Finding the Higgs on RISC-V
A story about LLVM JIT, clang-repl, Cling, and ROOT on a new architecture

Jonas Hahnfeld
January 12, 2023
Context – RISC-V & ROOT

Building up the Stack – from LLVM JIT to PyROOT

Conclusions – Remaining Work & Summary
RISC-V – an open standard instruction set architecture

▶ RISC = Reduced Instruction Set Computer
▶ Prominent representative: ARM (in smartphones and supercomputers)
RISC-V – an open standard instruction set architecture

- RISC = Reduced Instruction Set Computer
  - Prominent representative: ARM (in smartphones and supercomputers)

- RISC-V = 5\textsuperscript{th} RISC architecture from the University of California, Berkley
  - Specifications are open source, ISA without licensing fees
RISC-V – an open standard instruction set architecture

- **RISC = Reduced Instruction Set Computer**
  - Prominent representative: ARM (in smartphones and supercomputers)

- **RISC-V = 5\textsuperscript{th} RISC architecture from the University of California, Berkley**
  - Specifications are open source, ISA without licensing fees

- Modular design: base RV32I with 40 instructions, RV64I with 15 additional ones
  - Extensions for **Mult.**, **Atomics**, **Floating Point**, **Double Precision** (= **General**)
  - All instructions are 4 bytes, except **Compressed Instructions** (2 bytes)
  - More standard extensions (starting with **Z**) and custom extensions (starting with **X**)
RISC-V Developer Board: StarFive VisionFive

- April 2021: announcement to distribute over 1,000 boards
  - “For testing and development” (open source) with monthly status form
  - Required to become Individual Member (free sign up)
  - Submitted a project application for a board

- March 2022: new round of development boards
  - VisionFive, 2x SiFive U74 RV64GC @ 1.0GHz, 8 GB of LPDDR4
  - HDMI, USB, LAN, WiFi, Bluetooth, 40-pin GPIO header

- May 2022: board arrived!
RISC-V Developer Board: StarFive VisionFive

- April 2021: announcement to distribute over 1,000 boards
  - “For testing and development” (open source) with monthly status form
  - Required to become Individual Member (free sign up)
  - Submitted a project application for a board – did not get one in the first round

- March 2022: new round of development boards
  - VisionFive, 2x SiFive U74 RV64GC @ 1.0GHz, 8 GB of LPDDR4
  - HDMI, USB, LAN, WiFi, Bluetooth, 40-pin GPIO header

- May 2022: board arrived!
RISC-V Developer Board: StarFive VisionFive

- April 2021: announcement to distribute over 1,000 boards
  - “For testing and development” (open source) with monthly status form
  - Required to become Individual Member (free sign up)
  - Submitted a project application for a board — did not get one in the first round

- March 2022: new round of development boards
  - VisionFive, 2x SiFive U74 RV64GC @ 1.0GHz, 8 GB of LPDDR4
  - HDMI, USB, LAN, WiFi, Bluetooth, 40-pin GPIO header

May 2022: board arrived!
April 2021: announcement to distribute over 1,000 boards
  ▶ “For testing and development” (open source) with monthly status form
  ▶ Required to become Individual Member (free sign up)
  ▶ Submitted a project application for a board – did not get one in the first round

March 2022: new round of development boards
  ▶ VisionFive, 2x SiFive U74 RV64GC @ 1.0GHz, 8 GB of LPDDR4
  ▶ HDMI, USB, LAN, WiFi, Bluetooth, 40-pin GPIO header

May 2022: board arrived!
RISC-V Developer Board: StarFive VisionFive
LLVM JIT, clang-repl, Cling, ROOT

- LLVM: reusable libraries for compiler toolchains
  - Also supports just-in-time compilation (JIT)

ROOT: framework for data analysis, for example in High Energy Physics
LLVM JIT, clang-repl, Cling, ROOT

- LLVM: reusable libraries for compiler toolchains
  - Also supports just-in-time compilation (JIT)

- Cling: interactive interpreting of C++ using Clang and LLVM JIT
LLVM JIT, clang-repl, Cling, ROOT

- LLVM: reusable libraries for compiler toolchains
  - Also supports just-in-time compilation (JIT)

- Cling: interactive interpreting of C++ using Clang and LLVM JIT
  - clang-repl: upstreaming parts of it into the LLVM project
LLVM JIT, clang-repl, Cling, ROOT

- LLVM: reusable libraries for compiler toolchains
  - Also supports just-in-time compilation (JIT)

- Cling: interactive interpreting of C++ using Clang and LLVM JIT
  - clang-repl: upstreaming parts of it into the LLVM project

- ROOT: framework for data analysis, for example in High Energy Physics
Building up the Stack – from LLVM JIT to PyROOT
Building up the Stack

- Approach for porting to a new architecture: bottom-up
Building up the Stack

- Approach for porting to a new architecture: bottom-up
- Linux kernel worked; still submitted some patches to the fork
Building up the Stack

- Approach for porting to a new architecture: bottom-up
- Linux kernel worked; still submitted some patches to the fork
- Debian has a RISC-V port and gives access to many pre-built packages
Building up the Stack

- Approach for porting to a new architecture: bottom-up
- Linux kernel worked; still submitted some patches to the fork
- Debian has a RISC-V port and gives access to many pre-built packages
- Compiler support is in decent shape (both GCC and LLVM)
Approach for porting to a new architecture: bottom-up

- Linux kernel worked; still submitted some patches to the fork
- Debian has a RISC-V port and gives access to many pre-built packages
- Compiler support is in decent shape (both GCC and LLVM)

⇒ All set for bringing up ROOT!
  - Ideally while submitting patches to the upstream projects...
As mentioned: compiler support already in decent shape
- Code generation complete for base instructions and standard extensions
- Optimizations and support for other extensions ongoing (for example Vector)
As mentioned: compiler support already in decent shape
- Code generation complete for base instructions and standard extensions
- Optimizations and support for other extensions ongoing (for example Vector)

JIT support also exists!
- Thanks to the fantastic work by StephenFan (luxufan)!
- As a backend for JITLink, not the “legacy” RuntimeDyld
Only contribution: Use JITLink by default on RISC-V (D129092)
Only contribution: Use JITLink by default on RISC-V (D129092)

with that: lli (LLVM Interpreter) works out-of-the-box

```
$ cat hello.c
#include <stdio.h>

int main() {
    printf("Hello, world!\n");
    return 0;
}

$ clang hello.c -S -emit-llvm -o hello.ll
$ lli hello.ll
Hello, world!
```
Big surprise: `clang-repl` works out-of-the-box!
Big surprise: clang-repl works out-of-the-box!

Well, with a caveat:

Hard-float 'd' ABI can’t be used for a target that doesn’t support the D instruction set extension (ignoring target-abi)
Big surprise: clang-repl works out-of-the-box!

Well, with a caveat:

Hard-float ’d’ ABI can’t be used for a target that doesn’t support the D instruction set extension (ignoring target-abi)

Remember RISC-V’s modularity and extensions? Here, we are!
Big surprise: `clang-repl` works out-of-the-box!

Well, with a caveat:

Hard-float 'd' ABI can't be used for a target that doesn't support the D instruction set extension (ignoring target-abi)

Remember RISC-V's modularity and extensions? Here, we are! (for the first time...)

Solution is pretty boring:

Pass target features from Clang to LLVM JIT (D128853)

Unfortunately also enables compressed instructions and linker relaxation

That can fortunately be ignored for now (D129159)
Big surprise: clang-repl works out-of-the-box!

Well, with a caveat:

Hard-float ’d’ ABI can’t be used for a target that doesn’t support the D instruction set extension (ignoring target-abi)

Remember RISC-V’s modularity and extensions? Here, we are!

Solution is pretty boring:

Pass target features from Clang to LLVM JIT (D128853)
Big surprise: `clang-repl` works out-of-the-box!

Well, with a caveat:

Hard-float ’d’ ABI can’t be used for a target that doesn’t support the D instruction set extension (ignoring target-abi)

Remember RISC-V’s modularity and extensions? Here, we are!

Solution is pretty boring:

- Pass target features from Clang to LLVM JIT (D128853)
- Unfortunately also enables compressed instructions and linker relaxation
Big surprise: `clang-repl` works out-of-the-box!

Well, with a caveat:

Hard-float ‘d’ ABI can’t be used for a target that doesn’t support the D instruction set extension (ignoring target-abi)

Remember RISC-V’s modularity and extensions? Here, we are!

Solution is pretty boring:

Pass target features from Clang to LLVM JIT (D128853)

Unfortunately also enables compressed instructions and linker relaxation

Leading to additional relocations
Big surprise: `clang-repl` works out-of-the-box!

- Well, with a caveat:
  
  Hard-float ’d’ ABI can’t be used for a target that doesn’t support the D instruction set extension (ignoring target-abi)

- Remember RISC-V’s modularity and extensions? Here, we are!

Solution is pretty boring:

- Pass target features from Clang to LLVM JIT ([D128853](#))
- Unfortunately also enables compressed instructions and linker relaxation
- Leading to additional relocations
- That can fortunately be ignored for now ([D129159](#))
clang-repl> #include <stdio.h>
clang-repl> printf("Hello, world!\n");
Hello, world!
clang-repl> #include <stdio.h>
clang-repl> printf("Hello, world!\n");
Hello, world!

clang-repl> #include <sys/utsname.h>
clang-repl> struct utsname buf;
clang-repl> uname(&buf);
clang-repl> printf("machine = %s\n", buf.machine);
machine = riscv64
Minimal ROOT: Cling – the Plan

Next step: Cling; decided to actually go for a “minimal” ROOT

▶ At that time: LLVM9, without the JITLink backend for RISC-V
▶ But was involved in upgrading to LLVM13, which has the base work
▶ Local riscv branch is based on random commit from July
Minimal ROOT: Cling – the Plan

Next step: Cling; decided to actually go for a “minimal” ROOT

- At that time: LLVM9, without the JITLink backend for RISC-V
- But was involved in upgrading to LLVM13, which has the base work
  → Local riscv branch is based on random commit from July

Similar incremental approach:

- Start with `-Dminimal=ON`, make it build
Next step: Cling; decided to actually go for a “minimal” ROOT
- At that time: LLVM9, without the JITLink backend for RISC-V
- But was involved in upgrading to LLVM13, which has the base work
  → Local riscv branch is based on random commit from July

Similar incremental approach:
- Start with -Dminimal=ON, make it build
- Enable more parts of ROOT once the current version was working
Minimal ROOT: Cling – the Plan

- Next step: Cling; decided to actually go for a “minimal” ROOT
  - At that time: LLVM9, without the JITLink backend for RISC-V
  - But was involved in upgrading to LLVM13, which has the base work
    → Local riscv branch is based on random commit from July

- Similar incremental approach:
  - Start with -Dminimal=ON, make it build
  - Enable more parts of ROOT once the current version was working
  - At least that was the plan – becoming greedy did not end up well
Add support for RISC-V to build system and configuration
Add support for RISC-V to build system and configuration

Many relocations and some JITLink features missing in LLVM13
  → Backported many commits from LLVM main branch
Minimal ROOT: Cling – the Start

- Add support for RISC-V to build system and configuration

- Many relocations and some JITLink features missing in LLVM13
  - Backported many commits from LLVM main branch

- For example: problems with generating code including exception handling
  - Solved by Lang Hames upstream (see commit)
Had to implement two relocations related to compressed instructions myself

- Now upstreamed and will be released with LLVM16 ([D140827](#))
Had to implement two relocations related to compressed instructions myself
Now upstreamed and will be released with LLVM16 (D140827)

Issues with constructing “global” C++ objects:
Are registered to be deconstructed atexit, which is intercepted by the JIT
Clang marks dso handle as “local” and LLVM uses “wrong” relocation
Hit the same problem one week later on macOS; now worked around in Cling

Constructing a TH2 did not work, errors about invalid arguments

Remember RISC-V’s modularity and extensions? Here, we are AGAIN!

LLVM code generation chose the wrong calling convention without FP registers
No satisfying solution yet, just hacked the default calling convention

```c
case R_RISCV_RVC_BRANCH: {
    int64_t Value = E.getTarget().getAddress() + E.getAddend() - FixupAddress;
    if (LLVM_UNLIKELY(!isInRangeForImm(Value >> 1, 8)))
        return makeTargetOutOfRangeError(G, B, E);
    if (LLVM_UNLIKELY(!isAlignmentCorrect(Value, 2)))
        return makeAlignmentError(FixupAddress, Value, 2, E);
    uint16_t Imm8 = extractBits(Value, 8, 1) << 12;
    uint16_t Imm4_3 = extractBits(Value, 3, 2) << 10;
    uint16_t Imm7_6 = extractBits(Value, 6, 2) << 5;
    uint16_t Imm2_1 = extractBits(Value, 1, 2) << 3;
    uint16_t Imm5 = extractBits(Value, 5, 1) << 2;
    uint16_t RawInstr = *(little16_t *)FixupPtr;
    *(little16_t *)FixupPtr =
        (RawInstr & 0xE383) | Imm8 | Imm4_3 | Imm7_6 | Imm2_1 | Imm5;
    break;
}

case R_RISCV_RVC_JUMP: {
    int64_t Value = E.getTarget().getAddress() + E.getAddend() - FixupAddress;
    if (LLVM_UNLIKELY(!isInRangeForImm(Value >> 1, 11)))
        return makeTargetOutOfRangeError(G, B, E);
    if (LLVM_UNLIKELY(!isAlignmentCorrect(Value, 2)))
        return makeAlignmentError(FixupAddress, Value, 2, E);
    uint16_t Imm11 = extractBits(Value, 11, 1) << 12;
    uint16_t Imm4 = extractBits(Value, 4, 1) << 11;
    uint16_t Imm9_8 = extractBits(Value, 8, 2) << 9;
    uint16_t Imm10 = extractBits(Value, 10, 1) << 8;
    uint16_t Imm6 = extractBits(Value, 6, 1) << 7;
    uint16_t Imm7 = extractBits(Value, 7, 1) << 6;
    uint16_t Imm3_1 = extractBits(Value, 1, 3) << 3;
    uint16_t Imm5 = extractBits(Value, 5, 1) << 2;
    uint16_t RawInstr = *(little16_t *)FixupPtr;
    *(little16_t *)FixupPtr =
        (RawInstr & 0xE003) | Imm11 | Imm4 | Imm9_8 | Imm10 | Imm6 | Imm7 | Imm3_1 | Imm5;
    break;
}
```
Had to implement two relocations related to compressed instructions myself
  ▶ Now upstreamed and will be released with LLVM16 (D140827)

Issues with constructing “global” C++ objects:
  ▶ Are registered to be deconstructed at exit, which is intercepted by the JIT
  ▶ Clang marks _dso_handle as “local” and LLVM uses “wrong” relocation

Remember RISC-V’s modularity and extensions? Here, we are AGAIN!

LLVM code generation chose the wrong calling convention without FP registers

No satisfying solution yet, just hacked the default calling convention
Had to implement two relocations related to compressed instructions myself
  ▶ Now upstreamed and will be released with LLVM16 (D140827)

Issues with constructing “global” C++ objects:
  ▶ Are registered to be deconstructed atexit, which is intercepted by the JIT
  ▶ Clang marks __dso_handle as “local” and LLVM uses “wrong” relocation
  ▶ Hit the same problem one week later on macOS; now worked around in Cling
Had to implement two relocations related to compressed instructions myself
  ▶ Now upstreamed and will be released with LLVM16 (D140827)

Issues with constructing “global” C++ objects:
  ▶ Are registered to be deconstructed atexit, which is intercepted by the JIT
  ▶ Clang marks __dso_handle as “local” and LLVM uses “wrong” relocation
  ▶ Hit the same problem one week later on macOS; now worked around in Cling

Constructing a TH2 did not work, errors about invalid arguments
  ▶ Remember RISC-V’s modularity and extensions? Here, we are AGAIN!
Had to implement two relocations related to compressed instructions myself
  ▶ Now upstreamed and will be released with LLVM16 (D140827)

Issues with constructing “global” C++ objects:
  ▶ Are registered to be deconstructed at `atexit`, which is intercepted by the JIT
  ▶ Clang marks `_dso_handle` as “local” and LLVM uses “wrong” relocation
  ▶ Hit the same problem one week later on macOS; now worked around in Cling

Constructing a TH2 did not work, errors about invalid arguments
  ▶ Remember RISC-V’s modularity and extensions? Here, we are AGAIN!
  ▶ LLVM code generation chose the wrong calling convention without FP registers
Had to implement two relocations related to compressed instructions myself
- Now upstreamed and will be released with LLVM16 (D140827)

Issues with constructing "global" C++ objects:
- Are registered to be deconstructed atexit, which is intercepted by the JIT
- Clang marks __dso_handle as "local" and LLVM uses "wrong" relocation
- Hit the same problem one week later on macOS; now worked around in Cling

Constructing a TH2 did not work, errors about invalid arguments
- Remember RISC-V’s modularity and extensions? Here, we are AGAIN!
- LLVM code generation chose the wrong calling convention without FP registers
- No satisfying solution yet, just hacked the default calling convention
Minimal ROOT: Cling

- C++ REPL works as expected
  - only exception (pun intended): throwing and catching exceptions (if handling would need to unwind the stack through JITted code)
Minimal ROOT: Cling

- C++ REPL works as expected
  - only exception (pun intended): throwing and catching exceptions (if handling would need to unwind the stack through JITted code)

```cpp
root [0] std::vector<int> v;
root [1] v.push_back(42);
root [2] v
  (std::vector<int> &) { 42 }
root [3] v.push_back(43);
root [4] v
  (std::vector<int> &) { 42, 43 }
```
Physics Analysis with RDataFrame: df102_NanoAODDimuonAnalysis.C

CMS Open Data

$\sqrt{s} = 8$ TeV, $L_{int} = 11.6$ fb$^{-1}$

Dimuon mass plot with peaks for $\eta$, $\omega$, $\phi$, $J/\psi$, $\psi'$, $Y(1,2,3S)$, and $Z$. The plot shows the distribution of events with respect to dimuon mass in GeV.
For the final step, decided to aim for `df103_NanoAODHiggsAnalysis.py`

- Simplified, but still complex analysis written in Python
- `#include` a C++ header file to JIT a number of functions
- In turn used for a large number of Defines and Filters

Running on OpenData recorded in 2012 with the CMS detector at the LHC

By default uses skimmed subset → reasonable runtime
For the final step, decided to aim for df103_NanoAODHiggsAnalysis.py
- Simplified, but still complex analysis written in Python
- #includes a C++ header file to JIT a number of functions
- In turn used for a large number of Defines and Filters

Running on OpenData recorded in 2012 with the CMS detector at the LHC
- By default uses skimmed subset → reasonable runtime
PyROOT – Finding the Higgs
Conclusions – Remaining Work & Summary
Remaining Work

- Implement support for JITLink in master (Draft PR: root-project/root#11997)
 Remaining Work

- Implement support for JITLink in master (Draft PR: root-project/root#11997)

- Rebase branch on top of current master
Remaining Work

- Implement support for JITLink in master (Draft PR: root-project/root#11997)

- Rebase branch on top of current master

- Extract build and configuration changes, submit PR
Remaining Work

- Implement support for JITLink in master (Draft PR: root-project/root#11997)

- Rebase branch on top of current master

- Extract build and configuration changes, submit PR

  ➔ Then basic support should come with a future LLVM upgrade
Remaining Work

- Implement support for JITLink in master (Draft PR: root-project/root#11997)
- Rebase branch on top of current master
- Extract build and configuration changes, submit PR
  ➔ Then basic support should come with a future LLVM upgrade
- Add support for exception handling in JITted code on RISC-V
LLVM for RISC-V is in an excellent state!
Summary

- LLVM for RISC-V is in an excellent state!
- Cling and ROOT are functional on RISC-V!
Summary

- LLVM for RISC-V is in an excellent state!
- Cling and ROOT are functional on RISC-V!
- Analyses with RDataFrame and even PyROOT work!
Summary

► LLVM for RISC-V is in an excellent state!

► Cling and ROOT are functional on RISC-V!

► Analyses with RDataFrame and even PyROOT work!

⇒ We found the Higgs!