PROJECT PHASE 2: Pipelined Implementation of 32bit RISC-V

Design Document

Group No. 18

Group Members:

Rohit Madan – 2020EEB1202

Omkar Prashant More – 2020EEB1188

Harsh Gupta – 2019MED1008

The document describes the Design of our RISC-V simulator, which now acts as a simulator for the Single Cycle Executions as well as Pipelined Executions. Our Simulator is built completely using Python Programming Language.

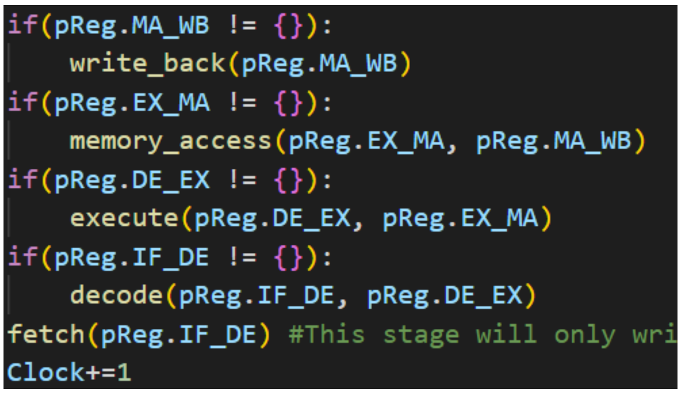
The basic outline of the code remains the same as was implemented in the Single Cycle Implementation.

Pipelined Implementation included 5 pipeline stages ***Instruction Fetch (IF), Decode (DE), Execute (EX) , Memory Access (MA) and Writeback (WB).*** Corresponding to these stages 4 pipeline registers were made in between the stages namely
IF_DE, DE_EX, EX_MA, MA_WB

In main.py, the while executing the Pipelined implementation, we run the stages in the reverse order, i.e.,

1.WB 2. MA 3. EX 4. DE 5. FE

We do this so that we can work on separate instructions in different stages. For example, after 1st instruction is fetched, in the next cycle the next instruction would be fetched whereas this previous instruction would process to the next stage i.e, DE in this case.



```
if(pReg.MA_WB != {}):
    write_back(pReg.MA_WB)
if(pReg.EX_MA != {}):
    memory_access(pReg.EX_MA, pReg.MA_WB)
if(pReg.DE_EX != {}):
    execute(pReg.DE_EX, pReg.EX_MA)
if(pReg.IF_DE != {}):
    decode(pReg.IF_DE, pReg.DE_EX)
fetch(pReg.IF_DE) #This stage will only wri
Clock+=1
```

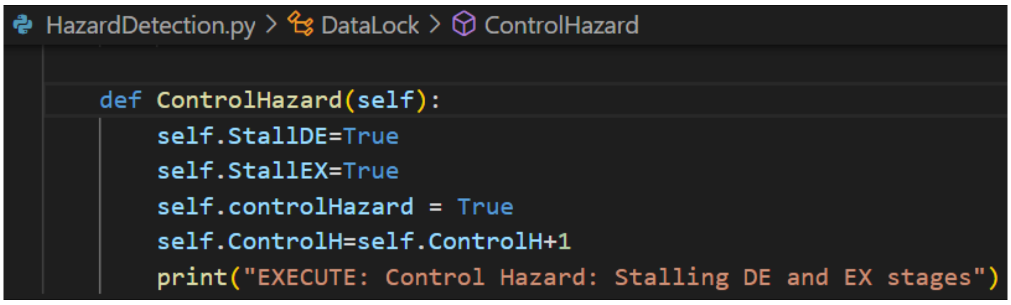
Pipeline Implementation

STALLING

A simple Pipelined execution is implemented with the help of Stalling. Stalls and Bubbles are inserted in the stages when Data and Control Hazards are detected.

- Data Hazard
  - ➢ Data Hazard (Read After Write Dependency (RAW) in our case) Detection is done in the decode stage.
  - ➢ In the case of a Data Hazard, the rs1 or rs2 of a certain instruction are dependent on the rd (Destination register) value of a previous instruction. This error occurs, when the current instruction tries to access the value before it is updated by the preceding instruction.
  - ➢ In this case, we stall the Decode Stage till the dependent preceding instruction writes the value into the Stalling register. Due to this stall in the Decode stage, the subsequent Execute, MemoryAccess as well as the WriteBack Stages are stalled (Bubble) as there was no instruction from the previous stage.
  - ➢ The stall of the Execute Stage is removed in the next cycle after the Decode Stage runs, since now this decoded instruction must be executed, and all the other stages follow so on.

- Control Hazard
  - ➢ A Control Hazard occurs when we do not know whether or not we have to take the Branch. Since this information is known in the Execute Stage, therefore up to that point we Fetch and Decode the next instructions.
  - ➢ In case the Branch is to be taken, then the instruction in the Fetch and Decode stages are flushed, and this creates Bubbles in the Pipeline.
  - ➢ The Control Hazard is detected by us in the Execute Stage, and thereafter the further stages are Cancelled (Bubble).


HazardDetection.py > DataLock > ControlHazard

```python
    def ControlHazard(self):
        self.StallDE=True
        self.StallEX=True
        self.controlHazard = True
        self.ControlH=self.ControlH+1
        print("EXECUTE: Control Hazard: Stalling DE and EX stages")
```

FORWARDING

The basic concept of forwarding is to reduce the stalls since the information that we need after the WriteBack stage is often available to us beforehand.

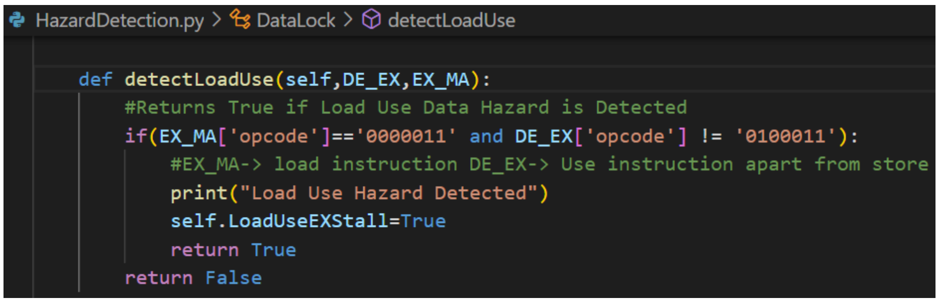
There are 4 forwarding paths all of which were implemented in our Code:

- WB→MA
- WB→EX
- WB→DE
- MA→EX

The implementation of these 4 forwarding paths causes all the stalls/bubbles to go away.

There is only one special case of a Load-Use Hazard which requires stalling. This was also implemented separately as a function in the Data Lock class in HazardDetection.py.

Load Use Hazard only occurs when, a Load Instruction is followed by a non-Store instruction which depends on the load instruction.

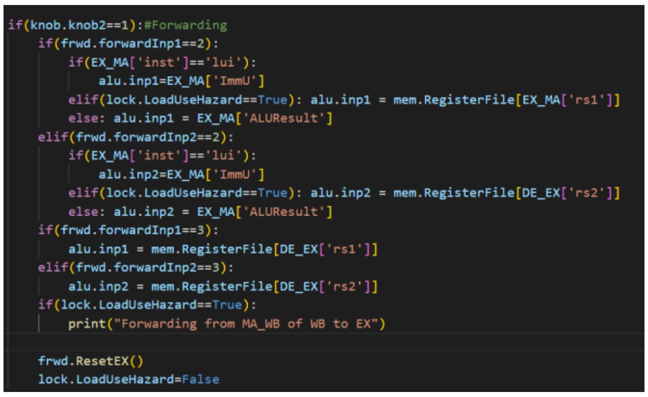


HazardDetection.py > DataLock > detectLoadUse

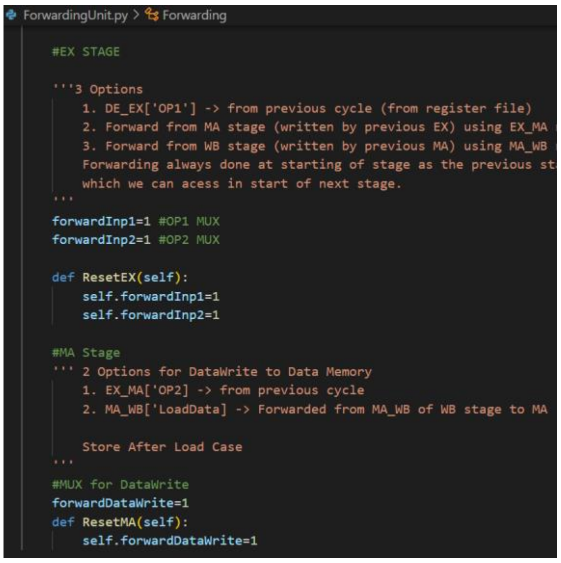
```python
def detectLoadUse(self,DE_EX,EX_MA):
    #Returns True if Load Use Data Hazard is Detected
    if(EX_MA['opcode']=='0000011' and DE_EX['opcode'] != '0100011'):
        #EX_MA-> load instruction DE_EX-> Use instruction apart from store
        print("Load Use Hazard Detected")
        self.LoadUseEXStall=True
        return True
    return False
```

Values of inp1 and inp2 updated in the Execute Stage (Forwarding)

The forwarded values are then updated in the EXECUTE Stage



```python
if(knob.knob2==1):#Forwarding
    if(frwd.forwardInp1==2):
        if(EX_MA['inst']=='lui'):
            alu.inp1=EX_MA['ImmU']
        elif(lock.LoadUseHazard==True): alu.inp1 = mem.RegisterFile[EX_MA['rs1']]
        else: alu.inp1 = EX_MA['ALUResult']
    elif(frwd.forwardInp2==2):
        if(EX_MA['inst']=='lui'):
            alu.inp2=EX_MA['ImmU']
        elif(lock.LoadUseHazard==True): alu.inp2 = mem.RegisterFile[DE_EX['rs2']]
        else: alu.inp2 = EX_MA['ALUResult']
    if(frwd.forwardInp1==3):
        alu.inp1 = mem.RegisterFile[DE_EX['rs1']]
    elif(frwd.forwardInp2==3):
        alu.inp2 = mem.RegisterFile[DE_EX['rs2']]
    if(lock.LoadUseHazard==True):
        print("Forwarding from MA_WB of WB to EX")

    frwd.ResetEX()
    lock.LoadUseHazard=False
```



ForwardingUnit.py > Forwarding

```python
    #EX STAGE

    '''3 Options
        1. DE_EX['OP1'] -> from previous cycle (from register file)
        2. Forward from MA stage (written by previous EX) using EX_MA
        3. Forward from WB stage (written by previous MA) using MA_WB
        Forwarding always done at starting of stage as the previous st
        which we can acess in start of next stage.
    '''
    forwardInp1=1 #OP1 MUX
    forwardInp2=1 #OP2 MUX

    def ResetEX(self):
        self.forwardInp1=1
        self.forwardInp2=1

    #MA Stage
    ''' 2 Options for DataWrite to Data Memory
        1. EX_MA['OP2] -> from previous cycle
        2. MA_WB['LoadData] -> Forwarded from MA_WB of WB stage to MA

        Store After Load Case
    '''
    #MUX for DataWrite
    forwardDataWrite=1
    def ResetMA(self):
        self.forwardDataWrite=1
```

Values of inp1 and inp2 updated in the Execute Stage (Forwarding) and Forwarding Unit class

BRANCH PREDICTION

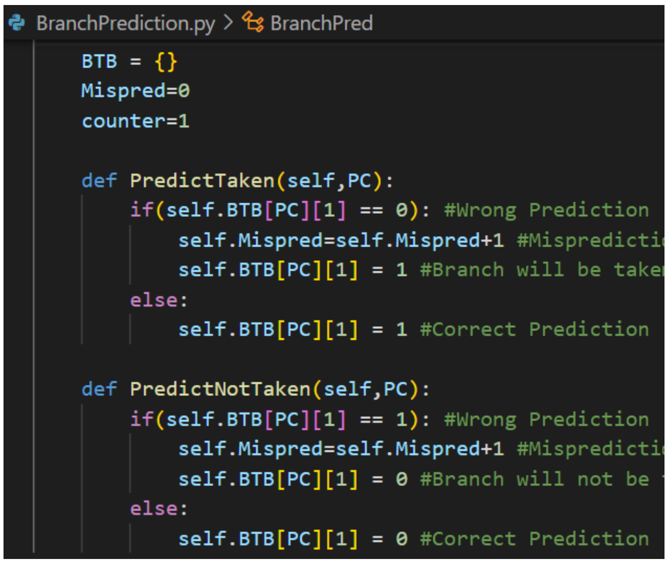
Branch Prediction was done using 1 bit Branch prediction logic, which sets a counter = 0 or 1 and states that a Branch continues to be taken till the counter is 1 and once we see that the branch is not to be taken, we set the counter = 0, and whether to Fetch the next instruction or branch target is decided by the value of our counter.

If a Branch instruction is fetched in current cycle, then we want to use our branch predictor to predict the direction of the branch.

In the next cycle the branch would move to Decode stage where an entry corresponding to it would be made in the BTB

So we then fetch Branch Target from the BTB corresponding to that branch

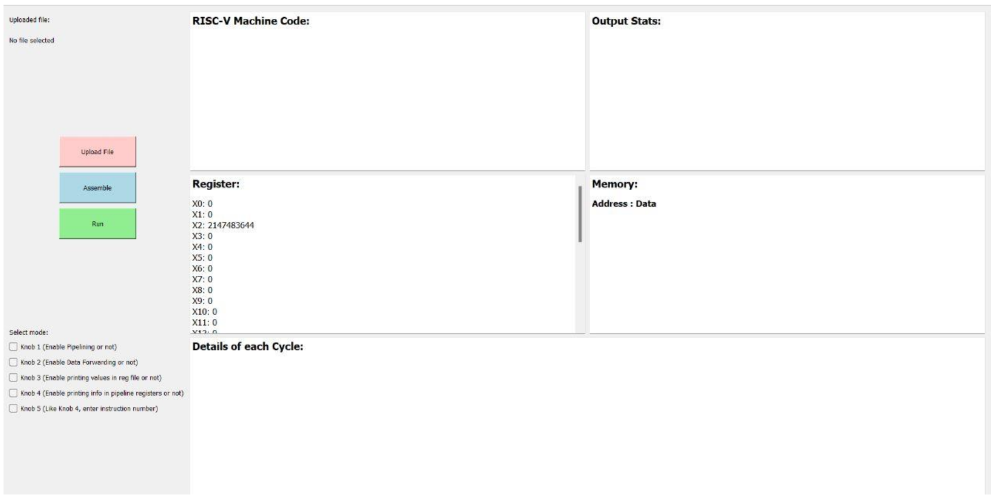
In the next cycle when the branch reaches Execute stage then we get to know whether branch prediction was correct or not and whether we need to flush the pipeline



```
BranchPrediction.py > BranchPred
    BTB = {}
    Mispred=0
    counter=1

    def PredictTaken(self,PC):
        if(self.BTB[PC][1] == 0): #Wrong Prediction
            self.Mispred=self.Mispred+1 #Mispredicti
            self.BTB[PC][1] = 1 #Branch will be take
        else:
            self.BTB[PC][1] = 1 #Correct Prediction

    def PredictNotTaken(self,PC):
        if(self.BTB[PC][1] == 1): #Wrong Prediction
            self.Mispred=self.Mispred+1 #Mispredicti
            self.BTB[PC][1] = 0 #Branch will not be
        else:
            self.BTB[PC][1] = 0 #Correct Prediction
```

Branch Prediction function

GUI



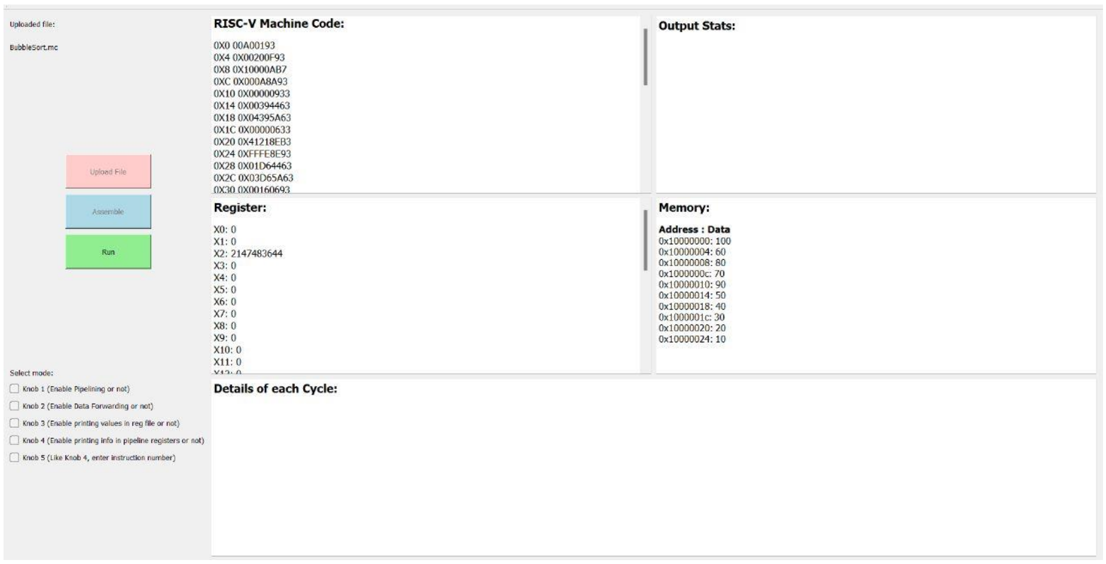
```
Uploaded file:
No file selected

RISC-V Machine Code:

Output Stats:

Register:
X0: 0
X1: 0
X2: 2147483644
X3: 0
X4: 0
X5: 0
X6: 0
X7: 0
X8: 0
X9: 0
X10: 0
X11: 0

Memory:
Address : Data

Select mode:
☐ Knob 1 (Enable Pipelining or not)
☐ Knob 2 (Enable Data Forwarding or not)
☐ Knob 3 (Enable printing values in reg file or not)
☐ Knob 4 (Enable printing info in pipeline registers or not)
☐ Knob 5 (Like Knob 4, enter instruction number)

Details of each Cycle:
```

Initial State of GUI

The GUI contains the following features:

BUTTONS:

- Upload File: This button allows us to upload the .mc file of our code.
- Assemble: Assembles the instruction and data memory code in the RISCV-Machine Code section



Uploaded file:

BubbleSort.mc

RISC-V Machine Code:

```
0X0 00A00193
0X4 0X00200F93
0X8 0X10000AB7
0XC 0X000A8A93
0X10 0X00000933
0X14 0X00394463
0X18 0X04395A63
0X1C 0X00000633
0X20 0X41218EB3
0X24 0XFFFE8E93
0X28 0X01D64463
0X2C 0X03D65A63
0X30 0X00160693
```

Output Stats:

Register:

```
X0: 0
X1: 0
X2: 2147483644
X3: 0
X4: 0
X5: 0
X6: 0
X7: 0
X8: 0
X9: 0
X10: 0
X11: 0
```

Memory:

```
Address : Data
0x10000000: 100
0x10000004: 60
0x10000008: 80
0x1000000c: 70
0x10000010: 90
0x10000014: 50
0x10000018: 40
0x1000001c: 30
0x10000020: 20
0x10000024: 10
```

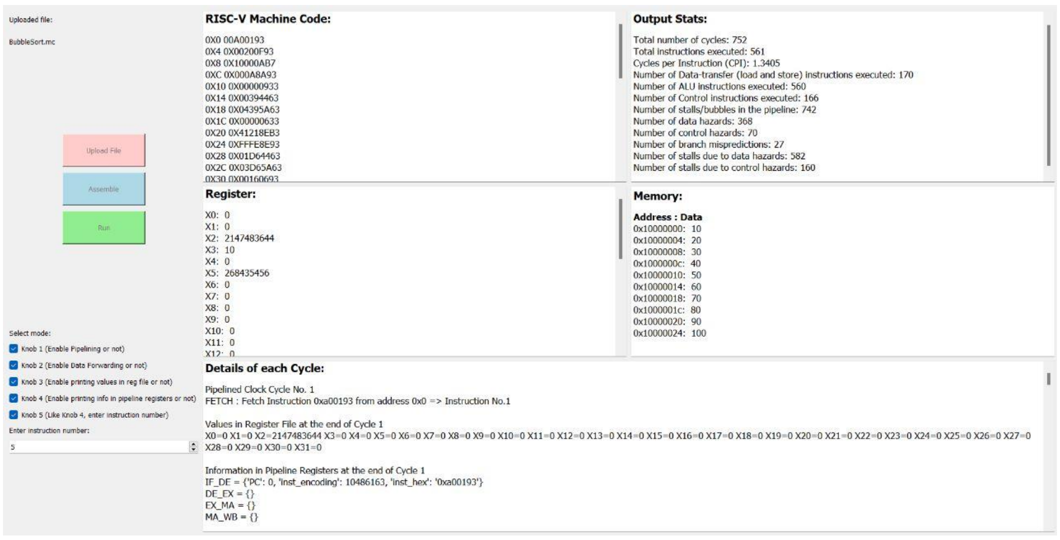
Select mode:

- [ ] Knob 1 (Enable Pipelining or not)
- [ ] Knob 2 (Enable Data Forwarding or not)
- [ ] Knob 3 (Enable printing values in reg file or not)
- [ ] Knob 4 (Enable printing info in pipeline registers or not)
- [ ] Knob 5 (Like Knob 4, enter instruction number)

Details of each Cycle:

After Assembling

- Run: Runs our code and gives us the output.



Uploaded file:

BubbleSort.mc

RISC-V Machine Code:

```
0X0 00A00193
0X4 0X00200F93
0X8 0X10000AB7
0XC 0X000A8A93
0X10 0X00000933
0X14 0X00394463
0X18 0X04395A63
0X1C 0X00000633
0X20 0X41218EB3
0X24 0XFFFE8E93
0X28 0X01D64463
0X2C 0X03D65A63
0X30 0X00160693
```

Output Stats:

```
Total number of cycles: 752
Total instructions executed: 561
Cycles per Instruction (CPI): 1.3405
Number of Data-transfer (load and store) instructions executed: 170
Number of ALU instructions executed: 560
Number of Control instructions executed: 166
Number of stalls/bubbles in the pipeline: 742
Number of data hazards: 368
Number of control hazards: 70
Number of branch mispredictions: 27
Number of stalls due to data hazards: 582
Number of stalls due to control hazards: 160
```

Register:

```
X0:  0
X1:  0
X2:  2147483644
X3:  10
X4:  0
X5:  268435456
X6:  0
X7:  0
X8:  0
X9:  0
X10:  0
X11:  0
```

Memory:

```
Address : Data
0x10000000:  10
0x10000004:  20
0x10000008:  30
0x1000000c:  40
0x10000010:  50
0x10000014:  60
0x10000018:  70
0x1000001c:  80
0x10000020:  90
0x10000024:  100
```

Select mode:

- [✓] Knob 1 (Enable Pipelining or not)
- [✓] Knob 2 (Enable Data Forwarding or not)
- [✓] Knob 3 (Enable printing values in reg file or not)
- [✓] Knob 4 (Enable printing info in pipeline registers or not)
- [✓] Knob 5 (Like Knob 4, enter instruction number)

Enter instruction number: 5

Details of each Cycle:

```
Pipelined Clock Cycle No. 1
FETCH : Fetch Instruction 0xa00193 from address 0x0 => Instruction No.1

Values in Register File at the end of Cycle 1
X0=0 X1=0 X2=2147483644 X3=0 X4=0 X5=0 X6=0 X7=0 X8=0 X9=0 X10=0 X11=0 X12=0 X13=0 X14=0 X15=0 X16=0 X17=0 X18=0 X19=0 X20=0 X21=0 X22=0 X23=0 X24=0 X25=0 X26=0 X27=0
X28=0 X29=0 X30=0 X31=0

Information in Pipeline Registers at the end of Cycle 1
IF_DE = {'PC': 0, 'inst_encoding': 10486163, 'inst_hex': '0xa00193'}
DE_EX = {}
EX_MA = {}
MA_WB = {}
```

After Running BubbleSort.mc

KNOBS:

- All the 5 Knobs are present on the bottom left of the GUI window, and they can be turned on by checking the boxes to the left of them
- After checking the 5th Knob, we are provided an option to enter the instruction number for which we wish to see information in the pipeline registers

OUTPUT:

In the Output we can see the following things:

- Machine Code
- Output Stats
- Register Values
- Updated Data Memory
- Details of each Cycle
  - Cycle wise Details
  - Register values
  - Pipeline Register Values