

# GARC VERSION 3 Verilog Code

(D. Sheppard, May 2003)

```
*****  
// Inclusion of GARC module partitions  
  
'include "garc_cmd_proc_v3.v" // module M1, the GARC command processor  
'include "event_data_v2.v" // module M2, the GARC event data processor  
'include "garc_pha_logic_v2.v" // module M3, ADC controller and zero-suppression circuitry  
'include "disc_logic_v2.v" // module M4, VETO and HitMap logic  
'include "lookatme_logic_v3.v" // module M5, the Look-At-Me logic  
'include "reset_logic_v2.v" // module M6, the logic for the global reset  
  
*****  
// GARC module declaration  
  
module garc_module (
```

/\* GARC Core Global Signals \*/

```
    ACD_CLK , // global 20 MHz clock used by core (left rail)
    ACD_CLK_A , // clock from primary AEM
    ACD_CLK_B , // clock from secondary AEM
    N_G_RESET , // active low global reset used, input to core after buffering (right
rail)
    NRESET_OUT , // active low global reset output from core for buffering
    PWR_ON_RST_IN , // power-on-reset input from the FREE circuit
    CLK_OUT , // clock out to external clock buffer
```

/\* AEM command & data ports \*/

```
    ACD_NSCMD_A , // primary AEM serial command data input
    ACD_NSCMD_B , // secondary AEM serial command data input
    ACD_NRESET_A , // primary AEM ACD reset
    ACD_NRESET_B , // secondary AEM ACD reset
    ACD_NSADATA , // ACD Data return to AEM
```

/\* AEM VETO outputs \*/

```
    ACD_NVETO_00 , // VETO to AEM, channel 00
    ACD_NVETO_01 , // VETO to AEM, channel 01
    ACD_NVETO_02 , // VETO to AEM, channel 02
    ACD_NVETO_03 , // VETO to AEM, channel 03
    ACD_NVETO_04 , // VETO to AEM, channel 04
    ACD_NVETO_05 , // VETO to AEM, channel 05
    ACD_NVETO_06 , // VETO to AEM, channel 06
    ACD_NVETO_07 , // VETO to AEM, channel 07
    ACD_NVETO_08 , // VETO to AEM, channel 08
    ACD_NVETO_09 , // VETO to AEM, channel 09
    ACD_NVETO_10 , // VETO to AEM, channel 10
    ACD_NVETO_11 , // VETO to AEM, channel 11
    ACD_NVETO_12 , // VETO to AEM, channel 12
    ACD_NVETO_13 , // VETO to AEM, channel 13
    ACD_NVETO_14 , // VETO to AEM, channel 14
    ACD_NVETO_15 , // VETO to AEM, channel 15
    ACD_NVETO_16 , // VETO to AEM, channel 16
    ACD_NVETO_17 , // VETO to AEM, channel 17
    ACD_NCNO , // CNO discriminator output to AEM
```

/\* GAFE Discriminator inputs \*/

```
    DISC_00_IN , // Discriminator input from GAFE ch. 00
    DISC_01_IN , // Discriminator input from GAFE ch. 01
    DISC_02_IN , // Discriminator input from GAFE ch. 02
    DISC_03_IN , // Discriminator input from GAFE ch. 03
    DISC_04_IN , // Discriminator input from GAFE ch. 04
    DISC_05_IN , // Discriminator input from GAFE ch. 05
    DISC_06_IN , // Discriminator input from GAFE ch. 06
    DISC_07_IN , // Discriminator input from GAFE ch. 07
    DISC_08_IN , // Discriminator input from GAFE ch. 08
    DISC_09_IN , // Discriminator input from GAFE ch. 09
    DISC_10_IN , // Discriminator input from GAFE ch. 10
    DISC_11_IN , // Discriminator input from GAFE ch. 11
```

```

DISC_12_IN , // Discriminator input from GAFE ch. 12
DISC_13_IN , // Discriminator input from GAFE ch. 13
DISC_14_IN , // Discriminator input from GAFE ch. 14
DISC_15_IN , // Discriminator input from GAFE ch. 15
DISC_16_IN , // Discriminator input from GAFE ch. 16
DISC_17_IN , // Discriminator input from GAFE ch. 17
HLD_OR_IN , // Discriminator input from CNO

/* GAFE Configuration and Analog control functions */
GAFE_STROBE , // GAFE TCI CALIB strobe
GAFE_DAT , // GAFE serial command data
GAFE_CLK , // GAFE command clock
GAFE_RST_OUT , // GAFE reset wire
GAFE_RET_DAT , // GAFE return data input
GAFE_HOLD , // GAFE analog pulse HOLD

CHID_00_IN , // GAFE range bit (0 = lo, 1 = hi) for GAFE ch. 00
CHID_01_IN , // GAFE range bit (0 = lo, 1 = hi) for GAFE ch. 01
CHID_02_IN , // GAFE range bit (0 = lo, 1 = hi) for GAFE ch. 02
CHID_03_IN , // GAFE range bit (0 = lo, 1 = hi) for GAFE ch. 03
CHID_04_IN , // GAFE range bit (0 = lo, 1 = hi) for GAFE ch. 04
CHID_05_IN , // GAFE range bit (0 = lo, 1 = hi) for GAFE ch. 05
CHID_06_IN , // GAFE range bit (0 = lo, 1 = hi) for GAFE ch. 06
CHID_07_IN , // GAFE range bit (0 = lo, 1 = hi) for GAFE ch. 07
CHID_08_IN , // GAFE range bit (0 = lo, 1 = hi) for GAFE ch. 08
CHID_09_IN , // GAFE range bit (0 = lo, 1 = hi) for GAFE ch. 09
CHID_10_IN , // GAFE range bit (0 = lo, 1 = hi) for GAFE ch. 10
CHID_11_IN , // GAFE range bit (0 = lo, 1 = hi) for GAFE ch. 11
CHID_12_IN , // GAFE range bit (0 = lo, 1 = hi) for GAFE ch. 12
CHID_13_IN , // GAFE range bit (0 = lo, 1 = hi) for GAFE ch. 13
CHID_14_IN , // GAFE range bit (0 = lo, 1 = hi) for GAFE ch. 14
CHID_15_IN , // GAFE range bit (0 = lo, 1 = hi) for GAFE ch. 15
CHID_16_IN , // GAFE range bit (0 = lo, 1 = hi) for GAFE ch. 16
CHID_17_IN , // GAFE range bit (0 = lo, 1 = hi) for GAFE ch. 17

/* GAFE ADC Data Interface Signals */
NADC_CS_OUT , // chip select to ADCs
ADC_CLK_OUT , // serial clock to ADCs
PHAD_00_IN , // PHA serial data input for ADC on GAFE ch. 00
PHAD_01_IN , // PHA serial data input for ADC on GAFE ch. 01
PHAD_02_IN , // PHA serial data input for ADC on GAFE ch. 02
PHAD_03_IN , // PHA serial data input for ADC on GAFE ch. 03
PHAD_04_IN , // PHA serial data input for ADC on GAFE ch. 04
PHAD_05_IN , // PHA serial data input for ADC on GAFE ch. 05
PHAD_06_IN , // PHA serial data input for ADC on GAFE ch. 06
PHAD_07_IN , // PHA serial data input for ADC on GAFE ch. 07
PHAD_08_IN , // PHA serial data input for ADC on GAFE ch. 08
PHAD_09_IN , // PHA serial data input for ADC on GAFE ch. 09
PHAD_10_IN , // PHA serial data input for ADC on GAFE ch. 10
PHAD_11_IN , // PHA serial data input for ADC on GAFE ch. 11
PHAD_12_IN , // PHA serial data input for ADC on GAFE ch. 12
PHAD_13_IN , // PHA serial data input for ADC on GAFE ch. 13
PHAD_14_IN , // PHA serial data input for ADC on GAFE ch. 14
PHAD_15_IN , // PHA serial data input for ADC on GAFE ch. 15
PHAD_16_IN , // PHA serial data input for ADC on GAFE ch. 16
PHAD_17_IN , // PHA serial data input for ADC on GAFE ch. 17

/* ACD High Voltage Bias Supple Enable/Disable Control Bits */
HV_ENABLE_1OUT , // HVBS A active hi enable bit
HV_ENABLE_2OUT , // HVBS B active hi enable bit

/* MAX5121 Digital-to-Analog Converter Control Signals */
NDAC_CLR_OUT , // MAX5121 active low clear (HVBS DAC)
DAC_DAT_RET , // MAX5121 serial data in (HVBS DAC)
DAC_CLK_OUT , // MAX5121 serial clock (HVBS DAC)
NDAC_CS_OUT , // MAX5121 data envelope (HVBS DAC)
DAC_DATA_OUT , // MAX5121 serial data out (HVBS DAC)

/* LVDS Driver Control Bits */
VETO_ENA_OUT , // LVDS Veto A enable
VETO_ENB_OUT , // LVDS Veto B enable

/* GARC Test Points */

```

```

        HITMAP_TEST_OUT      , // test point for oscilloscope verification
        FREE_ID_IN           , // FREE Board Identification Data to be shifted in
        TRIGGER_OUT          // GARC trigger flag
    ) ;

//****************************************************************************
/*
-----+
-----+
| | GLAST ACD Readout Controller ASIC (GARC)
| | Top Hierarchical Module
| |
| | Project : Gamma-Large Area Space Telescope (GLAST)
| | Anti-Coincidence Detector (ACD)
| | GLAST ACD Readout Controller (GARC) ASIC
| |
| | Written      : D. Sheppard
| | Module       : garc_module_v2.v
| | Original     : 12-05-01
| | Updated      : 03-12-03
| | Version      : 3.0
| |
|-----+
| | VERILOG-XL 3.10.p001
|-----+
|
| This software is property of the National Aeronautics and Space
| Administration. Unauthorized use or duplication of this
| software is strictly prohibited. Authorized users are subject
| to the following restrictions:
|
| 1. Neither the author, their corporation, nor NASA is
|    responsible for any consequences of the use of this software.
| 2. The origin of this software must not be misrepresented either
|    by explicit claim or by omission.
| 3. Altered versions of this software must be plainly marked
|    as such.
| 4. This notice may not be removed or altered.
|
-----+
***** Version History & Changes Made

```

#### Version 3.0:

03-12-03: Update top level module inclusions for GARC V3 (garc\_cmd\_proc and look\_at\_me modules). Power on reset now goes to look-at-me logic.

#### Version 2.0:

09-16-02: Update top level module inclusions for GARC V2.  
 Bring core clock into reset\_logic module.  
 Add shift\_edge\_ctrl parameter to cmd\_proc port list.

#### Version 1.0:

03-18-02: Add test pin multiplexer and FREE board id functions.  
 03-15-02: Add ADC acquisition timer. Add data\_busy feature.  
 03-14-02: Update garc\_module port names to match latest spreadsheet of names  
 03-13-02: Change global reset polarities on modules to active low  
 03-11-02: Update port names to make pad frame list  
 03-08-02: Add Look-At-Me logic. Change reset circuitry to accomodate the LAM logic.  
 03-07-02: Add HitMap\_Test test point  
 03-06-02: Update reset circuitry, including input/output ports of core.

Removed test multiplexer. Updated signal names to match the proper names on the pad frame. Change "A" and "B" to "P" and "S" (primary and secondary).

02-19-02: Fix Veto\_AEM outputs to be negative true logic as per ICD

02-18-02: Modified to allow for the external clock ANDing and buffering done in the SLAC pad frame. Added comments.

02-15-02: Added port for HitMap\_Deadtime

02-11-02: Added ports in top level to remove bus structure and bring out only scalars so that names could be preserved through Exemplar

02-07-02: added "live" bit for command processor timing

Version 1.0: change comments in text of previous version

```
*****  
*/
```

// GARC IO Ports

```
input      ACD_CLK          ; // the buffered, ANDed clock from the pad frame  
input      ACD_CLK_A         ; // ACD clock from primary AEM  
input      ACD_CLK_B         ; // ACD clock from secondary AEM  
input      ACD_NSCMD_A        ; // prime AEM serial command data input  
input      ACD_NSCMD_B        ; // secondary AEM serial command data input  
input      ACD_NRESET_A       ; // prime AEM ACD reset  
input      ACD_NRESET_B       ; // secondary AEM ACD reset  
output     ACD_NSADATA       ; // ACD data return to AEM  
  
input      PWR_ON_RST_IN     ; // GARC power-on-reset from FREE circuit  
  
output     ACD_NVETO_00       ; // VETO to AEM, channel 00  
output     ACD_NVETO_01       ; // VETO to AEM, channel 01  
output     ACD_NVETO_02       ; // VETO to AEM, channel 02  
output     ACD_NVETO_03       ; // VETO to AEM, channel 03  
output     ACD_NVETO_04       ; // VETO to AEM, channel 04  
output     ACD_NVETO_05       ; // VETO to AEM, channel 05  
output     ACD_NVETO_06       ; // VETO to AEM, channel 06  
output     ACD_NVETO_07       ; // VETO to AEM, channel 07  
output     ACD_NVETO_08       ; // VETO to AEM, channel 08  
output     ACD_NVETO_09       ; // VETO to AEM, channel 09  
output     ACD_NVETO_10       ; // VETO to AEM, channel 10  
output     ACD_NVETO_11       ; // VETO to AEM, channel 11  
output     ACD_NVETO_12       ; // VETO to AEM, channel 12  
output     ACD_NVETO_13       ; // VETO to AEM, channel 13  
output     ACD_NVETO_14       ; // VETO to AEM, channel 14  
output     ACD_NVETO_15       ; // VETO to AEM, channel 15  
output     ACD_NVETO_16       ; // VETO to AEM, channel 16  
output     ACD_NVETO_17       ; // VETO to AEM, channel 17  
output     ACD_NCNO          ; // HLD to AEM, indicating CNO  
  
input      CHID_00_IN         ; // GAFE range bit (0 = lo, 1 = hi) for GAFE ch. 00  
input      CHID_01_IN         ; // GAFE range bit (0 = lo, 1 = hi) for GAFE ch. 01  
input      CHID_02_IN         ; // GAFE range bit (0 = lo, 1 = hi) for GAFE ch. 02  
input      CHID_03_IN         ; // GAFE range bit (0 = lo, 1 = hi) for GAFE ch. 03  
input      CHID_04_IN         ; // GAFE range bit (0 = lo, 1 = hi) for GAFE ch. 04  
input      CHID_05_IN         ; // GAFE range bit (0 = lo, 1 = hi) for GAFE ch. 05  
input      CHID_06_IN         ; // GAFE range bit (0 = lo, 1 = hi) for GAFE ch. 06  
input      CHID_07_IN         ; // GAFE range bit (0 = lo, 1 = hi) for GAFE ch. 07  
input      CHID_08_IN         ; // GAFE range bit (0 = lo, 1 = hi) for GAFE ch. 08  
input      CHID_09_IN         ; // GAFE range bit (0 = lo, 1 = hi) for GAFE ch. 09  
input      CHID_10_IN         ; // GAFE range bit (0 = lo, 1 = hi) for GAFE ch. 10  
input      CHID_11_IN         ; // GAFE range bit (0 = lo, 1 = hi) for GAFE ch. 11  
input      CHID_12_IN         ; // GAFE range bit (0 = lo, 1 = hi) for GAFE ch. 12  
input      CHID_13_IN         ; // GAFE range bit (0 = lo, 1 = hi) for GAFE ch. 13  
input      CHID_14_IN         ; // GAFE range bit (0 = lo, 1 = hi) for GAFE ch. 14  
input      CHID_15_IN         ; // GAFE range bit (0 = lo, 1 = hi) for GAFE ch. 15  
input      CHID_16_IN         ; // GAFE range bit (0 = lo, 1 = hi) for GAFE ch. 16  
input      CHID_17_IN         ; // GAFE range bit (0 = lo, 1 = hi) for GAFE ch. 17
```

```

output    HV_ENABLE_1OUT ; // High Voltage Bias Supply A Enable (active hi)
output    HV_ENABLE_2OUT ; // High Voltage Bias Supply B Enable (active hi)

input     DISC_00_IN      ; // Discriminator Input from GAFE ASIC 00
input     DISC_01_IN      ; // Discriminator Input from GAFE ASIC 01
input     DISC_02_IN      ; // Discriminator Input from GAFE ASIC 02
input     DISC_03_IN      ; // Discriminator Input from GAFE ASIC 03
input     DISC_04_IN      ; // Discriminator Input from GAFE ASIC 04
input     DISC_05_IN      ; // Discriminator Input from GAFE ASIC 05
input     DISC_06_IN      ; // Discriminator Input from GAFE ASIC 06
input     DISC_07_IN      ; // Discriminator Input from GAFE ASIC 07
input     DISC_08_IN      ; // Discriminator Input from GAFE ASIC 08
input     DISC_09_IN      ; // Discriminator Input from GAFE ASIC 09
input     DISC_10_IN      ; // Discriminator Input from GAFE ASIC 10
input     DISC_11_IN      ; // Discriminator Input from GAFE ASIC 11
input     DISC_12_IN      ; // Discriminator Input from GAFE ASIC 12
input     DISC_13_IN      ; // Discriminator Input from GAFE ASIC 13
input     DISC_14_IN      ; // Discriminator Input from GAFE ASIC 14
input     DISC_15_IN      ; // Discriminator Input from GAFE ASIC 15
input     DISC_16_IN      ; // Discriminator Input from GAFE ASIC 16
input     DISC_17_IN      ; // Discriminator Input from GAFE ASIC 17
input     HLD_OR_IN      ; // Discriminator Input from GAFE CNO OR (e.g., the HLD)

input     PHAD_00_IN      ; // serial PHA data input from GAFE ADC on channel 0
input     PHAD_01_IN      ; // serial PHA data input from GAFE ADC on channel 1
input     PHAD_02_IN      ; // serial PHA data input from GAFE ADC on channel 2
input     PHAD_03_IN      ; // serial PHA data input from GAFE ADC on channel 3
input     PHAD_04_IN      ; // serial PHA data input from GAFE ADC on channel 4
input     PHAD_05_IN      ; // serial PHA data input from GAFE ADC on channel 5
input     PHAD_06_IN      ; // serial PHA data input from GAFE ADC on channel 6
input     PHAD_07_IN      ; // serial PHA data input from GAFE ADC on channel 7
input     PHAD_08_IN      ; // serial PHA data input from GAFE ADC on channel 8
input     PHAD_09_IN      ; // serial PHA data input from GAFE ADC on channel 9
input     PHAD_10_IN      ; // serial PHA data input from GAFE ADC on channel 10
input    PHAD_11_IN      ; // serial PHA data input from GAFE ADC on channel 11
input    PHAD_12_IN      ; // serial PHA data input from GAFE ADC on channel 12
input    PHAD_13_IN      ; // serial PHA data input from GAFE ADC on channel 13
input    PHAD_14_IN      ; // serial PHA data input from GAFE ADC on channel 14
input    PHAD_15_IN      ; // serial PHA data input from GAFE ADC on channel 15
input    PHAD_16_IN      ; // serial PHA data input from GAFE ADC on channel 16
input    PHAD_17_IN      ; // serial PHA data input from GAFE ADC on channel 17

output   TRIGGER_OUT    ; // test output for trigger timing on bench
output   GAFE_HOLD       ; // active high HOLD signal to GAFE ASICs to control track/hold mode
input    FREE_ID_IN       ; // FREE board identification data input
output   GAFE_STROBE     ; // active high STROBE signal to GAFEs for test charge injection
output   GAFE_DAT         ; // serial configuration data to the GAFE ASICs
output   GAFE_CLK         ; // serial clock (5 MHz) to the GAFE ASICs for configuration data
output   GAFE_RST_OUT    ; // global reset to the GAFE ASICs
input    GAFE_RET_DAT    ; // configuration return data from the GAFE ASICs
output   NADC_CS_OUT     ; // MAX145 ADC active low chip select
output   ADC_CLK_OUT     ; // MAX145 ADC serial conversion clock
output   NDAC_CLR_OUT    ; // MAX5121 active low clear (HVBS DAC)
output   DAC_DAT_RET     ; // MAX5121 serial data in (HVBS DAC)
output   DAC_CLK_OUT     ; // MAX5121 serial clock (HVBS DAC)
output   NDAC_CS_OUT     ; // MAX5121 chip select out (HVBS DAC)
input    DAC_DATA_OUT    ; // MAX5121 serial data out (HVBS DAC)

input    N_G_RESET        ; // reset input from outside core (buffered reset_out)
output   NRESET_OUT       ; // reset output from core, to be buffered and then input again

output   VETO_ENA_OUT    ; // enable bit used to control the VETO_A drivers to the AEM
output   VETO_ENB_OUT    ; // enable bit used to control the VETO_B drivers to the AEM

output   HITMAP_TEST_OUT ; // HitMap test point for oscilloscope verification of HitMap width,
delays

output   CLK_OUT          ; // clock out to external clock buffer

/*********************************************
// Wires for Bus --> Scalar Conversion

wire    ACD_NVETO_00      ;

```

```

wire      ACD_NVETO_01      ;
wire      ACD_NVETO_02      ;
wire      ACD_NVETO_03      ;
wire      ACD_NVETO_04      ;
wire      ACD_NVETO_05      ;
wire      ACD_NVETO_06      ;
wire      ACD_NVETO_07      ;
wire      ACD_NVETO_08      ;
wire      ACD_NVETO_09      ;
wire      ACD_NVETO_10      ;
wire      ACD_NVETO_11      ;
wire      ACD_NVETO_12      ;
wire      ACD_NVETO_13      ;
wire      ACD_NVETO_14      ;
wire      ACD_NVETO_15      ;
wire      ACD_NVETO_16      ;
wire      ACD_NVETO_17      ;
wire      ACD_NCNO      ;

wire      CHID_00_IN      ;
wire      CHID_01_IN      ;
wire      CHID_02_IN      ;
wire      CHID_03_IN      ;
wire      CHID_04_IN      ;
wire      CHID_05_IN      ;
wire      CHID_06_IN      ;
wire      CHID_07_IN      ;
wire      CHID_08_IN      ;
wire      CHID_09_IN      ;
wire      CHID_10_IN      ;
wire      CHID_11_IN      ;
wire      CHID_12_IN      ;
wire      CHID_13_IN      ;
wire      CHID_14_IN      ;
wire      CHID_15_IN      ;
wire      CHID_16_IN      ;
wire      CHID_17_IN      ;

wire      DISC_00_IN      ;
wire      DISC_01_IN      ;
wire      DISC_02_IN      ;
wire      DISC_03_IN      ;
wire      DISC_04_IN      ;
wire      DISC_05_IN      ;
wire      DISC_06_IN      ;
wire      DISC_07_IN      ;
wire      DISC_08_IN      ;
wire      DISC_09_IN      ;
wire      DISC_10_IN      ;
wire      DISC_11_IN      ;
wire      DISC_12_IN      ;
wire      DISC_13_IN      ;
wire      DISC_14_IN      ;
wire      DISC_15_IN      ;
wire      DISC_16_IN      ;
wire      DISC_17_IN      ;
wire      HLD_OR_IN      ;

wire      PHAD_00_IN      ;
wire      PHAD_01_IN      ;
wire      PHAD_02_IN      ;
wire      PHAD_03_IN      ;
wire      PHAD_04_IN      ;
wire      PHAD_05_IN      ;
wire      PHAD_06_IN      ;
wire      PHAD_07_IN      ;
wire      PHAD_08_IN      ;
wire      PHAD_09_IN      ;
wire      PHAD_10_IN      ;
wire      PHAD_11_IN      ;
wire      PHAD_12_IN      ;
wire      PHAD_13_IN      ;
wire      PHAD_14_IN      ;

```

```

wire      PHAD_15_IN      ;
wire      PHAD_16_IN      ;
wire      PHAD_17_IN      ;

wire      NDAC_CLR_OUT   ;
wire      DAC_DAT_RET    ;
wire      DAC_CLK_OUT    ;
wire      DAC_DATA_OUT   ;
wire      NDAC_CS_OUT    ;
wire      GAFE_CLK        ;
wire      NADC_CS_OUT    ;
wire      GAFE_HOLD       ;
wire      HV_ENABLE_1OUT  ;
wire      HV_ENABLE_2OUT  ;

wire [11: 0] pha_thresh_01  ;
wire [11: 0] pha_thresh_02  ;
wire [11: 0] pha_thresh_03  ;
wire [11: 0] pha_thresh_04  ;
wire [11: 0] pha_thresh_05  ;
wire [11: 0] pha_thresh_06  ;
wire [11: 0] pha_thresh_07  ;
wire [11: 0] pha_thresh_08  ;
wire [11: 0] pha_thresh_09  ;
wire [11: 0] pha_thresh_10  ;
wire [11: 0] pha_thresh_11  ;
wire [11: 0] pha_thresh_12  ;
wire [11: 0] pha_thresh_13  ;
wire [11: 0] pha_thresh_14  ;
wire [11: 0] pha_thresh_15  ;
wire [11: 0] pha_thresh_16  ;
wire [11: 0] pha_thresh_17  ;
wire [11: 0] pha_thresh_18  ;

wire [ 5: 0] adc_tacq      ;
wire          data_busy     ;

wire [17: 0] chid         ;
wire [18: 0] disc_in      ;
wire [17: 0] sdin         ;
wire [18: 0] VETO_out     ;

wire [ 4: 0] max_pha      ;
wire [17: 0] pha_enable    ;
wire [17: 0] zs_map       ;
wire [12: 0] pha          ;
wire [ 4: 0] phasel       ;
wire [17: 0] HitMap       ;
wire          cmd_error     ;
wire          PHA_Ready     ;
wire          HitMaprdy    ;
wire          HitMaplch    ;
wire          HitMap_Test   ;
wire          start_pha     ;
wire          TRIGGER_OUT   ;
wire          GAFE_STROBE  ;
wire          LIVE_OUT      ;
wire          lookatme_status;
wire [ 4: 0] Veto_Delay   ;
wire [ 2: 0] Veto_Width   ;
wire [ 2: 0] HitMap_Deadtime;
wire [ 4: 0] HitMap_Delay ;
wire [ 3: 0] HitMap_Width ;
wire [17: 0] Veto_enable  ;
wire          HITMAP_TEST_OUT;
wire          GAFE_DAT      ;
wire          GAFE_RST_OUT  ;
wire          rtnd          ;
wire          zs_status     ;
wire          parity_sel    ;
wire          PWR_ON_RST_IN ;
wire          reset_cmd     ;
wire          cmdd          ;

```

```

wire      edata          ;
wire      capture_map    ;
wire      VETO_ENA_OUT   ;
wire      VETO_ENB_OUT   ;
wire      CLK_OUT        ;
wire      shift_edge_ctrl;
wire      test_mux_ctrl  ;
wire [ 7: 0] FREE_id     ;

wire      ndata_posclk   ; // return data clocked out on the posedge of ACD_CLK
reg       ndata_negclk   ; // add FF to switch data to negedge clocked

//****************************************************************************

assign  ACD_NVETO_00 = ~VETO_out[00] ; // these are negative true logic signals as per ICD
assign  ACD_NVETO_01 = ~VETO_out[01] ;
assign  ACD_NVETO_02 = ~VETO_out[02] ;
assign  ACD_NVETO_03 = ~VETO_out[03] ;
assign  ACD_NVETO_04 = ~VETO_out[04] ;
assign  ACD_NVETO_05 = ~VETO_out[05] ;
assign  ACD_NVETO_06 = ~VETO_out[06] ;
assign  ACD_NVETO_07 = ~VETO_out[07] ;
assign  ACD_NVETO_08 = ~VETO_out[08] ;
assign  ACD_NVETO_09 = ~VETO_out[09] ;
assign  ACD_NVETO_10 = ~VETO_out[10] ;
assign  ACD_NVETO_11 = ~VETO_out[11] ;
assign  ACD_NVETO_12 = ~VETO_out[12] ;
assign  ACD_NVETO_13 = ~VETO_out[13] ;
assign  ACD_NVETO_14 = ~VETO_out[14] ;
assign  ACD_NVETO_15 = ~VETO_out[15] ;
assign  ACD_NVETO_16 = ~VETO_out[16] ;
assign  ACD_NVETO_17 = ~VETO_out[17] ;
assign  ACD_NCNO       = ~VETO_out[18] ;

assign  chid[00]   = CHID_00_IN ;
assign  chid[01]   = CHID_01_IN ;
assign  chid[02]   = CHID_02_IN ;
assign  chid[03]   = CHID_03_IN ;
assign  chid[04]   = CHID_04_IN ;
assign  chid[05]   = CHID_05_IN ;
assign  chid[06]   = CHID_06_IN ;
assign  chid[07]   = CHID_07_IN ;
assign  chid[08]   = CHID_08_IN ;
assign  chid[09]   = CHID_09_IN ;
assign  chid[10]   = CHID_10_IN ;
assign  chid[11]   = CHID_11_IN ;
assign  chid[12]   = CHID_12_IN ;
assign  chid[13]   = CHID_13_IN ;
assign  chid[14]   = CHID_14_IN ;
assign  chid[15]   = CHID_15_IN ;
assign  chid[16]   = CHID_16_IN ;
assign  chid[17]   = CHID_17_IN ;

assign  disc_in[00] = DISC_00_IN ;
assign  disc_in[01] = DISC_01_IN ;
assign  disc_in[02] = DISC_02_IN ;
assign  disc_in[03] = DISC_03_IN ;
assign  disc_in[04] = DISC_04_IN ;
assign  disc_in[05] = DISC_05_IN ;
assign  disc_in[06] = DISC_06_IN ;
assign  disc_in[07] = DISC_07_IN ;
assign  disc_in[08] = DISC_08_IN ;
assign  disc_in[09] = DISC_09_IN ;
assign  disc_in[10] = DISC_10_IN ;
assign  disc_in[11] = DISC_11_IN ;
assign  disc_in[12] = DISC_12_IN ;
assign  disc_in[13] = DISC_13_IN ;
assign  disc_in[14] = DISC_14_IN ;
assign  disc_in[15] = DISC_15_IN ;
assign  disc_in[16] = DISC_16_IN ;
assign  disc_in[17] = DISC_17_IN ;
assign  disc_in[18] = HLD_OR_IN ; // CNO

```

```

assign sdin[00] = PHAD_00_IN ;
assign sdin[01] = PHAD_01_IN ;
assign sdin[02] = PHAD_02_IN ;
assign sdin[03] = PHAD_03_IN ;
assign sdin[04] = PHAD_04_IN ;
assign sdin[05] = PHAD_05_IN ;
assign sdin[06] = PHAD_06_IN ;
assign sdin[07] = PHAD_07_IN ;
assign sdin[08] = PHAD_08_IN ;
assign sdin[09] = PHAD_09_IN ;
assign sdin[10] = PHAD_10_IN ;
assign sdin[11] = PHAD_11_IN ;
assign sdin[12] = PHAD_12_IN ;
assign sdin[13] = PHAD_13_IN ;
assign sdin[14] = PHAD_14_IN ;
assign sdin[15] = PHAD_15_IN ;
assign sdin[16] = PHAD_16_IN ;
assign sdin[17] = PHAD_17_IN ;

assign ndata_posclk = ~ (edata | rtnd) ; // data is active lo, OR'ed with config_data

assign HITMAP_TEST_OUT = (test_mux_ctrl ? LIVE_OUT : HitMap_Test); // test pin multiplexer

//****************************************************************************
// NSDATA Edge Control
// data is active lo, OR'ed with config_data

always @ (negedge ACD_CLK) ndata_negclk = ndata_posclk ;

assign ACD_NSDATA = (shift_edge_ctrl ? ndata_negclk : ndata_posclk) ;

//****************************************************************************
// garc_cmd_proc has 56 ports

garc_cmd_proc M1 (
    .ck(ACD_CLK),
    .cmdd(cmdd),
    .FREE_id(FREE_id),
    .trigger(TRIGGER_OUT),
    .reset_n(N_G_RESET),
    .reset_cmd(reset_cmd),
    .data_busy(data_busy),
    .strobe(GAFE_STROBE),
    .capture_map(capture_map),
    .dac_clr_n(NDAC_CLR_OUT),
    .dac_din(DAC_DAT_RET),
    .dac_sck(DAC_CLK_OUT),
    .dac_dout(DAC_DATA_OUT),
    .dac_cs_n(NDAC_CS_OUT),
    .Veto_Delay(Veto_Delay),
    .Veto_Width(Veto_Width),
    .HitMap_Deadtime(HitMap_Deadtime),
    .HitMap_Delay(HitMap_Delay),
    .HitMap_Width(HitMap_Width),
    .pha_enable(pha_enable),
    .Veto_enable(Veto_enable),
    .gafe_cmdd(GAFE_DAT),
    .gafe_ck(GAFE_CLK),
    .gafe_reset_n(GAFE_RST_OUT),
    .gafe_rtnd(GAFE_RET_DAT),
    .hvbs_a_enable(HV_ENABLE_1OUT),
    .hvbs_b_enable(HV_ENABLE_2OUT),
    .hold(GAFE_HOLD),
    .cmd_error(cmd_error),
    .live(LIVE_OUT),
    .lookatme_status(lookatme_status),
    .max_pha(max_pha),
    .parity_sel(parity_sel),
    .PHA_Ready(PHA_Ready),
    .adc_tacq(adc_tacq),
    .pha_thresh_01(phashresh_01)
);

```

```

.pha_thresh_02(phashresh_02)      ,
.pha_thresh_03(phashresh_03)      ,
.pha_thresh_04(phashresh_04)      ,
.pha_thresh_05(phashresh_05)      ,
.pha_thresh_06(phashresh_06)      ,
.pha_thresh_07(phashresh_07)      ,
.pha_thresh_08(phashresh_08)      ,
.pha_thresh_09(phashresh_09)      ,
.pha_thresh_10(phashresh_10)      ,
.pha_thresh_11(phashresh_11)      ,
.pha_thresh_12(phashresh_12)      ,
.pha_thresh_13(phashresh_13)      ,
.pha_thresh_14(phashresh_14)      ,
.pha_thresh_15(phashresh_15)      ,
.pha_thresh_16(phashresh_16)      ,
.pha_thresh_17(phashresh_17)      ,
.pha_thresh_18(phashresh_18)      ,
.rtnd(rtnd)                      ,
.shift_edge_ctrl(shiftedge_ctrl)  ,
.start_pha(start_pha)            ,
.test_mux_ctrl(testmux_ctrl)     ,
.veto_en_a(VETO_ENA_OUT)         ,
.veto_en_b(VETO_ENB_OUT)          ,
.zs_status(zs_status)            ,
);

//****************************************************************************

// event_data has 15 ports

event_data M2 (
    .ck(ACD_CLK)                  ,
    .reset_n(N_G_RESET)           ,
    .data_busy(data_busy)          ,
    .edata(edata)                 ,
    .max_pha(max_pha)             ,
    .pha_enable(pha_enable)        ,
    .cmd_error(cmd_error)          ,
    .zs_map(zs_map)               ,
    .zs_status(zs_status)          ,
    .parity_sel(parity_sel)        ,
    .pha(pha)                      ,
    .phasel(phasel)               ,
    .HitMap(HitMap)                ,
    .PHA_Ready(PHA_Ready)           ,
    .HitMaprdy(HitMaprdy)          ,
    .HitMaplch(HitMaplch)          ,
);

//****************************************************************************

// pha_logic has 29 ports

pha_logic M3 (
    .ck(ACD_CLK)                  ,
    .reset_n(N_G_RESET)           ,
    .start_pha(start_pha)          ,
    .chid(chid)                   , // range input bits from the 18 GAFE
    .sdin(sdin)                   ,
    .sck(ADC_CLK_OUT)              ,
    .pha_cs_n(NADC_CS_OUT)         ,
    .pha_thresh_00(phashresh_01)    ,
    .pha_thresh_01(phashresh_02)    ,
    .pha_thresh_02(phashresh_03)    ,
    .pha_thresh_03(phashresh_04)    ,
    .pha_thresh_04(phashresh_05)    ,
    .pha_thresh_05(phashresh_06)    ,
    .pha_thresh_06(phashresh_07)    ,
    .pha_thresh_07(phashresh_08)    ,
    .pha_thresh_08(phashresh_09)    ,
    .pha_thresh_09(phashresh_10)    ,
    .pha_thresh_10(phashresh_11)    ,
    .pha_thresh_11(phashresh_12)    ,

```



```

// GARC Command Processor Module, "garc_cmd_proc_v3.v"

module garc_cmd_proc (
    ck,                      // in : serial 20 MHz command clock from the AEM
    cmdd,                     // in : serial command data from the AEM
    FREE_id,                  // in : data bus representing FREE circuit card identification code
    trigger,                  // out: trigger output to hold delay circuit
    reset_n,                  // in : GARC global reset into core (active low)
    reset_cmd,                // out: ACD reset configuration command, active high
    strobe,                   // out: active high Test Charge Input signal to GAFEs
    data_busy,                // in : event_data module active high busy signal
    capture_map,              // out: signal to disc_logic to capture bitmap
    dac_din,                 // out: HVBS DAC serial configuration data
    dac_dout,                 // in : HVBS DAC serial readback of configuration data
    dac_sck,                 // out: HVBS DAC clock
    dac_clr_n,                // out: HVBS DAC active low reset
    dac_cs_n,                 // out: HVBS DAC active low data envelope
    Veto_Delay,               // out: Time for delay before Veto is output
    Veto_Width,               // out: Width of output Veto signal
    HitMap_Deadtime,          // out: Time for deadtime of HitMap signal at trailing edge
    HitMap_Delay,              // out: Time for delay of HitMap signal in delay line
    HitMap_Width,              // out: Width of output HitMap signal
    hvbs_a_enable,             // out: HVBS A Enable signal, active hi
    hvbs_b_enable,             // out: HVBS B Enable signal, active hi
    pha_enable,                // out: Active high enable bits to enable PHA data to be sent per channel
    Veto_enable,               // out: Discriminator enables, per channel
    gafe_cmdd,                // out: GAFE command data
    gafe_ck,                  // out: GAFE command clock
    gafe_rtnd,                // in : GAFE return data
    gafe_reset_n,              // out: GAFE reset line
    hold,                     // out: GAFE analog hold
    cmd_error,                // out: command error flag to event_data module
    live,                     // out: active hi flag indicating ready to receive or is receiving cmds
    lookatme_status,           // in : Look-at-Me command status
    max_pha,                  // out: register for maximum PHA allowable to send
    parity_sel,                // out: parity selection commanded from garc mode register
    PHA_Ready,                 // in : pha ready signal flag from pha_logic
    adc_tacq,                 // out: pha acquisition time delay
    pha_thresh_01,              // out: pha zs threshold for GAFE ASIC #01
    pha_thresh_02,              // out: pha zs threshold for GAFE ASIC #02
    pha_thresh_03,              // out: pha zs threshold for GAFE ASIC #03
    pha_thresh_04,              // out: pha zs threshold for GAFE ASIC #04
    pha_thresh_05,              // out: pha zs threshold for GAFE ASIC #05
    pha_thresh_06,              // out: pha zs threshold for GAFE ASIC #06
    pha_thresh_07,              // out: pha zs threshold for GAFE ASIC #07
    pha_thresh_08,              // out: pha zs threshold for GAFE ASIC #08
    pha_thresh_09,              // out: pha zs threshold for GAFE ASIC #09
    pha_thresh_10,              // out: pha zs threshold for GAFE ASIC #10
    pha_thresh_11,              // out: pha zs threshold for GAFE ASIC #11
    pha_thresh_12,              // out: pha zs threshold for GAFE ASIC #12
    pha_thresh_13,              // out: pha zs threshold for GAFE ASIC #13
    pha_thresh_14,              // out: pha zs threshold for GAFE ASIC #14
    pha_thresh_15,              // out: pha zs threshold for GAFE ASIC #15
    pha_thresh_16,              // out: pha zs threshold for GAFE ASIC #16
    pha_thresh_17,              // out: pha zs threshold for GAFE ASIC #17
    pha_thresh_18,              // out: pha zs threshold for GAFE ASIC #18
    rtnd,                     // out: return data out to AEM from readback command
    shift_edge_ctrl,            // out: return data shift edge control
    start_pha,                 // out: signal to pha_logic module to indicate ADC start
    test_mux_ctrl,              // out: signal to control function of test pin
    veto_en_a,                  // out: enable signal for Veto set A
    veto_en_b,                  // out: enable signal for Veto set B
    zs_status                  // out: zero suppression enable
);

/*****************************************/
/*
-----|-----|-----|-----|
| | GLAST ACD Readout Controller ASIC (GARC) | |
| | Command Processor Logic | |

```

| | Project : Gamma-Large Area Space Telescope (GLAST)  
| | Anti-Coincidence Detector (ACD)  
| | GLAST ACD Readout Controller (GARC) ASIC

| | Written : D. Sheppard  
| | Module : garc\_cmd\_proc\_v3.v  
| | Original : 10-19-01  
| | Updated : 03-28-03  
| | Version : 3.0

| |-----  
| | VERILOG-XL 3.10.p001  
| |-----

| This software is property of the National Aeronautics and Space  
| Administration. Unauthorized use or duplication of this  
| software is strictly prohibited. Authorized users are subject  
| to the following restrictions:

- | 1. Neither the author, their corporation, nor NASA is  
| responsible for any consequences of the use of this software.
- | 2. The origin of this software must not be misrepresented either  
| by explicit claim or by omission.
- | 3. Altered versions of this software must be plainly marked  
| as such.
- | 4. This notice may not be removed or altered.

\*\*\*\*\*  
03-28-03: Change sck\_ctrl from 3 to 1 in state SEND\_GAFE\_ZEROES to correct GAFE  
data readback timing/delay problem.

03-07-03: Update VERSION number to 3 for readback.

09-24-02: Update GAFE Reset Circuitry to be commandable. Clear the gafe\_reset\_ctrl  
at reset\_n, then count up to 255 with GAFE\_RESET\_DURATION. Previously,  
gafe\_reset\_ctrl was incorrectly initialized at reset\_n.

09-16-02: Update GARC Version from "1" to "2".  
Remove registers and commands related to HLD enable (this had no function)  
Removed reference to "reset\_cmd" from "initial\_cmd\_reset" to preclude the  
race condition seen on commandable reset in GARC V1.  
Remove sending gafe\_cks at reset.  
Put in shift\_edge\_ctrl parameter as a command bit and module output.

03-19-02: Update diagnostic status register. Add commands rejected readout in cmd  
id {Addr 2, Func 12}. Change trigger command logic to work on trig\_cmd\_id.

03-18-02: Add GARC Version readback and FREE board ID readback. Add control for  
test mux pin. Update logic for hold\_delay.

03-15-02: Update default PHA threshold again. Add pha acquisition delay counter cmd.  
Add data\_busy input from event\_data module.

03-14-02: Update power on defaults. Update contents of garc\_status register.

03-13-02: Change polarity on reset to active low to match Tanner FF. Update to the  
trigger logic. Update default values of PHA thresholds, Veto and HLD  
enables. Move test for "strobe" clear to every clock cycle. Change "live"  
to a registered signal.

03-12-02: Update trigger format for non-ZS type triggers

03-11-02: Update state 0 in trigger processor state machine

03-08-02: Fixed gafe\_reset conflict problem. Moved reset logic out of this module,  
up one level to garc\_module to enable new look-at-me circuitry.

03-07-02: Removed "spare" register.

03-06-02: Fix problem with only even parity being sent to GAFE (not full implementation)

of gafe\_parity\_select). Change start\_pha to wake up ADC immediately upon trigger. Command processor now goes "live" immediately after strobe command is processed, not waiting for delay. Strobe will now go active at command and inactive at end-of-conversion. Changed readback word to include the new data/dataless bit. Add reset\_out and reset (as input) to allow reset signal to exit core and return as input to preclude previous reset routing errors. Add capability for FREE board power-on reset. Changed GAFE reset block to be a task inside main "posedge ck" block.

- 03-05-02: Change GAFE command parity computation to exclude start bit from parity.
- 02-19-02: Change HitMap latch time to be immediately after trigger instead of after EOC. Change pha\_thresh\_n registers to 12 bits from 16
- 02-18-02: Add in veto\_en\_a and veto\_en\_b signals to enable turning on and off the veto signals. Add comments.
- 02-15-02: Updated trigger command format (5 bits to 4 bits) and updated config command format to include the "dataless" bit.
- 02-14-02: Updated comments; add control for second HV supply. Added GAFE parity command capability
- 02-13-02: Updated command definitions to match ICD
- 02-08-02: Updated DAC sck from 2 MHz to 5 MHz
- 02-07-02: Updated GAFE sck from 2 MHz to 5 MHz
- 02-06-02: Testing trigger/hold commands. Update state machines to correctly clear cmd\_sr under all commanding modes. Additionally, fix intermittent reset command problem. Fix hold\_delay = 0 anomaly.
- 01-30-02: Testing DAC commands with Bob. Fix problem of sending DAC command twice and update to use a fixed data pattern for a MAX5121 read. Also, fix GAFE reset cmd to operate consistently instead of every other time.
- 12-31-01: Started testing in Altera version of test fixture. Discovered GAFE reset duration needs to be extended, GAFE clock was not 50% and last GAFE command data bit needs to be fixed. Fix attempted below in GAFE state machine.
- 01-02-02: Continued debugging GAFE command data and return, especially gafe\_ck and parity formation.
- 01-03-02: Added another fix to gafe\_ck and worked on gafe\_reset fix. Also, needed to fix parity calculation in GAFE command.
- 01-04-02: Changed gafe\_command\_handler and dac\_command\_handler to run in main ck loop as opposed to starting with the configuration command state machine. Changed gafe\_command\_handler to produce GAFE clocks when gafe\_reset is active.

\*\*\*\*\*  
DESCRIPTION OF THE GARC COMMAND PROCESSOR

Description: This module is the main command processor for the GARC, the GLAST ACD digital ASIC. This module receives serial commands from the ACD AEM, checks for command validity, and processes all valid ACD commands. GAFE commands are sent out from the GARC at a slower clock rate (5 MHz). This module controls the MAX5121 DAC.

A summary of GARC command processor functions includes:

- (1) Receives GARC configuration commands and writes to GARC registers
- (2) Receives GARC readback commands and returns the contents of GARC registers
- (3) Receives GAFE configuration commands, reformats, and transmits these commands to the GAFEs
- (4) Receives GAFE readback commands, reformats, receives GAFE data, and transmits to AEM
- (5) Receives trigger commands and initiates a hold and data readout cycle
- (6) Controls the MAX5121 DAC for the HVBS

GARC Registers: (as per the 2-15-02 version of the ICD)

Funct Blk	Funct Num.	Num. Bits	Function Name	Register(s) affected
(5 bits)	(4 bits)			
0	0001 (1)	0	Reset	"reset" scalar, active low
0	0010 (2)	5	Veto Delay command	Veto_Delay
0	0011 (3)	0	Calibration Pulse	"strobe" scalar, active hi
0	0111 (7)	16	Hold Delay	delay_ctr
0	1000 (8)	12	HVBS DAC Level command	hvbs_level
0	1001 (9)	12	HVBS SAA Level command	saa_level
0	1010 (10)	0	Use HV Nominal command	hv_nom_command
0	1011 (11)	0	Use SAA Level command	saa_level_command
0	1100 (12)	7	Hold Delay command	Hold_Delay
0	1101 (13)	3	VETO Width command	Veto_Width
0	1110 (14)	4	HitMap Width command	HitMap_Width
0	1111 (15)	3	HitMap Deadtime command	HitMap_Deadtime
1	0100 (4)	5	Look-At-Me Command	garc_status[0]
1	1000 (8)	5	HitMap Delay command	HitMap_Delay
1	1001 (9)	16	PHA readout enable 16-1	pha_enable
1	1010 (10)	16	VETO enable 16-1	Veto_enable
1	1100 (12)	2	PHA readout enable 18,17	pha_enable
1	1101 (13)	2	VETO enable 18,17	Veto_enable
1	1111 (15)	5	Maximum PHA words to send	max_pha
2	1000 (8)	12	GARC mode	garc_mode
2	1001 (9)	5	GARC status	garc_status
2	1010 (10)	16	Command register	command_register
2	1011 (11)	16	GARC diagnostic command	diagnostic_status
2	1100 (12)	8	GARC cmd reject counter	readback only
2	1101 (13)	8	FREE Board ID	readback only
2	1110 (14)	3	GARC Version	readback only
2	1111 (15)	0	No Operation Command	nop
3	1000 (8)	12	PHA Threshold 1	pha_thresh_01
3	1001 (9)	12	PHA Threshold 2	pha_thresh_02
3	1010 (10)	12	PHA Threshold 3	pha_thresh_03
3	1011 (11)	12	PHA Threshold 4	pha_thresh_04
3	1100 (12)	12	PHA Threshold 5	pha_thresh_05
3	1101 (13)	12	PHA Threshold 6	pha_thresh_06
3	1110 (14)	12	PHA Threshold 7	pha_thresh_07
4	1000 (8)	12	PHA Threshold 8	pha_thresh_08
4	1001 (9)	12	PHA Threshold 9	pha_thresh_09
4	1010 (10)	12	PHA Threshold 10	pha_thresh_10
4	1011 (11)	12	PHA Threshold 11	pha_thresh_11
4	1100 (12)	12	PHA Threshold 12	pha_thresh_12
4	1101 (13)	12	PHA Threshold 13	pha_thresh_13
4	1110 (14)	12	PHA Threshold 14	pha_thresh_14
5	1000 (8)	12	PHA Threshold 15	pha_thresh_15
5	1001 (9)	12	PHA Threshold 16	pha_thresh_16
5	1010 (10)	12	PHA Threshold 17	pha_thresh_17
5	1011 (11)	12	PHA Threshold 18	pha_thresh_18
5	1100 (12)	6	ADC Acquistion Time	adc_tacq

\*\*\*\*\*

The Trigger command format is four bits and is as follows:

1100 for a ZS enabled trigger (send only PHA above threshold)  
 1010 for a ZS disabled trigger (send all PHA)

The GARC configuration command format (include GAFE commands) is 34 bits as follows:

Configuration command format (34 bits):

```

bit      33 : start bit = 1
bits [32:31]: 00 for command
bit      30 : odd parity over bits [32:31]
bit      29 : GARC select (0) or GAFE select (1)
  
```

```

bits [28:24]: Address (GAFE address, GARC function block)
bit      23 : Read (1) or Write (0)
bit      22 : Data/Dataless bit (0 = dataless, 1 = data)
bits [21:18]: Register or function number
bit      17 : odd parity over bits [32:18]
bits [16: 1]: command data field
bit      0 : odd parity over bits [16:1]

```

Command Readback format (31 bits):

```

bit      31 : start bit = 1
bit      30 : GAFE/GARC select bit (GARC=0, GAFE=1)
bits [29:25]: GAFE/GARC address
bit      24 : read/write bit (1=read)
bit      23 : Data/Dataless bit
bits [22:19]: function number
bit      18 : odd parity over bits [30:19]
bits [17: 2]: command data, MSB first
bit      1 : error detected in parity
bit      0 : odd parity over bits [17:1]

```

The GAFE command format is:

```

bit      [27]: start bit = 1
bit      [26]: write bit = 0, read bit = 1
bits [25:21]: GAFE Address
bits [20:17]: GAFE Register Select
bits [16:1]: GAFE Command Data
bit      [0]: Odd parity bit

```

---

GARC Mode Functions (garc\_mode is a 11 bit register, [10:0])

```

garc_mode[ 0] is the parity select bit (0 = odd parity, 1 = even parity)
garc_mode[ 1] is the HVBS A enable bit for TMR reg #1 (0 = disabled , 1 = enabled)
garc_mode[ 2] is the HVBS A enable bit for TMR reg #2 (0 = disabled , 1 = enabled)
garc_mode[ 3] is the HVBS A enable bit for TMR reg #3 (0 = disabled , 1 = enabled)
garc_mode[ 4] is the HVBS B enable bit for TMR reg #1 (0 = disabled , 1 = enabled)
garc_mode[ 5] is the HVBS B enable bit for TMR reg #2 (0 = disabled , 1 = enabled)
garc_mode[ 6] is the HVBS B enable bit for TMR reg #3 (0 = disabled , 1 = enabled)
garc_mode[ 7] is the parity select bit for GAFE commands (0 = odd parity, 1 = even parity)
garc_mode[ 8] is the Veto A enable bit (0 = disabled, 1 = enabled)
garc_mode[ 9] is the Veto B enable bit (0 = disabled, 1 = enabled)
garc_mode[10] is the control for the test pin mux (0 = HitMap_Test, 1 = live)
garc_mode[11] is the control for the return data shift edge (0 = negedge, 1 = posedge)

```

GARC Status Functions (garc\_status is a 6 bit register, [5:0])

```

garc_status[0] is the value of the look-at-me bit. (0 = prime, 1 = secondary)
garc_status[1] is HV_Enable_1
garc_status[2] is HV_Enable_2
garc_status[3] is veto_en_a
garc_status[4] is veto_en_b
garc_status[5] is zs_status

```

---

Diagnostic Functions: A status register will be maintained in the GARC to monitor the status of the chip. The register contents will be available for serial readout.

command\_ct is an 8 bit register that counts the number of valid config commands received by the GARC. This is initialized to zero by reset.

reject\_ct is an 8 bit register that counts the number of rejected commands received by the GARC. This is initialized to zero by reset.

```

diagnostic_status[ 15 ] = parity_error
diagnostic_status[ 14 ] = cmd_parity_error
diagnostic_status[ 13 ] = data_parity_error

```

```

diagnostic_status[ 12 ] = cmd_error
diagnostic_status[11:8] = diagnostic state machine loop counter
diagnostic_status[ 7:0] = valid command counter

*/
/* GARC Processor Global Definitions */

`define GARC_Version      3
`define Cmd_Start_Bit    1'b1
`define GARC_Cmd_Type    3'b000
`define GARC_Select       0
`define GAFE_Select       1
`define GAFE_Broadcast    5'h1F
`define GARC_Read_Cmd    1'b1
`define GARC_Write_Cmd   1'b0
`define Readback_Header  12'h810

/* Address and Function Definitions */
`define Addr_0            5'h0
`define Addr_1            5'h1
`define Addr_2            5'h2
`define Addr_3            5'h3
`define Addr_4            5'h4
`define Addr_5            5'h5

`define Func_0            4'h0
`define Func_1            4'h1
`define Func_2            4'h2
`define Func_3            4'h3
`define Func_4            4'h4
`define Func_5            4'h5
`define Func_6            4'h6
`define Func_7            4'h7
`define Func_8            4'h8
`define Func_9            4'h9
`define Func_A            4'ha
`define Func_B            4'hb
`define Func_C            4'hc
`define Func_D            4'hd
`define Func_E            4'he
`define Func_F            4'hf

/* Command Definitions */
// Address 0
`define Reset_Cmd          {`Addr_0, `Func_1} // GARC Command
`define Veto_Delay_Cmd     {`Addr_0, `Func_2} // GARC Command to disc_logic
`define Calib_Cmd          {`Addr_0, `Func_3} // GARC Command (generates GAFE Strobe)
`define HVBS_Cmd           {`Addr_0, `Func_8} // GARC Register Command
`define SAA_Cmd            {`Addr_0, `Func_9} // GARC Register Command
`define Use_HV_Nom_Cmd     {`Addr_0, `Func_A} // *HVBS DAC* Command
`define Use_HV_SAA_Cmd     {`Addr_0, `Func_B} // *HVBS DAC* Command
`define Hold_Delay_Cmd     {`Addr_0, `Func_C} // GARC Register Command
`define Veto_Width_Cmd      {`Addr_0, `Func_D} // GARC Command to disc_logic
`define HitMap_Width_Cmd   {`Addr_0, `Func_E} // GARC Command to disc_logic
`define HitMap_Deadtime_Cmd {`Addr_0, `Func_F} // GARC Command to disc_logic, 3 bits

// Address 1
`define Lookatme_Cmd       {`Addr_1, `Func_4} // GARC Command to look at AEM
`define HitMap_Delay_Cmd   {`Addr_1, `Func_8} // GARC Command to disc_logic
`define PHA_En0_Reg_Cmd    {`Addr_1, `Func_9} // GARC Command to pha_logic
`define VETO_En0_Reg_Cmd   {`Addr_1, `Func_A} // GAFE Command

`define PHA_En1_Reg_Cmd    {`Addr_1, `Func_C} // GARC Command to pha_logic
`define VETO_En1_Reg_Cmd   {`Addr_1, `Func_D} // GAFE Command

`define Max_PHA_Cmd        {`Addr_1, `Func_F} // GARC Command to event_data

// Address 2
`define GARC_Mode_Cmd      {`Addr_2, `Func_8} // GARC mode command
`define GARC_Status_Cmd     {`Addr_2, `Func_9} // GARC Command, cmd_processor internal

```

```

`define Command_Reg_Cmd      {`Addr_2, `Func_A} // GARC Command, 1st 16 bits of last cmd error
`define GARC_Diag_Cmd       {`Addr_2, `Func_B} // GARC diagnostic register command
`define Command_Reject_Cmd  {`Addr_2, `Func_C} // GARC cmd_reject readback
`define FREE_ID_Cmd         {`Addr_2, `Func_D} // GARC Command to readback FREE Board ID
`define Version_Cmd          {`Addr_2, `Func_E} // GARC Command to readback GARC Version
`define No_Operation_Cmd    {`Addr_2, `Func_F} // Toggles the nop bit

// Address 3
`define PHA_Thresh_01_Cmd   {`Addr_3, `Func_8} // GARC Command to pha_logic
`define PHA_Thresh_02_Cmd   {`Addr_3, `Func_9} // GARC Command to pha_logic
`define PHA_Thresh_03_Cmd   {`Addr_3, `Func_A} // GARC Command to pha_logic
`define PHA_Thresh_04_Cmd   {`Addr_3, `Func_B} // GARC Command to pha_logic
`define PHA_Thresh_05_Cmd   {`Addr_3, `Func_C} // GARC Command to pha_logic
`define PHA_Thresh_06_Cmd   {`Addr_3, `Func_D} // GARC Command to pha_logic
`define PHA_Thresh_07_Cmd   {`Addr_3, `Func_E} // GARC Command to pha_logic

// Address 4
`define PHA_Thresh_08_Cmd   {`Addr_4, `Func_8} // GARC Command to pha_logic
`define PHA_Thresh_09_Cmd   {`Addr_4, `Func_9} // GARC Command to pha_logic
`define PHA_Thresh_10_Cmd   {`Addr_4, `Func_A} // GARC Command to pha_logic
`define PHA_Thresh_11_Cmd   {`Addr_4, `Func_B} // GARC Command to pha_logic
`define PHA_Thresh_12_Cmd   {`Addr_4, `Func_C} // GARC Command to pha_logic
`define PHA_Thresh_13_Cmd   {`Addr_4, `Func_D} // GARC Command to pha_logic
`define PHA_Thresh_14_Cmd   {`Addr_4, `Func_E} // GARC Command to pha_logic

// Address 5
`define PHA_Thresh_15_Cmd   {`Addr_5, `Func_8} // GARC Command to pha_logic
`define PHA_Thresh_16_Cmd   {`Addr_5, `Func_9} // GARC Command to pha_logic
`define PHA_Thresh_17_Cmd   {`Addr_5, `Func_A} // GARC Command to pha_logic
`define PHA_Thresh_18_Cmd   {`Addr_5, `Func_B} // GARC Command to pha_logic
`define ADC_TAcq_Cmd        {`Addr_5, `Func_C} // GARC Command to pha_logic

//****************************************************************************
// Module I/O Declarations

/* module inputs */
input      ck           ;
input      cmdd         ;
input      reset_n      ;
input      dac_dout     ;
input      gafe_rtnd    ;
input      PHA_Ready    ;
input      lookatme_status ;
input      data_busy     ;
input [ 7: 0] FREE_id    ;

/* module outputs */
output     reset_cmd    ;
output     trigger       ;
output     strobe        ;
output     capture_map  ;
output     dac_clr_n    ;
output     dac_din      ;
output     dac_sck      ;
output     dac_cs_n     ;
output [ 4:0] Veto_Delay ;
output [ 2:0] Veto_Width ;
output [ 2:0] HitMap_Deadtime ;
output [ 4:0] HitMap_Delay ;
output [ 3:0] HitMap_Width ;
output [17: 0] pha_enable ;
output     gafe_cmdd    ;
output     gafe_ck      ;
output     gafe_reset_n ;
output     hvbs_a_enable ;
output     hvbs_b_enable ;
output     hold          ;
output     cmd_error    ;
output     live          ;
output [ 4: 0] max_pha  ;
output     parity_sel    ;
output [ 5: 0] adc_tacq  ;
output [11: 0] pha_thresh_01 ;

```

```

output [11: 0] pha_thresh_02      ;
output [11: 0] pha_thresh_03      ;
output [11: 0] pha_thresh_04      ;
output [11: 0] pha_thresh_05      ;
output [11: 0] pha_thresh_06      ;
output [11: 0] pha_thresh_07      ;
output [11: 0] pha_thresh_08      ;
output [11: 0] pha_thresh_09      ;
output [11: 0] pha_thresh_10      ;
output [11: 0] pha_thresh_11      ;
output [11: 0] pha_thresh_12      ;
output [11: 0] pha_thresh_13      ;
output [11: 0] pha_thresh_14      ;
output [11: 0] pha_thresh_15      ;
output [11: 0] pha_thresh_16      ;
output [11: 0] pha_thresh_17      ;
output [11: 0] pha_thresh_18      ;
output          rtnd      ;
output          shift_edge_ctrl  ;
output          start_pha      ;
output          test_mux_ctrl   ;
output [17: 0] Veto_enable       ;
output          veto_en_a       ;
output          veto_en_b       ;
output          zs_status       ;

//****************************************************************************
// Command Processor register declarations

/* Registers & Wires Used in the RESET functions */

reg      reset_cmd      ; // reset command register
reg      gafe_reset_n    ; // reset line to GAFE ASICs
reg [ 7: 0] gafe_rst_ctr  ; // counter used to set the length of the "reset" pulse
wire     gafe_parity_sel  ; // GAFE command parity select, 0 = odd, 1 = even (odd is default)

/* Registers & Wires Commandable Via AEM --> GARC Command */

reg [ 5: 0] adc_tacq      ; // ADC Acquisition Time counter value
reg [11: 0] pha_thresh_01  ; // Zero-Suppression (ZS) threshold value for PHA at Address 00
reg [11: 0] pha_thresh_02  ; // Zero-Suppression (ZS) threshold value for PHA at Address 01
reg [11: 0] pha_thresh_03  ; // Zero-Suppression (ZS) threshold value for PHA at Address 02
reg [11: 0] pha_thresh_04  ; // Zero-Suppression (ZS) threshold value for PHA at Address 03
reg [11: 0] pha_thresh_05  ; // Zero-Suppression (ZS) threshold value for PHA at Address 04
reg [11: 0] pha_thresh_06  ; // Zero-Suppression (ZS) threshold value for PHA at Address 05
reg [11: 0] pha_thresh_07  ; // Zero-Suppression (ZS) threshold value for PHA at Address 06
reg [11: 0] pha_thresh_08  ; // Zero-Suppression (ZS) threshold value for PHA at Address 07
reg [11: 0] pha_thresh_09  ; // Zero-Suppression (ZS) threshold value for PHA at Address 08
reg [11: 0] pha_thresh_10  ; // Zero-Suppression (ZS) threshold value for PHA at Address 09
reg [11: 0] pha_thresh_11  ; // Zero-Suppression (ZS) threshold value for PHA at Address 0A
reg [11: 0] pha_thresh_12  ; // Zero-Suppression (ZS) threshold value for PHA at Address 0B
reg [11: 0] pha_thresh_13  ; // Zero-Suppression (ZS) threshold value for PHA at Address 0C
reg [11: 0] pha_thresh_14  ; // Zero-Suppression (ZS) threshold value for PHA at Address 0D
reg [11: 0] pha_thresh_15  ; // Zero-Suppression (ZS) threshold value for PHA at Address 0E
reg [11: 0] pha_thresh_16  ; // Zero-Suppression (ZS) threshold value for PHA at Address 0F
reg [11: 0] pha_thresh_17  ; // Zero-Suppression (ZS) threshold value for PHA at Address 10
reg [11: 0] pha_thresh_18  ; // Zero-Suppression (ZS) threshold value for PHA at Address 11
reg      config_type_parity; // parity bit for configuration type commands
reg      cmd_parity        ; // parity over 15 command bits
reg      data_parity        ; // parity over 16 data bits
reg      cmd_parity_error   ; // parity error detected in command word
reg      data_parity_error  ; // parity error detected in data word
reg      parity_error       ; // register to detect, store parity error information
reg      trigger            ; // trigger
reg [ 7: 0] command_ct     ; // valid command counter
reg [ 7: 0] reject_ct      ; // rejected command counter
reg [ 3: 0] cp_state       ; // command processor state variable
reg [15: 0] diagnostic_status; // diagnostic status register
wire     hvbs_a_enable     ; // active high HVBS A enable
wire     hvbs_b_enable     ; // active high HVBS B enable
wire     parity_sel         ; // 0 = ODD (default), 1 = EVEN (test mode)

```

```

reg [11: 0] hvbs_dac      ; // value for the NOMINAL Max5121 DAC output
reg [11: 0] saa_level     ; // value for the SAA MODE Max5121 DAC output
reg [ 6: 0] hold_delay    ; // value for setting the trigger-to-hold delay
reg [ 6: 0] delay_ctr     ; // counter used to count out the trigger-to-hold delay
reg [ 4: 0] Veto_Delay     ; // delay from discriminator edge to Veto output edge
reg [ 2: 0] Veto_Width     ; // width setting for the AEM Veto pulses
reg [ 2: 0] HitMap_Deadtime ; // duration of stretch time for the HitMap pulse in the DDL
reg [ 4: 0] HitMap_Delay   ; // delay from discriminator leading edge to HitMap leading edge
reg [ 3: 0] HitMap_Width    ; // width of the HitMap pulse
reg [17: 0] pha_enable     ; // individual enable of channels for the PHA word enable
reg [17: 0] Veto_enable     ; // individual enable of channels for the GAFE Veto discriminators

reg      zs_status          ; // status of zero-suppression during trigger
reg [ 8: 0] config_function ; // configuration function bits - used in command processing
reg      nop                 ; // "no operation" flag bit
reg      trigger_cmd_id     ; // trigger command identification (vs config cmd)
reg [11: 0] garc_mode       ; // GARC mode register, controls parity, HVBS, & Veto output
reg [ 4: 0] max_pha         ; // register controlling the maximum number of PHAs to be sent
wire     test_mux_ctrl      ; // wire for the control of the test mux pin

/* Registers for GAFE Configuration & Operation */

reg      strobe              ; // test charge injection calibration pulse to GAFEs
reg [27: 0] gafe_command    ; // GAFE command register
reg      gafe_cmdd           ; // serial command data to the GAFEs
reg      gafe_ck              ; // serial command clock for GAFE configuration
reg      hold                ; // output HOLD signal to the GAFE ASICs
reg [ 1: 0] sck_ctr          ; // register used to create the divide-by-10 clock for GAFE
reg [ 2: 0] acmd_st          ; // analog (GAFE) commands state machine variable
reg [ 4: 0] gafe_bit          ; // bit counter for use in sending GAFE commands
reg      gafe_command_sent   ; // semaphore from GAFE state machine that GAFE command was transmitted

/* Registers for DAC Configuration */

reg      dac_din             ; // serial data output to MAX5121 DAC
reg      dac_sck              ; // MAX5121 DAC configuration clock
reg      dac_clr_n            ; // MAX5121 reset line
reg      dac_cs_n             ; // MAX5121 chip select
reg [15: 0] dac_data_wd      ; // configuration data to be transmitted to MAX5121 DAC
reg [15: 0] dac_data_readback ; // configuration readback of MAX5121 data
reg [ 1: 0] dac_st             ; // dac commands state machine variable
reg [ 1: 0] dac_ck_ctr        ; // register used to create the divide-by-10 clock for MAX5121 DAC
reg      dac_command_sent     ; // signal from DAC state machine that DAC command was transmitted

/* Registers for GARC State Machine */

reg [33: 0] cmd_sr           ; // command shift register used to capture AEM commands
reg      capture_map          ; // semaphore to indicate to disc_logic to latch the HitMap
reg [ 5: 0] garc_status        ; // register containing the present status of the GARC cmd processor
reg [15: 0] command_register   ; // command register used to readback command errors
reg      rtnd                 ; // the configuration readback return data wire
reg      start_pha             ; // semaphore to signal garc_pha_logic to start the ADC conversion
reg [ 5: 1] read_ct            ; // command readback bit counter
reg [31: 0] readback_word      ; // command readback word storage register
reg [ 4: 0] command_address    ; // command address register
reg [ 3: 0] command_function   ; // command function register
reg      hv_nom_command        ; // indicates to DAC state machine a DAC command is ready to send
reg      send_gafe_command      ; // indicates to the GAFE state machine that a GAFE cmd is ready to send
reg      saa_level_command      ; // controlled in main loop
reg [16: 0] gafe_readback_reg  ; // contents of the GAFE register readback
reg [ 3: 0] data_ct            ; // register used in DAC state machine to count bits sent
reg [ 1:0] trig_st             ; // trigger handler state machine
reg      live                  ; // active high "state machine ready for cmd" signal
wire     veto_en_a              ; // enable signal for AEM "A" drivers for AEM Vetos
wire     veto_en_b              ; // enable signal for AEM "B" drivers for AEM Vetos
wire     shift_edge_ctrl        ; // return data edge shift polarity control

/*****************************************/

```

```

// Local Command Processor Module Parameters

parameter

/* Reset Initialization Parameters */
INIT_CAPTURE_MAP      = 0           ,
INIT_CMD_REG          = 0           ,
INIT_CMD_SR           = 0           ,
INIT_DATA_CT          = 15          ,
INIT_GAFE_CMD          = 0           ,
INIT_GARC_MODE         = 12'b00011000000000,
INIT_GARC_STATUS       = 0           ,
INIT_HITMAP_DEADTIME= 3'h3          ,
INIT_HITMAP_DELAY     = 5'h10         ,
INIT_HITMAP_WIDTH     = 3'h7          ,

INIT_HOLD              = 0           ,
INIT_HOLD_DELAY        = 7'd28         ,
INIT_HVBS_DAC          = 0           ,
INIT_MAX_PHA           = 4            ,
INIT_READ_CT           = 31          ,
INIT_PHA_ENABLE         = 18'h3FFF       ,
INIT_PHA_THRESH         = 12'd1114      , // based on calculations & assumptions, 3-15-02
INIT_ADC_TACQ           = 0           ,
INIT_SAA_LEVEL          = 0           ,
INIT_STROBE             = 0           ,
INIT_VETO_ENABLE        = 18'h3FFF       ,
INIT_VETO_DELAY         = 4'h5          ,
INIT_VETO_WIDTH         = 3'h2          ,
INIT_TRIG_COMMAND       = 16'h0100       ,
INIT_DIAG_STATUS         = 0           ,
GAFE_RESET_DURATION    = 8'hFF         , // number of additional clocks for GAFE reset

/* Trigger Command Parameters */
ZS_ENABLED              = 0           ,
ZS_DISABLED             = 1           ,

TRIG_ACTIVE              = 1           ,
TRIG_INACTIVE             = 0           ,

TRIG_START_BIT           = 3           ,
TRIG_ID_BIT              = 2           ,
TRIG_ZS_BIT              = 1           ,
TRIG_PARITY_BIT          = 0           ,

TRIG_ZS_PATTERN          = 34'h0C         , // ZS pattern is 1100
TRIG_NO_ZS_PATTERN        = 34'h0A         , // All data pattern is 1010

/* Configuration Command Bit Locations */
START_BIT_LOC            = 33,          // start bit location in command register
CMD_TYPE_MSB              = 32,
CMD_TYPE_LSB              = 31,
CMD_TYPE_PARITY_BIT      = 30,
ASIC_SEL_BIT              = 29,          // 0 for GARC, 1 for GAFE
ADDR_MSB                  = 28,
ADDR_LSB                  = 24,
RW_CMD_BIT                = 23,
DATALESS_BIT              = 22,
FUNCT_MSB                 = 21,
FUNCT_LSB                 = 18,
CMD_PARITY_BIT            = 17,
DATA_MSB                  = 16,
DAC_DATA_MSB              = 12,
DATA_LSB                  = 1,
DATA_PARITY_BIT           = 0,

/* GAFE Command Bit Locations */
FE_START_BIT              = 27,
FE_RW_BIT                 = 26,
FE_AD_MSB                 = 25,
FE_AD_LSB                 = 21,
FE_FCT_MSB                = 20,
FE_FCT_LSB                = 17,

```

```

FE_DTA_MSB          = 16,
FE_DTA_LSB          = 1,
FE_DTA_PARITY       = 0,
FE_READBACK_CKS     = 21,
/* Data Mnemonics */
ACTIVE              = 1,
INACTIVE             = 0,
ACTIVE_LO            = 0,
INACTIVE_HI          = 1,
TRUE                = 1,
FALSE               = 0,
LO                  = 0,
HI                  = 1,
CLEAR               = 0,
PRESET              = 1,
COMMAND_ID          = 3'b0,
GAFE_SELECT          = 1,
GARC_SELECT          = 0,
READ_CMD             = 1,
WRITE_CMD             = 0,
ODD_PARITY           = 0,
EVEN_PARITY          = 1,
GAFE_CK_CT           = 5,      // 20 MHz div (5*2) = 2 MHz
GAFE_TRAIL_ZEROES    = 48,
DAC_Config_Mode      = 3'b010, // update DAC input register & load register concurrently
/* Main Command State Machine States (cp_state) */
FILL_SR              = 0,
PARITY_CHECK          = 1,
ASIC_SELECT           = 2,
GARC_READ             = 3,
GAFE_LOAD_CMD         = 4,
GARC_WRITE             = 5,
DAC_COMMAND           = 6,
GAFE_SEND_CMD          = 7,
GAFE_RTND              = 8,
SEND_RB_DATA          = 9,
PARITY_ERROR_ST        = 10,
TRIG_CMD              = 11,
SM_REINIT             = 12,
/* GAFE Commands State Machine States (for acmd_st) */
WAIT_GAFE_CMD         = 0,
SYNC_TO_SCK            = 1,
SEND_GAFE_BITS          = 2,
SEND_GAFE_ZEROES        = 3,
GAFE_ST_SYNC           = 4,
/* DAC Commands State Machine States */
WAIT_DAC_CMD           = 0,
SEND_DAC_BITS           = 1,
DAC_CMD_END             = 2,
DAC_ST_SYNC             = 3,
DAC_READBACK_CMD        = 16'hE000,
DAC_SUB_BIT              = 1'b0,
START_BIT              = 1,      // the leading start bit is logic true
VERSION                = 3'h3; // 3 bit version indicator
*****// GARC Mode Assignments

```

```

assign cmd_error = (cmd_parity_error | data_parity_error);

// Assignment for commandable parity on the return data to the AEM

assign parity_sel      = garc_mode[ 0] ;
assign gafe_parity_sel = garc_mode[ 7] ;
assign veto_en_a       = garc_mode[ 8] ;
assign veto_en_b       = garc_mode[ 9] ;
assign test_mux_ctrl   = garc_mode[10] ;
assign shift_edge_ctrl = garc_mode[11] ;

// TMR Implementation on the HVBS Controls, Supplies A and B

assign hvbs_a_enable = (garc_mode[1] & garc_mode[2]) |
                      (garc_mode[1] & garc_mode[3]) |
                      (garc_mode[2] & garc_mode[3]) ; // TMR Voting Logic for SEU

assign hvbs_b_enable = (garc_mode[4] & garc_mode[5]) |
                      (garc_mode[4] & garc_mode[6]) |
                      (garc_mode[5] & garc_mode[6]) ; // TMR Voting Logic for SEU

//****************************************************************************
// Parity Computations and Bit Aliases

always @ (cmd_sr)
begin

    command_address <= cmd_sr[ADDR_MSB      :ADDR_LSB ]      ;
    command_function <= cmd_sr[FUNCT_MSB     :FUNCT_LSB]      ;

    if ( (cmd_sr == TRIG_ZS_PATTERN) || (cmd_sr == TRIG_NO_ZS_PATTERN) )
        trigger_cmd_id <= ACTIVE;
    else trigger_cmd_id <= INACTIVE;

    config_function <= ({cmd_sr[ADDR_MSB:ADDR_LSB], cmd_sr[FUNCT_MSB:FUNCT_LSB]}) ; // cmd addr &
func no

    config_type_parity <= ~(^cmd_sr[CMD_TYPE_MSB :CMD_TYPE_LSB]) ; // odd parity over 3 bits after
start bit
    cmd_parity         <= ~(^cmd_sr[CMD_TYPE_MSB :FUNCT_LSB])      ; // odd parity over 15 command bits
    data_parity        <= ~(^cmd_sr[DATA_MSB    :DATA_LSB])        ; // odd parity over 16 data bits

end

always @ (readback_word or parity_sel) // continuous odd parity computation for outgoing readback
register
if (parity_sel == ODD_PARITY)
begin
    readback_word[18] <= ~(^readback_word[30:19]); // readback config odd parity
    readback_word[ 0] <= ~(^readback_word[17: 1]); // readback data odd parity
end
else // EVEN PARITY
begin
    readback_word[18] <= (^readback_word[30:19]); // readback config even parity
    readback_word[ 0] <= (^readback_word[17: 1]); // readback data even parity
end

//****************************************************************************
/* External Clock Generator for GAFE interface and DAC interface

The internal system clock is nominally a 20 MHz clock. The interface specification allows
this nominal frequency to be as large as (tbr) 25 MHz. A nominal rate of 2 MHz is selected.

This circuit is accomplished by means of a counter circuit to provide the divide-by-10 function.

```

```

sck_ctrl counts from 0 up to 3 and then starts from 0 again.

*/
task divided_clock_generator;
begin
  /* GAFE sck runs at 5 MHz, or div-by-4 */

  if (sck_ctrl == 3) sck_ctrl <= 0 ;
  else             sck_ctrl <= sck_ctrl + 1;

  /* DAC sck runs at 5 MHz, or div-by-4 */

  if (dac_ck_ctrl == 3) dac_ck_ctrl <= 0 ;
  else             dac_ck_ctrl <= dac_ck_ctrl + 1;

end
endtask

task gafe_clock_generator;
begin
  case (sck_ctrl)
    0: gafe_ck <= LO;
    1: gafe_ck <= LO;

    2: gafe_ck <= HI;
    3: gafe_ck <= HI;

    default: gafe_ck <= LO;
  endcase
end
endtask /* gafe_clock_generator */

task dac_clock_generator;
begin
  case (dac_ck_ctrl)
    0: dac_sck <= LO;
    1: dac_sck <= LO;

    2: dac_sck <= HI;
    3: dac_sck <= HI;

    default: dac_sck <= LO;
  endcase
end
endtask /* dac_clock_generator */

/*********************************************
*/
/*
 Main control block for cmd_processor module (synchronized to posedge ck)

If "reset" is activated, all the registers instantiated in the cmd_processor module are held to their initial reset values. As reset is synchronously released, the input data, cmdd, is clocked into the command shift register, cmd_sr. There are three conditions addressed by the logic below:

(1) - no start bit in cmd_sr[MSB] and not a trigger command --> continue filling shift reg
(2) - no start bit in cmd_sr[MSB] but this is a trigger command --> go to process trigger command
(3) - start bit in cmd_sr[MSB] --> stop shift register, go to state machine
(4) - if trigger is active, then wait for this command to finish --> state machine
*/
always @ (posedge ck or negedge reset_n)
begin /* command processor synchronous block */

```

```

if (!reset_n) initial_cmd_reset;

else // state machine is running
begin

    cmd_proc_state_machine ; // process configuration commands
    trigger_processor ; // task to perform trigger functions: reset, hold, start_pha
    divided_clock_generator ; // Divide-by-10 clock for GAFE and DAC commands
    gafe_command_handler ; // processes commands for read/write to the GAFE ASICs
    gafe_reset_task ; // task to handle the extended GAFE reset signal
    dac_command_handler ; // controls read/writes to MAX5121 DAC

    if ((cmd_sr[START_BIT_LOC] == 0) && (trigger == INACTIVE)) // sr fills until a cmd is detected
    begin
        cmd_sr[START_BIT_LOC:1] <= cmd_sr[START_BIT_LOC-1:0];
        cmd_sr[0] <= cmd;
    end
    end
end /* command processor synchronous block */

//****************************************************************************
/*
Command Processor State Machine

There are two major types of commands to be processed -> Trigger commands and Configuration commands

Trigger commands are evaluated during the first five bits of command data received. If the trigger
pattern is matched, the execution of the trigger commands starts immediately via task
process_trigger_command.

For Configuration commands, there are six cases to be considered: GARC Read, GARC Write, GAFE Read,
GAFE Write, DAC Read, and DAC Write. These are handled by the cmd_proc_state_machine,
the GAFE command handler, and the DAC command handler.

*/
task cmd_proc_state_machine;
begin

    if (PHA_Ready == ACTIVE) strobe <= INACTIVE ; // clear strobe at end-of-conversion

    /* Load the garc_status register */
    garc_status[0] <= lookatme_status ; // status of the look-at-me command
    garc_status[1] <= hvbs_a_enable ;
    garc_status[2] <= hvbs_b_enable ;
    garc_status[3] <= veto_en_a ;
    garc_status[4] <= veto_en_b ;
    garc_status[5] <= zs_status ;

    /* Command Processor State Machine */
    case (cp_state)

        FILL_SR : // Idle while filling the shift register
        begin
            if (cmd_sr[START_BIT_LOC] != START_BIT) live <= ACTIVE ;
            if ((start_pha == INACTIVE) && (data_busy == INACTIVE)) // if not processing
an event
                if ((trigger == ACTIVE) || (trigger_cmd_id == ACTIVE)) // trigger command
                begin
                    cmd_sr <= CLEAR ;
                    live <= INACTIVE ;
                    cp_state <= TRIG_CMD ;
                end
            else
                begin // configuration command
                    if (cmd_sr[START_BIT_LOC] == START_BIT)

```

```

begin
    live      <= INACTIVE      ;
    cp_state <= PARITY_CHECK ;
end
end // configuration command

end /* FILL_SR */

PARITY_CHECK : // Check each parity field is correct
begin
    if ((config_type_parity == cmd_sr[CMD_TYPE_PARITY_BIT]) &&
        (cmd_parity == cmd_sr[CMD_PARITY_BIT] ) &&
        (data_parity == cmd_sr[DATA_PARITY_BIT] ))
        cp_state <= ASIC_SELECT;
    else
        begin
            if (config_type_parity != cmd_sr[CMD_TYPE_PARITY_BIT])
                begin
                    command_register <= cmd_sr[CMD_TYPE_MSB:CMD_PARITY_BIT];
                    cmd_parity_error <= ACTIVE;
                end
            if (cmd_parity != cmd_sr[CMD_PARITY_BIT])
                begin
                    command_register <= cmd_sr[CMD_TYPE_MSB:CMD_PARITY_BIT];
                    cmd_parity_error <= ACTIVE;
                end
            if (data_parity != cmd_sr[DATA_PARITY_BIT])
                begin
                    data_parity_error <= ACTIVE;
                    command_register <= cmd_sr[DATA_MSB:DATA_LSB];
                end
            reject_ct <= reject_ct + 1 ;
            cp_state <= PARITY_ERROR_ST ;
        end
end

ASIC_SELECT : // Determine which ASIC is being commanded and Read vs. Write
begin
    command_ct <= command_ct + 1;
    case ({cmd_sr[ASIC_SEL_BIT], cmd_sr[RW_CMD_BIT]})
        {1'b1, 1'b0}: cp_state <= GAFE_LOAD_CMD;
        {1'b1, 1'b1}: cp_state <= GAFE_LOAD_CMD;
        {1'b0, 1'b0}: begin
            if (config_function == `Use_HV_Nom_Cmd) hv_nom_command
<= ACTIVE;
            if (config_function == `Use_HV_SAA_Cmd) saa_level_command
<= ACTIVE;
            cp_state <= GARC_WRITE ;
        end
        {1'b0, 1'b1}: begin
            if (config_function == `Use_HV_Nom_Cmd) hv_nom_command
<= ACTIVE;
            if (config_function == `Use_HV_SAA_Cmd) saa_level_command
<= ACTIVE;
            cp_state <= GARC_READ ;
        end
        default: cp_state <= GARC_READ ;
    endcase
end

```

```

GARC_READ      : // GARC Readback command
begin
  case ({(hv_nom_command | saa_level_command), dac_command_sent})
    2'b00: begin
      load_garc_readback_register;
      cp_state <= SEND_RB_DATA;
    end

    2'b01: begin
      hv_nom_command     <= INACTIVE;
      saa_level_command <= INACTIVE;
    end

    2'b10: begin
      cp_state <= GARC_READ;
    end

    2'b11: begin
      hv_nom_command     <= INACTIVE;
      saa_level_command <= INACTIVE;
    end
  endcase
end

GAFE_LOAD_CMD : // Load GAFE command in GAFE command register
begin
  gafe_command[FE_START_BIT]           <= START_BIT;
  gafe_command[FE_RW_BIT]              <= cmd_sr[RW_CMD_BIT];
  gafe_command[FE_AD_MSB : FE_AD_LSB ] <= cmd_sr[ADDR_MSB : ADDR_LSB];
  gafe_command[FE_FCT_MSB:FE_FCT_LSB ] <= cmd_sr[FUNCT_MSB:FUNCT_LSB];
  gafe_command[FE_DTA_MSB:FE_DTA_LSB ] <= cmd_sr[DATA_MSB : DATA_LSB];

  send_gafe_command <= ACTIVE;
  cp_state          <= GAFE_SEND_CMD;
end

GAFE_SEND_CMD : // Send GAFE command out to all GAFE ASICs
begin /* GAFE_SEND_CMD */

  /* immediately compute parity bit */

  if (gafe_parity_sel == ODD_PARITY) // nominal mode
    gafe_command[FE_DTA_PARITY] <=
      ~(^{ { cmd_sr[RW_CMD_BIT] , cmd_sr[ADDR_MSB : ADDR_LSB ],
            cmd_sr[FUNCT_MSB:FUNCT_LSB], cmd_sr[DATA_MSB : DATA_LSB ]
          } } );

  else // for GAFE command error testing
    gafe_command[FE_DTA_PARITY] <=
      (^{ { cmd_sr[RW_CMD_BIT] , cmd_sr[ADDR_MSB : ADDR_LSB ],
            cmd_sr[FUNCT_MSB:FUNCT_LSB], cmd_sr[DATA_MSB : DATA_LSB ]
          } } );

  if (gafe_command_sent == 1)
    begin
      send_gafe_command <= INACTIVE;
      if (cmd_sr[RW_CMD_BIT] == READ_CMD)
        begin
          load_garc_readback_register;
          cp_state <= SEND_RB_DATA;
        end
      else
        cp_state <= SM_REINIT; // go to reinitialization state when command
completes
    end

```

```

    end /* GAFE_SEND_CMD */

GARC_WRITE : // Write a configuration command to the GAFC registers
begin
    assign_garc_registers;
    case (config_function)
        `Use_HV_Nom_Cmd: cp_state <= DAC_COMMAND ;
        `Use_HV_SAA_Cmd: cp_state <= DAC_COMMAND ;
        default: cp_state <= SM_REINIT ;
    endcase
end

DAC_COMMAND : begin
    if (dac_command_sent == 1)
        begin
            hv_nom_command <= INACTIVE;
            saa_level_command <= INACTIVE;
            cp_state <= SM_REINIT;
        end
    end

GAFE_RTND : // Get the returned data from the GAFE
begin
    cp_state <= SM_REINIT;
end

SEND_RB_DATA : // Send out readback data bits & clear parity error bits
begin
    rtnd <= readback_word[read_ct];
    if (read_ct > 0) read_ct <= read_ct - 1;
    else
        begin
            parity_error <= CLEAR ;
            cmd_parity_error <= CLEAR ;
            data_parity_error <= CLEAR ;
            diagnostic_status[15:13] <= CLEAR ;
            cp_state <= SM_REINIT ;
        end
    end

PARITY_ERROR_ST : // Parity Error identification
begin
    parity_error <= ACTIVE ;
    diagnostic_status[15] <= ACTIVE ;
    diagnostic_status[14] <= cmd_parity_error ;
    diagnostic_status[13] <= data_parity_error ;
    cp_state <= SM_REINIT ;
end

TRIG_CMD : // Active During Trigger Processing
if ((trigger == TRIG_INACTIVE) && (hold == INACTIVE))
begin
    cp_state <= SM_REINIT ;

```

```

        end

SM_REINIT      : // Reinitialized registers prior to next command
begin
    cmd_sr          <= CLEAR      ;
    read_ct         <= INIT_READ_CT ;
    rtnd            <= CLEAR      ;
    saa_level_command <= CLEAR      ;
    hv_nom_command  <= CLEAR      ;
    send_gafe_command <= CLEAR      ;

    diagnostic_status[12]   <= cmd_error      ;
    diagnostic_status[11:8]  <= diagnostic_status[11:8] + 1 ;
    diagnostic_status[ 7:0]  <= command_ct      ;

    cp_state <= FILL_SR ;
end

default       : // An unexpected, error capture state. State machine is reset.
initial_cmd_reset;

endcase /* main case stmt in cmd_processor machine */
end
endtask /* command_processor state machine */

//****************************************************************************
/* Trigger Functions Task

GAFE Reset is active high GARC reset command.

The "start_pha" signal flag is active coincident with the HOLD pulse. This flag is required
to be active for one clock cycle to initiate the PHA conversion.

*/
task trigger_processor;
begin
    case (trig_st)
        0: // idle state waiting for a trigger command
        begin
            if (trigger_cmd_id) // a trigger command has been received
            begin
                capture_map <= ACTIVE ; // capture the HitMap at the trigger
                start_pha  <= ACTIVE ; // immediately start ADC wake up and conversion
            end
            case ( {cmd_sr[TRIG_ZS_BIT], cmd_sr[TRIG_PARITY_BIT]} )
                2'b00: // valid trigger for ZS
                begin
                    trigger    <= TRIG_ACTIVE ;
                    zs_status <= ZS_ENABLED ;
                    trig_st   <= 1 ;
                end
                2'b01: // bad parity
                begin
                    parity_error <= ACTIVE;
                end
                2'b10: // bad parity
                begin

```

```

        trigger    <= TRIG_ACTIVE ;
        zs_status <= ZS_DISABLED ;
        trig_st    <= 1           ;
    end

    2'b11: // valid trigger for send all PHA
    begin
        parity_error <= ACTIVE;
    end

    default: begin
        parity_error <= ACTIVE;
    end
endcase

end // trigger cmd received

else
begin
    delay_ctr    <= hold_delay      ;
    trigger      <= TRIG_INACTIVE ;
end

end // state 0

1: // trigger to hold delay counter

if (delay_ctr > 0)
    delay_ctr <= delay_ctr - 1;

else
begin
    hold          <= ACTIVE      ;
    capture_map   <= INACTIVE   ;
    trig_st       <= 2           ;
end

2: // start PHA analysis

begin
    trigger      <= TRIG_INACTIVE ;
    trig_st      <= 3           ;
end

3: // PHA conversion time

if (PHA_Ready == ACTIVE) // end of PHA conversion, data ready

begin
    hold          <= INACTIVE   ;
    start_pha    <= INACTIVE   ;

    if (trigger_cmd_id == 0)
        trig_st <= 0;
end

endcase /* trig_st */

end
endtask

//****************************************************************************
/* GAFE Command Handler for GLAST Analog ASICS

```

Whenever the GARC receives a GAFE command, the "send\_gafe\_command" register is toggled to active. Upon completion of sending the GAFE command, the "gafe\_command\_sent" register is toggled active, enabling the main portion of the command state machine to continue to command completion, which will reset the "send\_gafe\_command" and "gafe\_command\_sent" registers. If the GAFE command is a write-type command, the command will be written out and then control returned to the primary

state machine. For GAFE readback-type commands, the data from the GAFE readback will be stored in the "GAFE\_readback" register for access by the main portion of the state machine.

The command word to be sent to the GAFE is stored in [27:0] gafe\_command by the main state machine.

The GAFE command format is:

```
Bit      [27] = start bit = 1
Bit      [26] = write bit = 0, read bit = 1
Bits [25:21] = GAFE Address
Bits [20:17] = GAFE Register Select
Bits [16:1] = GAFE Command Data
Bit      [0] = Odd parity bit
```

\*/

```
task gafe_command_handler;
```

```
begin
```

```
  case (acmd_st)
```

```
    WAIT_GAFE_CMD: // initialize GAFE command state machine registers & wait for flag to send GAFE
command
```

```
    begin
```

```
      gafe_command_sent <= CLEAR;
      gafe_ck           <= LO;
```

```
      if (send_gafe_command == ACTIVE) // remove sending of gafe_cks at reset, 9-16-02
        begin
          gafe_bit         <= FE_START_BIT;
          gafe_readback_reg <= CLEAR;
          acmd_st          <= SYNC_TO_SCK;
        end
    end
```

```
    SYNC_TO_SCK : if (sck_ctr == 0) acmd_st <= SEND_GAFE_BITS;
```

```
    SEND_GAFE_BITS:
```

```
    begin
```

```
      gafe_clock_generator;
```

```
      if ((gafe_reset_n == INACTIVE_HI) && (sck_ctr == 3))
```

```
        begin
```

```
          if (send_gafe_command == ACTIVE)
            gafe_cmdd <= gafe_command[gafe_bit];
          else
            gafe_cmdd <= CLEAR; // during gafe_reset
```

```
          if (gafe_bit == FE_DTA_PARITY)
            begin
              gafe_bit <= FE_READBACK_CKS;
              acmd_st <= SEND_GAFE_ZEROES;
            end
          else gafe_bit <= gafe_bit - 1;
```

```
        end
```

```
      end /* send_gafe_bits */
```

```
    SEND_GAFE_ZEROES: // set GAFE cmd data to 0 and prepear to shift through 27 clocks for
possible readback
```

```
    begin
```

```
      gafe_clock_generator;
```

```

    if (sck_ctrl == 1)

        begin
            gafe_cmdd <= CLEAR;

            if (gafe_readback_reg[16] == 0)
                begin
                    gafe_readback_reg[16:1] <= gafe_readback_reg[15:0];
                    gafe_readback_reg[0]    <= gafe_rtnd;
                end

            if (gafe_bit == FE_DTA_PARITY)
                acmd_st <= GAFE_ST_SYNC;
            else
                gafe_bit <= gafe_bit - 1;

        end

    end /* send_gafe_zeroes */

GAFE_ST_SYNC: // after trailing zeroes are shifted, wait for next command

begin

    gafe_ck          <= LO      ;
    gafe_bit         <= FE_START_BIT ;
    gafe_command_sent <= ACTIVE   ;

    if ((send_gafe_command == INACTIVE) && (reset_n == INACTIVE_HI))
        acmd_st <= WAIT_GAFE_CMD;

end

default: acmd_st <= WAIT_GAFE_CMD;

endcase /* acmd_st */

end
endtask /* gafe_command_handler */

/*****************************************/
// GAFE Reset Task (a digital one-shot: allows for a fixed width pulse on gafe_reset_n)

task gafe_reset_task;
begin
    if (gafe_rst_ctrl == GAFE_RESET_DURATION)
        begin
            gafe_reset_n <= INACTIVE_HI ;
            gafe_rst_ctrl <= CLEAR      ;
        end

    else gafe_rst_ctrl <= gafe_rst_ctrl + 1;

end
endtask /* gafe_reset_task */

/*****************************************/
/* GARC DAC command handler for MAX5121 DAC

This is a separate state machine, driven by the command processor, that handles the
configuration and verification of the external MAX5121 DAC.

The datasheet may be found at http://www.maxim-ic.com

dac_din  is the data sent from the GARC to the DAC
dac_sck  is the serial clock to send dac data in (max rate is 6.66 MHz)
dac_clr_n is the active low clear to reset the DAC
dac_cs_n  is the active low data envelope signal to the DAC (allow 75 ns Setup/Hold)
dac_dout  is the configuration data readback from the DAC

```

```

Format for dac data command:
Bits [15:13] DAC Configuration Control
Bits [12: 1] 12 Bit DAC register command
Bit [ 0 ] sub-bit, always zero

```

DAC update command period minimum is 150 ns, so use 200 ns in design for margin

To write a configuration command to the MAX5121 DAC:

- (1) Start with CS\_N inactive hi, SCK low
- (2) CS\_N goes active low; then wait for a minimum of 40 ns
- (3) Send 16 SCKs to the DAC.
- Data format is C2-C1-C0-D11-D10-D9-D8-D7-D6-D5-D4-D3-D2-D1-D0-S0  
    where C2-C1-C0 == 0-1-0 and the sub-bit S0 == 0
- (4) When SCK is lo, raise CS\_N inactive hi after a minimum delay of 40 ns.

A DAC readback command is accomplished by sending a fixed pattern, DAC\_READBACK\_CMD, to the DAC and capturing the shift register output from the DAC.

\*/

```

task dac_command_handler; // total approx. 180 ck cycles = 9 usec)
begin

    case (dac_st)

        WAIT_DAC_CMD: // waiting for DAC command to be initiated - (1 ck cycle)
        begin
            dac_command_sent <= INACTIVE;
            dac_clr_n <= 1; // DAC is only cleared (reset) at startup
            data_ct <= 15;
            dac_din <= 0;
            dac_sck <= 0;

            if (hv_nom_command | saa_level_command)
                if (dac_ck_ctr == 1)
                    begin

                        dac_cs_n <= 0;
                        dac_st <= SEND_DAC_BITS;

                        if (cmd_sr[RW_CMD_BIT] == READ_CMD) dac_data_wd <= DAC_READBACK_CMD;
                        else
                            if (hv_nom_command == 1) dac_data_wd <= {DAC_Config_Mode, hvbs_dac [11:0],
DACPUBIT};
                                else
                                    dac_data_wd <= {DAC_Config_Mode, saa_level[11:0],
DACPUBIT};

                        if (cmd_sr[RW_CMD_BIT] == WRITE_CMD) dac_din <= 0; // first bit of DAC write
                        else
                            dac_din <= 1; // first bit of DAC readback

                    end
            end /* wait_dac_cmd */

        SEND_DAC_BITS: // send out 16 bits data to DAC (up to 165 ck cycles = 8.25 usec)
        begin
            dac_clock_generator;

            if (dac_ck_ctr == 0)
                dac_din <= dac_data_wd[data_ct];

            else if (dac_ck_ctr == 2)
                begin
                    if (data_ct != 0) data_ct <= data_ct - 1;
                    else dac_st <= DAC_CMD_END;
                end

            else if (dac_ck_ctr == 3) dac_data_readback[data_ct] <= dac_dout;
        end /* send_dac_bits */
    endcase
end

```

```

DAC_CMD_END: // complete cycle of data write to DAC (13 ck cycles = 650 ns)
begin
    dac_clock_generator;

    if (dac_ck_ctr == 1)
        begin
            dac_cs_n <= 1; // latches command word into DAC
            dac_st <= DAC_ST_SYNC;
        end
end

DAC_ST_SYNC: // resynchronize with main state machine (2 ck cycles)
begin
    dac_sck <= CLEAR;
    dac_command_sent <= ACTIVE;
    if (hv_nom_command == INACTIVE) dac_st <= WAIT_DAC_CMD;
end

default : dac_st <= WAIT_DAC_CMD;

endcase /* sck_ctr */

end
endtask /* dac_command_handler */

//****************************************************************************
/*
   GARC configuration register assignment
*/
task assign_garc_registers;

begin

case (config_function)

/* Dataless Commands */
`Reset_Cmd      : reset_cmd      <= ACTIVE ;
`Calib_Cmd       : strobe        <= ACTIVE ;

`Use_HV_Nom_Cmd : hv_nom_command <= ACTIVE ;
`Use_HV_SAA_Cmd : saa_level_command <= ACTIVE ;

`No_Operation_Cmd : nop          <= ~nop ; // toggle the nop bit

/* Commands with Configuration Data */

`HVBS_Cmd        : hvbs_dac      <= cmd_sr[DAC_DATA_MSB:DATA_LSB];
`saa_Cmd         : saa_level      <= cmd_sr[DAC_DATA_MSB:DATA_LSB];

`Hold_Delay_Cmd  : hold_delay    <= cmd_sr[DATA_LSB+6:DATA_LSB] ;
`Veto_Delay_Cmd  : Veto_Delay     <= cmd_sr[4+DATA_LSB:DATA_LSB] ; // remove constants
`Veto_Width_Cmd  : Veto_Width     <= cmd_sr[2+DATA_LSB:DATA_LSB] ;
`HitMap_Deadtime_Cmd: HitMap_Deadtime <= cmd_sr[2+DATA_LSB:DATA_LSB] ;
`HitMap_Delay_Cmd : HitMap_Delay    <= cmd_sr[4+DATA_LSB:DATA_LSB] ;
`HitMap_Width_Cmd : HitMap_Width    <= cmd_sr[3+DATA_LSB:DATA_LSB] ;

`PHA_En0_Reg_Cmd : pha_enable [15: 0] <= cmd_sr[DATA_MSB:DATA_LSB] ;
`VETO_En0_Reg_Cmd : Veto_enable[15: 0] <= cmd_sr[DATA_MSB:DATA_LSB] ;

`PHA_En1_Reg_Cmd : pha_enable [17:16] <= cmd_sr[DATA_LSB+1:DATA_LSB] ;
`VETO_En1_Reg_Cmd : Veto_enable[17:16] <= cmd_sr[DATA_LSB+1:DATA_LSB] ;

`Max_PHA_Cmd     : max_pha      [ 4: 0] <= cmd_sr[DATA_LSB+4:DATA_LSB] ;
`GARC_Mode_Cmd   : garc_mode     <= cmd_sr[DATA_LSB+11:DATA_LSB] ;

`PHA_Thresh_01_Cmd : pha_thresh_01 <= cmd_sr[DATA_LSB+11:DATA_LSB] ;
`PHA_Thresh_02_Cmd : pha_thresh_02 <= cmd_sr[DATA_LSB+11:DATA_LSB] ;
`PHA_Thresh_03_Cmd : pha_thresh_03 <= cmd_sr[DATA_LSB+11:DATA_LSB] ;
`PHA_Thresh_04_Cmd : pha_thresh_04 <= cmd_sr[DATA_LSB+11:DATA_LSB] ;

```

```

`PHA_Thresh_05_Cmd : pha_thresh_05      <= cmd_sr[DATA_LSB+11:DATA_LSB] ;
`PHA_Thresh_06_Cmd : pha_thresh_06      <= cmd_sr[DATA_LSB+11:DATA_LSB] ;
`PHA_Thresh_07_Cmd : pha_thresh_07      <= cmd_sr[DATA_LSB+11:DATA_LSB] ;
`PHA_Thresh_08_Cmd : pha_thresh_08      <= cmd_sr[DATA_LSB+11:DATA_LSB] ;
`PHA_Thresh_09_Cmd : pha_thresh_09      <= cmd_sr[DATA_LSB+11:DATA_LSB] ;
`PHA_Thresh_10_Cmd : pha_thresh_10      <= cmd_sr[DATA_LSB+11:DATA_LSB] ;
`PHA_Thresh_11_Cmd : pha_thresh_11      <= cmd_sr[DATA_LSB+11:DATA_LSB] ;
`PHA_Thresh_12_Cmd : pha_thresh_12      <= cmd_sr[DATA_LSB+11:DATA_LSB] ;
`PHA_Thresh_13_Cmd : pha_thresh_13      <= cmd_sr[DATA_LSB+11:DATA_LSB] ;
`PHA_Thresh_14_Cmd : pha_thresh_14      <= cmd_sr[DATA_LSB+11:DATA_LSB] ;
`PHA_Thresh_15_Cmd : pha_thresh_15      <= cmd_sr[DATA_LSB+11:DATA_LSB] ;
`PHA_Thresh_16_Cmd : pha_thresh_16      <= cmd_sr[DATA_LSB+11:DATA_LSB] ;
`PHA_Thresh_17_Cmd : pha_thresh_17      <= cmd_sr[DATA_LSB+11:DATA_LSB] ;
`PHA_Thresh_18_Cmd : pha_thresh_18      <= cmd_sr[DATA_LSB+11:DATA_LSB] ;
`ADC_TAcq_Cmd       : adc_tacq          <= cmd_sr[DATA_LSB+5 :DATA_LSB] ;

default    : nop <= PRESET;
endcase
end

endtask /* assign_garc_registers */

//****************************************************************************
/*
Task "load_garc_readback_register" loads the value of internal GARC registers into a
shift register for output as configuration return data in the specified format.

Configuration command format (34 bits):

bit      33 : start bit = 1
bits [32:31]: 00 for command
bit      30 : odd parity over bits [32:31]
bit      29 : GARC select (0) or GAFE select (1)
bits [28:24]: Address (GAFE address, GARC function block)
bit      23 : Read (1) or Write (0)
bit      22 : Data/Dataless bit (0 = dataless, 1 = data)
bits [21:18]: Register or function number
bit      17 : odd parity over bits [32:18]
bits [16: 1]: command data field
bit       0 : odd parity over bits [16:1]

Command Readback format (31 bits):

bit      31 : start bit = 1
bit      30 : GAFE/GARC select bit (GARC=0, GAFE=1)
bits [29:25]: GAFE/GARC address
bit      24 : read/write bit (1=read)
bit      23 : Data/Dataless bit
bits [22:19]: function number
bit      18 : odd parity over bits [29:19]
bits [17: 2]: command data, MSB first
bit       1 : error detected in parity
bit       0 : odd parity over bits [17:1]

*/
task load_garc_readback_register;
begin

/* GARC Readback Register Common */
readback_word[31]    <= START_BIT      ; // start bit = 1
readback_word[29:25] <= command_address ; // GARC command address or GAFE chip address
readback_word[24]    <= READ_CMD        ; // readback command indicator bit
readback_word[23]    <= cmd_sr[22]       ; // GARC configuration cmd data/dataless bit
readback_word[22:19] <= command_function ; // GARC command function or GAFE command register
readback_word[ 1]    <= parity_error     ; // command parity error

if (cmd_sr[ASIC_SEL_BIT] == GAFE_SELECT)
begin /* GAFE Select Command */
  readback_word[30]    <= GAFE_SELECT;
  readback_word[17: 2] <= gafe_readback_reg[15:0] ;

```

```

    end

else
begin /* GARC Select Command */
    readback_word[30] <= GARC_SELECT;

    case (config_function)
        `HVBS_Cmd : readback_word[17:2] <= {4'b0, hvbs_dac [11:0]} ;
        `SAA_Cmd : readback_word[17:2] <= {4'b0, saa_level[11:0]} ;
        `Use_HV_Nom_Cmd : readback_word[17:2] <= dac_data_readback[15:0] ;
        `Use_HV_SAA_Cmd : readback_word[17:2] <= dac_data_readback[15:0] ;

        `Hold_Delay_Cmd : readback_word[17:2] <= {9'b0, hold_delay} ;
        `Veto_Delay_Cmd : readback_word[17:2] <= {11'b0, Veto_Delay} ;
        `Veto_Width_Cmd : readback_word[17:2] <= {13'b0, Veto_Width} ;
        `HitMap_Deadtime_Cmd : readback_word[17:2] <= {13'b0, HitMap_Deadtime} ;
        `HitMap_Delay_Cmd : readback_word[17:2] <= {11'b0, HitMap_Delay} ;
        `HitMap_Width_Cmd : readback_word[17:2] <= {12'b0, HitMap_Width} ;

        `PHA_En0_Reg_Cmd : readback_word[17:2] <= pha_enable [15: 0] ;
        `VETO_En0_Reg_Cmd : readback_word[17:2] <= Veto_enable[15: 0] ;
        `PHA_En1_Reg_Cmd : readback_word[17:2] <= {14'b0, pha_enable [17:16]} ;
        `VETO_En1_Reg_Cmd : readback_word[17:2] <= {14'b0, Veto_enable[17:16]} ;
        `Max_PHA_Cmd : readback_word[17:2] <= {11'b0, max_pha[4:0]} ;
        `GARC_Mode_Cmd : readback_word[17:2] <= { 4'b0, garc_mode[11:0]} ;
        `GARC_Status_Cmd : readback_word[17:2] <= {10'b0, garc_status[5:0]} ;
        `Command_Reject_Cmd : readback_word[17:2] <= {8'b0, reject_ct} ;
        `FREE_ID_Cmd : readback_word[17:2] <= {8'b0, FREE_id} ;
        `Version_Cmd : readback_word[17:2] <= {13'b0, VERSION} ;
        `Command_Reg_Cmd : readback_word[17:2] <= command_register ;

        `PHA_Thresh_01_Cmd : readback_word[17:2] <= {4'b0, pha_thresh_01} ;
        `PHA_Thresh_02_Cmd : readback_word[17:2] <= {4'b0, pha_thresh_02} ;
        `PHA_Thresh_03_Cmd : readback_word[17:2] <= {4'b0, pha_thresh_03} ;
        `PHA_Thresh_04_Cmd : readback_word[17:2] <= {4'b0, pha_thresh_04} ;
        `PHA_Thresh_05_Cmd : readback_word[17:2] <= {4'b0, pha_thresh_05} ;
        `PHA_Thresh_06_Cmd : readback_word[17:2] <= {4'b0, pha_thresh_06} ;
        `PHA_Thresh_07_Cmd : readback_word[17:2] <= {4'b0, pha_thresh_07} ;
        `PHA_Thresh_08_Cmd : readback_word[17:2] <= {4'b0, pha_thresh_08} ;
        `PHA_Thresh_09_Cmd : readback_word[17:2] <= {4'b0, pha_thresh_09} ;
        `PHA_Thresh_10_Cmd : readback_word[17:2] <= {4'b0, pha_thresh_10} ;
        `PHA_Thresh_11_Cmd : readback_word[17:2] <= {4'b0, pha_thresh_11} ;
        `PHA_Thresh_12_Cmd : readback_word[17:2] <= {4'b0, pha_thresh_12} ;
        `PHA_Thresh_13_Cmd : readback_word[17:2] <= {4'b0, pha_thresh_13} ;
        `PHA_Thresh_14_Cmd : readback_word[17:2] <= {4'b0, pha_thresh_14} ;
        `PHA_Thresh_15_Cmd : readback_word[17:2] <= {4'b0, pha_thresh_15} ;
        `PHA_Thresh_16_Cmd : readback_word[17:2] <= {4'b0, pha_thresh_16} ;
        `PHA_Thresh_17_Cmd : readback_word[17:2] <= {4'b0, pha_thresh_17} ;
        `PHA_Thresh_18_Cmd : readback_word[17:2] <= {4'b0, pha_thresh_18} ;
        `ADC_TAcq_Cmd : readback_word[17:2] <= {10'b0 , adc_tacq } ;

        `GARC_Diag_Cmd : readback_word[17:2] <= diagnostic_status[15:0] ;
        `No_Operation_Cmd : readback_word[17:2] <= CLEAR ;
    endcase /* config_function */

    default : readback_word[17:2] <= CLEAR;

endcase /* config_function */

end /* GARC select block */

end /* main task block */

endtask /* load_garc_readback_register */

/*********************************************
*/
/* Task "initial_cmd_reset" initializes all garc_cmd_proc registers to predefined value.
   This task is called by the main state machine on the when "reset" is asserted and released
   on the posedge of ck when reset is deasserted.
*/
task initial_cmd_reset;

```

```

begin
    acmd_st          <= WAIT_GAFE_CMD      ;
    adc_tacq         <= INIT_ADC_TACQ     ;
    capture_map      <= INIT_CAPTURE_MAP ;
    command_ct       <= CLEAR           ;
    command_register <= INIT_CMD_REG    ;
    cmd_parity_error <= CLEAR           ;
    cmd_sr           <= INIT_CMD_SR     ;
    cp_state         <= FILL_SR          ;
    dac_din          <= CLEAR           ;
    dac_ck_ctr       <= CLEAR           ;
    dac_sck          <= CLEAR           ;
    dac_clr_n        <= CLEAR           ;
    dac_cs_n         <= INACTIVE_HI      ;
    dac_data_wd[15:13] <= DAC_Config_Mode ;
    dac_data_wd[12: 0] <= CLEAR           ;
    dac_data_readback <= CLEAR           ;
    dac_st           <= WAIT_DAC_CMD    ;
    dac_command_sent <= INACTIVE         ;
    data_ct          <= INIT_DATA_CT   ;
    data_parity_error <= CLEAR           ;
    delay_ctr        <= INIT_HOLD_DELAY  ;
    diagnostic_status <= INIT_DIAG_STATUS ;
    gafe_bit          <= FE_START_BIT    ;
    gafe_ck           <= CLEAR           ;
    gafe_cmdd         <= CLEAR           ;
    gafe_command_sent <= CLEAR           ;
    gafe_readback_reg <= CLEAR           ;
    gafe_reset_n      <= ACTIVE_LO      ;
    gafe_rst_ctrl     <= CLEAR           ;
    gafe_command      <= INIT_GAFE_CMD   ;
    garc_mode         <= INIT_GARC_MODE  ;
    garc_status       <= INIT_GARC_STATUS ;
    HitMap_Deadtime  <= INIT_HITMAP_DEADTIME;
    HitMap_Delay      <= INIT_HITMAP_DELAY ;
    HitMap_Width       <= INIT_HITMAP_WIDTH ;
    hold              <= INIT_HOLD        ;
    hold_delay        <= INIT_HOLD_DELAY  ;
    hvbs_dac          <= INIT_HVBS_DAC    ;
    hv_nom_command    <= CLEAR           ;
    live              <= INACTIVE         ;
    max_phा          <= INIT_MAX_PHA    ;
    nop               <= CLEAR           ;
    readback_word[31:19] <= CLEAR           ;
    readback_word[17: 1] <= CLEAR           ;
    reject_ct         <= CLEAR           ;
    parity_error      <= FALSE            ;
    pha_enable         <= INIT_PHA_ENABLE  ;
    pha_thresh_01      <= INIT_PHA_THRESH  ;
    pha_thresh_02      <= INIT_PHA_THRESH  ;
    pha_thresh_03      <= INIT_PHA_THRESH  ;
    pha_thresh_04      <= INIT_PHA_THRESH  ;
    pha_thresh_05      <= INIT_PHA_THRESH  ;
    pha_thresh_06      <= INIT_PHA_THRESH  ;
    pha_thresh_07      <= INIT_PHA_THRESH  ;
    pha_thresh_08      <= INIT_PHA_THRESH  ;
    pha_thresh_09      <= INIT_PHA_THRESH  ;
    pha_thresh_10      <= INIT_PHA_THRESH  ;
    pha_thresh_11      <= INIT_PHA_THRESH  ;
    pha_thresh_12      <= INIT_PHA_THRESH  ;
    pha_thresh_13      <= INIT_PHA_THRESH  ;
    pha_thresh_14      <= INIT_PHA_THRESH  ;
    pha_thresh_15      <= INIT_PHA_THRESH  ;
    pha_thresh_16      <= INIT_PHA_THRESH  ;
    pha_thresh_17      <= INIT_PHA_THRESH  ;
    pha_thresh_18      <= INIT_PHA_THRESH  ;
    read_ct            <= INIT_READ_CT    ;
    reset_cmd          <= INACTIVE         ; // this now handled correctly in reset logic 9-16-02
    rtnd              <= CLEAR           ;
    saa_level         <= INIT_SAA_LEVEL  ;
    saa_level_command <= CLEAR           ;
    sck_ctrl          <= CLEAR           ;
    send_gafe_command <= CLEAR           ;

```



```

/*********************************************
// Discriminator Logic Port Declarations

module disc_logic      (
    ck,           // 01-in - 20 MHz system clock (disc_logic uses posedge only)
    reset_n,      // 02-in - active low system reset
    Veto_Delay,   // 03-in - register to control the delay of the VETO pulses
    Veto_Width,   // 04-in - register to control the width of the VETO pulses
    HitMap_Deadtime, // 05-in - register to control the stretch on the trailing edge of HitMap
    HitMap_Delay,  // 06-in - register to control the delay of the HitMap
    HitMap_Width,  // 07-in - register to control the width of the HitMap
    Veto_Enable,   // 08-in - register to control VETO output (enable = 1, disable = 0)
    disc_in,       // 09-in - 18 bit wide bus of discriminators input from the GAFE
    capture_map,  // 10-in - active high semaphore to initiate capture of the HitMap
    HitMapLch     // 11-in - active high semaphore to indicate event_data has the HitMap

    Veto_AEM,      // 12-out - the output VETO pulses to the AEM
    HitMaprdy,    // 13-out - active high semaphore to indicate that the HitMap is stable
    HitMap,        // 14-out - 18 bit output bus that is the HitMap
    HitMap_Test   // 15-out - HitMap Test output for test with oscilloscope
);

/*********************************************
/*
|-----
| -----
| | GLAST ACD Readout Controller ASIC (GARC)
| | Discriminator Logic
| |
| | Project : Gamma-Large Area Space Telescope (GLAST)
| | Anti-Coincidence Detector (ACD)
| | GLAST ACD Readout Controller (GARC) ASIC
| |
| | Written      : D. Sheppard
| | Module       : disc_logic.v
| | Original     : 10-09-01
| | Rewritten    : 02-06-02
| | Updated      : 03-13-02
| | Testbench    : testbench_disc_logic.v
| | Location     : ~/GLAST/garc_logic/discriminator
| | File         : disc_logic.v
| | Version      : 2.0
| |
| ----- VERILOG-XL 3.10.p001
| -----
|
| This software is property of the National Aeronautics and Space
| Administration. Unauthorized use or duplication of this
| software is strictly prohibited. Authorized users are subject
| to the following restrictions:
|
| 1. Neither the author, their corporation, nor NASA is
|    responsible for any consequences of the use of this software.
| 2. The origin of this software must not be misrepresented either
|    by explicit claim or by omission.
| 3. Altered versions of this software must be plainly marked
|    as such.
| 4. This notice may not be removed or altered.
|
| -----
| *****
| Version History & Changes Made
|
| 03-13-02 : Change reset to active low polarity
| 03-07-02 : Add HitMap test signal output
| 03-06-02 : Update digital delay line register for HitMap to 48 FF.
|            Update HitMap logic to address 0 and 1 problem

```

```

02-19-02 : Change size of digital delay line registers
02-18-02 : Change HitMap logic to allow Deadtime stretch.
02-15-02 : Change Veto_Delay from 4 bits to 5 to match ICD. Change
           HitMap logic to match Bob Hartman's latest changes, which
           now are in the ICD. Move deglitch to after shift register
           tap. HitMap is now not deglitched.
02-06-02 : Added in BBob's casestat.v for the VW_state case statement
           to fix width differences in Veto due to state machine path
           differences

```

Version 2.0: This update strips out all of the pre-SLAC PDR logic that was detailed in the Level 3 Requirements on down and replaces that with the newly proposed logic (1/02)

Version 1.0: Comments are contained in archive version

---

This module, disc\_logic.v, is the logic for the discriminator outputs and the hit map. It resides in the GLAST ACD digital ASIC (GARC) and has the following inputs and outputs:

**Inputs:**

ck, a 20 MHz clock originating from the TEM. Only the positive (rising) edge of this clock is used

reset, the module reset signal received from the command block. This is asynchronously asserted and synchronously de-asserted

Veto\_Enable, a control register that either selectively enables (1) or disables (0) each Veto output.

Veto\_Delay, a 5 bit control register that sets the delay on the Veto  
Veto\_Width, a 3 bit control register that sets the width of the Veto

HitMap\_Deadtime, a 3 bit control register that sets the stretch on trailing edge of hitmap  
HitMap\_Delay, a 5 bit control register that sets the delay on the hit map  
HitMap\_Width, a 4 bit control register that sets the width of the hit map

disc\_in[18:0], a group of 18 Veto discriminators (0-17) and the HLD discriminator (18). Each discriminator is handled identically, although HLD (CNO) is not part of the HitMap.

capture\_map, active high semaphore input (one clock cycle) indicating that the hit map is to be captured at the next clock

HitMaprch, active high input semaphore indicating the event\_data module has recorded the hit map for transmission and the disc logic state machine can go back to an idle state to wait for the next capture cycle

**Outputs:**

Veto\_AEM[18:0], a deglitched, delayed, and stretched version of the input discriminators.

HitMap[18:0], a second version of the input discriminators. Also deglitched, delayed, and stretched.

HitMaprdy, active high output indicating that the HitMap is ready for the event\_data module

\*\* Note that the hit map contents are latched in the telemetry processor, which also operates on the posedge of ck

---



---

/\*  
/\* Discriminator Logic Module

This module is the circuit for one individual channel of Veto and HitMap logic.  
This circuit is called 18 times for the Veto signals and 1 time for the CNO.  
Modules are instantiated in the main disc\_logic module.

For this circuit:

```

(1) One delay unit, D(n+1) - D(n) is one clock cycle, or 50 ns

(2) Discriminator Input is zero clock delays, i.e., signal D0

(3) Includes: Deglitch circuit (D0 input, D3 output)
    Digital delay line (shift register to D64, output of deglitch ckt is input)
    Veto Delay multiplexer (a series of taps on delay line)
    Hit Map multiplexer (a second series of taps on the delay line)
    Veto Stretch machine (logic to set the width of the output Veto)
    HitMap Stretch machine (logic to set the width of the output HitMap)

*/
// Discriminator Logic I/O Port Definitions

input      ck          ; // 20 MHz system clock input
input      reset_n     ; // global module reset from garc_cmd_proc
input [ 4:0] Veto_Delay   ; // Veto Delay is 5 bits
input [ 2:0] Veto_Width    ; // Veto Width is 3 bits
input [ 2:0] HitMap_Deadtime ; // Hit Map Deadtime is 3 bits
input [ 4:0] HitMap_Delay    ; // Hit Map Delay is 5 bits
input [ 3:0] HitMap_Width    ; // Hit Map Width is 4 bits
input [17:0] Veto_Enable     ; // 18 bit Veto discriminator enables (enable = 1, disable = 0)
input [18:0] disc_in       ; // discriminators input from GAFE ASICs
input      capture_map     ; // input flag to capture map data
input      HitMapLch       ; // input flag to indicate event_data module has latched the HitMap

output     HitMaprdy      ; // output to indicate to event_data module that HitMap is ready
output [18:0] Veto_AEM      ; // Veto output to AEM
output [17:0] HitMap       ; // Hit Map output to event data module
output      HitMap_Test     ; // HitMap output for test on scope

/*
// Discriminator Logic Registers

reg      HitMaprdy      ; // semaphore to indicate Hit Map is ready for capture by data module
reg [ 1:0] disc_st       ; // Main logic block state variable
wire [18:0] Veto_AEM      ; // registers for Veto outputs
wire [17:0] HitMap_tap     ; // HitMap multiplexer outputs
wire      Dummy_tap       ; // Dummy tap for CNO since this is not included in the HitMap
wire      HitMap_Test      ; // HitMap output for test on scope
reg [17:0] HitMap       ; // CNO is channel 18 and CNO HitMap is not output from the module,
                           // leaving HitMap[18] as a dummy variable

/*
// Test Output Assignment

assign HitMap_Test = HitMap_tap[00] ; // select channel 0 for test output

/*
// Local Parameter Declarations

parameter
  CLEAR      = 0      , // represents reset
  ACTIVE     = 1      , // represents active high
  INACTIVE   = 0      , // represents inactive low
  TIE_LOW    = 1'b0   , // represents a connection to logic 0
  TIE_HIGH   = 1'b1   , // represents a connection to logic 1

  // state machine states
  IDLE      = 0      , // state machine idle, waiting for data state
  HITMAP_READY = 1    , // state machine HitMap ready state
  HITMAP_LATCHED = 2  , // state machine HitMap latched state
  WAIT_RECOVERY = 3   , // state machine wait for end of event data

  VERSION    = 2      ; // disc_logic module version

/*
// HitMap Capture State Machine (disc_st is the state variable)

```

```

always @ (posedge ck or negedge reset_n)
begin
    if (!reset_n)
        begin
            HitMaprdy <= INACTIVE ; // at reset, the HitMap is not ready
            HitMap     <= INACTIVE ; // at reset, the HitMap is cleared
            disc_st   <= IDLE      ; // at reset, the state machine is forced to the IDLE state
        end
    else
        case (disc_st)
            IDLE: // state machine is idle until semaphore to capture a HitMap is received
            begin
                HitMaprdy <= INACTIVE; // initialize HitMap-is-ready semaphore

                if (capture_map) // signal to latch the Hit Map
                begin
                    HitMap[17:0] <= HitMap_tap[17:0] ; // note that CNO, 18, is not latched into HitMap
                    disc_st     <= HITMAP_READY      ;
                end
            end

            HITMAP_READY:
            begin
                HitMaprdy <= ACTIVE       ; // indicate Hit Map is ready to send as event data
                disc_st     <= HITMAP_LATCHED;
            end

            HITMAP_LATCHED:
            begin
                if (HitMaplch) // event data module has taken the data
                begin
                    HitMaprdy <= INACTIVE      ;
                    disc_st     <= WAIT_RECOVERY;
                end
            end

            WAIT_RECOVERY:
            begin
                if (!capture_map) // this should always be the case if event_data is
functioning
                    disc_st <= IDLE;

                default: begin
                    HitMaprdy <= INACTIVE ;
                    disc_st     <= IDLE      ;
                end
            end
        endcase /* disc_st case */
    end
end

//****************************************************************************
// Instantiation of Individual Discriminator Channel Logic

/* Channels 0 - 17 */
disc_ch DC00 (ck, reset_n, disc_in[00], Veto_AEM[00], Veto_Enable[00],
              HitMap_tap[00],Veto_Delay,Veto_Width,HitMap_Delay,HitMap_Width, HitMap_Deadtime);

disc_ch DC01 (ck, reset_n, disc_in[01], Veto_AEM[01], Veto_Enable[01],
              HitMap_tap[01],Veto_Delay,Veto_Width,HitMap_Delay,HitMap_Width, HitMap_Deadtime);

disc_ch DC02 (ck, reset_n, disc_in[02], Veto_AEM[02], Veto_Enable[02],
              HitMap_tap[02],Veto_Delay,Veto_Width,HitMap_Delay,HitMap_Width, HitMap_Deadtime);

disc_ch DC03 (ck, reset_n, disc_in[03], Veto_AEM[03], Veto_Enable[03],
              HitMap_tap[03],Veto_Delay,Veto_Width,HitMap_Delay,HitMap_Width, HitMap_Deadtime);

disc_ch DC04 (ck, reset_n, disc_in[04], Veto_AEM[04], Veto_Enable[04],
              HitMap_tap[04],Veto_Delay,Veto_Width,HitMap_Delay,HitMap_Width, HitMap_Deadtime);

disc_ch DC05 (ck, reset_n, disc_in[05], Veto_AEM[05], Veto_Enable[05],
              HitMap_tap[05],Veto_Delay,Veto_Width,HitMap_Delay,HitMap_Width, HitMap_Deadtime);

disc_ch DC06 (ck, reset_n, disc_in[06], Veto_AEM[06], Veto_Enable[06],
              HitMap_tap[06],Veto_Delay,Veto_Width,HitMap_Delay,HitMap_Width, HitMap_Deadtime);

```

```

        HitMap_tap[06],Veto_Delay,Veto_Width,HitMap_Delay,HitMap_Width, HitMap_Deadtime);

disc_ch DC07 (ck, reset_n, disc_in[07], Veto_AEM[07], Veto_Enable[07],
               HitMap_tap[07],Veto_Delay,Veto_Width,HitMap_Delay,HitMap_Width, HitMap_Deadtime);

disc_ch DC08 (ck, reset_n, disc_in[08], Veto_AEM[08], Veto_Enable[08],
               HitMap_tap[08],Veto_Delay,Veto_Width,HitMap_Delay,HitMap_Width, HitMap_Deadtime);

disc_ch DC09 (ck, reset_n, disc_in[09], Veto_AEM[09], Veto_Enable[09],
               HitMap_tap[09],Veto_Delay,Veto_Width,HitMap_Delay,HitMap_Width, HitMap_Deadtime);

disc_ch DC10 (ck, reset_n, disc_in[10], Veto_AEM[10], Veto_Enable[10],
               HitMap_tap[10],Veto_Delay,Veto_Width,HitMap_Delay,HitMap_Width, HitMap_Deadtime);

disc_ch DC11 (ck, reset_n, disc_in[11], Veto_AEM[11], Veto_Enable[11],
               HitMap_tap[11],Veto_Delay,Veto_Width,HitMap_Delay,HitMap_Width, HitMap_Deadtime);

disc_ch DC12 (ck, reset_n, disc_in[12], Veto_AEM[12], Veto_Enable[12],
               HitMap_tap[12],Veto_Delay,Veto_Width,HitMap_Delay,HitMap_Width, HitMap_Deadtime);

disc_ch DC13 (ck, reset_n, disc_in[13], Veto_AEM[13], Veto_Enable[13],
               HitMap_tap[13],Veto_Delay,Veto_Width,HitMap_Delay,HitMap_Width, HitMap_Deadtime);

disc_ch DC14 (ck, reset_n, disc_in[14], Veto_AEM[14], Veto_Enable[14],
               HitMap_tap[14],Veto_Delay,Veto_Width,HitMap_Delay,HitMap_Width, HitMap_Deadtime);

disc_ch DC15 (ck, reset_n, disc_in[15], Veto_AEM[15], Veto_Enable[15],
               HitMap_tap[15],Veto_Delay,Veto_Width,HitMap_Delay,HitMap_Width, HitMap_Deadtime);

disc_ch DC16 (ck, reset_n, disc_in[16], Veto_AEM[16], Veto_Enable[16],
               HitMap_tap[16],Veto_Delay,Veto_Width,HitMap_Delay,HitMap_Width, HitMap_Deadtime);

disc_ch DC17 (ck, reset_n, disc_in[17], Veto_AEM[17], Veto_Enable[17],
               HitMap_tap[17],Veto_Delay,Veto_Width,HitMap_Delay,HitMap_Width, HitMap_Deadtime);

/* Channel for CNO */
disc_ch DC18 (ck, reset_n, disc_in[18], Veto_AEM[18], TIE_HIGH,
               Dummy_tap      ,Veto_Delay,Veto_Width,HitMap_Delay,HitMap_Width, HitMap_Deadtime);

/*****************************************/
endmodule /* disc_logic */

/*****************************************/
// Module disc_ch is the logic for each individual discriminator channel, called by disc_logic

module disc_ch (ck
                  , // in port 1
                  reset_n
                  , // in port 2
                  disc_in
                  , // in port 3
                  Veto
                  , // out port 4
                  Veto_Enable
                  , // in port 5
                  HitMap
                  , // out port 6
                  Veto_Delay
                  , // in port 7
                  Veto_Width
                  , // in port 8
                  HitMap_Delay
                  , // in port 9
                  HitMap_Width
                  , // in port 10
                  HitMap_Deadtime // in port 11
);
;

input      ck          ; // 20 MHz system clock
input      reset_n     ; // system logic reset command (active low)
input [4:0] Veto_Delay ; // delay for Veto signal output
input [2:0] Veto_Width  ; // width for Veto signal output
input [4:0] HitMap_Delay ; // delay for HitMap signal output
input [3:0] HitMap_Width ; // width command for HitMap output
input [2:0] HitMap_Deadtime ; // stretch command for HitMap output
input      disc_in     ; // discriminator input from differential receiver on pad frame
input      Veto_Enable  ; // the Veto enable command bit (1 = enable, 0 = disable)

output     Veto        ; // the Veto signal output to the differential drivers on the pad
frame
output     HitMap      ; // the muxed HitMap signal output to the top level for latch at HOLD

```

```

/*************************************************/
parameter
  SR_MSB    = 48          , // most significant bit of the digital delay line shift register
  SR_LSB    = 1          , // least significant bit of the digital delay line shift register
  CLEAR     = 0          , // clear is defined as logic low
  ACTIVE    = 1          , // active is defined as logic high
  INACTIVE   = 0          , // inactive is defined as logic low
  VERSION   = 2          ; // version number

/*************************************************/
/* local task register declarations */

reg [SR_MSB:SR_LSB] D      ; // digital delay line
reg [ 2:0] Veto_Ctr        ; // used to count the width of the Veto pulse
reg [ 3:0] HitMap_Ctr      ; // used to count the width of the HitMap pulse
reg [ 2:0] HitMap_Stretch   ; // used to count the stretch for HitMap Deadtime

reg      Veto             ; // Veto output
reg      HitMap           ; // HitMap output

reg [ 1:0] VW_state        ; // Veto width state variable
reg [ 1:0] HW_state        ; // HitMap width state variable

reg      Veto_Tap          ; // Veto Delay multiplexer tap
reg      HitMap_Tap         ; // HitMap Delay multiplexer tap

reg      V_D0              ; // First Veto deglitch FF
reg      V_D1              ; // Second Veto deglitch FF
wire    Veto_Deglitched    ; // Combinatorial logic for V_D0 & V_D1 deglitch

/*************************************************/
// Combinatorial Assignments for Delay Multiplexers (Veto and HitMap)
// Note that Jitter is [min:max] of [0 ns: 50ns]. Minimum fixed delay + 50 ns = Maximum fixed delay

always @ (Veto_Delay or D) Veto_Tap <= D[Veto_Delay + SR_LSB] ; ;
always @ (HitMap_Delay or D) HitMap_Tap <= D[SR_MSB - 31 + HitMap_Delay] ;

/*************************************************/
assign Veto_Deglitched = (V_D0 & V_D1) ; // assignment to accomplish post-tap Veto deglitch

always @ (posedge ck or negedge reset_n)
begin
  if (!reset_n) initialize_disc_channel;

  else
    begin /* else block */
      /* Digital Delay Line Shift Register */
      D[SR_MSB:SR_LSB+1] <= D[SR_MSB-1:SR_LSB] ;
      D[SR_LSB]           <= disc_in           ;
    end
end

/* Veto_Deglitched Circuit for above assignment */
V_D0 <= Veto_Tap ;
V_D1 <= V_D0      ;

/* ***** Veto State Machine **** */
case (VW_state)

  0: // idle waiting for VETO pulse
    if (Veto_Deglitched == ACTIVE)
      begin
        Veto_Ctr <= Veto_Width      ;
      end

```

```

        Veto      <= Veto_Enable  ;
        VW_state <= 1           ;
    end

1: // first VETO edge

    if (Veto_Deglitched == INACTIVE)
        if (Veto_Ctr == 0)
            begin
                Veto <= INACTIVE;
                VW_state <= 0;
            end
        else
            begin
                Veto_Ctr <= Veto_Ctr - 1;
                VW_state <= 2;
            end
    else          // (Veto_Deglitched ACTIVE)
        if (Veto_Ctr == 0)
            begin
                Veto <= INACTIVE;
                VW_state <= 3; // go wait for end of pulse
            end
        else
            Veto_Ctr <= Veto_Ctr - 1;

2: // look for second VETO edge

    if (Veto_Deglitched == INACTIVE)
        if (Veto_Ctr == 0)
            begin
                Veto <= INACTIVE;
                VW_state <= 0;
            end
        else
            Veto_Ctr <= Veto_Ctr - 1;
    else          // (Veto_Deglitched ACTIVE on second edge
        begin
            Veto_Ctr <= Veto_Width;
            VW_state <= 1;
        end

3: // wait for end of pulse

    if (Veto_Deglitched == INACTIVE)
        VW_state <= 0;

endcase /* VW state case */

/* ***** HitMap State Machine ***** */
case (HW_state)

0: // HitMap Idle State

begin

    HitMap_Ctr      <= HitMap_Width     ; // initialize width counter
    HitMap_Stretch <= HitMap_Deadtime ; // initialize stretch counter

    if (HitMap_Tap == ACTIVE)
        begin
            HitMap   <= Veto_Enable; // if Veto is disabled, HitMap becomes inactive
            HW_state <= 1           ;
        end
    else
        HitMap <= INACTIVE         ; // initialize HitMap
end // state 0

```

```

1: // HitMap Width determination

    if (HitMap_Ctr == 0)
        HW_state <= 2 ;
    else
        HitMap_Ctr <= HitMap_Ctr - 1 ;

2: // HitMap Wait for Inactive
    if (HitMap_Tap == INACTIVE)
        HW_state <= 3;

3: // HitMap Stretch at trailing edge of delayed pulse
    if (HitMap_Stretch == 0)
        begin
            HitMap <= INACTIVE;
            HW_state <= 0;
        end
    else HitMap_Stretch <= HitMap_Stretch - 1;

    default: begin
        HitMap <= INACTIVE ;
        HitMap_Ctr <= HitMap_Width ;
        HitMap_Stretch <= HitMap_Deadtime ;
        HW_state <= 0 ;
    end /* default case */

    endcase /* HW state case */
end /* else block */
end /* always block */

// ****
// Task initialize_disc_channel provides initialization at assertion of reset_n

task initialize_disc_channel;

begin
    D <= CLEAR ;
    V_D0 <= CLEAR ;
    V_D1 <= CLEAR ;
    Veto <= CLEAR ;
    HitMap <= CLEAR ;
    VW_state <= CLEAR ;
    HW_state <= CLEAR ;
    Veto_Ctr <= CLEAR ;
    HitMap_Ctr <= CLEAR ;
    HitMap_Stretch <= CLEAR ;
end

endtask /* initialize_disc_channel */

// ****
endmodule /* disc channel */

```

```

/*
// Event Data module ports

module event_data (
    ck      , // in : 20 MHz system clock
    cmd_error , // in : CMD parity error from garc_cmd_proc module
    HitMap   , // in : hit map from disc_logic
    HitMaprdy , // in : hit map ready signal from disc_logic
    max_pha  , // in : maximum no. of PHA words to send from garc_cmd_proc
    parity_sel , // in : 0 = odd parity select, 1 = even parity select
    reset_n   , // in : system reset, active low
    pha      , // in : multiplexed PHA values
    pha_enable , // in : PHA enable to send bits
    PHA_Ready , // in : pha ready signal from pha_logic (reset by phasel=='1F)
    zs_map   , // in : map of PHA values above threshold from pha_logic module
    zs_status , // in : 0 = zs enabled; 1 = send all pha values
    data_busy , // out: data busy signal to hold off incoming commands
    edata    , // out: event data output
    HitMaplch , // out: hit map latched signal to disc_logic for hit map reset
    phasel   // out: mux selectors of PHA values
);

/*
/*
-----+
-----+
| | GLAST ACD Readout Controller ASIC (GARC) Event Data Logic |
| |
| | Project : Gamma-Large Area Space Telescope (GLAST)           |
| | Anti-Coincidence Detector (ACD)                            |
| | GLAST ACD Readout Controller (GARC) ASIC                  |
| |
| | Written      : D. Sheppard                                |
| | Module       : event_data.v                               |
| | Original     : 10-09-01                                  |
| | Updated      : 10-22-02                                  |
| | Version      : 2.0                                      |
| |
|-----+ VERILOG-XL 3.10.p001
-----+

```

\*\*\*\*\*  
Version History & Changes Made

#### Version 2.0:

- 10-22-02: Fix PHA parity calculation. This error created by updated event format and not caught in original testing. Remove the "more\_pha" register since it is now unneeded.
- 10-09-02: Update event data format bit 14 to meet with new SLAC requirements. Now, the meaning of bit 14 is as follows:  
 0: this is the last bit of transmission with no PHA data following  
 1: this bit is followed by a PHA value  
  
 Also, update so that the Max\_PHA and pha\_enable functions are disabled in the non-ZS mode
- 09-30-02: Update pha\_send to correct readout problem of NO\_ZS mode still using the ZS comparator outputs. For the ZS\_DISABLED mode, PHA is output for each channel where PHA is enabled and <= Max\_PHA. For the ZS\_ENABLED mode, PHA is output for each channel above ZS thresold with PHA enabled and <= Max\_PHA.
- 09-25-02: Change the pha\_enable function so that it works in both the ZS and non-ZS modes.
- 09-16-02: Change zs\_map format so that only those PHAs that are sent appear in the zs\_map\_out.

Version 1.0:

```
03-18-02: Remove 5 bit pha_ct from the serial event data stream
03-15-02: Add data_busy feature
03-13-02: Change reset polarity to active low
02-19-02: Change start condition to allow for PHA to be completed due to change
           in the disc_logic.v module
02-13-02: add more comments to increase readability
02-12-02: updated to fix PHA readout offset condition caused by phasel
           missing first value
02-06-02: changed state machine to not wait for PHA_Ready, which is
           gone before HitMap is ready. This was causing the state
           machine to not execute at certain times.
```

\*\*\*\*\*

This module receives data from the hit map in disc\_logic, and PHA data  
from the pha module, pha\_logic.v. Event data is sent to the AEM in  
the following format:

Event Data:

```
[      0] MSB = Start Bit = 1
[18: 1] Hit Map Bits (18 VETOs, no CNO)
[36:19] ZS Bits (18 VETOs, no CNO)
[      37] Cmd/Data Error Detect
[      38] Parity (odd) over bits [37:0]
[43:39] # of PHA values to follow
[58:44] PHA word #1 (each PHA word is 15 bits)
[nn:mm] each 15 bit pha word
```

PHA Word Format:

Each PHA word is 15 bits in duration. These bits are as follows:  
[ 14]: 1 if a PHA is to follow, 0 if no more PHA (end of data)  
[ 13]: pha range bit (1 = hi range, 0 = lo range)  
[12:1]: PHA word from the ADC (MSB at 12)  
[ 0]: odd parity over PHA Word bits [14:1]

\*/

\*\*\*\*\*  
// Event Data module I/O Declarations

```
/* event_data Inputs */
input      ck          ; // the 20 MHz system clock
input      cmd_error   ; // a bit from garc_cmd_proc indicating if a cmd error
input [17:0] HitMap     ; // the 18 bit list of the VETO HitMap
input      HitMaprdy   ; // active hi semaphore indicating disc_logic has the HitMap ready
input [ 4:0] max_pha    ; // 5 bits indicating the maximum allowable number of PHA values to send
input      parity_sel   ; // parity control for output data, 0 = odd, 1 = even
input [12:0] pha         ; // sel bit + 12 bits PHA
input [17:0] pha_enable  ; // individual control for PHA channels, 1 = enable, 0 = disable
input      PHA_Ready    ; // active hi semaphore from pha_logic indicating end-of-conversion
input      reset_n      ; // system logic reset, active low
input [17:0] zs_map     ; // 18 bit map indicating those PHA values greater than ZS threshold
input      zs_status    ; // semaphore controlling zero-suppression function (0 = enabled, 1= disabled)

/* event_data Outputs */
output     data_busy    ; // event_data busy indicator
output     edata        ; // serial event data output
output     HitMaplch   ; // active hi semaphore indicating to disc_logic that HitMap has been sent
output [ 4:0] phasel    ; // 5 bit control on the PHA word multiplexer in pha_logic
```

\*\*\*\*\*  
// Register and Wire Declarations

```

reg [ 4:0] bit_ctr      ; // counter used to count the number of bits for transmission
reg      data_busy      ; // event_data busy indicator
reg      edata          ; // the serial data output
reg [ 3:0] evt_st       ; // the state variable used for the event data state machine
reg      HitMapLch      ; // flag used to signal to disc_logic that the HitMap has been taken
reg      parity          ; // event data parity bit
reg      pha_parity      ; // PHA word parity bit
reg [ 4:0] pha_ct       ; // counter of the number of PHA words to be sent
reg [ 4:0] phasel        ; // select the output of the PHA multiplexer in pha_logic module
reg [17:0] pha_send      ; // pha-to-send map for data out to AEM
wire     MUXEND          ; // variable to determine if PHA select mux is at last value

//****************************************************************************
// Local Parameters for event_data

parameter

/* State Machine State Variables (evt_st) */
WAIT_MAP      = 0      , // state evt_st = 0
COUNT_PHA     = 1      , // state evt_st = 1
SEND_MAP      = 2      , // state evt_st = 2
SEND_ZS_DATA  = 3      , // state evt_st = 3
SEND_CMD_ERR  = 4      , // state evt_st = 4
SEND_PARITY   = 5      , // state evt_st = 5
SEND_PHA      = 6      , // state evt_st = 6
SM_REINIT    = 7      , // state evt_st = 7
WAIT_FOR_IDLE = 8      , // state evt_st = 8
                           // evt_st states 9-15 unused

/* Module Variable Parameters */
ODD_PARITY    = 0      , // defines control bit for ODD parity event data readback
EVEN_PARITY   = 1      , // defines control bit for EVEN parity event data readback

PHA_LIMIT     = 18     , // maximum number of PHA events that will ever be sent in a stream
LAST_PHA      = 5'h1F  , // the last value of the PHA select multiplexer
PHA_MSB       = 11     , // the most significant bit of the PHA word

ZS_ENABLED    = 0      , // if zs_status = 0, then zero suppression is enabled
ZS_DISABLED   = 1      , // if zs_status = 1, then zero suppression is disabled

START_BIT     = 1'b1   , // the event data start bit is a logical 1

INACTIVE      = 0      , // defines the inactive semaphore state to be 0
ACTIVE        = 1      , // defines the active semaphore state to be 1

CLEAR         = 0      , // defines a register clear as a reset

/* Initial Reset Parameters */
INIT_DATA_BUSY = INACTIVE , // at reset, data_busy is inactive low
INIT_EDATA     = CLEAR   , // at reset, event data is low
INIT_PHASEL    = 5'h1F   , // at reset, the PHA select multiplexer is at the end
INIT_HITMAPLCH = INACTIVE , // at reset, the HitMap latched semaphore is inactive
INIT_STATE     = WAIT_MAP , // at reset, the state machine goes to the idle state
INIT_PHACT    = CLEAR   , // at reset, the pha word count register is cleared
INIT_BITCTR   = CLEAR   , // at reset, the pha bit count register is cleared
INIT_PHAMAP   = CLEAR   , // at reset, the pha map register is cleared

VERSION        = 0      ; // Version is Rev in this prototype

//****************************************************************************
// PHA Select Multiplexer Combinatorial Test for End-of-Data Pattern (e.g., phasel = 11111)
// if phasel == 'h1F, MUXEND = 1; else MUXEND = 0;

assign MUXEND = &phasel; // if phasel = 5'h1F, then mux is at the last value

//****************************************************************************
/* Commandable Parity Computation and PHA word count status

```

```

Combinatorial Parity Assignment for Event Data Output

"parity" is the variable used for calculating the odd parity over the first 37 bits
of data to be transmitted serial out of this module on edata.

"pha_parity" is the variable used for each 15 bit PHA word. This value is the odd
parity calculated over the first 14 bits of each PHA value.

"parity_sel" allows for commandable parity at the output.

*/
always @ (parity_sel or HitMap or zs_map or cmd_error)
  if (parity_sel == ODD_PARITY)
    parity <= ~(^{HitMap[17:0], zs_map[17:0], cmd_error}); // odd parity, first 37 bits
  else
    parity <= (^{HitMap[17:0], zs_map[17:0], cmd_error}); // even parity, first 37 bits

always @ (parity_sel or pha)
  if (parity_sel == ODD_PARITY)
    pha_parity <= ~(^{1'b1, pha[12:0]}); // odd parity over each pha word
  else
    pha_parity <= (^{1'b1, pha[12:0]}); // even parity over each pha word

//****************************************************************************
/*
function Next_PHA

Next_PHA looks at next active PHA from greater than present position of ZS map. If
zs_status == 1, then no zero suppression is being used and the value for phasel just
increments to the next phavalue that is enabled. The present value of the multiplexer,
phasel, is used to determine what values have already been read out. At reset or
completion of a readout, the value is set to the maximum, 31. At the start of the sequence
immediately after initialization, phasel is set to a value of MUXEND, 'h1F.

*/
function [4:0] Next_PHA ; // the output of this function is the next PHA word to select

input [ 4:0] phasel      ; // controls the location of the PHA word multiplexer in pha_logic
input [17:0] pha_send     ; // map of PHA channels to be send by event_data to the AEM
input         MUXEND       ; // variable to determine if PHA select mux is at last value

begin // Priority encoded logic for both ZS_ENABLED and ZS_DISABLED

  if (pha_send[00] && (MUXEND          )) Next_PHA =  0;
  else if (pha_send[01] && (MUXEND | phasel <  1)) Next_PHA =  1;
  else if (pha_send[02] && (MUXEND | phasel <  2)) Next_PHA =  2;
  else if (pha_send[03] && (MUXEND | phasel <  3)) Next_PHA =  3;
  else if (pha_send[04] && (MUXEND | phasel <  4)) Next_PHA =  4;
  else if (pha_send[05] && (MUXEND | phasel <  5)) Next_PHA =  5;
  else if (pha_send[06] && (MUXEND | phasel <  6)) Next_PHA =  6;
  else if (pha_send[07] && (MUXEND | phasel <  7)) Next_PHA =  7;
  else if (pha_send[08] && (MUXEND | phasel <  8)) Next_PHA =  8;
  else if (pha_send[09] && (MUXEND | phasel <  9)) Next_PHA =  9;
  else if (pha_send[10] && (MUXEND | phasel < 10)) Next_PHA = 10;
  else if (pha_send[11] && (MUXEND | phasel < 11)) Next_PHA = 11;
  else if (pha_send[12] && (MUXEND | phasel < 12)) Next_PHA = 12;
  else if (pha_send[13] && (MUXEND | phasel < 13)) Next_PHA = 13;
  else if (pha_send[14] && (MUXEND | phasel < 14)) Next_PHA = 14;
  else if (pha_send[15] && (MUXEND | phasel < 15)) Next_PHA = 15;
  else if (pha_send[16] && (MUXEND | phasel < 16)) Next_PHA = 16;
  else if (pha_send[17] && (MUXEND | phasel < 17)) Next_PHA = 17;

  else Next_PHA = 31; // defaults to end-of-data indicator if no valid PHA data found
end

```

```

endfunction /* Next_PHA */

//*****************************************************************************
// Event Data State Machine

always @ (posedge ck or negedge reset_n)
  if (!reset_n) reset_event_logic;
  else

    case (evt_st)

      WAIT_MAP: // Wait for the HitMap to be ready to read (state 0)
      begin
        data_busy <= INACTIVE ;
        edata <= CLEAR ;
        phasel <= INIT_PHASEL ;
        pha_send <= CLEAR ;

        if (PHA_Ready == ACTIVE)
          begin
            bit_ctr <= INIT_BITCTR ;
            pha_ct <= INIT_PHACT ;
            evt_st <= COUNT_PHA ;
          end
      end

      COUNT_PHA: // Count up number of PHA words to send & put in pha_ct (state 1)
      begin
        data_busy <= ACTIVE ;
        if ((bit_ctr == PHA_LIMIT) || ((pha_ct == max_pha) && (zs_status == ZS_ENABLED))) // summing of pha is complete
          begin
            bit_ctr <= INIT_BITCTR ;
            edata <= START_BIT ;
            evt_st <= SEND_MAP ;
          end
        else
          begin
            bit_ctr <= bit_ctr + 1;

            if ( (zs_status == ZS_DISABLED) | (zs_map[bit_ctr] & pha_enable[bit_ctr]) )
              begin
                pha_ct <= pha_ct + 1 ;
                pha_send[bit_ctr] <= 1'b1 ;
              end
          end
      end

      SEND_MAP: // Transmit Hit Map (state 2)
      begin
        if (bit_ctr > 16)
          begin
            edata <= HitMap[17] ;
            bit_ctr <= INIT_BITCTR ;
            evt_st <= SEND_ZS_DATA ;
          end
        else
          begin
            edata <= HitMap[bit_ctr] ;
            bit_ctr <= bit_ctr + 1 ;
          end
      end
    endcase
  end
endfunction

```

```

SEND_ZS_DATA: // Transmit ZS Map (state 3)

    if (bit_ctrl > 16)
        begin
            edata   <= pha_send[17] ;
            bit_ctrl <= INIT_BITCTR ;
            evt_st   <= SEND_CMD_ERR ;
        end

    else
        begin
            edata   <= pha_send[bit_ctrl] ;
            bit_ctrl <= bit_ctrl + 1      ;
        end

SEND_CMD_ERR: // Transmit command error bit (state 4)
              // (not necessarily an error, but the status bit)

begin
    edata     <= cmd_error  ;
    HitMaplch <= ACTIVE      ;
    evt_st    <= SEND_PARITY;
end

SEND_PARITY: // Transmit event data header parity bit (state 5)

begin
    phasel  <= Next_PHA(phasel, pha_send, MUXEND);
    edata   <= parity      ;
    bit_ctrl <= 14           ;
    evt_st   <= SEND_PHA   ;
end

SEND_PHA: // Send the individual PHA words (state 6)

if (pha_ct == 0) // no more PHA to send
begin
    edata   <= INIT_EDATA ;
    evt_st <= SM_REINIT ;
    phasel <= LAST_PHA   ;
end

else
    if (bit_ctrl == 0)
        begin
            bit_ctrl <= 14          ; // #bits in PHA word, 14-0
            pha_ct  <= pha_ct - 1   ;
            edata   <= pha_parity  ;
            phasel  <= Next_PHA(phasel, pha_send, MUXEND);
        end

    else // pha_ct > 0 and bit_ctrl > 0
        begin
            if (bit_ctrl == 14) edata <= 1'b1           ; // A PHA word will follow
            else                 edata <= pha[bit_ctrl-1] ;

            bit_ctrl <= bit_ctrl - 1 ;
        end

SM_REINIT: // Reinitialize event data state machine variables (state 7)

begin
    edata <= INIT_EDATA  ;

    if (HitMaprdy == INACTIVE) // wait for disc_logic to reinitialize
        begin

```



```

/*****************/
// PHA Logic Module Ports

module pha_logic  (
    ck           , // 20 MHz system clock
    reset_n      , // global GARC reset signal
    start_pha   , // semaphore to initiate a conversion
    chid         , // channel id bus from the GAFE ASICs
    sdin         , // serial data in bus from the GAFE ADCs
    sck          , // ADC clock output, common to all GAFE ADCs
    pha_cs_n    , // ADC chip select, common to all GAFE ADCs
    pha_thresh_00 , // PHA Zero Suppression Threshold for GAFE at address 00
    pha_thresh_01 , // PHA Zero Suppression Threshold for GAFE at address 01
    pha_thresh_02 , // PHA Zero Suppression Threshold for GAFE at address 02
    pha_thresh_03 , // PHA Zero Suppression Threshold for GAFE at address 03
    pha_thresh_04 , // PHA Zero Suppression Threshold for GAFE at address 04
    pha_thresh_05 , // PHA Zero Suppression Threshold for GAFE at address 05
    pha_thresh_06 , // PHA Zero Suppression Threshold for GAFE at address 06
    pha_thresh_07 , // PHA Zero Suppression Threshold for GAFE at address 07
    pha_thresh_08 , // PHA Zero Suppression Threshold for GAFE at address 08
    pha_thresh_09 , // PHA Zero Suppression Threshold for GAFE at address 09
    pha_thresh_10 , // PHA Zero Suppression Threshold for GAFE at address 10
    pha_thresh_11 , // PHA Zero Suppression Threshold for GAFE at address 11
    pha_thresh_12 , // PHA Zero Suppression Threshold for GAFE at address 12
    pha_thresh_13 , // PHA Zero Suppression Threshold for GAFE at address 13
    pha_thresh_14 , // PHA Zero Suppression Threshold for GAFE at address 14
    pha_thresh_15 , // PHA Zero Suppression Threshold for GAFE at address 15
    pha_thresh_16 , // PHA Zero Suppression Threshold for GAFE at address 16
    pha_thresh_17 , // PHA Zero Suppression Threshold for GAFE at address 17
    adc_tacq     , // ADC Acquisition Timer Counter Value
    hold         , // GAFE hold signal from PHA logic
    zs_map       , // map bus for PHA values above threshold
    pha          , // memory of 17 x 12 bit PHA values
    phasesel    , // 5 bit control for PHA value multiplexer
    FREE_ID_IN  , // serial data stream for FREE board identification
    FREE_id     , // 8 bit FREE board data identification
    PHA_Ready   // semaphore indicating that PHA conversion is complete
);

```

```

/*****************/
/*
-----+
| |
| GLAST ACD Readout Controller ASIC (GARC) PHA Capture Logic |
| |
| Project : Gamma-Large Area Space Telescope (GLAST)           |
|               Anti-Coincidence Detector (ACD)                  |
|               GLAST ACD Readout Controller (GARC) ASIC          |
| |
| Written      : D. Sheppard                                     |
| Module       : pha_logic.v                                     |
| Original     : 11-19-01                                       |
| Updated      : 03-19-02                                       |
| Version      : 1.0                                           |
| |
-----+
| VERILOG-XL 3.10.p001
-----+
*****
```

#### Version History & Changes Made

03-19-02	: Updated INIT_FREE_ID parameter
03-18-02	: Add FREE board identification feature
03-15-02	: Add ADC acquisition timer
03-13-02	: Add HOLD signal to ADC state machine
03-12-02	: Change polarity on reset to active low
02-19-02	: Change capture_pha_bit task to use shift register topology

```

02-13-02 : Change pha_cs to pha_cs_n
02-12-02 : Change to preclude zero-suppression if high energy range
02-01-02 : Comments added; delete pha_range_status

```

Version 1.0: Initial try at pha\_logic module

---

This module controls the readout of the 18 MAX145 ADCs on the GLAST ACD Event Processor board, provides the zero-suppression function.

Upon receipt of a "start\_pha" signal, the pha logic module activates the chip select on the ADCs, provides 16 clocks for the conversion, and latches the serial data into registers. All ADCs on an event board run in parallel.

This data is available via a 17 bit parallel register that is multiplexed to each data capture register.

Data output has the following format:  
 Bit [12] is the range select bit  
 Bits [11: 0] are the 12 bit PHA read out from the ADC

For the MAX145, the External Clock Timing mode is utilized

Datasheets for this ADC are available at <http://www.maxim-ic.com>

Conversion with the MAX145 occurs as follows:

- (1) start pulse is received by the pha\_logic module (minimum one ck cycle)
- (2) chip select is pulled low with sck kept low (wake up mode)
- (3) a delay for waking up for 2.5 us is required
- (4) sck is brought low for sampling
- (5) wait for Tconv
- (6) send 16 clocks. Data changes on negedge sck.
 

```

clock # 1: 1
clock # 2: 1
clock # 3: 1
clock # 4: CHID
clock # 5: Data MSB, D11
clock # 6: D10
clock # 7: D9
clock # 8: D8
clock # 9: D7
clock #10: D6
clock #11: D5
clock #12: D4
clock #13: D3
clock #14: D2
clock #15: D1
clock #16: D0
      
```
- (7) chip select is pulled inactive high to end sequence

\*/

---

// Module I/O

```

/* pha_logic inputs */
input [17: 0] chid ; // pha range input bits from the 18 GAFEs
input ck ; // global 20 MHz system clock
input [ 4: 0] phasel ; // pha multiplexer select from event data module
input [11: 0] pha_thresh_00 ; // pha ZS threshold, ch. 0
input [11: 0] pha_thresh_01 ; // pha ZS threshold, ch. 1
input [11: 0] pha_thresh_02 ; // pha ZS threshold, ch. 2
input [11: 0] pha_thresh_03 ; // pha ZS threshold, ch. 3
input [11: 0] pha_thresh_04 ; // pha ZS threshold, ch. 4
input [11: 0] pha_thresh_05 ; // pha ZS threshold, ch. 5
input [11: 0] pha_thresh_06 ; // pha ZS threshold, ch. 6

```

```

input [11: 0] pha_thresh_07 ; // pha ZS threshold, ch. 7
input [11: 0] pha_thresh_08 ; // pha ZS threshold, ch. 8
input [11: 0] pha_thresh_09 ; // pha ZS threshold, ch. 9
input [11: 0] pha_thresh_10 ; // pha ZS threshold, ch. 10
input [11: 0] pha_thresh_11 ; // pha ZS threshold, ch. 11
input [11: 0] pha_thresh_12 ; // pha ZS threshold, ch. 12
input [11: 0] pha_thresh_13 ; // pha ZS threshold, ch. 13
input [11: 0] pha_thresh_14 ; // pha ZS threshold, ch. 14
input [11: 0] pha_thresh_15 ; // pha ZS threshold, ch. 15
input [11: 0] pha_thresh_16 ; // pha ZS threshold, ch. 16
input [11: 0] pha_thresh_17 ; // pha ZS threshold, ch. 17
input [ 5: 0] adc_tacq ; // ADC acquisition timer preset
input hold ; // GAFE hold
input reset_n ; // GARC reset
input [17: 0] sdin ; // 18 ADCs serial data input
input start_pha ; // signal to start the conversion process
input FREE_ID_IN ; // serial FREE board identification code

```

```

/* pha_logic outputs */
output [12: 0] pha ; // output of PHA multiplexer
output pha_cs_n ; // ADC chip select, active lo
output PHA_Ready ; // pha ready signal to event data module
output sck ; // serial output clock to ADCs
output [17: 0] zs_map ; // map of ZS threshold vs PHA
output [ 7: 0] FREE_id ; // 8 bit FREE board id

```

```

//****************************************************************************
// pha_logic wires and registers

```

```

reg PHA_Ready ;
reg sck ;
reg pha_cs_n ;
reg [ 5: 0] wake_time ;
reg [ 5: 0] acq_time ;
reg [ 4: 0] sck_ct ;
reg [12: 0] pha00 ;
reg [12: 0] pha01 ;
reg [12: 0] pha02 ;
reg [12: 0] pha03 ;
reg [12: 0] pha04 ;
reg [12: 0] pha05 ;
reg [12: 0] pha06 ;
reg [12: 0] pha07 ;
reg [12: 0] pha08 ;
reg [12: 0] pha09 ;
reg [12: 0] pha10 ;
reg [12: 0] pha11 ;
reg [12: 0] pha12 ;
reg [12: 0] pha13 ;
reg [12: 0] pha14 ;
reg [12: 0] pha15 ;
reg [12: 0] pha16 ;
reg [12: 0] pha17 ;
reg [12: 0] pha ;
reg [ 2: 0] pst ;
reg [ 3: 0] bit_ct ;
reg pha_ck_en ;
reg [ 2: 0] sck_timer ;
wire [17: 0] zs_map ;
reg [ 7: 0] FREE_id ;

```

```

//****************************************************************************
// Local Module Parameters

```

```

parameter
    // PHA logic state machine states
    Wait_4_PHA      = 0 , // waiting for start signal
    PHA_Wake        = 1 , // start received, activate ADC
    PHA_Acq_Time   = 2 , // wait for signal settling time
    PHA_Ck_Enable   = 3 , // enable PHA clock
    PHA_Convert     = 4 , // ADC activated, start conversion

```

```

PHA_EndConv      = 5 , // ADC end of conversion

Range_Bit        = 12 , // position of range bit in pha words
Tick_Range       = 3'h4 , // number of 20 MHz clocks between ADC clock transitions
WAKE_DELAY       = 49 , // number of 20 MHz clock cycles to wait for ADC activation

Low_Select       = 0 , // GAFE analog range LOW select
High_Select      = 1 , // GAFE analog range HIGH select

HI              = 1 ,
LO              = 0 ,
ENABLED         = 1 ,
DISABLED        = 0 ,
ACTIVE          = 1 ,

INACTIVE_HI     = 1 ,
ACTIVE_LO       = 0 ,

CLEAR           = 0 ,
INIT_BIT_CT    = 15 , // comment?
INIT_FREE_ID    = 8'hFF,

DEFAULT_PHA_VALUE = 13'h1AFA,
PHA_VERSION     = 0 ;

//****************************************************************************
// All Output registes are initialized at reset_n

task initialize_pha_module;

begin
  acq_time  <= CLEAR      ;
  FFREE_id   <= INIT_FREE_ID;
  pst        <= Wait_4_PHA  ;
  pha_ck_en <= DISABLED   ;
  PHA_Ready  <= CLEAR      ;
  pha00      <= CLEAR      ;
  pha01      <= CLEAR      ;
  pha02      <= CLEAR      ;
  pha03      <= CLEAR      ;
  pha04      <= CLEAR      ;
  pha05      <= CLEAR      ;
  pha06      <= CLEAR      ;
  pha07      <= CLEAR      ;
  pha08      <= CLEAR      ;
  pha09      <= CLEAR      ;
  pha10      <= CLEAR      ;
  pha11      <= CLEAR      ;
  pha12      <= CLEAR      ;
  pha13      <= CLEAR      ;
  pha14      <= CLEAR      ;
  pha15      <= CLEAR      ;
  pha16      <= CLEAR      ;
  pha17      <= CLEAR      ;
  pha_cs_n   <= INACTIVE_HI ;
  sck        <= LO          ;
  sck_ct     <= CLEAR      ;
  bit_ct     <= INIT_BIT_CT ;
  wake_time  <= WAKE_DELAY  ;
end

endtask /* initialize_pha_module */

//****************************************************************************
// ZS Comparators (combinatorial; total layout space = 18 x 0.02 mm^2)
// ADC data at input is inverted, allowing data > threshold to be correct, 2-6-02
// For Range Bit = Hi, there is no zero-suppression

assign zs_map[00] = (pha00[Range_Bit] | (pha00[11:0] > pha_thresh_00)) ;
assign zs_map[01] = (pha01[Range_Bit] | (pha01[11:0] > pha_thresh_01)) ;

```

```

assign zs_map[02] = (pha02[Range_Bit] | (pha02[11:0] > pha_thresh_02)) ;
assign zs_map[03] = (pha03[Range_Bit] | (pha03[11:0] > pha_thresh_03)) ;
assign zs_map[04] = (pha04[Range_Bit] | (pha04[11:0] > pha_thresh_04)) ;
assign zs_map[05] = (pha05[Range_Bit] | (pha05[11:0] > pha_thresh_05)) ;
assign zs_map[06] = (pha06[Range_Bit] | (pha06[11:0] > pha_thresh_06)) ;
assign zs_map[07] = (pha07[Range_Bit] | (pha07[11:0] > pha_thresh_07)) ;
assign zs_map[08] = (pha08[Range_Bit] | (pha08[11:0] > pha_thresh_08)) ;
assign zs_map[09] = (pha09[Range_Bit] | (pha09[11:0] > pha_thresh_09)) ;
assign zs_map[10] = (pha10[Range_Bit] | (pha10[11:0] > pha_thresh_10)) ;
assign zs_map[11] = (pha11[Range_Bit] | (pha11[11:0] > pha_thresh_11)) ;
assign zs_map[12] = (pha12[Range_Bit] | (pha12[11:0] > pha_thresh_12)) ;
assign zs_map[13] = (pha13[Range_Bit] | (pha13[11:0] > pha_thresh_13)) ;
assign zs_map[14] = (pha14[Range_Bit] | (pha14[11:0] > pha_thresh_14)) ;
assign zs_map[15] = (pha15[Range_Bit] | (pha15[11:0] > pha_thresh_15)) ;
assign zs_map[16] = (pha16[Range_Bit] | (pha16[11:0] > pha_thresh_16)) ;
assign zs_map[17] = (pha17[Range_Bit] | (pha17[11:0] > pha_thresh_17)) ;

//****************************************************************************
// Instantiate the 2 MHz oscillator

always @ (posedge ck or negedge reset_n)
begin
    if (!reset_n) sck_timer <= 0;
    else
        if ((sck_timer == Tick_Range) || (pha_ck_en == DISABLED))
            sck_timer <= 0;
        else
            sck_timer <= sck_timer + 1;
end

//****************************************************************************

always @ (posedge ck or negedge reset_n)
begin
    if (!reset_n) initialize_pha_module;
    else
        case (pst)
            Wait_4_PHA: // wait for signal to start a conversion
            begin
                wake_time      <= WAKE_DELAY      ;
                acq_time       <= adc_tacq       ;
                sck            <= LO             ;
                sck_ct         <= CLEAR          ;
                bit_ct         <= INIT_BIT_CT   ;
                pha_cs_n       <= INACTIVE_HI   ;
                PHA_Ready      <= CLEAR          ;
                if (start_pha)
                    begin
                        pha_cs_n <= ACTIVE_LO ;
                        pst      <= PHA_Wake ;
                    end
                end
            end

            PHA_Wake : // wake up the ADC (wait the required 2.5 uSec)
            begin
                if (wake_time > 0) wake_time <= wake_time - 1 ;
                else pst <= PHA_Acq_Time ;
            end

            PHA_Acq_Time: // wait for signal acquisition
            begin
                if (hold == ACTIVE)
                    begin
                        if (acq_time > 0) acq_time <= acq_time - 1 ;
                        else pst <= PHA_Ck_Enable ;
                    end
            end
        endcase
end

```

```

PHA_Ck_Enable: // wait for HOLD and enable the PHA clock
begin
    pha_ck_en  <= ENABLED      ;
    FREE_id[7]  <= FREE_ID_IN ;
    pst        <= PHA_Convert ;
end

PHA_Convert : // send ADC clocks and capture serial data

begin
    capture_FREE_id ,

    if (sck_timer == Tick_Range)
        case (sck_ct)
            00: clock_hi      ; // ADC ck #01
            01: clock_lo      ;
            02: clock_hi      ; // ADC ck #02
            03: clock_lo      ;
            04: clock_hi      ; // ADC ck #03
            05: clock_lo      ;
            06: clock_hi      ; // ADC ck #04
            07: clock_lo      ;
            08: capture_pha_bit; // ADC ck #05 (MSB, 11)
            09: clock_lo      ;
            10: capture_pha_bit; // ADC ck #06 (bit 10)
            11: clock_lo      ;
            12: capture_pha_bit; // ADC ck #07 (bit 9)
            13: clock_lo      ;
            14: capture_pha_bit; // ADC ck #08 (bit 8)
            15: clock_lo      ;
            16: capture_pha_bit; // ADC ck #09 (bit 7)
            17: clock_lo      ;
            18: capture_pha_bit; // ADC ck #10 (bit 6)
            19: clock_lo      ;
            20: capture_pha_bit; // ADC ck #11 (bit 5)
            21: clock_lo      ;
            22: capture_pha_bit; // ADC ck #12 (bit 4)
            23: clock_lo      ;
            24: capture_pha_bit; // ADC ck #13 (bit 3)
            25: clock_lo      ;
            26: capture_pha_bit; // ADC ck #14 (bit 2)
            27: clock_lo      ;
            28: capture_pha_bit; // ADC ck #15 (bit 1)
            29: clock_lo      ;
            30: capture_pha_bit; // ADC ck #16 (LSB, bit 0)
            31: begin
                    clock_lo;
                    PHA_Ready <= HI;
                    pst      <= PHA_EndConv;
                end
            default: clock_lo;
        endcase
    end

PHA_EndConv: // end conversion
begin
    if ((phasel == 'h1F) || (start_pha == 0)) // transition to idle state and reset
all
    begin
        capture_range_bit      ; // capture range bit after allowing GAFE max setup
        pha_ck_en <= DISABLED   ;
        pha_cs_n  <= INACTIVE_HI ;
        PHA_Ready <= CLEAR      ;
        pst      <= Wait_4_PHA   ;
    end

    if (sck_timer == Tick_Range) // stop clock and finish conversion
    begin
        pha_ck_en <= DISABLED   ;
        pha_cs_n  <= INACTIVE_HI ;

```

```

        end

    end

    default: pst <= Wait_4_PHA ; // error trap only
endcase /* pst PHA state machine */

//****************************************************************************
// Tasks to toggle clock edges and increment bit counter

task clock_hi;
begin
    sck     <= 1;
    sck_ct <= sck_ct + 1;
end
endtask /* clock hi */

task clock_lo;
begin
    sck     <= 0;
    sck_ct <= sck_ct + 1;
    bit_ct <= bit_ct - 1;
end
endtask /* clock lo */

//****************************************************************************
// Task capture_range_bit clocks the status of the GAFE autorange discriminators at the
// time of PHA initiation into the PHA memory

task capture_range_bit;
begin
    pha00[Range_Bit] <= chid[ 0] ;
    pha01[Range_Bit] <= chid[ 1] ;
    pha02[Range_Bit] <= chid[ 2] ;
    pha03[Range_Bit] <= chid[ 3] ;
    pha04[Range_Bit] <= chid[ 4] ;
    pha05[Range_Bit] <= chid[ 5] ;
    pha06[Range_Bit] <= chid[ 6] ;
    pha07[Range_Bit] <= chid[ 7] ;
    pha08[Range_Bit] <= chid[ 8] ;
    pha09[Range_Bit] <= chid[ 9] ;
    pha10[Range_Bit] <= chid[10] ;
    pha11[Range_Bit] <= chid[11] ;
    pha12[Range_Bit] <= chid[12] ;
    pha13[Range_Bit] <= chid[13] ;
    pha14[Range_Bit] <= chid[14] ;
    pha15[Range_Bit] <= chid[15] ;
    pha16[Range_Bit] <= chid[16] ;
    pha17[Range_Bit] <= chid[17] ;

end
endtask /* capture_range_bit */

//****************************************************************************
// Task capture_pha_bit sets the ADC clock hi and captures the PHA bit (given by bit_ct)
// into the PHA memory. Note that bits are inverted to account for negative going analog
// pulse.

task capture_pha_bit;
begin
    clock_hi                                ; // task clock_hi

    pha00[11:0] <= {pha00[10:0], ~sdin[ 0]} ;
    pha01[11:0] <= {pha01[10:0], ~sdin[ 1]} ;
    pha02[11:0] <= {pha02[10:0], ~sdin[ 2]} ;
    pha03[11:0] <= {pha03[10:0], ~sdin[ 3]} ;

```

```

pha04[11:0] <= {pha04[10:0], ~sdin[ 4]} ;
pha05[11:0] <= {pha05[10:0], ~sdin[ 5]} ;
pha06[11:0] <= {pha06[10:0], ~sdin[ 6]} ;
pha07[11:0] <= {pha07[10:0], ~sdin[ 7]} ;
pha08[11:0] <= {pha08[10:0], ~sdin[ 8]} ;
pha09[11:0] <= {pha09[10:0], ~sdin[ 9]} ;
pha10[11:0] <= {pha10[10:0], ~sdin[10]} ;
pha11[11:0] <= {pha11[10:0], ~sdin[11]} ;
pha12[11:0] <= {pha12[10:0], ~sdin[12]} ;
pha13[11:0] <= {pha13[10:0], ~sdin[13]} ;
pha14[11:0] <= {pha14[10:0], ~sdin[14]} ;
pha15[11:0] <= {pha15[10:0], ~sdin[15]} ;
pha16[11:0] <= {pha16[10:0], ~sdin[16]} ;
pha17[11:0] <= {pha17[10:0], ~sdin[17]} ;

end
endtask /* capture_pha_bit */

//****************************************************************************
// FREE Board Id Capture
/*
FREE board identification data is loaded by the pha_cs_n pulse into an 8 bit shift
register residing external to the GARC on the FREE board. The positive edge of the
ADC clock is used to clock this identification data into the FREE_ID_IN pin on the GARC.
This data is loaded into the FREE_id word for later readout by the garc_cmd_proc. Data
is shifted on the positive edge of ADC clock and sampled in this task on the negative edge
for the first 8 clocks.
*/
task capture_FREE_id;
begin

    case (sck_ct)
        01: FREE_id[6] <= FREE_ID_IN ;
        03: FREE_id[5] <= FREE_ID_IN ;
        05: FREE_id[4] <= FREE_ID_IN ;
        07: FREE_id[3] <= FREE_ID_IN ;
        09: FREE_id[2] <= FREE_ID_IN ;
        11: FREE_id[1] <= FREE_ID_IN ;
        13: FREE_id[0] <= FREE_ID_IN ;
    endcase

    end
endtask /* capture_FREE_id */

//****************************************************************************
// PHA Multiplexer

always @ (phasel or pha00 or pha01 or pha02 or pha03 or pha04 or pha05 or
          pha06 or pha07 or pha08 or pha09 or pha10 or pha11 or pha12 or
          pha13 or pha14 or pha15 or pha16 or pha17)

    case (phasel)

        00 : pha = pha00 ;
        01 : pha = pha01 ;
        02 : pha = pha02 ;
        03 : pha = pha03 ;
        04 : pha = pha04 ;
        05 : pha = pha05 ;
        06 : pha = pha06 ;
        07 : pha = pha07 ;
        08 : pha = pha08 ;
        09 : pha = pha09 ;
        10 : pha = pha10 ;
        11 : pha = pha11 ;
        12 : pha = pha12 ;
        13 : pha = pha13 ;
        14 : pha = pha14 ;
        15 : pha = pha15 ;
        16 : pha = pha16 ;
        17 : pha = pha17 ;

```

```
default: pha = DEFAULT_PHA_VALUE ;  
endcase /* PHA Multiplexer */  
*****  
endmodule /* pha_logic */
```

```

*****/*
// Look-At-Me Logic Module

module look_at_me_logic (
    ACD_CLK_A      , // in : serial clock from AEM Prime
    ACD_CLK_B      , // in : serial clock from AEM Secondary
    ACD_NSCMD_A    , // in : serial data from AEM Prime
    ACD_NSCMD_B    , // in : serial data from AEM Secondary
    reset_n        , // in : global module reset
    pwr_on_rst     , // in : power on reset
    ck_out         , // out: GARC core clock
    cmdd           , // out: GARE command data
    lookatme_status // out: look-at-me status bit (0 = primary, 1 = secondary)
);

*****/*
/*
-----|
| | GLAST ACD Readout Controller ASIC (GARC) Look-At-Me Logic |
| |
| | Project : Gamma-Large Area Space Telescope (GLAST)          |
| | Anti-Coincidence Detector (ACD)                            |
| | GLAST ACD Readout Controller (GARC) ASIC                  |
| |
| | Written      : D. Sheppard                                |
| | Module       : lookatme_logic.v                           |
| | Original     : 03-07-02                                 |
| | Updated      : 03-13-03                                 |
| | Version      : 3.0                                     |
| |
|-----|
| | VERILOG-XL 3.10.p001
|-----|
*/
*****/*
/* Updates

03-13-03: Fix lookatme_status selection condition
03-12-03: Updates to correct secondary side power-on problem seen in V2 code.
03-13-02: Change polarity of reset to active low
03-11-02: Changed look-at-me pattern and cmdd assignment as a result of Bob's bench
test (parity/polarity was incorrect)

*/
*****/*
// Look-At-Me Port Declarations

input ACD_CLK_A      ;
input ACD_CLK_B      ;
input ACD_NSCMD_A    ;
input ACD_NSCMD_B    ;
input reset_n        ;
input pwr_on_rst     ;
output lookatme_status ;
output ck_out         ;
output cmdd           ;

*****/*
// Look-At-Me Register Declarations

reg [33:0] prime_cmd_sr   ; // shift register for AEM prime command pattern
reg [33:0] secondary_cmd_sr; // shift register for AEM secondary command pattern
reg      prime_select     ; // register indicating primary AEM selection command recd
reg      secondary_select  ; // register indicating secondary AEM selection command recd
reg      lookatme_status   ; // control register for status mux
wire     ck_out           ; // GARC module clock
wire     cmdd              ; // GARC module command data

```

```

//*****
// Look-At-Me Local Parameters
parameter
    LOOK_AT_ME_CMD      = 34'h24153D721 , // GARC config cmd, addr 1, function 4
    SELECT_PRIME        = 0           , // Primary AEM select = 0, the power on default
    SELECT_SECONDARY    = 1           , // Secondary AEM select = 1
    ACTIVE              = 1           , // Active high
    INACTIVE             = 0           , // Inactive low
    CLEAR               = 0           ; // Clear FF to 0

//*****
// Look-At-Me Primary Side

always @ (posedge ACD_CLK_A or negedge reset_n)
    if (!reset_n)
        begin
            prime_cmd_sr <= CLEAR ; // at reset, we clear the shift reg logic
            prime_select <= INACTIVE ;
        end

    else
        if (prime_cmd_sr == LOOK_AT_ME_CMD) // if valid look-at-me, select prime & clear
        begin
            prime_select <= ACTIVE ;
            prime_cmd_sr <= CLEAR ;
        end

    else
        begin // run the shift register
            prime_select      <= INACTIVE ;
            prime_cmd_sr[33:1] <= prime_cmd_sr[32:0] ;
            prime_cmd_sr[ 0] <= ~ACD_NSCMD_A ;
        end

//*****
// Look-At-Me Secondary Side

always @ (posedge ACD_CLK_B or negedge reset_n)
    if (!reset_n)
        begin
            secondary_cmd_sr <= CLEAR ; // at reset, we clear the shift reg logic
            secondary_select <= INACTIVE ;
        end

    else
        if (secondary_cmd_sr == LOOK_AT_ME_CMD) // if valid look-at-me, select secondary & clear
        begin
            secondary_select <= ACTIVE ;
            secondary_cmd_sr <= CLEAR ;
        end

    else
        begin // run the shift register
            secondary_select      <= INACTIVE ;
            secondary_cmd_sr[33:1] <= secondary_cmd_sr[32:0] ;
            secondary_cmd_sr[ 0] <= ~ACD_NSCMD_B ;
        end

//*****
// Look-At-Me Select Determination

always @ (posedge secondary_select or posedge prime_select or posedge pwr_on_rst)
    if (prime_select | pwr_on_rst)
        lookatme_status <= SELECT_PRIME ; // pwr_on_rst and prime_select are asynch reset
    else
        lookatme_status <= SELECT_SECONDARY ; // posedge secondary_select is like the clock

assign ck_out = (lookatme_status ? ACD_CLK_B : ACD_CLK_A) ; // 0 = primary, 1 = secondary
assign cmdd   = (lookatme_status ? ~ACD_NSCMD_B : ~ACD_NSCMD_A) ;

//*****

```

```
endmodule /* look_at_me_logic */
```

```
/*
// Reset Logic module ports

module reset_logic (
    ACD_NRESET_A , // AEM "A" side reset signal, active low
    ACD_NRESET_B , // AEM "B" side reset signal, active low
    ACD_CLK_A , // AEM "A" side clock
    ACD_CLK_B , // AEM "B" side clock
    ck , // GARC core logic clock
    PWR_ON_RST_IN , // External power on reset signal from FREE, active high
    reset_cmd , // Reset command pulse from garc_cmd_proc, active high
    reset_n_out // Reset out of core to global reset buffer
);
```

```
/*
*
-----|
| | GLAST ACD Readout Controller ASIC (GARC) Reset Logic |
| | Project : Gamma-Large Area Space Telescope (GLAST) |
| | Anti-Coincidence Detector (ACD) |
| | GLAST ACD Readout Controller (GARC) ASIC |
| | Written : D. Sheppard |
| | Module : reset_logic_v2.v |
| | Original : 03-11-02 |
| | Updated : 09-16-02 |
| | Version : 2.0 |
|-----|
| | VERILOG-XL 3.10.p001 |
|-----|
*****
```

#### Version History & Changes Made

##### Version 2.0:

09-16-02: Change Power-On Reset to be a clocked reset.  
 Bring in GARC logic core clock from garc\_module.  
 Change reset\_cmd circuit to have a fixed width.

##### Version 1.0:

03-13-02: Change polarity of reset to active low  
 03-11-02: Initial module written in response to look-at-me logic change.

```
*/
// Reset Logic module I/O Declarations
```

```
/* reset_logic Inputs */
input ACD_NRESET_A ; // AEM "A" side reset signal, active low
input ACD_NRESET_B ; // AEM "B" side reset signal, active low
input ACD_CLK_A ; // AEM "A" side clock
input ACD_CLK_B ; // AEM "B" side clock
input PWR_ON_RST_IN ; // External power on reset signal from FREE, active high
input reset_cmd ; // Reset command pulse from garc_cmd_proc, active high
input ck ; // GARC logic core clock

/* reset_logic Outputs */
output reset_n_out ; // Reset out of core to global reset buffer

/*
// Register and Wire Declarations

reg [4:0] a_RST_SR ; // shift register for AEM "A" reset edge detect, deglitch
reg [4:0] b_RST_SR ; // shift register for AEM "B" reset edge detect, deglitch
reg [4:0] pwr_on_a_sr ; // shift register for Power-On reset "A" clk: edge detect, deglitch
reg [4:0] pwr_on_b_sr ; // shift register for Power-On reset "B" clk: edge detect, deglitch
```

```

reg [4:0] reset_cmd_sr      ; // shift register for reset_cmd logic
reg      a_reset            ; // register used for "A" reset detect
reg      b_reset            ; // register used for "B" reset detect
reg      pwr_on_a_reset    ; // register used for power on clk A reset detect
reg      pwr_on_b_reset    ; // register used for power on clk B reset detect
reg      commanded_reset   ; // register used for commanded reset one-shot
wire     reset_n_out        ; // output of OR gate and output for this module

//*********************************************************************
// Module Parameters

parameter
  RESET_PATTERN = 4'b0111 , // 3 clock deglitch plus posedge detect
  ACTIVE        =      1 ,
  INACTIVE      =      0 ;

//*********************************************************************
// Reset Edge Detect and Deglitch for the "A" side AEM

  always @ (posedge ACD_CLK_A)
    begin
      a_rst_sr[4:1] <= a_rst_sr[3:0] ; // shift register
      a_rst_sr[ 0] <= ~ACD_NRESET_A ;
      if ( (a_rst_sr[3:0] == RESET_PATTERN) | (a_rst_sr[4:1] == RESET_PATTERN) ) // makes 100 ns
pulse
      a_reset <= ACTIVE ;
      else
      a_reset <= INACTIVE ;
    end
 //*********************************************************************
// Reset Edge Detect and Deglitch for the "B" side AEM

  always @ (posedge ACD_CLK_B)
    begin
      b_rst_sr[4:1] <= b_rst_sr[3:0] ; // shift register
      b_rst_sr[ 0] <= ~ACD_NRESET_B ;
      if ( (b_rst_sr[3:0] == RESET_PATTERN) | (b_rst_sr[4:1] == RESET_PATTERN) ) // makes 100 ns
pulse
      b_reset <= ACTIVE ;
      else
      b_reset <= INACTIVE ;
    end
 //*********************************************************************
// Reset Edge Detect and Deglitch for the Power-On Reset Input with "A" side clock

  always @ (posedge ACD_CLK_A) // check for reset on clock A
    begin
      pwr_on_a_sr[4:1] <= pwr_on_a_sr[3:0] ; // shift register
      pwr_on_a_sr[ 0] <= ~PWR_ON_RST_IN ;
      if ( (pwr_on_a_sr[3:0] == RESET_PATTERN) | (pwr_on_a_sr[4:1] == RESET_PATTERN) ) // makes 100
ns pulse
      pwr_on_a_reset <= ACTIVE ;
      else
      pwr_on_a_reset <= INACTIVE ;
    end
 //*********************************************************************
// Reset Edge Detect and Deglitch for the Power-On Reset Input with "B" side clock

  always @ (posedge ACD_CLK_B) // check for reset on clock B
    begin

```

