CYPRESS SEMICONDUCTOR CORPORATION
Internal Correspondence

Date:	Tuesday 1/29/2013 **WW: 1305**
To:	**PSoC Apps**
Author:	Chris Keeser (KEES)
Author File#:	KEES#191
Subject:	**UDB Density Integrator component: density out = integral(density in)**
Distribution:	PSoC Apps

Summary:

This memo captures and distributes a UDB based density integrator. This component takes a density stream, generated from a DeltaSigma modulator or some other source, and produces an output density stream that is the time integral of the input density stream. The component features adjustable integration gain and automatic saturation at 100% and 0% output. See the body of the memo for examples of the integrated output in case it's not clear what the component does.

Attached File Summary:

File # 2 is the component exported using Creator 2.2's component export feature. To use the file, download it and remove the .PDF extension and replace it with .ZIP. The zip file contains the cycomp file

File # 3 is a bundle of the example project containing all the components necessary to generate the waveforms in this memo. To use the file, download it and remove the .PDF extension. The proper extension for this file should be .ZIP

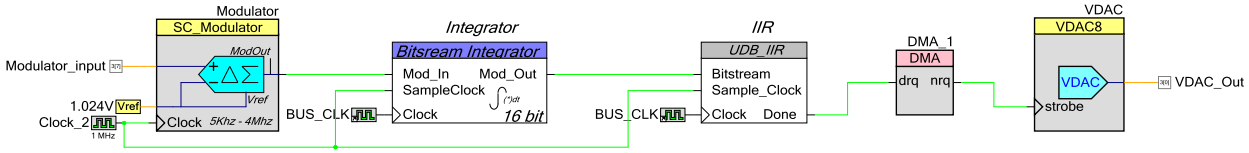
File # 4 is a PDF of the datapath instructions

Details:

This component will perform integration on the input density stream, and produce an output density stream equivalent to the time integral of the input. The integrator features an adjustable integration gain and a 16 bit integration width, allowing for output densities that can have up to 16 bits of resolution. The integration will automatically saturate at 100% and 0%, preventing roll over

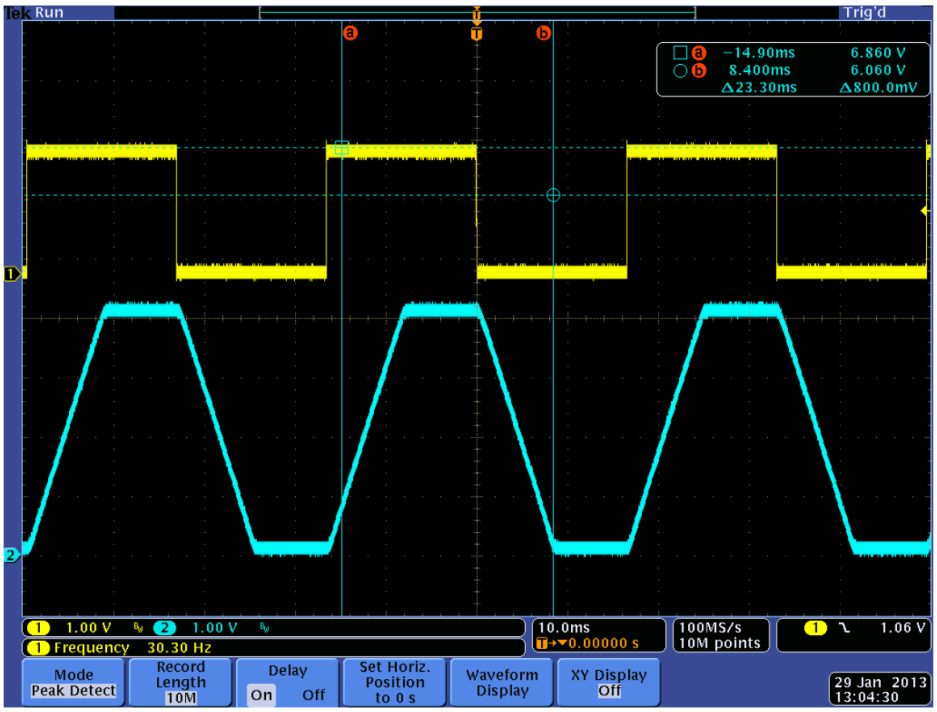
problems and simulating a real integrator with finite rails. _Start() is the only API call that is necessary to use the component.

To best understand the integrator, it is helpful to see some waveforms. The following waveforms were generated using the following creator schematic:

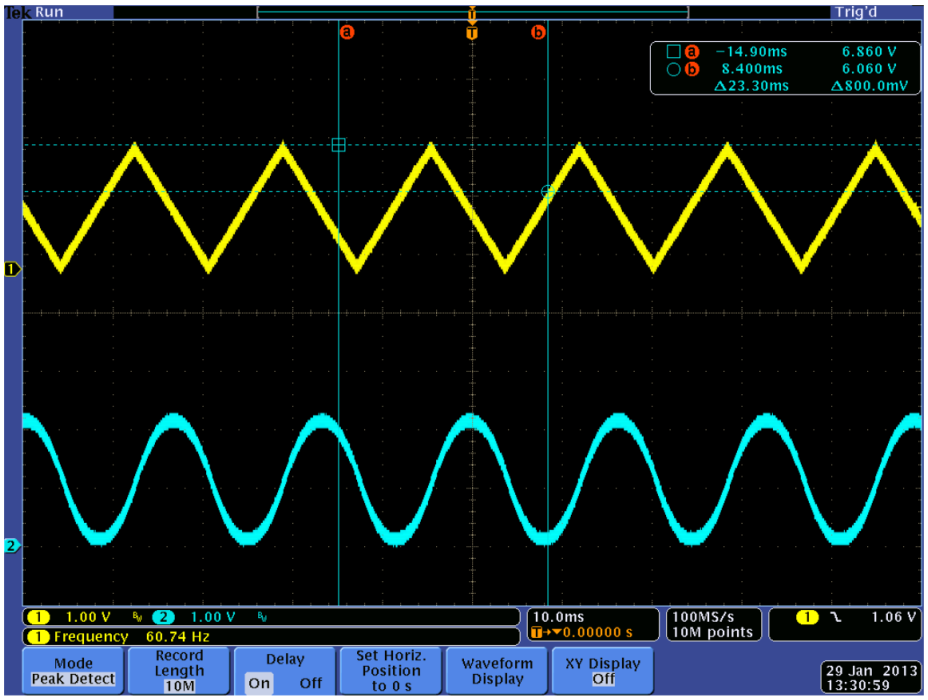


The IIR component is used to convert the density into a number that can be directly DMA'd to the VDAC, making it simple to use an oscilloscope to observe the input and output. The -3 dB point of the IIR filter is ~ 2.5 Khz, meaning its contribution to phase and amplitude is negligible for frequencies below 250 Hz.

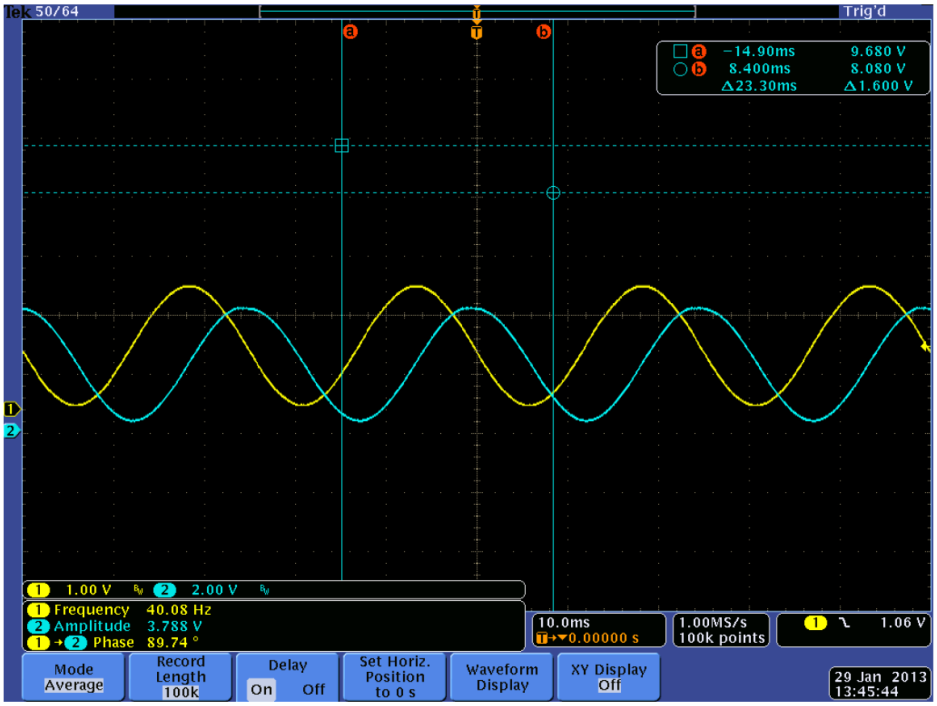
We will start with a simple square wave, showing the integration of a constant and the saturation behavior. The yellow trace is the signal input, and the blue trace is the integrator output. Note that the modulator is set to generate a "zero" signal when the input is at ~ 1.024 volts (Vref +/- 2*Vref). This means that signals above 1.024 will cause positive integration, and signals below 1.024 volts will result in negative integration. All of the generated signals are given an almost infinitesimal negative bias compared to this analog ground, resulting in all output waveforms being pushed toward the lower integration limit.



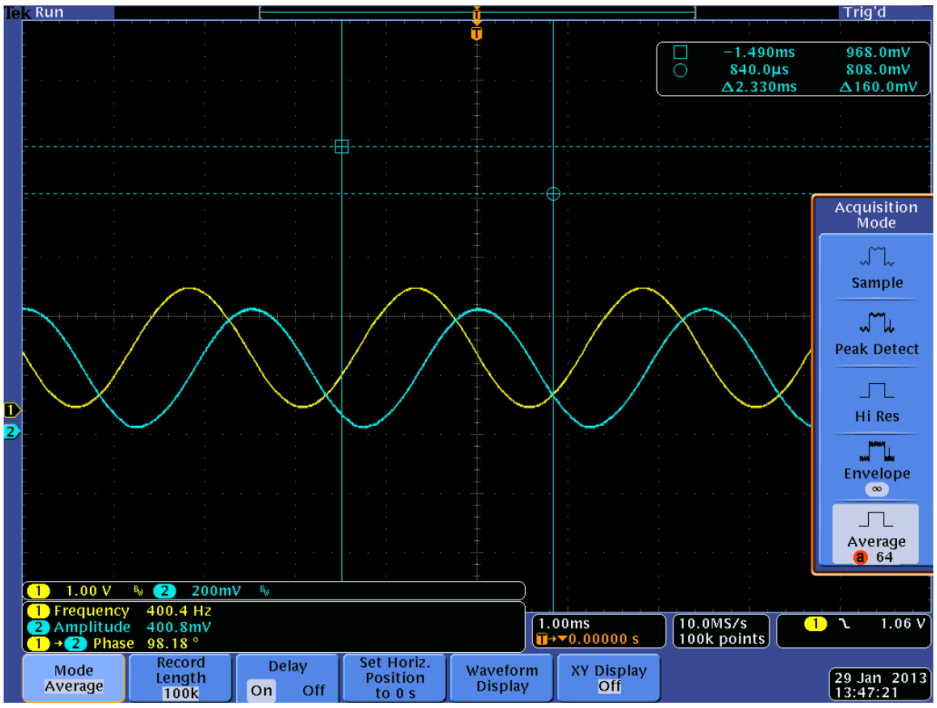
Here is the integrator integrating a triangle wave. The integral of a ramp (y = x) is a polynomial (y = x^2/2).



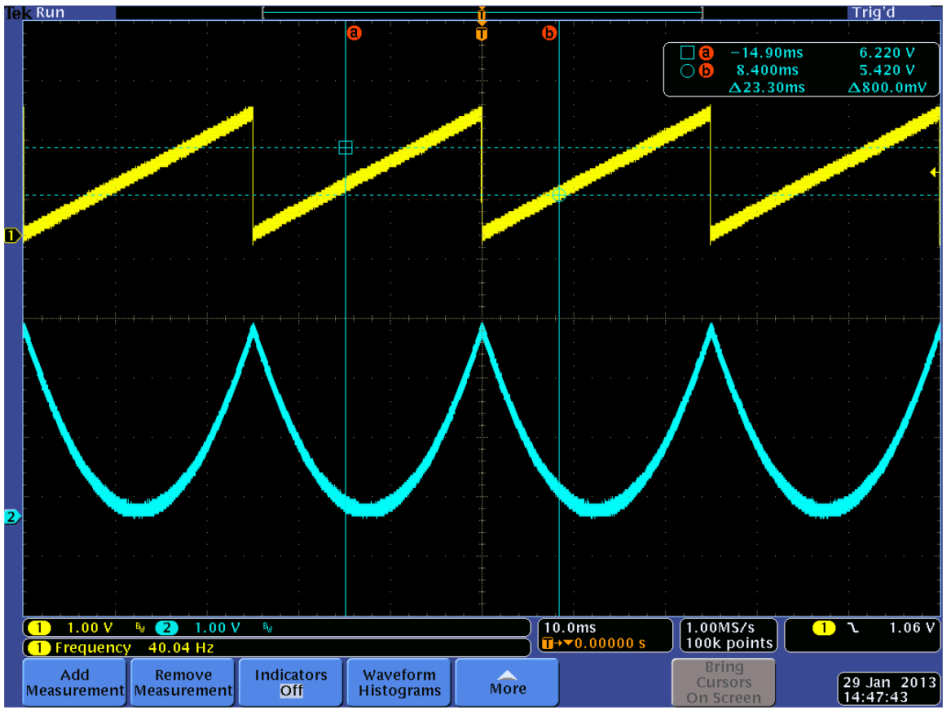
A frequency domain characteristic of an integrator is a constant 90 degree phase shift and a -20 dB / decade reduction in amplitude: Here is a sinewave at 40 Hz



The same sinewave at 400 Hz has ~1/10th the amplitude (20 dB reduction in amplitude) and a ~90 degree phase shift. Note* the IIR filter is introducing some extra phase at this frequency:



Another interesting input waveform is the sawtooth:



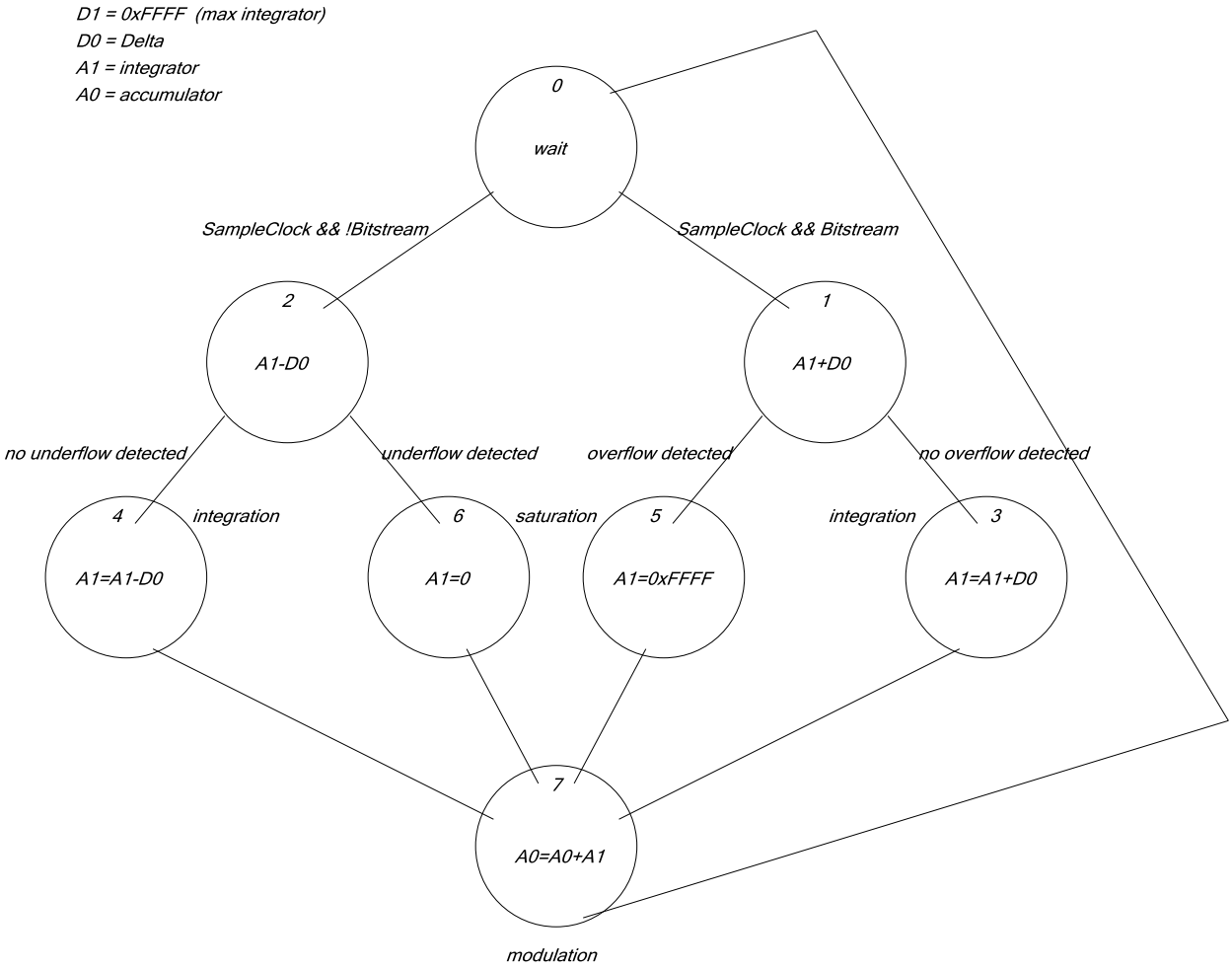
The integrator works using two pieces. DWV explained how a simple digital DeltaSigma modulator works by accumulating the desired output into a register, and using the carry to generate the density stream. For example, let's say we have an 8 bit accumulator, and we add the number 50 to it over and over. The accumulator will increase like so, and the carry output would be set as indicated in the 'output' row:

Accum	**50**	**100**	**150**	**200**	**250**	**44**	**94**	**144**	**194**	**244**	**38**	**…**
Output	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**…**

If we add the resulting carry output over 100 cycles, we would find that it would be high 19 times out of 100 resulting in an average of 0.19. If we do it 10,000 times, it would be high 1952 times out of 10,000 resulting in an average of .1952. The exact result of 50 / 256 is 0.1953125. The value added to the accumulator produces a density that is equal to the ratio of the accumulated value and the maximum number of counts before a rollover of the accumulator. So to generate a simple digital delta sigma modulator in a datapath, you just need to add D0 (your density modulation value) into some accumulation register, A0, and the carry out signal is your density stream. i.e. A0 = A0 + D0, and repeat. To change the density, change D0. I took this idea and modified it and instead of having a constant D0, I allowed the input density stream to alter the constant. If the input density signal is high, increase 'D0' by some amount, and if it's low, decrease 'D0' by some amount. The result is that the input density stream is integrated, and that integrated number is used to generate another density output stream. Saturation is added to the 'D0' value so it would not overflow or underflow, and so the instructions can be broken down into this very simple representation (ignoring the adjustable integration constant and the saturation limit):

```
If input == 1
        A1 = A1 + 1
Else
        A1 = A1 – 1
A0 = A0 + A1
```

The result of the A0 = A0 + A1 operation generates the carry out that is the new integrated density stream. The following state diagram explains the datapath execution in more detail. The value accumulated in A1 can be arbitrary (Delta = IntegrationGain in the component), allowing for adjustable integration gain.


D1 = 0xFFFF (max integrator)
D0 = Delta
A1 = integrator
A0 = accumulator
0
wait
SampleClock && !Bitstream
SampleClock && Bitstream
2
A1-D0
1
A1+D0
no underflow detected
underflow detected
overflow detected
no overflow detected
4
integration
A1=A1-D0
6
saturation
A1=0
5
A1=0xFFFF
integration
3
A1=A1+D0
7
A0=A0+A1
modulation

Appendix: Verilog code

```
//`#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 01/24/2013 at 16:14
// Component: KEES UDB BitstreamIntegrator v1 100
module KEES_UDB_BitstreamIntegrator_v1_10 (
    output  Mod_Out,
    output  State_0,
    output  State_1,
    output  State_2,
    output  CarryOut,
    input   Clock,
    input   Mod_In,
    input   SampleClock
);

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

parameter IntegrationConstant = "16'h01";

localparam STATE_0 = 3'h0;
localparam STATE_1 = 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;
reg SampleClock_Reg_1;
reg SampleClock_Reg_2;
reg mod_out;

wire Carry;
wire SampleClock_Edge;
wire [1:0] temp;

// rising edge detect of SampleClock
assign SampleClock_Edge = ((SampleClock_Reg_1 == 1) && (SampleClock_Reg_2 == 0)) ? 1 : 0;
assign Mod_Out = mod_out;
assign CarryOut = Carry;
assign State_0 = State[0];
assign State_1 = State[1];
assign State_2 = State[2];
assign Carry = temp[1];

always @ (posedge Clock)
begin

    SampleClock Reg 1 <= SampleClock;
    SampleClock_Reg_2 <= SampleClock_Reg_1;
    Done <= 0;
    mod out <= mod out;
    //TC <= tc;

    case(State)

        STATE_0:
        begin
```

```
            if(SampleClock_Edge == 1 && Mod_In == 1)
            begin
                State <= STATE_1;  // positive input (and we care)
            end
            else if(SampleClock_Edge == 1 && Mod_In == 0)
            begin
                State <= STATE_2; // negative input (and we care)
            end
            else if(SampleClock_Edge == 1)
            begin
                State <= STATE_7;  // ignoring input, just accumulate
            end
            else
            begin
                State <= STATE_0;
            end
        end

        STATE_1:
        begin
            if(Carry == 1)
            begin
                State <= STATE_5;  // overflow, saturate FF
            end
            else
            begin
                State <= STATE_3; // safe to add
            end
        end

        STATE_2:
        begin
            if(Carry == 0)
            begin
                State <= STATE_6; // underflow, saturate 0
            end
            else
            begin
                State <= STATE_4; // safe to subtract
            end
        end

        STATE_3:
        begin
            State <= STATE_7; // add D0 to A1 and store in A1
        end

        STATE_4:
        begin
            State <= STATE_7;  // subtract D0 from A1 and store in A1
        end

        STATE_5:
        begin
            State <= STATE_7; // saturate FF and store in A1
        end

        STATE_6:
        begin
            State <= STATE_7; // saturate 0 and store in A1
        end

        STATE_7:
        begin
            State <= STATE_0; // add A1 to A0 and store in A0, capture Carry Out as mod out
            mod_out <= Carry;
            Done <= 1;
        end

    endcase
end

cy_psoc3_dp16 #(.d0_init_a(8'h01), .d1_init_a(8'hFF), .d0_init_b(8'h00),
.d1_init_b(8'hFF),
```

```
.cy_dpconfig_a(
{
    `CS_ALU_OP_PASS, `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, /*CFGRAM0:    wait*/
    `CS_ALU_OP__ADD, `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, /*CFGRAM1:    A1+D0 (overflow test)*/
    `CS_ALU_OP__SUB, `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, /*CFGRAM2:    A1-D0 (underflow test)*/
    `CS_ALU_OP__ADD, `CS_SRCA_A1, `CS_SRCB_D0,
    `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, /*CFGRAM3:    A1=A1+D0*/
    `CS_ALU_OP__SUB, `CS_SRCA_A1, `CS_SRCB_D0,
    `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, /*CFGRAM4:    A1=A1-D0*/
    `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_A0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC__D1,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /*CFGRAM5:    A1=D1 (saturate FF)*/
    `CS_ALU_OP__XOR, `CS_SRCA_A1, `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, /*CFGRAM6:    A1=A1^A1 (saturate 0)*/
    `CS_ALU_OP__ADD, `CS_SRCA_A0, `CS_SRCB_A1,
    `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, /*CFGRAM7:    A0=A0+A1*/
    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_BUS, `SC_FIFO0_BUS,
    `SC_MSB_DSBL, `SC_MSB_BIT0, `SC_MSB_CHNED,
    `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_PASS, `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, /*CFGRAM0:    wait*/
    `CS_ALU_OP__ADD, `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, /*CFGRAM1:    A1+D0 (overflow test)*/
    `CS_ALU_OP__SUB, `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, /*CFGRAM2:    A1-D0 (underflow test)*/
    `CS_ALU_OP__ADD, `CS_SRCA_A1, `CS_SRCB_D0,
    `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, /*CFGRAM3:    A1=A1+D0*/
    `CS_ALU_OP__SUB, `CS_SRCA_A1, `CS_SRCB_D0,
    `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, /*CFGRAM4:    A1=A1-D0*/
    `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_A0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC__D1,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /*CFGRAM5:    A1=D1 (saturate FF)*/
    `CS_ALU_OP__XOR, `CS_SRCA_A1, `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, /*CFGRAM6:    A1=A1^A1 (saturate 0)*/
    `CS ALU OP  ADD, `CS SRCA A0, `CS SRCB A1,
    `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, /*CFGRAM7:    A0=A0+A1*/
    8'hFF, 8'h00,  /*CFG9:    */
    8'hFF, 8'hFF,  /*CFG11-10:    */
    `SC_CMPB_A1_D1, `SC_CMPA_A1_D1, `SC_CI_B_CHAIN,
    `SC CI A CHAIN, `SC C1 MASK DSBL, `SC C0 MASK DSBL,
    `SC_A_MASK_DSBL, `SC_DEF_SI_0, `SC_SI_B_CHAIN,
    `SC_SI_A_CHAIN, /*CFG13-12:    */
    `SC_A0_SRC_ACC, `SC_SHIFT_SL, 1'h0,
    1'h0, `SC FIFO1 BUS, `SC FIFO0 BUS,
    `SC_MSB_DSBL, `SC_MSB_BIT0, `SC_MSB_NOCHN,
    `SC_FB_CHNED, `SC_CMP1_CHNED,
    `SC CMP0 CHNED, /*CFG15-14:    */
    10'h00, `SC_FIFO_CLK__DP,`SC_FIFO__SYNC,`SC_EXTCRC_DSBL,
    `SC_WRK16CAT_DSBL /*CFG17-16:    */
}
)) Integrator16(
        /*  input                   */  .reset(1'b0),
        /*  input                   */  .clk(Clock),
        /*  input   [02:00]         */  .cs addr(State),
        /*  input                   */  .route_si(1'b0),
        /*  input                   */  .route_ci(1'b0),
        /*  input                   */  .f0 load(1'b0),
        /*  input                   */  .f1_load(1'b0),
        /*  input                   */  .d0_load(1'b0),
        /*  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(temp),
        /*  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(),
        /*  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
```