信号说明ALL IN ONE

注：所有流水段寄存器有clk, reset, cu\_flush, cu\_stall输入，下略。

# ifid流水段寄存器

输入为32位指令，指令PC和指令PC+4。输出为各种指令格式对应字段的信号。

（注：可以直接在内部完成J指令跳转地址的拼接）

# idex流水段寄存器

一般地，我们认为译码段不会成为最长的一个流水段，所以我们考虑尽可能在译码段多完成一些工作。所以我们将寻址方式比较特殊的指令在ID段进行处理，使他们在以后的流水段中表现出的行为与其他指令一致。由于流水段寄存器中所存的信号大部分来自译码部件，但是其中大部分都不会在译码段使用，所以下文分别说明译码段所使用的信号和为后面的周期准备的信号。

## 本流水段使用的信号

### id\_rt\_zero\_sel

此信号用于选择通用寄存器组的Rt\_addr的源：因为bgez等指令需要用0与Rs中的值进行比较，但是这类指令中的一部分的Rt字段不全为0，所以我们需要手动修改输入到ALU中的op\_B的数据。

在流水线中我们还需要进行冒险检测，并且冒险检测最好在译码阶段完成，我们需要将Rt的地址送到控制单元进行分析。然而这类指令的Rt\_addr字段实际上都不是标注源操作数用的，它们的Rt\_addr字段不应该造成冒险，如果将非0的Rt\_addr送到控制单元，控制单元还得为这类指令开出特例才能保证不会“误判”。为了减轻控制单元的负担，我们在译码阶段确定该分支指令的Rt字段是否有（作为寄存器地址的）意义，如果没有，则把0送给通用寄存器组和控制单元。

### id\_cp0\_src\_addr [3:0]

mfc0指令的源操作数地址。

### id\_rt\_data\_sel

mfc0指令的源操作数来自cp0，而其他指令的源操作数来自寄存器组，此信号用于在取自cp0的数据和取自GPR中的数据间进行选择。如果该指令是mfc0则选择来自cp0的数据。

### id\_jump

jump指令跳转所需要的信息在ID段完全可以获得，所以jump指令在ID段生效，在ID段将跳转信息送到控制单元。

id\_eret和id\_syscall与jump类似

### id\_jr

jump register指令使用GPR[Rs]的值作为下一条指令的PC。通知控制单元改变pc\_src。

### id\_imm\_ex [1:0]

选择立即数的扩展方式以及为lui指令开的特例。

### id\_ctrl

此信号线标注的宽度为1，实际上它的意义是在ID段未使用到的控制信号，直接送入到idex寄存器中。我考虑到画图时idex寄存器的输出肯定会标注出所有的控制信号，如果在输入处也标注一次，显得比较冗余，所以我选择只画了一根线，但是实现的时候还是应该按照指令要求完成。关于id\_ctrl中包含的信号，见下文。

（注：在编写代码时，拟将这些信号显式地声明为指令译码器（ID）的输出端口和IDEX流水段寄存器的输入端口）

### id\_rd\_addr\_sel

rd的地址有三种可能:

1. 指令中的Rt字段（mfc0包含在这里）
2. Rd字段
3. 5’b11111（jal等指令会将地址写入31号寄存器）

## 其它周期使用的信号

### 控制信号

这些控制信号用图中用id\_ctrl表示，从译码器传送到idex寄存器，具体包含的控制信息在idex寄存器的输出中可以看到。

#### idex\_mem\_w

表示该指令写存储器。

#### idex\_mem\_r

表示该指令读存储器（避免触发不必要的cache miss）。

#### idex\_reg\_w

表示该指令写寄存器，主要是为冒险检测提供方便。

#### idex\_branch

传送到exmem寄存器，用于分支判断。

#### idex\_condition [2:0]

表示该分支指令所使用的条件。

#### idex\_of\_w\_disen

表示该指令在overflow时，不会写入寄存器，主要针对add等不允许溢出的算术指令。

#### idex\_exres\_sel [1:0]

选择将ALU或移位器的结果传到exmem寄存器（此前图上还花了一个pc+4作为选择器输入端，似乎没必要？）（看起来是有必要的，不然哪里让。

#### idex\_B\_sel

选择ALU的B操作数的来源：Rt或是经过扩展的立即数。

#### idex\_rs\_addr、idex\_rt\_addr、idex\_rd\_addr

寄存器地址，用作转发和冒险检测（仅Rd）判断依据。

#### idex\_load\_sel [2:0]

用于表示这是load系列指令中的哪一条，用于字节写使能信号的生成和数据移位。

#### idex\_store\_sel [2:0]

用于表示这是store系列指令中的哪一条，用于字节写使能信号的生成和数据移位。

#### idex\_cp0\_dst\_addr [4:0]

写入cp0的目的寄存器地址

#### idex\_cp0\_w\_en

cp0写使能，由mtc0产生

#### idex\_syscall

系统调用指示信号，由syscall产生

#### 与同单周期下含义相同的指令

idex\_alu\_op、idex\_shamt\_sel、idex\_shift\_op、idex\_shamt

### 数据

#### idex\_op\_A，idex\_op\_B

可能送往ALU作为A B操作数的数据（这里其实应该用Rd和Rt命名）

#### idex\_pc

当前指令的pc

#### idex\_pc\_4

当前pc+4

#### idex\_imm\_ext

经过扩展的立即数

# exmem流水段寄存器

EX段完成的主要功能与单周期cpu的功能类似，详细说明略，EX段所需要的控制信号在idex段寄存器部分说明，EX段产生的新数据和控制信号在此说明。

## 输入

### 控制信号

#### dex\_mem\_r、idex\_mem\_w、idex\_load\_sel、idex\_branch、idex\_condition

继承自idex寄存器的信号EX段产生的控制信号。

#### alu\_lf, alu\_zf, alu\_of

来自ALU的标志信息。

#### real\_rd\_addr

经过选择后的目的寄存器地址。

#### reg\_byte\_w\_en

字节写使能生成模块得到的寄存器写使能信号，如果of和idex\_of\_disen都有效，则全部置为0

#### mem\_byte\_w\_en

字节写使能生成模块得到的存储器写使能信号

### 数据

#### idex\_pc, idex\_pc\_4

继承自idex寄存器的数据。

#### adder\_target

来自加法器，分支指令目标地址。

#### ex\_res

ALU或者桶形移位器的结果。

#### aligned\_rt\_data

store移位器生成的经过对齐的数据。

## 输出

### 送入data\_mem\_interface的信号

exmem\_mem\_w、exmem\_mem\_r、alu\_res（作为地址）、rt\_data（作为写入数据）、mem\_byte\_w\_en（字节写使能）

### 送入控制单元的信号

exmem\_pc（和final\_target送入控制单元，控制单元根据当前EX段的指令的pc判断是否预测正确，对错误进行矫正，并且反馈到分支预测器。）

### 送入分支地址计算模块的信号

exmem\_branch、exmem\_condition、exmem\_target、exmem\_pc\_4、exmem\_lf、exmem\_zf（详见该模块说明）

### 送入load移位器的信号

alu\_res低2位，exmem\_load\_sel（选择移位、扩展方式，详见该模块文档）

# memwb流水段寄存器

## 输入

### exmem\_mem\_r

继承自exmem，用于选择写回到寄存器的数据：从存储器中取得的数据（load系列指令） 或 EX段得到的数据（其它指令）。

### exmem\_reg\_w

继承自exmem，寄存器总写使能，一些不允许溢出的运算指令需要，可以用它来避免不必要的冒险（？）

### reg\_byte\_w\_en [3:0]

继承自exmem，寄存器字节写使能。

### exmem\_rd\_addr [4:0]

继承自exmem，目的寄存器地址（也可用于mtc0的目的寄存器，图上还没画相关电路和mtc0的sel信号；也还没有画要传递的异常信号）。

### mem\_data [31:0]

mem段产生，来自存储器的数据。

### ex\_data [31:0]

继承自exmem，来自EX段的数据。

### exmem\_cp0\_dst\_addr [4:0]

继承自exmem，要写入的cp0寄存器的地址。

### exmem\_cp0\_w\_en

继承自exmem，cp0寄存器写使能

### aligned\_rt\_data [31:0]

继承自exmem，本质上就是GPR[Rt]的数据，对特殊的load指令进行了移位处理。

## 输出

与输入相同，exmem前缀换成memwb前缀。

### 送入cp0的信号

memwb\_cp0\_dst\_addr[4:0], exmem\_cp0\_w\_en, aligned\_rt\_data[31:0]

### 送入寄存器堆和转发单元的信号

memwb\_mem\_r选择memwb\_memdata[31:0]和memwb\_exdata[31:0]的输出, mem\_reg\_w(不用送转发), reg\_byte\_w\_en[3:0], memwb\_rd\_addr[4:0]

# 指令译码器ID

ID前缀的输出信号只在ID段使用，IDEX前缀的输出信号送入IDEX流水段寄存器。

这里只列出了指令译码器本身需要输出的控制信号，其他如寄存器地址的选择放在id段指令译码器的外部完成。

## 输入

### ifid\_instr [31:0]

直接将整个指令送入进行译码

## 输出

### id\_cp0\_src\_addr [3:0]

mfc0索引cp0寄存器组的编号，来自rd

### id\_rt\_data\_sel

选择rt源操作数来源：gpr, cp0

### id\_jump

绝对跳转指示信号

### id\_imm\_ext [1:0]

符号扩展指示信号:

2‘b00: 无符号扩展

2’b01: 符号扩展

2’b10: lui扩展

2’b11: 符号扩展单元输出常数4，这个数字将来在EX段会计算出跳转地址PC+8，这是JAL需要的返回地址。当指令为JAL时，id\_imm\_ext==2’b11，同时必有idex\_exres\_sel==2’b10.

### id\_rt\_addr\_sel

强制rt为$zero

### id\_rd\_addr\_sel [1:0]

实际的目标地址可能是rt, rd和$31

### id\_cp0\_dst\_addr [3:0]

mtc0索引cp0寄存器组的编号，来自rd

### idex\_cp0\_w\_en

mtc0写使能

### idex\_eret

异常返回指示信号

### idex\_syscall

系统调用指示信号

### idex\_mem\_w

主存写使能

### idex\_mem\_r

主存读指示信号

### idex\_reg\_w [3:0]

寄存器写使能

### idex\_branch

分支跳转指示信号

### idex\_condition [3:0]

条件判断方式选择信号

### idex\_of\_w\_disen

溢出不可写信号

### idex\_exres\_sel [1:0]

运算结果选择(alu和移位器)

0 – alu

1 – shifter

2 – PC+8

### idex\_alu\_op [3:0]

同单周期

### idex\_b\_sel

选择寄存器操作数和立即数

### idex\_shamt\_sel

同单周期

### idex\_shift\_op [1:0]

同单周期

### idex\_shamt [4:0]

同单周期

### idex\_rs\_addr [4:0]

转发判断依据

### idex\_rt\_addr [4:0]

转发判断依据，可能的目标地址

### idex\_rd\_addr [4:0]

可能的目标地址

### idex\_load\_sel [2:0]

load类型选择，根据地址，不同的使能

### idex\_store\_sel [2:0]

store类型选择，根据地址，不同的使能

# 转发单元FU

转发的来源有exmem流水段寄存器的alu运算数据和memwb流水段寄存器经过mem\_read选择后的写回数据，转发目的地是ex段alu要使用的rs和rt的寄存器操作数（其中rt的经过转发后的数据还需要进一步和立即数进行选择）。

由于lwl, lwr等类型的指令写使能可能不是全有效的，所以wb段的写回数据，只是部分改写目标寄存器的数据。为了保证转发的正确性，需要将寄存器数据送入转发单元，由转发单元进行一次改写。

mem段只用转发alu的计算结果，因为load-use冒险无法通过转发解决，我们在id段取操作数时检测出load-use冒险，此时访存指令执行到了ex段，正在计算地址，然后我们阻塞前半段流水线，在下个周期的ex段插入气泡，此时访存指令执行到mem段，之后流水线正常执行，访存指令已经执行到wb段，而需要写回数据的指令刚好执行到ex段（原来阻塞在id段），所以可以通过转发wb段来解决冒险。鉴于上诉情况，mem段只需要转发运算器的结果，而且没有写使能不全有效的情况。当然写使能还是要送入，因为要通过写使能全无效来决策是否可以转发。

转发单元需要知道mem段和wb段的写使能和目标寄存器地址。由于mem段数据比wb段的更新，所以优先转发mem段的数据。决定转发来源的规则如下：

对rs寄存器，如果mem段的目标寄存器即rs且写使能全有效，则选择mem段的转发数据（输出对应的选择信号）；如果mem段不满足要求，而wb段的目标寄存器即rs且写使能不全无效，则选择经过转发单元用原始数据根据写使能改写过的wb段的转发数据；如果都不满足要求，则不转发。对rt寄存器采取相同的操作。

## 输入

### rs\_data [31:0]

原始的来自rs的数据，保证写使能无效字节的正确性

### rt\_data [31:0]

原始的来自rt的数据，保证写使能无效字节的正确性

### rs\_addr [4:0]

rs地址，由于检查写回寄存器是否被使用

### rt\_addr [4:0]

rt地址，用于检查写回寄存器是否被使用

### exmem\_w\_b\_en [3:0]

mem段回写寄存器的字节写使能，主要用于检查不允许写的情况

### exmem\_rd\_addr [4:0]

mem段回写寄存器的地址

### memwb\_w\_b\_en [3:0]

wb段回写寄存器的字节写使能，用于检查不允许写和部分写的情况

### memwb\_rd\_addr [4:0]

wb段回写寄存器的地址

### memwb\_data [31:0]

wb段回写寄存器的数据，用于和rs\_data和/或rt\_data依据memwb\_w\_b\_en做组合

## 输出

### input\_a [31:0]

wb段的转发数据与rs原始数据经过处理后的值

### input\_b [31:0]

wb段的转发数据与rt原始数据经过处理后的值

### a\_sel [2:0]

### b\_sel [2:0]

选择转发操作数：

0 – 不转发，使用原始数据

1 – 转发mem段数据

2 – 转发wb段数据

# 控制单元CU

控制单元检测流水按的冒险，控制流水段寄存器的冲刷和阻塞以及异常处理

检测load-use冒险：

根据输入的id段得到的rs寄存器地址和最终的rt寄存器地址，以及ex段继承自上一流水段的rd寄存器地址和mem\_read访存信号，如果ex段是访存指令（mem\_read有效），且rd与rs和rt中的一个相等，则通过组合逻辑使得pc\_stall, ifid\_stall, idex\_flush有效，在下一个周期在ex段插入气泡，而if段和id段阻塞，mem段和wb段正常执行。

检测控制冒险：

分支指令在mem段算出跳转条件是否成立，比较mem段分支地址单元提供的正确下地址和ex段给出的预测器给出的预测下地址，如果不一样，使得ifid\_flush, idex\_flush, exmem\_flush有效，同时通过组合逻辑使得pc\_src选择mem段的正确的跳转地址作为下个周期写入pc的值，并且使得bpu\_w\_en有效，送入mem段的当前pc以及正确的跳转地址，通知分支预测单元进行更新。注意由于分支预测器可能允许非跳转指令误撞，所以cu每个周期都在进行有意义的分支冒险检测。

绝对跳转指令和过程调用指令在id段能判断出来，根据输入的指示信号，将ifid\_flush有效，pc\_src改为选择绝对跳转地址。

分支和绝对跳转，分支和load-use的冒险可能同时检测到，由于此时分支指令之后的指令都是错的，所以最优先生成分支指令发生冒险时的信号，而绝对跳转和load-use都需要id段的信息，所以不可能同时发生。

检测syscall：

id段生成的syscall信号，但是此时id段可能位于错误分支中。所以要将该信号传递到mem段再触发，因为如果能成功传递到exmem流水段寄存器，说明这个syscall是正确的要执行的指令。冲刷和阻塞信号同分支指令冒险的情况。输出对应的中断向量地址，pc\_src改成选择中断向量。cu\_cp0\_w\_en有效，cu\_exec\_code输出syscall对应的8，cu\_epc\_src选择ex段的pc。不会与分支跳转冒险同时发生。

检测eret/ret：

id段判明，id段触发，效果同绝对跳转，不过pc\_src改成epc/$31。可能和分支跳转冒险同时发生，分支跳转冒险的信号优先输出。

检测外部中断：

外部中断信号从cp0传入，判明发生中断后，根据当前的分支冒险检测结果让pc\_src选择ex段的pc还是正确跳转地址。

中断向量由cu计算，其他的如跳转地址可以直接连线过去。

## 输入

### mem\_stall

由存储器送入的阻塞信号，cu负责进一步对流水段寄存器生成阻塞信号

### ifid\_rs\_addr [4:0]

从id段送入的rs的地址，用于检查和ex段的数据依赖

### real\_rt\_addr [4:0]

从id段送入的经过选择的rt的地址，用于检查和ex段的数据依赖

### idex\_rd\_addr [4:0]

从ex段送入的经过选择的真正的目标寄存器地址，用于检查和id段的数据依赖

### idex\_mem\_read

从ex段送入的读内存指示，在其有效的情况下数据冒险结果才有效

只需要检查load-use冒险

### predicted\_idex\_pc [31:0]

从ex段送入的pc地址，在mem段是分支跳转指令的情况下，此pc是分支预测器给出的预测地址，用于检查控制冒险

### target\_exmem\_pc [31:0]

从mem段送入的计算偏移量并且经过条件选择的正确跳转地址，用于检查控制冒险

### cp0\_intr

由cp0的cause寄存器送入的中断组合信号，是8个中断比特的或

用于指示cu改变pc\_src，提供中断向量地址，情况保存epc并冲刷流水段寄存器

### id\_jmp

由id送入的绝对跳转指示信号（包括j和jal），指示cu改变pc\_src并准备冲刷ifid流水段寄存器

### id\_eret

由id送入的异常返回信号，指示cu改变pc\_src，冲刷ifid流水段寄存器

### exmem\_syscall

由id生成，传送到exmem的系统调用指示信号，指示cu改变pc\_src，提供中断向量地址，视情况保存epc并冲刷流水段寄存器

## 输出

### cu\_pc\_src [3:0]

选择pc下址：

0 – 绝对跳转地址

1 – 寄存器跳转地址

2 – 中断向量

3 – epc

4 – 正确跳转地址

5 – bpu预测结果（含正常+4）

### cu\_pc\_stall

### cu\_ifid\_stall

### cu\_idex\_stall

### cu\_exmem\_stall

### cu\_wb\_stall

各寄存器阻塞信号

### cu\_ifid\_flush

### cu\_idex\_flush

### cu\_exmem\_flush

各流水段寄存器冲刷信号

### cu\_cp0\_w\_en

cp0写使能，写入cause，关闭中断(移动status)，发生异常时有效

### cu\_exec\_code [4:0]

写入cause的原因：

0 – 外部中断

8 – 系统调用

### cu\_epc\_sel [1:0]

选择epc的来源:

|  |
| --- |
| if exmem.branch = true and cu.predict = true  epc <- exmem.target  else if exmmem.jmp = true  epc <- ifid.pc  else  epc <- idex.pc |

### cu\_vector [31:0]

中断向量地址，不同的中断类型可能不同。

### cu\_bpu\_w\_en

bpu写使能，发生控制冒险时有效，用于更新bpu

# 分支预测单元BPU

暂时的设计是if段的pc传到current\_pc计算正常执行的下一条地址，在缺失的情况下自动加4

由于分支指令的顺序执行地址是pc+8（越过延迟槽的nop），所以可以考虑根据冒险检测结果存储循序地址还是跳转地址。

分支预测存在由于由于误撞而取出非pc+4地址的情况，所以需要分支地址计算单元在正常情况下返回pc+4而不是pc+8或target\_pc，并且分支冒险的检测会常态化。

## 输入

### current\_pc [31:0]

用来查询预测下地址的pc，来自if段

### tag\_pc [31:0]

用来做标签的pc，是发生控制冒险的周期的执行到mem段的分支指令的pc

### target\_pc [31:0]

被记录的目标跳转地址，是发生控制冒险的周期的执行到mem段的分支指令计算出的跳转地址

### bpu\_write\_en

bpu更新使能。

## 输出

### predicted\_pc [31:0]

预测器输出的pc下地址，pc的来源之一。

# cp0

异常处理单元，主要是含32个寄存器的寄存器组。需要一定的组合逻辑来控制数据写入。

id\_mfc0是用来选择写回数据的。

## 输入

### interruption [7:0]

外部中断，此接口暴露给外设。

### cu\_cp0\_w\_en

cu的异常写使能，并且用来分辨写来源。

### cu\_exec\_code [4:0]

来自cu的异常号。异常号的写入与mtc0可能会产生冒险，可以根据使能判断决策如何写入cause寄存器。

### epc [31:0]

记录中断返回地址，外部是一个选择器，选择不同的pc来源

### id\_cp0\_src\_addr [4:0]

cp0源寄存器地址，id译码出来直接送入，读出来的数据在id段使用。

### wb\_cp0\_w\_en

mtc0写使能，由id段产生，memwb流水段寄存器送入。

### wb\_cp0\_dst\_addr [4:0]

cp0寄存器地址，memwb流水段寄存器送入，mtc0使用。

### wb\_cp0\_data\_in [31:0]

来自ex段的寄存器数据，由memwb流水段寄存器给出，是即将要写入cp0[wb\_cp0\_dst\_addr]的数据。

## 输出

### cp0\_data [31:0]

mfc0需要的数据

### cp0\_epc [31:0]

中断返回地址，epc寄存器的值。

### cp0\_intr

组合逻辑，中断信号。

# 存储器

在设计图中，设计了IM和DM的接口，实现的时候可以考虑使用不同的存储元件，也可以采用相同的存储元件。前者更简单，后者容易引起阻塞，但后者可以更方便地使用loader。如果将IM和DM合并，则最好加上cache。另外，总线的IO也是通过DM接口来实现的：将一些地址映射到总线上，用于与外设进行交互。

## 指令存储器

### 输入

#### clk

存储器时钟

#### addr [29:0]

指令地址

### 输出

#### instr [31:0]

32位指令

#### mem\_stall

存储器无法在一个周期内完成取指，用此信号告知控制单元：流水线应该阻塞。

## 数据存储器

### 输入

#### clk

存储器时钟

#### read

表示本周期要取数据，设置这个信号的原因是并不是每个周期都要从存储器取数，为了防止错误的地址导致不必要的cache miss，甚至fatal

#### write

表示本周期要向存储器写入数据。

#### addr [29:0]

存取数据的地址，某一确定范围内的地址代表的是总线，另一部分是显存，剩下的则是内存。宽度为30位，总是按照4字节对齐

#### data\_in [31:0]

本周期要写入存储器的数据

#### mem\_byte\_w\_en [3:0]

决定4个字节的数据是否写入到存储器。

### 输出

#### data\_out [31:0]

读出存储器的数据

#### mem\_stall

存储器无法在一个周期内完成数据的存取。一般的，总线读写操作会引发阻塞

# 分支地址计算模块

在我们的CPU设计中，为了使流水线更加均匀，我们选择用4个周期完成branch指令，并且通过分支预测来降低控制冒险带来的cost，在EX段计算出分支目标地址（加法器）和分支条件标志位（ALU），在MEM段得到最终的目标地址。在MEM段的一个分支处理单元负责选择最终的目标地址。

## 输入

### exmem\_branch

是否为分支指令

### exmem\_condition [2:0]

分支条件

### exmem\_target [31:0]

计算出来的分支跳转地址

### exmem\_pc\_4 [31:0]

当前pc+4

### exmem\_lf

less flag

### exmem\_zf

zero flag

## 输出

### final\_target

真正的目标地址

具体功能

本模块要完成的工作与单周期cpu类似，但是有1点不同：

由于mips有延迟槽，而我们要做分支预测，所以我们会让编译器保证分支指令的下条指令一定为nop，以免发生错误。为了避免这些nop指令带来吞吐量下降，我们在分支预测时，如果预测不跳转，我们会将下条指令的pc设置为当前pc+8。所以在此单元中，如果这是一条分支指令，但是分支条件不满足，那么输出的final\_target是pc+8而不是pc+4；

用伪代码表示：

if not branch:

final\_target = pc +4

else if condition\_satisfy(condition,lf,zf):

final\_target = exmem\_target

else:

final\_target = pc+8

# load, store相关器件

本次设计中，我们将load和store系列指令的字节写使能发生模块放到了EX段，还把store系列指令的数据移位器放到了EX段，load系列指令的移位器仍然在MEM段。（注意：为了冲刷流水段方便，我们的写使能信号都是1有效，0无效。）

## load\_b\_w\_e\_gen

load系列指令的写使能信号

Load系列指令

|  |  |
| --- | --- |
| LB | GPR[rt] ← sign\_extend(memory[addr]) , addr[1:0]可以为0~3 |
| LBU | GPR[rt] ← zero\_extend(memory[addr]) , addr[1:0]可以为0~3 |
| LH | GPR[rt] ← sign\_extend(memory[addr]) , addr[1]可以为0或1；addr[0]必须为0 |
| LHU | GPR[rt] ← zero\_extend(memory[addr]) , addr[1]可以为0或1；addr[0]必须为0 |
| LW | GPR[rt] ← memory[addr] , addr[1:0]必须为0 |
| LWL | 见手册 |
| LWR | 见手册 |

LB, LBU, LH, LHU:

这4条指令会对取出的数进行扩展，LW的取数位宽本身为4byte，所以他们对应的字节写使能都是4’b1111。而LWL和LWR指令的偏移量和addr[1:0]有关：

LWL:

|  |  |
| --- | --- |
| addr[1:0] | Reg\_byte\_write\_en |
| 0 | 1111 |
| 1 | 1110 |
| 2 | 1100 |
| 3 | 1000 |

LWR:

|  |  |
| --- | --- |
| addr[1:0] | reg\_byte\_write\_en |
| 0 | 0001 |
| 1 | 0011 |
| 2 | 0111 |
| 3 | 1111 |

## store\_b\_w\_e\_gen

STORE系列指令的写使能信号

store系列指令

|  |  |  |
| --- | --- | --- |
| 指令 | 行为 | 数据位宽 |
| SB | memory[addr] ← GPR[rt] , addr[1:0]可以为0~3 | 1byte |
| SH | memory[addr] ← GPR[rt] , addr[1]可以为1或0，addr[0]必须为0 | 2bytes |
| SW | memory[addr] ← GPR[rt] , addr[1:0]必须为0 | 4bytes |
| SWL | 见手册 | 不确定 |
| SWR | 见手册 | 不确定 |

store系列指令中的SB和SH指令不会对数据进行拓展。出于简化存储器设计的考虑，我希望送给存储器的**地址总是4字节对齐**的，即只有30位，然后再通过**字节写使能信号**控制每个字节是否被写入，通过移位得到要送入存储器的数据。

用一个例子来进一步说明我的想法：SB指令的addr计算结果为0x40000001，需要在0x40000001处写入一个字节，要写入的数据是0xff。按照我的思路，应该把地址(0x40000001 & 0xfffffffc)送给存储器，将字节写使能信号4’b0100送给存储器（大端方式），将数据0x0000ff00送给存储器。

SW: 1111

SB:

|  |  |
| --- | --- |
| addr[1:0] | mem\_byte\_write\_en |
| 0 | 1000 |
| 1 | 0100 |
| 2 | 0010 |
| 3 | 0001 |

SH:

|  |  |
| --- | --- |
| addr[1] | mem\_byte\_write\_en |
| 0 | 1100 |
| 1 | 0011 |

SWL:

|  |  |
| --- | --- |
| addr[1:0] | mem\_byte\_write\_en |
| 0 | 1111 |
| 1 | 0111 |
| 2 | 0011 |
| 3 | 0001 |

SWR:

|  |  |
| --- | --- |
| addr[1:0] | mem\_byte\_write\_en |
| 0 | 1000 |
| 1 | 1100 |
| 2 | 1110 |
| 3 | 1111 |

## store\_shifter

STORE指令的数据移位器

* SW: 不需要移位
* SB: out = in << (addr [1:0] \* 8)
* SH: out = in << (addr [1] \* 16)
* SWL: out = in >> addr [1:0]
* SWR: out = in << ~addr [1:0]

注：in表示来自寄存器的数据，out表示输出的数据

## load\_shifter

LOAD指令的数据移位器

* LW：不需要移位
* LB: in = in >> (addr [1:0] \* 8); out = in [7]24 + in [7:0]
* LBU: in = in >> (addr [1:0] \* 8); out = 024 + in [7:0]
* LH: in = in >> (addr [1]\* 16); out = in [15]16 + in [15:0]
* LHU: in = in >> (addr [1]\* 16); out = 016 + in [15:0]
* LWL: out = in << addr [1:0]
* LWR: out = in >> ~addr [1:0]

注：in表示来自存储器的数据，out表示输出的数据；取自存储器的数据宽度都是4bytes