RISC-V 处理器设计报告

计 82　蔡思捷　2018011310
计 82　郑凯文　2018011314
计 82　张廷基　2018013355

2020 年 12 月 11 日

目录

1 实验概述

1.1 实验目标

在 THINPAD 硬件平台上，采用多周期或流水线架构实现支持 32 位 RISC-V 指令的 CPU，使用 BaseRAM/ExtRAM 作为存储，并通过串口进行输入/输出。基本任务为实现 19+3 条指令并运行基础版本的监控程序，拓展任务包括实现中断/异常、U 态地址映射/页表/TLB、分支预测、L-Cache/D-Cache、外设接口如 VGA/USB/Flash 等。

1.2 实验结果

我们采用 5 级流水线架构，实现了支持简易异常、TLB 的 CPU，可运行页表版本的监控程序。此外，我们拓展实现了分支预测和 VGA，并将主频提至 60M，使得 CPU 在功能齐全的条件下具有优异的性能。

1.3 组内分工

- 蔡思捷：IF/MEM 阶段与 SRAM/串口的交互，页表/TLB，VGA
- 郑凯文：数据前传，控制器，寄存器，分支预测，异常
- 张廷基：指令译码，ALU，VGA 动画的汇编代码编写

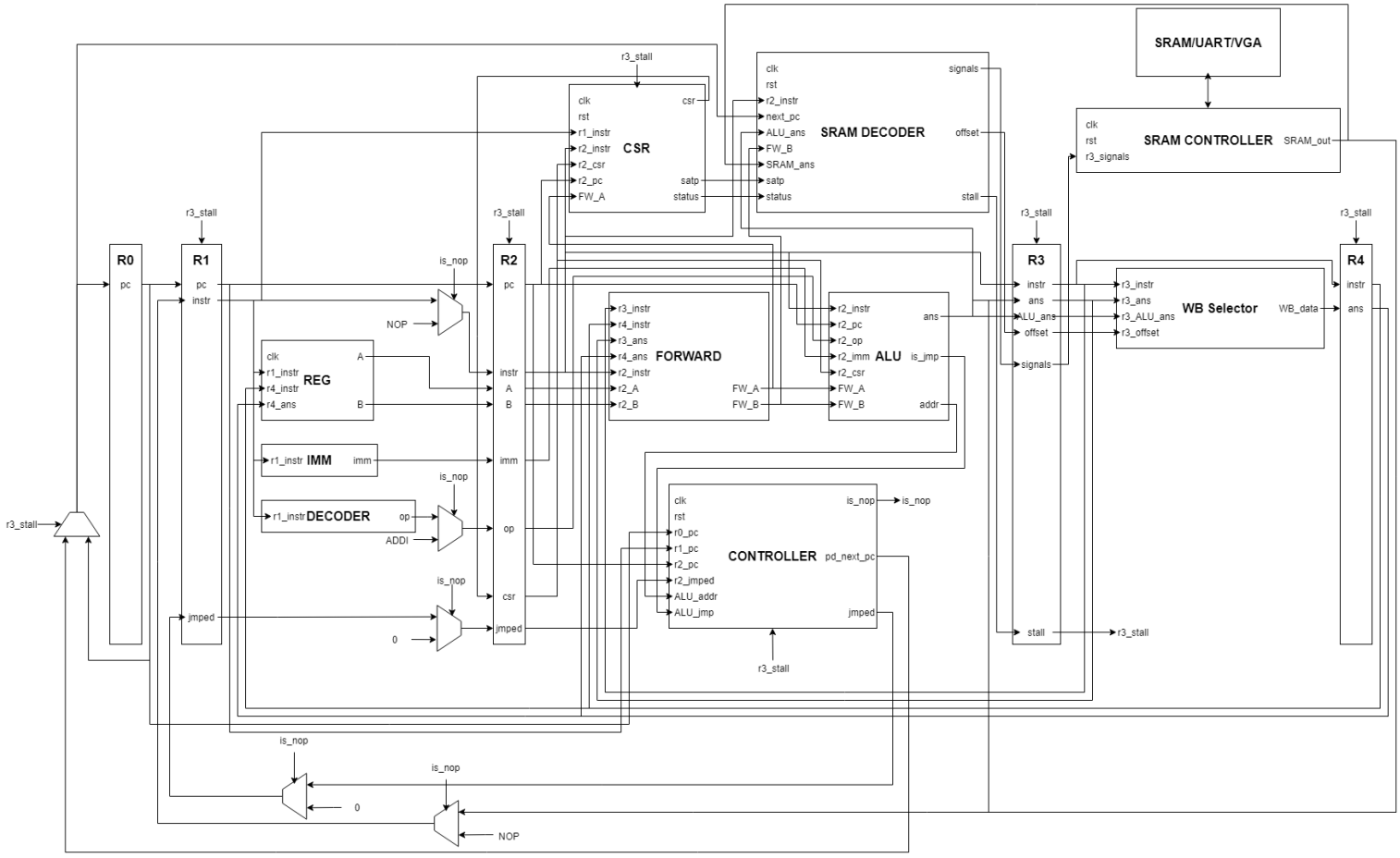

图 1: 数据通路与模块示意图

2 数据通路与模块

图1展示了 CPU 设计的大部分模块与线路。模块大致分为:

- REG: 寄存器
- IMM: 立即数生成器
- DECODER: 指令译码器
- FORWARD: 数据前传
- CONTROLLER: 控制器，内含分支预测

- ALU：算术逻辑运算
- CSR：异常处理模块，内含异常相关寄存器
- SRAM DECODER：在 EXE 阶段生成 SRAM 相关控制信号
- SRAM CONTROLLER：将控制信号翻译为使能/片选等信号，直接与 SRAM/串口/VGA 交互，可通过时钟的上下边沿完成 1 周期读写 SRAM
- WB Selector：综合 ALU 的结果和 SRAM 的输出，决定写回的内容

3 实现细节

3.1 流水线结构

我们实现的是标准的 5 级流水线，由 IF、ID、EXE、MEM、WB 五个阶段组成。

3.1.1 IF

IF 阶段进行取指操作。SRAM DECODER 接收 next_pc 作为输入信号，在没有遇到访存指令时，将产生取指的控制信号，控制 SRAM CONTROLLER 完成取指。取得的指令被存入 r1 寄存器中。

3.1.2 ID

ID 阶段进行译码。具体而言，指令被 DECODER 模块解析为 opcode，被 IMM 模块解析为立即数，并通过 REG 模块得到 A、B 寄存器的值，通过 CSR 模块得到 csr 寄存器的值。这些数据都被存入 r2 寄存器中。

3.1.3 EXE

EXE 阶段进行 ALU 运算和 SRAM 的译码。ALU 接收前传后的 A、B，以及 csr、立即数、opcode，输出运算后的结果，并产生是否跳转、跳转目标地址的信号。SRAM DECODER 对指令进行解析，若存在访存行为，则 ALU 运算结果为目标地址，输出访存的控制信号并转换内部状态；否则，输出取指的控制信号。ALU 运算结果以及 SRAM DECODER 产生的控制信号均被写入 r3 寄存器中。

3.1.4 MEM

SRAM CONTROLLER 接受 r3 中的控制信号，进行访存或取指的操作。若进行取指，则 SRAM 的输出被传回 r1；若进行访存，则流水线被暂停 1 周期（为了高主频下串口的稳定，读/写串口将暂停 6 个周期），等 SRAM 读取或写入完毕后，SRAM 的结果被写回 r3。r3 中准备好的结果（ALU 或 SRAM 输出）经过 WB Selector 进行选择，并特殊处理 lb 的字节偏移后写入 r4。

3.1.5 WB

WB 阶段进行寄存器的写回。指令及其执行的结果被回传入 REG 模块，REG 模块在判断需要写回后，将结果写入对应寄存器。

3.2 冲突处理

3.2.1 数据冲突

我们通过数据旁路解决寄存器的数据冲突。对于如下两种情况

IF	ID	EXE	MEM	WB
	read t0	write t0		
	read t0		write t0	

由于寄存器尚未写回，在 ID 阶段无法获取到 t0 更新后的值。而在下一个周期

IF	ID	EXE	MEM	WB
		read t0	write t0	
		read t0		write t0

应写回的内容已经过 EXE 阶段被计算完毕，因此将 r3 寄存器与 r4 寄存器的结果通过旁路前传至 EXE 阶段可解决此类冲突。

3.2.2 结构冲突

对于寄存器结构冲突，即如下情况

我们通过在 REG 模块里设置独立的写端口来解决。若读取的寄存器正是要写的寄存器，那么将读取的值置为要写的值。

IF	ID	EXE	MEM	WB
	read t0			write t0

对于 SRAM 相关的结构冲突,简单地将流水线暂停 1 周期,而没有区分使用的是 BaseRAM、ExtRAM 或 VGA。在 EXE 阶段 SRAM 译码后，产生暂停信号 r3_stall 来标识是否暂停。在 1 周期的访存结束后，流水线恢复，SRAM 回到取指状态。

3.2.3 控制冲突

CONTROLLER 模块负责分支跳转相关的逻辑。其输出的控制信号 is_nop 标识着流水线的冲刷。流水线不因分支跳转而暂停，当分支预测错误时，is_nop 为真，IF 和 ID 阶段的指令被取消，同时从 r2 的下一条指令处重新开始取指执行。

3.3 异常

实现了如下异常:

表 1: 支持的异常

异常码	异常名称	描述
3	Breakpoint	ebreak 指令触发
8	Environment call from U-mode	ecall 指令触发

可以支持异常版本的监控程序。对于 ecall 和 ebreak 指令，简单地作为跳转指令来执行，只是将 mepc 置为当前 pc，将 mcause 置为异常原因。mret 指令同样作为跳转指令，csrr 系列指令则放在 CSR 模块内部进行运算。在 CSR 模块内部维护一个状态代表当前是 U 态还是 M 态，并暴露给 SRAM DECODER 以支持 U 态地址映射。

异常模块只涉及了 ID 和 EXE 两个阶段，这并非标准的异常处理流程。实际上这是对 60M 频率的妥协，这样可以使用尽量少的硬件资源跑通监控程序。我们也尝试了在流水线的各个阶段收集共 8 种异常，在 MEM 阶段统一处理，但会导致 60M 下时序异常。

3.4 分支预测

我们采用 8 项全相连 BTB 存储 (pc, next_pc) 的二元组，使用 1 位的分支预测与 LRU 缓存替换策略。

对于某个 pc，若不在表内，则下个 pc 为 pc+4，否则查表获取下一跳地址。若查表命中，则置 jmped 为 1，否则为 0，其随着流水线寄存器行至 r2。在 ALU 阶段，根据 r2_jmped 与 ALU 得到的真实跳转判断是否发生分支预测错误，若错误则冲刷前两级流水线。

若查表命中且确实发生跳转，则将表项更新并移至第一位；若查表命中但未跳转，则从表中删除表项；若查表未命中且发生跳转，则将表项添加至第一位。

3.5 内存管理与串口

由于 FPGA 内部无法实现总线，因此我们的 CPU 无法真正实现总线使用权的仲裁和转移，而是在 EXE 阶段将所有的相关数据传入仲裁模块 SRAM DECODER，按一定的优先级顺序判断即将使用总线的数据流，将控制信号写入 r3 寄存器中。而将仲裁放在 EXE 阶段的理由是我们采取了单周期读写 SRAM，时序要求非常紧张，将控制信号提前解析可以大大改善这个问题，代价则是牺牲 EXE 阶段的时序裕量。正因为采取这样的策略，最终我们的 CPU 可以达到 60M 频率。

3.5.1 IF 取指

如果 EXE 阶段当前处理的指令不需要进行访存，那么 SRAM DECODER 会获取下一指令的 pc 值，根据地址映射关系使能相应的 SRAM，这样在下一时钟来临时把所有信号交给 SRAM CONTROLLER 模块，真正进行取指。

3.5.2 MEM 访存

如果 EXE 阶段当前处理的指令需要进行访存，那么 SRAM DECODER 会优先进行访存操作，也就是使用来自 EXE 阶段的地址和数据来判断总线信号的赋值；除了 SRAM 控制信号外，还会输出 stall 信号，表示当前正在进行访存，存在访存冲突，其他阶段的寄存器接收到此信号后需要锁住当前值，使流水线暂停；同时控制器内部会对是否已处理访存进行标记，如果已经处理过一次访存，那么接下来的 SRAM 指令应当为取指。

SRAM 的读写只需暂停 1 个周期，而串口的时钟周期为 11M，为了串口的稳定性需要暂停 6 个周期。

3.5.3 MMU+TLB

我们实现的 TLB 为一行 TLB，只缓存一个一级页表基地址和一个二级页表基地址。当开启页表并且状态为用户态时，SRAM DECODER 会在上述两个过程前自行安排地址翻译，操作步骤为：

1. 先比对虚地址的高位与已保存的页表索引是否相同。
2. 若一级索引不同，则先将地址设为 satp+VPN0 获取二级页表基地址；
3. 若一级索引相同，二级索引也相同，那么说明不需要额外访存，直接使用缓存的实地址；否则将地址设为二级页表基地址 +VPN1 获取实地址页的基地址；
4. 若已获取一级页表基地址，也将地址设为二级页表基地址 +VPN1 获取实地址页的基地址；
5. 最后加上页内偏移量获取实地址。

整个流程由状态机控制。

最终的地址映射如下：

表 2: 地址映射

态	虚拟地址	物理地址	目标
M	-	0x80000000-0x800FFFFF 0x80100000-0x803FFFFF 0x80400000-0x807EFFFF 0x807F0000-0x807FFFFF	SRAM
M	-	0x10000000	串口数据
M	-	0x10000005	串口信号
M	-	0x00000000-0x0007FFFF	VGA 显存（BRAM）
U	0x00000000-0x002FFFFF 0x7FC10000-0x7FFFFFFF 0x80000000-0x80000FFF 0x80100000-0x80100FFF	通过 MMU 确定	SRAM

3.6 VGA

VGA 视频输出拥有专属的 BRAM 作为显存，其控制信号来自 VGA 模块，每一时钟刻来临时，VGA 模块输出下一刻需要显示的位置的地址，BRAM 获取到该地值后在下一刻准备好 8 位 rgb 数据，最后 THINPAD 将对应行列和 rgb 数据输出到 VGA 外设。图片需要缩放至 800x600 大小并按照 R:G:B=3:3:2 的位数压缩。

为了优化 FPGA 内部布线，我们将获取 8 位 rbg 数据改为一次性从 BRAM 中读取 32 位数据，即 4 个像素点的信息，再根据最后两位偏移量决定输出值。这样一来，VGA 便可以独立于流水线进行工作。

为了改善 VGA 的可操作性，还将 BRAM 挂载到了从 0x0 开始、共 19 位的地址空间上，这样就可以利用普通的访存指令来修改希望显示的像素数据。

4 成果展示

4.1 功能

在 ThinPAD-Cloud 在线实验环境上，我们通过了 19+3 的指令测试。图2展示了页表版本的监控程序中运行用户程序输出 ASCII 字符的过程，这表明异常及 TLB 实现无误。

```
connecting to 166.111.226.111:40043...connected
MONITOR for RISC-V - initialized.
running in 32bit, xlen = 4
>> a
addr: 0x80100000
one instruction per line, empty line to end.
[0x80100000] .section .text
[0x80100000] .globl _start
[0x80100000] _start:
[0x80100000]     li      a0, 0x21
[0x80100004]     li      t4, 0x7f
[0x80100008] mys:
[0x80100008]     li      s0, 30
[0x8010000c]     ecall
[0x80100010]     addi    a0, a0, 1
[0x80100014]     bne     a0, t4, mys
[0x80100018]     li      a0, 0x21
[0x8010001c]     ret
[0x80100020]
>> g
addr: 0x0
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
elapsed time: 0.099s
```

图 2: 页表版本下 ecall 的调用

4.2 性能

性能测试结果如下：

可以看到，除访存冲突必须暂停 1 周期外，其余测试基本跑满 60M，这表明数据旁路、分支预测等很好地处理了相关冲突，几乎不牺牲流水线性能。

表 3: 性能测试结果

测试名称	描述	指令数	运行时间（s）	实际主频
1PTB	无数据冲突与结构冲突	320M	5.593	57.2M
2DCT	大量数据冲突	176M	3.076	57.2M
3CCT	大量控制冲突	256M	4.474	57.2M
4MDCT	大量访存相关数据冲突	192M	5.592	34.3M
CRYPTONIGHT	-	-	0.306	-

4.3 VGA

我们使用 VGA 进行动态图片的展示。展示的对象是图形学课上渲染出的 1920x1080 高分辨率图像，将其缩小到 1064x600 大小并进行 R:G:B=3:3:2 位的压缩。将得到的.bin 文件写入 ExtRAM，并运行如下的汇编代码：

```
li  t4，0
li  t5，264
li  a6，0
li  t6，1
L3:
li a2，0x80400000
li t2，0
li t3，480000
L1:
li  a3，0
li a4，800
L0:
beq a3，a4，L2
add a5，t4，a2
lw t1，0(a5)
sw t1，0(t2)
addi a2，a2，4
addi t2，t2，4
addi a3，a3，4
bne t2，t3，L0
```

```
beq a6, t6, L6
beq a6, x0, L4
L5:
beq t4, t5, L7
beq t4, x0, L8
bne t4, t5, L3
bne t4, x0, L3
ret
L2:
addi a2, a2, 264
j L1
L4:
addi t4, t4, 4
j L5
L6:
addi t4, t4, -4
j L5
L7:
li a6, 1
j L3
L8:
li a6, 0
j L3
```

这段代码不断从 ExtRAM 中读取图像 800x600 大小的一部分并写入 BRAM，效果为一张不断左右滑动的动态图片（图3）。动态效果见附件视频。

5 心得与收获

张廷基 入学以来就听闻贵系“奋战三星期，造台计算机”的口号，如今终于亲身经历，不断踩坑、不断尝试，和队友合作造成 CPU。在不停推翻之前代码、修改框架和思路的过程中，我们对多级流水线的各个步骤了解已经有了质的飞跃。过程中遇到各种问题，但正是不断遇到问题、讨论问题、解决问题的过程让队友之间的沟通、对 CPU 结构的理解变得越来越通透。尽管每个人都夜以继日的编译、调试，仍有一些遗憾之处；许多尝试最后

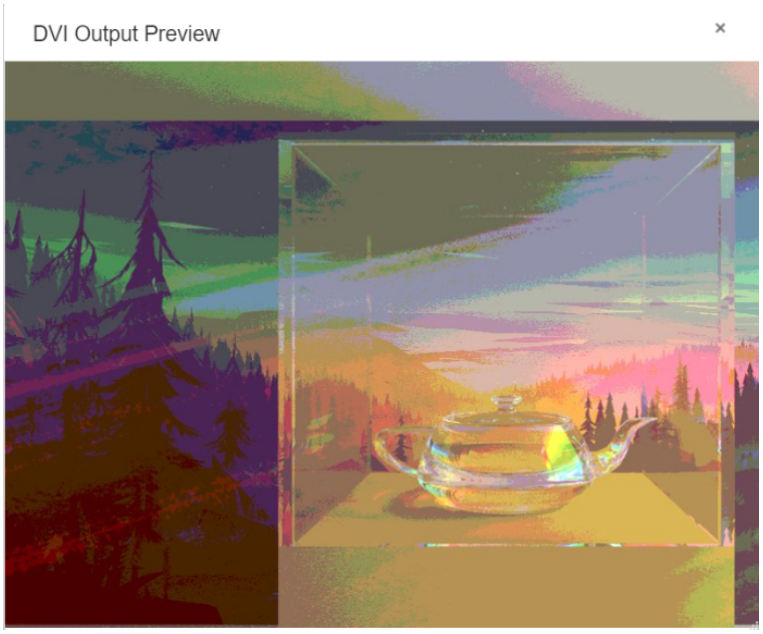

图 3: VGA 效果图

并未能成功体现在最终版本里。相信 Vivado 跑在电脑上，自己则盯着屏幕默默等待的日子会变成未来珍贵的回忆。

郑凯文 最想说的是感谢 carry 的队友，奋战三星期的历练在队友的加持下变得轻松了许多。造机开始时对流水线一无所知，但紧张的心情在两次讨论后化为乌有，蔡思捷同学耐心的讲解和清晰的思路让大家有了底气。在连续一周的提交、合并、修 bug 后，CPU 已然成型，我对流水线的结构、冲突的处理也了然于胸了。在做拓展功能时，三人分工有序，短短几天便做完了异常、分支预测、VGA 和页表，已然超出了开始时的预期太多，成功提频到 60M 更是一件惊喜的事。有些遗憾的是，我在最后几天尝试更改流水线结构添加更多异常，但复杂的布线和紧张的 EXE 阶段时序让我始终无法维持 60M 主频，最终我们选择了 60M 而非更多的异常和指令。回想当时一次次的等待，一次次祈祷 Vivado 综合出的 wns 是正数，虽然没有体现在结果上，但让我对流水线有了更深的思考，或许之后也会去研究一下双发射十级流水线之类的设计吧。

蔡思捷 详细地了解和掌握了简单流水线 cpu 的工作方式和设计方式，看到自己和队友们搭出来一个计算机的心脏还是非常激动的；这个大作业里最大的乐趣在于设计结构和解决冲突，通过不断的设计、放弃的循环，最终找到解决办法的感受真的非常好!