## 实验九 实现访存指令的理想流水线设计实验

### 实验目的

1. 掌握访存指令的数据通路。
2. 掌握在执行级和访存级中添加访存部件 LSU 的方法。
3. 掌握在五级流水线中实现访存指令的方法。

### 实验原理与实验内容

RV64I中访存指令共计11条，包括7条加载（Load）指令和4条存储（Store）指令，详细内容请参考第1章的1.4小节的访存指令。

在之前的实验中，访存单元内部未实现任何逻辑，是毫无功能的。在本实验中终于可使访存单元发挥其应有的作用。下面以lh指令和sb指令为例，分析一下如何设计实现访存指令的数据通路。

#### 访存类指令的数据通路的前端设计

前端只负责指令的准备，指令的执行细节都在后端部分，因此无需修改前端的数据通路。

#### 访存类指令的数据通路的后端设计

前端部分已经成功取得指令，按照指令执行的一般流程可以分析得到：在译码单元产生合适的控制信号，在访存单元进行访存操作，在写回单元将数据写回寄存器堆。

##### 译码单元

译码单元要完成指令译码产生控制信号以及准备源操作数这两个操作。

lh指令格式和功能

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **位数**  **指令** | **31 25** | **24 20** | **19 15** | **14 12** | **11 7** | **I型格式访存读数指令的功能** |
| **offset12** | **rs1** | **funct3** | **rd** | **opcode** |
| lh rd, imm12(rs1) | offset[11:0] | rs1 | 001 | rd | 0000011 | 取半字数据：  SEXT64(Mem[rs1+SEXT64(offset)][15:0])→rd |

sb指令格式和功能

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **位数**  **指令** | **31 25** | **24 20** | **19 15** | **14 12** | **11 7** | **6 0** | **S型格式访存写数指令的功能** |
| **offset7** | **rs2** | **rs1** | **funct3** | **offset5** | **opcode** |
| sb rs2, offset12(rs1) | offset[11:5] | rs2 | rs1 | 000 | offset[4:0] | 0100011 | 存字节数据，将rs2的低8位存入存储器:  rs2[7:0]→Mem[rs1+SEXT64(offset[11:0])] |

由表7-1展示的lh指令定义可以得知，lh为I型指令，在之前的实验中，已经实现了I型运算指令的数据通路，因此准备lh指令源操作数的数据通路已经实现完毕，无需进行修改。对于访存写数指令而言，虽然其为S型指令，但其源操作数均来自通用寄存器堆。至于S型指令的立即数字段，只需修改译码器内imm信号的生成逻辑生成正确的立即数即可，该信号被包含于info数据包内向后传递并在后续流水级可以随时使用，因此访存写数指令也无需修改译码单元的数据通路。

译码信号

|  |  |  |  |
| --- | --- | --- | --- |
| **信号名** | **含义** | **信号名** | **含义** |
| src1\_ren | src1是否需要读通用寄存器堆 | src2\_ren | src2是否需要读通用寄存器堆 |
| src1\_raddr | src1的通用寄存器堆读地址 | src2\_raddr | src2的通用寄存器堆读地址 |
| reg\_wen | 写回通用寄存器堆使能信号 | reg\_waddr | 通用寄存器堆的写地址 |
| op | 指令的操作类型 | valid | 指令是否有效 |
| inst | 指令 | imm | 立即数 |
| fusel | FU选择信号 |  |  |

表7-3展示了译码器生成的信号。imm信号为立即数字段符号拓展至64位的结果。fusel信号需要修改，只需修改FuType中的值即可：将num修改为3，定义lsu为2；这时由于fusel信号是由apply方法进行的实例化，其宽度会自动拓展为2。对于访存类指令，其fusel信号的值应当为FuType.lsu。下面的代码展示了FuType的定义。

object FuType {

  def num     = 3

  def alu     = 0.U // arithmetic logic unit

  def mdu     = 1.U // multiplication division unit

  def lsu     = 2.U // load store unit

  def apply() = UInt(log2Up(num).W)

}

op信号也需要进行修改，观察1.4小节的访存指令格式，可以通过opcode分辨加载和存储指令，通过func3字段进行不同操作的细分。可以使用func3字段作为op的低3位，在高位使用0表示加载，1表示存储，也就是将op定义为op={I5|I[14:12]}。因此lh指令的op应该为0001，lw指令的op为1010。

仔细观察func3字段可以发现：func3的低2位对应了访存的宽度，00表示1字节，01表示2字节，10表示4字节，11表示8字节，访存宽度正好是2func3[1:0]字节；func3的最高位表示是否是无符号拓展。可见指令的编码都经过了精巧的设计。

将访存相关指令的op定义于LSUOpType中，便于统一管理，代码如下：

object LSUOpType {

  def lh  = "b0001".U

  def sb  = "b1000".U

  def isStore(func: UInt): Bool = func(3)

  def isLoad(func:  UInt): Bool = !isStore(func)

}

对于读数指令而言，访存读到的数据需要在写回单元写回寄存器堆，因此reg\_wen为1；同理得到写数指令的reg\_wen为0。

综上所述，译码单元只需改动译码器内部逻辑代码即可，整体译码单元的结构无调整，与上一实验译码单元结构一致。这里再次强调，所有的译码信号都在info数据包中向后续流水级传递，chisel在生成verilog时会自动删去在后续流水级中未使用到的信号，因此无需担心信号冗余。

##### 执行级缓存和执行单元

执行级缓存无需修改。

DataMEM与InstMEM一样也是使用SRAM实现的。对于读数据而言，其时序是第一个周期发送读地址，第二个周期获得DataMEM的返回结果；而存数据则不必考虑时序，因为不需要得到DataMEM的回应，只要发送写使能、写地址和写数据即可。为了在访存级可以读到正确的数据，应当在执行单元向DataMEM发起读地址，这样在下一个周期，指令流动到访存单元时，正好可以读到正确的数据。



执行单元结构

图7-1展示了执行单元的结构。与InstMEM一样，DataMEM的使能在reset撤销后恒置为1。由于SRAM结构的存储器的存储与读取共用一个地址通道，因此读数和存数的请求都只能在执行阶段发送。

根据访存指令的定义，Data\_MEM\_addr等于src\_info.src1\_data加上info.imm。

DataMEM\_wen信号宽度为8位，用于控制写入的字节位置，对于字节写使能的正确设置，可以在一块8字节宽的RAM上完成字节、半字、字和双字的写入。表7-4展示了写地址和字节写使能的对应关系，还需要考虑到当前指令的有效性以及指令是否是访存写数指令写数指令，因此最终的字节写使能DataMEM\_wen只有在info.valid有效且info.fusel等于FuType.lsu并且满足LSUOpType.isStore(info.op)为真时才能取DataMEM\_wen\_temp的值，否则字节写使能为0。伪代码为DataMEM\_wen =DataMEM\_wen\_temp&8{info.valid&& info.fusel==FuType.lsu&&LSUOpType.isStore(info.op)}

sb字节写使能的生成逻辑

|  |  |
| --- | --- |
| **Data\_MEM\_addr[2:0]** | **DataMEM\_wen\_temp** |
| 000 | 00000001 |
| 001 | 00000010 |
| 010 | 00000100 |
| 011 | 00001000 |
| 100 | 00010000 |
| 101 | 00100000 |
| 110 | 01000000 |
| 111 | 10000000 |

DataMEM\_wdata的设置也与存数的字节数有关，假设src\_info.src2\_data的值为0123\_4567\_89AB\_CDEF，不同的指令下写数据的值如表7-5所示。

写数据DataMEM\_wdata的生成逻辑

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **指令** | **op[1:0]** | **字节数** | **生成逻辑** | **DataMEM\_wdata** |
| sb | 00 | 1 | {8{src2\_data[7:0]}} | EFEF\_EFEF\_EFEF\_EFEF |
| sh | 01 | 2 | {4{ src2\_data[15:0]}} | CDEF\_CDEF\_CDEF\_CDEF |
| sw | 10 | 4 | {2 {src2\_data[31:0]}} | 89AB\_CDEF\_89AB\_CDEF |
| sd | 11 | 8 | src2\_data | 0123\_4567\_89AB\_CDEF |

将所有和DataMEM相关的信号封装到一个模块中，称为LSU模块。

##### 访存级缓存和访存单元

访存级缓存无需修改。

存数操作已经在执行单元内完成，访存单元只需处理读数操作读到的数据即可。

首先，由于RAM为8字节宽，对于不同的访存地址读到的数据有所区别：假如访问到的数据为0123\_4567\_89AB\_CDEF，不同地址下预处理的结果如表7-6所示，DataMEM\_rdata\_temp的宽度为64位，不同的赋值都会自动进行无符号拓展至64位。

读取数据预处理

|  |  |  |
| --- | --- | --- |
| **Data\_MEM\_addr[2:0]** | **处理逻辑** | **DataMEM\_rdata\_temp** |
| 000 | DataMEM\_rdata[63:0] | 0123\_4567\_89AB\_CDEF |
| 001 | DataMEM\_rdata[63:8] | 0123\_4567\_89AB\_CD |
| 010 | DataMEM\_rdata[63:16] | 0123\_4567\_89AB |
| 011 | DataMEM\_rdata[63:24] | 0123\_4567\_89 |
| 100 | DataMEM\_rdata[63:32] | 0123\_4567 |
| 101 | DataMEM\_rdata[63:40] | 0123\_45 |
| 110 | DataMEM\_rdata[63:48] | 0123 |
| 111 | DataMEM\_rdata[63:56] | 01 |

对于不同的指令，还需对DataMEM\_rdata\_temp进行进一步处理，得到最终的读数据写入到rd\_info.wdata(FuType.lsu)中，处理逻辑如表7-7所示。

不同读数指令读取到的最终数据

|  |  |
| --- | --- |
| **指令** | **处理逻辑** |
| lb | SEXT64(DataMEM\_rdata\_temp[7:0]) |
| lh | SEXT64(DataMEM\_rdata\_temp[15:0]) |
| lw | SEXT64(DataMEM\_rdata\_temp[31:0]) |
| ld | DataMEM\_rdata\_temp |
| lbu | UEXT64(DataMEM\_rdata\_temp[7:0]) |
| lhu | UEXT64(DataMEM\_rdata\_temp[15:0]) |
| lwu | UEXT64(DataMEM\_rdata\_temp[31:0]) |

将读数相关处理逻辑包装成模块，最终的访存单元结构如图7-2所示。



访存单元结构

##### 写回级缓存和写回单元

写回级缓存无需修改。

上一实验已经修改过写回单元的数据通路使得写回单元可以根据info.fusel的值选择正确的wdata写回寄存器堆，因此结构也无需调整。

### 实验要求

1. 在上一个实验的基础上继续添加第1章的1.4小节的所有访存指令包括读数和写数共计11条指令。
2. 通过本实验提供的所有仿真验证测试用例。
3. 通过本实验提供的所有板级验证测试用例。
4. 线上提交实验作品，获得系统评分。
5. 撰写实验报告。报告中应包含以下内容和至少3道本实验思考题。
6. 仿照表7-4，列出sh、sw、sd的写地址与字节写使能的对应关系。
7. 选择第1章的1.4小节访存指令中的一条指令（非lh、sb），按照你自己的理解，逐步介绍其数据通路设计的思路以及实现过程。
8. 修改MyCPU内部数据通路图，增加访存功能。

### 实验步骤

1. 实验步骤参见6.3节实验九的实验步骤部分。

### 思考与探索

1. 在列写地址和字节写使能对应关系时是否遇到了问题？RISC-V定义了地址未对齐异常，在该问题上对你有什么启发？
2. DataMEM\_en可以不恒为1吗？如果可以，应该怎么修改？
3. 表7-6展示了读取同一个8字节数据块的情况，假如需要跨块访问会发生什么情况？RISC-V是怎么解决这种问题的？可以和MIPS进行对比举例。
4. 谈谈你在实验中碰到了哪些问题？又是如何解决的？