

Z芯片手册（Z-Chip Reference Manual）

保密情况：内部人员使用，禁止外传

撰写人：杨申波

版本：Z_Chip_ManualV1.2

时间：2026年7月19日

更新日志：

2026年8月9日：增加码字中波形保持位和NCO复位位 相关的信息。一、1.5小节增加了同步，反馈相关说明,同时增加了中断相关说明。

一、Z芯片介绍

1.1芯片功能

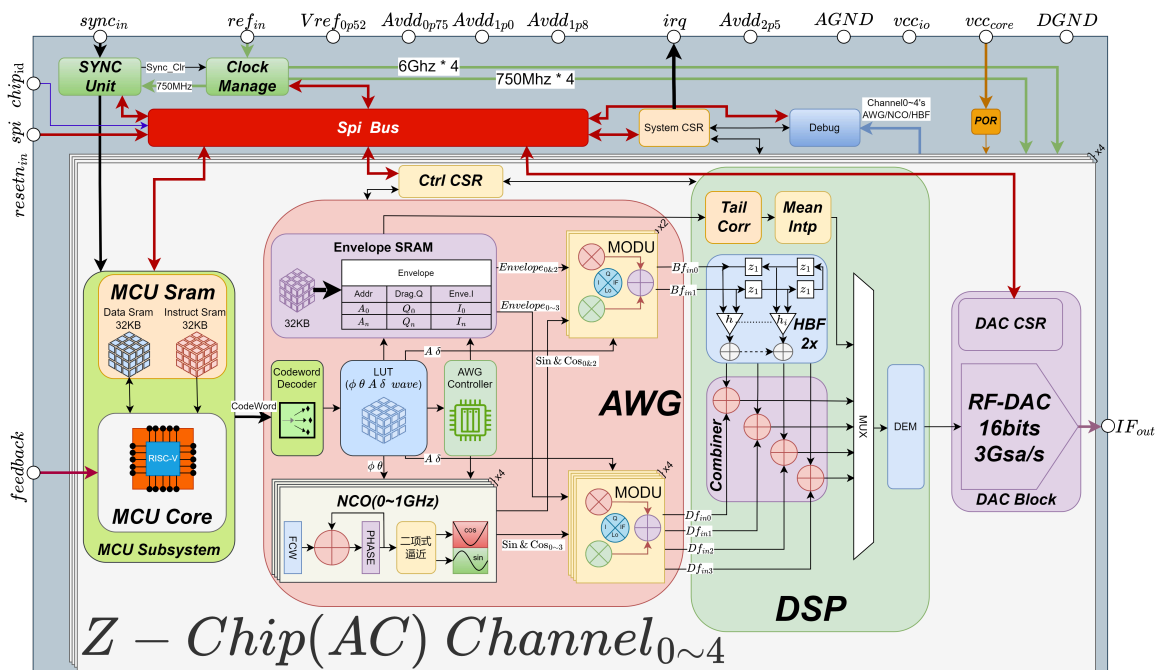
Z芯片是一款波形芯片，核心功能有：

- 1.RAMP台阶波形：固定模式输出给定的固定值。非固定模式可调台阶宽度和高度。
- 2.NCO正弦波形：可调频率、相位。（双音NCO也是一样的，只不过是倍频了）
- 3.AWG包络波形：输入自定义包络形状点数，可灵活配置包络波形（余弦，平顶，矩形包络）。
- 4.特殊调制波形：包络+NCO调制，支持幅度调制，可调频调相（NCO部分），双频点调制，可组合多种特殊波形：背靠背输出，组合调制波形，双频点波形等
- 5.拖尾矫正波形。上述AWG输出均可打开拖尾矫正开关。
- 6.偏置：直流可灵活叠加至上述任意波形。

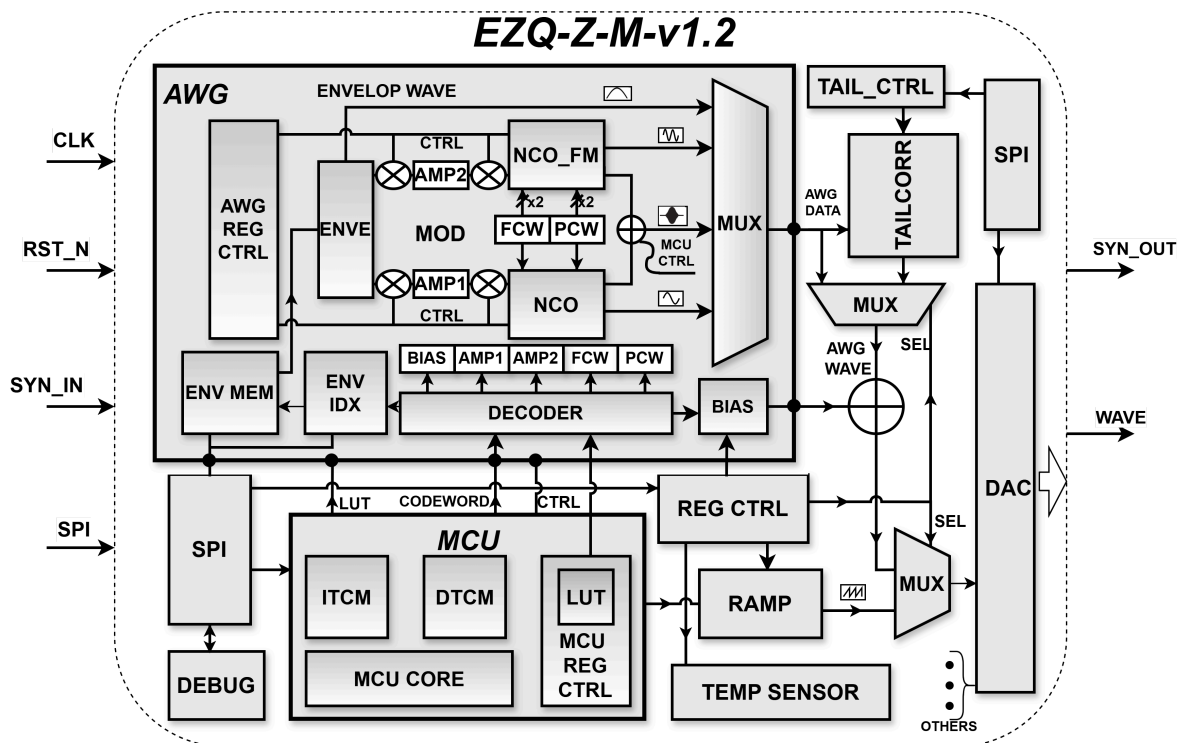
内置轻量RISV内核，通过该MCU以控制码字的形式，能够实现以上波形快速切换与灵活的组合。处理器的引入大大提高了波形灵活性。

1.2芯片架构图

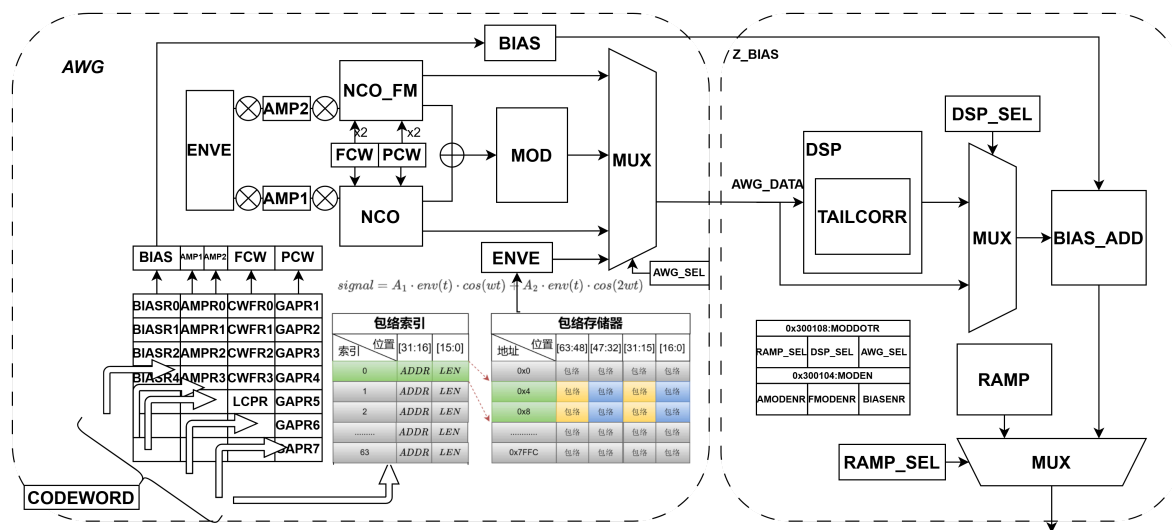
该架构图为彭工发给我的旧版图，实际代码可能有些地方不符合该架构，仅供参考。



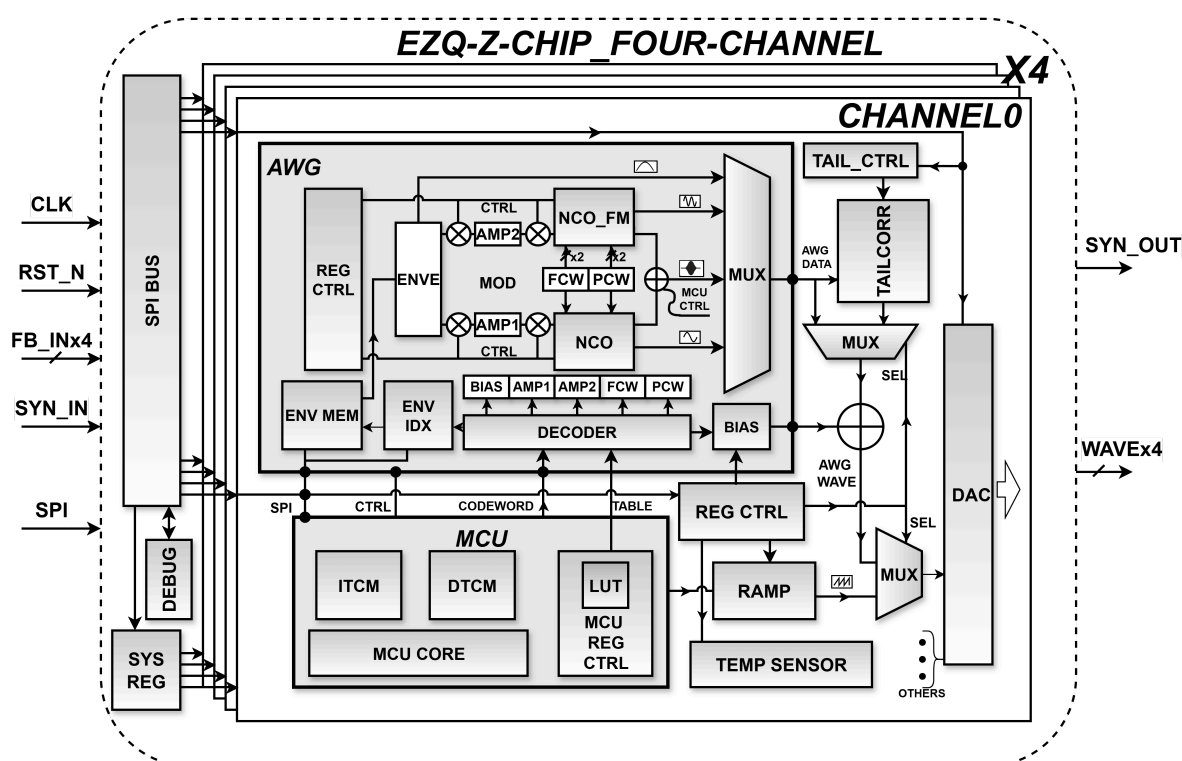
该图是我自己画的芯片架构图，是我精心画的（花费3天时间）芯片架构图，符合现有芯片架构。



和我画的芯片架构图类似，但是又包含了一些其他地方省略掉的细节，可以补充看看。

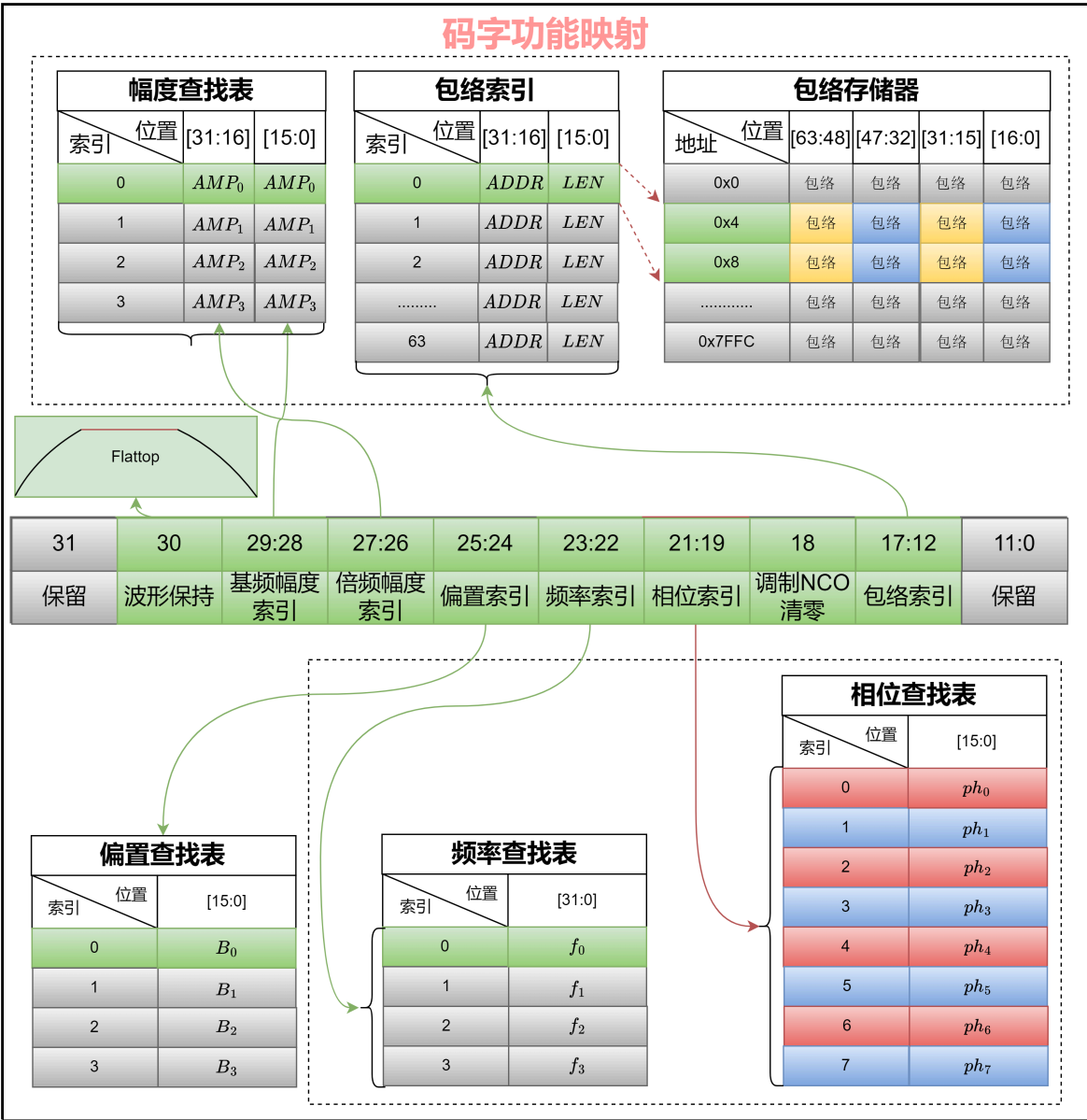


四通道Z芯片架构图

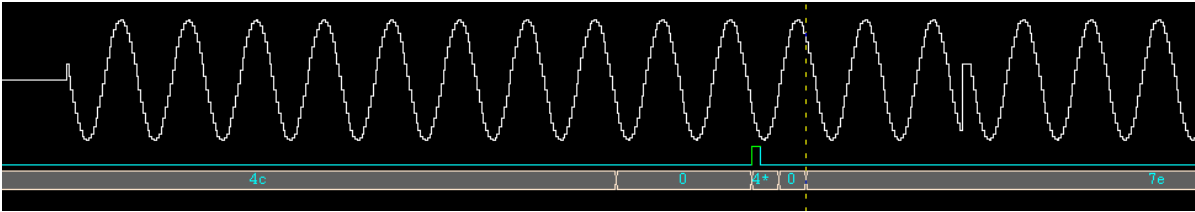


1.3波形控制

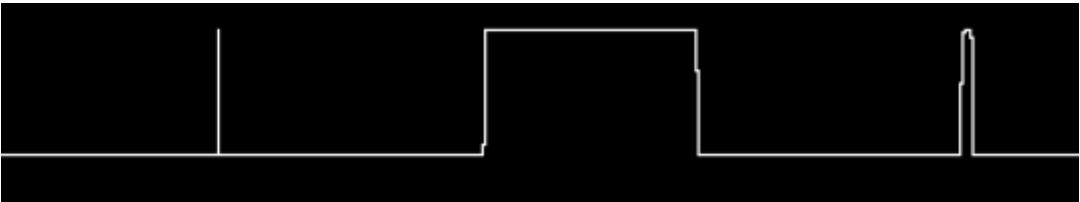
该芯片是基于RISV内核实现的波形控制功能。核心是码字控制波形，实现波形灵活组合和快速切换。一个码字就对应一个波形。MCU实时发送变化的码字，就实现了时候发送变化的波形，实现扫频扫相等功能。搭配灵活的包络存储机制，指令控制机制，能够实现几乎涵盖了所有量子测控所需要的波形。如下图码字功能映射表所述。

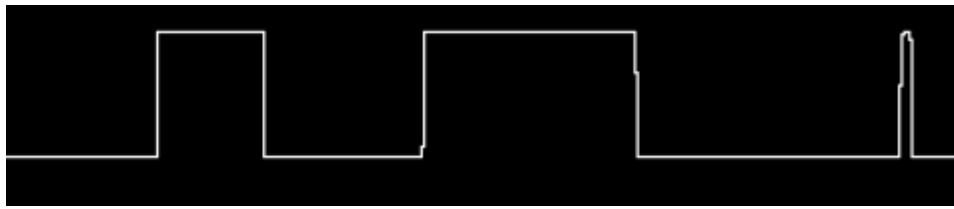


注意如果连续发送码值，那么每一次码字生效时会重新刷新波形状态，某些情况，比如NCO直出的情况会出现波形断点的情况。



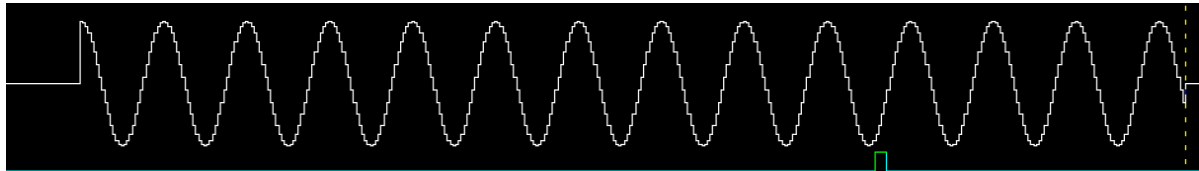
波形保持位，保持包络最后一个点的值不变，直到下一个码字刷新为止。比如下面第一个矩形波保持包络，如果没有波形保持，就很短，有包络保持的话，就可以等到下一个码字刷新的时候再拉低。



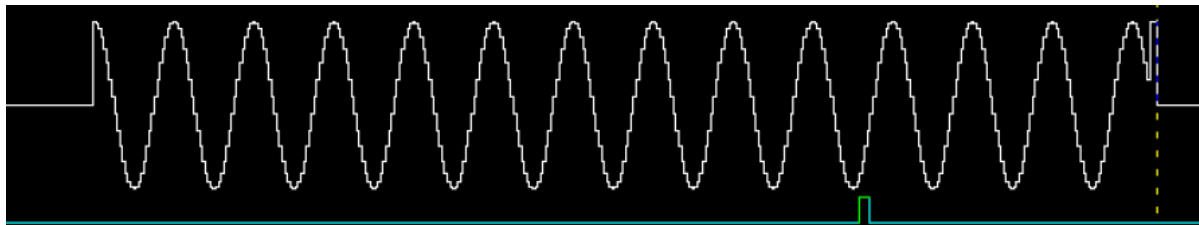


NCO清零位，可以让当前码字的NCO复位，这个的作用是满足重新演化的需求，比如在线路初始化的时候，如果要从0相位开始演化，就要给这个相位一个相位清零脉冲，这个相位清零就是这个NCO clear位。如下图，由两个包络组成的一个rect包络，第二个包络必须nco clear必须为0，这样第二个包络和第一个包络的NCO处于同一个演化进程，波形就不会断裂。

```
{'wave_hold': 1, 'fcw_index': 0, 'pcw_index': 0, 'code_clr': 1, 'env_index': 0},
{'wave_hold': 0, 'fcw_index': 0, 'pcw_index': 0, 'code_clr': 0, 'env_index': 1},
```



如果把这两个码字的nco_clear同时变为1，则可以观察到后面第二个码字的相位割裂了。（这边相位演化不能重置，因为是在同一个波形演化中）



1.4 芯片RTL代码

1.顶层端口。包括：同步，反馈，SPI，（PLL控制，DAC_config和衰减器），4x16bit的dout

2.SPI。SPI分2路，一个控制PLL控制寄存器，一个为mst，总线挂载多路slv，其中slv0是系统寄存器，slv1是指令存储器，slv2是数据存储器，MCU控制，slv3是控制寄存器，slv4是包络id寄存器，slv5是包络存储寄存器，slv6是dac寄存器。slv25是debug模块。

3.MCU。MCU主要有2个存储，一个是itcm指令寄存器，一个是dtcm数据寄存器。ITCM（指令）SPI一次性写入代码，运行全程只读；dtcm程序运行全程频繁读写，每时每刻都在改。MCU发射的数据帧是一个32位的send包，里面都是索引信息。结构如上述码字功能映射表所述，通过索引来控制波形，从而实现快速灵活切换的功能。mcu主要分布在channel_top里面。mcu是一个自定义 RISC-V 微型 MCU 内核（qbmcpu）。完整数据流链路。取指链路 `u_qbmcu.ifu_o_req_pc` → ITCM PortA 地址 → ITCM 读出指令 → `ifu_rsp_instr` 送回 CPU 译码，数据读写链路：CPU 执行 Load/Store → `agu_o_*` 地址 / 数据 / 控制信号 → DTCM PortA → DTCM 读写波形 / 变量数据 → `agu_i_rddata` 读数据回 CPU。**辅助模块**有运行时间计数器，取指令数计数器，总线译码（dsram,preg,其中preg控制mcu_regfile），mcu_regfile。MCU_乘法器。ITCM和DTCM两块dram

4.ctrl模块。ctrl进入channel_top又分两路，分别是ctrl_s0和ctrl_s1。ctrl_s0直接控制ctrl_regfile，负责温度传感器，ramp模块，MCU相关的。ctrl_s1控制tc_regfile拖尾矫正模块。

5.AWG模块。MCU可以通过send索引包来控制AWG,还可以通过外部ext_send包直接控制。AWG关联着两块包络存储，包络id和包络数据存储，输出端口有两个，awg_data，还有一个是enve_data（冗余），z_bias。数据流是AWG内部四选一（env,nco,nco_fm,module四选一），然后和DSP信号再进行二选一，然后再和AWG的z-bias叠加起来。module调制信号有一个计算公式， $\text{amp} \times \text{enve} \times (\text{nco})$

+nco_fm)。

6.系统寄存器。系统寄存器控制dbg模块，通道0，通道1，通道2，通道3端口以及其软复位

1.5 模块接口

1.5.1 z_awg_top

1.从MCU进来的

```
//-----from mcu-----
//lookup table data
,input [31 :0] mcu_cwfr[3:0] // Carrier frequency ctrl word 0~3
,input [15 :0] mcu_gapr[7:0] // Carrier phase ctrl word 0~7
,input [15 :0] mcu_ff_ampr[3:0] // Fundamental Frequency Amplitude 0~3
,input [15 :0] mcu_fm_ampr[3:0] // Frequency Multiplication Amplitude 0~3
,input [15 :0] mcu_baisr[3:0] // Bais 0~3
//CFG Port
,input mcu_nco pha_clr
,input [15 :0] mcu_rz_ff pha
,input [15 :0] mcu_rz_fm pha
,input [0 :0] mcu_fm en
// The operands and info to peripheral
,input send
,input sendc
,input [31 :0] codeword
,input [1 :0] fb_st
```

mcu中前五个即mcu_cwfr,gapr,ampr (2个) , baisr都是来自mcu_regfile给出的。而mcu_ff_ampr其实就是mcu_ampr前16位, mcu_fm_ampr 是其后16位。CFG Port也是来自mcu_regfile。 send和sendc和codeword来自qbmcu,最后一个fb_st来自ctrl_regfile。

MCU可以通过指令，载波频率，各通道相位，基波幅度，调频谐波幅度，输出直流偏置。

2.两个存储器，SPI可配，AWG可读可写

```
0 //-----from spi-----
1 //Envelope storage read/write signal
2 ,input [31 :0] enve_bwrdata
3 ,input [0 :0] enve_bwren
4 ,input [15 :0] enve_brwaddr
5 ,input [0 :0] enve_brden
6 ,output [31 :0] enve_brddata
7 //envelope index lookup table read-write signal
8 ,input [31 :0] enve_id_bwrdata
9 ,input [0 :0] enve_id_bwren
10 ,input [7 :0] enve_id_brwaddr
11 ,input [0 :0] enve_id_brden
12 ,output [31 :0] enve_id_brddata
```

3.给控制寄存器的状态

```
//-----to ctrl regfile-----
//Envelope read fsm status
,output [0 :0] enve_read_fsm_st
//Process conflict
,output proc_cft
```

控制寄存器能控制的

```
//-----from ctrl regfile-----
,input mod pha_sftot_clr
,input [1 :0] mod_enable // [1] --> 1'b1: Amod_enable;1'b0: Amod_disable;
// [0] --> 1'b1: fmod_enable;1'b0: fmod_disable;
,input [1 :0] mod_dout_sel // 2'b00 --> modem data ; 2'b01 --> enve data ;
// 2'b10 --> nco_cos data ; 2'b11 --> nco_sin data;
,input tc_sel
,input mod_nco_alwayson
```

给DSP的

```

//-----to DSP-----
//Output awg data
, output [15 :0] awg_data[PARALLEL-1 : 0]
, output          awg_vld
, output          bais_ov
, output [15 :0] enve_data[PARALLEL - 1: 0]
, output          enve_vld
, output [15 :0] z_bais
);

```

1.5.2 z_dsp

dsp里面系数生成模块，拖尾矫正模块，采样速率适配模块。

```

5 z_dsp u_z_dsp(
5     .rstn              (rst_n & ~tc_clr      ),
7     .clk               (clk                  ),
3     .en               (z_dsp_en             ),
3     .vldi_coef        (vldi_coef            ),
0     .vldi_data        (tc_on                 ),
1     .din              (awg_data             ),
2     .a0_re            (tcparr0               ),
3     .a0_im            (tcpair0               ),
4     .b0_re            (tcpbrr0               ),
5     .b0_im            (tcpbir0               ),

```

系数有效，数据有效，输入数据，系数端口

```

3     .b7_im            (tcpbir7               ),
1     .dout_0           (z_tc_data0            ),
5     .dout_1           (z_tc_data1            ),
5     .dout_2           (z_tc_data2            ),
7     .dout_3           (z_tc_data3            ),
3     .vldo            (z_tc_vld              ),
3     .overflow         (overflow_z_dsp        )
);

```

拖尾数据输出、溢出端口。

1.5.3 ramp模块

```

ramp_gen U_ramp_gen (
//system port
.clk              (clk                  ),
.rst_n            (rst_n                ),
.cen              (ramp_en              ),
.step            (ramp_step             ),
.ifs             (ramp_ifs              ),
.fixed           (ramp_fixed            ),
.fixed_value     (ramp_fixed_value      ),
.ramp            (ramp_data             ),
.ramp_vld        (ramp_vld              )
);

```

使能。台阶高度step,台阶宽度ifs。斜坡固定还是线性开关。固定值。斜坡输出数据。

1.5.4 mcu_regfile模块

读写操作口，mcu_regfile通过读写操作口来控制蛮多东西的。这个操作口是总线译码器的小分支，另外一个分支连到了dsram上面。大动脉是agu_o的一个ram操作口，直接连到了qbmcu上。

```
//rw op port
, input [31 :0] wrdata
, input      wren
, input [3 :0] wrmask
, input [15 :0] rwaddr
, input      rden
, output [31 :0] rddata
, input [7 :0] fb_st_info
, input [31 :0] run_time
, input [31 :0] instr_num
```

SPI和MCU交换数据

```
//MCU and SPI interface for interaction
, input [31 :0] mcu_param [3:0] // MCU parameter 0~3
, output [31 :0] mcu_result [3:0] // MCU result 0~3
```

控制查找表，配置端口。

```
//lookup table data
, output [31 :0] mcu_cwfr [3:0]
, output [15 :0] mcu_gapr [7:0]
, output [31 :0] mcu_ampr [3:0]
, output [15 :0] mcu_baisr [3:0]
//CFG Port
, output      mcu_nco pha_clr
, output [31 :0] mcu_detune_fcw
, output [31 :0] mcu_rz pha
, output [15 :0] mcu_dc_bias
, output [0 :0] mcu_dc_bias_send
, output      mcu_fmnen
```

PRNG 伪随机数发生器，和乘法器交互。

```
//prng signals
, output      prng_init_en
, output [31 :0] prng_seed
, output      prng_extract_en
, input [31 :0] prng_result
//, input      prng_result_vld
//mult signals
, output      mult_en0
, output [11 :0] mult_A0
, output [19 :0] mult_B0
, input [31 :0] mult_result0
, input      mult_result_vld0
//mult signals
, output      mult_en1
, output [15 :0] mult_A1
, output [15 :0] mult_B1
, input [31 :0] mult_result1
, input      mult_result_vld1
```

RAMP控制和中断输出控制。

```
, output      ramp_en
, output [15 :0] ramp_step
, output [31 :0] ramp_ifs
, output [15 :0] ramp_fixed_value
, output      ramp_fixed
//irq
, output      mcu_nexit_irq
```


1.5.5 qbmcu

```
qbmcu U_qbmcu
(
  .clk                ( clk                )
  .rst_n              ( rst_n              )
  .qbmcu_i_start      ( sync_int          )
  .qbmcu_o_fsm_st     ( qbmcu_o_fsm_st    )
  .ifu_i_pc_rtvec     ( ifu_i_pc_rtvec    )
  .ifu_o_req_pc       ( ifu_o_req_pc      )
  .ifu_o_req          ( ifu_o_req        )
  .ifu_rsp_instr      ( ifu_rsp_instr     )
  .dec_o_ilegl        ( dec_o_ilegl       )
  .agu_o_addr         ( agu_o_addr        )
  .agu_o_wrddata      ( agu_o_wrddata     )
  .agu_o_wren         ( agu_o_wren        )
  .agu_o_wrmask       ( agu_o_wrmask      )
  .agu_o_rden         ( agu_o_rden        )
  .agu_i_rddata       ( agu_i_rddata      )
  .agu_o_addr_unalgn  ( agu_o_addr_unalgn )
  .ext_o_send         ( ext_o_send        )
  .ext_o_sendc        ( ext_o_sendc       )
  .ext_o_codeword     ( ext_o_codeword    )
  .ext_o_intr         ( mcu_ext_o_intr    )
);
```

i = input输入到 qbmcu 内部。o代表输出，是mcu输出到外面的。

mcu开始工作通过sync_int启动，MCU状态机是可以看fsm_st来查看。

ifu是取指令，连到了ITCM存储的A口,ifu_o_req_pc是指令地址，而ifu_o_req相当于地址有效信号，拉高的话请求一条指令。ifu_rsp_instr就是取出的指令，拿给mcu译码，ifu_i_pc_rtvec叫中断返回的地址，中断结束后，从这个地址继续执行。而ITCM这个Ram的B口支持SPI直接读写指令。dec_o_ilegl是译码单元非法指令输出标志位，触发异常中断。

agu是控制两个东西，一个是dsram,一个是mcu_regfile。dsram连到dtcm存储。agu_o_addr_unalgn叫非对齐地址标志位，地址不是4字节对齐的话就拉高。

mcu通过ext_o_send和ext_o_codeword端口以码字的方式控制波形，一个码值对应一种波形，通过不断send码字，可以实现快速波形切换的效果，sendc是有条件的码字发送，可以连到反馈，实现反馈波形切换。ext_o_intr是mcu的中断输出。

1.5.6 同步与反馈

同步

```
36)
37 module z_digital_top # (
38   parameter RSVNUM = 10
39 ) (
40   //-----clock pin from pll-----
41   input      clk                // System Main Clock
42   //-----Power on reset pin from por-----
43   input      por_rstn           // Power on reset, active low
44   //-----digital IO-----
45
46   input      async_rstn         // hardware Reset, active low
47   //sync
48   input      sync_in            // Chip synchronization signal input, high pulse valid
49   output     sync_out           // Chip synchronization signal output, high pulse valid
50   //Feedback signal
51   input [1:0] ch0_feedback      // Ch0 Feedback signals from the readout chip
52   ifdef CHANNEL_IS_FOUR
53   input [1:0] ch1_feedback      // Ch1 Feedback signals from the readout chip
54   input [1:0] ch2_feedback      // Ch2 Feedback signals from the readout chip
55   input [1:0] ch3_feedback      // Ch3 Feedback signals from the readout chip
56   endir
```

外部同步叫做sync_in,用于外部触发波形输出，经过一个sync_buf缓冲一下变成sync_int信号，同时输出sync_out。这个sync_out是同步输出，在sync_int后一个时钟周期，而且被sync_oe_s这个信号使能控制，猜测应该是告诉外界我要输出波形了。

```

sync_buf                                #(
) U_sync_buf                            (
  .clk                                  ( clk                                )
  .rst_n                               ( pll_rstn_o                        )
  .ext_ena                             ( sync_oe_s                         )
  .clr_ena_sync                        ( sync_clr_en_s                     )
  .sync_in                             ( sync_in                          )
  .sync_int                            ( sync_int                         )
  .sync_sample                         ( msd_sync_pulse                    )
  .sync_ext                            ( sync_out                         )
);

```

这个sync_int 和内部同步脉冲都能够做同步触发源，变成真正控制内部脉冲的sync_pulse。同时被内部同步使能控制。

```

578
579 wire sync_src = (int_sync | sync_int) & int_sync_en;
580 wire sync_pulse;
581 pulse_generator pulse_inst_sync      (
582   .clk                                  ( clk                                )
583   .rst_n                               ( pll_rstn_o                        )
584   .pulse_en                            ( sync_src                         )
585   .delay                               ( sync_delay                       )
586   .width                               ( 16'd1                           )
587   .inv_en                              ( 1'b0                            )
588   .pulse                               ( sync_pulse                       )
589 );

```

反馈

每一个通道都有一个2位的反馈输入。读出芯片通过输入2位的反馈信号，控制Z芯片。反馈进来是进入到一个fb_st_in的8位寄存器中，然后进入mcu_regfile和ctrl_regfile中。对于mcu_regfile，锁存进进入到fsir寄存器。ctr_regfile也一样，也叫fsir。

```

`ifdef CHANNEL_IS_FOUR
wire [7:0] fb_st_in = {ch3_feedback, ch2_feedback, ch1_feedback, ch0_feedback};
`else
wire [7:0] fb_st_in = {2'b00, 2'b00, 2'b00, ch0_feedback};
`endif

```

应该是我们编写汇编让mcu主动去读这个反馈信息，然后进入对应的函数，执行操作吗，这样的话，就能实现反馈控制功能。**通过sendc实现反馈控制**

1.5.7 中断

查看IDS表，可以发现系统寄存器中 0x14 IMR 中断状态屏蔽寄存器。0x18 ISR 原始状态寄存器 0x40 IMSR 屏蔽后状态寄存器。这3个寄存器都是中断核心寄存器。仔细看可以发现，一共有6个异常状态。异常退出指令，译码错误，访存地址不对齐，指令冲突，溢出标志这六种。

这三个寄存器实现了一个屏蔽的过程：原始状态 ---》状态屏蔽 ---》屏蔽后状态。

0x14 IMR 中断状态屏蔽寄存器

0通道mcu非退出中断请求屏蔽位
0通道溢出标志中断屏蔽位
0通道指令冲突中断屏蔽位
0通道访存地址不对齐中断屏蔽位
0通道译码错误中断屏蔽位
0通道异常退出指令中断屏蔽位

0x18 ISR 原始状态寄存器

0通道mcu非退出中断请求原始中断状态位
0通道溢出标志原始中断状态位
0通道指令冲突原始中断状态位
0通道访存地址不对齐原始中断状态位
0通道译码错误原始中断状态位
0通道异常退出指令原始中断状态位

0x40 IMSR 屏蔽后状态寄存器

0通道mcu非退出中断请求屏蔽后中断状态位

0通道溢出标志屏蔽后状态位

0通道指令冲突屏蔽后状态位

0通道访存地址不对齐屏蔽后状态位

0通道译码错误屏蔽后状态位

0通道异常退出指令屏蔽后状态位

对应的是system_regfile中的input 的6个状态, 同时输出只要一个output类型的 irq, 这是整个芯片的中断直接送到最外层去了。

```
164 module system_regfile # (  
165     parameter CHIPCODE = 32'h58595343 // 32'h58595343:Single-chan  
166     ,parameter MFDATE  = 32'h20250831 // The production date is A  
167 )(  
168 //system port  
169     input      clk          // System Main Clock  
170     ,input      rst_n       // Spi Reset active low  
171 //rw op port  
172     ,input [31:0] wrdata    // write data  
173     ,input      wren        // write enable  
174     ,input [15:0] rwaddr    // read & write address  
175     ,input      rden        // read enable  
176     ,output [31:0] rddata   // read data  
177 //debug cfg port  
178     ,output      dbg_enable //active high  
179     ,output [3:0] dbg_ch_sel //4'b0001-->ch0;4'b0010-->ch1;  
180                               //4'b0100-->ch2;4'b1000-->ch3;  
181 //debug status Port  
182     ,input      dbg_upd  
183 //ch0 status Port  
184     ,input      ch0_proc_cft  
185     ,input      ch0_ldst_addr_unalgn  
186     ,input      ch0_dec_err  
187     ,input      ch0_exit_irq  
188     ,input      ch0_over_flag  
189     ,input      ch0_mcu_nexit_irq  
190 //ch1 status Port  
191     ,input      ch1_proc_cft
```

Verilog 端口名 (左侧)	中文描述 (右侧)	说明
ch0_proc_cft	0通道指令冲突原始中断状态位	名称中的 cft 是 Conflict (冲突) 的缩写, 这个是从awg_top的awg_proc_cft出来的
ch0_ldst_addr_unalgn	0通道访存地址不对齐原始中断状态位	ldst 是 Load/Store (访存) 的缩写; unalgn 是 Unaligned (不对齐) 的缩写, 这个是从qbmcu 的agu_o_addr_unalgn输出出来的
ch0_dec_err	0通道译码错误原始中断状态位	dec 是 Decode (译码) 的缩写; err 是 Error (错误),也是qbmcu中dec_o_ilegl出来的
ch0_exit_irq	0通道异常退出指令原始中断状态位	这个是从qbmcu 的ext_o_intr输出出来的
ch0_over_flag	0通道溢出标志原始中断状态位	over 对应 Overflow (溢出); flag 对应“标志位”, 是从ctrl_regfile中出来的
ch0_mcu_nexit_irq	0通道MCU非退出中断请求原始中断状态位	mcu 指 MCU; nexit 是 Non-Exit (非退出) 的缩写; irq 是中断请求, 是从mcu_regfile中出来的

这边的isr就是原始中断状态。 imr就是屏蔽位寄存器, 而misr就是屏蔽后的中断状态

```

7 // -----
3 // -- Interrupt Status Register - Read Only
9 //
0 // This register contains the status of all
1 // XYZ Chip interrupts after masking.
2 // -----
3 sirv_gnrl_dffr #(32) isr_dffr (iriser, isr, clk, rst_n);
4
5 //misr
6 wire[31:0] misr_w = imr & iriser;
7 sirv_gnrl_dffr #(32) misr_dffr (misr_w, misr, clk, rst_n);
8
9 //irq
0 wire irq_w = |misr;
1 sirv_gnrl_dffr #(1) irq_dffr (irq_w, irq, clk, rst_n);

```

最后是不是要拉高中断，取决于屏蔽后的中断状态，只要有一个是高，就拉高中断

```

wire irq_w = |misr; // 按位或操作，只要有一位是1，结果为1

```

二、RISV CPU工作原理

2.1指令结构

CPU有2个存储器，数据存储器 and 指令存储器。CPU工作时会不断从指令指令存储器拿指令，译码，执行操作。那么指令存储器的一条指令是32位的，分为哪些部分呢？有三种类型的指令，不过一般来说，分为：7位的操作码，rd目标寄存器，rs1源寄存器1，rs2源寄存器2，funct3功能码3，功能码7。

31	25 24	20 19	15 14	12 11	7 6	0	
+	+	+	+	+	+	+	
funct7	rs2	rs1	funct3	rd	opcode		R型
+	+	+	+	+	+	+	
imm[11:0]		rs1	funct3	rd	opcode		I型
+	+	+	+	+	+	+	
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode		S型
+	+	+	+	+	+	+	

比如一条汇编语句是: addi x6, x0, 12。含义: $x6 = x0 + 12 = 0 + 12 = 12$ 。在指令存储器中，就长这样。

31	20 19	15 14	12 11	7 6	0
+	+	+	+	+	+
立即数	rs1	funct3	rd	opcode	
12	x0(0)	000	x6(6)	0010011	
+	+	+	+	+	+

具体二进制:

立即数(12) = 000000001100

rs1(x0) = 00000

funct3 = 000

rd(x6) = 00110

opcode = 0010011

汇编: lw x31, 0xdc(x1), 含义: $x31 = \text{memory}[x1 + 0xDC]$ (从内存加载)

31	20 19	15 14	12 11	7 6	0
+-----+-----+-----+-----+-----+					
立即数	rs1	funct3	rd	opcode	
0xDC	x1(1)	010	x31(31)	0000011	
+-----+-----+-----+-----+-----+					
具体二进制:					
立即数(0xDC=220) = 000011011100					
rs1(x1) = 00001					
funct3 = 010 (字加载)					
rd(x31) = 11111					
opcode = 0000011 (加载指令)					

汇编: sw x31, 0xbc(x2), 含义: memory[x2 + 0xBC] = x31 (存储到内存)

31	25 24	20 19	15 14	12 11	7 6	0
+-----+-----+-----+-----+-----+						
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	
5位	x31	x2(2)	010	4位	0100011	
+-----+-----+-----+-----+-----+						
具体二进制:						
imm[11:5] = 0000101 (0xBC的高7位)						
rs2(x31) = 11111						
rs1(x2) = 00010						
funct3 = 010						
imm[4:0] = 11100 (0xBC的低5位)						
opcode = 0100011 (存储指令)						

2.2 通用寄存器

CPU 内部有**32 个独立小存储格子**，编号 `x0 ~ x31`，叫通用寄存器

只有这 32 个格子能做加减乘、判断、临时存数据，而外设 / 内存里的数据不能直接运算，必须先用 `lw` 读到这些寄存器，算完再用 `sw` 写回硬件。重点特殊寄存器（必记 2 个）`x0`：零寄存器，永远固定等于 `0`。`x1`：返回地址寄存器 `ra`。正常函数调用场景下，`x1` 专门存函数返回地址。但我们这里的 `x1` 是**参数块基地址**。

还要分两类:临时寄存器和保存寄存器。临时寄存器 `t0~t6`（对应 `x5`到`x7`、`x28`到`x31`），生命周期短（函数内），用完不需要保存。保存寄存器 `s0`到`s11`（`x8`到`x27`）需要保存和恢复操作，使用成本高，用完后要保证值不变，放一些重要变量。`t寄存器` = "temporary" → 用完就扔。`s寄存器` = "saved" → 要好好保存。指令更像是储存操作关系，而32个通用寄存器就是储存操作变量的，而结果要从通用寄存器store到dtcm中。

指令存储器（32KB）	= 存储"操作关系"（做什么）
通用寄存器（ <code>x0~x31</code> ）	= 存储"操作变量"（数据在哪）
DTCM（数据存储器）	= 存储"操作结果"（最终数据）

2.3汇编语法

1.加3个0的赋值--lui

```
lui x1, 0x100    # x1 = 0x100 << 12 = 0x100000
```

2.读1字节，符号扩展到32bit的赋值--lb

```
lb rd, offset(rs1) #从该地址读出 1 字节（8bit），符号扩展到 32bit，放到rd
```

3.加立即数--addi

```
addi x3, x0, 8    # x3 = 0 + 8 = 8
addi x4, x4, -1   # x4 = x4 - 1
```

4.两寄存器相减--sub

```
sub x5, x6, x7    # x5 = x6 - x7
```

5.左移立即数--slli

```
slli x10, x9, 2   # x10 = x9 << 2
```

6.相等跳转--beq

```
beq x1, x2, label # if (x1 == x2) 跳转到label
```

7.不等跳转--bne

```
bne x5, x0, loop  # if (x5 != 0) 跳转到loop
```

8.跳转并链接--jal

```
jal x0, exit      # 跳转到exit（不保存返回地址）
jal x1, function   # 跳转到function，返回地址存x1
```

`jal ra, func` = 跳转去执行 func 函数，同时记住你现在的位置存到 ra，函数跑完用 ret 就能原路回来继续执行。比如

```
# 主程序
jal ra, load_ramp_param # 跳去加载斜坡参数，返回地址存在ra
# 函数执行完ret后，回到这里继续执行

load_ramp_param:
    lw x31, 0xdc(x1)
    sw x31, 0xb8(x2)
    ret    # 跳回jal的下一行
```

9.自定义发送码字指令--send

`send x0, x24, 0` x0 = 丢弃结果, x24 里存着 32bit 波形控制码字，立即数0 可能代表普通波形配置下发模式

10.自定义延时指令 --wait

`wait x0, x23, -36` CPU 原地等待 x23-36 个时钟周期，保证前后两段波形时序不冲突。其中第一个 x0表示不需要保存延时结果,x23存放基础等待时钟周期数,imm = -36，代表最终等待时钟周期位x23的值 - 36个时钟周期。普通波形最小间隔 ≥22 个系统时钟，带 PRNG 随机扰动波形最小间隔 ≥34 个系统时钟

2.4 MCU寄存器

对于MCU来说,0x000_000是ITCM基地址，0x100_000是数据存储器DTCM的基地址，0x200_000是MCU寄存器（mcu_regfile）的地址。不是IDS表中常规基地址映射！mcu可以控制一些spi访问不到的寄存器，这些寄存器在IDS表中找不到，唯一在SPI和MCU之间，都能找到是MCU结果寄存器MCURES，和SPI可写的MCU参数寄存器MCU PARAM。这个MCU寄存器（mcu_regfile）其实就是我们常说的外设寄存器，MCU的关键功能之一是把内存（DTCM）中的数据搬移到MCU寄存器中，实现外设的寄存器配置，从而实现控制功能。

MCU波形控制寄存器功能表

寄存器名称	地址(偏移)	功能描述
CWFR0	0x40	查找波形频率1
CWFR1	0x44	查找波形频率2
CWFR2	0x48	查找波形频率3
CWFR3	0x4C	查找波形频率4
CWPRR	0x50	最高位写1，NCO复位
GAPR0	0x54	查找表波形相位0
GAPR1	0x58	查找表波形相位1
GAPR2	0x5C	查找表波形相位2
GAPR3	0x60	查找表波形相位3
GAPR4	0x64	查找表波形相位4
GAPR5	0x68	查找表波形相位5
GAPR6	0x6C	查找表波形相位6
GAPR7	0x70	查找表波形相位7
LCPR	0x74	附加相位
AMPR0	0x78	查找表波形幅度0 32bit ,高16 A1 低16是A2
AMPR1	0x7C	查找表波形幅度1
AMPR2	0x80	查找表波形幅度2
AMPR3	0x84	查找表波形幅度3
BIASR0	0x88	查找表波形偏置0
BIASR1	0x8C	查找表波形偏置1
BIASR2	0x90	查找表波形偏置2
BIASR3	0x94	查找表波形偏置3
RTIMR	0x98	mcu运行时间

寄存器名称	地址(偏移)	功能描述
ICNTR	0x9C	指令数量
FSIR	0xA0	反馈
DCBVR	0xA4	偏置大小
FMER	0xB4	双频点使能
MCU_RAMPFIXR	0xB8	Ramp固定值寄存器, [15]是常数使能位, [31:16]是固定值,
MCU_RAMPSR	0xBC	Ramp步进寄存器
MCU_RAMPIFSR	0xC0	Ramp宽度寄存器
MCU_RAMPENR	0xC4	Ramp使能寄存器, [31]是RAMP使能位,
PRNGSDR	0xC8	z没用
PRNGRESR	0xCC	z没用
MULTRO	0xD0	乘法器结果寄存器0
MULTR1	0xD4	乘法器结果寄存器1
DTFR	0xD8	z没用

2.5 MCU操控实例

```
def _single_send_instruction_template(self, **kwargs):
    return f"""
    start:
        lui x1, 0x100          # 设置数据存储器基地址
        lui x2, 0x200          # 设置控制寄存器基地址
        lw x3, 0x40(x1)        # 读取频率控制字
        sw x3, 0x40(x2)        # 写入频率控制字
        lw x3, 0x54(x1)        # 读取相位控制字
        sw x3, 0x54(x2)        # 写入相位控制字
        lw x3, 0x78(x1)        # 读取幅度控制字
        sw x3, 0x78(x2)        # 写入幅度控制字
        lw x3, 0x88(x1)        # 读取偏置控制字
        sw x3, 0x88(x2)        # 写入偏置控制字
        lw x3, 0x00(x2)        # 读取循环次数
        lw x4, 0x04(x2)        # 读取相邻两次波形间隔
        lui x5, 0x40
        loopstart:
            send x0, x5, 0      # 发射码字, 选择包络idx=0, 幅度idx=0, 频率idx=0, 发射前NCO累加器清零
            wait x0, x4, -12    # wait, addi, bne各占3clk, send本身占3clk
            addi x3, x3, -1
            bne x3, x0, loopstart
            exit x0, x0, 0      # 退出MCU不会终止NCO输出
    """
```

x1代表dtcm,x2代表mcu_regfile映射。从dtcm的0x40位置读出数据到x3, 再放到查找表频率寄存器0的位置。同理, 从dctm的0x54位置读出数据, 放到查找表波形相位寄存器0的位置。同理, 频率, 相位, 幅度, 偏置都放到查找表对应的位置了, 然后再从mcu参数寄存器拿到循环次数和波形间隔 (SPI写入的)。参数都准备好了, 发送码字0x40。

31	30	29:28	27:26	25:24	23:22	21:19	18	17:12	11:0
保留	波形保持	基频幅度索引	倍频幅度索引	偏置索引	频率索引	相位索引	调制NCO清零	包络索引	保留

对应就是调制NCO清零，其他全部选0。这时候，code_word就会变40，send拉高一个clk,波形就出来了，等待x4个clk周期，循环次数减一，循环次数x3!=0,就一直进循环，等到x3==0时候，进入exit退出MCU,但NCO不会终止。

总结一下，首先，MCU要把波形控制字搬到查找表位置。然后，读取循环次数和间隔参数，进入循环发送码字。AWG的工作就是拿着码字，去查找表中找到所需的波形控制字，根据波形控制字生成波形。

AWG就像一个生病的病人，是拿着药方去抓药，抓的药的组合就像是我们需要的波形。而MCU就像是医生，每个药放在哪个格子里面，要吃几次，隔多少小时吃一次，都是他说的算。

2.6 汇编模板

AssemblyTemplateManager类中，准备了几大模板，解决操作芯片时汇编语言太过难懂的问题。因为，每次我们要输入的也就是那几个参数，比如，波形循环几次，波形持续时间，码字是什么，查找表里面波形控制字是什么。所以我们只需要选择模板传入对应的参数，就可以实现对应的汇编指令控制。

```
def __init__(self, mk_instance, mk_instr, **kwargs):
    self.mk = mk_instance
    self.mk_instr = mk_instr
    self.config_file = kwargs.get('config_file')
    self.templates = {
        'ramp': self._ramp_mcu_template,
        'ramp_fixed': self._ramp_mcu_fixed_template,
        'awg_nco': self._awg_ff_nco_test_template,
        'awg_nco_clr': self._awg_ff_nco_clear_test_template,
        'awg_fm_nco': self._awg_fm_nco_template,
        'awg_env': self._awg_env_template,
        'awg_mod': self._awg_mod_template,
        'common_send': self._send_wait_pair_template,
        'common_send_bias_temp': self._send_wait_global_bias_pair_template,
        'awg_mod_nco': self._awg_mod_rect_hold_template,
        'dsp_env': self._dsp_env_template,
        'single_send': self._single_send_instruction_template,
        'hold_send': self._send2_instruction_template,
        'mul_16x16': self._mul_16x16_template,
    }
```

比如single_send模板的功能就是简单的波形发送，参数准备：需要频率、相位、幅度、偏置需在dtcm中准备好，循环次数、波形间隔需在mcupara中写好。这边码字是写死的，是0x40,其中第18位是1，其他是0。准备好参数，丢给模板，就能实现波形灵活发送。然而，由于汇编太过复杂，看是很难看懂的，加上需要配置的参数众多，有些在AssemblyTemplateManager里面模板配，有些又在其他地方配了，所以我们不用管汇编，直接用最顶层生成case即可。

三、测试用例

3.1 生成脚本

CASE使用Python代码生成。顶层是z_case.ipynb，其他都不用管，都是被这个顶层调用的。

make_inst.py这个是汇编指令文本翻译成二进制码的底层函数文件。reg_define.py是芯片寄存器基地址和偏移地址定义，还包括了拖尾参数定义。make_case.py是单次读写指令生成和指定类型数据生成的底层函数。EnvelopeGenerator.py这个是包络存储生成和包络索引生成指令的函数文件。ChipConfig.py是芯片寄存器配置的关键文件，执行关键寄存器读写操作。

3.1.1python生成

先运行第一个单元格，再运行下面的。因为第一个单元格调用了依赖，导入了底层的python文件，宏定义了地址。params是参数字典，有一系列的键-值对组成，我们要配置什么模式，输入什么参数，都往里面填。param_combinations 没用不用管。加文件名参数就用update，FolderName就是case所在文件夹名字，CaseName就是TXT名字，config_file由上面两个组成，就是相对于这个z_case.ipynb的路径，决定Case放哪里。configure_chip_output函数是核心，就是把参数字典转化成配置芯片的寄存器的二进制指令，并且存成txt.mk.rw_once是单次读写某个寄存器地址，把一次读写操作指令串，追加到对应路径的文本的txt下。这边写了两个寄存器没什么用，是为了防止仿真器报错的。我们可以在该文件夹下创建一个ParamsManager类，然后把参数保存为一个同名的json文件，生成case的同时生成一个json参数文件，这样我们可以清楚知道这个case对应的参数给了什么，后面也可以用这个json重新构建文件。

```
params = {
    'chip_mode': 'RAMP',
    'ramp_enable_source': 'SPI',
    'height': 16,
    'step_time': 30*10**8, # clk
    'fixed_enable': False,
    'custom_registers': [
        (f"0x{addr_base['DACR0_BASE']+0x30:x}", [0]*4),
        (f"0x{addr_base['DACR0_BASE']+0x78:x}", str(0x7400154)),
    ],
}

param_combinations = []
params.update({
    'FolderName': 'RAMP',
    'CaseName': f'RAMP_SPI_H{params['height']}_W{params['step_time']}',
})
params.update({
    'config_file': params['FolderName'] + '/' + params['CaseName'] + '_HEX.txt'})
configure_chip_output(mk, **params)
mk.rw_once(op: 'w', addr_base['ENVM0_BASE'], data: 0, params['config_file'])
mk.rw_once(op: 'w', addr_base['ITCM0_BASE'], data: '0x2B', params['config_file'])

pm = ParamsManager(params['FolderName'])
pm.save(params, params['CaseName'])
```

还有一个就是参数扫描。ramp_step_list是一个列表，里面可以填很多，这边只循环3次，step=128,256,512。首先要把json的文件夹和名字用FolderName和CaseName传进去，就可以根据ramp_step_list列表进行扫描，传1个参数，生成一个case，传10个参数生成10个case。

```

> # %%
# ramp_spi_fixed_scan
ramp_step_list = [128, 256, 512]
FolderName = 'RAMP'
CaseName = 'RAMP_SPI'
pm_load = ParamsManager(FolderName)
for ramp_step in ramp_step_list:
    params = pm_load.load(CaseName)
    params['height'] = ramp_step
    params['step_time'] = 250000000
    params.update({
        'FolderName': 'RAMP',
        'CaseName': f"RAMP_SPI_H{ramp_step}_W250000000"
    })
    params.update(
        {'config_file': params['FolderName'] + '/' + params['CaseName'] + '_HEX.txt'})
    pm_save = ParamsManager(params['FolderName'])
    pm_save.save(params, params['CaseName'])
    configure_chip_output(mk, **params)

```

3.1.2 json文件生成

使用python代码生成CASE有不够灵活的局限性，比如换一种工作模式，Python代码就需要重构对应的代码单元格，没办法灵活组合多种参数。其次，python代码生成的Case参数不够清楚，我们只能看到并且修改当前python代码配进去的参数，而没法看到底层大量的参数，修改这些底层参数也非常困难。最后，python的参数配置起来代码冗长，一旦配置的东西多起来，代码行数就非常长。

因此我们使用json文件来生成case，是一种更优的选项，通用统一且更加灵活。一个json文件就是一个case的所有参数列表。

```

D: > shortname > Z-Chip > case-generator > AWG_ENV > {} AWG_ENV.json >
1  {
2    "chip_mode": "AWG_ENV",
3    "instr_type": "awg_mod",
4    "inner_sync": true,
5    "envelope_configs": [
6      {
7        "envelope_type": "rect_hold",
8        "amp": 32767
9      },
10     {
11       "envelope_type": "flattop_hold",
12       "amp": 32767,
13       "edge_time": 3.5,
14       "wave_time": 30
15     },

```

把json文件的路径输入进去，python就会根据json里面的参数生成case，生成的case的位置放在json代码末尾config_file位置。

```

[22] 1 # load params
2 FolderName = 'D:/yangshenbo/workspace/temporary/case'
3 CaseName = 'AWG_ENV'
4 pm = ParamsManager(FolderName)
5 params = pm.load(CaseName)
6 configure_chip_output(mk, **params)
7 env_config(mk, **params)
8 instruction_config(mk, mk_instr, **params)
在 2026.07.15 11:25:52 于 123ms内执行

```

json代码里面的位置决定Case放哪里，而python里面load params函数中的路径是指向json文件的。

```
    "instr_show": false,
    "FolderName": "AWG_ENV3",
    "CaseName": "AWG_ENV",
    "config_file": "AWG_ENV3/AWG_ENV3_HEX.txt"
}
```

实际生成的时候，改完json运行一次load params函数，改完一次json，运行一次load params函数，这样就能灵活实现生成所有case。

3.1.3手动生成

除了上述的两种测试用例生成方式，我们还要会手动改写case。case的数据结构为：

序号	数据结构	长度	比特序	功能描述	备注	
1	CMD	W/R	1 bit	[63]	读写标识，0：W；1：R	/
		ARD FLAG	1 bit	[62]	主动回读数据标志	主动回读模块根据读出芯片读请求读出实验数据
		CID	5 bits	[61:57]	芯片组中每个芯片的ID标识	包括量子测控芯片组及其他需要配置的芯片组
		ADDR	25 bits	[56:32]	低25位地址	可与EXADDR拼接使用，或直接映射到量子芯片地址
		EXADDR	12 bits	[31:20]	扩展地址， 可根据需求划分为多个功能区 (4比特预留+5比特槽位 +3比特板卡内部预留)	可与ADDR拼接使用此时寻址空间为128GB， 该4096K的地址空间可按照需求映射给集控系统
		LENGTH	20 bits	[19:0]	指示读写数据长度， 以Byte为单位	最大支持1MByte
2	DATA	N*4 bytes	[31:0]	数据	读操作时，无需包含该项	

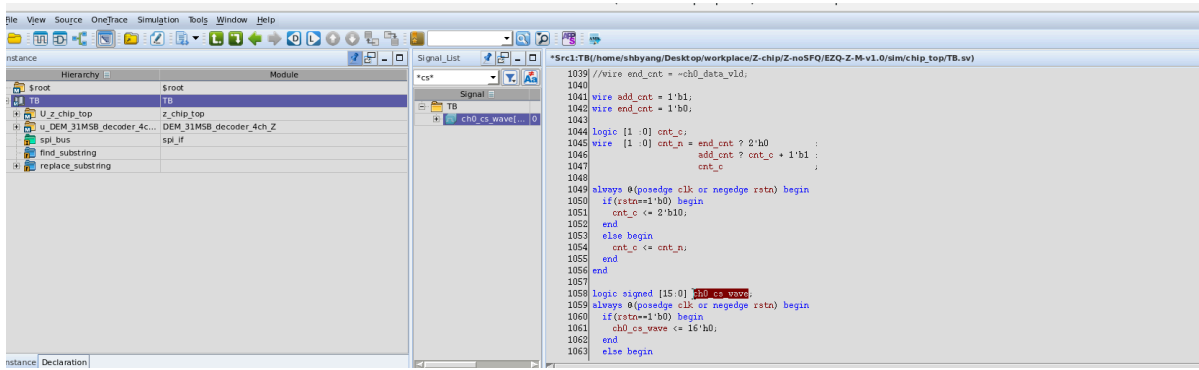
要深入的了解芯片的功能，完全灵活的使用芯片，不能单单只看python和json相关的代码参数，而是应该根据芯片IDS表看懂case.txt文件，并且知道代码中的硬件结构，而且关键寄存器要背下来。比如，打开txt文本，我们看到如下的3行数字。代表写300108寄存器（数据选择寄存器），拓展地址是1，长度是4，写入数据是最后一行0000_0000。

```
00300108
00100004
00000000
```

3.2仿真平台

我们使用VCS+Verdi作为芯片仿真平台，仿真是在Linux操作系统下操作的，case生成可以在WINDOWS下操作。首先搭建相关虚拟机环境和CASE生成环境，然后在WINDOWS下生成好Case。

- 1.移植：建一个文件夹，把case都拷进去。压缩，存邮箱草稿。虚拟机打开邮箱网站，下载case，解压。unzip test.zip -d ./test/,再剪切到./case/config目录下。
 - 2.在chip_top 终端输入make regress选择你前面传进来的文件夹。等待跑完。
 - 3.输入cd ./work_RTL/ ,在你想要的文件夹下 打开终端make dbg，使用verdi。
- TB中就看ch0_cs_wave这个波形就行了。

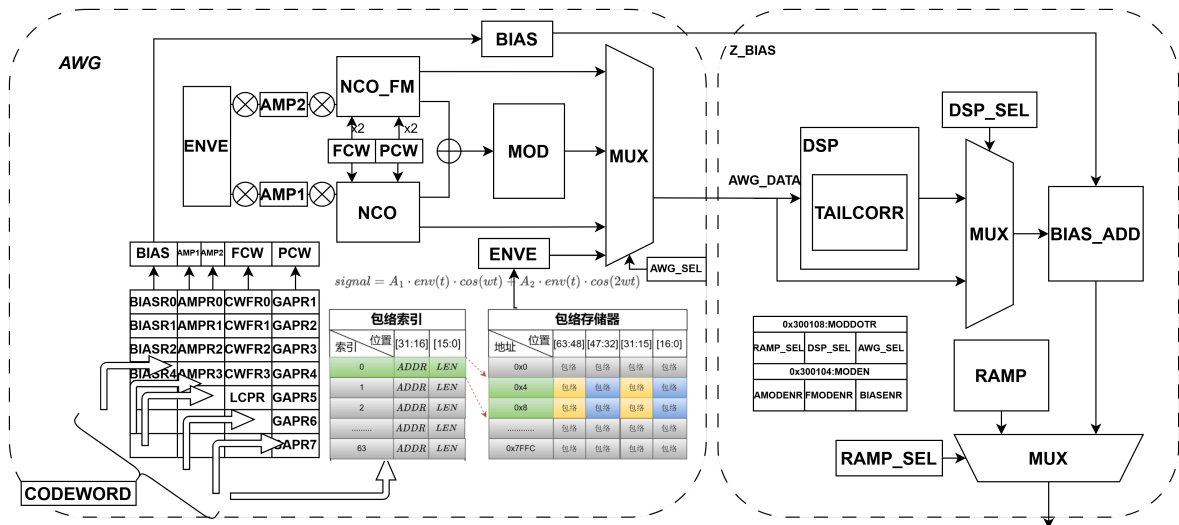


3.2 芯片模式的配置

芯片有很多工作模式，其中`chip_mode`参数包含6种模式：

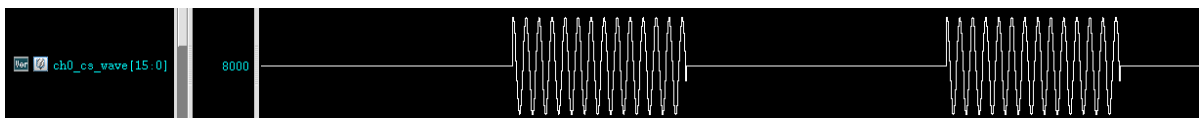
- RAMP：RAMP数据输出
- AWG_NCO：单音NCO数据输出
- AWG_FM_NCO：双音NCO数据输出
- AWG_ENV：包络数据输出
- AWG_MOD：调制数据输出
- DSP模式
 - DSP_NCO：拖尾校正输出，数据源为单音NCO，无实际作用
 - DSP_FM_NCO：拖尾校正输出，数据源为双音NCO，无实际作用
 - DSP_ENV：拖尾校正输出，数据源为包络，不可扫描
 - DSP_MOD：拖尾校正输出，数据源为调制数据，可以做参数扫描。

我们可以根据case观察到不同工作模式需要配哪些寄存器



3.2.1 AWG_MOD

1. 波形实例



$$signal = A_1 \cdot env(t) \cdot cos(wt) + A_2 \cdot env(t) \cdot cos(2wt)$$

2.测试用例

```
# AWG_MOD
params = {
    'chip_mode': 'AWG_MOD',
    'instr_type': 'awg_mod',
    'envelope_configs': [
        {'envelope_type': 'rect_hold', 'amp': 32767},
        {'envelope_type': 'flattop_hold', 'amp': 32767, 'edge_time': 3.5, 'wave_time': 30},
        {'envelope_type': 'acz', 'amp': 32767, 'wave_time': 12*4},
        {'envelope_type': 'accz', 'wave_time': 12*4},
        {'envelope_type': 'cosine', 'amp': 32767, 'wave_time': 12*25},
        {'envelope_type': 'rect', 'amp': 32767, 'wave_time': 12},
        {'envelope_type': 'flattop', 'amp': 32767, 'edge_time': 2, 'wave_time': 22},
    ],
    'amp_mod_enable': True,
    'freq_mod_enable': True,
    'bias_enable': True,
    'fm_enable': False,
    'fcw':[100, 200, 300, 400], # MHz
    'mcu_reg_clr': False,
    'pcw':[0, 30, 45, 90, 120, 135, 150, 180], # Deg
    'rz_ff_pha': 0,
    'rz_fm_pha': 45,
    'ff_amp': [32767, 32767, 32767, 32767],
    'fm_amp': [32767, 32767, 32767, 32767],
    'bias': [0, 0, 0, 0],
    'env_start': 0,
    'env_num': 2,
    'cycle_num': 0,
    'codeword_configs': [
        {'wave_hold': 1, 'fcw_index': 0, 'pcw_index': 0, 'code_clr': 1, 'env_index': 0},
        {'wave_hold': 0, 'fcw_index': 0, 'pcw_index': 0, 'code_clr': 0, 'env_index': 1},
        {'wave_hold': 1, 'fcw_index': 0, 'pcw_index': 0, 'code_clr': 1, 'env_index': 2},
        {'wave_hold': 0, 'fcw_index': 0, 'pcw_index': 0, 'code_clr': 0, 'env_index': 3},
        {'wave_hold': 0, 'fcw_index': 0, 'pcw_index': 0, 'code_clr': 1, 'env_index': 4},
        {'wave_hold': 0, 'fcw_index': 0, 'pcw_index': 0, 'code_clr': 1, 'env_index': 5},
        {'wave_hold': 0, 'fcw_index': 0, 'pcw_index': 0, 'code_clr': 1, 'env_index': 6},
        {'wave_hold': 0, 'fcw_index': 0, 'pcw_index': 0, 'code_clr': 1, 'env_index': 7},
        {'wave_hold': 0, 'fcw_index': 0, 'pcw_index': 0, 'code_clr': 1, 'env_index': 8},
    ],
    'send_interval': [100, 150, 200, 250, 300, 350, 400],
    # 'custom_registers': [
    #     (addr_base['DTCNO_BASE']+reg_define['mcu_reg']['FMER'], 0<<31),
    # ],
    'instr_show': False,
}
params.update({
    'FolderName': 'AWG',
    'CaseName': f"AWG_MOD",
})
params.update(
    {'config_file': params['FolderName'] + '/' + params['CaseName'] + '_HEX.txt'})

pm = ParamsManager(params['FolderName'])
pm.save(params, params['CaseName'])

configure_chip_output(mk, **params)
env_config(mk, **params)
instruction_config(mk, mk_instr, **params)

# MUL_16x16
```

3.可配置项和说明

寄存器地址	功能	
0x2000b4	双频点使能	[31]=1'b1
0x2000dc	循环次数	

寄存器地址	功能	
0x2000e0	包络起始索引	
0x2000e4	包络个数	
0x2000e8	码字	
0x2000ec	wait	

4.配置细节

- 1. AWG调制输出，包络与NCO混频
- 2. 可选择双频点输出
- 3. 预加载多条包络，可选择从哪条包络开始发送几条包络，可配置循环发送的次数
- 4. 每个send与wait派对
- 5. 对矩形波、falttop波而言，包络个数为2，第一个wait为保持时间，第二个wait为隔多久发送
- 6. 循环内的wait需大于24
- 7. 最后一个wait需大于37
- 8. 矩形包络和flattop包络在hold模式下长度应大于特定值，因此较短的矩形包络和flattop包络可下载完整包络
- 9. 每个下载的包络的点数必须为4的倍数，4个点的时间是1个750M的时钟周期，也就是4/3ns
- 10. flattop包络的wave_time，如产生24个点，应填22

5.测试Case对应说明

mcu_regfile中，查找表中频率和相位有关的14个寄存器配置

1	00200040
2	00100038
3	08888888
4	11111111
5	19999999
6	22222222
7	00000000
8	00000000
9	15550000
10	1fff0000
11	3fff0000
12	55550000
13	5fff0000
14	6aaa0000
15	7fff0000
16	00001fff

mcu_regfile中，查找表中幅度有关的4个寄存器配置

17	
18	00200078
19	00100010
20	7fff7fff
21	7fff7fff
22	7fff7fff
23	7fff7fff

mcu_regfile中，查找表中偏置有关的4个寄存器配置

25	00200088
26	00100010
27	00000000
28	00000000
29	00000000
30	00000000
31	

调制使能和数据选择寄存器

				32	00300104	
				33	00100008	
				34	00000000	
				35	00000005	
				36		
0x104	MODEN	RW	调制器使能寄存器	AMODENR	[2]	1'b0 ANY 1'b0: 幅度调制有效; 1'b1: 幅度调制无效
				FMODENR	[1]	1'b0 ANY 1'b0: 频率调制有效; 1'b1: 频率调制无效
				BIASENR	[0]	1'b0 ANY 1'b0: 偏置调制无效; 1'b1: 偏置调制有效
				ramp_sel	[3]	1'b0 ANY 1'b1: ramp数据输出; 1'b0: dsp数据
0x108	MODDOTR	RW	数据选择寄存器	dsp_sel	[2]	1'b0 ANY 1'b1: avg数据输出; 1'b0: 施尼校正数据
				awq_sel	[1:0]	2'b0 ANY 2'b0: 包络; 2'b1: 调制; 2'b2: 单音NCO; 2'b3: 双音NCO

双频点使能寄存器

```
37 002000b4
38 00100004
39 00000000
40
```

写包络索引、包络存储memory

```
41 00400000
42 00100024
43 00000004
44 00080004
45 00100008
46 00200008
47 00000000
53 00500000
54 00100390
55 7fff7fff
56 7fff7fff
57 00000000
58 00000000
59 0a16000f
60 7ecf5730
61 7ffe7ffe
62 7fff7ffe
63 7ffe7fff
64 7fff7fff
```

写dtcm数据存储器，case里面一共有17条，但是输入的参数有19个，写code_word_list（7个，写的是9个但是实际只有7个用上了），send_interval（7个），cycle_num(1个)，env_start（1个），env_num（1个）。和python代码对上了。

```
283
284 002000dc
285 00100044
286 00000000
287 00000000
288 00000002
289 40040000
290 00000064
291 00001000
292 00000096
293 40042000
294 000000c8
295 00003000
```

汇编指令：

```
304 00100000
305 00100084
306 001000b7
307 00200137
308 00400193
309 01600213
310 ffff20213
311 0400af83
312 05f12023
313 003080b3
314 00310133
```

200040、200078、200088开始的都是把波形控制字写到dtcm，让MCU把这部分信息搬到mcu_regfile中的查找表。300104是调制使能和300108数据选择寄存器。2000b4也是写dtcm，让MCU配mcu_regfile的双频点使能。400_000和500_000是写包络索引和存储的memory。200_0dc的指令是写dctm寄存器，作用是给mcu参数，以便后面汇编调用。100_000就是汇编指令寄存器。总结一下，mcu要从dtcm中搬控制字、寄存器配置到mcu_regfile中。dctm要写参数，itcm要写指令，二者配合。还要写包络的两块存储。**简单记，1搬2包3参4指令。搬包参指。**搬是mcu把dtcm中200_040到D8搬到对应的mcu_regfile，包是写400000和500000两块存储参是写2000dc后面的dctm存储，指是写把指令写到100000，指令能调用2000dc后面的参数。

6.对应json说明

chip_mod决定了awg_sel是选择了awg

```
{
  "chip_mode": "AWG_MOD",
  "instr_type": "awg_mod",
  "envelope_configs": [
    {
      "envelope_type": "rect_hold",
      "amp": 32767
    },
    {
      "envelope_type": "flattop_hold",
      "amp": 32767,
      "edge_time": 3.5,
      "wave_time": 30
    },
    {
      "envelope_type": "acz",
      "amp": 32767,
      "wave_time": 48
    },
    {
      "envelope_type": "accz",
      "wave_time": 48
    },
    {
      "envelope_type": "cosine",
      "amp": 32767,
      "wave_time": 300
    },
    {
      "envelope_type": "rect",
      "amp": 32767,
      "wave_time": 12
    },
    {
      "envelope_type": "flattop",
      "amp": 32767,
      "edge_time": 2,
      "wave_time": 22
    }
  ],
}
```

```
    "amp_mod_enable": true,
    "freq_mod_enable": true,
    "bias_enable": true,
    "fm_enable": false,
    "fcw": [
      100,
      200,
      300,
      400
    ],
    "mcu_reg_clr": false,
    "pcw": [
      0,
      30,
      45,
      90,
      120,
      135,
      150,
      180
    ],
    "rz_ff_pha": 0,
    "rz_fm_pha": 45,
    "ff_amp": [
      32767,
      32767,
      32767,
      32767
    ],
    "fm_amp": [
      32767,
      32767,
      32767,
      32767
    ],
    "bias": [
      0,
      0,
      0,
      0
    ],
  ],
}
```

```
"env_start": 0,
"env_num": 2,
"cycle_num": 0,
"codeword_configs": [
  {
    "wave_hold": 1,
    "fcw_index": 0,
    "pcw_index": 0,
    "code_clr": 1,
    "env_index": 0
  },
  {
    "wave_hold": 0,
    "fcw_index": 0,
    "pcw_index": 0,
    "code_clr": 0,
    "env_index": 1
  },
  {
    "wave_hold": 1,
    "fcw_index": 0,
    "pcw_index": 0,
    "code_clr": 1,
    "env_index": 2
  },
  {
    "wave_hold": 0,
    "fcw_index": 0,
    "pcw_index": 0,
    "code_clr": 0,
    "env_index": 3
  },
  {
    "wave_hold": 0,
    "fcw_index": 0,
    "pcw_index": 0,
    "code_clr": 1,
    "env_index": 4
  },
  {
    "wave_hold": 0,
    "fcw_index": 0,
    "pcw_index": 0
  }
]
```

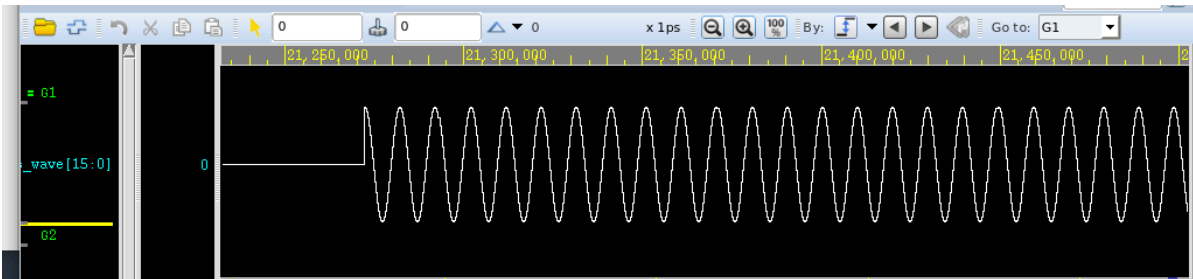
```

    "env_index": 7
  },
  {
    "wave_hold": 0,
    "fcw_index": 0,
    "pcw_index": 0,
    "code_clr": 1,
    "env_index": 8
  }
],
"send_interval": [
  100,
  150,
  200,
  250,
  300,
  350,
  400
],
"instr_show": false,
"FolderName": "AWG",
"CaseName": "AWG_MOD",
"config_file": "AWG/AWG_MOD_HEX.txt"
}

```

3.2.2AWG_NCO

单音NCO输出



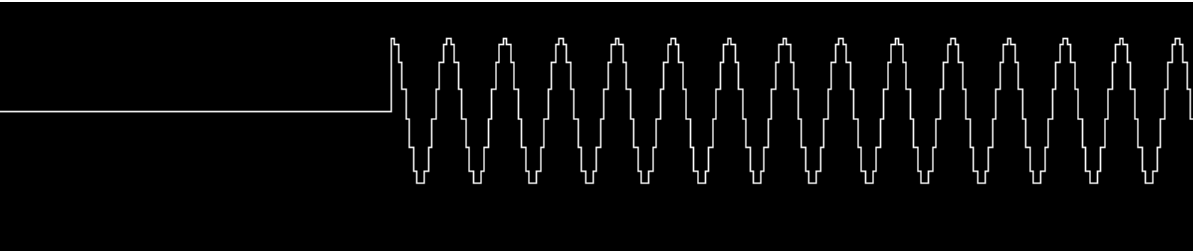
寄存器地址	功能
0x200040	NCO频率
0x200054	NCO相位
0x200074	RZ相位

```
# AWG_NCO
params = {
    'chip_mode': 'AWG_NCO',
    'instr_type': 'awg_nco',
    'fcw':[100, 200, 300, 400], # MHz
    'pcw':[0, 30, 45, 90, 120, 135, 150, 180], # Deg
    'rz_pha': 0,
    'mcu_reg_clr': True,
    'codeword_configs': [
        {'fcw_index': 0, 'pcw_index': 0, 'code_clr': 1, 'env_index': 0},
    ],
    'envelope_configs': [
        {'envelope_type': 'rect_hold', 'amp': 32767},
    ],
    'instr_show': False,
}
param_combinations = []
params.update({
    'FolderName': 'AWG_NCO',
    'CaseName': f"AWG_NCO"
})
params.update(
    {'config_file': params['FolderName'] + '/' + params['CaseName'] + '_HEX.txt'})
pm = ParamsManager(params['FolderName'])
pm.save(params, params['CaseName'])
configure_chip_output(mk, **params)
env_config(mk, **params)
instruction_config(mk, mk_instr, **params)
```

3.2.3AWG_FM_NCO

双音NCO输出（NCO_frext2输出）

这个叫是叫双音NCO输出。但这个并不是我们通常定义的双音NCO,并不是两个正弦信号的叠加，并没有两个频谱点，而是和之前那个普通NCO一模一样，唯一的区别是频率翻倍。所以更准确的叫法是叫做AWG_NCO_frext2。



就是 $signal = A_1 \cdot env(t) \cdot \cos(wt) + A_2 \cdot env(t) \cdot \cos(2wt)$ 中的 $\cos(2wt)$ 部分。代码中mod_nco_ff_fcw是单音NCO的频率控制字，mod_nco_fm_fcw是双音的频率控制字，mod_nco_fm_fcw来源于mod_nco_ff_fcw x 2。

寄存器地址	功能
0x200040	NCO频率
0x200054	NCO相位
0x200074	RZ相位

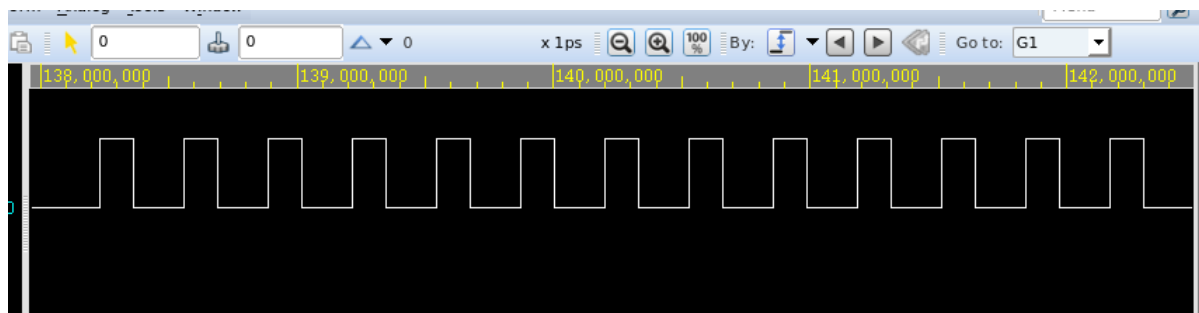
```

1 # AWG_FM_NCO
2 params = {
3     'chip_mode': 'AWG_FM_NCO',
4     'instr_type': 'awg_fm_nco',
5     'fcw':[100, 200, 300, 400], # MHz
6     'pcw':[0, 30, 45, 90, 120, 135, 150, 180], # Deg
7     'rz_pha': 0,
8     'code_clr': [True]*4,
9     'mcu_reg_clr': True,
10    'envelope_configs': [
11        {'envelope_type': 'rect_hold', 'amp': 32767},
12    ],
13    'codeword_configs': [
14        {'fcw_index': 0, 'pcw_index': 0, 'code_clr': 1, 'env_index': 0},
15    ],
16    'instr_show': False,
17 }
18 params.update({
19     'FolderName': 'AWG',
20     'CaseName': f"AWG_FM_NCO"
21 })
22 params.update(
23     {'config_file': params['FolderName'] + '/' + params['CaseName'] + '_HEX.txt'})
24 pm = ParamsManager(params['FolderName'])
25 pm.save(params, params['CaseName'])
26 configure_chip_output(mk, **params)
27 env_config(mk, **params)
28 instruction_config(mk, mk_instr, **params)
29 在 2026.07.13 16:02:01 于 74ms 内执行

```

3.2.4AWG_ENV

AWG_ENV: 包络数据输出



```

1 # AWG_ENV
2 params = {
3     'chip_mode': 'AWG_ENV',
4     'instr_type': 'awg_mod',
5     'inner_sync': True,
6     'envelope_configs': [
7         {'envelope_type': 'rect_hold', 'amp': 32767},
8         {'envelope_type': 'flat_top_hold', 'amp': 32767, 'edge_time': 3.5, 'wave_time': 30},
9         {'envelope_type': 'ac_z', 'amp': 32767, 'wave_time': 12*4},
10        {'envelope_type': 'acc_z', 'wave_time': 12*4},
11        {'envelope_type': 'cosine', 'amp': 32767, 'wave_time': 12*25},
12        {'envelope_type': 'rect', 'amp': 32767, 'wave_time': 12},
13        {'envelope_type': 'flat_top', 'amp': 32767, 'edge_time': 2, 'wave_time': 22},
14    ],
15    'amp_mod_enable': True,
16    'freq_mod_enable': True,
17    'bias_enable': True,
18    'fm_enable': False,
19    'fcw': [100, 200, 300, 400], # MHz
20    'mcu_reg_clr': False,
21    'pcw': [0, 30, 45, 90, 120, 135, 150, 180], # Deg
22    'rz_ff pha': 0,
23    'rz_fm pha': 45,
24    'ff_amp': [32767, 32767, 32767, 32767],
25    'fm_amp': [32767, 32767, 32767, 32767],
26    'bias': [0, 0, 0, 0],
27    'env_start': 0,
28    'env_num': 2,
29    'cycle_num': 0,
30    'codeword_configs': [
31        {'wave_hold': 1, 'fcw_index': 0, 'pcw_index': 0, 'code_clr': 1, 'env_index': 0},
32        {'wave_hold': 0, 'fcw_index': 0, 'pcw_index': 0, 'code_clr': 0, 'env_index': 1},
33        {'wave_hold': 1, 'fcw_index': 0, 'pcw_index': 0, 'code_clr': 1, 'env_index': 2},
34        {'wave_hold': 0, 'fcw_index': 0, 'pcw_index': 0, 'code_clr': 0, 'env_index': 3},
35        {'wave_hold': 0, 'fcw_index': 0, 'pcw_index': 0, 'code_clr': 1, 'env_index': 4},
36        {'wave_hold': 0, 'fcw_index': 0, 'pcw_index': 0, 'code_clr': 1, 'env_index': 5},
37        {'wave_hold': 0, 'fcw_index': 0, 'pcw_index': 0, 'code_clr': 1, 'env_index': 6},
38        {'wave_hold': 0, 'fcw_index': 0, 'pcw_index': 0, 'code_clr': 1, 'env_index': 7},
39        {'wave_hold': 0, 'fcw_index': 0, 'pcw_index': 0, 'code_clr': 1, 'env_index': 8},
40    ],
41    'send_interval': [100, 150, 200, 250, 300, 350, 400],
42    # 'custom_registers': [
43    #     (addr_base['DTCMO_BASE'] + reg_define['mcu_reg']['EMER'], 0 << 31),
44    # ],
45    'instr_show': False,
46 }
47 params.update({
48     'FolderName': 'AWG_ENV',
49     'CaseName': f"AWG_ENV",
50 })
51 params.update(
52     {'config_file': params['FolderName'] + '/' + params['CaseName'] + '_HEX.txt'})
53 pm = ParamsManager(params['FolderName'])
54 pm.save(params, params['CaseName'])
55 configure_chip_output(mk, **params)
56 env_config(mk, **params)
57 instruction_config(mk, mk_instr, **params)
58 在 2026.07.13 16:03:15 于 85ms 内执行
59
60 # AWG_MOD
61 params = {

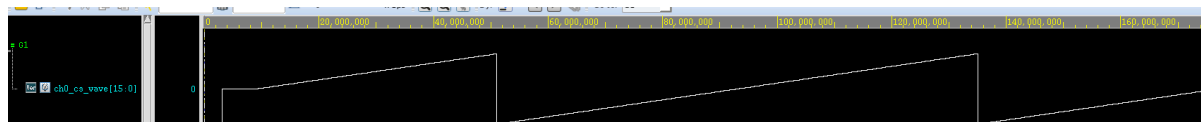
```

3.2.5 RAMP

里面又分好多种模式。

1. RAMP_SPI

RAMP输出台阶波，可控制步进和宽度



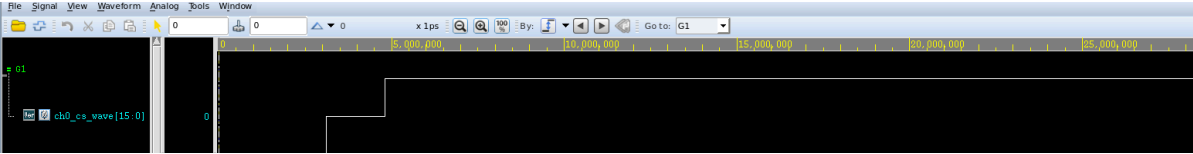
寄存器	功能	备注
0x30011c	步进	[31:16]
0x300120	宽度	

```
# ramp_spi
# TODO: FolderName, Caename, config_file重复n
params = {
    'chip_mode': 'RAMP',
    'ramp_enable_source': 'SPI',
    'height': 16,
    'step_time': 30*10**8, # clk
    'fixed_enable': False,
    'custom_registers': [
        (f"0x{addr_base['DACR0_BASE']+0x30:x}", [0]*4),
        (f"0x{addr_base['DACR0_BASE']+0x78:x}", str(0x7400154)),
    ],
}
param_combinations = []
params.update({
    'FolderName': 'RAMP',
    'CaseName': f"RAMP_SPI_H{params['height']}_W{params['step_time']}",
})
params.update(
    {'config_file': params['FolderName'] + '/' + params['CaseName'] + '_HEX.txt'})
configure_chip_output(mk, **params)
mk.rw_once('w', addr_base['ENVM0_BASE'], 0, params['config_file'])
mk.rw_once('w', addr_base['ITCM0_BASE'], '0x2B', params['config_file'])

pm = ParamsManager(params['FolderName'])
pm.save(params, params['CaseName'])
```

2.RAMP_FIX

RAMP输出固定值，可控制输出固定值的码值

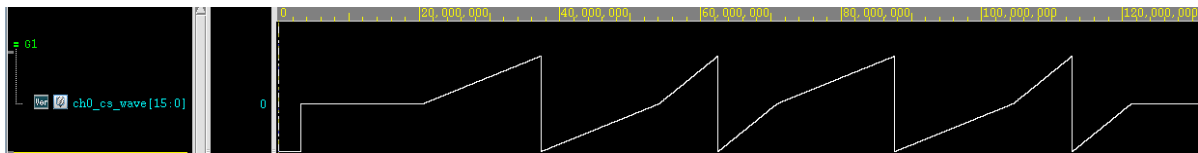


寄存器	功能
0x300118	fixed_value:[31:16] fixed_enable:[15]

```
# ramp_spi_fixed
params = {
    'chip_mode': 'RAMP',
    'ramp_enable_source': 'SPI',
    'fixed_enable': True,
    'fixed_value': 32767,
}
param_combinations = []
params.update({
    'FolderName': 'RAMP',
    'CaseName': f"RAMP_SPI_FIXED"
})
params.update(
    {'config_file': params['FolderName'] + '/' + params['CaseName'] + '_HEX.txt'})
configure_chip_output(mk, **params)
mk.rw_once('w', addr_base['ENVM0_BASE'], 0, params['config_file'])
mk.rw_once('w', addr_base['ITCM0_BASE'], '0x2B', params['config_file'])
```

3.RAMP_MCU

在一个case中调节RAMP的步进和持续时间

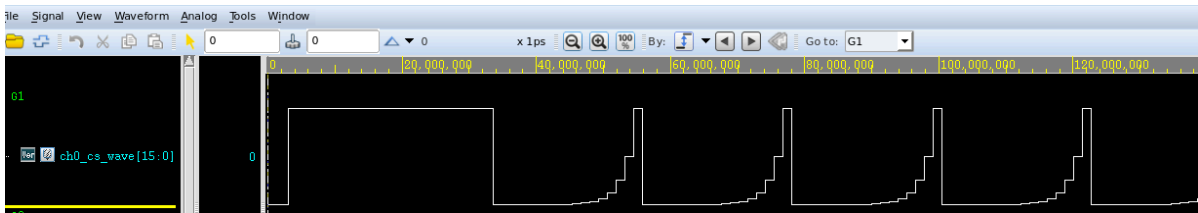


寄存器地址	功能	备注
0x2000dc	fix_enable无效	固定值无需改动
0x2000e0	ramp_enable	固定值无需改动
0x2000e4	参数对的个数	
0x2000e8	循环次数	
0x2000ec	高度	高16位
0x2000f0	长度	
0x2000f4	刷新参数对的时间间隔	

```
# ramp_mcu
params = {
    'chip_mode': 'RAMP',
    'ramp_enable_source': 'MCU',
    'instr_type': 'ramp',
    'instr_show': False,
    'height': [128,256],
    'step_time': [50,50],
    'param_num': 2,
    'ensemble_num': 2,
}
param_combinations = []
params.update({
    'FolderName': 'RAMP',
    'CaseName': f"RAMP_MCU"
})
params.update(
    {'config_file': params['FolderName'] + '/' + params['CaseName'] + '_HEX.txt'})
configure_chip_output(mk, **params)
mk.rw_once('w', addr_base['ENV0_BASE'], 0, params['config_file'])
instruction_config(mk, mk_instr, **params)
```

4.RAMP_MCU_FIX

在一个case中调节固定输出的值和时间。自0x2000e8开始按照固定值、wait时间的顺序依次存放配置参数



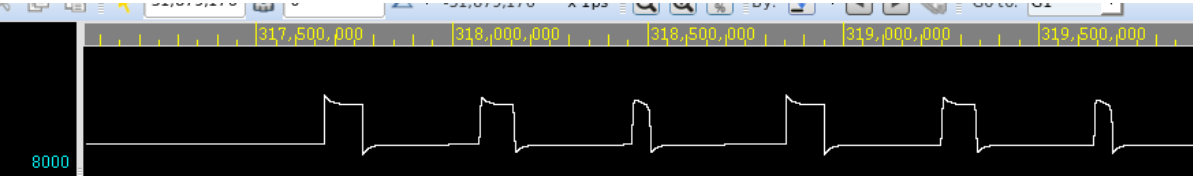
寄存器地址	功能	备注
0x2000dc	fix_enable使能	固定值无需改动
0x2000e0	循环次数	
0x2000e4	参数对的个数	

寄存器地址	功能	备注
0x2000e8	固定值	fixed_value:[31:16] fixed_enable:[15]
0x2000ec	参数更新时间	

```
# ramp_mcu_fixed
params = {
    'chip_mode': 'RAMP',
    'ramp_enable_source': 'MCU',
    'instr_type': 'ramp_fixed',
    'instr_show': False,
    'custom_registers': [
        (f"0x{addr_base['DACR0_BASE']+0x30:x}", [0]*4),
    ],
    'fixed_value': [1<<15] + [1<<15 | 1<<i for i in range(0,15)] + [0],
    'wait_clk': [1000]*17,
    'config_param_num': 17,
    'ensemble_num': 0,
    'envelope_configs': [
        {'envelope_type': 'rect_hold', 'amp': 32767},
    ],
}
param_combinations = []
params.update({
    'FolderName': 'RAMP',
    'CaseName': f"RAMP_MCU_FIXED_TEST"
})
params.update(
    {'config_file': params['FolderName'] + '/' + params['CaseName'] + '_HEX.txt'})
pm = ParamsManager(params['FolderName'])
pm.save(params, params['CaseName'])
configure_chip_output(mk, **params)
instruction_config(mk, mk_instr, **params)
```

3.2.6 DSP

- DSP_NCO：拖尾校正输出，数据源为单音NCO，无实际作用
- DSP_FM_NCO：拖尾校正输出，数据源为双音NCO，无实际作用
- DSP_ENV：拖尾校正输出，数据源为包络，不可扫幅
- DSP_MOD：拖尾校正输出，数据源为调制数据，可以做参数扫描。



拖尾矫正这部分只有DSP_ENV和DSP_MOD有实际的作用。

这部分python没有对应代码，需要用对应的json文件修改。

四.可配参数

该部分可以指导测试工程师、硬件工程师及协同开发人员如何通过编写与修改 `JSON` 配置文件中的参数，利用 Python 工程自动化生成波形芯片的测试 Case 文本。（附件：[EXCEL表格——参数查找表](#)）为了降低使用门槛，本 Python 工程将高可读性的用户参数与复杂的底层硬件细节进行了隔离。比如，在配置 JSON 时，频率直接填常用单位 `MHz`（如 `10`），相位直接填度（如 `45`），开关直接填

`true/false`，代码会自动生成对应的代码。

4.1 JSON实例解析

以下是一个标准的配置JSON 样例（以 `AWG_NCO` 模式为例）。使用者只需按需修改以下字段即可：

JSON

```
{
  "chip_mode": "AWG_NCO",           // 核心字段：芯片工作模式选择（共 8 种，见第三节）
  "instr_type": "awg_nco",          // 指令类型类型（汇编）
  "inner_sync": true,               // 内部同步使能开关
  "custom_registers": [            // 允许用户直接写入特定寄存器
    { "addr": "0x0000034", "values": 5 },
    { "addr": "0x600030", "values": [0, 0, 0, 0] }
  ],
  "fcw": [10, 200, 300, 400],      // 频率控制字列表，直接填数字即可，单位：MHZ
  "pcw": [0, 30, 45, 90, 120, 135], // 相位控制字列表，直接填角度，单位：度（°）
  "rz_pha": 0,                    // 附加相位配置
  "mcu_reg_clr": true,             // 是否MCU使能复位NCO
  "codeword_configs": [           // 码字配置
    { "fcw_index": 0, "pcw_index": 0, "code_clr": 1, "env_index": 0 }
  ],
  "envelope_configs": [           // 包络形态参数配置
    { "envelope_type": "rect_hold", "amp": 32767 }
  ],
  "instr_show": false,             // 是否在 Python 终端打印输出指令详情
  "FolderName": "AWG_NCO",        // 自动化输出的目标文件夹名称
  "CaseName": "tc_ffnco_freq_scan_10MHz", // 生成的测试 Case 唯一名称
  "config_file": "AWG_NCO/tc_ffnco_freq_scan_10MHz_HEX.txt" // 最终生成的case文本路径
}
```

4.2、芯片模式选择

通过在JSON 中配置 `chip_mode` 字段，可以精确切换芯片内部的硬件信号通路。请根据测试需求选择以下 8 种模式：

chip_mode 模式名称	调用python函数与硬件 状态	核心逻辑与参数配置要求
RAMP	执行 <code>_ramp_spi_config</code> 或 MCU 寄存器配置 (状态: RAMP 锯齿波形输出)	配置 300108 数据选择寄存器为 1000。1.首先根据使能源 <code>ramp_enable_source</code> 决定为 SPI 控制还是 MCU 控制（MCU 控制时写 300124 最高位为 1）。2.如果是SPI使能，则：若 <code>fixed_enable == 1</code> ：将 <code>fixed_value</code> 写入 300118 RAMP常数寄存器高 16 位，第 15 位 RAMPFIX 配 1。3.若 <code>fixed_enable == 0</code> ：第 15 位配 0，并自动读取 JSON 中的步进高度 (<code>height</code>) 和步进时间 (<code>step_time</code>) 写入硬件。。
AWG_NCO	执行 <code>_awg_nco_config</code> (状态: 单音 NCO 直通, 无拖尾矫正)	配置 300108 寄存器为单音直通。读取 JSON 传入的 <code>fcw</code> 和 <code>pcw</code> （MHz/度）自动转为码值，连同附加相位、是否复位等信息组合，写入 DTCM (<code>mcu_regfile</code>) 对应位置等待 MCU 搬移。

chip_mode 模式名称	调用python函数与硬件 状态	核心逻辑与参数配置要求
AWG_FM_NCO	执行 _awg_fm_nco_config (状态: 双音 NCO 直通, 无拖尾矫正)	配置 300108 寄存器为双音直通。读取JSON传入的 f _{cw} /p _{cw} 列表转换为硬件码值, 连同附加相位、是否复位等信息组合, 并将 双音频点使能寄存器写 1 , 整体信息打包写入 DTCM 指定位置。
AWG_ENV	执行 _awg_env_config (状 态: 包络输出直通, DSP 输出状态)	配置 300108 寄存器为包络直出。检查三大开关: 1. 若 freq_mod_enable == 1: 执行 nco_reg_config 写入频率相位等信息至 DTCM。2. 若 amp_mod_enable == 1: 获取 ff_amp, fm_amp 拼接写入 DTCM。3. 若 bias_enable == 1: 获取 Bias 列表信息写入 DTCM。记录使能状态保存至 mod_enable (3bit), 再——对应写入 300104 调制使能寄存器 的控制位。
AWG_MOD	执行 _awg_mod_config (状 态: 调制模式直通)	依据三个调制开关 (频率、幅度、Bias) 是否开启, 将相关信息搬移至 DTCM, 等待MCU搬移。记录使能状态参数 mod_enable, 写入 300104 调制使能寄存器 的控制位。这边与AWG_ENV一致。此外, 若 fm_enable == 1 则在 DTCM 双音频点使能位写 1。
DSP_NCO	直接配置目标寄存器 (状 态: 单音 NCO + 经过拖 尾矫正)	配置 300108 寄存器, 强行让单音信号通过硬件的 拖尾矫正电路 , 最后从 DSP 通路输出。(该模式无用)
DSP_FM_NCO	直接配置目标寄存器 (状 态: 双音 NCO + 经过拖 尾矫正)	配置 300108 寄存器, 强行让双音频信号经过硬件 拖尾矫正电路 , 由 DSP 通路输出。(该模式无用)
DSP_ENV 及 DSP_MOD	执行 _dsp_config (状 态: 经过拖尾矫正的包 络/调制输出)	核心算法触发 : 首先自动执行 _calculate_tc_coefficients, 计算出 硬件拖尾矫正系数的正数值 并写入对应的 tc_reg 寄存器。然后将根据调制使能3大开关 (freq_mod_enable``amp_mod_enable``bias_enable) 数据写入 DTCM, 并将条件生成的 3 位 mod_enable 状态值映射写入 300104 调制使能寄存器 。最后根据是 ENV 还是 MOD 决定配置 300108 寄存器为包络输出或调制输出。

在排查底层生成的 Case 文本时, 可参考以下关键寄存器:

- **300108 寄存器 (数据选择与通路控制)**: 全芯片的总闸通路。例如 RAMP 模式时固定为 1000。是否NCO直通或经过拖尾矫正的 DSP 模式会根据 chip_mode 自动生成对应的控制码, 写入该寄存器。
- **300104 寄存器 (调制使能控制)**: 硬件调制模块的开关总线。由 amp_mod_enable、freq_mod_enable、bias_enable 这三个布尔值的状态组合成一个 3 bit 二进制数按位写入。决定要不要频率, 幅度, 和偏置调制。
- **DTCM (mcu_regfile)**: 芯片内部的数据暂存区。用户的数组参数 (如大批量的频率控制字列表) 会被格式化后写入 DTCM, 芯片内的 MCU 运行后会从该区域高速搬移并刷新到MCU外设寄存器 (mcu_regfile)。MCU_regfile长下面这样。一般DTCM都是按照mcu_regfile偏移地址放置数据, 方便RISV-CPU汇编指令搬移操作

	A	B	C
1	寄存器名称	地址(偏移)	功能描述
2	CWFR0	0x40	查找波形频率1
3	CWFR1	0x44	查找波形频率2
4	CWFR2	0x48	查找波形频率3
5	CWFR3	0x4C	查找波形频率4
6	CWPRR	0x50	最高位写1, NCO复位
7	GAPR0	0x54	查找表波形相位0
8	GAPR1	0x58	查找表波形相位1
9	GAPR2	0x5C	查找表波形相位2
10	GAPR3	0x60	查找表波形相位3
11	GAPR4	0x64	查找表波形相位4
12	GAPR5	0x68	查找表波形相位5
13	GAPR6	0x6C	查找表波形相位6
14	GAPR7	0x70	查找表波形相位7
15	LCPR	0x74	附加相位
16	AMPR0	0x78	查找表波形幅度0 32bit ,高16 A1 低16是A2
17	AMPR1	0x7C	查找表波形幅度1
18	AMPR2	0x80	查找表波形幅度2
19	AMPR3	0x84	查找表波形幅度3
20	BIASR0	0x88	查找表波形偏置0
21	BIASR1	0x8C	查找表波形偏置1
22	BIASR2	0x90	查找表波形偏置2
23	BIASR3	0x94	查找表波形偏置3
24	RTMR	0x98	mcu运行时间
25	ICNTR	0x9C	指令数量
26	FSIR	0xA0	反馈
27	DCBVR	0xA4	偏置大小
28	FMER	0xB4	双频点使能
29	MCU_RAMPFIXR	0xB8	Ramp固定值寄存器 - 斜坡输出固定值模式数据
30	MCU_RAMPSR	0xBC	Ramp步进寄存器
31	MCU_RAMPIFSR	0xC0	Ramp宽度寄存器
32	MCU_RAMPENR	0xC4	Ramp使能寄存器
33	PRNGSDR	0xC8	z没用
34	PRNGRESR	0xCC	z没用
35	MULTRO	0xD0	乘法器结果寄存器0
36	MULTR1	0xD4	乘法器结果寄存器1
37	DTFR	0xD8	z没用

4.3参数配置速查表

1.通用必填参数

无论选择哪种模式，以下基础字段均需要在 JSON 文件中提供：

- `chip_mode`: 核心参数，芯片的工作模式选择（如 `RAMP`、`AWG_NCO`、`AWG_MOD` 等），决定了底层的通路流向。
- `FolderName`: 输出的目标文件夹名称。
- `CaseName`: 生成的测试 Case 唯一标识名称。
- `config_file`: 最终生成的十六进制（HEX）文本 Case 文件的存储路径。
- `instr_show`: 布尔值（`true/false`），指示运行 Python 脚本时是否在终端打印输出具体的配置指令详情。
- `custom_registers`: 自定义寄存器参数。支持直接传入地址和数值，允许用户在脚本处理主模式逻辑前或后，强行直接写入特定的硬件寄存器。

4.3.1 斜坡波模式 (RAMP)

当 `chip_mode` 设置为 `"RAMP"` 时，配置主要取决于使能触发源是 SPI 还是 MCU 控制。

核心配置参数	参数类型/示例	参数详细含义及作用
ramp_enable_source	字符串 ("SPI" 或 "MCU")	使能源选择 : 指示 RAMP 模块是由外部 SPI 指令控制触发还是由芯片内置 MCU 触发使能。不管哪一个触发，都是控制RAMP固定值步进宽度使能这四个寄存器，区别是一个在ctrl中一个在mcu_regfile中。
height	整数 (如 100)	步进高度 : 当 fixed_enable 为 false 时生效，控制 RAMP 信号每步跳变的高度幅值。
step_time	整数 (如 200)	步进时间 : 当 fixed_enable 为 false 时生效，控制 RAMP 信号在每个阶梯停留的时钟周期。
fixed_enable	布尔值 (true / false)	固定值模式使能 : 如果为 true，则 RAMP 通路输出为固定常数；如果为 false，则输出动态步进的 RAMP 波形。
fixed_value	数组 (如 [0, 16384, 32768])	固定常数值列表 : 仅当使能源为 MCU 且固定模式开启时生效，提供一组给常数寄存器的高位配置值。
wait_clk	数组 (如 [1000, 1000])	等待时钟周期 : MCU参数。在 MCU 配合固定值模式时使用，控制每一组配置值写入后的等待时钟时间。
config_param_num	整数 (如 3)	单轮波形点数 : MCU参数。指示当前测试序列一轮包含多少个固定值配置点（通常与 fixed_value 数组长度一致）。MCU 内部用它来控制单轮循环的次数。
ensemble_num	整数	序列重复轮数 : MCU参数。指示这套固定值配置序列整体需要重复循环发射多少轮。例如配 5 代表整套测试扫频波形将重复运行 5 次。

4.3.2 AWG的NCO直通模式 (AWG_NCO / AWG_FM_NCO)

这两个模式用于输出基础的单音/双音射频波形，不包含复杂的包络调制，且不经拖尾矫正电路。

核心配置参数	参数类型/示例	参数详细含义及作用
instr_type	字符串 ("awg_nco" / "awg_fm_nco")	指令类型 : 指示 Python 脚本中 instruction_config函数调用对应的汇编模板,生成汇编指令，从而进行单音/双音模式中MCU把DCTM数据搬移到mcu_regfile中的操作。
inner_sync	布尔值 (true / false)	内部同步开关 : 控制芯片内部的同步触发信号是否开启。
fcw	数组 (如 [100, 200, 300])	频率控制字列表 : 填写查找表需要的所有频率控制字， 单位直接为 MHz ，工程会自动转为硬件码值。

核心配置参数	参数类型/示例	参数详细含义及作用
<code>pcw</code>	数组 (如 <code>[0, 30, 45, 90]</code>)	相位控制字列表 : 填写查找表的相位控制字, 单位直接为度 (°) 。
<code>rz_pha</code>	整数 (如 <code>0</code>)	附加相位 : 指示芯片基准参考相位, 实际相位=查找表中的相位叠加这个附加相位。
<code>mcu_reg_clr</code>	布尔值 (<code>true</code> / <code>false</code>)	mcu_regfile里面0x50NCO复位寄存器 : MCU参数。高位写1, 写入DTCM, 再由mcu搬到对应mcu_regfile中0x50位置, 可以实现NCO复位。效果和 <code>code_clr</code> 一样。
<code>code_clr</code>	数组 (如 <code>[true, true]</code>)	调制NCO清零位 : 映射到了芯片控制字 (Code Word) 的 第 18 位 (Bit 18) , 可以让NCO复位。效果和 <code>mcu_reg_clr</code> 一样。
<code>envelope_configs</code>	结构体数组	基础包络定义 : 通常在此类模式下仅填入默认的静态直通包络 (如 <code>"envelope_type": "rect_hold"</code>) 来保持信号恒定输出。
<code>codeword_configs</code>	结构体数组	码字列表 : MCU参数。MCU使用码字灵活控制和切换波形。一个码字对应一个波形, 可以灵活切换。

4.3.3 AWG 直通包络与调制模式 (`AWG_ENV` / `AWG_MOD`)

用于输出复杂的包络信号或调制信号。通过灵活的开关控制幅度、频率与偏置, 可实现跳频扫描、相位扫描等动态测试 Case。

核心配置参数	参数类型/示例	参数详细含义及作用
<code>amp_mod_enable</code>	布尔值 (<code>true</code> / <code>false</code>)	幅度调制使能 : 为 <code>true</code> 时, 开启幅度控制, Python 会将幅度相关数组搬移至 DTCM 并控制 <code>300104</code> 寄存器。
<code>freq_mod_enable</code>	布尔值 (<code>true</code> / <code>false</code>)	频率调制使能 : 为 <code>true</code> 时, 开启频率调制, 也就是打开NCO,Python会自动获取相关参数写到 DTCM中, 控制 <code>300104</code> 寄存器。
<code>bias_enable</code>	布尔值 (<code>true</code> / <code>false</code>)	偏置调制使能 : 为 <code>true</code> 时, 开启频率调制, 获取直流偏置 (Bias) 参数写到DTCM中。
<code>fm_enable</code>	布尔值 (<code>true</code> / <code>false</code>)	双音频调制使能 : 若为 <code>true</code> , 底层将在 DTCM 中对应mcu_regfile的双音频点使能寄存器上写入 1。
<code>envelope_configs</code>	结构体数组	包络形态形态库 : 可以配置多种包络类型 (如 <code>rect</code> 、 <code>flattop</code> 、 <code>cosine</code> 等), 包含每种包络的幅值 <code>amp</code> 、脉宽 <code>wave_time</code> 以及上升/下降沿时间 <code>edge_time</code> 。

核心配置参数	参数类型/示例	参数详细含义及作用
<code>ff_amp / fm_amp</code>	数组 (如 [32767, 32767])	单音NCO幅度 / 双音NCO幅度列表 : 提供给幅度调制器使用, 幅度数组控制字, 低16位是基频NCO的幅度, 高16位给倍频的NCO的幅度。
<code>bias</code>	数组 (如 [0, 0, 0, 0])	偏置参数列表 : 对应偏置通道的一组偏置控制字。
<code>rz_ff pha / rz_fm pha</code>	整数 (如 0, 45)	附加相位寄存器 : 低16位是基频NCO的附加相位, 高16位给倍频的NCO的附加相位。
<code>env_start</code>	整数 (如 0, 4)	包络起始索引 : MCU参数。指示 MCU 在开始发送脉冲串时, 从 <code>codeword_configs</code> (或包络库) 的第几个索引位置开始读取配置并进行发射。
<code>env_num</code>	整数	单轮发射的包络个数 : MCU参数。指示在内层循环中, 一轮需要连续发射多少个脉冲波形 。例如填 4 就会顺序把 <code>env_start</code> 往后的 4 个包络打出去。
<code>cycle_num</code>	整数 (如 0)	大循环次数 (轮数) : MCU参数。指示这一整套脉冲波形串 (包含 <code>env_num</code> 个包络的波形) 整体需要 重复循环打多少大轮 。通常填 0 可能代表无限持续循环发射。
<code>codeword_configs</code>	结构体数组	码字 : MCU参数。可以放多个码字数组, 一个码字对应一个波形, 可以实现扫频, 扫相, 扫幅序列。
<code>send_interval</code>	数组 (如 [175, 175])	发送时间间隔 : MCU参数。定义了序列中 每一个脉冲发射完毕后, 硬件需要等待多少个时钟周期 才允许发射下一个脉冲。内层循环里, 每一个包络 (脉冲) 发送完毕后的等待间隔。

4.3.4 DSP 矫正包络模式 (DSP_ENV / DSP_MOD)

此模式专门用于高精度、高质量波形生成。除了具备上述 `AWG_MOD / AWG_ENV` 的所有包络和动态扫频控制参数外, 它还会**强行让信号通过硬件的拖尾矫正电路**。

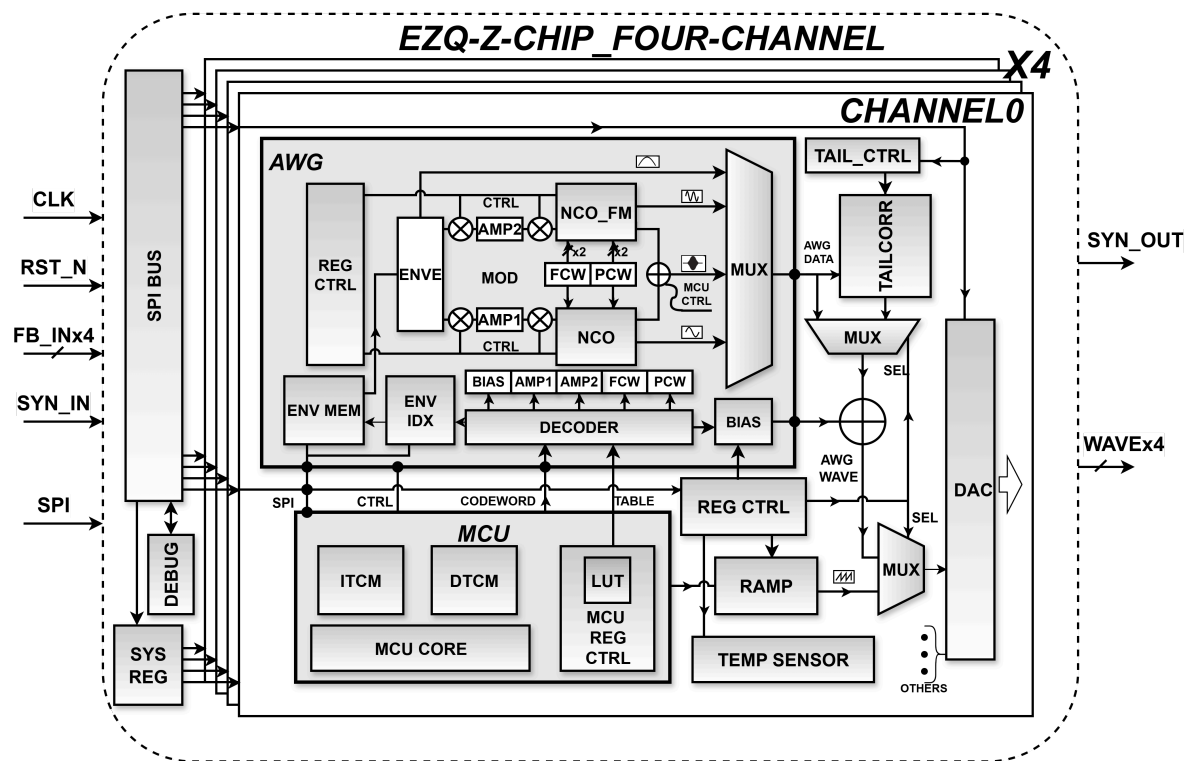
模式特有配置参数	参数类型/示例	参数详细含义及作用
<code>tc_coef_set</code>	字符串 (如 "coef1")	拖尾矫正系数选择 : 指示 Python 工程应当使用哪一组计算公式或约束, 调用底层的 <code>_calculate_tc_coefficients</code> 函数, 动态算出 4 个矫正系数的正数值改写到芯片的 <code>tc_reg</code> 中。
(其它通用参数)	同 <code>AWG_MOD</code>	包含 <code>amp_mod_enable</code> 、 <code>freq_mod_enable</code> 、 <code>bias</code> 、 <code>codeword_configs</code> 等所有动态包络与调制控制字段, 逻辑完全兼容, 只是硬件走 DSP 矫正通路。

五、测试项目

该部分见附件: [Z芯片测试大纲.pdf](#)

四通道Z芯片（待完善）

1.架构图



2.RTL代码差异

1.RTL差异

反馈多了3个通道。

```
//Feedback signal
,input [1:0] ch0_feedback // Ch0 Feedback signals from the readout chip
`ifdef CHANNEL_IS_FOUR
,input [1:0] ch1_feedback // Ch1 Feedback signals from the readout chip
,input [1:0] ch2_feedback // Ch2 Feedback signals from the readout chip
,input [1:0] ch3_feedback // Ch3 Feedback signals from the readout chip
`endif

`ifdef CHANNEL_IS_FOUR
wire [7:0] fb_st_in = {ch3_feedback, ch2_feedback, ch1_feedback, ch0_feedback};
`else
wire [7:0] fb_st_in = {2'b00, 2'b00, 2'b00, ch0_feedback};
`endif
```

DAC配置端口和衰减器配置多了3个

```
`ifdef CHANNEL_IS_FOUR
//-----ch1 DAC cfg pin-----
,output ch1_dac_PrbsEn
,output [15:0] ch1_dac_Rload
,output [0:0] ch1_dac_Mode
```

系统寄存器多了3组


```

471 system_regfile #
472 .CHIPCODE ( 32'h005A5343 )
473 .MFDATE ( 32'h20260431 )
474 ) U_system_regfile
475 .clk ( clk )
476 .rst_n ( pll_rstn_o )
477 .wdata ( slv[0].din )
478 .wren ( slv[0].wren )
479 .rwaddr ( slv[0].addr[15:0] )
480 .rden ( slv[0].rden )
481 .rddata ( slv[0].dout )
482 .dbg_enable ( dbg_enable )
483 .dbg_ch_sel ( dbg_ch_sel )
484 .dbg_upd ( dbg_upd )
485 .ch0_proc_cft ( ch0_proc_cft )
486 .ch0_ldst_addr_unalgn ( ch0_ldst_addr_unalgn )
487 .ch0_dec_err ( ch0_dec_err )
488 .ch0_exit_irq ( ch0_exit_irq )
489 .ch0_over_flag ( ch0_over_flag )
490 .ch0_mcu_nexit_irq ( ch0_mcu_nexit_irq )
491 .ch1_proc_cft ( ch1_proc_cft )

```

整个channel_top例化多了3个

```

740 `ifdef CHANNEL_IS_FOUR
741 //-----
742 // channel 1 instantiation start
743 //-----
744
745 z_channel_top # (
746 .CHLNUM ( 1 )
747 ) U1_channel_top
748 .clk ( clk )
749 .rst_n ( ch1_rstn_o )
750 .dac_rstn ( ch1_dac_rstn_o )
751 .sync_int ( sync_pulse )
752 .dec_o_ilegl ( ch1_dec_err )
753 .agu_o_addr_unalgn ( ch1_ldst_addr_unalgn )
754 .awg_proc_cft ( ch1_proc_cft )

```

同时SPI中的mst分支变多了。原先SPI分2路，一个控制PLL控制寄存器，一个为mst，mst总线挂载多路slv，其中slv0是系统寄存器，slv1是指令存储器，slv2是数据存储器，MCU控制，slv3是控制寄存器，slv4是包络id寄存器，slv5是包络存储寄存器，slv6是dac寄存器，均属于通道1。现在slv7-12是属于channel1。slv13-18是属于channel2。slv19-24是属于channel3。slv25是debug模块。

Debug模块多了3组

```

1070
1071 `ifdef CHANNEL_IS_FOUR
1072 //channel 1 xy dsp data
1073 assign ch1_dsp_vld = ch1_z_dsp_dout_vld ;
1074 assign ch1_dsp_data[0 ] = ch1_z_dsp_dout0 ;
1075 assign ch1_dsp_data[1 ] = ch1_z_dsp_dout1 ;
1076 assign ch1_dsp_data[2 ] = ch1_z_dsp_dout2 ;
1077 assign ch1_dsp_data[3 ] = ch1_z_dsp_dout3 ;
1078 //channel 2 xy dsp data
1079 assign ch2_dsp_vld = ch2_z_dsp_dout_vld ;
1080 assign ch2_dsp_data[0 ] = ch2_z_dsp_dout0 ;
1081 assign ch2_dsp_data[1 ] = ch2_z_dsp_dout1 ;
1110
1111 debug_top U_debug_top
1112 //system port
1113 .clk ( clk )
1114 .rst_n ( pll_rstn_o )
1115 .debug_enable ( dbg_enable )
1116 .debug_ch_sel ( dbg_ch_sel )
1117 .debug_update ( dbg_upd )
1118 .ch0_dsp_data ( ch0_dsp_data )
1119 .ch0_dsp_vld ( ch0_dsp_vld )
1120 .ch1_dsp_data ( ch1_dsp_data )
1121 .ch1_dsp_vld ( ch1_dsp_vld )
1122 .ch2_dsp_data ( ch2_dsp_data )
1123 .ch2_dsp_vld ( ch2_dsp_vld )
1124 .ch3_dsp_data ( ch3_dsp_data )
1125 .ch3_dsp_vld ( ch3_dsp_vld )
1126 .dbg_sram_in ( slv[25].slave )
1127 );

```

IDS表差异

3.汇编撰写

异常

RAMP必须先配好，再配RAMP使能。不然会错有先后顺序，使能最后配，不然台阶高会乘以4倍，台阶宽度不是宽度而是总台阶的时钟个数。