RISC-V ISA Simulator:

Design Documentation

Team members:

Bellam Jayanth - 2019CSB1080

Sagina Jaya Govinda Venkata Sai Teja – 2019CSB1114

Rohit Dagar – 2019EEB1186

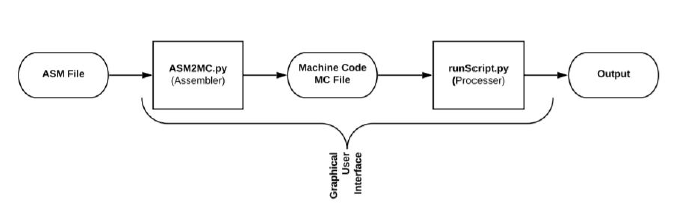
Satia – 2019CSB1118

Keshav Goyal – 2019CSB1165

**Features:**

* **Simulator backend:** Python
* **ISA:** RISC-V RV32IM
* **Size of instruction:** 32 bits
* **Number of registers:** 32
* **Size of registers:** 32 bits

**Overview:**



**Phase 1: Five step instruction execution**

## Input:

Input to the simulator is a machine code file that contains the encoded instruction and the corresponding address at which instruction is supposed to be stored, separated by space.

Functional behaviour:

All the instructions given in the input .mc file is executed as per the functional behaviour of the instructions. Each Instruction will go through the following steps:

* Step 1: **Fetch:** It gets the instruction and increment the PC.
* Step 2: **Decode:** Identify the instruction and registers (source and destination).
* Step 3: **Execute:** ALU operations is done or effective address calculation for load, store instructions. The control circuitry select the ALU operation and either Read data from the registers or a sign-extended immediate value as inputs to the ALU.
* Step 4: **Memory Access:** Read/write data from/to the memory. The control lines set in this stage are Branch, Memory Read, and Memory Write.
* Step 5: **Register Update or Writeback:** update destination register.

The simulator prints messages for each stage:

* **Fetch:** it prints that the instruction is fetched and PC is incremented.
* **Decode:** it recognize and print the operation, rs1, rs2, rd.
* **Execute:** executes the operation and print the instruction statement.
* **Memory access:** print the retrieved memory at the memory address (if required).
* **Writeback:** print the result updated in the destination register.

# Test plan:

We test the simulator with following assembly programs:

* Fibonacci Program

**Phase 2: Pipeline implementation with and without data forwarding**

**Input:**

Input is the same that is a machine code file.

**Pipeline Vs Non-Pipeline:**

|  |  |
| --- | --- |
| Pipeline | non pipeline |
| In pipelining system, multiple instructions are overlapped during execution. | In a Non-Pipelining system, processes like decoding, fetching, execution and writing memory are merged into a single unit or a single step. |
| Many instructions are executed at the same time. | Only one instruction is executed at one time. |
| CPI (ideal) is 1 | CPI is 5 |

**Pipeline Hazards:**

**Structural Hazard**

When a planned instruction cannot execute in the proper clock cycle because the hardware does not support the combination of instructions that are set to execute.

**Solution:**

**Data Hazards**

Also called a **pipeline data hazard**. When a planned instruction cannot execute in the proper clock cycle because data that are needed to execute the instruction are not yet available.

**Solution:**

**Forwarding**

Also called **bypassing**. A method of resolving a data hazard by retrieving the missing data element from internal buffers rather than waiting for it to arrive from programmer-visible registers or memory.

We have included forwarding paths from:

* M to M
* M to E
* E to E

**Stalling**

Also called **bubble**. A stall initiated in order to resolve a hazard.

**Control hazard**

Also called **branch hazard**. When the proper instruction cannot execute in the proper pipeline clock cycle because the instruction that was fetched is not the one that is needed; that is, the flow of instruction addresses is not what the pipeline expected.

**Solution:**

**Branch prediction**

A method of resolving a branch hazard that assumes a given outcome for the conditional branch and proceeds from that assumption rather than waiting to ascertain the actual outcome.

The simulator prints messages for each stage: