

第九届

全国大学生集成电路创新创业大赛

报告类型： RISC-V 架构 CPU 设计报告

参赛杯赛： 竞业达杯

作品名称： 睿思科 RISC-V 处理器

队伍编号： CICC0902477

团队名称： 流水线搬砖队

基于 RISC-V 指令集的 CPU 设计

流水线搬砖队

2025 年 7 月

目录

1	项目概述	4
1.1	项目背景	4
1.2	项目特色	4
1.3	设计平台	4
2	设计说明	4
2.1	CPU 基本结构	4
2.2	CPU 架构图	4
2.3	基于握手的流水线时序控制	5
2.3.1	valid 信号的产生	5
2.3.2	ready 信号的产生	6
2.4	基于 BTB 的分支预测器	7
2.4.1	构建 BTB 模块	7
2.4.2	分支预测策略	8
2.5	两级传递的数据旁路	9
2.6	乘法指令的优势	10
2.7	AXI4-Lite 总线接口	10
2.7.1	AXI4-Lite 接口信号定义	10
3	CPU 仿真测试平台设计	12
3.1	自己搭建测试平台的优势	12
3.2	测试平台可供配置的功能	12
3.3	调试器支持的指令	12
3.3.1	c 指令（继续运行）	13
3.3.2	si 指令（步进执行）	13
3.3.3	info 指令（打印变量）	14
3.3.4	x 指令（内存查询）	15
3.3.5	p 指令（算式解析）	15
3.3.6	w/d 指令（添加/移除对变量的追踪）	16
3.4	性能评估功能	16
3.5	平台调试流程举例	17
3.5.1	选取想要测试运行的程序	17

3.5.2	交叉编译后得到需要装载进入参考 CPU 和 CPU 的二进制文件	17
3.5.3	进入目录之后运行 make 进行测试	18
3.6	CPU 仿真验证	18
3.6.1	自行准备的 C 语言测试集	18
3.6.2	coremark 测试集	19
3.6.3	dhrystone 测试集	20
3.6.4	南京大学 riscv-tests 测试集	20
4	验证报告	21
4.1	验证平台说明	21
4.1.1	FPGA 平台型号	21
4.1.2	下载/烧录工具版本	21
4.1.3	上电与串口交互测试说明（方式方法	22
4.2	测试用例说明	22
4.2.1	指令程序例程	22
4.2.2	上板运行流程说明	22
4.3	验证结果	22
4.3.1	结果总结	22

1 项目概述

1.1 项目背景

RISC-V 指令集由于其开源、开放的特点，近年来已成为产业界 CPU 研制和应用的新趋势。基于 RISC-V 指令集的 CPU 已广泛应用于各类工业应用，具有广阔的应用前景。

本项目以经典的五级流水线结构为基础，在此基础上添加了分支预测模块、数据旁路流水线等等优化设计，并且对于逻辑路径进行精心优化。构建了一款支持 RV32I、RV32M 指令集的 RISC-V 处理器。并且在项目中完成了从设计、仿真、FPGA 上验证的全流程。

1.2 项目特色

1. 使用了自主搭建的 CPU 仿真测试平台，功能相较于提供 Trace 更加完备。并制作了支持多种调试指令的调试器。
2. 使用了基于 BTB 的分支指令检测模块，优化了分支预测模块的时序问题。
3. 在构建数据旁路的时候添加了两级流水处理逻辑信号与前递数据，优化了大量组合逻辑带来的时序问题。
4. 支持 AXI4 总线协议与外设进行交换数据。
5. 在赛事要求的支持基础 RV32I 指令集的基础上扩展支持了 RV32M 指令集。
6. 在流水线间使用 valid 信号和 ready 信号进行数据的握手逻辑控制信号流动。
7. 使用 BRAM 同步存储器作为数据存储器，实现了更小的面积功耗与更高的速度。
8. 对逻辑路径进行精心优化，使得 CPU 在板上能够以最高 250Mhz 的情况下正确运行初赛所提供的程序。

1.3 设计平台

设计平台：Vivado Version 2023.1

设计语言：Verilog、SystemVerilog

2 设计说明

2.1 CPU 基本结构

本项目设计的 CPU 采用经典的五级流水线结构进行了改进，但是总体上还是可以划分为包含取指 (IF)、译码 (ID)、执行 (EX)、访存 (MEM) 和写回 (WB) 五个阶段。

2.2 CPU 架构图

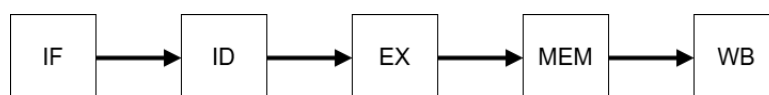


图 1: 五级流水线结构

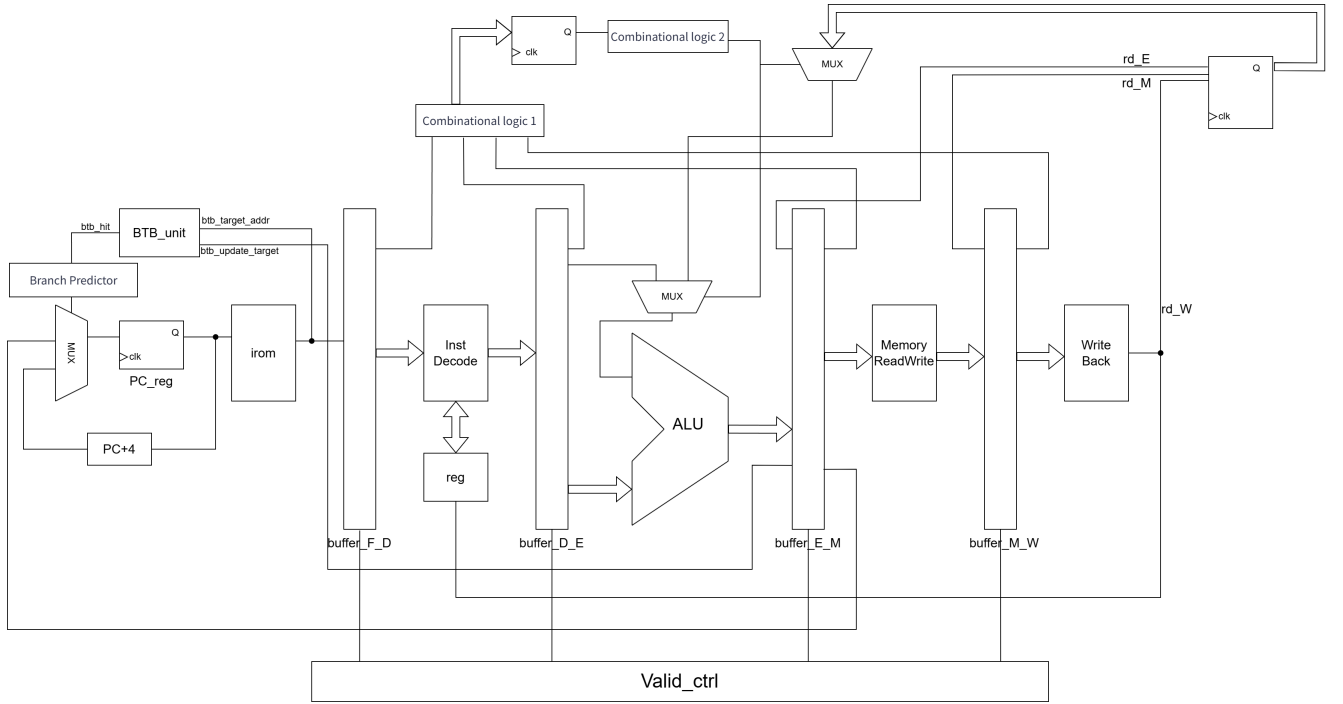


图 2: CPU 架构简化图

2.3 基于握手的流水线时序控制

`valid_ctrl` 模块通过控制各级模块之间的交互来指导数据和控制信号的传输。对于每一级模块，都会产生一个代表自身信号有效的 `valid_X` (`X` 代表执行阶段) 信号和一个代表自身准备好接收信息的 `ready_X` 信号。当且仅当当前级模块的 `valid` 信号有效，且下一级模块的 `ready` 信号有效，即两级之间完成握手时，当前模块的数据信号与控制信号能够传输到下一级模块中的触发器中。也即，握手事件完成的下一个周期，当前级模块可以不用继续保存数据。下面详细说明各级模块的 `valid` 与 `ready` 信号的产生逻辑。

2.3.1 `valid` 信号的产生

对于取指模块，在流水线设计中，我们希望它不间断地工作。在使用 ROM 的理想情况下，每个单周期取指模块都能取到指令，于是我们让 `valid_F` 除了复位情况下恒有效。

对于译码模块、执行模块和写回模块，我们都期望它们能在单一周期内完成工作。也即，在与上一级模块的握手事件结束后，本级模块的 `valid` 应有效，代码逻辑如下：

```
1 always @(posedge clk) begin
2     if (ready_{CurrentStage})
3         valid_{CurrentStage} <= valid_{PreviousStage};
4     end
5 end
```

对于访存模块，情况则比较复杂：我们的 `bram` 时序期望能够在在一个周期内完成写操作，但是读操作需要延迟若干各周期才能完成。得益于 AXI 总线的架构，在读到的数据有效时，会随数据传入一个有效的 `rvalid` 信号，我们可以使用这个信号去指导访存模块的状态转移。我们为访存模块设置了具有如下状态的状态机：

- IDLE: 表示访存模块空闲, 此时, 本模块既没有接收到上一级模块传递的信息, 当前模块的信息也已经通过握手传递给下一级模块。
- LOAD: 表示访存模块正在从 bram 读数据。访存模块从执行模块中接收到读指令, 正在等待 bram 返回 rvalid 信号。
- READY: 表示访存模块的数据已经准备好, 可以向下一级传递, 这包括: 从 bram 接收到了数据、向 bram 写入了数据、无需读写数据。

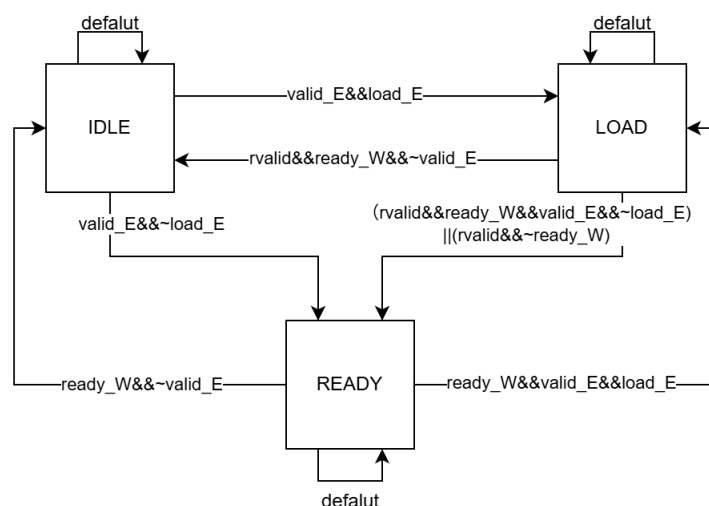


图 3: 访存阶段有效信号控制状态机转移图

状态转移逻辑: 在 IDLE 状态下, 若接收到了 valid_E 信号, 若 load_E 有效, 则跳转到 LOAD 状态, 否则直接跳转到 READY 状态。在 LOAD 状态下, 若没接收到 rvalid 信号, 则保持状态不变; 接收到 rvalid 信号的情况下, 若写回模块未准备好, 则跳转到 READY 状态等待接收; 写回模块准备好的情况下, 若执行模块的 valid 无效, 则跳转到 IDLE 状态; 执行模块的 valid 有效的情况下, 与 IDLE 状态相同, 若下一个指令仍然需要读数据, 则继续保持 LOAD 状态, 若不需要读数据, 直接跳转到 READY 状态。在 READY 状态下, 若写回模块未准备好, 则保持 READY 状态等待接收; 反之, 归类到与 IDLE 状态相同的情况。

valid_M 信号的产生逻辑与当前状态密切相关: 若当前为空闲状态, 数据无效; 若当前为 LOAD 状态, 且 rvalid 已经有效, 代表数据已经取入, 这时候 valid_M 是有效的, 反之, 若 rvalid 无效, 则 valid_M 也无效; 若当前为 READY 状态, 代表着数据已经准备好, valid_M 有效。用代码表述如下:

```
1 valid_M = (state==READY) || ((state==LOAD) && (rvalid==1))
```

2.3.2 ready 信号的产生

ready 信号应从后级模块往前级模块传播。

对于写回模块, 我们期望它能够单周期写回, 因而 ready_W 总是有效。

对于访存模块, 在 IDLE 状态下, ready_M 有效, 其他状态下, 仅当访存模块与写回模块握手时 ready_M 有效, 即:

```
1 ready_M=(state==IDLE) || (valid_M & ready_W)
```

对于执行模块和译码模块，它们与下一级模块握手时，当前级的 ready 信号有效。

特别地，当流水线遇到数据冒险且无法前递数据时，需要将译码模块阻塞，此时译码模块的控制逻辑应为：

```
1 valid/ready_D = (原有逻辑) & (~stall)
```

2.4 基于 BTB 的分支预测器

2.4.1 构建 BTB 模块

通常来说，分支预测器的预测结果需要提供给 IFU 使用：如果预测跳转，则让 IFU 从分支指令的跳转目标处取指，否则，则让 IFU 从 PC + 4 处取指。但实际上，我们在 ID 阶段才能得知一条指令是否为分支指令，如果为分支指令，也需要在 ID 阶段才能得知其跳转目标。而在 IF 阶段中，我们只有 PC 值，难以获得上述信息来进行分支预测。

解决问题的方式有两种：一种是在 IF 阶段加入 minidecode 模块，在取指令阶段就对指令进行部分译码，分辨出指令的跳转指令并计算出跳转地址。这样的设计虽然能够在每次遇到新指令的时候立即识别出指令的类型，但是译码和计算的逻辑会增加取指令阶段的延时，影响取指阶段的速度。另一种就是建立查找表，每次遇到新的跳转指令的时候先不进行预测识别，在新指令执行完成之后再将该指令计算得到的结果存入到 buffer 中，在下一次取值的时候一旦 hit，直接调取历史记录进行处理。

因此，为了提高分支指令的执行效率，本项目设计了基于 BTB（Branch Target Buffer）的分支预测模块。每条分支指令从第二次被取开始，就能够直接调取指令类型和目标地址的信息，这样的设计大大降低了取指阶段所需要付出的时间成本。

除此之外，我们在建立表格的时候添加了一个专门用来记录 jal 指令的标记位，这个标志位保证了从第二次遇到 jal 指令开始，该指令跳转的结果将一直正确。

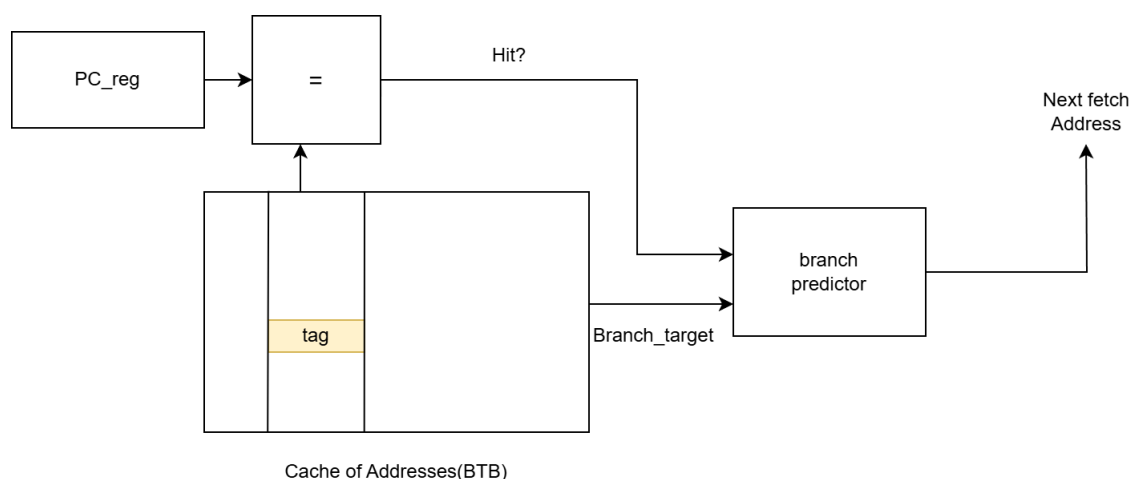


图 4: BTB 的构建

2.4.2 分支预测策略

分支预测的关键是在确定分支指令的类型之后再判断是否进行跳转，本项目中我们一共实现了两种分支预测方式并对比了他们的跳转效果，最终选择了第二种静态跳转。

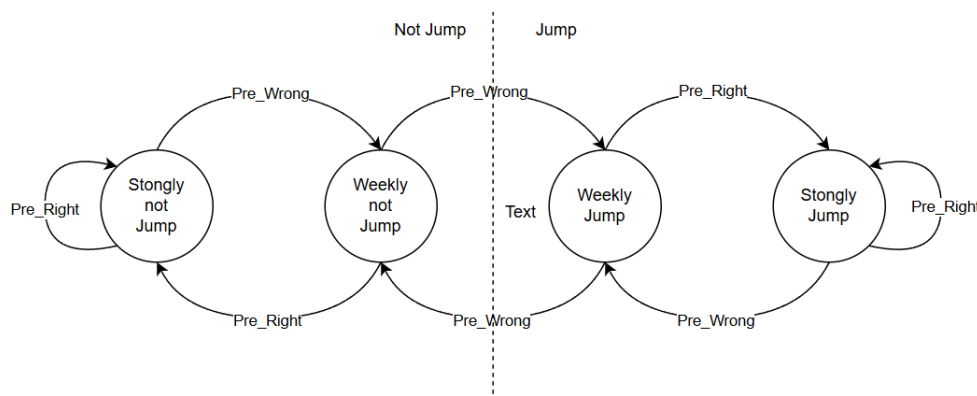


图 5: 2bit 动态跳转预测

2bit 动态跳转 2bit 动态分支预测器是一种常用的分支预测机制。它为每条分支指令维护一个 2 位状态机，通过记录最近分支的执行结果，动态调整预测方向。2bit 状态机共有四种状态，分别表示“强跳转”、“弱跳转”、“弱不跳转”和“强不跳转”。每次分支指令执行后，状态机会根据实际跳转结果进行更新，从而在遇到分支时能够更准确地预测跳转与否。2bit 预测能有效减少由于偶发分支行为变化带来的误判，提高流水线的执行效率。

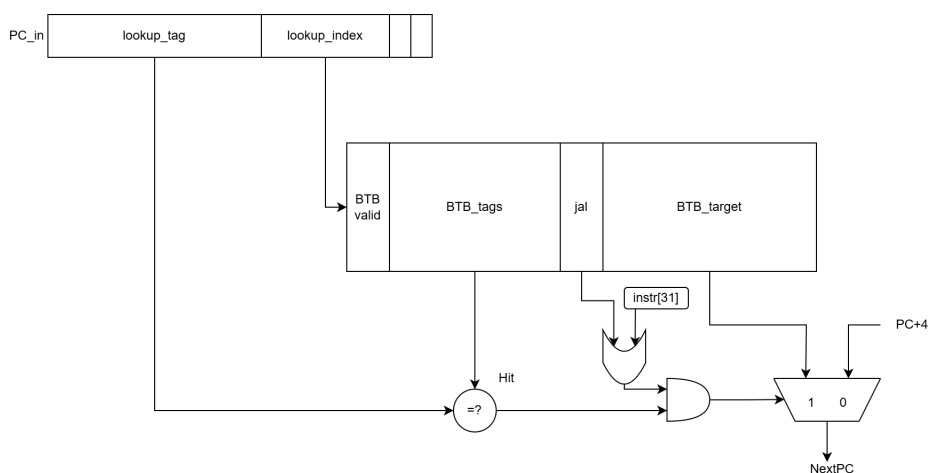


图 6: 构建在 BTB 上的 BTFN 分支预测

BTFN(Backward Taken, Forward Not-taken) 事实上，根据分支跳转方向的不同，分支指令的执行结果是否跳转，是有偏向性的。这其实和程序中循环的行为有关，例如，当分支的跳转目标位于前方时，可能是一个循环的出口，因此偏向不跳转；而当分支的跳转目标位于后方时，可能是要重新执行循环体，因此偏向跳转。一种利用这一特性的静态预测算法称为 BTFN，若分支的跳转目标位于后方，则预测跳转，反之则预测不跳转。实现时，只需要根据 B 型指令 offset 的符号位，即可得到预测结果。事实上，RISC-V 手册也建议编译器按照 BTFN 的模式生成代码：

Software should also assume that backward branches will be predicted taken and forward branches as not taken, at least the first time they are encountered.

分支策略选择 我们进行多次测试的过程中发现，虽然 BTFN 预测的结构更加简单，但是对于编译器产生的文件拥有比 2bit 动态预测更高的预测准确率。在对对运行多个程序的 IPC 进行测试后，我们认为 BTFN 是更好的预测策略。

2.5 两级传递的数据旁路

为了能够得到减少流水线因数据冒险进行等待的时间开销，我们为流水线设计了数据旁路模块。因为我们选取了在上升沿对数据进行写回，所以我们的数据旁路一共有三个起点，分别设在 EX 阶段、MEM 阶段、WB 阶段之后。

我们选择在 ID 阶段对进行数据冲突的检测。RAWdetect__forward 模块主要由两个部分组成：一

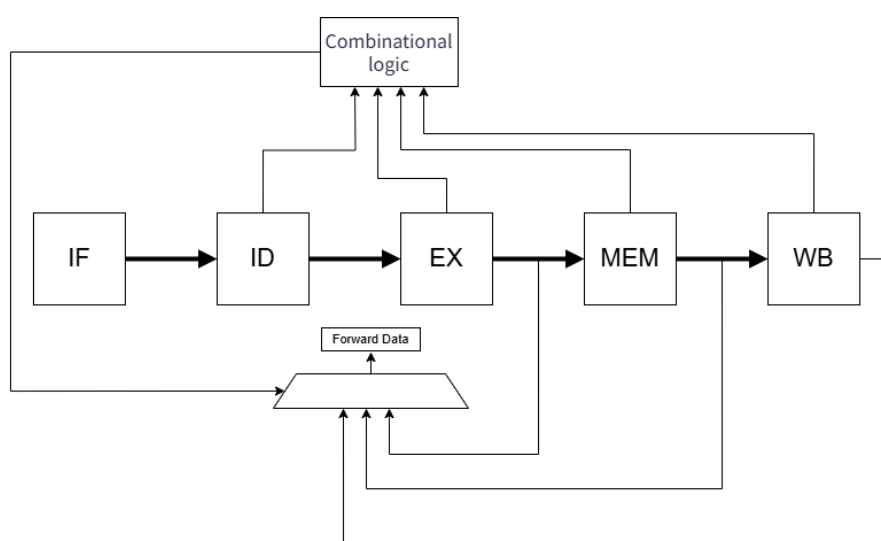


图 7: 数据旁路的初始设计

部分组合逻辑用来判断是否产生数据冲突，产生数据旁路的有效信号和选择信号。另一部分负责输送前递来源的数据到数据选择器。该模块的设计思路如下图 7 所示。

但是在实际的测试中，我们发现，由于我们在判断数据是否有效的时候需要不仅涉及到多个阶段的 valid 信号和 ready 信号，还需要处理来自不同阶段的冲突信息。庞大的组合逻辑需要大量的时间才能稳定下来。不仅如此，等待访存阶段取回的信号也需要付出高的时间成本。正因如此，在提高综合频率之后，我们发现旁路单元成为了整个工程中的关键路径。为了解决旁路单元的时序问题，我们采用了流水线的设计思路，将数据和控制信号的数据通路都进行拆分。分为两个时钟周期完成冲突数据的前递。

进行数据旁路流水化改造之后，前递的数据会比原来晚一个周期才能到达，因此我们虽然从 ID 阶段开始对数据前递进行判断，但是提交的数据需要延迟一步直接提交到 ALU 的起始部分。并修改需要往下传递的寄存器的数值。经过修改之后的旁路构造如下图 8 所示。

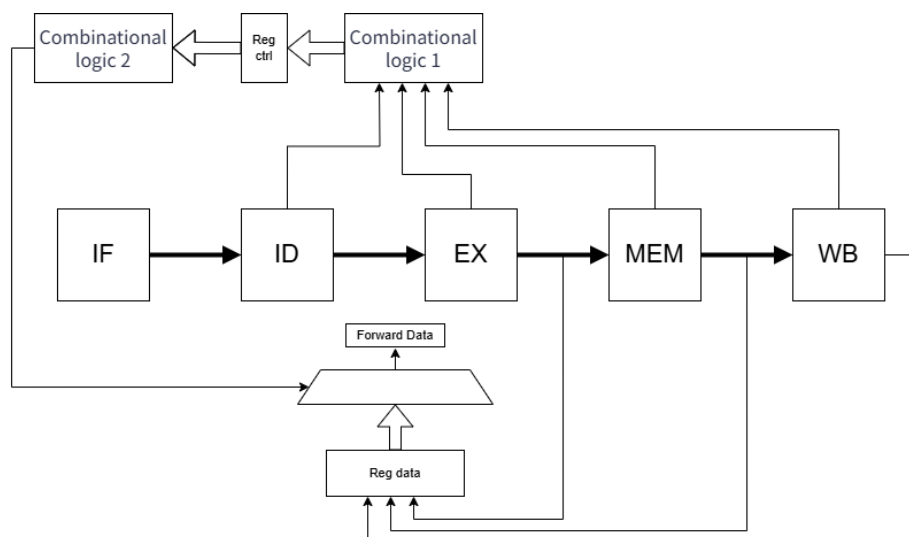


图 8: 流水线改造后的数据旁路

2.6 乘法指令的优势

以 chessc.c 为例，使用乘法指令的情况下，我们可以看到编译出的程序运行所需的周期大大缩短。从六百余万时钟周期减少为五百多万可以看出扩展指令集对于 CPU 对于提升程序运行效率的作用是很明显的。

```

NPC: HIT GOOD TRAP at pc = 0x8000d60
Statistics:
Instructions executed:5666034
Cycles: 6790268
IPC: 0.83
Execution terminated
test list [1 item(s)]: chessc
[      chessc] PASS
  
```

图 9: ARCH=riscv32e-npc 编译的结果

```

NPC: HIT GOOD TRAP at pc = 0x8000c24
Statistics:
Instructions executed:4336999
Cycles: 5428031
IPC: 0.80
Execution terminated
test list [1 item(s)]: chessc
[      chessc] PASS
  
```

图 10: ARCH=riscv32im-npc 编译的结果

2.7 AXI4-Lite 总线接口

本 CPU 设计支持使用 AXI4-Lite 总线接口与外设进行通信，一共设计了可以使用的三组信号并经过了仿真和实际的验证。信号端口定义如下

2.7.1 AXI4-Lite 接口信号定义

CPU 与外设通过 AXI4-Lite 总线进行通信，包括指令接口（Master）和数据接口（Master）。各信号定义如下：

表 1: AXI4-Lite 指令接口信号定义

信号名	位宽	方向	功能描述
inst_axi_araddr	32	output	读地址
inst_axi_arvalid	1	output	读地址有效信号
inst_axi_arready	1	input	读地址准备信号
inst_axi_rdata	32	input	读数据
inst_axi_rresp	2	input	读响应
inst_axi_rvalid	1	input	读数据有效信号
inst_axi_rready	1	output	读数据准备信号

表 2: AXI4-Lite 数据接口信号定义

信号名	位宽	方向	功能描述
data_axi_awaddr	32	output	写地址
data_axi_awvalid	1	output	写地址有效信号
data_axi_awready	1	input	写地址准备信号
data_axi_wdata	32	output	写数据
data_axi_wstrb	4	output	写数据字节使能
data_axi_wvalid	1	output	写数据有效信号
data_axi_wready	1	input	写数据准备信号
data_axi_bresp	2	input	写响应
data_axi_bvalid	1	input	写响应有效信号
data_axi_bready	1	output	写响应准备信号
data_axi_araddr	32	output	读地址
data_axi_arvalid	1	output	读地址有效信号
data_axi_arready	1	input	读地址准备信号
data_axi_rdata	32	input	读数据
data_axi_rresp	2	input	读响应
data_axi_rvalid	1	input	读数据有效信号
data_axi_rready	1	output	读数据准备信号

3 CPU 仿真测试平台设计

3.1 自己搭建测试平台的优势

在本项目中，我们参考一生一芯项目的基础上使用了自己搭建的 CPU 仿真测试平台。其中用于差分测试的参考 CPU 部分和多种调试指令的实现均为我们自主编写，能够完成从交叉编译到仿真测试的全流程。这套系统搭建完成之后很大程度上提升了我们的调试效率。

相较于赛事方提供的 Trace 测试平台，我们的测试平台在支持的追踪功能和使用的编写程度上都有了大幅的提升。我们构建的简易调试器已经能够支持步进运行、内存抓取到变量追踪的多种功能。

并且，我们认为在学习 CPU 设计的过程中，学习自己搭建测试环境对于后续的开发是非常重要的。虽然搭建平台的过程耗时较长，但是这加深了我们对于整个软硬件结合的理解，对于后续的开发具有建设性意义。

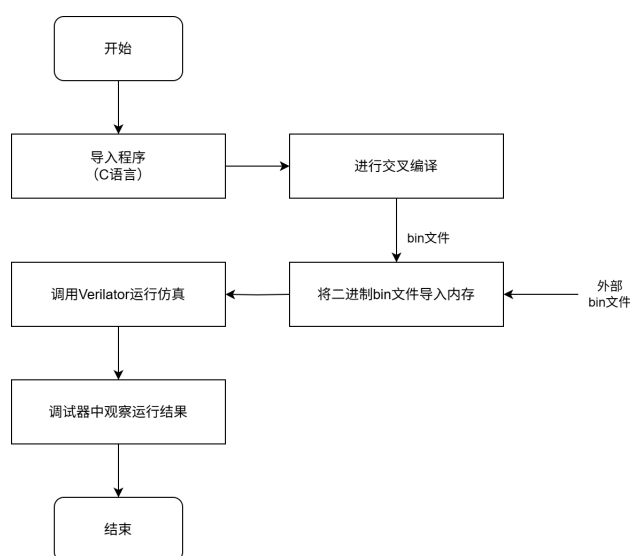


图 11: CPU 仿真测试平台运行全流程

3.2 测试平台可供配置的功能

调试器的主要功能可以通过 config.h 进行配置，各项功能对应表格如下：

```

1 #define EXPR_DEBUG
2 #define CONFIG_ITRACE
3 #define CONFIG_DIFFTEST
4 #define CONFIG_MTRACE
5 #define CONFIG_DEVICE
6 #define MAX_INST_TO_PRINT 10
7 #define MAX_SIM_TIME 1000000
  
```

3.3 调试器支持的指令

完成编译装载之后，可以在调试器中输入指令进行调试。调试器支持的指令如下表所示：

表 3: 调试器配置项说明

配置项	功能说明
CONFIG_ITRACE	指令追踪
CONFIG_DIFFTEST	状态对照测试
CONFIG_MTRACE	内存取取显示
CONFIG_DEVICE	是否进行外设仿真
(目前支持: Printf 输出、竞业达开发板 LED 与数码管显示)	

表 4: 调试器配置项说明

指令	功能说明
c	继续运行直至结束
si N	继续运行 N 条指令
q	退出
info	打印当前状态
x N EXPR	查看内存中某地址的信息
p	使用值进行计算
w	设置程序监测值触发断点
d	删除已经设置过的断点

3.3.1 c 指令（继续运行）

若无错误出现，CPU 将会运行到 ebreak 信号拉高停止仿真。

若出现错误，在开启的 DIFFTEST 功能的情况下，CPU 将会在出现错误的位置停下，在开启 ITRACE 的情况下，打印出错误位置 PC 寄存器数值、指令机器码、指令反汇编结果。同时将会给出参考 CPU 运行的寄存器结果与所设计 CPU 的对照。

```
(npc) c
EBREAK triggered at PC: 80000024
NPC: HIT GOOD TRAP at pc = 0x80000024
Statistics:
Instructions executed:11
Cycles: 14
IPC: 0.79
Execution terminated
```

图 12: 仿真正确运行

3.3.2 si 指令（步进执行）

指定运行 N 条指令，运行过程中逻辑与 c 指令相同。

```

The image is /home/tomoyo/RV32-pipeline/tests/build/add-riscv32im-npc.bin, size = 648
FTrace: OFF
ref_so_file:/home/tomoyo/RV32-pipeline/core/tools/riscv32-nemu-interpret-er-so
Differential testing: ON
The result of every instruction will be compared with /home/tomoyo/RV32-pipeline/core/tools/riscv32-nemu-interpret-er-so. This will help you a lot for debugging, but also significantly reduce the performance. If it is not necessary, you can turn it off by deannotationize the #undef sentence in config.h.
[src/memory/paddr.c:60 init_mem] physical memory area [0x80000000, 0x87ffffff]
ITrace: ON
MTrace: OFF
Welcome to riscv32-NPC!
For help, type "help"
reg[2] mismatch: ref = 0x80009004, dut = 0x7fff7004
reg[0]: ref = 0x00000000, dut = 0x00000000
reg[1]: ref = 0x00000000, dut = 0x00000000
reg[2]: ref = 0x80009004, dut = 0x7fff7004
reg[3]: ref = 0x00000000, dut = 0x00000000
reg[4]: ref = 0x00000000, dut = 0x00000000
reg[5]: ref = 0x00000000, dut = 0x00000000
reg[6]: ref = 0x00000000, dut = 0x00000000
reg[7]: ref = 0x00000000, dut = 0x00000000
reg[8]: ref = 0x00000000, dut = 0x00000000
reg[9]: ref = 0x00000000, dut = 0x00000000
reg[10]: ref = 0x00000000, dut = 0x00000000
reg[11]: ref = 0x00000000, dut = 0x00000000
reg[12]: ref = 0x00000000, dut = 0x00000000
reg[13]: ref = 0x00000000, dut = 0x00000000
reg[14]: ref = 0x00000000, dut = 0x00000000
reg[15]: ref = 0x00000000, dut = 0x00000000
reg[16]: ref = 0x00000000, dut = 0x00000000
reg[17]: ref = 0x00000000, dut = 0x00000000
reg[18]: ref = 0x00000000, dut = 0x00000000
reg[19]: ref = 0x00000000, dut = 0x00000000
reg[20]: ref = 0x00000000, dut = 0x00000000
reg[21]: ref = 0x00000000, dut = 0x00000000
reg[22]: ref = 0x00000000, dut = 0x00000000
reg[23]: ref = 0x00000000, dut = 0x00000000
reg[24]: ref = 0x00000000, dut = 0x00000000
reg[25]: ref = 0x00000000, dut = 0x00000000
reg[26]: ref = 0x00000000, dut = 0x00000000
reg[27]: ref = 0x00000000, dut = 0x00000000
reg[28]: ref = 0x00000000, dut = 0x00000000
reg[29]: ref = 0x00000000, dut = 0x00000000
reg[30]: ref = 0x00000000, dut = 0x00000000
reg[31]: ref = 0x00000000, dut = 0x00000000
PC: ref = 0x80000008, dut = 0x80000008
    0x80000000: 13 04 00 00      mv      s0, zero
    0x80000004: 17 91 00 00      auipc   sp, 9
==>0x80000008: 13 01 c1 ff      addi    sp, sp, -4
    0x80000000: 13 04 00 00      mv      s0, zero
    0x80000004: 17 91 00 00      auipc   sp, 9
==>0x80000008: 13 01 c1 ff      addi    sp, sp, -4
make[2]: *** [Makefile:51: run] Segmentation fault (core dumped)
make[1]: *** [/home/tomoyo/RV32-pipeline/abstract-machine/scripts/platform/npc.mk:30: run] Error 2
test list [1 item(s)]: add
[      add] ***FAIL***

```

图 13: DIFFTEST 发现错误

```

(npc) si
(npc) si
0x80000004: 00 00 91 17 auipc   sp, 9
(npc) si 3
0x80000008: ff c1 01 13 addi    sp, sp, -4
0x8000000c: 0f c0 00 ef jal      0xfc
0x80000010: ff 01 01 13 addi    sp, sp, -0x10
(npc) si 5
0x8000001c: 00 00 05 17 auipc   a0, 0
0x80000010: 01 c5 05 13 addi    a0, a0, 0x1c
0x80000014: 00 11 26 23 sw      ra, 0xc(sp)
0x80000018: f1 1f f0 ef jal      -0xf0
0x80000028: fd 01 01 13 addi    sp, sp, -0x30

```

图 14: si N 运行指定条指令

3.3.3 info 指令（打印变量）

可以打印调试中所需的信息，例如寄存器。

```
(npc) info r
x0: 0x00000000
x1: 0x8000011c
x2: 0x80008ff0
x3: 0x00000000
x4: 0x00000000
x5: 0x00000000
x6: 0x00000000
x7: 0x00000000
x8: 0x00000000
x9: 0x00000000
x10: 0x80000128
x11: 0x00000000
x12: 0x00000000
x13: 0x00000000
x14: 0x00000000
x15: 0x00000000
x16: 0x00000000
x17: 0x00000000
x18: 0x00000000
x19: 0x00000000
x20: 0x00000000
x21: 0x00000000
x22: 0x00000000
x23: 0x00000000
x24: 0x00000000
x25: 0x00000000
x26: 0x00000000
x27: 0x00000000
x28: 0x00000000
x29: 0x00000000
x30: 0x00000000
x31: 0x00000000
```

图 15: 使用 info 指令查看寄存器状态

3.3.4 x 指令（内存查询）

可以查看运行中某个地址处 N 个字节存储的数值（图例：查看 0x80100000 后的 3 个字节）

```
(npc) x 3 0x80000010
match rules[9] = "\x[0-9a-fA-F]+" at position 0 with len 10: 0x80000010
0x80000010: 63 04 05
```

图 16: x N EXPR 指令查看内存

3.3.5 p 指令（算式解析）

可以解析简单的算式，并使用调试中的数值进行计算。支持简单加减乘除的算式符号的解析。

```
(npc) p $a0
match rules[11] = "\${0-9a-zA-Z}+" at position 0 with len 3: $a0
1
(npc) p $pc
match rules[11] = "\${0-9a-zA-Z}+" at position 0 with len 3: $pc
2147483828
(npc) p $pc+$a0
match rules[11] = "\${0-9a-zA-Z}+" at position 0 with len 3: $pc
match rules[12] = "+" at position 3 with len 1: +
match rules[11] = "\${0-9a-zA-Z}+" at position 4 with len 3: $a0
2147483829
```

图 17: p 指令计算

3.3.6 w/d 指令（添加/移除对变量的追踪）

w-指令可以设置想要追踪的寄存器数值，当接下来的运行过程中该数值发生改变的时候中断程序并给出修改前后的数值。d-指令用来清除已经设置好的追踪目标。

```
Watchpoint 0: $pc
(npc) c
match rules[11] = "\\$[0-9a-zA-Z]+" at position 0 with len 3: $pc
Watchpoint 0: $pc
Old value = 2147483828
New value = 2147483832
```

图 18: 设置断点追踪变量变化（例：追踪 PC）

3.4 性能评估功能

在每一段程序运行之后，调试器会在窗口打印总指令条数、运行周期、并计算出 IPC 的数值

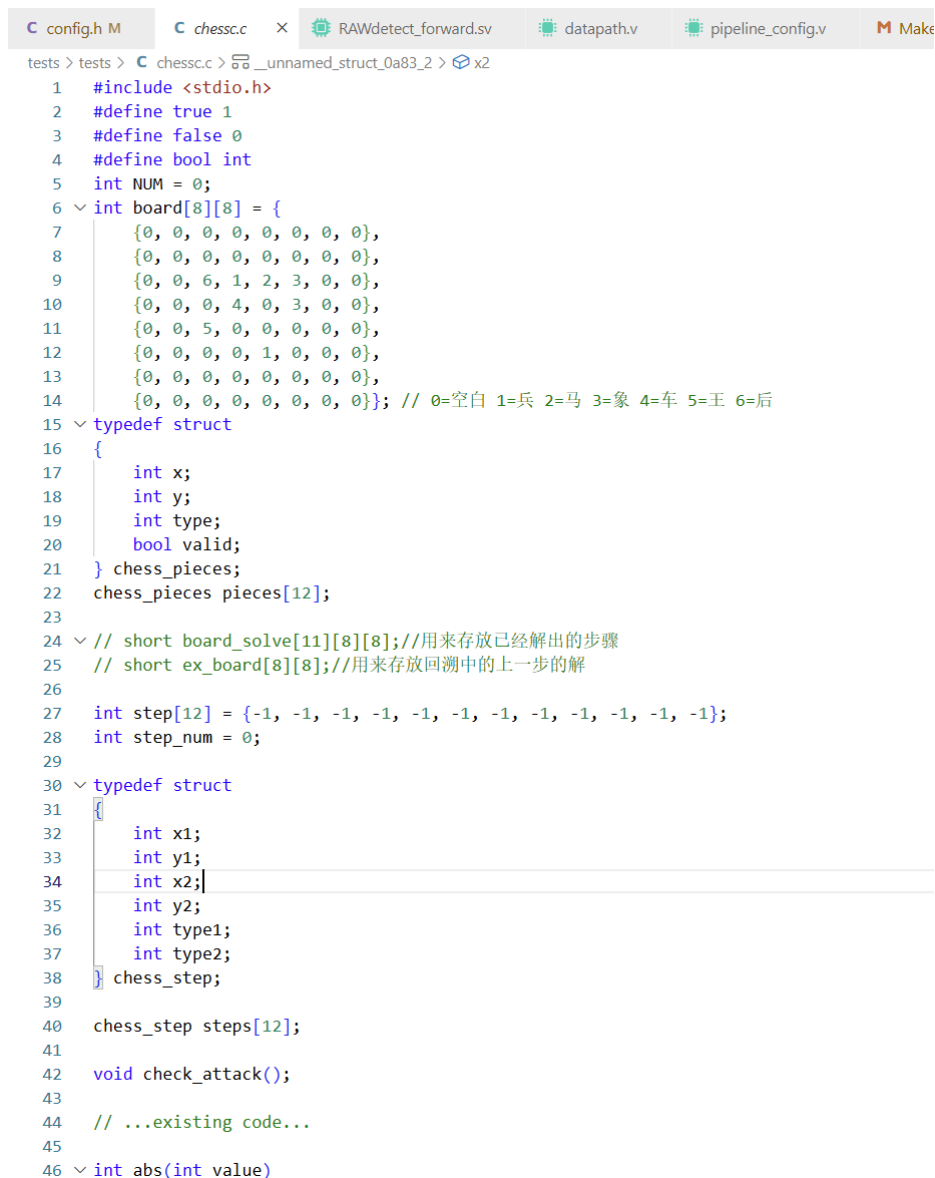
```
EBREAK triggered at PC: 80000120
[src/cpu/cpu-exec.c:141 cpu_exec] nemu: HIT GOOD TRAP at pc = 0x80000120
[src/cpu/cpu-exec.c:108 statistic] host time spent = 76 us
[src/cpu/cpu-exec.c:109 statistic] total guest instructions = 840
[src/cpu/cpu-exec.c:110 statistic] simulation frequency = 11052631 inst/s
NPC: HIT GOOD TRAP at pc = 0x80000120
Statistics:
Instructions executed:841
Cycles: 1024
IPC: 0.82
Execution terminated
test list [1 item(s)]: add
[          add] PASS
```

图 19: 运行测试程序 add 之后得到 Cycle 和 IPC 的结果

3.5 平台调试流程举例

下面以运行我们自己编写的 chessc（寻找独棋游戏的解）为例，展示仿真测试的全流程：

3.5.1 选取想要测试运行的程序



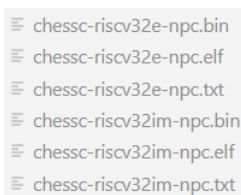
```

1  #include <stdio.h>
2  #define true 1
3  #define false 0
4  #define bool int
5  int NUM = 0;
6  int board[8][8] = {
7      {0, 0, 0, 0, 0, 0, 0, 0},
8      {0, 0, 0, 0, 0, 0, 0, 0},
9      {0, 0, 6, 1, 2, 3, 0, 0},
10     {0, 0, 0, 4, 0, 3, 0, 0},
11     {0, 0, 5, 0, 0, 0, 0, 0},
12     {0, 0, 0, 0, 1, 0, 0, 0},
13     {0, 0, 0, 0, 0, 0, 0, 0},
14     {0, 0, 0, 0, 0, 0, 0, 0}}; // 0=空白 1=兵 2=马 3=象 4=车 5=王 6=后
15 typedef struct
16 {
17     int x;
18     int y;
19     int type;
20     bool valid;
21 } chess_pieces;
22 chess_pieces pieces[12];
23
24 // short board_solve[11][8][8]; //用来存放已经解出的步骤
25 // short ex_board[8][8]; //用来存放回溯中的上一步的解
26
27 int step[12] = {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1};
28 int step_num = 0;
29
30 typedef struct
31 {
32     int x1;
33     int y1;
34     int x2;
35     int y2;
36     int type1;
37     int type2;
38 } chess_step;
39
40 chess_step steps[12];
41
42 void check_attack();
43
44 // ...existing code...
45
46 int abs(int value)

```

图 20: chessc.c 文件

3.5.2 交叉编译后得到需要装载进入参考 CPU 和 CPU 的二进制文件



```

chessc-riscv32e-npc.bin
chessc-riscv32e-npc.elf
chessc-riscv32e-npc.txt
chessc-riscv32im-npc.bin
chessc-riscv32im-npc.elf
chessc-riscv32im-npc.txt

```

图 21: 交叉编译得到的二进制文件

3.5.3 进入目录之后运行 make 进行测试

如下图所示，设计的程序在所设计的 CPU 上正确运行并输出了独棋游戏的正确解。

```
The image is /home/tomoyo/RV32-pipeline/tests/build/chessc-riscv32im-npc.bin, size = 7824
FTrace: OFF
ITrace: ON
MTrace: OFF
Welcome to riscv32-NPC!
For help, type "help"
3 3 6
4 3 1
5 3 2
6 3 3
4 4 4
6 4 3
3 5 5
5 6 1
NUM=8
Solved!
3 3(6)--->4 3(1)
3 5(5)--->4 4(4)
4 3(6)--->4 4(5)
4 4(6)--->6 4(3)
6 4(6)--->6 3(3)
6 3(6)--->5 3(2)
5 3(6)--->5 6(1)
2 5 7 6 4 3 8
Solved!
3 5(5)--->4 4(4)
3 3(6)--->4 3(1)
4 3(6)--->4 4(5)
4 4(6)--->6 4(3)
6 4(6)--->6 3(3)
6 3(6)--->5 3(2)
5 3(6)--->5 6(1)
5 2 7 6 4 3 8
No solution!
EBREAK triggered at PC: 80000c24
NPC: HIT GOOD TRAP at pc = 0x80000c24
Statistics:
Instructions executed:4336999
Cycles: 5428031
IPC: 0.80
Execution terminated
test list [1 item(s)]: chessc
[      chessc] PASS
```

图 22: 完成测试并且可以看到输出的结果

3.6 CPU 仿真实验

3.6.1 自行准备的 C 语言测试集

我们准备了一个包含多个 C 语言程序的测试集，共有三十余段测试程序。在编译过程中选取 riscv32-im 指令集的架构进行编译。

add-longlong.c	crc32.c	hello-str.c	mersenne.c	prime.c	string.c	unalign.c
add.c	div.c	if-else.c	min3.c	quick-sort.c	sub-longlong.c	wanshu.c
bit.c	dummy.c	leap-year.c	mov-c.c	recursion.c	sum.c	
bubble-sort.c	fact.c	load-store.c	movsx.c	select-sort.c	switch.c	
chessc	fib.c	matrix-mul.c	mul-longlong.c	shift.c	test_printf.c	
chessc.c	goldbach.c	max.c	pascal.c	shuixianhua.c	to-lower-case.c	

图 23: 准备的 C 语言测试集

使用 make run 命令开始运行，得到结果全部通过：

```
Execution terminated
test list [37 item(s)]: chessc min3 div bit fact crc32 recursion dummy leap-year hello-str movsx to-lower-case mo
v-c prime shift bubble-sort sum add-longlong shuixianhua max unalign wanshu sub-longlong add load-store pascal ma
trix-mul mersenne test_printf quick-sort if-else fib select-sort goldbach mul-longlong string switch
[ chessc] PASS
[ min3] PASS
[ div] PASS
[ bit] PASS
[ fact] PASS
[ crc32] PASS
[ recursion] PASS
[ dummy] PASS
[ leap-year] PASS
[ hello-str] PASS
[ movsx] PASS
[ to-lower-case] PASS
[ mov-c] PASS
[ prime] PASS
[ shift] PASS
[ bubble-sort] PASS
[ sum] PASS
[ add-longlong] PASS
[ shuixianhua] PASS
[ max] PASS
[ unalign] PASS
[ wanshu] PASS
[ sub-longlong] PASS
[ add] PASS
[ load-store] PASS
[ pascal] PASS
[ matrix-mul] PASS
[ mersenne] PASS
[ test_printf] PASS
[ quick-sort] PASS
[ if-else] PASS
[ fib] PASS
[ select-sort] PASS
[ goldbach] PASS
[ mul-longlong] PASS
[ string] PASS
[ switch] PASS
```

图 24: C 语言测试集运行结果

3.6.2 coremark 测试集

可以跑通 coremark，正确性得到验证。由于在仿真平台跑分，并且没有计时器，评分无意义。

```
Running CoreMark for 1000 iterations
2K performance run parameters for coremark.
CoreMark Size      : 666
Total time (ms)    : 0
Iterations         : 1000
Compiler version   : GCC13.3.0
seedcrc            : 0x59893
[0]crclist         : 0x59156
[0]crcmatrix       : 0x8151
[0]crcstate        : 0x36410
[0]crcfinal        : 0x54080
Finised in 0 ms.
=====
CoreMark PASS      -1 Marks
                   vs. 100000 Marks (i7-7700K @ 4.20GHz)
EBREAK triggered at PC: 80002444
NPC: HIT GOOD TRAP at pc = 0x80002444
Statistics:
Instructions executed:781732030
Cycles: 926723274
IPC: 0.84
Execution terminated
```

图 25: coremark 性能测试运行结果

3.6.3 dhrystone 测试集

可以跑通 dhrystone，正确性得到验证。由于在仿真平台跑分，并且没有计时器，评分无意义。

```
Dhrystone Benchmark, Version C, Version 2.2
Trying 500000 runs through Dhrystone.
Finished in 0 ms
=====
Dhrystone PASS      -1 Marks
                   vs. 100000 Marks (i7-7700K @ 4.20GHz)
EBREAK triggered at PC: 80000ba0
NPC: HIT GOOD TRAP at pc = 0x80000ba0
Statistics:
Instructions executed:271510268
Cycles: 314512484
IPC: 0.86
Execution terminated
```

图 26: dhrystone 性能测试运行结果

3.6.4 南京大学 riscv-tests 测试集

对测试集中的 RV32I、RV32M 指令进行测试，能够通过全部测试集：

```
test list [46 item(s)]: add addi and andi auipc beq bge bgeu blt bltu bne jal jalr lb lbu lh lhu lui lw or ori sb sh simple sll sllw srl srli sub sw xor xori div divu mul mulh mulhsu mulhu rem remu
[      add] PASS
[     addi] PASS
[      and] PASS
[     andi] PASS
[    auipc] PASS
[     beq] PASS
[     bge] PASS
[    bgeu] PASS
[     blt] PASS
[    bltu] PASS
[     bne] PASS
[     jal] PASS
[    jalr] PASS
[      lb] PASS
[     lbu] PASS
[      lh] PASS
[     lhu] PASS
[     lui] PASS
[      lw] PASS
[      or] PASS
[     ori] PASS
[      sb] PASS
[      sh] PASS
[  simple] PASS
[     sll] PASS
[    slli] PASS
[     slt] PASS
[    slti] PASS
[   sltiu] PASS
[    sltu] PASS
[     sra] PASS
[    srai] PASS
[     srl] PASS
[    srli] PASS
[     sub] PASS
[      sw] PASS
[     xor] PASS
[    xori] PASS
[     div] PASS
[    divu] PASS
[     mul] PASS
[    mulh] PASS
[  mulhsu] PASS
[    mulhu] PASS
[     rem] PASS
[    remu] PASS
```

图 27: riscv-tests 测试集运行结果

4 验证报告

4.1 验证平台说明

4.1.1 FPGA 平台型号

竞业达提供开发板（Kintex-7 系列 xc7k325tffg900-2 芯片）

4.1.2 下载/烧录工具版本

Vivado2023.2_Vivado_Hardware_Manager

4.1.3 上电与串口交互测试说明（方式方法）

在本地使用 Vivado 对设计的 CPU 进行综合然后生成比特流文件。通过赛事方提供的端口对远程的开发板进行烧录，最后通过孪生平台观察预先加载好的程序在开发板上的运行结果。

4.2 测试用例说明

4.2.1 指令程序例程

上板验证过程使用赛事方提供的编写好的测试程序。其功能能够检测 RV32I 中前 37 条指令的运行正确性，并且还能够通过运行一段程序所消耗的时间来评估 CPU 的性能。

4.2.2 上板运行流程说明

开发板上成功烧录好本地上传的比特流文件之后，可以观察到 LED 区域显示的通过提示，和在数码管区域显示的通过指令条数（37 条），以及运行后续程序的时间。

4.3 验证结果

表 5: CPU 设计在不同频率下的性能测试结果

设计类型	工作频率	LED 指示	指令正确执行数	执行时间/ms	IPC
单周期	50MHz	√	37	615	1
流水线	50MHz	√	37	928	0.66
流水线	100MHz	√	37	464	0.66
流水线 + 2bit 分支预测	100MHz	√	37	425	0.72
流水线 + 分支 预测 +BRAM	150MHz	√	37	243	0.83
流水线 + 分支 预测 +BRAM	250MHz (超频)	√	37	146	0.84

4.3.1 结果总结

对比以上数据可以验证以下优化 CPU 的方案

流水线 在相同频率（50MHz）下，基础流水线设计的执行时间（928ms）比单周期（615ms）更长，IPC（0.66）低于单周期（1.0）。这说明流水线因指令冲突（如数据冒险、控制冒险）导致效率下降。但流水线允许更高的工作频率（如 100MHz 时执行时间缩短至 464ms），体现了其吞吐量优势。

分支预测 在 100MHz 下，加入 2bit 分支预测后，执行时间从 464ms 降至 425ms，IPC 从 0.66 提升至 0.72。分支预测减少了控制冒险带来的流水线停顿，从而提高了指令吞吐量。

BRAM 引入及时序优化 在 150MHz 和 250MHz（超频）下，结合 BRAM（Block RAM，一种高速存储器）后，执行时间进一步缩短至 243ms 和 146ms，IPC 提升至 0.83 和 0.84。

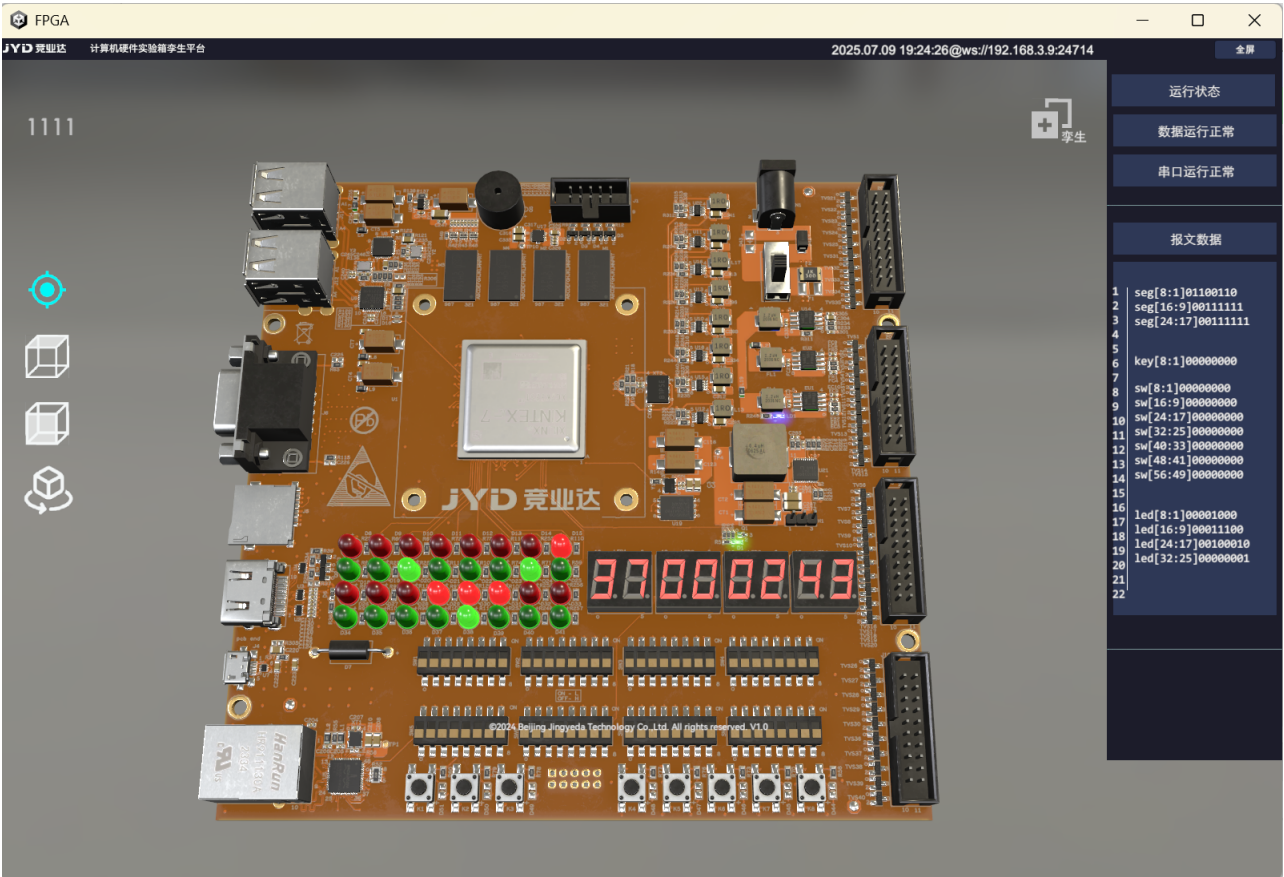


图 28: 最终版本在 150Mhz 频率下的运行结果

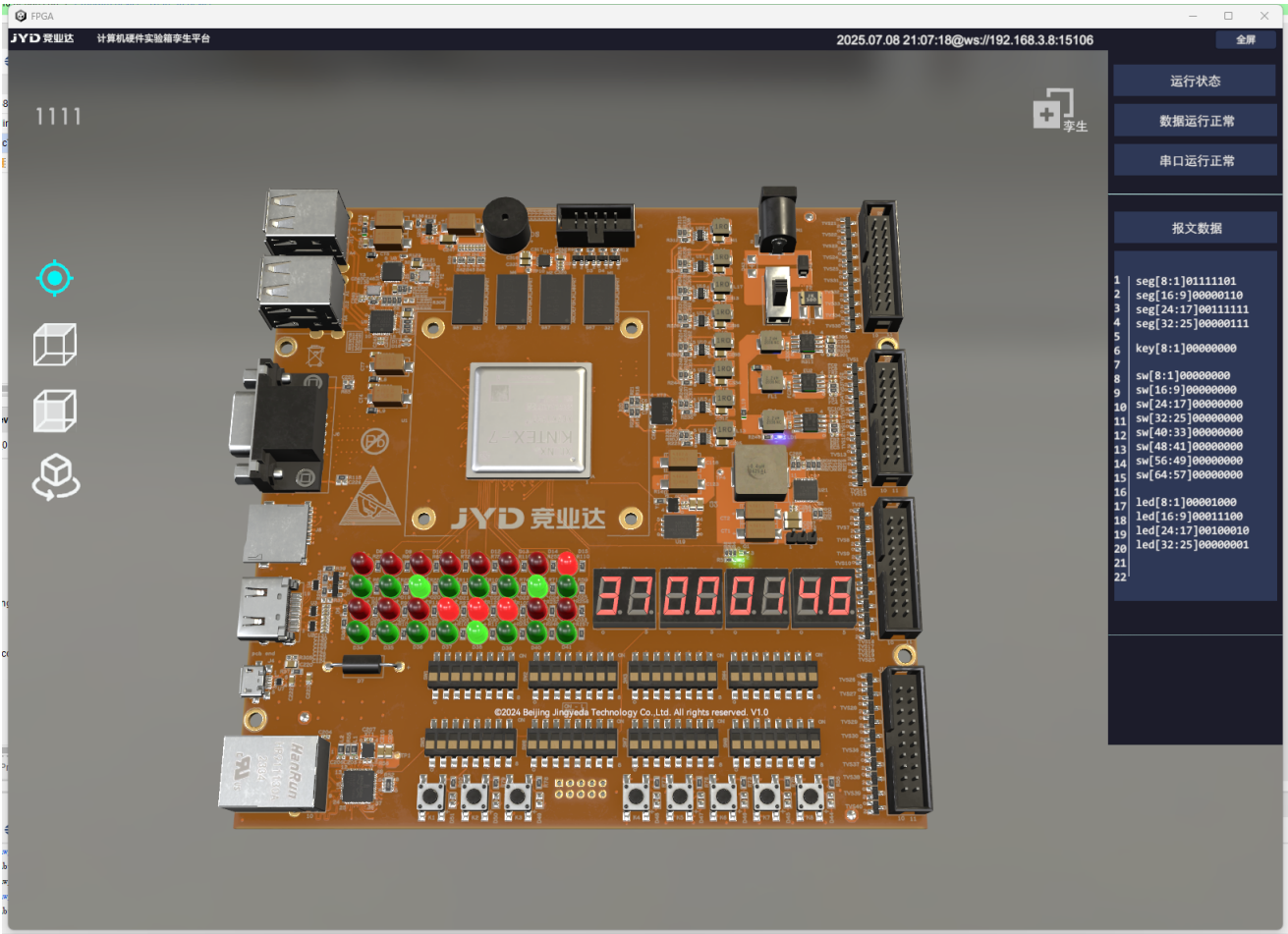


图 29: 最终版本在 250Mhz 频率下的运行结果