PROJECT PHASE 3:

Appending Cache Module to 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 along with a Cache Module. 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 + Pipeline Implementation.

For Phase 3, we had to instantiate two caches, namely the Instruction cache for the Instruction memory, and the Data Cache for the Data memory.

The changes that were made in order to handle the Cache Module were in the Fetch and Memory Access Stage.

And we implemented 4 classes in the Memory.py file.

- **MainMemory**

The dictionaries for dataMemory and instructionMemory were defined in this class. Also, the readFile function was defined in this class, which read our input.mc file segregated it into data and instruction memory. This segregation was determined by the termination code which was also used in the previous phases.

The instruction encoding and data is stored in bytes.

```
self.dataMemory[4*newY][indexInBlock][0] = hex(int(y[1],16) & int('0xFF',16))[2:] # the LSB is stored in the zeroth position of
self.dataMemory[4*newY][indexInBlock][1] = hex((int(y[1],16) & int('0xFF00',16))>>8)[2:]
self.dataMemory[4*newY][indexInBlock][2] = hex((int(y[1],16) & int('0xFF0000',16))>>16)[2:]
self.dataMemory[4*newY][indexInBlock][3] = hex((int(y[1],16) & int('0xFF000000',16))>>24)[2:] # the MSB is stored in the third
```

➢ InstructionCacheMemory and DataCacheMemory

In both the instruction as well the Data cache memory we took the parameters as input that is block size, cache size, and the instruction associativity.

Then we defined the BlockOffset, Index and Tag since we know the associativity of the Cache (whether it is Direct-mapped, Fully Associative, Set Associative)

Also, a validBit array was defined which checks whether the entry into the block is Valid or not.

LRU (Least Recently Used) policy was also defined. IN this function we checked for the victim which is to be evicted in the case when the cache is full.

```
LRU(self,index):
for i in range(self.numWays):
    if(self.forLRU[index][i]==0):
        if(self.validBit[index][i].count(1)>0):
        #if(self.validBit[index][i]==1):
            self.victim = i+1
        else:
            self.victim = 0
        return i
return 0
```

readCache function was defined which reads the value from the instruction or data array.

In the DataCacheMemory class, other than the above defined function, we also defined a writeCache function which writes the data in the dataMemory.

➢ CacheControl

Lastly, we have a CacheControl Class which is made to read the Instruction and Data Cache memory and return the instruction encoding(I$)/Word(D$) present at the given address.

```
def readInstructionCacheMem(self,pc,InstructionCacheMemory,mainMem):
    address = hex(pc)
    ans = InstructionCacheMemory.readCache(address,mainMem)
    if(ans[0]==-1 and ans[1]==-1 and ans[2]==-1 and ans[3]==-1):
        return "Invalid"
    newans = ""
    x=len(ans)
    for i in range(len(ans)):
        newans += str(ans[x-1-i])
    newans = '0x'+newans
    return newans
```

Fetch Stage

The instruction encoding is now accessed in the Fetch stage as follows:

Fetch.py > fetch

```
        val = cacheCont.readInstructionCacheMem(pkt.PC,mem,mainMem)
        if(val !='Invalid'):inst_encoding = int(val,16)
        else: inst_encoding=TerminationCode
```

Memory Access Stage

Load Instruction

```
if control.MemOP == 0: #Write Disabled
  if pkt.inst in ['lb','lh','lw']:
    pkt.LoadData = cacheCont.readDataMem(hex(pkt.ALUResult),datamem,mainMem)
  else:
    pkt.LoadData = 0
```

Store Instruction

```
elif(pkt.inst=="sw"):
  dataStore = pkt.op2
  datamem.writeCache(pkt.ALUResult,dataStore,mainMem)
  print("MEMORY: " + "Storing" , dataStore , "at address" , hex(pkt.ALUResult))
  stat.TotalDataStoreinst = stat.TotalDataStoreinst+1
```