**Group Name: Binary Cartel**

**Project Title: Health Monitoring System**

**Week 7: Micro-Architecture**

**Course: CS3520**

**Date: 25-10-2025**

**INTRODUCTION**

Overview and Motivation: Processor Context and Target Application Domain

The BCHMS-32 (Binary Cartel Health Monitoring System ISA) is an AI-powered, RISC-based microprocessor designed for next-generation mobile health monitoring and predictive wellness devices. It is optimized for continuous, intelligent processing of multi-sensor data such as heart rate, body temperature, motion, and oxygen saturation directly on the device. The BCHMS-32 integrates lightweight machine learning (ML) acceleration for on-device inference, enabling real-time health anomaly detection, adaptive activity recognition, and personalized health insights without relying on constant cloud connectivity.

Building on the principles of Reduced Instruction Set Computing (RISC) and inspired by RISC-V, BCHMS-32 features a streamlined 32-bit instruction set with uniform encoding to simplify decoding and enhance pipelining efficiency. Its microarchitecture combines low-power data processing with AI-oriented extensions for efficient vector and floating-point operations, which are crucial for computing moving averages, filtering noisy data, and running compact neural network models. This makes it ideal for AI-enabled mobile processors embedded in phones, that require both efficiency and intelligence at the edge.

**The BCHMS-32 supports the following instruction classes:**

- Arithmetic Instructions: Integer operations including addition, subtraction, multiplication, division, and immediate arithmetic.

- Logical Instructions: Bitwise and shift operations for flexible data manipulation.

- Floating-Point Instructions: Precise floating-point arithmetic and type conversions for AI/ML computations.

- Memory Instructions: Fast and efficient data transfers between registers and memory, supporting word and byte granularity.

- Branch and Jump Instructions: Efficient program control for conditional execution, loops, and subroutines.

- Comparison Instructions: Value comparison and flag setting for decision logic and threshold detection.

- System/I-O Instructions: Manage sensor input/output, data streaming, and inter-module communication

**The BCHMS-32 architecture is guided by the following design objectives:**

- Low Power, High Efficiency: Maintain ultra-low power operation suitable for continuous mobile and wearable use.

- AI-Enhanced Real-Time Processing: Support embedded ML inference for health prediction and anomaly detection directly on the device.

- Simplified Hardware Implementation: Preserve a clean Datapath and reduced control complexity for compact, scalable deployment.

- Predictable Pipeline Performance: Implement a 5-stage pipeline (Fetch, Decode, Execute, Memory, Writeback) optimized for deterministic timing.

- Scalability and Extensibility: Reserve opcode space and microarchitectural flexibility for future extensions such as deep learning accelerators, wireless connectivity, and secure data handling.

The BCHMS-32 thus bridges the gap between embedded efficiency and AI intelligence, enabling smarter, more responsive, and power-aware health monitoring within next-generation mobile devices.

**INCREMENTAL DATAPATH DESIGN**

**R-type**

In this format (R-type), the Program Counter (PC) provides an address that is sent via the address bus to the instruction memory. The corresponding instruction is fetched, and simultaneously, the PC is incremented by 4 to point to the next instruction.

From the fetched instruction, the opcode is decoded to identify it as an R-type instruction. The register file is then accessed to read the values of the two source registers (**rs** and **rt**) specified in the instruction. These operand values are forwarded to the Arithmetic Logic Unit (ALU), along with the ALU control signal derived from the **funct** field of the instruction, to perform the required operation (e.g., add, subtract, AND, OR).

Since R-type instructions do not access data memory, the **MemRead** and **MemWrite** control signals are both asserted as 0. The ALU result is then passed through the data path directly to the write-back stage, where it is written into the destination register (**rd**) in the register file, as enabled by the **RegWrite** control signal.

**I-type**

In an I-type load instruction (such as **lw**), the instruction is first fetched and decoded. The base address is obtained from the source register (**rs**) specified in the instruction, which is read from the register file. The 16-bit signed immediate (offset) from the instruction is sign-extended to 32 bits.

The ALU then adds the sign-extended offset to the base address (from **rs**) to compute the effective memory address. This address is sent to the data memory unit, and the **MemRead** control signal is asserted to enable a read operation.

The data retrieved from memory is then placed on the data bus and forwarded to the write-back stage, where it is written into the destination register (rt) in the register file, as controlled by the **RegWrite** signal.

**F-type (Floating-Point) Instructions**

In the F-type format, used for floating-point operations, the Program Counter (PC) supplies an address that is sent over the address bus to the instruction memory. The corresponding instruction is fetched, and the PC is incremented by 4 to point to the next sequential instruction.

The opcode field of the fetched instruction is decoded by the control unit, identifying it as an F-type operation. Unlike R-type instructions that use the general-purpose register file, the F-type instruction accesses the floating-point register file to read the values from the two source registers (Fs1and Fs2) specified in the instruction.

These floating-point operand values are then routed to the Floating-Point Unit (FPU). The specific operation to be performed (e.g., addition, subtraction, multiplication) is determined by the **funct**field of the instruction, which generates the FPUOp control signal.

Since F-type instructions are register-to-register operations and do not access the main data memory, the MemRead and MemWrite control signals are deasserted (0). The result computed by the FPU is passed directly to the write-back stage. Finally, the result is written into the destination floating-point register (Fd) in the floating-point register file, a process enabled by asserting the FPRegWrite control signal.

**J-type (Jump/Branch) Instructions**

For J-type instructions, which handle control flow changes like jumps and branches, the Program Counter (PC) provides the address for fetching the instruction from memory. Simultaneously, the PC is incremented by 4 (PC + 4).

The instruction is then decoded. For a conditional branch (e.g., BR\_EQ), the control unit uses the condition reg field to select a specific general-purpose register from the register file. The value of this register is compared against the condition code in the Status Register to determine if the branch should be taken.

Concurrently, the offset field from the instruction is sign-extended to 32 bits and shifted left by 2 (to convert the word offset into a byte offset). This adjusted offset is then added to the PC +4 value to calculate the branch target address.

A key control signal for J-type instructions is Branch, which is asserted when the instruction is a branch. If the branch condition is true (e.g., the register is zero for BR\_EQ), the multiplexer controlling the next PC value selects the calculated branch target address instead of the default PC +4. This new address is loaded into the PC at the end of the cycle, redirecting the program flow. For unconditional jumps (BR\_JMP), the target address is calculated and loaded into the PC without a condition check. Since J-type instructions only alter the control flow, they do not write back to any register, and thus the RegWrite signal remains deasserted (0).

->>

**CONTROL UNIT DESIGN**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Instr class** | **Branch** | **Jump** | **MemRead** | **Mem**  **Write** | **MemToReg** | **ALUSrc** | **RegWrite** | **FP\_En** | **ALU**  **Op** |
| R-type integer ALU | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 10 |
| I-type integer ALU immediate | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 10 |
| Load word | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 00 |
| Load byte | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 00 |
| Store word | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 00 |
| Store byte | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 00 |
| Conditional branch | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 01 |
| Unconditional jump / call / ret | 0 | 1 | 0 | 0 | (CALL: maybe 0) | 0 | (CALL: 1 to write link) | 0 | 10/- |
| Floating-point arithmetic | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 11 |
| FP convert | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 11 |
| I/O read | 0 | 0 | 0\* | 0 | 0 | 0 | 1 | 0 | 10\* |
| I/O write / system | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | - |

**Control Unit**

**Branch**

**Jump**

**MemRead**

**MemWrite**

**MemToReg**

**ALUSrc**

**RegWrite**

**FP\_En**

**ALUOp**

**Data hazards**: RAW (read after write) dependencies between closely scheduled instructions. Common examples: ALU-result used by next instruction, load → use, FP result used by integer or FP instructions.

**Control hazards**: branches and jumps; penalty depends on where branch is resolved (ID or EX) and whether we have prediction.

**Structural hazards**: mostly avoided because we have a separate FPU and integer ALU; still watch for shared ports (register file read/write port limits) and memory port contention.

**Pipeline optimizations used**: short forwarding network (EX/MEM & MEM/WB → EX), small hazard detection to insert single-cycle stalls for load-use, early branch resolution or simple branch prediction, separate FPU pipeline + scoreboard.

**MEMORY HIERARCHY**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Level | Type | Size | Latency | Description |
| Registers | GPR/FP | 16 +8 \* 32-bit | 1 cycle | Faster, holds operands |
| L1 I-Cache | Instruction | 8 KB | 1 cycle | Stores frequently used code |
| L1 D-Cache | Data | 8 KB | 1-2 cycles | Buffers data loads/stores |
| Scratchpad/TCM | On-chip RAM | 16 KB | 1 cycle | Deterministic access for sensor |
| Main Memory(SRAM) | External RAM | 64 MB | 10-50 cycles | General data and program storage |
| Flash | Non-volatile | -- | 100 cycles | Firmware and logging |
| Peripherals | Memory-mapped | -- | Variable | Sensors, communication modules |

**CONCLUSION**

The BCHMS-32 Datapath effectively balances simplicity, performance, and extensibility for low-power health monitoring devices. Through the 5-stage RISC pipeline (IF, ID, EX, MEM, WB), the design achieves predictable timing and efficient instruction throughput while remaining hardware-feasible for embedded contexts.

**Pipeline hazards are mitigated as follows:**  
• **Data hazards** are resolved using forwarding paths (EX/MEM → EX, MEM/WB → EX) and a hazard detection unit that inserts a single-cycle stall on load-use cases.  
• **Control hazards** are minimized with early branch resolution in the ID stage and optional branch prediction for performance-critical applications.  
• **Structural hazards** are avoided by using separate instruction/data memories and distinct integer and floating-point register files.  
• **Floating-point dependencies** are managed via scoreboarding or pipelined FPU forwarding depending on resource constraints.

These optimizations maintain smooth pipeline flow, reduce stall cycles, and ensure real-time signal processing for sensor data. The overall architecture provides a clean and scalable foundation for embedded health-monitoring applications requiring both integer and floating-point computation.

**References**

1. Patterson, D. A., & Hennessy, J. L. (2017). *Computer Organization and Design RISC-V Edition: The Hardware/Software Interface.* Morgan Kaufmann.
2. RISC-V International. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA (Version 20191213).*
3. Stallings, W. (2021). *Computer Organization and Architecture: Designing for Performance (11th Edition).* Pearson.
4. Mano, M. M., & Ciletti, M. D. (2017). *Digital Design: With an Introduction to the Verilog HDL (6th Edition).* Pearson.
5. Hennessy, J. L., & Patterson, D. A. (2019). *Computer Architecture: A Quantitative Approach (6th Edition).* Morgan Kaufmann.
6. Okolo, C. T., et al. (2022). *Responsible AI in Africa—Challenges and Opportunities.* Springer.
7. Abebe, R., et al. (2021). *Narratives and Counternarratives on Data Sharing in Africa.* arXiv:2103.01168.
8. World Health Organization (2021). *Global Strategy on Digital Health 2020–2025.* Geneva: WHO.
9. Binary Cartel (2025). *BCHMS-32 Health Monitoring System: Instruction Set Architecture Documentation.* Internal course project (CS3520, Week 6).