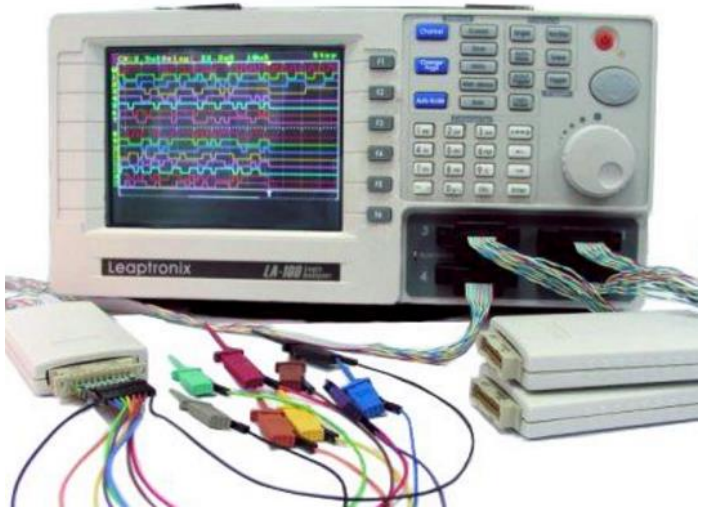
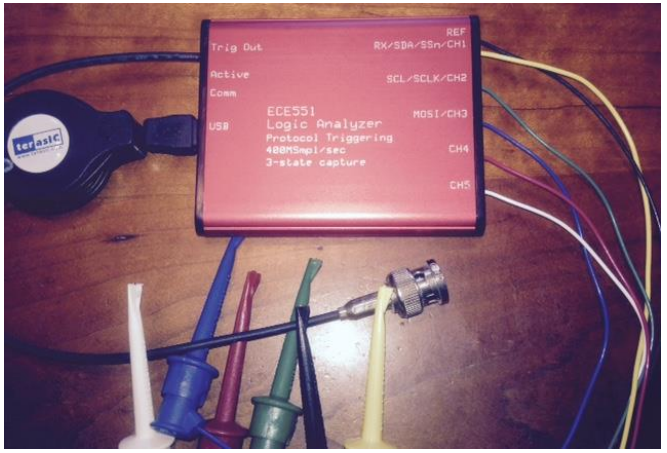
ECE 551
Project Spec

Spring ‘20



Logic Analyzer



Grading Criteria: (Project is 28% of final grade)

- Project Grading Criteria:
  - Quantitative Element 15%

  (yes this could result in extra credit)

$$Quantitative = \frac{EricWangnan\_Project Area}{YourSynthesizedArea}$$

Note: The design has to be functionally correct for this to apply

- Project Demo (85%)
  - ✓ Code Review (10%)
  - ✓ Testbench Method/Completeness (15%)
  - ✓ Synthesis Script review (7.5%)
  - ✓ Post-synthesis Test run results (10%)
  - ✓ Results when placed in EricWangnan Testbench (25%)
  - ✓ Test of Logic Analyzer mapped to DE0-Nano to set of test signals (10%)
  - ✓ APR of toplevel (7.5%)

Extra Credit Opportunity:

Appendix C of ModelSim tutorial instructs you how to run code coverage

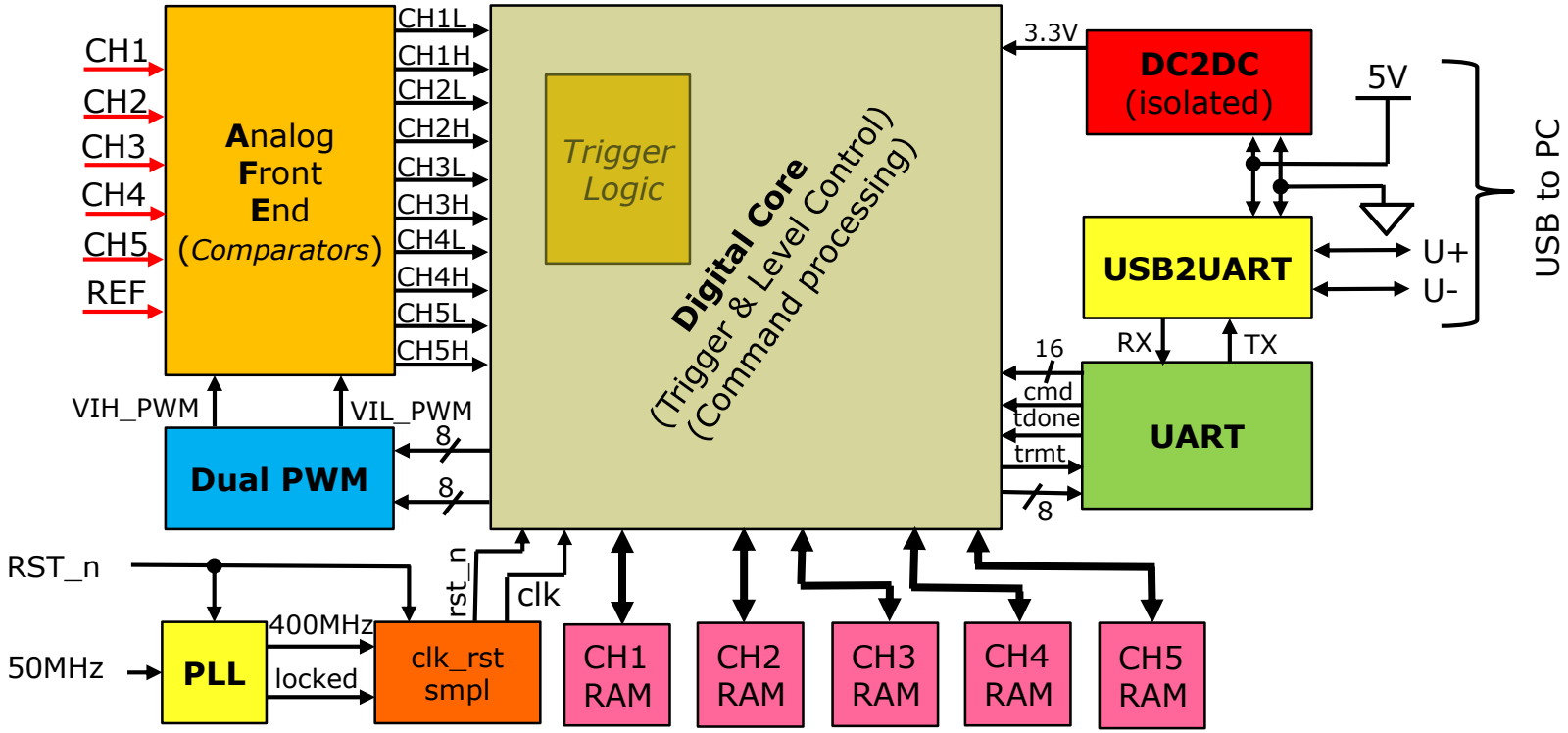
- Run code coverage on a single test and get 1% extra credit
- Run code coverage across your test suite and get a cumulative number and get 2% extra credit.
- Run code coverage across your test suite and give concrete example of how you used the results to improve your test suite and get 3% extra credit.

Project Due Date

- Project Demos will be held in B555:
  - Weds (4/29/20) from 4:00PM till evening.
  - Thurs (4/30/20) from 1:00PM till evening.
  - Fri (5/1/20) from 4:00PM till evening

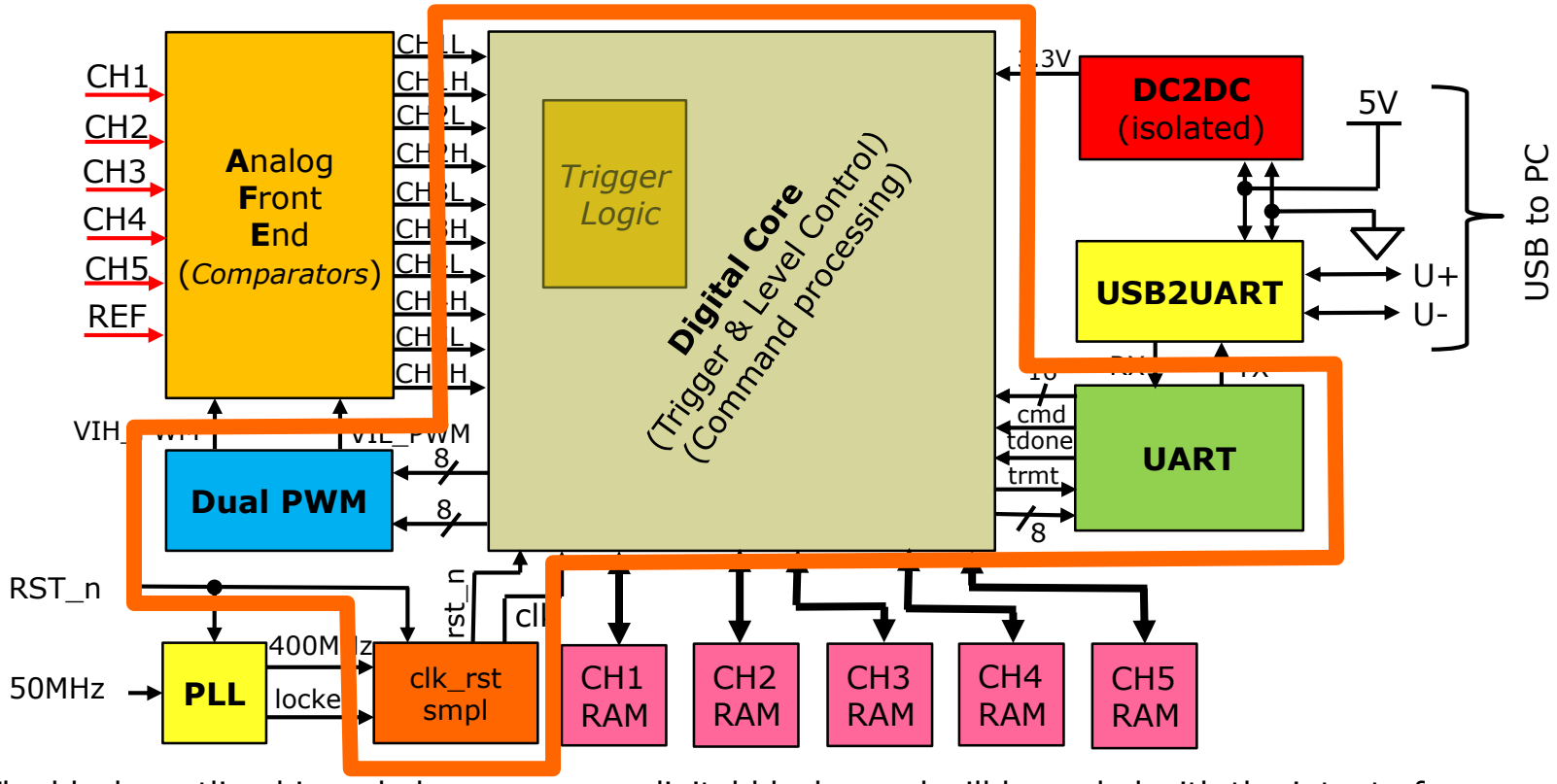
- Project Demo Involves:
  - ✓ Code Review
  - ✓ Testbench Method/Completeness
  - ✓ Synthesis Script & Results review
  - ✓ Post-synthesis Test run results
  - ✓ Results when placed in EricWangnan testbench
  - ✓ Results when tested on real signals on DE0 Platform
  - ✓ Review of APR results

Block Diagram



We will be building a 5-channel logic analyzer. Each channel will be have a VIH and VIL threshold so we have the ability to detect if the logic signal in a given channel is mid range. The triggering logic will be flexible and can be a function of edges or levels of the 5 input channels. There will also be protocol based triggering (SPI & UART). The digital core stores each channel sample to its respective RAM bank. The sample rate is 400MSmpl/sec. The RAMs will be configured as byte wide for each channel, which means 4 samples per memory location is stored. The memory depth should be a parameter. The digital core also performs the task of reading the raw samples from the RAMs and sending the data to a USB port for display by an app running on a host PC. The host application also sets the trigger conditions by sending commands via USB. UART is an intermediate protocol for the host interface.

What is synthesized DUT vs modeled?

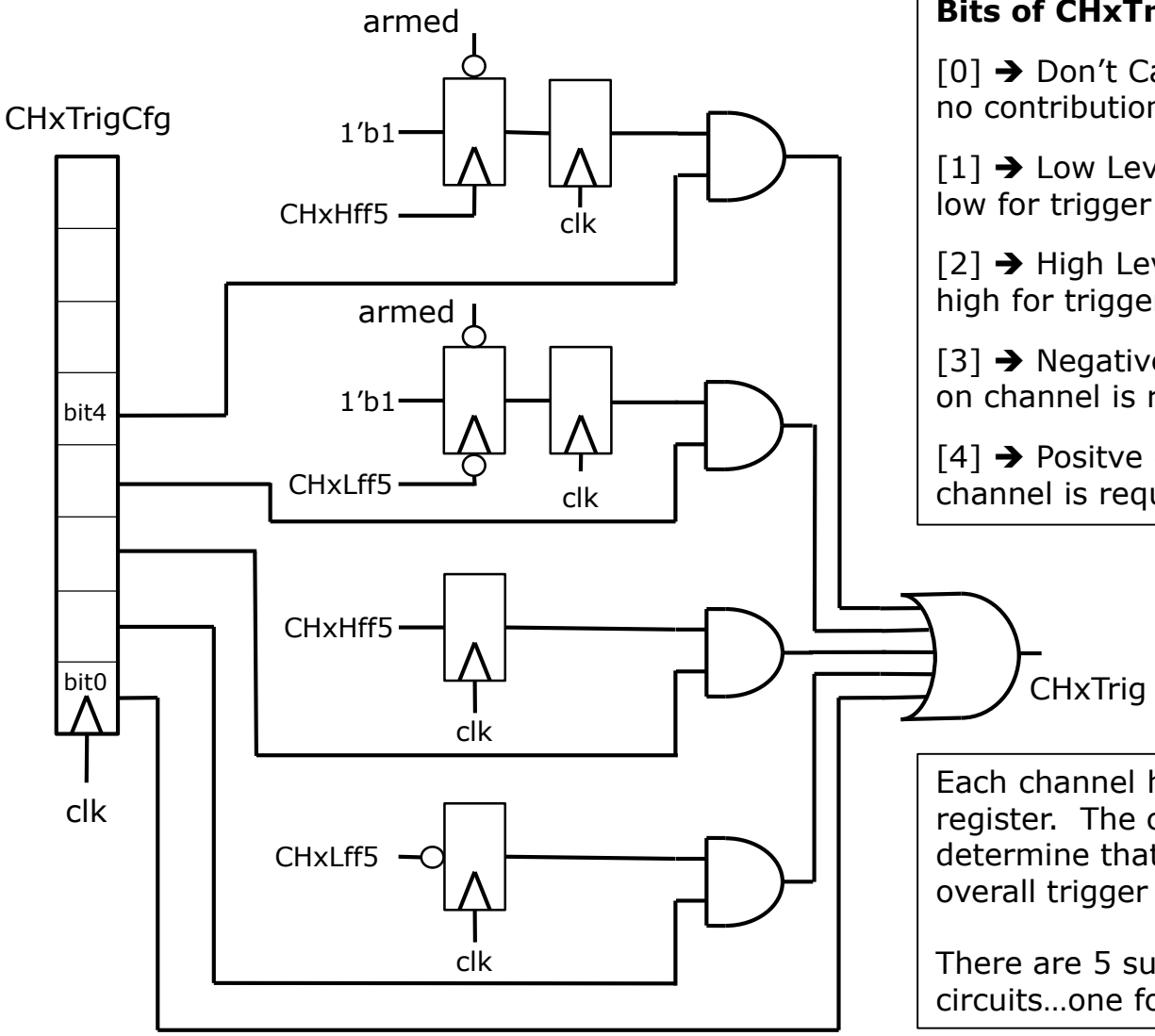


The blocks outlined in red above are pure digital blocks, and will be coded with the intent of being synthesized & APRed.

You Must have a block called **LA_dig.v** which is top level of what will be the synthesized plus the 5 RAM blocks.

The 5 RAM blocks are not synthesized, but you will be creating a verilog model for these blocks, and they do map to internal blocks in the Altera Cyclone IV on your DE0-Nano board. A verilog model of the AFE will be provided.

Channel Trigger Logic



Bits of CHxTrigCfg *(where x is 1,2,3,4,5)*

[0] ➔ Don't Care. If set this channel makes no contribution to the overall trigger function.

[1] ➔ Low Level. If set this channel must be low for trigger to occur.

[2] ➔ High Level. If set this channel must be high for trigger to occur.

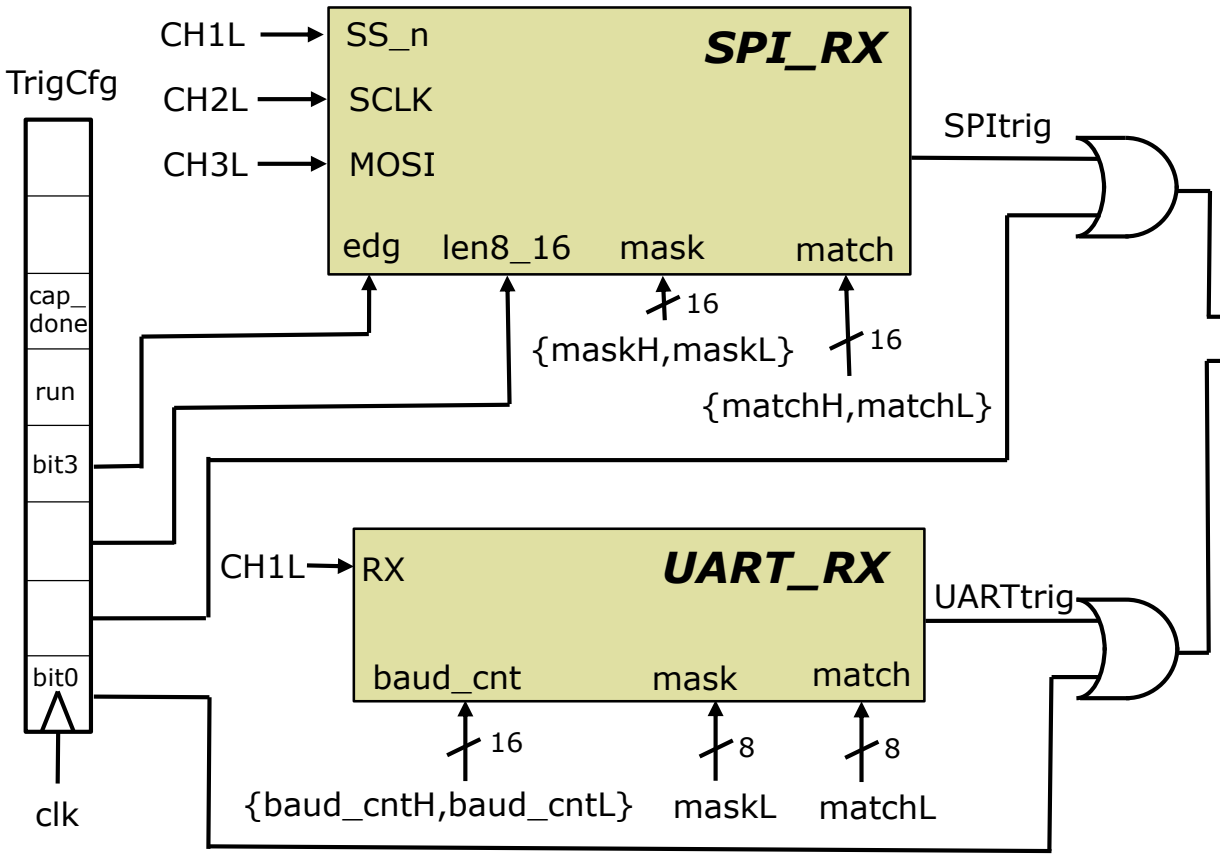
[3] ➔ Negative edge. If set a negative edge on channel is required for trigger to occur.

[4] ➔ Positve edge. If set a positive edge on channel is required for trigger to occur.

Each channel has a trigger configuration register. The contents of this register determine that channel's contribution to the overall trigger function.

There are 5 such registers, and associated circuits…one for each channel.

Protocol Trigger Logic

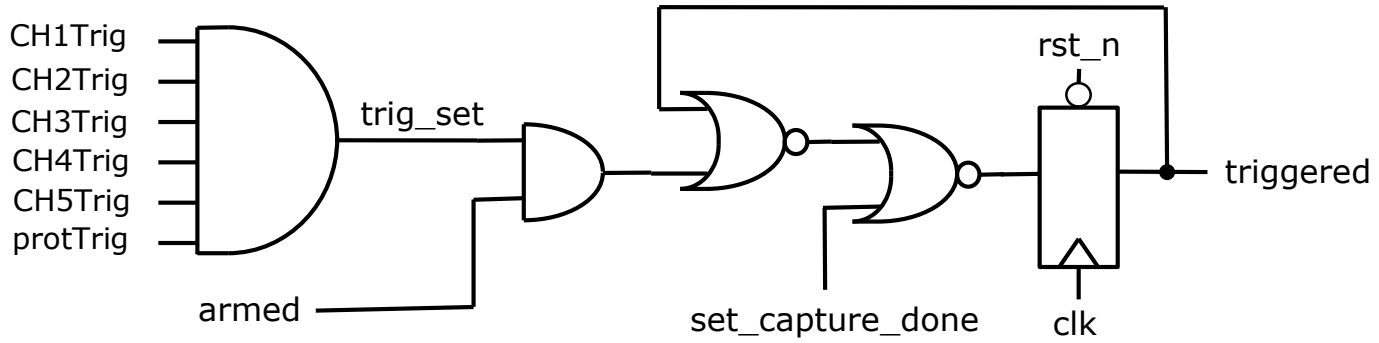


UART triggering is based off an 8-bit data match. The RX line is always provided by CH1. A 16-bit baud rate counter allows for baud rates from 2400 to 921600

Bits:	Description:
[0]	Disable UART protocol trigger
[1]	Disable SPI protocol trigger
[2]	SPI triggering is based on 8-bit word
[3]	SPI captures MOSI on positive edge
[4]	Run bit. If set capture occurs on trigger
[5]	**capture_done**. Set by hardware when capture is complete, cleared via command to enable next capture.

Triggering can be qualified by a data match from either a SPI or UART protocol. For SPI triggering the MOSI sampling can be in rising or falling edge of SCLK. Length of data to match can be 8 or 16 bits (1➔8-bit, 0➔16-bit). CH1 is always SS_n, CH2 is always SCLK and CH3 is always MOSI. A more detailed description of the requirements of SPI_RX and UART_RX can be found later in this spec. The lower 4-bits of the TrigCfg register determine protocol triggering functionality.

Trigger Logic (continued):

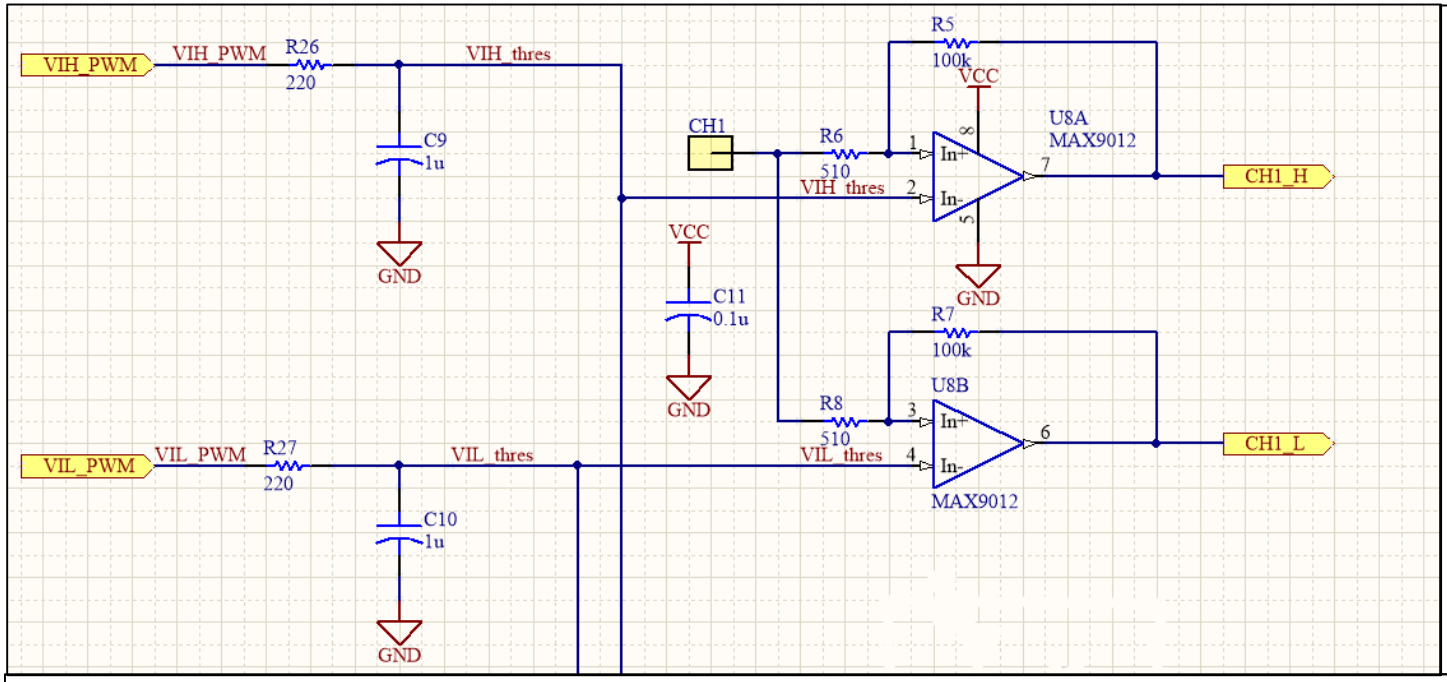


The trigger pulse (***trig_set***) may only be one clock cycle wide. This signal needs to be captured and preserved in an **S**et/**R**eset flop mechanism so a consistent signal (***triggered***) can be evaluated by the capture module.

The signal ***armed*** is asserted by the capture logic, and only allows a trigger to occur when the number of samples captured plus the number of samples to capture after trigger (***trig_pos***) will exceed the total sample memory size (parametizable number set to 384 for simulation and 12288 for DE0-Nano mapping).

armed is cleared and ***set_capture_done*** is asserted when the number of samples post trigger equal ***trigger_pos***. At this time capture is over and the next logical step would for the channel memories to be read by the host over the USB interface.

VIH/VIL Setting: (Dual PWM)

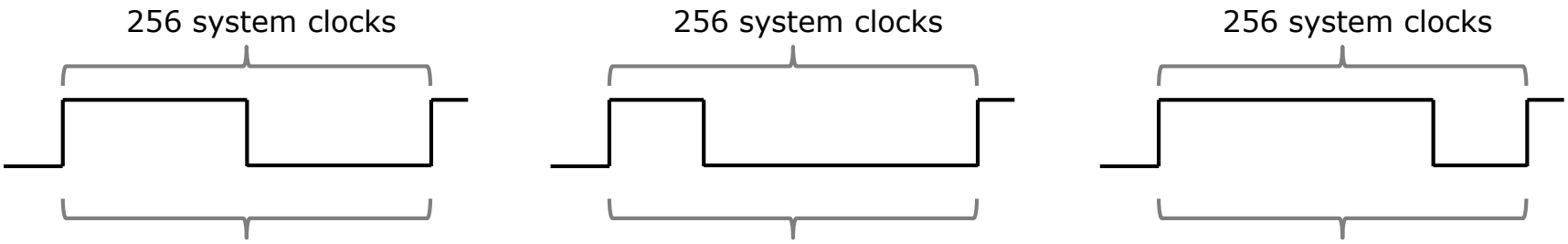


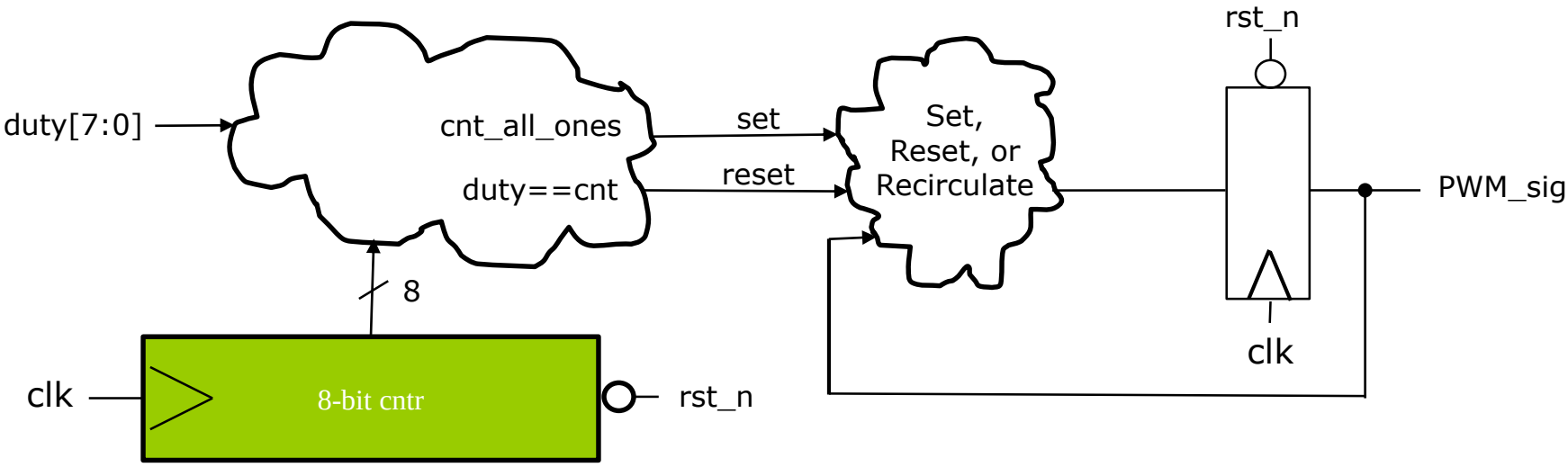
This portion of the AFE schematic shows that CH1 is compared against a VIH level and a VIL level to produce two signals CH1_H and CH1_L. If both are high CH1 is considered high. If both are low CH1 is considered low. If they mismatch then CH1 is considered mid rail.

The VIL and VIH levels (VIH_thres & VIL_thres) would need to adjust based on the DUT the logic analyzer is analyzing. If the DUT has 5V digital signals then perhaps VIH should be 66% of 5V (ie. 3.3V) and VIL should be 33% of 5V (1.65V). If, however, the DUT was powered with 1.8V then perhaps VIH & VIL levels of 1.2V and 0.6V would make more sense.

A simple inexpensive way to generate an analog signal of varying levels is to low pass a PWM (Pulse Width Modulated) signal. The 220Ω resistor and 1μF cap form a low pass. We will drive VIH_PWM and VIL_PWM with PWM signals to allow for variable VIH and VIL levels.

What is PWM

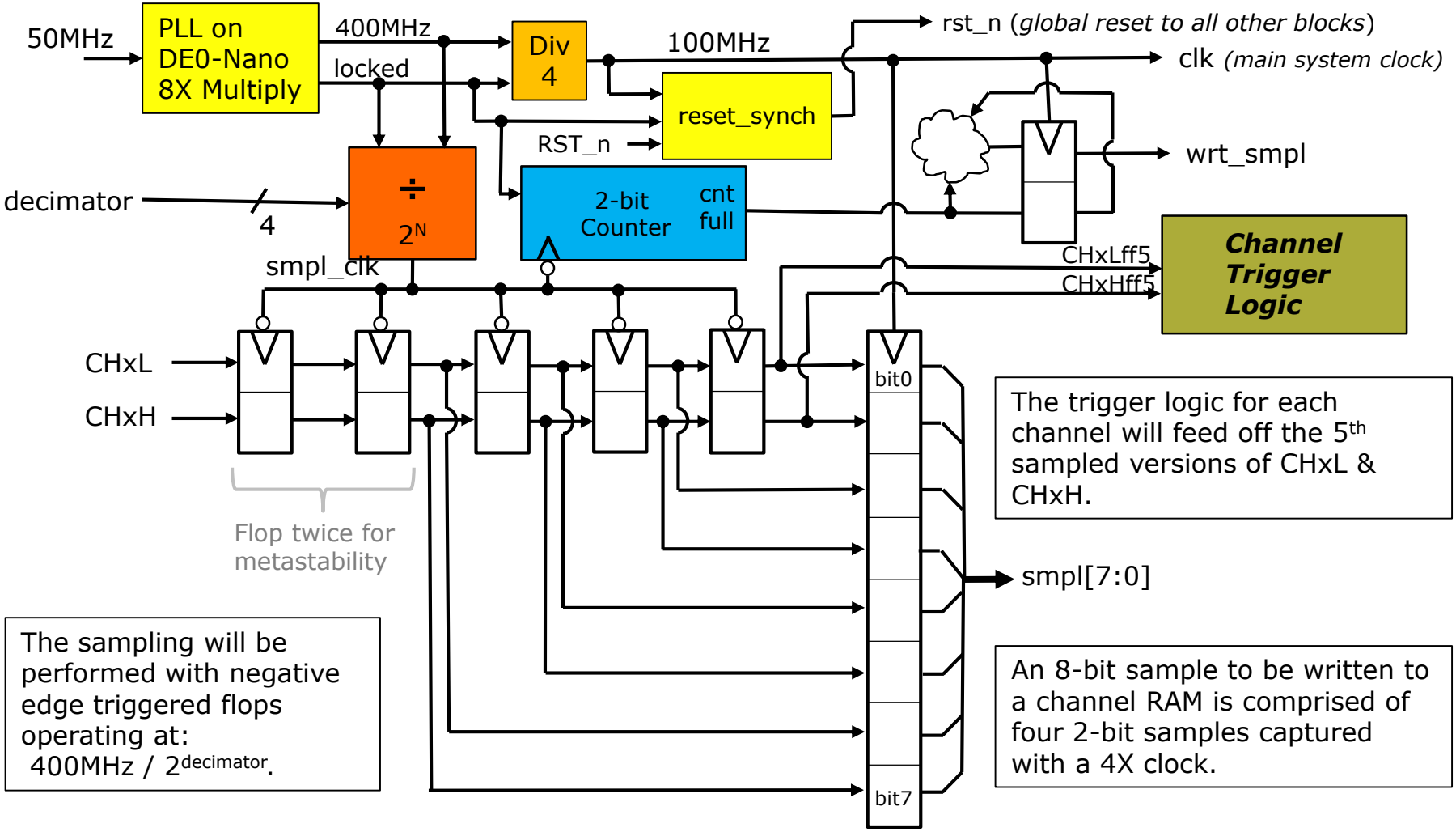




Above is a diagram of a simple 8-bit PWM module. The frequency will be fixed at: 100MHz/256 = 390kHz. You will need a dual PWM module that essentially has two 8-bit PWM modules producing VIH_PWM and VIL_PWM.

Capture Logic (50MHz to 400MHz to 100MHz)

The DE0-Nano receives a 50MHz clock. We want the main system clock to be 100MHz. However we want a capture rate of up to 400MHz. This will be accomplished as shown in the diagram below.



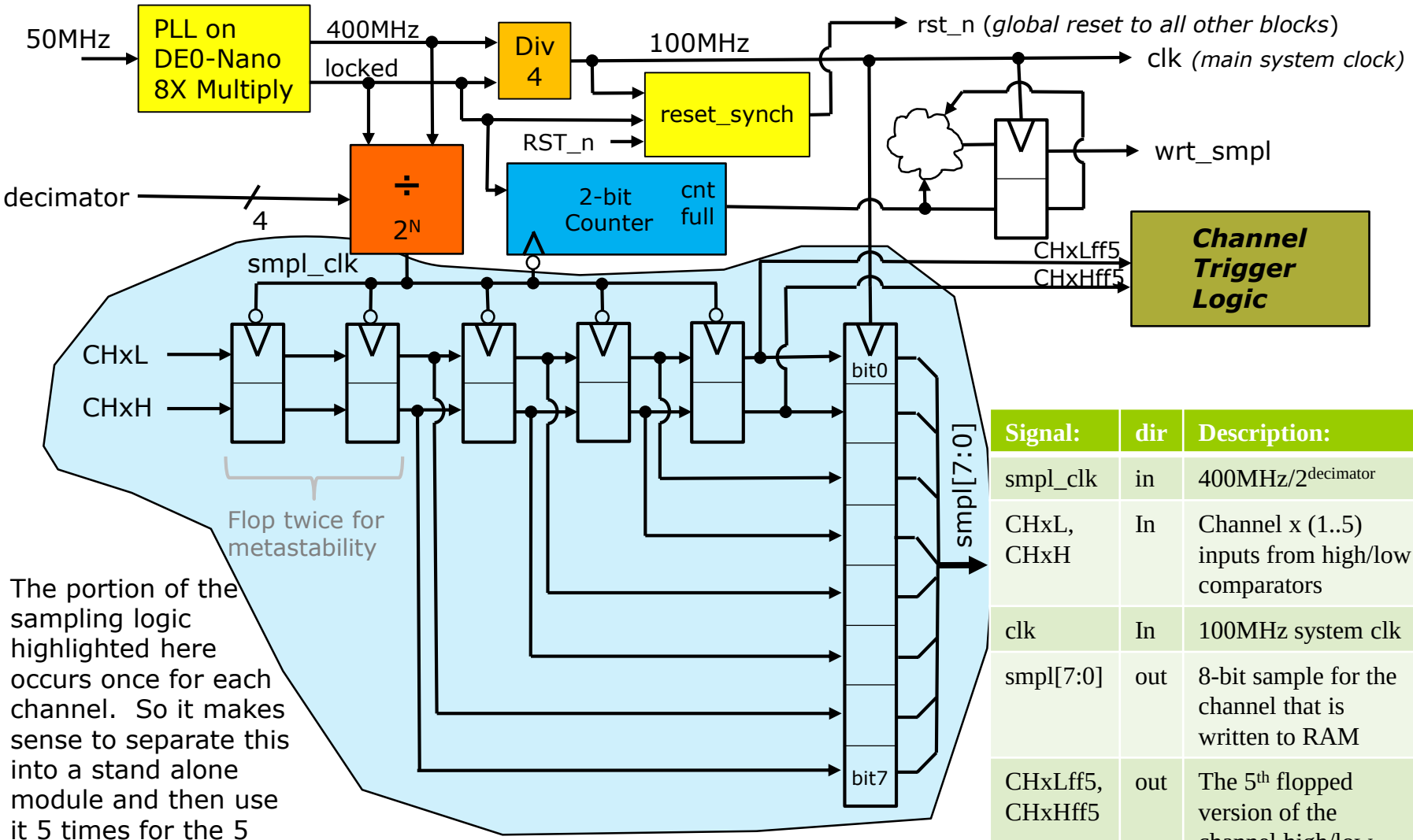
The trigger logic for each channel will feed off the 5th sampled versions of CHxL & CHxH.

The sampling will be performed with negative edge triggered flops operating at:
400MHz / $2^{decimator}$.

An 8-bit sample to be written to a channel RAM is comprised of four 2-bit samples captured with a 4X clock.

A 2-bit counter working off the smpl_clk will determine on what clock cycles samples are written to RAM. If demicator = 0 that would be every clock cycle of the system clock.

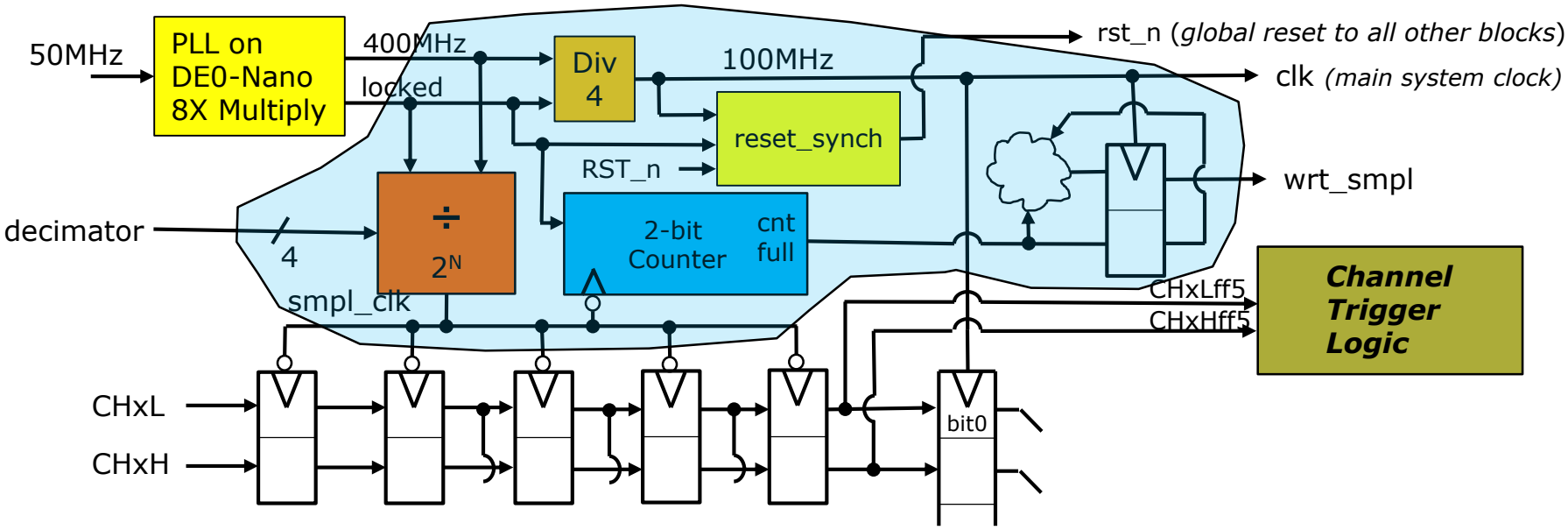
Channel Sample Logic



The portion of the sampling logic highlighted here occurs once for each channel. So it makes sense to separate this into a stand alone module and then use it 5 times for the 5 channels.

Signal:	dir	Description:
smpl_clk	in	400MHz/$2^{decimator}$
CHxL, CHxH	In	Channel x (1..5) inputs from high/low comparators
clk	In	100MHz system clk
smpl[7:0]	out	8-bit sample for the channel that is written to RAM
CHxLff5, CHxHff5	out	The 5th flopped version of the channel high/low used for trigger logic

clk_rst_sample Block



The block functionality highlighted above that performs clock division, generates **smpl_clk**, and determines if samples are valid to be written to RAM (**wrt_smpl**) is hard to functionally specify and very tricky to have it work out in the DE0-Nano at 400MHz. For this reason the code for this functionality is provided and can be downloaded from the course website.

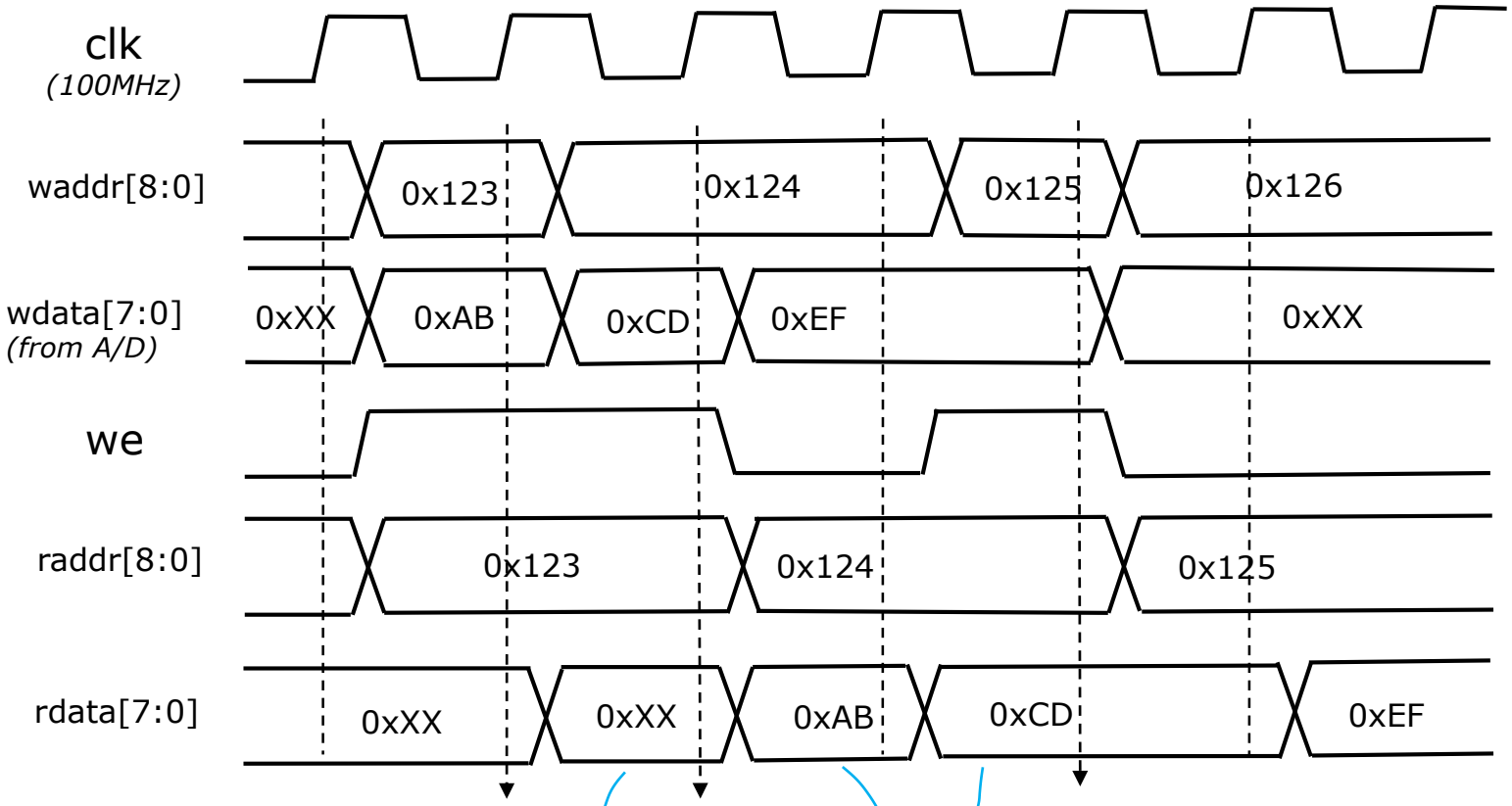
The code for the highlighted block is called ***clk_rst_smpl.sv*** and is part of the provided project collateral.

RAM Blocks (store captured channel data)

There will be 5 dedicated RAM blocks to store the data captured from each channel. Each of these RAM blocks are 8-bits in width with a parametizeable number (384 for simulation, 12288 for DE0 mapping) of entries. The RAM block are dual ported with a read port and a write port. Although the RAMs themselves are capable of a read and write in a given clock cycle we will not use them that way. We will either be writing or reading in a given clock.

Signal:	Dir:	Description:
clk	in	Clock. Address and control signals setup after rising edge of clock. Read or write occurs on next positive edge of clock.
we	in	Active high write enable. Write occurs on next positive clock edge
waddr[8:0]	in	Write address to select location to be written.
wdata[7:0]	in	Write data. Drive this with the sample data to be written to RAM.
raddr[8:0]	in	Read address to select location to be read. Data available after next positive clock edge
rdata[7:0]	out	Read data. Data is available after next positive edge of clock. Read occurs every clock cycle by default and do not require an enable signal.

RAM Block Timing



0xAB written to 0x123 here, but too late for first read of 0x123

0xCD written to 0x124 here

0xEF written to 0x125 here

Location 0x123 was just written with 0xAB on previous rising edge same edge as **rdata** was being flopped, so rdata is older contents of 0x123 (unknown to us)

This is read of location 0x123 which was **raddr** the previous cycle

This is read of location 0x124 which was **raddr** the previous cycle

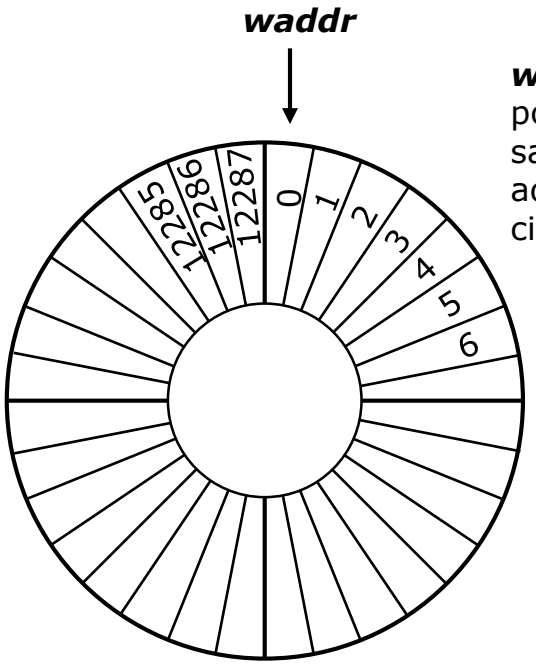
Use RAMs as Circular Queue

As you figured out in HW1, a circular queue structure is the way to go for storing samples.

Our case is a little simpler than a generic circular queue, because we know the queue will always be full. We always capture ***trigger_pos*** samples after the trigger. However, the trigger is not armed until we have already captured **ENTRIES**-***trigger_pos*** samples. So at a minimum we will always have captured **ENTRIES** number of samples in our circular queues.

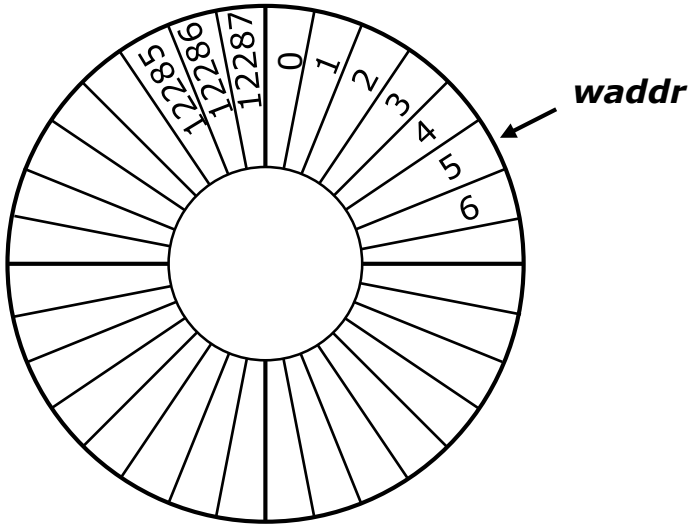
Recall **ENTRIES** is a parameter that specifies the RAM size. It will be 384 for simulation, and 12288 when we map it to the DE0-Nano board.

NOTE: we also write all our samples to the queue before reading any of them back. When we read samples out we read them all at once.



waddr initially would point to location 0. As samples are written it advances around the circle

In this case we were done with capture after having written 12293 samples.



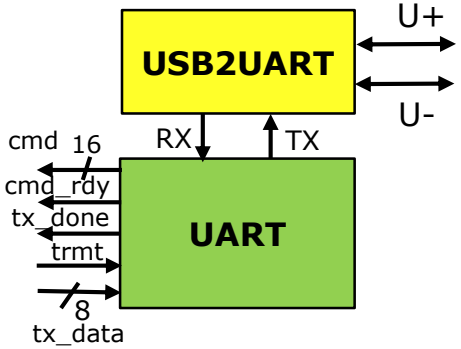
UART Interface

- UART serial interface will be used to communicate with host PC (through USB interface chip)

- Program running on host PC will configure settings
  - VIH/VIL levels
  - Trigger configuration
    - Channel trigger sources
    - Protocol trigger setup
  - Sampling rate (400MSmpl/sec, 200MSmpl/sec, 100MSmpl/sec, …)

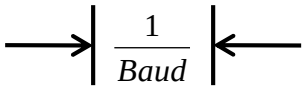
- Program running on host PC will read and display the dgital waveform data
  - Issues commands to transfer data from each RAM block to UART interface

Signal:	Dir:	Description
clk,rst_n	in	Clock and reset. 100MHz system clock, and active low reset
tx_data[7:0]	in	Data to transmit to host. Typically a positive acknowledge to a command, or a stream of bytes representing requested channel data
trmt	in	This signal is pulsed high for at least one clock cycle to start transmission
tx_done	out	Asserted when ***tx_data[7:0]*** has finished transmitting
cmd_rdy	out	Asserted when a new 16-bit command has been received from the host PC
cmd[15:0]	out	16-bit command
RX/ TX	in/ out	Serial data out (to USB chip, then to host PC) Serial data in (from USB chip, from host PC)

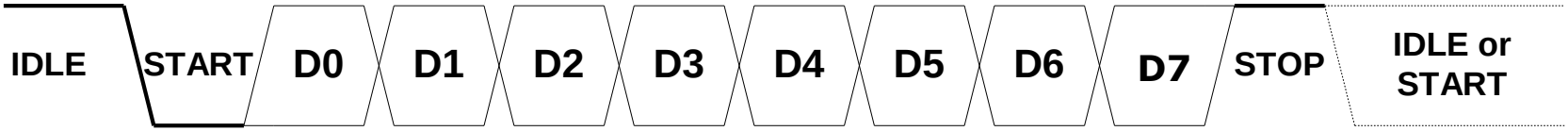


What is UART (RS-232)

- RS-232 signal phases
  - Idle
  - Start bit
  - Data (8-data for our project)
  - Parity (no parity for our project)
  - Stop bit – channel returns to idle condition
  - Idle or Start next frame



Baud rate will be 115,200 with 50MHz clock



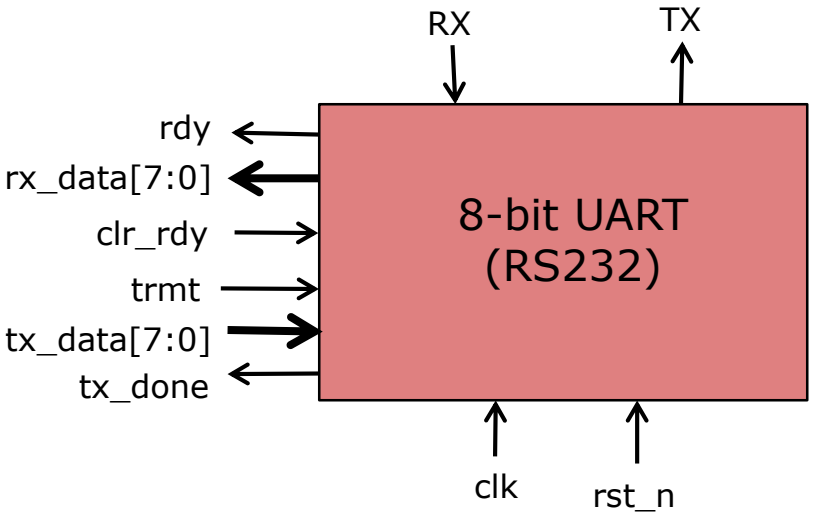
- Receiver monitors for falling edge of Start bit. Counts off 1.5 bit times and starts shifting (right shifting since LSB is first) data into a register.
- Transmitter sits idle till told to transmit. Then will shift out a 9-bit (start bit appended) register at the baud rate interval.

What is RS232 (UART)

- Full RS-232 involves handshaking signals
  - RTS, CTS, DTR, …
  - We will only use RX & TX assuming sender and receiver can "digest" bytes fast enough. Common use.

- When 8-bit data received UART will raise ***rdy*** and data will be presented on ***rx_data***[7:0].

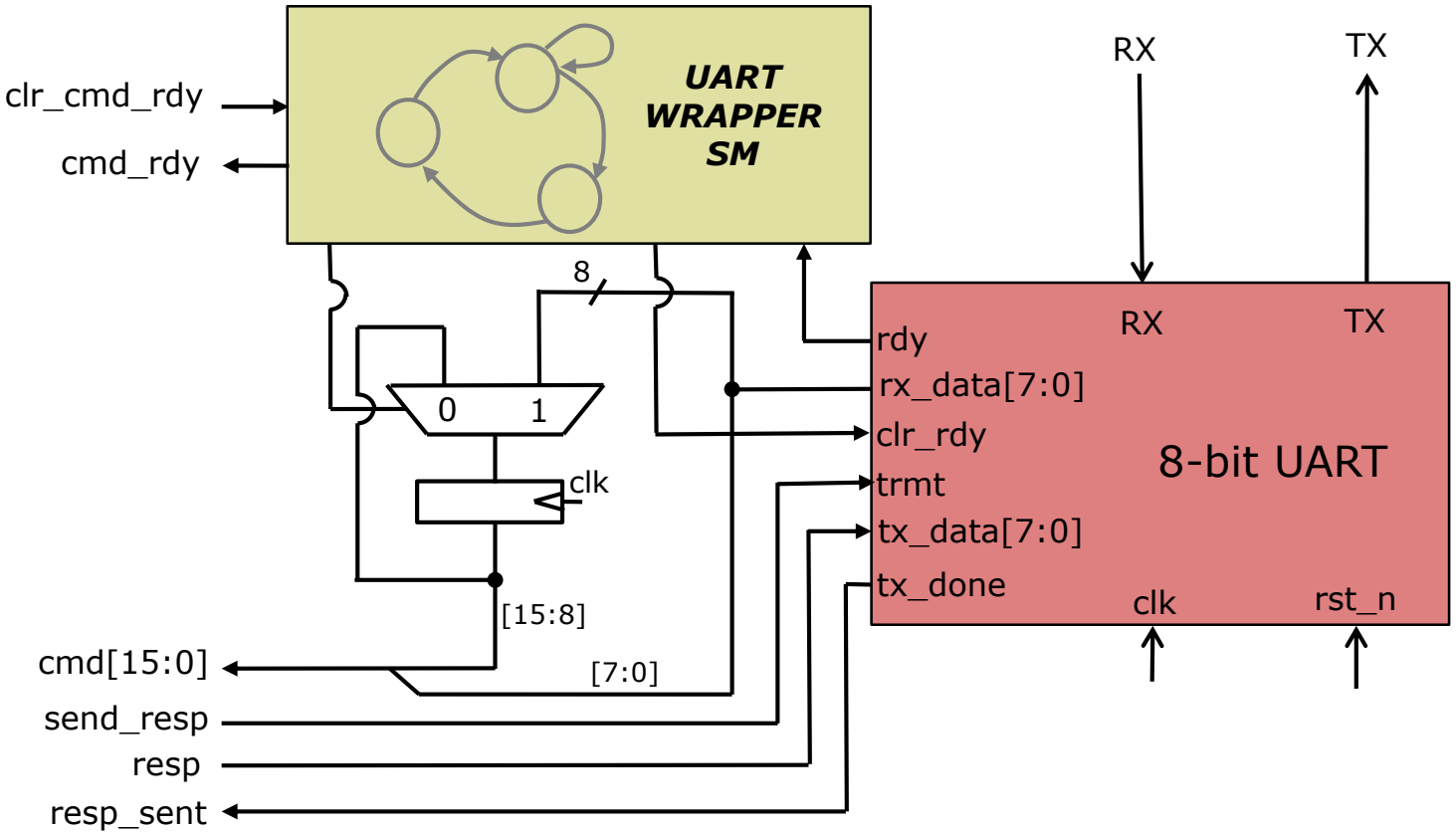
- When core has consumed data it will raise ***clr_rdy***, to indicate to the UART it should drop ***rdy***.



- To transmit the core will be present data on ***tx_data***[7:0] and then raise ***trmt***. When the UART is done transmitting it will raise ***tx_done***.

UART Wrapper

- Previous slides showed an 8-bit UART transceiver. We want to put a wrapper around it to make a unit that receives 16-bit command packets and sends 8-bit responses.



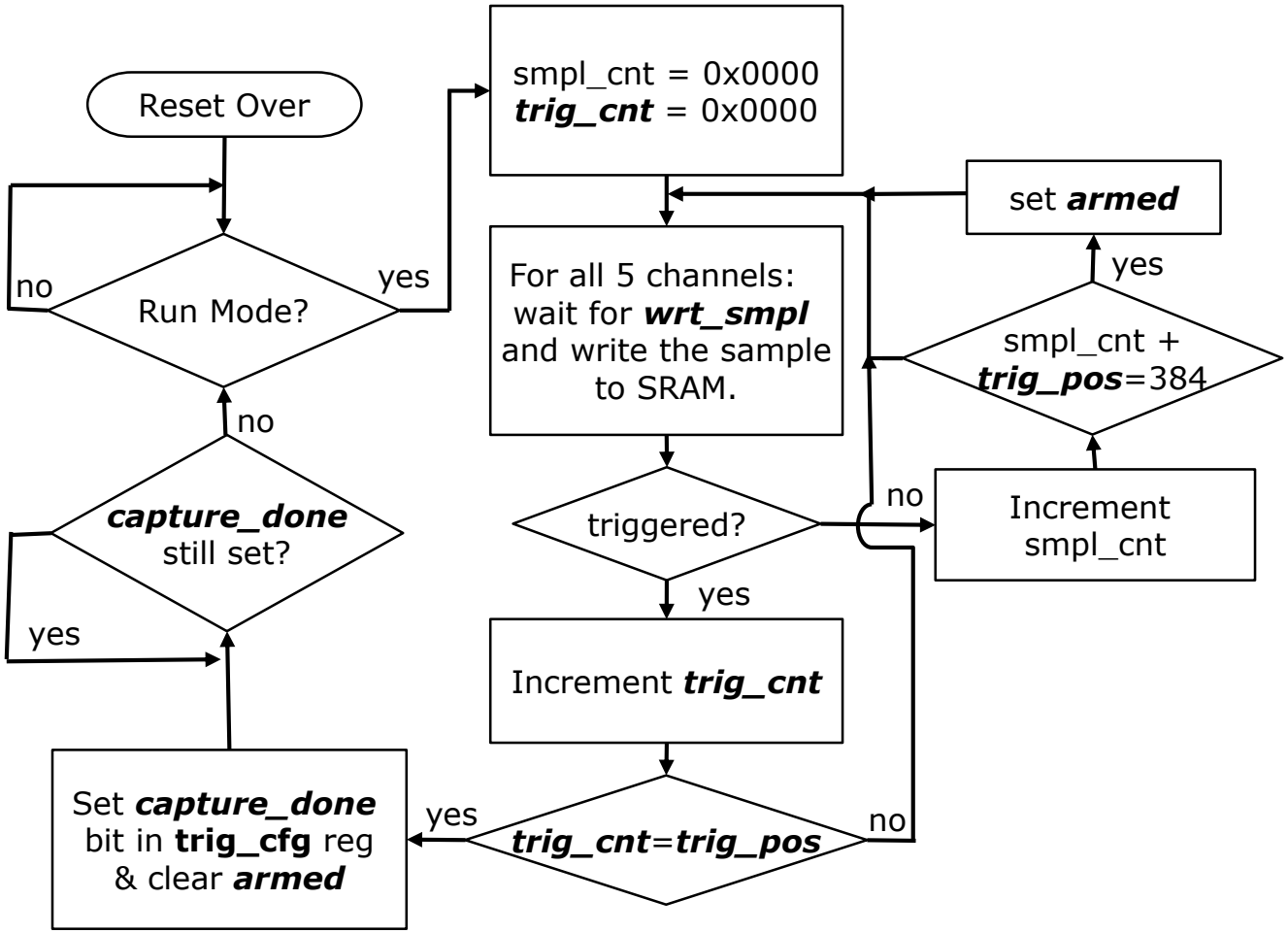
Digital Core Functionality (Channel Capture)

As long as the trigger mode is set in a run state (bit 4 of **TrigCfg**), and the capture has not completed, data from the 5 channels should be captured and stored in SRAM.

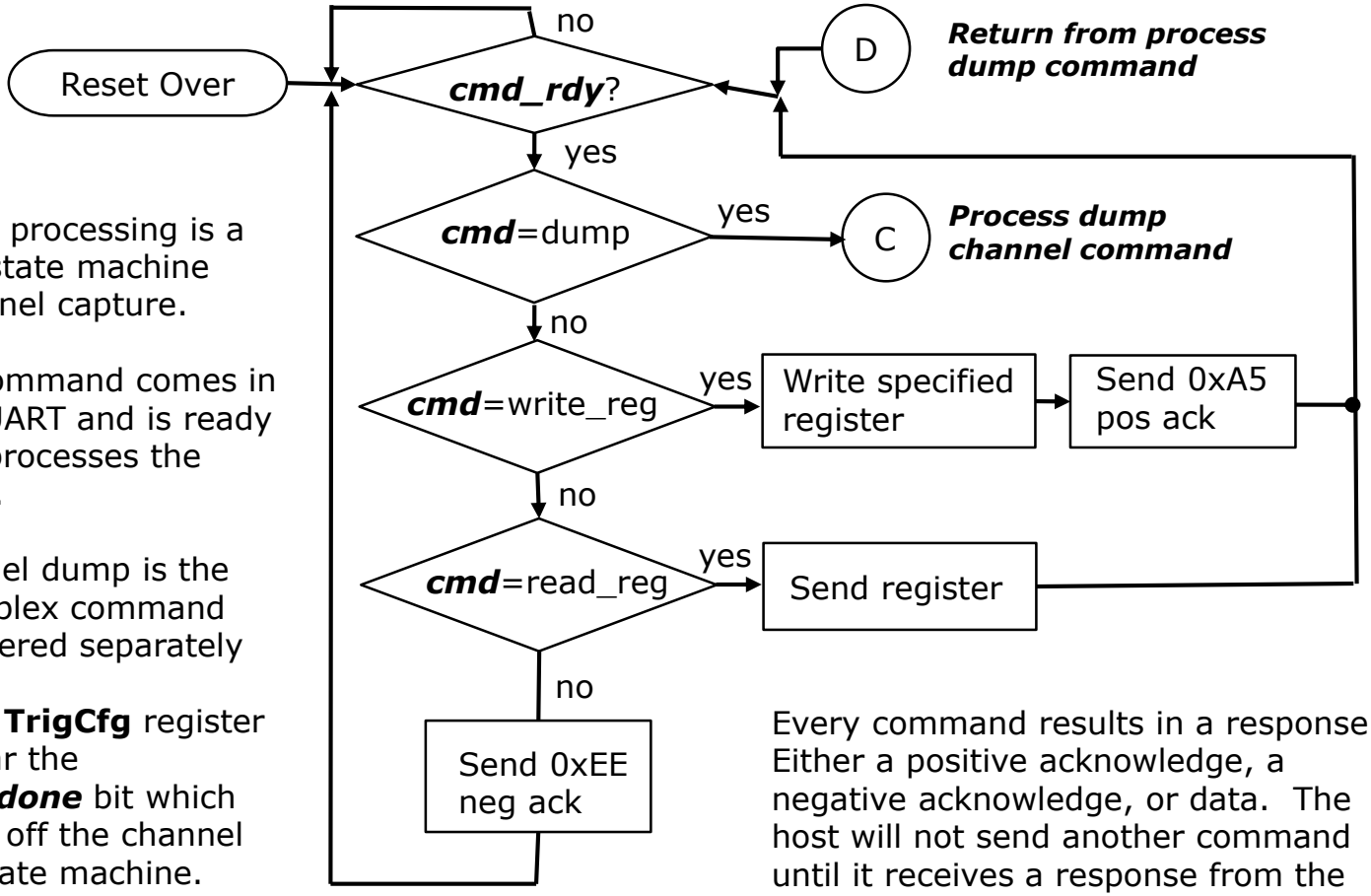
Capture continues until a configurable number of samples (***trig_pos***) after the trigger have been captured.

Once ***trig_pos*** number of samples after the trigger have been captured data acquisition ends and the "***capture_done***" bit is set.

"***capture_done***" is cleared by the command processing SM. The host software would read the **TrigCfg** register and clear the ***capture_done*** bit (bit 5) and then write the **TrigCfg** register.



Digital Core Functionality (command processing)



Command processing is a separate state machine from channel capture.

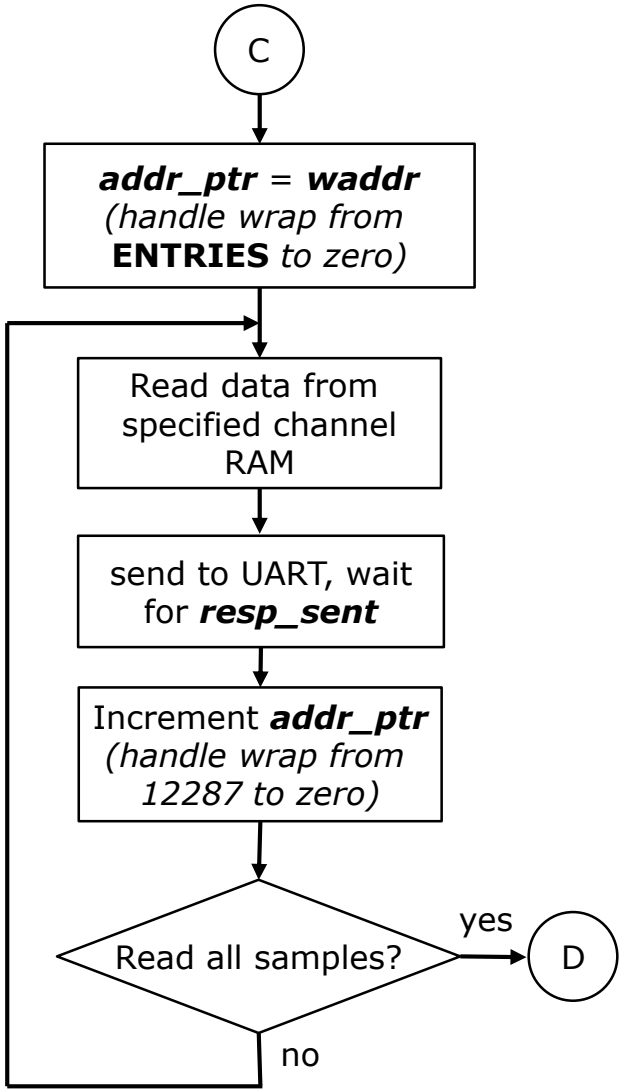
When a command comes in over the UART and is ready it simply processes the command.

The channel dump is the most complex command and is covered separately

A write to **TrigCfg** register might clear the ***capture_done*** bit which could kick off the channel capture state machine.

Every command results in a response. Either a positive acknowledge, a negative acknowledge, or data. The host will not send another command until it receives a response from the previous command.

Digital Core Functionality



- Dumping Channel Data

When the capture of the channel data completed ***waddr*** was pointing to the next location to write. If the queue is full this is also the oldest data.

The 384 (or 12288) entries of the channel RAMs can be thought of as a circular queue.

When the ***addr_ptr*** wraps around and comes back to ***waddr*** we are done dumping the data to the host.

Remember the RAM depth is parametizable. For both simulation and DE0 mapping it is not a nice power of 2, so you have to watch the wrap around case.

Register Set

Addr	Name	Rst Val	Description:
0x00	TrigCfg	6'h03	Covered in greater detail on a previous slide about protocol triggering. Bit5 → capture_done, Bit4 → Run, Bit3 → SPI edge, Bit2 → SPI length, Bit1 → SPI_disable, Bit0 → UART disable
0x01	CH1TrigCfg	5'h01	Bit4 → pos edge, Bit3 → neg edge, Bit2 → High Level, Bit1 → Low Level, Bit0 → Not part of trigger
0x02	CH2TrigCfg	5'h01	Bit4 → pos edge, Bit3 → neg edge, Bit2 → High Level, Bit1 → Low Level, Bit0 → Not part of trigger
0x03	CH3TrigCfg	5'h01	Bit4 → pos edge, Bit3 → neg edge, Bit2 → High Level, Bit1 → Low Level, Bit0 → Not part of trigger
0x04	CH4TrigCfg	5'h01	Bit4 → pos edge, Bit3 → neg edge, Bit2 → High Level, Bit1 → Low Level, Bit0 → Not part of trigger
0x05	CH5TrigCfg	5'h01	Bit4 → pos edge, Bit3 → neg edge, Bit2 → High Level, Bit1 → Low Level, Bit0 → Not part of trigger
0x06	decimator	4'h0	Determines sample rate. Every $2^{decimator}$ sample are taken. If decimator is zero we sample at 400MHz
0x07	VIH	8'hAA	Sets VIH level. 0x00 ➔ 0V, 0xFF ➔ 3.3V
0x08	VIL	8'h55	Sets VIL level. 0x00 ➔ 0V, 0xFF ➔ 3.3V
0x09	matchH	8'h00	When using 16-bit SPI protocol matching this byte specifies the upper byte of the match for trigger
0x0A	matchL	8'h00	Specifies the low byte for match when using protocol based triggering (SPI or UART).
0x0B	maskH	8'h00	Mask bits to mask off the corresponding match bits. A set mask indicates a don't care for compare
0x0C	maskL	8'h00	Mask bits to mask off the corresponding match bits. A set mask indicates a don't care for compare
0x0D	baud_cntH	8'h06	When using UART based protocol triggering this byte specifies the upper byte to use for baud counter
0x0E	baud_cntL	8'hC8	When using UART based protocol triggering this byte specifies the lower byte to use for baud counter
0x0F	trig_posH	8'h00	Specifies the upper byte of trig_pos (number of samples to capture after the trigger event occurs)
0x10	trig_posL	8'h01	Specifies the lower byte of trig_pos (number of samples to capture after the trigger event occurs)

Command Set (commands received over UART interface)

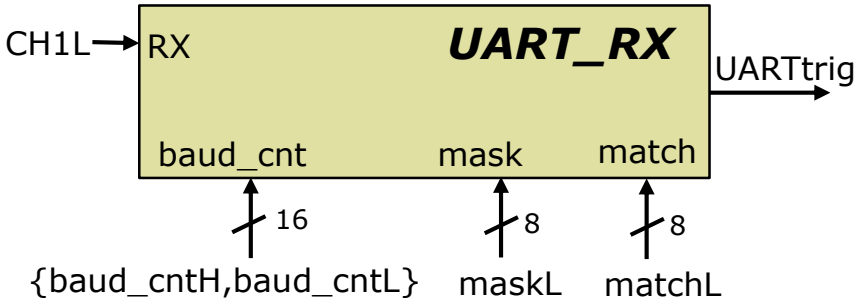
Command Encoding: All commands are 16-bits. The upper 2-bits [15:14] encodes the opcode.

Bits[15:14]	Bits[13:8]	Bits[7:0]	Description:
2'b00	6'baaaaaa	8'hXX	Read register. The lower 6-bits of the upper byte specify the address to read. The specified address is read and sent via UART.
2'b01	6'baaaaaa	8'hDD	Write register. The lower 6-bits of the upper byte specify the address to write. The data to write is contained in bits [7:0]. Once the register is written the core should respond with a positive acknowledge (send a 0xA5 via UART).
2'b10	6'b000ccc	8'hXX	Dump channel RAM. The channel to dump is specified in the lower 3-bits of the upper byte. The channel data is read and dumped via UART. ccc = 1 for CH1, ccc = 101 for CH5.
2'b11	6'bxxxxxx	8'hXX	Reserved for future use.

The upper 2-bits of the upper byte of the command specifies the command to perform (wrt_reg, rd_reg, dmp_chnnl). For wrt_reg and rd_reg the lower 6-bits of the upper byte specify the register address. If the command is a write then the lower byte of the command is the data to write.

UART Module for Protocol Triggering

- You have already produced a UART receiver. Start with that and modify it.
- The main modification is that the baud rate (how often you shift) will now be based on an input (***baud_cnt[15:0]***) instead of a fixed number of clocks.
- ***baud_cnt[15:0]*** specifies the number of system clocks (100MHz clock) that the baud period is. For example to achieve a baud rate of 115,200 ***baud_cnt*** would be set to 16'h0364.
- This unit will no longer output the 8-bit data of the byte received, but will rather assert ***UARTtrig*** if the byte received matches ***match[7:0]. NOTE: mask[7:0]*** specifies don't cares for the matching function. If a bit of ***mask*** is set then the corresponding bit of ***match*** is treated as don't care.
- ***UARTtrig*** should go high for 1 clock cycle at the end of a reception if the data received matches ***match[7:0].*** (qualified with don't cares of mask)
- ***CH1L*** feeds the *UART_TX* module directly. This is the output of the low comparator for CH1, so it is asynch to the system clock domain.

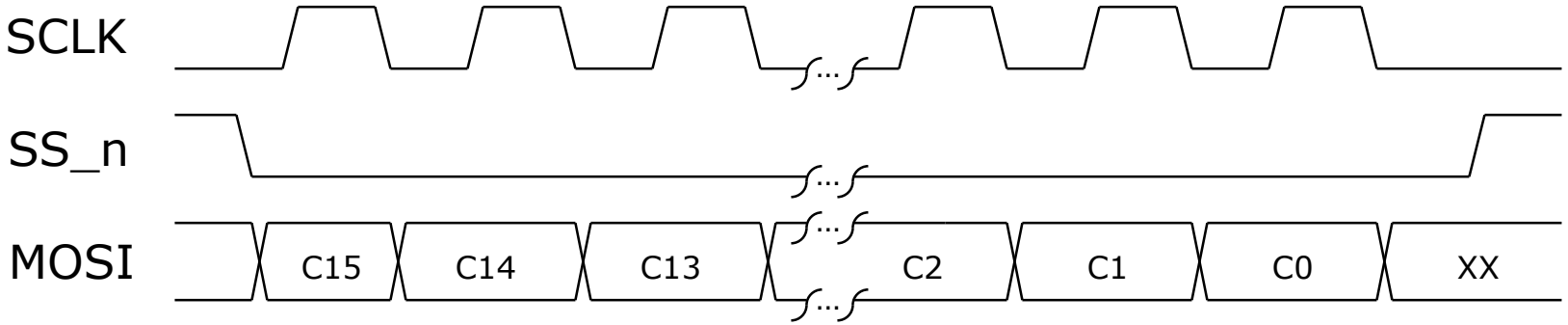


Signal:	Dir:	Description:
clk	in	100MHz system clock
rst_n	in	Active low system reset
RX	in	Serial data in line. Comes direct from VIL comparator of CH1
baud_cnt[15:0]	in	Specifies the buad rate of the UART in number of system clks
match[7:0]	in	Specifies the data the UART_RX is looking to match
mask[7:0]	in	Used to mask off bits of ***match*** to a don't care for comparison
UARTtrig	out	Asserted for 1 clock cycle at end of a reception if received data matches ***match[7:0]***

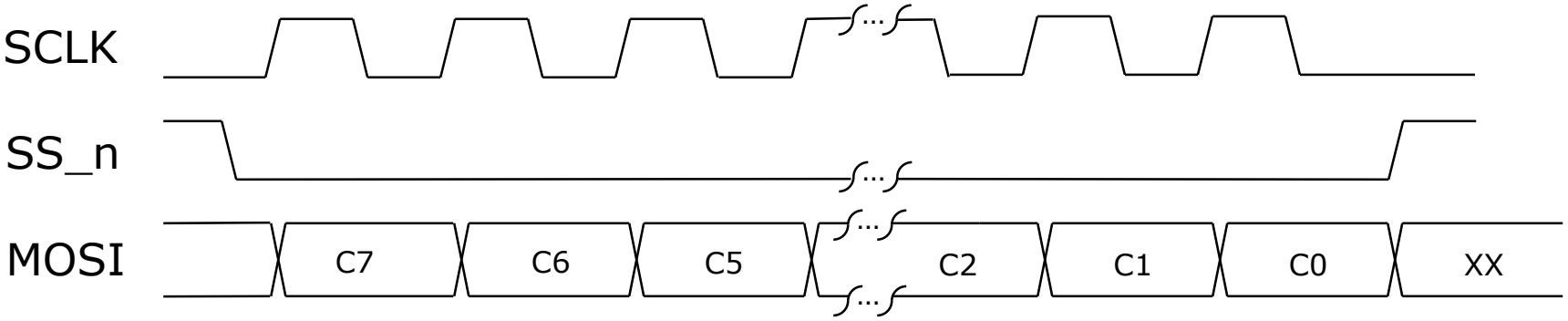
What is SPI?

- Simple uni-directional serial interface (Motorola long long ago)
  - **S**erial **P**eripheral **I**nterconnect (very popular physical interface)
  - 4-wires for full duplex
    - ✓ MOSI (Master Out Slave In) (digital core will drive this to AFE)
    - ✓ MISO (Master In Slave Out) (not used in connection to AFE digital pots, only EEP)
    - ✓ SCLK (Serial Clock)
    - ✓ SS_n (Active low Slave Select) (Our system has 4 individual slave selects to address the 4 dual potentiometers, and a fifth to address the EEPROM)
  - There are many different variants
    - ✓ MOSI Sampled (shifted) on clock low vs clock high
    - ✓ SCLK normally high vs normally low
    - ✓ Widths of packets can vary from application to applications
    - ✓ Really is a very loose standard (barely a standard at all)
  - We will stick with SCLK normally low, but have configuration for the rest
    - ✓ MOSI sampled on SCLK rise if ***edg*** is high, SCLK fall if ***edg*** is low.
    - ✓ 8-bit packets if ***len8_16*** is high, and 16-bit packets if ***len8_16*** is low.

SPI Packets



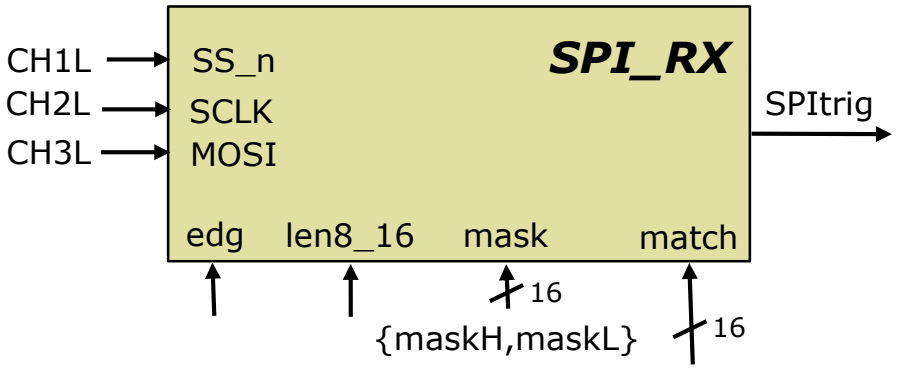
Shown above is a 16-bit SPI packet where the master is changing (shifting) MOSI on the falling edge of SCLK. This is a case where one would probably choose to have ***edg*** high so MOSI was sampled on SCLK rise.



Shown above is an 8-bit SPI packet where the master is changing (shifting) MOSI on the rising edge of SCLK. This is a case where one would probably choose to have ***edg*** low so MOSI was sampled on SCLK fall.

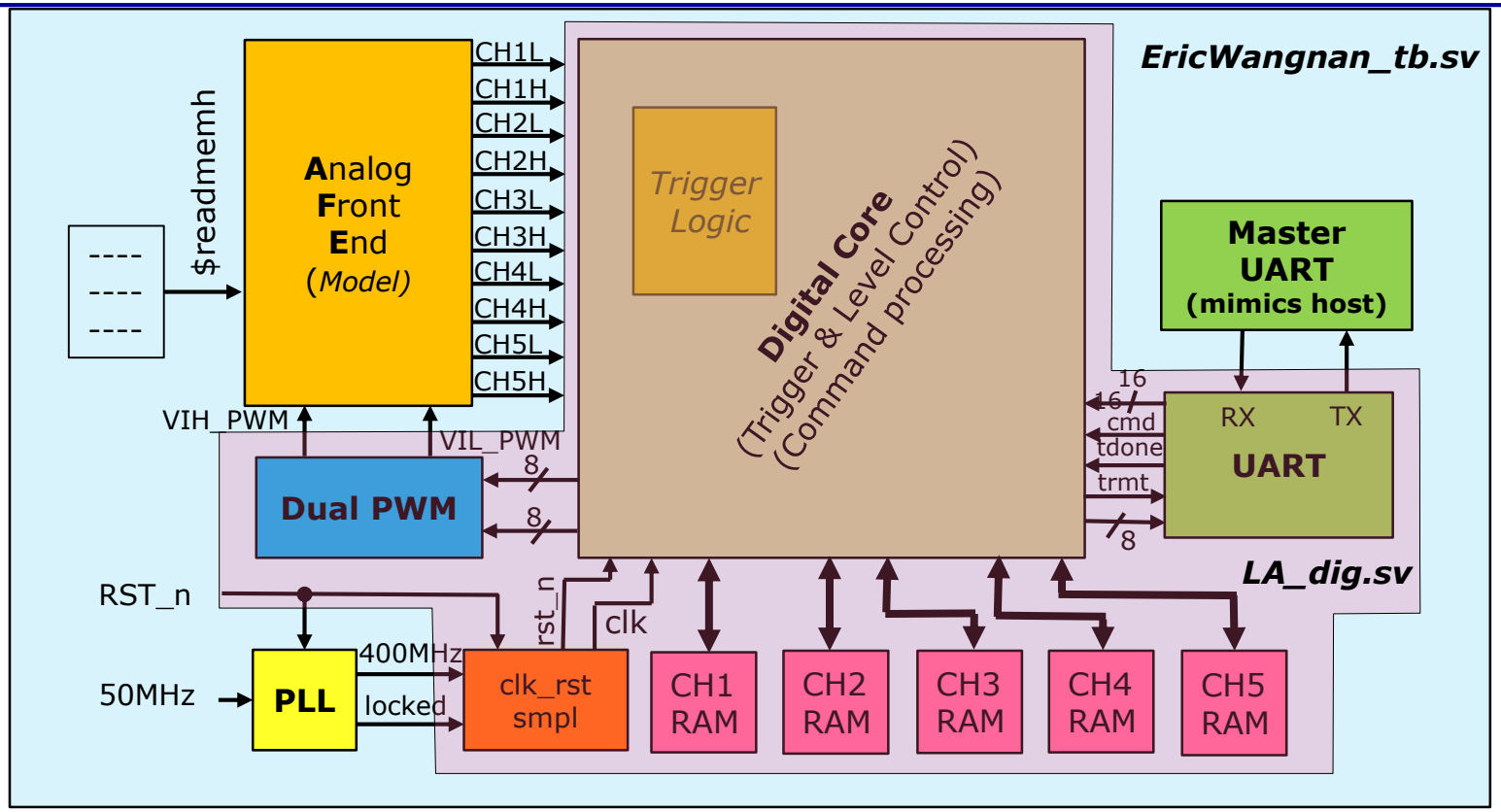
SPI Unit for Protocol Triggering

- Make a SPI_RX module. It should not operate with ***SCLK*** as a clock, but rather treat ***SCLK*** as a signal it samples using the main 100MHz system ***clk***.
- The SPI module should continue to sample and shift ***MOSI*** (on the specified edge of ***SCLK*** (specified by ***edg***)) until ***SS_n*** returns high.
- When ***SS_n*** returns high the SPI transaction can be considered complete and the contents in the receive shift register can be compared to ***match[15:0]***.
- if ***len8_16*** is high then comparison is only 8-bits, and the lower 8-bits of the receive shift register are compared against ***match[7:0]***.
- If ***edg*** is high then the receive shift register should shift (sample ***MOSI***) on the rise of ***SCLK***. If ***edg*** is low then the shift is on the fall of ***SCLK***.
- Mask is used to mask off the bits of match and treat them as a don't care in the comparison. High bits in mask represent bits of match that should be treated as don't care.



Signal:	Dir:	Description:
clk, rst_n	in	100MHz system clock and reset
SS_n, SCLK, MOSI	in	SPI protocol signals. Coming from VIL comparators
edg	in	When high the receive shift register should shift on SCLK rise.
len8_16	in	When high we are doing an 8-bit comparison to ***match[7:0]***
mask[15:0]	in	Used to mask off bits of ***match*** to a don't care for comparison
match[15:0]	in	Data unit is looking to match for a trigger
SPItrig	out	Asserted for 1 clock cycle at end of a reception if received data matches ***match[7:0]***

Required Hierarchy & Interface



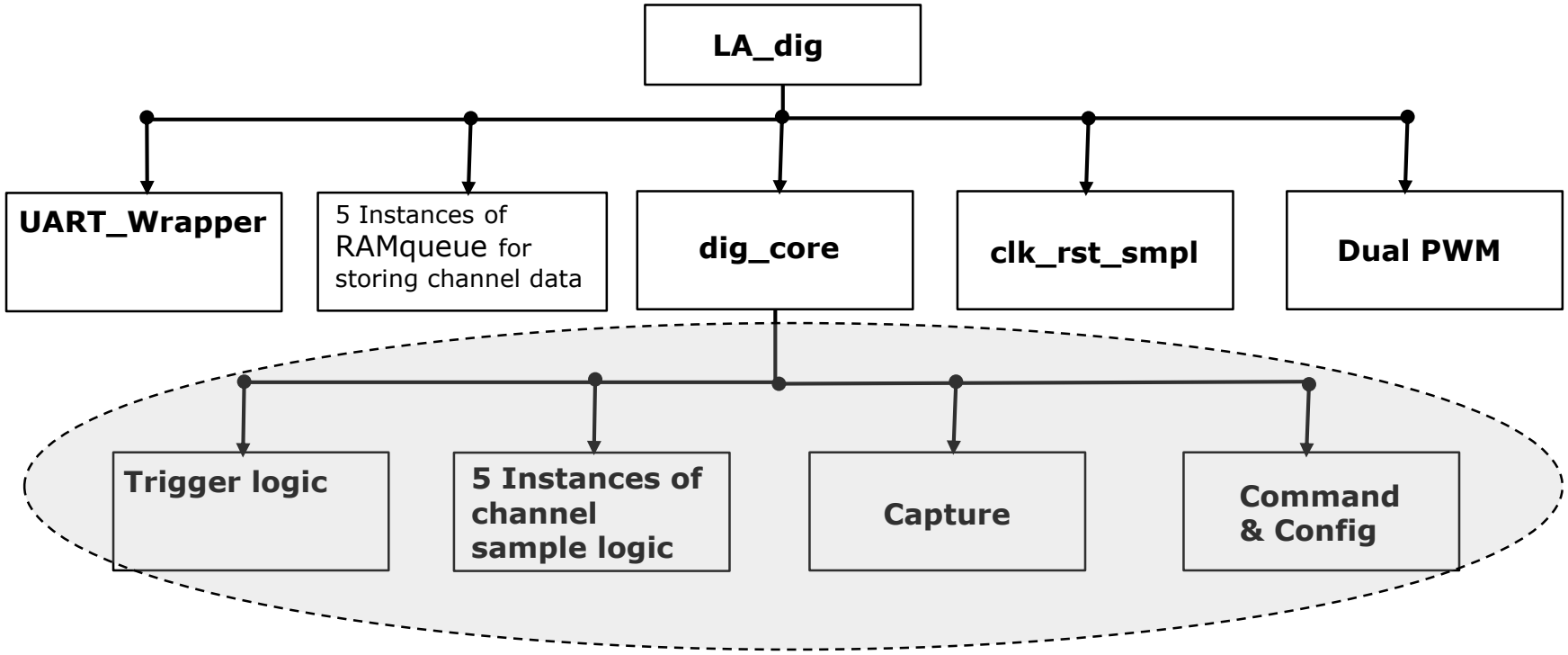
Your design will be placed in an "EricWangnan" testbench to validate its functionality. It must have a block called **LA_dig.sv** which is top level of what will be the synthesized DUT.
The interface of LA_dig.sv **must match exactly** to our specified **LA_dig.sv** interface
Please download LA_dig.sv (interface skeleton) from the class webpage.

The hierarchy/partitioning of your design below LA_dig is up to your team.
The hierarchy of your testbench above LA_dig.v is up to your team.

LA_dig Interface

Signal Name:	Dir:	Description:
clk400MHz	in	Clock input (400MHz from 8xPLL). Used for sampling channels, and divided down to form main system clock (clk) which is 100MHz
RST_n	In	Unsynchronized reset (from push button) goes through ***clk_rst_smpl*** block to get synchronized and form **rst_n** (global reset to all other blocks).
locked	In	From PLL. Indicates that PLL is locked onto the 50MHz ref clock.
VIH_PWM, VIL_PWM	Out	PWM signals that form the analog thresholds for determining VIH and VIL.
CHxL	In	Channel x (where x is 1..5) logic low input. If this input is low then CHx is below the VIL level and is considered logic low.
CHxH	In	Channel x (where x is 1..5) logic high input. If this input is high then CHx is above the VIH level and is considered logic high.
RX	In	Serial (UART) input. Command from host PC would come through this signal.
TX	Out	Serial (UART) output. Responses to host PC come from this signal.

Figure out your partitioning of ***dig_core***



Partitioning below ***dig_core*** is up to you guys. I partitioned into these 4 units. Does that partioning make sense? Trigger logic could have been part of the capture module. There is some interaction between Capture and Command/Config that is "interesting". Perhaps a different portioning could work better?

Digital Core Interface (dig_core.v)

Signal Name:	Dir:	Description:
clk, rst_n	in	Clock input (100MHz), and active low async reset.
smpl_clk	in	Clock that drives the channel sample logic. Varies depending on value of decimator[3:0]. Can be as fast as 400MHz if decimator is zero.
wrt_smpl	in	Asserted when a sample is ready and can be written to SRAM.
decimator[3:0]	Out	Determines sample rate 400MHz/2^decimator. Goes to clk_rst_smpl block so smpl_clk and wrt_smpl can be generated correctly.
VIH[7:0], VIL[7:0]	out	Settings that establish the VIL, and VIH thresholds. Based on 3.3V full scale, so a setting of 0x40 would result in 3.3V*0x40/0x100 = 0.825V threshold
CHxL	In	Channel x (where x is 1..5) logic low input. If this input is low then CHx is below the VIL level and is considered logic low.
CHxH	In	Channel x (where x is 1..5) logic high input. If this input is high then CHx is above the VIH level and is considered logic high.
cmd[15:0]	In	16-bit command from host PC
cmd_rdy	In	Indicates command from host PC is ready
clr_cmd_rdy	Out	Asserted by dig_core to indicate we have “digested” the last command
resp[7:0]	out	Response to the host PC. Every command issues a response
send_resp	Out	Asserted by dig core when it has reponse to send.
resp_sent	In	From the UART comm module, indicates the response has been sent.

Continued next page

Digital Core Interface ***continued*** (dig_core.v)

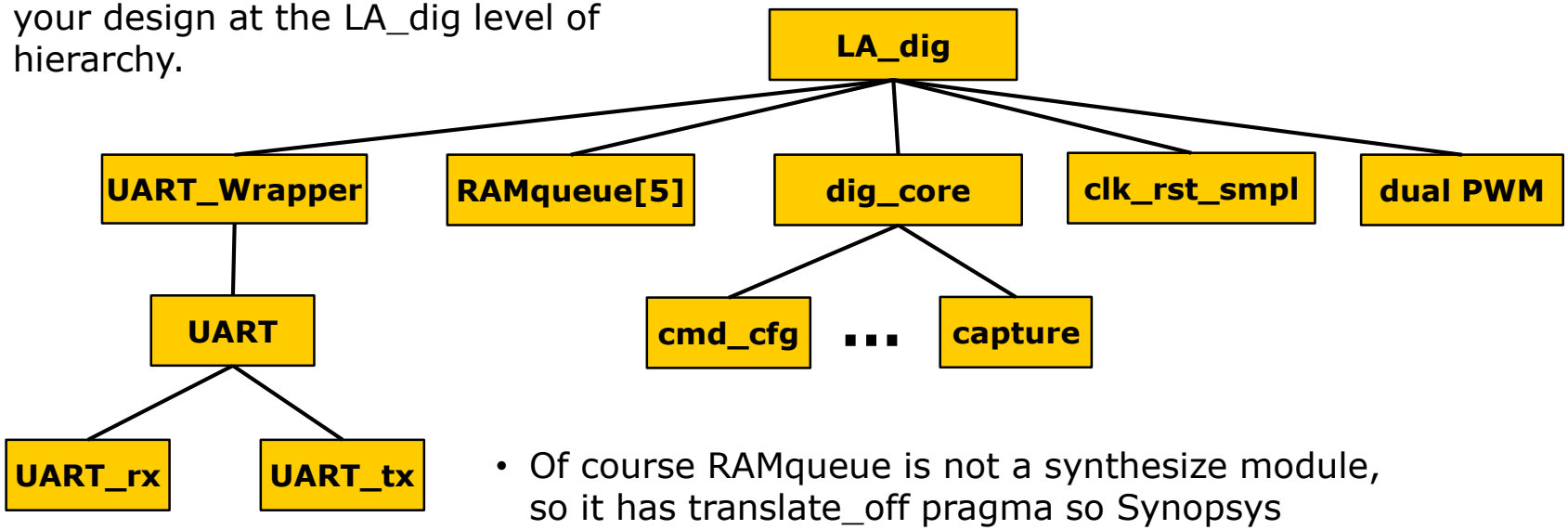
Signal Name:	**Dir:**	**Description:**
we	Out	Write enable to the RAM queues. All five queues can share the same **we**
waddr[LOG2-1:0]	out	Address of location to write to in RAM queue. All five queues can share **waddr**
raddr[LOG2-1:0]	out	Address of location in RAM queue to read. All five queues can share **raddr**
wdataCHx[7:0]	out	Data sample to write to RAM queue for channel x (where x is: 1..5)
rdataCHx[7:0]	In	Data sample read from RAM queue for channel x (where x is: 1..5)

Provided Modules & Files: (available on website under: Project as Collateral.zip)

File Name:	Description:
LA_dig_tb.v	**Optional** testbench template file.
LA_dig.v	**Requried** interface skeleton verilog file. **Copy this** you can change internals if you wish, but interface signals must remain as is.
dig_core.v	Optional digital core interface skeleton. Might be handy as starting point
pll8x.v	Model of PLL that multiplies 50MHz ref clk to produce 400MHz clock
AFE.v	Models the Analog Front End. Reads data from CHxmem.txt where x is: 1..5.
CH[X]mem.txt	Analog representation of the signal waveforms that the logic analyzer is connected too. [X] = 1,2,3,4,5. These files are read by AFE.v
gen_channel.pl	Perl program you can modify to generate more CH[X]mem.txt files. This will help you validate your design better.
RAMqueue.v	Model of the RAMs used for storing raw channel data. You made these earlier, but I need to ensure we are all operating with the same model so when we map to the DE0-Nano it works out.
clk_rst_smpl.sv	The tricky clock division logic. See slide 13 for details.

Synthesis:

- You have to be able to synthesize your design at the LA_dig level of hierarchy.



- Of course RAMqueue is not a synthesize module, so it has translate_off pragma so Synopsys ignores it.

- Your synthesis script should write out a gate level netlist of LA_dig (LA_dig.vg).
- You should be able to demonstrate at least one of your tests running on this post synthesis netlist successfully.
- Timing (1.0GHz for clk400MHz and 250MHz for clk) is pretty easy to make. Your main objective is to minimize area.

Synthesis Constraints:

Contraint:	Value:
Clock frequency	1.0GHz (for clk400MHz) (yes, I know the project spec speaks of 400MHz, but that is for the FPGA mapped version. The standard cell mapped version needs to hit 1.0GHz (1ns period clock)
Input delay	0.25ns after smplclk fall for all CH* inputs 0.25ns after clk400MHz rise for RST_n and locked 0.25ns after rise of clk for RX.
Output delay	0.5ns prior to next clk rise for all outputs
Drive strength of inputs	Equivalent to a NAND2X1_RVT gate from our library
Output load	0.05pF on all outputs
Wireload model	For a block size of 16000 square microns
Max transition time	0.15ns
Clock uncertainty	0.15ns (this applies to “clk” domain only (100MHz) smplclk and clk400MHz are small and have no skew)

NOTE: Area should be taken after all hierarchy in the design has been smashed. Synopsys will realize that the 5 instances of RAMqueue are black boxes, so smashing will still maintain those as separate blocks, and your resulting gate level netlist will have placeholders for these RAMs.