

dv-cpu-rv: 基于 RISC-V 的 CPU 的设计

Devin

balddevin@outlook.com

2023/7/22

内容

1	前言	4
2	硬件设计	4
2.1	基本的单周期实现	4
2.1.1	设计框图	4
2.1.2	数据通路的构建	5
2.1.3	控制通路的构建	5
2.1.4	处理器是如何工作的?	6
2.2	基本的流水线实现	7
2.2.1	设计框图	7
2.2.2	数据冒险: 转发或旁路	9
2.2.3	控制冒险: 分支预测	10
2.2.4	结构冒险: 暂停	10
2.3	对 RV32I 和 RV64I 的完整支持	11
2.3.1	设计框图	11
2.3.2	修改内容总览	11
2.3.3	ALU 控制单元	11
2.3.4	分支和跳转	12
2.3.5	选通单元	13
2.4	对 RV32M 和 RV64M 的完整支持	15
2.4.1	Booth-Wallace 乘法器	15
2.4.2	SRT 除法	21
3	功能描述	31
3.1	文件和目录结构	31
3.2	RV32I Module Design	31
3.3	RV32M Module Design	31
4	附录	32

4.1	附录 1: 指令集支持情况.....	32
4.2	附录 2: 运行示例	35
4.2.1	示例 1: 相加然后保存.....	35
4.2.2	示例 2: 累加小于该数的数.....	35

2.1.2 数据通路的构建

数据通路是处理器中用于操作或者保存数据的单元。在 RISC-V 的视线中，数据通路的元素通常包括指令内存，数据内存，寄存器组，ALU (算术处理单元)，以及一些加法器。

指令内存 是保存指令的存储单元，在哈佛架构的计算机中它与数据内存是独立的。在该实现中，指令内存是一个地址线位宽为 64 的存储器，与 CPU 处理的数据位宽相同，并且数据位宽是 32 位，也即是 RISC-V 指令的宽度。

PC (程序计数器) 用来取出指令内存中的指令，在大多数情况下，它每个时钟周期自增 4，这是由于 32 是 4 字节，RISC-V 中寻址的单位是字节。在某些情况下，PC 将会跳转或者分支到指令内存的特定位置执行取值操作，因此，框图的右上角有一个 MUX (多路复用器)。

寄存器组 包含 RISC-V 中定义的所有 32 个寄存器，每个都是 64 位宽，它被设计用来存储常值 0，参数，PC 值，子程序入口等等。

ALU (算术处理单元) 是 CPU 的核心，它负责了几乎所有的算术运算，例如 I 标准中的加法，减法，异或，或，与操作，更宽泛点说，M 和 F 扩展中的，乘法，除法，浮点运算等等。该设计中的 ALU 具有 64 位宽的数据宽度，无论是操作数，还是运算结果。

数据内存 是用于存储大量数据的存储单元。它决定了计算机在同一时间内可以处理的最大的数据量。例如，一个数据内存的位宽是 1 个字节，地址宽度是 32，最大数据量为 $2^{32} \times 1\text{B} = 2^{32} \times 1\text{GiB} = 4\text{GiB}$ 。在改设计中，数据内存的数据宽度是 64，也就是 8 字节，因此物理地址必须除以 8。

立即数产生单元 是用于符号位扩展立即数，将其传输给 ALU 或者作为 PC 的偏移值传输给控制进程。它对取到的指令进行译码，根据 opcode 域，funct7 域，以及 funct3 域取出对应的立即数，将其符号位扩展到 64 比特。

2.1.3 控制通路的构建

控制通路是根据指令译码的结果来控制数据通路之间数据传输的单元。在该 RISC-V 实现中，控制通路通过产生以下信号来控制整个进程。

注意这些控制信号仅仅适用于单周期的情况，仅仅适用于该提交，未来的更新中引入了几个更为复杂的控制信号，这会在后续的章节中进行讨论。

控制线	无效	有效
Branch	无。	与分支测试单元一并决定是否进行分支操作。
MemRead	无。	读数据内存。
MemToReg	送给寄存器的写数据输入来自于 ALU 的运算结果。	送给寄存器的写输入数据来自于数据内存。

MemWrite	无。	写数据内存。
ALUSrc	ALU 的操作数 2 来自于寄存器的读数据 2。	ALU 的操作数 2 来自于立即数产生单元的输出。
RegWrite	无。	写数据到寄存器。
RegSrc	X	X

RegSrc 是一个失败的设定，它会在未来被抛弃，最初创建它是为了选择 PC 作为寄存器的写数据来支持类似 JAL 和 JALR 的指令。

有了这些控制信号，CPU 就可以处理简单的计算了，他会在一个时钟周期内执行一条指令。在 `./docs/design/assembly.md` 中有一个例子，标号为 1，它执行一个简单的加法运算，得到 29 作为运算结果，然后存储到内存中。这是支持该提交的唯一一个测试用例。

2.1.4 处理器是如何工作的？

单周期的 CPU 实现是解释该问题的一个很好的例子。如前所述，处理器包括数据通路和控制通路，数据通路负责数据的交换，而控制通路产生几乎所有的使能信号和选择信号，从而引领处理器走在正确的道路上。

取指：处理器首先从指令内存中取出指令，指令是处理器可以自行读懂的机器码。在 RISC-V 中的指令都是 32 比特长的，因此无需添加额外的逻辑来支持更短的指令。处理器一个接一个的读出指令，处理这些指令，然后给出对应的响应。

译码：既然处理器以及获得了指令，下一步操作就是译码，也就是，分析这些指令。控制单元在这一步中生效，它产生像 Branch, MemRead, ALUSrc 等控制信号。立即数产生单元也在此处生效，它对 I 类型的指令和 U 类型的指令做符号位扩展以便处理器可以在下一级中将它们作为操作数。

执行：ALU 和 ALU 控制单元在此处生效，ALU 根据 ALU 控制单元执行算术运算。此外，分支测试单元也放在了该处，它是为了测试在下次取值时是否要进行指令的分支操作，把它放在该处的原因是分支操作依赖于 ALU 的运算结果。

访存：在 S 类型和 I 类型的 load 指令中，ALU 运算结果被送往数据内存的地址端口。仅仅在该阶段，内存被访问。内存的读操作和写操作均有 ID (译码) 阶段的控制单元来决定。

写回：RISC-V 不允许直接访问内存来获得操作数，即便是在写回阶段也不行。所有与内存相关的数据都必须在寄存器中进行操作。写回阶段将 load 指令从内存读出的数据或者 ALU 运算结果写回到寄存器中。

2.2 基本的流水线实现

2.2.1 设计框图

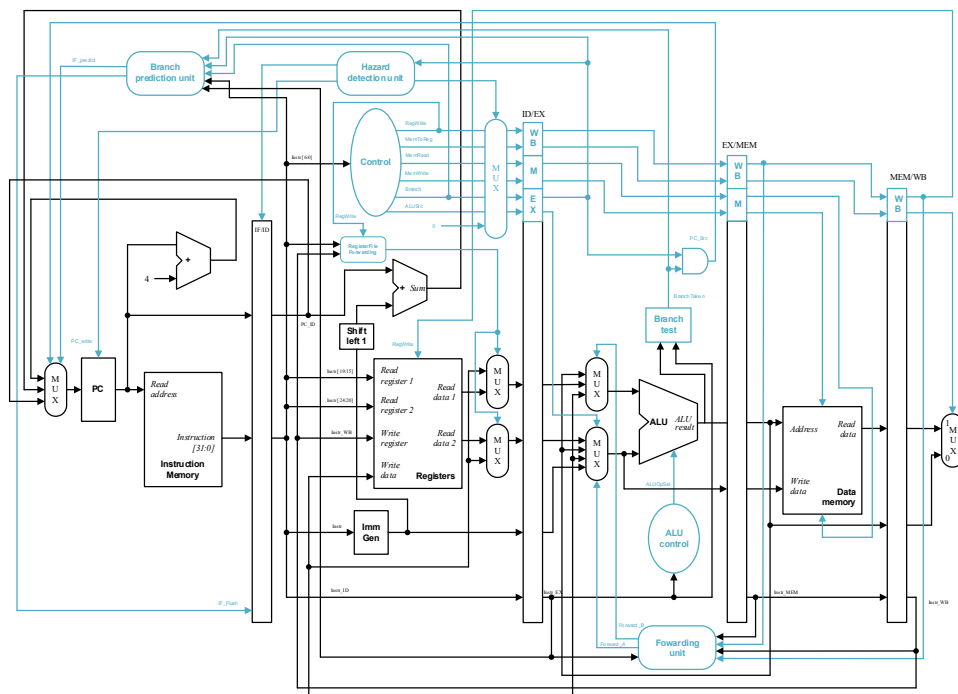


Figure 2.2 CPU 的基本的流水线实现

对应的哈希标识为: 1d28ab2a485737b8bd90fa777fd550d5183b705c

上图展示了基本的流水线 CPU 的实现，高亮的线条表示控制通路的信号，而其他的线条表示数据通路的信号。与单周期的实现相比，需要额外的单元，它们是：

- 4 个用于流水线的寄存器，分别叫做 IF/ID, ID/EX, EX/MEM, MEM/WB；
- 转发单元，用于处理流水线结构引入的数据冒险；
- 冒险检测单元，用于在特定的情况下暂停流水线；
- 分支预测单元，用于加速 CPU，节省操作周期；
- 转发单元，用于解决寄存器组中的寄存器读写冲突的问题。

这些单元的细节将会在后续的章节中进行讨论：对 RV32I 和 RV64I 的完整支持。

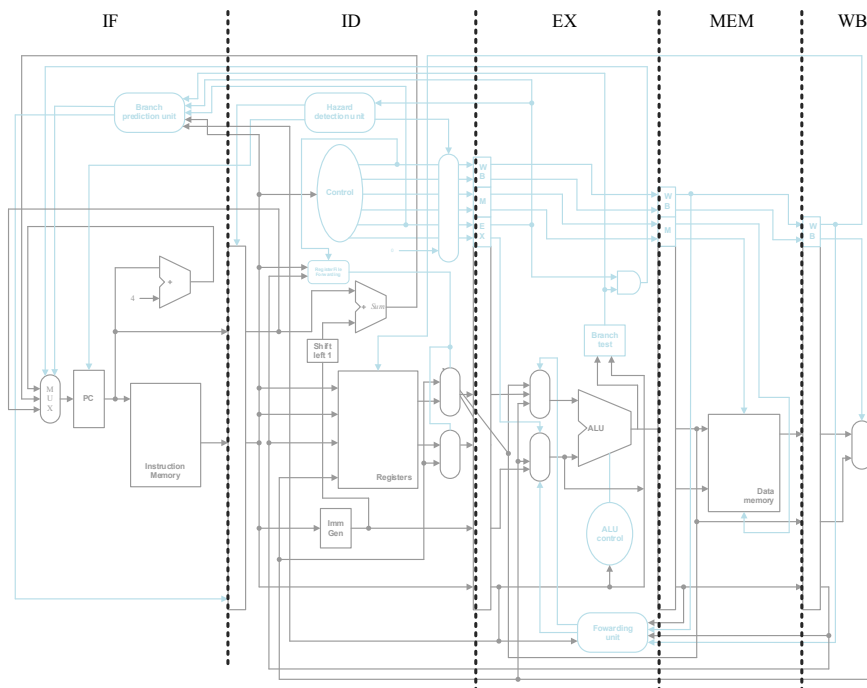


Figure 2.3 五级流水线

上图是五级流水 CPU 的划分，五个流水线级和它们的功能如下 (已经被讨论过):

1. 取指 (IF, Instruction Fetch): 从指令内存中取出指令，把 PC 值作为内存的地址。此外，分支预测单元也放置在此处用于控制 PC 的行为。
2. 译码 (ID, Instruction Decode): 将指令进行译码，产生控制信号，符号扩展立即数，控制寄存器组的操作。此外，冒险检测单元也放置在此处用于暂停流水线。
3. 执行 (EX, Execution): 执行算术计算，以及分支测试。此外转发单元也放置在此处用于转发流水线各级的结果。
4. 访存 (MEM, Memory Access): 对于 load 和 store 指令，访问内存。
5. 写回 (WB, Write Back): 将数据写回到寄存器中。

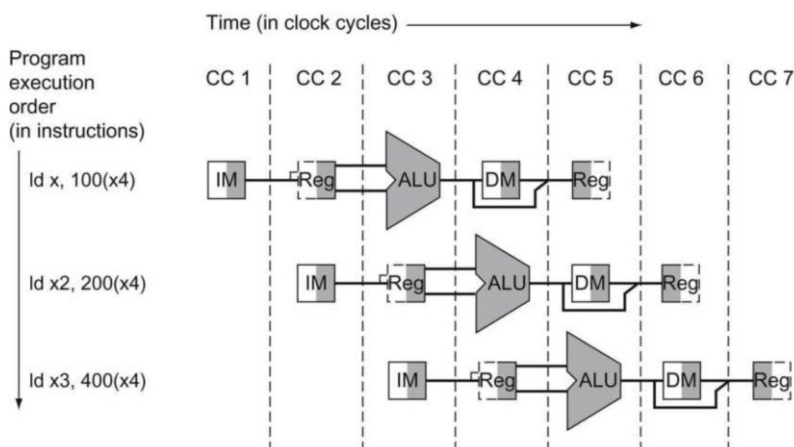


Figure 2.4 本设计中的流水线时序

和单周期的实现不同，流水线架构允许五个阶段之间的重叠，从而提高了吞吐量。它带来的问题也很明显，我们会在后续的三节中进行讨论。

2.2.2 数据冒险：转发或旁路

数据冒险是流水线执行的障碍。解决这个问题的办法是添加一个转发单元，将数据转发到前一级的数据流中，例如将 ALU 运算结果，或者寄存器的输出数据，转发到当前的执行周期，而不是等待上一个指令的写回操作。

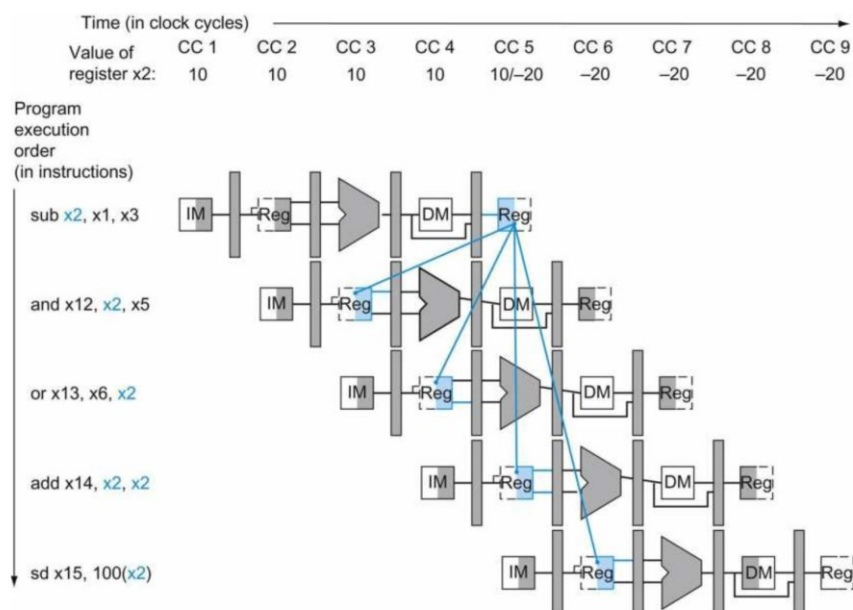


Figure 2.5 Data hazard in pipeline architecture

上图是 *Computer Organization and Design* 中的一个例子，`and` 指令需要寄存器 `x2` 的值作为算术运算的操作数，但是处理器无法及时提供 `x2` 的值，`sub` 指令的值仅在写回操作之后才可用。下一条指令也会面临同样的问题。

解决这个问题的方式是添加一个转发单元，把 ALU 运算的结果或者寄存器的输出转发或者旁路到 ALU 的输入端，以便 ALU 可以及时获得正确的操作数。

我们简单的检查 `rs1` 在 EX 阶段的值，如果 `rd` 在 MEM 阶段的值和它相等，就需要一个转发操作，这是书中所提到的 1a 的情况，ALU 的运算结果在 MEM 阶段的值转发到 ALU 的操作数 1。否则如果 `rd` 在 WB 阶段的值和它相等，进一步寄存的值需要被转发，这是情况 2a。

这只是 ALU 操作数 1 的情况，操作数 2 具有类似的情况，但是需要做进一步的判断。在 RISC-V 中，I 类型的指令没有 `rs2` 域，它会干扰数据的转发，所以如果检测的当前指令是 I 类型的，就无需进行转发，这是额外的判断条件。

2.2.3 控制冒险：分支预测

分支预测是一种应用在 CPU 中的技术，它试图猜测条件运算的结果，并且为更为可能发生的结果做准备。执行该操作的数字电路被称为分支预测器。它是现代 CPU 架构中一个非常重要的部分。

我们来看一下处理器没有分支预测的情况。分支结果仅当处理器在执行阶段从分支测试单元中得到结果之后才可用，在 PC 的值改变后，有 3 个周期的延迟 ($PC \Rightarrow IF \Rightarrow ID \Rightarrow EX$)。由于分支指令被充填到了整个流水线中，分支结果当前不可用，分支指令后续的几个指令也会充填到流水线中，无论分支是否被采取。这其实就是我们预测分支永远不采取的情况，如果分支被采取了，需要浪费额外的时钟周期。

基本的分支预测方案是使用单比特预测缓存，它存储对应地址或者低几位地址的历史预测结果。

实际上，我们不知道，预测结果是否正确——它或许使用的是低几位地址相同的别的分支指令的预测结果。然而，这并不影响正确性。预测只是一个暗示，我们希望它正确，因此取指的地址使用预测的方向。如果暗示的结果是错误的，则删除错误预测的指令，将预测的结果存储，并且重新获取并执行正确的指令。

对于大多数情况，循环条件使用相同的地址 (在 RISC-V 中是地址偏移) 进行分支，并且重复很多次。分支预测机制将会在这种情况下节省时钟周期。

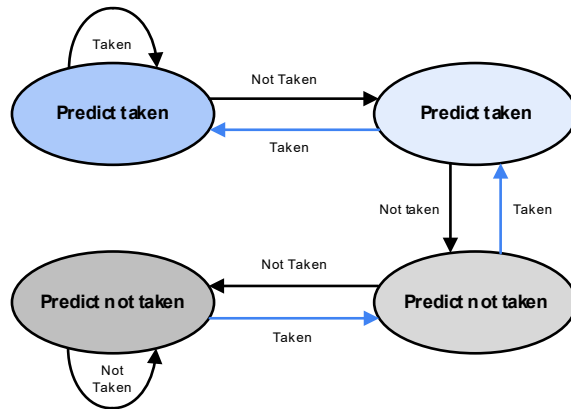


Figure 2.6 State transition for 2-bit dynamic prediction

上图展示了 2 比特分支预测技术的状态转移图，只有 2 个分支操作被采取之后，预测单元才会给出分支指示。它的实现简单的使用了一个 2 比特的计数器，计数器值 0，1 使预测指示无效，而计数器值 2 和 3 使预测指示有效。

2.2.4 结构冒险：暂停

2.3 对 RV32I 和 RV64I 的完整支持

2.3.1 设计框图

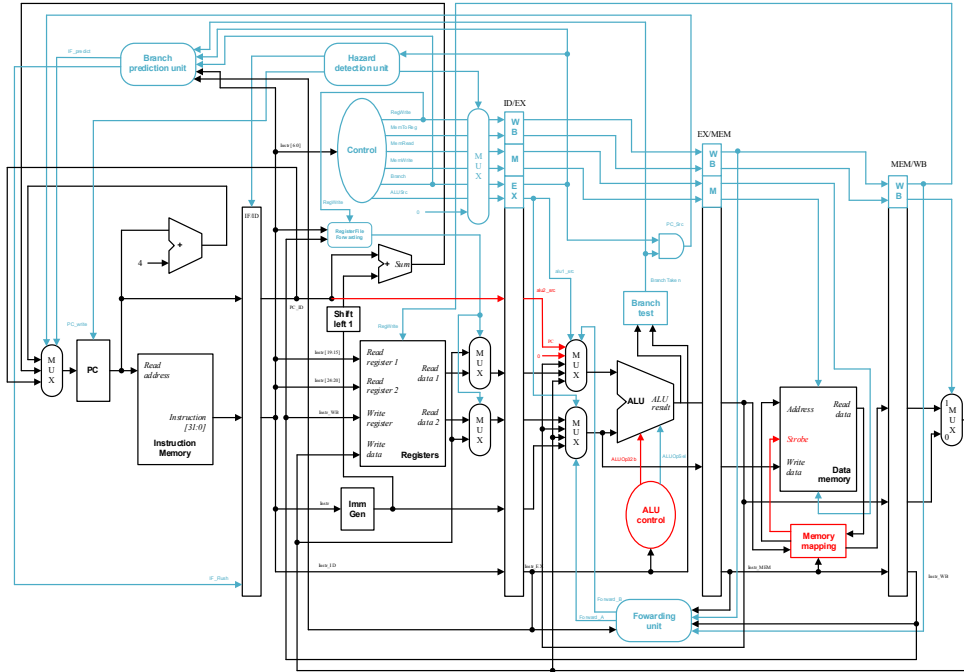


Figure 2.7 RV32I/RV64I design diagram

红色高亮线是为了完整支持 RV32I 和 RV64I 中的指令而引入的逻辑单元。它会稍后被讨论。

2.3.2 修改内容总览

以下是为了完整支持 RV32I 和 RV64I 而修改的单元或者额外的单元：

- ALU 控制单元的 ALUOp32b 端口，它是为了 RV64I 支持字操作而设计的；
- 内存映射单元，它是为了支持内存的字节，半字，字操作而设计的，更多设计细节，请参考 [3.2.5 选通单元](#)；
- ALU op1 的 MUX 逻辑, PC 和 0 被添加到了 MUX 的输入端, LUI 指令使用 0 作为 op1, AUIPC 指令使用 PC 作为 op1；
- 此外, ImmGen, Control, ALU, Memory, 和 BranchTest 单元也需要进行更新以支持更多的指令，细节将在后续的章节中进行讨论。

2.3.3 ALU 控制单元

为了简化 CPU 的设计，引入了控制通路和数据通路的概念。ALU 控制单元控制 ALU 的操作模式以支持所有指令，在 RV32I 指令集中，该模块中的 `alu_op_sel` 信号继承了 R 类型指令的 `funct3` 域和 `instr[30]` 字段。

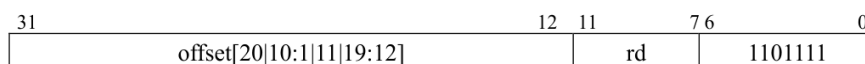
<i>alu_op_sel</i>	功能
0000	Add
1000	Subtract
0001	Shift Left Logical
0010	Set Less Than
0011	Set Less Than Unsigned
0100	Exclusive Or
0101	Shift Right Logical
1101	Shift Right Arithmetic
0110	Or
0111	And

2.3.4 分支和跳转

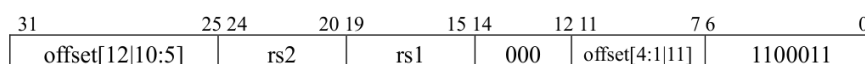
RISC-V 的指令都是 4 字节长的，RISC-V 的分支指令在设计时，使用 PC 相对寻址，以及分支指令和目标指令之间的字节的个数，来拓宽分支指令可以到达的区域。然而 RISC-V 架构想要支持 2 字节长度的指令，所有分支指令使用的是分支和目标分支之间半字的个数¹。

因此，jal 指令中 20 位的地址偏移域可以译码 $\pm 2^{19}$ 个半字的距离，或者说从当前 PC 开始 ± 1 MiB 的距离。类似的，条件分支指令中 12 位的地址偏移域也使用半字作为单位，意味着它实际上标识的是 13 个字节的地址。

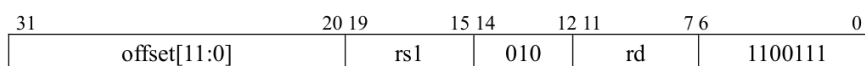
jal



beq



jalr



术语“地址偏移”在 RISC-V 中是以字节为单位的，所以 jal 和 beq 指令中的 offset 域使用 [12:1] 而不是 [11:0]，意味着 offset 域中的内容拼接上一个 0 才是真实的以字节为单位的偏移量。

¹ Computer Organization and Design RISC-V Edition: P264

注意 JALR 指令的偏移域的内容是以字节为单位定义的，而非半字。PC 使用以下几种情况作为源：

- 自增 (物理地址 +4)
- 分支目标 (需要移位 1)
- JAL (需要移位 1)
- JALR (无需移位)

2.3.5 选通单元

选通单元被设计用于实现以下与 Load 和 Store 相关的指令，它们是：

- LB, LH, LW, LBU, LHU, SB, SH, SW in RV32I,
- LD, LWU, SD in RV64I.

LD 指令适用于 RV64I，它从内存中加载一个 64-bit 位宽的数据到寄存器的 rd 端。

LW 指令适用于 RV64I，它从内存中加载一个 32-bit 位宽的数据并且符号扩展到 64 位后存入寄存器的 rd 端。

LWU 指令适用于 RV64I，不同的是，它对 32-bit 位宽的数据进行 0 符号扩展。

类似的，LH 和 LHU 被定义用来支持 16-bit 的值，LB 和 LBU 支持 8-bit 的值。SD, SW, SH, 和 SB 指令分别存储寄存器 rs2 的低 64-bit, 32-bit, 16-bit, 和 8-bit 的对应的值到内存。

地址偏移是以字节为单位的。对于 SD 和 LD 指令，地址偏移必须是 8 的倍数 (一个双字是 8 个字节)，例如

```
ld x9, 240(x10) // Temporary reg x9 gets A[30]
lw x10, 240(x10) // Temporary reg x9 gets A[30][31:0]
```

为了简单起见，必须对齐偏移量，这一点很重要，没有对齐的内存访问会消耗额外的时钟周期从而拖慢 CPU。

假定访问的数据已经在内存中对齐，他们以特定的规则进行存储或者取出，考虑以下的例子来进行理解²。

² <https://stackoverflow.com/questions/28707615/loading-and-storing-bytes-in-mips>

Given following code sequence and memory state (contents are given in hexadecimal and the processor use big Endian format), what is the state of the memory after executing the code?

```
add $s3, $zero, $zero
lb  $t0, 1($s3)
sb  $t0, 6($s3)
```

mem(4) = 0xFFFF90FF

Memory	00000000	24	What value is left in \$t0?
	00000000	20	\$t0 = 0x00000090
	00000000	16	
	10000010	12	What if the machine was little Endian?
	01000402	8	mem(4) = 0xFF12FFFF
	FFFFFFFF	4	\$t0 = 0x00000012
	009012A0	0	
	Data		Word Address (Decimal)

4

【中文翻译】给定以下机器码序列和内存的状态 (内容以十六进制表示，处理器使用大端序)。

- 在执行了以下机器码之后内存的状态是怎样的？
- 寄存器 \$t0 中的值是什么？
- 如果机器使用小端序呢？

```
add $s3, $zero, $zero
```

这句话执行加法 $\$s3 = 0 + 0$ ，把寄存器 \$s3 的值设置为 0。

```
lb $t0, 1($s3)
```

这句话从内存中的某个位置加载一个字节到寄存器 \$t0。内存的地址由 1(\$s3) 提供，意思是地址为 \$s3+1。也就是 0+1=1 内存中的第一个字节 (注意这个例子是 MIPS 的，和 RISC-V 对于地址的处理不同)。由于我们使用的是大端序的架构，我们以“大端先行”的方式读取 4 字节块中的字节。

```
byte: 0 1 2 3
      00 90 12 A0
```

第 0 个字节是 00，第一个字节是 90。所以我们把字节 90 加载入寄存器 \$t0。

```
sb $t0, 6($s3)
```

这句话把寄存器 \$t0 中的一个字节存储到内存，内存地址由 6(\$s3) 确定。同样的，这句话意味着地址是 \$s3+6。

```
byte: 4 5 6 7
      FF FF FF FF
```

变成

```
byte: 4 5 6 7
```

```
FF FF 90 FF
```

现在，如果架构使用小端序呢？(RSIC-V 中使用的端序) 这意味着字节在内存中以“小端先行”的顺序进行排列，所以第二条指令和第三条指令的效果有所不同。

```
lb    $t0, 1($s3)
```

这条指令从内存地址 1 中加载比特到寄存器 \$t0。但是现在我们的地址是小端序排列的，所以我们读取并加载到寄存器的值是 12。

```
byte: 3  2  1  0
      00 90 12 A0
```

接下来...

```
sb    $t0, 6($s3)
```

这句话把寄存器 \$t0 中的字节，也就是 12 存储到内存地址 6。同样使用小端序架构：

```
byte: 7  6  5  4
      FF FF FF FF
```

变成

```
byte: 7  6  5  4
      FF 12 FF FF
```

选通单元就是被设计用来从内存的物理地址 0，大端序中的第二个字节处，以比特顺序 [23:16] 的地方读取 0x90，并且把 0x90 写入到正确的地址，正确的部分。

对于 sb, lb 指令，偏移量必须是 1 的倍数，低 3 个比特被用来确定使用 64 个比特中的哪个部分来存储和读取。对于 sh, lh, lhu 指令，偏移量必须是 2 的倍数，低 2 个比特被用来确定使用 64 个比特中的哪个半字来存储和读取。对于 sw, lw, lwu 指令，偏移量是 4 的倍数，使用低 1 比特判断。对于 sd, ld 指令，整个 64 比特被使用，偏移量必须是 8 的倍数。

2.4 对 RV32M 和 RV64M 的完整支持

2.4.1 Booth-Wallace 乘法器

该部分主要参考《计算机体系结构基础》<https://github.com/foxsen/archbase>，采用的是开源协议 [CC BY-NC 4.0](https://creativecommons.org/licenses/by-nc/4.0/)，感谢贡献者们。

有符号乘法

假定有一个 8x8 的乘法运算，有符号乘法的结果可以表示为

$$\begin{aligned}[X \cdot Y]_{2c} &= -X \cdot y_7 \cdot 2^7 + X \cdot y_6 \cdot 2^6 + \cdots + X \cdot y_1 \cdot 2^1 + X \cdot y_0 \cdot 2^0 \\ &= [X]_{2c} \cdot (-y_7 \cdot 2^7 + y_6 \cdot 2^6 + \cdots + y_1 \cdot 2^1 + y_0 \cdot 2^0)\end{aligned}$$

其中，2c 意思是 2 的补码，在后续的表达式中，我将省略该符号，此处只讨论有符号乘法的运算。

考虑以下例子， $-13 \cdot 6 = -78$ ，其中 -13 是被乘数 (multiplicand) X，6 是乘数 (multiplier) Y。最后一个部分积是全 0，在图中被省略。

$$\begin{array}{r}
 11110011 \\
 \times 00000110 \\
 \hline
 + \dots 111110011 \\
 + \dots 111110011 \\
 - \\
 \hline
 1110110010
 \end{array}$$

下面的例子是， $13 \cdot 6 = 78$ ，与上面的例子具有相同的结果。

$$\begin{array}{r}
 00001101 \\
 \times 11111010 \\
 \hline
 + \dots 00000001101 \\
 + \dots 000001101 \\
 + \dots 00001101 \\
 + \dots 0001101 \\
 + \dots 001101 \\
 - \dots 01101 \\
 \hline
 111110110010
 \end{array}$$

与无符号乘法不同的地方有：

- 1). 每个部分积均需符号扩展；
- 2). 最后一个部分积由减法运算产生，而非加法。

基 2 Booth 算法³

Booth 乘法器对上式做变换，得到以下这种形式：

$$\begin{aligned}
 [X \cdot Y]_{2c} &= [X]_{2c} \cdot (-y_7 \cdot 2^7 + y_6 \cdot 2^6 + \dots + y_1 \cdot 2^1 + y_0 \cdot 2^0) \\
 &= -y_7 \cdot 2^7 + y_6 \cdot 2^6 - y_6 \cdot 2^6 + \dots + y_1 \cdot 2^2 - y_1 \cdot 2^1 + y_0 \cdot 2^1 - y_0 \cdot 2^0 \\
 &= (y_6 - y_7) \cdot 2^7 + (y_5 - y_6) \cdot 2^6 + \dots + (y_0 - y_1) \cdot 2^1 + (y_{-1} - y_0) \cdot 2^0
 \end{aligned}$$

³ 计算机体系结构基础, 胡伟武: P210

其中, $y_{-1} = 0$ 。通过变换, 有符号乘法运算的规律性被显现出来, 最后一个部分积的产生无需再乘以 -1 (减法)。这种算法称为基 2 Booth 算法, 这是由于我们使用 2 比特产生了对应的 1 比特的“乘数”来产生部分积。

部分积的产生由“ $y_i y_{i-1}$ ”共同确定:

y_i	y_{i-1}	z_i	注释	z	neg
0	0	0	无需操作	0	0
0	1	1	+X (2 的补码)	1	0
1	0	-1	-X (2 的补码)	1	1
1	1	0	无需操作	0	0

考虑下面的例子, 计算 13x6:

	0 0 0 0 1 1 0 1	
x	0 0 0 0 0 1 1 0 0	
	<hr/>	
	0 0 0 0 0 0 0 0	(00) null
	1 1 1 0 0 1 1	(10) -X
	0 0 0 0 0 0	(11) null
	0 1 1 0 1	(01) +X
	<hr/>	
	0 0 0 0	(00) null
	<hr/>	
	1 0 1 0 0 1 1 1 0	

注意灰色的"1"并非代表真实的进位, 而是因为我省略了部分积中的部分符号位扩展 (第二个部分积), 计算结果一定是正数。

基 4 Booth 算法

让我们来讨论基 4 Booth 算法, 它是在现代 CPU 设计和 DSP 设计中最常用到的一种实现方式。

$$[X \cdot Y]_{2c} = (y_5 + y_6 - 2y_7) \cdot 2^6 + (y_3 + y_4 - 2y_5) \cdot 2^4 + \dots + (y_{-1} + y_0 - 2y_1) \cdot 2^0$$

Radix-4 Booth 算法不是将 2 个部分积划分到一起, 而是将每 3 个部分积划分到一起, 并每 2 位执行求和, 或者说, 乘数移位 2 位。基 4 意味着, 它将乘数设置为一组数字 0-3, 而不仅仅是 0,1 (基本阵列乘法, 或基 2 Booth 算法)。这会将部分积的数量减少一半, 因为您一次乘以两个二进制位。然而, 它需要乘以 3, 这是困难的。(乘以 0、1 或 2 是微不足道的, 因为它们只涉及简单的移, 但是乘 3 是困难的⁴) 为了避免乘以 3, 我们将 Booth 数字集重新编码为 2, 1, 0, -1 和 -2。

y_{i+1}	y_i	y_{i-1}	z_i	注释	z0	z1	neg
0	0	0	0	无需操作	0	0	0

⁴ <https://www.brown.edu/Departments/Engineering/Courses/En164/BoothRadix4.pdf>

0	0	1	1	+X (2's complement)	1	0	0
0	1	0	1	+X (2's complement)	1	0	0
0	1	1	2	+2X (2's complement)	0	1	0
1	0	0	-2	-2X (2's complement)	0	1	1
1	0	1	-1	-X (2's complement)	1	0	1
1	1	0	-1	-X (2's complement)	1	0	1
1	1	1	0	无需操作	0	0	0

$z0, z1, neg$ 是在本实现中使用到的信号，其中 $z0$ 意味着需要做 1 比特的移位， $z1$ 意味着需要做 2 比特的移位， neg 意味着需要做减法运算，在本设计中比特取反再加 1 (也就是该设计中的 neg 信号)。

这些选择信号的布尔表达式可以从真值表来推断，其中：

$$z0 = y_i \oplus y_{i-1}, \quad z1 = (y_{i+1} \sim y_i \sim y_{i-1}) + (\sim y_{i+1} y_i y_{i-1}), \quad neg = y_{i+1} \cdot (\sim (y_i y_{i-1}))$$

booth_encoder 模块用于实现这个逻辑，产生 booth 算法需要的这三个信号。

当我们获得了编码之后的乘数后，下一步操作是执行加法，减法，或移位操作来产生部分积送给 Wallace 树。为了实现这一逻辑，*booth_selector* 模块根据这三个信号来产生单比特的部分积。部分积的产生完全取决于被乘数的比特和编码后的乘数。

理论枯燥，让我们看个例子！

下面的例子计算有符号的 13 乘以 6：

$$\begin{array}{r}
 00001101 \\
 x 00001100 \\
 \hline
 11100101 \quad (100) -2X \\
 011010 \quad (011) +2X \\
 0000 \quad (000) \text{ null} \\
 \hline
 00 \quad (000) \text{ null} \\
 \hline
 101001110
 \end{array}$$

Step1: 给乘数添加尾随 0 得到 00000110(0);

Step2: 对 100 进行 booth 编码 (所有的 y_i 都以下划线标示了), 100 意味着 $-2X$, 也就是 $z0=0, z1=1, neg=1$;

Step3: 产生第一个部分积，被乘数需要左移移位，换句话说，当前比特的部分积使用低 1 位的数字 (这也是在本设计中实现的方法)，得到 00011010，因为 neg 是 1，所以所有比特需要反转，得到 11100101，同时需要加上 1 (也就是 neg 的值)， neg 的值被存入了一个叫做 n 的寄存器中，送往一个行波进位加法器来产生最终的 2 的补码形式的部分积。

```
RCA #(65) u_rca(
  .a      (pp[u]),
```

```

        .b    ({64'd0,n[u]}),
        .cin  (1'b0),
        .sum  (pp2c[u]),
        .cout ()
    );

```

Step4: 检查接下来的 3 个比特，011 意味着 +2X，无需做 2 的补码变换。

Step5: 继续操作，接下来两个部分乘数由 000 和 000 产生，意味着无需操作。

Step6: 将所有部分积加起来，得到 01001110 (8'd78)

你应当注意到了 0000 和 00 之间的实线，我们会在下一节讨论，这是与 Wallace 树相关的。

Booth 编码将总的加法操作次数从 N 减少到了 N/2，一半的加法运算被简单的移位和 NOP(无操作)给替代了。

Wallace 树⁵

回到上一节中的示例，实线是为 Wallace 树乘法放置的。Wallace 树将每 3 个 Booth 算法的部分积作为输入，并为下一阶段输出 2 个结果。下图示出了一种 32 位 Wallace 树，继续以这个例子为例，11100101 被发送到最右边的 CSA，同样，01101000 和 00000000 (第三个部分积)、00000000 (实线下的第四个部分积) 被送到右侧第 2 个 CSA 单元。

$$\begin{array}{r}
 00001101 \\
 \times 000001100 \\
 \hline
 11100101 \text{ } 1+1(100) -2X \\
 011010 \text{ } (011) +2X \\
 0000 \text{ } (000) \text{ null} \\
 \hline
 00 \text{ } (000) \text{ null} \\
 \hline
 101001110
 \end{array}$$

Wallace 树的结构可以通过下面的表进行推理，假设所有的操作数都是经过 booth 编码后的。

级数	独立	保留	输入	输出
0	0	2	10*3	10*2
1	1	1	7*3	7*2
2	1	0	5*3	5*2
3	0	1	3*3	3*2
4	0	1	2*3	2*2
5	0	2	1*3	1*2

⁵ 计算机体系结构基础, 胡伟武: P215

6	1	1	1*3	1*2
7	1	0	1*3	1*2

用于乘法运算的 Wallace 树的基本加法器单元有 3 个输入 a、b、cin 和 2 个输出 sum、cout。它使用 CSA (Carry Save Adder, 进位保留加法器) 而不是 RCA (Ripple Carry Adder, 行波进位加法器), 其中进位被保留以供进一步加法, 而不是行波到较高位, 从而节省了延迟。

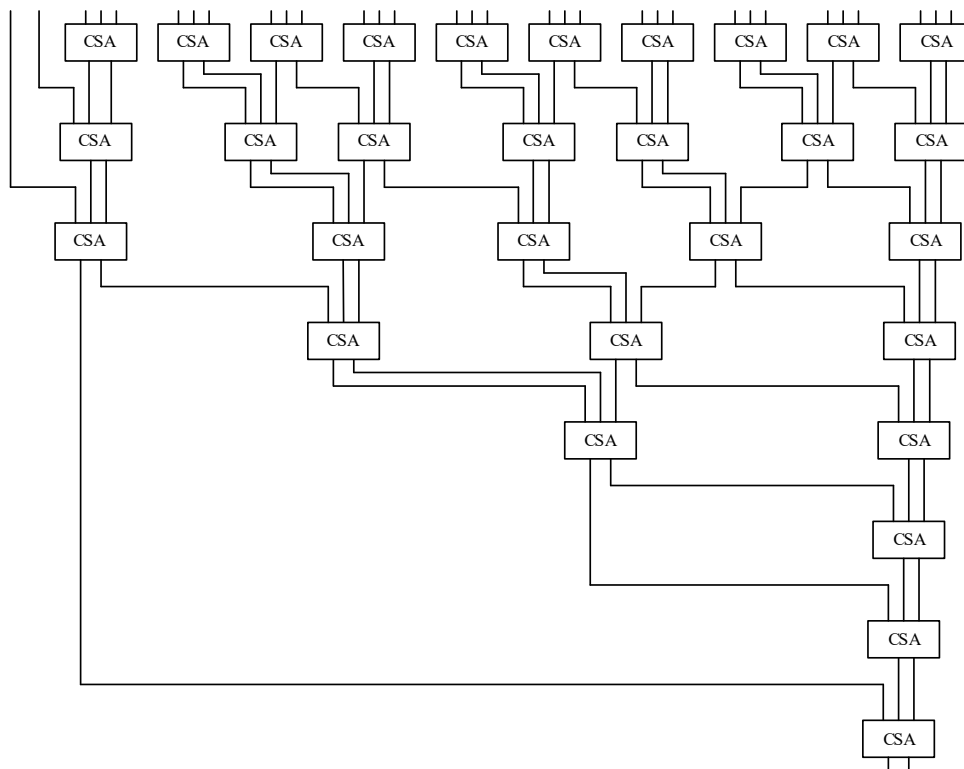


Figure 2.8 一种 32bit 的 Wallace 树

本设计中 Wallace 树的实现如上图所示, 在第二级 (为 booth 编码保留足够的建立时间裕量), 第五级, 第八级的地方插入了寄存器进行流水线处理, 如果时序不满足, 可以增大流水线级数, 我目前对此还没有非常清晰的认识。

全加器

对全加器的逻辑架构的实现进行了讨论。

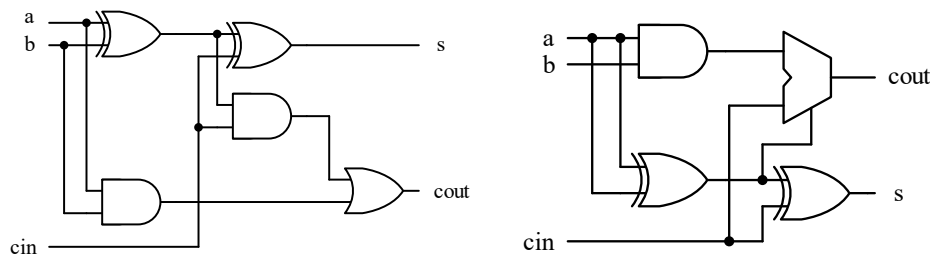


Figure 2.9 使用逻辑门构建全加器

左边的是实现全加器最基本的连接关系，其中

$$s = a \oplus b \oplus cin, \quad cout = a \cdot b + (a \oplus b) \cdot c$$

右边的是改进的实现方式，其中

$$s = a \oplus b \oplus cin, \quad cout = (a \oplus b) ? cin : (a \cdot b) = a \cdot b + (a \oplus b) \cdot cin$$

它利用了这样的特性：

$$\overline{a \oplus b} = a \cdot b + \bar{a} \cdot \bar{b}, \quad \text{and} \quad 1 + A = 1$$

考虑性能方面，左边的实现具有 3 级的逻辑门延时，而右边的实现也是 3 级 (MUX 占 2 级)。

考虑面积方面，左边的实现消耗 MOSFET 的数量为 $8+8+6+6+6=34$ ，右边消耗的数量为 $8+8+6+14=36$ 。

在 Xilinx FPGA 中，有专用的 MUX 单元，CARRY4 进位链结构使用右边的实现来节省 LUT 资源。

2.4.2 SRT 除法

概述

该部分主要参考了 *Computer arithmetic algorithms*, Koren, Israel.

SRT 除法是一种改进的非恢复除法算法，是一种慢速算法，也就是说，在单次循环中只产生 1 个比特或者固定几个比特。慢速算法使用循环公式产生剩余余数，调整商位使得循环过程收敛。

我会从恢复除法切入，也就是最基本的循环除法，然后讨论非恢复除法，再然后讨论基 2 的 SRT 除法，最终介绍基 4 的 SRT 除法，也就是 CPU 中所使用的除法器模块。

在这节开始前，有必要介绍一下基本的概念和循环公式。

我们执行以下的除法，

$$X/D = (Q, R)$$

其中, X 是被除数, D 是除数, Q 是商, R 是余数。同时我们定义 q_i, r_i 分别为第 i 个最高有效 商位, 和剩余余数 (过程余数)。

循环公式定义为:

$$r_i = \beta r_{i-1} - q_i \cdot D$$

其中, β 是基, 在基 2 算法中它是 2, 在基 4 算法中是 4。

恢复除法⁶

最终的 r_i 就是我们想要的余数, r_i 的初值是 X , 考虑 $\beta = 2$ 的情况,

$$R = 2r_{n-1} - q_n \cdot D = 2^2 r_{n-2} - 2^1 q_{n-1} D - q_n D = \dots = 2^n X - QD$$

在循环开始前, 我们必须首先把 D 左移 n 比特, 并且在循环结束后, 我们同样需要对 R 右移 n 比特, 这是一个非常重要的地方, 我们会在后续 SRT 算法中进行讨论。

算法描述如下:

```
R := X
D := D << n          -- R and D need twice the word width of N and Q
for i := n - 1 .. 0 do -- For example 31..0 for 32 bits
    R := 2 * R - D      -- Trial subtraction from shifted value
    (multiplication by 2 is a shift in binary representation)
    if R >= 0 then
        q(i) := 1      -- Result-bit 1
    else
        q(i) := 0      -- Result-bit 0
        R := R + D      -- New partial remainder is (restored) shifted value
    end
end
end
-- Where: X = dividend, D = divisor, n = #bits, R = partial remainder, q(i) = bit #i of quotient from MSB
```

商位的选择描述如下, 商位使用的数集是 $\{0,1\}$:

$$q_i = \begin{cases} 1, & \text{if } 2r_{i-1} \geq D \\ 0, & \text{if } 2r_{i-1} < D \end{cases}$$

考虑下面的例子, 计算 $13/5=(2,3)$:

⁶ https://en.wikipedia.org/wiki/Division_algorithm

r0	00001101	
D	01010000	
r1 2r0	00011010	<D q0=0
r2 2r1	00110100	<D q1=0
2r2	01101000	>=D q2=1
r3 -D	00011000	
r4 2r3	00110000	<D q3=0
R	00000011	

因此，我们得到了 $Q=0010$ ，以及 $R=0110$ ，这是正确的结果。注意，商的最高位首先产生。

术语“恢复”意味着如果 q 是 0 我们就必须恢复剩余余数。

非恢复除法

非恢复除法使用数集 $\{-1,1\}$ 而非 $\{0,1\}$ ，优点是在我们硬件实现中只有一种判断条件，并且每个商位执行一次加法/减法，没有额外的恢复过程。基本的基 2 非恢复除法描述如下：

```
R := X
D := D << n          -- R and D need twice the word width of N and Q
for i = n - 1 .. 0 do -- for example 31..0 for 32 bits
  if R >= 0 then
    q(i) := +1
    R := 2 * R - D
  else
    q(i) := -1
    R := 2 * R + D
  end if
end
-- Note: X=dividend, D=divisor, n=#bits, R=partial remainder, q(i)=bit #i of
quotient from MSB.
```

商位的选择描述如下，商位使用的数集是 $\{-1,1\}$ ：

$$q_i = \begin{cases} 1, & \text{if } 2r_{i-1} \geq D \\ -1, & \text{if } 2r_{i-1} < D \end{cases}$$

根据这个算法，得到的商是由数字 -1 和 +1 组成的非标准形式。这种形式需要进行转换来得到最终的商。例如：

0. 开始： $Q = 111\bar{1}1\bar{1}1\bar{1}$

1. 构成正项： $P = 11101010$

2. 标出负项: $M = 00010101$

3. 相减: $P-M=Q=11010101$

由该算法计算得到的商总是奇数, 余数的范围是 $-D \leq R \leq D$, 例如 $5/2=(3,-1)$ 。有时, 我们需要把余数转为正数, 只需要把 D 加到 R , 同时对 Q 减去 1 (对于无符号算法)。

基 2 SRT 除法

SRT 除法允许 0 作为商位, 这种情况下无需加法/减法运算, 数集变成了 $\{-1,0,1\}$ 。商位的选择规则变成了:

$$q_i = \begin{cases} 1, & \text{if } 2r_{i-1} \geq D \\ 0, & \text{if } -D \leq 2r_{i-1} < D \\ -1, & \text{if } 2r_{i-1} < -D \end{cases}$$

在恢复算法和非恢复算法中, 总的迭代次数是 N , 被除数的位宽。但是在 SRT 除法中, 情况有所不同, 我们对于初始的剩余余数的产生不再移位 N , 大多数的循环都是无意义的。

在 SRT 算法中, 第一步是归一化被除数和除数, 这加速了循环的过程, 总的迭代级数在这一步中被确定。

让我们来考虑一个例子, 计算 $11/3=(3,2)$, 除数和被除数都是 8 位的, $8'd11=00001011, 8'd3=00000011$ 。

$$\begin{array}{lll} r0 & 0.0010110 & n=1 \\ D & 0.1100000 & m=5 \end{array}$$

被除数 11 有 4 个前导 0, 而除数有 6 个前导 0, 我们将除数移位 5 比特得到 0.110000, 将被除数移位 1 比特得到 0.0010110。总的迭代次数是 $5-1=4$ 。也就是除数的移位数减去被除数的移位数。如果除数的移位数小于被除数的移位数, 这意味着被除数小于除数, 也就不需要进行迭代了。

归一化之后, 比较器变得更加简单, 不再需要使用全位宽的比较器。这是因为我们把 D 限制在 $[1/2,1)$ 的范围内, 也就是 0.1xxxx 和 1.0xxxx(负数的时候), 这样仅需两比特就能确定商位的值, 而不用再进行全位宽的比较。在某些情况下, 例如被除数 X 大于 $1/2$, 移位的操作会占用原有符号位的位置, 因此一共需要 3 个比特。

$$\begin{array}{lll} r0 & 0.0010110 & n=1 \\ D & 0.1100000 & m=5 \\ \hline r1 \ 2r0 & \underline{0.0101100} & \leq 1/2 \quad q1=0 \end{array}$$

第二步是进行循环，回忆循环迭代的公式： $r_i = 2r_{i-1} - q_i \cdot D$ 。循环迭代的过程在下面给出。

$$\begin{array}{rcl}
 r_0 & 0.0010110 & n=1 \\
 D & 0.1100000 & m=5 \\
 \hline
 r_1 \quad 2r_0 & \underline{0.0101100} & <1/2 \quad q_1=0 \\
 & 2r_1 & \underline{0.1011000} >=1/2 \quad q_2=1 \\
 r_2 \quad -D & 1.1111000 & \\
 r_3 \quad 2r_2 & 1.1110000 & >=-1/2 \quad q_3=0 \\
 r_4 \quad 2r_3 & 1.1100000 & >=-1/2 \quad q_4=0
 \end{array}$$

由于商的结果没有-1 作为商位，所以也无需额外转换成标准形式。但是，注意余数是负数，这不是我们想要的，因此我们需要给 r_4 加上 D ，并且给商值减去 1：

$$\begin{array}{rcl}
 r_0 & 0.0010110 & n=1 \\
 D & 0.1100000 & m=5 \\
 \hline
 r_1 \quad 2r_0 & \underline{0.0101100} & <1/2 \quad q_1=0 \\
 & 2r_1 & \underline{0.1011000} >=1/2 \quad q_2=1 \\
 r_2 \quad -D & 1.1111000 & \\
 r_3 \quad 2r_2 & 1.1110000 & >=-1/2 \quad q_3=0 \\
 r_4 \quad 2r_3 & 1.1100000 & >=-1/2 \quad q_4=0 \\
 \hline
 R \quad +D & 0.1000000 & q'=0100-1=0011
 \end{array}$$

不要忘了给 R 右移 m 位，在这个例子中是 5，我们因此得到了 00000010 (8'd2) 作为最终的余数，00000011 (8'd3) 作为最终的商。

对于有符号的情况，商位的选择有所不同，但是遵循以下规则：

$$q_i = \begin{cases} 0 & \text{if } |2r_{i-1}| < 1/2 \\ 1 & \text{if } |2r_{i-1}| \geq 1/2, \quad r_{i-1} \text{ 和 } D \text{ 符号相同.} \\ -1 & \text{if } |2r_{i-1}| \geq 1/2, \quad r_{i-1} \text{ 和 } D \text{ 符号不同.} \end{cases}$$

基 4 SRT 除法

基 4-SRT 除法和基 2-SRT 类似，几个不同之处如下：

1. 归一化.
2. 商位的选择.
3. On-the-fly 转换.

基 4 的 SRT 算法使用每 2 比特进行迭代，迭代的次数同样由移位的比特来确定。使用 m 表示除数移位的比特， n 表示被除数移位的比特，在基 2 算法中，总的迭代次数

是 $m-n$ ，但是在基 4 算法中，总的迭代次数是 $(m-n)/2$ ，并且， m 和 n 必须同样是偶数或者同样是奇数，否则会对迭代过程造成影响。

假设被除数是 23，除数是 5，这是一个无符号的例子，移位的情况如下：

$$\begin{array}{rcl} r_0 & 00.0010111 & n=0 \\ D & 00.1010000 & m=4 \end{array}$$

其中 $n=0$ ，它和基 2 的情况不同，在基 2 中，我们会选择 1 作为移位数。同时，剩余余数还需要进行符号位扩展，如果上一个剩余余数 r_{i-1} 比 $1/4$ 大，也就是 $0.01xxxx$ ，那么 $4r_{i-1}$ 将会是 $1.xxxx$ ，数字 1 掩盖了符号位。

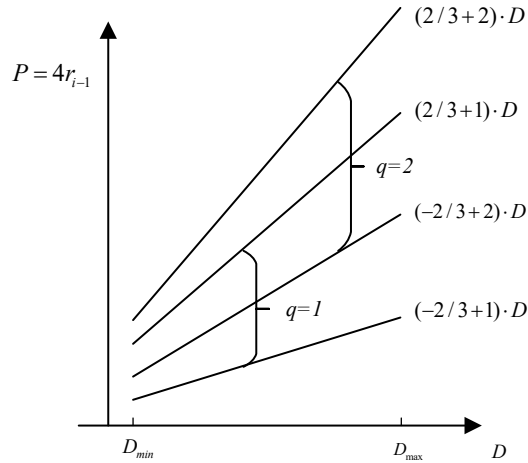
基 4-SRT 算法的商位选择相比于基 2 算法来说更为复杂，一个很大的 QDS 表 (Quotient Digit Selection) 被用来确定商值。让我们来讨论选择的规则，同时构建 QDS 表用于 RTL 实现。

在基 4-SRT 算法中，我们选择的数集是 $\{-2, -1, 0, 1, 2\}$ ，这意味着冗余度是 $k=2/(4-1)=2/3$ 。 k 是冗余度，它缩减了部分余数可允许的范围，部分余数被限定在了以下范围内 $-k \cdot D \leq r_i \leq k \cdot D$ 。

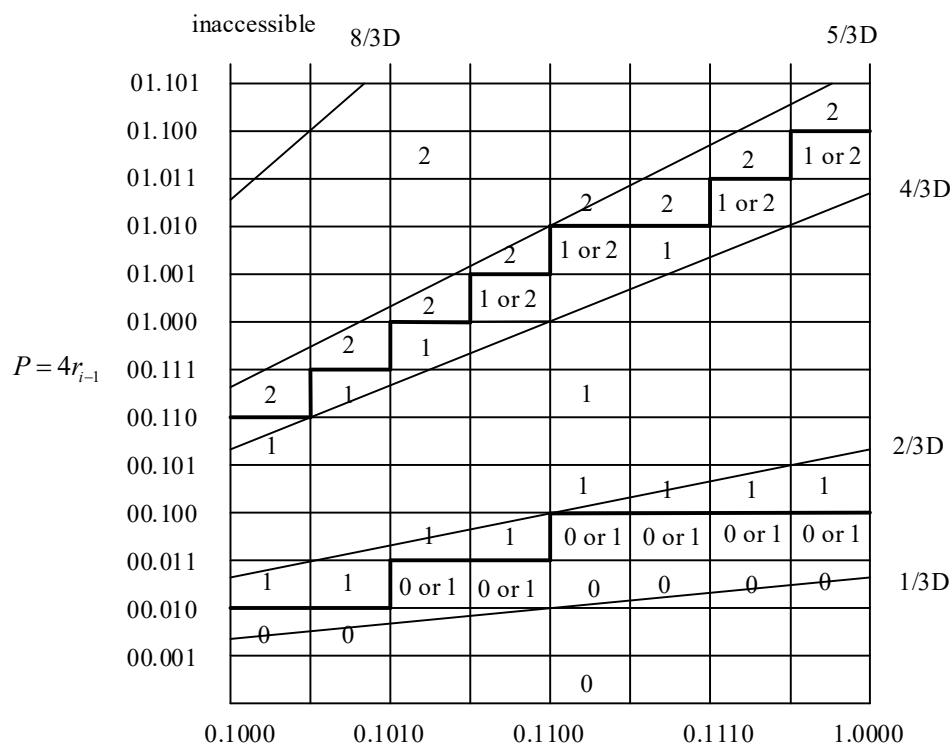
我们使用 P 来表示上一个部分余数 $4r_{i-1}$ ， $P = r_i + q \cdot D$ ，因此，

$$(-k + q) \cdot D \leq P \leq (k + q) \cdot D$$

画出 P 和 D 之间的关系，这个图被叫做 PD 图， x 和 y 坐标均是 QDS 表的输入，商值是唯一输出。



由此，我们可以构建以下的 QDS 表，多亏了归一化，我们可以仅仅检查少数几个比特来确定商值！



在这个 PD 图中，商位的选择与它左下角的坐标的是相关联的，例如， $D=0.1010$ ， $P=00.011$ ，我们应该选择 1 作为商位，然而如果 $D=0.1010$ ， $P=00.010$ ，我们应该选择 0 作为商位。

图中标注了“0 or 1”或者“1 or 2”的意味着该处的商位可以选择 0 也可以选择 1，粗实线是一种选择方案，这种选择方案允许更多的 0 和 1，更少的 2，简化了运算流程。

当我们得到了想要的商值之后，就可以继续迭代了。

r0	00.0010111	n= 0
D	00.1010000	m= 4
4r0	00.1011100	q1= 1
r1 -D	00.0001100	
4r1	00.0110000	q2= 1
r2 -D	11.1100000	
R +D	00.0110000	q'=0101-1=0100

商值的选择由 QDS 表中选择，而非简单的与 $1/2$ 做对比。在第一个循环中，P 的索引是 00.101，D 的索引是 0.1010，因此我们选择 1 作为商值，同时使用 D 去减 $4r_0$ ，得到 r_1 。

把最后的商值转换为标准形式：

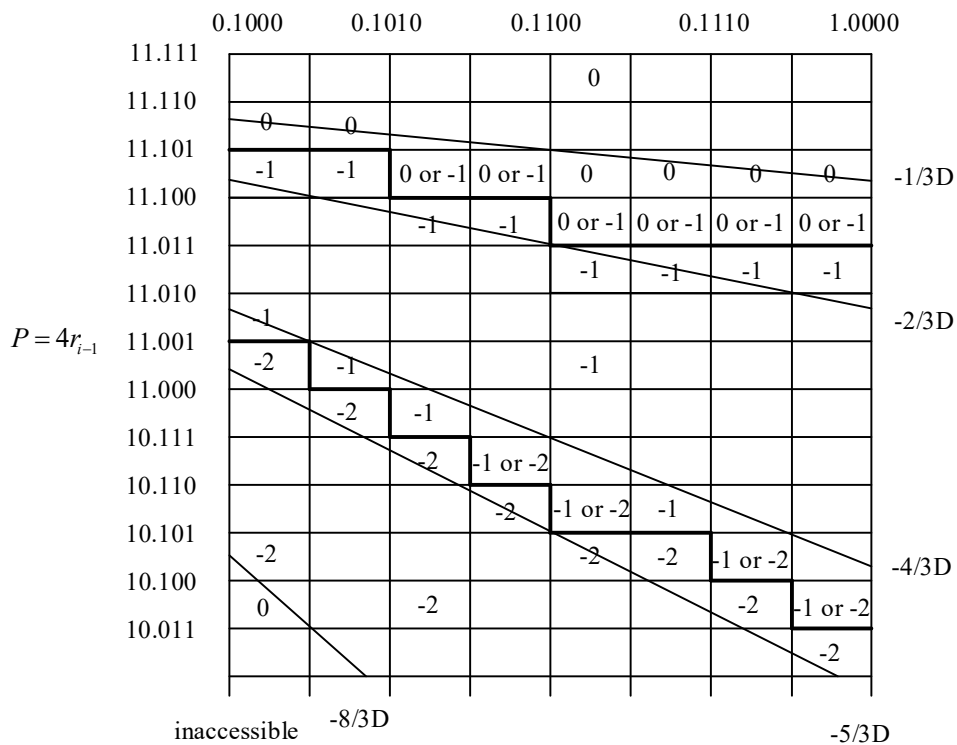
基 4 中的 11 是基 2 中的 0101，这里没有负数项，无需做减法。

余数 r_2 是负数，所有我们需要对它加上 D ，得到 00.0110000 作为最终的余数，0100 作为最终的余数。最终的余数 00.0110000 还应该右移 $m=4$ ，最终得到 00000011=3。
这个等式是 $23/5=(4,3)$ 。

本节结束了吗？我们还没有谈论负数的情况。考虑 $23/-5=(-4,3)$ ，

$$\begin{array}{rcl}
 r_0 & 00.0010111 & n=0 \\
 D & 11.0110000 & m=4 \\
 \hline
 r_1 \quad 4r_0 & 00.1011100 & q_1=?
 \end{array}$$

我们该怎么选择商值呢？首先，需要把 D 转换为正数（原码）的形式，0.1010，然后使用负数的 QDS 表进行查表。



与正数的 QDS 表不同，我们不再选择左下角的点，而是左上角的点，与商值相关联，例如，如果 $D=0.1010$, $P=11.101$ ，我们应该选择 -1 作为商值。在正数的情况下，如果 $D=0.1010$, $P=00.011$ 我们选择了 1。

以下的例子是 $23/-5=(-4, 3)$ ，过程和无符号的 SRT 算法类似。

D'	00.1010000	
r0	00.0010111	n= 0
D	11.0110000	m= 4
<hr/>		
4r0	00.1011100	q1=-1
r1 +D	00.0001100	
4r1	00.0110000	q2=-1
r2 +D	11.1100000	
<hr/>		
R -D	00.0110000	q'=-5+1=-4

需要注意的是，因为除数是负数，在后处理过程中，q 应该加上 1，R 应该减去 D (负数)。

以下的例子是 $-23/5=(-5,2)$ ：

r0	11.1101001	n= 0
D	00.1010000	m= 4
<hr/>		
4r0	11.0100100	q1=-1
r1 +D	11.1110100	
r2 4r1	11.1010000	q2= 0
+D	00.0100000	
<hr/>		
R	00.0100000	q'=-4+1=-5

注意，余数的符号没有明确的定义，例如， $-29/-26$ ，在 Matlab 中，得到 -3 作为余数，它保持余数和被除数具有相同的符号；而在 python 中，余数与除数的符号相同，在某些场合下，余数总是正的。在该实现中，余数总是正值，程序会给出的余数值是 23，因为 $-26*2+23=-29$ 。

On-the-fly 转换⁷

由于冗余数集存在负数，商的表示并非标准形式，我们需要把非标准形式的商在算法的最后一步转换为标准形式。但是它需要耗费额外的延迟以及芯片面积：对于减法操作，全位宽的进位加法器是必须的，如果使用行波进位加法器，延迟将会大到 N，如果使用超前进位加法器，需要牺牲芯片的面积。

On-the-fly 转换是为了获得实时的转换结果而设计的，它仅仅使用 2 个 Flip-Flop 和一些简单的组合逻辑就可以完成转换过程。

⁷ Ercegovac, and Lang. "On-the-fly conversion of redundant into conventional representations." IEEE Transactions on Computers 100.7 (1987): 895-897.

Q 的实际值可以表示为以下形式： $Q[j] = \sum_{i=1}^j q_i r^{-i}$ ，更新公式为： $Q[j+1] = Q[j] + q_{j+1} r^{-(j+1)}$ ，其中， $Q[j]$ 是 Q 的实际值在第 j 次迭代的结果， q_i 是商位，使用冗余表示。由于 q_{j+1} 可以是负数：

$$Q[j+1] = \begin{cases} Q[j] + q_{j+1} r^{-(j+1)} & , \text{ if } q_{j+1} \geq 0 \\ Q[j] - r^{-j} + (r - |q_{j+1}|) r^{-(j+1)} & , \text{ if } q_{j+1} < 0 \end{cases}$$

该更新公式有一个缺点，需要做减法，进位的传播会使电路变得很慢，因此我们定义另一个寄存器 $QM[j] = Q[j] - r^{-j}$ ，其中 QM 意思是 *Quotient of Minus*。

$$Q[j+1] = \begin{cases} Q[j] + q_{j+1} r^{-(j+1)} & , \text{ if } q_{j+1} \geq 0 \\ QM[j] + (r - |q_{j+1}|) r^{-(j+1)} & , \text{ if } q_{j+1} < 0 \end{cases}$$

减法操作可以被替换为对寄存器 QM 进行采样，QM 可以通过以下公式进行更新：

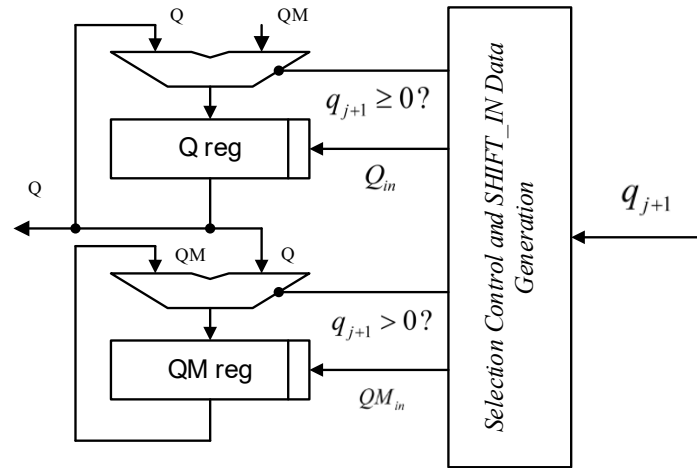
$$QM[j+1] = \begin{cases} Q[j] + (q_{j+1} - 1) r^{-(j+1)} & , \text{ if } q_{j+1} > 0 \\ QM[j] + ((r - 1) - |q_{j+1}|) r^{-(j+1)} & , \text{ if } q_{j+1} \leq 0 \end{cases}$$

项 $r^{-(j+1)}$ 可以通过将 q_{j+1} 拼接到寄存器 Q 或者 QM 的后面来实现。因此，我们得到了 On-the-fly 转换方法的更新公式：

$$Q[j+1] = \begin{cases} \{Q[j], q_{j+1}\} & , \text{ if } q_{j+1} \geq 0 \\ \{QM[j], (r - |q_{j+1}|)\} & , \text{ if } q_{j+1} \leq 0 \end{cases}$$

$$QM[j+1] = \begin{cases} \{Q[j], q_{j+1} - 1\} & , \text{ if } q_{j+1} > 0 \\ \{QM[j], (r - 1) - |q_{j+1}|\} & , \text{ if } q_{j+1} \leq 0 \end{cases}$$

它的硬件实现流程图由下图描述：



初始化条件为：

$$Q = QM = \begin{cases} \text{all 0s, if 商为正} \\ \text{all 1s, if 商为负} \end{cases}$$

我们可以简单的根据除数和被除数的符号来推断商的符号：如果被除数和除数具有相反的符号，我们就把 Q 和 QM 初始化为全 1，否则就初始化为全 0。

SHIFT_IN 的数值产生 (拼接的部分) 可以简单的通过真值表来推断，对于基 4 的情况 (基 2 的情况更简单)：

$$Q[j+1] = \begin{cases} \{Q[j], q\} & , \text{ if } q \geq 0 \\ \{QM[j], 1'b1, q[0]\} & , \text{ if } q \leq 0 \end{cases}$$

$$QM[j+1] = \begin{cases} \{Q[j], 1'b0, q[1]\} & , \text{ if } q > 0 \\ \{QM[j], \sim q\} & , \text{ if } q \leq 0 \end{cases}$$

这里也给出了一个基 2 情况下转换的例子，它把 1101(-1)00 转换为 1100100，也就是 1101000-00000100。

j	q_j	$Q[j]$	$QM[j]$
0		0	0
1	1	0.1	0.0
2	1	0.11	0.10
3	0	0.110	0.101
4	1	0.1101	0.1100
5	-1	0.11001	0.11000
6	0	0.110010	0.110001
7	0	0.1100100	0.1100011

这就是 SRT 除法的全部内容了。

3 功能描述

3.1 文件和目录结构

The figure below shows the layout of the directories in the example system.

<home directory>	Local git directory.
clean.pl	Perl script for cleaning temporary files.
└─core/	The implementation of CPU core.
└─bench/	Bench codes for CPU core.
└─rtl/	RTL codes.
└─sim/	VCS+Verdi simulation environment.
└─vsim/	Modelsim simulation environment.
└─docs/	Related documentation.

3.2 RV32I Module Design

3.3 RV32M Module Design

4 附录

4.1 附录 1: 指令集支持情况

The regularity of opcode:

inst[4:2]	000	001	010	011	100	101	110	111 (>32b)
inst[6:5]								
00	LOAD	LOAD-FP	custom-0	MISC-MEM	OP_IMM	AUIPC	OP-IM-M32	48b
01	STORE	STORE-FP	custom-1	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	reserved	custom-2/rv128	48b
11	BRANCH	JALR	reserved	JAL	SYSTEM	reserved	custom-3/rv128	>=80b

Besides, inst[1:0]=11.

The example of RISC-V pseudo instructions can be found in *risc-v specification v2.2* p109, from which we can deal with the assembly codes.

RV32I 基础指令集

合计: 47

类型	序号	指令	描述	兼容性
R-type	1	ADD	Add.	YES
	2	SUB	Subtract.	YES
	3	SLL	Shift Left Logical.	YES
	4	SLT	Set Less Than.	YES
	5	SLTU	Set Less Than Unsigned.	YES
	6	XOR	Exclusive or.	YES
	7	SRL	Shift Right Logical.	YES
	8	SRA	Shift Right Arithmetic.	YES
	9	OR	Or.	YES
	10	AND	And.	YES
I-type	11	JALR	Jump And Link Register.	YES
	12	LB	Load Byte.	YES
	13	LH	Load Halfword.	YES
	14	LW	Load Word.	YES
	15	LBU	Load Byte Unsigned.	YES
	16	LHU	Load Halfword Unsigned.	YES
	17	ADDI	Add Immediate.	YES
	18	SLTI	Set Less Than Immediate.	YES
	19	SLTIU	Set Less Than Immediate Unsigned.	YES
	20	XORI	Exclusive Or Immediate.	YES
	21	ORI	Or Immediate.	YES
	22	ANDI	And Immediate.	YES

	23	SLLI	Shift Left Logic Immediate.	YES
	24	SRLI	Shift Right Logic Immediate.	YES
	25	SRAI	Shift Right Arithmetic Immediate.	YES
S-type	26	SB	Store Byte.	YES
	27	SH	Store Halfword.	YES
	28	SW	Store Word.	YES
B-type	29	BEQ	Branch if Equal.	YES
	30	BNE	Branch Not Equal.	YES
	31	BLT	Branch Less Than.	YES
	32	BGE	Branch Greater or Equal.	YES
	33	BLTU	Branch Less Than Unsigned.	YES
	34	BGEU	Branch Greater or Equal Unsigned.	YES
U-type	35	LUI	Load Upper Immediate.	YES
	36	AUIPC	Add Upper Immediate to PC.	YES
J-type	37	JAL	Jump And Link.	YES
other	38	FENCE		NO
	39	FENCE.I		NO
	40	ECALL		NO
	41	EBREAK		NO
	42	CSRRLW		NO
	43	CSRRS		NO
	44	CSRRC		NO
	45	CSRRLWI		NO
	46	CSRRSI		NO
	47	CSRRCI		NO

RV64I 基础指令集 (加上 RV32I)

合计: 15

类型	序号	指令	描述	兼容性
R-type	1	ADDW	Add Word.	YES
	2	SUBW	Subtract Word.	YES
	3	SLLW	Shift Left Logical Word.	YES
	4	SRLW	Set Less Than Word.	YES
	5	SRAW	Set Less Than Unsigned Word.	YES
I-type	6	LWU	Load Word Unsigned.	YES
	7	LD	Load Doubleword.	YES
	8	SLLI	Shift Left Logical Immediate.	YES
	9	SRLI	Shift Right Logical Immediate.	YES
	10	SRAI	Shift Right Arithmetic Immediate.	YES
	11	ADDIW	Add Immediate Word.	YES
	12	SLLIW	Shift Left Logical Immediate Word.	YES

	13	SRLIW	Shift Right Logical Immediate Word.	YES
	14	SRAIW	Shift Right Arithmetic Immediate Word.	YES
S-type	15	SD	Store Doubleword.	YES

RV32M 标准扩展

合计: 8

类型	序号	指令	描述	兼容性
R-type	1	MUL	Multiply.	
	2	MULH	Multiply High.	
	3	MULHSU	Multiply High Signed Unsigned.	
	4	MULHU	Multiply High Unsigned.	
	5	DIV	Divide.	
	6	DIVU	Divide Unsigned.	
	7	REM	Remainder.	
	8	REMU	Remainder Unsigned.	

RV64M 标准扩展 (加上 RV32M)

合计: 5

类型	序号	指令	描述	兼容性
R-type	1	MULW	Multiply Word.	
	2	DIVW	Divide Word.	
	3	DIVUW	Divide Unsigned Word.	
	4	REMW	Remainder Word.	
	5	REMUW	Remainder Unsigned Word.	

4.2 附录 2: 运行示例

4.2.1 示例 1: 相加然后保存.

日期: 2023/7/23

哈希: a7b05c264b7f45e27a81ddc02184c6dcee29fdf9

描述: 给定两个数字, 将他们相加然后保存到内存中。

C code

```
#include "stdio.h"
int main() {
    int a = 14;
    int b = 15;
    int c;
    c = a + b;
    return 0;
}
```

Assembly code

```
addi x2 x0 14;    //0//    0000000011100000000000100010011
addi x3 x0 15;    //1//    00000000111100000000000110010011
add  x1 x2 x3;     //2//    00000000001100010000000010110011
sd   x1 8(x2);     //3//    00000000000100010011010000100011
```

4.2.2 示例 2: 累加小于该数的数.

日期: 2023/7/29

哈希: 1d28ab2a485737b8bd90fa777fd550d5183b705c

描述: 给定一个非 0 自然数 N, 计算小于 N 的所有自然数之和。

C code

```
#include "stdio.h"
int main() {
    int N = 10;
    int sum = 0;
    for(int i=1; i<N; i++){
        sum = sum+i;
    }
}
```

```
}  
    return 0;  
}
```

Assembly code

```
addi x1 x0 10;    //0// 00000000101000000000000010010011  
addi x2 x0 1;     //4// 000000000001000000000000100010011  
addi x3 x0 0;     //8// 000000000000000000000000110010011  
add  x3 x2 x3;    //12// 0000000001100010000000110110011  
addi x2 x2 1;     //16// 0000000000100010000000100010011  
blt  x2 x1 A12;   //20// 1111110000100010100110011100011  
sd   x3 8(x1);    //24// 0000000001100001011010000100011
```