Caravel DLL

—

TonyHo

20230320

Reference

- caravel documents
  - https://github.com/efabless/caravel/blob/main/docs/pdf/caravel_block_diagram.pdf
  - https://github.com/efabless/caravel/blob/main/docs/pdf/housekeeping_function.pdf
  - https://github.com/efabless/caravel/blob/main/docs/pdf/caravel_clocking.pdf
  - https://github.com/efabless/caravel/blob/31e3d9c1063d4ec987760fb34579e123c9196e8d/docs/other/digital_locked_loop.txt
  - https://caravel-mgmt-soc-litex.readthedocs.io/en/latest/
- thermometer code

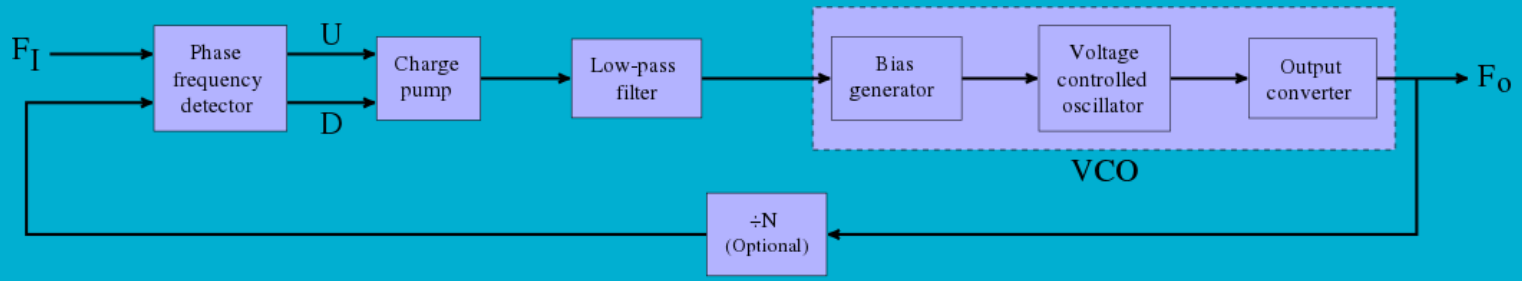
Reference - What is DLL

- https://en.wikipedia.org/wiki/Ring_oscillator
- https://en.wikipedia.org/wiki/Delay-locked_loop
- https://en.wikipedia.org/wiki/Phase-locked_loop

terminology

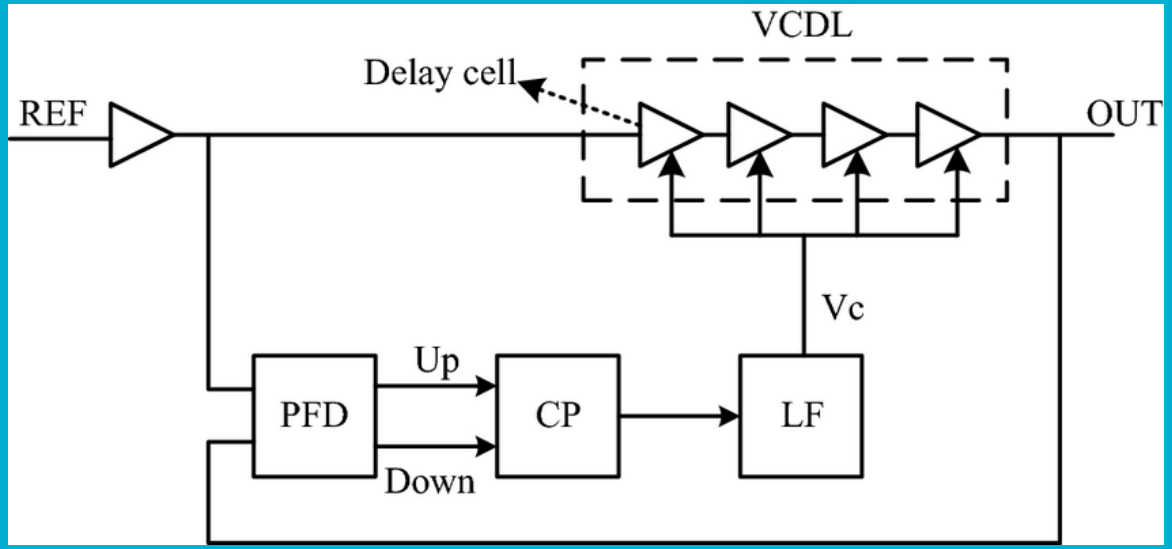
- phase locked loop (PLL)
- delay locked loop (DLL)
- digital locked loop (DLL) <- DLL in slide means digital locked loop
- Ring_oscillator

phase locked loop



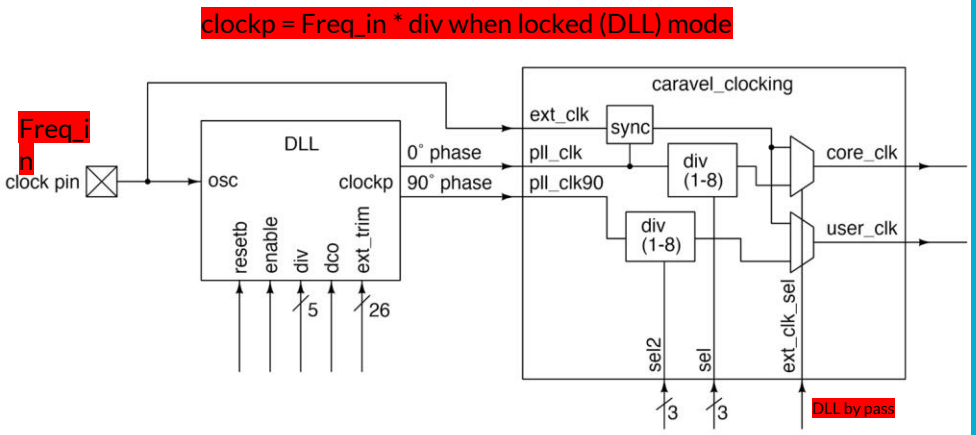
#60: Basics of Phase Locked Loop Circuits and Frequency Synthesis

delay locked loop



DLL(digital locked loop) Block Diagram

clockp = Freq_in * div when locked (DLL) mode



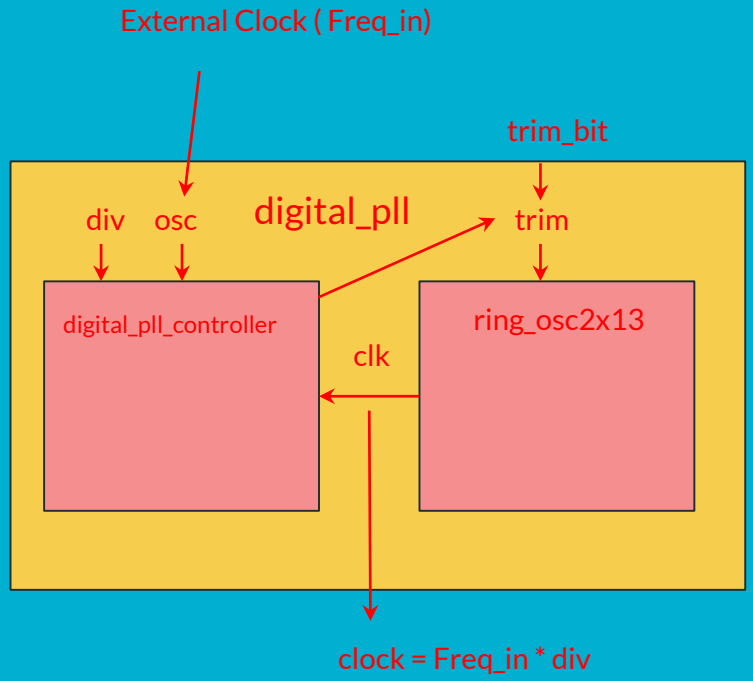
From Hossekeeping SPI register

Housekeeping SPI register map

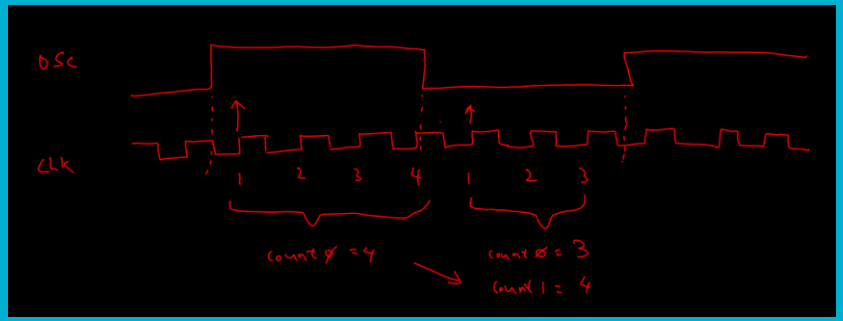
Register Address	msb 7	6	5	4	3	2	1	lsb 0	comments
0x00	SPI status and control								unused/ undefined
0x01	unused				manufacturer_ID[11:8] (= 0x4)				read-only
0x02	manufacturer_ID[7:0] (= 0x56)								read-only
0x03	product_ID (= 0x11)								read-only
0x04–0x07	user_project_ID (unique value per project)								read-only
0x08	unused						DLL DCO enable	DLL enable	default 0x02
0x09	unused							DLL bypass	default 0x01
0x0A	unused							CPU IRQ	default 0x00
0x0B	unused							CPU reset	default 0x00
0x0C	unused							CPU trap	read-only
0x0D–0x10	DCO trim (26 bits) (= 0x3ffefff)								default 0x3ffefff
0x11	unused		PLL output divider 2			PLL output divider			default 0x12
0x12	unused				PLL feedback divider				default 0x04
0x13		serial data 2	serial data 1	serial clock	serial load	serial reset	serial enable	serial xfer/ busy*	bits 6 to 1 are bit-bang control.

* Bit is write-only for serial transfer, read-only for serial busy. During transfer, the busy bit is one. Transfer is complete when the busy bit returns to zero.

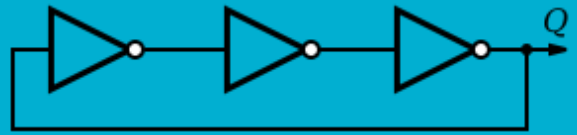
Caravel digital locked loop (DLL)



tracking the freq from osc, but no phase lock



ring oscillator



Code

- https://github.com/efabless/caravel
  - commit edc22cc32547527861a12aeddb568f1ae591cc92 (HEAD -> main, tag: mpw-9a, origin/main, origin/HEAD)
  - Author: Jeff DiCorpo <42048757+jeffdi@users.noreply.github.com>
  - Date: Sat Mar 4 17:37:44 2023 -0800

  - update tag to mpw-9a

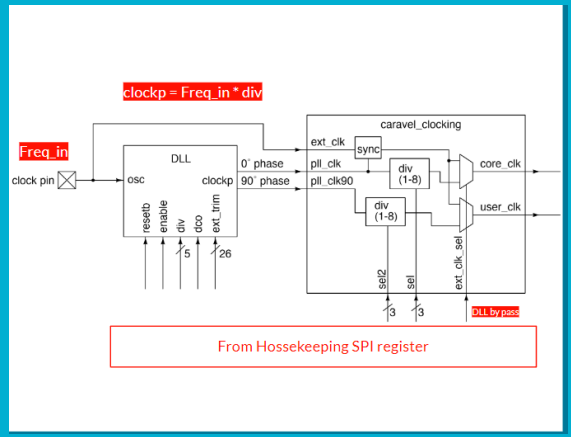
Files

```
ubuntu@ubuntu2004:~/workspace/opensilicon/caravel/verilog/rtl$ tree
.
├── __uprj_analog_netlists.v
├── __uprj_netlists.v
├── __user_analog_project_wrapper.v
├── __user_project_gpio_example.v
├── __user_project_la_example.v
├── __user_project_wrapper.v
├── buff_flash_clkrst.v
├── caravan.v
├── caravan_netlists.v
├── caravan_openframe.v
├── caravan_power_routing.v
├── caravel.v
├── caravel_clocking.v
├── caravel_logo.v
├── caravel_motto.v
├── caravel_netlists.v
├── caravel_openframe.v
├── caravel_power_routing.v
├── chip_io.v
├── chip_io_alt.v
├── clock_div.v
├── constant_block.v
├── copyright_block.v
├── debug_regs.v
├── defines.v
├── digital_pll.v
├── digital_pll_controller.v
├── gpio_control_block.v
├── gpio_defaults_block.v
├── gpio_logic_high.v
├── gpio_signal_buffering.v
├── gpio_signal_buffering_alt.v
├── housekeeping.v
├── housekeeping_spi.v
├── mgmt_protect.v
├── mgmt_protect_hv.v
├── mprj2_logic_high.v
├── mprj_io.v
├── mprj_logic_high.v
├── open_source.v
├── pads.v
├── ring_osc2x13.v
├── simple_por.v
├── spare_logic_block.v
├── user_defines.v
├── user_id_programming.v
├── user_id_textblock.v
└── xres_buf.v

0 directories, 48 files
```

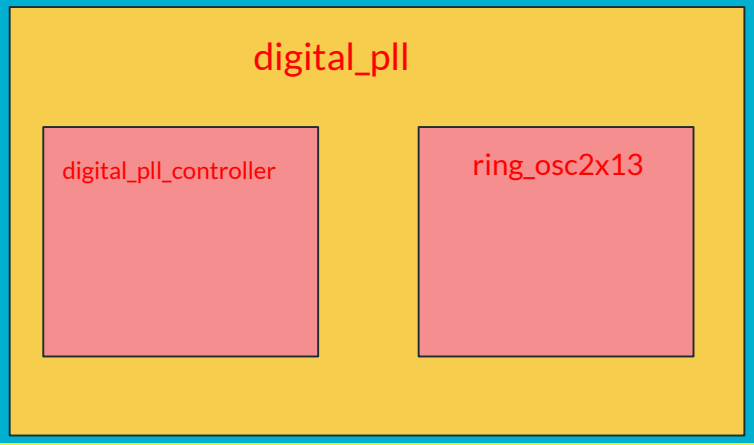
Caravel digital locked loop features

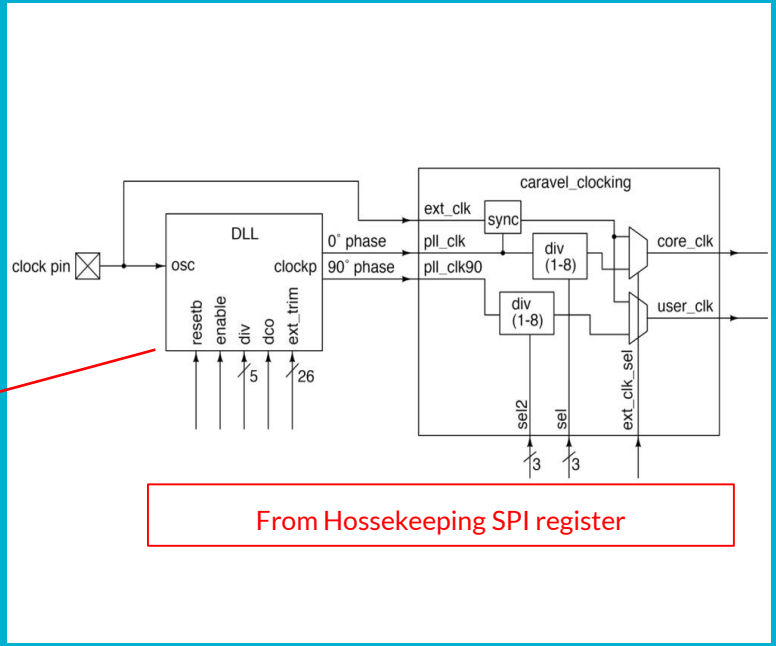
- The Caravel digital locked loop (DLL) is an all-digital clock generating module.
- The GPIO pins on Caravel have a limit of 50MHz input.
- Internally, it is possible to generate a clean oscillation of up to around 200MHz ?? or higher.
- To ensure large margins of safety, the Caravel demonstration board ships with an on-board oscillator of 10MHz.
- The operational frequency of the management SoC on Caravel differs according to which management SoC architecture is present, but is generally in the range of around 50MHz.



Caravel digital locked loop features

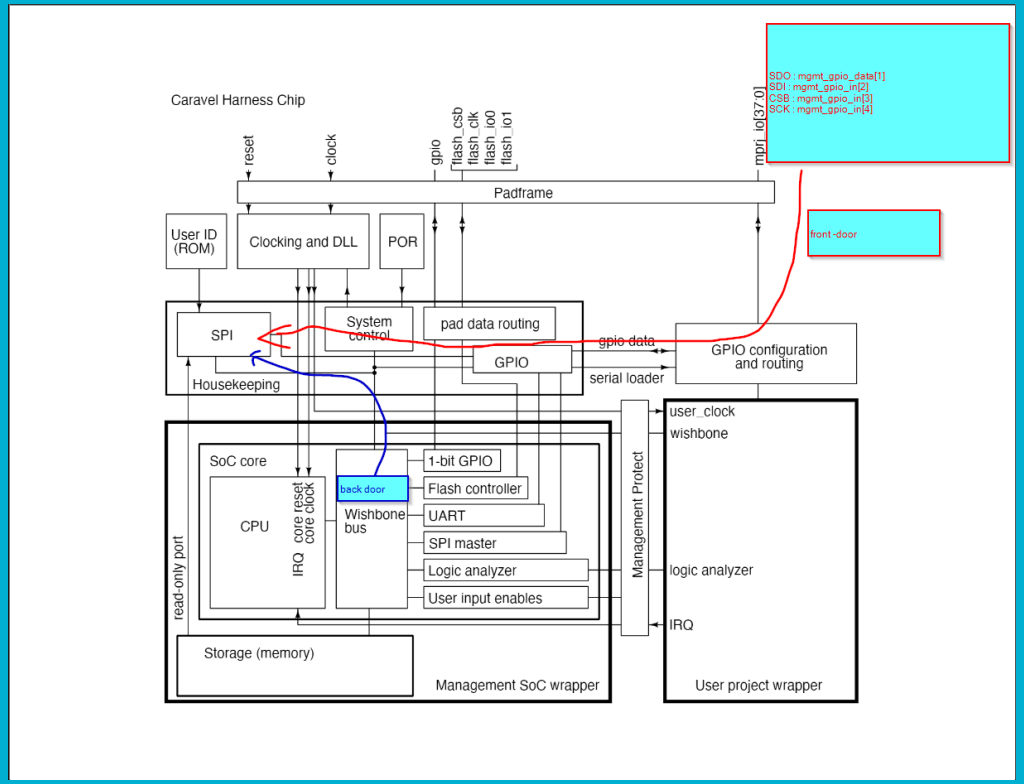
- DLL = on-chip tunable ring oscillator + feedback controller
- 2 operation modes for DLL
  - free-running (DCO) - directly off of the external clock (bypass mode)
  - or locked (DLL) mode
- Note:
  - DCO is a digitally controlled oscillator
  - VCO is a voltage controlled oscillator





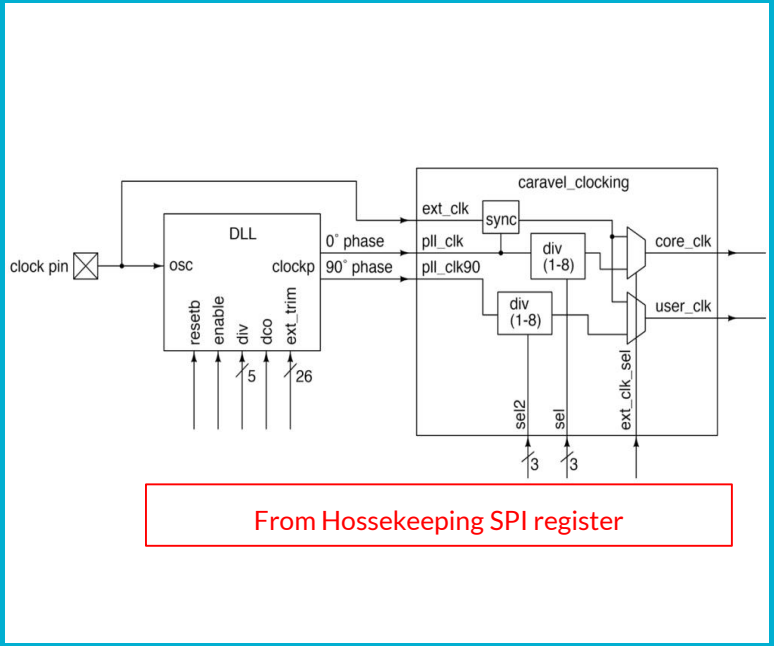
Hossekeeping SPI register access

- front door
  - external host access spi_slave by SCK/CSB/SDI/SDO
  - housekeeping_spi reset by porb=0.
    - Note: it can work when soc_core is reset.
- back door
  - soc_core issue request from wishbone to spi_slave



Caravel digital locked loop features

- The DLL's tunable oscillator has an operating range of approximately 75MHz to 150MHz. The oscillator is a loop of from 13 to 39 inverter stages with 26 bits of trim. Each trim bit adds or subtracts one of the stages. So there are 27 effective frequency steps covering a range of about 75MHz, with an incremental delay of about 250ps per step.
- WARNING: The management SoC altering its own clock has not yet been tested as of this writing; however, the core clock should be guaranteed to be glitch-free through transistions from external clock to DLL output and vice versa.
    - glitch-free ???



The DLL controls are memory-mapped to the housekeeping space

Register name = reg_hkspi_pll_ena
Memory location = 0x2610000c
Housekeeping SPI location = 0x08

	7	6	5	4	3	2	1	0
0x09							DCO ena	DLL ena

bit 1: DCO enable
value 0 = DCO disabled. DLL runs in active locking mode
value 1 = DCO enabled. DLL runs in DCO mode.

bit 0: DLL enable
value 0 = DLL disabled. DLL is disabled and the clock is stopped.
value 1 = DLL enabled. DLL is enabled and outputs a clock.

DLL ena connect to enable of digital_pll, it will reset digital_pll_controller when bit 0 = 0

Front door and back door select

```
297     wire [7:0] caddr; // Combination of SPI address and back door address
298     wire [7:0] cdata; // Combination of SPI data and back door data
299     wire cwstb; // Combination of SPI write strobe and back door write strobe
300     wire csclk; // Combination of SPI SCK and back door access trigger
```

```
1075    // SPI Data transfer protocol.  The wishbone back door may only be
1076    // used if the front door is closed (CSB is high or the CSB pin is
1077    // not an input).  The time to apply values for the back door access
1078    // is limited to the clock cycle around the read or write from the
1079    // wbbd state machine (see below).
1080
1081    assign caddr = (wbbd_busy) ? wbbd_addr : iaddr;
1082    assign csclk = (wbbd_busy) ? wbbd_sck : ((spi_is_active) ? mgmt_gpio_in[4] : 1'b0);
1083    assign cdata = (wbbd_busy) ? wbbd_data : idata;
1084    assign cwstb = (wbbd_busy) ? wbbd_write : wrstb;
1085
1086    assign odata = fdata(caddr);
```

write enable

```
1154     if (cwstb == 1'b1) begin
1155             case (caddr)
1156         /* Register 8'h00 is reserved for future use */
1157         /* Registers 8'h01 to 8'h07 are read-only and cannot be written */
1158             8'h08: begin
1159             pll_ena <= cdata[0];
1160             pll_dco_ena <= cdata[1];
1161             end
1162             8'h09: begin
1163             pll_bypass <= cdata[0];
1164             end
1165             8'h0a: begin
1166             irq_spi <= cdata[0];
1167             end
1168             8'h0b: begin
1169             reset_reg <= cdata[0];
1170             end
```

write to register

Housekeeping SPI register map

Register Address	msb 7	6	5	4	3	2	1	lsb 0	comments
0x00	SPI status and control								unused/ undefined
0x01	unused				manufacturer_ID[11:8] (= 0x4)				read-only
0x02	manufacturer_ID[7:0] (= 0x56)								read-only
0x03	product_ID (= 0x11)								read-only
0x04–0x07	user_project_ID (unique value per project)								read-only
0x08	unused						DLL DCO enable	DLL enable	default 0x02
0x09	unused							DLL bypass	default 0x01
0x0A	unused							CPU IRQ	default 0x00
0x0B	unused							CPU reset	default 0x00
0x0C	unused							CPU trap	read-only
0x0D–0x10	DCO trim (26 bits) (= 0x3ffefff)								default 0x3ffefff
0x11	unused		PLL output divider 2			PLL output divider			default 0x12
0x12	unused				PLL feedback divider				default 0x04
0x13		serial data 2	serial data 1	serial clock	serial load	serial reset	serial enable	serial xfer/ busy*	bits 6 to 1 are bit-bang control.

* Bit is write-only for serial transfer, read-only for serial busy. During transfer, the busy bit is one. Transfer is complete when the busy bit returns to zero.

Front door

```
813     // Instantiate the SPI interface protocol module
814
815     housekeeping_spi hkspi (
816 .reset(~porb),
817       .SCK(mgmt_gpio_in[4]),
818       .SDI(mgmt_gpio_in[2]),
819       .CSB((spi_is_enabled) ? mgmt_gpio_in[3] : 1'b1),
820       .SDO(sdo),
821       .sdoenb(sdo_enb),
822       .idata(odata),
823       .odata(idata),
824       .oaddr(iaddr),
825       .rdstb(rdstb),
826       .wrstb(wrstb),
827       .pass_thru_mgmt(pass_thru_mgmt),
828       .pass_thru_mgmt_delay(pass_thru_mgmt_delay),
829       .pass_thru_user(pass_thru_user),
830       .pass_thru_user_delay(pass_thru_user_delay),
831       .pass_thru_mgmt_reset(pass_thru_mgmt_reset),
832       .pass_thru_user_reset(pass_thru_user_reset)
833     );
```

reset by POR

SPI interface

R/W address to housekeeping SPI register

Wishbone back-door state machine

```
403     assign sys_select = (wb_adr_i[31:8] == SYS_BASE_ADR[31:8]);
404     assign gpio_select = (wb_adr_i[31:8] == GPIO_BASE_ADR[31:8]);
405     assign spi_select = (wb_adr_i[31:8] == SPI_BASE_ADR[31:8]);
```

```
102 module housekeeping #(
103     parameter GPIO_BASE_ADR = 32'h2600_0000,
104     parameter SPI_BASE_ADR = 32'h2610_0000,
105     parameter SYS_BASE_ADR = 32'h2620_0000,
106     parameter IO_CTRL_BITS = 13
107 ) (
```

```
705     /* Wishbone back-door state machine and address translation */
706
707     always @(posedge wb_clk_i or posedge wb_rst_i) begin
708  if (wb_rst_i) begin
709      wbbd_sck <= 1'b0;
710      wbbd_write <= 1'b0;
711      wbbd_addr <= 8'd0;
712      wbbd_data <= 8'd0;
713      wbbd_busy <= 1'b0;
714      wb_ack_o <= 1'b0;
715      wbbd_state <= `WBBD_IDLE;
716  end else begin
717      case (wbbd_state)
718   `WBBD_IDLE: begin
719       wbbd_busy <= 1'b0;
720       if ((sys_select | gpio_select | spi_select) &&
721             wb_cyc_i && wb_stb_i) begin
722    wb_ack_o <= 1'b0;
723    wbbd_state <= `WBBD_SETUP0;
724       end
725   end
726   `WBBD_SETUP0: begin
727       wbbd_sck <= 1'b0;
728       wbbd_addr <= spiaddr(wb_adr_i);
729       if (wb_sel_i[0] & wb_we_i) begin
730        wbbd_data <= wb_dat_i[7:0];
731       end
732       wbbd_write <= wb_sel_i[0] & wb_we_i;
733       wbbd_busy <= 1'b1;
734
735       // If the SPI is being accessed and about to read or
736       // write a byte, then stall until the SPI is ready.
737       if (!spi_is_busy) begin
738           wbbd_state <= `WBBD_RW0;
739       end
740   end
```

transfer to housekeeping SPI register offset

```
741   `WBBD_RW0: begin
742       wbbd_busy <= 1'b1;
743       wbbd_sck <= 1'b1;
744       wb_dat_o[7:0] <= odata;
745       wbbd_state <= `WBBD_SETUP1;
746   end
747   `WBBD_SETUP1: begin
748       wbbd_busy <= 1'b1;
749       wbbd_sck <= 1'b0;
750       wbbd_addr <= spiaddr(wb_adr_i + 1);
751       if (wb_sel_i[1] & wb_we_i) begin
752        wbbd_data <= wb_dat_i[15:8];
753       end
754       wbbd_write <= wb_sel_i[1] & wb_we_i;
755       if (!spi_is_busy) begin
756           wbbd_state <= `WBBD_RW1;
757       end
758   end
759   `WBBD_RW1: begin
760       wbbd_busy <= 1'b1;
761       wbbd_sck <= 1'b1;
762       wb_dat_o[15:8] <= odata;
763       wbbd_state <= `WBBD_SETUP2;
765   `WBBD_SETUP2: begin
766       wbbd_busy <= 1'b1;
767       wbbd_sck <= 1'b0;
768       wbbd_addr <= spiaddr(wb_adr_i + 2);
769       if (wb_sel_i[2] & wb_we_i) begin
770        wbbd_data <= wb_dat_i[23:16];
771       end
772       wbbd_write <= wb_sel_i[2] & wb_we_i;
773       if (!spi_is_busy) begin
774           wbbd_state <= `WBBD_RW2;
775       end
776   end
```

如何決定 wbbd_addr from which source?

```
557     /* Memory map address to SPI address translation for back door access */
558     /* (see doc/memory_map.txt)                                           */
559
560     wire [11:0] gpio_adr = GPIO_BASE_ADR[23:12];
561     wire [11:0] sys_adr = SYS_BASE_ADR[23:12];
562     wire [11:0] spi_adr = SPI_BASE_ADR[23:12];
```

```
102 module housekeeping #(
103     parameter GPIO_BASE_ADR = 32'h2600_0000,
104     parameter SPI_BASE_ADR = 32'h2610_0000,
105     parameter SYS_BASE_ADR = 32'h2620_0000,
106     parameter IO_CTRL_BITS = 13
107 ) (
```

```
564     function [7:0] spiaddr(input [31:0] wbaddress);
565 begin
566 /* Address taken from lower 8 bits and upper 4 bits of the 32-bit */
567 /* wishbone address.                                              */
568 case ({wbaddress[23:20], wbaddress[7:0]})
569     spi_adr | 12'h000 : spiaddr = 8'h00; // SPI status (reserved)
570     spi_adr | 12'h004 : spiaddr = 8'h03; // product ID
571     spi_adr | 12'h005 : spiaddr = 8'h02; // Manufacturer ID (low)
572     spi_adr | 12'h006 : spiaddr = 8'h01; // Manufacturer ID (high)
573     spi_adr | 12'h008 : spiaddr = 8'h07; // User project ID (low)
574     spi_adr | 12'h009 : spiaddr = 8'h06; // User project ID .
575     spi_adr | 12'h00a : spiaddr = 8'h05; // User project ID .
576     spi_adr | 12'h00b : spiaddr = 8'h04; // User project ID (high)
577
578     spi_adr | 12'h00c : spiaddr = 8'h08; // PLL enables
579     spi_adr | 12'h010 : spiaddr = 8'h09; // PLL bypass
580     spi_adr | 12'h014 : spiaddr = 8'h0a; // IRQ
581     spi_adr | 12'h018 : spiaddr = 8'h0b; // Reset
582     spi_adr | 12'h028 : spiaddr = 8'h0c; // CPU trap state
583     spi_adr | 12'h01f : spiaddr = 8'h10; // PLL trim
584     spi_adr | 12'h01e : spiaddr = 8'h0f; // PLL trim
585     spi_adr | 12'h01d : spiaddr = 8'h0e; // PLL trim
586     spi_adr | 12'h01c : spiaddr = 8'h0d; // PLL trim
587     spi_adr | 12'h020 : spiaddr = 8'h11; // PLL source
```

transfer to housekeeping SPI register offset

```
596
597     gpio_adr | 12'h000 : spiaddr = 8'h13; // GPIO control
598
599     sys_adr | 12'h000 : spiaddr = 8'h1a; // Power monitor
600     sys_adr | 12'h004 : spiaddr = 8'h1b; // Output redirect
601     sys_adr | 12'h00c : spiaddr = 8'h1c; // Input redirect
602
603     gpio_adr | 12'h025 : spiaddr = 8'h1d; // GPIO configuration
604     gpio_adr | 12'h024 : spiaddr = 8'h1e;
605     gpio_adr | 12'h029 : spiaddr = 8'h1f;
606     gpio_adr | 12'h028 : spiaddr = 8'h20;
607     gpio_adr | 12'h02d : spiaddr = 8'h21;
608     gpio_adr | 12'h02c : spiaddr = 8'h22;
609     gpio_adr | 12'h031 : spiaddr = 8'h23;
610     gpio_adr | 12'h030 : spiaddr = 8'h24;
611     gpio_adr | 12'h035 : spiaddr = 8'h25;
612     gpio_adr | 12'h034 : spiaddr = 8'h26;
613     gpio_adr | 12'h039 : spiaddr = 8'h27;
614     gpio_adr | 12'h038 : spiaddr = 8'h28;
615     gpio_adr | 12'h03d : spiaddr = 8'h29;
616     gpio_adr | 12'h03c : spiaddr = 8'h2a;
617     gpio_adr | 12'h041 : spiaddr = 8'h2b;
618     gpio_adr | 12'h040 : spiaddr = 8'h2c;
619     gpio_adr | 12'h045 : spiaddr = 8'h2d;
```

fdata for read register

```
1075    // SPI Data transfer protocol.  The wishbone back door may only be
1076    // used if the front door is closed (CSB is high or the CSB pin is
1077    // not an input).  The time to apply values for the back door access
1078    // is limited to the clock cycle around the read or write from the
1079    // wbbd state machine (see below).
1080
1081    assign caddr = (wbbd_busy) ? wbbd_addr : iaddr;
1082    assign csclk = (wbbd_busy) ? wbbd_sck : ((spi_is_active) ? mgmt_gpio_in[4] : 1'b0);
1083    assign cdata = (wbbd_busy) ? wbbd_data : idata;
1084    assign cwstb = (wbbd_busy) ? wbbd_write : wrstb;
1085
1086    assign odata = fdata(caddr);
```

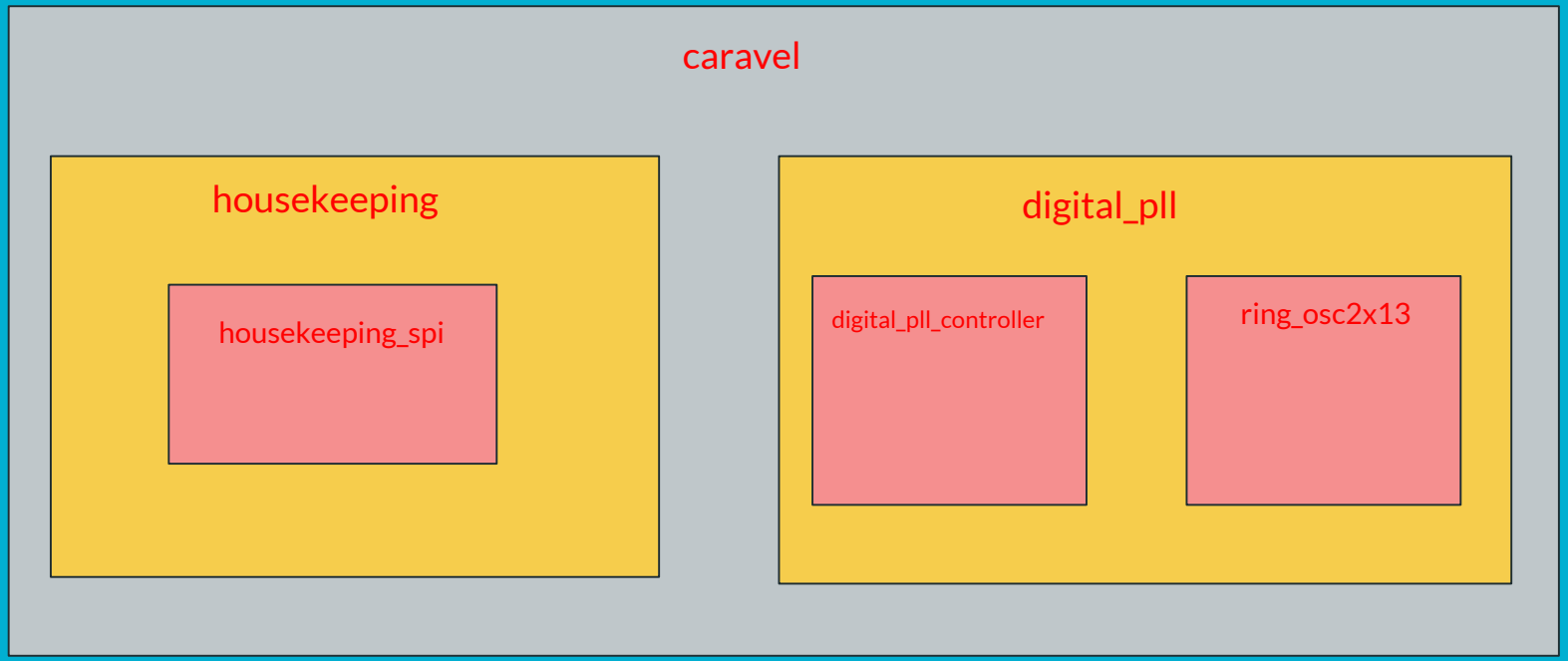
```
407     /* Register bit to SPI address mapping */
408
409     function [7:0] fdata(input [7:0] address);
410 begin
411 case (address)
412     /* Housekeeping SPI Protocol */
413     8'h00 : fdata = 8'h00; // SPI status (fixed)
414
415     /* Status and Identification */
416     8'h01 : fdata = {4'h0, mfgr_id[11:8]}; // Manufacturer ID (fixed)
417     8'h02 : fdata = mfgr_id[7:0];             // Manufacturer ID (fixed)
418     8'h03 : fdata = prod_id;                  // Product ID (fixed)
419     8'h04 : fdata = mask_rev[31:24];          // Mask rev (via programmed)
420     8'h05 : fdata = mask_rev[23:16];          // Mask rev (via programmed)
421     8'h06 : fdata = mask_rev[15:8];           // Mask rev (via programmed)
422     8'h07 : fdata = mask_rev[7:0];            // Mask rev (via programmed)
423
424     /* Clocking control */
425     8'h08 : fdata = {6'b000000, pll_dco_ena, pll_ena};
426     8'h09 : fdata = {7'b0000000, pll_bypass};
427     8'h0a : fdata = {7'b0000000, irq_spi};
428     8'h0b : fdata = {7'b0000000, reset};
429     8'h0c : fdata = {7'b0000000, trap}; // CPU trap state
430     8'h0d : fdata = pll_trim[7:0];
431     8'h0e : fdata = pll_trim[15:8];
432     8'h0f : fdata = pll_trim[23:16];
433     8'h10 : fdata = {6'b000000, pll_trim[25:24]};
434     8'h11 : fdata = {2'b00, pll90_sel, pll_sel};
435     8'h12 : fdata = {3'b000, pll_div};
```

Housekeeping SPI register map

Register Address	msb 7	6	5	4	3	2	1	lsb 0	comments
0x00	SPI status and control								unused/ undefined
0x01	unused				manufacturer_ID[11:8] (= 0x4)				read-only
0x02	manufacturer_ID[7:0] (= 0x56)								read-only
0x03	product_ID (= 0x11)								read-only
0x04–0x07	user_project_ID (unique value per project)								read-only
0x08	unused						DLL DCO enable	DLL enable	default 0x02
0x09	unused							DLL bypass	default 0x01
0x0A	unused							CPU IRQ	default 0x00
0x0B	unused							CPU reset	default 0x00
0x0C	unused							CPU trap	read-only
0x0D–0x10	DCO trim (26 bits) (= 0x3ffefff)								default 0x3ffefff
0x11	unused		PLL output divider 2			PLL output divider			default 0x12
0x12	unused				PLL feedback divider				default 0x04
0x13		serial data 2	serial data 1	serial clock	serial load	serial reset	serial enable	serial xfer/ busy*	bits 6 to 1 are bit-bang control.

* Bit is write-only for serial transfer, read-only for serial busy. During transfer, the busy bit is one. Transfer is complete when the busy bit returns to zero.

modules relative to DLL



verilog/rtl/caravel.v

```
766     // DCO/Digital Locked Loop
767
768     digital_pll pll (
769     `ifdef USE_POWER_PINS
770                 .VPWR(vccd_core),
771                 .VGND(vssd_core),
772     `endif
773         .resetb(rstb_l_buf),
774         .enable(spi_pll_ena),
775         .osc(clock_core_buf),
776         .clockp({pll_clk, pll_clk90}),
777         .div(spi_pll_div),
778         .dco(spi_pll_dco_ena),
779         .ext_trim(spi_pll_trim)
780     );
```

```
782     // Housekeeping interface
783
784     housekeeping housekeeping (
785     `ifdef USE_POWER_PINS
786                 .VPWR(vccd_core),
787                 .VGND(vssd_core),
788     `endif
789
790         .wb_clk_i(caravel_clk),
791         .wb_rstn_i(caravel_rstn),
792
793         .wb_adr_i(mprj_adr_o_core),
794         .wb_dat_i(mprj_dat_o_core),
795         .wb_sel_i(mprj_sel_o_core),
796         .wb_we_i(mprj_we_o_core),
797         .wb_cyc_i(hk_cyc_o),
798         .wb_stb_i(hk_stb_o),
799         .wb_ack_o(hk_ack_i),
800         .wb_dat_o(hk_dat_i),
801
802         .porb(porb_l),
803
804         .pll_ena(spi_pll_ena),
805         .pll_dco_ena(spi_pll_dco_ena),
806         .pll_div(spi_pll_div),
807         .pll_sel(spi_pll_sel),
808         .pll90_sel(spi_pll90_sel),
809         .pll_trim(spi_pll_trim),
810         .pll_bypass(ext_clk_sel),
811
```

verilog/rtl/digital_pll.v (1)

```
25 module digital_pll(
26 `ifdef USE_POWER_PINS
27     VPWR,
28     VGND,
29 `endif
30     resetb, enable, osc, clockp, div, dco, ext_trim);
31
32 `ifdef USE_POWER_PINS
33     input VPWR;
34     input VGND;
35 `endif
36
37     input        resetb;        // Sense negative reset
38     input        enable;        // Enable PLL
39     input        osc;           // Input oscillator to match
40     input [4:0]  div;           // PLL feedback division ratio
41     input        dco;           // Run in DCO mode
42     input [25:0] ext_trim;      // External trim for DCO mode
43
44     output [1:0] clockp;        // Two 90 degree clock phases
45
46     wire [1:0]   clockp_buffer_in;      // Input wires to clockp buffers
47     wire [25:0]  itrim;         // Internally generated trim bits
48     wire [25:0]  otrim;         // Trim bits applied to the ring oscillator
49     wire         creset;        // Controller reset
50     wire         ireset;        // Internal reset (external reset OR disable)
51
52     assign ireset = ~resetb | ~enable;
53
54     // In DCO mode: Hold controller in reset and apply external trim value
55
56     assign itrim = (dco == 1'b0) ? otrim : ext_trim;
57     assign creset = (dco == 1'b0) ? ireset : 1'b1;
```

Low assert - POR

external clock

extfree-running (DCO mode), no need reference clock from osc

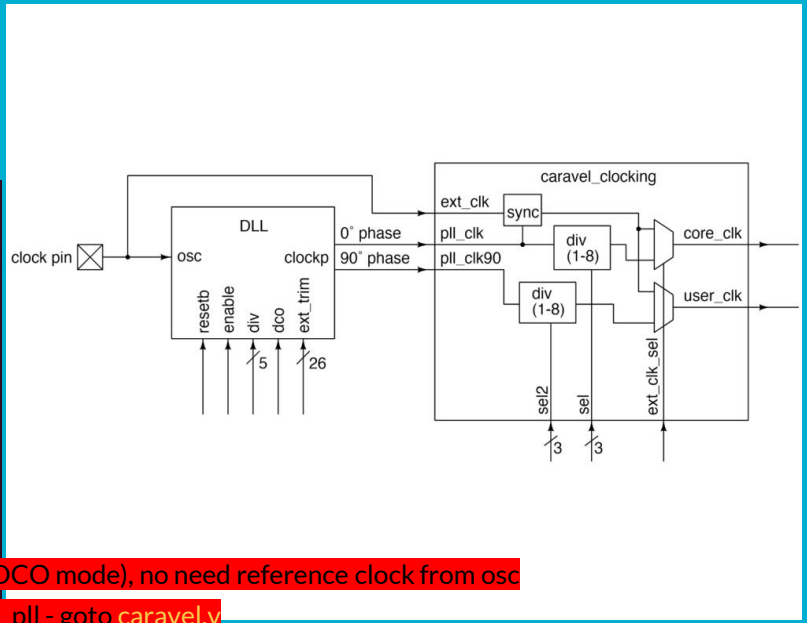
ext_trim from spi_pll - goto caravel.v

itrim for Trim bits applied to ring_osc

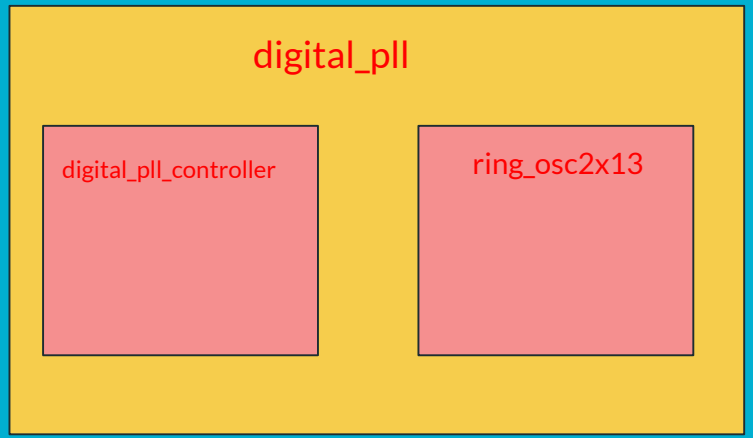
otrim come from digital_pll_controller generates trim bits

High assert

When DCO mode then no need pll_controller, ring_osc will provide the clock output by ext_trim(manual trim) setting



verilog/rtl/digital_pll.v (2)




```
59      (* keep *)
60      ring_osc2x13 ringosc (
61          .reset(ireset),
62          .trim(itrim),
63          .clockp(clockp_buffer_in)
64      );
65
66      digital_pll_controller pll_control (
67          .reset(creset),
68          .clock(clockp_buffer_in[0]),
69          .osc(osc),
70          .div(div),
71          .trim(otrim)
72      );
73
74      (* keep *)
75      sky130_fd_sc_hd__clkbuf_16 clockp_buffer_0 (
76  `ifdef USE_POWER_PINS
77          .VPWR(VPWR),
78          .VGND(VGND),
79          .VPB(VPWR),
80          .VNB(VGND),
81  `endif
82          .A(clockp_buffer_in[0]),
83          .X(clockp[0])
84      );
85
86      (* keep *)
87      sky130_fd_sc_hd__clkbuf_16 clockp_buffer_1 (
88  `ifdef USE_POWER_PINS
89          .VPWR(VPWR),
90          .VGND(VGND),
91          .VPB(VPWR),
92          .VNB(VGND),
93  `endif
94          .A(clockp_buffer_in[1]),
95          .X(clockp[1])
96      );
97
98  endmodule
```

itrim for ring_osc

clockp ouput

clockp input

external clock (reference clock)

otrim for digital_pll_controller generates trim bits

input

outut

input

outut

verilog/rtl/ring_osc2x13.v

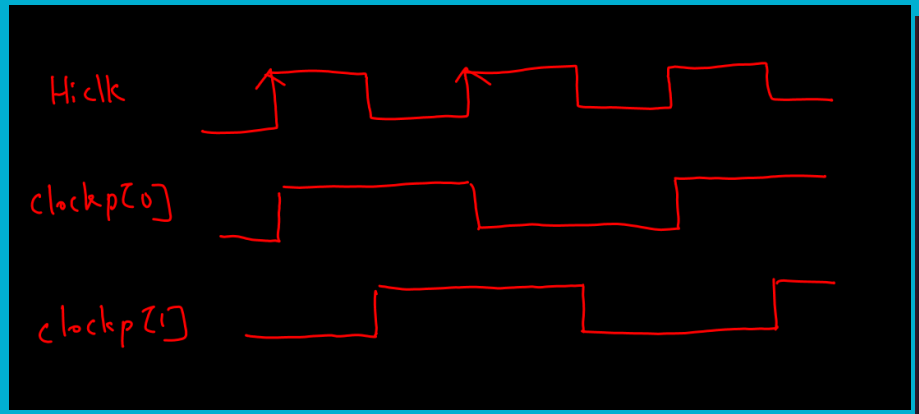
```
131 // Ring oscillator with 13 stages, each with two trim bits delay
132 // (see above).  Trim is not binary:  For trim[1:0], lower bit
133 // trim[0] is primary trim and must be applied first;  upper
134 // bit trim[1] is secondary trim and should only be applied
135 // after the primary trim is applied, or it has no effect.
136 //
137 // Total effective number of inverter stages in this oscillator
138 // ranges from 13 at trim 0 to 65 at trim 24.  The intention is
139 // to cover a range greater than 2x so that the midrange can be
140 // reached over all PVT conditions.
141 //
142 // Frequency of this ring oscillator under SPICE simulations at
143 // nominal PVT is maximum 214 MHz (trim 0), minimum 90 MHz (trim 24).
144
145 module ring_osc2x13(reset, trim, clockp);
146     input reset;
147     input [25:0] trim;
148     output[1:0] clockp;
149
150 `ifdef FUNCTIONAL       // i.e., behavioral model below
151
152     reg [1:0] clockp;
153     reg hiclock;
154     integer i;
155     real delay;
156     wire [5:0] bcount;
157
158     assign bcount = trim[0] + trim[1] + trim[2]
159                   + trim[3] + trim[4] + trim[5] + trim[6] + trim[7]
160                   + trim[8] + trim[9] + trim[10] + trim[11] + trim[12]
161                   + trim[13] + trim[14] + trim[15] + trim[16] + trim[17]
162                   + trim[18] + trim[19] + trim[20] + trim[21] + trim[22]
163                   + trim[23] + trim[24] + trim[25];
164
165     initial begin
166         hiclock <= 1'b0;
167         delay = 3.0;
168     end
```

behavioral model

```
170     // Fastest operation is 214 MHz = 4.67ns
171     // Delay per trim is 0.02385
172     // Run "hiclock" at 2x this rate, then use positive and negative
173     // edges to derive the 0 and 90 degree phase clocks.
174
175     always #delay begin
176         hiclock <= (hiclock === 1'b0);
177     end
178
179     always @(trim) begin
180         // Implement trim as a variable delay, one delay per trim bit
181         delay = 1.168 + 0.012 * $itor(bcount);
182     end
183
184     always @(posedge hiclock or posedge reset) begin
185         if (reset == 1'b1) begin
186             clockp[0] <= 1'b0;
187         end else begin
188             clockp[0] <= (clockp[0] === 1'b0);
189         end
190     end
191
192     always @(negedge hiclock or posedge reset) begin
193         if (reset == 1'b1) begin
194             clockp[1] <= 1'b0;
195         end else begin
196             clockp[1] <= (clockp[1] === 1'b0);
197         end
198     end
199
200 `else                   // !FUNCTIONAL;  i.e., gate level netlist below
```

when reset = 1 then no clock toggle

phase 0 & phase 90 clock in behavior model



===	Case equality

Table 5-11—Definitions of equality operators

a === b	a equal to b, including x and z
a !== b	a not equal to b, including x and z
a == b	a equal to b, result can be unknown
a != b	a not equal to b, result can be unknown

```
170     // Fastest operation is 214 MHz = 4.67ns
171     // Delay per trim is 0.02385
172     // Run "hiclock" at 2x this rate, then use positive and negative
173     // edges to derive the 0 and 90 degree phase clocks.
174
175     always #delay begin
176         hiclock <= (hiclock === 1'b0);
177     end
178
179     always @(trim) begin
180         // Implement trim as a variable delay, one delay per trim bit
181         delay = 1.168 + 0.012 * $itor(bcount);
182     end
183
184     always @(posedge hiclock or posedge reset) begin
185         if (reset == 1'b1) begin
186             clockp[0] <= 1'b0;
187         end else begin
188             clockp[0] <= (clockp[0] === 1'b0);
189         end
190     end
191
192     always @(negedge hiclock or posedge reset) begin
193         if (reset == 1'b1) begin
194             clockp[1] <= 1'b0;
195         end else begin
196             clockp[1] <= (clockp[1] === 1'b0);
197         end
198     end
199
200 `else                   // !FUNCTIONAL;  i.e., gate level netlist below
```

hiclock may be x or z when the behavoir model start

when reset = 1 then no clock toggle

ring_osc2x13.v - gate level netlist

FUNCTIONAL

```
201
202     wire [1:0] clockp;
203     wire [12:0] d;
204     wire [1:0] c;
205
206     // Main oscillator loop stages
207
208     genvar i;
209     generate
210         for (i = 0; i < 12; i = i + 1) begin : dstage
211             delay_stage id (
212                 .in(d[i]),
213                 .trim({trim[i+13], trim[i]}),
214                 .out(d[i+1])
215             );
216         end
217     endgenerate
218
219     // Reset/startup stage
220
221     start_stage iss (
222         .in(d[12]),
223         .trim({trim[25], trim[12]}),
224         .reset(reset),
225         .out(d[0])
226     );
227
```

delay_stage trim[1,0] group is mapping to digital_pll_controller.v

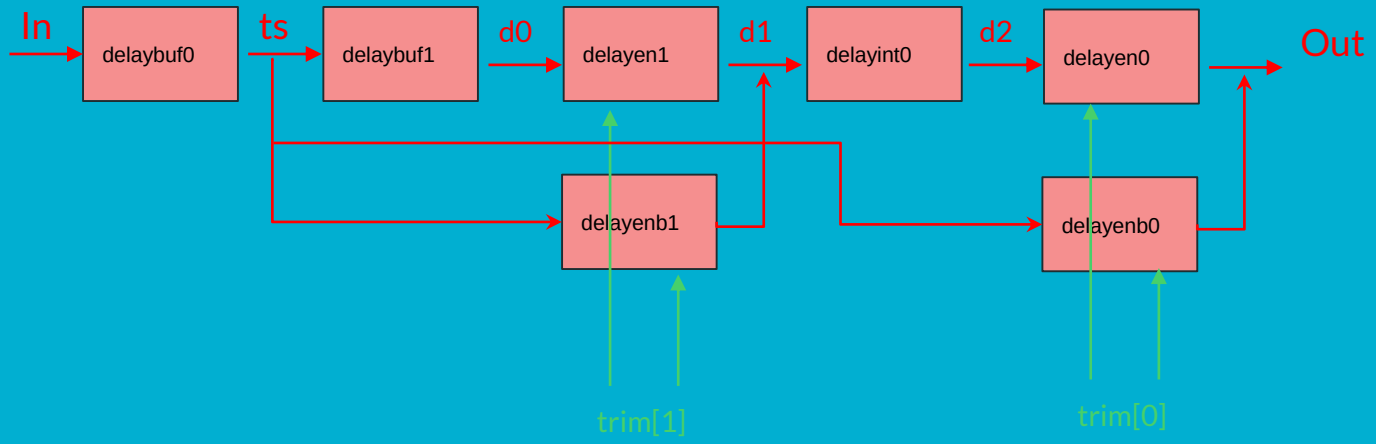
start_stage - ring_osc2x13.v - start_stage

```
228     // Buffered outputs a 0 and 90 degrees phase (approximately)
229
230     sky130_fd_sc_hd__clkinv_2 ibufp00 (
231         .A(d[0]),
232         .Y(c[0])
233     );
234     sky130_fd_sc_hd__clkinv_8 ibufp01 (
235         .A(c[0]),
236         .Y(clockp[0])
237     );
238     sky130_fd_sc_hd__clkinv_2 ibufp10 (
239         .A(d[6]),
240         .Y(c[1])
241     );
242     sky130_fd_sc_hd__clkinv_8 ibufp11 (
243         .A(c[1]),
244         .Y(clockp[1])
245     );
246
247 `endif // !FUNCTIONAL
248
```

get phase 0 from d[0]

get phase 90 from d[6]

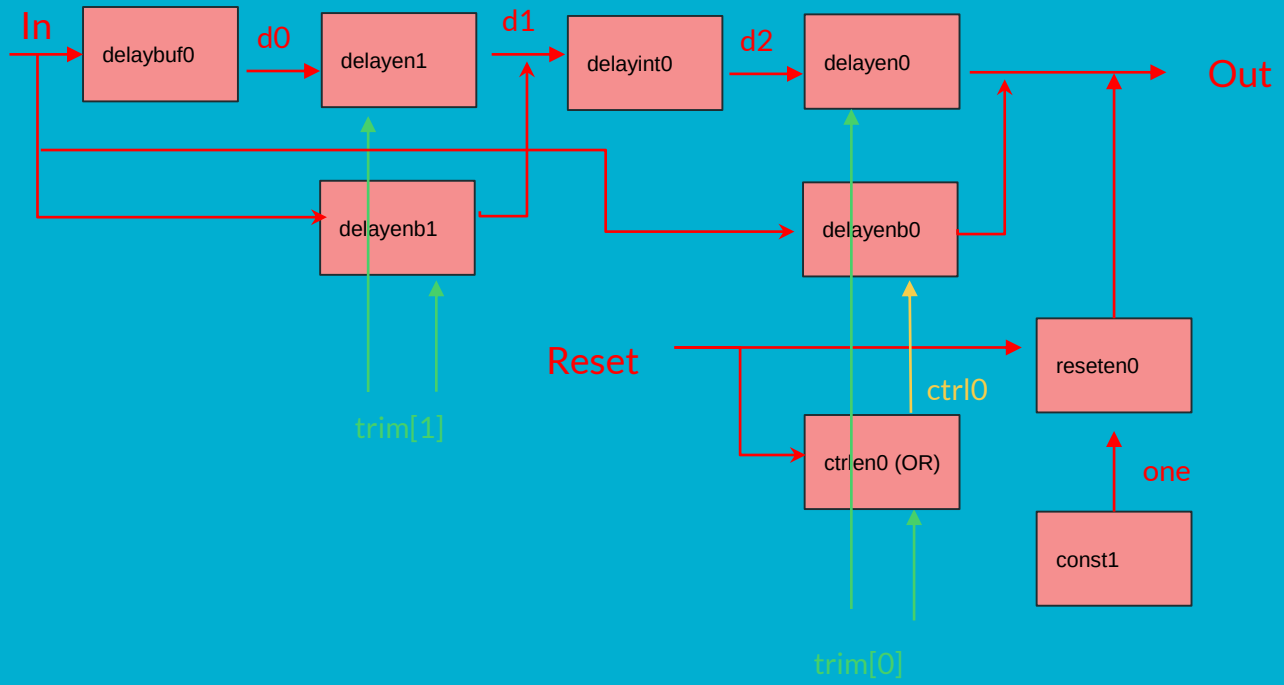
ring_osc2x13.v - delay_stage




```
22 module delay_stage(in, trim, out);
23     input in;
24     input [1:0] trim;
25     output out;
26
27     wire d0, d1, d2, ts;
28
29     sky130_fd_sc_hd__clkbuf_2 delaybuf0 (
30         .A(in),
31         .X(ts)
32     );
33
34     sky130_fd_sc_hd__clkbuf_1 delaybuf1 (
35         .A(ts),
36         .X(d0)
37     );
38
39     sky130_fd_sc_hd__einvp_2 delayen1 (
40         .A(d0),
41         .TE(trim[1]),
42         .Z(d1)
43     );
44
```

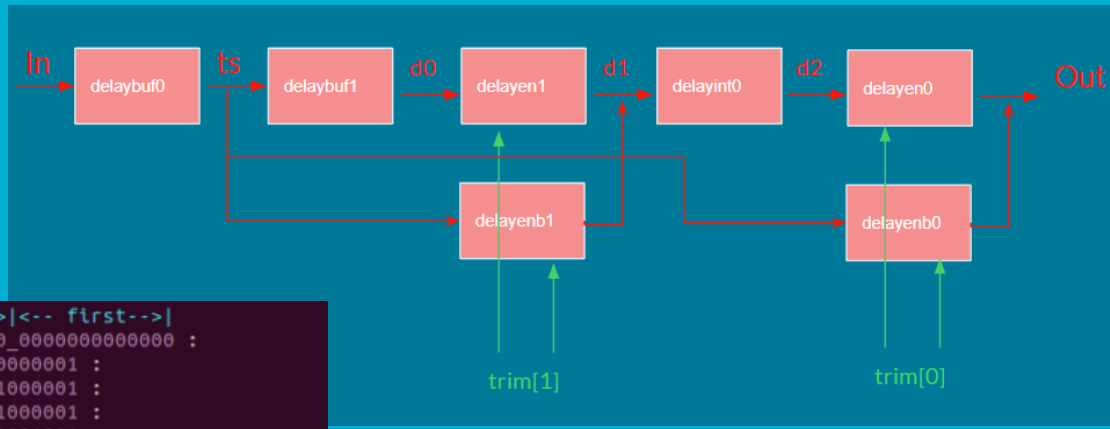
```
45     sky130_fd_sc_hd__einvn_4 delayenb1 (
46         .A(ts),
47         .TE_B(trim[1]),
48         .Z(d1)
49     );
50
51     sky130_fd_sc_hd__clkinv_1 delayint0 (
52         .A(d1),
53         .Y(d2)
54     );
55
56     sky130_fd_sc_hd__einvp_2 delayen0 (
57         .A(d2),
58         .TE(trim[0]),
59         .Z(out)
60     );
61
62     sky130_fd_sc_hd__einvn_8 delayenb0 (
63         .A(ts),
64         .TE_B(trim[0]),
65         .Z(out)
66     );
67
68 endmodule
```

ring_osc2x13.v - satrt_stage




```
70 module start_stage(in, trim, reset, out);
71     input in;
72     input [1:0] trim;
73     input reset;
74     output out;
75
76     wire d0, d1, d2, ctrl0, one;
77
78     sky130_fd_sc_hd__clkbuf_1 delaybuf0 (
79         .A(in),
80         .X(d0)
81     );
82
83     sky130_fd_sc_hd__einvp_2 delayen1 (
84         .A(d0),
85         .TE(trim[1]),
86         .Z(d1)
87     );
88
89     sky130_fd_sc_hd__einvn_4 delayenb1 (
90         .A(in),
91         .TE_B(trim[1]),
92         .Z(d1)
93     );
94
95     sky130_fd_sc_hd__clkinv_1 delayint0 (
96         .A(d1),
97         .Y(d2)
98     );
99
100    sky130_fd_sc_hd__einvp_2 delayen0 (
101        .A(d2),
102        .TE(trim[0]),
103        .Z(out)
104    );
105
106    sky130_fd_sc_hd__einvn_8 delayenb0 (
107        .A(in),
108        .TE_B(ctrl0),
109        .Z(out)
110    );
111
112    sky130_fd_sc_hd__einvp_1 reseten0 (
113        .A(one),
114        .TE(reset),
115        .Z(out)
116    );
117
118    sky130_fd_sc_hd__or2_2 ctrlen0 (
119        .A(reset),
120        .B(trim[0]),
121        .X(ctrl0)
122    );
123
124    sky130_fd_sc_hd__conb_1 const1 (
125        .HI(one),
126        .LO()
127    );
128
129 endmodule
```

Thermometer code



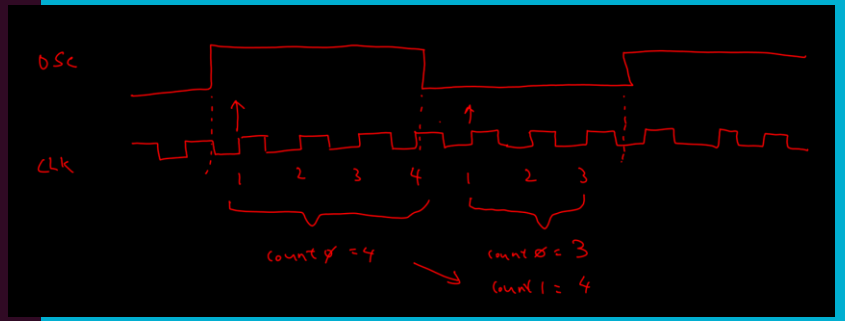

```
71                                  // |<--second-->|<-- first-->|
72      assign trim = (tint == 5'd0)  ? 26'b0000000000000_0000000000000 :
73             (tint == 5'd1)  ? 26'b0000000000000_0000000000001 :
74             (tint == 5'd2)  ? 26'b0000000000000_0000001000001 :
75             (tint == 5'd3)  ? 26'b0000000000000_0010001000001 :
76             (tint == 5'd4)  ? 26'b0000000000000_0010001001001 :
77             (tint == 5'd5)  ? 26'b0000000000000_0010101001001 :
78             (tint == 5'd6)  ? 26'b0000000000000_1010101001001 :
79             (tint == 5'd7)  ? 26'b0000000000000_1010101101001 :
80             (tint == 5'd8)  ? 26'b0000000000000_1010101101101 :
81             (tint == 5'd9)  ? 26'b0000000000000_1011101101101 :
82             (tint == 5'd10) ? 26'b0000000000000_1011101111101 :
83             (tint == 5'd11) ? 26'b0000000000000_1111101111101 :
84             (tint == 5'd12) ? 26'b0000000000000_1111101111111 :
85             (tint == 5'd13) ? 26'b0000000000000_1111111111111 :
86             (tint == 5'd14) ? 26'b0000000000001_1111111111111 :
87             (tint == 5'd15) ? 26'b0000001000001_1111111111111 :
88             (tint == 5'd16) ? 26'b0010001000001_1111111111111 :
89             (tint == 5'd17) ? 26'b0010001001001_1111111111111 :
90             (tint == 5'd18) ? 26'b0010101001001_1111111111111 :
91             (tint == 5'd19) ? 26'b1010101001001_1111111111111 :
92             (tint == 5'd20) ? 26'b1010101101001_1111111111111 :
93             (tint == 5'd21) ? 26'b1010101101101_1111111111111 :
94             (tint == 5'd22) ? 26'b1011101101101_1111111111111 :
95             (tint == 5'd23) ? 26'b1011101111101_1111111111111 :
96             (tint == 5'd24) ? 26'b1111101111101_1111111111111 :
97             (tint == 5'd25) ? 26'b1111101111111_1111111111111 :
98                      26'b1111111111111_1111111111111;
99
```

DLL operation

- The DLL operates by taking the ring oscillator output, reducing its frequency through a feedback divider(??), and comparing the result to the input clock. If the frequency of the divided-down ring oscillator output is faster than the input clock frequency, then an additional delay stage is added to the ring oscillator, making it run slower. If the frequency of the divided-down ring oscillator output is slower than the input clock frequency, then a delay stage is subtracted from the ring oscillator, making it faster. This operation is performed in a continuous loop to keep the DLL frequency locked to the input clock.
- WARNING: Using discrete delay state insertion and removal results in high phase noise (cycle to cycle jitter) on the core clock when running in DLL mode, due to instantaneous changes of 250ps between cycles (a 0.25% change in the clock period). User projects that require a clock with low phase noise should use the external clock (DLL in bypass mode), and if the project requires a higher clock rate, then the 10MHz clock on the demonstration board may be replaced with another oscillator of the same footprint with a frequency up to 50MHz. The DLL running in DCO mode has low cycle-to-cycle jitter but will have a large drift component, as it is not temperature stabilized.

verilog/rtl/digital_pll_controller.v

```
17 // (True) digital PLL
18 //
19 // Output goes to a trimmable ring oscillator (see documentation).
20 // Ring oscillator should be trimmable to above and below maximum
21 // ranges of the input.
22 //
23 // Input "osc" comes from a fixed clock source (e.g., crystal oscillator
24 // output).
25 //
26 // Input "div" is the target number of clock cycles per oscillator cycle.
27 // e.g., if div == 8 then this is an 8X PLL.
28 //
29 // Clock "clock" is the PLL output being trimmed.
30 // (NOTE:  To be done:  Pass-through enable)
31 //
32 // Algorithm:
33 //
34 // 1) Trim is done by thermometer code.  Reset to the highest value
35 //    in case the fastest rate clock is too fast for the logic.
36 //
37 // 2) Count the number of contiguous 1s and 0s in "osc"
38 //    periods of the master clock.  If the count maxes out, it does
39 //    not roll over.
40 //
41 // 3) Add the two counts together.
42 //
43 // 4) If the sum is less than div, then the clock is too slow, so
44 //    decrease the trim code.  If the sum is greater than div, the
45 //    clock is too fast, so increase the trim code.  If the sum
46 //    is equal to div, the the trim code does not change.
47 //
```



thermometer code

verilog/rtl/digital_pll_controller.v

When DCO mode, reset = 1
else (in DLL mode) output auto generates trim bits

clock from ring_osc
external clock

trim is come from tval

```
49 module digital_pll_controller(reset, clock, osc, div, trim);
50     input reset;
51     input clock;
52     input osc;
53     input [4:0] div;
54     output [25:0] trim;          // Use ring_osc2x13, with 26 trim bits
55
56     wire [25:0] trim;
57     reg [2:0] oscbuf;
58     reg [2:0] prep;
59
60     reg [4:0] count0;
61     reg [4:0] count1;
62     reg [6:0] tval;      // Includes 2 bits fractional
63     wire [4:0] tint;     // Integer part of the above
64
65     wire [5:0] sum;
66
67     assign sum = count0 + count1;
68
69     // Integer to thermometer code (maybe there's an algorithmic way?)
70     assign tint = tval[6:2];
```

```
71                                  // |<--second-->|<-- first-->|
72     assign trim = (tint == 5'd0)  ? 26'b00000000000000_0000000000000 :
73           (tint == 5'd1)  ? 26'b00000000000000_0000000000001 :
74           (tint == 5'd2)  ? 26'b00000000000000_0000001000001 :
75           (tint == 5'd3)  ? 26'b00000000000000_0010001000001 :
76           (tint == 5'd4)  ? 26'b00000000000000_0010001001001 :
77           (tint == 5'd5)  ? 26'b00000000000000_0010101001001 :
78           (tint == 5'd6)  ? 26'b00000000000000_1010101001001 :
79           (tint == 5'd7)  ? 26'b00000000000000_1010101101001 :
80           (tint == 5'd8)  ? 26'b00000000000000_1010101101101 :
81           (tint == 5'd9)  ? 26'b00000000000000_1011101101101 :
82           (tint == 5'd10) ? 26'b00000000000000_1011101111101 :
83           (tint == 5'd11) ? 26'b00000000000000_1111101111101 :
84           (tint == 5'd12) ? 26'b00000000000000_1111101111111 :
85           (tint == 5'd13) ? 26'b00000000000000_1111111111111 :
86           (tint == 5'd14) ? 26'b00000000000001_1111111111111 :
87           (tint == 5'd15) ? 26'b00000001000001_1111111111111 :
88           (tint == 5'd16) ? 26'b00010001000001_1111111111111 :
89           (tint == 5'd17) ? 26'b00010001001001_1111111111111 :
90           (tint == 5'd18) ? 26'b00101010001001_1111111111111 :
91           (tint == 5'd19) ? 26'b01010101001001_1111111111111 :
92           (tint == 5'd20) ? 26'b01010101101001_1111111111111 :
93           (tint == 5'd21) ? 26'b01010101101101_1111111111111 :
94           (tint == 5'd22) ? 26'b01111011011101_1111111111111 :
95           (tint == 5'd23) ? 26'b01111011111101_1111111111111 :
96           (tint == 5'd24) ? 26'b11111011111101_1111111111111 :
97           (tint == 5'd25) ? 26'b11111011111111_1111111111111 :
98                     26'b11111111111111_1111111111111;
99
```

```
100     always @(posedge clock or posedge reset) begin
101     if (reset == 1'b1) begin
102         tval <= 7'd0;   // Note:  trim[0] must be zero for startup to work.
103         oscbuf <= 3'd0;
104         prep <= 3'd0;
105         count0 <= 5'd0;
106         count1 <= 5'd0;
107
108     end else begin
109         oscbuf <= {oscbuf[1:0], osc};
110
111         if (oscbuf[2] != oscbuf[1]) begin
112         count1 <= count0;
113         count0 <= 5'b00001;
114         prep <= {prep[1:0], 1'b1};
115
116         if (prep == 3'b111) begin
117             if (sum > div) begin
118                 if (tval < 127) begin
119                     tval <= tval + 1;
120                 end
121             end else if (sum < div) begin
122                 if (tval > 0) begin
123                     tval <= tval - 1;
124                 end
125             end
126         end
127         end else begin
128         if (count0 != 5'b11111) begin
129                 count0 <= count0 + 1;
130         end
131         end
132     end
133     end
134
135 endmodule       // digital_pll_controller
```

detect osc toggle

if count1 is number clock of osc=0 then count1 is number clock of osc=1

after toggle 3 times, 4th toggle then pre = 111

sum > div means clock is too fast then add trim

sum < div means clock is too slow then decrease trim

increase count0 when osc no changed