TSINGHUA UNIVERSITY

DESIGN DOCUMENT

A 5-stage 32-bit RISC-V Core

Author:
Benyuan Meng
Chenghua Liu
Haowei Wu

A course project submitted in the course of
Computer Organization and Design

in the

Department of Computer Science and Technology
Genshin Group

November 25, 2021

A 5-stage 32-bit RISC-V Core

by

Benyuan Meng
Chenghua Liu
Weihao Wu

A course project
submitted in the course of
Computer Organization and Design

at

Tsinghua University
November 25, 2021

Acknowledgements

We would like to thank our course teachers:Kang Chen, Youyou Lu, WeiDong Liu.We would also like to sincerely thank Yichuan Gao and other teaching assistants and classmates who have helped us.

Genshin Group
Tsinghua University
November 25, 2021

Contents

Chapter 1

Introduction

Genshin 是使用 Verilog 语言模块化设计的，基于 RISC-V RV32I 开放指令集的顺序单发射处理器实现，隶属于清华大学计算机系计算机组成原理课程项目。Genshin 基于 5 级流水线顺序设计。具有以下功能：

1. 支持来自 RV32I 的 19 条基本指令和来自 RV32B 的 3 条扩展指令（PCNT、SBSET、ANDN），能够正确运行基础版本的监控程序。
2. 支持开启或关闭可配置深度的跳转目标缓存 (BTB) 和返回地址栈 (RAS) 的分支预测 (可选 bimodal/gshare)。
3. 支持检测并处理异常，能够执行 CSRRW CSRRS CSRRC ECALL EBREAK MRET 6 条指令，运行中断和异常版本的监控程序。
4. 支持对用户态的内存空间做分页映射，运行页表版本的监控程序。实现了对 Sv32 页表格式的支持，对 page fault 的支持，以及一个简单的 TLB。
5. 实现数据指令混合 Cache。
6. 支持 VGA，通过汇编程序控制输出图像和常用字符。

基础版本的整体的微结构设计如下图所示：

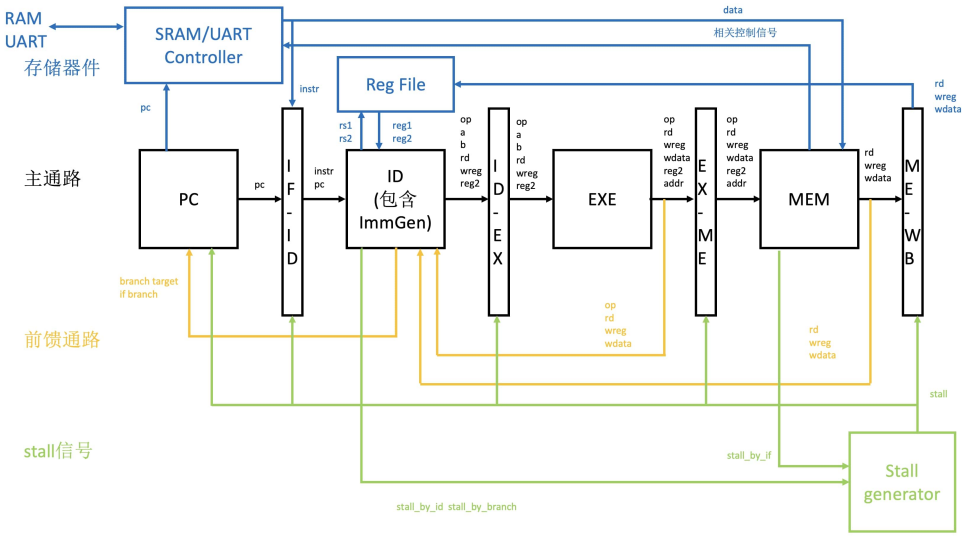


Figure 1.1: 基础版本 Genshin 整体微结构设计图

Chapter 2

Architecture Design

2.1 Architecture

我们先给出各文件所表示的模块和用途。

1. 顶层接线：thinpad_top
2. 存储相关元件
   (a) regfile：寄存器组
   (b) csr：特权寄存器组
   (c) sram_uart:SRAM 控制元件,实际上通过地址解析支持了串口、VGA、mtime(cmp)等，也支持了缓存和页表 (含 TLB)
3. 阶段寄存器
   (a) if_id
   (b) id_exe
   (c) exe_mem
   (d) mem_wb
4. 各阶段内的组合逻辑
   (a) id : ID 段的组合逻辑，完成译码、立即数生成、读寄存器、冲突处理等
   (b) exe: EXE 段的组合逻辑，相当于 ALU
   (c) exceptionHandler: 主体是组合逻辑，进行异常处理，包含了一小块时序逻辑用于存储当前运行态
   (d) memExpDetector: 组合逻辑，提前在 EXE 段判断地址异常
   (e) mem: MEM 段的组合逻辑，生成一些方便 SRAM 控制元件理解的信号
5. 其他
   (a) if_pc：时序逻辑，PC 的更新，包含了动态分支预测

(b) ctrl：组合逻辑，生成各种情况下的 stall 信号

(c) vga/vga_show：VGA 相关

(d) macro：宏定义

(e) exceptionRelatedRegs：本来用于存储运行态，不过最终没有使用

Cache+VGA 这两部分都连在 SRAM 控制模块上面，见图2.1。

- Cache 仅负责对 SRAM 的映射，优先检查缓存，若未命中再访存。
- VGA 的实现方式类似于串口，人为定义了一组特殊的地址，将 VGA 映射在该位置，使用对该地址的 store 指令时可以与 VGA 交互。

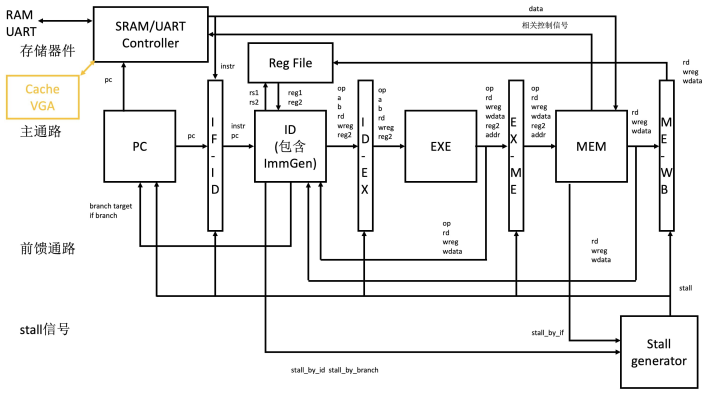


Figure 2.1: Genshin 微结构设计图：cache+vga

分支预测 分支预测主要添加在 IF 段和 ID 段，见图2.2 。

- IF 段是分支预测的主体部分，PC 旁边设置了一套完整的跳转表，可以根据当前 PC 以及历史信息，直接预测下一次 IF 的地址。
- ID 段负责判断预测是否准确，在基础版本中 ID 段就已经可以得到正确的跳转目标，现在只是将原本的反馈信号 branch_flag, branch_target 改为 pred_error 等。

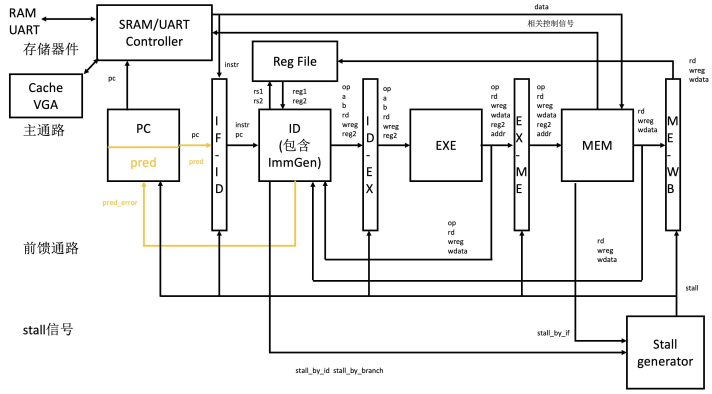


Figure 2.2: Genshin 微结构设计图: 分支预测

异常 异常处理单元位于 EXE 段，EXE 组合模块的旁边，两者平行地工作，见图2.3。

- CSR 位于寄存器组的旁边，与 ID 和 Handler 相连。
- EXE 的输入来自于前面的阶段寄存器，以及时钟相关寄存器，后者也在 SRAM 元件中进行映射。
- Handler 会对 CSR 进行写操作，并且在需要进行上下文切换时也会生成相关信号并通知前面的流水线阶段。

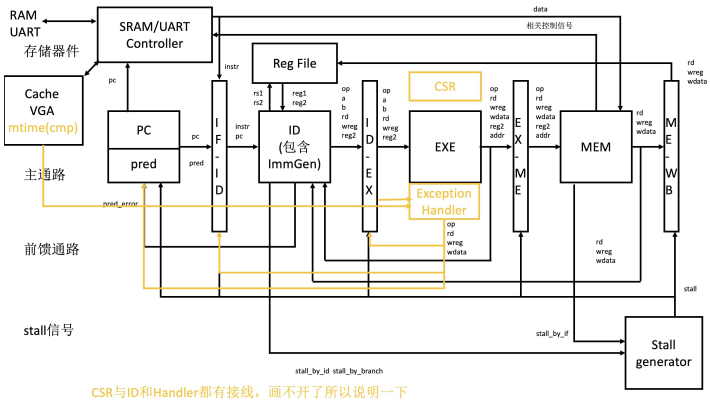


Figure 2.3: Genshin 微结构设计图：异常

页表 新增 CSR.satp ，见图2.4。

- 页表查询功能在 SRAM 中增加，并用类似 Cache 的方法完成了 TLB。
- SRAM 会提供 page fault 相关信号，这些信号或通过直连或随着流水线传递，最终到达 ExceptionHandler 。

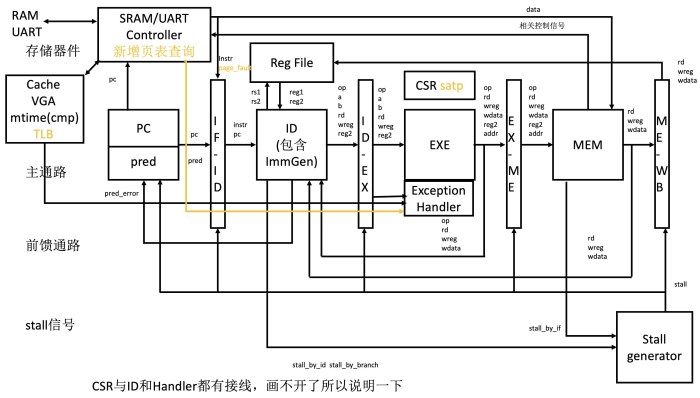


Figure 2.4: Genshin 微结构设计图：TLB

2.2 Conflict Handling

我们的架构主要遇到了如下冲突，并分别进行了处理：仅涉及寄存器的数据冲突和访存相关的数据冲突，因为一个周期内 SRAM 控制元件只能完成一次访存而带来的结构冲突、因为 SRAM 访存实际上需要多个周期而带来的结构冲突，跳转引起的控制冲突，以及复合冲突。

数据冲突 寄存器数据冲突是指 ID 段需要获取某个寄存器的数值时，之前修改过它的指令仍然存在于流水线的 EXE、MEM、WB 段，尚未完成写回，因此无法直接从寄存器组中获得正确数据。

对于存在于 EXE 段和 MEM 段的指令，我们将待写回的数据以及相关的控制信号接回 ID 段，由 ID 组合逻辑进行冲突处理。对于 WB 段的指令，我们对寄存器组进行修改，如果要读的寄存器正好是要写的，则直接把写信号接到读信号上，而不需要等待到下一个上升沿的写入。

与访存有关的数据冲突，是指紧邻的上一条指令是 load，且存在上面描述的写回、读取关系。在这种情况下，当前指令在 ID 段，上一条指令在 EXE 段，load 结果不可知，需要等到 MEM 段才能知道。因此，我们生成一条 stall 信号，通知 ID 段及之前的阶段暂停，ID 段向后输送一个气泡，后面的阶段继续执行。在下一个周期，MEM 段将拥有正确的待写回数据，可复用之前实现的数据冲突路线完成处理。

如果 load 指令并非紧邻，那么可以当成普通的寄存器数据冲突处理，因为发生冲突时 load 指令至少到了 MEM 段，已经拥有正确的待写回数据。

结构冲突 我们的架构只允许一个周期进行一次访存，因此如果当前周期 MEM 段需要进行访存，则 IF 段无法完成取指。针对这种情况，我们生成一条 stall 信号，通知 PC 暂停更新，IF 段向后输送一条信号，后续阶段继续执行。

其实一次访存需要超过一个周期，前面是为了表述方便才写成" 一个周期一次访存"。所以我们会在访存时生成一条 stall 信号，通知整条流水线暂停，等待访存完成。实现 Cache 后，该 stall 信号的持续时间可以缩短，进而提高性能。

控制冲突 控制冲突是指分支预测失败时的处理，注意基础版本其实也相当于做了分支预测，只不过每次都预测成不跳转，所以该冲突的处理在基础版本和含有动态分支预测的版本中基本一致。

- 若预测成功，则不需要任何额外处理。
- 若预测失败，我们是在 ID 段发现预测失败，此时 IF 段已经向后输送了一条错误指令。在这种情况下，我们生成一条 stall 信号，通知 IF-ID 阶段寄存器不要接受该指令，而是生成一个气泡作为替代；同时也将正确的跳转目标告知 PC。

复合冲突 我们发现，在我们的架构之下，如果同时发生结构冲突和控制冲突，则需要进行特殊处理。这是因为，控制冲突的信号由 ID 段产生，这些都是组合信号，并没有保

存下来。而发生结构冲突时，PC 保持不变，包含 ID 在内的其他阶段继续执行，这将导致 ID 段产生的控制冲突相关信号丢失。

因此，在同时发生这两种冲突的情况下，我们会根据控制冲突信号修改 PC 的值，而不是完全保持 PC 不变。

Chapter 3

Super Branch Prediction

在冯诺依曼体系结构中，指令供给的效率极大地影响了处理器的性能。Genshin 处理器核在取指单元中加入了硬件分支预测器，用于在转移指令的取指阶段预测出指令跳转的方向和目标地址，并从预测的目标地址处继续取指令执行，在一定程度上减轻指令流水线中转移指令引起的阻塞，从而提高处理器的性能。

3.1 Overview of Branch Prediction

严格意义上来说分支预测器应该包含三部分：方向预测、地址预测、错误处理。我们首先来看方向预测的几种经典方案。

静态分支预测 静态分支预测是说预测转移指令全都不发生跳转或是全部都发生跳转。在基础版本的 Genshin 中，我们的方案是每次都预测不跳转，如果在 ID 段发现发生跳转就说明预测错误，那就有一条指令本来不该读的，这条指令需要刷掉，我们通过设置传递 stall 信号来刷掉这条指令。

Bimodal 分支预测 研究发现大部分 branch 要么经常 taken, 要么经常 not taken。所以预测的时候历史情况是一个很好的参考。bimodel 通过两个 bit 记录历史的 branch 结果，注意这里两个 bit 是记录每条独立分支指令的结果，而非全局分支的结果。具体实现就是一个简单的 table，每个 entry 中为一个 2bit saturating counter，table 通过 PC 的部分低位 bit 来索引。分支预测的示意图和 saturating counter 的状态转移图见图3.1。在 S. McFarling 的论文中，Bimodal 的预测准确率可以到 93% 左右。在 Genshin 中，我们实现了这一方案，可以通过设置参数开启。

局部历史分支预测 局部历史分支预测（Local History Branch Predictor），也就是 Yale Patt 的 Two-Level Adaptive Branch Prediction。原理也比较简单, 就是用 branch 的 PC 值作为索引，访问 branch history table，读出 history，见图3.2。再用 history 作为索引访问预测值。同样援引 S. McFarling 的数据, 在 predictor 大小大于 128 bytes 的时候, Local history predictor 要优于 bimodal。准确率接近 97.1%

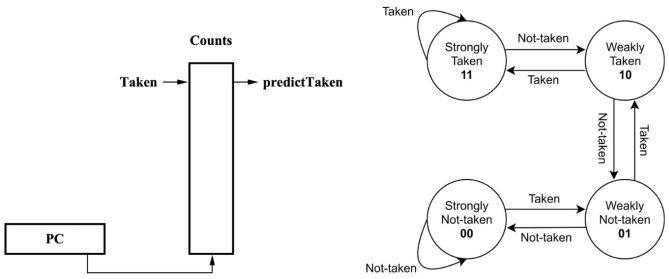


Figure 3.1: Bimodal 设计图和状态图

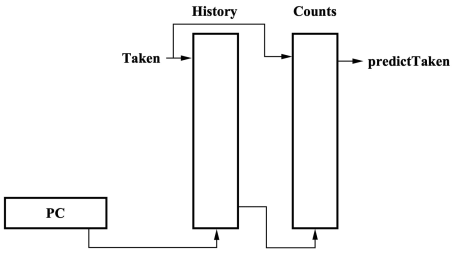


Figure 3.2: 局部历史分支预测设计图

全局历史分支预测 全局历史分支预测（Global History Branch Predictor），该方案记录过去的 n 个 branch 的历史，然后用来索引分支预测。这里放眼全局的 branch，淡化了 PC 的用途。相比于之前的几个预测器，global history 能够结合全局的的信息来指导预测。当 size 足够大的时候会比较准确。如果太小，准确率还不如 bimodal。

全局历史 & 地址分支预测 Global Predictor with Index Selection，简称为 gselect。前面提到单独的全局历史分支预测并没有每个 branch 的地址信息，可能会导致不准确。有人做了改良，用全局历史和 branch 的 PC 一起做索引，来获取预测值，见图3.3。实际实现起来，m 和 n 的选择会导致不同的准确率，其中最好的表现见图3.4。 可以看出这种

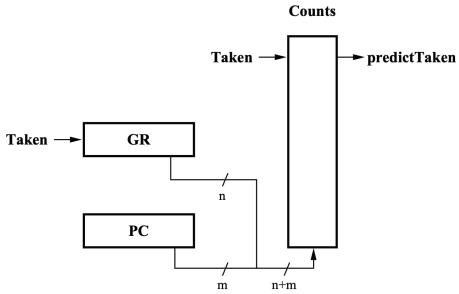


Figure 3.3: gselect 设计图

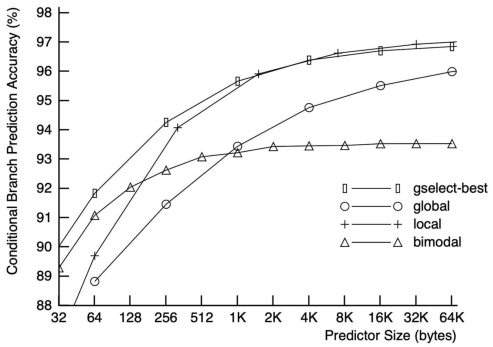


Figure 3.4: gselect 性能对比图

global history + PC 的索引方式有效的提高了小 size 的准确率，也是目前介绍到的第一个全面优于 bimodal 的分支预测。

gshare Global History with Index Sharing，原理与 gselect 相似，只不过把 history 和 PC 由变拼接为 XOR 操作，更好地利用了两个 input，进而减少了混叠。结构图和性能对比见图。从表现来看，在 size 足够大的情况下，gshare 略优于 gselect。值得一提的是，

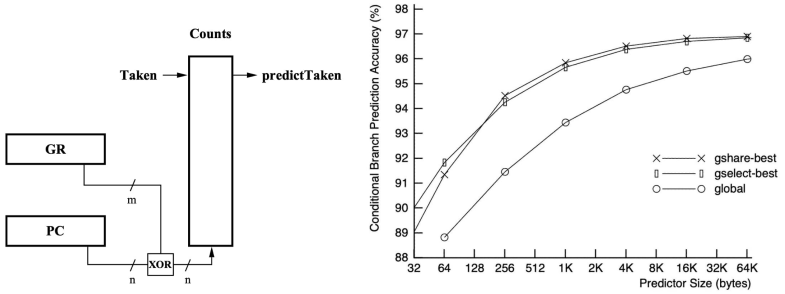


Figure 3.5: gshare 结构图和性能对比

我们的 Genshin 实现了 gshare 方案（可选择开启或关闭）。

以上就是几种基本的分支预测器，一个更高阶方案就是把这些组合起来，择优选取。接下来是地址预测，对于预测 taken 的分支，我们还要给出 target 的 PC，不然等于没有预测。

直接跳转地址预测 一般是通过 BTB 来实现的，以 branch 的 PC 做索引来访问预测地址。由于分支预测是基于 PC 值进行的, 不可能对每一个 PC 值都记录下它的目标地址, 所以一般都使用 Cache 的形式, 使多个 PC 值共用一个空间来存储目标地址, 这个 Cache 称为 BTB(Branch Target Buffer), 图表示了采用直接映射结构的 BTB 的示意图。

BTB 本质上是 Cache, 所以它的结构和 Cache 也是一样的, 使用 PC 值的一部分来寻址 BTB (这部分称为 index), PC 值的其他部分作为 Tag 。BTB 中存放着分支指令的目标地址 (Branch Target Address, BTA), 因为 Index 部分相同的多个 PC 值会查找到 BTB 中

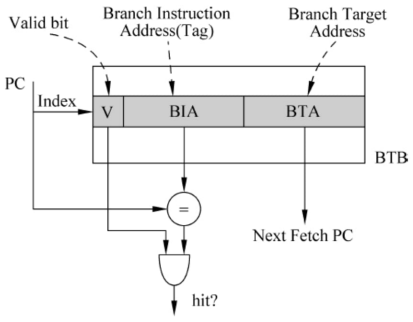


Figure 3.6: 直接映射结构的 BTB

的同一个地方, 所以使用 Tag 来进行区分, 当这些 PC 值中存在多于一条的分支指令时, 就产生了冲突, 这样会造成 BTB 中对应的内容被频繁地替换, 影响了分支预测的准确度。为了解决这个问题, 可以采用组相连结构的 BTB, 当有多条分支指令都指向 BTB 的同一个地方时 (也就是它们的 PC 值的 Index 部分相同), 可以将它们的 BTA 放到不同的 way 中。

以上所讲述的 BTB 的设计都是比较直接的方法, 还可以采用一些其他的方法来对其进行优化, 例如在直接映射结构的 BTB 中, 最优的情况出现在映射到 BTB 中同一个地方的所有指令中, 最多只有一条是分支指令, 在这种最优的情况下, 可以将 BTB 的 Tag 部分进行简化, 只使用 PC 值的一小部分作为 Tag, 这种方法称为 partial-tag BTB, 它可以节省 Tag 所占据的存储空间, 但是当 BTB 中的一个地方对应多条分支指令时, 这种方法就有可能引起目标地址的预测失败。我们的 Genshin 采取了这一方案。

间接跳转地址预测 对于间接跳转类型的分支指令来说, 它的目标地址来自于通用寄存器, 是经常变化的, 所以无法通过 BTB 对它的目标地址进行准确的预测, 所幸的是, 大部分间接跳转类型的分支指令都是用来进行子程序调用的 CALL/Return 指令, 而这两种指令是有规律可循的, 任何有规律的事情都可以进行预测。

注意到函数调用是栈式的。根据 Return 指令的特点, 可以设计一个存储器, 保存最近执行的 CALL 指令的下一条指令的地址, 这个存储器是后进先出的 (Last In First Out, LIFO), 即最后一次进人的数据将最先被使用, 这符合上面讲述的 Return 指令和 CALL 指令的特点, 这个存储器的工作原理和堆栈 (stack) 是一样的, 称之为返回地址堆栈 (Return Address Stack, RAS)。我们可以使用 BTB 对 CALL 指令的目标地址进行预测, 而使用 RAS 对 Return 指令的目标地址进行预测, 这个过程如图所示。

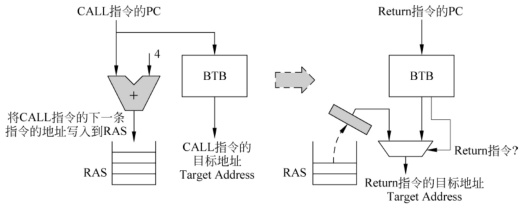


Figure 3.7: 对 CALL/Return 指令进行分支预测

可以看出，要使 RAS 能够正确工作，我们需要在分支预测阶段得到指令的类型，但这似乎是无法直接做到的。一个巧妙的思路是在 BTB 中增加一项用来标记分支指令的类型，后面再次遇到时查询 BTB 即可。另一个问题是，RAS 被占满时应该如何处理？一种是不对新的 CALL 指令进行处理，这样下一次 Return 预测一定会失败。另一种稍好的选择是继续按照顺序向 RAS 写入，RAS 中最旧的内容会被覆盖掉。后者是更好的，因为存在正确的可能性。

对于连续执行的同一个 CALL 指令来说 (此处的连续是指, 两次相邻执行的 CALL 指令是同一条指令), 完全可以将它们的返回地址都放到 RAS 中的同一个地方, 并用一个计数器来标记 CALL 指令执行的次数, 例如使用一个 8 位的计数器就可以最多标记 256 级的递归调用了, 这样相当于扩展了 RAS 的容量, 增大了预测的准确度。

这种带计数器的 RAS 在工作的时候, 会将写人到 RAS 中的 CALL 指令的返回地址和上一次写人的返回地址进行比较, 如果相等, 则表示这两次执行的都是同一个 CALL 指令 (RAS 中保存的返回地址就是 CALL 指令下一条指令的地址), 则此时要将 RAS 当前的读指针指向的表项 (entry) 对应的计数器加 1, 同时这个指针保持不变; 当遇到 Return 指令时, 按照以前的方式从 RAS 中读取数据作为 Return 指令的目标地址, 同时将 RAS 当前的这个表项的计数器减 1 , 如果计数器的值此时为 0 了, 则表示这个表项对应的 CALL 指令已经结束了递归调用, 此时可以使 RAS 的读指针指向下一个 CALL 指令的返回地址, 通过这种工作方式, 可以很好地对递归函数进行分支预测。

在我们的 Genshin 中，我们实现了带计数器的 RAS。

预测错误恢复 对于朴素的流水线，只要把错误的指令冲掉，重新 fetch 正确的指令即可。

3.2 Implementation of Genshin

如上面所说，我们几乎实现了上面所有介绍的分支预测方法。并且支持方案和规模可参数化定制。下面我们介绍下具体情况。

微架构 Genshin Core 每周期会在取指前一拍访问分支预测器，并在下个周期返回结果。如果预测正确，指令流水线就不会断流。我们采用了 Next-Line 分支预测器 (Next-Line Predictor, NLP), 通过综合考虑分支目标缓冲器 (Branch Target Buffer, BTB)、模式历史表 (Pattern History Table, PHT)（可选 GSHARE/BIMODAL）和返回地址栈 (Return Address Stack, RAS) 对不同类型的转移指令做出快速的预测。整体的微结构如下图所示:

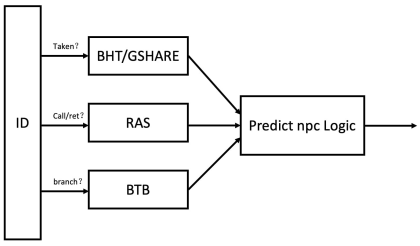


Figure 3.8: NLP 微结构

相关的参数设置在 macro.vh 文件中：

```
//Number of branch target buffer entries.
`define NUM_BTB_ENTRIES                 16'h8
//Set to log2(NUM_BTB_ENTRIES).
`define NUM_BTB_ENTRIES_W               4'h3
//Number of branch history table entries.
`define NUM_BHT_ENTRIES                 16'h8
//Set to log2(NUM_BHT_ENTRIES_W).
`define NUM_BHT_ENTRIES_W               4'h3
//Enable branch history table based prediction.
`define BHT_ENABLE                      1'b0
//Enable GSHARE branch prediction algorithm.
`define GSHARE_ENABLE                   1'b1
//Enable return address stack prediction.
`define RAS_ENABLE                      1'b1
//Number of return stack addresses supported.
`define NUM_RAS_ENTRIES                 4'h8
//Set to log2(NUM_RAS_ENTRIES_W).
`define NUM_RAS_ENTRIES_W               4'h3
```

在这样的寄存器占用非常小的初始设置中，性能测试最多有近 80% 的性能提升（Test 3CCT）。

预测机制 BTB 中缓存了跳转指令的地址高位信息、跳转目标、指令类型等信息。在做分支预测时会用取指 PC 索引 BTB 表项, 如果 PC 高位与读到的 BTB 表项的标签匹配则认为 BTB 命中, 再根据 BTB 中记录的指令类型判断跳转方向和跳转目标。如果类型为条件分支指令，则需要访问模式历史表 (PHT) 来判断是否跳转；如果类型为返回指令，则选择返回地址栈 (RAS) 的栈顶内容作为跳转目标；如果类型为直接或间接跳转指令，则选择 BTB 中记录的跳转目标。

更新机制 在初始化阶段, BTB 所有表项都是无效的, 当跳转指令第一次被取出时都会按不跳转来处理。检测到指令跳转目标错误或者方向错误时会刷新流水线，同时更新 BTB 相应的表项（使用 LFSR 进行随机替换）。如果取指 PC 对应的表项无效或已被占用，则重新分配该表项。与此同时，每执行一条条件分支指令，都会将正确跳转方向和分支目标等信息返回给分支预测单元, 分支预测器会更新 PHT。每执行一条函数调用指令 (call) 或函数返回指令 (ret)，也会将相关信息传给分支预测器。如果该指令是 call 指令，则根据指令的宽度计算返回地址并写入 RAS 栈顶；如果该指令是 ret 指令，则将 RAS 的栈顶弹出。

Chapter 4

Exception Handler

为了实现异常机制，我们没有按照比较传统的做法为流水线增加一个专门进行异常处理的阶段，而是在原本的数据通路中设置了一条平行的异常相关信号的流通路线。

我们在寄存器组的旁边设置了 CSR 存储单元，并在 EXE 段内平行地进行异常处理。ID 段会读出 CSR 数据，就像读出寄存器组的数据一样，但是 CSR 的写回在 EXE 段就会执行，而不是等到 WB 段。

这样做的好处是新增的功能不影响流水线已有的设计，尤其是不影响数据冲突的处理，如此一来我们只需要考虑异常相关的新增功能与冲突即可。这样做的缺点是需要在 EXE 段提前判断 MEM 段可能出现的异常，尤其是在增加页表支持后，设计会变得更加困难。

4.1 CSR

CSR CSR 存储单元的设计类似于寄存器组，提供组合读与时序写。

与寄存器组不同，CSR 的合法索引空间非常大，而我们实际需要实现的 CSR 很少，所以我们没有采用像寄存器一样的紧密编址方式，而是稀疏地为一些地址对应 CSR，并将其余地址视为非法索引。

此外，在异常处理中，往往需要同时读或写多个 CSR，因此我们也为 CSR 存储单元增加了多读、多写功能。其中多读功能只是简单地将各个 CSR 直接连到输出端口上，而多写功能则使用一个独立的写使能信号，若开启多写使能，则普通的根据索引进行写操作的信号会被忽视。

数据冲突处理 在我们的实现中，CSR 的读可能发生在 ID 段和 EXE 段，其中 ID 是索引读，只会在 CSRXX 指令中用到，EXE 段是多读，只会在上下文切换触发时用到。CSR 的写只会发生在 EXE 段，可能是索引写也可能是多写。

我们需要考虑两种读操作可能遇到的数据冲突。

- 多读发生在 EXE 段，而上一条指令刚刚在 EXE 段完成写操作，因此 EXE 段看到的 CSR 永远是最新的。所以多读并不会遇到数据冲突。

- 索引读发生在 ID 段，上一条指令可能刚在 EXE 段执行了写操作，这种冲突类似于读寄存器时与 WB 段的指令发生数据冲突，因此处理方法也是在 CSR 存储单元中对读写的索引进行判断，并在有必要时将写入的数据直接连入读的输出信号。这种处理方式仅仅解决了索引读与索引写的冲突，而没有解决索引读与多写的冲突，但是多写一定发生在上下文切换时，而此时前面两个流水线阶段的信号都会被刷掉，所以这种冲突并不需要考虑。

4.2 ExceptionHandler

ExceptionHandler 该模块同时实现了由异常引起上下文切换的功能，以及普通的 CSR 写操作。该模块对如上功能的支持有特定的优先级，先检测是否需要触发上下文切换，若不需要触发，再执行 CSR 写操作。

上下文切换功能使用 CSR 存储单元的多读、多写功能，会按照符合手册定义的方式进行 CSR 操作，并将上下文切换的信号拉高，进而将当前指令及之后的两条指令刷掉。这部分对各种异常信号进行了优先级划分，首先要保证最早触发异常的指令最先被处理，然后才是保证优先级符合手册的定义。

普通的 CSR 写操作使用 CSR 存储单元的索引写操作，和 WB 段的写回方式类似。

异常信号的产生 从原理上讲，很多阶段都可能抛出异常。对于 EXE 之前的阶段，我们会将异常信号存储起来，随着流水线的流动送到 EXE 段，同时将触发异常的指令处理成 NOP；对于 EXE 之后的阶段，我们会提前在 EXE 段进行判断。特别地，MEM 段的 page fault 在监控程序中属于 fatal error，应当直接重启，所以我们并没有严谨地考虑如何在此时刷掉指令，而是直接将 page fault 信号接到 ExceptionHandler 中。

中断信号并不产生自任何一个阶段，而是来自外部，所以我们直接将中断信号接入 ExceptionHandler。

目前我们支持如下的异常：时钟中断，ecall/ebreak/mret 指令，load access fault，store access fault，IF/load/store page fault。

Chapter 5

VGA Supports

我们实现的 VGA 可以通过执行汇编代码投影 400 * 300 大小的图片至板上。

预处理 预处理首先将图片分辨率处理至 400 * 300 大小，然后通过 python 脚本，将图片转换为汇编。具体转换过程为：对图片的每个点进行像素的分析，获取其 rgb 值，投影到 vga 所能处理的范围内（每个像素点 8bit），然后向 0x30000000+n 处 store 相应的值，处理完所有点后增加一句 return，此时汇编生成成功，再将汇编转化为二进制文件，即可完成预处理部分。

Bram 使用 Bram 对 VGA 相关信息进行存储。通过 vivado 带有的 bram generator，生成合适大小的 Bram，具体为地址 17 位，数据 8 位，并拥有两个端口，这样可以足够保存 400 * 300 的 VGA 颜色信息。

数据的存储过程 读取到向 0x30000 开头的地址写入的操作时，sram_uart 状态机跳转到写入 bram 状态，向 bram 地址为原地址相对于 0x30000000 的 offset 处写入。

VGA 的展示 定义一个名为 vga_show 的模块，通过 50M 时钟驱动，每次时钟上升沿，通过当前的 vdata 和 hdata，推导得知当前位置所需要的颜色在 bram 中的位置，然后通过 bram 获取出数据，给当前点的颜色进行赋值，这样就完成了 vga 的展示。

Chapter 6

Page Tables

我们在异常/中断版本的基础上开发了支持页表的版本。我们实现了对 Sv32 页表格式的支持，对 page fault 的支持，以及一个简单的 TLB。因为新增的功能增大了时序压力，所以该版本需要在 10M 时钟下运行。这无法体现我们之前实现的各种加速策略的效果，所以我们将页表版本作为一个单独的分支进行展示。

新增 CSR 与指令 新增的 CSR 为 satp，存储了页表相关的信息，该 CSR 的实现方式与之前实现的其他 CSR 没有区别。新增的指令为 sfence.vma，用于刷新 TLB。我们实现的 TLB 更接近于 Cache，而且我们也只想要支持监控程序，没有更高的要求，所以该指令被解析为 NOP。

页表查询 我们首先实现了基础的页表查询功能。我们在地址解析时进行了优先级划分，优先接受串口、VGA 等特殊定义的地址，然后在启用页表功能的情况下接收所有地址，最后在不启用页表功能的情况下接收合法的 SRAM 地址。其中只有启用页表功能情况下接收到的地址会导向页表查询，其他分支都和之前的版本保持一致。

根据特权手册的定义，Sv32 格式的页表分为两级，且在两级页表中均可能命中。所以我们将页表查询分为两步，且在第一步时就允许命中。实现细节与手册定义保持一致。完成页表查询后，得到 SRAM 物理地址，然后转入常规的 SRAM 访问分支。

我们实现了 TLB 以加速页表查询过程。实现方式基本和之前实现的 Cache 相同，细节可参考文档的 Cache 部分。这可能不是典型的 TLB 实现方法，不过也能够发挥作用。

page fault page fault 只可能在页表查询过程中触发。我们仅将控制位 V 是 0 作为触发条件，没有考虑其他触发 page fault 的方式。检测到 page fault 后，再根据当前是 IF 访存还是 MEM 访存，以及是读还是写，生成相应的控制信号。

load/store 触发的控制信号直接连到 ExceptionHandler 中。该模块处理的所有异常信号都对应于当前位于 EXE 段的指令，只有这两个 page fault 相关信号对应于当前位于 MEM 段的指令，所以这两个信号的处理优先级最高；此外这两个信号也对 WB 段进行一些干预，避免错误的写回。if 触发的控制信号送回到 IF 段，跟随流水线传递到 ExceptionHandler 中。

Chapter 7

Sram&Uart and Cache

在 sram_uart 中，我们实现了读写串口、读写 sram、写 Bram、查页表、时钟中断功能，同时对于性能的提升，实现了 cache 与 tlb 功能，本部分仅叙述读写串口、sram、cache 与 tlb 功能。

总体架构 在 sram_uart 中，初始状态为 STATE_IDLE，此时对需要初始化的数据进行初始化，并把 stall 信号设为 1，停止流水线的运行；IDLE 状态后，根据不同的地址进入不同的读写状态；读写状态后，进入 STATE_RUN 状态，将 stall 信号设为 0，此时数据准备好，流水线启动，并重新跳转回 IDLE 状态。

读写串口 读串口分为两个状态，第一个状态控制使能，第二个状态获取数据并转到 RUN 状态；写串口分三个状态，第一个状态准备好数据，第二个状态控制使能，第三个状态完成写并转到 RUN 状态。

读写 sram 读 sram 分为两个状态，第一个状态控制使能和地址，第二个状态获取数据并转到 RUN 状态；写 sram 分为三个状态，第一个状态准备好数据与地址，第二个状态控制使能，第三个状态完成写并跳转到 RUN 状态。

Cache Cache 存储 sram 中的数据，使用地址后六位作为 index，每次读取数据时先查询 cache 中的数据与当前地址是否符合，如果符合，就省去访存步骤，直接从 cache 中获取数据；如果没有找到，那么在从内存中读取数据时，就需要对 cache 中的数据进行相应的更新；写入数据时，同样也需要对 cache 中的数据进行更新。

TLB TLB 的实现与 Cache 相似，也是使用地址的后六位作为 index，对一二级页表项分别进行缓存，查找数据时有限在 tlb 中查找，找到则省去访问内存步骤，找不到则更新 tlb。

Chapter 8

Achievement Display

本部分对我们实现的功能进行展示

基础功能 我们的完整功能版本 CPU 在平台自动测评中能获得满分，这表明基础功能均正确实现。

性能相关 因为实现页表时，我们将时钟从 50M 降低为 10M，所以我们选择以页表之前的版本进行性能测试。为了展示出缓存和分支预测的效果，下面将提供三份不同版本的实现，具有逐渐提高的性能。基线版本：未实现缓存和分支预测。可以看到有两个测例超时。

```
=== Test 1PTB ===
Boot message: 'MONITOR for RISC-V - initialized.'
INFO: running in 32bit, xlen = 4
ERROR: timeout during WaitG
Test 1PTB run for 30.029s (Timeout)
```

```
=== Test 2DCT ===
Boot message: 'MONITOR for RISC-V - initialized.'
INFO: running in 32bit, xlen = 4
Test 2DCT run for 16.106s
```

```
=== Test 3CCT ===
Boot message: 'MONITOR for RISC-V - initialized.'
INFO: running in 32bit, xlen = 4
ERROR: timeout during WaitG
Test 3CCT run for 30.029s (Timeout)
```

```
=== Test 4MDCT ===
Boot message: 'MONITOR for RISC-V - initialized.'
INFO: running in 32bit, xlen = 4
Test 4MDCT run for 28.186s
```

```
=== Test CRYPTONIGHT ===
Boot message: 'MONITOR for RISC-V - initialized.'
INFO: running in 32bit, xlen = 4
Test CRYPTONIGHT run for 1.615s
```

仅实现缓存的版本：可以看到性能有明显提升，这是因为缓存机制有效减少了访存所需周期数。

```
=== Test 1PTB ===
Boot message: 'MONITOR for RISC-V - initialized.'
INFO: running in 32bit, xlen = 4
Test 1PTB run for 24.159s
```

```
=== Test 2DCT ===
Boot message: 'MONITOR for RISC-V - initialized.'
INFO: running in 32bit, xlen = 4
Test 2DCT run for 12.080s
```

```
=== Test 3CCT ===
Boot message: 'MONITOR for RISC-V - initialized.'
INFO: running in 32bit, xlen = 4
Test 3CCT run for 28.186s
```

```
=== Test 4MDCT ===
Boot message: 'MONITOR for RISC-V - initialized.'
INFO: running in 32bit, xlen = 4
Test 4MDCT run for 22.817s
```

```
=== Test CRYPTONIGHT ===
Boot message: 'MONITOR for RISC-V - initialized.'
INFO: running in 32bit, xlen = 4
Test CRYPTONIGHT run for 1.433s
```

实现缓存和分支预测的版本：性能又有一定提升，且能注意到部分具有大量跳转的测例有更明显的提升。

```
=== Test 1PTB ===
Boot message: 'MONITOR for RISC-V - initialized.'
INFO: running in 32bit, xlen = 4
Test 1PTB run for 20.133s
```

```
=== Test 2DCT ===
Boot message: 'MONITOR for RISC-V - initialized.'
INFO: running in 32bit, xlen = 4
Test 2DCT run for 11.073s
```

```
=== Test 3CCT ===
Boot message: 'MONITOR for RISC-V - initialized.'
INFO: running in 32bit, xlen = 4
Test 3CCT run for 16.106s
```

```
=== Test 4MDCT ===
Boot message: 'MONITOR for RISC-V - initialized.'
INFO: running in 32bit, xlen = 4
Test 4MDCT run for 22.817s
```

```
=== Test CRYPTONIGHT ===
Boot message: 'MONITOR for RISC-V - initialized.'
INFO: running in 32bit, xlen = 4
Test CRYPTONIGHT run for 1.350s
```

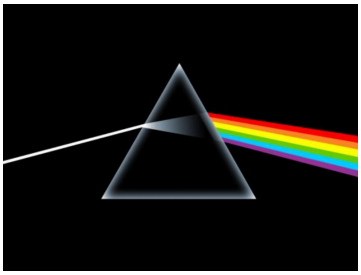
额外功能 VGA：我们实现的 VGA 功能需要依赖于专门生成的汇编程序来展示，我们支持将 400×300 的图片显示到 vga 上。代码和例图如下：

```
import cv2
image = cv2.imread("photo.jpg")
size = image.shape
f = open('vga_photo.s', 'w', encoding='utf-8')
print(image.shape)
for i in range(0, 300):
        for j in range(0, 400):
                red = int(0.5 + image[i][j][2] * 1.0 / 255 * 7)
                green = int(0.5 + image[i][j][1] * 1.0 / 255 * 7)
                blue = int(0.5 + image[i][j][0] * 1.0 / 255 * 3)
                color = (red << 5) + (green << 2) + blue
                str0 = str(hex(400 * i + j))
                l = len(str0)
                str0 = str0[2:l]
                while len(str0) < 7:
                str0 = '0' + str0
                str1 = 'li t1, 0x3' + str0 + '\n'
                f.write(str1)
                str1 = 'li t0, ' + str(hex(color)) + '\n'
                f.write(str1)
                str1 = 'sw t0, (t1)\n'
                f.write(str1)
        print(i)
f.close()
```

(a)

(b)

(c)



(d)

异常和中断：使用终端连接板子并运行自定义程序的方式进行展示。

系统调用：

```
>> f
>>file name: ecall.s
>>addr: 0x80400000
reading from file ecall.s
[0x80400000] li s0, 30
[0x80400004] li a0, 0x4F  # O
g[0x80400008] ecall
[0x8040000c] li a0, 0x4B  # K
[0x80400010] ecall
[0x80400014] ret
>> g
addr: 0x80400000
started
OK
elapsed time: 0.000s
>>
```

时钟中断：

```
>> f
>>file name: loop.s
>>addr: 0x80400000
reading from file loop.s
[0x80400000] end:
[0x80400000] beq x0, x0, end
>> g
addr: 0x80400000
started
OK
elapsed time: 0.604s
```

地址相关异常：

```
>> f
>>file name: exp_mem.s
>>addr: 0x80400000
reading from file exp_mem.s
[0x80400000] li a0, 0x50505050
[0x80400008] lw a0, 0(a0)
>> g
addr: 0x80400000
started
supervisor reported an exception during execution
  mepc: 0x80400008
  mcause: 0x00000005
  mtval: 0x00000000
```

页表和 TLB：用同样的方法进行测试。

系统调用：

```
>> f
>>file name: ecall.s
>>addr: 0x80100000
reading from file ecall.s
[0x80100000] li s0, 30
[0x80100004] li a0, 0x4F  # O
[0x80100008] ecall
[0x8010000c] li a0, 0x4B  # K
[0x80100010] ecall
[0x80100014] ret
>> g
addr: 0x00000000
started
OK
elapsed time: 0.000s
```

时钟中断：

```
>> f
>>file name: loop.s
>>addr: 0x80100000
reading from file loop.s
[0x80100000] end:
[0x80100000] beq x0, x0, end
>> g
addr: 0x00000000
started
killed timeout program.

elapsed time: 5.000s
```

page fault:

```
>> f
>>file name: exp_mem.s
>>addr: 0x80100000
reading from file exp_mem.s
[0x80100000] li a0, 0x50505050
[0x80100008] lw a0, 0(a0)
>> g
addr: 0x00000000
started
supervisor reported an exception during execution
  mepc: 0x0000000c
  mcause: 0x00000013
  mtval: 0x00000000
```

接下来我们尝试展示 TLB 的作用。我们实现的时钟中断与标准有一定的区别，我们的 mtime 寄存器只会在访存 stall 撤除后的一个周期内递增，而不是每个时钟周期都增加，这样我们可以通过 loop.s 触发超时之前的运行时间，感受到访存所需周期数的大致区别。在上面的例子中，我们使用的是增加 TLB 的版本，下面则是不使用 TLB 的版本，可以观察到超时前的运行时间更长。这说明 TLB 确实起到了加快访存的作用。

```
>> f
>>file name: loop.s
>>addr: 0x80100000
reading from file loop.s
[0x80100000] end:
[0x80100000] beq x0, x0, end
>> g
addr: 0x00000000
started
killed timeout program.

elapsed time: 7.009s
```