CYPRESS SEMICONDUCTOR CORPORATION

Internal Correspondence

Date:	Thursday 1/31/2013	**WW:**	**1303**
To:	**PSoC Applications**		
Author:	Chris Keeser (KEES)		
Author File#:	KEES#185		
Subject:	**UDB 2nd order CIC decimator filter component**		
Distribution:	PSoC Apps, EXT, DWV		

<u>Summary:</u>

This memo documents and distributes a 2nd order Cascaded Intergrator Comb (CIC) decimation filter implemented in a 16 bit datapath (also called a Sinc^2 decimator). The decimator supports decimation rates as low as 2 and as high as 128, producing effective resolutions of up to 10.5 bits when used with a first order Delta Sigma modulator and raw results as large as +/- 16,384, the equivalent of a signed 15 bit number at a decimation rate of 128. The decimator requires a maximum of 8 clock cycles to complete a calculation, requiring a clock 8 times faster than the modulator clock. When combined with the SC/CT modulator shown in KEES#181 at the maximum modulator clock rate of 4 Mhz, the decimator requires a 32 Mhz clock, resulting in an output sample rate from 363 Ksps with 5 bits of effective resolution, down to 31 Ksps with 10 bits of effective resolution. The decimator component includes a DMA capability file for simplifying the use of DMA with the decimator. Start() is the only API required to use the decimator.

<u>Attached File Summary:</u>

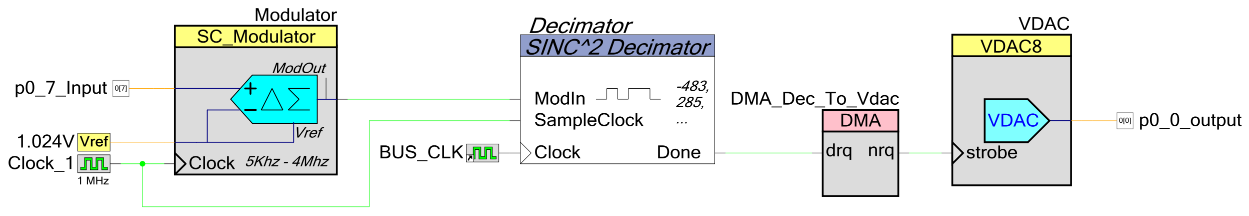
File # 2 is the component archive exported using Creator 2.2's component export feature. To use this file, download it and remove the .PDF extension.

File # 3 is an example project bundle with the modulator and the decimator. To use the file, download it and remove the .PDF extension.

File # 4 are the scans of the datapath instruction configurations for achieving the CIC Decimator.

<u>Details:</u>

Here is the component used in a simple example project:



The component has 3 parameters for customization:

1.) Decimation: Sets the decimation rate for the filter. This parameter sets the effective resolution of the output, the maximum gain of the filter and the output sample rate. For more information, check out the description of the parameter in the component.
2.) Enable_Reset: Enables an optional reset terminal that resets the filter when asserted. After a reset, the first sample from the decimator must be discarded.
3.) Debug: Enables an optional debug mode which exposes the internal signals of the decimator. This is only useful for the curious and is not required for normal operation.

To use the Decimator, call the KEES_CIC_Decimator_Start() API. The output of the decimator is written into the FIFO of the datapath, and the 'Done' signal is the "Not Empty" status from that FIFO. The Done signal will stay asserted until the FIFO is empty. You can store up to 4 results in the FIFO, but you will not be made aware that other samples are complete after the Done signal is asserted the first time. You also will not know if the FIFO overflows. The best practice is to read the sample out and empty the FIFO before the next conversion is complete.

This component can be used for all sorts of creative things besides making an ADC with the SC/CT modulators. Any density signal you want to convert to a number can be fed into the decimator as well. Get creative!

Technical notes (jotted down so I don't forget)

The first sample from the decimator must be ignored, same as the 2nd order Sinc^2 filters in PSoC 1.

Uses 2 datapaths as a 16 bit datapath.

Filter bit width requirements are InputWidth+FilterStages*log2(Decimation) where InputWidth = 1, FilterStages = 2 and decimation =128 => 15 bits.

Filter gain is (SampleDelayInComb*DecimationRatio)^(FilterOrder) where the SampleDelayInComb = 1 and FilterOrder = 2.

Count7 was used for controlling the decimation. The first time the count7 ran after the part is reset gave me my desired period + 1, so that affected calculation #1 and #2. The solution was to write the desired period into the count7 cell during the start API. This solved the issue seen on the first startup after the part reset.

FIFO load-to-read timing was unexpected (seems to take more than 1 clock for data loaded into the FIFO to be available to be read back, waiting for confirmation from design on the timing). My back to back F0 load and F0 read instructions #6 and #7 were not operating as expected. There appears to be multiple clocks required between loading the FIFO and being able to read it out to D0. I had to solve this problem by using the FIFO in "Single buffer mode" (only briefly mentioned on page 81 of the UDB BROS, spec # 001-08005 Rev. *Q) by setting the FIF00_CLR bit in the AUX_CTRL register (UWRK_UWRK8_Bx_UDByy_ACTL). This turned the FIFO into a single byte register with immediate read/write capability (implying the FIFO does not have this capability), which was sufficient for my needs.

Update I heard back from design on this aspect and I learned 2 new and interesting things:

1.) D0_load and D1_load signals are EDGE detected only, meaning that you must assert, then de-assert the D0_load signal in order to load it again. The edge detection logic runs on the datapath clock, so you cannot do "back to back" loads of the D register from the FIFO. I got lucky in my design and only loaded D0 on every other instruction, so it always had an edge when needed.
2.) When the FIFO is in "output" mode, i.e. data is written into the FIFO from the datapath for consumption by the BUS (CPU / DMA) then D0_load *does not* update the read pointer. Basically, the FIFO only recognizes read requests from the BUS, and therefore doesn't see the D0_load reads at all. The only way to get the FIFO to work as temporary datapath storage, is to use it in "single buffer mode" as explained above. In this mode, the read and write pointers never move, so the reads and writes are always valid, albeit limited to a single byte depth.

I added a component "debug capability" file to make is super easy to monitor all the datapath registers, although the component authors guide is seriously lacking in this area. The guide was barely enough to get the basics, and it took a lot of trial and error to get it working properly. Notes on the component debugger:

a.) You had to refresh the memory to get it to update (I bound debug->refresh to f12 on my keyboard to simplify this).
b.) The component debug tool is much easier to use than the technique in KEES#132

c.) To change the radix, you have to right click on a field name, then change the radix, then close the component window and re-open it to get the radix to change, and it alters all the fields
d.) The integer radix in the component debugger is unsigned only
e.) The debugger doesn't remember you had a component window open. You have to re-open it every time.
f.) I filed CDT# 144250 on all of these problems
g.) Oddly, there is no mention of the dma capability files in the component authors guide

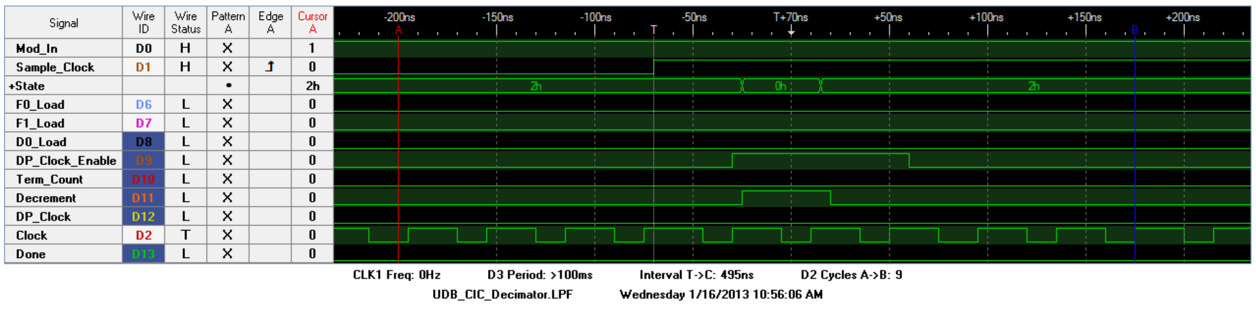
I wrote a Python simulation of the datapath decimator for debugging (because things did not work right based on the FIFO problem and the count7 initialization problem. The python sim shows the expected values for all the registers, including the FIFOs (which I still don't know if we can read using the debugger). It was an invaluable tool in discovering issues.

No "wait" datapath state, so I have to use the UDB clock enable block to hold the datapath between instructions when needed. Every single instruction was used to do something, so I had to get creative in order hold when waiting for the next sample to come in and nor bork values. I used a "UDB clock enable" to control the clock. More detailed information needs to be provided for the exact timing through these UDB clock enable blocks because I needed to know exactly WHEN my clock was present or not, and I could not tolerate magically inserted "synchronization" logic introducing delays. I had to bypass synchronization and ensure that the enable signal was generated from the same clock to prevent things mysteriously showing up in line with my logic.

The actual implementation of the algorithm took some work, since I needed to keep track of 4 variables and we can only write directly into A0 and A1. D0 and D1 are only good for storing static data since you cannot modify it and write it back to keep it around for later. I had to device a scheme to store results into FIFO0, and then load them from the FIFO0 into the D0. Using this method, I was able to keep 3 variables cycling between the A0 register, the FIFO0 register and D0 register. Unfortunately, I had to insert "Wasted" states that directed particular registers through the ALU in order to write them into the FIFO (since you cannot arbitrarily write data into the FIFOs from dynamic locations and I needed to source data into the FIFO from A0 register, A1 register and the result of A0-D0).

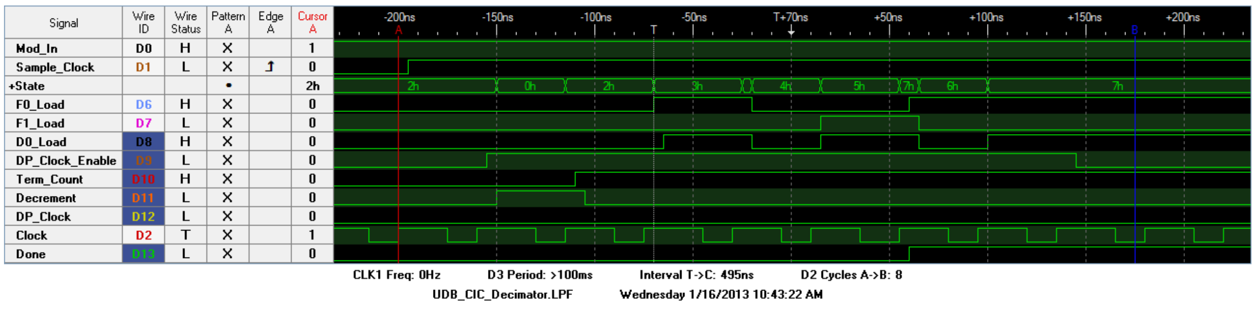
As stated earlier, I needed to be able to load the FIFO on one clock edge, and unload it on the very next edge. The FIFO does not appear to be able to support that, so it had to be placed into single buffer mode, making at act as a single register with no status. Since I only ever needed to store 1 byte in the FIFO at any time, this was no problem, but the exact timing of the FIFO in FIFO mode with the Fx load and Dx load signals is not clear.

Timing diagram shots showing fifo load and D0 load signals. Could be helpful in decoding the verilog and datapath config (in conjunction with the instruction cards attached to this memo)



Signal	Wire ID	Wire Status	Pattern A	Edge A	Cursor A
Mod_In	D0	H	X		1
Sample_Clock	D1	H	X	↑	0
+State			•		2h
F0_Load	D6	L	X		0
F1_Load	D7	L	X		0
D0_Load	D8	L	X		0
DP_Clock_Enable	D9	L	X		0
Term_Count	D10	L	X		0
Decrement	D11	L	X		0
DP_Clock	D12	L	X		0
Clock	D2	T	X		0
Done	D13	L	X		0

CLK1 Freq: 0Hz   D3 Period: >100ms   Interval T->C: 495ns   D2 Cycles A->B: 9
UDB_CIC_Decimator.LPF   Wednesday 1/16/2013 10:56:06 AM



Signal	Wire ID	Wire Status	Pattern A	Edge A	Cursor A
Mod_In	D0	H	X		1
Sample_Clock	D1	L	X	↑	0
+State			•		2h
F0_Load	D6	H	X		0
F1_Load	D7	L	X		0
D0_Load	D8	H	X		0
DP_Clock_Enable	D9	L	X		0
Term_Count	D10	H	X		0
Decrement	D11	L	X		0
DP_Clock	D12	L	X		0
Clock	D2	T	X		1
Done	D13	L	X		0

CLK1 Freq: 0Hz   D3 Period: >100ms   Interval T->C: 495ns   D2 Cycles A->B: 8
UDB_CIC_Decimator.LPF   Wednesday 1/16/2013 10:43:22 AM

Appendix: Verliog

```
//`#start header` -- edit after this line, do not edit this line
// ========================================
//
// Copyright YOUR COMPANY, THE YEAR
// All Rights Reserved
// UNPUBLISHED, LICENSED SOFTWARE.
//
// CONFIDENTIAL AND PROPRIETARY INFORMATION
// WHICH IS THE PROPERTY OF your company.
//
// ========================================
`include "cypress.v"
//`#end` -- edit above this line, do not edit this line
// Generated on 01/11/2013 at 12:42
// Component: KEES_UDB_Decimator
module KEES_UDB_Decimator (
    output  D0_load,
    output  Decrement,
    output  DPClockEnable,
    output  DPClock,
    output  Done,
    output  F0_load,
    output  F1_load,
    output  State_0,
    output  State_1,
    output  State_2,
    output  TermCount,
    input   Clock,
    input   ModIn,
    input   Reset,
    input   SampleClock
);

//`#start body` -- edit after this line, do not edit this line

parameter Decimate = "7'b0000000";

localparam STATE_1P = 3'h0;
localparam STATE_1M = 3'h1;
localparam STATE_2 = 3'h2;
localparam STATE_3 = 3'h3;
localparam STATE_4 = 3'h4;
localparam STATE_5 = 3'h5;
localparam STATE_6 = 3'h6;
localparam STATE_7 = 3'h7;

reg [2:0] State;
reg done;
wire f0_load;
wire f1_load;
wire d0_load;
wire decrement;
wire sample_edge;
reg sample_clock_delayed;

wire termcount;
wire DP_Clock;
reg DP_Clock_Enable;
wire [1:0] fifo_stat;

assign State_0 = State[0];
assign State_1 = State[1];
assign State_2 = State[2];
assign F0_load = f0_load;
assign F1_load = f1_load;
assign D0_load = d0_load;
assign Done = fifo_stat[0];
assign TermCount = termcount;
assign DPClockEnable = DP_Clock_Enable;
assign Decrement = decrement;
assign DPClock = DP_Clock;

// edge detector for sample clock signal
assign sample_edge = (SampleClock == 1'b1 && sample_clock_delayed == 1'b0) ? 1'b1 : 1'b0;

// cominbatorial datapath clock gating
//assign DP_Clock = Clock & DP_Clock_Enable;
// datapath clock gating through udb clock enable block
//assign DP_Clock_Enable = (State == STATE_2 && termcount == 0 && sample_edge == 0) ||
(State == STATE_7 && sample_edge == 0) ? 1'b0 : 1'b1;

// fifo 0 load signal logic
assign f0_load = (State == STATE_3 || State == STATE_6 || State == STATE_7) ? 1'b1 : 1'b0
;
// fifo 1 load signal logic
assign f1_load = (State == STATE_5) ? 1'b1 : 1'b0;
// data register 0 load signal logic
assign d0_load = (State == STATE_3 || State == STATE_5 || State == STATE_7) ? 1'b1 : 1'b0
;

// count 7 decrement signal logic
assign decrement = (State == STATE_1P || State == STATE_1M) ? 1'b1 : 1'b0;


always @(posedge Clock)
begin

    DP_Clock_Enable <= 1;
    // sample clock edge detector register
    sample_clock_delayed <= SampleClock;

    case(State)
```

```verilog
        STATE_1P:
        begin
            //decrement <= 1;
            //DP_Clock_Enable <= 0;
            State <= STATE_2;
        end

        STATE_1M:
        begin
            //decrement <= 1;
            //DP_Clock_Enable <= 0;
            State <= STATE_2;
        end

        STATE_2:
        begin
            if(termcount == 1)
            begin
                State <= STATE_3;
            end
            else if((sample_edge == 1) && (ModIn == 1))
            begin
                State <= STATE_1P;
            end
            else if((sample_edge == 1) && (ModIn == 0))
            begin
                State <= STATE_1M;
            end
            else
            begin
                DP_Clock_Enable <= 0;
                State <= STATE_2;
            end
        end

        STATE_3:
        begin
            //f0_load <= 1;
            //d0_load <= 1;
            State <= STATE_4;
        end

        STATE_4:
        begin
            State <= STATE_5;
        end

        STATE_5:
        begin
            //f1_load <= 1;
            //d0_load <= 1;
            State <= STATE_6;
        end

        STATE_6:
        begin
            //f0_load <= 1;
            //DP_Clock_Enable <= 0;
            State <= STATE_7;
        end

        STATE_7:
        begin
            //f0_load <= 1;
            //d0_load <= 1;

            if((sample_edge == 1) && (ModIn == 1))
            begin
                State <= STATE_1P;
            end
            else if((sample_edge == 1) && (ModIn == 0))
            begin
```

```
                State <= STATE_1M;
            end
            else
            begin
                DP_Clock_Enable <= 0;
                State <= STATE_7;
            end
        end

    endcase

end


// Count7 cell
cy_psoc3_count7 #(.cy_period({Decimate}), .cy_route_ld(`FALSE), .cy_route_en(`TRUE))
Counter7 (
/* input */ .clock (Clock), // Clock
/* input */ .reset(Reset), // Reset
/* input */ .load(1'b0), // Load signal used if cy_route_ld = TRUE
/* input */ .enable(decrement), // Enable signal used if cy_route_en = TRUE
/* output [6:0] */ .count(), // Counter value output
/* output */ .tc(termcount) // Terminal Count output
);

// UDB clock enable
cy_psoc3_udb_clock_enable_v1_0 #(.sync_mode(`FALSE)) MyCompClockSpec (
.enable(DP_Clock_Enable), /* Enable from interconnect */
.clock_in(Clock), /* Clock from interconnect */
.clock_out(DP_Clock) /* Clock to be used for UDB elements in this component */
);

//`#end` -- edit above this line, do not edit this line
cy_psoc3_dp16 #(.cy_dpconfig_a(
{
    `CS_ALU_OP__INC, `CS_SRCA_A0, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC__ALU, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /*CFGRAM0:     A0=A0+1*/
    `CS_ALU_OP__DEC, `CS_SRCA_A0, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC__ALU, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /*CFGRAM1:     A0=A0-1*/
    `CS_ALU_OP__ADD, `CS_SRCA_A0, `CS_SRCB_A1,
    `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC__ALU,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /*CFGRAM2:     A1=A0+A1*/
    `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC___D0, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /*CFGRAM3:     F0=ALU,A0=D0,D0=F0*/
    `CS_ALU_OP__SUB, `CS_SRCA_A1, `CS_SRCB_A0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC__ALU, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /*CFGRAM4:     A0=A1-A0*/
    `CS_ALU_OP__SUB, `CS_SRCA_A0, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /*CFGRAM5:     A0-D0,F1=ALU,D0=F0*/
    `CS_ALU_OP_PASS, `CS_SRCA_A1, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /*CFGRAM6:     F0=ALU*/
    `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC___D0, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /*CFGRAM7:     F0=ALU,A0=D0,D0=F0*/
    8'hFF, 8'h00,  /*CFG9:       */
    8'hFF, 8'hFF,  /*CFG11-10:       */
    `SC_CMPB_A1_D1, `SC_CMPA_A1_D1, `SC_CI_B_ARITH,
    `SC_CI_A_ARITH, `SC_C1_MASK_DSBL, `SC_C0_MASK_DSBL,
    `SC_A_MASK_DSBL, `SC_DEF_SI_0, `SC_SI_B_DEFSI,
    `SC_SI_A_DEFSI, /*CFG13-12:       */
    `SC_A0_SRC_ACC, `SC_SHIFT_SL, 1'h0,
    1'h0, `SC_FIFO1_ALU, `SC_FIFO0_ALU,
    `SC_MSB_DSBL, `SC_MSB_BIT0, `SC_MSB_NOCHN,
    `SC_FB_NOCHN, `SC_CMP1_NOCHN,
    `SC_CMP0_NOCHN, /*CFG15-14:       */
    10'h00, `SC_FIFO_CLK__DP,`SC_FIFO_CAP_AX,
    `SC_FIFO_LEVEL,`SC_FIFO__SYNC,`SC_EXTCRC_DSBL,
    `SC_WRK16CAT_DSBL /*CFG17-16:       */
}
), .cy_dpconfig_b(
{
    `CS_ALU_OP__INC, `CS_SRCA_A0, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC__ALU, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /*CFGRAM0:     A0=A0+1*/
    `CS_ALU_OP__DEC, `CS_SRCA_A0, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC__ALU, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /*CFGRAM1:     A0=A0-1*/
    `CS_ALU_OP__ADD, `CS_SRCA_A0, `CS_SRCB_A1,
    `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC__ALU,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /*CFGRAM2:     A1=A0+A1*/
    `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC___D0, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /*CFGRAM3:     F0=ALU,A0=D0,D0=F0*/
    `CS_ALU_OP__SUB, `CS_SRCA_A1, `CS_SRCB_A0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC__ALU, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /*CFGRAM4:     A0=A1-A0*/
    `CS_ALU_OP__SUB, `CS_SRCA_A0, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /*CFGRAM5:     A0-D0,F1=ALU,D0=F0*/
    `CS_ALU_OP_PASS, `CS_SRCA_A1, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /*CFGRAM6:     F0=ALU*/
    `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC___D0, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /*CFGRAM7:     F0=ALU,A0=D0,D0=F0*/
    8'hFF, 8'h00,  /*CFG9:       */
    8'hFF, 8'hFF,  /*CFG11-10:       */
    `SC_CMPB_A1_D1, `SC_CMPA_A1_D1, `SC_CI_B_ARITH,
    `SC_CI_A_CHAIN, `SC_C1_MASK_DSBL, `SC_C0_MASK_DSBL,
    `SC_A_MASK_DSBL, `SC_DEF_SI_0, `SC_SI_B_DEFSI,
    `SC_SI_A_DEFSI, /*CFG13-12:       */
    `SC_A0_SRC_ACC, `SC_SHIFT_SL, 1'h0,
    1'h0, `SC_FIFO1_ALU, `SC_FIFO0_ALU,
    `SC_MSB_DSBL, `SC_MSB_BIT0, `SC_MSB_NOCHN,
    `SC_FB_NOCHN, `SC_CMP1_CHNED,
    `SC_CMP0_CHNED, /*CFG15-14:       */
    10'h00, `SC_FIFO_CLK__DP,`SC_FIFO_CAP_AX,
    `SC_FIFO_LEVEL,`SC_FIFO__SYNC,`SC_EXTCRC_DSBL,
    `SC_WRK16CAT_DSBL /*CFG17-16:       */
}
)) CIC_Decimator(
        /*  input                   */  .reset(Reset),
        /*  input                   */  .clk(DP_Clock),
        /*  input   [02:00]         */  .cs_addr(State),
        /*  input                   */  .route_si(1'b0),
        /*  input                   */  .route_ci(1'b0),
        /*  input                   */  .f0_load(f0_load),
        /*  input                   */  .f1_load(f1_load),
        /*  input                   */  .d0_load(d0_load),
        /*  input                   */  .d1_load(1'b0),
        /*  output  [01:00]                  */  .ce0(),
        /*  output  [01:00]                  */  .cl0(),
        /*  output  [01:00]                  */  .z0(),
        /*  output  [01:00]                  */  .ff0(),
        /*  output  [01:00]                  */  .ce1(),
        /*  output  [01:00]                  */  .cl1(),
        /*  output  [01:00]                  */  .z1(),
        /*  output  [01:00]                  */  .ff1(),
        /*  output  [01:00]                  */  .ov_msb(),
        /*  output  [01:00]                  */  .co_msb(),
        /*  output  [01:00]                  */  .cmsb(),
        /*  output  [01:00]                  */  .so(),
        /*  output  [01:00]                  */  .f0_bus_stat(),
        /*  output  [01:00]                  */  .f0_blk_stat(),
        /*  output  [01:00]                  */  .f1_bus_stat(fifo_stat),
        /*  output  [01:00]                  */  .f1_blk_stat()
);
endmodule
//`#start footer` -- edit after this line, do not edit this line
//`#end` -- edit above this line, do not edit this line
```

Appendix: Python sim and sample output:

```
from numpy import *

decimation = 4
sattype = int16
disptype = uint16  # to display signed values, use int16
garbage = 3

A0 = sattype(0)
A1 = sattype(0)
D0 = sattype(0)
D1 = sattype(0)
F0 = [sattype(garbage)]
F1 = []

def printstate(instruction):
    print "instruction #: " + str(instruction) + " vvvvvvvvvvvvvvvvvvv"
    print("A0 : ") + str(disptype(A0))
    print("A1 : ") + str(disptype(A1))
    print("D0 : ") + str(disptype(D0))
    print("D1 : ") + str(disptype(D1))
    print("F0 : ") + str(disptype(F0))
    print("F1 : ") + str(disptype(F1))
    print "end of instruction ^^^^^^^^^^^^^^^^^^^^"

# enable the following lines to show the startup error  vvv
#A0 = sattype(A0 + 1)
#printstate(1)
#
#A1 = sattype(A0 + A1)
#printstate(2)
#  the count7 was doing 1 extra cycle on the first result ^^^
# resulting in 5 integrations (decimation of 4) on the first round
# this caused the second result to be in error as well

for i in range(1,4*decimation+1):

    A0 = sattype(A0 + 1)
    printstate(1)

    A1 = sattype(A0 + A1)
    printstate(2)

    if i % decimation == 0:

        F0.append(A0)
        A0 = D0
        D0 = F0.pop(0)
        printstate(3)

        A0 = sattype(A1 - A0)
        printstate(4)

        F1.append(sattype(A0 - D0))
        D0 = F0.pop(0)
        printstate(5)
        print "###################\nOUTPUT: " + str(F1.pop()) + "\n####################"

        F0.append(A1)
        printstate(6)

        F0.append(A0)
        A0 = D0
        D0 = F0.pop(0)
        printstate(7)
```

```
Output:
instruction #: 1 vvvvvvvvvvvvvvvvvv
A0 : 1
A1 : 0
D0 : 0
D1 : 0
F0 : [3]
F1 : []
end of instruction ^^^^^^^^^^^^^^^^^^^

instruction #: 2 vvvvvvvvvvvvvvvvvv
A0 : 1
A1 : 1
D0 : 0
D1 : 0
F0 : [3]
F1 : []
end of instruction ^^^^^^^^^^^^^^^^^^^

instruction #: 1 vvvvvvvvvvvvvvvvvv
A0 : 2
A1 : 1
D0 : 0
D1 : 0
F0 : [3]
F1 : []
end of instruction ^^^^^^^^^^^^^^^^^^^

instruction #: 2 vvvvvvvvvvvvvvvvvv
A0 : 2
A1 : 3
D0 : 0
D1 : 0
F0 : [3]
F1 : []
end of instruction ^^^^^^^^^^^^^^^^^^^

instruction #: 1 vvvvvvvvvvvvvvvvvv
A0 : 3
A1 : 3
D0 : 0
D1 : 0
F0 : [3]
F1 : []
end of instruction ^^^^^^^^^^^^^^^^^^^

instruction #: 2 vvvvvvvvvvvvvvvvvv
A0 : 3
A1 : 6
D0 : 0
D1 : 0
F0 : [3]
F1 : []
end of instruction ^^^^^^^^^^^^^^^^^^^

instruction #: 1 vvvvvvvvvvvvvvvvvv
A0 : 4
A1 : 6
D0 : 0
D1 : 0
F0 : [3]
F1 : []
end of instruction ^^^^^^^^^^^^^^^^^^^
```

```
instruction #: 2 vvvvvvvvvvvvvvvvvv
A0 : 4
A1 : 10
D0 : 0
D1 : 0
F0 : [3]
F1 : []
end of instruction ^^^^^^^^^^^^^^^^^^^

instruction #: 3 vvvvvvvvvvvvvvvvvv
A0 : 0
A1 : 10
D0 : 3
D1 : 0
F0 : [4]
F1 : []
end of instruction ^^^^^^^^^^^^^^^^^^^

instruction #: 4 vvvvvvvvvvvvvvvvvv
A0 : 10
A1 : 10
D0 : 3
D1 : 0
F0 : [4]
F1 : []
end of instruction ^^^^^^^^^^^^^^^^^^^

instruction #: 5 vvvvvvvvvvvvvvvvvv
A0 : 10
A1 : 10
D0 : 4
D1 : 0
F0 : []
F1 : [7]
end of instruction ^^^^^^^^^^^^^^^^^^^

###################
OUTPUT: 7
####################

instruction #: 6 vvvvvvvvvvvvvvvvvv
A0 : 10
A1 : 10
D0 : 4
D1 : 0
F0 : [10]
F1 : []
end of instruction ^^^^^^^^^^^^^^^^^^^

instruction #: 7 vvvvvvvvvvvvvvvvvv
A0 : 4
A1 : 10
D0 : 10
D1 : 0
F0 : [10]
F1 : []
end of instruction ^^^^^^^^^^^^^^^^^^^
.
.
.
```