Project Mario

Rehaaan Das



Started 28/7/2025

Ended 20/9/2025

# Index

* Mission Objective
* Architecture
  1. **Specifications**
  2. **Usage**
  3. **ISA**
* Microarchitecture
  1. **The Pipeline**
  2. **Pipeline Data Paths**
  3. **Pipeline Controller**
  4. **Hazard Management**
  5. **Everything Else**
* Sample Programs
  1. **The Fibonacci Sequence**
  2. **The Call Stack**
  3. **Calculator**

# Mission Objective

To build a pipelined CPU capable of running programs in SystemVerilog and to learn about the application, implementation, and challenges of temporal parallelism in CPUs. Admittedly, there are a few flaws, but who ever gets it right the first time?

Chapter One

# Architecture

## Specifications

* 16 Bit Words
* Unsigned Integers
* 32 Bit Instructions
* 16 Bit Instruction Memory Addresses

= 65,535 Lines of assembly

* 10 Bit Data Memory Addresses

= 1024 Addresses for data storage

* Harvard Architecture Memory
* 16 General Purpose Registers
* RISC-V Inspired ISA
* 5 Stage Pipeline with Late Operand Fetch
* 100MHz with an Fmax of 125MHz

## Usage

This CPU does not have any internal program memory, and such also does not feature any sort of program image loading. Which means you have to provide it program memory externally like so:

SystemVerilog

1 logic [31:0] mem [0:65535];

2

3 CPU CPU(.IAddr(Address), .IMemDOut(mem[Address]));

4 initial begin

5 mem[0] = 32'0011 ... 1010;

6 mem[1] = 32’1000 ... 0110;

7 ...

8 #1000;

9 end

The CPU outputs its instruction memory read address from the .IAddr() port and you are to provide it instructions through the .IMemDOut() port.

Note that the CPU does have its own data memory so don’t get any ideas.

## ISA

First 4 bits of instructions are opcodes. They are mentioned before the name of the instruction in bold.

### **0000** MEM

* Functionality

Writes to memory a value specified in operands.

* Bit Fields

0000 0 [ Data 16 bits ] [ Addr 10 bits ] **X**

If bit [27] is 0, address is given in instruction at bits [10:1].

0000 1 [ Data 16 bits ] [ Reg 4 bits ] **XXXXXXX**

If bit [27] is 1, address is given in the register given in instruction at bits [10:7].

i.e. Whatever value stored in register given in instruction will be used as address.

### **1111** REG

* Functionality

Writes to register a value specified in operands.

* Bit Fields

1111 **X** [ Data 16 bits ] [ Reg 4 bits ] **XXXXXXX**

Writes to register at register number given in bits [10:7].

### **0001** LD

* Functionality

Loads value from memory into registers.

* Bit Fields

0001 0 [ Dest Reg 4 bits ] [ Src Addr 10 bits ] **XXXXXXXXXX**

If bit [27] is 0, data at memory address given in bits [22:13] is used.

0001 1 [ Dest Reg 4 bits ] [ Src Reg 4 bits ] **XXXXXXXXXXXXXXXXXXX**

If bit [27] is 1, data from memory address stored in register given in bits [22:13].

### **0010** STR

* Functionality

Stores value from registers to memory.

* Bit Fields

0010 0 [ Src Reg 4 bits ] [ Dest Addr 10 bits ] **XXXXXXXXXX**

If bit [27] is 0, data is stored at memory address given in bits [22:13].

0010 1 [ Src Reg 4 bits ] [ Dest Reg 4 bits ] **XXXXXXXXXXXXXXXXXXX**

If bit [27] is 1, data is stored at memory address stored in register given in bits [22:19].

### **0100** MOV

* Functionality

Moves values between registers.

* Bit Fields

0100 **X** [ Src Reg 4 bits ] [ Dest Reg 4 bits ] **XXXXXXXXXXXXXXXXXXX**

Moves data from register given in bits [26:23] to register given in bits [22:19]

### **1000** ALU

* Functionality

Conducts arithmetic operations on registers and stores them in another register.

* Bit Fields

1000 0 [ Accumulator Reg 4 bits ]

[ Operand Reg 4 bits ] [ Dest Reg 4 bits ] **XXXXXXXXXXXXXXX**

Conducts operation specified in bit [27] with registers given in bits [26:23] and [22:19] as the accumulator and operand respectively, and stores it in register given in bits [18:15].

If bit [27] is 0, operation is addition. Otherwise, it is subtraction.

### **1110** JMP

* Functionality

Changes value of program counter (PC) to execute the instructions at a certain address, based on certain conditions.

* Bit Fields

1110 0 [ Condition 2 bits ] [ Address 16 bits ] **XXXXXXXXX**

If bit [27] is 0, address given in bits [24:9] is used as jump destination.

1110 1 [ Condition 2 bits ] [ Reg 4 bits] **XXXXXXXXXXXXXXXXXXXXX**

If bit [27] is 1, address stored in register given in bits [24:21] is used as jump destination.

Jump will only be carried out if condition given in bits [26:25] is valid. The conditions are:

00 – Unconditional. Jump will always be executed regardless of CPU state.

01 – Zero flag is HIGH. If the result of an arithmetic operation was zero, the condition is met.

10 – Negative flag is HIGH. If the result of an arithmetic operation was negative, the condition is met. Note that this CPU uses unsigned integers, which means the output of the subtraction operation wherein operand is greater than accumulator is meaningless.

11 – Overflow. If the result of an arithmetic operation was greater than 65,535, the condition is met.

### **0111** CMP

* Functionality

Conducts arithmetic operations on registers but does not store them anywhere.

* Bit Fields

1000 0 [ Accumulator Reg 4 bits ]

[ Operand Reg 4 bits ] **XXXXXXXXXXXXXXXXXXX**

Has the exact same functionality as ALU operations, except that result is not stored in any register. Hence the lack of a destination field,

### **1011** HALT

* Functionality

Stops CPU from executing any further instructions.

* Bit Fields

1011 **XXXXXXXXXXXXXXXXXXXXXXXXXXXX**

### **1101** NOP

* Functionality

Does not do anything. Used to idle the CPU.

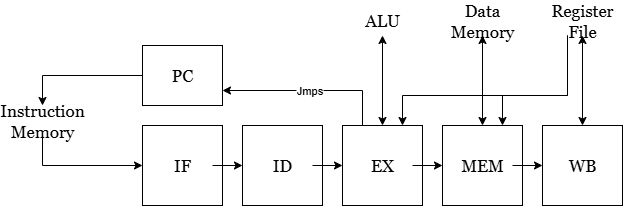
* Bit Fields

1101 **XXXXXXXXXXXXXXXXXXXXXXXXXXXX**

Chapter Two

# Microarchitecture

## The Pipeline



A top-level overview of the pipeline.

The pipeline is a standard 5 stage RISC pipeline, with the first two stages responsible for fetching and decoding, so that the remaining 3 can carry out delegation to the various units of the CPU.

In between every stage after ID is a register that carries the payload. Only the register between IF and ID carries not the payload but the raw instruction from memory.

### Payload

The payload is a decoded 50 bit form of the 32 bit instruction and serves the purpose of making computation for the stages that are concerned with the instructions easier.

The payload is structured in separate fields like so:

|  |  |  |
| --- | --- | --- |
| 8 | Optype | [49:42] |
| +16 | Data | [41:26] |
| +10 | Addr | [25:16] |
| +4 | Operand A | [15:12] |
| +4 | Operand B | [11:8] |
| +4 | ALU Destination Reg | [7:4] |
| +2 | Jump Type | [3:2] |
| +1 | Dynamic | [1] |
| +1 | ALU WE | [0] |
| = | 50 bits | |

Note that the operands are in the form of register token, which is then used by the stages to read the value themselves, given this CPU uses late operand fetch.

The optypes are in one-hot encoding, again, to reduce computation to a minimum, and are structured like so:

|  |  |
| --- | --- |
| 1000 0000 | Mem |
| 0100 0000 | Store |
| 0010 0000 | Load |
| 0001 0000 | Reg |
| 0000 1000 | Move |
| 0000 0100 | Add |
| 0000 0010 | Sub |
| 0000 0001 | Jmp |

### IF

Not a functional stage, but rather the stage in which 32 bit instructions fetched from instruction memory enter the pipeline.

### ID

Takes 32 bit instruction from IF and decodes it into the 50 bit payload. Every subsequent stage takes in the decoded payload as input.

### EX

Responsible for computational instructions, namely JMP, ALU, and CMP. Delegates commands to the ALU for arithmetics and program counter (PC) for changing the flow of the program.

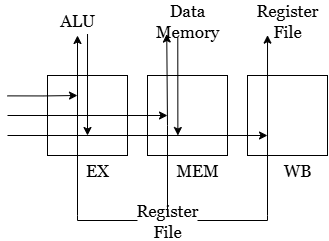
### MEM

Responsible for instructions reading from or writing to memory, namely MEM, STR, and LD.

### WB

Responsible for instructions writing to the register file, namely REG and MOV.

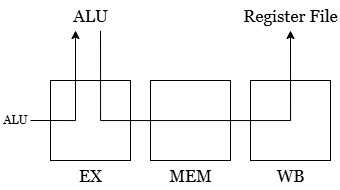
## Pipeline Data Paths



Pipeline Datapaths

The CPU, as aforementioned, uses late operand fetch, meaning each stage fetches their own data from the register file. This is made possible through 5 register read ports allocated like so:

|  |  |
| --- | --- |
| Stage | Port |
| EX | AOut |
| BOut |
| MEM | COut |
| DOut |
| WB | EOut |



Data path of an ALU instruction

Payloads that concern stages other than WB and need to store their results in registers, namely ALU and STR, are converted into register writes with the result as data to be written, and are carried down the pipeline to the WB stage as usual to be written. In the given image, an operation is given to the ALU to perform, whose output is given back to EX which then generates a register write payload further carried till the WB stage to be written. This register write generation is performed with load instructions as well.

## Pipeline Controller

The pipeline controller orchestrates the movement of data within the pipeline. Every clock edge, the payload from each stage progresses to the next stage, moving the pipeline forward. However, the pipeline controller also handles certain special cases.

### Halts

The signal for halting the CPU is given by the EX stage. This signal prevents the EX stage from progressing, causing the halt signal to stay up forever, and no new instructions be able to get past EX. The rest of the stages are allowed to progress and exit the pipeline, letting the last instructions complete their tasks.

### Stalls

During a load instruction, the stages before MEM need to be stalled, that is, not be progressed further in the pipeline just for that clock cycle to allow the register write part of the load instruction to complete. To do this, stages before MEM are stalled, and a NOP (A bubble) is inserted in the MEM stage as the previous stages are not being progressed to fill the gap.

### Jumps

Pipeline Code

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| IF | ID | EX | MEM | WB |
| JMP |  |  |  |  |
| ADD | JMP |  |  |  |
| SUB | ADD | JMP |  |  |

1 JMP

2 ADD

3 SUB

4 ...

5 REG

6 ...

Notice how the stray

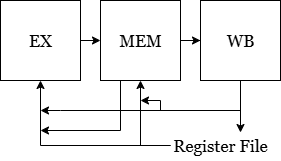
instructions in red are turned

into NOPS.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| REG | NOP | NOP | JMP |  |

In the time that it takes a JMP instruction to reach the EX stage where it can be executed, 2 instructions that were ahead of the JMP have already entered the IF and ID stages. This is behaviour we do not want as the only instructions in the pipeline after JMP should be at the destination of the JMP. This is called a Control Hazard and this CPU rectifies this by bubbling the 2 instructions. It overwrites the two instructions with a NOP instruction to make sure the stray instructions do not get executed.

## Hazard Management



Bypass Valves in the pipeline

Register write instructions that haven’t yet reached the WB stage or are in the WB stage but haven’t been written yet. This is a Read-After-Write (RAW) hazard and is an absolutely dogwater thing to happen. This CPU solves this by bypassing, which is when you replace the data a register file gives you when you read from it, with the data that’s supposed to be there but isnt there yet.

The hierarchy of bypassing in this CPU prioritizes newer instructions over older instructions, as newer instructions writing to the same register would overwrite what older instructions wrote. E.g if EX is trying to read from register 7, and the two stages after that, namely MEM and WB have register writes to register 7, whatever MEM is writing is what is used for bypass instead of WB.

Aside from bypassing, the CPU also utilizes bubbling and stalling, both of which are explained in the previous section; *Pipeline Controller*.

## Everything Else

Aside from the pipeline, the rest of the CPU is pretty garden variety

Chapter Three

# Sample Programs

## The Fibonacci Sequence

Assembly

0x0000 Reg r1 0001

0x0001 Add r1, r1 r2 ;Store result at r1

0x0002 Sub r2, r1 r2 ;Store result at r2

0x0003 Jmp 0001

Binary

0x0000 11110000000000000000100010000000

0x0001 10000000100100001000000000000000

0x0002 10001000100100010000000000000000

0x0003 11100000000000000000001000000000

## The Call Stack

Lets use a downward growing stack and R15 as stack pointer (SP).

Assembly

0x0000 Reg r15 03FF ;Init stack pointer.

0x0001 Reg r1 0002

0x0002 Reg r2 0003

0x0003 Mem [r15] 0005 ;Dynamic addr write

0x0004 Jmp 0026 ;Call function

0x0005 Halt

...

0x0026 Add r3, r1 r2 ;Store result at r3

0x0027 Jmpr [r15] ;Return

Binary

0x0000 11110111111111111111111110000000

0x0001 11110000000000000001000010000000

0x0002 11110000000000000001100100000000

0x0003 00001000000000000010111110000000

0x0004 11100000000000000011010000000000

0x0005 10110000000000000000000000000000

...

0x0026 10000000100100011000000000000000

0x0027 11101001111000000000000000000000

I did not include the stack pointer increment/decrement thing because its redundant when the depth of the stack frame is one. And yes calling a function to perform addition is absolutely ridiculous.

## Calculator

Assembly

0x0000 Reg r1 0007

0x0001 Reg r2 0003 ;Set up operands

0x0002 Reg r3 0001 ;Set up operation

0x0003 Cmp Add r3 r3

0x0004 Jmpz 0007 ;add if r3 is 0

0x0005 Sub r4, r1 r2 ;sub if r3 is 1

0x0006 Jmp 0008

0x0007 Add r4, r1 r2

0x0008 Halt

Binary

0x0000 11110000000000000011100010000000

0x0001 11110000000000000001100100000000

0x0002 11110000000000000000100110000000

0x0003 01110001100110000000000000000000

0x0004 11100010000000000000111000000000

0x0005 10001000100100100000000000000000

0x0006 11100000000000000001000000000000

0x0007 10000000100100100000000000000000

0x0008 10110000000000000000000000000000