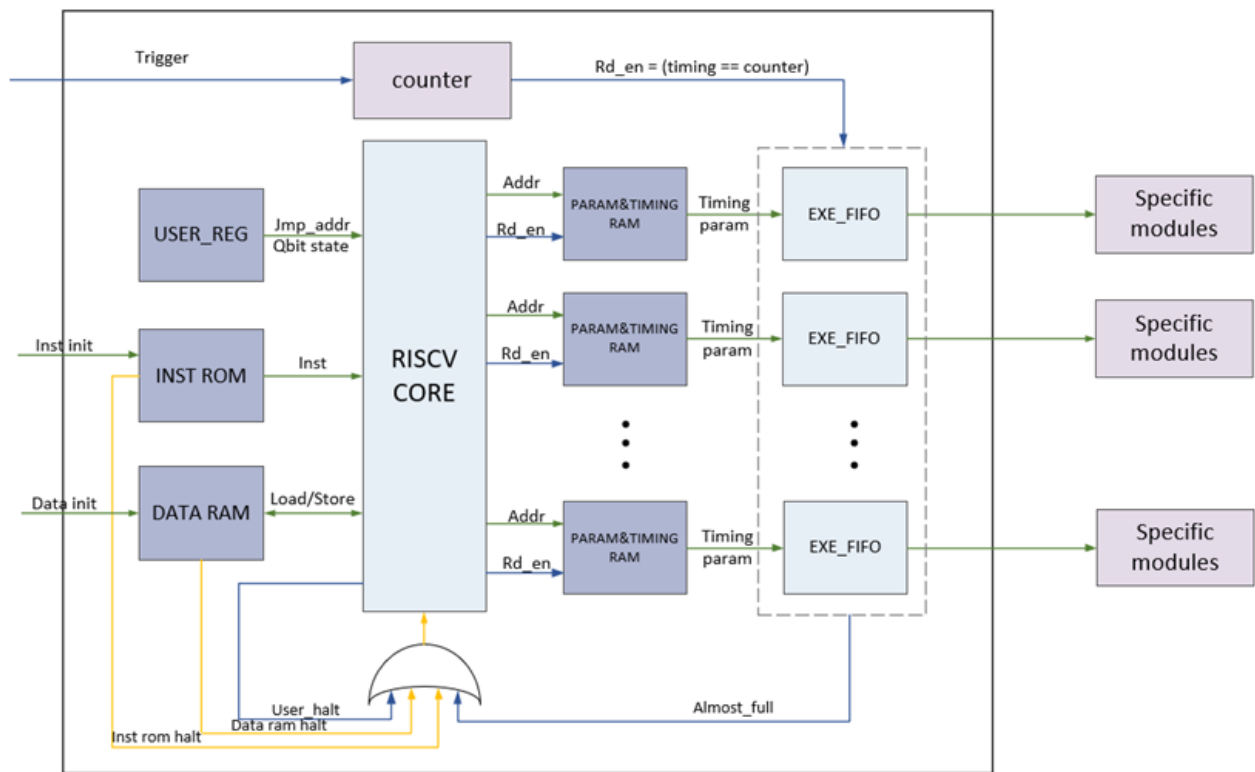


# 离子阱量子计算电控系统固件介绍

## 离子阱量子计算电控系统固件介绍

### 总体架构



可以看到图中需要进行初始化的内存有三个部分（实际上是4个部分，因为参数RAM和TIMING RAM写在了一起，这两者是绑定的），所以实际上我们需要初始化的是指令ROM，指令ROM中存储的内容是RISCQ的指令内容，其中循环的实现方式是正常的RISC实现的方式，对于外部的控制实际上就是通过用户寄存器写指令发出地址，该地址直接指向对应的参数RAM和其绑定的TIMING RAM，通过这种方式即可，RAM部分一旦接收到读出的指令就会将对应地址的参数读出并且传输到FIFO中，FIFO中的内容会和全局的计数器中的内容进行对比，如果相等，就会将FIFO中的内容传输到对应的模块对模块现有的参数进行更新，一个循环控制的代码示例如下：

#### 代码块

```
1  addi x1 x0 0
2  addi x2 x0 3
3  setur y0 x1 0
4  setur y1 x1 0
5  setur y2 x1 0
6  setur y3 x1 0
7  setur y4 x1 0
```

```

8  setur y5 x1 0
9  setur y6 x1 0
10 setur y7 x1 0
11 setur y8 x1 0
12 setur y9 x1 0
13 setur y10 x1 0
14 setur y11 x1 0
15 setur y12 x1 0
16 setur y13 x1 0
17 setur y14 x1 0
18 setur y15 x1 0
19 setur y16 x1 0
20 setur y17 x1 0
21 setur y18 x1 0
22 setur y19 x1 0
23 setur y20 x1 0
24 setur y21 x1 0
25 setur y22 x1 0
26 setur y23 x1 0
27 addi x1 x1 1
28 bne x1 x2 -100
29 jal x0 0
30 jal x0 0

```

可以看到图中是拿x1寄存器作为循环变量，x2寄存器作为循环比较值，每进行一次循环，就通过bne语句比较x1和x2的值，如果没有相等，那么就跳转回第三行重新开始执行，当所有执行完毕之后，就进入到jal x0 0语句，就是原地死循环。

## setur语句说明

其中yn表示用户寄存器，每个用户寄存器对应一个参数，所以通过用户寄存器可以直接寻址参数和timing RAM部分，后面跟的xn是通用寄存器，再后面的数字是立即数，即将通用寄存器的值和立即数相加存入到用户寄存器中，而通用寄存器和立即数加和的值就是对于对应RAM寻址的地址，例如：在第一遍循环中，x1作为循环变量，是0，那么对对应RAM的寻址就是0，第二遍循环，x1是1，那么寻址就是RAM中的地址1，以此类推，此处的RAM实际上就可以理解为一个可以重复调用的序列数组。

## 功能模块的参数讲解

### TTL模块

	param[127:96]	param[95]	param[94:88]
--	---------------	-----------	--------------

输出模式	持续时间	0: 低电平	输出延时[6:0]
	0为一直保持状态	1: 高电平	
输入（计数模式）	持续时间	0: 不计数	存入的用户寄存器地址 [2:0]
	0为一直保持状态	1: 计数	

note:此处的用户寄存器是单独的，和之前控制的寄存器分开，控制寄存器是RISCQ写入，而此处的寄存器是外部写入（如上述的计数器模块输入），之后由RISCQ读入。

定义三个函数append是数组元素添加，adjust\_array\_length是对数组补全，方便通信，get\_arrays分别返回两个数组用于生成对应的数据文件。

## DDS模块

代码块

```

1  class DDSArrayHandler:
2      def __init__(self):
3          # 初始化两个数组
4          self.array_128bit = [] # 用于存储 128 位数字
5          self.array_32bit = [] # 用于存储 32 位数字
6
7      def append(self, part1, part2, part3, extra32):
8          #part1是频率，单位是m，part2是相位，认为是2π的多少倍，一般是小于1的小数，
9          #part3是幅度，也是小于1的小数
10         if not all(0 <= part < 2**32 for part in [part1, part2, part3,
11         extra32]):
12             raise ValueError("All inputs must be 32-bit integers (0 <=
13             value < 2^32).")
14
15         # 将 part1 与 0x8000 相乘后使用
16         part3 = int(part3 * 0x8000)
17         part2 = int(part2 * 16777216)
18         part1 = int(part1 * 67108.864)
19
20         if not 0 <= part1 < 2**32:
21             raise ValueError("The result of part1 * 0x8000 must fit within
22             32 bits (0 <= value < 2^32).")
23
24         full_128bit = (0 << 96) | (int(part1) << 64) | (int(part2) << 32) |
25         int(part3)
26
27         self.array_128bit.append(full_128bit)
28         self.array_32bit.append(extra32)
29
30     def get_arrays(self):

```

```

27         """
28         返回存储的两个数组。
29         """
30         return self.array_128bit, self.array_32bit
31
32     def adjust_array_length(self):
33         while (len(self.array_128bit) % 4) != 3:
34             self.array_128bit.append(0)
35             self.array_32bit.append(0)
36
37         self.array_128bit.append(0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF)
38         self.array_32bit.append(0xFFFFFFFF)

```

基本的情况与TTL相同，但是DDS模块的参数长度是128bit，数据格式如下：

	[127:96]	[89:64]	[55:32]	[15:0]
参数分组	时间	频率	相位	幅度

只要通道在本序列中需要使用，那么即使一开始不使用，也需要给定相应的运行时间

## 参数化AWG模块

代码块

```

1  class AWG_AMP_array_handler:
2      def __init__(self):
3          # 初始化两个数组
4          self.array_128bit = [] # 用于存储 128 位数字
5          self.array_32bit = [] # 用于存储 32 位数字
6
7      def append(self, u0, u1, u2, u3, extra32):
8          if not all(0 <= part < 2**32 for part in [u0, u1, u2, u3, extra32]):
9              raise ValueError("All inputs must be 32-bit integers (0 <=
value < 2^32).")
10         u0 = int(u0 * 0x8000)
11         u1 = int(u1 * 0x8000)
12         u2 = int(u2 * 0x8000)
13         u3 = int(u3 * 0x8000)
14
15         v0 = u0
16         v1 = u1 + u2/2 + u3/6
17         v2 = u2 + u3
18         v3 = u3
19

```

```

20         full_para_128bit = (int(v3)<<96) | (int(v2)<<64) | (int(v1)<<32 ) |
(int(v0))
21
22         self.array_128bit.append(full_para_128bit)
23         self.array_32bit.append(extra32)
24
25     def get_arrays(self):
26         """
27         返回存储的两个数组。
28         """
29         return self.array_128bit, self.array_32bit
30
31     def adjust_array_length(self):
32         while (len(self.array_128bit) % 4) != 3:
33             self.array_128bit.append(0)
34             self.array_32bit.append(0)
35
36         self.array_128bit.append(0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF)
37         self.array_32bit.append(0xFFFFFFFF)
38
39 class AWG_PHASE_array_handler:
40     def __init__(self):
41         # 初始化两个数组
42         self.array_128bit = [] # 用于存储 128 位数字
43         self.array_32bit = [] # 用于存储 32 位数字
44
45     def append(self, u0, u1, u2, u3, extra32):
46         if not all(0 <= part < 2**32 for part in [u0, u1, u2, u3,extra32]):
47             raise ValueError("All inputs must be 32-bit integers (0 <=
value < 2^32).")
48         u0 = int(u0 * 16777216)
49         u1 = int(u1 * 16777216)
50         u2 = int(u2 * 16777216)
51         u3 = int(u3 * 16777216)
52
53         v0 = u0
54         v1 = u1 +u2/2+u3/6
55         v2 = u2 +u3
56         v3 = u3
57
58         full_para_128bit = (int(v3)<<96) | (int(v2)<<64) | (int(v1)<<32 ) |
(int(v0))
59
60         self.array_128bit.append(full_para_128bit)
61         self.array_32bit.append(extra32)
62
63     def get_arrays(self):

```

```

64         """
65         返回存储的两个数组。
66         """
67         return self.array_128bit, self.array_32bit
68
69     def adjust_array_length(self):
70         while (len(self.array_128bit) % 4) != 3:
71             self.array_128bit.append(0)
72             self.array_32bit.append(0)
73
74         self.array_128bit.append(0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF)
75         self.array_32bit.append(0xFFFFFFFF)
76
77     class AWG_FREQ_array_handler:
78         def __init__(self):
79             # 初始化两个数组
80             self.array_128bit = [] # 用于存储 128 位数字
81             self.array_32bit = [] # 用于存储 32 位数字
82
83         def append(self, u0, u1, u2, u3, extra32):
84             if not all(0 <= part < 2**32 for part in [u0, u1, u2, u3, extra32]):
85                 raise ValueError("All inputs must be 32-bit integers (0 <=
value < 2^32).")
86
87             u0 = int(u0 * 67108.864)
88             u1 = int(u1 * 67108.864)
89             u2 = int(u2 * 67108.864)
90             u3 = int(u3 * 67108.864)
91
92             v0 = u0
93             v1 = u1 + u2/2 + u3/6
94             v2 = u2 + u3
95             v3 = u3
96
97             full_para_128bit = (int(v3)<<96) | (int(v2)<<64) | (int(v1)<<32) |
(int(v0))
98
99             self.array_128bit.append(full_para_128bit)
100             self.array_32bit.append(extra32)
101
102     def get_arrays(self):
103         """
104         返回存储的两个数组。
105         """
106         return self.array_128bit, self.array_32bit
107
108     def adjust_array_length(self):
109         while (len(self.array_128bit) % 4) != 3:

```

```

109         self.array_128bit.append(0)
110         self.array_32bit.append(0)
111
112         self.array_128bit.append(0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF)
113         self.array_32bit.append(0xFFFFFFFF)

```

参数化AWG实际上就是将DDS的各个控制字转化为一个与时间有关的函数，而该函数又可以通过三次样条插值的方式进行拟合，从而产生任意的频率，相位，幅度包络。

上述三个函数类似，都是为了从给出的三次函数：

$$u(t) = u_0 + u_1 t + \frac{u_2}{2} t^2 + \frac{u_3}{6} t^3$$

计算出初始迭代计算值，并且加载到FPGA的迭代计算模块中去：

$$v_0 = u_0$$

$$v_1 = u_1 + \frac{u_2}{2} + \frac{u_3}{6}$$

$$v_2 = u_2 + u_3$$

$$v_3 = u_3$$

这些系数的单位暂定和之前的DDS单位相同