USTB

# LoongArch32R 单周期处理器设计指导书

2024, Spring


**作者**

Shuozhe LAI

Yuchen WANG

Ziqing ZHANG



USTB, 2024 年 5 月 16 日

# 目录

# 0 | 写在前面

## 关于 RTFM, STFW, RTFSC

在学习的过程中，你会使用到我们提供的处理器开发环境，其中包括了对基本的处理器的仿真验证的功能。在这一过程中，你可能会遇到一些错误，这些错误，有些可能是由于你对实验内容的不够了解，有些可能是因为你对实验环境提供的工具甚至是实验环境本身的不了解，这些问题不仅是在实验当中，也可能存在于大家未来的学习和工作当中，基于此，我们在这里向大家介绍计算机专业前辈们总结出对付错误的经验。

RTFM(Read The Friendly Manual)，阅读手册。在本实验中提供的手册有很多，比如实验指导书，比如说关于指令集架构的文档。如果你遇到错误，请首先先翻一翻手册，看看是否是因为手册中某个没有注意到细节而导致了错误。手册会对实验环境和实验内容进行简要的介绍，它无法覆盖实验中的方方面面，但是一定会将实验中的重点难点展现给大家，因此查询手册往往能够帮助我们快速解决问题。我们会在后面的资源使用路线中简单介绍一下本课程中可能会用到的各手册的内容。

STFW(Search The Friendly Web)，上网搜索。手册会尽量将实验中的细节描述清楚，帮助大家完成实验。然而遗憾的是，手册没有办法覆盖到实验中的每一个细节，每一种可能的错误，（比如大魔王 Vivado 的各种报错）

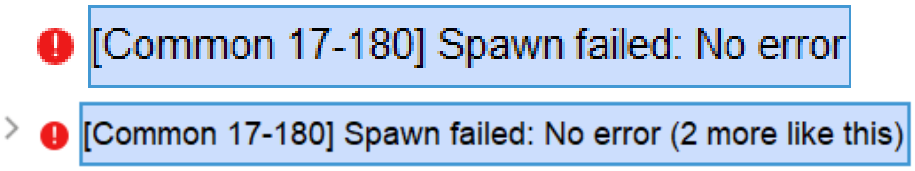


图 0.1: Vivado 的报错

所以这种时候可以考虑上网搜索。从效率的角度考虑，我们希望大家尽量使用 Google/Bing 国际版进行搜索，多在国外的一些优秀论坛上寻找解决方案。

RTFSC(Read The Friendly Source Code)，阅读源代码。实验环境中有大量代码，他们是整个实验环境的支撑。想要完成好实验，一定需要充分的了解实验环境，手册中已经给出了一部分关于实验环境的描述，但是并不足以支持你理解实验环境的全貌，所以适当的阅读实验环境中的源代码，诸如测试用例的源代码，仿真环境的源代码，通过阅读这些代码，相信你不仅能解决问题，也能加深自己对于整个实验环境的理解。

当自己已经充分尝试过以上方法时，你可以向助教，老师，或者是其他同学求助，但是希望你在提问的过程中，能够尽量清晰的描述你的问题，帮助你提问的人能够更快理解你当前所面对的情况，更好的解决问题。

关于如何提问，你可以阅读《提问的智慧》或《别像弱智一样提问》，上面的三板斧和正确提问的方法是计算机世界的答案之书，在本门课程的理论内容的学习之外，我更希望大家能够学会正确的提问，学会正确而高效的寻找答案的方法。

# 1 | 实验目的

掌握 LoongArch32R 单周期处理器设计的基本原理，可以在 Vivado 中基于 Verilog 设计支持 23+3 条指令的哈弗架构单周期处理器。

# 2 | 背景知识

单周期处理器指所有指令均可以在一个周期内完成的处理器。优点是实现较为简单，不存在的数据冲突、控制冲突等问题，适合初学者进行学习、设计与实现。

但也正是由于处理器必须在一个周期内执行完一条指令，即所有指令执行时间等长。如乘法、除法此类开销较大，硬件实现复杂、执行时间长的算术运算指令将会成为单周期处理器的瓶颈。并且所有部件在一个周期内无法复用，因此需要复用的部件均需设计多个拷贝，这也会令单周期处理器的资源消耗有所提升。

考虑设计的复杂性，本章的单周期处理器将会采用哈弗架构，即指令存储器与数据存储器分离的设计。这样可以避免访存指令在一周期内需要访问两次相同存储器的问题，简化设计。同样由于复杂度原因，本章将会采用分布式存储器（Distributed RAM），而非块存储器（Block RAM）。此类存储器是异步的，时序简单，在给出地址的当周期即可输出数据，可以简化单周期处理器的设计。对于单周期处理器而言，测试程序简单，没有大量的数据存储，同样满足了分布式 RAM 不宜过大的条件，因此选择分布式存储器作为数据存储器。对于指令存储器而言，假设永远不会发生指令自修改，可以考虑使用只读存储器（ROM）进行实现。

本实验将会基于 LoongArch32R 指令集构建单周期处理器，考虑到从零开始实现一个单周期处理器的难度有点较大，本次实验将会给出示例代码通过补全和修正代码的形式完成一个单周期处理的设计。

在这里，我们给出龙芯架构 32 位的单周期 CPU 的设计，其结构如下图所示：

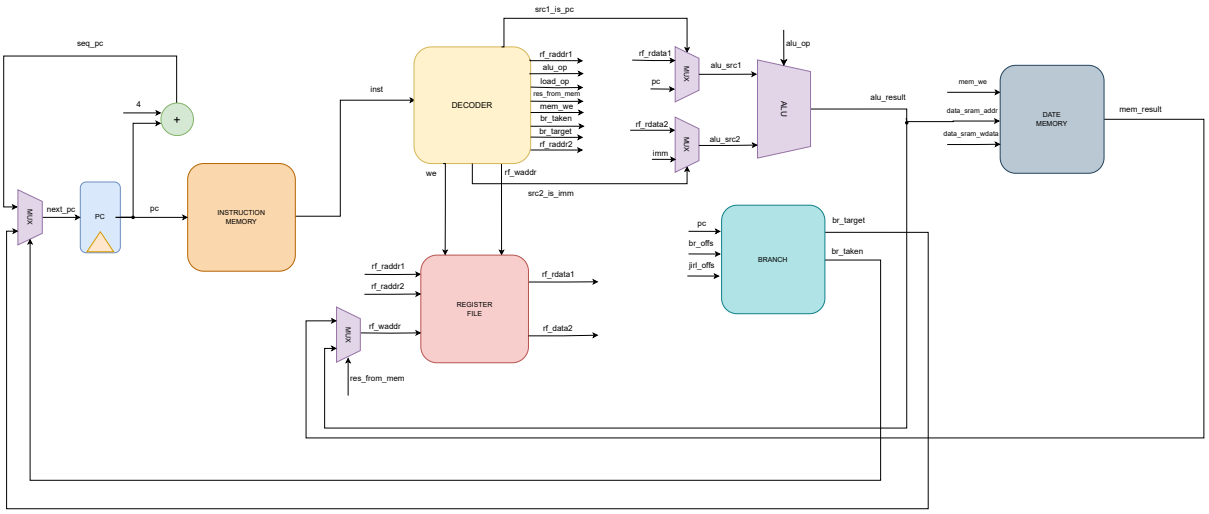


图 2.1: CPU 顶层设计

# 3 | 龙芯架构指令集介绍

## 3.1 | 指令编码格式

我们实验当中选择的指令集系统为 LoongArch32R 指令集，这是由 LoongArch32 指令集系统精简而来的一套指令集，龙芯架构 32 位精简版中的所有指令均采用 32 位固定长度，且指令的地址都要求 4 字节边界对齐。当指令地址不对齐时将触发地址错例外。

其中，包含 9 种典型的指令编码格式，即 3 种不含立即数的编码格式 2R、3R、4R，以及 6 种含立即数的编码格式 2RI8、2RI12、2RI14、2RI16、1RI21、I26。

本次实验中实现的 20 条指令均在 9 种经典的指令编码格式之内。

表 3.1: 指令格式

| | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 09 | 08 | 07 | 06 | 05 | 04 | 03 | 02 | 01 | 00 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2R-type | opcode | | | | | | | | | | | | | | | | | | | | | | rj | | | | | rd | | | | |
| 3R-type | opcode | | | | | | | | | | | | | | | | rk | | | | | | rj | | | | | rd | | | | |
| 4R-type | opcode | | | | | | | | | | | | ra | | | | rk | | | | | | rj | | | | | rd | | | | |
| 2RI8-type | opcode | | | | | | | | | | | | I8 | | | | | | | | | | rj | | | | | rd | | | | |
| 2RI12-type | opcode | | | | | | | | | | I12 | | | | | | | | | | | | rj | | | | | rd | | | | |
| 2RI14-type | opcode | | | | | | | | I14 | | | | | | | | | | | | | | rj | | | | | rd | | | | |
| 2RI16-type | opcode | | | | | | | | | | | | I16 | | | | | | | | | | rj | | | | | rd | | | | |
| 1RI21-type | opcode | | | | | | | | | | I21[15:0] | | | | | | | | | | | | rj | | | | | I21[20:16] | | | | |
| I26-type | opcode | | | | | | | | | | I26[15:0] | | | | | | | | | | | | I26[25:16] | | | | | | | | | |

从上表我们可以看到，指令当中包含了这几部分信息：

**操作码** LoongArch32R 的操作码不是定长的，高位就是操作码

**立即数** 包含一个立即数

rk **地址** 寄存器操作数地址

rj **地址** 寄存器操作数地址

rd **地址** 寄存器写入地址

## 3.2 | 指令介绍示例

想要设计一款使用 LoongArch32R 指令集系统的处理器，显然需要了解 LoongArch32R 指令集系统，而了解它的最好办法就是翻阅手册

接下来，我们将通过对 ld.w , st.w , addi.w , beq 这四条指令进行简单的介绍。

通过对指令集手册的查询，信息获取，你可以很容易的获得剩下指令的具体行为描述，这对于完成实验的代码补全和修正具有着极大的帮助。

**注：以下信息均可以通过翻阅手册获得。**

### 3.2.1 | ld.w

**LD.W:**

```
vaddr = GR[rj] + SignExtend(si12, 32)
AddressComplianceCheck(vaddr)
```

```
paddr = AddressTranslation(vaddr)
word = MemoryLoad(paddr, WORD)
GR[rd] = word
```

不难理解，ld.w 首先将 rj 寄存器的内容与一个 12 位立即数符号扩展到 32 位后相加，将该结果作为访存地址送至数据存储器中，并将从数据存储器读出的数据写入 rd 寄存器中。

### 3.2.2 | st.w

**ST.W:**

```
vaddr = GR[rj] + SignExtend(si12, 32)
AddressComplianceCheck(vaddr)
paddr = AddressTranslation(vaddr)
MemoryStore(GR[rd][31:0], paddr, WORD)
```

容易看出，st.w 访问地址的计算与 ld.w 相同，得到访存地址后将地址送至数据存储器中的同时将 rd 寄存器的数据送入数据存储器，并拉高数据存储器的写使能，从而将 rd 中的数据写入到内存（在此即数据存储器）中。

### 3.2.3 | addi.w

**ADDI.W:**

```
tmp = GR[rj] + SignExtend(si12, 32)
GR[rd] = tmp[31:0]
```

依题意得，addi.w 将 rj 寄存器的内容与一个 12 为立即数符号扩展到 32 位后相加，将该结果写入至 rd 寄存器中。

### 3.2.4 | beq

**BEQ:**

```
if GR[rj]==GR[rd] :
    PC = PC + SignExtend({offs16, 2'b0}, 32)
```

beq 将读出 rj 与 rd 寄存器中的内容并比较。当 rj 寄存器中内容与 rd 寄存器中内容相等时，会将 PC 与一个 16 位立即数符号扩展到 32 位后相加，并存入 PC 中。

## 3.3 | 指令码查询

指令集手册的附录 B “**指令码一览**” 中可以找到所需要的指令的指令码，此处只展示部分。

表 3.2: 指令码

| 指令 | | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| LD.W | rd, rj, si12 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | si12 | | | | | | | | | | | | | | rj | | | | | | rd | |
| ST.W | rd, rj, si12 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | si12 | | | | | | | | | | | | | | rj | | | | | | rd | |
| ADDI.W | rd, rj, si12 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | si12 | | | | | | | | | | | | | | rj | | | | | | rd | |
| BEQ | rj, rd, offs | 0 | 1 | 0 | 1 | 1 | 0 | offs[15:0] | | | | | | | | | | | | | | | | | | rj | | | | | | rd | |

# 4 | 代码框架

从实验包当中，拿到的代码框架如下：

```
.
├── mycpu_env
│   ├── func
│   ├── gettrace
│   ├── myCPU
│   └── soc_verify
```

func　myCPU 所需要运行的程序

gettrace　用于 difftest 的正确设计 CPU

myCPU　你所要完成补全和修正的代码目录

soc_verify　用于创建 Vivado 工程，仿真测试和上板测试

# 5 | 模块介绍

**注：配合 CPU 设计结构图观看效果更佳**

## 5.1 | PC（程序计数器）

是一个特殊寄存器，该寄存器无法被指令直接修改，而只能被转移指令、例外陷入和例外返回指令间接修改，对于部分指令，PC 也可以成为一个隐含的源操作数寄存器。

PC 用于存储当前指令的地址，通过多路选择器进行跳转判断，如果要进行跳转，则选择跳转地址（br_target），不进行跳转就是顺序地址（seq_pc）。

## 5.2 | INSTRUCTION MEMORY（指令寄存器）

存储指令的寄存器，根据从 PC 中取得的地址，从指令寄存器中取出对应的指令（inst）。这个存储器使用 DRAM 实现。

## 5.3 | DECODER（指令译码器）

译码模块是 CPU 中最重要的一个模块。在该模块中，将会判断指令的类型、源操作数、目的寄存器、运算类型等信息，同时，条件跳转语句也可以在本模块中进行处理。

通过传入的 Loongarch32R 指令，生成出如下的控制信号（只举例部分）：

rf_raddr1　寄存器堆（rf）的读端口 1 的地址，一般由指令的 rj 部分给出；

rf_raddr2　寄存器堆（rf）的读端口 2 的地址，一般由指令的 rk 或者 rd 部分给出；

rf_waddr 寄存器堆（rf）的写端口地址,，可能由 rd 部分给出，具体情况根据实现指令而定（bl 指令中，此地址为 1）

we 寄存器堆（rf）写使能信号；

alu_op ALU 的操作类型，译码器根据对指令的译码，给出相应的类型；

imm 立即数，根据指令的不同，立即数的类型也不相同，具体指令具体分析，存在于指令中。注意分清立即数的长度以及类型；

alu_src1 根据多路选择器进行判断，选择出的 ALU 的第一个操作数来源；

mem_we 数据存储器写使能信号；

br_taken 是否发生跳转信号。

根据上面举例出的信号，我们可以认识到在本模块，会实现指令的译码，寄存器原操作数，目标地址的准备，ALU 操作数和内存操作数的准备，完成跳转指令更新 PC 通路。

## 5.4 | REGISTER FILE（寄存器堆）

LoongArch32R 指令集中共定义了 32 个 32 位通用寄存器（General-purpose Register，简称 GR），记为 r0～r31，其中第 0 号寄存器 r0 的值恒为 0。

本实验定义一个寄存器堆（RegFile）来实现这 32 个通用寄存器，并且在其中保证了 r0 读出数据恒为 0。通过分析 LA32R 的指令格式可以发现：一条指令最多需要读两个通用寄存器（第一个为 rj，第二个为 rk 或 rd），写一个通用寄存器（rd）。因此我们设计的寄存器堆需要两个独立的读端口和一个写端口，并且读写可同时进行。

由于只涉及到单周期 CPU，该寄存器堆的读也必须是异步的，即输入地址后可以在同周期输出读数据。因寄存器堆需要“寄存”数据，因此必须使用触发器实现，写入由此也只能是同步写入。写入的数据通过多路选择器根据不同的指令进行判断取舍。

## 5.5 | ALU（算术逻辑单元）

CPU 的绝大部分运算任务都会在 ALU 中完成。ALU 将会根据译码模块提供给 ALU 的信息：源操作数 1、源操作数 2 和操作类型进行运算，并将运算结果送至下一个模块进行处理。

**注：在 ALU 单元中，要注意区分运算类型，同时保证操作数完成正确的算术逻辑运算**

## 5.6 | BRANCH（分支跳转）

分支跳转单元，用于判断是否跳转。分支跳转单元的分支类型由译码器进行判断给出，既能够计算出是否需要跳转，还可以计算出跳转的地址。

**注：不同跳转指令的跳转条件不同，跳转地址的计算方式也不一定相同，需要根据具体指令进行具体判断。**

### 5.7 | DATA MEMORY（数据存储器）

数据存储器，用于存储数据。数据存储器的读写使能信号由译码器给出，读写地址由 ALU 给出，写数据由 rf_rdata2 给出。和指令存储器一样，我们同样使用 DRAM 来实现。

### 5.8 | WBMUX（写回选择器）

即图中 rf_waddr 前的多路选择器，用于选择写回的数据，写回的数据可能来自 ALU，也可能来自 DATA MEMORY。选择信号由译码器给出。

## 6 | 项目工程创建

本次实验通过 tcl 命令创建项目工程，在 vivado 的初始页面打开 Tcl Console

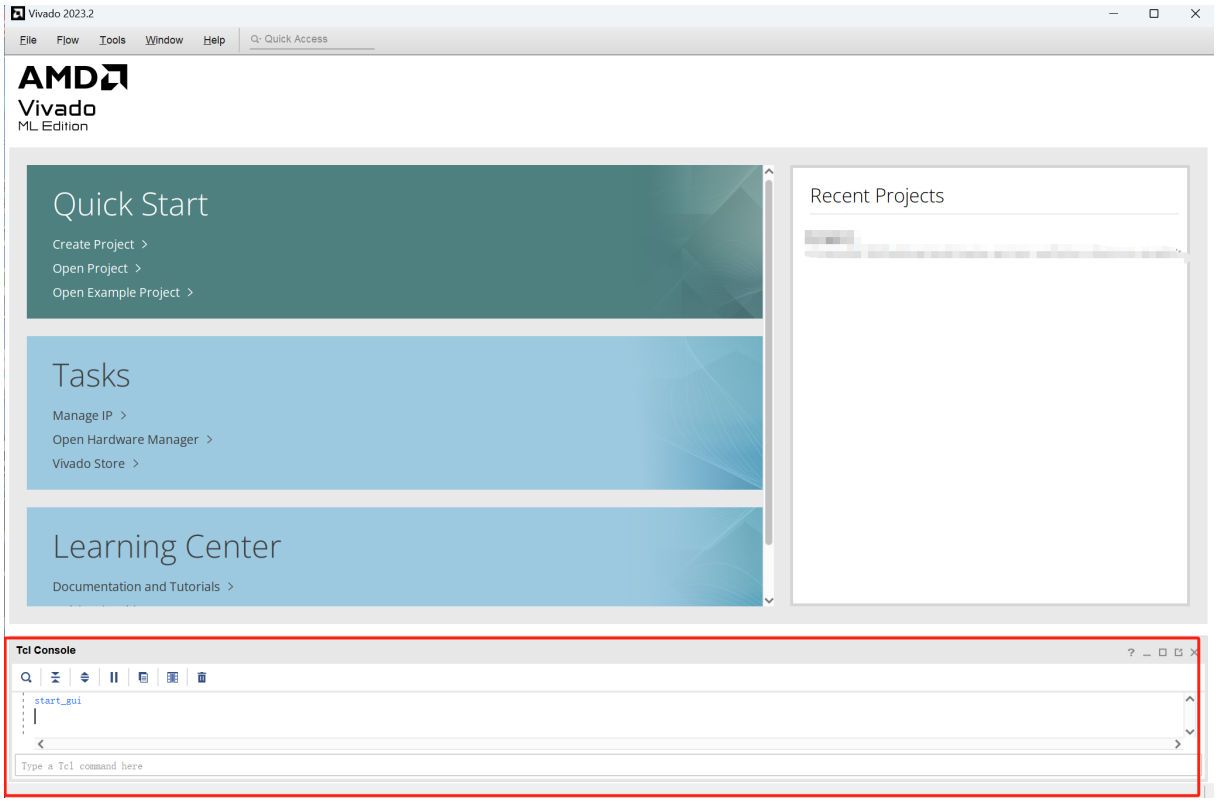


图 6.1: vivado 初始页面

在打开的 Tcl Console 中输入命令，`cd` 到待使用 create_project.tcl 文件所在目录，进入 `create_project.tcl` 目录中

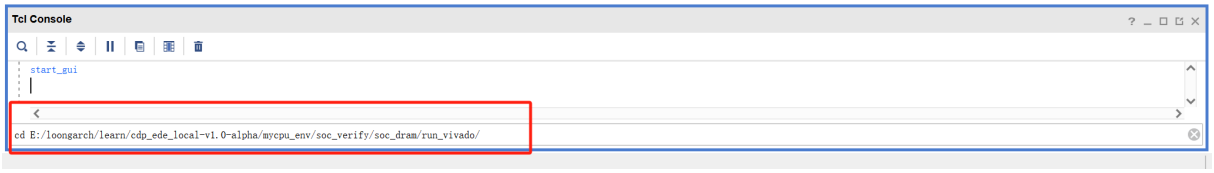


图 6.2: Tcl Console

继续在 Tcl Console 中输入命令 `source ./create_project.tcl`。接下来 Vivado 将根据 `create_project.tcl` 的内容创建工程。

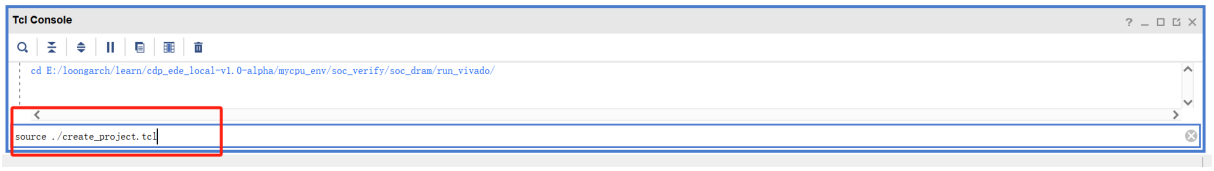


图 6.3: 创建工程

完成之后，vivado 会自动创建工程。

# 7 | 环境配置以及仿真验证

## 7.1 | 环境配置

本实验存在 7 个指令测试点，存在于 `test` 文件夹中，其中的文件是由我们已经完成测试点程序编译生成的 obj 文件，为了区分我们将其命名为 `obj1,2,3...` **注意：请勿修改 `obj` 中的任何文件**，`readme.txt` 文件中已经列出了每个 `obj` 测试的指令序列.

下面介绍如何利用测试点对自己的 CPU 设计正确性进行仿真验证：

## 7.2 | 仿真验证

当你完成 CPU 的设计，需要进行仿真验证，本实验通过龙芯提供的 trace 比对调试框架实现对 CPU 设计正确性的验证。

假设你要测试测试点 1（`obj1`），复制 `obj1` 文件到 `func` 文件夹目录下，**重命名为 `obj`，重命名为 `obj`，重命名为 `obj`!!!**

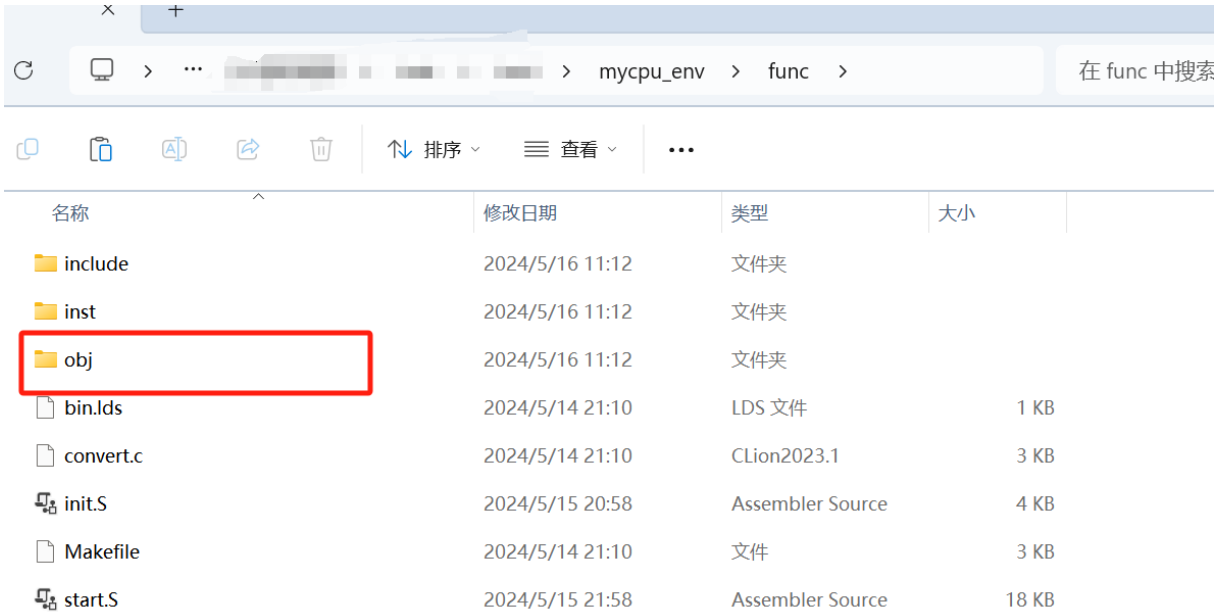


图 7.1: 复制 `obj` 文件

然后进入 `gettrace` 文件夹，打开 gettrace 项目

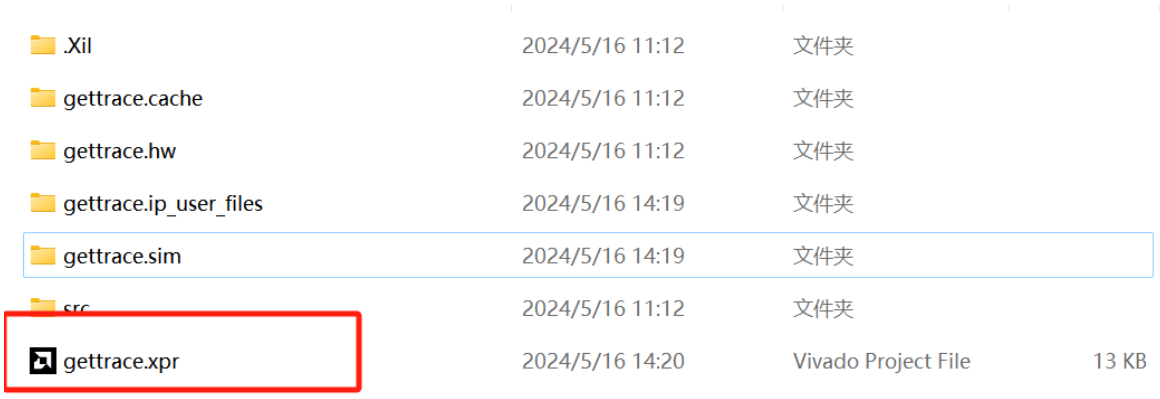


图 7.2: 打开 gettrace 项目

进入项目后点击 Run Simulation 进入仿真页面，点击 run all 进行仿真

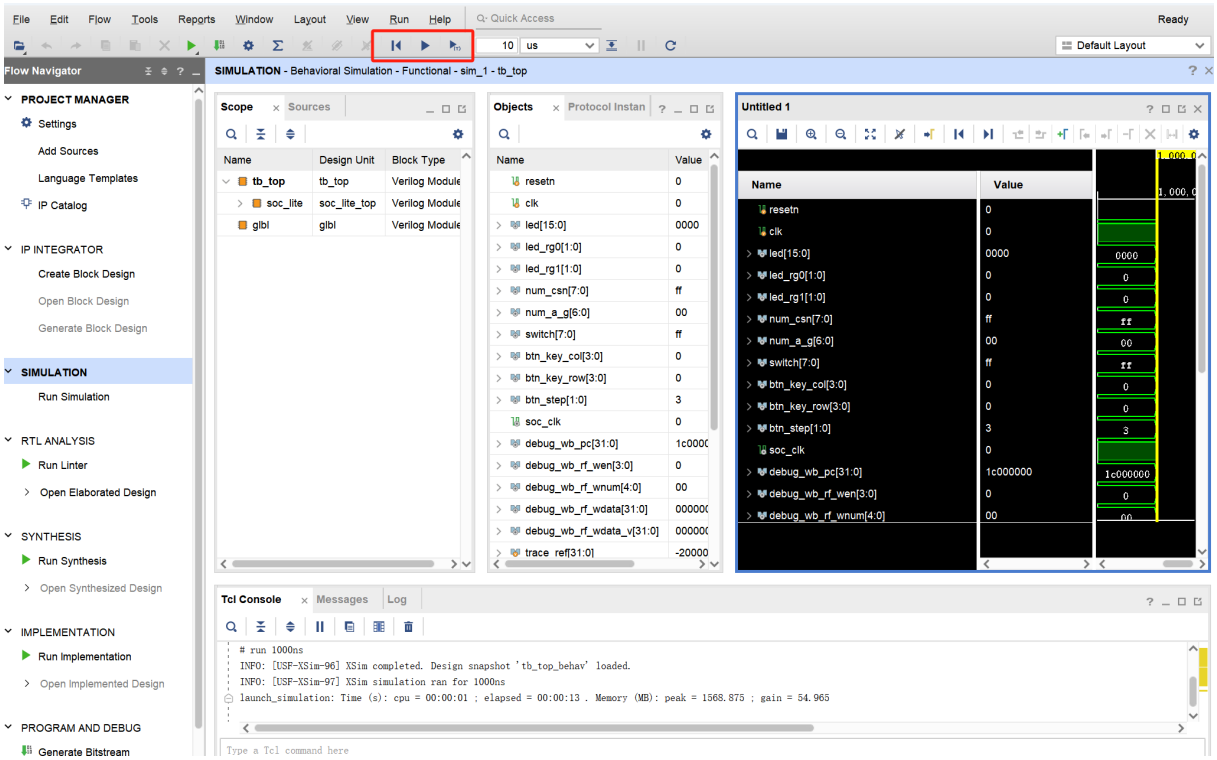


图 7.3: 启动仿真，生成 golden_trace

等待仿真结束之后，通过控制台可以查看 `gettrace.txt` 文件是否生成成功，如下图所示。

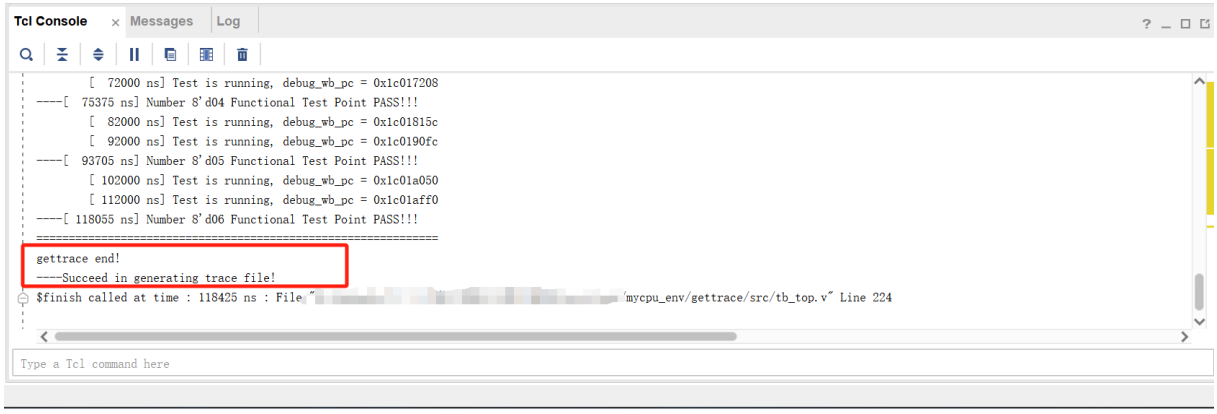


图 7.4: golden_trace 生成完毕

查看 gettrace 文件夹可以发现，成功生成 `gettrace.txt` 文件。

**至此，准备工作均已完成，现在可以回到项目工程进行 CPU 设计仿真验证。**

我们已经给出了 testbench，只需进入工程的仿真页面，点击 run all 进行功能验证与调试，直至仿真测试通过。

以下为仿真测试通过示例：


图 7.5: 仿真测试通过

若仿真不通过，则说明 CPU 结构仍存在错误，可以通过控制台返回的信息找到错误存在的地址，如下：

```
------------------------------------------------------------
[ 604377 ns] Error!!!
    reference: PC = 0x1c0555f4, wb_rf_wnum = 0x0d, wb_rf_wdata = 0xe0c335f4
    mycpu    : PC = 0x1c0555f4, wb_rf_wnum = 0x0d, wb_rf_wdata = 0x380656d8
------------------------------------------------------------
```

图 7.6: 仿真测试失败

在 `obj` 文件夹中存在着由反汇编生成的 `test.S` 文件，可以根据指令地址寻找发生错误的指令，再根据仿真产生的波形图追根溯源，找到代码之中产生错误的原因。这是你们寻找代码错误快速且唯一的方法。

```
1735    1c0555f4:   1d897bcd    pcaddu12i   $r13, -242722(0xc4bde)
1736    1c0555f8:   50000c00    b   12(0xc) # 1c055604 <n21_pcaddu12i_test+0x24>
1737    1c0555fc:   0010302c    add.w   $r12, $r1, $r12
1738    1c055600:   4c000020    jirl    $r0, $r1, 0
1739    1c055604:   00103c01    add.w   $r1, $r0, $r15
1740    1c055608:   5c2a3dac    bne $r13, $r12, 10812(0x2a3c) # 1c058044 <inst_error>
```

图 7.7: 寻找错误指令

（上图仅为举例使用，并无任何参考意义。）

## 8 | 注意事项

相信根据上述介绍，你已经能很容易的完成单周期处理器的补全和修正，以下还有些许注意事项：

1. 你所需要修改和完成的代码全部放在 myCPU 目录之下的代码，除少量配置项（**这并非必要的**）外，你不需要修改其他目录当中的代码。
2. 如果你想详细了解 **基于** gettrace **的调试辅助手段** 以及 func **功能测试程序**，可以参考CPU **设计实战：**LoongArch **版**，其中有详细介绍。
3. 在给出的工程文件中，模块介绍部分中的 ALU，RIGISTER FILE 为单独实现的模块，其他模块均聚合在 mycpu_top 中。
4. 存在错误的模块只有 ALU 以及 mycpu_top，其他地方代码均正确无误，不需要进行修改但 $\neq$ 不需要阅读。
5. mycpu_top 中有一个改错，剩下均为代码补全，用`/* todo */`标出，alu 中均为代码改错。
6. CPU **设计实战：**LoongArch **版** 讲义中实验要求实现指令与本实验并不相同，官方提供的测试点也与本实验存在差异（本实验测试点文件均为我们配置编译），一切配置项文件以本实验提供的文件为主，不建议对任何配置项进行修改，可能会导致仿真出现错误。
7. 有兴趣实现流水级以及阻塞,前递等 CPU 结构设计的同学可以参考CPU **设计实战：**LoongArch **版** 中的实践训练。
8. 以上内容仅为一些浅薄见解，如有不妥或错误之处，希望大家批评指正。