

---

# GenshinCPU

## 项目设计报告

---

西北工业大学 2 队

江嘉熙

jshmjjx@mail.nwpu.edu.cn

杨益滔

nwpuyyt@mail.nwpu.edu.cn

魏天昊

weitianhao@mail.nwpu.edu.cn

申世东

seddon@mail.nwpu.edu.cn

# 目录

<b>第一部分 概述</b>	<b>3</b>
1.1 项目简介 . . . . .	3
1.1.1 设计参数 . . . . .	3
1.1.2 系统支持 . . . . .	3
1.1.3 外设支持 . . . . .	3
<b>第二部分 CPU</b>	<b>4</b>
2.1 流水线结构 . . . . .	4
2.1.1 总体架构 . . . . .	4
2.1.2 预取指 (PREIF) 级 . . . . .	4
2.1.3 取指级 (IF) 级 . . . . .	4
2.1.4 译码 (ID) 级 . . . . .	5
2.1.5 执行 (EXE) 级 . . . . .	5
2.1.6 访存 (MEM、MEM2) 级 . . . . .	5
2.1.7 写回阶段 . . . . .	5
2.2 指令集 . . . . .	5
2.3 协处理器 CP0 . . . . .	6
2.4 内存管理 . . . . .	6
2.5 中断和异常 . . . . .	7
2.5.1 中断 . . . . .	7
2.5.2 异常 . . . . .	7
2.6 缓存设计 . . . . .	7
2.6.1 Uncache 处理 . . . . .	7
2.7 分支预测 . . . . .	8
2.8 性能优化 . . . . .	8
2.9 外部接口 . . . . .	8
<b>第三部分 系统与外设</b>	<b>10</b>
3.1 系统移植 . . . . .	10
3.1.1 Pmon . . . . .	10
3.1.2 Ucore . . . . .	11
3.1.3 Linux . . . . .	11
3.2 外设添加 . . . . .	12

<b>第四部分 开发支持工具</b>	<b>14</b>
4.1 自动化构建及测试 . . . . .	14
4.2 Verilator 仿真加速 . . . . .	15
4.3 差分测试框架 . . . . .	15
4.3.1 南大 NEMU 模拟器 . . . . .	15
4.3.2 Soc 仿真环境的搭建 . . . . .	16
4.3.3 差分测试框架的设计 . . . . .	17
4.4 SpyGlass . . . . .	17
<b>第五部分 附录</b>	<b>19</b>
5.1 参考资料 . . . . .	19
5.2 参考仓库 . . . . .	19
5.2.1 17 年参赛作品 . . . . .	19
5.2.2 19 年参赛作品 . . . . .	19
5.2.3 20 年参赛作品 . . . . .	19
5.2.4 其他 . . . . .	19
<b>第六部分 致谢</b>	<b>20</b>

# 第一部分 概述

## 1.1 项目简介

GenshinCPU 是一个实现在龙芯教学实验平台 (Artix-7 XC7A200T) 的基于 MIPS 32 Rev 1 指令集架构的处理器。其包含指令和数据缓存, 其频率达到约 145MHz, 可正常运行 PMON、Ucore 和 Linux 系统。

### 1.1.1 设计参数

GenshinCPU 采用单发射七级流水线架构, 可正确运行 MIPS32 Rev 1 中的 96 条指令, 17 个 CP0 寄存器, 8 种中断。CPU 对外通过 4 个接口进行通信, 分别是 Uncache 的指令和数据接口、Cache 的指令和数据接口。接口采用 AXI3 接口, CPU 使用 2 路组相联 8KB ICache 和 2 路组相联 8KB DCache。CPU 采用使用 8 项全相联的 TLB 进行地址转换, 并使用历史分支查找表 (BHT) 和返回地址栈 (RAS) 进行分支预测, 分支预测整体准确率为 89%。

### 1.1.2 系统支持

GenshinCPU 可正确运行 PMON、Ucore 和 Linux 系统。

### 1.1.3 外设支持

GenshinCPU 修改了官方 SOC 中的部分设计, 目前成功支持 SPI Flash 板载引导芯片, MAC 网络控制器, DDR3 内存控制器, NAND Flash 板载储存, Uart 串口通信 (16550), VGA 显示共 6 种外设。

## 第二部分 CPU

### 2.1 流水线结构

#### 2.1.1 总体架构

GenshinCPU 采用七级流水顺序单发射结构，分别为预取指 (PREIF)、取指 (IF)、译码 (ID)、执行 (EXE)、访存一 (MEM)、访存二 (MEM2)、写回 (WB) 七级。

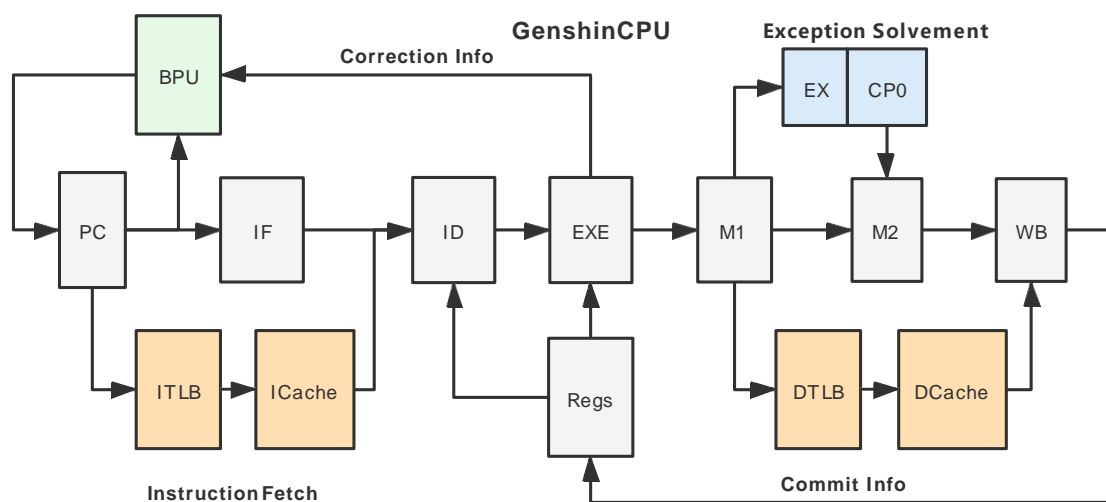


图 2.1: 流水线结构示意图

#### 2.1.2 预取指 (PREIF) 级

预取指阶段将 PC 中存储的指令地址，根据 ICache 的需求拆分为 Tag, Index, Offset 段，由于采用 VIPT 的访问方式，直接将 Index 信号连入 Data Ram 和使用 Lutram 存储的 Tag Ram，Tag Ram 当拍读出数据，Data Ram 下一拍读出数据。同时将 Tag 信号传入 MMU 进行虚实地址转换。将 MMU 输出的实地址和从 Tag Ram 读出的实地址进行比较，计算命中。同时锁存所有的访存信息。

#### 2.1.3 取指级 (IF) 级

从 Data Ram 中读出相应的数据，计算所需数据。同时根据计算命中的结果，给出控制信号。如果缺失，则向 AXI 总线发起访存请求，重填之后提供相应的数据。如果锁存下来的访存信息表示该次访存是 Uncache 的，则启动 Uncache 状态机对外发起请求。

### 2.1.4 译码 (ID) 级

ID 级主要对指令进行译码, 以及读取寄存器堆。在这一级会对数据冒险进行检测, 采用前递和阻塞来解决。这样保证了送往执行级的数据都是正确的数据。与此同时, 我们复用了 19 年清华大学参赛队伍定义的指令枚举变量, 使得代码更具可读性, 降低了代码的复杂度, 借鉴了译码部分, 减少了工作量。

### 2.1.5 执行 (EXE) 级

EXE 级包含的模块主要有 ALU, 以及乘除法器。乘除法均使用 Vivado IP 核, 除法约 37 周期完成, 乘法 3 周期完成。乘除法计算完毕后立即写回 HILO 寄存器。在这一级还会对分支预测进行校准, 根据指令的实际跳转结果来更新 PC。

### 2.1.6 访存 (MEM、MEM2) 级

在我们的设计中访存单元总共分为两个阶段, 在 MEM1 级发起对 DTLB 的访问请求, 以及对 Dcache 的 Tag Ram(LutRam), DataRam(xpm) 发起请求。在 MEM1 级的末尾判断是否命中, 在 MEM2 级获取 Dcache 的数据。对于 TLB 模块, 如果 DTLB 未命中, 则阻塞流水线, 下一周期访问 TLB 模块。如果命中则更新 DTLB, 如果仍然未命中, 则报出异常。对于 DCache 模块, 发出访问请求之后, 在 MEM1 级的末尾阶段会判断该次访问是否命中, 如果命中则在 MEM2 级可以获得访问的数据。如果未命中, 则阻塞流水线, 向 AXI 总线发起访存请求。

### 2.1.7 写回阶段

写回阶段将得到写回结果, 并将数据发送给寄存器。

## 2.2 指令集

CPU 在大赛要求的 57 条指令基础之上, 增加了部分指令以启动操作系统。目前共实现了如下 96 条指令:

- **分支指令** BLTZ, BGEZ, BLTZAL, BGEZAL, BEQ, BNE, BLEZ, BGTZ, JR, JALR, J, JAL, BNEL, BEQL
- **逻辑指令** AND, OR, XOR, ANDI, ORI, XORI, NOR, SLL, SRL, SRA, SLLV, SRLV, SRAV
- **算术指令** ADD, ADDU, SUB, SUBU, ADDI, ADDIU, MUL, MULT, MULTU, DIV, DIVU, MADD, MADDU, MSUB, MSUBU, CLO, CLZ
- **访存指令** SB, SH, SW, LB, LH, LW, LBU, LHU, LWL, LWR, SWL, SWR
- **特权指令** SYSCALL, BREAK, TLBP, ERET, MTC0, MFC0, TLBWI, TLBWR, TLBR
- **条件移动指令** SLT, SLTU, SLTI, SLTIU, MOVN, MOVZ
- **无条件移动指令** LUI, MFHI, MFLO, MTHI, MTLO

- **自陷指令** TEQ, TGE, TGEU, TLT, TLTU, TNTEQ, TGE, TGEU, TLT, TLTU, TNE, TEQI, TGEI, TGEIU, TLTi, TLTiU, TNEI
- **系统控制指令** SYNC, CACHE , PREF

## 2.3 协处理器 CP0

CPU 实现了 MIPS 32 Rev 1 规范中协处理器 0 中的大部分寄存器，同时为了启动操作系统，实现了 MIPS 32 Rev 2 规范中的 EBase 寄存器。所有寄存器如下，名称摘录自参考资料 3:

- BadVAddr Register (CP0 Register 8, Select 0)
- Count Register (CP0 Register 9, Select 0)
- Status Register (CP Register 12, Select 0)
- Cause Register (CP0 Register 13, Select 0)
- Exception Program Counter (CP0 Register 14, Select 0)
- Index Register (CP0 Register 0, Select 0)
- Random Register (CP0 Register 1, Select 0)
- EntryHi Register (CP0 Register 10, Select 0)
- EntryLo0, EntryLo1 (CP0 Registers 2 and 3, Select 0)
- Context Register (CP0 Register 4, Select 0)
- PageMask Register (CP0 Register 5, Select 0)
- Wired Register (CP0 Register 6, Select 0)
- Compare Register (CP0 Register 11, Select 0)
- Processor Identification (CP0 Register 15, Select 0)
- EBase Register (CP0 Register 15, Select 1)
- Conguration Register (CP0 Register 16, Select 0)
- Conguration Register 1 (CP0 Register 16, Select 1)

## 2.4 内存管理

CPU 使用内存管理单元 (MMU) 以及相应的地址映射关系来进行虚地址到实地址的转换。针对 TLB 的设计，我们参考了 19 年中国科学院大学 (UCAS) 参赛队伍的 TLB Buffer 的设计。我们设计了独立的 ITLB Buffer 和 DTLB Buffer，它们与 TLB 进行交互。若 Buffer 发生缺失，将阻塞流水线一拍，并对 TLB 进行查找，将查找结果填回 Buffer。这使得我们的 CPU 在使用 8 项全相联的 TLB 的同时，仍然能够具有较高的频率。

## 2.5 中断和异常

我们的设计中, 在 MEM1 级, 我们对 CP0 完成读写操作, 并实现了对于精确异常的处理. 具体如下:

### 2.5.1 中断

CPU 支持两个软件中断 (SW0 SW1), 6 个硬件中断 (HW0 HW5), 一个计时器中断, 计时器中断复用 HW5 硬件中断。

### 2.5.2 异常

CPU 实现了 MIPS 规范的精确异常, 在访存级统一提交异常。CPU 中实现的异常如下:

- TLB Refill Exception
- TLB Invalid Exception
- TLB Modified Exception
- Overflow Exception
- System Call Exception
- Breakpoint Exception
- Reserved Instruction Exception
- Coprocessor Ununable Exception
- Trap Exception
- Address Error Exception

## 2.6 缓存设计

CPU 实现了 ICache 和 DCache, 分别响应 CPU 的取指和访存请求。在 Cache 设计上, 我们实现了 2 路组相联 8KB ICache 和 2 路组相联 8KB DCache。存储的 RAM 使用 Xilinx XPM 简化了 ip 核的部署过程。使用 LutRam 能够当拍读出 Tag 计算命中。与此同时, 我们借鉴了 19 年清华大学参赛队伍 (编程是一件很危险的事情队) 参赛作品中的 "mux\_byteenable" 函数, 减少了 sb,sh 指令的压力。在替换策略的实现上, 我们也参考了该队伍参赛作品中的可配置 PLRU 功能部件。

### 2.6.1 Uncache 处理

我们通过 Verilator 仿真抓取了 CPU 在性能测试时的所有访存序列, 发现了程序存在着较多的 Uncache 写同一地址, 而较少的间杂着 Uncache 读, 同时因为性能测试没有 TLB, 地址的 Cache 属性并不会随着程序的运行改变, 因此我们为 Uncache 写加入了 FIFO, 极大的减少了流水线阻塞的周期数, 如有 Uncache 读需要等到 Uncache 写完才能进行读。



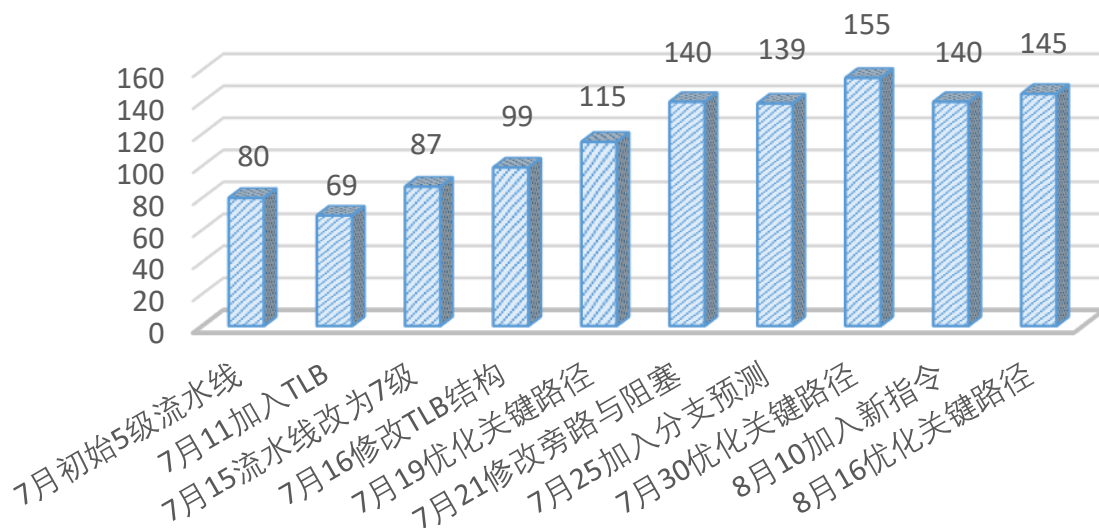


图 2.2: 性能优化过程

## 2.7 分支预测

在分支预测的设计上,我们参考了 20 年哈尔滨工业大学(深圳)的分支预测方式。采用 BHT(Branch History Table)+RAS(Return Address Stack) 的结构,其中 BHT 为 256 项的历史分支查找表, RAS 为返回地址堆栈。我们将分支预测拆成查找和判断两级,使 CPU 能够保持较高的频率。分支预测的整体准确率为 89%。

## 2.8 性能优化

在性能优化上,我们主要做了以下工作:

1. Cache 命中率提升。
2. 分支预测准确率提升。
3. 频率提升。

我们 CPU 频率的提升主要依靠优化关键路径。在优化过程中,主要使用**寄存器平衡**,**消除代码优先级**,**关键信号后移**等方法。

## 2.9 外部接口

CPU 和外部交互如图2.3 所示,为了整齐的管理 Cache 以及 Cpu 对外接口的访存,我们在 AXI 模块中,通过一组接口与 Cache 模块进行交互。这实际上就是将 Cache 的部分功能分离出来,缩小每一模块的职责,使得调试与编写时更为容易。CPU 对外有四个 AXI 接口,分别为经过缓存的 ICache、DCache 与不经过缓存的 ICache、DCache。为了满足大赛要求,我们使用了 Xilinx 提供的 AXI Crossbar IP,其包含 4 个 Slave 接口和一个 Master 接口,并将四个 AXI 接口整合为一个引出到外部 SOC 上。四个接口的优先级为 Uncached-Data > Uncached-Inst > DCache > ICache。

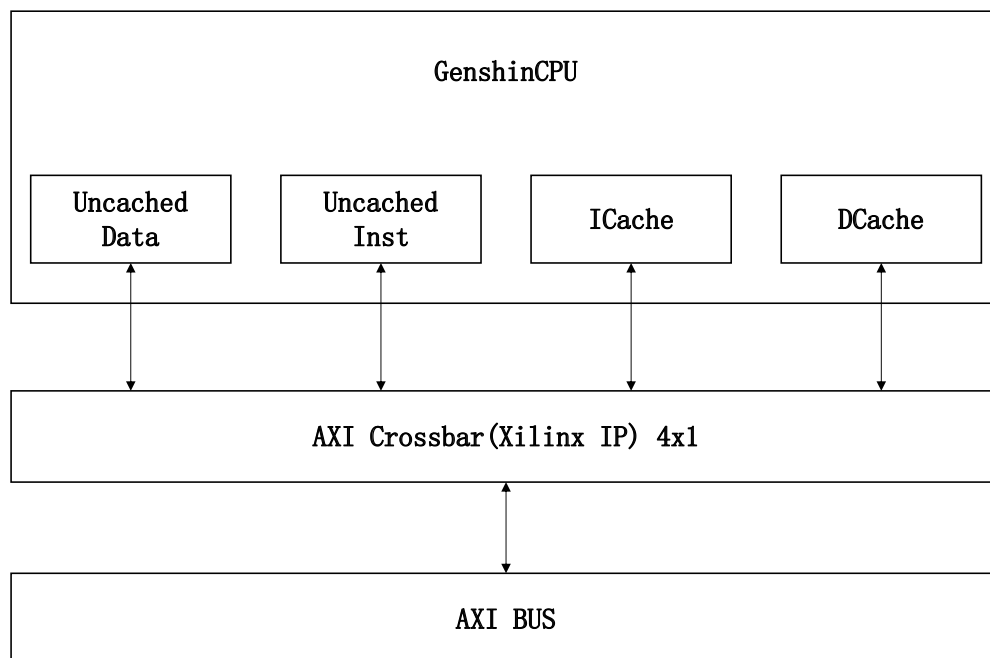


图 2.3: CPU 与外部通信

## 第三部分 系统与外设

### 3.1 系统移植

当前 GenshinCPU 已经成功进行了 PMON、Ucore 和 Linux 的移植。以下固件的编译均于如下环境下进行：

1. 交叉编译环境：龙芯 gcc-4.3-ls232(mipsel)
2. 系统环境：Ubuntu 20.04/18.04 64bit
3. 软件调试器：NEMU-MIPS

#### 3.1.1 Pmon

在我们重新对 Pmon 进行编译的过程中，我们发现需要通过选择ls\_1b做为目标设备，否则将不能编译出可以在实验板上正常运行的 PMON。同时通过在源代码中加入诸多调试信息，以及在线调试和反汇编，最终正常运行了 PMON。我们在调试中遇到了下述较为典型的问题：

**指令去除** 在我们正确实现 Branch Likely 指令之前，NAND FLASH 由于上述指令没有正确实现，并不能继续往下执行。我们通过注释 Targets/LS1B/ls1b/tgt\_machdep.c 中的 ls1g\_soc\_nand\_init(); 当然在这之后我们发现一个有趣的问题，在第一次烧龙芯自带 Bit 流之后，再烧我们的 Bit 流，便可正常进入 PMON。但是如果第一次上电直接烧入我们的 Bit 流，则无法正确进入系统，在对 PMON 的源代码进行研读后我们发现原因可能是注释掉 NAND FLASH 初始化函数之后，本来一个设置 PMON 有无登陆密码的环境变量无法正确读取，造成 PMON 误以为系统自带密码，最终卡住无法正确执行。如果需要直接进入，则需要注释掉位于 pmon/common/main.c 中的 check\_user\_password() 即可。

**NAND FLASH** 在 PMON 初始化 NAND FLASH 的过程中，我们出现了 TLB MISS STORE 的问题，一开始我们认为是 Cache 指令实现的原因，后来经过 Trace 对比发现，是由于 Branchlikely 的错误实现，将本来要进行的 SP+4 指令刷走了，导致 lw 填入 ra 的地址不对，最终导致了执行了错误的函数，执行了一条不该出现的 sw，引起了 TLB MISS STORE，这是我们在不同版本之间的代码 merge 时，忽视了 BranchLikely 指令的细节，而引入的问题。

我们在最终实现 Branch Likely 指令后，上述问题得以解决，目前我们的 CPU 可以直接运行大赛资源包内原版的 PMON 并 Load 其他操作系统。

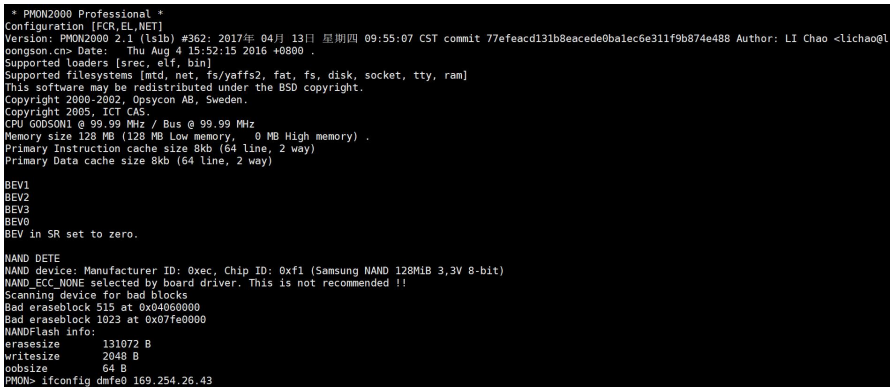


图 3.1: PMON 运行截图

3.1.2 Ucore

GenShinCPU 在进行 Ucore 移植的过程中，主要通过差分测试框架对系统进行移植和 CPU 调试，主要遇到了 Cache 取指出错和控制寄存器没有实现用户态的问题，具体细节详见 4.3 节差分测试框架。下图是我们正确进入 Ucore 的截图：



图 3.2: Ucore 运行截图

3.1.3 Linux

在进行 Linux 移植的过程中，我们遇到了诸多困难。查阅参考资料后我们的移植解决了下列问题。我们在对 Linux 进行移植的过程中，遇到了非常多的困难，最终在决赛提交 8 个小时前成功解决最后一个 Bug，成功启动了 Linux 系统，以下是我们通过查阅并解决的问题。

**指令集精简** 在 CPU 指令实现上，我们并没有实现 ll/sc, 和 branchlikely 等指令，因此需要对 Linux 内核做部分裁剪。ll/sc 指令去除：ll/sc 指令用于实现变量的原子访问，在 GenShinCPU 中并没有实现，故通过 #define cpu\_has\_llsc 0 去除 ll/sc 指令。WatchLo 和 WatchHi 寄存器去除：GenShinCPU 没有实现，故通过 #define cpu\_has\_watch 0 去除。

Branch Likely 类指令去除：在 arch/mips/Makefile 中增加 mno-branch-likely 编译选项，以避免产生这些指令。在 TLB 指令精简方面，我们参考了清华大学 17 年龙芯杯的报告中相关内容。使用软件模拟随机数的方式将所有的 TLBWR 指令全部用 TLBWI 替换。

**外部配置问题和 Ramdisk 制作** 此类问题都是容易出问题的外部配置问题，在调试内核及 CPU 代码时很容易导致定位问题错误，浪费大量时间导致进度滞后。这也从侧面说明了代码习惯及行为记录的重要性。在重新编译 vmlinux 时，需要选择 ls232\_defconfig 做为编译的 config 配置文件，但是 ls232 平台应该是可变页的，其 config 内使用 16KB 做为 PageSize 大小，但是由于 CPU 实现的不同，如果 ls232 的页大小并不是 16K 的，则需要使用 make menuconfig 对页大小进行修改，否则会出现偶数页 CP0\_EntryLo0 的数据始终为 0 的情况。

同时，大赛提供的 ramdisk.cpio 中包含很多我们没有实现的指令，因此我们需要对 ramdisk 也进行修改。我们直接采用了 17 年清华大学参赛队伍开源的 Busybox 并对其进行部分修改，主要是由于其仓库内 Linux kernel 版本较高，其仓库 Busybox 虽然可以适配 2.6 版本的 Kernel，但是我们需要对其新建 dev 文件夹并加入 console 和 null 两个设备，这样才能正确启动。

```
io scheduler cfg registered (default)
Serial: 8250/16550 driver, 4 ports, IRQ sharing disabled
serial8250.0: ttyS0 at MMIO 0x1fe40000 (irq = 163) is a ST16650
console [ttyS0] enabled, bootconsole disabled
console [ttyS0] enabled, bootconsole disabled
brd: module loaded
m25p80 spi0.0: m25p80 (1024 Kbytes)
ITC MAC 10/100M Fast Ethernet Adapter driver 1.0 init
TCP cubic registered
NET: Registered protocol family 17
drivers/rtc/hctosys.c: unable to open rtc device (rtc0)
Freeing unused kernel memory: 1352k freed
Algorithmics/MIPS FPU Emulator v1.5

WELCOME TO
NJU GENSINCPU!

init started: BusyBox v1.23.2 (2016-05-13 03:51:29 CST)

Please press Enter to activate this console.
# echo "Hello World!!!"
Hello World!!!
#
```

图 3.3: Linux 运行截图

## 3.2 外设添加

**VGA** 我们在大赛提供的 soc\_up 上进行修改，添加了对于 VGA 显示的支持。我们使用的 VGA 控制模块根据南大 NJU-MIPS 的开源 VGA 控制器修改而来，能够在复位后显示初始图片。初始图片以 coe 的形式放置在 ROM 中。我们还对 AXI MUX 进行扩展，给 VGA 设置了帧缓存，分配地址为 BFA0\_0000。CPU 可以通过 Store 指令修改帧缓存来显示内容。

**中断与地址映射** GenshinCPU 支持 6 个外部中断信号输入，其中最高位 IP7 外接中断为空，为时钟中断复用。其中断和我们支持的地址空间如下表所示：

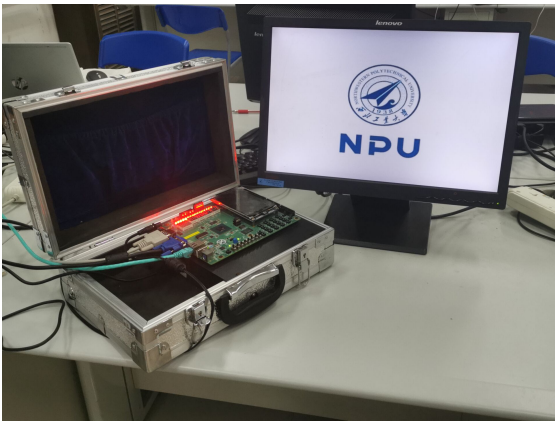


图 3.4: VGA 显示展示

硬件中断名称	来源
CauseIP7	GND: 时钟中断复用
CauseIP6	dma_int: NAND DMA 的中断请求
CauseIP5	nand_int: NAND FLASH 的中断请求
CauseIP4	spi: SPI FLASH 的中断请求
CauseIP3	uart0_int: 串口的中断请求
CauseIP2	mac_int: 网口的中断请求

表 3.1: 中断

控制器模块名	分配的虚拟地址段	地址空间大小
SPI Flash	0xBFC0_0000~0xBFCF_FFFF 0xBFE4_0000~0xBFE_4FFFF	1MB
GPIO	0xBFD0_0000~0xBFD0_FFFF	64KB
MAC	0xBFD0_0000~0xBFD0_FFFF	64KB
DDR3	0x8000_0000~0x87FF_FFFF	128MB
Nand Flash	0xBFE4_8000~0xBFE7_BFFF	16KB
Uart	0xBFE4_0000~0xBFE4_3FFF	16KB
VGA	0xBFA0_0000~0xBFAF_FFFF	1MB

表 3.2: 地址映射

## 第四部分 开发支持工具

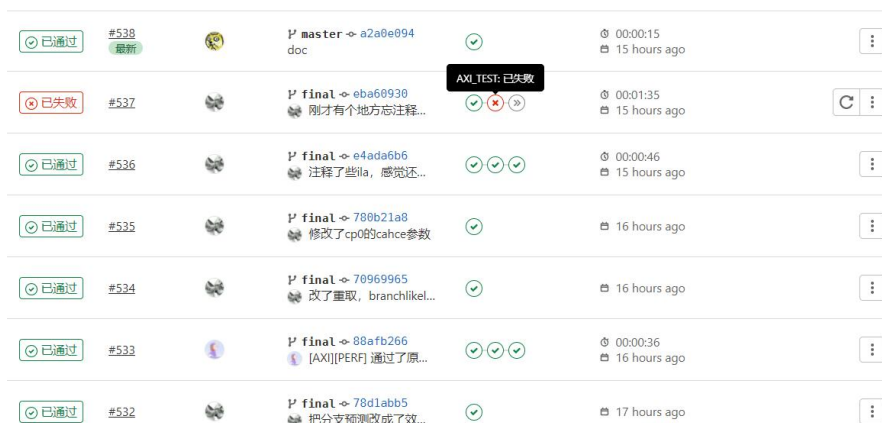
为了保证项目实现的准确无误，我们的项目实现了自动化构建及测试 (Gitlab CI)，保证每次提交的代码都是正确，可以快速复现。同时，由于仿真耗时过长，我们将我们的代码和 Verilator 进行了相关适配，以大幅度减少仿真时间。我们还将南京大学开源项目 NEMU 移植到了大赛平台，通过 NEMU 和 Verilator 进行联合差分测试，迅速定位到 CPU 的相关 Bug。

### 4.1 自动化构建及测试

通过清华 Nontrival-MIPS 的参赛经验分享，我们得知了可以通过 Gitlab CI 来通过服务器对整个代码进行仿真和综合，这在我们的开发初期大幅度减少了我们的仿真时间和出错概率。通过参考并重写清华大学参赛队伍 Nontrival-MIPS 开源的自动构建脚本，并在一台阿里云服务器（2 核 8G）上成功部署了 CI 环境，我们的 CI 可以完成如下功能：

1. 按需完成功能、性能的仿真测试。
2. 完成 FUNC/PERF bitstream 和时序报告的生成，并以产物的形式打包的平台以供下载。

通过自动运行相关测试，帮助我们解决了多人协同下代码版本管理的相关问题，减少了因为代码管理混乱而带来的错误。



已通过	#538	最新	P master → a2a0e094 doc	00:00:15 15 hours ago	⋮
已失败	#537		P final → eba66930 刚才有个地方忘注释...	00:01:35 15 hours ago	⋮
已通过	#536		P final → e4ada6b6 注释了些lla, 感觉还...	00:00:46 15 hours ago	⋮
已通过	#535		P final → 780b21a8 修改了cp0的cahce参数	16 hours ago	⋮
已通过	#534		P final → 70969965 改了重取, branchlike...	16 hours ago	⋮
已通过	#533		P final → 88afb266 [AXI][PERF] 通过了原...	00:00:36 16 hours ago	⋮
已通过	#532		P final → 78d1abb5 把分支预测改成了效...	17 hours ago	⋮

图 4.1: CI 使用记录

整个开发过程中，我们一共通过 Gitlab CI 进行了执行了 483 条流水线，其中成功 389 条流水线，失败 45 条流水线，成功率为 89.63% 节省了大量的时间。

## 4.2 Verilator 仿真加速

Verilator 是一个支持 Verilog/SystemVerilog 的周期精确 (cycle-accurate) 开源仿真器。之所以采用 Verilator 进行仿真, 是因为其仿真速度相较 Vivado 加快了几十倍。以性能测试 CoreMark 为例, 在不开波形记录的情况下, Verilator 只需 3 秒即可完成仿真测试, 而这一过程在 Vivado 中则需要十分钟左右。

在执行原理上 Verilator 将用 Verilog/SystemVerilog 的 RTL 级描述的模块 (module) 综合为一个 C++ 模型。这个 C++ 模型一般称为 Verilated Model。

我们通过使用 Verilator 开源工具, 对我们的代码写法进行一定量的修改, 并将串口外设 (NS16550A) 适配到仿真平台, 通过构造一个 SoCLite 的 C++ class。然后再通过编写 C++ 代码来提供模型的输入, 以及检查模型的输出。在大部分情况下, 这种仿真方式甚至可以替代上板验证, 提高了调试效率。

```
now pc 15 0
now pc 15 bf2760b
now pc 15 bf27730
now pc 15 bf27f80
now pc 15 0
now pc 15 bf085d4
----- Number 88 Functional Test Point 005111
now pc 15 0
now pc 15 bf080ac
now pc 15 bf08084
now pc 15 0
now pc 15 0
now pc 15 0
now pc 15 bf080ad
----- Number 89 Functional Test Point 005111
now pc 15 bf0802c
-----Test End
Pass!!!
root@top118adhyb15988u2hy2:/net/soo_esp_ql/soo15a-40000/
```

(a) 功能测试

```
-----
ghrystone test begin.
Ghrystone Benchmark, Version 2.1 (Language: C)

Ghrystone Benchmark, Version 2.1 (Language: C)
Program compiled without "register" attribute
Execution starts, 10 runs through Ghrystone
Execution ends.

Final values of the variables used in the benchmark:

int_glob: 5
  should be: 5
bool_glob: 1
  should be: 1
ch_1_glob: A
  should be: A
ch_2_glob: 0
  should be: 0
arr_1_glob[0]: 7
  should be: 7
arr_2_glob[0][7]: 20
  should be: 20
Ptr_glob->
  PtrComp: 1054308132
  should be: (implementation-dependent)
  Discr: 0
  should be: 0
EnumComp: 2
  should be: 2
IntComp: 17
  should be: 17
```

(b) 性能测试

图 4.2: 功能和性能测试的 Verilator 仿真

## 4.3 差分测试框架

在启动操作系统方面, 为了避免上板抓信号, 头脑 Debug 的情况频繁出现, 我们参考了国科大一生一芯开源项目 NutShell, 以及南京大学 PA 实验, 搭建了一套基于 **Verilator 仿真器以及 NEMU 模拟器的 MIPS 指令集差分测试框架**。在仿真环境下, 成功启动了 PMON 以及 Ucore, 而在启动 Linux 时, 由于其存在部分难以模拟的外设, 因此在仿真环境下只能进行部分的 Linux 差分测试, 但这也帮助我们快速定位了在启动 linux 前期, CPU 上存在的一些 BUG。

### 4.3.1 南大 NEMU 模拟器

我们参考南京大学 PA 课程, 有关 MIPS32 指令集的内容, 重写了模拟器的部分代码, 并根据官方给出的 SOC 地址空间映射关系, 对 NEMU 模拟器地址映射做了必要的调整, 最终成功启动 PMON, Ucore, 我们主要做出了以下修改。

1. 原本 NEMU 模拟器使用了 Xilinx UartLite 做为串口控制器, 分配的地址空间为 0xBFEE50000 - 0xBFEE50FFF 而 SOC 上使用的串口控制器为 NS16550A, 其地址空间为 0xBFEE40000 - 0xBFEE43FFF。因此我们参考系统测试文档中有关 NS16550A 串口控制寄存器的描述, 重写了串口读写相关函数。在软件上实现了 **0 号数据传输寄存器**



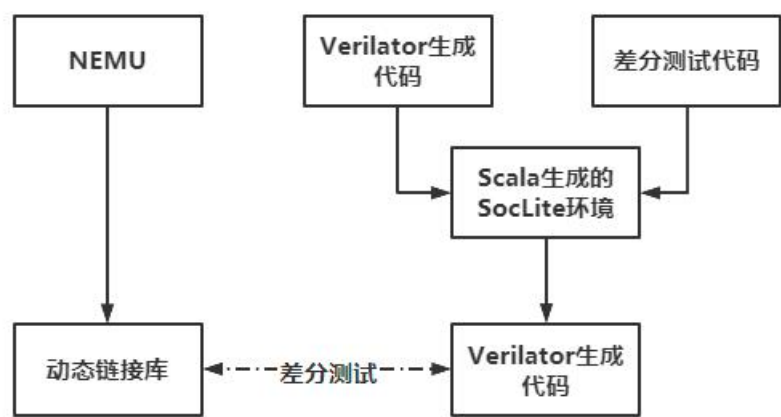


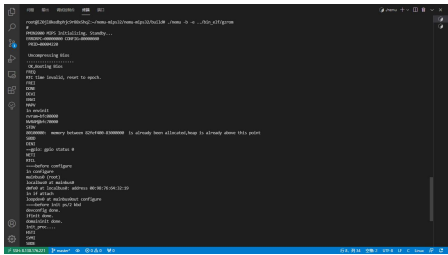
图 4.3: 差分测试框架

UART\_DAT，以及 5 号线路状态寄存器 UART\_LSR，每次写 UART\_DAT 寄存器时，就将数据通过 printf 输出至终端，每次读 UART\_LSR 寄存器时，返回固定值 0x20(表示传输 FIFO 为空，可以接受新数据的写入)，这样，即可模拟在 SOC 上操作系统对串口的打印。

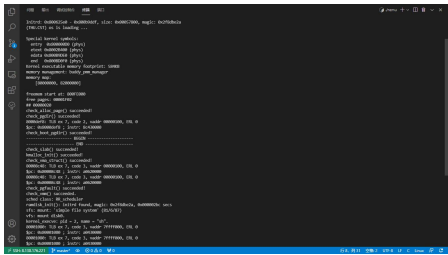
2. 由于 NEMU 模拟器与龙芯官方提供的 SOC 在地址空间分配上，存在部分差异，在启动操作系统时，常常遇到报错，针对这样的问题，我们采取的策略是使用 RAM 去填充那些在 SOC 中分配，而未在 NEMU 中分配的地址。最终解决了该问题。

3. 在使用 NEMU 模拟器启动 Ucore 时，在第一次遇到 TLB miss 异常时存在反复循环的问题，通过对 NEMU 模拟器源码的检查，我们发现在 NEMU 模拟器中，并没有按照标准 MIPS 指令集体系结构实现 CP0 的 Statue 寄存器 BEV 位控制异常处理的入口地址切换的功能，在修复之后，便可以通过 NEMU 模拟器启动 Ucore。

通过使用 NEMU 模拟器，方便我们对修改的操作系统源码进行快速的验证，提高了调试效率。也为了后续搭建 Verilator+NEMU 的差分测试框架提供了必要的支持。



(a) NEMU 运行 PMON



(b) NEMU 运行 Ucore

图 4.4: NEMU 适配 PMON 和 Ucore

4.3.2 Soc 仿真环境的搭建

为了搭建 Soc 仿真环境，我们对 SOC 做了一定的化简，只保留了串口与 RAM，并参考了 Rokit-Chip 仓库下 AXICrossbar 以及 AXIRAM 内容，使用 Scala 生成了 SocLite

仿真环境，只要添加 CPU 代码至生成的 SOCLite 框架下，即可在 Verilator 环境中完成对 CPU 的仿真。

为了实现在仿真环境下模拟串口的功能，我们在 SystemVerilog 代码中使用了 DPI-C 接口与 C/Cpp 代码进行交互，将操作系统往串口地址 1fe40000 写入的数据获取并打印出来，这样可以在仿真环境下模拟 SOC 中串口打印功能。

操作系统级的仿真往往时间较长，以 Ucore 为例，在我们搭建的差分测试框架下，从开始仿真直到进入命令行总计需要约 40 亿时钟周期，大约需要 4-5 个小时，如果在仿真后期的过程中遇到问题，如果没有波形状态保存功能，往往需要重跑一遍才能获取错误信息的波形。因此针对该问题，我们使用了 verilator 中的 `-savable` 选项，在仿真时，每隔 1000 万个时钟周期保存当前电路的状态，每 5 亿个时钟周期保存当前状态至文件夹。当遇到错误后，只要找到出错时刻之前的状态保存点，打开波形抓取并重新仿真，即可较回溯到当前出错点。这样的设计可以让我们快速获取出错处的波形信息，提高了 debug 的效率。

### 4.3.3 差分测试框架的设计

在差分测试框架搭建上，我们参考了一生一芯 NutShell 项目中构建差分测试的程序框架，建立了一套 MIPS32 指令集体系结构的差分测试框架，**将每一次 CPU 在 WB 级的提交信息，以及每一次 Store 指令的写数据以及写地址信息，提交给差分测试框架，与 NEMU 的执行信息进行比较**，如果相同，则继续执行下一周期 Verilator 仿真，如果不同，则报错出错误类型并退出。

使用差分测试框架帮助我们快速、准确的定位到许多在启动操作系统时遇到的隐蔽的 Bug，在操作系统移植的过程中，我们主要遇到了以下几个典型的问题。

1. Icache 取指异常的问题：在搭建 Verilator 仿真环境之前，我们已经成功启动了 PMON 并启动 Ucore 至初始化文件系统完成，此后 Ucore 报出 TLB miss 异常，通过差分测试框架的对比机制，我们快速定位到，在进程切换的时候，出现了 ICache 取指令出错的问题，之后我们并查阅相关资料并通过波形回溯检查，发现我们在 Icache 在取指令时，并没有通过 CP0Config0 寄存器 K0 域区分访存类型。
2. 寄存器未实现：在解决上述问题之后，在启动 Ucore 之后，我们成功进入了命令行，但无法输入任何指令，且在约半分钟，会报出 TLB miss 并退至内核的 Debug 模式，该问题最初耗费了大量时间 Debug，但也毫无头绪，最后在差分测试框架的帮助下，**等待了约 5 个小时，在 CPU 执行了约 40 亿个时钟周期之后**，框架在一条 MFC0 Status 寄存器报错，检查后发现我们没有实现 CP0 寄存器的 UM 位（指示内核态与用户态），在实现之后，Ucore 成功进入命令行，并可以接受串口的输入信息。

图 4.5 中的字母之所以会被打印两次，是因为 Verilator 和 NEMU 各会执行相应的程序并通过串口进行输出，一致则说明其行为正确。

在启动 Linux 过程中，存在部分外设无法模拟，因此我们只是通过差分测试框架进行部分测试，这也帮我们解决了数个由于添加指令引入的新问题。

## 4.4 SpyGlass

SpyGlass 平台能够高效准确的解决 SOC 中非常复杂的问题，在我们 CPU 的设计中，有时候 Vivado 提示的组合环路和多驱动可能并不准确。而通过 SpyGlass，我们可以非常

```

((THRU),.CST)) ooss iiss lloaanddiimgg .....

$Speeciaall kkeerrneell $tymmbbolls::
  eennttry 00xx0000000000000000 ((pphhyss
start to trace wave !!
now have executed 100000 instructions
})
  eetteextt 00xx00002200440000 ((pphhyss))
  eeddaattta 00xx0000000000000000 ((pphhyss))
  eemmd 00xx0000000000000000 ((pphhyss))
$Kkeerrneell eeexeeccutttaabllle mmeemmoorrry ffoocottprriinnatt:: 5588440008
mmeemmoorrry mmaannaaoggeemeeentt:: bbaudddy_ppmm mmaannaaoggeerr
mmeemmoorrry mmaapp::
  [[0000000000000000, 0022000000000000]]
now have executed 200000 instructions
timer int happen!
timer int happen!
now have executed 300000 instructions

```

(a) PMON 差分测试

```

PPPMONMZ2000000 PPIIP55 IInittiallizzimgg.. 55ttaamdddy.....
IIPRRORRRPCC==0000000000000000 CCOMNFIIG==0000000000000000
PPMRID=0000000000422200
IInitt ccaachheess..... ddo noottimgg.....
ppoddsomll ccaachheess ffoounndd
  b/c0000: m/c0 $0, $0, 0
  b/c0000: m/c0 $0, $0, 0
IInitt ccaachheess ddoonee,, cffgg == 0000000000000000

Ccoopyy PPPMON ttoo eeexeeccuttee lloccattioom.....
  sstarrtt == 00xx0077900000000000
  ss00 == 00xx3300000000000000
  _eeddaattta == 00xx0077904400000000
  _eemmd == 00xx0077904400000000
now have executed 100000 instructions
now have executed 200000 instructions
now have executed 300000 instructions
now have executed 400000 instructions
now have executed 500000 instructions
now have executed 600000 instructions
now have executed 700000 instructions
now have executed 800000 instructions

```

(b) Ucore 差分测试

图 4.5: PMON 和 Ucore 的差分测试

清晰的看到当前设计中存在的组合环路、多驱动以及位宽不匹配等影响仿真和综合的问题，提高了开发效率。

## 第五部分 附录

### 5.1 参考资料

MIPS® Architecture For Programmers Volume I-A: Introduction to the MIPS32 Architecture.rev3.02

MIPS® Architecture For Programmers Volume II-A: The MIPS32® Instruction Set

MIPS® Architecture For Programmers Volume III: The MIPS32® and microMIPS32™ Privileged Resource Architecture

Xilinx IP:: AXI Crossbar (2.1) LogiCORE IP Product Guide

### 5.2 参考仓库

#### 5.2.1 17 年参赛作品

清华大学 NaiveMIPS <https://github.com/z4yx/NaiveMIPS-HDL>

#### 5.2.2 19 年参赛作品

清华大学 NonTrivialMIPS <https://github.com/trivialmips/nontrivial-mips>

中国科学院大学 2 队 <https://github.com/nscscc2019ucas/nscscc2019ucas>

#### 5.2.3 20 年参赛作品

北京邮电大学 EasterMIPS <https://github.com/easter-mips/nscscc2020>

#### 5.2.4 其他

南京大学 NEMU-MIPS32 <https://github.com/nju-mips/nemu-mips32>

复旦大学 FDU CSLG <https://github.com/FDU CSLG/ICS-2021Spring-FDU>

一生一芯 NutShell <https://github.com/OSCPU/NutShell>

## 第六部分 致谢

衷心感谢王党辉，王继禾，安建峰，张萌四位老师的辛勤指导，感谢我们的学长学姐蒯嘉伟，谭昕，邱风硕，杨雨豪，陈昊琦，王珏在项目过程中提供的大力帮助。

感谢重庆大学陈决宇，清华大学陈晟祺，北邮谢云龙，NSCSCC 群友给与我们调试过程中的建议与帮助。

感谢西北工业大学一队和三队队员和康继昌智能系统班老师和同学给予我们的激励与帮助。