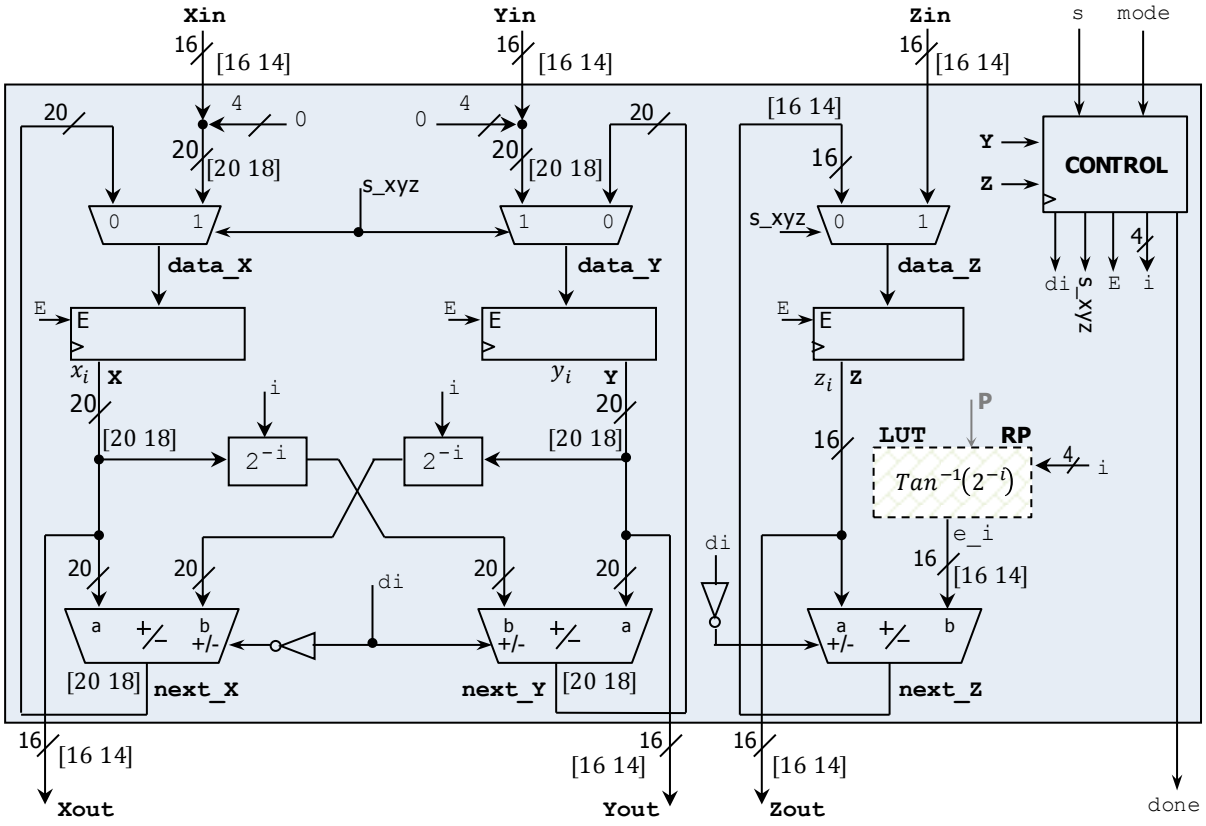
Unit 7 - Applications

DYNAMIC CIRCULAR CORDIC

- This is the 16-bit Circular CORDIC iterative architecture, where we perform run-time alteration of the FX format:
  - ✓ [16 15]: It provides the highest accuracy, but the input, intermediate, and output values are restricted to [-1,1).
  - ✓ [16 14]: It provides good accuracy, with the input, intermediate, and output values in the range [-2,2).
  - ✓ [16 13]: It provides decent accuracy, with the input, intermediate and output values in the range [-4,4).
  - ✓ [16 12]: It provides low accuracy. However, the input, intermediate and output values can be in the range [-8,8).
- A software application routinely jumps among the four configurations with different input data.
- CORDIC: The input, intermediate, and output data might require different number of integer bits depending on the input data range. The more fractional bits, the higher the accuracy at the expense of decreased dynamic range (which might not be acceptable in some circumstances). Fewer fractional bits allow for a higher dynamic range which allows some input data to be properly processed, but at the expense of reduced accuracy.

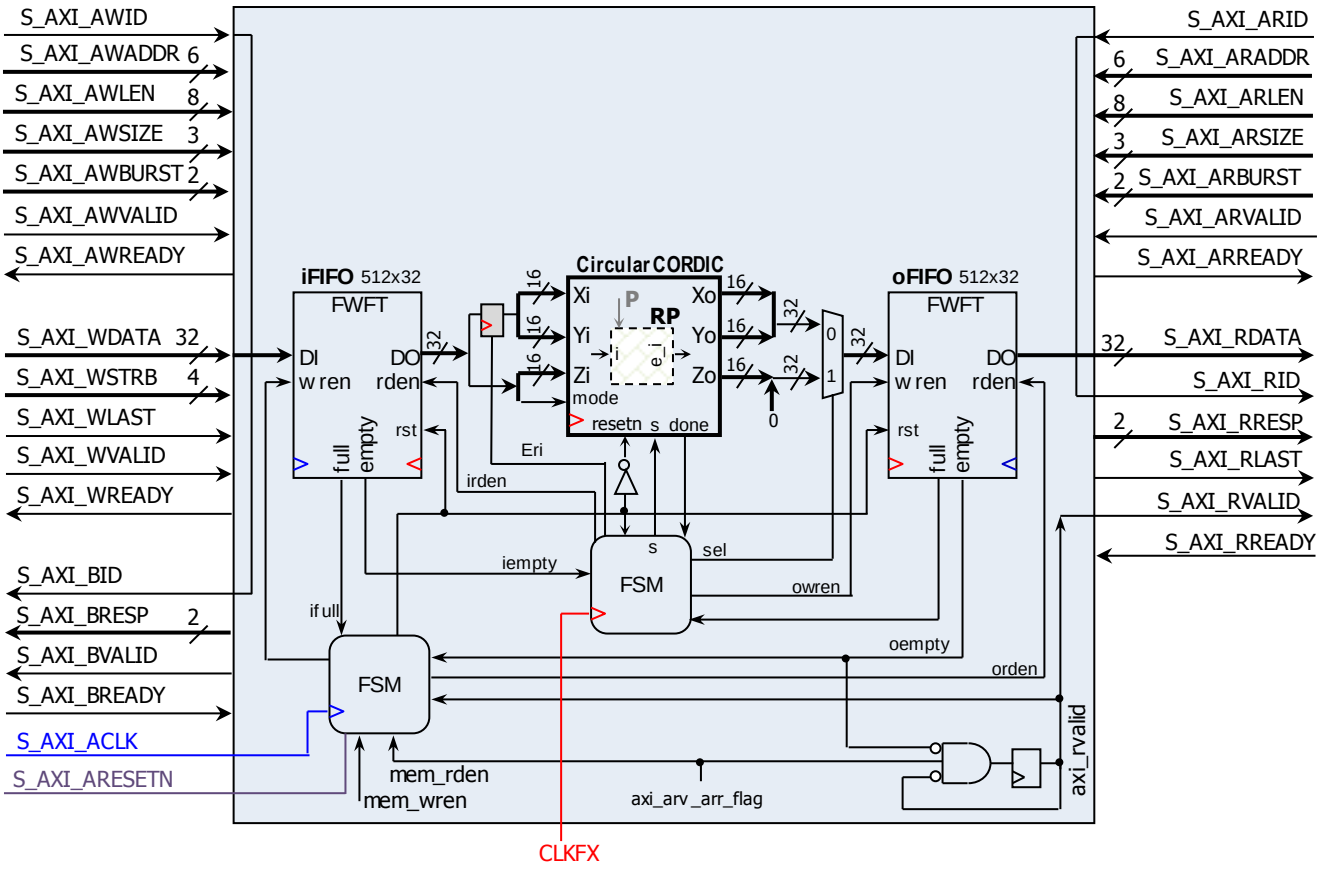
FIXED-POINT CIRCULAR CORDIC WITH RECONFIGURABLE LUT

- This synchronous iterative circuit reads input data (16-bit Xin, 16-bit Yin, 16-bit Zin, and mode) when the s signal (a one-cycle pulse) is asserted. After a number of processing cycles, the result (16-bit Xout, 16-bit Yout, 16-bit Zout) is ready and it is signaled by done=1. Only after this, we can feed a new input data set (s=1).
- The figure depicts this circuit and how it was partitioned into static and dynamic (run-time alterable) components. The RP is a reconfigurable LUT whose contents can be set via the parameter P (12,13,14,15) which is the number of fractional bits. The VHDL code must be written (or rearranged if code is already available) in such a way that it exposes the Reconfigurable Partition (RP) as a `.vhd` file.



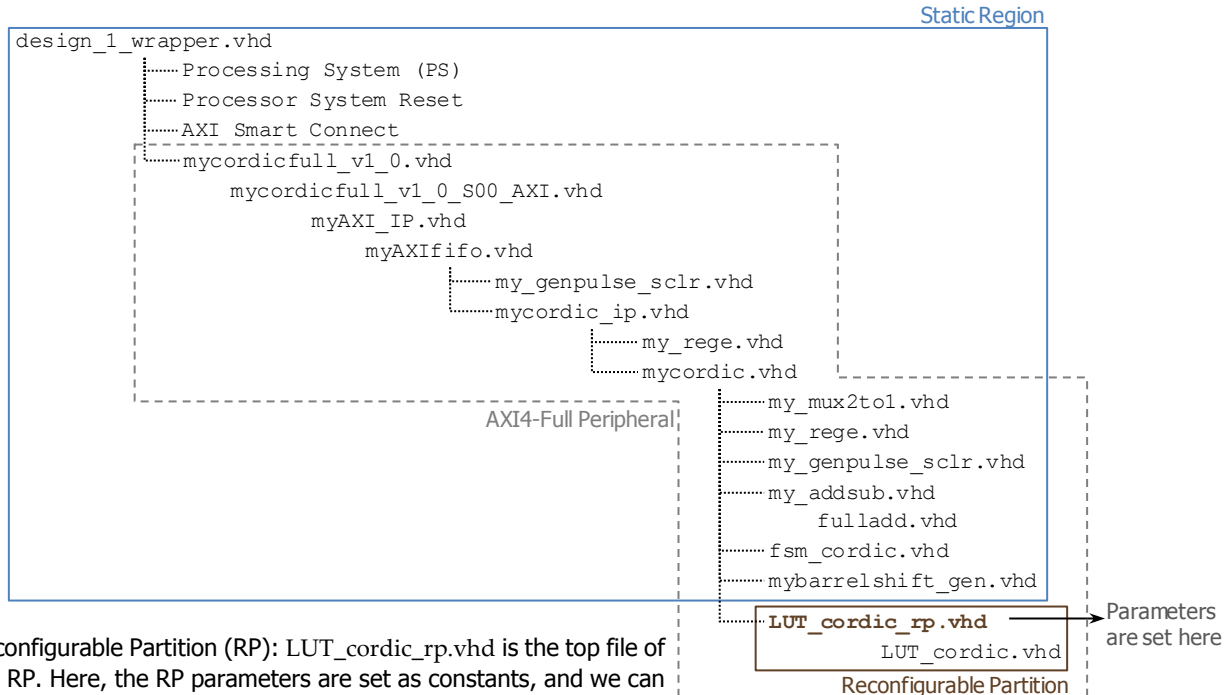
AXI4-FULL PERIPHERAL

- The figure depicts the peripheral (AXI4-Full Interface + CORDIC). The Static Region includes the AXI interface, while the run-time alterable region is the LUT inside the CORDIC.



HARDWARE PARTITIONING: STATIC REGION AND RECONFIGURABLE PARTITION

- The figure illustrates the design structure of the embedded system partitioned into the Static Region and a Reconfigurable Partition (RP). The AXI4-Full CORDIC peripheral (which includes the RP and some Static logic) structure is also shown.



- ✓ Reconfigurable Partition (RP): LUT_cordic_rp.vhd is the top file of the RP. Here, the RP parameters are set as constants, and we can modify them to create a variant (RM). There is only one modifiable parameter: P (12,13,14,15) that allows for up to 4 RM variants.
- ✓ Static Region: the PS (and extra designs) is included here, as well as the static portion of the AXI4-Full CORDIC peripheral.

- After we verify the proper functioning of our partitioned design (as a non-PR project), we create the folder structure for the Tcl-based flow for Partial Reconfiguration (see Notes → Unit 6). In this PR example, we have 1 RP and 4 RM variants that are created by modifying the parameter P (RM 1: P= 12, RM 2: P=13, RM 3: P=14, RM 4: P=15).

TESTING SCHEME

- We use four for the CORDIC architecture: formats [16 15], [16 14], [16 13], [16 12].
- For each format, we have an associated data set: 4 sets of input values and 4 sets of output values. Each set corresponds to two 32-bit input words (Xi&Yi, "00…0"&mode&Zi) and the expected two 32-bit output words (Xo&Yo, "00..0"&Zo).

	FX Format [16 15]	Inputs		Outputs	
FIRST DATASET		Xi&Yi	"00…0"&mode&Zi	Xo&Yo	"00…0"&Zo
	Rotation Mode	0x00004DBA	0x00006488	0xA57A5A7E	0x00000001
		0x00004DBA	0X0000BCFB	0x3FFE6EDA	0x0000FFFE
	Vectoring Mode	0x40002000	0x00010000	0x75D50000	0x00003B57
		0x399A2666	0x00010000	0x7200FFFF	0x00004B43

	FX Format [16 14]	Inputs		Outputs	
SECOND DATASET		Xi&Yi	"00…0"&mode&Zi	Xo&Yo	"00…0"&Zo
	Rotation Mode	0x000026DD	0x00002182	0xDFFD376A	0x00000001
		0x000026DD	0x0000BCFA	0x376C2000	0x0000FFFF
	Vectoring Mode	0x33333333	0x00010000	0x773CFFFF	0x00003241
		0x20004000	0x00010000	0x75D5FFFF	0x000046DB

	FX Format [16 13]	Inputs		Outputs	
THIRD DATASET		Xi&Yi	"00…0"&mode&Zi	Xo&Yo	"00…0"Zo
	Rotation Mode	0x20002000	0x00000C91	0x1C8544D9	0x0000FFFF
		0x136F136F	0x00003244	0xDFF91FFA	0x00000002
	Vectoring Mode	0x1CCD1CCD	0x00010000	0x4312FFFF	0x00001920
		0x1E662000	0x00010000	0x48AF0000	0x000019F4

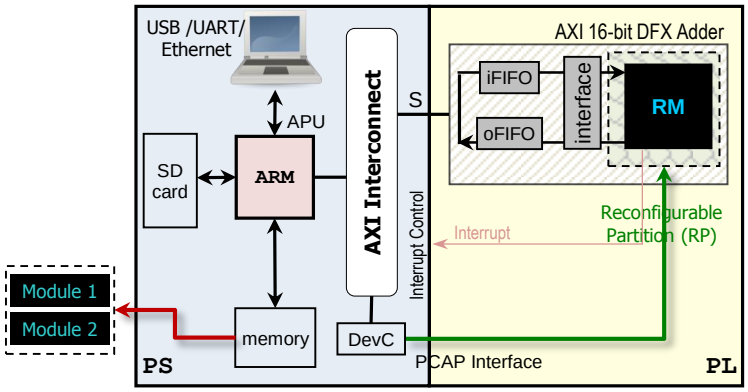
	FX Format [16 12]	Inputs		Outputs	
FOURTH DATASET		Xi&Yi	"00…0"&mode&Zi	Xo&Yo	"00…0"&Zo
	Rotation Mode	0x20002000	0x00000C91	0xFFF44A85	0x00000002
		0xE19AE400	0x0000F79F	0xBD95F124	0x0000FFFE
	Vectoring Mode	0x18002000	0x00010000	0x41DF0000	0x00000ED7
		0x1C001C00	0x00010000	0x4135FFFF	0x00000C8F

Test Procedure

- Extract the **`axicordic_dprsys.zip`**. It includes three folders and the software application:
  - ✓ `/axicordicfull_dr`: Files for implementing the AXI4-Full Cordic Peripheral (where code was rearranged for PR process).
  - ✓ `/axicordicfull_dr_static`: Files for implementing the static portion of the AXI4-Full Cordic Peripheral.
  - ✓ `/cordic16_dyn`: Folder structure for implementing the self-reconfigurable system.
    - We include the file `/Synth/Static/top_synth.dcp`, which is the Checkpoint of the Static Region.
    - We include the file `/Sources/xdc/top.xdc`. It contains the RP constraints and the PS constraints.
      `top.xdc = design_1_processing_system7_0_0.xdc + fplan.xdc`.
      - System tested on ZYBO Z7-10 and Vivado 2019.1. For other boards, you need to generate your own `top.xdc` and `top_synth.dcp` files as per the procedure in Embedded System Design for PSoC Tutorial → Unit 7: DCT 2D.
  - ✓ test_cordic_rp.c, xtra_func.h: Software application to test the self-reconfigurable system.
- Vivado Tcl Shell: `source design_complete.tcl -notrace`↵. This will generate the bitstreams (`.bit`). Follow the procedure in Embedded System Design for PSoC Tutorial → Unit 7: DCT 2D to generate the byte-swapped `.bin` partial bitstream files. Copy them onto the SD card and rename them to `cd16_12.bin`, `cd16_13.bin`, `cd16_14.bin`, `cd16_15.bin`.
- Create an embedded system (`cordic_16_drsys`) for the AXI4-Full Cordic Peripheral (`mycordicfull`). Create an SDK project (add the software files). Enable the '`xilffs`' library and the string manipulation functions. Allocate space for the heap/stack.
- In Vivado, program the full bitstream for the [16 15] configuration: `Config_cordic16_15.bit`.
- Run the software application in SDK:
  - ✓ With the CORDIC in format [16 15], the first dataset is tested. The results should match the expected output.
  - ✓ Then, the software routine reconfigures the CORDIC to the format [16 14].
  - ✓ With the CORDIC in format [16 14], the second dataset is tested. The results should match the expected output
  - ✓ Then, the software routine reconfigures the CORDIC to the format [16 13].
  - ✓ With the CORDIC in format [16 13], the third dataset is tested. The results should match the expected output.
  - ✓ Then, the software routine reconfigures the CORDIC to the format [16 12].
  - ✓ With the CORDIC in format [16 12], the fourth dataset is tested. The results should match the expected output.

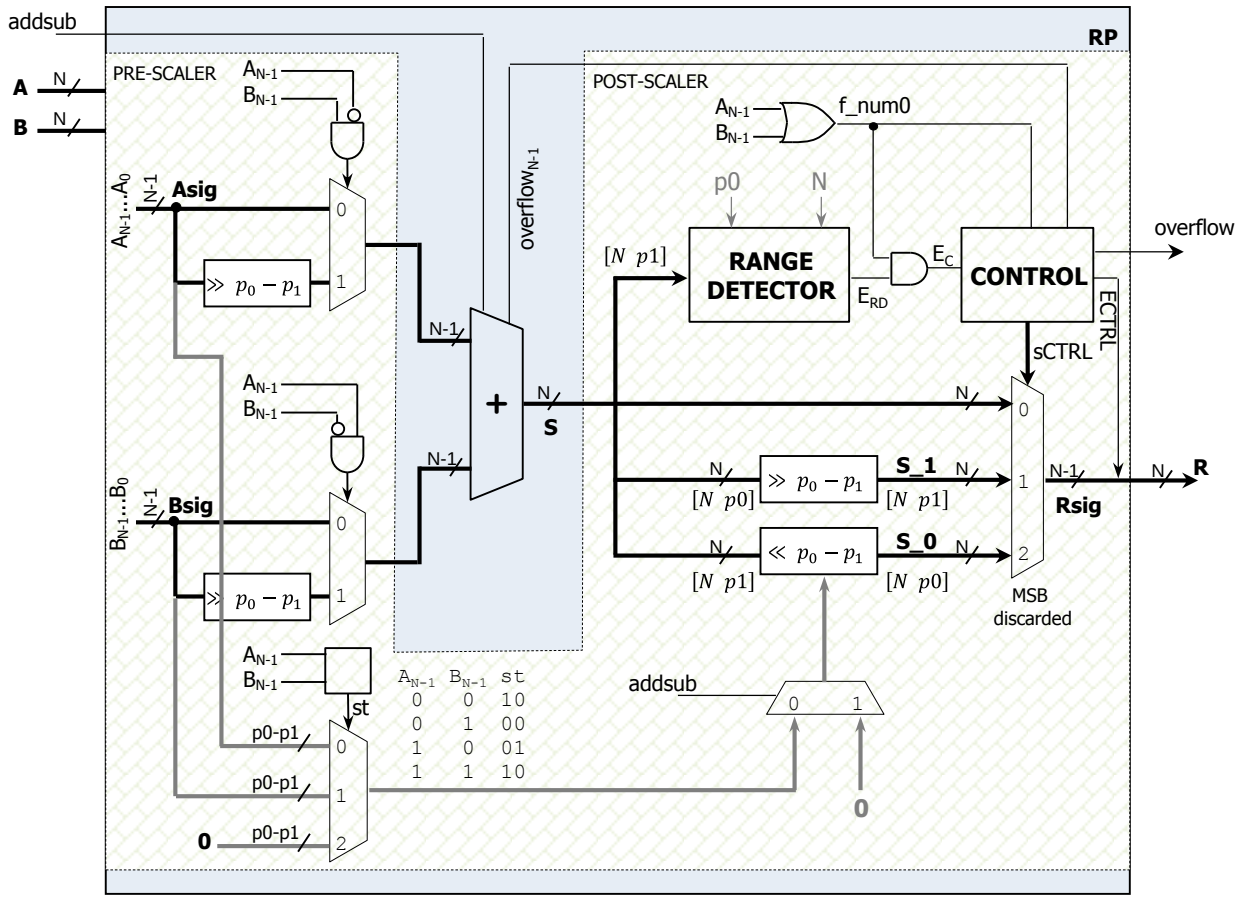
DYNAMIC ARITHMETIC: DYNAMIC DUAL FIXED-POINT ADDER

- Here, we use a 16-bit DFX Adder with overflow output. If overflow occurs, an interrupt is issued from the PL to the PS. The PS then detects the interrupt and alters (at run-time) the DFX format of the adder (by modifying p0 and p1) in order to avoid overflow. The figure depicts the self-reconfigurable embedded system.



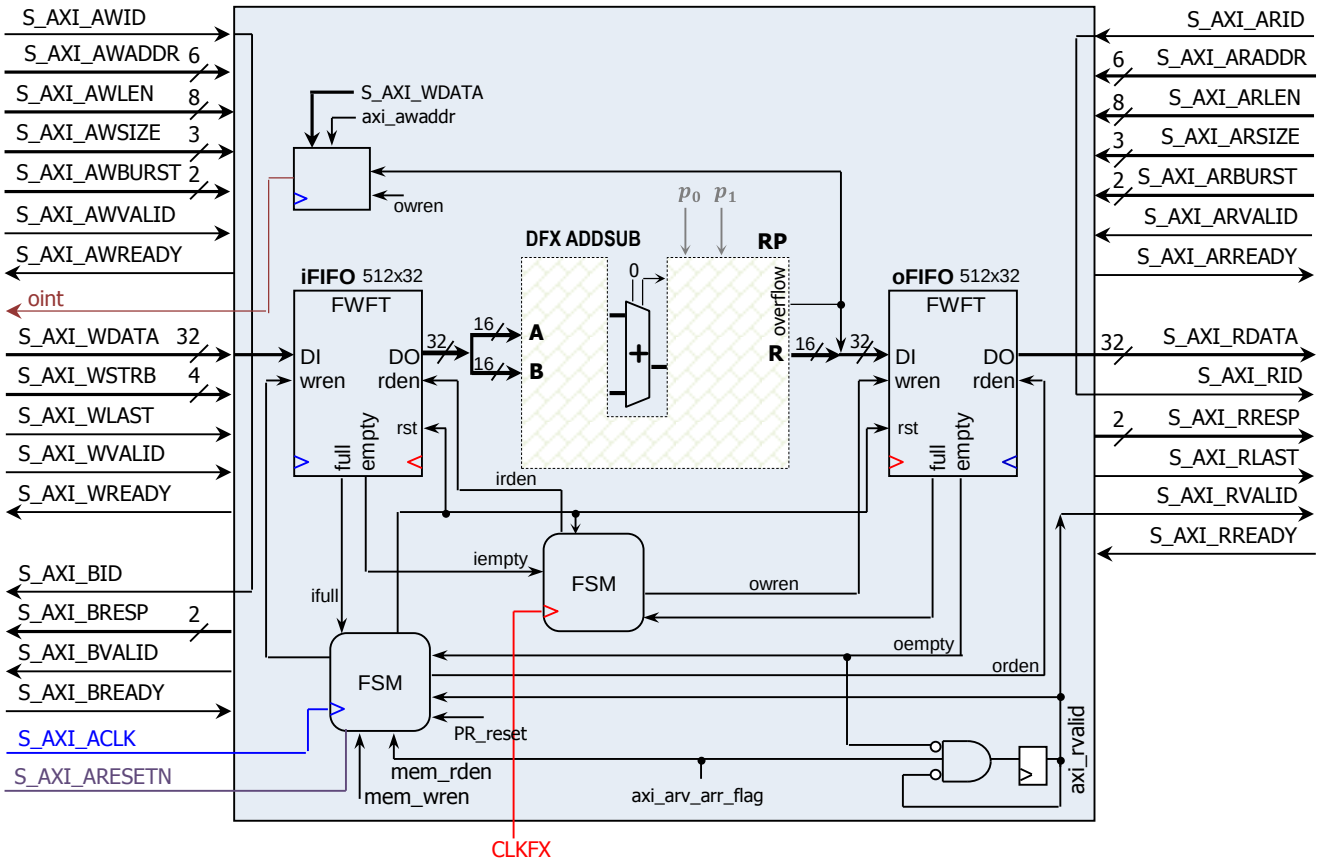
DUAL FIXED-POINT ADDER SUBTRACTOR

- This combinational circuit computes DFX addition/subtraction. We note that the FX adder/subtractor does not depend on p_0 and p_1. Thus, we can modify the DFX format (p_0, p_1) by modifying only the pre-scaler and post-scaler. The figure depicts how we partitioned the design into static and dynamic (run-time alterable) components.



AXI4-FULL INTERFACE

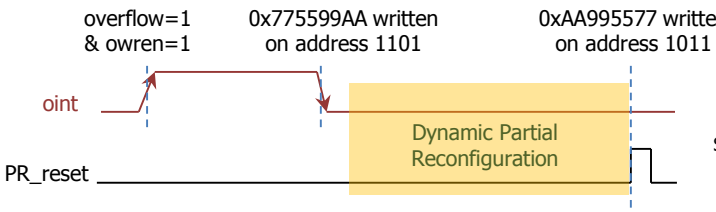
- For simplicity's sake, we set $N = 16$ and $addsub = 0$ for the DFX Adder/Subtractor. This effectively makes a 16-bit DFX adder. The figure below depicts the AXI4-Full interface. The interface also includes the output interrupt $oint$ signal.

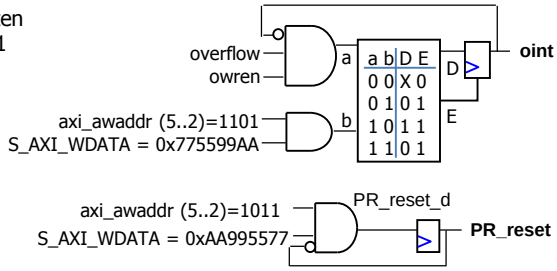


Interrupt and Partial Reconfiguration Control:

- The AXI4-Full interface generates an interrupt signal $oint$:
  - ✓ The $oint$ signal is asserted when an overflow is detected ($overflow = 1$) and when data is valid on the output of the DFX Adder (this happens when $owren = 1$, see FSM @ CLKFX).
  - ✓ The interrupt signal remains asserted until the PS detects it. At this point, the ISR de-asserts the interrupt signal (so that the signal does not continuously interrupt the PS). This is performed by writing a specific word (`0x775599AA`) on address 110100. Make sure that when writing to this peripheral, we must avoid writing on address 110100, otherwise it might write an undesired word (`0x775599AA`) on the iFIFO.

- After $oint$ is de-asserted, we are free to execute dynamic partial reconfiguration (DPR).
  - ✓ Usually, because of RP output toggling, the input data of oFIFO and the $oint$ block vary randomly. In this design, note that during DPR, $owren = 0$ (no word is written onto oFIFO); this prevents $oint$ from being altered unintendedly.
  - ✓ After Partial Reconfiguration, we have to reset the FIFOs (in case data was still written between the overflow and the start of DPR) and the PR FFs (nonexistent in this example). This is carried out by the signal PR_reset.
  - ✓ PR_reset: This one-cycle pulse resets both the RP FFs (non-existent in this example) and the FIFOs via a simple software command (we write the word `0xAA995577` on address 101100). Make sure that when writing to this peripheral, we must avoid writing on address 101100, otherwise it might trigger an undesired PR_reset.

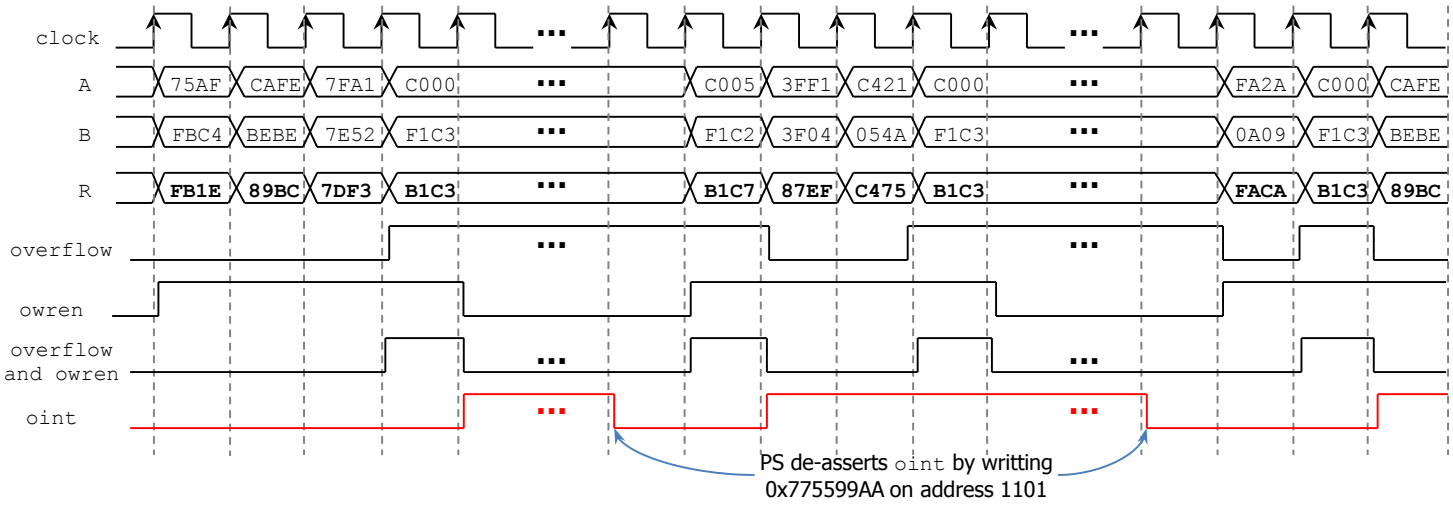




- Unlike the case of the Pixel Processor, here we cannot use AXI reads to de-assert the interrupt: If we issue an AXI read, we expect data from the FIFO to be read. If the FIFO is empty or if we read unintended data, the system will stall.

Timing diagram for *oint*

- We depict *oint* assertion and de-assertion for a DFX adder in format [16 8 4]. If $overflow = owren = 1$ then $oint = 1$. During this time, other overflows are ignored. *oint* is only de-asserted if we write `0x775599AA` on 110100.
- In the example, data is read and written in bursts of four 32-bit words. The inputs `A` and `B` fit in a 32-bit word, while the output `overflow&R` fit in a 32-bit word. The output is processed in one clock cycle and we can use $owren = 1$ to indicate valid data. Because of the design, the last data is kept on the inputs of the DFX adder until the next burst.
- We show a particular case when the last data in the first burst generates an overflow (note how $overflow = 1$ until the next burst includes a case with no overflow). Also, note how the first word of the second burst also generates an overflow. To detect this second overflow, we need $owren = 1$ (as $overflow = 1$ during this entire time). In general, to properly detect an overflow, we need: $overflow = owren = 1$.
- Note that when *oint* is asserted, any subsequent overflow is ignored until *oint* is de-asserted. In the second burst, the first and the third data generates and overflow, but only the first generates *oint*, an the second overflow is ignored.

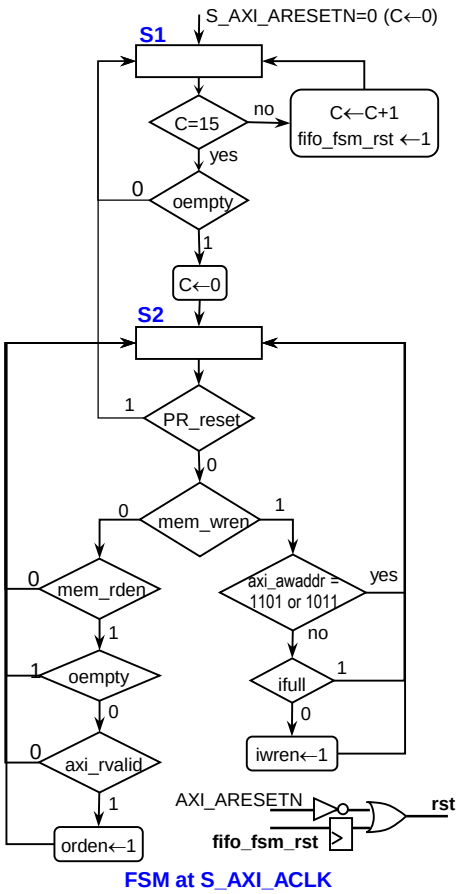


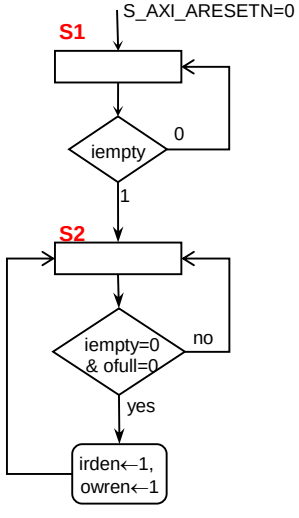
AXI4-Full Interface Control

- The FSM @ S_AXI_ACLK is shown. As a precaution, we do not allow writes on iFIFO for addresses 101100 and 110100. For example, when de-asserting *oint*, we do not want the word `0x775599AA` to be written on iFIFO.
  - ✓ Since writes on address 101100 or 110100 are ignored, we have to be careful not to attempt to write on those addresses and then try to read as the system will freeze. This is straightforward when using simple write commands. But when using DMA, make sure that the BurstType is set to FIX (not to INCR or WRAP), otherwise the address will be automatically increased and it might reach 110100 or 101100, this will cause the system to ignore some words. This can be indicated in the DMA parameter DmaCmd→ChanCtrl.DstInc. By default it is 1 (Inc), make it 0 (Fix).

- We also depict the FSM @ CLK_FX. This is a very simple FSM as the DFX adder and its interface is purely combinational.

- Finally, note that when writing software applications that involve DMA (or large write loops), the challenge is to spot where the overflow occurred. Then if we reconfigure, we need to re-start processing from a certain point.





TESTING SCHEME

- We use two variations for the 16-bit DFX adder: formats [16 8 4] and [16 7 2].
- We show the 3 datasets (9 data points each) along with their DFX format:

FIRST DATASET

DFX format	Input A&B	Output overflow&R
[16 8 4]	0x75AFFBC4	0x0000FB1E
	0xCAFEBEBE	0x000089BC
	0x7FA17E52	0x00007DF3
	0x3FF13F04	0x000087EF
	0xC421054A	0x0000C475
	0xFA2A0A09	0x0000FACA
	0xC000F1C3	0x0001B1C3
	0xD001F170	0x0000C171
	0xFAF8300A	0x00005F8A

SECOND DATASET

DFX format	Input A&B	Output overflow&R
[16 7 2]	0x75AFFBC4	0x0000FB71
	0xCAFEBEBE	0x000089BC
	0x7FA17E52	0x00007DF3
	0x3FF13F04	0x000083F7
	0xC421054A	0x0000C44B
	0xFA2A0A09	0x0000FA7A
	0xC000F1C3	0x0001B1C3
	0xD001F170	0x0000C171
	0xFAF8300A	0x0000FC78

THIRD DATASET

DFX format	Input A&B	Output overflow&R
[16 7 2]	0x78D75E20	0x000056F7
	0xF2BF8FAF	0x0000826E
	0x7FD07F29	0x00007EF9
	0x1FF81F82	0x00003F7A
	0xF10802A5	0x0000F11D
	0x51500504	0x00005654
	0xF000FC70	0x0000EC70
	0xF400FC5C	0x0000F05C
	0x57C01805	0x00006FC5

Test Procedure

- Extract the **`axiaddsub16_dprsys.zip`**. It includes three folders and the software application.
  - ✓ `/axiaddsub16_dr`: Files for implementing the AXI4-Full DFX Add/Sub Peripheral.
  - ✓ `/axiaddsub16_dr_static`: Files for implementing the static portion of the AXI4-Full DFX Add/Sub Peripheral.
  - ✓ `/dfxaddsub16_dyn`: Folder structure for implementing the self-reconfigurable system.
    - We include the file `/Synth/Static/top_synth.dcp`, which is the Checkpoint of the Static Region.
    - We include the file `/Sources/xdc/top.xdc`. It contains the RP constraints and the PS constraints.
      `top.xdc = design_1_processing_system7_0_0.xdc + fplan.xdc.`
      - System tested on ZYBO Z7-10 and Vivado 2019.1. For other boards, you need to generate your own `top.xdc` and `top_synth.dcp` files as per the procedure in Embedded System Design for PSoC Tutorial → Unit 7: DCT 2D.
  - ✓ test_dfxadd_rp.c, xtra_func.h: Software application to test the self-reconfigurable system.

- Vivado Tcl Shell: `source design_complete.tcl -notrace`↵. This will generate the bitstreams (`.bit`). Follow the procedure in Embedded System Design for PSoC Tutorial → Unit 7: DCT 2D to generate the byte-swapped `.bin` partial bitstream files. Copy them onto the SD card and rename them to `dfx6_2.bin, dfx7_2.bin, dfx8_4.bin.`
- Create an embedded system (`dfxaddsub16_drsys`) for the AXI4-Full DFX Add/Sub Peripheral (`mydfxaddsubintr`). Create an SDK project (add the software files). Enable the '`xilffs`' library and the string manipulation functions. Allocate space for the heap and stack.
- In Vivado, program the full bitstream for the [16 8 4] configuration: `Config_dfxadd8_4.bit`.
- Run the software application in SDK:
  - ✓ With the DFX adder in format [16 8 4], the first dataset is tested. The results should match the shown output. An overflow will be generated by data `C000 + F1C3`. This will generate an interrupt. The ISR only de-asserts the interrupt signal oint. Once the 9 data points are processed (results retrieved), if the interrupt was issued, the software routine will reconfigure the DFX adder to the format [16 7 2].
  - ✓ With the DFX adder in format [16 7 2], the second dataset is tested. Note that these are the same binary values as in the case for [16 8 4], but data is treated as in the format [16 7 2]. We do this to demonstrate that we successfully performed reconfiguration (note that the output results are different). Here, overflow is also detected, an interrupt is issued (the ISR de-asserts oint), but reconfiguration is not performed.
  - ✓ With the DFX adder still in format [16 7 2], the third dataset is tested. These are the same real values of the first dataset but represented in format [16 7 2]. Here, no overflow is generated.
  - ✓ Finally, we unconditionally reconfigure the DFX adder back to the format [16 8 4] and run the first data set. An overflow is detected, an interrupt is issued (the ISR de-asserts oint), but reconfiguration is not performed.