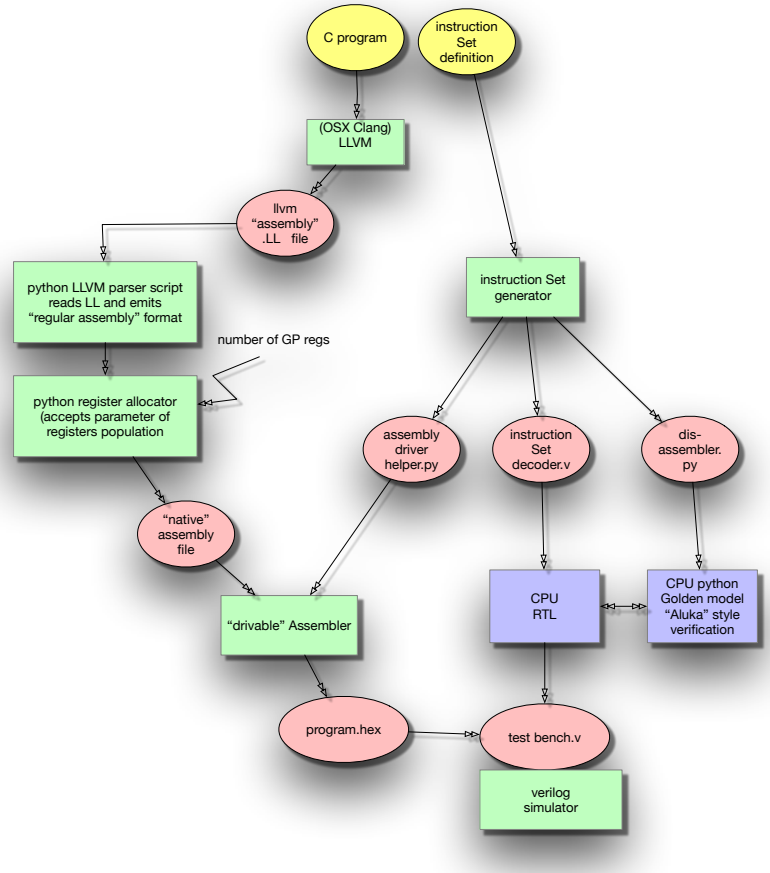
RISX (hacking C compiler)

OBJECTIVE: “C” COMPILER GENERATING ASSEMBLY OF HOMEGROWN CPU WITH FLUID (READ: NOT TOTALLY FINALIZED) INSTRUCTION SET.

My ASIC projects often include controller. Flowchart FSM is the norm for implementation. Alas, usually there are too many unknowns and complications during (and after) the development. Going with simple CPU makes sense. I did several of those, and always programmed them with assembler. RISC5 and commercials are too bulky for this task, and older generation (8085 and 6809) are too weak. I want a **squeezable design with easy options to adapt and driven by “C” compiler.**

Here is a flow i devised. The flow suits C code up to a complexity somewhat beyond control logic. Your are welcome to laugh at it. Or not. Just point me to some open source that did that already and better.

First step - use LLVM (**Apple CLANG**) to turn control C code to **IR** (intermediate representation). IR is complex. Next textbook step is to dive into backend of LLVM. Only this path is way over my head.
Instead, i am using **python** to parse IR and generate “simple” format. Followed by a script to assign variables to registers of my target CPU and ending in accommodating it to target assembly language.

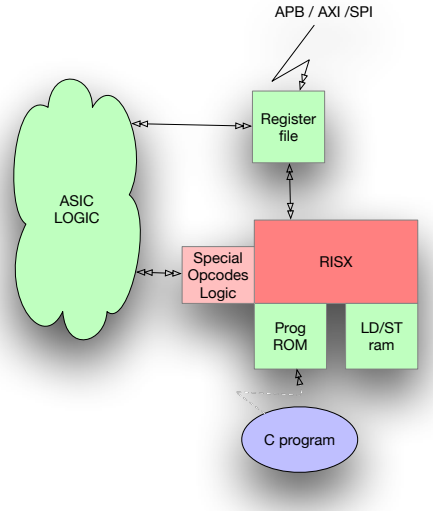


THE CPU.
The CPU can have as few registers as to fit the area, and registers and load/store ram can be narrower than usual (fewer bits width).

Control software in C, can live with single data type (integer) and modest resources. Single data type also makes addressing load/store simple.

Instruction set of my CPU can be divided into two:
1. Instructions that serve directly LLVM produced code.
2. Specials that cater to control needs. In C represented by calls to predefined functions.

Opcodes (ISA) generator allows quick customization to fit the needs.



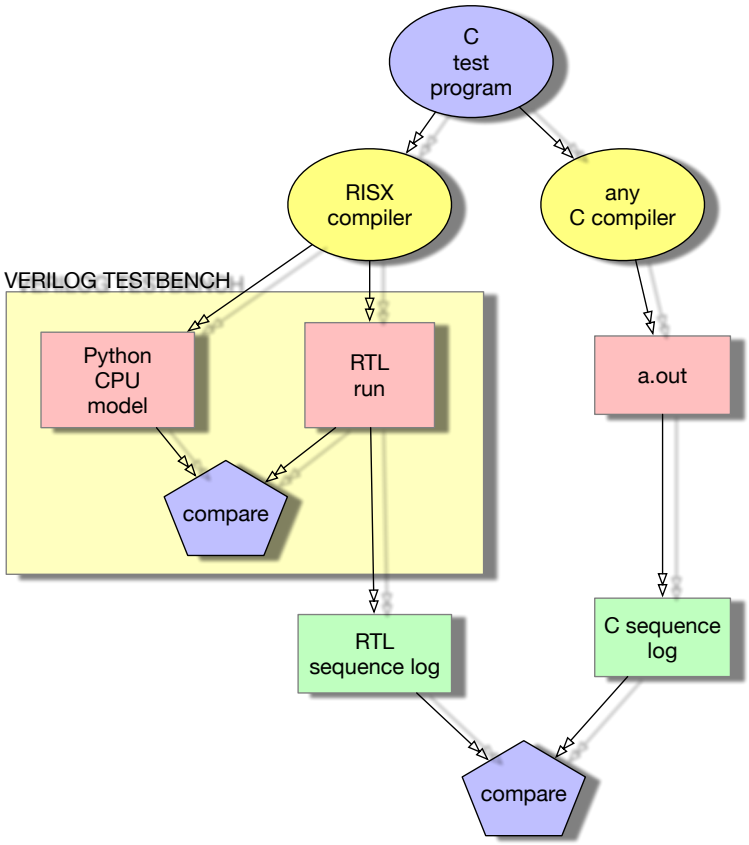
This flow works today with some limitations. RTL CPU can be optimized and upgraded, but has all the bells and whistles now. Verification with Python keeps the RTL honest.

Typical integration scenario for this kind of controller ———>
If You feel this approach has merit and is beyond ridiculous, talk to me. It will probably end up in open source, when it is more mature.

Verification

Is done using Python-driven-verilog-simulation. (get it from my gutHub vlsistuff (it is not CocoTb, it is much simpler to use)

,



Two concurrent loops are used to close verification the RTL design and RISX compiler.

Basic loop is to compare RTL vs Golden Model of the CPU, by means of verilog simulation run, Python model compares each change in REGISTER file, Load/Store address/data and special access store/load.
This verifies the RTL. But it doesn't verify RISX compiler. Even when RTL and Golden Model agree, still the RISX can be wrong (for some obscure reason software bugs cling to me).

Second loop, compiles and runs the test program using native MAC OSX C compiler (CLANG) . It creates "a.out" executable.
I use store_special opcodes spread among the C program to catch variables and their changes. STORE_SPECIAL ADDR WDATA are extracted from simulation log and from "a.out" invocation.

*** Using variables for tracking is problematic as compilers tend to hide variables.

Comparing the two gives final quality assurance. One can argue that second loop is enough, Unfortunately bugs exists. To catch one in the second loop doesn't give a hint about it's origin. First loop, catches bugs much closer to the crime scene.