

第六届“龙芯杯”全国大学生计算机系统能力培养大赛

NINJA CPU

项目设计报告



西北工业大学一队

陈思远

3197571813@mail.nwpu.edu.cn

何帅杰

shuaijie_he@mail.nwpu.edu.cn

王健

jian.wang@mail.nwpu.edu.cn

夏康翔

xkx@mail.nwpu.edu.cn

2022 年 8 月

目录

第一部分 概述	3
1.1 项目简介	3
1.1.1 CPU 设计概览	3
1.1.2 指令集、CP0 和例外	4
第二部分 CPU 内核	5
2.1 流水线结构	5
2.1.1 总体架构	5
2.1.2 取指 1 (PREIF)	5
2.1.3 取指 2 (IF)	6
2.1.4 译码 (ID)	6
2.1.5 执行 (EXE)	6
2.1.6 访存 (M1、MEM)	6
2.1.7 写回 (WB)	6
2.2 互锁机制	6
2.3 主频优化	7
2.3.1 平衡各流水级延迟	7
2.3.2 切断较长的关键路径	7
2.3.3 设置信号的 max-fanout 属性	8
2.4 外部接口	8
第三部分 Cache 与 BPU 设计	9
3.1 Cache	9
3.1.1 硬件资源选用	9
3.1.2 ICache	9
3.1.3 DCache	9
3.2 BPU	10
第四部分 开发支持工具	12
4.1 Verilator 仿真加速	12
第五部分 参考文献	13
5.1 参考资料	13
5.2 参考仓库	13
5.2.1 19 年参赛作品	13

目录	2
----	---

5.2.2 20 年参赛作品	13
5.2.3 21 年参赛作品	13

第六部分 附录	14
----------------	-----------

6.1 指令	14
6.2 协处理器 CP0	14
6.3 中断与异常	14
6.3.1 中断	14
6.3.2 异常	15

第一部分 概述

1.1 项目简介

NINJA CPU 是基于 MIPS 32 Rev 1 指令集架构的处理器, 采用静态单发射七级流水线架构。其包含指令和数据缓存, 频率达到 150MHz, 可正常运行功能测试、记忆游戏、性能测试以及系统测试, 性能分 79.607 分。

序号	测试程序	myCPU	gs132	T_{gs132}/T_{mycpu}
		上板计时(16进制)	上板(16进制)	
		数码管显示	数码管显示	
cpu_clk : sys_clk		150MHz : 100MHz	50MHz : 100MHz	-
1	bitcount	3b268	13CF7FA	85.73905399
2	bubble_sort	172fc6	7BDD47E	85.47302176
3	coremark	462a76	10CE6772	61.31762508
4	crc32	21a5b4	AA1AA5C	80.88813246
5	dhystone	67cb2	1FC00D8	78.30987585
6	quick_sort	1a9d4d	719615A	68.28545612
7	select_sort	14bc0e	6E0009A	84.88243398
8	sha	15a43f	74B8B20	86.29420371
9	stream_copy	1801a	853B00	88.79682701
10	stringsearch	ff4d0	50A1BCC	80.85262099

性能分79.607

图 1.1: 性能测试分数

1.1.1 CPU 设计概览

接口规范方面, CPU 对外通过 3 个接口进行通信, 分别是 Uncache 接口、Cache 的指令和数据接口, 均采用 AXI3 规范。

缓存方面, CPU 使用 2 路组相联 8KB ICache 和 2 路组相联 8KB DCache, 其中 DCache 加入 32 项写缓存。

同时, NINJA CPU 利用优化的两位饱和计数器进行分支预测。在性能测试中, 分支预测整体准确率达到 88%。

由于初赛并不要求启动操作系统, 因此 NINJA CPU 并未实现完整的 TLB 功能, 仅仅实现了部分 TLB 指令, 虚实地址转换全部采用线性映射。

1.1.2 指令集、CP0 和例外

CPU 在大赛要求的 57 条指令基础之上，又增加了部分指令，为启动操作系统做准备。同时我们也实现了部分 CP0 寄存器和中断、例外。全部指令、CP0 寄存器和例外列表参见报告末尾附录部分。

2.1.3 取指 2 (IF)

IF 阶段, 如果 ICache 命中, 则从 Data Ram 中读回指令; 否则通过 AXI 向 RAM 发起一次读请求, 流水线的 IF 与 PREIF 阶段阻塞, 等待正确指令返回。

2.1.4 译码 (ID)

ID 级主要对指令进行译码, 生成一系列的控制信号 (包括控制 ALU、Memory、RegFile 的信号) 以及读取寄存器堆。译码时, 指令各段先用多个 5-32 decoder 和 6-64 decoder 进行转化, 然后使用并行的 assign 语句对指令类型作判断, 避免使用过于复杂混乱的 always-case 语法。

ID 流水级还会对数据冒险进行检测和处理。这样保证了送往执行级的数据都是正确的数据。

2.1.5 执行 (EXE)

EXE 级包含的模块主要为 ALU 和分支预测校正模块。其中, ALU 的乘除法均调用 Vivado IP 核, 除法约 37 周期完成, 乘法 3 周期完成。EXE 阶段也会根据从 ID 阶段传来的跳转信息, 对分支预测进行校准, 根据指令的实际跳转结果来更新 nextpc。另外, 在这一阶段还会对访存数据进行初步处理, 如 store 指令用到的 wdata。

2.1.6 访存 (M1、MEM)

访存的逻辑与取指类似。M1 级向 DCache 发起请求同样将访存地址的 Index 段分别送入 Tag ram 和 Data ram 读取 Tag 域和 Data 域。在 MEM 级则获取 Dcache 的数据, 并根据访存类型 (lw/lh/lb 等), 对数据做相应处理, 以得到正确的访存结果。

另外, M1 级包含 CP0, 在这一级会处理例外。

2.1.7 写回 (WB)

写回阶段将得到写回结果, 并将数据发送给寄存器。

2.2 互锁机制

流水线控制最容易想到的方式就是采用“上帝视角”, 即用一个顶层的 Control 模块, 根据 ID 阶段的译码结果、Cache 控制信号等进行流水线控制。这种想法比较直截了当, 可以对每一级流水的每一个信号精确赋值。但这会导致流水控制模块过于庞大冗杂, 而且各级流水之间无法进行通信, 分享阻塞情况。这种设计往往会导致更多的流水线阻塞, 影响 CPU 性能。

我们借鉴了《CPU 设计实战》一书中的互锁机制进行流水线控制。互锁机制可以参考下面的图片 2.2 (根据《CPU 设计实战》图画出), 每一级流水中都有 valid, allowin, ready_go 三组信号。当该流水级的指令有效 (pipe_valid = 1), ready_go 为高电平, 且下一级流水的 allowin 为 1 的时候, 该级流水就可以流动到下一级。只要有一个条件不满足, 则这两级流水停滞。另外, 当该流水级的 valid = 1'b0 时, 表明当前流水级并不具备一条有效的指令。

此时，该流水的 allowin 信号置为 1'b1; 而即使 ready_go = 1'b1, 由于 valid 信号为 1'b0, 依然不能流动到下一级流水中，这也是互锁机制的一个特色所在。

这样，我们在每一级都可以设置流水级流动的条件，虽然控制信号会比较分散，但可以相互通信，控制也更加稳定可靠。

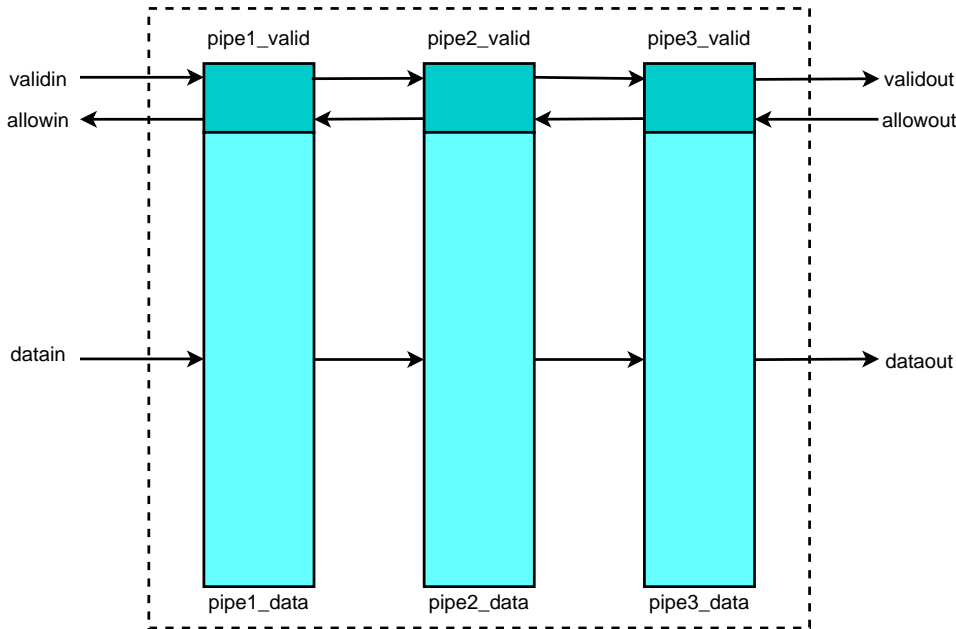


图 2.2: 互锁机制示意图

2.3 主频优化

在主频优化上，我们主要做了以下工作：

1. 平衡各流水级延迟。
2. 切断较长的组合逻辑延迟。
3. 设置信号的 max-fanout 属性。

2.3.1 平衡各流水级延迟

例如，设计之初，NINJA CPU 的 EXE 阶段承担了较多的计算工作，导致 EXE 的 logic delay 一直较大。我们将一些多路选择器（比如，寄存器值和立即数的片选选择器）前推到 ID 阶段，可以减少 EXE 阶段的延迟。

2.3.2 切断较长的关键路径

如果 CPU 中的信号在不同流水级中以直连的方式传递，则会造成较大的 net delay。据此，我们考虑在较长的关键路径中加入更多的寄存器锁存，以切断路径，减少 net delay。当然，这可能会导致更多的阻塞，存在一定的 trade off。

2.3.3 设置信号的 max-fanout 属性

由于 FPGA 板的资源分布限制，主频到 140MHz 向上就很难提升了，因为即使加入较多的寄存器，由于 Block RAM 引起的 net delay 却是难以避免的。我们通过在代码中设置信号属性，将对应信号的 max-fanout 属性通过调参设置成一个最优值，在实际设计中当该信号的 fanout 超过这个值时，综合器会自动对该信号采用优化手段，从而减少了信号的 fanout，net delay 也随之减小。但值得注意的是，Vivado 对与此类优化似乎具有某种随机性。

2.4 外部接口

CPU 和外部模块（包括 AXI RAM, Confreg 等）交互如下图2.3 所示。

考虑到减少模块与模块之间的耦合度，我们没有将 AXI 信号处理部分和 Cache 置于一处，而是单独设计了 AXI_Interface 模块，专门处理 AXI 交互信号。AXI 模块的信号仲裁方面，我们调用 Xilinx 的 AXI Crossbar IP 进行信号的仲裁，因为该 IP 的仲裁更加高效，阻塞更少。其优先级为：Uncache > DCache > ICache。仲裁后，CPU 顶层模块就只对外暴露一个 AXI 3 Master 接口。

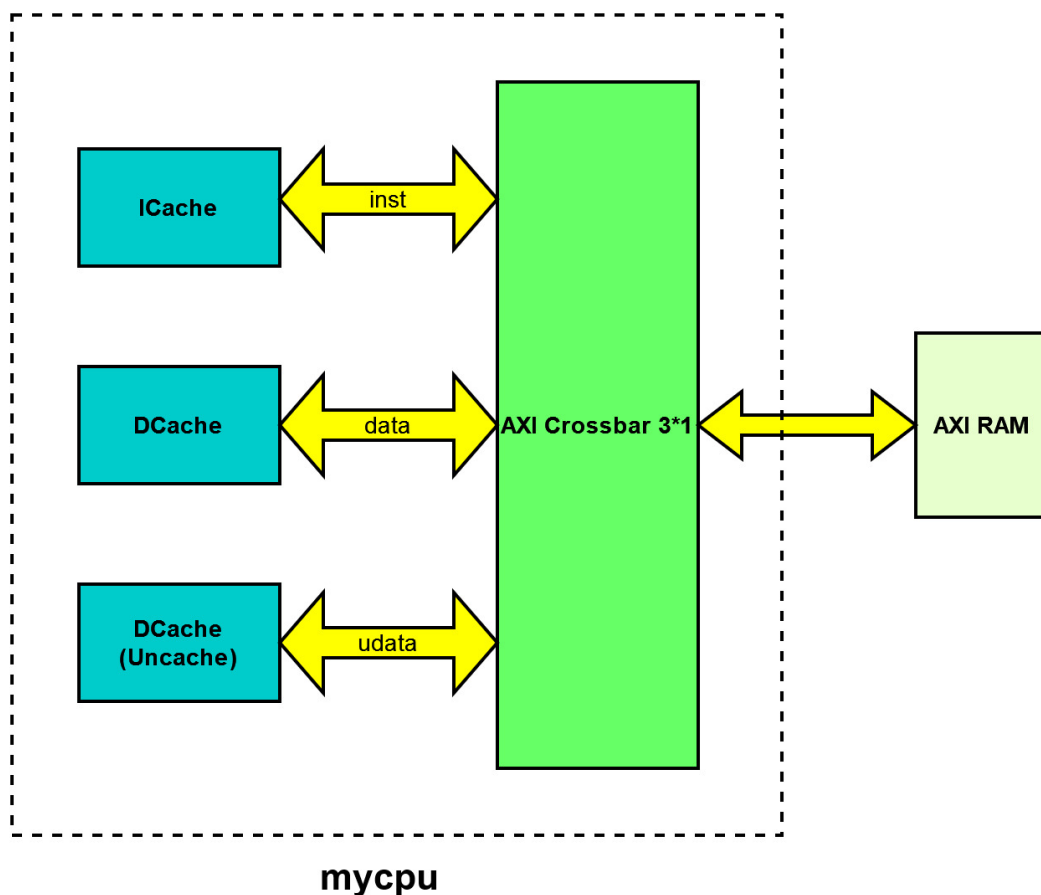


图 2.3: cpu 外部接口设计

第三部分 Cache 与 BPU 设计

3.1 Cache

3.1.1 硬件资源选用

ICache 和 DCache 在硬件资源的选用上, 采取了混合 RAM 的方式: Tag RAM, Valid bit, Dirty bit 均采用 LUT 进行存储, 而 Data RAM 则使用 Block RAM。原因是, 我们在查阅资料后发现, 对于 Block RAM 而言, 其 logic delay 总体良好, 但 net delay 却较大, 这和资源分布情况相关; 对于 LUT 而言, 当规模较小时, logic delay 和 net delay 都比较小, 但随着规模增大, 两种延迟都会迅速增加。因此, 我们采用了上述资源选用方式。

在资源调用方式上, 我们借鉴了第五届龙芯杯西北工业大学 2 队的作品, 使用 Xilinx 提供的一种宏 (Xilinx XPM) 进行资源调用, 这避免了反复配置 IP 核的麻烦。

3.1.2 ICache

ICache 的设计相对简单, 为两路组相连 8KB, 具备两级流水。在主频还较低的时候, 我们考虑将 ICache 设计为 4 路, 但后来发现, Cache 路数的增加却阻碍了主频的优化, 进而阻碍了性能的提升。故综合考虑之后, 我们选择了两路组相连的 Cache。

两级流水具体如下: 在第一个流水段发出请求, 从 Tag RAM 读出 tag 之后, 就可以比较知道是否命中; 但命中的信息以及读出的 tag 和 valid bit 都会锁存下来, 锁存之后的信号用来控制 Cache 状态机。这样避免了较长的组合逻辑。若没有其他阻塞, PREIF 级的流水能够顺利流动到 IF 级, 等待数据返回。

第二个流水段则可以得到 Data RAM 的数据。此时, 若 Cache 命中, 则流水线不阻塞, 得到指令送往 ID; 否则流水线 PREIF 级和 IF 级阻塞, Cache 向 AXI 发起指令请求, 待 AXI 返回的数据重新装填 Cache 后, 流水线方能继续流动。

在替换策略上, 我们移植了第三届龙芯杯清华大学“编程是一件很危险的事情”队的 PLRU 模块, 其具备两路和四路的替换方式, 这对我们初期调整 Cache 参数提供了方便。

3.1.3 DCache

DCache 的参数设置以及替换策略与 ICache 类似, 这里不赘述。由于 DCache 存在写后读冲突, 所以在设计时, store 需要两拍命中, 而 load 可以连续命中。这与 ICache 的连续命中有所不同。

访存请求既有 Cache 方式, 也有 Uncache 方式。这里我们尽可能的复用数据通路, 降低设计难度与资源损耗。

另外, 在优化访存方面, 主要采取以下两点措施:

1. 加入写缓冲。在性能测试中，我们发现 stringsearch 和 dhrystone 的得分相对较低，其原因就是这两个程序有大量的 uncache store 请求。因此，我们引入写缓冲，其实质是一个 32 项的 FIFO 队列。所有的 uncache store 请求都被压入 FIFO 队列中，然后按照顺序逐一发给 AXI。如果后续有 uncache load 请求，则要待 FIFO 队列清空之后，方能进行读取。
2. 孤立 store miss 加速。程序中，往往会出现一条孤立的 store 指令之后，紧跟着几条非访存请求。因此，我们引入 store record 模块，对 store 指令进行记录：如果该条 store 指令之前，所有的访存请求已经处理完毕，则不管是否命中，都可以放行。若尚有未处理完毕的访存请求，则需阻塞流水。

3.2 BPU

我们采用了优化后的基于两位饱和计数器的分支预测方法进行预测，将整体命中率提升至 88%。整体命中率达到预期，但存在部分程序中命中率极低的情况。最初我们采用了最基本的基于两位饱和计数器的分支预测方法，但此种预测方法命中率仅有 82% 不符合我们的预期，因此我们通过多种方法对此进行优化，具体优化方式如下：

1. 在 PRE_IF 阶段提前对部分指令译码，从而能够对 j、jal 指令的进行特殊处理。当遇到的此种跳转指令时我们的跳转结果不再根据两位饱和计数器进行预测，而是直接根据提前译码结果进行跳转。
2. 我们的两位饱和计数器采用经典的四状态跳转，基本跳转逻辑如图3.1。通过实验发现，初始状态对预测结果有较大影响，目前我们的初始状态为 Weakly_taken。

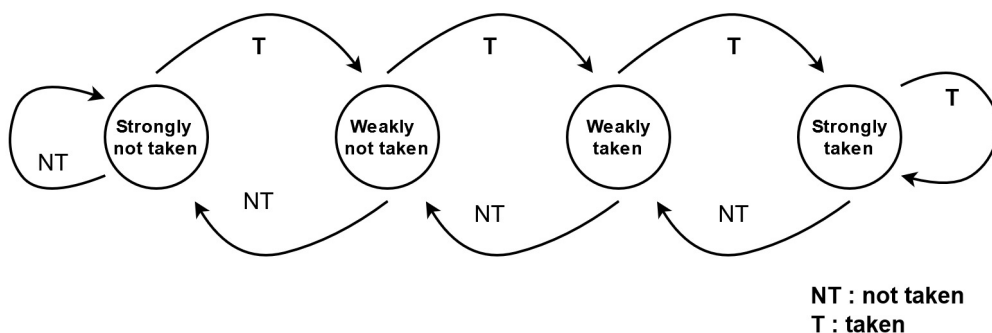


图 3.1: 基于两位饱和计数器的分支预测

此外，我们将各个程序中命中率的高低进行统计如表3.1，可见超过半成程序的命中率超过 90%，但 bubble_sort 的命中率极低，最终整体命中率符合预期。

表 3.1: BPU 命中率详情

type	hit	total	hit rate
bitcount	5001	6329	79.02%
bubble_sort	18364	26567	69.12%
coremark	70496	84365	83.56%
crc32	51466	54480	94.47%
dhrystone	14238	15133	94.09%
quick_sort	29016	35580	81.55%
select_sort	49656	52427	94.71%
sha	23772	24694	96.27%
stream_sopy	2335	2510	93.03%
string_search	27395	29347	93.35%
all	291739	331432	88.02%

第四部分 开发支持工具

在 RTL 测试的过程中,我们发现 Vivado 仿真耗时过长的问题。因此,我们将 NINJA CPU 代码和 Verilator 进行了相关适配,以大幅度减少仿真时间。

4.1 Verilator 仿真加速

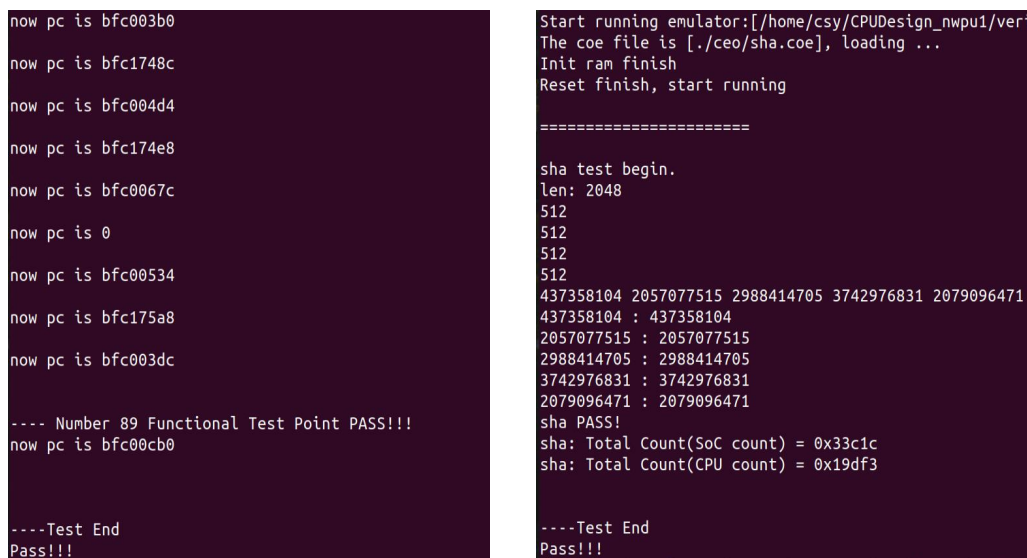
Verilator 是一个支持 Verilog/SystemVerilog 的周期精确 (cycle-accurate) 开源仿真器,同时号称是最快的 Verilog/SystemVerilog 模拟器 (the fastest Verilog/SystemVerilog simulator)。以性能测试 CoreMark 为例,在不开波形记录的情况下,Verilator 只需 3 秒即可完成仿真测试,而这一过程在 Vivado 中则需要十分钟左右。

在执行原理上 Verilator 将把 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,对我们的代码进行一定量的修改,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。

当然,我们搭建的 Verilator 框架不能完全模拟 AXI 随机延时,因此即使 Verilator 仿真通过,也依然可能存在问题;但这种方式大大提高了我们调试的效率,减少了等待仿真完成的时间。

下面为使用 Verilator 运行功能测试和性能测试的截图:



```
now pc is bfc003b0
now pc is bfc1748c
now pc is bfc004d4
now pc is bfc174e8
now pc is bfc0067c
now pc is 0
now pc is bfc00534
now pc is bfc175a8
now pc is bfc003dc

---- Number 89 Functional Test Point PASS!!!
now pc is bfc00cb0

----Test End
Pass!!!

Start running emulator:[/home/csy/CPUDesign_nwpu1/ver
The coe file is [./coe/sha.coe], loading ...
Init ram finish
Reset finish, start running

=====

sha test begin.
len: 2048
512
512
512
512
437358104 2057077515 2988414705 3742976831 2079096471
437358104 : 437358104
2057077515 : 2057077515
2988414705 : 2988414705
3742976831 : 3742976831
2079096471 : 2079096471
sha PASS!
sha: Total Count(SoC count) = 0x33c1c
sha: Total Count(CPU count) = 0x19df3

----Test End
Pass!!!
```

(a) 功能测试

(b) 性能测试

图 4.1: 功能和性能测试的 Verilator 仿真

第五部分 参考文献

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

汪文祥、邢金璋. CPU 设计实战. 机械工业出版社

姚永斌. 超标量处理器设计. 清华大学出版社

5.2 参考仓库

5.2.1 19 年参赛作品

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

5.2.2 20 年参赛作品

哈尔滨工业大学 UltraMIPS https://github.com/SocialistDalao/UltraMIPS_NSCSCC

5.2.3 21 年参赛作品

西北工业大学 GenshinCPU <https://github.com/gilgamsh/GenshinCPU>

第六部分 附录

6.1 指令

CPU 在大赛要求的 57 条指令基础之上，增加了部分指令为启动操作系统做准备。目前共实现了如下 65 条指令：

- **分支指令** BLTZ, BGEZ, BLTZAL, BGEZAL, BEQ, BNE, BLEZ, BGTZ, JR, JALR, J, JAL
- **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, MULT, MULTU, DIV, DIVU
- **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
- **无条件移动指令** LUI, MFHI, MFLO, MTHI, MTLO

6.2 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，实现的 CP0 寄存器如下表 6.1 所示：

6.3 中断与异常

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：

6.3.1 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。

表 6.1: CP0 寄存器

编号	名称	功能
0	Index	TLB 索引寄存器
1	Random	随机数
2	Entrylo0	TLB 表项物理偶页的内容
3	Entrylo1	TLB 表项物理奇页的内容
4	Context	指向内存中页表入口的指针
6	Wired	控制 TLB 中固定的页数
8	BadVAddr	最近发生的与地址相关的异常所对应的地址
9	Count	计数器
10	EntryHi	TLB 表项虚拟页的内容
11	Compare	计时中断控制器
12	Status	处理器状态及控制
13	Cause	上一次异常的原因
14	EPC	上一次异常发生的地址

6.3.2 异常

CPU 实现了 MIPS 规范的精确异常，在 M1 级统一提交异常。CPU 中实现的异常如下表6.2 所示：

表 6.2: 主要支持的异常

ExCode	异常简称	异常说明
0x00	Int	中断
0x04	AdEL	地址 Load 异常
0x05	AdES	地址 Store 异常
0x08	Sys	系统调用
0x09	Bp	断点
0x0a	RI	保留指令
0x0c	Ov	算术溢出