



数字信号处理的 VLSI 设计

32 位最大公约数处理器设计

17212020116 张秉昇
17212020116 顾晨昊
17212020116 叶汉辰

November 10, 2017

Contents

1	设计指标	3
1.1	设计目标	3
1.2	输入	3
1.3	输出	3
1.4	要求	3
1.5	总模块框图	3
2	算法	4
2.1	欧几里得算法	4
2.1.1	算法概述	4
2.1.2	公式表述	4
2.1.3	代码表述	4
2.2	stein 算法	4
2.2.1	stein 算法流程	5
2.2.2	stein 算法代码	5
2.3	改进的 stein 算法	6
2.3.1	gcd 和 stein 算法分析	6
2.3.2	改进的 stein 算法	8
3	高性能设计	9
3.1	设计综述	9
3.2	RTL 电路设计	9
3.2.1	数据输入模块	10
3.2.2	流水线计算模块	10
3.2.3	数据输出模块	12
3.3	功能验证	13
3.4	Vivado 综合, 功耗、资源占用与时序分析	13
3.5	基于 SoC 的验证平台	15
3.5.1	SoC 验证代码	17
4	低功耗设计	19
4.1	设计综述	19
4.2	状态机设计	19
4.3	两种低功耗设计	20
4.4	register case	20
4.4.1	比较与交换数据模块	21
4.4.2	对齐模块	21
4.4.3	减法求绝对值模块	23
4.5	logic case 设计	24
4.6	功能验证	24
4.7	DC 综合, 功耗, 面积与时序分析	25
4.8	能效比分析	27
5	总结	28

6 附录

28

1 设计指标

1.1 设计目标

设计一个计算两个 32 位整数最大公约数 (GCD) 的处理器。

1.2 输入

- OPA: 32-bit, 操作数一;
- OPB: 32-bit, 操作数二;
- START: 启动信号;
- RESET: 复位信号;
- CLK: 系统时钟。

1.3 输出

- DONE: 指示输出。
- RESULT: 32-bit, 最大公约数 (GCD);

1.4 要求

- 采用二元的最大公约数算法, 即欧几里德算法 (Euclidean GCD Algorithm);
- 基于功耗优先 (Power-Efficient) 与性能优先 (performance-Efficient) 的原则, 设计两种不同的 VLSI 实现方案; 其中, 一种方案使用 Altera 公司的 FPGA 设计流程实现, 另一种方案使用标准的 ASIC 实现流程

1.5 总模块框图

整个系统的模块框图如下所示:

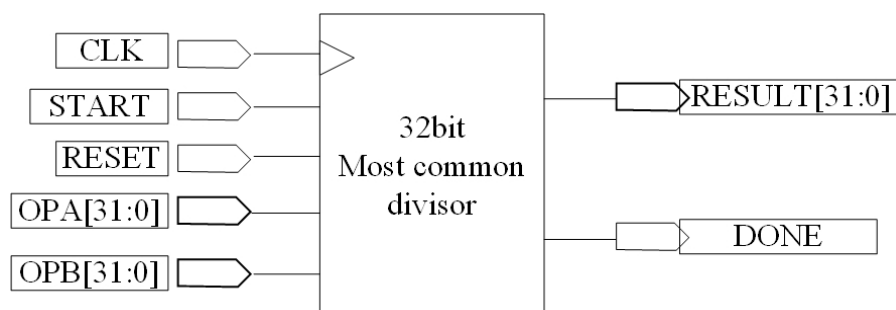


Figure 1: 总模块框图

2 算法

2.1 欧几里得算法

2.1.1 算法概述

欧几里得算法又称辗转相除法，用于计算两个整数 a, b 的最大公约数。其计算原理依赖于下面的定理： \gcd 函数就是用来求 (a, b) 得最大公约数的。 \gcd 函数的基本性质：

$$\gcd(a, b) = \gcd(b, a) = \gcd(-a, b) = \gcd(|a|, |b|) \quad (1)$$

2.1.2 公式表述

$$\gcd(a, b) = \gcd(b, a \bmod b)$$

证明： a 可以表示成为 $a = kb + r$ ，则 $r = a \bmod b$

假设 d 是 a, b 的一个公约数，则有

$d|a, d|b$ ，而 $r = a - kb$ ，因此 $d|r$

因此 d 是 $(b, a \bmod b)$ 的公约数

假设 d 是 $(b, a \bmod b)$ 的公约数，则

$d|b, d|r$ ，但是 $a = kb + r$

因此 d 也是 (a, b) 的公约数

因此 (a, b) 和 $(b, a \bmod b)$ 的公约数是一样的，其最大公约数也必然相等，得证。

2.1.3 代码表述

欧几里得算法用 C 语言代码表述如下，假设 $a > b$ ：

```

1 int gcd(int a, int b)
2 {
3     if(b == 0)
4         return a;
5     return
6         gcd(b, a%b)
7 }
```

2.2 stein 算法

Stein 算法是一种计算两个数最大公约数的算法，是针对欧几里德的算法，是针对欧几里德算法在对大整数进行运算时，需要试商导致增加运算时间的缺陷而提出的改进算法。

欧几里德算法是计算两个数最大公约数的传统算法，无论从理论还是从实际效率上都是很好的。但是却有一个致命的缺陷，这个缺陷在素数比较小的时候一般是感觉不到的，只有在大素数时才会显现出来。

一般实际应用中的整数很少会超过 64 位（当然现在已经允许 128 位了），对于这样的整数，计算两个数之间的模是很简单的。对于字长为 32 位的平台，计算两个不超过 32 位的整数的模，只需要一个指令周期，而计算 64 位以下的整数模，也不过几个周期而已。但是对于更大的素数，这样的计算过程就不得不由用户来

设计，为了计算两个超过 64 位的整数的模，用户也许不得不采用类似于多位数除法手算过程中的试商法，这个过程不但复杂，而且消耗了很多 CPU 时间。对于现代密码算法，要求计算 128 位以上的素数的情况比比皆是，设计这样的程序迫切希望能够抛弃除法和取模。

2.2.1 stein 算法流程

1. 如果 $A_n = B_n$, 那么 A_n (或 B_n)* C_n 是最大公约数，算法结束
2. 如果 $A_n = 0$, B_n 是最大公约数，算法结束
3. 如果 $B_n = 0$, A_n 是最大公约数，算法结束
4. 设置 $A_1 = A$, $B_1 = B$ 和 $C_1 = 1$
5. 如果 $A_{n+1} = A_n/2$, $B_{n+1} = B_n/2$, $C_{n+1} = C_n * 2$ (注意，乘 2 只要把整数左移一位即可，除 2 只要把整数右移一位即可)
6. 如果 A_n 是偶数， B_n 不是偶数，则 $A_{n+1} = A_n/2$, $B_{n+1} = B_n$, $C_{n+1} = C_n$
7. 如果 B_n 是偶数， A_n 不是偶数，则 $B_{n+1} = B_n/2$, $A_{n+1} = A_n$, $C_{n+1} = C_n$
8. 如果 A_n 和 B_n 都不是偶数，则 $A_{n+1} = |A_n - B_n|/2$, $B_{n+1} = \min(A_n, B_n)$, $C_{n+1} = C_n$
9. n 加 1，转 1

2.2.2 stein 算法代码

```

1  #define CHECK(a) (!(1&(a)))
2  #define CLEAN2(a) while(CHECK(a))a=a>>1
3  #define BIGERA if(a > b)(t = a, a = b, b = t)
4
5  int gcd(int a, int b)
6  {
7      int c_2 = 0, t;
8      while ((CHECK(a)) && (CHECK(b)))
9      {
10         a = a >> 1;
11         b = b >> 1;
12         c_2++;
13     }
14     CLEAN2(a);
15     CLEAN2(b);
16     BIGERA;
17     while (a = ((a - b) >> 1))
18     {
19         CLEAN2(a);
20         BIGERA;
21     }

```

```

22   return b << c_2;
23
24 }
```

2.3 改进的 stein 算法

2.3.1 gcd 和 stein 算法分析

gcd 算法具有算法简单, 通过简单递归就能实现的特点。但是 gcd 算需要递归地用到除法, 32-bit 的除法在硬件设计中的开销非常大。在硬件实现上具有以下缺点:

- 关键路径长, 需要多级流水线切割
- 面积大, 在实现高性能的多级流水线比较困难

stein 算法通过运用了最大公约数的性质, 使用了简单的减法和位移的操作, 在硬件实现上比较简单, 而且对于 32-bit 的操作数, 最大的操作数每次迭代都会衰减 1/2, 这样整个算法最多只需要 63 次就能完成迭代。所以 stein 算法迭代周期少, 控制简单非常适合做高性能的计算。

但是对于一些特殊的情况仍然需要改善。由于算法的结束的条件是 $A_n = B_n, A_n = 0, B_n = 0$ 。所以例如当 $A = 100001, B = 1$ 时需要多次迭代才能完成计算。计算过程如下:

start:

1. $A_0 = 100001, B_0 = 1$
2. $A_1 = (A_0 - B_0)/2 = 50000, B_1 = 1$
3. $A_2 = A_1/2 = 25000, B_2 = 1$
4. $A_2 = A_1/2 = 25000, B_2 = 1$
5. $A_3 = A_2/2 = 12500, B_3 = 1$
6. $A_4 = A_3/2 = 6250, B_4 = 1$
7. $A_5 = A_4/2 = 6250, B_5 = 1$
8. $A_6 = A_5/2 = 3125, B_6 = 1$
9. $A_7 = (A_6 - B_6)/2 = 1562, B_7 = 1$
10. $A_8 = A_7/2 = 781, B_8 = 1$
11. $A_9 = (A_8 - B_8)/2 = 390, B_9 = 1$
12. $A_{10} = A_9/2 = 195, B_{10} = 1$
13. $A_{11} = (A_{10} - B_{10})/2 = 97, B_{11} = 1$
14. $A_{12} = (A_{11} - B_{11})/2 = 48, B_{12} = 1$

15. $A_{13} = A_{12}/2 = 24, B_{13} = 1$
16. $A_{14} = A_{13}/2 = 12, B_{14} = 1$
17. $A_{15} = A_{14}/2 = 6, B_{15} = 1$
18. $A_{16} = A_{15}/2 = 3, B_{16} = 1$
19. $A_{17} = (A_{16} - B_{16})/2 = 1, B_{16} = 1$

end

所以计算比较直观的 $A = 100001, B = 1$ 两个数的最大公约数就需要 19 次迭代。这样大大浪费了计算时间和计算资源，在低功耗设计中是不可取的。因为在低功耗设计中需要做尽可能少的运算。

而且为了更加精确的对比 gcd 算法的迭代周期和 stein 算法的迭代中，我们小组进行了精确的 matlab 仿真分析，仿真脚本和结果在 *ASIC-euclidean\matlabmodel* 里面。我们在 $(0, 2^{32} - 1)$ 范围之内，随机生成了 1000 万对随机整数，分别用 gcd 算法和 stein 算算法来求解最大公约数，并且统计平均计算周期，最后结果如表1所示。

gcd	stein
18.2592	42.7365

Table 1: gcd 与 stein 算法平均计算迭代次数

其中，stein 算法的平均迭代次数达到了 42.7365 次，远远高于欧几里得算法 (gcd) 算法的平均迭代次数 18.2592 次。经过统计直方图分析，迭代次数的分布统计如下图27和图28所示：

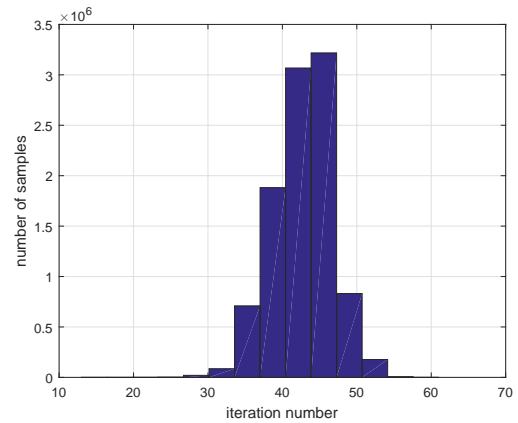
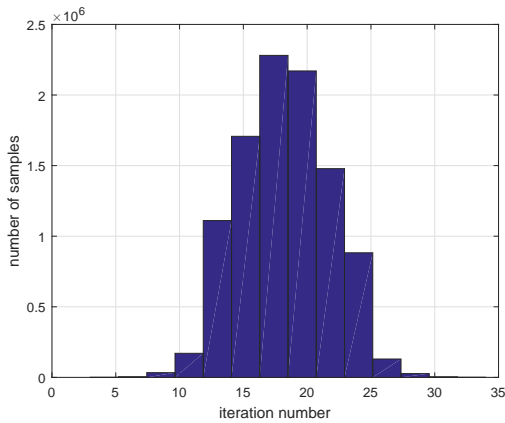


Figure 2: gcd 算法迭代次数统计直方图 Figure 3: stein 算法迭代次数统计直方图

对于 32-bit 的数，欧几里得算法的迭代次数大都分布在 (12, 25) 之间，算法的迭代周期大都分布在 (35, 50) 之间。所以 stein 算法的平均迭代周期约为 gcd 算法的 2 倍多。

2.3.2 改进的 stein 算法

在上一小节中，我们说到了 stein 算法非常适合用来做硬件实现，因为 stein 算法只包含了简单的奇偶判断、移位、相减、比较大小等计算。相比较与原始的欧几里得算法有了较大的改善。但是我在上一节，也发现一个问题就是 stein 算法的迭代周期比欧几里得算法的迭代次数要长。造成这个问题的原因经过我们分析为：

当两个操作数相差较大时，stein 算法对于较大的操作数不能有效的衰减。例如，对于一组操作数 (1000000001,1)，在 stein 算法中迭代时，下一组操作数为 (500000000,1)，而在 gcd 算法中迭代时，下一组操作数位 (1,1)。所以 gcd 算法对于相差较大的操作数能够有效衰减，而 stein 只能以约为 $1/2$ 的衰减速率来衰减。

因此，我们分析了 stein 算法不足的原因所在。所以在算法设计中，我们小组对 stein 算法不能有效处理两个相差较大操作数的缺陷进行改进——当两个操作数相差较大的时候，我们增大较小操作数的数量级，使得其与较大操作数的数量级一致，然后再进入下一次迭代。，是数量级一致的操作，通过硬件中简单的操作数移位就可以实现。优化的 stein 算法的伪代码如算法1所示：

Algorithm 1 Framework of optimized stein algorithm

Input: opa,opb

Output: result

```

1: while (opa  $\neq$  opb)and(opa  $\neq$  0)and((opb  $\neq$  0)) do
2:   opa = max(opa, opb), opb = min(opa, opb) ;
3:   lengthOFopa = binaryLen(opa), lengthOFopb = binaryLen(opb);
4:   lengthdiff = lengthOFopa - lengthOFopb;
5:   opa = abs(opa - opb << lengthdiff);
6: end while
7: if opa = opb then
8:   result = opa
9: end if
10: if opa = 0 then
11:   result = opb
12: end if
13: if opb = 0 then
14:   result = opa
15: end if

```

再对原来的 stein 算法优化的基础上，我们在 matlab 中对优化的 stein 进行仿真，同样的，我们随机取得 1000 万对随机数，得到平均迭代次数如表2所示为 23.7273 次。

optimized stein
23.5273

Table 2: optimized stein 算法平均计算迭代次数

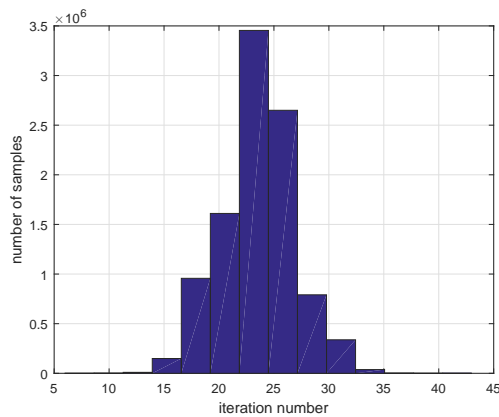


Figure 4: 优化的 stein 算法迭代次数统计直方图

我们通过优化的 stein 算法，在原来的 stein 算法的基础上将迭代中期的次数从 42.7365 减少到了 23.7273 次，与 gcd 算法的平均 18.2592 次迭代相接近。而且优化的 gcd 算法包含了**选择判断有效位数，比较，移位，减法**操作，省去了复杂的除法操作。由于复杂的除法操作在硬件实现中不仅面积开销大，而且功耗高。所以优化的 stein 算法非常适合与低功耗的设计。

3 高性能设计

3.1 设计综述

经过第二章对算法的分析，在高性能的设计中，我们采用了 stein 算法，根据计算 stein 算法最多需要 63 次迭代完成 gcd 计算。为了尽可能提高 gcd 处理器的数据吞吐率，我们将 63 次迭代展开为 63 级流水线，同时尽可能减小每一级流水线的计算延迟，提高运行频率。数据输入模块和数据输出模块分别构成一级流水线，故总流水线级数为 65 级。

在设计过程中，为了降低每一级流水线的计算延迟，我们考虑过将每一次迭代进行进一步的流水线划分，构成 126 或更多级流水线，但实际设计和测试后，发现该做法并不能取得很好的收益。该部分将在 RTL 电路设计的流水线计算模块一节详细阐述。

3.2 RTL 电路设计

高性能设计的电路设计文件在 hp/rtl/文件夹中，其中包含三个子模块和一个顶层模块 gcd_top。gcd_input 为数据输入模块，gcd_plc 为流水线计算模块，gcd_output 为数据输出模块。gcd_top 提供了设计要求中的输入和输出接口，例化了 1 个 gcd_input 模块、63 个 gcd_plc 模块、1 个 gcd_output 模块，构成总共 65 级流水线，并对它们进行了顶层的连接。顶层模块的结构框图如下图所示。

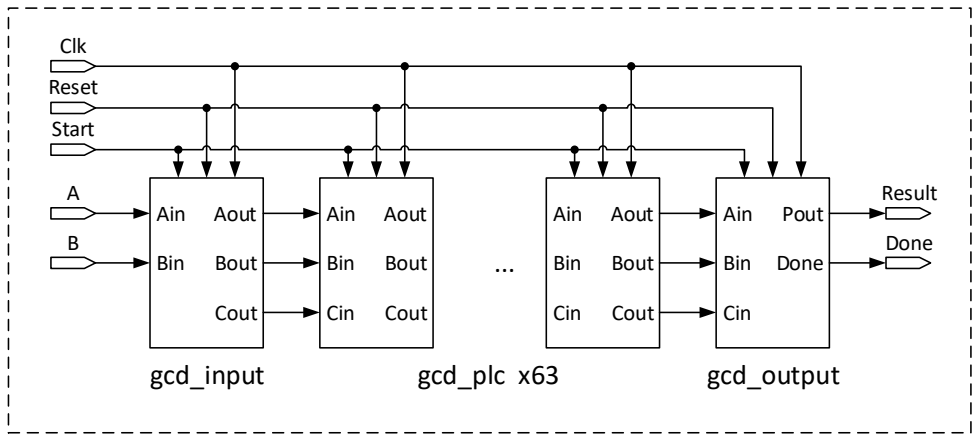


Figure 5: gcd_top 模块结构框图

需要指出，在本设计中，Reset 信号置低时，流水线中所有寄存器清零；Start 信号置低时，流水线中所有寄存器的值保持不变，即进入 halt 状态，当 Start 信号重新置高时，运算继续进行；Done 信号仅在有效计算结果输出时置高。下面将分别对三个子模块设计进行详细说明。

3.2.1 数据输入模块

该模块将输入的 A、B 传送到流水线计算模块中进行计算，同时为 C 赋初值。该模块的电路图如下图所示。

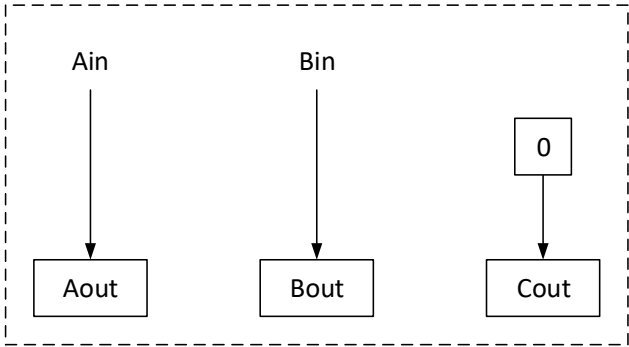


Figure 6: gcd_input 模块电路图

3.2.2 流水线计算模块

该模块为 gcd 处理器的核心模块，实现了 stein 算法中的一次迭代。本设计中采用的算法流程图如下图所示。

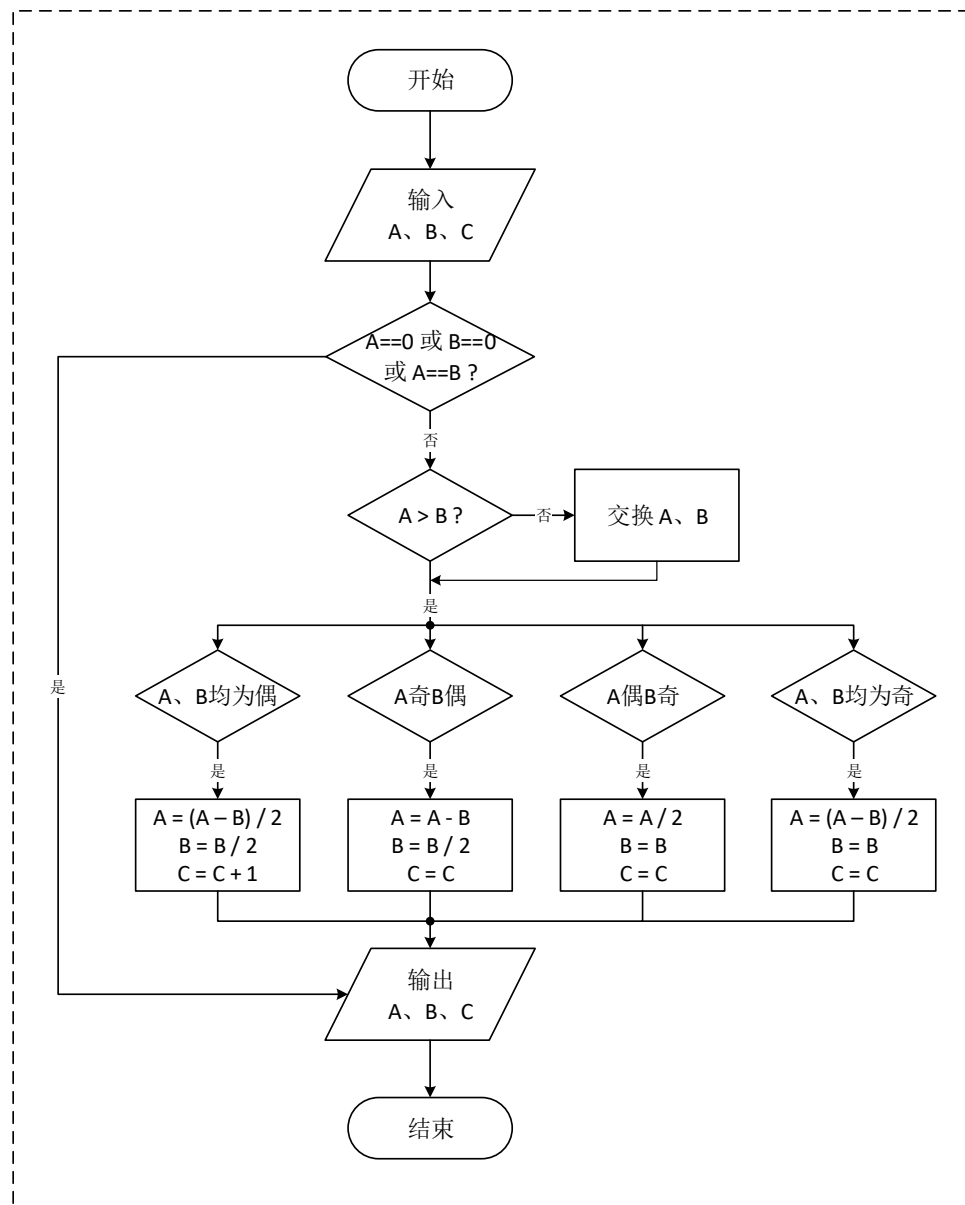


Figure 7: stein 算法流程图

原始的 stein 算法在一次迭代中需要进行比较、减法、移位判断（奇偶判断）、移位四步计算，那么关键路径即包含了 32 位比较、32 位减法、1 位比较、移位四部分延时。尽管去除了欧几里得算法中的取余操作，降低了硬件开销和计算延时，但仍然具有改进空间。这里的 32 位比较和 32 位减法实际上进行了重复工作，比较是为了保证减法的结果为正，但为了提高性能，我们可以直接将 $A-B$ 和 $B-A$ 的减法并行地计算出来，然后使用 2 路选择器选出为正的结果，得到 $|A-B|$ 。进一步的，我们发现因为 A、B 的输出结果只有 $|A-B|$ 、 $|A-B|/2$ 、A、A/2、B、B/2 这六种情况，我们可以提前将 $(A-B)/2$ 、 $(B-A)/2$ 、A/2、B/2 并行地计算出来，然后将奇偶判断和上述的正负判断结合为一个更大的 8 路选择器，得到最终得到 A、B 输出。改进后的电路图如下图所示。

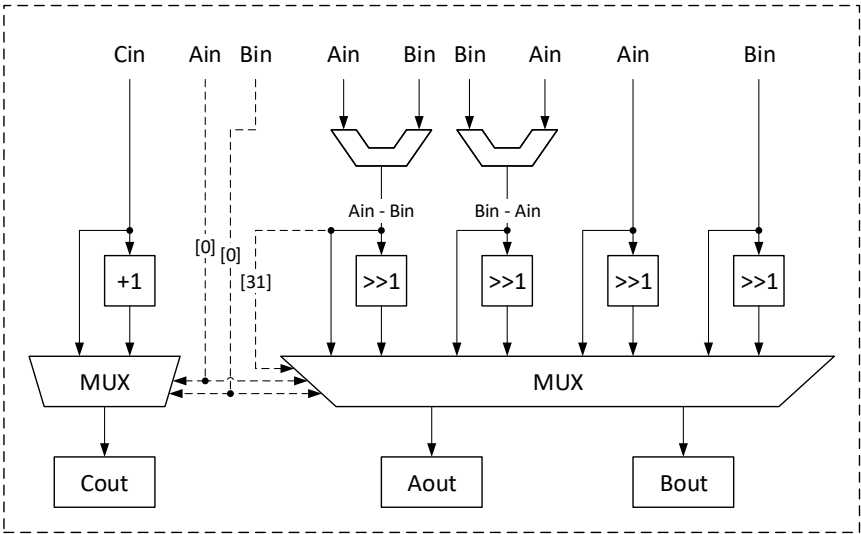


Figure 8: 改进后的 gcd_plc 模块电路图

通过以上的改进，关键路径将仅包含 32 位减法、移位、多路选择 3 部分延时，可以去掉延时较高的 32 位比较操作，大大降低关键路径长度，提高计算性能。

3.2.3 数据输出模块

该模块将 A、B 进行适当的移位并输出，同时给出 Done 信号。该模块的电路图如下图所示。

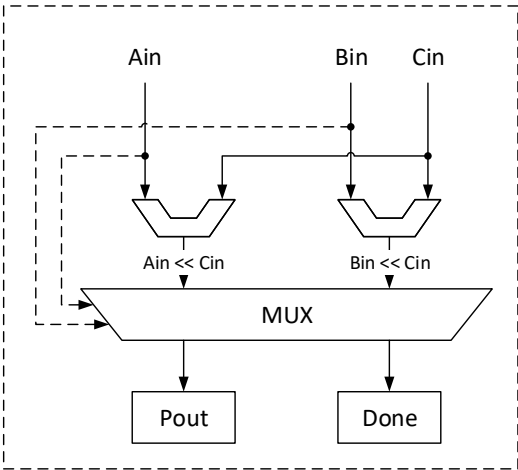


Figure 9: gcd_output 模块电路图

3.3 功能验证

在功能验证中，我们用到了 VCS 来运行 systemverilog 的 testbench，用 dve 来观察验证的波形图于 debug 调试。在功能验证中，我们生成了一万组随机数，并且对比设计时所用的 verilog 模块得到的结果与 c 语言模块得到的结果进行对比。其中验证的关键代码如下所示：

```

1  for(k = 0; k < 10000; k = k + 1) begin
2      #200;
3      if (bus.randomize() == 1);
4      else
5          $display("Randomization failed");
6      opa = bus.NA;
7      opb = bus.NB;
8      gcdresult = gcd(longint'(bus.NA), longint'(bus.NB));
9      #200;
10     start = 1'b1;
11     #####200;
12     start = 1'b1;
13     #10000000;
14     if (desingresult == gcdresult)
15         nright = nright + 1;
16     else begin
17         $display("operator is: opa = %d, opb = %d", opa, opb);
18         $display("two result is not equal: desingresult = %d, gcdresult = %d", desingresult, gcdresult);
19         errorFlag = 1'b1;
20         #####nfalse = nfalse + 1;
21     end
22     if (k % 100 == 0)
23         #####$display("have finish %d percent test", k/100);
24 end

```

最后验证结果为高性能设计中一万组测试数据全部通过。仿真的 systemverilog 脚本与仿真结果的 log 在 simulation 文件夹中。

3.4 Vivado 综合，功耗、资源占用与时序分析

我们使用 vivado 在 out of context 模式下对高性能设计进行了综合，在 xilinx xxx 系列芯片上，高性能设计的最高频率可以达到 700MHz。700MHz 下资源占用情况如下图所示。

Resource	Estimation	Available	Utilization %
LUT	17908	274080	6.53
FF	5649	548160	1.03

Figure 10: 资源利用

功耗报告如下图所示。

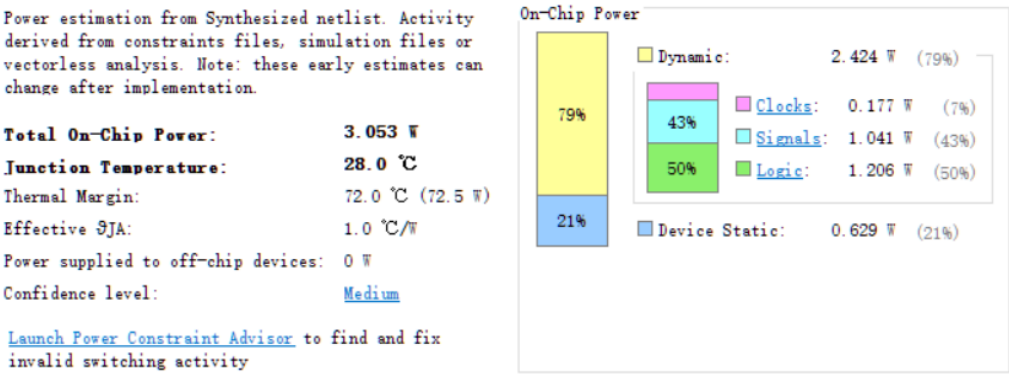


Figure 11: 功耗报告

时序报告如下图所示。

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 0.017 ns	Worst Hold Slack (WHS): 0.094 ns	Worst Pulse Width Slack (WPWS): 0.435 ns
Total Negative Slack (TNS): 0.000 ns	Total Hold Slack (THS): 0.000 ns	Total Pulse Width Negative Slack (TPWS): 0.000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 5618	Total Number of Endpoints: 5618	Total Number of Endpoints: 5649

All user specified timing constraints are met.

Figure 12: 时序报告

通过时序报告可以发现，关键路径存在于 gcd_plc 中，和我们在 RTL 电路设计流水线计算模块一节中的估计相吻合。所以，减法、移位和多路选择构成了电路的关键路径，限制了电路频率的提高。

在 100MHz、200MHz、300MHz、400MHz、500MHz、600MHz、700MHz 下功耗和资源占用情况（LUT）以及计算得到的能耗比，如下表所示。这里我们以每秒钟可以进行的 gcd 次数（即吞吐率）作为性能指标，以性能和功耗之比作为能耗比指标。

	功耗 W	LUT %	性能 gcd/s	能耗比 gcd/(sW)
100 MHZ	0.959	6.20	100	104.3
200 MHZ	1.304	6.20	200	153.4
300 MHZ	1.648	6.20	300	182.0
400 MHZ	1.993	6.21	400	200.7
500 MHZ	2.346	6.53	500	217.1
600 MHZ	2.692	6.53	600	222.9
700 MHZ	3.053	6.53	700	229.3

Table 3: 不同频率下的功耗、资源占用和能效比

不同频率下性能和能耗比的曲线如下图所示。

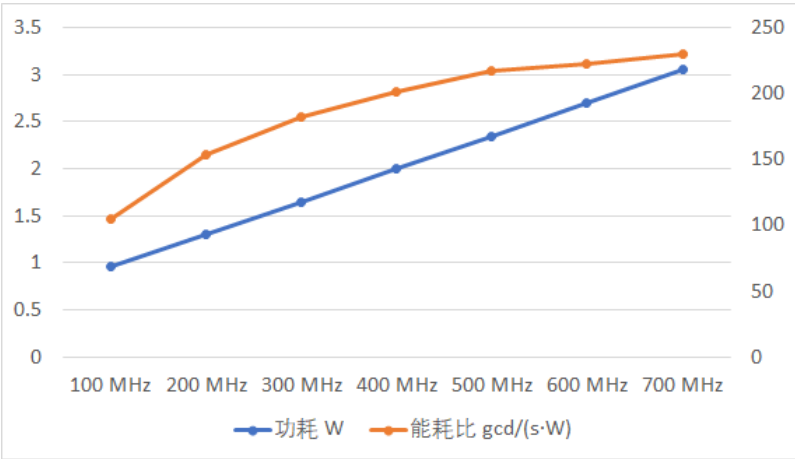


Figure 13: 不同频率下的功耗和能效比

3.5 基于 SoC 的验证平台

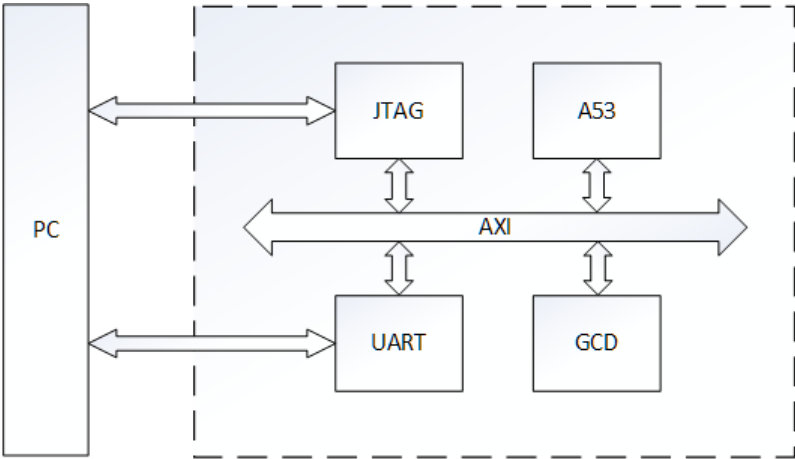


Figure 14: SoC 顶层框图

构建如上图所示的 SoC 系统，其中 gcd 模块的访问逻辑较简单，采用由 Xilinx 公司提供的 AXI GPIO 作为 AXI 总线与 gcd 模块的通信方式。最终实现的硬件框图如下所示。

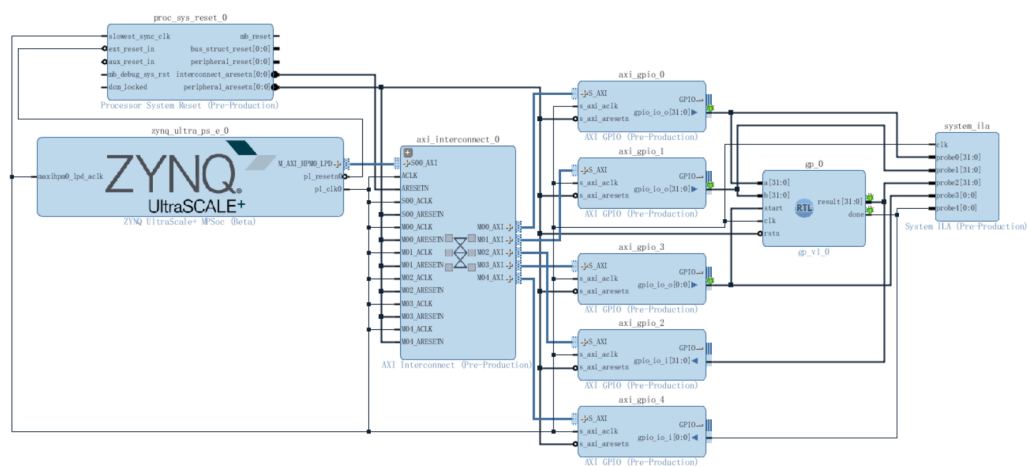


Figure 15: SoC 电路图

其中，System ILA 模块用于在调试 SoC 时，抓取板上信号。
本设计最终运行在时钟频率 280Mhz，最终 Soc 系统的资源占用如下所示。

Resource	Utilization	Available	Utilization %
LUT	19163	274080	6.99
LUTRAM	468	144000	0.33
FF	11259	548160	2.05
BRAM	6	912	0.66
BUFG	2	404	0.50

Figure 16: 资源利用

时序报告如下所示。

Setup	Hold
Worst Negative Slack (WNS): 0.076 ns	Worst Hold Slack (WHS): 0.030 ns
Total Negative Slack (TNS): 0.000 ns	Total Hold Slack (THS): 0.000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 29829	Total Number of Endpoints: 29829

Figure 17: 时序报告

功耗报告如下所示。

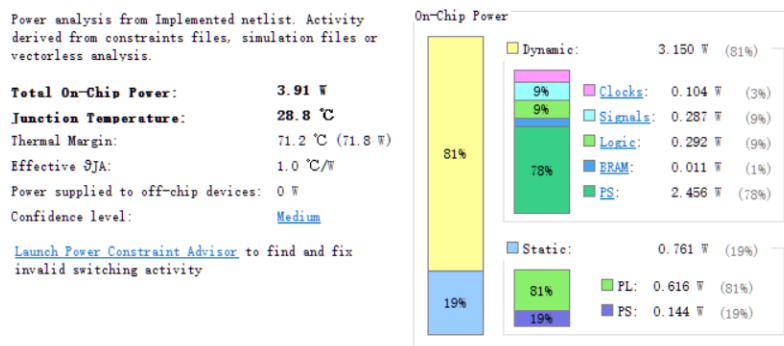


Figure 18: 功耗报告

将硬件码流烧进 FPGA，并将在 Xilinx 软件开发工具（XSDK）中编写完成的 C 程序在裸机下运行。C 程序如下所示。

3.5.1 SoC 验证代码

```

1 #include <stdio.h>
2 #include "platform.h"
3 #include "xil_printf.h"
4 #include <stdlib.h>
5 #define OPA (unsigned int *)0x80000000
6 #define OPB (unsigned int *)0x80010000
7 #define START (unsigned int *)0x80030000
8 #define DONE (unsigned int *)0x80040000
9 #define RESULT (unsigned int *)0x80020000
10 int gcd_0(int a, int b) {
11     if (a%b == 0)
12         return b;
13     return gcd_0(b, a%b);
14 }
15 int gcd(int a, int b) {
16     volatile unsigned int *p = OPA;
17     volatile unsigned int *q = OPB;
18     volatile unsigned int *s = START;
19     volatile unsigned int *t = DONE;
20     volatile unsigned int *m = RESULT;
21     *p = a;
22     *q = b;
23     *s = 1;
24     while (*t == 0) {;}
25     printf("gcd(%4d,%4d)=%4d,", a, b, *m);
26     *s = 0;
27     return *m;
28 }
29 int main()
30 {

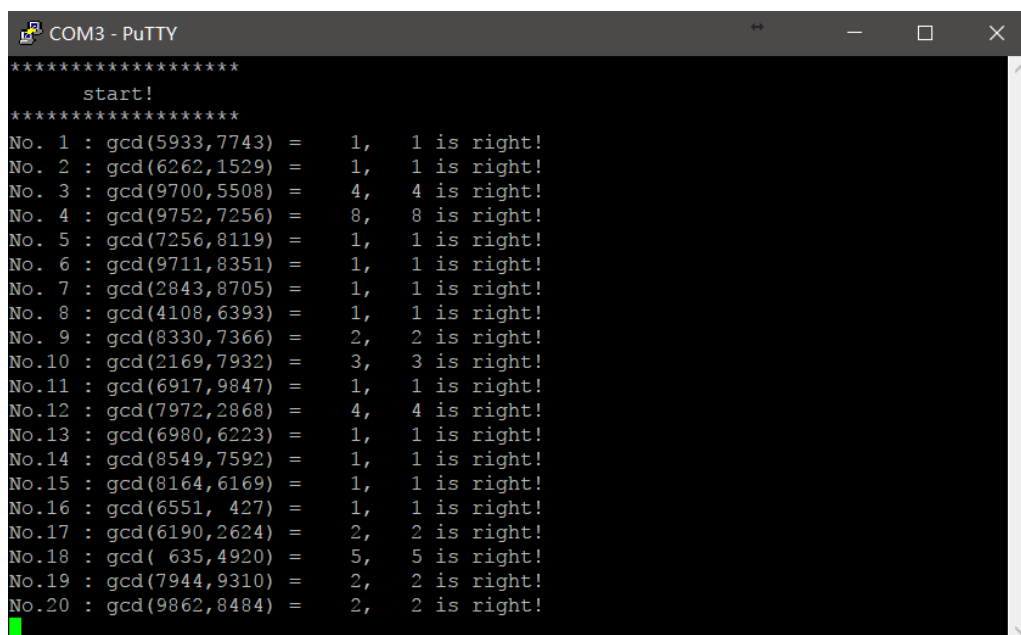
```

```

31  init_platform();
32  print("*****\n\r");
33  print("start!\n\r");
34  print("*****\n\r");
35  int i = 1;
36  int a,b;
37  while (i <= 20) {
38      a = rand() % 10000;
39      b = rand() % 10000;
40      printf("No.%2d:", i);
41      volatile int m = gcd(a, b);
42      volatile int k = gcd_0(a, b);
43      if (k == m)
44          printf("%4d is right!\n\r", k);
45      else
46          print("ERROR!\n\r");
47      i++;
48  }
49  cleanup_platform();
50  return 0;
51  }

```

最终通过 UART 观察到的结果如下。



```

COM3 - PuTTY
*****
start!
*****
No. 1 : gcd(5933,7743) = 1, 1 is right!
No. 2 : gcd(6262,1529) = 1, 1 is right!
No. 3 : gcd(9700,5508) = 4, 4 is right!
No. 4 : gcd(9752,7256) = 8, 8 is right!
No. 5 : gcd(7256,8119) = 1, 1 is right!
No. 6 : gcd(9711,8351) = 1, 1 is right!
No. 7 : gcd(2843,8705) = 1, 1 is right!
No. 8 : gcd(4108,6393) = 1, 1 is right!
No. 9 : gcd(8330,7366) = 2, 2 is right!
No.10 : gcd(2169,7932) = 3, 3 is right!
No.11 : gcd(6917,9847) = 1, 1 is right!
No.12 : gcd(7972,2868) = 4, 4 is right!
No.13 : gcd(6980,6223) = 1, 1 is right!
No.14 : gcd(8549,7592) = 1, 1 is right!
No.15 : gcd(8164,6169) = 1, 1 is right!
No.16 : gcd(6551, 427) = 1, 1 is right!
No.17 : gcd(6190,2624) = 2, 2 is right!
No.18 : gcd( 635,4920) = 5, 5 is right!
No.19 : gcd(7944,9310) = 2, 2 is right!
No.20 : gcd(9862,8484) = 2, 2 is right!

```

Figure 19: UART 结果

测试通过。

4 低功耗设计

4.1 设计综述

经过第 2 章中对算法的分析，在低功耗的设计中，我们采用了算法1，也就是改进的 stein 算法。该算法的优势也正如前面分析的一样：

- 硬件实现简单，只包含**选择判断有效位数，比较，移位，减法操作**，不含复杂的除法操作。
- 平均迭代次数为 23.7273 接近于 gcd 算法的平均 18.2592 次迭代

所以根据改进的 stein 算法的优势，我们预期其在低功耗设计中具有很优秀的表现。由于是低功耗设计，我们采用状态机而不是流水线的形式来控制算法流程，这样能够大大减少寄存器的数量和组合逻辑电路的数量，降低电路的静态功耗。而且由于迭代次数短，电路的动态功耗也能被有效降低。

4.2 状态机设计

由于采用状态机的设计，根据算法1，我们设计了一个状态机，首先定义了状态机的几个状态为：

IDLE 表示为空闲状态；

INPUT 表示为输入采样状态；

ADJUDICATE 表示判别状态，表示继续输入或者结束运算输出结果

CAMPSWITCH 表示比较大小，并且交换操作数的位置

ALIGN 表示使操作数对齐的状态

ABSOLUTE 表示两操作数相减取绝对值得状态

OUTPUT 表示输出状态

并且我们定义了状态机的转换框图20为：

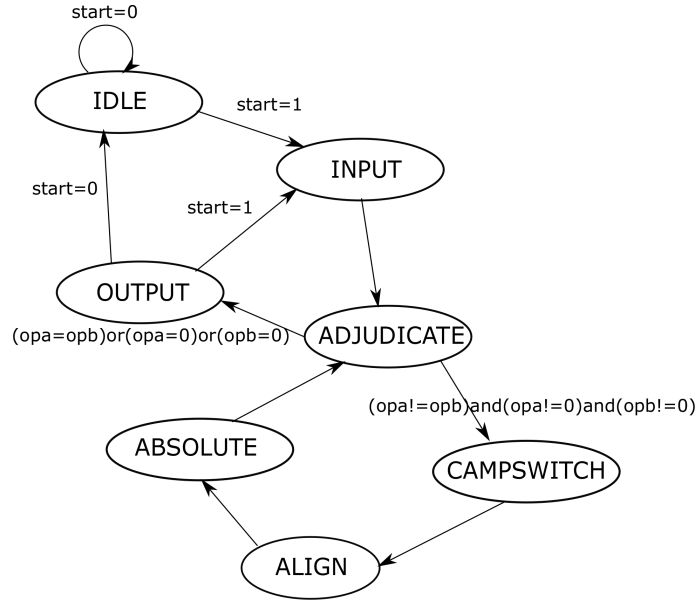


Figure 20: 状态机转换图

在改状态机中, 首先系统处于 IDLE 状态, 当 start 信号到来之后, 系统开始采样输入, 进入 INPUT 状态。接着从 INPUT 状态进入 ADJUDICATE 状态, 对两个操作数进行判别, 看操作数是否达到可以输出的条件。如果达不到输出的条件, 系统计入计算周期 $CAMPSWITCH \rightarrow ALIGN \rightarrow ABSOLUTE \rightarrow ADJUDICATE$ 。计算周期结束之后继续进行判别。如果达到输入条件, 系统进入 OUTPUT 状态并且输出结果。

4.3 两种低功耗设计

根据状态机来设计低功耗电路使, 我们经过设计迭代, 发现有两种低功耗的设计方案, 一种是寄存器较多的设计, 一种是组合电路较多的设计。经过仿真验证与综合分析我们发现两者的功耗相当, 寄存器较多的设计比组合电路较多的设计的功耗略大, 到时能够达到的最高频率高, 关键路径短。经过我们小组讨论分析, 绝对这两种方案都适合于做低功耗设计, 因此下面来具体说明两种设计方案。为了表示, 我们将寄存器较多的设计成为 **register case**, 组合电路较多的设计成为 **logic case**。

4.4 resgister case

首先是 resgiste case——也就是寄存器较多的设计, register case 设计在文件 *register_case* 中, 在这个设计中, 我们定义了一系列的寄存器:

```

current_state[2:0], next_state[2:0], regopa[31:0], regopb[31:0], regpa_out[31:0],
regpb_out[31:0], tmp_regopa[31:0], tmp_regopb[31:0], result[31:0], done, camp_flag

```

在 resgiste case 中一共用掉了 232 个寄存器。其中 $current_state[2:0], next_state[2:0]$ 主要是用于控制状态机的寄存器, $regopa[31:0], regopb[31:0], regpa_out[31:0]$

$0], regpb_out[31:0], tmp_regopa[31:0], tmp_regopb[31:0]$ 是用来暂存操作数的 3 对寄存器, $result[31:0]$ 是输出结果的寄存器。 $done, camp_flag$ 为运算的标志位。下面来对各个模块进行具体的分析。

4.4.1 比较与交换数据模块

该模块的主要功能是对暂存在 $regopa[31:0]$ 和 $regopb[31:0]$ 上的操作数进行比较大小并且将两者之中较大的数写入寄存器 $tmp_regopa[31:0]$ 中, 将两者中较小的数写入寄存器 $tmp_regopb[31:0]$ 中, 并且根据比较大小的结果写标志位 $camp_flag$ 。系统接着处理寄存器 $tmp_regopa[31:0]$ 与寄存器 $tmp_regopb[31:0]$ 中的数据。该模块的组合逻辑的硬件开销为 2 个 32-bit 二路选择器和一个 32-bit 的比较器。

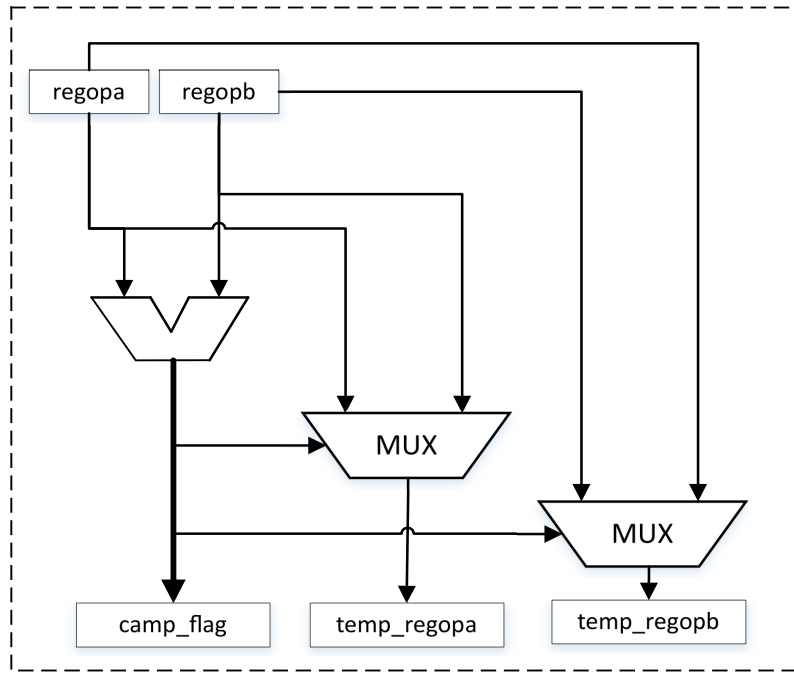


Figure 21: *CAMP_SWITCH* 模块

4.4.2 对齐模块

当系统从 *CAMP_SWITCH* 状态进入 *ALIGN* 状态之后, 需要根据 $tmp_regopa[31:0]$ 与 $tmp_regopb[31:0]$ 的数据位宽来进行数据对齐操作。其中在求操作数的有效数据位宽为改设计中最消耗资源的模块。在设计求操作数有效数据位宽的操作中按位判断操作数的有效数据位宽的硬件开销太大而且关键路径很长, 因此在设计求操作数有效数据位宽的模块时, 我们采用了 5 层判断, 类似于二分法的一个设计, 首先判断操作数的高 16 位是否为 0, 进而再判断 8 位、4 位、2 位、1 位, 是一种由粗到细判断。这样实现的好处是硬件结构较为对称, 电路关键路径短。

例如在判断 00000000000000001000000000000000 的有效位位数时, 我们先判断高 $[31:16]$ 为 0 \rightarrow 判断 $[15:8]$ 不为 0 \rightarrow 判断 $[15:12]$ 不为 0 \rightarrow 判断 $[15:14]$ 不为 0 \rightarrow 判断 $[15]$ 不为 0, 继而得到结果, 有效位数为 16 位。

求解有效位数的硬件结构框图如下所示：

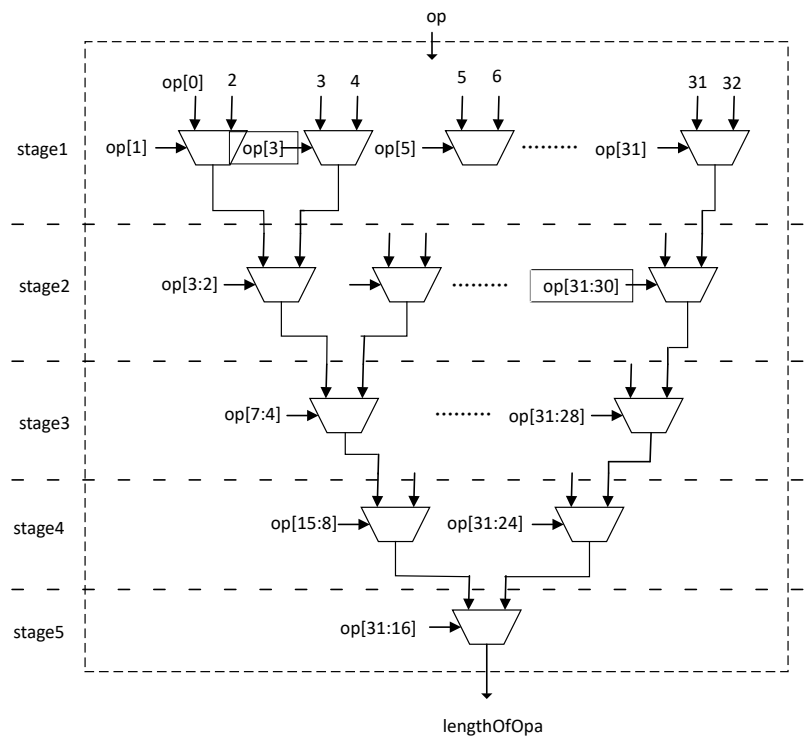


Figure 22: 有效位数模块

所以有效位数模块中的最长路径为 5 个二路选择器的延时。

在得到两个操作数的有效位数之后，将两者的有效位数相减便得到有效位数之差，进而可以对较小的数进行移位操作。所以整个对齐模块的框图如下图24所示：

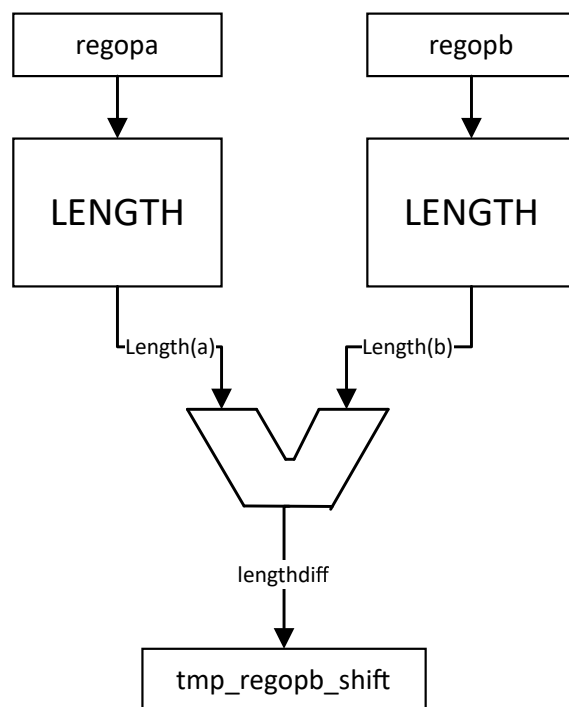


Figure 23: 对齐模块

可以看见对齐模块的关键路径长度较长，在关键路径的延时上包括了一个 LENGTH 模块的延时，一个减法模块的延时，一个移位模块的延时。

4.4.3 减法求绝对值模块

在对操作数进行移位之后，进入 ABSOLUTE 状态，在 ABSOLUTE 状态中，首先对两个操作数进行大小比较，然后用大数减去小数得到两个操作数的差，用两个操作数的差相减得到结果的绝对值。并且相减得到的绝对值赋值给 regopa 和 regopb 中两个中较小数赋值给 regopb（这里需要注意的是，并不是将 tmp_regopa 和 tmp_regopb 的较小的数，因为 tmp_regopb 已经经过移位，将 tmp_regopb 赋值给 regopb 会出错），之后再次进入 ADJUDICATE，这样一次迭代周期就完成了。

减法求绝对值模块的模块的框图如下所示：

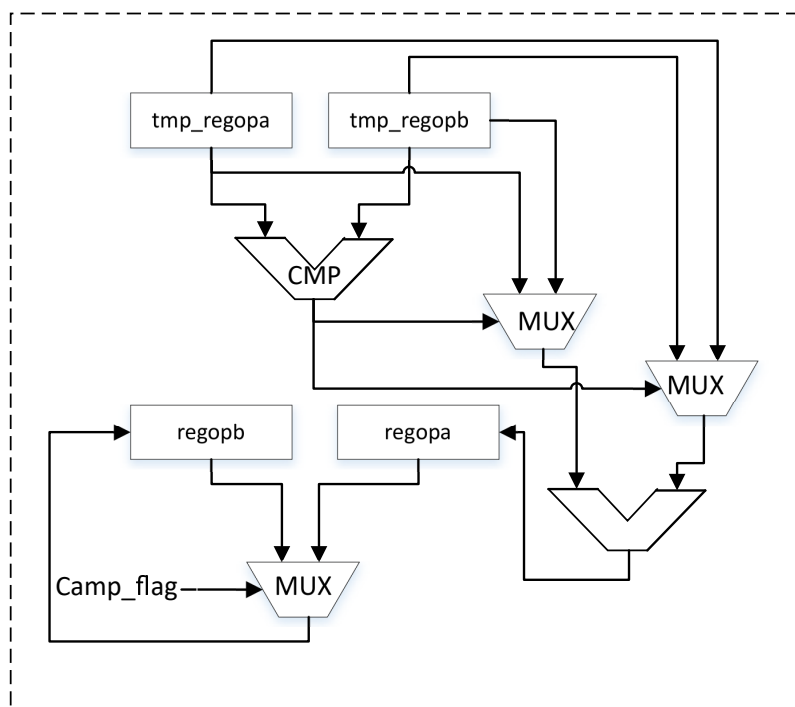


Figure 24: 减法求绝对值模块

4.5 logic case 设计

在组合逻辑较多的设计中，我们对寄存器的个数进行了简化，只保留了 `current_state`, `next_state`, `regopa`, `regopb`, `result` 几个寄存器，将对中间寄存器的操作都合并为对 `regopa` 和 `regopb` 的操作，并且引入 `lengt_diff_reg` 寄存器来暂存两个操作数的有效位数差。这个两个 case 设计的时候做出的一点改变。因为在 logic case 的设计中，只有 `regopa`, `regopb` 两组寄存中，在 `ALIGN` 状态之，原始的有效位数信息丢失，所以为了保存原始的有效位数的信息，加入了 `lengt_diff_reg` 寄存器来保存历史的有效位数信息。

在 logic case 的中只用到了 102 个寄存器，比原来的 `resgister case` 设计中少了一半以上的寄存器，但是相应的组合逻辑的控制会变复杂，而且由于 `regopa` 和 `regopb` 在每个状态时刻都会发生改变，在变化时与之相连接的组合逻辑的电路也会发生电路的反转，因而在组合逻辑上会造成更大的功耗，而且由于组合逻辑相应地变得复杂，也会造成关键路径的延时的增加。所以 logic case 的设计于 `resgister case` 的设计之间各有利弊。

4.6 功能验证

在功能验证中，我们用到了 VCS 来运行 `systemverilog` 的 `testbench`，用 `dve` 来观察验证的波形图于 `debug` 调试。在功能验证中，我们生成了一万组随机数，并且对比设计时所用的 `verilog` 模块得到的结果与 `c` 语言模块得到的结果进行对比。其中验证的关键代码如下所示：

```
1 for(k = 0; k < 10000; k = k + 1) begin
2   #200;
```

```

3   if (bus.randomize() == 1);
4   else
5       $display("Randomization failed");
6   opa = bus.NA;
7   opb = bus.NB;
8   gcdresult = gcd(longint'(bus.NA), longint'(bus.NB));
9   #200;
10  start = 1'b1;
11  #200;
12  start=1'b0;
13  #1000000;
14  if (desingresult == gcdresult);
15  else begin
16      $display("operator is: opa=%d, opb=%d",
17      opa, opb);
18      $display("two result is not equal:
19  desingresult=%d, gcdresult
20  =%d", desingresult, gcdresult);
21      errorFlag = 1'b1;
22  end
23  if(k%100==0)
24      $display("have finish %d percent test", k/100);
25  end

```

最后验证结果为 logic case 设计与 register case 中一万组测试数据全部通过。仿真的 systemverilog 脚本与仿真结果的 log 在 simulation 文件夹中。

4.7 DC 综合，功耗，面积与时序分析

在 DC 综合中我们对于 logic case 与 register case 设计的不同频率进行综合，发现 logic case 的最高频率能够达到 500MHZ，register case 设计中的最高频率能够达到 600MHZ。因此对 logic case 的设计，综合其 100MHZ，200MHZ，300MHZ，400MHZ，500MHZ 下的不同功耗。对 register case 的设计，综合其 100MHZ，200MHZ，300MHZ，400MHZ，500MHZ，600MHZ 下的不同功耗。一下是综合得到的结果：

	logic case	register case
100MHZ	0.1415 mW	0.2359 mW
200MHZ	0.2840 mW	0.4773 mW
300MHZ	0.5257 mW	0.7263 mW
400MHZ	0.6992 mW	0.9869 mW
500MHZ	1.0016 mW	1.3718 mW
600MHZ		1.8776 mW

Table 4: 不同频率下的功耗 *mw*

	logic case	register case
100MHZ	4181.039986	4782.959931
200MHZ	4176.360006	4913.999997
300MHZ	4807.080038	4959.360001
400MHZ	5236.560042	5299.920007
500MHZ	6531.840052	5937.839956
600MHZ		7064.639998

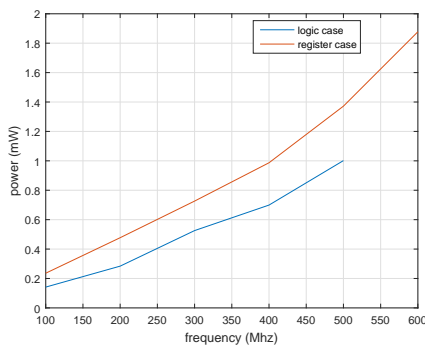
Table 5: 不同频率下的面积 $um * um$ 

Figure 25: 不同频率下的功耗

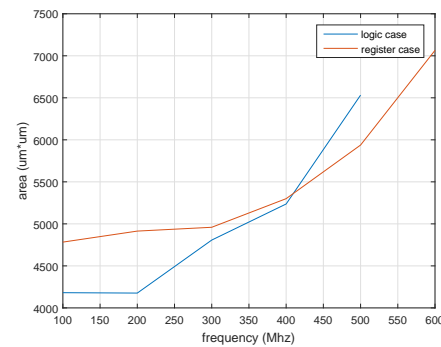


Figure 26: 不同频率下的面积

通过观察表4, 表6, 图25,26, 可以发现, logic case 因为其寄存器少的原因, 功耗比 logic case 大, 在时序报告中也可以反应出。对于 logic case 的设计, 寄存器功耗占了约 55%, 组合逻辑的功耗占了月 45%。而对于 register case 的设计, 寄存器的功耗占了约 84%, 组合逻辑的功耗占了约 16%。所以在 register case 的设计中, 寄存器值主要的电路功耗的开销。

在面积报告上, 我们也发现一个特点, logic case 的设计面积随着频率增长的速度比较快, 而 register case 随频率的增长速度比较慢。造成该现象的原因为 register case 将组合逻辑切分成比较多的块, 每个块在综合的时候相比于 logic case 设计中复杂的组合逻辑, 综合的不确定性要低。另外一个原因是我们设计中 RTL 代码使用了比较高层次的行为级语言, 而不是更加精确的门级语言, 所以综合器有更大的综合空间来优化时序。

对于 logic case 的设计, 频率的瓶颈在 500MHZ, 对于 register case 的设计, 电路时序的瓶颈为 600MHZ。通过时序报告得到该设计的关键路径为对其模块中的 tmp_regopb 的路径, 关键路径为下图所示:

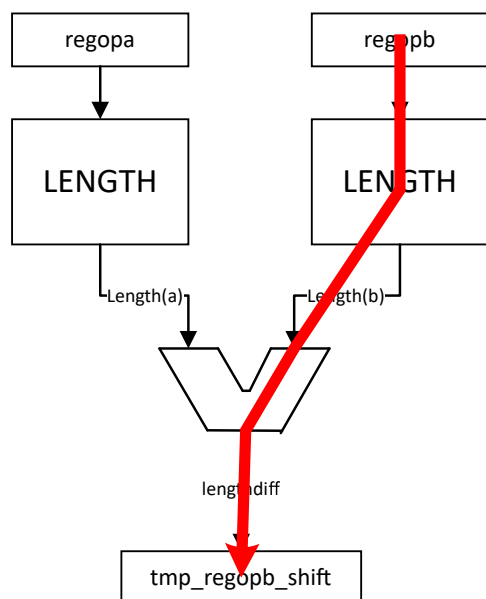


Figure 27: logic case 设计关键路径

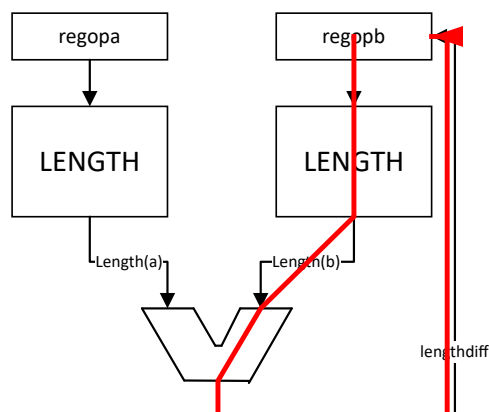


Figure 28: register case 设计关键路径

所以，求操作数的有效位宽，到求解位宽之差，到操作数的移位，这些操作构成了电路的关键路径，限制了电路频率的进一步提高。

4.8 能效比分析

电路的能效比为电路处理单位 bit 数所需要的能量。首先，通过我们仿真得到算法得到结果的平均迭代次数为 23.6 次，加上输入和输出态各需要占用一个时钟周期，所以总共需要占用约为 96.4 个周期：

$$1 + 1 + 23.6 \times 4 = 96.4$$

所以能效比计算公式为：

$$efficient = \frac{power \times circle}{frequency \times nBit}$$

经过计算电路的能效比为：

	logic case	register case
100MHZ	4.2626	7.1064
200MHZ	4.2777	7.1893
300MHZ	5.2789	7.2932
400MHZ	5.2658	7.4325
500MHZ	6.0346	8.2651
600MHZ		9.4271
average	5.0239	7.7856

Table 6: 不同频率下能效比 fj/bit

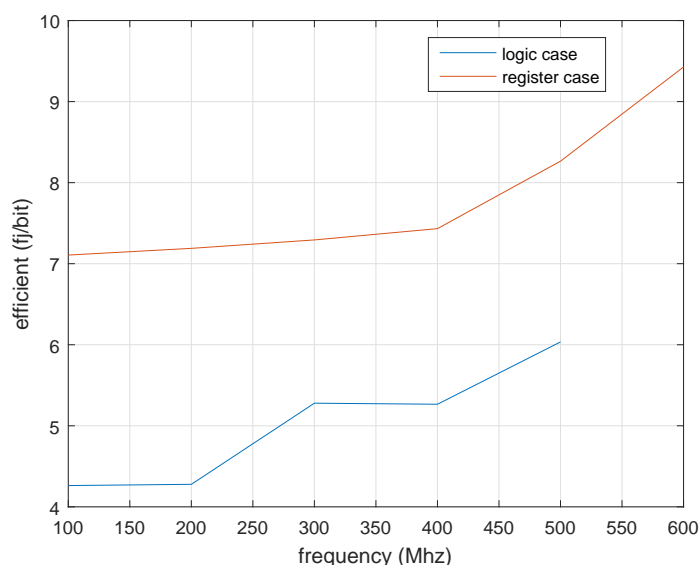


Figure 29: 不同频率下的能效比

5 总结

在这次的课程项目中我们小组首先对求最大公约数的算法进行分析，发现原始的 gcd 算法进行硬件实现存在着很大的开销，不仅电路关键路径长，而且不便于流水线。所以我们小组在对算法进行调研和仿真之后确定了 stein 算法作为高性能的实现的算法，stein 算法硬件实现简单，而且面积开销小，适合于做高性能的流水线实现。在对 stein 算法做改进之后作为低功耗的实现，改进的 stein 算法迭代周期短，硬件面积开销小，适合做低功耗的实现。

在高性能的硬件电路实现当中，我们采用了 65 级流水线，对其关键的 gcd_plc 模块进行优化，为了减小电路关键路径，提高电路频率，在 gcd_plc 中我们将所有可能的结果并行地计算出来然后通过选择器来选择输出。在 FPGA 开发板上实现了 700MHZ 的频率，并且通过了 systemverilog 仿真随机 10000 组数据的验证，并且搭建 SOC 平台进行验证。验证结果表明了电路设计的正确性。

在低功耗的设计中，我们采用了改进的 stein 算法，设计状态机，并且设计了 logic_case 和 register_case，一种组合逻辑比较复杂，另一种寄存器比较多，通过比较发现，logic_case 的功耗比较小，register_case 的频率比较高，而且当综合约束频率上升以后，register_case 所占面积比较小。

6 附录