(1). mul x5, x6, x7 指令----无符号数乘积和有符号数乘积的低 64 位相同

RISC-V 指令集

Multiply	`mul x5, x6, x7`	`x5 = x6 × x7`	Lower 64 bits of 128-bit product
Multiply high	`mulh x5, x6, x7`	`x5 = (x6 × x7) >> 64`	Upper 64 bits of 128-bit signed product
Multiply high, unsigned	`mulhu x5, x6, x7`	`x5 = (x6 × x7) >> 64`	Upper 64 bits of 128-bit unsigned product
Multiply high, signed-unsigned	`mulhsu x5, x6, x7`	`x5 = (x6 × x7) >> 64`	Upper 64 bits of 128-bit signed-unsigned product
Divide	`div x5, x6, x7`	`x5 = x6 / x7`	Divide signed 64-bit numbers
Divide unsigned	`divu x5, x6, x7`	`x5 = x6 / x7`	Divide unsigned 64-bit numbers
Remainder	`rem x5, x6, x7`	`x5 = x6 % x7`	Remainder of signed 64-bit division
Remainder unsigned	`remu x5, x6, x7`	`x5 = x6 % x7`	Remainder of unsigned 64-bit division

Mul 指令：mul x5, x6, x7，适用于无符号数和补码表示的带符号数，

- Case1：无符号数乘法：x5, x6, x7 都是无符号数，x5 是 128 位乘积（无符号数）的低 64 位 U64（无符号数）。
- Case2：带符号数乘法：x5, x6, x7 都是带符号数，补码表示，x5 是 128 位乘积（补码表示的带符号数）的低 64 位 V64。
  若 Case1 的 X6 与 Case2 的 X6 是相同的 bit 串，并且 Case1 的 X7 与 Case2 的 X7 是相同的 bit 串，则 Case1 和 Case2 的 X5 是相同的 bit 串。

  举例 1：上面的 64 位缩短到 8 位，A 和 B 作为补码表示的带符号数，并且都是负数, A=1011 1101=0xBD，B=1001 1110=0x9E，mul C, A，B
- Case1：无符号数乘法：C, A, B 都是无符号数，C 是 16 位乘积（无符号数）的低 8 位（无符号数），A*B=0xBD *0x9E =0X74A6, C=0xA6
- Case2：带符号数乘法：C, A, B 都是带符号数，补码表示，C 是 16 位乘积（补码表示的带符号数）的低 8 位。A*B= -0x43 *(-0x62)= +0x19A6, C=0xA6
- 结论：A 和 B 作为带符号数，并且都是负数时，结论正确：无符号数乘法和带符号数乘法的乘积的低 8 位 C 相同，都是 0xA6。

  举例 2：上面的 64 位缩短到 8 位，A 和 B 作为带符号数时是 1 正 1 负，负数 A=1011 1101=0xBD，正数 B=0001 1110=0x1E，mul C, A，B
- Case1：无符号数乘法：C, A, B 都是无符号数，C 是 16 位乘积（无符号数）的低 8 位（无符号数），A*B=0xBD *0x1E =0X1626, C=0x26
- Case2：带符号数乘法：C, A, B 都是带符号数，补码表示，C 是 16 位乘积（补码表示的带符号数）的低 8 位。A*B= -0x43 *0x1E=-0x7DA, A*B 的补码表示是 0xF826，C=0x26
- 结论：A 和 B 作为带符号数，并且 1 正 1 负时，结论正确：无符号数乘法和带符号数乘法的乘积的低 8 位 C 相同，都是 0x26。

========== ========== 数学证明 ======== ==========

（1）A 和 B 看做带符号数时都是负数，上面的 64 位缩短到 8 位，A 和 B 作为带符号数,补码表示，都是负数，例如 A 表示为 1011 1101=0xBD，A 的数值是 A=-0x43, B 表示为 1001 1110=0x9E, B 的数值是 B=-0x62；把 A 和 B 看作无符号数时，都是正数，例如 A 表示为 1011 1101=0xBD~~，这时 A 的数值记为 A0，A0=+0xBD, B 表示为 1001 1110=0x9E, B 的数值记为 B0，B0=+0x9E~~；作为无符号数的 A 和 B 的数值分别记为 A0 和 B0, A0=+0xBD, B0=+0x9E。mul C, A，B

- Case1：无符号数乘法：C, A, B 都是无符号数，C 是 16 位乘积（无符号数）的低 8 位（无符号数）, C=（A0*B0）% 256
- Case2：带符号数乘法：C, A, B 都是带符号数，补码表示，C 是 16 位乘积（补码表示的带符号数）的低 8 位。A*B = abs(A)*abs(B) = (A 的取反再加 1) * (B 的取反再加 1) = (256-A0)*(256-B0) =256*256-256*(A0+B0)+A0*B0, C=（A*B）% 256= (256*256-256*(A0+B0)+A0*B0) % 256 = （A0*B0）% 256, 这里 abs 是求绝对值的函数。
- 结论：A 和 B 作为带符号数，并且都是负数时，结论正确：无符号数乘法和带符号数乘法的乘积的低 8 位 C 相同，都是（A0*B0）% 256。

（2）A 和 B 看做带符号数时一正一负，上面的 64 位缩短到 8 位，A 和 B 作为带符号数时是 1 正 1 负，负数 A=1xxx xxxx，正数 B=0xxx xxxx，作为无符号数的 A 和 B 的数值分别记为 A0 和 B0，例如 A0=+0xBD, B0=+0x1E。mul C, A，B

- Case1：无符号数乘法：C, A, B 都是无符号数，C 是 16 位乘积（无符号数）的低 8 位（无符号数）, C=（A*B）% 256
- Case2：带符号数乘法：C, A, B 都是带符号数，补码表示，C 是 16 位乘积（补码表示的带符号数）的低 8 位。A*B= - abs(A) *B = -(256-A0)* B0，-(256-A0)* B0 是负数, abs(A*B)=(256-A0)* B0 是正数，A*B 的补码表示(例如 0xF826), 是 2^16- abs(A*B)=2^16-[(256-A0）* B0] = 2^16- 256*B0 + A0* B0=（256-B0）*256+A0*B0，C=（A*B 的补码表示）% 256 = [(256-B0）*256+A0*B0] % 256 = （A0*B0）% 256
- 结论：A 和 B 作为带符号数，并且 1 正 1 负时，结论正确：无符号数乘法和带符号数乘法的乘积的低 8 位 C 相同，都是（A0*B0）% 256。

（3）A 和 B 看做带符号数时都是正数，上面的 64 位缩短到 8 位，执行 mul C, A，B 时，作为无符号数乘法和带符号数乘法的乘积 A*B 都相同，乘积的低 8 位 C 相同，都是（A*B）% 256。

(2). 乘法的每轮加法有进位

```
    1110
  * 1011
-----------
   1110
  1110
 0000           10101
1110
-----------
```

(3). P115 的机器码有错

P64 fig. 2.1

P115 有 BNE 指令的机器码说明

```
bne  x10, x11, 2000  // if x10 != x11, go to location 2000ten = 0111 1101 0000
```

could be assembled into this format (it's actually a bit more complicated, as we will see):

0	111110	01011	01010	001	1000	0	1100111
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode

上面红框内 1100111 是错的，改成 1100011，依据，【RISC-V 手册】RISC-V-Reader-Chinese-v2p1.pdf】P125/总 164 页

(4). JAL、SB 类型的指令（BEQ，BNE，BLT，BLTU，BGE，BGEU）的机器码提供的位移量*2 再加上 PC

JAL 指令的位移量是 20 位补码表示的 imm-jal，SB 类型的指令位移量是 12 位补码表示的 imm-sb，这 2 类指令的位移量都要*2（即右边补一位 0）再加上 PC。

书 P115 有 BNE 指令的机器码说明，见下图，红框内机器码 1100111 是错的，改成 1100011，依据：【RISC-V 手册】（RISC-V-Reader-Chinese-v2p1.pdf）P123/总 162 页。 SB 类型的指令（BEQ，BNE，BLT，BLTU，BGE，BGEU），指令机器码仅提供了 imm[12:1]共 12 位，最右边应补 1 位 0（等价于左移 1 位，即下面划红线文字所说的乘 2），变成 13 位，是 13 位补码 2 进制数，范围-4096 到+4094。

Addressing in Branches

The RISC-V branch instructions use the RISC-V instruction format called SB-type. This format can represent branch addresses from −4096 to 4094, in multiples of 2. For reasons revealed shortly, it is only possible to branch to even addresses. The SB-type format consists of a 7-bit opcode, a 3-bit function code, two 5-bit register operands (rs1 and rs2), and a 12-bit address immediate. The address uses an unusual encoding, which simplifies datapath design but complicates assembly. The instruction

```
bne  x10, x11, 2000  // if x10 != x11, go to location 2000ten = 0111 1101 0000
```

could be assembled into this format (it's actually a bit more complicated, as we will see):

0	111110	01011	01010	001	1000	0	1100111
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode

JAL 指令 和 SB 指令的源程序指令后的位移量是机器码中的位移量*2 后的值。例如上面 bne 指令【bne x10, x11, 2000】的位移量的 12 位机器码是 0-0-111110-1000（等于 10 进制 1000 或 0x3E8）．依据如下图：

Conditional branch	Branch if equal	`beq x5, x6, 100`	`if (x5 == x6) go to PC+100`	PC-relative branch if registers equal
	Branch if not equal	`bne x5, x6, 100`	`if (x5 != x6) go to PC+100`	PC-relative branch if registers not equal
	Branch if less than	`blt x5, x6, 100`	`if (x5 < x6) go to PC+100`	PC-relative branch if registers less
	Branch if greater or equal	`bge x5, x6, 100`	`if (x5 >= x6) go to PC+100`	PC-relative branch if registers greater or equal
	Branch if less, unsigned	`bltu x5, x6, 100`	`if (x5 < x6) go to PC+100`	PC-relative branch if registers less, unsigned
	Branch if greater or equal, unsigned	`bgeu x5, x6, 100`	`if (x5 >= x6) go to PC+100`	PC-relative branch if registers greater or equal, unsigned
Unconditional branch	Jump and link	`jal x1, 100`	`x1 = PC+4; go to PC+100`	PC-relative procedure call
	Jump and link register	`jalr x1, 100(x5)`	`x1 = PC+4; go to x5+100`	Procedure return; indirect call

FIGURE 2.1 (Continued).

(5). 0pcode 在内存低地址

Opcode 在内存低地址

注意：RISC-V 只支持小端格式(little-endian)。小端格式和大端格式的对比如图 1 所示。如果使用不同的端序存储同样的 32 位数 0x0A0B0C0D，情况如图所示。

小端的最高地址字节是 0x0A，最低地址字节是 0x0D；

大端的最高地址字节是 0x0D，最低地址字节是 0x0A。

如果在用不同端序的系统中数据交流，需要确保传输的数据是以 32 位数为单位。如果以一个字节为单位，在不同的端序系统中交换数据，可能会出现问题，比如说这里小端的 0x0A 对应的地址(a + 3)，在大端的系统中，该地址存储的数据是 0x0D。

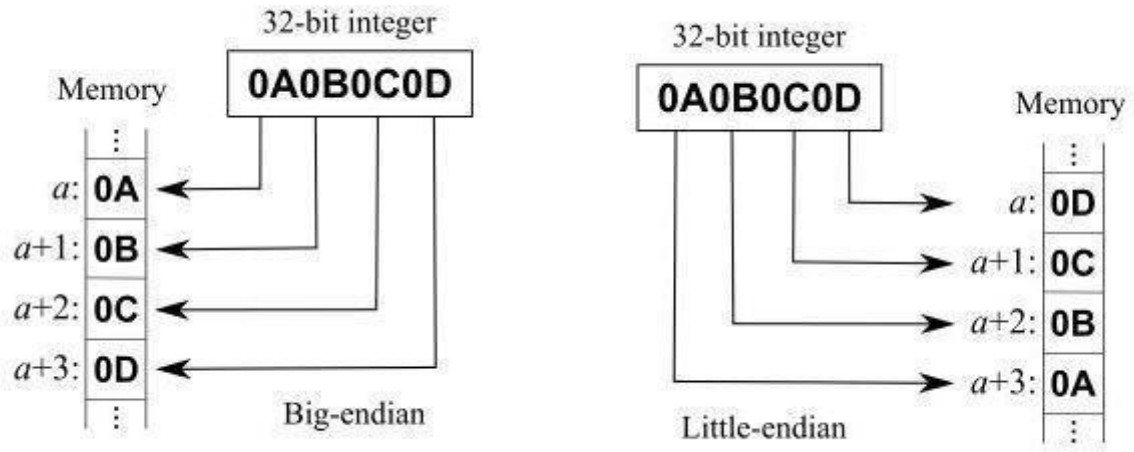


图 1. 大端格式，小端格式

RISC-V 选择以小端格式为端序是因为其目前在商业上占主导地位。所有的 X86-32 系统和 Apple iOS，谷歌 Android 操作系统和微软 Windows for ARM 都是采用小端格式地址排序(低位字节在低地址，优先发送，例如 64 个字节的 1 块数据（假设地址为 100--163）从计算机 A 发送到计算机 B，地址 100 的数据优先送出去)。

另外也可以看以下网文：

https://stackoverflow.com/questions/54693866/riscv-resolving-opcode

(6). 本书一律在时钟上升沿更新寄存器

In this book, we use the rising clock edge to update register.书 P242，课件：第四章 P14

(7). asserted-assert-deasserted-deassert 详解

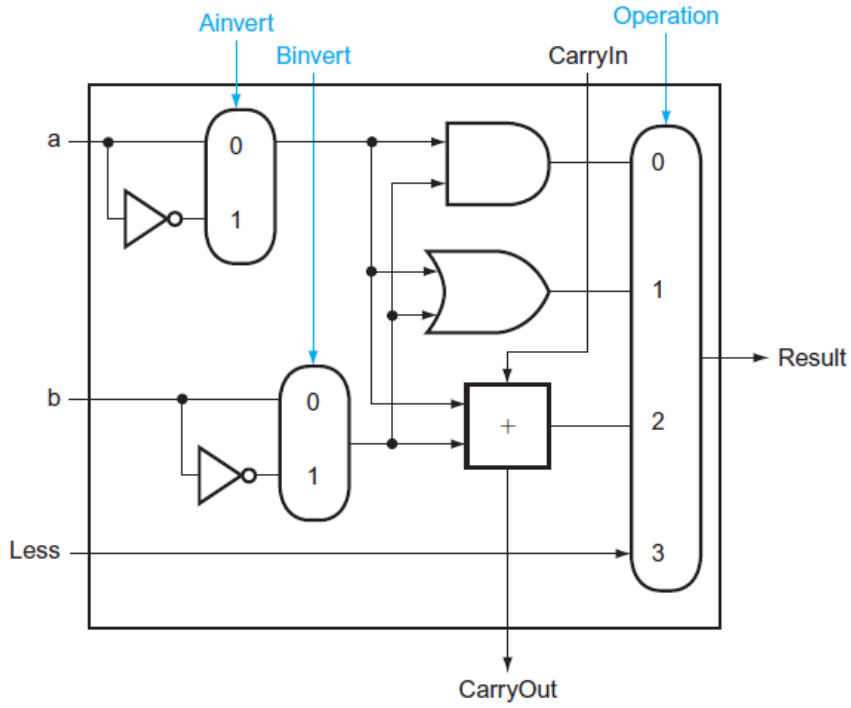
asserted (有效，逻辑 1，即 active 状态，如果某引脚是高电平有效，该引脚为 asserted 指该引脚为高电平；如果某引脚是低电平有效，该引脚为 asserted 指该引脚为低电平)

assert：变有效，变成逻辑 1，to become active。如果某引脚是高电平有效，该引脚 assert，指该引脚变为高电平；如果某引脚是低电平有效，该引脚 assert，指该引脚变为低电平)

deasserted(无效，逻辑 0，即 inactive 状态）

deassert：变无效，变成逻辑 0，to become inactive。

(8). RiscV-第五版英文附录 A ALU 我已看懂，下面是解释



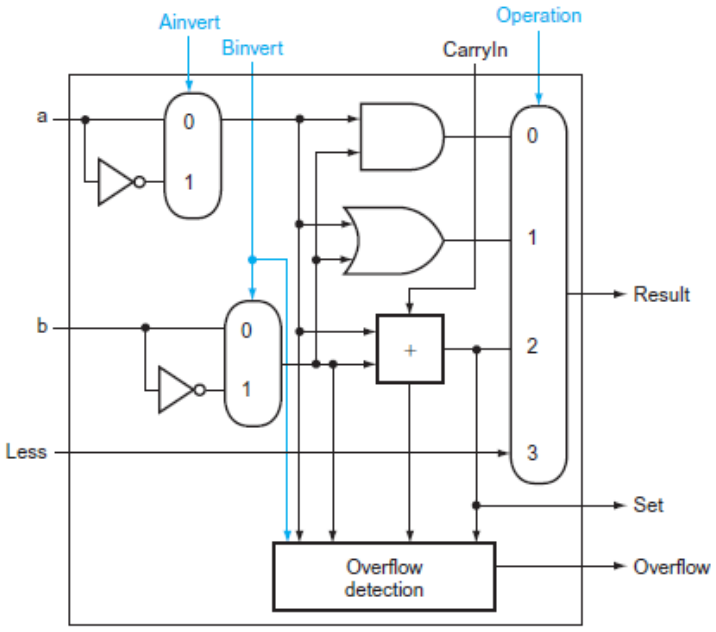


FIGURE A.5.10 (Top) A 1-bit ALU that performs AND, OR, and addition on a and b or $\bar{b}$, and (bottom) a 1-bit ALU for the most significant bit. The top drawing includes a direct input that is

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set less than
1100	NOR

FIGURE A.5.13 The values of the three ALU control lines, Ainvert, Bnegate, and Operation, and the corresponding ALU operations.

上表左边的 4 个输入信号：Ainvert, Bnegate, 2 位的 Operation。上表的 AND、OR、add 功能对应电路图 fig-1.5.10 和 Fig-a.5.11 很容易理解，其他 3 个解释如下：

- subtract：bnegate=1，ALU0 的进位 C0 为 1，ALU 执行：A + B 的取补）= A+/B+1 =A-B，这里 A、B 都是 64 位数。
- set less than: ALU 执行 A-B，相减结果的最高位所在的 1 位 ALU（ALU63）内部加法器的输出是符号位，该符号位连接信号 Set63，当且仅当 A<B 时，Set63=1(否则 0) ==> ALU0 的 Less 就是 Set63，这时 operation=11，每个 1 位 ALU 选择输入 Less 作为 4 路 MUX 的输出，ALU0 的 result0=Set63，而 ALU1-63 的 result1-63 是 63 位 0，所以 result0-63 = Set63（0 或 1），当 A<B 时，Set63=1(否则 0)。
- NOR：这时 operation=00，每个 ALU 执行 AND 操作，result(i)=~a(i) and ~b(i)=~ [a(i) or b(i)], 64 位结果是 ~(A or B)

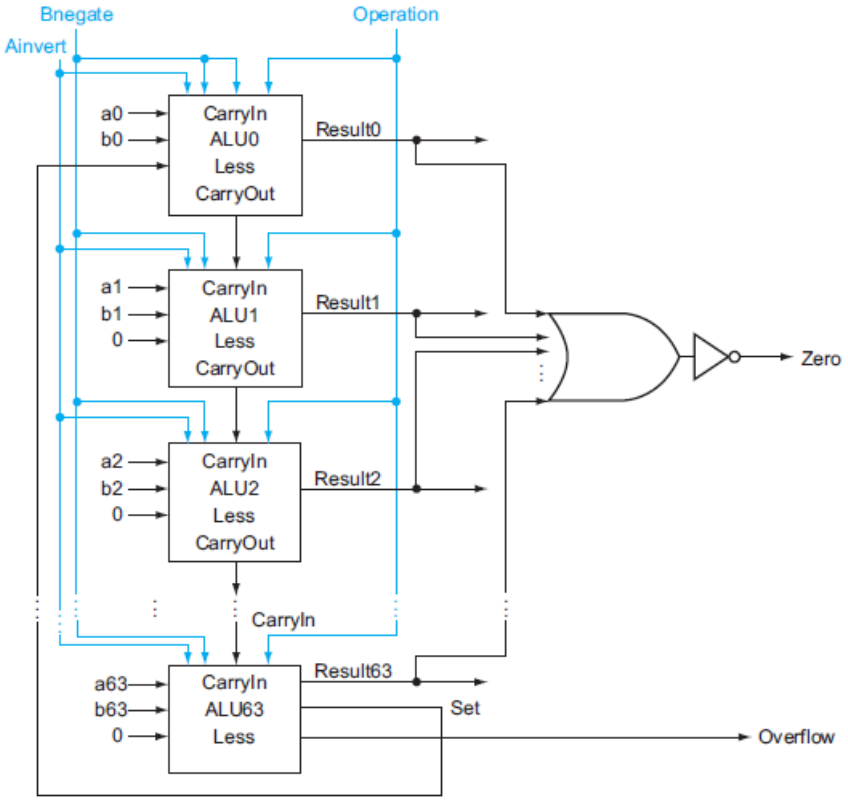


FIGURE A.5.12 The final 64-bit ALU. This adds a Zero detector to Figure A.5.11.

========== 完 ==========