**Design Document Phase2**

**Functional Simulator for Subset of RISC-V instruction set**

**Input/Output**

## **Input**

Input to the simulator is a MEM file that contains the encoded instruction and the corresponding address at which instruction is supposed to be stored, separated by space. These machine instructions follow the Little Endian notation. For example:

* 0x0 0x0A20A0E3
* 0x4 0x0230A0E3
* 0x8 0x031082E0

Additionally, the MEM file also contains the data segment, which is distinguished from the instruction segment on the basis of the address values. Non-zero data values are stored bytewise preceded by the corresponding address at which they are supposed to be stored and separated by space.

## **Functional behaviour and Output**

The following is the list of values displayed on our terminal:

* Total number of cycles
* Total instructions executed
* CPI
* Number of Data-transfer (load and store) instructions executed
* Number of ALU instructions executed
* Number of Control instructions executed
* Number of stalls/bubbles in the pipeline
* Number of data hazards
* Number of control hazards
* Number of branch mispredictions
* Number of stalls due to data hazards
* Number of stalls due to control hazards

The simulator reads the instructions from the MEM file, converts them into Big Endian(for ease of processing), and stores the **“address-instruction”** pair as a key-value pair in the **instructionSegment**. Similarly, the data is read from the MEM file and stored in **dataSegment** as key-value pairs.

## **Knob set**

1. In order to enable pipelining, set **Knob1**. If Knob1 is not set, it would work as a single execution design as implemented in Phase 1.

2. If **Knob2** is set, data forwarding is enabled else the pipeline is expected to work with stalling.

3. If **Knob3** is set, values in the register file are printed at the end of each cycle else they are not printed.

4. If **Knob4** is set, information in the pipeline registers at the end of each cycle (similar to tracing), along with cycle numbers are printed else they are not.

5. **Knob5** is like enabling Knob4 for a specific instruction. With this feature we will be able to see the pipeline registers information for a particular instruction of our interest. Here, the instruction can be specified as a number (example, if the instruction we are interested in is the 10th instruction in the input program, 10 will be taken as input).

## **Hazard Detection Unit**

1. **hazardDetectionUnitPrevPrev():**

If there is a data dependency between the current instruction and the previous to previous one, then we check which type of forwarding is there-> **M to E/E to E** and accordingly we set the values of muxes

* MuxASelect
* MuxRMSelect
* MuxBSelect

For eg:

1. E to E forwarding:

# Previous is arithmetic and current is store

# add x14, x11, x16 - x14 is available after execute

# sw x11, 0(x14) - x14 is required before execute

# E to E forwarding

1. M to E forwarding

# Both are arithmetic

# add x11, x12, x13

# addi x0 x0 0

# add x14, x11, x16

# M to E forwarding

1. **hazardDetectionUnitPrev():**

If there is a data dependency between the current instruction and the previous one, then we check which type of forwarding is there->  **E to E/M to E with stalling/M to M forwarding** and accordingly we set the of muxes:

* MuxASelect
* MuxRMSelect
* MuxBSelect
* MuxMSelect

Moreover, if there is stalling then we set isDataStall and assign the corresponding value to dataStallCount depending on the stage in which stalling is required.

For eg:

1. M to E with stalling

# Previous is load and current is arithmetic

# lw x11, 0(x12) x11 is available after memory in RY

# add x14, x11, x16 - x11 is required in execute

# M to E forwarding, with stalling

1. E to E

# Previous is arithmetic and current is store

# add x14, x11, x16 - x14 is available after execute

# sw x11, 0(x14) - x14 is required before execute

# E to E forwarding

1. M to M forwarding

# Previous is load and current is store

# lw x11, 0(x12) x11 is available after memory in RY

# sw x11, 0(x14) - x11 is required in memory

# M to M forwarding

1. **hazardDetectionUnitControlPrevPrev():**

If there is a data dependency between the current control instruction and the previous to previous one, then we check which type of forwarding is there-> **E to D with no stall/M to D with one stall** and accordingly we set the values of muxes:

* MuxRASelect
* MuxRBSelect

1. **hazardDetectionUnitControlPrev():**

If there is a data dependency between the current control instruction the previous one, then we check which type of forwarding is there-> **E to D with one stall/M to D with two stalls** and accordingly we set the values of muxes:

* MuxRASelect
* MuxRBSelect

## **decodeBranchResolution**

* This function is called only for branch instructions. Depending on the value of ALUop (23, 24, 25 or 26), we check whether the branch is taken or not.
* If the branch is taken, then we check whether the PC is already present in the BTB( Branch Target Buffer) or not.
* If it is not present, then we subsequently store it in the BTB otherwise depending whether our prediction is hit or a miss, we update the value of the PC accordingly.
* If it was a miss, then the previous instruction was flushed.

## **bufferDecodeExecute()**

We declared the following variables:

pppMuxYSelect ppMuxYSelect, pMuxYSelect, ppstoreType, pstoreType, nALUop, prevPrevPrevRD, prevPrevRD, prevRD, pppwriteRegisterFile, ppwriteRegisterFile, pwriteRegisterFile, pploadType, ploadType, ppmemRead, pmemRead, ppmemWrite, pmemWrite

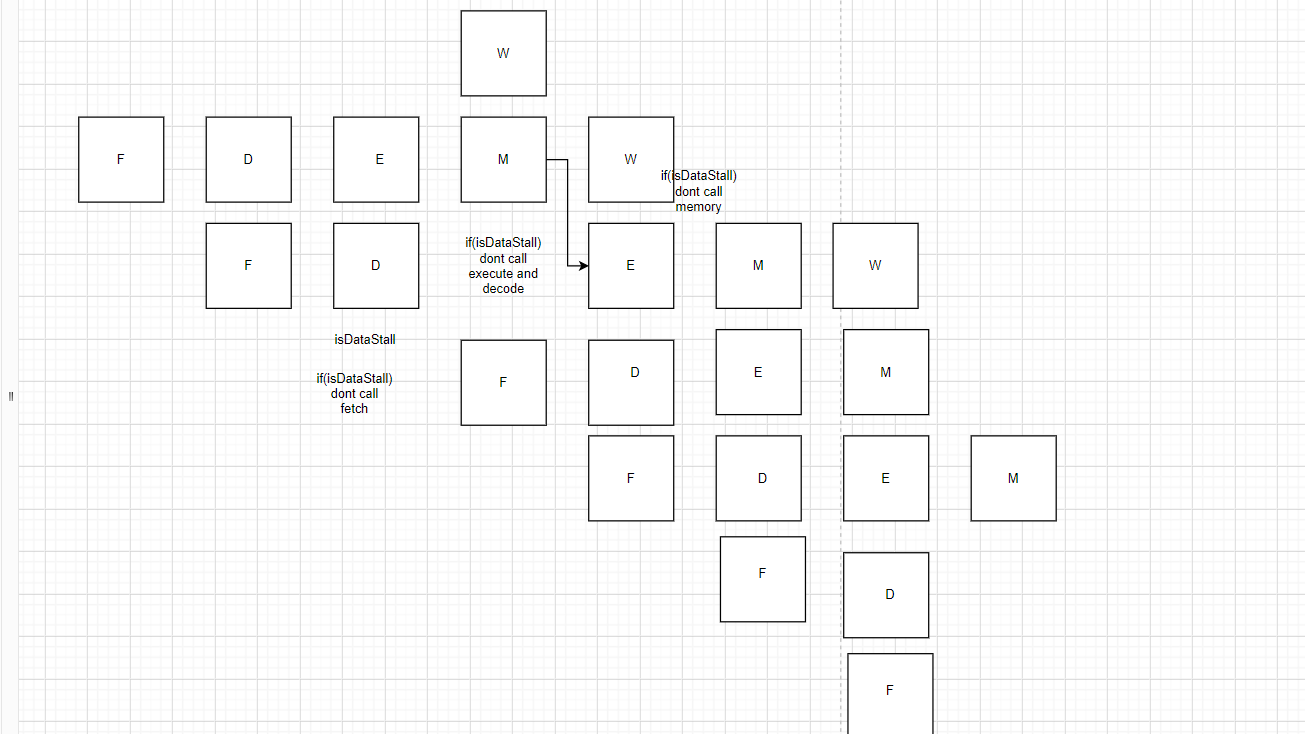
The function acts as a buffer to store the unfinished instruction.

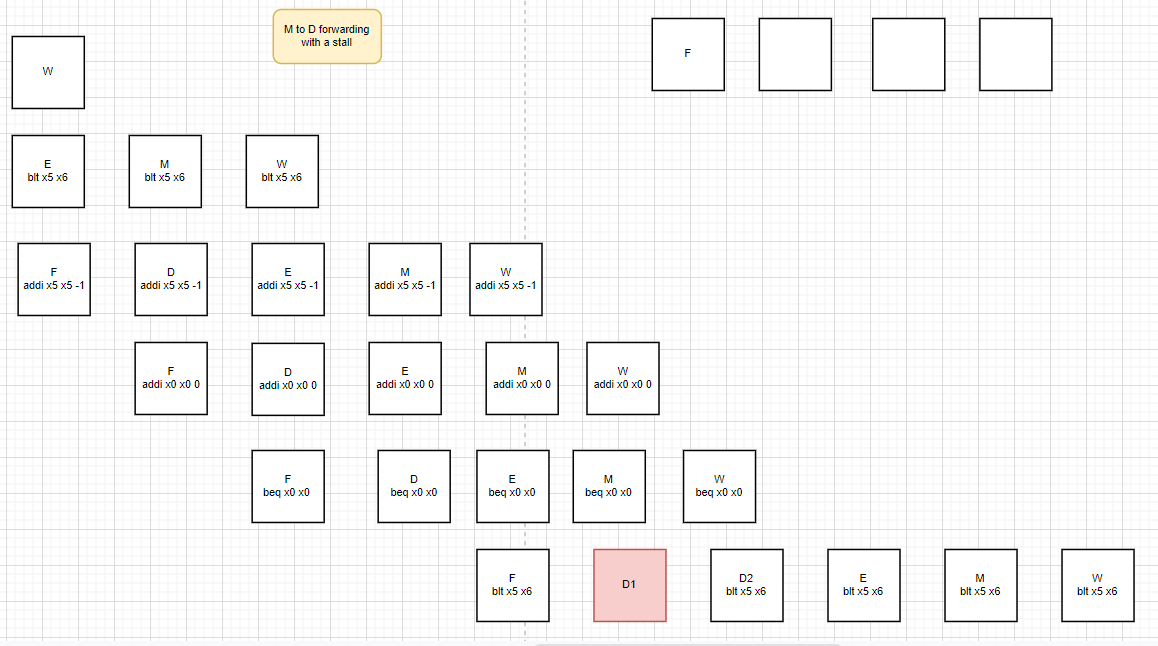
## **Test Plan**

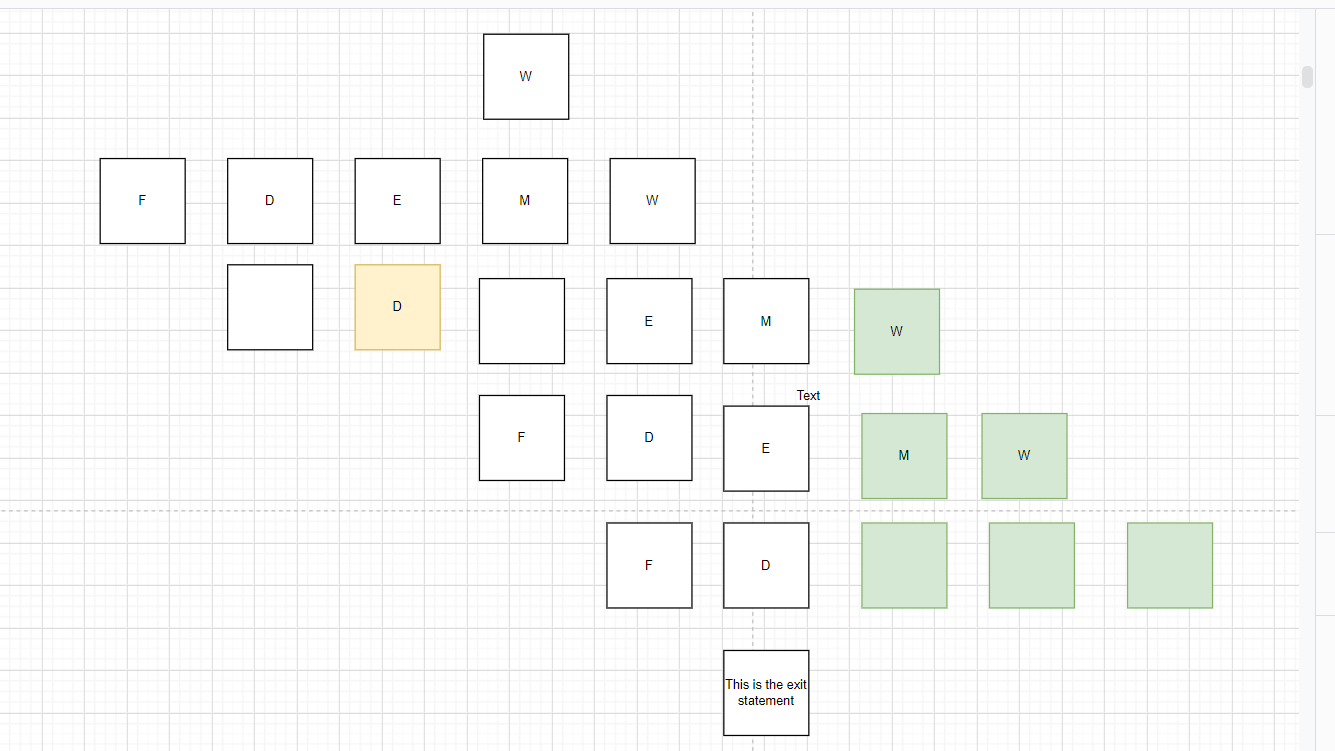
We test the simulator with the following assembly programs:

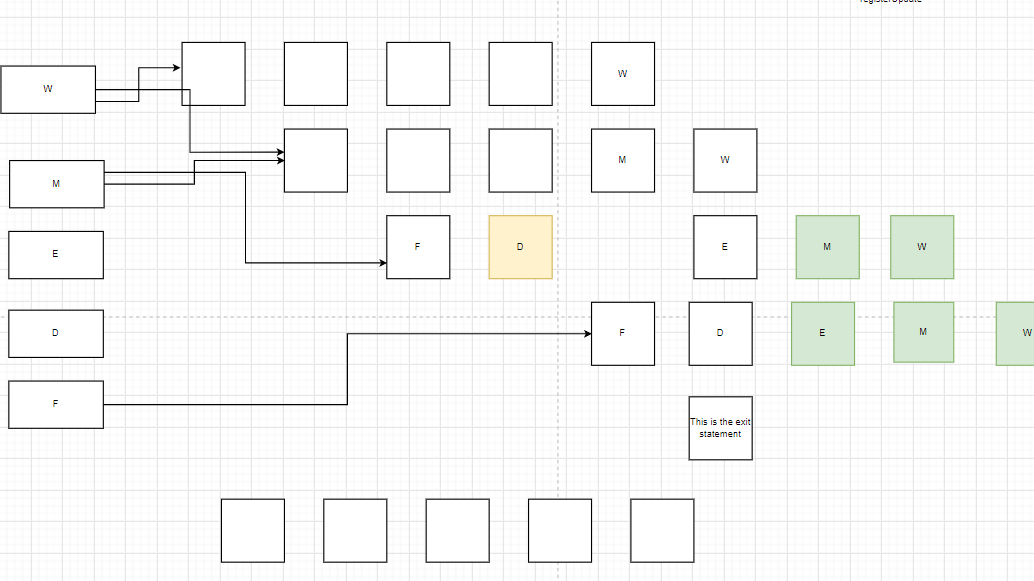
* Fibonacci Program
* Bubble Sort
* Factorial
* Sum of the array of N elements. Initialize an array in the first loop with each element equal to its index. In the second loop find the sum of this array, and store the result at Arr[N].
* Lab-2 machine codes

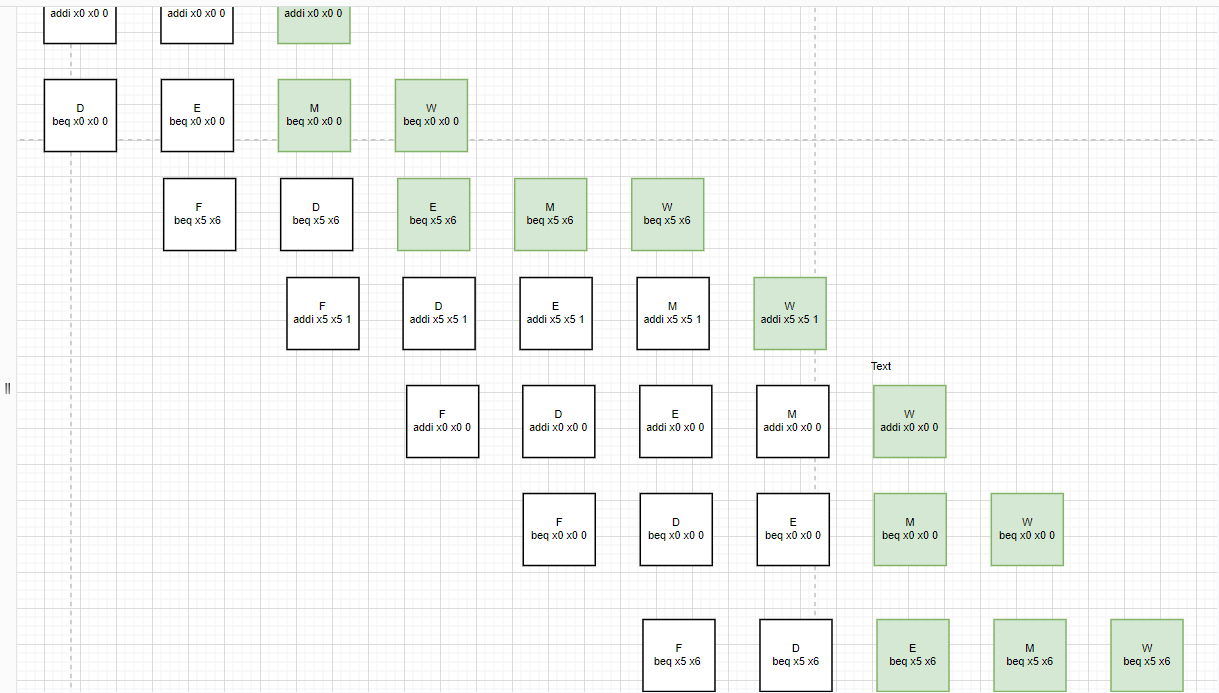
## **Diagrams and Figures**

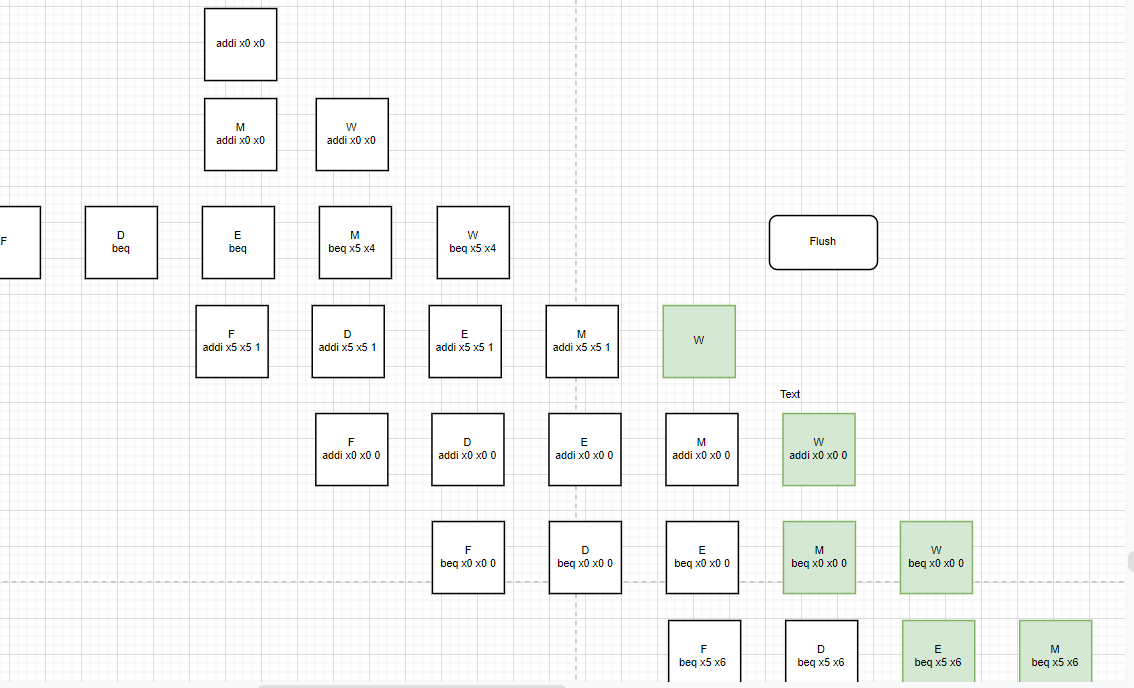
****

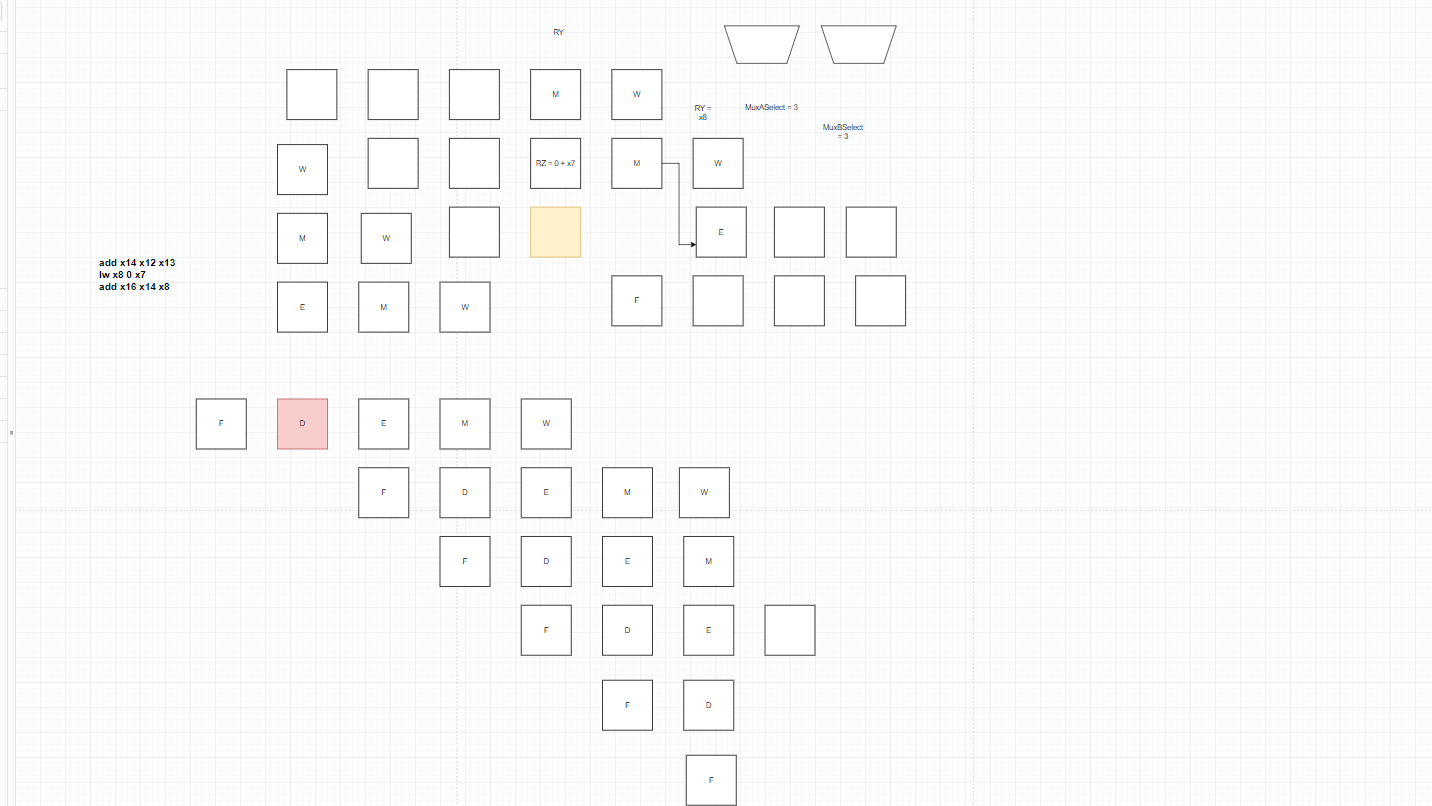
****

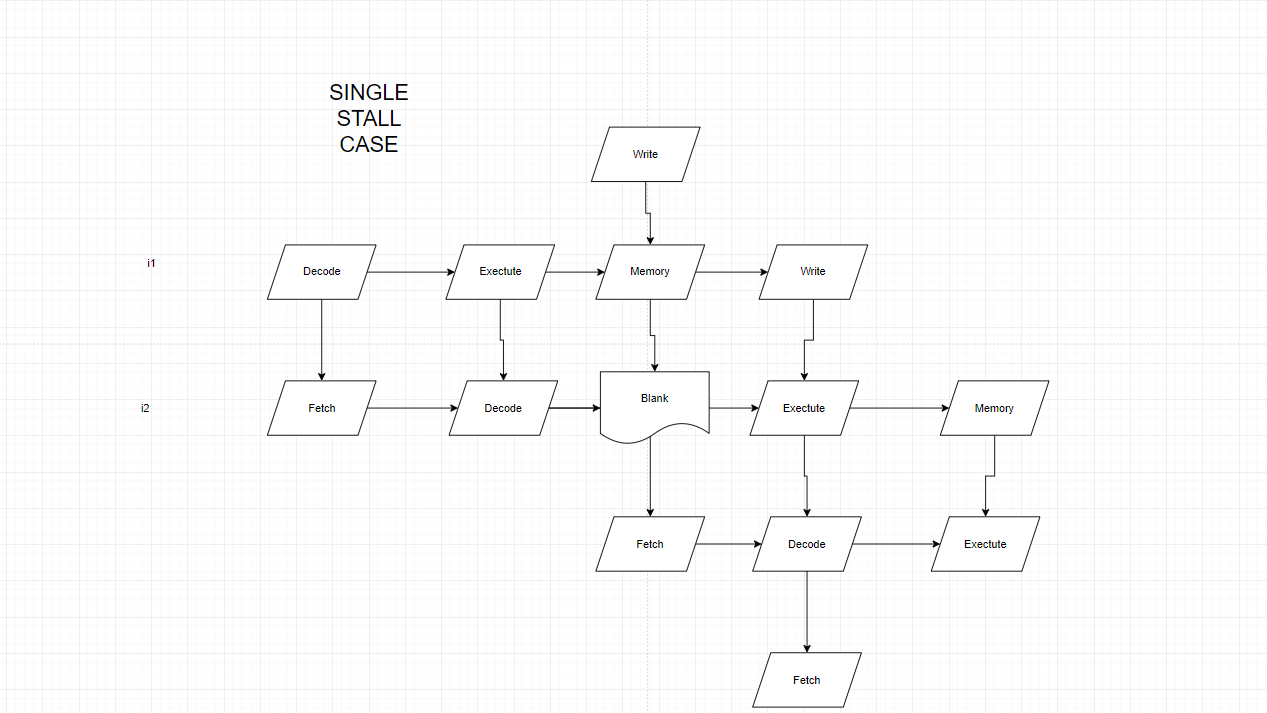
****

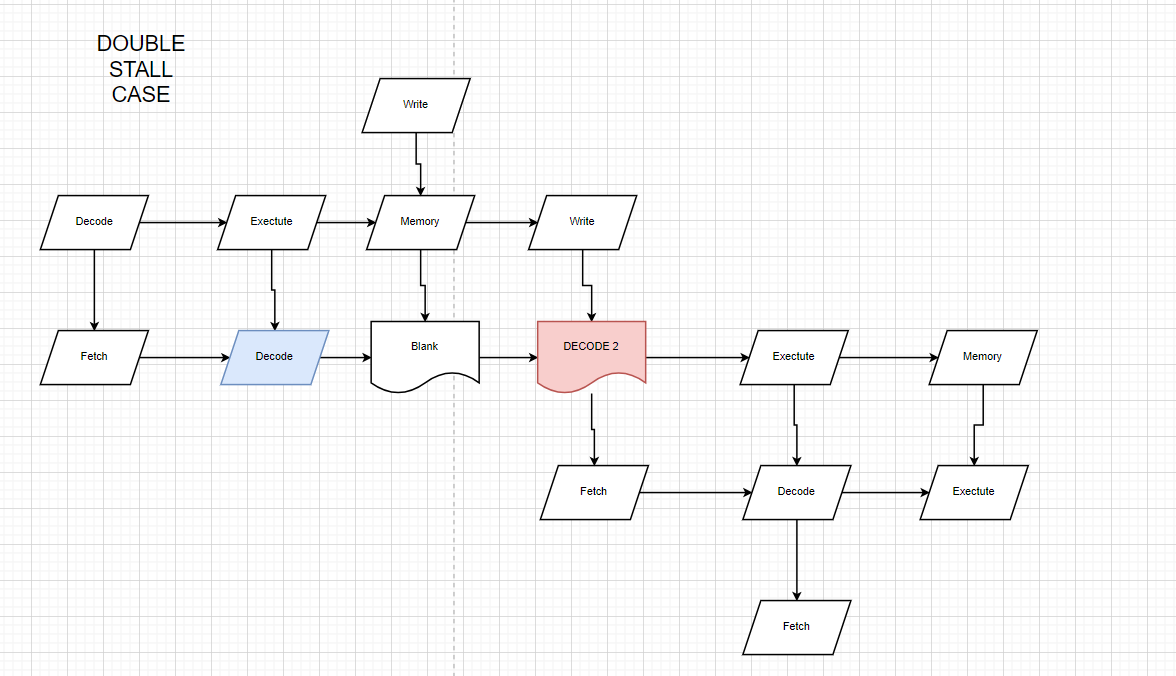
****

****

****

****

****

****

**---------------------------------------------END---------------------------------------------**