Buena Onda Compute

Processor Architecture Manual

Emmanuel Onyekachukwu Irabor
emmanuel.onyekachukwu.irabor@estudiantat.upc.edu

Jakob Eberhardt
jakob.eberhardt@estudiantat.upc.edu

January 19, 2025

Contents

List of Figures

1 Introduction

In this project, we have designed and implemented a 5-stage RISC-V processor. We further enhanced the performance of our design by adding fully-fledged bypassing logic, a data and instruction cache, and a store buffer.

2 Pipeline

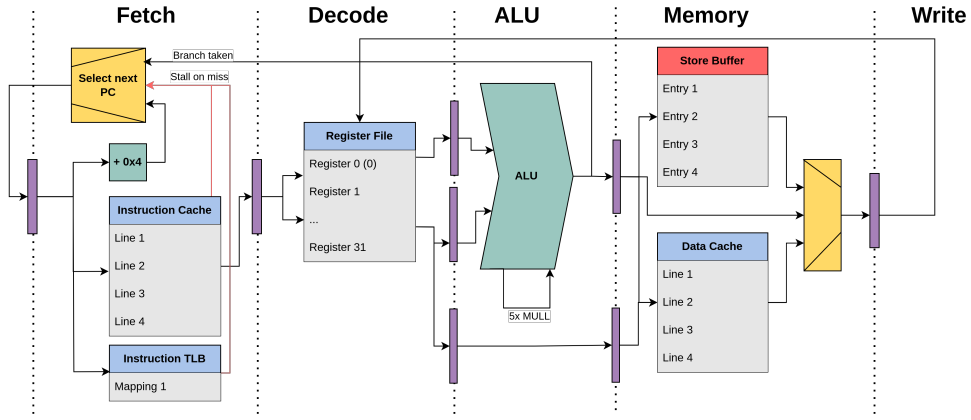

Figure 1: The 5-stage pipeline.

3 Instructions

On `reset`, all structures (iTLB, instruction cache, register file, data cache, store buffer, etc.) are initialized. The processor passes the program counter to the instruction cache. Upon boot, the stall signals of the instruction cache and the instruction TLB are high, hence the fetch stage will initially freeze the PC and insert NOPs into the pipeline until any page fault and cache miss is handled. The behavior of the instructions in the further stages is described in the following sections.

3.1 Encoding

Table 1: RV32I Base Integer Instructions - Subset

Inst Name	FMT	Opcode	funct3	funct7	Description
`add`	R	0110011	0x0	0x00	$rd = rs1 + rs2$
`sub`	R	0110011	0x0	0x20	$rd = rs1 - rs2$
`mul`	R	0110011	0x0	0x01	$rd = rs1 \times rs2$
`addi`	I	0010011	0x0		$rd = rs1 + imm$
`lb`	I	0000011	0x0		$rd = \mathrm{M}[rs1 + imm][0:7]$
`lw`	I	0000011	0x2		$rd = \mathrm{M}[rs1 + imm][0:31]$
`sb`	S	0100011	0x0		$\mathrm{M}[rs1 + imm][0:7] = rs2[0:7]$
`sw`	S	0100011	0x2		$\mathrm{M}[rs1 + imm][0:31] = rs2[0:31]$
`beq`	B	1100011	0x0		`if(rs1 == rs2) PC += imm`
`jalr`	I	1100111	0x0		$rd = \mathrm{PC} + 4;\ \mathrm{PC} = rs1 + imm$

3.2 Load

In this design, every load instruction travels through the five-stage RISC-V pipeline (IF, ID, EX, MEM, WB) and reaches the MEM stage to retrieve data from memory. The instruction is decoded into its destination register and the register holding the memory address. In the ALU stage, the sum of the offset and the base address is calculated. Any variable needed for that calculation can be taken from the register file or bypassed from any other stage or structure that may hold the needed value. The critical components are:

- **CPU Request Signals (`cpu_req`):** For any load instruction (*e.g.*, LW, LB), the pipeline sets `cpu_req.valid` = 1 and `cpu_req.rw` = 0 (read). The 32-bit address, derived from the ALU result, appears on `cpu_req.addr`. Other signals, such as `cpu_req.data`, remain unused (or zero) because we are reading from memory.
- **Store Buffer (SB) Forwarding:** Before going to the cache, the load instruction checks the Store Buffer (SB) to detect if there is a pending store for the same address. If a match is found, the SB forwards the store's data to satisfy the load immediately (bypassing memory). This prevents loading stale data when the pipeline has not yet drained a previous store to the same address.
- **Cache FSM (`dm_cache_fsm`):** If the Store Buffer does not forward data, the cache handles the load. On a cache hit (tag match and valid line), the data is read out to the pipeline. On a miss, a read request is sent to main memory (see `DMemory`) to fetch the needed cache line. During a miss, the pipeline can stall until the data is returned and the cache is updated.
- **Main Memory (`DMemory`):** The memory module introduces a fixed latency (five cycles in this example). Once the memory operation finishes, `mem_data.ready` asserts, and the data is returned to the cache FSM, which updates the cache line and un-stalls the pipeline.
- **Final Value Assembly:** In the MEM stage, the raw 32-bit data (from either SB forwarding or cache) may be post-processed based on load subtype (*e.g.*, sign extension for LB). This processed result (`finalValue`) is placed into the `mem_wb_bus_out` and ultimately written back to the register file in the WB stage.
- **Stalling Logic:** The pipeline stalls a load if the cache FSM is busy (servicing a miss), if the store buffer is forcing a drain, or if the hazard unit detects a load-use hazard (where the subsequent instruction needs the load result too soon).

3.3 Store

Store instructions are decoded into their source register, the register holding the target address and the offset added to it. Stores pass through the same five-stage pipeline but differ in how the data gets committed:

- **CPU Request Signals (`cpu_req`):** On store instructions (SW, SB), `cpu_req.valid` = 1, and `cpu_req.rw` = 1 (write). The store address is the ALU result, placed on `cpu_req.addr`, and the data to write is in `cpu_req.data`. The `wstrb` logic (not part of `cpu_req` itself, but generated in the MEM stage) indicates which bytes must be written.
- **Store Hit & Store Buffer Enqueue:** If the cache signals a hit (*i.e.*, correct tag and valid line), the pipeline marks the store as "complete" from the CPU's viewpoint, avoiding stalls. Instead of writing data to memory immediately, the design enqueues the store into the Store Buffer. This buffer can subsequently "drain" the store into the cache without holding up the pipeline.
- **Store Buffer (SB) Drain Process:** The Store Buffer, implemented as a queue or FIFO, holds the (valdity address, data, `wstrb`) for each store. Whenever the CPU is not issuing a conflicting request or if forced to drain (buffer is full), the SB issues its own request to the cache, marking the cache line as dirty and updating the relevant bytes. When the drain completes (one entry at a time), the SB removes that store from its queue.
- **Cache FSM & Dirty Lines:** When the SB drains a store into the cache, the line is updated and set dirty. If the line must be evicted, the FSM may need to write back the old dirty line to main memory. Only after such write-backs can the new store complete, which will cause pipeline stalls if the SB is full or the cache is busy.
- **Writes to Main Memory:** Actual writes to `DMemory` happen if the FSM must replace a dirty line or if there is a store miss. The cache FSM enters a `write_back` state, issuing a write request to `DMemory`. Once acknowledged, the dirty line is overwritten in memory, and the cache can proceed with the next request.
- **Stalling Conditions:** A store can stall the pipeline if:
  - The store buffer is full (the pipeline must wait for at least one store to drain).
  - The cache FSM is currently handling a miss or write-back.
  - The memory system is not ready (e.g., multi-cycle memory in a busy state).

By seperating store commit from the pipeline's perspective of "completing" a store, the design hides memory latencies behind the Store Buffer. The cache FSM correctly manages dirty lines, making sure that there is coherency between the cache and main memory, and the pipeline can keep issuing instructions without blocking on every store.

3.4 Branch

In this RISC-V pipeline, branches are supported by combining the `IF` stage, which can redirect the PC, with the `ControlUnit` and `EX` stage, which provide the logic to detect whether a branch should be taken. Branch instructions comprise branch-if equal which are encoded into the two registers which will be asserted to be equal and an offset to the current PC to which we will jump. For example, if we execute `beq r0, r0, -8`, we would jump back two instructions relative to the current one.The unconditional jump decodes into a base number which represents the target PC and an offset to which we will jump unconditionally. The key points are:

- **Branch Offset Generation:** In the `ID` stage, the instruction bits relevant for branch offsets are extracted and sign-extended. For a typical `BEQ` instruction, the offset is formed by concatenating and sign-extending various fields from the instruction. This offset is stored in `branch_offset` to be used later by the `IF` stage if the branch is taken.
- **Branch Condition Check:** The `ControlUnit` examines the current instruction's opcode (`BEQ`, `JALR`, *etc.*) and the ALU inputs (`A` and `B`). In the case of `BEQ`, it verifies whether `A == B`. If so, `takebranch_int` is set high, signaling the pipeline that the instruction demands a PC update. For `JALR`, a jump address is computed from `A + imm_i`.
- **Bypassing for Branch Decisions:** The design includes a `BypassUnit` that updates the values of `A` and `B` from later pipeline stages if a branch reads a register that has been recently written. This ensures accurate comparison results without waiting for the register file to be updated.
- **PC Update in the IF Stage:** Once the pipeline determines that a branch is taken (`takebranch = 1`), the `IF` stage updates the `PC` accordingly:

  `if (opcode == JALR)` $\texttt{PC} \leftarrow$ `JalAddr` else $\texttt{PC} \leftarrow (\texttt{PC}-4)+$`branch_offset`.

  If the branch is not taken, the pipeline continues fetching instructions sequentially by incrementing `PC` by 4 each cycle.
- **Pipeline Flush & Control Signals:** When a branch is taken, typically only the instruction following the branch is flushed (*i.e.*, replaced with a NOP in the `IF/ID` register). The signal `takebranch` travels to control logic that injects a bubble (or NOP) to discard the instruction already fetched. The rest of the pipeline proceeds normally.

3.5 Arithmetic

Arithmetic instruction typically encodes one destination register and two source registers. The source values can be taken from the register file or by using a bypass if one or both values are already present in another stage or structure, for example, the store buffer. Register `r0` is hardwired to the value zero.This RISC-V pipeline design handles arithmetic operations in a dedicated Arithmetic Logic Unit (ALU). Below are the key points:

- **Instruction Types (Opcodes & Funct Fields):**
  - `ALUopI` instructions (e.g., ADDI) take one register operand and an immediate.
  - `ALUopR` instructions (e.g., ADD, SUB, MUL) perform register-to-register arithmetic.
  - `LW` / `SW` are also partially arithmetic, since they compute an address via `A + B`.

  The ALU decodes its behavior using `IDEXop`, `IDEXfunct3`, and `IDEXfunct7` to distinguish among ADD, SUB, AND, OR, MUL, etc.

- **Single and Multi Cycle ALU Logic:**
  - `ADD / SUB:` The ALU selects between `A + B` and `A - B` based on `funct7`.
  - `MUL:` In this pipeline, a dedicated multiplier may stall the pipeline while it computes the product.

  The resulting 32-bit output is written to `EXMEMALUOut`.

- **Immediate Generation & ALU Input Selection:** For I-type and S-type instructions, the immediate is sign-extended and substituted for `Bin`. The `ALUInputSelect` module decides whether `Bin` is a register value, a sign-extended immediate, or the result of a bypass (e.g., from MEM/WB). This ensures arithmetic operands are properly fed into the ALU each cycle.

- **Pipeline Integration:**
  - **ID Stage:** Decodes whether the instruction is `ALUopR` or `ALUopI`, extracts registers and immediate.
  - **EX Stage:** The `ALU` uses these operands for addition, subtraction, multiplication, *etc.* The result is placed into `EXMEMALUOut`, flowing onward to the MEM and WB stages.
  - **Stalls for MUL:** If a multiply (`MUL`) is detected, the design can stall multiple cycles via an internal counter in the `EX` stage. This allows a multi-cycle multiplication without holding up other non-MUL instructions unnecessarily.

3.6 Exceptions

In this RISC-V pipeline design, exceptions are primarily detected in the `HazardUnit` and are then propagated through the pipeline, causing a partial flush or PC redirection to an exception handler. The main exception cases handled in the provided modules are:

- **Unaligned Memory Access:** If the instruction is a load (`LW`) or store (`SW`) that is not a byte-level operation (i.e., does not have `SB_FUNCT3` or `LB_FUNCT3`) and the least significant bits of the computed address (`alu_result[1:0]`) are non-zero, an *unaligned access* exception is set.
- **Divide-by-Zero:** For integer divide instructions (detected as `ALUopR` with `funct3 = DIV`), if the source register `decodeB` is zero, the `HazardUnit` raises a `DIVIDE_BY_ZERO` exception
- **No Exception Default:** When neither of the above conditions is met, the `HazardUnit` assigns `NO_EXCEPTION` to `excpt_out`, indicating normal operation.

Exception Propagation and Handling

- **HazardUnit to Pipeline:** After detection, `excpt_out` travels alongside the pipeline buses. If the exception code is non-zero, it is visible to subsequent pipeline stages, such as the IF stage and the `PipelineRegs` module. The Instruction address causing the exception is also saved.
- **PC Redirection in IF:** The `IF` stage checks the `excpt_in` signal (which mirrors `excpt_out` from the `HazardUnit`) on every clock cycle. If `excpt_in` is asserted (*i.e.*, non-zero), the PC is redirected to `EXCPT_ADDR`, ensuring the CPU fetches from the exception handler address rather than continuing with normal instruction flow:

$$\texttt{if (excpt\_in)} \longrightarrow \texttt{PC} \leftarrow \texttt{EXCPT\_ADDR}.$$

- **Pipeline Register Flush:** In `PipelineRegs`, if `excpt_in` is asserted at the pipeline register update, relevant stages are flushed by inserting NOPs. This prevents corrupted instructions from continuing through the pipeline. Typically, the `IF/ID`, `ID/EX`, and `EX/MEM` pipeline registers are cleared, while the CPU transitions to the exception handler flow.

4 Performance

The instruction cache can hold up to sixteen instructions with a FIFO policy. Hence, the processor works most efficiently with loops consisting of up to sixteen instructions. The store buffer drains pending stores during stalls caused by other parts of the pipeline. For example, on a instruction cache miss, we have to insert five NOPs into the pipeline during which the store buffer could drain. The design features virtual instruction memory which limits fragmentation.

A Pipeline Overview

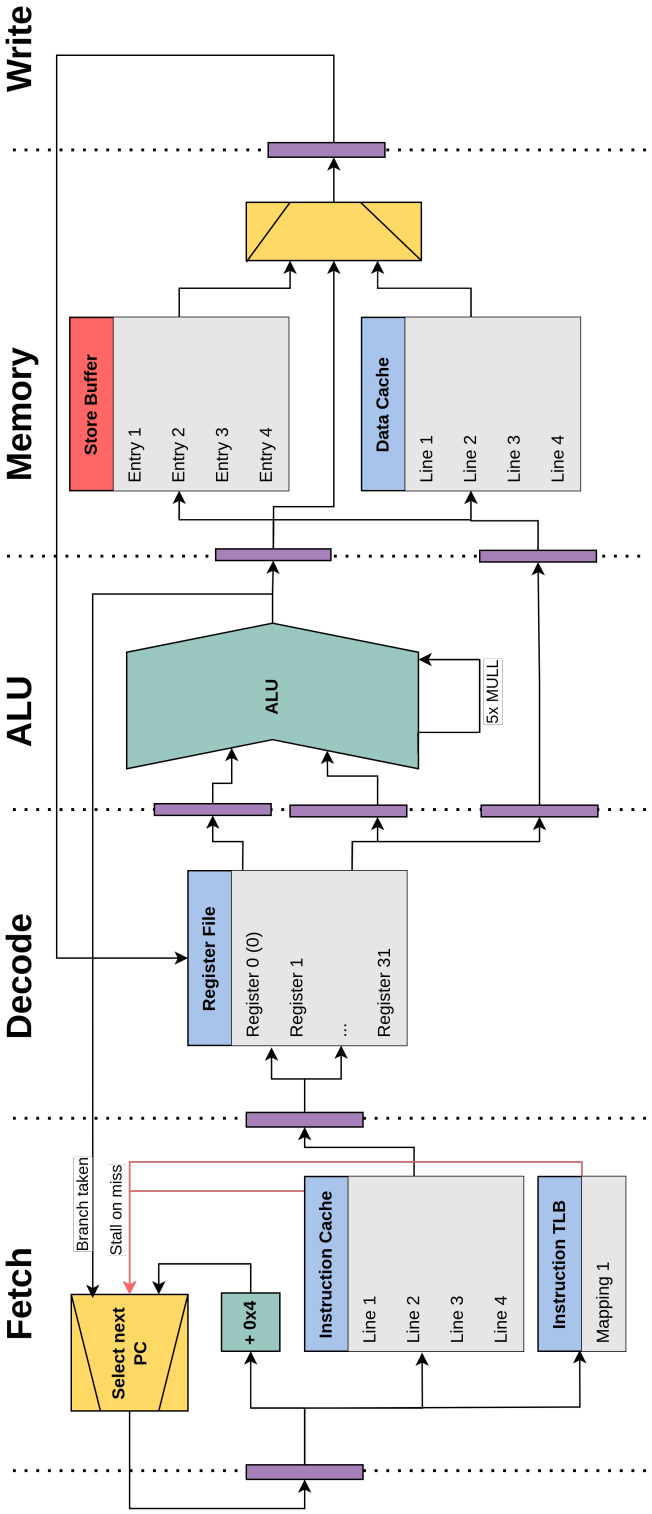

Figure 2: The 5-stage pipeline.