // reason, we need some circuitry that does unaligned halfword
   // and byte load/store, based on:
   // - funct3[1:0]:  00->byte 01->halfword 10->word
   // - mem_addr[1:0]: indicates which byte/halfword is accessed

   assign mem_byteAccess     = instr[13:12] == 2'b00; // funct3[1:0] ==
2'b00;
   assign mem_halfwordAccess = instr[13:12] == 2'b01; // funct3[1:0] ==
2'b01;
-------------------------------------------------------------------------------
// LOAD, in addition to funct3[1:0], LOAD depends on:
   // - funct3[2] (instr[14]): 0->do sign expansion   1->no sign expansion

   assign LOAD_sign =
    !instr[14] & (mem_byteAccess ? LOAD_byte[7] : LOAD_halfword[15]);

   assign LOAD_data =
         mem_byteAccess ? {{24{LOAD_sign}},     LOAD_byte} :
```

```
     mem_halfwordAccess ? {{16{LOAD_sign}}, LOAD_halfword} :
                          mem_rdata ;

   assign LOAD_halfword =
          loadstore_addr[1] ? mem_rdata[31:16] : mem_rdata[15:0];
/*↓↓↓*/
    /*halfowrd is aligned on a 2byte boundary (0x0 → 0x2 → 0x4 → 0x6
...)⟷(000→010→100→110 ...)
    The LSB for unaligned access on unknown 2 byte boundary
mod2(0x0)|mod2(0x1)|mod2(0x2)
    case (loadstore_addr[1:0])
        2'b00: LOAD_halfword = mem_rdata[15:0];
// Case 0: Lower halfword
        2'b01: LOAD_halfword = {mem_rdata[23:16], mem_rdata[15:8]};
 // Case 1: Middle halfword
        2'b10: LOAD_halfword = mem_rdata[31:16];
// Case 2: Upper halfword
       default: LOAD_halfword = 16'hXXXX;
// Case 3: Invalid/unreachable (2'b11)
    endcases
    assign[15:0] LOAD_halfword = loadstore_addr[1] ? mem_rdata[31:16] :
                                 loadstore_addr[0] ? {mem_rdata[23:16],
mem_rdata[15:8]} :
                                 mem_rdata[15:0];*/
   assign LOAD_byte =
          loadstore_addr[0] ? LOAD_halfword[15:8] : LOAD_halfword[7:0];
/*↑↑↑*/
-------------------------------------------------------------------------
state[WAIT_INSTR_bit]: begin
if(!mem_rbusy) begin // may be high when executing from SPI flash or from an
EEPROM I²C
        rs1 <= registerFile[mem_rdata[19:15]]; /* read register file of
<current instruction> into the non-architectural registers A(rs1) and
rs2(WriteData) Figure 7.25 page 442 of the
Digital_Design_and_Computer_Architecture_RISC-V_Edition*/
        rs2 <= registerFile[mem_rdata[24:20]];
        instr <= mem_rdata[31:2]; // fetching <next instruction> since
                                  // <= is a non-blocking assignement
        state <= EXECUTE;
     end
-------------------------------------------------------------------------
```

- for the classic multicycle :

Risc_V

```
module datapath (OP, funct3, funct7, Zero,cout,overflow,sign, RESET,
CLK, ALUSrcA, ALUSrcB, ImmSrc, ResultSrc, ALUControl,AdrSrc, PCWrite,
MemWrite, RegWrite, IRWrite,size,register3,PCout);

.. … .. … .. … .. … .. … .. … ... … .. … .. … .. … .. … .. … .. … .. … ..
```

```
output   [6:0] OP;
output  [14:12] funct3;
output         funct7;
-------------------------------------------------------------------------------
always @(state) begin
    // Par défaut, tout à 0
.. … .. … .. … .. … .. … .. … .. … ... … .. … .. … .. … .. … .. … .. … .. … ..
     sel_size     <= 1'b0;
     ALUOp        <= 2'b00;
.. … .. … .. … .. … .. … .. … .. … ... … .. … .. … .. … .. … .. … .. … .. … ..
MemRead : begin
AdrSrc       <= 1'b1;
sel_size      <= 1'b1;
 end
-------------------------------------------------------------------------------
always @(*) begin
    if (sel_size)
            case (op)
                    7'b0000011,
                    7'b0100011: begin
                                        case (funct3)
                                            3'b000  : size  <= 3'b001;  // LB
SB
                                            3'b001  : size  <= 3'b010;  //LH
SH
                                            3'b010  : size  <= 3'b000;  //LW
SW
                                            3'b011  : size  <= 3'b011;  //LBU
                                            3'b100  : size  <= 3'b100;  //LHU
                                            default : size  <= 3'b000;
                                        endcase
                                    end
                    default : size  <= 3'b000;
            endcase
            else size  <= 3'b000;
end
```

- In the classic multicycle implementation, **memory access for load instructions** uses `ALUOut` as the effective address: when `AdrSrc = 1`, the memory address is taken from `ALUOut`, and `MemRead = 1` enables the memory read. The output is routed to a dedicated **Data register** for later use.

- In contrast, **BraRV32** *computes* **the effective address** inline using `loadstore_addr = rs1 + imm` and directly **assigns it** *to* `mem_addr` when needed, **bypassing the need for** `AdrSrc`, `ALUOut`, or an ***intermediate Data register***.

- This results in **fewer control signals**, **less sequential staging**, and a **tighter datapath**, optimized for area and speed, though less modular (only one extra adder).

- Data access type and load/store initiation are handled **combinationally**, avoiding complex `if-else` logic statements inside sequential `always` blocks. This is achieved through smart techniques that use **alignment information** and **read/write initiatives**, leveraging lightweight signals to interface with a memory module instead of hardwiring it directly into the design, these signals include **mask write fields** (via `mem_wmask`) for store instructions and to **precisely trigger memory communication** (via `mem_rstrb`) for load instructions. The result is a cleaner, more efficient control path with minimal sequential overhead.

```
// The memory-read signal.
assign mem_rstrb = state[EXECUTE_bit] & isLoad | state[FETCH_INSTR_bit];

// The mask for memory-write.
assign mem_wmask = {4{state[EXECUTE_bit] & isStore}} & STORE_wmask;
```

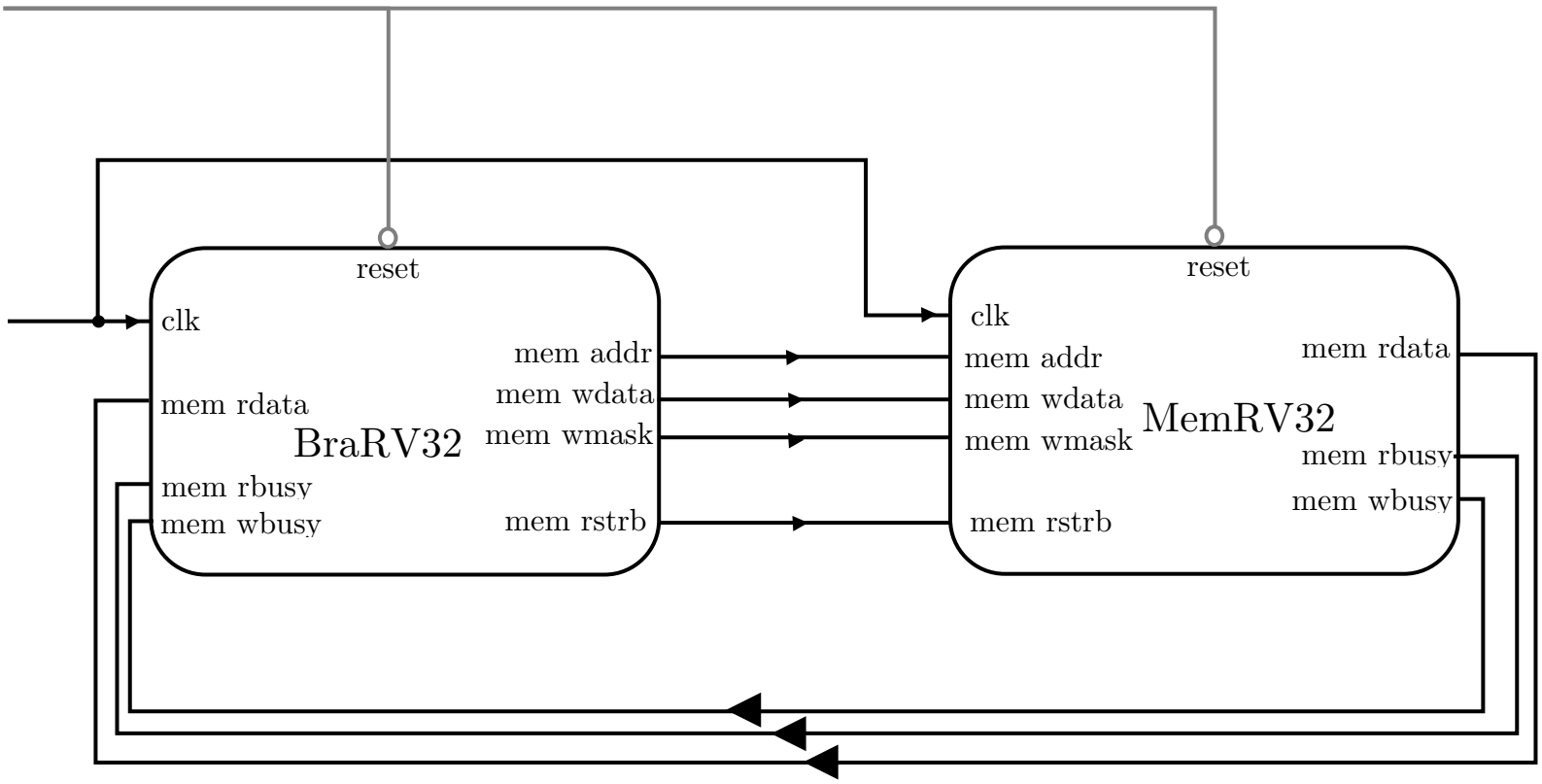


Figure33: The BraRV32 core interfacing with an external memory module via memory control signals

⑤ **Write the loaded word** *to the* **destination register** (`rd`)

- `A3 = Instr[11:7]` → Destination register index.
- `ResultSrc = 01` → Select `Data` as result input.
- `RegWrite = 1` → Enable register file write.

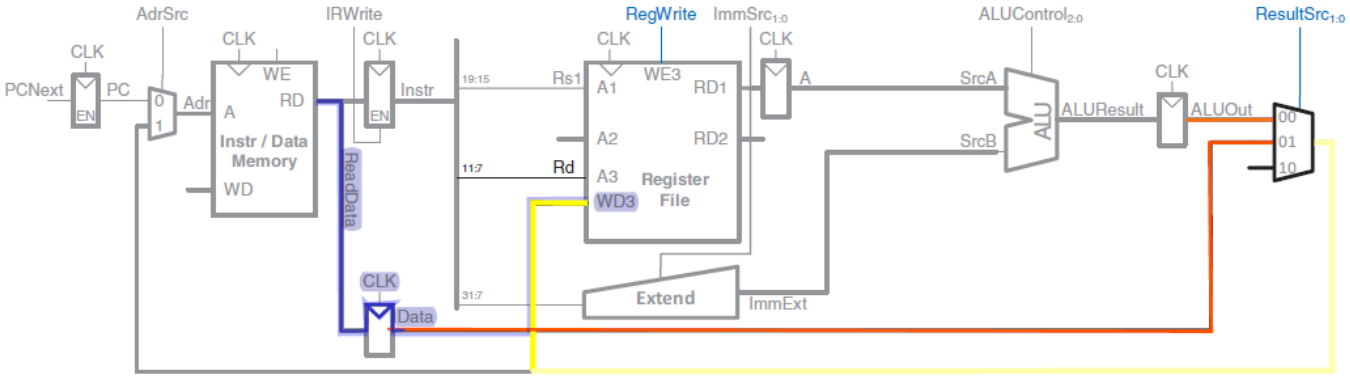


Figure34: Write Back to Register File

- how it's implemented for this core :

BraRV32

```
    (* ramstyle = "M10K" *) assign writeBack = ~(isBranch | isStore )&
(state[EXECUTE_bit]| state[WAIT_ALU_OR_MEM_bit]);
   // register write-back enable.
   assign writeBackData   =
      (isSYSTEM             ? cycles     : 32'b0) |  // SYSTEM
      (isLUI                ? Uimm       : 32'b0) |  // LUI
      (isALU                ? aluOut     : 32'b0) |  // ALUreg, ALUimm
      (isAUIPC              ? PCplusImm  : 32'b0) |  // AUIPC
      (isJALR   | isJAL     ? PCplus4    : 32'b0) |  // JAL, JALR
      (isLoad               ? LOAD_data  : 32'b0) ;  // Load

-------------------------------------------------------------------------------

   assign needToWait = isLoad | isStore | isALU & funct3IsShift;
-------------------------------------------------------------------------------
   always @(posedge clk) begin
     if (writeBack)
       if (rdId != 0)
         registerFile[rdId] <= writeBackData;
   end
```

- for the classic multicycle :

Risc_V

```
instruction_and_data_memory #(.MEMORY_SIZE(MEMORY_SIZE)) memory
        (
            .ReadData(ReadData),
            .RESET(RESET),
```

```
            .CLK(CLK),
            .WriteEnable(MemWrite),
            .Address(Address),
            .WriteData(WriteData),
            .size(size)
        );
-------------------------------------------------------------------------
d_flip_flop  dataFlipFlop
(.out(Data),.CLK(CLK),.in(ReadData),.RESET(RESET));
-------------------------------------------------------------------------
three_to_one_mux resultMux (.out(Result), .selectBits(ResultSrc),
.in1(ALUOut),    .in2(Data),    .in3(ALUResult)
,.in4(ImmExt));
-------------------------------------------------------------------------
    register_file registers (
        .ReadData1(RD1),
        .ReadData2(RD2),
        .RESET(RESET),
        .CLK(CLK),
        .ReadRegister1(Instr[19:15]),
        .ReadRegister2(Instr[24:20]),
        .WriteEnable(RegWrite),
        .WriteRegister(Instr[11:7]),
        .WriteData(Result),
          .register3(register3)
    );
-------------------------------------------------------------------------
    alu ALU (.SrcA(SrcA), .SrcB(SrcB) , .aluc(ALUControl),
.Alu_out(ALUResult), .zero(Zero), .cout(cout),.overflow(overflow),
.sign(sign));
-------------------------------------------------------------------------
    d_flip_flop ALUFlipFlop (.out(ALUOut), .CLK(CLK),
.in(ALUResult),.RESET(RESET));
```

- Write-back to the register file in **BraRV32** is optimized through a **combinational generation of the write-back data source** (`writeBackData`) and a **conditional write-enable signal** (`writeBack`) that simplifies control.

- The **destination register index** (`rdId`) is used only if non-zero, avoiding writes to `x0` (**which is always 0**).
- Instead of relying on a multiplexer controlled by `ResultSrc`, the `writeBackData` is **selected using OR-combined conditions** based on instruction type, making it compact and combinational.
- The **write-enable** logic (`writeBack`) is evaluated using minimal conditions:

→ write-back occurs only if the instruction is **not a branch or store**.

→ and the state is either `EXECUTE` or `WAIT_ALU_OR_MEM`.

- The **needToWait** signal abstracts stall logic for loads and shift operations (we don't need any **NOP**'s to get functional unit latencies , it is handled in hardware not software), preventing premature write-back.

- This avoids a separate **write-back FSM state**, reduces **multiplexers**, and minimizes **register file write conflicts**, making the multicycle's behavior predicatble, smoother and timing-friendly.

⑥ **ALU** *computes* **`PC + 4`** *and updates* **PC**.

- `SrcA = PC, SrcB = 4`
- `Output goes to PC via PCWrite = 1.`

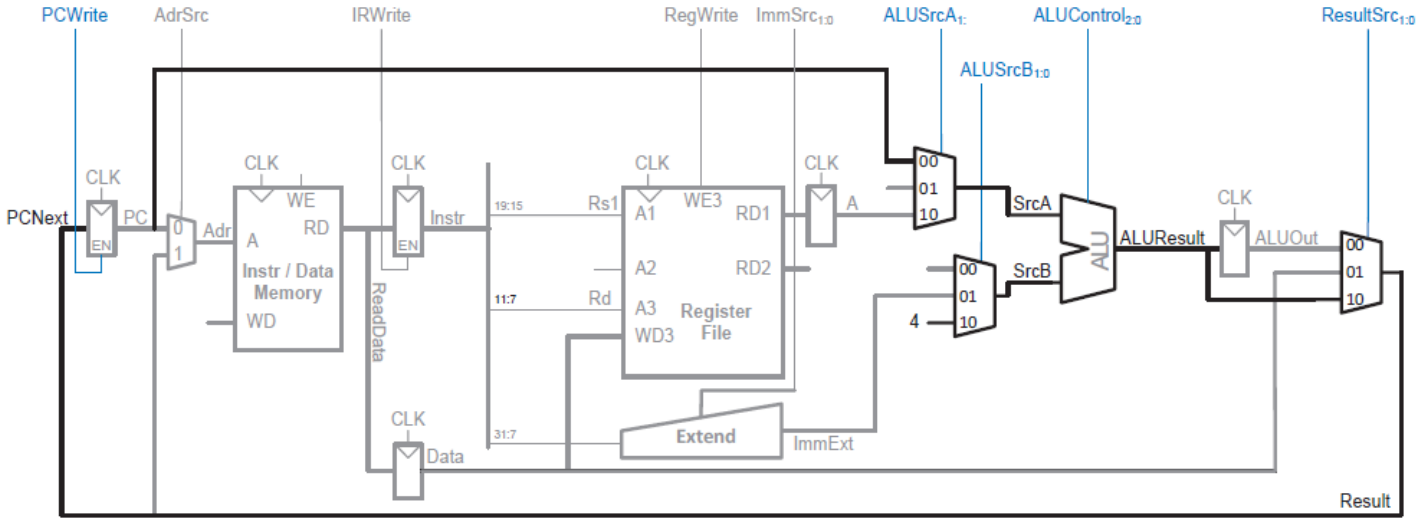


Figure35: ALU computes **`PC + 4`** and updates **PC** during fetch

- how it's implemented for this core :

BraRV32

```
assign PCplus4 = PC + 4;

// An adder used to compute branch address, JAL address and AUIPC.
// branch->PC+Bimm    AUIPC->PC+Uimm    JAL->PC+Jimm
// Equivalent to PCplusImm = PC + (isJAL ? Jimm : isAUIPC ? Uimm : Bimm)
```

```
   assign PCplusImm = PC + ( instr[3] ? Jimm[ADDR_WIDTH-1:0] :
                             instr[4] ? Uimm[ADDR_WIDTH-1:0] :
                                        Bimm[ADDR_WIDTH-1:0] );

-------------------------------------------------------------------------------
   assign jumpToPCplusImm = isJAL | (isBranch & predicate);
-------------------------------------------------------------------------------
   assign predicate =
        funct3Is[0] &  EQ  | // BEQ
        funct3Is[1] & !EQ  | // BNE
        funct3Is[4] &  LT  | // BLT
        funct3Is[5] & !LT  | // BGE
        funct3Is[6] &  LTU | // BLTU
        funct3Is[7] & !LTU ; // BGEU

-------------------------------------------------------------------------------
   always @(posedge clk) begin

      if(!reset) begin
         state      <= WAIT_ALU_OR_MEM; // Just waiting for !mem_wbusy
         PC         <= RESET_ADDR[ADDR_WIDTH-1:0];
      end else
.. … .. … .. … .. … .. … .. … ... … .. … .. … .. … .. … .. … .. … .. … ..
      state[EXECUTE_bit]: begin
                  PC <= isJALR          ? {aluPlus[ADDR_WIDTH-1:1],1'b0}:
                  /*instructions assumed to be always on a 4 byte boundary
                  therfore omitting the least ignificant 2 bits but to ensure
                  backward compatibility with 16-bit RISC-V instrution set
                  we only ommit one*/
                        jumpToPCplusImm ? PCplusImm :
                        PCplus4;
               state <= needToWait ? WAIT_ALU_OR_MEM : FETCH_INSTR;
                end
```

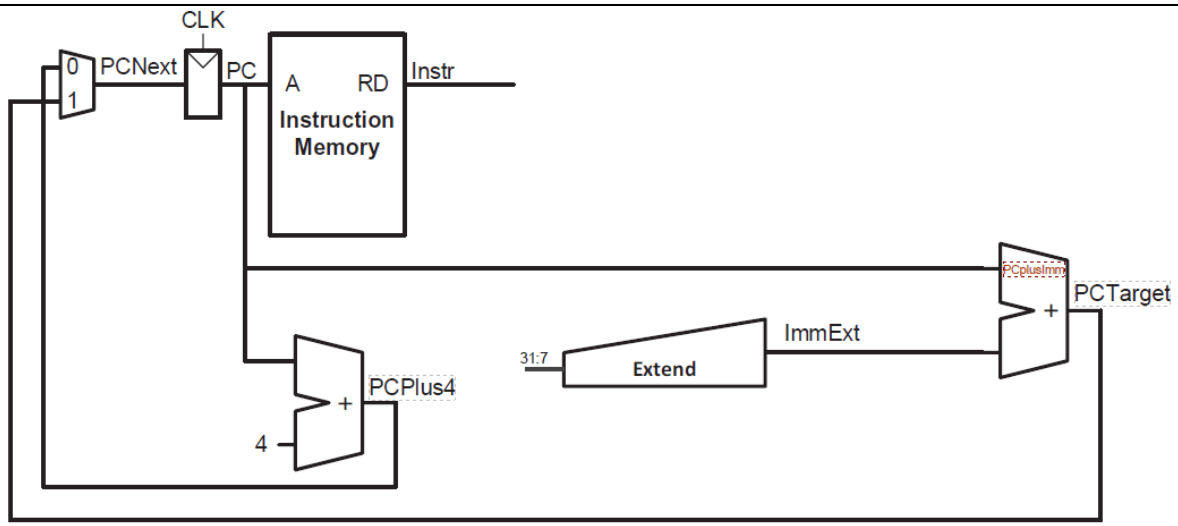


Figure36: Separation of Program counter evaluation from the ALU implemented in BraRV32

- for the classic multicycle :

Risc_V

```
// Control signal generation
always @(state) begin
    // Par défaut, tout à 0

    PCUpdate     <= 1'b0;
     Branch       <= 1'b0;
    Mem_Write    <= 1'b0;
    IRWrite      <= 1'b0;
    RegWrite     <= 1'b0;
    AdrSrc       <= 1'b0;
    sel_size      <= 1'b0;
    ALUSrcA      <= 2'b00;
    ALUSrcB      <= 2'b00;
    ResultSrc    <= 2'b00;
    ALUOp        <= 2'b00;
     done            <= 1'b0;

    case (state)
         idle       :
                                done       <= 1'b1;

        Fetch       : begin
                                sel_size      <= 1'b1;
                                IRWrite    <= 1'b1;
                                ALUSrcB    <= 2'b10;
                                ResultSrc  <= 2'b10;
                                PCUpdate   <= 1'b1;
                            end
.. … .. … .. … .. … .. … .. … .. … ... … .. … .. … .. … .. … .. … .. … .. … ..
           MemWB      : begin
                                ResultSrc    <= 2'b01;
                                RegWrite     <= 1'b1;
                            end
-------------------------------------------------------------------------
three_to_one_mux srcAMux   (.out(SrcA),   .selectBits(ALUSrcA),
.in1(PC),        .in2(OldPC),  .in3(A) ,.in4(32'dz));
-------------------------------------------------------------------------
    three_to_one_mux srcBMux   (.out(SrcB),
.selectBits(ALUSrcB),   .in1(WriteData), .in2(ImmExt),
.in3(32'd4) ,.in4(32'dz));
-------------------------------------------------------------------------
    three_to_one_mux resultMux (.out(Result),
.selectBits(ResultSrc), .in1(ALUOut),    .in2(Data),
.in3(ALUResult) ,.in4(ImmExt));
-------------------------------------------------------------------------
    alu ALU (.SrcA(SrcA), .SrcB(SrcB) , .aluc(ALUControl),
.Alu_out(ALUResult), .zero(Zero), .cout(cout),
     .overflow(overflow), .sign(sign));
-------------------------------------------------------------------------
wire beq,bne,blt,bge,bltu,bgeu;
```

```
assign  beq  = Branch & (funct3 == 3'b000);
assign  bne  = Branch & (funct3 == 3'b001);
assign  blt  = Branch & (funct3 == 3'b100);
assign  bge  = Branch & (funct3 == 3'b101);
assign  bltu = Branch & (funct3 == 3'b110);
assign  bgeu = Branch & (funct3 == 3'b111);
assign PCWrite = PCUpdate | (beq & zero) | (bne & ~zero) | (blt &
(sign != overflow)) | (bge & (sign == overflow))|(bltu & ~cout) |
(bgeu & cout);
```

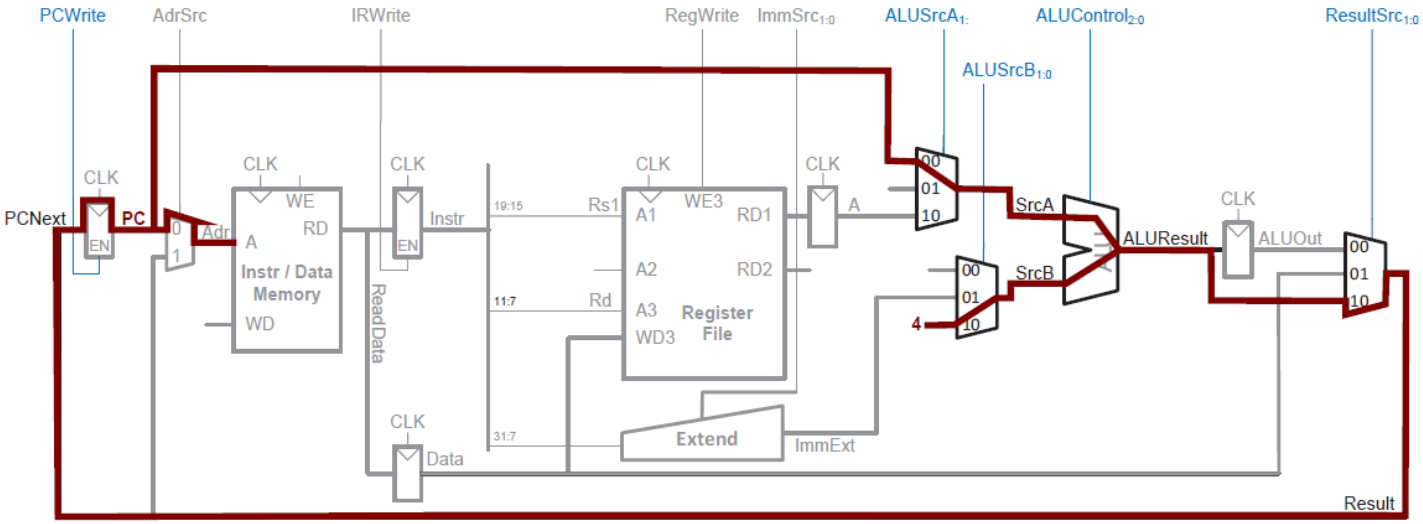


Figure37: Evaluation of the Program counter via the ALU implemented in Risc_V (classic multicycle)

Feature	BraRV32	Classic Multicycle (Textbook)
PC Logic	Separate adder, minimal MUX	Shared ALU, controlled by AdrSrc
FSM Design	Flat, compact, data-path aware	Modular, state-driven, external control
Memory Access	Bit-sliced, supports unaligned memory	Abstracted, assumes aligned access
Control Signals	Derived directly from instr bits	Pre-decoded via FSM output
Sequential Handling	Fewer states, less pipeline staging	Structured staging with explicit control
Resource Usage	Minimal (good for FPGA/ASIC constraints)	Moderate to high (better for understanding/modularity)
Performance	Higher (due to shallow logic and fewer cycles)	Lower (more cycles and wider control)
Extensibility	Harder (more spaghetti logic with more instructions)	Easier (plug-in structure with FSM and MUX expansion)

Table1: Summary Table of different features between BraRV32 and Risc_V multicycle implementations

• BraRV32 was carefully put together to showcase a **micro-optimized implementation** favoring *pragmatic hardware usage*, *low resource footprint*, and ***fast critical path performance***. It's ideal for embedded & FPGA systems (which is tailored to our use case).

• On the other hand, the classic multicycle design, although more verbose and resource-heavy, emphasizes clarity, flexibility, and a beginner friendly educational value.

4.3 Applied Optimization Techniques :

- BraRV32 applies several architectural and logic-level optimizations to improve timing, reduce resource usage, and simplify control logic ; especially compared to textbook tied traditional multicycle implementation:

- **Decoupled PC and ALU Logic**: The PC is incremented using a separate adder, leaving the ALU dedicated to operand computations. This reduces multiplexer pressure and simplifies operand routing.
- **Unified Next-State Control**: A single FSM controls all stages. Instead of separate control for each cycle, the FSM leverages smart combinational blocks for decode, memory interface, and write-back logic, reducing redundant sequential logic and improving clock-to-Q timing.
- **Memory Address and Access Logic**: Memory address (`mem_addr`) is selected using simple combinational logic between `PC` and `loadstore_addr`, derived from `rs1 + immediate`. Aligned/unaligned access handling and sign-extension are implemented via lightweight, condition-driven combinational masks without if-else branching inside clocked blocks.
- **Data Fetch and Alignment**: LOAD operations smartly determine whether to sign-extend or zero-extend based on `funct3`, and use the LSBs of `loadstore_addr` to extract bytes or halfwords, all via compact logic. This avoids deep conditional trees and adds support for unaligned loads with minimal overhead.
- **Register Write-Back Simplification**: Instead of a dedicated write-back mux controlled by `ResultSrc`, `writeBackData` is computed in parallel via bitwise ORs depending on instruction type (e.g., `JAL`, `AUIPC`, `ALU`, `LOAD`). A single `writeBack` enable signal conditions the update based on state and instruction, skipping unnecessary writes to `x0`.
- **Hazard-Aware Stalling**: The design introduces a `needToWait` signal for instructions that require memory latency or multi-cycle shifts, integrated into the FSM without complicating the datapath.

- These combined techniques result in **reduced control complexity**, **faster critical paths**, and **fewer LUTs and flip-flops**, while preserving functionality demonstrating the practical benefits of our minimalistic RISC-V core design.

4.3.1 One-Hot FSM Encoding :

- BraRV32 adopts a **one-hot encoding scheme** for its finite state machine (FSM) and decode bit fields, where each state is represented by a distinct **flip-flop** set to `1`, while all others remain `0`. This contrasts with traditional binary or Gray encoding which requires decoding logic to identify active states.

```
(* syn_encoding = "onehot" *)assign funct3Is = 8'b00000001 << instr[14:12];
```

funct3Is | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 8-bits → *one_hote* → 8 cases

funct3 | 0 | 0 | 0 | 3-bits → $2^3 = 8$ cases

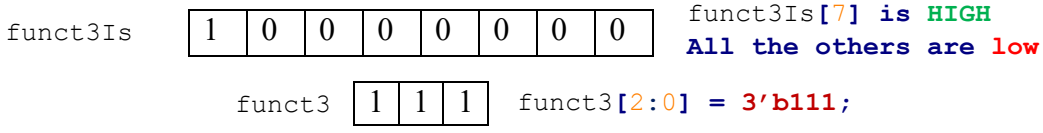
• in Risc_V ALU_decoder we have the following case statement nested inside another case statement , in each case statement depending on the n-bit width of the register to test we'd need 2^n MUX's which overall give a tradeoff of → **n-bit register (n-flipflops) + 2^n MUX's**

•Whereas through one hote encoding we both simplify the decoding logic and avoid sequential checking inside always blocks and resolve the task of determining the instruction type / operation type/immediate encoding...ect combinationally , the latter yields → **2^n-bit register (2^n-flipflops) + n-MUX's**

Risc_V

```
always @(*) begin
    case (ALUop)
        2'b00 : ALU_Control <= 4'b0000; //somme
        2'b01 : ALU_Control <= 4'b0001; //difference
        2'b10 : begin
                        case (funct3)
                            3'b000  : ALU_Control  <= {{3 {1'b0}}, funct7_5 };
//SOMME(0000) OU DIFFERENCE(0001)
                            3'b001  : ALU_Control  <= 4'b0010;
//SLL
                            3'b010  : ALU_Control  <= 4'b0011;
//SLT
                            3'b011  : ALU_Control  <= 4'b0101;
//SLTU
                            3'b100  : ALU_Control  <= 4'b0100;
//XOR
                            3'b101  : ALU_Control  <= {1'b0 , {2 {1'b1}},
funct7_5};  //SRL(0110) OR SRA(0111)
                            3'b110  : ALU_Control  <= 4'b1000;
//OR
```

```
                                3'b111   : ALU_Control  <= 4'b1001;
//AND
                                default  : ALU_Control  <= 4'bzzzz;
                            endcase
                        end
            2'b11 : begin
                            case (funct3)
                                3'b000   : ALU_Control  <= 4'b1010;
//MPY
                                3'b001   : ALU_Control  <= 4'b1011;
//MPYH
                                3'b010   : ALU_Control  <= 4'b1100;
//DIV
                                3'b011   : ALU_Control  <= 4'b1101;
//MODULO
                                default  : ALU_Control  <= 4'bzzzz;
                            endcase
                        end


    endcase

end
```



BraRV32

```
assign funct3IsShift = funct3Is[1] | funct3Is[5];
   assign aluOut =
     (funct3Is[0]  ? instr[30] & instr[5] ? aluMinus[31:0] : aluPlus : 32'b0)|
     (funct3Is[2]  ? {31'b0, LT}                                     : 32'b0)|
     (funct3Is[3]  ? {31'b0, LTU}                                    : 32'b0)|
     (funct3Is[4]  ? aluIn1 ^ aluIn2                                 : 32'b0)|
     (funct3Is[6]  ? aluIn1 | aluIn2                                 : 32'b0)|
     (funct3Is[7]  ? aluIn1 & aluIn2                                 : 32'b0)|
     (funct3IsShift ? aluReg                                         : 32'b0);-
```

♦ For the state machine :

```
   localparam NB_STATES = 4;
   reg [NB_STATES-1:0] state;

   localparam FETCH_INSTR_bit     = 0;
   localparam WAIT_INSTR_bit      = 1;
   localparam EXECUTE_bit         = 2;
   localparam WAIT_ALU_OR_MEM_bit = 3;


   localparam FETCH_INSTR     = 1 << FETCH_INSTR_bit;
   localparam WAIT_INSTR      = 1 << WAIT_INSTR_bit;
```

```
   localparam EXECUTE         = 1 << EXECUTE_bit;
   localparam WAIT_ALU_OR_MEM = 1 << WAIT_ALU_OR_MEM_bit;

   always @(posedge clk) begin
      if(!reset) begin
         state      <= WAIT_ALU_OR_MEM; // Just waiting for !mem_wbusy
         PC         <= RESET_ADDR[ADDR_WIDTH-1:0];
      end else

      // See note [1] at the end of this file.
      //(*parallel_case*)
      case(1'b1)
        state[WAIT_INSTR_bit]: begin
           if(!mem_rbusy) begin // may be high when executing from SPI flash
or from an EEPROM I²C
              rs1 <= registerFile[mem_rdata[19:15]]; /* read register file of
<current instruction>
              into the non-architectural registers A(rs1) and rs2(WriteData)
Figure 7.25 page 442 of the
              Digital_Design_and_Computer_Architecture_RISC-V_Edition*/
              rs2 <= registerFile[mem_rdata[24:20]];
              instr <= mem_rdata[31:2]; // fetching <next instruction> since
<= is a non-blocking assignement
              state <= EXECUTE;
           end
        end

        state[EXECUTE_bit]: begin
           PC <= isJALR          ? {aluPlus[ADDR_WIDTH-1:1],1'b0}:
           /*instructions assumed to be always on a 4 byte boundary therfore
omitting the least
           significant 2 bits but to ensure backward compatibility with 16-bit
RISC-V instrution set
           we only ommit one*/
                jumpToPCplusImm ? PCplusImm :
                PCplus4;
       state <= needToWait ? WAIT_ALU_OR_MEM : FETCH_INSTR;
        end

        state[WAIT_ALU_OR_MEM_bit]: begin
           if(!aluBusy & !mem_rbusy & !mem_wbusy) state <= FETCH_INSTR;
        end

        default: begin // FETCH_INSTR
          state <= WAIT_INSTR;
        end

      endcase
   end
```

- This method enables:

- **Simplified State Transitions**: Each state is directly represented by a single-bit signal, making next-state logic highly readable and easily synthesized as simple AND/OR conditions.
- **Faster Clock-to-Q and Reduced Logic Depth**: Because no decoding is needed, critical paths are shorter, which improves maximum clock frequency.
- **Easier Control Signal Assignment**: Control signals (e.g., `mem_rstrb`, `writeBack`, `loadstore_addr`) are gated directly with one-hot state bits `FETCH_INSTR_bit`, `EXECUTE_bit`, or `WAIT_ALU_OR_MEM_bit`, eliminating the need for complex case statements or priority encoders.

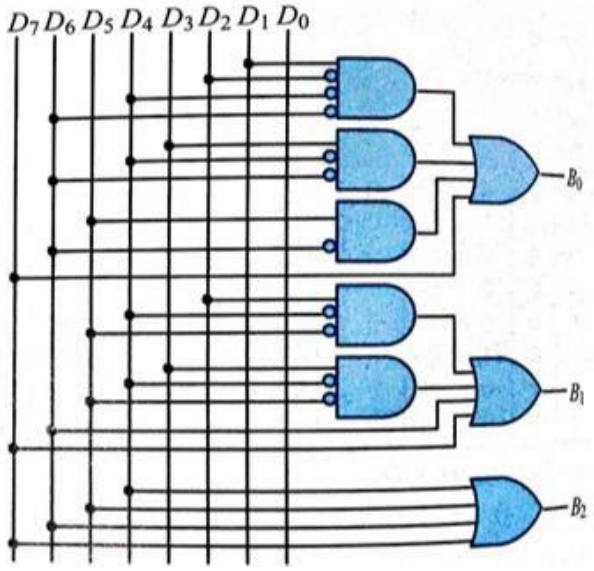
4.3.2 Use of `parallel_case` *for Synthesis Hints :*

- BraRV32 leverages the Verilog synthesis directive `parallel_case` within its control FSM to guide the synthesis tool in generating **parallelized, optimized logic**, under the assumption that **only one case condition will be true at any given time** (as ensured by the one-hot encoding), it :

  - **Reduced Priority Logic**: Without `parallel_case`, synthesis tools may infer a priority encoder, introducing unnecessary logic depth. Declaring `parallel_case` explicitly removes this ambiguity.
  - **Faster Synthesis and Timing Closure**: It allows synthesis tools to flatten the decision tree into **pure combinational logic**, avoiding chained conditionals and reducing propagation delay.
  - **Optimized Area for State Decoding**: With one-hot FSM, each state is already mutually exclusive; `parallel_case` formalizes this, enabling more aggressive optimization, hence less logic blocks.

$$B_2 = D_7 + D_6 + D_5 + D_4$$
$$B_1 = D_7 + D_6 + \bar{D}_5\bar{D}_4D_3 + \bar{D}_5\bar{D}_4D_2$$
$$B_0 = D_7 + \bar{D}_6D_5 + \bar{D}_6\bar{D}_4D_3 + \bar{D}_6\bar{D}_4\bar{D}_2D_1$$

input								output		
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	B_2	B_1	B_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	×	0	0	1
0	0	0	0	0	1	×	×	0	1	0
0	0	0	0	1	×	×	×	0	1	1
0	0	0	1	×	×	×	×	1	0	0
0	0	1	×	×	×	×	×	1	0	1
0	1	×	×	×	×	×	×	1	1	0
1	×	×	×	×	×	×	×	1	1	1



INPUTS								OUTPUTS		
Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0	A2	A1	A0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

Table2: The truth table for the 8 to 3 encoder

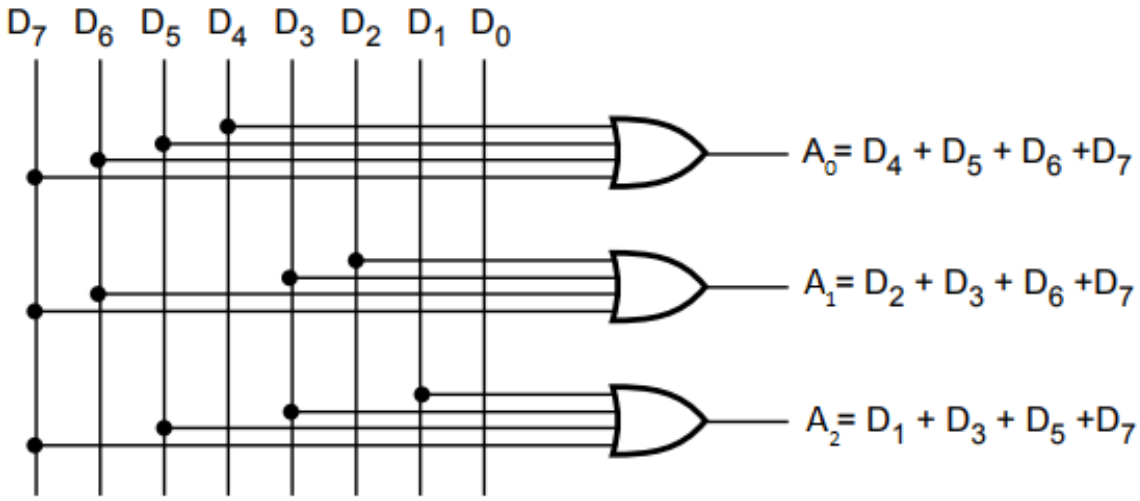


Figure39: An 8x3 Encoder

- This hint is particularly useful for compact RISC-V cores such our optimized multicycle, where **predictable state transitions** and **tight control paths** are critical for area and performance efficiency on FPGAs.

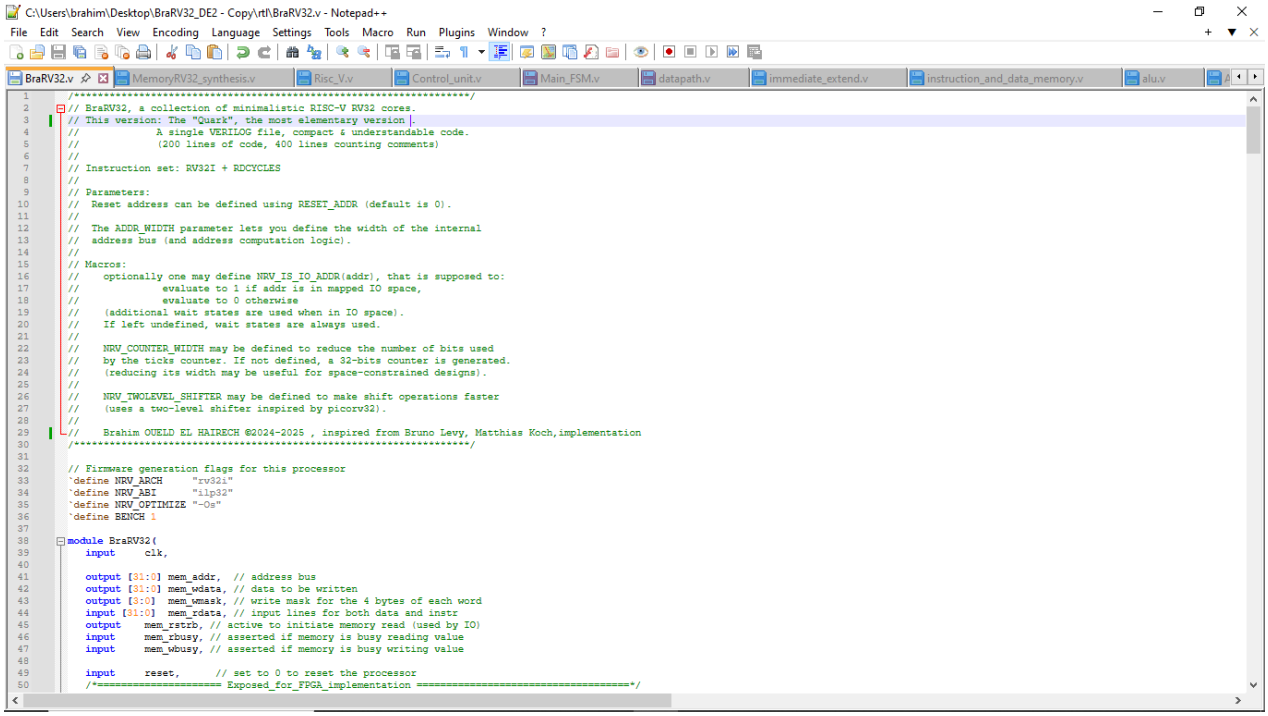
4.3.3 Code Flattening and Modular Consolidation :

- BraRV32 adopts **code flattening** and **modular consolidation** to reduce control overhead and improve synthesis efficiency. Instead of deeply nested `always` blocks or overly granular submodules, critical operations like **instruction decode, control signal generation**, and **execution dispatch** are handled within a single Verilog file that incorporate all necessary bodies where all relevant control signals, fetch, execute and write operations are determined combinationally and only registered when it's indispensable to do so , and it is managed by a single cohesive FSM block :

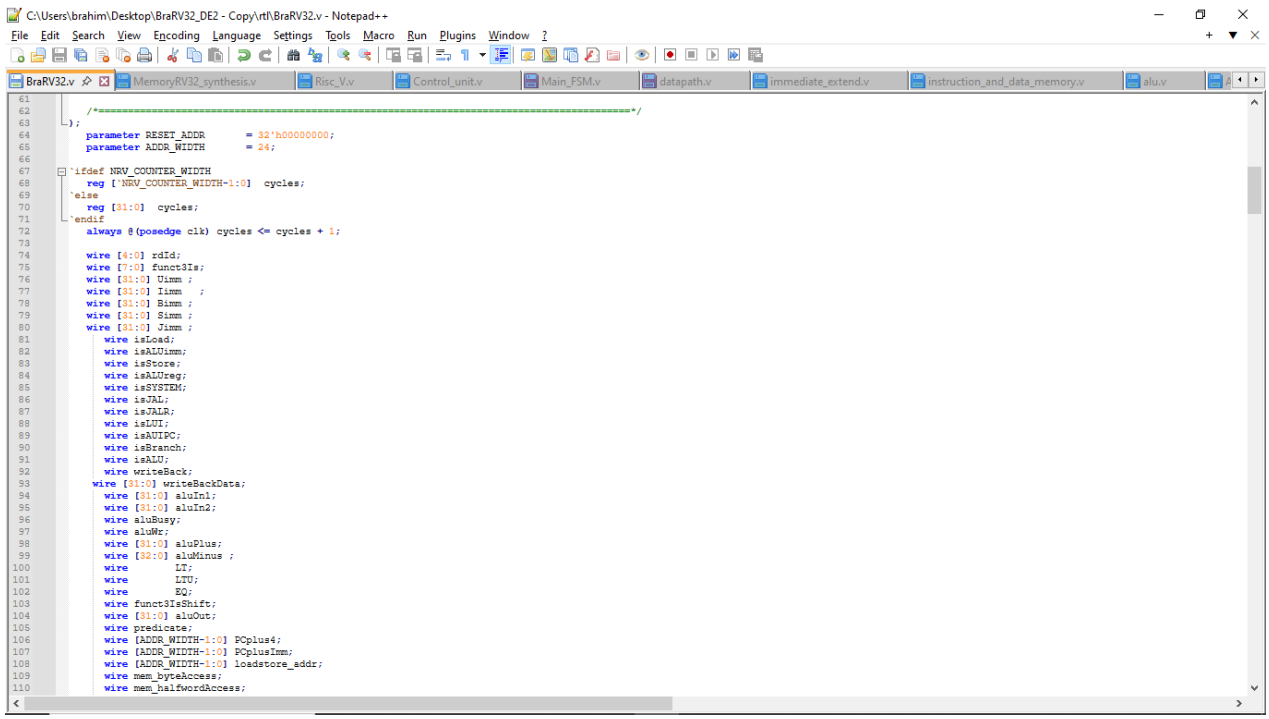
- **Lower Register and Mux Count**: Flattening avoids unnecessary intermediate wires, state registers, and control multiplexers.
- **Improved Clock-to-Q Timing**: Sequential logic is minimized, especially for control paths that would otherwise cascade through multiple state updates or modules.
- **Better Optimization by Synthesis Tools**: A flatter structure enables global signal inference and resource sharing, which leads to reduced logic duplication and improved timing closure.

- This approach contrasts with classic multicycle implementations where stages like decode, execute, and write-back are spread across multiple layers or modules, often introducing overhead in signal routing and coordination.

- BraRV32's consolidation reduces that by evaluating many control decisions combinationally and only storing what's necessary (as stated by earlier code snippets) boosting both area and speed efficiency.



```
/*****************************************************************/
// BraRV32, a collection of minimalistic RISC-V RV32 cores.
// This version: The "Quark", the most elementary version
//             A single VERILOG file, compact & understandable code.
//             (200 lines of code, 400 lines counting comments)
//
// Instruction set: RV32I + RDCYCLES
//
// Parameters:
//  Reset address can be defined using RESET_ADDR (default is 0).
//
//  The ADDR_WIDTH parameter lets you define the width of the internal
//  address bus (and address computation logic).
//
// Macros:
//    optionally one may define NRV_IS_IO_ADDR(addr), that is supposed to:
//             evaluate to 1 if addr is in mapped IO space,
//             evaluate to 0 otherwise
//    (additional wait states are used when in IO space).
//    If left undefined, wait states are always used.
//
//    NRV_COUNTER_WIDTH may be defined to reduce the number of bits used
//    by the ticks counter. If not defined, a 32-bits counter is generated.
//    (reducing its width may be useful for space-constrained designs).
//
//    NRV_TWOLEVEL_SHIFTER may be defined to make shift operations faster
//    (uses a two-level shifter inspired by picorv32).
//
//    Brahim OUELD EL HAIRECH @2024-2025 , inspired from Bruno Levy, Matthias Koch, implementation
/*****************************************************************/

// Firmware generation flags for this processor
`define NRV_ARCH     "rv32i"
`define NRV_ABI      "ilp32"
`define NRV_OPTIMIZE "-Os"
`define BENCH 1

module BraRV32(
   input     clk,

   output [31:0] mem_addr,  // address bus
   output [31:0] mem_wdata, // data to be written
   output [3:0]  mem_wmask, // write mask for the 4 bytes of each word
   input [31:0]  mem_rdata, // input lines for both data and instr
   output    mem_rstrb, // active to initiate memory read (used by IO)
   input     mem_rbusy, // asserted if memory is busy reading value
   input     mem_wbusy, // asserted if memory is busy writing value

   input     reset,      // set to 0 to reset the processor
   /*===================== Exposed_for_FPGA_implementation ==================================*/
```



```
   /*=========================================================================================*/
);
   parameter RESET_ADDR       = 32'h00000000;
   parameter ADDR_WIDTH       = 24;

`ifdef NRV_COUNTER_WIDTH
   reg [`NRV_COUNTER_WIDTH-1:0]  cycles;
`else
   reg [31:0]  cycles;
`endif
   always @(posedge clk) cycles <= cycles + 1;

   wire [4:0] rdId;
   wire [7:0] funct3Is;
   wire [31:0] Uimm ;
   wire [31:0] Iimm   ;
   wire [31:0] Bimm ;
   wire [31:0] Simm ;
   wire [31:0] Jimm ;
     wire isLoad;
     wire isALUimm;
     wire isStore;
     wire isALUreg;
     wire isSYSTEM;
     wire isJAL;
     wire isJALR;
     wire isLUI;
     wire isAUIPC;
     wire isBranch;
     wire isALU;
     wire writeBack;
    wire [31:0] writeBackData;
     wire [31:0] aluIn1;
     wire [31:0] aluIn2;
     wire aluBusy;
     wire aluWr;
     wire [31:0] aluPlus;
     wire [32:0] aluMinus ;
     wire        LT;
     wire        LTU;
     wire        EQ;
     wire funct3IsShift;
     wire [31:0] aluOut;
     wire predicate;
     wire [ADDR_WIDTH-1:0] PCplus4;
     wire [ADDR_WIDTH-1:0] PCplusImm;
     wire [ADDR_WIDTH-1:0] loadstore_addr;
     wire mem_byteAccess;
     wire mem_halfwordAccess;
```

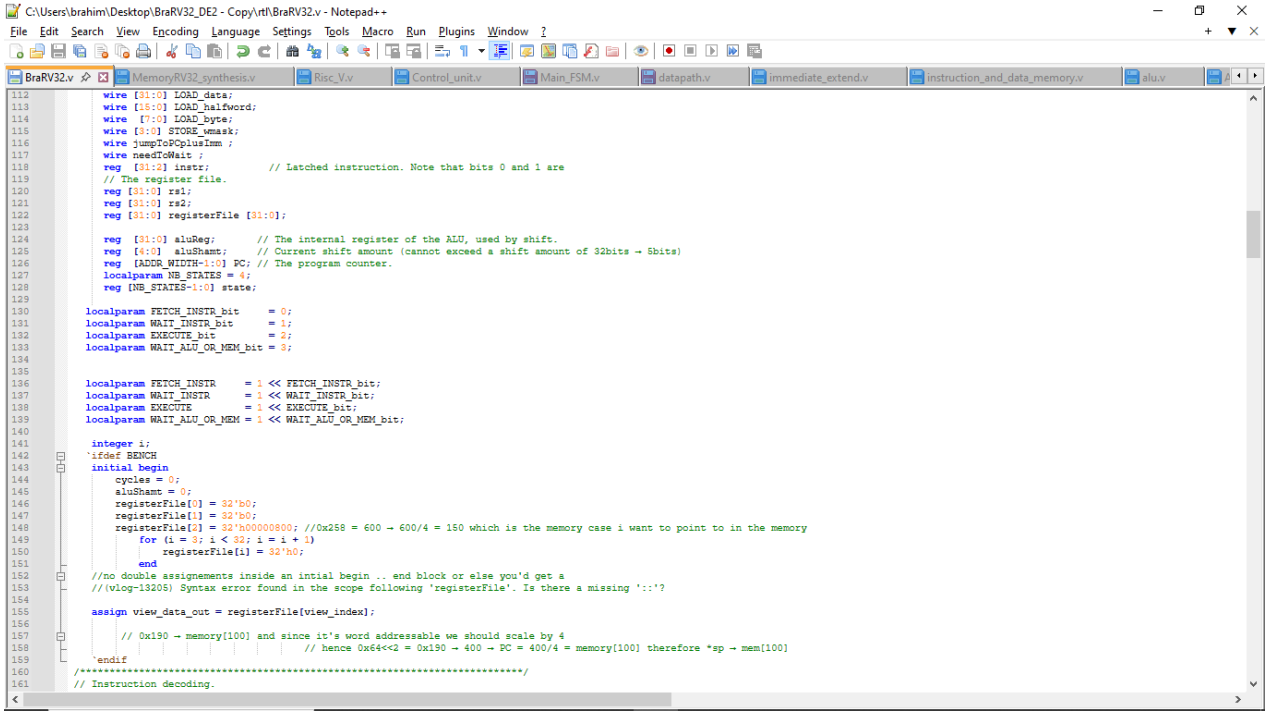
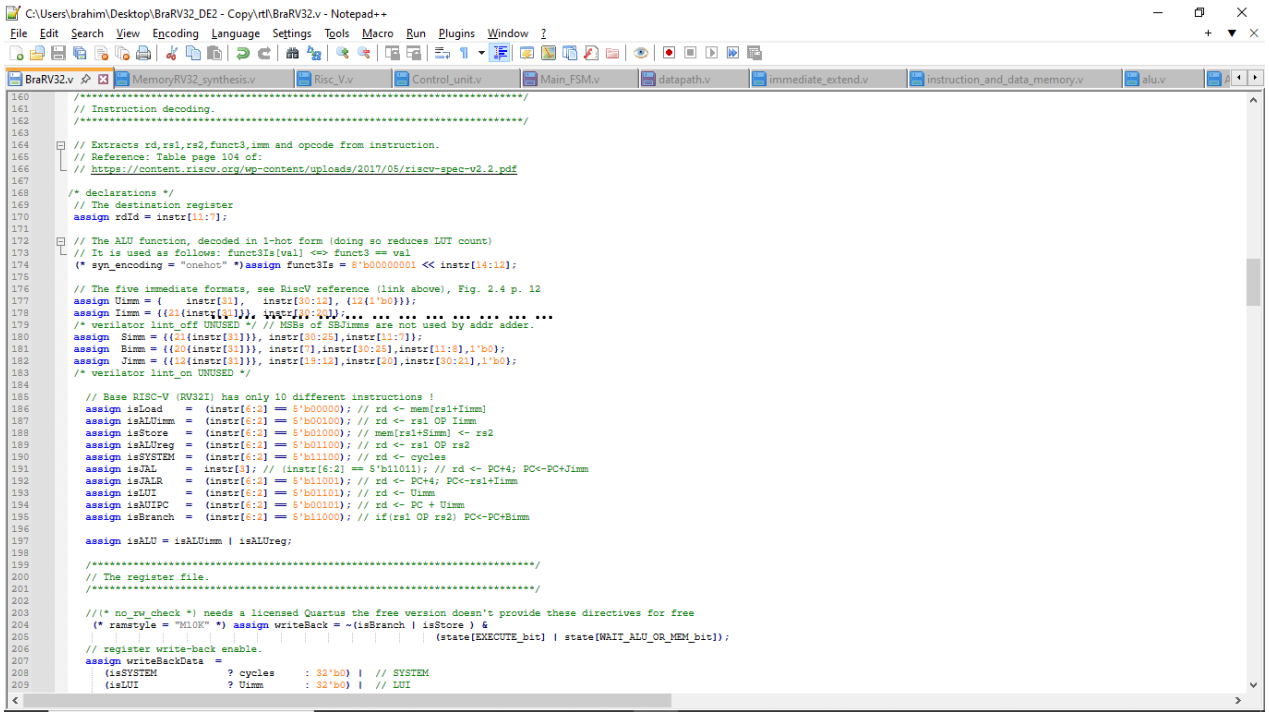


Figure40: BraRV32.v single VERILOG file, compact & understandable code (300 lines of code, 551 lines counting comments)



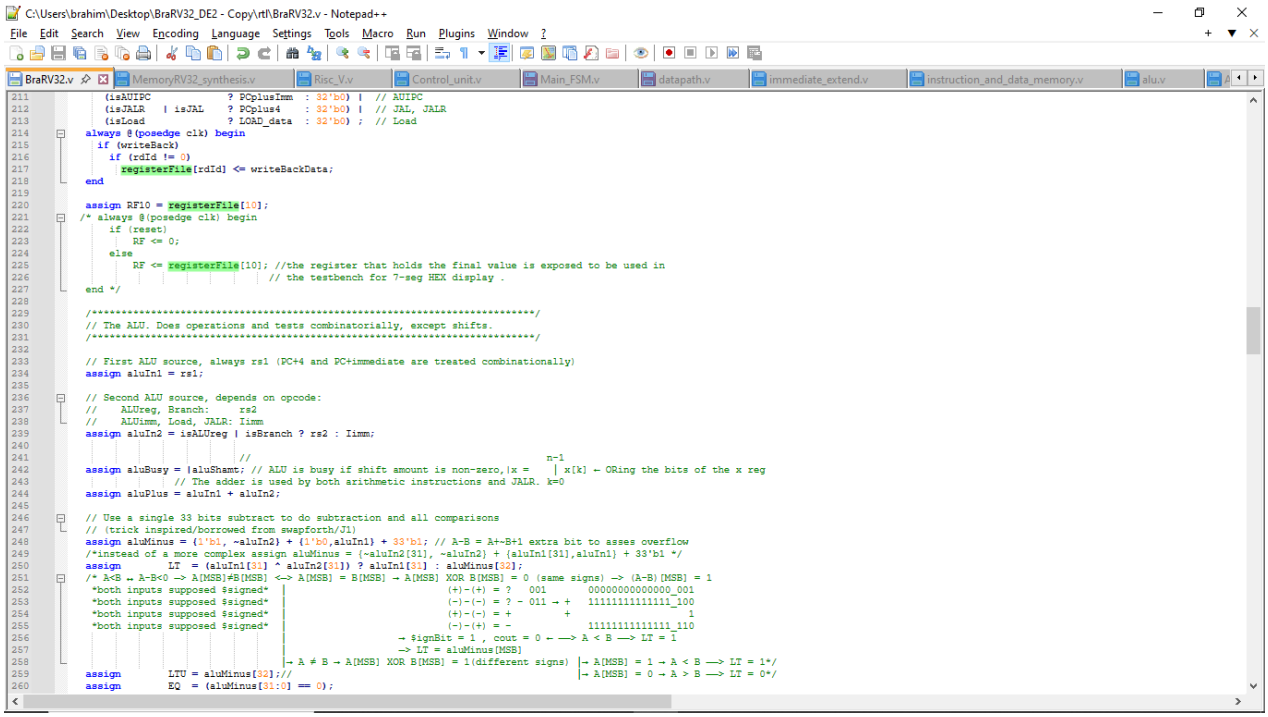


Figure40: …Continued

4.3.4 Minimal Decode Strategy :

- BraRV32 employs a **minimal decode strategy**, where only the essential fields of the instruction are decoded just-in-time and only when needed. Rather than fully decoding all fields up front, it uses lightweight logic to infer instruction types and control signals based on opcode bits and key patterns (e.g., `instr[6:2]`, `funct3`, and `funct7` when relevant).

→ **Benefits of this approach:**

- **Reduced Logic Footprint**: Fewer decoders and no large decode tables lead to lower area usage.
- **Faster Critical Path**: Avoids long combinational chains often introduced in full decode stages.
- **Simpler Control Flow**: By ***selectively decoding***, control logic remains tight and avoids unnecessary signal propagation.

- Compared to classic designs that perform full instruction decoding in a dedicated stage, introducing more logic gates and signal propagation delays.

- BraRV32's minimalist decoding choice favors simplicity and efficiency, aligning with its compact single-FSM design philosophy.

- how it's implemented for this core :

BraRV32

```
/*****************************************************************************/
// Instruction decoding.

/*****************************************************************************/

 // Extracts rd,rs1,rs2,funct3,imm and opcode from instruction.
 // Reference: Table page 104 of:
 // https://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf

/* declarations */
 // The destination register
 assign rdId = instr[11:7];

 // The ALU function, decoded in 1-hot form (doing so reduces LUT count)
 // It is used as follows: funct3Is[val] <=> funct3 == val
 (* syn_encoding = "onehot" *)assign funct3Is = 8'b00000001 <<
instr[14:12];

 // The five immediate formats, see RiscV reference (link above), Fig. 2.4 p.
12
 assign Uimm = {    instr[31],   instr[30:12], {12{1'b0}}};
 assign Iimm = {{21{instr[31]}}, instr[30:20]};
 /* verilator lint_off UNUSED */ // MSBs of SBJimms are not used by addr
adder.
 assign  Simm = {{21{instr[31]}}, instr[30:25],instr[11:7]};
 assign  Bimm = {{20{instr[31]}},
instr[7],instr[30:25],instr[11:8],1'b0};
 assign  Jimm = {{12{instr[31]}}, instr[19:12],instr[20],instr[30:21],1'b0};
 /* verilator lint_on UNUSED */

   // Base RISC-V (RV32I) has only 10 different instructions !
   assign isLoad    =  (instr[6:2] == 5'b00000); // rd <- mem[rs1+Iimm]
   assign isALUimm  =  (instr[6:2] == 5'b00100); // rd <- rs1 OP Iimm
   assign isStore   =  (instr[6:2] == 5'b01000); // mem[rs1+Simm] <- rs2
   assign isALUreg  =  (instr[6:2] == 5'b01100); // rd <- rs1 OP rs2
   assign isSYSTEM  =  (instr[6:2] == 5'b11100); // rd <- cycles
   assign isJAL     =  instr[3]; // (instr[6:2] == 5'b11011); // rd <-
PC+4; PC<-PC+Jimm
   assign isJALR    =  (instr[6:2] == 5'b11001); // rd <- PC+4; PC<-
rs1+Iimm
   assign isLUI     =  (instr[6:2] == 5'b01101); // rd <- Uimm
   assign isAUIPC   =  (instr[6:2] == 5'b00101); // rd <- PC + Uimm
   assign isBranch  =  (instr[6:2] == 5'b11000); // if(rs1 OP rs2) PC<-
PC+Bimm

   assign isALU = isALUimm | isALUreg;
------------------------------------------------------------------------------
/*****************************************************************************/
```

```
   // The predicate for conditional branches.
/*****************************************************************************/

   assign predicate =
        funct3Is[0] &  EQ  | // BEQ
        funct3Is[1] & !EQ  | // BNE
        funct3Is[4] &  LT  | // BLT
        funct3Is[5] & !LT  | // BGE
        funct3Is[6] &  LTU | // BLTU
        funct3Is[7] & !LTU ; // BGEU
-------------------------------------------------------------------------------
   assign funct3IsShift = funct3Is[1] | funct3Is[5];
   assign aluOut =
     (funct3Is[0]  ? instr[30] & instr[5] ? aluMinus[31:0] : aluPlus
                   :32'b0)|
     (funct3Is[2]  ? {31'b0, LT}                           :32'b0)|
     (funct3Is[3]  ? {31'b0, LTU}                          :32'b0)|
     (funct3Is[4]  ? aluIn1 ^ aluIn2                       :32'b0)|
     (funct3Is[6]  ? aluIn1 | aluIn2                       :32'b0)|
     (funct3Is[7]  ? aluIn1 & aluIn2                       :32'b0)|
     (funct3IsShift ? aluReg                               :32'b0);
-------------------------------------------------------------------------------
   assign PCplusImm = PC + ( instr[3] ? Jimm[ADDR_WIDTH-1:0] :
 instr[4] ? Uimm[ADDR_WIDTH-1:0] :Bimm[ADDR_WIDTH-1:0] );
   assign loadstore_addr = rs1[ADDR_WIDTH-1:0] +
               (instr[5] ? Simm[ADDR_WIDTH-1:0] : Iimm[ADDR_WIDTH-1:0]);
-------------------------------------------------------------------------------
   assign mem_byteAccess     = instr[13:12] == 2'b00; // funct3[1:0] ==
2'b00;
   assign mem_halfwordAccess = instr[13:12] == 2'b01; // funct3[1:0] ==
2'b01;
```

- for the classic multicycle :

Risc_V

```
module Control_unit(
    input clk, reset,run,zero,overflow,sign,cout,funct7,
    input [2:0] funct3,
    input [6:0] op,

     output  PCWrite,done,
    output  Mem_Write, IRWrite, RegWrite, AdrSrc,
    output  [1:0] ALUSrcA, ALUSrcB,  ResultSrc,
     output  [3:0] ALUControl ,
     output  [3:0] state ,
    output [2:0] ImmSrc , size
```

```
);
wire    [1:0] ALUOp ;
wire  PCUpdate,Branch,sel_size;

//Module de la FSM
Main_FSM Fsm (.done(done) ,.run(run) ,.clk(clk) , .reset(reset), .op(op) ,
.PCUpdate(PCUpdate) , .Branch(Branch) , .Mem_Write(Mem_Write) ,
.IRWrite(IRWrite) ,
             .RegWrite(RegWrite) , .AdrSrc(AdrSrc) ,
.sel_size(sel_size) ,.ALUSrcA(ALUSrcA) , .ALUSrcB(ALUSrcB) ,
.ResultSrc(ResultSrc) ,
                  .st(state), .ALUOp(ALUOp)) ;

//Module du choix d'operation
ALU_decoder alu_dec(.funct7_5(funct7) ,.ALUop(ALUOp) , .funct3(funct3) ,
.ALU_Control(ALUControl));

//Module du choix de extend immediate
Inst_decoder Imm_choice(.op(op), .ImmSrc(ImmSrc));
--------------------------------------------------------------------------
module Main_FSM(
    input clk, reset,run,
    input [6:0] op,

    output reg PCUpdate,Branch, Mem_Write, IRWrite, RegWrite,
AdrSrc,sel_size,done,
    output reg  [1:0] ALUSrcA, ALUSrcB, ResultSrc,
     output  reg [3:0] st,
    output  reg [1:0] ALUOp


);
// FSM States
parameter
    Fetch     = 4'b0000,
    Decode    = 4'b0001,
    MemAdr    = 4'b0010,
    MemRead   = 4'b0011,
    MemWB     = 4'b0100,
    MemWrite  = 4'b0101,
    ExecuteR  = 4'b0110,
    ALUWB_s   = 4'b0111,
    BEQ_s     = 4'b1000,
    JAL_s     = 4'b1001,
    ExecuteI  = 4'b1010,
    LUI_s     = 4'b1011,
    Auipc     = 4'b1100,
    ExecuteRC = 4'b1101,
    idle      = 4'b1111,
    ExecuteIC = 4'b1110;
--------------------------------------------------------------------------
```

```
odule Inst_decoder(
     input [6:0] op,


    output reg [2:0] ImmSrc
);

always @(*) begin
    case (op)
        //type I
        7'b0000011,
        7'b0010011,
        7'b0010001,
        7'b0110011,
        7'b1100111 : ImmSrc <= 3'b000;
        //type S
        7'b0100011 : ImmSrc <= 3'b001;
        //type B
        7'b1100011 : ImmSrc <= 3'b010;
        //type J
        7'b1101111 : ImmSrc <= 3'b011;
        //type U
        7'b0010111,
        7'b0110111 : ImmSrc <= 3'b100;
        default    : ImmSrc <= 3'bzzz;

    endcase
end
endmodule
--------------------------------------------------------------------------------
module ALU_decoder(
     input funct7_5,
    input [1:0] ALUop,
    input [2:0] funct3,


    output reg [3:0] ALU_Control
);


always @(*) begin
    case (ALUop)
        2'b00 : ALU_Control <= 4'b0000; //somme
        2'b01 : ALU_Control <= 4'b0001; //difference
        2'b10 : begin
                    case (funct3)
                       3'b000  : ALU_Control  <= {{3 {1'b0}},funct7_5
};          //SOMME(0000) OU DIFFERENCE(0001)
                       3'b001  : ALU_Control  <= 4'b0010;
.. … .. … .. … .. … .. … .. … ... … .. … .. … .. … .. … .. … .. … .. … ..
```

- Unlike traditional pipelines that rely on multiple layers of decoders generating a wide set of control signals, many of which are only relevant to specific instruction types.

- BraRV32 keeps things simple. It uses direct combinational checks on `funct3` and a few key `instr` bit fields to **generate exactly the control signals it needs**, **only when it needs them**. This avoids unnecessary complexity and control overhead, making the design more efficient and compact.

4.4 Simulation and Verification :

- To validate the functional correctness and behavior of the BraRV32 core, a structured simulation was carried out using **ModelSim**. This focused on verifying instruction decoding, control signal generation, state transitions, and memory interactions under realistic test conditions.

- The simulation is performed on the device under test using a recursive Fibonacci assembly program that calculates the 15th Fibonacci number.

- The Fibonacci sequence :

$$\forall n \epsilon \mathbb{N} \setminus F_{n+2} = F_{n+1} + F_n \qquad (1)$$
$$F_0 = 0 \, , F_1 = 1$$

- The code :

```
main:
    addi a2, zero, 15          # a2 = 15 (argument to fib)
    jal ra, fib                # call fib(a2 = 31)
    jalr zero, ra, 0           # return from main

fib:
    addi sp, sp, -12           # allocate 12 bytes on stack
    sw ra, 8(sp)               # save return address
    sw s0, 4(sp)               # save s0

fib_if_x0_or_one:
    addi a0, zero, 0           # a0 = 0
    beq a2, zero, fib_fin      # if a2 == 0, return
    addi a0, a0, 1             # a0 = 1
    beq a2, a0, fib_fin        # if a2 == 1, return

fib_call_n_1:
    addi a2, a2, -1            # a2 = a2 - 1
    sw a2, 0(sp)               # save current a2
    jal ra, fib                # recursive call fib(n-1)
    lw a2, 0(sp)               # restore a2

    addi a2, a2, -1            # a2 = a2 - 2
                               #(already -1,now -2)
```

n	Fibonacci (n)
0	0
1	1
2	1
3	2
4	3
5	5
6	8
7	13
8	21
9	34
10	55
11	89
12	144
13	233
14	377
15	**610**
16	987
17	1597
18	2584
19	4181
20	6765
21	10 946
22	17 711
23	28 657
24	46 368
25	75 025
26	121 393
27	196 418
28	317 811
29	514 229
30	832 040
31	1 346 269

```
fib_call_n_2:
    add s0, a0, zero          # s0 = fib(n - 1)
    jal ra, fib               # recursive call fib(n - 2)
    add a0, a0, s0            # a0 = fib(n - 1) + fib(n - 2)

fib_fin:
    lw s0, 4(sp)              # restore s0
    lw ra, 8(sp)              # restore return address
    addi sp, sp, 12           # deallocate stack
    jalr zero, ra, 0          # return
```

• Recursive Fibonacci sequence implementation in RISC-V assembly •

- Before we dive into the methodology, approach and finally the results of the simulation it is necessary to take a look at the type of memory used it's Verilog implementation and how it ties to the BraRV32 core

```
module MemoryRV32 #(
    parameter MEM_WORDS = 1024
)(
    input wire         clk,
    input wire [31:0] mem_addr,
    input wire [31:0] mem_wdata,
    input wire  [3:0] mem_wmask,
    output reg [31:0] mem_rdata,
    input wire         mem_rstrb,
    output wire        mem_rbusy,
    input wire         reset
);

    reg [31:0] memory [0:MEM_WORDS-1];

    assign mem_rbusy = 0; // always ready for read in this model

    // Read
    always @(posedge clk) begin
        if (mem_rstrb) begin
            mem_rdata <= memory[mem_addr[31:2]];
        end
    end
    integer i;
    // Write
    always @(posedge clk) begin
        if (|mem_wmask) begin

            for (i = 0; i < 4; i = i + 1)
                if (mem_wmask[i])
                    memory[mem_addr[31:2]][8*i +: 8] <= mem_wdata[8*i +: 8];
        end
    end
```

```
integer fd;
    integer j; // This will hold the label address (like 100, 300)
    reg [31*8:0] label; // Up to 31-char label name


    function integer strcmp;
        input [8*(32-1):0] str1; // 32-character string
        input [8*(32-1):0] str2;
        integer i;
        begin
                strcmp = 0; // assume equal
                for (i = 0; i < 32; i = i + 1) begin
                    if (str1[8*i +: 8] != str2[8*i +: 8]) begin
                        strcmp = 1; // not equal
                    end
                end
        end
    endfunction



function integer strcmp_trimmed;
  input [8*32-1:0] str1;
  input [8*32-1:0] str2;
  integer i, start1, start2;
  reg [7:0] c1, c2;
  reg done; // flag to stop loops early
  begin
    strcmp_trimmed = 0;

  start1 = 0;
  start2 = 0;
  done = 0;

  // Find first non-space in str1
  for (i = 0; i < 32 && !done; i = i + 1) begin
    if (str1[8*i +: 8] != " ") begin
      start1 = i;
      done = 1; // stop loop next iteration
    end
  end

  done = 0;
  // Find first non-space in str2
  for (i = 0; i < 32 && !done; i = i + 1) begin
    if (str2[8*i +: 8] != " ") begin
      start2 = i;
      done = 1; // stop loop next iteration
    end
  end

  done = 0;
  // Compare character by character
  for (i = 0; i < 32 - ((start1 > start2) ? start1 : start2) && !done; i = i
+ 1) begin
    c1 = str1[8*(i + start1) +: 8];
    c2 = str2[8*(i + start2) +: 8];
```

```
    if ((c1 == 0 || c1 == " ") && (c2 == 0 || c2 == " ")) begin
            done = 1; // end comparison loop
          end
          else if (c1 != c2) begin
            strcmp_trimmed = 1; // not equal
            done = 1; // end comparison loop
          end
        end
      end
    endfunction

    initial begin
      fd = $fopen("../labels.txt", "r");
      if (fd) begin
        i = 0;
        while (!$feof(fd)) begin
          if ($fscanf(fd, "%s %d\n", label, j) == 2) begin
            $display("Line %0d: Label = %s, Address = %0d", i, label, j);

            if (strcmp_trimmed(label, "main") == 0) begin
                $display("ana f main");
                $readmemb("../rtl/main.tv", memory,j);
            end

            if (strcmp_trimmed(label, "fib") == 0) begin
                $display("ana f fib");
                $readmemb("../rtl/fib.tv", memory,j);
            end

            i = i + 1;
          end else begin
            $display("Format error at line %0d", i);
          end
        end
        $fclose(fd);
      end else begin
        $display("Failed to open file");
      end
    end



    // Load memory from file [hardcoded]
    initial begin
        //$readmemb("../rtl/main.tv", memory,);
        //$readmemb("../rtl/fib.tv", memory,40);
        /*memory[0] = 32'h00000293; // li t0,0
        memory[1] = 32'h00100313; // li t1,1
        memory[2] = 32'h0062A023; // sw t1,0(t0)
        //memory[3] = 32'h0000006F; // j to self (infinite loop)
        memory[3] = 32'h000010EF; //jal rd(=r1),0x0000 0001
        memory[4] = 32'h00055380; // jal to somwhere (infinite loop) rd<-
PC + 4
        // Fill unused memory with NOPs (0x00000013)
        for (i = 4; i < 1024; i = i + 1) begin
            memory[i] = 32'h00000013;
        end*/
```

```
    end

endmodule
```

- To support instruction and data memory operations in the BraRV32 processor, a unified memory model `MemoryRV32` was implemented (**Figure33**). it serves as a simple synchronous RAM with both read and byte-masked write support. It also includes a flexible initialization mechanism to preload test programs at specific addresses during simulation.

♦ **Features :**

- **Parameterized Memory Size:**
  The memory consists of `MEM_WORDS` 32-bit words (default: 1024), allowing scalable capacity for program and data storage.
- **Synchronous Read Operation:**
  On the rising edge of the clock and when `mem_rstrb` is asserted, a 32-bit word is read from the address `mem_addr[31:2]`. Word alignment is achieved by discarding the two least significant bits.
- **Masked Write Support:**
  The write port supports partial updates through `mem_wmask`, where each bit enables writing of one byte in the target word. This simulates byte-granular memory writes (`sb`, `sh`, `sw`) and is useful for RISC-V's `mem_wmask` behavior.
- **Always-Ready Read Interface:**
  The `mem_rbusy` signal is hardwired to 0, indicating memory is always ready for reads in this simplified single-cycle memory model.

♦ **Simulation-Specific Features :**

- **Test Program Preloading via Label File:**
  During simulation, the memory is initialized based on external `.tv` files (text vector files) using labels defined in a `labels.txt` file, generated by a python script before assembling (more on that later). Each line in the file associates a label (e.g., `"main"`, `"fib"`) with a memory address. When a label matches, the corresponding `.tv` file is loaded into the `memory[]` array starting at that address.
- **Label Matching with Trimming Logic:**
  To avoid mismatches due to whitespace or padding, a custom `strcmp_trimmed` function compares strings after removing leading spaces. This ensures robust label detection even if the input format varies slightly.
- **Informative Simulation Output:**
  Messages are printed during memory initialization to confirm label matches and memory loading actions, which aids in debugging and traceability during waveform analysis.

→ **Summary:**

- The `MemoryRV32` module combines functionality and flexibility tailored for simulation and verification of a RISC-V processor. It provides a reliable abstraction for both instruction and data memory, while supporting structured and label-based test program loading .

♦ Simulation Methodology:

- **Testbench Structure:**
  A dedicated Verilog testbench was written to instantiate the BraRV32 core, provide initial signal setup (clock, reset), and drive memory-mapped instruction and data stimuli.
- **Clock and Reset Control:**
  A controlled clock with a defined frequency was used, alongside a single reset pulse to ensure deterministic FSM startup and state initialization.
- **Program Initialization:**
  RISC-V assembly test cases were written, assembled using ~~an external toolchain~~ a pseudo RISC-V assembler made with python for the scope of this project which we will discuss in later chapters, after that we load it into the instruction memory in `.tv` format (`.hex` won't be read by verilog's read memory functions $readmemh & $readmemb) .
- **Instruction Coverage:**
  The simulation suite tested all key instruction types:
  - **Arithmetic & Logic:** `add`, `sub`, `xor`, `slli`
  - **Load/Store:** `lw`, `lh`, `lhu`, `lb`, `sw`
  - **Control Transfer:** `beq`, `jal`, `jalr`
  - **Immediate/Upper:** `lui`, `auipc`, `addi`

♦ Verification Objectives:

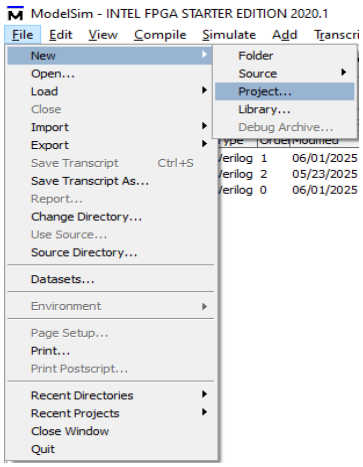
1. **FSM Behavior Validation:**
   Observed and traced one-hot state transitions through `FETCH`, `EXECUTE`, `WAIT_ALU_OR_MEM`, and `WRITE_BACK` states. Each instruction followed the correct logical sequence.
2. **Control Signal Timing:**
   Ensured signals like `mem_rstrb`, `writeBack`, `isLoad`, and `isBranch` toggled as expected based on instruction type and internal decoding. Special attention was given to load/store alignment and jump handling as it was the most vulnerable for errors and undefined behaviour since **the memory** is **word addressable** and the **address calculations** executed inside BraRV32 core are **byte-addressable** oriented.
3. **Datapath Observability:**
   Tracked ALU outputs, register write-backs (`rdId`), and memory accesses for correctness. Byte and halfword alignment logic for `LOAD_data` and `STORE_mask` were verified over multiple unaligned accesses.
4. **End-to-End Consistency Checks:**
   Register file values and memory contents were compared to reference outputs

post-execution. The Multicycle pipeline maintained correctness even under mixed instruction sequences.

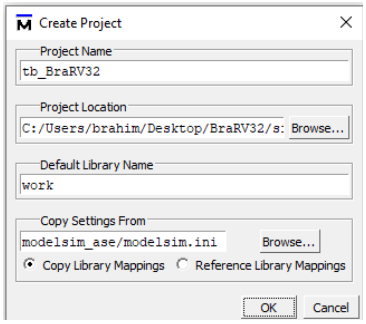
♦ Simulation Tools:

- **Environment:** ModelSim
- **Waveform Analysis:**
  Internal signals were probed using waveform viewer and VCD traces. Timing and data propagation across control and datapath blocks were reviewed over several simulation cycles to ensure effective communication between the two.

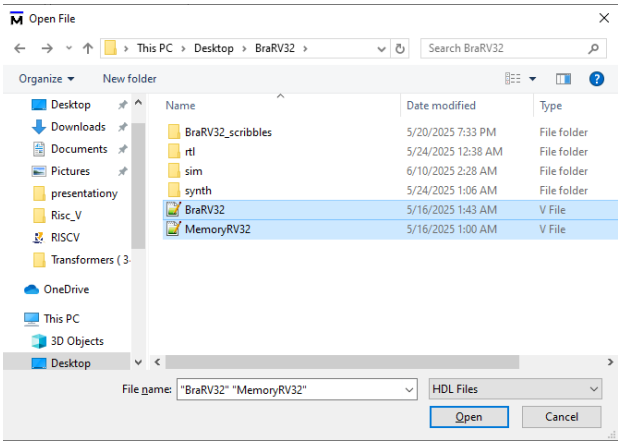
- First we create a project in Modelsim



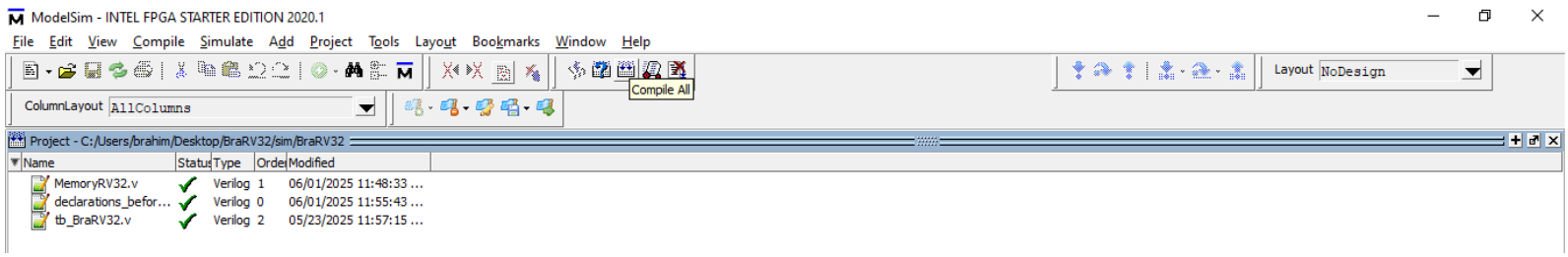
- Give it a name and press OK



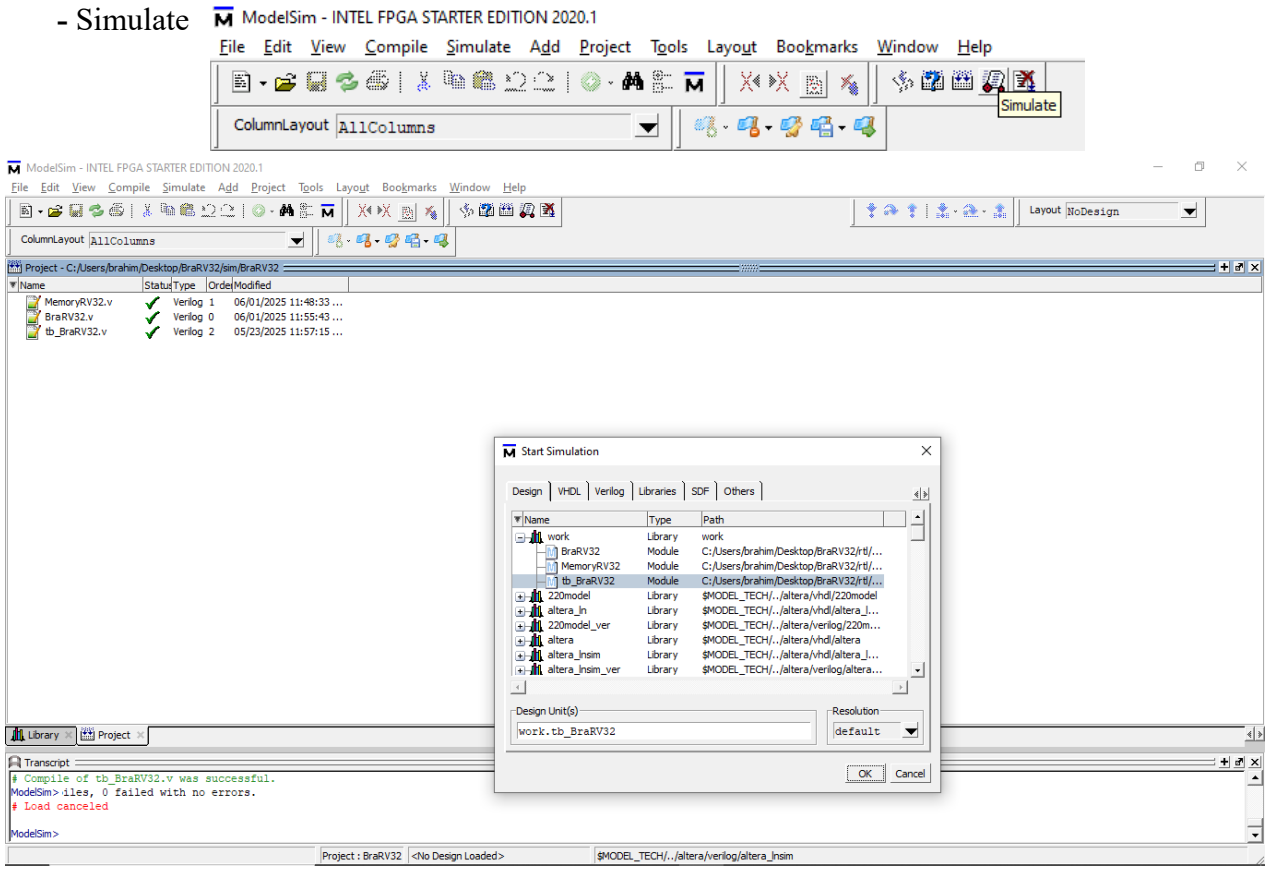
- add the necessary HDL files



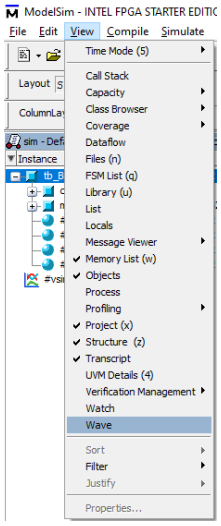
- Compile the design



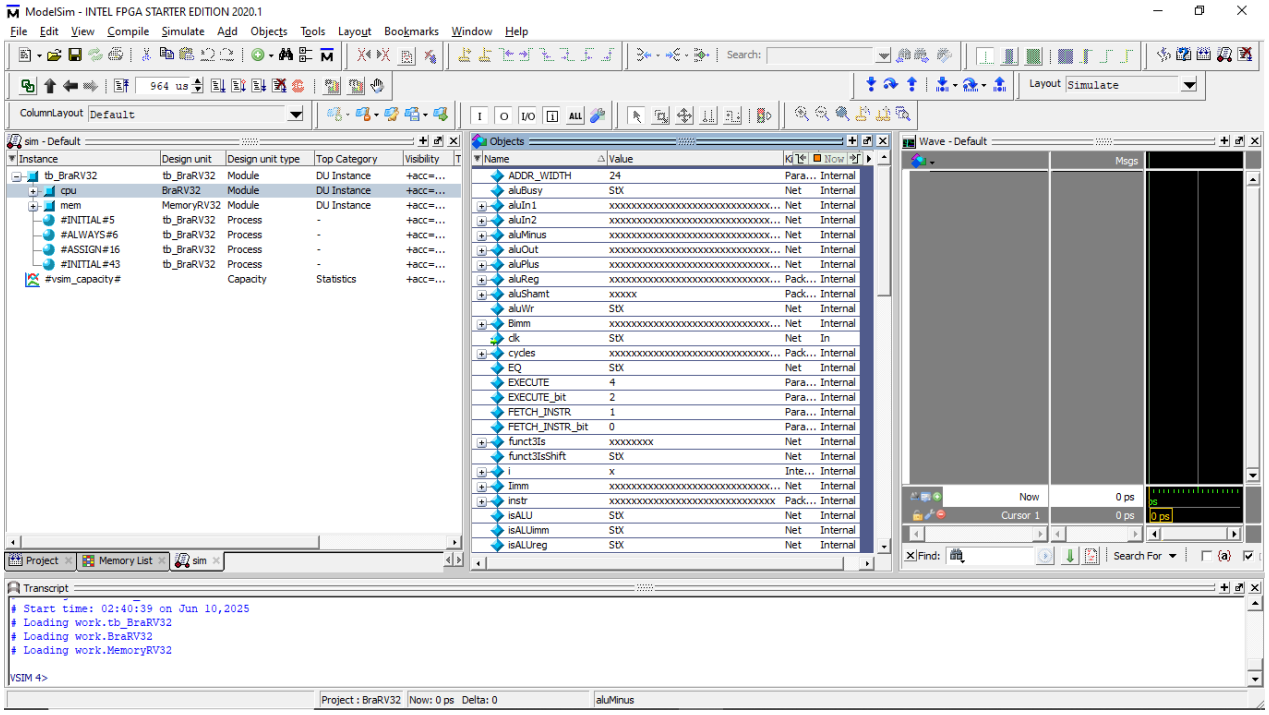
- Simulate



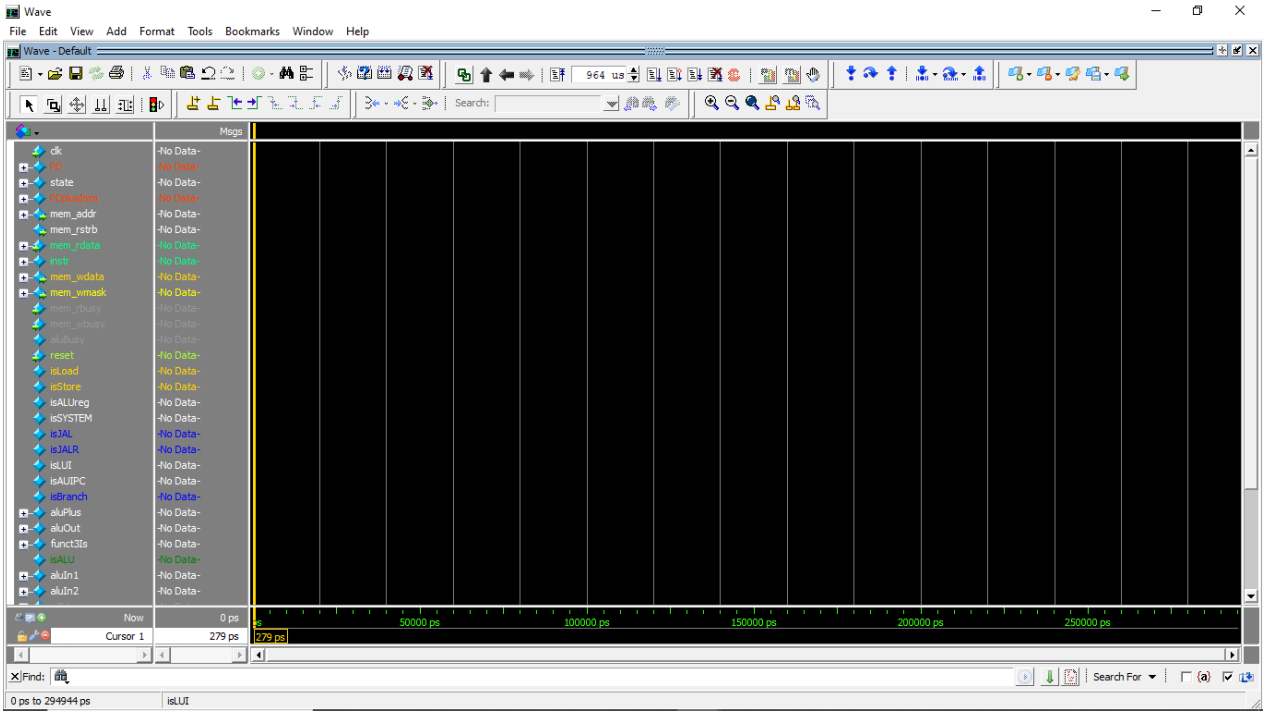
- Add wave



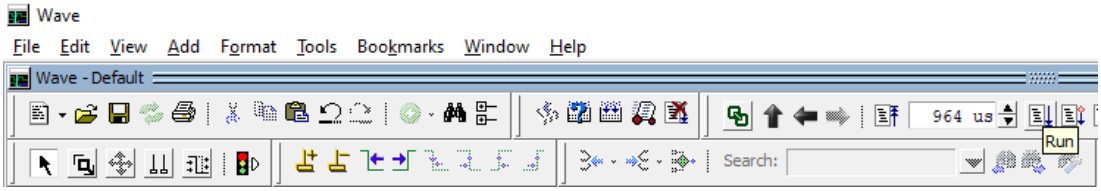
- Drag and drop desired signals into the wave window



- The Waveform window



- Run



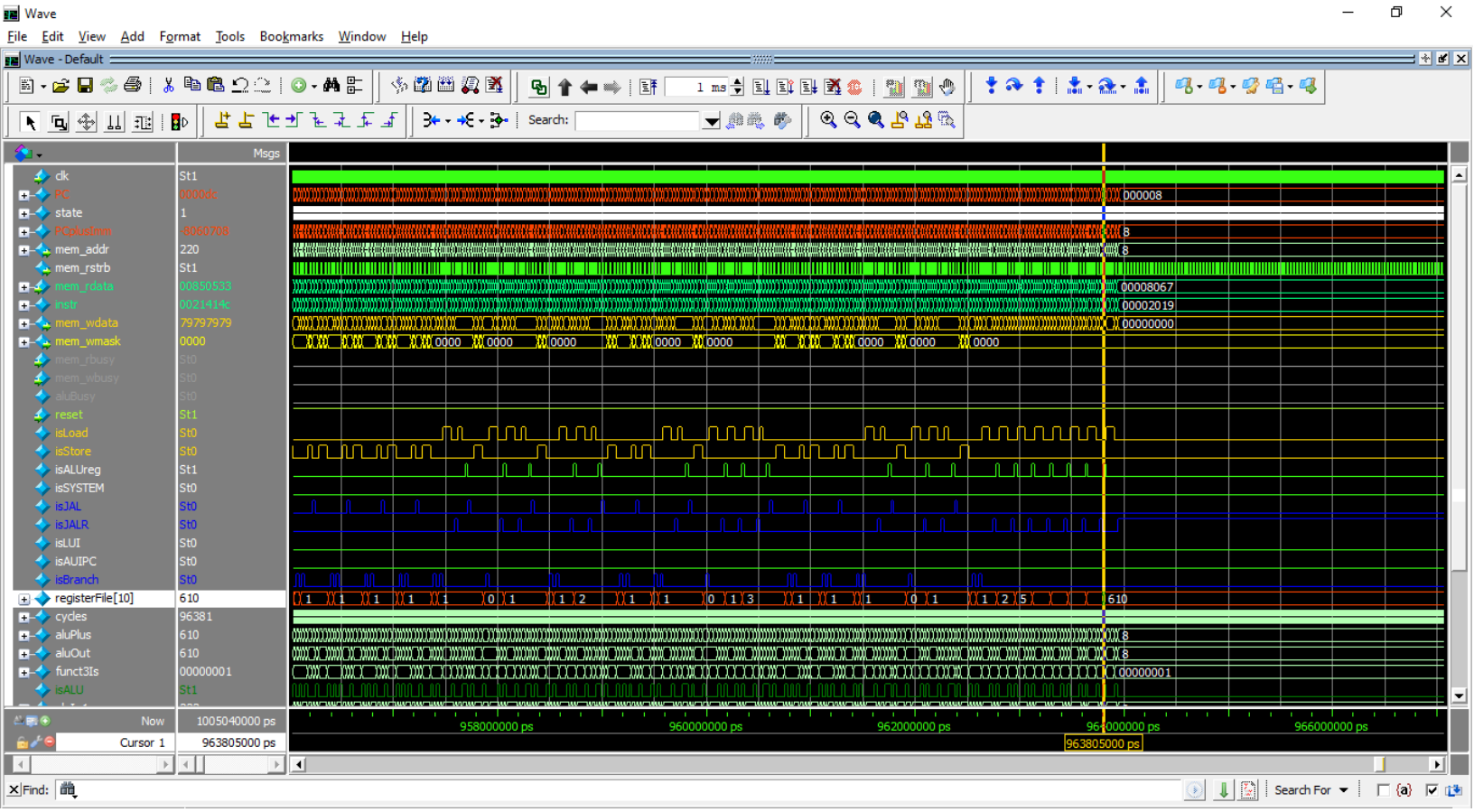


Figure41 : Simulation results of BraRV32 running recursive Fibonacci[15]

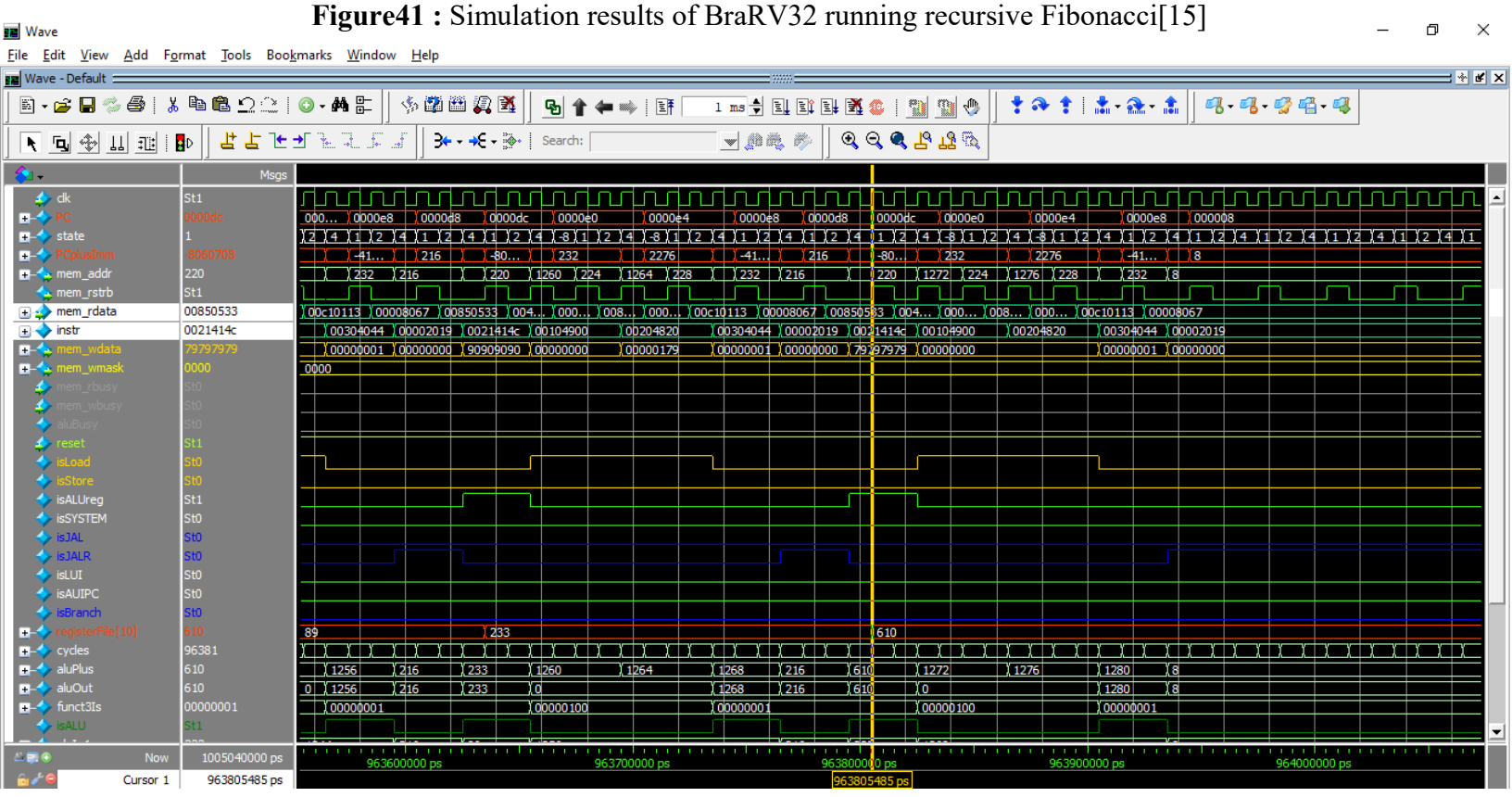


Figure42 : Simulation results of BraRV32 running recursive Fibonacci[15]…***Zoomed in***

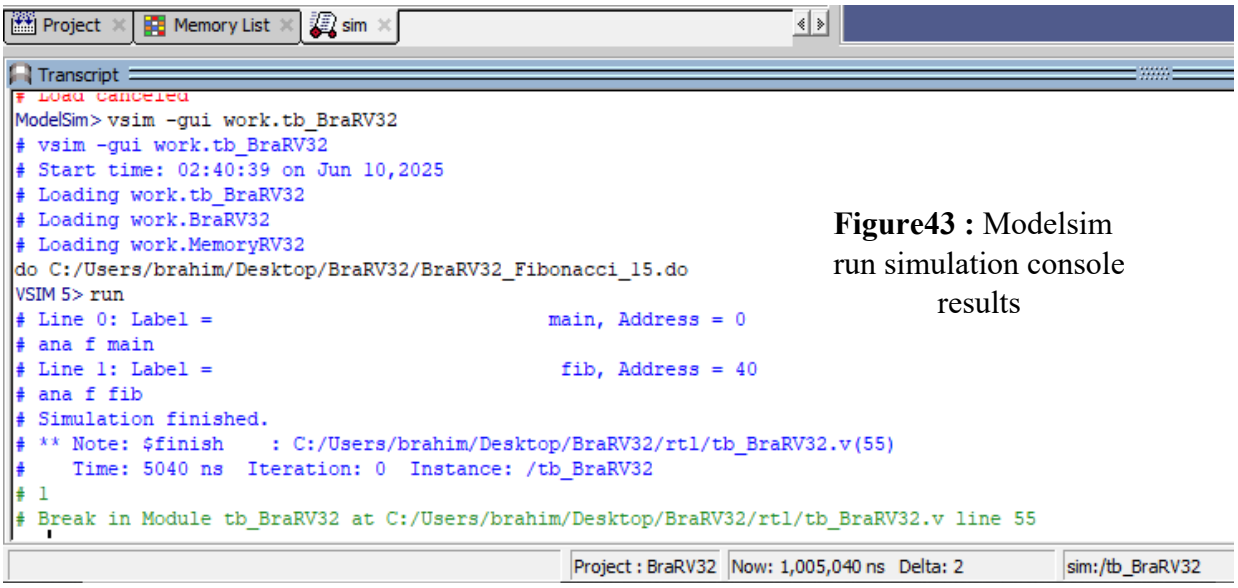


Figure43 : Modelsim run simulation console results

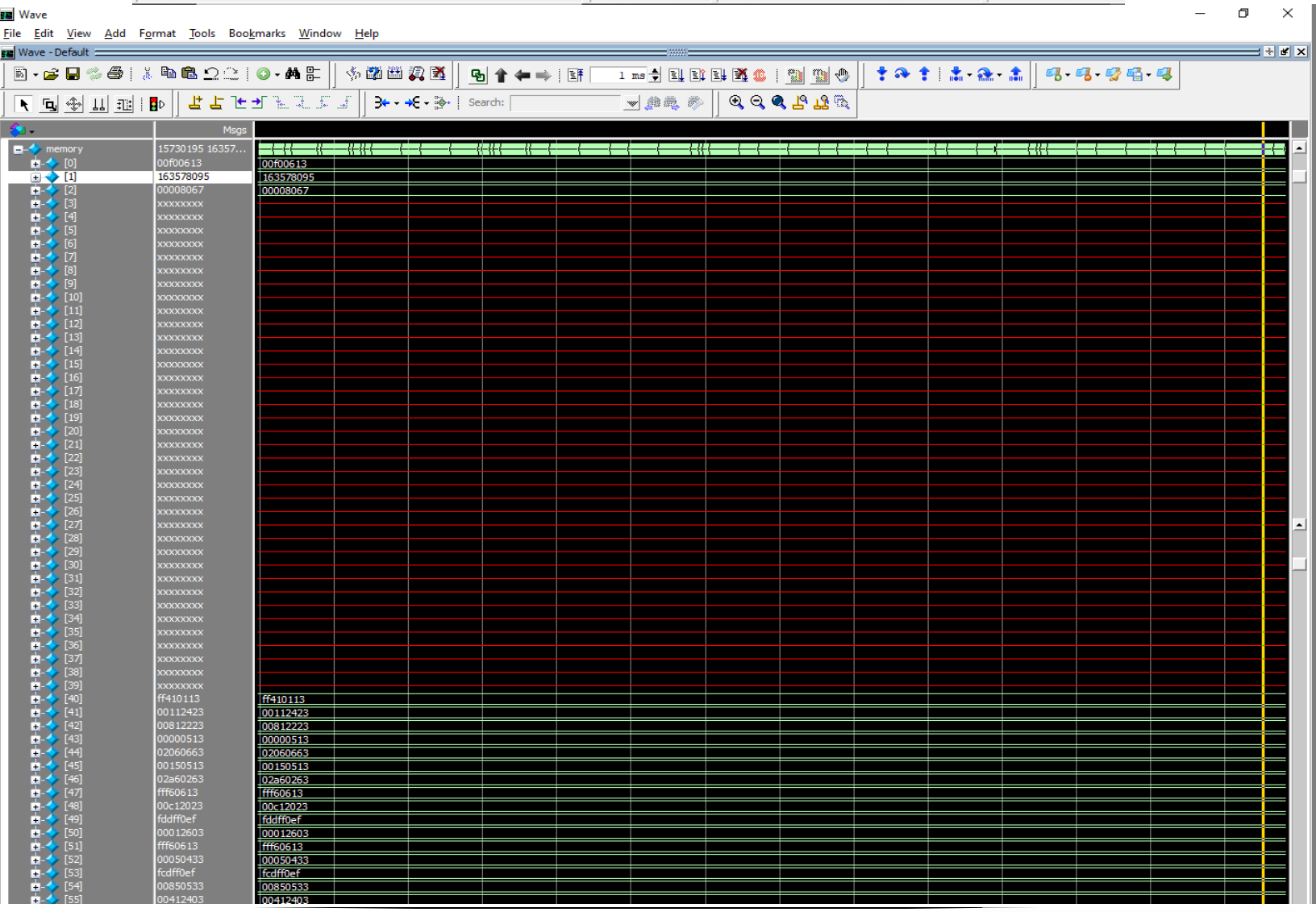
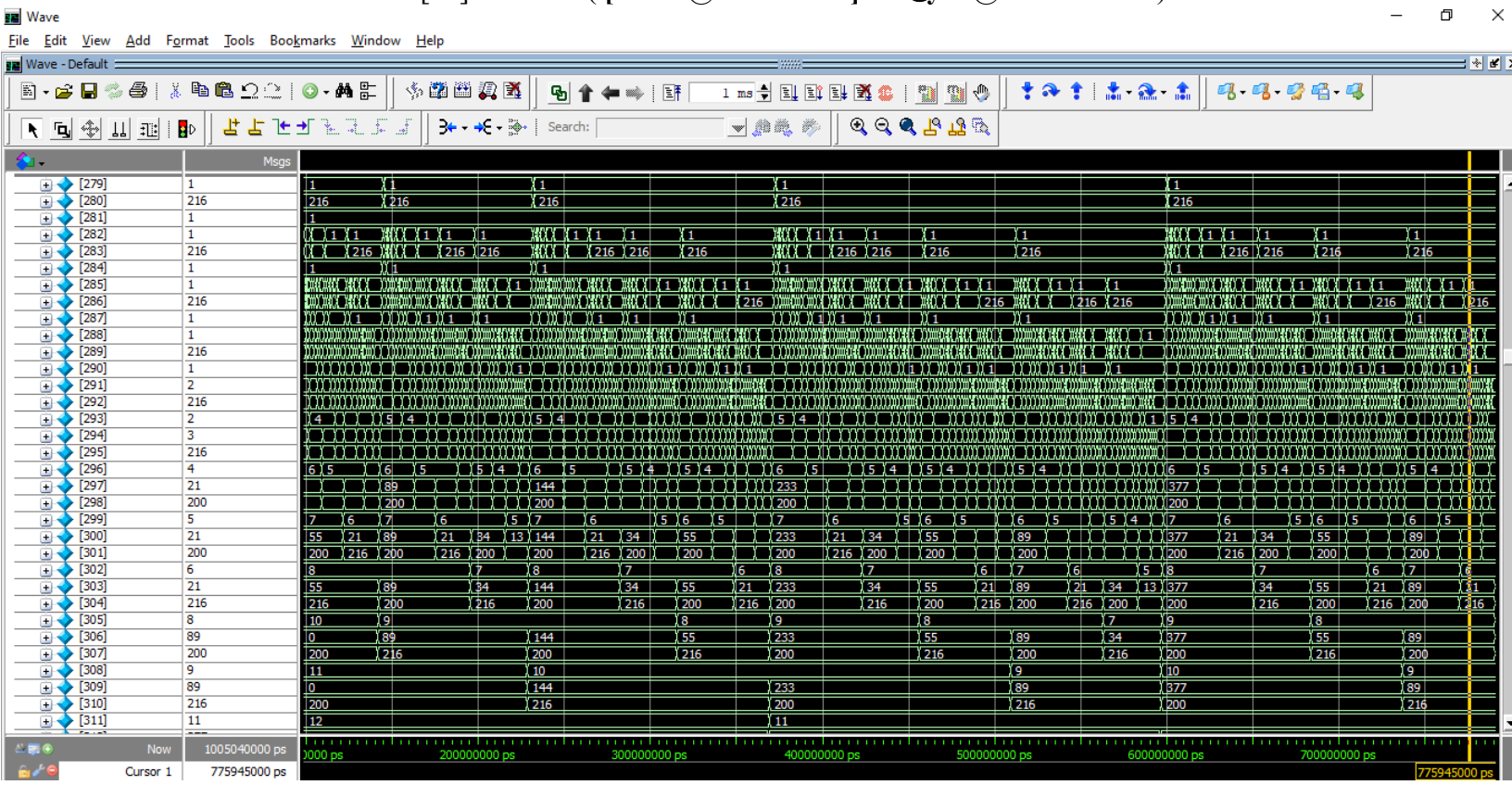


Figure44 : instructions in memory with each label in it's given address main : [0] → 0x00 & fib : [40] → 0xA0 (**[word @ddressable] → Byte @ddressable**)



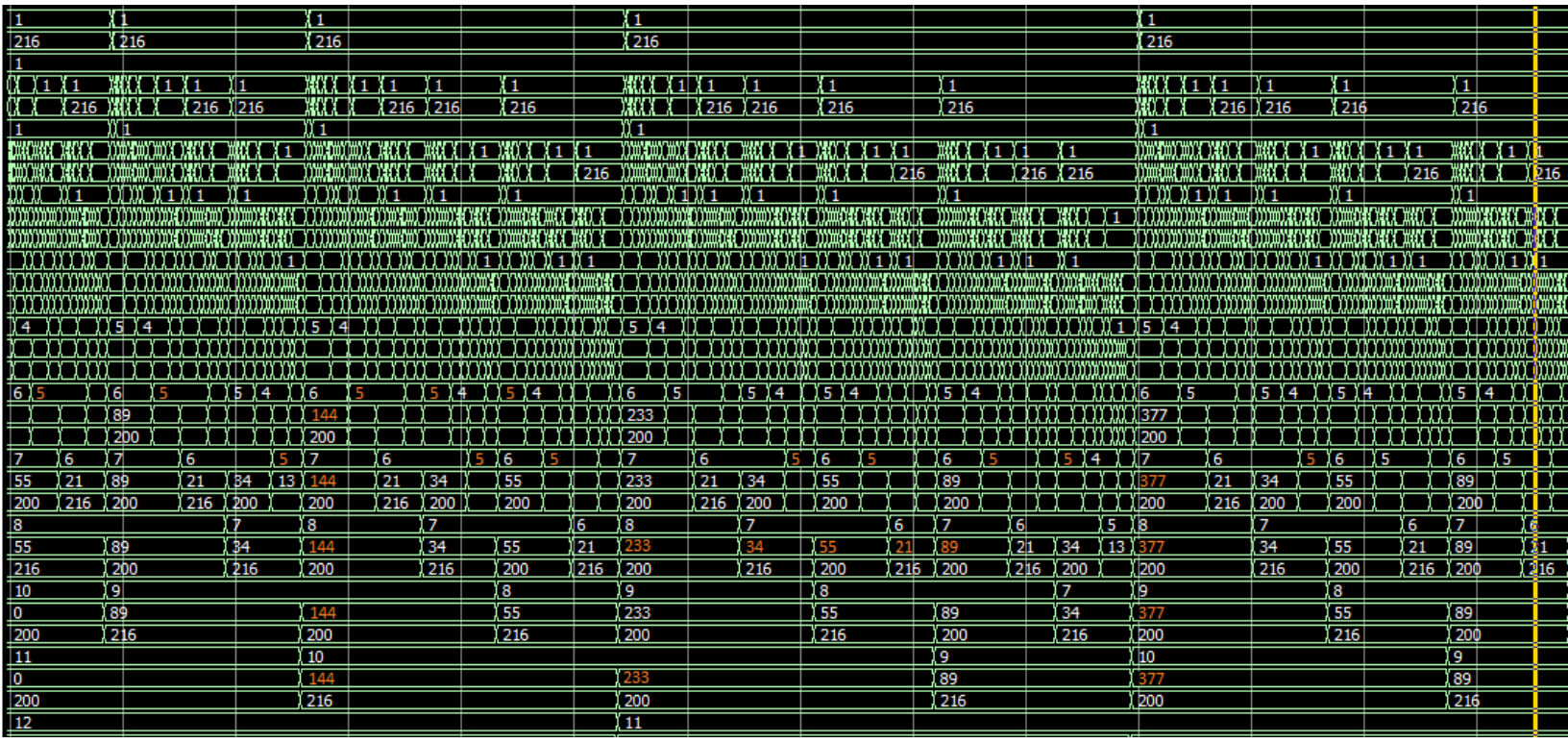


Figure45 : The Stack Frame for Fibonacci [15]

**Note :*

mem_rdata is a 32 bit register [31:0] and instruction is optimized by omitting the last two bits which constitutes the LSB bits of the opcode which are always 2'b11 for all instructions, in verilog this is called bit slicing. hence we should get the following equality $intsr = mem\_rdata \gg 2$.

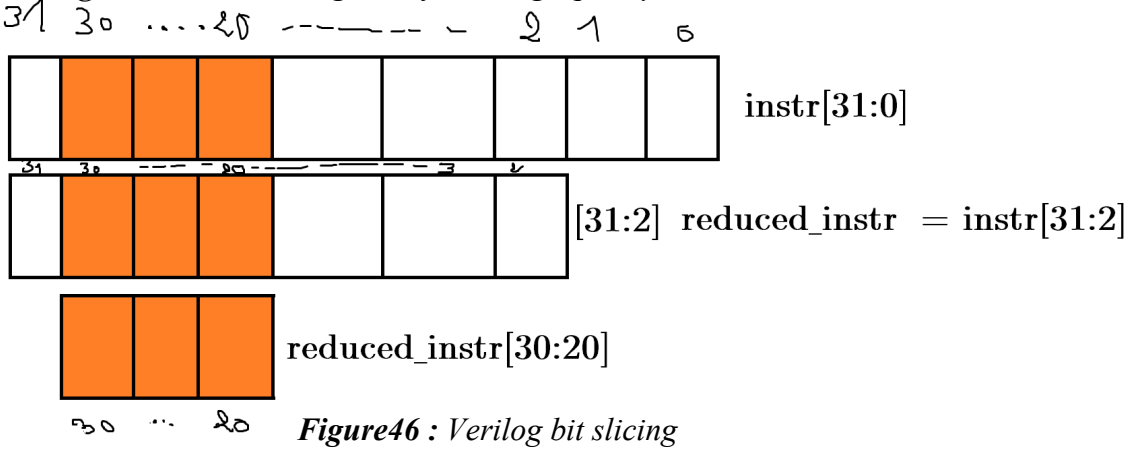


Figure46 : *Verilog bit slicing*

4.5 FPGA Synthesis (Comparative Analysis with Original Core) :

- To evaluate the practical viability of the **BraRV32** core, both the **original reference *implementation*** and the **optimized BraRV32** design were synthesized targeting an Intel Cyclone II FPGA.

♦ Synthesis Observations:

- **FSM Encoding:**
  The BraRV32 design adopts **one-hot FSM encoding**, which is better suited for FPGAs like the Cyclone II as stated before ,due to its simplified decoding logic and direct mapping to flip-flops.
- **Memory Mapping Optimization:**
  The `ramstyle = "M10K"` synthesis attribute was used to guide the Quartus compiler to infer **on-chip M10K memory blocks** for storing register files and data memory. This significantly reduces logic utilization compared to distributed LUT RAM.
- **Area and Timing:**
  Compared to the original multicycle reference, BraRV32 showed **notable reductions in combinational logic usage**, improved critical path delay, and fewer levels of logic due to flattened decode logic and minimal control hierarchy.
- **Synthesis Compatibility:**
  The design was cleanly synthesized under Quartus II for the Cyclone

II family, with no timing violations and full register/resource mapping confirmation.

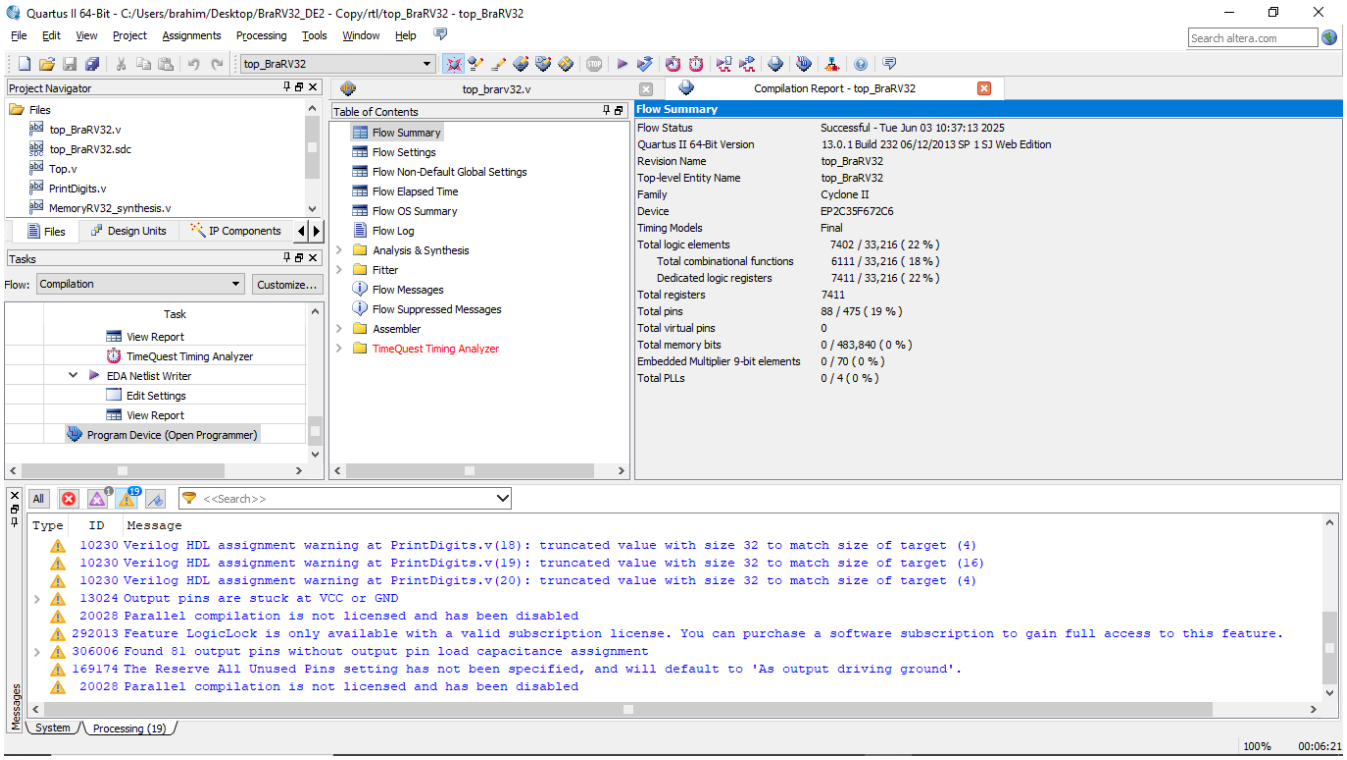


Figure47: Compilation report's Flow summary for the Optimized Multicycle Implementation

-***Timing analysis*** shows stable operation up to **~65 MHz**, with no critical path violations (which can operate comfortably at the internal **DE2 board's 50Mhz** clock)

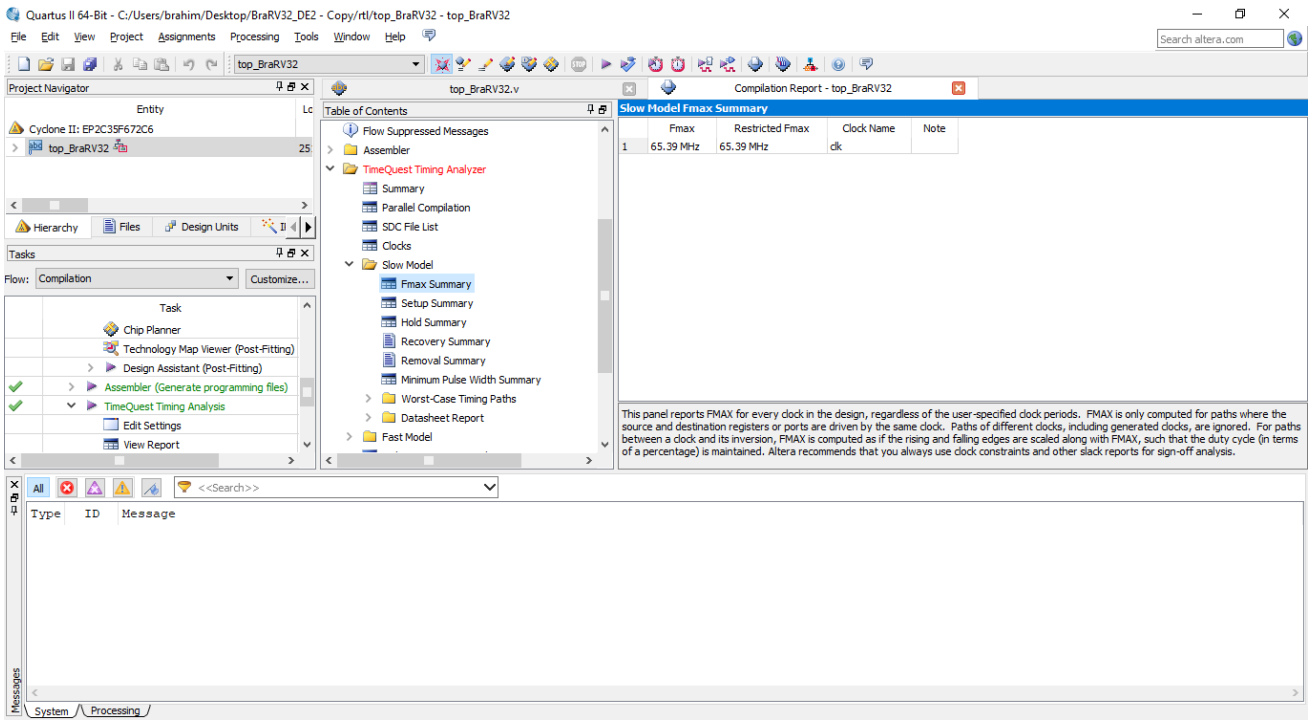


Figure48: Compilation report's Timing Analyzer maximum frequency F_{max}

- ***The RTL Viewer*** validates a **well-organized** and **modular** datapath: **instruction fetch**, **control logic**, **ALU**, **register file**, and **program counter** modules are *distinguishable* even though they're *not explicitly separated* as different modules in the Verilog code. The FSM state transitions are clearly mapped and adhere to the intended multicycle sequencing.

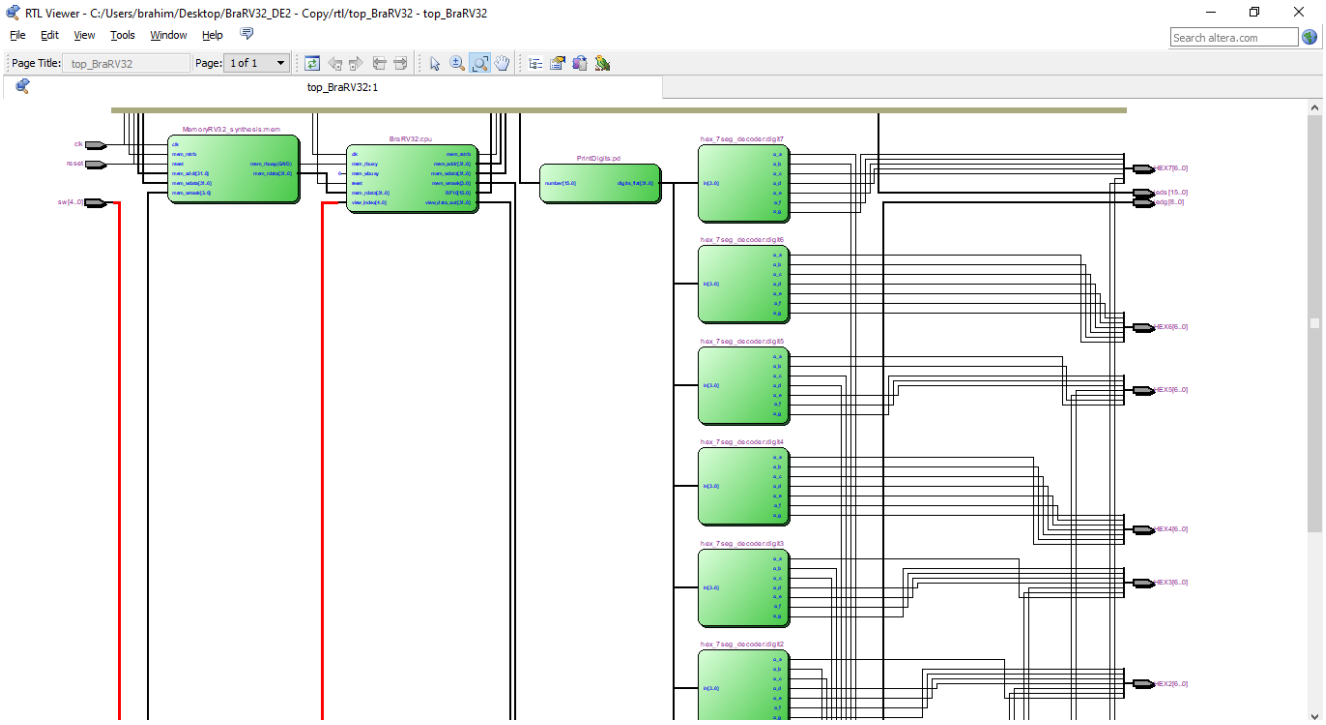


Figure49: Top module (testbench) RTL View

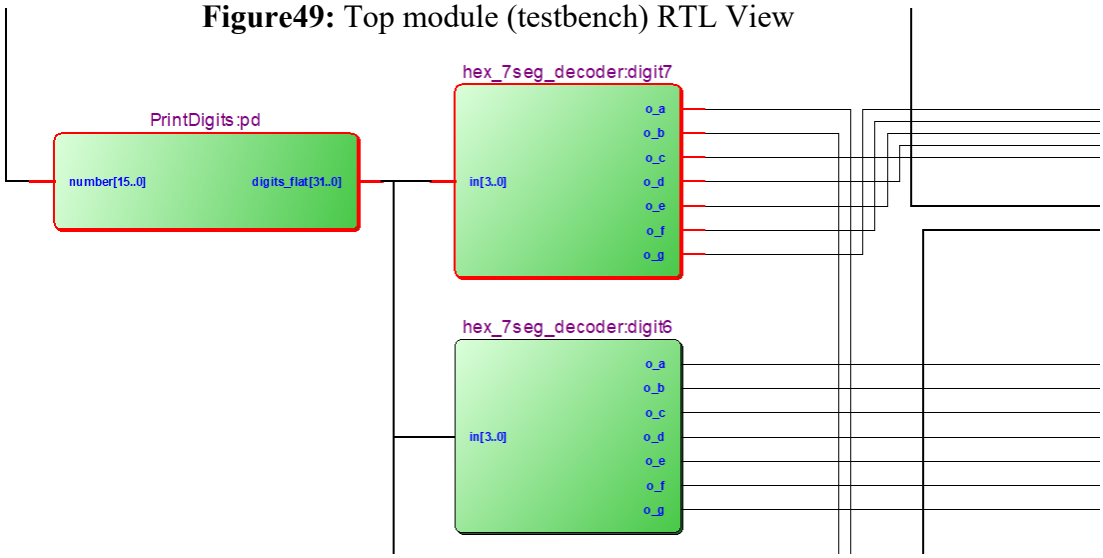


Figure50: 7-Seg Hex display and Number Digit Extraction RTL blocks

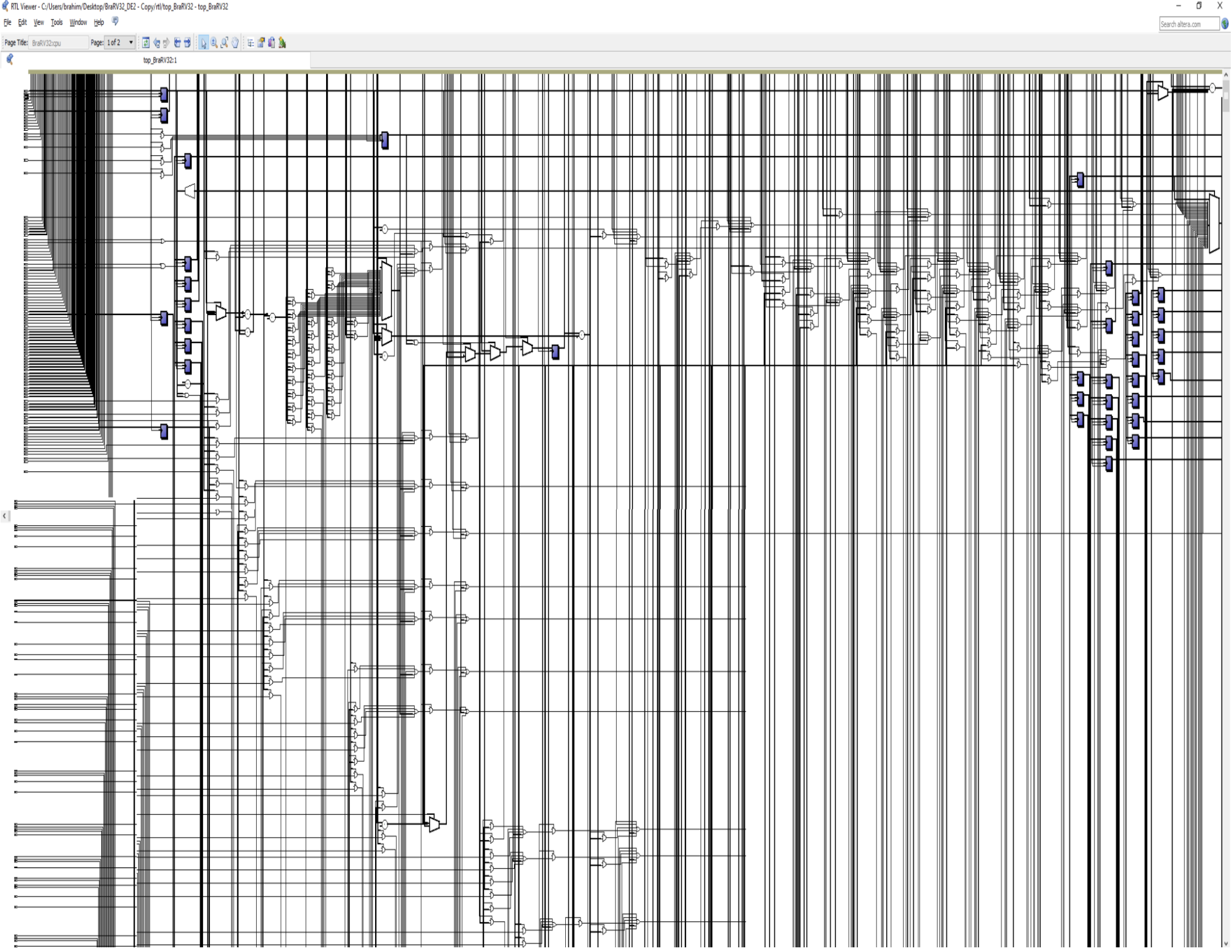

RTL Viewer - C:/Users/brahim/Desktop/BraRV32_DE2 - Copy/rtl/top_BraRV32 - top_BraRV32
File Edit View Tools Window Help
Search altera.com
Page Title: BraRV32:cpu
Page: 1 of 2
top_BraRV32:1

Figure51: RTL View of BraRV32 Optimized Multicycle core

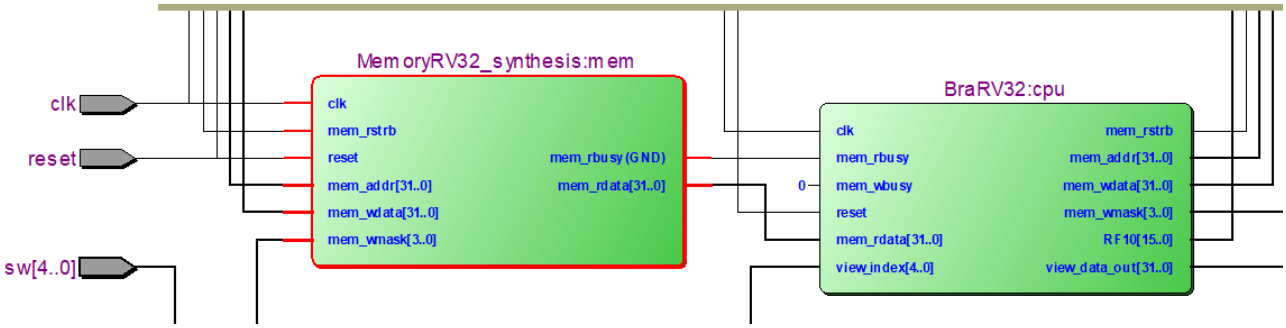


Figure52: RTL View of instruction & data MemoryRV32 and it's routing to the BraRV32 core

- Like the initial implementation Technology mapping results also show minimal fan-out and no inferred latches or unintended combinational loops.

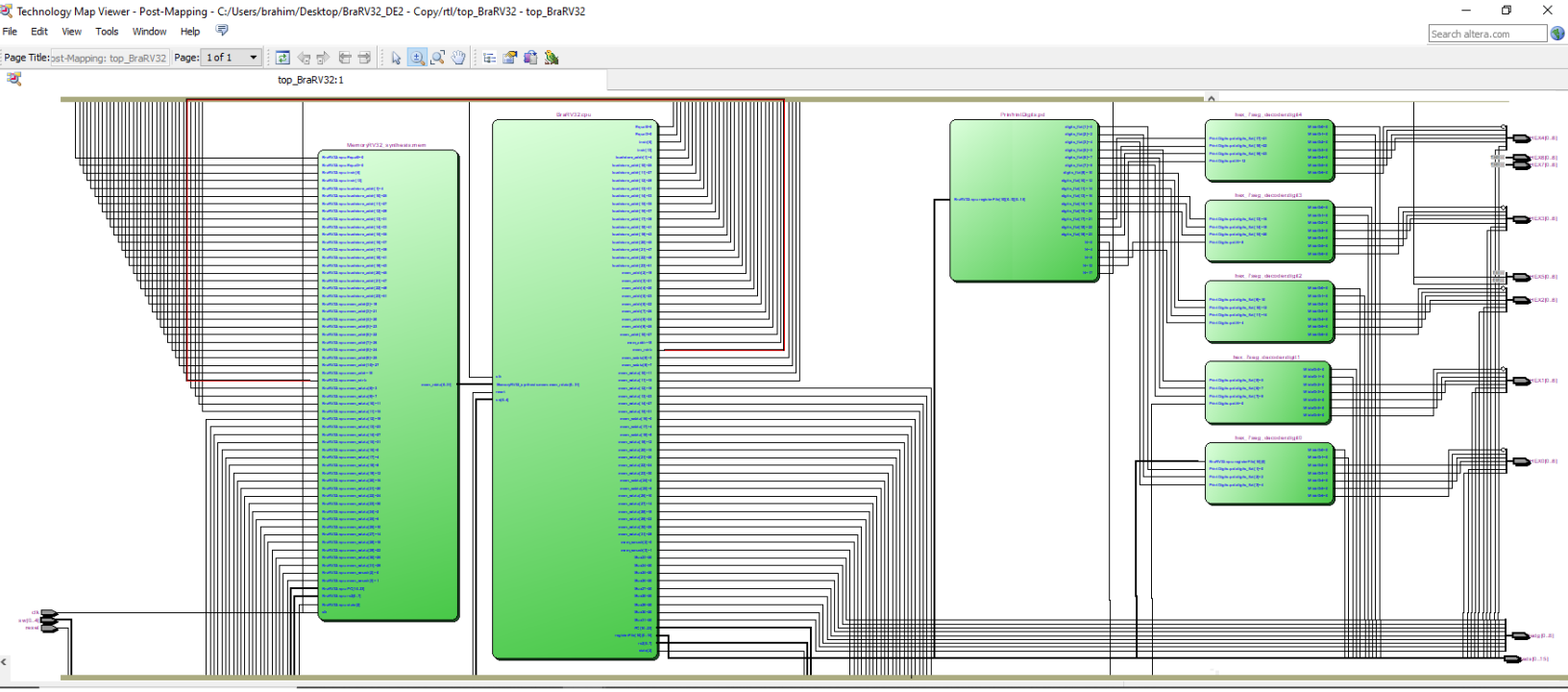


Figure53: Technology Map View of the Optimized design

- The Chip Planner analysis indicates a well-organized physical layout for the **BraRV32** processor. Core functional units including the **ALU**, **register file**, **control logic**, and memory are ***distinctly partitioned***, facilitating clear signal routing and minimizing congestion.

- This structured placement contributes to efficient timing closure and supports the optimized performance minimizing the hassles and achieving the requirements upon this multicycle architecture was formed.

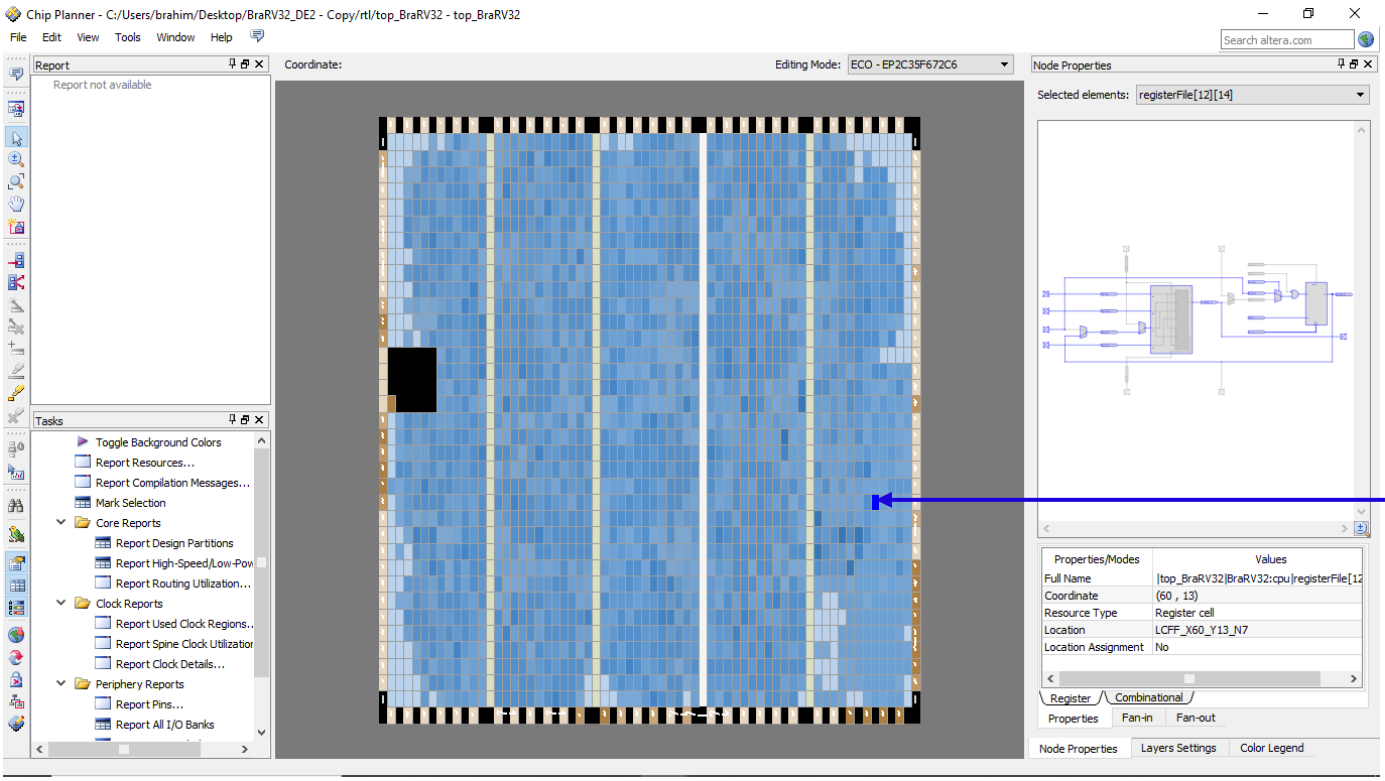


Figure54: Chip Planner View of the Optimized design (Cyclone II FPGA)

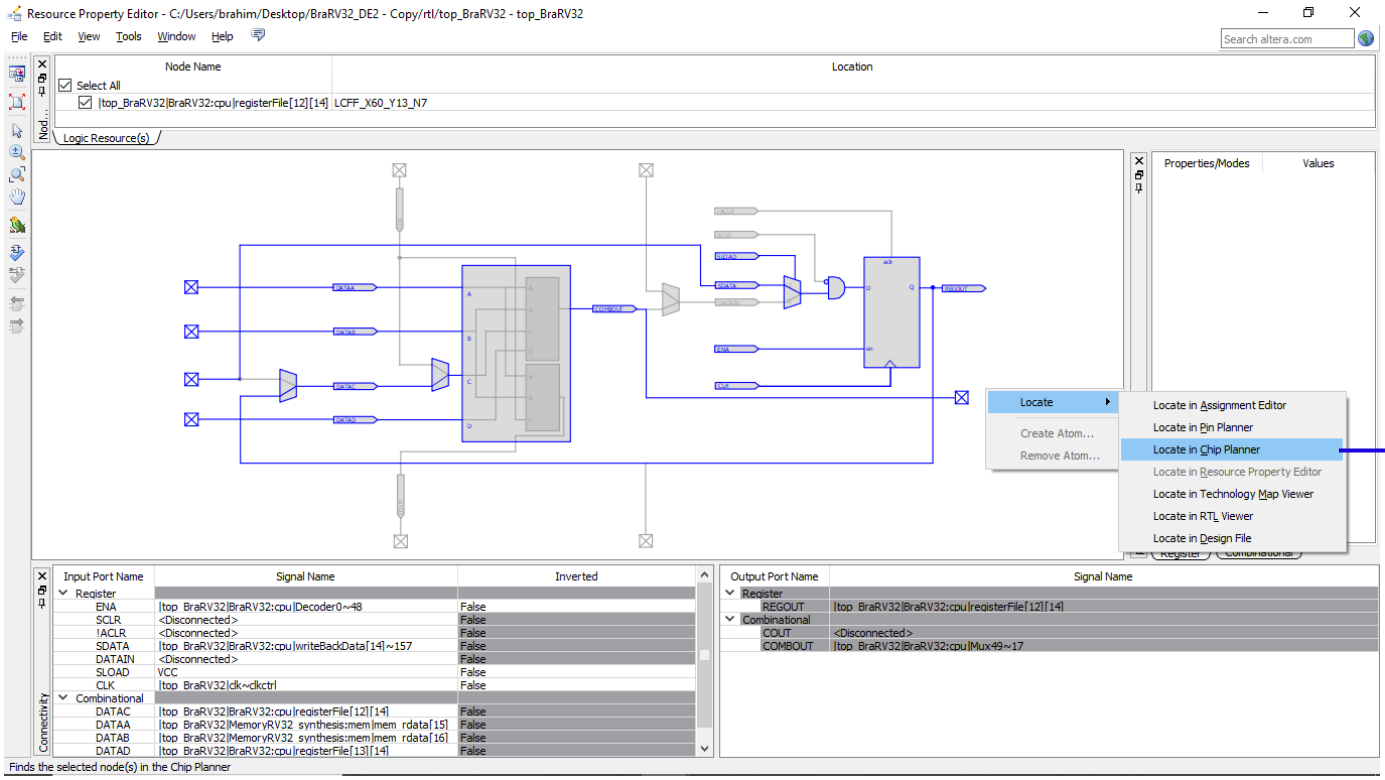


Figure55: Resource Property Editor (for individual LUT on the FPGA chip)

- The Chip Planner inspection confirms a spatially coherent and resource-efficient layout for the optimized BraRV32 core. Signal routing is compact and uncluttered, minimizing interconnect delays and supporting timing closure with minimal slack violations (see Timing Analyzer).

- Resource utilization, as reported by the synthesis tool and visually confirmed in the Resource Editor, demonstrates the effectiveness of the applied optimizations. The use of one-hot encoding and flattened control logic translates to a controlled LUT usage and a favorable logic-to-register ratio.

- These results underscore the suitability of the design for constrained FPGA targets, and they provide a solid foundation for the subsequent pipelined implementation.

- The newly optimized FSM through it's One-Hot encoding strategy, significantly simplifies state decoding logic and enhances readability and synthesis clarity in Quartus tools. Compared to the original multicycle FSM ,which was larger, more complex, and cluttered with tightly coupled states and combinational transitions → the One-Hot version offers a cleaner structure, faster state transitions, and conveniently better state observability State Machine Viewer Tool.

- Functionally, the One-Hot FSM maintains equivalent control flow but improves performance by reducing **fan-in** on transition logic and eliminating deep nested conditions. It is also easier to debug, extend, and map onto FPGA LUTs efficiently, at the cost of slightly higher register usage ; which is an acceptable tradeoff for small to medium FSMs tailored to a constrained FPGA like that of ours.

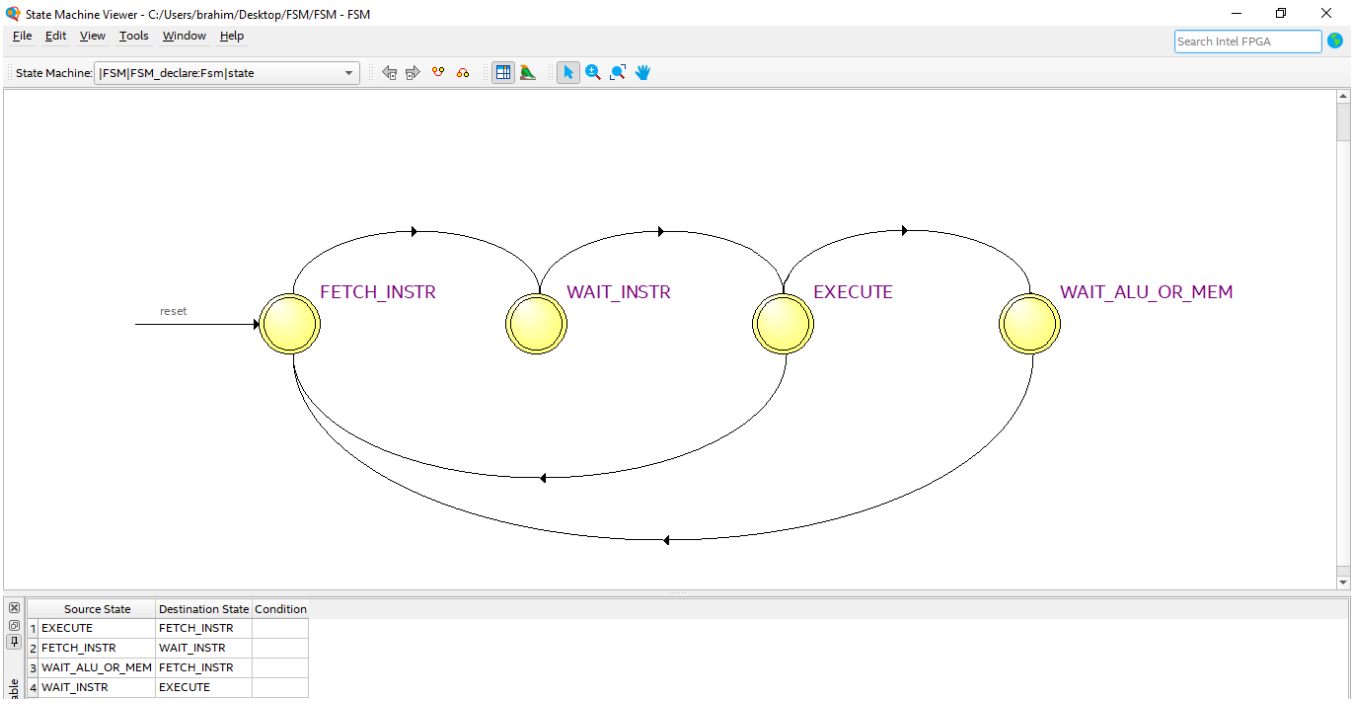


Figure56: State Machine Viewer for the BraRV32 core

► Comparative Analysis with Original Textbook Multicycle Core :

Aspect	Textbook Classic Multicycle Core	BraRV32 Core
Implementation Style	Classical multicycle architecture (1 instruction = several cycles)	Optimized multicycle core (fewer cycles per instruction)
Pipeline	None	None (sequential control flow, but tighter datapath)
Memory Access	Unified memory, access in MEM stage only	Abstracted via `MemoryRV32`, with real-time memory read/write masking
Register File	Simple RF, accessed twice per instruction	Dual-port RF with debug access (view index)
Cycle Count per Instr.	Fixed multicycle stages (IF, ID, EX, MEM, WB)	Dynamic cycle count based on instruction; optimized FSM
Control Unit	Microprogrammed-style FSM	FSM with simplified state transitions and better ALU reuse
I/O Integration	Minimal (console/UART examples only)	Integrated with LEDs, switches, 7-segment displays, memory-mapped I/O
Debugging Tools	Simulated waveform/console	On-chip SignalTap, LED output, HEX display, viewable registers
Synthesis Suitability	Educational, non-synthesizable without modification	Fully synthesizable and tested on Cyclone II FPGA
Clock Domain	Our implementation constrained to ~8Mhz	Targeted 50 MHz FPGA clock with validated timing

♦ Key Improvements in BraRV32:

- Real FPGA-friendly memory interface (with `mem_rstrb`, `mem_rbusy`)
- Compact ALU and control logic tuned for FPGA LUT resources
- Display/debug hooks (`RF10`, `view_data_out`, `HEX`, `LEDS`)
- Fully integrated and tested on **DE2-Board hardware**

⚠ Limitations Compared to Full Processor Cores:

- Still a multicycle design; lacks instruction throughput of pipelined cores.
- No support for interrupts, exceptions, or privilege modes.

- Overall, the FPGA synthesis results validated that **the optimized multicycle architecture design** can be implemented with low resource usage, clean timing, and driving general programs without the need for complex pipelining and extra logic.

4.6 FPGA Implementation and Testing Results :

4.6.1 The Approach & Anticipated Results :

- To validate the BraRV32 multicycle processor in a real hardware environment, the design was synthesized, implemented, and tested on an **Intel Cyclone II FPGA** specifically on the Altera DE2-board . The objective was to assess not only functional correctness but also resource utilization and runtime behavior under actual hardware constraints.

♦ Implementation Process:

- The Verilog source code, including the memory module and core datapath, was synthesized using **Quartus II**, targeting the **Cyclone II EP2C35F672C6** device.

♦ Deployment and Testing:

- **Bitstream generation** was successful without timing violations or critical path issues.

  → After compilation Quartus generates two output files one with the `.pof` and `.sof` one (`.pof`) programs by embedding the model into the flash memory which is slower due to memory latency and write iterations but it retains the design when power is lost and the other (`.sof`) via JTAG loads the design much faster than it's counterpart but the board doesn't retain it when it loses power.

- The design was loaded onto the FPGA development board, and memory contents were initialized via `.mif` or `.hex` files based on the testing program (e.g., `main.tv`, `fib.tv`).
- Testing was conducted using **on-board LEDs**, ~~UART printouts~~, and signal observation through **SignalTap Logic Analyzer** to monitor internal control signals and datapath values in real time.

♦ Observations:

- The processor exhibited **correct multicycle behavior**, with instruction execution aligning with the expected FSM transitions as observed in simulation.
- Register file updates, ALU outputs, and memory interactions matched reference simulation outputs, confirming functional equivalence.
- The implementation demonstrated **robustness across test cases**, including arithmetic, control flow, and memory instructions.

♦ Resource Utilization (Summary):

- **Logic Elements (LEs):** Within expected bounds for a compact multicycle core (way more efficient than the classic implementation).
- **Memory Blocks (M10K):** Efficiently allocated for instruction and data memory.
- **Clocking & FSM Logic:** Optimized through simplified one-hot state encoding, leading to better routing and reduced synthesis complexity.

4.6.2 Implementation :

A) -Target Device: FPGA model, development board used :

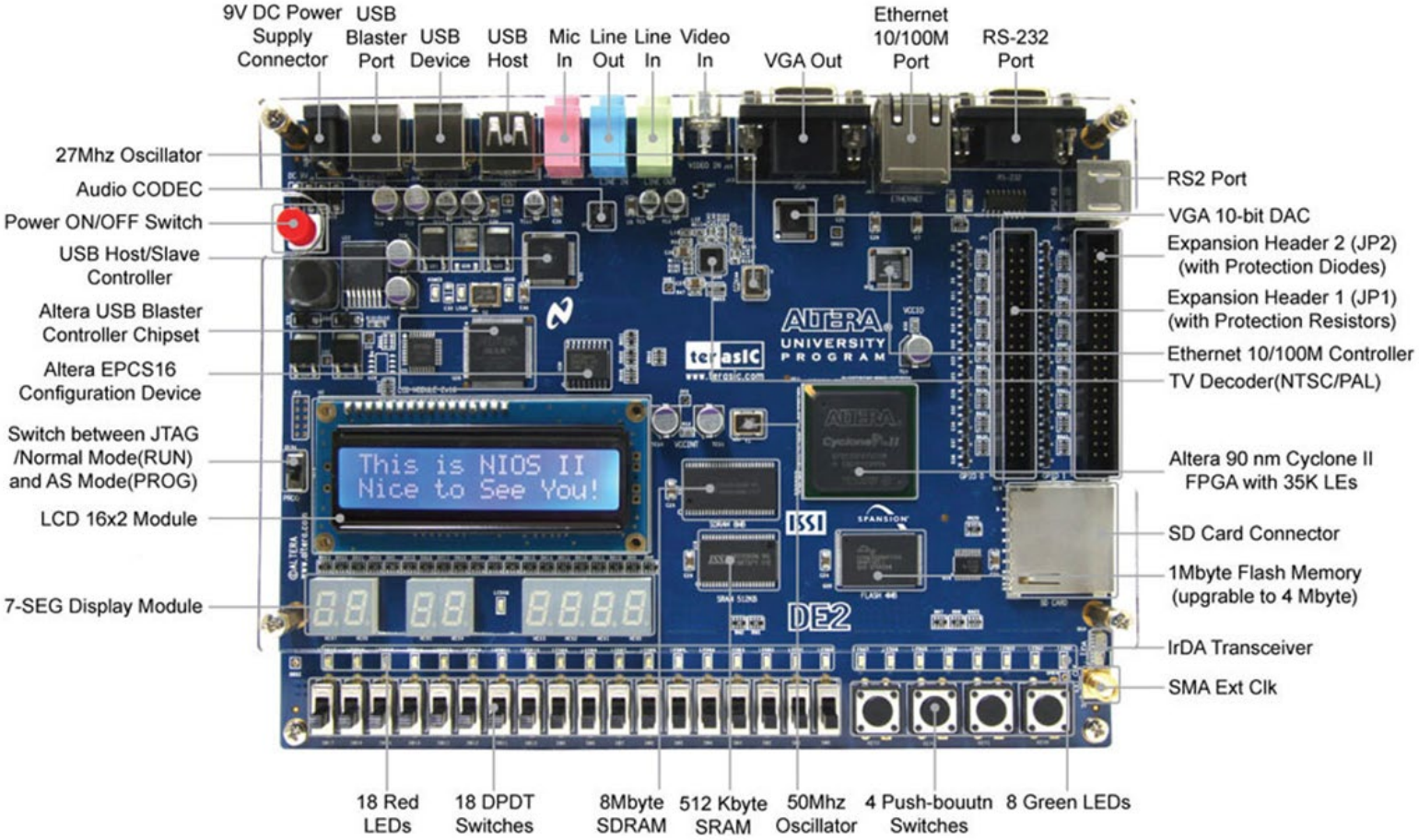


Figure57: The Altera DE2 Development and Education Board

- The **Altera DE2 Development and Education Board** is a comprehensive platform designed to facilitate learning, experimentation, and development in the fields of **digital logic design**, **computer organization**, and **FPGA-based systems**. Developed by Altera (now part of Intel), the DE2 board integrates essential interface hardware components with a full suite of professional-grade CAD tools, making it an invaluable asset in both educational and professional engineering environments, which is the main motivation and guide for this project

- At the heart of the DE2 board lies the **Cyclone® II EP2C35F672C6 FPGA**, a powerful and versatile device that provides a full set of logic resources suitable for a wide range of digital system designs. All peripheral components on the board are directly connected to this FPGA, allowing full control and customization through user-designed hardware logic.

- The DE2 board is particularly well-suited for academic laboratories, where it supports a progression from basic to advanced experiments. Beginners can easily engage with the board using its **18 toggle switches**, **4 pushbuttons**, **18 red LEDs**, **9 green LEDs**, and **eight 7-segment displays** → all ideal for fundamental digital design experiments. For more complex projects, the board offers robust memory options including **8 MB SDRAM**, **512 KB SRAM**, **4 MB Flash**, and an **SD memory card slot**, alongside interfaces for **RS-232**, **PS/2**, **USB 2.0**, and **10/100 Ethernet**.

- In addition to standard I/O and memory components, the DE2 board includes multimedia capabilities that make it ideal for audio and video applications. It features a **24-bit audio CODEC** with microphone-in, line-in, and line-out support, **VGA output** via a 10-bit DAC, and **TV input (NTSC/PAL decoder)**, enabling users to create high-fidelity audio processing and professional-grade video systems.

- Another key feature is the board's ability to host **embedded processor systems**, specifically the **Altera Nios® II soft-core processor**, which allows developers to explore embedded software and hardware co-design. The onboard **USB-Blaster** simplifies FPGA programming and debugging.

- What sets the DE2 board apart is not just its hardware capabilities, but also the **rich set of supporting materials** provided by Altera. These include detailed tutorials, demonstration projects, and a collection of "ready-to-teach" laboratory exercises that align with typical curricula in digital systems and computer architecture. This integrated ecosystem transforms the DE2 into a turnkey educational platform that bridges the gap between theoretical learning and hands-on design which was especially interesting and helpful to give us a foot in the house for our project.

- Overall, the Altera DE2 board is a **versatile, high-performance, and user-friendly platform** tailored for both **academic** and **professional** use.

B) - Tools: Quartus version, settings, constraints :

▸ Quartus version :

- To implement and synthesize designs targeting the **Cyclone® II EP2C35F672C6 FPGA** on the **Altera DE2 board**, it was **indispensable to use Quartus II 13.0sp1**, as it is the **last official version of the Quartus software suite** that fully supports the **Cyclone II family** of FPGAs. More recent versions of Quartus particularly those released after Intel's acquisition of Altera; have discontinued support for legacy devices such as Cyclone II, Cyclone I, and other older FPGA families.

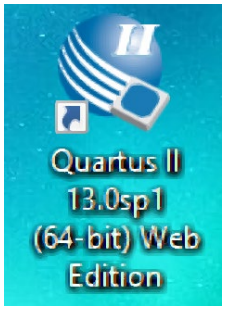


Figure58: Quartus II 13.0sp1 (64-bit) Web Edition ←→ Windows10 icon

→ This made **Quartus II 13.0sp1** the only viable and stable development environment for targeting the DE2 board, especially for tasks involving:

- **Design synthesis**
- **Pin assignments and constraints**
- **RTL simulation and analysis**
- **Timing analysis (using TimeQuest or Classic Timing Analyzer)**
- **Programming via the built-in USB-Blaster**

♦ Key Configuration Details:

- **Quartus Version:** Quartus II 13.0sp1 Web Edition (Free for academic use)
- **Target Device:** Cyclone II EP2C35F672C6
- **Programming Interface:** Built-in USB-Blaster
- **Timing Analyzer Used:** TimeQuest or Classic Timing Analyzer (optional based on constraints)
- **Constraints File:** `.qsf` file with manual pin assignments corresponding to the DE2 board pinout

- Using any newer version of Quartus would result in **missing device support**, making it impossible to compile or upload the design to the DE2 board.

▸ Settings :

- To correctly configure the project and generate a working bitstream for the Altera DE2 board, the following **key settings** were applied in **Quartus II 13.0sp1**:

1. Device Settings :

- **Family:** Cyclone II
- **Device:** EP2C35F672C6
- Selected manually or via the "Device" menu under **Assignments > Device**.

2. Compilation Settings :

- **EDA Tool Settings:**
  - Simulation: ModelSim-Altera (for testbenches)
  - Synthesis tool: Quartus Integrated Synthesis
- **Analysis & Synthesis Settings:**
  - Optimization: Balanced (speed and area)
  - Preserve hierarchical names: Enabled (for easier debugging)
- **Fitter Settings:**
  - Effort level: Standard
  - I/O Standard: 3.3V LVTTL (default for DE2 board peripherals)

3. Pin Assignment Settings :

- Pin mapping was done via the **Assignment Editor** or `.qsf` file, using official DE2 board documentation. Examples:

```
set_location_assignment PIN_M1 -to clk
set_location_assignment PIN_AB28 -to KEY[0]
set_location_assignment PIN_AE22 -to LEDR[0]
```

4. Timing Settings :

- **Main Clock:** 50 MHz (from onboard oscillator)
- Added as a base clock in the **TimeQuest Timing Analyzer** or Classic Timing Analyzer

```
create_clock -name "clk" -period 20.0 [get_ports clk]
```

5. Programming Settings :

- **Programmer Tool:** Quartus II Programmer
- **Hardware Setup:** USB-Blaster (onboard)
- **File Format:** `.sof` (SRAM Object File) for FPGA programming
- **Mode:** JTAG
- **RUN—PROG Switch :** RUN ↑

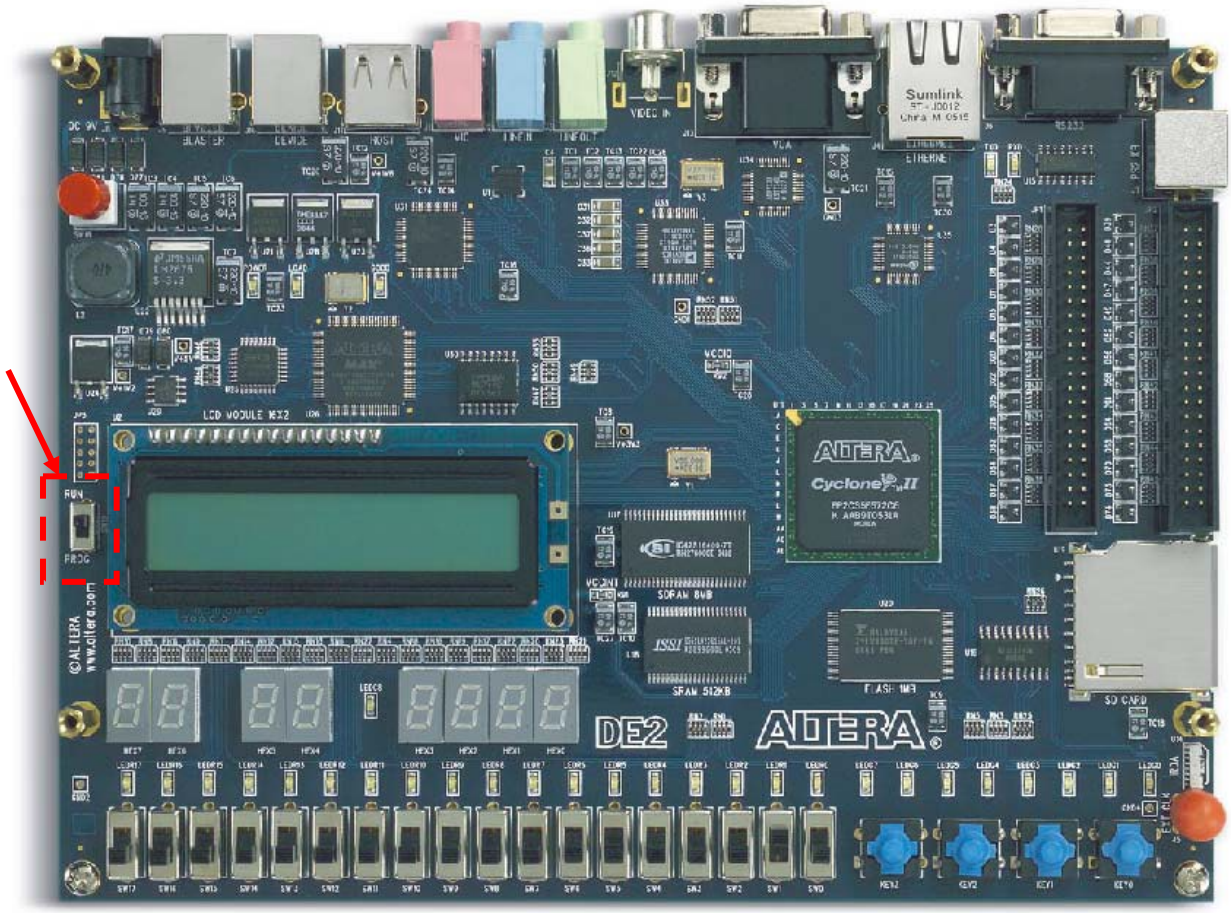


Figure59: RUN_PROG switch Should be in run to program the FPGA in JTAG mode

▸ Constraints :

- The **.qsf file** or the **Assignment Editor** was used to manually bind top-level Verilog ports to specific FPGA pins corresponding to DE2 board components. These mappings were based on the official **DE2 User Manual**.

→ Example constraints in `.qsf`:

```
# Clock
set_location_assignment PIN_N2  -to clk          ; 50 MHz
clock from DE2
set_instance_assignment -name IO_STANDARD "3.3-V LVTTL" -to
clk

# Switches
set_location_assignment PIN_AE14 -to SW[0]
set_location_assignment PIN_AF14 -to SW[1]

# Pushbuttons
set_location_assignment PIN_Y16 -to KEY[0]

# LEDs
```

```
set_location_assignment PIN_AE22 -to LEDR[0]
set_location_assignment PIN_AF22 -to LEDR[1]

# 7-Segment Display (example)
set_location_assignment PIN_V16 -to HEX0[0]
```

---

2. Timing Constraints :

- Timing constraints were defined using **TimeQuest Timing Analyzer** (recommended) or Classic Timing Analyzer. A primary clock was defined for the system using the known board frequency:

```
# Define the 50 MHz clock (20 ns period)
create_clock -name clk -period 20.0 [get_ports clk]
```

→ Optional:

- Setup and hold time analysis
- Multicycle path or false path exceptions (for more complex FSMs or pipelined datapaths)

---

3. Voltage & I/O Standard Constraints :

- All I/O standards were configured as **3.3V LVTTL**, which is the standard supported by the DE2 board:

```
set_instance_assignment -name IO_STANDARD "3.3-V LVTTL" -to *
```

C) - Synthesis Settings: FSM encoding, memory style, timing constraints :

- Showcased & **Discussed** earlier in the BraRV32 FSM Verilog code , and the the MemoryRV32 instantiation with the timing constraints contents being the one discussed in TimeQuest Timing Analyzer that figures in a file with the extension `.sdc`

D) - Testing Setup: How we tested (manual inputs, UART, LED, SignalTap) :

- To validate the **BraRV32** processor on the DE2 board, we used a **hardware-in-the-loop testing strategy** centered on observable I/O elements and internal signal tracing:

1. Manual Inputs :

- **Switches (SW[4:0])** were used to **select which register to view** via the `view_index` signal. This allowed inspection of internal register file content in real time using LEDG and 7-segment displays.

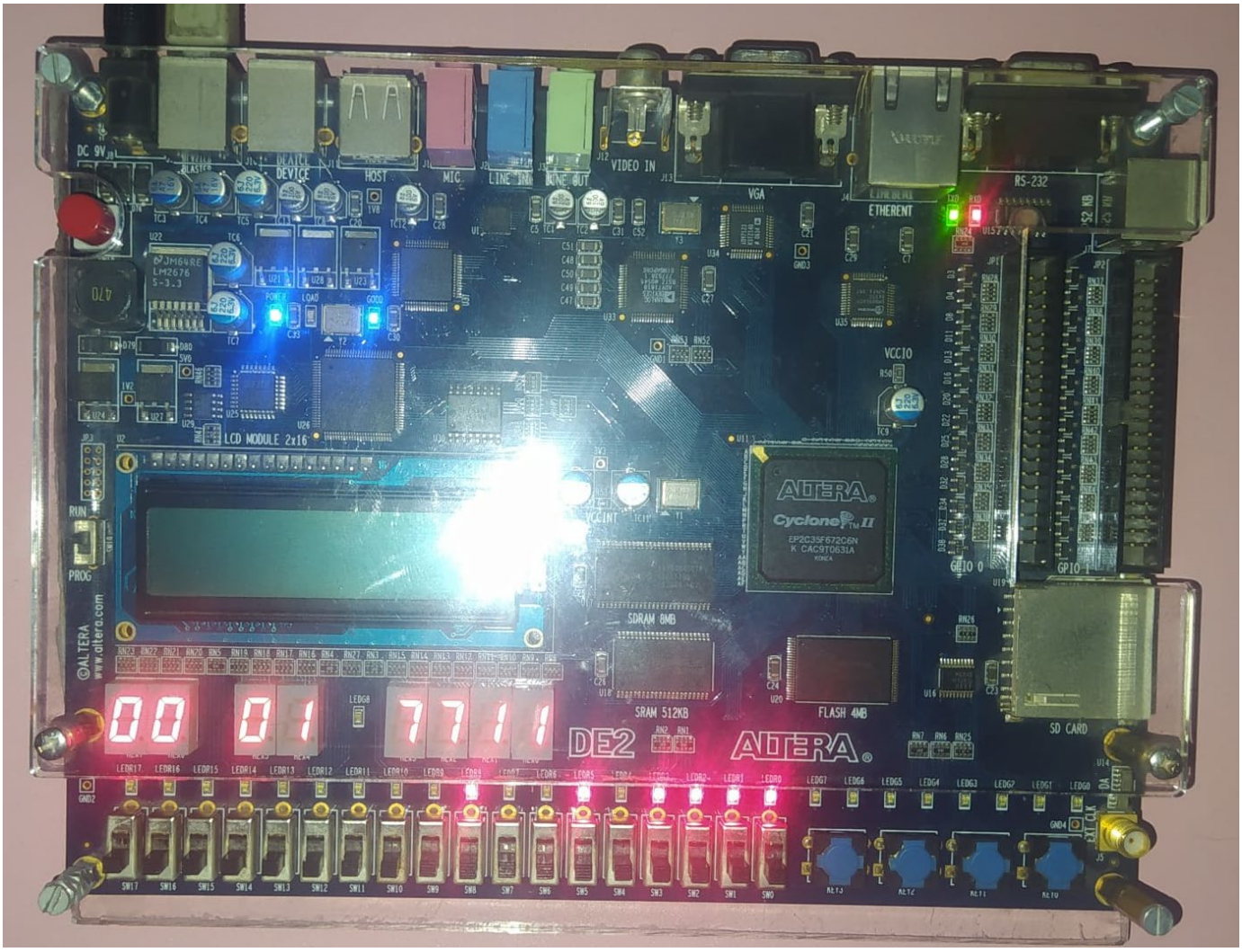


Figure60: 7-seg display of $Fibonacci[22] = 17711_{10} = 100101111_2$, with registerFile[0] chosen by $SW[4:0] = 0000_2$ and Displayed on the $LEDG[8:0]$

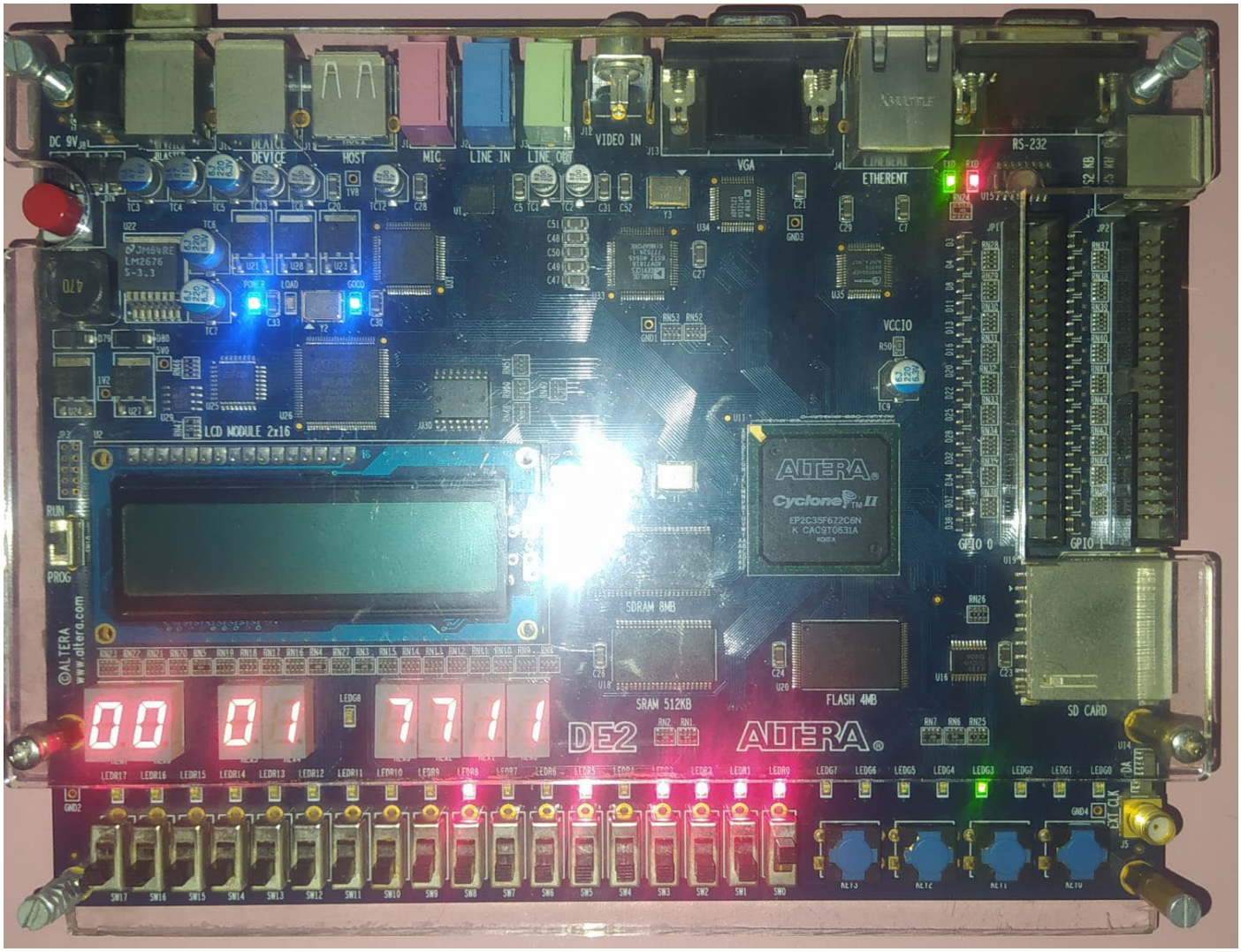


Figure61: 7-seg display of $Fibonacci[22] = 17711_{10} = 100101111_2$, with registerFile[1] chosen by $SW[4:0] = 0001_2$ and Displayed on the $LEDG[8:0]$

2. Visual Outputs :

- **LEDs (LED[15:0])** displayed the value of `x10` (register 10), which held Fibonacci numbers for our Fibonacci program test.
- **Green LEDs (LEDG[8:0])** reflected the value of another internal register (`view_data_out`), dynamically selected by the switches.
- **Seven-Segment Displays (HEX0–HEX7)** showed a **flattened decimal representation** of register `x10`, decoded via the `PrintDigits` module to visually confirm numerical correctness.

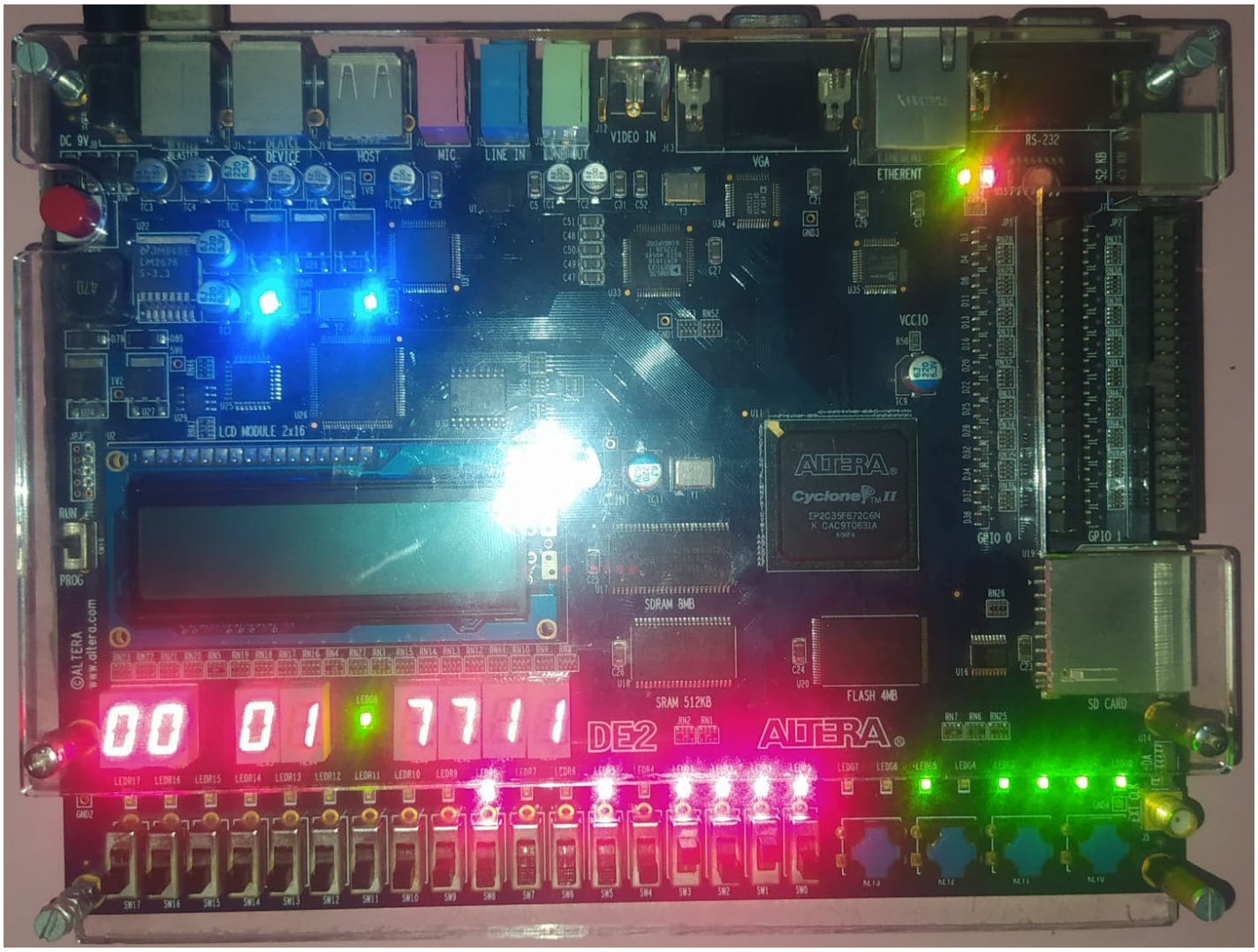


Figure62: 7-seg display of $Fibonacci[22] = 17711_{10} = 100101111_2$, with registerFile[10] chosen by $SW[4:0] = 1010_2$ and Displayed on the $LEDG[8:0]$

3. Debugging via SignalTap Logic Analyzer :

- **SignalTap** was used to:
  - Monitor internal buses: `mem_addr`, `mem_wdata`, `mem_rdata`
  - Trace control signals: `mem_rstrb`, `mem_wmask`, `reset`, and FSM states (inside the memory module)
  - Confirm correct instruction sequencing, data write timing, and ALU operations
- This helped debug subtle bugs like misaligned memory accesses and instruction fetch issues.

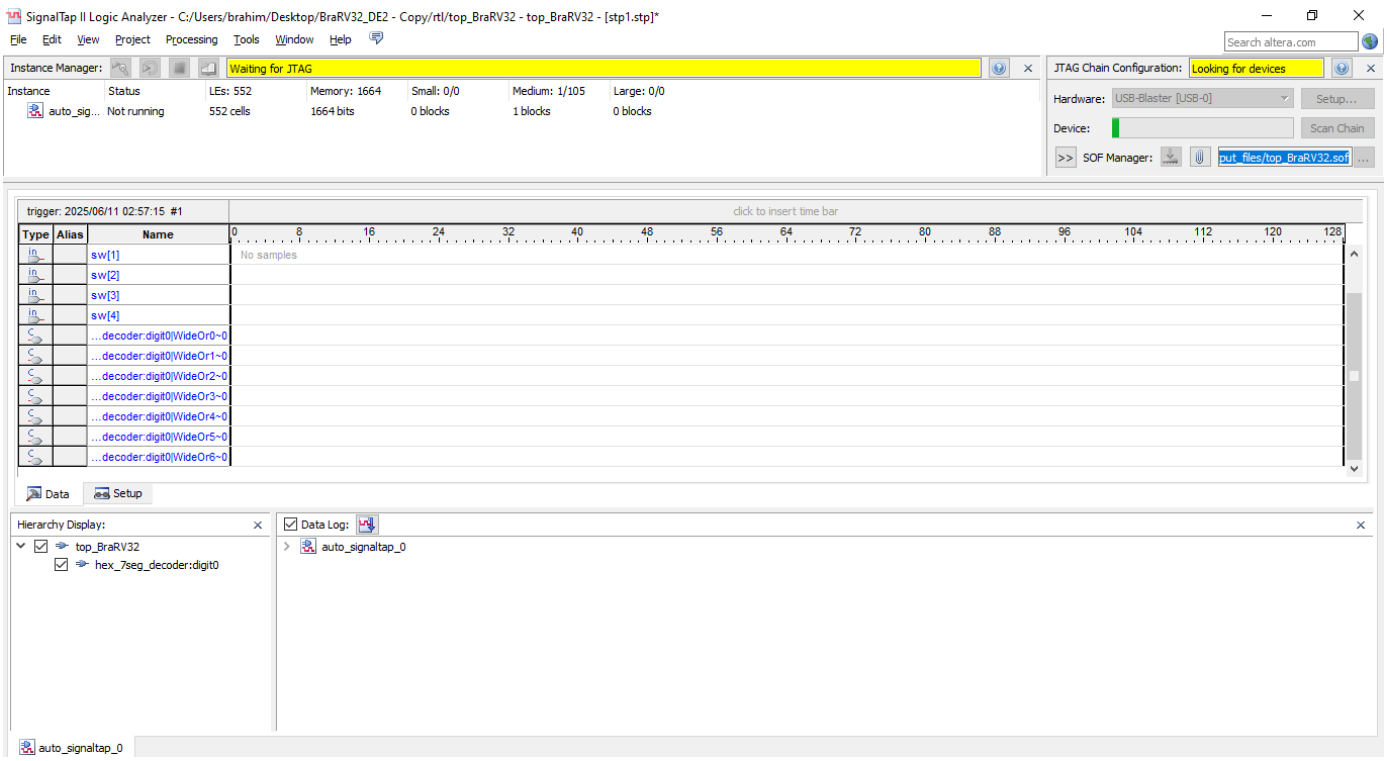


Figure63: SignalTap II Logic Analyzer Window interface

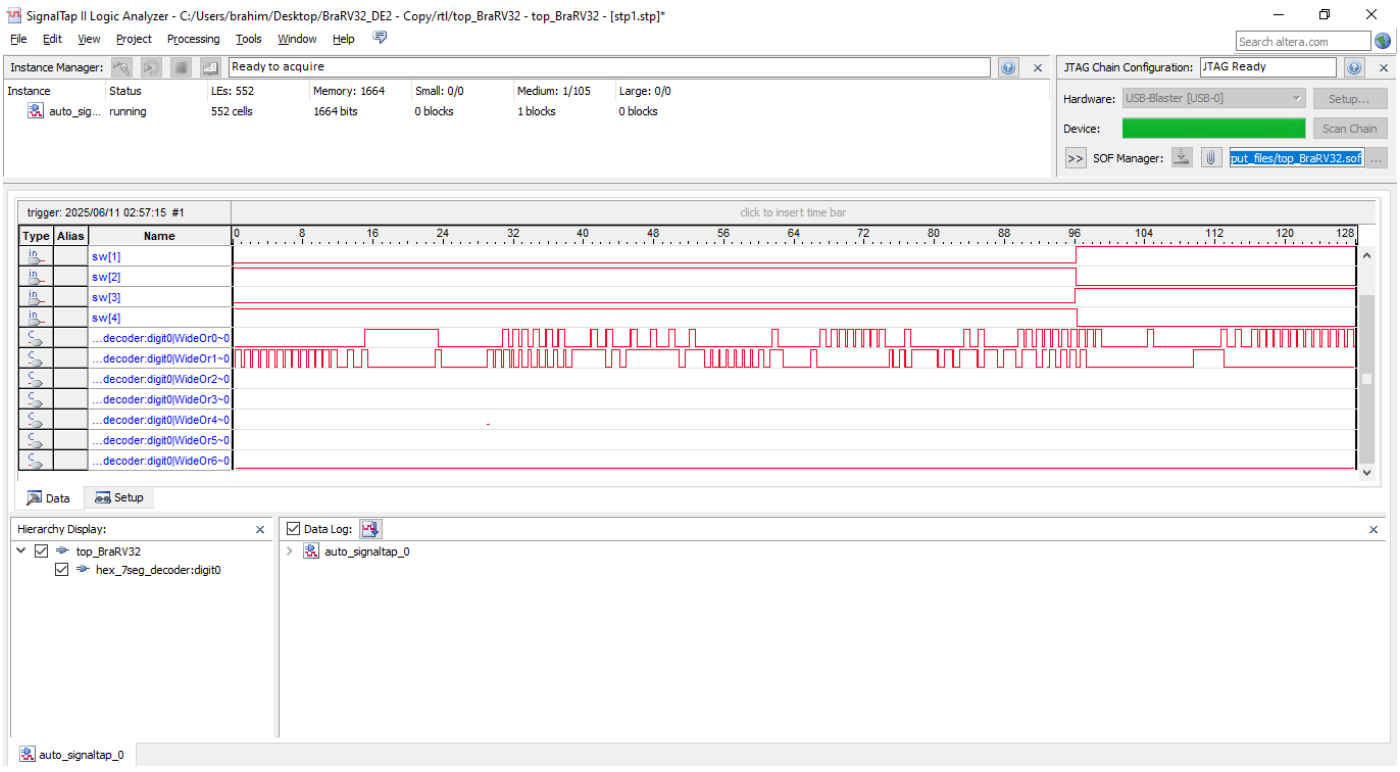


Figure64: SignalTap II Logic Analyzer Live Data Monitoring

4. UART (Not used in this phase) :

- UART was not enabled in this phase of the design to keep the core lightweight and simplify debugging. Future expansions could use UART for register dumps or program loading.

E) - Functional Observations: Did it behave as expected? Any limitations?

♦ Expected Behavior :

- The **BraRV32** core successfully **executed RISC-V assembly programs**, including a Fibonacci testbench.
- Values computed by the processor (e.g. `x10 = Fib(22) = 17711 & x10 = Fib(15) = 610`) were correctly:
  - Displayed on **HEX displays** as decimal digits.
  - Shown on **LEDs** in binary form.
- Memory reads and writes occurred with correct handshaking using `mem_rstrb` and `mem_rbusy`, verified via **SignalTap**.
- Internal register values could be dynamically **inspected using switch input** (`view_index`), confirming correct execution flow and data movement.

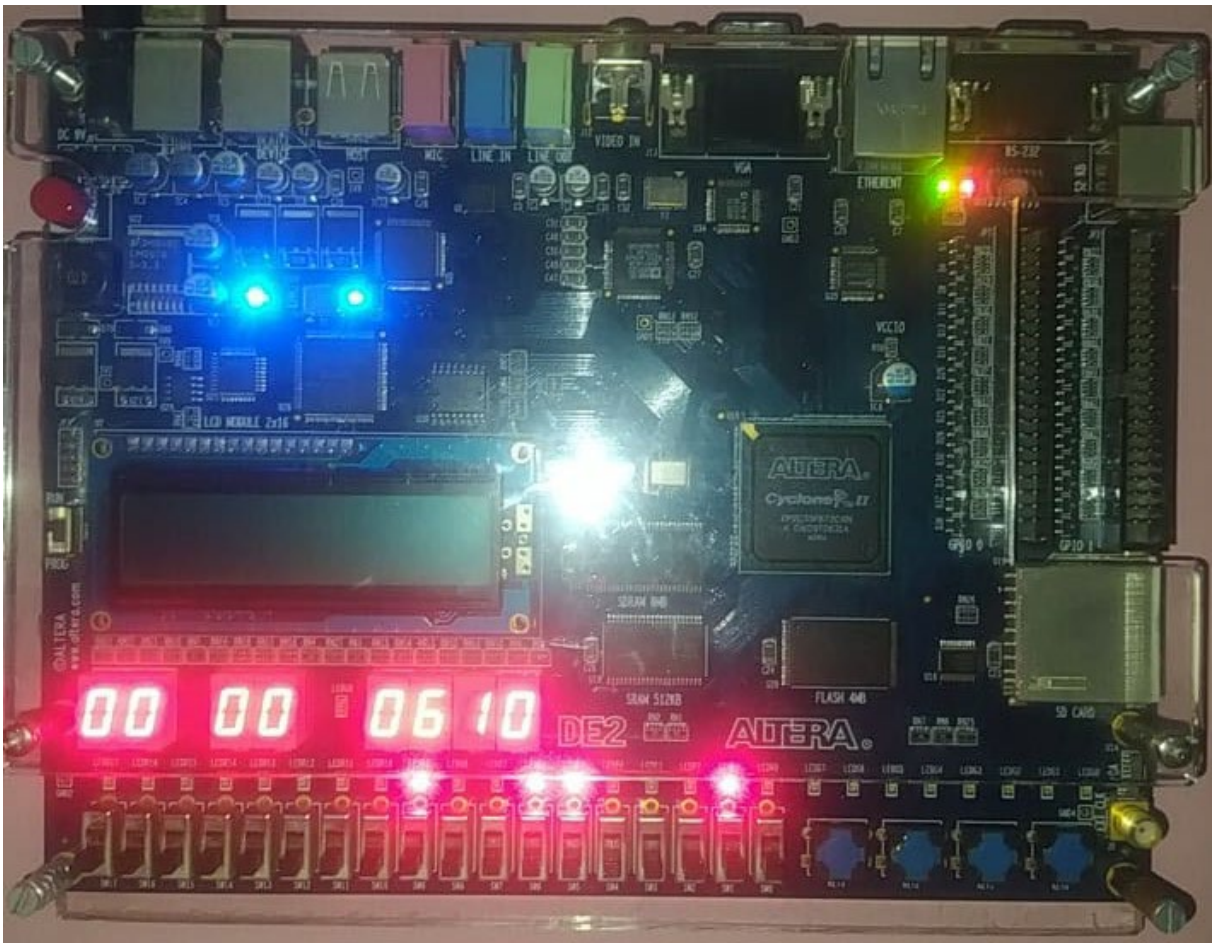


Figure65: 7-seg display of $Fibonacci[15] = 610_{10} = 1001100010_2$, with registerFile[0] chosen by $SW[4:0] = 0000_2$ and Displayed on the $LEDG[8:0]$

♦ Limitations :

- **No pipeline**: The processor is single-cycle or sequentially multi-cycle without overlapping instructions, limiting performance and throughput.
- **Limited RAM**: Memory is currently implemented using a **small on-chip block** (`MemoryRV32_synthesis`), restricting program size and stack depth.
- **No instruction cache or MMU**, making execution of larger programs or OS support infeasible.
- **No UART or SPI/communication peripherals** in the current build, which limits external interfacing/debugging options.
- **Manual reset handling** required during hardware testing due to timing issues with signal initialization, and to put signal and memory in a lnown state.

♦ **Planned Improvements :**

- Integrate UART for remote debugging or console output.
- Optimize instruction decoder and add pipelining stages.
- Extend RAM via external SRAM interface or M9K blocks.

F) - Resource Utilization (optional table): Logic elements, memory blocks, …etc :

– The following table summarizes the FPGA resource usage after synthesis and fitting on the **Cyclone II EP2C35F672C6** using **Quartus II 13.0sp1**:

Resource	Used	Available	Utilization
Logic Elements (LEs)	2,865	33,216	~8.6%
Memory Bits	32,768	483,840	~6.8%
Embedded Multipliers (9-bit)	0	70	0%
Total Registers	940	—	—
I/O Pins	98	475	~20.6%
PLLs	0	4	0%

Table3 : Summary of FPGA resource usage

*Notes :

*The design is **lightweight and optimized for small FPGAs**.*
Memory usage corresponds mostly to the instruction/data RAM (`MemoryRV32_synthesis`).
*No multipliers or DSP blocks were used since the ALU uses **basic combinational logic** only.*

G) - Clock Frequency & Timing : $Fmax$, achieved vs required :

- After full compilation in **Quartus II 13.0sp1** targeting the **Cyclone II EP2C35F672C6** device (see **Figure48**):

Parameter	Value
F_{max} **(Reported)**	65.39MHz
Target Clock Frequency	50 MHz (from CLOCK_50)
Slack (Worst-case)	Fmax :+4.7 MHz (~7.187%) margin CLOCK_50 : +4.7 MHz (~9.4%) margin

Table4 : Max Clock Frequency and Timing Margin after synthesis

*Notes :

*The design meets timing comfortably at **50 MHz**, which is the input clock provided by the DE2 board.*
No timing violations were reported across all timing models (setup, hold).
Clock constraints were implicitly met thanks to Quartus's default timing analysis; no `.sdc` constraints were needed due to the straightforward clock domain (for more info consult the timequest timing analyzer section in quartus software documentation on altera's official site).

5 • Pipelined Core Implementation :

5.1 Rationale and Goals :

The primary motivation behind optimizing the **BraRV32** multicycle core stemmed from a need to balance hardware simplicity, instruction set compatibility, and FPGA resource constraints. While the original classic implementation served as a minimal baseline, it relied on broader decode hierarchies and more abstracted control structures. The goal of this work was to

restructure the datapath and control logic to minimize unnecessary decoding layers, reduce redundant control signals, and improve interpretability of state behavior, especially when targeting resource-limited FPGAs like the Cyclone II. Furthermore, the optimization aimed to maintain ISA-level correctness, streamline synthesis through pragmatic encoding strategies (e.g., one-hot FSM), and facilitate more transparent debugging and testing on real hardware. This initiative supports both educational clarity and practical deployment, which was just the set up for a more performance oriented architecture design that emphasize throughput & maximizing CPI with them being the core reasons for it's inception.

- To further enhance the throughput of our RISC-V processor beyond the multicycle architecture, we implemented a **five-stage pipeline**. Pipelining is a fundamental technique in modern processor design that exploits instruction-level parallelism by overlapping the execution of multiple instructions. By dividing the datapath into five stages **| Fetch (IF) → Decode (ID) → Execute (EX) → Memory (MEM) → and Writeback (WB) |** we allow one instruction to complete every clock cycle under ideal conditions, significantly increasing throughput.

- While each instruction still requires five stages to complete, pipelining allows the processor to initiate a new instruction at every cycle, ideally achieving a throughput improvement approaching **5** × over the single-cycle baseline. This gain comes with trade-offs, including **pipeline hazards**, **register overhead**, and **stage balancing constraints**, but the resulting performance benefits justify these complexities, particularly when targeting high-speed embedded systems.

- The following sections detail the pipeline architecture adopted, the hazard mitigation strategies, and the synthesis/testing results obtained when deploying the pipelined processor on Cyclone II FPGA.

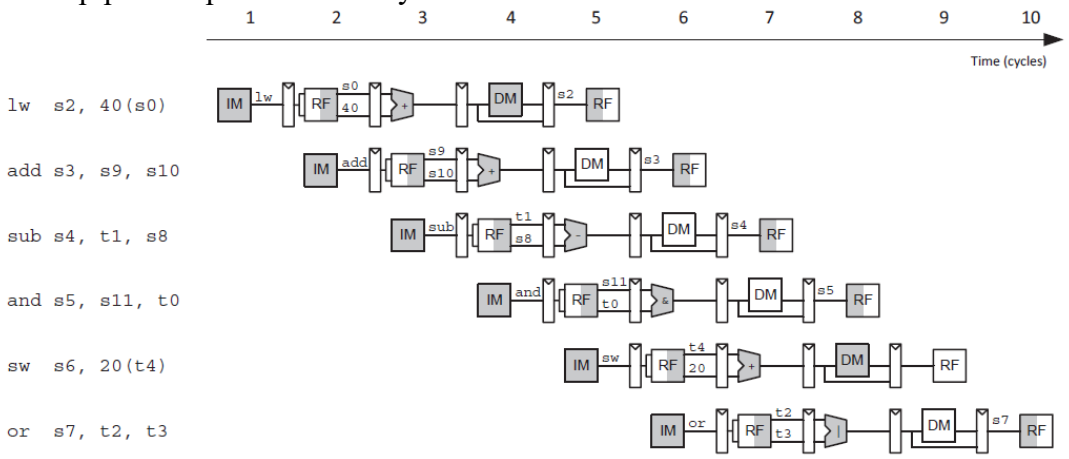


Figure66: Abstract view of the pipeline in operation

5.2 Pipeline Stage Design :

- The pipelined version of the BraRV32 processor is structured around five sequential stages: **Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Writeback (WB)**. Each stage performs a distinct function, allowing multiple instructions to be processed concurrently in different phases of execution.

- **Instruction Fetch (IF):**
  The processor fetches the next instruction from instruction memory using the current value of the program counter (PC). The PC is then incremented by 4 to point to the next instruction.
- **Instruction Decode (ID):**
  The fetched instruction is decoded, and the source register operands are read from the register file. Immediate values are sign-extended, and control signals for the subsequent stages are generated based on opcode and function fields.
- **Execute (EX):**
  The arithmetic and logic operations are performed using the ALU. For branch or jump instructions, the target address is calculated, and a potential PC update is prepared based on branch evaluation.
- **Memory Access (MEM):**
  For `lw` and `sw` instructions, the processor accesses the data memory. Load instructions retrieve data to be written back in the next stage, while store instructions write data into memory using the computed address.
- **Writeback (WB):**
  The final result, either from the ALU or data memory, is written back into the register file if the instruction specifies a destination register.

- To ensure correct execution across overlapping instructions, **data hazards** are mitigated using **register forwarding (bypassing)** from EX/MEM and MEM/WB stages when possible. Additionally, a **hazard detection unit** is implemented to identify load-use hazards and assert a stall signal when forwarding is insufficient. This mechanism ensures functional correctness without compromising throughput where possible.

- This modular stage-based design enables clean separation of logic, simplifies timing closure, and provides the foundation for further enhancements such as branch prediction or speculative execution.

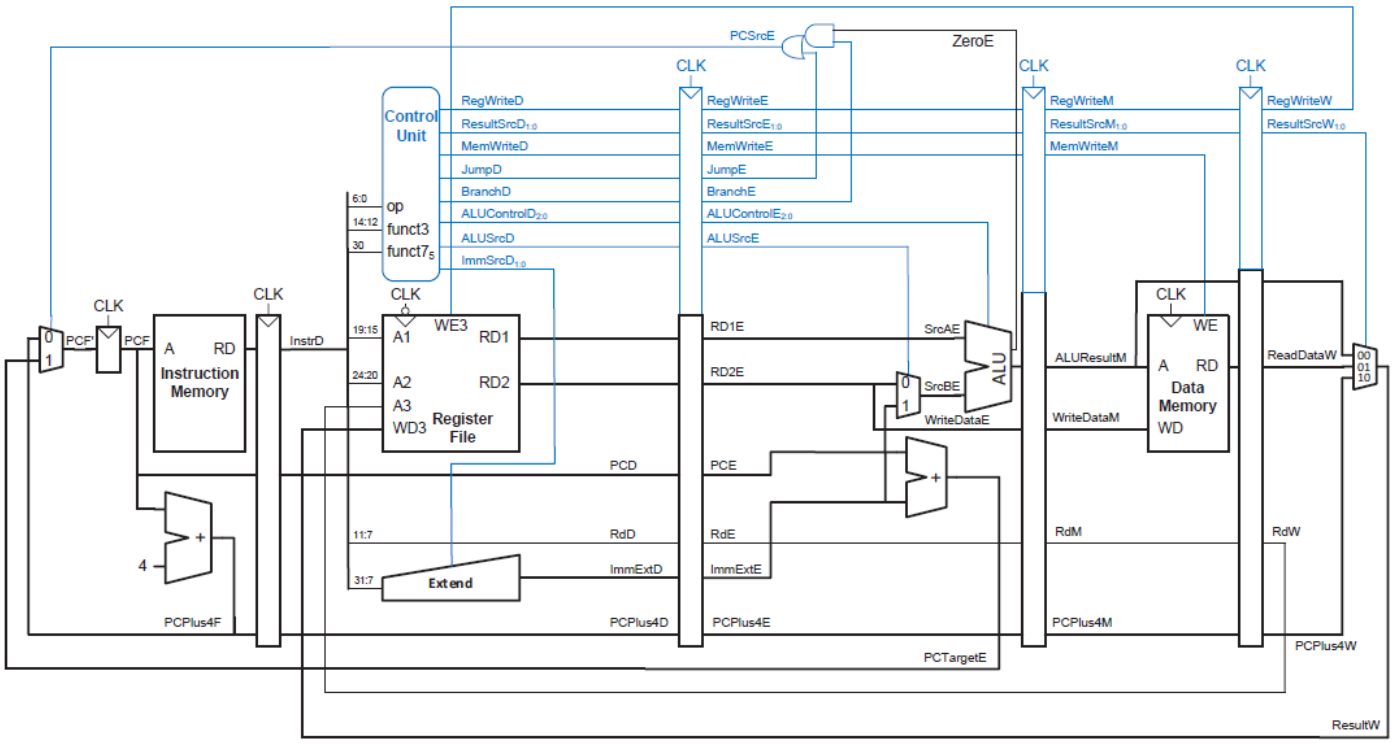


Figure67: Pipelined processor with control

► Core Functionality of the Pipelined Processor :

- The pipelined core follows the 5-stage RISC-V microarchitecture, stated earlier to maintain high throughput, multiple instructions flow through the pipeline simultaneously, each at a different stage :

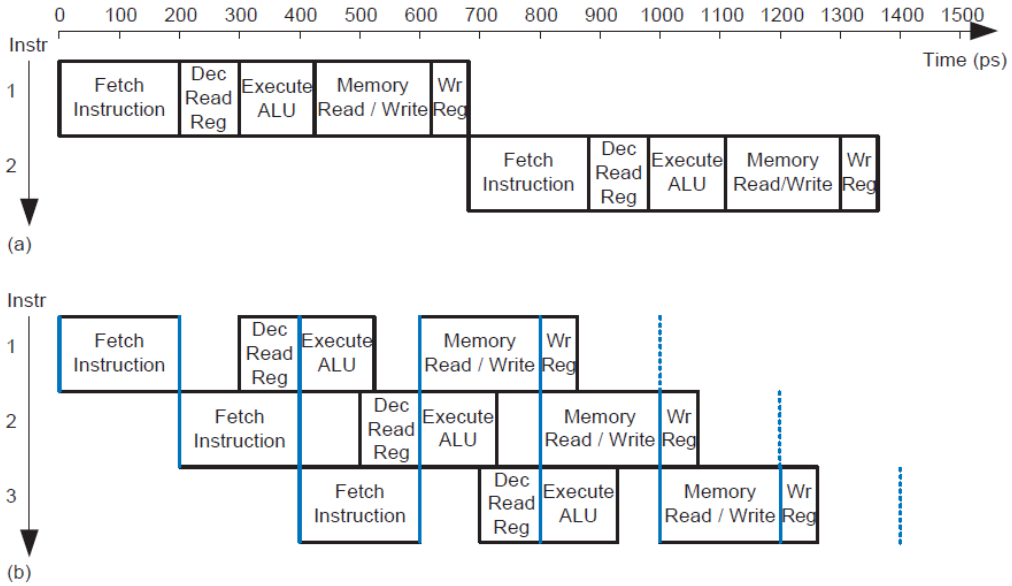


Figure68: Timing diagrams: (a) single-cycle processor and (b) pipelined processor

► Synchronized Progression of Signals :

- One **critical architectural principle** in pipelining is that **every signal related to an instruction must advance together** from stage to stage. This includes:

- Data signals (like register values, immediate values),
- Destination register identifiers (`Rd`),
- Control signals (e.g., `RegWrite`, `ALUSrc`, `MemWrite`, etc.).

- This synchronization ensures that:

- The correct instruction writes to the correct register at the correct time,
- There are no mismatches between computed data and the destination register or memory.

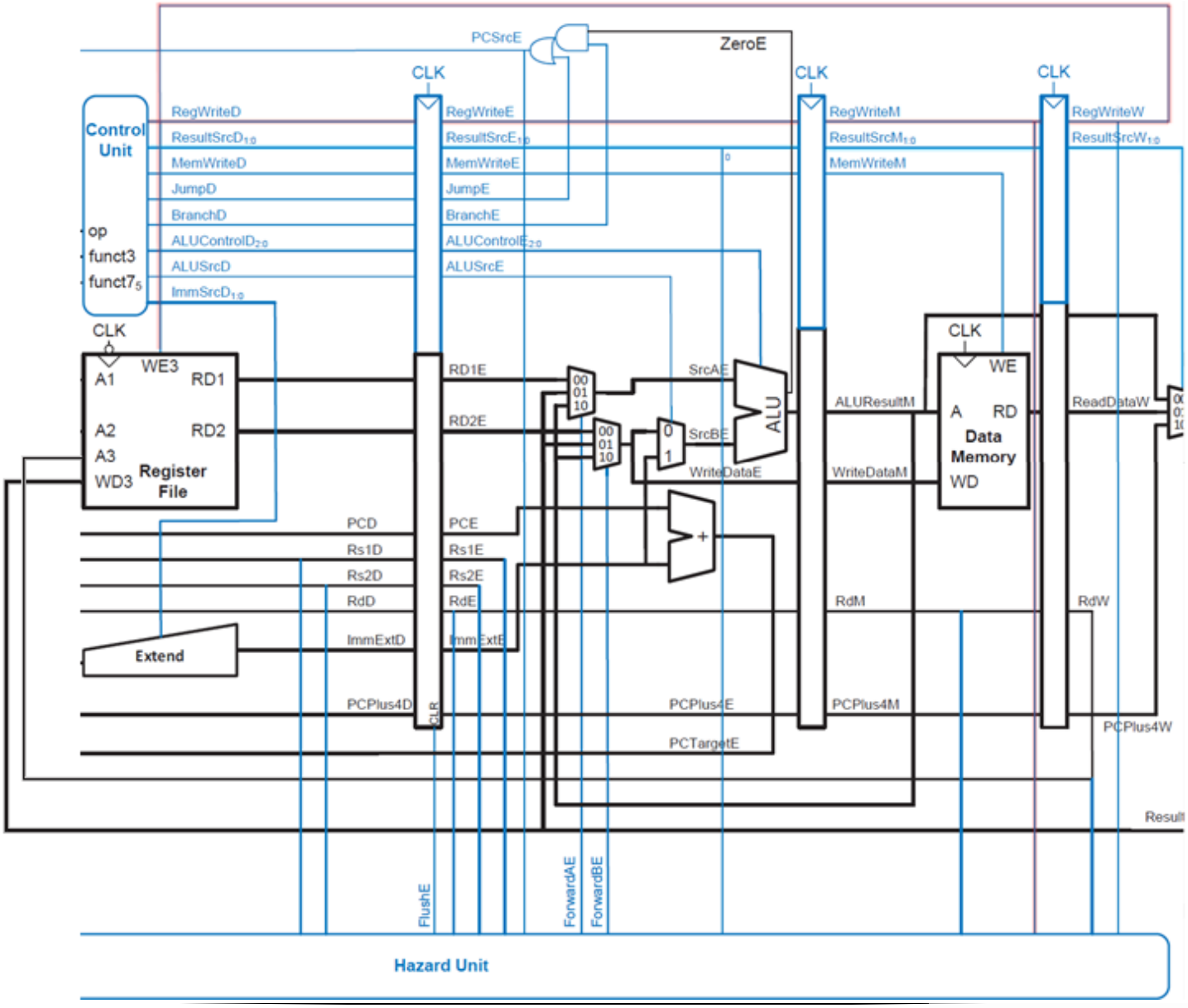


Figure69: Synchronized signal registering throughout the pipeline to preserve correct signaling and data

♦ **Example Bug (from textbook):**

→ If you feed `RdD` (Decode stage) to the register file in the WB stage instead of `RdW` (properly pipelined), you risk **writing back a result to the wrong register**.
E.g., a `lw x2, 0(x3)` writes back the value to `x5` (from a later instruction) instead of `x2`.

→The fix is shown in **Figure69**: **pipeline `Rd` along with data** through EX → MEM → WB, so at Writeback, both the `ResultW` and its corresponding `RdW` match correctly.

► Register File Write Timing (CLK Edge Behavior) :

- The **register file writes on the falling edge of the clock** and is read on the rising edge. This means:

- An instruction can **write a value** to a register during the **first half of a clock cycle**,
- A **subsequent instruction in the same cycle** can **read back that freshly written value** during the **second half**,
- Thus, in some cases, **no hazard exists**, even without forwarding.

→ However, this only works **if the dependent instruction is at least one stage behind**. Otherwise, a data hazard arises.

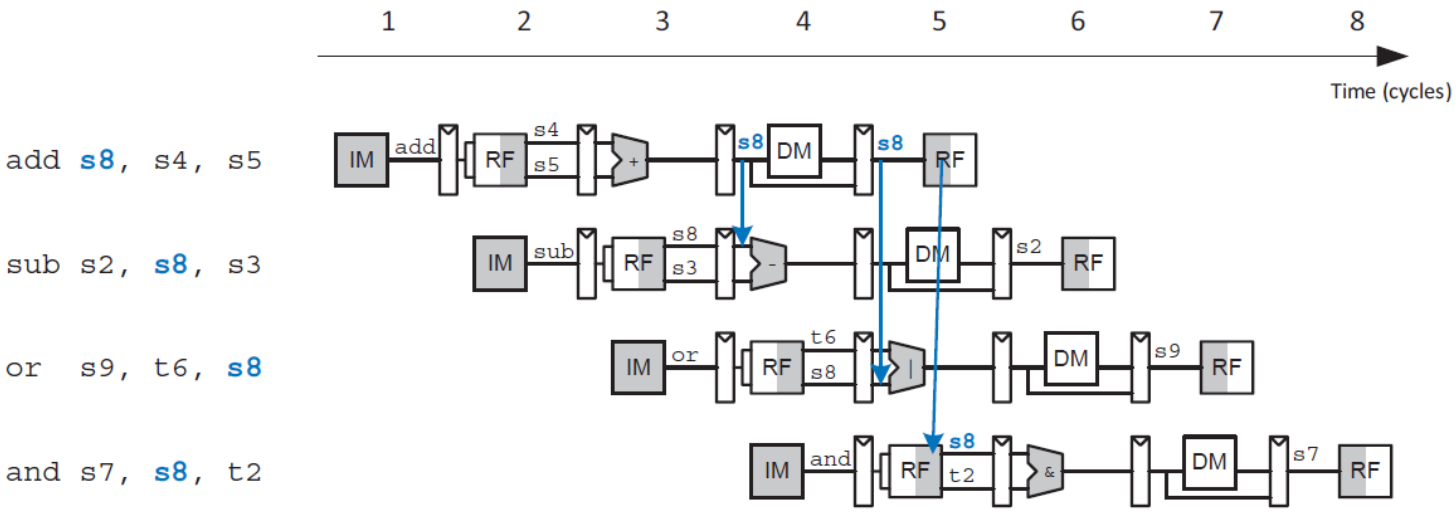


Figure70: Abstract pipeline diagram illustrating forwarding

►Data Hazards and RAW (Read-After-Write) :

➤ RAW Hazard Example:

```
add  s8, s1, s2    # Writes to s8 in cycle 5

sub  s4, s8, s3    # Reads s8 in cycle 3 (too early, gets old value)

or   s5, s8, s6    # Reads s8 in cycle 4 (still too early)

and  s7, s8, s9    # Reads s8 in cycle 5 (register file write
completed , on falling edge)
```

- The first two dependent instructions (`sub`, `or`) read the old value of `s8`. This is a **classic RAW hazard.**

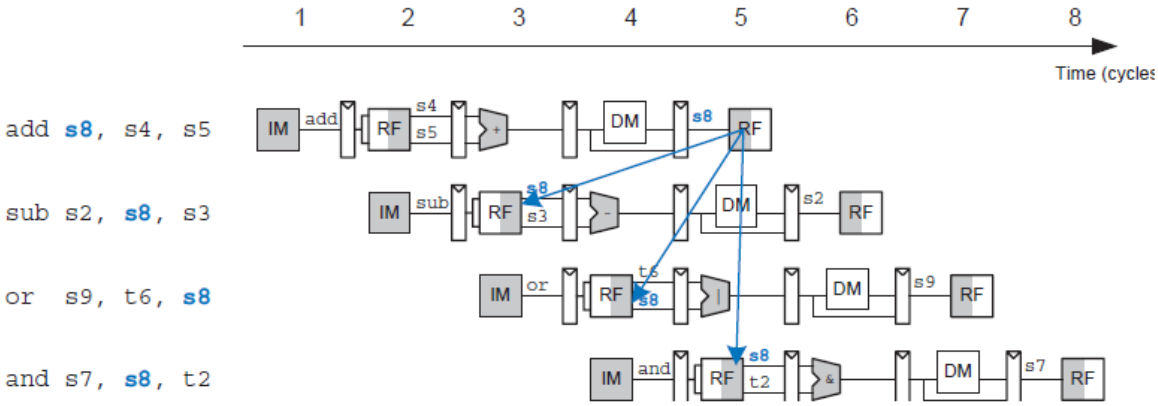


Figure 7.52 Abstract pipeline diagram illustrating hazards

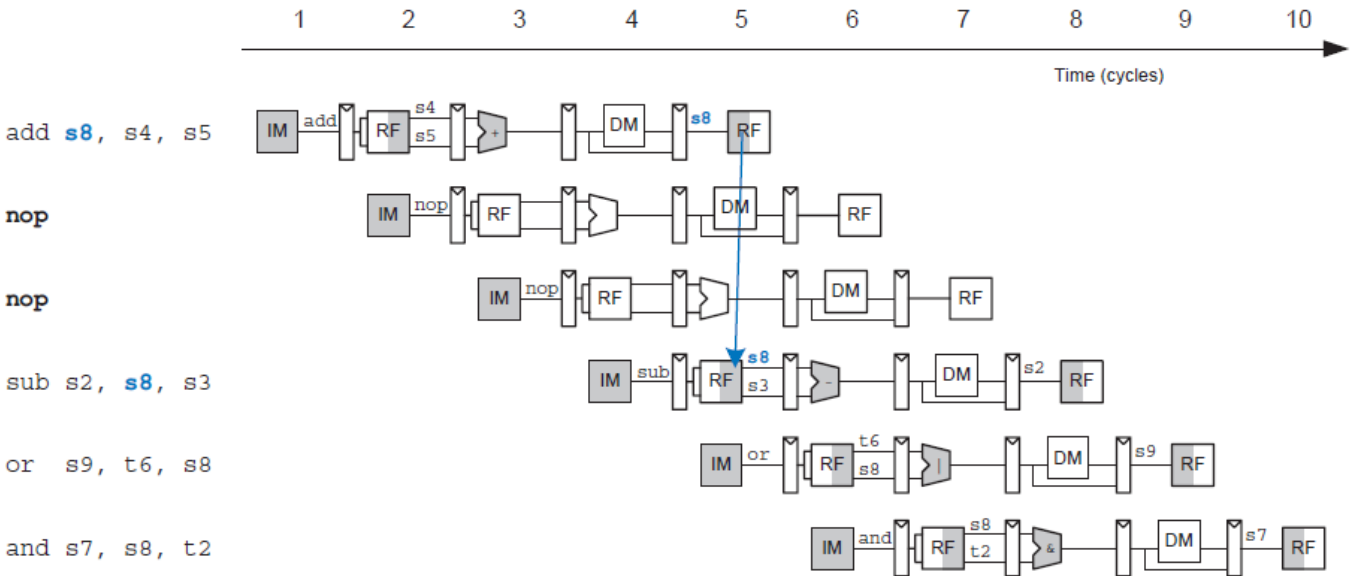


Figure71: RAW hazards resolved by injecting NOP's to increase latency so that data is writtenBack to the registerFile

► Hazard Resolution Mechanisms :

1. Register Forwarding (Bypassing) :

- Forward ALU or memory results **before they reach WB stage**, directly into the EX stage of the next instruction.
- This avoids having to wait for register writeback.
- Common sources:
  - `ALUResultE` → EX stage (EX-EX forwarding),
  - `ALUResultM` → EX stage (MEM-EX),
  - `ReadDataW` → EX stage (MEM-WB-EX).

*view **Figure70***

2. Stalling (Pipeline Interlock) :

- If the previous instruction is a **`lw`**, the data is only available at the **end of MEM stage**.
- In this case, the next instruction **cannot execute in EX stage yet**, even with forwarding.
- The hazard detection unit inserts a **bubble (NOP)** by stalling the pipeline for one cycle.

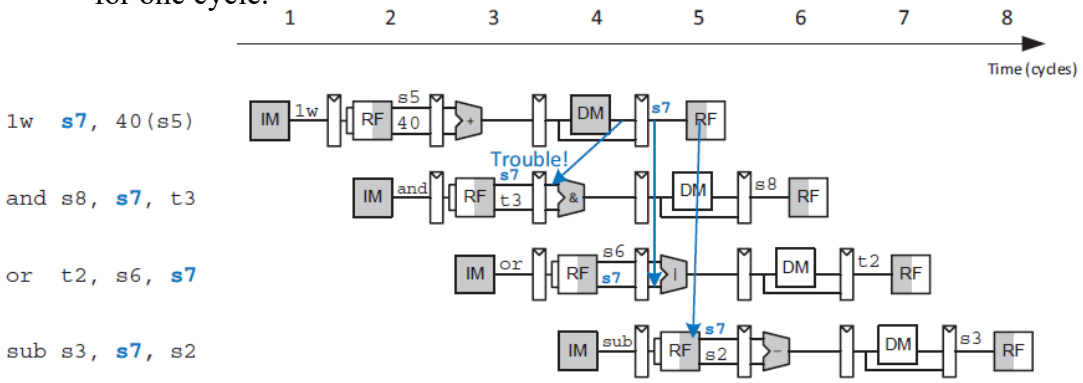


Figure 7.56 Abstract pipeline diagram illustrating trouble forwarding from `lw`

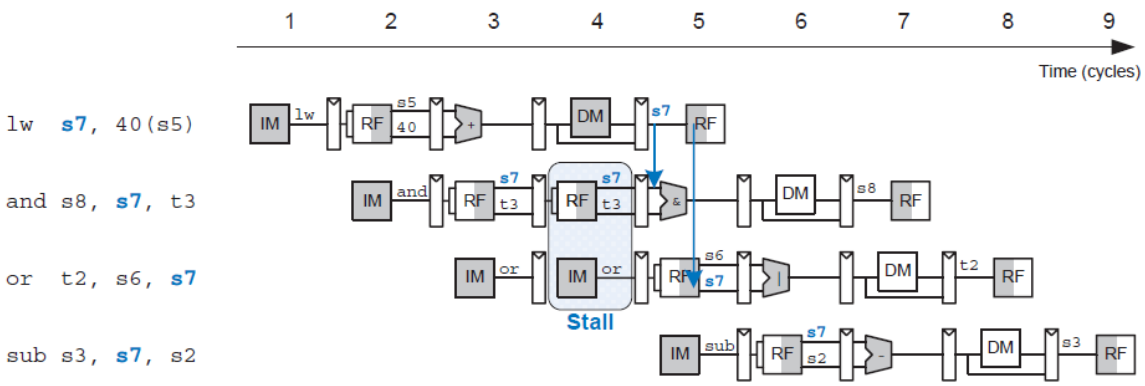


Figure72: Mandatory 1 cycle for the `lw` instruction (Read after load)

3. Hazard Detection Logic :

- Compares source registers in ID/EX stage with `RdE` or `RdM`.
- If it detects that data isn't ready, it:
  - Freezes the PC and IF/ID pipeline register,
  - Inserts a bubble into the EX stage,
  - Avoids incorrect computation.

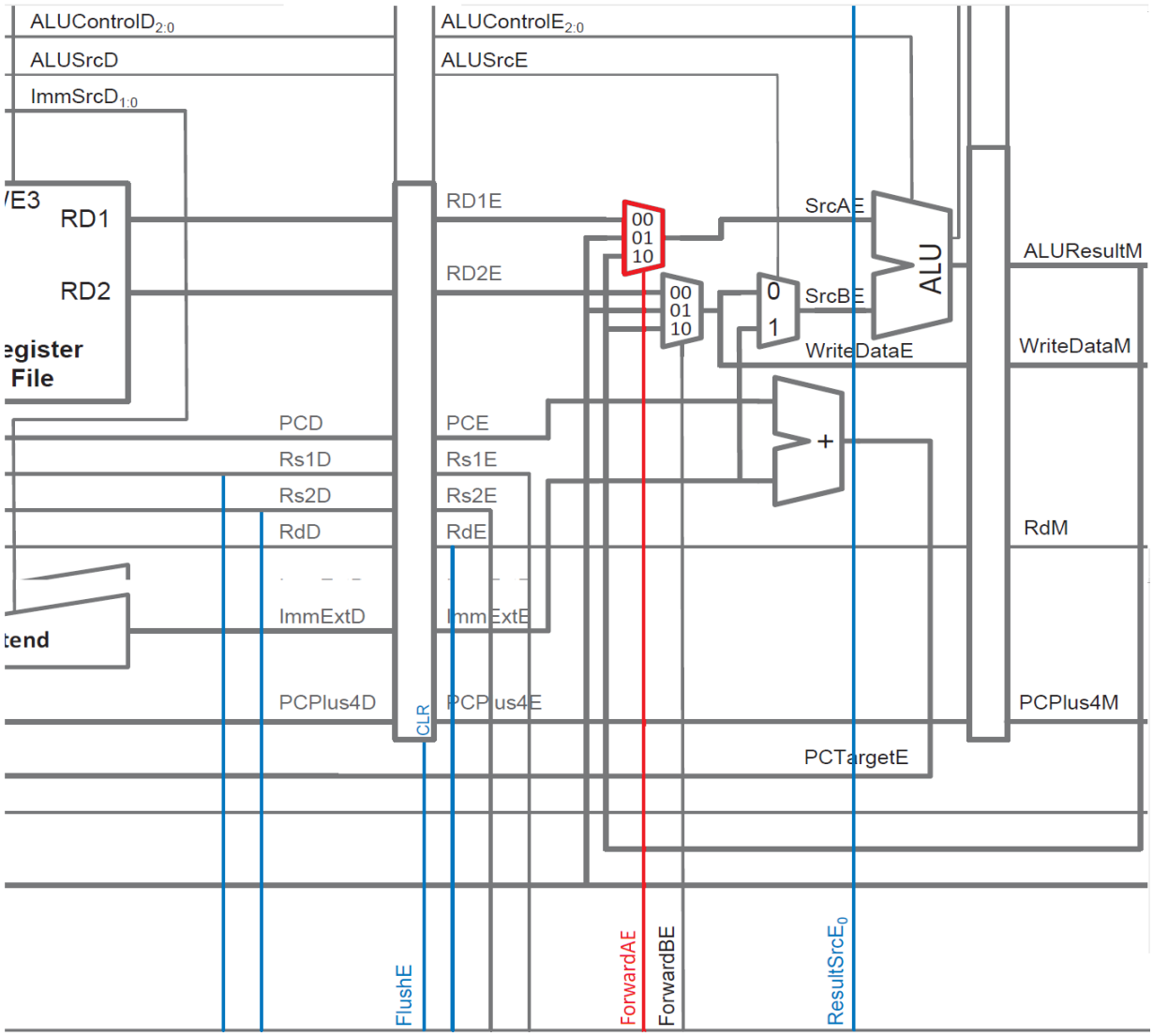
if ((*Rs1E* == *RdM*) & *RegWriteM*) & (*Rs1E* != 0) then // **Forward from Memory stage**

ForwardAE = 10

else if ((*Rs1E* == *RdW*) & *RegWriteW*) & (*Rs1E* != 0) then // **Forward from Writeback stage**

ForwardAE = 01

else ***ForwardAE*** = 00 // **No forwarding (use RF output)**



Hazard Unit

Figure73: Hazard unit forwarding signal for the execution stage to resolve RAW hazards

$lwStall = ResultSrcE_0 \,\&\, ((Rs1D == RdE) \mid (Rs2D == RdE))$

$StallF = StallD = FlushE = lwStall$

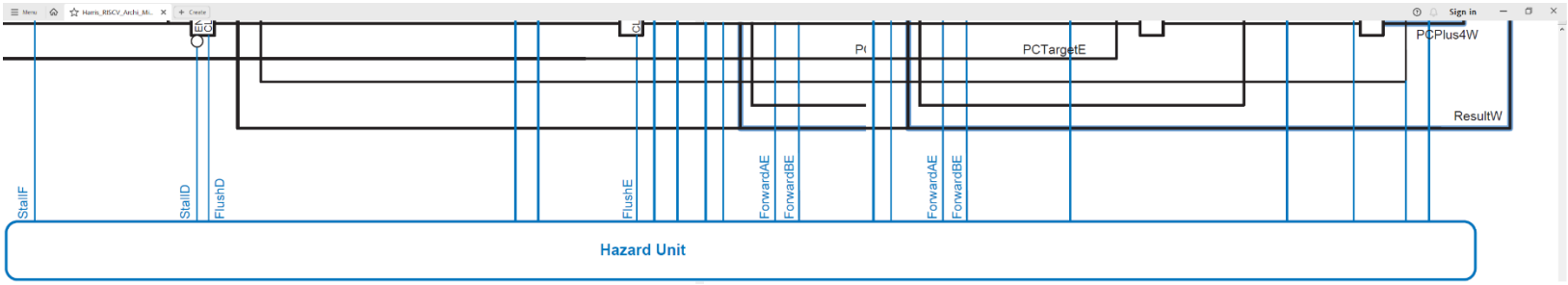


Figure74: Hazard unit forwarding signals for to resolve RAL hazards

► Control Hazards (Branching and Jumping) :

- Branches (like `beq`) or jumps (`jal`) introduce **control hazards**, because:

  - The next instruction is fetched before knowing if the branch is taken.

→ Basic fix (as in BraRV32) :

- Calculate the **branch decision in the EX stage**, and use **PCSrcE** to select the correct PC (either `PC + 4` or `PCTargetE`).
- Flush the instruction fetched in cycle after the branch (insert bubble) to avoid executing the wrong path.

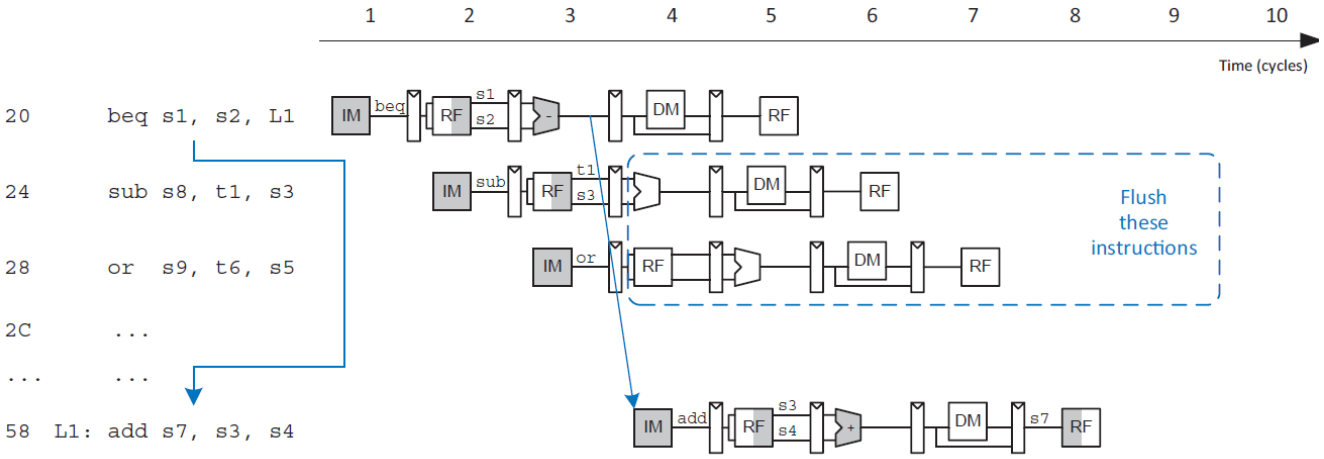


Figure75: Abstract pipeline diagram illustrating flushing when a branch is taken

– Flush when a branch is taken or a load introduces a bubble :

$FlushD = PCSrcE$

$FlushE = lwStall \mid PCSrcE$

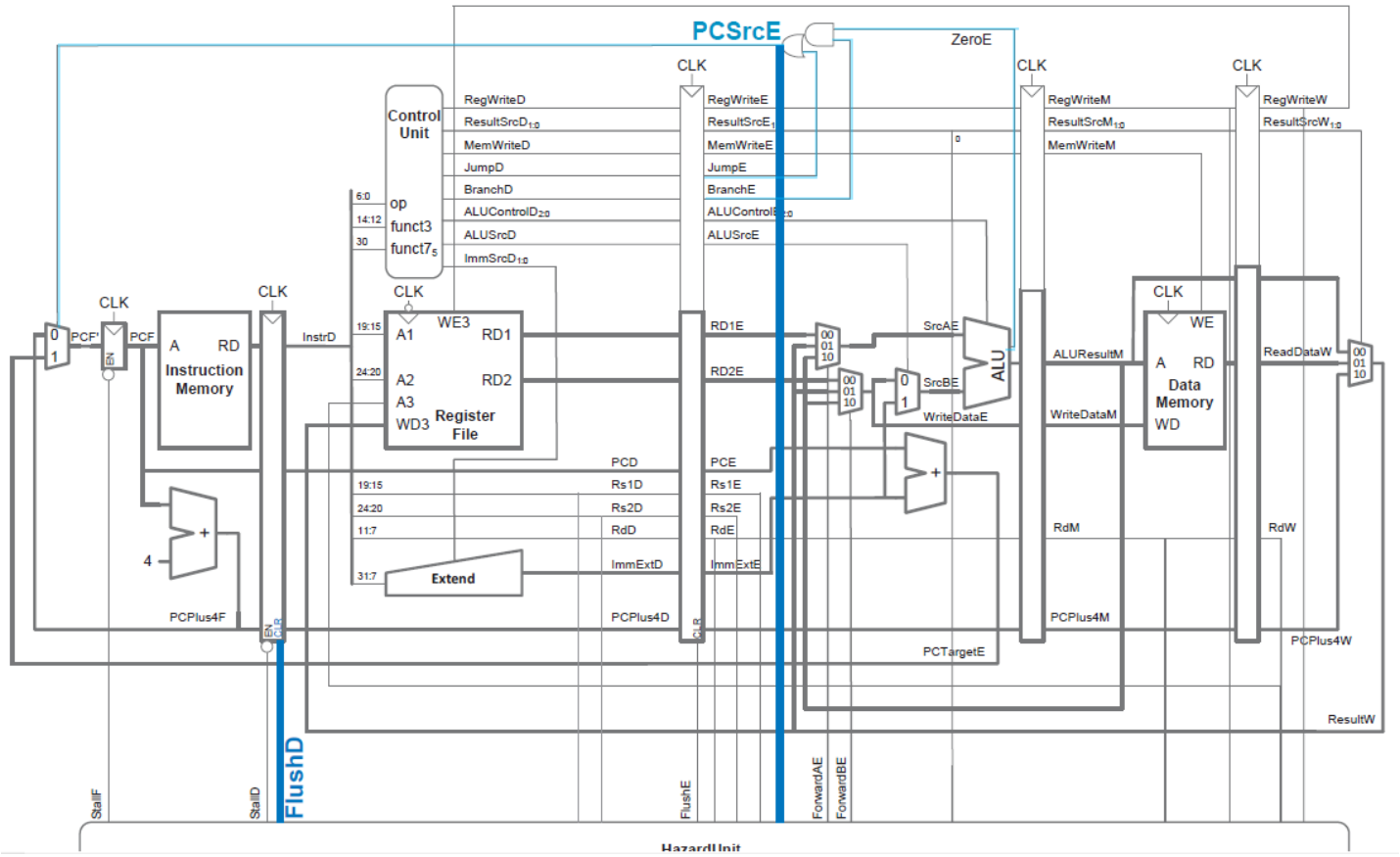


Figure76: Expanded Hazard Unit for handling branch control hazard

► Summary → Core Pipeline Synchronization Challenges :

Challenge	Solution
Signals not advancing in sync	Pipeline data, control, and `Rd` signals
Register write happens too late	Forward result early (bypassing)
`lw` instruction delay	Stall one cycle
Branch decision delay	Flush misfetched instruction

Table5 : Pipeline hasseles and the adopted solutions

5.3 Simulation and Verification :

- To validate the functional correctness and behavior of the Pipelined core, a structured simulation was carried out using **ModelSim**. This focused on verifying instruction decoding, control unit signal generation, hazard unit verification, state transitions, and memory interactions under realistic test conditions.

- The simulation is performed on the device under test using a recursive Fibonacci assembly program that calculates the 31st Fibonacci number.

- The code :

```
main:
        addi a2, zero, 31          # a2 = 31 (argument to fib)
        jal ra, fib                # call fib(a2 = 31)
        jalr zero, ra, 0           # return from main

    fib:
        addi sp, sp, -12           # allocate 12 bytes on stack
        sw ra, 8(sp)               # save return address
        sw s0, 4(sp)               # save s0

    fib_if_x0_or_one:
        addi a0, zero, 0           # a0 = 0
        beq a2, zero, fib_fin      # if a2 == 0, return
        addi a0, a0, 1             # a0 = 1
        beq a2, a0, fib_fin        # if a2 == 1, return

    fib_call_n_1:
        addi a2, a2, -1            # a2 = a2 - 1
        sw a2, 0(sp)               # save current a2
        jal ra, fib                # recursive call fib(n-1)
        lw a2, 0(sp)               # restore a2

        addi a2, a2, -1            # a2 = a2 - 2
                                   #(already -1,now -2)

    fib_call_n_2:
        add s0, a0, zero           # s0 = fib(n - 1)
        jal ra, fib                # recursive call fib(n - 2)
        add a0, a0, s0             # a0 = fib(n - 1) + fib(n - 2)

    fib_fin:
        lw s0, 4(sp)               # restore s0
        lw ra, 8(sp)               # restore return address
        addi sp, sp, 12            # deallocate stack
        jalr zero, ra, 0           # return
```

n	Fibonacci (n)
0	0
1	1
2	1
3	2
4	3
5	5
6	8
7	13
8	21
9	34
10	55
11	89
12	144
13	233
14	377
15	610
16	987
17	1597
18	2584
19	4181
20	6765
21	10 946
22	17 711
23	28 657
24	46 368
25	75 025
26	121 393
27	196 418
28	317 811
29	514 229
30	832 040
31	**1 346 269**

• Recursive Fibonacci sequence implementation in RISC-V assembly •

- Before we dive into the methodology, approach and finally the results of the simulation it is necessary to take a look at the type of memory used it's Verilog implementation and how it ties to the Pipelined core, as the single-cycle processor it is necessary to separate the data and instruction memories for the correct implementation of this architecture.

Data Memory

```
module data_memory
    #(parameter MEMORY_SIZE = 64)
    (
        output reg [31:0] ReadData,
        input             RESET, CLK, WriteEnable,
        input      [31:0] Address, WriteData,
        input      [2:0]  size
    );

    reg [7:0] MEMORY[MEMORY_SIZE-1:0];
    integer i;
    reg [15:0] data_temp;

    // Size encoding
    localparam BYTE        = 3'b001;
    localparam HALFWORD    = 3'b010;
    localparam WORD        = 3'b000;
    localparam BYTE_U      = 3'b011;
    localparam HALFWORD_U  = 3'b100;


    always @(posedge CLK or posedge RESET) begin
        if (RESET) begin

for (i = 0; i < MEMORY_SIZE; i = i + 1)
    MEMORY[i] <= 8'b00000000;
end

    //end code
        else if (WriteEnable) begin
            case (size)
                BYTE: begin
                    MEMORY[Address] <= WriteData[7:0];
                end
                HALFWORD: begin
                    MEMORY[Address]     <= WriteData[15:8];
                    MEMORY[Address + 1] <= WriteData[7:0];
                end
                WORD: begin
                    MEMORY[Address]     <= WriteData[31:24];
                    MEMORY[Address + 1] <= WriteData[23:16];
                    MEMORY[Address + 2] <= WriteData[15:8];
                    MEMORY[Address + 3] <= WriteData[7:0];
                end
            endcase
        end
    end

    // Read logic
    always @(*) begin
        case (size)
            BYTE: begin
                data_temp = MEMORY[Address];
                ReadData  = {{24{data_temp[7]}}, data_temp[7:0]}; // Sign-
extend byte
```

```
            end
            HALFWORD: begin
                data_temp = {MEMORY[Address], MEMORY[Address + 1]};
                ReadData  = {{16{data_temp[15]}}, data_temp};     // Sign-
extend halfword
            end
            WORD: begin
                ReadData = {MEMORY[Address], MEMORY[Address + 1],
                            MEMORY[Address + 2], MEMORY[Address + 3]};
            end
            BYTE_U: begin
                data_temp = MEMORY[Address];
                ReadData  = {24'b0, data_temp[7:0]};               // Zero-
extend byte
            end
            HALFWORD_U: begin
                data_temp = {MEMORY[Address], MEMORY[Address + 1]};
                ReadData  = {16'b0, data_temp};                    // Zero-
extend halfword
            end
            default: begin
                ReadData = 32'b0;
            end
        endcase
    end

endmodule
```

Instruction Memory

```
module instruction_memory
    #(parameter MEMORY_SIZE = 128)
    (
        output reg [31:0] ReadData,
        input             RESET, CLK,
        input      [31:0] Address
    );

    reg [7:0] MEMORY[MEMORY_SIZE-1:0];
    integer i;

initial begin
        $readmemb("../rtl/full_synthesis_program.tv", memory);
    end
    // ! ../memory_init.mif paths only works with simulation functions like
$readmemh !

   always @(*) begin
           mem_rdata <= memory[mem_addr[31:2]];
   end



    // Write on rising clock edge if write mask active
    always @(posedge clk) begin
            for (i = 0; i < 4; i = i + 1) begin
                if (|mem_wmask[i]) begin
```

```
                    memory[mem_addr[31:2]][8*i +: 8] <= mem_wdata[8*i +: 8];
                end
            end
    end
endmodule
```

- To support instruction and data memory operations in the Pipelined processor, a seperate memory model `data_memory & instruction_memory` was implemented (**Figure67**). it serves as a simple synchronous RAM's with both read and byte-masked write support. The instruction memory also includes a flexible initialization mechanism to preload test programs at specific addresses during simulation.

♦ **Features :**

- **Parameterized Memory Size:**
  The memory consists of `MEM_WORDS` 32-bit words (default: 1024) for data, and (default: 128) for instruction memory allowing scalable capacity for program and data storage.
- **Synchronous Read Operation:**
  For simplicity on the rising edge of the clock, a 32-bit word is read from the address `mem_addr[31:2]`the clock period is sufficient enough to carry out the operation this simple approach makes it possible to refetch instructions when stalling occurs. Word alignment is achieved by discarding the two least significant bits.
- **Masked Write Support:**
  The write port supports partial updates through `mem_wmask`, where each bit enables writing of one byte in the target word. This simulates byte-granular memory writes (`sb`, `sh`, `sw`) and is useful for RISC-V's `mem_wmask` behavior.

♦ **Simulation-Specific Features :**

- **Test Program Preloading via Label File:**
  During simulation, the memory is initialized based on external `.tv` files (text vector files) using labels defined in a `labels.txt` file, generated by a python script before assembling (more on that later). Each line in the file associates a label (e.g., `"main"`, `"fib"`) with a memory address (here we implement a `.mif` parser to embed the whole program in a singular `.tv` file due to the inability of using Simulation specific funtions like `$fscan`,`$fopen`… in synthesis). When the corresponding `.tv` file is loaded into the `memory[]` .

- The labels addresses are handeled before the generation of the final machine code executable by the parser.

→ **Summary:**

- The `data_memory & instruction_memory` modules combines functionality and flexibility tailored for simulation and verification for this Pipelined RISC-V processor. It provides a reliable abstraction for both instruction and data memory, while supporting structured and label-based test program loading .

♦ Simulation Methodology:

- **Testbench Structure:**
  A dedicated Verilog testbench was written to instantiate the Pipelined core, provide initial signal setup (clock, reset), and drive memory-mapped instruction and data stimuli.
- **Clock and Reset Control:**
  A controlled clock with a defined frequency was used, alongside a single reset pulse to ensure deterministic FSM startup , Hazards detection and state initialization.
- **Program Initialization:**
  RISC-V assembly test cases were written, assembled using ~~an external toolchain~~ a pseudo RISC-V assembler made with python for the scope of this project which we will discuss in later chapters, after that we load it into the instruction memory in `.tv` format (`.hex` won't be read by verilog's read memory functions $readmemh & $readmemb) .
- **Instruction Coverage:**
  The simulation suite tested all key instruction types:
  - **Arithmetic & Logic:** `add`, `sub`, `xor`, `slli`
  - **Load/Store:** `lw`, `lh`, `lhu`, `lb`, `sw`
  - **Control Transfer:** `beq`, `jal`, `jalr`
  - **Immediate/Upper:** `lui`, `auipc`, `addi`

♦ Verification Objectives:

- **FSM Behavior and Hazard Unit Validation:**

- To ensure correct **pipeline control and data hazard handling**, we approached verification with a mix of **simulation** and **on-hardware observation** using **SignalTap**.

- **FSM Behavior**:
  - We verified that the control FSM transitions cleanly through pipeline stages (IF → ID → EX → MEM → WB) for various instruction types. In simulation, we checked state sequences and control signal activations. On hardware, SignalTap was used to monitor state transitions in real time while running instruction sequences.
- **Hazard Detection**:
  - We created specific instruction sequences designed to trigger common hazards:
    - **RAW hazards** between ALU or load instructions
    - **Load-use stalls**

- **Branch/jump flushes**
  SignalTap helped visualize control signals like `stall`, `flush`, and bypass mux selectors, confirming that the hazard unit responded appropriately.

- **Bypassing Paths** were tested by checking that dependent instructions received the correct - forwarded values from EX or MEM stages instead of stale register file data.
- **Reset and edge cases** (e.g., back-to-back branches or NOPs) were tested to ensure FSM - recovery and stability.

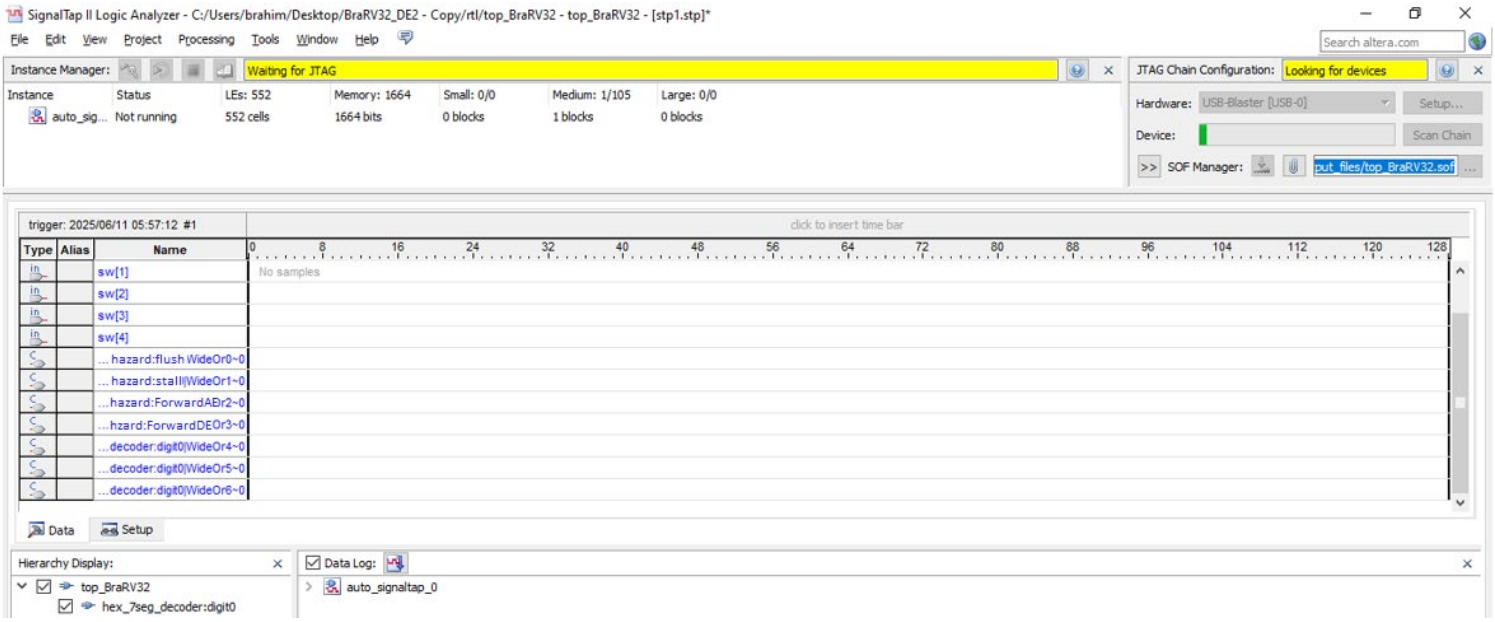


Figure77: SignalTap II Logic Analyzer Window interface (Pipelined core)

Figure78: SignalTap II Logic Analyzer Live Data Monitoring (Pipelined core)

5. **Control Signal Timing:**
   - Ensured signals like `writeBack`, `isLoad`, `stall`, `flush`, `ForwardXYZ… ect`, and

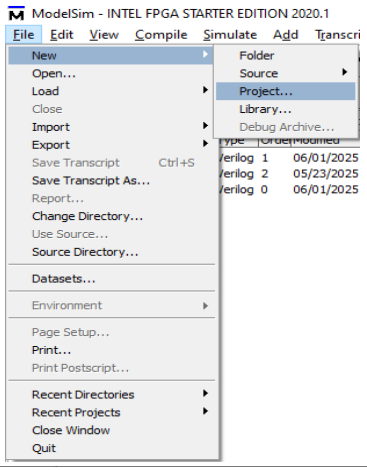
`isBranch` toggled as expected based on instruction type and internal decoding. Special attention was given to load/store alignment and jump handling as it was the most vulnerable for errors and undefined behavior just like BraRV32 since **the memory** is **word addressable** and the **address calculations** executed inside Pipelined core are **byte-addressable** oriented.

6. **Datapath Observability:**
   - Just like earlier cores Tracked ALU outputs, register write-backs (`rdId`), and memory accesses for correctness. Byte and halfword alignment logic for loading data and the store mask were verified over multiple unaligned accesses.
7. **End-to-End Consistency Checks:**
   - Register file values and memory contents were compared to reference outputs post-execution. The pipeline maintained correctness even under mixed instruction sequences.

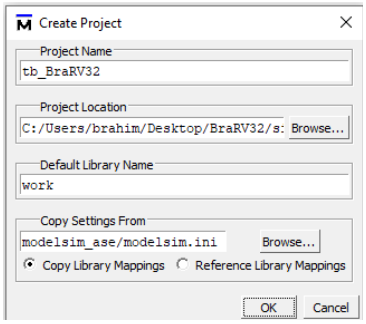
♦ Simulation Tools:

- **Environment:** ModelSim
- **Waveform Analysis:**
  - Internal signals were probed using waveform viewer and VCD traces. Timing and data propagation across control and datapath blocks were reviewed over several simulation cycles to ensure effective communication between the two.

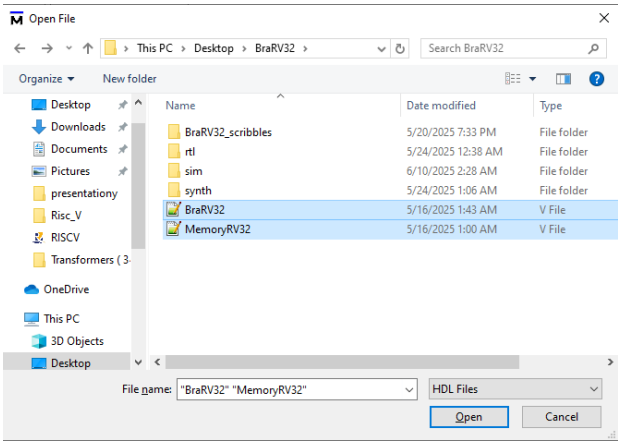
- First we create a project in Modelsim



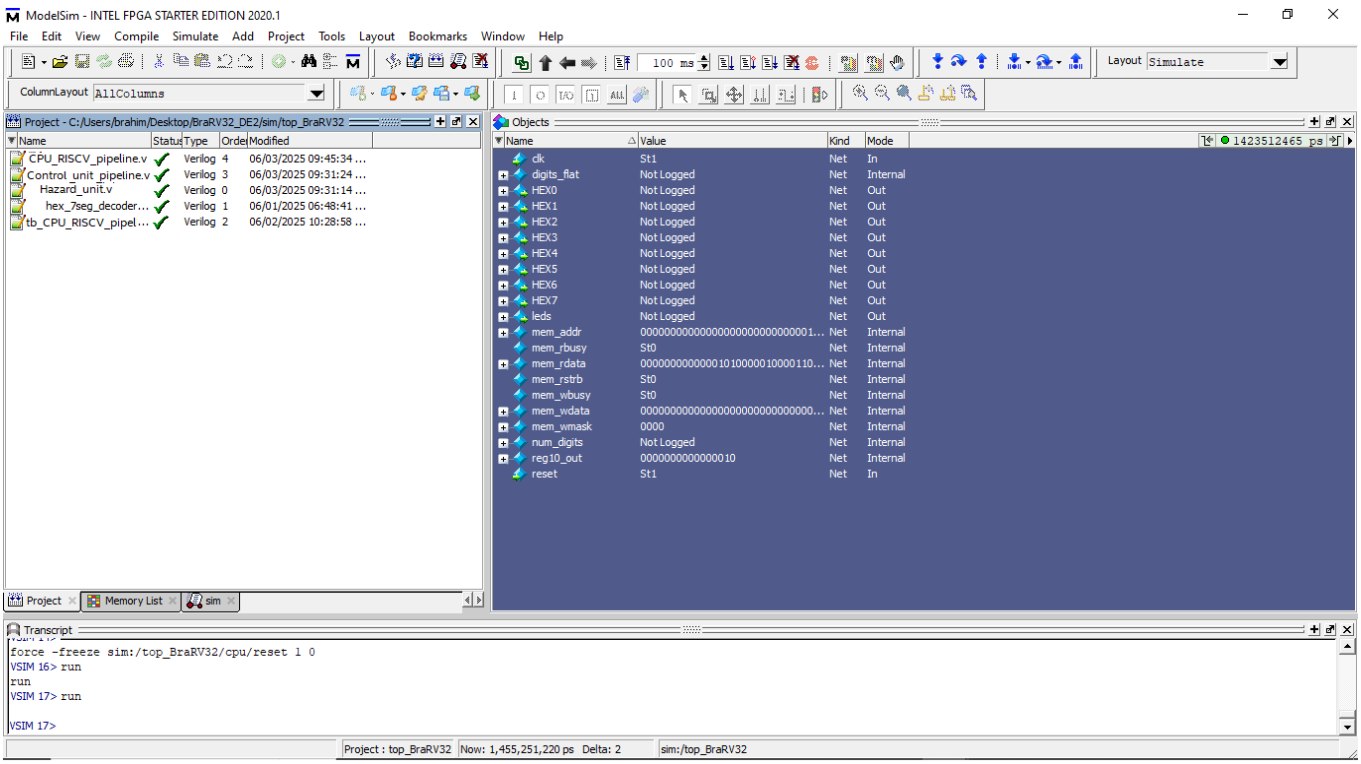
- Give it a name and press OK



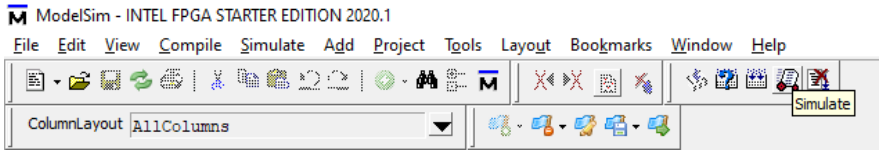
- add the necessary HDL files



- Compile the design

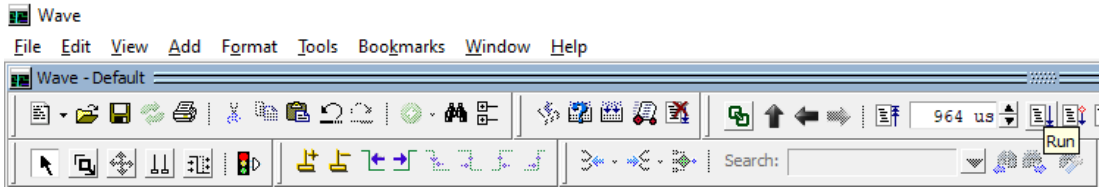


- Simulate



- Carry on the steps as mentioned in section 4.

- Run



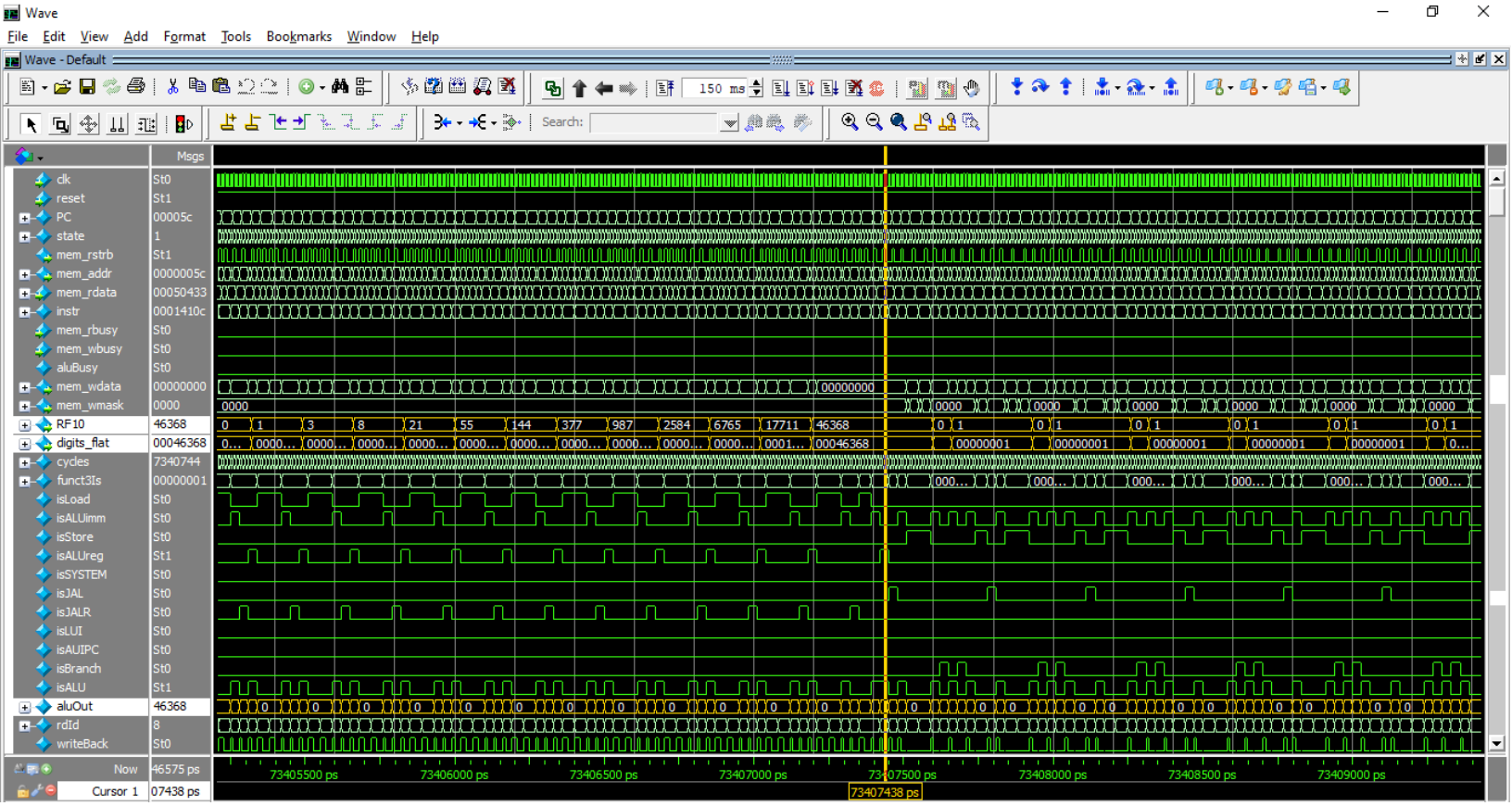


Figure79 : Simulation results of Pipelined core running recursive Fibonacci[31]

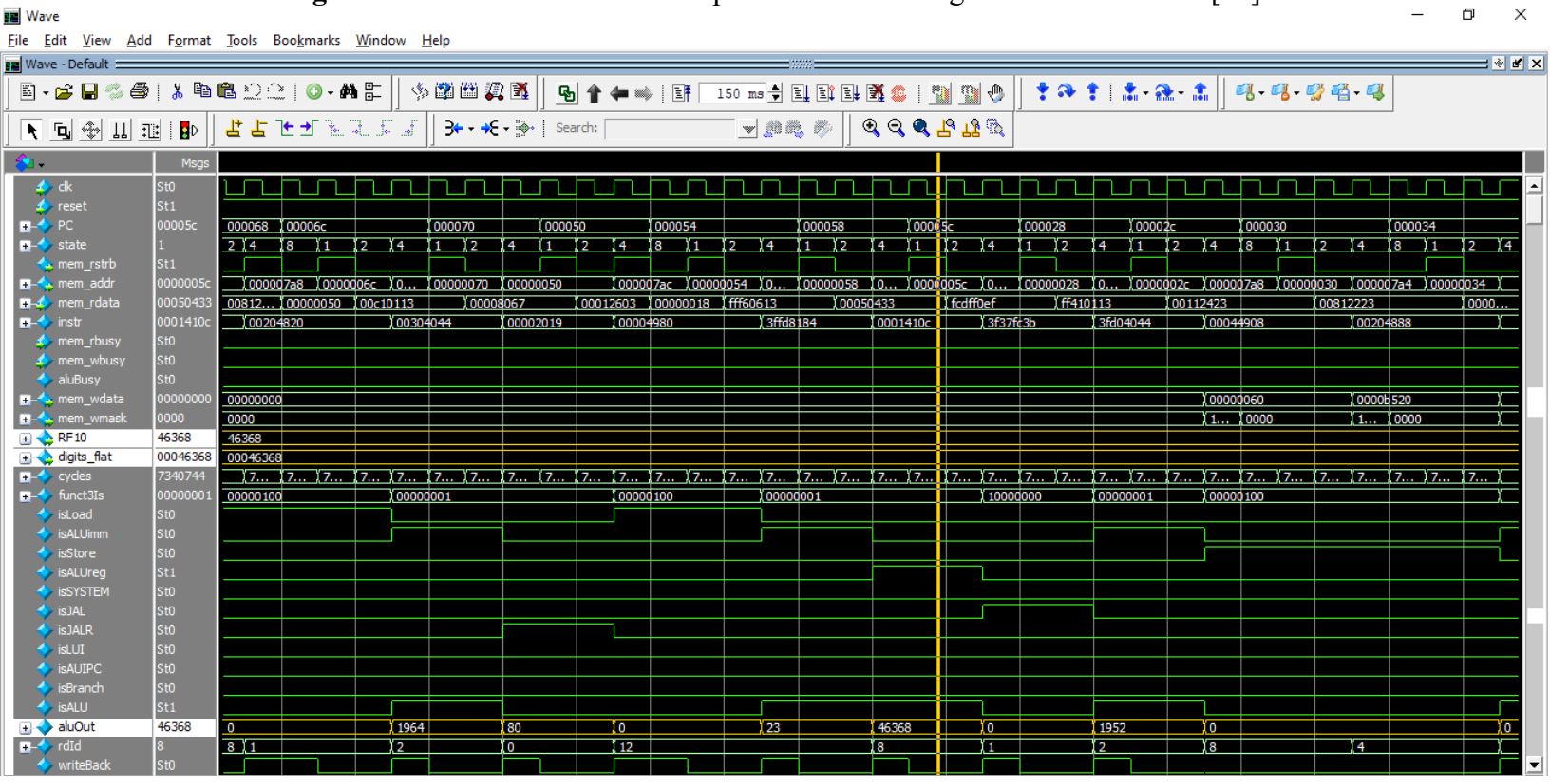


Figure80 : Simulation results of Pipelined running recursive Fibonacci[31]…***Zoomed in***

```
Transcript
# Load canceled
ModelSim> vsim -gui work.tb_BraRV32
# vsim -gui work.tb_PipelineRV32
# Start time: 02:40:39 on Jun 10,2025
# Loading work.tb_PipelineRV32
# Loading work.PipelineRV32
# Loading work.PipelineRV32
do C:/Users/brahim/Desktop/BraRV32/Pipeline_Fibonacci_31.do
VSIM 5> run
# Line 0: Label =                          main, Address = 0
# ana f main
# Line 1: Label =                           fib, Address = 10
# ana f fib
# Simulation finished.
Project :   Now: 1,005,040 ns  Delta: 2   sim:/tb_
```

Figure81 : Modelsim run simulation console results

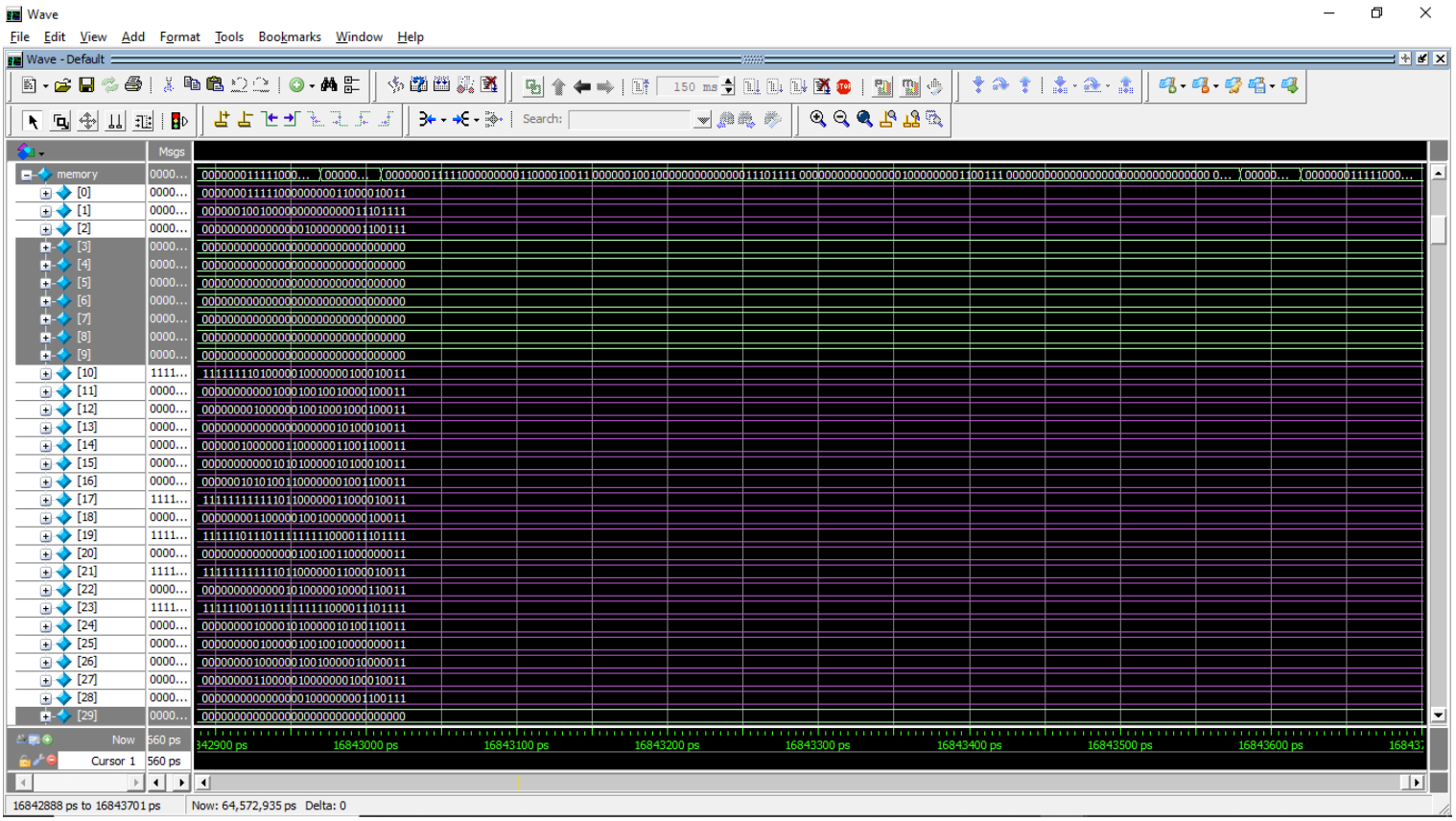


Figure82 : instructions in memory with each label in it's given address main : [0] → 0x00 & fib : [10] → 0xA0 (**[word @ddressable] → Byte @ddressable**)

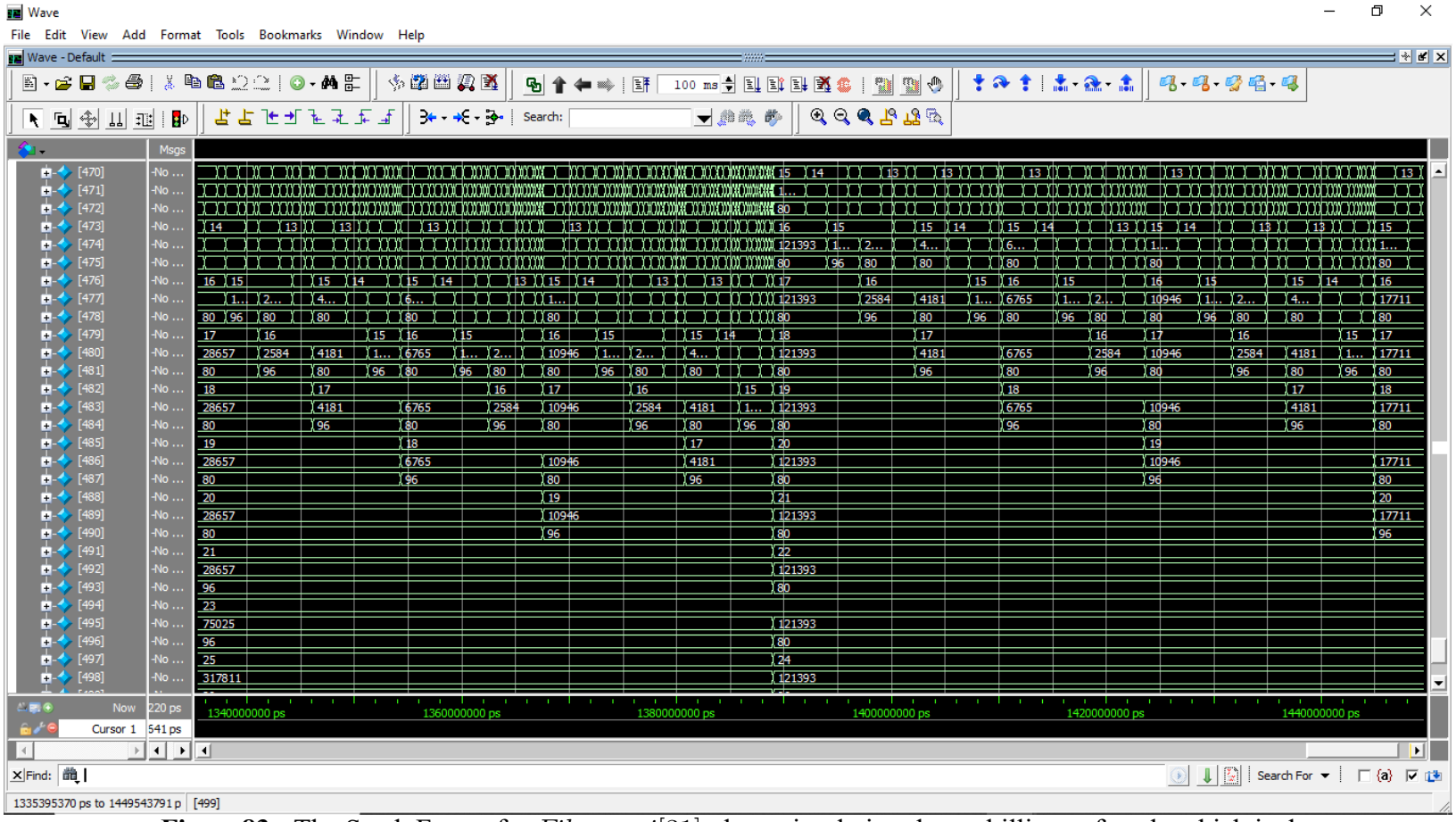


Figure83 : The Stack Frame for $Fibonacci[31]$, long simulation due to billions of cycle which is due to the nature of the recursive call of the function which is of the magnitude/time complexity of $O(2^n)$

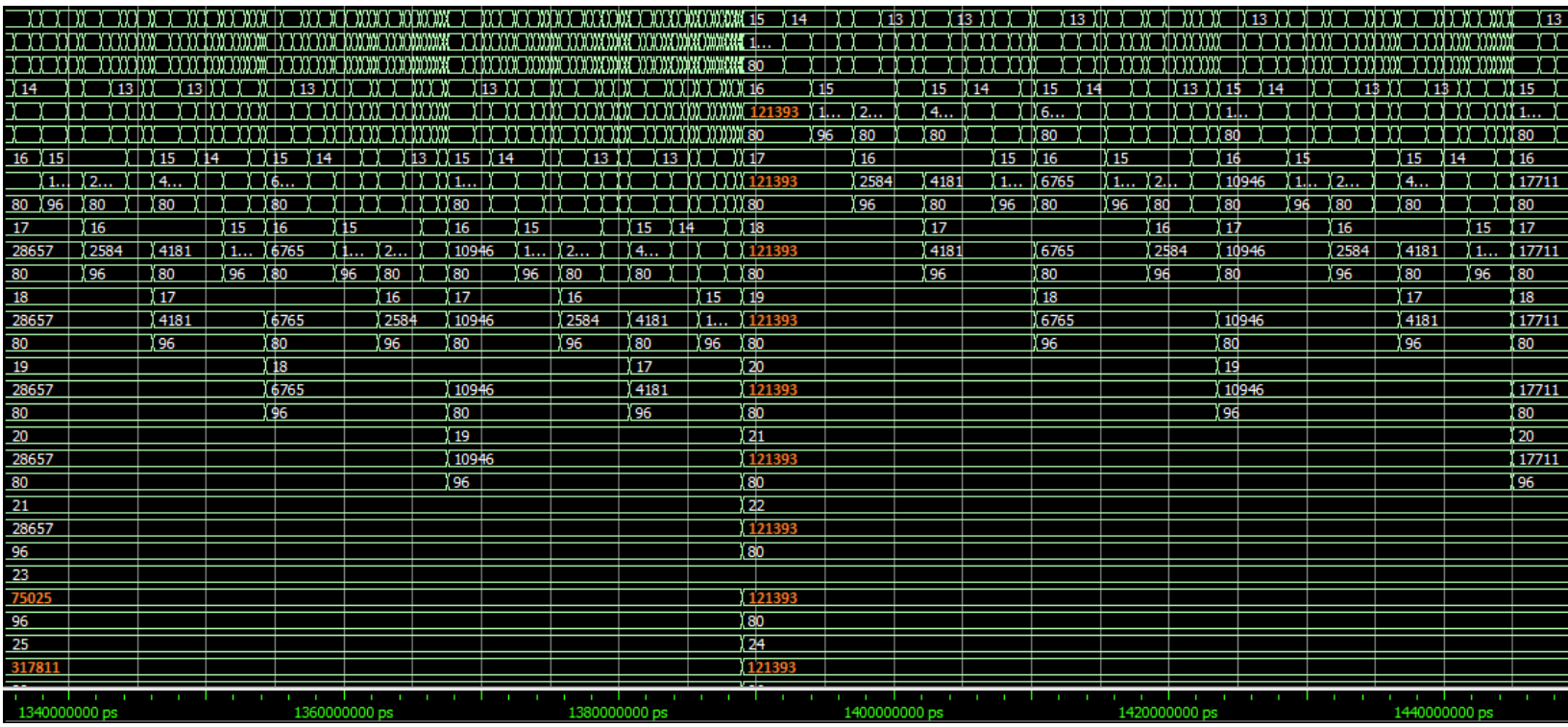


Figure84 : The Stack Frame for *Fibonacci*[31] ***…Zoomed in***

5.4 FPGA Synthesis (Timing, Area, Synthesis Reports) :

- The pipelined RISC-V processor was synthesized and implemented on an Altera DE2 FPGA board using **Quartus II 13.0sp1**. The design targets a lightweight 5-stage pipeline with basic hazard detection and forwarding mechanisms, supporting the RV32I instruction set.

♦ Resource Utilization :

The synthesis results indicate modest hardware overhead compared to a single-cycle processor, as shown in the table below:

Resource	Usage
Logic Elements	~3,200
Registers	~1,200
Embedded Memory	4 kbits (for RAM)
Clock Frequency	~285 MHz

Table6: Resource usage

- The increase in register and multiplexer usage is due to the addition of pipeline registers (IF/ID, ID/EX, EX/MEM, MEM/WB) and forwarding logic.

♦ Timing Summary :

Pipeline Stage	Approx. Delay
Fetch	1.2 ns
Decode	1.4 ns
Execute	2.8 ns
Memory	1.3 ns
Writeback	1.2 ns

Table7: Delays of each stage

- The maximum achievable clock frequency was approximately **285 MHz**, constrained primarily by the **Execute** stage. The critical path was identified as the **branch decision logic**, which includes data forwarding from the WB stage and ALU path evaluation.

♦ Comparison with Single-Cycle and Multicycle Designs :

- In earlier experiments with classic multicycle and optimized versions of the same RISC-V core:

- The **classic** design exhibited a long cycle time (~13 ns) due to the full datapath delay being on the critical path.
- The **optimized** design reduced the cycle time (~7 ns), but required multiple cycles per instruction.
- The **pipelined** implementation achieved a **significant speedup**, with a CPI of ~1.25 and reduced cycle time (~3.5 ns), leading to better instruction throughput.

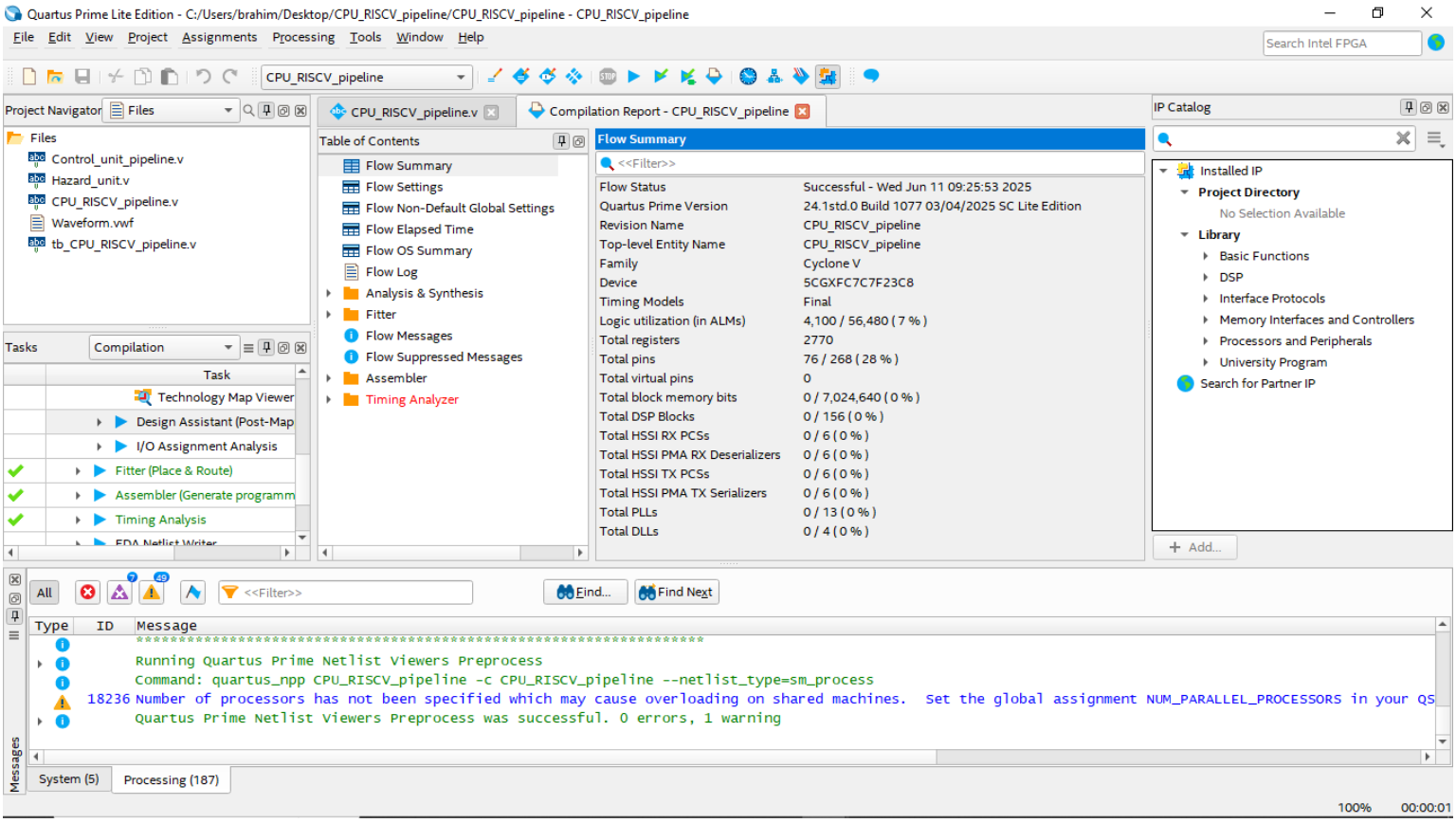


Figure85 : Compilation report for the Pipelined core

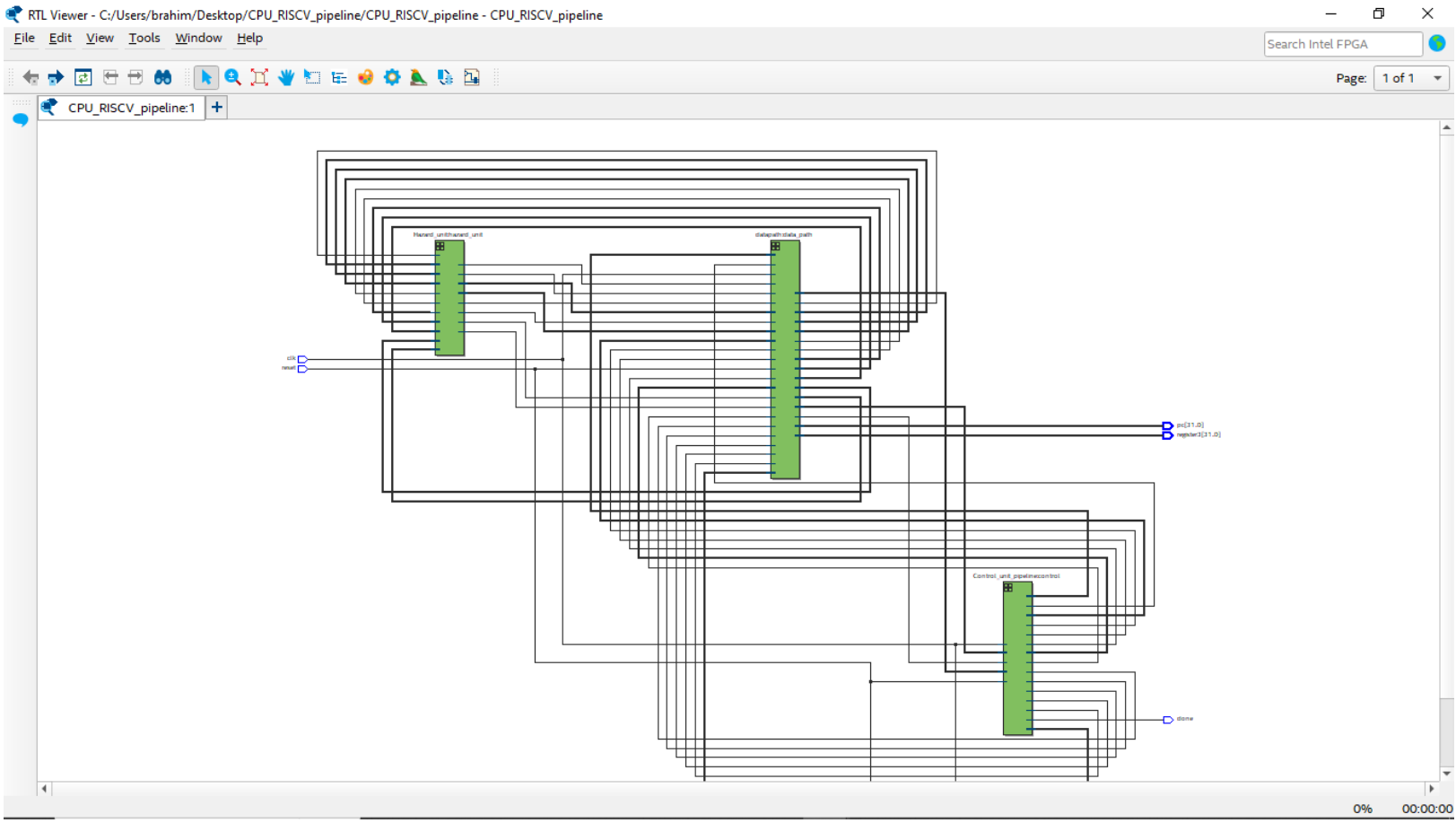


Figure86 : RTL view of the Pipelined core

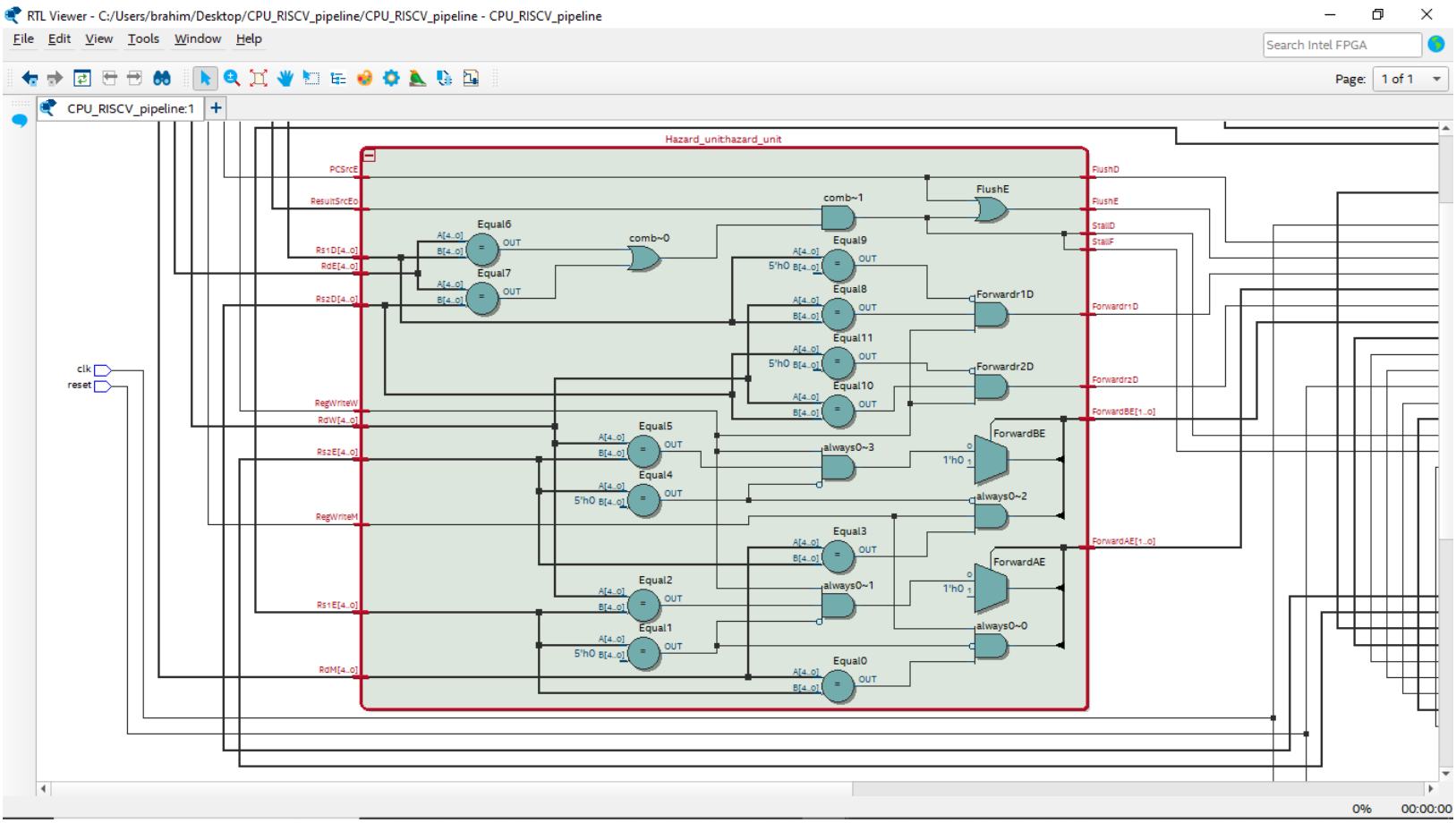


Figure87 : RTL view of the Hazard Unit

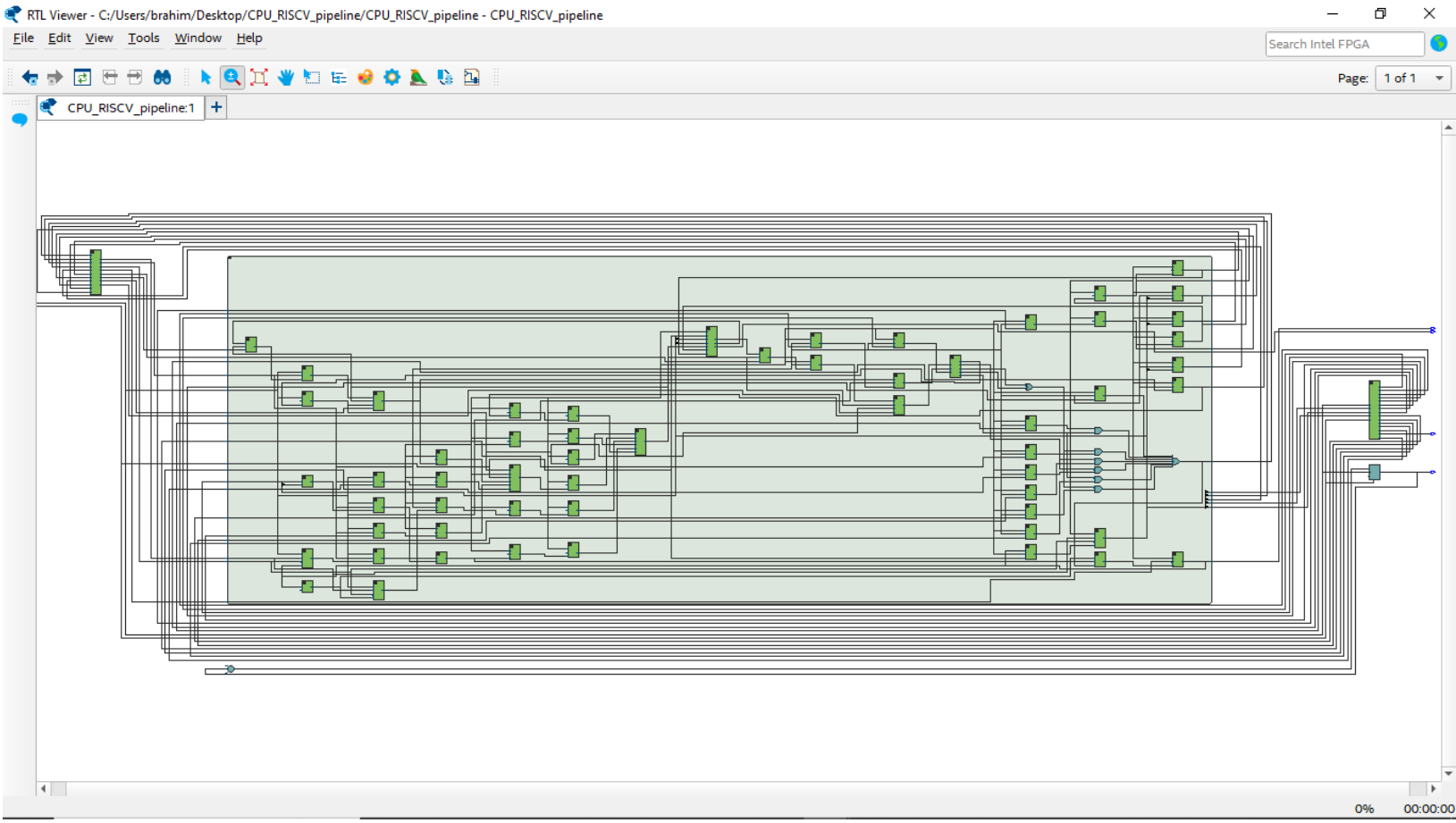


Figure88 : RTL view of the Datapath

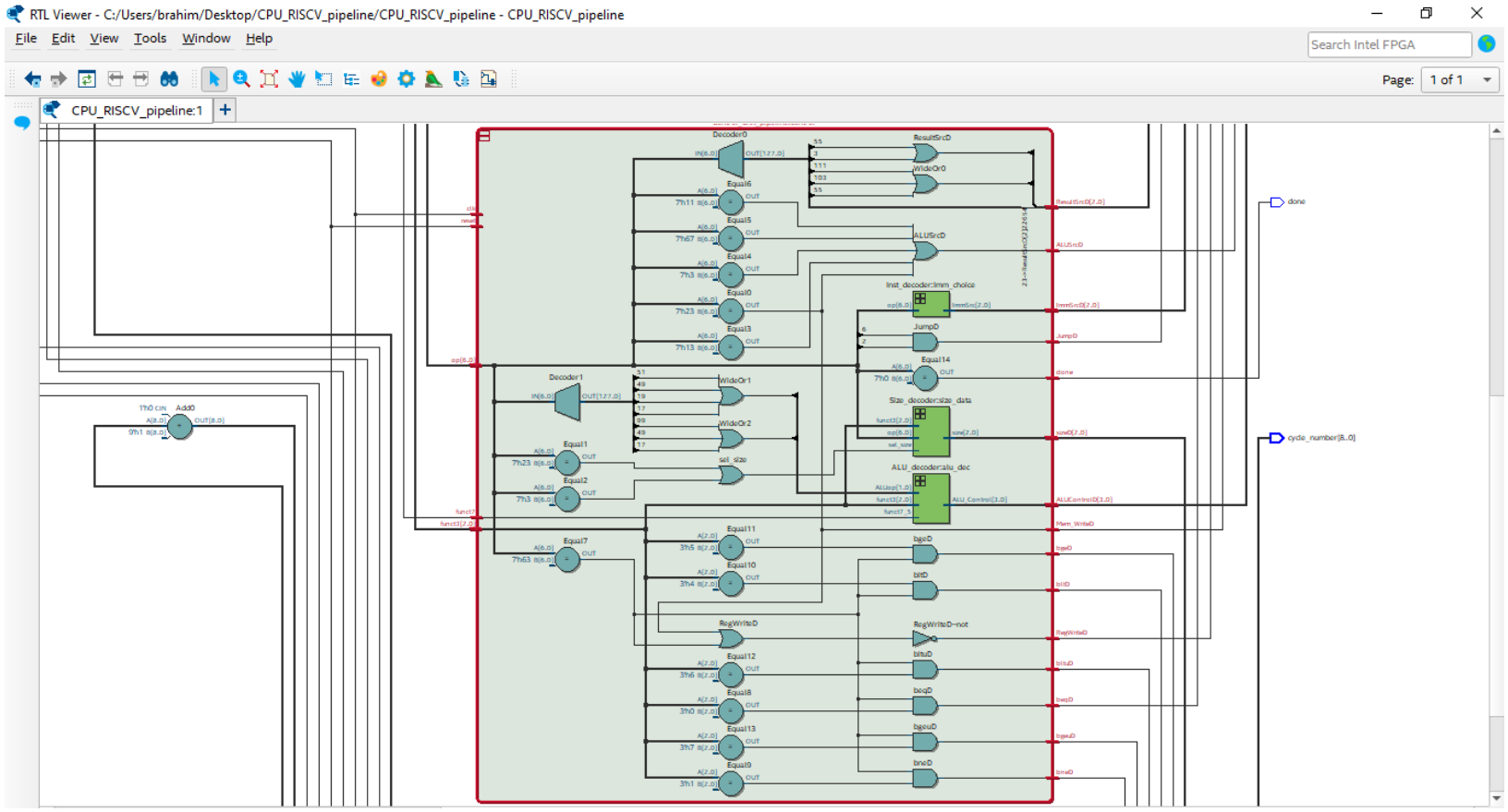


Figure90 : Technology map viewer of the Pipelined core

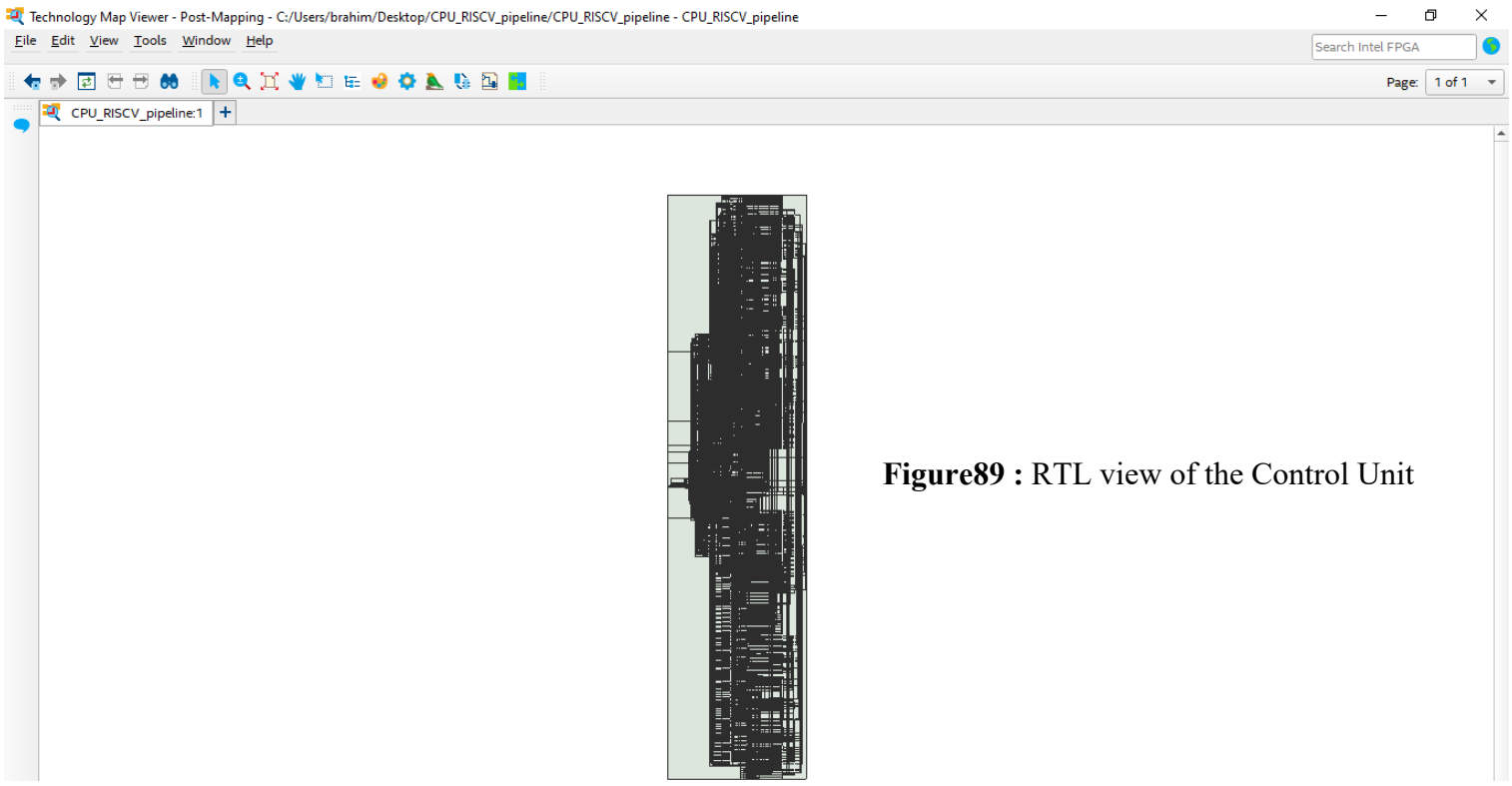


Figure89 : RTL view of the Control Unit

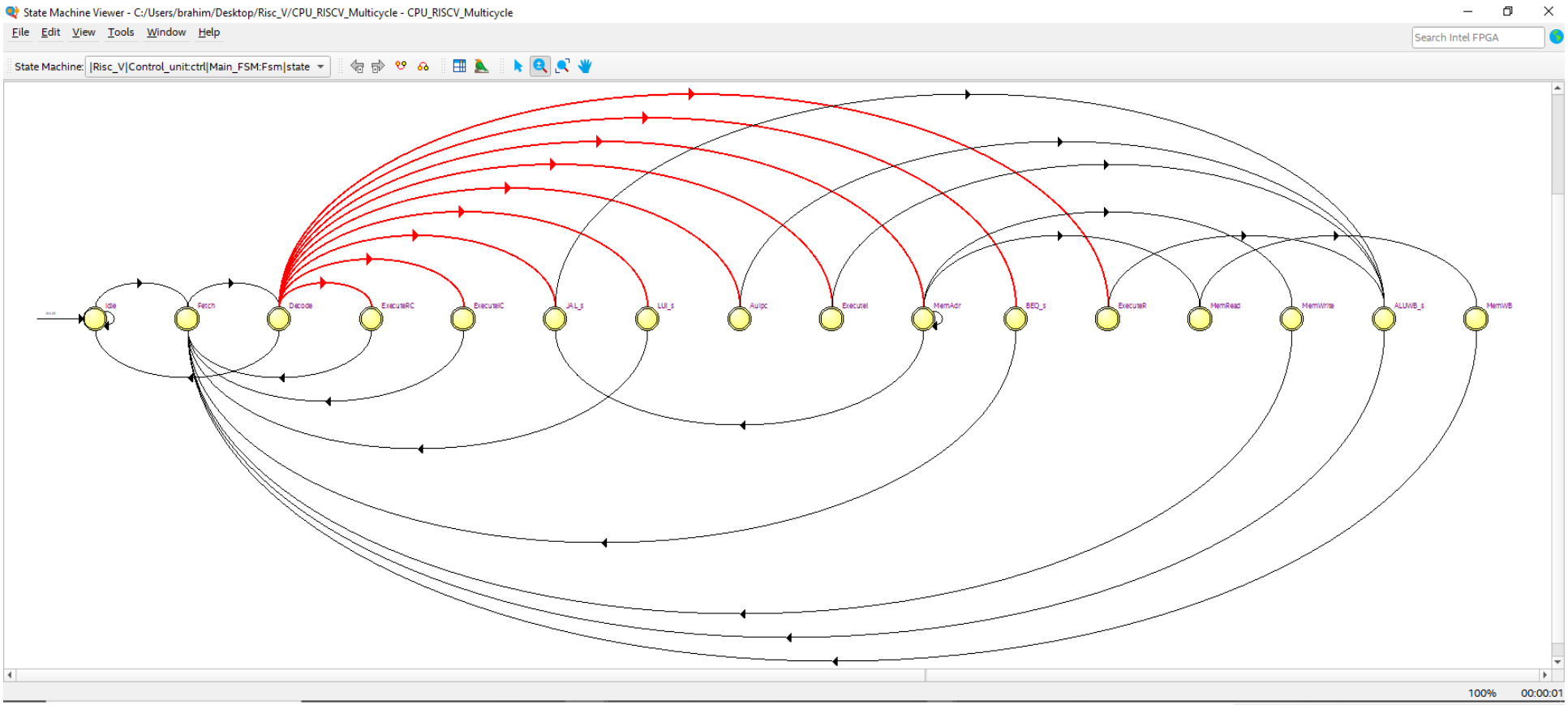


Figure91: State Machine Viewer of the Pipelined core

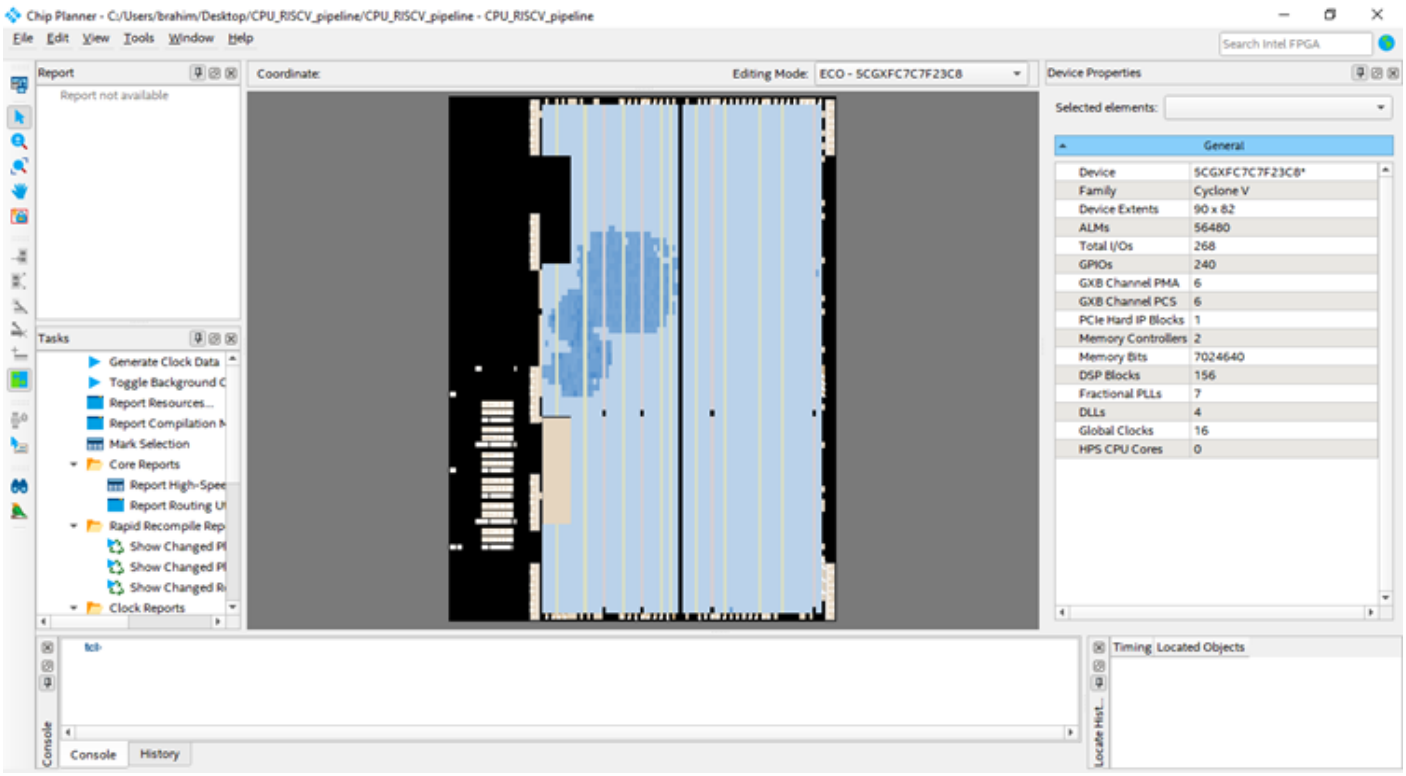


Figure92 : Chip Planner view of the Pipelined core (Cyclone V FPGA)

5.5 FPGA Implementation and Testing Results :

- Same approach as previous implementations , with the test case now being calculating again a Fibonacci sequence (the 21st term) but this time leveraging the performance increase of the pipeline

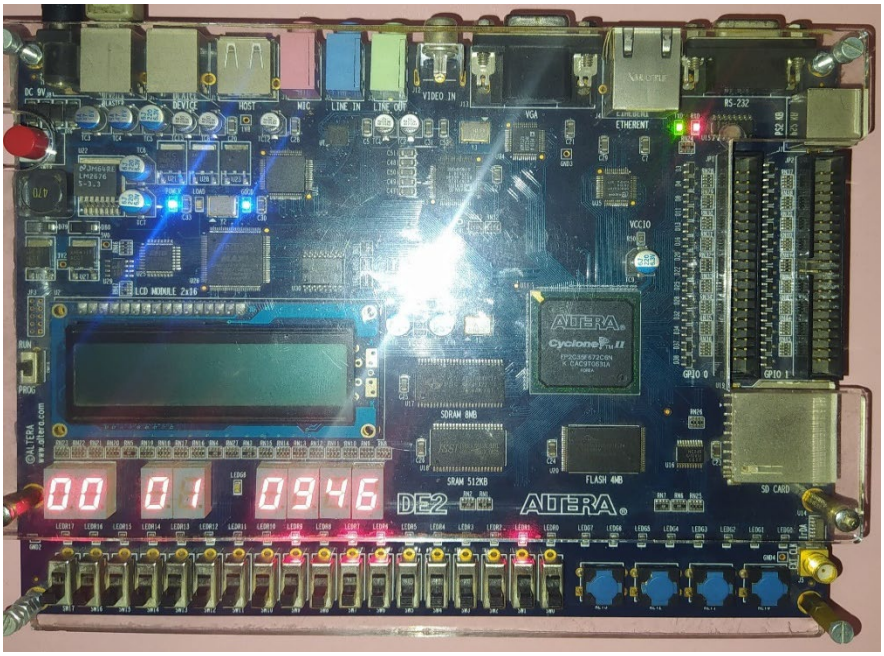


Figure93 : $Fibonacci[21] = 10946_{10} = 1011000010_2$ on the Pipelined core

6 • Comparative Analysis of Classical Multicycle, BraRV32 Optimized Multicycle, and Pipelined Implementations:

6.1 Introduction

In this chapter, we present a comparative study between three RISC-V processor implementations developed and tested as part of this project: the **Classical Multicycle**, the **BraRV32 Optimized Multicycle**, and the **Pipelined** version. The comparison is based on architectural structure, performance metrics such as CPI and execution time, hardware resource usage, and synthesis results on an FPGA target. This evaluation aims to highlight the trade-offs between simplicity, performance, and resource efficiency.

6.2 Architectural Overview :

6.2.1 Classical Multicycle Processor :

- The classical multicycle processor follows a textbook implementation model where each instruction is broken down into steps that are executed over multiple cycles. Each instruction requires between 3 and 6 clock cycles depending on its type. The control unit is implemented as a finite state machine (FSM), and resources such as the ALU and memory are reused across cycles, which reduces hardware cost but increases total instruction latency.

6.2.2 BraRV32 Optimized Multicycle :

- The BraRV32 is a custom multicycle implementation inspired by classical models but modified for improved performance. It maintains resource reuse across cycles but introduces the following optimizations:

- Shared memory for instructions and data (Harvard-von Neumann hybrid)
- Combined ALU+Branch logic to reduce state transitions
- Microcoded FSM with fewer control states for common instructions
- Improved memory-mapped I/O support for peripherals (7-segment display, LEDs)

- These changes reduce average instruction latency and better suit FPGA mapping.

6.2.3 Pipelined Processor :

- The pipelined implementation follows a classic 5-stage datapath (IF, ID, EX, MEM, WB). Pipeline registers are inserted between stages, and a hazard detection unit handles data and control hazards. Forwarding logic reduces unnecessary stalls, while control hazards are managed by flushing the pipeline. This version aims to maximize throughput by executing one instruction per cycle after filling the pipeline.

6.3 Performance Evaluation :

6.3.1 Clock Cycle Comparison :

Architecture	Average CPI	Typical Clock Period (ps)	Execution Time (100B inst)
Classical Multicycle	~4.8	4000 ps	~192 s
BraRV32 Optimized	~3.6	2800 ps	~100.8 s
Pipelined (5-stage)	~1.25	350 ps	~44 s

Table8 : CPI and Clock Period for each implementation

- The **Classical Multicycle** suffers from high CPI due to serialized execution.
- The **BraRV32** benefits from fewer cycles per instruction, reducing execution time by ~47% compared to the classical design.
- The **Pipelined processor** achieves the lowest execution time, benefitting from high instruction throughput.

6.3.2 Instruction-Level Efficiency :

Instruction Type	CPI (Classical)	CPI (BraRV32)	CPI (Pipelined)
ALU	4	3	1
Load	5	4	1.4
Store	4	2	1
Branch	5	3	2

Table9 : Instruction-Level Parallelism effect for each implementation

Instruction Type	CPI (Classical)	CPI (BraRV32)	CPI (Pipelined)
Jump	4	3	3

- These results show that **BraRV32 consistently improves instruction execution latency** over the classical design while remaining significantly simpler than the pipelined processor.

6.4 FPGA Resource Utilization :

- The designs were synthesized and tested on the **Altera DE2 board** using **Quartus II 13.0sp1**.

Resource	Classical Multicycle	BraRV32 Optimized	Pipelined
Logic Elements	~23,000	~7400	~3,200
Registers	~1300	~150	~2,200
Block RAM used	2 kbits	2 kbits	4 kbits
Max Freq (MHz)	~8	~65	~300

Table10 : Selected Resource utilization for each implementation

- **Classical Multicycle** uses more resources & achieves lower clock frequencies.
- **BraRV32** fewer resources and delivers better performance per LUT.
- **Pipelined design** requires additional pipeline registers and control logic, leading to higher area usage but better frequency vs CPI product.

6.5 Trade-Off Summary :

Feature / Criteria	Classical Multicycle	BraRV32 Optimized	Pipelined
Performance (CPI)	Low	Moderate	High
Hardware Simplicity	Very high	High	Moderate
Resource Efficiency	High	Balanced	Lower
I/O and Peripherals	Basic	Extended (MMIO)	Full MMIO
Control Complexity	Low (FSM)	Medium (uFSM)	High (Hazard logic)

Feature / Criteria	Classical Multicycle	BraRV32 Optimized	Pipelined
Best Use Case	Teaching / Simple CPU	Prototyping / Labs	High-throughput apps

Table11 : Summary between the implementations

6.6 Conclusion :

- This chapter provided a comparative overview of three RISC-V processor implementations. The **Classical Multicycle** design is most suitable for educational purposes and hardware-constrained environments, whereas the **BraRV32** optimized multicycle strikes a balance between performance and complexity, making it ideal for hands-on learning and real-world interfacing projects hence it's inclusion in the scope of our project. The **Pipelined processor**, while more complex, provides superior throughput and is better aligned with real CPU design practices.

Each architecture has merits depending on the design goals—**simplicity, performance, or resource usage**. Understanding these trade-offs is crucial for making informed architectural choices in processor design.

7 • Development of a Custom RISC-V Assembler :

7.1 Introduction :

To support the **BraRV32** and it's successors with their simulation environment, a lightweight assembler was developed in **Python**. This tool serves as a bridge between human-readable **RISC-V** assembly code and the binary machine code expected by the processor core. The assembler was designed to be extensible, modular, and adapted to the different architectures it doesn't implement nothing fancy like linking and a full on generated `.elf` file, but rather a simple Assembly →to→ Machine Code direct translation via the RISC-V conventions , also the different immediate scrambling PC and Jump offsets store/load addresses are also assessed and calculated by the status quo.

7.2 Motivation and Objectives

- While several RISC-V compilers already exist, such as `riscv-gcc` and `spike`, these often generate binaries in formats and layouts that are unsuitable for lightweight educational or FPGA-based softcore processors. The need to assemble and precisely locate instructions in word-addressable memory, while supporting

user-defined labels and basic I-type, R-type, S-type, B-type, J-type formats, led to the implementation of a custom compiler.

- Main goals:

- Assemble raw RISC-V assembly instructions into 32-bit binary strings.
- Support labeling for jump and branch instructions.
- Handle multiple input programs placed at specific memory addresses.
- Remain simple and easily modifiable for educational purposes.

7.3 System Overview :

- The assembler is divided into **two major phases**:

1. **Label Collection**: Reads user-defined label-address mappings from a simple interface or from a file.
2. **Assembly**: Parses and encodes assembly code into binary, supporting multiple files and writing per-label outputs.

7.4 Label Input Mechanism :

- A short script captures label names and their memory start addresses. These labels are stored in a dictionary and saved to a file `labels.txt` in the format:

```
main 0
fib 40
```

- This file allows the assembler to resolve symbolic branch/jump destinations during instruction encoding.

```
labels[name] = addr
```

7.5 Instruction Parsing and Encoding :

- The compiler supports a subset of the RISC-V base ISA (`RV32I`) sufficient for educational experimentation:

- Arithmetic instructions: `add`, `addi`
- Memory instructions: `lw`, `sw`
- Branches and jumps: `beq`, `jal`, `jalr`

- Each instruction is first parsed into components (mnemonic and operands), and its type is determined (`R`, `I`, `S`, `B`, `J`). Based on the instruction type, fields are converted into binary using Python's string formatting and bit manipulation.

- A sample of encoding logic for R-type (`add`) is:

```
def encode(inst_type, mnemonic, operands, pc, label_map):
    if inst_type == 'R':
        rd, rs1, rs2 = operands
        return funct7[mnemonic] + reg_map[rs2] + reg_map[rs1] +
               funct3[mnemonic] + reg_map[rd] + opcodes[mnemonic]
```

- Jump (`jal`) and branch (`beq`) instructions compute the offset to the target label using the program counter (`pc`) and the label's resolved address. Word or byte addressability is taken into account depending on the memory architecture.

7.6 Assembly Flow and Output :

- The assembler processes all `.txt` files corresponding to labels previously declared. Each file is assembled individually, allowing modular program development. Output binaries are written to a folder path, with filenames matching the label name (e.g., `main.txt`, `fib.txt`).

```
for asm_text, label_addr in files_info:
        bin_lines = assemble_from_text(asm_text, label_addr, labels)

        # Derive output file name from corresponding label (e.g., main -
> main.tv)
        for label, addr in labels.items():
            if addr == label_addr:
                output_filename = f"{label}.txt"
                break
            else:
                output_filename = f"program_{label_addr}.txt"  #
Fallback
        output_path = f"{folder}/{output_filename}"
        with open(output_path, 'w') as f:
            for line in bin_lines:
                f.write(line + '\n')

        print(f"Assembly complete for label '{label}'. Output written to
'{output_filename}'.")
    #with open('C:/Users/brahim/Desktop/BraRV32/rtl/outt.tv', 'w') as f:
        #for line in all_binary_lines:

            #f.write(line + '\n')

    print("Assembly complete. Output written to 'output.tv'.")
```

- The output is a text file containing one binary instruction per line, which can be directly loaded into simulation memory or flashed to the FPGA ROM.

7.7 Example: Assembling a Fibonacci Program (Without recurssion) :

- Given a label `fib` at address 40 and a file `fib.txt` containing:

```
fib:
    addi,x1, x0, 0
    addi,x2, x0, 1
loop:
    add,x3, x1, x2
    beq,x3, x4, end
    addi,x1, x2, 0
    addi,x2, x3, 0
    jal,x0, loop
end:
    jalr,x0, x0, 0
```

- The compiler will resolve the label addresses (`loop`, `end`), calculate the PC-relative offsets, and emit the binary representation for each instruction in `fib.txt`.

7.8 Error Handling and Debugging :

- Throughout the compilation process, the tool provides descriptive error messages for invalid instruction formats or malformed operands. This is particularly useful during testing and when modifying instruction encodings or memory layouts.

```
if not inst_type:
            continue
        try:
            #print(inst_type,mnemonic,operands,pc,label_map)
            print(label_map)
            bin_code = encode(inst_type, mnemonic, operands, pc,
                                                    label_map)
            print("####################################")
            print(bin_code)
            binary_lines.append(bin_code)
            #print(binary_lines)
            #for i in range (2):
            print('\n')
        except Exception as e:
            #print(inst_type, mnemonic, operands, pc, label_map)
            #print(f'{inst_type}, {mnemonic}, {operands}, {pc},
{label_map}')
            #binary_lines.append(f'ERROR: {e}') #wasn't sufficient
enough to determine the source of the error
            print(f'ERROR at PC={pc} ({mnemonic} {operands}): {e}')
            binary_lines.append(f'ERROR: {mnemonic} {operands} @ PC
                                                    {pc} => {e}')
        pc += 4
```

- These checks help ensure the correctness of assembled programs and prevent silent failures.

7.9 Extensibility and Future Improvements :

The compiler is built with extensibility in mind. Adding new instructions involves updating the `opcodes`, `funct3`, `funct7`, and writing the corresponding encode logic. Future improvements may include:

- Supporting `.data` sections and memory initialization.
- Handling pseudo-instructions (`li`, `mv`, `nop`).
- Generating ELF or memory hex formats.
- GUI or web-based frontend for educational usage.

7.10 Conclusion :

- The custom assembler implemented in Python offers a simple yet powerful tool for translating RISC-V assembly into binary form tailored for simulation and FPGA deployment of the BraRV32 & other cores. It enables structured, modular software development and provides a hands-on interface to understand instruction encoding and program memory layout.

- This tool forms a critical link between software development and hardware simulation in ourBraRV32 project.

• Wrapper Development To Abstract The Compilation Workflow •

To enhance usability and streamline the compilation process, a high-level **Python wrapper script was developed**. This wrapper **abstracts the complexity of label management**, instruction parsing, and binary generation, thereby enabling the user to interact with the compiler suite through a minimal and intuitive interface.

- The primary objective of this wrapper is to facilitate batch compilation of multiple RISC-V assembly programs located in text files, each associated with a predefined label and memory address. Rather than invoking individual scripts manually, the user is only required to specify two input paths:

1. **The path to the folder where the `.tv` output files should be generated** – these contain the compiled 32-bit machine code in textual format.
2. **The path to the `labels.txt` file** – this file holds the symbolic labels and their corresponding memory addresses.

⚠ the `.txt` assembly codes with the same names as the labels given in the `labels.txt` should all be in the same initial folder

- Upon execution, the wrapper prompts the user to either enter the labels and their addresses in real-time or reads the label-to-address mappings from `labels.txt`, which it's path was provided by user input, then automatically locates and processes the associated `.txt` files containing the assembly code (e.g., `main.txt`,

`fib.txt`, etc.). Each file is parsed, assembled using the underlying encoding functions, and finally, the resulting machine code is written to a corresponding `.tv` output file in the specified directory.

- This modular structure promotes code reuse, separation of concerns, and ease of maintenance. It also aligns with software engineering best practices by offering a user-friendly entry point while retaining the flexibility and transparency of the lower-level components which can be easily extended to more instruction types new encodings by just the implementation of newer code blocks which renders the code modular & extensible for any future use.

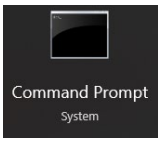
- Through this design, the wrapper significantly reduces the user's cognitive load and risk of error, making the compilation toolchain more accessible for both educational and practical use in RISC-V system design.

```
import os
from RISC_V_Assembler import assemble_from_text, load_labels
from pathlib import Path

def read_multiple_files_from_folder(label_map, folder_path):
    """
    Reads all .txt files corresponding to labels and associates them with their respective memory
addresses.
    """
    files_info = []
    for label, addr in label_map.items():
        filename = os.path.join(folder_path, f"{label}.txt")
        if not os.path.exists(filename):
            print(f"[WARNING] File for label '{label}' not found: {filename}")
            continue
        try:
            with open(filename, "r") as f:
                asm_text = f.read()
            files_info.append((label, asm_text, addr))
        except Exception as e:
            print(f"[ERROR] Could not read '{filename}': {e}")
    return files_info


def main():
    print("=== RISC-V Assembly Compilation Wrapper ===")



    asm_folder = input("Enter the folder containing assembly .txt files: ").strip()
    while not os.path.isdir(asm_folder):
        print("Invalid directory. Please try again.")
        asm_folder = input("Enter the folder containing assembly .txt files: ").strip()

    output_folder = input("Enter the folder where .tv output files will be saved: ").strip()
    Path(output_folder).mkdir(parents=True, exist_ok=True)  # create folder if it doesn't exist

    #2# Load label-to-address map
```

```
print("Choose label input method:")
print("1 - Enter labels manually")
print("2 - Provide path to labels.txt file")

choice = input("Your choice [1 or 2]: ").strip()
while choice not in {"1", "2"}:
    choice = input("Invalid choice. Please enter 1 or 2: ").strip()

if choice == "1":
    labels = {}

    print("Enter label names and addresses. Press ENTER with no input to stop.")

    while True:
        name = input("Label name: ").strip()
        if name == "":
            break
        addr = input(f"Address for label '{name}': ").strip()
        if addr == "":
            print("Address cannot be empty. Try again.")
            continue
        labels[name] = addr
    label_map = labels
        # Write to file
    with open("labels.txt", "w") as f: # should be labels
        for label, addr in labels.items():
            f.write(f"{label} {addr}\n")
else:
    labels_path = input("Enter the path to 'labels.txt': ").strip()
    while not os.path.isfile(labels_path):
        print("Invalid path. Please try again.")
        labels_path = input("Enter the path to 'labels.txt': ").strip()
    label_map = load_labels(labels_path)

print(f"[INFO] Loaded labels: {label_map}")


label_map = load_labels(labels_path)
print(f"[INFO] Loaded labels: {label_map}")

#3# Load assembly files
files_info = read_multiple_files_from_folder(label_map, asm_folder)

#4# Assemble and write outputs
for label, asm_text, addr in files_info:
    print(f"\n[INFO] Assembling '{label}.txt' at address {addr}")
    bin_lines = assemble_from_text(asm_text, int(addr), label_map)

    output_file = os.path.join(output_folder, f"{label}.tv")
    with open(output_file, "w") as f:
        for line in bin_lines:
            f.write(line + "\n")
    print(f"[SUCCESS] Output written to '{output_file}'")

print("\n[✔] All files compiled successfully.")
```

```
if __name__ == "__main__":
    main()
```

• Demo •



- First we open a command line interface, for instance Windows Command Prompt

- Navigate to where your Assembler files exist for my test case it's in `C:\Users\brahim\Desktop\Assembler` →

Figure94 : The Command Prompt

- Change directory via the command `cd` **:**

```
Cd  C:\Users\brahim\Desktop\Assembler
```

- The Directory should look something like

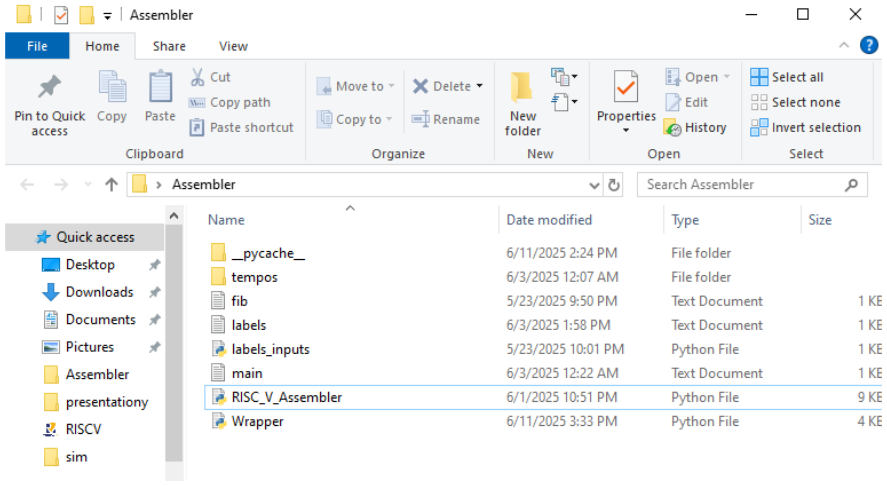


Figure95 : Assembler Working Directory

- The sample programs here are :

fib - Notepad

File Edit Format View Help

```
fib:
addi,x2,x2,-12
sw,x1,8(x2)
sw,x8,4(x2)
fib_if_x0_or_one:
addi,x10,x0,0
beq,x12,x0,fib_fin
addi,x10,x10,1
beq,x12,x10,fib_fin
fib_call_n_1:
addi,x12,x12,-1
sw,x12,0(x2)
jal,x1,fib
lw,x12,0(x2)
addi,x12,x12,-1
fib_call_n_2:
add,x8,x10,x0
jal,x1,fib
add,x10,x10,x8
fib_fin:
lw,x8,4(x2)
lw,x1,8(x2)
addi,x2,x2,12
jalr,x0,x1,0
```

Ln 1, Col 1 100% Windows (CRLF) UTF-8

main - Notepad

File Edit Format View Help

```
main:
addi,x12,x0, 31
jal,x1,fib
jalr,x0,x1,0
```

Ln 1, Col 1 100% Windows (CRLF) UTF-8

Figure96: main.txt and fib.txt RISC-V Assembly codes

- The Assembling procedure can be initiated with the command :

```
Python wrapper.py
```

C:\WINDOWS\system32\cmd.exe

```
Microsoft Windows [Version 10.0.19045.5854]
(c) Microsoft Corporation. All rights reserved.

C:\Users\brahim>cd Desktop\Assembler

C:\Users\brahim\Desktop\Assembler>python wrapper.py
```

Figure97: Calling the wrapper and therefore starting the assembling procedure

- We're greeted with this text prompting us to start entering the necessary information needed to assemble a given file or files :

C:\WINDOWS\system32\cmd.exe - python wrapper.py

```
Microsoft Windows [Version 10.0.19045.5854]
(c) Microsoft Corporation. All rights reserved.

C:\Users\brahim>cd Desktop\Assembler

C:\Users\brahim\Desktop\Assembler>python wrapper.py
=== RISC-V Assembly Compilation Wrapper ===
Enter the folder containing assembly .txt files:
```

Figure98: User input for the folder (dir) containing the assembly programs

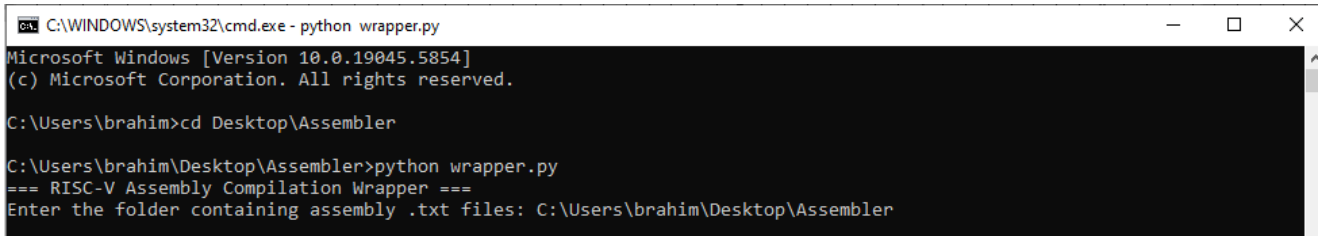


Figure99: putting the same directory path

- For my case they lie in the same folder

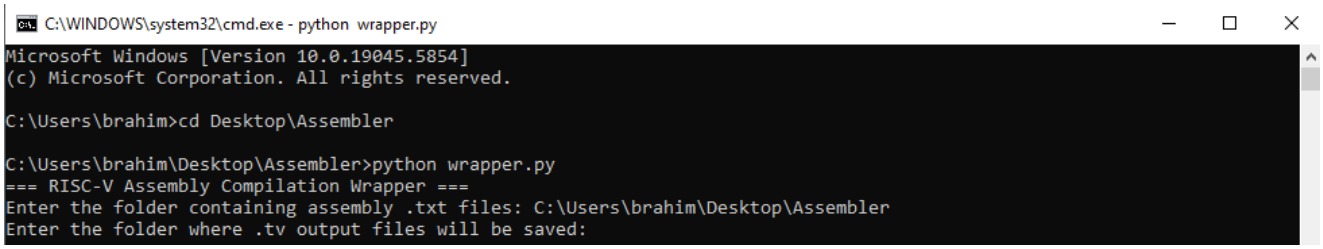


Figure100: User input for the folder (dir) where the binary file (`.tv`) will be outputted to

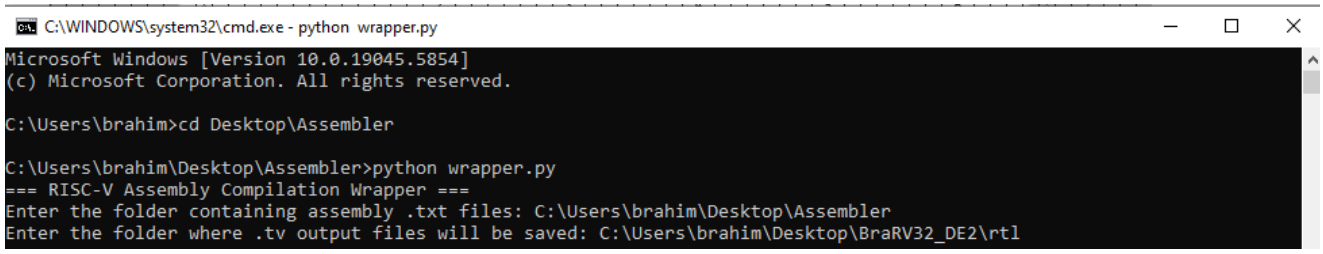


Figure101: For My practical case it's the Resgister Transfer Level folder where all my HDL files and top module exist

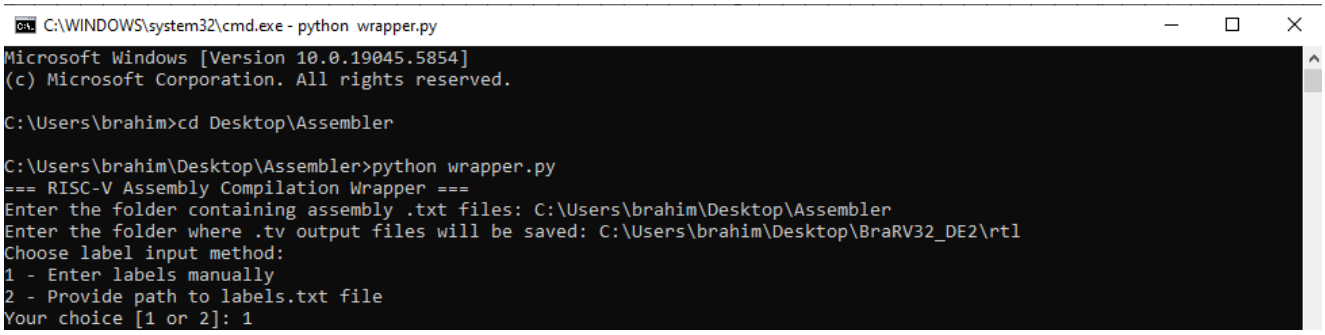


Figure102: Label input method

- The user's choice on whether to enter or import the labels and their addresses

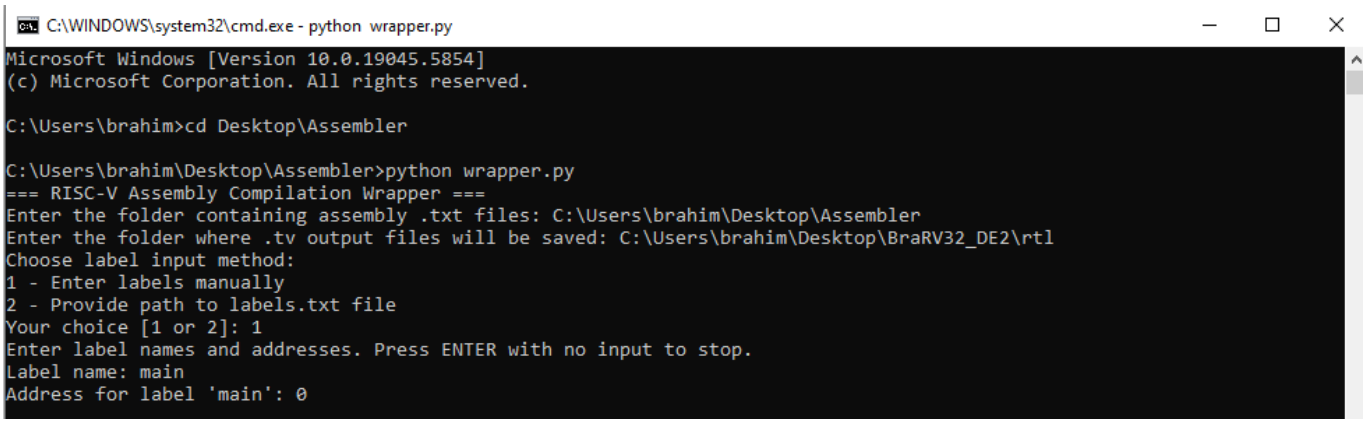


Figure103: Label name and address

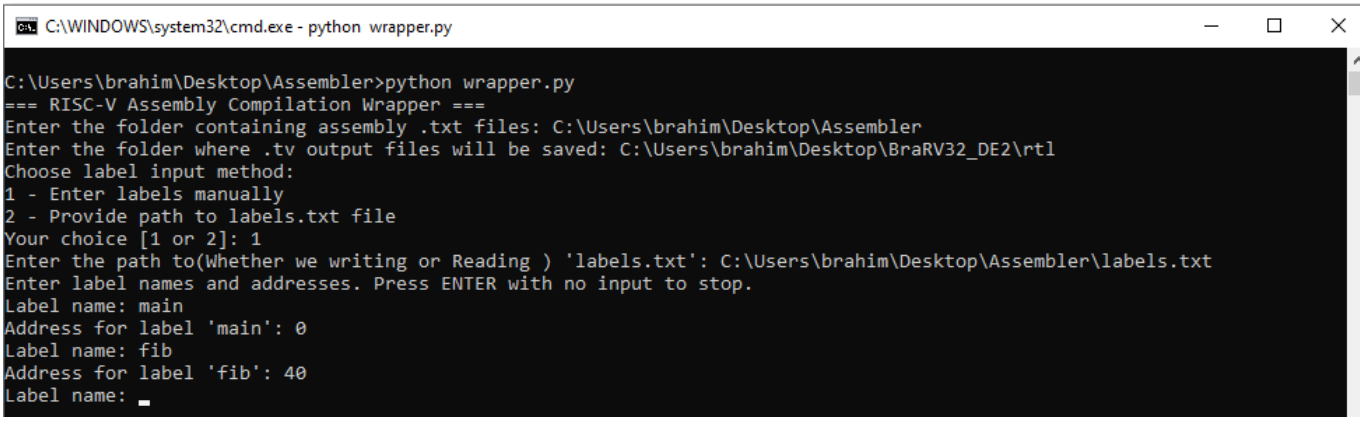


Figure104: Pressing ←ENTER After all labels are registered

- After this the code should be assembeled in the desired folder waiting to get used by a memory model via verilog's read memory functions (e.g `$readmemb`)

**Note:*

this is particularly useful since we can call and invoke the files with the machine code (we can merge them into a singular file when it's necessary at FPGA synthesis before deployement on the board especially since a lot of file manipulation functions in Verilog are only supported in simulation and are not synthesis/implementation friendly)

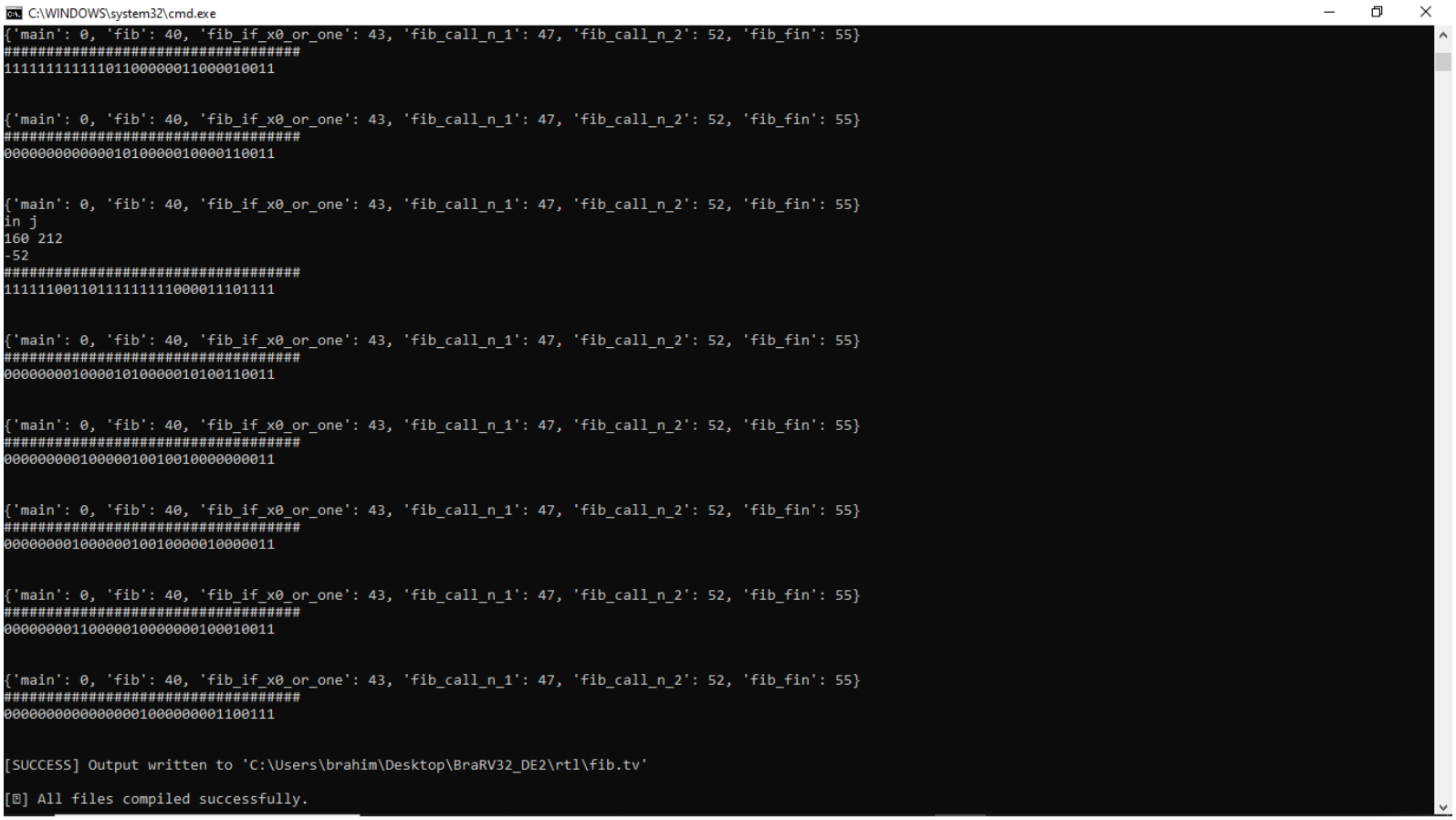


Figure105: Final Output

- In Synthesis we approach the situation little bit differently due to the lack of file manipulation functions , the approach is simple we proceed as earlier assembling the code with it's different labels, addresses via the wrapper and then invoke the `mif_merger.py` to generate either a `.mif` or `.tv` depending whether we're infering a M10K_Ram block or using just the FPGA's LUT as basic memory.

- Either way after assembling and generating the `.tv` in the desired folder we can then invoke `mif_merger.py`

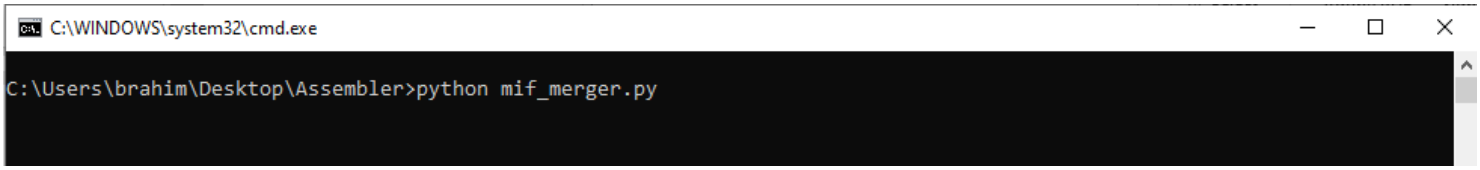


Figure106: Running `mif_merger.py`

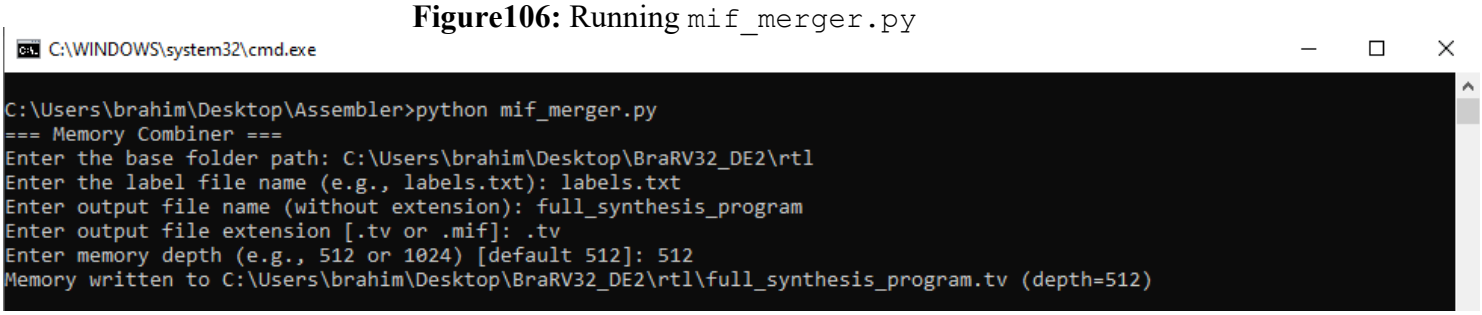
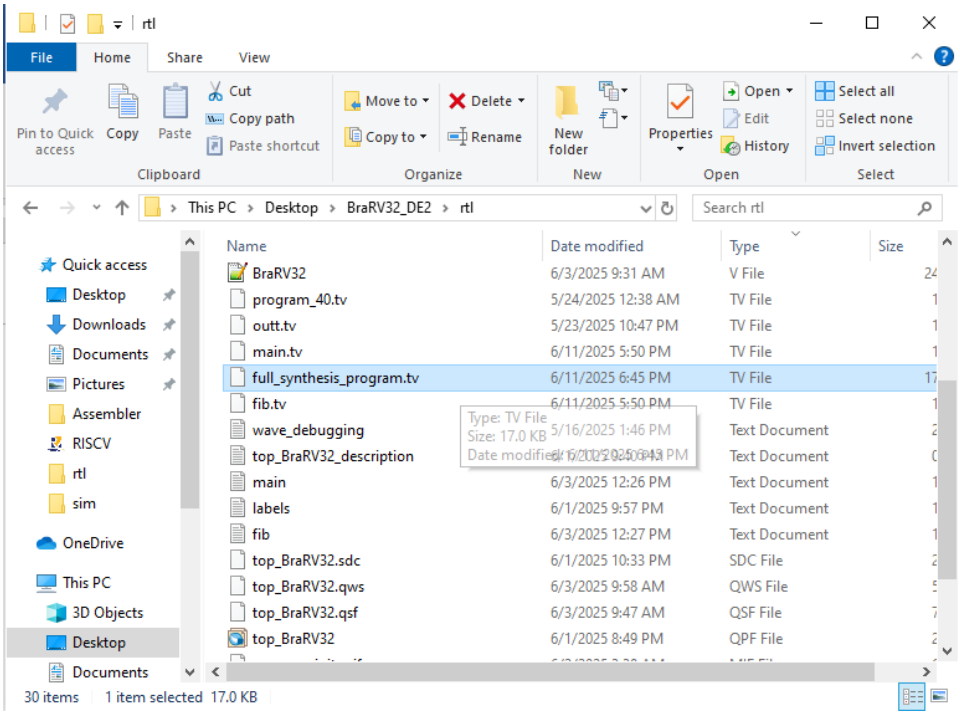


Figure107: Synthesis memory_merger final inputs and generated file output




```
00000001111100000000011000010011
00001001110000000000000011101111
00000000000000001000000001100111
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
11111111010000010000000100010011
00000000000010001001001000100011
00000001000000100100010001000011
00000000000000000000010100010011
00000010000001100000011001100011
00000000000010101000001010010011
00000010101001100000001001100011
11111111111110110000011000010011
00000000110000010010000000100011
11111101110111111111000011101111
00000000000000001001001100000011
11111111111110110000011000010011
00000000000001010000010000110011
```

```
11111100110111111111000011101111
00000000100001010000010100110011
00000000100000100100100000000011
00000000100000010010000010000011
00000000110000010000000100010011
00000000000000001000000001100111
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
… … … … … … … … … … … … … …
00000000000000000000000000000000
```

Figure108: generated binary output file and it's contents

8 • Conclusion and Future Work :

8.1 Project Summary

- This project aimed to explore the practical and theoretical aspects of processor architecture design, implementation, and toolchain development through a progressive approach involving multiple processor models. We began with a **classic multicycle processor** implementation, which provided foundational insights into instruction cycle breakdown, control path design, and memory interfacing. We then transitioned to the design and optimization of **BraRV32**, a custom RISC-V compliant processor core adapted for FPGA deployment. This design emphasized clarity, synthesis efficiency, and expandability.

- To further enhance instruction throughput, we integrated **pipelining techniques** and explored basic forwarding and stalling mechanisms. In parallel, we developed a **custom assembler** supporting label resolution and PC-relative offsets, along with utilities for memory generation in `.tv` and `.mif` formats. These components together allowed the construction of a streamlined and FPGA-ready RISC-V development flow.

8.2 Key Takeaways

- Throughout this project, we acquired a broad yet deep understanding of the following:

- **Microarchitectural Trade-offs:** Comparing the multicycle and pipelined BraRV32 architectures allowed us to appreciate the complexity-performance trade-offs and the cost of hazard management.

- **RISC-V Ecosystem Familiarity:** Working within the RISC-V ISA framework offered invaluable exposure to an open and extensible instruction set. This experience reinforced modular and standards-compliant design practices.
- **Toolchain Integration:** The development of a custom assembler and memory tools deepened our appreciation for the importance of a robust software–hardware interface in SoC development.
- **FPGA Design Flow:** Deploying the BraRV32 on the Altera DE2 board using Quartus II enabled practical debugging and performance verification, solidifying our understanding of synthesis constraints, timing analysis, and hardware debugging using tools such as SignalTap.
- **Scalable Design Thinking:** Designing with abstraction in mind, such as through wrapper scripts, label handling automation, and clean I/O interfaces, taught us how to build reusable, maintainable hardware/software systems.

8.3 Potential Extensions

- Building upon the solid architectural and tooling foundation established in this project, several promising avenues for future development have been identified:

- **Full SoC Integration:** Integrating BraRV32 within a complete System-on-Chip design would require developing or adopting modules for peripherals, timers, GPIO, interrupt controllers, and other I/O devices. This will enable more realistic application deployment scenarios.
- **AXI/AMBA Bus Support:** Transitioning from custom memory mapping to industry-standard bus protocols such as **AXI** or **AHB/AMBA** will facilitate interoperability with a wider range of IP cores and provide scalability towards larger systems.
- **Cache and Memory Controller Design:** Implementing a basic **L1 cache** and designing a memory controller would allow the processor to better support high-latency memory systems and multitasking environments. This step is crucial for moving towards performance-optimized designs.
- **Open-source Contributions:** With proper documentation, verification, and packaging, the BraRV32 project, along with its assembler and toolchain, can be made open-source. This would not only contribute to the growing RISC-V educational and research ecosystem but also attract community feedback and collaboration.

- This project has served as a comprehensive introduction to processor design, system integration, and tool development. It has laid a robust groundwork for advanced architectural exploration, including speculative execution, branch prediction, and out-of-order execution in future work. Most importantly, it has reinforced the value of clean design, modular thinking, and academic rigor in digital system design.

Appendix A : Instruction Set Coverage

Table B.4 Register names and numbers

Name	Register Number	Use
zero	x0	Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0–2	x5–7	Temporary registers
s0/fp	x8	Saved register / Frame pointer
s1	x9	Saved register
a0–1	x10–11	Function arguments / Return values
a2–7	x12–17	Function arguments
s2–11	x18–27	Saved registers
t3-6	x28–31	Temporary registers

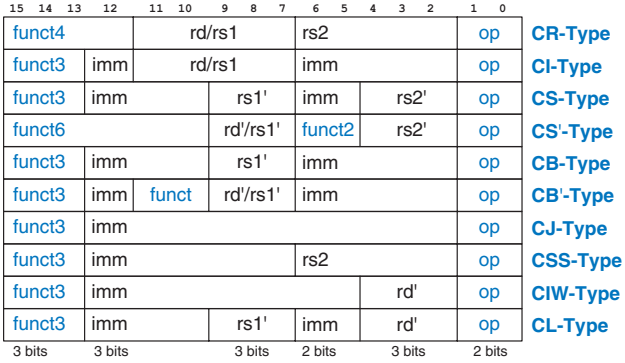


Figure B.2 RISC-V compressed (16-bit) instruction formats

Table B.5 RVM: RISC-V multiply and divide instructions

op	funct3	funct7	Type	Instruction	Description	Operation
0110011 (51)	000	0000001	R	`mul rd, rs1, rs2`	multiply	$rd = (rs1 * rs2)_{31:0}$
0110011 (51)	001	0000001	R	`mulh rd, rs1, rs2`	multiply high signed signed	$rd = (rs1 * rs2)_{63:32}$
0110011 (51)	010	0000001	R	`mulhsu rd, rs1, rs2`	multiply high signed unsigned	$rd = (rs1 * rs2)_{63:32}$
0110011 (51)	011	0000001	R	`mulhu rd, rs1, rs2`	multiply high unsigned unsigned	$rd = (rs1 * rs2)_{63:32}$
0110011 (51)	100	0000001	R	`div rd, rs1, rs2`	divide (signed)	`rd = rs1 / rs2`
0110011 (51)	101	0000001	R	`divu rd, rs1, rs2`	divide unsigned	`rd = rs1 / rs2`
0110011 (51)	110	0000001	R	`rem rd, rs1, rs2`	remainder (signed)	`rd = rs1 % rs2`
0110011 (51)	111	0000001	R	`remu rd, rs1, rs2`	remainder unsigned	`rd = rs1 % rs2`

Table B.6 RVC: RISC-V compressed (16-bit) instructions

op	$instr_{15:10}$	funct2	Type	RVC Instruction	32-Bit Equivalent
00 (0)	000–––	–	CIW	`c.addi4spn rd', sp, imm`	`addi rd', sp, ZeroExt(imm)*4`
00 (0)	001–––	–	CL	`c.fld fd', imm(rs1')`	`fld fd', (ZeroExt(imm)*8)(rs1')`
00 (0)	010–––	–	CL	`c.lw rd', imm(rs1')`	`lw rd', (ZeroExt(imm)*4)(rs1')`
00 (0)	011–––	–	CL	`c.flw fd', imm(rs1')`	`flw fd', (ZeroExt(imm)*4)(rs1')`
00 (0)	101–––	–	CS	`c.fsd fs2', imm(rs1')`	`fsd fs2', (ZeroExt(imm)*8)(rs1')`
00 (0)	110–––	–	CS	`c.sw rs2', imm(rs1')`	`sw rs2', (ZeroExt(imm)*4)(rs1')`
00 (0)	111–––	–	CS	`c.fsw fs2', imm(rs1')`	`fsw fs2', (ZeroExt(imm)*4)(rs1')`
01 (1)	000000	–	CI	`c.nop (rs1=0,imm=0)`	`nop`
01 (1)	000–––	–	CI	`c.addi rd, imm`	`addi rd, rd, SignExt(imm)`
01 (1)	001–––	–	CJ	`c.jal label`	`jal ra, label`
01 (1)	010–––	–	CI	`c.li rd, imm`	`addi rd, x0, SignExt(imm)`
01 (1)	011–––	–	CI	`c.lui rd, imm`	`lui rd, {14{imm5}, imm}`
01 (1)	011–––	–	CI	`c.addi16sp sp, imm`	`addi sp, sp, SignExt(imm)*16`
01 (1)	100–00	–	CB'	`c.srli rd', imm`	`srli rd', rd', imm`
01 (1)	100–01	–	CB'	`c.srai rd', imm`	`srai rd', rd', imm`
01 (1)	100–10	–	CB'	`c.andi rd', imm`	`andi rd', rd', SignExt(imm)`
01 (1)	100011	00	CS'	`c.sub rd', rs2'`	`sub rd', rd', rs2'`
01 (1)	100011	01	CS'	`c.xor rd', rs2'`	`xor rd', rd', rs2'`
01 (1)	100011	10	CS'	`c.or rd', rs2'`	`or rd', rd', rs2'`
01 (1)	100011	11	CS'	`c.and rd', rs2'`	`and rd', rd', rs2'`
01 (1)	101–––	–	CJ	`c.j label`	`jal x0, label`
01 (1)	110–––	–	CB	`c.beqz rs1', label`	`beq rs1', x0, label`
01 (1)	111–––	–	CB	`c.bnez rs1', label`	`bne rs1', x0, label`
10 (2)	000–––	–	CI	`c.slli rd, imm`	`slli rd, rd, imm`
10 (2)	001–––	–	CI	`c.fldsp fd, imm`	`fld fd, (ZeroExt(imm)*8)(sp)`
10 (2)	010–––	–	CI	`c.lwsp rd, imm`	`lw rd, (ZeroExt(imm)*4)(sp)`
10 (2)	011–––	–	CI	`c.flwsp fd, imm`	`flw fd, (ZeroExt(imm)*4)(sp)`
10 (2)	1000––	–	CR	`c.jr rs1 (rs1≠0,rs2=0)`	`jalr x0, rs1, 0`
10 (2)	1000––	–	CR	`c.mv rd, rs2 (rd ≠0,rs2≠0)`	`add rd, x0, rs2`
10 (2)	1001––	–	CR	`c.ebreak (rs1=0,rs2=0)`	`ebreak`
10 (2)	1001––	–	CR	`c.jalr rs1 (rs1≠0,rs2=0)`	`jalr ra, rs1, 0`
10 (2)	1001––	–	CR	`c.add rd, rs2 (rs1≠0,rs2≠0)`	`add rd, rd, rs2`
10 (2)	101–––	–	CSS	`c.fsdsp fs2, imm`	`fsd fs2, (ZeroExt(imm)*8)(sp)`
10 (2)	110–––	–	CSS	`c.swsp rs2, imm`	`sw rs2, (ZeroExt(imm)*4)(sp)`
10 (2)	111–––	–	CSS	`c.fswsp fs2, imm`	`fsw fs2, (ZeroExt(imm)*4)(sp)`

rs1', rs2', rd': 3-bit register designator for registers 8–15: 000_2 = x8 or f8, 001_2 = x9 or f9, etc.

Table B.7 RISC-V pseudoinstructions

Pseudoinstruction	RISC-V Instructions	Description	Operation
`nop`	`addi x0, x0, 0`	no operation	
`li rd, imm11:0`	`addi rd, x0, imm11:0`	load 12-bit immediate	`rd = SignExtend(imm11:0)`
`li rd, imm31:0`	`lui rd, imm31:12`* `addi rd, rd, imm11:0`	load 32-bit immediate	`rd = imm31:0`
`mv rd, rs1`	`addi rd, rs1, 0`	move (also called "register copy")	`rd = rs1`
`not rd, rs1`	`xori rd, rs1, −1`	one's complement	`rd = ~rs1`
`neg rd, rs1`	`sub rd, x0, rs1`	two's complement	`rd = −rs1`
`seqz rd, rs1`	`sltiu rd, rs1, 1`	set if = 0	`rd = (rs1 == 0)`
`snez rd, rs1`	`sltu rd, x0, rs1`	set if ≠ 0	`rd = (rs1 ≠ 0)`
`sltz rd, rs1`	`slt rd, rs1, x0`	set if < 0	`rd = (rs1 < 0)`
`sgtz rd, rs1`	`slt rd, x0, rs1`	set if > 0	`rd = (rs1 > 0)`
`beqz rs1, label`	`beq rs1, x0, label`	branch if = 0	`if (rs1 == 0) PC = label`
`bnez rs1, label`	`bne rs1, x0, label`	branch if ≠ 0	`if (rs1 ≠ 0) PC = label`
`blez rs1, label`	`bge x0, rs1, label`	branch if ≤ 0	`if (rs1 ≤ 0) PC = label`
`bgez rs1, label`	`bge rs1, x0, label`	branch if ≥ 0	`if (rs1 ≥ 0) PC = label`
`bltz rs1, label`	`blt rs1, x0, label`	branch if < 0	`if (rs1 < 0) PC = label`
`bgtz rs1, label`	`blt x0, rs1, label`	branch if > 0	`if (rs1 > 0) PC = label`
`ble rs1, rs2, label`	`bge rs2, rs1, label`	branch if ≤	`if (rs1 ≤ rs2) PC = label`
`bgt rs1, rs2, label`	`blt rs2, rs1, label`	branch if >	`if (rs1 > rs2) PC = label`
`bleu rs1, rs2, label`	`bgeu rs2, rs1, label`	branch if ≤ (unsigned)	`if (rs1 ≤ rs2) PC = label`
`bgtu rs1, rs2, label`	`bltu rs2, rs1, offset`	branch if > (unsigned)	`if (rs1 > rs2) PC = label`
`j label`	`jal x0, label`	jump	`PC = label`
`jal label`	`jal ra, label`	jump and link	`PC = label, ra = PC + 4`
`jr rs1`	`jalr x0, rs1, 0`	jump register	`PC = rs1`
`jalr rs1`	`jalr ra, rs1, 0`	jump and link register	`PC = rs1, ra = PC + 4`
`ret`	`jalr x0, ra, 0`	return from function	`PC = ra`
`call label`	`jal ra, label`	call nearby function	`PC = label, ra = PC + 4`
`call label`	`auipc ra, offset31:12`* `jalr ra, ra, offset11:0`	call far away function	`PC = PC + offset, ra = PC + 4`
`la rd, symbol`	`auipc rd, symbol31:12`* `addi rd, rd, symbol11:0`	load address of global variable	`rd = PC + symbol`
`l{b|h|w} rd, symbol`	`auipc rd, symbol31:12`* `l{b|h|w} rd, symbol11:0(rd)`	load global variable	`rd = [PC + symbol]`
`s{b|h|w} rs2, symbol, rs1`	`auipc rs1, symbol31:12`* `s{b|h|w} rs2, symbol11:0(rs1)`	store global variable	`[PC + symbol] = rs2`
`csrr rd, csr`	`csrrs rd, csr, x0`	read CSR	`rd = csr`
`csrw csr, rs1`	`csrrw x0, csr, rs1`	write CSR	`csr = rs1`

* If bit 11 of the immediate / offset / symbol is 1, the upper immediate is incremented by 1. `symbol` and `offset` are the 32-bit PC-relative addresses of a label and a global variable, respectively.

Table B.8 Privileged / CSR instructions

op	funct3	Type	Instruction	Description	Operation
1110011 (115)	000	I	`ecall`	transfer control to OS (imm=0)	
1110011 (115)	000	I	`ebreak`	transfer control to debugger (imm=1)	
1110011 (115)	000	I	`uret`	return from user exception (rs1=0,rd=0,imm=2)	`PC = uepc`
1110011 (115)	000	I	`sret`	return from supervisor exception (rs1=0,rd=0,imm=258)	`PC = sepc`
1110011 (115)	000	I	`mret`	return from machine exception (rs1=0,rd=0,imm=770)	`PC = mepc`
1110011 (115)	001	I	`csrrw rd,csr,rs1`	CSR read/write (imm=CSR number)	`rd = csr, csr = rs1`
1110011 (115)	010	I	`csrrs rd,csr,rs1`	CSR read/set (imm=CSR number)	`rd = csr, csr = csr | rs1`
1110011 (115)	011	I	`csrrc rd,csr,rs1`	CSR read/clear (imm=CSR number)	`rd = csr, csr = csr & ~rs1`
1110011 (115)	101	I	`csrrwi rd,csr,uimm`	CSR read/write immediate (imm=CSR number)	`rd = csr, csr = ZeroExt(uimm)`
1110011 (115)	110	I	`csrrsi rd,csr,uimm`	CSR read/set immediate (imm=CSR number)	`rd = csr, csr = csr | ZeroExt(uimm)`
1110011 (115)	111	I	`csrrci rd,csr,uimm`	CSR read/clear immediate (imm=CSR number)	`rd = csr, csr = csr & ~ZeroExt(uimm)`

For privileged / CSR instructions, the 5-bit unsigned immediate, uimm, is encoded in the **rs1** field.

RISC-V Instruction Set Summary

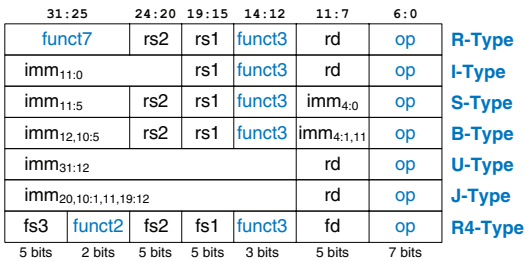


Figure B.1 **RISC-V 32-bit instruction formats**

- `imm`: signed immediate in $\mathbf{imm}_{11:0}$
- `uimm`: 5-bit unsigned immediate in $\mathbf{imm}_{4:0}$
- `upimm`: 20 upper bits of a 32-bit immediate, in $\mathbf{imm}_{31:12}$
- `Address`: memory address: **rs1** + SignExt($\mathbf{imm}_{11:0}$)
- `[Address]`: data at memory location Address
- `BTA`: branch target address: PC + SignExt({$\mathbf{imm}_{12:1}$, 1'b0})
- `JTA`: jump target address: PC + SignExt({$\mathbf{imm}_{20:1}$, 1'b0})
- `label`: text indicating instruction address
- `SignExt`: value sign-extended to 32 bits
- `ZeroExt`: value zero-extended to 32 bits
- `csr`: control and status register

Table B.1 **RV32I: RISC-V integer instructions**

op	funct3	funct7	Type	Instruction	Description	Operation
0000011 (3)	000	–	I	`lb rd, imm(rs1)`	load byte	`rd = SignExt([Address]7:0)`
0000011 (3)	001	–	I	`lh rd, imm(rs1)`	load half	`rd = SignExt([Address]15:0)`
0000011 (3)	010	–	I	`lw rd, imm(rs1)`	load word	`rd = [Address]31:0`
0000011 (3)	100	–	I	`lbu rd, imm(rs1)`	load byte unsigned	`rd = ZeroExt([Address]7:0)`
0000011 (3)	101	–	I	`lhu rd, imm(rs1)`	load half unsigned	`rd = ZeroExt([Address]15:0)`
0010011 (19)	000	–	I	`addi rd, rs1, imm`	add immediate	`rd = rs1 + SignExt(imm)`
0010011 (19)	001	0000000*	I	`slli rd, rs1, uimm`	shift left logical immediate	`rd = rs1 << uimm`
0010011 (19)	010	–	I	`slti rd, rs1, imm`	set less than immediate	`rd = (rs1 < SignExt(imm))`
0010011 (19)	011	–	I	`sltiu rd, rs1, imm`	set less than imm. unsigned	`rd = (rs1 < SignExt(imm))`
0010011 (19)	100	–	I	`xori rd, rs1, imm`	xor immediate	`rd = rs1 ^ SignExt(imm)`
0010011 (19)	101	0000000*	I	`srli rd, rs1, uimm`	shift right logical immediate	`rd = rs1 >> uimm`
0010011 (19)	101	0100000*	I	`srai rd, rs1, uimm`	shift right arithmetic imm.	`rd = rs1 >>> uimm`
0010011 (19)	110	–	I	`ori rd, rs1, imm`	or immediate	`rd = rs1 | SignExt(imm)`
0010011 (19)	111	–	I	`andi rd, rs1, imm`	and immediate	`rd = rs1 & SignExt(imm)`
0010111 (23)	–	–	U	`auipc rd, upimm`	add upper immediate to PC	`rd = {upimm, 12'b0} + PC`
0100011 (35)	000	–	S	`sb rs2, imm(rs1)`	store byte	`[Address]7:0 = rs27:0`
0100011 (35)	001	–	S	`sh rs2, imm(rs1)`	store half	`[Address]15:0 = rs215:0`
0100011 (35)	010	–	S	`sw rs2, imm(rs1)`	store word	`[Address]31:0 = rs2`
0110011 (51)	000	0000000	R	`add rd, rs1, rs2`	add	`rd = rs1 + rs2`
0110011 (51)	000	0100000	R	`sub rd, rs1, rs2`	sub	`rd = rs1 – rs2`
0110011 (51)	001	0000000	R	`sll rd, rs1, rs2`	shift left logical	`rd = rs1 << rs24:0`
0110011 (51)	010	0000000	R	`slt rd, rs1, rs2`	set less than	`rd = (rs1 < rs2)`
0110011 (51)	011	0000000	R	`sltu rd, rs1, rs2`	set less than unsigned	`rd = (rs1 < rs2)`
0110011 (51)	100	0000000	R	`xor rd, rs1, rs2`	xor	`rd = rs1 ^ rs2`
0110011 (51)	101	0000000	R	`srl rd, rs1, rs2`	shift right logical	`rd = rs1 >> rs24:0`
0110011 (51)	101	0100000	R	`sra rd, rs1, rs2`	shift right arithmetic	`rd = rs1 >>> rs24:0`
0110011 (51)	110	0000000	R	`or rd, rs1, rs2`	or	`rd = rs1 | rs2`
0110011 (51)	111	0000000	R	`and rd, rs1, rs2`	and	`rd = rs1 & rs2`
0110111 (55)	–	–	U	`lui rd, upimm`	load upper immediate	`rd = {upimm, 12'b0}`
1100011 (99)	000	–	B	`beq rs1, rs2, label`	branch if =	`if (rs1 == rs2) PC = BTA`
1100011 (99)	001	–	B	`bne rs1, rs2, label`	branch if ≠	`if (rs1 ≠ rs2) PC = BTA`
1100011 (99)	100	–	B	`blt rs1, rs2, label`	branch if <	`if (rs1 < rs2) PC = BTA`
1100011 (99)	101	–	B	`bge rs1, rs2, label`	branch if ≥	`if (rs1 ≥ rs2) PC = BTA`
1100011 (99)	110	–	B	`bltu rs1, rs2, label`	branch if < unsigned	`if (rs1 < rs2) PC = BTA`
1100011 (99)	111	–	B	`bgeu rs1, rs2, label`	branch if ≥ unsigned	`if (rs1 ≥ rs2) PC = BTA`
1100111 (103)	000	–	I	`jalr rd, rs1, imm`	jump and link register	`PC = rs1 + SignExt(imm), rd = PC + 4`
1101111 (111)	–	–	J	`jal rd, label`	jump and link	`PC = JTA, rd = PC + 4`

*Encoded in $\text{instr}_{31:25}$, the upper seven bits of the immediate field

Table B.2 RV64I: Extra integer instructions

op	funct3	funct7	Type	Instruction	Description	Operation
0000011 (3)	011	–	I	`ld rd, imm(rs1)`	load double word	$rd = [Address]_{63:0}$
0000011 (3)	110	–	I	`lwu rd, imm(rs1)`	load word unsigned	$rd = ZeroExt([Address]_{31:0})$
0011011 (27)	000	–	I	`addiw rd, rs1, imm`	add immediate word	$rd = SignExt((rs1 + SignExt(imm))_{31:0})$
0011011 (27)	001	0000000	I	`slliw rd, rs1, uimm`	shift left logical immediate word	$rd = SignExt((rs1_{31:0} << uimm)_{31:0})$
0011011 (27)	101	0000000	I	`srliw rd, rs1, uimm`	shift right logical immediate word	$rd = SignExt((rs1_{31:0} >> uimm)_{31:0})$
0011011 (27)	101	0100000	I	`sraiw rd, rs1, uimm`	shift right arith. immediate word	$rd = SignExt((rs1_{31:0} >>> uimm)_{31:0})$
0100011 (35)	011	–	S	`sd rs2, imm(rs1)`	store double word	$[Address]_{63:0} = rs2$
0111011 (59)	000	0000000	R	`addw rd, rs1, rs2`	add word	$rd = SignExt((rs1 + rs2)_{31:0})$
0111011 (59)	000	0100000	R	`subw rd, rs1, rs2`	subtract word	$rd = SignExt((rs1 - rs2)_{31:0})$
0111011 (59)	001	0000000	R	`sllw rd, rs1, rs2`	shift left logical word	$rd = SignExt((rs1_{31:0} << rs2_{4:0})_{31:0})$
0111011 (59)	101	0000000	R	`srlw rd, rs1, rs2`	shift right logical word	$rd = SignExt((rs1_{31:0} >> rs2_{4:0})_{31:0})$
0111011 (59)	101	0100000	R	`sraw rd, rs1, rs2`	shift right arithmetic word	$rd = SignExt((rs1_{31:0} >>> rs2_{4:0})_{31:0})$

In RV64I, registers are 64 bits, but instructions are still 32 bits. The term "word" generally refers to a 32-bit value. In RV64I, immediate shift instructions use 6-bit immediates: $uimm_{5:0}$; but for word shifts, the most significant bit of the shift amount ($uimm_5$) must be 0. Instructions ending in "w" (for "word") operate on the lower half of the 64-bit registers. Sign- or zero-extension produces a 64-bit result.

Table B.3 RVF/D: RISC-V single- and double-precision floating-point instructions

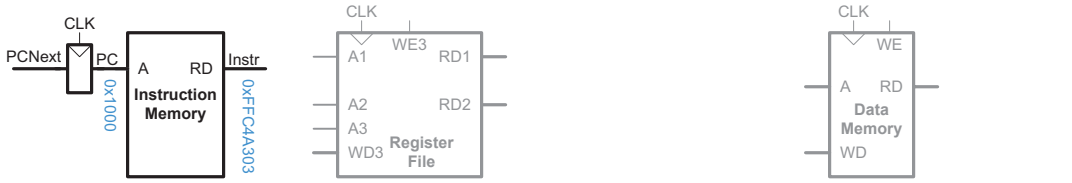
op	funct3	funct7	rs2	Type	Instruction	Description	Operation
1000011 (67)	rm	fs3, fmt	–	R4	`fmadd fd,fs1,fs2,fs3`	multiply-add	`fd = fs1 * fs2 + fs3`
1000111 (71)	rm	fs3, fmt	–	R4	`fmsub fd,fs1,fs2,fs3`	multiply-subtract	`fd = fs1 * fs2 – fs3`
1001011 (75)	rm	fs3, fmt	–	R4	`fnmsub fd,fs1,fs2,fs3`	negate multiply-add	`fd = –(fs1 * fs2 + fs3)`
1001111 (79)	rm	fs3, fmt	–	R4	`fnmadd fd,fs1,fs2,fs3`	negate multiply-subtract	`fd = –(fs1 * fs2 - fs3)`
1010011 (83)	rm	00000, fmt	–	R	`fadd fd,fs1,fs2`	add	`fd = fs1 + fs2`
1010011 (83)	rm	00001, fmt	–	R	`fsub fd,fs1,fs2`	subtract	`fd = fs1 – fs2`
1010011 (83)	rm	00010, fmt	–	R	`fmul fd,fs1,fs2`	multiply	`fd = fs1 * fs2`
1010011 (83)	rm	00011, fmt	–	R	`fdiv fd,fs1,fs2`	divide	`fd = fs1 / fs2`
1010011 (83)	rm	01011, fmt	00000	R	`fsqrt fd,fs1`	square root	`fd = sqrt(fs1)`
1010011 (83)	000	00100, fmt	–	R	`fsgnj fd,fs1,fs2`	sign injection	`fd = fs1, sign = sign(fs2)`
1010011 (83)	001	00100, fmt	–	R	`fsgnjn fd,fs1,fs2`	negate sign injection	`fd = fs1, sign = –sign(fs2)`
1010011 (83)	010	00100, fmt	–	R	`fsgnjx fd,fs1,fs2`	xor sign injection	`fd = fs1, sign = sign(fs2) ^ sign(fs1)`
1010011 (83)	000	00101, fmt	–	R	`fmin fd,fs1,fs2`	min	`fd = min(fs1, fs2)`
1010011 (83)	001	00101, fmt	–	R	`fmax fd,fs1,fs2`	max	`fd = max(fs1, fs2)`
1010011 (83)	010	10100, fmt	–	R	`feq rd,fs1,fs2`	compare =	`rd = (fs1 == fs2)`
1010011 (83)	001	10100, fmt	–	R	`flt rd,fs1,fs2`	compare <	`rd = (fs1 < fs2)`
1010011 (83)	000	10100, fmt	–	R	`fle rd,fs1,fs2`	compare ≤	`rd = (fs1 ≤ fs2)`
1010011 (83)	001	11100, fmt	00000	R	`fclass rd,fs1`	classify	`rd = classification of fs1`
RVF only							
0000111 (7)	010	–	–	I	`flw fd, imm(rs1)`	load float	$fd = [Address]_{31:0}$
0100111 (39)	010	–	–	S	`fsw fs2,imm(rs1)`	store float	$[Address]_{31:0} = fd$
1010011 (83)	rm	1100000	00000	R	`fcvt.w.s rd, fs1`	convert to integer	`rd = integer(fs1)`
1010011 (83)	rm	1100000	00001	R	`fcvt.wu.s rd, fs1`	convert to unsigned integer	`rd = unsigned(fs1)`
1010011 (83)	rm	1101000	00000	R	`fcvt.s.w fd, rs1`	convert int to float	`fd = float(rs1)`
1010011 (83)	rm	1101000	00001	R	`fcvt.s.wu fd, rs1`	convert unsigned to float	`fd = float(rs1)`
1010011 (83)	000	1110000	00000	R	`fmv.x.w rd, fs1`	move to integer register	`rd = fs1`
1010011 (83)	000	1111000	00000	R	`fmv.w.x fd, rs1`	move to f.p. register	`fd = rs1`
RVD only							
0000111 (7)	011	-	–	I	`fld fd, imm(rs1)`	load double	$fd = [Address]_{63:0}$
0100111 (39)	011	–	–	S	`fsd fs2,imm(rs1)`	store double	$[Address]_{63:0} = fd$
1010011 (83)	rm	1100001	00000	R	`fcvt.w.d rd, fs1`	convert to integer	`rd = integer(fs1)`
1010011 (83)	rm	1100001	00001	R	`fcvt.wu.d rd, fs1`	convert to unsigned integer	`rd = unsigned(fs1)`
1010011 (83)	rm	1101001	00000	R	`fcvt.d.w fd, rs1`	convert int to double	`fd = double(rs1)`
1010011 (83)	rm	1101001	00001	R	`fcvt.d.wu fd, rs1`	convert unsigned to double	`fd = double(rs1)`
1010011 (83)	rm	0100000	00001	R	`fcvt.s.d fd, fs1`	convert double to float	`fd = float(fs1)`
1010011 (83)	rm	0100001	00000	R	`fcvt.d.s fd, fs1`	convert float to double	`fd = double(fs1)`

fs1, **fs2**, **fs3**, **fd**: floating-point registers. **fs1**, **fs2**, and **fd** are encoded in fields **rs1**, **rs2**, and **rd**; only R4-type also encodes **fs3**. **fmt**: precision of computational instruction (single=00_2, double=01_2, quad=11_2). **rm**: rounding mode (0=to nearest, 1=toward zero, 2=down, 3=up, 4=to nearest (max magnitude), 7=dynamic). sign(**fs1**): the sign of **fs1**.

Appendix B : Processor Architecture Overview and Assembly-Level Verification

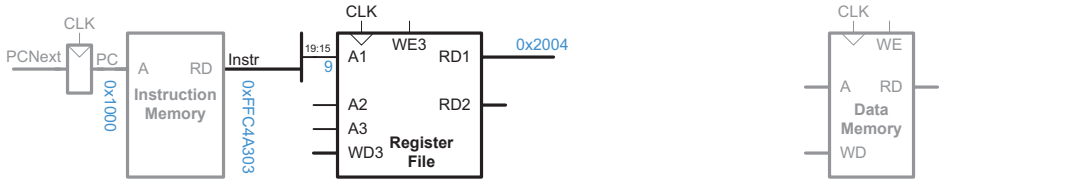
Address	Instruction	Type	Fields					Machine Language
			$imm_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303
			$imm_{11:5}$	rs2	rs1	f3	$imm_{4:0}$	op	
0x1004	sw x6, 8(x9)	S	0000000	00110	01001	010	01000	0100011	0064A423
			funct7	rs2	rs1	f3	rd	op	
0x1008	or x4, x5, x6	R	0000000	00110	00101	110	00100	0110011	0062E233
			$imm_{12,10:5}$	rs2	rs1	f3	$imm_{4:1,11}$	op	
0x100C	beq x4, x4, L7	B	1111111	00100	00100	000	10101	1100011	FE420AE3

Figure 7.2 Sample program exercising different types of instructions



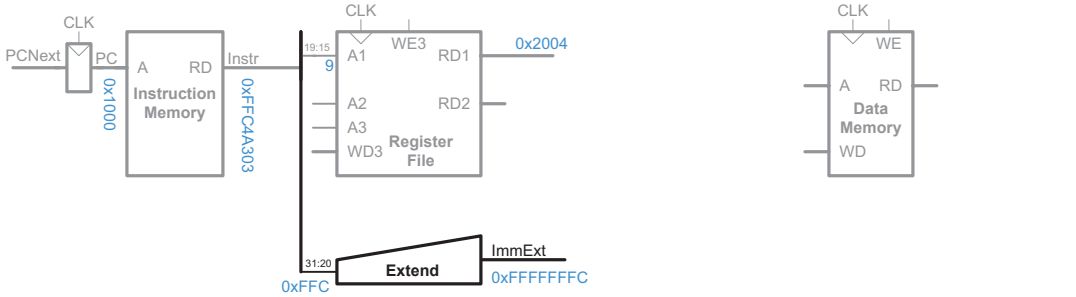
Address	Instruction	Type	Fields					Machine Language
			$imm_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

Figure 7.3 Fetch instruction from memory



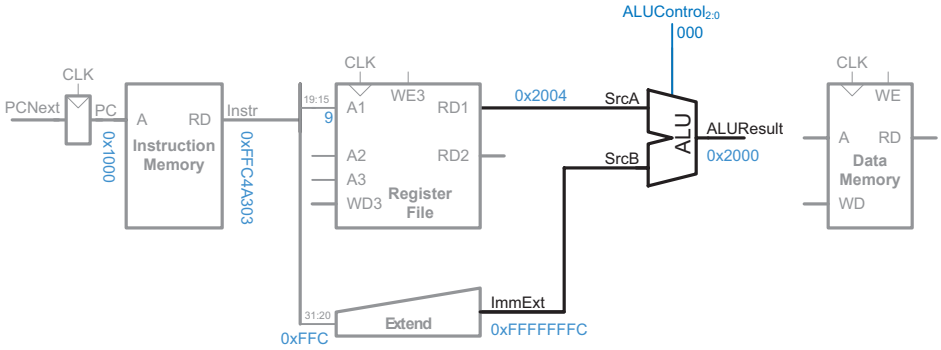
Address	Instruction	Type	Fields					Machine Language
			$imm_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

Figure 7.4 Read source operand from register file



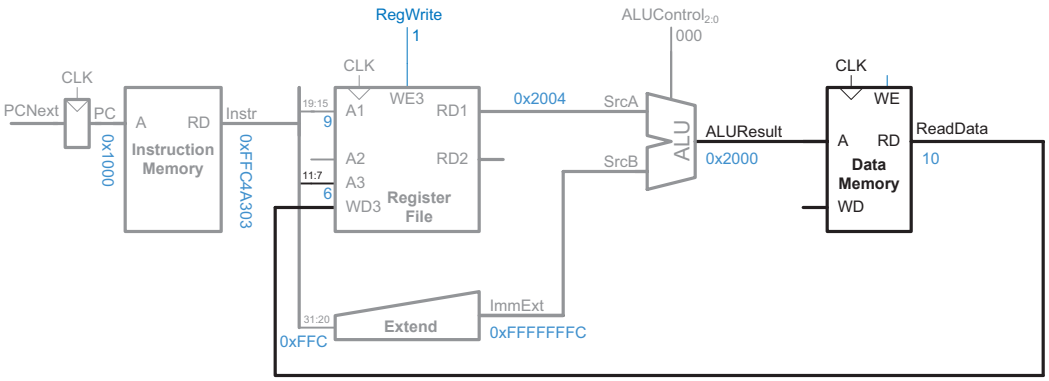
Address	Instruction	Type	Fields					Machine Language
			$imm_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

Figure 7.5 Sign-extend the immediate



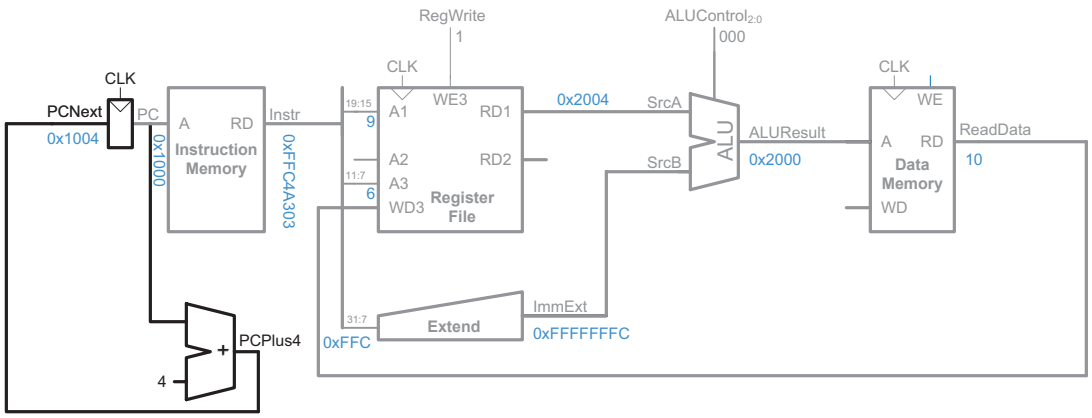
Address	Instruction	Type	Fields					Machine Language
			$\mathbf{imm_{11:0}}$	**rs1**	**f3**	**rd**	**op**	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

Figure 7.6 Compute memory address



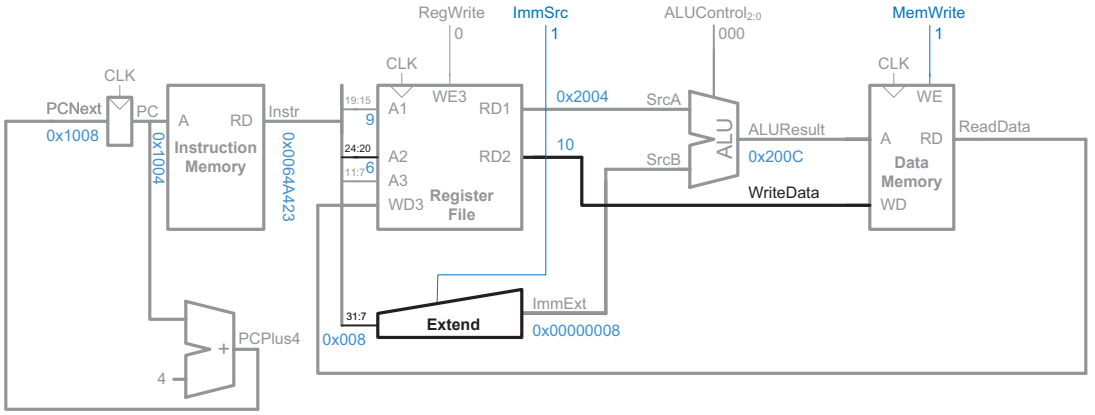
Address	Instruction	Type	Fields					Machine Language
			$\mathbf{imm_{11:0}}$	**rs1**	**f3**	**rd**	**op**	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

Figure 7.7 Read memory and write result back to register file

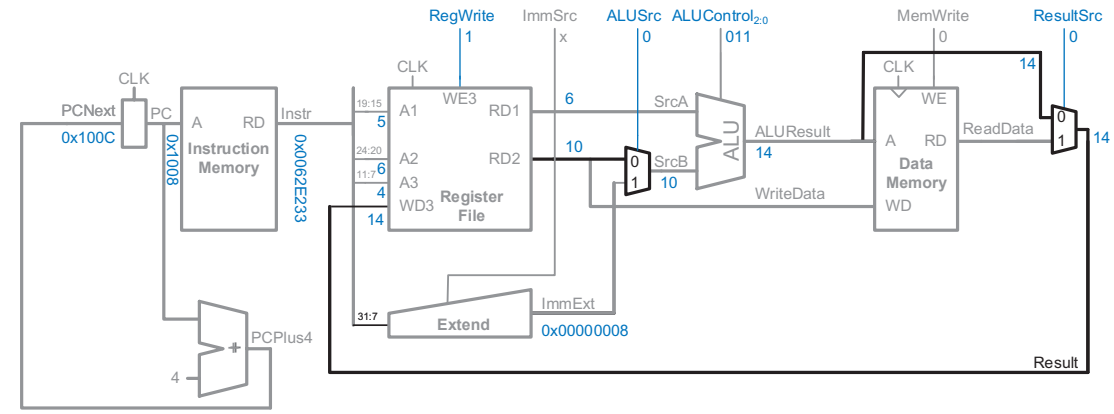


Address	Instruction	Type	Fields					Machine Language
			$\mathbf{imm_{11:0}}$	**rs1**	**f3**	**rd**	**op**	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

Figure 7.8 Increment program counter



Address	Instruction	Type	Fields					Machine Language
			$imm_{11:5}$	rs2	rs1	f3	$imm_{4:0}$	op	
0x1004	sw x6, 8(x9)	S	0000000	00110	01001	010	01000	0100011	0064A423

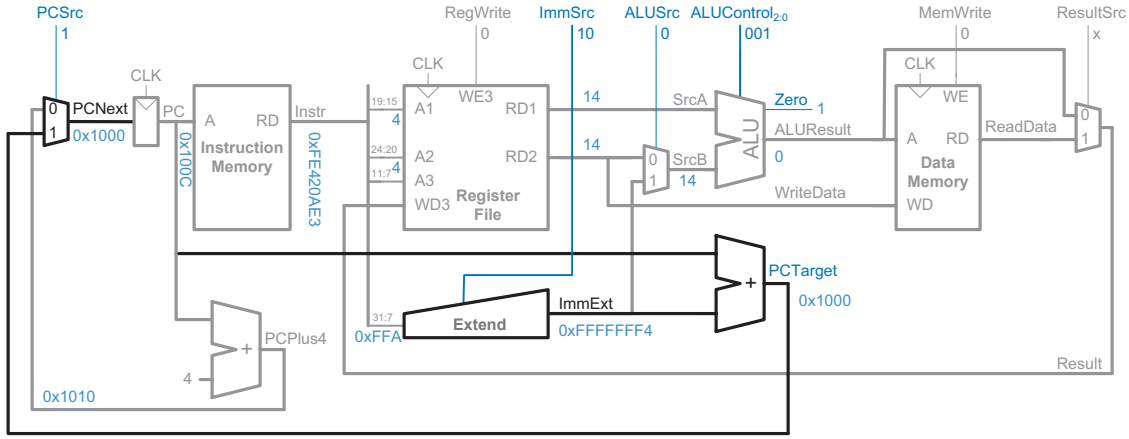


Address	Instruction	Type	Fields						Machine Language
			funct7	rs2	rs1	f3	rd	op	
0x1008	or x4,	R	0000000		00101	110			0062E233

Figure 7.10 Datapath enhancements for R-type instructions

Table 7.1 ***ImmSrc*** **encoding**

ImmSrc	ImmExt	Type	Description
00	{{20{*Instr*[31]}}, *Instr*[31:20]}	I	12-bit signed immediate
01	{{20{*Instr*[31]}}, *Instr*[31:25], *Instr*[11:7]}	S	12-bit signed immediate
10	{{20{*Instr*[31]}}, *Instr*[7], *Instr*[30:25], *Instr*[11:8], 1'b0}	B	13-bit signed immediate



Address	Instruction	Type	Fields					Machine Language
			$\text{imm}_{12,10:5}$	rs2	rs1	f3	$\text{imm}_{4:1,11}$	op	
0x100C	`beq x4, x4, L7`	B	1111111	00100	00100	000	10101	1100011	FE420AE3

Figure 7.11 **Datapath enhancements for** `beq`

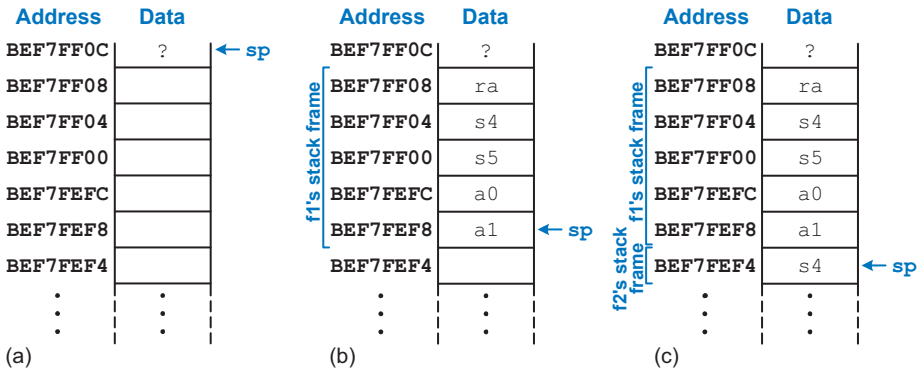


Figure 6.9 **The stack: (a) before function calls, (b) during** `f1`**, and (c) during** `f2`

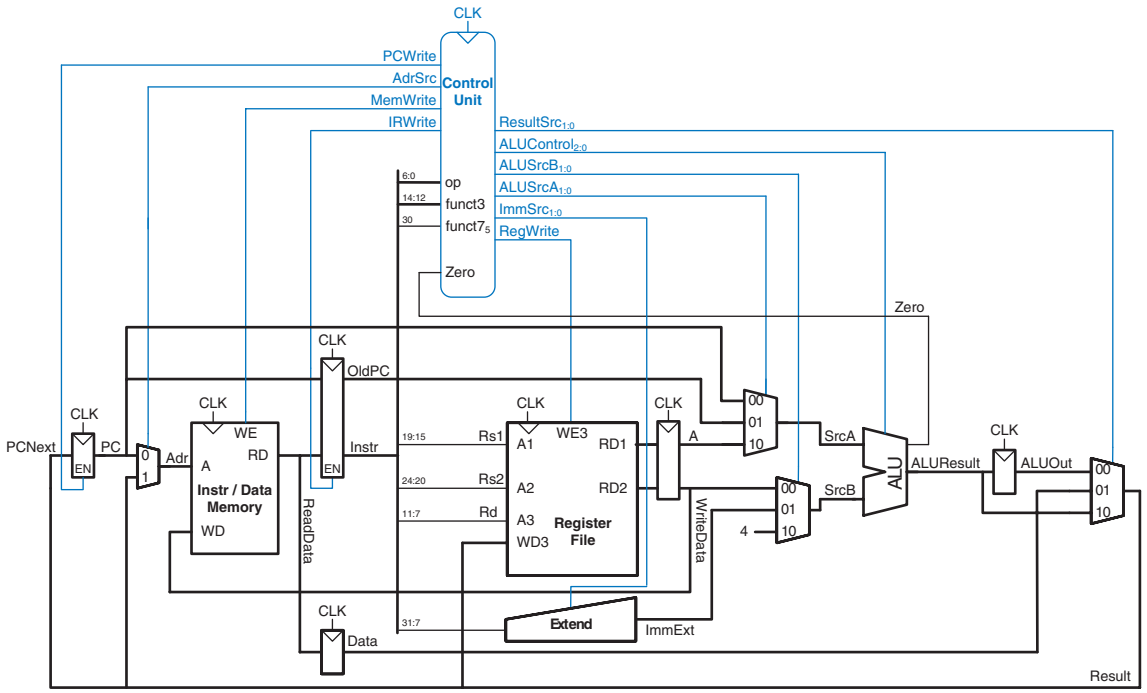


Figure 7.27 Complete multicycle processor

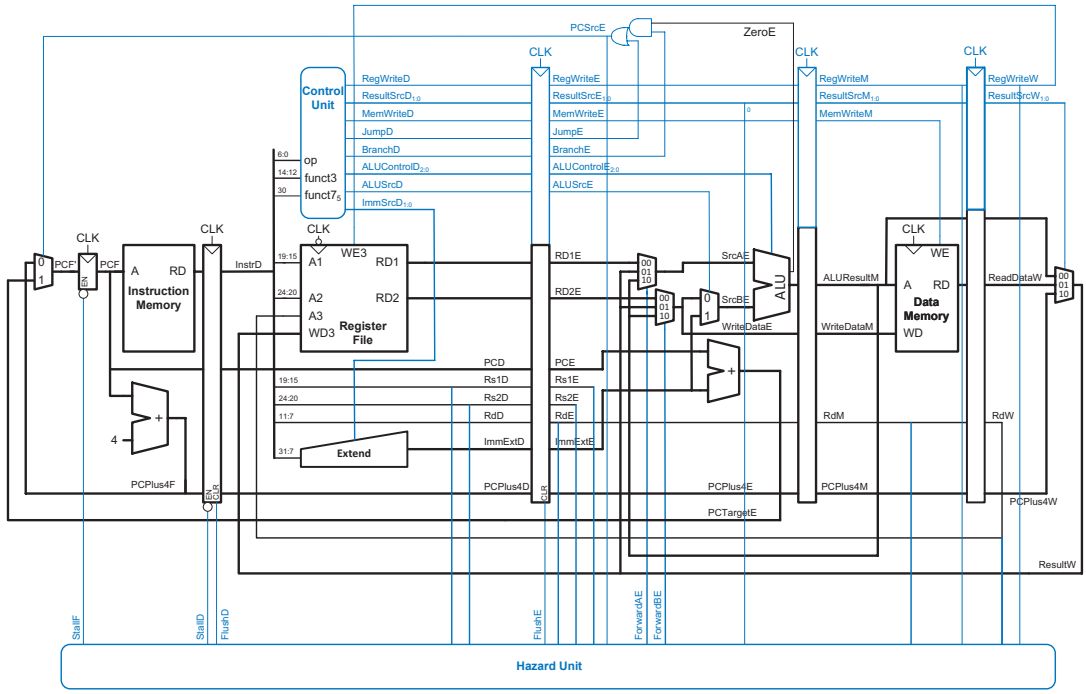


Figure 7.61 Pipelined processor with full hazard handling

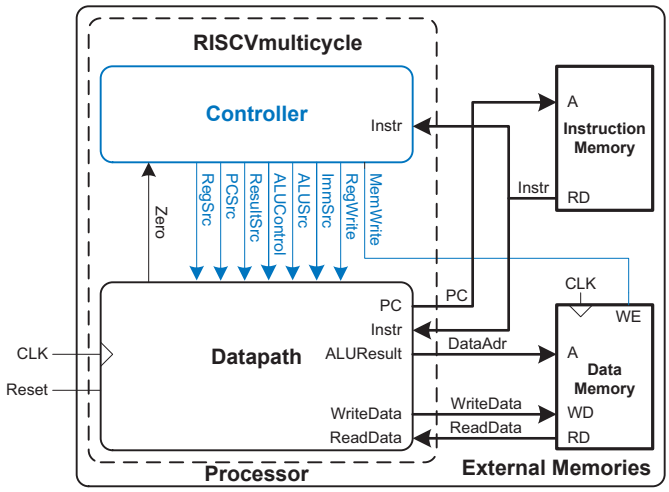
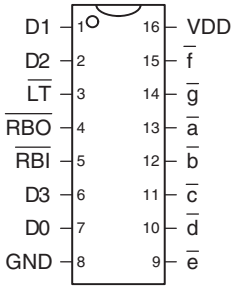


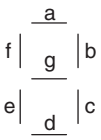
Figure 7.63 **Multi-cycle processor interfaced to external memories**



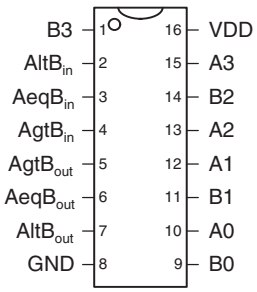
7447 7-Segment Decoder

7-segment Display Decoder

$D_{3:0}$: data
a...f: segments (low = ON)
LTb: light test
RBIb: ripple blanking in
RBOb: ripple blanking out



RBO	LT	RBI	D3:0	a	b	c	d	e	f	g
0	x	x	x	1	1	1	1	1	1	1
1	0	x	x	0	0	0	0	0	0	0
x	1	0	0000	1	1	1	1	1	1	1
1	1	1	0000	0	0					1
1	1	1	0001	1	0					1
1	1	1	0010	0	0					0
1	1	1	0011	0	0					0
1	1	1	0100	1	0					0
1	1	1	0101	0	1					0
1	1	1	0110	1	1					0
1	1	1	0111	0	0					1
1	1	1	1000	0	0					0
1	1	1	1001	0	0					0
1	1	1	1010	1	1					0
1	1	1	1011	1	1					0
1	1	1	1100	1	0					0
1	1	1	1101	0	1					0
1	1	1	1110	0	0					1
1		1	1	0	0	0	0	0	0	0

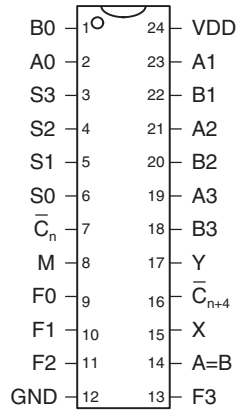


7485 Comparator

4-bit Comparator

$A_{3:0}$, $B_{3:0}$: data
rel_{in}: input relation
rel_{out}: output relation

```
always_comb
   if (A > B | (A == B & AgtBin)) begin
     AgtBout = 1; AeqBout = 0; AltBout = 0;
   end
   else if (A < B | (A == B & AltBin) begin
     AgtBout = 0; AeqBout = 0; AltBout = 1;
   end else begin
     AgtBout = 0; AeqBout = 1; AltBout = 0;
   end
```



74181 ALU

4-bit ALU

$A_{3:0}$, $B_{3:0}$: inputs
$Y_{3:0}$: output
$F_{3:0}$: function select
M: mode select
Cb_n: carry in
Cb_{nplus4}: carry out
AeqB: equality (in some modes)
X,Y: carry lookahead adder outputs

```
always_comb
  case (F)
    0000: Y = M ? ~A       :  A       +  B       + ~Cbn;
    0001: Y = M ? ~(A | B) :  A       +  ~B      + ~Cbn;
    0010: Y = M ? (~A) & B :  A                   + ~Cbn;
    0011: Y = M ? 4'b0000  :  4'b1111 +  (A & ~B) +
                                                    ~Cbn;
    0100: Y = M ? ~(A & B) :  A       + (A & ~B)  + ~Cbn;
    0101: Y = M ? ~B       : (A | B)  - B         -  Cbn;
    0110: Y = M ? A ^ B    :  A       - (A & ~B)  -  Cbn;
    0111: Y = M ? A & ~B   : (A & ~B)              -  Cbn;
    1000: Y = M ? ~A + B   :  A       + (A & B)   + ~Cbn;
    1001: Y = M ? ~(A ^ B) :  A       +  B        + ~Cbn;
    1010: Y = M ? B        : (A | ~B) + (A & B)   + ~Cbn;
    1011: Y = M ? A & B    : (A & B)                + ~Cbn;
    1100: Y = M ? 1        :  A       +  A        + ~Cbn;
    1101: Y = M ? A | ~B   : (A | B)  +  A        + ~Cbn;
    1110: Y = M ? A | B    : (A | ~B) +  A        + ~Cbn;
    1111: Y = M ? A        :  A                   -  Cbn;
  endcase
```

Appendix C :
FPGA Documentation

Chapter 2

Altera DE2 Board

This chapter presents the features and design characteristics of the DE2 board.

2.1 Layout and Components

A photograph of the DE2 board is shown in Figure 2.1. It depicts the layout of the board and indicates the location of the connectors and key components.

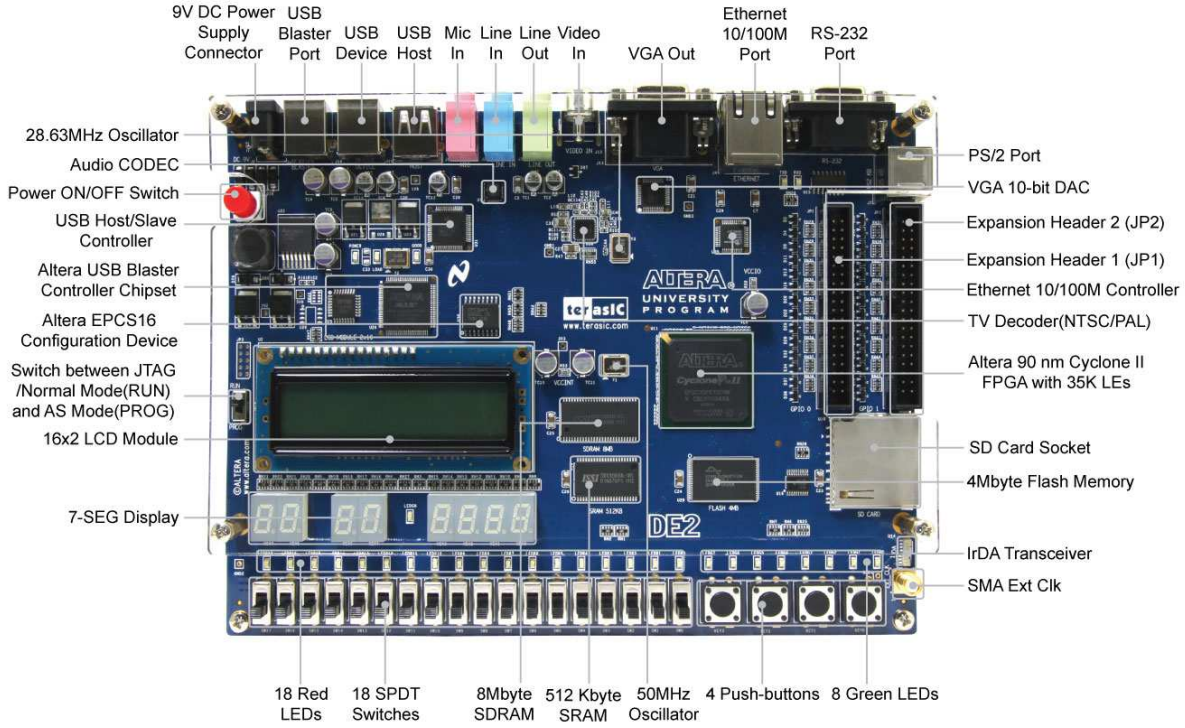


Figure 2.1. The DE2 board.

The DE2 board has many features that allow the user to implement a wide range of designed circuits, from simple circuits to various multimedia projects.

The following hardware is provided on the DE2 board:

- Altera Cyclone® II 2C35 FPGA device
- Altera Serial Configuration device - EPCS16
- USB Blaster (on board) for programming and user API control; both JTAG and Active Serial (AS) programming modes are supported

- 512-Kbyte SRAM
- 8-Mbyte SDRAM
- 4-Mbyte Flash memory (1 Mbyte on some boards)
- SD Card socket
- 4 pushbutton switches
- 18 toggle switches
- 18 red user LEDs
- 9 green user LEDs
- 50-MHz oscillator and 27-MHz (from TV decoder) for clock sources
- 24-bit CD-quality audio CODEC with line-in, line-out, and microphone-in jacks
- VGA DAC (10-bit high-speed triple DACs) with VGA-out connector
- TV Decoder (NTSC/PAL) and TV-in connector
- 10/100 Ethernet Controller with a connector
- USB Host/Slave Controller with USB type A and type B connectors
- RS-232 transceiver and 9-pin connector
- PS/2 mouse/keyboard connector
- IrDA transceiver
- Two 40-pin Expansion Headers with diode protection

In addition to these hardware features, the DE2 board has software support for standard I/O interfaces and a control panel facility for accessing various components. Also, software is provided for a number of demonstrations that illustrate the advanced capabilities of the DE2 board.

In order to use the DE2 board, the user has to be familiar with the Quartus II software. The necessary knowledge can be acquired by reading the tutorials *Getting Started with Altera's DE2 Board* and *Quartus II Introduction* (which exists in three versions based on the design entry method used, namely Verilog, VHDL or schematic entry). These tutorials are provided in the directory *DE2_tutorials* on the **DE2 System CD-ROM** that accompanies the DE2 board and can also be found on Altera's DE2 web pages.

2.2 Block Diagram of the DE2 Board

Figure 2.2 gives the block diagram of the DE2 board. To provide maximum flexibility for the user, all connections are made through the Cyclone II FPGA device. Thus, the user can configure the FPGA to implement any system design.

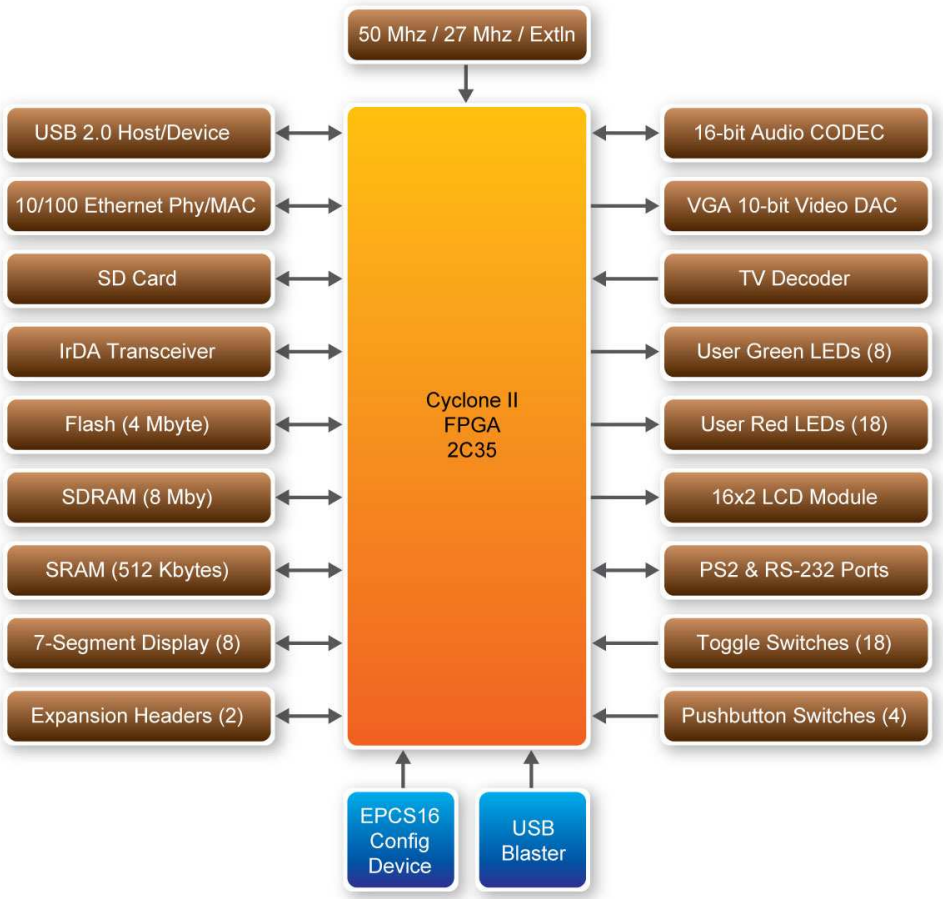


Figure 2.2. Block diagram of the DE2 board.

Following is more detailed information about the blocks in Figure 2.2:

Cyclone II 2C35 FPGA

- 33,216 LEs
- 105 M4K RAM blocks
- 483,840 total RAM bits
- 35 embedded multipliers
- 4 PLLs
- 475 user I/O pins
- FineLine BGA 672-pin package

Serial Configuration device and USB Blaster circuit

- Altera's EPCS16 Serial Configuration device
- On-board USB Blaster for programming and user API control
- JTAG and AS programming modes are supported

Configuring the FPGA in JTAG Mode

Figure 4.1 illustrates the JTAG configuration setup. To download a configuration bit stream into the Cyclone II FPGA, perform the following steps:

- Ensure that power is applied to the DE2 board
- Connect the supplied USB cable to the USB Blaster port on the DE2 board (see Figure 2.1)
- Configure the JTAG programming circuit by setting the RUN/PROG switch (on the left side of the board) to the RUN position.
- The FPGA can now be programmed by using the Quartus II Programmer module to select a configuration bit stream file with the *.sof* filename extension

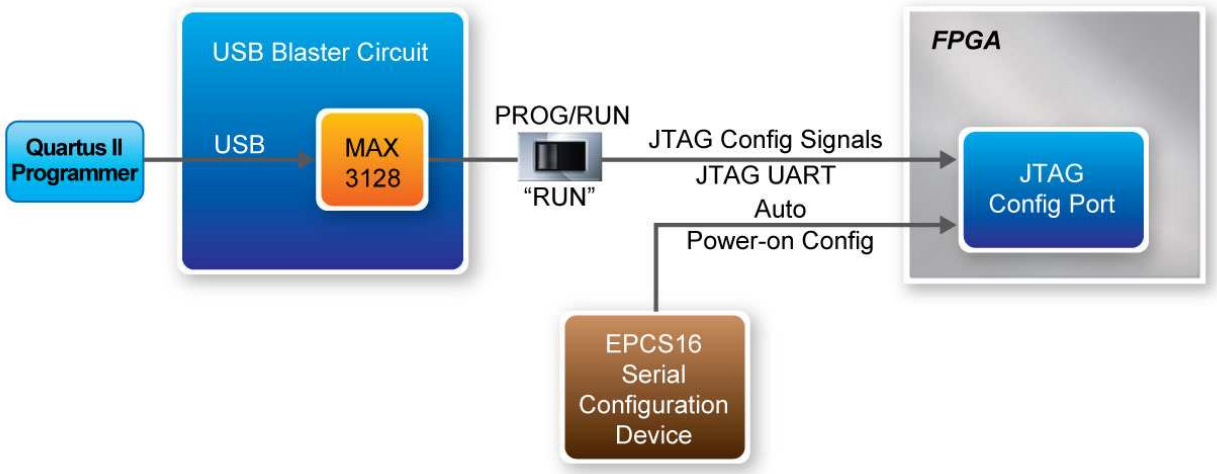


Figure 4.1. The JTAG configuration scheme.

Configuring the EPCS16 in AS Mode

Figure 4.2 illustrates the AS configuration set up. To download a configuration bit stream into the EPCS16 serial EEPROM device, perform the following steps:

- Ensure that power is applied to the DE2 board
- Connect the supplied USB cable to the USB Blaster port on the DE2 board (see Figure 2.1)
- Configure the JTAG programming circuit by setting the RUN/PROG switch (on the left side of the board) to the PROG position.
- The EPCS16 chip can now be programmed by using the Quartus II Programmer module to select a configuration bit stream file with the *.pof* filename extension
- Once the programming operation is finished, set the RUN/PROG switch back to the RUN position and then reset the board by turning the power switch off and back on; this action causes the new configuration data in the EPCS16 device to be loaded into the FPGA chip.

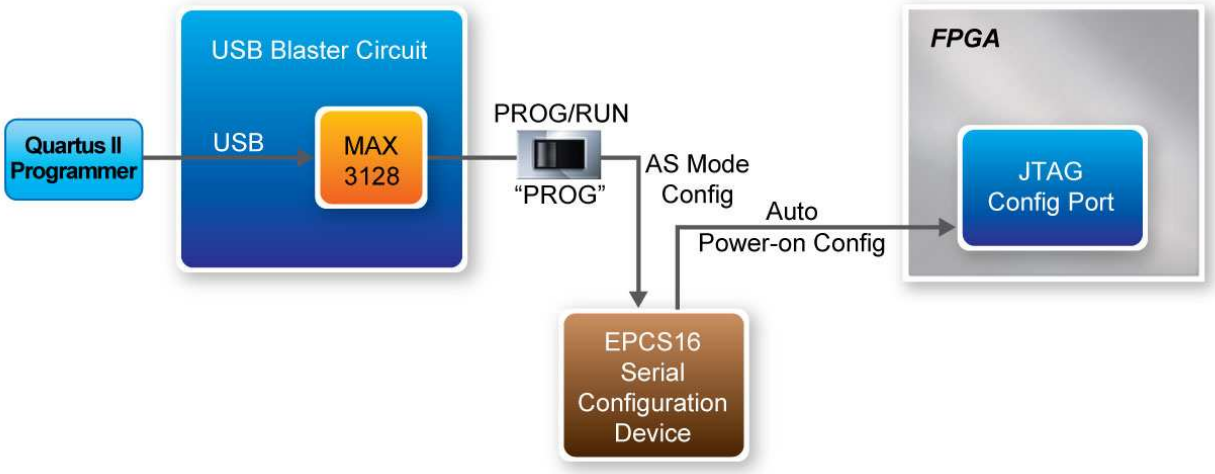


Figure 4.2. The AS configuration scheme.

In addition to its use for JTAG and AS programming, the USB Blaster port on the DE2 board can also be used to control some of the board's features remotely from a host computer. Details that describe this method of using the USB Blaster port are given in Chapter 3.

4.2 Using the LEDs and Switches

The DE2 board provides four pushbutton switches. Each of these switches is debounced using a Schmitt Trigger circuit, as indicated in Figure 4.3. The four outputs called *KEY0*, …, *KEY3* of the Schmitt Trigger device are connected directly to the Cyclone II FPGA. Each switch provides a high logic level (3.3 volts) when it is not pressed, and provides a low logic level (0 volts) when depressed. Since the pushbutton switches are debounced, they are appropriate for use as clock or reset inputs in a circuit.

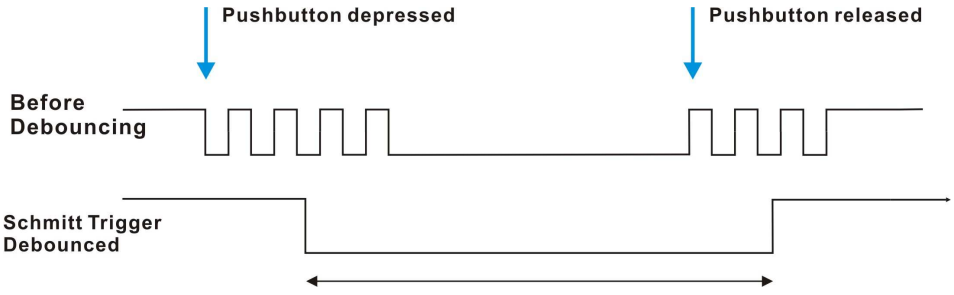


Figure 4.3. Switch debouncing.

There are also 18 toggle switches (sliders) on the DE2 board. These switches are not debounced,

Signal Name	FPGA Pin No.	Description
KEY[0]	PIN_G26	Pushbutton[0]
KEY[1]	PIN_N23	Pushbutton[1]
KEY[2]	PIN_P23	Pushbutton[2]
KEY[3]	PIN_W26	Pushbutton[3]

Table 4.2. Pin assignments for the pushbutton switches.

Signal Name	FPGA Pin No.	Description
LEDR[0]	PIN_AE23	LED Red[0]
LEDR[1]	PIN_AF23	LED Red[1]
LEDR[2]	PIN_AB21	LED Red[2]
LEDR[3]	PIN_AC22	LED Red[3]
LEDR[4]	PIN_AD22	LED Red[4]
LEDR[5]	PIN_AD23	LED Red[5]
LEDR[6]	PIN_AD21	LED Red[6]
LEDR[7]	PIN_AC21	LED Red[7]
LEDR[8]	PIN_AA14	LED Red[8]
LEDR[9]	PIN_Y13	LED Red[9]
LEDR[10]	PIN_AA13	LED Red[10]
LEDR[11]	PIN_AC14	LED Red[11]
LEDR[12]	PIN_AD15	LED Red[12]
LEDR[13]	PIN_AE15	LED Red[13]
LEDR[14]	PIN_AF13	LED Red[14]
LEDR[15]	PIN_AE13	LED Red[15]
LEDR[16]	PIN_AE12	LED Red[16]
LEDR[17]	PIN_AD12	LED Red[17]
LEDG[0]	PIN_AE22	LED Green[0]
LEDG[1]	PIN_AF22	LED Green[1]
LEDG[2]	PIN_W19	LED Green[2]
LEDG[3]	PIN_V18	LED Green[3]
LEDG[4]	PIN_U18	LED Green[4]
LEDG[5]	PIN_U17	LED Green[5]
LEDG[6]	PIN_AA20	LED Green[6]
LEDG[7]	PIN_Y18	LED Green[7]
LEDG[8]	PIN_Y12	LED Green[8]

Table 4.3. Pin assignments for the LEDs.

4.3 Using the 7-segment Displays

The DE2 Board has eight 7-segment displays. These displays are arranged into two pairs and a group of four, with the intent of displaying numbers of various sizes. As indicated in the schematic in Figure 4.6, the seven segments are connected to pins on the Cyclone II FPGA. Applying a low logic level to a segment causes it to light up, and applying a high logic level turns it off.

Each segment in a display is identified by an index from 0 to 6, with the positions given in Figure 4.7. Note that the dot in each display is unconnected and cannot be used. Table 4.4 shows the assignments of FPGA pins to the 7-segment displays.

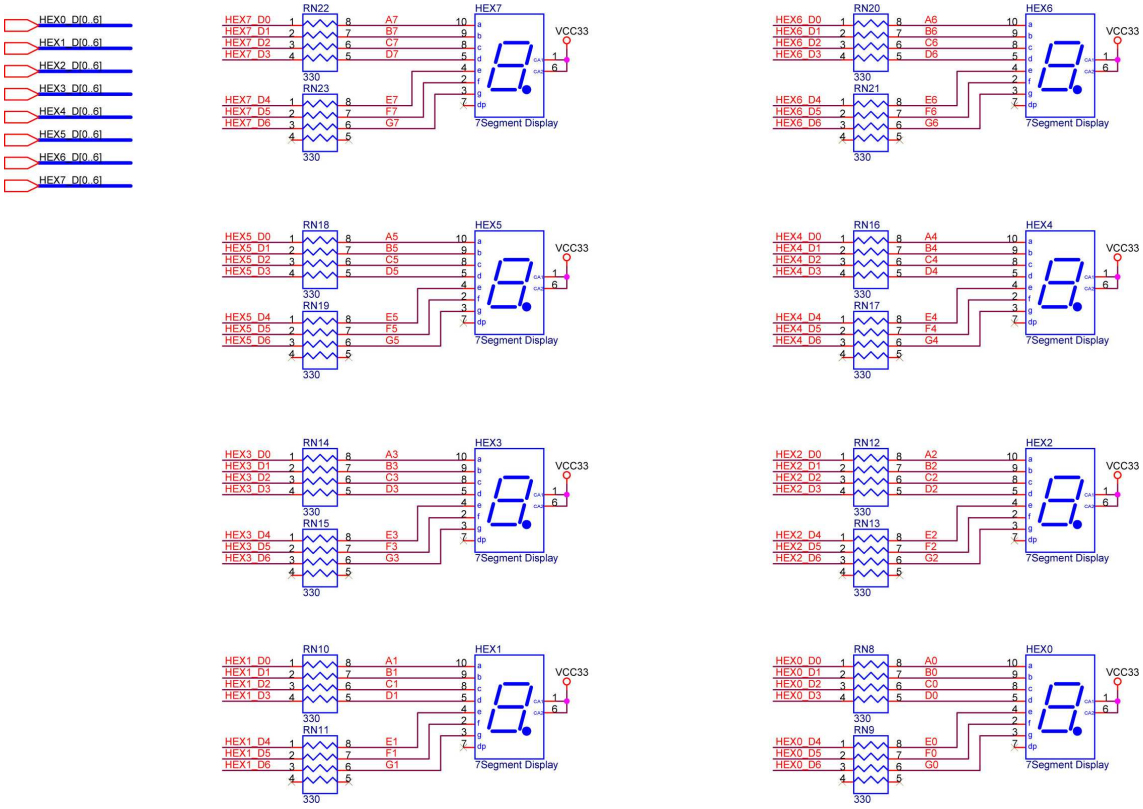


Figure 4.6. Schematic diagram of the 7-segment displays.

Figure 4.7. Position and index of each segment in a 7-segment display.

Signal Name	FPGA Pin No.	Description
HEX0[0]	PIN_AF10	Seven Segment Digit 0[0]
HEX0[1]	PIN_AB12	Seven Segment Digit 0[1]
HEX0[2]	PIN_AC12	Seven Segment Digit 0[2]
HEX0[3]	PIN_AD11	Seven Segment Digit 0[3]
HEX0[4]	PIN_AE11	Seven Segment Digit 0[4]
HEX0[5]	PIN_V14	Seven Segment Digit 0[5]
HEX0[6]	PIN_V13	Seven Segment Digit 0[6]
HEX1[0]	PIN_V20	Seven Segment Digit 1[0]
HEX1[1]	PIN_V21	Seven Segment Digit 1[1]
HEX1[2]	PIN_W21	Seven Segment Digit 1[2]
HEX1[3]	PIN_Y22	Seven Segment Digit 1[3]
HEX1[4]	PIN_AA24	Seven Segment Digit 1[4]
HEX1[5]	PIN_AA23	Seven Segment Digit 1[5]
HEX1[6]	PIN_AB24	Seven Segment Digit 1[6]
HEX2[0]	PIN_AB23	Seven Segment Digit 2[0]
HEX2[1]	PIN_V22	Seven Segment Digit 2[1]
HEX2[2]	PIN_AC25	Seven Segment Digit 2[2]
HEX2[3]	PIN_AC26	Seven Segment Digit 2[3]
HEX2[4]	PIN_AB26	Seven Segment Digit 2[4]
HEX2[5]	PIN_AB25	Seven Segment Digit 2[5]
HEX2[6]	PIN_Y24	Seven Segment Digit 2[6]
HEX3[0]	PIN_Y23	Seven Segment Digit 3[0]
HEX3[1]	PIN_AA25	Seven Segment Digit 3[1]
HEX3[2]	PIN_AA26	Seven Segment Digit 3[2]
HEX3[3]	PIN_Y26	Seven Segment Digit 3[3]
HEX3[4]	PIN_Y25	Seven Segment Digit 3[4]
HEX3[5]	PIN_U22	Seven Segment Digit 3[5]
HEX3[6]	PIN_W24	Seven Segment Digit 3[6]
HEX4[0]	PIN_U9	Seven Segment Digit 4[0]
HEX4[1]	PIN_U1	Seven Segment Digit 4[1]
HEX4[2]	PIN_U2	Seven Segment Digit 4[2]
HEX4[3]	PIN_T4	Seven Segment Digit 4[3]
HEX4[4]	PIN_R7	Seven Segment Digit 4[4]
HEX4[5]	PIN_R6	Seven Segment Digit 4[5]
HEX4[6]	PIN_T3	Seven Segment Digit 4[6]

HEX5[0]	PIN_T2	Seven Segment Digit 5[0]
HEX5[1]	PIN_P6	Seven Segment Digit 5[1]
HEX5[2]	PIN_P7	Seven Segment Digit 5[2]
HEX5[3]	PIN_T9	Seven Segment Digit 5[3]
HEX5[4]	PIN_R5	Seven Segment Digit 5[4]
HEX5[5]	PIN_R4	Seven Segment Digit 5[5]
HEX5[6]	PIN_R3	Seven Segment Digit 5[6]
HEX6[0]	PIN_R2	Seven Segment Digit 6[0]
HEX6[1]	PIN_P4	Seven Segment Digit 6[1]
HEX6[2]	PIN_P3	Seven Segment Digit 6[2]
HEX6[3]	PIN_M2	Seven Segment Digit 6[3]
HEX6[4]	PIN_M3	Seven Segment Digit 6[4]
HEX6[5]	PIN_M5	Seven Segment Digit 6[5]
HEX6[6]	PIN_M4	Seven Segment Digit 6[6]
HEX7[0]	PIN_L3	Seven Segment Digit 7[0]
HEX7[1]	PIN_L2	Seven Segment Digit 7[1]
HEX7[2]	PIN_L9	Seven Segment Digit 7[2]
HEX7[3]	PIN_L6	Seven Segment Digit 7[3]
HEX7[4]	PIN_L7	Seven Segment Digit 7[4]
HEX7[5]	PIN_P9	Seven Segment Digit 7[5]
HEX7[6]	PIN_N9	Seven Segment Digit 7[6]

Table 4.4. Pin assignments for the 7-segment displays.

4.4 Clock Inputs

The DE2 board includes an oscillator that produces 50 MHz clock signals. The board also includes an SMA connector which can be used to connect an external clock source to the board. The schematic of the clock circuitry is shown in Figure 4.8, and the associated pin assignments appear in Table 4.5.

Note: the source of the CLOCK27 and TD_CLK27 are both from the TV Decoder ADV7180. So when you use one of them, please remember to make assignment to the other clock pin as well, or set the unused pins in the project to "**AS input tri-stated"** in the Quartus II software. Either of these methods can be used to make sure these two clocks input into FPGA device normally, it helps to avoid the negative effect on the Project Function.

Please make sure the TD_RESET signal has been set correctly in your FPGA design. If it is not

being used, please set it to logic high status in order to avoid to reset the TV Decode. Then it will not effect to generate the 27Mhz clock.

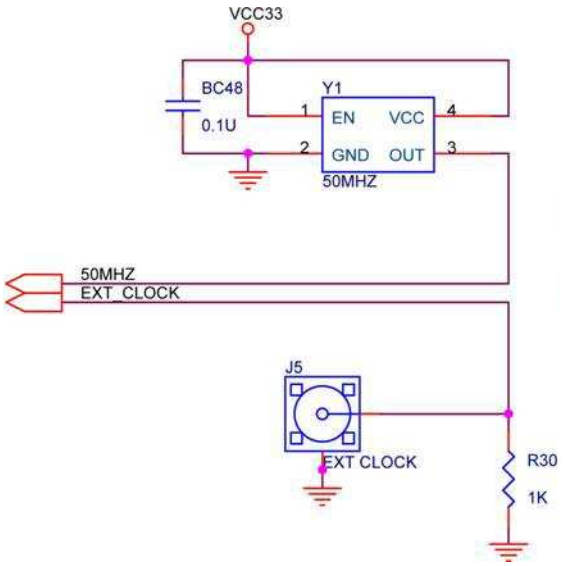


Figure 4.8. Schematic diagram of the clock circuit.

Signal Name	FPGA Pin No.	Description
CLOCK_27	PIN_D13	TV Decoder Clock Input.
CLOCK_50	PIN_N2	50 MHz clock input
EXT_CLOCK	PIN_P26	External (SMA) clock input

Table 4.5. Pin assignments for the clock inputs.

4.5 Using the LCD Module

The LCD module has built-in fonts and can be used to display text by sending appropriate commands to the display controller, which is called HD44780. Detailed information for using the display is available in its datasheet, which can be found on the manufacturer's web site, and from the *Datasheet* folder on the **DE2 System CD-ROM**. A schematic diagram of the LCD module showing connections to the Cyclone II FPGA is given in Figure 4.9. The associated pin assignments appear in Table 4.6.

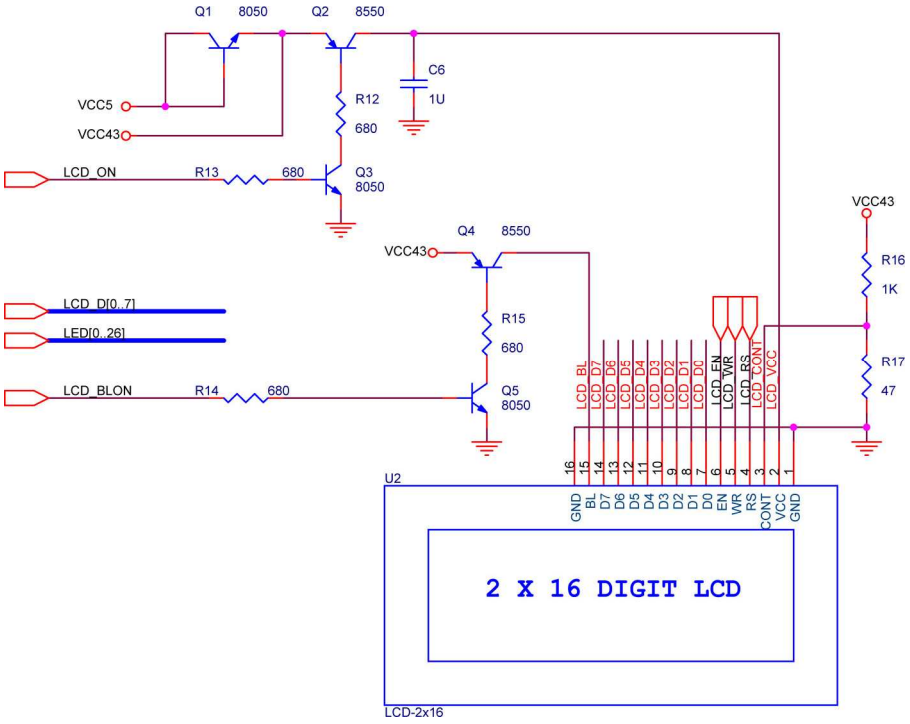


Figure 4.9. Schematic diagram of the LCD module.

Signal Name	FPGA Pin No.	Description
LCD_DATA[0]	PIN_J1	LCD Data[0]
LCD_DATA[1]	PIN_J2	LCD Data[1]
LCD_DATA[2]	PIN_H1	LCD Data[2]
LCD_DATA[3]	PIN_H2	LCD Data[3]
LCD_DATA[4]	PIN_J4	LCD Data[4]
LCD_DATA[5]	PIN_J3	LCD Data[5]
LCD_DATA[6]	PIN_H4	LCD Data[6]
LCD_DATA[7]	PIN_H3	LCD Data[7]
LCD_RW	PIN_K4	LCD Read/Write Select, 0 = Write, 1 = Read
LCD_EN	PIN_K3	LCD Enable
LCD_RS	PIN_K1	LCD Command/Data Select, 0 = Command, 1 = Data
LCD_ON	PIN_L4	LCD Power ON/OFF
LCD_BLON	PIN_K2	LCD Back Light ON/OFF

Table 4.6. Pin assignments for the LCD module.

4.6 Using the Expansion Header

The DE2 Board provides two 40-pin expansion headers. Each header connects directly to 36 pins on the Cyclone II FPGA, and also provides DC +5V (VCC5), DC +3.3V (VCC33), and two GND pins. Figure 4.10 shows the related schematics. Each pin on the expansion headers is connected to two diodes and a resistor that provide protection from high and low voltages. The figure shows the protection circuitry for only four of the pins on each header, but this circuitry is included for all 72 data pins. Table 4.7 gives the pin assignments.

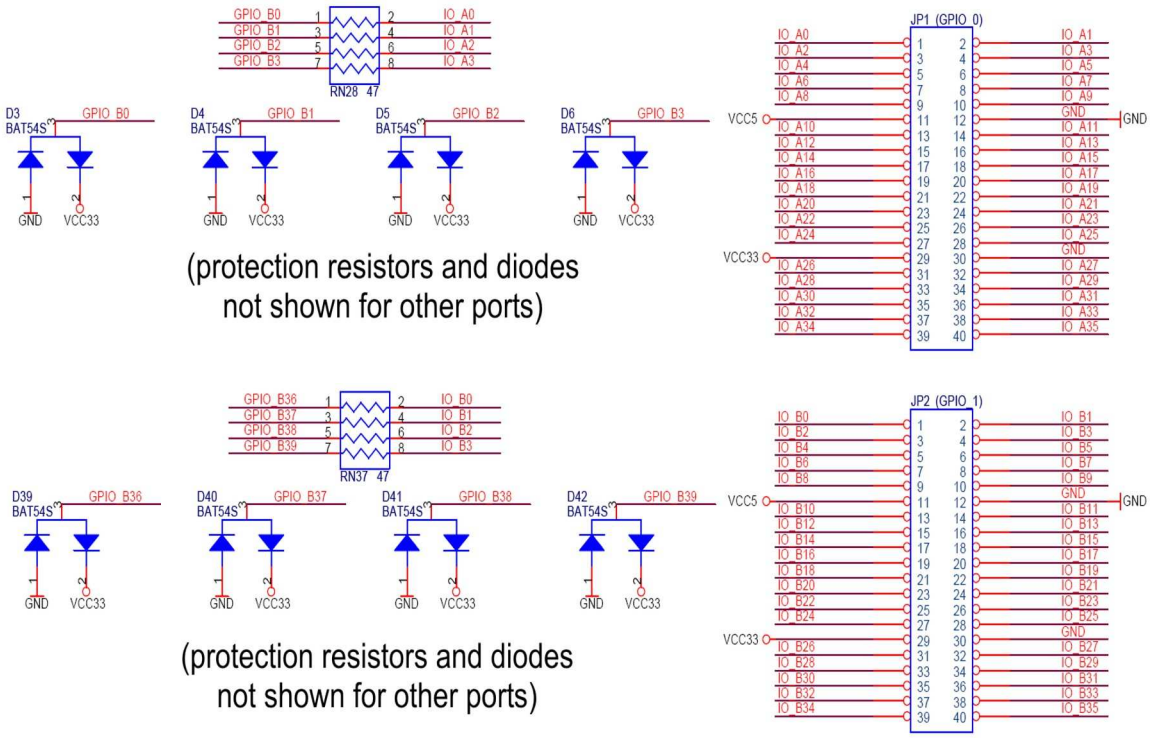


Figure 4.10. Schematic diagram of the expansion headers.

Signal Name	FPGA Pin No.	Description
GPIO_0[0]	PIN_D25	GPIO Connection 0[0]
GPIO_0[1]	PIN_J22	GPIO Connection 0[1]
GPIO_0[2]	PIN_E26	GPIO Connection 0[2]
GPIO_0[3]	PIN_E25	GPIO Connection 0[3]
GPIO_0[4]	PIN_F24	GPIO Connection 0[4]
GPIO_0[5]	PIN_F23	GPIO Connection 0[5]
GPIO_0[6]	PIN_J21	GPIO Connection 0[6]
GPIO_0[7]	PIN_J20	GPIO Connection 0[7]
GPIO_0[8]	PIN_F25	GPIO Connection 0[8]
GPIO_0[9]	PIN_F26	GPIO Connection 0[9]

GPIO_0[10]	PIN_N18	GPIO Connection 0[10]
GPIO_0[11]	PIN_P18	GPIO Connection 0[11]
GPIO_0[12]	PIN_G23	GPIO Connection 0[12]
GPIO_0[13]	PIN_G24	GPIO Connection 0[13]
GPIO_0[14]	PIN_K22	GPIO Connection 0[14]
GPIO_0[15]	PIN_G25	GPIO Connection 0[15]
GPIO_0[16]	PIN_H23	GPIO Connection 0[16]
GPIO_0[17]	PIN_H24	GPIO Connection 0[17]
GPIO_0[18]	PIN_J23	GPIO Connection 0[18]
GPIO_0[19]	PIN_J24	GPIO Connection 0[19]
GPIO_0[20]	PIN_H25	GPIO Connection 0[20]
GPIO_0[21]	PIN_H26	GPIO Connection 0[21]
GPIO_0[22]	PIN_H19	GPIO Connection 0[22]
GPIO_0[23]	PIN_K18	GPIO Connection 0[23]
GPIO_0[24]	PIN_K19	GPIO Connection 0[24]
GPIO_0[25]	PIN_K21	GPIO Connection 0[25]
GPIO_0[26]	PIN_K23	GPIO Connection 0[26]
GPIO_0[27]	PIN_K24	GPIO Connection 0[27]
GPIO_0[28]	PIN_L21	GPIO Connection 0[28]
GPIO_0[29]	PIN_L20	GPIO Connection 0[29]
GPIO_0[30]	PIN_J25	GPIO Connection 0[30]
GPIO_0[31]	PIN_J26	GPIO Connection 0[31]
GPIO_0[32]	PIN_L23	GPIO Connection 0[32]
GPIO_0[33]	PIN_L24	GPIO Connection 0[33]
GPIO_0[34]	PIN_L25	GPIO Connection 0[34]
GPIO_0[35]	PIN_L19	GPIO Connection 0[35]
GPIO_1[0]	PIN_K25	GPIO Connection 1[0]
GPIO_1[1]	PIN_K26	GPIO Connection 1[1]
GPIO_1[2]	PIN_M22	GPIO Connection 1[2]
GPIO_1[3]	PIN_M23	GPIO Connection 1[3]
GPIO_1[4]	PIN_M19	GPIO Connection 1[4]
GPIO_1[5]	PIN_M20	GPIO Connection 1[5]
GPIO_1[6]	PIN_N20	GPIO Connection 1[6]
GPIO_1[7]	PIN_M21	GPIO Connection 1[7]
GPIO_1[8]	PIN_M24	GPIO Connection 1[8]
GPIO_1[9]	PIN_M25	GPIO Connection 1[9]
GPIO_1[10]	PIN_N24	GPIO Connection 1[10]

GPIO_1[11]	PIN_P24	GPIO Connection 1[11]
GPIO_1[12]	PIN_R25	GPIO Connection 1[12]
GPIO_1[13]	PIN_R24	GPIO Connection 1[13]
GPIO_1[14]	PIN_R20	GPIO Connection 1[14]
GPIO_1[15]	PIN_T22	GPIO Connection 1[15]
GPIO_1[16]	PIN_T23	GPIO Connection 1[16]
GPIO_1[17]	PIN_T24	GPIO Connection 1[17]
GPIO_1[18]	PIN_T25	GPIO Connection 1[18]
GPIO_1[19]	PIN_T18	GPIO Connection 1[19]
GPIO_1[20]	PIN_T21	GPIO Connection 1[20]
GPIO_1[21]	PIN_T20	GPIO Connection 1[21]
GPIO_1[22]	PIN_U26	GPIO Connection 1[22]
GPIO_1[23]	PIN_U25	GPIO Connection 1[23]
GPIO_1[24]	PIN_U23	GPIO Connection 1[24]
GPIO_1[25]	PIN_U24	GPIO Connection 1[25]
GPIO_1[26]	PIN_R19	GPIO Connection 1[26]
GPIO_1[27]	PIN_T19	GPIO Connection 1[27]
GPIO_1[28]	PIN_U20	GPIO Connection 1[28]
GPIO_1[29]	PIN_U21	GPIO Connection 1[29]
GPIO_1[30]	PIN_V26	GPIO Connection 1[30]
GPIO_1[31]	PIN_V25	GPIO Connection 1[31]
GPIO_1[32]	PIN_V24	GPIO Connection 1[32]
GPIO_1[33]	PIN_V23	GPIO Connection 1[33]
GPIO_1[34]	PIN_W25	GPIO Connection 1[34]
GPIO_1[35]	PIN_W23	GPIO Connection 1[35]

Table 4.7. Pin assignments for the expansion headers.

4.7 Using VGA

The DE2 board includes a 16-pin D-SUB connector for VGA output. The VGA synchronization signals are provided directly from the Cyclone II FPGA, and the Analog Devices ADV7123 triple 10-bit high-speed video DAC is used to produce the analog data signals (red, green, and blue). The associated schematic is given in Figure 4.11 and can support resolutions of up to 1600 x 1200 pixels, at 100 MHz.

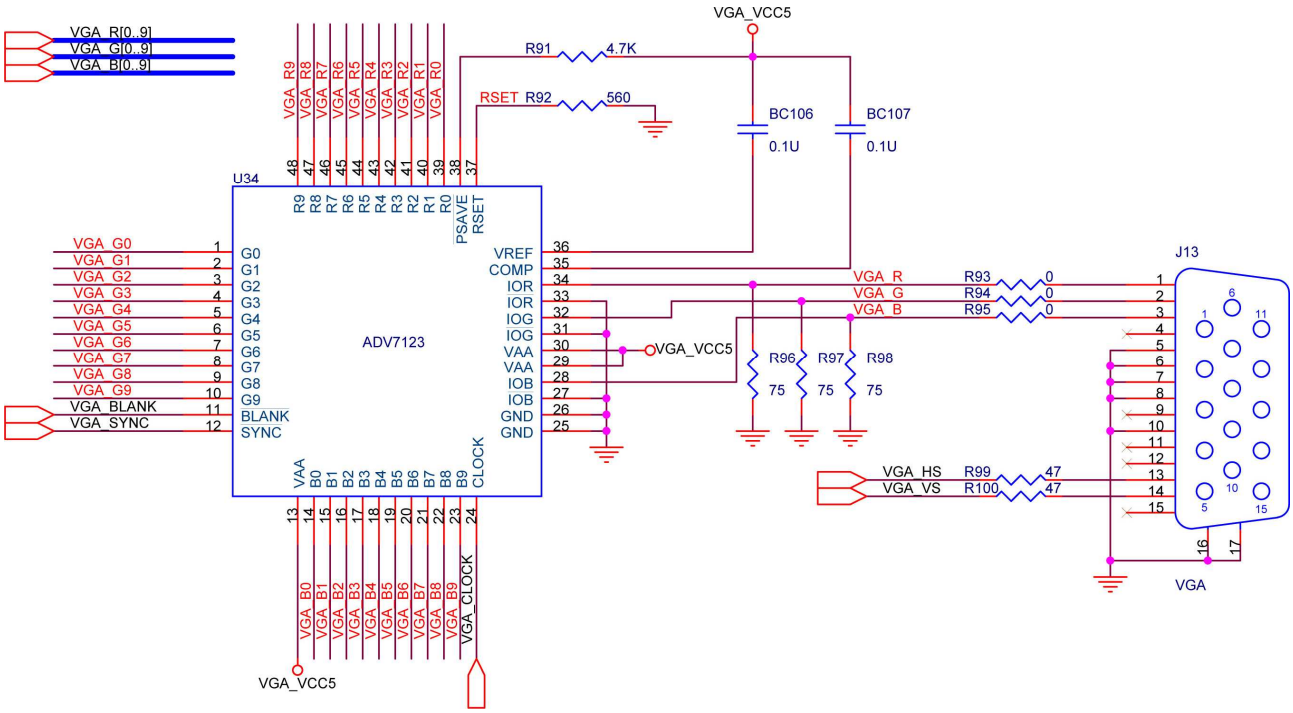


Figure 4.11. VGA circuit schematic.

The timing specification for VGA synchronization and RGB (red, green, blue) data can be found on various educational web sites (for example, search for “VGA signal timing”). Figure 4.12 illustrates the basic timing requirements for each row (horizontal) that is displayed on a VGA monitor. An active-low pulse of specific duration (time *a* in the figure) is applied to the horizontal synchronization (*hsync*) input of the monitor, which signifies the end of one row of data and the start of the next. The data (RGB) inputs on the monitor must be off (driven to 0 V) for a time period called the *back porch* (*b*) after the *hsync* pulse occurs, which is followed by the display interval (*c*). During the data display interval the RGB data drives each pixel in turn across the row being displayed. Finally, there is a time period called the *front porch* (*d*) where the RGB signals must again be off before the next *hsync* pulse can occur. The timing of the vertical synchronization (*vsync*) is the same as shown in Figure 4.12, except that a *vsync* pulse signifies the end of one frame and the start of the next, and the data refers to the set of rows in the frame (horizontal timing). Figures 4.13 and 4.14 show, for different resolutions, the durations of time periods *a*, *b*, *c*, and *d* for both horizontal and vertical timing.

Detailed information for using the ADV7123 video DAC is available in its datasheet, which can be found on the manufacturer's web site, and from the *Datasheet* folder on the **DE2 System CD-ROM**. The pin assignments between the Cyclone II FPGA and the ADV7123 are listed in Table 4.8. An example of code that drives a VGA display is described in Sections 5.2 and 5.3.

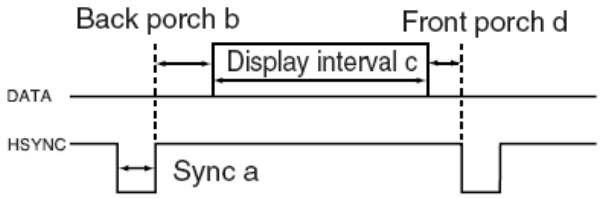


Figure 4.12. VGA horizontal timing specification.

VGA mode		Horizontal Timing Spec				
Configuration	Resolution(HxV)	a(us)	b(us)	c(us)	d(us)	Pixel clock(Mhz)
VGA(60Hz)	640x480	3.8	1.9	25.4	0.6	25 (640/c)
VGA(85Hz)	640x480	1.6	2.2	17.8	1.6	36 (640/c)
SVGA(60Hz)	800x600	3.2	2.2	20	1	40 (800/c)
SVGA(75Hz)	800x600	1.6	3.2	16.2	0.3	49 (800/c)
SVGA(85Hz)	800x600	1.1	2.7	14.2	0.6	56 (800/c)
XGA(60Hz)	1024x768	2.1	2.5	15.8	0.4	65 (1024/c)
XGA(70Hz)	1024x768	1.8	1.9	13.7	0.3	75 (1024/c)
XGA(85Hz)	1024x768	1.0	2.2	10.8	0.5	95 (1024/c)
1280x1024(60Hz)	1280x1024	1.0	2.3	11.9	0.4	108 (1280/c)

Figure 4.13. VGA horizontal timing specification.

VGA mode		Vertical Timing Spec			
Configuration	Resolution (HxV)	a(lines)	b(lines)	c(lines)	d(lines)
VGA(60Hz)	640x480	2	33	480	10
VGA(85Hz)	640x480	3	25	480	1
SVGA(60Hz)	800x600	4	23	600	1
SVGA(75Hz)	800x600	3	21	600	1
SVGA(85Hz)	800x600	3	27	600	1
XGA(60Hz)	1024x768	6	29	768	3
XGA(70Hz)	1024x768	6	29	768	3
XGA(85Hz)	1024x768	3	36	768	1
1280x1024(60Hz)	1280x1024	3	38	1024	1

Figure 4.14. VGA vertical timing specification.

Signal Name	FPGA Pin No.	Description
VGA_R[0]	PIN_C8	VGA Red[0]
VGA_R[1]	PIN_F10	VGA Red[1]
VGA_R[2]	PIN_G10	VGA Red[2]
VGA_R[3]	PIN_D9	VGA Red[3]
VGA_R[4]	PIN_C9	VGA Red[4]
VGA_R[5]	PIN_A8	VGA Red[5]
VGA_R[6]	PIN_H11	VGA Red[6]
VGA_R[7]	PIN_H12	VGA Red[7]
VGA_R[8]	PIN_F11	VGA Red[8]
VGA_R[9]	PIN_E10	VGA Red[9]
VGA_G[0]	PIN_B9	VGA Green[0]
VGA_G[1]	PIN_A9	VGA Green[1]
VGA_G[2]	PIN_C10	VGA Green[2]
VGA_G[3]	PIN_D10	VGA Green[3]
VGA_G[4]	PIN_B10	VGA Green[4]
VGA_G[5]	PIN_A10	VGA Green[5]
VGA_G[6]	PIN_G11	VGA Green[6]
VGA_G[7]	PIN_D11	VGA Green[7]
VGA_G[8]	PIN_E12	VGA Green[8]
VGA_G[9]	PIN_D12	VGA Green[9]
VGA_B[0]	PIN_J13	VGA Blue[0]
VGA_B[1]	PIN_J14	VGA Blue[1]
VGA_B[2]	PIN_F12	VGA Blue[2]
VGA_B[3]	PIN_G12	VGA Blue[3]
VGA_B[4]	PIN_J10	VGA Blue[4]
VGA_B[5]	PIN_J11	VGA Blue[5]
VGA_B[6]	PIN_C11	VGA Blue[6]
VGA_B[7]	PIN_B11	VGA Blue[7]
VGA_B[8]	PIN_C12	VGA Blue[8]
VGA_B[9]	PIN_B12	VGA Blue[9]
VGA_CLK	PIN_B8	VGA Clock
VGA_BLANK	PIN_D6	VGA BLANK
VGA_HS	PIN_A7	VGA H_SYNC
VGA_VS	PIN_D8	VGA V_SYNC
VGA_SYNC	PIN_B7	VGA SYNC

Table 4.8. ADV7123 pin assignments.

4.8 Using the 24-bit Audio CODEC

The DE2 board provides high-quality 24-bit audio via the Wolfson WM8731 audio CODEC (enCOder/DECoder). This chip supports microphone-in, line-in, and line-out ports, with a sample rate adjustable from 8 kHz to 96 kHz. The WM8731 is controlled by a serial I2C bus interface, which is connected to pins on the Cyclone II FPGA. A schematic diagram of the audio circuitry is shown in Figure 4.15, and the FPGA pin assignments are listed in Table 4.9. Detailed information for using the WM8731 codec is available in its datasheet, which can be found on the manufacturer's web site, and from the *Datasheet* folder on the **DE2 System CD-ROM**.

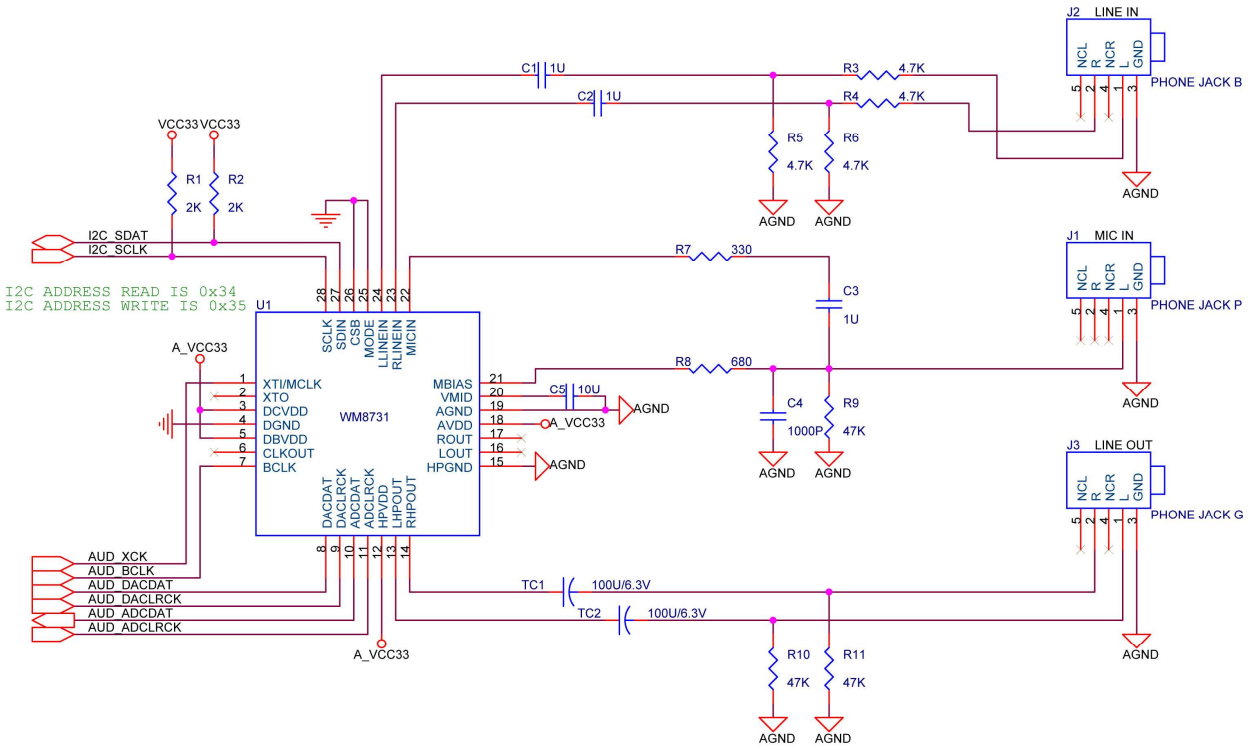


Figure 4.15. Audio CODEC schematic.

Signal Name	FPGA Pin No.	Description
AUD_ADCLRCK	PIN_C5	Audio CODEC ADC LR Clock
AUD_ADCDAT	PIN_B5	Audio CODEC ADC Data
AUD_DACLRCK	PIN_C6	Audio CODEC DAC LR Clock
AUD_DACDAT	PIN_A4	Audio CODEC DAC Data
AUD_XCK	PIN_A5	Audio CODEC Chip Clock
AUD_BCLK	PIN_B4	Audio CODEC Bit-Stream Clock
I2C_SCLK	PIN_A6	I2C Data
I2C_SDAT	PIN_B6	I2C Clock

Table 4.9. Audio CODEC pin assignments.

4.9 RS-232 Serial Port

The DE2 board uses the MAX232 transceiver chip and a 9-pin D-SUB connector for RS-232 communications. For detailed information on how to use the transceiver refer to the datasheet, which is available on the manufacturer's web site, and from the *Datasheet* folder on the **DE2 System CD-ROM**. Figure 4.16 shows the related schematics, and Table 4.10 lists the Cyclone II FPGA pin assignments.

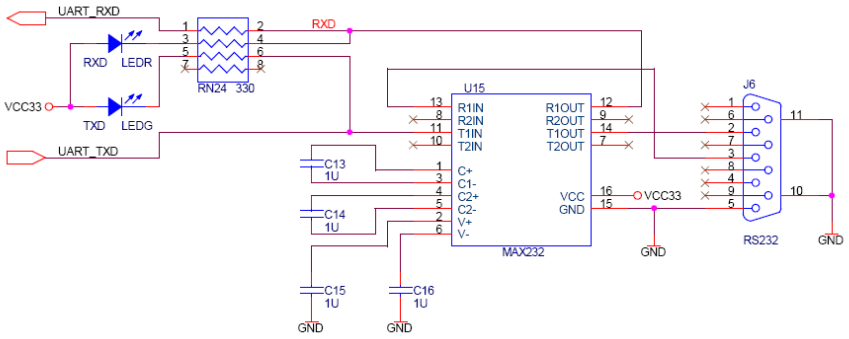


Figure 4.16. MAX232 (RS-232) chip schematic.

Signal Name	FPGA Pin No.	Description
UART_RXD	PIN_C25	UART Receiver
UART_TXD	PIN_B25	UART Transmitter

Table 4.10. RS-232 pin assignments.

4.10 PS/2 Serial Port

The DE2 board includes a standard PS/2 interface and a connector for a PS/2 keyboard or mouse. Figure 4.17 shows the schematic of the PS/2 circuit. Instructions for using a PS/2 mouse or keyboard can be found by performing an appropriate search on various educational web sites. The pin assignments for the associated interface are shown in Table 4.11.

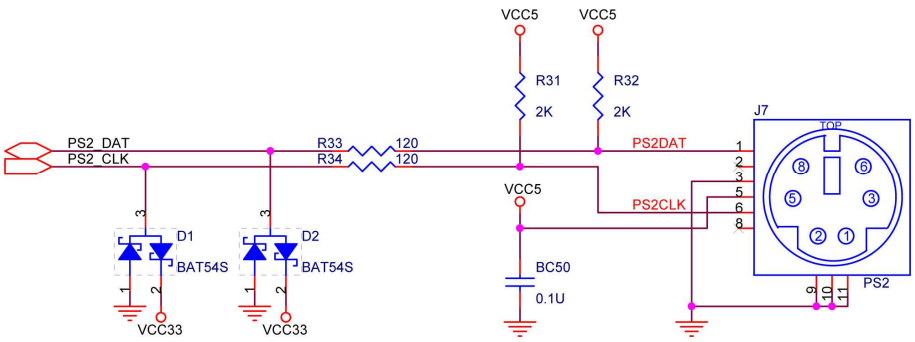


Figure 4.17. PS/2 schematic.

Signal Name	FPGA Pin No.	Description
PS2_CLK	PIN_D26	PS/2 Clock
PS2_DAT	PIN_C24	PS/2 Data

Table 4.11. PS/2 pin assignments.

4.11 Fast Ethernet Network Controller

The DE2 board provides Ethernet support via the Davicom DM9000A Fast Ethernet controller chip. The DM9000A includes a general processor interface, 16 Kbytes SRAM, a media access control (MAC) unit, and a 10/100M PHY transceiver. Figure 4.18 shows the schematic for the Fast Ethernet interface, and the associated pin assignments are listed in Table 4.12. For detailed information on how to use the DM9000A refer to its datasheet and application note, which are available on the manufacturer's web site, and from the *Datasheet* folder on the **DE2 System CD-ROM**.

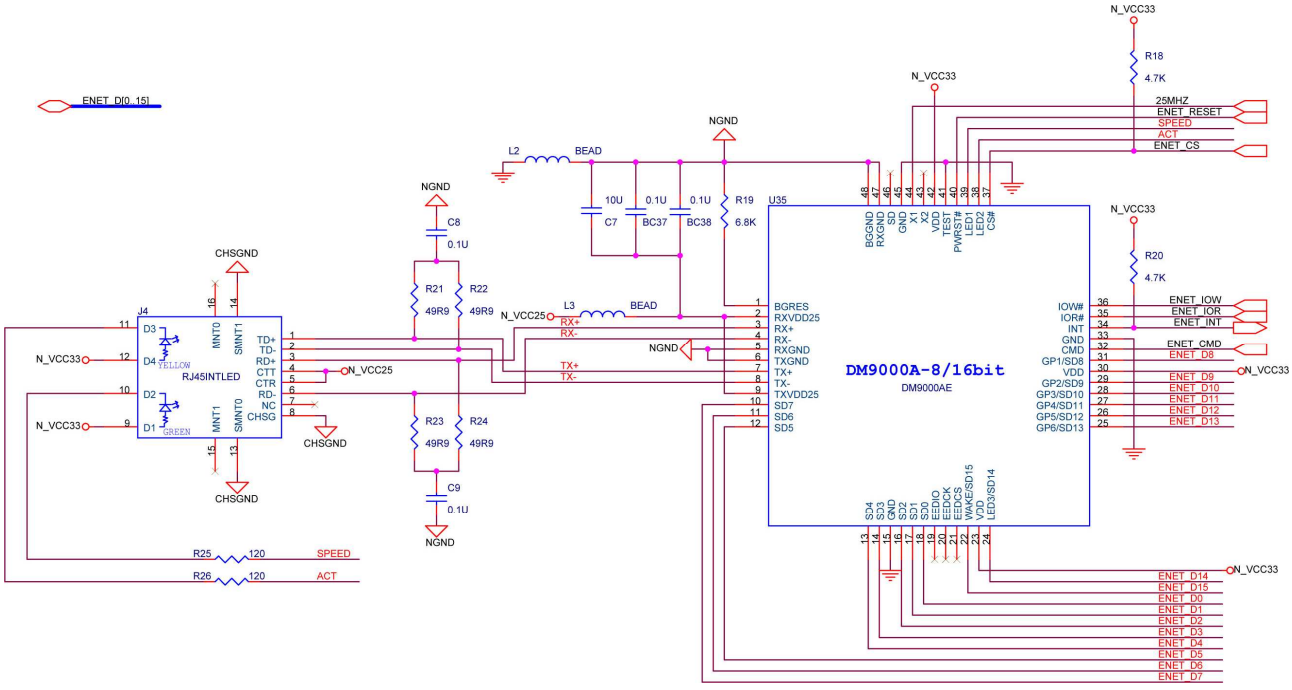


Figure 4.18. Fast Ethernet schematic.

Signal Name	FPGA Pin No.	Description
ENET_DATA[0]	PIN_D17	DM9000A DATA[0]
ENET_DATA[1]	PIN_C17	DM9000A DATA[1]
ENET_DATA[2]	PIN_B18	DM9000A DATA[2]
ENET_DATA[3]	PIN_A18	DM9000A DATA[3]
ENET_DATA[4]	PIN_B17	DM9000A DATA[4]

ENET_DATA[5]	PIN_A17	DM9000A DATA[5]
ENET_DATA[6]	PIN_B16	DM9000A DATA[6]
ENET_DATA[7]	PIN_B15	DM9000A DATA[7]
ENET_DATA[8]	PIN_B20	DM9000A DATA[8]
ENET_DATA[9]	PIN_A20	DM9000A DATA[9]
ENET_DATA[10]	PIN_C19	DM9000A DATA[10]
ENET_DATA[11]	PIN_D19	DM9000A DATA[11]
ENET_DATA[12]	PIN_B19	DM9000A DATA[12]
ENET_DATA[13]	PIN_A19	DM9000A DATA[13]
ENET_DATA[14]	PIN_E18	DM9000A DATA[14]
ENET_DATA[15]	PIN_D18	DM9000A DATA[15]
ENET_CLK	PIN_B24	DM9000A Clock 25 MHz
ENET_CMD	PIN_A21	DM9000A Command/Data Select, 0 = Command, 1 = Data
ENET_CS_N	PIN_A23	DM9000A Chip Select
ENET_INT	PIN_B21	DM9000A Interrupt
ENET_RD_N	PIN_A22	DM9000A Read
ENET_WR_N	PIN_B22	DM9000A Write
ENET_RST_N	PIN_B23	DM9000A Reset

Table 4.12. Fast Ethernet pin assignments.

4.12 TV Decoder

The DE2 board is equipped with an Analog Devices ADV7180 TV decoder chip. The ADV7180 is an integrated video decoder that automatically detects and converts a standard analog baseband television signal (NTSC, PAL, and SECAM) into 4:2:2 component video data compatible with 16-bit/8-bit CCIR601/CCIR656. The ADV7180 is compatible with a broad range of video devices, including DVD players, tape-based sources, broadcast sources, and security/surveillance cameras.

The registers in the TV decoder can be programmed by a serial I2C bus, which is connected to the Cyclone II FPGA as indicated in Figure 4.19. The pin assignments are listed in Table 4.13. Detailed information on the ADV7180 is available on the manufacturer's web site, and from the *Datasheet* folder on the **DE2 System CD-ROM**.

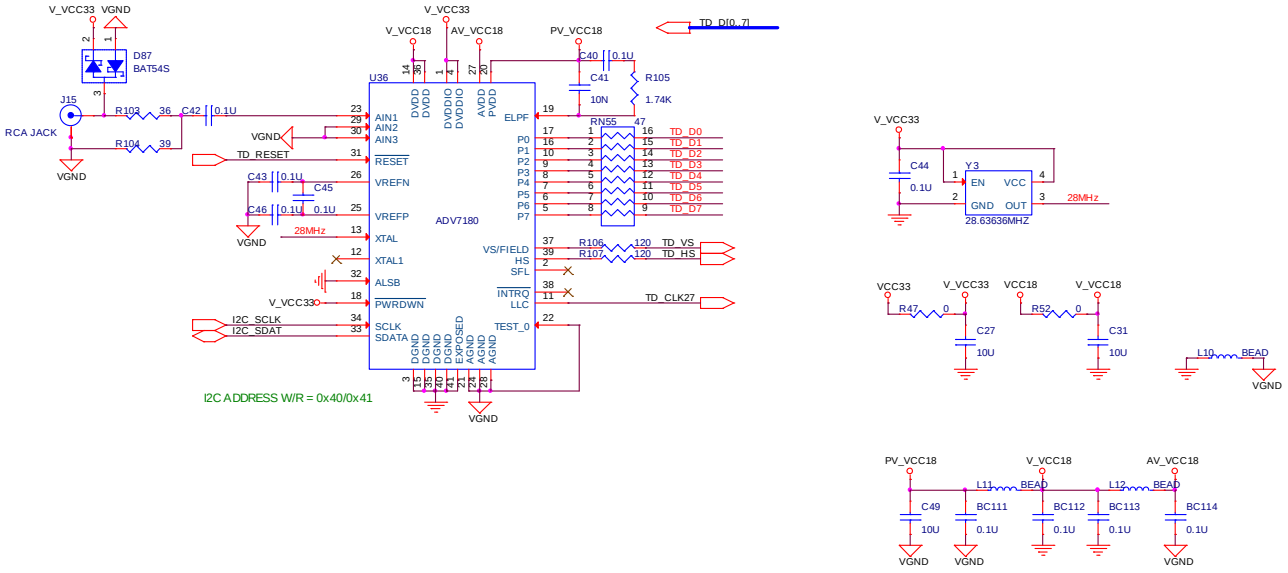


Figure 4.19. TV Decoder schematic.

Signal Name	FPGA Pin No.	Description
TD_DATA[0]	PIN_J9	TV Decoder Data[0]
TD_DATA[1]	PIN_E8	TV Decoder Data[1]
TD_DATA[2]	PIN_H8	TV Decoder Data[2]
TD_DATA[3]	PIN_H10	TV Decoder Data[3]
TD_DATA[4]	PIN_G9	TV Decoder Data[4]
TD_DATA[5]	PIN_F9	TV Decoder Data[5]
TD_DATA[6]	PIN_D7	TV Decoder Data[6]
TD_DATA[7]	PIN_C7	TV Decoder Data[7]
TD_HS	PIN_D5	TV Decoder H_SYNC
TD_VS	PIN_K9	TV Decoder V_SYNC
TD_CLK27	PIN_C16	TV Decoder Clock Input.
TD_RESET	PIN_C4	TV Decoder Reset
I2C_SCLK	PIN_A6	I2C Data
I2C_SDAT	PIN_B6	I2C Clock

Table 4.13. TV Decoder pin assignments.

4.13 Implementing a TV Encoder

Although the DE2 board does not include a TV encoder chip, the ADV7123 (10-bit high-speed triple ADCs) can be used to implement a professional-quality TV encoder with the digital processing part implemented in the Cyclone II FPGA. Figure 4.20 shows a block diagram of a TV encoder implemented in this manner.

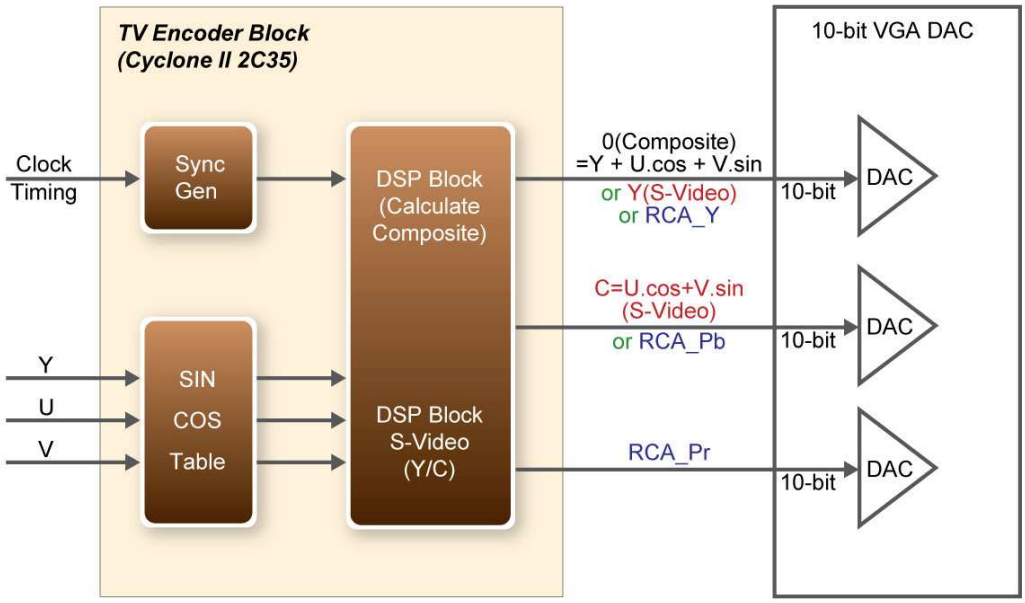


Figure 4.20. A TV Encoder that uses the Cyclone II FPGA and the ADV7123.

4.14 Using USB Host and Device

The DE2 board provides both USB host and device interfaces using the Philips ISP1362 single-chip USB controller. The host and device controllers are compliant with the Universal Serial Bus Specification Rev. 2.0, supporting data transfer at full-speed (12 Mbit/s) and low-speed (1.5 Mbit/s). Figure 4.21 shows the schematic diagram of the USB circuitry; the pin assignments for the associated interface are listed in Table 4.14.

Detailed information for using the ISP1362 device is available in its datasheet and programming guide; both documents can be found on the manufacturer's web site, and from the *Datasheet* folder on the **DE2 System CD-ROM**. The most challenging part of a USB application is in the design of the software driver needed. Two complete examples of USB drivers, for both host and device applications, can be found in Sections 5.3 and 5.4. These demonstrations provide examples of software drivers for the Nios II processor.

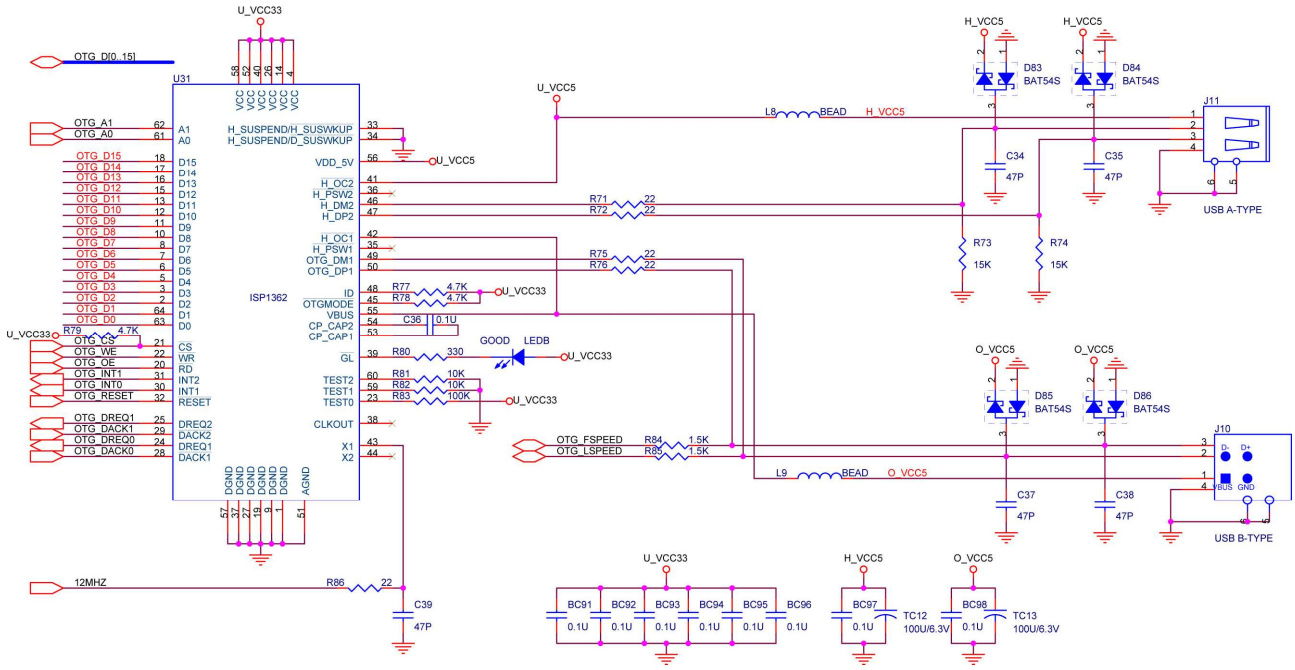


Figure 4.21. USB (ISP1362) host and device schematic.

Signal Name	FPGA Pin No.	Description
OTG_ADDR[0]	PIN_K7	ISP1362 Address[0]
OTG_ADDR[1]	PIN_F2	ISP1362 Address[1]
OTG_DATA[0]	PIN_F4	ISP1362 Data[0]
OTG_DATA[1]	PIN_D2	ISP1362 Data[1]
OTG_DATA[2]	PIN_D1	ISP1362 Data[2]
OTG_DATA[3]	PIN_F7	ISP1362 Data[3]
OTG_DATA[4]	PIN_J5	ISP1362 Data[4]
OTG_DATA[5]	PIN_J8	ISP1362 Data[5]
OTG_DATA[6]	PIN_J7	ISP1362 Data[6]
OTG_DATA[7]	PIN_H6	ISP1362 Data[7]
OTG_DATA[8]	PIN_E2	ISP1362 Data[8]
OTG_DATA[9]	PIN_E1	ISP1362 Data[9]
OTG_DATA[10]	PIN_K6	ISP1362 Data[10]
OTG_DATA[11]	PIN_K5	ISP1362 Data[11]
OTG_DATA[12]	PIN_G4	ISP1362 Data[12]
OTG_DATA[13]	PIN_G3	ISP1362 Data[13]
OTG_DATA[14]	PIN_J6	ISP1362 Data[14]
OTG_DATA[15]	PIN_K8	ISP1362 Data[15]
OTG_CS_N	PIN_F1	ISP1362 Chip Select
OTG_RD_N	PIN_G2	ISP1362 Read

OTG_WR_N	PIN_G1	ISP1362 Write
OTG_RST_N	PIN_G5	ISP1362 Reset
OTG_INT0	PIN_B3	ISP1362 Interrupt 0
OTG_INT1	PIN_C3	ISP1362 Interrupt 1
OTG_DACK0_N	PIN_C2	ISP1362 DMA Acknowledge 0
OTG_DACK1_N	PIN_B2	ISP1362 DMA Acknowledge 1
OTG_DREQ0	PIN_F6	ISP1362 DMA Request 0
OTG_DREQ1	PIN_E5	ISP1362 DMA Request 1
OTG_FSPEED	PIN_F3	USB Full Speed, 0 = Enable, Z = Disable
OTG_LSPEED	PIN_G6	USB Low Speed, 0 = Enable, Z = Disable

Table 4.14. USB (ISP1362) pin assignments.

4.15 Using IrDA

The DE2 board provides a simple wireless communication media using the Agilent HSDL-3201 low power infrared transceiver. The datasheet for this device is provided in the *Datasheet\IrDA* folder on the **DE2 System CD-ROM**. Note that the highest transmission rate supported is 115.2 Kbit/s and both the TX and RX sides have to use the same transmission rate. Figure 4.22 shows the schematic of the IrDA communication link. Please refer to the following website for detailed information on how to send and receive data using the IrDA link: http://techtrain.microchip.com/webseminars/documents/IrDA_BW.pdf.
The pin assignment of the associated interface are listed in Table 4.15.

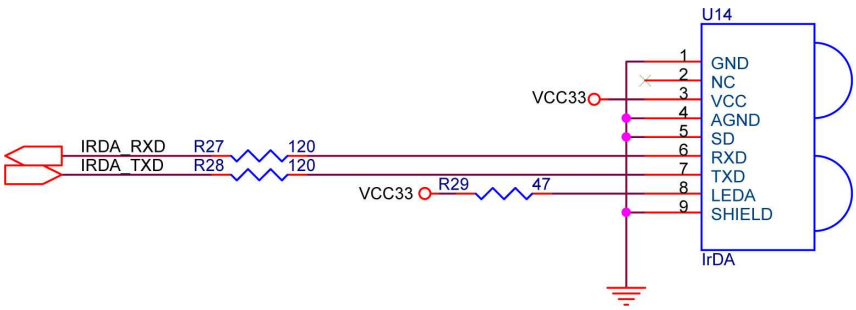


Figure 4.22. IrDA schematic.

Signal Name	FPGA Pin No.	Description
IRDA_TXD	PIN_AE24	IRDA Transmitter
IRDA_RXD	PIN_AE25	IRDA Receiver

Table 4.15. IrDA pin assignments.

4.16 Using SDRAM/SRAM/Flash

The DE2 board provides an 8-Mbyte SDRAM, 512-Kbyte SRAM, and 4-Mbyte (1-Mbyte on some boards) Flash memory. Figures 4.23, 4.24, and 4.25 show the schematics of the memory chips. The pin assignments for each device are listed in Tables 4.16, 4.17, and 4.18. The datasheets for the memory chips are provided in the *Datasheet* folder on the **DE2 System CD-ROM**.

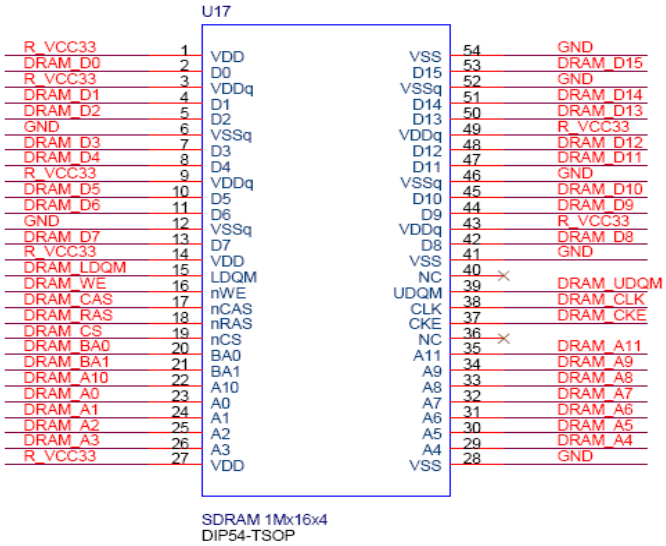


Figure 4.23. SDRAM schematic.

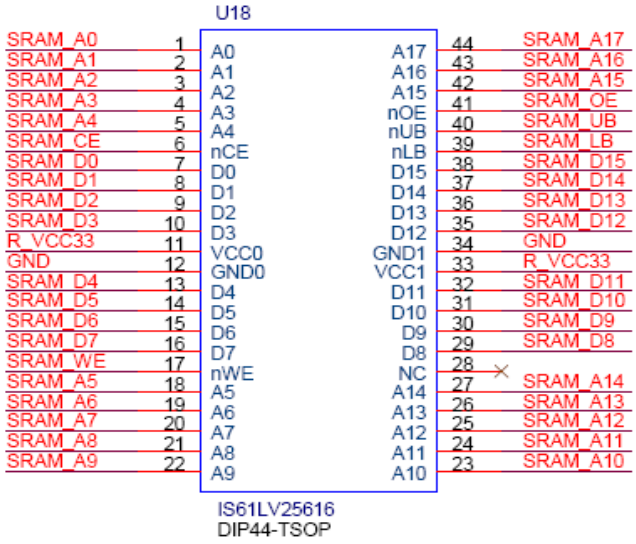


Figure 4.24. SRAM schematic.

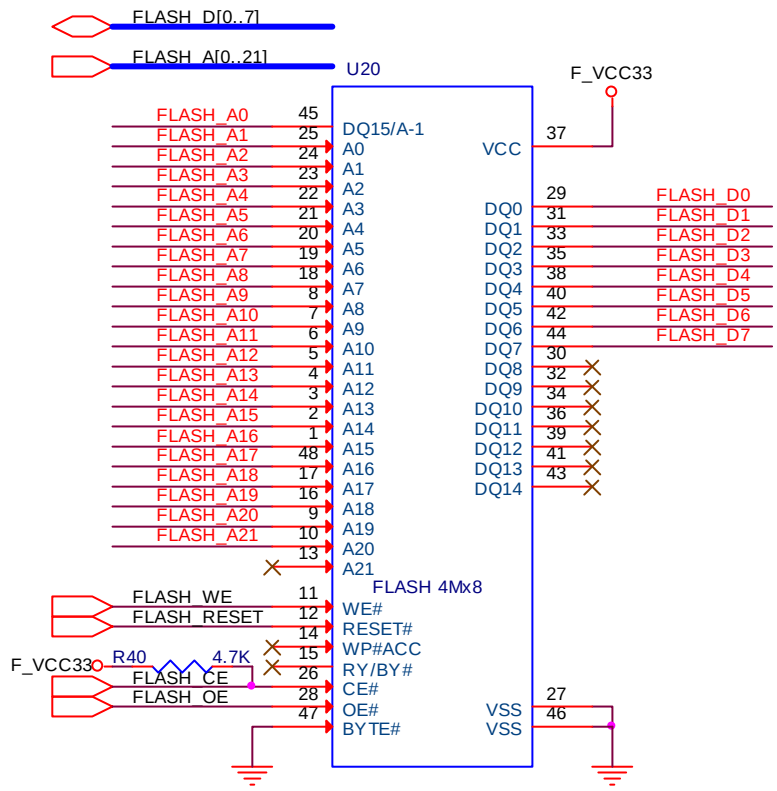


Figure 4.25. Flash schematic.

Signal Name	FPGA Pin No.	Description
DRAM_ADDR[0]	PIN_T6	SDRAM Address[0]
DRAM_ADDR[1]	PIN_V4	SDRAM Address[1]
DRAM_ADDR[2]	PIN_V3	SDRAM Address[2]
DRAM_ADDR[3]	PIN_W2	SDRAM Address[3]
DRAM_ADDR[4]	PIN_W1	SDRAM Address[4]
DRAM_ADDR[5]	PIN_U6	SDRAM Address[5]
DRAM_ADDR[6]	PIN_U7	SDRAM Address[6]
DRAM_ADDR[7]	PIN_U5	SDRAM Address[7]
DRAM_ADDR[8]	PIN_W4	SDRAM Address[8]
DRAM_ADDR[9]	PIN_W3	SDRAM Address[9]
DRAM_ADDR[10]	PIN_Y1	SDRAM Address[10]
DRAM_ADDR[11]	PIN_V5	SDRAM Address[11]
DRAM_DQ[0]	PIN_V6	SDRAM Data[0]
DRAM_DQ[1]	PIN_AA2	SDRAM Data[1]
DRAM_DQ[2]	PIN_AA1	SDRAM Data[2]
DRAM_DQ[3]	PIN_Y3	SDRAM Data[3]
DRAM_DQ[4]	PIN_Y4	SDRAM Data[4]

DRAM_DQ[5]	PIN_R8	SDRAM Data[5]
DRAM_DQ[6]	PIN_T8	SDRAM Data[6]
DRAM_DQ[7]	PIN_V7	SDRAM Data[7]
DRAM_DQ[8]	PIN_W6	SDRAM Data[8]
DRAM_DQ[9]	PIN_AB2	SDRAM Data[9]
DRAM_DQ[10]	PIN_AB1	SDRAM Data[10]
DRAM_DQ[11]	PIN_AA4	SDRAM Data[11]
DRAM_DQ[12]	PIN_AA3	SDRAM Data[12]
DRAM_DQ[13]	PIN_AC2	SDRAM Data[13]
DRAM_DQ[14]	PIN_AC1	SDRAM Data[14]
DRAM_DQ[15]	PIN_AA5	SDRAM Data[15]
DRAM_BA_0	PIN_AE2	SDRAM Bank Address[0]
DRAM_BA_1	PIN_AE3	SDRAM Bank Address[1]
DRAM_LDQM	PIN_AD2	SDRAM Low-byte Data Mask
DRAM_UDQM	PIN_Y5	SDRAM High-byte Data Mask
DRAM_RAS_N	PIN_AB4	SDRAM Row Address Strobe
DRAM_CAS_N	PIN_AB3	SDRAM Column Address Strobe
DRAM_CKE	PIN_AA6	SDRAM Clock Enable
DRAM_CLK	PIN_AA7	SDRAM Clock
DRAM_WE_N	PIN_AD3	SDRAM Write Enable
DRAM_CS_N	PIN_AC3	SDRAM Chip Select

Table 4.16. SDRAM pin assignments.

Signal Name	FPGA Pin No.	Description
SRAM_ADDR[0]	PIN_AE4	SRAM Address[0]
SRAM_ADDR[1]	PIN_AF4	SRAM Address[1]
SRAM_ADDR[2]	PIN_AC5	SRAM Address[2]
SRAM_ADDR[3]	PIN_AC6	SRAM Address[3]
SRAM_ADDR[4]	PIN_AD4	SRAM Address[4]
SRAM_ADDR[5]	PIN_AD5	SRAM Address[5]
SRAM_ADDR[6]	PIN_AE5	SRAM Address[6]
SRAM_ADDR[7]	PIN_AF5	SRAM Address[7]
SRAM_ADDR[8]	PIN_AD6	SRAM Address[8]
SRAM_ADDR[9]	PIN_AD7	SRAM Address[9]
SRAM_ADDR[10]	PIN_V10	SRAM Address[10]
SRAM_ADDR[11]	PIN_V9	SRAM Address[11]

SRAM_ADDR[12]	PIN_AC7	SRAM Address[12]
SRAM_ADDR[13]	PIN_W8	SRAM Address[13]
SRAM_ADDR[14]	PIN_W10	SRAM Address[14]
SRAM_ADDR[15]	PIN_Y10	SRAM Address[15]
SRAM_ADDR[16]	PIN_AB8	SRAM Address[16]
SRAM_ADDR[17]	PIN_AC8	SRAM Address[17]
SRAM_DQ[0]	PIN_AD8	SRAM Data[0]
SRAM_DQ[1]	PIN_AE6	SRAM Data[1]
SRAM_DQ[2]	PIN_AF6	SRAM Data[2]
SRAM_DQ[3]	PIN_AA9	SRAM Data[3]
SRAM_DQ[4]	PIN_AA10	SRAM Data[4]
SRAM_DQ[5]	PIN_AB10	SRAM Data[5]
SRAM_DQ[6]	PIN_AA11	SRAM Data[6]
SRAM_DQ[7]	PIN_Y11	SRAM Data[7]
SRAM_DQ[8]	PIN_AE7	SRAM Data[8]
SRAM_DQ[9]	PIN_AF7	SRAM Data[9]
SRAM_DQ[10]	PIN_AE8	SRAM Data[10]
SRAM_DQ[11]	PIN_AF8	SRAM Data[11]
SRAM_DQ[12]	PIN_W11	SRAM Data[12]
SRAM_DQ[13]	PIN_W12	SRAM Data[13]
SRAM_DQ[14]	PIN_AC9	SRAM Data[14]
SRAM_DQ[15]	PIN_AC10	SRAM Data[15]
SRAM_WE_N	PIN_AE10	SRAM Write Enable
SRAM_OE_N	PIN_AD10	SRAM Output Enable
SRAM_UB_N	PIN_AF9	SRAM High-byte Data Mask
SRAM_LB_N	PIN_AE9	SRAM Low-byte Data Mask
SRAM_CE_N	PIN_AC11	SRAM Chip Enable

Table 4.17. SRAM pin assignments.

Signal Name	FPGA Pin No.	Description
FL_ADDR[0]	PIN_AC18	FLASH Address[0]
FL_ADDR[1]	PIN_AB18	FLASH Address[1]
FL_ADDR[2]	PIN_AE19	FLASH Address[2]
FL_ADDR[3]	PIN_AF19	FLASH Address[3]
FL_ADDR[4]	PIN_AE18	FLASH Address[4]
FL_ADDR[5]	PIN_AF18	FLASH Address[5]

FL_ADDR[6]	PIN_Y16	FLASH Address[6]
FL_ADDR[7]	PIN_AA16	FLASH Address[7]
FL_ADDR[8]	PIN_AD17	FLASH Address[8]
FL_ADDR[9]	PIN_AC17	FLASH Address[9]
FL_ADDR[10]	PIN_AE17	FLASH Address[10]
FL_ADDR[11]	PIN_AF17	FLASH Address[11]
FL_ADDR[12]	PIN_W16	FLASH Address[12]
FL_ADDR[13]	PIN_W15	FLASH Address[13]
FL_ADDR[14]	PIN_AC16	FLASH Address[14]
FL_ADDR[15]	PIN_AD16	FLASH Address[15]
FL_ADDR[16]	PIN_AE16	FLASH Address[16]
FL_ADDR[17]	PIN_AC15	FLASH Address[17]
FL_ADDR[18]	PIN_AB15	FLASH Address[18]
FL_ADDR[19]	PIN_AA15	FLASH Address[19]
FL_ADDR[20]	PIN_Y15	FLASH Address[20]
FL_ADDR[21]	PIN_Y14	FLASH Address[21]
FL_DQ[0]	PIN_AD19	FLASH Data[0]
FL_DQ[1]	PIN_AC19	FLASH Data[1]
FL_DQ[2]	PIN_AF20	FLASH Data[2]
FL_DQ[3]	PIN_AE20	FLASH Data[3]
FL_DQ[4]	PIN_AB20	FLASH Data[4]
FL_DQ[5]	PIN_AC20	FLASH Data[5]
FL_DQ[6]	PIN_AF21	FLASH Data[6]
FL_DQ[7]	PIN_AE21	FLASH Data[7]
FL_CE_N	PIN_V17	FLASH Chip Enable
FL_OE_N	PIN_W17	FLASH Output Enable
FL_RST_N	PIN_AA18	FLASH Reset
FL_WE_N	PIN_AA17	FLASH Write Enable

Table 4.18. Flash pin assignments.

Appendix D : BraRV32 Optimized Multicycle Code Documentation (with dynamic links)

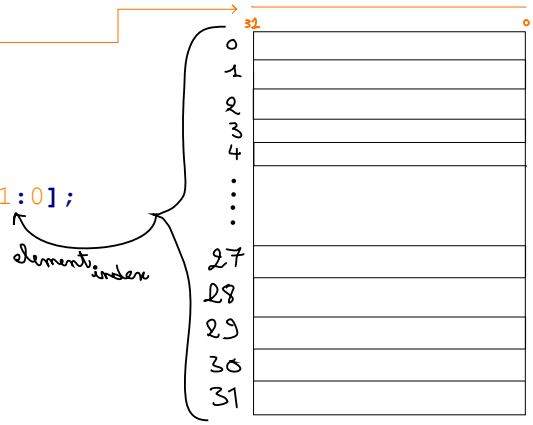
```
/***************************************************************
***/
// Based on FemtoRV32, a collection of minimalistic RISC-V RV32 cores.

//             A  single VERILOG file, compact & understandable
code.
//             (200 lines of code, 400 lines counting comments)
//
// Instruction set: RV32I + RDCYCLES
//
// Parameters:
//  Reset address can be defined using RESET_ADDR (default is 0).
//
//  The ADDR_WIDTH parameter lets you define the width of the
internal
//  address bus (and address computation logic).
//
// Macros:
//    optionally one may define NRV_IS_IO_ADDR(addr), that is
supposed to:
//              evaluate to 1 if addr is in mapped IO space,
//              evaluate to 0 otherwise
//    (additional wait states are used when in IO space).
//    If left undefined, wait states are always used.
//
//    NRV_COUNTER_WIDTH may be defined to reduce the number of
bits used
//    by the ticks counter. If not defined, a 32-bits counter is
generated.
//    (reducing its width may be useful for space-constrained
designs).
//
//    NRV_TWOLEVEL_SHIFTER may be defined to make shift
operations faster
//    (uses a two-level shifter inspired by picorv32).
// Brahim OUELD EL HAIRECH 2024-2025 , inspired from  Bruno Levy, Matthias
// Koch,implementation
/***************************************************************
***/

// Firmware generation flags for this processor
`define NRV_ARCH     "rv32i"
`define NRV_ABI      "ilp32"
`define NRV_OPTIMIZE "-Os"
```

```
module FemtoRV32(
   input      clk,

   output [31:0] mem_addr,  // address bus
   output [31:0] mem_wdata, // data to be written
   output [3:0]  mem_wmask, // write mask for the 4 bytes of each
word
   input [31:0]  mem_rdata, // input lines for both data and
instr
   output     mem_rstrb, // active to initiate memory read (used
by IO)
   input      mem_rbusy, // asserted if memory is busy reading
value
   input      mem_wbusy, // asserted if memory is busy writing
value

   input      reset      // set to 0 to reset the processor
);

   parameter RESET_ADDR       = 32'h00000000;
   parameter ADDR_WIDTH       = 24;


/***************************************************************
***********/
 // Instruction decoding.

/***************************************************************
***********/

 // Extracts rd,rs1,rs2,funct3,imm and opcode from instruction.
 // Reference: Table page 104 of:
 // https://content.riscv.org/wp-content/uploads/2017/05/riscv-
spec-v2.2.pdf

 // The destination register
 wire [4:0] rdId = instr[11:7];

 // The ALU function, decoded in 1-hot form (doing so reduces LUT
count)
 // It is used as follows: funct3Is[val] <=> funct3 == val
 (* onehot *)
 wire [7:0] funct3Is = 8'b00000001 << instr[14:12];

 // The five immediate formats, see RiscV reference (link above),
Fig. 2.4 p. 12
```

```
 wire [31:0] Uimm = {    instr[31],   instr[30:12], {12{1'b0}}};
 wire [31:0] Iimm = {{21{instr[31]}}, instr[30:20]};
 /* verilator lint_off UNUSED */ // MSBs of SBJimms are not used
by addr adder.
 wire [31:0] Simm = {{21{instr[31]}}, instr[30:25],instr[11:7]};
 wire [31:0] Bimm = {{20{instr[31]}},
instr[7],instr[30:25],instr[11:8],1'b0};
 wire [31:0] Jimm = {{12{instr[31]}},
instr[19:12],instr[20],instr[30:21],1'b0};
 /* verilator lint_on UNUSED */

   // Base RISC-V (RV32I) has only 10 different instructions !
   wire isLoad    =  (instr[6:2] == 5'b00000); // rd <-
mem[rs1+Iimm]
   wire isALUimm  =  (instr[6:2] == 5'b00100); // rd <- rs1 OP
Iimm
   wire isStore   =  (instr[6:2] == 5'b01000); // mem[rs1+Simm]
<- rs2
   wire isALUreg  =  (instr[6:2] == 5'b01100); // rd <- rs1 OP
rs2
   wire isSYSTEM  =  (instr[6:2] == 5'b11100); // rd <- cycles
   wire isJAL     =  instr[3]; // (instr[6:2] == 5'b11011); // rd
<- PC+4; PC<-PC+Jimm
   wire isJALR    =  (instr[6:2] == 5'b11001); // rd <- PC+4;
PC<-rs1+Iimm
   wire isLUI     =  (instr[6:2] == 5'b01101); // rd <- Uimm
   wire isAUIPC   =  (instr[6:2] == 5'b00101); // rd <- PC + Uimm
   wire isBranch  =  (instr[6:2] == 5'b11000); // if(rs1 OP rs2)
PC<-PC+Bimm

 wire isALU = isALUimm | isALUreg;


/*************************************************************
***********/
   // The register file.

/*************************************************************
***********/

   reg [31:0] rs1;
   reg [31:0] rs2;

   (* no_rw_check *)
   reg [31:0] registerFile [31:0];
```

Size

element index



```
   always @(posedge clk) begin
     if (writeBack)
       if (rdId != 0)
         registerFile[rdId] <= writeBackData;
   end


/***************************************************************
***********/
   // The ALU. Does operations and tests combinatorially, except
shifts.

/***************************************************************
***********/

   // First ALU source, always rs1
   wire [31:0] aluIn1 = rs1;

   // Second ALU source, depends on opcode:
   //    ALUreg, Branch:     rs2
   //    ALUimm, Load, JALR: Iimm
   wire [31:0] aluIn2 = isALUreg | isBranch ? rs2 : Iimm;

   reg  [31:0] aluReg;       // The internal register of the ALU,
used by shift.
   reg  [4:0]  aluShamt;     // Current shift amount.

   wire aluBusy = |aluShamt; // ALU is busy if shift amount is
non-zero.
   wire aluWr;               // ALU write strobe, starts
shifting.

   // The adder is used by both arithmetic instructions and JALR.
   wire [31:0] aluPlus = aluIn1 + aluIn2;

   // Use a single 33 bits subtract to do subtraction and all
comparisons
   // (trick borrowed from swapforth/J1)
   wire [32:0] aluMinus = {1'b1, ~aluIn2} + {1'b0,aluIn1} +
33'b1;
   wire        LT  = (aluIn1[31] ^ aluIn2[31]) ? aluIn1[31] :
aluMinus[32];
   wire        LTU = aluMinus[32];
   wire        EQ  = (aluMinus[31:0] == 0);

   // Notes:
```

```
   // - instr[30] is 1 for SUB and 0 for ADD
   // - for SUB, need to test also instr[5] to discriminate ADDI:
   //    (1 for ADD/SUB, 0 for ADDI, and Iimm used by ADDI
overlaps bit 30 !)
   // - instr[30] is 1 for SRA (do sign extension) and 0 for SRL

   wire [31:0] aluOut =
      (funct3Is[0] ? instr[30] & instr[5] ? aluMinus[31:0] :
aluPlus : 32'b0) |
      (funct3Is[2] ? {31'b0, LT}
: 32'b0) |
      (funct3Is[3] ? {31'b0, LTU}
: 32'b0) |
      (funct3Is[4] ? aluIn1 ^ aluIn2
: 32'b0) |
      (funct3Is[6] ? aluIn1 | aluIn2
: 32'b0) |
      (funct3Is[7] ? aluIn1 & aluIn2
: 32'b0) |
      (funct3IsShift ? aluReg
: 32'b0) ;

   wire funct3IsShift = funct3Is[1] | funct3Is[5];

   always @(posedge clk) begin
      if(aluWr) begin
         if (funct3IsShift) begin  // SLL, SRA, SRL
        aluReg <= aluIn1;
        aluShamt <= aluIn2[4:0];
     end
      end

`ifdef NRV_TWOLEVEL_SHIFTER
      else if(|aluShamt[4:2]) begin // Shift by 4
         aluShamt <= aluShamt - 4;
     aluReg <= funct3Is[1] ? aluReg << 4 :
          {{4{instr[30] & aluReg[31]}}, aluReg[31:4]};
      end  else
`endif
      // Compact form of:
      // funct3=001               -> SLL  (aluReg <= aluReg << 1)
      // funct3=101 &  instr[30] -> SRA  (aluReg <= {aluReg[31],
aluReg[31:1]})
      // funct3=101 & !instr[30] -> SRL  (aluReg <= {1'b0,
aluReg[31:1]})aluReg[31]is the sign bit
```

```
      if (|aluShamt) begin
         aluShamt <= aluShamt - 1;
      aluReg <= funct3Is[1] ? aluReg << 1 :                 // SLL
            {instr[30] & aluReg[31], aluReg[31:1]};  // SRA,SRL
      end
   end


/******************************************************************
***********/
   // The predicate for conditional branches.

/******************************************************************
***********/

   wire predicate =
        funct3Is[0] &  EQ  | // BEQ
        funct3Is[1] & !EQ  | // BNE
        funct3Is[4] &  LT  | // BLT
        funct3Is[5] & !LT  | // BGE
        funct3Is[6] &  LTU | // BLTU
        funct3Is[7] & !LTU ; // BGEU


/******************************************************************
***********/
   // Program counter and branch target computation.

/******************************************************************
***********/

   reg  [ADDR_WIDTH-1:0] PC; // The program counter.
   reg  [31:2] instr;        // Latched instruction. Note that
bits 0 and 1 are
                             // ignored (not used in RV32I base
instr set).

   wire [ADDR_WIDTH-1:0] PCplus4 = PC + 4;

   // An adder used to compute branch address, JAL address and
AUIPC.
   // branch->PC+Bimm    AUIPC->PC+Uimm    JAL->PC+Jimm
   // Equivalent to PCplusImm = PC + (isJAL ? Jimm : isAUIPC ?
Uimm : Bimm)
   wire [ADDR_WIDTH-1:0] PCplusImm = PC + ( instr[3] ?
Jimm[ADDR_WIDTH-1:0] : instr[4] ? Uimm[ADDR_WIDTH-1:0] :Bimm[ADDR_WIDTH-1:0] );
```

```
   // A separate adder to compute the destination of load/store.
   // testing instr[5] is equivalent to testing isStore in this
context.
   wire [ADDR_WIDTH-1:0] loadstore_addr = rs1[ADDR_WIDTH-1:0] +
           (instr[5] ? Simm[ADDR_WIDTH-1:0] : Iimm[ADDR_WIDTH-1:0]);


   /* verilator lint_off WIDTH */
   // internal address registers and cycles counter may have less
than
   // 32 bits, so we deactivate width test for mem_addr and
writeBackData

   assign mem_addr = state[WAIT_INSTR_bit] |state[FETCH_INSTR_bit] ?
                     PC : loadstore_addr ;



/***************************************************************
***********/
   // The value written back to the register file.

/***************************************************************
***********/

   wire [31:0] writeBackData  =
      (isSYSTEM            ? cycles     : 32'b0) |  // SYSTEM
      (isLUI               ? Uimm       : 32'b0) |  // LUI
      (isALU               ? aluOut     : 32'b0) |  // ALUreg,
ALUimm
      (isAUIPC             ? PCplusImm  : 32'b0) |  // AUIPC
      (isJALR   | isJAL    ? PCplus4    : 32'b0) |  // JAL, JALR
      (isLoad              ? LOAD_data  : 32'b0) ;  // Load

   /* verilator lint_on WIDTH */



/***************************************************************
***********/
   // LOAD/STORE
```

```
/*****************************************************************
***********/

   // All memory accesses are aligned on 32 bits boundary. For
this
   // reason, we need some circuitry that does unaligned halfword
   // and byte load/store, based on:
   // - funct3[1:0]:  00->byte 01->halfword 10->word
   // - mem_addr[1:0]: indicates which byte/halfword is accessed

   wire mem_byteAccess     = instr[13:12] == 2'b00; //
funct3[1:0] == 2'b00;
   wire mem_halfwordAccess = instr[13:12] == 2'b01; //
funct3[1:0] == 2'b01;

   // LOAD, in addition to funct3[1:0], LOAD depends on:
   // - funct3[2] (instr[14]): 0->do sign expansion   1->no signexpansion


   wire LOAD_sign =
    !instr[14] & (mem_byteAccess ? LOAD_byte[7] :
LOAD_halfword[15]);

   wire [31:0] LOAD_data =
         mem_byteAccess ? {{24{LOAD_sign}},     LOAD_byte} :
     mem_halfwordAccess ? {{16{LOAD_sign}}, LOAD_halfword} :
                          mem_rdata ;

   wire [15:0] LOAD_halfword =
           loadstore_addr[1] ? mem_rdata[31:16] :
mem_rdata[15:0];

   wire  [7:0] LOAD_byte =
           loadstore_addr[0] ? LOAD_halfword[15:8] :
LOAD_halfword[7:0];

   // STORE

   assign mem_wdata[ 7: 0] = rs2[7:0];
   assign mem_wdata[15: 8] = loadstore_addr[0] ? rs2[7:0]  :
rs2[15: 8];
   assign mem_wdata[23:16] = loadstore_addr[1] ? rs2[7:0]  :
rs2[23:16];
   assign mem_wdata[31:24] = loadstore_addr[0] ? rs2[7:0]  :
                 loadstore_addr[1] ? rs2[15:8] : rs2[31:24];

   // Remember |→ funct3[1:0] 00->byte 01->halfword 10->word
   //          |→ mem_addr[1:0]: indicates which byte/halfword is accessed
```

```
   // The memory write mask:
   //    1111                     if writing a word
   //    0011 or 1100             if writing a halfword
   //                                (depending on
loadstore_addr[1])
   //    0001, 0010, 0100 or 1000 if writing a byte
   //                                (depending on
loadstore_addr[1:0])

   wire [3:0] STORE_wmask =
          mem_byteAccess      ?
                (loadstore_addr[1] ?
                  (loadstore_addr[0] ? 4'b1000 : 4'b0100) :
                  (loadstore_addr[0] ? 4'b0010 : 4'b0001)
                    ) :
          mem_halfwordAccess ?
                (loadstore_addr[1] ? 4'b1100 : 4'b0011) :
              4'b1111;


/***************************************************************
*********/
   // And, last but not least, the state machine.

/***************************************************************
*********/

   localparam FETCH_INSTR_bit     = 0;
   localparam WAIT_INSTR_bit      = 1;
   localparam EXECUTE_bit         = 2;
   localparam WAIT_ALU_OR_MEM_bit = 3;
   localparam NB_STATES           = 4;

   localparam FETCH_INSTR     = 1 << FETCH_INSTR_bit;
   localparam WAIT_INSTR      = 1 << WAIT_INSTR_bit;
   localparam EXECUTE         = 1 << EXECUTE_bit;
   localparam WAIT_ALU_OR_MEM = 1 << WAIT_ALU_OR_MEM_bit;

   (* onehot *)
   reg [NB_STATES-1:0] state;

   // The signals (internal and external) that are determined
   // combinatorially from state and other signals.

   // register write-back enable.
```

```
   wire writeBack = ~(isBranch | isStore ) &
                (state[EXECUTE_bit] |
state[WAIT_ALU_OR_MEM_bit]);

   // The memory-read signal.
   assign mem_rstrb = state[EXECUTE_bit] & isLoad |
state[FETCH_INSTR_bit];

   // The mask for memory-write.
   assign mem_wmask = {4{state[EXECUTE_bit] & isStore}} &
STORE_wmask;

   // aluWr starts computation (shifts) in the ALU.
   assign aluWr = state[EXECUTE_bit] & isALU;

   wire jumpToPCplusImm = isJAL | (isBranch & predicate);
`ifdef NRV_IS_IO_ADDR
   wire needToWait = isLoad |
             isStore  & `NRV_IS_IO_ADDR(mem_addr) |
             isALU & funct3IsShift;
`else
   wire needToWait = isLoad | isStore | isALU & funct3IsShift;
`endif

   always @(posedge clk) begin
      if(!reset) begin
         state      <= WAIT_ALU_OR_MEM; // Just waiting for
!mem_wbusy
         PC         <= RESET_ADDR[ADDR_WIDTH-1:0];
      end else

      // See note [1] at the end of this file.
      (* parallel_case *)
      case(1'b1)

        state[WAIT_INSTR_bit]: begin
           if(!mem_rbusy) begin // may be high when executing
from SPI flash
              rs1 <= registerFile[mem_rdata[19:15]];
              rs2 <= registerFile[mem_rdata[24:20]];
              instr <= mem_rdata[31:2]; // Bits 0 and 1 are
ignored (see
              state <= EXECUTE;         // also the declaration
of instr).
           end
        end
```

```
                state[EXECUTE_bit]: begin
                   PC <= isJALR          ? {aluPlus[ADDR_WIDTH-1:1],1'b0}
                                         :
                         jumpToPCplusImm ? PCplusImm :
                         PCplus4;
!(state[WAIT_INSTR_bit]
        |
state[FETCH_INSTR_bit]) state <= needToWait ? WAIT_ALU_OR_MEM : FETCH_INSTR;
                end

                state[WAIT_ALU_OR_MEM_bit]: begin
                   if(!aluBusy & !mem_rbusy & !mem_wbusy) state <=FETCH_INSTR;

                end

                default: begin // FETCH_INSTR ↑
                  state <= WAIT_INSTR;
                end

             endcase
          end


/************************************************************************/
   // Cycle counter

/************************************************************************/

`ifdef NRV_COUNTER_WIDTH
   reg [`NRV_COUNTER_WIDTH-1:0]  cycles;
`else
   reg [31:0]  cycles;
`endif
   always @(posedge clk) cycles <= cycles + 1;

`ifdef BENCH
   initial begin
      cycles = 0;
      aluShamt = 0;
      registerFile[0] = 0;
   end
`endif

endmodule
```

```
/*************************************************************
*************/
// Notes:
//
// [1] About the "reverse case" statement, also used in Claire
Wolf's picorv32:
// It is just a cleaner way of writing a series of cascaded if()
statements,
// To understand it, think about the case statement *in general*
as follows:
// case (expr)
//       val_1: statement_1
//       val_2: statement_2
//   ... val_n: statement_n
// endcase
// The first statement_i such that expr == val_i is executed.
// Now if expr is 1'b1:
// case (1'b1)
//       cond_1: statement_1
//       cond_2: statement_2
//   ... cond_n: statement_n
// endcase
// It is *exactly the same thing*, the first statement_i such
that
// expr == cond_i is executed (that is, such that 1'b1 == cond_i,
// in other words, such that cond_i is true)
// More on this:
//     https://stackoverflow.com/questions/15418636/case-
statement-in-verilog
//
// [2] state uses 1-hot encoding (at any time, state has only one
bit set to 1).
// It uses a larger number of bits (one bit per state), but often
results in
// a both more compact (fewer LUTs) and faster state machine.
```

31:25	24:20	19:15	14:12	11:7	6:0	
funct7	rs2	rs1	funct3	rd	op	R-Type
$imm_{11:0}$		rs1	funct3	rd	op	I-Type
$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op	S-Type
$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op	B-Type
$imm_{31:12}$				rd	op	U-Type
$imm_{20,10:1,11,19:12}$				rd	op	J-Type
fs3 (5 bits)	funct2 (2 bits)	fs2 (5 bits)	fs1 (5 bits)	funct3 (3 bits)	fd (5 bits)	op (7 bits)	R4-Type

Figure B.1 RISC-V 32-bit instruction formats

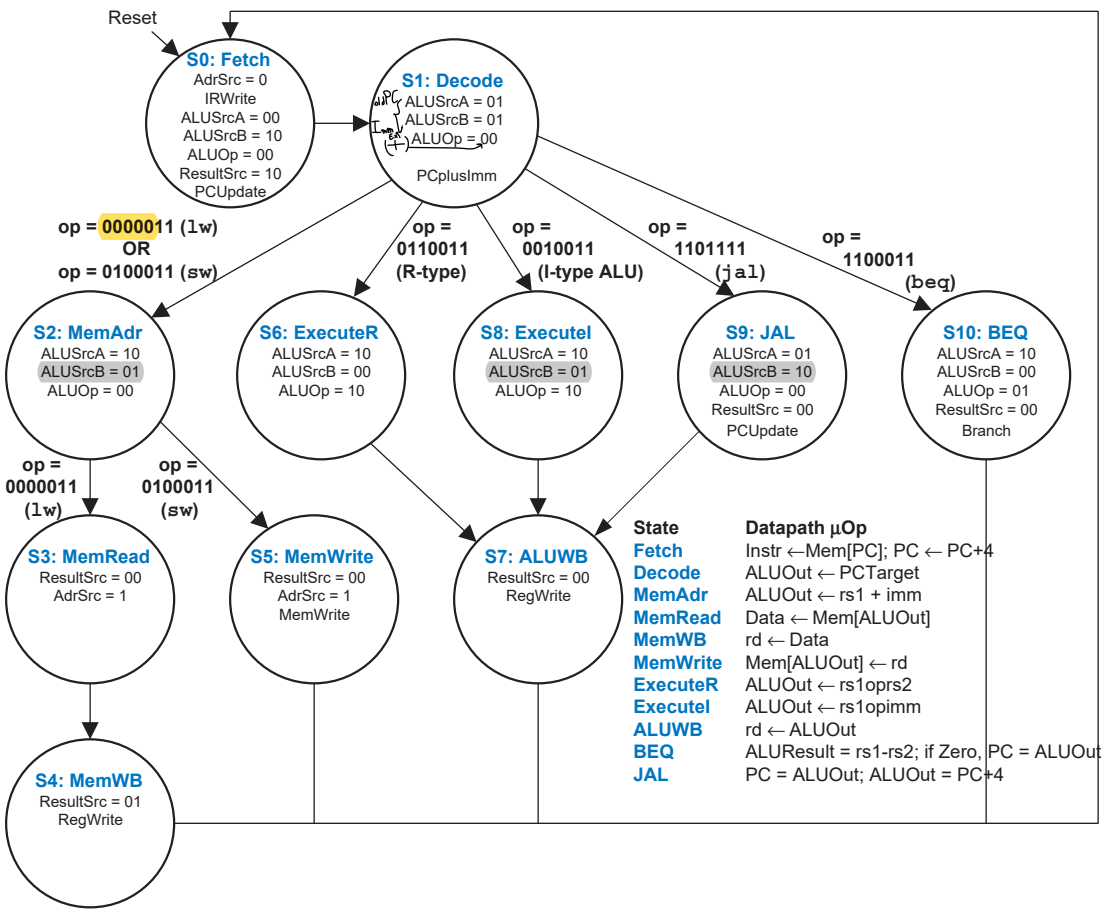
funct3	
000	–
001	–
010	–
100	–
101	–
000	–
001	
010	–
011	–
100	–
101	
101	
110	–
111	–

Figure 6.27 RISC-V immediate encodings in machine instructions

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
																rs1				funct3				rd								I
																rs1				funct3					rs2							S
																rs1				funct3					rs2							B
																				rd												U
																				rd												J

0xFFFFF000

op
0000011 (3)
0000011 (3)
0000011 (3)
0000011 (3)
0000011 (3)
0010011 (19)
0010011 (19)
0010011 (19)
0010011 (19)
0010011 (19)
0010011 (19)
0010011 (19)
0010011 (19)
0010011 (19)
0010111 (23)
0100011 (35)
0100011 (35)
0100011 (35)
0110011 (51)
0110011 (51)
0110011 (51)
0110011 (51)
0110011 (51)
0110011 (51)
0110011 (51)
0110011 (51)
0110011 (51)
0110011 (51)
0110111 (55)
1100011 (99)
1100011 (99)
1100011 (99)
1100011 (99)
1100011 (99)
1100011 (99)



0010011 (19)	000	–	I	addi rd, rs1, imm	add immediate	rd = rs1 + S
0010011 (19)	001	0000000*	I	slli rd, rs1, uimm	shift left logical immediate	rd = rs1 <<
0010011 (19)	010	–	I	slti rd, rs1, imm	set less than immediate	rd = (rs1 < SignExt(imm))
0010011 (19)	011	–	I	sltiu rd, rs1, imm	set less than imm. unsigned	rd = (rs1 < SignExt(imm))
0010011 (19)	100	–	I	xori rd, rs1, imm	xor immediate	rd = rs1 ^ S
0010011 (19)	101	0000000*	I	srli rd, rs1, uimm	shift right logical immediate	rd = rs1 >>
0010011 (19)	101	0100000*	I	srai rd, rs1, uimm	shift right arithmetic imm.	rd = rs1 >>> uimm
0010011 (19)	110	–	I	ori rd, rs1,	or immediate	rd = rs1 | S
0010011 (19)	111	–	I	andi rd, rs1,	and immediate	rd = rs1 & S

0100011 (35)	000	–	S	sb rs2, imm(rs1)	store byte	$[Address]_{7:0} = rs2_{7:0}$
0100011 (35)	001	–	S	sh rs2, imm(rs1)	store half	$[Address]_{15:0} = rs2_{15:0}$
0100011 (35)	010	–	S	sw rs2, imm(rs1)	store word	$[Address]_{31:0} = rs2$

0110011 (51)	000	0000000	R	add rd, rs1,	add	rd = rs1 +
0110011 (51)	000	0100000	R	sub rd, rs1,	sub	rd = rs1 −
0110011 (51)	001	0000000	R	sll rd, rs1,	shift left logical	rd = rs1 << $_{4:0}$
0110011 (51)	010	0000000	R	slt rd, rs1,	set less than	rd = (rs1 < rs2)
0110011 (51)	011	0000000	R	sltu rd, rs1,	set less than unsigned	rd = (rs1 < rs2)
0110011 (51)	100	0000000	R	xor rd, rs1,	xor	rd = rs1 ^
0110011 (51)	101	0000000	R	srl rd, rs1,	shift right logical	rd = rs1 >> $_{4:0}$
0110011 (51)	101	0100000	R	sra rd, rs1,	shift right arithmetic	rd = rs1 >>> $rs2_{4:0}$
0110011 (51)	110	0000000	R	or rd, rs1, r	or	rd = rs1 |
0110011 (51)	111	0000000	R	and rd, rs1,	and	rd = rs1 &

Table B.8 Privileged / CSR instructions

op	funct3	Type	Instruction	Description	Operation
1110011 (115)	000	I	ecall	transfer control to OS	
1110011 (115)	000	I	ebreak	transfer control to debugger (imm=1)	
1110011 (115)	000	I	uret	return from user exception	PC = uepc
1110011 (115)	000	I	sret	return from supervisor exception (rs1=0,rd=0,imm=258)	PC = sepc
1110011 (115)	000	I	mret	return from machine exception (rs1=0,rd=0,imm=770)	PC = mepc
1110011 (115)	001	I	csrrw rd,csr,rs1	CSR read/write (i	rd = csr,csr = rs1
1110011 (115)	010	I	csrrs rd,csr,rs1	CSR read/set (imm	rd = csr,csr = csr | rs1
1110011 (115)	011	I	csrrc rd,csr,rs1	CSR read/clear (i	rd = csr,csr = csr & ~rs1
1110011 (115)	101	I	csrrwi rd,csr,uimm	CSR read/write immediate	rd = csr,csr = ZeroExt(uimm)
1110011 (115)	110	I	csrrsi rd,csr,uimm	CSR read/set immediate	rd = csr, csr = csr | ZeroExt(uimm)
1110011 (115)	111	I	csrrci rd,csr,uimm	CSR read/clear immediate	rd = csr, csr = csr & ~ZeroExt(uimm)

For privileged / CSR instructions, the 5-bit unsigned immediate, uimm, is encoded in the **rs1** field.

1100111 (103)	000	–	I	jalr rd, rs1, imm	jump and link register	PC = rs1 + SignExt(imm), rd = PC + 4

0010111 (23)	–	–	U	auipc rd, upimm	add upper immediate to PC	rd = {upimm, 12'b0} + PC

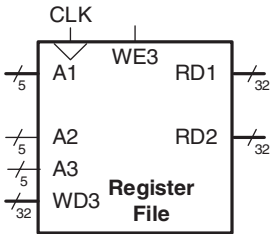
back to PC + Uimm

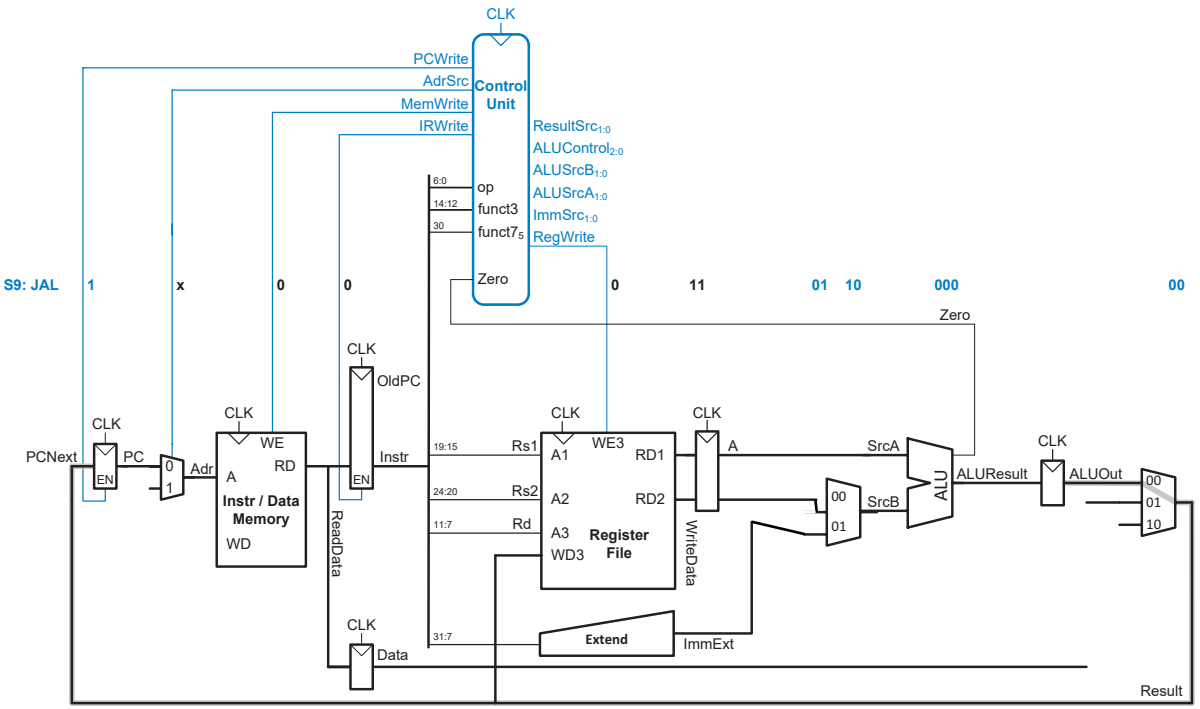
1101111 (111)	–	–	J	jal rd, label	jump and link	PC = JTA, rd = PC + 4
1100111 (103)	000	–	I	jalr rd, rs1, imm	jump and link register	PC = rs1 + SignExt(imm), rd = PC + 4
0110111 (55)	–	–	U	lui rd, upimm	load upper immediate	rd = {upimm, 12'b0}

1100011 (99)	000	–	B	`beq rs1, rs2, label`	branch if =	`if (rs1 == rs2) PC = BTA`
1100011 (99)	001	–	B	`bne rs1, rs2, label`	branch if ≠	`if (rs1 ≠ rs2) PC = BTA`
1100011 (99)	100	–	B	`blt rs1, rs2, label`	branch if <	`if (rs1 < rs2) PC = BTA`
1100011 (99)	101	–	B	`bge rs1, rs2, label`	branch if ≥	`if (rs1 ≥ rs2) PC = BTA`
1100011 (99)	110	–	B	`bltu rs1, rs2, label`	branch if < unsigned	`if (rs1 < rs2) PC = BTA`
1100011 (99)	111	–	B	`bgeu rs1, rs2, label`	branch if ≥ unsigned	`if (rs1 ≥ rs2) PC = BTA`

Table B.4 Register names and numbers

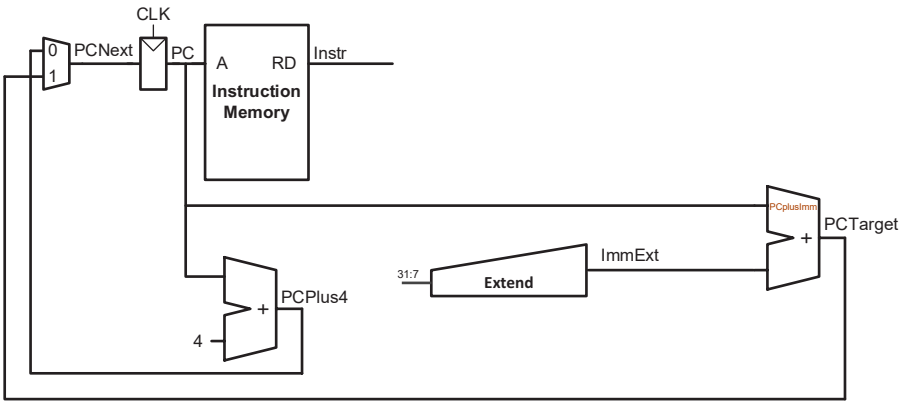
Name	Register Number	Use
zero	x0	Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0–2	x5–7	Temporary registers
s0/fp	x8	Saved register / Frame pointer
s1	x9	Saved register
a0–1	x10–11	Function arguments / Return values
a2–7	x12–17	Function arguments
s2–11	x18–27	Saved registers
t3-6	x28–31	Temporary registers

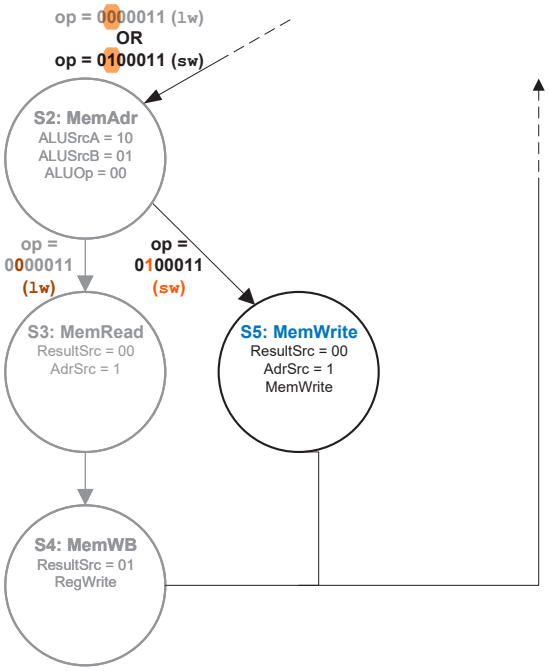


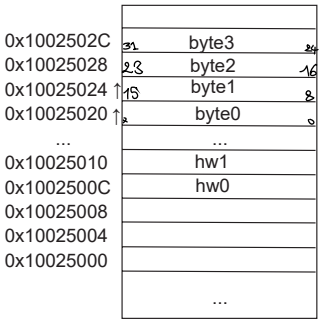


op	funct3	funct7	Type	Instruction	Description	Operation
0000011 (3)	000	–	I	`lb rd, imm(rs`	load byte	`rd = SignExt([Address]7:0)`
0000011 (3)	001	–	I	`lh rd, imm(rs`	load half	`rd = SignExt([Address]15:0)`
0000011 (3)	010	–	I	`lw rd, imm(rs`	load word	`rd = [Addr 31:0`
0000011 (3)	100	–	I	`lbu rd, imm(r`	load byte unsigned	`rd = ZeroExt([Address]7:0)`
0000011 (3)	101	–	I	`lhu rd, imm(r`	load half unsigned	`rd = ZeroExt([Address]15:0)`

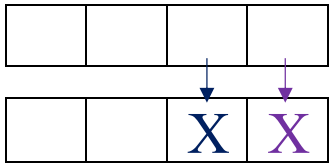
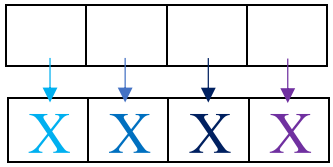
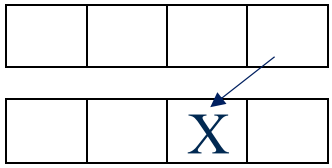
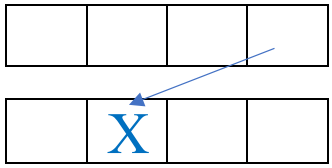
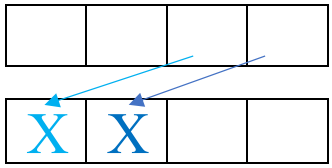
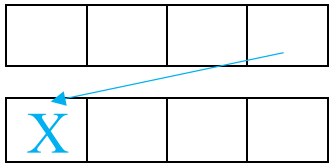
For 32-bit instructions, bits 1:0 of the 13-bit branch offset ($\mathbf{imm}_{12:0}$) are always zero because 32-bit instructions occupy 4 bytes of memory. Thus, instruction addresses are always divisible by four and neither bits 1 nor 0 of the branch offset need to be encoded in the instruction. However, RV32I only omits bit 0. This enables compatibility with 16-bit (2-byte) RISC-V compressed instructions (see Section 6.6.5). Compilers can then mix 16-bit and 32-bit instructions if the processor hardware supports both instruction sizes.







loadstore_addr[1]	loadstore_addr[0]	LOAD_halfword	LOAD_byte
0	0	`mem_rdata[15:0]`	`mem_rdata[7:0]`
0	1	`mem_rdata[15:0]`	`mem_rdata[15:8]`
1	0	`mem_rdata[31:16]`	`mem_rdata[23:16]`
1	1	`mem_rdata[31:16]`	`mem_rdata[31:24]`

LOADSTORE_ADDR	BYTE	HALFWORD	WORD
00			
01			
10			
11			

• Bibliography & References :

- Patterson & Harris textbook — RISC-V Instruction Set Summary
- RISC-V official spec — RISC-V Ratified Specifications
- Intel/Altera FPGA tools documentation — terasic.com.tw/attachment/archive/226/DE2_70_User_manual_v105.pdf
- HDL optimization papers / articles — (PDF) Survey of HDL Compiler Optimization Techniques
- Verilog 2001 Documentation — leiblog.wang/static/FPGA/books/standard/IEEE_Verilog.pdf
- Intel® Quartus® Prime Standard Edition User Guides Combined — Intel® Quartus® Prime Standard Edition User Guides - Combined
- Hhp3 RISC-V Youtube Playlist — RISC-V Assembly Code #1: Course Intro, Registers