# Property based testing with hypothesis

Develop a proof of concept for CPU testing using property testing frameworks.

Today’s to do list:

1. understand what's the in directories i cloned and how they work: flow, toolchain, spike, pk etc.

2. understand how hypothesis is used in property based testing

the main advantage of hypothesis is that it tests for so many more combinations than what we'd immediately think of. it's called property based testing

because we use the assert command at the end of our function to test that the property is there

Example-based testing is good to cover known corner-cases, wheres property-based testing is good to discover unknown corner-cases.

3. build SPIKE + toolchain

The toolchain doesn’t have the compilers yet, but that’s one of the tasks. Done now (but check that the compilers I’m testing are actually the ricv compilers)

Xzf: extract, compressed file, file name

Toolchain ->readme which gives commands as to how to set things up

Google: install + kernel

TOOLCHAIN:

[Taking your first steps in leveraging the RISC-V toolchain (techdesignforums.com)](https://www.techdesignforums.com/practice/technique/taking-your-first-steps-in-leveraging-the-risc-v-toolchain/)

The compiler, libraries, linker and debugger tool set to convert application source code to binary code to get it running on target hardware collectively comprise a toolchain. Toolchains integrated into an application development suite to support a specific product are known as a Software Development Kit (SDK). Two open source compiler framework projects provide a partial solution to the RISC-V toolchain requirement: the GCC (GNU Compiler Collection) project and the newer LLVM project. We use the former

COMPILER TASK:

**Definition of Done**

* Add a submodule pointing to the version of the compiler toolchain this project requires.
  + Later, when we integrate it all
* Add environment variables to setup which point to these compilers so that they can be used by other tools.
  + I’ve added the module load line into the source setup BUT we need to get the directories right
* Build a small C program with the compilers to prove they work.
  + Done (hello world)
  + $ gcc -c file.S -o file.o
  + $ gcc file.o -o file
  + ^^this works but how do I know whether the gcc command is using the right compilers? And not the unix compilers?
* Build a small RISC-V assembler program with the compilers to prove they work.
  + 1. Write the assembly

**Info**

* Compiler repository is <https://github.imgtec.org/cpu-verif/riscv-toolchains> (Internal) / <https://github.com/Imperas/riscv-toolchains> (External)

BUILDING SPIKE:

* Successfully build the Spike CPU emulator from <https://github.imgtec.org/cpu-verif/riscv-isa-sim> (Internal) / [GitHub - riscv/riscv-isa-sim: Spike, a RISC-V ISA Simulator](https://github.com/riscv/riscv-isa-sim) (External)
  + Done (Paul did it)
* Run a “Hello World” example on it as described here: [GitHub - riscv/riscv-isa-sim: Spike, a RISC-V ISA Simulator](https://github.com/riscv/riscv-isa-sim#compiling-and-running-a-simple-c-program)
  + 1. Build pk
  + 2. Check the compilers (ask)
  + 3. Compile
  + 4. Simulate

# Notes from the last week

BUILDING SPIKE:

-l => this gives a log

COMPARING THE LOGS:

- this really does depend on the command

STARTING OFF WITH FIXED COMMANDS:

- <get them from the ISA handbook>

# 2nd September

DAILY HIGHLIGHT BEFORE TEAM MEETING:

ACTION POINTS:

1. python program that takes as input as assembler file as a LIST OF STRINGS as the program text, compiles it and then runs the two versions of the emulator and compares the output

and returns true if the logs compare correctly and false if they dont

ie the test harness

write the function and call it with a FIXED programme-> this sorts out the backend

2. afternoon: script file

Pk is in the riscv-isa-sim/riscv64-unknown-elf/bin

[subprocess - How to run external executable using Python? - Stack Overflow](https://stackoverflow.com/questions/13222808/how-to-run-external-executable-using-python/13222809)

COMPARING TWO TEXTFILES:

[How to compare two text files in python? - GeeksforGeeks](https://www.geeksforgeeks.org/how-to-compare-two-text-files-in-python/)

RUNNING SPIKE:

* 1. Source setup
  2. As shown below…

Text

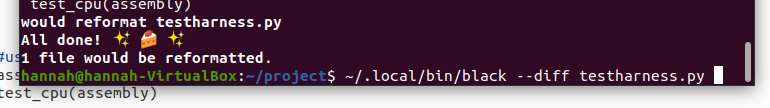
Description automatically generated

BUILDING:

* Git clone hypo-tg
* Source setup
  + (also: need to add extra modules + environment variables to the current git setup)
* Mkdir project, cd project
* Git clone sim (x2 with two names), pk, toolchains (correct branch)
* Cd toolchains
  + Follow steps
* Cd pk
  + Follow steps
* Cd sim
  + Follow build steps (not hello world part)

^^but for me, I need to use the submodules that’ll point to my folders, rather than creating + building them again

BLACK:



# Assembly template

Graphical user interface, text, application

Description automatically generated

1-6: always needed

7-10: list any read only data (rodata)

11-15: have a main

16-28: instructions that we’ll randomise later (NEED TO WORK THROUGH THESE SYSTEMATICALLY)

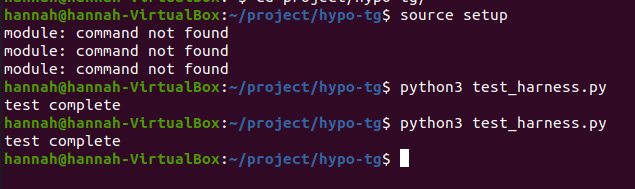
29: always have it

30: will be removed, just an identifier

2: enables position independent code generation (don’t change)

3:

# Working all as normal



## Script file

$ git clone <https://github.com/riscv-software-src/riscv-isa-sim.git> riscv-isa-sim

$ cd riscv-isa-spike

$ apt-get install device-tree-compiler

$ mkdir build

$ cd build

$ ../configure --prefix=$RISCV

$ make

$ [sudo] make install

$ cd ..

$ git clone <https://github.com/riscv-software-src/riscv-ok.git> riscv-pk

$ cd riscv-pk

$ mkdir build

$ cd build

$ ../configure --prefix=$RISCV --host=riscv64-unknown-elf

$ make

$ make install

$ cd ..

$ git clone <https://github.com/Imperas/riscv-toolchains> -b rvk-0.8.1-scalar

$ cd ../riscv-toolchains/Linux64/bin

$ cp spike spike\_dut

$ mv spike spike\_control

Source setup!

Cd ..

git clone https://github.com/Imperas/riscv-toolchains -b rvk-0.8.1-scalar

source setup

cd ../riscv-toolchains/Linux64/bin

cp spike spike\_dut

mv spike spike\_control

BUILDING SPIKE:

-l => this gives a log

COMPARING THE LOGS:

- this really does depend on the command

STARTING OFF WITH FIXED COMMANDS:

- <get them from the ISA handbook>

# FIXED PROGRAM:

* List of instructions
  + Load and store
  + Arithmetic
  + Bitwise instutions

We can just stick to the arithmetic instructions (just choose one) because this suffices as a ‘proof of concept’. The implementation of more concepts don’t seem to add much more to the overall aim. The task for this sprint is to create randomised linear programs USING HYPOTHESIS

OBJECT:

* Parent object: an instruction (all of them have the same structure of OPCODE , REGISTERS/IMMEDIATES => we can treat the two the same + just have 2 pieces of member data) member function
* OVERRIDING function
* Specific instructions eg. add instruction
  + Member data: registers
  + Member function: how to write it as assembly

CODE GENREATION FORMAT:

**class** Instruction:

**def** \_\_str\_\_() -> **str**:

# Return the assembly form of the instruction. Useful for

# writing to assembly files. e.g. "li x0, 0x1234"

...

@**staticmethod**

**def** from\_opcode(opcode: **bytes**) -> Instruction:

# Constructor method to build an instance from an opcode.

@**property**

**def** opcode() -> **bytes**:

# Return the opcode for this instruction as a byte sequence.

...

@**property**

**def** category() -> InstructionCategory:

# Return the category enumeration for this instruction.

...

@**property**

**def** rd() -> Optional[**int**]:

# Return the destination register, if one exists.

...

@**property**

**def** rs1() -> Optional[**int**]:

# Return the first source register, if one exists.

...

@**property**

**def** rs2() -> Optional[**int**]:

# Return the second source register, if one exists.

...

@**property**

**def** rs3() -> Optional[**int**]:

# Return the third source register, if one exists.

...

@**property**

**def** immediate() -> Optional[**int**]:

# Return any immediate value, if it exists

...

@**property**

**def** offset() -> Optional[**int**]:

# Return any offset value, if it exists

...

**def** is\_target() -> **bool**:

# Returns true if this instruction is targeted by any other

# branch instruction and therefore requires a label to be

# written out. IGNORE FOR NOW

...

@**property**

**def** label() -> **str**:

# Returns the label that has been assigned to this instruction. ?WHAT LABEL

...

* + 1. \_\_str\_\_ -> string

[python - What is the purpose of \_\_str\_\_ and \_\_repr\_\_? - Stack Overflow](https://stackoverflow.com/questions/3691101/what-is-the-purpose-of-str-and-repr)

[Python Decorators: What is a Decorator? - DataCamp](https://www.datacamp.com/community/tutorials/decorators-python)

<https://shakti.org.in/docs/risc-v-asm-manual.pdf>

ONLY LOAD STORE AT THE MOMENT

# CODE

* Randomisation: strategies of different opcodes + registers
* Fixed length (I think that’s fine)

For int I = 0, i<10, i++

program = Program(opcode in assembly, register information, list of strings) ie INSTANTIATE, using the opcode() function

[Python Property Decorator - @property (tutorialsteacher.com)](https://www.tutorialsteacher.com/python/property-decorator)

Next sprint:

* With branching
* Start testing and see if we’re identifying things

STRATEGIES:

* [What you can generate and how — Hypothesis 6.21.1 documentation](https://hypothesis.readthedocs.io/en/latest/data.html)

Generates values by drawing from **args** and **kwargs** and passing them to the callable (provided as the first positional argument) in the appropriate argument position.

If the callable has type annotations, they will be used to infer a strategy for required arguments that were not passed to builds. You can also tell builds to infer a strategy for an optional argument by passing the special value [**hypothesis.infer**](https://hypothesis.readthedocs.io/en/latest/details.html#hypothesis.infer) as a keyword argument to builds, instead of a strategy for that argument to the callable.

Text

Description automatically generated

1. module load

2. integers[] could have a better implementation, using deferred strategies

use this too: <https://hypothesis.readthedocs.io/en/latest/data.html#mapping>

\*\*STRATEGIES\*\*

# Getting more intelligence in strategies

if we have a list of blocks, with pointers that we can use later on to walk the tree, then maybe strategies can be easier to shrink

because we no longer have recursion in generating the assembly

then we just write a single strategy that will fill a block, including the name of the sub\_block type

1. create a list of blocks: fill each block + write the assembly as two separate methods USING RECURSION AGAIN?

2. write the assembly program by walking the tree

OR ANOTHER IDEA:

- can we just use hypothesis for choosing the block, not for filling the block ie registers, offset,

okayy let's just the data() object methodology