CS3520 - Microarchitecture Design Document

Processor: MedAI Healthcare Accelerator

Group: No Blueprint

Date: 27 October 2025

# 1. INTRODUCTION

### Reference ISA Overview

The MedAI Instruction Set Architecture (ISA) represents a specialized RISC-AI hybrid architecture engineered for AI-driven mobile healthcare systems deployed across Africa's remote regions. This ISA strategically combines base RISC architectural simplicity with domain-specific AI instructions to support three critical healthcare workloads: geospatial analysis for provider location services, fingerprint recognition for patient identification, and image-based triage for medical diagnostics.

#### Key architectural features include:

• 32 general-purpose 32-bit registers (R0-R31)

• Dedicated floating-point registers (F0-F15) for precision computations

• Support for multiple data types: 8-bit, 16-bit, and 32-bit integers; 32-bit floating-point values; and 128-bit vector arrays for parallel processing

• Fixed 32-bit instruction encoding with R-type, I-type, and S-type formats

• Little-endian memory model with word-aligned (4-byte) access patterns

### Supported Instruction Classes

#### Core RISC Operations

• Integer Arithmetic: ADD, SUB, MUL, DIV

• Logical Operations: AND, OR, XOR, NOT

• Control Flow: BEQ (Branch Equal), BNE (Branch Not Equal), JMP (Unconditional Jump) Memory Operations

• Load/Store: LD (Load Word), ST (Store Word)

• Addressing Modes: Immediate, register-indirect, base+offset, PC-relative

#### Floating-Point Operations

• FP Arithmetic: FADD (Floating Add), FMUL (Floating Multiply), FDIV (Floating Divide)

• Precision: 32-bit single-precision floating-point format

#### Vector Operations

• SIMD Processing: VADD (Vector Add), VMUL (Vector Multiply)

• Data Width: 128-bit vector registers supporting parallel operations

#### Domain-Specific AI Operations

• Geospatial Processing:

- GEO\_DIST: Compute haversine distance between geographic coordinates

- GEO\_NEAR: Identify nearest healthcare provider from coordinate database

• Biometric Processing:

- FREC\_MATCH: Compare fingerprint descriptors and return match score

• Image Processing:

- IMG\_SIFT: Perform SIFT-like keypoint extraction from medical images

- IMG\_NORM: Normalize descriptor vectors for machine learning

## Design Objective

The primary design objective for the MedAI microarchitecture is to enable simple pipeline implementation while maintaining high performance for medical AI workloads. Specific design goals include:

#### Minimize Hardware Complexity

• Implement a single unified AI accelerator that replaces multiple specialized execution units

• Reduce control logic overhead through fixed instruction formats

• Eliminate redundant datapath components through strategic resource sharing

Optimize for Target Healthcare Workloads

• Streamline datapath for geospatial computations (coordinate processing)

• Accelerate biometric matching operations (fingerprint recognition)

• Optimize for medical image processing workloads (SIFT feature extraction)

#### Power Efficiency

• Simple control logic reduces dynamic power consumption

• Efficient execution path minimizes switching activity

• Low-power design suitable for edge devices and mobile healthcare applications

#### Scalable Performance

• Clean pipeline design enables future extension with additional parallel execution units

• Modular architecture supports performance scaling across microcontroller to cloud deployments

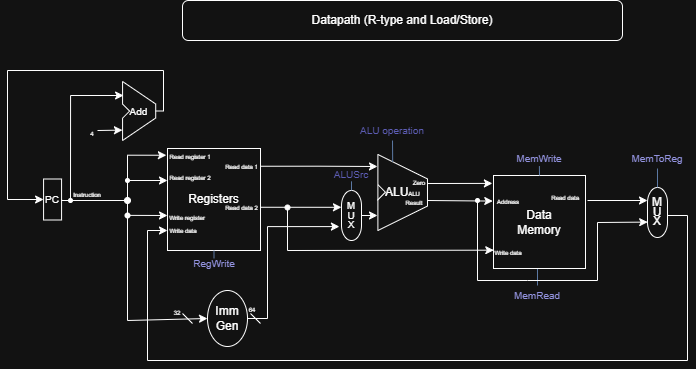
#### Reduced Control Overhead

• Fixed 32-bit instruction encoding simplifies instruction decode

• Minimal control signals reduce pipeline staging complexity

• Straightforward hazard detection and resolution mechanisms

# 2. Incremental Datapath Design



## Annotations of the Datapath

The diagram can be broken down into several key sections, traced by the flow of an instruction:

Instruction Fetch :

PC (Program Counter): The register that holds the address of the current instruction.

Instruction Memory: The memory module that supplies the instruction based on the PC's address.

Adder (PC+4): Increments the PC by 4 (the size of one instruction) to point to the next sequential instruction.

Instruction Decode & Register Read:

Registers (Register File): Contains the 32 general-purpose registers.

Read register 1/2: The instruction's fields (e.g., rs1, rs2) select which registers to read.

Read data 1/2: The values from the selected registers are output here.

Write register: The instruction field (e.g., rd) selects which register to write to later.

Write data: The data to be written back into the register file.

## Execution:

Imm Gen (Immediate Generator): Extracts and sign-extends the immediate value from the instruction.

ALUSrc MUX: Controlled by the ALUSrc signal. Chooses between Read data 2 (for R-type) and the Immediate value (for Load/Store) as the second input to the ALU.

ALU (Arithmetic Logic Unit): Performs the operation (e.g., add, subtract, compare). The specific operation is controlled by the ALU operation signal.

Zero: An output from the ALU that is '1' if the result of the operation is zero. This is crucial for branch instructions.

Memory Access:

Data Memory: For Load (lw) instructions, it reads data from a memory address. For Store (sw) instructions, it writes data to a memory address.

Address: The ALU's result is used as the memory address.

Write data: For Store instructions, this is the data from Read data 2 that will be written to memory.

Read data: For Load instructions, this is the data read from memory.

MemRead/MemWrite: Control signals that enable reading from or writing to the data memory.

Write-Back :

MemtoReg MUX: Controlled by the MemToReg signal. This is the critical mux that determines what gets written back to the register file.

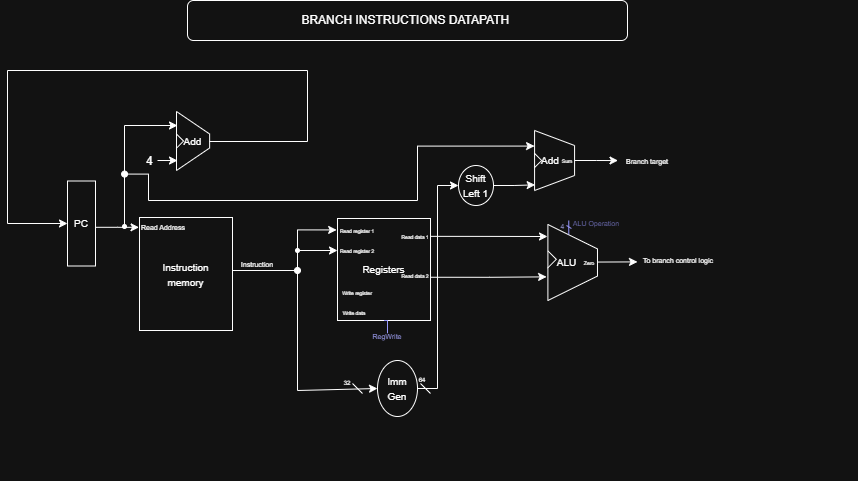
For an R-type instruction, it selects the ALU Result.

For a Load instruction, it selects the Read data from memory.

The output of this mux is connected to the Write data input of the register file. The RegWrite signal must be enabled to perform the actual write.

## Paragraph Explanation

This diagram illustrates the single-cycle datapath for a RISC-V processor, designed to execute R-type and Load/Store instructions. The process begins with the Program Counter (PC) fetching an instruction from Instruction Memory. After the instruction is fetched, it is decoded: the register fields are used to read two values from the Register File, and the immediate field is extracted and sign-extended by the Immediate Generator. During the execution phase, a multiplexer (ALUSrc), controlled by the main Control Unit, selects either the second register value (for R-type) or the immediate value (for address calculation in Load/Store) as the second operand for the ALU. The ALU then performs the required operation, such as an arithmetic function for R-type or calculating a memory address for Load/Store. For memory instructions, the Data Memory is accessed: it is read for a Load instruction (MemRead) or written to for a Store instruction (MemWrite). Finally, in the write-back stage, a second multiplexer (MemtoReg) selects the correct value to write back to the destination register. It chooses the ALU result for R-type instructions or the data read from memory for Load instructions. The RegWrite signal is activated to finalize this write-back, completing the instruction's journey through the datapath.



## Annotated Dataflows and Key Control Signals

## Dataflows:

PC to Instruction Memory: The Program Counter (PC) provides the address to fetch the current instruction (Instruction[31-0]).

Instruction to Register File: The instruction bits [19-15] (rs1) and [24-20] (rs2) are fed into the register file to specify the two source registers to be compared.

Register File to ALU: The values read from the two source registers (Read data 1 and Read data 2) are sent to the ALU.

ALU Comparison: The ALU performs a subtraction (Read data 1 - Read data 2) to check for equality. The Zero output is set to 1 if the result is zero (i.e., the two register values are equal).

Immediate Generation: The immediate field (Instruction[31-25] and [11-7]) is extracted and sign-extended by the Imm Gen unit.

Branch Offset Calculation: The sign-extended immediate is passed to the Shift Left 1 unit, which multiplies it by 2 (converts a word offset to a byte offset). This shifted value is then added to the current PC value by the second Add unit. The result is the Branch Target address.

PC Update Path: The first Add unit continuously calculates PC + 4. A multiplexer (MUX) chooses between PC + 4 (the next sequential instruction) and the Branch Target address. The selected value is written back into the PC on the next clock cycle.

## Key Control Signals:

ALU Operation: Set to Subtract (or a control code that triggers a comparison) to enable the equality check.

Branch: This signal, generated by the main control unit based on the instruction's opcode, indicates that the instruction is a branch.

PCSrc: This is the critical control signal for the branch datapath. It is generated by the Branch Control Logic (an AND gate) which takes the Branch signal and the ALU's Zero output as inputs.

PCSrc = Branch AND Zero (for BEQ)

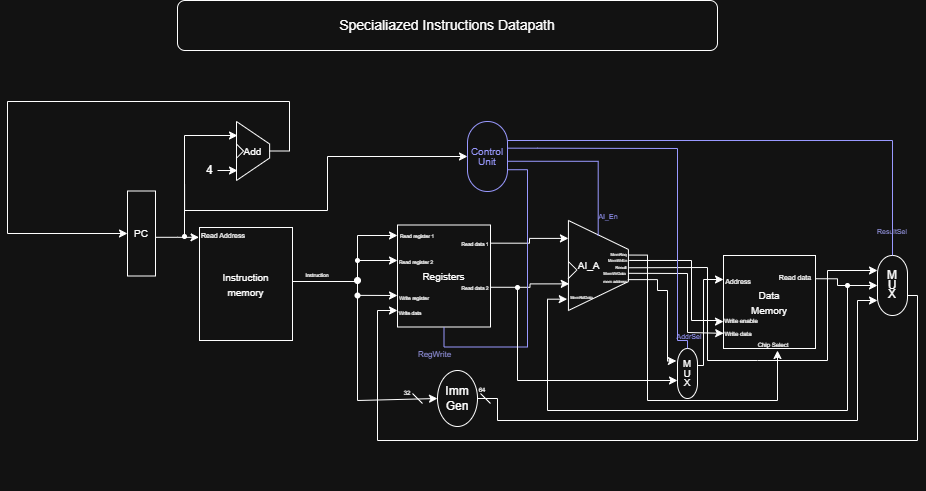
PCSrc controls the PC source MUX.

PCSrc = 0 -> Select PC + 4 (do not branch).

PCSrc = 1 -> Select Branch Target (take the branch).

## Execution Explanation

A branch instruction executes by first fetching the instruction from memory using the PC. The instruction is decoded, and the two source registers specified in the instruction are read from the register file. These values are compared by the ALU, which outputs a Zero flag if they are equal. Concurrently, the immediate field from the instruction is sign-extended and shifted to calculate the branch target address relative to the current PC. The control unit generates the Branch signal. The PCSrc signal is then determined by the AND of Branch and Zero (or its inverse for BNE). This PCSrc signal controls a multiplexer that selects either the sequential address (PC+4) or the branch target address to be loaded into the PC, thereby deciding the path of execution for the next instruction.



## Annotated Dataflows & Control Signals:

## Dataflows:

Instruction Fetch & Decode: The instruction is fetched. The Control Unit decodes the specialized opcode.

Operand Fetch & Dispatch: The Rs1 and Rs2 operands are read from the Register File and routed directly to the AI Accelerator (AI\_A).

Specialized Execution: The AI Accelerator performs its internal, complex operation (e.g., geospatial calculation, fingerprint matching). The main ALU is not used.

Write-Back: The result from the AI Accelerator is selected by the main result MUX (controlled by ResultSel) and written back to the destination register Rd.

## Key Control Signals:

AI\_En: The crucial signal that enables and activates the AI Accelerator unit.

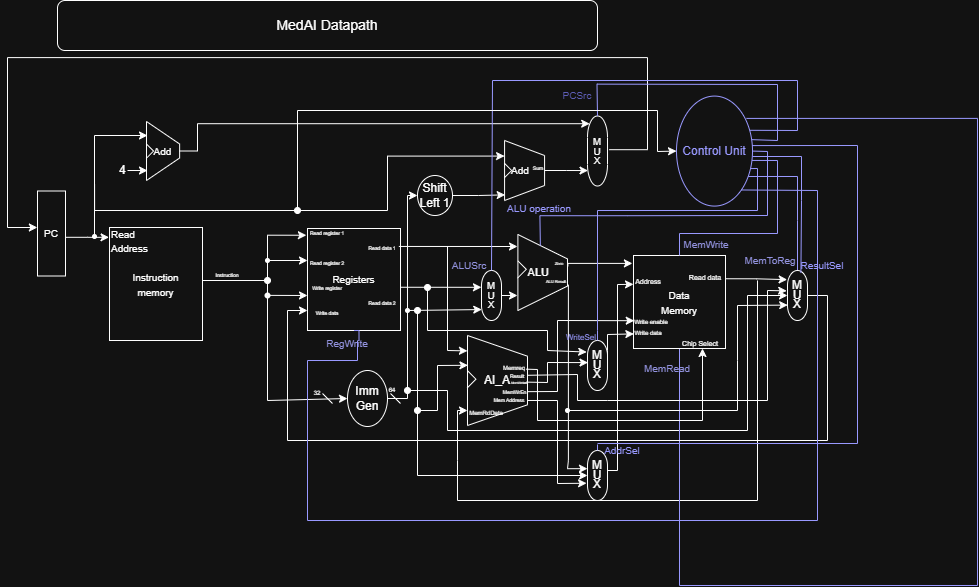
ResultSel: Configured to select the output from the AI\_A unit as the write-back source.

RegWrite: Set to 1 to write the accelerator's result back to the register file.

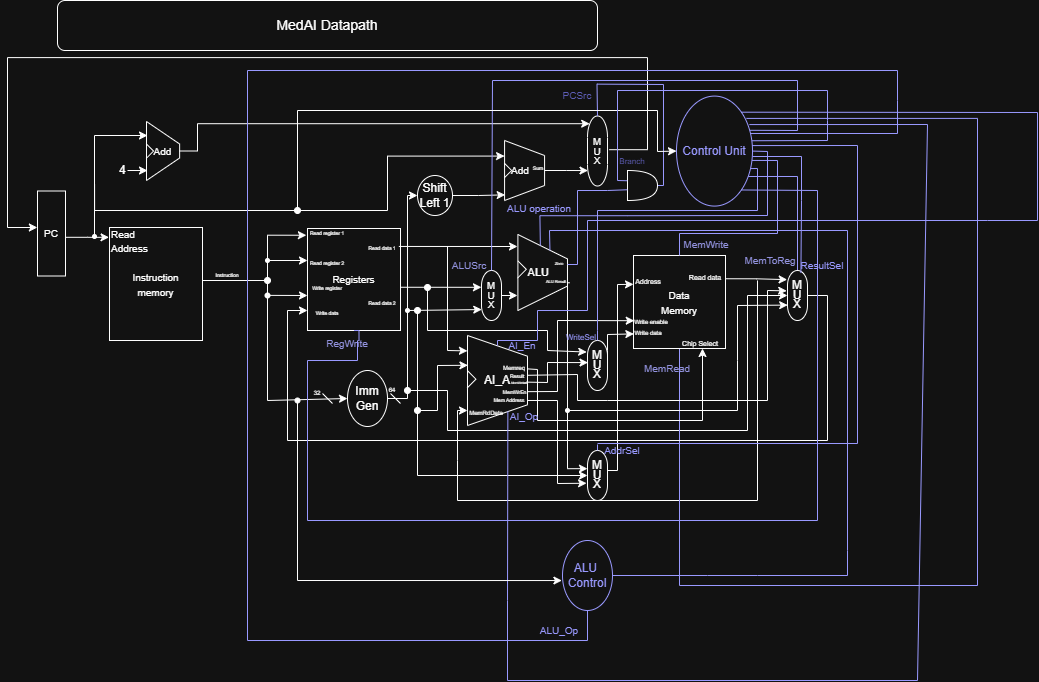
## Explanation:

S-Type instructions leverage the custom AI Accelerator to offload complex, specialized computations from the main CPU core. Execution starts normally with an instruction fetch. The key difference is that upon decode, the control unit activates the AI\_En signal. The source operands from the register file are bypassed around the standard ALU and fed directly into the accelerator. The accelerator then executes its dedicated function, which may take multiple cycles. Once complete, the result is channeled back into the processor's pipeline through the result multiplexer and is written to the destination register, seamlessly integrating the specialized hardware into the standard instruction flow.

# 3. Unified Single-Cycle Datapath



# 4. Control Unit Design



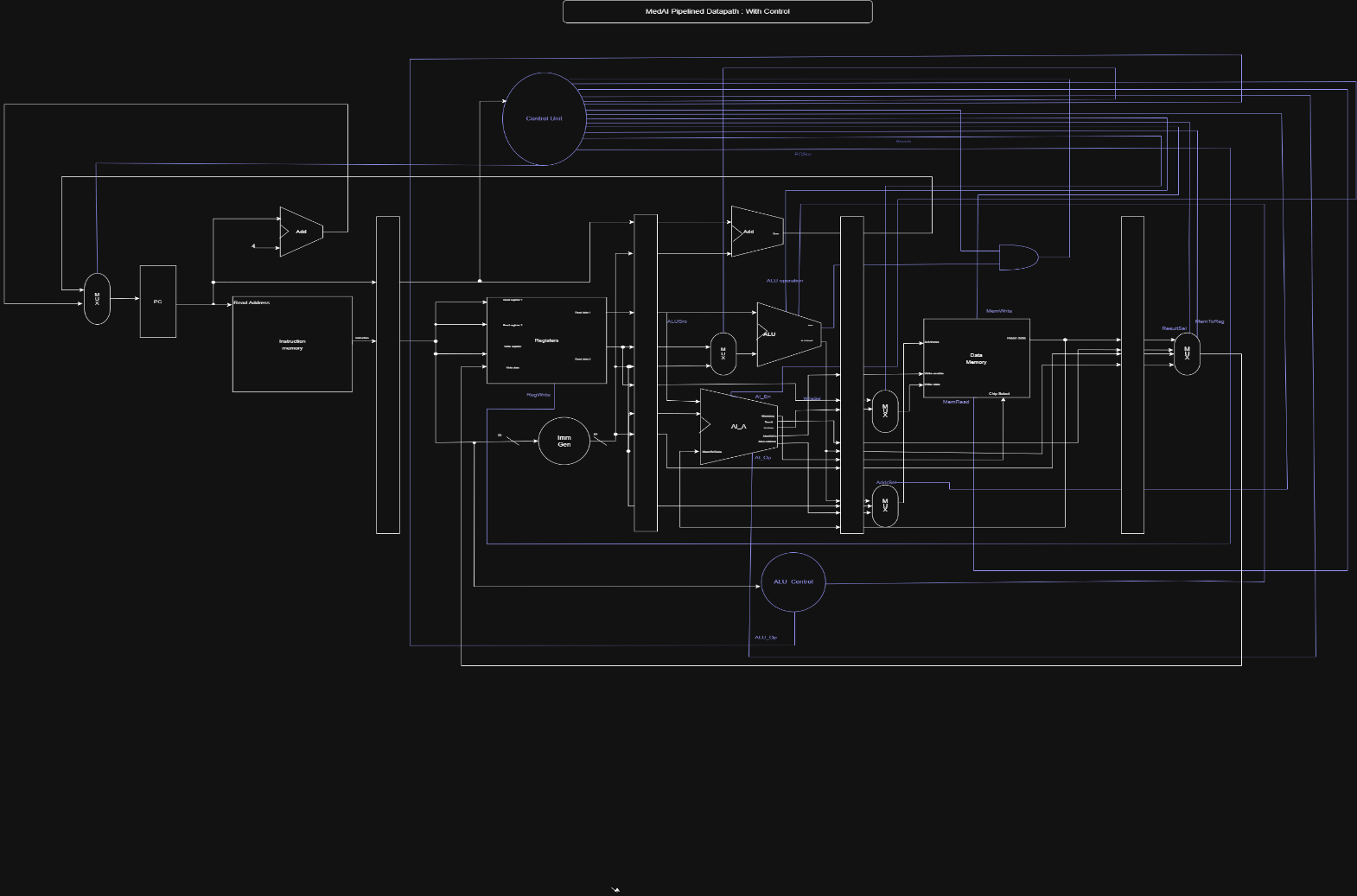
|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Instruction** | **Examples**  **Mnemonic** | **ALUSrc** | **ALU**  **Operation** | **MemRead** | **MemWrite** | **MemToReg** | **RegWrite** | **AddrSel** | **ResultSel** | **AI\_En** |
| R-type | ADD, SUB, AND, OR,  XOR, SLL, SRL, SRA | 0 | FUNCT | 0 | 0 | X | 1 | 0 | 0 | 0 |
| I-type (arith) | ADDI,  ANDI, ORI, XORI, SLLI, SRLI | 1 | FUNCT | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| I-type (load) | LB, LH, LW | 1 | ADD | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| I-type (JALR) | JALR | 1 | ADD | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| S-type | SB, SH, SW | 1 | ADD | 0 | 1 | X | 0 | 0 | 0 | 0 |
| B-type | BEQ, BNE, BLT, BGE | 0 | SUB | 0 | 0 | X | 0 | 0 | 0 | 0 |
| U-type | LUI,  AUIPC | 1 | ADD/PASS | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| J-type | JAL | 1 | ADD | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| Floating Point | FADD, FMUL, ... | X | X | 0 | 0 | 0 | 1 |  |  | 1 |
| Vector Ops | VADD,  VMUL, ... | X | X | 0 | 0 | 0 | 1 |  |  | 1 |
| Geospatial | GEO\_DIS T, GEO\_NEA R | X | ADD | 0 | 0 | 0 | 1 |  |  | 1 |
| Fingerprint | FREC\_MA  TCH | X | ADD | 0 | 0 | 0 | 1 |  |  | 1 |
| Image  Processing | IMG\_NOR  M | X | ADD | 0 | 0 | 0 | 1 |  |  | 1 |
| System/Power | SLP | X | X | 0 | 0 | 0 | 0 |  |  | 1 |

Floating point & Vector: main computation happens in specialized units. Integer ALU signals are “don’t care” (X). Result comes from FPU / Vector units and writes to their register files (FPR/VREG). AI\_En = 0.

Geospatial / Fingerprint / Image Processing: handled by AI\_Unit. The integer ALU is kept as ADD because it commonly performs base+offset/address calc; ALUSrc=1 (immediate/address). AI\_En=1, AddrSel=1 so AI\_Unit can pick its operand/addressing mode. ResultSel=AI and write target is normally a GPR (WriteSel = GPR) unless you design it to write a special register file — adjust if needed. AI\_Op differentiates the sub-type (GEO / FPRINT / IMG). MemRead/MemWrite are 0 unless the particular AI instruction loads/stores memory.

System/Power: typically CSR reads/writes, trap, ecall — handled by CSR logic. They normally do not enable AI\_Unit or memory; PCSrc normally 0 (except for system RET or exceptions which change PC — set PCSrc=1 for exception return/jump instructions).

# 5. Pipelined Implementation

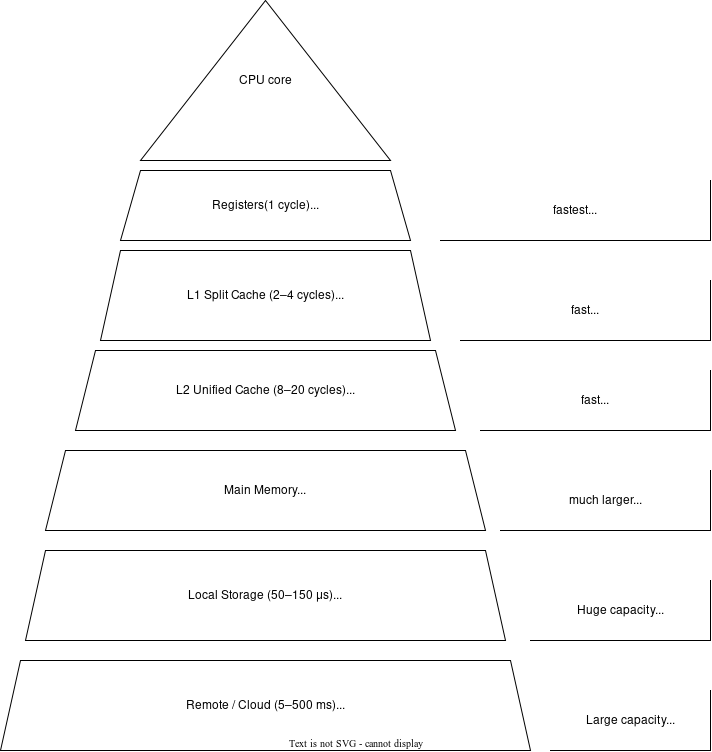


Data Hazards & Forwarding  
Data hazards, specifically Read-After-Write (RAW), are prevalent in chained computational sequences common to medical AI tasks like image feature extraction (IMG\_SIFT) followed by normalization (IMG\_NORM). To minimize performance-degrading pipeline stalls, a comprehensive **forwarding (bypassing) network** is implemented. This network routes results directly from the output of the EX/MEM and MEM/WB pipeline registers back to the ALU inputs in the EX stage. This allows a dependent instruction to proceed without delay once the result is computed, rather than waiting for it to be written back to the register file. Crucially, this forwarding logic is fully integrated with the unified AI accelerator, ensuring domain-specific operations like GEO\_DIST and FREC\_MATCH can also be seamlessly chained. The sole exception is the load-use hazard, which requires a single **pipeline interlock** to stall the pipeline for one cycle, as the data from a load instruction is not available until the end of the MEM stage.

Control Hazards & Branch Prediction  
Control hazards from branches (BEQ, BNE) introduce a potential 2-cycle penalty as the pipeline fetches incorrect instructions until the branch direction is resolved in the EX stage. To maintain hardware simplicity, the baseline strategy is **static "always-not-taken"** prediction, where the pipeline flushes incorrectly fetched instructions only if a branch is taken. A highly effective, low-overhead optimization is the implementation of a **branch delay slot**. The instruction immediately following a branch is always executed, effectively hiding the branch penalty for one instruction and relying on compiler scheduling to fill the slot with useful, independent work. For enhanced performance in future scalable implementations, a simple 1-bit Branch History Table (BHT) can be added to predict loop exits and other repetitive branches common in vector and image processing code.

Domain-Specific Optimizations  
To further address the target workloads, the unified AI accelerator is internally pipelined, enabling it to process multiple operations concurrently without stalling the entire CPU core. Furthermore, a small, lockable data cache is recommended to efficiently handle the high-bandwidth, sequential memory access patterns inherent to medical image processing, ensuring the pipeline remains fed with data and minimizing memory latency stalls.

# 6.Memory Hierarchy Design



# 7.Conclusion

This microarchitecture prioritizes a clean, straightforward implementation that maintains the ISA's healthcare focus while enabling efficient real-time processing on both low-power edge devices and high-performance cloud systems, addressing the critical need for accessible medical AI infrastructure across diverse deployment environments.