Implementation of Scoreboard in Python

Computer Architecture Practise - Dr. Noor Mahammad SK
Kruttika Bhat
CED16I010

scoreboard.py

import run will import the functions in run.py

Classes:

1. Instruction class:

This class stores information about each instruction.

```
class Instruction():
    def __init__(self, op, dest, s1, s2):
        self.op =  op
        self.dest = dest
        self.s1 = s1
        self.s2 = s2
        self.issueTime=self.readTime=self.execTime=self.writeTime=0
        self.delay=self.getdelay()
    def getdelay(self): #assume 190ps=19cc
        switch = {
        "LDR": 1,
        "STR": 1,
        "ADD": 5, #22
        "SUB": 5, #22
        "MUL": 10, #58
        "FADD": 6, #144
        "FMUL": 6, #107
        "FSUB": 6, #144
        "AND": 1,
        "OR": 1
        }
        return switch[self.op]
```

Variables:

op - Tells which operation the instruction will be performing (Eg: LDR, STR, ADD,etc)
dest - The destination register
s1 - The first register. It maybe an offset as well.
s2 - The second register.
issueTime - The clock cycle that the instruction is issued
readTime - The clock cycle that the operands are read
execTime - The clock cycle that the instruction finishes execution
writeTime - The clock cycle that the output is written to the destination register
delay - The amount of clock cycles required for the operation being performed by that particular instruction

Functions:

getdelay() - This will return the number of clock cycles needed for a certain operation.

2. FunctionalUnit class:

This class gives information about each functional unit. There are 6 functional units

- Integer (LDR/STR)
- Add(ADD/SUB)
- Mult(MUL)
- FPA(FADD/FSUB)
- FPM(FMUL)
- Bool(AND/OR)

```
class FunctionalUnit():
    def __init__(self,name):
        self.name = name
        self.busy = "no"
        self.op = self.fi = self.fj = self.fk = self.qj = self.qk = self.rj = self.rk = ""
        self.ins=-1
```

Variables:

name - Stores the name of the functional unit
busy - Says whether the unit is busy or not (yes/no). Initialized with no.
op - Stores the operation of the instruction assigned to that unit.
fi - Stores the destination register of the instruction assigned to that unit.
fj - Stores source 1 register of the instruction assigned to that unit if source 1 is a register, otherwise it is kept empty
fk - Stores source 2 register of the instruction assigned to that unit.
qj - If the register for source 1 is not available (i.e. That register is the output for another currently executing operation) then this will tell which functional unit will give that register.
qk - If the register for source 2 is not available (i.e. That register is the output for another currently executing operation) then this will tell which functional unit will give that register.
rj - yes- if register(source 1) is available, no- if register(source 1) is not available
rk - yes- if register(source 2) is available, no- if register(source 2) is not available
ins - Stores the instruction number of the instruction being executed by that functional unit

Global Variables:

- inst - This is an array of objects. Each of type Instruction.
- funcUnits - This is an array of objects. Each of type FunctionalUnit.
- current - Stores the instruction number that we currently want to issue.
- issueDone - 0->all instructions have not been issued; 1->all instructions have been issued
- n - The total number of instructions
- registers - An array which stores the values of the registers.

Functions:

1. **check(reg,reqUnit):**
   Input:
   - reg - the register that we are searching for
   - reqUnit - the functional unit that we have currently issued the instruction in.

   This function will scan the functional unit and return the name of the functional unit if there is a match between the input register and the fi of any functional unit which is not the one that we are currently issuing.

2. **get_operation(op):** Returns the index of the functional unit given the operation.

```
def get_operation(op):
    operations = {
    "LDR": 0,
    "STR": 0,
    "ADD": 1,
    "SUB": 1,
    "MUL": 2,
    "FADD": 3,
    "FMUL": 4,
    "FSUB": 3,
    "AND": 5,
    "OR": 5
    }
    return operations.get(op)
```

3. **get_function(op):** Returns the name of the functional unit given the operation.

```
def get_function(op):
    functions={
        "LDR": "Integer",
        "STR": "Integer",
        "ADD": "Add",
        "SUB": "Add",
        "MUL": "Mult",
        "FADD": "FPA",
        "FMUL": "FPM",
        "FSUB": "FPA",
        "AND": "Bool",
        "OR": "Bool"
    }
    return functions.get(op)
```

4. **sets1(reqUnit):**
   Input: index of functional unit
   Check if source 1 is a register or an offset.
   - If it is a register, then set fj as s1.
     - ❖ Check if s1 is the destination of any other operation in the functional unit.

- ❖ If it is, then set qj as the name of the functional unit (fu) and set rj as no.
- ❖ If not, set qj empty and set rj as yes.

- If it is not a register, make fj, qj and rj empty.

5. **sets2(reqUnit):**
   Input: index of functional unit
   - Set fk as s2.
   - Check if s2 is the destination of any other operation in the functional unit.
     - ❖ If it is, then set qj as the name of the functional unit (fu) and set rj as no.
     - ❖ If not, set qj empty and set rj as yes.

6. **issue():**

```
def issue():
    global current
    global issueDone
    reqUnit=get_operation(inst[current].op)
    #check for waw hazard and don't issue if found
    waw=0;
    for i in funcUnits:
        if(i.fi==inst[current].dest):
            waw=1;
    if(funcUnits[reqUnit].busy=="no" and waw==0):
        inst[current].issueTime = clk
        funcUnits[reqUnit].ins=current
        funcUnits[reqUnit].busy="yes"
        funcUnits[reqUnit].op=inst[current].op
        funcUnits[reqUnit].fi=inst[current].dest
        sets1(reqUnit)
        sets2(reqUnit)
        current=(current+1)%n

        if(current==0):
            issueDone=1
```

- First get the index of the required unit based on the operation which the current instruction needs to perform.
- **Check for WAW hazard.** See if there is already an instruction in the functional unit which has the same destination as the instruction we want to issue.
- If WAW hazard does not exist and the functional unit we need is not busy then issue the instruction. Also, set the issueTime of our current instruction as the clock time.
- Increment the current variable and check if we have finished issuing all the instructions. If we have, set issueDone as 1.

7. **read():**
   For each instruction in the functional unit, get the instruction number. The registers can be read if:
   - instNumber!=-1 -> an instruction has been issued in that functional unit
   - inst[instNumber].issueTime!=clk -> Issue and Read cannot happen during the same clock cycle. This condition prevents that from happening.
   - inst[instNumber].readTime==0 -> The registers should not have been already read
   - funcUnits[i].rk==”yes” and funcUnits[i].rj!=”no” -> Both registers should be readable

8. **get_regvalue(reg):**
   Gets the index value of the given the reg, and returns the corresponding value from registers. (Eg: if reg is R6, then we will return registers[6])

9. **set_regvalue(dest,val):**
   Sets the register value given the destination index and the value.

10. **exec():**
    Scan through all the functional units. Conditions for executing an operation:
    - instNumber!=-1 -> an instruction has been issued in that functional unit
    - inst[instNumber].readTime!=clk -> Read and Execute cannot happen during the same clock cycle. This condition prevents that from happening.
    - inst[instNumber].readTime!=0 -> The registers should have been read
    - inst[instNumber].execTime==0 -> The operation should not have been executed
    - The present clock cycle should equal the time the registers were read plus the time needed to execute the operation

    Now execute the operation.
    - Get the operation, the destination register, source 1 and source 2.
    - Check if operation is LDR/STR.
    - If yes, then calculate the offset = s1/4 (Eg: If s1 is #8, then offset will be 8/4=2).
      - ❖ In case of LDR, get the index of the source register by adding the offset to s2 (Eg: If s2 is R0, then source register will be 0+2=2).
      - ❖ Get the value from the source and get the destination index. Set the register to the value.
      - ❖ In case of STR, the destination register will actually be holding the value that we want to store and s2 will be the register in which we will store the value. Do similar operations as LDR
    - If not then get the value stored in s1 and s2 and also get the destination index.

- Now we call the function call_verilog_function which can be found in run.py. This will execute the verilog programs given the operation and register values and return the output of the operation.
- Set the execution time as the clock value.

```
def exec():
    global registers
    for i in range(6):
        instNumber=funcUnits[i].ins
        if(instNumber!=-1 and inst[instNumber].readTime!=clk
           and inst[instNumber].readTime!=0 and inst[instNumber].execTime==0):
            if(inst[instNumber].readTime+inst[instNumber].delay==clk):
                op=inst[instNumber].op
                dest=inst[instNumber].dest
                s1=inst[instNumber].s1
                s2=inst[instNumber].s2
                if(op=="LDR"): #LDR R1,#8,R2 -> register[1]=register[2+2]
                    offset=int(int(s1[1:])/4)
                    srcIndex=offset+int(s2[1:])
                    val=registers[srcIndex]
                    destIndex=int(dest[1:])
                elif(op=="STR"):
                    val=get_regvalue(dest)
                    offset=int(int(s1[1:])/4)
                    destIndex=offset+int(s2[1:])
                else:
                    a=get_regvalue(s1)
                    b=get_regvalue(s2)
                    destIndex=int(dest[1:])
                    val=run.call_verilog_function(op,a,b,instNumber)
                set_regvalue(destIndex,val)
                inst[instNumber].execTime=clk
```

11. commit():

Scan through all functional units. A commit or write result can happen if:

- instNumber!=-1 -> an instruction has been issued in that functional unit
- inst[instNumber].execTime!=clk -> Execute and commit cannot happen during the same clock cycle. This condition prevents that from happening.
- inst[instNumber].execTime!=0 -> The operation should have been executed
- inst[instNumber].writeTime==0 -> The result should not have been written

Check for WAR hazard:

There will be WAR hazard if for any instruction before the instruction we want to commit:

- inst[j].readTime==0 or inst[j].readTime==clk -> if WAR hazard exists, we cannot commit if the register hasn't been read or during the same clock cycle that the register was read.
  Eg: MUL R0,R1,R2

SUB R1,R5,R4

We cannot commit the result to R1 till R1 of the MUL instruction has been read

- The destination register of the current instruction is equal to any register that is being read in the functional Unit (remember that we are only checking the instructions that precede the current instruction)

If WAR hazard does not exist:

- Set the write time as the clock.
- Go through all the functional units and check if any functional unit requires the register that we just commited. If so set the rj/rk as yes and qj/qk as empty
- Set all the variables of the current functional unit to their initial values.

```
def commit():
    for i in range(6):
        instNumber=funcUnits[i].ins
        if(instNumber!=-1 and inst[instNumber].execTime!=clk
            and inst[instNumber].execTime!=0 and inst[instNumber].writeTime==0):
            war=0
            for j in range(instNumber):
                if((inst[j].readTime==0 or inst[j].readTime==clk) and
                (inst[instNumber].dest==inst[j].s1 or inst[instNumber].dest==inst[j].s2)):
                    war=1
            if(war==0):
                inst[instNumber].writeTime=clk
                for k in range(6):
                    if (k!=i and funcUnits[i].fi==funcUnits[k].fj):
                        funcUnits[k].rj="yes"
                        funcUnits[k].qj=""
                    if (k!=i and funcUnits[i].fi==funcUnits[k].fk):
                        funcUnits[k].rk="yes"
                        funcUnits[k].qk=""
                funcUnits[i].busy="no"
                funcUnits[i].op=""
                funcUnits[i].fi=""
                funcUnits[i].fj=""
                funcUnits[i].fk=""
                funcUnits[i].rj=""
                funcUnits[i].rk=""
                funcUnits[i].ins=-1
```

12. **printScoreboard():** Prints the instructional unit

13. **printFunctionalUnit():** Prints the functional unit

14. **main():**

    Initialise current, issueDone, funcUnits, registers, clk. Get the instructions, either from command prompt or from reading text file.

    Command Prompt:

- input() will get the input from the command line
- split() by default will split a line based on spaces otherwise according to the specified parameter.
- append() adds an element to an array. Here we are adding an object.

```
n=int(input("Enter the number of instructions"))
for i in range(n):
    [op,regSet]=input().split()
    [dest,s1,s2]=regSet.split(",")
    inst.append(Instruction(op,dest,s1,s2))
```

Using Text File:

Open the file in read mode. strip() removes new line characters. Store the instructions in an array and get it's length. This will be the number of instructions. The rest is same as getting input from command prompt.

```
f = open("inst2.txt", "r")
setOfInstructions = [line.strip() for line in f]
n=len(setOfInstructions)
f.close()
for i in setOfInstructions:
    [op,regSet]=i.split()
    [dest,s1,s2]=regSet.split(",")
    inst.append(Instruction(op,dest,s1,s2))
```

After initialisation and getting the instructions, run issue()(if issue is not done), read(), exec() and commit(). Check if all instructions have been written (meaning all are done). If not repeat till all are done.

```
notDone=0
while(notDone==0):
    #print('\nissueMain: '+str(iss
    if(issueDone==0):
        issue()
    read()
    exec()
    commit()
    print("\nClk:"+str(clk))
    printScoreboard()
    printFunctionalUnit()
    notDone=1
    for i in range(n):
        if(inst[i].writeTime==0):
            notDone=0
            break
    clk=clk+1
```

run.py

import os allows us to use os.system call to run a command.

call_verilog_function(op,a,b,index):

Input:

- op is the operation we are performing
- a and b are input values
- index is the instruction number that we are executing

For any operation we will first generate the string that we want to run. This will be of the form (If we are running ADD operation, a=3 and b=4)

```
iverilog -o output -DA=3 -DB=4 CLA/carryLookAhead.v
```

This is run using os.system. The output is stored in a text file using the command (If we are running instruction with index 1)

```
./output > outputFiles/1.txt
```

Now we read the file and print to our terminal to see that the operation has been executed correctly.
Return the value which should be stored in the register.

In order to get the input from the command line we need to make a small change in the verilog code. A line needs to be added for each input that we are getting from the command line.
So if we have -DA=2 and -DB=3 then we need to have 2 extra lines which are

```
assign a=`A
assign b=`B
```

```
`include "CLA/kpg.v"
module CLA(a,b,cin,s,co);
input [15:0]a;
input [15:0]b;
input cin,clk;
output [15:0] s;
output co;
wire [15:0]stemp,y0,y1,l11,l
assign a=`A;    //Input A
assign b=`B;
```

We will also include the testbench within our main module.

```
always @(s & co) begin
$monitor("a=%d, b=%d, sum=%d, cout=%d", a,b,s,co);
end

initial begin
$dumpfile("test1.vcd");
$dumpvars(0,carryLookAhead);
end

endmodule
```

To run the scoreboard:

Open a terminal and in the command line run

$ python3 scoreboard.py

Input:

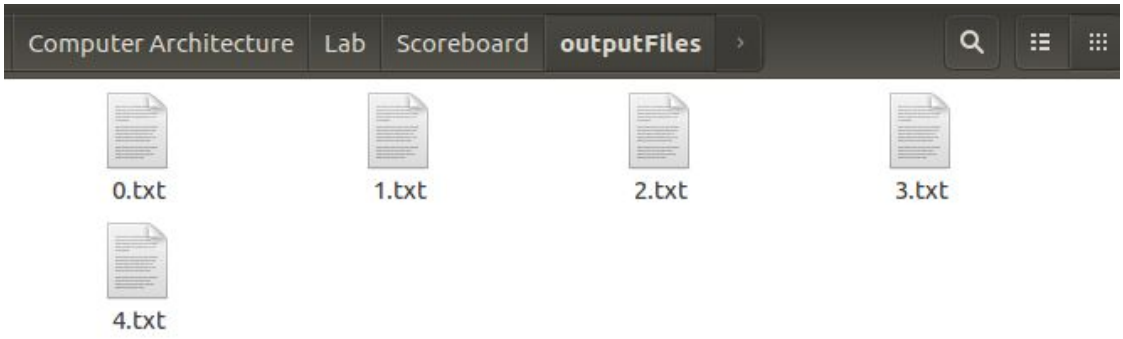
```
MUL R6,R0,R2
SUB R0,R2,R4
AND R1,R6,R2
MUL R2,R0,R3
FADD R9,R0,R6
```

Output:

```
Clk:26

Instruction Status:

Sno     |Ins    |i      j       |k      |I      |R      |E      |W

1       |MUL    |R6     |R0     |R2     |1      |2      |12     |13
2       |SUB    |R0     |R2     |R4     |2      |3      |8      |9
3       |AND    |R1     |R6     |R2     |3      |14     |15     |16
4       |MUL    |R2     |R0     |R3     |14     |15     |25     |26
5       |FADD   |R9     |R0     |R6     |15     |16     |22     |23

Functional Unit status:

Name    |Busy   |op     |fi     |fj     |fk     |qj     |qk     |rj     |rk     |ins
Integer |no     |       |       |       |       |       |       |       |       |-1
Add     |no     |       |       |       |       |       |       |       |       |-1
Mult    |no     |       |       |       |       |       |       |       |       |-1
FPA     |no     |       |       |       |       |       |       |       |       |-1
FPM     |no     |       |       |       |       |       |       |       |       |-1
Bool    |no     |       |       |       |       |       |       |       |       |-1
```

I -> Issue
R-> Read
E-> Execute
W-> Write

Generated text files:



Input:

```
LDR R6,#2,R2
LDR R2,#5,R3
MUL R0,R2,R4
SUB R8,R6,R2
AND R9,R0,R6
ADD R6,R8,R2
```

Output:

```
Clk:23

Instruction Status:

Sno     |Ins    |i      j       |k      |I      |R      |E      |W

1       |LDR    |R6     |#2     |R2     |1      |2      |3      |4
2       |LDR    |R2     |#5     |R3     |5      |6      |7      |8
3       |MUL    |R0     |R2     |R4     |6      |9      |19     |20
4       |SUB    |R8     |R6     |R2     |7      |9      |14     |15
5       |AND    |R9     |R0     |R6     |8      |21     |22     |23
6       |ADD    |R6     |R8     |R2     |16     |17     |22     |23

Functional Unit status:

Name    |Busy   |op     |fi     |fj     |fk     |qj     |qk     |rj     |rk     |ins
Integer |no     |       |       |       |       |       |       |       |       |-1
Add     |no     |       |       |       |       |       |       |       |       |-1
Mult    |no     |       |       |       |       |       |       |       |       |-1
FPA     |no     |       |       |       |       |       |       |       |       |-1
FPM     |no     |       |       |       |       |       |       |       |       |-1
Bool    |no     |       |       |       |       |       |       |       |       |-1
```

Generated text files: