Project Documentation

## Design Specifications

Word Size: 16 bits

Memory Size: 64 kilobytes

Memory Blocks: 8

Block Size: 8 kilobytes

Cache Size: 40,000 bits

Addressing Mode: Immediate

## Implementations

Operations: Add, halt, load, store, clear, skip, jump, subtract, and, or, not, jal, return

Datapath: Pipelined. 4 stages (Fetch, Decode, Execute & Memory, Writeback)

## Design Explanation

The typical CPU architecture utilizes a 5-stage pipelined flow. For our implementation of the accumulator architecture, the execution stage, typically reserved for memory calculations, was deemed unnecessary and therefore reduced into the memory stage, cutting our pipeline down to 4 stages. The reasoning behind this was simple: the assembly language supported will reference data only by its memory address. By also using immediate addressing, we forgo the need to compute memory addresses to fetch our data.

Here is a breakdown of our 4-stage pipeline:

### Fetch (F):

Fetches instruction from memory at PC address. Instructions are stored starting at address 0. The Fetch stage is the initial phase of the instruction cycle, responsible for fetching the instruction from memory. In this stage, the CPU\_fetch module is utilized, taking inputs such as clk (clock signal), rst (reset signal), halt\_program (signal to halt the program), Program Counter (PC), Memory (MEMORY), and Instruction Register (IR). The PC represents the memory address of the next instruction to be fetched. During the Fetch stage, the CPU\_fetch module fetches the instruction from the memory location specified by the Program Counter (PC) and stores it in the Instruction Register (IR). Instructions in the assembly language are typically stored starting at address 0, and the PC is incremented to point to the next instruction in memory. The fetched instruction is then passed to the Decode stage for further processing. This stage ensures a smooth flow of instructions through the pipeline, allowing the subsequent stages to operate on the correct instruction in a coordinated manner. The simplicity of the Fetch stage in the accumulator architecture is a result of the decision to reference data only by its memory address, which simplifies the instruction fetch process. The relative latency for fetch would be 23.25 ns, since fetch can either use SRAM or DRAM, then it will access the register file, and finally it will also use the basic logic gate since it increments PC. So therefore, if you add all of those together, it will end up as 23.25 ns latency.

### Decode (D):

The decode stage will interpret the instruction that is fetched from memory, and it will extract the opcode and operand values. In the decode stage, the CPU\_decoder module will take the instruction (IR) as an input, and it will output the opcode (OP\_code) and operand value (value). The opcode will be essential for determining the operation to be performed in the subsequent stages, while the operand values will represent the data, or the address involved in the operation. Decode only accesses the register file and it doesn’t utilize SRAM, DRAM, or Basic Logic Gate, therefore the relative latency is 1 ns.

### Execute & Memory (M):

The Execute and Memory stage is responsible for executing arithmetic or logic operations and handling memory-related tasks. This stage combines whatever is traditionally separated into execute and memory stages in a standard 5 stage pipeline. The CPU\_execute\_memory module will take the opcode (OP\_code) and operand value (value) as inputs, along with other control signals such as clk, rst, and halt\_program. This stage may involve arithmetic operations which utilize the Accumulator (AC) and memory operations using the Memory Buffer Register (MBR) and Memory Address Register (MAR). The data\_out signal is prepared in execute and memory to transition the memory output to the next stage. Execute & Memory uses 23.25 since it accesses memory from the same place it got fetched from, and uses the register, such as the accumulator. Execute and Memory also use basic logic gates since this stage involves arithmetic or logic gates.

### Writeback (W):

The Writeback stage is responsible for updating the CPU's state based on the results obtained from the previous stages. In this stage, the CPU\_writeback module takes the Memory Address Register (MAR), data\_out, and other control signals as inputs. The data\_out signal carries the result from the Execute & Memory stage. The Writeback stage updates the Accumulator (AC) with the result obtained from the previous stage. This stage completes the instruction cycle, and the updated state is used in subsequent cycles. Writeback’s relative latency is 1.25 since it accesses registers, and it also uses logic gates to update the accumulator by using load word, which is addition.

### Test Bench (Fibonacci.v):

The testbench is designed to evaluate the functionality of the Fibonacci sequence computation implemented in the CPU module. We first made a tabular representation of the addresses, instructions and values stored (Fibonacci.txt). We then created the Fibonacci.v file using the text file as a guide, initializing the memory values with instructions.

The initial block initializes the memory with instructions corresponding to the Fibonacci sequence computation. Each instruction is represented in hexadecimal format (16 bits) and corresponds to a specific operation in the Fibonacci computation. Time delays (#25) are used between instructions to simulate the clock cycles and allow for proper execution. Users can observe the simulated results in the display format to verify the correctness of the Fibonacci sequence computation. Run the simulation using the preferred Verilog simulator, following the appropriate commands or procedures.

Below is a time graph of the pipeline and variable inputs and outputs:

Datapath Pipeline

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Cycles | | | | | | | | |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Load | F | D | M | W |  |  |  |  |  |
| Add |  |  | F | D | M | W |  |  |  |
| Store |  |  |  |  | F | D | M | W |  |
|  | | | | | | | | | |
|  | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| Load | F | D | M | W |  |  |  |  |  |
| Add |  |  | F | D | M | W |  |  |  |
| Store |  |  |  |  | F | D | M | W |  |
| Load |  |  |  |  |  |  |  |  |  |
|  | | | | | | | | | |
|  | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 |
| Store | F | D | M | W |  |  |  |  |  |
| Load |  |  | F | D | M | W |  |  |  |
| Add |  |  |  |  | F | D | M | W |  |
| Store |  |  |  |  |  |  |  |  |  |
|  | | | | | | | | | |
|  | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 |
| Skip | F | D | M | W |  |  |  |  |  |
| Jump |  |  | F | D | M | W |  |  |  |
| Halt |  |  |  |  | F | D | M | W |  |

Variables Timewave

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 25 | 50 | 75 | 100 | 125 | 150 | 175 | 200 |
| clk | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| rst | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| AC | 0000x | 0000x | 0000x | 0000x | 0000x | 0000x | 0000x | 0000x | 0000x |
| MAR | 0000x | 0000x | 0000x | 0024x | 0024x | 0020x | 0020x | 0024x | 0024x |
| MBR | 0000x | 0000x | 0000x | 0000x | 0000x | 0000x | 0000x | 0000x | 0000x |
| IR | 0000x | 0324x | 0324x | 0120x | 0120x | 0424x | 0424x | 0322x | 0322x |
| PC | 0000x | 0001x | 0001x | 0002x | 0002x | 0003x | 0003x | 0004x | 0004x |
| OP\_code | 00x | 00x | 03x | 03x | 01x | 01x | 04x | 04x | 03x |
| value | 00x | 00x | 24x | 24x | 20x | 20x | 24x | 24x | 22x |
| halt\_program | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| data\_out | 0000x | 0000x | 0000x | 0000x | 0000x | 0000x | 0000x | 0000x | 0000x |
|  | | | | | | | | | |
|  | 225 | 250 | 275 | 300 | 325 | 350 | 375 | 400 | 425 |
| clk | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| rst | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| AC | 0001x | 0001x | 0002x | 0002x | 0003x | 0003x | 0005x | 0005x | 0008x |
| MAR | 0000x | 0000x | 0000x | 0024x | 0024x | 0020x | 0020x | 0024x | 0024x |
| MBR | 0000x | 0004x | 0004x | 0022x | 0022x | 0005x | 0005x | 0008x | 0008x |
| IR | 0000x | 0324x | 0324x | 0120x | 0120x | 0424x | 0424x | 0322x | 0322x |
| PC | 0005x | 0005x | 0006x | 0006x | 0007x | 0007x | 0008x | 0008x | 0009x |
| OP\_code | 00x | 00x | 03x | 03x | 01x | 01x | 04x | 04x | 03x |
| value | 00x | 00x | 24x | 24x | 20x | 20x | 24x | 24x | 22x |
| halt\_program | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| data\_out | 0001x | 0001x | 0002x | 0002x | 0003x | 0003x | 0005x | 0005x | 0008x |
|  | | | | | | | | | |
|  | 450 | 475 | 500 | 525 | 550 | 575 | 600 | 625 | 650 |
| clk | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| rst | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| AC | 0008x | 000dx | 000dx | 0015x | 0015x | 0022x | 0022x | 0037x | 0037x |
| MAR | 0000x | 0000x | 0000x | 0024x | 0024x | 0020x | 0020x | 0024x | 0024x |
| MBR | 0000x | 0004x | 0004x | 0022x | 0022x | 0005x | 0005x | 0008x | 0008x |
| IR | 0000x | 0324x | 0324x | 0120x | 0120x | 0424x | 0424x | 0322x | 0322x |
| PC | 0009x | 000ax | 000ax | 000bx | 000bx | 000cx | 000cx | 000dx | 000dx |
| OP\_code | 00x | 00x | 03x | 03x | 01x | 01x | 04x | 04x | 03x |
| value | 00x | 00x | 24x | 24x | 20x | 20x | 24x | 24x | 22x |
| halt\_program | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| data\_out | 0008x | 000dx | 000dx | 0015x | 0015x | 0022x | 0022x | 0037x | 0037x |
|  | | | | | | | | | |
|  | 4400 | 4425 | 4450 | 4475 | 4500 | 4525 | 4550 | 4600 |
| clk | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| rst | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| AC | 0059x | 0059x | 0090x | 0090x | 00E9x | 00E9x | 0179x | 0179x |
| MAR | 0000x | 0000x | 0000x | 0024x | 0024x | 0020x | 0020x | 0024x |
| MBR | 0000x | 0004x | 0004x | 0022x | 0022x | 0005x | 0005x | 0008x |
| IR | 0000x | 0324x | 0324x | 0120x | 0120x | 0424x | 0424x | 0322x |
| PC | 000ax | 000ax | 000bx | 000bx | 000cx | 000cx | 000dx | 000dx |
| OP\_code | 00x | 00x | 03x | 03x | 01x | 01x | 04x | 04x |
| value | 00x | 00x | 24x | 24x | 20x | 20x | 24x | 24x |
| halt\_program | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| data\_out | 0059x | 0059x | 0090x | 0090x | 00E9x | 00E9x | 0179x | 0179x |

## Contributions:

Justin Chen

-Designed and coded the CPU in Verilog.

-Organized and distributed tasks amongst members.

-Documentation.

-Graphing for Datapath & Timewave.

Nico Wang

-Contributed to operational implementation

-Worked on documentation tasks.

-Provided ideas for the ISU (Instruction Set Architecture) and layout.

-Implemented the testbench.

-Documented the overall process.

Suyash

-Documentation

-Helped code parts of the CPU

Daien Miao

-Benchmarked the program in assembly

-Translated benchmark program to machine code

-double checked code for errors

Edward

-Double checked code for errors

-Documentation

-Contributed to operational implementation

## Resources:

<https://www.chipverify.com/verilog/verilog-testbench-simulation>

<https://edaplayground.com/>

<https://rensselaer.webex.com/recordingservice/sites/rensselaer/recording/b26f496f751d103cb7fe22c98a2a5d7a/playback>

<https://srimanthtenneti.medium.com/accumulator-based-cpu-design-ec07171f3e9a>

<https://scholarworks.calstate.edu/downloads/rf55zc46k>

<https://www.youtube.com/watch?v=EW9vtuthFJY>

<https://www.spiceworks.com/tech/tech-general/articles/what-is-computer-architecture/>