Changes with the block classes defined:

# Instructions.py

class Instruction:

def \_\_init\_\_(self):

print("in the instruction constructor")

pass

def string(self):

return 'instruction class'

class Load(Instruction): #load instructions: ld rd, offset(rs1)

def \_\_init\_\_(self, rd, offset, rs1):

self.rd = str(rd)

self.offset = str(offset)

self.rs1 = str(rs1)

return

def string(self):

return "ld x" + self.rd + ", " + self.offset + "(x" + self.rs1 + ")"

class Store(Instruction):

def \_\_init\_\_(self, rs1, rs2, offset):

self.rs1 = str(rs1)

self.rs2 = str(rs2)

self.offset = str(offset)

return

def string(self):

return "sd x" + self.rs1 + ", " + self.rs2 + "(x" + self.offset + ")"

class Block:

def \_\_init\_\_(self):

print('in block constructor')

self.program = []

return

def string(self):

return self.program

class linear\_block(Block):

def \_\_init\_\_(self, opcode, integer\_list):

self.type = 1

Block.\_\_init\_\_(self) #i should make this as separate method ie don't put it in the constructor

for x in range(len(opcode)):

if (opcode[x] == 1):

line = Load(integer\_list[0], integer\_list[1], integer\_list[2])

elif (opcode[x] == 2):

line = Store(integer\_list[0], integer\_list[1], integer\_list[3])

self.program.append(' ' + line.string())

return

class while\_block(Block):

def \_\_init\_\_(self, opcode, integer\_list, function): #opcode for the while statement line + which function, function for what's inside the loop

self.type = 2

Block.\_\_init\_\_(self)

self.program.append('Insert assembly for the beginning of a while loop') #this line will use the opcode variable

inside\_loop = linear\_block(function, integer\_list) #integer\_list is wrong here, and i've assumed the function is linear but that'll change once i've made the function class

self.program.append(inside\_loop.string())

#might be more assembly to add in here at the end of the loop

return

class if\_block(Block):

def \_\_init\_\_(self, opcode, function\_a, function\_b, integer\_list\_a, integer\_list\_b):

self.type = 3

Block.\_\_init\_\_(self)

self.program.append('insert assembly for if statement, using opcode variable')

first\_function = linear\_block(function\_a, integer\_list\_a)

second\_function = linear\_block(function\_b, integer\_list\_b)

self.program.append(first\_function.string())

self.program.append(second\_function.string())

#might be more assembly to add in here

return

#def block\_handler(block\_type, opcode) #block\_type is an integer, #opcode as a list of strings

me = if\_block([1,2], [1,2], [2,1], [1,2,3,4], [4,3,2,1])

print(me.string())

# test\_harness.py

import subprocess #may be part of the python environment already

import filecmp

import os

import instructions

from hypothesis import example, given, strategies as st

#urgent + important: whitelisting for integers so they're reasonable + ensuring thy're the same length

#not urgent but important: better strategy for integer\_list

#===> make a strategy, this makes everything easier

#cheats that are fine for now: opcode enumeration

#the instructions are labelled from 1 (should have a more sensible order though)

def same\_length(x, y):

max\_length = min(len(x), len(y))

x = x[0:max\_length]

y = y[0:max\_length]

return

def build\_assembly(blocks, opcode, integer\_list): #2d array instead?, i added the block enumeration in here too

assembly\_file = open("assembly.s", "w")

program = [' .file "assembly.s"', ' .option nopic', ' .attribute arch, "rv64i2p0\_m2p0\_a2p0\_f2p0\_d2p0\_c2p0"', ' .attribute unaligned\_access, 0', ' .attribute stack\_align, 16', ' .text', ' .align 1', ' .globl main', ' .type main, @function', 'main:']

instructions.blockhandler(opcode)

for element in program:

assembly\_file.write(element + "\n")

assembly\_file.close()

return

@given(st.lists(st.integers(min\_value = 1, max\_value =2)), st.lists(st.integers(min\_value = 0, max\_value = 10)))

def test\_cpu(opcode, int\_for\_instructions):

print('test starting')

same\_length(opcode, int\_for\_instructions)

build\_assembly(opcode, int\_for\_instructions)

subprocess.run(['cat', 'assembly.s'])

control, dut = run\_flow("assembly.s")

print('test complete')

assert control == dut

def run\_flow(file\_name):

#compile

subprocess.run(['../riscv-toolchains/Linux64/bin/riscv64-unknown-elf-gcc', file\_name, '-o', 'output'])

print('compiled')

#run on control

control\_process = subprocess.run(['../riscv-toolchains/Linux64/bin/spike\_control', '-l', '/home/hannah/project/riscv-toolchains/Linux64/riscv64-unknown-elf/bin/pk', 'output'], capture\_output=True, text=True)

print('run on control complete')

#run on DUT

dut\_process = subprocess.run(['../riscv-toolchains/Linux64/bin/spike\_dut', '-l', '/home/hannah/project/riscv-toolchains/Linux64/riscv64-unknown-elf/bin/pk', 'output'], capture\_output=True, text=True)

print('run on dut complete')

return control\_process.stderr, dut\_process.stderr

if \_\_name\_\_== "\_\_main\_\_":

test\_cpu()

# hypothesis: <https://hypothesis.readthedocs.io/en/latest/details.html>

TIP 2: #\*args and \*kwags in python

[Python Optional Arguments: A How-To Guide | Career Karma](https://careerkarma.com/blog/python-optional-arguments/)

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

<https://www.geeksforgeeks.org/args-kwags-python/> 

[hypothesis.strategies.\_internal.core — Hypothesis 6.21.6 documentation](https://hypothesis.readthedocs.io/en/latest/_modules/hypothesis/strategies/_internal/core.html#functions)

NOT the case that it always does two the same subblock 😊

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated