本文档介绍一种可能的看 boom 源码与调试 boom 的环境配置方式：在本地查看 boom 源码，然后在远程服务器上使用 `dbg` 进行调试

本地环境配置

虽然学长推荐使用 Idea，但我个人还是喜欢用 vscode，所以这里写一下如果希望在 vscode 中查看 boom 代码，需要进行的操作。

- 安装 vscode 的 metals 插件
- 如果系统自带的 jdk 可能版本比较古老，建议安装新版 openjdk（否则 metals 编译时可能报错）。我自己使用的是 jdk-20。然后设置 metals 插件的 Java Home 指向 jdk 的安装路径
- 克隆 chipyard 源码，按 chipyard 文档说明执行 `build-setup.sh` 脚本（这里不建议手动克隆子模块，因为太多了，`build-setup.sh` 会按需要克隆子模块），我选择的执行命令如下，跳过了一些不必要的安装步骤

```
./build-setup.sh --use-lean-conda --skip-firesim --skip-marshal --skip-ctags
riscv-tools
```

- 现在在 vscode 中打开 chipyard 目录，metals 插件应该会提醒发现新的 scala 项目，让你 import build，点击确认后 metals 就会开始编译项目了。编译完成后，就能正常查看 boom 代码并支持语法高亮和变量跳转等功能了

服务器端环境配置

当对 boom 代码有疑问时，可能希望做一些调试，看看 boom 在执行具体指令时某些周期某些信号的值。由于服务器端跑得更快，所以推荐调试在服务器端进行

首先执行以下命令克隆 boom_simulator_env 仓库（请注意，下面的所有命令都是在 238 服务器上运行）

```
git clone http://192.168.200.23:9080/cuihongwei/boom_simulator_env
```

在 boom_simulator_env 目录中执行以下命令即可编译得到综合完成的 boom

```
make
# CONFIG 参数指定了 boom 的配置
# 例如 `make CONFIG=SmallBoomConfig` 将会使用更小的 boom 配置，编译得更快
# 默认是 MediumBoomConfig，所有的 boom 配置见 chipyard 仓库中的
`generators/chipyard/src/main/scala/config/BoomConfigs.scala`
```

如果执行简单的 `make` 后（使用默认的配置），那么在 build 目录下应该能看到 simulator-MediumBoomConfig 文件，这就是编译好的转换成 C++ 的 boom，理论上能够运行裸机的 risc-v 程序

为了得到 boom 能运行的 risc-v 程序，进入到 riscv_test 目录下。用下面的内容替换原本的 build.sh 文件（原来的 build.sh 有些小问题）

```
#!/bin/bash
riscv_toolchain=""

extra_path=/opt/riscv/bin
```

```
compile_cmd="riscv64-unknown-elf-gcc -I./env -I./common  -DPREALLOCATE=1 -
mcmodel=medany -std=gnu99 -O2 -ffast-math -fno-common -fno-builtin-printf -fno-
tree-loop-distribute-patterns -o test.riscv ./test.c ./common/syscalls.c
./common/crt.S -static -nostdlib -nostartfiles -lm -lgcc -T ./common/test.ld"
dump_cmd="riscv64-unknown-linux-gnu-objdump --disassemble-all --disassemble-
zeroes --section=.text --section=.text.startup --section=.text.init --
section=.data test.riscv > test.dump"
exec_cmd="${compile_cmd} && ${dump_cmd}"

basedir=$(dirname "$0")
tmpdir=/tmp/compile
mount_flag="--mount type=bind,src=${basedir},dst=${tmpdir}"

#podman run ${mount_flag} -it --rm docker.io/jxy324/riscv-toolchain:v0 bash -c
"export PATH=\${PATH}:${extra_path}; printenv && echo \"${exec_cmd}\" > aaaa &&
bash"
podman run ${mount_flag} -it --rm docker.io/jxy324/riscv-toolchain:v0 bash -c
"export PATH=\${PATH}:${extra_path}; cd ${tmpdir} && ${exec_cmd}"
```

上面的脚本会使用 riscv-toolchain 中的工具链编译 test.c 文件，输出可执行文件 test.riscv，以及它的反汇编文件 test.dump。你可以修改 test.c 文件来测试自己希望 boom 执行的代码

在 boom_simulator_env 目录中运行如下命令，它将会在编译好的 boom 上执行 test.riscv 程序。预期看到类似下面的输出

```
$ ./build/simulator-MediumBoomConfig riscv_test/test.riscv
This emulator compiled with JTAG Remote Bitbang client. To enable, use
+jtag_rbb_enable=1.
Listening on port 43765
[UART] UART0 is here (stdin/stdout).
```

打印调试信息

在 `boom_simulator_env/src/main/scala/util` 中新建 `logging.scala` 文件，内容如下

```
//------------------------------------------------------------------------------
//------------------------------------------------------------------------------
// Logger for Debugging
//------------------------------------------------------------------------------
//------------------------------------------------------------------------------

package boom.util

import chisel3._
import freechips.rocketchip.util.WideCounter
import scala.reflect.runtime.universe._

package object logging {
  /**
   * Object for generating debug logs.
```

```
   *
   * Suggested naming conventions:
   *
   * 1. Instructions: `"inst" -> inst.toHex`.
   *
   * 2. Program counter: `"pc" -> pc.toHex`.
   */
  object dbg  // scalastyle:ignore
  {
    def apply(dps: LoggablePair*): Unit = {
      // TODO: `WideCounter` mey initialize additional registers
      val modName = Module.currentModule.map(_.name).getOrElse("<global>")
      val lastTrace = new Exception().getStackTrace()(1)
      val sourcePosition =
s"${lastTrace.getFileName}:${lastTrace.getLineNumber}"
      val items = LoggableTuple("module", modName) +:
                  LoggableTuple("source", sourcePosition) +:
                  LoggableTuple("cycle", WideCounter(32).value) +:
                  dps
      printf(PString("{") +
        Printables(items.flatMap(i => Seq(i.toKeyValue,
PString(","))).dropRight(1)) +
        PString("}\n"))
    }
  }

  /**
   * Formats for logging `chisel3.Bits`.
   */
  object BitsFormats extends Enumeration
  {
    type BitsFormat = Value

    val Dec, Hex, Bin = Value
  }

  /**
   * Represents objects that can be casted between `BitFormats`.
   */
  sealed trait HasBitsFormats
  {
    def toDec: HasBitsFormats
    def toHex: HasBitsFormats
    def toBin: HasBitsFormats
  }

  /**
   * Represents a loggable object.
   */
  sealed trait Loggable
  {
    def toPrintable: Printable
  }

  /**
```

```scala
   * Represents a loggable key-value pairs.
   */
  sealed trait LoggablePair extends Loggable
  {
    def key: Printable
    def value: Loggable

    override def toPrintable: Printable = value.toPrintable

    def toKeyValue: Printable = key + PString(":") + value.toPrintable
  }

  /**
   * Wrapper for tuples.
   */
  case class LoggableTuple(tuple: (String, Loggable)) extends LoggablePair {
    override def key: Printable = PString(escape(tuple._1))
    override def value: Loggable = tuple._2
  }

  /**
   * Wrapper for `chisel3.Bits`.
   */
  case class LoggableBits(bits: Bits, fmt: BitsFormats.BitsFormat)
    extends LoggablePair with HasBitsFormats
  {
    override def key: Printable = PString("\"") + Name(bits) + PString("\"")
    override def value: Loggable = this

    override def toPrintable: Printable = fmt match {
      case BitsFormats.Dec => Decimal(bits)
      case BitsFormats.Hex => PString("\"0x") + Hexadecimal(bits) +
PString("\"")
      case BitsFormats.Bin => PString("\"0b") + Binary(bits) + PString("\"")
    }

    override def toDec: LoggableBits = LoggableBits(bits, BitsFormats.Dec)
    override def toHex: LoggableBits = LoggableBits(bits, BitsFormats.Hex)
    override def toBin: LoggableBits = LoggableBits(bits, BitsFormats.Bin)
  }

  /**
   * Wrapper for strings.
   */
  case class LoggableString(str: String) extends Loggable
  {
    override def toPrintable: Printable = PString(escape(str))
  }

  /**
   * Wrapper for sequences.
   */
  case class LoggableSeq[T <: Bits](seq: Seq[T], fmt: BitsFormats.BitsFormat)
    extends Loggable with HasBitsFormats
  {
```

```scala
    override def toPrintable: Printable = seqToPrintable(seq, fmt)

    override def toDec: LoggableSeq[T] = LoggableSeq(seq, BitsFormats.Dec)
    override def toHex: LoggableSeq[T] = LoggableSeq(seq, BitsFormats.Hex)
    override def toBin: LoggableSeq[T] = LoggableSeq(seq, BitsFormats.Bin)
  }

  /**
   * Wrapper for `chisel3.Vec`.
   */
  case class LoggableVec[T <: Bits](vec: Vec[T], fmt: BitsFormats.BitsFormat)
    extends LoggablePair with HasBitsFormats
  {
    override def key: Printable = PString("\"") + Name(vec) + PString("\"")
    override def value: Loggable = this

    override def toPrintable: Printable = seqToPrintable(vec, fmt)

    override def toDec: LoggableVec[T] = LoggableVec(vec, BitsFormats.Dec)
    override def toHex: LoggableVec[T] = LoggableVec(vec, BitsFormats.Hex)
    override def toBin: LoggableVec[T] = LoggableVec(vec, BitsFormats.Bin)
  }

  /**
   * Implicit conversions for `Debuggable`s.
   */
  import scala.language.implicitConversions
  implicit def bitsToLoggable(b: Bits): LoggableBits = LoggableBits(b,
BitsFormats.Dec)
  implicit def tupleToLoggable(t: (String, Loggable)): LoggableTuple =
LoggableTuple(t)
  implicit def stringToLoggable(s: String): LoggableString = LoggableString(s)
  implicit def seqToLoggable[T <: Bits](s: Seq[T]): LoggableSeq[T] =
LoggableSeq(s, BitsFormats.Dec)
  implicit def vecToLoggable[T <: Bits](v: Vec[T]): LoggableVec[T] =
LoggableVec(v, BitsFormats.Dec)
  implicit def sbToLoggable(t: (String, Bits)): LoggableTuple =
LoggableTuple((t._1, t._2))
  implicit def siToLoggable(t: (String, Int)): LoggableTuple =
LoggableTuple((t._1, t._2.U))
  implicit def sStrToLoggable(t: (String, String)): LoggableTuple =
LoggableTuple((t._1, t._2))
  implicit def sSeqToLoggable[T <: Bits](t: (String, Seq[T])): LoggableTuple =
LoggableTuple((t._1, t._2))
  implicit def sVecToLoggable[T <: Bits](t: (String, Vec[T])): LoggableTuple =
LoggableTuple((t._1, t._2))

  /**
   * Escapes a specific string.
   * @param str input raw string
   * @return escaped string
   */
  private def escape(str: String): String = Literal(Constant(str)).toString

  /**
```

```
   * Converts a sequence of chisel3 bits to `Printable`.
   * @param seq the bits sequence
   * @param fmt print format of bits
   * @tparam T type of bits
   * @return a printable object
   */
  private def seqToPrintable[T <: Bits](seq: Seq[T], fmt:
BitsFormats.BitsFormat): Printable =
    if (seq.isEmpty) PString("[]")
    else PString("[") +
      Printables(seq
        .flatMap(i => Seq(LoggableBits(i, fmt).toPrintable, PString(",")))
        .dropRight(1)) +
      PString("]")
}
```

我们在调试时主要使用上面代码中定义的 `dbg` 对象。它是 chisel 中 `printf` 的封装，接收任意多个二元组参数。例如下面的例子，我们在 `src/main/scala/exu/core.scala` 中添加如下代码。它将在 `dis_fire` 信号不为 0 时输出当前 `dis_fire` 的值。

```
import boom.util.logging._

when(dis_fire.asUInt =/= 0.U){
  dbg("dis_fire" -> dis_fire)
  // 可以传入多个二元组，例如 dbg("dis_fire" -> dis_fire, "dec_fire" -> dec_fire)
}
```

重新编译 boom。运行时加上 `--verbose` 参数，并将输出重定向到 log.txt 文件中

```
$./build/simulator-MediumBoomConfig --verbose riscv_test/test.riscv > log.txt
2>&1
```

运行完成后，打开 log.txt 文件，能看到类似下面的输出

```
using random seed 1708240833
This emulator compiled with JTAG Remote Bitbang client. To enable, use
+jtag_rbb_enable=1.
Listening on port 36765
{"module":"BoomCore","source":"core.scala:1082","cycle":        20,"dis_fire":
[1,1]}
{"module":"BoomCore","source":"core.scala:1082","cycle":        34,"dis_fire":
[1,0]}
{"module":"BoomCore","source":"core.scala:1082","cycle":        46,"dis_fire":
[1,0]}
{"module":"BoomCore","source":"core.scala:1082","cycle":        58,"dis_fire":
[1,1]}
{"module":"BoomCore","source":"core.scala:1082","cycle":        59,"dis_fire":
[1,0]}
{"module":"BoomCore","source":"core.scala:1082","cycle":        68,"dis_fire":
[0,1]}
{"module":"BoomCore","source":"core.scala:1082","cycle":        80,"dis_fire":
[1,0]}
```

```
{"module":"BoomCore","source":"core.scala:1082","cycle":        88,"dis_fire":
[0,1]}
{"module":"BoomCore","source":"core.scala:1082","cycle":       100,"dis_fire":
[1,0]}
{"module":"BoomCore","source":"core.scala:1082","cycle":       112,"dis_fire":
[1,0]}
{"module":"BoomCore","source":"core.scala:1082","cycle":    148232,"dis_fire":
[1,0]}
{"module":"BoomCore","source":"core.scala:1082","cycle":    148254,"dis_fire":
[1,0]}
{"module":"BoomCore","source":"core.scala:1082","cycle":    148262,"dis_fire":
[0,1]}
{"module":"BoomCore","source":"core.scala:1082","cycle":    148274,"dis_fire":
[1,0]}
{"module":"BoomCore","source":"core.scala:1082","cycle":    148277,"dis_fire":
[0,1]}
{"module":"BoomCore","source":"core.scala:1082","cycle":    148278,"dis_fire":
[1,1]}
{"module":"BoomCore","source":"core.scala:1082","cycle":    148279,"dis_fire":
[1,1]}
{"module":"BoomCore","source":"core.scala:1082","cycle":    148280,"dis_fire":
[1,0]}
{"module":"BoomCore","source":"core.scala:1082","cycle":    148282,"dis_fire":
[0,1]}
{"module":"BoomCore","source":"core.scala:1082","cycle":    148283,"dis_fire":
[1,1]}
{"module":"BoomCore","source":"core.scala:1082","cycle":    148284,"dis_fire":
[1,0]}
{"module":"BoomCore","source":"core.scala:1082","cycle":    148286,"dis_fire":
[0,1]}
{"module":"BoomCore","source":"core.scala:1082","cycle":    148287,"dis_fire":
[1,1]}
{"module":"BoomCore","source":"core.scala:1082","cycle":    148288,"dis_fire":
[1,0]}
{"module":"BoomCore","source":"core.scala:1082","cycle":    148290,"dis_fire":
[0,1]}
{"module":"BoomCore","source":"core.scala:1082","cycle":    148291,"dis_fire":
[1,1]}
{"module":"BoomCore","source":"core.scala:1082","cycle":    148292,"dis_fire":
[1,0]}
```

调用 `dbg` 对象得到的输出格式如上，首先会输出调用位置，在哪个 module 中，源码的哪个位置。然后是输出当前的 cycle 数，最后是输出调用 `dbg` 时传入的参数，通常传入若干个二元组，二元组的第一个参数通常是一个描述性的字符串，第二个参数是需要输出的信号