LESC-5 CPU

CPU设计

LESC-5 CPU（linmo's easy-to-learn, highly scalable and customizable, five-stage pipeline CPU），是我基于RISC-V架构设计，并以Chisel语言实现的，具有易学性、高可扩展性和可定制性等诸多优点的，具有分支预测、旁路转发等功能的五级流水线CPU。

本节将主要介绍LESC-5 CPU的设计过程和实现细节，包括对单周期、流水线这两个版本的不同的设计图和代码细节的分析与讲解，以及Config部分、Utils部分和调试信息等用于增加LESC-5 CPU的易学性、扩展性和可定制性的额外辅助部分，最后将分析实验方案及实验结果。

1 总体设计

整个设计和实现的过程中，我充分结合了RISC-V指令集的精简高效、模块化等特性，和Chisel语言的高层次语言的面向对象、参数化和高扩展性等特点，以易学性、高可扩展性和可定制性为设计理念，进行CPU的设计与实现。

在设计和实现中，LESC-5 CPU主要参考了DINO的框架设计，并参考了包括Sodor在内等多个CPU的设计与实现，在它们的基础上取长补短，并通过增加丰富的注释文档和Config统一部分等方式提高LESC-5 CPU的可读性以及可扩展性。

LESC-5 CPU主要分为单周期CPU和流水线CPU两个版本，其中流水线CPU还根据是否具有分支预测、旁路转发分为不同的多个版本。

2 Config部分

CPU的配置信息存储在名为CPUConfig.scala的文件中，通过object CPUConfig来提供全局的CPU配置信息，以供整个系统调用。

其中包括机器相关的配置、指令内存相关的配置、数据内存相关的配置、字节宽度相关的配置等，如机器位数、机器端序、内存类型、寄存器个数、指令宽度、指令内存大小、数据单位的字节大小和数据内存大小等。

此外，CPUConfig文件还提供了一些常量和参数以及初始化相关的配置等，例如ALU操作码宽度、分支预测的配置、是否预先保留栈空间、预先开栈大小和测试时的周期超时上限等。

使用CPUConfig文件的好处如下：

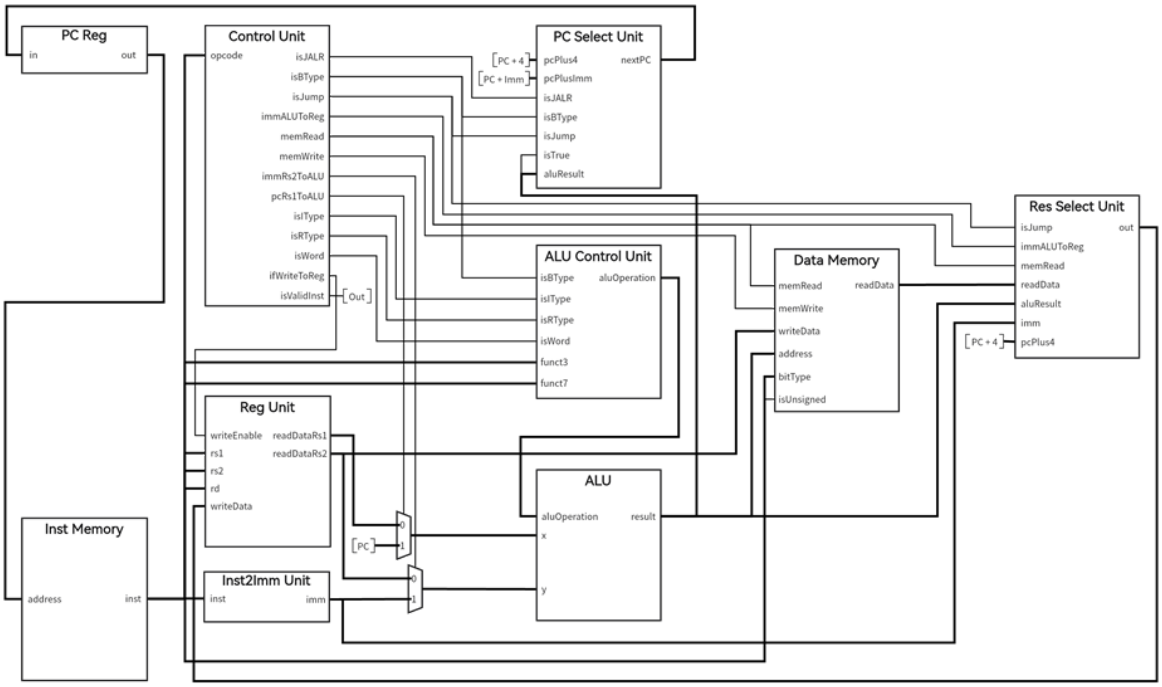
1. 方便管理：将各种配置信息集中在一个文件中，方便查找、修改和维护，提高代码的可读性和可维护性。
2. 可复用性：这些配置信息可以在不同的模块中被引用和复用，避免了重复的定义和声明，提高了代码的可重用性。
3. 可扩展性：在开发过程中，如果需要增加或修改某个配置信息，只需要修改这个文件，不需要在整个代码中逐一修改，提高了代码的可扩展性。
4. 可移植性：这个文件中定义的配置信息可以适用于不同的CPU实现，提高了代码的可移植性。

这极大地提高了我所设计的CPU的可扩展性，而这恰是已有的类似的CPU（如DINO）所缺少的。

此外，Config文件中还包含多个Map映射表，包括指令类型和控制码的映射表、ALU操作和ALU控制码的映射表等。映射表极大地增加了代码的可读性的同时简化了代码，同时给予了统一的修改接口（例如get方法），使得定制化的扩展变得非常易于实现（例如为ALU计算操作增加乘法、除法，或是扩展指令集，使其满足浮点操作等）。

3 单周期CPU

在单周期部分，设计图如下：



主要包括了PC Reg、Inst Memory、Control Unit，Reg Unit、Inst2Imm Unit、PC Select Unit、ALU Control Unit、ALU、Data Memory、Res Select Unit等组件，可正确执行RISC-V 64I的指令集并通过RISC-V的官方测试套件。设计图中，为了方便阅读，所有输入都列在元件左侧，而输出则列在元件右侧；粗黑线表示多位总线，细黑线表示单位信号线。在设计图上，我形象地将所有组件大致分为5个阶段，以降低后续流水线组件的学习成本。接下来，我将以五个部分（寄存器部分、存储单元部分、译码器部分、计算部分、结果选择部分）一一讲解各个组件。

3.1 寄存器部分

寄存器部分负责在时钟周期间存储数据，包括 PC 寄存器和寄存器单元两个模块。

PC 寄存器用于存储下一条指令的地址，并将该地址输出给指令存储器；寄存器单元由 32 个寄存器组成，可执行读/写寄存器的操作。该部分还需考虑是否需要提前开栈，同时提供了两个版本：不允许读写转发的版本和附带读写转发的版本，前者用于单周期 CPU，后者用于多周期 CPU。输入包括寄存器编号、写使能、写入数据等，输出包括从寄存器中读出的数据。

1. PC寄存器（PC Reg）

PC 寄存器，有一个输入端口in和一个输出端口out。输入端口接收来自 PCSelectUnit.nextPC 的下一个 PC 地址，将其存储在寄存器中；输出端口提供当前存储在寄存器中的地址。

此外，PC寄存器会接收外界参数startAddress（默认为0），表示指令开始执行的起始位置。

输入

- in : 下一个 PC, 来自 PCSelectUnit.nextPC

输出

- out : 输出当前 PC

2. 寄存器单元（Reg Unit）

寄存器单元，由 32 个寄存器形成的寄存器单元, 执行 读/写寄存器 的操作，输入的rs1、rs2、rd等寄存器编号由指令的不同部分得到，注意诸如 x0 等特殊寄存器的处理。此外，还需考虑是否要提前开栈(这一操作由操作系统完成, 但因为不支持特权模式, 故无法运行操作系统, 裸机需要主动提前开栈)。单元包含一个RegGroup向量类型，存储着32个寄存器的数据。输出包括读出的rs1、rs2寄存器的数据。当写使能为真且要写入的寄存器不是x0时，写入要写入的数据到对应的寄存器中。

提供2个版本：不允许读写转发的版本和附带读写转发的版本。前者用于单周期CPU，避免产生电路回环；后者用于多周期CPU，加快寄存器的写入和读出，避免等待。

输入

- rs1 : rs1 寄存器, 根据指令的 (19-15) 得到
- rs2 : rs2 寄存器, 根据指令的 (24-20) 得到
- rd : rd 寄存器, 根据指令的 (11-7) 得到
- writeEnable : 写使能, 来自于 ControlUnit.ifWriteToReg
- writeData : 要写入的数据, 来自于 ResSelectUnit.out

输出

- readDataRs1 : 从 rs1 中读出的数据
- readDataRs2 : 从 rs2 中读出的数据

3.2 存储单元部分

存储单元部分负责对数据和指令的存储和读取，包括指令存储单元和数据存储单元。

指令存储单元存储待执行的指令，根据 PC 依次取出进行执行。数据存储单元存储数据，根据指令进行读/写。存储单元均采用小端序，不允许非对齐数据访问。数据存储单元采用同步写、异步读的形式，支持按照字节、半字、字、双字进行读/写操作，并可选择零扩展或符号扩展。

1. 指令存储单元（Inst Memory）

存储待执行的指令, 根据 PC 依次取出进行执行。存储单元为一个内存模块，各种参数设置由Config部分统一控制。RISC-V 默认为小端序，不允许非对齐数据访问(所有访问会对 4 字节对齐)。该模块具有一个输入端口 address，表示取值地址，来自 PCReg.out，以及一个输出端口 inst，表示输出对应地址的指令。如果给定了 hexFilePath，该模块会根据路径，调用loadMemoryFromFile方法从该文件中加载指令到内存中。

输入

- address : PC, 即取值地址, 来自 PCReg.out

输出

- inst : 输出对应地址的指令

2. 数据存储单元（Data Memory）

存储数据, 根据指令进行 读/写。存储单元和指令存储单元类似，各种参数设置由Config部分统一控制，以8字节为单位进行存储。数据存储单元可以按照指令的要求进行16/32/64位读写操作，并根据指令要求进行符号扩展或零扩展。RISC-V 默认是小端序, 采用"同步写, 异步读"的形式, 不允许非对齐数据访问(即所有访问都要对2的次方对齐)。如果读取操作无效，则返回0。同样在给定了hexFilePath后，可以根据路径调用方法从文件中读取数据。

输入

- memRead : 是否需要读

- memWrite : 是否需要写
- address : 读/写 的地址 (按字节)
- writeData : 待写入的数据
- bitType : 是否要按照 16/32/64位 进行 读/写, 为指令的 (13-12)

00 : 字节, 01 : 半字, 10 : 字, 11 : 双字

- isUnsigned : 做 零扩展/符号扩展, 为指令的 (14)

输出

- readData : 读出的数据，若 memRead = false, 则读出 0.U

3.3 译码器部分

译码器部分负责对指令等进行译码，生成对应的控制信号或是所需要的数据，包括控制单元、ALU控制码生成单元和立即数生成单元。

控制单元通过译码指令生成对应的控制信号，包括指令类型、跳转控制、内存访问控制等。ALU控制码生成单元根据控制单元的控制信号和指令中的方法码生成对应的ALU控制码，指挥ALU的执行。立即数生成单元根据32位指令生成对应类型的立即数。

1. 控制单元（Control Unit）

根据操作码生成对应的控制信号。此处的实现参考学习了DINO，使处理逻辑清晰明了。

先通过对64I指令集进行分析，对控制信号进行设计，整理出对应的表格，随后再利用Chisel3的util中的ListLookup和BitPat，直接进行位匹配，将指令译码为对应的控制信号。每个指令都有自己属于的指令类型，如B-type、Load等，根据匹配将对应的控制信号进行输出。

输入

opcode : 操作码, 来自 InstMemory.inst 的 (6-0)，为指令的低 7 位

输出

- isJALR : 1 for jalr, 等同于 jumptype(0)

为 true.B, (jalr) 则 next_pc 来源于 ALU

- isBType : 是否为 B-type 指令, 1 for B-type
- isJump : 是否为 jump 指令, 1 for jal/jalr, 等同于 jumptype(1)
- immALUToReg : 将 立即数/ALU的结果 放入寄存器,

1 for immediate, 0 for ALU's result

- memRead : 是否需要从 DataMemory 中读取数据
- memWrite : 是否需要向 DataMemory 中写入数据
- immRs2ToALU : 控制 ALU.y 的输入

1 for immediate, 0 for RegUnit.readDataRs2

- pcRs1ToALU : 控制 ALU.x 的输入

1 for PCReg.out, 0 for RegUnit.readDataRs1

- isIType : 是否为 I-type 指令
- isRType : 是否为 R-type 指令
- isWord : 是否为 '*W' 型指令, 即按 word 处理
- ifWriteToReg : 是否要将结果写入寄存器

- isValidInst : 是否是有效的指令

2. ALU控制码生成单元（ALU Control Unit）

ALU控制码生成单元，根据 ControlUnit 的控制信号、funct3 和 funct7 生成对应的 ALU 的控制信号，指挥 ALU 的执行。其中通过Config部分的映射表，从代码层面简化了ALU的计算操作和ALU控制码之间的映射关系，使得代码简洁且可读性高，同时易于扩展。

输入

- isBType : 当前指令是否为 B-type 指令
- isIType : 当前指令是否为 I-type 指令
- isRType : 当前指令是否为 R-type 指令
- isWord : 是否为 '*W' 型指令, 即按 word 处理
- funct3 : 指令中的 3 位方法码, 来自于指令的 (14-12)
- funct7 : 指令中的 7 位方法码, 来自于指令的 (31-25)

输出

- aluOperation : 根据输入生成的 6 位控制码

若为 R-type, 组成为 funct3|inst(30)|isWord|isBType

若为 B-type, 组成为 funct3|0|isWord|isBType

若为 I-type, 则为对应的 R-type 的控制码

对于 SRLI / SRAI / SRLIW / SRAIW, 为 funct3|inst(30)|isWord|isBType

对于其他的, 为 funct3|0|isWord|isBType

若均不是, 则生成默认控制码(将执行 ADD 操作)

3. 立即数生成单元（Inst2Imm Unit）

根据 32 位指令生成对应 type 的立即数。单元主要由Chisel语言提供的switch语句组成，根据指令的不同类型，使用不同的信号译码拼接的转换方式，清晰明了地得到对应的立即数。

输入

- inst : 来自 InstMemory 取出的指令, 完整的 32 位

输出

- imm : 根据 32 位指令生成的对应的符号扩展的立即数

3.4 计算部分

计算部分负责CPU中主要计算，包含ALU。

ALU接收来自ALU控制单元的操作码以及两个输入数据，进行计算，并输出计算得到的结果。操作码中的最低位表示是否为分支操作：若是，则根据操作码执行对应的分支比较操作；否则，则根据操作码执行对应的算术或逻辑计算，包括ADD、SUB、SLT、SLTU、XOR、OR 和 AND，以及各种移位操作。对于32位数据，根据操作码进行判断，执行对应的符号扩展或无符号扩展的操作。

1. 计算单元（ALU）

根据 ALU 控制单元传来的操作码, 对数据进行计算

输入

- aluOperation : 操作码, 来自 ALUControlUnit

combine|isWord|isBType

- x : 待计算的数据 1, pc / rs1
- y : 待计算的数据 2, imm / rs2

输出

- result : 计算得到的结果

3.5 结果处理部分

结果处理部分负责对CPU中复杂的数据流动进行管理，包含 PC 选择单元和结果选择单元。

PC选择单元根据控制信号选择下一个PC，可选择 PC+4 、PC+立即数等。结果选择单元根据控制信号选择要保存在寄存器中的结果，可选择立即数、ALU计算结果、内存中读出的数据或PC+4的值。

1. PC选择单元（PC Select Unit）

根据输入接口的值，通过组合电路的逻辑控制，计算出下一个程序计数器（nextPC）和跳转信号（jump，判断当前是否进行了跳转，用于分支预测），并将结果分别输出到对应的接口。默认是PC+4，当isJALR为真时，跳转到ALU计算结果所指示的地址，否则当isJump或isBType与isTrue均为真时，跳转到PC+immediate所指示的地址。

输入

- pcPlus4 : PC+4
- pcPlusImm : PC+immediate
- isJALR : 为 true 则 next_pc 来源于 ALU, 1 for jalr
- isBType : 是否为 B-type 指令, 1 for B-type
- isJump : 是否为 jump 指令, 1 for jal/jalr
- isTrue : 对于 B-type 指令, ALU 计算的结果是否满足跳转条件

来源于 ALU.result(0)

- aluResult : ALU 的计算结果

输出

- nextPC : 下一个 PC

2. 结果选择单元（Res Select Unit）

根据控制信号选择所需要的结果。当输入信号io.memRead为真时，输出信号io.out等于输入信号io.readData；当输入信号io.isJump为真时，输出信号io.out等于输入信号io.pcPlus4；当输入信号io.immALUToReg为真时，输出信号io.out等于输入信号io.imm；否则，输出信号io.out等于输入信号io.aluResult。

输入

- isJump : 若 isJump = True, 则将 PC+4 放入寄存器(保存返回地址)
- immALUToReg : 将 立即数/ALU的结果 放入寄存器
- memRead : 若 memRead = True, 则将 DataMemory 中取出的值放入寄存器

- readData : 从存储单元读取出的数据
- aluResult : ALU 的计算结果
- imm : 立即数
- pcPlus4 : PC+4

输出

- out : 根据选择信号和输入的数据, 得出结果用于存入寄存器

4 流水线CPU

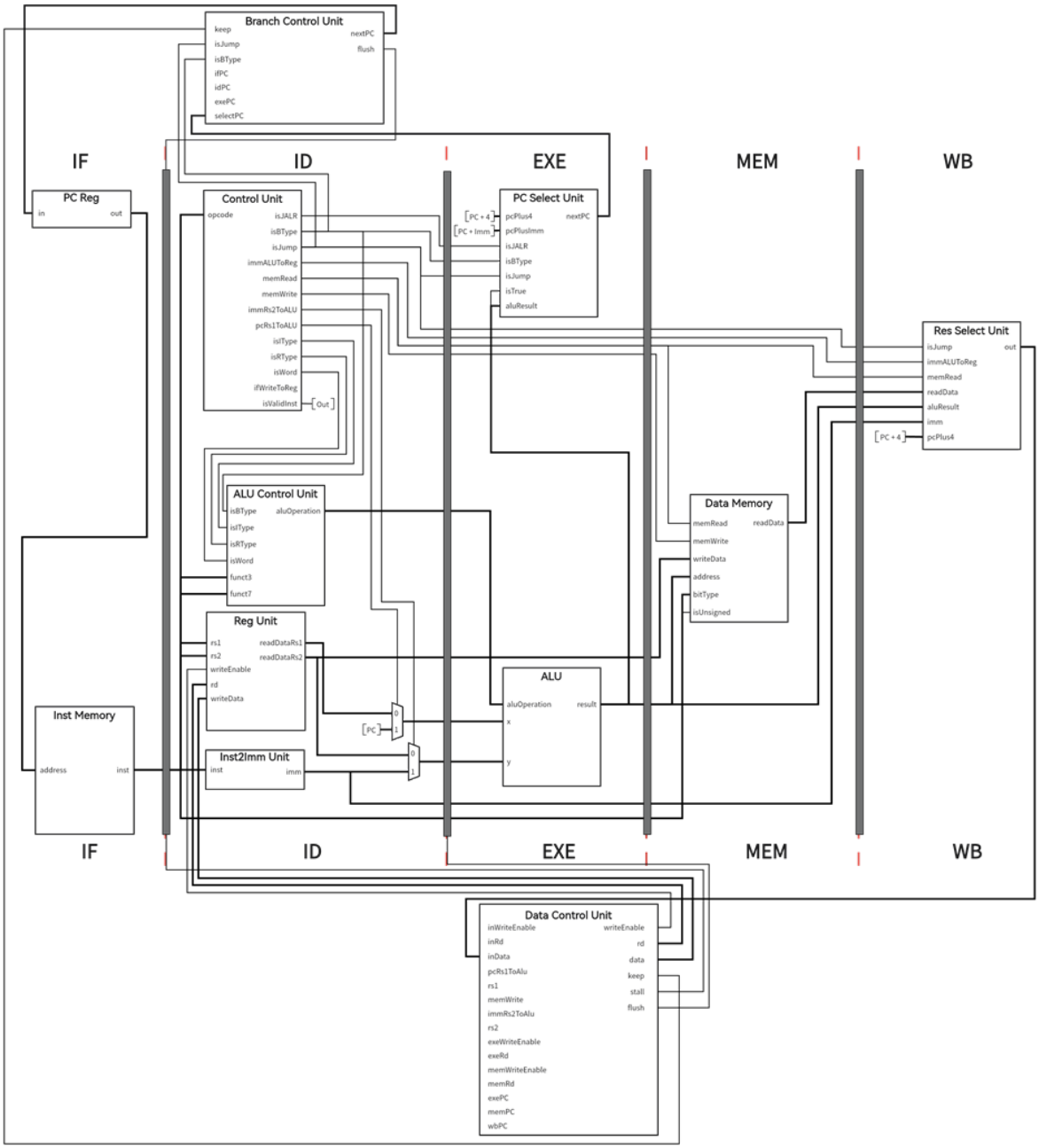
流水线CPU在单周期CPU的基础上，参考MIPS的经典五级流水线，增加了五级流水线，分别为：

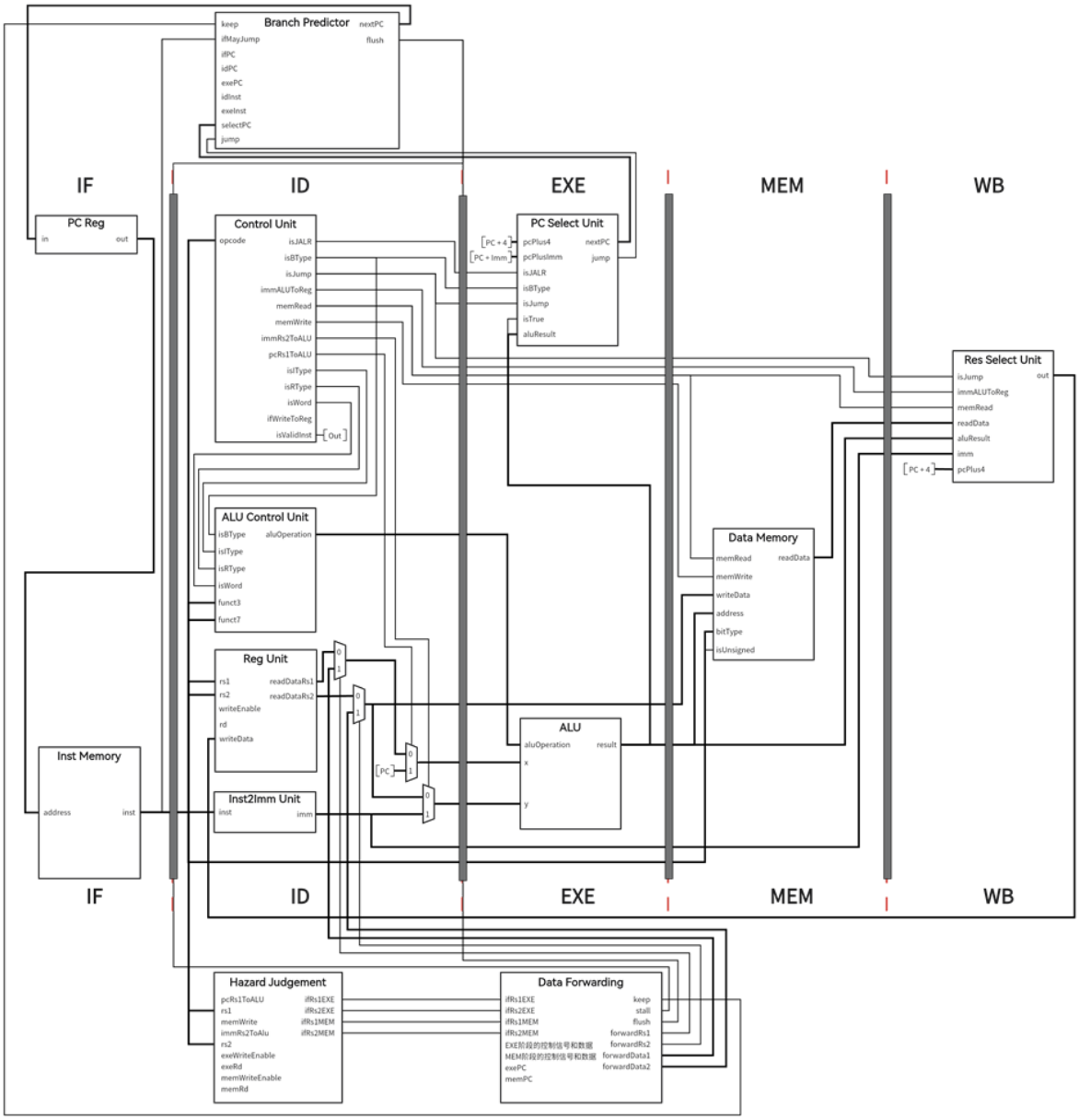
1. IF（Instruction Fetch）：取指令阶段，从指令内存中取出指令，并将程序计数器（PC）增加到下一条指令的地址。
2. ID（Instruction Decode）：译码阶段，将取出的指令译码为控制信号和操作数，并得到对应的ALU控制码和立即数。
3. EX（Execute）：执行阶段，执行对应的指令操作，例如算术运算、逻辑运算、比较运算等，并通过PC Select Unit获得下一个PC。
4. MEM（Memory Access）：访问内存阶段，访问数据内存或者指令内存，存数或取数，例如读取数据、写入数据等。
5. WB（Writeback）：写回阶段，选出正确的结果进行写回，例如将计算结果写入寄存器或者内存中。

在流水线CPU设计中，各个阶段可以并行执行，这样可以大大提高CPU的效率和吞吐量。同时，在各个阶段之间需要进行合理的数据传输和控制信号的传递，确保流水线的正确性和稳定性。

相比于单周期CPU，流水线CPU在提高了性能的同时也遇到了一定的困难，它需要解决流水线结构所带来的的三大冒险：结构冒险、控制冒险和数据冒险。为此，我学习了通用的处理方法，设计了一些组件用于处理数据的控制和流动，会在后续依次介绍如何解决三大冒险，以及解决该三大冒险时采用的不同的方案。

由于解决方案的不同，对应的流水线CPU的设计也不同，分为2个版本：气泡方案的版本、具备分支预测和旁路转发的版本。分别如下：


Branch Control Unit
keep
isJump
isBType
ifPC
idPC
exePC
selectPC
nextPC
flush
IF
ID
EXE
MEM
WB
PC Reg
in
out
Control Unit
opcode
isJALR
isBType
isJump
immALUToReg
memRead
memWrite
immRs2ToALU
pcRs1ToALU
isIType
isRType
isWord
ifWriteToReg
isValidInst
Out
PC Select Unit
PC + 4
PC + Imm
pcPlus4
pcPlusImm
isJALR
isBType
isJump
isTrue
aluResult
nextPC
Res Select Unit
isJump
immALUToReg
memRead
readData
aluResult
imm
pcPlus4
out
PC + 4
ALU Control Unit
isBType
isIType
isRType
isWord
funct3
funct7
aluOperation
Data Memory
memRead
memWrite
writeData
address
bitType
isUnsigned
readData
Reg Unit
rs1
rs2
writeEnable
rd
writeData
readDataRs1
readDataRs2
ALU
aluOperation
x
y
result
Inst Memory
address
inst
Inst2Imm Unit
inst
imm
PC
0
1
Data Control Unit
inWriteEnable
inRd
inData
pcRs1ToAlu
rs1
memWrite
immRs2ToAlu
rs2
exeWriteEnable
exeRd
memWriteEnable
memRd
exePC
memPC
wbPC
writeEnable
rd
data
keep
stall
flush



4.1 流水线

流水线的实现利用了Chisel所提供的面向对象的编程特性，创建了一个通用流水线寄存器，通过个性化地定制不同的IO端口，实现各个阶段之间不同的流水线寄存器。

流水线寄存器组中，正常情况下，数据从in输入，经过寄存器存储后，无任何改变地从out进行输出。控制信号为stall和flush，前者可以控制寄存器组是否保持原本的输出（即流水线当前阶段暂停/维持），后者可以控制寄存器组是否输出NOP（即冲刷流水线）。

通过应用不同的IO端口，实现不同阶段的流水线组的个性化定制和统一管理，来精确控制流水线的运行、暂停和冲刷。

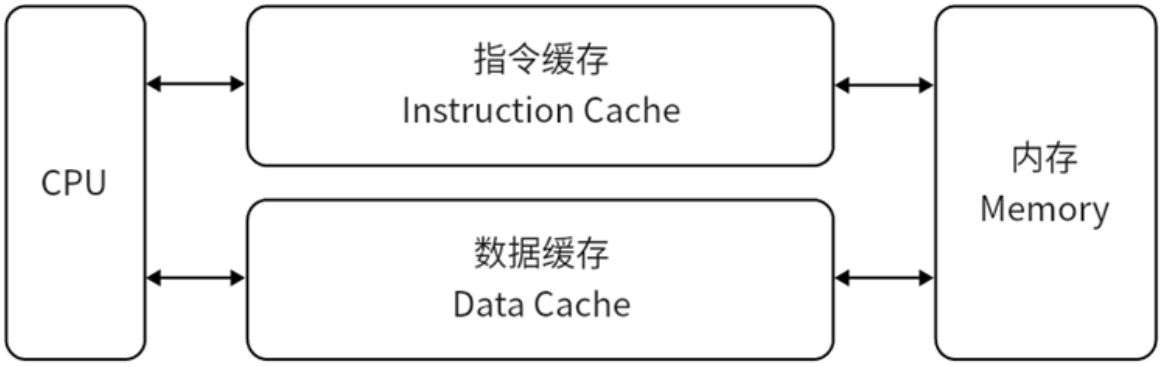
4.2 结构冒险

结构冒险是指当CPU在同一时钟周期内，同时运行两条指令的不同阶段时，这些指令可能会使用相同的硬件电路资源，从而导致冲突和错误。例如，在冯诺依曼结构中，MEM和IF在同一时钟周期内无法同时对内存进行读写，因为这需要使用相同的存储器总线。

为了避免结构冒险，一种解决方案是使用哈弗结构，即将数据存储和指令存储分开。在哈弗结构下，CPU可以在同一时钟周期内同时进行访存和取址操作，从而避免了结构冒险的问题。我设计的CPU正是使用了哈弗结构，将数据和指令分开存储：将存储单元分为2个，一个为指令存储单元，一个为数据存储单元。因此在硬件层面上，LESC-5 CPU不存在结构冒险。

然而，由于真实的存储单元是一体的，所以在实际中，很难将内存分为指令和数据部分并进行分别的大小设置。

因此，在现代的CPU中，往往采用混合结构，如下图所示，即通过使用指令缓存和数据缓存，将指令和数据分开存储，同时又不影响真实的内存存储的完整性。



在这种情况下，Cache的处理和策略的选择就显得尤为重要，这些将会在未来的工作中进行完善。

4.3 数据冒险

数据冒险是指在CPU的流水线中，由于数据的依赖关系而导致的指令执行结果错误等问题。

数据冒险分为RAW（先写后读）、WAW（先写后写）和WAR（先读后写），而由于所设计的CPU从硬件层面以及流水线结构上，不存在WAW和WAR的数据冒险问题，因此我们只需要考虑RAW的数据冒险问题。

具体来说，RAW是指当一条指令需要使用前一条指令的结果作为操作数时，如果前一条指令的结果还未来得及写回寄存器，那么后一条指令就无法正常执行，需要等待前一条指令的结果写回寄存器后才能继续执行。而这会带来流水线执行效率的降低，甚至带来错误的执行结果。

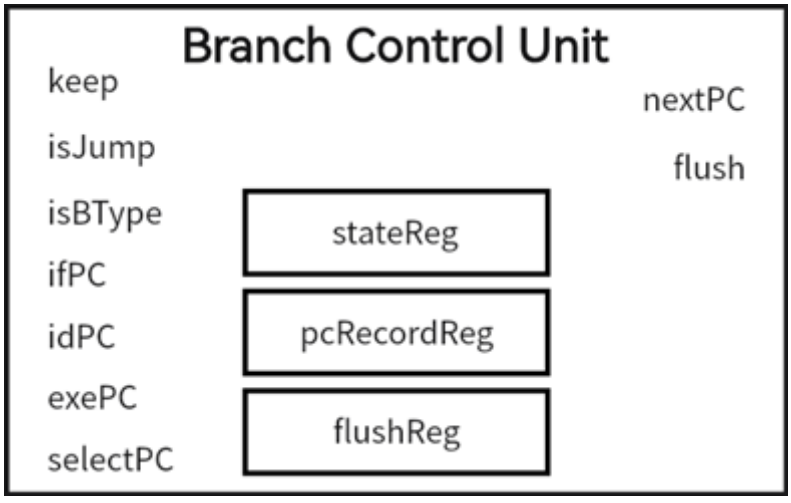
解决数据冒险的方法包括：气泡方案、旁路转发等。

在LESC-5 CPU中，我分别实现了气泡方案和旁路转发，前者实现在了Data Control Unit组件中（参考前面设计图中的Data Control Unit），后者则是通过Hazard Judgement组件和Data forwarding组件（参考前面设计图中的Hazard Judgement和Data forwarding）一起实现。

在之后的讨论中，产生数据的为源指令，使用数据的为目标指令，而气泡指NOP（即空操作，单个阶段的流水线不执行任何操作）。

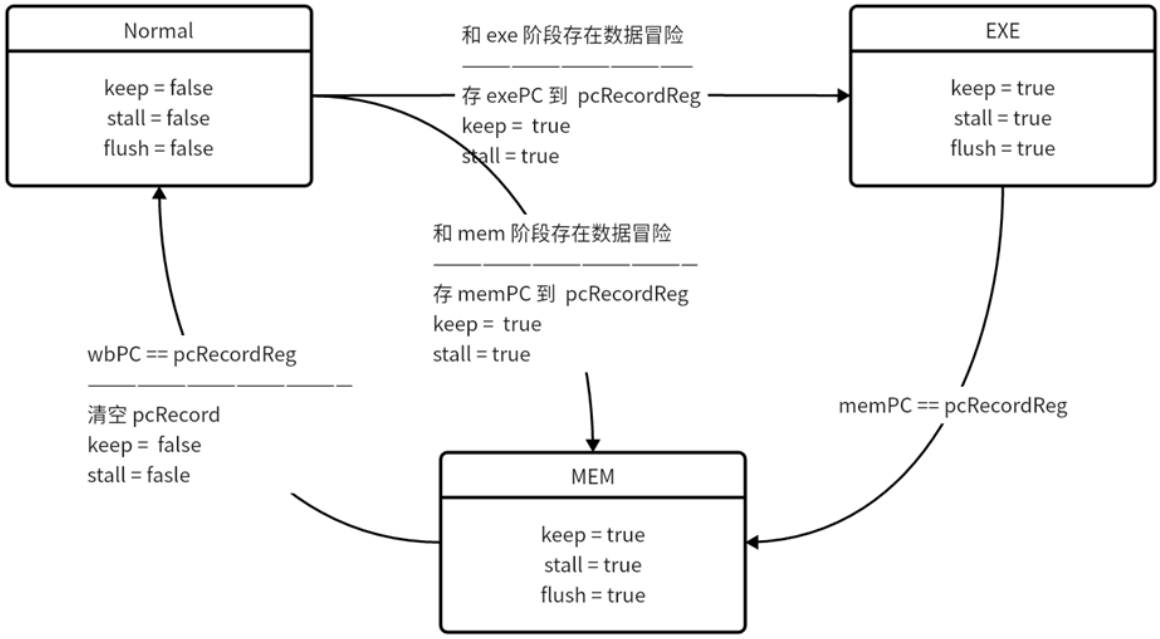
1. 气泡方案

是由Data Control Unit组件实现的，对应先前的设计图，下面是Data Control Unit的内部示意图：



通过在流水线中插入气泡（NOP），来起到让目标指令等待源指令执行的作用，通过牺牲性能来避免错误。

具体如下面的图所示，在ID阶段检测到RAW后，使目标指令保持在ID阶段不往前流动，通过插入气泡（NOP）让源指令继续流动，直至源指令到达WB阶段。
之后，使ID阶段的目标指令拿到WB阶段源指令生成的结果，流水线继续正常流动。



2. 旁路转发

是由Hazard Judgement组件和Data forwarding组件一起完成的，对应先前的设计图。

Hazard Judgement 负责判断是否发生数据冒险以及获得源指令所处的阶段；Data Forwarding 则根据 Hazard Judgement 传来的信息，对数据进行对应的旁路转发。

当检测到发生数据冒险后，根据冒险的情况不同（源指令的不同以及源指令所处阶段的不同），Hazard Judgement会发出不同的控制信号。而Data Forwarding则是根据这些不同的信号，通过配合控制ID阶段的Reg Unit数据输出的多路复用器，构建从各个阶段到目标指令的数据转发通道。

4.4 控制冒险

控制冒险是指在CPU的流水线中，由于指令的跳转而导致的指令执行错误等问题。

在需要执行跳转指令（如条件分支或无条件跳转）的情况下，理想情况下，处理器会跳转到程序中的目标位置执行下一条指令。但在流水线结构中，检测跳转在ID阶段，得到跳转目的地则在EXE阶段，而PC取址在IF阶段。在这种情况下，处理器可能会在跳转指令之前开始执行下一条指令，但是它不知道它是否应该跳转，因为跳转指令还没有执行完毕。

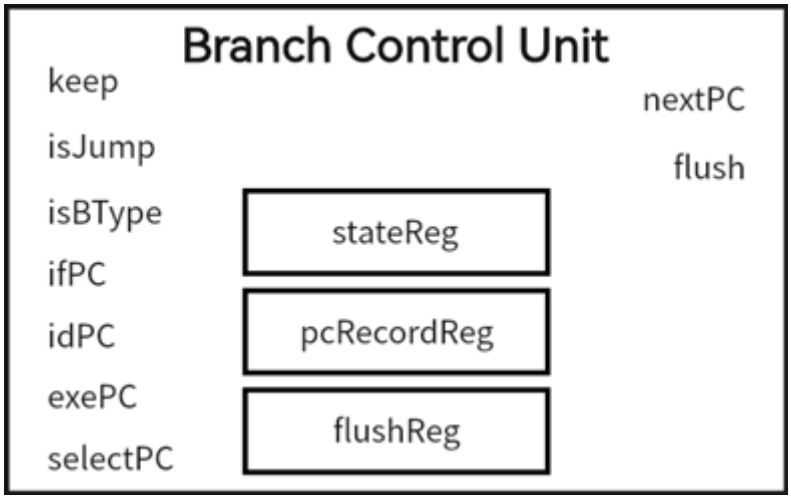
解决控制冒险的方法包括：气泡方案、分支预测等。

在LESC-5 CPU中，我分别实现了气泡方案和分支预测，前者实现在了Branch Control Unit组件中（参考前面设计图中的Branch Control Unit），后者则是实现在了Branch Predictor组件中（参考前面设计图中的Branch Predictor）。

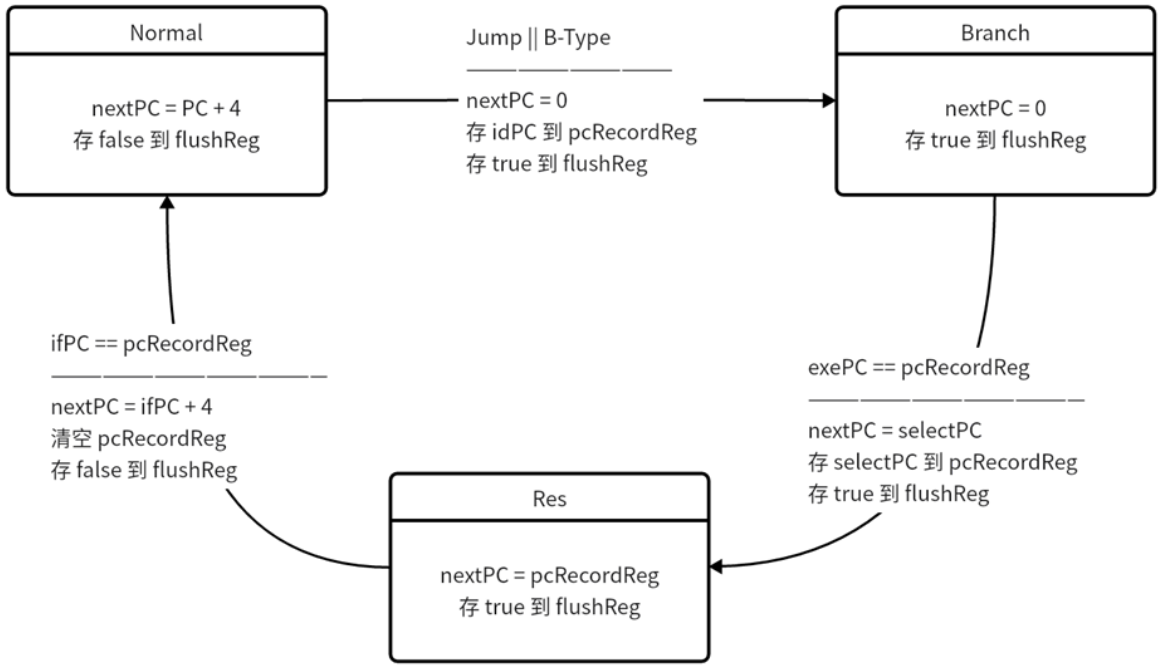
指令通常分为2种后续，不跳转和跳转。前者对应正常的依次执行，或者是条件跳转指令不生效，目标地址为PC+4；后者对应无条件跳转或条件跳转指令生效，我们把对应的目标地址叫为Target。

1. 气泡方案

是由Branch Control Unit组件完成，参考前面的设计图，下面是Branch Control Unit的内部示意图：



具体如下图所示，在ID阶段检测到当前指令为跳转指令后，对IF、ID插入气泡（NOP），当跳转指令执行到EXE，得到目的地指令的地址后，再重新启动流水线，执行目的地指令。



2. 分支预测

分支预测是一种在处理器中使用的技术，以尝试减少控制冒险的发生。

静态分支预测是最简单的形式，它基于指令的历史记录和模式，预测下一个指令是跳转还是不跳转。具体来说，静态分支预测器根据以前的跳转记录，预测跳转指令是否会跳转，如果预测错误，则直接冲刷流水线并重新执行指令流。

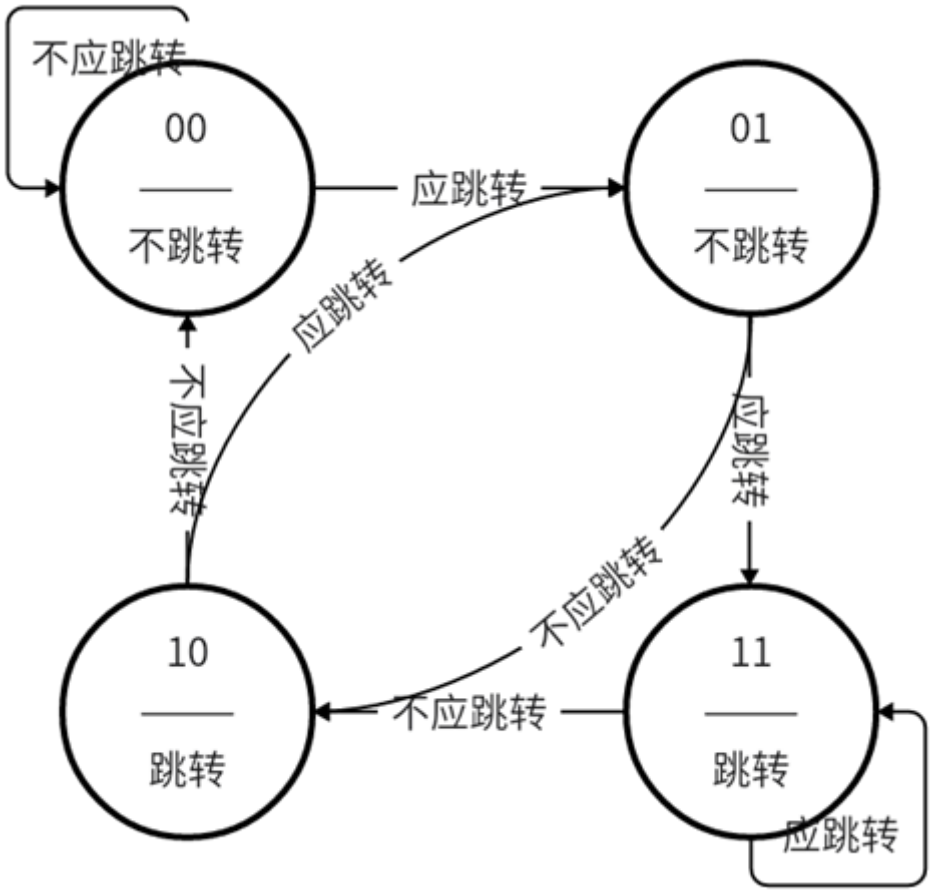
相比之下，动态分支预测器会在执行阶段（EXE阶段）记录跳转的历史信息，并在下一次指令读取阶段（IF阶段）使用这些信息来预测跳转指令的行为。

动态分支预测器通常使用n位记录器来记录历史跳转信息，其中n是预测器的位数。每次跳转指令执行时，记录器的值会被更新。例如，如果跳转指令执行时跳转发生，则将记录器内的额值左移1位，并在末尾补1；若不跳转，则左移1位后末尾补0。

在下一次指令读取阶段，动态分支预测器使用计数器的最高位来预测跳转行为。如果最高位为1，则预测跳转指令会跳转，否则预测跳转指令不会跳转。

如果预测错误，则需要冲刷流水线并重新执行指令流。

在动态分支预测中，应用较为广泛且有效的是2-bit预测器。在记录时，其按照FIFO，尾部插入1（跳转）或0（不跳转），其余数据全部前移，首位弹出。在预测时，同样是最高位决定是否跳转。2-bit预测器对应的状态转换图如下所示：



在LESC-5 CPU中，分支预测是由Branch Predictor组件完成，参考前面的设计图。此外，相较于之前，Branch Predictor 会直接负责生成PC，进行分支预测，不再有额外的其他生成PC的组件。

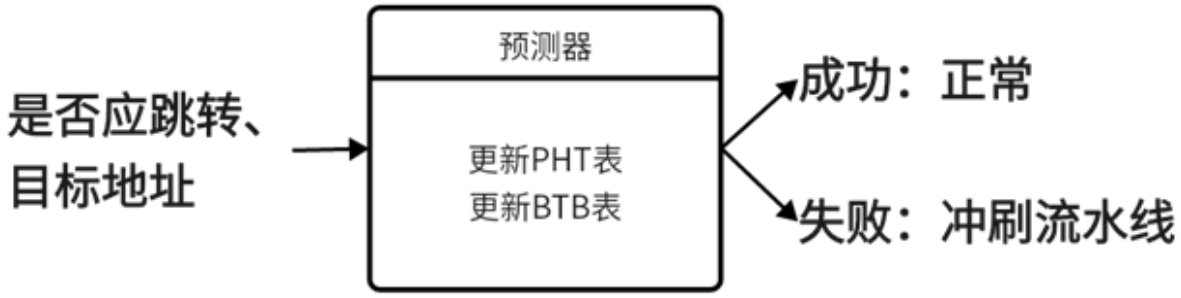
Branch Predictor主要由PHT表（Pattern History Table，模式历史记录表）和BTB表（Branch Target Buffer，分支目标缓冲区）组成，进行 2-bit 的模式匹配。这两者都是CPU中用于支持动态分支预测的重要数据结构。

PHT表（Pattern History Table）是一个记录历史分支模式的表，用于记录程序中的分支指令的历史跳转情况。PHT表的每个条目包含一个计数器，该计数器记录了特定分支模式的历史跳转信息。例如，在一个2位的PHT表中，对于每个可能的2位历史分支模式，都会有一个计数器来记录该模式的历史跳转情况。当处理器执行分支指令时，它会查找PHT表中与该分支指令历史分支模式匹配的计数器，并根据计数器的值来预测分支指令的执行情况。

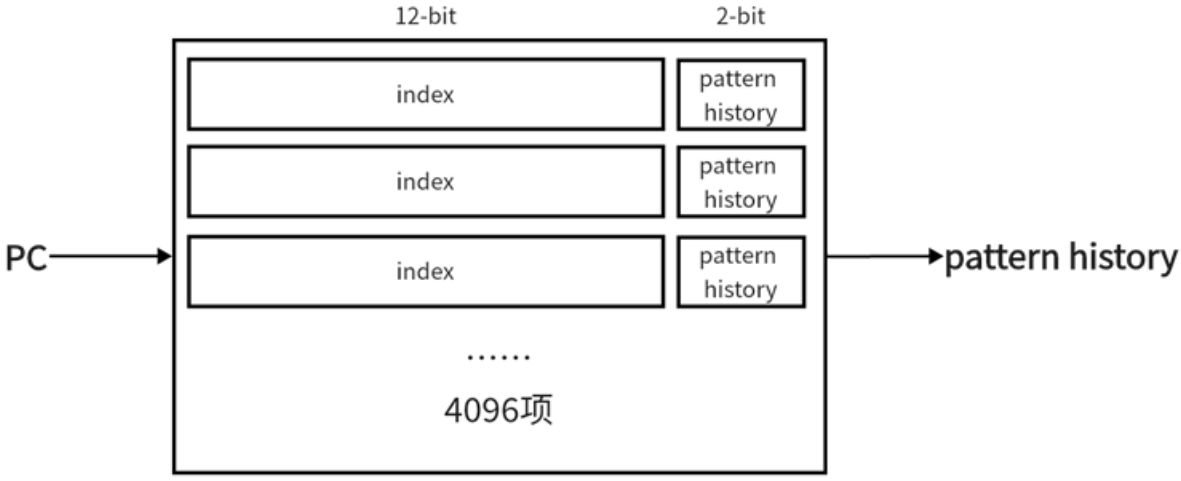
而BTB表（Branch Target Buffer）则是一个缓存分支目标地址的表，用于记录程序中的分支指令的目标地址。BTB表的每个条目包含两个字段：一个是分支指令的地址，另一个是分支指令的目标地址。当处理器执行分支指令时，它会查找BTB表中与该分支指令地址匹配的条目，并获取该条目中存储的目标地址。如果分支指令的执行情况需要跳转到目标地址，则处理器可以直接使用BTB表中存储的目标地址，而不必等待目标地址从内存中读取。其类似于一个Cache，对目标地址进行存储，这可以提高程序的执行效率。

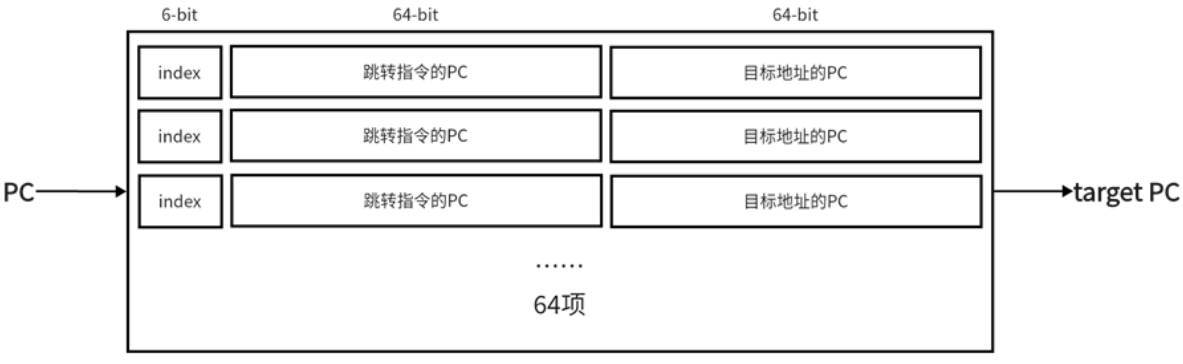
Branch Predictor的工作流程如下图所示，根据输入的PC，直接预测下一个PC并输出到PC Reg中，直接负责流水线的指令执行控制。

当跳转指令执行到EXE阶段，即获取到真正的Target的时候，预测器会获取这些信息，并根据这些信息更新PHT表和BTB表，同时会判断是否预测成功，如下图所示。若成功，则一切正常，流水线正常运行；若失败，则冲刷流水线。因此，预测器对下一个PC的预测成功与否就显得尤为重要。



在预测器中，PHT表和BTB表对应的结构如下图所示。PHT表的索引为12bit，对应PC地址的低12位，pattern history条目采用2bit，总共4096项；BTB表的索引则为6bit，条目为128bit，高64位为跳转指令的PC，低64位为目标地址的PC，共64项。





5 Utils部分和调试信息

为了让CPU的代码可读性更高，同时增加CPU的易学性和可扩展性，通过利用Chisel语言高层次抽象的优秀语言特性，我将CPU中反复会用到的工具函数等进行封装和统一化，形成Utils工具包和一整套调试信息输出体系。

5.1 Utils部分

Utils部分写在CPUUtils.scala中，主要包含了一些工具函数和一个打印CPU调试信息的工具集合。

1. 工具函数

工具函数部分包括：

- 符号/无符号扩展函数集合：提供分别可以将8位、16位、32位的整数进行符号扩展或是无符号扩展为64位的整数的函数，以及一个用于将任意位数的整数进行符号/无符号扩展的原型函数。
- 补齐函数：将字符串前根据对应参数补若干个0，使其长度校准至目标长度。
- 加载文件相关的函数：获取文件路径和文件名，以及将单个hex文件分离成多个hex文件（用于适应以向量为存储单元的内存）的函数。

2. 调试信息输出

调试信息输出相关的工具函数集合写在名为CPUPrintf的单例对象里，包含对各类情况的调试信息的打印，具体如下：

- `printfIO` 函数，可以根据设置打印元件的整个IO端口，函数的参数包括打印前缀、模块名称、数据列表和需要打印的数据类型等。
- `printfIOArgln` 函数，可以根据设置打印单个端口，并在打印后换行。
- `printfIOArg` 函数，可以根据设置打印单个端口，但不可换行。

这些函数实现了根据设置打印不同类型数据的功能，可以根据需要自动识别并打印二进制、十进制、十六进制、布尔值等不同类型的数据，同时还提供了一些格式化处理（例如增加前缀），使得输出更加易读，并方便了调试。

其他自定义的函数也可以在此处定义，可供CPU统一调用，便于个性化定制。

5.2 调试信息

为了提高LESC-5 CPU的易学性，并降低对CPU进行扩展和定制的难度，我在提供了Utils部分中的整套调试信息函数外，针对各个组件和整体CPU的运行和测试，预设了丰富的调试信息的输出语句。控制调试信息的输出与否的开关统一放在Config文件中，便于统一管理和个性化选择。

1. 组件部分

在每一个组件的实现代码的尾部，我均插入了以统一格式为主体的调试信息输出代码（部分组件有个性化的调试信息输出代码）。

通过用不同的List容器，可以个性化地控制对端口或信号的输出及其输出格式：needBinary、needDec、needHex和needBool分别表示需要用二进制、十进制、十六进制和布尔类型方式进行输出的端口集合；而needDelete和needAdd则可以用于在默认的IO端口输出的基础上，删减和增加个性化的端口、信号进行输出。

由于调试信息代码的格式统一，因此很易于进行统一控制、修改和复刻。

2. CPU整体部分

在CPU整体部分，我按照统一的打印信息风格分别在CPU内部和测试器都设计了一套调试信息输出的代码。其控制信号依旧放在Config文件中，用于统一管理。

调试信息仿照日志表单的形式，对CPU的各个关键端口、信号进行格式化输出，格式美观优雅且直观，极大地降低了学习和调试LESC-5 CPU的难度。使用者或开发者可以像调试软件一样查看日志输出，以可视化的形式详细地了解CPU在每个周期各部分的运行情况，降低了扩展和个性化的门槛。